Primjena SOA uzoraka u razvoju web aplikacija

Grđan, Zoran

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Zoran Grđan

Primjena SOA uzoraka u razvoju web aplikacija

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Zoran Grđan

Matični broj: 46350/17-R

Studij: Informacijsko i programsko inženjerstvo

Primjena SOA uzoraka u razvoju web aplikacija

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Kermek Dragutin

Varaždin, veljača 2021.

Zoran Grđan

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Rad se sastoji od dva dijela, prvi dio teorijski opisuje servise i servisne uzorke, dok se u drugom dijelu rada ti uzorci primjenjuju kod kreiranja aplikacije. U ovom radu govoriti će se o tome kakve sve vrste servisa postoje. Za opisane vrste servisa biti će objašnjene njihove prednosti i mane te u kojim trenucima je bolje koristiti kojeg. Nakon prolaska kroz osnove o servisima biti će objašnjena servisno orijentirana arhitektura, na koji način se ona koristi u poslovnom smislu i kako se primjenjuje u fazi programiranja. U radu će biti objašnjeno nekoliko uzoraka i principa od kojih će neki biti implementirani u programskom dijelu rada. U programskom dijelu rada biti će napravljena aplikacija pod nazivom „Robna Kontrola“ koja će biti izrađena u skladu sa servisno orijentiranim principima.

Ključne riječi: soap;rest;servisi;soa;arhitektura

Sadržaj

1.Uvod

Servisno orijentirana arhitektura (SOA) je izraz koji se već godinama pojavljuje u poslovnom svijetu, a pojam SOA proizlazi iz termina kojeg je koristio IBM tijekom strateškog zaokreta proizvodnje hardvera i softvera prema nuđenju IT usluga. Danas je servisno orijentirana arhitektura jedan od važnih aspekata koje je potrebno uzeti u obzir kada se planira kreiranje neke poslovne aplikacije jer je nužno da se poslovni i tehnološki aspekti usklade kako bi mogli zajedno evoluirati i razvijati se. SOA omogućava maksimalnu fleksibilnost i proširivost aplikacije koju podržava, a fleksibilnost i proširivost arhitekture utječu na znatno pojednostavljenje životnog ciklusa poslovnih aplikativnih sustava za sve njegove sudionike. Pojednostavljivanje životnog ciklusa odražava se tako da se mogu primijeniti brze promjene u poslovnim zahtjevima. SOA predstavlja stvarnost, a ne neki trik izmišljen na brzinu. Riječ je o procesu kojem je potrebno puno vremena i ozbiljnog planiranja kako bi se uspješno implementirao, iako sama prilagodba i korištenje servisno orijentirane arhitekture nije garancija da će funkcionirati unutar organizacije. Da bi organizacija mogla funkcionirati na taj način nužno je da se organizacijom kvalitetno upravlja. Sama SOA nije nešto što se može kupiti kao gotov proizvod, to je prije svega put sveukupne transformacije poslovanja poduzeća koji je zasnovan dijelom na optimizaciji poslovnih procesa, a dijelom na IT resursima koji ga podržavaju.

Ovim radom želi se istražiti skup servisnih uzoraka koji se mogu primijeniti kod raznih zadataka i problema s kojima se susreću poduzeća i razvojni inženjeri. Kroz rad će biti spomenuto niz tehnologija koje se mogu i koje će biti korištene u radu.

Rad je koncipiran u dva glavna dijela. U prvom dijelu teorijski se obrađuje tema servisno orijentirane arhitekture zajedno s nizom drugih tema koje su blisko vezane uz SOA arhitekturu. Definirani su različiti uzorci dizajna od kojih će se neki upotrijebiti u drugom dijelu rada. Drugi dio rada posvećen je izradi programskog zadatka. Kroz taj dio rada prikazat će se niz uzoraka koji se mogu koristiti ovisno o problemu s kojim se razvojni inženjeri i arhitekti sustava mogu susresti. Uz sami primjer, objasnit će se i zašto se u pojedinom problemu koristi određeni uzorak te postoji li možda neki drugi alternativni uzorak koji je mogao biti korišten za određeni problem.

2. Metode i tehnike rada

Tehnologije koje su korištene u izradi programskog dijela aplikacije su Angular[1], za izradu korisničkog dijela aplikacije i SpringBoot[2] za izradu servisa koji se upotrebljavaju za rad s podacima koji se prikazuju na korisničkoj strani aplikacije. Za bazu je odabran MySQL[3] bazni servis koji je besplatan, a za server je odabran Apache[4] koji je pokretan za omogućavanje pristupa bazi.

Angular je okvir (eng. *Framework*) korišten za razvoj aplikacija razvijen 2009. godine od strane Google-a. Aplikacija izrađena u Angularu je klijentska aplikacija koja se sastoji od HTML i TypeScript dijela. Sam Angular je napisan u TypeScript-u. Implementacija glavnih i opcionalnih funkcionalnosti Angulara nalazi se u bibliotekama koje se uvoze ovisno o tome koja je funkcionalnost potrebna za izradu aplikacije. Sve aplikacije koje su napisane u Angularu sadrže module kojih može biti jedan ili više.[16, str 1] Aplikacija koja je rađena sastoji se od jednog modula i nekolicine komponenti. Komponentama se definiraju pogledi koje vide korisnici.

Za poslužiteljski dio aplikacije korištena je „SpringBoot“ platforma koja je besplatni okvir baziran na Javi te se koristi za izradu mikro servisa. Verzija koja se koristila je „2.3.2.RELEASE“.

Mikro servis je arhitektura koja omogućava programerima da neovisno razvijaju i implementiraju usluge. Svaka pokrenuta usluga ima svoj proces i time se postiže lagani model za podršku poslovnim aplikacijama. Mikro servisi nude neke određene pogodnosti [17]:

- jednostavno postavljanje,
- jednostavna skalabilnost,
- kompatibilan je s kontejnerima,
- potrebna je minimalna konfiguracija,
- potrebno manje vrijeme izrade.

Uzevši u obzir sve pogodnosti koje nudi SpringBoot, poslužiteljski dio aplikacije napisan je pomoću tog okvira. Osim SpringBoot-a, korišten je i Gradle[5] verzije 6.5.1 za automatizaciju izrade.

Da bi se u potpunosti mogla realizirati izgradnja aplikacije potrebno je odrediti u čemu će biti podržana baza podataka. Za samu realizaciju baze odabran je MySQL. MySQL je sustav

[1]https://angular.io/guide/architecture
[2]https://spring.io/projects/spring-boot
[3]https://www.mysql.com/
[4]https://httpd.apache.org/
[5]https://gradle.org/releases/

za upravljanje relacijskim bazama podataka koji nudi razne opcije za strukturiranje upita na jeziku SQL koji se koristi za dodavanje, uklanjanje i izmjenu podataka u bazi podataka. Temelji se na standardnim naredbama poput DELETE, SELECT, INSERT i UPDATE.

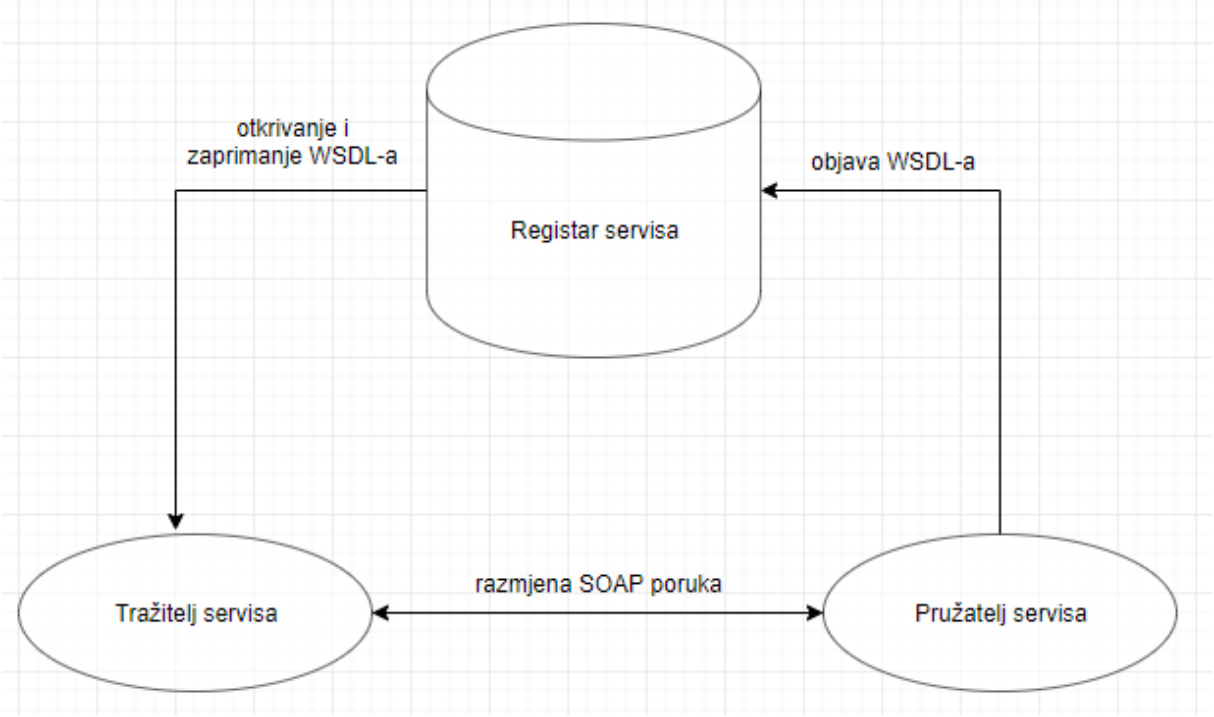
3. Postanak servisno orijentirane arhitekture

XML (eng. *Extensible Markup Language*) je jezik za označavanje podataka koji postoji još od kasnih 60-ih godina, a razvio se iz popularnog SMGL-a (eng. *Standard Generalized Markup Language*). Sami XML stekao je popularnost tijekom poslovnog pokreta kasnih 90-ih gdje su skriptni jezici na strani poslužitelja poslovanje putem interneta učinili mogućim i održivim. Kroz korištenje XML-a razvojni inženjeri mogli su u bilo kojem dijelu informacije koja se prenosi preko internetskih protokola pridodati značenje i kontekst. XML nije korišten samo za predstavljanje podataka na standardiziran način, već je i sam jezik poslužio kao osnova za niz dodatnih specifikacija. Pomoću XML-a autorizirani su XSD (eng. *XML Schema Definition Language*) i XSL (eng. *eXtensible Stylesheet Language*) gdje je XSD preporuka koja određuje kako formalno opisati elemente u XML dokumentu, a XSL standardni način opisivanja kako transformirati strukturu XML dokumenta u XML dokument različite arhitekture. Navedene tehnologije su postale ključni dijelovi osnovnog skupa XML tehnologija. Predstavljanje arhitekture XML podataka predstavlja temeljni sloj servisno orijentirane arhitekture. Unutar njega, XML uspostavlja format i strukturu poruka koje putuju kroz servise. XSD sheme čuvaju integritet i valjanost podataka unutar poruka dok se XSLT koristi za omogućavanje komunikacije između različitih reprezentacija podataka putem mapiranja. [1, str 164]

U 2000. godini W3C (eng. *World Wide Web Consortium*) je zaprimio specifikaciju za komunikacijski protokol koji je neovisan o platformi, a baziran je na XML-u pod nazivom SOAP (eng. *Simple Object Access Protocol*). Ovaj zahtjev koji je došao na početku bio je zamišljen da se parametarski podaci koji se prenose između komponenata serijski prevedu u XML, prenesu i zatim deserijaliziraju natrag u njihov izvorni format. Uskoro su mnoge korporacije i proizvođači softvera počeli uviđati sve veći potencijal za unaprjeđenje e-poslovanja oslanjajući se na besplatni internetski komunikacijski okvir. To je dovelo do ideje o stvaranju čiste, internetski distribuirane tehnologije koja bi mogla utjecati na koncept standardiziranog komunikacijskog okvira kako bi se premostila ogromna razlika koja je postojala između organizacija i unutar njih. Ovaj koncept naposljetku je nazvan web usluga ili dobro poznati web servis. Najvažniji dio web servisa je javno sučelje koje servisu dodjeljuje identitet i omogućuje njegovo pozivanje. Uslijed razvoja web servisa pojavio se jezik opisa web servisa WSDL (eng. *Web Service Description Language*) kojeg je W3C prvi puta zaprimio 2001. godine i od tada ga je nastavio koristiti. Na samom kraju razvoja standarda web usluga prve generacije nalazi se UDDI (eng. *Universal Description, Discovery and Integration)* specifikacija koja omogućuje stvaranje standardiziranih opisnih registara servisa

i izvan granica organizacije. UDDI pruža mogućnost da se web servisi registriraju na središnjoj lokaciji odakle ih mogu otkriti tražitelji servisa.[1, str 165-166]

SOA je klasificirana na različite načine, često ovisno o tehnologiji implementacije koja se koristi za kreiranje servisa. Erl [1, str 168] navodi da je SOA arhitektura modelirana oko tri osnovne komponente: registar servisa, tražitelj servisa i pružatelj servisa.



Slika 1:Arhitektura web servisa

(Izvor: izrada autora prema [1, str 168])

Iz prethodne slike može se vidjeti da se web servis sastoji od tri osnovne komponente. U ovom slučaju WSDL služi kao dokument koji opisuje web servis. Registar servisa pruža standardizirani format za registriranje servisa koje se nude, dok SOAP pruža format za razmjenu poruka između tražitelja i pružatelja servisa.

4. Servisno orijentirana arhitektura (SOA)

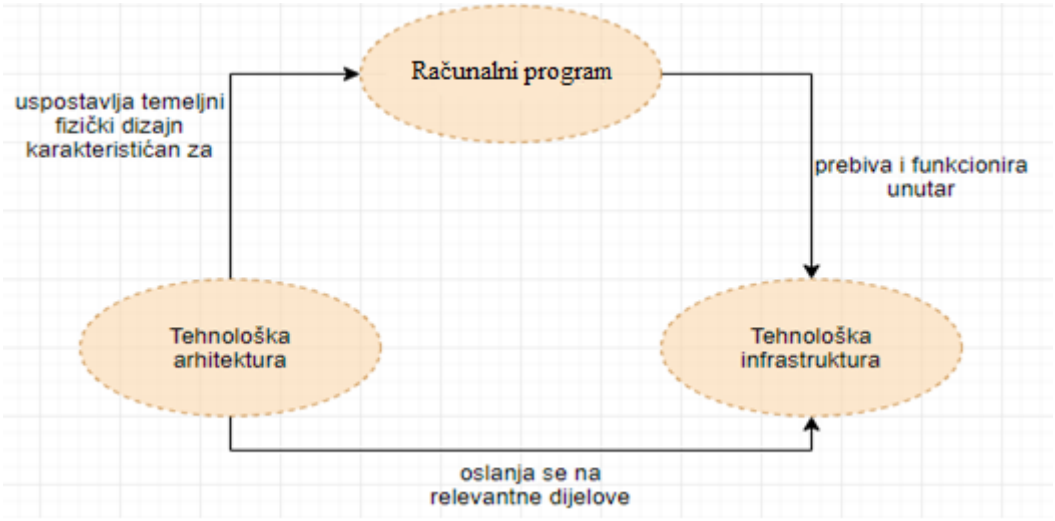
„Servisno orijentirana arhitektura predstavlja arhitekturni model kojemu je cilj poboljšati agilnost i isplativost poduzeća uz istovremeno smanjivanje opterećenja IT-a na cjelokupnu organizaciju. To se postiže pozicioniranjem servisa tako da oni služe kao primarna sredstva preko kojih je predstavljena logika za rješavanje problema. SOA podržava servisnu orijentaciju u realizaciji strateških ciljeva poduzeća povezanih sa servisno orijentiranim računalstvom.“ [2, str 37]

Erl [2, str 37] navodi da se SOA implementacija može sastojati od kombinacija tehnologija, proizvoda, sučelja za programiranje API (eng. *Application Programming Interfaces*), podupirućih infrastrukturnih proširenja i raznih drugih dijelova. Stvarna kompleksnost implementirane servisno orijentirane arhitekture je jedinstvena unutar svakog pojedinog poduzeća, no to je obilježeno uvođenjem novih tehnologija i platformi koje posebno podržavaju stvaranje, izvršavanje i evoluciju servisno orijentiranih rješenja. Kao rezultat toga, izgradnja tehnološke arhitekture oko servisno orijentiranog arhitekturnog modela uspostavlja okruženje prikladno za logiku rješenja koja je bila dizajnirana u skladu s načelima projektiranja servisno orijentiranog dizajna.

4.1. Arhitekturne osnove

Da bi se bolje moglo razumjeti sljedeće poglavlje potrebno je objasniti neke osnovne pojmove koji su bitni za razumijevanje [2, str 27]:

- arhitektura tehnologije – temeljni fizički dizajn nečega,
- tehnološka infrastruktura – temeljno tehnološko okruženje koje uključuje softver i hardver,
- računalni program – samostalni sustav koji može biti kupljeni proizvod ili razvijena aplikacija.



Slika 2: Međusobna ovisnost elemenata

(Izvor: izrada autora prema [2, str 27])

Pojam arhitektura posuđen je od tradicionalnog povezivanja s dizajnom i izgradnjom zgrada i objekata. Podrijetlo te riječi pomaže da se uspostavi analogija koja je korisna za razlikovanje tehnološke arhitekture i tehnološke infrastrukture. Za primjer može poslužiti zgrada koja ima izražen fizički dizajn u obliku nacrta ili opće specifikacije. Međutim, zgrada postoji u određenom okruženju. To okruženje može, ali i ne mora pružiti veliku potporu zgradi kako bi ona ispunila svoju svrhu. Na primjer, ured ili stambena zgrada koja je smještena unutar grada podržavaju ulice, elektrane, komunalne usluge, kabeli i mnogi drugi resursi koje osigurava gradska okolina. Takvo okruženje koje pruža podršku može se poistovjetiti s tehnološkom infrastrukturom. Da bi takva zgrada mogla iskoristiti sve prednosti tih infrastrukturnih proširenja, njezin fizički dizajn ih mora integrirati kao dio svoje službene arhitekture. Dakle, arhitektura specifikacije za zgradu će obuhvatiti dijelove okolne infrastrukture koji su relevantni za zgradu.[2, str 27] Isto takvo preklapanje postoji u IT svijetu,a na koji način se to odražava biti će objašnjeno u nastavku.

4.2. Tehnološka arhitektura

Tehnološka arhitektura izražava osnovne i temeljne aspekte fizičkog dizajna za neki dio tehnologije. Dok većina hardvera ima svoju arhitekturnu tehnologiju u tipičnom IT poduzeću najviše je potrebno obratiti pažnju na arhitekturnu strukturu aplikacije, tj. softvera koji se razvija. [2, str 27-28] Ako se radi o kupljenom softveru potrebno je razumjeti njegov unutarnji dizajn kako bi se mogla osigurati kompatibilnost s već uspostavljenom okolinom u poduzeću. Za programe koji se razvijaju unutar poduzeća fizički dizajn postaje odgovornost

poduzeća koje razvija taj program. Prilikom dizajniranja novog softvera potrebno je uzeti u obzir okruženje u kojem će taj softver biti implementiran i na koji način će ispunjavati svoju svrhu. U većini poduzeća već postoje okruženja koja su u obliku poslužitelja, operacijskih sustava i ostalih platformi. Svi nabrojani dijelovi smatraju se tehnološkom infrastrukturom.

Opseg tehnološke arhitekture može varirati ovisno o tome što se projektira. Erl [2, str 28] navodi sljedeće tipove:

- arhitektura komponenti - u distribuiranom računalnom okruženju ovo predstavlja fizičku strukturu pojedinačnog softverskog programa koji postoji kao komponenta,
- arhitektura aplikacije - tehnološka arhitektura s ograničenim fizičkim granicama u okruženju implementacije određene aplikacije ili sustava, a u distribuiranom računalnom okruženju može obuhvatiti više arhitekturi komponenata,
- arhitektura integracije - tehnološka arhitektura dvije ili više povezanih aplikacija ili sustava, uključujući tehnologije, resurse ili proširenja dodana kako bi se omogućila njihova integracija,
- tehnološka arhitektura poduzeća - za razliku od komponentne, aplikacijske i integracijske arhitekture koje se najčešće dokumentiraju u specifikacijama dizajna prije izrade programa, tehnološka arhitektura poduzeća često završi kao dokumentiranje onoga što već postoji unutar poslovnog okruženja.

4.3. Tehnološka infrastruktura

Unutar tipičnog IT poduzeća, tehnološka infrastruktura predstavlja okolinu u kojoj se programi implementiraju. Kao i pojam arhitektura, infrastruktura također može biti kvalificirana zajedno sa softverom ili hardverom kako bi se odredili pojedini dijelovi tog okruženja. Uobičajeni oblici hardverske infrastrukture uključuju:

- poslužitelje i radne stanice,
- usmjerivače, vatrozide i mrežnu opremu,
- rezervno napajanje, kabele i ostalu računalnu opremu.

Vrste softvera koje se obično smatraju dijelom tehnološke infrastrukture poduzeća uključuju [2, str 30-31]:

- operacijski sustavi i API sustavi,
- radno okruženje i agenti na razini sustava,
- baze podataka i direktoriji,
- programi za upravljanje transakcijama i redovima poruka,

- posrednički softver i adapteri,
- upravljanje korisničkim računima i tehnologije vezane uz sigurnost.

Ono što općenito razlikuje tehnologiju koja je dio infrastrukture od one koja je ekskluzivna za određenu komponentu ili aplikaciju je to da je dostupna na više aplikacija ili sustava te stoga postoji kao resurs poduzeća. Primjer uobičajenog softvera koji može biti klasificiran kao dio infrastrukture ili specifičan za arhitekturu aplikacije je baza podataka.

4.4. Računalni program

Program u smislu računalnog programa je izvršni softver koji se pokreće na računalu. Bolja definicija bi bila da je to skup uputa računalu što i kako nešto učiniti i izvesti.[18] Računalni program može biti kupljen ili napravljen prema nečijem dizajnu. Dio dizajna programa može biti dokumentiran preko specifikacije arhitekture aplikacije. Obično je taj dio usredotočen na pozadinu s naglaskom na cjelokupnu strukturu programa, resursne zahtjeve i tehnologiju. Tipična specifikacija arhitekture neke aplikacije često se nadopunjuje s dodatnim vrstama projektne dokumentacije, kao što su funkcionalne specifikacije koje ilustriraju tijek i stil sučelja korisničkih programa. Tu se još pojavljuju i detaljni dokumenti dizajna koji uspostavljaju rutine i algoritme za programiranje. Ovisno o konvencijama, metodologijama ili željama IT odjela te dodatne informacije o dizajnu mogu se, ali se i ne moraju smatrati dijelom službene tehnološke arhitekture aplikacije. [2, str 32]

4.5. Dizajnerski okvir

U dokumentiranju okvira dizajna nalaze se pojmovi koje je Erl [2, str 33-34] podijelio na sljedeći način:

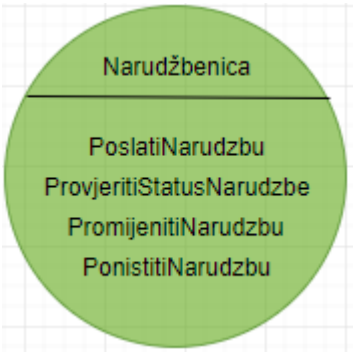
- karakteristika dizajna – svojstvo programa ili tehnološke arhitekture od koje proizlazi način na koji je dizajniran, a može biti konkretne kvalitete, isto kao što je i program sastavljen od komponenata koje daju finu ili grubu strukturu,
- princip dizajna – prihvaćena praksa u industriji s određenim ciljem dizajna pri čemu se servisno orijentirana paradigma dizajna sastoji od niza dizajnerskih načela koja se zajednički primjenjuju za postizanje ciljeva servisno orijentiranog računalstva,
- uzorak dizajna – dokazano rješenje zajedničkog projektnog problema dokumentiranog u konzistentan format.
- standard dizajna – konvencija dizajna prilagođena pojedinačno od strane organizacije kako bi se pružilo pouzdano rješenje za podršku specifičnom poslovanju organizacije.

4.6. Servis

Servis je jedinica logike rješenja prema kojoj je servisna orijentacija primijenjena u velikoj mjeri, a pri čemu možemo razlikovati jedinicu logike kao servisa od jedinice logike koja postoji kao zaseban objekt ili komponenta. Nakon konceptualnog modeliranja servisa, faze servisno orijentiranog dizajna i projektiranja, servis se implementira kao fizički neovisni program sa specifičnim dizajnerskim karakteristikama koje podržavaju postizanje strateških ciljeva povezanih sa servisno orijentiranim računalstvom. [2, str 37]

4.7. Sposobnosti servisa

Svakom servisu dodijeljen je vlastiti funkcionalni kontekst koji se sastoji od skupa funkcija ili sposobnosti koje su povezane s tim kontekstom. Servis se može smatrati spremnikom funkcionalnosti povezanih sa zajedničkim ciljem. Pojam sposobnosti servisa ne utječe na način na koji je servis implementiran. Ovaj pojam može biti koristan tijekom faza modeliranja servisa kada fizički dizajn servisa još nije određen. [2, str 38]

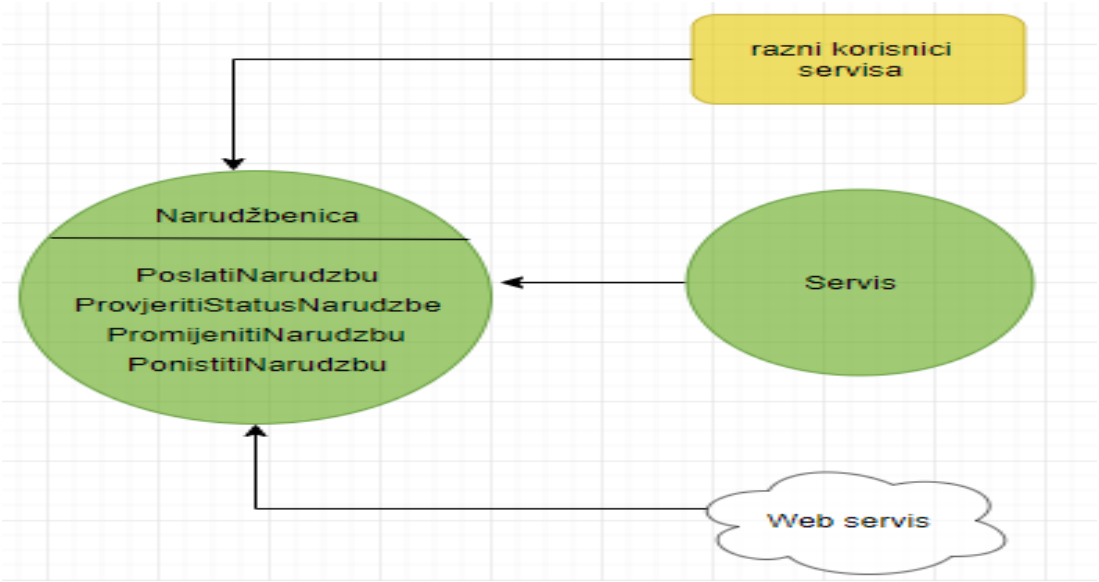


Slika 3: Primjer servisa i njegovih funkcionalnosti
(Izvor: izrada autora prema [2, str 37])

Na slici 3 prikazan je primjer servisa koji ima dodijeljen nekakav kontekst i pripadne funkcionalnosti koje može izvršavati. Iz navedenih funkcionalnosti može se vidjeti da su te funkcionalnosti međusobno povezane.

4.8. Korisnici servisa

Kada program pozove i komunicira sa servisom program je označen kao korisnik servisa. Vrlo je važno razumjeti da se ovaj pojam odnosi na privremenu ulogu koja je preuzeta od strane programa kada uključuje servis u razmjeni podataka. Kao dobar primjer može poslužiti aplikacija za radnu površinu koja služi za razmjenjivanje poruka sa servisom. Kada je aplikacija u interakciji sa servisom smatra se da je aplikacija korisnik servisa. Također, može se osmisliti servis za pozivanje i interakciju s drugim servisima. U tom slučaju servis privremeno postaje korisnik servisa.[2, str 39]



Slika 4: Prikaz različitih vrsta korisnika servisa
(Izvor: izrada autora prema [2, str 39])

Slika 4 prikazuje različite vrste korisnika servisa. U različitim slučajevima korisnici servisa mogu biti drugi servisi, web servisi i razni drugi korisnici koji mogu koristiti servis.

4.9. Mediji za implementaciju servisa

Važno je promatrati i pozicionirati SOA kao arhitektonski model koji je neutralan za bilo koju tehnološku platformu. Ovime se poduzeću daje sloboda da neprestano slijedi strateške ciljeve povezane sa servisnom orijentacijom iskorištavajući neprestani napredak tehnologije. Erl [2, str 44] navodi da servisi mogu biti implementirani kao:

- komponenta,
- web servis,
- REST servis.

Ovisno o načinima na koje servisi mogu biti implementirani, svaka implementacijska tehnologija koja se može koristiti za stvaranje distribuiranog sustava može biti prikladna za servisnu orijentaciju.

4.9.1. Servis kao komponenta

Komponenta je program dizajniran tako da bude dio distribuiranog sustava. Komponenta pruža tehničko sučelje usporedivo s tradicionalnom primjenom sučelja kroz koje izlaže javne sposobnosti kao metode, a čime se omogućuje pozivanje iz drugih programa. Uz to same komponente obično se oslanjaju na specifične razvojne platforme i izvršne tehnologije,npr. komponente mogu biti izrađene pomoću Java ili .NET alata te se tada razmjenjuju u radnom okruženju koje je sposobno podržavati odgovarajuće komponente komunikacijske tehnologije.[2, str 45]

4.9.2. Servis kao Web servis

Web servis je tijelo logike rješenja koje pruža fizički odvojen tehnički ugovor koji se sastoji od WSDL definicije i jedne ili više XML definicijskih shema. Ugovor o web servisu izlaže javne mogućnosti kao operacije uspostavljajući tehničko sučelje. Servisna orijentacija može se primijeniti na dizajn web servisa. Web servisi pružaju arhitekturni model u kojem dobavljači ne mogu utjecati na izgled ugovora o servisu što pogoduje nekolicini ciljeva koji su povezani sa servisnom orijentacijom. [2, str 45]

4.9.3. REST servis

Reprezentativni prijenos stanja - REST (eng. *Representational State Transfer*) osigurava način konstruiranja distribuiranih sustava na temelju pojma resursa. REST servisi su programi dizajnirani s naglaskom na jednostavnost, skalabilnost i upotrebljivost. Tako dizajnirani servisi mogu se dodatno oblikovati primjenom načela servisne orijentacije.[19]

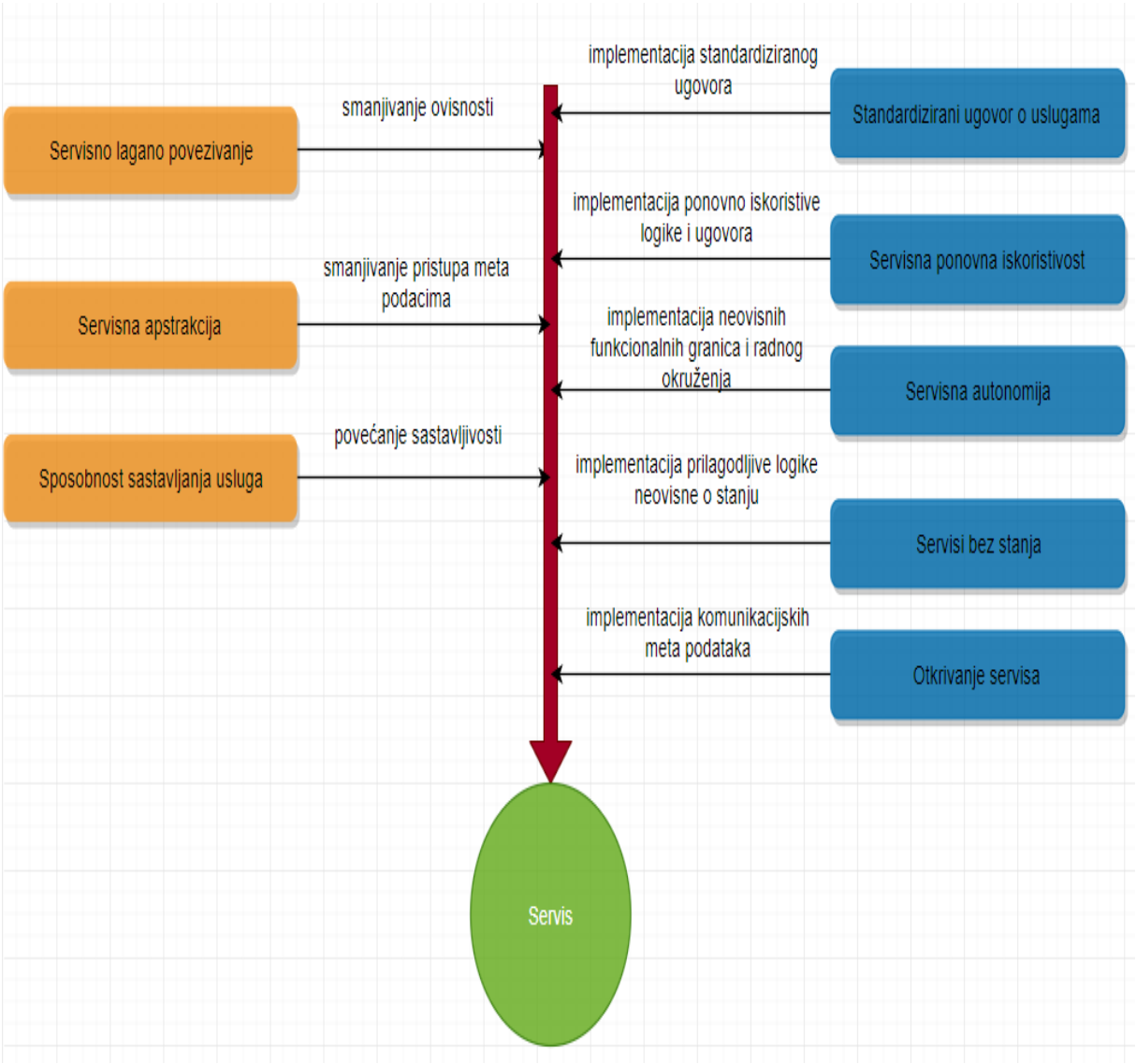
5. Osnove servisno orijentiranog računalstva

Servisno orijentirano računalstvo temelji se na ostvarivanju određenog stanja. To traži da se uzmu u obzir i dodatna razmatranja u pogledu dizajna u svemu što se izgradi da bi svi pokretni dijelovi servisno orijentiranog rješenja mogli podržavati ostvarivanje tog stanja, njegovati rast i evoluciju. To ciljano stanje je atraktivno i povezano s nizom specifičnih ciljeva i povlastica. Da bi se potpuno mogla razumjeti servisno orijentirana arhitektura potrebno je poznavati metodu pomoću koje se ostvaruju i postižu ti ciljevi te što je krajnji rezultat tih ciljeva i koje su njihove koristi. [2, str. 48]

Razumijevanje navedenih pojmova omogućuje procjenu zahtjeva tehnološke arhitekture. Erl [2, str 48-50] navodi osam različitih principa dizajna koji su dio servisno orijentirane paradigme pri čemu se svaki od tih principa bavi ključnim aspektom dizajna servisa osiguravajući specifične karakteristike dizajna koje se ostvaruju na svakom servisu. Kada se zajednički smisleno primjenjuju, principi servisno orijentiranog dizajna oblikuju logiku rješenja u nešto što se može nazvati servisno orijentirano. U nastavku su ukratko opisana načela servisno orijentiranog dizajna [2, str 48-50]:

- standardizirani ugovor o servisima – servisi koji se nalaze unutar istog inventara servisa su u sukladnosti s istim standardima ugovora,
- servisno lagano povezivanje – ugovori o servisima nameću niske zahtjeve klijenata i sami su odvojeni od okruženja,
- apstrakcija servisa – ugovori o servisima sadrže samo bitne informacije i informacije o servisima su ograničene ovisno o tome što je objavljeno u ugovorima o servisima,
- ponovna iskoristivost servisa – servisi sadrže i izražavaju logiku koja se ponovno može iskoristiti,
- autonomija servisa – servisi ostvaruju visoku razinu kontrole kroz vrijeme trajanja izvršenja,
- servisi bez stanja – servisi smanjuju potrošnju resursa odgađanjem upravljanja tako dugo dok to nije potrebno,
- otkrivanje servisa – servisi se dopunjuju komunikacijskim meta podacima koji se mogu učinkovito otkriti i interpretirati,
- sposobnost sastavljanja servisa – servisi su učinkoviti sudionici sustava bez obzira na to kolika je veličina i složenost kompozicije.

Iduća slika će prikazati neke perspektive o tome kako ova načela utječu na dizajn servisa.



Slika 5: Prikaz utjecaja načela servisno orijentiranog dizajna
(Izvor: izrada autora prema [2, str 50])

Na slici se može vidjeti da primjena načela na desnoj strani rezultira konkretnim konstrukcijskim karakteristikama, dok se načela s lijeve strane obično ponašaju kao regulatorni utjecaji, osiguravajući tako ravnotežu servisno orijentirane aplikacije. Kao što je prethodno spomenuto, rješenje se smatra servisno orijentirano onda kad je servis usmjeren u značajnoj mjeri. Samo razumijevanje paradigme dizajna je kao takvo nedovoljno. Za dosljednu i uspješnu primjenu servisne orijentacije zahtijeva se tehnološka arhitektura koja je prilagođena i odgovara preferencijama dizajna, pogotovo na početku kada se servisi isporučuju, a posebno kada se servisi prikupljaju i sastavljaju u složene kompozicije.

5.1. Strateški ciljevi servisno orijentiranog računalstva

Usmjerenost na servise započeta je kao pristup i potpora dizajnu u postizanju ciljeva i prednosti koji su povezani sa SOA-om i servisno orijentiranim računalstvom pri čemu različiti autori navode različite prednosti, a neke od najvažnijih su navedene su u nastavku.

- Povećana unutarnja interoperabilnost – servisi koji se nalaze unutar određene granice su osmišljeni tako da su prirodno kompatibilni na takav način da se mogu učinkovito sastaviti i promijeniti kao odgovor na promjenjive poslovne zahtjeve. [4, str 97]
- Povećana federacija – servisi uspostavljaju jedinstveni ugovorni sloj koji skriva nejednakost omogućavajući im da se individualno upravljaju i razvijaju. [2, str 51]
- Povećane opcije za diverzifikaciju dobavljača – servis orijentiran na okruženje je baziran na dobavljačkoj neutralnosti arhitekturnog modela, dozvoljavajući organizaciji da si razvije arhitekturu u tandemu s poslovnim okruženjem bez ograničenja na karakteristike dobavljačke platforme. [2, str 51]
- Povećano usklađivanje poslovnih i tehnoloških domena – neki su servisi osmišljeni s poslovno usmjerenim funkcionalnim kontekstom koji im omogućuje da preslikavaju i razvijaju poslovanje organizacije. [4, str 100]
- Povećanje ROI – većina servisa se isporučuje i gleda kao IT sredstvo od kojeg se očekuje da će pružiti vrijednost koja premašuje trošak isporuke i vlasništva. [4, str 100]
- Povećana organizacijska agilnost – novi i promjenjivi poslovni zahtjevi mogu biti puno brže ispunjeni ako se osigura okruženje u kojem se rješenja mogu sastaviti ili unaprijediti uz smanjeni napor, koristeći ponovnu upotrebljivost i interoperabilnost postojećih servisa.[2, str 51]
- Smanjeno IT opterećenje – poduzeće kao cjelina je u cijelosti pojednostavljeno kao rezultat prethodno opisanih ciljeva i prednosti koje omogućuju IT-u da bolje podupire organizaciju pružajući veću vrijednost uz manje troškova. [4, str 102-103]

5.2. Četiri karakteristike SOA-e

U prethodnom poglavlju objašnjena je servisno orijentirana paradigma dizajna zajedno s ciljevima. Sada je potrebno obratiti pažnju na fizički dizajn servisno orijentiranih rješenja i njihovo okruženje. U svrhu ostvarivanja ciljeva u servisnoj orijentaciji Erl [2 , str 52-60] navodi četiri bazne karakteristike koje je potrebno gledati u svrhu ostvarivanja forme SOA-e, a to su:

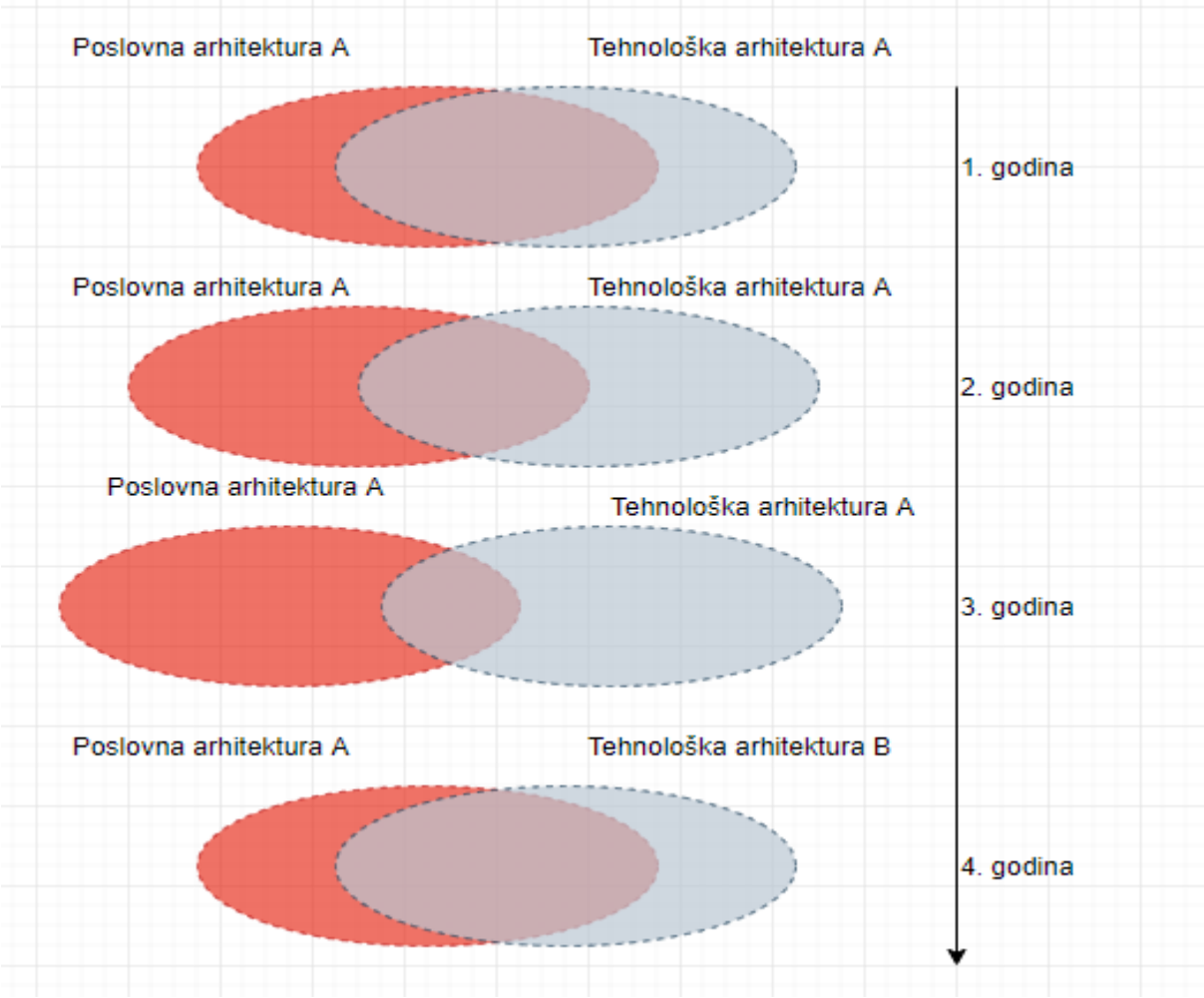
- poslovno upravljana – tehnološka arhitektura nalazi se na istoj razini kao i trenutna poslovna arhitektura, pri čemu je kontekst redovno održavan da bi tehnološka arhitektura mogla evoluirati i razvijati se zajedno s poslovnim dijelom kroz vrijeme,
- dobavljačko neutralna – arhitekturni model nije baziran na dobavljačkoj platformi, omogućujući tako kombiniranje ili zamjenu različitih tehnologija dobavljača na način kako bi se povećalo ispunjavanje poslovnih zahtjeva na trajnoj bazi,
- usmjereno na poduzeće - opseg arhitekture predstavlja značajan segment poduzeća, omogućujući ponovnu uporabu i kompoziciju servisa te upotrebu servisno orijentiranih rješenja koja obuhvaćaju tradicionalne aplikacijske silose,
- usmjereno na kompoziciju – arhitektura nasljeđuje mehanizam podržavanja preko ponovljivih agregacijskih servisa, dozvoljavajući prilagodbu stalnim promjenama putem agilnih sklopova servisnih kompozicija.

Ove karakteristike pomažu razlikovati SOA-u od drugih arhitekturnih modela i također definiraju temeljne zahtjeve koje tehnološka arhitektura mora ispuniti da bi u potpunosti bila servisno orijentirana.

5.2.1. Poslovno upravljana

Tradicionalne tehnološke arhitekture u većini slučajeva su osmišljene kao podrška rješenjima koja su isporučena da bi kratkoročno zadovoljavale poslovne zahtjeve. Zbog veličine posla koje poduzeće obuhvaća, dugoročni strateški ciljevi poduzeća često se ne uzimaju u obzir prilikom definiranja arhitekture. Zbog takvog pristupa kao rezultat se često javlja razlika koja nastaje u tehničkom okruženju i poslovnom smjeru zajedno sa zahtjevima organizacije. To postupno razdvajanje poslovnog i tehnološkog dijela rezultira time da tehnološka arhitektura više ne može ispunjavati poslovne zahtjeve, ali se i sve teže prilagođava promjenjivim poslovnim potrebama. Kada je tehnološka arhitektura poslovno orijentirana, poslovna vizija, ciljevi i zahtjevi su postavljeni kao primarni dijelovi koji utječu na arhitekturni model. To povećava potencijalno usklađivanje tehnologije i poslovanja te samim

time dozvoljava da se tehnološka arhitektura razvija zajedno s organizacijom u cijelosti. Rezultat je stalno povećanje vrijednosti i životnog vijeka arhitekture. [2, str 53]



Slika 6: Prikaz razdvajanja poslovne i tehnološke arhitekture kroz godine
(Izvor: izrada autora prema [2, str 54])

Na slici se može primijetiti da je su najčešće kod samog početka unutar poduzeća poslovna i tehnološka arhitektura na istoj razini. No postoji velika šansa da tehnološka arhitektura s godinama neće moći pratiti razvoj poslovne arhitekture te se tu kroz nekoliko godina javlja sve veća razlika. Budući da se kroz godine njihova sinkronizacija sve više smanjuje to utječe i na smanjeno ispunjavanje obaveza. Najčešće se na kraju desi slučaj da se mora razviti sasvim nova tehnološka arhitektura koja resetira taj cijeli ciklus ponovno na početak.

6. Arhitektura Web servisa

Servisi su odgovorni za internetsku komunikaciju između računala. Računala ih koriste za međusobno komuniciranje putem interneta. U stvari, to su samo sučelja koja koriste web stranice koje se nalaze na uređajima krajnjih korisnika. Podaci koji su povezani pohranjuju se na udaljenom poslužitelju i prenose se na klijentski uređaj putem API-ja koji pružaju web servisi nekog drugog korisnika. API-ji mogu koristiti različite arhitekture za prijenos podataka s poslužitelja na klijenta. Dugo vremena je SOAP bio protokol koji se koristio za prijenos poruka gotovo na svakom web servisu. U posljednje vrijeme razvojni inženjeri moraju izrađivati lagane web stranice i mobilne aplikacije koje nisu tako zahtjevne. Zbog toga je REST protokol dobio na važnosti i vrlo brzo je postao popularan. U 2018. godini većina javnih internetskih servisa prebacila se na REST protokol te se prijenos podataka vršio u kompaktnom i jednostavnom JSON formatu za razmjenu poruka. I jedan i drugi protokol se još uvijek koriste ovisno o samoj potrebi. [4, str 37-42]

6.1. SOAP vs REST

Ako se gleda usporedba između SOAP i REST protokola zajednička stvar im je ta da oba omogućuju stvaranje vlastitog programskog sučelja koji omogućuje prijenos podataka iz jedne aplikacije u drugu. Kreirano sučelje omogućuje zaprimanje zahtjeva, ali i slanje odgovora na te zaprimljene zahtjeve putem raznih internetskih protokola. Neki od protokola su HTTP i SMTP. [5] Mnoge popularne web stranice pružaju javna sučelja za svoje korisnike. Jedan od primjera takvog sučelja je Google Maps koji ima javni REST API koji omogućuje prilagođavanje Google Maps-a s vlastitim sadržajem. Postoje razna sučelja koja su kreirale tvrtke za svoju internu upotrebu. SOAP i REST su dva različita protokola koji po pitanju prijenosa podataka pristupaju s potpuno različite točke gledišta. SOAP je standardizirani protokol koji šalje poruke koristeći protokole kao što su HTTP i SMTP. Specifikacije SOAP protokola su službeni web standardi, a razvija ih W3C. Za razliku od SOAP-a, REST nije protokol, već je to arhitekturni stil. REST arhitektura postavlja skup smjernica koje treba slijediti ako se želi pružiti RESTful web servis. [6]

Budući da je SOAP službeni protokol, to za sobom nosi stroga pravila i napredne sigurnosne značajke kao što su ugrađena ACID (eng. *Atomacity, Consistency, Isolation, Durability*) sukladnost i autorizacija. Veća složenost zahtijeva više propusnosti i resursa što može dovesti do sporijeg vremena učitavanja stranice. REST je stvoren za rješavanje problema SOAP-a jer ima fleksibilniju arhitekturu. Sastoji se samo od labavih smjernica i omogućuje razvojnim inženjerima da preporuke provode na svoj način. Omogućuje različite

formate poruka kao što su HTML, JSON, XML i obični tekst, dok SOAP dopušta samo XML. Pošto je RESTful web servis jednostavnije arhitekture , sami servisi imaju bolje performanse. Zbog toga je postao nevjerojatno popularan u eri mobilne telefonije u kojoj je čak nekoliko sekundi bitno da bi se ostalo na samom vrhu i ostvarilo što veće prihode.[6]

6.2. REST

REST je arhitektonski stil koji definira skup preporuka za oblikovanje slabo povezanih aplikacija koje koriste HTTP protokol za prijenos podataka. REST ne propisuje kako je potrebno implementirati principe na nižoj razini. Umjesto toga, smjernice omogućuju razvojnim inženjerima da implementiraju detalje prema vlastitim potrebama. Web servisi koji su izgrađeni u skladu s REST arhitektonskim stilom nazivaju se RESTful web servisi. [7]

Da bi se uspješno moglo izraditi REST sučelje potrebno je slijediti šest arhitektonskih ograničenja [20]:

- uniformno sučelje – zahtjevi različitih klijenata trebaju izgledati isto, odnosno isti resurs ne bi trebao imati više od jedne adrese,
- odvajanje klijenta i poslužitelja – klijent i poslužitelj trebaju djelovati neovisno te bi trebali međusobno komunicirati samo putem zahtjeva i odgovora,
- bez stanja – ne bi trebalo biti niti jedne sesije na strani poslužitelja te bi svaki zahtjev trebao sadržavati sve informacije koje poslužitelj treba znati,
- resursi koji se mogu spremiti – odgovori poslužitelja trebaju sadržavati podatke o tome mogu li se podaci koji se šalju spremiti ili ne, a resursi koji se dohvaćaju za spremanje trebali bi imati broj verzije kako bi se izbjeglo traženje istih podataka više puta,
- slojeviti sustav – možda postoji nekoliko slojeva poslužitelja između klijenta i poslužitelja koji vraća odgovor, ali to ne bi trebalo utjecati na zahtjev niti na odgovor,
- kod na zahtjev – ovaj dio nije obavezan, ali kada je potrebno može sadržavati izvršni kod koji klijent može izvršiti.

Na internetu se može pronaći puno primjera REST web servisa, posebice jer puno društvenih mreža (npr. Facebook) nude REST sučelja kako bi razvojni inženjeri mogli bez problema u svoje aplikacije integrirati usluge koje su ponuđene preko sučelja. Takvi javni API-ji dolaze s detaljnom dokumentacijom gdje se mogu dobiti sve informacije koje su potrebne da bi se uspješno mogli povući podaci s API-ja.

6.2.1. REST HTTP metode

Kod slanja zahtjeva REST servisa koriste se HTTP metode. Metode i njihove operacije prikazane su u sljedećoj tablici.

Tabela 1: Prikaz REST HTTP metoda sa pripadajućim operacijama
(Izvor: izrada autora prema [8])

HTTP zahtjev	Operacija
POST	Kreiranje
GET	Čitanje
PUT	Zamjena
PATCH	Modifikacija
DELETE	Brisanje

POST je glagol koji se najčešće koristi za stvaranje novih resursa, konkretno se koristi za stvaranje podređenih resursa. Drugim riječima, pri stvaranju novog resursa pomoću POST zahtjeva servis vodi računa o tome da se tom novom resursu dodijeli novi specijalni ključ po kojemu će se on u budućnosti moći prepoznati. Nakon uspješnog kreiranja u zaglavlju se vraća status 201 što znači da je resurs uspješno kreiran. POST nije potpuno siguran, stoga se preporuča za zahtjeve koji nisu od velike važnosti. Upućivanje dva identična POST zahtjeva najvjerojatnije će rezultirati s dva izvora koji sadrže iste informacije. [8]

GET zahtjev koristi se za čitanje ili dohvaćanje reprezentacije resursa. Povratna informacija najčešće se vraća u XML ili JSON formatu i HTTP odgovoru 200 (OK). U slučaju da se desi pogreška, vraća se kod pogreške 404 (eng. *Not Found*) ili 400 (eng. *Bad Request*). Prema dizajnu HTTP specifikacije GET zahtjevi koriste se samo za čitanje podataka, a ne za njihovo mijenjanje. Stoga ako se upotrebljava na ovaj način, smatra se sigurnim jer se servis može pozivati bez rizika da će doći do promijene podataka. Prvi poziv servisa ima isti učinak kao i da se isti zahtjev pošalje 10 puta, rezultat će na kraju biti isti. Važno je napomenuti da se nesigurne operacije ne smiju izlagati putem GET zahtjeva, tj. nikada se pomoću tog zahtjeva ne bi smjeli mijenjati nikakvi resursi na poslužitelju. [8]

PUT zahtjev najčešće se koristi za opciju ažuriranja. Međutim, PUT se također može koristiti za stvaranje resursa u slučaju kada klijent umjesto poslužitelja odabere ID resursa. Drugim riječima, ako je PUT upućen na ID adresu resursa koji ne postoji, a zahtjev sadrži

sve potrebne podatke za opisivanje resursa, resurs bi se mogao kreirati i spremiti. Kreiranje resursa ovom metodom poželjno je izbjegavati jer za to već postoji POST metoda koja se koristi isključivo za kreiranje resursa. Nakon uspješnog ažuriranja resursa vraća se odgovor sa statusom 200 ili 204 (eng. *No Content*) ako se u tijelu poruke ne šalje nikakav sadržaj. PUT zahtjev nije siguran za rad jer mijenja stanje na poslužitelju te je potrebno oprezno raditi s tim pozivom. [8]

DELETE zahtjev je jednostavan za razumijevanje, a koristi se za brisanje resursa identificiranog s adresom. Nakon uspješnog brisanja vraća se HTTP status s kodom 200 zajedno s tijelom odgovora. Nakon što se prvi put uspješno izvrši brisanje nekog resursa, ako se ponovi taj isti poziv na tom resursu se ne događa ništa jer on ne postoji, tj. obrisan je u prethodnom pozivu.[8]

6.2.2. REST i JSON

Upotreba REST arhitekture omogućuje pružateljima API-ja da isporučuju podatke u više formata kao što su običan tekst, HTML, XML, YAML i JSON koji je jedan od njegovih najomiljenijih značajki. Zahvaljujući sve većoj popularnosti REST-a, lagani i lako čitljivi JSON formati su vrlo brzo stekli popularnost jer su prikladni za jednostavnu i brzu razmjenu podataka. JSON (eng. *JavaScript Object Notation)* je jednostavan format za analizu i razmjenu podataka. Iako JSON u svom imenu sadrži „JavaScript“ moguće ga je koristiti i s ostalim programskim jezicima. Datoteke zapisane u JSON formatu sastoje se od skupa parova imena/vrijednosti i poredanih popisa vrijednosti koje su univerzalne strukture podataka, a koristi ih većina programskih jezika. Zbog toga se JSON može lako integrirati s bilo kojim programskim jezikom.[9] U nastavku su prikazani XML i JSON primjeri zapisa.

- XML zapis

```
<sadrzaj>
<ime>ime</ime>
<prezime>prezime</prezime>
<popisAdresa>
<adresa>
<kucniBroj>kucni broj</kucniBroj>
<imeUlice>ime ulice</imeUlice>
<posta>naziv poste</posta>
</adresa>
</popisAdresa>
</sadrzaj>
```

- JSON zapis

```
{
   "ime":"ime",
   "prezime":"prezime",
   "popisAdresa":{
      "adresa":[
         {
            "kucniBroj":"kucni broj",
            "imeUlice":"ime ulice",
            "posta":"naziv poste"
         }
      ]
   }
}
```

Iz primjera zapisa vidljivo je da je JSON jednostavniji oblik zapisa zbog čega ga je jednostavnije čitati i pisati. U većini slučajeva je idealan za razmjenu podataka putem interneta. Ipak, XML i dalje ima neke prednosti kao što je npr postavljanje metapodataka unutar oznaka te bolje upravljanje s pomiješanim sadržajem.[9]

6.3. SOAP

SOAP (eng. *Simple Object Access Protocol*) ili u prijevodu jednostavan protokol za pristup objektima, je protokol za razmjenu podataka u decentraliziranim i distribuiranim okruženjima. Prednost SOAP-a je što može raditi s bilo kojim protokolom aplikacijskog sloja, kao što su HTTP, SMTP, TCP ili UDP. Podaci koji se vraćaju su u XML formatu. Sigurnost, autorizacija i rad s greškama ugrađeni su u protokol za razliku od REST-a.[10]

SOAP protokol slijedi formalni i standardizirani pristup koji određuje kako kodirati XML datoteke koje vraća API. U stvarnosti ta poruka je obična XML datoteka koja se sastoji od sljedećih dijelova [11]:

- omotnica (eng. *envelope*) – u omotnici se nalaze početne i završne oznake poruke,
- zaglavlje (eng. *header*) – sadrži neobavezne atribute poruke koji omogućavaju proširenje SOAP poruke na modularan i decentraliziran način,
- tijelo (eng. *body*) – sadrži XML podatke namijenjene primatelju,
- greška (eng *fault*) – sadrži informacije o pogreškama koje su se dogodile tijekom obrade poruke.

U nastavku je dan SOAP primjer poruke.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
<soap:Header>
</soap:Header>
<soap:Body>
  <soap:Fault>
  </soap:Fault>
</soap:Body>
</soap:Envelope>
```

Iz primjera je moguće vidjeti sve prethodno navedene dijelove SOAP poruke. Trenutno su ti dijelovi prazni jer je ovo sam primjer strukture poruke.

SOAP se i dalje koristi za web servise na razini poduzeća za koja je potrebna visoka razina sigurnosti i područja gdje se provode složene transakcije. API-ji za sustave plaćanja, upravljanje identitetom,financijske i telekomunikacijske usluge najčešće su korišteni primjeri upotrebe SOAP-a. Jedan od najpoznatijih SOAP API-ja je javni PayPal koji omogućuje prihvaćanje plaćanja putem PayPal-a i kreditnom karticom jednostavnim dodavanjem gumba na web stranicu. Pomoću toga im se dopušta prijava putem PayPal-a i omogućuje im se izvršavanje drugih radnji koje su vezane uz tu uslugu.[9]

Osim što je SOAP vrlo pouzdan, on može biti vrlo dobro rješenje u slučajevima kad se ne može koristiti REST. Iako trenutno većina davatelja internetskih usluga želi razmjenjivati zahtjeve i odgovore bez stanja, postoje slučajevi u kojima se jednostavno stanje mora čuvati i odraditi operaciju koja je zahtijevana. [11]

6.4. Usporedba SOAP vs REST

Unatoč popularnosti REST-a, SOAP i dalje ima svoje mjesto u svijetu web servisa. Osnove karakteristike SOAP-a i REST-a prikazane su tablicom u nastavku.

Tabela 2: Usporedba SOAP i REST zahtjeva
(Izvor: izrada autora prema [9])

	SOAP	**REST**
Značenje	Simple Object Access Protocol	Representational State Transfer
Dizajn	Standardizirani protokol sa predefiniranim pravilima koja se prate	Arhitekturni stil koji ima labave vodilje i preporuke
Pristup	Funkcijski orijentiran	Podatkovno orijentiran
Stanje	Bez stanja, ali moguće je pretvoriti API da ima stanje	Bez stanja, nema sesija sa strane servera
Sigurnost	WS-sigurnost sa SSL podrškom, ugrađena ACID usklađenost	Podržava HTTPS i SSL
Performanse	Zahtijeva veću propusnost i računalnu snagu	Zahtijeva manje resursa
Format poruke	Samo XML	Običan tekst, HTML, XML, JSON, YAML
Protokoli za prijenos	HTTP, SMTP, UDP i ostali	Samo HTTP
Preporučeni za	Poslovne aplikacije, aplikacije visoke sigurnosti, financijske usluge, telekomunikacijske usluge	Javni API-ji za web servise, mobilne usluge, društvene mreže
Prednosti	Jaka sigurnost, standardiziranost, proširivost	Skalabilnost, bolje performanse, fleksibilnost
Nedostaci	Loše performanse, veća složenost, manja fleksibilnost	Manja sigurnost, nije prikladna za distribuirana okruženja

U tablici su prikazane glavne značajke REST-a i SOAP-a te se prema njima može vidjeti iz kojeg razloga se koristi SOAP, a iz kojeg REST.

7. Uzorci servisno orijentirane arhitekture

U ovom poglavlju biti će objašnjen uzak skup uzoraka od kojih će nekolicina nakon toga biti korištena kod arhitekture i implementacije aplikativnog dijela rada.

7.1. Temeljni uzorci inventara (eng. *Foundational Inventory Patterns*)

Uzorci koji se nalaze u ovom području su temeljni za definiranje servisno orijentiranog arhitekturnog modela. Problemi s dizajnom riješeni su pomoću uzoraka koji pomažu u strukturiranju arhitektura kojima je cilj uspostaviti fleksibilno i agilno okruženje koje je pogodno logici rješenja koje je dizajnirano u skladu sa servisnom orijentacijom. Ovisno o arhitekturnom modelu na koji su više orijentirani, Erl navodi četiri osnovne podijele koje su prikazani tablicom u nastavku.

Tabela 3: Orijentacija temeljnih uzoraka inventara
(Izvor: izrada autora prema [2, str 113])

	Poslovno orijentiran	**Prodajno neutralan**	**Orijentiran na poduzeće**	**Kompozicijski orijentiran**
Inventar poduzeća	X	X	X	X
Domenski inventar	X	X	X	X
Logička centralizacija	X		X	X
Normalizacija servisa				X
Servisni slojevi	X		X	X
Kanonski protokol			X	X
Kanonska shema			X	X

7.1.1. Normalizacija servisa (eng. *Service Normalization*)

Granica servisa definirana je funkcionalnim kontekstom i skupnim granicama svojih mogućnosti. Kada se radi o servisima koje isporučuje više projektnih timova, postoji rizik da će se neke funkcionalnosti početi preklapati. To dovodi do denormalizacije inventara što može uzrokovati neke od problema kao što su [2, str 131]:

- nemogućnost uspostavljanja servisnih sposobnosti i
- složenija arhitektura u kojoj servisi s funkcionalnošću koja se preklapa mogu prestati biti sinkronizirani jer pružaju iste funkcije na različite načine.

Da bi se taj problem izbjegao, servisi su kolektivno modelirani prije nego što se kreiraju njihovi pojedinačni fizički ugovori. To pruža mogućnost da se svaka granica servisa isplanira na način kojim bi se osiguralo da se ne preklapa s drugim servisima. Rezultat je popis usluga s većim stupnjem funkcionalne normalizacije. [2, str 131]

Ciljevi ovog uzorka najbolje se ostvaruju korištenjem autonomije na razini servisa tijekom faze modeliranja servisa. Koraci su podijeljeni na sljedeći način [2, 132-133]:

- identificiranje i dekompozicija poslovnog procesa,
- raspoređivanje pojedinih dijelova procesa u odgovarajuće nove ili postojeće servise,
- provjeravanje da se dvije servisne granice ne preklapaju.

Prethodno spomenuti koraci su dijelovi većeg procesa modeliranja servisa koji uzima u obzir i mnoga druga razmatranja. Da bi se u potpunosti mogla primijeniti normalizacija servisa potrebno je da se taj proces provodi iterativno, jednom za svaki poslovni proces koji je povezan s istim servisnim područjem. Kroz te iteracije se granice i funkcionalni kontekst servisa neprestano usavršavaju i validiraju.

7.1.2. Servisni slojevi (eng. *Service Layers*)

Unutar uobičajenog inventara servisa postoje servisi koji imaju slične funkcionalne kontekste. Međutim ti servisi mogu se različito dizajnirati i implementirati ovisno o prirodi projekta kojeg je potrebno isporučiti. To dovodi do propuštanja prilike za uspostavljanje dosljednosti u definiranju granica servisa. Rezultat je popis servisa koji se ne mogu lako podijeliti u skupine za apstrakciju poddomene. [2, str 143]

Arhitektura servisnog inventara može formalno uspostaviti klasifikacijske profile koji će predstavljati uobičajene vrste servisa unutar određenog inventara. Ti profili nazivaju se servisni modeli od kojih svaki predstavlja jedinstveni skup dizajnerskih karakteristika povezanih s dobro definiranom servisnom kategorijom. Servisni modeli čine osnovu ovog uzorka u toj skupini servisa koje su u skladu s jednim modelom te tako uspostavljaju srodni logički arhitekturni sloj povezanih funkcionalnosti. Primjenom servisnih slojeva osiguravaju se dizajnirani servisi koji odgovaraju uobičajenim tipovima s istim temeljnim karakteristikama koje su izvedene iz zajedničkih modela servisa. Ti servisi mogu formirati logičke domene unutar inventara koje se mogu razvijati i biti upravljane kao grupe. [2, str 144]

Servisni slojevi generalno se definiraju prije implementacije servisnih inventara. Tijekom faze modeliranja servisnih inventarnih planova, planira se funkcionalna priroda servisnih kandidata koja pomaže u određivanju slojeva koji su najprikladniji za određeni problem. Stoga bilo koji dati inventar servisa može imati različite slojeve. Jedino pravilo je to da postoje najmanje dva sloja, a Erl [2, str 146] navodi dva osnovna sloja:

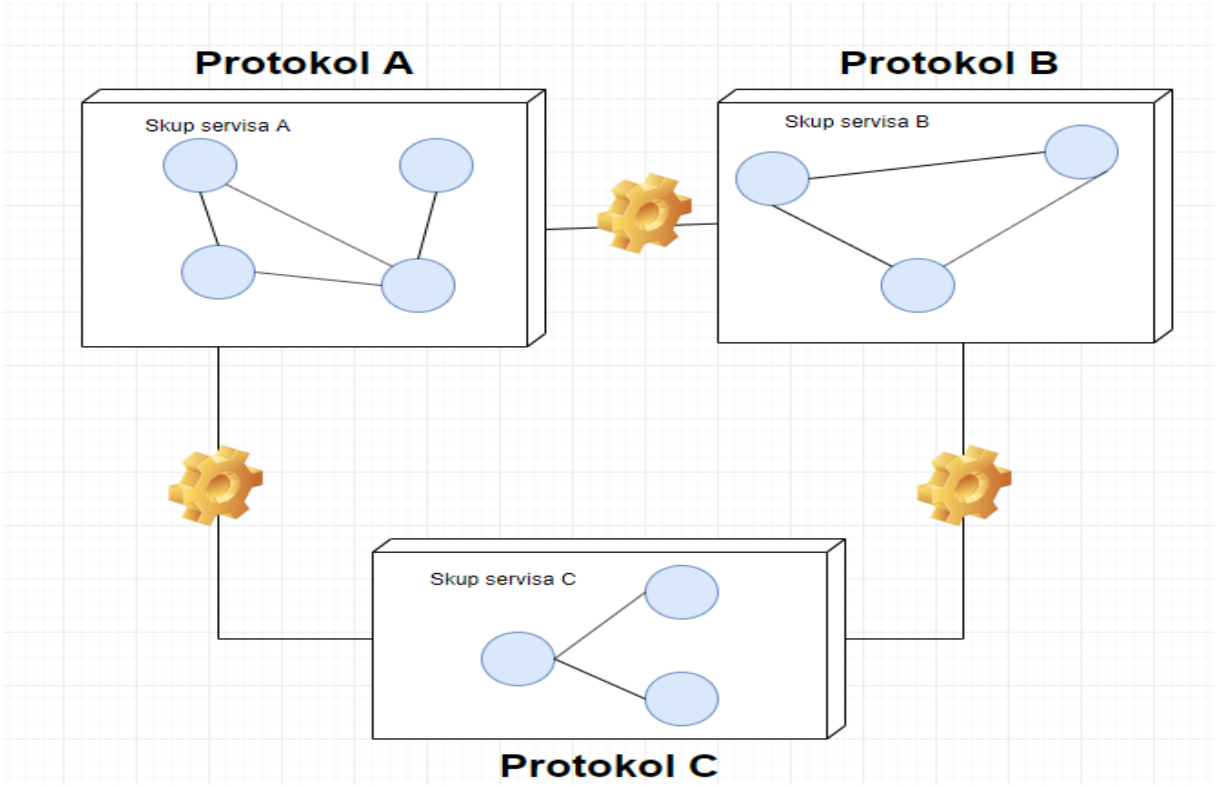
- sloj koji predstavlja jednonamjensku (neagnostičku) logiku i
- sloj koji predstavlja višenamjensku (agnostičku) logiku.

Utvrđivanje koji se modeli servisa trebaju ili ne trebaju upotrebljavati unutar servisnog inventara zahtijeva poznavanje tipova logika koje se nalaze unutar granica samog inventara. Stoga se servisni slojevi često razvijaju uz ponovljene faze iteracija servisno orijentirane analize, a ponekad čak zahtijevaju reviziju prethodno modeliranih servisnih kandidata. Rezultat toga je povećanje vremena i truda potrebnog za definiranje planova servisnog inventara. Izuzetak je kada su modeli servisa već uspostavljeni kao dizajnerski standardi te se u tom slučaju mogu koristiti kao osnova za planiranje ispravnih servisnih slojeva od samog početka. Jednom kada su slojevi odabrani, oni postaju dizajnerski standardi za cijeli inventar, pri čemu se svaki naknadno definirani servis mora uklopiti u neki od uspostavljenih servisnih slojeva. Nakon što su servisi izgrađeni u skladu s osnovnim modelima servisa tu strukturu je vrlo teško promijeniti bez da se ne poremeti cijeli inventar servisa. [2, str 148]

7.1.3. Kanonski protokol (eng. *Canonical Protocol*)

Servisi koji podržavaju različite komunikacijske tehnologije ugrožavaju interoperabilnost, potencijalno ograničavaju količinu podataka potrošačima i uvode potrebu za nepoželjnim protokolom koji treba raditi izmjenu nad podacima. Da bi se taj problem mogao izbjeći ili ukoliko ga je potrebno riješiti, potrebno je uspostaviti jedinstvenu tehnološku komunikacijsku arhitekturu preko koje će servisi moći vršiti interakciju. Svaki servis stoji kao samostalni program. Kada ti programi razmjenjuju informacije, potrebno je uspostaviti vezu uz korištenje komunikacijske tehnologije. Kada su programi osmišljeni za korištenje različitih komunikacijskih tehnologija, oni tada nisu kompatibilni i ne mogu razmjenjivati informacije bez uključivanja posebnog programa koji može prevesti jednu komunikacijsku tehnologiju u drugu. Za takve stvari se koristi *„Bridging“* protokol. Izgradnja servisa s različitim implementacijskim tehnologijama nije neuobičajena, ali dopuštanje izgradnje servisa koji koriste različite komunikacijske tehnologije unutar iste arhitekture može kao rezultat donijeti mnoga ograničenja. Npr., grupe servisa temeljene na istim komunikacijskim okvirima vjerojatno će se isporučiti zajedno kao dio istog projekta. Ako će bilo koji od tih servisa trebati biti povučen u neku novu servisnu skupinu koja se nalazi u nekom drugom projektu i uz to da se u tom novom projektu koristi drugačiji komunikacijski protokol to će uzrokovati probleme

kod njihovog međusobnog povezivanja, a možda čak to povezivanje neće biti niti moguće. [2, str 151]



Slika 7: Slučaj komunikacije između servisnih inventara
(Izvor: izrada autora prema [2, str 151])

Na slici iznad prikazana su tri servisna inventara. Svaki od tih inventara sadrži unutar sebe servise koji međusobno komuniciraju. Da bi ti servisi mogli međusobno komunicirati oni koriste određeni komunikacijski protokol preko kojeg komuniciraju. U slučaju iznad, javlja se problem kada se pokušava izvršiti povezivanje svih tih servisnih inventara međusobno. Erl [2, str 153] nudi rješenje kako bi se taj problem mogao izbjeći ili riješiti. Potrebno je upotrijebiti jedinstveni komunikacijski protokol koji će se koristiti. Sve tehnologije koje su povezane u najnižem dijelu s komunikacijskim okvirom koji je ujedno i standardiziran, jamči nam da će biti podržana osnovna tehnološka kompatibilnost između servisa. Da bi se osiguralo da svi servisi budu međusobno sposobni za učinkovito komuniciranje, potrebno je pažljivo odabrati komunikacijsku tehnologiju koja će se koristiti. Tu postoje različiti okviri koji se mogu koristiti za učinkovito komuniciranje. Najčešće korišteni protokoli za transport i prijenos poruka su HTTP i SOAP koji su široko podržani i standardizirani. Međutim, čak i kada se koriste web servisi u sklopu s nekim od odabranih protokola, potrebno je primijeniti jedinstvenu verziju kako bi se izbjegla moguća razlika u verzijama protokola. Zbog toga se

koristi posebni profil koji je vjerojatno ključni dio ovog uzorka kao sredstvo koje osigurava tehnološku kompatibilnost između različitih verzija koje se koriste kao tehnološki standardi kod izrade web servisa.

Erl [2, str 155] navodi nekoliko ključnih razloga koje je potrebno uzeti u obzir prilikom standardiziranja na jedan komunikacijski protokol:

- zrelost i pouzdanost – ovisno o tome koji je protokol izabran, servisna interakcija će biti ograničena samim okvirom koji će biti odabran pa je zbog toga potrebno dobro procijeniti ukupnu zrelost i pouzdanost komunikacijske tehnologije koja se odabire,
- dugovječnost – ako postoje nedoumice da bi se kroz neko dogledno vrijeme mogla napustiti tehnologija u kojoj se želi implementirati komunikacija, potrebno je uzeti u obzir rizike koji se javljaju odabirom istog,
- trošak – određeni odabiri komunikacijskih protokola mogu nositi niz skrivenih troškova, a neki od tih troškova mogu se odnositi na rješavanje nedostataka u samom protokolu, ali tu mogu nastati i drugi troškovi ako je protokol dio neke vlasničke platforme koja zahtijeva naknade za licenciranje.

Iz prethodno navedenog zaključujem da je kanonski protokol zapravo vrlo bitan uzorak koji je potrebno dobro proučiti jer u slučaju da se on izostavi mogu se pojaviti razni problemi kod povezivanja web servisa. Zbog toga, ovaj uzorak bit će korišten u programskom dijelu aplikacije.

7.2. Uzorci sloja logičkog inventara (eng. *Logic Inventory Layer Patterns*)

Servisni slojevi služe za organiziranje servisa unutar inventara u logičke skupine. Svaki sloj je temeljen na vrsti servisa i stoga predstavlja skup koji je u skladu s tom vrstom. Te vrste odgovaraju industrijskim klasifikacijama koje se referiraju kao servisni modeli. Erl [2, str 164] navodi tri najčešća servisna modela:

- model uslužnog servisa (eng. *Utility Service Model*) – tip servisa koji pruža generičku logiku obrade koja nije klasificirana kao poslovna logika pri čemu je servisna logika idealna za višekratnu upotrebu,
- model entitetnog servisa (eng. *Entity Service Model*) – poslovno orijentiran servisni tip koji se izvodi od jednog ili više poslovnih subjekata koji se također može višekratno koristiti,

- model servisnog zadatka (eng. *Task Service Model*) - poslovno orijentirani servis namjerno kreiran da ne bude višenamjenski iz razloga jer je njegov funkcionalni opseg ograničen na jednonamjensku logiku poslovnog procesa.

Ta tri modela u direktnoj su vezi s tri uzorka dizajna [2, str 164]:

- apstrakcija usluge (eng. *Utility apstraction*) – koristi se za uspostavu servisnog sloja koji predstavlja uslužne servise,
- apstrakcija entiteta (eng. *Entity Abstraction*) –predstavlja entitetske servise,
- apstrakcija procesa (eng. *Process Abstraction*) – formira servisni sloj koji se sastoji od servisnih zadataka.

Kada su u pitanju servisni modeli i servisni slojevi, uvijek se radi razlika između poslovne i ne poslovne logike. Logika koja je klasificirana kao poslovno usmjerena, izvedena je iz modela specifikacija i poslovne analize. Primjeri takvih dokumenata uključuju definiciju poslovnih procesa, specifikacije, ontologije, taksonomije, modela logičkih podataka, dijagrame referenci poslovnih subjekata i razne druge dokumente koji se odnose na poslovnu, podatkovnu i arhitekturu informacija općenito. [2, str 165]

7.2.1. Apstrakcija usluge

Postoje vrste logike koje su potrebne za automatizaciju bilo kojeg poslovnog zadatak, ali se uvijek može pronaći generička funkcionalnost koja nema veze s poslovnim modelima. IT okruženje koristi različite tehnologije, proizvode, baze podataka i druge resurse koji nude različite funkcionalnosti i koriste se u razne svrhe. Vrsta logike koja se ne fokusira na poslovanje može se smatrati uslužnom logikom. U slučaju kada se želi napraviti funkcionalnost koja treba biti automatizirana u poslovnom procesu često dolazi do miješanja logike poslovnih procesa, poslovnih pravila i drugih oblika poslovne logike. U tom slučaju pojavljuju se miješani servisi koji onemogućavaju strateški dizajn i upravljanje uslužnom logikom na praktičan način. Za primjer se može uzeti funkcionalnost generičke obrade koja može riješiti više međuovisnih problema, a ugrađena je zajedno s logikom specifičnom za poslovne procese. Kod takvih slučajeva teško je izdvojiti generičku logiku zasebno da bi ona mogla biti ponovno iskorištena u nekim drugim slučajevima. [2, str 169]

Da bi se taj problem izbjegao, višenamjenske poslovno orijentirane funkcionalnosti servisa grupirane su u zasebne servise. Budući da ti servisi nisu specifični za nijedan zadatak, oni se mogu ponovno upotrijebiti za automatizaciju više zadataka. Rezultat je na kraju servisni sloj koji je definiran i upravljan. Takav način obrada zajednički je za sva poduzeća, ali postupak apstrakcije međuovisnih funkcionalnosti u logičke jedinice koje se mogu ponovno koristiti je izazovan zadatak. Jedan od izazova koji se stalno javlja u vezi s

takvim servisima je definiranje odgovarajućih servisnih konteksta. Za razliku od konteksta poslovnih usluga koji se mogu izvesti iz postojećih poslovnih modela, funkcionalni kontekst takvih servisa često je prepušten arhitektu ili razvojnom inženjeru. Stoga može biti izazovno postaviti servisni kontekst tako da on bude pogodan za ponovnu upotrebu. Erl [2, str 170] je naveo smjernice koje je dobro slijediti kada se želi napraviti apstrakcija:

- potrebno je izbjegavati servise koji spajaju puno funkcionalnosti zajedno iz razloga jer je takve servise teško izgraditi na način da se oni mogu ponovno koristiti, a sve to može uzrokovati lošu povezanost koja može voditi do ogromnih servisa,
- potrebno je vrlo jasno i konkretno definirati funkcionalni kontekst za svaki servis, ali je potrebno pružiti i prostor za mogućnost razvijanja zajedno s inventarom jer za razliku od poslovnih servisa koji imaju stroga pravila, kod uslužnih servisa te granice mogu se povećavati sve dok se čuva nadređeni kontekst,
- preporuča se korištenje kanonskog izražavanja za osiguravanje stvaranja ugovora o servisu koji su lako razumljivi jer uslužne servise obično proizvode tehnološki stručnjaci pa uvijek postoji opasnost da neki detalji u tim ugovorima pred javnošću budu previše tehnološki ili nejasno napisani.

Tijekom postupka modeliranja servisnih procesa, logika uslužnog sloja već je prethodno konceptualizirana. Nakon toga, kada su servisni ugovori spremni za definiranje, potrebno je primijeniti poseban postupak koji je usmjeren prema dizajnu uslužnih servisa kako bi se mogli riješiti jedinstveni problemi povezani s određenom vrstom servisa.

7.2.2. Apstrakcija entiteta

Kada se poslovna logika pokušava prikazati apstraktno nastoji se grupirati logiku tako da je ta grupa povezana s određenim zadatkom ili poslovnim procesom. Sva logika koja se potencijalno može ponovno upotrijebiti spojena je s jednonamjenskom procesnom logikom. Takvim grupiranjem gubi se potencijal ponovne upotrebe te logike. Poslovni analitičari koji su stručni u području entiteta često se razlikuju od onih koji su stručni na procesnoj razini. Kada se logika entiteta i procesa grupira zajedno za podršku automatizacije poslovnog zadatka taj dio je u vlasništvu analitičara koji je odgovoran za definiranje poslovnog procesa. Zbog toga se može dogoditi da se propusti prilika za provođenjem pravila koja se provode kod poslovnih entiteta, karakteristika i njihovih međusobnih odnosa. [2, str 175]

Svaka poslovna organizacija bavi se različitim poslovima koje je potrebno riješiti, bili to poslovi vezani uz ljude, dokumente, proizvode ili neke druge stvari. Te stvari zapravo se nazivaju poslovnim entitetima. Kako organizacije mijenjaju načine na koji posluju, vrlo je vjerojatno da će postojati potreba za kreiranjem novih procesa ili će se javiti potreba za

mijenjanjem starih procesa. No bez obzira na mijenjanje ili dodavanje novih procesa, obično su u te procese i dalje uključivani isti poslovni entiteti. U slučaju kada se žele dizajnirati višenamjenski servisi koji bi se mogli ponovno iskoristiti, potrebno je izgraditi servise koji se temelje na poslovnim entitetima. Takvi servisi su višenamjenski jer se svaki može iskoristiti pri automatizaciji različitih zadataka. [2, str 176] Ovaj uzorak dijeli višenamjensku poslovnu logiku u posebnu grupu servisa koji imaju višenamjensku logiku, a baziraju se na poslovnim entitetima.

Postupak dizajniranja mora biti dovršen kako bi se moglo krenuti s kreiranjem standardiziranih ugovora o servisima koji se temelje na kontekstu poslovnih entiteta. Često je logički ili organizacijski entitetski model podataka osnova za te kontekste. Dobiveni servisni sloj sastoji se od višenamjenskih poslovnih servisa grupiranih u posebne cjeline koje se mogu ponovno upotrijebiti u brojnim različitim poslovnim procesima. Svaki entitetski servis može biti upravljan grupom koja se sastoji od poslovnih analitičara koji imaju odgovarajuću stručnost kako bi se servis neprekidno mogao razvijati u skladu s poslovanjem uzevši u obzir očuvanje cjelovitosti servisa i njegovih mogućnosti.[2, str 176]

7.2.3. Apstrakcija procesa

Servisi se mogu osmisliti tako da se jednonamjenska i višenamjenska logika grupiraju u svakom servisu. To se može dogoditi u slučaju kad servise isporučuju projektni timovi pojedinačno ili kada se servisna orijentacija ne poštuje kao dio metode isporuke. Erl [2, str 182] navodi da ovakav pristup ima i svoje posljedice:

- smanjuje se mogućnost upotrebe načela dizajna ponovne upotrebe na širokoj razini,
- nameće se složenost kada su u pitanju poslovni procesi i entiteti kojima upravljaju različiti pojedinci različite stručnosti,
- teško se primjenjuje višenamjenski kontekst te se smanjuju vjerojatnost za uspješno apstrahiranje logike jednonamjenskog servisa u legitimne servise.

"Poslovna logika obuhvaća više entitetskih granica servisa koji su apstrahirani u posebni funkcionalni kontekst povezan sa servisnim modelom.“[2, str 183] Ova definicija navodi da se kreiranjem nadređenog sloja servisa, koji je odgovoran za zadržavanje poslovnog toka i logike servisne kompozicije potrebne za provođenje nadređenog poslovnog procesa, može povećati prilagodljivost jer je svaka poslovna logika podložna poslovnim promjenama. Kao rezultat, mogućnost pristupa i održavanja logike u zasebnom skupu servisa može smanjiti napor potreban da se odgovori na promjene, istovremeno štiteći višenamjenske servise u drugim slojevima od utjecaja koji izazivaju te promjene. Zbog toga

jer je dostupan popis servisa, poslovne promjene često će izazvati potrebu obrade višenamjenskih servisa bez njihove direkte izmjene. [2, str 183-185]

7.3. Uzorci centralizacije inventara (eng. *Inventory Centralization Patterns*)

Ranije spomenuti uzorci dizajna usmjereni su na organiziranje servisa u logičke grupe, što bi značilo da njihova primjena ne utječe na fizičku lokaciju pojedinih servisa. Kao primjer može poslužiti da se ne očekuje da svi servisi unutar sloja apstrakcije entiteta borave na istom računalu ili poslužitelju.

Prema Erl [2, str 192] postoje četiri podjele koje govore o fizičkim aspektima arhitekture inventara servisa, a to su:

- centralizacija procesa (eng. *Process Centralization*) –logika koja je povezana s različitim poslovnim procesima treba se zadržati na istom mjestu,
- centralizacija sheme (eng. *Shema Centralization*) – standardizira sheme kao fizički neovisne dijelove arhitekture inventara tako da se one mogu dijeliti između servisa, ali i koristiti neovisno o servisima,
- centralizacija politike (eng. *Policy Centralization*) – pomaže uspostaviti globalne i domenske politike koje su fizički izolirane, ali ujedno mogu biti i dijeljene te primjenjivane na više servisa,
- centralizacija pravila (eng. *Rule Centralization*) – usredotočena je na razdvajanje procesne logike i pohrane podataka specifične za upravljanje podacima poslovnih pravila.

Svaki od ovih navedenih centralizacijskih uzoraka podiže određeni stupanj standardizacije inventara. To je važan i ponavljajući aplikacijski zahtjev koji ujedno utječe i na primjenu svih uzoraka koji su navedeni. Za razliku od uzoraka koji su prethodno bili presudni za dizajn inventara, fizički uzorci centralizacije inventara mogu se smatrati bolje specifičnim. Navedeni uzorci se preporučuju, ali nisu nužni za uspostavljanje osnovnog servisnog inventara. [2, str 193]

7.3.1. Centralizacija procesa

U okruženjima koja sadrže velike količine servisnih inventara uobičajen je zahtjev za istovremenom podrškom automatizacije više poslovnih procesa. Logika poslovnih procesa obuhvaća poslovne entitete koji mogu biti smješteni u pojedinačne servise. Iako ti servisi postoje kao vršni članovi inventara servisa, činjenica da su oni neovisno implementirani

rezultira time da je organizacijska poslovna logika fizički distribuirana na više lokacija. Kada se javi potreba za promjenama, mogućnost za učinkovito proširenje ili čak kombiniranje procesne poslovne logike je smanjena jer je temeljnu logiku svakog pogođenog servisa potrebno pregledati i promijeniti ovisno o promjeni. Zbog prirode različitih poslovnih pravila neki procesi se ne mogu provoditi u stvarnom vremenu. Umjesto toga mogu uzrokovati dugotrajne servisne aktivnosti koje mogu trajati nekoliko minuta, sati, pa čak i dana. Neovisne implementacije servisa moraju biti opremljene tako da olakšaju te zahtjeve. U slučajevima kad je to tehnički izvedivo, ponavljanja proširenja implementacije u brojnim servisnim okruženjima mogu postati dosadna i naporna, pogotovo kada se radi o servisima koji su distribuirani na različitim fizičkim poslužiteljima, a možda čak i na različitim platformama.[2, str 193]

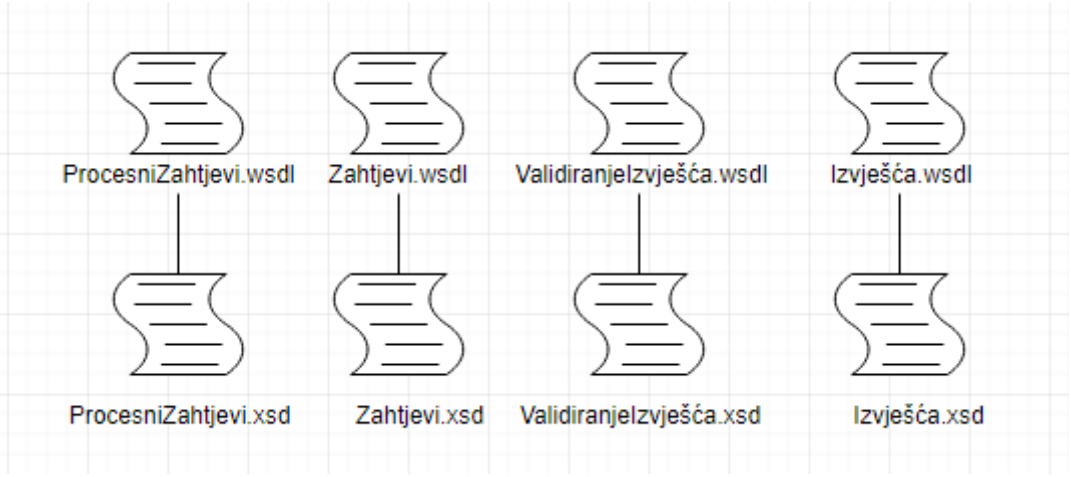
Zadaci koji se trebaju obavljati najčešće se implementiraju kao pojedinačni web servisi. Budući da svaki program sadrži ugrađenu logiku poslovnih procesa to rezultira fizički decentraliziranom arhitekturom. Da bi se taj problem mogao izbjeći, logika nadređenog poslovnog procesa koja predstavlja neke ili sve poslovne procese unutar određene domene je centralizirana na jedno mjesto. U platformu za upravljanje smješta se logika koja se u isto vrijeme kontinuirano i centralizirano izvršava i održava. Servisi koji obavljaju zadatke se i dalje mogu implementirati kao zasebni web servisi, no oni su dio platforme dok se logika zajedničkog poslovnog procesa nalazi i upravlja na središnjem mjestu. Da bi se ovaj uzorak dizajna mogao implementirati, Erl [2, str 195] navodi što takve moderne upravljačke platforme i njihove okoline najčešće sadržavaju:

- grafički korisnički alat koji omogućava korisnicima da se izražavaju i održavaju logiku poslovnih procesa,
- radno okruženje koje je u mogućnosti pružati dijeljenje upravljačke platforme servisnih zadataka i odgovarajući skup definicija poslovnih procesa koje su kreirane pomoću korisničkog alata,
- značajke koje su u skladu s industrijskim standardima koje se odnose na logičko izražavanje i izvršavanje poslovnih procesa.

Na kraju se može izvući zaključak da se logika za specifični poslovni problem definira pomoću korisničkog alata i zatim se upakira pomoću servisa, dok radno okruženje pokreće taj servis i omogućuje servisima da izvršavaju svoju logiku osiguravajući da se poštuje glavni zahtjev koji je navedeni, a to je podrška automatizacije više poslovnih procesa.

7.3.2. Centralizacija sheme

Prilikom izgradnje servisa za veća poslovna okruženja, kontekst svake usluge obično neće biti direktno vezan za jedno tijelo podatka. Kao primjer može se navesti servis potraživanja koji predstavlja zbirku funkcija vezanih uz potraživanja i stoga će biti primarno odgovoran za obradu podataka o zahtjevima. Iako je servis pozicioniran kao primarna krajnja točka za taj dio funkcionalnosti, vjerojatno neće biti jedini servis koji je napravljen za rad s podacima o zahtjevima. Kao rezultat toga, javlja se potreba za duplikatima shematskih modela što dovodi do definiranja servisnih ugovora s ponavljajućim sadržajem. Modeli podataka su preko takvih ugovora standardizirani, ali suvišna i decentralizirana primjena shema uvodi stalne izazove kod upravljanja i održavanja. [2 , str 200]



Slika 8 Prikaz WSDL definicija zajedno sa setom XML shema

(Izvor: izrada autora prema [2, str 201])

Na slici je prikazan skup WSDL definicija za koje je prilagođen odgovarajući skup XML shema. Ovo izgleda kao vrlo čista arhitektura, ali zapravo može unijeti redundanciju sadržaja. Već na ovom primjer može se primijetiti da se neki dijelovi preklapaju.

Sheme se mogu dizajnirati i implementirati neovisno o servisnim mogućnostima koje ih koriste za predstavljanje strukture i tipiziranje poruke. Kao rezultat toga, arhitektura sheme može se uspostaviti i standardizirati kao nešto odvojeno od roditeljskog servisnog sloja. Na primjer, ako je definirana jedna shema koja predstavlja podatke o zahtjevima, svaka usluga koja ima mogućnost obrađivanja zahtjeva koristila bi istu shemu. [2, str 201]

Slika 9: WSDL sheme koje imaju zajedničke XML sheme
(Izvor: izrada autora prema [2, str 201])

Na slici su prikazane WSDL definicije koje imaju zajedničke XML sheme koje dijele iste podatkovne modele poruka. Dobro je znati kako micanje suvišnog sadržaja iz svake sheme rezultira i shemama manjeg formata.

Ono što je bitno u ovom uzorku je to da je ovo uzorak za kreiranje neovisne shematske arhitekture. Ta arhitektura možda već postoji u organizaciji pogotovo ako su već prije poduzeti ozbiljni napori vezani uz standardizaciju XML sheme. Ako je sheme potrebno definirati kao dio SOA inicijative, preporučuje se da se one izrade prije završetka pojedinačnih ugovora o servisima. U idealnom slučaju uvrštavanje zasebnog sloja sheme uzima se u obzir nakon dovršetka plana servisnog inventara. Erl [2, str 202] predlaže slijedeće korake u pripremi za isporuku fizičkog inventara servisa:

- dovršiti nacrte servisnog inventara kako bi se uspostavio konceptualni prikaz planiranih servisa unutar inventara,
- odrediti centralizirane definicije sheme koje će predstavljati zajedničke poslovne entitete i skupove informacija koje će servisi obrađivati u inventaru,
- napraviti definiciju sheme primjenom dizajnerskih standarda kako bi se mogla osigurati dosljednost i normalizacija,
- kreirati WSDL definicije koristeći standardne sheme gdje god je to moguće, dopunjujući ugovor s potrebnim shemama specifičnim za servis.

7.4. Uzorci implementacije inventara (eng. *Inventory Implementation Patterns*)

U ovom poglavlju govorit će se uzorcima koji rješavaju probleme u vezi s dizajnom arhitekture inventara servisa. U ovom dijeli nalazi se skup specijaliziranih uzoraka koji pomažu u rješavanju problema na razini implementacije.

Svaki od navedenih uzoraka u nastavku cilja na određeno područje arhitekture inventara [2, str 226]:

- dvojni protokol (eng. *Dual Protocol*) – pruža fleksibilno rješenje koje se bavi izazovima uspostavljanja kanonskog komunikacijskog protokola,
- kanonski resursi (eng. *Canonical Resources*) – zagovaraju standardizaciju temeljnih tehnologija,
- spremište stanja (eng. State Repository) i servisi bez stanja (eng. *Stateful Services*) – pružaju alternativna rješenja za smještaj podataka o stanju izvođenja,
- servisna mreža (eng. *Service Grid*) – predlaže rješenje za smještaj stanja i toleranciju na greške.

Uz navedene uzorke pojavljuju se još dva uzorka usmjerena na rješavanje arhitekturnih briga izvan inventara za okruženja u kojima postoji više inventurnih domena ili kad postoji potreba za komunikaciju izvan granice inventara [2, str 226]:

- krajnja točka inventara (eng. *Inventory EndPoint*) - specijalizirani servis koji u ime servisa komunicira s vanjskim potrošačima,
- međudomenski uslužni sloj (eng. *Cross-Domain Utility Layer*) – predlaže dizajnersko rješenje koje mijenja lice inventurne domene protežući zajednički sloj uslužnih servisa preko inventurnih granica.

7.4.1. Dvojni protokol

Kao što je navedeno ranije, kanonski protokol zahtijeva da svi servisi budu napravljeni u istim komunikacijskim tehnologijama, ali može se pojaviti niz situacija u kojima se ne može udovoljiti svim zahtjevima koji trebaju biti implementirani u servisu. U slučaju kada odabranu komunikacijsku tehnologiju nije moguće koristiti po cijelom skupu servisa koji se razvija dolazi do ugrožavanja interoperabilnosti servisa, što znači da možda u nekim slučajevima oni ne bi mogli međusobno komunicirati. U takvim situacijama kada se posеže za drugom vrstom protokola za razmjenu podataka dolazi do narušavanja osnovnih načela kanonskog protokola.

Erl [2, str 228] navodi na koji način se mogu izbjeći situacije u kojima bi se mogla narušiti interoperabilnost kreiranog sustava nudeći kao rješenje kreiranje dvije razine servisa unutar istog inventara:

- primarna razina bazirana na preferiranom protokolu,
- sekundarna razina bazirana na alternativnom protokolu.

Ovakav način izgradnje servisa omogućuje korištenje sekundarnog protokola kad god se desi situacija da primarni protokol ne obuhvaća mogućnost rješavanja svih zahtjeva servisa. Takav način rješavanja omogućuje da se servisi bazirani na alternativnom protokolu promjene i premjeste u primarni protokol kada je to moguće.

Kao primjer standardnog protokola za transport i razmjenu poruka koji pruža velik opseg funkcionalnosti, a uz to je i dio tehnološke platforme koja možda nije prikladna za sve vrste usluga je razmjena SOAP poruka preko HTTP protokola. Servisi izgrađeni kao web servisi mogu uspostaviti standardizirani komunikacijski okvir temeljen na tim tehnologijama, ali ovakav izbor može izazvati neka pitanja [2, str 228]:

- SOAP uvodi obradu poruka koja možda nije primjerena servisima koji trebaju razmjenjivati granularne podatke ili ih isti potrošači trebaju pozvati više puta tijekom istog poslovnog procesa,
- dodatna obrada koja se odnosi na razmjenu poruka može se smatrati neprikladnom za usluge koje su fizički smještene na istom poslužitelju i ne zahtijevaju udaljenu komunikaciju,
- servis može kao zahtjev imati posebnu funkcionalnost koja se ne može implementirati na tehnološkoj platformi za web servise zbog nedostatka podrške dobavljača ili zbog ne podržanog standardna web servisa.

Svi servisi implementirani na alternativnom protokolu iz razloga jer bazni protokol tehnološki ne zadovoljava sve kriterije koji su potrebni za implementaciju servisne funkcionalnosti, smatraju se kandidatima za prelazak na primarni protokol čim tehnologija to dopusti.

7.4.2. Spremišta stanja

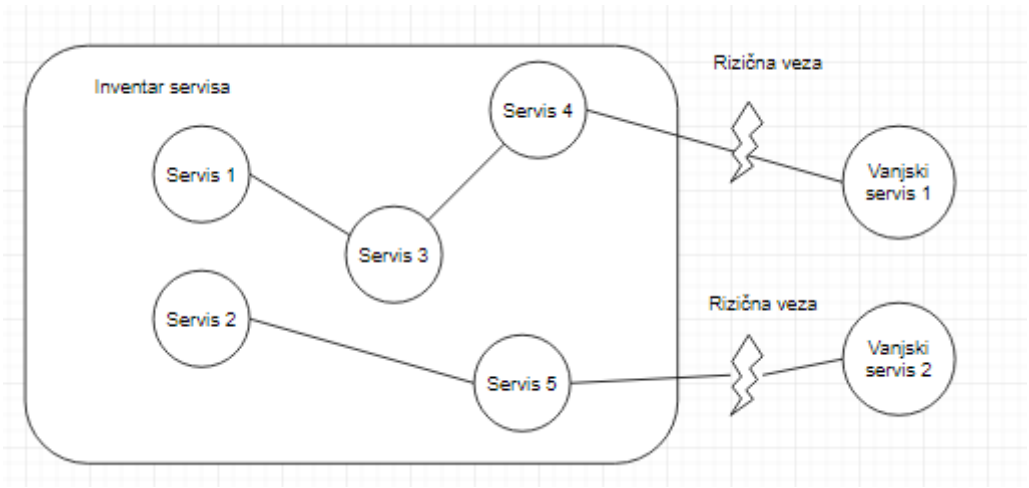
Velike količine podataka stanja koje se spremaju u memoriju za podršku aktivnosti unutar servisne grupe mogu izazvati veliku potrošnju memorije. Kod dugotrajnih aktivnosti uslijed povećanja potrošnje memorije može se osjetiti pad skalabilnosti servisa. Postoje slučajevi u kojima je potrebno iste podatke ili datoteke dohvatiti više puta tijekom servisne aktivnosti. Složeni sustavi često se susreću s produženim obradama tijekom kojih se podaci

ne zahtijevaju. U tom razdoblju podaci se pohranjuju u memoriji i troše resurse koji su možda potrebniji negdje drugdje tijekom izvođenja. [2, str 243]

Erl [2, str 243] kao rješenje predlaže primjenu spremišta stanja koje je korišteno kao arhitektonsko proširenje koje je dostupno bilo kojem servisu u svrhu privremene odgode korištenja podataka. Korištenjem spremišta stanja servisima se omogućuje uklanjanje nepotrebnih podataka stanja unutar memorije. Upotrebom spremišta stanja resursi koji se koriste za čuvanje podataka mogu biti iskorišteni za izvođenje drugih zadataka. Implementacija navodi kako se najčešće baza podataka koristi kao spremište podataka o stanju. Baza podataka nalazi se na istom fizičkom poslužitelju kako bi se minimiziralo vrijeme rada potrebno za spremanje i dohvaćanje podataka. Drugi navedeni način implementacije je taj da se kreiraju namjenske tablice unutar postojeće baze podataka. Kreiranje namjenskih tablica je manje učinkovita opcija, ali isto kao i prva opcija omogućuje privremeno spremanja podataka o stanju.

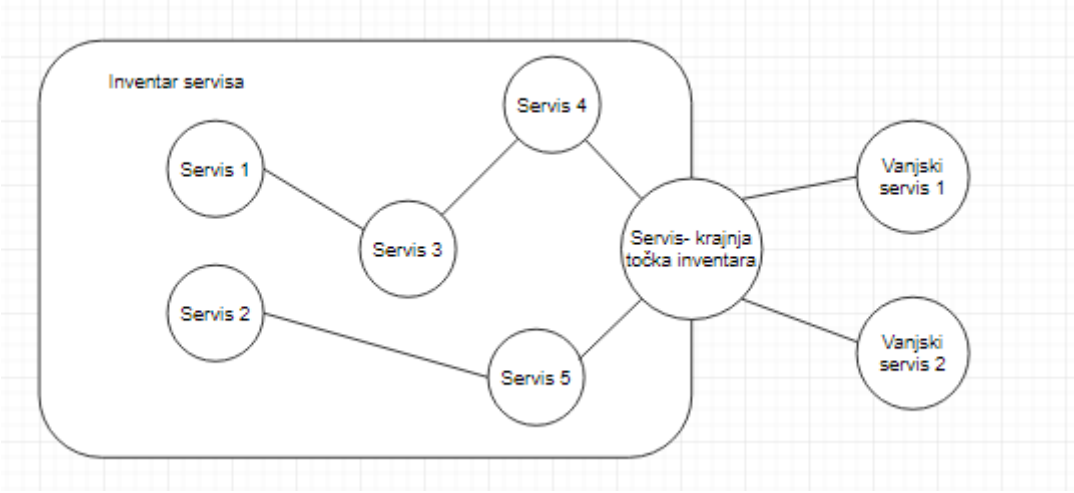
7.4.3. Krajnja točka inventara

Servisi koji obuhvaćaju neku određenu funkcionalnost nalaze se unutar istog servisnog inventara. Bilo koji od servisa koji se nalazi unutar grupe može sadržavati funkcionalnost koja je potrebna drugim servisima koji se nalaze izvan tog inventara. Iz sigurnosnih i upravljačkih razloga nije poželjno sve servise izložiti vanjskim korisnicima. Kada se pojavi slučaj da se neka od postojećih funkcionalnosti može izložiti vanjskim potrošačima servisa, neovisno radi li se o servisima unutar ili izvan organizacije, uvijek je potrebno obratiti pozornost na očuvanje interoperabilnosti, sigurnosti i privatnosti jer su to dijelovi u kojima se najčešće javljaju problemi kod izlaganja funkcionalnosti servisa vanjskim potrošačima. [2, str 261]



Slika 10: Povezivanje servisa iz inventara sa vanjskim servisima
(Izvor: izrada autora prema [2, str 261])

Na slici 10 prikazan je inventar servisa koji se sastoji od 5 različitih servisa. U prvom dijelu ovog poglavlja govori se o problemu s kojim se susreću razvojni timovi kada postoji mogućnost da se neke funkcionalnosti dostupne unutar inventara mogu iskoristi u servisima koji se nalaze izvan tog inventara. Rizično je dati direktan pristup vanjskim servisima unutar nekog inventara. Direktnim povezivanjem kreira se rizična veza koja utječe na sigurnost i privatnost servisa koji se nalaze unutar nekog inventara. Za rješavanje ovog problema kreira se posebni posrednički servis koji je pozicioniran na mjestu ulaska u popis inventara za potrošače koji se nalaze izvan inventara, a trebaju pristupiti izvornim uslugama koje se nalaze unutar inventara. Takav servis može se konfigurirati tako da u sebi sadržava logiku za interakciju s potrošačima i služi kao broker koji omogućuje lakšu i sigurniju komunikaciju vanjskih servisa s internim servisima inventara.



Slika 11: Povezivanje servisa iz inventara preko krajnje točke inventara sa vanjskim servisima

(Izvor: izrada autora prema [2, 261])

Na slici 11 prikazan je isti inventar servisa kao i na slici broj 10. U ovom slučaju dodan je još jedan servis koji predstavlja krajnju točku inventara. Taj servis je zadužen da pruži sigurnu komunikaciju između servisa unutar i servisa izvan inventara servisa. Spajanjem servisa smještenih unutar inventara s vanjskim servisima na način koji je prikazan na slici 11 ima nekoliko prednosti [2, str 262]:

- omogućuje se slobodnije upravljanje nad servisima unutar servisnog inventara iz razlog jer postoje situacije u kojima je potrebno mijenjati osnovne funkcionalnosti servisa, a upotrebnom krajnje točke moguće je uvesti logiku koja će i dalje odgovarati vanjskim,

- krajnja točka inventara može se prilagoditi vanjskim potrošačkim programima za koje je potrebno dodati niz sigurnosnih ograničenja kako bi se zadovoljili svi uvjeti za pristup inventaru servisa,
- omogućuje se kreiranje više zasebnih krajnjih točaka za svaku skupinu potrošača, a kao primjer može se navesti servis koji se koristi za komunikaciju s drugim inventarom servisa koji se nalazi unutar organizacije, dok se primjerice druga krajnja točka može implementirati tako da komunicira sa servisima ili programima koji se nalaze izvan organizacije,
- omogućuje se prilagodba servisa na komunikacijske protokole ili tehnologije koje koriste vanjski potrošači za razmjenu poruka.

Upotreba ovog uzorka povećava slobodu s kojom se servisi mogu kroz vrijeme razvijati i upravljati. Kod kreiranja servisa na taj način javlja se potreba za uvođenjem novih ugovora o servisima koje će netko trebati održavati. Uvođenjem novih ugovora i njihovim održavanjem javljaju se i dodatni troškovi koje je potrebno uračunati kod projektiranja.

7.5. Temeljni servisni uzorci

Uzorci dizajna spomenuti u ovom poglavlju predstavljaju korake koji su potrebni za podjelu i organiziranje rješenja servisa te mogućnosti pružanja podrške. Erl [2, str 296] uzorke dijeli u dvije skupine po kojima se primjenjuju:

- uzorci identifikacije servisa –definira se cjelokupna logika rješenja koja je potrebna za rješavanje određenog problema dok se dijelovi te logike pogodni za enkapsulaciju servisa naknadno filtriraju,
- uzorci definicije servisa – osnovni funkcionalni kontekst servisa je definiran i koristi se za organiziranje dostupne logike servisa dok se u višenamjenskim kontekstima servisna logika razdvaja na individualne mogućnosti.

Kada se identifikacija i definicija servisa spoje zajedno, potrebno je slijediti korake koji se mogu smatrati primitivnim postupkom modeliranja servisa. Svrha tog postupka je navođenje primitivnih razmatranja prilikom oblikovanja servisa u kandidate za daljnji razvoj.

7.5.1. Uzorci identifikacije servisa

Uzorci dizajna koji spadaju pod uzorke identifikacije servisa u osnovi provode razdvajanje problema podržanog servisnom orijentacijom. Tijekom postupka razdvajanja problema logika rješenja se razgrađuje i identificiraju se dijelovi pogodni za enkapsulaciju. Kao rezultat proizlazi osnovna logika koja nije organizirana, ali je spremna da se pretvori u

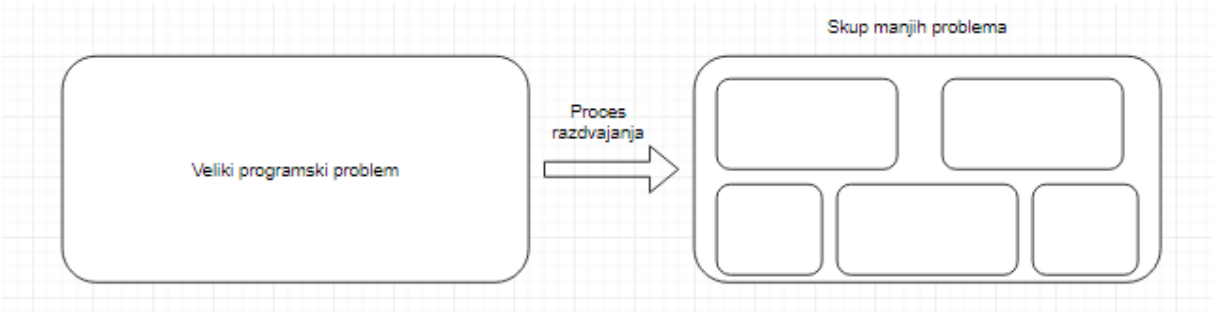
servise primjenom uzoraka i upotrebom servisno orijentiranih principa. Uzorci su podijeljeni na iduću način [2, str 297]:

- funkcijska dekompozicija,
- servisna enkapsulacija.

7.5.2. Funkcijska dekompozicija (eng. *Functional Decomposition*)

Za rješavanje složenog poslovnog problema potrebno je stvoriti odgovarajuću količinu manjih logika rješenja. Većina poslovnih procesa koji zahtijevaju automatizaciju predstavljaju velike probleme pa je zbog toga jedan od načina rješavanja problema automatizacije zapravo izgradnja aplikacije. [2, str 300]

Funkcionalna dekompozicija u osnovi je primjena teorije razdvajanja problema. Taj princip softverskog inženjerstva potiče razdvajanje većeg složenog problema na manje probleme za koje se mogu graditi odgovarajuće jedinice logike rješenja.



Slika 12: Proces razdvajanja problema
(Izvor: izrada autora prema [2, str 301])

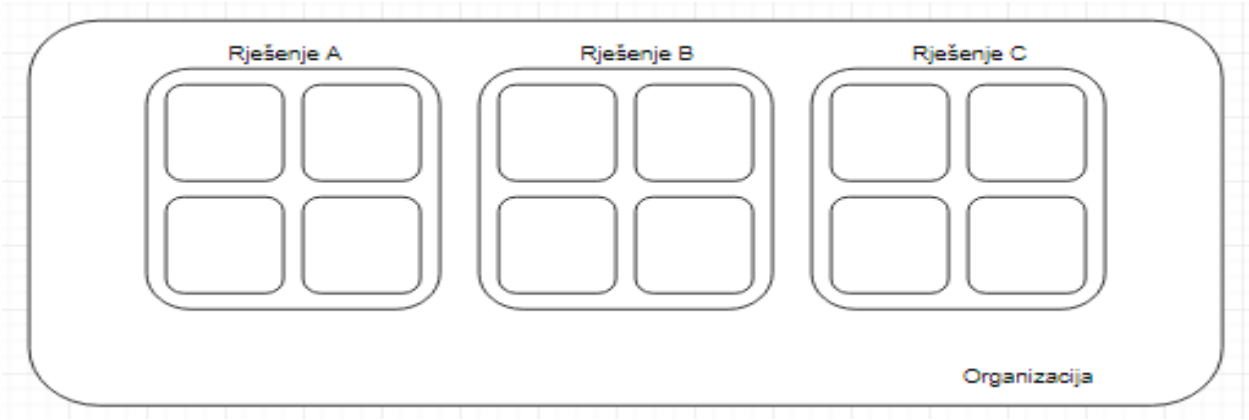
Na slici iznad vidljiv je način na koji se radi dekompozicija. Kao početni problem nalazi se veliki programski problem koji se kroz proces razdvajanja razdvaja na skup manjih problema koje je puno jednostavnije riješiti. Obrazloženje za ovakav način rješavanja problema je taj da se veći problem može lakše i učinkovitije riješiti kada se odvoji na manje dijelove. Svaka izgrađena jedinica logike rješenja postoji kao zasebno tijelo logike odgovorno za rješavanje jedne ili više manjih identificiranih problema.

Uzorak predstavlja početnu točku procesa koji započinje funkcionalnim razdvajanjem, a zatim nastavlja oblikovanjem razdvojene logike u servise prema drugim uzorcima koji se koriste. [2, str 301]

7.5.3. Servisna enkapsulacija

Skup sličnih programa koji predstavljaju veće dekomponirano tijelo logike rješenja može i dalje postojati unutar granica aplikacije. Mnogi stariji distribuirani sustavi izgrađeni su na taj način. Odluka o podijeli logike rješenja na manje jedinica najčešće je uzrokovana sljedećim razmatranjima [2, str 305]:

- povećanje skalabilnosti odvajanjem dijelova sustava koji su konkurentni,
- poboljšanje sigurnosti izoliranjem određenih dijelova sustava s posebnim zahtjevima pristupa i privatnosti,
- povećanje pouzdanosti distribucijom kritičnih dijelova sustava na više fizičkih poslužitelja,
- postizanje nominalne ponovne upotrebe unutar granica sustava ili unutar ograničenog dijela poduzeća.



Slika 13: Poduzeće koje se sastoji od distribuiranih rješenja
(Izvor: izrada autora prema [2, str 306])

Na slici se može vidjeti prikaz distribuiranog poduzeća. Organizacija ima niz rješenja koja upotrebljava te su manja rješenja spojena u veća ovisno o funkcionalnostima koje obavljaju i ovisno o problemu koji rješavaju.

Erl [2, str 301] navodi da postoje razni primjeri kada se poduzeće sastoji od više distribuiranih rješenja i tu se je potrebno suočiti s mnogim izazovima u pogledu dizajna i upravljanja, kao što su [2, str 301]:

- značajne količine redundacije,
- neučinkovita isporuka aplikacija,
- natrpana, prevelika tehnička okruženja,
- složena infrastruktura i isprepletena arhitektura poduzeća,

- složena i skupa integracija,
- uvijek rastući IT troškovi.

Da bi se izbjegli takvi problemi logika rješenja koristi se kao resurs poduzeća. Takav resurs može se enkapsulirati i izložiti kao servis. To znači da logika može činiti osnovu za novi servis ili se ta logika može zaokružiti unutar već postojećeg servisa, a najvjerojatnije je to onda nova funkcionalnost.

7.5.4. Uzorci definicije servisa

Identifikacija logike pogodne za enkapsulaciju servisa, grupiranje i distribuciju u različitim funkcionalnim kontekstima uspostavljaju temeljne granice servisa. Te granice postaju sve važnije kako se popis servisa prikuplja i primjenjuju uzorci. Primjer je normalizacija servisa pomoću koje se izbjegava funkcionalno preklapanje. Za definiranje najprikladnije granice za servis potrebno je uspostaviti najprikladniji funkcionalni kontekst. To određuje koja funkcionalnost pripada i izvan servisne granice. Sljedeći skup uzoraka pomaže u određivanju pružanjem kriterija za to da li se logika servisa treba smatrati agnostičkom ili neagnostičkom. [2, str 311]

7.5.5. Agnostički kontekst

Višenamjenska logika koja je grupirana zajedno s jednonamjenskom logikom kao rezultat isporučuje servise s malim ili nikakvim potencijalom ponovne upotrebe, a unosi redundantnost u poduzeće. Da bi se taj problem mogao izbjeći potrebno je izolirati logiku koja nije specifična za jednu svrhu u zasebne servise s izrazitim agnostičkim kontekstom. Upotrebom ovog uzorka višekratna logika rješenja diže se na razinu poduzeća što potencijalno uzrokuje povećanje složenosti dizajna i probleme upravljanja poduzećem. [2, str 313]

7.5.6. Neagnostički kontekst

Neagnostička logika koja nije servisno orijentirana može ugroziti učinkovitost sustava servisa koji koriste agnostički servisi. Ovaj problem može se riješiti pozicioniranjem logike servisnog rješenja prikladnog za enkapsulaciju pozicioniranjem unutar servisa u kojem su službeni članovi servisnog inventara. Ne očekuje se da će upotrebom ovog uzorka biti pružen potencijal za ponovnu upotrebu, ali su i dalje neagnostički servisi neovisno o svemu podložni pravilima servisne orijentacije.[2, str 319-320]

7.6. Uzorci implementacije servisa

Svaki od sljedećih uzoraka dizajna na neki način mijenja fizičku strukturu servisa. Većina uzoraka smatra se specijaliziranim što znači da se trebaju koristiti u za to određenim slučajevima. Neki uzorci koriste se vrlo rijetko, a gotovo nikada se ne koriste svi zajedno. Najvažniji uzorak koji će biti detaljnije opisan u nastavku je Service Facade iz razloga jer taj uzorak uvodi ključnu komponentu u arhitekturu servisa koja može pomoći u razvoju servisa. Uzorci koji spadaju pod uzorke za implementacije servisa su sljedeći [12]:

- servisna fasada (eng. *Service Facade*),
- redundantna implementacija (eng. *Redundant Implementation*),
- servisno umnožavanje podataka (eng. *Service Data Replication*),
- djelomično odlaganje stanja (eng. *Partial state Deferral*),
- djelomična validacija (eng. *Partial Validation*),
- UI Mediator.

7.6.1. Servisna fasada(eng. *Service Facade*)

Odabrani servis sadržava tijelo logike koje je odgovorno za obavljanje određenog posla. Kada je servis podložan promjenama zbog mogućih izmjena u ugovoru ili osnovne primjene, takva servisna logika može završiti proširena i prilagođena promjeni zbog koje je tražena. Uvođenje promjene najčešće uzrokuje razne izazove koji se tiču dizajna i vremena, a to je najbolje opisano u primjeru kroz nekoliko koraka [2, str 334]:

- postavljen je uvjet da servis mora podržavati više ugovora te se zbog toga uvodi nova logika odlučivanja i zahtijeva se novi način za obradu različitih vrsta ulaznih i izlaznih poruka,
- uzorci korištenja resursa koji se dijele, a kojima servis pristupa se mijenjaju što uzrokuje promjenu ponašanja servisa koje na kraju negativno utječe na postojeće korisnike,
- nadogradnja servisa je obavljena, a to uzrokuje promjenu osnovne poslovne logike kako bi se prilagodila novoj logici.

Logika servisne fasade umetnuta je u arhitekturu servisa radi uspostavljanja jednog ili više slojeva apstrakcije. Ti slojevi kreirani su tako da mogu prihvatiti buduće promjene ugovora o servisu zajedno s implementacijama koje se nadovezuju na osnovnu servisnu implementaciju. Erl [2, str 334] navodi da logika servisne fasade čini dio cjelokupne logike servisa, ali da se razlikuje od osnovne logike tako da se kod osnovne logike očekuje pružanje raspona funkcija odgovornih za provođenje funkcionalnosti koje su navedene u

ugovoru o servisu dok logika fasade kao primarnu odgovornost ima pružanje dopunske logike koja podržava osnovnu logiku servisa.

7.7. Uzorci zaštite servisa (eng. Service Security Patterns)

Servisno orijentirana rješenja obično se sastoje od više međusobno spojenih servisa. Svaki pokretni dio unutar arhitekture može postati potencijalna meta za narušavanje sigurnosti, a ako se gleda sa servisne strane taj pokretni dio bila bi svaka razmjena podataka između servisa i potrošača. Zbog toga servisne arhitekture često moraju biti opremljene dodatnim kontrolama koje im omogućuju da izdrže klasične oblike napada. Kao minimalni set uzoraka navedeni su idući [2, str 376-397]:

- zaštita od iznimaka (eng. *Exception Shielding*) – osigurava sigurnost podataka o pogrešci ili iznimci generiranoj od strane servise tako da se ti podaci filtriraju i šalju potrošačima na siguran način,
- pregled poruka (eng. *Message Screening*) –provjerava ulazne podatke i poruke u kojima traži potencijalno štetan sadržaj,
- pouzdan podsustav (eng. *Trusted Subsystem*) – koristi mehanizme uz pomoć kojih se onemogućuje potrošačima da izravno pristupe servisnim resursima sa svojim korisničkim podacima,
- čuvar servisnog perimetra (eng. *Service Perimeter Guard*) – uvodi novu vrstu uslužnog servisa koji u ime internih servisa obavlja zajedničke sigurnosne funkcije za vanjske korisnike

7.8. Uzorci projektiranja servisnog ugovora (eng. *Service Contract Design Patterns*)

Servisna orijentacija stavlja veliki naglasak na dizajn ugovora o servisima. Načelo dizajniranja standardiziranog ugovora o servisima zahtijeva da svi ugovori unutar inventara budu u skladu s istim pravilima i konvencijama. Kod dizajna ugovora o servisima javljaju se sljedeći uzorci [2, str 400]:

- razdvojen ugovor (eng. *Decoupled Contract*) i centralizacijski ugovor(eng. *Contract Centralization*) – smatraju se ključnima za dizajn servisa jer kada se kombiniraju pozicioniraju ugovor kao neovisan, a opet središnji dio arhitekture servisa,
- konkurentni ugovor (eng. *Concurrent Contracts*), denormalizacija ugovora (eng. *Contract Denormalization*) i apstrakcija validacije (eng. *Validation Abstraction)* –

pružaju različite tehnike za rad s više vrsta potrošača, a mogu se primijeniti samostalno ili zajedno.

7.9. Uzorci zaštite interakcije servisa (eng. *Service Interaction Security Patterns*)

Prilikom dizajniranja poslovnih rješenja koja moraju biti sigurna, a sastoje se od složenih servisa, dolazi do toga da se servisi podvrgavaju različitim scenarijima upotrebe od kojih svaki od njih može predstavljati potencijalni sigurnosni rizik ili zahtjev. Dizajn učinkovitih sustava zahtijeva da servisi budu pripremljeni za niz sigurnosnih izazova prilikom interakcije. Ti problemi rješavaju se pomoću idućih uzoraka [2, str 640]:

- povjerljivost podataka (eng. *Data Confidentiality*),
- provjera izvornosti podataka (eng. *Data Origin Authentication*),
- izravna provjera autentičnosti (eng. *Direct Authentication*),
- posredovana provjera autentičnosti (eng. *Brokered Authentication*).

Svaki od četiri prethodno navedenih uzoraka dizajna na svoj način rješava pojedine sigurnosne zahtjeve.

7.9.1. Povjerljivost podataka (eng. *Data Confidentiality*)

Podaci koji su sadržani unutar poruke mogu se kretati nesigurnim mrežama, a to može biti kretanje unutar mreže organizacije pa sve do kretanja preko javnih mreža. Uobičajeni pristup zaštite poruke na transportnom sloju je upotreba šifrirane veze između usluge i njegovog potrošača pomoću tehnologija kao što su TLS i SSL. Te tehnologije pružaju zaštitu podataka od točke do točke tako da sakrivaju podatke od potencijalnog prisluškivača između dvije točke u mreži. Sigurnost transportnog sloja učinkovita je po pitanju razmjene podataka jer su servisi i korisnici tog servisa u većini slučajeva unaprijed definirani, a rizik od neželjene izloženosti osjetljivim podacima putem nepovjerljivih posrednika obično je nizak. No tu postoje razne druge prijetnje [2, str 640]:

- agenti i usluge koje posjeduju podatke mogu ih vidjeti jer podaci koji se nalaze u poruci nisu kriptirani,
- osjetljivi podaci mogu biti ranjivi dok se nalaze u redu čekanja, bazi podataka ili datoteci, a programi za prisluškivanje koji se nalaze duž mreže mogu pristupiti tim podacima u slučaju kada poruke izađu iz sigurne mreže na javnu mrežu.

Da bi se u potpunosti zaštitio sadržaj poruke, primjenjuju se tehnologije šifriranja sloja poruke tako da je sigurnost podataka ugrađena u poruku i ostaje s njom te je dostupna samo

ovlaštenim primateljima. Postoje dvije vrste kriptografije koje pružaju povjerljivost podataka, a to su simetrična i asimetrična kriptografija. Oba postupka imaju sličan način rada, no svaki od njih ima svoje jedinstvene karakteristike. Kada je u pitanju simetrična kriptografija pošiljatelj i primatelj dijele zajednički ključ za šifriranje i dešifriranje. Cijeli princip simetričnog algoritma šifriranja oslanja se na zajednički tajni ključ i asimetrični algoritam šifriranja koji transformira podatke. Kod asimetrične kriptografije pošiljatelj šifrira podatke jednim ključem, a primatelj koristi drugi ključ za dešifriranje teksta. Šifriranje i odgovarajući ključ za dešifriranje često se nazivaju parom javnih/privatnih ključeva. U slučajevima kada se između servisa i korisnika događa više razmjena poruka prva razmjena poruka može biti asimetrično kriptirana i služiti za razmjenu nove zajedničke tajne, dok se daljnja komunikacija provodi simetrično. [13]

7.9.2. Posredovana provjera autentičnosti

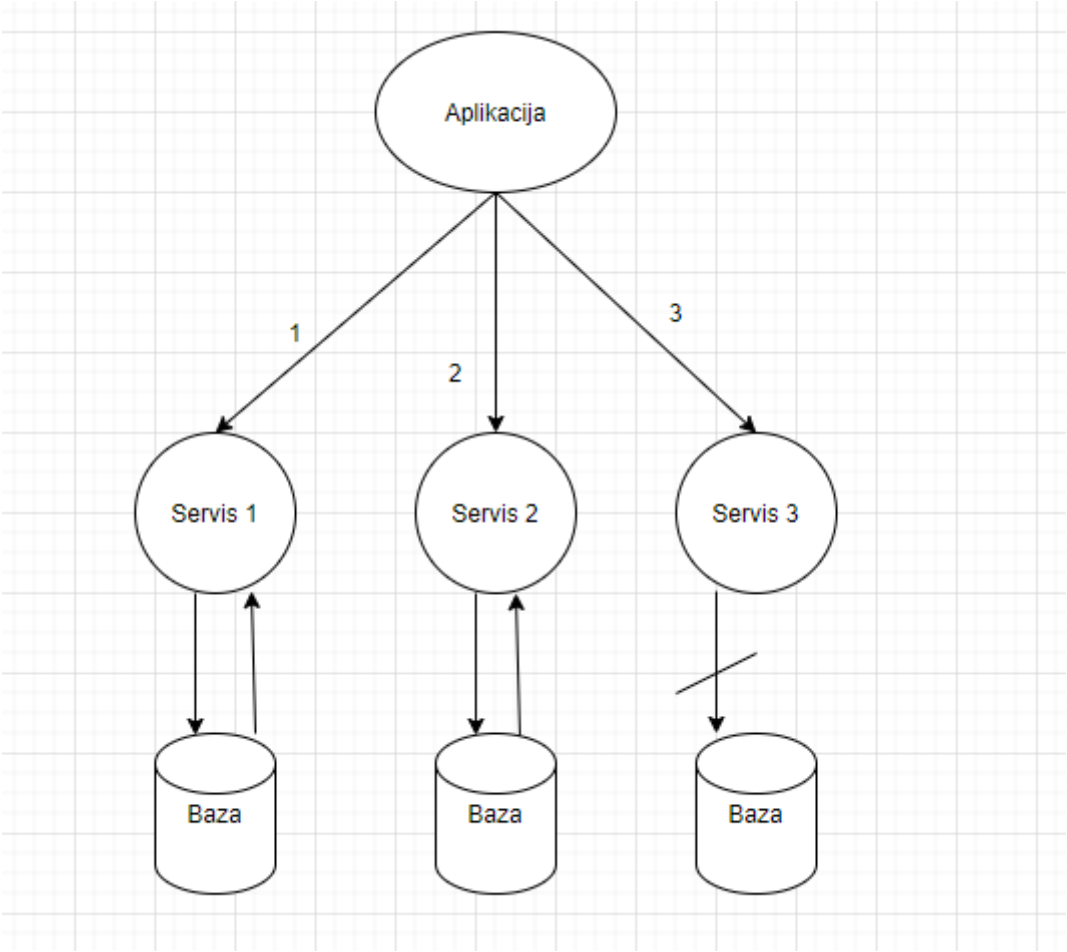
Servisi često imaju širok skup korisnika od kojih mnogi nisu poznati servisu. Uspostavljanje provjerene i sigurne komunikacije između korisnika i servisa često zahtijeva opsežnu komunikaciju koja može ometati dinamičku interakciju korisnika i servisa. Ako potrošač ima potrebu pristupati više različitih servisnih inventara koji imaju implementiranu zaštitu, javlja se problem da korisnik mora negdje spremiti podatke, a to može izazvati dodatne sigurnosne prijetnje. [2, str 661]

Broker se uvodi kao dodatno arhitekturno proširenje za provjeru autentičnosti iz razloga jer je sposoban provjeriti podatke korisnika bez potrebe da korisnici imaju izravne veze s uslugama kojima trebaju pristupiti. Korisnici i servisi vjeruju posredniku koji radi provjeru autentičnosti, a to brokeru omogućuje da postane centralizirani mehanizam za provjeru autentičnosti podataka u cjelokupnoj arhitekturi inventara. Takav sustav može se implementirati tako da se korisnik predstavi posredniku i potvrdi svoj identitet, a potom zauzvrat dobije žeton (eng. *Token*) koji korisnik koristi za daljnju komunikaciju s ostalim servisima. Žeton je dovoljan da utvrdi identitet korisnika jer su u njemu zapisani svi potrebni podaci koji ga identificiraju. Korisniku je omogućeno da više puta neprekidano komunicira sa servisom koristeći isti žeton koji je izdao posrednik. [2, str 662-665]

7.10. Transakcija atomarne usluge (eng. Atomic Service Transaction)

Aktivnosti koje se moraju izvršavati kroz nekolicinu servisa ponekad se ne izvrše kako treba ili se uopće ne izvrše. Ako dođe do pogreške u procesu vrlo je vjerojatno da će se ugroziti integritet rješenja i arhitekture. Da bi se takvi događaji izbjegli potrebno je servise

umotati u transakciju sa opcijom vraćanja prvobitnog stanja ukoliko se sve aktivnosti unutar transakcije ne izvedu kako treba. Da bi se bolje moglo razumjeti kako taj uzorak funkcionira najbolje je pogledati sliku ispod. [2, str 623]



Slika 14: Primjer transakcije atomarne usluge

(Izvor: izrada autora prema [2, str 624])

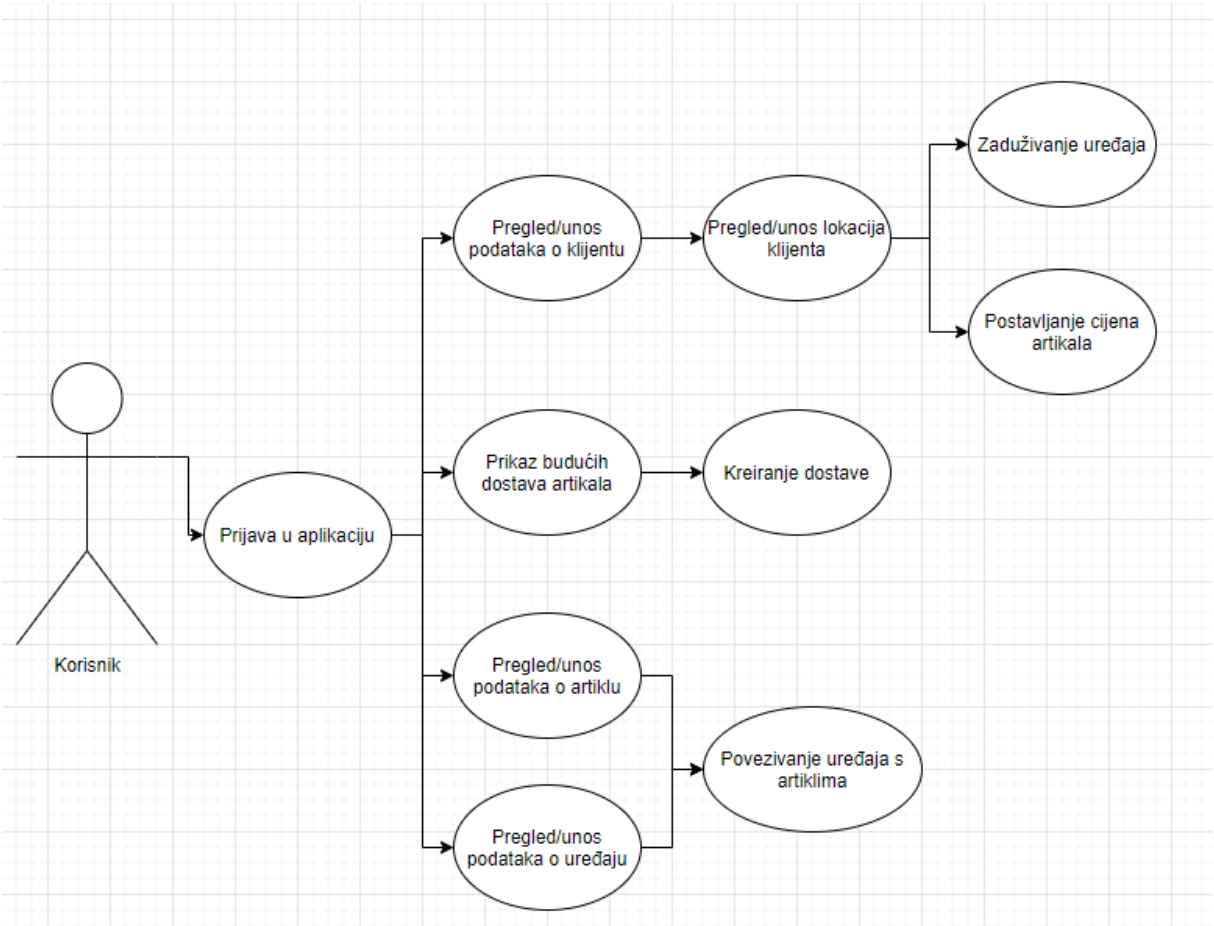
Na slici iznad može se vidjeti da na samom vrhu nalazi aplikacija koja šalje zahtjeve numerirane pod 1,2 i 3. Prikazani servisi na slici obuhvaćeni su u jednu transakciju, a da transakciju bude uspješna, potrebno je da se izvrše aktivnosti na svim servisima koji se nalaze u toj transakciji. Servis 1 i Servis 2 uspješno su završili svoje aktivnosti i podaci su uspješno upisani u bazu. Kod poziva trećeg servisa došlo je do nekakve pogreške i podaci nisu upisani u bazu. U ovom slučaju narušen je integritet podataka u prve dvije baze podataka i potrebno je u prve dvije baze vratiti početno stanje u kojima su se nalazile prije početka transakcije.

8. Izrada aplikacije – primjena SOA uzoraka

Za potrebe demonstriranja upotrebe SOA uzoraka kreirana je aplikacija pod imenom „Robna Kontrola“. Korisnički dio aplikacije sastoji se od nekolicine ekrana koji služe za rad s podacima. Glavna zadaća aplikacije je da prvo omogući unos svih uređaja, a potom unos artikala. Nakon toga potrebno je povezati artikle s uređajima kako bi se točno znalo koji artikl ide u koji uređaj. Nakon što se unesu svi uređaji i artikli mogu se dodavati kupci koji koriste usluge koje poduzeće nudi. Svakom od dodanih kupaca moguće je dodati lokacije kojima dodijelimo uređaje. Kada se unesu svi potrebni podaci na stranici dostava mogu se kreirati dostave koje olakšavaju praćenje budućih dostava i njihovu distribuciju.

8.1. Dijagram korištenja aplikacije

U svrhu boljeg razumijevanja rada aplikacije u nastavku je prikazan dijagram slučaja korištenja aplikacije „Robna Kontrola“.



Slika 15: Dijagram korištenja aplikacije "Robna Kontrola"

(Izvor: izrada autora)

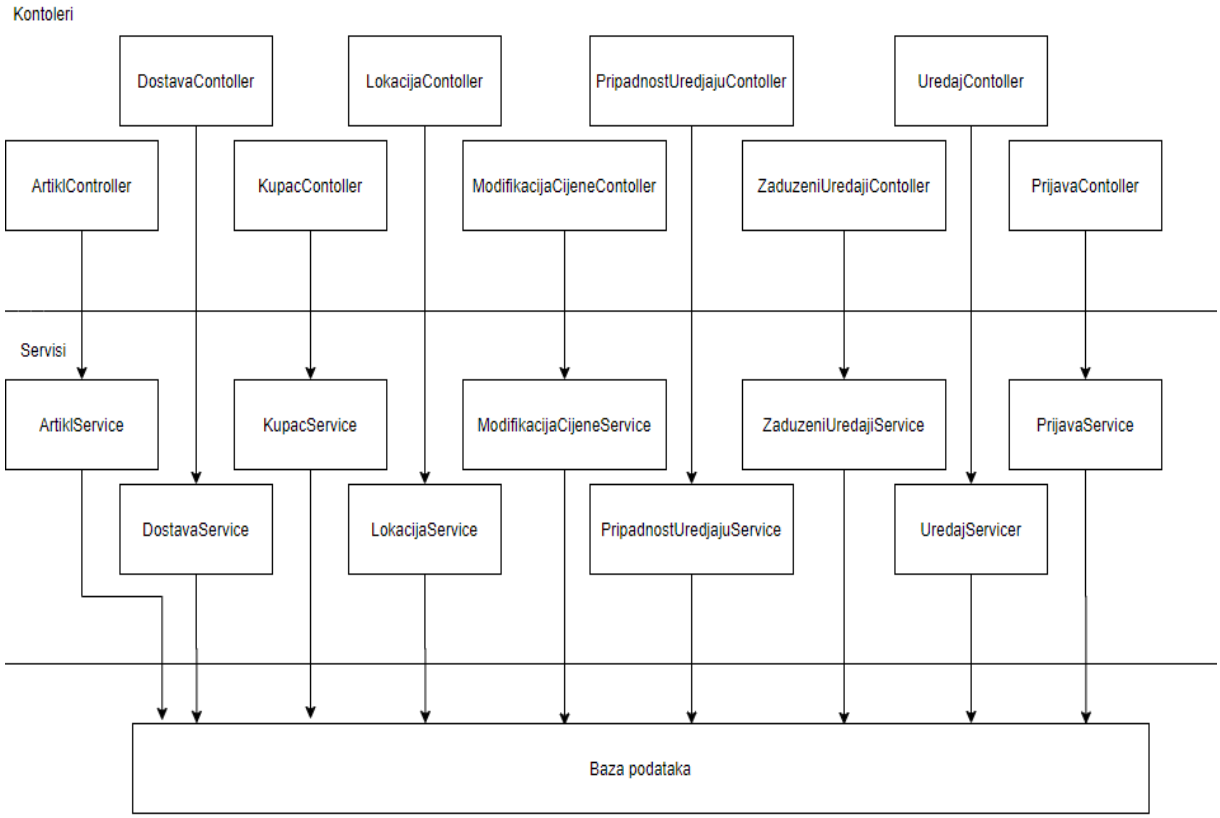
Na samom početku korištenja aplikacije potrebno se prijaviti. Nakon što se izvrši prijava u aplikaciju, korisnik može odabrati jednu od četiri glavne akcije koje su ponuđene u izborniku. Svaka od te četiri akcije ima svoju zadaću i obavlja određenu funkciju. Akcije će biti objašnjene na način kao što je prikazano na slici od vrha prema dnu slike. Prva akcija je omogućava pregled i unos podataka o klijentu. Nakon samog unosa podataka o klijentu moguće je odabrati iduću akciju, a to je unos lokacija klijenata. Klijent nije ograničen na samo jedan objekt i samo jednu lokaciju te je iz tog razloga omogućen unos lokacije u kojoj se nalaze svi potrebni podaci koji će se koristiti u aplikaciji. Svaka lokacija koristi određene uređaje o kojima je potrebno voditi evidenciju te se za svaku lokaciju unosi novi set podataka o uređajima. Uz opciju zaduživanja uređaja tu postoji i opcija u kojoj se može za svaku lokaciju definirati cijena artikla koji se koristi u određenom uređaju. Druga akcija uključuje prikaz dostava koje je potrebno realizirati u razdoblju koje je određeno ciklusima. Ciklusi se ponavljaju svakih deset dana. Za svaku ponuđenu lokaciju postoji mogućnost kreiranja nove dostave. Kod kreiranja nove dostave prikazan je popis uređaja koji se nalaze na lokaciji i popis artikala koji su vezani uz uređaje koji se nalaze na toj lokaciji. Treća akcija omogućava unos artikala koji su u ponudi, a unose se podaci poput šifre, naziva, jedinične mjere i cijene. Četvrta opcija je vezana uz unos i pregled uređaja koji se nalaze u ponudi. Nakon unosa uređaja i artikala moguće je raditi povezivanje artikala s uređajima jer svaki uređaj ne koristi svaku vrstu artikla.

8.2. Realizacija implementacije aplikacije

Da bi se moglo krenuti u realizaciju implementacije aplikacije potrebno je upoznati se s poslovnim problemom. Upoznavanjem s poslovnim problemom dobije se bolja slika toga što se točno treba napraviti i na koji način bi se to moglo realizirati. U samom startu ova aplikacija je gledana kao jedan veliki problem. Taj problem je bilo potrebno konkretizirati i napraviti model koji će odgovarati zadanom poslovnom problemu. Za te potrebe prvo je napravljena baza podataka koja je osnovni model koji govori o tome koji atributi će se koristiti.

8.3. Arhitektura poslužiteljske aplikacije

Poslužiteljski dio aplikacije izveden je koristeći Spring razvojni okvir i napisan je u programskom jeziku Java. Spring je odabran za korištenje iz razloga jer ima odličnu podršku za kreiranje REST servisa uz mnoštvo opcija koje omogućavaju jednostavnu konfiguraciju aplikacije. Aplikacija ima slojevitu arhitekturu koja se sastoji od dva sloja. Na slici u nastavku prikazana je arhitektura aplikacije.



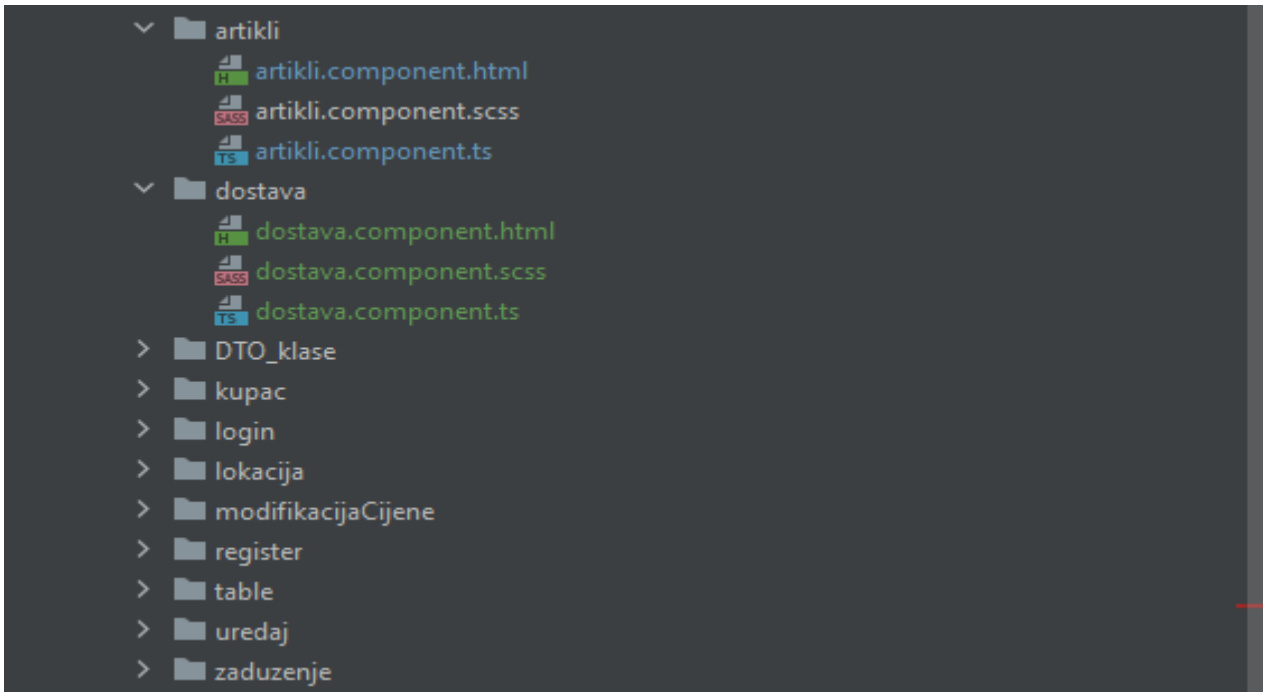
Slika 16: Arhitektura Spring aplikacije

(Izvor: izrada autora)

U prvom sloju aplikacije nalaze se kontroleri (eng. *Controllers*) kojima je glavna zadaća da definiraju na kojoj putanji je dostupna određena metoda REST servisa. Uz to svaki kontroler ima definirane ulazne i izlazne parametre koji se koriste kod poziva pojedine metode. Svaki kontroler poziva niz metoda definiranih u idućem sloju. U drugom sloju aplikacije nalaze se servisi (eng. *Services*) u kojima se nalazi sva poslovna logika REST servisa. Unutar tih servisa zajedno s poslovnom logikom nalazi se i logika za komunikaciju s bazom.

8.4. Arhitektura korisničkog dijela aplikacije

Korisnički dio aplikacije realiziran je pomoću razvojnog okvira Angular. Angular se koristi iz razloga jer se sastoji od niza „node_module“ klasa koje je moguće preuzeti putem interneta i koristiti unutar projekta. Pomoću tih modula razvojnim inženjerima se omogućuje korištenje predložaka koji omogućuju jednostavniju realizaciju u rješavanju problema koji je potrebno riješiti. Još jedan razlog iz kojeg je Angular odabran kao razvojni okvir korisničkog dijela je njegova modularnost. Modularnost znači da se aplikacija sastoji od niza komponenata koje mogu raditi neovisno jedna od druge ako su tako implementirane. U nastavku se mogu vidjeti komponente koje su korištene unutar aplikacije.



Slika 17 Prikaz komponenata korištenih u aplikaciji

(Izvor : izrada autora)

Sve komponente osim DTO_klase koje služe za razmjenu podataka imaju jednaku strukturu kao artikli i dostava. Svaka komponenta sastoji se od html, scss i ts dijela. Za primjer možemo uzeti komponentu artikli. Komponenta artikli sastoji se od artikli.component.html i artikli.component.scss datoteka koje objedinjuju ukupni dizajn i izgled ekrana za komponentu koja služi za rad s artiklima. U datoteci artikli.component.ts nalazi se logika napisana u Typescript-u koja omogućuje upravljanje s podacima i akcijama koje se odvijaju unutar ove komponente. Svaka od prikazanih komponenti u pravilu se sastoji od tri spomenuta dijela, ali tu postoje i izuzeci i ne mora svaka komponenta imati sva tri dokumenta da bi ona bila nazvana komponentom. Većina komponenti navedenih na slici iznad imaju svoju adresu preko koje im se može pristupiti. Prikaz adresa za pristup komponentama prikazan je u obliku koda koji se nalazi u datoteci app-routing-module.ts.

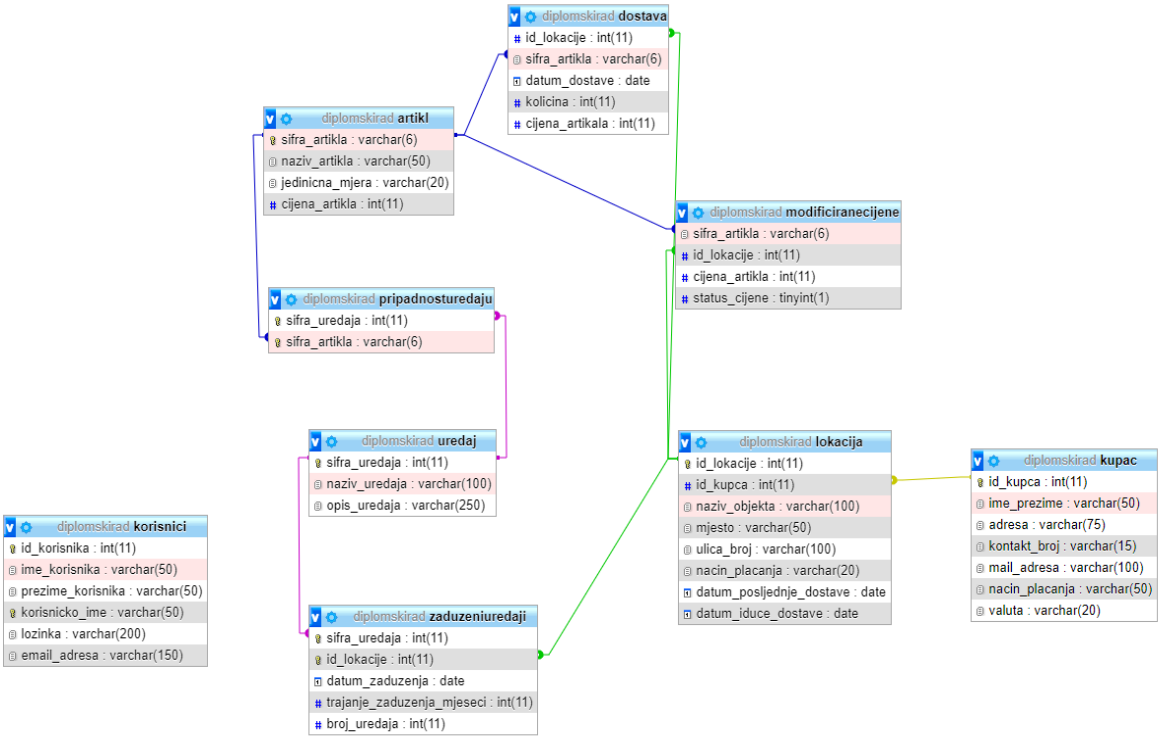
```
const appRoutes: Routes = [
  { path: '', component: KupacComponent, canActivate: [AuthGuard] },
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent },
  { path: 'artikli', component: ArtikliComponent, canActivate: [AuthGuard]
},
  { path: 'lokacija', component: LokacijaComponent, canActivate:
[AuthGuard] },
  { path: 'uredaj', component: UredajComponent, canActivate: [AuthGuard] },
```

```
  { path: 'zaduzenje', component: ZaduzenjeComponent, canActivate: [AuthGuard] },
  { path: 'modifikacija-cijene', component: ModifikacijaCijeneComponent, canActivate: [AuthGuard] },
  { path: 'dostava', component: DostavaComponent, canActivate: [AuthGuard] },
  // otherwise redirect to home
  { path: '**', redirectTo: '' }
];
```

U kodu iznad može se vidjeti da svaka komponenta ima definiranu svoju putanju preko koje se može otvoriti. Isto tako može se vidjeti da jedine komponente koje nisu zaštićene su login i register jer njima mora biti omogućen direktni pristup jer su to komponente koje se koriste za prijavu i registraciju. Sve ostale komponente štite se preko AuthGuard-a koji provjerava postoji li prijavljeni korisnik. Ako postoji prijavljeni korisnik on se pušta na željenu putanju, a ako ne postoji onda se vraća na početnu stranicu za prijavu.

8.4.1. Baza podataka

Za bolje razumijevanja načina na koji se spremaju podaci unutar aplikacije i načina na koju su oni međusobno povezani na slici u nastavku vidljiv je prikaz baze.



Slika 18: Prikaz baze aplikacije

(Izvor: izrada autora)

Slika 17 prikazuje dizajn baze podataka koji se koristiti u aplikaciji. U cilju boljeg razumijevanja prikazane baze napravljeno je objašnjenje svake pojedine tablice. Tablica korisnici nije ni na koji način povezana uz samu aplikaciju. Pošto u samoj aplikaciji nema nikakvih ograničenja po pitanju pristupa podacima od strane prijavljenih korisnika, nema nikakve povezanosti s drugim tablicama. Drugi razlog zašto tablica korisnici nije povezana s ostatkom baze je taj da korisnici mogu biti neovisni o aplikaciji i u aplikaciju se mogu prijaviti svi korisnici koji su pohranjeni u navedenoj tablici. Tablica kupac je bazna tablica u kojoj se nalaze osnovni podaci o kupcu te je ona povezana s tablicom lokacija. U tablici lokacija nalaze se osnovni podaci potrebni za identifikaciju lokacije, uz to ovdje se nalazi ograničenje da lokacija može pripadati samo jednom kupcu, ali kupac može imati više lokacija. Tu se dolazi do tablice uređaja u koju se spremaju osnovni podaci o uređajima, ista takva tablica je i tablica artikl u kojoj se nalaze osnovni podaci o artiklu. Ostale tablice služe za povezivanje

tih triju osnovnih tablica. Tablica zaduzeniuredaji je povezna tablica za tablicu uredaj i lokacija. Pošto svaka lokacija može imati više uređaja i svaka vrsta uređaja može biti zadužena na više lokacija, potrebna je evidencija koji uređaj se nalazi na kojoj lokaciji uz neke dodatne opisne atribute. Da bi se mogla realizirati funkcionalnost povezivanja uređaja s artiklima tu je napravljena tablica pripadnosturedaju. Specifična funkcionalnost je ta da svaki artikl ima svoju baznu cijenu no za svaku lokaciju može se postaviti unikatna cijena, uz to je potrebno voditi evidenciju svih cijena pojedinog artikla za pojedinu lokaciju. Za realizaciju posljednje funkcionalnosti prikaza i kreiranja dostava napravljena je tablica dostava. Svaka dostava sadrži više artikala koji trebaju biti dostavljeni.

8.5. Primjena uzoraka

Nakon kreiranja baze podataka kreće se u realizaciju izgradnje aplikacije. Da bi se moglo krenuti s izgradnjom aplikacije potrebno je odabrati SOA uzorke koji najbolje odgovaraju postavljenim zahtjevima. U nastavku je opisano na koji način se koristio koji uzorak. Neki od uzoraka su poslovno orijentirani i zahtijevaju kreiranje dokumentacije, dok ostali imaju primjere realizacije u programskom kodu aplikacije.

8.5.1. Funkcijska dekompozicija

Prilikom kreiranja ove aplikacije postavljeni zahtjevi predstavljaju jedan veliki problem. Tako složeni problem potrebno je podijeliti na manje probleme. U ovom slučaju radi se dekompozicija na niz funkcija koje trebaju biti implementirane. U nastavku se nalazi popis razdijeljenih funkcionalnosti:

- registracija i prijava u aplikaciju,
- prikaz, kreiranje, uređivanje i brisanje kupaca,
- prikaz, kreiranje, uređivanje i brisanje artikala,
- prikaz, kreiranje, uređivanje i brisanje uređaja,
- povezivanje uređaja s artiklima,
- prikaz, kreiranje, uređivanje i brisanje lokacija,
- pridruživanje lokacije kupcu,
- zaduživanje uređaja po lokacijama,
- postavljanje cijena artikala,
- postavljanje cijena artikala ovisno o lokaciji,
- prikaz i kreiranje dostava.

8.5.2. Temeljni uzorci inventara

Iz temeljnih uzoraka inventara odabrani su:

- kanonski protokol,
- normalizacija servisa.

Kanonski protokol odabran je iz razloga da svi servisi koji su implementirani mogu međusobno komunicirati. Svi servisi napravljeni su na istoj verziji i koriste iste protokole za razmjenu podataka, a to je Http protokol.

Normalizacija servisa odabrana je iz razloga da ne dođe do redundantne logike. Da bi se normalizacija servisa mogla provesti potrebno je dizajnirati funkcionalnu granicu pojedinog servisa. U nastavku se nalazi napravljena funkcionalna granica za svaki pojedini servis.

1. ArtiklService
    - ograničen na rad s artiklima (dohvat, kreiranje, izmjena i brisanje artikla),
2. KupacService
    - ograničen na rad s kupcima (dohvat, kreiranje, izmjena i brisanje kupaca),
3. LokacijaService
    - ograničen na rad s lokacijama (dohvat, kreiranje, izmjena i brisanje lokacija),
4. UređajService
    - ograničen na rad s uređajima (dohvat, kreiranje, izmjena i brisanje uređaja),
5. PripadnostUredajuService
    - ograničen na kreiranje ovisnosti između artikla i uređaja i dohvat tih ovisnosti,
6. ZaduzeniUredajiService
    - ograničen na rad za kreiranje ovisnosti između uređaja i lokacije, izmjenu i dohvat,
7. ModifikacijaCijeneService
    - ograničen na rad za postavljanje cijene artikala za pojedinu lokaciju, izmjenu i dohvat,
8. DostavaService
    - ograničen na rad za kreiranje dostava za pojedine lokacije i dohvat,
9. LoginService
    - ograničen na rad za kreiranje novih korisnika i prijavu.

Ovaj inventar servisa napravljen je iz analize procesa koji se koriste u aplikaciji. Definiranjem funkcionalnih granica inventara servisa smanjuje se rizik kreiranja servisa koji će imate iste funkcionalnosti.

8.5.3. Zaštita od iznimaka

Idući uzorak koji je realiziran u aplikaciji je zaštita od iznimaka (eng. *Exception Shielding*). Kao što je prethodno opisano, ovaj uzorak se koristi kako korisnici koji koriste aplikaciju ili osobe koji žele doći do osjetljivih podataka ne bi preko iznimke mogle doći do tih podataka. Ovo je možda i jedan od nužnijih uzoraka koji bi morao biti implementiran u svakoj poslovnoj aplikaciji, pogotovo u aplikacijama koji rade s osjetljivim podacima. Sam uzorak implementiran je tako da se svaka iznimka koja je uhvaćena filtrira i tek onda proslijedi krajnjem korisniku. Za samu realizaciju tog uzorka prvo je bilo potrebno definirati vlastitu iznimku.

```
public class RobnaKontrolaException extends Exception {
    private Exception causedByException = null;
    private String message;
    private HttpStatus httpStatus;
}
```

U kodu iznad se može primijetiti da je napravljena vlastita iznimka koja u sebi sadržava određene podatke. Ti podaci su uzrok iznimke, poruka i http status. To je skup definiranih podataka koji se vraćaju krajnjem korisniku i opisuju mu problem.
No kreiranje nove iznimke nije bilo dovoljno. Jedna od opcija koje nudi Spring je i anotacija „ControllerAdvice“. Upotrebom ove notacije Spring dozvoljava pisanje globalnog koda koji može biti primijenjen na širok raspon servisa.

```
@ControllerAdvice
public class GlobalErrorHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(RobnaKontrolaException.class)
    public ResponseEntity<Object>
RobnaKontrolaException(RobnaKontrolaException exception, WebRequest
request) {
        return handleExceptionInternal(exception.getCausedByException(),
exception.getMessage(),
                new HttpHeaders(), exception.getHttpStatus(), request);
    }
}
```

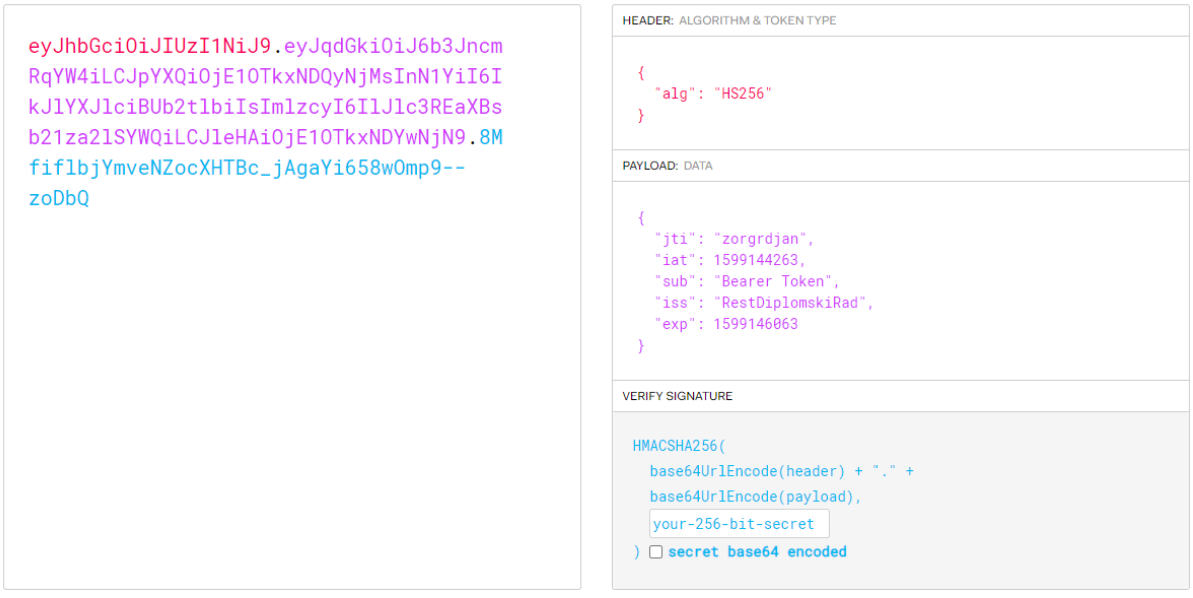
U kodu iznad prikazana je„GlobalErrorHandler“ klasa kojoj je glavni zadatak da uhvati svaku iznimku koja se dogodi u servisu i kreira odgovor u kojem se nalaze podaci o tome koja generička iznimka se dogodila, personalizirana poruka za korisnika i http status odgovora.
U svakom upravljaču je implementirano upravljanje iznimkama. Iznimkama je moguće upravljati na različitim razinama. Pošto je riječ o korisnicima koji će koristiti tu aplikaciju, sve iznimke koje se vračaju na korisničku stranu aplikacije napisane su u obliku poruke.

```
catch (SQLIntegrityConstraintViolationException sqlException) {
            throw new RobnaKontrolaException(sqlException, "Artikl
            vec postoji", HttpStatus.CONFLICT);}
catch (SQLException sqlException){
            throw new RobnaKontrolaException(sqlException, "Dogodila
            se pogreska kod kreiranja artikla",
            HttpStatus.BAD_REQUEST);}
```

U kodu koji je naveden iznad može se vidjeti način na koji se upravlja s iznimkama. U ovom primjeru može se vidjeti da ako se uhvati greška tipa „SQLIntegrityConstraintViolationException“ ispisuje jedan tip poruke, a ako se uhvati greška „SQLException“ ispisuje se druga poruka. Ovim načinom obuhvaćeni su svi servisi koji se pozivaju unutar poslužiteljskog dijela aplikacije.

8.5.4. Posredovana provjera autentičnosti

Idući upotrijebljeni uzorak je posredovana provjera autentičnosti. Budući da direktna identifikacija ponekad nije praktična, odabrana je posredovana provjera. Kod posredovane provjere postoji posrednik kojemu se šalju podaci za prijavu. Posrednik te podatke provjeri i korisniku vraća žeton kojim može pristupati servisu. U implementaciji ovog uzorka korištena je biblioteka „jsonwebtoken“ kod poslužiteljskog dijela aplikacije. Funkcioniranje posredovane provjere na primjeru jednog json web žetona prikazano je na slici u nastavku.



Slika 19: Prikaz kodiranog i dekodiranog web žetona

(Izvor: izrada autora)

Svaki žeton sastoji se od tri dijela. Prvi dio žetona je zaglavlje (eng. *Header*) u kojem je naveden algoritam i tip žetona. Srednji dio žetona na slici je ljubičaste boje i sadrži podatke o žetonu. U tim podacima nalaze se identifikator, vrijeme izdavanja žetona, za koga je izdan žeton, tko je izdao žeton i vrijeme isteka. U trećem dijelu nalazi se potpis. Važno je napomenuti da je svaki žeton moguće dekodirati i pročitati, ali ako se neki od podataka promijeni, potpis više neće odgovarati i poslužiteljska strana će odbiti zahtjev u kojem se nalazi izmijenjeni žeton. U aplikaciji je žeton implementiran tako da se kod prijave u aplikaciju servisu šalju podaci potrebni za prijavu. Ako ti podaci odgovaraju i nalaze se u tablici korisnika, korisniku se izdaju dva žetona. Prvi žeton služi za mogućnost korištenja servisa, dok se drugi žeton koristi za obnovu prvog žetona. Žeton za rad vrijedi 30 minuta, dok žeton za refresh vrijedi duže. Ako je korisnik neaktivan više od 30 minuta, žeton ističe i osoba se mora ponovno prijaviti.

```
public String createJWT(String id, String issuer, String subject, long
                  ttlMillis) {
        SignatureAlgorithm signatureAlgorithm = SignatureAlgorithm.HS256;
        long nowMillis = System.currentTimeMillis();
        Date now = new Date(nowMillis);
byte[]apiKeySecretBytes=DatatypeConverter.parseBase64Binary(SECRET_KEY);
Key signingKey= new SecretKeySpec(apiKeySecretBytes,
signatureAlgorithm.getJcaName());

JwtBuilder builder = Jwts.builder().setId(id)
.setIssuedAt(now)
.setSubject(subject)
.setIssuer(issuer)
.signWith(signatureAlgorithm, signingKey);

if (ttlMillis >= 0) {
long expMillis = nowMillis + ttlMillis;
Date exp = new Date(expMillis);
builder.setExpiration(exp);
}

return builder.compact();
    }
```

U kodu iznad može se vidjeti na koji način se kreira žeton. Prvo je potrebno odrediti koji će se algoritam koristiti za potpis. U ovom primjeru koristi se algoritam HS256. Dohvaća se trenutno vrijeme i kreira se ključ za potpis. Preko „JwtBuilder“ klase kreira se novi žeton, pridružuju mu se podaci i žeton se izdaje.

U nastavku se može vidjeti na koji je način izvedena implementacija korištenja žetona na klijentskoj strani aplikacije. Prilikom slanja svakog zahtjeva prema poslužiteljskoj strani aplikacije poruka koja se želi poslati se presreće i provjerava se sadržaj http zahtjeva. Prvo se radi provjera postoji li prijavljeni korisnik, ako postoji i ako taj korisnik posjeduje žeton, u

zahtjev se dodaje zaglavlje tipa „Authorization“ u kojem se nalazi žeton koji se dalje provjerava na poslužiteljskoj strani.

```
intercept(request: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
    const headers = new HttpHeaders({
      'Content-Type': 'application/json'
    });
    request = request.clone({
      setHeaders: {
        'Content-Type': `application/json`
      }
    });
    let currentUser = this.authenticationService.currentUserValue;
    if (currentUser && currentUser.token) {
      if ((request.urlWithParams.toString()).includes('refreshToken')) {
        return next.handle(request);
      }
      this.provjeraIstekaTokena(currentUser.token);
      request = request.clone({
        setHeaders: {
          Authorization: `Bearer ${currentUser.token}`
        }
      });
    }

  return next.handle(request);
 }
```

Servisi su kreirani tako da se prije samog ulaska u metodu provjerava postoji li zaglavlje tipa „Authorization“. Ako postoji zaglavlje tog tipa, iz zaglavlja se vadi vrijednost i šalje se na provjeru. Ako je žeton izdan od strane poslužitelja i nije istekao, ulazi se u sam servis koji radi poslovnu logiku. Ako je žeton istekao ili je neispravan, vraća se odgovor tipa „UNAUTHORIZED 401“, u tom slučaju korisnički dio aplikacije podešen je da se ukoliko je neko prijavljen izvrši odjava iz aplikacije.

```
@RequestMapping(value = "/korisnik", method = RequestMethod.POST)
    public String kreiranjeKorisnika(@RequestBody KorisnikDTO korisnik,
@RequestHeader("Authorization") String token) throws RobnaKontrolaException
{

        tokenService.provjeriHeader(token);
        String odgovor = kupacController.kreiranjeKorisnika(korisnik);
        return odgovor;
  }
```

Iz prethodno prikazanog koda može se vidjeti na koji način je zapravo implementirano korištenje i provjeravanje žetona koji je izdan klijentu. U pozivu svake metode prvo se provjerava postoji li u zahtjev u zaglavlju sa žetonom. Ako u zaglavlju postoji žeton, žeton se provjerava i ako je žeton pravilno potpisan, nastavlja se s izvođenjem ostatka metode, u

suprotnom se korisniku vraća odgovor „Token istekao“. U kodu prikazanom u nastavku može se vidjeti na koji način se izvodi provjera ispravnosti žetona.

```
public Claims decodeJWT(String jwt) {
        try {
  Claims claims = Jwts.parser()
.setSigningKey(DatatypeConverter.parseBase64Binary(SECRET_KEY))
                    .parseClaimsJws(jwt).getBody();
            return claims;
        } catch (Exception e) {
            System.out.println(e);
        }
        return null;
    }
```

Da bi se izbjegla automatska odjava iz aplikacije uslijed isteka važenja žetona na korisničkoj strani aplikacije implementirana je provjera isteka važenja žetona. Provjera se provodi na način da se usporedi sistemsko vrijeme i vrijeme isteka žetona. Ukoliko je vrijeme žetona isteklo poziva se metoda koja traži dohvat novog žetona.

```
  provjeraIstekaTokena(token: string): void {
    const datum: Date = new Date();
    const helper = new JwtHelperService();
    const vrijemeTrenutno: number = datum.getTime();
    const vrijemeIstekaTokena: number =
helper.getTokenExpirationDate(token).getTime();
    const razlikaVremena = vrijemeIstekaTokena - vrijemeTrenutno;
    const razlikaMinute = razlikaVremena / 60000;
    if (vrijemeTrenutno > vrijemeIstekaTokena) {

this.authenticationService.refreshToken(this.authenticationService.currentU
serValue.refreshToken);
    }
  }
```

8.5.5. Transakcija atomarne usluge

Idući korišteni uzorak bio je transakcija atomarne usluge (eng *Atomic service transaction*). Ako u sustavu postoji transakcija koja se mora obaviti i ta transakcija povezuje dva ili više poziva servisa ovaj uzorak je idealan za korištenje. Način na koji radi taj uzorak objašnjen je prethodno. Ovaj uzorak izabran je iz razloga jer je kreirana transakcija koja mora u potpunosti biti provedena, u slučaju da jedan servis ne uspije provesti proces, svi prethodni provedeni procesi vraćaju bazu na stanje koje je bilo prije same provedbe zahtjeva. U ovom slučaju transakcija je sljedeća. U pitanju su dva servisa, servis koji upravlja artiklima i servis koji upravlja modificiranjem cijene. Da bi se izbjeglo ponavljanje funkcionalnosti modificiranja cijene proizvoda, kreira se transakcija koja funkcionira na način opisan u nastavku. Korisnik dodaje novi artikl u aplikaciju i poziva servis za unos artikla. Tim pozivom započinje transakcija. Nakon kreiranja novog artikla, unutar servisa zove se drugi servis koji za sve postojeće lokacije dodaje artikl i cijenu kako bi se ona mogla modificirati.

Ako je taj dio unosa u bazu neuspješan, servis vraća prethodnom servisu poruku da transakcija nije uspjela i početni servis vraća bazu u stanje koje je bilo prije samog kreiranja artikla.

```
      tokenService.provjeriHeader(token);
        String odgovor = artiklController.kreiranjeArtikla(artikl, token);
        if(odgovor.contains("400"))
        { artiklController.brisanjeArtikla(artikl.getSifraArtikla());
        }
        return odgovor;
```

U prvom dijelu uzorka poziva se metoda za kreiranje artikla. Ako je nakon provedbe kreiranja artikla odgovor „400“ artikl se briše iz baze jer nešto vezano uz artikl nije bilo dobro kreirano u bazi i takav zapis artikla ne smije ostati zapisan. U nastavku je prikazan način kreiranja novog artikla unutar baze.

```
public String kreiranjeArtikla(Artikl artikl, String token) throws
RobnaKontrolaException {
        Connection con = BazaService.dohvatiConnection();
        try {
            con.setAutoCommit(false);
            String upit = "INSERT INTO `artikl`(`sifra_artikla`,
`naziv_artikla`, `jedinicna_mjera`,`cijena_artikla`) VALUES (?,?,?,?)";
            PreparedStatement preparedStmt = con.prepareStatement(upit);
            preparedStmt.setString(1, artikl.getSifraArtikla());
            preparedStmt.setString(2, artikl.getNazivArtikla());
            preparedStmt.setString(3, artikl.getJedinicnaMjera());
            preparedStmt.setInt(4, artikl.getCijenaArtikla());
            preparedStmt.execute();
            con.commit();
            if (!this.dodajCijenuArtikluPOST(artikl, token)) {
                HttpStatusDTO httpStatusDTO = new HttpStatusDTO("400",
"Greška, artikl nije kreiran");
                Gson json = new Gson();
                return json.toJson(httpStatusDTO);
            } else {
                HttpStatusDTO httpStatusDTO = new HttpStatusDTO("201",
"Artikl je uspjesno kreiran");
                Gson json = new Gson();
                return json.toJson(httpStatusDTO);
            }
        } catch (SQLIntegrityConstraintViolationException sqlException) {
            throw new RobnaKontrolaException(sqlException, "Artikl vec
postoji", HttpStatus.CONFLICT);
        } catch (SQLException sqlException){
            throw new RobnaKontrolaException(sqlException, "Dogodila se
pogreska kod kreiranja artikla", HttpStatus.BAD_REQUEST);
        }
        finally {
            BazaService.zatvoriConnection(con);
            BazaService.zatvoriConnection(preparedStmt);
            BazaService.zatvoriConnection(rs);
        }
    }
```

Nakon kreiranja novog artikla unutar baze potrebno je dodati cijenu tom artiklu za svakog korisnika koji se nalazi u bazi, taj dio je automatiziran i stavlja se bazna cijena artikla koju je moguće promijeniti. U nastavku se zatim poziva servis putem POST metode koji odrađuje modifikaciju cijena proizvoda. Ako ta modifikacija prođe uspješno, transakcija se time završava, a ako poziv servisa vrati negativan odgovor, artikl se briše iz baze i baza se vraća u početno stanje.

```
public boolean dodajCijenuArtikluPOST(Artikl artikl, String token) {
        try {
            URL urlForGetRequest = new URL(adresaServisa
                    + artikl.getSifraArtikla() + "/" +
artikl.getCijenaArtikla());
            String readLine = null;
            HttpURLConnection conection = (HttpURLConnection)
urlForGetRequest.openConnection();
            conection.setRequestMethod("POST");
            conection.setRequestProperty("Content-Type", "application/json;
utf-8");
            conection.setRequestProperty("Authorization", token);
            conection.setRequestProperty("Accept", "application/json");
            conection.setDoOutput(true);
            int responseCode = conection.getResponseCode();
            if (responseCode == HttpURLConnection.HTTP_OK) {
                BufferedReader in = new BufferedReader(
                        new InputStreamReader(conection.getInputStream()));
                StringBuffer response = new StringBuffer();
                while ((readLine = in.readLine()) != null) {
                    response.append(readLine);
                }
                in.close();
                if (response.toString().contains("false")) {
                    return false;
                } else {
                    return true;
                }
            } else {
                return false;
            }
        } catch (IOException exc) {
            return false;
        }
    }
```

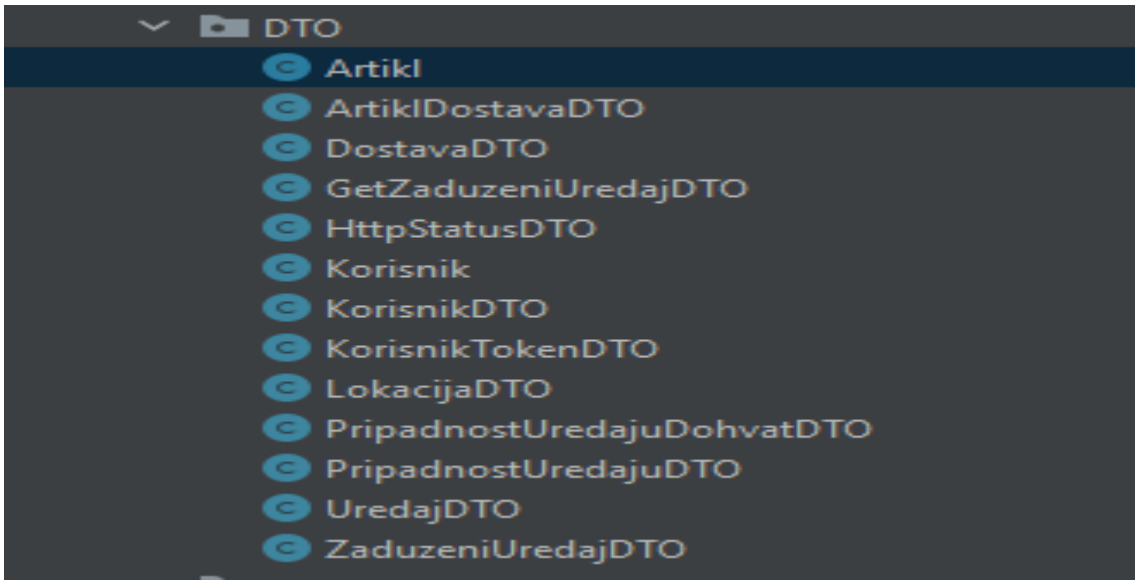
8.6. Primjena principa

Da bi se neki određeni uzorci mogli implementirati u izgradnji poslovnih aplikacija, potrebno je koristiti određene servisne principe. Neki od tih principa su prikazani i korišteni kod aplikacije „Robna Kontrola“.

8.6.1. Standardizirani servisni ugovor

Prvi princip koji je korišten je standardizirani servisni ugovor. Ovaj princip koristio se kako bi se svi servisi koji se nalaze unutar istog servisnog inventara jednako dizajnirali i ne bi bilo velikih razlika. U primjeru aplikacije ne postoji pisana dokumentacija u kojoj bi svaka usluga bila detaljno opisana. No standardi koji su bili primijenjeni kod svakog kreiranog servisa su bili idući:

- GET metoda korištena samo za dohvat podataka,
- POST metoda korištena samo za kreiranje podataka,
- PUT metoda korištena samo za izmjenu podataka,
- DELETE metoda korištena samo za brisanje podataka,
- POST metodom se uvijek šalju podaci u modelu,
- POST/PUT/DELETE metode vraćaju status i poruku,
- Servisi koriste standardizirane modele za razmjenu poruka.



Slika 20: Prikaz standardnih modela koji se koriste

(Izvor: izrada autora)

Na slici 19 mogu se vidjeti svi modeli koji su definirani u svim servisima koji se koriste za razmjenu poruka.

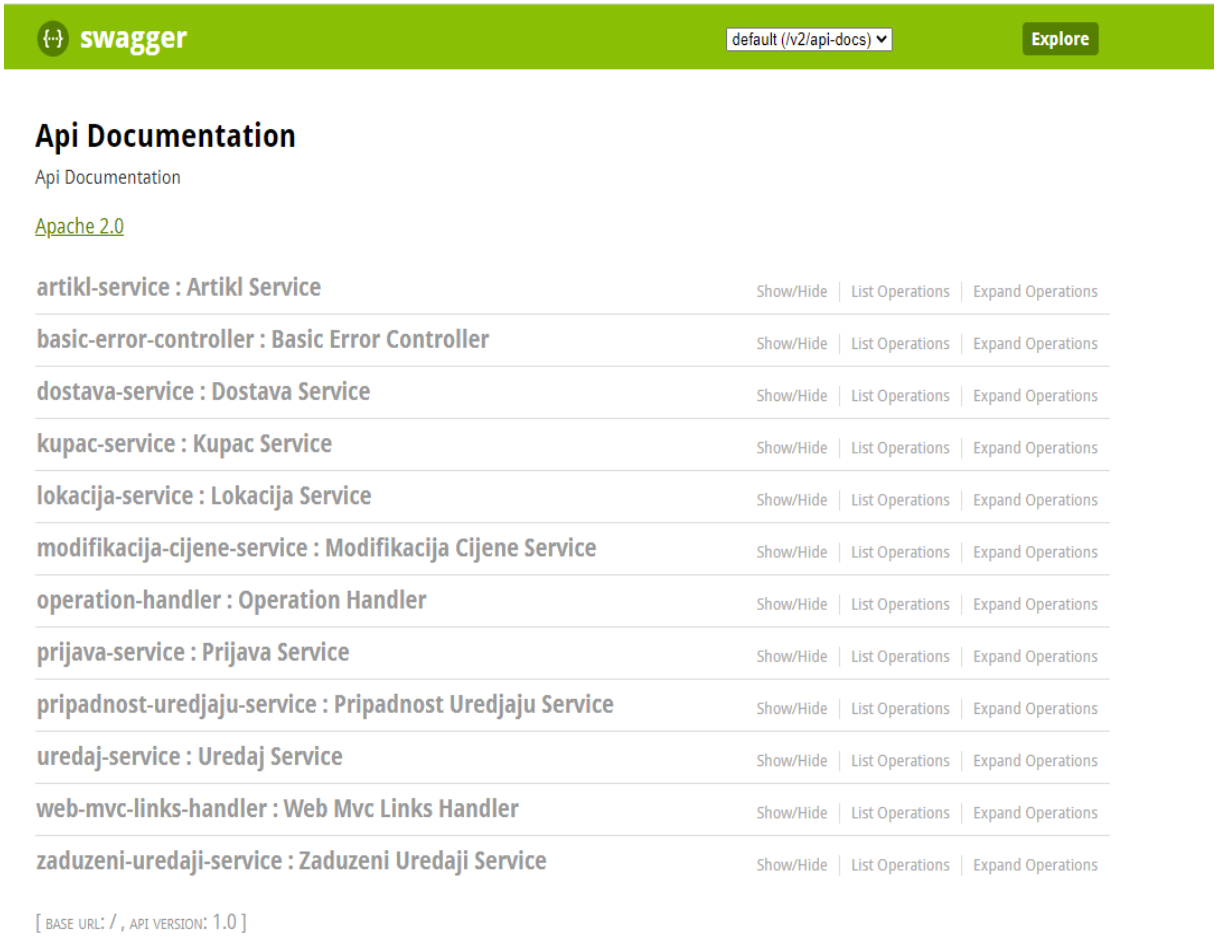
8.6.2. Servisno labavo povezivanje

Servisno labavo povezivanje je idući princip koji je korišten kod kreiranja servisa. Svi servisi koji su kreirani neovisni su jedan o drugom. Skup servisa koji radi s artiklom je

neovisan o servisu koji radi s kupcem ili bilo kojim drugim servisom. Logika koja se nalazi u pojedinom servisu nikako ne utječe na druge servise.

8.6.3. Otkrivanje servisa

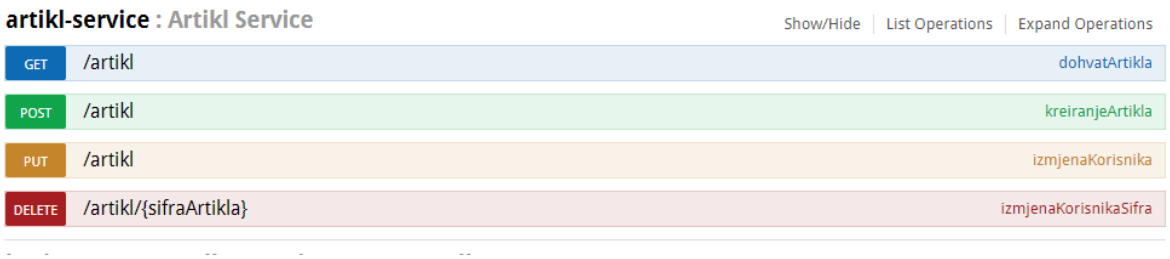
Princip otkrivanje servisa u ovom slučaju realiziran je malo drugačije nego se opisuje u primjerima. U ovom slučaju ne šalju se meta podaci koji se mogu otkriti i interpretirati već se koristi korisničko sučelje koje te podatke prikaže na strukturiran način. Korisničko sučelje koje se koristi u ovom primjeru je swagger. Swagger je razvojni okvir za opisivanje servisa, a odabran je iz razloga jer je vrlo jednostavan za korištenje i razumljiv je razvojnim inženjerima i testerima. Nakon pokretanja servera i poslužiteljskog dijela aplikacije swagger je dostupan na adresi : http://localhost:8080/swagger-ui.html#/.



Slika 21: Prikaz korisničkog sučelja za pregled servisa

(Izvor: izrada autora)

Na slici iznad može se vidjeti da su prikazani svi servisi koji su dostupni na određenom serveru. Na pritisak bilo kojeg servisa vide se podržane metode tog servisa.



Slika 22: Prikaz metoda Artikl Service

(Izvor: izrada autora)

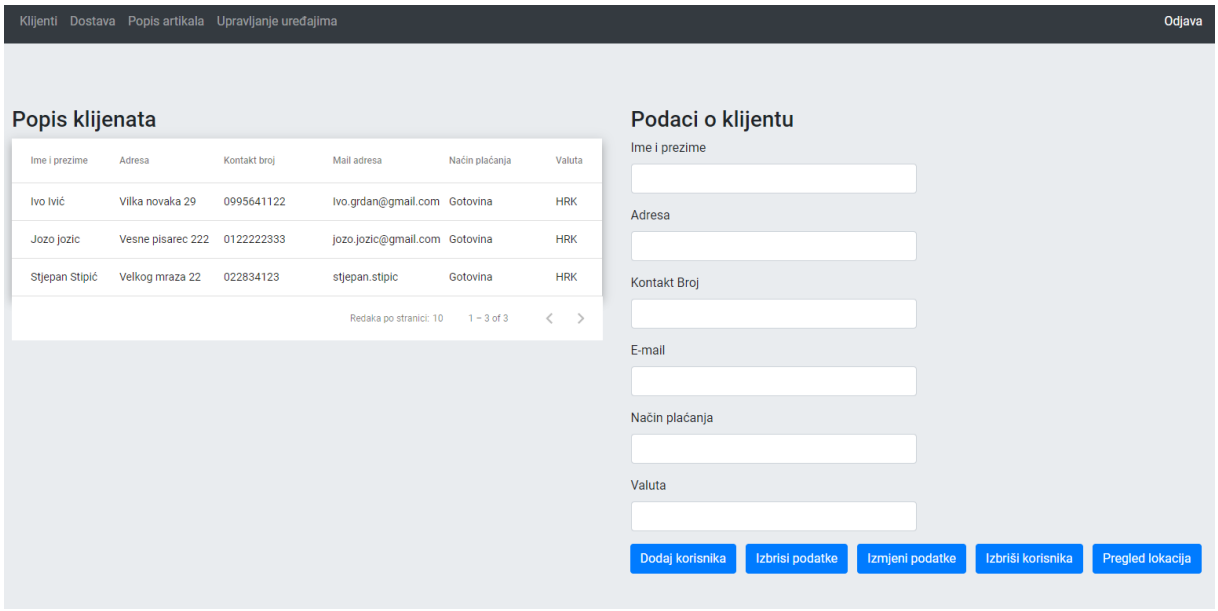
Na slici 21 vidljive su podržane metode servisa artikl. Moguće je i pritiskom na svaku od tih metoda dobiti detaljniji pregled koji opisuje na koji način je moguće pozvati metodu, koji su parametri potrebni za slanje i koja zaglavlja su potrebna za poziv metode na servisu.

8.6.4. Servis bez stanja

Idući princip koji je upotrijebljen u aplikaciji je princip servisa bez stanja. Trenutno se u postojećim servisima ne pamti nikakvo stanje, svi podaci koji se trebaju pamtiti su direktno zapisani u bazi. Ovakav način rada servisa omogućava veliku skalabilnost i iskoristivost.

8.7. Izgled ekrana aplikacije

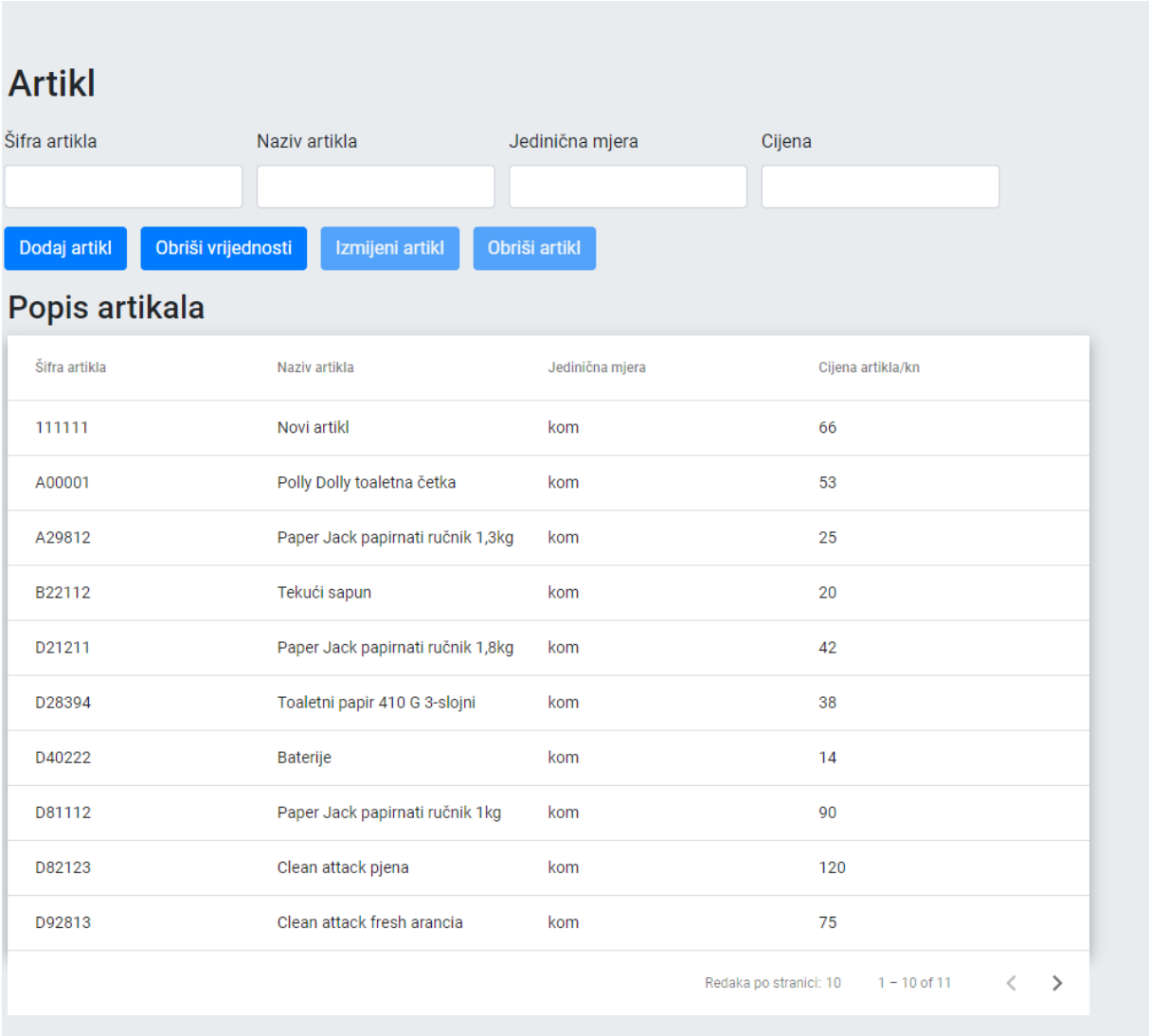
Kreirana aplikacija sastoji se od nekoliko ekrana koji služe za pregled, unos, izmjenu i brisanje podataka koji se unose. Početna stranica koja se otvara nakon prijave u aplikaciju je stranica klijenata. Na stranici klijenata može se vidjeti popis klijenata s kojima se surađuje te se za te klijente u nastavku mogu unositi osnovni podaci koji su potrebni za rad s klijentom. Prikaz ekrana klijenta prikazan je na slici u nastavku.



Slika 23: Prikaz ekrana klijenti

(Izvor: izrada autora)

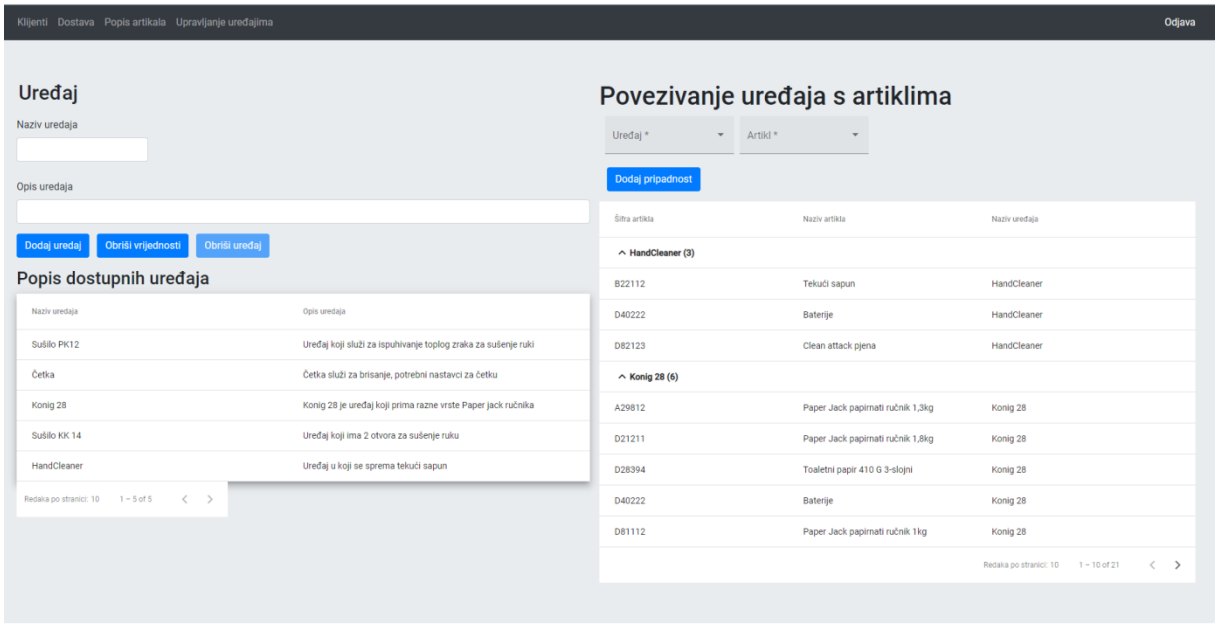
Odabirom u navigacijskom izborniku na „Popis artikala“ otvara se ekran u kojemu se nalaze popis artikala s podacima poput šifre, naziva, jedinične mjere i cijene. Na istom tom ekranu moguće je raditi unos i sve ostale izmjene vezane uz artikl. Ekran se može vidjeti na slici u nastavku.



Slika 24: Prikaz ekrana popis artikala

(Izvor: izrada autora)

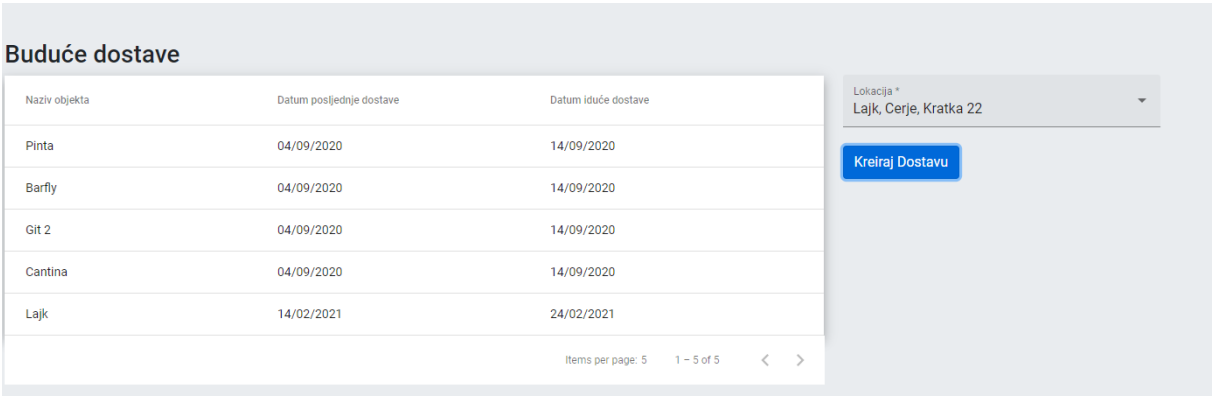
Odabirom „Upravljanje uređajima“ otvara se ekran u kojem su prikazani svi dostupni uređaji. U nastavku ekrana nalazi se funkcionalnost koja omogućuje spajanje uređaja s pripadajućim artiklima koji su u ponudi. Svi ti podaci prikazani su u obliku tablice dok su podaci povezanosti uređaja unutar tablice grupirani prema uređaju. Ekran je vidljiv na slici u nastavku.



Slika 25: Prikaz ekrana upravljanje uređajima

(Izvor: izrada autora)

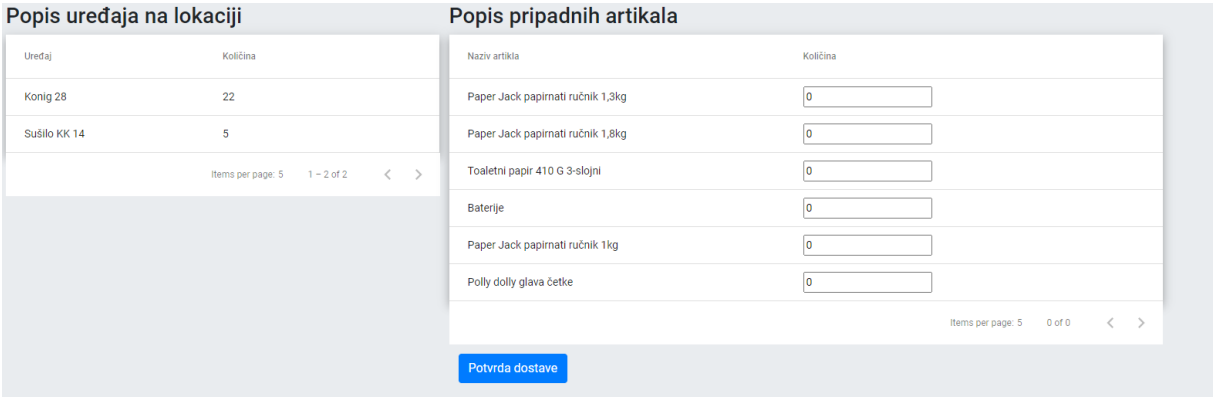
U nastavku još postoji nekoliko unosnih ekrana sličnih prethodno navedenim. Preostali ekrani su ekrani za unos lokacije, zaduživanje uređaja za odabranu lokaciju i određivanje cijene specifične za lokaciju. Glavni ekran ove aplikacije je ekran zadužen za dostave. Ekran je prikazan u nastavku.



Slika 26: Prikaz ekrana dostava

(Izvor: izrada autora)

Nakon odabira lokacije za koju se želi kreirati dostava u nastavku ekrana otvara se drugi dio ekrana u kojem se nalazi prikaz uređaja na odabranoj lokaciji i popis artikala koji pripadaju tim uređajima, nakon unosa količine artikala potvrđuje se dostava i ispisuje se ukupna cijena dostave.



Slika 27: Prikaz ekrana dostava nakon odabira lokacije

(Izvor: izrada autora)

9. Zaključak

Cilj ovog rada bio je prikazati na koji način se može pristupiti razvoju aplikacije koristeći servisno orijentiranu arhitekturu. Na početku rada objašnjene su sve tehnologije koje su korištene u realizaciji aplikacije. Rad se sastoji od dva dijela. Prvi dio rada sastoji se od teorijskih koja su primijenjena u drugom dijelu rada. Opisane su karakteristike SOA-e te su objašnjeni osnovni dijelovi poput infrastrukture i arhitekture. Temeljem uzoraka koji su obrađeni kroz rad odabran je skup uzoraka koji je iskorišten kod implementacije aplikacije. Kako su servisi danas neizostavni dio aplikacija, jer omogućuju brzu i jednostavnu komunikaciju putem komunikacijskih protokola, u aplikaciji je kreirano više servisa koji zajedno komuniciraju. Svaki servis može biti implementiran na više načina. U teorijskom dijelu moglo se vidjeti da postoje dvije vrste standarda koji se koriste kod implementacije web servisa, a to su REST i SOAP servisi. Kao što je navedeno u radu, svaki od tih standarda ima neke svoje prednosti i nedostatke. Kod implementacije aplikacije „Robna Kontrola“ odabran je REST standard jer zahtijeva manje resursa kod same obrade. Odabirom REST standarda odabran je i Spring Boot razvojni okvir jer pruža odličnu podršku za rad s REST servisima. Manjak sigurnosti servisa riješen je implementacijom sigurnosnih uzoraka koji štite aplikaciju od neželjenih korisnika. Od samih sigurnosnih uzoraka važno je istaknuti posredovanu provjeru autentičnosti koja koristi žeton za održavanje komunikacije između poslužiteljskog i korisničkog dijela aplikacije. Korištenjem uzoraka koji spadaju u dio sigurnosti radi se na poboljšanju sigurnosti aplikacije. Puno uzoraka koji su upotrijebljeni u izgradnji aplikacije nisu direktno vezani uz programiranje. Neki od takvih uzoraka koji su korišteni pokazuju da je potrebno vrlo oprezno pristupati izgradnji nekog sustava ili aplikacije. Na samom kraju rada prikazani su neki od ekrana od kojih se sastoji aplikacija te je ukratko opisano na koji način se oni mogu koristiti.

Ovim radom pokazan je razvoj jedne jednostavne aplikacije uz korištenje servisno orijentiranih principa. Kroz rad se mogu vidjeti koraci koji su rađeni u razvoju aplikacije te kojim tijekom je išao razvoj. Ovaj način razvoja nije standard ili pravilo koje se mora koristiti, ali je u ovom primjeru prikazano planiranje, razvoj i korištenje aplikacije. Važno je naglasiti da kod izgradnje aplikacije treba biti oprezan jer ako se radi o razvoju velikog sustava potrebno je dobro poznavati razvojne alate i okruženje za koji se razvija. Za razvoj većih sustava potrebno je imati mnogo iskustva da bi se dio pogrešaka mogao izbjeći u ranijoj fazi izgradnje sustava. Područje servisno orijentirane arhitekture vrlo je široko i kompleksno, ali poznavanje te arhitekture sigurno može samo pomoći kod izgradnje aplikacija.

Popis literature

[1] T. Erl, "Service-Oriented Architecture: Concepts, Technology, and Design", Prentice Hall PTR, 2005.

[2] T. Erl, "SOA Design Patterns", Prentice Hal lPTR, 2009.

[3] T. Erl, "SOA Principles of Service Design", Prentice Hall PTR, 2007.

[4] T. Erl, "SOA with REST: Principles, Patterns & Constraints for building Enterprise Solutions with REST", Prentice Hall PTR, 2013.

[5] C. Wodehouse, "SOAP vs. REST: A Look at Two Fifferent API styles", 2017. Dostupno: https://www.upwork.com/hiring/development/soap-vs-rest-comparing-two-apis/ [pristupano: 10.10.2019.]

[6] L. Ibanez, "SOAP vs REST", 2018. Dostupno: https://medium.com/@LazaroIbanez/soap-vs-rest-68faf2ea970e [pristupano: 12.10.2019.]

[7] RESTful API, "Basic Guidelines" Dostupno: https://blog.restcase.com/restful-api-basic-guidelines/ [pristupano: 12.10.2019.]

[8] RestApiTutorial, "Using HTTP Methods for RESTful Services", Dostupno: https://www.restapitutorial.com/lessons/httpmethods.html [pristupano: 22.10.2019.]

[9] A. Monus, "SOAP vs REST vs JSON comparison", 2019., Dostupno: https://raygun.com/blog/soap-vs-rest-vs-json/ [pristupano: 11.11.2019.]

[10] M. Rouse, "SOAP (Simple Object Access Protocol)", Dostupno: https://searchapparchitecture.techtarget.com/definition/SOAP-Simple-Object-Access-Protocol [pristupano: 13.11.2019.]

[11] IBM Knowledge Center, "The structure of a SOAP message" , Dostupno: https://www.ibm.com/support/knowledgecenter/en/SSMKHH_10.0.0/com.ibm.etools.mft.doc/ac55780_.htm [pristupano: 13.11.2019.]

[12] SOA Patterns , Dostupno: https://patterns.arcitura.com/soa-patterns [pristupano: 23.11.2019.]

[13] Symmetric vs. Asymmetric Encription – What are differences? , Dostupno: https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences [pristupano: 24.11.2019.]

[14] Public Key Cryptography and Digital Signatures, Dostupno: https://medium.com/coinmonks/a-laymans-explanation-of-public-key-cryptography-and-digital-signatures-1090d4bd072e [pristupano: 24.11.2019.]

[15] Fusion Middleware Securing Applications with Oracle Platform Security Services, Dostupno:

https://docs.oracle.com/middleware/12213/opss/JISEC/idstoreadm.htm#JISEC9189 [pristupano: 24.11.2019.]

[16] Angular , Dostupno : https://Angular.io/ [pristupano: 12.06.2020.]

[17] SpringBoot , Dostupno : https://Spring.io/projects/Spring-Boot [pristupano: 12.06.2020.]

[18] Program Definition , Program, Dostupno : https://techterms.com/definition/program [pristupano: 12.06.2020.]

[19] What is REST , Dostupno: https://restfulapi.net/ [pristupano: 12.06.2020.]

[20] REST API tutorial , Dostupno: https://restfulapi.net/rest-architectural-constraints/ [pristupano: 12.06.2020.]

Popis slika

Popis tablica