Android mobilna aplikacija za uživo ažuriranje stanja u prometu

Bucić, Matej

Sveučilište u Zadru

Odjel za informacijske znanosti

Stručni prijediplomski studij

Informacijske tehnologije

Matej Bucić

Android mobilna aplikacija za uživo ažuriranje stanja u prometu

Završni rad

Zadar, 2024.

Sveučilište u Zadru
Odjel za informacijske znanosti
Stručni prijediplomski studij
Informacijske tehnologije

Android mobilna aplikacija za uživo ažuriranje stanja u prometu

Završni rad

Student/ica:
Matej Bucić

Mentor/ica:
Doc. dr. sc. Tomislav Jakopec

Zadar, 2024.

Izjava o akademskoj čestitosti

Ja, **Matej Bucić,** ovime izjavljujem da je moj **završni** rad pod naslovom **Android mobilna aplikacija za uživo ažuriranje stanja u prometu** rezultat mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na izvore i radove navedene u bilješkama i popisu literature. Ni jedan dio mojega rada nije napisan na nedopušten način, odnosno nije prepisan iz necitiranih radova i ne krši bilo čija autorska prava.

Izjavljujem da ni jedan dio ovoga rada nije iskorišten u kojem drugom radu pri bilo kojoj drugoj visokoškolskoj, znanstvenoj, obrazovnoj ili inoj ustanovi.

Sadržaj mojega rada u potpunosti odgovara sadržaju obranjenoga i nakon obrane uređenoga rada.

Zadar, 13. kolovoza 2024.

SAŽETAK

Ovaj završni rad obuhvaća razvoj android aplikacije za praćenje stanja u prometu u stvarnom vremenu s fokusom na korištenje modernih tehnologija poput Firebase-ovog alata za rad s NoSQL udaljenom bazom podataka Cloud Firestore, Room lokalne baze podataka i Google Maps API-ja. Aplikacija omogućuje korisnicima pregled trenutnog stanja na cestama kao što su kolone auta na cestama, radarske kamere, prometne nesreće i policijske kontrole. Rad je podijeljen na nekoliko ključnih cjelina. Unutar prvog dijela rada su odrađene osnovne arhitektonske smjernice koje su korištene unutar aplikacije s naglaskom na Clean arhitekturu i MVVM obrazac što omogućuje jasnu raspodjelu odgovornosti između slojeva aplikacije. Također se unutar rada spominje Hilt kao korišteni alat za primjenu dependency injection-a. Drugi dio rada detaljno opisuje odabir udaljenog servera gdje je izabran Firebase-ov alat Cloud Firestore radi svojih prednosti u praćenju podataka unutar stvarnog vremena i svoje jednostavnosti upravljanja istim. Također je razjašnjena upotreba Room baze podataka kao rješenje za lokalnu pohranu podataka kako bi se omogućila sinkronizacija s podatcima dohvaćenim na udaljenom serveru. U završnom dijelu rada je pojašnjena integracija Google Maps API-ja gdje su opisane funkcionalnosti poput dodavanja markera na kartu. Dodatno je objašnjeno korištenje fragmentacije i navigacije unutar aplikacije korištenjem Android Navigation Component-a. Konačno, rad obrađuje i korištenje Kotlin coroutine-a za upravljanje nitima, s naglaskom na optimizaciju aplikacije prilikom rada s bazom podataka i mrežnim pozivima. Čitatelj može pratiti razvojni proces i tehničke detalje aplikacije putem GitHub repozitorija na sljedećoj poveznici: https://github.com/mbuc1c/Radarisha.

Ključne riječi: Android, Clean arhitektura, MVVM, Firebase, Cloud Firestore, Google Maps API, Room, Hilt, Kotlin coroutines

POPIS KORIŠTENIH KRATICA

API - Application Programming Interface (sučelje za programiranje aplikacija)

DAO - Data Access Object (objekt za pristup podatcima)

SQL - Structured Query Language (jezik za rad s relacijskim bazama podataka)

NoSQL - Not Only SQL (baze podataka koje ne koriste relacijske modele)

ORM - Object-Relational Mapping (tehnika koja omogućava mapiranje objekata u relacijske baze podataka)

UI - User Interface (korisničko sučelje)

SDK - Software Development Kit (skup razvojnih alata i biblioteka)

MVVM - Model-View-ViewModel (arhitektonski obrazac za odvajanje korisničkog sučelja od poslovne logike)

SSOT - Single Source of Truth (jedinstveni izvor istinitih podataka)

UDF - Unidirectional Data Flow (jednosmjerni tijek podataka)

XML - Extensible Markup Language (jezik za strukturiranje podataka)

DI - Dependency Injection (umetanje zavisnosti u klasu)

JSON - JavaScript Object Notationt (format za prijenos podataka)

DAO - Data Access Object (obrazac za pristup podatcima)

SHA-1 - Secure Hash Algorithm 1 (algoritam za enkripciju podataka)

SADRŽAJ

UVOD

U današnjem ubrzanom svijetu, praćenje stanja u prometu u stvarnom vremenu postaje ključna potreba za vozače. Obzirom na tu potrebu, ovaj rad približava razvoj jedne android aplikacije koja omogućuje korisnicima trenutni pregled stanja na prometnim cestama poput gužvi, kamera za brzinu, policijske kontrole i prometnih nezgoda na cesti koristeći suvremene razvojne tehnologije i najbolje prakse za razvoj mobilnih aplikacija. Ono što ovu aplikaciju čini posebno korisnom i dinamičnom jest činjenica da je ona „*crowdsourced*“ platforma unutar koje sami korisnici imaju mogućnost dodavanja i ažuriranja njenog sadržaja.

Ovaj završni rad ima cilj objasniti korake i alate korištene u razvoju ove aplikacije, uključujući alate za razvoj modernih android aplikacija kao što su Hilt za *dependency injection*, fragmentacija i navigacija unutar fragmenata, Room, Google Maps API, te Firebase i njegovu uslugu Cloud Firestore korištenu za pohranu podataka na udaljenome serveru.

Rad je strukturiran na način da prvo uvodi osnovne standarde arhitekture koji su primijenjeni, poput Clean arhitekture i MVVM (*Model-View-ViewModel*) obrasca, objašnjavajući zašto su odabrani i kako doprinose održivosti, proširivosti i pokrivenosti testovima aplikacije. Nadalje, razmatrat će se izbor specifičnih tehnologija i alata, te objašnjenje zašto su korišteni uz razmatranje mogućih alternativnih rješenja. Kroz analizu i praktične primjere, rad pruža uvid u izazove i rješenja u izgradnji pouzdane i prilagodljive mobilne aplikacije za praćenje stanja u prometu u stvarnom vremenu.

1. Osnovni standardi arhitekture

Svaka dobro organizirana aplikacija mora zadovoljavati osnovne principe moderne i dobro definirane arhitekture kako bi se postigla skalabilnost, proširivost, održivost i kvalitetna pokrivenost testovima u budućem razvoju. Osnovna ideja iza dobre arhitekture jest odvajanje odgovornosti različitih komponenti aplikacije kako bi se omogućila lakša prilagodba promjenama, ponovno korištenje koda i jednostavno održavanje.

Svaku aplikaciju možemo promatrati kao skup međusobno povezanih slojeva. U najosnovnijem smislu, aplikacija se sastoji od korisničkog sučelja (UI) i podatkovnih modela (data model), koji su zaduženi za upravljanje podatcima koji će se prikazivati korisniku. Osnovna arhitektonska ideja je odvajanje tih odgovornosti u zasebne module te dodjeljivanje svakoj komponenti specifičnih zadataka za koje su zaduženi. Time postižemo princip jedinstvene odgovornosti (*Single Responsibility Principle*), ali i ključne principe skalabilnosti, pokrivenosti testovima, te olakšane suradnje između modula. Jedan od ključnih koncepata koji se koristi u modernoj arhitekturi je SSOT (*Single Source of Truth*), što znači da postoji jedan centralni izvor istine za određeni set podataka. SSOT princip nalaže da podatci budu pohranjeni na jednom mjestu, najčešće u bazi podataka ili unutar *ViewModel-a*, a sve druge komponente aplikacije dohvaćaju te podatke kroz kontrolirane metode kako bi se osigurala konzistentnost i integritet podataka. Na primjer, unutar aplikacije, baza podataka ili *ViewModel* može biti SSOT jer je jedini izvor koji sadrži i modificira stvarne podatke, dok svi ostali slojevi aplikacije samo prikazuju podatke ili vrše akcije koje rezultiraju izmjenama u tom jedinstvenom izvoru. Također, unutar ove arhitekture koristi se *Unidirectional Data Flow* (UDF), koji definira jednosmjerni tijek podataka kroz aplikaciju, čime se osigurava jasnoća i predvidljivost ponašanja sustava. Podatci prolaze kroz tri sloja aplikacije: prezentacijski sloj (*UI layer*), podatkovni sloj (*Data layer*) i domenski sloj (*domain layer*), a izmjene podataka idu u jednom smjeru kako bi se smanjila složenost i omogućilo bolje praćenje i *debug-iranje* aplikacije.[1]

[1] Android Developers Documentation mrežna stranica, https://developer.android.com/topic/architecture [pristupljeno 9. rujna 2024.]

1.1. Clean arhitektura: princip i primjena

Clean arhitektura, koju je predložio Robert C. Martin[2], oslanja se na koncept odvajanja poslovne logike od tehničkih detalja poput upravljanja podatcima ili komunikacije s korisničkim sučeljem. Ovo odvajanje omogućava da aplikacija bude fleksibilnija i otpornija na promjene, jer se promjene u jednom sloju (npr. u korisničkom sučelju) ne reflektiraju direktno na ostatak sustava. Clean arhitektura oslanja se na slojevitu podjelu aplikacije koja poštuje principe *loose coupling-a* i SSOT-a, što znači da svaki sloj ima svoju ulogu i neovisno funkcionira od ostalih.

U aplikaciji obrađenoj u ovom radu, implementirana su tri osnovna sloja:

1. Prezentacijski sloj (*UI layer*)
2. Podatkovni sloj (*Data layer*)
3. Domenski sloj (*Domain layer*)

Svaki od tih slojeva ima jasnu ulogu i odgovornosti, čime se postiže modularnost i održivost koda. Unutar aplikacije slojevi su definirani modulima.

Slika 1. Prikaz modula korištenih u projektnom zadatku[3]

Na slici 1. su prikazani moduli kreirani unutar aplikacije kako bi se postigla odgovarajuća raspodjela slojeva unutar aplikacije. Modul koji prezentira prezentacijski sloj (*presentation module*) je poznat i kao aplikacijski modul (*app module*).

1.1.1. Prezentacijski sloj (*UI layer*)

Zadatak prezentacijskog sloja je da prikazuje podatke aplikacije na korisničkom sučelju. Ako dođe do promjene unutar podataka iz podatkovnog sloja ili iz domenskog sloja, bilo interakcijom korisnika ili eksternim promjenama, poput odgovora sa udaljenog servera,

[2] Martin, Robert C., Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017., str. 287 - 293

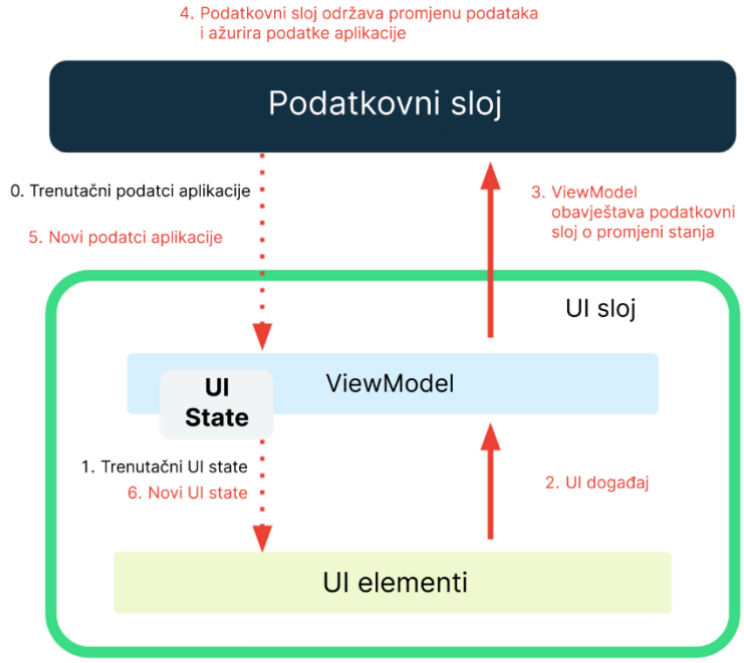
[3] Snimka zaslona iz praktičnog dijela rada

prezentacijski sloj ima zadaću prikazati iste promjene na korisničkom sučelju. Prezentacijski sloj se sastoji od:

1. UI elemenata koji daju prikaz podataka na ekranu.
2. *State holders-a* poput *ViewModel-a* koji drže podatke i šalju ih na korisničko sučelje te obavljaju logičke zadatke s tim podatcima.

Unutar aplikacijskog rješenja namijenjenog za temu ovog završnog rada, Korišteni su *Views* UI elementi za prikaz podataka, no postoji i alternativa kroz korištenje *Jetpack Compose-a,* modernijeg pristupa gradnji korisničkog sučelja unutar android aplikacija.[4]

Ključna komponenta za pohranjivanje *UI state-a* je *ViewModel* unutar koje su pohranjeni svi potrebni podatci za prikaz na korisničkom sučelju. Podatci koji su pohranjeni unutar *ViewModel-a* se ili zaprimaju iz podatkovnog sloja i tako šalju prema UI klasi, poput aktivnosti ili fragmenta, ili se iz UI klase putem korisnikove interakcije s aplikacijom podatci modificiraju i šalju prema podatkovnom sloju. Takva komunikacija između podatkovnog i prezentacijskog sloja je srž UDF-a (*Unidirectional Data Flow*). Na taj način ViewModel pohranjuje ažurne podatke unutar sebe.[5]



Slika 2. Dijagram ciklusa događaja i podataka unutar UDF-a[6]

[4] Android Developers Documentation mrežna stranica, https://developer.android.com/topic/architecture#ui-layer, [pristupljeno 9. rujna 2024.]
[5] Android Developers Documentation mrežna stranica, https://developer.android.com/topic/architecture/ui-layer, [pristupljeno 9. rujna 2024.]
[6] Android Developers Documentation mrežna stranica, https://developer.android.com/static/topic/libraries/architecture/images/mad-arch-ui-udf-in-action.png, [pristupljeno 9. rujna 2024.]

1.1.2. Podatkovni sloj (*Data layer*)

Podatkovni sloj je zadužen za upravljanje svim podatcima unutar aplikacije. To uključuje komunikaciju s udaljenim serverima, lokalnom bazom podataka te bilo kojim drugim izvorima podataka. Uloga podatkovnog sloja je da je zadužen za dohvaćanje podataka, njihovo pohranjivanje i ažuriranje te slanje u domenski ili prezentacijski sloj. Može se reći kako podatkovni sloj posjeduje poslovnu logiku same aplikacije. Podatkovni sloj čine *data source* klase koje ujedno služe i kao izvori podataka prenamijenjenog izvora. Unutar aplikacije je moguće posjedovati beskonačno ili pak nijedan izvor podataka ovisno o zahtjevima i potrebama aplikacije. Svaka *data source* klasa mora raditi na samo jednom izvoru podataka. Također podatkovni sloj se sastoji od repozitorijskih klasa čija je uloga definirati logiku kako će podatkovni sloj manipulirati podatcima i koje će izvore koristiti za pojedine zadatke. Svaki tip podatka ima svoj repozitorij. Na konkretnom primjeru aplikacije, postoje *UserRepository* koji definira logiku samo za obradu korisničkih podataka unutar aplikacije i *RadarRepository* koji definira logiku samo za radare koji se prikazuju na korisničkom sučelju. Oba repozitorija slijede istu logiku, a imaju različite zadaće. Repozitoriji unutar svojih konstruktora definiraju izvore podataka. Ovaj sloj također koristi SSOT princip, jer osigurava da svi podatci dolaze iz jednog izvora istine, bilo da je riječ o udaljenom serveru ili lokalnoj bazi podataka. Repozitorij klase su odgovorne za:

1. Izlaganje podataka ostatku aplikacije
2. Centraliziranje promjena podataka
3. Rješavanje sukoba između više izvora podataka
4. Apstrahiranje izvora podataka od ostatka aplikacije
5. Sadrže poslovnu logiku aplikacije[7]

Unutar aplikacije su korišteni „*remote*“ i „*local*“ izvori podataka, svaki definiran svojim sučeljem. Za udaljene izvore podataka je korišten Cloud Firestore, kojim je omogućena brza i efikasna pohrana podataka na udaljeni server i praćenje promjena istih podataka u stvarnome vremenu. Za lokalne izvore podataka je korišten Room kao lokalna baza podataka čime osiguravamo pristup podatcima izvan internetske mreže te njihovu sinkronizaciju s udaljenim podatcima kada korisnik nema pristup internetu.

[7] Android Developers Documentation mrežna stranica, https://developer.android.com/topic/architecture/data-layer, [pristupljeno 10. rujna 2024.]

1.1.3. Domenski sloj (*Domain layer*)

Domenski sloj je opcionalan sloj, ali se često koristi u složenijim aplikacijama kako bi se enkapsulirala kompleksna poslovna logika, ili poslovna logika upotrjebljena preko više *ViewModel* klasa od tehničkih detalja. Ovaj sloj sadrži poslovne entitete, logiku i pravila aplikacije koja su neovisna o korisničkom sučelju i o izvorima podataka. Unutar domenskog sloja se mogu pronaći *use case* klase s kojima je definirana jedna funkcionalnost gdje operiraju nad podatcima dohvaćenima iz repozitorija.[8]

Na primjer, ako primjer aplikacije prati prometne događaje, u domenski sloj se može jednostavno dodati logika poput filtriranja relativnih radara za korisnika ili filtriranje radara prema određenoj udaljenosti koju korisnik odabere. Ovaj sloj omogućava izolaciju ključne poslovne logike, što čini aplikaciju fleksibilnijom i omogućava jednostavniju promjenu korisničkog sučelja ili podatkovnog sloja bez modificiranja poslovne logike.

1.2. MVVM obrazac: uloga i prednosti

MVVM (*Model-View-ViewModel*) je arhitektonski obrazac koji se najčešće koristi u razvoju android aplikacija. Ovaj obrazac pruža jasnu strukturu za povezivanje poslovne logike s korisničkim sučeljem, omogućujući bolju pokrivenost testovima i održivost aplikacije. MVVM obrascem, kod odvajamo u tri osnovne komponente:

- Model
- *View*
- *ViewModel*

Model ima jednostavnu ulogu pohranjivanja i pružanja podataka unutar klase kojom je definiran. Model je učestalo povezan s podatkovnim slojem putem raznoraznih izvora podataka. U slučaju aplikacije, podatci prikupljeni sa udaljenog servera ili lokalne baze podataka koji su predstavljeni svojim entitetom tj. svojom klasom, mapirani su u entitet prilagođen za prezentacijski sloj aplikacije korištenjem kreiranih *mapper-a*[9]. Entitet *RadarMarker*[10] za

[8] Android Developers Documentation mrežna stranica, https://developer.android.com/topic/architecture#domain-layer, [pristupljeno 10. rujna 2024.]
[9] GitHub repozitorij projektnog zadatka, https://github.com/mbuc1c/Radarisha/blob/main/presentation/src/main/java/com/bucic/radarisha/mapper/RadarPresentationMapper.kt, [pristupljeno 11. rujna 2024.]
[10] Github repozitorij projektnog zadatka, https://github.com/mbuc1c/Radarisha/blob/main/presentation/src/main/java/com/bucic/radarisha/entities/RadarMarker.kt, [pristupljeno 11. rujna 2024.]

prezentacijski sloj konkretno korišten kao model unutar MVVM obrasca, korišten je ponajviše za prikaz istih entiteta na korisničkom sučelju. *View* predstavlja korisničko sučelje aplikacije s kojim će korisnik vršiti interakciju. Unutar ove aplikacije, *view* je implementiran kroz tradicionalne *View* komponente pisane u XML formatu. Alternativa tom rješenju stoji u korištenju modernije metode kreiranja korisničkog sučelja korištenjem *Jetpack Compose-a*. Za prikaz *view* komponenti mogu se koristiti aktivnosti (*AppCompatActivity*) ili fragmenti. *ViewModel* je ključna komponenta MVVM-a. njegova uloga je upravljati podatcima koji se prikazuju u *View* sloju, te služi kao posrednik između korisničkog sučelja i poslovne logike. *ViewModel* je poznat, kako smo prije naveli, i kao *state holder* čija je funkcija osigurati da se sve promjene u modelu automatski ažuriraju i prenesu na korisničko sučelje. Samim time se postiže željeni rezultat UDF-a. MVVM obrazac poboljšava pokrivenost testovima budući da je *ViewModel* odvojen od *View-a*. Samim tim je lakše testirati poslovnu logiku neovisno o korisničkom sučelju. Također slijedi *loose coupling* princip gdje MVVM odvaja poslovnu logiku od prikaza korisničkog sučelja, što rezultira olakšanoj prilagodbi i danjem proširenju aplikacije bez velikih izmjena ključnih komponenti.[11]

Također, MVVM olakšava korištenje alata za reaktivno programiranje kao što su *LiveData* i *Flow* uz pomoć kojih je promjene na podatcima moguće automatski ažurirati na sučelje korisnika čim se izmjena dogodi. Samim time se smanjuje količina potrebnog koda i osigurava ažurnost podataka i brza sinkronizacija između podatkovnog i prezentacijskog sloja.

1.3. Odabir alata za *dependency injection*: Hilt

Dependency Injection (DI) je važan obrazac dizajna kojim pojednostavljujemo upravljanje zavisnostima unutar aplikacije, pružajući bolju modularnost, skalabilnost i pokrivenost testovima. DI omogućuje vanjsko injektiranje potrebnih zavisnosti umjesto da svaka klasa sama stvara svoje instance, što na posljetku značajno olakšava održavanje koda i smanjuje međusobnu povezanost između klasa. U svijetu androida, jedan od najpopularnijih alata za implementaciju DI-a je Hilt, koji je izgrađen na bazi *Dagger DI framework-a*. Hilt je poseban po tome što je izgrađen s pojednostavljenim API-jem ciljanim za android ekosustav.[12]

[11] GeeksforGeeks mrežna stranica, https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/, [pristupljeno 11. rujna 2024.]
[12] Android Developers Documentation mrežna stranica, https://developer.android.com/training/dependency-injection, [pristupljeno 12. rujna 2024.]

Hilt daje mogućnosti za jednostavnom integracijom *dependency injection-a* eliminirajući potrebu za ručnom konfiguracijom modula i komponenti uz korisne anotacije koje značajno smanjuju složenost kodiranja i preglednost samog koda. Prilikom kompiliranja aplikacije, Hilt kreira graf koji pruža nužne zavisnosti među klasama. Ako postoji bilo kakav problem ili neispravno postavljanje zavisnosti, Hilt će prikazati grešku prilikom kompiliranja i dati na uvid gdje je nastala greška.[13]

Neke od glavnih prednosti Hilt-a su jednostavnost po kojoj je vrlo poznat. Za razliku od Dagger-a koji koristi složene konfiguracije, Hilt koristi unaprijed definirane android specifične anotacije koje olakšavaju konfiguraciju *dependency injection-a*. Klase se jednostavno označuju odgovarajućim anotacijama, a Hilt se brine da se te klase kreiraju i brine se o upravljanju njihovim zavisnostima kroz životni vijek aplikacije. Hilt također olakšava testiranje tako što nudi mogućnost jednostavnog injektiranja „lažnih" zavisnosti unutar jediničnih testova. Na primjer, umjesto stvarne zavisnosti poput stvarne baze podataka ili mrežnog API-ja za rad na udaljenom serveru, uz pomoć Hilt-a moguće je kreirati lažne zavisnosti koje su korištene isključivo za svrhu testiranja funkcija aplikacije. Samim time su sačuvani podatci aplikacije i testovi postaju brži, izoliraniji i jednostavniji za praćenje i održavanje. Hilt je ujedno značajan iz razloga što pruža modularnost aplikaciji, jer svaki sloj ima mogućnost upravljanja svojim zavisnostima. S tom mogućnosti znatno je olakšano održavanje koda i dodavanje novih funkcionalnosti. Na konkretnom primjeru, izrađena su tri različita modula za upravljanje mrežnim zavisnostima, bazom podataka i poslovnom logikom[14] kako međusobno ne bi bili povezani čime kod postaje znatno pregledniji. Da bi se Hilt mogao koristiti u android aplikaciji, potrebno je prvo dodati potrebne *plugin-ove* i ovisnosti unutar *build.gradle* datoteke. Unutar aplikacije *plugin-ovi* i ovisnosti su dodane u *build.gradle* datoteku prezentacijskog modula. Zatim je važno anotirati aplikacijsku klasu s anotacijom @HiltAndroidApp što daje aplikaciji na znanje da je Hilt zadužen za kreiranje zavisnosti unutar aplikacije. Nakon toga, Hilt automatski upravlja životnim ciklusima i kreira potrebne komponente za svaku aktivnost, fragment, *ViewModel* ili ostale klase definirane unutar modula. Hilt se oslanja na nekoliko ključnih anotacija kako bi definirao gdje i kada se zavisnosti injektiraju. Neke od najučestalijih anotacija korištenih unutar projekta su @Inject koja se koristi za označavanje konstruktora, polja ili metoda kojima će Hilt dodijeliti zavisnosti. Također su korištene anotacije poput

[13] Repčík Tomáš, 2023., https://tomas-repcik.medium.com/dependency-injection-with-hilt-in-android-development-e23fc636d65c, [pristupljeno 27. rujna 2024.]
[14] GitHub repozitorij projektnog zadatka, https://github.com/mbuc1c/Radarisha/tree/main/presentation/src/main/java/com/bucic/radarisha/di/module, [pristupljeno 12. rujna 2024.]

@HiltViewModel koja govori Hilt-u da je riječ o *ViewModel* klasi, kao i anotacija @AndroidEntryPoint koja se koristi za anotaciju svih aktivnosti i fragmenata kako bi Hilt znao koristiti te klase za injektiranje zavisnosti. Konkretan primjer korištenja tih anotacija iz primjera aplikacije unutar svih *ViewModel* klasa[15] i unutar aktivnosti i fragmenata[16]. Kao što je moguće zaključiti, za konstruiranje *ViewModel* klasa nužno je korištenje @Inject anotacije koju prati „*constructor*" operator kojim definiramo konstruktor klase iz razloga što su unutar konstruktora navedene zavisnosti same *ViewModel* klase. Ova aplikacija koristi *use case* klase kao zavisnosti unutar *ViewModel-a*, no moguće je direktno postaviti i repozitorijske klase u slučaju da aplikacija ne koristi *use case-ove*. Za razliku od *ViewModel-a,* aktivnosti i fragmenti nemaju svoje zavisnosti pa samim time i nije potrebno injektiranje putem konstruktora. Jedan od ključnih dijelova Hilt-a su moduli. Moduli definiraju kako Hilt stvara objekte i upravlja zavisnostima koje ne mogu biti injektirane putem konstruktora. Modul je klasa anotirana s @Module popraćena s anotacijom @InstallIn koja govori Hilt-u koja android klasa će koristiti taj modul ili unutar koje klase će se instalirati. Unutar modula, funkcije zadužene za kreiranje objekata moguće je definirati uz pomoć dvije vrste anotacija. Prva anotacija korištena unutar aplikacije je @Provides. Ona je najčešće korištena za stvaranje objekata klasa kojima je vlasnik vanjska biblioteka poput Room-a ili ako se sučelje mora kreirati uz pomoć *Builder* obrasca. Druga anotacija je @Binds koja se koristi u slučaju da je potrebno injektirati određenu implementaciju sučelja. Ona omogućuje Hilt-u da zna koju klasu koristiti kad god je traženo neko sučelje. Za razliku od @Provides anotacije, @Binds anotacija nema tijelo kojim se definira na koji će način Hilt kreirati objekt, već funkcionira na principu definiranja apstraktnih metoda za čiji argument stoji implementacija sučelja, a kao povratni tip metode stoji samo sučelje.[17]

Također se koristi anotacija @Singleton koja definira Hilt-u da će jednom kreirati taj objekt i njegov životni vijek će trajati kroz cijeli životni ciklus aplikacije i pritom će se ta ista instanca koristiti gdje god je potrebno.[18] Primjer korištenja @Singleton anotacije je prikazan u

[15] GitHub repozitorij projektnog zadatka, https://github.com/mbuc1c/Radarisha/blob/main/presentation/src/main/java/com/bucic/radarisha/ui/radar/RadarViewModel.kt#L19, [pristupljeno 12. rujna 2024.]
[16] GitHub repozitorij projektnog zadatka, https://github.com/mbuc1c/Radarisha/blob/main/presentation/src/main/java/com/bucic/radarisha/ui/radar/RadarActivity.kt#L29, [pristupljeno 12. rujna 2024.]
[17] Android Developers Documentation mrežna stranica, https://developer.android.com/training/dependency-injection/hilt-android, [pristupljeno 12. rujna 2024.]
[18] Vivo Manuel, 2020., https://medium.com/androiddevelopers/scoping-in-android-and-hilt-c2e5222317c0, [pristupljeno 12. rujna 2024.]

aplikativnom rješenju kod kreiranja Room baze podataka koja zahtjeva mnogo resursa za kreiranje.[19]

Unutar aplikacije za praćenje stanja u prometu, Hilt je korišten za upravljanje zavisnostima poput *ViewModel-a*, aktivnosti, fragmenata, repozitorija, API servisa, baze podataka, *use caseova* i drugih. Na primjer, kako je već navedeno, *ViewModel* zavisi o *use case* klasama koje čuvaju poslovnu logiku aplikacije. One zavise o repozitorij klasama koje određuju odakle će se podatci prikupljati. Repozitorijske klase zavise o *data source* klasama koje na posljetku vrše stvarnu komunikaciju bilo sa bazom podataka ili sa udaljenim serverom na Cloud Firestore-u. Korištenjem Hilt biblioteke riješili smo problem da svaka komponenta ne treba sama ručno stvarati svoju instancu već sam Hilt omogućuje injektiranje potrebnih zavisnosti u odgovarajuće klase.

2. Odabir udaljenog servera za pohranu podataka

Pri izgradnji aplikacija koje zahtijevaju kontinuirano prikupljanje i upravljanje podatcima unutar stvarnog vremena, bitno je izabrati kvalitetnog i praktičnog poslužitelja. Za ovaj projekt je korišten Cloud Firestore kao udaljena baza podataka zbog njegovih značajnih prednosti u odnosu na alternativna rješenja.

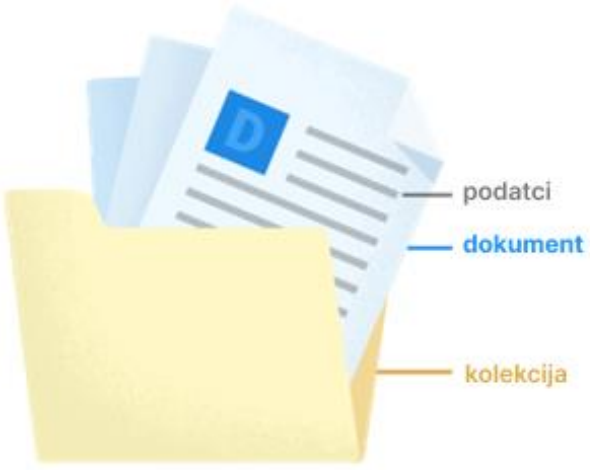
Jedna od alternativa koju nudi sam Cloud Firebase je korištenje njegove *Realtime Database* distribucije. Iako *Realtime Database* zvuči kao pravi izbor za slučaj aplikacije, mogućnosti koje nudi sam Cloud Firestore su sasvim dovoljne za produkciju aplikacije za praćenje prometa. Još jedna od dobrih alternativa je kreiranje vlastitog REST API-ja. Kreiranje vlastitog REST API-ja bi bilo dobro rješenje radi same fleksibilnosti oko samog koda gdje imamo potpunu kontrolu nad strukturom baze podataka, ali također jedna od mana jest sama kompleksnost koja bi oduzela fokus na sam razvoj mobilne aplikacije unutar ovog rada. Stoga je Cloud Firestore odličan izbor radi svoje jednostavnosti gdje smanjuje vrijeme samog razvoja i omogućuje brzu integraciju te unutar sebe podržava ažuriranja u stvarnom vremenu bez utrošenog dodatnog napora.

[19] GitHub repozitorij projektnog zadatka, https://github.com/mbuc1c/Radarisha/blob/main/presentation/src/main/java/com/bucic/radarisha/di/module/DatabaseModule.kt#L24, [pristupljeno 12. rujna 2024.]

2.1. Cloud Firestore

Cloud Firestore je fleksibilna i skalabilna NoSQL baza podataka, temeljena na Google Cloud infrastrukturi koja služi za pohranu i sinkronizaciju podataka koja niz platformi unutar kojih se može implementirati kao što su mobilne i web aplikacije. Poput *Realtime Database-a*, Cloud Firestore nudi praćenje podataka i njihovu sinkronizaciju s klijentom putem metoda za osluškivanje promjena na serveru koje nude i podršku rada u slučajevima gdje nema internet konektivnosti ili je latencija pre visoka što aplikaciju čini responzivnom na zahtjeve klijenta.[20]

Pristup Cloud Firestore-u za android aplikaciju je osiguran direktno korištenjem svoga SDK-ja. Kako je već navedeno, Cloud Firestore se temelji na NoSQL bazi podataka, gdje su podatci mapirani parovima ključ-vrijednost i pohranjeni unutar dokumenata. Dokumenti su pohranjeni unutar kolekcija koje su organizirane kao kontejneri koji su korišteni za organizaciju baze podataka i kreiranje poziva na bazu podataka. Dokumenti podržavaju spremanje raznoraznih tipova podataka poput brojeva, *string-ova* pa tako i kompleksnih objekata. Također podržavaju kreiranje kolekcija unutar dokumenata gdje se mogu nizati i tako kreirati hijerarhijska struktura podataka. To znači da podatke možemo organizirati i pohranjivati na fleksibilan način, bez prethodnog definiranja rigidne strukture kao što se to radi u slučaju relacijskih baza podataka. Takav način nudi prednost u bržem razvoju i prilagodbi aplikacije prema njenim budućim izmjenama.[21]



Slika 3. Slikoviti prikaz organizacije podataka unutar Cloud Firebase-a[22]

Unutar konkretnog primjera aplikacije, postoje dvije osnovne kolekcije:

[20] Firebase mrežna stranica, https://firebase.google.com/docs/firestore, [pristupljeno 13. rujna 2024.]
[21] Firebase mrežna stranica, https://firebase.google.com/docs/firestore/data-model, [pristupljeno 13. rujna 2024.]
[22] Firebase mrežna stranica, https://firebase.google.com/static/docs/firestore/images/structure-data.png, [pristup 13. rujna 2024.]

- *users*: kreirana za pohranu dokumenata koji predstavljaju svakog registriranog korisnika aplikacije
- *radars*: kreirana za pohranu radara prijavljenih od strane korisnika aplikacije.

Korisnici imaju osnovne informacije poput *username* polja koje predstavlja korisničko ime i *password* polja koje predstavlja korisnikovu lozinku kojom se prijavljuju na svoj profil. Radari su nešto složeniji po pitanju podataka koji su pohranjeni unutar njihovih dokumenata kao što su koordinate radara na kojima je postavljen, tip radara koji je kreiran, datum i vrijeme kada je radar kreiran i kada je radar ažuriran te univerzalni identifikator korisnika koji je kreirao isti radar. Unutar svakog radar dokumenta se kreira *reliability* kolekcija koja sprema dokumente koji predstavljaju pouzdanost radara. *Reliability* dokumenti su kreirani od strane drugih korisnika koji nisu kreirali radar. Ti dokumenti posjeduju podatke kada je dokument kreiran i ažuriran, unikatan identifikator korisnika koji je glasao te sam glas koji je prikazan u obliku *Boolevog* tipa. Svakom dokumentu koji je kreiran je automatski dodijeljen unikatan identifikator u obliku *string-a* koji je nasumično generiran kako bi se postigla bolja konkurentnost i samim time spriječilo dupliciranje identifikatora. Univerzalnim identifikatorima se vrši poziv na sam dokument i uz pomoć njega su dohvaćeni svi potrebni podatci.

Iako je Cloud Firestore odabran zbog svoje strukture i prednosti, moguće je koristiti i Firebase-ovu *Realtime Database* bazu podataka. Ova baza podataka također omogućuje praćenje podataka u stvarnom vremenu, no postoji nekoliko ključnih razloga zašto je Cloud Firestore preferiran. Što se same strukture podataka tiče kao što je navedeno, Cloud Firestore je strukturiran od kolekcija i dokumenata dok *Realtime Database* koristi jednostavnu JSON strukturu gdje su podatci pohranjeni kao veliko stablo s granama što kasnije dovodi do komplikacija kada aplikacija postane složenija. Cloud Firestore koristi uređene funkcije za upite na server koji su implementirani putem SDK-ja i nude mogućnosti složenijih upita za razliku od *Realtime Database-a* koji nudi osnovne upite. To znači ako postoji potreba za izvršavanjem složenijeg upita, prvotno se moraju učitati veći dijelovi podataka te ih je potrebno naknadno filtrirati na klijentskoj strani. Cloud Firestore koristi gRPC kako bi omogućio brzu, pouzdanu i učinkovitu komunikaciju između servera i klijenta. Ovo posebno pomaže kada aplikacija mora izvesti kompleksnije operacije ili upravljati velikim brojem zahtjeva odjednom kao što su upiti s filtriranjem, sortiranjem ili grupiranjem podataka što gRPC protokol čini odličnim za složenije i skalabilne aplikacije. Za razliku od Firestore-a, *Realtime database* koristi *WebSocket* za uspostavljanje stalne veze između klijenta i servera. To znači da promjene u bazi dolaze odmah

do klijenta u stvarnome vremenu bez potrebe da klijent stalno šalje zahtjeve za novim podatcima. Takav pristup je idealan za aplikacije kojima je potrebna konstantna i brza dostupnost novim podatcima kao što su podatci unutar aplikacija za komunikaciju ili kod igara. Cloud Firestore je dizajniran za takozvano horizontalno skaliranje što znači da bolje funkcionira kada aplikacija ima veću bazu korisnika. Također nudi podršku na klijentskoj strani van internetske mreže, što omogućava korisnicima nesmetani rad na aplikaciji i kada izgube internetsku vezu te pritom sinkronizira sve promjene kada se vrati veza. S druge strane *Realtime Database* također nudi podršku korisnicima koji su van internetske mreže, ali sama skalabilnost aplikacije vrlo lako može postati problematična kako potrebe aplikacije rastu. Razlog tomu je što R*ealtime Database* baza podataka nije dizajna da se lako nosi s velikim količinama podataka i kompleksnim upitima. Samim time ima veću tendenciju postati manje učinkovita u većim aplikacijama.[23]

Postavljanje Cloud Firestore-a je poprilično intuitivno i jednostavno i ono zahtjeva kreiranje projekta i odabira Firebase-ovih alata te samog izbora baze podataka. Taj proces se izvršava putem konzole na službenoj Firebase web stranici. Nakon što se postavi server potrebno je implementirati zavisnosti unutar *build.gradle* dokumenta unutar aplikacijskog modula, ili u slučaju aplikacije, prezentacijskog modula. Cloud Firestore nudi jednostavan rad s podatcima korištenjem jednostavnih i intuitivnih funkcija kojima je moguće dodavati nove podatke u definiranu kolekciju, čitati iste podatke, brisati te ažurirati podatke ovisno prema potrebama koje aplikacija zahtjeva. Svaka modifikacija se odvija putem *FirebaseFirestore* klase putem koje se poziva referenca na kolekciju ili dokument unutar nje. Instanca *FirebaseFirestore* klase je pružana putem Hilt-a unutar mrežnog modula.[24] Podatke s kojima radi *FirebaseFirestore* klasa mogu biti pružani na dva različita načina. Jedan od njih je putem *data class* objekata koji omogućuju strukturiran rad s podatcima. Drugi način je putem *Map* objekata koji pružaju više fleksibilnosti u slučajevima kada struktura podataka nije unaprijed poznata ili se može naknadno mijenjati.[25]

[23] Firebase mrežna stranica, https://firebase.google.com/docs/firestore/rtdb-vs-firestore, [pristupljeno 13. rujna 2024.]
[24] GitHub repozitorij projektnog zadatka, https://github.com/mbuc1c/Radarisha/blob/main/presentation/src/main/java/com/bucic/radarisha/di/module/NetworkModule.kt#L25, [pristupljeno 13. rujna 2024.]
[25] Firebase mrežna stranica, https://firebase.google.com/docs/firestore/quickstart, [pristupljeno 13. rujna 2024.]

2.2. Praćenje podataka u stvarnom vremenu

Jedan od glavnih razloga zašto je Cloud Firestore odabran kao baza podataka za ovu aplikaciju je njegova sposobnost praćenja podataka unutar stvarnog vremena putem *onSnapshotListener* mehanizma. Ovaj mehanizam omogućava aplikaciji da osluškuje Cloud Firestore bazu podataka kada se podatci promjene unutar nje. Kada dođe do promjene u podatcima, klijent zaprima obavijesti o toj promjeni bez potrebe za ručnim osvježavanjem podataka što čini aplikaciju reaktivnom. Ovaj mehanizam je ključan za razvoj ove aplikacije kako bi se osigurali najažurniji podatci o stanju na prometnicama poput novih prijavljenih prometnih nesreća i ostalih radara kako bi korisnik lakše mogao planirati svoj put. Kada se koristi *onSnapshotListener*, aplikacija registrira „slušatelja“ na određenu kolekciju ili dokument unutar Cloud Firestore-a. Na taj način se osluškuju sve promjene unutar te kolekcije ili dokumenta te se automatizmom ažurirani podatci šalju klijentu. Na primjeru aplikacije, kada jedan korisnik prijavi prometnu nesreću, svi korisnici će automatski zaprimiti novo stanje i vidjet će novu prijavljenu prometnu nesreću na svojoj karti bez potrebe za ponovnim učitavanjem podataka.[26]

onSnapshotListener funkcionira na principu da klijentu šalje samo promjene umjesto cijelog poziva što je puno efektivnije i jeftinije za resurse. Promjene registrira putem *QuerySnapshot* klase unutar čijeg atributa su spremljeni objekti klase *DocumentChange* koji nam služi kao referenca za izmijenjeni, novo kreirani ili uklonjeni dokument. *DocumentChange* klasa posjeduje atribut „*document*“ koji sprema objekt klase *QueryDocumentSnapshot* koji je naknadno mapiran u poželjni entitet.[27]

Za praćenje promjena unutar stvarnog vremena, korišten je *callback* mehanizam jer pruža direktan način obavještavanja kada dođe do promjena na bazi podataka, specifično, kada *onSnapshotListener* zaprimi informaciju o promjeni, on poziva *callback* metode unutar sebe kako bi izvršio željenu operaciju. *Callback-ovi* su definirani sučeljem gdje njihova implementacija stoji unutar aplikacije, u primjeru aplikacije unutar prezentacijskog sloja. Osim što je *callback* mehanizam jednostavan za podešavanje željenih funkcionalnosti, ovaj mehanizam također ima nekoliko mana. S vremenom, kako se složenost aplikacije povećava, tako može doći do problema poznatog kao „*callback hell*“. To je naziv za situaciju unutar koje

[26] Firebase mrežna stranica, https://firebase.google.com/docs/firestore/query-data/listen#kotlin+ktx_1, [pristupljeno 13. rujna 2024.]
[27] GitHub repozitorij projektnog zadatka, https://github.com/mbuc1c/Radarisha/blob/main/data/src/main/java/com/bucic/data/network/firestore/radar/RadarFireStoreImpl.kt#L39, [pristupljeno 13. rujna 2024.]

se višestruki *callback-ovi* međusobno isprepliću, što kod čini nepreglednim i teškim za održavanje. Također *callback-ovi* nisu uvijek idealni za testiranje asinkronih operacija.[28] S druge strane postoji dobra alternativa naprema *callback* poziva, a to je korištenje *Flow-a*. *Flow* je napredniji mehanizam za praćenje promjena unutar podataka. *Flow* je dio *Coroutines* biblioteke i samim time omogućuje asinkrono upravljanje podatcima kao tokovima koji emitiraju vrijednosti tijekom vremena dok su korišteni. S *Flow-ovima* je lakše upravljati podatcima jer će *onSnapshotListener* stalno emitirati nove podatke kada dođe do promjene unutar njih. *Flow* nudi opciju praćenja tokova paralelno čije se praćenje može jednostavno prekinuti ili pauzirati. Tokovi podataka unutar *Flow-a* se mogu lako pratiti koristeći jednu od ugrađenih funkcija *collect* gdje se svaka promjena može popratiti i postupati prema njoj.[29]

Unatoč svim navedenim prednostima, implementacija *callback* mehanizma je prevladala radi svoje jednostavnosti i brzine razvoja obzirom da zahtjevi koje oni moraju odraditi nisu pretežito složeni.

2.3. Sigurnost i autorizacija pristupa podatcima

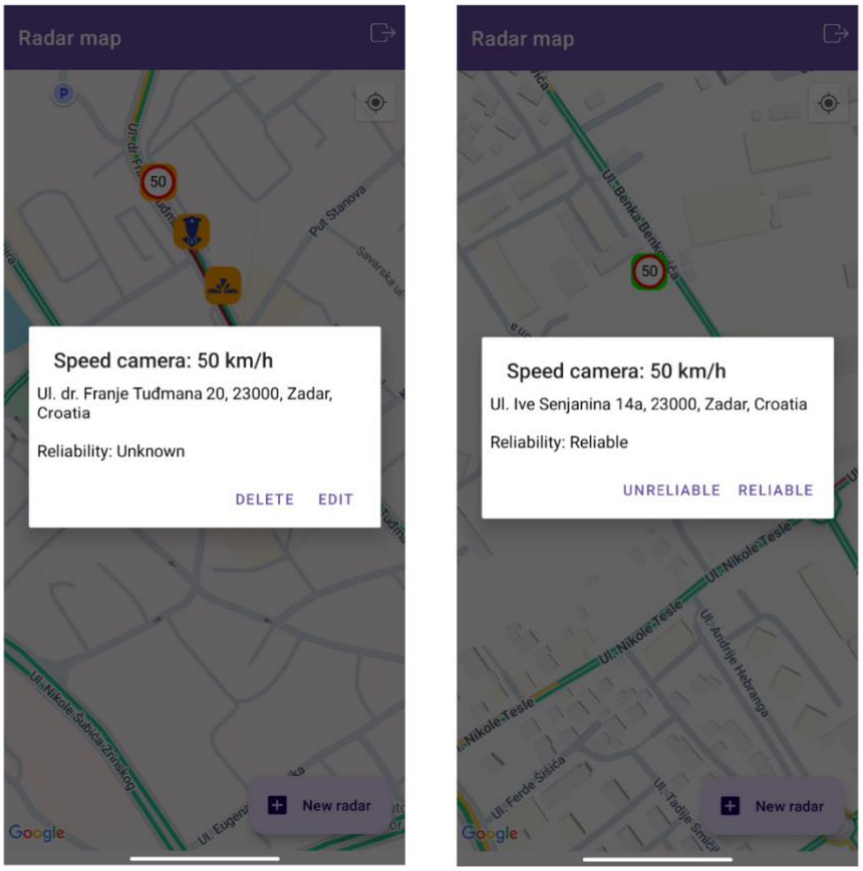
Sigurnost i zaštita podataka ključni su kod razvoja aplikacija s pohranom korisničkih podataka i njihovom obradom. Cloud Firestore nudi niz alata za zaštitu podataka, uključujući sigurnosna pravila koja se podešavaju unutar konzole i automatsku enkripciju podataka pohranjenih unutar te baze podataka. Podatci unutar Cloud Firestore-a su automatski kriptirani tijekom samog prijenosa i pohrane, što osigurava da su svi osjetljivi podatci zaštićeni od neovlaštenog pristupa.[30]

Cloud Firestore također omogućuje definiranje pravila pristupa korištenjem Firebase Security Rules-a, koji definiraju tko može vršiti zahtjeve poput čitanja ili pisanja podataka unutar baze podataka. Iz razloga što su pravila definirana van aplikacije, to jest unutar Firebase konzole, greške ne ugrožavaju podatke i podatci korisnika su uvijek zaštićeni.[31]

Za ovaj projekt su korištena prilagođena pravila pristupa koja osiguravaju da svaki korisnik može vršiti sve zahtjeve poput čitanja ili pisanja podataka. Za upravljanje pristupom uređivanja

[28] TechYourChance mrežna stranica, https://www.techyourchance.com/callback-hell-android/, [pristupljeno 13. rujna 2024.]
[29] Android Developers Documentation mrežna stranica, https://developer.android.com/kotlin/flow, [pristupljeno 13. rujna 2024.]
[30] Firebase mrežna stranica, https://firebase.google.com/support/privacy#security_information, [pristupljeno 16. rujna 2024.]
[31] Firebase mrežna stranica, https://firebase.google.com/docs/rules, [pristupljeno 16. rujna 2024.]

podataka vezanih uz korisnika, dodijeljena su pravila gdje svaki korisnik ima pravo kreirati korisnički račun unutar baze, dok pristup podatcima o korisnicima ovisi o tome ima li korisnik pristup ključnim informacijama poput korisničkog imena i lozinke čija je logika razrađena unutar aplikativnog rješenja. S druge strane postoje i pravila za upravljanje pristupom podatcima vezanih za radare gdje je korištena jednostavna logika gdje svaki korisnik može kreirati, čitati, ažurirati i brisati podatke o radarima. Sva potrebna logika za restrikciju dozvola korisnicima kod slanja zahtjeva vezanih za radare stoji u tom da je ažuriranje ili brisanje dokumenta dozvoljeno samo korisniku koji je kreirao isti tako što se za provjeru koristi unikatni identifikator korisnika i kreatora samog dokumenta. Svi ostali korisnici mogu samo čitati dokumente radara i glasati za njihovu pouzdanost.



Slika 4. Prikaz dijaloga radara[32]

Kako je definirano, svi korisnici imaju pristup kompletnom uređivanju podataka preko Cloud Firestore-a. a sva poslovna logika koja definira koji korisnici mogu modificirati koje podatke je definirana putem aplikacijskog rješenja čiji se primjer vidi na slici 4. gdje korisnik koji je kreirao radar, kada pritisne na njega, otvorit će mu se dijalog gdje ima opcije za brisanje

[32] Snimka zaslona iz praktičnog dijela rada

i uređivanje istog radara dok se korisnicima koji nisu kreirali taj radar prikazuje dijalog s opcijama za glasanje njihove pouzdanosti. Sigurnosna pravila su definirana na sljedeći način:

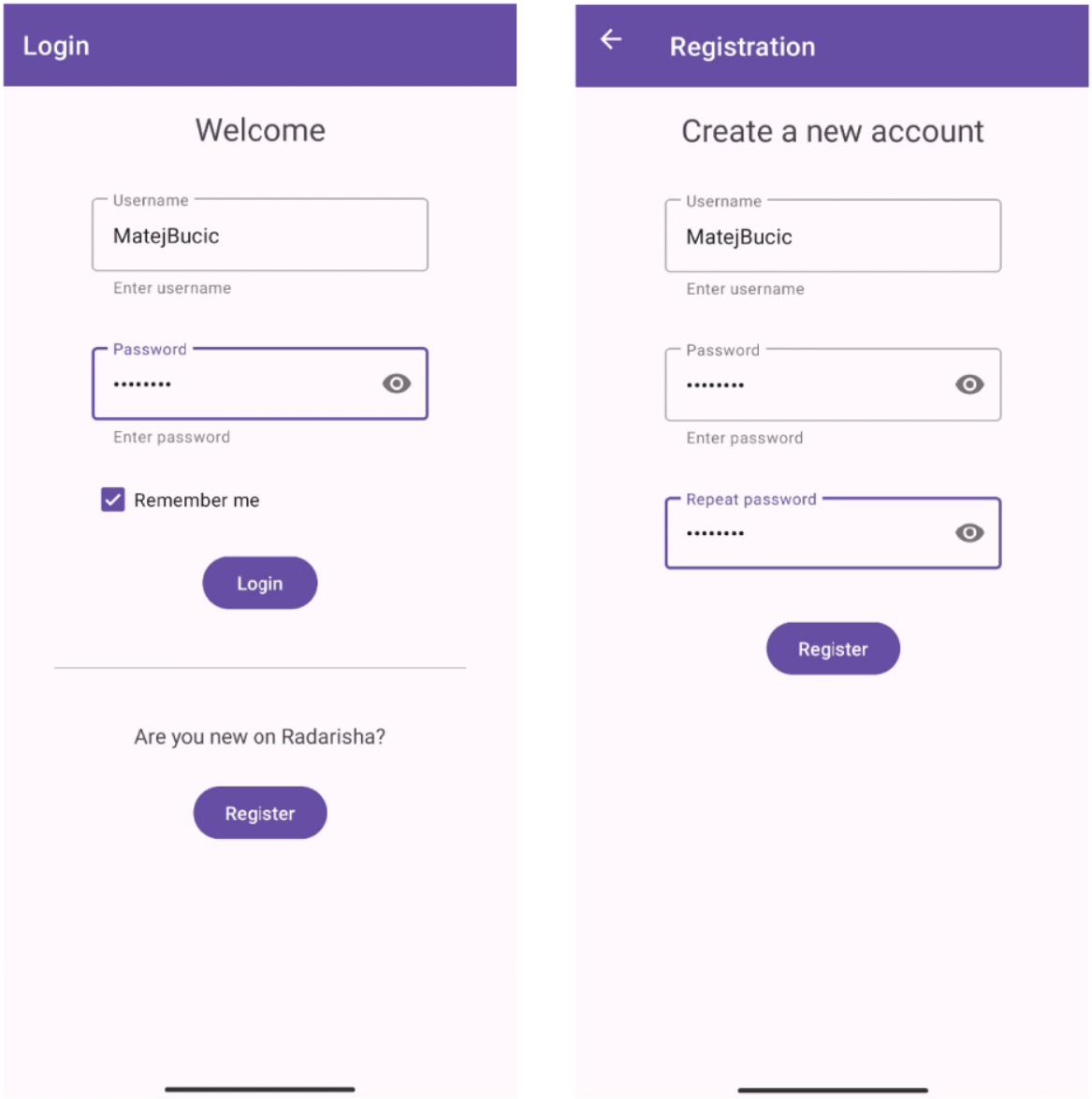
```
service cloud.firestore {
  match /databases/{database}/documents {

    // Rule for the users collection
    match /users/{uid} {
      allow read: if resource.data.keys().hasAny(['username']);
      allow create: if true;
    }

    match /radars/{uid} {

      // Allow read access to all documents in the radars collection
      allow read: if true;

      // Allow write (create) access to all users
      allow create: if true;

      // Allow delete access to all users
      allow delete: if true;

      // Allow update access only to the user who created the document
      allow update: if resource.data.creatorUid == request.resource.data.creatorUid;

      // Subcollection rules for radars
      match /{subcollection}/{uid} {
        allow read, write: if true; // Allow read/write access for authenticated users
      }
    }
  }
}
```

Slika 5. Pravila definirana za Cloud Firestore bazu podataka[33]

Iako Firebase nudi ugrađeni servis za autentifikaciju korisnika putem Google, Facebook ili e-mail prijava, unutar ovog projekta, upravljanje prijavom korisnika je ručno implementirao putem Firebase baze podataka. Premda *Firebase Authentication* servis izgleda kao dobro rješenje naspram ručne autentifikacije korisnika, radi pogodnosti koje nudi poput navedene autentifikacije putem treće strane preko Google, Facebook ili GitHub računa pa čak i putem telefonskog broja preko SMS poruke, *Firebase Authentication* servis ne pruža pristup podatcima korisnika putem konzole što je za potrebe projekta bilo potrebno. [34]

Aplikacija temelji na jednostavnosti prilikom registracije novih korisnika kojih se traži samo unos imena i lozinke kako se fokus aplikacije ne bi odmaknuo od glavne teme i kako bi se postigla maksimalna fleksibilnost i prilagodba kroz ručno upravljanje korisničkim računima.

[33] Snimka zaslona iz Firebase konzole, [snimljeno 16. rujna 2024]
[34] Firebase mrežna stranica, https://firebase.google.com/docs/auth, [pristupljeno 16. rujna 2024]



Slika 6. Prikaz autentifikacijskog grafičkog sučelja za korisnika[35]

Na slici 6. je prikazano korisničko sučelje na kojemu korisnik vrši autentifikaciju. Kada korisnik pokrene aplikaciju prvi put ili pak nije prijavljen, aplikacija će zahtijevati prijavu ili, ako je riječ o novom korisniku, registraciju korisnika prilikom koje korisnik mora unijeti svoje korisničko ime i lozinku koju također mora i ponoviti kao verifikaciju lozinke. Prilikom prijave, korisnik može označiti „*Remember me*“ polje kako bi ostao prijavljen u aplikaciji prilikom sljedećeg pokretanja.

[35] Snimka zaslona iz praktičnog dijela rada

3. Korištenje lokalne baze podataka

U svijetu razvoja mobilnih aplikacija, jedan od ključnih aspekata koji osigurava funkcionalnost aplikacije je učinkovito upravljanje podatcima. Mnoge aplikacije su predviđene za rad s većim količinama podataka, a s obzirom na to da korisnici ponekad i nemaju stalni pristup internetu, važno je osigurati lokalnu pohranu istih podataka i njihovu sinkronizaciju s udaljenim serverom. Za ovaj zadatak kod android aplikacija se najčešće koristi Room, koji je ujedno i službeni ORM alat za android aplikacije kojemu je svrha korištenja pristup bazi podataka.

Room biblioteka je ključna kako bi nadomjestila direktno korištenje i implementiranje SQLite baze podataka. Upravo on olakšava interakciju s SQLite bazom podataka i nudi jednostavniji način rada s relacijskom bazom kroz korištenje Java i Kotlin objekata. Obzirom da je Room osmišljen kao relacijska baza podataka, on omogućava da se podatci pohranjuju unutar tablica te nudi metode za upravljanje istim podatcima koristeći standardne SQL naredbe. Room nudi niz pogodnosti kao što su verifikacija SQL upita unutar kompiliranja koda i anotacija koje se koriste za definiranje klasa koje su korištene kao baze podataka, DAO klasa koje sadrže upite i entitet klasa koje predstavljaju tablice. Na taj način se umanjuje repetitivnost koda i smanjuje mogućnost stvaranja grešaka tijekom kodiranja.[36]

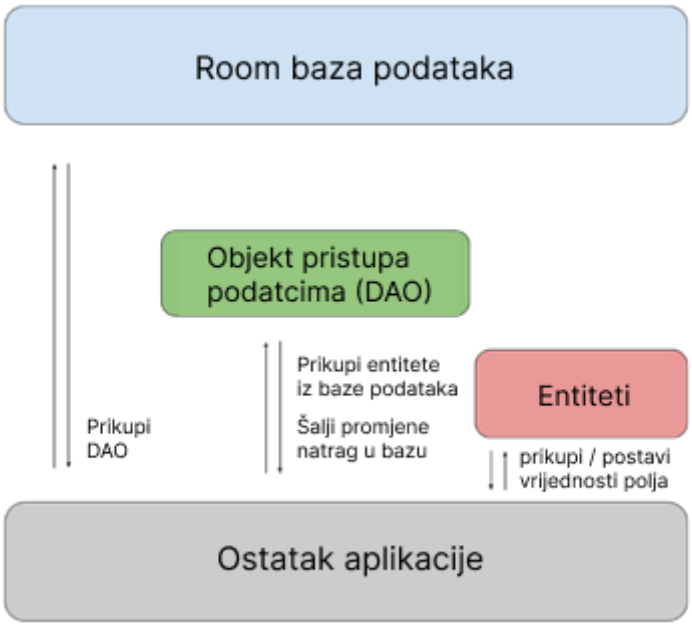
3.1. Implementacija Room baze podataka

Implementacija Room baze podataka unutar android aplikacije uključuje nekoliko ključnih koraka i komponenti koje omogućuju rad s podatcima. Cilj Room-a je pojednostavniti rad s SQLite-om, pružajući programeru jasnu i strukturiranu arhitekturu za rad s lokalnom bazom podataka.

Osnovna struktura Room-a uključuje tri glavne komponente:

- Entiteti
- DAO (*Data Access Object*)
- Baza podataka

[36] Android Developers Documentation mrežna stranica, https://developer.android.com/training/data-storage/room, [pristupljeno 16. rujna 2024.]



Slika 7. Dijagram arhitekture Room biblioteke[37]

Na slici 7. su prikazani odnosi i zaduženja između komponenti koje pruža Room biblioteka. Entiteti predstavljaju tablice unutar baze podataka. Svaka klasa koja je označena anotacijom @Entity se automatski registrira kao tablica, a svaka instanca te klase predstavlja red unutar te tablice. Svaki anotirani entitet Room baze podataka je predstavljen kao *data class* čija polja predstavljaju stupce unutar tablice. Naziv tablice entiteta se definira unutar same anotacije entiteta. Ta polja koriste svoj set anotacija kojima je moguće definirati pravila ponašanja tih polja kao što je polje koje označava primarni ključ koje se anotira s @PrimaryKey ili @ColumnInfo putem koje je moguće postaviti podatke o stupcu poput naziva samog stupca. Postavljanje naziva stupca nije nužno jer Room automatski imenuje stupce prema nazivu polja unutar klase. Glavna svrha entiteta je pretvoriti stvarne objekte poput korisnika i radara u podatke koji se mogu pohraniti u bazu podataka.[38]

DAO je komponenta Room-a koja omogućuje pristup podatcima unutar baze podataka. DAO je definiran sučeljem ili apstraktnom klasom i definira apstraktne metode koristeći SQL naredbe koje daju pristup bazi podataka. Tijekom kompiliranja aplikacije, Room automatski generira implementaciju DAO sučelja koje je definirano. Kako bi Room znao da je riječ o DAO

[37] Android Developers Documentation mrežna stranica, https://developer.android.com/static/images/training/data-storage/room_architecture.png, [pristupljeno 16. rujna 2024.]
[38] Android Developers Documentation mrežna stranica, https://developer.android.com/training/data-storage/room/defining-data, [pristupljeno 16. rujna 2024.]

komponenti, potrebno je sučelje ili apstraktnu klasu koja definira DAO anotirati s @Dao anotacijom. Metode unutar DAO komponenti se definiraju jednostavnim anotacijama poput @Insert koja definira metodu za umetanje entiteta unutar baze podataka, @Delete koja definira metodu za uklanjanje entiteta iz baze podataka te @Query koja definira proizvoljni upit na bazu podataka kao što je dohvaćanje svih entiteta iz baze podataka ili pak nekakav kompleksniji upit. Upiti su pisani kao SQL naredba unutar @Query anotacije.[39]

Klasa koja predstavlja bazu podataka predstavljena je @Database anotacijom. Ova klasa mora biti apstraktna i nasljeđivati Room-ovu *RoomDatabase* klasu. Unutar @Database anotacije se moraju definirati svi entiteti koje baza koristi unutar niza. Ta klasa također treba pružiti metode uz pomoć kojih Room daje instance DAO objekata tijekom kompiliranja aplikacije. To znači da je klasa koja definira bazu podataka most između podataka i aplikacije. Jedan od ključnih elemenata korištenih za postavljanje Room baze podataka je *Builder* koji se koristi za kreiranje instance baze podataka. Korištenjem *Builder-a* se postavlja konfiguracija baze podataka, uključujući njeno ime, verziju baze i migracijske strategije koje se definiraju u slučaju izmjena unutar strukture baze. Nužno je da se instanca baze podataka kreira kao *singleton* komponenta kako bi osigurali kreiranje samo jednog objekta klase baze podataka kroz čitav životni vijek aplikacije. Ta instanca će biti korištena kad god je potreban rad s bazom podataka. Ovaj način je izuzetno bitan jer kreiranje više instanci baze može dovesti do problema s performansama jer je svaka instanca *RoomDatabase* klase zahtjevna i koristi mnogo resursa memorije a rijetko kada je potreban pristup više od jedne instance kroz cjelokupni životni ciklus aplikacije. Unutar ove aplikacije su kreirane dvije instance baze podataka[40]. Jedna je korištena za pohranjivanje i sinkronizaciju radara s udaljenog servera dok je druga zadužena za spremanje korisnika koji je trenutno prijavljen u aplikaciju. Podatci prijavljenog korisnika su spremljeni lokalno kako bi se lakše popratili podatci korisnika koji vrši daljnju interakciju s aplikacijom kao što je kreiranje novih radara, ažuriranje i brisanje svojih radara, te ocjenjivanje pouzdanosti drugih radara. Također je unutar lokalne baze pohranjen podatak želi li korisnik da aplikacija zapamti njegovu prijavu ili ne.

[39] Android Developers Documentation mrežna stranica, https://developer.android.com/training/data-storage/room/accessing-data, [pristupljeno 16. rujna 2024.]
[40] GitHub repozitorij projektnog zadatka, https://github.com/mbuc1c/Radarisha/tree/main/data/src/main/java/com/bucic/data/database, [pristupljeno 30. rujna 2024.]

3.2. Sinkronizacija lokalnih i udaljenih podataka

Jedna od glavnih potreba kod korištenja lokalne baze podataka, poput Room-a je potreba za sinkronizacijom lokalnih podataka i podataka s udaljenog servera. Unutar ovog projekta, podatci dohvaćeni s udaljene baze podataka unutar Cloud Firestore-a se sinkroniziraju i lokalno pohranjuju korištenjem Room-a. Jedna od glavnih razlika između Room-a i Cloud Firestore-a je njihova osnovna struktura podataka. Room je dizajniran na bazi SQLite-a koji koristi relacijske baze podataka koje koriste tablice dok je Cloud Firestore dizajniran kao NoSQL baza podataka koja svoje podatke organizira korištenjem dokumenata i kolekcija. Relacijski model Room-a omogućava pohranjivanje podataka unutar strukturiranih tablica unutar kojih su definirani podatci koje će svaki entitet sadržavati i sami odnosi među tablicama. Takav pristup je izuzetno koristan kada postoji potreba za definiranjem jasne i predvidive strukture podataka kao što su naprimjer radari pohranjeni unutar aplikacije gdje je jasno da svaki radar ima podatke poput koordinata gdje je pozicioniran, tipa radara i drugih. S druge strane Cloud Firestore koristi fleksibilniji model za spremanje podataka gdje svaki dokument može sadržavati različitu strukturu podataka, a odnosi između samih dokumenata nisu strogo definirani kao kod relacijskih baza podataka. Ovaj pristup omogućuje veću fleksibilnost u radu s podatcima koji nemaju jasnu i definiranu strukturu ili koji se često mijenjaju.

Kako bi se omogućio pristup podatcima korisniku kad nije moguć pristup internetu, potrebno je napraviti sinkronizaciju između ove dvije baze podataka. Unutar ove aplikacije, Room je korišten kao pružatelj podataka u slučaju da korisnik nema pristup internetu kako bi se ostvarilo pozitivno iskustvo prilikom korištenja aplikacije. U slučaju kada korisnik ima stabilan pristup internetu, za zaprimanje podataka se koristi Cloud Firestore. Moguće je kreirati monitor[41] koji ima ulogu praćenja internet konektivnosti kada korisnik opet dobije pristup gdje će se podatci ponovno sinkronizirati i ulogu praćenja i pružanja podataka preuzme Cloud Firestore. Proces sinkronizacije sa udaljenog servera na lokalnu Room bazu podataka je izvediv na nekoliko različitih načina kako bi se održala dosljednost tih podataka između lokalne i udaljene baze. Sinkronizaciju je moguće vršiti kroz procese poput periodičnih provjera promjena koristeći *Worker-e*[42] kojima je definirano da kroz određenu jedinicu vremena pokušaju sinkronizirati podatke. Također je moguće automatski ažurirati promjene unutar podataka s obzirom da se svaka promjena već prati u stvarnom vremenu, ali taj način bi bio

[41] Sharma Anubhav, 2023., https://khush7068.medium.com/how-to-observe-internet-connectivity-in-android-modern-way-with-kotlin-flow-7868a322c806, [pristupljeno 16. rujna 2024.]
[42] Aravind Manu, 2024., https://medium.com/@mobiledev4you/work-manager-android-6ea8daad56ee, [pristupljeno 16. rujna 2024.]

skup za resurse jer postoji mogućnost velike zajednice korisnika koji istovremeno prijavljuju velik broj novih podataka. Također je moguće uvesti funkciju da korisnik ima mogućnost ručne sinkronizacije kroz aplikaciju. Unutar ove aplikacije, problem sinkronizacije je razriješen jednostavnim pozivom kada se kreira mapa na kojima su prikazani svi radari koje aplikacije treba sinkronizirati. Taj način može biti nezahvalan jer postoji mogućnost da će korisnik samo jednom zaprimiti ažurne informacije nakon što uđe u aplikaciju, no smatrano je kako taj način neće predstavljati velike probleme.

4. Ostale tehnologije korištene za razvoj aplikacije

U razvoju složenih android aplikacija, osim osnovnih principa arhitekture i upravljanja podatcima, bilo to lokalnim ili spremljenim na udaljenim serverima, korištenje drugih tehnologija ima ključnu ulogu u ostvarivanju funkcionalnosti i prilagodljivosti aplikacije kao i o poboljšanju korisničkog iskustva. Aplikacija koju razmatramo implementira važne komponente poput Google Maps API-ja za pregled karata i prikaz ključnih podataka na njima te fragmentaciju za prikaz korisničkog sučelja i navigaciju među fragmentima. Te tehnologije doprinose boljem korisničkom iskustvu i olakšavaju rad s aplikacijom. Također se koriste *coroutine* za rad s nitima što doprinosi lakšoj organizaciji poslova koji će se izvršavati u pozadini bez ugrožavanja korisničkog iskustva kao što su dohvaćanje podataka iz baze ili pozivi na udaljeni API.

4.1. Integracija Google Maps API-ja

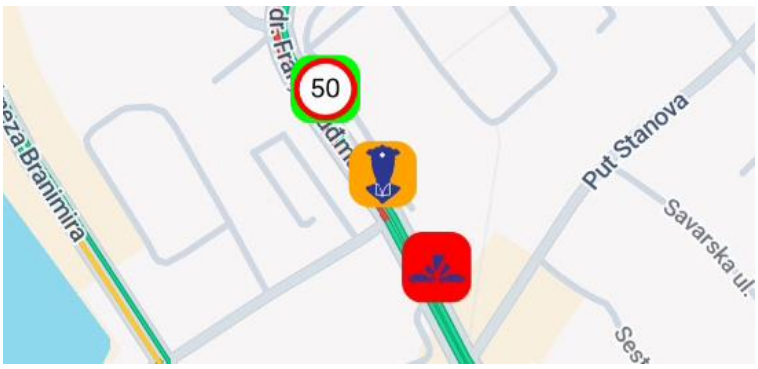
Google Maps API je jedan od ključnih alata kada je riječ o radu s kartama unutar android aplikacija. Ovaj API nudi niz tehnologija koji omogućuje prikaz interaktivnih karata i dodavanje oznaka, poligona i putanja. Uz pomoć Google Maps API-ja je moguće korisnicima pružati različite podatke unutar stvarnog vremena poput gužvi na cestama pa tako i funkcionalnosti koje su implementirane od strane aplikacije kao što su radari predstavljeni markerima na mapi sa proizvoljnom ikonom.

Pristup Google Maps API-ju zahtjeva kreiranje projekta preko Google Cloud konzole, kreiranje računa za naplatu i povezivanja istog s projektom te omogućavanje pristupa API-ima ili SDK-ovima koji su potrebni za korištenje te generiranje API ključa koji se koristi za autentifikaciju aplikacije prilikom korištenja Google-ovih servisa. Važno je postaviti

odgovarajuća ograničenja za API ključ kako bi ključ bio zaštićen od zlouporabe. Ograničenja se mogu postaviti za određenu aplikaciju unutar koje se želi koristiti definiranjem paketa unutar kojeg se API koristi i definiranjem SHA-1 certifikata te aplikacije.[43]

Da bi se prikazala karta unutar aplikacije važno je implementirati sve potrebne zavisnosti za pristup Google Maps SDK-u. Također je važno definirati API ključ unutar AndroidManifest.xml datoteke kako bi aplikacija imala pristup API-ju.[44] Za prikaz mape je zadužen *MapFragment* ili *MapView*. Prilikom implementacije, potrebno je stvoriti instancu karte koristeći metodu *getMapAsync* koja osigurava pravilnu inicijalizaciju mape prije negoli se na njoj krenu izvoditi potrebne operacije. Ova metoda je asinkrona čime se osigurava da korisničko sučelje aplikacije ostane responzivno dok se karta učitava. Važno je unutar fragmenta koji implementira mapu koristiti *OnMapReadyCallback* sučelje čija se metoda *onMapReady* poziva kada mapa bude kreirana. Unutar te metode se dodaju funkcionalnosti koje su potrebne koje definiraju ponašanje mape kao što je prikaz gužvi na prometnicama, početne koordinate koje će mapa prikazati ili pak markeri koji će biti postavljeni na mapi.[45]

Jedna od glavnih funkcionalnosti koje pruža Google Maps API je dodavanje markera na kartu. Markeri su vizualni indikatori na karti koji prikazuju određene točke interesa. U konkretnom slučaju, točke interesa prikazuju radare kamera za brzinu, prometne nesreće i policijske kontrole kao što slika 8. prikazuje. Za dodavanje markera na kartu se koristi metoda *addMarker* unutar koje se definiraju opcije koje će marker primijeniti poput geografskih koordinata markera, naslova koji marker prikazuje i ikone kojom je simboliziran marker.[46]



Slika 8. Prikaz svih radara i njihovih indikatora pouzdanosti[47]

[43] Google Maps API Documentation mrežna stranica, https://developers.google.com/maps/documentation/android-sdk/get-api-key, [pristupljeno 17. rujna 2024.]
[44] Google Maps API Documentation mrežna stranica, https://developers.google.com/maps/documentation/android-sdk/config, [pristupljeno 17. rujna 2024.]
[45] Google Maps API Documentation mrežna stranica, https://developers.google.com/maps/documentation/android-sdk/map, [pristupljeno 17. rujna 2024.]
[46] Google Maps API Documentation mrežna stranica, https://developers.google.com/maps/documentation/android-sdk/marker, [pristupljeno 17. rujna 2024.]
[47] Snimka zaslona iz praktičnog dijela rada

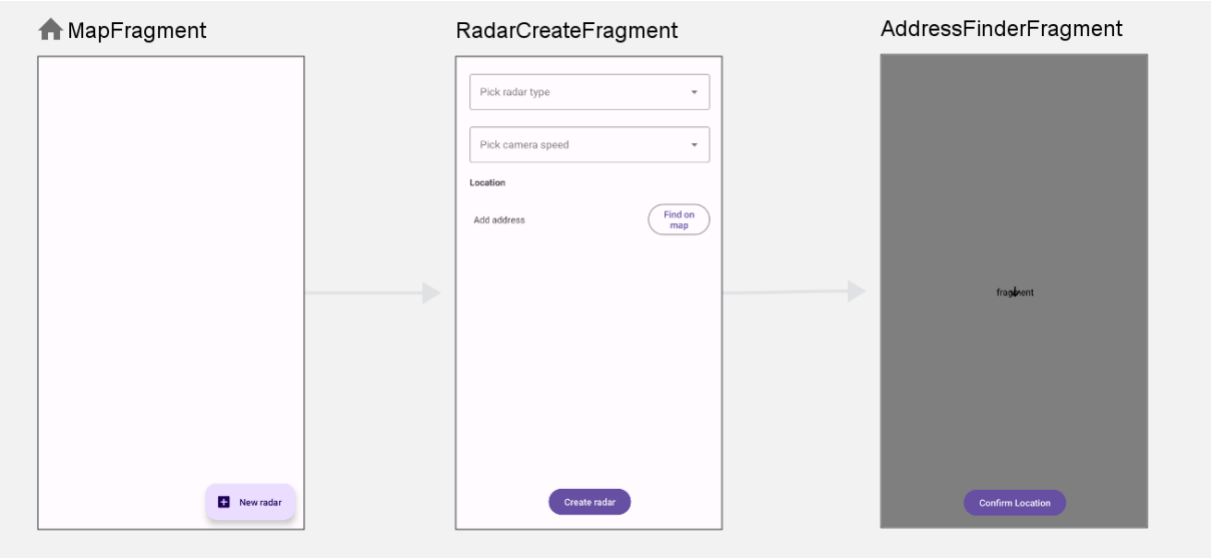
4.2. Fragmentacija i navigacija

Navigacija između različitih ekrana aplikacije je jedan od najvažnijih faktora za korisničko iskustvo tijekom korištenja aplikacije. Ekrani koji se koriste za potrebe navigacije kroz aplikaciju su predstavljeni kao fragmenti. Svaki fragment predstavlja jednu destinaciju unutar aplikacije i imaju vlastiti životni ciklus, koji je povezan s aktivnosti unutar koje je i spremljen. Fragmenti su spremljeni unutar stoga koji je vezan za navigaciju među njima što doprinosi fleksibilnosti gdje se više fragmenata može izmjenjivati putem jedne aktivnosti što na posljetku doprinosi kvalitetnijoj optimizaciji aplikacije. Upravo tako je zaobiđen problem stvaranja novih aktivnosti kroz korištenje aplikacije. Dobar primjer fragmentacije je demonstriran unutar aplikacije gdje korisnik kada uđe u glavni dio aplikacije prvo vidi mapu sa svim radarima, a kada pritisne gumb za kreiranje novog radara, navigacija će ga odvesti do fragmenta za kreiranje novog radara. Za kretanje unatrag kroz stog je zadužen „*Up button*“ koji se nalazi na vrhu korisničkog sučelja na navigacijskoj traci. Svrha njega je ukloniti najgornji fragment unutar stoga i prikazati fragment prije njega. Njegova svrha nije izaći iz aplikacije, već samo navigacija između fragmenata unatrag. [48]

Za upravljanje navigacijom unutar aplikacije se koristi Android Navigation Component. To je alat koji pojednostavljuje upravljanje složenim navigacijskim shemama unutar aplikacija. Kako bi se zaobišlo ručno postavljanje odnosa i tranzicija između fragmenata, Navigation Component nudi praktični navigacijski graf koji definira sve potrebne tranzicije između fragmenata unutar jedne datoteke. Putem Navigation Component grafa je lako definirati i animacije koje se mogu koristiti za prijelaze što doprinosi ugodnijem korisničkom iskustvu. Jednako tako mogu se definirati i podatci koji se prenose među grafovima putem *Safe Args* dodatka. Ovaj dodatak osigurava prijenos tipiziranih argumenata prilikom navigacije između fragmenata. Za tu svrhu, *Safe Args* generira objekte i *builder* klase, kako bi omogućio siguran prijenos podataka. Pomoću *Safe Args* dodatka, omogućeno je da podatci budu direktno povezani s fragmentom kojem su proslijeđeni, čime se osigurava preglednost i otpornost koda na greške.[49]

[48] Android Developers Documentation mrežna stranica, https://developer.android.com/guide/navigation/principles, [pristupljeno, 18. rujna 2024.]
[49] Android Developers Documentation mrežna stranica, https://developer.android.com/guide/navigation/use-graph/safe-args, [pristupljeno, 18. rujna 2024.]

c



Slika 9. Navigacijski graf za glavni dio aplikacije[50]

Na slici 9. je prikazan primjer grafa iz aplikacijskog rješenja gdje je definiran primjer od prije gdje se iz fragmenta koji nudi prikaz mape, pritiskom na gumb za kreiranje novog radara, otvara novi fragment za kreiranje novog radara, a iz tog fragmenta je moguće navigirati na sljedeći fragment čije je zaduženje izbor lokacije radara putem mape. Unutar navigacijskog grafa je definirana samo putanja navigacije čija će se metoda aktivirati kasnije unutar koda.

Osim navigacijskog grafa, Navigation Component tvori i *NavHostFragment* komponenta koja se koristi kao kontejner koji djeluje kao mjesto za prikazivanje definiranih fragmenata unutar navigacijskog grafa. On preuzima kontrolu nad navigacijom i omogućuje korištenje navigacijskog grafa za navigaciju između fragmenata unutar jednog dijela korisničkog sučelja. *NavHostFragment* je definiran kao fragment unutar aktivnosti na koji je povezan sam graf tako da je on zadužen za prikaz i izmjenu fragmenata tijekom korištenja aplikacije. Također, ključna komponenta Navigation Component-a je i *NavController* koji predstavlja objekt zadužen za upravljanje navigacijom unutar *NavHostFragmenta*. On preuzima akcije koje pokreću prijelaz između različitih fragmenata, pa tako vrši operacije nakon korisnikove interakcije s „*Up button-om*“. NavController koristi metode definirane unutar navigacijskog grafa za tranzicije između grafova.[51]

[50] Snimka zaslona iz praktičnog dijela rada
[51] Android Developers Documentation mrežna stranica, https://developer.android.com/guide/navigation, [pristupljeno 18. rujna 2024.]

Unutar aplikacije su definirane dvije glavne aktivnost, jedna za autentifikaciju, a druga za glavni dio aplikacije. Svaka aktivnost posjeduje svoj navigacijski graf. Ideja je odvojiti zadatke na taj način kako bi se prilikom pokretanja aplikacije odredila startna aktivnost. Na primjer ako se korisnik prijavio u aplikaciju i zatražio da aplikacija zapamti njegovu prijavu, nema potrebe prikazivati autentifikacijsko sučelje korisniku, već ga automatski proslijediti na glavni dio aplikacije.

4.3. Korištenje *Coroutine-a* za upravljanje niti

Kada je riječ o upravljanju zadatcima u pozadini aplikacije, *coroutine* u Kotlinu pružaju jednostavan i učinkovit način za rad s asinkronim operacijama. *Coroutine* su dizajnirane kako bi pojednostavnile upotrebu niti i samom primjenom njih je smanjena kompleksnost i ne čitljivost koda. One omogućuju izvršavanje zadataka u pozadini tako što ne blokiraju glavnu nit što rezultira ugodnijim korisničkim iskustvom jer se glavna nit aplikacije zadužena za prikaz grafičkog sučelja ne dira i samim tim ostaje interaktivna za korisnika. Ova funkcija je izuzetno bitna kada postoji potreba za obavljanjem dugotrajnih zadataka poput dohvaćanja podataka s udaljenog servera ili rada s bazom podataka. *Coroutine* automatski upravljaju prebacivanjem između niti na temelju definiranog dispečera.[52] Postoji nekoliko vrsta dispečera:

- *Dispatchers.Main*
- *Dispatchers.IO*
- *Dispatchers.Default*

Svaki dispečer ima svoju ulogu i razlog zašto se poziva. Uloga *Dispatchers.Main-a* je primarno da upravlja zadatcima koji se odvijaju na glavnoj niti poput zadataka zaduženih za ažuriranje korisničkog sučelja. Za razliku od *Main-a, Dispatchers.IO* se koristi kod zadataka koji uključuju jednostavne i brže zadatke koji uključuju unos/izlaz, poput rada s lokalnom bazom podataka ili vršenje mrežnih poziva na udaljene servere. *Dispatchers.Default* se također koristi za pozadinske zadatke samo što je razlika što je on zadužen za zadatke koji su CPU-intenzivni kao što je renderiranje slika. Pokretanje *coroutine-a* je moguće na dva načina. Prvi način je pozivom *launch* metode koja nema ulogu vračanja rezultata. Drugi način je korištenjem *async* metode koja može vratiti rezultat s pozivom *suspend* funkcije *await*. Obično se koristi *launch* kod poziva *coroutine-a* iz regularnih metoda, dok se *async* može koristiti za poziv

[52] Android Developers Documentation mrežna stranica, https://developer.android.com/kotlin/coroutines, [pristupljeno, 19. rujna 2024.]

coroutine unutar druge *coroutine*. Putem ovih metoda je moguće definirati dispečera koji će *coroutine-a* koristiti i zadatak koji će izvršiti. Ako se ne definira nijedan dispečer, *coroutine-a* će koristiti *Main* dispečera kao svoju zadanu vrijednost. Dispečeri se također mogu izmjenjivati unutar zadataka definiranih unutar metoda za pokretanje *coroutine-a* korištenjem *withContext* funkcije. Metode se aktiviraju korištenjem *CoroutineScope-a* putem kojeg se zadatci mogu i zaustaviti kada je to potrebno. Neke Kotlin biblioteke pružaju svoj *CoroutineScope* za određene klase koje imaju životni ciklus kao što su *ViewModel* koji ima *viewModelScope* i *Lifecycle* koji ima *lifecycleScope*.[53]

5. Rasprava

Razvijena aplikacija „*Radarisha*“ za svrhu ovog završnog rada nudi pregled kamera za brzinu, prometnih nesreća i policijskih kontrola te prikaz gužvi na prometnicama putem karte. Iako su unutar aplikacije već implementirane ključne tehnologije i standardi za razvoj aplikacijskih rješenja poput Clean arhitekture, MVVM-a, Hilt biblioteke za upravljanje zavisnostima, Room baze podataka za lokalnu pohranu te *Coroutine* i *Flow* za višenitnost i reaktivno upravljanje s podatcima, ipak postoji mogućnost za dodatna poboljšanja.

Kako bi se poboljšala sinkronizacija podataka kako bi korisnik imao što relevantniji prikaz lokalnih podataka u slučaju nestanka mrežne konekcije, moguće je koristiti *WorkManager* komponente koja omogućuje obavljanje određenih zadataka u pozadini aplikacije. Na taj način bi se moglo odrediti sinkroniziranje podataka kroz određene vremenske intervale. Kako nebi dolazilo do greške kada *WorkManager* ne može sinkronizirati podatke radi odsutnosti mrežne konekcije i kako bi uštedjeli na resursima, upravljanje *WorkManager-ovog* rada se može vršiti putem monitora koji će slati informaciju o dostupnosti ili odsutnosti mrežne konekcije.

Iako aplikacija implementira Clean i MVVM arhitekturu koji su prema standardima Google-a idealno rješenje za razvoj android aplikacija, prikaz komponenti unutar prezentacijskog sloja se služi zastarjelim *View* komponentama. Alternativno rješenje je prebacivanje na *Jetpack Compose* za izgradnju korisničkog sučelja koji za razliku od *View* komponenti ne koristi zastarjeli format putem XML datoteka već se definira putem koda što tijekom razvoja postaje mnogo preglednija solucija. Prilikom pokretanja aplikacije, za prikaz *splash* ekrana se koristi zastarjeli način definiranjem ponašanja unutar aktivnosti. Kako bi se

[53] Android Developers Documentation, https://developer.android.com/kotlin/coroutines/coroutines-adv, [pristupljeno 19. rujna 2024.]

postigla jednostavnost i praktičnost, pa tako i smanjio obujam pisanog koda, moguće je koristiti *Splash* Screen API koji omogućuje jednostavniji prikaz početnog ekrana i lakšu implementaciju animacija i željenog ponašanja.

Za autentifikaciju korisnika, korišteno je ručno kreiranje profila koji se pohranjuju na Cloud Firestore što vrlo lako može dovesti do kompleksnosti s upravljanjem korisnicima i sigurnosnih propusta. Korištenjem *Firebase Authentication-a* bi se riješio taj problem gdje korisnik ima mogućnost autentifikacije putem Google-a, Facebook-a ili putem SMS poruke.

Jedan od problema koji može nastati prilikom daljnje nadogradnje aplikacije je takozvani „*callback hell*“ gdje se višestruki *callback-ovi* isprepliću i samim time postaju kompleksni za održavanje. Taj problem se može riješiti direktnim korištenjem *Flow-a* gdje će svaku promjenu *Flow* emitirati dalje.

Na posljetku, moguće je korigirati definiranje biblioteka unutar *build.gradle* datoteke. Biblioteke unutar aplikacije su definirane na dva različita načina. prvi način je korištenjem *version catalog* datoteke unutar koje su sve verzije korištenih biblioteka spremljene, a drugi način je direktnim imenovanjem unutar *build.gradle* datoteke. Korištenje *version catalog-a* za centralizirano upravljanje verzijama biblioteka će smanjiti nekonzistentnost u imenovanju korištenih biblioteka. Također bi se trebala obratiti pozornost na optimizaciju resursa kroz izvlačenje *string-ova* i dimenzija u odgovarajuće XML datoteke čime se lakše vrše promjene na većoj razini aplikacije i osigurava lakša čitljivost koda.

Aplikacija „*Radarisha*“ nudi solidnu osnovu za daljnji razvoj gdje implementira ključne tehnologije. Unatoč tomu postoji značajan prostor za poboljšanja poput automatizirane sinkronizacije podataka, boljeg upravljanja podatcima dohvaćenima s udaljenog servera, te modernizacije korisničkog sučelja. Integracijom ovih poboljšanja bi se poboljšalo iskustvo korisnika, poboljšala bi se sigurnost i optimizacija performansi same aplikacije.

6. Zaključak

Razvijanje android mobilne aplikacije za uživo ažuriranje stanja u prometu oslanjalo se na nekoliko ključnih tehnologija kao što su Firebase-ov Cloud Firestore za pohranu podataka na udaljenoj bazi podataka, Room baza podataka za lokalno spremanje podataka te Google Maps API za prikazivanje informacija na karti. Korištenje Clean arhitekture i MVVM obrasca omogućilo je jasnu podjelu odgovornosti između različitih slojeva aplikacije čime je postignuta visoka razina skalabilnosti, održivosti i pokrivenosti testovima. Također korištenjem navigacije i fragmentacije se olakšao rad s različitim sučeljima dok je korištenje *coroutine-a* optimiziralo rad s asinkronim zadatcima bez blokiranja glavne niti aplikacije.

Iako je aplikacija funkcionalna i efikasna, postoji prostor za daljnja poboljšanja. Na primjer, mogla bi se poboljšati sigurnost implementacijom naprednijih pravila pristupa podatcima u Cloud Firestore-u ili integracijom *Firebase Authentication-a* za bolju kontrolu pristupa korisnika. Dodatno, u budućim iteracijama je moguće koristiti *Flow* umjesto callback-ova za praćenje promjena podataka u stvarnom vremenu, čime bi se pojednostavnilo upravljanje tokovima podataka. Daljnji razvoj aplikacije mogao bi uključivati bolju optimizaciju rada s mrežom uz pomoć monitora kao i implementaciju naprednijih značajki poput integracije vanjskih servisa za obogaćivanje podataka. Sve ove promjene bi dodatno unaprijedile korisničko iskustvo i performanse aplikacije.

Android mobile application for live traffic updates

Summary

This final thesis covers the development of an Android application for real-time traffic monitoring, focusing on the use of modern technologies such as Firebase's Cloud Firestore, a NoSQL remote database solution, the Room local database, and Google Maps API. The application allows users to view current road conditions, including traffic jams, speed cameras, traffic accidents, and police patrols. The thesis is divided into several key sections. The first part of the thesis addresses the basic architectural guidelines used within the application, with an emphasis on Clean Architecture and the MVVM pattern, which facilitates a clear separation of responsibilities between different application layers. The thesis also discusses the use of Hilt as the chosen tool for implementing dependency injection. The second part of the thesis provides a detailed explanation of the selection of the remote server, where Firebase's Cloud Firestore was chosen due to its advantages in real-time data tracking and ease of management. Additionally, the use of the Room database as a solution for local data storage is clarified, highlighting its role in synchronizing data retrieved from the remote server. The final part of the thesis explains the integration of Google Maps API, outlining features such as adding markers to the map. Moreover, the thesis explains the use of fragmentation and navigation within the application, leveraging the Android Navigation Component. Lastly, it covers the use of Kotlin coroutines for thread management, focusing on optimizing the application when working with databases and network calls. Readers can follow the development process and technical details of the application via the GitHub repository at the following link: https://github.com/mbuc1c/Radarisha.

Keywords: Android, Clean Architecture, MVVM, Firebase, Cloud Firestore, Google Maps API, Room, Hilt, Kotlin coroutines.

Popis literature

[1] Android Developers, *Developer guides*, 2024., Dohvaćeno iz: https://developer.android.com/guide.

[2] Android Developers, *Android's Kotlin-first approach*, 2024. Dohvaćeno iz: https://developer.android.com/kotlin/first.

[3] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 2017.

[4] GeeksforGeeks, *MVVM (Model View ViewModel) Architecture Pattern in Android*, 2022., Dohvaćeno iz: https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/.

[5] M. Vivo, *Scoping in Android and Hilt*, 2020., Dohvaćeno iz: https://medium.com/androiddevelopers/scoping-in-android-and-hilt-c2e5222317c0.

[6] Firebase, *Developer documentation for Firebase*, 2024., Dohvaćeno iz: https://firebase.google.com/docs.

[7] Vasiliy, *Callback Hell in Android*, 2020., Dohvaćeno iz: https://www.techyourchance.com/callback-hell-android/.

[8] A. Sharma, *How to Observe Internet Connectivity in Android — Modern Way with Kotlin Flow*, 2023., Dohvaćeno iz: https://khush7068.medium.com/how-to-observe-internet-connectivity-in-android-modern-way-with-kotlin-flow-7868a322c806.

[9] M. Aravind, *Work Manager – Android*, 2024., Dohvaćeno iz: https://medium.com/@mobiledev4you/work-manager-android-6ea8daad56ee.

[10] Google for Developers, *Google Maps Platform Documentation*, 2024., Dohvaćeno iz: https://developers.google.com/maps/documentation.

[11] T. Repčík, *Dependency injection with Hilt in Android development*, 2023., Dohvaćeno iz: https://tomas-repcik.medium.com/dependency-injection-with-hilt-in-android-development-e23fc636d65c.