Univerzitet “Džemal Bijedić”

Dokumentacija Softverske Arhitekture
eCommerce Web API

Predmet : Softverska arhitektura
Projekat: eCommcerce Web API
Autor: Armin Smajlagic
Profesor: Dražena Gašpar

Fakultet Informacijskih Tehnologija

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 (Aplikacijskog Programskog interfejsa) koji nudi pristup podacima pohranjenim unutar baza podataka. Dakle prezentacija nije neki standardan GUI već web API koji se lagano integriše u bilo koji GUI.

Motivacija ovog projekta jeste demonstracija najpoznatijih softverskih arhitektura današnjice i to je ujedno 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 trgovine. Ovo se odnosi na inicijalne podatke unutar baze (proizvodi i kategorije) na kojima se dalje mogu generisati drugi podaci (narudžbe, korpe, kupci i sl.).

1.2 Obim

Obim je sam eCommerce odnosno standardne usluge jednog eCommerca prema kupcima i administratorima tog sistema. Interna infrastruktura (arhitektura i njena implementacija) ne posjeduje specifičnu logiku jedne vrste virtualne trgovine što obim znatno proširuje u daljnim upotrebama sistema.

Sistem pokriva jednu virtualnu trgovinu koje je veoma jednostavna za modifikaciju u ponovnu upotrebu u drugim vrstama virtualnih trgovina.

Sistem se dijeli na dva dijela jedan namjenjen za administratore a drugi za kupce.

Administratori kontrolišu cjelokupan sistem odnosne sve podatke koje kupci generišu i vide tokom upotrebe sistema.

Kao što je spomenuto u uvodu kupac je u mogućnosti da pregleda proivode a na osnovu toga dodaje ih u korpu. Ako je zadaovoljan sa sadržajem korpe moće izvršiti „checkout" i time kreirati narudžbu.

Administrator dodaje kategorije i proizvode u odgovarajuće kategorije koje kupac može da pregleda. Također može da pregleda korpe i narudžbe ta da ih ukloni ako to želi. I na kraju administrator ima uvid u sve kupce i njegove detalje (Detaljniji uvid mogućnosti sistema je naveden u tački 4).

1.3 Definicije 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	Podtazumjeva se authentikacija i authorizacija od strane sigurnosnog entiteta unutar sistema (Kod mene je to API Gateway).
Load balancing	Balansiranje opterečenja gdje se pod opterčenjem podrazumjevaju zahtjevi prema API-ju a balansiranje randomizirani algoritam dodjele zahtjeva više instanci istog servisa.

1.4 Zainteresovan strane

Zainteresovane strane su kupci (klijenti) i administratori, odnosno korisnici sistema.

Kupci se registruju na sistem i nakon toga ga koriste kako bi ostvarili kupovinu.

Administratori imaju posebne ključeve sa kojima se njihove mogućnosti podižu u odnosu na kupca i time stiću mogućnost upravljanja sistema na najvišem nivou (podatke).

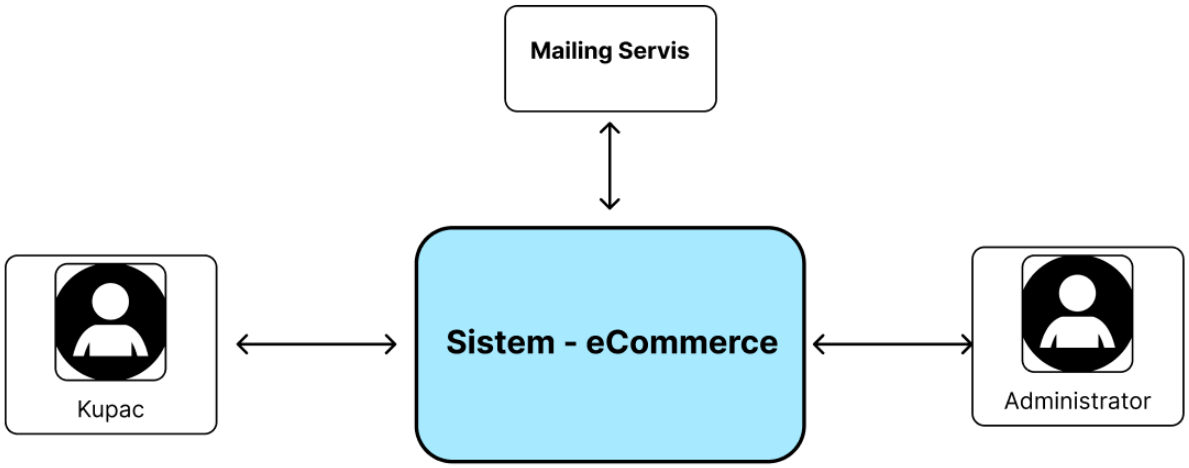
Dalje imamo arhitekte, dizajnere, implementatore (programere) i testere sistema.

Pošto projekat nije korišten nemamo zaintereseranih strana van korisnika i razvijača sistema (odnosi se na menadžment raznih nivoa).

2. Predstavljanje Softverske Arhitekture

2.1 Kontekst Sistema

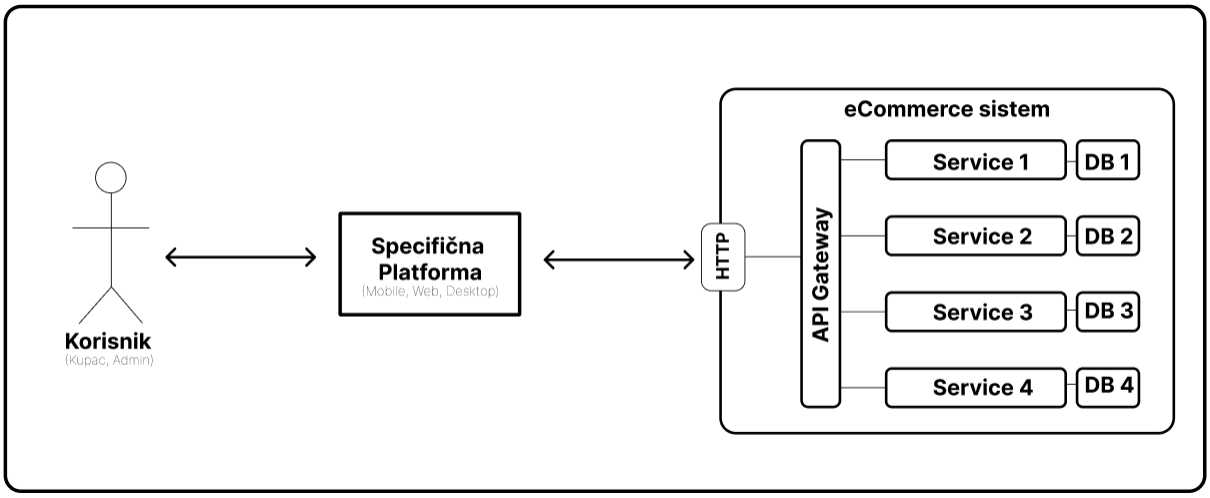
Kontekst sistema uključuje korisnike, sistem i externe aktere kao email servis. Van sistema imam email servis koji služi za potvrdu kreiranja narudžbe. I ako je sistem eCommerce nisam implementirao plaćanje jer su mnogobrojni načini plaćanja, te zahtijevaju vrijeme koje sam ja radije investirao u pravilnu implementaciju arhitektura. Kontekst sistema ću da dodatno obrazložim dijagramom konteksta.



Dijagram Konteksta eCommerce sistema

2.2 Interakcija Korisnika

Interakcija korisnika sistema je unificirana i to preko API-ija. Razlika pristupa koji administratori vrše od kupaca jeste pristupni ključ koji nudi odgovarajuća prava korisnicima. 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.



2.3 Ciljevi Arhitekture

Cilj je da su ponudi rješenje koje je brzo, efikasno i otporno na greške.

Dakle svojstva spomenuta iznad 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 sankcionisati.

Također želimo unificiran i jednostavan pristup za sve klijente, te određeni nivo sigurnosti.

Ovi ciljevi se mogu postići kvalitetnom analizom i formulisanjem potreba klijenata (u narednoj tački 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.

Time smo uzeli u obzir najbitnije ciljeve arhitekture (arhitektura).

3. Karakteristike Softverske Arhitekture

U ovoj tački 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.

3.1 Dostupnost

Sistem treba da bude up and running kada je definisano da bude dostupan. Cilj je omogućiti klijentima usluge bez prekida. Također je moguće da u toku životnog vijeka rješenja sistem padne ali je neophodno da se pravovremeno i vrati u radno stanje (sposobnost recovery-a). Eventualno ako dođe do greške u pokušaju pristupa API-ja od strane klijenata želimo određeni nivo **odpornosti na greške**. To je svojstvo otpornosti na greške (**resiliency**) koji se može postići i na samim klijentima ne ovisno od API-ja. Trebamo uzeti u obzir i klijentske napade tipa (DOS – Denial Of Service) koji može narušiti svojstvo **dostupnosti**. Ovo se može riješiti API Gateway-om (van Web API) ili unutar samog API-ja ograničavanjem HTTP poziva na server u definisanoj jedinici vremena (dublji diskurs u tački 5).

Cilj postizanja karakteristike dostupnosti je minimizirati vrijeme ne dostupnosti servisa.

3.2 Modifibilnost / Skalabilnost

Uzimanjem u obzir vjerojatnoću da će se eventualno sistem izmijeniti i eventualno proširiti potrebno je obratiti pažnju i na sposobnost sistema da se izmjeni i proširi. Na primjer ako želimo dodati novi sistem uplate ili modifikovati već postojeći neophodno je rješenje učiniti dovoljno jednostavnim za **development**, **testiranje** i **isporuku**. Želimo nekako ukloniti zavisnost komponenata (učiniti ga modularnim i nezavisnim o drugim modulima) kako bi eliminirali kompleksnost i olakšali proces razvoja novih ili modifikaciju postojećih komponenti sistema.

Generalno ova svojstva bi se mogla razdvojiti ali ciljem lakšeg razumijevanja potreba, ja sam ih sve grupisao u ovu tačku.

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

3.3 Performanse

Sistem nema prevelike zahtjeve prema performansama jer je u ranoj fazi gdje one nisu problem. Ali u pri skaliranju sistema može doći do redukcije performansi. Performanse se valuta koju trampimo za drugo svojstva/kvalitete kada imamo potreba za njima, tako da je poprilično bitan faktor. Modularnost spomenuta u **tački 3.2** rezultira u padu performansi ukoliko imamo bilo kakve zavisnosti između samih modula pa treba obratiti pažnju pri raspodjeli odgovornosti modulima.

Određene odgovornosti prema ovom svojstvu ne moramo u potpunosti predati arhitekturi softvera. Problemi kao soritiranje proizvoda, pretraga proizvoda kreiranje novih entiteta u bazi možemo predati razvoju samog sistema (korištenjem algoritama manje vremenske kompleksnosti i raznih kreacijskih dizajn paterna). Također možemo povećanje performanse postići korištenjem samog API Gateway-a tako što spremimo odgovore servera i vraćamo spremljene podatke mjesto da ponovno zahtjevamo podatke od servera (tkz. keširanje).

Performanse su valuta koju trapimo za druga svojstva i kvalitete koje želimo da imamo u dugoročnom smislu.

3.4 Deployability

Ovo je dodatak na **tačku 3.2** gdje smo obratili pažnju na isporuku sistema. Ovo je poprilično bitno svojstvo pogotovu u modernim metodologijama rada gdje imamo redovno isporuke sistema. Također ukoliko je neka greška pronađena u kritičnom trenutku bitnost ovog svojstva je veoma bitna. Svojstvo isporuke je usko vezano sa svojstvima razvoja i testiranja, jer ako je lagano bilo razviti rješenje i testirati vjerojatno je bilo lagano i isporučiti. Modularnost je zapravo ključna kada adresiramo ovaj problem. Jer ako želimo izmijenit specifičnu funkcionalnost sistema ona se nalazi u modulu koji je odgovoran za to pa je potrebno samo njega ponovno i isporučiti a ne ponovno čitav monolitni sistem. Također manje isporuke koštaju manje vremena što direktno utiče i na svojstvo dostupnosti sistema (manje je sistem ne dostupan).

4. Značajni use-case-ovi

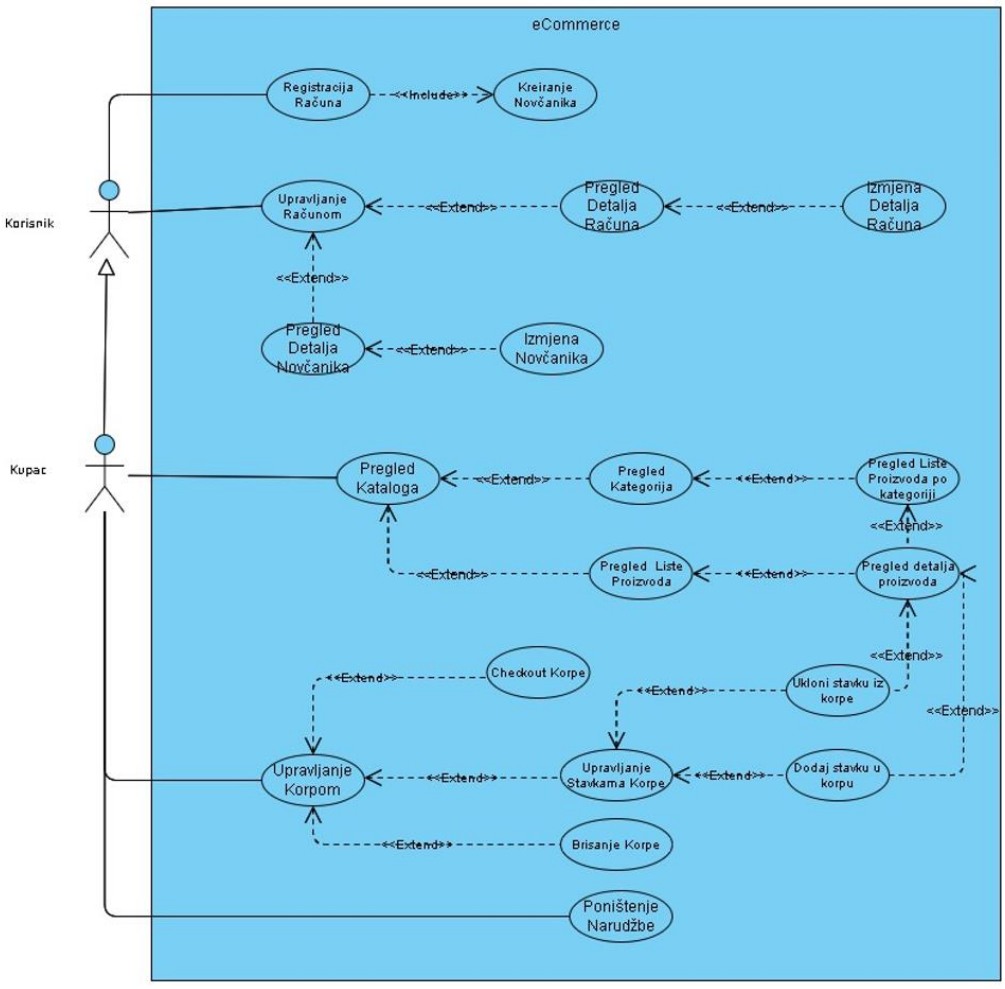
Use-case-ovi se dijele na dvije glavne skupine. Prvi skup predstavlja use-case-ove namijenjene za klijenta a drugi za menadžment (upravljanje podatak koje klijent vidi i generiše).

To ću da obrazložim i demonstriram opisom i use-case dijagramom u tačkama 4.1 Klijent i 4.2 Menadžment.

4.1 Klijent

Klijent se sastoji od nekoliko use-case-ova, i to :

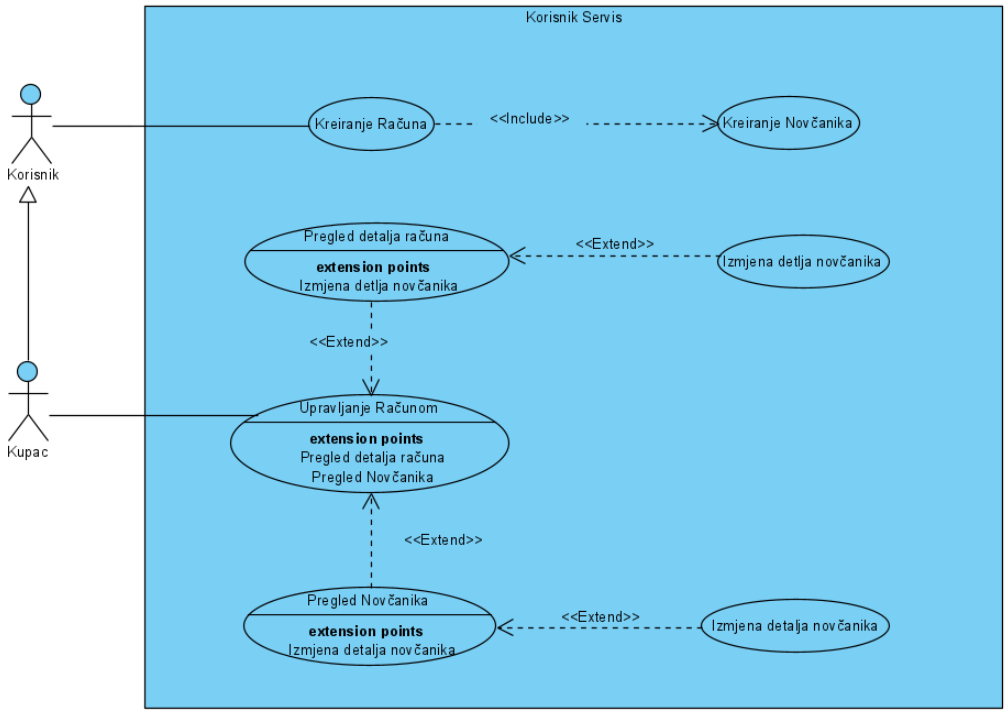
- Kreiranje i upravljanje računom i novčanikom
- Upravljanje korpom i kreiranje\cancel narudžbe
- Pregled kataloga



Cjelokupna klijentska strana (akumlacija svih use-case-ova namjenjenih za kupca

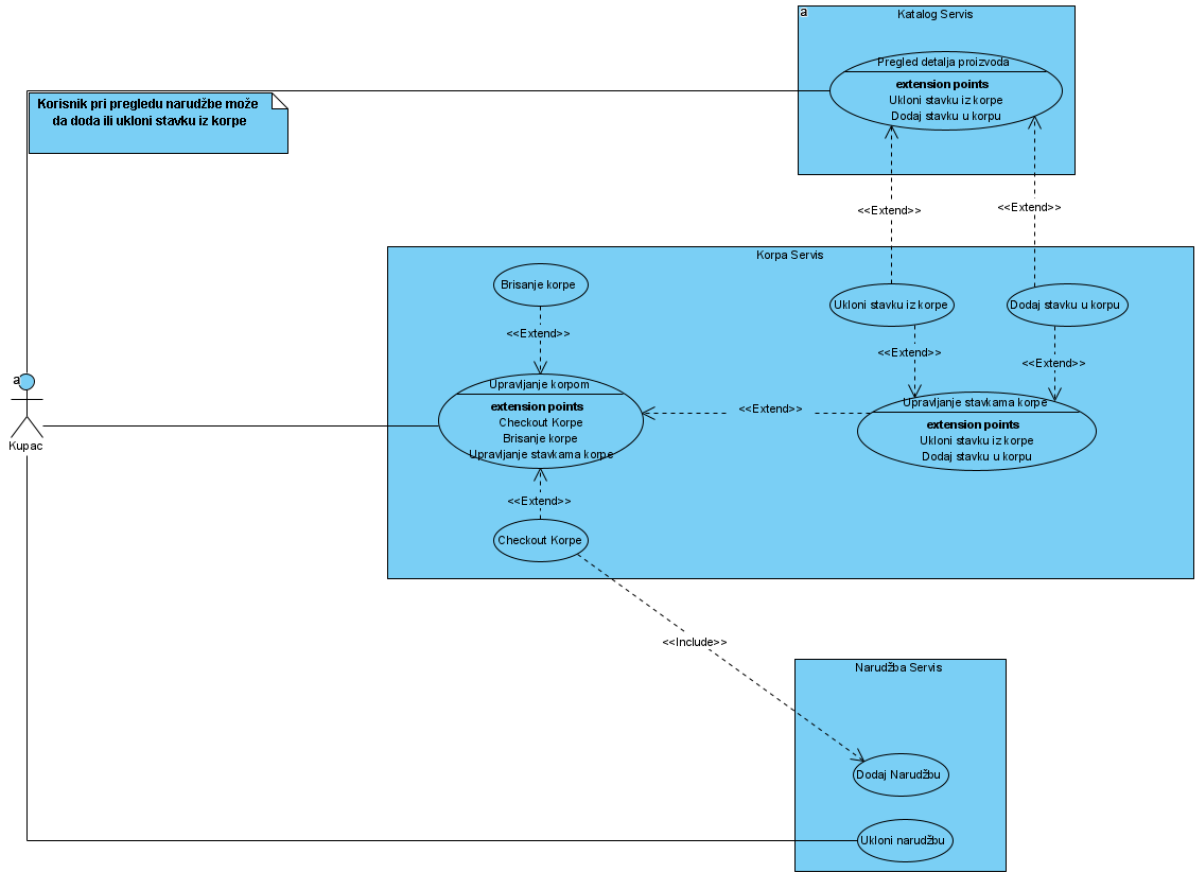
4.1.1 Kreiranje i upravljanje računom i novčanikom

Kupci bi prije svega trebali biti u mogućnosti da kreiraju svoj račun (što automatski kreira i virutalni novčanik) kako bi mogli vršiti kreiranje narudžbi. Pored kreacije također bi trebao moći da pregleda detalje svog računa i novčanika, te eventualno (ne nužno) ih izmjeni.



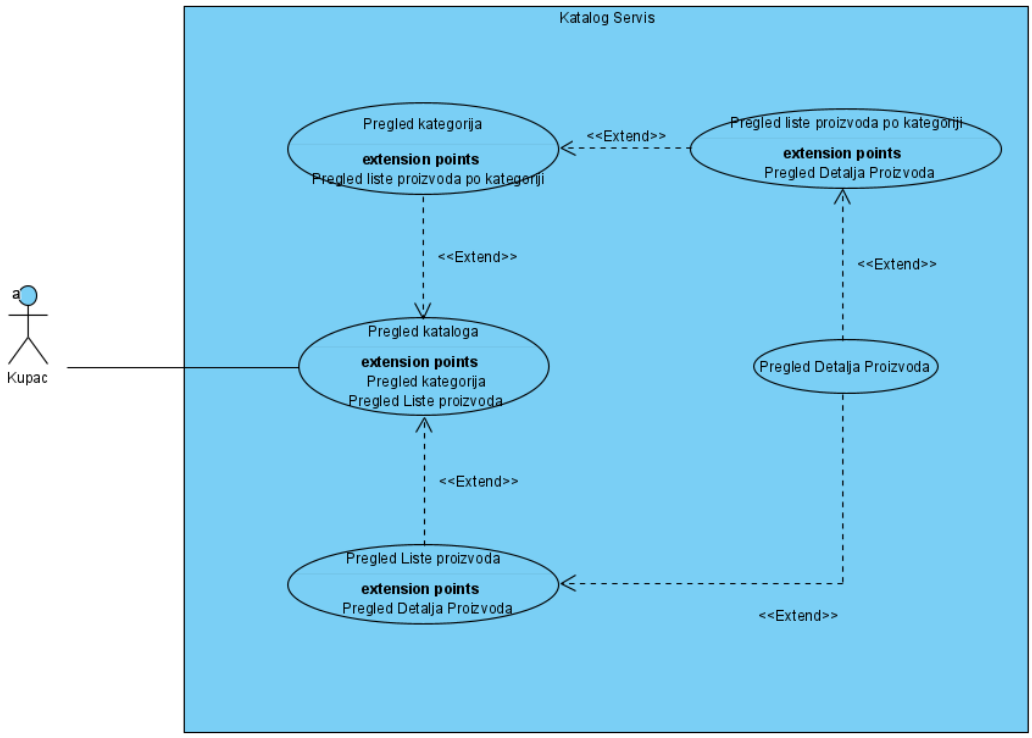
4.1.2 Upravljanje korpom i kreiranje\cancel narudžbe

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.



4.1.3 Pregled kataloga

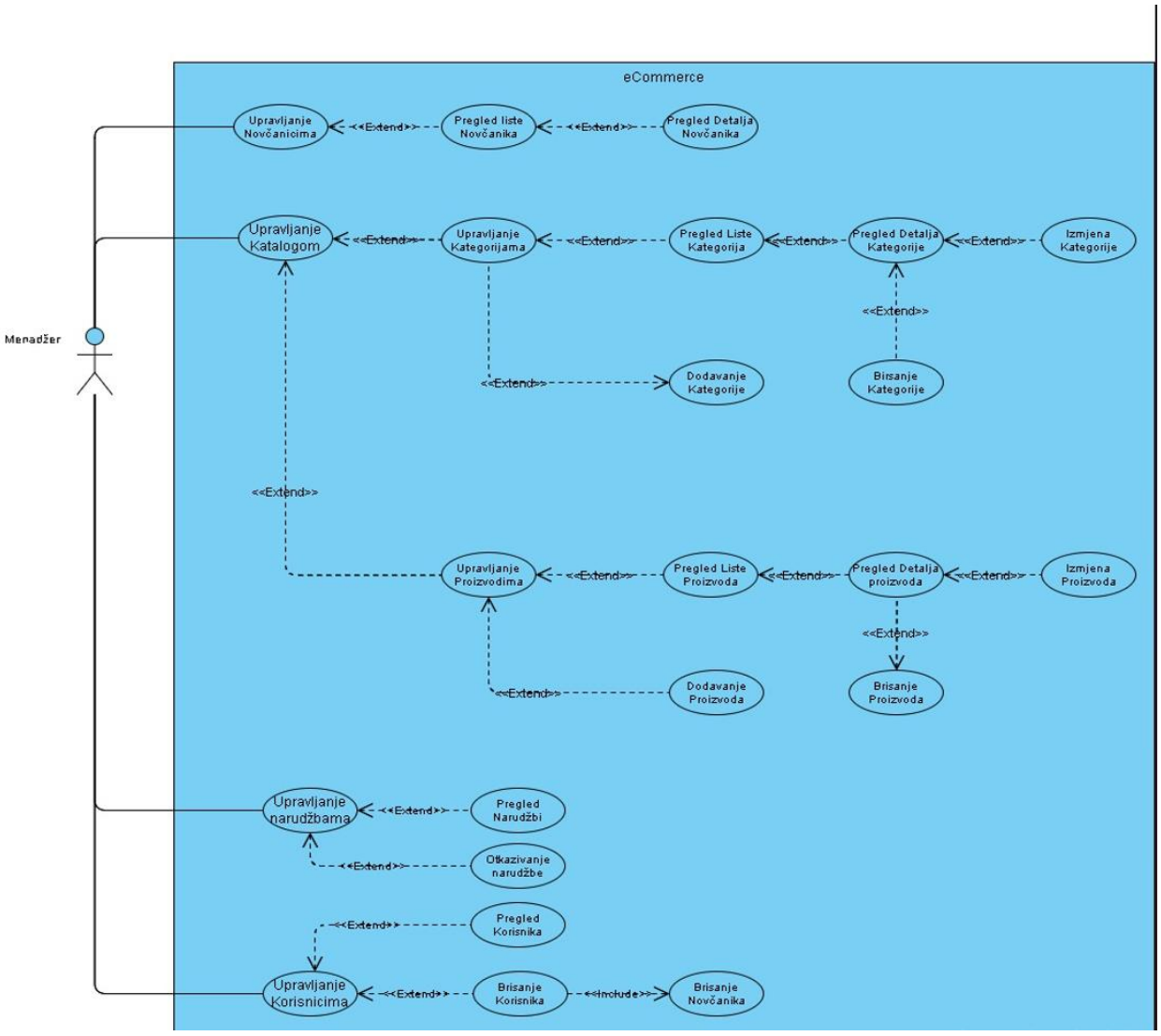
Korisnik bi trebao da pregleda proizvode (opcionalno po kategorijama - filter) prije dodavanja stavki u korpu i kreiranje narudžbe.



4.2 Mendažment

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

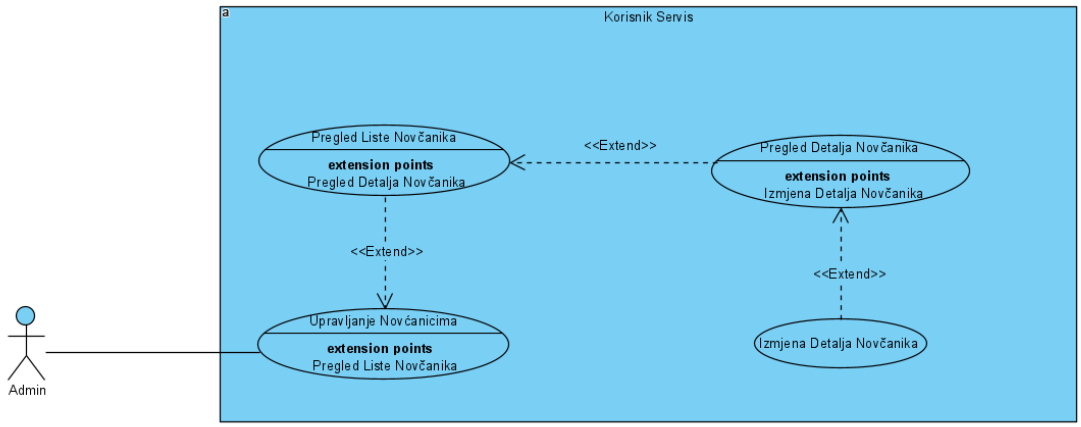
- Upravljanje novčanicima
- Upravljanje katalogom
- Upravljanje Korisnicima
- Upravljanje Narudžbama



Cjelokupna menadžment strana (akumlacija svih use-case-ova namjenjenih za administratore/menadžere)

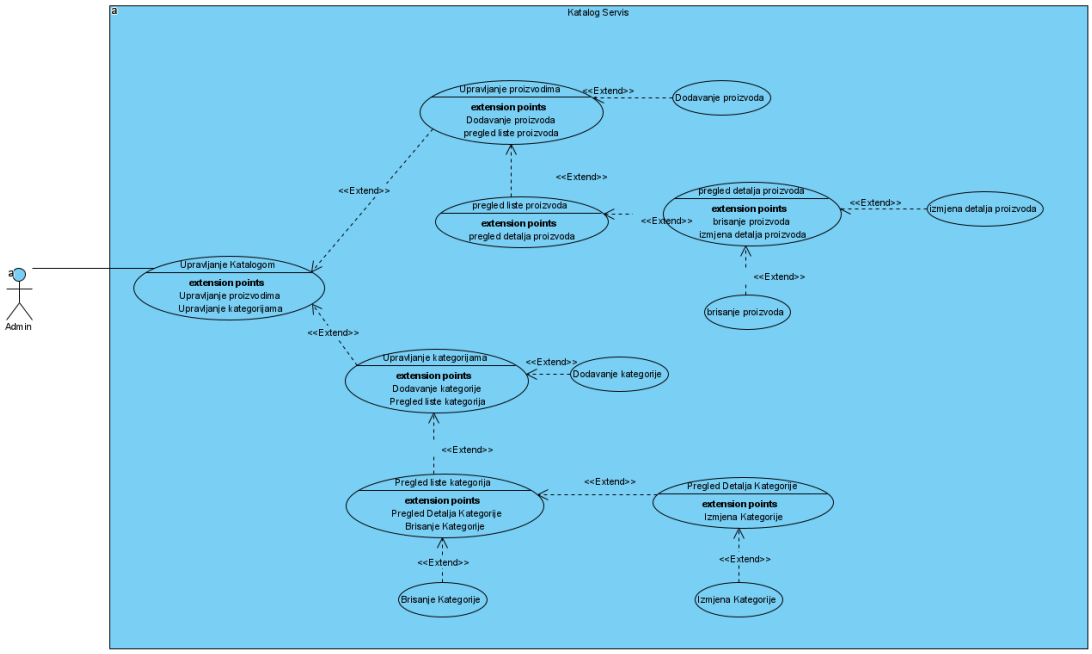
4.2.1 Upravljanje Novčanicima

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



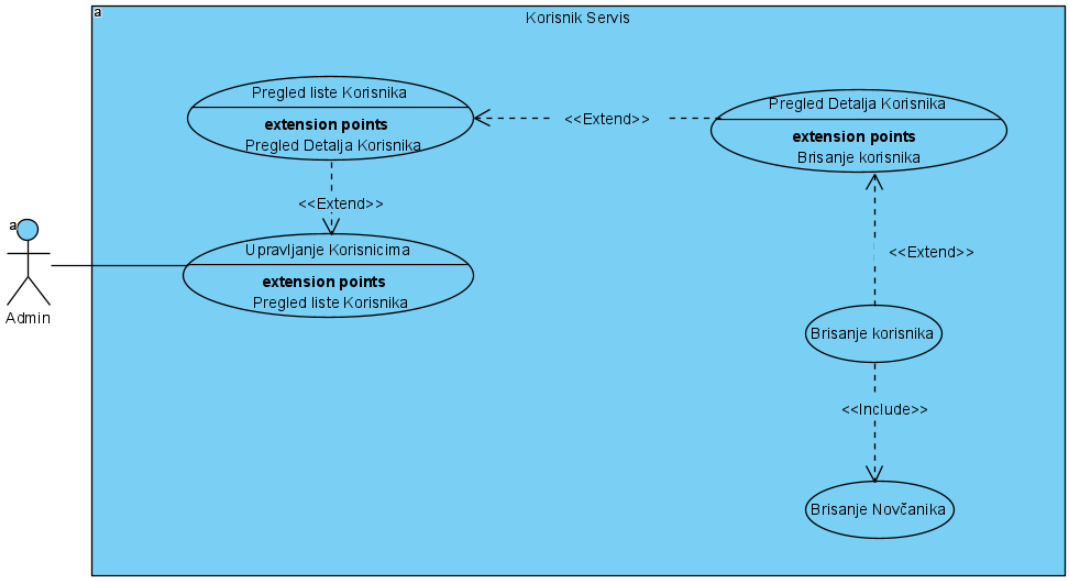
4.2.2 Upravljanje Katalogom

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.



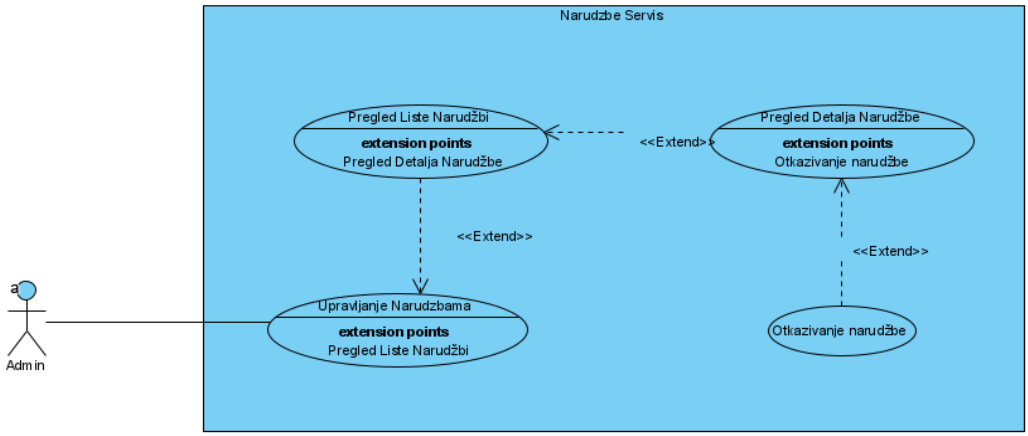
4.2.3 Upravljanje Korisnicima

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).



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 .



5. Vrste (paterni) softverske arhitekture

Vrste softverskih arhitektura koje ćemo analizirati su mikroservisna (REST-based topology), višeslojna (5-Layer) i clean arhitektura. Sve tri imaju određene prednosti i nedostatke te ćemo ih u nastavku analizirati kako bi odabrali onu koja nam najviše odgovara u odnosu na odabrane ključne atribute kvaliteta.

Također se unutar sistema nalazi i event-driven arhitektura u sklopu third-party biblioteke (RabbitMQ) koju neću analizirati jer je nisam ja definisao niti implementirao.

5.1 Paterni i kakarteristike

Mikroservisna arhitektura za cilj ima podijeliti sistem na više manjih servisa koji mogu ali i ne moraju biti ovisni jedan o drugom (ovisnosti servisa ima udar na performanse). Ova arhitektura nudi mnoge prednosti za razliku od monolitnih ukoliko pravilno definišemo gralunarnost i zavisnost. Kako je sistem podijeljen u servise to znatno olakšava **razvoj** i **testiranje** pojedinih servisa (manje kompleksni i manje veličine). **Isporuka** je još jedan benefit zbog toga što možemo samo jedan sistem 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 tkz. Load balancing (raspodjelu opterećenja poziva), odnosno korištenjem randomiziranih algoritama pozivati random instance istog servisa i time smanjiti opterećenje. Ovo utiče na svojstvo **dostupnosti** odnosno povečava dostupnost sistema. Pored ovoga možemo imat zero down time isporuke (isporuka koja ne zahtjeva da sistem bude ne dostupan). Ovo se može postići tako što isporučimo na novo mjesto modifikovani servis i redirektujemo pozive sa klijenta na novu modifikovanu instancu a ugasimo stari.

Slojevita arhitektura je rješenje koje je široko rasprostranjeno radi niskih troškova izrade. Radi se o monolitnoj strukturi koja se sastoji od horizontalnih slojeva gdje svaki ima određenu odgovornost unutar sistema. Zavisno od potreba (specifikacija) broj slojeva može da varira. Bitan koncept zastupljen u ovaj arhitekturi je izolacija slojeva. Dakle svaki sloj je izolovan u smislu njegovog znanja o postajanju drugih slojeva ili njihovoj internoj implementaciji. Na ovo se nadograđuje sposobnost da svaki sloj bude otvoren ili zatvoren za druge slojeve, odnosno utiče na tok korisničkih zahtjeva. Ukoliko je sloj otvoren on ne mora da bude posjećen pri protoku zahtjeva i obrnuto za zatvoreni. Prednosti ove arhitekture su **performanse** i troškovi izrade (lakoća razvoja).

Clean arhitektura je tu kao čisto rješenje za rješavanje raznih problema. Cilj je da se razvijači softvera manje brinu o refakaktoringu standardnog koda (da se ne vračaju na stari kod) jer clean arhitektura enforca pristup rješavanju standardnih problema. Često se koristi uz CQRS dizajn patern (Command and Query Resposnsiblity segragation) kako bi se razdvojile odgovornosti čitanja i pisanja u bazu podataka ili neki drug vid pohrane (npr. In memory). Ideja je da se definiše jezgro koje je specifično za domenu (aplikacijski entiteti i poslovna logika) koja je no

ovisna o pohrani, prstenovanju (API ili Web klijent) i infrastrukturi (pohrana, te mailing, sms servisi i sl.)

I ona se sastoji od slojeva ali ovisnost je drugačija od klasične slojevite te slojevi više podsjećaju na slojeve luka (povrće) nego steka horizontalnih slojeva. Ovisnost slojeva je usmjerena tako da slojevi iznad idu prema jezgru (nukleusu sistema) koji drži domen specifičan programski kod. (Slika slojevite arhitekture je dostupna u **tački 6** Logički pogled). Prezentacijski sloj preuzima korisničke zahtjeve (npr. API ili user interfejs), infrastrukturni po potrebi komunicira sa bazom ili pošalje neki mail, aplikacijski sloj validira korisnički zahtjev, preuzima podatke iz infrastrukturnog sloja po potrebi ih dodatno pripremi i vrati prezentacijskom. Centralni sloj domena (jezgro/nukleus) drži sve entitet i specifičnu ponašanje (use-case-a entiteta).

Rezultat je čist programski kod sa minimalnim potrebama refaktoringa zbog specifične strukture što smanjuje troškove održavanja. Također arhitektura nudi neovisnost o bazi što olakšava njenu izmjenu , neovisnost o interfejsu (UI) što čini lagano izmjenu interfejsa ili upotrebu više njih, testiranje poslovne logike je lakše jer se ne mora čitavo rješenje testirati i neovisno je od aplikacijskog frejmworka što nudi mogućnost upotrebe više različitih biblioteka i tehnologija unutar jednog sistema.

5.2 Ograničena Paterna

Slojevita arhitektura ima ograničenja kao što su otežana skalabilnost i isporuka zbog njene monolitne arhitekture. Također performanse mogu biti oštećene pri propagaciji zahtjeva kroz sve slojeve.

Clean arhitektura i ako asocira na monolitnu nije toliko ograničena kao slojevita radi njene ne ovisnosti o slojevima, ali ima sličan udar na performanse. I ako ne ovisnost ostatka aplikacije od UI-a ili baze podataka, koliko često zapravo mi mijenjamo bazu ili UI. Dakle i ako je to benefit ne koristi se tako često. Isporuka može imati određene poteškoće ako dodamo novu funkcionalnost u sistem jer moramo izvršiti izmjenu kroz čitav tkz. vertikalni slice (isječak), dakle svaki sloj mora biti modifikovan na specifičnim dijelovima te se svaki mora ponovno isporučiti.

Mikroservisna nudi mnoge benefite kojima se može nadomjestiti za nedostatke kao što su performanse. Kako je sistem distribuiran na više manjih servisa bila kakva ovisnost jednog servis o drugi pri usluživanju korisničkog zahtjeva mogu imati udarac na performanse. Također ne možemo standardno pristupiti sigurnosti jer sistem ima više ulaznih tačaka.

5.3 Odluke i obrazloženja

Sve arhitekture koje sam naveo iznad zajedno sa određenim modifikacijama mogu biti odlično rješenje za moj eCommerce projekat.

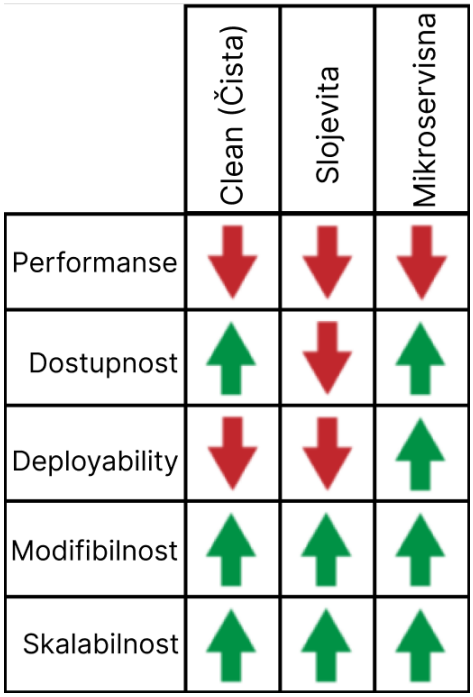
Sigurnost i performanse se mogu uskladiti sa dodatnim tehnologijama. Event driven arhitektura koja je poznata po svojim performansama se može uključiti u rješenje korištenjem dodatne tehnologije RabbitMQ koja se bavi servisnom komunikacijom. Svaki event (događaj) se može poslati u event-bus te zainteresovani mogu istim tim busem i preuzeti je. Time se smanjuje udar na performanse i tight-coupling servisa. 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) ako je stavimo u jedan servis unutar mikroservisne arhitekture može se malo korigovati a problemi sa dostupnosti potpuno eliminisati. U ovom slučaju slojevita bi bila pod arhitektura unutar mikroservisne. Isti problemi pronađeni u clean arhitekturi se mogu na isti način optimizovati.

Svaki servis se može instancirati više puta korištenjem Docker-a, dakle isporučiti isti servis više puta i time poboljšati fault tolerance i dostupnost.

Clean arhitektura i ako ima specifičnu strukturu koja otežava development, poboljšava pristup rješavanja standardnih problema oko upravljanja korisnicima i narudžbama koje padaju u domenu problema eCommerce sistema kojim se bavim u ovoj dokumentaciji.

I finalno glavni benefit mikroservisne arhitekture i event-driven komunikacije imidžu servisa nudi adaptibilnost prema drugim tehnologijama. Npr. Ako želim Java Spring API dodati kao novi servis te komunicirati sa drugim .NET servisima unutar sistema ja nebi imao problem jer je komunikacija standardizirana unutar evet-driven arhitekture RabbitMQ-a.



Slika 10.1 Vizualna ilustracija bitnih kvaliteta arhitektura

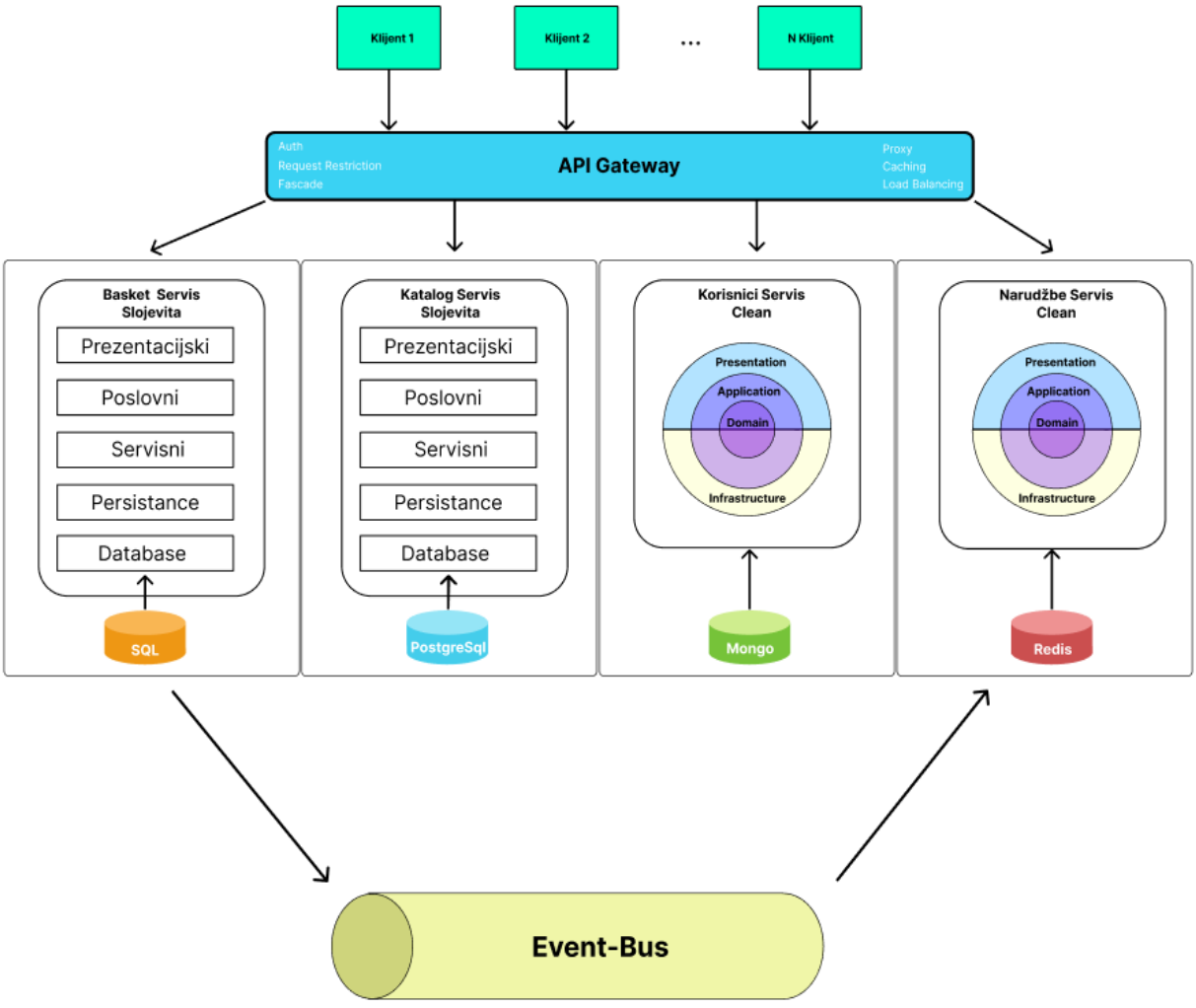
6. Logički pogled

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 6.1 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.

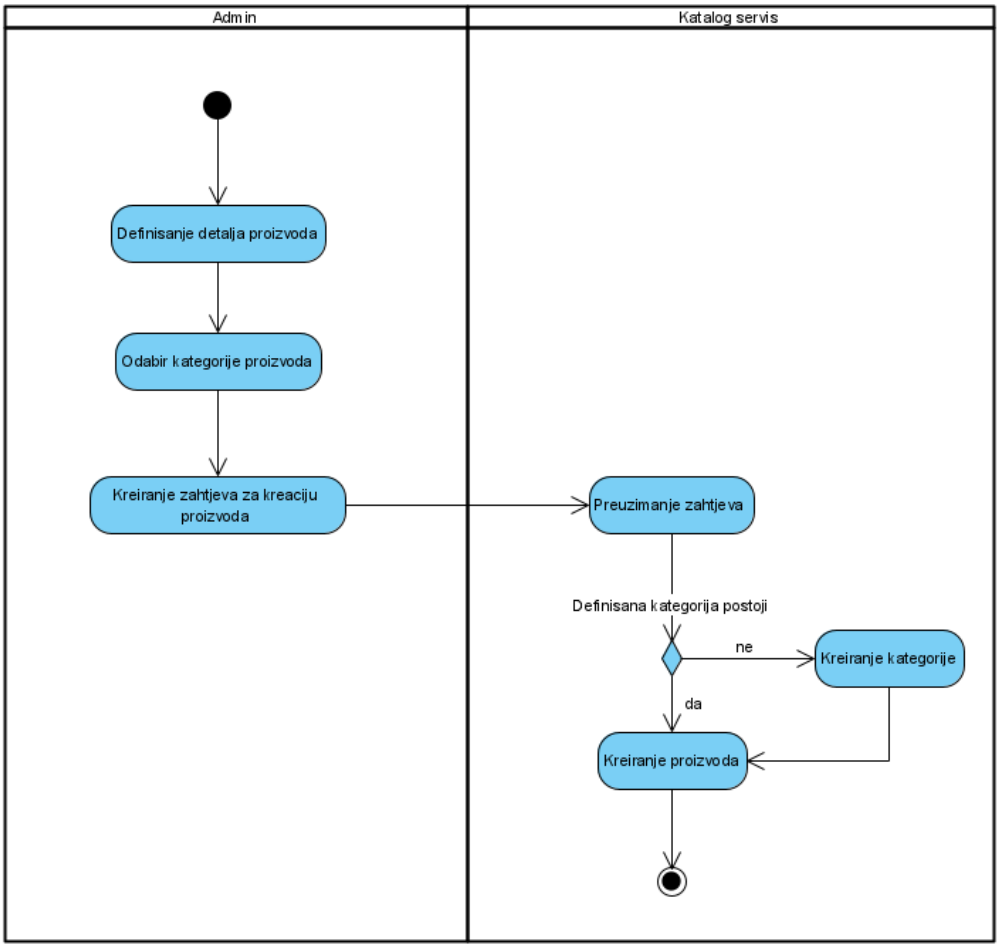
Glavne aktivnosti koje ću obrazložiti dijagramom su:

- Upravljanje katalogom
- Upravljanjem korpom
- Upravljanje narudžbama

7.1 Upravljanje Katalogom

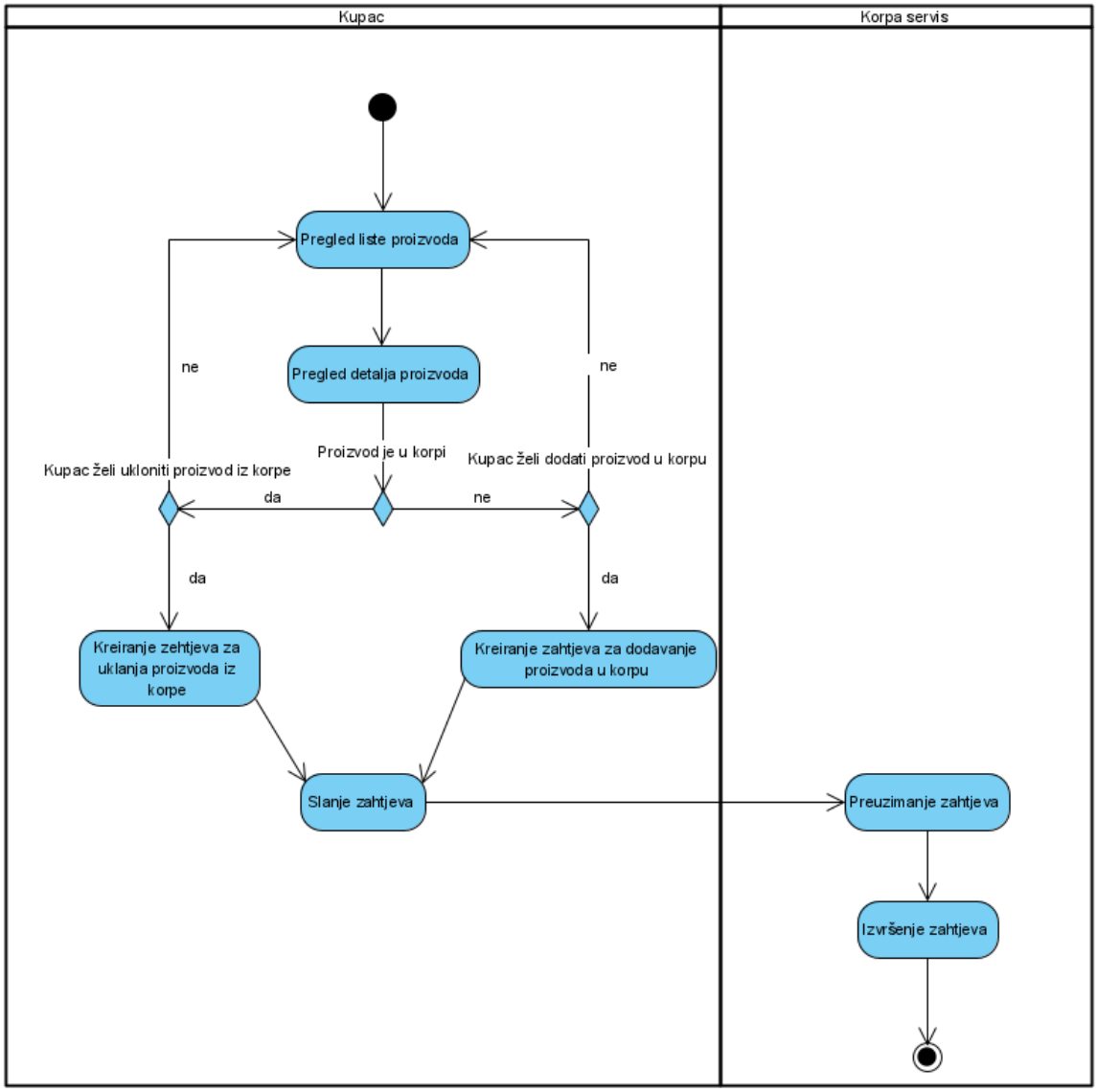
Bitno je da administratori sistema imaju mogućnost da dodaju novi proizvod kada trgovina počne nabavljati tu liniju proizvoda što ću prikazati na ovom dijagramu.

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.



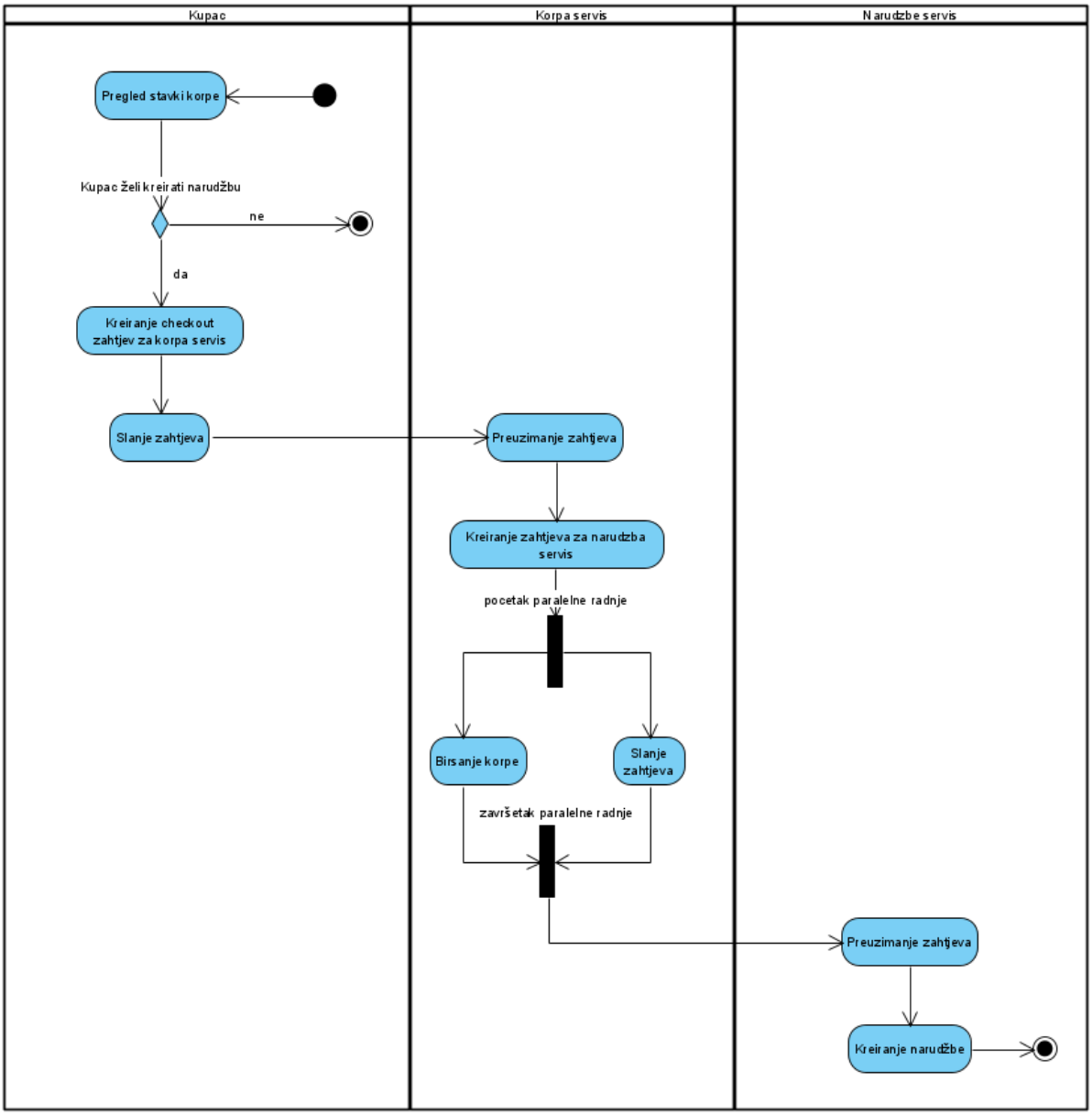
7.3 Upravljanje Narudžbama

Postoje dvije varijante ovog dijagrama. Jedan se odnosi na kupca a drugi na administratora. Zbog toga ću dodatno obogatiti ovu tačku sa 7.3.1 i 7.3.2

7.3.1 Kreiranje narudžbe (Kupac)

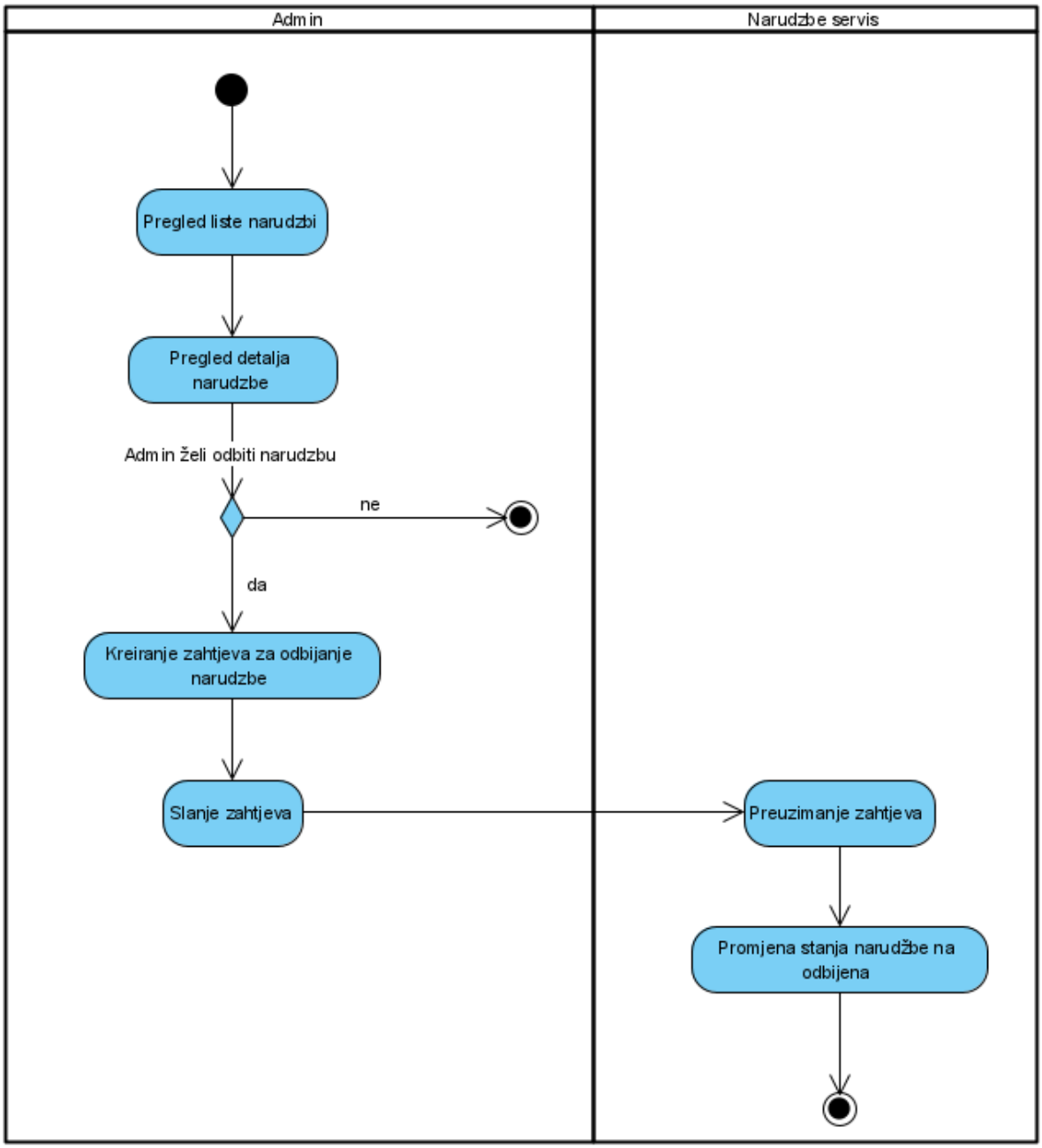
Aktivnost pregled stavki korpe se sačinjava od manjih akcija kreiranje HTTP GET zahtjeva prema korpa servisu, slanje zahtjeva te sa strane korpa servisa preuzimanje i izvršenje zahtjeva.

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 je obrisati.

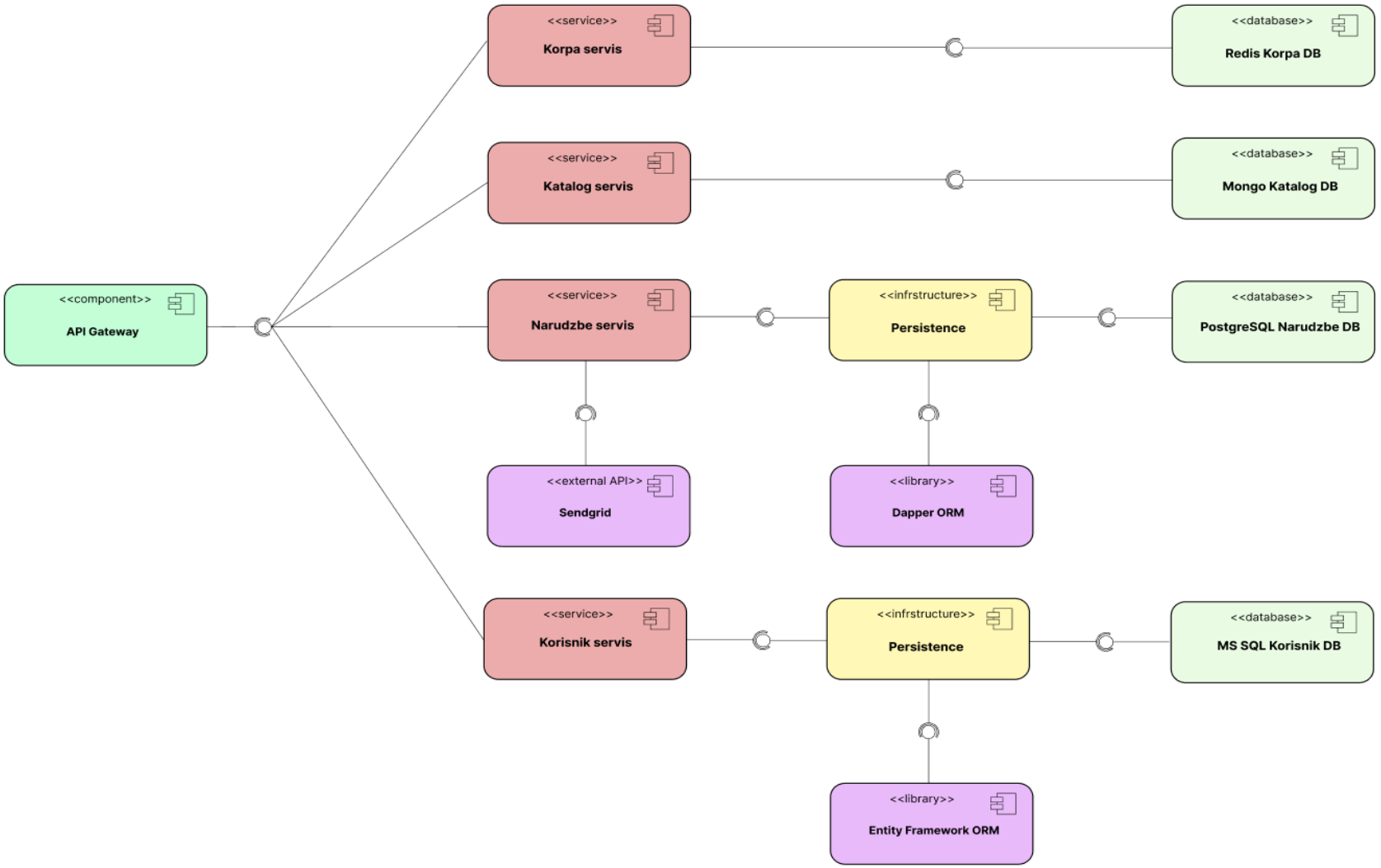


Ostali dijagrami aktivnosti su poprilično slični za ostatak procesa koje se dešavaju pri interakciji sa sistemom pa ih nisam dokumentirao. Inače se radi o generičkom upravljanju gdje se korisniku nudi mogućnost CRUD operacija ne specifičnom setu podataka (formiranje zahtjeva na klijentu, slanje na odgovarajući 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, a cjelokupan proces replikacije je brz i jednostavan. Samim time veličina aplikacije i performanse se mogu znatno promijeniti po potrebama klijenta.

Ovo je moguće zahvaljujući modernim DevOps rješenjima kao što je Docker, Azure i Kubernetis. Sama arhitektura može ograničiti skalabilnosti i performanse što u ovom slučaju nije zbog mikroservisne arhitekture.

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 bazamao podataka (no sql – redis & mongodb) poboljšao performanse i odziv servisa.

Također kako je clean arhitektura poprilično kompleksna naspram ostalih arhitekutra (arhitektu ra unutar i van mog projekta) po pitanju implementacije, development jeste otežan, ali kada se prevaziđu te poteškoče na development strani dobijamo mnoge kvalitete.

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.

Same kvalitete softverskog rješenja eCommerce su dublje pređene u tački 5.3 gdje imamo vizalnu ilustraciju šta donosi ovaj izbor arhitektura po pitanju kvaliteta.

11. Reference

[1] Software Architecture in Practice (Third Edition),
Len Bass, Paul Clements, Rick Kazman, Parson, 2015.

[2] Materijali sa predavanja,
Dražena Gašpar, FIT, 2020.

[3] Software architecture patterns
Mark Richards, O'REILLY, 2015.

[4] Wikipedia