Mobilna aplikacija za razmjenu poruka putem REST API-a

Rupčić, Tin

VELEUČILIŠTE SUVREMENIH INFORMACIJSKIH TEHNOLOGIJA

STRUČNI PRIJEDIPLOMSKI STUDIJ INFORMACIJSKIH TEHNOLOGIJA

Tin Rupčić

ZAVRŠNI RAD

MOBILNA APLIKACIJA ZA RAZMJENU PORUKA PUTEM REST API-A

Zagreb, listopada 2024.

Veleučilište suvremenih informacijskih tehnologija
10000 Zagreb, Ulica Vjekoslava Klaića 7

Studij:	Stručni prijediplomski studij informacijskih tehnologija smjer programiranje
Student:	**Tin Rupčić**
Matični broj:	2017003

Zadatak završnog rada

Predmet:	Distribuirano objektno programiranje
Naslov:	**Mobilna aplikacija za razmjenu poruka putem REST API-a**
Zadatak:	Napisati mobilnu aplikaciju za android operativni sustav korištenjem Java programskog jezika za razmjenu jednostavnih poruka među korisnicima. Aplikacija mora komunicirati s poslužiteljem putem REST API-a kreiranog za ovu namjenu. U uvodu opisati Representational State Transfer (REST) arhitekturu te na konkretnom primjeru pokazati njene glavne karakteristike i način implementacije.
Mentor:	Tomislav Soldo, pred.
Zadatak uručen kandidatu:	30.10.2023.
Rok za predaju rada:	29.10.2024.
Rad predan:	_____

Povjerenstvo:

dr. sc. Damir Delija, prof. struč. stud.	član predsjednik	_____
Tomislav Soldo, pred.	mentor	_____
Zoran Radek, pred.	član	_____

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

POPIS KODOVA

1. UVOD

Digitalno doba donijelo je sa sobom brojne promjene u načinu komunikacije i razmjene informacija. Mobilne aplikacije postale su neizostavni dio našeg svakodnevnog života. Omogućuju nam obavljanje svakodnevnih zadataka iz naslonjača te brzu i praktičnu komunikaciju s drugima, bez obzira na udaljenost. SMS postaje sve manje popularan dok se aplikacije za razmjenu poruka putem interneta sve više koriste. Za realizaciju ovog projekta odabrani su specifični alati i tehnologije koji odgovaraju najnovijim standardima u industriji softverskog razvoja.

Fokus ovog završnog rada je razvoj mobilne aplikacije te implementacija pozadinskog (engl. *backend*) sustava. Prilikom razvoja pozadinskog sustava (engl. *backend*) koristi se *Kotlin* – moderni jezik koji omogućuje brzo pisanje koda s manje pogrešaka zahvaljujući svojoj sigurnosti tipova i konciznosti. *Intellij* IDE (engl. *integrated development environment, IDE*) je korišten zbog svojih naprednih funkcionalnosti koje olakšavaju razvoj, kao što su inteligentno auto-dovršavanje koda, analiza kvalitete koda i integracija s različitim sustavima za upravljanje bazama podataka.

Baza podataka, koja se koristi, *PostgreSQL*, poznata je po svojoj robusnosti i podršci za kompleksne upite što je čini idealnom za aplikacije koje zahtijevaju pouzdano skladištenje i brzu obradu velikih količina podataka. Također, *PostgreSQL* nudi izvrsnu podršku za simultano korištenje i pouzdanost transakcija što je ključno za aplikacije za razmjenu poruka.

Za razvoj klijentskog dijela aplikacije odabran je Android Studio i Java programski jezik. Java, obzirom na svoju široku upotrebu i stabilnost, pruža odličnu osnovu za razvoj pouzdanih i efikasnih mobilnih aplikacija. Android Studio, s druge strane, nudi bogat set alata i resursa za dizajniranje, razvijanje i testiranje Android aplikacija što omogućava efikasnu realizaciju projekta.

Istražuje se postupak razvoja mobilne aplikacije, proučava osnovni koncepti vezani uz REST API, mobilni razvoj i odabir tehnologija. Analizirani su koraci razvoja pozadinskog dijela aplikacije te izazovi koji se mogu pojaviti. Na kraju rada proizlazi zaključak o prednostima i mogućim usavršavanjima ovakvog sustava u budućnosti.

Drugi dio rada bazira se na teoriji osnovnih pojmova vezanih uz mobilni razvoj, razvoj pozadinskog sustava i upotrebu baza podataka. Potom slijedi osvrt na Reprezentacijski prijenos stanja (engl. *Representational State Transfer Application Programming Interface*) koji je ključan za komunikaciju između klijentskih i poslužiteljskih aplikacija, te se razmatra problematika i potreba za razmjenom poruka putem REST API-ja. Obuhvaćen je i razvoj

pozadinskog sustava uključujući korištenje okvira (engl. *framework*) kao što su *Spring Boot* za brzu izradu aplikacija i JPA za upravljanje podacima i njihovom pohranom. U trećem dijelu, metodologija detaljno opisuje korake koji su potrebni za razvoj mobilne aplikacije i pozadinskog sustava, počevši od planiranja i analize zahtjeva, dizajniranja arhitekture, implementacije funkcionalnosti, testiranja i na kraju, implementacije. Četvrti dio rada prikazuje analizu i rezultate istraživanja te su predstavljene prednosti korištenja Kotlin-a i Java-e. Također su opisane performanse same aplikacije, njena skalabilnost i sigurnost koje su ključni faktori za uspješnost na tržištu.

Na kraju je zaključak o cijelom radu i smjernice za moguća istraživanja i unaprjeđenja koji proizlazi iz cilja istraživačkog rada: pružiti dublje razumijevanje procesa razvoja mobilnih aplikacija za razmjenu poruka te istražiti izazove i prednosti navedene tehnološke domene.

2. TEORIJSKI PREGLED PROBLEMA I OPIS TEHNOLOGIJA

U ovom poglavlju su detaljnije analizirane ključne tehnologije korištene za razvoj ove aplikacije. Koncepti poput Heksagonalne arhitekture bit će ključni za razumijevanje projekta.

2.1. Reprezentacijski prijenos stanja i sučelje za programiranje aplikacija

REST (engl. *Representational State Transfer*) *API* (engl. *Application Programming Interface*) je standardizirani način komunikacije između klijenta i poslužitelja putem interneta. *REST* ili *RESTful* je arhitekturni stil koji se temelji na principima prema kojima se trebaju ponašati internet sustavi.

Bez stanja je, što znači da svaki zahtjev klijenta sadrži sve potrebne informacije za obradu tog zahtjeva, bez potrebe za održavanjem stanja sesije na strani poslužitelja.

Modeliran je tako da su resursi osnovne jedinice podataka kojima se radi u sustavu. Svaki resurs ima jedinstveni *URI* (engl. *Uniform Resource Identifier*) koji ga identificira. Primjeri resursa u ovom sustavu mogu biti korisnici, poruke, razgovori.

Sučelje za programiranje aplikacija koristi standardne *HTTP* metode kao što su *GET*, *POST*, *PUT*, *DELETE* za izvršavanje operacija nad resursima:

- GET se koristi za dohvat resursa
- POST za stvaranje novog resursa
- PUT za ažuriranje postojećeg resursa
- DELETE za brisanje resursa

Resursi se mogu predstavljati u različitim formatima kao što su *JSON* (engl. *JavaScript Object Notation*), *XML* (engl. *eXtensible Markup Language*) ili *HTML* (engl. *HyperText Markup Language*), što ovisi o potrebama klijenta i poslužitelja.

Korištenjem *REST*-a u aplikaciji omogućuje se jednostavna komunikacija s pozadinskim sustavom putem standardnih HTTP zahtjeva. Mobilna aplikacija može slati zahtjeve za dohvat, stvaranje, ažuriranje ili brisanje poruka, a pozadinski sustav će te zahtjeve obraditi i odgovoriti rezultatima.

Prednosti korištenja *REST* -a u razvoju mobilne aplikacije uključuju jednostavnu integraciju s različitim platformama. Važno je pravilno dizajnirati kako bi se osigurala jasna i konzistentna komunikacija između klijenta i poslužitelja.

2.2. Razmjena poruka putem REST API-a: Problem i potreba

Razmjena poruka oslanja se na osnovne principe *REST* arhitekture kako bi omogućila komunikaciju između klijenta i poslužitelja. Kao što je navedeno ranije, klijentska aplikacija šalje zahtjeve prema poslužitelju koristeći standardne *HTTP* metode poput *POST*, *GET*, *PUT* i *DELETE* kako bi stvorila, dohvatila, ažurirala ili obrisala poruke.

Ovo su osnovne *HTTP* metode te njihov način rada prikazan kodom koji se koristi u *Spring* okviru (engl. *framework*):

- Klijentska aplikacija šalje *POST* zahtjev prema *REST API*-ju s podacima nove poruke. Pozadinski poslužitelj (engl. *server*) prima zahtjev, provjerava autentičnost i autorizaciju korisnika te sprema poruku u bazu podataka.
- Klijentska aplikacija šalje *GET* zahtjev prema *REST API*-ju kako bi dohvatila poruke. Pozadinski poslužitelj (engl. *server*) dohvaća poruke iz baze podataka i vraća ih klijentskoj aplikaciji u odgovoru.
- Klijentska aplikacija šalje *PUT* zahtjev prema *REST API*-ju s ažuriranim podacima poruke. Pozadinski poslužitelj (engl. *server*) provjerava i ažurira podatke poruke u bazi podataka.
- Klijentska aplikacija šalje *DELETE* zahtjev prema *REST API*-ju za brisanje određene poruke. Pozadinski poslužitelj (engl. *server*) provjerava i briše poruku iz baze podataka.

Prednosti razmjene poruka su da *API* omogućuju skalabilnost aplikacije jer omogućuju distribuiranu arhitekturu te fleksibilnost što omogućava komunikaciju između različitih platformi i tehnologija. Prednost je još i jednostavnost integracije, što znači da *API* omogućuju jednostavnu integraciju s klijentskim aplikacijama što olakšava razvoj i održavanje.

Komponente za ostvarivanje funkcionalnosti ove aplikacije su pozadinski poslužitelj (engl. *server*)i baza podataka. Potrebno je razviti pozadinski poslužitelj (engl. *server*) koji obrađuje zahtjeve klijentskih aplikacija, provjerava autorizaciju korisnika te upravlja porukama u bazi podataka. Potrebno je imati i bazu podataka za pohranu poruka i ostalih potrebnih podataka. *PostgreSQL* će se koristiti kao relacijska baza podataka. Važno je implementirati mehanizme ovjeravanja autentičnosti i autorizacije kako bi se osigurala sigurnost razmjene poruka i zaštitila privatnost korisnika.

Nastavak ovog rada fokusira se na istraživanju metodologije razvoja te na probleme mobilne aplikacije. Analiziraju se i koraci koji su potrebni za implementaciju.

Iako se spominju ovjeravanje autentičnosti i autorizacija kao ključne komponente,

implementacija sigurnosnih mehanizama može biti izazovna. Neispravno implementirani sigurnosni slojevi mogu dovesti do curenja osjetljivih informacija ili neovlaštenog pristupa porukama. Aplikacije za razmjenu poruka zahtijevaju brz internet za pouzdanu komunikaciju, a neprikladna arhitektura pozadinskog sustava može rezultirati lošom performansom što može utjecati na korisničko iskustvo. Iako REST API omogućuju skalabilnost, implementacija skalabilnog sustava može biti kompleksna. Potrebno je osigurati da sustav može rasti s povećanjem broja korisnika i poruka što zahtijeva pažljivo planiranje i arhitekturu. Ako mobilna aplikacija radi u offline načinu rada ili ima sporu internet vezu, može doći do problema sa sinkronizacijom. Zato je potrebno implementirati mehanizme koji omogućuju pouzdanu sinkronizaciju između klijenta i poslužitelja.

Rješavanje ovih problema zahtijeva pažljivo planiranje, implementaciju najboljih praksi i kontinuiranu podršku i održavanje sustava.

2.2.1. Razvoj pozadinskog sustava

Pozadinski sustav ove mobilne aplikacije ključna je komponenta koja omogućava komunikaciju između korisnika. U tu svrhu koristiti će se *Spring Boot* koji je popularan okvir (engl. *framework*) za razvoj Java aplikacija te implementaciju *REST API*-a. *Spring Boot* olakšava brzo pokretanje projekata i integraciju s drugim alatima i tehnologijama pružajući tako efikasno rješenje za razvoj pozadinskog dijela za mobilnu aplikaciju.

U ovu svrhu *Kotlin* je odabran kao programski jezik za razvoj pozadinskog sustava. On pruža prednosti poput izražajne sintakse, statičke tipizacije te kompatibilnosti s Java ekosustavom. Integracija ovog programskog jezika s *Intellij IDEA*, razvojnim okruženjem, olakšava razvoj i optimizaciju kôda.

U nastavku su navedeni neki od bitnih pojmova vezanih za pozadinski sustav.

Za upravljanje ovisnostima (engl. *Dependencies*) i izgradnju projekta koristi se *Maven*. *Maven* je alat za upravljanje projektima i njihovim ovisnostima, omogućuje jednostavno definiranje konfiguracija i učinkovito upravljanje bibliotekama.

Podaci korisnika i poruka pohranjuju se u *PostgreSQL* bazi podataka. *PostgreSQL* je snažan sustav za upravljanje bazama podataka koji podržava složene upite i transakcije, čineći ga pogodnim za ovu vrstu aplikacije.

Kako bi se olakšalo upravljanje bazom podataka i dodali nove verzije shemi, koriste se *Flyway* skripte. *Flyway* omogućuje migraciju baze podataka na sustavan način, osiguravajući dosljednost i kontrolu nad verzijama sheme.

Arhitektura pozadinskog sustava temelji se na heksagonalnom pristupu. Ova arhitektura

promiče jasnu odvojenost između poslovne logike, sučelja i infrastrukturnih slojeva. To olakšava testiranje te održavanje aplikacije.

Kako bi korisnici primali obavijesti o novim porukama, integrira se *Firebase Notification* servis. Ovaj servis omogućuje jednostavno slanje obavijesti korisnicima putem notifikacija iniciranih od strane klijenta (engl. *push notification*).

2.2.2. Spring Boot

Spring Boot je popularan okvir baziran na *Spring*-u, korišten za razvoj Java aplikacija koji olakšava brzo pokretanje projekata i integraciju s drugim alatima i tehnologijama te omogućuje efikasno razvijanje pozadinskog sustava.

Ovaj okvir omogućuje jednostavno pokretanje projekta bez potrebe za kompleksnom konfiguracijom što ubrzava proces razvoja čime se smanjuje potreba za ručnim podešavanjem. Lako se integrira s drugim projektima u Spring ekosustavu, poput *Spring Data* za rad s bazama podataka ili *Spring Security* za sigurnost aplikacije.

Podržava *Maven* ili *Gradle* za upravljanje ovisnostima projekta, omogućujući jednostavno dodavanje vanjskih biblioteka i alata.

Korištenjem *Spring Boot*-a kao temelja za pozadinsku aplikaciju, olakšava se razvoj i održavanje pružajući pouzdano i efikasno rješenje za izgradnju mobilne aplikacije.

2.2.3. Jakarta Persistence API

Jakarta *Persistance API* ili *JPA* je Java specifikacija koja pruža standardizirani način za upravljanje relacijskim podacima u Java aplikacijama. U kombinaciji s *Spring Boot*-om, JPA olakšava razvoj pozadinskog dijela aplikacije pružajući efikasne mehanizme za povezivanje s bazom podataka, mapiranje objekata na relacijske tablice i izvođenje operacija nad podacima. To se postiže korištenjem anotacija poput:

- *@Entity* anotacija se koristi za označavanje Java klasa koje predstavljaju entitete u bazi podataka. Klasa označena ovom anotacijom mora imati primarni ključ koji je označen anotacijom *@Id*.
- *@Table* anotacija koristi se za postavljanje imena tablica u bazi. Klasa koja ima anotaciju *@Entity* i ne traži se da se tablica zove po klasi, dodaje se *@Table* anotaciju.
- *@Column* anotacija se koristi za prilagodbu imena stupca u tablici baze podataka, tj ako naziv polja u klasi ne odgovara nazivu stupca u tablici.

Spring Boot automatski prepoznaje ove anotacije i upravlja entitetima tijekom izvršavanja aplikacije.

Spring Boot, uz podršku *JPA*, omogućuje automatsko stvaranje tablica u bazi podataka na

temelju definiranih *JPA* entiteta što znači da se struktura baze podataka može definirati pomoću Java objekata, bez potrebe za ručnim stvaranjem tablica. Omogućuje izvođenje kompleksnih upita nad bazom podataka pomoću *JPQL* (engl. *Java Persistence Query Language*). *Spring Boot* omogućuje izvršavanje tih upita kroz JPA repozitorije koji će biti prikazani u dijelovima kôda ovog rada.

Jedna od čestih anotacija je @Transactional. Korištenjem te anotacije, *Spring Boot* definira transakcije čime se osigurava dosljednost podataka. U slučaju da transakcija, odnosno upit na bazu ne prođe, funkcija automatski radi proces vraćanja baze podataka u prethodno definirano stanje, obično radi oporavka od pogreške (engl. *rollback*) te dovodi stanje tablice na početno.

Sučelje poput *JpaRepository*, *Spring Boot* automatski generira implementaciju repozitorija za osnovne *CRUD* (Kreiraj (engl. Create), Čitaj (engl. Read), Ažuriraj (engl. Update) i Obriši (engl. Delete)) operacija i omogućuje prilagodbu dodavanjem metoda s prilagođenim upitima. Automatski prepoznaje elemente entiteta te kroz pisanje metode daje korisniku predloške.

U konačnici, kombinacija *Spring Boota* i JPA pruža moćan alat za upravljanje podacima u aplikacijama. Olakšava razvoj pozadinskog dijela aplikacije pružajući standardiziran pristup upravljanju podacima, što rezultira pouzdanim aplikacijama.

2.2.4. Intellij i Kotlin

Intellij IDEA je integrirano razvojno okruženje (engl. *integrated development environment, IDE*) koje pruža napredne alate za razvoj aplikacija. Kao što je navedeno na početku, za razvoj pozadinskog sustava koristit će se *Kotlin* programski jezik. *Kotlin* je moderni, statički tipizirani programski jezik koji se razvio iz potreba programera koji su tražili alternativu postojećim jezicima kao što je Java. Razvijen od strane *JetBrains*-a, iste tvrtke koja stoji iza *IntelliJ* IDEA.

Jedna od ključnih prednosti Kotlina je njegova povezanost s postojećim Java kodom. To znači da se *Kotlin* može koristiti za pisanje novog koda u postojećim Java projektima, omogućavajući postupno uvođenje novih značajki i modernizaciju koda bez potrebe za potpunim prelaskom na *Kotlin*. Također, *Kotlin* se može koristiti za izradu Android aplikacija, što je postalo izuzetno popularno zbog sintakse koja je često čišća u odnosu na Javu, no u ovom projektu to je iznimka.

2.2.5. Maven i upravljanje ovisnostima

Maven je popularan alat za upravljanje projektima u Java ekosustavu. Omogućuje jednostavno definiranje ovisnosti projekta putem XML datoteke što olakšava dodavanje i ažuriranje biblioteka i alata potrebnih za razvoj aplikacije. *Maven* osigurava dosljednost u

procesu izgradnje projekta kroz standardizirane konvencije i automatsko upravljanje ovisnostima (engl. *dependency*).

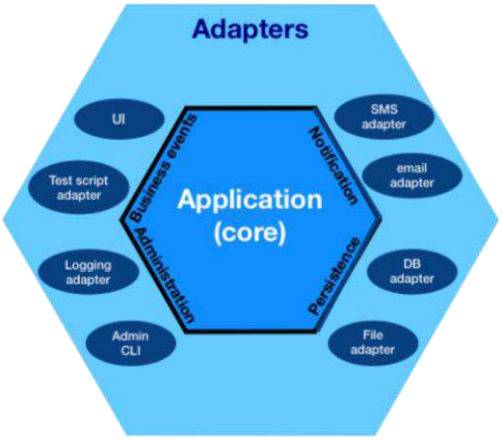
2.2.6. PostgreSQL i Flyway Skripte

PostgreSQL je moćna relacijska baza podataka otvorenog izvora (engl. *open-source*) koja pruža visoku razinu pouzdanosti i fleksibilnosti. Korištenjem *Flyway* skripti, olakšava se upravljanje verzijama baze podataka. *Flyway* omogućuje jednostavno praćenje i primjenu promjena u strukturi baze podataka kroz postavljanje novih verzija skripti. To omogućuje jednostavnu nadogradnju i održavanje baze tijekom vremena, što je ključno za razvoj i skaliranje aplikacije.

2.2.7. Heksagonalna arhitektura

Heksagonalna arhitektura, također poznata kao Portovi i Adapteri arhitektura, ističe se svojom jasnom razdiobom odgovornosti i modularnošću. Ključna ideja iza ove arhitekture je razdvajanje aplikacijske logike od vanjskih utjecaja, poput baze podataka ili vanjskih servisa, čime se postiže veća fleksibilnost, održavanje testova i sustava.

U heksagonalnoj arhitekturi (Slika 1), aplikacijska logika je smještena u središte ili jezgru sustava. Ova jezgra ne ovisi o konkretnim implementacijama vanjskih komponenti, već koristi apstrakcije, poput sučelja ili portova, kako bi komunicirala s vanjskim svijetom. Ovi portovi definiraju načine komunikacije s vanjskim sustavima.



Slika 1. Heksagonalna arhitektura (Wikipedia, 2024.)

Adapteri su komponente odgovorne za implementaciju konkretnih detalja komunikacije s vanjskim svijetom. Primjerice, adapter za bazu podataka implementira način povezivanja s konkretnom bazom podataka i izvršavanje SQL upita.

Prednosti heksagonalne arhitekture uključuju razdvajanje komponenti što omogućuje lakše

dodavanje, mijenjanje ili uklanjanje dijelova sustava bez utjecaja na druge dijelove. Omogućuje jednostavno pisanje jediničnih (engl. *unit*) testova jer se mogu simulirati ili zamijeniti vanjski sustavi s lažnim implementacijama (engl. *mocks*).

Heksagonalna arhitektura je postala popularna u modernom softverskom inženjerstvu kao odgovor na kompleksnost i promjene u zahtjevima.

2.2.8. Docker

Docker je popularna tehnologija koja omogućuje pakiranje aplikacija i njihovih ovisnosti u kontejnere. Kontejneri osiguravaju dosljedno okruženje tijekom razvoja, testiranja i produkcije aplikacija.

2.3. Android

Android aplikacija za razmjenu poruka razvijena je uz pomoć Android Studija, vodećeg integriranog razvojnog okruženja za Android platformu. Java programski jezik je odabran za implementaciju korisničkog sučelja i logike aplikacije.

Kako bi komunikacija s pozadinskim sustavom bila učinkovitija, odabrana je *Retrofit* biblioteku.

Korisničko sučelje aplikacije dizajnirano je putem XML-a. Korištenjem Android-ovih komponenti i materijalnog dizajna, postiže se privlačan izgled te korisniku prijateljski oblik aplikacije.

2.3.1. Mobilni razvoj i Android Studio

Mobilni razvoj predstavlja proces izrade aplikacija za mobilne uređaje kao što su pametni telefoni i tableti. Android operativni sustav, razvijen od strane Google-a, zauzima veliki udio na tržištu mobilnih uređaja. Za razvoj aplikacija za Android platformu jedno od ključnih razvojnih okruženja je Android Studio. Ovo okruženje pruža alate za razvoj, testiranje i optimizaciju aplikacija te olakšava integraciju s raznim Android uređajima.

2.3.2. Java programski jezik

Java je popularan široko korišten programski jezik u svijetu mobilnog razvoja. Njezina platforma omogućuje da se isti kôd može izvoditi na različitim operativnim sustavima. Java je poznata po svojoj sintaktičkoj jasnoći i snažnoj podršci za objektno orijentirano programiranje. U kontekstu mobilnog razvoja za Android Java se često koristi za pisanje funkcionalnosti i logike aplikacija.

2.3.3. Retrofit za komunikaciju putem REST API-a

Retrofit je popularna biblioteka za Android koja olakšava implementaciju komunikacije

putem *REST API*-a. Omogućuje jednostavno definiranje API poziva koristeći Java sučelja te automatski pretvara odgovore u odgovarajuće objekte. Ova biblioteka olakšava rad s HTTP zahtjevima, rukovanje odgovorima i upravljanje mrežnim zahtjevima.

2.3.4. Firebase notifikacije

Za postizanje trenutne komunikacije i obavještavanja korisnika o novim porukama, korišten je *Firebase*. *Firebase* pruža usluge temeljene u oblaku (engl. *cloud-based*), uključujući komunikaciju u stvarnom vremenu (engl. *real-time*), koja je ključna za ovu mobilnu aplikaciju.

U ovom poglavlju dobiven je uvid u temeljne aspekte koji se obrađuju te su istražene ključne tehnologije i alate koji su upotrjebljeni za razvoj mobilne aplikacije za razmjenu poruka putem REST API-a.

2.4. SWOT analiza

Tablica 1. Swot analiza

SNAGE (engl. *STRENGTHS*)	SLABOSTI (engl. *WEAKNESSES*)
Stabilnost i sigurnost aplikacije. Jednostavno sučelje i intuitivno korisničko iskustvo. Integracija sa Firebase-om za brzu i pouzdanu komunikaciju. Kontinuirane nadogradnje i podrška.	Potencijalna složenost korištenja za nove korisnike. Nedostatak nekih naprednih značajki koje konkurencija može imati. Ograničenje na određene platforme i uređaje. Potreba za stalnim održavanjem i ažuriranjem aplikacije.
PRILIKE (engl. *OPPORTUNITIES*)	PRIJETNJE (engl. *THREATS*)
Rastući broj korisnika mobilnih aplikacija. Povećana potražnja za sigurnom i privatnom komunikacijom. Mogućnost integracije s novim tehnologijama za poboljšanje funkcionalnosti. Međunarodno širenje primjene za novo tržište	Pojava novih konkurentnih proizvoda na tržištu. Promjene u tehnološkom okruženju koje zahtijevaju brzu prilagodbu. Gubitak korisnika zbog nezadovoljstva učinkom ili sigurnošću. Regulatorna ograničenja ili zakonske promjene koje utječu na način korištenja aplikacije.

Ova SWOT analiza (Tablica 1, Slika 2) pruža uvid u trenutno stanje naše chat aplikacije, identificira snage i slabosti te prepoznaje prilike i prijetnje kojima se suočavamo na tržištu. To omogućuje bolje razumijevanje pozicije i osmišljavanje strategije za poboljšanje i rast aplikacije u budućnosti.

Slika 2. SWOT (https://www.canva.com)

3. ANALIZA I RAZMATRANJE IMPLEMENTACIJSKIH ASPEKATA KLIJENTA I POSLUŽITELJA

U ovom poglavlju detaljnije se razmatra problematika vezana uz implementaciju mobilne aplikacije te kako se ona povezuje s teorijskim konceptima i praksom. Chat aplikacija, kao što je opisana u uvodu, zahtijeva složenu arhitekturu i pravilno upravljanje podacima kako bi pružila korisnicima funkcionalnosti za slanje i primanje poruka, upravljanje kontaktima te pristupanje povijesti poruka.

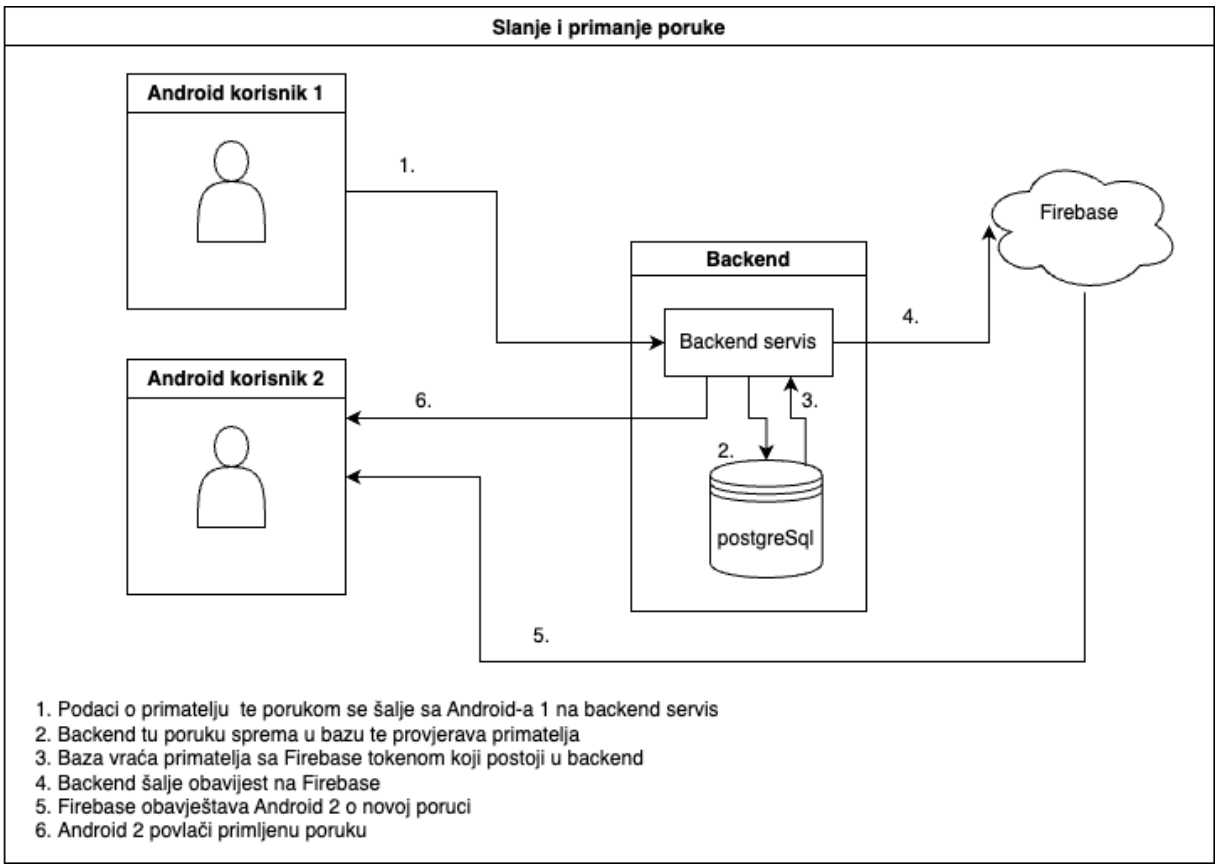
Implementacija sigurnosnih mehanizama i autorizacije je od visoke važnosti kako bi se osiguralo da samo ovlašteni korisnici imaju pristup aplikaciji i njenim funkcionalnostima. Ovo uključuje ovjeravanje autentičnosti korisnika prilikom prijave, upravljanje pravima pristupa te zaštitu privatnosti korisničkih podataka.

Aplikacija mora biti učinkovita kako bi mogla podržati rastući broj korisnika i poruka. To zahtijeva optimizirano upravljanje bazom podataka, brzu obradu i dostavu poruka te prilagodljivu infrastrukturu koja može rasti s potrebama korisnika.

Važno je dizajnirati korisničko sučelje koje je intuitivno za korištenje i pruža ugodno iskustvo korisnicima. To uključuje pregledne liste kontakata i razgovora, jednostavan način slanja i primanja poruka. Proces rada aplikacije opisan je u crtežu (Slika 3) koji objašnjava komunikaciju između dva korisnika putem mobilne aplikacije, preko pozadinskog dijela te *Firebase*-a.

Mobilna aplikacija može biti dostupna na engleskom i hrvatskom jeziku, automatski se prilagođava jeziku telefona. Ova prilagodljivost omogućuje korisnicima da koriste aplikaciju na jeziku koji preferiraju ili koji odgovara postavkama njihovog uređaja.

Chat aplikacija zahtijeva redovito održavanje i nadogradnje kako bi se osigurala stabilnost, sigurnost i ispravan rad. To uključuje praćenje sigurnosnih propisa, ispravke grešaka, poboljšanja performansi te implementaciju novih značajki u skladu s potrebama korisnika.



Slika 3. Chat aplikacija (draw.io)

3.1. Sigurnost i Autorizacija

Sigurnost podrazumijeva zaštitu korisničkih podataka, sprječavanje neovlaštenog pristupa sustavu te osiguravanje integriteta i povjerljivosti komunikacije. Autorizacija se odnosi na upravljanje pravima pristupa korisnika i određivanje radnji koje svaki korisnik može izvršavati unutar aplikacije.

Sigurnost i autorizacija ključni su dijelovi svake chat aplikacije. Za osiguranje sigurnosti i autorizacije, često se koristi metoda ovjeravanja autentičnosti putem tokena. U ovom slučaju korišten je *Bearer* token koji se šalje s pozadinskog sustava na android te s androida klijenta ponovno na pozadinski sustav.

Na strani pozadinskog sustava provjerava se valjanost *Bearer* tokena pristiglog s Android klijenta pri svakom zahtjevu. Pozadinski sustav koristi sigurnosne mehanizme kao što su JWT (JSON Web Token) za generiranje i provjeru tokena. Nakon što se token provjeri i ovjeravanje autentičnosti korisnika uspješno izvrši, korisniku se omogućuje pristup određenim resursima aplikacije.

Na android platformi, token se obično sprema lokalno kako bi se omogućilo automatsko slanje tokena pri svakom zahtjevu na pozadinski sustav. To se postiže pohranom tokena u lokalnu pohranu podataka uređaja, poput *SharedPreferences*. Kada korisnik izvrši prijavu ili

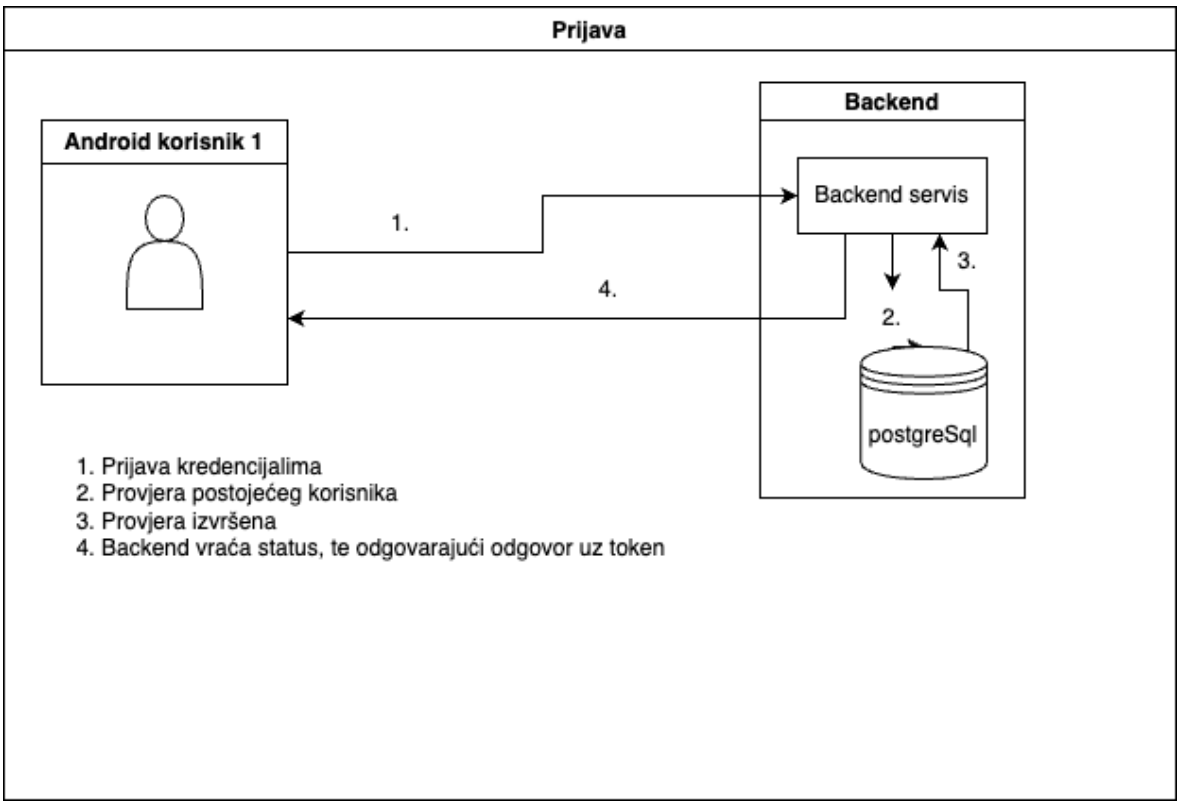
ovjeravanje autentičnosti unutar aplikacije, token se generira i sprema lokalno, zatim se taj token koristi kao dio autorizacijskog zaglavlja pri svakom zahtjevu prema pozadinskom sustavu.

U procesu ovjeravanja autentičnosti na android uređaju, korisnik se obično prijavljuje putem korisničkog imena i lozinke. Kada se ovjeravanje autentičnosti uspješno izvrši, poslužitelj generira *Bearer* token koji se zatim šalje natrag na android uređaj (Slika 4).

Kada token istekne, korisnik će biti odjavljen iz aplikacije ili će biti potrebno ponovno izvršiti prijavu. U tom slučaju, korisnik ponovno unosi svoje korisničke podatke, a poslužitelj generira novi *Bearer* token koji se šalje natrag na android uređaj. Nakon toga, novi token se pohranjuje lokalno i koristi se u daljnjim zahtjevima.

Ovaj pristup osigurava kontinuiranu sigurnost aplikacije jer se korisniku automatski dodjeljuje novi token nakon isteka valjanosti. Na taj način se osigurava da pristup aplikaciji ostaje siguran. Navdeni postupak pruža jednostavan i transparentan korisnički doživljaj jer korisnik ne mora ručno upravljati ovjeravanjem autentičnosti ili generiranjem tokena.

Korištenje *Bearer* tokena na ovaj način omogućuje siguran prijenos podataka između android klijenta i pozadinskog sustava te osigurava da samo ovlašteni korisnici imaju pristup resursima aplikacije. Ovaj pristup osigurava da se korisnički podaci šalju na siguran način te zaštitu od neovlaštenog pristupa.



Slika 4. Prijava korisnika

3.1.1. Bearer Token

Kada se korisnik prijavljuje u mobilnu aplikaciju, njihovi pristupni podaci (korisničko ime/lozinka) sigurno se šalju na poslužitelj. Nakon uspješnog ovjeravanja autentičnosti, poslužitelj generira *Bearer* token.

Taj token sadrži informacije o korisnikovom identitetu i dozvolama. Poslužitelj šalje token natrag u mobilnu aplikaciju kao dio odgovora na ovjeravanje autentičnosti. Mobilna aplikacija sigurno pohranjuje token lokalno na uređaju što uključuje *SharedPreferences*. Kada mobilna aplikacija treba pristupiti resursima na pozadinskom dijelu, uključuje *Bearer* token u zaglavlju zahtjeva. Token djeluje kao dokaz korisnikove autentičnosti i autorizacije.

Poslužitelj prima zahtjev i izvlači *Bearer* token iz zaglavlja zahtjeva. Zatim provjerava valjanost tokena kako bi osigurao njegovu autentičnost i potvrdio da nije istekao.

Ako je token valjan i korisnik je autoriziran za pristup traženom resursu, poslužitelj ispuni zahtjev, ako nije, vraća pogrešku koja ukazuje na neovlašteni pristup.

Ako token istekne, korisnik mora ponovno ovjeriti autentičnost pružajući svoje pristupne podatke. Zatim poslužitelj generira novi *Bearer* token, i postupak se ponavlja.

3.1.2. Firebase

Firebase je platforma koja pruža razne alate za razvoj mobilnih i web aplikacija. Omogućava razvoj bržih i sigurnijih aplikacija bez potrebe za ikakvom implementacijom infrastrukture. Za integraciju *Spring Boot* aplikacije s *Firebase*-om, koristi se *Firebase Admin SDK*. On *SDK* omogućuje komunikaciju između *Spring Boot* aplikacije i *Firebase* servisa. Neki od *Firebase* servisa su:

- baza podataka u stvarnom vremenu (engl. *Realtime Database)*
- *Firestore* u oblaku (engl. *Cloud Firestore)*
- ovjeravanje autentičnosti (engl. *Authentication)*

U ovom slučaju, *Firebase Admin SDK* se koristi za obavljanje operacija kao što su ovjeravanje autentičnosti korisnika i izvršavanje funkcija u Cloud-u.

Što se tiče integracije sa android-om, *Firebase* pruža niz Android SDK-ova koji olakšavaju integraciju. Korištenjem *Firebase SDK*-ova, može se integrirati:

- ovjeravanje autentičnosti
- analitika
- baza podataka
- notifikacije inicirane od strane klijenta (engl. *push notification*)

Chat aplikacija koristi notifikacije inicirane od strane klijenta te ovjeravanje autentičnosti kako bi različiti uređaji mogli komunicirati jedno s drugim.

3.2. Optimizacija pozadinskog sustava i komunikacija s mobilnom aplikacijom

U kontekstu postizanja visoke učinkovitosti i skalabilnosti chat aplikacije, ključno je razmotriti načine upravljanja podacima na poslužitelju te kako ti podaci dolaze do krajnjih korisnika.

Učinkovitost aplikacije može se postići korištenjem optimalnih arhitektonskih obrazaca i tehnologija za poslužitelja, kao i pažljivim dizajnom *API*-ja. Kontroleri, koji su odgovorni za obradu zahtjeva korisnika, trebaju biti optimizirani kako bi brzo i efikasno obradili zahtjeve.

U kontekstu upravljanja bazom podataka važno je koristiti efikasnu tehniku kao što je optimizacija upita kako bi se osigurala brza obrada podataka.

Kada je riječ o slanju podataka na android aplikaciju, važno je optimizirati komunikaciju između poslužitelja i mobilnog klijenta. To uključuje korištenje kompresije podataka, minimiziranje broja zahtjeva te korištenje brzih i pouzdanih protokola za prijenos podataka. Pažljivija implementacija upravljanja vezama s bazom podataka na poslužitelju može smanjiti vrijeme protoka podataka i poboljšati performanse mobilne aplikacije.

Implementacija optimiziranih kontrolera i učinkovitih strategija upravljanja bazom podataka na poslužitelju, zajedno s dobro osmišljenim pristupom komunikaciji s mobilnom aplikacijom, ključni su elementi za postizanje visoke učinkovitosti chat aplikacije. Osiguravaju brzu i pouzdanu uslugu za korisnike, čak i u uvjetima visokog opterećenja i rasta broja korisnika.

4. PRAKTIČNI RAD - MOBILNA APLIKACIJA ZA RAZMJENU PORUKA PUTEM REST API-A

4.1. Poslužitelj (pozadinski dio aplikacije)

Praktični rad na ovoj aplikaciji koristi *Spring Boot* u kombinaciji s *JPA* (engl. *Java Persistence API*) za implementaciju poslužiteljskog sustava. U nastavku je opis kako se koristi *Spring Boot* s *JPA* za registraciju korisnika, upravljanje korisnicima, porukama te komunikaciju s *Firebase*-om.

U *Spring Boot* aplikacijama obično se koristi MVC (engl. *Model-View-Controller*) arhitektura. U ovom slučaju, koristimo tri glavna sloja, Kontroler (engl. *Controller*), Servis (engl. *Service*) i Repozitorij (engl. *Repository*) slojevi.

- Kontroler služi za obradu *HTTP* zahtjeva i vraćanje odgovora. Implementirat će se različite krajnje točke (engl. *endpoint*) rute za registraciju korisnika, upravljanje korisnicima, slanje i primanje poruka te komunikacija s *Firebase*-om.
- Service je sloj koji sadrži poslovnu logiku aplikacije. Implementirat će se metode za obradu zahtjeva koje su primljene od kontrolera. Implementirat će se i logika za registraciju korisnika, upravljanje korisnicima, slanje i primanje poruka.
- Repozitorij je sloj za pristup podacima. Ovdje će biti definiran *JPA* repozitorij za pristup podacima u bazi podataka. Primjerice, repozitorij za korisnike omogućuje spremanje i dohvaćanje korisničkih podataka iz baze.

Kontroler sadrži krajnje točke u koji prima podatke o korisniku (kao što su korisničko ime, lozinka, e-mail adresa) te poziva odgovarajuću metodu iz Service sloja za spremanje korisnika u bazu podataka putem *JPA* repozitorija.

Implementiraju se krajnje točke za upravljanje korisnicima za slanje i primanje poruka između korisnika, za komunikaciju s *Firebase*-om.

Za komunikaciju s *Firebase*-om, slanje notifikacija korisnicima putem *Firebase* Cloud *Messaging*-a, implementira se odgovarajuća metoda u Service sloju koja koristi *Firebase* SDK za slanje poruka na odgovarajuće uređaje na temelju tokena koje je aplikacija dobila prilikom registracije korisnika.

4.1.1. Kontroler

U nastavku je prikazana kontroler klasa (Kôd 1. Kontroler) gdje su prikazane dvije metode koje obrađuju *HTTP GET* i *HTTP POST* zahtjeve:

- login() obrađuje *HTTP GET* zahtjev na /api/user/login putanji. Prihvaća tri parametra: korisničko ime (engl. *username*), lozinka (engl. *password)* i prezentaciju znakova(engl. *token)*, a zatim poziva odgovarajuću metodu u *UserService*-u za ovjeravanje autentičnosti korisnika.
- *register*() metoda koja obrađuje HTTP POST zahtjeve na */api/user* putanji. Prihvaća objekt *user* koji se prosljeđuje kao JSON tijelo zahtjeva, a zatim poziva odgovarajuću metodu u *UserService*-u za registraciju novog korisnika.

UserController klasa predstavlja kontroler koji obrađuje HTTP zahtjeve povezane s korisnicima:

- *@RestController* je oznaka koja označava ovu klasu kao kontroler u *Spring MVC*-u, što znači da će automatski rukovati *HTTP* zahtjevima.
- *@RequestMapping("/api/user")* je anotacija koja određuje osnovni *URL* put koji će biti korišten za sve metode u ovom kontroleru.

```
@RestController
@RequestMapping("/api/user")
class UserController(
    val userService: UserService,
) {
    @GetMapping("/login")
    fun login(
        @RequestParam username: String,
        @RequestParam password: String,
        @RequestParam token: String,
    ): ResponseEntity<UserDto> =
        ResponseEntity.status(HttpStatus.OK).body(
            userService.findByUsernameOrEmailWithPass(username, password,
token),
        )

    @PostMapping
    fun register(@RequestBody user: UserRegisterRequest):
ResponseEntity<MessageBody> =

ResponseEntity.status(HttpStatus.OK).body(MessageBody(userService.insert(u
ser)))
```

Kôd 1. Kontroler

4.1.2. Servis

Service je sučelje definirano funkcijama koje se mogu izvršiti nad korisničkim podacima. Po slijedu ili konvenciji, obično se metode servisa pozivaju u repozitoriju.

Ovo sučelje (Kôd 2. Servis) ima dvije metode:

- *findByUsernameOrEmailWithPass()* za pronalazak korisnika na osnovu korisničkog imena ili e-maila uz provjeru lozinke i tokena
- insert() za registraciju novog korisnika na osnovu objekta *UserRegisterRequest*.

Kao servisnu komponentu u *Spring*-u koristi se anotacija *@Service* te *@Validated* radi provjere validnosti podataka:

- *@Service* anotacija daje informacije *Spring*-u da će se upravljati njegovim životnim ciklusom i moći će se injektirati ili ubaciti u druge komponente kao što je *UserController*.
- *@Validated* oznaka koja označava da će *Spring* provjeravati validnost parametara prilikom poziva metoda ovog servisa što omogućuje primjerenu validaciju podataka prije nego što se pozovu operacije nad korisničkim podacima.

```
@Service
@Validated
interface UserService {

    fun findByUsernameOrEmailWithPass(usernameOrEmail: String, password:
String, token: String): UserDto?

    fun insert(user: UserRegisterRequest): String
```

Kôd 2. Servis

4.1.3. Implementacija servisa

Ova klasa (engl. *class*) implementira *UserService* interface (Kôd 3. Implementacija servisa). U svom konstruktoru kao parametre ima: UserRepository koji omogućuje pristup podacima o korisnicima te *FirebaseRepository* koji omogućuje komunikaciju sa *Firebase* servisima:

- *findByUsernameOrEmailWithPass()* metoda pretražuje *Firebase* token koristeći *firebaseRepository*. Ako token nije pronađen, izvršava se dodatna logika, pretražuje te vraća korisnika iz repozitorija na osnovu korisničkog imena ili e-pošte i lozinke. Ako je token pronađen, priprema se zahtjev za spremanje podataka korisnika u *Firebase* tablicu.
- insert() metoda direktno ubacuje zahtjev (engl. *request*) u tablicu.

```
@Service
@Validated
class UserServiceImpl(
    val userRepository: UserRepository,
    val firebaseRepository: FirebaseRepository,
) : UserService {

    override fun findByUsernameOrEmailWithPass(usernameOrEmail: String,
password: String, token: String): UserDto? {
        val fire = firebaseRepository.findFirebaseToken(token)
        if (fire == null) {
            val user =
userRepository.findByUsernameOrEmail(usernameOrEmail, password)
            val firebaseRequest = FirebaseUserRequest(
                token,
                UserRequest(
                    userId = user?.userId?.value,
                    userName = user?.userName!!.value,
                    phoneNumber = user.phoneNumber.value,
                    userEmail = user.userEmail.value,
                ),
            )
            firebaseRepository.save(firebaseRequest.requestToDomain())
        }

        return userRepository.findByUsernameOrEmail(usernameOrEmail,
password)?.toDto()
            ?: throw DomainNotFoundException("User $usernameOrEmail not
found.")

    override fun insert(user: UserRegisterRequest): String =
userRepository.save(user.requestToDomain())
```

Kôd 3. Implementacija servisa

4.1.4. Implementacija repozitorija

UserRepositoryImpl je klasa koja implementira sučelje *UserRepository*. Repozitorij u *Spring*-u predstavlja sloj koji pruža apstrakciju za rad s podacima. U ovom slučaju repozitorij je klasa koja definira funkcije za izvršavanje operacija kao što su dohvaćanje entiteta po korisničkom ID-u, ili spremanje novih entiteta (Kôd 4. Implementacija repozitorija).

@Repository anotacija označava da će *Spring* automatski implementirati sve osnovne *CRUD* operacije za ovaj repozitorij. Kao što je navedeno ranije, *@Transactional* označava da će se sve metode u ovom repozitoriju izvršavati u okviru transakcije:

- *findChatByUserOrByMessageUserId(userId: Int)* metoda pretražuje korisnike koji su uključeni u razgovor na osnovu ID-a korisnika. Ovo se postiže pozivom odgovarajuće

metode u *userEntityJpaRepository* koja vraća skup entiteta korisnika, a zatim se ti entiteti mapiraju na domenske objekte.

- *insert(user: User)* metoda dodaje novog korisnika u repozitorij. Koristi se Spremi (engl. *save*) metoda *userEntityJpaRepository*-a za spremanje novog korisnika u bazu podataka.

```
@Repository
@Transactional(Transactional.TxType.REQUIRED)
internal class UserRepositoryImpl(
    private val userEntityJpaRepository: UserEntityJpaRepository,
) : UserRepository {

    override fun findChatByUserOrByMessageUserId(userId: Int): Set<User> =
userEntityJpaRepository
.findChatByUserOrByMessageUserId(userId).setToDomain()

    @Transactional
    override fun insert(user: User): User {
        userEntityJpaRepository.save(
            UserEntity(
                user.userId?.value,
                user.userName.value,
                user.password.value,
                user.userEmail.value,
                user.phoneNumber.value,
            ),
        )
        return user
    }
```

Kôd 4. Implementacija repozitorija

4.1.5. JPA Repozitorij

Ovdje je definicija *Spring Data JPA* repozitorija koji nasljeđuje *JpaRepository*. To automatski daje implementaciju osnovnih metoda poput spremi (engl. *save*), pronaći po ID-u (engl. *findById*), obrisati (engl. *delete*), itd. Uz to je i dodatni upit koji je definiran kroz *@Query* anotaciju. Ovaj upit vraća skup entiteta korisnika koji su uključeni u razgovor na osnovu ID-ja korisnika (Kôd 5. JPA repozitorij).

```
@Repository
internal interface UserEntityJpaRepository : JpaRepository<UserEntity,
String> {
    @Query(
        """
SELECT users.* FROM users
JOIN participants ON participants.user_id = users.user_id
```

```
JOIN chats ON chats.chat_id = participants.chat_id
WHERE chats.chat_id
IN (
SELECT chat_id FROM participants
JOIN users ON participants.user_id = users.user_id
WHERE users.user_id = :userId
)
AND users.user_id != :userId
""",
        nativeQuery = true,
    )
    fun findChatByUserOrByMessageUserId(userId: Int): Set<UserEntity>
```

Kôd 5. JPA repozitorij

4.2. Android

Za implementaciju android aplikacije potrebne su komponente koje omogućavaju korisniku nesmetani te prijateljski oblik rada. Neke od najvažnijih su Aktivnosti (engl. *activity*), *RecyclerView, ViewModel, NotificationManager, Retrofit, AsyncTask, SharedPreferences i Intent:*

- Aktivnosti (engl. *activity*) su ključne komponente android aplikacije koje predstavljaju jedan ekran kojim korisnik može upravljati. Koriste se za upravljanje i prikazivanje korisničkog sučelja.
- *RecyclerView* koristi se za prikazivanje dinamičkih popisa podataka kao što su popisi razgovora ili poruka. Omogućuje efikasno upravljanje velikim skupovima podataka i podržava različite raspone prikaza. Obično su prikazane liste poredane po elementima jedan ispod drugog.
- *ViewModel* koristi se za čuvanje i upravljanje podacima specifičnim za korisničko sučelje. On pomaže u očuvanju podataka tijekom promjene konfiguracije uređaja poput rotacije zaslona.
- *Retrofit* je biblioteka za rad s *HTTP* zahtjevima u Android aplikacijama. Koristi se za definiranje i izvršavanje *HTTP* zahtjeva prema *REST API*-ju pozadinskog sustava.
- *AsyncTask* se koristi za izvođenje asinkronih operacija u Android aplikacijama. To uključuje slanje *HTTP* zahtjeva na pozadinski sustav kako bi se izbjeglo blokiranje glavne niti (engl. thread).
- *SharedPreferences* koristi se za pohranu jednostavnih podataka kao što su korisnički tokeni ili postavke aplikacije. Ovo je korisno za očuvanje stanja aplikacije između sesija ili za pohranu osjetljivih podataka lokalno na uređaju.
- *NotificationManager* omogućuje upravljanje notifikacijama koje se prikazuju

korisniku. To je važno za obavještavanje korisnika o novim porukama ili događajima u aplikaciji.

- *Intenti* se koriste za komunikaciju između različitih komponenti aplikacije kao što su pokretanje novih aktivnosti. Omogućuju integraciju različitih dijelova aplikacije i omogućuje korisnicima da lako prolaze kroz aplikaciju.

Prije nego se detaljnije opiše što se događa u kodu, razmotrit će se nekoliko primjera radi boljeg razumijevanja funkcioniranja aplikacije:

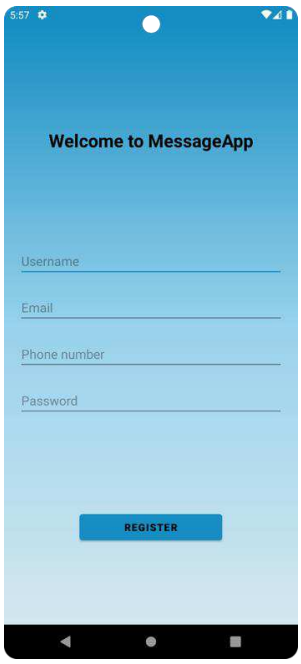
- Unos i slanje poruka: Korisnik otvara aplikaciju, vidi polje za unos teksta poruke i gumb za slanje. Nakon što unese poruku i pritisne gumb za slanje, aplikacija bi trebala poslati poruku na poslužitelj preko internetske veze.
- Odgovor poslužitelja: Nakon što je poruka poslana, aplikacija čeka odgovor od poslužitelja. Ako je slanje poruke uspješno, poslužitelj bi trebao potvrditi primitak poruke. Ako je došlo do pogreške, poslužitelj bi trebao poslati odgovarajući statusni kod kako bi aplikacija znala da je došlo do problema.

4.2.1. Registracija i prijava korisnika

Korisnici se mogu registrirati na aplikaciju koristeći svoje osobne podatke poput e-maila, korisničkog imena, broja telefona i lozinke (Slika 5).

Nakon registracije, korisnici se prijavljuju u aplikaciju koristeći svoje korisničke podatke. U procesu registracije korisnika te mobilne aplikacije na *Firebase* platformi, android šalje zahtjev za dobivanje jedinstvenog identifikatora uređaja, poznatog kao token. Kada korisnik prvi put pokrene aplikaciju na svom android uređaju, *Firebase* generira ovaj token i šalje ga natrag aplikaciji. Nakon što aplikacija primi token, šalje ga na poslužitelj kako bi se registrirao korisnikov uređaj.

Ovaj token je ključan jer omogućuje identifikaciju uređaja u *Firebase* sustavu. Kada se korisnik prijavi u aplikaciju, token se koristi radi osiguranja da se informacije i poruke pravilno usmjeravaju na odgovarajući uređaj. Poslužitelj pohranjuje ove tokene kako bi mogao uskladiti komunikaciju između mobilnih uređaja i *Firebase* platforme te osigurava da mobilni uređaji mogu primati obavijesti i druge relevantne informacije koje su im namijenjene.



Slika 5. Registracija

4.2.2. Stvaranje i pridruživanje razgovorima

Korisnici mobilne aplikacije imaju mogućnost stvaranja novih razgovora ili pridruživanja postojećim (Slika 6). Kada korisnik pošalje poruku nekom korisniku, automatski se dodaje u taj razgovor. Grupni razgovori su u ovoj aplikaciji isključeni. Kako bi se korisnicima pružila bolja iskustva, moguće je razmotriti implementaciju opcije za poziv na razgovor ili proširenje mogućnosti dodavanja sudionika u postojeće razgovore što bi dodatno obogatilo korisničko iskustvo i funkcionalnost aplikacije.

Slika 6. Kontakti

4.2.3. Slanje i prijem poruka

Korisnici mogu slati poruke u razgovorima u kojima sudjeluju (Slika 7). Poruke se šalju na poslužitelj putem internetske veze, a zatim se distribuiraju svim sudionicima razgovora u stvarnom vremenu. Ime razgovora je i ime same osobe s kojom pošiljatelj razgovara.

Slika 7. Chat

```
private void initializeLayoutMessageField() {
    edit_gchat_message = findViewById(R.id.edit_gchat_message);
    Button button_gchat_send = findViewById(R.id.button_gchat_send);
    edit_gchat_message.setSelectAllOnFocus(true);
    edit_gchat_message.setOnKeyListener(this);
    button_gchat_send.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            if (!edit_gchat_message.getText().toString().isEmpty()) {
                sendMessage(new MessageRequest(
                                edit_gchat_message.getText().toString(),
                                new ChatRequest(
userChatDto.getChatDto().getChatId(),
userChatDto.getChatDto().getChatName(),
                                        new UserRequest(
                                                userDto.getUserId(),
                                                userDto.getUserName(),
userDto.getPhoneNumber(), userDto.getUserEmail()
                                        )
                                ), new UserRequest(
                                userDto.getUserId(),
                                userDto.getUserName(),
userDto.getPhoneNumber(), userDto.getUserEmail()
                        )
                        )
                );
            }
        }
    });
}
```

Kôd 6. Inicijalizacija elemenata

Ovaj dio koda omogućuje korisniku da unese tekst poruke, a zatim ga pošalje pritiskom na gumb za slanje poruke:

- *initializeLayoutMessageField() (*Kôd 6. Inicijalizacija elemenata*)* koristi se za inicijalizaciju elemenata korisničkog sučelja koji su relevantni za unos i slanje poruka.
- Najprije se pronalazi Uredi tekst (engl. *EditText),* polje za unos poruke (*edit_gchat_message*) i gumb za slanje poruke (*button_gchat_send*) putem njihovih resursnih identifikatora.
- Zatim se postavlja svojstvo *setSelectAllOnFocus(true)* na Uredi tekst (engl. *EditText)* polje što omogućuje automatsko označavanje cijelog teksta prilikom dobivanja fokusa.
- Postavlja se slušatelj događaja na Uredi tekst (engl. *EditText)* polje kako bi se pratila pritisnuta tipka. Metoda *setOnKeyListener(this)* ukazuje da će trenutna aktivnost biti odgovorna za obradu događaja pritiska tipke.

- Na gumb za slanje poruke postavlja se slušatelj klika (*OnClickListener*) što znači da će se određena akcija izvršiti kada korisnik klikne na gumb.
- Definira se *onClick()* metoda koja se poziva kada korisnik klikne na gumb za slanje poruke.
- Uvjetno se provjerava je li tekst poruke (*edit_gchat_message*) prazan prije slanja. Ako tekst nije prazan, stvara se objekt *MessageRequest* koji sadrži tekst poruke, informacije o razgovoru (*ChatRequest*) i informacije o korisniku koji šalje poruku (UserRequest).
- Nakon toga poziva se metoda *sendMessage()* koja se koristi za slanje poruke s odgovarajućim parametrima.

```
public void sendMessage(MessageRequest messageRequest) {
    Call<MessageBody> call =
AppController.userClient.sendMessage(messageRequest);
    call.enqueue(new Callback<MessageBody>() {
        @Override
        public void onResponse(Call<MessageBody> call,
Response<MessageBody> response) {
            if (response.isSuccessful()) {
                edit_gchat_message.setText("");
            }
        }
        @Override
        public void onFailure(Call<MessageBody> call, Throwable t) {
            Utils.showToast(MessageActivity.this, t.getMessage());
        }
    });
}
```

Kôd 7. Pošalji poruku

Ovaj dio koda (Kôd 7. Pošalji poruku) omogućuje slanje poruke putem *Retrofit* klijenta na odgovarajuću krajnju točku unutar pozadinskog sustava, a zatim obrađuje odgovor od poslužitelja te reagira na uspješan ili neuspješan rezultat slanja poruke. Ova metoda predstavlja asinkroni zahtjev za slanje poruke putem određene krajnje točke. Metoda *sendMessage()* koja je bez povratnog tipa, odgovorna je za slanje poruke putem koristeći *Retrofit* klijent.

Ako je odgovor uspješan, prazni se tekstualno polje za unos poruke (*edit_gchat_message*) kako bi korisnik mogao unijeti novu poruku. *onFailure()* poziva se ako je došlo do greške prilikom slanja zahtjeva ili obrade odgovora te se prikazuje obavijest korisniku o grešci pomoću *Toast* poruke.

4.2.4. Pregled Povijesti Poruka

Korisnici mogu pregledati povijest poruka unutar svakog razgovora.

Povijest poruka prikazuje se kronološki, omogućujući korisnicima da prate prethodne razgovore. Na svakoj poruci prikazano je vrijeme slanja poruke te je između svake poruke postavljen datum slanja.

4.2.5. Pročitana poruka

Kada korisnik pročita poruku unutar razgovora, prikaz poruke u listi poruka može se promijeniti kako bi se označilo da je poruka već pročitana. Jedan od načina da se to postigne je promjena stila fonta iz podebljanog (engl. *bold*) u normalni stil.

Ovaj vizualni signal pomaže sudionicima razgovora da lako identificiraju koje su poruke već pregledane. Na taj način korisnici mogu brzo skenirati listu poruka i fokusirati se na one poruke koje još nisu pročitali. Implementacija ovog mehanizma za označavanje pročitanih poruka pridonosi boljoj upotrebljivosti aplikacije za chat tako što korisnicima pruža jasne signale o statusu poruka unutar razgovora.

4.2.6. Upravljanje Kontaktima

Kada se korisnici registriraju u aplikaciju, automatski su dodani u imenik svih ostalih korisnika aplikacije. To znači da će svaki novi korisnik biti vidljiv i dostupan za komunikaciju svim ostalim korisnicima. Ovaj automatizirani proces olakšava korisnicima pronalazak i komunikaciju s prijateljima, obitelji i kolegama unutar aplikacije bez potrebe za ručnim dodavanjem kontakata.

4.2.7. Notifikacije

Kada korisnici nisu aktivni unutar aplikacije, još uvijek mogu primati obavijesti ili notifikacije o novim porukama. Aplikacija koristi notifikacije kako bi obavijestila korisnike o važnim događajima dok nisu aktivni u aplikaciji. Na primjer, kada korisnik primi novu poruku u chatu, aplikacija šalje notifikaciju na uređaj korisnika. Notifikacije sadrže osnovne informacije o događaju poput imena osobe koja je poslala poruku i naslov poruke što omogućuje korisnicima da brzo prepoznaju važnost notifikacije i odluče hoće li odgovoriti ili ne.

5. ZAKLJUČAK

Ovaj rad obuhvatio je proces razvoja mobilne aplikacije za razmjenu poruka putem *REST API*-a. Počevši od teorijskog pregleda osnovnih pojmova kao što su *REST API*, mobilnog razvoja i upotrebe baze podataka pa sve do detaljnog opisa ključnih tehnologija i alata, dobiva se sveobuhvatan uvid u cjelokupan proces.

REST API pruža jednostavan i brz način za implementiranje funkcionalnosti razmjene poruka. Kroz analizu problema, potreba i prednosti takvog pristupa, pravilno dizajnirani i implementiran sustav za razmjenu poruka može donijeti brojne koristi korisnicima uključujući brzu i pouzdanu komunikaciju te fleksibilnost.

Implementacija takvog sustava zahtijeva pažljivo planiranje, pravilnu arhitekturu i korištenje odgovarajućih tehnologija i alata. Kroz ovaj rad istražena je i predstavljena najbolja praksa u razvoju pozadinskog dijela aplikacije koristeći *Kotlin, Spring Boot, PostgreSQL* i *Docker*, kao i razvoj korisničkog sučelja mobilne aplikacije koristeći Java i Android Studio.

Iako razvoj mobilne aplikacije donosi mnoge prednosti, postoji i niz izazova i potencijalnih problema koji se mogu pojaviti, uključujući sigurnost, performanse, sinkronizaciju poruka te održavanje sustava. Rješavanje ovih problema zahtijeva kontinuiranu podršku i održavanje te primjenu najboljih praksi u razvoju i upravljanju aplikacijom.

Uz sve navedene tehnologije i metodologije, osigurano je da mobilna aplikacija bude stabilna, sigurna i jednostavna za korištenje. Kroz kontinuiranu podršku i nadogradnje, ova bi aplikacija mogla pružiti korisnicima visokokvalitetno iskustvo komunikacije. Ovaj rad ne samo da pruža dublje razumijevanje procesa razvoja mobilnih aplikacija, već i postavlja temelje za buduće istraživanje i unaprjeđenje u području komunikacijskih tehnologija.

Budući razvoj ove mobilne aplikacije može uključivati integraciju novih tehnologija i pristupa kako bi se poboljšala funkcionalnost i korisničko iskustvo. Kontinuirano praćenje trendova i povratnih informacija korisnika omogućit će prilagodbu i unaprjeđenje aplikacije kako bi se zadovoljile promjenjive potrebe i očekivanja korisnika. Kroz ovo istraživanje i implementaciju, postavljeni su temelji za daljnji razvoj mobilnih aplikacija i poboljšanje načina komunikacije u digitalnom dobu.

Uzimajući sve navedeno u obzir, možemo zaključiti da je razvoj mobilne aplikacije za razmjenu poruka putem *REST API*-a složen proces koji zahtijeva disciplinarni pristup, ali i pravilno planiranje. Implementacija i održavanje mogu rezultirati visokokvalitetnom i korisnički orijentiranom aplikacijom koja će poboljšati način komunikacije među korisnicima. Istraživanje je obuhvatilo i analizu korisničkog iskustva kako bi se osiguralo da su dizajn i

funkcionalnosti mobilne aplikacije intuitivni i privlačni korisnicima. Ovo je važan korak u osiguravanju uspješne interakcije između korisnika i aplikacije, što doprinosi povećanju angažmana i zadovoljstva korisnika.

Detaljno su istražene i implementirane sigurnosne mjere kako bi se zaštitili podaci korisnika i osigurala privatnost tijekom razmjene poruka putem aplikacije. To je ključni aspekt, posebno u kontekstu osjetljivih informacija koje se mogu razmjenjivati putem mobilnih aplikacija.

Tijekom razvoja, posebna pažnja posvećena je i optimizaciji performansi kako mobilne aplikacije tako i pozadinskog sustava. Uključuje prilagodbu resursa, upravljanje memorijom i brzinu odziva kako bi se osiguralo glatko iskustvo korištenja aplikacije bez zastoja ili kašnjenja. Jedan od ključnih ciljeva ovog istraživanja bio je i osiguravanje skalabilnosti aplikacije, omogućavajući joj da se prilagodi rastućem broju korisnika i zahtjeva. To je postignuto korištenjem prilagodljive arhitekture i tehnologija.

Kontinuirano testiranje i nadogradnje aplikacije su neizostavan dio procesa razvoja kako bi se otkrili i riješili eventualne greške (engl. *bug*) ili nedostaci te osigurala kompatibilnost s različitim platformama i uređajima.

LITERATURA

1. Android Notifications, https://developer.android.com/develop/ui/views/notifications (pristupljeno 10. 3. 2024.)
2. Database Migrations with Flyway, Bealdung, https://www.baeldung.com/database-migrations-with-flyway (pristupljeno 12. 3. 2024.)
3. Drawio, draw.io, https://app.diagrams.net/ (pristupljeno 23. 3. 2024.)
4. Firebase for Android, https://firebase.google.com/docs/android/setup (pristupljeno 11.3.2024.)
5. Iuliana Cosmina, Rob Harrop, Chris Schaefer, Clarence Ho - Pro Spring 5_ An In-Depth Guide to the Spring Framework and Its Tools-Apress (2017), Introducing JPA 2.1, pp. 416, 399 – 405
6. Spring Boot (2023), https://spring.io/guides/gs/spring-boot (pristupljeno 10. 3. 2024.)
7. Square, Inc. (2022). Retrofit, https://square.github.io/retrofit/ (pristupljeno 10. 3. 2024.)
8. Using Firebase Cloud, https://www.baeldung.com/spring-fcm (pristupljeno 11.3.2024)
9. Wikipedia - Hexagonal architecture (software), https://en.wikipedia.org/wiki/Hexagonal_architecture_(software) (pristupljeno 10.3.2024.)

SAŽETAK

Završni rad istražuje implementaciju kôda u mobilnoj aplikaciji i pozadinskom sustavu, s fokusom na Android platformu. Provodi se detaljna analiza performansi i funkcionalnosti, uz korištenje tehnologija poput *Firebase*-a za pozadinski sustav i android infrastrukturu te *Spring Boot*-a za razvoj poslužiteljskog dijela. Rad također obuhvaća analizu arhitekture sustava kako bi se istaknula međusobna interakcija između mobilne aplikacije i pozadinskog sustava.

Ključne riječi: Android, Mobilna aplikacija, Pozadinski sustav, Arhitektura, *Firebase*, *Spring Boot*.

SUMMARY

This paper explores the implementation of code in a mobile application and backend, with a focus on the Android platform. A detailed analysis of performance and functionality is carried out, with the use of technologies such as Firebase for backend and android infrastructure and Spring Boot for development of the server part. The paper also includes an analysis of the system architecture in order to highlight the mutual interaction between the mobile application and the backend system.

Keywords: Android, Mobile application, Backend, Arhitecture, Firebase, Spring Boot.