UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Aleksandra D. Dotlić

IMPLEMENTACIJA AKTOR MODELA U RAZVOJU SISTEMA ZASNOVANIH NA MIKROSERVISNOJ ARHITEKTURI

master rad

Beograd, 2023.

Mentor:

dr Vladimir Filipović, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Saša Malkov, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Aleksandar Kartelj, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Divni i Danilu

Naslov master rada: Implementacija Aktor modela u razvoju sistema zasnovanih na mikroservisnoj arhitekturi

Rezime: U ovom radu analizira se primena aktor modela i mikroservisne arhitekture u razvoju reaktivnih sistema. Implementacija ovih koncepta prikazana je na konkretnom primeru aplikacije za internet prodavnicu, realizovane u .NET okruženju korišćenjem Akka.NET biblioteke. Fokus je na primeni aktor modela i reaktivnih principa u mikroservisnoj arhitekturi, definisanju jasnih granica između mikroservisa, kao i na njihovoj međusobnoj komunikaciji kroz mehanizam Akka.Cluster. Takođe, u radu su analizirane prednosti i izazovi ovakvog pristupa, uključujući modularnost, skalabilnost, otpornost na greške i lakše održavanje sistema. S obzirom na to da je primena Akka.NET i Akka.Cluster za implementaciju mikroservisa u praksi retka i dokumentacija oskudna, cilj rada je da analizira implementaciju, identifikuje prednosti i izazove, i pruži relevantne informacije koje bi mogle doprineti unapređenju dokumentacije za ovu oblast.

Ključne reči: Aktor model, Akka.NET, reaktivni sistemi, mikroservisi

Sadržaj

Glava 1

Uvod

U današnjem digitalnom svetu, postoji velika potreba za izgradnjom složenijih sistema koje odlikuje skalabilnost, elastičnost, visoka odzivnost i otpornost na greške. Sistemi dizajnirani na ovakav način, tako da bolje reaguju na svoje okruženje i na druge sisteme, nazivaju se reaktivni sistemi [10]. Tradicionalni pristupi razvoju softvera često se suočavaju sa izazovima u razvoju ovakvih sistema.

Jedan od ključnih koncepata koji pruža moćan mehanizam za izgradnju reaktivnih sistema je aktor model.

Aktor model predstavlja matematički model za konkurentna izračunavanja koji je zasnovan na konceptu „aktora", koji predstavljaju nezavisne entitete koji međusobno komuniciraju razmenom poruka [12]. Inspirisan je teorijom paralelnog i distribuiranog računarstva, ali je njegova implementacija i šira upotreba dugo ostala ograničena zbog tehničkih izazova i nedostatka odgovarajućih radnih okvira. Kroz godine, razvoj aktor modela se ubrzao, a radni okviri poput Akka.NET [5], Microsoft Orleans [2] i Proto.Actor [4]. su se pojavili kako bi pružili moćne alate za implementaciju aktor modela u praksi.

Sa druge strane, kao najpopularniji model arhitekture softvera poslednjih godina, izdvaja se mikroservisna arhitektura. Mikroservisna arhitektura predstavlja modernu arhitekturu za razvoj softvera koja se fokusira na organizaciju sistema u skup manjih servisa, tzv. mikroservisa. Mikroservisi su nezavisne komponente koje pružaju usluge drugim servisima i sa njima komuniciraju. Cilj mikroservisne arhitekture je da se poveća fleksibilnost i proširivost aplikacije podelom na male labavo vezane servise koji se mogu nezavisno razvijati i skalirati [13].

Kombinacija aktor modela i mikroservisne arhitekture pruža prednosti kao što su modularnost, skalabilnost, otpornost na greške i lakše održavanje sistema. Svaki

mikroservis može biti nezavisan i može se razvijati timski, što omogućava brže iteracije i fleksibilnost u razvoju. Aktor model dodaje asinhronu prirodu komunikacije i granularno upravljanje stanjem, što vodi ka boljoj reaktivnosti i visokoj otpornosti sistema na opterećenje [9].

U ovom radu se prikazuje implementacija navedenih koncepata na konkretnom primeru aplikacije za internet prodavnicu. Aplikacija je implementirana u .NET okruženju [3] korišćenjem Akka.NET biblioteke. Kroz implementaciju sistema je detaljno razmotreno kako se aktor model i reaktivni principi primenjuju u implementaciji mikroservisne arhitekture. Takođe, razmotreno je i kako modelovati takav sistem, tj. podeliti odgovornosti i napraviti jasne granice između mikroservisa. Rad obuhvata i opis komunikacije između mikroservisa putem aktora pomoću mehanizma Akka.Cluster. Primena Akka.NET i Akka.Cluster za implementaciju mikroservisa u praksi je relativno retka, a dokumentacija je prilično oskudna. Stoga, cilj je analizirati ovakvu implementaciju i njene prednosti i izazove, kao i pružiti relevantne informacije i ukoliko je moguće, doprineti unapređenju dokumentacije za ovu oblast.

Glava 2

Aktor model

Aktor model predstavlja matematički model za konkurentna izračunavanja. Baziran je na konceptu „aktora" kao nezavisnih jedinica izvršavanja čija se međusobna komunikacija odvija razmenom poruka [12]. Nastao je sedamdesetih godina prošlog veka a njegov kreator je Karl Hjuit (*eng. Carl Hewitt*). Kasnije, osamdesetih godina prošlog veka, ovaj model je dobio na popularnosti u akademskim krugovima kao efikasan način za programiranje distribuiranih sistema. U devedesetim godinama postao je deo mnogih industrijskih sistema i programskih jezika, primer takvog jezika je Erlang. Danas, aktor model je postao veoma značajna tehnologija za razvoj modernih distirbuiranih sistema.

Postoji nekoliko radnih okvira koji su zasnovani na aktor modelu, a neki od njih su Akka.NET, Microsoft Orleans i Proto.Actor. Svaki od njih ima svoj pristup i filozofiju u implementaciji aktor modela. Akka.NET pruža fleksibilnost u konfiguraciji i upravljanju životnim ciklusom aktora, Orleans se fokusira na automatizaciju raspodele i skaliranje kroz koncept virtuelnih aktora, dok Proto.Actor pruža minimalistički pristup i jednostavnost u implementaciji aktorskog modela.

Aktor model je osmišljen kao alternativa klasičnom modelu izračunavanja sa deljenim stanjem i sinhronim pozivima funkcija. Umesto toga, ovaj model je ponudio asinhronu komunikaciju između nezavisnih jedinica izvršavanja koje izbegavaju blokirajuću sinhronizaciju i mogu da funkcionišu paralnelno. Korišćen je i kao radni okvir za teorijsko razumevanje konkurentnosti i kao teorijska osnova za implementaciju nekih konkurentnih sistema.

Glavne karakteristike aktor modela su:

- **Izolacija stanja.** Aktori su samostalne jedinice izvršavanja koje čuvaju svoje unutrašnje stanje i ne dele ga sa drugim aktorima, osim ukoliko nekom aktoru

ne pošalju poruku.

- **Asinhronst.** Aktori međusobno komuniciraju slanjem poruka što omogućava visok stepen asinhronosti.

- **Visok nivo apstrakcije.** Aktor model pruža visok nivo apstrakcije koji olakšava razvoj distribuiranih sistema. Umesto da programeri moraju da brinu o detaljima upravljanja procesima i sinhronizaciji podataka, aktor model omogućava da se fokusiraju na rešavanje poslovnih problema.

- **Tolerancija na greške.** Svaki aktor je samostalna jedinica izvršavanja koja može da bude skalirana i replikovana, što znači da ako jedan aktor ne radi kako treba, drugi mogu da preuzmu njegovu ulogu.

- **Garancija dostave poruka.** Aktor model pruža mehanizme koji omogućavaju sigurnu i pouzdanu dostavu poruka između aktora. To obezbeđuje da poruke budu isporučene na odgovarajući način i da se izbegnu problemi kao što su gubljenje poruka ili dupliranje njihove obrade. Postoje tri osnovna nivoa garancije dostave poruka:

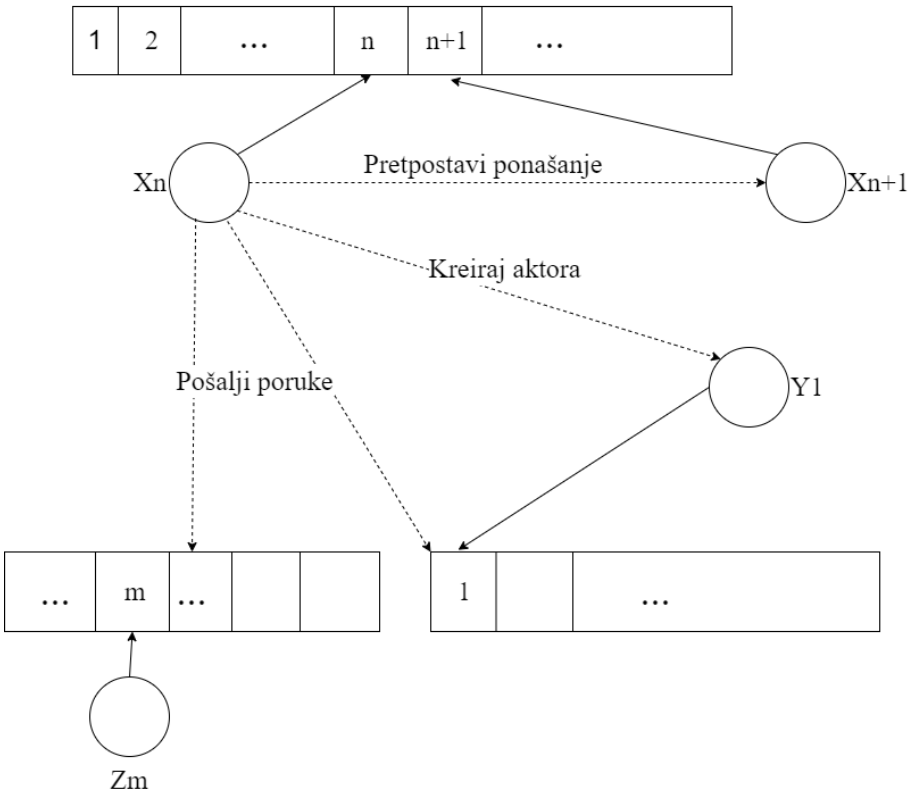
  - *Barem jednom (eng. At least once).* Ovaj nivo garancije osigurava da će poruka biti isporučena barem jednom, ali ne garantuje da će biti obrađena samo jednom. Prilikom slanja poruke, može se desiti da se ona izgubi ili da dođe do greške u obradi. U ovom slučaju, pošiljalac može ponoviti slanje poruke kako bi bila sigurna njena dostava.
  - *Najviše jednom (eng. At most once).* Ovaj nivo garancije obezbeđuje da poruka neće biti duplirana, ali ne garantuje njenu isporuku u slučaju greške. Poruka se šalje jednom i neće biti ponovljena, čak i ako dođe do greške ili gubitka poruke.
  - *Tačno jednom (eng. Exactly once.)* Ovaj nivo garancije obezbeđuje da poruka bude tačno jednom isporučena i obrađena. To znači da neće biti duplikata poruka niti izostanaka njene isporuke. Ovo je najjača garancija dostave poruka, ali je i najizazovnija za postizanje, posebno u sistemima sa mogućnošću grešaka ili prekida rada.

2.1 Osnovni koncepti

Aktor je jedinica izračunavanja koja kao odgovor na poruku može konkurentno da izvršava tri osnovne akcije:

- Slanje konačnog broja poruka drugim aktorima.
- Kreiranje konačnog broja novih aktora.
- Pretpostavljanje novog ponašanja po prijemu poruka.

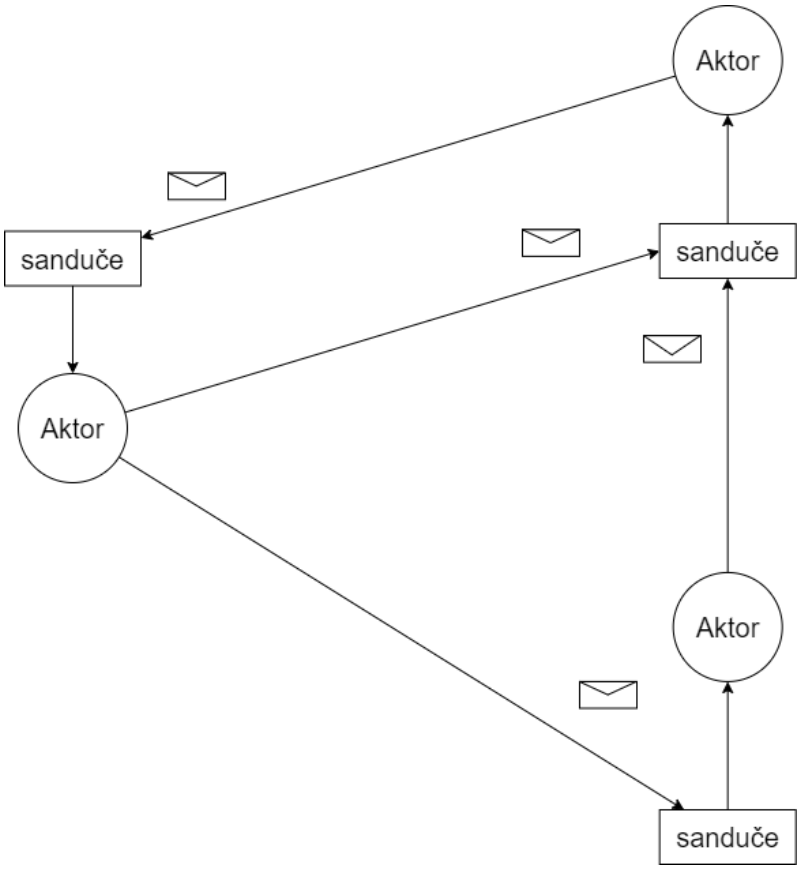
Na slici 2.1 dati su aktori X_n, X_{n+1}, Y_1 i Z_m. Aktor X_n može da pošalje poruku aktoru Z_m, kreira aktor Z_1 i da pretpostavi ponašanje aktora X_{n+1}. Navedene akcije mogu biti izvršene konkurentno i nije bitan redosled njihovog izvršavanja. Takođe, dve poruke koje su konkurentno poslate mogu biti primljene u bilo kom redosledu.



Slika 2.1: Ilustracija akcija izračunavanja u aktor sistemu.

Poruka je osnovni način komunikacije između aktora u aktor modelu. Aktor može da primi poruku od drugog aktora i da na nju odgovori. Poruka je obično asinhrona

i aktor ne mora da čeka na odgovor pre nego što nastavi sa radom. Svaki aktor ima svoje sanduče (*eng. mailbox*) koje predstavlja mesto gde se čuvaju primljene poruke koje su neobrađene. Može se predstaviti kao red čekanja poruka koje aktor prima. Ovaj koncept omogućava asinhronu komunikaciju između aktora, što znači da aktori ne moraju da čekaju jedan na drugog da bi izvršili svoje zadatke (Slika 2.2). Svaki aktor ima jedinstvenu adresu koja se koristi za slanje poruka i identifikaciju aktora u sistemu.



Slika 2.2: Komunikacija između aktora u sistemu.

Komunikacija je moguća samo između povezanih aktora, jedan aktor može da primi informaciju samo od aktora sa kojim je direktno povezan. Poruke u ovom modelu su potpuno razdvojene od pošiljaoca, kada se jednom poruka pošalje, ona postaje odgovornost primaoca. Poruke mogu da sadrže i adrese aktora što omogućava primaocu da odgovori na primljenu poruku. Za razliku od objektno orijentisanog modela, gde su objekti bazirani na konceptu klase, aktori su bazirani na konceptu ponašanja.

Ponašanje aktora u aktor modelu formalnije se može objasniti navođenjem principa indukcije aktora [11]:

- Pretpostavimo da za aktor X pri kreiranju važi svojstvo P.
- Dalje pretpostavimo da ako za X vaši svojstvo P kada obrađuje poruku, onda ima svojstvo P i pri obradi naredne poruke.
- Tada X uvek ima svojstvo P.

Aktor model može biti korišćen kao radni okvir za modelovanje i razumevanje različitih konkurentnih sistema. Jedan primer bi bila elektronska pošta gde su nalozi modelovani kao aktori a adrese naloga kao njihove adrese .

2.2 Prednosti korišćenja aktor modela

U nastavku su navedene prednosti korišćenja aktor modela u odnosu na tradicionalni objektno-orijentisani model.

- **Konkurentnost i trka za resursima.** Aktori ne dele stanje, što znači da su svi podaci koje aktor poseduje privatni i ne mogu se direktno menjati od strane drugih aktora. Ovo omogućava izolaciju stanja i smanjuje potencijalne probleme vezane za sinhronizaciju.

  Asinhrona komunikacija omogućava aktorima da nastave sa izvršavanjem i rade nezavisno jedan od drugog. Aktor koji šalje poruku ne čeka na odgovor pre nego što nastavi sa izvršavanjem što omogućava povećanje paralelizma u sistemu i povećava efikasnost korišćenja raspoloživih resursa.

- **Skalabilnost.** Aktor model omogućava vertikakno i horizontalno sklaliranje.

  U aktor modelu, aktori se izvršavaju u zasebnim procesima ili nitima, što omogućava paralelno izvršavanje. Asinhrona komunikacija između aktora omogućava da se aktori izvršavaju nezavisno jedan od drugog, što povećava efikasnost korišćenja resursa.

  Aktori se mogu izvršavati na različitim računarima, što omogućava bolje iskorišćavanje resursa i smanjuje potencijalne probleme vezane za centralizovani sistem.

- **Robusnost i otpornost na greške** Aktori su odvojeni entiteti koji ne dele stanje, što znači da greške koje se jave u jednom aktoru neće uticati na druge aktore. Ovo povećava robusnost sistema i smanjuje rizik da se greška širi na ceo sistem. Ako se pojavi greška u jednom aktoru, taj aktor se može resetovati ili zameniti novim aktorom, a ostali aktori u sistemu mogu nastaviti sa radom.

2.3 Akka.NET

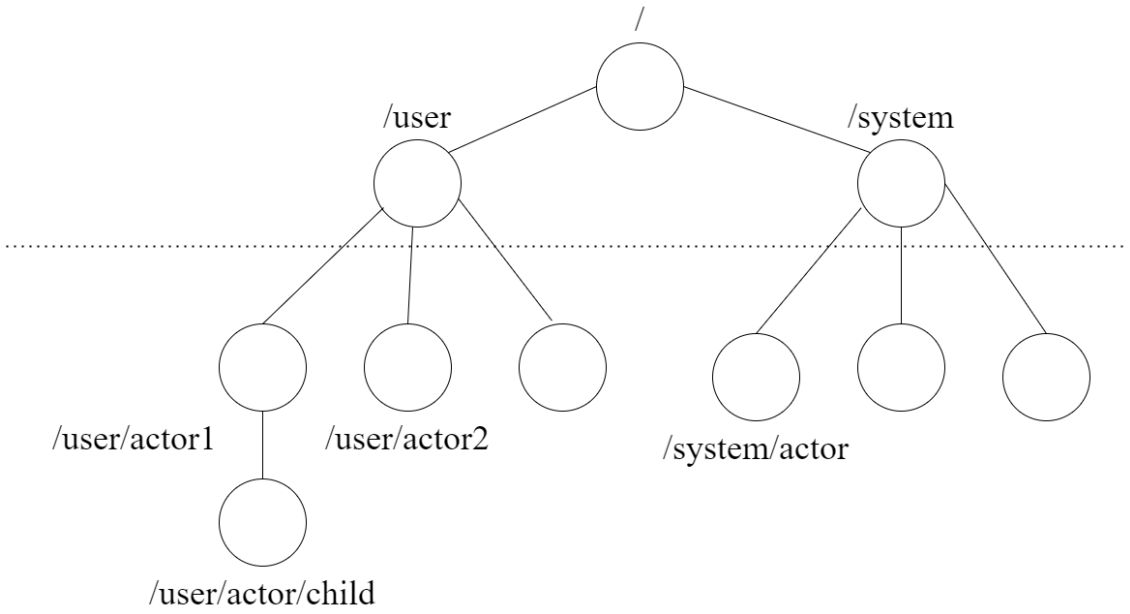
Akka.NET je radni okvir otvorenog koda za razvoj visoko konkukrentnih, skalabilnih i sistema otpornih na greške. Zasnovan je na konceptu aktor modela. Ovaj radni okvir predstavlja .NET implementaciju popularnog Akka [1] radnog okvira, koji je originalno napisan u programskom jeziku Scala.

Glavna komponeneta Akka.NET-a je aktorski sistem (*eng. actor system*) koji upravlja kreiranjem, izvršavanjem i uništavanjem aktora. Sastoji se od nadzornih aktora (*eng. guardian*) koji nadziru i upravljaju ostalim aktorima u sistemu. Svaki aktor predstavlja zaseban entitet koji ima svoje stanje i ponašanje. Komunikacija između aktora odvija se asinhronim slanjem poruka [5].

Akka.NET pruža hijerarhiju nadzora (*eng. supervision hierarchy*) koja je data u strukturi stabla, gde se u svakom čvoru nalazi jedan aktor. Pri inicijalizaciji aktor sistema, inicijalizuju se i tri nadzorna aktora: glavni nadzorni aktor (/) i njegova dva direktna potomka, korisnički (/user) i sistemski (/system) aktor. Svaki aktor koji kreira korisnik je potomak korisničkog aktora.

Hijerarhija nadzora omogućava upravljanje životnim ciklusom aktora kao i upravljanje greškama na centralizovan i fleksibilan način. Kada se neki aktor zaustavi, svi njegovi potomci se rekurzivno takođe zaustavljaju. U slučaju da u nekom aktoru dođe do greške, taj aktor se privremeno zaustavlja i informacija o grešci se šalje njegovom roditelju koji dalje odlučuje kako da obradi dobijenu grešku. Opisana hijerarhija prikazana je na slici 2.3.

Akka.NET pruža podršku za kreiranje skalabilnih distribuiranih sistema pomoću modula Akka.Cluster koji omogućava povezivanje više aktora u skup (klaster). Ovaj modul omogućava efikasno upravljanje komunikacijom između aktora u klasteru i podržava dinamičko dodavanje i uklanjanje aktora iz klastera. Takođe, pruža podršku za perzistenciju pomoću modula Akka.Persistence što omogućava čuvanje stanja aktora čak i nakon restartovanja sistema, što je ključno za pouzdanost i održivost sistema [14].



Slika 2.3: Hijerarhija aktora u sistemu Akka.NET.

Akka.NET omogućava razvoj aplikacija u različitim programskim jezicima, uključujući C#, F# i VB.NET. Može se koristiti za razvoj različitih vrsta aplikacija, uključujući IoT aplikacije, finansijske aplikacije, sistemski softver i druge [8].

Glava 3

Mikroservisna arhitektura

Mikroservisna arhitektura je moderna arhitektura za razvoj softvera koja podrazumeva organizaciju sistema u skup manjih samostalnih servisa, tzv. mikroservisa. Mikroservisi su nezavisne komponente koje pružaju usluge drugim servisima i sa njima komuniciraju. Ova arhitektura ima za cilj da omogući lakši razvoj, testiranje, održavanje i skaliranje sistema, kao i bolju otpornost na greške i promene u zahtevima [13] .

Dakle, mikroservisi predstavljaju nezavisne jedinice koje rade zajedno kako bi obavljali određene funkcije sistema. Svaki mikroservis se može razvijati, implementirati i održavati nezavisno od ostalih mikroservisa, što značajno olakšava razvoj i testiranje softvera. Mikroservisi su takođe fokusirani na zadovoljavanje jednog ili nekoliko poslovnih zahteva, što znači da su manji i jednostavniji za razvoj i održavanje.

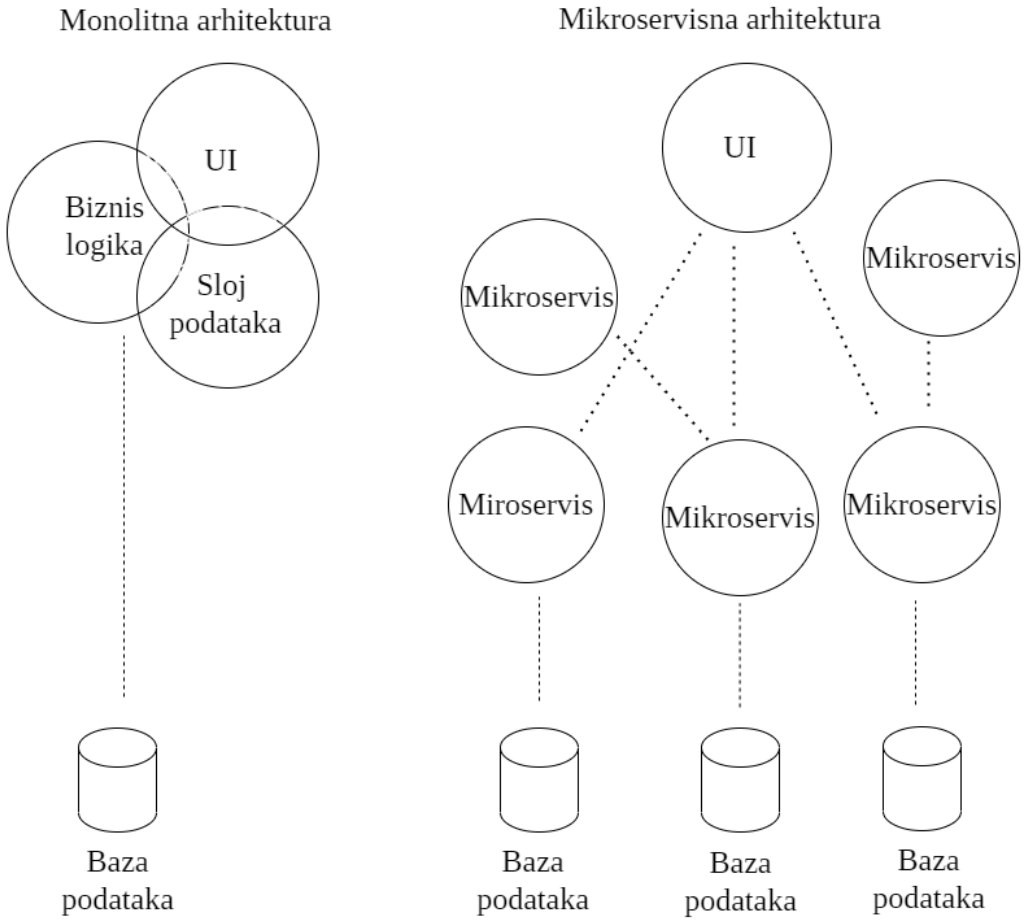
3.1 Razvoj mikroservisne arhitekture

U prošlosti, softverski sistemi su najčešće razvijani kao monolitne aplikacije tj. aplikacije koje rade u jednom procesu. Ovakvi sistemi su često bili teški za održavanje i skaliranje, jer je i najmanja promena uticala na sistem u celini. Ova teška zavisnost često je dovodila do toga da se promene usporavaju ili uopšte ne primenjuju. Greške pri izvršavanju na nekom delu sistema mogu uticati na dostupnost cele aplikacije.

Mikroservisna arhitektura je nastala sa ciljem da poveća fleksibilnost i proširivost aplikacije podelom na male labavo vezane servise koji se mogu nezavisno razvijati, testirati i postavljati na server. Svaki mikroservis je dizajniran tako da implementira jednu funkcionalnost koja ima jasno definisane granice. Takođe, svaki mikroservis

se pokreće kao poseban proces i ima svoje pomoćne resurse koje ne treba da deli sa ostalim mikroservisima.

Ideja je zasnovana na servisno orijentisanoj arhitekturi (SOA). U odnosu na SOA, mikroservisi su manje granularnosti i svaki od njih može imati zaseban sistem za skladištenje podataka. Implementacija komunikacije se uglavnom zasniva na univerzalno poznatim stilovima kao što su REST *(eng. Representational state transfer)*, gRPC *(eng. Remote procedure call)* i sličnim protokolima i oni se mogu razlikovati među servisima. Na slici 3.1 dat je jedan primer monolitne i mikroservisne arhitekture.



Slika 3.1: Primer monolitne i mikroservisne arhitekture.

Prednosti i nedostaci mikroservisne arhitekture

Prednosti mikroservisne arhitekture u odnosu na monolitnu su brojne. U nastavku su navedene neke od njih.

- Svaki mikroservis može biti potpuno odvojen od ostalih, čime je omogućena jasna podela posla u okviru timova što olakšava razvoj, testiranje i isporuku. Ova nezavistnost pruža fleksibilnost i omogućava upotrebu različitih tehnologija u svakom mikroservisu.
- Kada je potrebno skalirati aplikaciju, moguće je dodati nove instance samo onim mikroservisima koji zahtevaju veći kapacitet, čime je omogućeno skaliranje na nivou mikroservisa.
- Mikroservisne aplikacije imaju veću otpornost na greške u odnosu na monolitne aplikacije. Na primer, ako neki mikroservis ima gust saobraćaj, moguće je pokrenuti dodatnu instancu tog mikroservisa i time smanjiti opterećenje na svakoj instanci. Takođe, u slučaju pada jednog mikroservisa, ostali mikroservisi i dalje mogu nastaviti sa radom, čime se održava dostupnost i funkcionalnost aplikacije.

Mikroservisna arhitektura takođe ima i određene nedostatke koje je potrebno uzeti u obzir prilikom njene primene. U nastavku su navedeni neki od njih.

- Sa povećanjem broja servisa, operativni poslovi kao što su održavanje, nadgledanje i upravljanje servisima postaju sve zahtevniji. Takođe, koordinacija između timova koji rade na različitim mikroservisima može biti izazovna.
- Iako mikroservisi u teoriji omogućavaju nezavisno skaliranje svake komponente, u praksi se često javljaju problemi. Na primer, može se desiti da određena komponenta dostigne svoju maksimalnu granicu skaliranja pre nego što se postigne očekivano opterećenje. To znači da ta komponenta ne može efikasno podneti veći broj zahteva ili pružiti potrebnu podršku. Ova situacija može biti rezultat tehničkih ograničenja, kao što su nedovoljni hardverski resursi ili ograničenja specifična za određenu tehnologiju ili platformu na kojoj se servis izvršava.
- Povećanje broja mikroservisa i njihovo skaliranje mogu dovesti do značajnog povećanja troškova, posebno ako se podaci moraju prenositi kroz veliki broj komponenti.

- Kada se broj mikroservisa značajno poveća, upotreba deljenih biblioteka može postati problematična. Promene u deljenom kodu mogu zahtevati značajan razvojni napor, uglavnom zbog testiranja. Na primer, ukoliko dođe do promena u deljenom kodu, svaki mikroservis koji koristi tu biblioteku mora se ponovo testirati, što može biti zahtevan i intenzivan proces.

- Mikroservisi mogu postati štetni kada se primenjuju van svog originalnog konteksta. Na primer, u slučaju premalih timova ili projekata, uvođenje mikroservisa može dodati nepotreban sloj složenosti.

Važno je naglasiti da, iako mikroservisi nude mnoge prednosti, njihova primena nije pogodna za svaki projekat. Odluka o korišćenju mikroservisne arhitekture treba da se donese na osnovu detaljne analize konkretnog slučaja, uzimajući u obzir sve potencijalne prednosti i mane.

Glava 4

Reaktivni sistemi

U ranijim godinama razvoj aplikacija je uglavnom bio orijentisan na sinhronu obradu, gde aplikacija čeka na izvršavanje zahteva pre nego što nastavi sa daljim radom. Međutim, razvoj veb tehnologija i porast broja korisnika koji koriste internet, doveo je do potrebe za skalabilnijim i fleksibilnijim sistemima koji su otporni na greške i mogu se brzo prilagođavati prmenama u zahtevima korisnika ili okruženju. Sistemi koji su dizajnirani na ovaj način, nazivaju se reaktivni sistemi [10].

4.1 Reaktivni manifest

Za reaktivne sisteme važi da su (Slika 4.1):

- **Visoko odzivni (*eng. Responsive*).** Sistem ima sposobnost da brzo reaguje na događaje i da obezbedi interaktivnost sa korisnikom u realnom vremenu. U reaktivnom sistemu, događaji se ne obrađuju asinhrono, što omogućava paralelno i nezavisno izvršavanje različitih zadatka u sistemu. Asinhrona obrada oslobađa sistem od blokiranja i čekanja na završetak pojedinih operacija, omogućavajući mu bolju reaktivnost.

  Visoka odzivnost sistema je posebno važna u aplikacijama koje zahtevaju interakciju sa korisnikom, kao što su veb aplikacije ili aplikacije za mobilne uređaje. U ovim aplikacijama, korisnik očekuje brze odgovore na svoje zahteve, kao što su klik na dugme ili unos teksta. Ako sistem nije dovoljno odzivan, korisnik može dobiti utisak da je aplikacija spora, što može dovesti do frustracije i lošeg iskustva korišćenja aplikacije.

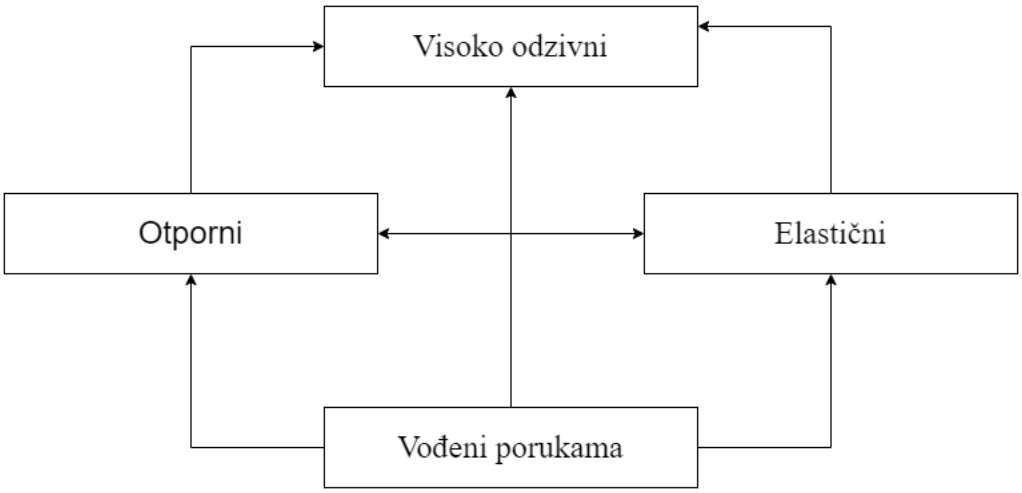
- **Otporni na greške (*eng. Resilient*).**

Sistem ima sposobnost da se oporavi od grešaka i otkaza bez gubitka podataka ili performansi. Svaka komponenta u reaktivnom sistemu je izolovana, što znači da se greška u jednoj komponenti neće proširiti na druge komponente. To znači da delovi sistema mogu otkazati i oporaviti se bez ugrožavanja sistema u celini. U reaktivnom sistemu, greške i otkazi se očekuju kao normalni deo rada, a sistemi su dizajnirani tako da budu otporni na njih.

- **Elastični (*eng. Elastic*).** Sistem je sposoban da se prilagođava promenama u zahtevima i opterećenju skaliranjem svojih resursa. Elastičnost je posebno važna u aplikacijama koje imaju promenljive zahteve i opterećenja, kao što su veb aplikacije sa velikim brojem korisnika, ili IoT aplikacije sa velikim brojem uređaja. U ovim aplikacijama, opterećenje se može dramatično povećati ili smanjiti u kratkom vremenskom periodu, što može dovesti do pada performansi ili preopterećenja sistema, ako sistem nije dovoljno elastičan.

- **Vođeni porukama (*eng. Message driven*).** Komunikacija između komponenti se vrši asinhronom razmenom poruka, umesto korišćenja direktnog poziva metoda ili funkcija. Na taj način, obezbeđuje se međusobna nezavisnost komponenti sistema, uspostavlja se jasna granica između komponenti i „labavo povezivanje" (eng. loose coupling).

Navedene osobine reaktivnih sistema čine skup principa koji se naziva reaktivni manifest (*eng. Reactive manifesto*) [6].



Slika 4.1: Reaktivni sistemi.

4.2 Reaktivno programiranje i reaktivni sistemi

Reaktivno programiranje je programska paradigma koja se fokusira na obradu tokova podataka (*eng. data streams*) i propagaciju promena kroz sistem. Ovaj pristup je izuzetno koristan u okruženjima koja zahtevaju visoku interaktivnost i odzivnost, kao što su moderne veb i mobilne aplikacije.

U reaktivnom programiranju podaci su predstavljeni kao kontinuiranu tokovi koji se mogu transformisati i reagovati na promene. Svaka promena stanja u sistemu se automatski propagira kroz sve delove sistema koji su od nje zavisni. Ovo omogućava aplikacijama da na elegantan način reaguju na promene i ažuriraju se u realnom vremenu. Takođe, ovaj pristup omogućava efikasnu asinhronu obradu, što znači da operacije koje mogu potrajati ne blokiraju izvršavanje ostatka programa. Kombinacija asinhrone obrade i automatske propagacije promena dovodi do povećane odzivnosti i interaktivnosti aplikacija.

Reaktivno programiranje i reaktivni sistemi su dva različita, ali usko povezana koncepta. Oba koncepta se bave izgradnjom sistema koji mogu efikasno i pouzdano reagovati na promene. Kako je već opisano u poglavlju 4.1, reaktivni sistemi su sistemi koji su dizajnirani da budu otporni, skalabilni i fleksibilni, a jedan od načina na koji se mogu implementirati je upravo reaktivno programiranje.

4.3 Implementacija reaktivnog sistema

Jedan od načina da se dobro implementira jedan reaktivni sistem jeste kombinacija aktor modela i mikroservisne arhitekture. Kao što je navedeno u poglavlju 2, aktor model pruža izuzetnu skalabilnost, otpornost na greške i efikasnu obradu podataka. Aktori obrađuju poruke asinhrono, što znači da se sistem može brzo i efikasno prilagoditi dinamičnim zahtevima korisnika.

S druge strane, kako je opisano u poglavlju 3, mikroservisna arhitektura pruža skalabilnost, fleksibilnost i olakšava održavanje sistema. Svaki mikroservis je odgovoran samo za jednu funkcionalnost, što omogućava brzo i efikasno skaliranje i razvoj sistema.

Implementacija može biti organizovana kao skup odvojenih mikroservisa koji komuniciraju preko aktora putem poruka. Svaki mikroservis je odgovoran za jednu specifičnu funkcionalnost i sadrži aktore koji obrađuju poruke koje se šalju iz drugih mikroservisa. Aktori u mikroservisima treba da budu razdvojeni tako da se bave

samo jednom funkcionalnošću.

Ova implementacija reaktivnog sistema omogućava brzu i efikasnu obradu podataka, a dinamičko prilagođavanje zahtevima korisnika omogućava veću fleksibilnost i bolje performanse u realnom vremenu.

Glava 5

Opis razvijenog sistema

U nastavku će biti prikazan dizajn i implementacija jednog reaktivnog sistema na primeru aplikacije za internet prodavnicu. Implementacija je zasnovana na mikroservisnoj arhitekturi i aktor modelu.

Osnovne funkcionalnosti aplikacije su:

- Pregled dostupnih proizvoda.
- Pretraga i ažuriranje proizvoda iz baze.
- Dodavanje i uklanjanje proizvoda iz korpe.
- Kreiranje i otkazivanje narudžbine.

Aplikacija je napisana u .NET okruženju, a korišćen je i Akka.NET radni okvir koji je detaljnije opisan u poglavlju 2.3. Izvorni kod projekta je objavljen na GitHub-u i može se pronaći u [7].

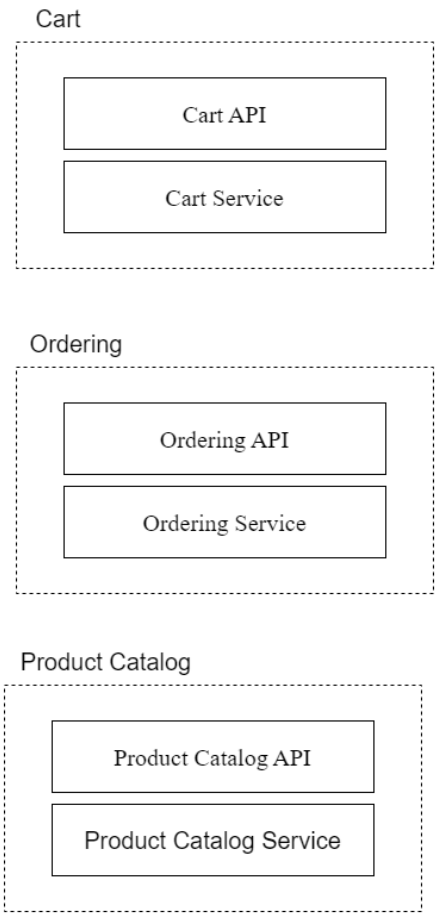
5.1 Arhitektura sistema

Aplikacija je organizovana u tri glavna mikroservisa od kojih je svaki odgovoran za specifičnu funkcionalnost:

1. *Product Catalog.* Ovaj mikroservis sadrži informacije o dostupnosti proizvoda kao i mogućnost ažuriranja proizvoda u bazi.
2. *Ordering.* Ovaj mikroservis sadrži funkcionalnosti koje su vezane za korisničke narudžbine, kao što je kreiranje i otkazivanje narudžbina.

3. *Cart.* Ovaj mikroservis sadrži funkcionalnosti koje su vezane za korisničke korpe za kupovinu. Konkretno, zadužen je za dodavanje i uklanjanje proizvoda iz korpe.

Svaki mikroservis sadrži dve komponente: servis, koji upravlja odgovarajućim procesom i API (*eng. Application Programming Interface*) koji služi kao ulazna tačka za interakciju sa tim servisom. Na slici 5.1 dat je grafički prikaz arhitekture ovog sistema.

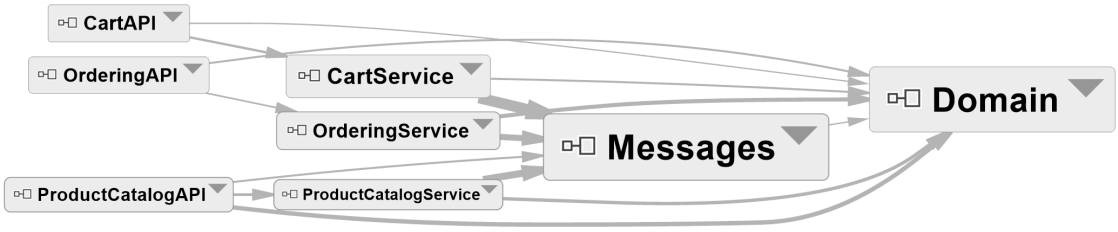


Slika 5.1: Arhitektura aplikacije za internet prodavnicu.

Svi mikroservisi komuniciraju razmenom poruka putem Akka.NET aktora, a komunikacija je omogućena korišćenjem modula Akka.Cluster. Za perzistenciju, umesto baza podataka, korišćen je Akka.Persistence radni okvir koji pruža mehanizme za trajno čuvanje stanja aktora. Svakim novim pokretanjem aplikacije, stanje aktora

se obnavlja iz trajnog skladišta i svi podaci su ponovo dostupni, čime se osigurava da se podaci ne gube u slučaju prekida rada.

Na slici 5.2 prikazana je struktura kao i međusobne zavisnosti između projekata.



Slika 5.2: Struktura projekta

Domenski entiteti i poruke

Domenski enttiteti smešteni su u projektu pod nazivom *Domain* koji je deljen među svim mikroservisima. Implementirani su sledeći entiteti:

- *Product.* Klasa koja predstavlja određeni proizvod. Sadrži informacije o proizvodu kao što su jedinstveni identifikator, naziv, cena, količina na stanju i rezervisana količina.
- *Order.* Klasa koja predstavlja jednu narudžbinu. Sadrži listu stavki narudžbine (*OrderItem*), cenu i status koji govori da li je narudžbina uspešno kreirana.
- *Cart.* Klasa koja predstavlja korpu za kupovinu i sadrži listu izabranih stavki (*CartItem*). Klasa (*CartItem*) pored osnovnih informacija o proizvodu, sadrži i referencu na identifikator proizvoda i nije u direktnoj zavisnosti sa klasom Product. Ovo omogućava izolaciju i fleksibilnost u upravljanju proizvodima, jer se informacije o proizvodima mogu održavati odvojeno od korpe, stavki i narudžbina.

Objekti koji predstavljaju poruke koje aktori međusobno razmenjuju smešteni su u projektu pod nazivom *Messages*. Grupisani su u dva prostora imena, *Commands* (skup komandi koje aktori šalju jedni drugima) i *Events* (skup događaja koji predstavljaju odgovor na te komande). Komande predstavljaju zahteve za izvršavanje određene akcije od strane drugog aktora, kao što su dodavanje proizvoda u korpu

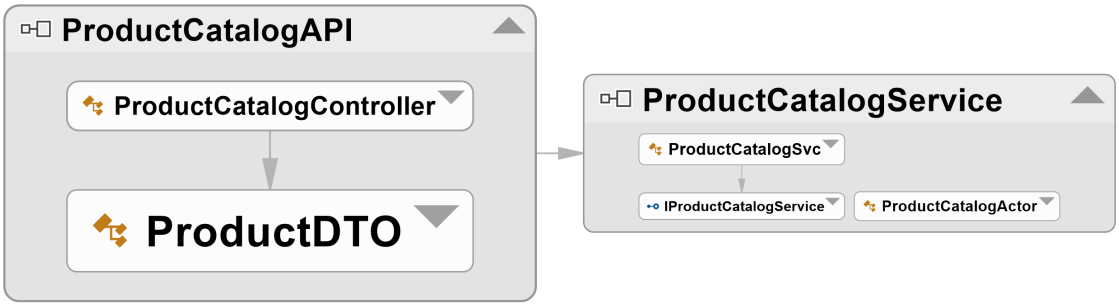
(*AddToCart*) i kreiranje narudžbine (*CreateOrder*). Događaji predstavljaju informacije o nečemu što se desilo u sistemu ili promeni stanja. Oni obaveštavaju druge aktore o nekom relevantnom događaju, ali ne zahtevaju direktnu akciju. Neki od primera događaja su informacija da je korpa za kupovinu uspešno ažurirana (*CartUpdateSuccess*) ili informacija da je narudžbina uspešno kreirana (*OrderSuccess*).

Kao i u slučaju projekta *Domain*, i ovaj projekat je deljen među mikroservisima. Odluka da poruke budu smeštene u deljeni projekat je posledica ograničenja koje nameće radni okvir Akka.NET u kontekstu serijalizacije i deserijalizacije. Da bi ovaj proces uspešno funkcionisao, Akka.NET treba da bude u stanju da jasno identifikuje tip poruke, što uključuje informacije o prostoru imena i imenu klase. Problem nastaje kada dva mikroservisa definišu identične poruke, ali u različitim prostorima imena. Akka.NET će te poruke tretirati kao potpuno različite, iako imaju istu strukturu. Kao rezultat, dva aktora iz različitih servisa neće moći da komuniciraju.

5.2 Opis mikroservisa

Mikroservis *Product Catalog*

Mikroservis *Product Catalog* je zadužen za upravljanje dostupnim proizvodima. Arhitekturalno je podeljen na servis koji je zadužen za obavljanje odgovarajućih operacija nad proizvodima (*ProductCatalogService*) i REST API (*ProductCatalogAPI*) (slika 5.3).



Slika 5.3: Mikroservis *Product Catalog*.

Glavna komponenta servisa *ProductCatalogService* je aktor *ProductCatalogActor* koji je zadužen za upravljanje proizvodima u bazi. Nasleđuje klasu *ReceivePersistentActor* (modul *Akka.Persistence*) čime je omogućeno trajno čuvanje stanja proizvoda. Kada se aplikacija pokrene, stanje proizvoda se učitava u listu *Products* u okviru

aktora. U slučaju da postoji prethodno sačuvano stanje, ono se vraća korišćenjem metode *Recover*, a prilikom ažuriranja, koristi se metoda *Persist* kako bi se trajno sačuvale promene o stanju proizvoda. U okviru ovog aktora definisane su sledeće komande:

- *ReserveProduct.* Po prijemu ove komande, aktor *ProductCatalogActor* vrši pokušaj rezervacije određenog proizvoda. Ako je rezervacija uspešna, aktor vraća poruku *ReserveProductSuccess* sa novom verzijom proizvoda. U suprotnom, šalje se poruka *ReserveProductFailed*, signalizirajući da rezervacija nije moguća.
- *ReleaseProductReservation.* Po prijemu ove komande, *ProductCatalogActor* vrši pokušaj oslobađanja rezervacije proizvoda. Po uspešnom oslobađanju, vraća se poruka *ReleaseProductReservationSuccess.*
- *GetAllProducts.* Po prijemu ove komande, aktor vraća listu svih proizvoda u katalogu.
- *UpdateInventory.* Po prijemu ove komande, *ProductCatalogActor* vrši pokušaj ažuriranja količine određenog proizvoda. Ako je ažuriranje uspešno, aktor vraća poruku *InventoryStatus* koja sadrži informacije o ažuriranom proizvodu.

U situacijama kada više aktora pokušava da ažurira isti proizvod istovremeno, postoji mogućnost pojave konflikta koji mogu da dovedu do nekonzistentnosti podataka. Kako bi se ovo izbeglo, u okviru entiteta *Product* definisiano je polje *Version* koje označava njegovu trenutnu verziju. Svaki aktor koji vrši ažuriranje proizvoda prvo proverava trenutnu verziju proizvoda pre nego što izvrši promene. Ako je trenutna verzija proizvoda koju aktor čita jednaka verziji koju je poslednje pročitao, to znači da nema konflikata i aktor može sigurno izvršiti promene. Time se obezbeđuje da samo jedan aktor može uspešno izvršiti ažuriranje proizvoda u datom trenutku.

U nastavku je data implementacija ovog aktora u programskom jeziku C#.

```
1 using Akka.Actor;
2 using Akka.Persistence;
3 using Domain.Entities;
4 using Messages.Commands;
5 using Messages.Events;
6
7 namespace ProductCatalogService.Actors
```

```
8  {
9      public class ProductCatalogActor : ReceivePersistentActor
10     {
11         private Dictionary<int, Product> Products = new Dictionary<
       int, Product>();
12         public ProductCatalogActor()
13         {
14             Recover<Product>(product =>
15             {
16                 Products[product.Id] = product;
17             });
18             Command<ReserveProduct>(reserveProduct =>
19             {
20                 if (Products.TryGetValue(reserveProduct.ProductId,
       out var product))
21                 {
22                     if (product.Quantity >= reserveProduct.Quantity
        && product.Version == reserveProduct.Version)
23                     {
24                         product.ChangeQuantity(-reserveProduct.
       Quantity);
25                         product.IncreaseReservedQuantity(
       reserveProduct.Quantity);
26                         product.Version++;
27                         Sender.Tell(new ReserveProductSuccess(
       reserveProduct.ProductId, reserveProduct.Quantity, product.
       Version));
28                         Persist(product, _ =>
29                         {
30                             Products[reserveProduct.ProductId] =
       product;
31                         });
32                     }
33                     else
34                     {
35                         Sender.Tell(new ReserveProductFailed(
       reserveProduct.ProductId, reserveProduct.Quantity));
36                     }
37                 }
38                 else
39                 {
40                     Sender.Tell(new ProductNotFound(reserveProduct.
```

```
ProductId));
41                  }
42              });
43              Command<ReleaseProductReservation>(
releaseProductReservation =>
44              {
45                  if (Products.TryGetValue(releaseProductReservation.
ProductId, out var product))
46                  {
47                      product.ChangeQuantity(
releaseProductReservation.Quantity);
48                      product.DecreaseReservedQuantity(
releaseProductReservation.Quantity);
49                      product.Version++;
50                      Sender.Tell(new
ReleaseProductReservationSuccess(releaseProductReservation.
ProductId, releaseProductReservation.Quantity));
51                      Persist(product, _ =>
52                      {
53                          Products[releaseProductReservation.
ProductId] = product;
54                      });
55                  }
56                  else
57                  {
58                      Sender.Tell(new ProductNotFound(
releaseProductReservation.ProductId));
59                  }
60              });
61              Command<GetAllProducts>(getAllProducts =>
62              {
63                  Sender.Tell(new GetAllProducts(Products.Values.
ToList()));
64              });
65
66              Command<LookupProduct>(lookupProduct =>
67              {
68                  if (Products.TryGetValue(lookupProduct.ProductId,
out var product))
69                  {
70                      if (product.CheckAvailability())
71                      {
```

```
72                          Sender.Tell(new InventoryStatus(
    lookupProduct.ProductId, product.Quantity, product.Version));
73                      }
74                      else
75                      {
76                          Sender.Tell(new ProductNotFound(
    lookupProduct.ProductId));
77                      }
78                  }
79                  else
80                  {
81                      Sender.Tell(new ProductNotFound(lookupProduct.
    ProductId));
82                  }
83              });
84
85              Command<UpdateInventory>(updateInv =>
86              {
87                  if (Products.TryGetValue(updateInv.ProductId, out
    var product))
88                  {
89                      product.ChangeQuantity(updateInv.Quantity);
90                      product.Version++;
91                      Sender.Tell(new InventoryStatus(updateInv.
    ProductId, product.Quantity, product.Version));
92                      Persist(product, _ =>
93                      {
94                          Products[updateInv.ProductId] = product;
95                      });
96                  }
97                  else
98                  {
99                      Sender.Tell(new ProductNotFound(updateInv.
    ProductId));
100                 }
101             });
102             Command<AddProduct>(addProduct =>
103             {
104                 if (Products.TryGetValue(addProduct.Product.Id, out
     var product))
105                 {
106                     product.ChangeQuantity(addProduct.Product.
```

```
    Quantity);
107                     product.Version++;
108                     Sender.Tell(new InventoryStatus(addProduct.
    Product.Id, product.Quantity, product.Version));
109                     Persist(product, _ =>
110                     {
111                         Products[addProduct.Product.Id] = product;
112                     });
113                 }
114                 else
115                 {
116                     Products.Add(addProduct.Product.Id, addProduct.
    Product);
117                     Sender.Tell(new InventoryStatus(addProduct.
    Product.Id, addProduct.Product.Quantity, 1));
118                     Persist(addProduct, _ =>
119                     {
120                         Products[addProduct.Product.Id] =
    addProduct.Product;
121                     });
122                 }
123             });
124         }
125         public override string PersistenceId => nameof(
    ProductCatalogActor);
126     }
127 }
```

ProductCatalogAPI je REST API implementiran u ASP.NET koji predstavlja ulaznu tačku za interakciju sa servisom *ProductCatalogService*. U nastavku je data implementacija kontrolera *ProductCatalogController*.

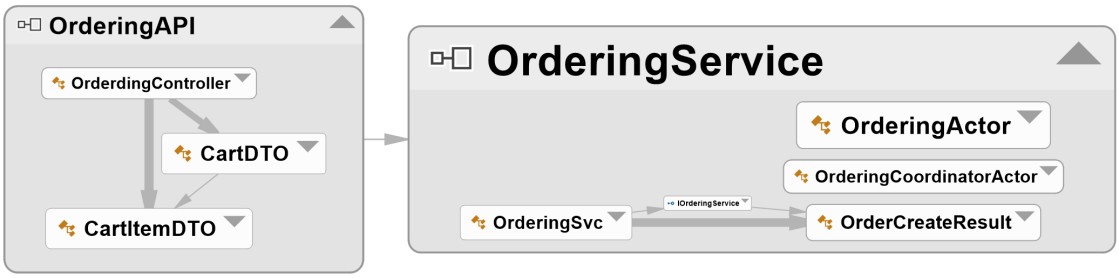
```
1  using Domain.Entities;
2  using Microsoft.AspNetCore.Mvc;
3  using ProductCatalogService;
4
5  namespace ProductCatalogAPI.Controllers
6  {
7      [Route("api/[controller]")]
8      [ApiController]
9      public class ProductCatalogController : ControllerBase
10     {
11         private readonly IProductCatalogService
```

```
ProductCatalogService;

        public ProductCatalogController(IProductCatalogService
productCatalogService)
        {
            ProductCatalogService = productCatalogService;
        }
        [HttpGet("{productId}")]
        public async Task<ActionResult> LookupProduct(int productId
)
        {
            var isFound = await ProductCatalogService.LookupProduct
(productId);
            if (isFound == false)
            {
                return NotFound();
            }
            return Ok();
        }

        [HttpGet]
        public async Task<ActionResult<List<ProductDTO>>>
GetAllProducts()
        {
            var result = await ProductCatalogService.GetAllProducts
();
            if (result == null)
                return NotFound();
            var products = new  List<ProductDTO>();
            foreach (var item in result)
            {
                var productDTO = new ProductDTO
                {
                    productId = item.Id,
                    Title = item.Title,
                    Price= item.Price,
                    Inventory    = item.Quantity
                };

                products.Add(productDTO);
            }
            return Ok(products);
```

```
48          }
49
50          [HttpPost]
51          public async Task<ActionResult<int>> AddProduct(int
    productId, string title, decimal price)
52          {
53              var product = new Product(productId, title, price);
54              var result = await ProductCatalogService.AddProduct(
    product);
55              return Ok(result.ProductId);
56          }
57      }
58  }
```

Mikroservis *Ordering*

Mikroservis *Ordering* je zadužen za upravljanje narudžbinama, odnosno njihovim kreiranjem i otkazivanjem. Arhitekturalno je podeljen na *OrderingAPI* i *OrderingService* (slika 5.4). U okviru *OrderingService*-a definisana su dva aktora:



Slika 5.4: Mikroservis *Ordering*.

- *OrderingCoordinatorActor*. Ovaj aktor predstavlja koordinatora između aktora *OrderingActor*, *ProductCatalogActor* i *CartCoordinatorActor*. U okviru ovog aktora definisane su sledeće komande:
  - *CreateOrder*. Po prijemu ove komande, aktor kreira instancu aktora *OrderingActor* za tu narudžbinu i šalje zahtev aktoru *ProductCatalogActor* za proveru dostupnosti i rezervaciju proizvoda u korpi. Tokom ovog procesa, sistem proverava dostupnost svakog proizvoda i rezerviše traženu količinu proizvoda samo ako je ona dostupna. Ako su svi proizvodi uspešno rezervisani, koordinator šalje zahtev aktoru *OrderingActor* za obradu

narudžbine. Rezultat zahteva se asinhrono prosleđuje trenutnom aktoru i pošiljaocu poruke što omogućava da se nastavi sa izvršavanjem drugih operacija bez blokiranja i čekanja na odgovor. Odgovor će biti prosleđen na odgovarajuće adrese, trenutni aktor će ga obraditi u okviru odgivarajućih komandi (*OrderSuccess*, *OrderFailed*), a pošiljaoc će dobiti odgovor. Rezervacija proizvoda je implementirana kako bi se sprečila višestruka rezervacija istog proizvoda ili prodaja veće količine proizvoda nego što je dostupno.

  - *CancelOrder.* Po prijemu ove komande, aktor šalje zahtev za otkazivanje narudžbine aktoru *OrderingActor*-u. Rezultat zahteva se asinhrono prosleđuje trenutnom aktoru i pošiljaocu poruke što omogućava da se nastavi sa izvršavanjem drugih operacija bez blokiranja i čekanja na odgovor. Odgovor će biti prosleđen na odgovarajuće adrese, trenutni aktor če ga obraditi u okviru komande *OrderCanceled*, a pošiljaoc će dobiti odgovor o uspešnosti zahteva.

- *OrderingActor.* Ovaj aktor je zadužen za obradu jedne narudžbine, odnosno za obradu poruka za kreiranje i otkazivanje narudžbine (*CreateOrder* i *CancelOrder*). Nasleđuje klasu *UntypedPersistentActor* čime je omogućeno trajno čuvanje stanja narudžbine, čak i nakon prekida rada aplikacije. U okviru ovog aktora definisane su sledeće komande:

  - *CreateOrder.* Kada aktor primi poruku za kreiranje narudžbine, on proverava da li već postoji aktivna narudžbina. Ako postoji, šalje poruku o neuspehu. U suprotnom, kreira novu instancu Order objekta koji predstavlja narudžbinu i perzistira je kao događaj *OrderCreated*. Nakon perzistencije, šalje poruku o uspehu pošiljaocu poruke.
  - *CancelOrder.* Kada aktor primi poruku za otkazivanje narudžbine, on proverava da li postoji aktivna narudžbina. Ako ne postoji, šalje poruku o neuspehu. U suprotnom, menja status narudžbine i perzistira taj događaj kao *OrderCanceled*. Nakon perzistencije, šalje poruku o uspehu pošiljaocu poruke.

  Kroz upotrebu perzistentnog skladišta i metoda *Persist*, aktor trajno čuva stanje narudžbine i njene promene. Prilikom rekonstrukcije stanja, aktor koristi metodu *OnRecover* kako bi obradio perzistirane događaje i ažurirao svoje stanje u skladu sa njima.

U nastavku je data implementacija klase *OrderingCoordinatorActor*.

```
using Akka.Actor;
using Messages.Commands;
using Messages.Events;

namespace OrderingService.Actors
{
    public class OrderingCoordinatorActor: ReceiveActor
    {
        private readonly IActorRef ProductCatalogActor;
        private readonly Dictionary<string, IActorRef>
    OrderingActors = new Dictionary<string, IActorRef>();
        private readonly IActorRef CartCoordinatorActor;

        public OrderingCoordinatorActor(IActorRef
    productCatalogActor, IActorRef cartCoordinatorActor)
        {
            ProductCatalogActor = productCatalogActor;
            CartCoordinatorActor = cartCoordinatorActor;

            ReceiveAsync<CreateOrder>(async cmd =>
            {
                var reservedCount = 0;
                List<ReserveProductResult> reserveResults = new
    List<ReserveProductResult>();
                string orderingActorGuid = Guid.NewGuid().ToString
    ();
                IActorRef orderingActor = Context.ActorOf(Props.
    Create(() => new OrderingActor(orderingActorGuid)), nameof(
    OrderingActor) + "-"  + orderingActorGuid);
                OrderingActors.Add(orderingActorGuid, orderingActor
    );
                foreach (var item in cmd.Cart.CartItems)
                {
                    var inventoryStatus = await ProductCatalogActor
    .Ask<InventoryStatus>(new LookupProduct(item.ProductId));
                    if (inventoryStatus == null || inventoryStatus.
    AvailableQuantity == 0)
                        orderingActor.Tell(new OrderFailed(
    orderingActorGuid, "Product not found"));

                    var reserveProductResult = await
```

```
    ProductCatalogActor.Ask<ReserveProductResult>(new ReserveProduct
    (item.ProductId, item.Quantity, inventoryStatus.Version));
32                  reserveResults.Add(reserveProductResult);
33                  if (reserveProductResult is
    ReserveProductSuccess reserveProductSuccess)
34                  {
35                      reservedCount++;
36                  }
37              }
38
39              if (reservedCount == cmd.Cart.CartItems.Count)
40              {
41                  await orderingActor.Ask<OrderResult>(cmd).
    PipeTo(Self, Sender);
42              }
43              else
44              {
45                  foreach (var item in reserveResults.OfType<
    ReserveProductSuccess>())
46                  {
47                      ProductCatalogActor.Tell(new
    ReleaseProductReservation(item.ProductId, item.Quantity));
48                  }
49                  CartCoordinatorActor.Tell(new ClearCart());
50
51                  if (reserveResults.Any(x => x is
    ReserveProductFailed))
52                  {
53                      Sender.Tell(new OrderFailed("Version
    mismatch occurred while reserving the products."));
54                  }
55                  else
56                  {
57                      Sender.Tell(new OrderFailed("Not all
    products could be reserved."));
58                  }
59              }
60          });
61
62          ReceiveAsync<OrderSuccess>(async orderSuccess =>
63          {
64              Sender.Tell(new OrderSuccess(orderSuccess.OrderId))
```

```
    ;
65                  CartCoordinatorActor.Tell(new ClearCart());
66                  IActorRef orderingActor = OrderingActors[
    orderSuccess.OrderId];
67                  orderingActor.Tell(new ClearOrder());
68              });
69              ReceiveAsync<OrderFailed>(async orderFailed =>
70              {
71                  IActorRef orderingActor = OrderingActors[
    orderFailed.OrderId];
72                  orderingActor.Tell(new ClearOrder());
73                  CartCoordinatorActor.Tell(new ClearCart());
74                  Sender.Tell(new OrderFailed(orderFailed.Message));
75              });
76              ReceiveAsync<OrderCanceled>(async orderCancelled =>
77              {
78                  IActorRef orderingActor = OrderingActors[
    orderCancelled.OrderId];
79                  orderingActor.Tell(new ClearOrder());
80                  Sender.Tell(new OrderCanceled(orderCancelled.
    OrderId));
81                  CartCoordinatorActor.Tell(new ClearCart());
82              });
83              ReceiveAsync<CancelOrder>(async cmd =>
84              {
85                  IActorRef orderingActor = OrderingActors[cmd.Order.
    Id];
86
87                  var releaseTasks = cmd.Order.OrderItems.Select(item
     => ProductCatalogActor.Ask<ReleaseProductReservation>(new
    ReleaseProductReservation(item.ProductId, item.Quantity))).
    ToList();
88                  await Task.WhenAll(releaseTasks);
89                  await orderingActor.Ask<OrderResult>(cmd).PipeTo(
    Self, Sender);
90              });
91          }
92      }
93  }
```

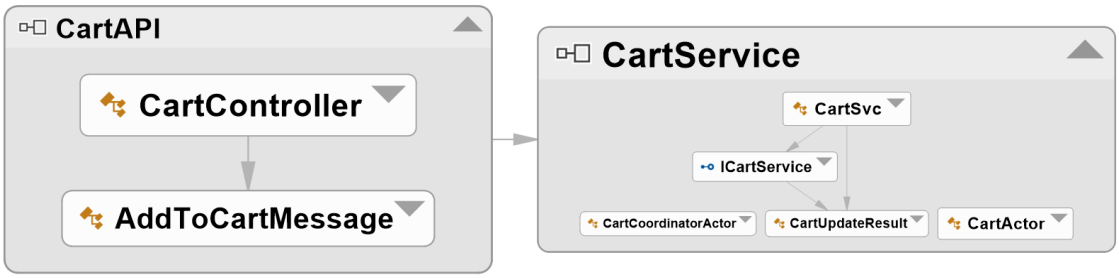
OrderingAPI je REST API koji je implementiran u ASP.NET koji predstavlja ulaznu tačku za interakciju sa servisom *OrderingService*. U nastavku je data imple-

mentacija klase *OrderingAPI.Controllers.OrderingController*.

```
using Domain.Entities;
using Microsoft.AspNetCore.Mvc;
using OrderingService;

namespace OrderingAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class OrderdingController : ControllerBase
    {
        private readonly IOrderingService OrderingService;
        public OrderdingController(IOrderingService orderingService
    )
        {
            OrderingService = orderingService;
        }
        [HttpPost]
        public async Task<ActionResult> CreateOrder([FromBody]
    CartDTO cart)
        {
            List<CartItem> cartItems = new List<CartItem>();
            foreach (var item in cart.CartItems)
            {
                CartItem cartItem = new CartItem(item.ProductId,
    item.Quantity, item.Price);
                cartItems.Add(cartItem);
            }
            Cart c = new Cart(cart.Id, cartItems);
            var result = await OrderingService.CreateOrder(c);
            return Ok(result);
        }
    }
}
```

Mikroservis *Cart*

Mikroservis *Cart* je zadužen za upravljanje korpom za kupovinu. Arhitekturalno je podeljen na *CartService* i *CartAPI* (slika 5.5). U okviru servisa *CartService* definisana su dva aktora:



Slika 5.5: Mikroservis *Cart*.

- *CartActor*. Ovaj aktor je zadužen za obradu poruka za pregled (*GetCart*), dodavanje (*AddToCart*) i uklanjanje proizvoda iz korpe (*RemoveFromCart*). Nasleđuje klasu PersistentActor čime je omogućeno trajno čuvanje stanja korpe za kupovinu čak i nakon prekida rada aplikacije.
- *CartCoordinatorActor*. Ovaj aktor predstavlja koordinatora između aktora *CartActor* i *ProductCatalogActor*. U okviru ovog aktora definisane su sledeće komande:
  - *AddToCart*. Kada aktor primi ovu komandu on šalje zahtev za proveru dostupnosti proizvoda aktoru *ProductCatalogActor*. Ukoliko je proizvod dostupan, aktor šalje poruku aktoru *CartActor* da doda proizvod u korpu. Rezultat tog zahteva se asinhrono prosleđuje trenutnom aktoru, koji će ga obraditi u okviru komande *CartUpdateSuccess* ili *CartUpdateFailed*, kao i pošiljaocu.
  - *RemoveFromCart*. Po prijemu ove komande, aktor šalje poruku aktoru *CartActor* da ukloni proizvod iz korpe. Rezultat tog zahteva se asinhrono prosleđuje trenutnom aktoru, koji će ga obraditi u okviru komande *CartUpdateSuccess* ili *CartUpdateFailed*, kao i pošiljaocu.
  - *ClearCart*. Na zahtev za brisanje korpe, aktor šalje poruku aktoru CartActor da izvrši brisanje korpe.

U nastavku je data implementacija klase *CartCoordinatorActor*.

```
1 using Akka.Actor;
2 using Messages.Commands;
3 using Messages.Events;
4
5 namespace CartService.Actors
6 {
```

```
7     public class CartCoordinatorActor : ReceiveActor
8     {
9         private readonly IActorRef CartActor;
10        private readonly IActorRef ProductCatalogActor;
11        public CartCoordinatorActor(IActorRef cartActor, IActorRef
    productCatalogActor)
12        {
13            CartActor = cartActor;
14            ProductCatalogActor = productCatalogActor;
15
16            ReceiveAsync<AddToCart>(async cmd =>
17            {
18                try
19                {
20                    var inventoryStatus = await ProductCatalogActor
    .Ask<InventoryStatus>(new LookupProduct(cmd.ProductId));
21                    if (inventoryStatus.AvailableQuantity >= cmd.
    Quantity)
22                    {
23                        await CartActor.Ask<CartUpdateResult>(cmd).
    PipeTo(Self, Sender);
24                    }
25                    else
26                    {
27                        Sender.Tell(new CartUpdateFailed($"Not
    enough inventory for {cmd.ProductId}"));
28                    }
29                }
30                catch (Exception ex)
31                {
32                    Sender.Tell(new CartUpdateFailed($"An error
    occurred while updating the cart: {ex.Message}"));
33                }
34            });
35            ReceiveAsync<RemoveFromCart>(async cmd =>
36            {
37                await CartActor.Ask<CartUpdateResult>(cmd).PipeTo(
    Self, Sender);
38            });
39            Receive<CartUpdateSuccess>(cartUpdateSuccess =>
40            {
41                Sender.Tell(new CartUpdateSuccess());
```

```
42              });
43              Receive<CartUpdateFailed>(cartUpdateFailed =>
44              {
45                  Sender.Tell(new CartUpdateFailed(cartUpdateFailed.
    ErrorMessage));
46              });
47              Receive<ClearCart>(clearCart =>
48              {
49                  CartActor.Tell(new ClearCart());
50              });
51          }
52      }
53  }
```

CartAPI je REST API koji predstavlja ulaznu tačku za interakciju sa servisom *CartService*. U nastavku je data implementacija klase *CartAPI.Controllers.CartController*.

```
1
2  using CartService;
3  using Domain.Entities;
4  using Microsoft.AspNetCore.Mvc;
5
6  namespace CartAPI.Controllers
7  {
8      [Route("api/[controller]")]
9      [ApiController]
10     public class CartController : ControllerBase
11     {
12         private readonly ICartService CartService;
13         public CartController(ICartService cartService)
14         {
15             CartService = cartService;
16         }
17         [HttpGet]
18         public async Task<ActionResult<Cart>> GetCart()
19         {
20             var cart = await CartService.GetCart();
21             if (cart == null)
22             {
23                 return NotFound();
24             }
25             return Ok(cart);
26         }
```

```
27          [HttpPost]
28          public async Task<ActionResult> AddItemToCart([FromBody]
    AddToCartMessage message)
29          {
30              await CartService.AddToCart(message.ProductId, message.
    Quantity, message.Price);
31              return Ok();
32          }
33
34          [HttpPost("{cartId}/items/{itemId}")]
35          public async Task<IActionResult> RemoveItemFromCart(int
    itemId, int quantity)
36          {
37              await CartService.RemoveFromCart(itemId, quantity);
38              return Ok();
39          }
40      }
41  }
```

5.3 Komunikacija između mikroservisa

Glavna ideja iza ove implementacije je bila da se aktor model implementira u svim aspektima sistema, pa i u komunikaciji između mikroservisa, ali tako da oni i dalje ostanu nezavisni.

Za komunikaciju između mikroservisa do sada je najčešće korišćen „publish-subscribe" mehanizam [9]. Ovaj mehanizam funkcioniše tako što svi mikroservisi mogu da se „pretplate" (*eng. subscribe*) na određene događaje, a kada se neki događaj „objavi" (*eng. publish*) oni na isti mogu da reaguju. Implementacija ovog mehanizma može se postići kroz različite tehnologije i protokole, kao što su redovi poruka(*eng. message queues*), RabbitMQ, Apache Kafka i slično.

Međutim, da bi mikroservisi komunicirali kao aktori, korišćen je mehanizam klasteringa pomoću Akka.Cluster biblioteke. Ova biblioteka pruža ugrađene mehanizme za upravljanje i komunikaciju između aktora smeštenih u različitim klasterima. Svrha korišćenja Akka.Cluster-a je ostvarivanje decentralizovane komunikacije i koordinacije između mikroservisa putem distribuiranog sistema aktora.

U ovoj implementaciji svaki od mikroservisa (*Ordering, Cart, ProductCatalog*) smešten je u svoj zaseban klaster. Mikroservis se pridružuje klasteru na osnovu odeđene konfiguracije koja uključuje podešavanje identifikatora klastera, adrese i

porta na kojem će mikroservis osluškivati dolazne konekcije, kao i određivanje rola koje mikroservis obavlja u klasteru.

U nastavku je dat primer konfiguracije klastera za mikroservis Ordering.

```
akka {
  actor {
    provider = "Akka.Cluster.ClusterActorRefProvider, Akka.Cluster"
  }
  remote {
    dot-netty.tcp {
      hostname = "127.0.0.1"
      port = 8083
    }
  }
  cluster {
    seed-nodes = [
      "akka.tcp://ShopCluster@localhost:8081"
    ]
  }
}
```

Svaki klaster može biti konfigurisan i skaliran nezavisno od drugih klastera, pružajući fleksibilnost u upravljanju i skaliranju mikroservisa prema njihovim specifičnim potrebama. Nakon što su mikroservisi pridruženi klasterima, mogu komunicirati međusobno putem poruka. Ovo se može postići slanjem poruka između aktora u različitim klasterima. Svaki mikroservis može imati referencu na aktore u drugim klasterima i koristiti tu referencu za slanje poruka.

U nastavku će, kroz primer Ordering mikroservisa, biti objašnjena njegova konfiguracija i komunikacija sa aktorima iz drugih mikroservisa.

Na početku rada, mikroservis inicijalizuje aktor sistem i svoje glavne aktore u klasteru *OrderingActor* i *OrderingCoordinatorActor*). Kako je u ovom mikroservisu neophodna komunikacija sa aktorom iz drugog mikroservisa, neophodno je u okviru njegove implementacije referisati na taj aktor. To se postiže pozivom *ActorSelection* na osnovu adrese aktora u klasteru. U nastavku je data implementacija klase Program.cs mikroservisa Ordering, gde se detaljno može videti opisana konfiguracija.

```
using Akka.Actor;
using Akka.Cluster;
using Akka.Cluster.Tools.PublishSubscribe;
using Akka.Configuration;
using OrderingService;
```

```
6  using OrderingService.Actors;
7
8  var builder = WebApplication.CreateBuilder(args);
9  var config = ConfigurationFactory.ParseString(File.ReadAllText("
       akka.conf"));
10
11 //Create actor system and get cluster
12 using var system = ActorSystem.Create("OrderingAPI", config);
13 var cluster = Cluster.Get(system);
14 List<Address> addresses = new List<Address>();
15 addresses.Add(cluster.SelfAddress);
16 cluster.JoinSeedNodes(addresses);
17 var mediator = DistributedPubSub.Get(system).Mediator;
18
19 //Get productCatalogActor
20 var productCatalogActorSelection = system.ActorSelection("akka.tcp
       ://ProductCatalogAPI@127.0.0.1:8082/user/productCatalogActor");
21 var productCatalogActorRef = await productCatalogActorSelection.
       ResolveOne(TimeSpan.FromSeconds(25));
22
23 //Get CartActor
24 var cartActorSelection = system.ActorSelection("akka.tcp://
       CartAPI@127.0.0.1:8081/user/cartActor");
25 var cartActorRef = await cartActorSelection.ResolveOne(TimeSpan.
       FromSeconds(25));
26
27 var cartCoordinatorActorSelection = system.ActorSelection("akka.tcp
       ://CartAPI@127.0.0.1:8081/user/cartCoordinatorActor");
28 var cartCoordinatorActorRef = await cartActorSelection.ResolveOne(
       TimeSpan.FromSeconds(25));
29
30 //Create orderingCoordinatorActor
31 var orderingCoordinatorActorRef = system.ActorOf(Props.Create(() =>
        new OrderingCoordinatorActor(productCatalogActorRef,
       cartCoordinatorActorRef)), "orderingCoordinatorActor");
32
33 mediator.Tell(new Put(orderingCoordinatorActorRef));
34 var orderingSvc = new OrderingSvc(orderingCoordinatorActorRef);
35 builder.Services.AddSingleton<IOrderingService>(orderingSvc);
36
37 // Add services to the container.
38 builder.Services.AddControllers();
```

```
39
40 // Learn more about configuring Swagger/OpenAPI at https://aka.ms/
       aspnetcore/swashbuckle
41 builder.Services.AddEndpointsApiExplorer();
42 builder.Services.AddSwaggerGen();
43 var app = builder.Build();
44
45 // Configure the HTTP request pipeline.
46 if (app.Environment.IsDevelopment())
47 {
48     app.UseSwagger();
49     app.UseSwaggerUI();
50 }
51 app.UseHttpsRedirection();
52 app.UseAuthorization();
53 app.MapControllers();
54 app.Run();
```

5.4 Perzistencija

U ovom sistemu perzistencija se ostvaruje upotrebom biblioteke *Akka.Persistence* biblioteke, koja pruža mehanizme za trajno čuvanje stanja aktora. Ovo je važno za sisteme koji imaju zahtev za održavanje stanja aktora tokom vremena i žele da se to stanje sačuva čak i ako se sistem obustavi ili pokrene ponovo. Svaki mikroservis koristi sopstveni perzistentni sloj za čuvanje podataka, omogućavajući nezavisno upravljanje stanjem svakog aktora.

Implementacija perzistencije se postiže kroz nasleđivanje odgovarajućih klasa u okviru biblioteke *Akka.Persistence*. Na primer, klasa *UntypedPersistentActor* se koristi za aktore koji imaju perzistentno stanje. Aktori mogu sačuvati događaje ili snimke stanja na odgovarajući perzistentni sloj (na primer, SQL bazu podataka) koristeći metode poput *Persist* ili *PersistAsync*. Kada se aktor rekonstruiše, može se povratiti prethodno stanje putem metode *OnRecover*.

Perzistentni sloj se konfiguriše u okviru svakog mikroservisa, gde se definiše sistem za upravljanje bazom podataka (u ovom slučaju, SQL Server), veza ka bazi podataka i ostale relevantne postavke. Baza podataka se sastoji od tri tabele:

- *EventJournal*, koja sadrži informacije o svakom događaju koji se dešavao nad aktorima. Svaki put kada se pošalje poruka aktoru i on je obradi, generiše se

novi događaj koji se zapisuje u ovoj tabeli. Ovi događaji se mogu povratiti i koristiti za obnovu stanja aktora nakon ponovnog pokretanja aplikacije. Ovaj pristup omogućava visok stepen pouzdanosti sistema i sprečava gubitak podataka u slučaju neočekivanih prekida ili pada aplikacije.

- *SnapshotStore*, koja sadrži trenutno stanje aktora koje je sačuvano kao snimak (*eng. snapshot*). Ova tabela služi da se ubrza proces rekonstrukcije stanja aktora, tako što se umesto čitanja svih događaja od početka, može pročitati samo najnoviji snimak i nastaviti čitanje događaja od tog trenutka.

- *Metadata*, koja sadrži metapodatke o stanju aktora, kao što su poslednji broj sekvence događaja i vreme kada je poslednji događaj zapisan. Ovi podaci se koriste da bi se omogućila efikasnija čitanja događaja određenog aktora, kao i da bi se proverilo da li je događaj već obrisan ili nije

Svaki mirkoservis konfigurisan je tako da koristi ovu biblioteku i svaki od njih ima posebno definisanu bazu podataka. U nastavku je data konfiguracija perzistentnog sloja za mikroservis *Ordering*.

```
1  akka.persistence {
2      journal {
3          plugin = "akka.persistence.journal.sql-server"
4          sql-server {
5              class = "Akka.Persistence.SqlServer.Journal.
       SqlServerJournal, Akka.Persistence.SqlServer"
6              connection-string = "Data Source=(localdb)\\
       MSSQLLocalDB;Initial Catalog=OrderingDB;Integrated Security=True
       "
7              schema-name = dbo
8              table-name = EventJournal
9              auto-initialize = on
10         }
11     }
12     snapshot-store {
13         plugin = "akka.persistence.snapshot-store.sql-server"
14         sql-server {
15             class = "Akka.Persistence.SqlServer.Snapshot.
       SqlServerSnapshotStore, Akka.Persistence.SqlServer"
16             connection-string = "Data Source=(localdb)\\
       MSSQLLocalDB;Initial Catalog=OrderingDB;Integrated Security=True
       "
```

```
17              schema-name = dbo
18              table-name = SnapshotStore
19              auto-initialize = on
20          }
21      }
22  }
```

Implementacija perzistencije na ovaj način pruža pouzdan mehanizam za čuvanje stanja aktora u mikroservisnoj arhitekturi, omogućavajući skalabilnost i elastičnost sistema, kao i otpornost na greške.

Glava 6

Zaključak

Reaktivni sistemi su se u velikom broju slučajeva pokazali kao ključni za izgradnju modernih aplikacija koje moraju biti otporne, elastične i skalabilne. Kombinovanje aktor modela i mikroservisne arhitekture sa reaktivnim principima omogućava kreiranje sistema koji je visoko odzivan, tolerantan na greške i sposoban da se prilagodi promenljivim zahtevima.

U ovom radu dat je prikaz aktor modela, mikroservisne arhitekture i reaktivnih sistema. Razvijena je aplikacija za internet prodavnicu zasnovana na ovim konceptima.

Aktor model se pokazao kao moćan pristup za izgradnju reaktivnih sistema. Aktori kao jedinice izvršavanja pružaju visoku granularnost i izolaciju, što olakšava paralelno izvršavanje i skaliranje sistema. Moguće je lako dodavati nove instance aktora i mikroservisa kako bi se nosili sa rastućim opterećenjem. Sistem se može horizontalno skalirati i distribuirati na više čvorova. Asinhrona komunikacija i paralelno izvršavanje doprinose visokoj odzivnosti sistema i smanjuju potrebu za čekanjem na odgovore. Razlaganje aplikacije na manje, autonomne servise omogućava efikasno upravljanje funkcionalnostima i lakše održavanje i proširenje sistema. Svaki mikroservis se može skalirati, nadograditi i zameniti bez uticaja na ostale delove sistema.

U poređenju sa tradicionalnim pristupom, gde se koristi klasična monolitna arhitektura ili arhitektura sa više niti, aktor model se izdvaja po svojoj fleksibilnosti i skalabilnosti. Mikroservisi omogućavaju nezavisno razvijanje i održavanje komponenti sistema, dok aktor model pruža apstrakciju od složenosti višenitnog programiranja i olakšava razvoj sistema sa paralelnim i distribuiranim izvršavanjem. Ova implementacija omogućava brže vreme odziva, bolju skalabilnost i otpornost

na greške.

Sa druge strane, ovakva implementacija nosi i neke izazove i mane. Kompleksnost sistema se povećava, jer zahteva pravilno modeliranje aktora i konfiguraciju klastera. Aktor model nije univerzalno rešenje i može biti nepotrebno složen za manje aplikacije ili scenarije koji ne zahtevaju visoku paralelnost i distribuciju. Takođe, potrebno je pažljivo razmotriti granulaciju mikroservisa i definisati jasne granice između njih kako bi se izbegla prekomerna međusobna zavisnost.

Literatura

[1] Akka documentation. https://akka.io/.

[2] Microsoft orleans documentation. https://learn.microsoft.com/en-us/dotnet/orleans/.

[3] .net documentation. https://learn.microsoft.com/en-us/dotnet/.

[4] Proto.actor. https://proto.actor/.

[5] What is akka.net? https://getakka.net/articles/intro/what-is-akka.html.

[6] The reactive manifesto. https://www.reactivemanifesto.org/, 2014.

[7] Akkashop. https://github.com/AlexandraDotlic/AkkaShop.git, 2023.

[8] Christian Baxter and Richard Imaoka. *Mastering Akka.* Packt Publishing, Birmingham, UK, 2016.

[9] Jonas Bonér. *Reactive microservices architecture.* O'Reilly Media, Incorporated, 2016.

[10] Anthony Brown. *Reactive applications with Akka.NET.* Manning Publications Co., Shelter Island, NY, 2019.

[11] Carl Hewitt. Actor Model of Computation: Scalable Robust Information Systems, 2015. http://arxiv.org/abs/1008.1459.

[12] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[13] Sam Newman. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media, 1st edition, February 2015.

[14] Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka in Action.* Manning Publications, Shelter Island, NY, 2016.