# Mobilna aplikacija za kućne ljubimce

**Lukić, Ante**



**SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU**
**ELEKTROTEHNIČKI FAKULTET**

**Sveučilišni studij**

# MOBILNA APLIKACIJA ZA KUĆNE LJUBIMCE

**Diplomski rad**

**Ante Lukić**

**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:** | Ante Lukić |
| **Studij, smjer:** | Sveučilišni diplomski studij Računarstvo |
| **Mat. br. pristupnika, god.** | D1301R, 07.10.2022. |
| **JMBAG:** | 0165082071 |
| **Mentor:** | izv. prof. dr. sc. Josip Balen |
| **Sumentor:** | |
| **Sumentor iz tvrtke:** | |
| **Predsjednik Povjerenstva:** | doc. dr. sc. Krešimir Romić |
| **Član Povjerenstva 1:** | izv. prof. dr. sc. Josip Balen |
| **Član Povjerenstva 2:** | Matej Arlović, univ. mag. ing. comp. |
| **Naslov diplomskog rada:** | Mobilna aplikacija za kućne ljubimce |
| **Znanstvena grana diplomskog rada:** | **Programsko inženjerstvo (zn. polje računarstvo)** |
| **Zadatak diplomskog rada:** | U diplomskom radu potrebno ja razviti mobilnu aplikaciju za vođenje brige o kućnim ljubimcima. Glavne značajke aplikacije su : 1. Pronalazak korisnika koji su prijavljeni da vode brigu o ljubimcima i filitriranje istih. Implementacija bi bila putem mape (Google Maps i sl.) i putem liste. Korisnik može birati koja mu opcija više odgovara. 2. Društveni zid gdje korisnici mogu objavljivati tekst, slike ili video te na taj način promovirati svoju uslugu. 3. Razgovor u stvarnom vremenu kako bi se korisnici mogli upoznati i dogovarati oko brige o svojim ljubimcima. Mobilna |
| **Datum ocjene pismenog dijela diplomskog rada od strane mentora:** | 09.09.2024. |
| **Ocjena pismenog dijela diplomskog rada od strane mentora:** | Izvrstan (5) |
| **Datum obrane diplomskog rada:** | 23.09.2024 |
| **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:** | 30.09.2024. |

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

## IZJAVA O IZVORNOSTI RADA

**Osijek,** 30.09.2024.

| | |
|---|---|
| **Ime i prezime Pristupnika:** | Ante Lukić |
| **Studij**: | Sveučilišni diplomski studij Računarstvo |
| **Mat. br. Pristupnika, godina upisa:** | D1301R, 07.10.2022. |
| **Turnitin podudaranje [%]:** | 6 |

Ovom izjavom izjavljujem da je rad pod nazivom**: Mobilna aplikacija za kućne ljubimce**

izrađen pod vodstvom mentora izv. prof. dr. sc. Josip Balen

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

Tema diplomskog rada je izrada mobilne aplikacije za Android platformu. U današnje vrijeme sve više ljudi posjeduje kućne ljubimce bilo da su oni psi, mačke ili ostale životinje. Također, ljudi počinju udomljavati kućne ljubimce kako bi im pružili moralnu podršku u njihovoj svakodnevici.

Kroz povijest ljudi su vidjeli benefite udomljavanja životinja, a to su najčešće bili psi i mačke. Jedan od najvećih poteškoća s kojim se susreću vlasnici životinja je njihovo privremeno udomljavanje dok vlasnici nisu u mogućnosti da vode brigu o njima. Nažalost, zbog tog problema su životinje patile jer vlasnici nisu bili u mogućnosti pronaći odgovarajuće rješenje te su životinje završile na ulici bez adekvatne osobe da vodi brigu o njima. Diplomski rad predstavlja aplikaciju koja pokušava povezati vlasnike ljubimaca sa korisnicima koji su prijavljeni za brigu o životinjama uz mogućnost naknade. Također, aplikacija povezuje vlasnike ljubimaca s drugim vlasnicima pomoću mogućnosti za slanje poruka u stvarnom vremenu i objavljivanje statusa o svojim najdražima.

Izrada mobilne aplikacije za Android platformu zahtjeva razumijevanje barem Kotlin, XML, YAML i JSON programskih jezika. Također, potrebno je poznavati Android operacijski sustav te osnove korisničkog iskustva. Kako bi se korisnici mogli povezati s drugima i komunicirati potreban je pristup internetu te razvijanje server aplikacije koja će obrađivati podatke i spremati ih u bazu podatka. Aplikacija u diplomskom radu ukomponira sve zahtjeve izrade mobilne aplikacije, pisana je u Kotlin programskom jeziku kao i server aplikacija, a sučelje je izrađeno pomoću Jetpack Compose programskog okvira. Aplikacija poštuje pravila dizajna te koristi biblioteku Material 3 kako bi prikazala najnovije i moderne komponente na korisničkom sučelju.

## 1.1. Zadatak diplomskog rada

Zadatak diplomskog rada je objasniti proces izrade složene mobilne aplikacije za Android platformu koja koristi server aplikaciju sa udaljenom bazom podataka. Također, cilj je objasniti tehnologije koje se koriste poput Android operacijskog sustava, Firebase, Google Maps API, Jetpack Compose itd. Također, potrebno je navedene tehnologije implementirati u praktičnom dijelu diplomskog rada te kreirati mobilnu aplikaciju koja omogućava kreiranje korisničkog računa, objavljivanje statusa na društvenom zidu, objavljivanje oglasa za udomljavanje, komunikacije u stvarnom vremenu, prikaz korisničkih lokacija na karti svijeta itd. Aplikacija je izvedena pomoću Android Studio razvojnog okruženja (engl. *IDE – Integrated development environment*), IntelliJ IDEA razvojnog okruženja i Git sustava za nadzor inačica verzije *(engl. VCS – Version Control System*).

# 2. ANDROID OPERACIJSKI SUSTAV

Android je najpopularniji operacijski sustav koji se natječe sa Apple operacijskim sustavom. Programerska anketa provedena 2017. godine je pokazala da 64.8% programera mobilnih aplikacija proizvodi aplikacije za Android platformu [1].

Android operacijski sustav je najrasprostranjeniji sustav za mobilne uređaje, tablete, televizore, satove, aute itd. Razlog tome je to što je on otvorenog koda što znači da bilo tko može doprinijeti njegovom razvitku ili ga preuzeti te primijeniti u svoju svrhu. Aplikacije za Android platformu su se pisale u Java programskom jeziku sve do 2019. godine kada je Google proglasio Kotlin službenim programskim jezikom. Iza sebe Android sustav kao temelj koristi Linux operacijski sustav koji je također otvorenog koda što govori o tome kako je moguće preuzeti sustav te ga primijeniti za svoje svrhe.

## 2.1. Povijest Android Sustava

Tablica 2.1. Prikazuje koja API (engl. *Application Programming interface*) verzija je distribuirana koliko posto u svijetu. Može se vidjeti kako su zadnjih 5 verzija najrasprostranjenije. Jedan od razloga je automatsko ažuriranje sustava u mobitelima s kojim se promiće korištenje novih verzija.

| Verzija | API verzija | Distribucija |
|---|---|---|
| Android 4.4 (KitKat) | 19 | 0.3% |
| Android 5 (Lollipop) | 21 | 0.1% |
| Android 5.1 (Lollipop) | 22 | 0.8% |
| Android 6 (Marshmallow) | 23 | 1.4% |
| Android 7 (Nougat) | 24 | 1.0% |
| Android 7.1 (Nougat) | 25 | 1.0% |
| Android 8 (Oreo) | 26 | 1.5% |
| Android 8.1 (Oreo) | 27 | 4.3% |
| Android 9 (Pie) | 28 | 8.4% |
| Android 10 (Q) | 29 | 13.6% |
| Android 11 (R) | 30 | 19.0% |
| Android 12 (S) | 31 | 14.7% |
| Android 13 (T) | 33 | 20.9% |
| Android 14 (U) | 34 | 13.0% |

***Tablica 2.1**. Prikaz Android sustava i postotak njihove korištenosti [3].*

Razvoj Android sustava je započeo 2003. u kompaniji Android, Inc. koja je kupljena od kompanije Google 2005 godine. Beta verzija sustava je izašla 5.11.2007. godine, a programski razvojni alat (engl. *Software Development kit SDK*) je izašao 12.11.2006. godine. Prvi mobilni uređaj koji je imao Android sustav bio je HTC Dream, odnosno T-Mobile G1, a izašao je 23, rujna 2008. [2].

## 2.2. Procesi u Android sustavu

Proces je instanca programa koji se izvršava. Proces u Android sustavu može biti u jednom od 5 stanja u bilo kojem vremenu od najprioritetnijeg do onog s najmanjim prioritetom [4].

### 2.2.1. Prednji proces

Prednji proces (engl. *Foreground procesu*) je proces u kojem korisnik vidi korisničko sučelje aplikacije te vrši interakciju s njom. Proces je u ovom stanju ukoliko postoji aplikacijska aktivnost (engl. *Activity*) nad kojom korisnik vrši interakciju. Također, proces je u stanju prednjeg procesa ukoliko postoji Prijamnik emitiranja (engl. *Broadcast receiver*) koji se izvršava ili ukoliko postoji Servis (engl. Service) koji trenutno izvršava programski kod u jednom od njegovih uzvratnih poziva (engl. *callback*). Ukoliko uređaju manjka memorije proces u ovom stanju se može ugasiti kako bi se očuvala memorija ali generalno u ovom stanju uređaj dijeli memoriju na stranice kako bi program ostao dostupan korisniku za korištenje.

U sustavu postoji u bilo kojem trenutku nekolicina prednjih procesa i oni su ubijeni kao zadnja nada u slučaju da je ostalo toliko vrlo malo memorijskog prostora da se ni oni ne mogu nastaviti izvršavati [5].

### 2.2.2. Vidljivi proces

Vidljivi proces je proces koji izvršava posao koji ima pažnju korisnika, te samo ubijanje procesa bi pogoršalo korisničko iskustvo korisniku. Proces je u ovom stanju ukoliko je vidljiva aplikacijska aktivnost ali ne u prednjem planu (*onPause()* metoda je pozvana). To je recimo slučaj kad je prikazan dijalog korisniku. Također, proces je u ovom planu ukoliko postoji servis koji je vidljiv korisniku što govori sustavu da proces izvršava nešto čega je korisnik svjestan ili mu je to prezentirano (ukoliko je glazba puštena te se može vidjeti status pjesme iz obavijesti). Ukoliko proces održava servis kojeg koristi sustav, a korisnik je njega svjestan poput žive pozadine onda je proces također u ovom stanju.

### 2.2.3. Servisni proces

Servisni proces je proces koji sadržava Servis te je pozvana *startService()* metoda. Većinom ovi procesi nisu vidljivi korisniku ali često izvršavaju zadatke koji su u korisnikovom interesu (preuzimanje datoteka u pozadini, sinkroniziranje baze podataka…). Sustav može držati ove procese sve dok ne ponestane memorije za procese u prednjem planu. Također, ovakvi procesi mogu trajati dugo (više od pola sata) pa im tako Android sustav može smanjiti prioritet kako bi oslobodio mjesta drugim procesima.

### 2.2.4. Procesi u privremenoj memoriji

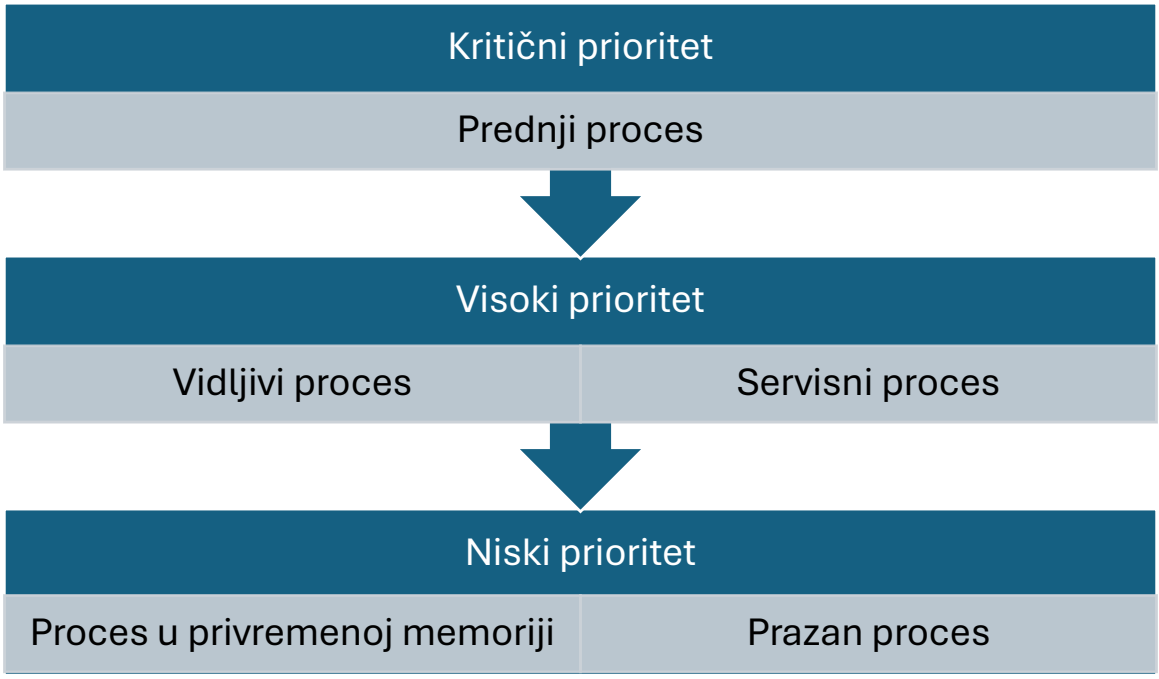
Procesi u privremenoj memoriji nisu trenutno potrebni pa ih sustav može ubiti kako bi mogao osloboditi memoriju. Ovi procesi služe kako bi njihovo ponovno pokretanje bilo brže, te kako bi se moglo vratiti zadnje stanje aplikacije. Obično se oslobađa memorija od najstarijeg procesa u privremenoj memoriji pa sve do najmlađeg. Aplikacije koje imaju implementirana rješenja spremanja trenutnog stanja mogu vratiti svoje zadnje stanje prilikom ponovnog pokretanja čak i ukoliko sustav ubije aplikaciju. Primjer trenutnog stanja je vrijednost upisana u tražilicu ukoliko je to zadnji ekran koji je korisnik vidio.

Dobro vođeni sustav sadrži nekoliko procesa u privremenoj memoriji koji su uvijek dostupni kako bi se efikasno moglo mijenjati aplikacije, te ih učestalo briše iz memorije ako je to potrebno. Samo u vrlo kritičnim situacijama sustav dolazi do točke u kojoj se svi procesi u privremenoj memoriji čiste iz memorije te se kreće čistiti servisne procese [5].

### 2.2.5. Prazan proces

Prazan proces ne sadržava nikakve podatke o aplikaciji te postoje samo kako bi se ubrzalo pokretanje aplikacija ali također mogu biti ubijeni od strane sustava kako bi se dobila memorija za drugi proces. Android sustav najčešće briše prazne procese i procese u pozadini kako bi omogućio memoriju drugom procesu jer obično aplikacije zahtijevaju puno memorije. Procesi mogu sadržavati više niti koje izvršavaju posao poput UI ili glavne *(*engl. *Main*) niti za crtanje po korisničkom sučelju ili I/O (engl. *Input/Output*), odnosno Ulaz/Izlaz niti za komunikaciju van same aplikacije.

Za razliku od operacijskih sustava za računala poput Windows ili MacOS, Android sustav ne treba ručno manipuliranje procesima odnosno nije potreban upravitelj zadataka (engl. *Task Manager*). Aplikacije koje su u pozadini su najčešće u pozadini te ih Android sustav čisti iz memorije ukoliko je dodatna memorija potrebna za trenutni vidljivi proces.




***Slika 2.1.*** *Prikaz prioriteta procesa u Android sustavu.*

## 2.3. Upravljanje memorijom

Android vrijeme izvođenja (engl. *Android Runtime ART*) i Dalvik virtualni stroj koriste dijeljenje i mapiranje memorije kao metode upravljanja memorijom. To znači da svaka memorija koju aplikacija promijeni bilo alociranjem novih objekata ili korištenjem preslikanih stranica ona ostaje u radnoj memoriji (RAM memorija) i ne može se maknuti sa stranice. Jedini način da se oslobodi memorija od aplikacije je da se otpuste sve reference na objekt koji drži aplikacija, te na taj način se dozvoli memoriji da bude dostupna sakupljaču smeća. Postoji jedna iznimka, a to je da se bilo koja datoteka koja se mapirala u memoriju bez promjene, poput programskog koda može straničenjem ukloniti iz radne memorije ukoliko sustav odluči koristiti taj memorijski prostor za drugi proces [6].

### 2.3.1. Sakupljač smeća

ART ili Dalvikov virtualni stroj vodi brigu o svakoj memorijskoj alokaciji. Jednom kada je odlučeno da se dio memorijskog prostora više ne koristi, taj memorijski prostor se oslobađa te se vraća na hrpu bez intervencije razvojnog programera. Taj mehanizam za oslobađanje nekorištene memorije se naziva sakupljač smeća, a on ima dva cilja:

1. Uočiti objekte procesa koji se ne mogu dohvatiti u budućnosti
2. Osloboditi resurse koje su inicijalno zauzeli ti objekti

Memorijska hrpa u Android sustavu se bazira na generacijama. To znači da se dio memorije koji su nedavno alocirani pripadaju mladoj generaciji. Ukoliko objekt ostane dovoljno dugo

aktivan biti će promoviran u staru generaciju, a zatim u trajnu generaciju. Svaka generacija ima svoj limit na količinu memorije koje objekti u njoj mogu zauzeti. Prilikom prikupljanja objekata unutar generacije sustav pokrene sakupljač smeća da se oslobodi memorija. Vrijeme trajanja sakupljanja smeća ovisi o generaciji kojoj objekt za prikupljanje pripada. Iako je proces sakupljanja smeća poprilično brz i dalje može utjecati na performanse aplikacije. Prilikom razvoja aplikacije ne vodi se računa o sakupljanju smeća, te se ne pokušava manipulirati samim procesom. Sustav sadrži vodeći set kriterija o tome kada pokrenuti sakupljanje smeća. Ukoliko se sakupljanje smeća dogodi prilikom prikazivanja zahtjevne aplikacije ili nečeg sličnog ono će povećati potrebno vrijeme izvršavanja što dovodi do odgađanja izvršavanja programskog koda.

### 2.3.2. Dijeljena memorija

Svaka aplikacija želi dio radne memorije, pa Android sustav dijeli stranice radne memorije procesima na način da se svaki proces račva iz već postojećeg procesa. Glavni proces započinje prilikom podizanja sustava te on učitava resurse, programski kod programskog okvira. Većina statičnih podataka se mapira u proces. Ovom tehnikom se dozvoljava podacima da se dijele između procesa, te da se maknu iz stranice kada je to potrebno. Većinom Android sustav dijeli istu radnu memoriju svim procesima koristeći eksplicitno alociranu regiju dijeljene memorije.

### 2.3.3. Dijeljenje memorije na stranice

Radna memorija je podijeljena u stranice. Obično, svaka stranica zauzima 4 KB memorijskog prostora. Stranice mogu biti slobodne ili zauzete. Slobodne stranice predstavljaju nekorištenu radnu memoriju, a korištene stranice su radna memorija koju sustav aktivno koristi, a podijeljena je u kategorije:

- Privremena memorija – Stranica podržana datotekom u memoriji
- Privatna – Stranica koju koristi samo jedan proces i ne može se dijeliti
- Čista - Nepromijenjena kopija datoteke u memoriji, može se obrisati kako bi se povećala slobodna memorija
- Prljava – Promijenjena kopija datoteke u memoriji koja se može premjestiti ili sažeti
- Dijeljena – Stranica koju koristi više procesa
- Anonimna – Stranica koja nema kopiju datoteke u memoriji.

## 2.4. Arhitektura Android sustava

Arhitektura Android sustava se dijeli na slojeve gdje su više razine zaslužne za interakciju s krajnjim korisnikom, a niže za komunikaciju sa sklopovljem u samog uređaja [7].
Razlikujemo 5 razina:

1. Aplikacijska razina (najviša razina)
2. Aplikacijski okvir
3. Biblioteke
4. Android radno okruženje
5. Linux jezgra

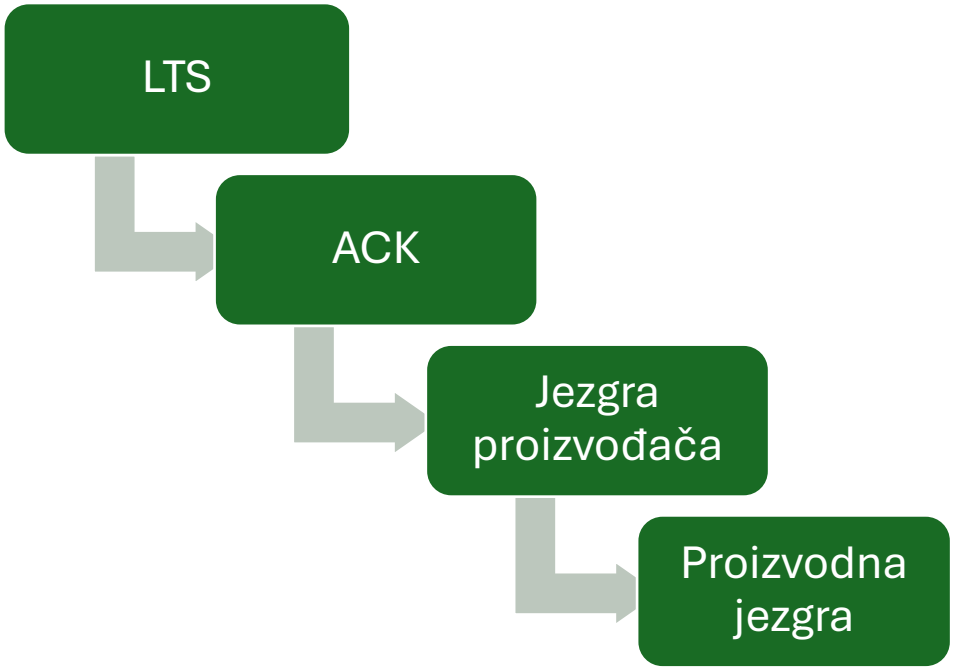

***Slika 2.2.*** *Arhitektura Android sustava.*

### 2.4.1. Linux jezgra

Osnova Android platforme je Linux jezgra. Android vrijeme izvođenja ovisi o Linux jezgri za funkcionalnosti, kao što su upravljanje nitima i niskom razinom memorije [8].

Android jezgra je osnovana na *upstream Linux Long Term Supported (LTS)* jezgri. Ona je spojena sa Android specifičnim dijelovima kako bi se formirale *Android Common* jezgre (ACKs). Novije ACK jezgre (još poznatije i kao GKI jezgre) podržavaju odvajanje od sklopovlje - agnostičkog generičkog osnovnog jezgrinog programskog koda te GKI moduli koji se mogu odvojiti od modula različitih proizvođača koji su ovisni o specifičnom sklopovlju.

Linux jezgra također je zaslužna za upravljanje procesima i memorijom. Također, ono pruža apstraktno sučelje sklopovlja programima što dovodi do toga da prilikom pisanja ili čitanja datoteke aplikacije ne vode brigu o tipu diska, medijskog sadržaja ili tipa datotečnog sustava. Kako je Linux jezgra *open-source* svaki od proizvođača ju nadograđuje ili uz pomoć nje nadograđuje Android sustav za svoje primjene. Kako je jezgra zadužena za upravljanje sklopovljem i mrežom tako omogućuje različitim proizvođačima da poboljšavaju svoj eko-sustav. To se može vidjeti kod Samsung uređaja kako se nesmetano spajaju i komuniciraju s drugim Samsung uređajima, te često postoje i aplikacije napravljene za tu primjenu. Te aplikacije često automatski koriste Bluetooth vezu bez da korisnik ima velikog saznanja o tome. Kako bi uređaji različitih proizvođača komunicirali potrebno je uspostaviti Bluetooth vezu i ručno slati podatke.




***Slika 2.3.*** *Prikaz prinosa do proizvodne jezgre u Android sustavu [9].*

### 2.4.2. Apstraktna razina sklopovlja

Apstraktna razina sklopovlja (engl. *hardvare abstraction layer HAL*) pruža standardno sučelje koje izlaže mogućnosti sklopovlja uređaja višoj razini arhitekture. HAL se sastoji od nekoliko modula biblioteka od kojih svaka implementira sučelje za specifični tip sklopovlja. Kad programski okvir uputi poziv za pristup sklopovlju, Android sustav učitava čitavu biblioteku za tu komponentu sklopovlja. Apstraktna razina sklopovlja skriva upravljačke implementacije niske razine i osigurava kompatibilnost na različitim uređajima. Također, pruža konzistentno sučelje za različita sklopovlja što pojednostavljuje razvoj i održavanje.

### 2.4.3. Android radno okruženje

Za svaki uređaj sa verzijom Androida 5 ili više, svaka aplikacija izvršava svoj proces u svojoj instanci Android radnog okruženja (ART). ART je napisan kako bi izvršavao više virtualnih strojeva na uređajima sa malo memorije uz pomoć Dalvik Executable format (DEX) datoteka, a *bytecode* format dizajniran specifično za Android je optimiziran sa minimalnim utiskom na memoriju. Neki od razvojnih. Alata pretvaraju Java programski kod u *DEX bytecode* koji se može pokrenuti na Android platformi. Android radno okruženje uključuje:

- Prijevremenu kompilaciju kao i kompilaciju na vrijeme
- Optimizirani sakupljač smeća
- Od Android 9 verzije, konverzija DEX datoteka u kompaktniji strojni kod

Prije Android radnog okruženja, Dalvik je predstavljao radno okruženje. Ukoliko se aplikacija može izvršiti u Android radnom okruženju, onda se može i na Dalvik okruženju.

### 2.4.4. Biblioteke

Većina osnovnih komponenti i servisa unutar Android sustava poput ART i HAL su napravljeni uz pomoć *nativnog* programskog koda koji zahtjeva *nativne* biblioteke napisane u C i C++ programskim jezicima. Android platforma pruža Java programski okvir kako bi izložila neke funkcionalnosti od ovih biblioteka svojim aplikacijama. Kako je Android sustav otvoren za razvijanje, biblioteke se mogu razvijati za osobne potrebe. Ukoliko je potrebna visoka kontrola nad sklopovljem ili specijalno ponašanje, uz pomoć C i C++ programskog jezika se kreira biblioteka.

### 2.4.5. Aplikacijski okvir

Čitav niz mogućnosti je izložen aplikacijskom sloju putem Application Programming Interface (API) koji je napisan u Java programskom jeziku. Ovi API formiraju blokove koji su potrebni da bi se izgradila Android aplikacija i olakšala uporaba osnovnog sustava, komponenti i servisa. Aplikacijski okvir ima svrhu pružanja različitih funkcija koje mogu olakšati razvoj same aplikacije. Kako je API napisan u Java programskom jeziku on se prevodi u *bytecode* koji se kasnije izvršava. Ovaj sloj sadrži programske okvire poput View programskog okvira za izradu sučelja, menadžera za obavijesti, menadžer aktivnosti, itd.

### 2.4.6. Aplikacije

Android sustav dolazi sa setom osnovnih aplikacija poput slanja SMS poruka, kalendara, internet pretraživača, kontakata itd. Aplikacije su uključene u platformu nemaju poseban status među aplikacijama koje je korisnik instalirao. To dovodi do toga da internet pretraživač, izrađen od treće strane, može postati glavni pretraživač na uređaju. Osnovne aplikacije u sustavu osim što omogućavaju korištenje korisniku, one također pružaju korištenje svojih funkcionalnosti drugim aplikacijama, što olakšava posao razvojnim programerima. Primjerice ne moraju kreirati svoju kameru već mogu iskoristiti već postojeću aplikaciju kamere u svoje svrhe. Također, Android sustav pruža mogućnost odabira aplikacije za različite svrhe. Često uređaji sadrže više aplikacija za otvaranje datoteka pdf formata i ostalih. Prilikom otvaranja datoteke korisnika se pita s kojom aplikacijom želi otvoriti tu datoteku. Moguće je postaviti i zadanu aplikaciju za otvaranje tih datoteka. Ukoliko postoji zadana, uvijek će se s njom otvarati te datoteke i sustav neće pitati korisnika s kojom aplikacijom da otvori datoteku.

# 3. TEHNOLOGIJE POTREBNE ZA IZRADU ANDROID APLIKACIJE

Kao što je već spomenuto, Kotlin programski jezik je proglašen glavnim programskim jezikom za razvoj aplikacija u Android operacijskom sustavu 2019. godine. Međutim, poznavanje samo jednog jezika za razvoj aplikacija nije sasvim dovoljno jer za razvoj aplikacije se koriste još i:

1. XML (eXtensible Markup Language) – Jezik koji služi za spremanje podataka i opisivanje istih. Prije izlaska Jetpack Compose programskog okvira za opisivanje korisničkog sučelja, XML se koristio kako bi se opisalo korisničko sučelje u View programskom okviru. Također, i dalje se koristi za opisivanje podataka aplikacije u Manifest dokumentu koji se čita prilikom inicijalizacije aplikacije. Dokument sadrži meta-podatke, dopuštenja potrebna za rad aplikacijom, sve aktivnosti unutar aplikacije, početnu aktivnost koja će se pokrenuti, zadanu temu aplikacije itd. Također, kako je već spomenuto XML je jezik za opisivanje i spremanje podataka pa je tako inicijalno pronašao svoju primjenu u opisivanju podataka u komunikaciji sa serverom. Koristi se i danas, ali u manjoj količini zbog JSON jezika koji je lakši za čitati, te je fleksibilniji.
2. JSON (JavaScript Object Notation) – Jezik koji je jednostavan i služi za razmjenu podataka. Također, jednostavan je za *parsiranje* i generiranje programskog koda. U razvoju mobilnih aplikacija to je jezik koji se najčešće koristi za razmjenu podataka sa serverom. Za razliku od XML jezika koji funkcionira uz pomoć oznaka (engl. *Tag*), JSON koristi ključ-vrijednost format i sortirane liste za opisivanje podataka.

## 3.1. Jetpack Compose

Jetpack Compose je programski okvir izrađen od Google-a za izradu korisničkog sučelja. Za razliku od starog View programskog okvira koji se temelji na nasljeđivanju, Jetpack Compose se temelji na kompoziciji, te je samim time komponente puno lakše graditi. Za izradu starih View komponenti bilo je potrebno znati XML za definiranje sučelja i Java/Kotlin programski jezik za opisivanje funkcionalnosti. Za razliku od View programskog okvira koji opisuje kako treba izgledati te manipulira vidljivost određene komponente, u Jetpack Composeu izgled korisničkog sučelja ovisi isključivo o podacima koji dolaze, te se komponente neće ni inicijalizirati ukoliko nema potrebnih podataka.

Koristeći Kotlin za kreiranje korisničkog sučelja, programer uzima odgovornost za crtanje linije koja odvaja poslovnu logiku od korisničkog sučelja. Programer ima slobodu da napravi što ima najviše smisla u danoj situaciji, te ne treba odgovarati i biti ograničen limitima operacijskog sustava. To znači velik broj implicitnih ovisnosti koje postoje između korisničkog sučelja i poslovne logike postaju eksplicitnima [10].

### 3.1.1. Izrada osnovnih komponenti

Jetpack Compose uz pomoć Material biblioteke također razvijene od Google-a pruža osnovne komponente kako bi se olakšalo razvijanje aplikacija. Svaka od komponenti je *composable* funkcija. Composable je anotacija u programskom kodu koja govori sustavu kako upravo ta funkcija opisuje izgled komponente ili ekrana korisničkog sučelja. Neke od osnovnih komponenti su *Button, Text, InputField, Card* i slično. Svaka od tih funkcija sadrži mnoštvo parametara koji služe prilagodbi na određenu temu aplikacije, ali i kako bi olakšali drugim korisnicima tih komponenti lakše implementiranje i prilagođavanje svojim potrebama. Kako bi se komponente mogle složiti po redu koriste se komponente *Column, Row* i *Box.* One slažu komponente u kolonu, red ili jednu ispred druge. Također, postoje *layouti* poput Constraint Layout koji pomažu u raspoređivanju komponenti no oni, za razliku od View programskog okvira nisu toliko poželjni te se koriste samo u slučajevima gdje je potrebna dodatna kontrola rasporeda.

Kao što se može vidjeti iz programskog koda 3.1. postoji mnoštvo parametara u komponenti za unos teksta. Jedan od vrlo bitnih parametara za kojeg je poželjno da je u svakoj komponenti je *Modifier*. To je parametar koji služi za pozicioniranje i određivanje visine, širine, interakcije, i dr. komponente.

TextField.kt

```
@Composable
fun TextField(
    value: String,
    onValueChange: (String) → Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    readOnly: Boolean = false,
    textStyle: TextStyle = LocalTextStyle.current,
    label: @Composable (() → Unit)? = null,
    placeholder: @Composable (() → Unit)? = null,
    leadingIcon: @Composable (() → Unit)? = null,
    trailingIcon: @Composable (() → Unit)? = null,
    prefix: @Composable (() → Unit)? = null,
    suffix: @Composable (() → Unit)? = null,
    supportingText: @Composable (() → Unit)? = null,
    isError: Boolean = false,
    visualTransformation: VisualTransformation = VisualTransformation.None,
    keyboardOptions: KeyboardOptions = KeyboardOptions.Default,
    keyboardActions: KeyboardActions = KeyboardActions.Default,
    singleLine: Boolean = false,
    maxLines: Int = if (singleLine) 1 else Int.MAX_VALUE,
    minLines: Int = 1,
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },
    shape: Shape = TextFieldDefaults.shape,
    colors: TextFieldColors = TextFieldDefaults.colors()
)
```

***Programski kod 3.1**: Prikaz parametara komponente za unos teksta.*

### 3.2.2. *Compose* kompajler

Kako je već spomenuto, komponente su predstavljene funkcijama sa anotacijom *Composable* i iako postoje različiti kompajleri anotacija koji ovisno o anotaciji generiraju programski kod, Jetpack Compose koristi svoj kompajler. On služi za generiranje programskog koda, vodi brigu o pozivanju komponenti, kompoziciji, spajanju, interoperabilnosti sa View programskim okvirom, itd.

Kada Jetpack Compose izvede *composable* funkcije prvi puta, on prati sve komponente koje su pozvane kako bi se opisalo korisničko sučelje. Zatim, prilikom promjene stanja, Jetpack *Compose* rasporedi ponovno izvođenje kompozicije. Tada se ponovno izvedu sve komponente koje su se možda promijenile promjenom stanja [11] .

Ponavljanje kompozicije je temelj Jetpack Compose programskog okvira jer se izvodi relativno brzo, ali može biti zahtjevno za procesor, te je potrebno voditi brigu o tome da samo komponente koje su promijenjene ponovno prođu kompoziciju. Jedan od načina na koji kompajler prepoznaje koje su komponente možda promijenjene je pomoću stabilnih i nestabilnih vrijednosti. Komponente s parametrima koji su nestabilnog tipa uvijek prolaze kompoziciju jer kompajler nije siguran jesu li promijenjene. Nestabilni tipovi su svi ne primitivni tipovi iz klasa koje nisu prošle kroz kompajler (ako su iz drugog modula), te složene klase poput ViewModel klasa sa poslovnom logikom.

### 3.2.3. Compose navigacija

Navigacija je jedan od ključnih elemenata u Android sustavu, jer se s njom prelazi sa jednog ekrana na drugi te se u View programskom okviru koristio FragmentManager ili Jetpack Navigation sa navigacijskim grafom. Iako su komponente funkcije i u teoriji je moguće imati samo jedan ekran, te pozivati funkcije ovisno o nekom parametru, to je izuzetno zahtjevno i vrlo teško za održavati, posebice na velikim aplikacijama sa mnoštvom različitih tipova ekrana. Također, ne smije se zaboraviti kako Android podržava gumb unazad koji može ili prekinuti akciju (npr. spustiti tipkovnicu) ili se vratiti na prethodni ekran.

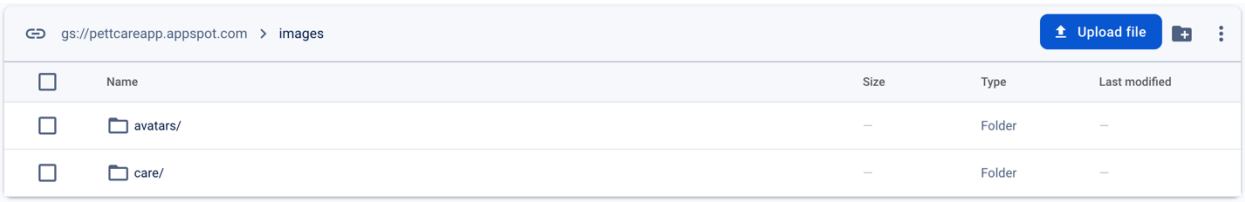
Jetpack Compose navigacija koristi stog kao i prethodne navigacije, te se na stog stavljaju ekrani koji se onda miču po potrebi (npr. pritisak gumba za unazad). Stog je tu vrlo korisna struktura podataka jer on radi na principu „zadnji ulazi – prvi izlazi“ (LIFO) što znači da zadnja vrijednost koja je ušla u stog je prva koja izlazi iz njega. Koristi se *serialized* (može se prevesti na razinu *byte* te slati unutar programskog okvira) objekt ili klasa koja predstavlja rutu. Ruta opisuje kako doći do destinacije i sadrži sve informacije o toj destinaciji [12]. Ruta se koristi kako bi se izvršila navigacija u navigacijskom grafu te je još potrebno imati domaćina navigacije nad kojim

se izvršava navigacija. Koristeći preporučenu arhitekturu u modernom razvoju Android aplikacija to je obično jedna aktivnost koja predstavlja i jedinu aktivnost u aplikaciji. Kao i u Jetpack Navigation koristi se navigacijski kontroler za izvršavanje navigacije jer je on upoznat sa trenutno prikazanim ekranom i stanjem na stogu.

## 3.2. Google Firebase

Google Firebase je aplikacija koja dozvoljava programerima razvijanje iOS, Andorid i Web aplikacija. Firebase pruža alate za praćenje analitika, izvještaje o rušenju aplikacije, kreiranju eksperimenata za svrhe marketinga [13]. Također, sadrži bazu podataka za spremanje, datotečni sustav u oblaku sa spremanje datoteka većih veličina poput slika, videa i slično.

Firebase sadrži svoju konzolu na internetu koja olakšava korištenje svojih usluga. Neke od usluga su besplatne, odnosno gotovo sve usluge su dostupne besplatno za isprobati. Te usluge olakšavaju razvijanje aplikacija jer nije potrebno imati serversko rješenje. Jedna od većih aplikacija koja je koristila Firebase sa nekoliko miliona korisnika je BeReal. Firebase također omogućava objavljivanje aplikacija što može poslužiti kompanijama sa timom QA inženjera, jer na lak način uz malo koraka mogu doći do testne verzije aplikacije. Za potrebe aplikacije u diplomskom radu korišten je datotečni sustav u oblaku za spremanje korisničkih slika. Kao što se može vidjeti sa slike 3.1. pomoću konzole se može pristupiti svim objavljenim slikama unutar aplikacije. Kako je aplikacija vrlo mala, te nije objavljena za javnost proces je vrlo jednostavan i ne zahtjeva nikakve dodatne servise. U praksi je procesuiranje i prikazivanje slika vrlo zahtjevno ukoliko se želi napraviti kvalitetno, te zahtjeva zaseban razvojni tim. U praksi se koristi *content delivery network (CDN)* što predstavlja mrežu rasprostranjenih servera koja dozvoljava korisniku da koristi njemu najbliži server kako bi mu podaci bili servirani na najbrži način. Također, slike se dijele na različite veličine, te ih aplikacije traže ovisno o potrebi tako da slika sa visokom kvalitetom velike rezolucije ne bude servirana kako bi se servirala kao avatar na slici od 32x32 piksela gdje je njezinu kvalitetu nemoguće uočiti, a korisniku će se duže vremena učitavati te će potrošiti više internetskih podataka za njezino slanje.




***Slika 3.1*** *Prikaz Firebase konzole za datotečni sustav u oblaku.*

## 3.3. Gradle razvojni alat

Kako bi se aplikacija koja se razvija bila u mogućnosti distribuirati pomoću Android trgovine aplikacijama ona se mora sažeti u datoteku sa *.apk* ekstenzijom. Kako bi to bilo moguće svi resursi aplikacije se prevode zajedno sa programskim kodom. Kada je aplikacija sažeta u datoteku sa *.apk* ekstenzijom ona se može potpisati, testirati, izdati i distribuirati. Android Studio je razvojni alat koji se koristi za razvoj Android aplikacija, te on koristi Gradle razvojni alat za automatizaciju i upravljanje razvojnim procesima. Gradle dozvoljava izradu različitih konfiguracija što omogućava razvojnim programerima da izrade svoje konfiguracije koje im olakšavaju izradu i testiranje aplikacija poput konfiguracije koja pokreće specifičnu aktivnost kako ne bi morali izvršiti nekoliko navigacija da bi došli do nje. Android programski dodatak za Gradle radi zajedno s razvojnim alatom kako bi omogućio procese i konfiguraciju postavki koji su zasebni za stvaranje i testiranje Android aplikacija [14]. Gradle razvojni alat ima dodatke koji su korisni kod građenja velikih aplikacija sa mnoštvom modula jer on stavlja module u privremenu memoriju pa u slučaju da ništa u modulu ili modulu o kom taj modul ovisi nije promijenjeno Gradle će ga zanemariti i na taj način štedi vrijeme građenja aplikacije. Također, Gradle omogućava građenje aplikacija u paraleli ovisno koliko je procesorskih jezgri dostupno i na taj način također, štedi vrijeme građenja aplikacije.

Gradle zajedno sa Android Studio razvojnim alatom omogućava konfiguraciju različitih verzija na način:

- Tip razvijanja – građenje aplikacije ovisno o fazi razvijanja pa tako se može graditi aplikacija specifično za:
    - Izradu (engl. *development*) - najčešće služi razvojnim programerima
    - Testiranje (engl. *staging*) – najčešće služi QA inženjerima
    - Izdavanje (engl. *release*) – najčešće služi za distribuciju krajnjim korisnicima

  Osim što Gradle razvojni alat služi za razvoj i građenje aplikacije on također omogućava manipulaciju različitih implementacija pa će tako na primjer R8 i ProGuard biti uključeni samo kada je tip razvijanja izdavanje. Time se uštedilo vrijeme građenja aplikacije, a oni će sažeti i osigurati programski kod samo kada je aplikacija namijenjena za izdavanje krajnjim korisnicima.
- Okus – građenje aplikacije ovisno o različitim verzijama. Omogućava definiranje različitih okusa ovisno o samom proizvodu. Recimo da se aplikacija naplaćuje, uz pomoć programskih okusa moguće je definirati građenje aplikacije za besplatnu i plaćenu verziju.
- Varijanta razvijanja – Spaja konfiguraciju o okusima zajedno sa konfiguracijom o tipu razvijanja, pa će tako svaka konfiguracija o okusu imati sve konfiguracije o tipu razvijanja

- Manifest ulazi – Omogućava dinamično dodavanje vrijednosti poput imena aplikacije, verzije i slično ovisno o konfiguraciji aplikacije
- Ovisnosti – Upravlja verzijama ovisnosti, te javlja razvojnom programeru ukoliko koristi stariju verziju
- Potpisivanje – Automatski potpisuje aplikaciju uz pomoć certifikata i ključa
- Podrška više aplikacija – Omogućava građenje više aplikacija i uzimanje samo potrebnog programskog koda ovisno o definiranoj potrebi

Kako bi se Gradle razvojni alat mogao koristiti za razvoj Android aplikacije potrebno je definirati minimalno 3 različita dokumenta:

1. *settings.gradle.kts* – Nalazi se u korijenu projekta i govori Gradle razvojnom alatu koje sve datoteke unutar repozitorija ulaze u izradu aplikacije. U ovom dokumentu je potrebno definirati svaki modul koji aplikacija koristi

settings.gradle.kts

```
pluginManagement {
    repositories {
        google()
        mavenCentral()
        gradlePluginPortal()
    }
}
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
    }
}

rootProject.name = "My Application"
include(":app")
```

**Programski kod 3.2.** *Prikaz settings.gradle.kts dokumenta za aplikaciju*

2. *build.gradle.kts* (Niže razine) – Definira se u svakom modulu aplikacije i sadrži informacije o kompajlerima koji se koriste za taj modul, ovisnostima u modulu, definira je li modul aplikacija ili biblioteka, definira ponašanje modula za različite okuse, tipove razvijanja itd.

3. *build.gradle.kts* (Najviše razine) – Definira verzije i dodatke (engl. *plugin*) zajedno s njihovim verzijama koje aplikacija koristi

settings.gradle.kts

```
buildscript {
    dependencies {
        classpath(libs.google.services)
        classpath(libs.google.services.maps)
    }
}
// Top-level build file where you can add configuration options common to all sub-projects/modules.
@Suppress("DSL_SCOPE_VIOLATION") // TODO: Remove once KTIJ-19369 is fixed
plugins {
    alias(libs.plugins.androidApplication) apply false
    alias(libs.plugins.kotlinAndroid) apply false
        kotlin("jvm") version "1.9.0"
        kotlin("plugin.serialization") version "1.9.22"
    alias(libs.plugins.googleGmsGoogleServices) apply false
}
true // Needed to make the Suppress annotation work for the plugins block
```

**Programski kod 3.3** *Prikaz build.gradle.kts najviše razine dokumenta za aplikaciju*

Kao što se može primijetiti svaki Gradle dokument sadrži ekstenziju *.kts* koja označava da je taj dokument napisan uz pomoć Kotlin *Domain Specific Language (DSL)*. To znači, može se koristiti Kotlin programski jezik, a ne Groovy koji je inače zadani jezik. To smanjuje vrijeme izrade aplikacije jer razvojni programeri ne moraju poznavati drugi programski jezik ukoliko trebaju pisati različite implementacije Gradle zadataka.

Kako je Gradle zaslužan za deklariranje biblioteka koje se koriste u samoj aplikaciji, potrebno je poznavati i njihove verzije. Također, kod aplikacija sa više modula gdje svaki modul ima svoju *build.gradle.kts* datoteku, briga o verzijama može dovesti do problema. Tako je pronađeno rješenje gdje se napravi zaseban modul sa svim verzijama i bibliotekama koje se koriste, kao i aplikacijskim konstantama poput *minSdk* svojstva uz pomoć kojeg se određuje najmanja verzija Android sustava potrebna da bi se pokrenula aplikacija. Da bi to bilo moguće sve vrijednosti moraju imati ključnu riječ *const* jer ona označava da će se vrijednost inicijalizirati u *compile* vremenu te će biti poznata prilikom gradnje aplikacije. Drugi način je postavljanje zadanog kataloga u datoteci *libs.toml* gdje se prilikom sinkroniziranja Gradle datoteka inicijaliziraju sve vrijednosti napisane u toj datoteci, te se kasnije mogu koristiti u drugim Gradle datotekama.

# 4. OSNOVNE ANDROID KOMPONENTE

Pod osnovnim komponentama u razvoju aplikacija za Android operacijski sustav smatramo dodatke od kojih se aplikacija može, ali ne mora sastojati. One mogu biti ulazne točke u aplikaciju ili vršiti operacije same aplikacije. Također, aplikacija ih može izdvojiti tako da ostale aplikacije imaju korištenje tog dodatka u svojim aplikacijama. Primjer toga je aplikacija kamere koju svaka druga aplikacija može koristiti. Svaka od komponenti sadrži svoj životni ciklus kojeg se pridržava i njoj se pridjeljuju metode stvaranja i uništavanja kako bi se mogle izvršiti određene operacije u tim trenutcima.

## 4.1. Aktivnosti

Aktivnost je ulazna točka za interakciju s korisnikom. Predstavlja jedan zaslon s korisničkim sučeljem [15]. Aplikacija se može sastojati od jedne ili više aktivnosti u začetku Android operacijskog sustava je najčešće svaki ekran bila zasebna aktivnost. Dolaskom *Fragment* komponente taj način razvoja aplikacija se promijenio jer su Aktivnosti zahtjevne za memoriju uređaja, te je poželjnije imati manji broj aktivnosti u aplikaciji. U idealnom slučaju, aplikacija se sastoji od samo jedne aktivnosti. Aktivnost predstavlja ulaznu točku u aplikaciju ako se dozvoli pristup tome. Čest primjer toga su ulazak u elektronsku poštu, kameru, kartu, itd. iz druge aplikacije. Aktivnost vodi brigu o:

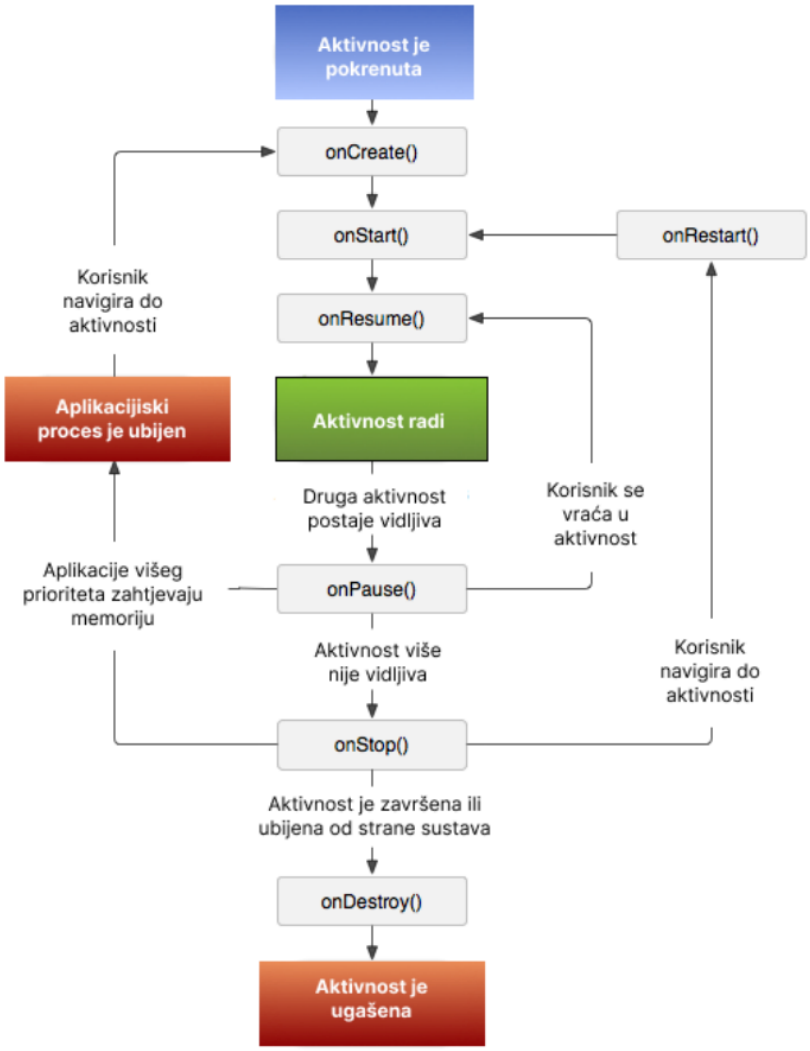
- Korisnikovim interakcijama sa zaslonom
- Procesima u pozadini koji obavljaju poslovnu logiku
- Poznavanju i postavljanju prioriteta prethodnih procesa do kojih bi se korisnik mogao vratiti
- Održavanju stanja u slučaju da je aktivnost uništena zbog postavljanja aplikacije u pozadinu

Sa slike 4.1. se mogu vidjeti metode u životnom ciklusu aplikacije:

- *onCreate()* – Prva metoda prilikom stvaranja aktivnosti. Sadrži parametar tipa *Bundle* u koji se spremaju vrijednosti kako bi se očuvale u slučaju uništavanja aktivnosti. Pozove se jednom u životnom ciklusu aktivnosti, u slučaju da je aplikacijski proces ubijen onda svi životni procesi kreću iz početka.
- *onStart()* – Poziva se nakon *onCreate()* ili nakon vraćanja aktivnosti iz pozadine.
- *onResume()* – Metoda koja slijedi *onStart()* metodi. Ova metoda se može i zasebno pozvati u slučaju da je aplikacija djelomično vidljiva. Primjerice, kada je ekran podijeljen između

dvije aplikacije to dovodi do pauziranja aktivnosti kada nije u fokusu pozivanja *onResume()* metode čim se fokus vrati.

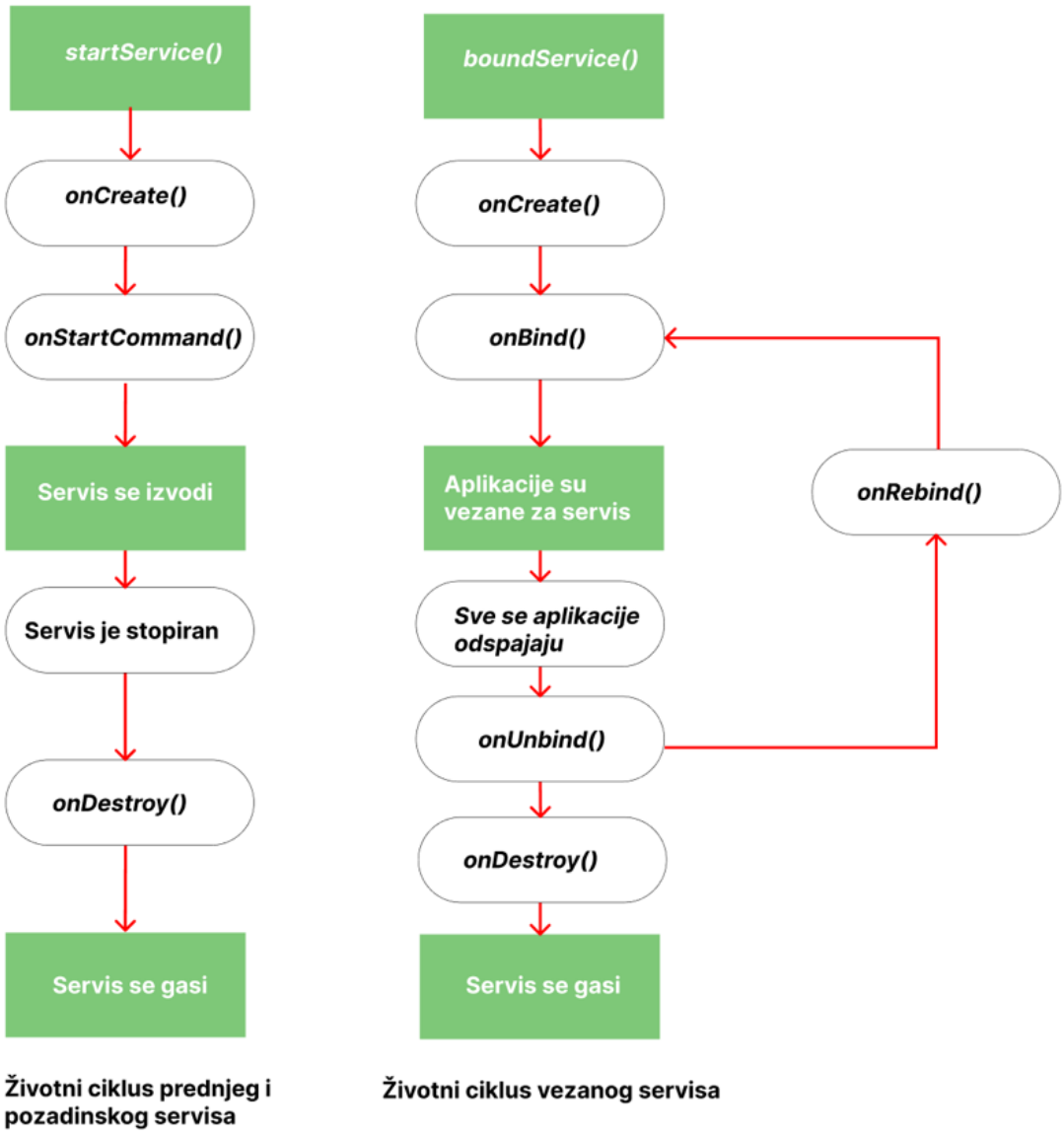
- *onPause()* – Ova metoda se poziva prilikom gubljena fokusa aplikacije, odnosno čim je aplikacija djelomično vidljiva. Također, poziva se i prilikom uništavanja aplikacije zbog navigacije na drugu ili gašenja aplikacije. Prilikom vraćanja fokusa aplikaciji poziva se *onResume()* metoda
- *onStop()* – Ova metoda se poziva prilikom uništavanja aktivnosti ili odlaskom aktivnosti u pozadinu. Nakon vračanja aktivnosti iz pozadine, aplikacija poziva *onStart()* metodu
- *onDestroy()* – Metoda koja označava uništavanje aktivnosti. Do toga dolazi prilikom uništavanja aktivnosti zbog navigacije na drugu ili zbog navigacije na drugu aktivnost.




***Slika 4.1*** *Prikaz životnog ciklusa aktivnosti [16].*

## 4.2. Servis

Servis je aplikacijska komponenta koja može izvoditi operacije dužeg trajanja u pozadini. Ne pruža korisničko sučelje. Jednom pokrenut, servis može trajati neko vrijeme, čak i nakon što se korisnik prebaci na drugu aplikaciju. Dodatno, komponenta može imati interakciju sa drugim servisom i izvoditi komunikaciju između procesa [17].




***Slika 4.2**. Životni ciklus servisa [18].*

### 4.2.1. Usluga u prvom planu

Usluga u prvom planu (engl. *foreground service*) izvodi posao koji može biti vidljiv korisniku. Primjerice, aplikacija za emitiranje glazbe može predstaviti stalnu obavijest korisniku sa detaljima o pjesmi koju izvodi. Prednji servisi moraju biti prikazani sa obavijesti kako bi korisnik imao znanja da rade. One mogu raditi čak i ako korisnik s njima ne vrši interakciju.

### 4.3.2. Pozadinski servis

Pozadinski servis (engl. *background service*) izvodi posao koji nije vidljiv korisniku. Primjerice, sinkroniziranje baze podataka sa serverom. Pozadinski servis obično izvodi posao koji je zahtjevan i traje duže vremena. Za razliku od izvršavanja posla u drugoj programskoj niti, servis sadrži svoj životni ciklus i može trajati i dok aplikacija nije aktivna. Pozadinski servis nazivno kreće iz glavne programske niti koja služi za crtanje po korisničkom sučelju pa je potrebno ili kreirati novu ili se prebaciti na drugu u slučaju korištenja Kotlin Coroutines.

### 4.3.3. Vezani servis

Servis je vezan kada ga aplikacija veže uz pomoć *boundService*() metode. Vezani servis pruža komponentama klijent-server sučelje gdje mogu imati interakciju sa servisom, slati zahtjeve, primati rezultate, vršiti komunikaciju između procesa. Vezani servis ima životni ciklus koji traje jednako koliko i aplikacija koja ga veže. Više aplikacija može biti vezano za proces, ali čim se sve odvežu proces završava.

## 4.3. Prijamnik emitiranja

Prijamnik emitiranja (engl. *broadcast receiver*) je komponenta koja dozvoljava sustavu dostavljanje vanjskih događaja aplikaciji, kako bi ona mogla reagirati na događaje koji utječu na čitav sustav. Kako su prijamnici emitiranja ulaz u aplikaciju, sustav može dostaviti događaj aplikaciji koja se trenutno ne izvodi [15]. Prijamnici emitiranja obično ne pružaju korisničko sučelje, ali mogu prikazati obavijest kako bi obavijestili korisnika kada se događaj dogodi. Aplikacije također mogu emitirati događaje drugim aplikacijama kako bi im javile da se na primjer, preuzela određena količina podataka koje mogu koristiti. Prijamnici emitiranja zapravo predstavljaju ulaz u druge komponente i nisu namijenjeni za izvođenje posla. Aplikacije se mogu registrirati da primaju specifična emitiranja. Kada se emitiranje pošalje sustav automatski usmjerava emitiranja na aplikacije koje su se pretplatile da zaprime taj tip emitiranja [19].

## 4.4. Pružatelj sadržaja

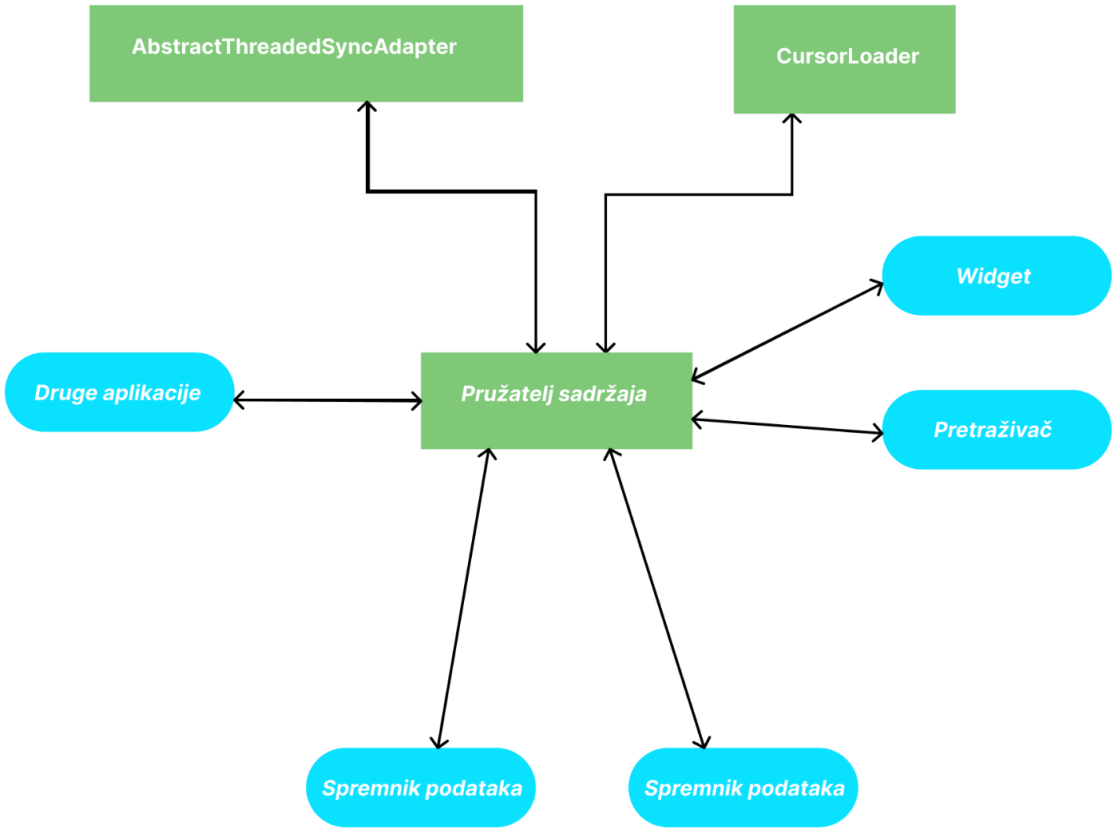
Pružatelj sadržaja upravlja dijeljenim setom aplikacijskih podataka koji se mogu spremiti u datotečni sustav, bazu podataka, internet ili bilo koji drugi oblik spremišta kojem aplikacija može pristupiti. Pomoću pružatelja sadržaja, druge aplikacije mogu dohvaćati i ažurirati podatke, ako im pružatelj sadržaja to dozvoli [15].

Pružatelj sadržaja predstavlja ulaz u aplikaciju za objavljivanje imenovanih podataka koji su identificirani uz pomoć URI scheme. Samim time, aplikacija može odlučiti kako želi da predstavi podatke koje sadrži u URI imenskom prostoru. Aplikacija pruža ta URI identifikacijska imena

drugim entitetima kako bi ih oni mogli koristiti da pristupe podacima. Moguće je dodijeliti URI koji ne zahtjeva da se aplikacija izvodi za njihovo postojanje. Dakle, URI će postojati i u slučaju da se aplikacija ne izvodi, ali prilikom pristupanja podataka aplikacija se mora izvoditi. Također, postoje URI sa visokom sigurnošću koji zahtijevaju dopuštenje da se pristupi podacima. Sustav u tom slučaju može pružiti kratkotrajnu dozvolu da druga aplikacija pristupi podacima skrivenim iza URI identifikacijskog imena.

Kao što se može vidjeti sa slike 4.3 pružatelj sadržaja upravlja dozvolom pristupanja razini podataka u aplikaciji sa nekoliko različitih programskih sučelja. Ta sučelja su:

- Dijeljenje pristupa aplikaciji kroz drugu aplikaciju
- Slanje podataka na malu aplikaciju na početnom zaslonu (engl. *widget*)
- Vraćanje posebnih rezultata pretraživanja unutar aplikacije kroz programski okvir za pretraživanje
- Sinkroniziranje aplikacijskih podataka sa serverom koristeći *AbstractThreadedSyncAdapter*
- Učitavanje podataka na korisničko sučelje koristeći *CursorLoader*




***Slika 4.3*** *Veza između prijemnika emitiranja i drugih komponenti [20].*

# 5. IZRADA APLIKACIJE

Izrada aplikacije je složen proces koji zahtijeva planiranje izvedbe i definiranje potrebnih koraka kako bi se mogla izraditi na najbolji i kvalitetan način. Aplikacija kreće od ideje koju prati istraživanje tržišta, dizajna i potrebnih tehnologija koje će biti uključene u izradu. Istraživanjem tržišta se mogu vidjeti već gotove aplikacije i problemi koji su riješeni s njima. Cilj svakog projekta, odnosno aplikacije je da bude jedinstven, jednostavan i da rješava problem korisniku. Mobilna aplikacija za diplomski rad se naziva PettCare i rješava problem udomljavanja kućnih ljubimaca kada njihovi vlasnici nisu u mogućnosti. Cilj aplikacije je povezati vlasnike ljubimaca kako s osobama koje su voljne voditi brigu o ljubimcima tako i s drugim vlasnicima ljubimaca kako bi se mogli družiti, dijeliti svoja iskustva i naučiti nešto novo. Briga o kućnom ljubimcu može biti izuzetno zahtjevna jer oni mogu zahtijevati veliku posvećenost, ovisno o životinji. Naravno, zbog modernog i dinamičnog života nekada to može postati problem, te se ne može pružiti potrebna posvećenost kućnim ljubimcima u određenom vremenskom razdoblju. Postoje različite aplikacije koje pokušavaju riješiti ovaj problem poput Wag!, Rover i GoWalkies. U svijetu aplikacija od velike je važnosti biti prvi na tržištu, jer je onda konkurencija nepostojeća. Također, nekad aplikacije mogu postojati, ali samo za određeno područje, odnosno mogu biti dostupne, ali ne imati korisnike u određenim državama, regijama i slično.

## 5.1. Planiranje izrade aplikacije

U planiranje izrade aplikacije spadaju različiti faktori koji direktno utječu na njezinu kvalitetu, resurse i vrijeme potrebno za izradu. Kvaliteta, resursi i vrijeme potrebni su za izradu su 3 glavna faktora bilo kojeg projekta i u praksi je uvijek moguće imati dva od tri faktora pokrivena. Ukoliko je bitno vrijeme izrade da bude što manje onda će biti slabija kvaliteta ili će se više resursa potrošiti. Ako je potrebna što veća kvaliteta to će dovesti do dužeg vremena izrade aplikacije ili povećanja troškova zbog povećanja projektnog kadra kako bi se aplikacija mogla završiti na vrijeme. Ukoliko su resursi ograničeni to može dovesti do povećanja vremena izrade aplikacije zbog nemogućnosti proširenja projektnog kadra ili do pada kvalitete kako bi se aplikacija izdala na vrijeme. U ovom diplomskom radu se tražila što veća kvaliteta sa produljenim vremenom izrade aplikacije.

S tehničke strane planiranje aplikacije uključuje: definiranje funkcijskih zahtjeva, dizajn korisničkog sučelja, dizajn sustava aplikacije, odabir prave arhitekture, odabir tehnologije, itd. Svaka faza ovisi o prethodnoj te je potrebno odraditi kvalitetno svaki korak, a pogotovo početne korake definiranja i dizajniranja. Greške u ta dva koraka su najskuplje jer zahtijevaju vraćanje iz faze razvoja u tu fazu, te ponovno implementiranje dijela razvoja.

### 5.1.3. Odabir arhitekture mobilne aplikacije

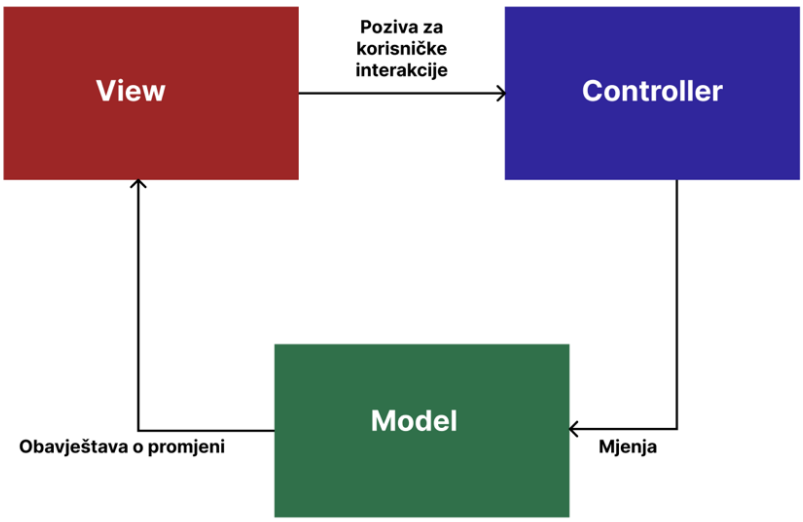
Arhitektura računalnog programa je oblik dan tom programu od strane izvođača. Forma tog oblika je u komponentama, njihovom redoslijedu i komunikaciji. Zadaća tog oblika je da olakša razvoj, izdavanje, operaciju i održavanje programa sadržanog u njemu [21].

Razvijanje Android aplikacija je poznato po tome kako sam Google često preporuča koju arhitekturu aplikacije treba koristiti. Samim time, većina aplikacija počinje koristiti preporučenu arhitekturu. Arhitektura sama po sebi nije jedinstveno rješenje, te ona zapravo predstavlja dobre smjernice za razvoj aplikacije. Kako bi aplikacija bila izvedena korektno potrebna je konzistencija u načinu pisanja i pridržavanje pravila koje sama arhitektura predlaže.

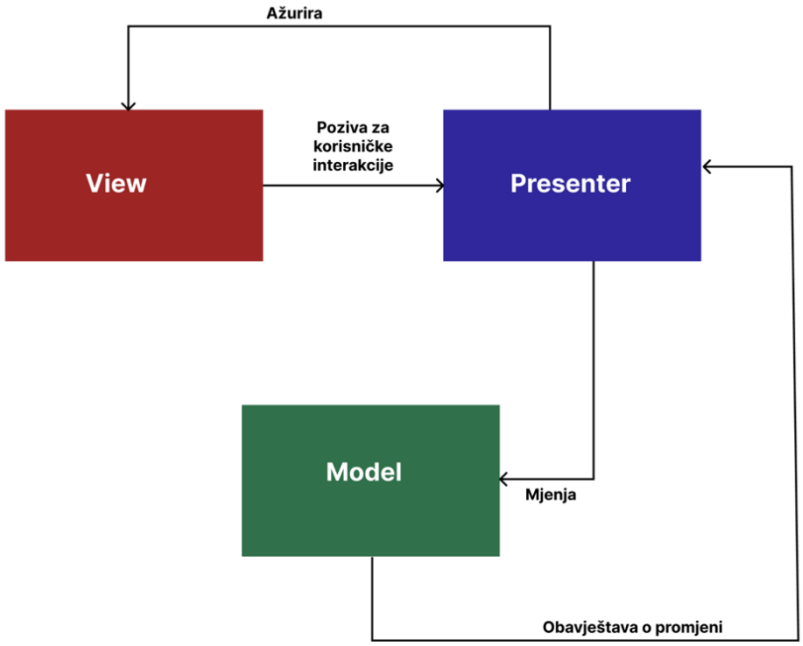
Aplikacije su kroz povijest koristile: Model-View-Controler (MVC*),* Model-View-Presenter (MVP), Model-View-ViewModel (MVVM), Model-View-Intent (MVI), Clean Arhitecture (C) i *MVVM+C.* Potrebno je znati kako svaka od ovih arhitektura ima svoje mane, te je moguće da bilo koja od ovih arhitektura može predstavljati rješenje za određenu aplikaciju. Model-View-Controler *(MVC)* arhitektura je jedna od prvih arhitektura korištenih u Android aplikacijama. View, odnosno korisničko sučelje poziva Controller ukoliko korisnik izvrši interakciju poput pritiska gumba, a također sluša *Model* kako bi saznao za bilo kakvu promjenu svojih podataka. Controller mora upravljati sa nadolazećim interakcijama od korisnika, te on može biti pokretač u komunikaciji aplikacije s vanjskim aplikacijama poput servera kako bi se dohvatili podatci. Također, Controller je zaslužan za promjenu podataka u Model sloju. Sloj Model u MVC arhitekturi sadrži poslovnu logiku i zaslužan je za očuvanje podataka. Kako bi se podatci očuvali, on vodi brigu o validaciji, transformaciji i obavještavanju o promjeni podataka koje sadrži. Ukratko, sloj Model obuhvaća podatke i pruža sučelje za njihovo dohvaćanje i promjenu. Prednost ove arhitekture je *unidirectional data flow*, odnosno u ovoj arhitekturi svi podatci se kreću od vrha prema dolje, a svi događaji od dna (korisničkog sučelja) prema gore (sloj zaslužan za komunikaciju s drugim aplikacijama). Mana ove arhitekture je ta što ne može podržati složenije potrebe aplikacije, odnosno složeni projekti koristeći ovu aplikaciju mogu doći do stanja gdje je teško testirati programski kod i sve je usko povezano. To dovodi do povećanog vremena potrebnog za dodavanje novih stvari u aplikaciju i ispravljanje pogrešnih stvari u njoj.

Model-View-Presenter (MVP) arhitektura predstavlja jednu od jednostavnijih arhitektura jer se sadrži od korisničkog sučelja, podataka i ljepila, odnosno veze između podataka i korisničkog sučelja koju nazivamo Presenter. Presenter je zaslužan za obradu korisničkih interakcija i ažuriranje podataka na temelju njih. Također, on sluša promjene u podatcima, predaje ih korisničkom sučelju kako bi korisniku bili prikazani najrelevantniji i najnoviji podatci. Sloj

korisničkog sučelja i sloj podataka su relativno slični slojevima u MVC arhitekturi. Arhitektura sama po sebi je vrlo jednostavna, te uspostavlja jasna pravila. U praksi kod izrade Android aplikacija ova arhitektura je predstavljala problem zbog dinamične promjene konfiguracije poput promjene orijentacije uređaja koja dovodi do uništavanja sloja korisničkog sučelja te ponovne inicijalizacije. U začetcima arhitekture Presenter je bio predstavljen kao Activity ili Fragment. Kako je promjena orijentacije predstavljala problem, Presenter je postao objekt van životnog ciklusa kako bi mogao izvršavati svoju namjenu bez brige o brisanju podataka u slučaju promjene konfiguracije. Ukoliko se usporede slike 5.1. i 5.2. može se vidjeti kako su arhitekture vrlo slične. To je zato što je MVP arhitektura nastala od MVC arhitekture i primijećuje se kako je jedna od ključnih promjena zapravo separacija, odnosno odvajanje sloja korisničkog sučelja od sloja podataka. To pravilo odvajanja je vrlo dobra praksa jer povećava koheziju između slojeva i lakše je razvijati i održavati svaki sloj, jer sloj podataka u ovoj arhitekturi sada indirektno utječe na sloj korisničkog sučelja. Mana ove arhitekture je ta što sloj Presenter dobiva dodatni posao, a to je ažurirati podatke za korisničko sučelje, što može biti problem kod velikih projekata s mnoštvom poslovne logike jer može dovesti to božanskih klasa koje su ogromne, teško čitljive i teške su za održavanje. Ukoliko se usporedi s Clean arhitekturom, velika je mana nedostatak Domain sloja koji definira svoje modele. Samim time, ukoliko se promijene modeli na Serveru, to uzrokuje velike promjene za modele u MVP arhitekturi. Nedostatak neovisnog sloja često uzrokuje teže održavanje ali u različitim primjenama uzrokuje i brži razvoj kod jednostavnijih aplikacija.

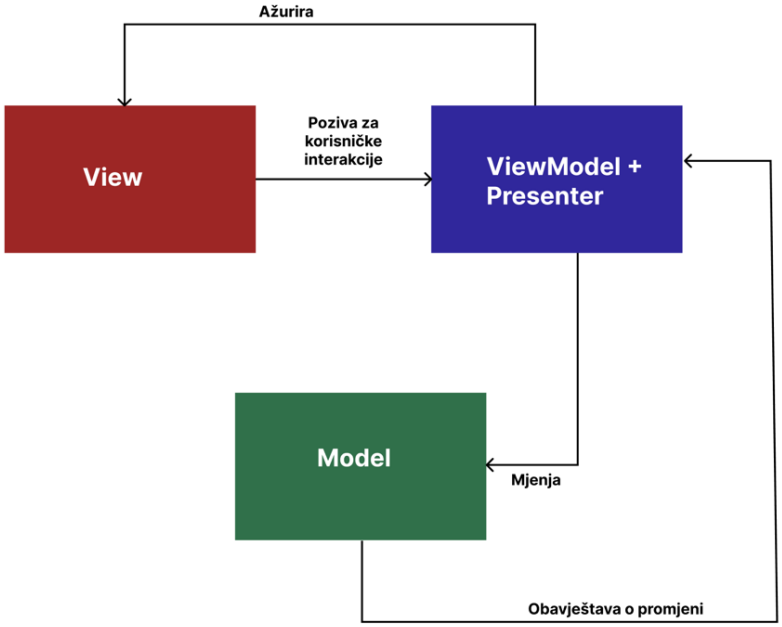



***Slika 5.1.*** *Prikaz MVC arhitekture [22].*



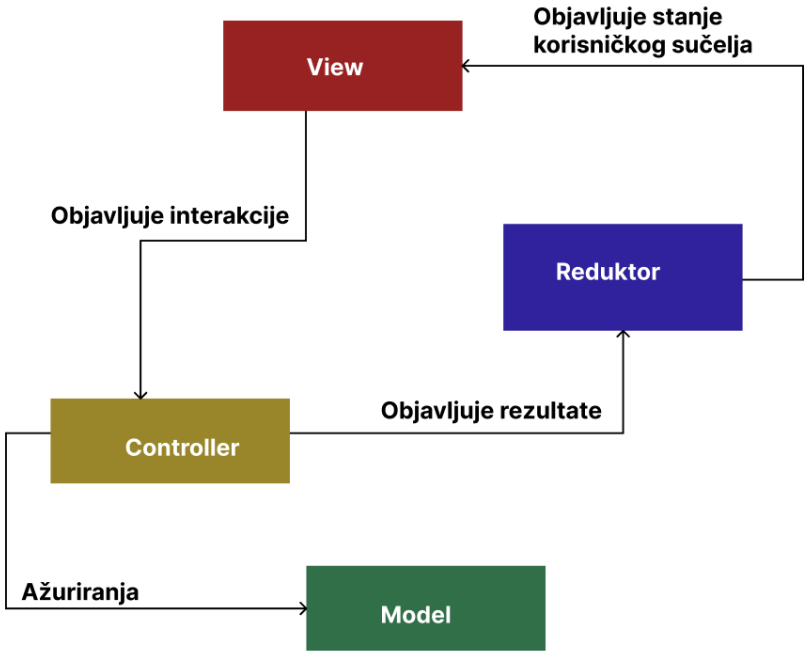

***Slika 5.2**. Prikaz MVP arhitekture [22].*

Model-View-ViewModel (MVVM) arhitektura je nastala iz MVP *arhitekture,* te u praksi ima veliku primjenu u projektima različitih veličina zbog dobre kohezije i jednostavnosti implementiranja. Aplikacije sa *MVVM* arhitekturom većinom sadrže nekakvu vrstu Presenter sloja koji je odgovoran za povezivanje ViewModel sloja s korisničkim sučeljem. *MVVM* implementacije često gledaju da imaju dvosmjerno povezivanje s podatcima, tako da poslovna logika za ažuriranje leži u deklarativno implementiranom korisničkom sučelju. Presenter sloj transformira podatke iz sloja podataka na ViewModel sloj. Za velike operacije poput ispunjavanja forme, sloj Presenter uzima ViewModel i koristi informacije da ažurira podatke. Također, Presenter odrađuje potrebne promjene poput reduciranja nepotrebnih podataka kako bi podatci koji su dostupni ViewModel sloju bili samo oni koji mu zapravo trebaju. Presenter uzima podatke u obliku koji su potrebni sloju Model, te ih mijenja u format koji je prikladan za korisničko sučelje poput transformiranja valute, vremena, itd. Sa slike 5.3. se može vidjeti kako je *MVVM* arhitektura gotovo pa ista kao i MVP arhitektura prikazana na slici 5.2. Vrlo bitna promjena je uvođenje ViewModel sloja koji je također i Android komponenta. ViewModel klase imaju životni ciklus koji odgovara aktivnosti i ne uništavaju se prilikom promjene konfiguracije što ih čini idealnim za čuvanje stanja korisničkog sučelja. Nažalost, kod aplikacija sa zahtjevnom poslovnom logikom ViewModel klase mogu postati božanske kao i u MVP arhitekturi te biti ogromne i teške za održavanje i testiranje.




***Slika 5.3.*** *Prikaz MVVM arhitekture [22].*

Model-View-Intent (MVI) arhitektura je potaknuta Redux arhitekturom koja je primarno smišljena za razvoj web aplikacija. Isto kao i u *MVC* arhitekturi korisničko sučelje objavljuje interakcije Controller sloju, ali u ovoj arhitekturi postoje nova dva faktora, a to su reduktor (engl. *Reducer)* i stanje korisničkog sučelja (engl. *ViewState)*. Stanje korisničkog sučelja je isto kao i ono u *MVVM* arhitekturi, ali se temelji na nepromjenjivosti. Korisničko sučelje je u ovoj arhitekturi funkcija koja ovisi o trenutnom stanju korisničkog sučelja. To znači da svakom promjenom stanja, korisničko sučelje se ažurira kako bi uspješno prikazalo stanje. Kako stanje korisničkog sučelja može biti izuzetno zahtjevno, jer ekran može prikazivati liste različitih podataka. Reduktor je zaslužan za ažuriranje samog stanja. Dakle, korisničko sučelje objavljuje interakciju koju je korisnik izvršio, zatim Controller obrađuje interakciju, odnosno, ažurira podatke te javlja reduktoru koja se interakcija izvršila. Primjerice, u slučaju da korisnik doda jedan artikl liste u svoje favorite sloj korisničkog sučelja će javiti Controller sloju što je korisnik napravio te će Controller javiti sloju podataka da je artikl dodan u favorite, te će sloj podataka izvršiti poslovnu logiku. Controller će također javiti reduktoru što se dogodilo i reduktor će ažurirati artikl kako bi korisniku bilo prikazano da je artikl dodan u favorite. Intent u imenu arhitekture predstavlja akciju koju korisničko sučelje javlja Controlleru. Ova arhitektura je u praksi korištena u aplikacijama s velikim brojem različitih interakcija. Također, princip reduktora i korisničkog sučelja se također koristi kao komponenta i u drugim arhitekturama, jer na dobar način odvaja posao ažuriranja korisničkog stanja  i uzima taj posao drugim klasama koje su u mogućnosti da postanu ogromne.




***Slika 5.4.*** *Prikaz MVI arhitekture [22].*

Clean Architecture (C) je arhitektura koja pokušava spojiti različita rješenja drugih arhitektura u jedno. Sastoji se od različitih slojeva te pokušava biti:

- Neovisna o programskom okviru
- Laka za testiranje,
- Neovisna o korisničkom sučelju
- Neovisna o bazi podataka
- Neovisna o poslovnoj logici i pravilima

Kada se svi ti problemi spoje u jedno dobije se slojevita arhitektura sa slike 5.5. Vanjski slojevi predstavljaju mehanizme, a unutarnji poslovnu logiku. Unutarnji slojevi ne znaju ništa o vanjskom sloju i niti jedno ime iz vanjskog sloja ne bi smjelo biti spomenuto u niti jednom od unutarnjih funkcija. Entiteti obuhvaćaju poslovna pravila i predstavljaju strukture podataka i metode, a slučajevi upotrebe predstavljaju poslovnu logiku specifičnu za aplikaciju. Oni bi zapravo trebali obuhvaćati svu poslovnu logiku. Ova arhitektura je vrlo fleksibilna i ne mora se sadržavati od 4 sloja. Problem ove strukture može biti veliki broj nastalih klasa koje nemaju posebnu funkciju osim mapiranja te su identične poput onih u vanjskom sloju. Također, u slučaju da aplikacija ne sadrži veliku poslovnu logiku klase sa slučajevima upotrebe samo pozivaju funkciju drugog sloja.


***Slika 5.5.*** *Slojevi Clean arhitekture [21].*

MVVM + C je arhitektura koja je zapravo spoj MVVM i Clean arhitekture gdje postoje slojevi poput onih u Clean arhitekturi, ali se također koristi i ViewModel i *unidirectional flow* iz MVVM arhitekture. Ova arhitektura je korištena i u izradi aplikacije za diplomski rad gdje se zapravo svaki dodatak (engl. *feature*) sastoji od četiri sloja, a to su:

- Presentation – Prezentacijski sloj aplikacije koji sadrži korisničko sučelje, prezentacijske entitete, stanje korisničkog sučelja i ViewModel
- Domain – Nezavisni sloj aplikacije koji ne poznaje niti jedan vanjski programski okvir niti bilo koji drugi sloj aplikacije. Zaslužan za obuhvaćanje poslovne logike
- *Data* – Podatkovni sloj koji prati *Repository pattern*. Zaslužan za komunikaciju sa servisnim slojem aplikacije i ubacivanje podataka u reaktivne tokove
- *Service* – Vanjski i servisni sloj koji je zaslužan za komunikaciju sa serverom

## 5.2. Autentifikacija korisnika

Kako bi korisnici mogli imati svoj račun u aplikaciji potrebno im je omogućiti prijavu i kreiranje računa unutar same aplikacije. Iz tog razloga prilikom pokretanja aplikacije korisnici mogu na početnom zaslonu odabrati da li oni žele kreirati novi ili se prijaviti u postojeći korisnički račun. Korisnici se mogu prijaviti u sustav s adresom elektroničke pošte i zaporkom koju odaberu.

S programskog koda 5.1. i 5.2. se mogu vidjeti podatci potrebni za prijavu u sustav i kreiranje korisničkog računa. Dakle, za kreiranje korisničkog računa potrebno je: ime i prezime, zaporka, adresa elektroničke pošte, spol i datum rođenja. Korisnik ako želi može dodati i vlastitu sliku koja je u početku tipa *Uri*, a to je kako je već i spomenuto oblik podatka u Android sustavu koji pokazuje na određenu datoteku, u ovom slučaju sliku. Ta slika se prije komuniciranja sa serverom oko kreiranja računa šalje na Firebase Storage kako bi se tamo spremila u datotečni sustav kako bi ju se moglo dobiti pomoću tekstualnog linka. Na taj način u bazu podataka spremamo primitivan podatak VARCHAR umjesto složenog tipa podatka koji ne garantira da će se slika ponovno prikazati (ukoliko je obrisana s uređaja).

LoginRequestParams.kt

```
data class LoginRequestParams(
    val email: String,
    val password: String,
)
```

SignInUserParams.kt

```
data class SignInUserParams(
    val name: String,
    val surname: String,
    val password: String,
    val email: String,
    val gender: String,
    val dateOfBirth: String,
    val photo: Uri?,
)
```

***Programski kod 5.1. i 5.2.*** *Programski kod za kreiranje i prijavu korisničkog računa.*

S programskog koda 5.3. se može vidjeti slučaj upotrebe (engl. *Use case)* koji prikazuje komuniciranje s drugim slojem. Iako se to može činiti kao kršenje arhitekture, ono to zapravo nije. AuthenticationRepository je sučelje definirano u Domain sloju aplikacije koja je neovisna o drugim slojevima, a Data sloj koji nije neovisan u sebi sadrži implementaciju tog sučelja. Također, može se vidjeti sa slike kako klasa pretvara rezultat dobiven u informaciju koja javlja je li posao dovršen uspješno i dalje se izvršava ili je neuspješno dovršen.

SignInUser.kt

```
class SignInUser(
    private val authenticationRepository: AuthenticationRepository,
) {
    fun result(): Flow<SignInData> = authenticationRepository.signInResults().map(::mapSignInResponse)

    suspend fun request(params: SignInUserParams) = authenticationRepository.requestSignIn(params)

    private fun mapSignInResponse(data: BaseResponse<Boolean>): SignInData = when (data) {
        is BaseResponse.Loading → SignInData(isSuccess = false, errorType = null, isLoading = true)
        is BaseResponse.Error → SignInData(isSuccess = false, errorType = data, isLoading = false)
        is BaseResponse.Success → SignInData(isSuccess = true, errorType = null, isLoading = false)
    }
}
```

***Programski kod 5.3.*** *Programski kod klase slučaja upotrebe za kreiranje korisničkog računa.*

## 5.3. Početni zaslon

Početni zaslon, odnosno prvi zaslon koji korisnik nakon prijave u sustav sadrži dvije stranice. Prva stranica je lista koja prikazuje sve objave za privremeno udomljavanje ljubimaca, a druga stranica je karta napravljena pomoću Google Maps API pomoću koje korisnici mogu vidjeti lokacije oglasa za udomljavanje životinja. Kako je prikazivanje podataka na karti složeno zbog mogućnosti velikog broja podataka koriste se klasteri. Pomoću klastera korisnik može vidjeti samo jedan podatak koji u sebi sadrži broj oglasa koji se skrivaju iza njega te korisnik prilikom približavanja karte otkriva te oglase. Kako bi se korisniku omogućilo listanje između dvije stranice koristi se plutajući gumb koji se nalazi iznad plutajuće donje navigacije.

S programske slike 5.4. se može vidjeti kako se u sloju podataka dohvaćaju podatci te se dodaju u reaktivni tok podataka. Svaki puta kada se želi pristupiti novim podatcima, primjerice zbog listanja kroz kartice, šalje se nova stranica serveru. Na taj način server šalje određeni broj stranica, i održava rezultat malim, kako korisnik ne bi trošio mobilne podatke i aplikacija svoje resurse uzalud. Sloj podataka vraća podatke dekorirane s dodatnim tipom podatka Flow. Kotlin Flow predstavlja reaktivan tip podatka u Kotlin programskom jeziku. Oni mogu biti topli i hladni, odnosno mogu emitirati vrijednosti tek kada imaju slušatelja ili čak i kada nemaju slušatelja prisutnog. Također, razlikujemo Shared Flow i State Flow tipove podataka. Shared Flow se koristi kod operacija koje su trenutne i ne treba čuvati njihovo stanje, poput pritiska gumba ili objavljivanje stranice. State Flow se koristi kada je potrebno čuvati stanje za nove slušatelje. To ga čini savršenim za uporabu u čuvanju stanja korisničkog sučelja unutar ViewModel klase. On u sebi sadrži filtriranje identičnih vrijednosti, pa tako nije moguće objaviti dvije iste vrijednosti zaredom, odnosno u slučaju objavljivanja vrijednosti koja je identična onoj već sadržanoj u *State* Flow tipu podatka, ta vrijednost se neće objaviti. Velika prednost s radom kod reaktivnih tokova je ta što se može raditi aplikacija na temelju interakcija pa tako za svaku korisnikovu interakciju u koliko ispunjava uvijete šalje se nova stranica serveru kako bi se osvježili podatci. Samim time bilo tko može objaviti novu stranicu, a reaktivni tok će automatski poslati poziv serveru. Koristeći *mapLatest* operator može se komunicirati sa serverom jer on predstavlja *lambda* oblik koji će se izvršiti ukoliko dođe do objavljivanja nove vrijednosti u programski tok. Da bi se programski tok mogao izvršiti potrebno je imati barem jednog slušatelja. U arhitekturi aplikacije taj slušatelj je obično ViewModel klasa koja zatim transformira te podatke u stanje korisničkog sučelja. Također, dodatnim funkcijama je moguće odmah u startu objaviti početnu stranicu, pa se tako ne treba emitirati vrijednost u startu iz nekog drugog sloja. Važno je napomenuti kako je reaktivni sloj

vezan za instancu klase u kojoj se nalazi pa tako u slučaju više instanci HomeRepositoryImpl klase svaka od njih će imati svoj zaseban reaktivni tok.

HomeRepositoryImpl.kt

```
internal class HomeRepositoryImpl(
    private val carePostService: CarePostService,
    private val userService: UserService,
) : HomeRepository {

    private val pagePublisher = MutableSharedFlow<Int>(1)

    override suspend fun publishPage(value: Int) {
        pagePublisher.emit(value)
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    override fun result(): Flow<BaseResponse<HomeData>> = pagePublisher
        .mapLatest {
            handleResponse(carePostService.carePosts(it.toString()))
        }
        .loadingOnStart()
        .onStart {
            BaseResponse.Loading
        }

    private suspend fun handleResponse(response: BaseApiResponse<CarePostsResponseApi>?): BaseResponse<HomeData> {
        if (response?.data == null) return BaseResponse.Error.Network
        val userInformations = response.data.items.mapNotNull { getUserInformation(it.creatorId) }
        val postsWithUserInfo = response.data.items.mapNotNull { post ->
            userInformations.find { user -> user.id == post.creatorId }?.let { user ->
                post to user
            }
        }
        return BaseResponse.Success(getHomeData(postsWithUserInfo))
    }

    private suspend fun getUserInformation(userId: String): UserResponseApi? = userService.getUserById(userId)?.data

    private fun getHomeData(postsWithUserInfo: List<Pair<CarePostResponseApi, UserResponseApi>>) = HomeData(
        profiles = postsWithUserInfo.map { (post, user) ->
            CarePost(
                id = post.id,
                name = user.name,
                photoUrl = user.photoUrl,
                description = post.description,
                price = post.price,
                location = LatLng(post.lat, post.lon),
                address = post.address,
                postImageUrl = post.photoUrl,
                creatorId = user.id,
            )
        },
    )
}
```

***Programski kod 5.4.*** *Sloj podataka za početni zaslon u aplikaciji.*

Jedna od velikih prednosti Kotlin programskog jezika, što se tiče čitljivosti samog jezika, su proširive funkcije (engl. *Extension functions)* koje se mogu dodati na bilo koju klasu, čak i na klase koje pripadaju vanjskim bibliotekama. S programskog koda 5.5. može se vidjeti primjer jedne takve funkcije koja se kasnije može koristiti kao dodatak funkcijama koje vraćaju tip Kotlin Flow. Ona omogućava da se emitira stanje učitavanja samo prilikom prvog pozivanja funkcije kako bi se automatski moglo korisniku prikazati učitavanje i na taj način informirati ga da se

podatci dohvaćaju sa servera. Proizvoljan je broj funkcija koje se mogu dodati u lanac na funkcije ovog tipa što uvelike olakšava rad s podatcima i održavanje same akcije. Jedan od čestih slučaja prakse ovakvog pisanja programskog koda je dodavanje funkcije koja će ispisivati sve rezultate koji se objave u reaktivni tok.

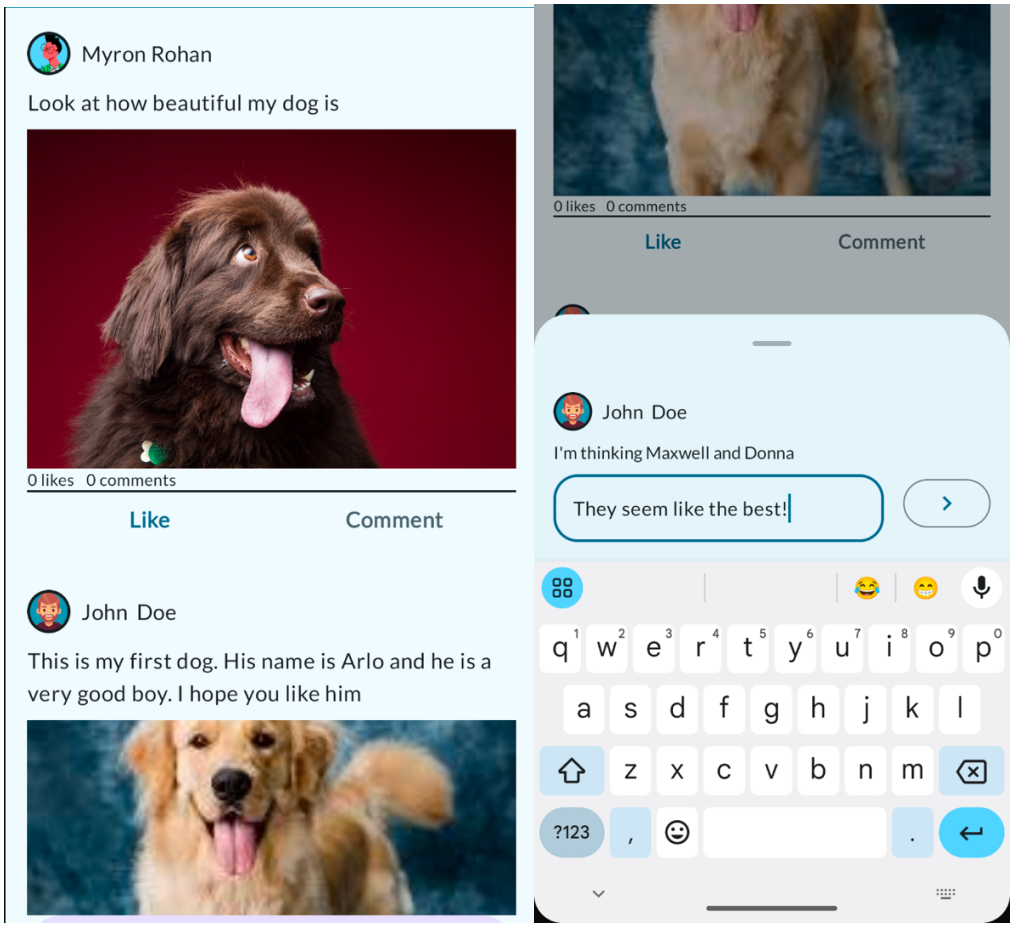

***Programski kod 5.5.*** *Prikaz implementacije funkcije proširenja.*

## 5.4. Društveni zid

Društveni zid je dio aplikacije zaslužen za povezivanje korisnika s drugim vlasnicima ljubimaca. Ideja za tim dodatkom je potaknuta od drugih društvenih mreža gdje korisnici mogu objavljivati sadržaj koji drugi korisnici mogu označiti sa „sviđa mi se“ ili komentirati. Na taj način se pokušava zadržati korisnika u aplikaciji i potaknuti ga da i on sam dijeli sadržaj. Također, korisnici koji objavljuju oglase za privremeno čuvanje ljubimaca mogu koristiti društveni zid za objavljivanje kućnih ljubimaca u njihovoj brizi i na taj način stvoriti povjerenje kod drugih korisnika da ih odaberu za čuvanje svojih ljubimaca.

Kao što se može vidjeti sa slika 5.6. i 5.7. Korisnici mogu objavljivati vizualni i tekstualni sadržaj, a ispod linije se nalaze dva tekstualna gumba koji omogućavaju interakciju korisnika s objavom. Također, ukoliko korisnik pritisne na ime ili sliku vlasnika objave aplikacija ga vodi na njegov profil gdje može vidjeti sve ostale objave. Prilikom pritiska gumba za komentiranje otvara se ekran koji se naziva Bottom Sheet, a on je predstavljen kao dinamičan ekran koji se prikazuje ispred trenutnog, te inicijalno zauzima 50% visine ekrana. On omogućava korisniku da prilikom interakcije s ekranom poveća njegovu visinu na 100% gdje je onda prikazan kao i bilo koji drugi klasični ekran u aplikaciji. Ukoliko mu korisnik snizi visinu na 0% onda ga ujedno i miče sa stoga za navigaciju i on se briše iz memorije. Na taj način korisnik može dinamično listati kroz komentare svake objave. Također, ukoliko korisnik prolista dolje kroz komentare, on dolazi do mogućnosti kreiranja vlastitog komentara koji će biti vidljiv svim korisnicima. Ukoliko korisnik označi objavu sa „sviđa mi se“, aplikacija šalje tu informaciju serveru i u bazi podataka se ažurira broj oznaka „sviđa mi se“ za tu određenu objavu. Ovaj način procesuiranja informacije je pogodan

za male aplikacije s malo korisnika. Društvene aplikacije poput Instragram, Facebook, TikTok, itd. koriste Redis ili nekakav drugi servis koji sprema ove informacije u *cache*, odnosno privremenu memoriju kako bi spriječili skupe pozive prema bazi podataka. Na taj način štede resurse i omogućavaju korisnici neometano kretanje po aplikaciji. Jedna od čestih metoda je također i optimistično ažuriranje gdje se korisniku prikaže da je uspješno označio objavu sa „sviđa mi se“ iako server aplikacija još uvijek nije primila i procesuirala taj zahtjev. To se radi u slučajevima gdje je poziv serveru dugotrajan što dovodi do situacije da prilikom pritiska gumba on ne promjeni svoj oblik ili boju da naznači korisniku uspješno označavanje, pa korisnik svojim ne znanjem pritisne još nekoliko puta misleći da označavanje nije bilo uspješno.




***Slike 5.6. i 5.7.*** *Prikaz društvenog zida i komentara jedne objave*

ViewModel klasa za društveni zid obavlja nekoliko zadataka, a to su:

- Dohvaćanje podataka
- Označavanje objava sa „sviđa mi se“
- Prikazivanje i sakrivanje komentara
- Navigacija do korisnikovog profila
- Objavljivanje i ažuriranje komentara

U konstruktoru klase se mogu vidjeti tri varijable, odnosno tri korisnička slučaja iz Domain sloja arhitekture. Oni u sebi sadrže poslovnu logiku koja je u ovom slučaju provlačenje podataka do zadnjeg sloja koji je zadužen za komunikaciju sa serverom. Neki od korisničkih slučaja prepisuju operacijsku funkciju *invoke* što omogućava pozivanje varijable poput funkcije te smanjenje koda. Svaka akcija koja vodi do komunikacije sa serverom se poziva u *launchInIO* funkciji kako bi aplikacija izvršila potrebnu komunikaciju sa serverom na niti ulaz/izlaz i na taj način oslobodila glavnu nit da izvršava poslove poput crtanja po korisničkom sučelju. Navigacija se izvršava pomoću *publishNavigationAction* funkcije koja predaje programskom bloku varijablu tipa Router koja poznaje svaki ekran te kako doći do njega.

SocialWallViewModel.kt

```
...
    fun likePost(postId: String) {
        launchInIO {
            likeSocialPost.invoke(postId)
        }
    }

    fun showComments(postId: String) {
        postIdForCommentsShown = postId
        updateUiState { state →
            state.copy(comments = state.posts.firstOrNull { it.id == postId }?.commentsToShow)
        }
    }

    fun dismissComments() {
        updateUiState { state →
            state.copy(comments = null)
        }
    }

    fun showProfile(profileId: String) {
        publishNavigationAction {
            it.profile(profileId)
        }
    }

    fun updateComment(value: String) {
        updateUiState { state →
            state.copy(comment = value)
        }
    }

    fun postComment() {
        launchInIO {
            postComment.invoke(postIdForCommentsShown.orEmpty(), uiState.value.comment)
            dismissComments()
        }
    }

    fun nextPage() {
        launchInIO {
            if (shouldUpdatePage.not()) return@launchInIO
            page++
            getSocialWallPost.publishPage(page)
        }
    }
...
```

***Programski kod 5.5.*** *Programski kod ViewModel klase za društveni zid.*

### 5.4.1. Osnovna *ViewModel* klasa

U programskom kodu 5.6. možemo vidjeti osnovnu apstraktnu klasu *BaseViewModel* koja je generička i omogućava svim ostalim *ViewModel* klasama da ju naslijede i koriste njezine metode. To se radi kako bi se obuhvatila bitna logika aplikacije poput korištenja Kotlin Coroutines i navigacije te omogućava pisanje programskog koda samo jednom. Ovoj klasi je potrebno predate osnovno stanje korisničkog sučelja i Router varijablu pomoću koje će se izvršavati navigacija. Ova klasa ima nekoliko odgovornosti, a to su:

- Čuvanje stanja korisničkog sučelja
- Asinkrono ažuriranje stanja korisničkog sučelja pomoću *updateUiState*
- Navigaciju pomoću *publishNavigationAction*
- Navigaciju nazad (identično kao kada korisnik pritisne sustavni gumb "za vraćanje")
- Izvršavanje posla na niti ulaz/izlaz i na glavnoj niti

BaseViewModel.kt

```
abstract class BaseViewModel<T>(
    val router: Router,
    initialViewState: T,
) : ViewModel() {

    private val _uiState = MutableStateFlow(initialViewState)
    val uiState = _uiState.asStateFlow()

    fun updateUiState(state: (T) → T) {
        viewModelScope.launch {
            _uiState.update {
                state(it)
            }
        }
    }

    inline fun publishNavigationAction(block: (Router) → Unit) {
        block(router)
    }

    protected fun goBack() = publishNavigationAction { router →
        router.goBack()
    }

    inline fun launchInIO(crossinline block: suspend (CoroutineScope) → Unit) {
        viewModelScope.launch {
            withContext(Dispatchers.IO) {
                block(this)
            }
        }
    }

    inline fun launchInMain(crossinline block: suspend (CoroutineScope) → Unit) {
        viewModelScope.launch {
            withContext(Dispatchers.Main) {
                block(this)
            }
        }
    }
}
```

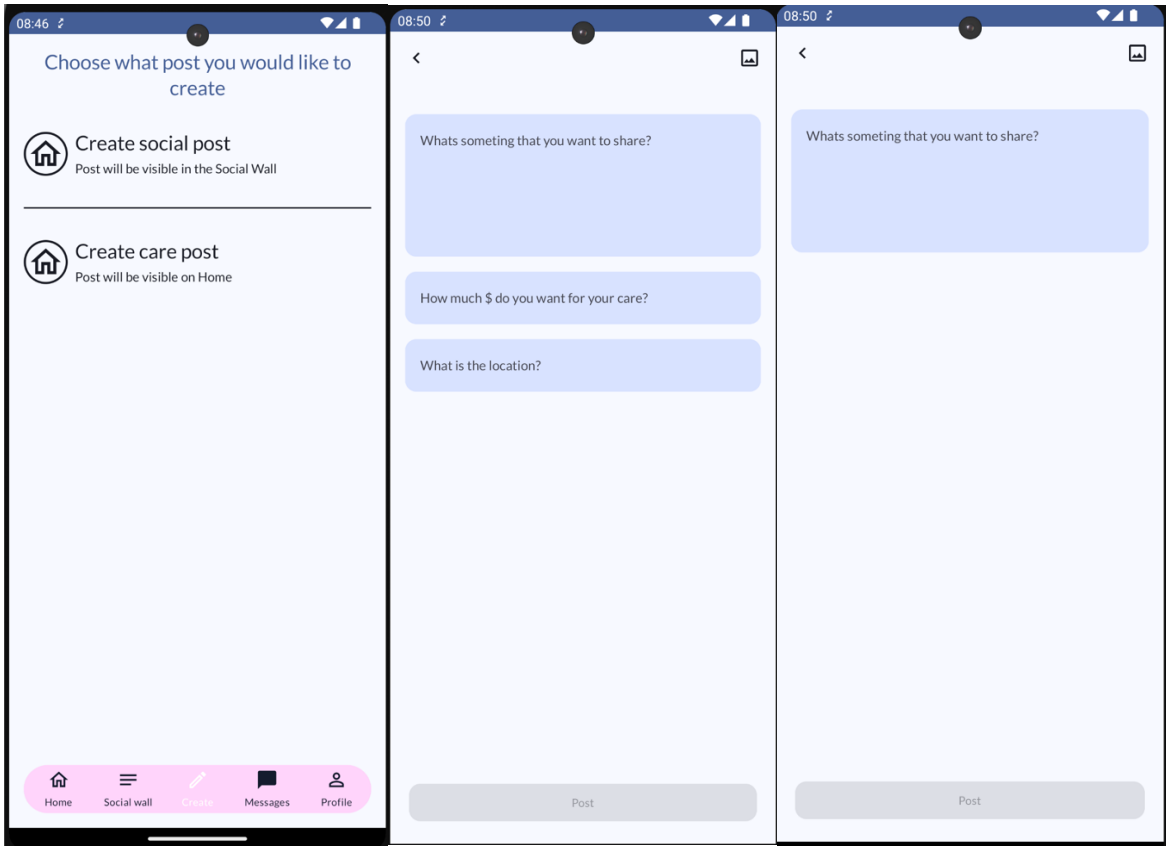
***Programski kod 5.6.*** *Prikaz BaseViewModel apstrakne klase.*

## 5.5. Kreiranje sadržaja

Kako bi postojao sadržaj na početnom zaslonu, na društvenom zidu potrebno je imati mogućnost kreiranja sadržaja. Ovdje dolazi dio za kreiranje sadržaja koji omogućava korisnicima kreiranje oglasa za udomljavanje životinja i objava za društveni zid. Prvi zaslon koji korisnici vide im omogućava odabir kakvu objavu žele, za oglas ili za društveni zid. Zatim, nakon što korisnik odabere jednu od dvije opcije korisniku se prikazuje zaslon koji omogućava ispunjavanje informacija. Kako se radi o dvije različite objave, korisniku se za svaki odabir prikazuje drugačiji zaslon. Kod kreiranja oglasa za privremeno udomljavanje ljubimaca korisniku se prikazuju 3 različita unosa podataka:

1. Prvi unos se odnosi na opis oglasa gdje korisnik može napisati potrebne informacije o sebi ili svojoj usluzi.
2. U drugom unosu korisnik zapisuje cijenu njegove usluge u Američkim dolarima
3. Treći unos se odnosi na lokaciju samog korisnika, odnosno lokaciju s koje će izvršavati svoju uslugu

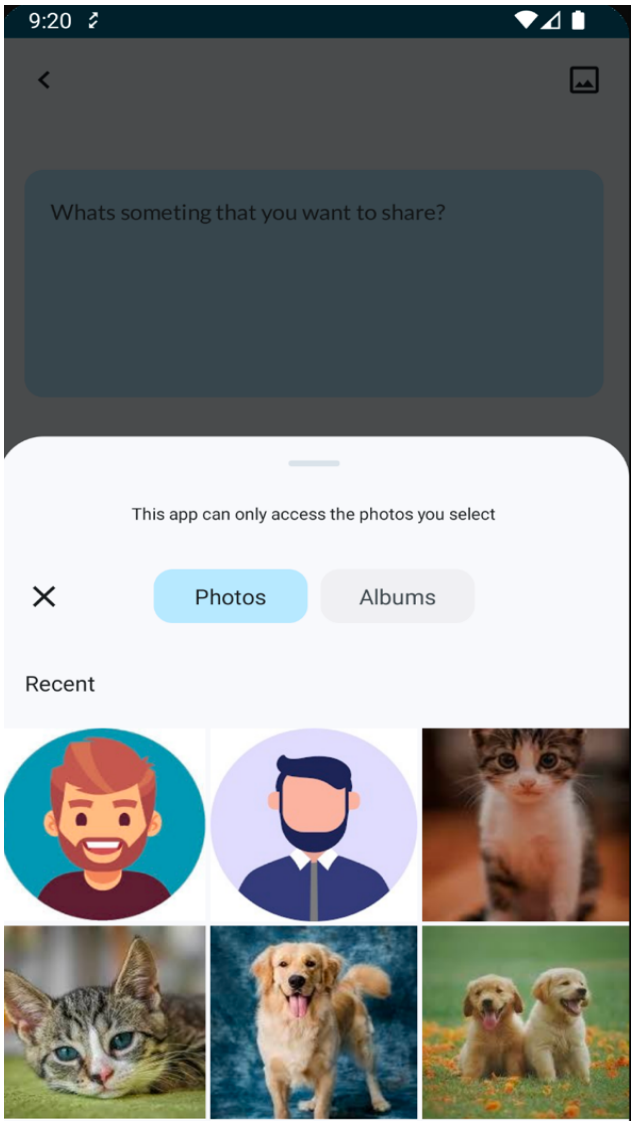
Ukoliko se radi o objavi za društveni zid korisnik unosi samo opis, odnosno tekst koji želi objaviti.




***Slika 5.8**. Dio za kreiranje sadržaja u aplikaciji.*

Kao što je vidljivo sa slike 5.8. za obje objave korisnici mogu dodati i medijski sadržaj, odnosno sliku. Pritiskom na ikonu slike u gornjem desnom kutu korisniku se otvara njegova

galerija u BottomSheet tipu ekrana. Ovaj prikaz galerije je ugrađen u Android sustav i potrebno je samo pozvati ga i ne treba implementirati izgled zaslona.




***Slika 5.9.*** *Prikaz galerije u aplikaciji*

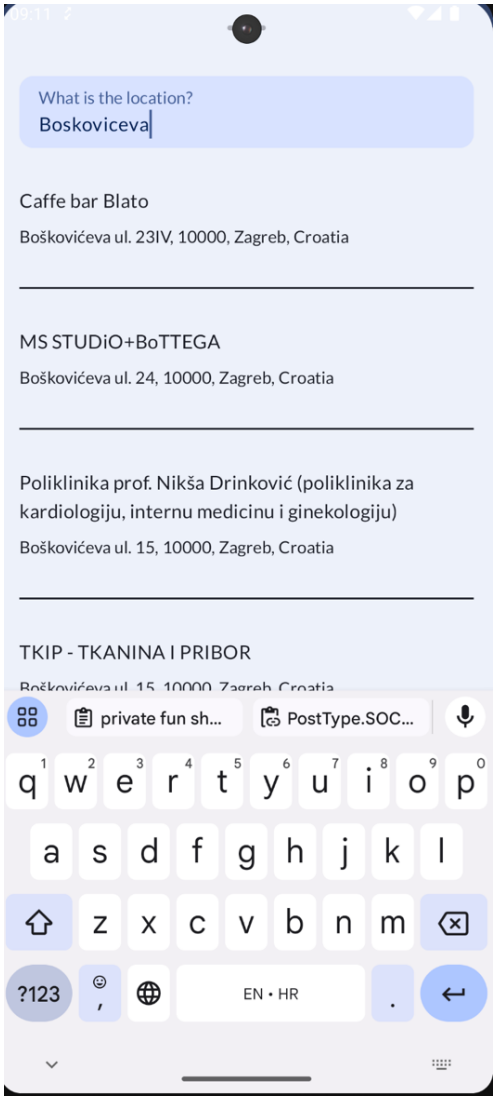
Gumb koji kreira objavu je inicijalno onemogućen sve dok se ne ispune određeni uvjeti za taj tip objave, odnosno dok se ne ispune potrebne informacije. U programskom kodu 5.7. se može vidjeti funkcija zaslužna za omogućavanje gumba za kreiranje objave. Ovisno o tipu objave koji može biti *SOCIAL* i *CARE*, izvršava se provjera. Za objavu koja se prikazuje na društvenom zidu je potrebno dodati tekst, a za objavu o pružanju usluga je potrebno ispuniti tekst, adresu i cijenu. Može se primijetiti kako slika nije dio uvjeta, a to je zato što se ne želi tjerati korisnika da objavljuje medijski sadržaj ukoliko ne želi i na taj način se sprječava prikazivanje nepotrebnih slika u aplikaciji samo kako bi se uvjet ispunio. Također, za provjeru uvjeta se koristi funkcija *isNotBlank*, a ne *isNotEmpty* iz razloga što *isNotBlank* također provjerava je li dodan samo razmak u tekst, odnosno je li tekst sadrži slova ili ne.

CreatePostViewModel.kt

```
private fun shouldEnablePostButton() = updateUiState { state →
        val shouldEnable = if (postType == PostType.SOCIAL) {
            state.text.isNotBlank()
        } else {
            state.text.isNotBlank() && state.address?.isNotBlank() == true && state.price.isNotBlank()
        }
        state.copy(isPostButtonEnabled = shouldEnable)
    }
```

***Programski kod 5.7.*** *Funkcija za omogućavanje gumba za objavu.*

Prilikom pritiska na unos adrese otvara se novi zaslon koji je tipa BottomSheet ali za razliku od galerije, ovaj zaslon se otvara s maksimalnom visinom, odnosno predstavljen je kao običan zaslon, iako ga korisnik i dalje može zatvoriti ukoliko povuče prstom od vrha zaslona prema dolje. Ovaj zaslon je zaseban zato što je za njega potrebna zasebna logika. U pozadini se koristi Google Places API pomoću kojeg se dobivaju informacije o adresi i lokacijama.




***Slika 5.10.*** *Zaslon za odabir lokacije.*

Klasa LocationService vidljiva u programskom kodu 5.8. je zaslužna za komuniciranje s Google servisom koji pruža informacije o lokacijama. Taj servis je implementiran u privatnoj varijabli koja je dostupna u konstruktoru same klase. Varijabla *placesClient* svakim novim unosom slova prima novu vrijednost teksta zajedno s listom vrijednosti koje se žele dohvatiti, a to su koordinate, adresa i ime lokacije. Na kraju se poziva funkcija *await* koja blokira izvršavanje sve dok se ne dohvate potrebni podaci.

LocationService.kt

```
class LocationService(private val placesClient: PlacesClient) {
    suspend fun getLocationsFromQuery(query: String): SearchByTextResponse = placesClient.searchByText(
        SearchByTextRequest.newInstance(
            query,
            listOf(Place.Field.LAT_LNG, Place.Field.ADDRESS, Place.Field.NAME),
        ),
    ).await()
}
```

***Programski kod 5.8.*** *Servis zaslužan za dohvaćanje lokacija*

Zaslon zaslužan za kreiranje oba tipa objava je napravljen na način da je kreiran jedan zaslon koji podržava kreiranja oba tipa objava. To je izvršeno na način da zaslon prima argument, odnosno vrijednost koja govori zaslonu o kojem tipu objave je riječ. S programskog koda 5.9. se može primijetiti kako funkcija ne prima sadržaj tog tipa. To je iz razloga što u implementaciji postoji funkcija istog imena koja prima sadržaj, kreira ViewModel klasu i sve ostale klase potrebne za zaslon. Implementacija je napravljena na taj način zato da bi se mogao vidjeti zaslon prilikom kreiranja sadržaja jer funkcija s implementacijom ovisi o jednostavnim klasama, a ne o kompleksnim klasama poput ViewModel klase. Ovaj pristup je moguć i ne zahtjeva ponovno iscrtavanja čitavog zaslona prilikom promjena vrijednosti zato što su sve vrijednosti u funkciji stabilne. U samoj implementaciji se koristi LazyColumn koji omogućava listanje prema dolje kroz ekran i prikazivanje lista. Njegov ekvivalent u View programskom okviru je RecyclerView. On se koristi zbog mogućnosti proširenog unosa ukoliko se unosi velika količina teksta. Svaka komponenta se nalazi unutar *item* programskog bloka što govori listi da je ovo još jedna komponenta. Komponente koje se prikazuju samo u kreiranju objava za usluge su dodatno dekorirane s AnimatedVisibility što omogućava njihovo animirano prikazivanje u slučaju da trebaju biti vidljive. Animaciju je moguće dodati, ali nije potrebno ukoliko ne postoji poseban zahtjev za tim jer AnimatedVisiblity sadrži animaciju koja je predefinirana. Kod rasporeda samog zaslona se koriste LazyColumn i Box kao njegov roditelj. Razlog takve implementacije je potreba da gumb zaslužan za kreiranje objave bude centriran na dnu ekrana i ne ovisi o broju mogućih

unosa, odnosno o vrsti objave, a to je moguće pomoću Box komponente. Svaka moguća akcija je implementirana pomoću *lambda* funkcija koje na kraju pozivaju ViewModel metodu.

CreatePostScreen.kt

```kotlin
@Composable
private fun ScreenContent(
    imageLauncher: ManagedActivityResultLauncher<PickVisualMediaRequest, Uri?>,
    uiState: CreatePostUiState,
    onTextUpdate: (String) -> Unit,
    toggleAddressPickerVisibility: (Boolean) -> Unit,
    onPriceUpdate: (String) -> Unit,
    onNavigateBack: () -> Unit,
    onPostClicked: () -> Unit,
    modifier: Modifier = Modifier,
) {
    Box(modifier = modifier) {
        LazyColumn(
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.spacedBy(dimensionResource(id = R.dimen.spacing_4)),
        ) {
            item {
                Header(
                    onNavigateBack = onNavigateBack,
                    imageLauncher = imageLauncher,
                    modifier = Modifier.fillMaxWidth(),
                )
            }
            item {
                AnimatedVisibility(visible = uiState.photo != null) {
                    Box(modifier = Modifier.fillMaxWidth()) {
                        PhotoPicker(
                            photoUri = uiState.photo,
                            onPhotoPicked = null,
                            modifier = Modifier
                                .size(dimensionResource(id = R.dimen.registration_photo_size))
                                .background(Color.Gray)
                                .align(Alignment.Center),
                            launcher = imageLauncher,
                        )
                    }
                }
            }
            item {
                PettCareInputField(
                    value = uiState.text,
                    placeHolderText = stringResource(id = R.string.post_text_placeholder),
                    labelText = stringResource(id = R.string.post_text_placeholder),
                    onValueChange = onTextUpdate,
                    maxLines = Int.MAX_VALUE,
                    minLines = TEXT_MIN_LINES,
                    keyboardOptions = KeyboardOptions.Default,
                    modifier = Modifier.padding(horizontal = dimensionResource(id = R.dimen.spacing_4)),
                )
            }
            item {
                AnimatedVisibility(visible = uiState.postType == PostType.CARE) {
                    CarePostAdditionalInputFields(
                        address = uiState.address,
                        price = uiState.price,
                        toggleAddressPickerVisibility = toggleAddressPickerVisibility,
                        onPriceUpdate = onPriceUpdate,
                    )
                }
            }
        }
        PettCareProceedButton(
            text = stringResource(id = R.string.post_button_text),
            isEnabled = uiState.isPostButtonEnabled,
            onClick = onPostClicked,
            textStyle = MaterialTheme.typography.bodySmall,
            modifier = Modifier
                .align(Alignment.BottomCenter)
                .fillMaxWidth()
                .padding(dimensionResource(id = R.dimen.spacing_5)),
        )
    }
}
```

***Programski kod 5.9.*** *Programski kod za kreiranje zaslona za kreiranje objava.*

## 5.6. Profil

Profil je dio aplikacije koji omogućava korisniku da vidi svoje informacije i objave isto kao i od drugih korisnika. Ovaj dodatak omogućava korisnicima da bolje upoznaju druge korisnike s drugim korisnicima putem njihovih informacija i objava, te steknu povjerenje u korisnika koji pruža usluge čuvanja ljubimaca. Informacije korisnika su prikazane pomoću Chip komponente koja ne pruža interakciju već služi za prikazivanje informacija.


***Slika 5.11**. Profil korisnika*

Velika prednost Jetpack Compose programskog okvira se može vidjeti u implementaciji ovog dodatka gdje se zapravo poziva čitav zaslon unutar zaslona za profil. Jetpack Compose promovira ponovno korištenje programskog koda i pisanje Composable funkcija na način da se s lakoćom mogu iskoristiti. Ista implementacija se može postići i u View programskom okviru s FragmentContainerom, ali je i dalje taj zaslon poprilično zaseban. Ovdje se pomoću parametara funkcije može čitav zaslon podesiti ispunjava potrebe. Kako zaslon društvenog zida sadrži listu koja se može listati, informacije o korisniku uvijek ostaju „zalijepljene“ za ekran, a samo se objave listaju. Pisanjem funkcije zaslona su kreirane druge komponente poput ProfileInformation koje se

mogu koristiti u drugim zaslonima. Ukoliko se pogleda implementacija kod zaslona za kreiranje objava, implementacija sadrži LazyColumn na koji se može dodati svojstvo *spacedBy* koji će sve komponente razdvojiti s predanom vrijednosti. U dodatku profil to nije moguće zbog korištenja Column komponente i potrebno je zasebno dodati Spacer komponentu sa željenom visinom, te će ta komponenta predstavljati prazan prostor između komponenti.

ProfileScreen.kt

```
@Composable
private fun ProfileScreen(
    uiState: ProfileUiState,
    onPostComment: () → Unit,
    onLikePost: (String) → Unit,
    onShowComments: (String) → Unit,
    onDismissComments: () → Unit,
    onCommentTextChanged: (String) → Unit,
    loadMore: () → Unit,
    modifier: Modifier = Modifier,
) {
    Column(modifier = modifier) {
        ProfileAvatar(
            name = uiState.name,
            photoUrl = uiState.photoUrl,
            modifier = Modifier.fillMaxWidth(),
        )

        HorizontalDivider(
            thickness = dimensionResource(id = R.dimen.divider_thickness),
            color = MaterialTheme.colorScheme.onBackground,
        )

        Spacer(modifier = Modifier.height(dimensionResource(id = R.dimen.spacing_2)))

        ProfileInformation(email = uiState.email, gender = uiState.gender, dateOfBirth = uiState.dateOfBirth)

        Spacer(modifier = Modifier.height(dimensionResource(id = R.dimen.spacing_2)))

        HorizontalDivider(
            thickness = dimensionResource(id = R.dimen.divider_thickness),
            color = MaterialTheme.colorScheme.onBackground,
        )

        Spacer(modifier = Modifier.height(dimensionResource(id = R.dimen.spacing_2)))

        SocialWallScreen(
            uiState = SocialWallUIState(uiState.posts, uiState.comments, uiState.comment),
            onDismiss = onDismissComments,
            onProfileClick = {},
            onLikeClick = onLikePost,
            onCommentsClick = onShowComments,
            onUpdateComment = onCommentTextChanged,
            onPostComment = onPostComment,
            loadMore = loadMore,
            title = stringResource(id = R.string.social_wall_title_profile),
        )
    }
}
```

***Programski kod 5.10.*** *Implementacija profil zaslona.*

## 5.7. Pomoćni alati u izradi

Kako bi se olakšala izrada aplikacija često se koriste pomoćni alati koji skraćuju vrijeme izrade, osiguravaju kvalitetu same aplikacije, pomažu u testiranju i automatiziraju procese. U diplomskom radu su se koristili alati poput:

- Postman aplikacija za testiranje *REST API* zahtjeva prema serverskoj aplikaciji
- GitHub Actions na GitHub platformi za automatiziranje procesa pomoću skripti
- Material Theme Builder aplikacija za pomoć izrade dizajn vrijednosti same aplikacije
- Statička analiza za održavanje konzistentnosti i kvalitete programskog koda

### 5.7.1. *Postman* aplikacija

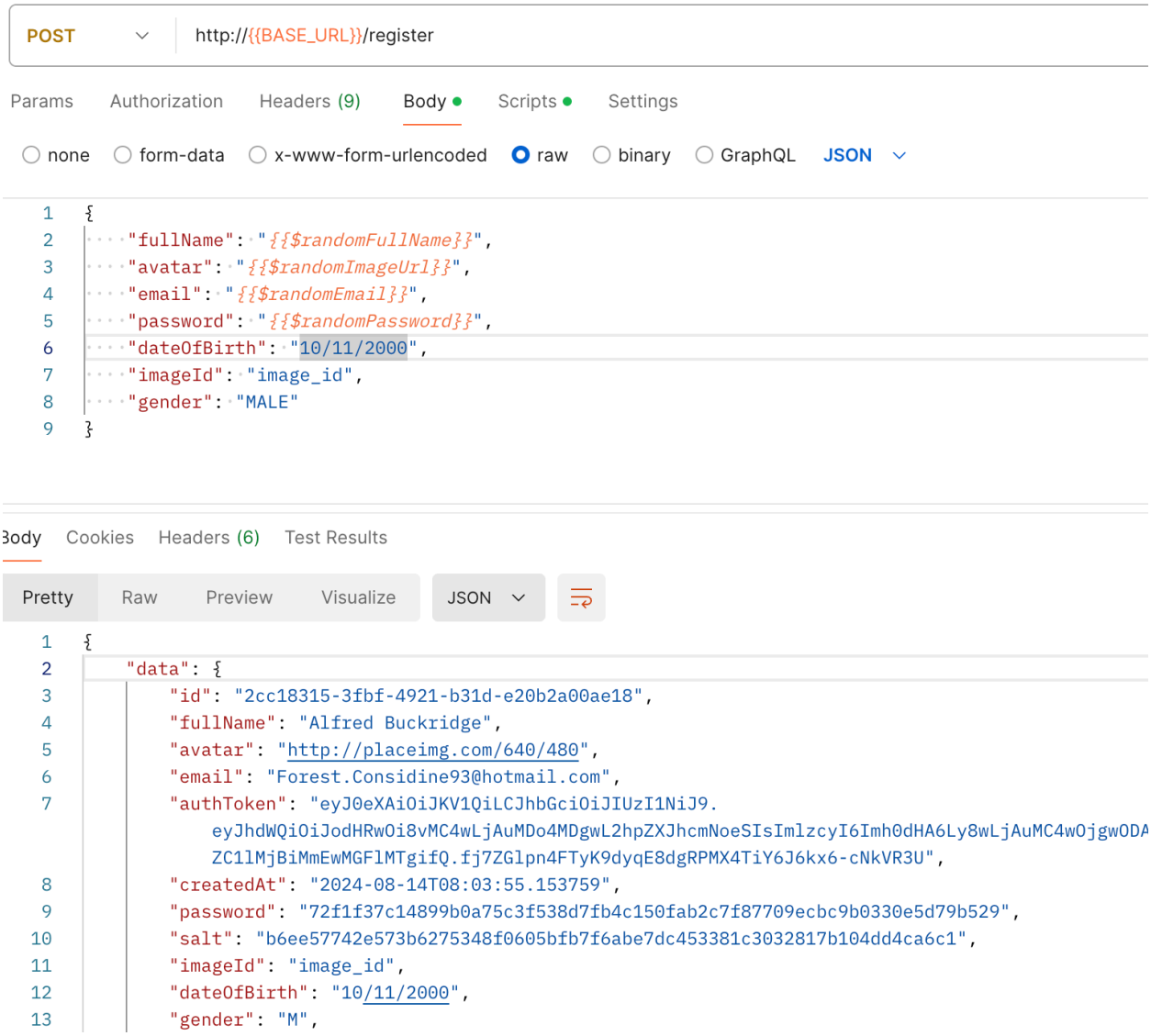
Postman je aplikacija za testiranje serverskih aplikacija različitih protokola poput *REST API* ili *GRAPHQL* protokola. Serverska aplikacija u diplomskom radu koristi *REST API* te je iz tog razloga ona odabrana u Postman aplikaciji. Kao što je prikazano na slici 5.12. aplikacija omogućava odabir metode koja se šalje, ona je u ovom slučaju *POST* što znači da se podaci šalju i primaju. Također, aplikacija omogućava dodavanje parametara, autorizacije, zaglavlja, tijela u potrebnom formatu i skripti koje pomažu prilikom testiranja i neovisne su o zahtjevu. Za tijelo zahtjeva koje se šalje također postoje različite funkcije koje omogućavaju dohvaćanje različitih podataka, bez da se moraju ručno upisati (primjerice *randomFullName).* Također, moguće je definirati više okruženja unutar aplikacije što omogućava mijenjanje različitih okruženja poput razvojnog, testnog i produkcijskog. Ovaj diplomski rad koristi samo jedno okruženje koje u sebi ima definiranu varijablu *BASE_URL*, odnosno ona sadrži *url* vrijednost servera koja je uvijek ista te se na nju dodaju rute prema kojima serverska aplikacija razumije kako obraditi zahtjev. Na slici je prikazan zahtjev za registraciju korisnika koja vraća vrijednost *authToken*. To je vrijednost *JWT* tokena, odnosno jednostavan tip autorizacije samog korisnika. Token ima određenu vremensku valjanost nakon koje više ne vrijedi, pa se mora kreirati novi. Taj token se mora slati u svakom zahtjevu kojem je to potrebno u obliku *„Authorization: Bearer“* i zatim vrijednost tokena u zaglavlju.

Kao što je vidljivo u programskom kodu 5.11. skripta omogućava dohvaćanje tokena iz odgovora serverske aplikacije i spremanje u trenutno okruženje. To olakšava dodavanje tokena u zaglavlje i njegovu dinamičnu promjenu jer više nije potrebno kopirati i zalijepiti isti na svaki zahtjev koji se šalje serveru unutar same aplikacije.

scripts

```
const responseJson = pm.response.json();
var bodyToken = responseJson.data.authToken;
pm.environment.set("JWT_TOKEN", bodyToken)
```

***Programski kod 5.11.*** *Prikaz skripte za spremanje JWT tokena.*



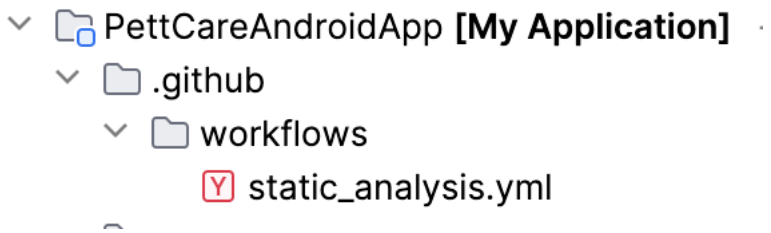

***Slika 5.12.*** *Prikaz zahtjeva u Postman aplikaciji*

### 5.7.2. Github Actions

GitHub Actions je platforma za kontinuiranu integraciju i dostavu koja omogućava automatizaciju građenja aplikacije, testiranje i isporučivanje. Moguće je kreirati tijek rada koji grade i testiraju za svaki *pull request* u repozitoriju ili isporučuju spojeni dio koda u produkciju.

Pružaju Linux, Windows i macOS virtualne strojeve za izvođenje tijeka rada. Također, moguće je imati i svoje izvođače [23].

U diplomskom radu se koristi za provedbu statička analize koja provjerava promijenjeni dio koda. Kako bi to bilo moguće, mora postojati određena skripta. Kako bi GitHub Actions prepoznao koju akciju treba poduzeti potrebno je dodati datoteke u samom repozitoriju. Slika 5.13. prikazuje korišteni datotečni sustav za izvođenje.




***Slika 5.13.*** *Prikaz datotečnog sustava za GitHub Actions.*

Datoteka *static_analysis.yml* je datoteka napisana u YAML jeziku te su u njoj zapisani detalji izvođenja.

static_analysis.yml

```yaml
name: Static Analysis
on: pull_request
jobs:
  staticAnalysis:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: ./gradlew staticAnalysis
```

***Programski kod 5.12.*** *static_analysis.yml datoteka*

Za potrebe izvođenja potrebno je dodati ime, kada se skripta izvodi i poslovi, te opisati poslove. Dakle, za statičku analizu ime je „Static Analysis“ i izvodi se na svakom *pull request* događaju, a to ubraja kreiranje i dodavanje novog *git commita*. Za posao je potrebno označiti koji izvođač izvodi posao, pa je u diplomskom radu odabran Linux Ubuntu izvođač koji treba koristiti treću verziju akcije i pokrenuti skriptu „./gradlew staticAnalysis“ koja će biti objašnjena kasnije.

### 5.7.3. Alat za odabir teme

Material Theme Builder je alat izrađen od tima zaslužnog za Material dizajn sustav i biblioteku. Material je dio cjeline za izradu Android aplikacije jer je glavna biblioteka za izradu korisničkog sučelja napravljena od strane *Googlea*. Sam alat omogućava promjenu boja i fontova u stvarnom vremenu uz paralelan prikaz na demo aplikaciji kako bi korisnik bio u mogućnosti

vidjeti kako određene vrijednosti utječu na aplikaciju ukoliko su vrijednosti točno postavljene. Sam alat isporučuje programski kod napisan u Kotlin programskom jeziku za Jetpack Compose programski okvir. Alat podržava različite teme poput teme za svijetli i tamni način rada, visok i nizak kontrast, itd. Za potrebe diplomskog rada je odabran samo svijetli i tamni način rada. Razvojni programer s druge strane mora poznavati koje se boje koriste u kojim elementima aplikacije. Također, bitno je napomenuti da u praksi za ovaj dio izrade same aplikacije je zaslužan dizajn tim koji omogućava razvojnom programeru da vidi koje vrijednosti iz teme su korištene.

Programski kod 5.13. prikazuje generirane vrijednosti u svijetloj temi. Za korištenje generiranih vrijednosti potrebno je koristiti klasu MaterialTheme koja je prepisana u samom generiranju vrijednosti. Kako je generirana prepisana vrijednost dodana samoj temi aplikacije razvojni programer može odmah pristupiti tim vrijednostima. Cijela aplikacija se nalazi u temi jer se aplikacija pokreće iz glavne aktivnosti, a ona postavlja temu te se navigacija dodaje u temu, a samo se jedna navigacija koristi u cijeloj aplikaciji. Tako jednostavna i dobra arhitektura prati načela spomenuta ranije i prikazuje kako odvajanje briga omogućava jednostavno kreiranje aplikacije, jer razvojni programer uz pomoć sučelja može pristupiti svim vrijednostima bez da zna koja je stvarna vrijednost određenog atributa.

Theme.kt

```
private val lightScheme = lightColorScheme(
    primary = primaryLight,
    onPrimary = onPrimaryLight,
    primaryContainer = primaryContainerLight,
    onPrimaryContainer = onPrimaryContainerLight,
    secondary = secondaryLight,
    onSecondary = onSecondaryLight,
    secondaryContainer = secondaryContainerLight,
    onSecondaryContainer = onSecondaryContainerLight,
    tertiary = tertiaryLight,
    onTertiary = onTertiaryLight,
    tertiaryContainer = tertiaryContainerLight,
    onTertiaryContainer = onTertiaryContainerLight,
    error = errorLight,
    onError = onErrorLight,
    errorContainer = errorContainerLight,
    onErrorContainer = onErrorContainerLight,
    background = backgroundLight,
    onBackground = onBackgroundLight,
    surface = surfaceLight,
    onSurface = onSurfaceLight,
    surfaceVariant = surfaceVariantLight,
    onSurfaceVariant = onSurfaceVariantLight,
    outline = outlineLight,
    outlineVariant = outlineVariantLight,
    scrim = scrimLight,
    inverseSurface = inverseSurfaceLight,
    inverseOnSurface = inverseOnSurfaceLight,
    inversePrimary = inversePrimaryLight,
    surfaceDim = surfaceDimLight,
    surfaceBright = surfaceBrightLight,
    surfaceContainerLowest = surfaceContainerLowestLight,
    surfaceContainerLow = surfaceContainerLowLight,
    surfaceContainer = surfaceContainerLight,
    surfaceContainerHigh = surfaceContainerHighLight,
    surfaceContainerHighest = surfaceContainerHighestLight,
)
```

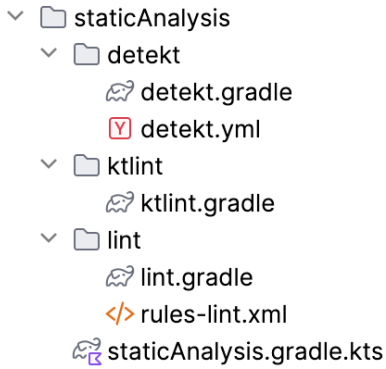
***Programski kod 5.13.*** *Generirane vrijednosti u svijetloj temi*

### 5.7.4. Statička analiza

Statička analiza je alat, odnosno proces koji se nalazi u procesu verifikacije same aplikacije, te provjerava kvalitetu programskog koda i uočava potencijalne probleme aplikacije. U diplomskom radu se sadrži od:

1. Android Lint
2. Ktlint
3. Detekt

Svaki od njih sadrži svoja pravila koja je moguće potisnuti ukoliko su van konteksta aplikacije. Neka od pravila su: biblioteke koriste stariju verziju, nepotrebna prazna linija, prevelik broj parametara u funkciji, sučelju ili klasi, prevelika funkcija, itd. Kako bi se statička analiza mogla postaviti potrebno je dodati svaku od tih biblioteka u aplikaciju te dodati datoteku s kojom se pravila mogu potisnuti. Za Detekt se također koriste dodatna pravila od X (prijašnje *Twitter)* tima koja propisuju pravila za pravilno pisanje programskog koda u Jetpack Compose programskom okviru.




***Slika 5.14.*** *Prikaz datoteka potrebnih za statičnu analizu.*

Svaka od biblioteka sadrži svoju „build.gradle“ datoteku kako bi se mogle dodati u *Gradle task* koji je potreban kako bi se provjera programskog koda mogla izvršiti. U „staticAnalysis.gradle.kts“ datoteci se definira *Gradle tzask* koji se poziva u skripti za kontinuiranu integraciju i isporuku. Također, moguće je pokrenuti i skriptu u terminalu u repozitoriju, te izvršiti provjeru lokalno. Za definiranje zadatka za izvršavanje potrebno je dodati svaku *gradle* datoteku koja će se izvršiti i naznačiti kako svaki od zadataka ovisi jedan o drugom. Također, postoje pravila koja se preklapaju, pa se tako često jedna prazna linija uoči u više biblioteka. Ovakav tip registriranja zadataka je moguć za bilo koji posao pa se često u praksi registriraju zadatci za kontroliranje verzije. Oni se mogu provesti i zasebno, ali ukoliko se registrira kao zadatak, onda je moguće provesti više zadataka odjednom pa se nerijetko dodaje baš statička analiza prije *git commit* naredbe za spremanje verzije programskog koda.

staticAnalysis.gradle.kts

```
apply(from = "$rootDir/staticAnalysis/lint/lint.gradle")
apply(from = "$rootDir/staticAnalysis/detekt/detekt.gradle")
apply(from = "$rootDir/staticAnalysis/ktlint/ktlint.gradle")

val staticAnalysis by tasks.registering {
    group = "verification"
    description = "Generate StaticAnalysis reports."

    dependsOn(tasks.named("ktlint"))
    dependsOn(tasks.named("lint"))
    dependsOn(tasks.named("detekt"))
}
```

***Programski kod 5.14.*** *Definiranje gradle zadatka.*

## 5.8. Serverska aplikacija

Serverska aplikacija je napisana u Ktor programskom okviru koji koristi Kotlin programski jezik. Razlog odabira programskog okvira je prethodno iskustvo s istim i poznavanje Kotlin programskog jezika. Također, programski okvir je jednostavan za uspostavljanje ruta i komunikaciju s bazom podataka. Sama aplikacija sadrži tri sloja gdje je prvi sloj ruta, zatim repozitoriji te servis koji je zaslužan za komunikaciju s udaljenom bazom podataka. Aplikacija ne izvodi dodatne operacije nad podatcima već samo dohvaća, ažurira, briše i sprema podatke u bazu podataka.

### 5.8.1. Baza podataka

Baza podataka u diplomskom radu je PostgresQL koja koristi SQL programski jezik te je ona relacijska baza podataka. Baza podataka sadrži šest tablica koje su povezane stranim ključevima.

Tablice su:

- Korisnik
    - *id* – primarni ključ
    - *full_name* – ime korisnika
    - *avatar* – url slike korisnika
    - *email* – email korisnika
    - *password* – enkriptirana zaporka korisnika
    - *salt* – dodatak zaporci koji služi za verifikaciju
    - *image_id* – id slike u datotečnom sustavu
    - *created_at* – vrijeme kreiranja korisnika

    - *date_of_birth* – datum rođenja korisnika
    - *gender* – spol korisnika
- Socijalna objava
    - *id* – primarni ključ
    - *creator_id* – id korisnika koji je kreirao objavu (referencira tablicu Korisnik)
    - *photo_url* – url slike ukoliko je dodatana
    - *photo_id* – id slike u datotečnom sustavu ukoliko je dodana
    - *num_of_likes* – broj oznaka „Sviđa mi se“ na objavi
    - *text* – text objave
    - *created_at* – datum kreiranja objave
- Objava za njegu
    - *id* – primarni ključ
    - *creator*_id - id korisnika koji je kreirao objavu (referencira tablicu Korisnik)
    - *photo_url* – url slike ukoliko je dodatana
    - *photo_id* – id slike u datotečnom sustavu ukoliko je dodana
    - *latitude* – zemljopisna širina korisnika
    - *longitude* – zemljopisna dužina korisnika
    - *address* – adresa koju je korisnik postavio
    - *price* – cijena koju je korisnik postavio
    - *description* – opis
    - *created*_at – datum i vrijeme kreiranja objave
- Razgovori
    - *id* – primarni ključ
    - *first_user_id* – id prvog korisnika (referencira tablicu Korisnik)
    - *second_user_id* – id drugog korisnika (referencira tablicu Korisnik)
- Komentari
    - *id* – primarni ključ
    - *post_id* – id objave
    - *creator_id* - id korisnika koji je kreirao komentar (referencira tablicu Korisnik)
    - *text* – text komentara
    - *created*_at – datum i vrijeme kreiranja objave
- Poruke
    - *id* – primarni ključ

- *sender_id* – id pošiljatelja (referencira tablicu Korisnik)
- *text* – text poruke
- *chat_id* – id razgovora (referencira tablicu Razgovori)
- *created*_at - datum i vrijeme kreiranja objave

Kako bi se baze podataka mogle prepoznati i spojiti za udaljenom bazom podataka koristi se tvornica baza podataka uz pomoć Hikari biblioteke. Baza podataka se kreira uz pomoć vrijednosti koje su dodane u konfiguracijskoj datoteci kako ne bi bile vidljive u programskom kodu. Također, prilikom kreiranja dodaju se sve tablice koje se koriste te vrijednosti poput *urla* udaljene baze podataka, životnog ciklusa, veličine „bazena“, bloka za transakcije itd. Funkcija *dbQuery* je funkcija koja omogućava izvođenje nad tablicom te se koristi u svakoj od klasa u *Service* sloju. Ona prima blok programskog koda, a dobiva kontekst transakcije pa s time olakšava pisanje programskog koda i izbjegava nepotrebno reproduciranje istog.

DatabaseFactory.kt

```
object DatabaseFactory {

    fun init(config: ApplicationConfig) {
        val driverClassName = config.property("ktor.database.driverClassName").getString()
        val jdbcURL = config.property("ktor.database.jdbcURL").getString()
        val username = config.property("ktor.database.user").getString()
        val password = config.property("ktor.database.password").getString()
        val defaultDatabase = config.property("ktor.database.database").getString()
        val connectionPool = createHikariDataSource(
            url = "$jdbcURL/$defaultDatabase?user=$username&password=$password",
            driver = driverClassName,
        )
        val database = Database.connect(connectionPool)
        transaction(database) {
            SchemaUtils.create(UserTable)
            SchemaUtils.create(SocialPostTable)
            SchemaUtils.create(CarePostTable)
            SchemaUtils.create(CommentsTable)
            SchemaUtils.create(MessagesTable)
            SchemaUtils.create(ChatsTable)
        }
    }

    private fun createHikariDataSource(
        url: String,
        driver: String,
    ) = HikariDataSource(HikariConfig().apply {
        driverClassName = driver
        jdbcUrl = url
        maximumPoolSize = 15
        isAutoCommit = false
        transactionIsolation = "TRANSACTION_REPEATABLE_READ"
        maxLifetime = 0
        validate()
    })

    suspend fun <T> dbQuery(block: () -> T): T = withContext(Dispatchers.IO) {
        transaction { block() }
    }
}
```

***Programski kod 5.15.*** *Tvornica baze podataka*

### 5.8.2. Rute

Rute (engl. *endpoints)* označavaju ulazne točke u server aplikaciju. Zajedno s metodama pristupa kao što su GET, PUT, POST, DELETE serverska aplikacija prepoznaje koju operaciju treba izvršiti. Također, metode pristupa, osim što se razlikuju po primjeni, one se razlikuju i po mogućnosti slanja podataka, pa tako GET metoda služi isključivo za dohvaćanje podataka, a PUT za slanje. Postoje razne metode, ali ove četiri su najkorištenije i najpoznatije. Diplomski rad se većinski sastoji od GET i POST metoda jer je većina operacija za dohvaćanje, ažuriranje i spremanje podataka.

AuthRoutes.kt

```
fun Route.register(
    repository: UserRepository,
    tokenConfig: TokenConfig
) {
    post("/register") {
        val params = call.receive<RegisterUser>()
        val result = repository.registerUser(params, tokenConfig)
        call.respond(result.statusCode, result)
    }
}


fun Route.login(
    repository: UserRepository,
    tokenConfig: TokenConfig
) {
    post("/login") {
        val request = call.receiveNullable<AuthRequest>() ?: kotlin.run {
            call.respond(HttpStatusCode.BadRequest)
            return@post
        }
        val result = repository.loginUser(params = request, tokenConfig)
        call.respond(result.statusCode, result)
    }
}

fun Route.authenticate() {
    authenticate {
        get("/authenticate") {
            call.respond(HttpStatusCode.OK, BaseResponse.SuccessResponse(data = true))
        }
    }
}

fun Application.authRoutes() {
    val tokenConfig by inject<TokenConfig>()
    val userRepository by inject<UserRepository>()
    routing {
        register(
            repository = userRepository,
            tokenConfig = tokenConfig
        )
        login(
            repository = userRepository,
            tokenConfig = tokenConfig
        )
        authenticate()
    }
}
```

***Programski kod 5.16.*** *Rute za autentifikaciju.*

Rute se u Ktor programskom okviru jednostavno mogu postaviti s metodom koja za parametre prima tekstualnu vrijednost rute i blok koda koji se treba izvršiti. Pomoću Koin biblioteke za ubrizgavanje ovisnosti se kreira odgovarajuća klasa iz Repository sloja koja se zatim poziva u samim rutama kako bi se izvršio odgovarajući postao. Svaki odgovor je dekoriran klasom BaseResponse koja može biti ili uspješna ili neuspješna. Ona u sebi također sadrži vrijednost, poruku i statusni kod kako bi se iz odgovora na klijentu moglo znati što više informacija. Ktor programski okvir također omogućava i dohvaćanje puteva, parametara i tijela zahtjeva koji se zatim verificiraju i šalju dalje repozitoriju na obrađivanje. Ukoliko zahtjev ne sadrži sve što je potrebno ili ima i viška, server automatski šalje *„Bad Request“* status.

### 5.8.3. Spremanje u bazu podataka

Kako Repository sloj zapravo samo izvršava komunikaciju sa *Service* slojem, *Service* sloj uz rute sadrži poslovnu logiku. S programskog koda 5.17. je moguće vidjeti kako se izvršavaju upiti na bazu podataka za tablicu razgovora. Bitno je napomenuti kako se za dodavanje novog retka uvijek koristi InsertStatement kako bi se moglo doći do retka koji je umetnut, dok to za dohvaćanje nije slučaj. Također, može se vidjeti logika dohvaćanja razgovora između dva korisnika. Naime, tablica razgovora služi isključivo za dohvaćanje jedinstvene vrijednosti za razgovore koja se kasnije koristi kako bi se poruke mogle dohvatiti i spremiti. Način na koji se dohvaća je taj da se uvijek gleda je li vrijednost oba korisnika jednaka jednoj od dvije spremljene vrijednosti, te ako nije, kreira se novi red u tablici. Dohvaćanjem identifikacijske vrijednosti razgovora mogu se dohvatiti razgovori, odnosno poruke između dva korisnika spremljena u tablicu poruke. Kako je ova aplikacija namijenjena samo za potrebe diplomskog rada i nema veliki broj korisnika, indeksiranje za potrebe bržeg dohvaćanja iz baze podataka nije potrebno. Kreiranje modela iz retka je moguće uz pomoć indeksiranja retka s imenom stupca kako bi se dobio atribut same klase. Takav pristup je moguć zbog korištenja datoteke Kotlin Exposed koja olakšava rad s bazom podataka. Koristi već predefinirane operatore poput *select, insert,* itd. umjesto pisanja SQL programskog jezika. Također, omogućava pisanje SQL programskog jezika u slučaju da je potrebno napraviti složenu operaciju. Iako je Kotlin *null safe* jezik, zbog mijenjanja konteksta i ulaženja u različite programske blokove potrebno je kreirati te inicijalizirati varijablu kojoj se kasnije daje vrijednost i koristi se kao povratna vrijednost funkcije. U pravilu, funkcija će vratiti *null* vrijednost samo u slučaju logičke greške, inače vraća vrijednost ili *exception*, odnosno ili funkcija uspije ili hvatamo grešku te ju ispisujemo u konzolu.

AuthRoutes.kt

```
class ChatServiceImpl : ChatService {

    override suspend fun getChat(param: GetChatRequest): Chat? {
        var chat: Chat? = null
        var statement: InsertStatement<Number>? = null

        dbQuery {
            chat = ChatsTable.select {
                (ChatsTable.secondUserId eq param.secondUserId) or
                        (ChatsTable.firstUserId eq param.secondUserId) and
                        (ChatsTable.secondUserId eq param.firstUserId) or
                        (ChatsTable.firstUserId eq param.firstUserId)
            }.firstNotNullOfOrNull(::rowToChat)
        }

        return if (chat == null) {
            dbQuery {
                statement = ChatsTable.insert {
                    it[id] = UUID.randomUUID().toString()
                    it[firstUserId] = param.firstUserId
                    it[secondUserId] = param.secondUserId
                }
            }
            rowToChat(statement?.resultedValues?.first())
        } else {
            chat
        }
    }

    override suspend fun getUserChats(userId: String): List<Chat>? {
        var chats: List<Chat>? = null

        dbQuery {
            chats = ChatsTable.select {
                (ChatsTable.secondUserId eq userId) or (ChatsTable.firstUserId eq userId)
            }.mapNotNull(::rowToChat)
        }

        return chats
    }

    private fun rowToChat(row: ResultRow?): Chat? =
        row?.let {
            Chat(
                id = row[ChatsTable.id],
                firstUserId = row[ChatsTable.firstUserId],
                secondUserId = row[ChatsTable.secondUserId]
            )
        }
}
```

***Programski kod 5.17.*** *Service za razgovore.*

### 5.8.3. Straničenje

Straničenje (engl. *pagination*) je tehnika implementiranja dohvaćanja podataka na način stranica gdje se podatci inkrementalno dobivaju u manjim dijelovima, pa tako dohvaćanja objava neće se obaviti tako da se vrate sve objave spremljene u udaljenu bazu podataka već određeni broj. Kada korisnik dođe do kraja aplikacija šalje novi zahtjev serveru za novom stranicom podataka i tako dok se ne dohvate sve objave. To se može izvesti s brojem stranica i s tehnikom popularno nazvanom *endless scroll.* U diplomskom radu se koristi tehnika *endless scroll* i za nju je potrebno slušati stanje LazyColumn komponente pomoću kojeg se dohvaća koja je trenutno objava vidljiva. Uz pomoć konstante vrijednosti koja govori koliko objava mora ostati do kraja da bi se poslao

zahtjev za novom stranicom pokušava dati dojam korisniku da ne smetano lista aplikacijom. U zahtjevu se dodaju parametri s veličinom stranice i brojem stranice kako bi server znao koje podatke treba dohvatiti. Kao što je vidljivo u programskom kodu 5.18. Algoritam za dohvaćanje objava se radi na način da se uz pomoć operatora *limit* dodaje veličina stranice i *offset* odnosno, od kojeg broja se uzima limit. Ta vrijednost se dobiva na način da se broj stranice množi s veličinom stranice.

SocialPostServiceImpl.kt

```
override suspend fun getPosts(userId: String?, limit: Int?, offset: Long?): List<SocialPost> {
    var posts: List<SocialPost?>? = null

    dbQuery {
        posts = if (userId != null) {
            SocialPostTable.select {
                (SocialPostTable.creatorId eq userId)
            }.limit(
                n = limit ?: 0,
                offset = offset ?: 0L
            ).sortedBy { it[SocialPostTable.createdAt] }.map(::rowToSocialPost)
        } else {
            SocialPostTable.selectAll()
                .limit(
                    n = limit ?: 0,
                    offset = offset ?: 0L
                ).sortedBy { it[SocialPostTable.createdAt] }.map(::rowToSocialPost)
        }
    }
    return posts?.filterNotNull() ?: emptyList()
```

***Programski kod 5.18.*** *Algoritam za stranično dohvaćanje iz baze podataka.*

U programskom kodu 5.19. je vidljivo kako uz pomoć LaunchedEffect komponente je moguće pratiti je li korisnik „dosegao dno“ kako bi se moglo dohvatiti još objava. Program će izvršiti blok u LaunchEffect komponenti tek onda kada se ona promjeni, a DerivedState je posebna vrsta stanja u JetpackCompose programskom okviru koja ažurira svoju vrijednost samo kada je to potrebno. Ovaj tip varijable se koristi kada su promjene češće nego što je potrebno ažurirati korisničko sučelje. Tako se vrijednost stanja mijenja samo kada treba prijeći iz *true* u *false* i obrnuto.

SocialWallScreen.kt

```
val listState = rememberLazyListState()

val reachedBottom: Boolean by remember {
    derivedStateOf {
        val lastVisibleItem = listState.layoutInfo.visibleItemsInfo.lastOrNull()
        lastVisibleItem?.index != 0 && lastVisibleItem?.index == listState.layoutInfo.totalItemsCount - BUFFER
    }
}

LaunchedEffect(reachedBottom) {
    if (reachedBottom) loadMore()
}

LazyColumn(
    state = listState,
    modifier = modifier,
) {
```

***Programski kod 5.19.*** *Algoritam za praćenje vidljivosti objava*

# 6. Zaključak

U današnjem svijetu koji konstantno napreduje i životni stil je vrlo brz, vrlo je teško pronaći kvalitetnu i pristupačnu uslugu za brigu o ljubimcima kada vlasnici nisu u mogućnosti. Ovaj problem se na području Balkana obično rješavao komunikacijom s drugim ljudima, odnosno pomoću preporuke ili uslugom bližnjih. Ovakav pristup ograničava kvalitetu usluge ljudi i cijenu same usluge. Nažalost, to dovodi do ne adekvatne njege o kućnim ljubimcima u određenom vremenskom periodu.

U sklopu ovog diplomskog rada kreiran je sustav sličan društvenim mrežama. Pomoću udaljene baze podataka i serverske aplikacije, korisnici su u mogućnosti komunicirati u stvarnom vremenu s drugim korisnicima, objavljivati i pregledavati sadržaj i odabrati njegu za svoje ljubimce. Zbog korištenja udaljene baze podataka i serverske aplikacije, korisnik je u mogućnosti pristupiti aplikaciji i svom računu preko bilo kojeg Android mobilnog uređaja, te se njegovi podatci neće izbrisati prilikom korištenja drugog mobilnog uređaja.

Android obilna aplikacija je osmišljena tako da zajedno sa serverskom aplikacijom omogućava kreiranje sadržaja koji može imati tekstualni i medijski sadržaj, te interakciju s drugim objavama. Također, omogućava kreiranje oglasa za pružanje usluga brige o ljubimcima što može pomoći drugim korisnicima kao novi financijski prihod. Korisnik može u odabrati oglase ovisno o lokaciji koja je omogućena uz pomoć Google Maps API i Google Places API. Ukoliko želi, korisnik ima mogućnost komuniciranja s drugim korisnicima u stvarnom vremenu. To omogućava korisnicima razmjene ideja i međusobnu edukaciju o proizvodima i načinima koji njima pomažu u njihovoj svakodnevici s njihovim kućnim ljubimcima. Korisnik ima mogućnost kreiranja različitih tipova oglasa te dodavanja medijskog sadržaja koji je spremljen u Firebase Storage datotečnom sustavu. Svaki oglas ima mogućnost označavanja sa „sviđa mi se“ što potiče korisnike da kreiraju sadržaj i dijele svoja iskustva i ideje sa drugim korisnicima. To dovodi do novih poznanstava i edukacije. Također, korisnici mogu ostaviti svoje mišljenje ili uputiti pitanje kreatoru objave pomoću komentara na objavi.

Mobilna aplikacija zajedno sa serverskom je implementirana uz pomoć najnovije tehnologije u svijetu te prati sva pravila modernog razvijanja Android aplikacija za mobilne uređaje. Tehnologije poput Jetpack Compose, Firebase, Google Places API i Google Maps API imaju primjenu i u najvećim i najkorištenijim aplikacijama u svijetu. Također, aplikacija pruža svoju arhitekturu koja potiče inovativnost i dostupna je ostalim razvojnim programerima za korištenje u svoje svrhe.

# LITERATURA


[1] Editorial: Mobile and the 2017 Developer Survey Results, SitePoint, 29-Mar-2017.

[2] Elprocus, Android and Android versions, 29.5.2022., dostupno na: https://www.elprocus.com/what-is-android-introduction-features-applications/

[3] Android Distribution Chart, 10.6.2024. dostupno na: https://www.composables.com/tools/distribution-chart

[4] Nimesh K. Ekanayake, Deparment of Information Technology, Android Operating System dostupno na: https://www.researchgate.net/publication/325257105_Android_Operating_System

[5] Procesi i životni ciklus, 11.6.2024. dostupno na: https://developer.android.com/guide/components/activities/process-lifecycle

[6] Pregled upravljanja memorijom, 11.6.2024. dostupno na: https://developer.android.com/topic/performance/memory-overview

[7] Elinux, Android arhitecture, 11.6.2024., dostupno na: https://elinux.org/Android_Architecture

[8] Platform architecture, 11.6.2024., dostupno na https://developer.android.com/guide/platform

[9] The Generic Kernal Image project, 16.6.2024., dostupno na: https://source.android.com/docs/core/architecture/kernel/generic-kernel-image

[10]D.B. & P.P, Buketa & Prasad, Jetpack Compose by Tutorials, Kodeco, 2023.

[11] Lifecycle of Composables, 16.6.2024., dostupno na: https://developer.android.com/develop/ui/compose/lifecycle

[12] Design your navigation graph, 16.6.2024., dostupno na: https://developer.android.com/guide/navigation/design

[13] FIREBASE – OVERVIEW AND USAGE, International Research Journal of Modernization in Engineering Technology and Science, Volume:03/Issue:12/December-2021

[14]Android Developers, Build, 16.6.2024, dostupno na: https://developer.android.com/studio/build

[15] Application foundamentals, 17.6.2024., dostupno na: https://developer.android.com/guide/components/fundamentals

[16] Android developers, Activity, 17.6.2024., dostupno na: https://developer.android.com/reference/android/app/Activity

[17] Android developers, Services, 17.6.2024., dostupno na: https://developer.android.com/develop/background-work/services

[18] Geeks for geeks, Services in Android with example, 17.6.2024., dostupno na: https://www.geeksforgeeks.org/services-in-android-with-example/

[19] Android developers, Broadcasts overview, 18.6.2024., dostupno na: https://developer.android.com/develop/background-work/background-tasks/broadcasts
[20] Android developers, Content provider basics, 18.6.2024., dostupno na: https://developer.android.com/guide/topics/providers/content-provider-basics
[21] Clean Architecture, Robert C. Martin, Pearson Education, Inc, 2018
[22] Android Architecture Components, Mark L. Murphy, CommonsWare, Siječanj 2019.
[23] GitHub Actions, 16.8.2024, dostupno na: https://docs.github.com/en/actions/about-github-actions/understanding-github-actions

# SAŽETAK

Razvojem mobilnih uređaja i društvenih mreža, korisnici žele imati sve informacije dostupne i vjerodostojne. Oglašavanje verbalnim putem gotovo pa više i ne postoji te ljudi vjeruju osobnom profilu, odnosno portfelju različitih osoba. Društvene mreže više nisu samo mjesto za upoznati ljude i uživati, već postaju mjesto za pronaći osobu za posao, verificirati iskustva osobe. Bitno je održavati osobni profil i predstaviti se u najboljem svijetlu na društvenim mrežama jer je to dodatni način vrednovanja osobe. Aplikacija diplomskog rada pokušava riješiti upravo taj problem. Pružajući novu platformu korisnici mogu upoznati i pronaći osobu za čuvanje njihovih ljubimaca, te također i dijeliti svoja iskustva sa drugim osobama. Objava može biti za čuvanje ili društvena, svaka objava ima dio za tekst i dio za sliku, ali objava za čuvanje mora imati i cijenu i adresu. Korisnicima je također omogućeno označavati osobu sa „Sviđa mi se“ te ju komentirati i dijeliti svoja mišljenja o njoj. Za kreiranje same aplikacije bilo je potrebno kreirati čitavog sustava koji odgovara funkcionalnim zahtjevima same aplikacije i odlučilo se za serversku i mobilnu Android aplikaciju sa udaljenom bazom podataka, Firebase Storage udaljenim računalom u oblaku i Google Maps Api servisom za prikazivanje karte i dohvaćanje informacija o adresi. Za programski jezik je odabran Kotlin jer odgovara izradi kako mobilne tako i serverske aplikacije. Za udaljenu bazu podataka je odabrana PostgreSQL baza podataka zbog njezine jednostavnosti, primjene i prijašnjeg poznavanja. Također, za kod izrade Android aplikacije prati se najmoderniji tehnološki alati, biblioteke i programski okviri pa tako za izradu korisničkog sučelja se koriste Jetpack Compose i Material, a čitava aplikacija je napisana unutar Model-View-ViewModel + Clean arhitekture. Aplikacija poštiva sva pravila modernog razvoja mobilnih aplikacija pa tako i *Unidirectional flow* i programiranje sa reaktivnim tokovima programiranja.

Ključne riječi: Android, Android Arhitektura, Android Komponente, Jetpack Compose, Kotlin.

## ABSTRACT

Mobile app for pets

With the development of smartphones and social networks, users want to have all the information available and trustworthy. Advertising by word is almost non-existent. People are increasingly starting to believe in a person's profile. Even though social networks are a place to connect with people, they are becoming a place to find the right person for the job and also, a tool to verify a person's job experience. It is important to maintain a personal profile and introduce yourself in the best way possible. The application of this master's work is trying to solve exactly that problem. By providing a new platform, users can share their experiences with other users. A post can be about caring for pets and a social one. Each post can have both pictures and text, but a caring post needs to have the price and address of the user. Users can also like a post or comment their opinions on it. In order to create the application, it was decided to go with the server and Android application with the remote database, Firebase Storage as a cloud, and Google Maps API service for showing maps and getting information about an address. For a programming language, it was decided to use Kotlin because it suits both server and Android applications. For the remote database, PostgreSQL was chosen due to its simplicity, usage, and preexisting knowledge. Also, for making the Android application, the latest tech stack and tools were used. For the UI framework, the application uses Jetpack Compose with Material and the whole application Model-View-ViewModel + Clean architecture. The application follows all the modern rules for developing an application, so it has Unidirectional Flow and reactive programming.



Keywords: Android, Android Architecture, Android Components, Jetpack Compose, Kotlin.

# ŽIVOTOPIS

Ante Lukić rođen je 10. studenog 2000. godine u Osijeku. 2015. godine završava Osnovnu školu Vladimira Nazora u Đakovu, a Srednju strukovnu školu Antuna Horvata u Đakovu, smjer Tehničar za mehatroniku završava 2019. te iste godine upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, smjer Računarstvo na preddiplomskom sveučilišnom studiju. Ante završava 3 godine preddiplomskog studija 2022. godine te upisuje diplomski studij Programsko Inženjerstvo iste godine.

Ante Lukić

_____