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

Sveučilišni studij

MOBILNA APLIKACIJA ZA ORGANIZACIJU I UPRAVLJANJE DOGAĐAJIMA

Diplomski rad

Davor Štajcer

Osijek, 2022.

Sadržaj

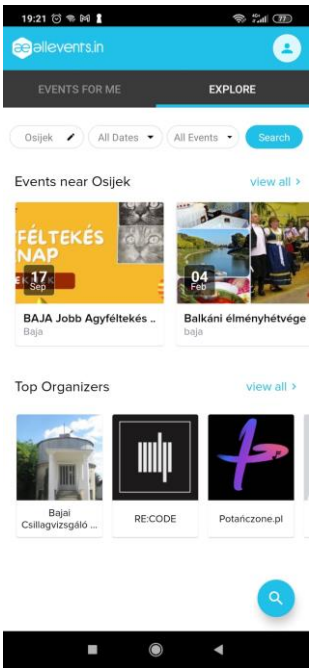
1. UVOD

Mobilne aplikacije predstavljaju jedan od glavnih aspekta zbližavanja čovjeka s mobilnim uređajem. Čovjek ih svakodnevno koristi i predstavljaju glavni način komunikacije između ljudi (*WhatsApp, Instagram, Facebook,..*). Također, čovjek koristi aplikacije kako bi udovoljiobilo kojim od svojih pet osjetila. Iz tog razloga popularnije aplikacije, kao što su društvene mreže, sadrže puno slika, video uradaka i kratkih tekstova. Uz to, imaju jako dobre performanse te sadrže puno funkcionalnosti koje trebaju obavljati svoje poslove organizirano kako bi se korisniku što čišće elementi ekrana prikazali na ekran. Načini na koje se organiziraju ovi poslovi te načini na koji se oni razdjeljuju u manje cjeline se zove arhitektura i ona se u različitim oblicima primjenjuje na sve vrste aplikacija, ne samo na društvene mreže. U ovome radu će se proći kroz arhitekturu kompleksne aplikacije na razini većih komponenti i na razini pojedinih manjih komponenti. Također, objasnit će se glavni dijelovi rada aplikacije, njena upotreba, formiranje komponenata arhitekture u odnosu na njihove namjene i alternativni načini implementacije arhitekture. Ideja aplikacije jest pružiti korisnicima platformu gdje mogu pregledavati događaje napravljene od strane drugih ljudi u svojem gradu ili selu. Cilj je povezati korisnike unutar događaja prvobitno preko grupnog razgovora, a kasnije sastankom uživo. Korisnicima je pregled događaja bolje dočaran *Google Maps* uslugom i drugim sučeljima koje *Google* organizacija pruža poput sučelja za dohvaćanje korisnikove trenutne lokacije te mapiranja te lokacije u informaciju o gradu ili selu u kojem se korisnik trenutno nalazi. Svaki korisnik posjeduje profil s određenim osobnim informacijama i time se jednoznačno određuje svaki korisnik. Korisnici primaju obavijesti čime se dodatno poboljšava korisnikovo iskustvo u aplikaciji. Aplikacija u ovome radu će biti izrađena koristeći *Flutter* radni okvir pisan u *Dart* programskom jeziku. Razvojno okruženje koje će se koristiti u projektu je *Visual Studio Code*. Aplikacija će biti integrirana s *Firebase* platformom koja će služiti kao baza podataka i sredstvo za registraciju i prijavu korisnika. Većina podataka potrebnih za prikaz na ekranu će se povlačiti pomoću jedne od *Firebase* usluga.

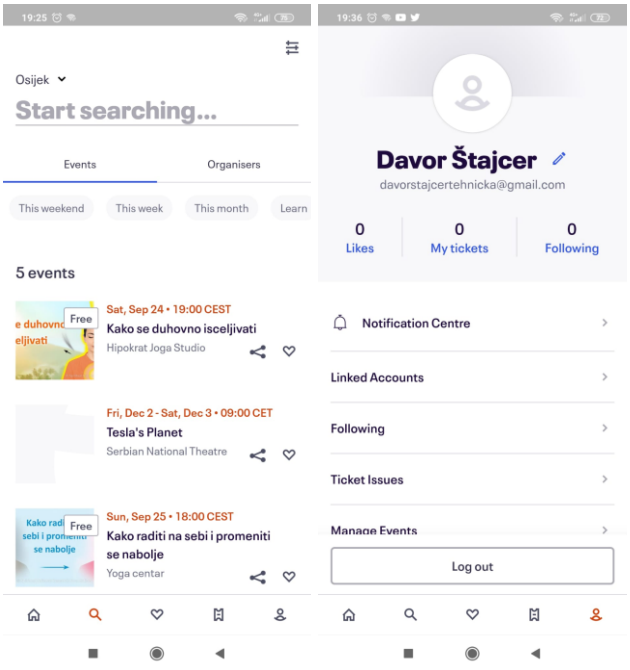
Prvobitno se u radu prolazi kroz postojeća rješenja istog problema, navode se njihove funkcionalnosti te razlike njih i rješenja u ovome radu. Zatim se u trećem poglavlju prolazi kroz korištene tehnologije i alate koji su korišteni pri izradi aplikacije. U četvrtom poglavlju se prolazi kroz programsko rješenje problema. Opisuju se stvari korištene od *Flutter* radnog okvira, arhitektura aplikacije i baze podataka te se povezuje izgled pojedinog ekrana s dijelovima arhitekture. Na kraju se u zaključku ukratko opisuje i komentira rad te se navode moguće nadogradnje na aplikaciju.

2. POSTOJEĆA RJEŠENJA

Postoje slične aplikacije koje pružaju dio postojećih funkcionalnosti aplikacije koja se analizira u radu, ali pružaju i neke funkcionalnosti koje ova aplikacija nema. Primjeri sličnih aplikacija su aplikacija *All Events in City* [1], *Eventbrite* [2], *Fever* [3], *Meetup* [4] te *Agorify* [5]. *All Events in City* je aplikacija koja pruža organizatorima platformu za objavljivanje događaja i korisnicima platformu za pretragu događaja po gradovima (Slika 2.1.). Svaki korisnik ima svoj profil. Uglavnom aplikacija ne pruža nekakve drugačije funkcionalnosti od aplikacije koja se analizira u radu. *Eventbrite* je aplikacija koja proširuje funkcionalnosti *All Events in City* aplikacije tako što uvodi obavijesti u aplikaciju, mogućnost kupnje karata preko aplikacije, mogućnost praćenja organizatora, te bolje korisničko iskustvo s detaljnijim informacijama na ekranu (Slika 2.2.). Ideja plaćenih događaja je dobra jer uvodi u igru sponzore i profit aplikacije jer je posrednik. Aplikacija u ovome radu se više fokusira na formiranje događaja dostupnih svima i komunikaciju ljudi unutar tog događaja.

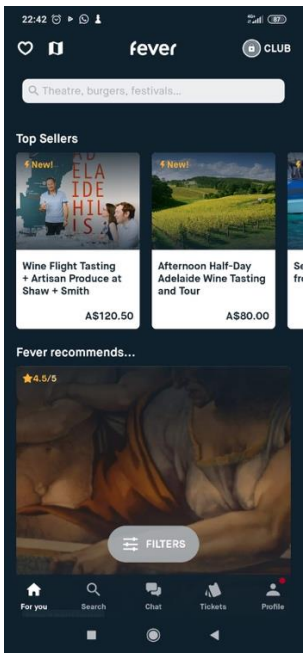


Slika 2.1 Sučelje *All Events in City* aplikacije

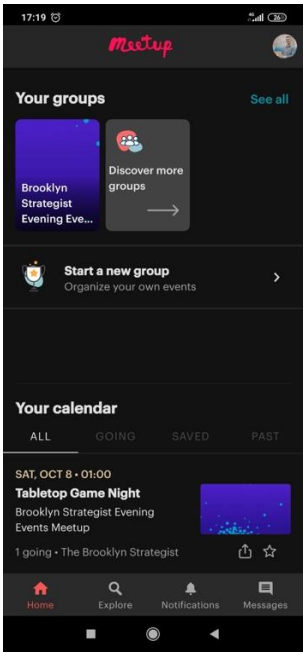


Slika 2.2. Sučelje *Eventbrite* aplikacije

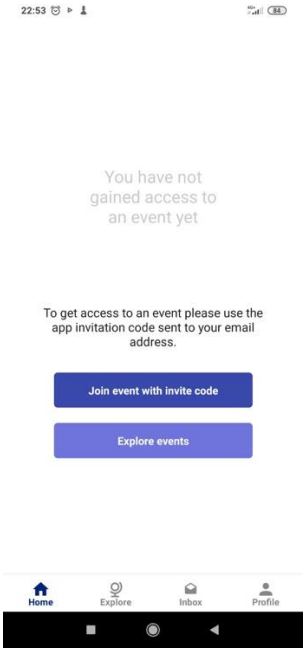
Aplikacije *Fever* (Slika 2.3.), *Meetup* (Slika 2.4.), te *Agorgy* (Slik 2.5.) pružaju korisniku razmjenjivnje poruka, ali samo sa jednom osobom, za razliku od aplikacije u radu koja pruža korisniku mogućnost komunikacije sa svim osobama unutar događaja. Aplikacije pružaju pregled događaja u svim gradovima, ne samo u gradu u kojem se korisnik trenutno nalazi.



Slika 2.3. Sućelje *Fever* aplikacije.



Slika 2.4. Sučelje *Meetup* aplikacije



Slika 2.5. Sučelje *Agorify* aplikacije

3. KORIŠTENE TEHNOLOGIJE I ALATI

U ovome poglavlju će se opisati što je *Flutter* radni okvir, njegova arhitektura te način na koji se stvaraju aplikacije pomoću njega. Opisati će se *Dart* programski jezik pomoću kojega je napisana aplikacije, te će se ukratko opisati *Firebase* platforma.

3.1. Flutter radni okvir

Flutter je razvojno okruženje razvijeno od strane *Google* firme. *Google* je stavio programski kod od *Flutter* radnog okvira javno na internet kako bi svi mogli vidjeti dijelove od kojega je on strukturiran i kako bi mogli možda i nešto novo napraviti te tako nadograditi funkcionalnosti koje *Flutter* radni okvir pruža developerima (engl. *Open Source*). Pušten je javnosti 2017. godine, ali je verzija 1.0.0 izašla u prosincu 2018. godine [6]. Dok se platforme za razvijanje nativnih mobilnih aplikacija kao što je Android i Xcode fokusiraju samo na aplikacije za Android uređaje te samo na aplikacije za iOS uređaje, Flutter radni okvir daje developerima mogućnost razvijanja mobilnih aplikacija za mobitele koji podržavaju i Android operacijske sustave i iOS operacijske sustave. Nadalje, Flutter radni okvir ne staje samo na mobilnim aplikacijama već nudi developerima i mogućnost razvoja web i računalnih aplikacija, čiji je izgled i čije su funkcionalnosti razvijene uzimajući u obzir funkcionalnosti koje računalo pruža.

Flutter SDK (engl. *Standard Development Kit*) pruža stroj za prikazivanje dvodimenzionalnih informacija korisniku na ekranu, koji ima dobru potporu za prikazivanje i skaliranje teksta na mobilnim ekranima. Nadalje, *Flutter* radni okvir pruža velik broj predefinirani komponenti koje predstavljaju temelj razvoja mobilnih aplikacija s *Flutter* radnim okvirom – objekti *Widget* klase, sučelja za testiranje funkcionalnih jedinica, sučelja za integracijsko testiranje i testiranje objekata *Widget* klase, *Dart DevTools* alat koji služi za prikazivanje karakteristika aplikacije pri radu, otklanjanje pogrešaka, pregled informacija korisnih pri razvoju aplikacija za developere te CLI (engl. *Command Line Interface*) naredbe za kreiranje, izradu, testiranje te *kompilaciju* aplikacija [7].

Flutter razvojno okruženje koristi Dart VM (engl. *Virtual Machine*) za procesiranje aplikacija razvijenih na Linux, Windows i MacOS operacijskim sustavima. Dart VM koristi JIT (engl. *Just In Time*) *kompilaciju* koda koja pruža uštedu vremena pri razvoju aplikacija kao što je mogućnost brzog osvježavanja aplikacije (engl. *Hot Reload*). To funkcionira tako da JIT, dok developer razvija mobilnu aplikaciju i mijenja postojeći programski kod dok već postoji instanca aplikacije koja je pokrenuta, ubacuje novonastali ili promijenjeni kod i aplikaciju koja

je pokrenuta. U većini slučajeva će promjene biti vidljive s primjenom operacije brzog osvježavanja.

Flutter radni okvir je građen na principu kompozicije, a ne nasljeđivanja. Kompozicija nalaže kako se jedna klasa A sastoji od drugih klasa, to jest njene *dependencije* su klase B i C kao što se vidi na lijevom dijelu Programskog koda 3.1. Kod nasljeđivanja klasa C mora implementirati sve funkcionalnosti klasa A i B, što se vidi na desnome dijelu Programskog koda 1. Tu je glavna razlika između nasljeđivanja i kompozicije, kod kompozicije A klasa može interno birati koje funkcionalnosti klasa B i C želi iskoristiti, dok je s nasljeđivanjem ona forsirana implementirati sva ponašanja koja klase A i B zahtijevaju. *Flutter* radni okvir preferira kompoziciju i sama gradnja aplikacije je zasnivana na kompoziciji, što se može vidjeti iz primjera na Programskom kodu 3.1. Klase se prosljeđuju drugim klasama i tako se opisuje ono što se prikazuje na ekranu. Većina tih klasa dijele iste funkcionalnosti tako što nasljeđuju klasu *Widget*. Skoro sve u *Flutter* radnom okviru su objekti *Widget* klase. *Widget* objekti se prosljeđuju drugim *Widget* objektima i tako, kompozicijom, dobivamo i programski kod koji je napisan na čitljiv način i lagano se prepoznaju dijelovi ekrana. Na Programskom kodu 3.2. se lagano prepoznaje kako je nekakav tekst centriran unutar *Scaffold* objekta.

```
class A {
  final int smth;
  const A(this.smth);
}

class B {
  final int smthElse;
  const B(this.smthElse);
}

class C {
  final A a;
  final B b;
  C(this.a,this.b);
}
```

```
abstract class A {
  abstract int smth;
}

abstract class B {
  abstract int smthElse;
}

class C implements A, B{

    @override
  int smth;

  @override
  int smthElse;

  C(this.smth,this.smthElse);
}
```

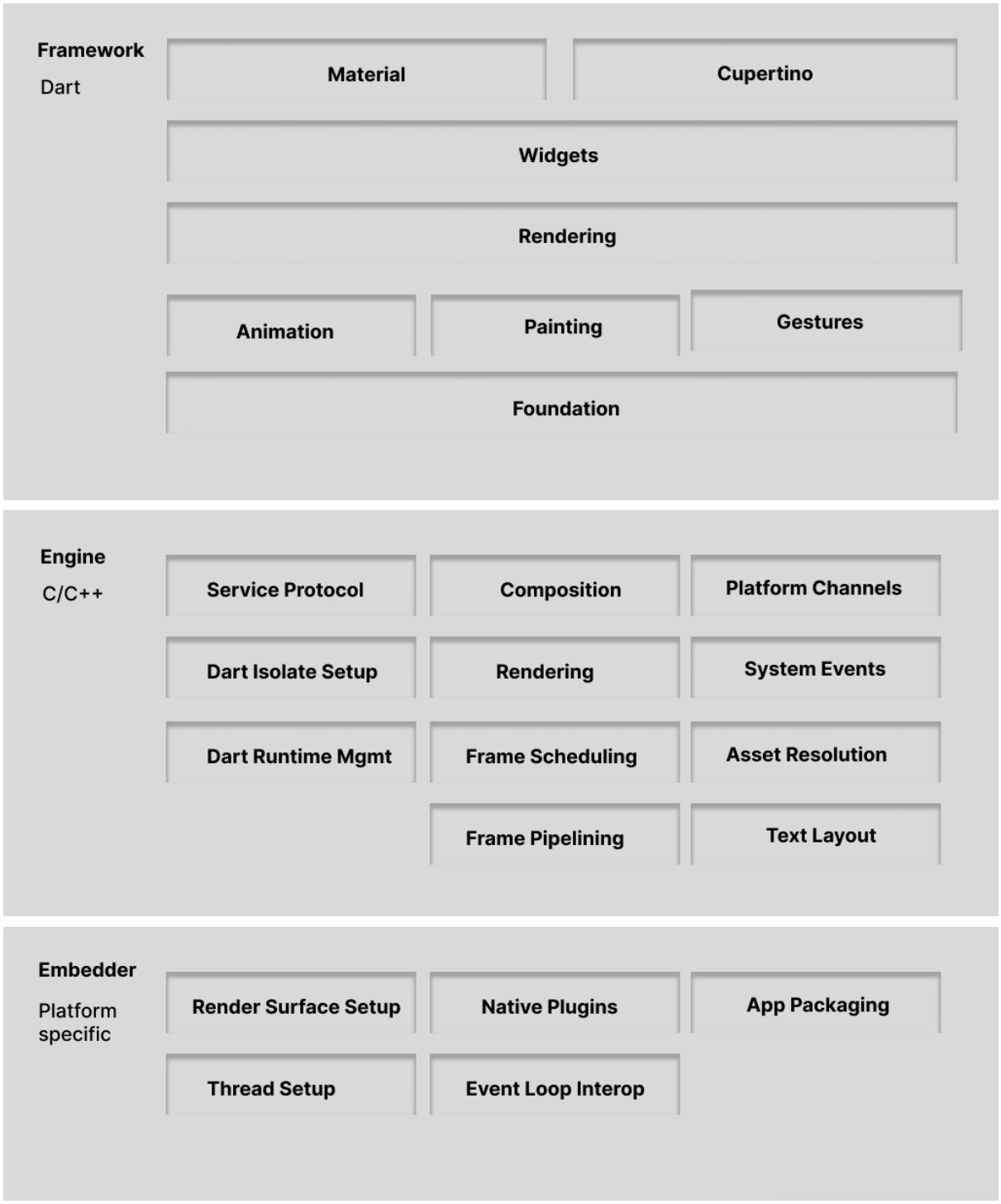
Programski kod 3.1. Kompozicija i nasljeđivanje

```
class ExampleApp extends StatelessWidget {
const ExampleApp({Key? key}) : super(key: key);

@override
Widget b Korisničko sučelje ld(BuildContext context)
{
return const MaterialApp(
home: Scaffold(
body: Center(
child: Text('Some text'),
),
),
);
}
```

Programski kod 3.2. Kompozicija u *Flutter* razvojnom okruženju

3.1.1. Arhitektura

Arhitektura *Flutter* radnog okvira je formirana u slojevima koji se mogu proširivati. Svaki sloj ima ograničen pristup sloju ispod njega, tj. slojevi ispod pružaju sučelje koje opisuje njegove funkcionalnosti sloju gore (Slika 3.1.). Također svaki dio *Flutter* razvojnog okruženja je dizajniran kako bi bio opcionalan i može se zamijeniti. Krećući od donjeg *Embedder* sloja, aplikacija napravljena u *Flutter* radnom okviru izgleda operacijskom sustavu koji ju pokreće kao svaka druga aplikacija dizajnirana u prikladnim razvojnim okruženjima. *Embedder* sloj komunicira direktno s operacijskom sustavom i traži dopuštenja za pristup servisima kao što je ekran. Ako je operacijski sustav Android onda je ovaj sloj napisan u Java i C++ programskom jeziku, za MacOS i iOS to je Objective-C/C++ te C++ za Windows i Linux. Srednji sloj je *Flutter Engine*, koji je srž *Flutter* radnog okvira te je najčešće napisan u C++ programskom jeziku. Srednji sloj pruža implementacije glavnih sučelja u *Flutter* radnom okviru kao što je grafika pomoću *Skia* biblioteke za dvodimenzionalnu grafiku, tekst, datoteke, arhitekturu biblioteka i dr. Developeri pristupaju srednjem sloju preko *Flutter* radnog okvira koji je napisan u *Dart* programskom jeziku. Ovaj sloj sadrži podslojeve koji pružaju klase koje se koriste u gotovo svakoj aplikaciji razvijenoj s *Flutter* radnim okvirom. *Rendering* sloj pruža apstrakcije za prikazivanje stvari na ekranu tako što kreira stablo *Render* objekata. *Render* objekti se mogu dinamički manipulirati u stablu te svaka promjena stabla automatski osvježava ekran i prikazuje promjene. *Widgets* sloj predstavlja kompoziciju apstrakcija. Sve klase su apstraktne i njihovi međuodnosi su opisani kompozicijom. Svaki *Widget* objekt u *Widgets* sloju ima *Render* objekt iz *Rendering* sloja koji je napravljen po njemu [6] [8].



Slika 3.1. Arhitekturalni slojevi *Fluttera*

3.1.2. Widget objekti

Widget objekti u *Flutter* radom okviru predstavljaju osnovnu jedinicu građenja kompozicije, tj. osnovnu jedinicu strukturiranja izgleda aplikacije. Pomoću kompozicije, kao što je rečeno u prethodnome poglavlju, formiraju hijerarhiju *Widget* objekata. *Widget* koji je na vrhu te hijerarhije predstavlja početnu točku aplikacije i zove se korijenski *Widget* objekt. Na Programskom kodu 3.3. je korijenski *Widget* MyApp. Korijenski *Widget* objekt se

prosljeđuje kao parametar metodi *runApp()* koja se poziva unutar funkcije *main()*, koja je prva funkcija koja se pokreće pri pokretanju aplikacije. Sve stvari koje se trebaju inicijalizirati prije početka rada aplikacije se mogu napraviti unutar *main()* funkcije. Nadalje, na Programskom kodu 3.3. se može vidjeti kako *Widget* objekt sadrži metodu *build()*. Metoda *build()* predstavlja sadržaj koji *Widget* objekt prikazuje i svaki *Widget* objekt mora implementirati tu metodu koja vraća objekt tipa *Widget*, tj. vraća stablo *Widget* objekata. S kompozicijom i implementacijom *build()* metode se postigao način rekreiranja određenog dijela ekrana, tj. samo dijela *Widget* stabla koji se promijenio tako da se ponovo poziva *build()* metoda od promijenjenog *Widget* objekta. Iz ovog razloga je bitno da *build()* metoda ne sadrži nikakvu logiku tešku za procesiranje već se ona treba obraditi na asinkron način.

S Programskog koda 3.3. se može primijetiti da *Widget* klasa nasljeđuje klasu *StatelessWidget*. *Flutter* radni okvir pruža dvije vrste *Widget* objekata: *StatelessWidget* i *StatefullWidget*. *StatelessWidget* objekti predstavljaju *Widget* objekte koji se ne mijenjaju s obzirom na nekakvo svoje stanje, dok se *StatefullWidget* objekti mijenjaju s obzirom na stanje koje je promjenjivo. Na primjer, ako se prikazuje nekakav brojač na ekranu, onda *Widget* objekt koji je odgovoran za prikazivanje teksta na ekran kontrolira stanje koje sadrži trenutni broj brojača i kada se on promjeni *Widget* stablo će se rekreirati.

```
void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
 const MyApp({super.key});

 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   home: Scaffold(
    appBar: AppBar(
     title: const Text('My Home Page'),
    ),
    body: Center(
     child: Builder(
      builder: (context) {
       return Column(
        children: [
         const Text('Hello World'),
```

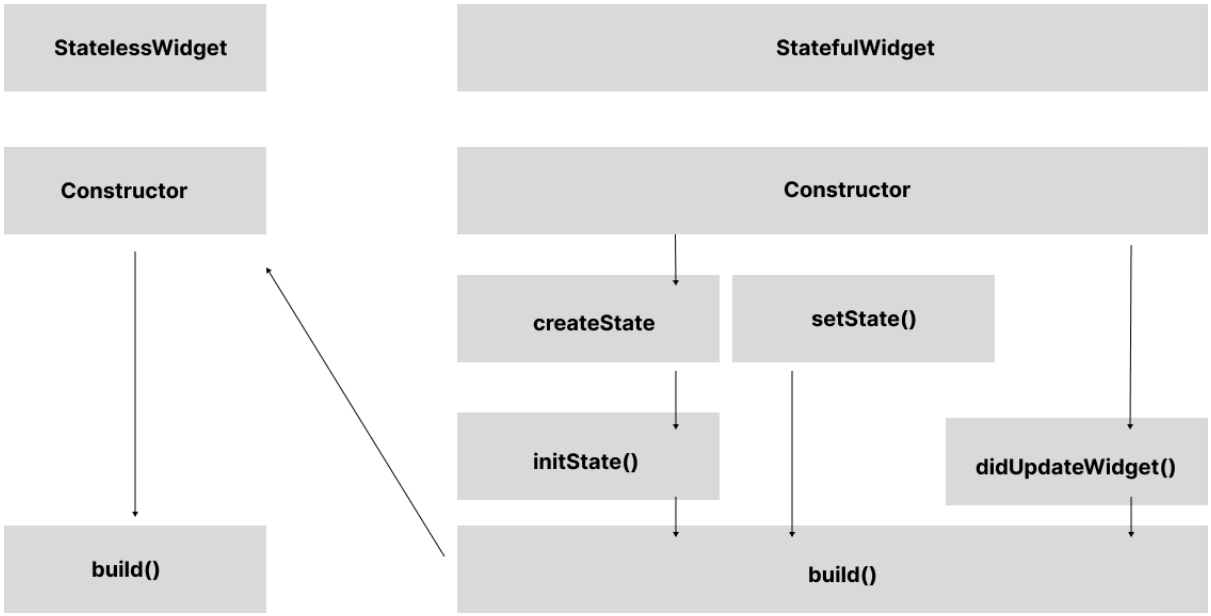
Programski kod 3.3. Ulazna točka aplikacije

StatefullWidget objekti sadrže metode koje se okidaju u različitim dijelovima životnog vijeka objekta (engl. *Lifecycle Methods*). To su metode *initState(), didChangeDependecies()* i *dispose()* kao što se vidi na Programskom kodu 3.4. Developer u tim metodama može iskoristiti

trenutke za obradu dijela koda: nakon što *Flutter* razvojno okruženje kreira *StatefulWidget* objekt – *initState()*, kada se stanje od *Widget* objekta uklanja – *onDispose()*,kada se neki od *dependencija Widget* objekta promjene i prvi puta odmah poslije *initState()* metode – *didChangeDependencies()*. Također, postoji metoda *setState()* kojoj developer predaje metodu gdje mijenja parametre *State* objekta pripadajućeg *StatefullWidget* objekta. Tijek poziva metoda se može vidjeti na Slici 3.2.

```
class StatefullExample extends StatefulWidget {
  const StatefullExample({Key? key}) : super(key: key);

  @override
  State<StatefullExample> createState() => _StatefullExampleState();
}

class _StatefullExampleState extends State<StatefullExample> {

@override
  void initState() {
    // TODO: implement initState
    super.initState();
  }

  @override
  void didChangeDependencies() {
    // TODO: implement didChangeDependencies
    super.didChangeDependencies();
  }

  @override
  void dispose() {
    // TODO: implement dispose
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return const Center(
      child: Text("Statefull example"),
    );
  }
}
```

Programski kod 3.4. *StatefullWidget*



Slika 3.2. Metode životnog vijeka *StatefullWidget* objekta

3.2. Dart programski jezik

Dart je programski jezik za razvijanje aplikacija na bilo kojoj platformi. *Flutter* razvojno okruženje koristi *Dart* programski jezik jer je optimiziran za funkciju brzog osvježavanja te web, mobilne i računalne platforme. Programski jezik pruža korištenje tipova podataka u programskome kodu na siguran način (engl. *Type Safety*), što znači da vrijednost varijabli ne može biti *null* osim ako developer izričito ne naznači da im vrijednost može biti *null*. Kompajler s ovim informacijama može prepoznati dijelove koda u kojem neka varijabla može biti *null* a ne bi smjela biti. Nadalje, s time se izbacuje velik broj *null* provjera varijabli tako što se uvode novi operatori '!', '?', '??' te *'bool* ? *firstCode* : *secondCode*' (Programski kod 3.5.) [9][10].

```
int nullSafetyExample(int? num, String? word) {
  int lnght = word?.length ?? 0;
  return num == null ? lnght : lnght + num;
}
```

Programski kod 3.5. Sigurnost protiv *null* vrijednosti

3.3. Firebase platforma

Firebase je platforma za razvijanje aplikacija koja pruža developerima usluge kao što su registracija i prijava, baza podataka koja informira korisnike o promjeni podataka u stvarnome vremenu, spremanje datoteka u oblaku, nadzor aplikacije, funkcije koje služe za okidanje određenog dijela koda pri izvršavanju CRUD operacija nad *Firestore* bazi podataka, *Remote Config* koji služi za postavljanje podataka bitnih za inicijalizaciju aplikacije i za

postavljanje nekakvih funkcionalnosti aplikacije bez forsiranja korisnika da ažurira svoju aplikaciju. Također dopušta izbacivanje aplikacija na *Firebase App Distribution* platformu gdje testeri mogu testirati različite verzije aplikacija prije nego li se aplikacija pusti u produkciju. Kod razvoja većine aplikacija postoji problem prijave i registracije korisnika. Developeri koji se upuste sami u taj pothvat često naiđu na prepreke koje zagušuju tok razvoja softvera i produžuje vremenski rok potreban za dostavu aplikacije na tržište. *Firebase* pruža jednostavno sučelje gdje developeri sa samo par linija koda i integracijom *Firebase* platforme u svoju aplikaciju mogu iskoristiti *Firebase* način registracije i prijave koji se pokazao vrlo učinkovitim kroz ovih desetak godina postojanja *Firebase* platforme [11]. U izradi ove aplikacije je korišten *Firebase Authentication, Firestore Database, Storage* te *Cloud Functions*.

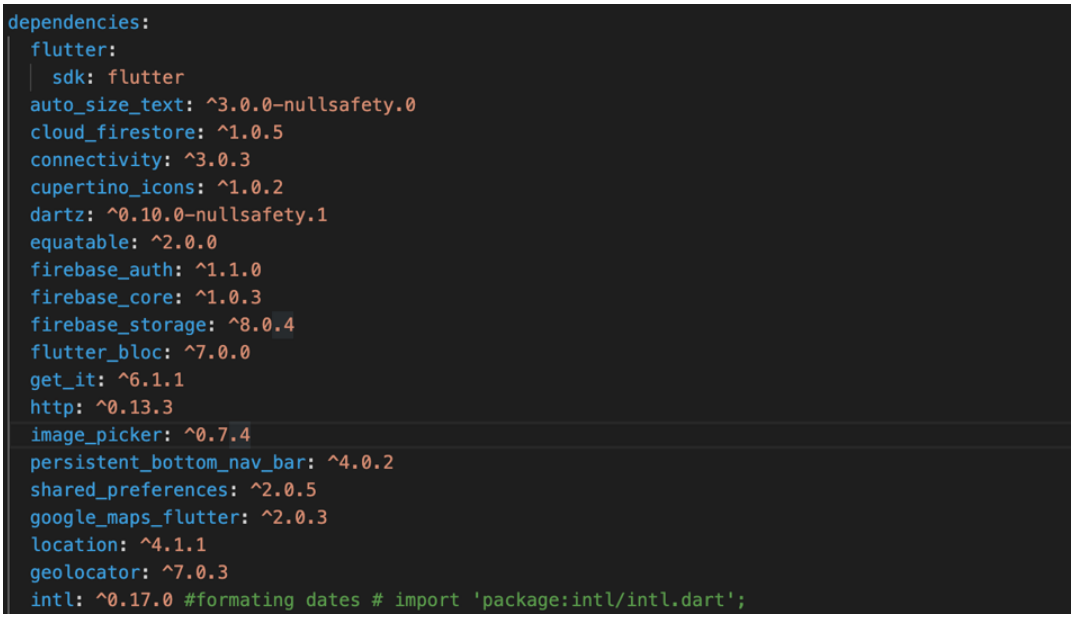
4. PROGRAMSKO RJEŠENJE

Aplikacija u ovome radu se razdvaja na više funkcionalnosti: registracija i prijava u aplikaciju, kreiranje događaja, pregled svih događaja oko lokacije u kojoj se korisnik trenutno nalazi, pregled obavijesti za pojedinog korisnika, pregled i uređivanje profila korisnika, pregled svih grupnih razgovora u kojima se korisnik nalazi te sudjelovanje u razgovorima u stvarnome vremenu. Prije nego što se detaljnije pojasni svaka od funkcionalnosti aplikacije, proći će se način na koji je svaki od ovih funkcionalnosti formiran u kodu, tj. arhitektura pojedine funkcionalnosti te će biti pojašnjena kao i kontroliranje stanja pojedinih komponenata aplikacije (engl. *State Management*).

4.1. Korištene biblioteke

Razvijanje mobilnih aplikacija s *Flutter* radnim okvirom znatno ovisi o bibliotekama koje prave drugi developeri i o njihovom održavanju tih biblioteka. Svaka firma ima preferirani set biblioteka koje obično koristi pri izradi mobilnih aplikacija. Biblioteke koje su korištene pri izradi ove aplikacije se mogu vidjeti na Slici 4.1. Kako bi se koristile *Firebase* usluge moraju se dodati *Firebase* biblioteke. Nadalje, svaka aplikacije koja dohvaća podatke s nekakvog servera mora imati provjeru povezanosti na internet. Aplikacija u radu koristi *Connectivity* biblioteku koja pruža jednostavno sučelje za provjeru povezanosti na internet. Bitna stvar kod izrade bilo kakvog softvera jest kako developer predaje objekte konstruktorima drugih klasa, tj. kako se *dependencije* od objekata predaju drugim objektima (engl. *Dependency Injection*). U ovoj aplikaciji se taj problem rješava sa *GetIt* bibliotekom.

Preko *GetIt* instance se registriraju metode koje proizvode tražene objekte. U Programskom kodu 4.1. se vidi princip registriranja kreacije novih objekata pojedinih klasa: objekti se mogu registrirati u *GetIt* kao *singleton* instance, tj. da uvijek postoji samo jedna instanca objekta te klase i kao *factory* instance, tj. tako da se na svaki zahtjev novog objekta te klase kreira novi objekt. Biblioteke se navode u pubspec.yaml datoteci. Nadalje, većina aplikacije uz komunikaciju sa serverom može podatke dobiti i iz lokalne baze podataka. U ovoj aplikaciji se koristi biblioteka *SharedPreferences* za spremanje podataka. Podaci se lokalno spremaju kao tekstovi u JSON formatu i u obliku '*key* : *value*'. Pošto je u aplikaciji korištena *Google Maps* platforma, mora se dodati *Google Maps* biblioteka napravljena isključivo za *Flutter* radni okvir. Biblioteka *Geolocator* je korištena za pronalaženje trenutne lokacije korisnika, tj. dohvaćanje podataka o zemljopisnoj dužini i širini.



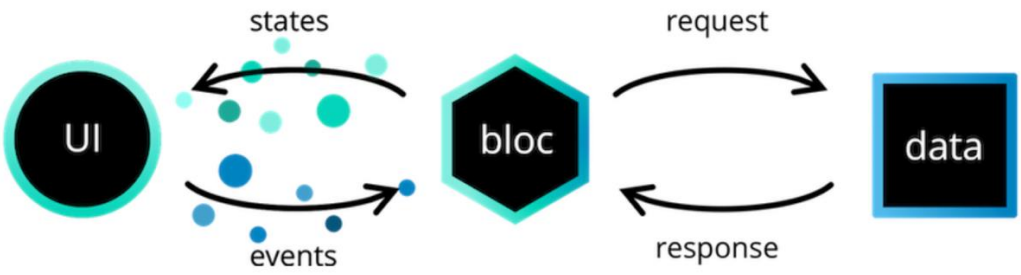
Slika 4.1. Biblioteke

```
getIt.registerFactory<AuthBloc>(() =>
    AuthBloc(logUserIn: getIt(), signUserIn: getIt(), signUserOut: getIt()));
 getIt.registerLazySingleton(() => LogUserIn(getIt()));
 getIt.registerLazySingleton(() => SignUserIn(getIt()));
 getIt.registerLazySingleton(() => SignUserOut(getIt()));
```

Programski kod 4.1. Registracija kreacije objekata sa *GetIt* bibliotekom

4.2. Upravljanje stanjem ekrana aplikacije

U svim aplikacijama na profesionalnoj razini se koriste određene tehnike upravljanja stanjem ekrana, stanjem dijela ekrana ili nečega drugog (engl. *State Management*). U ovoj aplikaciji se za upravljanje stanjem koristi biblioteka *Bloc*. *Bloc* biblioteka služi za upravljanje stanjem dijela ekrana u *Flutter* razvojnom okruženju. Korisničko sučelje šalje *Bloc* objektu događaje na osnovu kojih *Bloc* objekt šalje zahtjeve za dohvaćanje podataka koje treba prikazati na ekran. Nakon dohvaćanja podataka *Bloc* objekt procesira podatke i mijenja stanje prikladno. *Widget* sluša na promjenu stanja *Bloc* objekta i ponovo gradi dio *Widget* stabla koji ovisi o tom stanju, što se može vidjeti na Slici 4.2. *Widget* procesira stanje *Bloc* objekta pomoću *BlocBuilder* objekta. *BlocBuilder* objekt prima *builder()* metodu koja kao argumente ima *Context* objekt i objekt koji reprezentira stanje *Bloc* objekta [12]. U toj *builder()* metodi se rade provjere za pojedina stanja na osnovu kojih se pokazuju određeni *Widget* objekti. U Programskom kodu 4.2. se primjerice na *EventsOverviewLoading* stanje prikazuje okrugla traka za napredak (engl. *Circular Loading Spinner*)



Slika 4.2. Koncept *Bloc* biblioteke [13]

```
return BlocBuilder<EventsOverviewBloc, EventsOverviewState>(
    builder: (context, state) {
      if (state is EventsOverviewLoading)
        return Center(
          child: CircularProgressIndicator(),
        );
      if (state is EventsOverviewNetworkFailure)
        return NetworkErrorWidget(
          state.message,
          onReload: () {},
        );
      if (state is EventsOverviewServerFailure)
        return ServerErrorWidget(
```

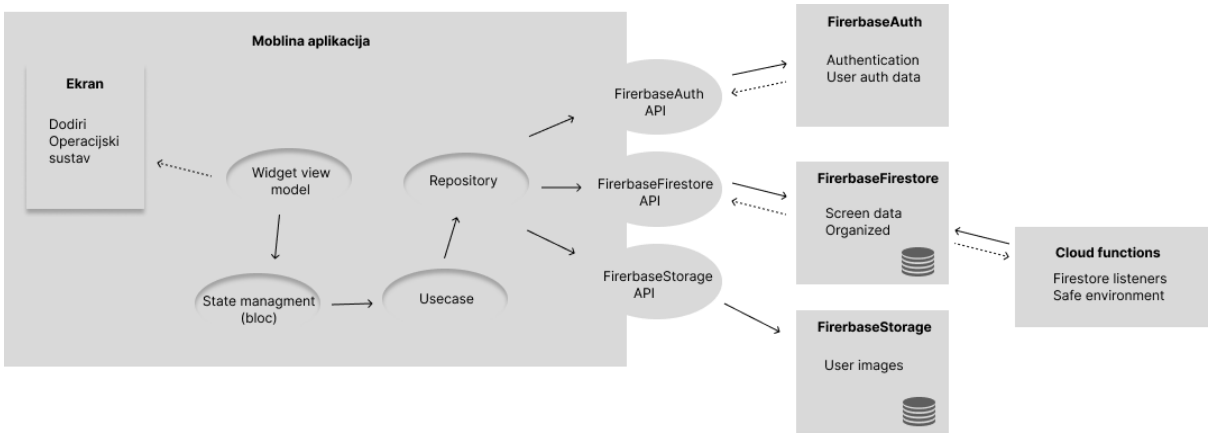
Programski kod 4.2. Procesiranje stanja *Bloc* objekta unutar *Widget* objekta

Umjesto *Bloc* objekta postoji i *Cubit* objekt koji je pojednostavljena verzija *Bloc* objekta. Razlika između njih dvoje je u tome što *Widget Bloc* objektu šalje događaje na slavinu (engl. *Pipe*) od njegovog *Stream* objekta, dok *Widget* na *Cubit* objektu poziva konkretne metode [14].

4.3. Arhitektura aplikacije i baze podataka

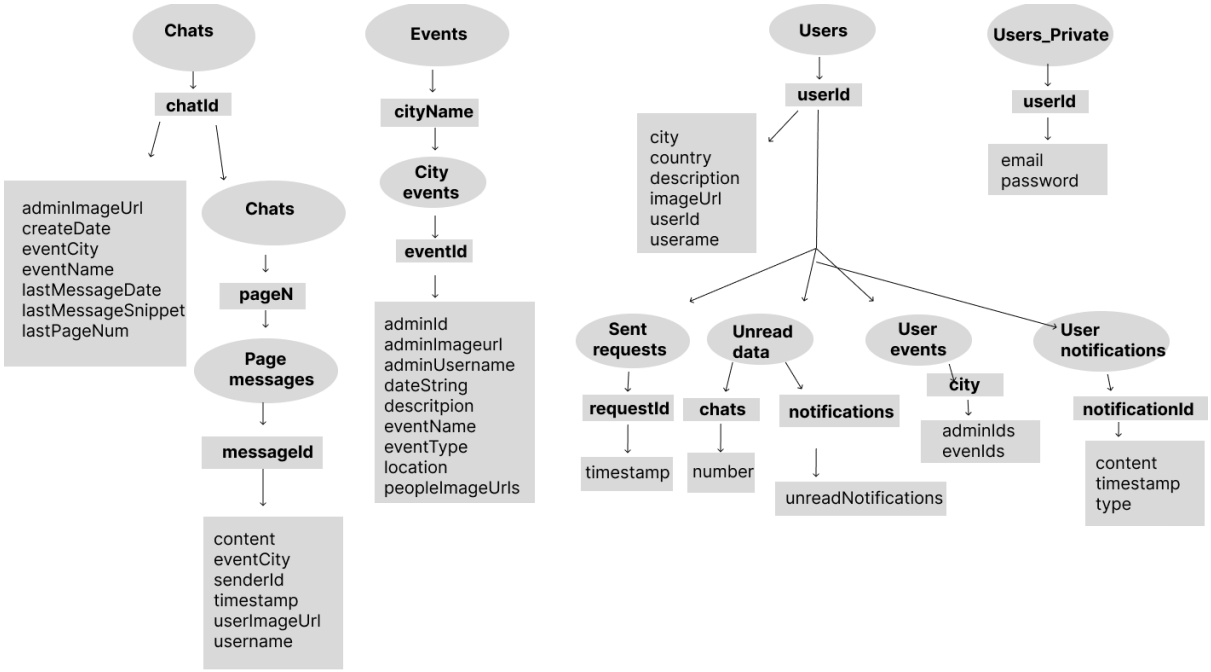
Arhitektura predstavlja način na koji developer organizira svoje datoteke, način na koji je softver odvojen na logičke komponente, način na koji te komponente komuniciraju jedno s drugima te na način na koji podaci prolaze kroz svaku od pojedinih komponenata. Arhitektura aplikacije analizirane u radu je prikazana na Slici 4.3. i dijeli se na dva glavna dijela: arhitektura mobilne aplikacije i arhitektura *Firebase* usluge. Mobilna aplikacija komunicira s *Firebase* uslugama preko njihovih javnih sučelja. *Firebase Authentication* usluga pruža usluge registracija i prijave korisnika u aplikaciju. Također, pruža analizu korištenja aplikacije tako što daje podatke o dnevnom i mjesečnom broju aktivnih korisnika. Postoje više načina registracije korisnika kao što su elektronička pošta, *Google* registracija, *Facebook* registracija, *GitHub* registracija, registracija mobitelom, itd. U ovoj aplikaciji se koristi samo način registracije i prijave korisnika putem elektroničke pošte (više načina može biti uključeno). U

aplikaciji se *Firebase Authentication* sučelje koristi kako bi se pristupilo trenutno prijavljenom korisniku i koristi se *Stream* objekt koji naznačuje promjenu stanja prijavljenosti korisnika.



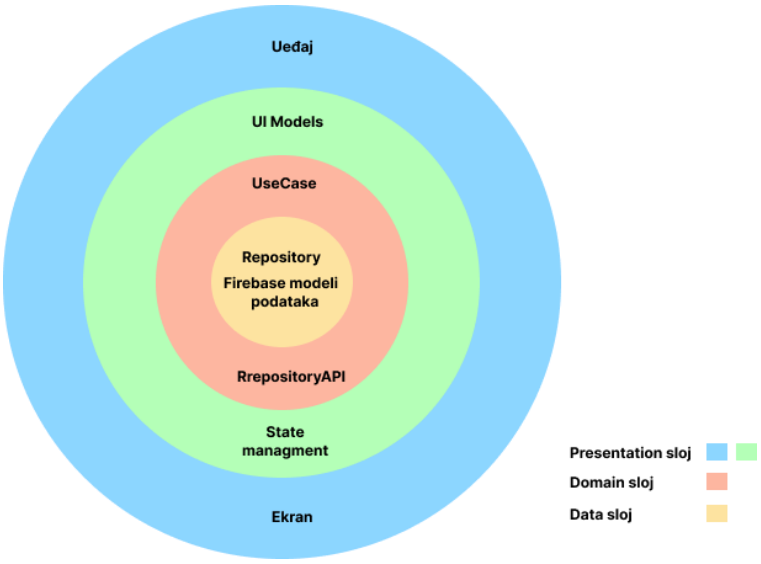
Slika 4.3. Arhitektura aplikacije

Firebase Firestore predstavlja NoSQL bazu podataka gdje se čuvaju i manipuliraju podaci potrebni za izradu ekrana aplikacije. Proces planiranja izgleda aplikacije i podataka kojih korisnik želi vidjeti prethodi izradi arhitekture *Firestore* baze podataka. Većinu informacija prema kojima će *Firestore* model biti modeliran za taj određeni dio aplikacije se može dobiti razmatrajući podatke potrebe za prikaz korisničkog sučelja. Na Slici 4.4. se upravo iz toga razloga mogu prepoznati funkcionalnosti aplikacije i odrediti za koji ekran su određene informacije potrebne. *Firebase Storage* je usluga koja omogućuje spremanje datoteka u oblaku. U ovoj aplikaciji se ova usluga koristi za spremanje



Slika 4.4. *Firebase Firestore* arhitektura

slika u JPEG formatu. Aplikacija je podijeljena na funkcije koje obavlja. Arhitektura aplikacije je podijeljena na tri sloja na razini pojedinih funkcionalnosti: sloj podataka, sloj domena i prezentacijski sloj (Slika 4.5.). Prezentacijski sloj služi za prikazivanje sadržaja korisniku na ekranu, služi za procesiranje korisnikovog dodira i ostalih akcija koje mijenjaju izgled ekrana te drži poslovnu logiku koja zove sloj domena kako bi dobio informacije na osnovu kojih se određuje novo stanje ekrana. Sloj domena sadrži metode i klase koje svojim imenom govore developeru što rade – npr. *getPersonOib()*. Sloj domena za dohvaćanje podataka zove repozitorij iz sloja podataka. Repozitoriji imaju svoj apstraktne klase i svoje konkretne implementacije. Apstraktne klase se pružaju sloju domena i on ne zna što se događa unutar konkretnih implementacija. Repozitoriji dohvaćaju podatke iz lokalnih izvora podataka ili šalju zahtjeve serveru. Sloj domena ne zna odakle je repozitorij dohvatio te podatke i ne zanima ga, njega zanimaju samo rezultati. Repozitoriji sam određuju hoće li dohvatiti podatke lokalno ili će dohvatiti podatke sa servera. Repozitorij zove lokalne izvore podataka ili klijente koji prave HTTP/HTTPS zahtjeve [15].



Slika 4.5. Arhitektura na razini funkcionalnosti

Na Slici 4.6. je prikazan grubi opis dohvaćanja podataka sa *Firestore* baze podataka, pretvaranja modela baze podataka u model podataka koji se prikazuje na ekranu, tretiranje grešaka pri dohvaćanju podataka i mijenjanje stanja ekrana. Ako postoji internetska konekcija repozitorij šalje preko *Firebase Firestore* sučelja zahtjev za dohvaćanje događaja pomoću identifikacijskog broja događaja. Dobiva se model podataka događaja baze podataka koji se pretvara u podatke potrebne za prikaz na ekranu. Vraća se *Right* objekt iz *Either* biblioteke koji u ovome slučaju predstavlja uspješno dohvaćanje podataka. Sloj domene propagira *Event* objekt prezentacijskom sloju, tj. *Cubit* objektu.

Widget

```
if(state is GotEvent){
  return EventWidget(state.event)
}else if(state is ErrorState) {
  return ErrorWidget();
}
return LoadingWidget();
```

Cubit

```
getEvent(String id){
  response = usecase.getEvent(id);
  response.fold(
   (failure) ⇒ emit(ErrorState()),
   (event) ⇒ emit(GotEvent(event))
  )
}
```

Usecase

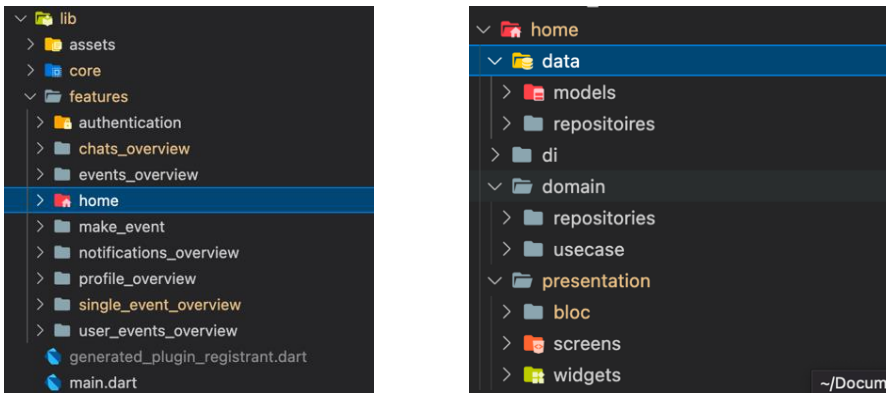
```
Either<Failure,Event> getEvent(String id){
  return repository.getEvent();
}
```

Respository

```
Either<Failure,Event> getEvent(String id){
  if(!isConnected)
    return Left(NetworkFailure());
  try{
    eventModel = getEvent(id);
    event = eventModel.toUiModel();
    return Right(response);
  }catch{
    return Left(ServerFailure());
  }
}
```

Slika 4.6. Grubi opis interakcije slojeva

Ako dođe do greške prilikom dohvaćanja podataka s *Firestore* baze podataka mobilnoj aplikaciji se vraća *Left* objekt koji signalizira pogrešku. *Error* objekt koji se uhvati pri dohvaćanju podataka sa *Firestore* baze podataka se transformira u *Failure* objekt. Razlog tome je što *Firestore* može generirati velik broj pogrešaka, a izgled ekrana ne ovisi o vrsti pogreške koju *Firestore* baci. Pravi se razlika između greške pri dohvaćanju podataka zbog *Firestore* pogreške i zbog toga što nema internetske konekcije. Organizacija arhitekture pojedine funkcionalnosti mobilne aplikacije se također može vidjeti u organizaciji datoteka projekta (Slika 4.7.). Datoteke su organizirane za pojedine funkcionalnosti na 3 glavne datoteke koje odgovaraju imenima slojeva. Uz te datoteke se nalazi i datoteka u kojoj se nalazi kod za kreiranje objekta pružanjem objekata o kojim on ovisi, te datoteka koja sadrži neke korisne funkcije.



Slika 4.7. Organizacija datoteka

4.3.1. Alternativna arhitektura aplikacije

Tijekom izrade diplomskog rada su se mogli izvući prijedlozi promjene arhitekture koje mijenjaju strukturu datoteka i koje mijenjanju protok informacija iz sloja podataka u prezentacijski sloj. Prvi prijedlog jest zamjena strukture datoteka. Trenutno je aplikacija raspoređena po funkcionalnostima koje pruža korisniku i svaka od tih funkcionalnosti sadrži tri sloja. Ako se zanemare ta tri sloja i gledaju se samo objekti koje svaka funkcionalnost sadrži, oni su: *Widget*, repozitoriji, pomoćni objekti, *Bloc* ili *Cubit* objekti te možda neki servisi i menadžeri. Promjena organizacije datoteka proizlazi iz sljedećeg: funkcionalnosti aplikacije mogu i često dijele repozitorije. Stoga je prijedlog pomicanje implementacije i apstrakcije repozitorija u datoteku u kojoj se nalaze klase i funkcije koje više slojeva koriste. Na ovaj način bi svaka funkcionalnost bila podijeljena prema onome što sadrži na ekranu. Naravno, može biti slučajeva gdje više funkcionalnosti mogu dijeliti isti *Widget* objekt ili čak *Bloc*i ili *Cubit* objekt, no takvi slučajevi su rijetki i ovise o dizajnu aplikacije.

Drugi prijedlog jest izbacivanje *Either* objekta. *Either* objekti se moraju raspakirati u jednome trenutku. Ako se za dobivanje podataka potrebnih za prikaz na ekran trebaju dohvatiti dvije stvari, može doći do raspakiravanja *Either* objekta unutar drugog *Either* objekta. Ovakva situacije se primjerice događa u Programskom kodu 4.3. gdje se prvobitno treba dobiti informacija o trenutnoj lokaciji korisnika prije dohvaćanja podataka o događajima. Jedna informacija je u jednom repozitoriju a druga u drugome repozitoriju. Raspakiravanje u raspakiravanju se može izbjeći ako repozitoriji ne vraća *Either* objekt već *UseCase* objekt interpretira greške i vraća *Either* objekt. Solucija za rješenje ovog problema uključuje izbacivanje *Either* objekta, stoga varijacija na arhitekturu ne uključuje uopće *Either* objekte. Varijacija također podrazumijeva hvatanje grešaka u repozitoriju, ali umjesto da se greške pretvaraju u *Failure* objekte, greške se pretvaraju u manji, konačni i razumljiv set grešaka koji se propagiraju do *Bloc* ili *Cubit* objekta. Tako se umjesto raspakiravanja programski kod piše normalno unutar *try-catch* bloka i u *catch* bloku se nalazi logika za mijenjanje stanja ekrana na osnovu grešaka. Također, u sloju domena se koriste *UseCase* objekti koji uvijek pozivaju repozitorij i samo prosljeđuju njegov rezultat. Korisnost *UseCase* objekta proizlazi u tome što riječima opisuje što se događa, tj. opisuju se podaci potrebni prezentacijskom sloju, i kako bi se odradila neka dodatna logika ako je potrebno. No, odrađivanje dodatne logike nije potrebno ako se dobro rasporedi posao između komponenata arhitekture. Stoga, varijacija na postojeću arhitekturu ne bi imala sloj domena.

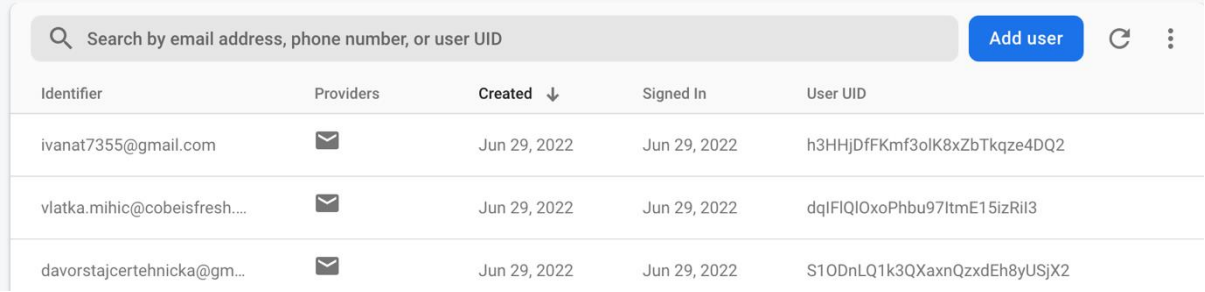
```
Stream<EventsOverviewState> _onScreenInitialized() async* {
 final locationResponse = await locationService.getLocation();
 yield* locationResponse.fold((
  locationFailure,
 ) async* {
  if (locationFailure is NetworkFailure)
   yield EventsOverviewNetworkFailure(locationFailure.message);
  else
   yield EventsOverviewServerFailure(locationFailure.message);
 }, (
  currentPosition,
 ) async* {
  final Either<Failure, List<Event>> response =
     await getEventsForCurrentLocation(
    LatLng(currentPosition.latitude, currentPosition.longitude),
   );

  yield* response.fold((failure) async* {
   if (failure is NetworkFailure)
    yield EventsOverviewNetworkFailure(failure.message);
   else
    yield EventsOverviewServerFailure(failure.message);
  }, (listOfEvents) async* {
   yield EventsOverviewLoaded(events: listOfEvents);
  });
 });
}
```

Programski kod 4.3. Mijenjanje stanja *Bloc* objekta

4.4. Prijava korisnika

Prijava na aplikaciju se vrši pomoću *Firebase* platforme koja daje mogućnost korištenja *Firebase Authentication* alata. *Firebase Authentication* alat omogućuje developeru registraciju korisnika s više mogućih metoda: elektroničkom poštom, registracija mobitelom, anonimna prijava, *Google* registracija, *Facebook* registracija, *GitHub* registracija i dr. U ovoj aplikaciji se koristi registracija i prijava elektroničkom poštom. Na *Firebase* platformi developer može vidjeti pregled svih korisnika koji su trenutno registrirani na aplikaciji, s kojom metodom su registrirani i u koje vrijeme, što se vidi na Slici 4.8. Ulazak u aplikaciju je podijeljen na dva dijela: registracija i prijava. Kako bi se korisnik prijavio u aplikaciju mora se prethodno registrirati . Na Slici 4.9. se vidi način na koji je korisniku prikazana registracija i prijava unutar aplikacije. *Firebase Authentication* alat daje vrlo jednostavno sučelje developerima za prijavu i registraciju korisnika što se vidi na Programskom kodu 4.4 i Programskom kodu 4.5. Također, *Firebase Authentication* alat pruža developeru kroz svoje sučelje način dolaska do podataka o

trenutno prijavljenom korisniku, a u slučaju da niti jedan korisnik nije trenutno logiran, onda je parametar koji sadrži podatke o korisniku vrijednosti *null*.



Slika 4.8. Pregled registriranih korisnika na *Firebase* platformi

```
await firebaseAuth.createUserWithEmailAndPassword(
  email: parameters.email,
  password: parameters.password,
);
```

Programski kod 4.4. Registracija s *Firebase Authentication* alatom

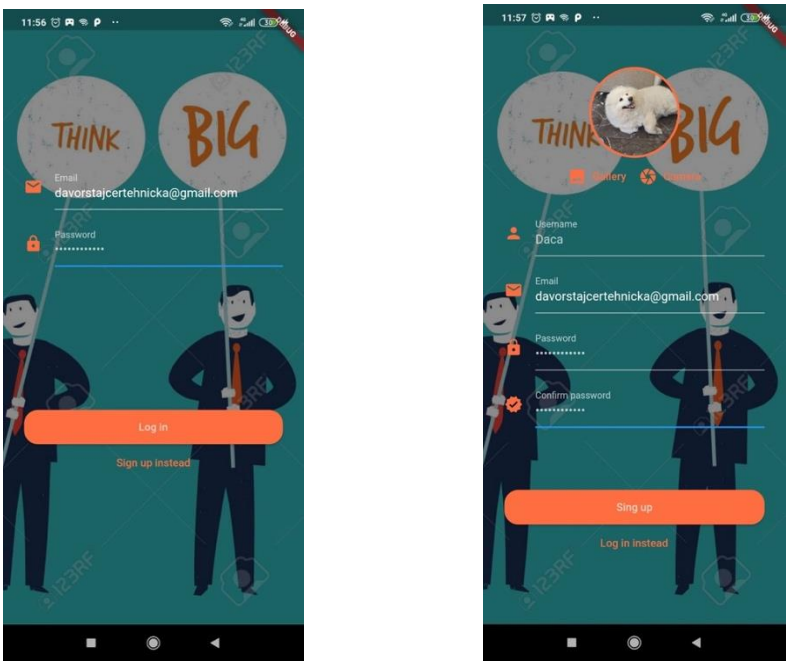
```
await firebaseAuth.signInWithEmailAndPassword(
  email: parameters.email,
  password: parameters.password,
);
```

Programski kod 4.5. Prijava s *Firebase Authentication* alatom

Developer također odjavi korisnika iz aplikacije preko *Firebase Authentication* sučelja s metodom *logOut()*. *Firebase Authentication* sučelje također pruža *Stream* objekt u kojem se pojavljuju podaci kada se korisnik prijavi, odjavi ili kada se registrira. Ovaj *Stream* objekt je iskorišten od strane *Widget* tako da je napravljen *Bloc* objekt koji se pretplati na taj *Stream* objekt i na osnovu događaja mijenja svoje stanje i ponovo stvara korisničko sučelje na način prikazan u Programskom kodu 4.6.

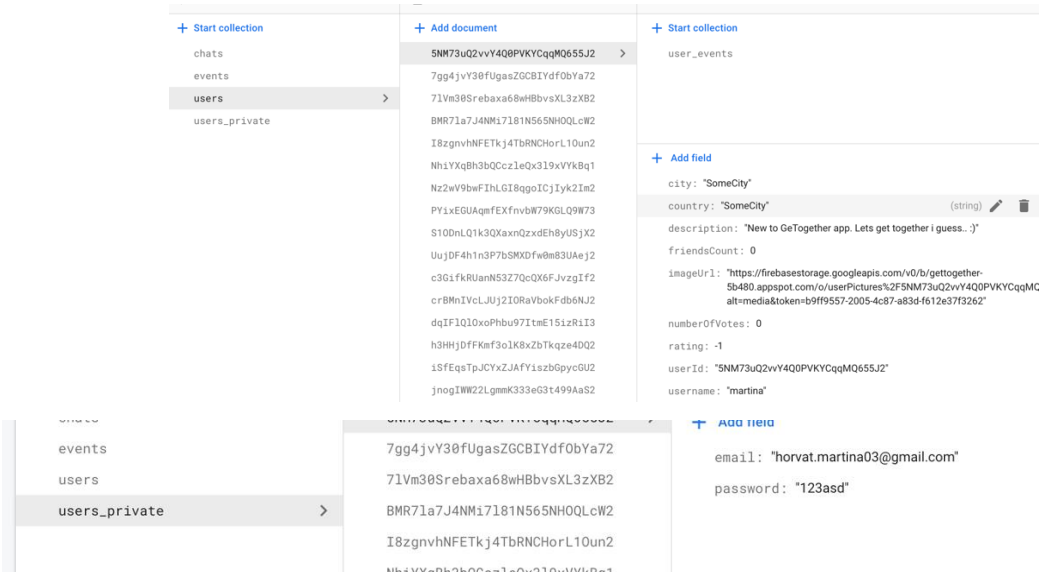
```
builder: (context, state) {
  if (state is AuthenticationCheckInitialState)
    return SplashScreen();

  if (state is UserLoggedInState) return HomeScreenWidget();
  return AuthScreen();
},
```

Programski kod 4.6. Reagiranje *Widget* objekta na promjenu stanja prijave korisnika



Slika 4.9. Prijava i registracija korisnika

Nakon registracije korisnika se na *Firebase* platformi automatski spremaju podaci o korisniku. Također, podaci o korisniku se spremaju u *Firebase Firestore NoSQL* bazu podataka. Posebno se spremaju osjetljivi podaci o korisniku kao što su elektronička pošta i lozinka što se može vidjeti na Slici 4.10. Svaki korisnik dodatno sadrži podatke o događajima na koje je prijavljen.

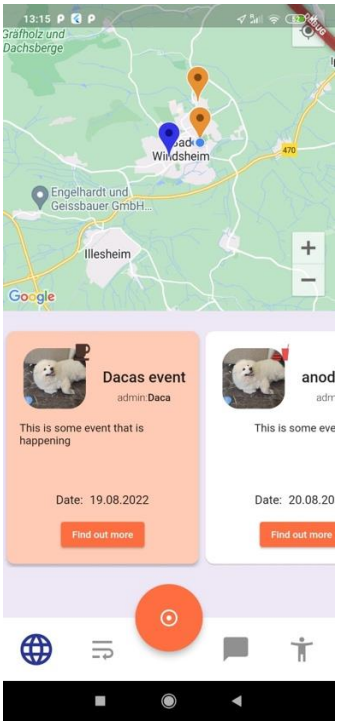


Slika 4.10. Podaci o korisniku na *Firebase Firestore* platformi

4.5. Pregled događaja

Nakon korisnikove prijave u aplikaciju otvara mu se ekran gdje može vidjeti sve događaje u njegovoj blizini. Korisnik može pregledavati događaje u listi ili može pregledavati

događaje na *Google Maps* karti. Ovdje se mogu vidjeti pojedine informacije o događajima kao što su ime događaja, lokacija događaja, korisnik koji je napravio događaj, datum odvijanja događaja te opis događaja. Sve navedeno se može vidjeti na Slici 4.11. Na ovome ekranu se pomoću *Google Geolocator* usluge dohvaćaju podaci o korisnikovoj trenutnoj lokaciji, ako je korisnik odobrio u postavkama mobitela da aplikacija smije pristupiti njegovoj trenutnoj lokaciji. Nakon dohvaćanja njegove trenutne lokacije se podaci o lokaciji, tj. zemljopisna širina i dužina pomoću *Google Geocoding* sučelja pretvaraju u podatke o trenutnome gradu ili selu gdje se korisnik nalazi.



Slika 4.11. Pregled događaja

Postupak dohvaćanja trenutne lokacije je prikazan na u Programskom kodu 4.7., a postupak pretvaranja lokacije u grad ili selo u Programskom kodu 4.8. S tim podacima se može pomoću *Google Maps* biblioteke za *Flutter* razvojno okruženje prikazati korisnikova trenutna lokacija i lokacija događaja u njegovoj okolini, tj. gradu ili selu. Na *Firebase Firestore* bazi podataka se događaji spremaju za pojedini grad jer je to i princip aplikacije – prikazivanje događaja u korisnikovom trenutnom mjestu (Slika 4.12.). Pošto u aplikaciji postoji ekran gdje se prikazuju svi događaji za pojedinog korisnika, u bazi podataka se za svakog korisnika održavaju identifikacijski brojevi događaja u kojima je korisnik uključen i razlikuju se događaji u kojima je korisnik administrator i u kojima samo sudjeluje (Slika 4.13.). Na ovaj način se mogu odjednom povući svi događaji za korisnika nakon ulaska u aplikaciju. Također, to znači da će događaji biti duplicirani u bazi podataka. Događaj koji ima N korisnika će biti N puta u

bazi podataka. Obično je ovakvo ponašanje nešto što se nastoji izbjegavati, ali NoSQL baze podataka rade na principu iskorištavanja količine operacija čitanja podataka. Također, više operacija čitanja i pisanja podataka predstavljaju veći financijski trošak. No, operacija čitanja podataka je daleko jeftinija od operacije pisanja podataka. Dupliciranjem podataka tako da podaci budu generalno organizirani u cjelini u ovisnosti o tome kojem ekranu pripadaju pruža optimalan broj operacija čitanja i pruža jednostavnije upite na *Firebase Firestore* bazu podataka [16].

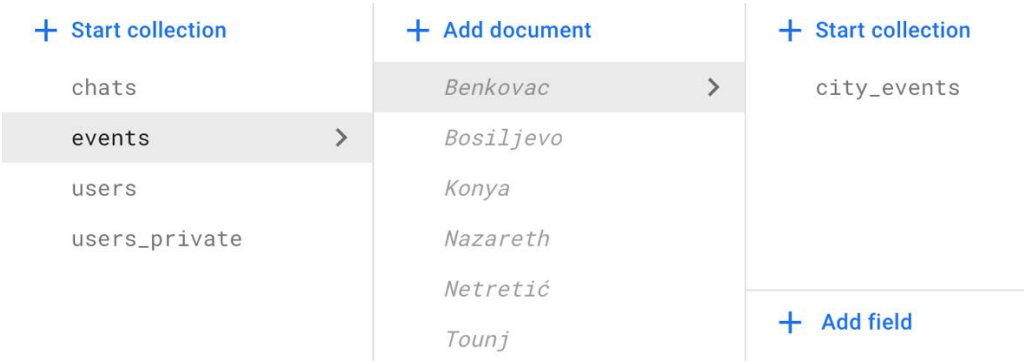
```
if (permission == LocationPermission.denied) {
      permission = await Geolocator.requestPermission();
      if (permission == LocationPermission.denied) {
        return Left(
            LocationFailure(message: 'Location permissions are denied'));
      }
    }
    if (permission == LocationPermission.deniedForever) {
      return Left(LocationFailure(
          message:
              'Location permissions are permanently denied, we cannot request permissions.'));
    }
    final position = await Geolocator.getCurrentPosition();
    return Right(position);
```

Programski kod 4.7. Dohvaćanje trenutne lokacije korisnika

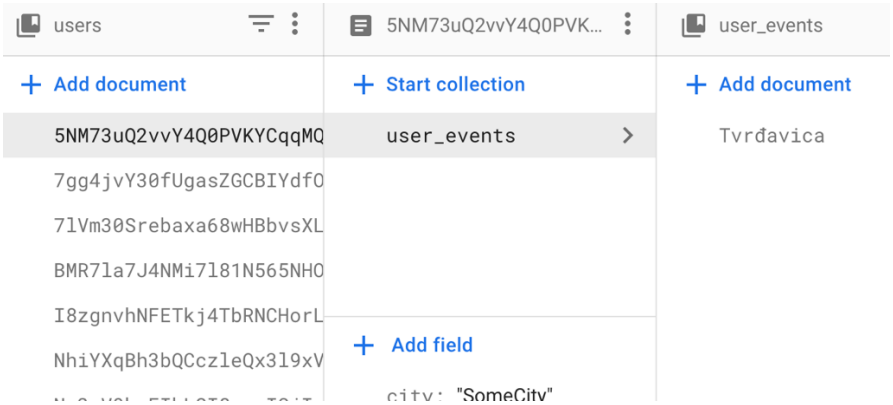
```
  final response = await http.get(
    Uri.https("maps.googleapis.com", "/maps/api/geocode/json", {
      "latlng": "${location.latitude},${location.longitude}",
      "key": "AIzaSyDyi4MtJcldL85gBRT_STdjg8ckpCxpFY4",
    }),
  );
  final jsonRepsonse = json.decode(response.body);
  final List<dynamic> results = jsonRepsonse["results"];
  String? city;
  results.forEach((result) {
    (result["address_components"] as List).forEach((adressComponent) {
      (adressComponent["types"] as List).forEach((type) {
        if (type == "locality") city = adressComponent["long_name"];
        if (type == "administrative_area_level_1" && city == null)
          city = adressComponent["long_name"];
      });
    });
  });
  return city;
```

Programski kod 4.8. Pretvaranje lokacije u podatke o gradu ili selu

Pri izradi *Google Maps* prikaza se prvobitno moraju dohvatiti događaji za pojedini grad koji su spremljeni na *Firestore* bazi podataka (Slika 4.12.) i kreiraju se markeri za svaki događaj. Ekran na Slici 4.14. se sastoji od dva dijela ako se zanemari donja navigacija: *Google Maps* prikaz i prikaz liste događaja. Ako korisnik odabere događaj iz liste događaja, lista se animira do toga događaja i *Google Maps* prikaz se pomiče do odabranog događaja. Obrnuto također vrijedi. *Google Maps* prikaz i prikaz liste događaja svoje akcije prosljeđuju *EventPickCubit* objektu koji ima jedno moguće stanje koje reprezentira odabrani događaj.



Slika 4.12. Događaji u *Firebase Firestore* bazi podataka



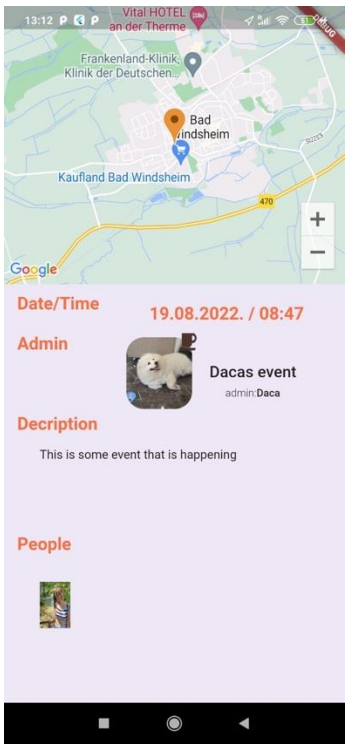
Slika 4.13. Događaji za pojedinog korisnika

EventPickCubit objekt se stavlja u stablo *Widget* objekata iznad *Widget* objekta koji predstavlja ove dvije navedene cjeline na ekranu. Pošto je poželjno da se u Flutter aplikaciji sve rastavlja na više *Widget* objekata ova situacija je česta pojava. Kreiranje i dohvaćanje događaja se sa strane korisnika odvija pomoću *EventsRepository* klase, što se vidi na Programskom kodu 4.9.

```
abstract class EventsRepository {
  Future<Either<Failure, List<Event>>> getAllEvents(LatLng currentLocation);
  Future<Either<Failure, Success>> createEvent(CreateEventData event);
}
```

Programski kod 4.9. *Events Repository*

Odabirom nekog od događaja s liste se korisnik vodi na ekran prikazan na Slici 4.14. Ovdje korisnik vidi informacije o događaju kao što su datum i vrijeme, informacije o kreatoru događaja, opis događaja, slike ljudi koji dolaze na događaj te *Google Maps* prikaz lokacije događaja. Prema ovim informacijama je model događaja modeliran u kodu klasom *Event* (Programski kod 4.10.). Korisnik može poslati kreatoru događaja zahtjev za pridruživanje događaju. Administrator dobiva obavijest o zahtjevu za pridruživanje događaju. Obavijesti su objašnjene u idućem poglavlju.



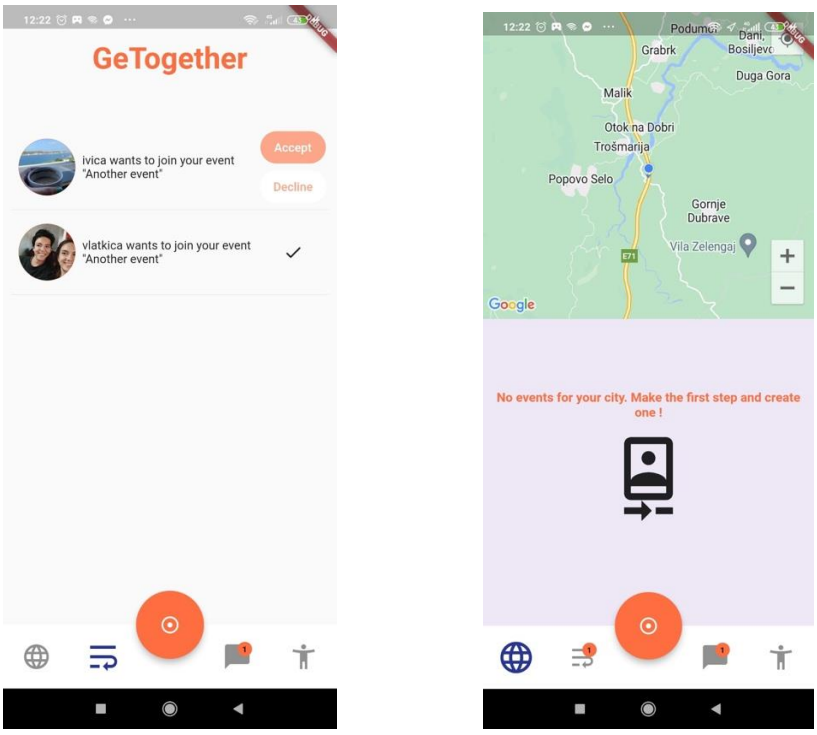
Slika 4.14. Pregled jednog događaja

```
class Event extends Equatable {
  final String eventId;
  final EventType eventType;
  final String dateString;
  final String timeString;
  final LatLng location;
  final String adminId;
  final String adminUsername;
  final String adminImageUrl;
  final int adminRating;
  final int numberOfPeople;
  final String eventName;
  final String description;
  final Map<String, dynamic> peopleImageUrls;
```

Programski kod 4.10. Model događaja

4.6. Obavijesti

U aplikaciji korisnik može primati notifikacije i postoji ekran gdje korisnik pregledava sve primljene notifikacije i njegove odgovore na njih, ako postoje. Obavijesti se sastoje od dva dijela: pregled svih obavijesti i prikazivanje nepročitanih obavijesti (Slika 4.15.). Uzimajući to u obzir, u *Firestore* bazi podataka su napravljene dvije *Firestore* kolekcije vezane za notifikacije. Jedna *Firestore* kolekcija koja osigurava podatke za prikaz svih notifikacija od korisnika, a druga za brojanje korisnikovih nepročitanih obavijesti. Pošto su obje *Firestore* kolekcije vezane za pojedinog korisnika stavljene su unutar *Firestore* kolekcije korisnika (Slika 4.16.). U aplikaciji postoje informativne obavijesti i obavijesti koje nude korisniku mogućnost prihvaćanja ili odbijanja zahtjeva od drugih korisnika. Model događaja u *Firestore* bazi podataka pokriva sve potrebe aplikacije, a to su podaci potrebni za prikaz na ekran, podaci potrebni za odgovor korisnika na zahtjev te ako je on pozitivan dodavanje korisnika u događaj, dodavanje događaja u korisnikovu listu događaja te dodavanje korisnika u grupni razgovor događaja.



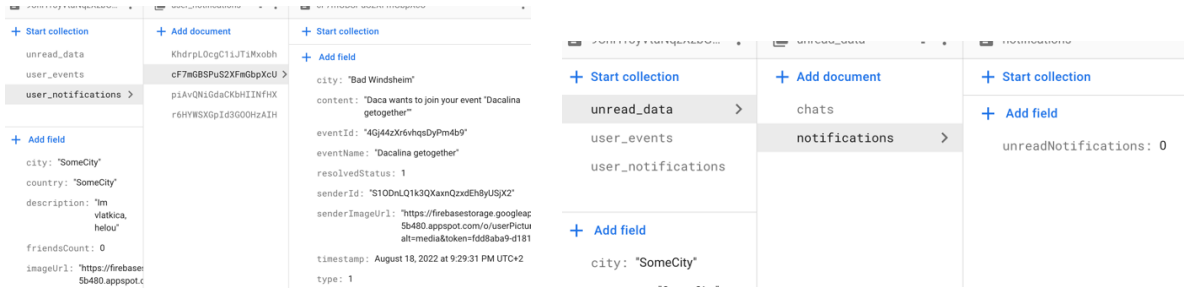
Slika 4.15. Pregled i prikaz nepročitanih obavijesti

Sav posao nakon pozitivnog odgovora na korisnikov zahtjev se ne odvija s klijentske strane već su napravljene *Firestore Cloud* funkcije. Puno posla se treba napraviti nakon korisnikovog pozitivnog odgovora. *Firestore Cloud* funkcije pružaju sigurno okruženje izvršavanja koda koji modificira stanje Firestore baze podataka, dok s klijentske strane svi zahtjevi moraju proći provjere i zaštite kao što su *Firestore* pravila (Slika 4.17.).

Funkcionalnosti obavijesti su nadopunjene *Firestore Cloud* metodama. Nakon što se notifikacija doda u *Firestore* bazu podataka okida se funkcija *onNotificationAdded()* koja povećava broj nepročitanih notifikacija koje korisnik ima (Programski kod 4.11.). Također, ako korisnik primi notifikaciju na koju treba dati odgovor, nakon davanja odgovora na notifikaciju se okida metoda *onJoinNotificationResolved()*. Nakon izvršavanja metode podnositelj zahtjeva je obaviješten o odgovoru.

```
export const onNotificationAdded = functions.firestore
    .document("users/{userId}/user_notifications/{notificationId}")
    .onCreate(async (change, context) => {
        functions.logger.log(`NOTIFICATION ADDED`)
        let notificationData = change.data()
        let notificationType: number = notificationData.type
        if (notificationType == null) return
        await incrementUnreadNotifications(context.params.userId, notificationData.eventId)
        if (notificationType == NotificationType.join.valueOf())
            await onJoinNotificationCreated(notificationData.senderId, notificationData.eventId)

export const onJoinNotificationResolved = functions.firestore
    .document("users/{userId}/user_notifications/{notificationId}")
    .onUpdate(async (change, context) => {
        let notificationDataBefore = change.before.data()
        let notificationDataAfter = change.after.data()
        functions.logger.log(`NOTIFICATION MODIFIED TO RESOLVED STATUS ->
${notificationDataAfter.resolvedStatus}, BEFORE -> ${notificationDataBefore.resolvedStatus}`)
        if (notificationDataBefore.resolvedStatus != NotificationResolved.pending.valueOf()) return
        if (notificationDataAfter.resolvedStatus == NotificationResolved.accepted.valueOf())
            await sendAcceptedNoitifcation(notificationDataAfter.senderId, notificationDataAfter.eventName,
notificationDataAfter.eventId)
        else
            await sendRejectedNoitifcation(notificationDataAfter.senderId, notificationDataAfter.eventName,
notificationDataAfter.eventId)
        await deleteResolvedRequest(notificationDataAfter.senderId, notificationDataAfter.eventId)


    })
```

Programski kod 4.11. *Cloud funkcije notifikacija*



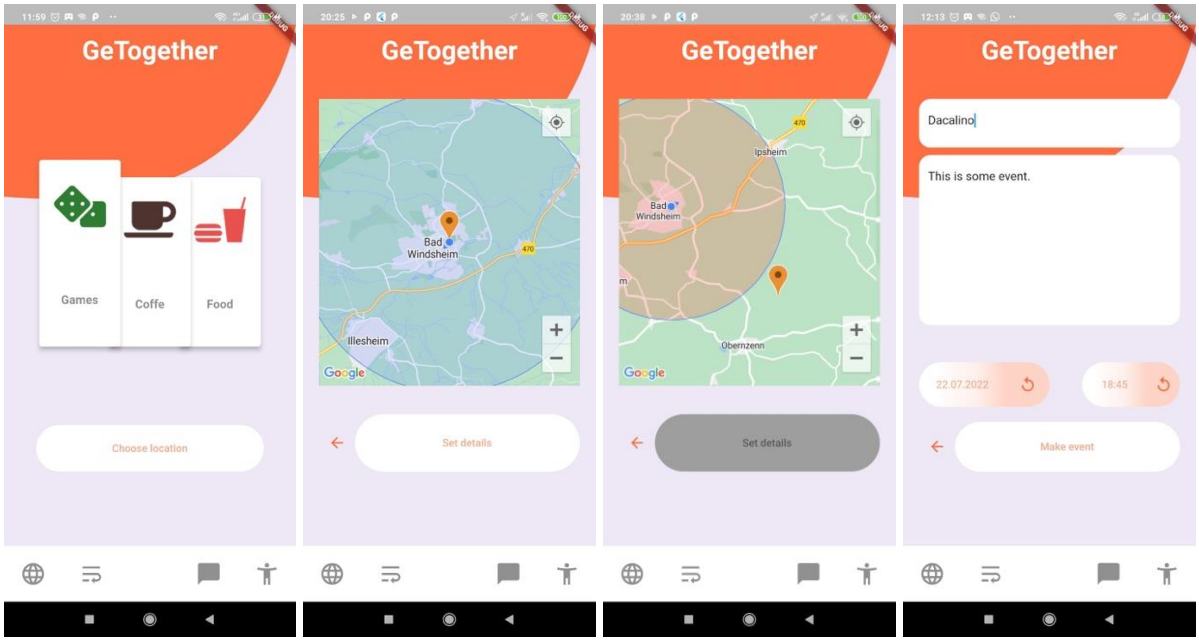
Slika 4.16. *Firebase Firestore* pregled notifikacija i nepročitane notifikacije

Slika 4.17. *Firestore* pravila

4.7. Kreiranje događaja

Korisnik ima mogućnost kreiranja događaja. Ekrani za kreiranje događaja su prikazani na Slici 4.18. Prilikom kreiranja događaja korisnik bira jednu od 3 vrste događaja: igre, kava i hrana. Nadalje, korisnik bira lokaciju događaja. Na ovome ekranu korisnik može izabrati lokaciju događaja samo u radijusu od pet kilometara od njegove trenutne lokacije. Lokacija se bira tako što se drži marker koji označava lokaciju i pomiče se po ekranu. Radijus od pet kilometara je označen na *Google Maps* karti s plavom kružnicom. Ako korisnik odvuče marker po ekranu na mjesto izvan radijusa od pet kilometara kružnica se popunjava crvenom bojom označavajući grešku i gumb za prelazak na idući korak se mijenja u tamniju boju indicirajući da je deaktiviran. Nadalje, nakon lokacije dolazi ekran s postavljanjem detalja događaja. Na tome ekranu se opisuju detalji događaja kao što su ime, opis, vrijeme i datum događaja. U svakome koraku postoji opcija za vratiti se na prethodni korak i promijeniti prethodno unesene podatke. Nakon unosa svih podataka potrebnih za kreaciju događaja, događaj se može kreirati. Nakon kreacije događaja korisnik je vraćen na ekran pregleda događaja gdje se pojavljuje novonastali događaj. Stanje kreiranja događaja kontrolira *EventCubit* objekt, a prikaz odabranog tipa događaja kontrolira *EventCardOrderCubit* objekt (Slika 4.18.). *EventCardOrderCubit* objekt drži listu tipova događaja. Tip događaja koji je prvi u listi predstavlja karticu vrste događaja koja će biti ispred svih kartica, na Slici 4.18. su to igre. Inicijalno stanje je stanje u kojemu je kava na prvome mjestu. Na promjenu redoslijeda tipa događaja se obavještava *EventCardOrderCubit* i *EventCubit* također pošto on drži stanje kompletnog stvaranja događaja. Stanje stvaranja događaja je reprezentirano stanjima:

EventStateUnfinished – korisnik popunjava parametre događaja ali nije ispunio sve događaje, *EventStateFinished* – korisnik je validno ispunio sve konfiguracije događaja, *EventLocationOutOfRange* – korisnik je unijo lokaciju koja je izvan pet kilometara od trenutne lokacije korisnika, *EventStateFailure* – dogodila se greška pri stvaranju događaja, *EventStateCreated* – događaj je stvoren (Slika 4.19.).

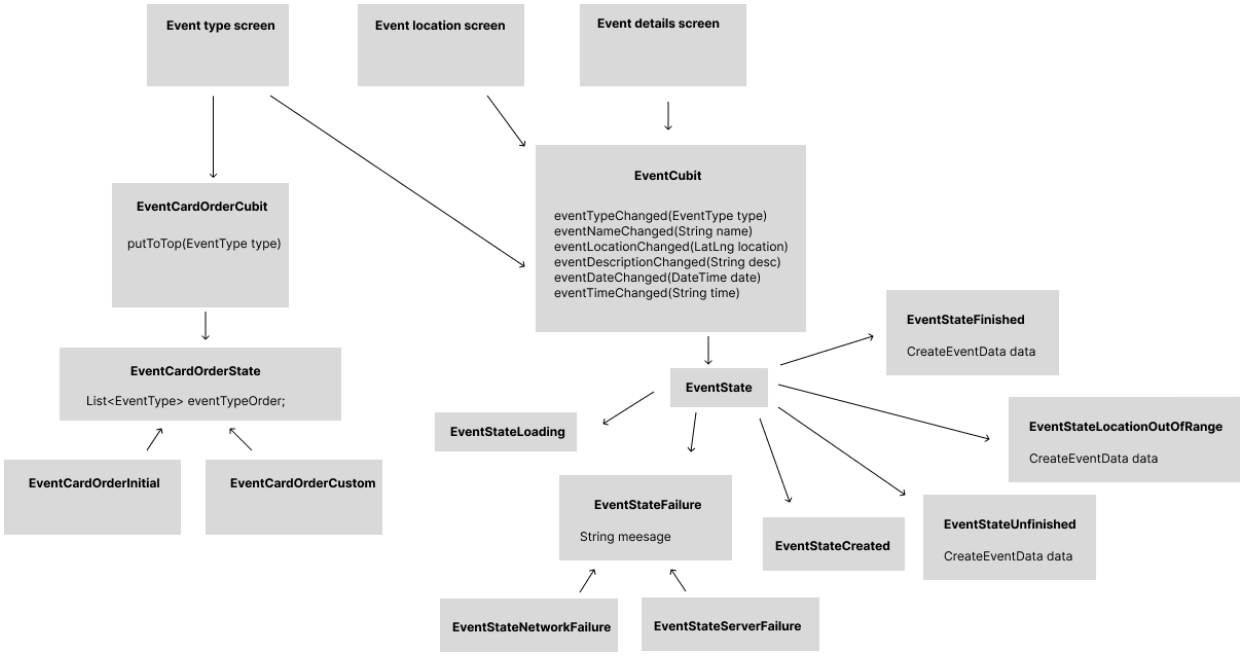


Slika 4.18. Kreiranje događaja

Događaji su organizirani u *Firestore* bazi podataka po gradovima, što znači da za spremanje događaja u bazu podataka mora biti poznato mjesto u kojem se korisnik trenutno nalazi. Za dohvaćanje korisnikove trenutne lokacije, pretvaranja te lokacije u grad i slušanje promjena korisnikove trenutne lokacije služi *LocationService* (Programski kod 4.12.). Pri kreiranju događaja repozitorij koristi usluge *LocationService* objekta. *Firestore Cloud* funkcija na Programskom kodu 4.13. je zadužena za brisanje grupnog razgovora događaja nakon njegovog brisanja iz baze podataka. Također, nakon što se obriše grupni razgovor događaja, korisnici koji su imali nepročitane poruke iz tog razgovora i dalje imaju informaciju o tim nepročitanim porukama. Stoga, ova *Firestore Cloud* funkcija se brine i za održavanje broja nepročitanih poruka tako da makne utjecaj nepročitane poruke od obrisanog razgovora.

```
abstract class LocationService {
  final NetworkInfo networkInfo;
  Future<Either<Failure, Position>> getLocation();
  Future<String?> mapLocationToCity(LatLng location);
  Stream<Position> userLocationStream();
  LocationService({required this.networkInfo});
}
```

Programski kod 4.12. *LocationService* API



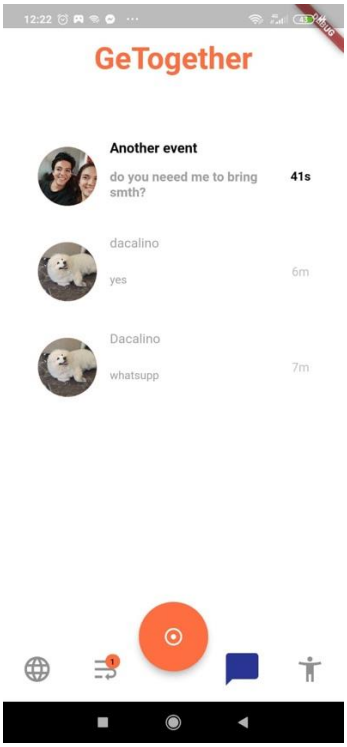
Slika 4.19. Reprezentacija stanja kreiranja događaja

```
export const onEventDeleted = functions.firestore
    .document("events/{city}/city_events/{eventId}")
    .onDelete(async (snapshot, context) => {
        let eventData: FirebaseFirestore.DocumentData = snapshot.data()
        functions.logger.log(`chat id -> ${context.params.eventId}`)
        await deleteEventGroupChat(context.params.eventId)
        let idPicMap = eventData.peopleImageUrls as Map<string, any>
        functions.logger.log(`id Pic map -> ${Object.entries(idPicMap).keys}`)
        let eventUserIds : Array<string> = Array();
        Object.entries(idPicMap).forEach(([userId, imageUrl]) => {
            eventUserIds.push(userId)
        })
        eventUserIds.push(eventData.adminId)
        for(var userId of eventUserIds){
            await  removeEventIdFromUser(userId,  context.params.eventId,  context.params.city,  eventData.eventName,
eventData.adminId)
            await admin.firestore()
            .collection("users")
            .doc(userId)
            .collection("unread_data")
            .doc("chats")
            .update({
                "newlyMessagedChats" : admin.firestore.FieldValue.arrayRemove(userId),
                "number" : admin.firestore.FieldValue.increment(-1),
            })

        }
        functions.logger.log(`admin id -> ${eventData.adminId}`)
```

Programski kod 4.13. *Cloud* funkcije događaja

4.8. Poruke

Nakon što korisnik kreira događaj stvara se grupni razgovor. Kako administrator prihvaća zahtjeve korisnika da se pridruže njegovom događaju, tako se povećava broj korisnika koji sudjeluju u grupnom razgovoru. Razlikuju se dvije funkcionalnosti aplikacije vezane za razmjene poruka: pregled razgovora (Slika 4.20.) i razmjenjivanje poruka unutar razgovora (Slika 4.21.). Pregled poruka se sastoji od svih grupnih razgovora u kojem korisnik sudjeluje. Svaki od tih razgovora sadrži informacije o zadnjem korisniku koji je poslao poruku, imenu događaja, kratki sadržaj poruke te proteklo vrijeme od slanja poruke. Ako korisnik ima otvoren ekran na Slici 4.20. i netko pošalje poruku u taj razgovor, taj razgovor će se podignuti na prvo mjesto iznad svih razgovora i tekst će postati podebljan kako bi označio korisniku da ima nepročitanih poruka u razgovoru. Ovaj efekt se postigao sa *Firestore* mogućnosti slušanja podataka gdje se podaci na ekranu mogu prikazivati u stvarnome vremenu.



Slika 4.20. Pregled razgovora

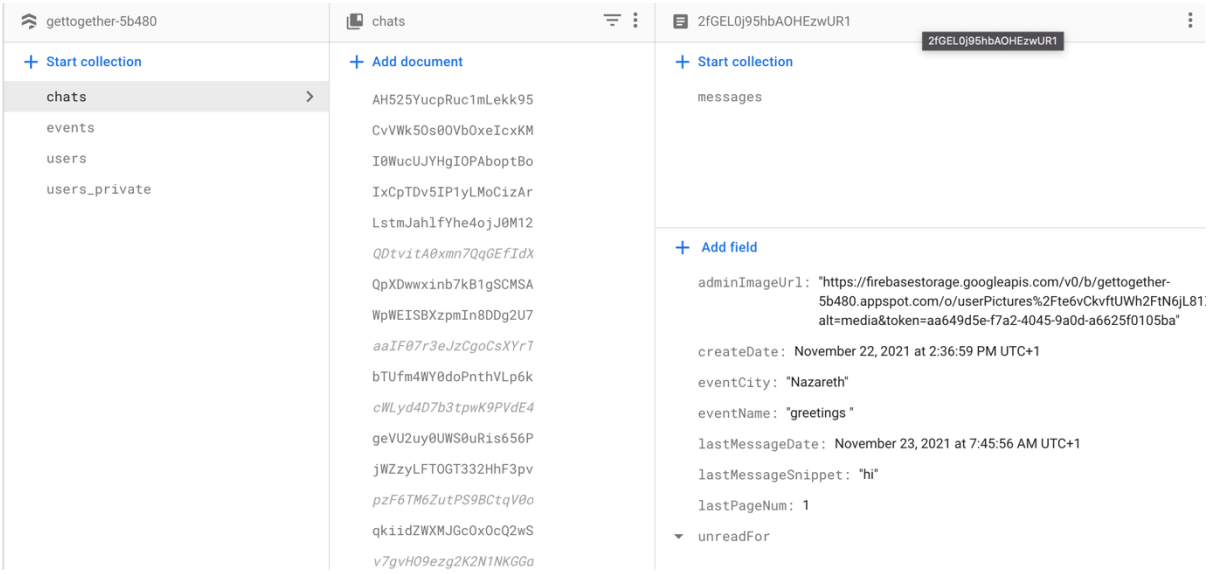
Slika 4.21. Razmjenjivanje poruka

Prvobitno se povlače svi trenutni razgovori u kojima je korisnik uključen. Nakon toga se slušaju promjene u svim razgovorima u kojima je korisnik prisutan (Programski kod 4.14.). Nakon detektiranja promjene u jednome od razgovora koji se paralelno slušaju, dobiva se model podataka o razgovoru koji se stavlja na ekran te se na osnovu trenutnog poretka razgovora i novonastale promjene u razgovoru stvara novi poredak. U tom novom poretku se organiziraju razgovori po nepročitanim i pročitanima. Nepročitani razgovori se stavljaju na vrh. Unutar tih kategorija se razgovori dodatno organiziraju po vremenu slanja.

```
final response = await getInitialChatSnippets(event.chatIds);
yield response.fold<ChatSnippetsState>((failure) {
  if (failure is NetworkFailure)
    return ChatSnippetNetworkFailure(failure.message);
  else
    return ChatSnippetServerFailure(failure.message);
}, (chatSnippets) {
  listenToChatSnippetsChange.call(this, chatSnippets);
  return ChatSnippetsLoaded(chatSnippets);
});
```

Programski kod 4.14. Dohvaćanje i slušanje promjena podataka

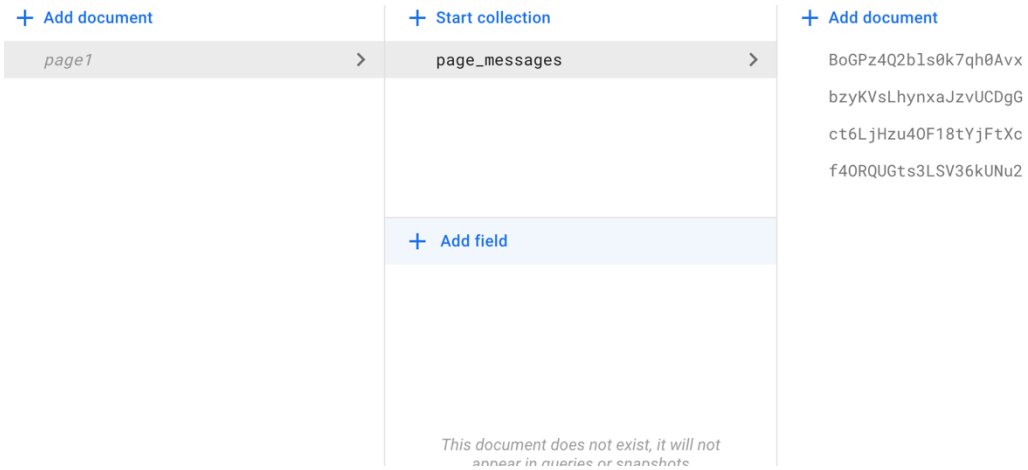
Ako korisnik klinke na jedan od razgovora prikazuje se ekran na Slici 4.21. Na ovome ekranu korisnik ima pregled svih poruka razmijenjenih u razgovoru, čak i onih prije korisnikova uključenja u razgovor. Narančastom bojom su predstavljeni oblaci s porukama drugih ljudi unutar grupnog razgovora, a bijelom bojom oblaci s porukama koje korisnik šalje. Poruke se također primaju u stvarnome vremenu pomoću *Firestore* slušanja podataka. Poruke u *Firestore* bazi podataka su organizirane po razgovorima (Slika 4.22.) i po stranicama (Slika 4.23.). Svaka stranica sadrži najmanje 10 poruka. Ako jedan razgovor ima velik broj poruka učitavanje poruka može trajati dugo vremena, pogotovo ako je loša internetska konekcija. Iz tog razloga se radi raspodjela poruka na stranice.



Slika 4.22. Organizacija razgovora u bazi podataka

Kako korisnik pomiče listu prema gore na ekranu sa Slike 4.21., dolazi do kraja učitanih poruka. Pred kraj učitanih poruka šalje se zahtjev na *Firestore* bazu podataka za dohvaćanje iduće stranice poruka. Kada korisnik pošalje poruku u razgovor trebaju se tri stvari napraviti: ažurirati pregled zadnje poruke u razgovoru, ažurirati ukupan broj stranica koje postoje za razgovor te dodati poruku u nepročitane poruke kod ostalih korisnika razgovora. Sve ovo se ne

radi s klijentske strane već se radi u sigurnom okruženju s *Cloud* funkcijama s Programskog koda 4.15. Ako se brzo šalju poruke, može se dogoditi da više od deset poruka završi u jednom razgovoru.



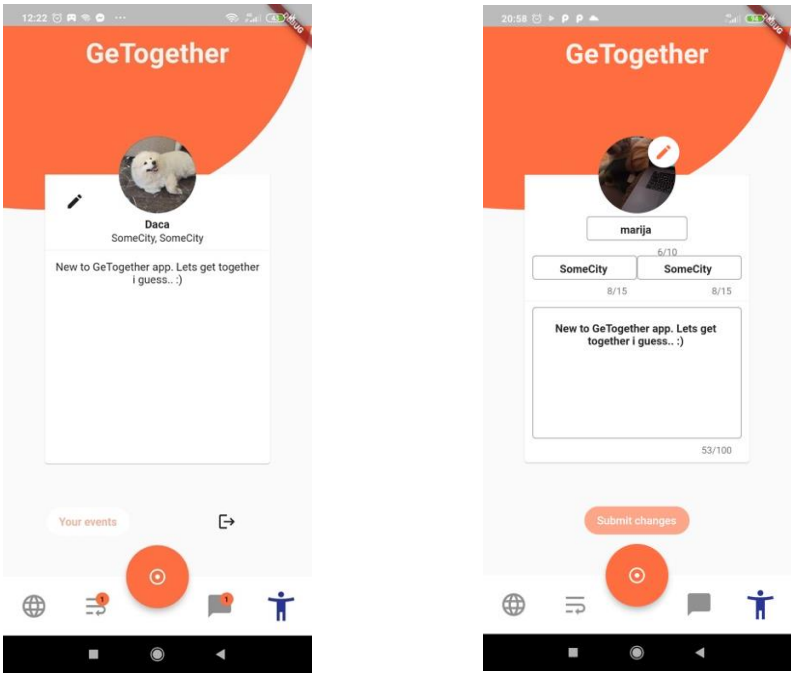
Slika 4.23. Organizacija poruka unutar razgovora

```
exports.onMessageCreated = functions.firestore
    .document("chats/{chatId}/messages/{page}/page_messages/{messageId}")
    .onCreate(async (change, context) => {
    await onMessageCreated_1.updateChatSnippet(change, context);
    await onMessageCreated_1.checkShouldAddNewPage(change, context);
    await onMessageCreated_1.addMessageToUnreadMessages(change, context);
});
```

Programski kod 4.15. *Cloud* funkcija nakon nove poruke

4.9. Profil

Korisnik ima mogućnost pregleda svog profila i događaja u kojima je korisnik prisutan. Informacije koje su dostupne korisniku na pregled su njegova slika profila, ime, grad u kojem se nalazi te opis profila. Slika se može mijenjati tako što se uslika slika pomoću kamere ili postavljanjem jedne od slika iz galerije mobitela. Broj znakova koje korisnik može unijeti prilikom promjene opisa profila je ograničen kako bi tekst stao na određenu fiksnu površinu ekrana. Također, korisnik ima opciju uređivanja svih informacija (Slika 4.24.). Na ovome ekranu se korisniku nudi i opcija prekida prijave unutar aplikacije (engl. *Log Out*). Nakon prekida prijave korisnik se vraća na ekran za registraciju i prijavu.



Slika 4.24. Pregled profila

5. ZAKLJUČAK

U ovome diplomskom radu detaljno je prikazana izrada mobilne aplikacije za organizaciju i upravljanje događajima koristeći *Flutter* razvojno okruženje. Uporaba *Flutter* razvojnog okruženja daje prednost stvaranja aplikacije na dvije mobilne platforme koristeći jedan programski kod. Posebna pažnja se predaje arhitekturi aplikacije pošto je *Flutter* novija tehnologija i ljudi posjeduju slična, ali različita rješenja implementacije arhitekture *Flutter* aplikacije. Ovom aplikacijom je demonstrirana moć *Flutter* radnog okvira i moć *Firerbase* alata na velikoj i kompleksnoj aplikaciji.

U radu je komentirana arhitektura koja se koristila pri izradi aplikacije ali i modifikacije na tu arhitekturu koje su se uočile pri i nakon izrade aplikacije. Postojeće aplikacije koje povezuju ljude su orijentirane na povezivanje ljudi preko interneta. No, u diplomskome radu se razvijala aplikacija gdje je Internet posrednik i naglasak je na komunikaciji i interakciji ljudi u stvarnome svijetu. Aplikacija time pomaže korisnicima da više koriste tehnologije kako bi više bili prisutni u stvarnome svijetu, a ne da ih tehnologija zarobljava u fiktivnom svijetu nula i jedinica.

Moguće nadogradnje na trenutnu izvedbu aplikacije su uvođenje više tipova događaja, uvođenje događaja koji se plaćaju kako bi korisnici mogli pristupiti, verifikacija korisnika sa identifikacijskim dokumentom kako bi broj stvarnih osoba koje dolaze na događaj bio jednak broju virtualnih osoba u aplikaciji, mogućnost pregleda profila drugih korisnika, mogućnost razgovora samo sa jednom osobom, ograničavanje broja osoba po događaju i dr.

6. LITERATURA

[1] All Events in City, rujan 2022. [Mrežno]. Dostupno na: https://allevents.in/

[2] Eventbrite, rujan 2022. [Mrežno]. Dostupno na: https://www.eventbrite.com/

[3] Fever, rujan 2022. [Mrežno]. Dostupno na: https://feverup.com/en

[4] Meetup, rujan 2022. [Mrežno]. Dostupno na: https://www.meetup.com/

[5] Agorify, rujan 2022. [Mrežno]. Dostupno na: https://www.meetup.com/

[6] Flutter dokumentacija, [Mrežno]. Dostupno na: https://docs.flutter.dev/

[7] „What Is The Flutter Framework“, svibanj 2021. [Mrežno]. Dostupno na: https://www.perfecto.io/blog/what-is-flutter-framework

[8] A. Velykyy, „Flutter: a full introduction to the framework“, lipanj 2020. [Mrežno]. Dostupno na: https://www.axon.dev/blog/flutter-a-full-introduction-to-the-framework#:~:text=The%20Flutter%20framework%20was%20designed,building%20block%20of%20the%20application

[9] Dart dokumentacija, [Mrežno]. Dostupno na: https://dart.dev/

[10] J. Sande, M. Galloway, „Dart Apprentice“, travanj 2021.

[11] Firebase dokumentacija, [Mrežno]. Dostupno na: https://firebase.google.com/docs

[12] F. Angleov, „Flutter Bloc library“, siječanj 2022. [Mrežno]. Dostupno na: https://pub.dev/packages/flutter_bloc

[13] Reso Coder, Flutter Bloc, rujan 2022. [Mrežno]. Dostupno na: https://resocoder.com/2020/08/04/flutter-bloc-cubit-tutorial/

[14] W. Arshad, „Managing State in Flutter Pragmatically: Discover how to adopt the best state management approach for scaling your Flutter app“, kolovoz 2021.

[15] P. Zverkov, „Flutter Clean Arhitecture“, siječanj 2020.

[16] P. Mainkar, S. Giordano, „Google Flutter Mobile Development Quick Start Guide“, ožujak 2019.

SAŽETAK

Diplomski rad se bazira na izradi mobilne aplikacije s Flutter radnim okvirom koristeći arhitekturu i principe koje se koriste u praksi. Prolaze se karakteristike pojedinih alata korištenih pri izradi aplikacije. Objašnjava se arhitektura aplikacije, arhitektura baze podataka i korištenje ostalih *Firebase* usluga. Podaci u bazi podataka se modeliraju prema izgledu ekrana. Kombiniraju se arhitekture aplikacije, baze podataka i ostalih vanjskih javnih sučelja korišteni pri izradi aplikacije. Analizira se arhitektura provedena u aplikaciji i komutiraju se moguće promjene koje su primijećene prilikom izrade velike i kompleksne aplikacije kao što je ova. Aplikacija je generalno podijeljena na funkcionalnosti: prijava i registracija, pregled događaja, pregled obavijesti, stvaranje događaja, pregled korisnikovog profila, pregled grupnih razgovora te komunikacija unutar tih razgovora. Cilj aplikacije je prikazivanje događaja korisnicima u njihovom gradu ili selu u kojem se trenutno nalaze. Korisnik se pridružuje događaju tako što pošalje zahtjev administratoru događaja koji je obaviješten putem notifikacija unutar aplikacije. Nakon što administrator prihvati poziv za pridruživanje događaju, korisnik se pridružuje grupnome razgovoru u kojem sudjeluju svi korisnici događaja. Svaki korisnik posjeduje profil koji ga jednoznačno određuje u aplikaciji. Bilo koji korisnik može napraviti novi događaj. Prilikom stvaranja novog događaja lokacija je ograničena unutar radijusa od pet kilometara. Svaki ekran aplikacije je analiziran i povezan s implementacijom u aplikaciji i bazi podataka.

Ključne riječi: arhitekture, bloc, cubt, događaji, flutter, poruke, upravljanje stanjem

ABSTRACT

Mobile application for organization and management of events master's thesis is based on building a Flutter mobile application using architecture and concepts that are used in professional practice. Characteristics of individual tools used for developing the application are commented on. Architecture of the application is explained, architecture of Firestore database and usage of other Firebase tools is explained. Data inside the Firestore database is modeled with screen look in mind. Architecture of the application, database and other public interfaces are combined while making the application. Architecture of the application is analyzed and possible variations to a big and complex app like this one is explained. Application is generally separated into a few features: authentication, events overview, notifications overview, user profile overview, group chats overview and communication in those group chats. Goal of the application is to show users events that are in their city or village and meeting with people from that event. User can join an event by sending a join request to the event administrator. After the administrator accepts the request for joining the event, user enters group chat of the event. Every user has a profile that identifies them in the app. When creating a new event the location is limited to five kilometers around the users current location. Every screen is analysed and connected with its implementation in the application and in the database.

Key words: Arhitecture, Bloc, Cubit, Dart, Events, Flutter, Firebase, Messaging, State managment

ŽIVOTOPIS

Davor Štajcer rođen je 26. prosinca 1998. Godine u Zagrebu. Osnovnu školu je upisao i završio u Slavonskom Brodu. Nakon završetka osnovne škole upisuje srednju Tehničku školu u Slavonskom Brodu, smjer elektrotehničar, koju završava 2017. godine te u istoj godini upisuje preddiplomski studij elektrotehnike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Preddiplomski studij završava 2020. godine i iste godine upisuje diplomski studij na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Drugi semestar provodi na razmjeni studenata u Turskoj, grad Konya. Nakon završetka prve godine diplomskog studija se zapošljava u firmi SPIN d.o.o. gdje razvija mobilne i mrežne aplikacije u Ionic radnom okviru. Nakon dva mjeseca prelazi u agenciju COBE d.o.o.

_____

Davor Štajcer