Višeplatformska mobilna aplikacija za potporu udomljivanju kućnih ljubimaca korištenjem prilagodljive programske arhitekture i kolaborativnog filtriranja

Odobašić, Tomislav

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:727297

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-21**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni diplomski studij

Višeplatformska mobilna aplikacija za potporu udomljivanju kućnih ljubimaca korištenjem prilagodljive programske arhitekture i kolaborativnog filtriranja

Diplomski rad

Tomislav Odobašić

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za ocjenu diplomskog rada na sveučilišnom diplomskom studiju

Ocjena diplomskog rada na sveučilišnom diplomskom studiju

Ime i prezime pristupnika:	Tomislav Odobašić
Studij, smjer:	Sveučilišni diplomski studij Elektrotehnika, Komunikacije i
Mat. br. pristupnika, god.	D-1375, 07.10.2021.
JMBAG:	0165073588
Mentor:	prof. dr. sc. Goran Martinović
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 1:	prof. dr. sc. Goran Martinović
Član Povjerenstva 2:	izv. prof. dr. sc. Josip Balen
Naslov diplomskog rada:	Višeplatformska mobilna aplikacija za potporu udomljivanju kućnih ljubimaca korištenjem prilagodljive programske arhitekture i kolaborativnog filtriranja
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskom dijelu diplomskog rada treba opisati probleme i izazove udomljivanja kućnih ljubimaca, te analizirati mogućnosti višeplatformskog razvoja, korištenja prilagodljivih programskih arhitektura i sustava preporuka zasnovanog na kolaborativnom, ali i drugim oblicima filtriranja. Na temelju analize postojećih rješenja za potporu udomljivanju kućnih ljubimaca, te mogućnosti višeplatformskog razvoja i prilagodljive arhitekture, treba definirati funkcionalne i nefunkcionalne zahtjeve na mobilnu aplikaciju, predložiti arhitekturu aplikacije, te dizajn i način prikaza navedenih

Datum ocjene pismenog dijela diplomskog rada od strane mentora:	13.09.2024.
Ocjena pismenog dijela diplomskog rada od strane mentora:	Izvrstan (5)
Datum obrane diplomskog rada:	26.09.2024.
Ocjena usmenog dijela diplomskog rada (obrane):	Izvrstan (5)
Ukupna ocjena diplomskog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije diplomskog rada čime je pristupnik završio sveučilišni diplomski studij:	01.10.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 01.10.2024.

Ime i prezime Pristupnika:	Tomislav Odobašić
Studij:	Sveučilišni diplomski studij Elektrotehnika, Komunikacije i informatika
Mat. br. Pristupnika, godina upisa:	D-1375, 07.10.2021.
Turnitin podudaranje [%]:	5

Ovom izjavom izjavljujem da je rad pod nazivom**: Višeplatformska mobilna aplikacija za potporu udomljivanju kućnih ljubimaca korištenjem prilagodljive programske arhitekture i kolaborativnog filtriranja**

izrađen pod vodstvom mentora prof. dr. sc. Goran Martinović

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD **1**

2. PREGLED STANJA U PODRUČJU I IZAZOVI U UDOMLJIVANJU KUĆNIH LJUBIMACA **2**

2.1. Izazovi i problemi udomljivanja kućnih ljubimaca **2**

2.2. Pregled stanja u području višeplatformskog razvoja, prilagodljivih programskih arhitektura i filtriranja podataka **3**

2.3. Pregled sličnih aplikacija **4**

2.3.1. Aplikacija Petfinder 4

2.3.2. Aplikacija PawBoost 4

2.3.3. Aplikacija 11pets 5

2.3.4. Aplikacija WoofTrax 6

3. MODEL I ARHITEKTURA VIŠEPLATFORMSKE MOBILNE APLIKACIJE **7**

3.1. Mogućnosti višeplatformskog razvoja **7**

3.2. Korištenje prilagodljivih programskih arhitektura **8**

3.3. Sustavi preporuka zasnovani na kolaborativnom i drugim oblicima filtriranja **8**

3.3.1. Sustavi preporuka zasnovani na filtriranju na temelju sadržaja 9

3.3.2. Sustavi preporuka zasnovani na kolaborativnom filtriranju 9

3.3.3. Hibridni sustavi preporuka 12

3.4. Funkcionalni zahtjevi **13**

3.5. Nefunkcionalni zahtjevi **13**

3.6. Arhitektura višeplatformske aplikacije **14**

4. PROGRAMSKO RJEŠENJE VIŠEPLATFORMSKE APLIKACIJE **18**

4.1. Korištene tehnologije i alati **18**

4.1.1. Programski okvir Flutter 18

4.1.2. Programski jezik Dart 18

4.1.3. Firebase 19

4.1.4. Razvojno okruženje Visual Studio Code (VSCode) 19

4.2. Prikaz programskog rješenja po komponentama **19**

4.2.1. Programska implementacija prijave korisnika 19

4.2.2. Programska implementacija registriranja korisnika 21

4.2.3. Programska implementacija upravljanja korisničkim profilima 22

4.2.4. Programska implementacija kreiranja objava životinja za udomljivanje 25
4.2.5. Programska implementacija pregleda i pretrage životinja 27
4.2.6. Programska implementacija stvaranja zahtjeva za udomljivanjem 28
4.2.7. Programska implementacija upravljanja zahtjevima 30
4.2.8. Programska implementacija geolokacije 32
4.2.9. Programska implementacija komunikacije među korisnicima 36
4.2.10. Programska implementacija sustava preporuka 37
4.2.11. Programska implementacija odjave korisnika 42

5. NAČIN KORIŠTENJA I ISPITIVANJE VIŠEPLATFORMSKE APLIKACIJE S ANALIZOM REZULTATA 43

5.1. Način korištenja višeplatformske aplikacije 43

5.2. Prikaz rada višeplatformske aplikacije 44

5.3. Ispitivanje rada višeplatformske aplikacije 51

5.3.1. Ispitivanje višeplatformske aplikacije s gledišta sustava preporuka 52
5.3.2. Ispitivanje višeplatformske aplikacije s gledišta korisničkog iskustva 56
5.3.3. Ispitivanje višeplatformske aplikacije s gledišta programske arhitekture 59

5.4. Analiziranje rezultata ispitivanja višeplatformske aplikacije 61

5.4.1. Analiza ispitivanja sustava preporuka 61
5.4.2. Analiza ispitivanja korisničkog iskustva 62
5.4.3. Analiza ispitivanja programske arhitekture 63

ZAKLJUČAK 65

LITERATURA 66

SAŽETAK 70

ABSTRACT 71

ŽIVOTOPIS 72

PRILOZI 73

1. UVOD

Digitalna transformacija značajno mijenja način na koji se rješavaju svakodnevni problemi, uključujući i proces udomljivanja kućnih ljubimaca. Uobičajeni načini udomljivanja ljubimaca često su spori i neučinkoviti, jer se sam proces suočava s nizom izazova kao što su neučinkovita komunikacija skloništa i potencijalnih udomitelja. Korištenje mobilnih aplikacija u ovom području omogućuje bržu i učinkovitiju komunikaciju između potencijalnih udomitelja i skloništa za životinje. Razvoj mobilnih aplikacija nije lak proces i postavlja mnoge izazove, posebice kada je cilj izraditi rješenja koja su prilagodljiva različitim platformama. Višeplatformski razvoj nudi rješenje za ove izazove, jer s jednom kodnom bazom omogućuje razvoj aplikacija za više operacijskih sustava. Prilagodljive programske arhitekture osiguravaju fleksibilnost, skalabilnost i dugoročne održivosti aplikacije, što je ključno prilikom izrade višeplatformskih rješenja, dok korištenje sustava preporuka korisnicima omogućuje personalizirano korisničkog iskustvo.

Ovaj diplomski rad bavi se razvojem višeplatformske mobilne aplikacije koja olakšava proces udomljivanja kućnih ljubimaca, koristeći prilagodljivu programsku arhitekturu i sustav preporuka temeljen na kolaborativnom filtriranju. Cilj aplikacije je olakšati korisnicima pronalazak odgovarajućeg ljubimca, uzimajući u obzir njihove preferencije i uvjete udomljivanja, čime se poboljšava uspješnost i učinkovitost cijelog procesa, te uz ostvarenje ovog cilja pružiti i zadovoljavajuće korisničko iskustvo.

U drugom poglavlju opisani su izazovi i problemi trenutnog stanja udomljivanja kućnih ljubimaca, te je dan pregled primjera sličnih mobilnih aplikacija. Treće poglavlja daje uvid u mogućnosti višeplatformskog razvoja, prednosti korištenja prilagodljive programske arhitekture, te opisuje funkcionalne i nefunkcionalne zahtjeve aplikacije. U četvrtom poglavlju opisane su tehnologije korištene prilikom izrade aplikacije, te implementacija glavnih funkcionalnosti aplikacije. U petom poglavlju objašnjen je način korištenja i prikazan rad aplikacije. Također, provedeno je ispitivanje rada aplikacije, te obavljena analiza rezultata ispitivanja.

2. PREGLED STANJA U PODRUČJU I IZAZOVI U UDOMLJIVANJU KUĆNIH LJUBIMACA

U ovom poglavlju prikazan je pregled stanja u području višeplatformskog razvoja, prilagodljivih programskih arhitektura i sustava preporuka, te su opisani opći izazovi i problemi udomljivanja kućnih ljubimaca.

2.1. Izazovi i problemi udomljivanja kućnih ljubimaca

Neka su skloništa za životinje prekrasna mjesta, dok su druga odvratna deponija. Njihov odnos prema životinjama, namjena i kapaciteti uvelike se razlikuju. Mogu ih voditi vlada, lokalna humana društva ili privatni pojedinci. Neki se financiraju samo donacijama, dok drugi dobivaju poreznu potporu. Ponekad novac od poreza dolazi s uvjetom da se neke životinje moraju predati eksperimentatorima. Treba uložiti sve napore kako bi se eliminirala ova politika. Neka skloništa primaju samo pse ili samo mačke, ali većina prima oboje, kao i manje životinje poput zečeva, hrčaka i egzotičnih ptica. Neki se mogu ispravno nositi s divljim životinjama, a drugi upućuju hitne slučajeve divljih životinja prirodoslovcima ili rehabilitatorima divljih životinja. Ako je objekt za divlje životinje u blizini, on obrađuje sve dolazne divlje životinje. Zbog oštrih prostornih ograničenja, većina skloništa eutanazira stare, teško bolesne ili agresivne životinje, kao i one koje nakon određenog broja dana nisu zatražene ili nisu udomljene.

Jedan od velikih problema za azile i skloništa su prenapučenost skloništa do kojeg dolazi kada broj životinja koje čekaju na udomljenje premaši kapacitete skloništa [1], što dovodi do niza drugih problema, uključujući smanjenje kvalitete njege, povećan stres za životinje, i naposljetku otežano upravljanje resursima. Osim toga, nedostatak svijesti među potencijalnim udomiteljima o prednostima udomljivanja kućnih ljubimaca u odnosu na kupovinu dodatno smanjuje šanse za pronalazak domova za ove životinje.

Kućni ljubimci koji dolaze iz loših uvjeta često pate od socijalnih i psiholoških problema, što dodatno otežava njihovu prilagodbu u novim domovima. Starije i bolesne životinje suočavaju se s dodatnim izazovima, jer potencijalni udomitelji strahuju od visokih troškova liječenja. Što se tiče pasa, također, postoji stigma prema mješancima, jer ljudi često preferiraju čistokrvne pse, što dodatno smanjuje šanse za udomljenje mješanaca. Ovi izazovi zahtijevaju koordinirane napore kako bi se poboljšala svijest javnosti, pružila podrška udomiteljima i kako bi se promoviralo udomljivanje kao najbolji izbor za pružanje ljubavi i brige napuštenim i odbačenim životinjama.

2.2. Pregled stanja u području višeplatformskog razvoja, prilagodljivih programskih arhitektura i filtriranja podataka

Višeplatformski razvoj mobilnih aplikacija omogućuje izgradnju aplikacija koje mogu raditi na više operativnih sustava koristeći jednu kodnu bazu. Tri glavne tehnologije za razvoj mobilnih aplikacija su nativne, web i višeplatformske aplikacije. Nativne aplikacije pružaju najbolje performanse jer su specifično razvijene za jednu platformu, npr. Android ili iOS koristeći se alatima poput *Android Studia* i *Xcodea*. Web aplikacije su najjeftinije i najlakše za razvoj, i razvijaju se pomoću alata kao što su HTML, CSS i *JavaScript*. Iako dostupne s bilo kojeg web preglednika, web aplikacije obično ne nude istu razinu performansi kao nativne i višeplatformske aplikacije [28]. Višeplatformske aplikacije predstavljaju kompromis između troškova razvoja i performansi. Kombiniraju značajke web i nativnih aplikacija, te na taj način omogućuju pristup nativnim funkcijama dok koriste jednu kodnu bazu. *React Native,* razvijen od strane *Facebooka* 2015. godine, omogućuje razvoj aplikacija za iOS i Android koristeći kod napisan u *JavaScriptu*. *Flutter* također omogućava jedinstveni kod za sve platforme, kao i *Xamarin* koji koristi C# i .NET, te pruža pristup nativnim API-ima.

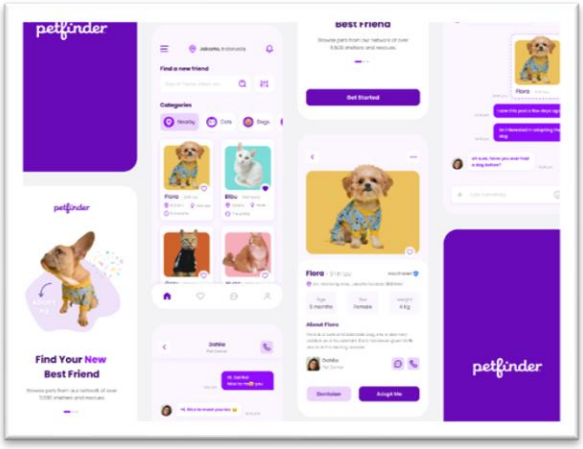
Tradicionalni pristupi dizajniranju programskog rješenja često uključuju definiranje svih uslužnih komponenti unaprijed, temeljeni na iskustvu dizajnera, što može znatno ograničiti fleksibilnost sustava i ponovnu upotrebu komponenti. Korištenje ovakvih metoda može otežati prilagodbu i proširenje aplikacija jer se komponente razvijaju prije nego što sustav ima potrebu koristiti ih [10]. Suprotnost prethodno opisanim pristupima čine prilagodljive programske arhitekture koje čine ključnu ulogu u osiguravanju modularnosti, fleksibilnost i jednostavnosti proširenja aplikacije. Prilagodljive programske arhitekture omogućuju brzu modifikaciju i prilagodbu koda bez potrebe za velikim i značajnim promjenama u postojećem kodu, što omogućuje brzo reagiranje na promjene potreba korisnika i tehnoloških zahtjeva.

Sustavi preporuka se u današnje vrijeme široko koriste u *on-line* aplikacijama, gdje pomažu korisnicima pronaći relevantne informacije u situacijama gdje su preplavljeni informacija [34]. Sustavi preporuka vodećih tvrtki kao što su Amazon, Google, Rakuten, TikTok vrlo su sofisticirani modeli koje pokreće napredno strojno učenje, i sve kvalitetnije omogućuju personalizirano iskustvo korisnika na mreži na način da vide, čuju, te kupuju stvari za koje se čini da su odabrane baš za određenog korisnika. Bez obzira preporučuju li proizvode, ponude ili neki drug sadržaj, svi sustavi preporuka u konačnici određuju što korisnika čini više ili manje kompatibilnim s artiklom ili dijelom sadržaja [35].

2.3. Pregled sličnih aplikacija

2.3.1. Aplikacija Petfinder

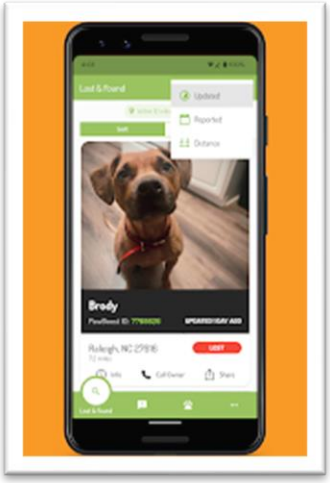
Aplikacija Petfinder pruža uslugu udomljivanja kućnih ljubimaca upravljajući web i mobilnom aplikacijom. Aplikacija je osmišljena kako bi pomogla korisnicima u pronalaženju kućnih ljubimaca koje je moguće usvojiti u njihovom lokalnom području i pruža niz značajki koje proces pretraživanja čine lakšim i učinkovitijim [2]. Aplikacija Petfinder nudi nekoliko ključnih značajki koje poboljšavaju iskustvo udomljivanja kućnih ljubimaca. Korisnici imaju opcije korištenja filtera za pretraživanja za pronalaženje ljubimaca prema različitim kriterijima poput lokacije, pasmine, dobi i veličine. Ljubimci imaju profile koji prikazuju fotografiju, informacije o temperamentu, zdravlju i povijesti životinje. Na slici 2.1 nalazi se prikaz aplikacije Petfinder.



Sl. 2.1 Aplikacija Petfinder [2]

2.3.2. Aplikacija PawBoost

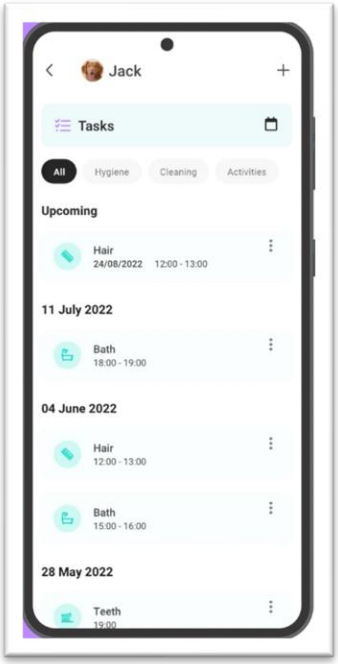
Aplikacija PawBoost usmjerena je prvenstveno na ponovno spajanje izgubljenih ljubimaca s njihovim vlasnicima, ali također uključuje i značajke za udomljivanje kućnih ljubimaca. Aplikacija koristi pristup temeljen na zajednici kako bi pomogla locirati izgubljene ljubimce i pronaći domove za one koje je moguće udomiti [3]. Aplikacija PawBoost nudi par ključnih značajki za pomoć u pronalaženju izgubljenih ljubimaca, kao što su objavljivanje upozorenja za izgubljene ljubimce, pretraživanje baze podataka izgubljenih i pronađenih ljubimaca. Korisnici koji pronađu ljubimca mogu izraditi izvješće o pronađenom ljubimcu. Aplikacija PawBoost dostupna je na iOS i Android platformi, što ju čini pristupačnom širokom spektru korisnika. Na slici 2.2 nalazi se prikaz aplikacije PawBoost.



Sl. 2.2 Aplikacija PawBoost [3]

2.3.3. Aplikacija 11pets

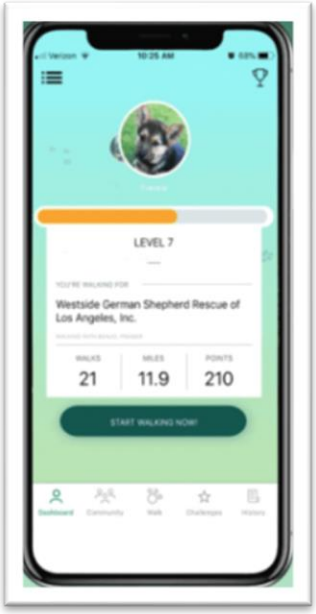
Aplikacija 11pets služi za upravljanje njegom kućnih ljubimaca dizajnirana za podršku vlasnicima kućnih ljubimaca u pogledima brige o njima, uključujući upravljanje zdravljem, njegovanje, i udomljivanje kućnih ljubimaca. Korisnici mogu pohranjivati zdravstvene zapise, pratiti povijest zdravlja, te primati podsjetnike važnih zadataka poput cijepljenja i veterinarskih pregleda. Aplikacija također uključuje planer dotjerivanja za zadatke poput kupanja i rezanja noktiju, aplikacija dodatno nudi i savjete za njegu ljubimca [4]. Aplikacija pruža i mogućnost udomljivanja kućnih ljubimaca putem popisa ljubimaca dostupnih za udomljivanje iz skloništa i spasilačkih organizacija. Na slici 2.3 prikazan je izgled aplikacije 11pets.



Sl. 2.3 Aplikacija 11pets [4]

2.3.4. Aplikacija WoofTrax

Aplikacija WoofTrax jedinstvena je aplikacija osmišljena za promoviranje zdravlja kućnih ljubimaca. Aplikacija potiče šetanje pasa pretvarajući šetnje u donacije lokalnim skloništima za životinje. Aplikacija podržava funkcionalnost praćenja šetnji s psom bilježeći trajanje, udaljenost i rutu svake šetnje. Korisnici također imaju mogućnost doniranja dodatnih sredstava skloništima za životinje ili organizacijama za spašavanje. Omogućen je pregled i prikaz ploče s najboljim rezultatima, gdje korisnici mogu pregledavati vlastite statistike hodanja, te ih uspoređivati s rezultatima drugih korisnika. Na slici 2.4 nalazi se prikaz aplikacije WoofTrax.



Sl. 2.4 Aplikacija WoofTrax [33]

3. MODEL I ARHITEKTURA VIŠEPLATFORMSKE MOBILNE APLIKACIJE

U ovom poglavlju opisane su mogućnosti višeplatformskog razvoja, prilagodljivih programskih arhitektura i sustava preporuka zasnovanim na kolaborativnom ali i drugim oblicima filtriranja, definirani su funkcionalni i nefunkcionalni zahtjevi, te je opisana arhitektura aplikacije.

3.1. Mogućnosti višeplatformskog razvoja

Globalni rast mobilnog svijeta odavno je premašio udio razvoja stolnih računala. Tehnološke inovacije u mobilnom računalstvu, zajedno s povećanim mogućnostima mobilnih uređaja, dramatičnim poboljšanjem upotrebljivosti i omjera izgleda i dojma prijenosnih uređaja, zajedno s njihovim sve nižim cijenama, rezultirale su golemim brojem mobilnih uređaja na tržištu. Jednostavan pristup ovoj tehnologiji i brz rast broja kupljenih mobilnih uređaja rezultirali su velikom potražnjom za mobilnim aplikacijama [5]. Poželjni zahtjevi bilo kojeg višeplatformskog okvira identificirani su kako je navedeno u nastavku: podrška za više mobilnih platformi, bogato korisničko sučelje, pozadinska komunikacija, sigurnost, podrška za proširenja aplikacije, potrošnja energije, pristup ugrađenim značajkama, te otvoreni kod. Koristeći programski okvir *Flutter* može se napisati jedna kodna baza koja podjednako dobro radi na Android sustavu, kao i na iOS-u. Za razliku od izvorne aplikacije koja zadovoljava zahtjeve određene operacije sustava i njegovog SDK-a kao što su Android ili iOS, *Flutter* aplikacija je kompatibilna s više operativnih sustava i može raditi i na Android i iOS pametnim telefonima i tabletima kao što je objašnjeno u [6] i [7]. Ključne mogućnosti višeplatformskog razvoja navedene su u nastavku:

- Jednostavnija održivost, s obzirom na to da se koristi jedna kodna baza, održavanje i ažuriranje aplikacije postaje jednostavnije
- Brži razvoj, koristi se jedan kod za više platformi što znatno smanjuje vrijeme razvoja jer se izbjegava potreba za pisanjem zasebnog koda za svaku platformu
- Smanjenje troškova, uzimajući u obzir da se jedan tim bavi razvojem koda za sve platforme, troškovi razvoja su niži
- Konzistentno korisničko iskustvo, višeplatformski alati omogućuju konzistentan dizajn i funkcionalnost na različitim uređajima
- Upotreba raznih alata i okvira, postoji mnogo alata poput *Fluttera, React Nativea, Xamarina* i drugih koji omogućuju razvoj višeplatformskih aplikacija s različitim prednostima

Međutim, u razvoju višeplatformskog koda postoje i mnogi izazovi, kao što su moguća ograničenja performansi i prilagodbi koje su specifične za svaku platformu. Zato je ključno odabrati pravu tehnologiju i strategiju razvoja koja odgovara specifičnim potrebama aplikacije [8].

3.2. Korištenje prilagodljivih programskih arhitektura

Svijet se razvija, a potrebe korisnika brzo se mijenjaju s vremenom. Brz odgovor na trenutne zahtjeve korisnika kritičan je čimbenik uspjeha bilo koje organizacije koja razvija programska rješenja. Prilagodljiva arhitektura uključuje mehanizme koji mogu promijeniti svoje ponašanje kako bi izašli u susret korisnicima i njihovim zahtjevima [9]. Prilagodljive programske arhitekture omogućuju razvoj aplikacija koje su lako prilagodljive promjenama u zahtjevima, skaliranju i tehnologiji.

Osnovni pristupi u prilagodljivim arhitekturama su modularnost, što omogućuje zamjenu, dodavanje ili uklanjanje modula bez utjecaja na cijelu aplikaciju, korištenje dizajnerskih obrazaca koji omogućuju fleksibilnost u implementaciji različitih funkcionalnosti i promjenama zahtjeva, skalabilnost, arhitekture se mogu prilagoditi povećanju opterećenja ili dodavanju novih funkcionalnosti bez značajnog utjecaja na performanse, te integracija sa suvremenim alatima i tehnologijama [10]. Korištenje prilagodljivih arhitektura ključno je za održavanje dugoročnih aplikacija, jer kao što je i prethodno opisano omogućava brzu reakciju na promjene u tehnologiji, tržištu ili potrebama korisnika, uz minimalne troškove i vrijeme.

3.3. Sustavi preporuka zasnovani na kolaborativnom i drugim oblicima filtriranja

Sustavi preporuka su algoritmi koji daju personalizirane prijedloge za stavke koje su najrelevantnije za svakog korisnika. S velikim porastom dostupnih online sadržaja, korisnici su preplavljeni izborom i informacijama. Zbog toga je ključno da platforme ponude preporuke artikala svakom korisniku, kako bi se potencijalno povećalo zadovoljstvo korisnika. Sustavi za preporuku drže korisnike zainteresiranima za ono što stranica i dalje preporučuje [11].

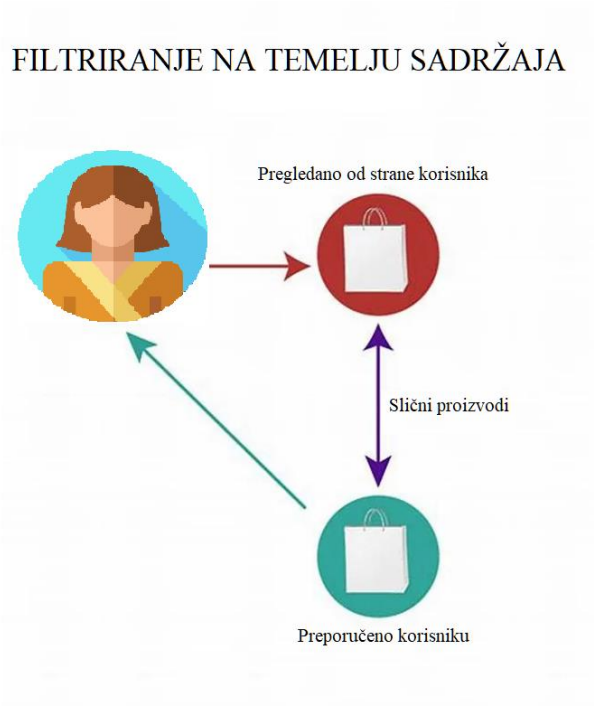
Mehanizmi za preporuke pružaju personalizirano korisničko iskustvo, pomažući svakom pojedinom potrošaču da identificira i otkrije svoje omiljene filmove, TV emisije, digitalne proizvode, knjige, članke itd. Ovakvi sustavi pomažu poduzećima da povećaju prodaju i donose korist potrošačima. Uz sustave preporuka, potrošači mogu lako pronaći proizvode, promovirati jednostavnost korištenja i natjerati potrošače da nastave koristiti platformu [12]. Sustav preporuka je mehanizam za filtriranje podataka koji koristi koncepte dubokog učenja i algoritme za predlaganje potencijalnih proizvoda ovisno o prethodnim preferencijama korisnika ili

sekundarnom filtriranju. Koncept iza ovakvih algoritama je pronalaženje obrazaca u ponašanju potrošača ili sličnog potrošača prema usluzi ili proizvodu. Metoda prikupljanja podataka uvelike se razlikuje ovisno o vrsti prodanih proizvoda ili usluga [13]. Sustavi preporuka općenito se mogu podijeliti na tri glavne vrste: filtriranje na temelju sadržaja, kolaborativno filtriranje i hibridni sustavi preporuka.

3.3.1. Sustavi preporuka zasnovani na filtriranju na temelju sadržaja

Filtriranje na temelju sadržaja preporučuje stavke slične onima koje su se korisniku sviđale u prošlosti. Prema [29], na slici 3.2 prikazan je proces stvaranja preporuke sustava preporuka temeljenog na filtriranju na temelju sadržaja . Ovaj se pristup usredotočuje na karakteristike samih predmeta. Glavne značajke sustava preporuka temeljenog na filtriranju na temelju sadržaja opisane su u nastavku:

1. Značajke stavke – preporuke se temelje na značajkama ili atributima stavki, što može biti npr. žanr, autor, opis za knjige ili filmove
2. Korisnički profili – korisnički profil izrađuje se na temelju stavki s kojima su bili u interakciji, uzimajući u obzir značajke tih stavki [14]



Sl. 3.2 Filtriranje na temelju sadržaja [29]

3.3.2. Sustavi preporuka zasnovani na kolaborativnom filtriranju

Kolaborativno filtriranje je metoda koju koriste sustavi preporuka kako bi korisnicima predložili stavke na temelju preferencija i ponašanja drugih korisnika. Iako se ovakvi sustavi preporuka mogu razlikovati prema tehnikama, naposljetku se svi koriste za generiranje preporuka [15]. Ovaj

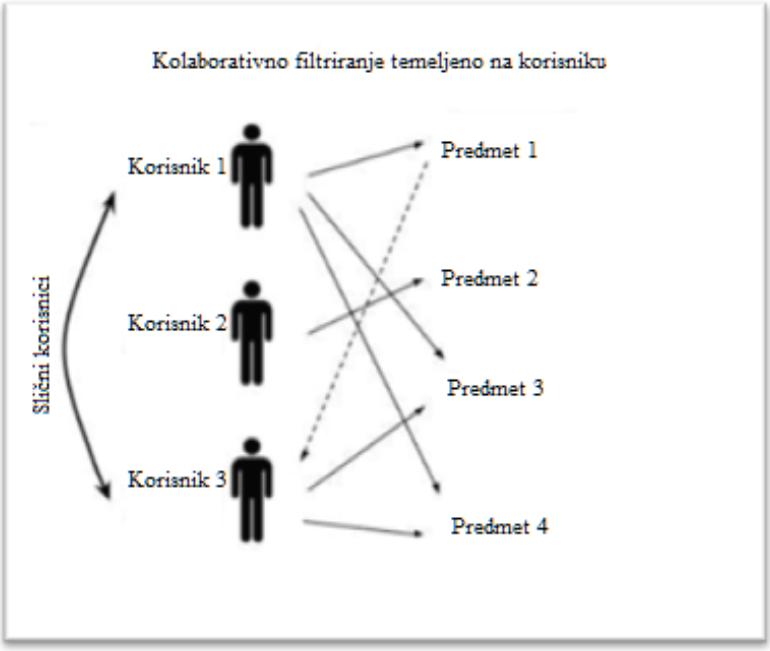
pristup koristi kolektivnu inteligenciju grupe korisnika za davanje preporuka. Kolaborativno filtriranje može biti temeljeno na korisniku, i temeljeno na predmetu. Ključni koncepti kolaborativnog filtriranja opisani su u nastavku:

1. Matrica interakcije korisnik-predmet – ključna komponenta kolaborativnog filtriranje je matrica interakcije između korisnika i stavki, gdje redovi predstavljaju korisnike, stupci predstavljaju stavke, a vrijednosti u matrici označavaju interakcije između korisnika i stavki
2. Mjere sličnosti – kolaborativno filtriranje oslanja se na mjerenje sličnosti između korisnika ili stavki. Uobičajene mjere sličnosti uključuju kosinusnu sličnost, Pearsonovu korelaciju i Euklidsku udaljenost

Kolaborativno filtriranje temeljeno na korisniku preporučuje stavke korisniku na temelju stavki koje su se svidjele sličnim korisnicima. Osnovna ideja za predviđanje ocjena korisnika je da koristi ocjene drugih korisnika čije je ocjenjivanje stavki slično prvobitnom korisniku. Kada postoji rijetkost, s nekoliko dostupnih ocjena izračuni sličnosti mogu biti slabi i može doći do nepreciznog računanja predviđanja [16]. Pretpostavlja se da će se korisnici koji su se složili u prošlosti, složiti i u budućnosti. Proces kolaborativnog filtriranja temeljenog na korisniku opisan je u nastavku:

1. Identificiranje sličnih korisnika – izračunavanje sličnosti između ciljanog korisnika i drugih korisnika koristeći mjere sličnosti.
2. Odabiranje susjedstva – odabire se podskup korisnika koji su najsličniji ciljanom korisniku
3. Generiranje preporuke – preporučuju se stavke koje su se svidjele sličnim korisnicima, ali s kojima ciljani korisnik još nije stupio u interakciju

Za primjer se uzima da stavka ili predmet predstavlja film, te ako su korisnik 1 i korisnik 3 slično ocijenili mnoge filmove, a korisniku 1 se svidio film koji korisnik 3 nije još pogledao, taj se film preporučuje korisniku 3, kao što je prema [30] prikazano na slici 3.3.

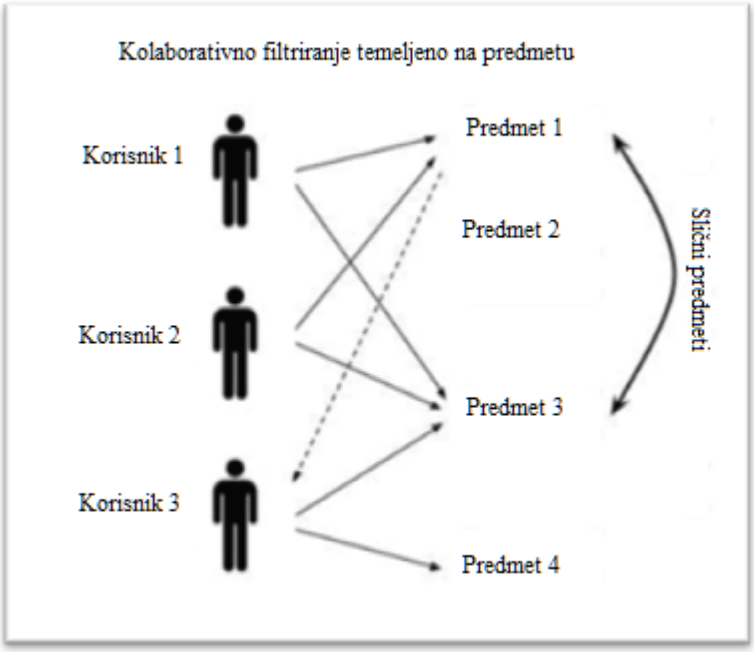


Sl. 3.3 Kolaborativno filtriranje temeljeno na korisniku [30]

Kolaborativno filtriranje temeljeno na predmetu preporučuje predmete na temelju sličnosti među predmetima. Ova tehnika nalikuje na tehniku temeljenu na korisniku, ali tehnika koristi sličnosti između predmeta za odabir susjednih stavki za predviđanje korisničkih preferencija na ciljani predmet. Rezultati kolaborativnog filtriranja na temelju predmeta efektivniji su u predviđanju nego tehnika temeljena na korisniku i često se implementira za sustave preporuka [17]. Pretpostavlja se da će slične stavke isti korisnici ocijeniti na sličan način. Način rada kolaborativnog filtriranja temeljenog na predmetu opisan je u nastavku:

1. Identificiranje slične stavke – izračunavanje sličnosti između stavki na temelju ocjena koje su dali korisnici
2. Odabiranje susjedstva – odabire se podskup stavki koje su najsličnije stavkama s kojima je ciljani korisnik bio u interakciji
3. Generiranje preporuke – preporučuju se stavke koje su slične onima koje su se korisniku svidjele ili s kojima je bio interakciji

Na primjer, ako se korisniku svidjela određena knjiga, algoritam kolaborativnog filtriranja temeljen na predmetu preporučuje druge knjige koje su se svidjele korisnicima kojima se također svidjela originalna knjiga, kao što je prema [30] prikazano na slici 3.4.



Sl. 3.4 Kolaborativno filtriranje temeljeno na predmetu [30]

Kolaborativno filtriranje je moćna i široko korištena tehnika preporuke koja iskorištava kolektivne preferencije korisnika za generiranje personaliziranih preporuka. Identificirajući sličnosti između korisnika ili predmeta, daje smislene prijedloge, iako se također suočava s izazovima kao što su problem hladnog pokretanja, rijetkost i skalabilnost [18]. Različita poboljšanja i hibridni pristupi često se koriste za ublažavanje ovih problema i povećanje učinkovitosti kolaborativnih sustava filtriranja.

3.3.3. Hibridni sustavi preporuka

Hibridni sustavi preporuka kombiniraju više tehnika preporučivanja kako bi se poboljšala učinkovitost i kvaliteta generiranja preporuka, te prevladala ograničenja pojedinačnih prethodno opisanih metoda. Glavne značajke hibridnih sustava preporuka opisane se u nastavku:

1. Kombiniranje pristupa – kombinira metode filtriranja temeljene na sadržaju i metode suradnje
2. Fleksibilnost – može prilagoditi težinu ili doprinos svake metode na temelju konteksta ili potreba korisnika

Hibridni sustav može koristiti kolaborativno filtriranje za identifikaciju sličnih korisnika, a zatim filtriranje temeljeno na sadržaju za preporučivanje stavki na temelju atributa koji su se svidjeli sličnim korisnicima [27].

3.4. Funkcionalni zahtjevi

Funkcionalni zahtjevi aplikacije mogu se definirati na dva načina. Prvi način usmjeren je na funkcije, gdje se funkcionalni zahtjev definira kao neka funkcija koju sustav treba biti sposoban odraditi. Drugi način fokusira se na ponašanje sustava, gdje funkcionalni zahtjevi opisuju i specificiraju ulaze i izlaze iz sustava, te odnose između njih [31]. Neki primjer čestih funkcionalnih zahtjeva mobilnih, ali i web aplikacija su prijava, registracija, odjava korisnika iz aplikacije, upravljanje korisničkim profilom, pretraga i listanje sadržaja koji aplikacija pruža, stvaranje novog sadržaja, obavijesti i notifikacije, transakcije i plaćanja, te komuniciranje između korisnika. Funkcionalni zahtjev definira radnju koju sustav mora moći izvršiti.

Višeplatformska aplikacija za potporu udomljivanju kućnih ljubimaca treba omogućiti funkcionalnost autentifikacije korisnika, odnosno prijavu i registraciju korisnika u aplikaciju. Nakon prijave, korisniku treba biti omogućena funkcionalnost dovršavanja profila, gdje unosi dodatne podatke u odnosu na unesene prilikom registracije. Korisnicima treba biti omogućeno pregledavanje i pretraživanje objava o kućnim ljubimcima za udomljivanje, te označavanje objava ljubimaca kao omiljenih. Korisnicima mora biti omogućeno prikazivanje detalja svake objave životinja za udomljivanje. Aplikacija također treba implementirati funkcionalnost predaje zahtjeva za udomljivanjem, kao i prikazivanje lokacije ljubimca na karti te računanje udaljenosti i generiranje rute do lokacije. Korisnicima mora biti omogućeno upravljanje zahtjevima. Također u aplikaciji treba biti ostvarena funkcionalnost chat sustava koja će korisnicima omogućavati izravnu komunikaciju s drugim korisnicima aplikacije. Aplikacija treba omogućiti i generiranje i prikaz personaliziranih preporuka korisnicima u obliku objava životinja za udomljivanje.

3.5. Nefunkcionalni zahtjevi

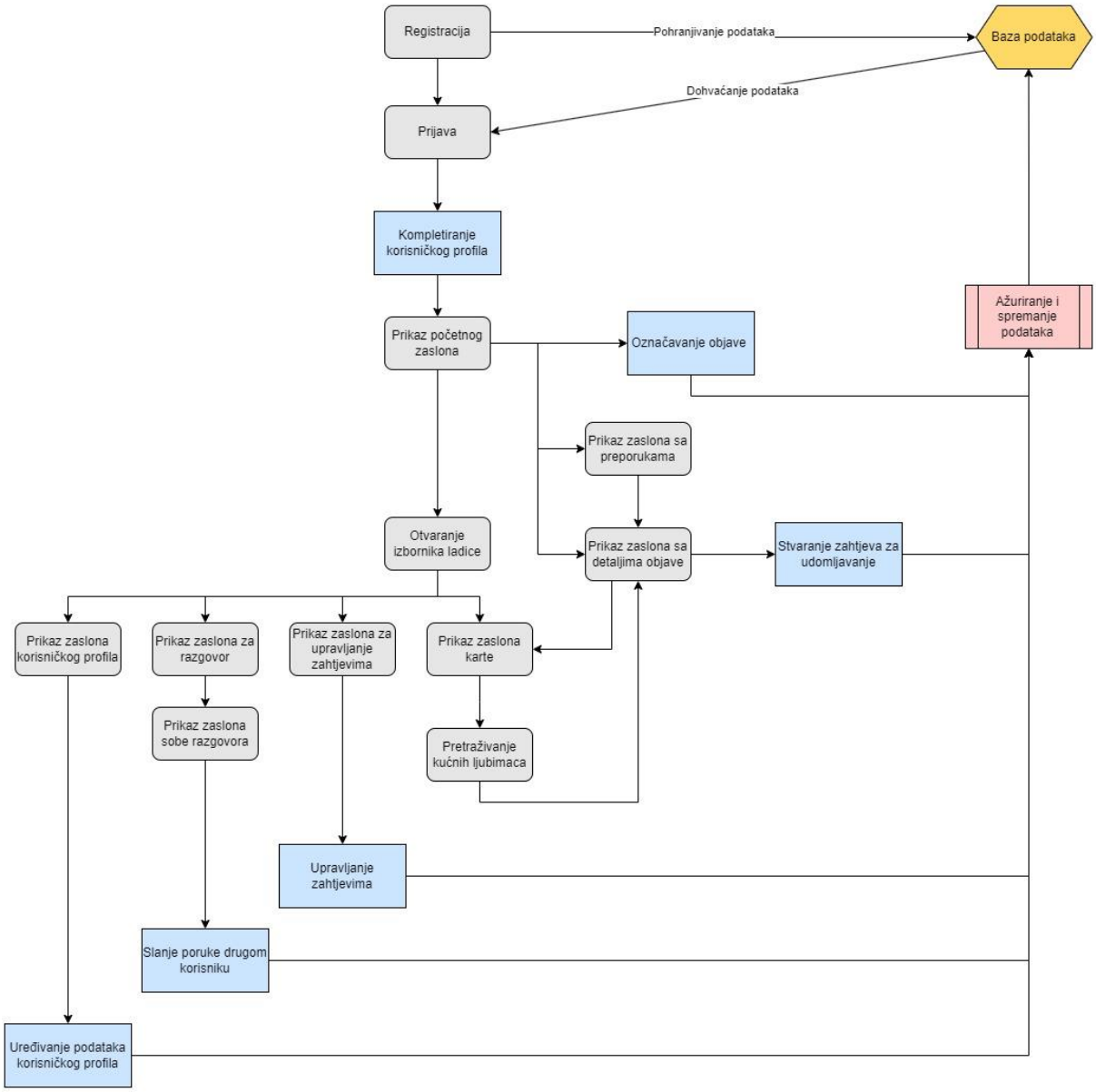
Nefunkcionalni zahtjevi fokusiraju se na ograničenja, ponašanje, kvalitetu i performanse sustava, odnosno specificiraju kako sustav treba raditi. Česti primjeri nefunkcionalnih zahtjeva definiraju i uključuju kvalitetu performanse, sigurnost, pouzdanost, skalabilnost, kompatibilnost, održavanje i vrijeme odziva sustava [32].

Aplikacija mora biti dizajnirana tako da brzo reagira na korisničke zahtjeve, s maksimalnim vremenom odaziva od 3 sekunde prilikom učitavanja podataka ili prikaza novih zaslona. Strujanje podataka, poput chat poruka i ažuriranja statusa, treba se odvijati u stvarnom vremenu uz minimalna kašnjenja. Dizajn aplikacije mora omogućiti skalabilnost, kako bi podržao porast broja korisnika i oglasa. Sigurnost korisničkih podataka mora biti osigurana strogo kontroliranom autentifikacijom i autorizacijom. Korisničko sučelje mora biti intuitivno i jednostavno za

korištenje, s konzistentnim navigacijskim elementima. Aplikacija također mora podržavati integraciju s vanjskim API-ima, poput *Google Maps*, kako bi bilo moguće omogućiti korištenje naprednih lokacijskih usluga.

3.6. Arhitektura višeplatformske aplikacije

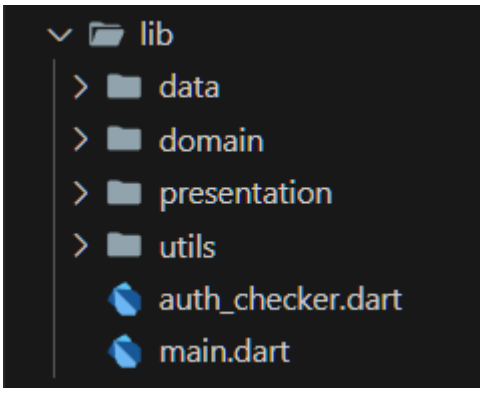
Arhitektura višeplatformske aplikacije prikazana je na slici 3.5. Tijek odvijanja višeplatformske aplikacije je detaljno opisan u četvrtom poglavlju. Nakon registracije i prve prijave, od korisnika se traži da kompletira korisnički profil pružanjem dodatnih informacija. Zatim korisnik dolazi na početni zaslon gdje je moguće pregledavati objave za ljubimce, filtrirati ih prema vrsti ili statusu udomljenja te otvoriti zaslon na kom se nalazi karta ili kreirati novi oglas. Detalji o ljubimcu uključuju informacije o pasmini, dobi, težini, lokaciji i kontakt podatke oglašivača. Korisnik ima mogućnost slati zahtjeve za udomljivanje, pratiti status svojih zahtjeva kroz posebni zaslon, te komunicirati s vlasnicima putem ugrađenog sustava za razgovor. Sustav preporuka, temeljen na favoritima korisnika, nudi slične kućne ljubimce, dok funkcionalnost karte omogućuje pregled lokacija ljubimaca i planiranje ruta. Sve ove funkcionalnosti omogućuju korisnicima jednostavan i učinkovit način za pronalaženje i udomljivanje kućnih ljubimaca.



Sl. 3.5 Arhitektura višeplatformske aplikacije

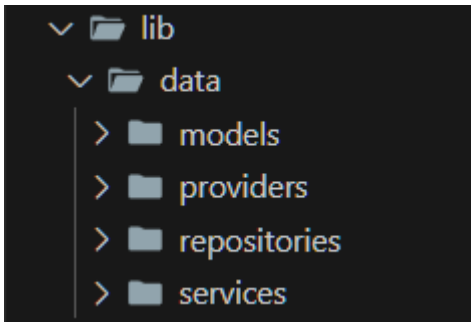
Arhitektura aplikacije organizirana je u jasno definirane slojeve koji odvajaju zadatke i održavaju modularnost. U korijenu projekta nalazi se datoteka *main.dart*, koja služi kao ulazna točka aplikacije. Sadrži logiku potrebnu za pokretanje aplikacije i uz *auth_checker.dart* služi kao početna točka za navigaciju između zaslona i upravljanje autentifikacijom. *Utils* mapa sadrži pomoćne funkcije i konstante koje se mogu koristiti u različitim slojevima diljem aplikacije. U ovoj mapi nalaze se funkcije kao što su funkcije za upravljanje i rukovanje slikama, funkcije za obradu nizova, te funkcije za rad s lokacijama.

Aplikacija je dalje podijeljena u nekoliko glavnih slojeva kao što je prikazano na slici 3.6.



Sl. 3.6 Prikaz arhitekture višeplatformske aplikacije

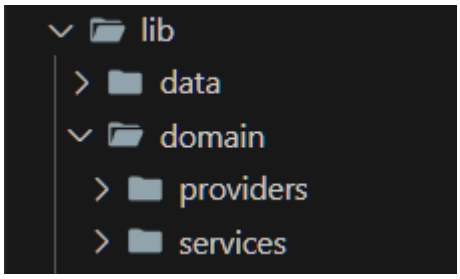
Podatkovni (*data*) sloj odgovoran je za interakciju s vanjskim izvorima podataka i upravljanje protokom podataka kroz aplikaciju. Uključuje repozitorije i usluge koji upravljaju komunikacijom s *Firebase* bazom i drugim API-ima, osiguravajući besprijekornu razmjenu podataka. Ovaj sloj također sadrži modele aplikacije koji definiraju strukturu ključnih entiteta kao što su korisnici, kućni ljubimci, poruke i zahtjevi za udomljivanje. Repozitoriji imaju zadatak upravljati interakcijama s vanjskim izvorima kao što je *Firebase*, dok usluge obavljaju specijalizirane funkcije poput dovršavanja profila. Apstrahiranjem podatkovnih operacija u repozitorije i usluge, sloj *data* učinkovito odvaja poslovnu logiku od složenosti dohvaćanja i pohrane podataka, osiguravajući čišću strukturu koju je lakše održavati. Arhitektura sloja *data* prikazana je na slici 3.7.



Sl. 3.7 Prikaz arhitekture sloja *data* višeplatformske aplikacije

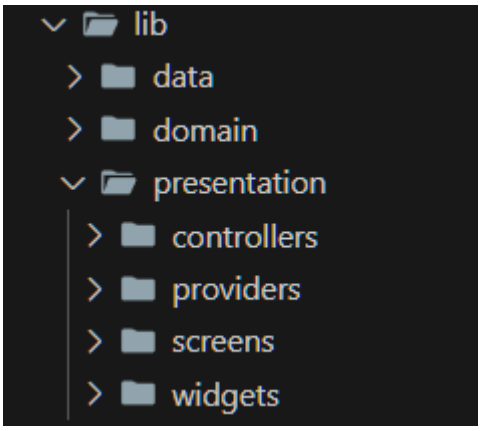
Sloj domene (*domain*) služi kao srednji sloj aplikacije, gdje se operacijama aplikacije upravlja na temelju interakcija korisnika i unosa podataka. Ovaj sloj sadrži klase i *providere* usluga koji definiraju protok podataka i interakciju s različitih dijelova aplikacije. *Provideri* dostavljaju podatke drugim dijelovima aplikacije reaktivno, upravljajući događajima kao što su provjera autentičnosti korisnika i logika preporuka, koja izračunava sličnosti između kućnih ljubimaca na temelju korisničkih preferencija. Usluge u ovom sloju obavljaju zadatke koji obuhvaćaju više repozitorija, kao što je izračunavanje ruta za objave o kućnim ljubimcima. Izoliranjem osnovne

logike od korisničkog sučelja, sloj domene osigurava lakše testiranje, skalabilnost i mogućnost održavanja. Arhitektura sloja *domain* prikazana je na slici 3.8.



Sl. 3.8 Prikaz arhitekture sloja *domain* višeplatformske aplikacije

Prezentacijski sloj (*presentation*) odgovoran je za sve interakcije s kojima se korisnik susreće za vrijeme korištenja aplikacije. Prilikom korištenja aplikacije korisnik se susreće s raznim UI (*user interface*) elementima poput *screenova* i *widgeta* koji su direktno povezani sa stanjem i poslovnom logikom aplikacije. *Screenovi* vizualno predstavljaju zaslone aplikacije i njihov raspored, dok su *widgeti* manje komponente koje mogu biti korištene za višekratnu upotrebu poput gumbova, obrazaca, naslova, tekstova i još mnogih drugih. Kontroleri služe za upravljanje interakcijama između korisničkog sučelja i poslovne logike. Stanjem u aplikaciji se upravlja *Riverpod providerima*. Na ovaj način se dopušta korisničkom sučelju da dinamički reagira na promjene stanja u aplikaciji i osigurava korisnicima najnovije ažurirane informacije i vizualne elemente na temelju trenutnog stanja aplikacije. Arhitektura sloja *presentation* prikazana je na slici 3.9.



Sl. 3.9. Prikaz arhitekture sloja *presentation* višeplatformske aplikacije

4. PROGRAMSKO RJEŠENJE VIŠEPLATFORMSKE APLIKACIJE

U ovom poglavlju opisane su korištene tehnologije za izradu aplikacije, kao i implementacija ključnih funkcionalnosti. Prilikom razvoja aplikacije korišten je android emulator Pixel 5 API 31 za praćenje i testiranje funkcionalnosti, uz očekivanu sličnu funkcionalnost i na ostalim platformama kao što je iOS.

4.1. Korištene tehnologije i alati

4.1.1. Programski okvir Flutter

Programski okvir *Flutter* je skup alata korisničkog sučelja, za izradu nativno kompiliranih aplikacija za mobitele, stolna računala i web s jednom bazom koda, koji je stvorio Google. Prvi put opisan 2015. godine, *Flutter* je objavljen u svibnju 2017. *Flutter* interno koristi Google u aplikacijama kao što su *Google Pay* i *Google Earth*.

Flutter pojednostavljuje proces stvaranja dosljednih, privlačnih korisničkih sučelja za aplikaciju na šest platformi koje podržava. Kodiranje aplikacije za jednu specifičnu platformu, kao što je npr. iOS, naziva se nativni razvoj aplikacija. Nasuprot tome, razvoj aplikacija na više platformi gradi aplikaciju za više platformi s jednom kodnom bazom [19].

U razvoju nativnih aplikacija, kodira se posebno za jednu platformu, što daje potpuni pristup izvornim mogućnostima uređaja. Ovakav pristup obično rezultira superiornom izvedbom i brzinom u usporedbi s razvojem aplikacija na više platformi. Nasuprot tome, pokretanje aplikacije na više platformi zahtijeva više resursa koda i razvojnih programera u izvornom razvoju aplikacija. Uz ove troškove, koordiniranje istovremenog pokretanja na različitim platformama uz održavanje dosljednog korisničkog iskustva može biti izazovno. *Flutter* ima značajke koje razvoj aplikacija na više platformi čine lakšim i visokoučinkovitim [20].

Alati prilagođeni programerima – Google je napravio *Flutter* s naglaskom na jednostavnost upotrebe. S alatima kao što je vruće ponovno učitavanje (engl. *hot reload*) programeri mogu pregledavati kako će promjene koda izgledati bez gubitka stanja. Drugi alati poput widget inspektora olakšavaju vizualizaciju i rješavanje problema s izgledima korisničkog sučelja [21].

4.1.2. Programski jezik Dart

Programski jezik *Dart* su dizajnirali Lars Bak i Kasper Lund, a razvio Google. Može se koristiti za razvoj web i mobilnih aplikacija, kao i poslužiteljskih i desktop aplikacija [22]. *Dart* je objektno orijentiran jezik koji se temelji na klasi sa sintaksom sličnom kao C jezik. Može se prevesti u strojni kod, *JavaScript* ili *WebAssembly*. Podržava sučelja, apstraktne klase, reificirane generike i

zaključivanje tipa. *Dart* je moderan i fleksibilan programski jezik koji kombinira jednostavno sa snažnim značajkama, što ga čini prikladnim za širok raspon aplikacija. Integracija programskog jezika *Dart* s programskim okvirom *Flutter* značajno je povećala njegovu popularnost, budući da programeri mogu iskoristiti mogućnosti *Darta* za učinkovitu izgradnju aplikacija visokih performansi za više platformi. *Dart* je izborno tipiziran, što omogućuje da se prilikom programiranja specificiraju tipovi za varijable kako bi se rano uhvatile pogreške i poboljšala čitljivost koda. Jezik nudi *Just-in-Time* (JIT) i *Ahead-of-Time* (AOT) kompilaciju, što olakšava brze razvojne cikluse s mogućnostima vrućeg ponovnog učitavanja i generiranjem optimiziranog izvornog strojnog koda za proizvodne implementacije. Uz to, *Dart* pojednostavljuje asinkrono programiranje putem ugrađene podrške za *async* i *await*, što je ključno za učinkovito rukovanje operacijama poput I/O bez blokiranja [23].

4.1.3. Firebase

Firebase je platforma za razvoj aplikacija koju pruža Google. Nudi paket alata i usluga temeljenih na oblaku koji su dizajnirani da olakšaju programerima izgradnju aplikacija. *Firebase* uključuje *Realtime Database* za sinkronizaciju podataka u stvarnom vremenu, *Firestore* za složenije strukture podataka, *Firebase* autentifikaciju za prijavljivanje korisnika [24]. *Firebase* također uključuje *Firebase Storage* za pohranu sadržaja poput fotografija, *Firebase Hosting* za brz web hosting, te *Firebase Cloud Messaging* za slanje obavijesti [25].

4.1.4. Razvojno okruženje Visual Studio Code (VSCode)

Visual Studio Code (*VSCode*) je uređivač izvornog koda koji je razvio Microsoft za Windows, macOS, Linux i web preglednike. Značajke uključuju podršku za ispravljanje pogrešaka, isticanje sintakse, inteligentno dovršavanje koda, isječke, refaktoriranje koda i ugrađenu kontrolu verzije s Gitom. Korisnici imaju mogućnost mijenjanja teme, prečaca na tipkovnici, postavki i instaliranja proširenja koja omogućuju dodatne funkcionalnosti. U anketi za razvojne programere *Stack Overflow* 2023. godine *VSCode* je rangiran kao najpopularnije razvojno okruženje [26].

4.2. Prikaz programskog rješenja po komponentama

U ovom potpoglavlju opisane su komponente višeplatformske mobilne aplikacije. Opisan je kod koji stoji iza funkcionalnosti i prikazani njegovi ključni dijelovi.

4.2.1. Programska implementacija prijave korisnika

Funkcionalnost prijave omogućuje prethodno registriranim korisnicima pristup aplikaciji. Za prijavu korisnika u sustav i omogućavanje pristupa aplikaciji koristi se funkcija *login* prikazana na slici 4.1. Ova funkcija prima dva argumenta, adresu e-pošte i lozinku korisničkog računa.

```
void login(String email, String password) async {
  state = const LoginStateLoading();
  try {
    await ref.read(authRepositoryProvider).signInWithEmailAndPassword(
          email,
          password,
        );
    state = const LoginStateSuccess();
  } catch (e) {
    state = LoginStateError(e.toString());
  }
}
```

Sl. 4.1 Metoda *login*

Unutar metode *login* poziva se asinkrona *signInWithEmailAndPassword* metoda kojoj se prosljeđuju prvobitno primljeni argumenti. Asinkrone metode omogućuju izvršavanje zadataka bez blokiranja ili čekanja završetka dugotrajnih operacija. To zapravo znači da metoda može započeti obavljati zadatak, kao što je na primjer dohvaćanje podataka, a potom nastaviti s drugim zadatcima dok čeka da se prvi završi. U *Flutteru* se asinkrone metode često kombiniraju s ključnom riječi *await*, što označava čekanje završetka te operacije prije nego što se nastavi sa sljedećim redom koda, bez blokiranja cijelog programa. Metoda *signInWithEmailAndPassword* koristi *Firebase* metodu za prijavu u sustav pomoću objekta *Firebase Auth* klase, koja pokušava prijaviti korisnika s navedenom lozinkom i adresom e-pošte. Ukoliko je postupak prijave uspješan, vraća se objekt *Firebase User*, koji predstavlja autentificiranog korisnika. Ovaj objekt sadrži bitne informacije, za daljnji boravak u aplikaciji, poput userID-a. Funkcija također sadrži *try-catch* blok kako bi uhvatio potencijalne neželjene iznimke. Ako je uhvaćena iznimka *FirebaseAuthException*, provjerava se specifični kod pogreške kako bi se pružila odgovarajuća povratna informacija korisniku. Ukoliko je kod pogreške: *'invalid-email'* korisniku se izbacuje poruka *'Incorrect e-mail provided.'* , *'invalid-credential'* korisniku se izbacuje poruka *'User not found.'*, ili ako je kod pogreške neki drugi korisniku se izbacuje opća poruka *'An error occurred. Please try again later'*. Metoda *signInWithEmailAndPassword* prikazana je na slici 4.2.

```
Future<User?> signInWithEmailAndPassword(
    String email, String password) async {
  try {
    final result = await _auth.signInWithEmailAndPassword(
      email: email,
      password: password,
    );
    return result.user;
  } on FirebaseAuthException catch (e) {
    if (e.code == 'user-not-found') {
      throw AuthException('User not found');
    } else if (e.code == 'wrong-password') {
      throw AuthException('Wrong password');
    } else {
      throw AuthException('An error occured. Please try again later');
    }
  }
}
```

Sl. 4.2 Metoda *signInWithEmailAndPassword*

4.2.2. Programska implementacija registriranja korisnika

Ova funkcionalnost omogućuje novim korisnicima registraciju novog računa u aplikaciji. Proces uključuje interakciju s *Firebase* bazom za pohranu podataka novog korisnika. Korišteni su paketi *Firebase Auth*, *Cloud Firestore*, te *Firebase Storage*. Funkcionalnost registracije započinje pozivanjem metode *register* koja prima dva argumenta, adresu e-pošte i lozinku kao što je prikazano na slici 4.3. Metoda se na zaslonu za registraciju poziva pomoću *login* kontrolera. Unutar metode postoji *try-catch* blok unutar kojeg se pokušava stvoriti novi korisnik pomoću predanih argumenata lozinke i adrese e-pošte koji se zatim kao argumenti prosljeđuju metodi *createUserWithEmailAndPassword*, ukoliko novi korisnik nije uspješno stvoren hvata se iznimka '*User registration failed*'.

```
void register(String email, String password) async {
  state = const LoginStateLoading();
  try {
    final user = await ref.read(authRepositoryProvider).createUserWithEmailAndPassword(
            email,
            password,
          );
    if (user != null) {
      await ref.read(userRepositoryProvider).createUserProfile(user.uid);
      state = const LoginStateSuccess();
    } else {
      throw Exception('User registration failed');
    }
  } catch (e) {
    state = LoginStateError(e.toString());
  }
}
```

Sl. 4.3 Metoda *register*

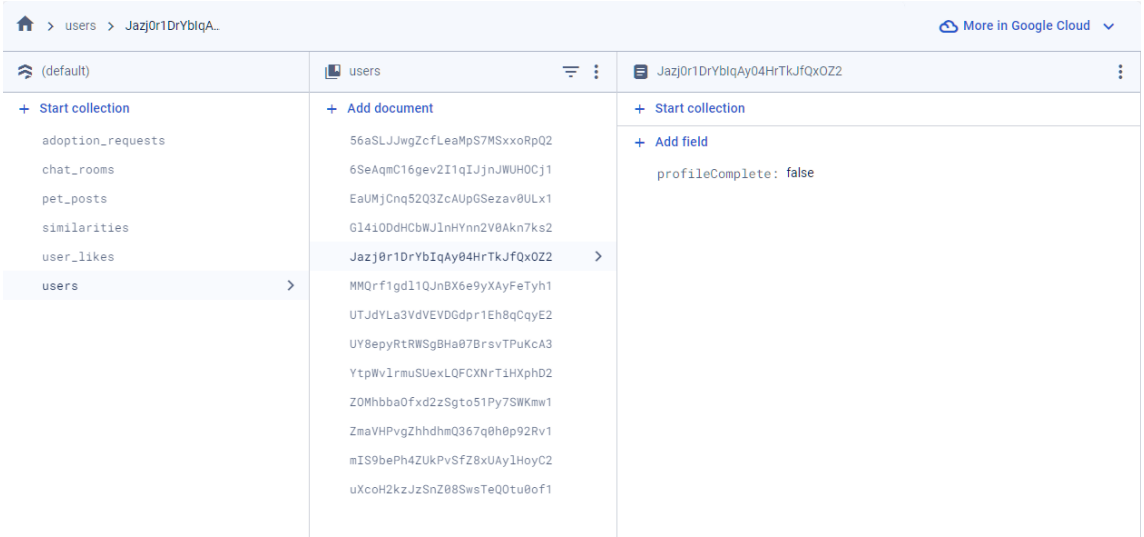
U metodi *createUserWithEmailAndPassword* preko objekta klase *Firebase Auth* i predefinirane funkcije paketa za stvaranje novog korisnika, vraća se vrijednost novo stvorenog korisnika u obliku *User* objekta. Sustav provjerava je li navedena adresa e-pošte već u upotrebi, te ako jest podiže iznimku *'The email is already in use'* koja se prikazuje na zaslonu u *SnackBar widgetu*. Isto tako sustav provjerava i na taj način osigurava da lozinka sadrži najmanje 6 znakova, u suprotnom hvata iznimku *'The password provided is too weak'*. Metoda je definirana tako da ako dođe do nekih drugih neočekivanih iznimaka, hvata se iznimka *'An error occurred. Please try again later'*. Sve prethodno opisano prikazano je na slici 4.3. Podaci novo stvorenog korisničkog računa se također spremaju u *Firebase* bazu podataka s jedinstvenim userID-em za svakog novog korisnika.

```
Future<User?> createUserWithEmailAndPassword(String email, String password) async {
  try {
    final result = await _auth.createUserWithEmailAndPassword(
      email: email,
      password: password,
    );
    return result.user;
  } on FirebaseAuthException catch (e) {
    if (e.code == 'email-already-in-use') {
      throw AuthException('The email is already in use.');
    } else if (e.code == 'weak-password') {
      throw AuthException('The password provided is too weak.');
    } else {
      throw AuthException('An error occurred. Please try again later.');
    }
  }
}
```

Sl. 4.4 Metoda *createUserWithEmailAndPassword*

4.2.3. Programska implementacija upravljanja korisničkim profilima

Nakon što korisnik stvori svoj novi korisnički račun, *Firebase Auth* metoda *createUserWithEmailAndPassword* automatski korisnika prijavljuje odmah nakon registracije računa. Prilikom registracije podatci korisnika se pohranjuju u *Firebase*, te se novom korisničkom računu dodjeljuje polje *profileComplete* i njegova inicijalna vrijednost *false* (Slika 4.5.). Ovo polje označava da korisnik još uvijek nije kompletirao svoj profil unutar aplikacije.



Sl. 4.5 Prikaz *Firestore* dokumenta novog korisničkog računa čiji profil nije kompletiran

Prilikom prijave u aplikaciju poziva se metoda *_checkProfileCompletion*. Ova metoda pomoću ID-a i vrijednosti polja *profileComplete* korisnika koji se prijavio provjerava u *Firebase* bazi podataka je li korisnik kompletirao svoj profil pozivajući funkciju *checkProfileCompletion*, koja prima korisnikov ID kao argument. Ova funkcija onda poziva funkciju *isProfileComplete* i prosljeđuje joj korisnikov ID kao argument. Metoda *isProfileComplete* dohvaća dokument korisnika pomoću njegovog ID-a iz kolekcije *'users'* u *Firebase Firestore* te provjerava vrijednost

polja *profileComplete*. Ukoliko je vraćena vrijednost *false* korisnik se usmjerava na zaslon predviđen za dopunjavanje profila, u suprotnom korisniku se omogućava pristup aplikaciji. Opisane funkcije prikazane su slikom 4.6.

```
Future<void> _checkProfileCompletion() async {
  final user = ref.read(authRepositoryProvider).currentUser;
  if (user != null) {
    final isProfileComplete = await ref.read(profileServiceProvider).checkProfileCompletion(user.uid);
    if (!isProfileComplete) {
      if (mounted) {
        Navigator.of(context).pushReplacement(
        MaterialPageRoute(builder: (context) => const ProfileCompletionScreen()),
      );
      }
    } else {
      if (mounted) {
        Navigator.of(context).pushReplacement(
        MaterialPageRoute(builder: (context) => const PreHomeScreen()),
      );
      }
    }
  }
}
```

```
Future<bool> checkProfileCompletion(String userId) async {
  return await userRepository.isProfileComplete(userId);
}
```

```
Future<bool> isProfileComplete(String uid) async {
  final doc = await _firestore.collection('users').doc(uid).get();
  return doc.exists && doc.data()?['profileComplete'] == true;
}
```

Sl. 4.6 Metode *_checkProfileCompletion*, *checkProfileCompletion* i *isProfileComplete*

Na zaslonu za kompletiranje profila klikom na gumb *Complete Profile* poziva se funkcija *completeProfile* koja prvo provjerava jesu li ispunjena sva tekstualna polja ispunjena, odnosno jesu li pružene sve tražene informacije potrebne da bi se korisnički profil kompletirao. Ako nisu korisniku se prikazuje poruka *'All fields must be filled out'* kao što je prikazano na slici 4.7. Ukoliko jesu, poziva se funkcija *completeUserProfile* koja prima podatke iz popunjenih polja na zaslonu kao argumente.

```
void completeProfile() async {
  if (nameController.text.isEmpty ||
      surnameController.text.isEmpty ||
      addressController.text.isEmpty ||
      phoneController.text.isEmpty ||
      favoritePetTypeController.text.isEmpty ||
      favoriteBreedController.text.isEmpty) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('All fields must be filled out'),
      ), // SnackBar
    );
    return;
  }
  await completeUserProfile(
    ref,
    nameController.text,
    surnameController.text,
    addressController.text,
    phoneController.text,
    favoritePetTypeController.text,
    favoriteBreedController.text
  );
  if (context.mounted) {
    Navigator.of(context).pushReplacement(
      MaterialPageRoute(builder: (context) => const PreHomeScreen()),
    );
  }
}
```

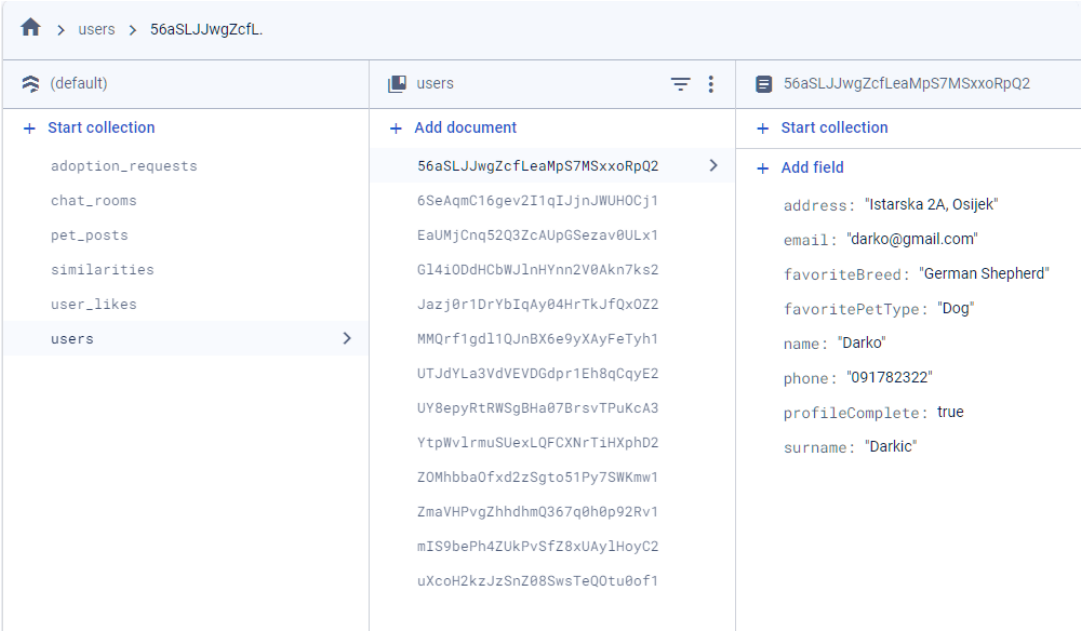
Sl. 4.7 Metoda *completeProfile*

Funkcija *completeUserProfile* zatim ažurira korisnikov dokument i postavlja vrijednosti određenih polja korisničkog računa na vrijednosti koje su pružene prilikom popunjavanja tekstualnih polja na zaslonu za kompletiranje profila, te mijenja vrijednost polja *profileComplete* na *true* kao što je prikazano slikom 4.8. Korisniku koji je kompletirao svoj profil pruža se pristup aplikaciji, te je usmjeren na početni zaslon aplikacije.

```
Future<void> completeUserProfile(
  String uid, String? email, String name, String surname, String address,
  String phone, String favoritePetType, String favoriteBreed)
  async {
  await _firestore.collection('users').doc(uid).update({
    'name': name,
    'email': email,
    'surname': surname,
    'address': address,
    'phone': phone,
    'favoritePetType': favoritePetType,
    'favoriteBreed': favoriteBreed,
    'profileComplete': true,
  });
}
```

Sl. 4.8 Metoda *completeUserProfile*

Na slici 4.9 prikazan je primjer dokumenta kompletiranog korisničkog profila.



Sl. 4.9 Prikaz *Firestore* dokumenta korisničkog računa čiji je profil kompletiran

Jednom kada korisnik ima pristup aplikaciji, pomoću izbornika ladice moguće je usmjeriti se do zaslona koji sadrži podatke profila korisnika. Na ovom zaslonu moguće je ažurirati podatke profila. Korisnik može urediti tekstualna polja, te pritiskom na gumb *'Save Changes'* pokreće se funkcija *_updateProfile* koja pomoću ID-a trenutno prijavljenog korisnika ažurira njegov *Firestore* dokument u kolekciji *'users'*, kao što je prikazano na slici 4.10.

```
void _updateProfile() async {
  final user = ref.read(authRepositoryProvider).currentUser;
  if (user != null) {
    await FirebaseFirestore.instance.collection('users').doc(user.uid).update({
      'name': nameController.text,
      'email': emailController.text,
      'surname': surnameController.text,
      'address': addressController.text,
      'phone': phoneController.text,
      'favoritePetType': favoritePetTypeController.text,
      'favoriteBreed': favoriteBreedController.text,
    });
  }
}
```

Sl. 4.10 Metoda *_updateProfile*

4.2.4. Programska implementacija kreiranja objava životinja za udomljivanje

Kreiranje objave životinje za udomljivanje postiže se funkcijom *submitPost* uslužne klase *PostSubmitService* koja kao argument prima objekt klase *PetPostModel* (slika 4.11). Ovaj objekt se kreira na temelju tekstualnih polja i odabrane slike životinje, kao što je prikazano na slici 4.12.

```
class PetPostModel {
  final String postId;
  final String userId;
  final String name;
  final String breed;
  final String type;
  final double weight;
  final int age;
  final String imageUrl;
  final bool isAdopted;
  final String petAddress;

  PetPostModel({
    required this.postId,
    required this.userId,
    required this.name,
    required this.breed,
    required this.type,
    required this.weight,
    required this.age,
    required this.imageUrl,
    this.isAdopted = false,
    required this.petAddress,
  });
```

Sl. 4.11 *PetPostModel* model klasa objave životinje za udomljivanje

```
final petPost = PetPostModel(
  postId: postId,
  userId: user.uid,
  name: formController.nameController.text,
  breed: formController.breedController.text.trim(),
  type: formController.typeController.text.trim(),
  weight: double.parse(formController.weightController.text),
  age: int.parse(formController.ageController.text),
  imageUrl: downloadUrl,
  petAddress: formController.locationController.text,
); // PetPostModel
```

Sl. 4.12 Kreiranje objekta koji se predaje kao argument funkciji *submitPost*

Da bi se odabrala slika životinje koja će se koristiti kao slika objave pokreće se funkcija *_pickImage* koja preko pomoćne klase *ImagePickerHelper* i paketa *ImagePicker* omogućava korisniku da odabere sliku iz galerija svog uređaja kao što je prikazano na slici 4.13.

```
void _pickImage() async {
  final image = await imagePickerHelper.pickImageFromGallery();
  if (image != null) {
    setState(() {
      formController.image = image;
    });
  }
}
```

```
Future<File?> pickImageFromGallery() async {
  final image = await ImagePicker().pickImage(source: ImageSource.gallery);
  return image != null ? File(image.path) : null;
}
```

Sl. 4.13 Metoda *_pickImage*

Nakon što je objekt *PetPostModel* klase potpuno inicijaliziran predaje se kao argument funkciji *submitPost* koja pohranjuje podatke objave u *Firestore* kolekciju *'pet_posts'* (slika 4.14).

```
Future<void> submitPost(PetPostModel petPost) async {
  await firestore.collection('pet_posts').doc(petPost.postId).set(petPost.toMap());
}
```

Sl. 4.14 Metoda *submitPost*

4.2.5. Programska implementacija pregleda i pretrage životinja

Riverpod provider „petPostsStreamProvider“ koji sluša promjene u *Firestore 'pet_posts'* kolekciji dohvaća dokumente koji predstavljaju popis objava životinja pomoću funkcije *getPetPosts* (slika 4.15).

```
final petPostsStreamProvider = StreamProvider<List<PetPostModel>>((ref) {
  final petRepository = ref.watch(petPostRepositoryProvider);
  return petRepository.getPetPosts();
});
```

```
Stream<List<PetPostModel>> getPetPosts() {
  return _firestore.collection('pet_posts').snapshots().map((snapshot) {
    return snapshot.docs.map((doc) {
      return PetPostModel.fromMap(doc.data());
    }).toList();
  });
}
```

Sl. 4.15 *Provider petPostsStreamProvider* i *getPetPosts* metoda

Zatim *checkBoxFilteredProvider*, koji sluša *petPostsStreamProvider* koji mu dobavlja listu objava i *checkBoxProvider* koji mu dobavlja stanje potvrdnog okvira, vraća filtriranu listu objava ovisno o stanju potvrdnog okvira (slika 4.16).

```
final checkBoxFilteredProvider = Provider.autoDispose<List<PetPostModel>>((ref) {
  final checkBox = ref.watch(checkBoxProvider);
  final petPosts = ref.watch(petPostsStreamProvider).value ?? [];

  if (checkBox==true) {
    return petPosts;
  } else {
    return petPosts
      .where((petPost) => petPost.isAdopted==false).toList();
  }
});
```

Sl. 4.16 *Provider checkBoxFilteredProvider*

Filtrirana lista koju *checkBoxFilteredProvider* daje dalje se filtrira ovisno o unesenom upitu o pretraživanju (engl. *search query*). Ovu listu vraća *filteredPetPostsProvider* koji sluša *searchQueryProvider* koji vraća stanje upita o pretraživanju, i *checkBoxFilteredProvider* koji vraća prethodno spomenutu filtriranu listu, kao što je prikazano na slici 4.17.

```
final filteredPetPostsProvider = Provider.autoDispose<List<PetPostModel>>((ref) {
  final searchQuery = ref.watch(searchQueryProvider);
  final checkBoxList = ref.watch(checkBoxFilteredProvider);

  if (searchQuery.isEmpty) {
    return checkBoxList;
  } else {
    return checkBoxList
        .where((petPost) => petPost.breed.toLowerCase().contains(searchQuery.toLowerCase()))
        .toList();
  }
});
```

Sl. 4.17 *Provider filteredPetPostsProvider*

Vrijednost upita o pretraživanju i stanje *providera searchQueryProvider* mijenja se dinamično tekstualnim unosom korisnika, slično vrijedi i za stanje potvrdnog okvira koje se mijenja klikom na okvir. Logika za promjenu stanja ova dva *providera* prikazana je na slici 4.18.

```
child: TextField(
  controller: controller,
  onChanged: (value) => ref.read(searchQueryProvider.notifier).state = value,
  decoration: InputDecoration(
    hintText: 'Search by breed...',
    prefixIcon: const Icon(Icons.search),
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10.0),
    ), // OutlineInputBorder
  ), // InputDecoration
), // TextField
```

```
Checkbox(
  value: checkBox,
  onChanged: (bool? value) {
    ref.read(checkBoxProvider.notifier).state = value!;
  }), // Checkbox
```

Sl. 4.18 Prikaz logike za promjenu stanja *providera*

4.2.6. Programska implementacija stvaranja zahtjeva za udomljivanjem

Stvaranje zahtjeva za udomljivanjem pojedinog ljubimca odrađuje se funkcijom *submitAdoptionRequest*. Funkcija prvo provjerava je li korisnik autentificiran pristupajući trenutno prijavljenom korisniku, ukoliko je ovdje vraćena vrijednost jednaka *null* vrijednosti, funkcija se zatvara. Zatim se dohvaća dokument zadane objave ljubimca iz *Firestore* kolekcije *'pet_posts'*, te nakon tog provjerava postoji li ovaj dokument, ukoliko ne postoji funkcija se ovdje zatvara. Nakon što su dohvaćeni podatci objave, u dvije zasebne varijable spremaju se *'userId'*, koji predstavlja ID vlasnika ove objave, i *'name'* koji predstavlja ime životinje u zadanoj objavi. Dohvaća se *Firestore* dokument trenutno prijavljenog korisnika kolekcije *'users'* preko korisnikovog ID-a, te se iz dokumenta u posebnu varijablu sprema *'name'* koja predstavlja ime korisnika koji stvara zahtjev. Zatim se kreira novi objekt klase *AdoptionRequestModel* prikazane slikom 4.19 koji sadrži jedinstveni ID ovog zahtjeva, ID objave, ID trenutnog korisnika koji stvara

zahtjev, korisnikovo ime, ime ljubimca vezanog za objavu na temelju koje se stvara zahtjev, status zahtjeva kojemu je inicijalna vrijednost automatski postavljena na *'pending'* i ID vlasnika ove objave.

```
class AdoptionRequestModel {
  final String requestId;
  final String postId;
  final String postOwnerId;
  final String requesterId;
  final String requesterName;
  final String petName;
  final String message;
  final DateTime requestDate;
  final String status; // 'pending', 'accepted', 'rejected'

  AdoptionRequestModel({
    required this.requestId,
    required this.postId,
    required this.postOwnerId,
    required this.requesterId,
    required this.requesterName,
    required this.petName,
    required this.message,
    required this.requestDate,
    required this.status,
  });
```

Sl. 4.19 *AdoptionRequestModel* model klasa zahtjeva za udomljivanjem

Zahtjev za udomljivanje se na kraju sprema u *Firestore* kolekciju *'adoption_requests'*. Nakon što je funkcija *submitAdoptionRequest* izvršena korisnik je obavješten putem poruke *'Adoption request submitted'* koja se ispisuje na zaslonu. Opisana funkcija *submitAdoptionRequest* prikazana je na slici 4.20.

```
Future<void> submitAdoptionRequest(PetPostModel petPost) async {
  final user = FirebaseAuth.instance.currentUser;
  if (user == null) return;

  final petPostSnapshot = await FirebaseFirestore.instance
      .collection('pet_posts')
      .doc(petPost.postId)
      .get();

  if (!petPostSnapshot.exists) {
    return;
  }

  final petPostData = petPostSnapshot.data();
  final postOwnerId = petPostData?['userId'];
  final petName = petPostData?['name'];

  final userDoc = await FirebaseFirestore.instance
      .collection('users')
      .doc(user.uid)
      .get();

  final requesterName = userDoc.data()?['name'];

  final request = AdoptionRequestModel(
    requestId: FirebaseFirestore.instance.collection('adoption_requests').doc().id,
    postId: petPost.postId,
    requesterId: user.uid,
    requesterName: requesterName ?? 'Unknown',
    petName: petName ?? 'Unknown',
    message: 'I am interested in adopting this pet.',
    requestDate: DateTime.now(),
    status: 'pending',
    postOwnerId: postOwnerId ?? 'Unknown',
  ); // AdoptionRequestModel

  await FirebaseFirestore.instance.collection('adoption_requests').doc(request.requestId).set(request.toMap());
}
```

Sl. 4.20 Funkcija *submitAdoptionRequest*

4.2.7. Programska implementacija upravljanja zahtjevima

Za upravljanje pristiglim zahtjevima definiran je *riverpod provider receivedRequestsProvider* koji vraća tok liste objekata tipa *AdoptionRequestModel*. U definiciji *providera* provjerava se postoji li trenutno prijavljen korisnik, ukoliko je ova vrijednost *null*, *provider* vraća prazan tok što sprječava dohvaćanje zahtjeva. Ako korisnik postoji, *provider* kreira upit, te dobavlja dokumente iz *Firestore* kolekcije *'adoption_requests'*, gdje upiti pretraživanja odgovaraju tome da je *'postOwnerId'* jednak korisnikovom ID-u i gdje je status zahtjeva *'pending'* kao što je prikazano na slici 4.21.

```
final receivedRequestsProvider = StreamProvider.autoDispose<List<AdoptionRequestModel>>((ref) {
  final user = ref.watch(authStateProvider).value;
  if (user == null) {
    return Stream.value([]);
  }
  return FirebaseFirestore.instance
    .collection('adoption_requests')
    .where('postOwnerId', isEqualTo: user.uid)
    .where('status', isEqualTo: 'pending')
    .snapshots()
    .map((snapshot) => snapshot.docs
    .map((doc) => AdoptionRequestModel.fromMap(doc.data()))
    .toList());
});
```

Sl. 4.21 *Provider receivedRequestsProvider*

Kada su zahtjevi dobavljeni korisnik ih može odobriti pozivajući funkciju *_acceptRequest*, ili odbiti pozivajući funkciju *_declineRequest*. Funkcija za odbijanje zahtjeva poziva funkciju *updateAdoptionRequestStatus* koja kao argument prima ID zahtjeva i predefiniran niz. Funkcija *updateAdoptionRequestStatus* pristupa dokumentu kolekcije *'adoption_requests'* preko ID-a koji joj je predan kao argument i ažurira njegov status na *'rejected'* kao što je prikazano na slici 4.22.

```
Future<void> _declineRequest(BuildContext context, WidgetRef ref, AdoptionRequestModel request) async {
  await ref.read(adoptionRequestRepositoryProvider).updateAdoptionRequestStatus(request.requestId, 'rejected');
}
```

```
Future<void> updateAdoptionRequestStatus(String requestId, String status) async {
  final requestDoc = _firestore.collection('adoption_requests').doc(requestId);
  await requestDoc.update({'status': status});
}
```

Slika 4.22 Metode *_declineRequest* i *updateAdoptionRequestStatus*

Funkcija za prihvaćanje zahtjeva koristi *provider* za repozitorij zahtjeva za udomljivanje i *provider* za repozitorij objava ljubimaca. Preko repozitorija za zahtjeve dohvaćaju se svi zahtjevi za odabranu objavu ljubimca i spremaju se u dodatnu varijablu. Zatim se poziva funkcija *updateAdoptionRequestStatus* koja ažurira status odabranog zahtjeva na *'accepted'*, nakon čega funkcija prolazi kroz sve ostale zahtjeve za istu tu objavu, te ažurira njihov status na *'rejected'*. Naposljetku se poziva funkcija *updatePetPostStatus* kojoj se daje kao argument ID odabrane objave. Ova funkcija ažurira vrijednost polja *'isAdopted'* odabrane objave na *true*. Logika funkcija *_acceptRequest* i *updatePetPostStatus* prikazana je na slikama 4.23 i 4.24.

```
Future<void> _acceptRequest(BuildContext context, WidgetRef ref, AdoptionRequestModel selectedRequest) async {
  final adoptionRequestRepository = ref.read(adoptionRequestRepositoryProvider);
  final petRepository = ref.read(petPostRepositoryProvider);
  try {
    final requestsStream = adoptionRequestRepository.getAdoptionRequestsForPost(selectedRequest.postId);
    final requests = await requestsStream.first;
    if (requests.isEmpty) {
      if (context.mounted) {
        ScaffoldMessenger.of(context).showSnackBar(const SnackBar(content: Text('No requests found for this pet.')));
      }
      return;
    }
    await adoptionRequestRepository.updateAdoptionRequestStatus(selectedRequest.requestId, 'accepted');
    for (var request in requests) {
      if (request.requestId != selectedRequest.requestId) {
        if (request.status != 'accepted') {
          try {
          await adoptionRequestRepository.updateAdoptionRequestStatus(request.requestId, 'rejected');
        } catch (e) {
          print('Error updating request ${request.requestId}: $e');
          if (context.mounted) {
            ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('Error rejecting request ${request.requestId}.')));
          }
        }}
      }
    }
    await petRepository.updatePetPostStatus(selectedRequest.postId);
  } catch (e) {
    if (context.mounted) {
      ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('Error accepting request: $e')));
    }
  }
}
```

Sl. 4.23 Funkcija *_acceptRequest*

```
Future<void> updatePetPostStatus(String postId) async {
  final postDoc = _firestore.collection('pet_posts').doc(postId);
  await postDoc.update({'isAdopted': true});
}
```

Sl. 4.24 Funkcija *updatePetPostStatus*

Funkcionalnost upravljanje zahtjevima podrazumijeva i dohvaćanje zahtjeva koje je korisnik prethodno već obradio, odnosno odobrio ili odbio. Poziva se funkcija *getDecisionHistory* koja kao argument prima ID prijavljenog korisnika. Dobavljaju se dokumenti *Firestore* kolekcije *'adoption_requests'* s upitima pretraživanja gdje je *'postOwnerId'* jednak ID-u korisnika i gdje je status zahtjeva *'accepted'* ili *'rejected'* kao što je prikazano na slici 4.25.

```
Future<List<AdoptionRequestModel>> getDecisionHistory(String userId) async {
  final snapshot = await _firestore
  .collection('adoption_requests')
  .where('postOwnerId', isEqualTo: userId)
  .where('status', whereIn: ['accepted', 'rejected'])
  .get();

  return snapshot.docs.map((doc) => AdoptionRequestModel.fromMap(doc.data())).toList();
}
```

Sl. 4.25 Funkcija *getDecisionHistory*

Funkcionalnost upravljanja zahtjevima također podrazumijeva i dohvaćanje zahtjeva koje je korisnik stvorio i poslao drugim korisnicima aplikacije. Dobavljaju se dokumenti *Firestore* kolekcije *'adoption_requests'* sa upitima pretraživanja gdje je *'requesterId'* jednak ID-u korisnika i gdje je status zahtjeva *'pending'* kao što je prikazano na slici 4.26.

```
final requestStream = FirebaseFirestore.instance
    .collection('adoption_requests')
    .where('requesterId', isEqualTo: userId)
    .where('status', isEqualTo: 'pending')
    .snapshots();
```

Sl. 4.26 Dobavljanje zahtjeva koje je korisnik stvorio

Funkcionalnost upravljanja zahtjevima također podrazumijeva i dohvaćanje zahtjeva na koje je korisnik dobio odgovor, odnosno koje su mu drugi korisnici odobrili ili odbili. Dobavljaju se dokumenti *Firestore* kolekcije *'adoption_requests'* s upitima pretraživanja gdje je *'requesterId'* jednak ID-u korisnika i gdje je status zahtjeva *'accepted'* ili *'rejected'*, slično kao i u prethodnom slučaju.

4.2.8. Programska implementacija geolokacije

Za ostvarivanje funkcionalnosti geolokacije i implementiranje *Google Maps* karte, kao i računanje distance, ruta i sličnih stvari korišteni su paketi: *Google Maps Flutter*, *Geocoding*, *Flutter Map Math* i *Flutter Polyline Points*. Na zaslonu karte za kreiranje i učitavanje markera poziva se funkcija *_loadMarkers*. Koristi se *provider mapServiceProvider* za dohvaćanje usluga koje

upravljaju s radom markerima, te *markersProvider* koji pruža podatke o markerima. *Provider markersProvider* prvo dohvaća trenutno prijavljenog korisnika i provjerava je li *null*, ako je *null provider* vraća praznu mapu. Ako je korisnik prijavljen, dohvaća se njegov profil preko *userRepositoryProvider providera* i *getUserProfile* metode, dok se objave ljubimaca dohvaćaju putem *petPostsStreamProvider providera*. Kada je korisnik dohvaćen, poziva se funkcija *locationFromAddress*, čiji se rezultat sprema u dodatnu varijablu, koja prima adresu kao argument. Ova funkcija pretvara predanu joj adresu u obliku niza u koordinate koje se koriste za kreiranje markera na karti. Zatim se provjerava je li lokacija uspješno izračunata ili je varijabla prazna, ako nije prazna sprema se u mapu pod ključem *'userLocation'*. Uz korisnikovu lokaciju, u mapu se spremaju i objave ljubimaca pod ključem *'petPosts'*, te *provider* naposljetku vraća kompletnu mapu s podacima. Kod *markersProvider providera* prikazan je slikom 4.27.

```
final markersProvider = FutureProvider.autoDispose<Map<String, dynamic>>((ref) async {
  final loggedUser = ref.watch(authRepositoryProvider).currentUser;

  if (loggedUser == null) {
    return <String, dynamic>{};
  }

  final user = await ref.watch(userRepositoryProvider).getUserProfile(loggedUser.uid);
  final petPosts = await ref.watch(petPostsStreamProvider.future);

  final data = <String, dynamic>{};

  print('ADRESA KORISNIKA: ${user!.address} ');

  if (user.address.isNotEmpty) {
    final userLocations = await locationFromAddress(user.address);
    if (userLocations.isNotEmpty) {
      final userLocation = userLocations.first;
      data['userLocation'] = userLocation;
    }
  }

  data['petPosts'] = petPosts;

  return data;
});
```

Sl. 4.27 *Provider markersProvider*

Zatim *_loadMarkers* funkcija iz mape koju daje *markersProvider* podatke pohranjuje u dodatne varijable, ove varijable se provjeravaju ponovno jesu li im vrijednosti *null*, i ako jesu funkcija prestaje s radom. Zatim se poziva *createMarkers* funkcija *mapService* usluge, koja kreira markere na temelju korisnikove lokacije i objava ljubimaca, gdje je svaki zaseban marker interaktivan, odnosno pri kliku poziva funkciju koja ažurira trenutno odabranu objavu i prikazuje modal s detaljima o ljubimcu. Na slici 4.28 je prikazana *createMarkers* funkcija.

```
Future<Set<Marker>> createMarkers({
  required Location userLocation,
  required List<PetPostModel> petPosts,
  required void Function(PetPostModel, Location, Location) onMarkerTap,
}) async {
  final markers = <Marker>{};

  markers.add(
    Marker(
      markerId: const MarkerId('userLocation'),
      position: LatLng(userLocation.latitude, userLocation.longitude),
      infoWindow: const InfoWindow(title: 'Your Location'),
      icon: BitmapDescriptor.defaultMarkerWithHue(BitmapDescriptor.hueBlue),
    ), // Marker
  );

  await Future.wait(
    petPosts.where((petPost) => petPost.petAddress.isNotEmpty).map((petPost) async {
      final petLocations = await locationFromAddress(petPost.petAddress);
      if (petLocations.isNotEmpty) {
        final petLocation = petLocations.first;
        markers.add(
          Marker(
            markerId: MarkerId(petPost.postId),
            position: LatLng(petLocation.latitude, petLocation.longitude),
            infoWindow: InfoWindow(title: petPost.name),
            icon: BitmapDescriptor.defaultMarker,
            onTap: () => onMarkerTap(petPost, userLocation, petLocation),
          ), // Marker
        );
      }
    }),
  );
  return markers;
}
```

Sl. 4.28 Funkcija *createMarkers*

Nakon što se markeri kreiraju, u *setState* metodi lista *_markers* se čisti i ponovno popunjava s novim markerima. Zatim se kao inicijalni položaj kamere postavlja korisnikova lokacija, osim ako nisu predani posebni argumenti početne pozicije kamere. Funkcija *_loadMarkers* prikazana je na slici 4.29.

```
Future<void> _loadMarkers() async {
  try {
    final mapService = ref.read(mapServiceProvider);
    final markersData = await ref.read(markersProvider.future);
    final userLocation = markersData['userLocation'] as Location?;
    final petPosts = markersData['petPosts'];
    List<PetPostModel> petPostsList = [];

    if (petPosts is List<PetPostModel>) {
      petPostsList = petPosts;
    } else if (petPosts != null) {
      throw TypeError();
    }
    if (userLocation == null || petPostsList.isEmpty) {
      return;
    }

    print('KORISNIKOVA LOKACIJA  $userLocation');

  final markers = await mapService.createMarkers(
    userLocation: userLocation,
    petPosts: petPosts,
    onMarkerTap: (petPost, userLoc, petLoc) {
      ref.read(selectedPetProvider.notifier).state = petPost;
      _showPetDetailsModal(context, petPost, userLoc, petLoc);
    },
  );
  setState(() {
    _markers.clear();
    _markers.addAll(markers);
  });
  LatLng targetPosition = widget.initialPosition ??
      LatLng(userLocation.latitude, userLocation.longitude);
  _mapController.animateCamera(
    CameraUpdate.newLatLngZoom(targetPosition, 15.0),
  );
  } catch (e) {
    print('Error in _loadMarkers: $e');
  }
}
```

Sl. 4.29 Funkcija *_loadMarkers*

Funkcija za prikazivanje rute na karti od korisnikove lokacije do lokacije markera objave životinje *showRoute* prikazana je na slici 4.30, prima dva argumenta, lokaciju korisnika i lokaciju na kojoj se nalazi ljubimac. Funkcija poziva metodu *getRoutePoints* kojoj se predaju koordinate korisnikove lokacije i koordinate lokacije objave ljubimca. Metoda *getRoutePoints* dohvaća točke rute između ove dvije lokacije, pomoću objekta klase *PolylinePoints* poziva se funkcija *getRouteBetweenCoordinates* koja vraća točke rute. Nakon što su točke rute dohvaćene, kreira se objekt klase *Polyline* u čijem se konstruktoru predaju prethodno dohvaćene točke rute. Zatim se ažurira stanje tako da se nova ruta dodaje u listu *_polylines*.

```
Future<void> showRoute(BuildContext context, Location userLocation, Location petLocation) async {
  final routePoints = await ref.read(polylineProvider.notifier).getRoutePoints(
    LatLng(userLocation.latitude, userLocation.longitude),
    LatLng(petLocation.latitude, petLocation.longitude),
  );

  final polyline = Polyline(
    polylineId: const PolylineId("route"),
    points: routePoints,
    color: Colors.blue,
    width: 5,
  ); // Polyline

  setState(() {
    _polylines.add(polyline);
  });
}
```

Sl. 4.30 Funkcija *showRoute*

Za računanje udaljenosti između dvije lokacije poziva se funkcija *calculateRoute* koja kao argumente prima dva parametra, lokaciju korisnika i lokaciju objave životinje. Poziva se metoda predefinirane *FlutterMapMath* klase *distanceBetween* kojoj se kao argumenti predaju koordinate lokacije korisnika i koordinate lokacije objave životinje (slika 4.31).

```
String calculateRoute(Location userLocation, Location petPostLocation) {
  double distance = FlutterMapMath().distanceBetween(
    userLocation.latitude,
    userLocation.longitude,
    petPostLocation.latitude,
    petPostLocation.longitude,
    "meters",
  );
  return distance.toStringAsFixed(0);
}
```

Sl. 4.31 Funkcija *calculateRoute*

4.2.9. Programska implementacija komunikacije među korisnicima

Za lakšu komunikaciju između korisnika u aplikaciju je implementiran chat sustav. Za dobavljanje liste korisnika koji su registrirani na aplikaciji koristi se *userStreamProvider* koji vraća dokumente kolekcije *'users'* kao što je prikazano na slici 4.32.

```
final userStreamProvider = StreamProvider.autoDispose<QuerySnapshot>((ref) {
  return FirebaseFirestore.instance.collection('users').snapshots();
});
```

Sl. 4.32 *Provider userStreamProvider*

Slanjem poruke, odnosno pozivanjem funkcije *sendMessage* kreira se novi dokument u *'chat_rooms' Firestore* kolekciji, čiji ID će odgovarati ID-u sobe. Unutar ovog dokumenta spremaju se ID-evi dva korisnika koji komuniciraju u toj sobi unutar *array fielda 'participants'*, te se u kolekciji *'chat_rooms'* stvara pod kolekcija *'messages'* gdje svaki dokument predstavlja poruku. Svaka poruka ima polja *'senderId'*, *'receiverId'*, *'text'*, *'timestamp'* i *'isRead'* kao što je prikazano na slici 4.33.

```
Future<void> sendMessage({
  required String chatRoomId,
  required String currentUserId,
  required String otherUserId,
  required String text,
}) async {
  final message = {
    'senderId': currentUserId,
    'receiverId': otherUserId,
    'text': text.trim(),
    'timestamp': FieldValue.serverTimestamp(),
    'isRead': false,
  };

  final chatRoomRef = _firestore.collection('chat_rooms').doc(chatRoomId);

  await chatRoomRef.set({
    'participants': FieldValue.arrayUnion([currentUserId, otherUserId])
  }, SetOptions(merge: true));

  await chatRoomRef.collection('messages').add(message);
}
```

Sl. 4.33 Funkcija *sendMessage*

Da bi se poruka poslala u točan razgovor, potrebno je odrediti ID sobe, što se rješava implementacijom funkcije *getChatRoomId* koja kao argument prima ID-eve korisnika koji komuniciraju. Na temelju proslijeđenih ID-eva generira se jedinstveni ID za chat sobu u formatu *'currentUserId_otherUserId'* ili obrnuto kao što je prikazano slikom 4.34.

```
String getChatRoomId(String currentUserId, String otherUserId) {
  return currentUserId.compareTo(otherUserId) > 0
      ? '${currentUserId}_$otherUserId'
      : '${otherUserId}_$currentUserId';
}
```

Sl. 4.34 Funkcija *getChatRoomId*

4.2.10. Programska implementacija sustava preporuka

U aplikaciji sustav preporuka implementiran je na dva načina, naime jedan sa metodom filtriranja temeljenoj na sadržaju, i drugi sa metodom kolaborativnog filtriranja. Sustav preporuka ostvaren sa metodom filtriranja na temelju sadržaja implementiran je na način da se izvršavaju određene kalkulacije i izračuni sličnosti pojedinih objava životinja za udomljivanje. Korisnik označava određene objave gdje se poziva funkcija *likePost* koja u *Firestore* dokument, *'user_likes'* koji nosi ID korisnika, u pod kolekciju *'liked_posts'* sprema ID objave koju je korisnik označio (slika 4.35). Ovo predstavlja praćenje objava koje se korisniku sviđaju kako bi se ti podatci dalje iskoristili za razvijanje sustava preporuka.

```
Future<void> likePost(String userId, String postId) async {
  final userLikesRef = _firestore
      .collection('user_likes')
      .doc(userId)
      .collection('liked_posts')
      .doc(postId);

  await userLikesRef.set({
    'likedAt': FieldValue.serverTimestamp(),
    'postId' : postId,
  });
```

Sl. 4.35 Funkcija *likePost*

Zatim se dobavljaju svi trenutno pohranjeni dokumenti iz *Firestore* kolekcije *'pet_posts'*, spremaju se u dodatnu varijablu, te se predaju kao argument funkciji *computeAndStoreSimilarities*. Funkcija *computeAndStoreSimilarities* vrši iteraciju kroz sve dokumente koji su predani kao argument, gdje po redu prvu odabranu objavu uspoređuje sa svakom sljedećom, te im se računaju sličnosti pozivajući funkciju *computeSimilarity*, te predajući prethodno spomenute dvije objave kao argumente. Funkcija *computeSimilarity* izvršava izračun sličnosti između dvije objave, uspoređujući prvo sličnosti atributa *breed* i *type*. Rezultati se pohranjuju u dodatne varijable gdje se varijabli dodjeljuje vrijednost 1.0 ako su *breed* i/ili *type* jednaki, ili 0.0 ako su različiti. Zatim se računaju sličnosti na temelju *age* atributa, gdje se izračunava apsolutna razlika godina između dva ljubimca, te se ta razlika dijeli s 10 da bi se normalizirao rezultat. Oduzimanjem dobivene normalizirane vrijednosti od 1, dobiva se rezultat koji se pohranjuje u dodatnu varijablu. Dakle, manja razlika u godinama rezultirati će u dodjeli većeg boda sličnosti. Na isti način se računaju i dodjeljuju bodovi i za *weight* atribut. Nakon što su izračunati i dodijeljeni bodovi na temelju ovih atributa, računa se ukupna sličnost tako što se ove varijable zbrajaju pomnožene s maksimalnom količinom boda koju pojedini parametar može nositi kao što je prikazano na slici 4.36.

```
double computeSimilarity(PetPostModel postA, PetPostModel postB) {

  // Similarity based on pet's breed (1 if same breed, 0 otherwise)
  double breedScore = postA.breed.toLowerCase() == postB.breed.toLowerCase() ? 1.0 : 0.0;

  // Similarity based on pet's type (1 if same type, 0 otherwise)
  double typeScore = postA.type.toLowerCase() == postB.type.toLowerCase() ? 1.0 : 0.0;

  // Similarity based on pet's age (closer ages result in higher score)
  double ageScore = 1 - (postA.age - postB.age).abs() / 10.0;

  // Similarity based on pet's weight (closer weights result in higher score)
  double weightScore = 1 - (postA.weight - postB.weight).abs() / 10.0;

  // Combine the scores alltogether to get the similarity score between these two pets
  double similarity = (0.4 * breedScore) + (0.4 * typeScore) + (0.1 * ageScore) + (0.1 * weightScore);

  return similarity;
}
```

Sl. 4.36 Funkcija *computeSimilarity*

Nakon što su sličnosti između dvije objave izračunate i pohranjene u dodatnu varijablu, kreira se novi dokument u *Firestore* kolekciji *'similarities'* čiji ID se sastoji od kombinacije ID-a dvije

objave za koje se računa sličnost. Ovaj dokument sadrži polja *'postIdA'*, *'postIdB'* koja predstavljaju ID-eve objava koje su uspoređivane, kao i polje *'similarityScore'* koje predstavlja izračunatu sličnost kao što je prikazano na slici 4.37.

```
Future<void> computeAndStoreSimilarities(List<PetPostModel> petPosts) async {
  for (int i = 0; i < petPosts.length; i++) {
    for (int j = i + 1; j < petPosts.length; j++) {
      double similarityScore = _similarityService.computeSimilarity(petPosts[i], petPosts[j]);

      await _firestore.collection('similarities').doc('${petPosts[i].postId}_${petPosts[j].postId}').set({
        'postIdA': petPosts[i].postId,
        'postIdB': petPosts[j].postId,
        'similarityScore': similarityScore,
      });

      await _firestore.collection('similarities').doc('${petPosts[j].postId}_${petPosts[i].postId}').set({
        'postIdA': petPosts[j].postId,
        'postIdB': petPosts[i].postId,
        'similarityScore': similarityScore,
      });
    }
  }
}
```

Sl. 4.37 Funkcija *computeAndStoreSimilarities*

Dohvaćanje personaliziranih preporuka za korisnika vrši se pozivom *fetchRecommendedPosts* funkcijom koja kao argument prima ID korisnika i listu objava koje je korisnik označio kao da mu se sviđaju. Iz *Firestore* kolekcije *'similarities'* dohvaćaju se dokumenti kojima je vrijednost *'postIdA'* polja jednaka jednoj od vrijednosti iz liste objava koje se korisniku sviđaju, i kojima je vrijednost *'similarityScore'* polja veća od predefinirane granične vrijednosti 0.6. Zatim se u dohvaćenim dokumentima provjerava sadrži li lista označenih objava vrijednost polja *'postIdB'*, jer ako sadrži to bi značilo da se korisniku sviđaju obje objave koje su uspoređene i ovaj dokument se preskače. Ukoliko lista ne sadrži ID druge objave, što zapravo znači da se korisniku sviđa jedna od dvije objave koje su uspoređivane, iz kolekcije *'pet_posts'* se dohvaća dokument čiji ID odgovara vrijednosti *'postIdB'* polja, te se objava pod ovim dokumentom dodaje u listu koja predstavlja listu objava koje se preporučuju korisniku. Prethodno opisana logika prikazana je slikom 4.38.

```
Future<List<PetPostModel>> fetchRecommendedPosts(String userId, List<String> likedPostIds) async {
  final List<PetPostModel> recommendedPosts = [];
  const double similarityThreshold = 0.6;

  for(String likedPostId in likedPostIds) {
    final querySnapshot = await _firestore
      .collection('similarities')
      .where('postIdA', isEqualTo: likedPostId)
      .where('similarityScore', isGreaterThan: similarityThreshold)
      .orderBy('similarityScore', descending: true)
      .limit(10)
      .get();

    for (var doc in querySnapshot.docs) {
      final data = doc.data();
      final postIdB = data['postIdB'];

      if (!likedPostIds.contains(postIdB)) {
        final postSnapshot = await _firestore.collection('pet_posts').doc(postIdB).get();
        final postData = postSnapshot.data();

        recommendedPosts.add(PetPostModel(
          postId: postSnapshot.id,
          userId: postData!['userId'],
          isAdopted: postData['isAdopted'],
          name: postData['name'],
          age: postData['age'],
          weight: postData['weight'],
          breed: postData['breed'],
          imageUrl: postData['imageUrl'],
          type: postData['type'],
          petAddress: postData['petAddress'],
        ));
      }
    }
  }
  return recommendedPosts;
}
```

Sl. 4.38 Funkcija *fetchRecommendedPosts*

Sustav preporuka temeljen na kolaborativnom filtriranju implementiran je na način da se prate interakcije i ponašanje drugih korisnika, te se korisniku preporučuje sadržaj koji su sviđa ostalim korisnicima sličnih preferencija. Ovo je ostvarena na način da se unutar funkcije *_getCollaborativeRecommendations* pozivaju tri pomoćne funkcije pomoću kojih se dolazi do preporuka, te se ove preporuke vraćaju ovom funkcijom kao što je prikazano na slici 4.39.

```
Future<List<PetPostModel>> _getCollaborativeRecommendations(String userId) async {
  final recommendationRepo = RecommendationRepository();

  List<String> likedPosts = await recommendationRepo.getUserLikedPosts(userId);
  List<String> similarUsers = await recommendationRepo.getSimilarUsers(userId, likedPosts);
  List<PetPostModel> recommendedCollabPosts = await recommendationRepo.getCollabRecommendedPosts(similarUsers, likedPosts);

  return recommendedCollabPosts;
}
```

Sl. 4.39 Funkcija *_ getCollaborativeRecommendations*

Funkcija *getUserLikedPosts* služi za dohvaćanje liste ID-jeva objava koje je korisnik označio kao da mu se sviđaju kao što je prikazano na slici 4.40.

```
Future<List<String>> getUserLikedPosts(String userId) async {

  CollectionReference likedPostsRef = FirebaseFirestore.instance
      .collection('user_likes')
      .doc(userId)
      .collection('liked_posts');

  QuerySnapshot likedPostsSnapshot = await likedPostsRef.get();

  List<String> likedPosts = likedPostsSnapshot.docs.map((doc) => doc.id).toList();

  return likedPosts;
}
```

Sl. 4.40 Funkcija *getUserLikedPosts*

Lista koju funkcija *getUserLikedPosts* se zatim kao argument predaje funkciji *getSimilarUsers*. Funkcija *getSimilarUsers* pronalazi korisnike koji imaju slične interese kao trenutni korisnik, odnosno pronalazi korisnike koji su označili da im se sviđa barem jednu istu objavu kao i trenutni korisnik kao što je prikazano na slici 4.41.

```
Future<List<String>> getSimilarUsers(String currentUserId, List<String> currentUserLikes) async {
  QuerySnapshot userSnapshots = await FirebaseFirestore.instance.collection('user_likes').get();
  List<String> similarUsers = [];

  for (var doc in userSnapshots.docs) {
    if (doc.id != currentUserId) {
      QuerySnapshot otherUserLikesSnapshot = await FirebaseFirestore.instance
          .collection('user_likes')
          .doc(doc.id)
          .collection('liked_posts')
          .get();

      List<String> otherUserLikes = otherUserLikesSnapshot.docs.map((doc) => doc.id).toList();

      if (otherUserLikes.any((petId) => currentUserLikes.contains(petId))) {
        similarUsers.add(doc.id);
      }
    }
  }
  return similarUsers;
}
```

Sl. 4.40 Funkcija *getSimilarUsers*

Na kraju funkcija *getCollabRecommendedPosts* koja kao argumente prima liste vraćene iz prethodno objašnjenih funkcija. Ova funkcija generira listu preporučenih objava životinja za trenutnog korisnika temeljene na objavama koje su označili slični korisnici, a koje trenutni korisnik još nije lajkao kao što je prikazano na slici 4.41.

```
Future<List<PetPostModel>> getCollabRecommendedPosts(List<String> similarUsers, List<String> currentUserLikes) async {
  Set<String> recommendedPostIds = {};

  for (String userId in similarUsers) {
    QuerySnapshot userLikesSnapshot = await FirebaseFirestore.instance
        .collection('user_likes')
        .doc(userId)
        .collection('liked_posts')
        .get();

    List<String> likedPosts = userLikesSnapshot.docs.map((doc) => doc.id).toList();
    for (var postId in likedPosts) {
      if (!currentUserLikes.contains(postId)) {
        recommendedPostIds.add(postId);
      }
    }
  }

  List<PetPostModel> recommendedPosts = [];
  for (String postId in recommendedPostIds) {
    DocumentSnapshot postSnapshot = await FirebaseFirestore.instance.collection('pet_posts').doc(postId).get();
    if (postSnapshot.exists) {
      final data = postSnapshot.data() as Map<String, dynamic>;
      recommendedPosts.add(PetPostModel(
        postId: postId,
        userId: data['userId'],
        isAdopted: data['isAdopted'],
        name: data['name'],
        age: data['age'],
        weight: data['weight'],
        breed: data['breed'],
        imageUrl: data['imageUrl'],
        type: data['type'],
        petAddress: data['petAddress'],
      ));
    }
  }
  return recommendedPosts;
}
```

Sl. 4.41 Funkcija *getCollabRecommdedPosts*

4.2.11. Programska implementacija odjave korisnika

Odjava korisnika je implementirana na jednostavan način, pritiskom na gumb *Sign Out* pomoću izbornika ladice poziva se funkcija *signOut* koja pomoću instance *FirebaseAuth* klase poziva funkciju *signOut* koja odjavljuje prijavljenog korisnika iz sustavakao što je prikazano na slici 4.42.

```
Future<void> signOut() async {
  await _auth.signOut();
}
```

Sl. 4.42 Funkcija *signOut*

5. NAČIN KORIŠTENJA I ISPITIVANJE VIŠEPLATFORMSKE APLIKACIJE S ANALIZOM REZULTATA

U ovom poglavlju objašnjen je način korištenja višeplatformske aplikacije, aplikacija je ispitana u par slučajeva, te su rezultati ispitivanja analizirani.

5.1. Način korištenja višeplatformske aplikacije

Kada se aplikacija prvi put pokrene, korisniku se prikazuje zaslon za prijavu u aplikaciju gdje korisnik unosi adresu e-pošte i lozinku kako bi dobio pristup aplikaciji. Ukoliko korisnik nema stvoren korisnički račun, pritiskom na tekstualni gumb *Sign up* usmjerava se na zaslon za registraciju. Na zaslonu za registraciju da bi korisnik kreirao korisnički račun potrebno je unijeti adresu e-pošte, lozinku, te potvrditi lozinku, ukoliko je adresa e-pošte valjanog oblika, lozinka minimalno 6 znakova, i polja za lozinku i potvrde lozinke jednaka, pritiskom na gumb *Register* korisnik se uspješno registrira. Nakon što se korisnik registrira, automatski se prijavljuje u aplikaciju i usmjerava na zaslon za kompletiranje profila, gdje je potrebno unijeti sljedeće parametre: ime, prezime, adresu, broj telefona, omiljenu životinju i omiljenu pasminu životinje. Nakon što je profil kompletiran korisnik se usmjerava na početni zaslon aplikacije.

Na početnom zaslonu aplikacije izlistane su objave životinja za udomljivanje, gdje korisnik ima mogućnost filtrirati prikazanu listu po pasmini životinje pomoću trake za pretraživanje. Korisnik također može filtrirati listu, tako da se prikazuju samo objave životinja koje dosada nisu još udomljene, pomoću potvrdnog okvira *Show already adopted*. Korisnik ima mogućnost označiti svaku objavu kao da mu se sviđa pritiskom na ikonu praznog srca ispod objave koja mu se sviđa, prilikom pritiska na ikonu, ikona srca se mijenja u puno srce te mijenja boju u crvenu. Korisnik također ima mogućnost pregledati detalje svake objave pritiskom na gumb *View Details* ispod objave, prilikom čega se otvara zaslon s detaljima objave.

Zaslon s detaljima objave prikazuje ime, pasminu, težinu, starost, lokaciju i sliku životinje, kao i ime i broj telefona korisnika koji je kreirao objavu. Na ovom zaslonu korisnik ima mogućnost poslati zahtjev za udomljivanje pritiskom na gumb *Request Adoption*, te prikazati lokaciju životinje na zaslonu s kartom pritiskom na dugme *Show on Map*. Na zaslonu s kartom, lokacije životinja označene su markerima crvene boje, dok je lokacija trenutno prijavljenog korisnika označena s markerom plave boje. Pritiskom na crveni marker otvara se donji modalni list koji prikazuje sliku i ime životinje koja se nalazi na navedenoj lokaciji, kao i tri gumba *View Details*, koji korisnika ponovno usmjerava na zaslon s detaljima objave, *Cancel* koji zatvara donji modalni

list, i *Get Route* gumb. Pritiskom na *Get Route* gumb pojavljuje se kontejner sadržaja koji prikazuje sliku, ime životinje, udaljenost do lokacije životinje i na karti na zaslonu se iscrtava ruta od korisnikovog markera do markera životinje. Kontejner također sadrži i dva gumba: *View Details* i *Close*. Korisnik na početnom zaslonu može pristupiti zaslonu za kreiranje nove objave pritiskom na plutajući gumb, koji se širi i otkriva dva nova gumba: *Open map* i *Create post*. Gumb *Open map* na pritisak otvara zaslon s kartom, a gumb *Create post* usmjerava korisnika na zaslon za kreiranje objave. Da bi kreiranje objave bilo moguće potrebno je učitati sliku životinje iz galerije mobilnog uređaja, te popuniti ostale parametre koji se traže: ime, pasmina, vrsta, težina, starost i lokacija životinje. Nakon što je sve prethodno navedeno pruženo pritiskom na gumb *Submit Post* otvara se dodatni dijalog koji pita korisnika je li siguran da želi objaviti ovu objavu, gdje korisnik može odustati pritiskom na *Cancel* gumb ili nastaviti s objavljivanjem pritiskom na *Submit* gumb.

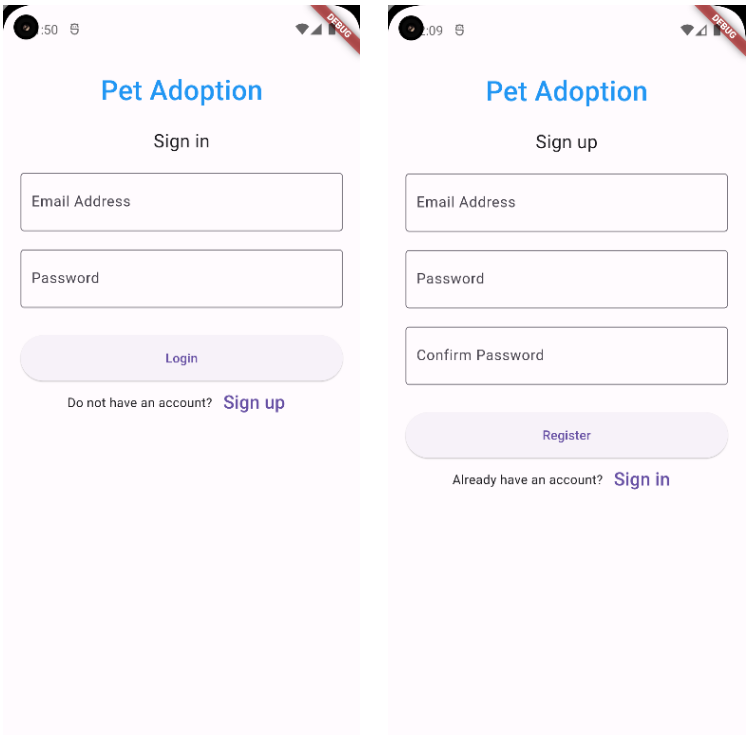
Aplikacija također posjeduje izbornik ladice implementiran pomoću kojeg je olakšana navigacija do određenih zaslona. Izbornikom ladice moguće je pristupiti sljedećim zaslonima: *Home*, *Profile*, *Chat*, *My Requests*, *Map*. *Home* vraća korisnika na početni zaslon. *Profile* usmjerava korisnika na zaslon na kojemu je moguće ažurirati podatke korisničkog računa. *Chat* usmjerava korisnika na zaslon na kojemu su izlistana imena i prezimena korisnika registriranih u aplikaciji, te dodatnim pritiskom na ime i prezime korisnik se usmjerava na zaslon koji predstavlja sobu za razgovor između korisnika. Zatim *My Requests* usmjerava korisnika na zaslon koji je predviđen za upravljanje zahtjevima. Na ovom zaslonu korisnik ima mogućnost odobriti ili odbiti pristigle zahtjeve za objave ljubimaca za udomljivanje koje je kreirao, korisnik također ima uvid u zahtjeve koje je prethodno obradio, zahtjeve koje je poslao koji su još na čekanju tj. obrađuju se, te zahtjeve koji predstavljaju povijest korisnikovog udomljivanja životinja. *Map* korisnika usmjerava na zaslon s kartom, te je od tada funkcionalnost ista kao što je i prethodno objašnjeno.

Na izborniku ladice na dnu se nalazi gumb *Sign Out* koji odjavljuje korisnika iz aplikacije, te ga automatski usmjerava na zaslon za prijavu. Na početnom zaslonu također postoji donji navigacijski bar preko kojeg se korisnik može usmjeravati na *Home* i *Recommended* zaslon. Zaslon *Recommended* korisniku prikazuje personalizirane preporuke objava slične objavama koje je prethodno označio kao da mu se sviđaju.

5.2. Prikaz rada višeplatformske aplikacije

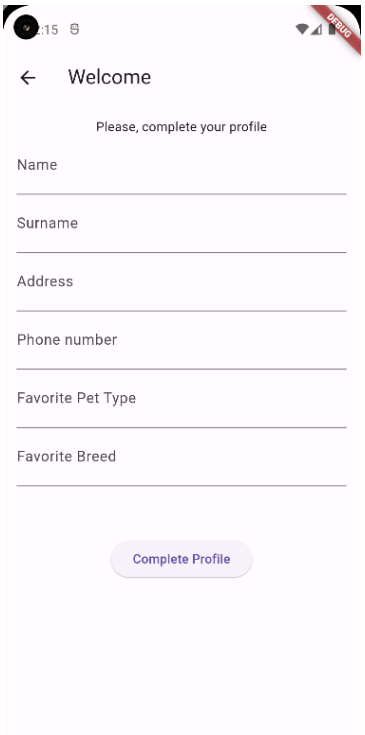
U sljedećem potpoglavlju prikazani su primjeri rada aplikacije, s naglaskom na ključne funkcionalnosti i interakcije korisnika s aplikacijom. Slike koje se koriste za prikaz rada aplikacije snimljene su na Pixel 5 API 31 android emulatoru.

Pri prvom pokretanju aplikacije korisniku se prikazuje zaslon za prijavu korisnika prikazan na slici 5.1. Potrebno je da korisnik unese adresu e-pošte i lozinku, te pritisne gumb *Login*, nakon čega se, ukoliko postoji korisnički račun s navedenim podatcima pohranjen u bazi podataka, dopušta korisniku pristup aplikaciji. Ukoliko je pružena adresa e-pošte u krivom formatu korisnik se obavještava porukom na zaslonu *'Incorrect e-mail provided'*, ukoliko ne postoji korisnički račun povezan s pruženim podatcima ispisuje se *'User not found'*, i ukoliko dođe do neočekivanih iznimaka ispisuje se poruka *'An error occured. Please try again later.'*.



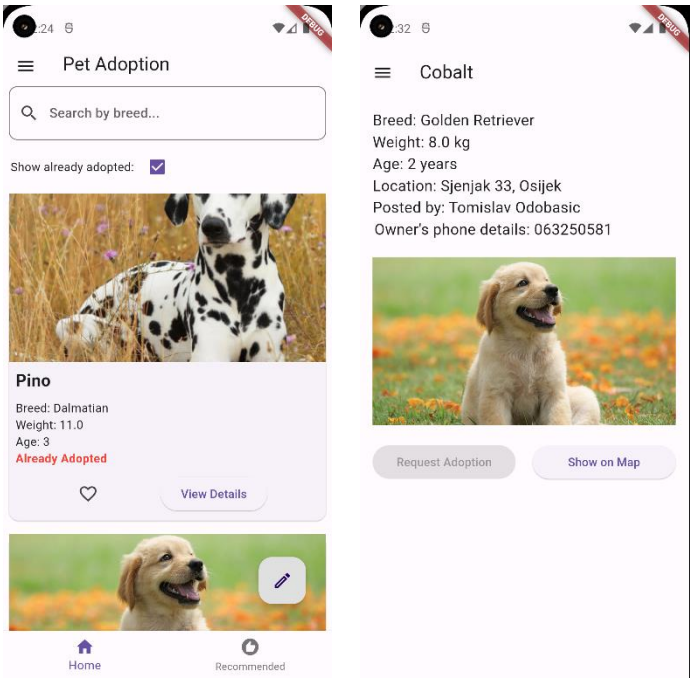
Sl. 5.1 Zaslon za prijavu i zaslon za registraciju

Ukoliko korisnik nema stvoren račun, klikom na tekstualni gumb *Sign up* usmjerava se na zaslon za registraciju. Da bi korisnik stvorio račun potrebno je popuniti tekstualna polja i to redom: adresa e-pošte, lozinka, potvrda lozinke. Nakon što je korisnik popunio polja, prilikom čega adresa e-pošte posjeduje odgovarajući format, te je lozinka minimalno 6 simbola duga, pritiskom na gumb *Register* stvara se novi korisnički račun s upisanim podatcima. Registracija korisnika automatski prijavljuje u aplikaciju, te se korisnik prilikom prve prijave usmjerava na zaslon za kompletiranje profila prikazan slikom 5.2. Zaslon za kompletiranje profila od korisnika traži unos dodatnih podataka, kao što su redom: ime, prezime, adresa, broj telefona, omiljena vrsta životinje i omiljena vrsta pasmine. Nakon što se polja popunu s odgovarajućih podatcima, te pritisne gumb *Complete Profile* korisnikov prethodno kreiran račun koji je pohranjen u bazi podataka se ažurira s novim podatcima, te se korisniku omogućuje pristup aplikaciji usmjeravanjem na početni zaslon.



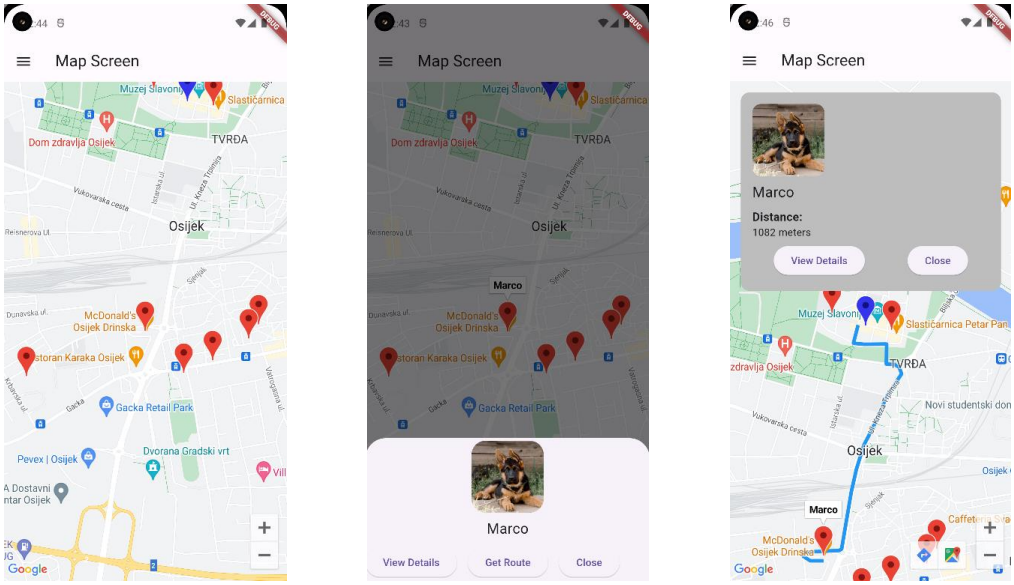
Sl. 5.2 Zaslon za kompletiranje profila

Jednom kada se korisnik nalazi na početnom zaslonu (slika 5.3) ima mnoge mogućnosti. Iz baze podataka dohvaćaju se sve objave kreirane od strane korisnika koji su prijavljeni u aplikaciji te se prikazuju na početnom zaslonu u obliku liste koja se može pomicati. Svaka objava u listi prikazuje sliku, ime, pasminu, težinu i starost životinje, te također ukoliko je objavljena životinja već udomljena, dodatno se ispisuje crvenim slovima *Already Adopted*. Korisnik pomoću trake za pretraživanje i okvira za potvrdu ima mogućnost filtrirati prikazujuću listu. Svaka od objava se može označiti pritiskom na gumb u obliku ikone, te pritiskom na gumb *View Details* korisnik se usmjerava na zaslon koji prikazuje detalje objave (slika 5.3) kao što su: ime, pasmina, težina, starost i lokacija životinje, kao i podatke o korisniku koji je kreirao objavu. Ispod detalja na ovom zaslonu nalaze se dva gumba, *Request Adoption* i *Show on Map*. Gumb *Request Adoption* šalje zahtjev za udomljivanje korisniku koji je kreirao objavu, ukoliko je prijavljeni korisnik vlasnik objave gumb je onemogućen, kako bi se spriječilo da korisnik sam sebi pošalje zahtjev za udomljivanje. Drugi gumb *Show on Map* pritiskom usmjerava korisnika na zaslon karte gdje se početna pozicija kamere postavlja na lokaciju životinje preko čije objave je korisnik usmjeren na zaslon.



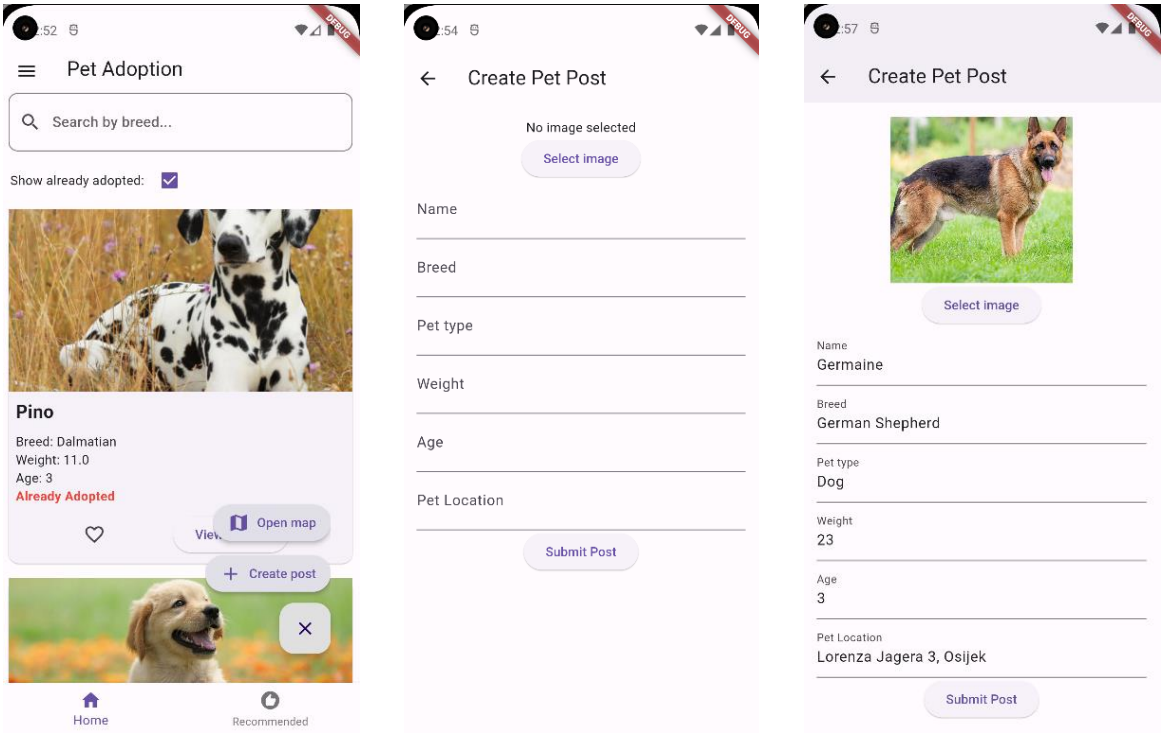
Sl. 5.3 Početni zaslon i zaslon sa detaljima objave

Na zaslonu koji prikazuje kartu, prikazan na slici 5.4, prikazuju se markeri crvene boje na lokacijama svih životinja za koje su kreirane objave, te se prikazuje marker plave boje na lokaciji trenutno prijavljenog korisnika. Svaki od crvenih markera se može kliknuti, prilikom čega se pojavljuje donji modalni list (slika 5.4) koji sadrži sliku i ime životinje koja se nalazi na toj lokaciji kao i tri gumba. *View Details* koji ima istu funkciju kao što je ranije objašnjeno, *Close* koji zatvara donji modalni list, te *Get Route* pritiskom kojeg se pojavljuje kontejner s podatcima životinje i udaljenost od lokacije korisnika do lokacije životinje, te se iscrtava potencijalna ruta kojom korisnik može doći do lokacije životinje kao što je prikazano slikom 5.4.



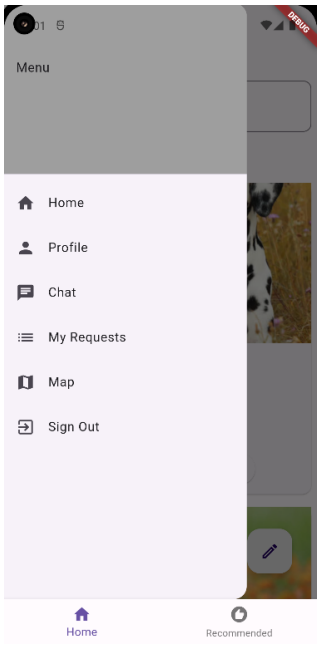
Sl. 5.4. Zaslon s prikazom karte i njegove funkcionalnosti

Korisnik također ima mogućnost kreiranja nove objave na zaslonu za kreiranje objave do kojeg se može usmjeriti koristeći plutajući gumb na početnom zaslonu (slika 5.5), te pritiskom na dodatni gumb *Create post*. Na ovom zaslonu od korisnika se traži učitavanje slike iz galerija mobilnog uređaja te unos podataka životinje navedenim redoslijedom: ime, pasmina, vrsta, težina, starost, lokacija kao što je prikazano na slici 5.5. Na slici 5.5 također je prikazan primjer ispunjenih podataka za kreiranje objave. Nakon što su uneseni svi traženi parametri u odgovarajućem obliku, pritiskom na gumb *Submit Post* pojavljuje se dijalog obavijesti koji pita korisnika je li siguran da želi stvoriti objavu s ovim detaljima, gdje korisnik može pritisnuti *Cancel* što poništava sve unesene parametre ili *Submit* što dovršava radnju kreiranja objave.



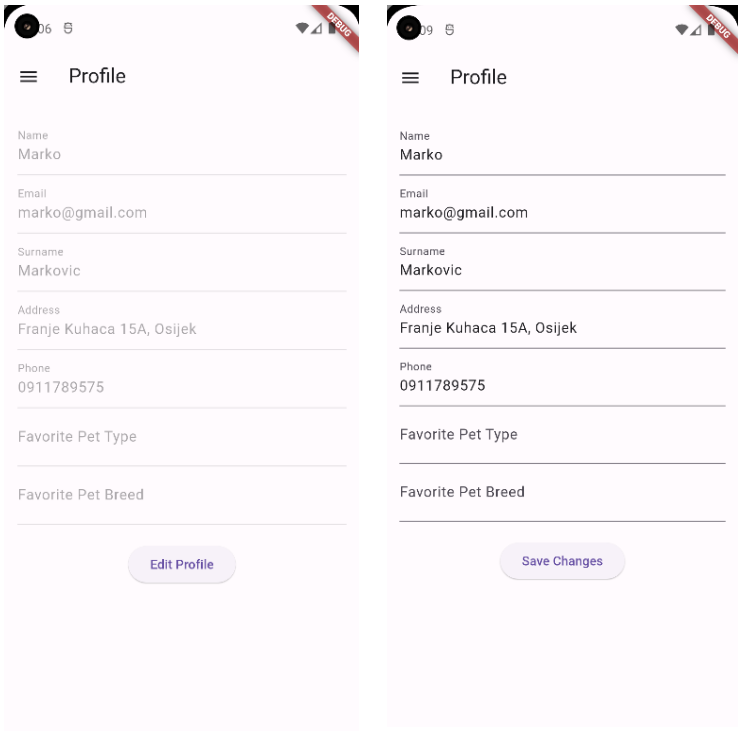
Sl. 5.5 Navigacija do i prikaz zaslona za kreiranje objave

Navigacija kroz aplikaciju i pristupanje raznim zaslonima aplikacije dodatno je omogućeno implementacijom izbornika ladice kao što je prikazano na slici 5.6.



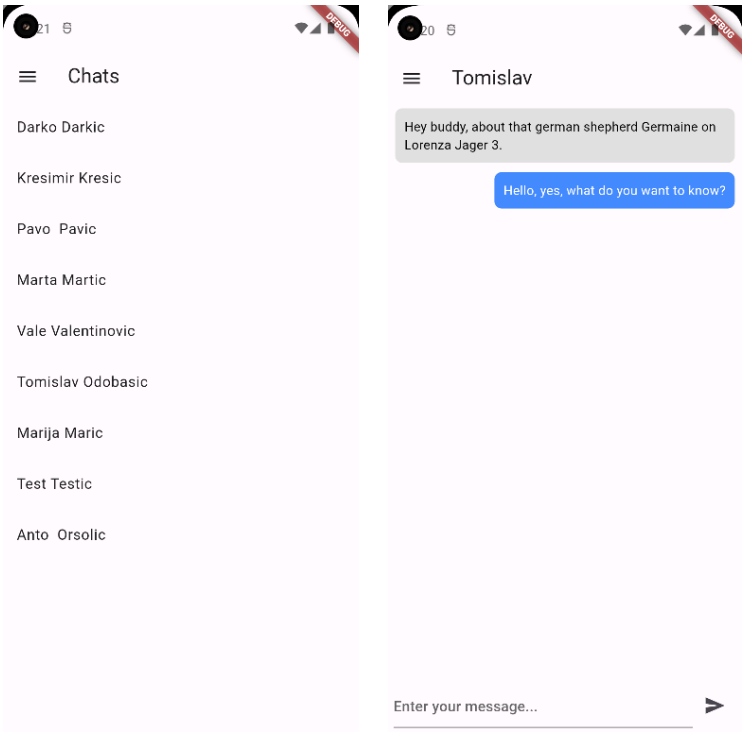
Sl. 5.6 Prikaz izbornika ladice

Klikom na *Home* pločicu na izborniku ladice korisnik se usmjerava na početni zaslon. Klikom na *Profile* pločicu korisnik se usmjerava na zaslon koji prikazuje detalje o korisničkom profilu kao što je prikazano na slici 5.7. Na ovom zaslonu korisnik ima mogućnost ažuriranja profila. Pritiskom gumba *Edit Profile* tekstualna polja se omogućavaju te ih se može urediti, te nakon uređivanja klikom na gumb *Save Changes* podatci profila se ažuriraju.



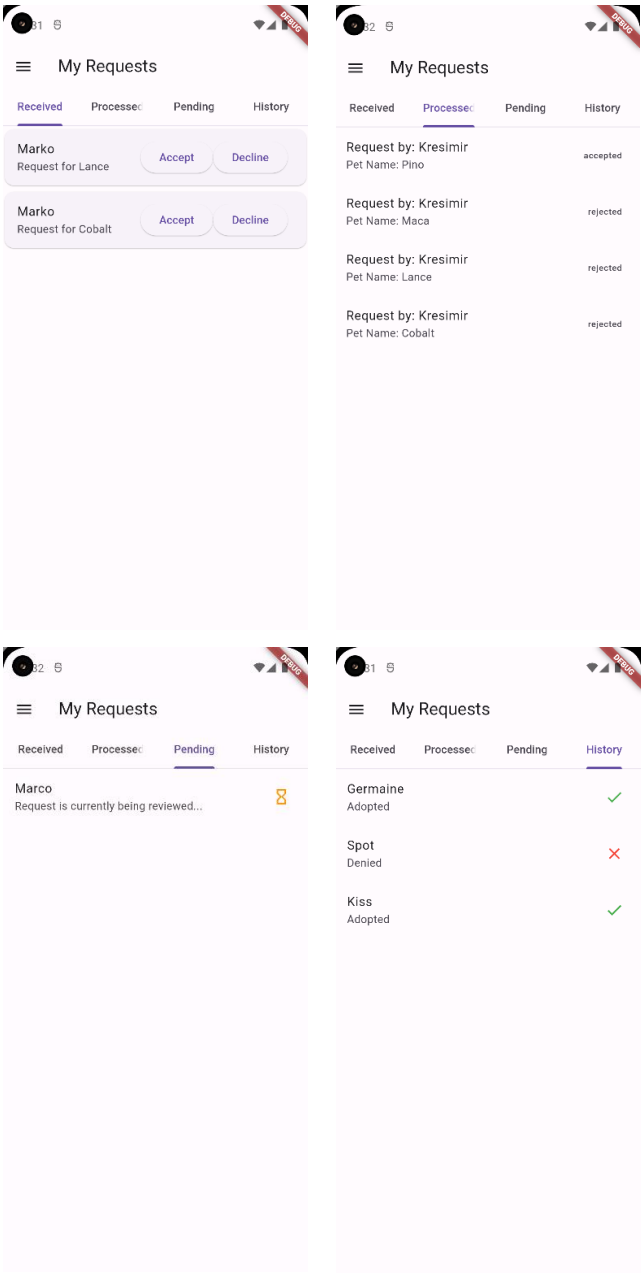
Sl. 5.7 Prikaz zaslona s detaljima korisničkog profila

Klikom na pločicu *Chat* na izborniku ladice korisnik se usmjerava na zaslon koji prikazuje registrirane korisnike u aplikaciji s kojima je moguće započeti razgovor. Klikom na pločicu bilo kojeg korisnika, korisnik se usmjerava na zaslon koji predstavlja sobu za razgovor između ova dva korisnika kao što je prikazano na slici 5.8. Funkcionalnost chat sustava je implementirana u aplikaciju kako bi se omogućila bolja i lakša međusobna komunikacija između korisnika, te kako bi se poboljšalo korisničko iskustvo. U sobi za razgovor korisnici mogu razmjenjivati dodatne informacije i detalje, te formirati dogovore oko preuzimanja životinja koje žele udomiti.



Sl. 5.8 Prikaz i primjer funkcionalnosti chat sustava

Klikom na *My Requests* pločicu na izborniku ladice korisnik se usmjerava na zaslon koji je predviđen za upravljanje zahtjevima. Na ovom zaslonu nalaze se 4 taba. U prvom tabu korisniku se prikazuju pristigli zahtjevi za udomljivanjem poslani za neku od životinja za koje je korisnik kreirao objavu, korisnik ove zahtjeve može odobriti ili odbiti. U drugom tabu korisniku se prikazuju zahtjevi koje je ranije obradio. U trećem tabu korisniku se prikazuju zahtjevi koje je korisnik poslao drugim korisnicima, a koji su još na čekanju. U četvrtom tabu korisniku se prikazuje povijest svih zahtjeva u obliku životinja koje je udomio ili koje mu drugi korisnici nisu odobrili. Sve opisane funkcionalnosti svakog taba prikazane su na slici 5.9.



Sl. 5.9 Prikaz tabova *My Requests*

Klikom na pločicu *Map* na izborniku ladice korisnik se ponovno usmjerava na zaslon karte, te naposljetku klikom na pločicu *Sign Out* korisnik se odjavljuje iz aplikacije i usmjerava se na zaslon za prijavu.

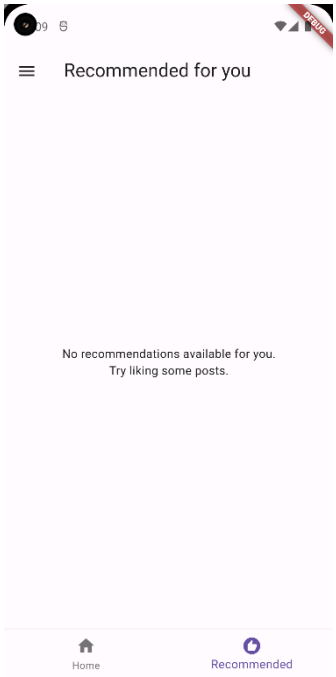
5.3. Ispitivanje rada višeplatformske aplikacije

U nastavku će biti opisani slučajevi korištenja višeplatformske mobilne aplikacije u svrhu ispitivanja kvalitete sustava preporuka, korisničkog iskustva i korištene programske arhitekture. Aplikacija je ispitivana na android emulatoru Pixel 5 API 31, gdje se slični rezultati ispitivanja očekuju i na ostalim platformama, poput iOS-a.

5.3.1. Ispitivanje višeplatformske aplikacije s gledišta sustava preporuka

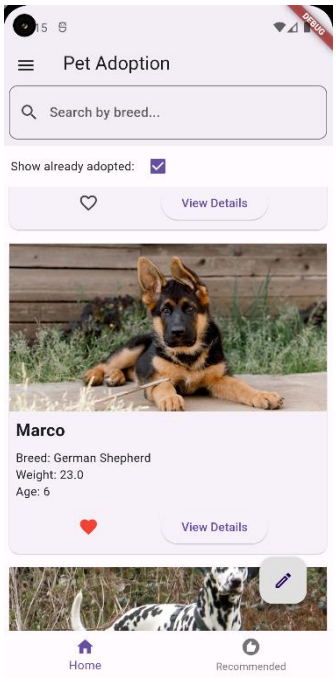
Ispitivanje aplikacije koje se odnosi na sustav preporuka u nastavku bit će provedeno u tri različita slučaja za obje metode filtriranja pomoću kojih je ostvaren sustav preporuka, prvo će biti objašnjena metoda filtriranja prema sadržaju.

U prvom slučaju prikazuje se zaslon s preporukama kada korisnik tek počne koristiti aplikaciju, odnosno još nije označio niti jednu objavu sa sviđanjem. U drugom slučaju prikazat će se zaslon s preporukama nakon što je korisnik označio jednu objavu, te u trećem slučaju nakon što je korisnik označio više objava. Na slici 5.10 prikazan je zaslon preporuka kada korisnik nije označio niti jednu objavu sa „sviđa mi se“, te se u tom slučaju korisniku na zaslonu ispisuje tekst *'No recommendations available for you. Try liking some posts.*'



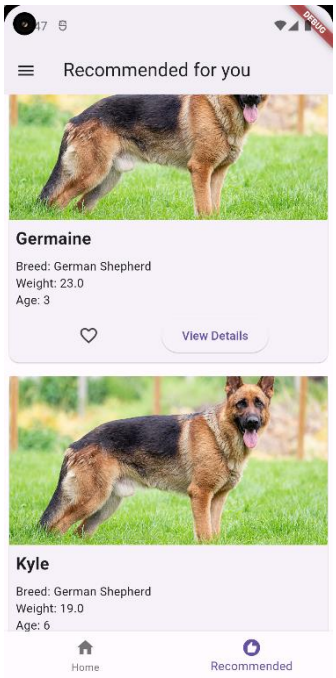
Sl. 5.10 Prikaz zaslona preporuka filtriranja na temelju sadržaja u slučaju kada korisnik nema označenih objava

U sljedećem slučaju opisana je situacija u kojoj korisnik označava samo jednu objavu. U primjeru je sa sviđa mi se označen pas Marco, pasmine *'German Shepherd'*, težine 23.0 i starosti 6 kao što je prikazano na slici 5.11.



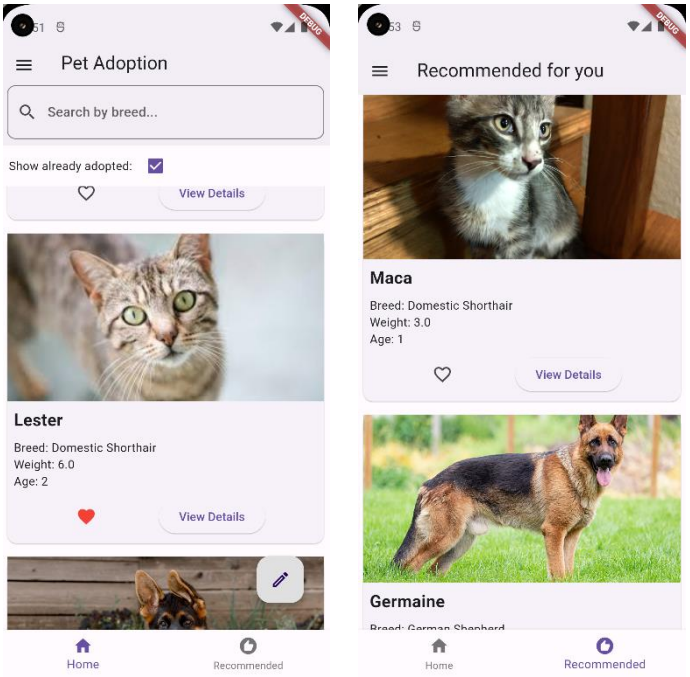
Sl. 5.11 Označena objava psa Marco

Nakon što je Marco označen, korisnik se preko donjeg navigacijskog bara usmjerava na zaslon za preporuke, te mu se prikazuju preporuke u odnosu na ono što je označio. Na slici 5.12 vidljiv je na zaslonu preporuka sada novo ažurirani popis personaliziranih preporuka za korisnika. Korisniku su preporučeni psi Germaine i Kyle, obadva psa su pasmine *'German Shepherd'*, što potvrđuje sličnost s prethodno označenim psom Marco.



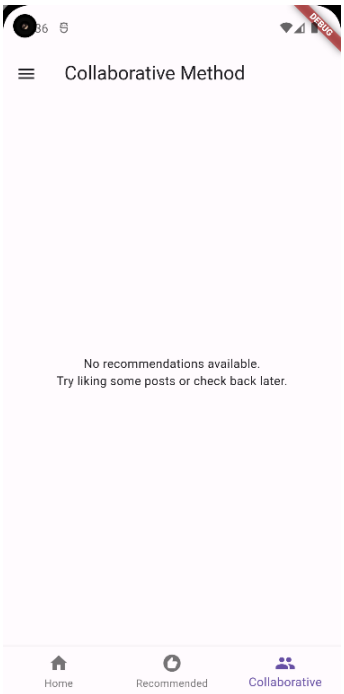
Sl. 5.12 Prikaz zaslona preporuka filtriranja na temelju sadržaja nakon što je korisnik označio objavu sa „sviđa mi se“

Za ispitivanje trećeg slučaja označava se još jedna objava, ovaj puta mačka Lester koja je pasmine *'Domestic Shorthair'*, što znači da je su za sada označene dvije objave. Na slici 5.13 vidljivo je kako zaslon za preporuke sada uz psa Germaine preporuča i mačku Maca koja je pasmine *'Domestic Shorthair'* što također potvrđuje sličnost s mačkom Lester.



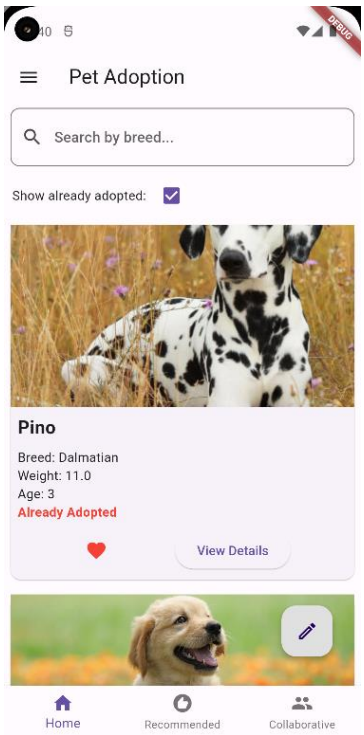
Sl. 5.13 Označavanje objave mačke Lester sa „sviđa mi se“ i prikazivanje novo ažurirane liste personaliziranih preporuka

Slijedi ispitivanje sustava preporuka koji je implementiran metodom kolaborativnog filtriranja. Analogno prethodnim slučajevima ispitivanja, prvo se prikazuje zaslon s preporukama kada korisnik tek počne koristiti aplikaciju, te je na slici 5.14 prikazan zaslon preporuka.



Sl. 5.14 Prikaz zaslona preporuka kolaborativnog filtriranja u slučaju kada korisnik nema označenih objava

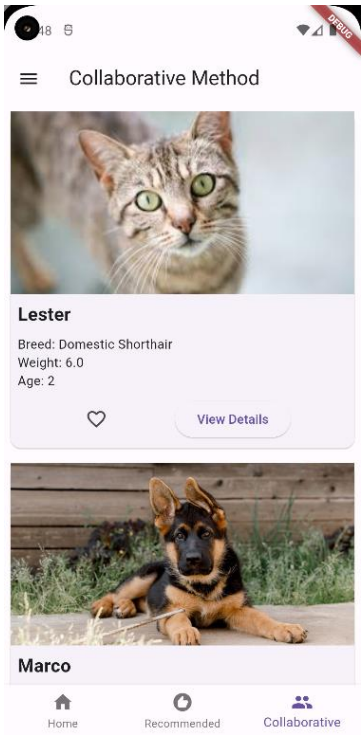
U sljedećem slučaju opisana je situacija u kojoj korisnik označava objavu psa Pino, pasmine '*Dalmatian*', težine 11.0 i starosti 3 kao što je prikazano na slici 5.15.



Sl. 5.15 Označena objava psa Pino

Nakon što je pas Pino označen, korisnik se ponovno usmjerava na zaslon za prikaz preporuka generiranih kolaborativnom metodom filtriranja. Sustav pronalazi korisnike koji su također

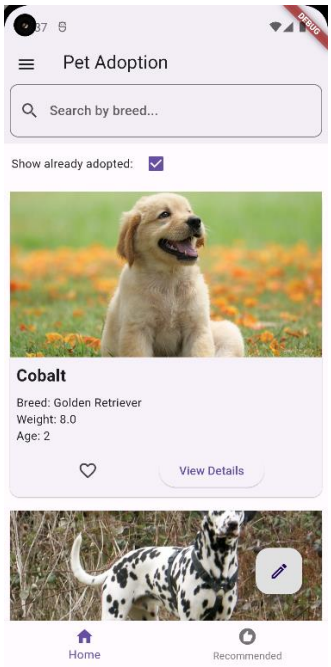
označili objavu psa Pino kao da im se sviđa, te generira preporuke na temelju ostalih objava koje se ovim korisnicima sviđaju i prikazuje ih trenutnom korisniku kao što je prikazano na slici 5.16. Na slici se može vidjet da se korisniku preporučuju objave mačke Lester i psa Marco, što znači da se korisnicima kojima se sviđa objava psa Pino također sviđaju i ove objave.



Sl. 5.16 Prikaz zaslona preporuka kolaborativnog filtriranja nakon što je korisnik označio objavu sa „sviđa mi se“

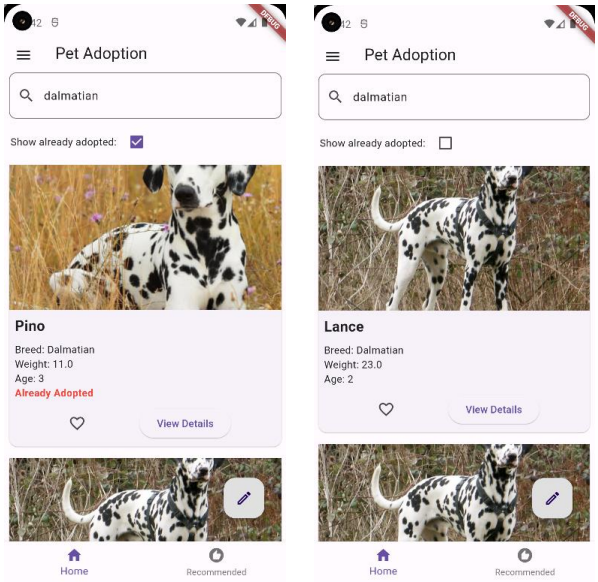
5.3.2. Ispitivanje višeplatformske aplikacije s gledišta korisničkog iskustva

Ispitivanje aplikacije koje se odnosi na korisničko iskustvo također će biti provedeno u par slučajeva. Kroz slučajeve obratit će se pozornost na intuitivnost korištenja aplikacije, koliko je intuitivna funkcionalnost pretrage objava i sortiranje liste na početnom zaslonu, navigacija kroz aplikaciju, te obavijesti aplikacije. Kao što je ranije opisano, korisnik na početnom zaslonu ima mogućnost sortiranja liste objava pomoću trake za pretraživanje i potvrdnog okvira koji se nalaze pri vrhu zaslona. Na slici 5.17 prikazan je početni zaslon s prikazom objava bez korištenja dodatnih mogućnosti za filtriranje, gdje je vidljivo da se prikazuje pas pasmine *'Golden Retriever'* i pas pasmine *'Dalmatian'*.



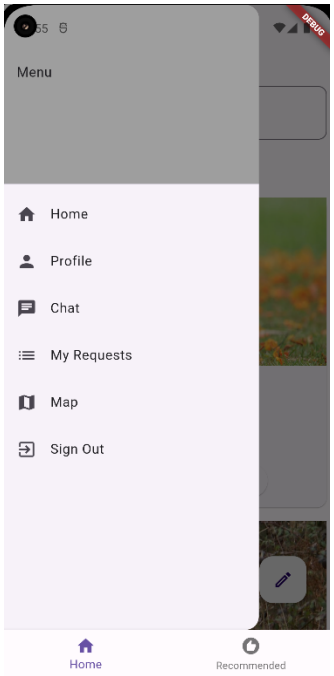
Sl. 5.17 Početni zaslona s prikazom liste objava bez filtracije

Na slici 5.18 prikazana su dva slučaja, prvi kada korisnik u traku za pretraživanje upiše *'dalmatian'* na osnovu čega se lista objava koja se prikazuje na početnom zaslonu filtrira, te prikazuje samo objave životinja koje su prethodno upisane pasmine. I drugi slučaj kada se uz dodatnu opciju filtriranja pomoću trake za pretraživanje koristi i potvrdni okvir. Potvrdni okvir ukoliko je označen lista objava prikazuje objave životinja i koje su već udomljene i koje nisu još udomljene. Početno stanje potvrdnog okvira je uvijek označeno, te je zadani prikaz liste uvijek takav da prikazuje i udomljene i neudomljene životinje. Na slici 5.18 je također prikazana prethodno filtrirana lista s dodatnim filtriranjem pomoću potvrdnog okvira. Uklanja se oznaka potvrdnog okvira, te lista zatim prikazuje samo objave životinja koje nisu udomljene.



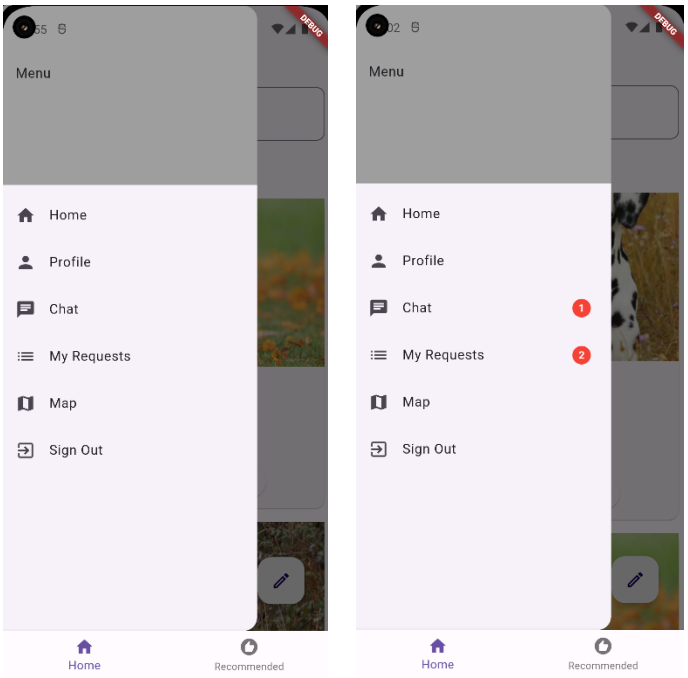
Sl. 5.18 Mogućnosti filtriranja liste prikaza pomoću trake za pretraživanje i potvrdnog okvira

Navigacija kroz aplikaciju olakšana je korisniku implementacijom izbornika ladice pomoću kojeg korisnik na bilo kojem zaslonu može jednostavno pristupiti bilo kojem drugom zaslonu. Ovakav način navigacije korisniku je prilično intuitivan i jednostavan. Pri otvaranju izbornika ladice postoji 6 različitih pločica izbornika pomoću kojih korisnik ima mogućnost usmjeriti se na bilo koji zaslon aplikacije klikom na određenu pločicu. Svaka pločica izbornika sadrži ikonu i tekst, koji zajedno opisuju o kojem se zaslonu radi. Izbornik ladice prikazan je na slici 5.19.



Sl. 5.19 Izbornik ladice za jednostavnu navigaciju kroz aplikaciju

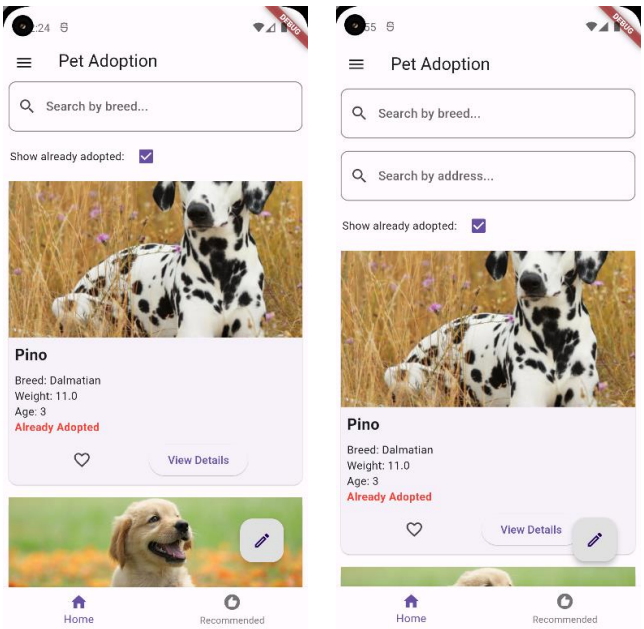
Uzimajući u obzir da aplikacija podržava chat sustav i sustav upravljanja zahtjevima, omogućeno je intuitivno upućivanje korisnika na pristigle obavijesti. Pristigle obavijesti prikazuju se u obliku okruglog crvenog indikator s brojem obavijesti u sredini. Unutar ovog indikatora prikazuje se broj zahtjeva vezan za poruke i zahtjeve koje je korisnik primio. Na slici 5.20 prikazan je prvo izbornik ladice kada korisnik nema pristiglih obavijesti, a zatim je prikazan izbornik ladice kada korisnik ima jednu obavijest poruke i dvije obavijesti, odnosno dva zahtjeva koja čekaju obradu poslana od drugih korisnika.



Sl. 5.20 Izbornik ladice kada korisnik nema obavijesti i kada ima obavijesti

5.3.3. Ispitivanje višeplatformske aplikacije s gledišta programske arhitekture

Iako je ispitivanje i analiza na temelju programske arhitekture manje vizualna, slučaj ispitivanja programske arhitekture bit će odrađen tako što će se implementirat nova funkcionalnost na početni zaslon. Bit će dodana nova traka za pretraživanje na početnom zaslonu ispod već postojeće trake. Nova traka će korisniku omogućiti filtriranje objava životinja po adresi. Na slici 5.21 prikazan je početni zaslon bez modifikacije, te početni zaslon nakon implementacije nove funkcionalnosti. Da bi nova funkcionalnost radila, kreirana je nova traka za pretraživanje koja je smještena ispod stare trake.



Sl. 5.21 Početni zaslon prije i poslije dodavanja nove funkcionalnosti

Dodan je novi *StateProvider* koji će slušati vrijednost upita nove trake za pretraživanje *newSearchQueryProvider* prikazan slikom 5.22.

```
final newSearchQueryProvider = StateProvider.autoDispose<String>((ref) => '');
```

Sl. 5.22 *Provider newSearchQueryProvider*

Zatim je modificirana logika za dohvaćanje, odnosno filtriranje liste objava tako da podržava i filtriranje po adresi. Na slici 5.23 prikazana je logika za filtriranje liste objava prije, a na slici 5.24 logika za filtriranje liste objava poslije modifikacije.

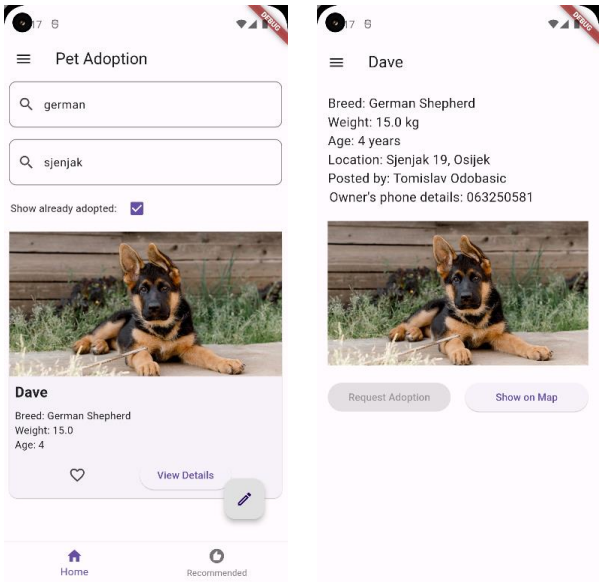
```
final filteredPetPostsProvider = Provider.autoDispose<List<PetPostModel>>((ref) {
  final searchQuery = ref.watch(searchQueryProvider);
  final checkBoxList = ref.watch(checkBoxFilteredProvider);

  if (searchQuery.isEmpty) {
    return checkBoxList;
  } else {
    return checkBoxList
        .where((petPost) => petPost.breed.toLowerCase().contains(searchQuery.toLowerCase()))
        .toList();
  }
});
```

Sl. 5.23 Logika za filtriranje liste objava po pasmini

```
final filteredPetPostsProvider = Provider.autoDispose<List<PetPostModel>>((ref) {
  final searchQuery = ref.watch(searchQueryProvider);
  final checkBoxList = ref.watch(checkBoxFilteredProvider);

  final newSearchQuery = ref.watch(newSearchQueryProvider); // Novo

  if (searchQuery.isEmpty && newSearchQuery.isEmpty) {
    return checkBoxList;
  } else {
    final filteredByBreed = checkBoxList.where((petPost) {
      return searchQuery.isEmpty || petPost.breed.toLowerCase().contains(searchQuery.toLowerCase());
    }).toList();

    final filteredByAddress = filteredByBreed.where((petPost) {
      return newSearchQuery.isEmpty || petPost.petAddress.toLowerCase().contains(newSearchQuery.toLowerCase());
    }).toList();

    return filteredByAddress;
  }
});
```

Sl. 5.24 Logika za filtriranje liste objava po pasmini i po adresi

Naposljetku na slici 5.25 prikazan je primjer filtriranja objave po obadva parametra, po pasmini i po adresi. U traku za pretraživanje po pasmini upisuje se *'german'* što filtrira listu i prikazuje objave životinja koje su pasmine *'German Shepherd'*, te se zatim u traku za pretraživanje po adresi upisuje 'sjenjak' što prethodnu listu dodatno filtrira i prikazuje samo pse pasmine *'German Shepherd'* koji se nalaze na Sjenjaku.



Sl. 5.25 Primjer filtriranja liste objava pomoću novo implementirane funkcionalnosti

5.4. Analiziranje rezultata ispitivanja višeplatformske aplikacije

U nastavku će biti analizirani rezultati ispitivanja višeplatformske mobilne aplikacije iz prethodnog potpoglavlja na temelju sustava preporuka, korisničkog iskustva i programske arhitekture. Analiza je provedena na temelju rezultata dobivenih korištenjem android emulatora Pixel 5 API 31, s očekivanjem da će aplikacija jednako učinkovito funkcionirati i na ostalim platformama.

5.4.1. Analiza ispitivanja sustava preporuka

Analizom rezultata ispitivanja temeljenih na sustavima stvaranja preporuka, cilj je evaluirati učinkovitost i preciznost implementiranog sustava preporuka u aplikaciji. Ispitana su dva sustava preporuka implementirana na dva različita načina filtriranja kroz različite scenarije testiranja, gdje se ponašanje sustava promatra ovisno o interakciji korisnika s objavama. Prvo je ispitan sustav preporuka ostvaren metodom filtriranja na temelju sadržaja, a zatim sustav preporuka ostvaren metodom kolaborativnog filtriranja.

U prvom scenariju ispitivanja sustava preporuka koji je ostvaren metodom filtriranja na temelju sadržaja, prikazuje se zaslon s preporukama u slučaju kada korisnik nije označio ni jednu objavu kao „sviđa mi se“. Zaslon ne prikazuje ni jednu objavu preporuke, nego prikazuje tekst koji obavještava korisnika da trenutno nema generiranih personaliziranih preporuka, gdje je sadržaj teksta „*No recommendations available for you. Try liking some posts*“. Ovakav ishod pokazuje da sustav preporuka ispravno reagira na nedostatak korisničkih podataka, predlažući korisniku ostvarivanje interakcije s objavama kako bi sustav imao na temelju čega generirati personalizirane preporuke.

U drugom scenariju, korisnik označava objavu o psu imena Marco koji je pasmine „*German Shepherd*“, težine 23 i starosti 6. Zaslon preporuka nakon označavanja spomenute objave prikazuje objave pasa iste pasmine „German Shepherd“. Ovaj rezultat potvrđuje da je sustav preporuka uspješno identificirao slične objave na temelju atributa pasmine, vrste, težine i starosti objave koju je korisnik ranije označio.

U trećem scenariju, korisnik označava još jednu objavu, ovaj puta objavu mačke Lester pasmine „*Domestic Shorthair*“, nakon čega se na zaslonu preporuka prikazuje uz prethodno preporučene pse pasmine „*German Shepherd“* nova preporuka za mačku Maca iste pasmine kao mačka Lester. Ovaj ishod uspješno potvrđuje da sustav preporuka ispravno proširuje svoje preporuke na temelju novih preferencija korisnika. Sva tri scenarija potvrđuju da sustav preporuka radi prema očekivanjima, učinkovito identificira slične objave na temelju prethodnih korisnikovih interakcije. Sustav također pokazuje sposobnost dinamičkog prilagođavanja preporuka u realnom vremenu, što je ključna karakteristika dobro dizajniranih sustava preporuka.

Analogno ispitivanju sustava preporuka koji je implementiran metodom filtriranja temeljenoj na sadržaju, zaslon za prikazivanje preporuka kolaborativnog filtriranja u početku ne prikazuje niti jednu objavu preporuke. Korisnik se zatim vraća na početni zaslon gdje označava objavu jedne životinje kao da mu se sviđa, nakon čega sustav u bazi podataka pronalazi druge korisnike koji su također označili ovu objavu, te korisniku generira preporuke temeljene na drugim objavama koje su pronađeni korisnici označili. Ovakav tok stvaranja preporuka potvrđuje da sustav preporuka ostvaren metodom kolaborativnog filtriranja na ispravan način generira preporuke temeljene na kolaborativnom filtriranju.

5.4.2. Analiza ispitivanja korisničkog iskustva

Analizom rezultata ispitivanja temeljenih na korisničkom iskustvu analizira se intuitivnost aplikacije kroz mogućnosti pretraživanja objava, jednostavnost navigacije kroz aplikaciju, te učinkovitost obavijesti za poruke i zahtjeve što dodatno doprinosi intuitivnosti korištenja. Najprije je ispitana intuitivnost prilikom pretraživanja objava na početnom zaslonu.

U početnom scenariju korisniku se prikazuje zadani prikaz objava životinja, odnosno prikazuju sve objave koje se trenutno dohvaćaju iz baze podataka, među kojima se nalaze i objave psa pasmine „*Golden Retriever*“ i psa pasmine „*Dalmatian*“. Prilikom korištenja trake za pretraživanje i unosom teksta 'dalmatian' u traku, prikaz liste objava se trenutačno filtrira i prikazuju se samo relevantne objave pasa koji su pasmine „*Dalmatian*“. Ova funkcionalnost potvrđuje da je pretraga jednostavna i učinkovita, bez potrebe za dodatnim objašnjenima korisniku. Brzo i jasno filtriranje

rezultata poboljšava korisničko iskustvo na način da korisniku omogućuje da u nekoliko koraka pronađe točno ono što traži. Zatim se dodatno na prethodni slučaj koristi i oznaka potvrdnog okvira za dodatno filtriranje. Potvrdni okvir je po zadanom označen, što prikazuje sve objave, uključujući i objave već udomljenih životinja. Nakon što se oznaka okvira ukloni, lista se automatski ažurira i prikazuje samo objave životinja koje nisu udomljene, što pomaže korisniku pri sužavanju pretrage. Vidljivo je da aplikacija pruža napredne filtere bez narušavanja jednostavnosti korištenja, što osigurava korisnicima jednostavnu pretragu kroz objave.

Navigacija kroz aplikaciju olakšana je korisnicima putem izbornika ladice, putem kojeg se omogućuje pristup svim ključnim zaslonima s bilo kojeg zaslona. Dodavanje ikona i tekstualnih oznaka u izborniku ladice čini navigaciju intuitivnim i jednostavnim za korištenje. Na primjer, ikona kućice s tekstom „*Home*“ jasno upućuje korisnika da se radi o navigaciji na početni zaslon aplikacije. Ovakav sustav navigacije poboljšava korisničko iskustvo jer je potreba za višestrukim klikovima ili otežanim i složenim navigacijskim putevima minimalizirana. Jasne ikone i tekstovi prikazani na izborniku ladice doprinose lakoći korištenja aplikacije.

Sustav obavještavanja korisnika o pristiglim novim nepročitanim porukama ili pristiglim zahtjevima implementiran je tako da jasno ukazuje korisniku na točan broj zahtjeva ili poruka i za koju je točno funkcionalnost obavijest. Ovaj dizajn je prepoznatljiv i intuitivan jer korisnici prilikom prvog otvaranja izbornika ladice mogu primijetiti nove obavijesti.

5.4.3. Analiza ispitivanja programske arhitekture

Analiza rezultata ispitivanja temeljenog na programskoj arhitekturi temelji se na analiziranju fleksibilnost i prilagodljivosti sustava kroz implementaciju nove funkcionalnosti na početni zaslon aplikacije. Nova funkcionalnost se odnosi na dodavanje nove trake za pretraživanje koja će omogućuje pretraživanje po adresi. Dodavanje nove funkcionalnost zahtijevalo je minimalne izmjene u postojećem kodu, naime nova traka za pretraživanje dodana je jednostavno unutar postojećeg korisničkog sučelja bez potrebe za velikim promjenama u logici prezentacijskog sloja.

Koristi se novi *riverpod provider newSeachQueryProvider* za praćenje unosa u novu traku za pretraživanje, što jasno ukazuje na odvojenost odgovornosti između različitih dijelova sustava. Vidljivo je kako korisničko sučelje i logika filtriranja rade zajedno na modularan način. Modificiranje logike za filtriranje objava kako bi podržavala novu funkcionalnost, odnosno filtriranje po adresi, također ukazuje na aspekt prilagodljivosti programske arhitekture. Prije dodavanja nove funkcionalnost, logika je filtrirala objave prema pasmini životinja, a nakon

dodavanja nove funkcionalnosti, ista logika je proširena i prilagođena da omogućuje i filtriranje prema adresi.

Prilikom proširivanja logike za filtriranje nije bilo potrebe za potpunom rekonstrukcijom koda, što ukazuje na to da je poslovna logika postavljena na način koji omogućuje lako proširivanje. Provedeno testiranje također pokazuje i skalabilnost arhitekture. Dodavanje novih funkcionalnosti, ne narušava performanse aplikacije niti zahtijeva prevelike promjene u sustavu. Modularna arhitektura uz to što olakšava proširenje aplikacije, također smanjuje složenost održavanja i testiranja aplikacije. Kako je svaka funkcionalnost odvojena i jasno definirana, nova traka za pretraživanje implementirana je i testirana odvojeno od ostalih dijelova aplikacije. Ovaj slučaj ispitivanja pokazuje kako modularna i slojevita arhitektura olakšava proširenje funkcionalnosti bez potrebe za značajnim promjenama u drugim dijelovima aplikacije.

Budući da je aplikacija razvijena u programskom okviru *Flutter*, osim na android platformi, može se bez poteškoća pokretati i na iOS platformi. Iako je aplikacija ispitivana i analizirana na android platformi, slična funkcionalnost, performanse, i korisničko iskustvo očekuje se na obje platforme.

ZAKLJUČAK

Ovaj diplomski rad analizira probleme i izazove udomljivanja kućnih ljubimaca, istražuje mogućnosti višeplatformskog razvoja mobilnih aplikacija, prilagodljivih programskih arhitektura i sustava preporuka koji se temelje na kolaborativnom filtriranju. Na temelju provedene analize, razvijena je mobilna aplikacija koja olakšava proces udomljivanja kućnih ljubimaca pružajući korisnicima intuitivno korisničko iskustvo. Razvijena aplikacija na uspješan način integrira ključne funkcionalnosti poput autentifikacije korisnika, stvaranja korisničkog profila, prikaz informacija o ljubimcima putem objava, unos i kreiranje novih objava životinja, obrada zahtjeva za udomljivanje, razmjenjivanje poruka s drugim korisnicima, te pretraga životinja pomoću *Google Maps* karte. Implementirani sustav preporuka omogućuje korisnicima prikaz personaliziranih preporuka temeljenih na korisnikovom prethodnom iskustvu i interakciji unutar aplikacije. Korišteni razvojni alati, poput programskog okvira *Fluttera* i paketa *Riverpod*, omogućili su izradu modularne i prilagodljive programske arhitekture koja može biti proširena i nadograđena

Ispitivanje aplikacije provedeno je na android platformi pomoću emulatora Pixel 5 API 31. Analizom programskog rješenja utvrđeno je da aplikacija uz zadovoljavanje funkcionalnih i nefunkcionalnih zahtjeva, pruža intuitivno, jednostavno i učinkovito korisničko iskustvo. Aplikacija uspješno omogućuje ubrzavanje procesa udomljivanja kućnih ljubimaca, dok je sustav preporuka pokazao visok stupanj personaliziranja preporuka. S obzirom na to da je aplikacija razvijena u programskom okviru *Flutter*, očekuje se da će sličan učinak biti postignut i na drugim platformama, poput iOS-a, uz usporedivu funkcionalnost, performanse i korisničko iskustvo. Za budući rad na aplikaciji preporučuje se dodatna optimizacija sustava preporuka, uključujući istraživanje naprednijih tehnika filtriranja i personalizacije, a također se preporučuje provođenje daljnjih ispitivanja korisničkog iskustva kako bi se potencijalno identificirale moguće smjernice za daljnje poboljšanje i nadogradnju aplikacije.

LITERATURA

[1] K.J. Scoresby, E.B. Strand, Z. Ng, K.C. Brown, C.R. Stilz, K. Strobel, C.S. Barroso, M. Souza, Pet Ownership and Quality of Life: A Systematic Review of the Literature, Veterinary Sciences, vol. 8, No.12, str. 332, prosinac 2021.

[2] Petfinder [online], dostupno na https://www.petfinder.com/ [pristupljeno 12. lipnja 2024.]

[3] PawBoost [online], dostupno na https://www.pawboost.com/site/how-it-works [pristupljeno 12. lipnja 2024.]

[4] 11pets [online], dostupno na https://www.11pets.com/en/about [pristupljeno 12. lipnja 2024.]

[5] C. M. Pinto, C. Coutinho, From Native to Cross-platform Hybrid Development, International Conference on Intelligent Systems, str. 669-676, Funchal, 2018.

[6] I. Dalmasso, S. K. Datta, C. Bonnet, N. Nikaein, Survey, comparison and evaluation of cross platform mobile application development tools, 9th International Wireless Communications and Mobile Computing Conference, str. 323-328, Sardinia, 2013.

[7] Y. Cheon, C. Chavez, Converting Android Native Apps to Flutter Cross-Platform Apps, International Conference on Computational Science and Computational Intelligence, str. 1898-1904, Las Vegas, 2021.

[8] N. M. Hui, L. B. Chieng, W. Y. Ting, H. H. Mohamed, M. R. Hj Mohd Arshad, Cross-platform mobile applications for android and iOS, 6th Joint IFIP Wireless and Mobile Networking Conference, str. 1-4, Dubai, 2013.

[9] N. Ali, J.-E. Hong, Creating adaptive software architecture dynamically for recurring new requirements, International Conference on Open Source Systems & Technologies, str. 67-72, Lahore 2017.

[10] R. Gao, Research on a Service-oriented Dynamic Adaptive Software Architecture, 16th International Conference on Computer Science & Education, str. 732-735, Lancaster, 2021.

[11] J. Yu, Collaborative Filtering Recommendation with Fluctuations of User' Preference, IEEE International Conference on Information Communication and Software Engineering, str. 222-226, Chengdu 2021.

[12] Z. Wang, A. Maalla, M. Liang, Research on E-Commerce Personalized Recommendation System based on Big Data Technology, IEEE 2nd International Conference on Information Technology, Big Data and Artificial Intelligence, str. 909-913, Chongqing 2021.

[13] K. Pham, What are Recommendation Systems? [online], Medium, 2022, dostupno na: https://medium.com/@khang.pham.exxact/what-are-recommendation-systems-6bb5036042db [pristupljeno 18. lipnja 2024.]

[14] J. Murek, E. Kavlakoglu, What is content-based filtering? [online], IBM, dostupno na https://www.ibm.com/topics/content-based-filtering [pristupljeno 18. lipnja 2024.]

[15] N. Shrivastava, S. Gupta, Analysis on Item-Based and User-Based Collaborative Filtering for Movie Recommendation System, 5th International Conference on Electrical, Electronics, Communication, Computer Technologies and Optimization Techniques, str. 654-656, Mysuru, 2021.

[16] M. M. Reddy, P. Mohandas, A User-based Collaborative Filtering Method to deal with Sparsity in Recommendation Systems by an unsupervised learning of Users' Hidden Preferences, IEEE World Conference on Applied Intelligence and Computing, str. 14-20, Sonbhadra 2022.

[17] S. Puntheeranurak, T. Chaiwitooanukool, An Item-based collaborative filtering method using Item-based hybrid similarity, IEEE 2nd International Conference on Software Engineering and Service Science, str. 469-472, Beijing 2011.

[18] V. Kurama, What Is Collaborative Filtering: A Simple Introduction [online], builtin, 2022, dostupno na https://builtin.com/data-science/collaborative-filtering-recommender-system [pristupljeno 18. lipnja 2024.]

[19] A. Miola, Flutter Complete Reference: Create Beautiful, Fast and Native apps for Any Device, Nezavisno objavljeno, 2020.

[20] Amazon Web Service, What is Flutter? [online], Amazon Web Service, 2024., dostupno na https://aws.amazon.com/what-is/flutter/ [pristupljeno 15. lipnja 2024.]

[21] L. Jarett, Introducing the Flutter Consulting Directory [online], Medium, 2023, dostupno na https://medium.com/flutter/introducing-the-flutter-consulting-directory-f6fc4c1d2ba3 [pristupljeno 15. lipnja 2024.]

[22] J. Sande, M. Galloway, Dart Apprentice (First Edition): Beginning Programming with Dart, Amazon Digital Services LLC, 2021.

[23] Dart dev [online], dostupno na https://dart.dev/language/type-system [pristupljeno 15. lipnja 2024.]

[24] D. Stevenson, What is Firebase? The complete story, abridged. [online], Medium, 2018, dostupno na https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0 [pristupljeno 16. lipnja 2024.]

[25] Firebase – introduction [online], dostupno na https://www.geeksforgeeks.org/firebase-introduction/ [pristupljeno 16. lipnja 2024.]

[26] L. Stanton, How To Run Code In VS Code [online], Alphr, 2023, dostupno na https://www.alphr.com/vs-code-run-code/ [pristupljeno 16. lipnja 2024.]

[27] B.I. Deniz, Hybrid Recommender System [online], kaggle, 2021, dostupno na https://www.kaggle.com/code/iambideniz/hybrid-recommender-system [pristupljeno 18. lipnja 2024.]

[28] N. Koram, R. Garg, Review on Mobile App Development: Tools and Techniques, IEEE World Conference on Applied Intelligence and Computing, str. 260-266, Sonbhadra, 2023.

[29] A. Zainurrohman, Content-Based Recommender System Using NLP [online], Medium, 2021, dostupno na: https://arif-zai-nur-rohman.medium.com/content-based-recommender-system-using-nlp-445ebb777c7a [pristupljeno 9. rujna 2024.]

[30] P. Upadhyaya, Study of Mathematical Model for User-based Collaborative Filtering and Item-based Collaborative Filtering, ResearchGate, 2023.

[31] K. Gruber, J. Huemer, A. Zimmermann, R. Maschotta, Integrated description of functional and non-functional requirements for automotive systems design using SysML, 7th IEEE International Conference on System Engineering and Technology, str. 27-31, Shah Alam, 2017.

[32] R. Damirchi, A. Amini, Non-Functional Requirement Extracting Methods for AI-based Systems: A Survey, 13th International Conference on Computer and Knowledge Engineering, str. 535-539, Mashhad, 2023.

[33] WoofTrax [online], dostupno na: https://wooftrax.com/ [pristupljeno 12. lipnja 2024.]

[34] V.W. Anelli, A. Bellogín, T. Di Noia, D. Jannach, C. Pomo, Top-N Recommendation Algorithms: A Quest for the State-of-the-Art, Proceedings of the 30th ACM Conference on User Modeling, Adaptation and Personalization, Association for Computing Machinery, str. 121-131, New York, 2022.

[35] L. Kugler, How Today's Recommender Systems Use Machine Learning to Cater to Your Every Whim, Communications of the ACM, vol. 67, No. 8, str. 14-16, kolovoz 2024.

SAŽETAK

Ovaj diplomski rad bavi se analizom problema i izazova udomljivanja kućnih ljubimaca, te razvojem višeplatformske mobilne aplikacije koja korisnicima olakšava proces udomljivanja. U radu su analizirane mogućnosti višeplatformskog razvoja mobilnih aplikacija, korištenje prilagodljivih programskih arhitektura te sustavi preporuka s naglaskom na metodu kolaborativnog filtriranja. Na temelju analize postojećih rješenja, definirani su funkcionalni i nefunkcionalni zahtjevi aplikacije. Mobilna aplikacija omogućuje registraciju i prijavu korisnika, stvaranje korisničkih profila, pretragu i kreiranje objava životinja za udomljivanje, slanje i obradu zahtjeva za udomljivanje, razmjenjivanje poruka između korisnika, te je implementiran sustav preporuka temeljen na kolaborativnom filtriranju za generiranje personaliziranih preporuka. Aplikacija je razvijena korištenjem programskog okvira Flutter, programskog jezika *Dart* i biblioteke *Riverpod* za upravljanje stanjima, a s *Visual Studio Codeom* kao razvojnom okolinom. Provedeno je ispitivanje funkcionalnosti aplikacije na android emulatoru, s očekivanjem da će aplikacija pružati sličnu funkcionalnost, performanse i korisničko iskustvo i na drugim platformama. Analiza rezultata pokazuje da aplikacija uspješno olakšava proces udomljivanja ljubimaca, pruža ugodno i intuitivno korisničko iskustvo uz personalizirane preporuke korisnicima, te da je sustav fleksibilan i prilagodljiv, čime se potvrđuje modularnost i skalabilnost arhitekture.

Ključne riječi: mobilna aplikacija, prilagodljiva programska arhitektura, sustav preporuka, udomljivanje ljubimaca, višeplatformski razvoj.

ABSTRACT

Cross-platform mobile application for supporting pet adoption using adaptive programming architecture and collaborative filtering

This thesis deals with the analysis of the problems and challenges of pet adoption, and the development of a multi-platform mobile application that facilitates the adoption process for users. The paper analyzes the possibilities of multi-platform development of mobile applications, the use of adaptive software architectures and recommendation systems with an emphasis on the method of collaborative filtering. Based on the existing analysis of the solution, the functional and non-functional requirements of the application were defined. The mobile application enables users to register and log in, create user profiles, search and create announcements of animals for adoption, send and process adoption requests, exchange messages between users, and a recommendation system based on collaborative filtering is implemented to generate personalized recommendations. The application was developed using the Flutter framework, the Dart programming language and the Riverpod library, with Visual Studio Code as the development environment. Functionality testing was conducted on an android emulator, with the expectation that the application will also offer similar functionality, performance, and user experience on other platforms. The analysis of results shows that the application successfully facilitates the pet adoption process, provides a pleasant and intuitive user experience with personalized recommendations for users, and that the system is flexible and adaptable, which confirms the modularity and scalability of the architecture.

Keywords: adaptive software architecture, cross-platform development, mobile application, pet adoption, recommender system.

ŽIVOTOPIS

Tomislav Odobašić rođen je 4. prosinca 1998. godine u Slavonskom Brodu. Osnovnu školu je upisao i završio u Orašju. Nakon završetka osnovne škole upisuje srednju Strukovnu Školu Orašje u Orašju, smjer elektrotehničar, koju završava 2017. godine. Tokom iste godine upisuje preddiplomski studij elektrotehnike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku.

PRILOZI

Prilog 1. Diplomski rad u datoteci docx

Prilog 2. Diplomski rad u datoteci pdf

Prilog 3. Diplomski rad u datoteci pptx

Prilog 4. Programski kod višeplatformske mobilne aplikacije