UNIVERZITET
"DŽEMAL BIJEDIĆ"
U MOSTARU

Dokumentacija Softverske Arhitekture

eCommerce Web API

Predmet : Softverska arhitektura

Predmetni Profesor: Dražena Gašpar

Smjer: Softverski Inženjering

Student: Armin Smajlagic | IB190093

Sadržaj

1. Uvod

1.1 Svrha

Ovaj projekat nudi standardne eCommerce funkcionalnosti za kupce i administratore.

Sa kupceve strane to podrazumjeva :

- Pregled proizvoda
- Dodavanje proizvoda u korpu
- Te checkout narudžbe (kreiranje narudžbe za stavke iz korpe)

Sa administratorske strane to podrazumjeva :

- Upravljanje proizvodima (dodavanje, brisanje i uređivanje)
- Upravljanje kupcima (dodavanje, brisanje i uređivanje)
- Upravljanje narudžbama (dodavanje, brisanje i uređivanje)

Sve navedene mogućnosti dolaze u formi API-ja koji nudi pristup podacima pohranjenim unutar baza podataka. Dakle prezentacijski nivo nije neki standardan GUI već Web API koji se lagano integriše u bilo koji Web bazirani sistem(UI/Non-UI).

Motivacija ovog projekta jeste demonstracija najpoznatijih softverskih arhitektura današnjice i to je bio i jedini cilj kroz njegovu implementaciju. Svaka pojedina usluga spomenuta iznad je organizirana u zasebni neovisni modul sa vlastitom bazom podataka.

eCommerce kao posebna domena nije bila cilj i motive već sama softverska arhitektura primijenjena kao generalno rješenje.

Proizvodi, odnosno same kategorije proizvoda su postavljane na što generalniji način tako da se lagano mogu modifikovati za druge oblike komerca. Ovo se odnosi na jedinstvene inicijalne podatke proizvodi i kategorije na kojima se dalje mogu generisati drugi podaci kao narudžbe, korpe, kupci i sl.

1.2 Obim

Obim je sam eCommerce odnosno standardne usluge jednog eCommerca prema kupcima i administratorima tog sistema. Sistem pokriva jednu virtualnu trgovinu koje je veoma jednostavna za modifikaciju u ponovnu upotrebu u drugim vrstama virtualnih trgovina. Evidencija skladišta i plaćanje nije obuhvaćena u projekat. Daljnje dijeljene sistema po uslugama se vrši na administratore i kupce.

Administratori kontrolišu cjelokupan sistem odnosne sve podatke koje kupci generišu i vide tokom upotrebe sistema. Oni imaju uvid u sve kupce i njegove detalje (narudžbe, kupovine, korpe). U poglavlju 4 (Bitni Use-Case)se dublje razmatraju mogućnosti administratora.

Pored nadzora administratori dodaju kategorije i proizvode koje kupac može da pregleda. Kao što je spomenuto u uvodu kupac je u mogućnosti da pregleda proizvode a na osnovu toga dodaje ih u korpu. Ako je zadovoljan sa sadržajem korpe može izvršiti „checkout“ i time kreirati narudžbu. Također može da pregleda korpe i narudžbe ta da ih ukloni ako to želi. Ovime se objedinjuju sve zainteresovane strane (tj. administratori i kupci).

1.3 Često korišteni koncepti i skraćenice

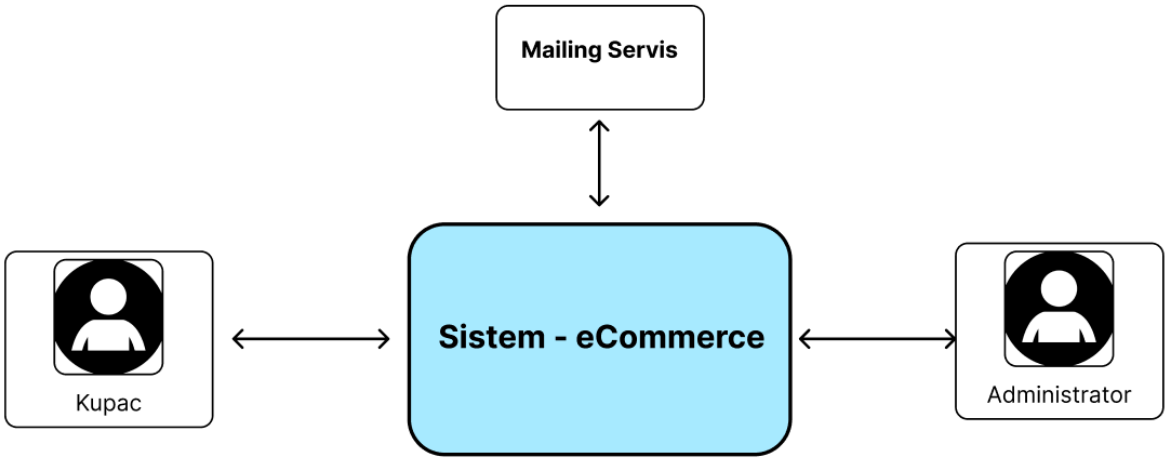
API	Aplikacijski Programski Interfajs
HTTP	Hyper Text Transfer Protocol
API Gateway	Među komponenta između klijenta i servisa mikroservisne arhitekture
GUI	Grafički korisnički interfejs
Caching	Keširanje (spremanje) odgovora servisa na HTTP zahtjev klijenta radi uštede resursa
Fascade	To je tkz. fasada koja je enkorporirana u API Gateway zapravo potiče od dizajn paterna fascade koji predstavlja unificiranu pristupnu tačku za više resursa.
Auth	Podrazumijeva se autentikacija i autorizacija od strane sigurnosnog entiteta unutar sistema (IDS4, Gateway i sl.)
Load balancing	Balansiranje opterećenja na servise sistema slučajnom raspodjelom.

2. Predstavljanje Softverske Arhitekture

2.1 Kontekst Sistema

Kontekst sistema uključuje korisnike, sistem i externe aktere kao email servis. Van sistema imam email servis koji služi za potvrdu kreiranja narudžbe. Kupac i administrator su u odnosu sa sistemom što može izazvati interakciju sa eksternim servisom za slanje mailova (kupovine, registracije).

Kontekst je da dodatno obrazložen dijagramom konteksta na slici 1

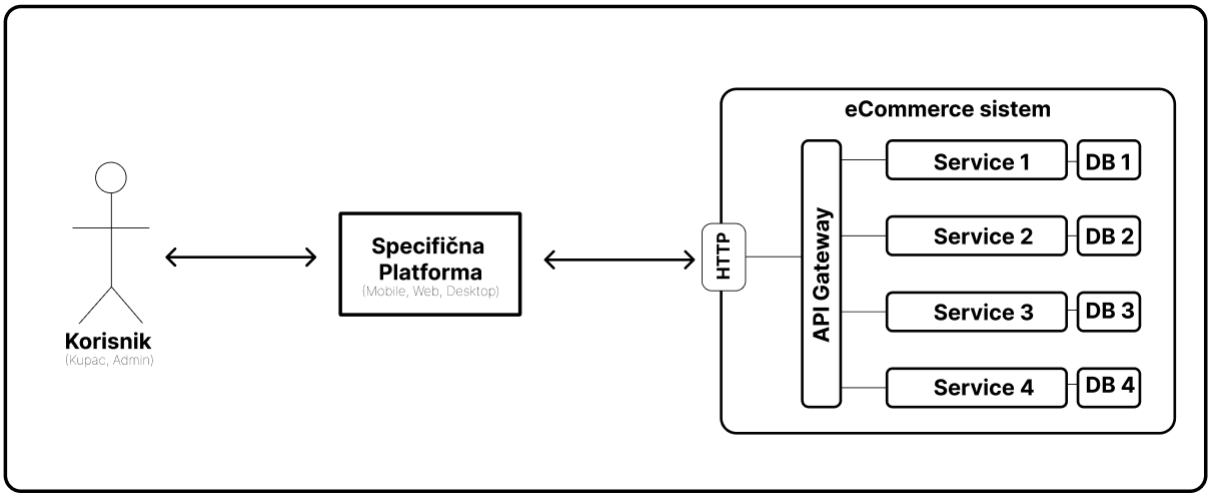


Slika 1. Dijagram Konteksta eCommerce sistema

2.2 Interakcija Korisnika

Interakcija korisnika sistema je ujedinjena putem API servera. Jedina razlika pristupa pored potencijalnog korisničkog interfejsa je pristupni ključ (JWT) koji se dodjeljuje kupcima i administratorima (privilegije su sadržane u samom JWT). Na isti nače se razlikuju i kupci. Pošto se radi o API interakcija je pogodna samo za druge vrste softvera kao što su desktop, web i mobilne aplikacije. Oni ostvaraju komunikaciju putem HTTP-a i time dobivaju usluge sistema.

Kako bi se bolje razumjela interakcija koristit ću dijagram interakcije (slika 2).



Slika 2. Dijagram Interakcije

2.3 Ciljevi Arhitekture

Cilj je da su ponudi rješenje koje je brzo, efikasno i otporno na greške. Spomenuta svojstva ne smiju se mijenjati ako imamo veći broj korisnika, ili imamo neku modifikaciju sistema koju želimo isporučiti. Ukoliko dođe do greške želimo da se sistem brzo oporavi i da se negdje spremi ta greška kako bi se moglo pravovremeno to riješiti. Također želimo unificiran i jednostavan pristup za sve klijente a da se održi određeni nivo sigurnosti.

Ovi ciljevi se mogu postići kvalitetnom analizom i formulisanjem potreba klijenata (u narednom poglavlju se vrši analiza), te implementacijom rješenja zasnovanog na toj analizi. Pored potreba klijenta uzeti ću u obzir i dizajn i implementaciju, dakle želimo obratiti pažnju na kompleksnost rješenja radi skalabilnosti i modifibilnosti sistema. Komplikovano rješenje će imati efekta na implementacijski tim, kvalitetu i vremenu potrebnom za implementaciju.

Ovime su objedinjeni najbitnije ciljeve arhitekture.

3. Karakteristike Softverske Arhitekture

U ovom poglavlju ću analizirati potrebe klijenata (mobile, desktop, web) za specifičnim karakteristikama (kvalitetama). Cilj je doći do najpovoljnijeg rješenja, odnosno pravilne odluke pri odabiru arhitektura. Pored klijentskih potreba uzeti ću u obzir i lakoću razvoja, testiranja, isporuke i mogućnost proširenja radi proces izrade samog rješenja.

Sve spomenute kvalitete je potrebno uzeti u obzir kako bi se postiglo dugoročno upotrebljivo eCommerce rješenje a olakšao razvoj i održavanje sistema .

3.1 Dostupnost

Sistem treba da bude spremno za upotrebu kada je potrebno da radi. Nakon toga cilj je omogućiti klijentima usluge bez prekida. Također je moguće da u toku životnog vijeka rješenja sistem padne pa je neophodno da se što prije vrati u radno stanje (eng. recovery). Eventualno ako dođe do greške u pokušaju pristupa API-ja od strane klijenata želimo određeni nivo otpornosti na greške (eng. resiliency). Trebamo uzeti u obzir i klijentske napade tipa (DOS – Denial Of Service) koji može narušiti svojstvo **dostupnosti**. Ovo se može riješiti sa API Gateway-om koji blokira zahtjeva ograničavanjem HTTP poziva na server u definisanoj jedinici vremena (detaljnije o potencijalnim rješenjima u poglavlju 5).

Cilj postizanja karakteristike dostupnosti je da se minimizira vrijeme ne dostupnosti usluga API-ja.

3.2 Modifibilnost i Skalabilnost

Uzimanjem u obzir da se sistem mora eventualno izmijeniti i proširiti potrebno je obratiti pažnju i na sposobnost sistema da se lagano modificira. Na primjer ako želimo dodati novi modul za uplate (eng. scale) ili ako želimo modifikovati već postojeće module neophodno je rješenje učiniti dovoljno jednostavnim za **razvoj i expanziju**. Želimo nekako ukloniti zavisnost komponenata (učiniti ga modularnim i nezavisnim o drugim modulima) kako bi eliminirali veze i olakšali proces razvoja novih ili modifikaciju postojećih komponenti sistema.

Generalno ova svojstva bi se mogla razdvojiti ali pošto su mogu zajedno uzeti u obzir ja sam ih grupisao. Pored ove dvije kvalitete mogla bi se uzeti u obzir testibilnost radi dugoročnog i pravilnog razvoja i održavanja.

Želimo sistem pojednostaviti radi procesa development, ekspanzije koje bi rezultirala u skalabilnosti i modifibilnosti sistema.

3.4 Deployability

Nakon bilo koje izmjene sistema potrebno je to rješenje isporučiti korisnicima. Kako se planira da se sistem skalira i modificira potrebno je obratiti pažnju na isporuke novih verzija sistema. Uz dodatne tehnologije kao docker, kubernetis, jenkins, azure i sl. isporuka se može olakšati čak i vrijeme nedostupnosti minimizirati na nulu. Mikroservisna arhitektura može imati i druge benefite pored isporuke kada se koriste spomenute tehnologije kao što je replikacija servisa (poboljšava svojstvo dostupnosti).

Želimo da rješenje bude laganu isporučiti sa minimalnim vremenom nedostupnosti

3.4 Testibility

Uz dodatak na skalabilnost, modifibilnost i isporuku koje padaju na development tima da obave napominjem i bitnost testibilnosti. Skup od ova 4 svojstva su potrebna kako bi se postiglo rješenje koje je dugoročno lagano za razvoj i održavanje. Svaki od modula sistema bi trebao biti lagan za testiranje kako bi se što prije popravile greške i isporučilo kvalitetno rješenje.

Želimo osigurati kvalitet softvera testiranjem pa je bitno da je proizvod testibilan

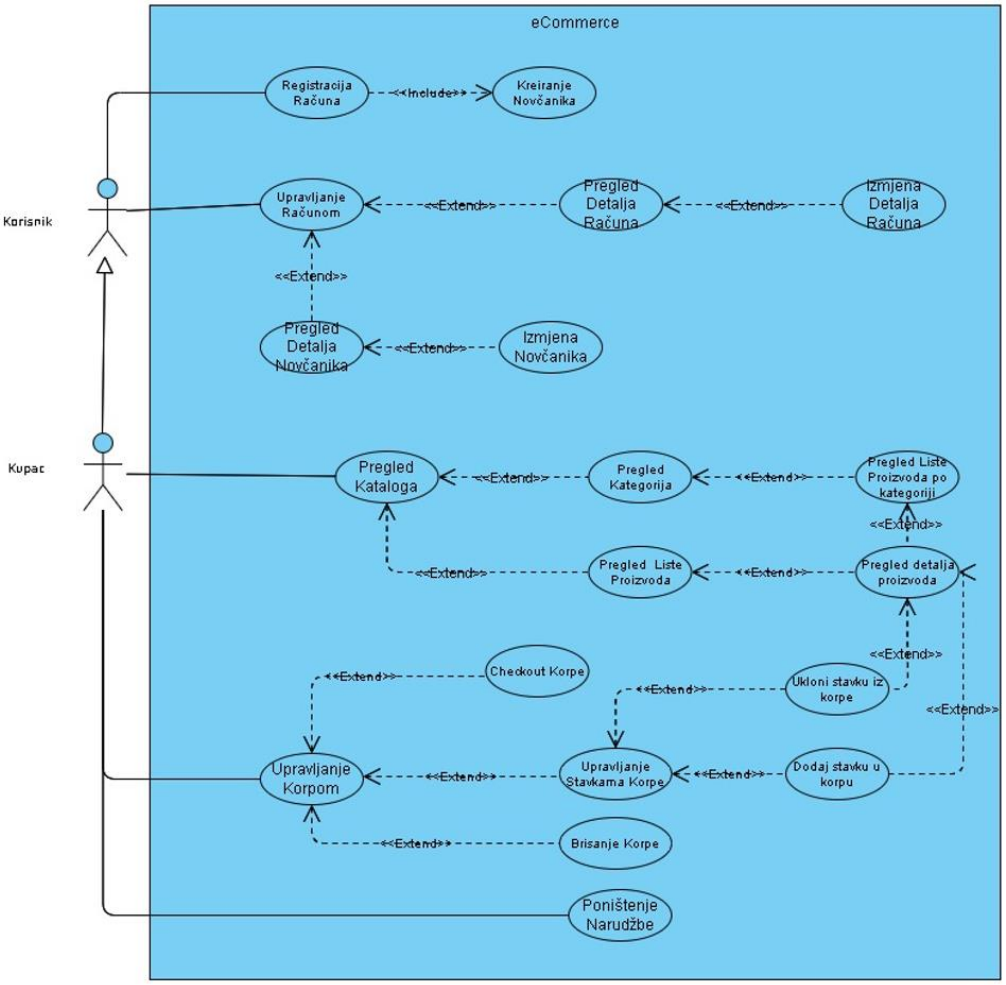
4 .Značajni use-case-ovi

Potrebe klijenata se dijele na dvije glavne skupine. Prvi skup potreba je namijenjene za klijenta a drugi za menadžment. Sve potrebe klijenata od strane sistema su opisane use-case dijagramima. Admin kao i klijent su predstavljeno zasebnim nizom dijagrama za svaku od primjena.

4.1 Klijent

Klijent je potreban sistem za sljedeće radnje.

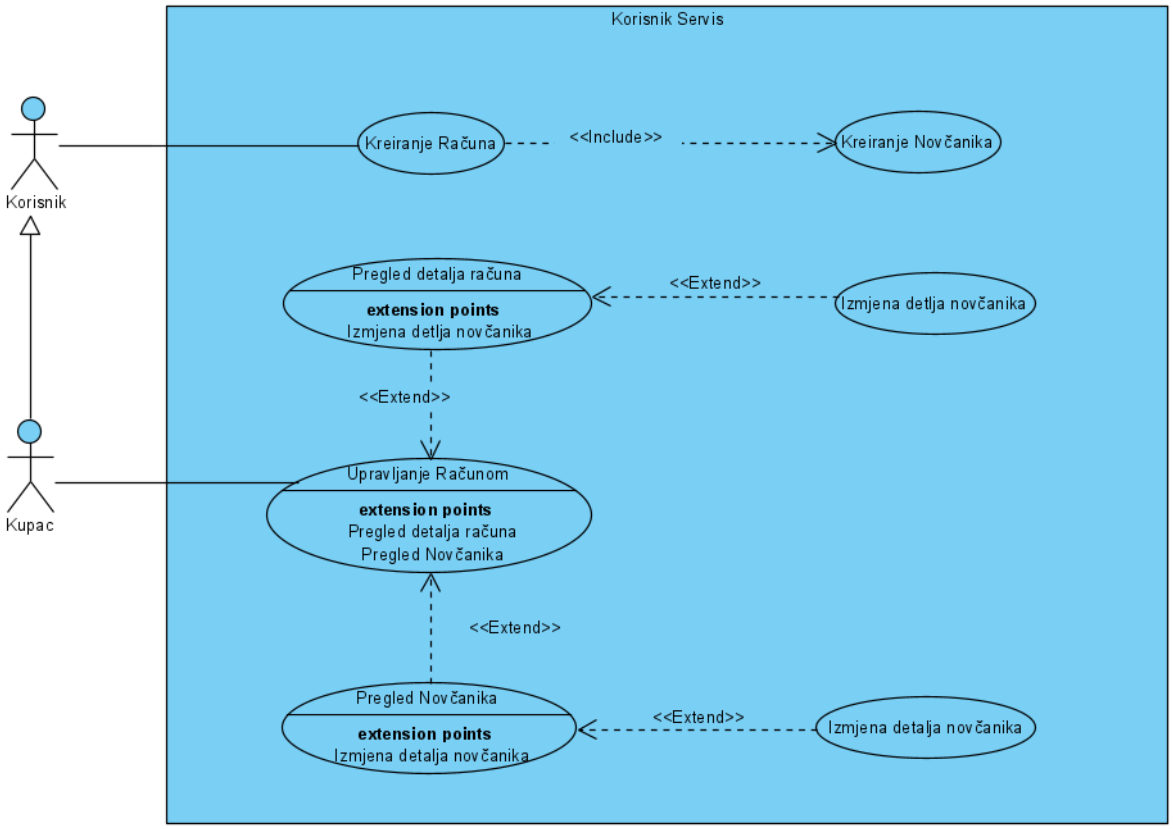
- Kreiranje i upravljanje računom i novčanikom
- Upravljanje korpom i kreiranje\cancel narudžbe
- Pregled kataloga



Slika 3. Akumulacija svih use-case dijagrama namjenjenih za kupca

4.1.1 Kreiranje i upravljanje računom i novčanikom

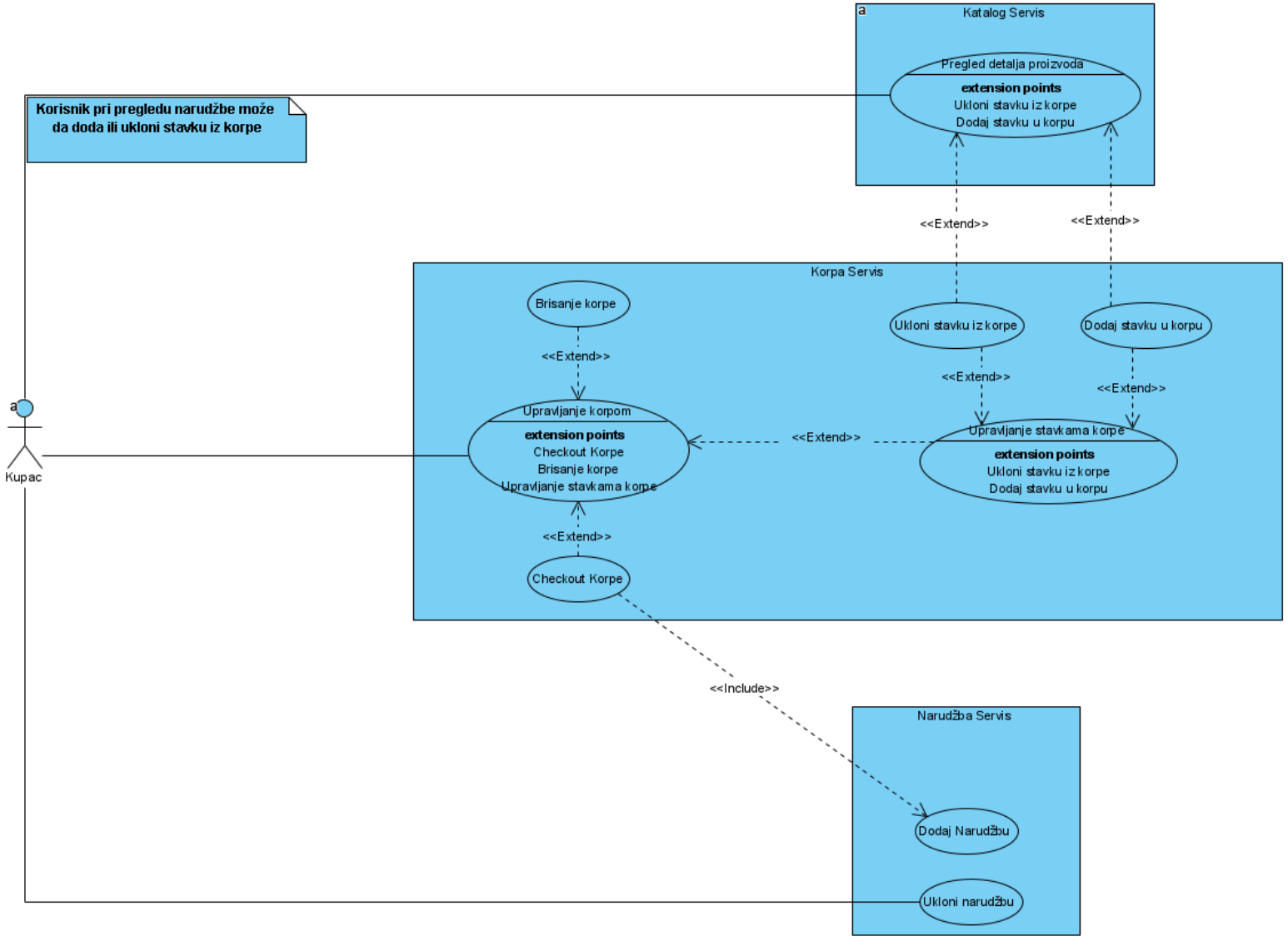
Kupci bi prije svega trebali biti u mogućnosti da kreiraju svoj račun (što automatski kreira i virutalni novčanik) kako bi mogli vršiti naručivanje i kupovinu proizvoda. Pored kreiranja također bi trebao moći da pregleda detalje svog računa i novčanika te eventualno ih izmjeniti.



Slika 4. Use-Case dijagram za kreiranje i upravljanje računima i novčanicima

4.1.2 Upravljanje korpom i kreiranje\cancel narudžbe

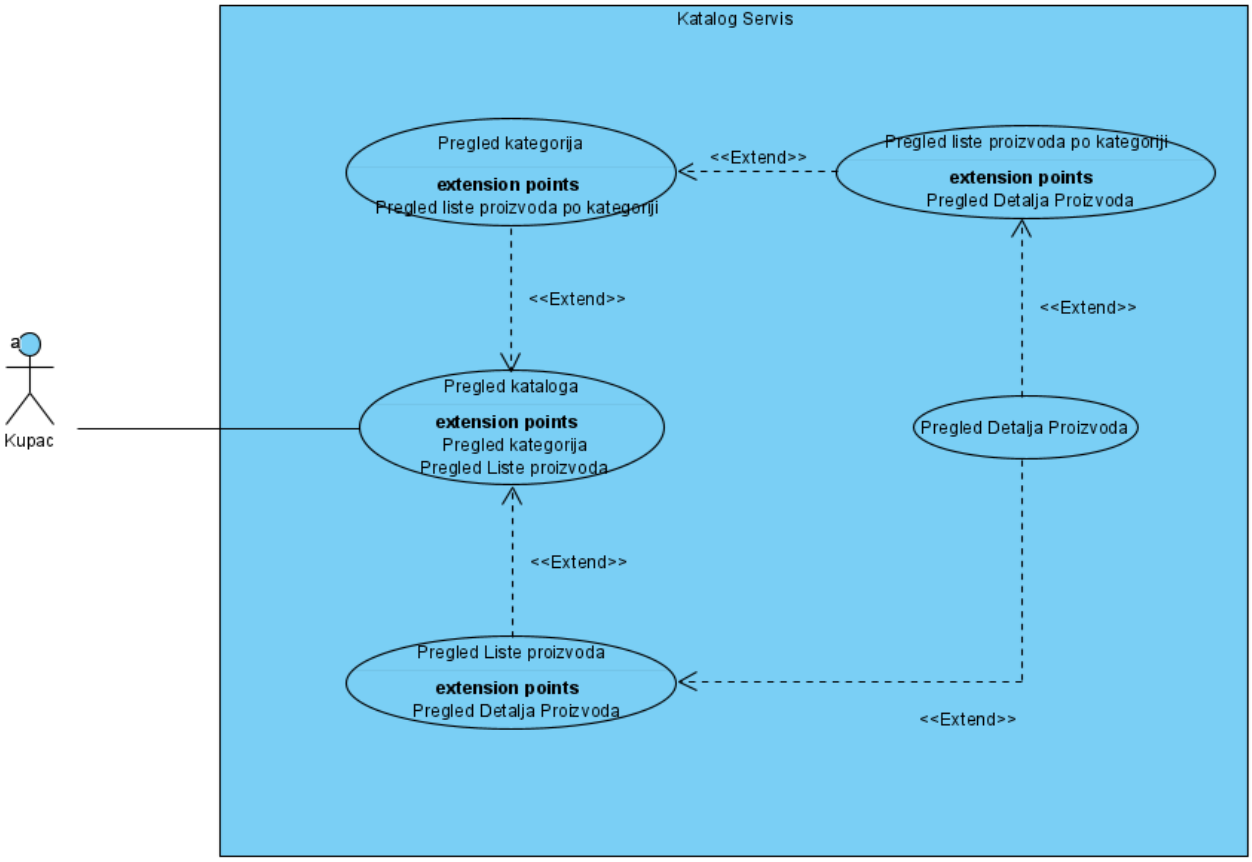
Pošto se radi o mikroservisima, moduli sistema su odvojeni pa je potreban inter-sistemska komunikacija. Korisnik bi trebao biti u mogućnosti dodati stavku u korpu pri pregledu kataloga. Ukoliko je zadovoljan sa sadržajem korpe trebao bi moći da kreira narudžbu na osnovu nje. I naravno da obriše svoju narudžbu ili isprazni korpu.



Slika 5. Usa Case dijagram za upravljanje korpama i narudžbama

4.1.3 Pregled kataloga

Korisnik bi trebao da pregleda proizvode (opcionalno po kategorijama - filter) prije dodavanja stavki u korpu i kreiranje narudžbe.

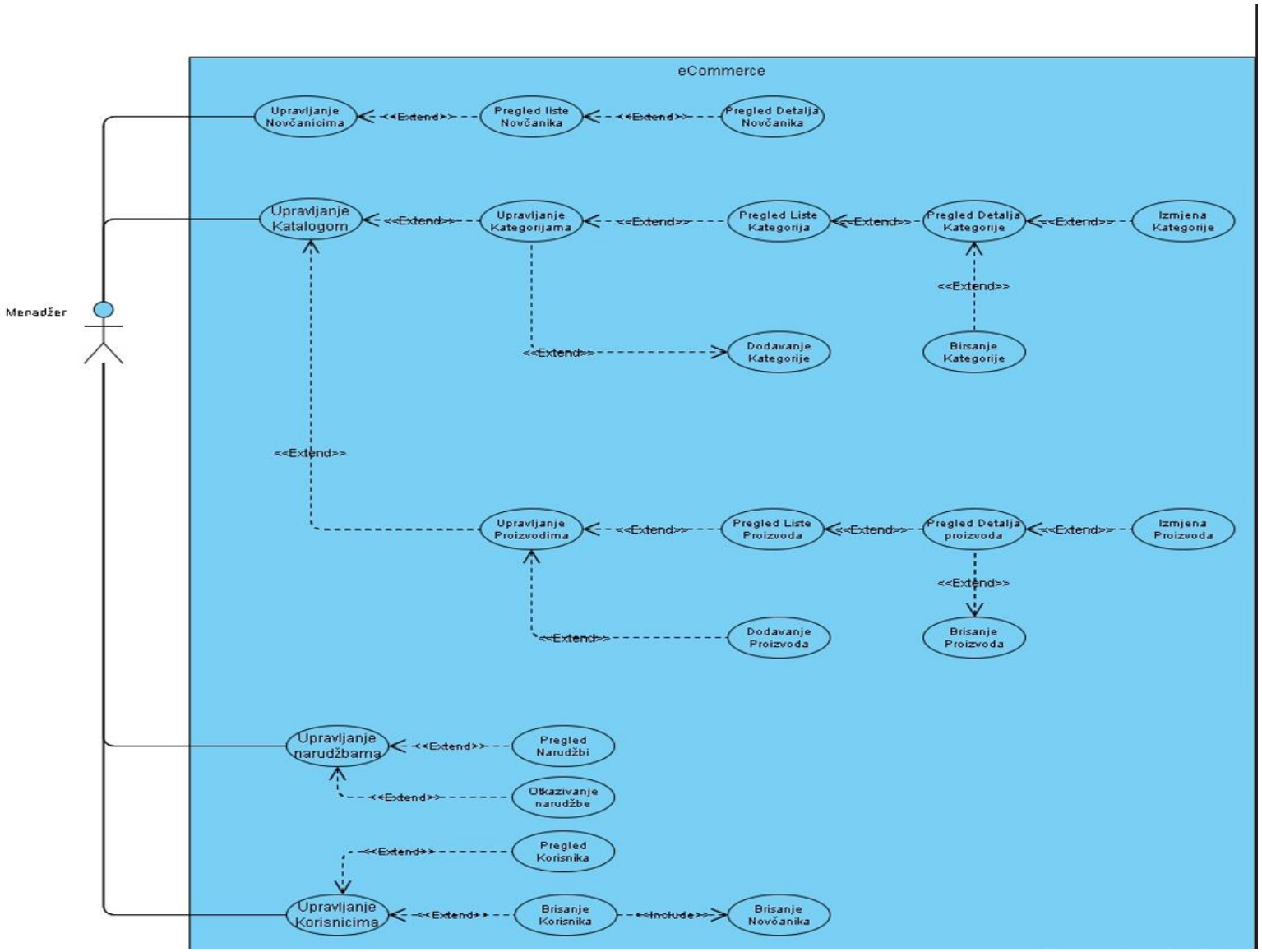


Slika 5. Use Case Pregleda Kataloga

4.2 Menadžment

Menadžment se sastoji od nekoliko use-case-a, i to :

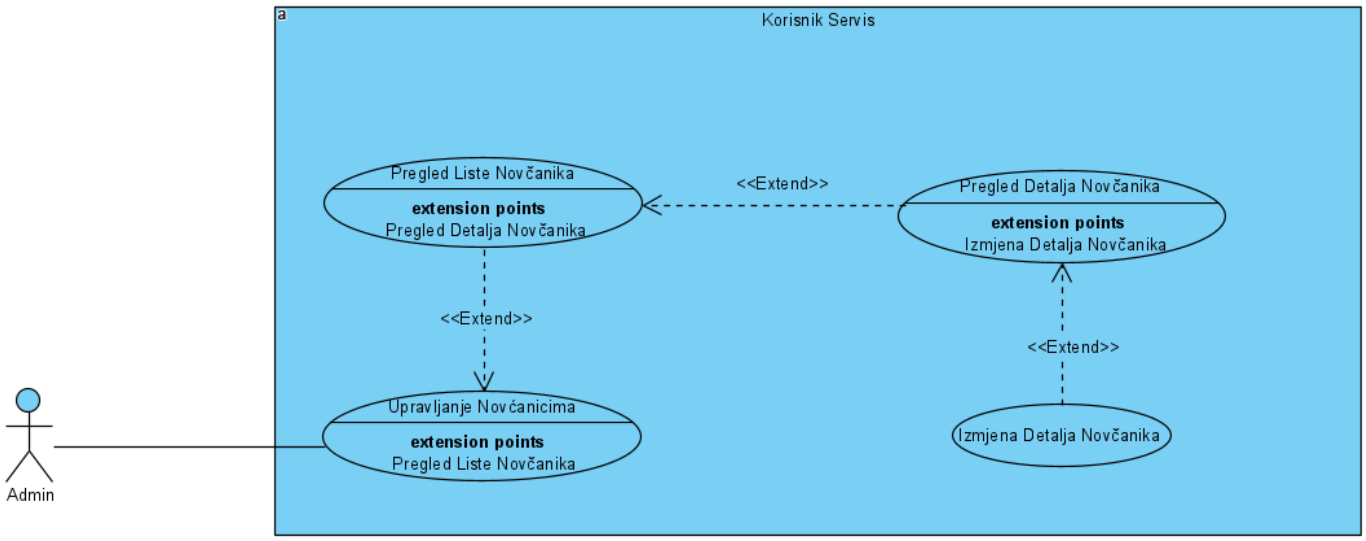
- Upravljanje novčanicima
- Upravljanje katalogom
- Upravljanje Korisnicima
- Upravljanje Narudžbama



Slika 7. Use Case akumulacija svih potreba na menadžment strani

4.2.1 Upravljanje Novčanicima

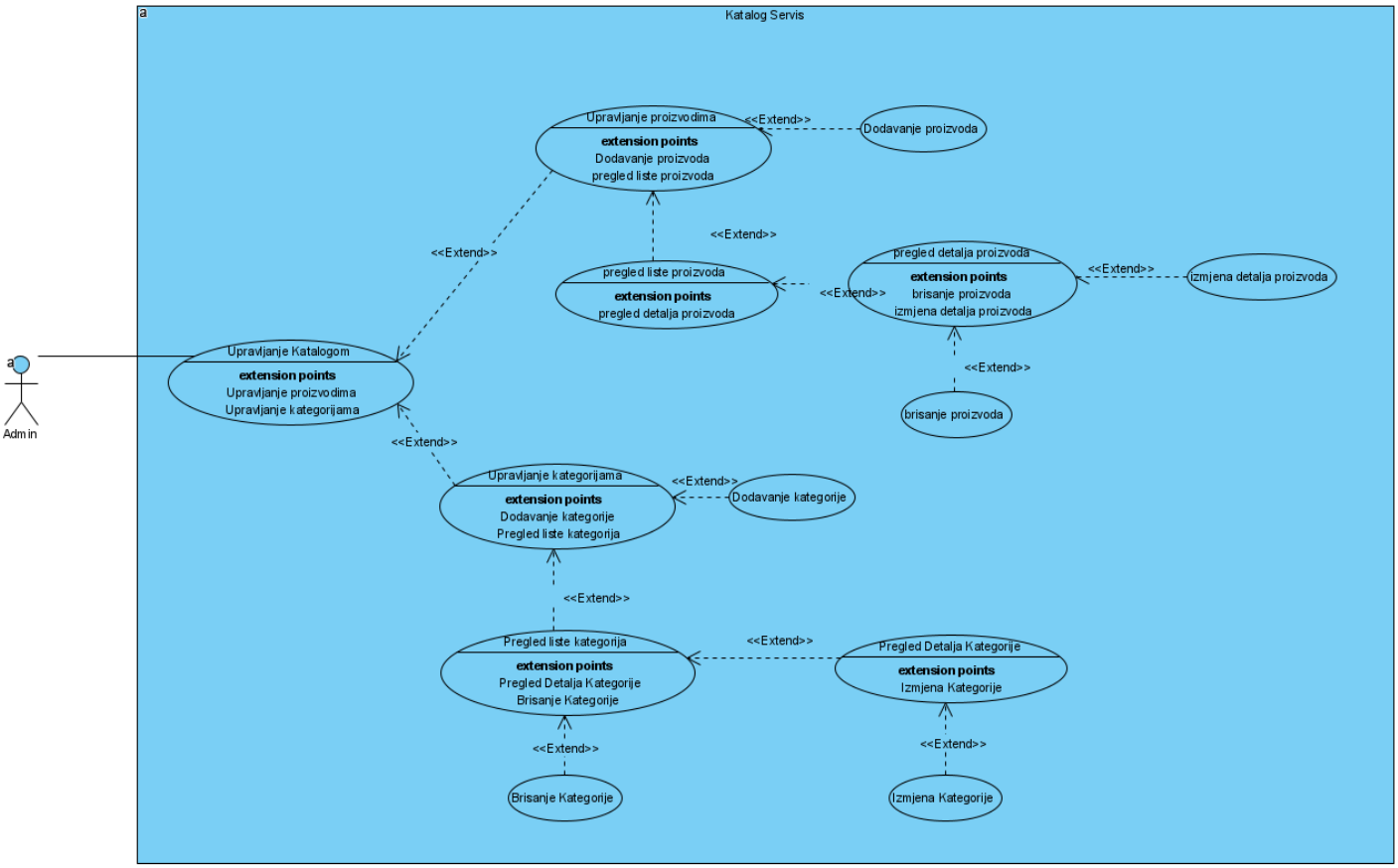
Članovi menadžment tima bi trebali biti u mogućnosti da pregledaju sve novčanike korisnika, te ukoliko žele da izmjene detalje novčanika pojedinih korisnika.



Slika 8. Use Case upravljanja novčanikom

4.2.2 Upravljanje Katalogom

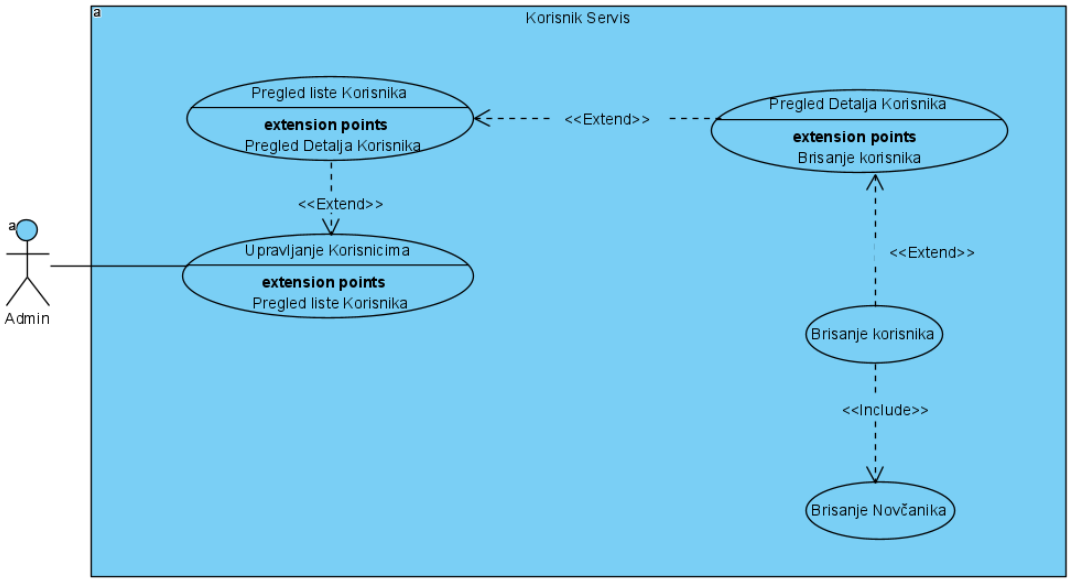
Članovi menadžment tima bi trebali biti u mogućnosti da pregledaju kategorije i proizvode te ukoliko žele da izmjene njihove detalje, obrišu ih ili dodaju novu kategoriju ili proizvod u odgovarajuće liste.



Slika 9. Use-Case upravljanje katalogom

4.2.3 Upravljanje Korisnicima

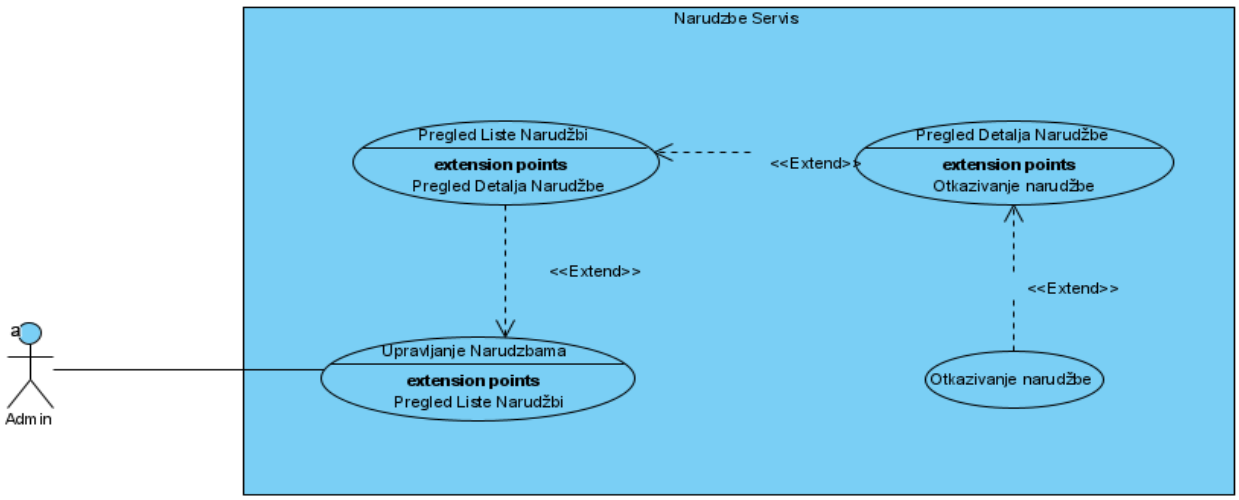
Članovi menadžment tima bi trebali biti u mogućnosti da pregledaju liste korisnika, detalje pojedinih korisnika, te ako žele da obrišu korisnika (i time obrišu i novčanik).



Slika 10. Use case upravljanje korisnicima

4.2.4 Upravljanje Narudžbama

Članovi menadžment tima bi trebali biti u mogućnosti da pregledaju listu kreiranih narudžbi, njihove detalje, te ukoliko žele da otkažu narudžbu .



Slika 11. Use case upravljanje narudžbama

5. Vrste (paterni) softverske arhitekture

Vrste softverskih arhitektura koje ćemo analizirati su mikroservisna (REST-based topology), slojevita i clean arhitektura. Sve tri arhitekture su potencijalno rješenje.

5.1 Paterni i kakarteristike

Mikroservisna arhitektura za cilj ima podijeliti sistem na više manjih jedinki (servisa) koji mogu ali i ne moraju biti ovisni jedan o drugom . Ova arhitektura nudi mnoge prednosti za razliku od monolitnih ukoliko joj pravilno definišemo gralunarnost i zavisnost. Kako je sistem podijeljen u servise to znatno olakšava **modifibilnost** i **testiranje** pojedinih servisa (manje kompleksni i manje veličine). **Isporuka** je još jedan benefit zbog toga što možemo samo jedan modul sistema isporučiti a ne moramo čitav sistem. Pri isporuci možemo isporučiti više instanci istog servisa te redirektovati pozive ukoliko jedan servis padne. Korištenje API Gatewaya možemo i izvršiti raspodjelu opterećenja na servise (eng. load balancing), odnosno korištenjem randomiziranih algoritama pozivati slučajnim odabirom instance istog servisa i time smanjiti opterećenje na jednu servis. Ovo utiče na svojstvo **dostupnosti** odnosno povečava **dostupnost** sistema. Mogućnost replikacije servisa znatno pomaže postizanju **skalabilnog** rješenja jer se pozivi mogu raspodijeliti po spomenutim instancama. Kako su pojedini servisi ograničeni jednom odgovornošću jednostavno ih je **skalirati** a pri tome zadržati jednostavnost.

Slojevita arhitektura je odlično rješenje ukoliko želimo da sistem bude jednostavan i lagan za modifikaciju. Radi se o monolitnoj strukturi koja se sastoji od horizontalnih slojeva gdje svaki ima određenu odgovornost unutar sistema. Prednosti ove arhitekture su **modifibilnost** zbog njene jednostavnosti, kao i **testibilnost** jer se slojevi mogu testirati neovisno jedan od drugog. Sa druge strane **isporučljivost** nije jedna od prednosti jer njena monolitna struktura zahtjeva isporuku cjelokupnog sistema dali to bilo nekoliko linija koda ili čitav novi sloj. Također bitno za spomenuti jeste da ako se slojevita ne implementirao kvalitetno **modifibilnost** bi mogla biti ugrožena jer bi svaka modifikacija mogla da zahtjeva izmjenu na svakom od slojeva. I finalno **skalabilnost** i **dostupnost** nisu poželjne kod ove arhitekture zbog njene monolitne strukture. Ukoliko puno korisnika optereti sistem i on padne nema jednostavnog rješenja kao kod mikroservisne arhitekture.

Clean arhitektura je tu kao čisto rješenje za **skalabilne**, **modifibilne** i **testabilne** aplikacije. Cilj je da se developeri manje brinu o refakaktoringu standardnog koda jer clean arhitektura zahtjeva poseban pristup rješavanju standardnih problema kao komunikacija sa bazom podataka ili korištenje eksternih usluga kao SMS, mail i sl. Ideja je da se definiše jezgro koje je specifično za domenu (entiteti i poslovna logika) koja je ne ovisna o pohrani i infrastrukturi (baza podataka, te

mailing, sms servisi i sl.) Ova neovisnost domene od infrastrukture pomaže pri **testiranju** novih verzija sistema. **Modifibilnost** sistema je još jedna prednost zbog toga što je domena odvojena od infrastrukture pa nema izmjena kroz čitav tkz. slice. Pored prethodno spomenutih karakteristika **skalabilnost** je jedan od benefita ove arhitekture. Ono što je čini **skabilnom** za razliku od slojevite jeste to što se radi o modulima a ne o slojevima. Ti moduli su takve strukture da odvajaju poslovnu logiku od infrstrukture i time olakšava **modifikacije, testiranja i skaliranja**. Finalno spominjem da se radi o monolitnoj arhitekturi koja trpi o ukoliko sistem padne pa je **(ne)dostupnost** jedan nedostatak. Pored toga i **isporuke** su otežane jer je struktura monolitna.

5.2 Ograničenja Paterna

Slojevita arhitektura ima ograničenja kao što su otežana **skalabilnost** i **isporuka** zbog njene monolitne arhitekture. Pored toga **modifibilnost** može biti ugrožena ukoliko se slojevi i njihove ovisnosti ne definišu i implmenetiraju korketno. Također ukoliko jedan sloj izazove pad, čitav sistem će da padne i time ga učiniti nedostupnim. Zaključujem da je skalabilnost, ispručljivost, dostupnost i opcionalno modifibilnost slabije karakteristike ove arhitekture.

Clean je monolitna arhitektura koja ima prednosti u odnosu na slojevitu ali ipak trpi od slabe **dostupnosti**. Ukoliko jedan modul uzrokuje pad, čitav sistem će da padne što će ga učiniti ne dostupnim. Pored ovoga da li bile manje ili veće modifikacije sistema, zbog njegove monolitne strukture, isporuka je ograničena. Zaključujem da su isporučljivost i dostupnost nedostatci ove arhitekture.

Mikroservisna nema ograničenja u odnosu na karakteristike spomenute u pogavlju 3. Ona nije idealna arhitektura ali svih 5 spomenutih u poglavlju 3 su uredu.

5.3 Odluke i obrazloženja

Nakon razmatranja predonosti i nedostataka spomenutih arhitektura odabrao sam **mikroservisnu** arhitekturu. Njene kvalitete opisuju sve ono što želim da moj sistem ima. Pored toga svaki servis može imate svoju arhitekturu, pa ću u skladu sa time definisati i pod arhitekture svakog od servisa. Slojevita i clean nisu odabrane kao glavna arhitektura ali imaju druge poželjne karakteristike pa ću kao **pod arhitekture** koristiti njih.

Sigurnost i performanse se mogu uskladiti sa dodatnim tehnologijama. Pošto mikroservisna radi sa izoliranim modulima potrebna je komunikacija između pojedinih servisa. Rabbit MQ (Event bazirana biblioteka) može pomoći pri komunikaciji servisa. Time se smanjuje udar na performanse i usko vezivanje servisa (eng. tight coupling). Sigurnost sa druge strane se može optimizovati upotrebom Ocelot API Gatewaya koji autentikuje i po potrebi autorizira korisnike prije usmjeravanja zahtjeva na odgovarajuću adresu.

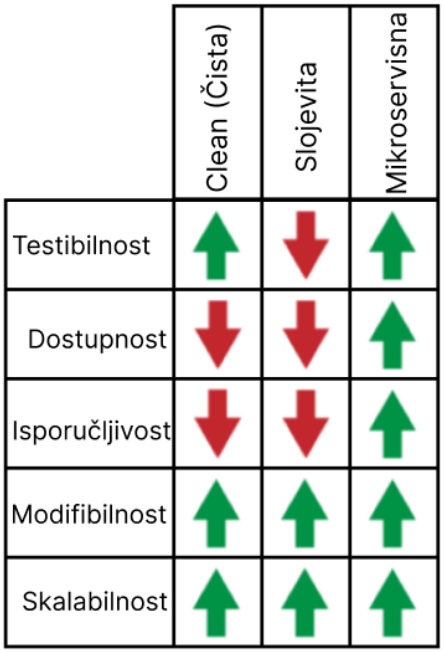
Slojevita i ako ima problema sa isporukom (i time i dostupnosti za vrijeme njene isporuke) ako je stavimo u jedan servis unutar mikroservisne arhitekture može se **eliminsati problemi sa dostupnosti**. U ovom slučaju slojevita bi bila pod arhitektura unutar mikroservisne. Isti problemi pronađeni u clean arhitekturi se mogu na isti način riješiti.

Svaki servis se može replicirati više puta korištenjem Docker-a, dakle isporučiti isti servis više puta i time poboljšati **fault tolerance i dostupnost**, a opterećenje raspodijeliti na različite instance servisa.

Clean arhitektura poboljšava pristup rješavanja standardnih problema oko upravljanja korisnicima i narudžbama koje padaju u domenu problema eCommerce sistema. Ona nudi **testibilnost**, **skalabilnost** i **modifibilnost** koje su željene kvalitete našeg sistema. Pored toga ograničenje spomenuto za dostupnost se može nadomjestiti ukoliko je promatramo kao jedan od servisa mikroservisne arhitekture.

Napomena: Pri izradi servisa u implementiranom projektu se koristi DDD (Domain-Driven-Development). Korištena literatura DDD-a je spomenuta u referencama.

Slika 12 i 13 prikazuju karaktersitike zasebnih arhitektura i sistema nakon odabira mikroservisne kao glavne arhitekture.



Slika 13. Karakteristike Zasebnih Arhitektura

Slika 14. Karakteristike sistema kada se koriste slojevita i clean kao pod arhitekture mikroservisne

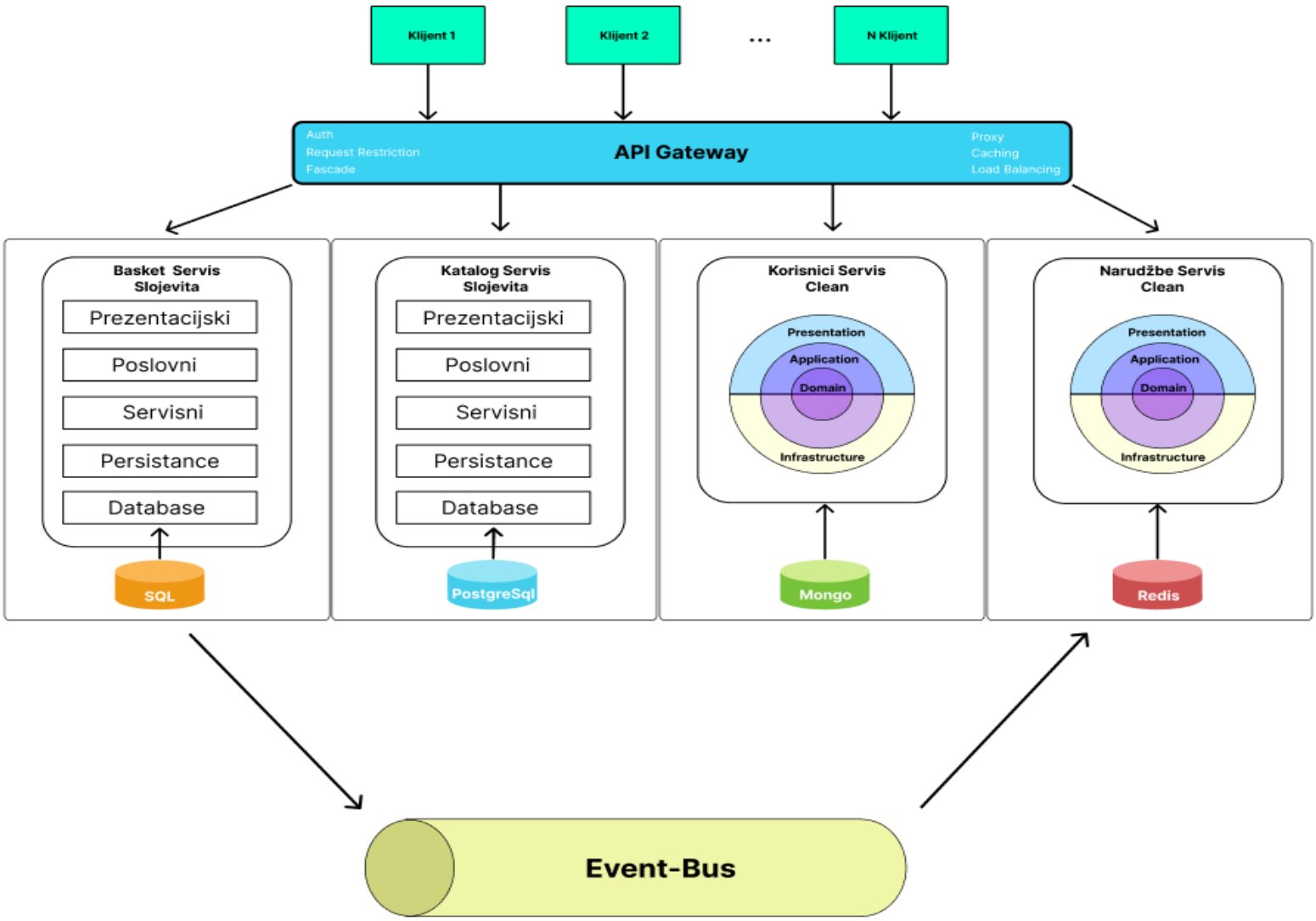
6. Logički pogled

Logički pogled je usmjeren na koje funkcionalnosti sistem pruža krajnjim korisnicima.

Svaki servis ima pristup svojoj bazi podatak. Time su posve izolovani jedan od drugoga. Komunikacija između servisa se postiže korištenje third-party biblioteke RabbitMQ koja sama po sebi definiše event-driven arhitekturu. Najbitnije za istaknuti kod RabbitMQ-a jeste event-bus koji nudi mogućnost servisima da dodaju poruke u redove (queue, FIFO strukture podataka) i da preuzimaju poruke ukoliko su namijenjene njima.

Servisi se agregiraju u centralnu lokaciju API Gateway koji pojednostavljuje komunikaciju tako što svi pozivi servisima usmjeravaju prema njemu i on ih adekvatno usmjeri. Pored pojednostavljenja pozivanja servisa također se bavi balansiranjem poziva (random pozivanje različitih instanci istog servisa kako bi se smanjilo opterećenje na jedan servis). Pored toga ograničava pozive ukoliko su previše učestali i kešira podatke (kešira odgovore servisa te uslužeje ih klijentima kao takve bez da poziva servis ponovo).

API Gateway je pristupna točka svim klijentima koji poštuju HTTP protokol.



Slika 13. Logicki prikaz (Dijagram)

7. Procesni Pogled

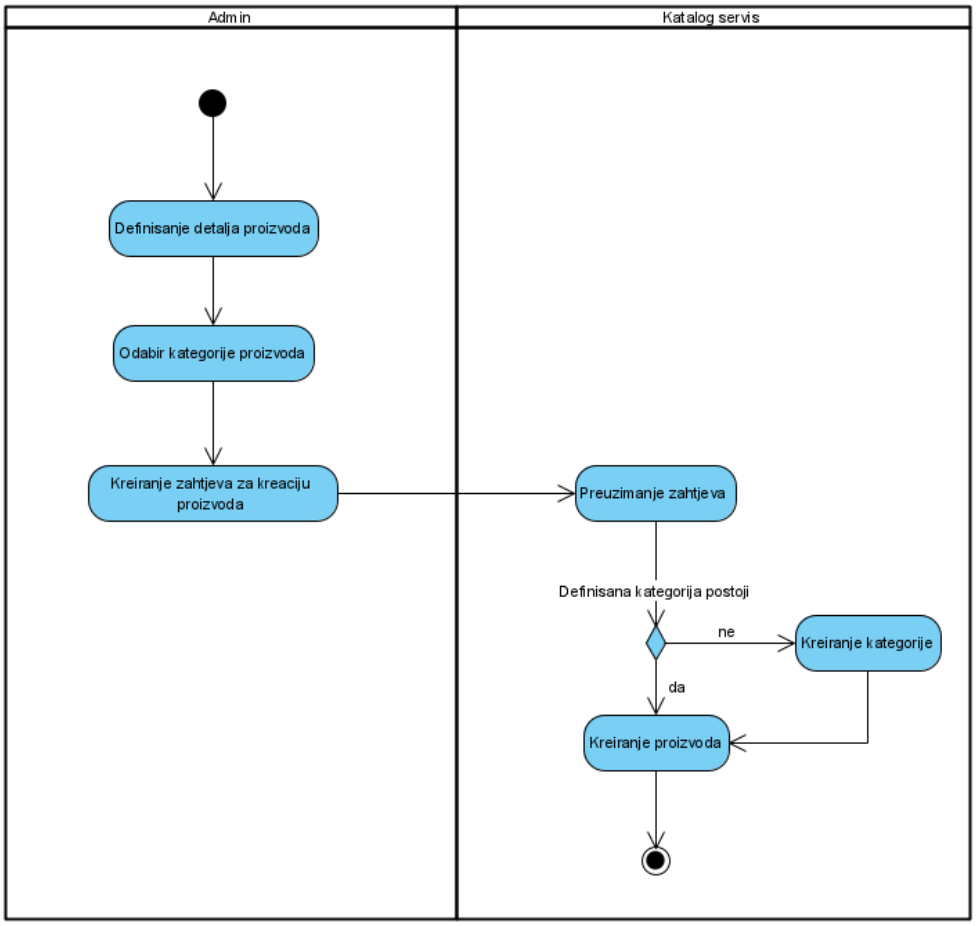
Kako bi prikazali dinamičke aspekte sistema i glavne procese koji se u njemu dešavaju koristit ćemo dijagram aktivnosti. Aktivnosti su zapravo kompozicija manjih cjelina zvane akcije.

Glavne aktivnosti koje ću obrazložiti dijagramom su:

- Upravljanje katalogom
- Upravljanjem korpom
- Upravljanje narudžbama

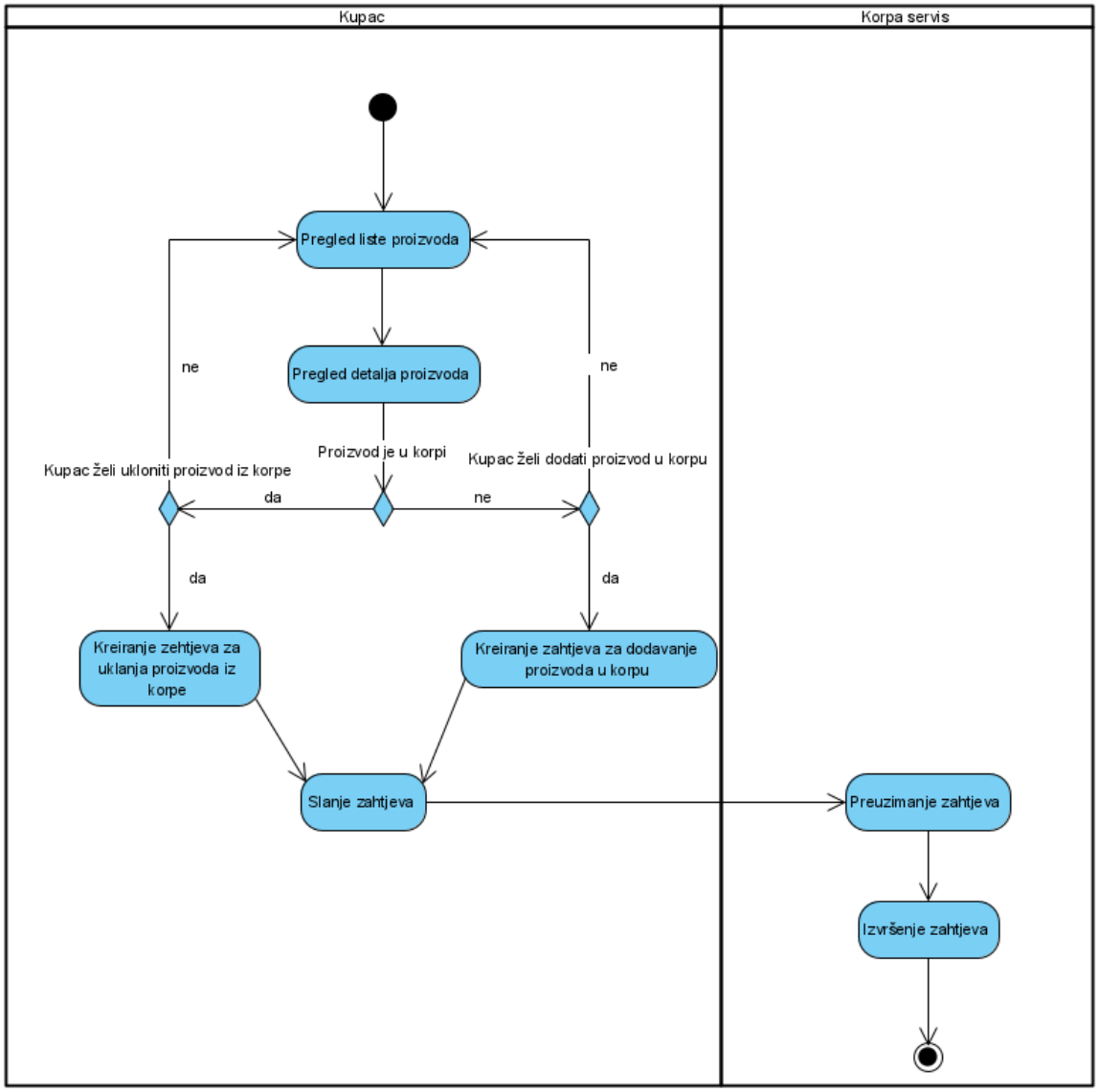
7.1 Upravljanje Katalogom

Bitno je da administratori sistema imaju mogućnost da dodaju novi proizvod kada trgovina počne nabavljati tu liniju proizvoda što ću prikazati na ovom dijagramu. Potrebno je definisati detalje proizvoda kojeg želimo kreirati i njegovu kategoriju u koju pada. Zatim se kreira HTTP POST zahtjev i šalje na katalog servis. Katalog servis preuzima taj zahtjev i na osnovu njega kreira proizvod. Ukoliko definisana kategorija ne postoji ona se kreira.



7.2 Upravljanje Korpom

Kupac bi pri pregledu detalja proizvoda trebao biti u mogućnosti da doda ili ukloni taj proizvod iz svoje korpe. Također ako korisnik ne želi da to učini može se vratiti na pregled liste svih proizvoda i ponoviti isti proces. Dijagram bi se dodatno mogao obogatiti izborom dali želi da nastavi proces upravljanje korpe, odnosno dali se želi ponovno vratiti na pregled liste proizvoda. Zbog kompleksnosti dijagrama sam izuzeo taj dio.



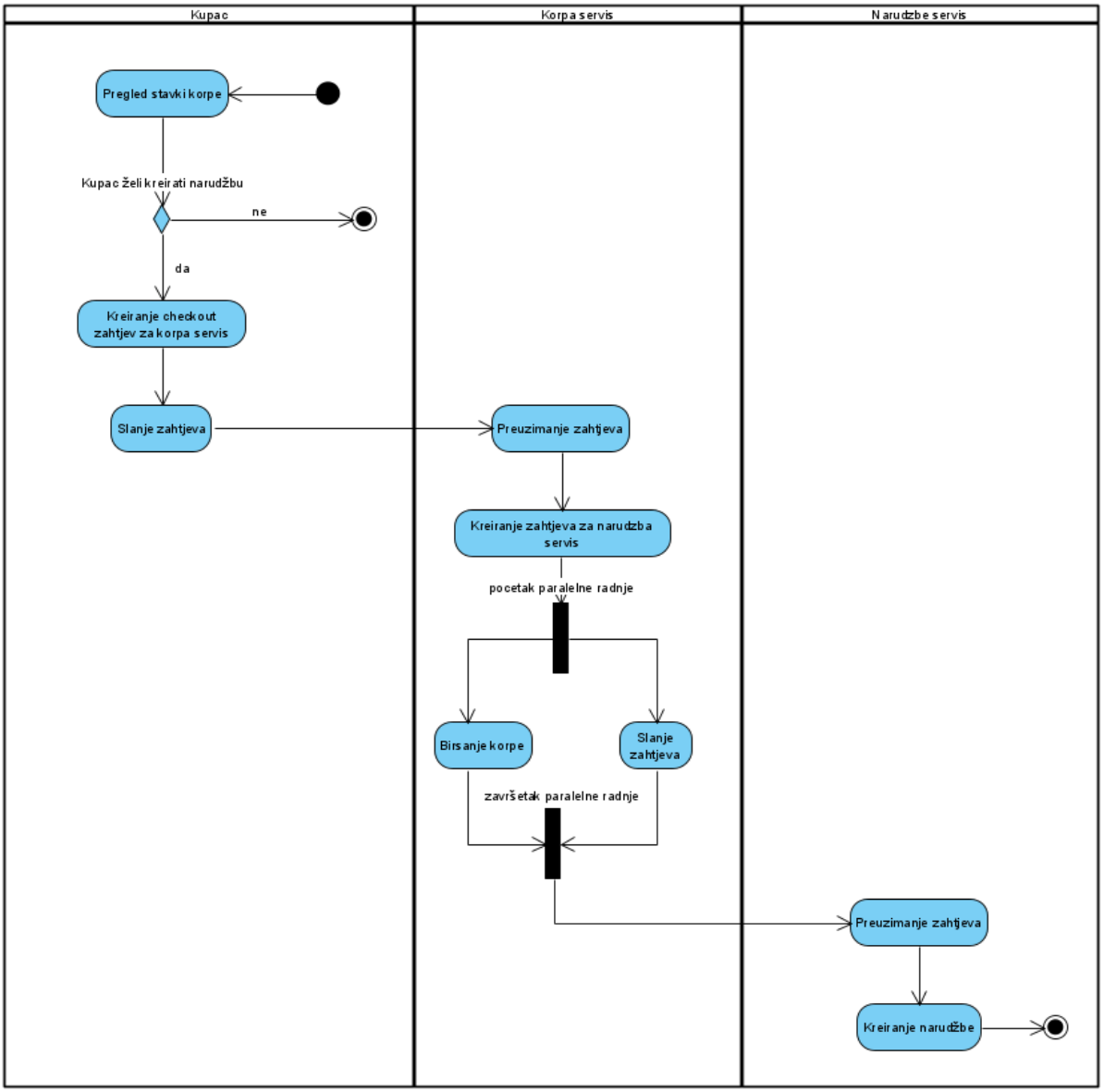
7.3 Upravljanje Narudžbama

Postoje dvije varijante ovog dijagrama. Jedan se odnosi na kupca a drugi na administratora. Zbog toga ću dodatno obrazložiti tačku 3 poglavlja 7 sa 7.3.1 i 7.3.2

7.3.1 Kreiranje narudžbe (Kupac)

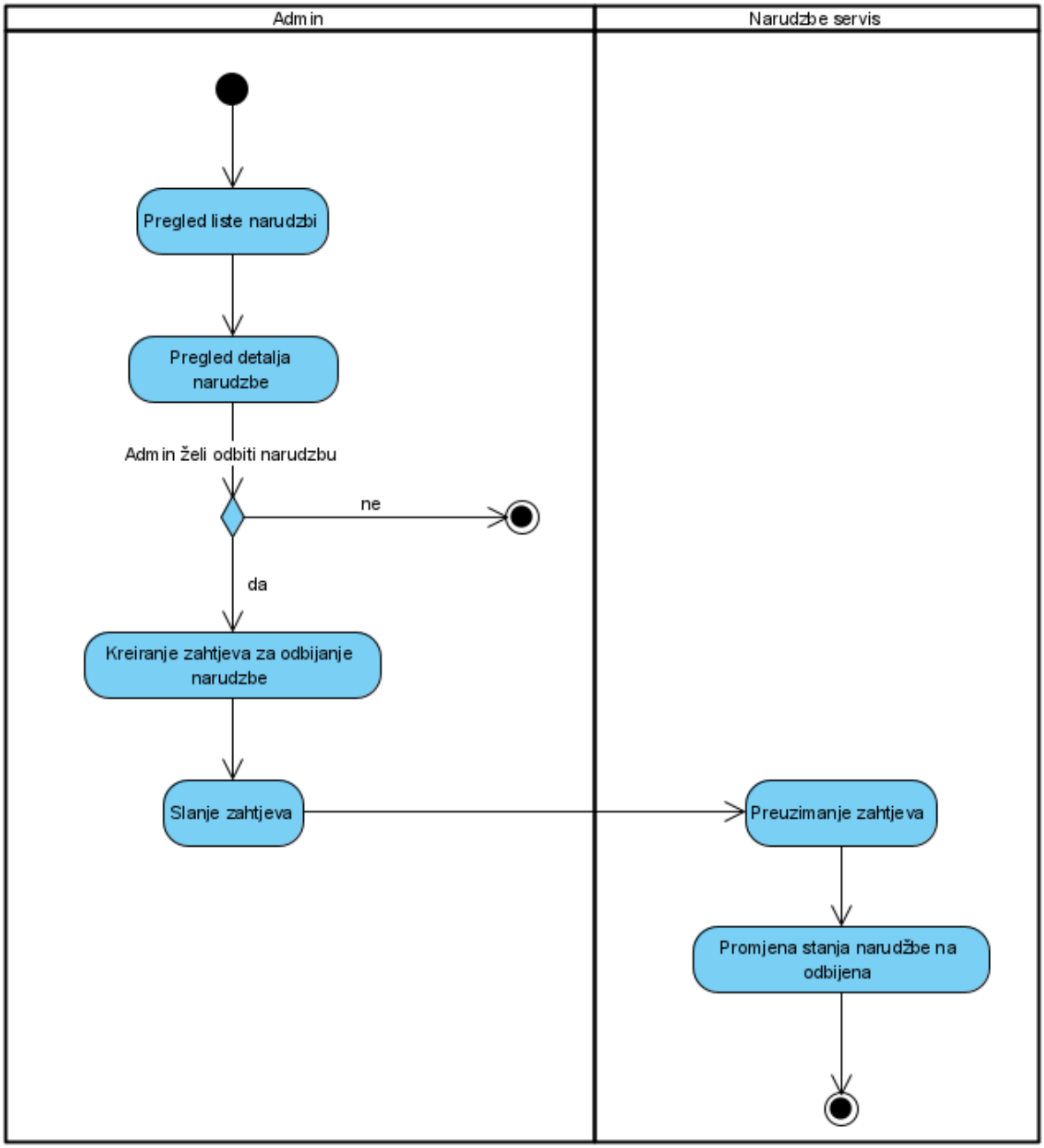
Aktivnost pregled stavki korpe se sačinjava od manjih akcija kreiranje HTTP GET zahtjeva prema korpa servisu, slanje zahtjeva te sa strane korpa servisa preuzimanje i izvršenje zahtjeva.

Checkout zahtjev je zapravo zahtjev kreiranja narudžbe kojeg preuzima korpa servis zatim briše (prazni tu korpu) i šalje dalje zahtjev prema narudzba servisu. Proces slanje zahtjeva i brisanja korpe je paralelna.



7.3.2 Upravljanje Narudžbama (Admin)

Pregled liste narudzbi je aktivnost koja se sastoji od manjih akcija kreiranje, slanje i preuzimanje zahtjeva kao i pregled detalja narudzbe. Nakon uvida u detalje korisnik može da otkaže narudžbu što će promjeniti njeno stanje ali ne i potpuno.

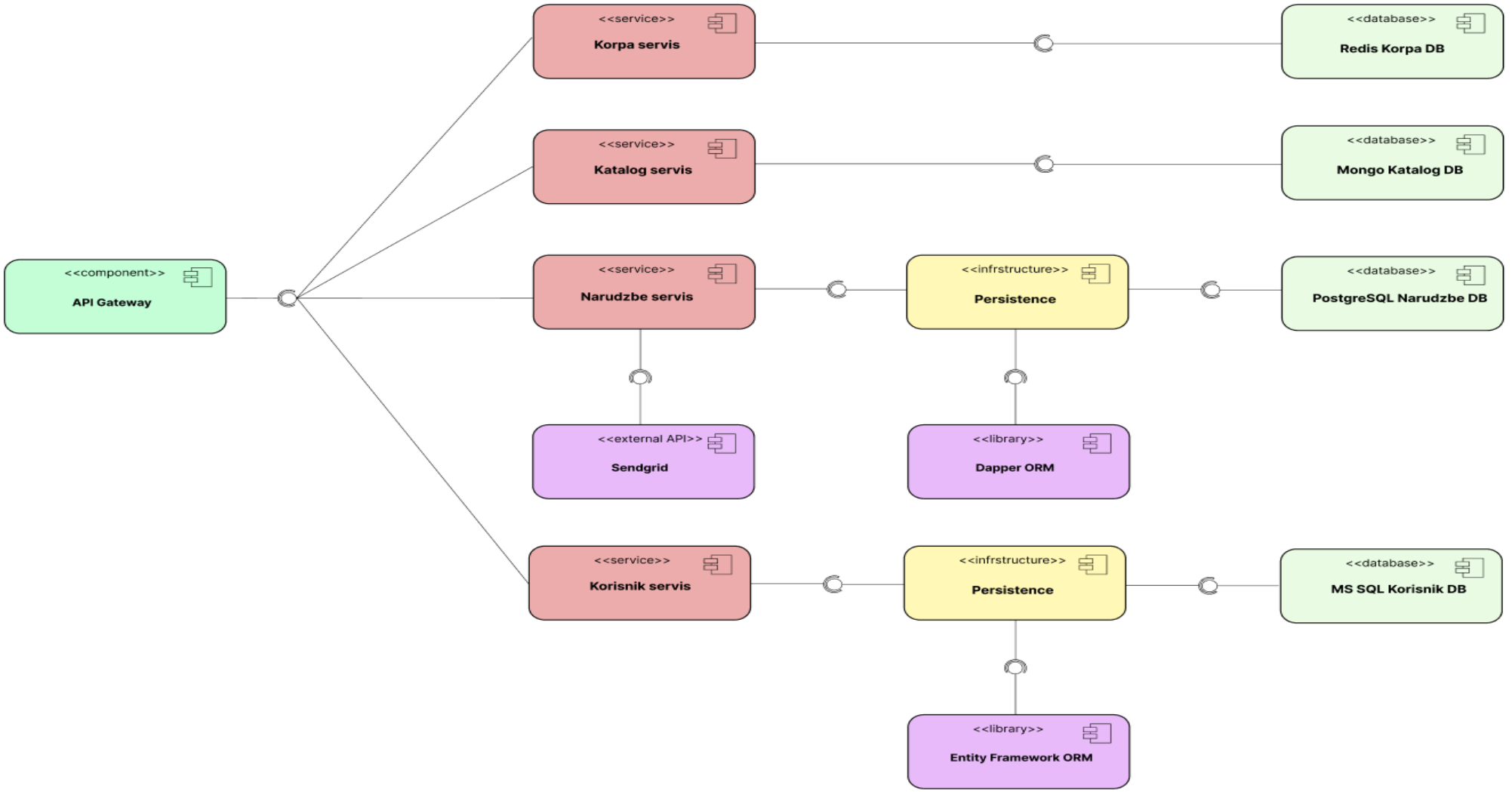


Ostali dijagrami aktivnosti su poprilično slični za ostatak procesa koje se dešavaju pri interakciji sa sistemom pa ih nisam dokumentirao. Inače se radi o generičkom upravljanju gdje se korisniku nudi mogućnost CRUD operacija ne specifičnom setu podataka (formiranje zahtjeva na klijentu, slanje na odgovarajući servis, preuzimanje i obrada zahtjeva na servisu).

8. Implementacijski pogled

Implementacijski pogled prikazuje arhitekturalne odlike vezane za implementaciju. Za predstavljanje ovog pogleda ću koristiti dijagram komponenti.

Glavne komponente ovog dijagrama su : API gateway, servisi, infrastrukturen kompononte, baze podataka i eksterni API.



9. Veličina i performanse

Pošto je moje rješenje eCommerce-a generalizovan moguće je da se brzo nadogradi da podržava veći broj trgovina. To direktno može da utiče na performanse. Jednostavno rješenje bi bilo replikacija istih servisa (isporuka istih servisa više puta) što bi popravilo performanse raspodjelom opterećenja. Pored toga servisi se mogu distribuirati na više servera različitih resursa ovisno o potrebama pojedinih servisa.

Ovo je moguće zahvaljujući modernim DevOps rješenjima kao što je Docker, Azure i Kubernetis. Bitnost skaliranja se nalazi i u vremenu potrebnom za nadogradnju sistema. Kako su servisi izolovana isporuka je lagan a clean arhitektura sama poe sebi nudi benefit testibilnosti kao i skalabilnosti. Kombinacija ove dvije arhitekture su odlične za skaliranje i modifikaciju sistema i sve ostalo što se veže na izmjenu sistema. Također slojevita je lagana za razumjeti pa je brz proces i njene nadogradnje što je korisno ukoliko želimo da povećamo sistem.

10. Kvalitete

Na naj višem nivou se nalazi mikroservisna arhitektura koja nudi mnoge benefite zahvaljujući svojoj modularnosti. Svaki servis se može distribuirati na više različitih servera, te zbog minimalne zavisnosti servisa deployment je veoma lagan.

Određene performanse jesu niže izborom mojih arhitektura za implementaciju projekta ali sam veoma brzim bazama podataka (no sql – redis & mongodb) poboljšao performanse i odziv baza a time i servisa.

Također kako je clean arhitektura poprilično kompleksna naspram ostalih arhitekutra (arhitektura unutar i van mog projekta) po pitanju implementacije, ali dugoročno osigurava skalabilnost, modifibilnost i testibilnost kako je domena odvojena od infrstrukture.

Postoje vjerovatno druge implementacije i arhitekture za domenu eCommerce koje se mogu pokazati kao kvalitetnije, ali ove demonstriraju najpopularnije principe modernog razvoja softvera što zasigurno obećava dugoročnu upotrebljivost godinama unaprijed. Mnogi načini implementacije spomenutih arhitektura se svakodnevno poboljšaju i javno objavljuju što developerima ovog projekta može znatno olakšati posao.

11. Reference

- **Software Architecture in Practice by Len Bass, Paul Clements, Rick Kazman, Parson, 2015.**
- **Software architecture patterns by Mark Richards O'REILLY 2015**
- **Materijali sa predavanja by Dražena Gašpar FIT 2022**
- **Implementing Domain-Driven Design by Vaughn Vernon & Eric Evens**
- **Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evens**