UNIVERZITET U NIŠU

ELEKTRONSKI FAKULTET

**PROJEKTOVANJE I IMPLEMENTACIJA SIGURNE MIKROSERVISNE APLIKACIJE**

Seminarski rad

Studijski program: Računarstvo i informatika

Modul: Inženjerstvo podataka

Student: Mentor:

Aleksandar Stojković, br. ind. 1915 Prof. dr Dragan Janković

Prof. dr Petar Rajković

Niš, avgust 2025. godina

PROJEKTOVANJE I IMPLEMENTACIJA SIGURNE MIKROSERVISNE APLIKACIJE

**SAŽETAK**

Evolucija tehnika razvoja softvera teži ka postizanju što bržeg razvoj što efikasnijih aplikacija. Samim tim, veliki akcenat se stavlja na paralelizaciji razvoja. Iz tog ugla, mikroservisna arhitektura predstavlja optimalan izbor, te je jasno zašto je upravo ona sve popularnija danas. Ipak, kako u projektovanju softvera ne postoji rešenje koje je univerzalno bolje od ostalih, ni razvoj aplikacije kao skup komunicirajućih mikroservisa nije bez svojih nedostataka. U ovom slučaju, ti nedostaci se prvenstveno ogledaju u pitanjima bezbednosti i pouzdanosti.

Naime, neophodnost međusobne komunikacije mikroservisa za pravilno funkcionisanje sistema ujedno predstavlja i povećanu površinu za napade malicioznih aktera, kao i povećan broj potencijalnih tačaka otkaza zbog nepouzdanosti prenosa informacija preko mreža. Cilj ovog rada jeste da prouči mogućnosti zaštite komunikacije između mikroservisa i sprečavanja napada na nju, primenom principa koji garantuju poznavanje identiteta učesnika u komunikaciji kao i bezbednost razmenjenih poruka od neovlašćenog pristupa. Takođe, razmatraju se i tehnika kojima se postiže veći nivo pouzdanosti pri komunikaciji, odnosno pritupi koji garantuju da poruka bude isporučena makar jednom. Deo rada je, pored toga, posvećen i problemu distribuiranih transakcija i načinima za delimično prevazilaženje tog problema.

Na samom početku rada, u okviru drugog poglavlja, dato je objašnjenje različitih tipova arhitektura, njihovih prednosti i mana. Zatim, u trećem poglavlju, sledi opis TLS protokola za bezbednu komunikaciju. Nakon toga, u četvrtom poglavlju, dat je presek projektnih obrazaca za postizanje pouzdane komunikacije u mikroservisnim arhitekturama. Peto poglavlje je posvećeno konkretnoj demonstrativnoj implementaciji i upotrebi prethodno navedenih principa i tehnika. Na kraju, u šestom poglavlju dat je zaključak sa osvrtom na moguća unapređenja sistema.

**Ključne reči**: mikroservisna arhitektura, komunikacija, bezbednost, TLS, pouzdanost, distribuirane transakcije.

SECURE MICROSERVICE APPLICATION DESIGN AND IMPLEMENTATION

**ABSTRACT**

The evolution of software development techniques tends toward achieving faster development of more efficient applications. Consequently, a strong emphasis is placed on parallelizing the development. From this perspective, the microservice architecture represents the optimal choice, which explains why it is becoming an increasingly popular option today. However, as is the case in software design, there is no single solution that is clearly superior to all others; thus, developing an application as a set of communicating microservices also comes with its own drawbacks. In this case, the main concerns are security and reliability.

Specifically, the necessity for microservices to communicate in order to function correctly also increases the attack surface for malicious actors and the number of potential system failures due to the unreliability of network communication. The aim of this paper is to examine the possibilities for securing communication between microservices and preventing attacks on it by applying principles that ensure knowledge of the participants’ identities and protect the exchanged messages from unauthorized access. Furthermore, techniques that aim to increase the reliability of communication—such as approaches that guarantee at-least-once delivery of messages—are also considered. Additionally, part of this paper is dedicated to the issue of distributed transactions and to methods that can help mitigate this problem to some extent.

At the beginning of the paper, in the second chapter, an explanation of different types of architectures is provided, along with their advantages and disadvantages. Then, in the third chapter, the TLS protocol for secure communication is described. After that, the fourth chapter presents an overview of design patterns aimed at achieving reliable communication in microservices. The fifth chapter is dedicated to a concrete demonstrative implementation and the application of the previously mentioned principles and techniques. Finally, the sixth chapter contains a conclusion with a reflection on possible further improvements to the system.

**Keywords**: microservice architecture, communication, security, TLS, reliability, distributed transactions.

SPISAK KORIŠĆENIH SKRAĆENICA

SKRAĆENICE

UX **U**ser e**x**perience

UI **U**ser **I**nterface

API **A**pplication **P**rogramming **I**nterface

REST **R**epresentational **S**tate **T**ransfer

WSDL **W**eb **S**ervice **D**efinition **L**anguage

XML E**x**tensible **M**arkup **L**anguage

ESB **E**nterprise **S**ervice **B**us

DoS **D**enial **o**f **S**ervice

TLS **T**ransport **L**ayer **S**ecurity

TCP **T**ransmission **C**ontrol **P**rotocol

ISO **I**nternational **S**tandardization **O**rganization

OSI **O**pen **S**ystems **I**nterconnection

IP **I**nternet **P**rotocol

SSL **S**ecure **S**ocket **L**ayer

MAC **M**essage **A**uthentication **C**ode

AEAD **A**uthenticated **E**ncryption with **A**ssociated **D**ata

JSON **J**ava**S**cript **O**bject **N**otation

JWT **J**SON **W**eb **T**oken

RPC **R**emote **P**rocedure **C**all

TCC **T**ry-**C**onfirm/**C**ancel

2PC **2** **P**hase **C**ommit

Sadržaj

[**1.** **UVOD** 6](#_Toc204276716)

[**2.** **ARHITEKTURE APLIKACIJA** 7](#_Toc204276717)

[**2.1.** Monolitna arhitektura 8](#_Toc204276718)

[**2.2.** Slojevita arhitektura 10](#_Toc204276719)

[**2.3.** Modularni monolit 11](#_Toc204276720)

[**2.4.** Servisno-orijentisana arhitektura 12](#_Toc204276721)

[**2.5.** Mikroservisna arhitektura 14](#_Toc204276722)

[**3.** **BEZBEDNA KOMUNIKACIJA** 16](#_Toc204276723)

[**3.1.** TLS 18](#_Toc204276724)

[**3.2.** Autentifikacija i autorizacija klijenata 23](#_Toc204276725)

[**4.** **POUZDANA KOMUNIKACIJA KOD MIKROSERVISA** 25](#_Toc204276726)

[**4.1.** Obrasci za upravljanje distribuiranim transakcijama 27](#_Toc204276727)

[**4.2.** Obrasci za pouzdano slanje poruka 31](#_Toc204276728)

[**5.** **IMPLEMENTACIJA** 33](#_Toc204276729)

[**5.1.** Auth mikroservis 34](#_Toc204276730)

[**5.2.** Keš mikroservis 42](#_Toc204276731)

[**5.3.** Profil mikroservis 48](#_Toc204276732)

[**5.4.** Distribuirane transakcije 54](#_Toc204276733)

[**6.** **ZAKLJUČAK** 60](#_Toc204276734)

[**7.** **LITERATURA** 61](#_Toc204276735)

## **UVOD**

Upotreba kompjuterskih programa je danas neizostavni deo svakodnevnog života. Programi, odnosno softver, su svoju primenu našli u svim sferama društva. Uz tako veliku različitost primena prirodno sledi i velika raznovrsnost i kompleksnost softvera. Pored toga, zbog konkurencije između različitih proizvođača softvera stalno se traže nove metode i pristupi koji pružaju veći kvalitet ali i veću brzinu razvoja softvera. Međutim ne postoji nijedan pristup koji je univerzalno bolji od drugih, već svi oni imaju određene prednosti i mane, te je potrebno izabrati onaj pristup koji najviše odgovara datim okolnostima razvoja konkretnog softvera.

Jedan od trenutno najaktuelnijih pristupa pri razvoju softvera je upotreba takozvane mikroservisne arhitekture. Mikroservisna arhitektura je pristup razvoju jedne aplikacije kao skupa malih usluga (servisa), tako da se svaka razvija, pokreće, izvršava i održava nezavisno od ostalih. Pri tome, ti servisi po potrebi razmenjuju informacije korišćenjem nekog protokola komunikacije. Glavna ideja ovakvog pristupa je da se upravo tom nezavisnošću između servisa olakša i ubrza njihov razvoj jer nezavisnost servisa znači da se oni mogu posebno u paraleli razvijati od strane različitih timova, a da pri tome timovi praktično ne moraju da imaju nikakve informacije o razvojnim odlukama drugih timova. Međutim, neizbežna potreba da servisi komuniciraju međusobno, sa sobom povlači određeni broj novih problema koje je neophodno rešiti za bezbedan rad aplikacije. Pored komunikacije između servisa, aplikacija komunicira, odnosno razmenjuje poruke i sa samim korisnicima aplikacije što takođe donosi određene probleme koje je potrebno rešiti.

Komunikacija preko interneta sama po sebi nije bezbedna. Dakle, potrebno je preduzeti određene mere kako bi se osiguralo da se ta komunikacija izvrši na način koji garantuje da će poslate poruke zapravo stići do svog odredišta a da u toku samog njihovog prenosa do odredišta nisu izmenjene niti da je bilo koje strano lice moglo da pročita njihov sadržaj. Postoje više pristupa, odnosno projektnih obrazaca za rešavanje prethodno navedenih problema. Oni sami po sebi takođe imaju različite prednosti i manje te ih je potrebno birati spram potreba aplikacija.

Ovaj rad se bavi prethodno navedenim temama, konkretno rad će biti dat u sledećoj formi. Drugo poglavlje rada biće posvećeno poređenju različitih arhitektura, njihovim prednostima i manama i generalnim savetima za njihovu upotrebu, sa akcentom na mikroservisnoj arhitekturi. U trećem poglavlju će protokoli koji enkripcijom obezbeđuju bezbednu komunikaciju preko interneta biti detaljnije objašnjeni. Zatim, u četvrtom poglavlju, biće reči o obrascima koji omogućavaju pouzdanu komunikaciju čak i prilikom privremenih nedostupnosti komunikacionog medijuma. Peto poglavlje je posvećeno komunikaciji sa korisnicima aplikacija. U šestom poglavlju je opisana konkretna implementacija sistema od tri mikroservisa u slučaju razvoja aplikacije tipa društvene mreže, sa konkretno izabranim pristupima i obrascima koji odgovaraju datim zahtevima aplikacije. Naposletku u sedmom poglavlju je dat zaključak zajedno sa diskusijom o mogućim budućim unapređenjima ovakvog sistema.

## **ARHITEKTURE APLIKACIJA**

Arhitektura aplikacije opisuje obrasce i tehnike korišćene za projektovanje i izgradnju aplikacije. Arhitektura daje mapu puta i najbolje prakse koje treba pratiti prilikom razvoja aplikacije, tako da se na kraju dobija dobro struktuirana aplikacija [1]. Dakle arhitekturom se smatra skup svih pravila, nezavisnih od same tehnologije koja se koristi za razvoj aplikacije, koje treba pratiti prilikom donošenja odluka u toku razvoja softvera.

U suštini, arhitekture se sastoje od projektnih obrazaca. Projektni obrazac opisuje ponovljivo rešenje nekog problema [1]. Kako projektni obrasci pa i same arhitekture predstavljaju pristupe rešavanju problema koji se prošli proveru vremena preporučljiva je njihova upotreba prilikom razvoja aplikacija.

Glavni faktori evolucije arhitekture softvera jesu zahtevi korisnika i težnja ka ostvarivanju što boljeg iskustva korisnika (UX) kao i zahtevi koji proističu iz konkurencije različitih proizvođača softvera. Iskustvo korisnika se generalno odnosni na iskustvo koje klijent ili korisnik ima prilikom interakcije sa proizvodom, sistemom ili servisom. Iskustvo korisnika uzima u obzir sve utiske korisnika o proizvodu ili servisu, uključujući lakoću upotrebe, pristupačnost, vizuelni dizajn, funkcionalnosti korisničkog interfejsa (UI) i emocionalni utisak proizvoda [2]. Ovi faktori generalno za cilj imaju što brži razvoj što kvalitetnijeg softvera. Pri tome, se kvalitet softvera odnosi na stepen u kome softver odgovara zahtevima definisanim za taj softver i stepen u kome on zadovoljava potrebe svojih korisnika. Formalno se definiše kao sposobnost softvera da zadovolji navedene i implicitne potrebe prilikom upotrebe u specificiranim okolnostima [3].

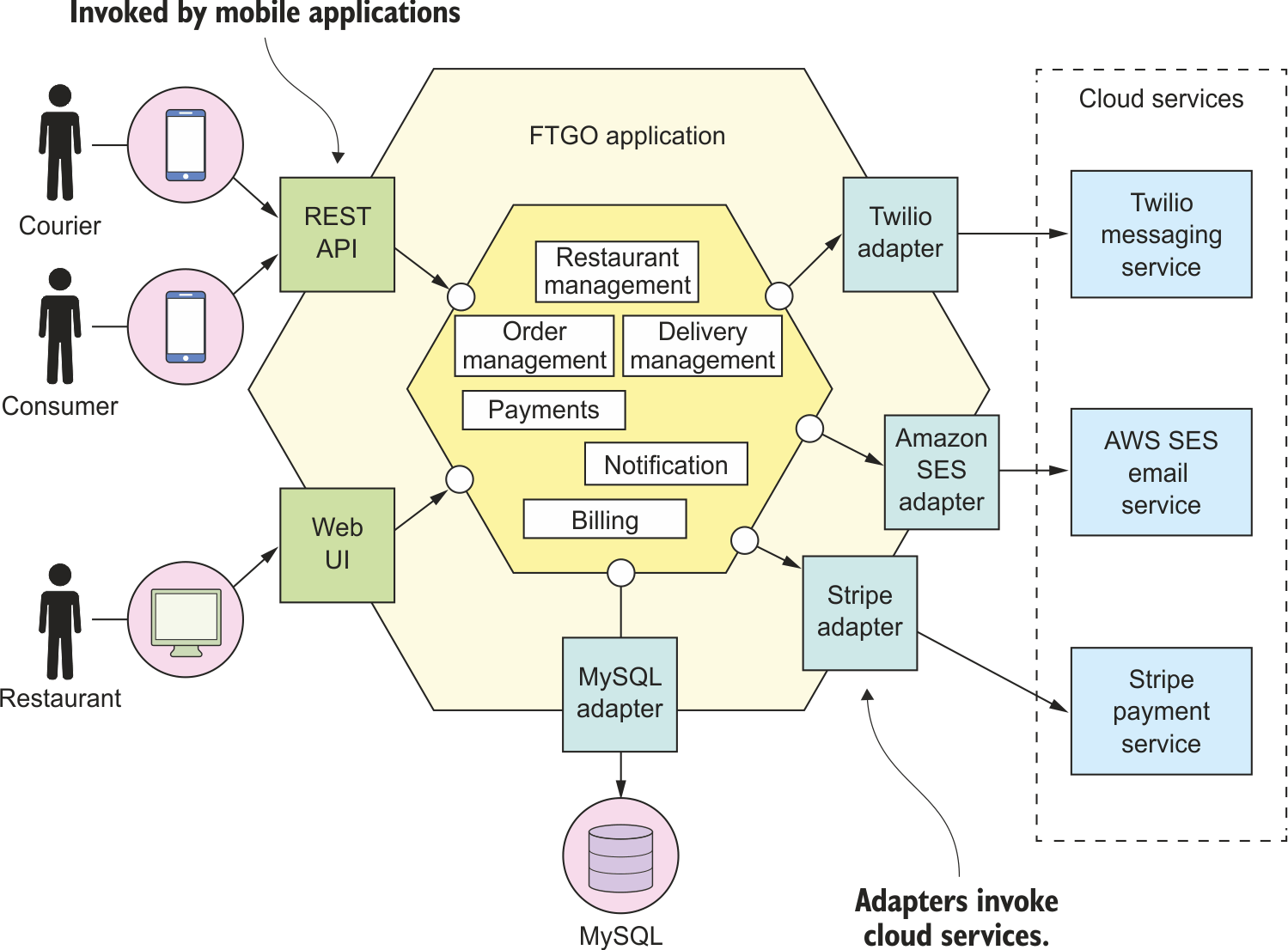
Shodno tome, kvalitet softvera se ogleda kroz osobine softvera kao što su:

* Visoka dostupnost: korisnik očekuje da može da koristi aplikaciju u bilo kom njemu odgovarajućem trenutku.
* Niska latentnost softvera: svaka operacija se treba izvršiti u razumnom vremenu i to po mogućnosti što kraćem.
* Pouzdanost softvera: od softvera se očekuje da ne pravi greške, da podaci koje pruža budu tačni i da zahtevi korisnika uspešno budu izvršeni.
* Otpornost na greške: čak i kada dođe do greške odnosno pada softvera od njega se očekuje da se oporavi uz minimalne poremećaje rada.
* Skalabilnost softvera: sposobnost softvera da odgovori na povećanje broja zahteva srazmernim povećanjem upotrebe resursa. Važno je razlikovati vertikalnu i horizontalnu skalabilnost. Vertikalna skalabilnost se postiže upotrebom jačih računara odnosno komponenti računara za izvršavanje aplikacije i samim time je jasno ograničena. Horizontalna skalabilnost se postiže upotrebom većeg broja računara na kojima se izvršava deo ili cela aplikacija u paraleli sa ostalim računarima. Ovaj pristup postizanja skalabilnosti u teoriji nije ograničen te je zato često od većeg interesa nego vertikalno skaliranje.

Težnja za zadovoljavanjem ovih faktora je upravljala evolucijom arhitektura koje se koriste prilikom razvoja softvera. Arhitekture se mogu podeliti na: monolitnu arhitekturu, slojevitu arhitekturu, modularno-monolitnu arhitekturu, servisno-orijentisanu arhitekturu i mikroservisnu arhitekturu. Gde je danas mikroservisna arhitektura najpopularnija, do te mere da se arhitekture koje njoj prethode smatraju takozvanim legacy arhitekturama. Međutim, opšte prihvaćeni stav je da je bolje početi razvoj aplikacije praćenjem arhitektura monolitnog tipa pa tek onda po potrebi izdvajati mikroservise [4].

### Monolitna arhitektura

Monolitna arhitektura predstavlja tradicionalni model softverske aplikacije, koja je izgrađena kao jedinstvena celina, samostalna i nezavisna od drugih aplikacija. Termin ,,monolit“ se često povezuje sa nečim velikim i statičnim, što prilično dobro oslikava prirodu monolitne softverske arhitekture. U monolitnoj arhitekturi, celokupan sistem funkcioniše kao jedna velika celina sa jedinstvenom bazom koda, u kojoj su sve poslovne funkcionalnosti čvrsto povezane. Izmena kod ovakvog tipa aplikacije povlači izmene na svim nivoima aplikacije, pristupanjem bazi koda i ponovnim build-om i deployment-om izmenjene verzije interfejsa sa servisne strane. Ovakav pristup je ograničavajući i vremenski zahtevan [5].



Slika 2.1. Primer monolitne arhitekture, slika preuzeta iz [6]

Na slici 2.1 je dat primer jedne monolitne arhitekture aplikacije za naručivanje hrane. Kao što je prethodno pomenuto, aplikacija je jedinstvena celina zadužena za obavljanje svih funkcija počevši od korisničkih interfejsa za komunikacije sa različitim tipovima korisnika spram njihovih potreba. Naime naručiocima hrane, dostavljačima i restoranima koji tu hranu pripremaju nisu iste funkcionalnosti na raspolaganju i od interesa, te samim tim svaka od ovih grupa korisnika interaguje za zasebnim interfejsom, koji dalje, u zavisnosti od organizacije aplikacije, mogu da komuniciraju različitim pristupima koji ne moraju nužno biti međusobno izolovani. Pored toga, u jezgru aplikacije nalazi se poslovna logika, odnosno skup modula zaduženih za izvršavanje svih funkcionalnosti koje aplikacija pruža. Moduli mogu biti logički izdvojeni i izolovani, u smislu da su zaduženi za različite grupe poslovnih funkcionalnosti ali suštinski su deo jedne jedinstvene aplikacije. Logičko izdvajanje se najčešće vrši putem definisanja različitih klasa za svaku od funkcionalnosti. Zatim, postoje i adapteri koji su zaduženi za komunikaciju sa spoljnim nezavisnim servisima. U adaptere spadaju ulazni adapteri, poput REST API-ja i Web UI-a sa slike, koji služe za komunikaciju sa korisničkim interfejsima odnosno, samim tim i korisnicima. Izlazni adapteri, s druge strane, služe da omoguće poslovnoj logici da pristupi bazi podataka, u ovom konkretnom slučaju MySQL bazi, pored toga izlazni adapteri pozivaju cloud servise.

Prednosti pri upotrebi monolitne arhitekture, koje su posebno izražene pri razvoju manjih aplikacija, odnosno pri razvoju aplikacije od strane manjeg broja programera, su:

* Lakoća razvoja: aplikacija se sastoji od jedne baze koda, te je jednostavnije razviti je [2][5].
* Lak deployment: monolitna arhitektura se sastoji od jednog izvršnog fajla odnosno od jednog direktorijuma što čini deployment jednostavnijim [2][5].
* Održavanje: pored samog deployment-a zbog manjeg broja komponenti monolitne arhitekture su lakše za održavanje [2][5].
* Lako debagovanje i testiranje: kako se sav kod nalazi na jednom mestu lakše je pratiti propagaciju zahteva i pronaći problem [2][5].
* Performanse: zbog centralizovane baze koda jedan API može da izvrši celokupan zahtev, te nema gubitka performansi zbog komunikacije [2][5].
* Bezbednost: kako je monolitna arhitektura zatvoren sistem, prostor za izvršenje sajber napada je minimalan. Naime, svaka komunikacija preko nekog medijuma, najčešće interneta pruža novu metu za napad. Kako se kod monolita nužno vrši samo komunikacija sa korisnicima preko API-ja tako su API jedine mete potencijalnog napada [2][5].

Shodno tome, monolitne aplikacije mogu biti prilično efektivne dok ne dostignu određeni nivo veličine, nakon čega skaliranje postaje izazovno. Sitne izmene jedne funkcije zahtevaju kompajliranje i testiranje celokupne aplikacije što se kosi sa modernim pristupom agilnog programiranja [5]. Dakle nedostaci monolitne arhitekture su:

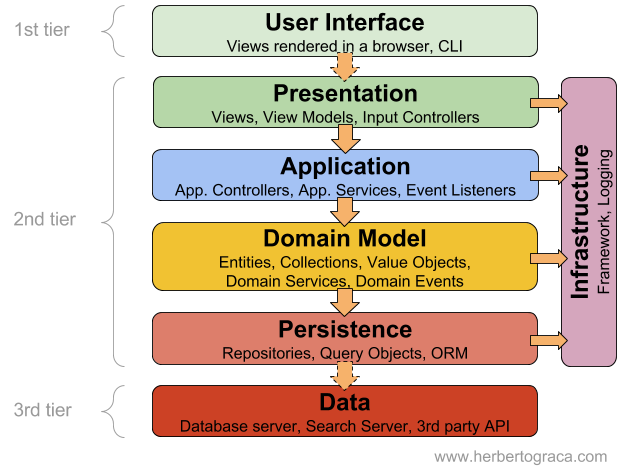
* Sporost razvoja: razvoj velikih monolitnih aplikacija je kompleksan i spor [5].
* Skalabilnost: monolitne aplikacije podržavaju i vertikalno i horizontalno skaliranje. Horizontalno skaliranje se postiže izvršavanjem kompletnih replika aplikacije na različitim računarima odnosno serverima. Replike se nalaze iza load balancer-a koji je zadužen za izbor kojoj replici treba proslediti svaki od zahteva. Na ovaj način se postiže zadovoljavajući nivo skalabilnosti. Međutim, svi zahtevi se izvršavaju nad istom bazom podataka što potencijalno može negativno uticati na performanse. Ukoliko opterećenje nije preveliko za bazu, sve replike komuniciraju sa istom instancom baze. Ali, sa porastom broja korisnika aplikacije i samim time broja replika u nekom trenutku postaje neophodna podela baze na klaster baza. Podela na klaster, u zavisnosti od mogućnosti same baze koja se upotrebljava, može biti izvršena vertikalnom podelom slogova u bazi, horizontalnom podelom slogova i kreiranjem replika baze koje se nalaze u takozvanom master-slave odnosu [7]. Pored svega ovoga, skaliranje monolitne aplikacije nije potpuno optimalno iz razloga što se uvek replicira celokupno aplikacija što znači da se troše resursi i na delove aplikacije koji ne podležu povećanom broju zahteva. Na primer, deo za registraciju korisnika, u opštem slučaju, treba da obradi mnogo manji broj zahteva u nekom trenutku nego recimo deo koji omogućava razmenu poruka između korisnika.
* Pouzdanost: postojanje greške u jednom modulu može utiče na celokupnu aplikacije i da dovede i do pada aplikacije [5].
* Otpornost prema izmeni tehnologija i manjak fleksibilnosti: kako je monolitna aplikacija tesno povezan zatvoren sistem, integrisanje novih tehnologija može biti teško [2].
* Skupoća izmena: izmene su spore, a pored toga veoma su zahtevne iz ugla deployment-a jer svaka izmena zahteva deployment celokupnog monolita [5].

### Slojevita arhitektura

Stil slojevite arhitekture je jedan od istorijski najčešće korišćenih stilova. Glavna ideja slojevite arhitekture je da se moduli odnosno komponente sa fizički sličnim funkcionalnostima organizuju u horizontalne slojeve. Posledica ovakvog pristupa je da svaki sloj vrši neku specifičnu ulogu u aplikaciji. U okviru slojevite arhitekture ne postoje ograničenja po pitanju broja slojeva aplikacije, pošto je cilj postojanja slojeva promovisanje koncepta razdvajanja uloga (separation of concerns). Slojevita arhitektura apstrahuje pogled na sistem kao celinu, pružajući pri tome dovoljno detalja za razumevanje uloga i odgovornosti svakog pojedinačnog sloja i njihovog međusobnog odnosa [8].

Veza između svaka dva slojeva može biti poziv funkcije, upit, objekat ili bilo koji konektor koji prenosi zahteve ili informacije [8]. Broj i izbor slojeva je fleksibilan ali su slojevi poput prezentacionog sloja, sloja biznis logike, sloja za perzistenciju i sloja baza podataka uvek prisutni. Prezentacioni sloj sadrži sve klase odgovorne za prezentovanje korisničkog interfejsa, odnosno, vizualizacije krajnjim korisnicima, pored toga ovaj sloj sadrži logiku za komunikaciju sa internet pretraživačem. Poželjno je da korisnik direktno interaguje sa ovim slojem [8]. Sloj biznis logike sadrži svu logiku koja je neophodna za postizanje funkcionalnih zahteva. Ovaj sloj se bavi agregacijom podataka, izračunavanjima i kreiranjem upita [8]. Sloj za perzistenciju i sloj baza podataka je zadužen za pribavljanje i skladištenje podataka i sastoji se od logičkih i fizičkih apsekata potrebnih za izvršenje istih. Logička šema specificira konceptualni model podataka, dok fizička šema implementira logički model u fizičkoj platformi baze [8]. Na slici 2.2 je dat primer slojevite arhitekture.

U užem smislu značenja monolitna arhitektura se odnosi na činjenicu da se cela aplikacija sastoji od jednog projekta i na sam način obrade zahteva u okviru tog jednog projekta, tako da svi delovi aplikacije učestvuju prilikom izvršavanja jednog zahteva. U širem smislu značenja monolitne arhitekture svrstavaju se sve aplikacije koje se sastoje od samo jednog projekta, te se može smatrati da je slojevita arhitektura jedna od njenih pod tipova, razlog za to je činjenica da su slojevi samo logički izolovani dok su fizički i dalje deo jednog projekta te se vrši deployment cele aplikacije odjednom. Dakle slojevita arhitektura pruža viši nivo modularnosti od monolitne u užem smislu, ali nema pravu modularnost, odnosno izmene jednog sloja utiču na sve ostale slojeve što dovodi do takozvanog ,,ripple“ efekta. Slojevita arhitektura je generalno jeftinija za razvoj i održavanje kao i sam monolit zbog svoje objedinjenosti. Skaliranje kao i kod monolitnih aplikacija može biti izazovno jer podrazumeva repliciranje aplikacije što znači da se ne skalira samo onaj deo aplikacije koji zahteva više resursa. Glavna prednost slojevite arhitekture je njena jednostavnost, kao što je slučaj i generalno sa monolitnom arhitekturom, funkcionalnosti su raspoređene na veoma intuitivan način i njihova horizontalna podela ih čini lakim za testiranje i debagiranje. Jedna je od fleksibilnijih monolitnih pristupa jer izmena tehnologije sloja, ukoliko je aplikacije dobro projektovana utiče na najviše dva susedna sloja. Međutim kod većih aplikacija postaje nezahvalna za održavanje kao i ostale monolitne arhitekture [10].



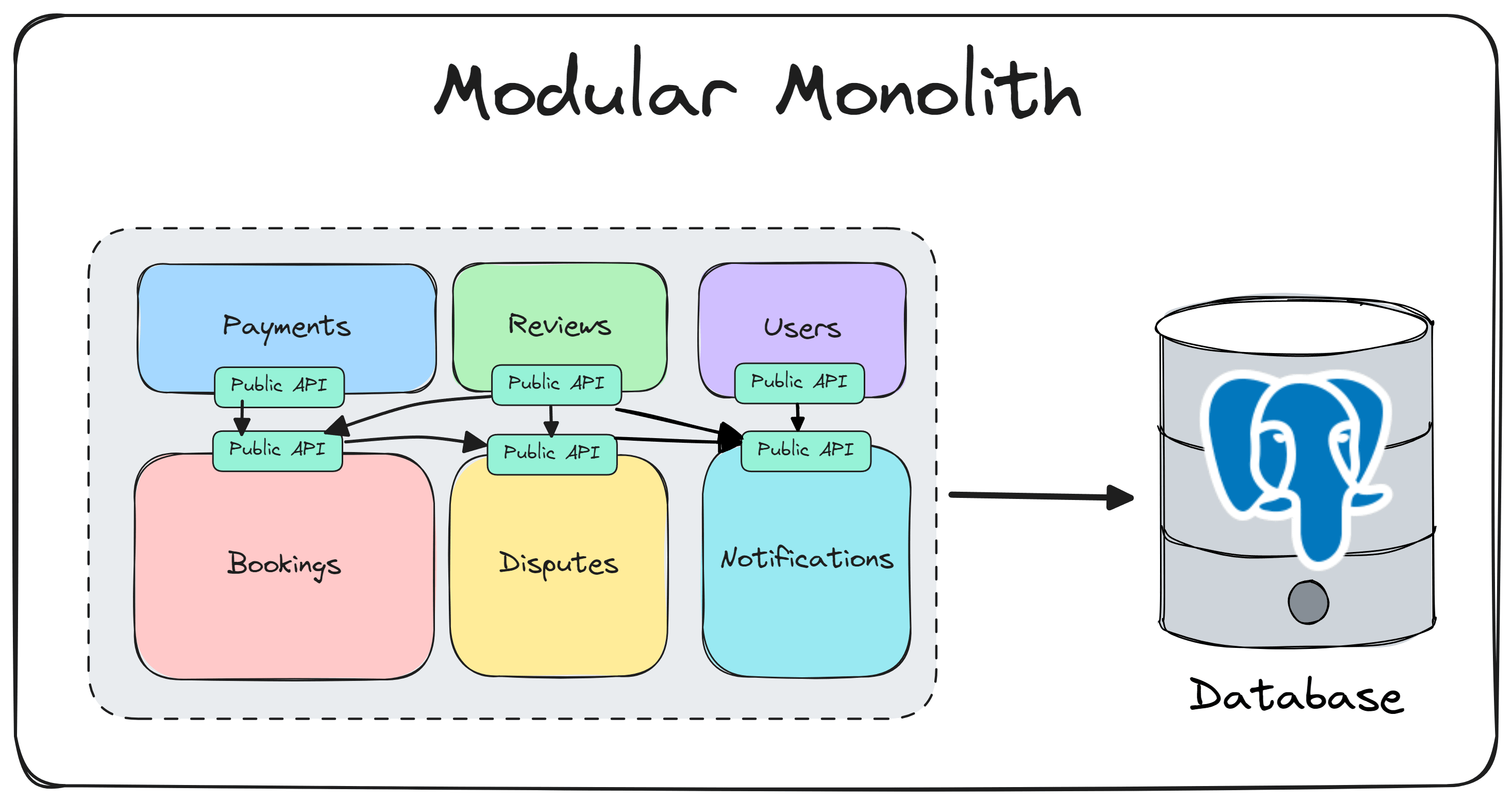
Slika 2.2. Slojevita arhitektura, slika preuzeta iz [9]

### Modularni monolit

Modularni monolit je obrazac softverske arhitekture koji strateški kombinuje jednostavnost monolitne arhitekture sa prednostima mikroservisne arhitekture. Kod ovog pristupa sistem je organizovan u slabo spregnute module, tako da je svaki modul razgraničen dobro definisanim okvirima i eksplicitnim zavisnostima od drugih modula [12]. Cilj je postizanje nezavisnosti i izolacije svakog modula, omogućavajući im pri tome da funkcionišu nezavisno dok se deploy-uju kolektivno kao jedna celina [12]. Na slici 2.3 je dat primer aplikacije turističke agencije razvijene kao modularni monolit.

Modularni monolit je pokušaj kombinacije benefita monolitne arhitekture i mikroservisne arhitekture. Kao i kod mikroservisne arhitekture to su i kod njega logičke funkcionalne celine izdvojene i dok kod mikroservisne arhitekture takva jedna celina čini jedan mikroservis kod modularnog monolita jedna logička celina je jedan moduo. Ispravno projektovani moduli su slabo spregnuti, samo po potrebi pozivaju jedne druge preko API-ja [12]. Ovakav pristup omogućava paralelno razvijanje modula sa minimalnom potrebom za komunikacijom između timova koji te module razvijaju, dakle brzina razvoja je blizu nivoa brzine razvoja mikroservisa u slučaju da na razvoju radi veći broj timova, u suprotnom benefit brzine razvoja nije prisutan ni kod jedne od ovih dveju arhitektura. Moduli nisu potpuno nezavisni jer dele zajedničku šemu baze podataka. Takođe pošto je ovo ipak tip monolitne arhitekture svi moduli se deploy-uju zajedno i svi se izvršavaju u okviru istog virtuelnog okruženja. Sama aplikacija je, dakle, jedinstven proces. Na ovaj način se postižu benefiti mikroservisnog pristupa za razvoj i fleksibilnost prema izmenama dok se zadržava jednostavnost testiranja i debagovanja monolitne arhitekture [13].

Za razliku od mikroservisa a nalik na monolitnu arhitekturu, svaki moduo mora biti razvijen korišćenjem iste tehnologije. Pored toga, mana koja je takođe zajednička sa monolitom je neizolovanost grešaka, tako da greška u jednom modulu može da dovede do pada cele aplikacije, odnosno da utiče i na sve ostale module. Modularni monolit ne pruža nikakve dodatne benefite za skalabilnost aplikacije u odnosnu na monolitnu arhitekturu [10]. Skaliranje se kao i kod monolita vrši replikacijom cele aplikacije i korišćenjem load balancer-a.



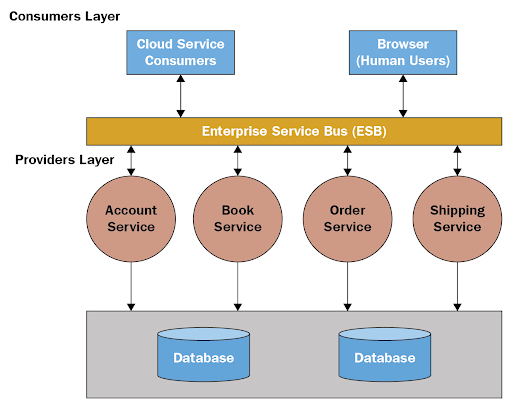
Slika 2.3. Primer modularnog monolita, slika preuzeta iz [14]

### Servisno-orijentisana arhitektura

Servisno-orijentisana arhitektura je metod razvoja softvera koji se zasniva na korišćenju komponenti, koje se nazivaju servisi, za razvoj aplikacija. Svaki servis pruža poslovne mogućnosti, servisi mogu da komuniciraju jedni sa drugima preko različitih platformi i programskih jezika [15]. Kod servisno orijentisane arhitekture akcenat je na ponovnoj upotrebi servisa u različitim sistemima i na kombinaciji vise nezavisnih servisa za izvršavanje kompleksnih zadataka. Na primer ako neki sistem zahteva korišćenje autentifikacije za više funkcionalnosti, onda se autentifikacija izdvaja kao poseban i jedinstven servis koji se koristi na svim mestima u aplikaciji. Na slici 2.4 dat je primer servisno-orijentisane arhitekture za aplikaciju za onlajn naručivanje.

Glavne komponente servisno-orijentisane arhitekture su:

* Servis: osnovni gradivni blok ove arhitekture, mogu biti privatni, tj. dostupni samo internim korisnicima neke organizacije i mogu biti javni odnosno dostupni svima preko interneta. Svaki servis se sastoji iz tri glavna delova:
  + Implementacija servisa: konkretan kod koji gradi logiku i izvršava specifičnu funkcionalnost tog servisa [15].
  + Ugovor servisa: definiše servis iz ugla uslova i odredbi tog servisa, poput preduslova, cene upotrebe i kvaliteta servisa [15].
  + Interfejs servisa: servisi međusobno komuniciraju pomoću interfejsa. Interfejs definiše kako se poziva servis kako bi izvršio svoju funkcionalnost ili razmenio podatke. Na taj način se postiže da se servisi mogu koristiti kao crna kutija, sa veoma malo ili čak i bez znanja o tome kako konkretno rade [15]. Interfejs se definiše korišćenjem jezika za definisanje web servisa (WSDL), zasnovanim na XML-u [16].
* Provajder servisa: osoba ili organizacija koja kreira, održava i pruža jedan ili više servisa za upotrebu drugima [15].
* Konzumer servisa: zahteva od provajdera servisa izvršenje nekog konkretnog servisa. Može biti ceo sistem, aplikacija ili drugi servisa [15].
* Registar servisa: skladište informacija o servisima i pristupanju servisa, kome se može pristupiti preko mreže [15].
* Servisna magistrala (ESB): obrazac koji opisuje kako treba razviti centralnu komponentu za integraciju servisa. Sva komunikacija se vrši preko servisne magistrale i ona je zadužena za transformaciju modela podataka, vezu između servisa, rutiranje i razmenu poruka nezavisno od komunikacionih protokola [16].



Slika 2.4. Primer servisno-orijentisane arhitekture, slika preuzeta iz [19]

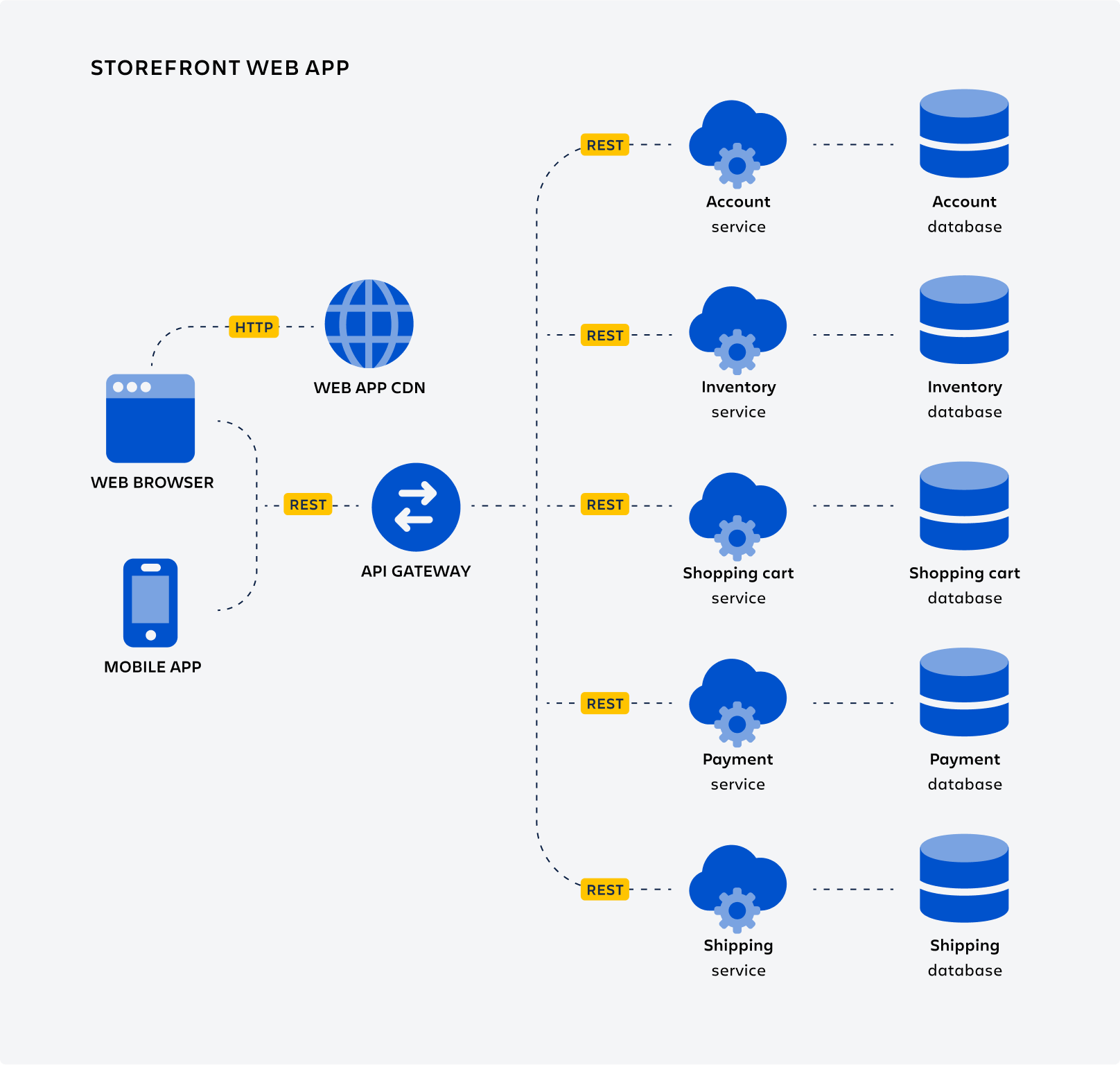
Za razliku od modularnog monolita kod servisno-orijentisane arhitekture su servisi zaista izolovani programi, tako da je svaki servis slabo spregnut sa ostalim servisima i to samo zbog potreba komunikacije sa drugim servisima. Ovakav pristup omogućava nezavisno deploy-ovanje, modifikaciju i održavanje svakog od servisa, što znatno može da ubrza vreme razvoja. Sa druge strane testiranje i debagovanje postaje značajno zahtevnije jer uključuje izvršenje više nezavisno razvijanih delova aplikacija kao i komunikacije između njih. Takođe svaki dodatni vid komunikacije ujedno predstavlja i potencijalnu metu za sajber napade te je neophodno obratiti pažnju na mere bezbednosti prilikom razvoja servisne magistrale.

Ova arhitektura se smatra prethodnikom mikroservisne arhitekture i sa sobom donosi mnoge benefite koji su takođe prisutni kod mikroservisa. Međutim, glavna razlika ova arhitekture u odnosu na mikroservise jeste ujedno i njena najveća slabost, naime, sva komunikacija se vrši preko servisne magistrale što je čini jedinstvenom tačkom otkaza [17]. Pored toga, kako je svaki servis jedna funkcionalna celina izolovana tako da nema ponavljanja koda, odnosno, samo jedan servis izvršava neku funkcionalnost, tako komunikacija i međusobna zavisnost servisa postaju veoma komplikovane što može učiniti dalje izmene i održavanje veoma napornim [15]. Takođe deljenje iste baze podataka, kao što je i slučaj sa monolitom može predstavljati usko grlo po pitanju performansi [15]. Odvojenost servisa omogućava njihovo nezavisno skaliranje, postignuvši na taj način viši nivo skalabilnosti u odnosnu na monolitne sisteme [18], međutim, servisna magistrala kao centralna komponenta za komunikaciju nije veoma skalabilna te servisno orijentisana arhitektura ne postiže isti nivo skalabilnosti kao i mikroservisna arhitektura. Sa druge strane servisno orijentisana arhitektura ima minimalno ponavljanje koda, što ne važi sa mikroservise, te je i vreme razvoja aplikacije kraće kod ove arhitekture [18].

### Mikroservisna arhitektura

Mikroservisna arhitektura je arhitekturni stil razvoja aplikacija kao kolekcije dva ili više servisa, tako da su servisi obično organizovani oko nekog poslovnog procesa i najčešće je svaki servis privržen samo jednom manjem timu [20]. Mikroservisna arhitektura deli aplikaciju na niz servisa koji se mogu nezavisno deploy-ovati i koji komuniciraju preko API-ja, tako da se svaki servis može skalirati nezavisno [21]. Mikroservisi se najčešće implementiraju praćenjem obrasca ,,baza po servisu“, koji nalaže da svakoj bazi treba da pristupa najviše jedan mikroservis. Važno je naglasiti da ovaj obrazac ne ograničava mikroservise da ne smeju da perzistiraju podatke u većem broju baza podataka, dok god nijedan drugi mikroservise ne pristupa tim istim bazama [22]. Ukoliko su nekom mikroservisu potrebni podaci drugog ili izvršenje neko zahteva dejstvo više od jednog mikroservisa, onda se podaci prikupljaju, odnosno, zahtevi prosleđuju preko API-ja mikroservisa, gde njihova komunikacija može biti sinhrona ili asinhrona. Na slici 2.5 dat je primer aplikacije za onlajn naručivanje izgrađene od mikroservisa.

Prednosti upotrebe mikroservisne arhitekture pre svega leže u brini razvoj većih aplikacija. Veća brzina razvoja je postignuta mogućnostima paralelnog razvoja mikroservisa nezavisno jedni od drugih od strane različitih manjih timova [21]. Dodatno svaki tim može nezavisno da donosi odluke o tehnologijama koje će koristiti za razvijanje mikroservisa na osnovu konkretnog poslovnog problema za koji je on zadužen i na osnovu prethodnih iskustva članova tima. Kako su mikroservisi relativno malih veličina to je njihovo testiranje, debagovanje i izmene po potrebi lako izvesti [23]. Međutim testiranje i debagovanje celokupnog sistema naročito operacija za čije izvršavanje je potrebna komunikacija između većeg broja mikroservisa postaje znatno komplikovanije. Skalabilnost mikroservisnog sistema je jedna od ključnih osobina ove arhitekture, naime, kako su delovi poslovne logike izolovani i nezavisno se mogu deploy-ovati to znači da se precizno, na osnovu broja zahteva, mogu replicirati servisi kako bi se zadovoljile potrebe korisnika [24]. Na taj način nema gubljenja resursa pri skaliranju. Dodatno velika prednost mikroservisa je manjak jedinstvene tačke otkaza, naime, ako usled greške neki od mikroservisa padnu ostali mikroservisi će svoje izvršenje nastaviti nesmetano, sem u situaciji gde im je neophodna komunikacija sa palim mikroservisom. U takvim situacijama treba obezbediti da funkcionalni mikroservis što pre detektuje pad drugog mikroservise i da obavesti korisnika o nemogućnosti ispunjenja zahteva. Detektovanje pada mikroservisa se najčešće vrši korišćenjem unapred zadatog perioda vremena za koji se čeka na odgovor, tako da, ukoliko po isteku tog perioda nije došlo do odgovora onda se servis proglašava nedostupnim. Jedno moguće unapređenje ovog pristupa je implementacija takozvanog obrasca prekidača kola (circuit breaker pattern).



Slika 2.5. Primer mikroservisne arhitekture, slika preuzeta iz [21]

Sa druge strane kod mikroservisa postoji značajno ponavljanje koda, na primer ako različiti mikroservisi koriste isti tip baze podataka za perzistenciju podataka najčešće će svaki od timova nezavisno razviti svoj adapter za komunikaciju sa bazom podataka. Pored toga, sama kompleksnost mikroservisnih sistema je značajno veća od ostalih pristupa, deployment je složen i zahteva specijalizovano znanje, takođe komunikacija između mikroservisa povlačim za sobom neophodnost dobro izgrađenih mreža što, sve u svemu, dovodi do veće cene izvršenja aplikacije.

U poređenju sa prethodno navedenim arhitekturama, mikroservisna arhitektura se smatra za naslednika servisno-orijentisane arhitekture, međutim po svojoj prirodi ona je sličnija modularnom monolitu jer su kod oba ova pristupa izdvojene poslovne logičke celine. S druge strane servisno-orijentisana arhitektura više odgovara slojevitoj arhitekturi, jer su obe ove arhitekture fokusirane na izdvajanju i izolovanju tehničkih funkcionalnosti pre nego poslovne logike. Pored toga treba naglasiti da se slojevita arhitekture i servisno-orijentisana arhitektura generalno smatraju zastarelim. Sa svime prethodno navedenim u vidu, generalni stav za izbor arhitekture prema potrebama projekata je sledeći: za projekte male do srednje veličine sa dobro definisanim zahtevima najbolje je izabrati modularni monolit, ako je pak prioritet jednostavnost i što manji broj operacijskih složenosti treba uzeti u obzir i slojevitu arhitekturu, za velike i kompleksne projekte, koji zahtevaju visok nivo skalabilnosti i fleksibilnosti mikroservisi imaju najviše smisla [10].

## **BEZBEDNA KOMUNIKACIJA**

Komunikacija preko interneta je nezaobilazni deo upotrebe bilo koje web aplikacije. U najmanju ruku neizbežna je komunikacija između korisničkog interfejsa (frontend), preko internet pretraživača, sa serverom na kome se izvršava logika aplikacije (backend). Od velike važnosti je da ta komunikacija bude bezbedna i za korisnike aplikacije i za samu aplikaciju. Naime, korisnici neretko upotrebom neke aplikacije razmenjuju senzitivne informacije sa backend-om. Posebno su kritične aplikacije koje imaju pristup računima korisnika odnosno omogućavaju im da razmene svoj novac za neku uslugu, ali i ostale aplikacije koje barem pamte profil korisnika i zahtevaju upotrebu korisničkog imena i šifre pri prijavljivanju na datu aplikaciju, gde pritom korisnici uglavnom ne koriste različitu šifru za svaku od aplikacija, zahtevaju da komunikacija bude bezbedna.

Pored komunikacije sa korisnicima kod mikroservisne arhitekture prisutna je i komunikacija između mikroservisa. Svi mikroservisi se nalaze sa unutrašnje strane mreže na kojoj su deploy-ovani što doprinosi bezbednosti njihove komunikacije, međutim to nije dovoljno da bi komunikacija zapravo bilo bezbedna te se rešavanju ovog problema pristupa praćenjem takozvanih ,,zero trust“ strategija, koje nalažu da se umesto pokušaja obezbeđivanja mreže, svaka individualna veza između korisnika, uređaja, aplikacija i podataka smatra nesigurnom te da se uvek pri uspostavljanu veze iznova utvrđuje verodostojnost zahteva, odnosno utvrđuje se identitet pošiljaoca i proverava se validnost poruke [25].

Do sada je bilo reči samo o bezbednoj razmeni informacija, međutim bezbednost komunikacije je značajno širi problem od toga. Treba uzeti u obzir da svaki mehanizam za komunikaciju predstavlja određeni vid otvora, odnosno ulaza u inače izolovani sistem, te kao takav predstavlja metu potencijalnog malicioznog napada. Jasno je da je površina za napad znatno veća u distribuiranim sistemima, poput mikroservisa, gde je površina za komunikaciju veća nego kod monolitnih sistema.

Neki od najčešćih napada koji se sreću su:

* Traffic sniffing: prisluškivanje odnosno presretanje paketa preko interneta sa ciljem hvatanja poverljivih informacija. Prilikom slanja paketa preko interneta oni se kreću od jednog čvora mreže do drugog sve dok ne stignu do svog odredišta. Kod ovog napada maliciozni akter se pozicionira na putu paketa do odredišta i pribavlja njihov sadržaj pre nego što ih prosledi dalje [26].
* Spoofing: Napadač se korisniku lažno predstavlja kao neka druga osoba, uređaj ili servis kako bi zadobio korisnikove lične informacije, stekao novac, proširio maliciozan softver ili ukrao podatke [27].
* Man-in-the-middle: Praktično objedinjuje prethodno navedena dva tipova napada, gde se napadač pozicionira između korisnika i nekog servisa i interaguje sa obojicom. Tako da se servisu predstavlja kao korisnik a korisniku kao servis, na taj način pokušavajući da zaobiđe bezbednosne mere tako što će na primer proslediti korisniku zahtev za autentifikaciju koji je on dobio a zatim proslediti servisu korisnikov odgovor kao svoj, pribavivši na taj način privilegije dozvoljene tom korisniku poput upotrebe njegove kartice za plaćanje [28].
* Denial of Service (DoS): Pokušaj napadača da preoptereti servis ili mrežu sa ciljem da oslabi njene performanse ili je učini kompletno neupotrebljivom [29].
* Replay attack: Uključuje hvatanje i ponovno slanje autentifikacionih informacija ili generalno poruka za uspostavljanje pristupa, sa namerom za postizanje nedozvoljenog pristupa [30].

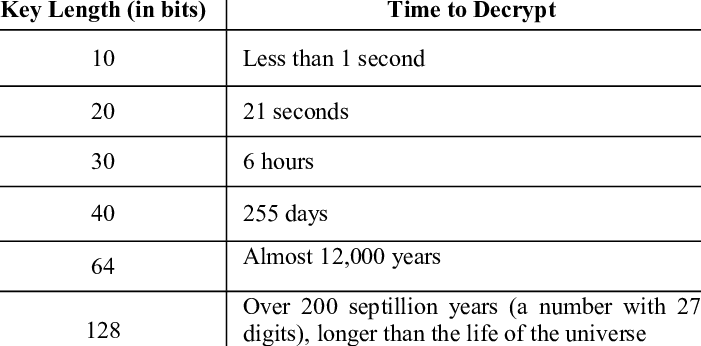
Pored prethodno navedenih napada, napadač može da iskoristi bilo koju slabost odnosno bag u nekim od postojećih protokola, jedan od najpoznatijih primera takvog slučaja je takozvani Heartbleed bag koji je predstavljao ozbiljnu slabost OpenSSL kriptografske biblioteke. Takođe bilo koja greška u projektovanju ili u bezbednosnoj konfiguraciji servisa može biti zloupotrebljena.

Kako bi se postigla bezbedna komunikacija potrebno primeniti model CIA trijade [31]. Ime ovog modela nastalo je od inicijala triju principa kojih se treba pridržavati i to su poverljivost (confidentiality), integritet (integrity) i dostupnost (availability).

Poverljivost se odnosi na nastojanja da se podaci održe tajnim ili privatnim. Da bi se ovo postiglo pristup informacijama mora biti kontrolisan kako bi se sprečilo namerno ili slučajno, nedozvoljeno deljenje podataka. Ključna komponenta održavanja poverljivosti je nastojanje da informacijama mogu da pristupaju samo oni koji su autorizovani da to i učine [31]. Konkretno da bi se to učinilo potrebno je da svaki korisnik ima svoj jedinstveni identitet odnosno profil i kredencijale koji bi trebalo da budu poznati samo njemu i pomoću kojih će, putem prijavljivanja, dokazati svoj identitet. Pored toga svakom od korisničkih profila potrebno je dodeliti određeni skup privilegija, odnosno autorizaciju da izvrši one akcije koje su mu dozvoljene.

Integritet uključuje težnju ka garantovanju da su podaci pouzdani, da im nije pristupano bez ovlašćenja i da nisu neovlašćeno izmenjeni [31]. Kako bi se ovo postiglo koriste se protokoli koji koriste enkripciju podataka poput TLS protokola. TLS protokol takođe garantuje i identitet pošiljaoca u komunikaciji. Postoji i obostrana verzija, takozvani mTLS kod koje obe strane komunikacije garantuju svoj identitet. Više reče o ovom protokolu će biti u narednom poglavlju.

Enkripcija je u suštini proces zaštite podataka korišćenjem matematičkih modela koji transformišu podatke na takav način da je samo pomoću ključa moguće poništiti tu transformaciju i dobiti originalan podatak. Dva najčešća tipova enkripcije su simetrična i asimetrična enkripcija. Kod simetrične enkripcije koristi se isti ključ za enkripciju i dekripciju, dakle za uspešnu komunikaciju neophodno je da i pošiljalac i primalac poruke poseduju isti ključ. Simetrična enkripcija generalno zahteva manju količinu kompjuterskih resursa za enkripciju i dekripciju te je manje kašnjenje prilikom njene upotrebe [32]. Kod asimetrične enkripcije koriste se dva različita ključa za enkripciju i dekripciju podataka. Jedan od ključeva je javni ključ i poznat je svim učesnicima komunikacija, takođe nije neophodno kriti ga od potencijalno malicioznih aktera, ovaj ključ se koristi za enkripciju podataka i bilo ko samim time može da pošalje enkriptovane podatke. Drugi ključ je privatni ključ, služi za dekripciju podataka i dostupan je samo jednom učesniku u komunikaciji [32]. Neki algoritmi za simetričnu enkripciju su Data Encryption Standard (DES), Triple DES, Advanced Encryption Standard (AES). Za asimetričnu enkripciju se najčešće koriste algoritmi Rivest-Shamir-Adelman (RSA) i Elliptic Curve Cryptography (ECC). Važno je imati na umu da u teoriji nijedan od algoritama enkripcije nije bezbedan od napada grube sile koji podrazumeva isprobavanje ključeva za dekripciju podataka sve dok se ne dobije smisleni rezultat. Zato je važno izabrati dovoljno kompleksan ključ kao što je pokazano na slici 3.1.



Slika 3.1. Vreme potrebno za brute force dekripciju na osnovu veličine ključa, slika preuzeta iz [33]

Dostupnost kod CIA trijade se odnosi na činjenicu da i ako su podaci poverljivi i obezbeđen je njihov integritet, ti podaci nisu od velike koriti osim ako su dostupni kada su potrebni. Dakle ostvarivanje dostupnosti se odnosi na sve aktivnosti koje nastoje da prevaziđu napade ili greške koje mogu dovesti do pada aplikacije. Što uključuje razne metode rezervnih kopija i klastera koji se izvršavaju na fizički različitim lokacijama sa ciljem da mogu da preuzmu ulogu ako neka od drugih instanci klastera privremeno ili trajno postane nedostupna.

### TLS

Glavni cilj TLS protokola je da pruži bezbedan kanal između dva učesnika u komunikaciji, sa jedinim uslovom da se na transportnom nivou koristi protokol pouzdanog transporta [34]. Trenutno je aktuelna verzija 1.3 ovog protokola, verzija 1.2 je i dalje u upotrebi i smatra se relativno bezbednom dok se verzija 1.1 smatra zastarelom i više nije u upotrebi. Verzije TLS protokola nisu backwards compatible u smislu da nije moguća komunikacija između različitih verzija ali je format protokola osmišljen tako da je sadržaj poruka isti kod verzija 1.3 i 1.2. TLS je evoluirao od SSL protokola koji je razvijen od strane Netscape-a. TLS verzija 1.0 je zapravo počela svoj razvoj kao SSL verzija 3.1 ali je ime promenjeno pre publikacije kako bi se indiciralo da protokol više nije asociran sa Netscape-om. Međutim i danas se oba termina koriste kada se misli na TLS protokol. Konkretno bezbedni kanal omogućava komunikaciju sa sledećim svojstvima:

* Autentifikacija: serverska strana kanala je uvek autentifikovana, dok je moguća i opciona autentifikacija klijentske strane. Autentifikaciju je moguće izvršiti i pomoću algoritama asimetrične enkripcije i pomoću algoritama simetrične enkripcije [34].
* Poverljivost: Podaci koji su poslati preko kanala nakon njegovog uspostavljanja su samo vidljivi krajnjim učesnicima u komunikaciji. TLS sam po sebi ne skriva dužinu razmenjenih podataka, međutim učesnici u komunikaciji mogu to sami da izvrše kako bi učinili dužinu poruka nepoznatom i poboljšali nivo bezbednosti protiv tehnika analize saobraćaja [34].
* Integritet: izmena podataka poslatih preko kanala nije moguća bez njene detekcije od učesnika u komunikaciji [34].

Dva protokola su glavne komponente TLS-a:

* Protokol rukovanja (Handshake protocol) koji autentifikuje komunicirajuće strane, pregovara o kriptografskim režimima i parametrima i uspostavlja zajednički ključni materijal. Protokol rukovanja je dizajniran da bude otporan na izmene napadača, odnosno aktivni napadač nije u mogućnosti da natera pregovarajuće strane da izaberu drugačije parametre nego što bi to učinile da veza nije pod napadom [34].
* Protokol zapisa (Record protocol) koristi parametre koji su uspostavljeni protokolom rukovanja kako bi obezbedio saobraćaj između strana koje komuniciraju. Ovaj protokol deli saobraćaj na niz zapisa tako da je svaki nezavisno zaštićen korišćenjem ključa za saobraćaj [34].

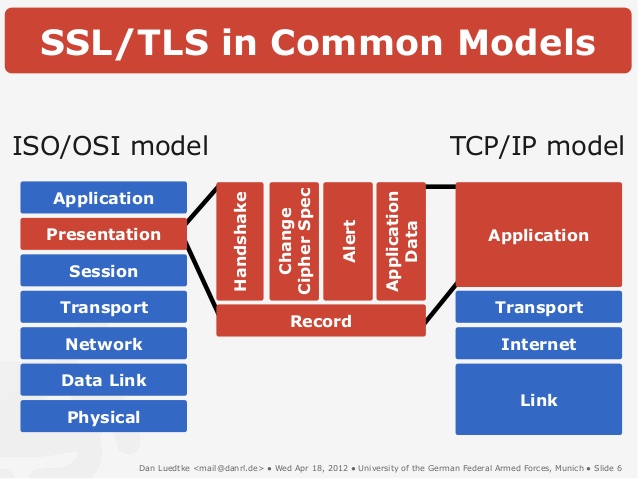
Na OSI referentnom modelu, prikazanom na slici 3.2, TLS se nalazi na nivou sesije, dakle oslanja se na protokole transportnog nivoa i zahteva upotrebu pouzdanih protokola transportnog nivoa. Sa druge strane TLS je nezavisan od protokola koji se koriste na višim nivoima te se može koristi uz bilo koji od protokola aplikativnog nivoa, što uključuje upotrebu HTTP protokola odnosno njegove bezbedne verzije HTTPS bazirane upravo na TLS protokol. Takođe protokoli aplikativnog nivou koji se često koriste pri komunikaciji između mikroservisa poput gRPC protokola i publish/subscribe protokola poput MQTT protokola mogu da koriste TLS kako bi osigurali svoju komunikaciju. Međutim, TLS standard ne specificira kako protokoli višeg nivoa koriste sam protokol TLS-a za postizanje sigurnosti, tj. kako da inicijalizuju rukovanje i kako da interpretiraju razmenjene autentifikacione sertifikate, to je ostavljeno na rasuđivanje onih koji projektuju i implementiraju protokole višeg nivoa [34].

Konekcija koja koristi TLS protokol se inicijalizuje sekvencom poznatom kao TLS rukovanje. Sekvenca rukovanja između klijenta i server podrazumeva ramenu poruka kojom se postiže dogovor vezan za parametre njihove dalje interakcije pomoću TLS-a. Samo rukovanje se sastoji iz tri faze: razmena ključeva, server parametri i autentifikacija [35]. TLS handshake je grafički prikazan na slici 3.3.

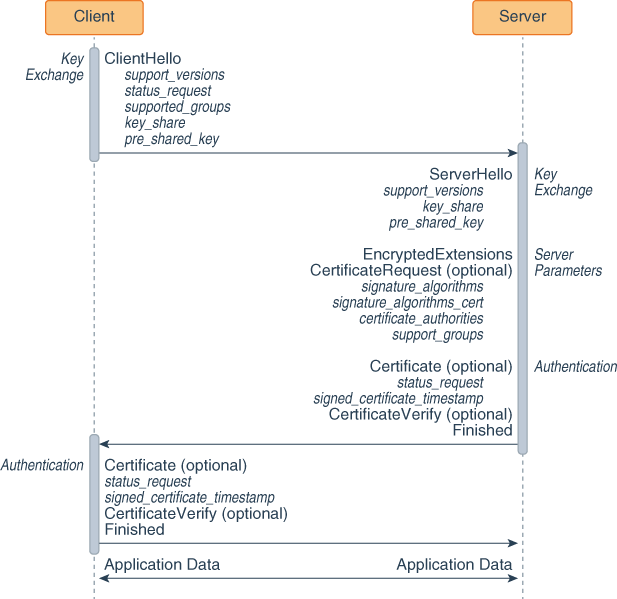
Razmena ključeva započinje tako što klijent šalje ClientHello poruku serveru, klijent je u obavezi da prvo pošalje ovu poruku pri povezivanju sa serverom. Klijent takođe ponovo šalje ovu poruku ako je server odgovorio HelloRetryRequest porukom [34]. ClientHello poruka sadrži četiri seta opcija:

* Listu šifrarskih paketa, odnosno lista kombinacija algoritama za enkripciju i heširanje sadržaja koje klijent podržava.
* Podržane grupe, ekstenzija koja indicira grupe u Diffie-Hellman ili Elliptic Curve Diffie-Hellman algoritmima za razmenu ključeva koje klijent podržava. Grupe se ovde odnose na matematičke osobine algoritma koje klijent podržava. Takođe u okviru ove ekstenzije se šalju podaci neophodni za izračunavanje deljenog ključa.
* Algoritmi za potpisivanje, ekstenzija koja indicira koje algoritme za potpisivanje klijent podržava. To su algoritmi pomoću kojih se utvrđuje identitet servera i potencijalno klijenta.
* Unapred podeljeni ključevi, ekstenzija koja sadrži listu identiteta ključeva poznatih klijentu, ne same vrednosti ključeva već samo njihove identitete.

Ukoliko server izabere da ne koristi unapred podeljene ključeve, onda on generiše svoje setove opcija za listu šifrarskih paketa, podržanih grupa i algoritama za potpisivanje. Ukoliko nema preklapanja između njegovih setova opcija i klijentovih server šalje HandshakeFailure poruku. S druge strane, ako server izabere da koristi unapred podeljene ključeve, on mora u svom odgovoru da naznači koji ključ konkretno je izabran i da se takođe po potrebi usaglasi opcije za razmenu ključeva sa klijentom. U oba slučajeva nakon što su parametri uspešno izabrani server šalje ServerHello poruku kojom obaveštava klijenta o tome da li će koristiti unapred podeljen ključ kao i identifikator tog ključa i koje opcije iz setova opcija je izabrao [34].

Slika 3.2. TLS u OSI i TCP/IP modelu, slika preuzeta iz [36]

Sledeća faza rukovanja je server parametri u kojoj nakon svoje ServerHello poruke server šalje poruke EncrytedExtensions i CertificateRequest, koje sadrže informacije kako će se nastaviti ostatak rukovanja. EncryptedExtensions poruka je obavezna i šalje se odmah nakon ServerHello poruke. Ovo je prva poruka koja je enkriptovana utvrđenim ključevima. Ona sadrži ekstenzije koje nisu neophodne za uspostavljanje kriptografskog konteksta. U suštini predstavlja odgovore na opcije klijenta koje nisu bile neophodne za kreiranje ključeva za enkripciju ali za koje server takođe mora da da svoje vrednosti, poput imena servera, protokola aplikacionog nivoa i podržanih verzija TLS-a. Nakon toga CertificateRequest je opciona poruka i označava da server takođe zahteva sertifikat od klijenta [34].

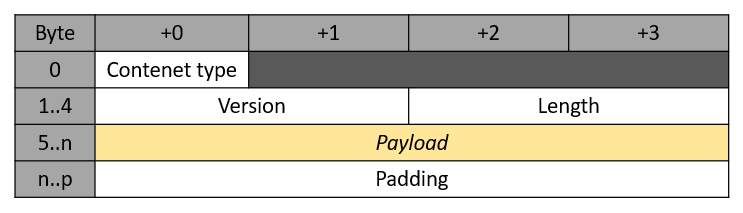


Slika 3.3. Handshake protokol kod TLS-a, slika preuzeta iz [35]

Poslednja faza rukovanja je faza autentifikacije. U okviru ove faze server šalje poruke Certificate, CertificateVeify i Finished. U okviru Certificate poruke šalje se sertifikat koji se koristi za autentifikaciju i po potrebi svi propratni sertifikati u lancu. Autentifikacija pomoću sertifikata nije dostupno ukoliko je izabrana opcija upotrebe unapred podeljenog ključa. CertificateVerify poruka se šalje nakon Certificate poruke i sadrži heš vrednost dobijenu na osnovu sertifikata i ključeva rukovanja. Cilj ove poruke je da dokaže da server stvarno poseduje privatni ključ kojim je sertifikat potpisan, na taj način je pokazano da je sertifikat stvarno poslat od strane servera. Nakon toga server šalje Finished poruku, to je prva poruka koje je kompletno enkriptovana krajnjim ključem i koja zadrži message authentication code (MAC) kreiran na osnovu prethodno poslatih poruka. Svrha MAC-a jeste konačna potvrda da klijent komunicira sa serverom i da su sve prethodno poruke stigle od tog istog servera, služi kao krajnja potvrda rukovanja od strane servera. Takođe kao deo faze autentifikacije klijent odgovara svojom Certificate porukom ukoliko je server to zahtevao CertificateRequest porukom. U tom slučaju klijent takođe šalje svoju CertificateVerify poruku sa istom svrhom kao i kod servera. Zatim, kao poslednju poruku rukovanja klijent šalje svoju Finished poruku kreiranu na isti način i sa istom svrhom kao i kod servera [34].

Record protokol prihvata poruke koje treba poslati, fragmentiše podatke na blokove odgovarajućih veličina (maksimalna veličina 16KB), obavlja zaštitu blokova (enkripcija i autentifikacije) i njihovo slanje. Takođe ovaj protokol je zadužen za verifikaciju primljenih fragmenta, njihovu dekripciju, ponovno sastavljanje i prosleđivanje klijentima višeg nivoa. Dakle record protokol opisuje na koji način se šalju i kako izgleda format poruka te se ovaj protokol koristi i u toku rukovanja, protokol rukovanja opisuje koje sve informacije i kojim redosledom klijent i server treba da dogovore ne i način na koji se te informacije šalju.

Svaki zapis (record) ima naznačen tip: handhsake, application\_data, alert ili change\_cipher\_spec. Kao što je prikazano na slici 3.4, pored tipa svaki zapis takođe sadrži polje za verziju protokola (iz legacy razloga), podatak od dužini poruke i payload, odnosno glavni sadržaj poruke koji zavisi od tipa poruke i po potrebi dopunu do odgovarajuće veličine u zavisnosti od izabranog algoritma za enkripciju. U toku formiranje paketa za slanje prvo se kreira paket koji sadrži nešifriranu verziju sadržaja. Nakon čega se sadržaj transformiše AEAD enkripcijom (Authenticated Encryption with Associated Data), odnosno vrši se enkripcija i autentifikacija u jednom koraku [34]. Na ovaj način sam payload sadrži MAC koji se proverava prilikom dekripcije kako bi identitet pošiljaoca bio potvrđen, što se razlikuje od 1.2 TLS verzije kod koje je postojalo posebno polje u header-u predviđeno za MAC.



Slika 3.4. Format header-a TLS Record protokola

Upotreba sertifikata je opciona mada predstavlja najčešći način za korišćenje TLS-a danas. Sertifikat je fajl koji sadrži sledeće informacije: ime domena za koji se sertifikat izdaje, kojoj osobi, organizaciji ili uređaju je izdat, koji izdavač sertifikata je izdao sertifikat, digitalni potpis izdavala sertifikata, poddomeni asocirani sa datim domenom, datum izdavanja sertifikata, datum isticanja sertifikata, javni ključ onoga kome je sertifikat izdat [37]. Ime domena je važno prilikom izdavanja sertifikata kako bi se potvrdilo da klijent komunicira sa serverom sa kojim želi da komunicira, naime izdavač sertifikata može bilo kome da izda validan sertifikat ali osoba kojoj je sertifikat izdat mora da dokaže da poseduje domen za koji se on izdaje, zato se prilikom provere sertifikata pored validnosti potpisa samog sertifikata ujedno proverava i da li se dobijeno ime domena poklapa sa očekivanim. Izdavač sertifikata je nezavisna treća strana koje izdaje sertifikate i koja mora da prati stroga pravile i standarde kako bi dobila titulu izdavača sertifikata. Izdavač potpisuje sertifikat svojim privatnim ključem, tako da se pomoću javnog ključa može potvrditi da je upravo sertifikat potekao od tog izdavača. Datum isticanja sertifikata se koristi kako bi se proverilo da li je sertifikat i dalje validan. Generalno kod sajber bezbednosti dobro je da svaki entitet vezan za bezbednost ima ograničeno vreme trajanja. Na posletku, javni ključ onoga kome je sertifikat izdat se koristi kako bi se kompletirala razmena ključeva. Treba uzeti u obzir da se poruke već nakon ServerHello poruke enkriptovane ali zajedničkim dogovorenim ključem, rukovanje je tek kompletno kada se razmene javni ključevi koji će se koristiti kasnije u komunikaciji, jer bilo šta što je enkriptovano javnim ključem može biti dekriptovano samo privatnim ključem. Postoje i sertifikati koji su potpisani od strane samog korisnika korišćenjem generisanog para javnog i privatnog ključa, takozvani self-signed sertifiakti, međutim njihova upotreba se ne preporučuje izvan razvoja aplikacije, odnosno u produkciji treba koristiti sertifikate izdate od strane validnog izdavača.

Upotreba TLS-a omogućava bezbednu komunikaciju između mikroservisa nezavisno načina njihove komunikacije. Konkretno između mikroservisa poželjna je upotreba mTLS-a koji podrazumeva obostranu razmenu sertifikata kako bi oba mikroservisa bili sigurni sa kim komuniciraju i kako bi sam sadržaj njihove komunikacije bio sakriven i bezbedan od izmena. Pored toga TLS se koristi i za zaštitu komunikacije između korisnika odnosno forntend-a i backend-a. Međutim nije realno očekivati od klijenta da prilože sertifikat za upotrebu aplikacije. Dakle komunikacije je bezbedna od prisluškivanje i izmena poruke poslatih sa obe strane između klijenta i servera korišćenjem dogovorenog deljenog ključa međutim samo server može da garantuje svoj identitet jer je samo on priložio svoj javni ključ. Očigledno je neophodna da server takođe bude siguran u identitet klijenta te se taj problem rešava drugačijim pristupom, opisanim u sledećem poglavlju.

### Autentifikacija i autorizacija klijenata

Autentifikacija korisnika podrazumeva utvrđivanje njegovog identiteta na osnovu kredencijala koji su jedinstveni za njega, dok autorizacija predstavlja skup prava pristupa korisnika [38].

Autentifikacija se zasniva na razmeni kredencijala korisnika, kredencijali su takođe poznati i kao faktori autentifikacije. Faktor autentifikacije je bilo koji dokaz identiteta korisnika. Faktori autentifikacije mogu biti:

* Faktor znanja: nešto što samo korisnik zna poput šifre, PIN-a ili odgovora na sigurnosna pitanja [38].
* Faktori posedovanja: nešto što samo korisnik poseduje, poput jednokratnih PIN-ova poslatih preko mobilnog telefona ili preko nekog fizičkog medijuma [38].
* Inherentni faktori: biometrijska svojstva korisnika poput lica ili otiska prsta [38].

Kada se korisnik prvi put registruje na neki sistem, on uspostavlja skup autentifikacionih faktora, u slučaju softvera ti faktori se pamte u nekoj bazi podataka. Nakon toga svaki put kada se korisnik prijavljuje on mora da priloži iste te faktore autentifikacije. Sistem proverava ono što je klijent priložio i vrši poređenje sa zapamćenim faktorima. Ukoliko se faktori poklapaju sistem veruje identitetu koji korisnik tvrdi da poseduje. U opštem slučaju moguća je upotreba proizvoljnog broja faktora, međutim u realnosti se najčešće koristi jedan, maksimalno dva faktora prilikom autentifikacije.

Autorizacija se zasniva na korisnikovim premisijama, odnosno politikama sistema o čemu je dopušteno korisniku da pristupa i šta korisnik može da učini sa tim pristupom [38]. Dakle svakom korisniku je pridružen skup premisija i pre svake akcije korisnika vrži se provera da li on sme da izvrši tu akciju. Postoje više tipova autorizacije:

* Autorizacija zasnovana na ulogama (Role-based access control): metod kojim se određuju premisije korisnika na osnovu njihovih uloga, svakom korisniku se dodeljuje jedna ili više uloga [38].
* Autorizacija zasnovana na atributima (Attribute-based access control): metod koji koristi atribute korisnika, objekata i akcija, poput korisnikovog imena, tipa resursa i vremena u toku dana za odlučivanja nivoa pristupa [38].
* Obavezan nivo pristupa: sistemi sprovode centralno definisanu politiku nivoa pristupa za sve korisnika [38].
* Diskreciona kontrola pristupa: sistem omogućava vlasnicima resursa da postave sopstveni nivo kontrole pristupa nad svojim resursima [38].

U praksi danas najčešće se vrši autorizacija pomoću JSON Web Tokena (JWT). JWT je otvoren standard koji definiše kompaktan i samo-zatvoren način za bezbednu transmisiju informacija između različitih strana, pomoću JSON objekata. Ove informacije mogu biti verifikovane i može im se verovati zato što su digitalno potpisane. JWT može da se potpiše korišćenjem tajne ili parom javni/privatni ključ. Dakle potpisani token služi da verifikuje integritet tvrdnji koje se nalaze u njemu, pored toga token može dodatno biti i enkriptovan kako bi sakrio tvrdnje koje se nalaze u njemu [39].

Prilikom registracije korisnik će izabrati svoje jedinstveno korisničko ime i šifru. Da bi korisnik mogao kasnije da se prijavi potrebno je zapamtiti njegove kredencijale. Međutim šifru nije preporučljivo pamtiti direktno u bazu već treba pamtiti heš šifre. Razlog za to je što ukoliko se pamti šifra direktno, bilo ko, ko ima pristup bazi bi mogao da sazna šifru svakog korisnika. Nije reč samo o nepoverenju prema administratorima baze podataka, već u slučaju da dođe do propusta prilikom podešavanja baze podataka, neko maliciozno lice potencijalno može da stekne informacije o sadržaju baze podataka. Heš je rezultat jednosmerne matematičke funkcije koja transformiše ulaz, u ovom slučaju šifru, u niz karaktera fiksne dužine. Funkcija je takva da za isti ulaz uvek daje isti izlaz, ne može se na osnovu heša odrediti originalna vrednost, male razlike u ulazu kod šifra dovode do velikih razlika heša i funkcija se brzo izračunava. Uz sve to, šifri se pridodaje nasumičan niz karaktera koji se naziva salt, sa ciljem prevencije napada kao što su napad dugine tabele. Rezultujući heš se dakle pamti u bazi podataka [40].

Zatim, prilikom prijavljivanja korisnik prilaže svoju šifru uz svoje korisničko ime, pribavlja se vrednost heša i vrednost korišćenog salt-a i primenom funkcije za heširanje se proverava da li je unesena šifra ispravna [40]. U bazi podataka se pored šifre korisnika takođe pamte i njegove premisije i to najčešće u obliku uloga. Ako je šifra koju je korisnik uneo tačno potvrđeno je da je prijava poslata od strane upravo tog korisnika. Na taj način je potvrđen njegov identitet. Međutim potrebno je potvrditi njegov identitet i prilikom svakog narednog zahteva. Nije produktivno zahtevati od korisnika da pošalje svoju šifru svaki put kada želi da obavi neku akciju. Umesto toga upravo se koriste JWT-ovi.

Nakon što je unesena ispravna šifra server generiše parove JWT-ova. Jedan token za pristup i jedan token za obnavljanje. Token za pristup u sebi sadrži osnovne informacije o korisniku poput njegovog jedinstvenog identifikatora i njegove uloge na osnovu koje se određuju premisije za izvršenje akcija kao i timestamp kada token za pristup ističe. Token za obnavljanje sadrži samo timestamp kada ističe i pamti se u bazi podataka za svakog korisnika. Oba tokena se šalju korisniku tako da se od njega očekuje da uz svaki naredni zahtev pošalje token za pristup kao garanciju svog identiteta, jer potpisani JWT je otporan na izmene. Pored toga često ovaj token sadrži informacije za autorizaciju poput uloge korisnika pa se ujedno prilikom svakog zahteva korisnika proverava da li mu je dozvoljeno da izvrši zahtevanu akciju. Kako se token za pristup često koristi, od velikog značaja je da je njegovo vreme trajanja veoma kratko, preporuka je da bude u rasponu od 5 do 15 minuta. S druge strane token za obnavljanje se korisniku šalje putem kolačića, tako da mu nije moguće pristupati iz skripti koje se izvršavaju u internet pretraživaču. Ovaj token klijent šalje serveru nakon što mu istekne token za pristup. Ukoliko se token za obnavljanje poklapa sa onim koji je sačuvan u bazi podataka, korisniku će biti izdat novi par tokena za pristup i tokena za obnavljanje. Ovakav način korišćenja tokena za pristup i obnavljanje smatra se bezbednim jer su tokeni za pristup kojima je moguće pristupiti iz skriptnog koda, odnosne koje je moguće ukrasti različitim cross-scripting napadima, kratkog trajanja, tako da je i u slučaju njihovom pristupa od strane malicioznih aktera značajno ograničena njihova mogućnost da taj token iskoriste. Sa druge strane tokenima za obnavljanje pristup nije moguć te oni imaju značajno duži period života čak i do dve nedelje [41].

Jasno je po nameni JWT-a da je potrebno validirati njihov potpis prilikom svakog zahteva korisnika. Ovo je nešto komplikovanije kod mikroservisnih arhitektura zato što putem API-ja klijent može da pošalje zahtev bilo kojem od mikroservisa. U ovakvim slučajevima postoje dva moguća pristupa, centralna autorizacija i distribuirana autorizacija [42].

Centralna autorizacija podrazumeva postojanje jednog mikroservisa koji je zadužen za validaciju tokena poslatih od strane korisnika. Što znači da sve svaki korisnički zahtev šalje ovom mikroservisu nakon čega on proverava validnost tokena i poziva odgovarajuću metodu traženog mikroservisa prosledivši mu pri tome uloge klijenta kako bi taj mikroservis mogao da utvrdi da li je klijentu dozvoljeno da izvrši traženu akciju. Alternativno originalni zahtev se šalje mikroservisu koji treba da ga izvrši a sam taj mikroservis komunicira sa mikroservisom za autorizaciju, odnosno šalje mu dobijeni token na validaciju. Prednost centralizovane autorizacije je to što je potrebno logiku autorizacije, odnosno validacije tokena implementirati samo jednom. Takođe dovoljno je parametre za validaciju tokena čuvati na samo jednom mestu. Nedostatak ovakvog pristupa je što je potrebno implementirati komunikaciju između svih mikroservisa i centralnog mikroservisa za autorizaciju i to što onda mikroservis za autorizaciju predstavlja usko grlo jer praktično svaki zahtev korisnika prolazi kroz ovaj mikroservis.

Distribuirana autorizacija znači da svaki mikroservis zasebno vrši proveru validnosti tokena i određuje da li je akcija dozvoljena na osnovu uloge zapisane u tokenu. Ovakav pristup znači da je neophodno posebni implementirati proveru validnosti za svaki mikroservis pojedinačno, što je značajna količina ponovljenog pisanja koda. Pored toga svaki mikroservis nezavisno definiše parametre validacije pa je potrebno sinhronizovati ih, naročito ako dođe do izmene parametara. Sa druge strane ovaj pristup pruža manju latentnost prilikom obrade korisničkih zahteva jer nema dodatne komunikacije sa još jednim mikroservisom. Takođe ne postoji usko grlo što znači bolja skalabilnost sistema.

## **POUZDANA KOMUNIKACIJA KOD MIKROSERVISA**

Mikroservisna arhitektura kao što je diskutovano u drugom poglavlju donosi prednosti pri razvoju velikih aplikacija ali takođe donosi određeni nivo dodatne tehničke kompleksnosti. Ta kompleksnost pre svega proističe iz potrebe za komunikacijom između mikroservisa. Dakle u slučaju korisničkih zahteva koji se potpuno izvršavaju u okviru jednog mikroservisa nema potrebe za dodatnim implementacijama i u tom slučaju sam mikroservis je praktično jedan manji izolovani monolit, što jeste najbolji scenario za upotrebu monolita. Situacija je komplikovanije kada je neophodna komunikacija između dva mikroservisa. Naime mikroservisi moraju da komuniciraju u dva slučaja:

* Kada klijentski zahtev utiče na više od jednog aspekta poslovne logike po kojoj su mikroservisi podeljeni te je izvršenje zahteva podeljeno na akcije koje svaki od mikroservisa treba da izvrši. Dakle kada je za izvršenje zahteva korisnika potrebna izmena perzistiranog stanja većeg broje entiteta različitih tipova. Na primer ako korisnik želi da rezerviše sobu u hotelu potrebno je sa njegovog računa naplatiti cenu sobe i naznačiti da ta soba više nije slobodna. Ovakav tip zahteva, koji se sastoji iz više delova se naziva transakcija. Te je kod mikroservisa potrebno obezbediti postojanje distribuiranih transakcija.
* Kada su za izvršenje klijentskog zahteva potrebne informacije koje su dostupne samo drugom mikroservisu. Odnosno kada jedan mikroservis traži informacije od drugog bez da dolazi do izmene stanja drugog mikroservisa.

Situacija je ovde složenija zato što je potrebno omogućiti komunikaciju između mikroservisa tako da oni mogu da traže informacije jedni od drugih i da takođe prosleđuju zahteve za izvršenje akcija jedni drugima. Pri tome treba uzeti u obzir da bilo koji od mikroservisa u jednom trenutku može otkazati i samim tim dovesti do neuspešnog izvršenja akcije, a takav prekid izolovanih mikroservisa je mnogo teže detektovati nego izuzetak koji bi se javio u slučaju monolitne implementacije. Pored toga monolit omogućava pisanje transakcija tako da je, ukoliko jedan korak ne uspe, moguće izvršiti operaciju ROLLBACK koja poništava sve prethodne korake. Ako svi koraci pak uspeju vrši se operacija COMMIT koje potvrđuje izmene. Na taj način se postiže da je sistem uvek u konzistentnom stanju. Generalno monolitne aplikacije omogućavaju korišćenje ACID transakcija. ACID označava skup osobina transakcija:

* Atomičnost (Atomicity): svaka transakcija se tretira kao jedinstvena operacija i ili će se potpuno izvršiti ili se uopšte neće izvršiti [43].
* Konzistentnost (Consistency): znači da sistem uvek prelazi iz jednog konzistentnog stanja u drugo, tj. nikada se ne može naći u nekonzistentnom stanju [43].
* Izolacija (Isolation): transakcije koje izvršavaju konkurentno ne vide međusobne izmene dok ne budu kompletne. Na najvišem nivou izolacije, efekat konkurentnog izvršenja transakcija i njihovog sekvencijalnog izvršenja je isti [43].
* Izdržljivost (Durability): nakon uspešnog izvršenja transakcije, odnosno nakon COMMIT operacije, efekti transakcije ostaju šta god da se desi [43].

Kod distribuiranih sistema na osnovu CAP teoreme sledi da je veoma teško garantovati ACID svojstva. Umesto toga ovakvi sistemi se češće opredeljuju za BASE model transakcije:

* U osnovi dostupno (Basically available): baza je uglavnom dostupno [43].
* Mekano stanje (Soft state): može doći do izmene stanja vremenom a ne samo kao direktna posledica akcije korisnika [43].
* Eventualno konzistentno (Eventually consistent): sistem će u nekom trenutku postići konzistentno stanje ali nema garancija o tome kada će se to desiti i različiti korisnici mogu da dobiju različite informacije o trenutnom stanju [43].

Pored tipa transakcija koje sistem obezbeđuje, važna karakteristika sistema takođe je i upotreba sinhrone ili asinhrone komunikacije [44]. Sinhrona komunikacija je blokirajući tip komunikacije što znači da se pošiljalac zahteva blokira i čeka na odgovor [44]. Sinhrona komunikacija podrazumeva ostvarivanje direktne veze između komunicirajućih mikroservisa, što je najsličnije pozivu metode kod monolita. Međutim ovakav pristup znači viši nivo spregnutosti sistema jer pozivajući mikroservis mora da poznaje dostupne metode mikroservisa koga poziva. Sinhrona komunikacija se najčešće vrši putem poziva REST API-ja ili preko nekog pristupa poziva udaljene procedure (remote procedure call), kao što je gRPC. Glavna prednost sinhronog tipa komunikacije je što direktna komunikacija daje povratnu informaciju o tome da li je zahtev uspešno izvršeno ali s druge strane veća spregnutost sistema dovodi do manje skalabilnosti jer mikroservisi moraju da čekaju jedan drugog pri komunikaciji. Alternativno, postoji i asinhrona komunikacija preko nekog posrednika koji se naziva broker, ovaj tip komunikacije je poznat i kao pošalji-i-zaboravi komunikacija [44]. Zasniva se na publish subscribe principima komunikacije gde pošiljalac objavljuje poruku na neku temu preko brokera a svi zainteresovani servisi se prijavljuju da slušaju na tu temu i kada poruka bude objavljena broker obaveštava sve zainteresovane servise kod kojih se poziva callback funkcija za obradu pristigle poruke. Na ovaj načini mikroservisi su potpuno izolovani i mogu da izvršavaju instrukcije svojim tempom što omogućava veći nivo skalabilnosti sistema. Međutim kod ovog tipa komunikacija ne postoji direktna povratna informacija o uspešnosti izvršenja zahteva. Takođe kako bi se obezbedio ovaj tip komunikacije neophodno je postojanje brokera poruka. Broker je server koji se sposoban da prima i prosleđuje poruke na osnovu zadatih kriterijuma. Neki od brokera koji se koriste danas su NATS, Kafka, RabbitMQ…

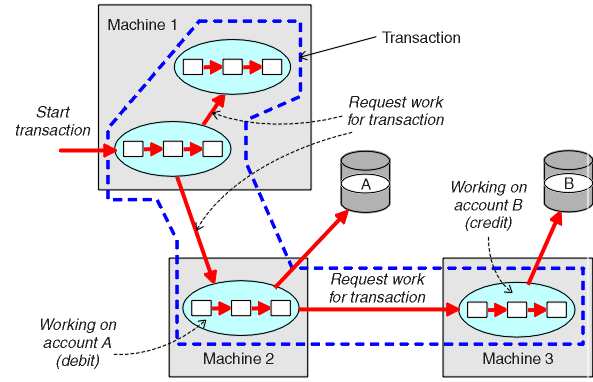
Postoje različiti projektni obrasci koji služe za obezbeđivanje pouzdane komunikacije i mogućnost upotrebe transakcija kod mikroservisnih sistema u zavisnosti tipa transakcije koja je potrebna i tipa komunikacije koja se koristi. Obrasci koji omogućuje ostvarivanje distribuiranih transakcija su: Dvofazna potvrda (2 Phase commit), Saga i Probaj-Potvrdi-Poništi (Try-Confirm/Cancel). Uz ove obrasce, ako se za razmenu poruka koristi asinhroni tip komunikacije, potrebno je dodatno implementirati obrasce koji garantuju da je poruka dostavljena poput: Outbox i Inbox obrazaca, obrasca Odvajanja upita od dužnosti (Command Query Responsibility Segregation) i obrasca Izvora događaja (Event Sourcing). Poslednji obrazac je karakterističan po tome što ujedno predstavlja i tip arhitekture sistema.

### Obrasci za upravljanje distribuiranim transakcijama

Distribuirana transakcija je transakcija koje se izvršava kroz više procesa, obično na različitim uređajima. Svaki proces radi za transakciju, kao što je prikazano na slici 4.1, na kojoj je elipsom označen posao koji se obavlja ne jednom uređaju dok strelice između njih označavaju pozive udaljenih procedura [45]. Za izvršenje distribuiranih transakcija neophodno je obezbediti postojanje dveju karakteristika:

* Povratak stanja procesa na prethodno stanje. Ovo se može postići ili definisanjem parova operacija tako da jedna vrši neku akciju a druga je poništava. Potrebno je da stanje sistema pre izvršenja ovih akcija i nakon izvršenja obe akcije bude isto. Ili je potrebno da sistem beleži svoje prethodno stanja tako da je moguć povratak na to stanje nezavisno od izvršene izmene [45].
* Protokol za potvrdu. Odnosno protokol koji će voditi računa o tome koji delovi sistem treba da izvrše neku akciju i da li je ta akcija uspešno izvršena [45].

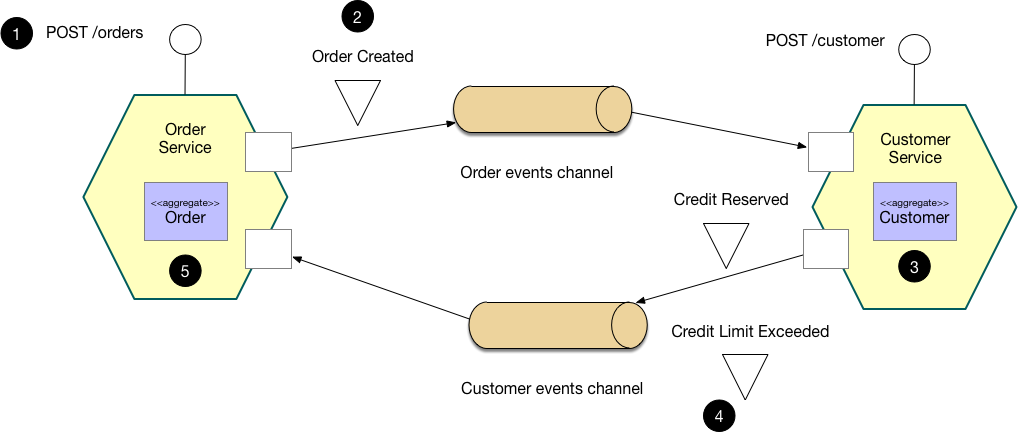
Dvofazna potvrda je projektni obrazac koji omogućava izvršenje transakcija najbližih ACID transakcijama, ali značajno negativno utiče na performanse. Kod ovog obrasca jedan od procesa koji učestvuju u transakciji ima ulogu koordinatora. Koordinator nadgleda aktivnosti drugih učesnika u transakciji kako bi obezbedio konzistentan ishod transakcije. Kao što se može zaključiti iz imena sam obrazac se sastoji iz 2 faze. Prva faza je faza pripreme, u kojoj koordinator šalje poruke svim procesima koji učestvuju u transakciji, sa ciljem da ih obavesti da izvrše akcije i pripreme se za potvrdu. Prilikom pripreme svaki proces mora da garantuje da može da potvrdi transakciju i na taj način je trajno zabeleži. Nakon što je poslao takvu potvrdu proces više nema pravo da nezavisno odluči o prekidu izvršenja svoje akcije. Ukoliko proces ne može da garantuje izvršenje akcije, zbog pada baze ili zbog poslovnih ograničenja, npr. nedostatak novca na računu, on mora da signalizira prekid izvršenja. Druga i poslednja faza ovog obrasca je faza odluke. Koordinator proces je dobio odgovore svih procesa koji učestvuju u transakciji i treba da donese odluku na osnovu njih i da informiše ostale procese o svojoj odluci. Ako su svi procesi signalizirali da mogu da potvrde akciju onda koordinator donosi odluku da treba potvrditi transakciju i šalje poruku svim ostalim procesima da izvrše operaciju COMMIT. Ako je pak, barem jedan proces obavestio koordinatora da ne može da potvrdi izvršenje onda koordinator javlja svim učesnicima u transakciji da ponište svoje izmene. U svakom slučaju ili će svi procesi izvršiti izmene ili nijedan neće [45].



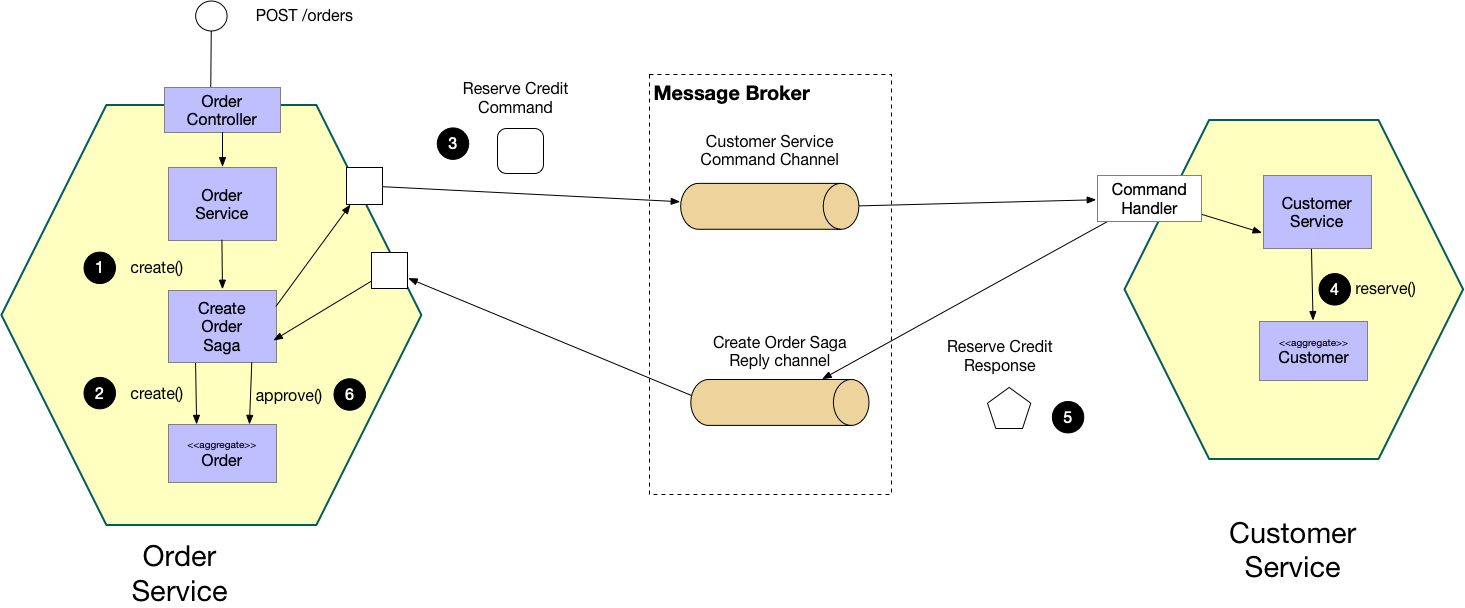
Slika 4.1. Primer distribuirane transakcije, slika preuzeta iz [45]

Najveća slabost ovog obrasca je što zahteva upotrebu zaključavanja resursa. Odnosno, komunikacija sa koordinatorom je blokirajuća, tj. svi mikroservisi koji učestvuju u izvršenju transakcije se blokiraju, za sve operacije, sem za komunikaciju sa koordinatorom, dok se transakcija ne završi, što značajno šteti performansama i smanjuje skalabilnost sistema. Takođe u opštem slučaju postoji scenario u kojem ovaj obrazac dovodi sistem do nekonzistentnog stanja. Ako proces pošalje poruku koordinatoru da je spreman da potvrdi i padne u međuvremenu pre nego što na nalog koordinatora izvrši COMMIT operaciju onda će svi ostali procesi izvršiti svoj deo operacija sem procesa koji je pao. Te sistem neće biti u konzistentnom stanju. Kako bi se ovaj problem prevazišao potrebno je implementirati logovanje izvršenih zahteva tako da proces nakon ponovnog podizanja može nadomesti izgubljene operacije.

Za razliku od Dvofazne potvrde obrazac Sage ne nastoji ka atomičnom izvršenju transakcija i dovodi sistem u stanje eventualne konzistentnosti, ali je, zbog loših performansi Dvofazne potvrde, češći izbor kod mikroservisne arhitekture. Obrazac Sage funkcioniše kao niz lokalnih transakcija. Svaka lokalna transakcija ažurira svoju bazu i šalje poruku ili objavljuje događaj preko brokera, kako bi izazvala izvršenje sledeće lokalne transakcije Sage. Ako neka lokalna transakcija padne, bilo zbog tehničkih grešaka bilo zbog poslovnih ograničenja, Saga izvršava niz kompenzujućih transakcija koje poništavaju originalne promene prethodnih lokalnih transakcija [46]. Postoje dva načina koordinacije kod Saga obrasca: koreografija i orkestracija. Primer Sage bazirane na koreografiji je dan na slici 4.2 dok je primer Sage bazirane na orkestraciji dat na slici 4.3.



Slika 4.2. Primer Sage bazirane na koreografiji, slika preuzeta iz [46]

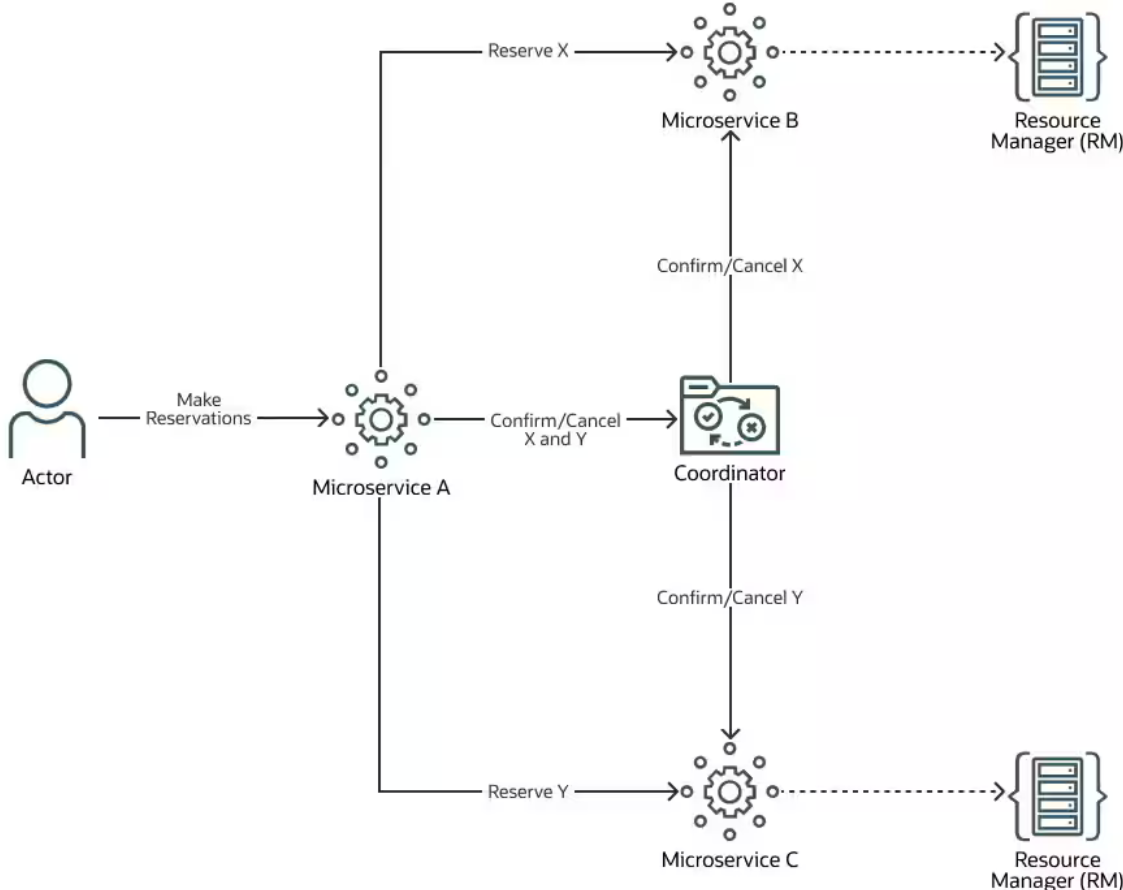


Slika 4.3. Primer Sage bazirane na orkestraciji, slika preuzeta iz [46]

Saga bazirana na koreografiji nema centralno upravljanje već svaki mikroservis koji završi svoju lokalnu transakciju treba da izazove izvršenje sledeće lokalne transakcije. Najlakše je to učiniti objavljivanjem poruke koja započinje tu lokalnu transakcije preko brokera. Na ovaj način servis ne mora da zna kako tačno da pozove metodu koja započinje izvršenje lokalne transakcije kod drugog mikroservisa. Ako u nekom trenutku neka lokalna transakcije ne uspe onda se u suprotnom smeru propagiraju poruke o neuspehu i izvršavaju lokalne transakcije koje kompenzuju originalnu izmenu [46].

Saga bazirana na orkestraciji ima centralni orkestrator koji jedini može da započne izvršenje neke lokalne transakcije. Dakle u ovom slučaju svaki mikroservis o svom uspehu odnosno neuspehu obaveštava orkestatora koji zatim po potrebi započinje izvršenje sledeće lokalne transakcije ili poziva kompenzujuće funkcije [46].

Nedostaci ovog projektnog obrasca su nepostojanje automatskog poništavanja transakcija, odnosno ne postoji globalna ROLLBACK operacija već je za svaku transakciju neophodno implementirati kompenzujuću transakciju, nedostatak izolacije transakcija, naime kako se koraci transakcije izvršavaju sekvencijalno, neki drugi korisnički zahtev se može izvršiti pre kraja transakcije što znači da taj zahtev može vratiti nekonzistentnu informaciju korisniku. Na primer u aplikaciji za rezervisanje soba hotela korisnik može da dobije informaciju da je soba slobodna i ako je u toku izvršenje transakcije koja upravo rezerviše tu sobu. Takođe važno je i obezbediti da se poruka za početak nove lokalne transakcije sa sigurnošću pošalje ali to se postiže korišćenjem drugih projektnih obrazaca i nije deo same Sage.



Slika 4.4. Ilustracija obrasca Try-Confirm/Cancel, slika preuzeta iz [47]

Kao kompromis između dva prethodno navedena obrasca koristi se obrazac Probaj-Potvrdi/Poništi (TCC). Ovaj obrazac predstavlja transakcioni protokol u okviru koga se resursi drže u rezervisanom stanju dok se transakcija ne potvrdi ili ne poništi [47]. Na slici 4.4 data je grafička ilustracija ovog obrasca. Veoma je sličan Dvofaznoj potvrdi jer oba obrasca imaju centralni koordinator za transakcije sa glavnom razlikom što nema blokiranja servisa dok transakcija traji već se resursi koji su deo transakcije u bazi označavaju posebnim stanjem rezervisanosti što sprečava ostale klijente da kao kod Sage dobiju netačne informacije prilikom čitanja podataka. Obrazac se sastoji iz dve faze:

* Faza Pokušaj: Najčešće onaj mikroservis koji započinje transakciju je ujedno i koordinator transakcije. U ovoj fazi on šalje zahteve svim mikroservisima koji učestvuju u transakciji da probaju da rezervišu resurse. Pri tome svaki od mikroservisa treba da vidi da li je u stanju da izvrši svoj deo transakcije na osnovu ograničenja poslovne logike. Koji god da je ishod te provere o tome se obaveštava koordinator.
* Potvrdi/Poništi faza: Na osnovu odgovara od strane mikroservisa koji učestvuju u transakciji koordinator odlučuje da li treba izvršiti ili poništiti transakciju. Isto kao kod Dvofazne potvrde ukoliko je barem jedan od mikroservisa odgovorio da nije moguće izvršiti transakciju ona se poništava. Ukoliko se donese odluka o potvrdi transakcije rezervisani resursi bivaju trajno iskorišćeni i operacija se potvrđuje u bazi podataka. Dok ukoliko se transakcija poništava rezervisani resursi se vraćaju ponovo u dostupne.

Na ovaj način postiže se atomičnost transakcije bez blokiranja. Međutim sistem nije uvek u konzistentnom stanju, pogotovo između operacija pokušaj i poništi. Problemi ovog obrasca uključuju kompleksnost implementacije, potrebno je posebno implementirati svaku od Pokušaj, Potvrdi i Poništi operacija. Takođe ovaj obrazac zahteva prilagođavanje šeme baze podataka da bi resursi mogli da budu stavljeni u rezervisano stanje. Dalje neophodno je obezbediti da su operacije pokušaj idempotentne kako ne bi slučajno došlo do dvostrukog rezervisanja resursa od istog zahteva. Naposletku, takođe je neophodno obezbediti logiku za rešavanje situacija kada poruke Potvrdi ili Poništi koordinatora nisu dostavljeno zbog tehničkih problema poput pada mreže.

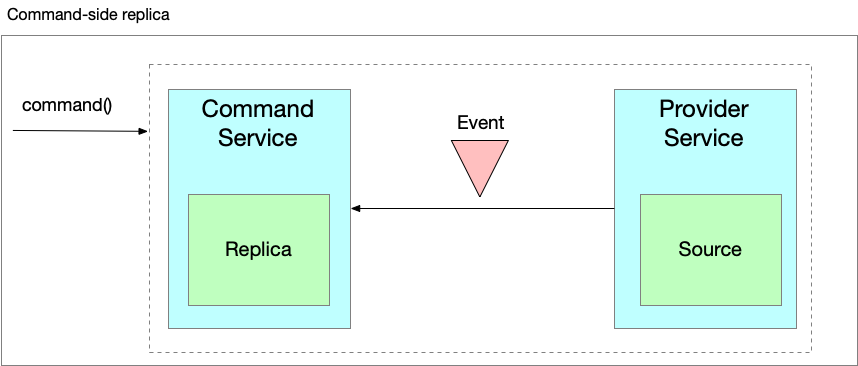
### Obrasci za pouzdano slanje poruka

Problem sa asinhronom komunikacijom je to što ne postoji povratna informacija da je poruka isporučena, što je veliki problem kod obrazaca transakcije. Naime, potrebno je garantovati da će se, nakon što je deo transakcije izvršen na jednom mikroservisu, poruka za izvršenje drugog dela transakcije uspešno poslati, čak i u slučaju pada brokera za komunikaciju. Alternativno rešenje je da se lokalna izmena baze i slanje poruke učine atomičnim kako bi se postiglo da u slučaju nemogućnosti slanja poruke lokalne izmene baze takođe budu poništene.

Outbox obrazac je projektni obrazac koji obezbeđuje garanciju da će poruka eventualno uspešno biti poslata. Funkcioniše tako što se ne šalje poruka direktno na broker nego se upiše u posebnu tabelu u bazi mikroservisa koja se naziva Outbox tabela i iz koje pozadinski proces čita zapise i prosleđuje ih na broker [48]. Pozadinski proces nastavlja povremeno da šalje zapis sve dok on ne bude označen kao kompletiran. Mikroservis će označiti zapis kao kompletiran nakon što dobije odgovor od mikroservisa kome šalje poruku. S obzirom da se zapis nalazi u lokalnoj bazi mikroservisa njegov upis u bazu se može izvršiti u okviru lokalne transakcije mikroservisa, nakon čega će poruka nastaviti da se šalje sve dok ne bude uspešno poslata.

Značajan problem koji pri tome treba rešiti je mogućnost da ista poruka bude više puta poslata mikroservisu koji možda nije imao prilike da još uvek obradi prvu poruku pa zato nije obavestio originalni mikroservis da je poruka obrađena. Postoje dva pristupa za rešavanje ovog problema, prvo moguće je učiniti efekte svih poruke idempotentnim tako da i ako je poruka stigla više puta efekat je isti kao da je stigla samo jednom. Drugi pristup je implementacija druge polovine ovog obrasca poznate kao Inbox obrazac. Kod ovog pristupa svakoj poruci se zadaje jedinstven identifikator. A primaoci poruka implementiraju mehanizam za pamćenje identifikatora primljenih poruka te ako ista poruke bude poslata više puta može se jednostavno odbaciti nakon prvog slanja. Sa druge strane, kada mikroservis koji je primio poruku šalje odgovor ne treba implementirati ovaj obrazac zato što nije potrebno garantovati da je odgovor stigao. Ako odgovor ne stigne samo će originalna poruka biti ponovo poslata i zbog idempotentnosti ili zabeleženog identifikatora ujedno će i odgovor samo ponovo biti poslat. Takođe poželjno je implementirati određeni vid vremenskog ograničenja koji će zaustaviti beskonačno slanje poruke i zabeležiti da je došlo do greške prilikom njenog slanja.

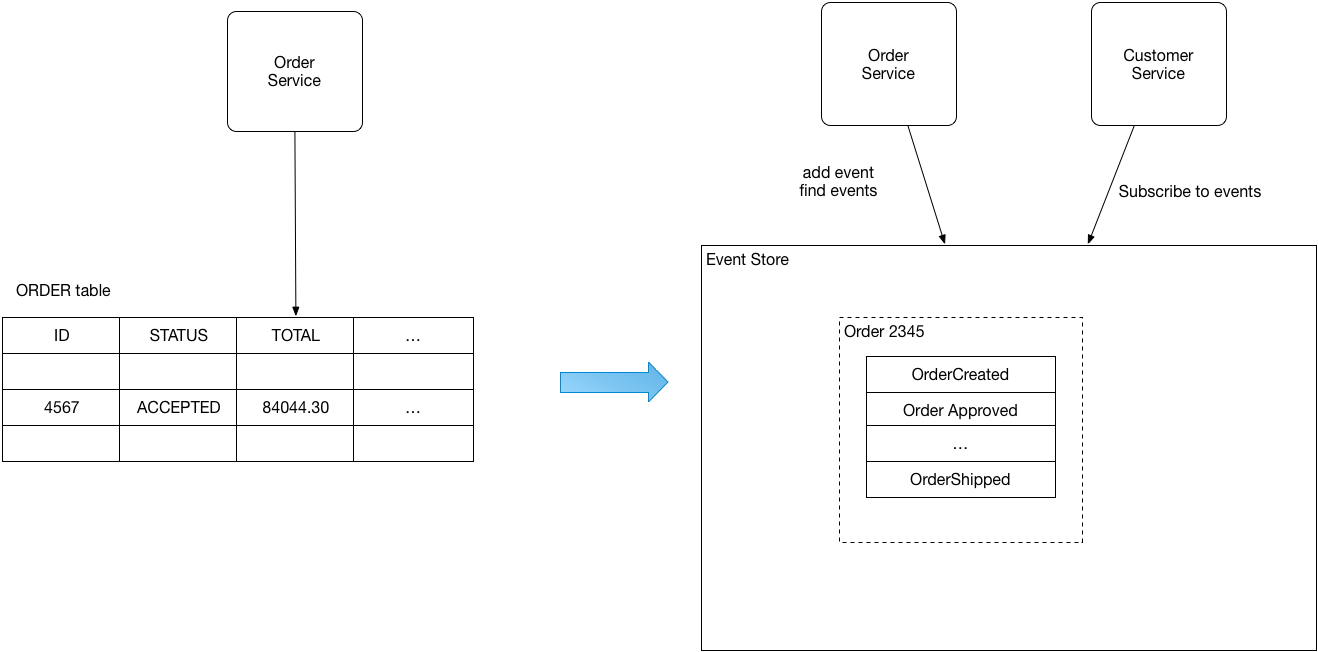
Obrazac Odvajanje upita od dužnosti (CQRS) takođe poznat i kao Replika sa strane komande (Command-side replica). Ovaj projektni obrazac definiše rešenje za problem prikupljanja informacija od drugih mikroservisa. Naime, ako neki korisnički zahtev upućen jednom mikroservisu zahteva i informacije od drugog mikroservisa neblokirajuća komunikacije nije pogodna jer prvi mikroservis ne može da nastavi sa radom dok ne dobije potrebne podatka. Sa druge strane blokirajuća sinhrona komunikacija znači spregnutost mikrservisa. CQRS rešava ovaj problem tako što nalaže kreiranje read-only replike baze podataka mikroservisa od kojeg se zahtevaju podaci [49]. Na taj način originalni mikroservis može iz jedne transakcije da pročita neophodne podatke. Pri tome se ova baza podataka održava konzistentnom sa glavnom bazom tako što drugi mikroservis emituje poruke pri svojim izmenama baze dok prvi mikroservis hvata ta izmene i ažurira svoju bazu. Na slici 4.5 nalazi se grafički prikaz ovog obrasca.



Slika 4.5. Grafička ilustracija CQRS projektnog obrasca, slika preuzeta iz [49]

Mana ovakvog pristupa je ponavljanje podataka, deljenje šeme baze između timova i neophodnost implementacije logike za održavanje replike baze konzistentnom. Uz to ova baza je eventualno konzistentna odnosno može se desiti da je originalna baza izmenjena ali da replika još uvek nije ažurirana.

Obrazac izvora događaja (Event sourcing) je još jedan od pristupa koji rešava problem atomičnosti izmena. Event sourcing je pristup projektovanja sistema tako da se umeto trenutnog stanja entiteta pamti niz izmena koje su izvršene nad tim entitetom, taj niz izmena se pamti u bazi stanja, dakle izmena trenutnog stanja se vrši tako što se u bazu stanja dodaje najnovija izmena[50]. Kako je upis u bazu atomičan to znači da je samim tim i cela izmena stanja atomična. Takođe preporučljivo je koristiti bazu koja ujedno funkcioniše kao broker poruka, npr. Kafka. Zatim, kada god je potrebno dobiti trenutno stanje nekog entiteta na osnovu zapisanih događaja izračunava se novo stanje. Kako bi se poboljšala efikasnost ovog pristupa povremeno se izračunava i pamti trenutno stanje entiteta da bio potreban manji broj operacija za izračunavanje trenutnog stanja [50]. Na slici 4.6 dat je primer prelaza pamćenja stanja entiteta u pamćenje niza koraka izmene stanja. Dodatni benefit ovakvog pristupa je pamćenje istorije stanja entiteta i mogućnosti povratka na proizvoljno stanja. S druge strane ovo je karakteristični i uglavnom neintuitivni pristup te zahteva određeno vreme prilagođenja. Takođe upiti prema bazi stanja su komplikovani i mogu biti neefikasni. Na kraju treba pomenuti još i da se ovaj projektni obrazac veoma dobro uklapa sa CQRS projektnim obrascem gde se izmene baze stanje emituju i izvršavaju nad read-only replikama.



Slika 4.6. Primer Event sourcing-a, slika preuzeta iz [50]

## **IMPLEMENTACIJA**

U nastavku slede relevantni delovi implementiranog sistema od 3 mikroservisa demo aplikacije tipa društvene mreže. Prvi mikroservis je napisan u .NET-u i obavlja funkcionalnosti vezane za autentifikaciju i autorizaciju. Podatke perzistira u PostgreSQL bazi podataka. Drugi mikroservis je implementiran u NestJS frejmvorku, služi za keširanje i podatke pamti u Redis bazi podataka. Treći mikroservis je implementiran u Gin frejmvorku za Golang programski jezik, služi za upravljanje profilima korisnika i podatke perzistira u Neo4j bazi.

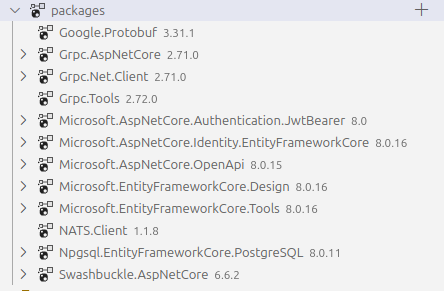
Kako ovakva aplikacije ne zahteva stroge transakcije Dvofazne potvrde i nema odgovarajuću strukturu za primenu Probaj-Potvrdi/Poništi obrasca, izabrana je Saga kao obrazac za implementaciju distribuiranih transakcija. Konkretno operacije koje zahtevaju primenu Sage su registracija kod koje se kreiraju osnovni podaci korisnika i pamte u PosgreSQL bazi a pored toga se u Redis bazi od strane drugog mikroservisa pamti kod za verifikaciju E-mail-a. Pored toga uspešna potvrda E-mail-a pored izmena u PostgreSQL bazi takođe treba da izazove kreiranje korisničkog profila od strane Gin mikroservisa.

Autorizacije se vrši JWT-ovima tako da je iz demonstrativnih razloga implementirana i centralizovana i distribuirana autorizacija. Distribuirana autorizacija je implementirana sinhronom komunikacijom i to konkretnom preko gRPC-ja. Asinhrona komunikacija se vrši posredstvom NATS brokera. Svi REST API zahtevaju upotrebu HTTPS-a odnosno koriste TLS protokol. Komunikacija preko NATS-a i gRPC-a takođe koristi TLS protokol. Pored toga implementiran je Outbox obrazac sa idempotentnim zahtevima.

### Auth mikroservis

Implementiran u .NET frejmvorku, kao servis za autentifikaciju i autorizaciju. Koristi AspNetCore Identity uz EntityFramework za automatsko kreiranje šema tabela u PostgreSQL bazi podataka. U nastavku ovog poglavlja biće dati isečci kod neophodni za rad ovog mikroservisa, što uključuje konkretna podešavanja frejmvorka ali i implementacije poput pozadinskog Outbox servisa. Transakcije koje se prostiru kroz veći broj mikroservisa će biti izdvojene u posebnom poglavlju.

Na slici 5.1.1 prikazani su paketi koji se koriste u mikroservisu. Redom sa slike to su Google.Protobuf koji omogućava rad sa Protobuf fajlovima, koji se koriste u gRPC komunikaciji. Za gRPC komunikaciju takođe je potrebno dodati Grpc.AspNetCore, Grpc.Net.Client i Grpc.Tools pakete. Od Microsoft.AspNetCore pakete koriste se Authentication.JwtBearer koji omogućava JWT autentifikaciju korišćenjem šeme koja mu se prosleđuje i koji se dodaje kao opcija prilikom uključivanja middleware-a za autentifikaciju. Identity.EntityFrameworkCore se koristi za automatsko kreiranje šema baza podataka za autentifikaciju, funkcioniše u sprezi za EntityFramework-ok. Paket OpenApi je standardni paket za .NET web aplikacije i vezan je za standard definisanja REST API-ja. Nakon toga slede EntityFramworkCore paketi, sam EntityFramework je Microsoft-ov frejmvork za rad sa relacionim bazama podataka. NATS.Client je paket za komunikaciju za NATS serverom. Paket Npgsql.EntityFrameworkCore.PostgreSQL je konkretan driver EntityFramework-a za PostgreSQL bazu podataka. Na kraju se nalazi još jedan paket koji je uvek deo Web aplikacije, Swashbuckle.AspNetCore koji služi za automatsko generisanje OpenAPI dokumentacije.



Slika 5.1.1. Paketi .NET mikroservisa

Na slici 5.1.2 je prikazano dodavanje middleware-a za autorizaciju i autentifikaciju u fajlu Program.cs, gde se servis za autentifikaciju podešava da radi sa JWT-ovima. Nakon toga sledi dodavanje IdentityCore servisa, tu je neophodno naznačiti POCO klasu koja sadrži neophodne property-je za registraciju i prijavljivanje. Podrazumevano se koristi Identity klasa UserIdentity. Zatim je ovde konkretno naznačeno da će se koristiti uloge i ostvarena je veza sa EntityFramework-om. Na kraju se vidi i samo podešavanje EntityFramework-a da koristi PostgreSQL driver. Dalje na slici 5.1.3 navedeni su svi servisi koji se koriste u aplikaciji tako da su izdvojeni prvo servisi koji su ugrađeni u .NET zatim su preostali servisi uređeni na osnovu scope-a i na kraju je pozadinski servis.



Slika 5.1.2. Middleware za autentifikaciju i autorizaciju.



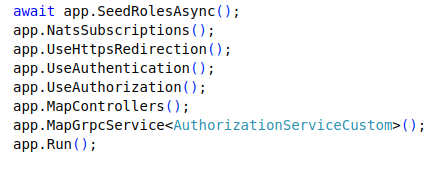
Slika 5.1.3. Upotrebljeni servisi.

Na slici 5.1.4 prikazano je podešavanje Kestrel .NET web servera. Konkretno podešeno je da aplikacija sluša na dva port-a. Ideja je da se port 7147 koristi za komunikaciju preko gRPC-a sa drugim mikroservisima a port 5253 za pritup REST API-ju. Međutim Kestrel ne dozvoljava filtriranje saobraćaja po portu te tehnički oba tipa komunikacije su dozvoljena na oba port-a. Uprkos tome, kako je port 7147 podešen da koristi mTLS pri komunikaciji to znači da klijenti neće moći da preko njega pristupe REST API-ju jer ne poseduju potrebni sertifikat. Slično port 5253 šalje podrazumevani .NET sertifikat te preko njega nije moguća komunikacija preko gRPC-ja jer to nije sertifikat koji će ostali servisi očekivati da dobiju. U produkciji podrazumevani .NET sertifikat treba zameniti sa sertifikatom koji je potpisan od strane nekog Certificate Authority-ja. Sertifikat koji se koristi za gRPC komunikaciju je generisan i potpisan lokalno, odnosno je takozvani self-signed sertifikat što je u ovom slučaju prihvatljivo jer se koristi samo za internu komunikaciju međutim svakako je uvek bolje koristiti sertifikat potpisan od nekog Certificate Authority-ja.

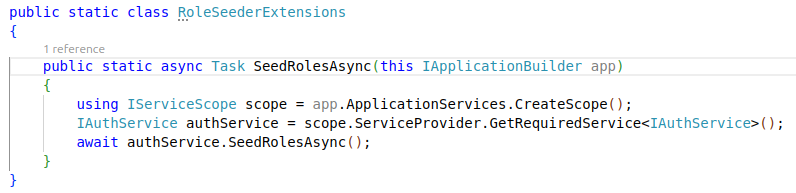


Slika 5.1.4. Podešavanje Kestrel-a.

Na slici 5.1.5 su navedene sve metode koje se pozivaju prilikom pokretanja aplikacije. SeedRoleAsync služi da u bazu unese postojeće uloge pri prvom pokretanju aplikacije i to je specijalno napisana extension metoda. Slično NatsSubscriptions postavlja callback metode za unapred definisane teme pri komunikaciji sa NATS-om. UseHttpsRedirection je ugrađena metoda u .NET koja preusmerava klijentske HTTP zahteve da koriste HTTPS. UseAuthentication i UseAuthorization uključuju upotrebu odgovarajućih middleware-a na svakom od pristiglih zahteva. MapControllers mapira kontrolere da se izvršavaju prilikom slanja zahteva na odgovarajuću rutu. MapGrpcService mapira pristigle gRPC zahteve na odgovarajuće server metode. I na kraju Run pokreće aplikaciju. Nakon toga slikama 5.1.6 i 5.1.7 pokazana je konkretna implementacija SeedRolesAsync i NatsSubscriptions metoda.



Slika 5.1.5. Startup metode.

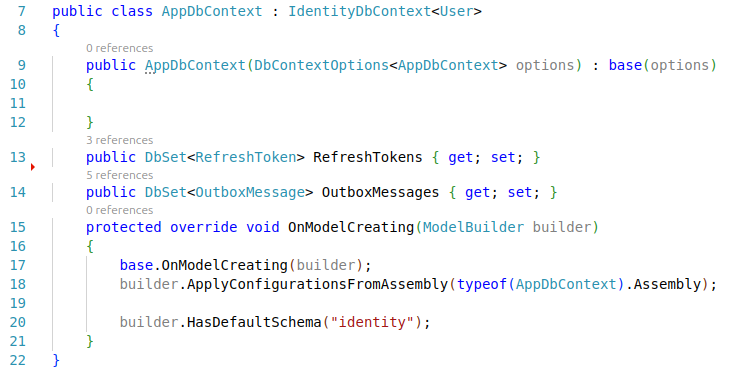


Slika 5.1.6. Implementacija SeedRoleAsync metode.

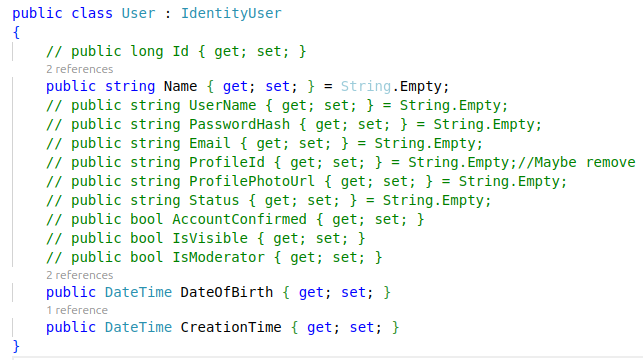


Slika 5.1.7. Implementacija NatsSubscriptions metode.

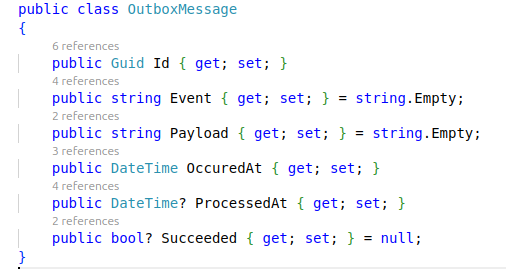
Dalje na slici 5.1.8 prikazana je implementacija klase AppDbContext koje se koristi za podešavanje EntityFramework-a i njegovog Code-First pritupa pri kreiranju baza podataka. Nakon čega slede slike 5.1.9, 5.1.10 i 5.1.11 koje prikazuju važnije entiteta, redom to su User, OutboxMessage i RefreshToken.



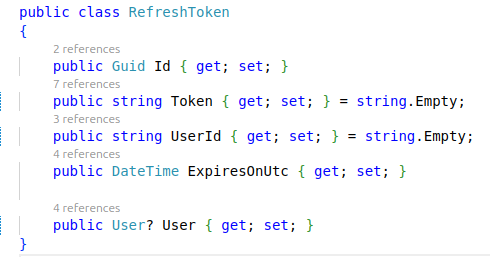
Slika 5.1.8. EntityFramework context.



Slika 5.1.9. User klasa



Slika 5.1.10. OutboxMessage klasa.

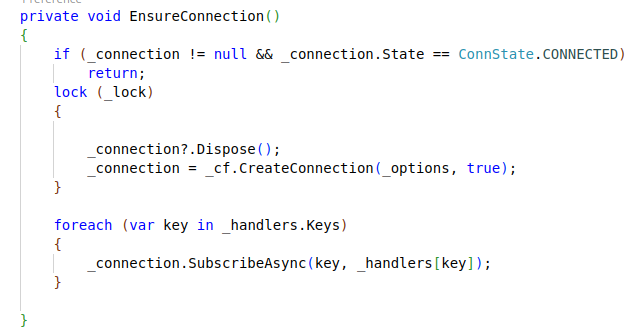


Slika 5.1.11. RefreshToken klasa.

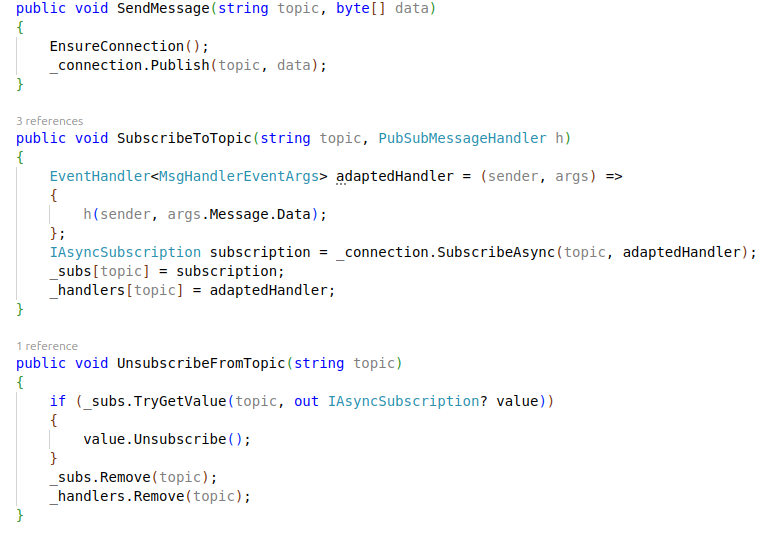
Na slici 5.1.12 prikazano je podešavanje veze sa NATS brokerom korišćenjem TLS-a. Broker šalje sertifikat a sa druge strane servis garantuje svoj identitet autentifikacijom preko korisničkog imena i šifre. Ovaj pristup takođe omogućava preko da se mikroservisu dopusti da objavljuje i sluša samo na određenim temama, ta podešavanja se vrše preko NATS konfiguracionih fajlova. Zatim, slika 5.1.13 sadrži implementaciju metode koja se koristi da prilikom pada NATS brokera ponovo uspostavi vezu i postavi subskripcije koje se ranije postojale. Nakon toga na slici 5.1.14 prikazane su metode za objavljivanje poruka i subskripciju i uklanjanje subskripcije na neku temu.



Slika 5.1.12. NATS konekcija.



Slika 5.1.13. Implementacija metode za ponovno uspostavljanje konekcije.

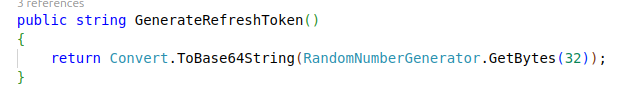


Slika 5.1.14. Implementacija metoda za slanje poruka, subskripciju na temu i uklanjanje subsrkipcije.

Na slikama 5.1.15 i 5.1.16 prikazane su implementacije metoda za kreiranje Access i Refresh tokena:

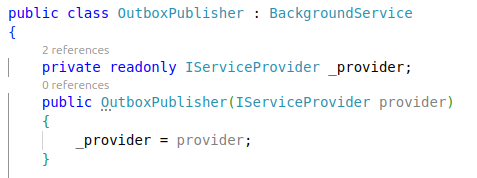


Slika 5.1.15. Kreiranje Access tokena.



Slika 5.1.16. Kreiranje Refresh tokena.

Dalje, na slici 5.1.17 je prikazana definicija klase i konstruktora pozadinskog Outbox servisa. Bitan implementacioni detalj je da klasa mora da implementira BackgroundService interfejs. Dalje na slici 5.1.18 je prikazana glavna metode Outbox servisa koja sa razmakom od 5 sekundi uzima zadnjih 10 poruka iz Outbox tabele i objavljuje ih preko brokera za poruke. Na kraju na slici 5.1.19 data je i implementacija metoda koja izaziva trenutno slanje Outbox poruke sa zadatim identifikatorom.



Slika 5.1.17. OutboxPublisher klasa.



Slika 5.1.18. Metoda za povremeno slanje Outbox poruka.



Slika 5.1.19. Metoda za slanje Outbox poruke sa zadatim identifikatorom.

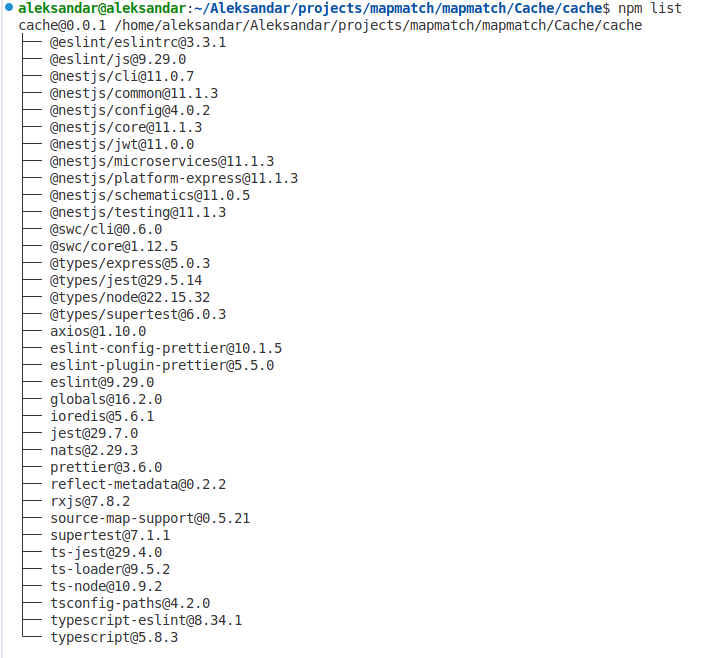
### Keš mikroservis

Keš mikroservis je implementiran i NestJS frejmvorku. Pamti podatke u Redis bazi podataka. Redis je in-memory baza podataka što omogućava izuzetno brzi pristup podacima i što čini ovu bazu pogodnom za upotrebu kao keš. Pored toga Redis podržava pamćenje vremenski ograničenih slogova tako da se prilikom isteka zadatog perioda izvrši specificirana callback funkcija, ova funkcionalnost čini Redis veoma zgodnim za pamćenje vremenski ograničenih kodova poput koda za verifikaciju mail-a, što je upravo način na koji je ovde u ovoj demonstrativnoj aplikaciji i iskorišćen.

Na slici 5.2.1 je data lista korisnički instaliranih npm paketa ovog mikroservisa. Većina paketa sa liste se automatski instaliraju kreiranjem NestJS aplikacije. Dodatno instalirani paketi su:

* @nestjs/config: koristi se za kreiranje modula za upravljanje konfiguracijom aplikacije.
* @nestjs/jwt: paket za rad sa JWT-ovima.
* @nestjs/microservices: paket koji obezbeđuje funkcionalnosti specifične za mikroservisa.
* axios: paket za slanje http poruka.
* ioredis: driver za Redis bazu podataka.
* nats: driver za komunikaciju sa NATS brokerom.

Na slici 5.2.2 je dat sadržaj main.ts fajla u kome se definiše konekcija sa NATS brokerom. Mikroservisne NestJS aplikacije su koncipirane na takav način da se prvo definišu sve konekcije i API koje će taj mikroservis koristiti. Ovde je konkretno podešena sertifikat koji se očekuje od brokera i korisničko ime i šifra kojom ovaj mikroservis dokazuje svoj identitet. NATS broker takođe omogućava i autentifikaciju putem mTLS-a i putem tokena pa je ovde radi različitosti primera izabrana autentifikacija korisničkim imenom i šifrom. Nakon toga se metodom startAllMicroservices() kreiraju objekti za komunikacije definisane iznad i pokreće se NestJS lifecycle pozivom funkcije init(). Važna napomena ovde je da se ukoliko mikroservis pruža komunikaciju preko REST API-ja potrebno je pozvati metodu listen() kojoj se prosleđuje broj porta preko koga se pristupa API-ju.

Slika 5.2.1. Korisnički instalirani npm paketi.



Slika 5.2.2. Sadržaj main.ts fajla.

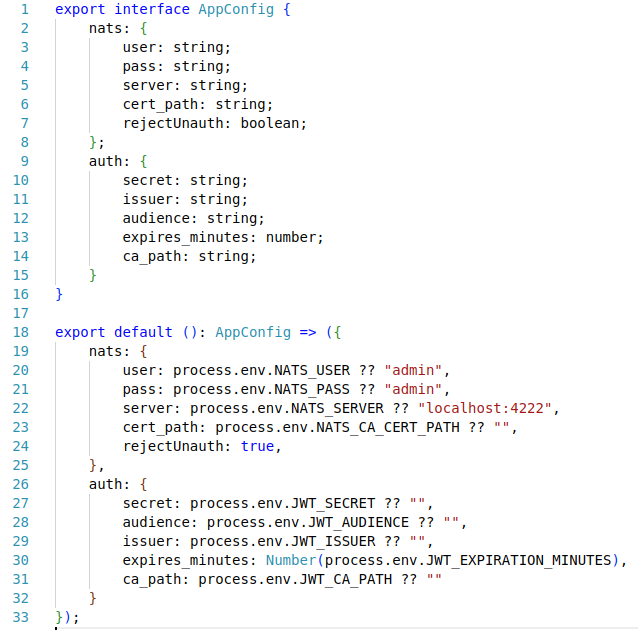
Na slici 5.2.3. data je implementacija fajla cache.module.ts. NestJS forsira modularnu arhitekturu aplikacije pa se za svaki moduo nezavisno definišu zavisnosti i klase koje se mogu injektovati koje se ovde nazivaju Providers. Takođe jedna specifičnost NestJS je što je neophodno ponoviti kod iz main.ts fajla kod definisanje modul fajla koji komunicira, u ovom slučaju, sa NATS brokerom.



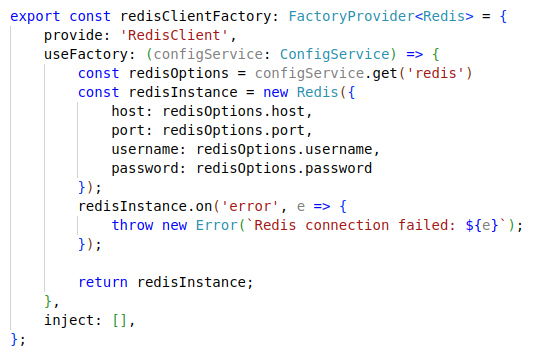
Slika 5.2.3. Cache module.

Na slici 5.2.4 je data implementacija konfiguracionog modula. Zatim na slici 5.2.5 je prikazana definicija factory-ja za konekcije sa Redis bazom podataka. Factory u NestJS je jedan od tipova Provider-a pomoću koji se mogu injektovati i pomoću kojih se kasnije mogu dinamički kreirati potrebni objekti. U ovom slučaju kreiraju se konekcije sa Redis bazom. Potrebne su barem dve konekcije, jedna za unos slogova u bazu i jedna koja će sve vreme osluškivati da li je istekao ključ za čije isticanje je definisana callback funkcija. Na slici 5.2.6 je data implementacija repozitorijum klase za Redis bazu. Na slici 5.2.7 je dat kod gde se konekcija sa Redis bazom koristi za slušanje emitovanih događaja isteka ključeva i izvršava se određena funkcija na osnovu vrednosti ključa.

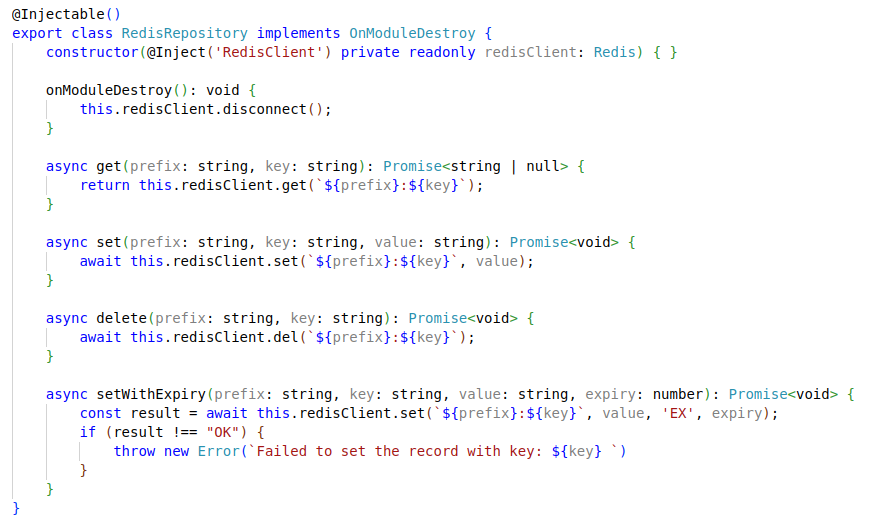
Dalje na slici 5.2.8 je data implementacija metoda koje objavljuje poruke na broker. Ovaj mikroservis pored preko brokera komunicira sa trećim servisom preko REST API-ja trećeg servisa te je zato, kao što je prikazano na slici 5.2.9, implementirana metoda za slanje http zahteva. U ovom slučaju konkretno https zahteva. NestJS mikroservis verifikuje identitet mikroservisa kome šalje zahtev tako što od njega očekuje sertifikat. Pored toga ovde je demonstrirana autentifikacija tokenom gde samo mikroservisi imaju pristup ključu tokena. Međutim ovaj pristup susceptibilan man-in-the-middle napadu jer token nije dovoljan za garanciju identiteta. Da bi se ovo donekle nadomestilo postavljeno je trajanje tokena na svega jedan minut.



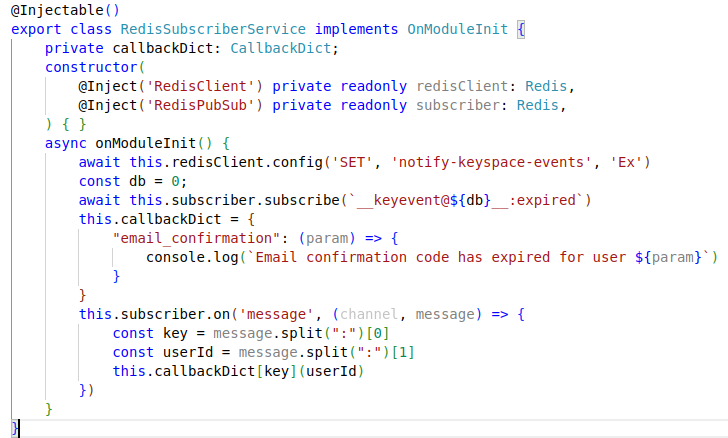
Slika 5.2.4. Implementacija konfiguracionog modula.



Slika 5.2.5. Definicija factory-ja za Redis konekcije.



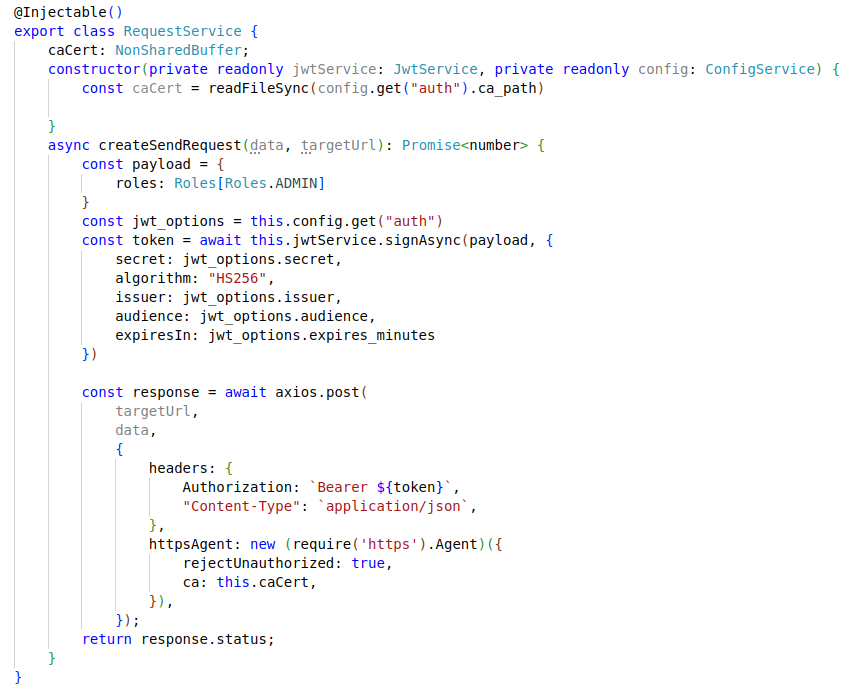
Slika 5.2.6. Implementacija repozitorijum klase za Redis bazu.



Slika 5.2.7. Redis callback za istek ključa.



Slika 5.2.8. Slanje poruke NATS brokeru.

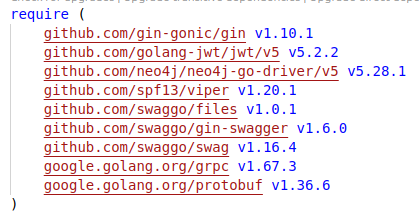


Slika 5.2.9 Metoda za slanje https poruka.

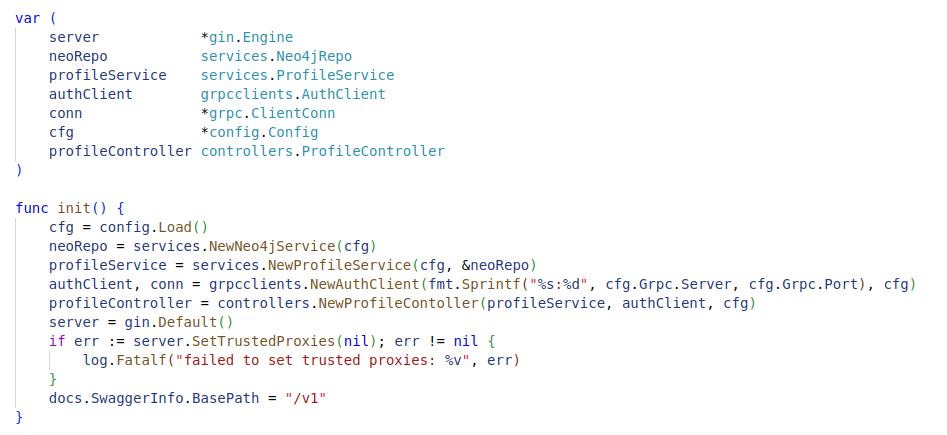
### Profil mikroservis

Profil mikroservis je implementiran u Gin frejmvorku za Golang programski jezik. Ovaj mikroservis je zadužen za obavljanje funkcionalnosti upravljanja profilima korisnika. Pored toga on je zadužen za beleženje prijateljstva između korisnika. Sa drugim servisima komunicira preko gRPC-ja i tako što sluša zahteve preko svog REST API-ja. Podatke pamti u Neo4j bazi podataka koja je graf baza podataka, što omogućava efikasan pristup profilima prijatelja kada se profil korisnika predstavi kao čvor a odnos prijateljstava kao veza između čvorova.

Na slici 5.3.1 prikazan je niz paketa koji su korišćeni prilikom implementacija ovog mikroservisa. Prvo paket je gin-gonic koji dodaje frejmvork za web aplikacije. Sledeći paket služi za rad sa JWT-ovima. Zatim potreban je i driver za komunikaciju za Neo4j bazom podataka. Viper se koristi za parsiranje konfiguracionog fajla. Files služi za rad sa fajlovima i ovde se konkretno koristi sa učitavanje sertifikata. Gin-swagger se koristi za pokretanje Swagger UI-a za lakše testiranje prilikom razvoja, slično Swag se koristi zajedno sa Gin-swagger-om i služi konkretno za generisanje OpenAPI dokumenta koji se koristi u Swagger-UI-u. Poslednja dva paketa omogućavaju gRPC komunikaciju i rad sa protofajlovima.



Slika 5.3.1. Paketi iskorišćeni u Profil mikroservisu

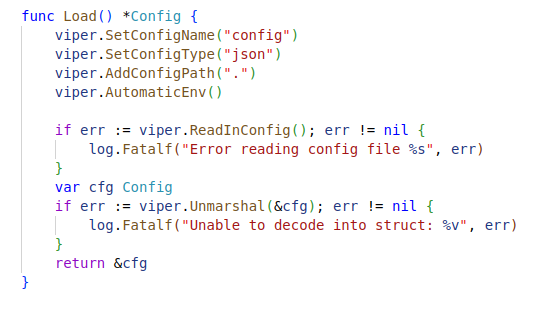


Slika 5.3.2. Funkcija init iz main.go fajla.

Golang je jezik sa velikim akcentom na jednostavnošću što mu omogućava postizanje visokih preformansi. Međutim to takođe znači da je često ručno potrebno obaviti neke stvari koje frejmvorkovi obično sami obavljaju. Na primer .NET i NestJS sami obavljaju dependency injection specificiranih klasa pri kreiranju klasa koje od njih zavise. Međutim kod Gin-a je potrebno ručno instanciranje korišćenih objekata i njihovo slanje funkcijama koja inicijalizuju od njih zavisne objekte klasa. Na slici 5.3.2 je ovaj proces prikazan u main.go fajlu. Dalje u istom fajlu izvršava se i funkcija koja pokreće neophodne konekcija ka bazama i konekcije za komunikaciju. U okviru ove funkcije se definišu različite rute REST API-ja i specificiraju se middlware-i koji se koriste za svaku od ruta. Kao što se može primetiti iznad funkcije sa ove slika i iznad metoda kontrolera sa slike 5.3.10 Swag koristi komentare kako bi generisao OpenAPI dokumentaciju.

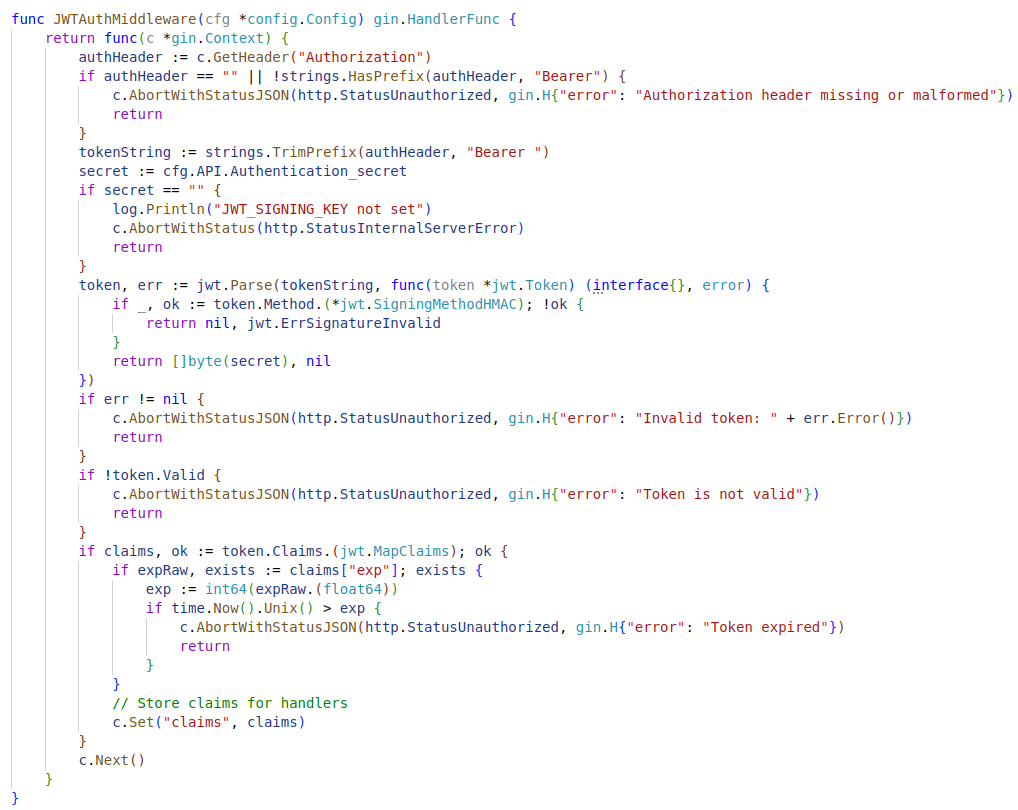


Slika 5.3.3. Funkcija main.



Slika 5.3.4. Učitavanje konfiguracionog fajla.

Na slici 5.3.4. dat je primer upotrebe Viper paketa za parsiranje konfiguracionih fajlova.



Slika 5.3.5. Middleware za lokalnu autorizaciju.



Slika 5.3.6. Middleware za preuzimanje identifikatora iz JWT-a.



Slika 5.3.7. Middleware za slanje zahteva za autorizaciju centralnom servisu za autorizaciju.



Slika 5.3.8. Middlware za proveru uloge korisnika.

Na slikama od 5.3.5-5.3.8 dati su primeri middlware-a. Na slici 5.3.5. je implemntacija middlware-a koji vrši proveru validnosti pristiglog JWT tokena, uz to da se zahtev automatski smatra nevalidnim ako ne sadrži token. Dalje na slici 5.3.6 se nalazi middlware koji izdvaja vrednost identifikatora iz Claims sekcije tokena. Slika 5.3.7 prikazuje udaljenu validaciju tokena, iz demonstrativnih razloga u ovom mikroservisu implementirana je i lokalna i udaljena validacija tokena. Na posletku 5.3.8 je middleware koji proverava da li korisnik poseduje potrebnu ulogu za izvršenje tražene metode.

Slika 5.3.9. prikazuje metodu koja služi za kreiranje objekta driver-a za Neo4j bazu podataka. Naime, golang ne sadrži klase te samim tim ne postoje ni konstruktori ali postoje struct-ovi kojima se mogu pridodati metode te se pomoću njih postiže ponašanje ekvivalentno postojanju klasa. Konvencija je da se ove metode koje igraju ulogu konstruktora nazivaju imenom struct-a sa ,,New“ dodatim kao prefiksom. Zatim su na slici 5.3.11 dati primeri metoda koje upravljaju Neo4j bazom. To su metode koje se pridodaju struct-u koji se instancira na slici 5.3.9.



Slika 5.3.9. Metoda za instanciranje veza za Neo4j bazom podataka.



Slika 5.3.10.



Slika 5.3.11. Metode za komunikaciju sa Neo4j bazom podataka.

Naposletku, na slici 5.3.10 je dat primer metode kontrolera. Kao što je već nagovešteno komentari se koriste za opis metode. Takođe još jedna specifičnost golang-a koje je bila primetna kroz veći deo prikazanih slika je činjenica da izuzeci ne postoje u golang-u. Ovo predstavlja i jednu od većih zamerki ovom programskom jeziku. Umesto toga upravljanje greškama se u golang-u vrši tako što svaka funkcija koja bi mogla da, gledano iz ugla ostalih programskih jezika, izazove izuzetak vraća tuple vrednosti gde je poslednja vrednost *nil* ako nije došlo do greške a u suprotnom je instance struct-a tog konkretnog tipa greške.

### Distribuirane transakcije

Kako bi se ostvarila distribuirana komunikacija neophodna je komunikacije između mikroservisa, bilo sinhrona ili asinhrona. Iako komuniciraju preko istog brokera svaki od mikroservisa ima različiti pristup pri kreiranju konekcije sa brokerom. Međutim, kada je u pitanju gRPC komunikacija svi servisi dele isti proto fajl te je izdvojen u ovom poglavlju i dat je na slici 5.4.1.



Slika 5.4.1. Protofile

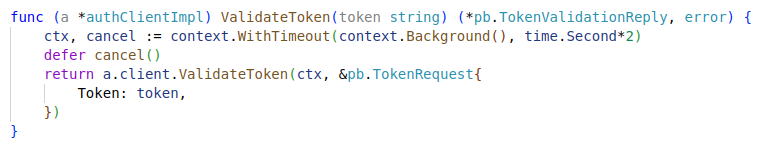


Slika 5.4.2. Implementacija gRPC servera.

Konekcija preko mTLS-a u .NET-u se uspostavlja podešavanje Kestrel-a za komunikaciju preko određenog porta i prikazana je na slici 5.1.4, dakle nije potrebno posebno podešavanje konekcije gRPC servera. U golang-u, s druge strane, je potrebno podesiti konkretnog klijenta da pri uspostavljanju konekcije koristi i zahteva odgovarajuće sertifikate. Meta koja ovo obavlja je data na slici 5.4.3. Dalje na slici 5.4.4 je dat primer slanja zahteva gRPC klijenta iz golang mikroservisa.



Slika 5.4.3. Inicijalizacija gRPC klijenta.



Slika 5.4.4. Slanje zahteva od gRPC klijenta.

Jedna od distribuiranih transakcija koja se izvršava u ovom sistemu se javlja prilikom registracije korisnika. Kada se nakon registracije kreira entitet korisnika, E-mail tog korisnika još uvek nije verifikovan. Zato se korišćenjem Outbox obrasca šalje poruka preko NATS broker keš mikorservisu koji pamti generisani kod za potvrdu E-mail-a određeni vremenski period. Dakle transakcija kreće slanjem zahteva za registraciju auth mikroservisu preko REST API-ja. Implementacija te metode kontrolera je data na slici 5.4.5. Zatim je na slikama 5.4.6 i 5.4.7 data implementacija servisa za kreiranje korisnika. Prvo se započinje transakcija nad PostgreSQL bazom u okviru koje se kreira korisnik ukoliko već ne postoji. Nakon toga se dodaje uloga korisnika. Podrazumevana uloga korisnika je ,,USER“. Zatim se, konkretno na slici 5.4.7, sledi generisanje koda za verifikaciju E-mail-a. Nakon čega se u bazu upisuje novi entitet Outbox poruke koje će čitati i slati pozadinski servis. Ovo je ujedno i poslednja operacija PostgreSQL baze podataka pa se transakcija ovde završava. Na kraju se pokreće pozadinski Task koji će poslati novo kreiranu poruku. Ako u bilo kom delu transakcije dođe do pojave izuzetka vrši se ROLLBACK operacija i izuzetak se prosleđuje nazad kontroleru tako da može o tome da obavesti korisnika. Na slici 5.4.8 dat je isečak iz tabele Outbox poruka.



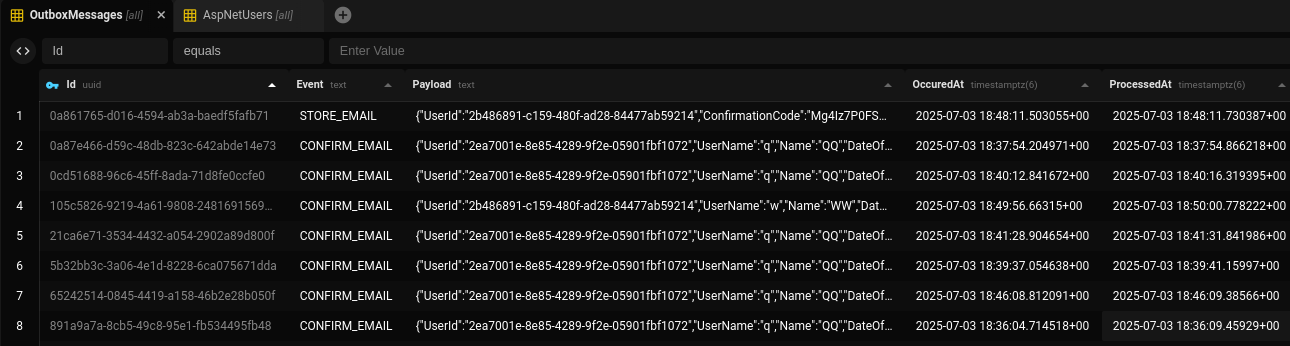
Slika 5.4.5. Implementacija metode kontrolera za registrovanje korisnika.



Slika 5.4.6. Prvi deo implementacije metode servisa za kreiranje korisnika.

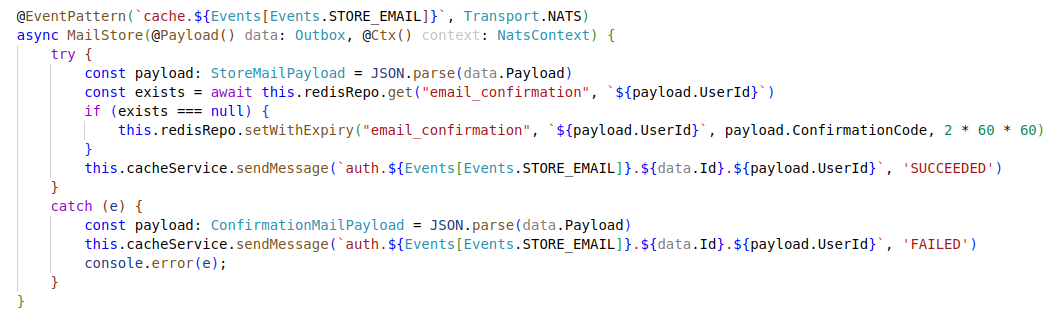


Slika 5.4.7. Drugi deo implementacije metode servisa za kreiranje korisnika.

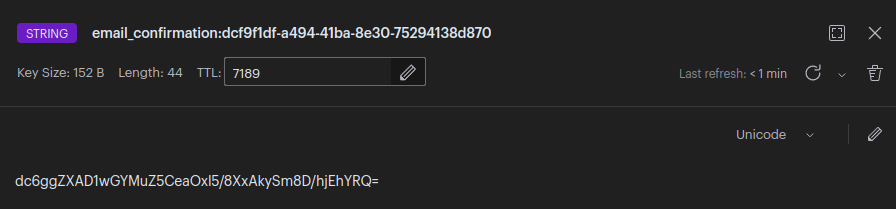


Slika 5.4.8. Primeri Outbox poruka.

NestJS keš servis sluša preko brokera poruke o kreiranju profila. Za svaku pristiglu poruku izvršava se definisana callback funkcija. Implementacija ove funkcije je data na slici 5.4.9, kao što je prethodno pomenuto neophodno je da ova funkcija bude idempotentna ili da se implementira sistem za praćenje već primljenih poruka. U ovom slučaju funkcija je idempotentna tako što se prvo vrši provera da li je ključ već dodat u Redis bazi i ako nije dodaje se dok ako jeste nakon čega se u oba slučaja šalje obaveštenje da je poruka primljena. U slučaju da dođe do neke greške, o grešci se obaveštava i Auth mirkoservis. Primer ključa za potvrdu Mail-a iz Redis baze je dat na slici 5.4.10. Nakon što odgovor Keš servisa uspešno stigne do Auth servisa obrađuje se metodom prikazanom na slici 5.4.11, tako da se, ukoliko je došlo do greške pri memorisanju konfirmacionog koda korisnika, briše entitet korisnika iz PostgreSQL baze podataka.



Slika 5.4.9. Metoda koja se izvršava kada stigne poruka da je korisnik kreiran.



Slika 5.4.10. Primer ključa iz Redis baze keš mikroservisa.



Slika 5.4.11. Obrada odgovora keš servisa.

Slične distribuirane transakcije su implementirane za operaciju konfirmacije E-mail-a koja obuhvata sva tri mikroservisa. Prvo se od Auth mikroservisa zahtev prosleđuje Keš mikroservisu na isti način kao i u prethodno opisanoj transakciji, zatim ukoliko Keš mikroservis potvrdi validnost prosleđenog koda, onda se Profil mikroservisu šalje zahtev preko REST API-ja, metodom prikazanom na slici 5.2.9, da kreira profil za novonastalog korisnika. Ovom transakcijom je kombinovana asinhrona i sinhrona komunikaciji pri implementaciji Saga obrasca.

Takođe udaljena validacija JWT-a predstavlja distribuiranu transakciju, naime middleware Profil mikroservisa preko gRPC-a poziva centralni Auth mikroservis koji validira token. Na slici 5.3.7 je prikazan middleware dok je na slici 5.4.4 prikazana implementacija gRPC metode koja poziva server, tako da je na slici 5.4.2 prikazana implementacija serverske obrade zahteva.

## **ZAKLJUČAK**

U okviru teorijskog dela ovog rada su najpre objašnjeni tipovi arhitektura za razvoj aplikacija. Konkretno objašnjene su arhitekture monolita, slojevitog monolita, modularnog monolita, servisno-orijentisane arhitekture i mikroservisne arhitekture. Takođe navedene su prednosti i mane svih ovih pristupa nakon čega je ukratko izveden presek situacija koje su najpogodnije za upotrebu svake od navedenih arhitektura. Nakon toga sledi pregled principa koji obezbeđuju sigurnost komunikacije preko mreže, što uključuje i komunikaciju između mikroservisa. Dat je, zatim, i kratak pregled tipičnih napada na web aplikacije. Glavni akcenat nakon toga je na objašnjenju TLS protokola kao najpopularnijeg protokola za bezbednu komunikaciju danas. Posle čega slede i opisi projektnih obrazaca koji služe da komunikaciju između mikroservisa učine pouzdanom, garantovanjem slanja poruke i omogućavanjem upotrebe distribuiranih transakcija.

U okviru praktičnog dela, dati su relevantni delovi kodova koji služe da demonstriraju primenu prethodno objašnjenih tehnika i principa. Implementirana su tri mikroservisa u tri različite tehnologije. Što se tiče komunikaciju prikazane su upotrebe sinhronih i asinhronih pritupa razmena informacija, što uključuje komuniciranje preko NATS brokera poruka, preko gRPC-ja i preko REST API-ja. Tako da je za svaki tip komunikacije uključena upotreba TLS protokola. Na pojedinim mestima iskorišćen je mTLS protokol dok su na drugim mestima korišćene alternativne tehnike autentifikacije, poput autentifikacije korisničkim imenom i šifrom. Takođe date su implementacije centralne i distribuirane autorizacije. Od projektnih obrazaca koji omogućavaju garantovano slanje poruka implementiran je Outbox obrazac koji u sklopu sa Sage obrascem omogućava upotrebu distribuiranih transakcija.

Dalja moguća unapređenja ovog sistema su dvojaka. Prvo je moguća implementacija preostalih obrazaca za demonstrativne svrhe. Drugo, što se tiče unapređenja konkretnog sistema iz ugla realnog razvoja aplikacija, moguće je implementirati obrasce kao što su Prekidač kola, koji služi da efikasno detektuje pad nekog dela sistema i postara se da zahtevi za koje su ti delovi sistema neophodni vrate poruke o grešci sa minimalnim gubitkom vremena na čekanje odgovora. Takođe postoje obrasci koji podrazumevaju implementaciju takozvanih side car servisa koji služe da upravljaju direktnom komunikacijom između mikroservisa (REST API i gRPC). Na ovaj način implementirana komunikacija omogućava efikasno ograničavanje količine saobraćaja i upravljanje dozvolama mikroservisa za komunikaciju sa tačno određenim mikroservisom.

## **LITERATURA**

[1] „What is an application architecture?“, dostupno na:

<https://www.redhat.com/en/topics/cloud-native-apps/what-is-an-application-architecture>, 17.7.2025.

[2] „What is user experience (UX)?“, dostupno na:

<https://www.ibm.com/think/topics/user-experience>, 17.7.2025.

[3] „What is Software Quality?“, dostupno na:

<https://www.computer.org/resources/what-is-software-quality>, 17.7.2025.

[4] „Monolith first“, dostupno na:

<https://martinfowler.com/bliki/MonolithFirst.html>, 17.7.2025.

[5] „Microservices vs. Monolithic architecture“, dostupno na:

<https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>, 17.7.2025.

[6] ,,Transition from a monolith to microservice architecture“, dostupno na:

<https://oceanobe.com/news/transition-from-a-monolith-to-microservice-architecture/706>, 17.7.2025.

[7] ,,Scaling Monolithic Applications“, dostupno na:

<https://medium.com/swlh/scaling-monolithic-applications-3c69193f942a>, 17.7.2025.

[8] ,,Layered architecture“, dostupno na:

<https://cs.uwaterloo.ca/~m2nagapp/courses/CS446/1195/Arch_Design_Activity/Layered.pdf>, 17.7.2025.

[9] ,,Layered Architecture“, dostupno na:

<https://herbertograca.com/2017/08/03/layered-architecture/>, 17.7.2025

[10] ,,Layered, Microservices, and Modular Monolithic“, dostupno na:

<https://medium.com/@shahrukhkhan_7802/layered-microservices-and-modular-monolithic-454efda8b2df>, 17.7.2025.

[11] ,,Layered monolith“, dostupno na:

<https://microservices.io/articles/draftZZZ/monolith-patterns/layered-monolith.html>, 17.7.2025.

[12] Ruoyu Su, Xiaozhou Li. ,,Modular Monolith: Is This the Trend in Software Architecture?“, arXiv: 2401.11867v1, 2024

[13] ,,Modular Monolith: The Balance Between Simplicity and Scalability in Software Architecture“, dostupno na: <https://medium.com/@alexglushenkov/modular-monolith-the-balance-between-simplicity-and-scalability-in-software-architecture-b34c0e4243e2>, 18.7.2025.

[14] ,,What Is a Modular Monolith?“, dostupno na:

<https://www.milanjovanovic.tech/blog/what-is-a-modular-monolith>, 18.7.2025.

[15] ,,What is SOA (Service-Oriented Architecture)?“, dostupno na:

<https://aws.amazon.com/what-is/service-oriented-architecture/>, 18.7.2025.

[16] ,,What is service-oriented architecture (SOA)?“, dostupno na:

<https://www.ibm.com/think/topics/soa>, 18.7.2025.

[17] ,,What’s the Difference Between SOA and Microservices?“, dostupno na:

<https://aws.amazon.com/compare/the-difference-between-soa-microservices/>, 18.7.2025.

[18] ,,SOA vs microservices: What’s the difference?“, dostupno na:

<https://www.atlassian.com/microservices/microservices-architecture/soa-vs-microservices>, 18.7.2025.

[19] ,,Service-Oriented Architecture vs. Microservices: Choosing the Right Option“, dostupno na: <https://www.openlegacy.com/blog/service-oriented-architecture-vs-microservices>, 18.7.2025.

[20] ,,What are microservices?“, dostupno na:

<https://microservices.io/>, 18.7.2025.

[21] ,,What are microservices? Guide to modern application architecture“, dostupno na:

<https://www.atlassian.com/microservices/microservices-architecture>, 18.7.2025.

[22] ,,Pattern: Database per service“, dostupno na:

<https://microservices.io/patterns/data/database-per-service.html>, 18.7.2025.

[23] ,,What are microservices?“, dostupno na:

<https://www.ibm.com/think/topics/microservices>, 18.7.2025.

[24] ,,Microservices“, dostupno na:

<https://aws.amazon.com/microservices/>, 18.7.2025.

[25] ,,What is zero trust?“, dostupno na:

<https://www.ibm.com/think/topics/zero-trust>, 18.7.2025.

[26] ,,What is sniffing in regards to data?“, dostupno na:

<https://www.lenovo.com/us/en/glossary/sniffing/?orgRef=https%253A%252F%252Fwww.google.com%252F>, 18.7.2025.

[27] ,,What is spoofing?“, dostupno na:

<https://www.cisco.com/site/us/en/learn/topics/security/what-is-spoofing.html>, 18.7.2025.

[28] ,,What is a man-in-the-middle (MITM) attack?“, dostupno na:

<https://www.ibm.com/think/topics/man-in-the-middle>, 18.7.2025.

[29] ,,Denial of Service (DoS) guidance“, dostupno na:

<https://www.ncsc.gov.uk/collection/denial-service-dos-guidance-collection>, 18.7.2025.

[30] ,,replay attack“, dostupno na:

<https://csrc.nist.gov/glossary/term/replay_attack>, 19.7.2025.

[31] ,,CIA Triad“, dostupno na:

<https://www.fortinet.com/resources/cyberglossary/cia-triad>, 19.7.2025.

[32] ,,What is encryption?“, dostupno na:

<https://cloud.google.com/learn/what-is-encryption>, 19.7.2025.

[33] Al-Ataby, Ali & Al-Naima, Fawzi. (2010). ,,A Modified High Capacity Image Steganography Technique Based on Wavelet Transform“. Int. Arab J. Inf. Technol.. 7. 358-364.

[34] ,,The Transport Layer Secuirty (TLS) Protocol Version 1.3“, dostupno na:

<https://www.rfc-editor.org/rfc/rfc8446>, 19.7.2025.

[35] ,,The TLS 1.3 Handshake“, dostupno na:

<https://www.ibm.com/docs/en/sdk-java-technology/8?topic=works-tls-13-handshake>, 19.7.2025.

[36] ,,Where exactly in the OSI model does TLS/SSL belong“, dostupno na:

<https://security.stackexchange.com/questions/195229/where-exactly-in-the-osi-model-does-tls-ssl-belong>, 19.7.2025.

[37] ,,What is an SSL certificate?“, dostupno na:

<https://www.cloudflare.com/learning/ssl/what-is-an-ssl-certificate/>, 20.7.2025.

[38] ,,Authentication vs. authorization: What’s the difference“, dostupno na:

<https://www.ibm.com/think/topics/authentication-vs-authorization>, 20.7.2025.

[39] ,,Introduction to JSON Web Tokens“, dostupno na:

<https://jwt.io/introduction>, 20.7.2025.

[40] ,,Adding Salt to Hashing: A Better Way to Store Passwords“, dostupno na:

<https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>, 20.7.2025.

[41] ,,What Are Refresh Tokens and How to Use Them Securely“, dostupno na:

<https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/>, 21.7.2025.

[42] ,,Centralized vs Distributed Authorization“, dostupno na:

<https://www.aserto.com/blog/centralized-vs-distributed-authorization>, 21.7.2025.

[43] ,,What’s the Difference Between an ACID and a BASE Database“, dostupno na:

<https://aws.amazon.com/compare/the-difference-between-acid-and-base-database/>, 21.7.2025.

[44] ,,Synchronous vs. asynchronous microservices communication patterns“, dostupno na:

<https://www.theserverside.com/answer/Synchronous-vs-asynchronous-microservices-communication-patterns>, 21.7.2025.

[45] ,,The two-phase commit process“, dostupno na:

<https://www.ibm.com/docs/ru/txseries/9.1.0?topic=SSAL2T_9.1.0/com.ibm.cics.tx.doc/concepts/c_two_phz_commit_process.htm>, 21.7.2025.

[46] ,,Pattern: Saga“, dostupno na:

<https://microservices.io/patterns/data/saga.html>, 21.7.2025.

[47] ,,Try-Confirm/Cancel Transaction Protocol“, dostupno na:

<https://docs.oracle.com/en/database/oracle/transaction-manager-for-microservices/24.2/tmmdg/tcc-transaction-model.html>, 21.7.2025.

[48] ,,Pattern: Transactional outbox“, dostupno na:

<https://microservices.io/patterns/data/transactional-outbox.html>, 21.7.2025.

[49] ,,Pattern: Command Query Responsibility Segregation (CQRS)“, dostupno na:

<https://microservices.io/patterns/data/cqrs.html>, 21.7.2025.

[50] ,,Pattern: Event sourcing“, dostupno na:

<https://microservices.io/patterns/data/event-sourcing.html>, 21.7.2025.