Mobilna aplikacija za upravljanje događajima

Puhanić, Antonio

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

Sveučilišni studij

Mobilna aplikacija za upravljanje događajima

Diplomski rad

Antonio Puhanić

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

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

Ocjena diplomskog rada na sveučilišnom diplomskom studiju

Ime i prezime pristupnika:	Antonio Puhanić
Studij, smjer:	Sveučilišni diplomski studij Računarstvo
Mat. br. pristupnika, god.	D1319R, 07.10.2022.
JMBAG:	0069085681
Mentor:	izv. prof. dr. sc. Josip Balen
Sumentor:	Matej Arlović, univ. mag. ing. comp.
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 1:	izv. prof. dr. sc. Josip Balen
Član Povjerenstva 2:	Davor Damjanović, univ. mag. ing. comp.
Naslov diplomskog rada:	Mobilna aplikacija za upravljanje događajima
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskm dijelu rada potrebno je proučiti mobilne aplikacije i sustave za upravljanje događajima te opisati korištene tehnologije za izradu mobilnih i web aplikacija. U praktičnom dijelu rada potrebno je koristeći Fluttter razvojno okruženje razviti mobilnu aplikaciju za upravljanje različitim događajima (poput konferencija i skupova) koje će imati sljedeće funkcionalnosti: registracija i prijava, upravljanje događajima, komunikacija između sudionika, prikaz lokacije, dodavanje slika i sl. "Tema rezervirana za: Antonio Puhanić"
Datum ocjene pismenog dijela diplomskog rada od strane mentora:	09.09.2024.
Ocjena pismenog dijela diplomskog rada od strane mentora:	Izvrstan (5)
Datum obrane diplomskog rada:	18.09.2024.
Ocjena usmenog dijela diplomskog rada (obrane):	Izvrstan (5)
Ukupna ocjena diplomskog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije diplomskog rada čime je pristupnik završio sveučilišni diplomski studij:	19.09.2024.

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

IZJAVA O IZVORNOSTI RADA

Osijek, 19.09.2024.

Ime i prezime Pristupnika:	Antonio Puhanić
Studij:	Sveučilišni diplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D1319R, 07.10.2022.
Turnitin podudaranje [%]:	3

Ovom izjavom izjavljujem da je rad pod nazivom**: Mobilna aplikacija za upravljanje događajima**

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

i sumentora Matej Arlović, univ. mag. ing. comp.

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

Potpis pristupnika:

SADRŽAJ

1. UVOD

Za potrebe učinkovitog pregledavanja informacija o različitim događajima vrlo korisna može biti aplikacija koja omogućuje funkcionalnosti za lakše praćenje i pregledavanje događaja. U ovome projektu izrađena je mobilna aplikacija za praćenje i upravljanje događajima. Fokus je na IEEE SST konferenciju gdje je aplikacija izrađena i prilagođena u svrhe te konferencije, ali također i postavljena tako da se može prilagoditi za ostale događaje u budućnosti.

Cilj je napraviti aplikaciju s kojom korisnik ima uvid u događanja tako da ih može pretraživati, filtrirati i označiti da ide na njih. Uz popis događaja također aplikacija ima sučelje koje sadrži dodatne informacije i promjene vezano uz konferenciju. Za potrebe dodatnih pitanja postoji sučelje koje je u obliku foruma gdje korisnici koji idu na konferenciju mogu postaviti pitanje i bilo tko može odgovoriti. Također ima mogućnost međusobnog dopisivanja korisnika.

Aplikacije poput ove pružaju izuzetnu korisnost sudionicima konferencija i događanja jer omogućuju brži i jednostavniji pristup informacijama, te poboljšavaju organizaciju i planiranje prisustvovanja različitim sesijama i radionicama. Korištenjem mobilne aplikacije, korisnici mogu imati sve potrebne informacije na dohvat ruke, što je posebno važno u dinamičnim okruženjima gdje se rasporedi i lokacije mogu brzo mijenjati.

Razvoj aplikacije počinje s detaljnom analizom zahtjeva i definiranjem funkcionalnosti koje aplikacija treba sadržavati. Nakon toga slijedi faza dizajna, gdje se pažnja posvećuje stvaranju intuitivnog i privlačnog korisničkog sučelja. Zatim slijedi faza implementiranja i kodiranja, te nakon toga faza testiranja i održavanja aplikacije.

1.1. Zadatak diplomskog rada

Zadatak ovog diplomskog rada je istražiti slične postojeće mobilne aplikacije za upravljanje događajima, te ih usporediti, vidjeti prednosti i nedostatke, utvrditi što je potrebno za IEEE SST konferenciju. Na osnovu navedenog istraživanja je potrebno postaviti zahtjeve na mobilnu aplikaciju. Od zahtjeva aplikacija mora imati mogućnost registracije i prijave, pregled događaja, pretraživanje i filtriranje događaja, mogućnost dopisivanja i postavljanja pitanja, objavljivanja informacija, sučelje za administratore u svrhu upravljanja događajima, objavama i sponzorima. Mobilna aplikacija je implementirana koristeći Flutter razvojno okruženje i Supabase backend kao uslugu.

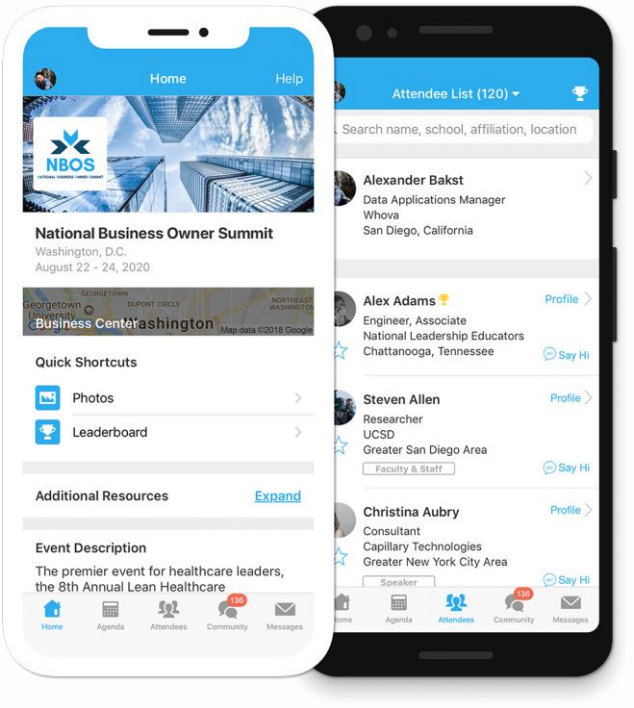
2. PREGLED PROGRAMSKIH RJEŠENJA

Aplikacije ovakvog tipa služe kako bi pomogle korisnicima jednostavno i brzo pronaći željene događaje i informacije o tim događajima. Često se koriste kada se događaju konferencije ili slične vrste događaja koje traju duže vrijeme, najčešće nekoliko dana. Sastoje se od više kraćih događanja gdje u ovom slučaju su to predavanja vezana uz IEEE SST(*International Conference on Smart Systems and Techonologies*) gdje su svake godine definirane različiti setovi tema na osnovu kojih će se održavati ta predavanja. Najvažnija funkcionalnost koju ovakva aplikacija treba pružati je agenda. Događaji su unaprijed definirani, te na aplikaciju događaje postavljaju administratori putem sučelja za dodavanje događaja. Bitno je imati zaslon za agendu koji je pregledan i pruža jednostavan način pretraživanja i filtriranja događaja prema datumu, imenu, organizatoru i opisu. Stime korisnik može jednostavno vidjeti cjelokupan plan i program te označiti si na što želi ići.

2.1. Pregled postojećih rješenja mobilnih aplikacija

Postoji nekolicina aplikacija za upravljanje događajima koje se zasnivaju na konceptu pregleda agende. Postoji nekoliko ključnih sličnosti koje definiraju ovakav tip aplikacije, ali se i razlikuju po namjeni, izgledu i načinu izrade.

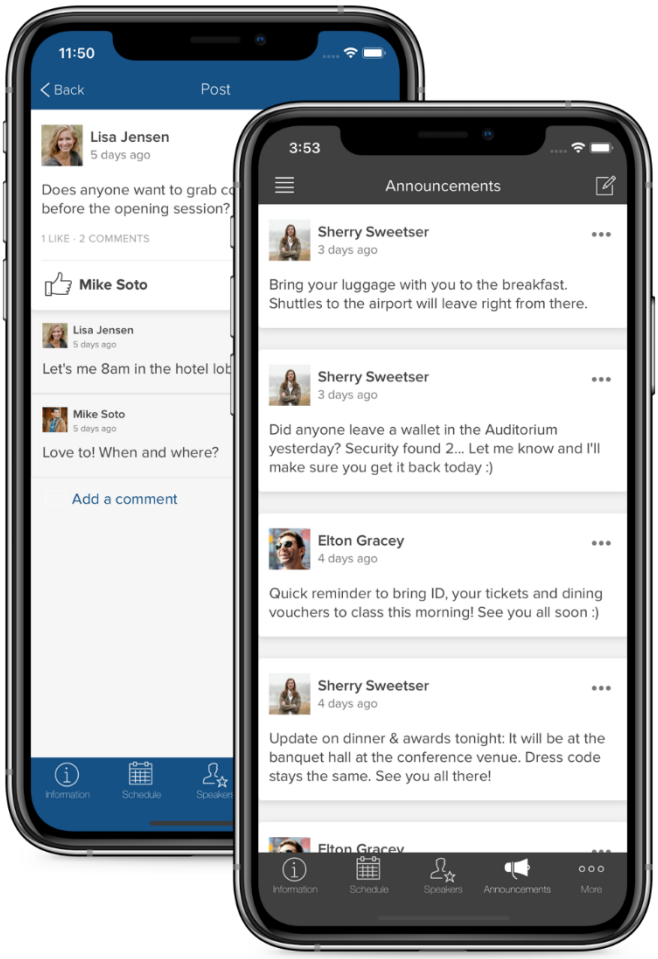
Na slici 2.1 se nalazi Whova mobilna aplikacija koja omogućava pregled događaja za konferencije koja koristi vrlo jednostavno i brzo sučelje, služiti će kao inspiracija za zaslon agende na ovoj aplikaciji. Jednostavnost sučelja je ujedno i velika prednost ove aplikacije [1].



Slika 2.1. *Prikaz sučelja Whova aplikacije [1]*

Ova aplikacija također ima fokus na promoviranje događaja i sakupljanje bodova koji služe kao manja igrica unutar aplikacije. Za potrebe aplikacije ovog diplomskog rada takve funkcionalnosti nisu potrebne. Sadržava tako sučelje za obavijesti i postavljanja pitanja što je bitno i za IEEE SST konferenciju te će se to implementirati u ovoj aplikaciji, ali na drugačiji način koristeći modernije korisničko sučelje.

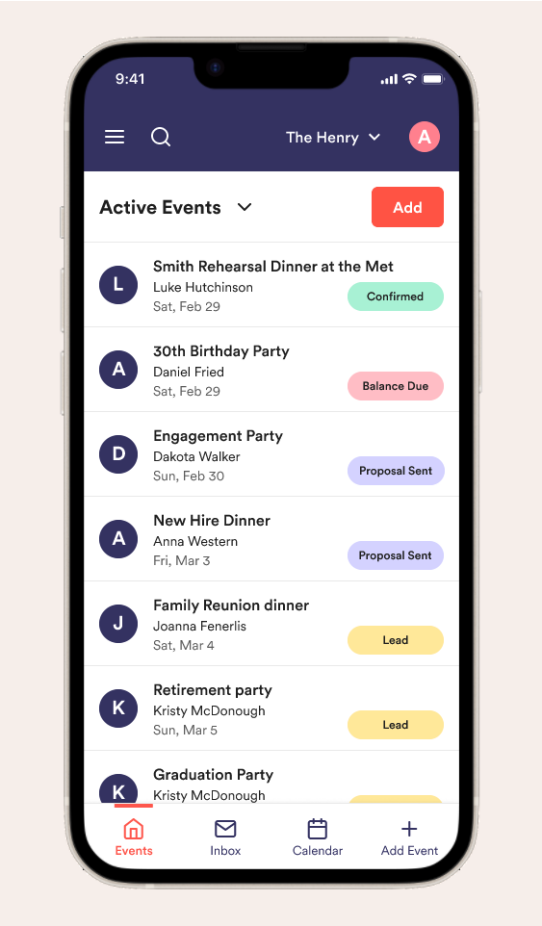
Yapp je također jedna od poznatijih aplikacija za upravljanje događajima [2]. Kao i ostale aplikacije sadrži agendu i popis registriranih ljudi, ali se fokusira puno više na objavljivanje slika u obliku objava današnjih društvenih mreža, kao što se može vidjeti na slici 2.2.



***Slika 2.2**. Prikaz sučelja Yapp aplikacije [2]*

Yapp ima svoju glavnu aplikaciju koja je napravljena za generalno upravljanje događajima, ali također pružaju izradu aplikacije prema specifičnim zahtjevima i potrebama korisnika i događaja. Na taj način je aplikacija ovog rada biti prvobitno napravljena, s ciljem da se prilagodi potrebama izvođenja IEEE SST konferencije.

Perfect Venue [3] je također jedna od poznatijih rješenja za upravljanje događajima. Funkcionira na sličan način kao ostale aplikacije, ali se razlikuje u tome što se fokusira na sučelje kalendara. Omogućuje integraciju postojećih aplikacija sa kalendarom, korisnik može odabrati događaj na koji planira doći te ga spremiti u svoj vlastiti kalendar ili u kalendar unutar Perfect Venue aplikacije. Na slici 2.3. se može vidjeti primjer sučelja u kojemu se upravlja događajima.



Slika 2.3. *Prikaz sučelja Perfect Venue aplikacije [3]*

2.2 Zahtjevi na programsko rješenje

Cilj je imati jednostavnu aplikaciju koja koristi moderne tehnike za bolje korisničko iskustvo kako bi se implementirala aplikacija koja pruža jednostavno rukovanje agendom. Koristi se Flutter razvojno okruženje za stvaranje korisničkog sučelja, te Supabase platforma za razvoj backend usluga.

Kroz prethodno navedene primjere mobilnih rješenja može se primijetiti da uvijek postoji implementacija agende, popis ljudi koji sudjeluju i sučelje koje na različite načine prikazuje dodatne informacije. To su ujedno i glavne funkcionalnosti koje treba ostvariti za potrebe ove aplikacije. Također za potrebe ove konferencije potrebno je implementirati sučelje za dodavanje dokumenata, sučelje za dodavanje i upravljanje sponzorima i sučelje za upravljanje ulogama korisnika. Potrebno je imati korisnike koji su administratori. Administratori imaju mogućnost upravljanja svim podatcima i događajima vezano za konferenciju. To uključuje kreiranje, brisanje i uređivanje podataka.

Funkcionalni zahtjevi koje aplikacija treba ispuniti su:

1. Registracija i prijava korisnika
2. Mogućnost prijave i registracije pomoću Google-a
3. Stvaranje, brisanje, uređivanje događaja
4. Stvaranje, brisanje, uređivanje sponzora,
5. Stvaranje, brisanje, uređivanje obavijesti
6. Pregledavanje, pretraživanje, sortiranje i filtriranje događaja
7. Označavanje da korisnik će ići na određeni događaj
8. Prikaz detalja o događaju
9. Prikaz sponzora
10. Prikaz obavijesti
11. Postavljanje pitanja i komentara
12. Prikaz svih polaznika
13. Postavljanje i preuzimanje dokumenata.
14. Mijenjanje uloga korisnika

Nefunkcionalni zahtjevi koje aplikacija mora ispuniti:

1. Trajno spremanje podataka događanja, korisnika, sponzora i dokumenata
2. Jednostavan postupak registracije i prijave
3. Jasno i pregledno sučelje s jednostavnim funkcionalnostima i gestama
4. Učitavanje podataka treba trajati do jedne sekunde
5. Učitavanje registracije i prijave treba trajati do jedne sekunde
6. Aplikacija treba biti skalabilna kako bi podržala buduća proširenja i veći broj korisnika
7. Treba biti dostupna preko 99.9% vremena s minimalnim prekidima rada
8. Treba biti prilagodljiva za različite veličine zaslona

3. KORIŠTENE TEHNOLOGIJE ZA IZRADU APLIKACIJE

Za izradu same aplikacije na mobilnom uređaju koristi se Flutter razvojno okruženje uz Dart programski jezik. Za izradu poslužiteljskog dijela aplikacije koristi se Supabase platforma za razvijanje. Glavni razlog zašto se koristi Flutter je jednostavnost i alate koje pruža za izradu aplikacija koje mogu se pokrenuti na više platformi što uključuje web aplikacije, desktop aplikacije, IOS i Android aplikacije s jednim izvornim kodom.

3.1. Flutter

Flutter je razvojno okruženje koje je razvio Google. Glavna namjena je izrada mobilnih aplikacija, ali također omogućuje izradu aplikacija za sve web preglednike i desktop aplikacije na Linuxu, Windows i MacOS operacijskim sustavima [4].

Zasniva se na konceptu *widgeta* gdje svaki *widget* predstavlja gradivni element aplikacije. *Widgeti* mogu sadržavati unutar sebe više *widgeta* te se pomoću stabla tih elemenata se iscrtavaju na zaslonu. Mogu biti *stateless* ili *statefull*. *Stateless widgeti* nemaju definirano stanje, tj. jednom kada su izrađeni ostaju nepromijenjeni za vrijeme svojega trajanja. Jednom kada su učitani, osim ako se ne dogodi neki vanjski događaj koji djeluje na taj *widget,* ostaje nepromijenjen. *Statefull widget* jer strukturiran drugačije te ima svoje promjenjivo unutarnje stanje. Kada se to stanje promjeni taj *widget* se ponovo iscrtava na zaslon te se mijenja dinamički ovisno o tome kako je napravljen da radi.

U Flutteru izrada sučelja se zasniva na nekoliko glavnih *widgeta* poput r*ow, column, center, text, scaffhold* i sl. Za pozicioniranje elemenata na zaslon se uglavnom postiže korištenjem i slaganjem *column* i *row widgeta* s kojima se definiraju kako su elementi aplikacije pozicionirani. Za prikazivanje ostalih vizualnih elemenata koriste se mnogi već predefinirani *widgeti* unutar Fluttera.

Aplikacija ima glavnu polaznu točku u *main()* funkciji u kojoj se konfigurira i definiraju akcije koje je potrebno izvršiti prije pokretanje same aplikacije. U *main()* se nalazi *runApp() funkcija* s kojom se pokreće aplikacija. Ta funkcija odmah prilikom pokretanja prima prvi *widget* koji predstavlja početni zaslon aplikacije. Taj *widget* , kao i svaki drugi, ima svoju *build()* metodu koja se koristi kako bi se prikazao sadržaj na zaslon.

Flutter također podržava instalaciju dodatnih paketa u projekt. Postoje tisuće različitih paketa koji pružaju razne funkcionalnosti od izgradnje korisničkog sučelja do mnoštvo vanjskih funkcionalnosti. Najvažniji paketi za ovaj projekt su *Bloc*, *getIt, freezed* i *supabase_flutter*. Način

funkcioniranja svakog paketa će biti objašnjen u budućim poglavljima u kojima će se vidjeti i implementacija istih.

3.2. Dart

Dart je objektno orijentirani programski jezik namijenjen za razvijanje aplikacija na bilo kojoj platformi [5]. Dart se koristi kao temelj za Flutter razvojno okruženje, koje primarno omogućava razvoj nativnih aplikacija za iOS i Android iz jednog koda.

Jedna od značajki je da je „*type-safe*" što znači da vrijednost varijabli uvijek se podudara sa tipom varijable. Iako Dart ne zahtijeva strogo definiranje tipa varijabli pri deklaraciji, koristi statičku analizu kako bi se osiguralo da vrijednosti koje se dodjeljuju varijablama odgovaraju njihovom tipu. Također ima ugrađenu značajku za sigurnost pri „*null error*" greškama, što znači da varijable ne smiju biti prazne osim ako u suprotnom varijabla nije označena da smije biti prazna. Cilj provođenja toga je da se smanji broj mogućih grešaka koji se mogu dogoditi prilikom izvođenja samog programa te da se olakša traženje i ispravljanje pogrešaka.

Dart ima veliki broj ugrađenih biblioteka koje sadrže razne funkcije koje se često koriste te olakšavaju analizu, izradu, izvedbu i prepravak koda. Neki primjeri biblioteka su „*dart:collection*" što pruža implementaciju povezanih listi, binarnih stabla i sl., „*dart:async*" što pruža podršku za rad s asinkronim funkcijama te mnogi drugi.

Dart ima *Just-In-Time* (JIT) kompajler za brzi razvoj i testiranje, te *Ahead-Of-Time* (AOT) kompilaciju za optimalne performanse u produkciji. Dart je dizajniran za skalabilnost, omogućavajući razvoj malih skripti kao i velikih, složenih aplikacija.

3.3. Supabase

Supabase je *BaaS* (engl. *Backend as a Service*) platforma za razvoj poslužiteljskih usluga i baza podatka. Pruža spektar funkcionalnosti uključujući bazu podataka, autentifikaciju, spremanje datoteka, *AI*, slušanje promjena u stvarnom vremenu i RPC funkcije [6]. Komunikacija između klijenta i Supabase poslužitelja se može implementirati uz pomoć *RestAPI* ili *GraphQL* protokola. Supabase koristi relacijsku bazu podataka koja podržava izvođenje PostgreSQL i SQL koda na samoj web stranici tj. poslužitelju. Podržava složene upite, indeksiranje, transakcije te visoku sigurnost i dostupnost podataka.

Supabase je povezan sa nekoliko različitih razvojnih okruženja od kojih jedan je Flutter. Postoji poseban „*flutter_supabase*" paket koji omogućuje povezivanje aplikacije sa bazom podataka te

ostale funkcionalnosti potrebne kako bi se ostvarila i olakšala komunikacija između klijenta i poslužitelja. Paket sadržava *Supabase.instance.client* što je instanca klijenta koji omogućuje funkcionalnosti i komunikaciju sa poslužiteljem.

Jedna od bitnih značajki *Supabase* platforme je što pruža sveobuhvatan sustav za rukovanje autentifikacijom korisnika i održavanje sesija. To uključuje prijavu, registraciju, resetiranje lozinki i prijave preko dodatnih vanjskih servisa pomoću *OAuth*.

Također podržava RPC (*Remote Call Procedure*) funkcije koje olakšavaju implementaciju složenih funkcija na serveru što omogućuje da prilikom poziva sa klijentske stranice se dostave potrebni podatci koji olakšavaju daljnju implementaciju i korištenje. RPC su funkcije koje poziva klijent, ali se izvršavaju na strani poslužitelja. Kada poslužitelj završi sa izvođenjem funkcije, rezultat se vrača nazad klijentu.

3.3 Upravljanje stanjima i BLoC

Upravljanje stanjima (engl. *State Managment)* je bitan koncept unutar svake Flutter aplikacije. Kao što ime nalaže to predstavlja način kako se upravlja stanjima unutar aplikacije i njenih pojedinačnih elemenata. Stanja definiraju što i kako će se prikazivati na zaslonu, kako će se pojedinačni elementi ponašati te koji će se podatci slati i primati preko API poziva.

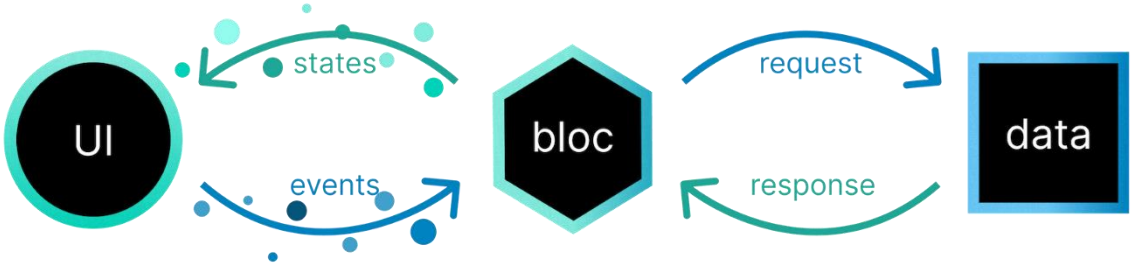
Flutter ima nekoliko načina upravljanjem stanjima unutar aplikacije poput korištenja *setState* metode unutar specifičnih *widgeta*, korištenjem *Change Notifier*, *Value Notifier* klasa i *InheritedWidget widgeta* [7]. Metoda *setState* je namijenjena tome da upravlja stanjem unutar pojedinačnog *widgeta*, dok ostali načini služe tome da pružaju stanje kroz više elemenata aplikacije.

Iako Flutter pruža sve što je potrebno kako bi se upravljalo stanjima i podatcima, često se stvara problem u tome što je potrebno pisati puno „*boilerplate"* koda tj. koda koji se puno ponavlja i najčešće nije jednostavno takav kod održavati i proširivati. Stoga postoje vanjska rješenja koji olakšavaju upravljanje stanjima kroz pakete. Neki od popularnijih paketa su *Riverpod, BLoC, Provider*. Takvi paketi najčešće pružaju strukturiran i skalabilan način za upravljanje stanjima unutar aplikacije. Funkcioniraju tako što se proširuju na interne mehanike Flutter okruženja te pružaju sučelje i funkcionalnosti koje olakšavaju cijeli proces pisanja koda i upravljanja stanjima.

Jedan od paketa koji se koristi za aplikaciju u ovom radu je *BLoC* (engl. *Business Logic Component*). Primarna svrha ovoga paketa je upravljanje stanjima tako da odvoji korisničko

sučelje tj. prezentacijski sloj od sloja poslovne logike. Paket daje apstrakciju i funkcionalnost kako bi se to postiglo.

Uz sliku 3.1. može se vidjeti glavni princip kako *Bloc* arhitektura funkcionira. *Bloc* se nalazi između podataka i korisničkog sučelja. Sadrži sva stanja i rukuje svim povezanim događajima koje se mogu dogoditi kroz aplikaciju [8].



Slika 3.1. *Službeni prikaz načina funkcioniranja BLoC paketa [8]*

Svaki *Bloc* je predstavljen klasom koja reagira na promjene aplikacije te emitira svoje stanje ovisno o promjenama. Tijek odvijanja aplikacije se sastoji od toga da kada se dogodi neki događaj, *Bloc* reagira na taj događaj te ovisno o tome obavlja određene operacije i funkcije koje je potrebno kako bi prikupilo ili promijenilo podatke i emitiralo svoje novo stanje na koje se onda sučelje promijeni.

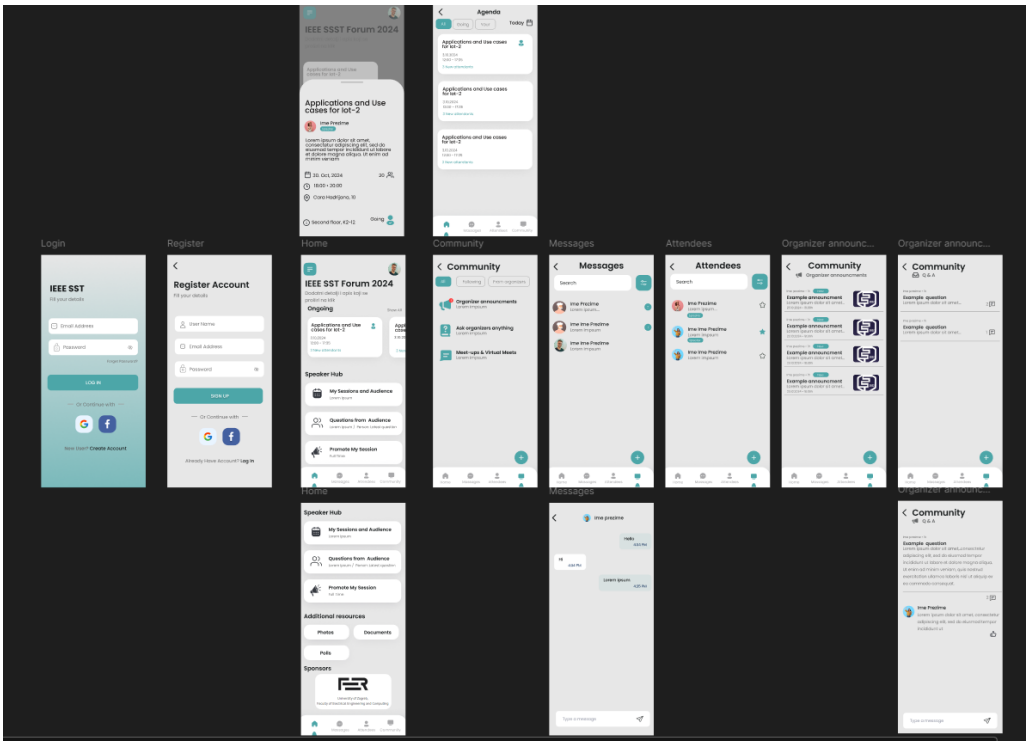
Bloc se sastoji od unaprijed definiranih klasa događaja tj. (engl. *Events)* i klasa stanja (engl. *States).* Događaji i stanja se objedinjuju u svojoj *Bloc* klasi gdje se onda za svaki događaj definira unutarnja metoda koja rukuje time i emitira stanje cijelog *Bloca*. Važna razlika kod ovog pristupa je što se stanja emitiraju umjesto da im se direktno pristupi pozivanjem određenim varijablama. Stanje *Bloca* ostatak aplikacije sluša i reagira kada god se dogodi promjena.

Također postoji *Cubit* klasa što je pojednostavljena verzija *Bloc* klase. Razlikuje se po tome što nema definirane događaje. Ne mogu se pratiti prijelazi stanja, nego umjesto klase za događaje koriste klasične funkcije da emitira stanje *Cubita*. Najčešće se koristi za upravljanje manjim podatcima poput teksta unutar formi, *boolean* vrijednosti varijabli i slično.

4. IZRADA APLIKACIJE

U ovom poglavlju će biti objašnjeno kako je aplikacija izrađena, načini na koje su se navedene tehnologije koristile, te će se pokazati struktura aplikacije. Svrha aplikacije je korisnicima olakšati i pomoći pri pregledu događaja i ostalih stvari vezano za konferenciju. Postoji mnogo tehnika s kojima se može to postići. Također je bitno osim intuitivnog korisničkog sučelja pružiti jednostavnost prilikom registracije i prijave. Uz klasični način registracije i prijave preko e-maila također je potrebno implementirati registraciju i prijavu pomoću Google i Facebook OAuth autentifikacije. To je funkcionalnost koja znatno pojednostavljuje cjelokupan proces registracije i pomaže korisniku da prilikom korištenja aplikacije brže dođe to potrebnih informacija.

Za aplikaciju je prvobitno izrađen *mock-up* dizajn u *Figmi* koji se može vidjeti na slici 4.1.



Slika 4.1. *Prvobitni dizajn za sučelje aplikacije*

Aplikacija se dijeli na tri glavna dijela:

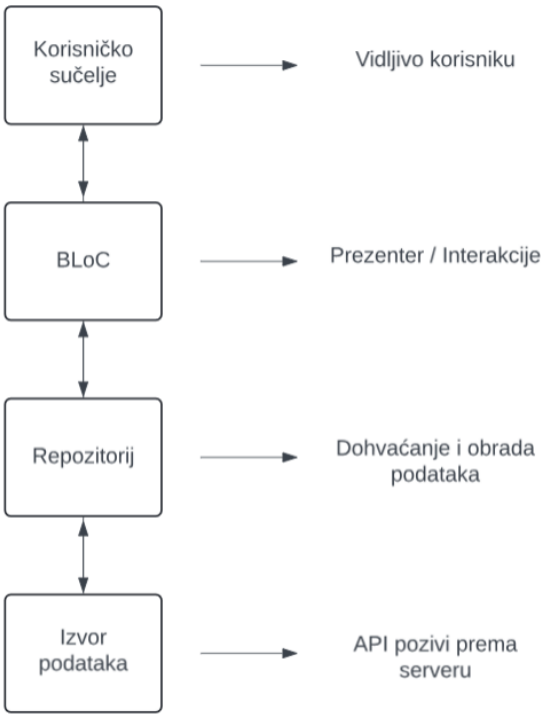
1. Korisničko sučelje
2. Sučelje za administratora
3. Sučelje za prijavu i registraciju

Ovaj način podjele je bitan jer utječe na način kako će se zasloni ugnježđivati te kako će se korisnik koji se prijavljuje u aplikaciju usmjeriti na prikladan zaslon ovisno o tome je li administrator ili ne. Prilikom instalacije i otvaranje aplikacije prvo što korisnik vidi je zaslon za prijavu. Ako

korisnik već ima registriran račun, treba samo unijeti svoje podatke za prijavu te ovisno o tome je li administrator ili klasični korisnik, biti će preusmjeren na prikladno sučelje. U slučaju da korisnik nema račun može izabrati jednu od opcija za registraciju. Supabase omogućava da prilikom završetka registracije korisnik može odmah koristiti aplikaciju bez dodatnog koraka ručnog prijavljivanja ili potvrđivanja email adrese.

4.1. Struktura aplikacije

Projekt ima oblik *Bloc (engl.* Business Logic Component*)* strukturne arhitekture što je ujedno i ime paketa koji se koristi kako bi se ova arhitektura realizirala. To je arhitektura koja odvaja sučelje aplikacije od poslovne logike s ciljem da se olakša proširenjima koda, testiranjem i da se postigne bolja iskoristivost koda. Na slici 4.2 se može vidjeti oblik ove arhitekture.



Slika 4.2. *Prikaz arhitekture koristeći BLoC*

To je obrazac za arhitekturu koji se više puta koristi u aplikaciji za različite setove podataka. Na prethodnom dijagramu se može vidjeti način kako je postavljeno upravljanje strukturom aplikacije.

Aplikacija započinje s interakcijom korisnika. Ovisno o tome što korisnik napravi potrebno je prikazivati elemente na sučelju dinamično s time što se događa. Npr. ako korisnik stisne na gumb koji dohvaća podatke, prvo je potrebno je započeti taj proces u pozadini te za vrijeme tog procesa prikazati grafički da se podatci učitavaju. Dio aplikacije koji se brine za taj proces je BLoC.

Repozitoriji su važan dio dohvaćanja podataka. Primarna svrha repozitorija je odrediti prvo je li potrebno dohvatiti podatke lokalno ili tražiti podatke iz vanjskog izvora. Nakon što je to određeno, repozitorij dohvaća podatke koje onda treba prilagoditi u potreban model koji će se proslijediti nazad na *Bloc* komponentu kako bi se time moglo rukovati i izgraditi potrebno sučelje za korisnika.

Izvor podataka predstavlja komunikaciju sa poslužiteljem i bazom podataka preko API-ja. U ovom slučaju je to *Supabase* platforma. Podatci najčešće dolaze u JSON obliku, te je stoga potrebno kada podatci dođu iz izvora podataka na klijentsku stranu da repozitorij pretvori JSON podatke u prikladne objekte kako bi ti podatci pravilno propagirali kroz ostatak aplikacije.

4.2. Struktura direktorija

Kako bi projekt bio proširiv i jednostavan za održavanje, uz kvalitetan kod bitno je imati i dobro postavljenu strukturu direktorija. Prilikom generiranja Flutter aplikacije postoji nekoliko početnih direktorija. To su direktoriji koji sadrže kod kako bi se aplikacija mogla pokrenuti na različitim platformama, te uz njih najvažniji direktorij pod nazivom „*lib*“ u koji se piše kod za samu aplikaciju [9]. Postoje dva najčešća Flutter standarda prilikom razvijanja aplikacije i održavanja strukture direktorija, a to su direktoriji s fokusom na slojeve domene i direktoriji s fokusom na značajke.

U ovom radu se koristi struktura gdje će se aplikacija razdijeliti na domene tj. slojeve, a to su prezentacijski sloj, podatkovni sloj i sloj domene. Prezentacijski sloj sadrži sve zaslone, BLoC-ove i widget-e koji služe za prikaz sučelja korisniku. Sloj domene služi za definiranje sučelja, modela i modula. Sloj podataka sadrži konstante, izvore podataka, implementacije sučelja koji su definirani u sloju domene i ostale funkcionalnosti i podatke potrebne za aplikaciju.

Za svaku aplikaciju struktura direktorija će uvijek biti drugačija jer svaka aplikacija je drugačija i različiti programeri koriste drugačiji pristup prilikom izrade aplikacije. Ne postoji univerzalno rješenje koje se koristi za svaku aplikaciju. Ovi standardi služe kao smjernice prilikom izrade aplikacije.

4.3. Postavljanje Flutter aplikacije

Kako bi bilo uopće moguće uspostaviti komunikaciju potrebno je spojiti aplikaciju sa Supabase poslužiteljem. Potrebno je imati poveznicu i API ključ koji se generiraju na Supabase stranici prilikom izrade baze podataka. Budući da su to osjetljive informacije, radi sigurnosti ti podatci se

spremaju u *„.env“* datoteku unutar projekta. Za to se koristi paket za čitanje varijabli okruženja i paket za Supabase prije pozivanje funkcije za pokretanje aplikacije gdje se postavlja kod koji omogućava spajanje na taj server. U programskom kodu 4.1. se može vidjeti način na koji su se prethodno spomenuti paketi koristili.

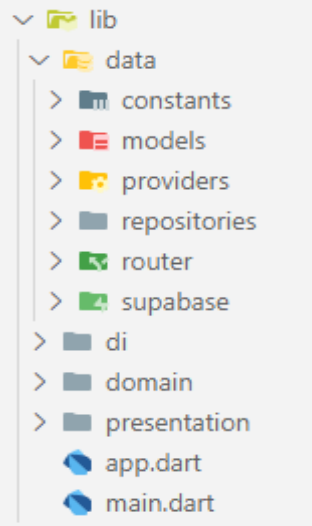
```
await dotenv.load();
await Supabase.initialize(
  url: dotenv.env['URL']!,
  anonKey: dotenv.env['API_KEY']!,
);
```

Programski kod 4.1. *Prikaz koda za spajanje na server*

Također je potrebno postaviti direktorije na osnovu već prethodno objašnjenih slojeva. Na slici 4.3. mogu se vidjeti korišteni direktoriji. Pod direktorijem *„constants“* nalaze se klase koje sadrže varijable s unaprijed definiranim vrijednostima poput putanji aplikacije, stilovima za tekst i boje aplikacije.

Direktorij *models* sadrži klase svih korištenih modela unutar aplikacije što uključuje modele za prikaz podataka poput događaja i objava, ali također uz to modele za unos teksta i slično. Direktorij *repositories* i *supabase* sadrži sve repozitorije i izvore podataka.

Također se može vidjeti i *„di“* direktorij koji služi za ubrizgivanje ovisnosti za cijelu aplikaciju. Sadrži dvije datoteke *dependency_injection.dart“* i *„dependency_injection.config.dart“*. To su datoteke koje služe kao bi se upravljalo instancama klasa kroz cijelu aplikaciju.



Slika 4.3. *Prikaz strukture direktorija aplikacije*

Uz paket *getIt* označavaju se klase sa anotacijama poput *injectable* ili *singleton*, nakon što su klase označene *getIt* paket se brine o instancama tih klasa te pruža njihovo instanciranje.

Kada se klasa označi sa prikladnom anotacijom, paket *build_runner* se aktivira te generira kod za upravljanje instancama tih klasa. U programskom kodu 4.2. se može vidjeti jedan primjer u kojem se koristi anotacija za stvaranje klase koristeći *singleton* obrazac.

```
@LazySingleton()
class SupabaseEventRepository {
  final SupabaseEventApi _supabaseApi;

  SupabaseEventRepository(this._supabaseApi);
```

Programski kod 4.2. *Primjer označavanja klase sa anotacijom*

4.4. Registracija i prijava

Za stvaranje i prijavljivanje korisnika koristi se funkcija koju pruža Supabase na strani poslužitelja. Potrebno je implementirati sučelje i logiku prijavljivanja u aplikaciji na klijentskoj strani. Budući da se koristi *Bloc* arhitektura potrebno je napraviti izvor podataka, repozitorij, *Bloc* klasu i sučelje. Na programskom kodu 4.3. se može vidjeti da repozitorij kroz konstruktor prima klasu za izvor podataka i klasu koja poziva metode za autentifikaciju.

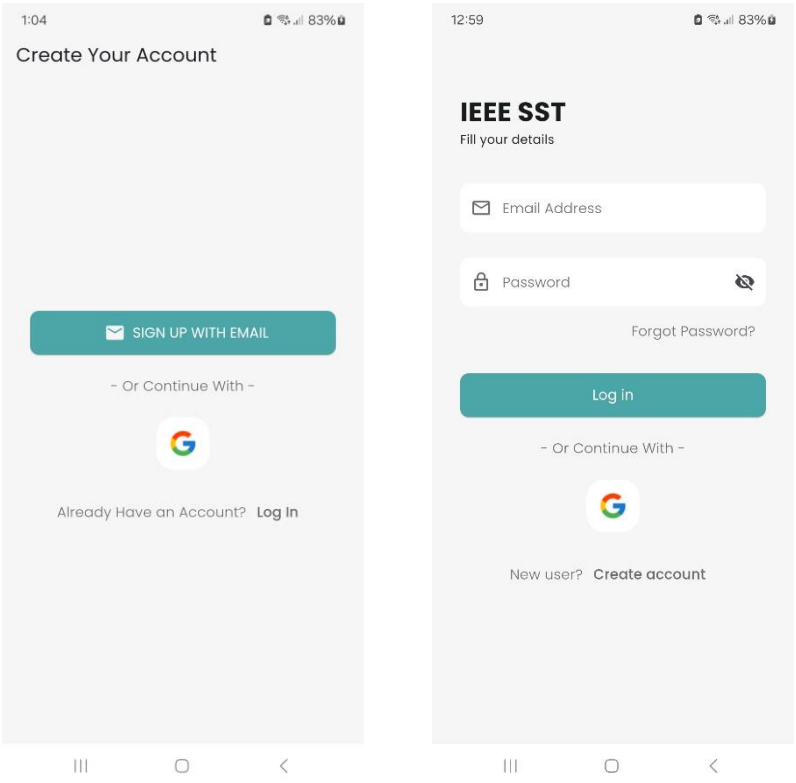
```
@LazySingleton(as: AuthenticationRepository)
class SupabaseAuthRepository implements AuthenticationRepository {
  SupabaseAuthRepository(this._supabase, this._profileApi);
  final SupabaseClient _supabase;
  final SupabaseProfileApi _profileApi;
```

Programski kod 4.3. *Dio koda klase koja poziva Supabase funkcije za autentifikaciju*

Instance tih klasa dostavlja paket *getIt*, zbog toga nigdje u kodu aplikacije nije potrebno ručno ih instancirati. Klasa *SupabaseAuthRepository* sadrži metode koje primaju podatke za registraciju što korisnik ručno unosi te ih prosljeđuje i poziva funkcije na serveru za registraciju.

Sljedeći korak je implementacija sučelja za prijavu i registraciju. U prezentacijskom sloju je potrebno napraviti direktorije koji predstavljaju određene dijelova sučelja, u ovom slučaju potrebno je imati dva direktorija za svaki od zaslona, za prijavljivanje i registraciju. Svaki direktoriji sadrži poddirektorije za zaslone, widgete i *Bloc* klase.

Cilj je napraviti sučelje koje ima jednostavan proces autentifikacije. Način realizacije sučelja za prijavu se može vidjeti na slici 4.4.



Slika 4.4. *Izgled sučelja za prijavu i registraciju*

U aplikaciji postoji klasičan korisnik i korisnik koji je administrator. Klasični korisnik može pregledavati događaje, označavati događaje, postavljati pitanja te komunicirati s ostalim korisnicima. Administratori imaju svoje odvojeno sučelje na kojemu se stvara i upravlja događajima, sponzorima i obavijestima. Prilikom prijave iz tablice za profile u bazi podatka provjerava se je li korisnik administrator ili ne, ovisno o ulozi korisnika pomoću *Bloc* paketa se korisnik preusmjerava na prikladan zaslon.

U programskom kodu 4.4. se nalazi *BlocConsumer* što je *widget* koji objedinjuje *BlocListener* i *BlocBuilder widgete* tj. objedinjuje funkcionalnost izgradnje elemenata i izvođenja funkcija. Nakon što korisnik unese svoje podatke i pritisne na tipku za prijavu, koji su upravljani pomoću *LoginBloc* klase, aktivira se događaj koji šalje zahtjev za prijavu korisnika na server.

```
child: BlocConsumer<LoginBloc, LoginState>(
  listener: (context, state) {
    if (state.status.isSuccess && state.isAdmin) {
      context.go(RoutePaths.adminHomeScreen);
    }
    if (state.status.isSuccess && !state.isAdmin) {
      context.go(RoutePaths.home);
    }
  },
```

***Programski kod 4.4**. Dio koda koji prati stanje i preusmjerava korisnika*

Klasa stanja za prijavu sadrži sve podatke potrebne za prijavljivanje, stanje statusa poziva na server, validaciju unesenih podataka i provjeru je li korisnik koji se prijavljuje administrator. Uz navedeno stanje može se u programskom kodu 4.5. primijetiti da se koriste još dva različita paketa, *freezed* i *Formz.*

```
@freezed
class LoginState with _$LoginState {
  const factory LoginState({
    @Default(FormzSubmissionStatus.initial) FormzSubmissionStatus status,
    @Default(Email.pure()) Email email,
    @Default(Password.pure()) Password password,
    @Default(false) bool isValid,
    @Default('') String errorMessage,
    @Default(false) bool isAdmin,
  }) = _LoginState;
}
```

***Programski kod 4.5.** Kod s kojim se definirana stanje Bloc klase za prijavu*

Paket *freezed* je generator koda, koji uz kada je neka klasa označena prikladnom anotacijom s ključnom riječi *freezed,* generira kod koji se može koristiti nad instancama te klase. Taj kod su najčešće operacije koje klase generalno zahtijevaju, poput operacija za usporedbu, kopiranje i sl.

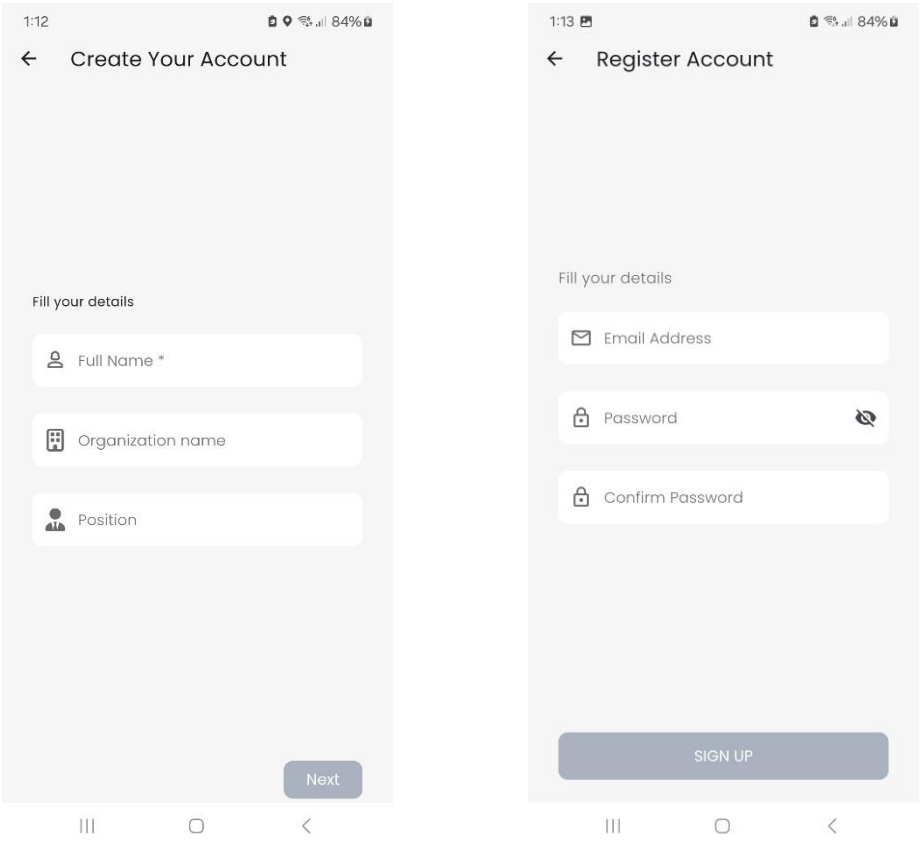
Prednost korištenja ovog paketa je što smanjuje količinu koda ju je potrebno ručno pisati za svaku od klasa. Budući da su paketi *freezed* i *Bloc* povezani koristiti će se *copyWith* metoda za promjenu dijelova stanja *LoginBloc* klase. Drugi paket koji se koristi je *Formz* koji služi za validaciju unesenih podataka u polja za prijavu i praćenje statusa poziva funkcije na poslužitelju.

U programskom kodu 4.6. se nalazi metoda za prijavu iz *LoginBloc* klase. Kada se navedena metoda pozove, mijenja se stanje tako što se postavi da je funkcija pokrenuta sa svojstvom *„FormzSubmissionStatus.inProgress“*. Nakon toga poziva se funkcija iz repozitorija za prijavu gdje se prosljeđuju uneseni podatci na poslužitelj, te se izvršava autentifikacija. Iz povratnih podataka prijave se postavlja stanje ovisno o tome je li prijavljeni korisnik administrator ili ne. Kada se ta promjena dogodi korisnik je preusmjeren na odgovarajuće sučelje.

```
Future<void> _onSubmitted(
  _Submitted event,
  Emitter<LoginState> emit,
) async {
  try {
    emit(state.copyWith(status: FormzSubmissionStatus.inProgress));

    final loginResponse = await _authRepository.signInWithEmailAndPassword(
      state.email.value,
      state.password.value,
    );

    if (loginResponse.user!.userMetadata!['role'] == 'admin') {
      emit(state.copyWith(
        status: FormzSubmissionStatus.success,
        isAdmin: true,
      ));
    } else {
      emit(state.copyWith(
        status: FormzSubmissionStatus.success,
        isAdmin: false,
      ));
    }
  } on AuthException catch (e) {
    emit(state.copyWith(
      status: FormzSubmissionStatus.failure,
      errorMessage: e.message,
    ));
  }
}
```

Programski kod 4.6. *Povezivanje događaja sa stanjem prijave unutar BLoC klase za prijavu*

Za registriranje korisnika, implementacija je ostvarena na sličan način. Razlikuje se u procesu unošenja podataka zato što registracija zahtjeva više različitih podataka za unos kroz dva zaslona kako je prikazano na slici 4.5.



Slika 4.5. *Prikaz sučelja za registraciju*

4.5. Autentifikacija

Proces autentifikacije je odvojen od logike upisivanja podataka za prijavu i registraciju. Za potrebe praćenja stanja autentifikacije koristi se odvojeno stanje tj. *Bloc* koji će pratiti je li trenutni korisnik prijavljen ili ne. Također se koristi stanje za greške, ako se dogodi pogreška prilikom autentificiranja, Supabase će vratiti odgovarajuću poruku ovisno o pogreški. Implementacija ovih stanja se mogu vidjeti u programskom kodu 4.7. Zbog toga što Supabase paket pruža svoju definiciju „*User*“ modela potrebno je definirati generičku klasu za korisnika u slučaju da se u budućnost dogodi potreba za mijenjanjem Supabase platforme na drugu sličnu platformu.

```
@freezed
class AuthState with _$AuthState {
  const factory AuthState.authenticated(BaseUserModel user) = _Authenticated;
  const factory AuthState.unauthenticated() = _Unauthenticated;
  const factory AuthState.error(String message) = _Error;
}
```

Programski kod 4.7*. Stanja za Bloc autentifikacije*

Prilikom aplikacije korisnik može biti uspješno ili neuspješno autoriziran. Ovisno o tome što se dogodi prilikom autentifikacije također je moguće da se dogodi greška što zahtjeva dodatno stanje kojim treba upravljati ukoliko se to dogodi.

Supabase paket ima svoji tok podataka u kojem se prate stanja korisnika, kroz taj tok podataka se može tražiti i mijenjati autentifikacija korisnika. Stoga je prvo potrebno imati repozitorij koji će izdvojiti podatke i funkcionalnosti tog paketa. Kada se pokrene *signIn* metoda preko prethodno objašnjene funkcionalnosti za prijavu, tok podataka za autentifikaciju se ažurira te repozitorij i *Bloc* klase koje su definirane u programskom kodu 4.8. slušaju taj tok podataka i emitiraju prikladna stanja.

```
@freezed
class AuthEvent with _$AuthEvent {
  const factory AuthEvent.signOut() = _SignOut;
  const factory AuthEvent.onCurrentUserChanged(BaseUserModel? user) =
      _OnCurrentUserChanged;
  const factory AuthEvent.onInitialAuthEvent() = _OnInitialAuthEvent;
}

void _onSignOut(
  AuthEvent event,
  Emitter<AuthState> emit,
) async {
  await _supabaseAuthRepository.signOut();
}

void _startUserSubscription() {
  _authSubscription =
      _supabaseAuthRepository.getCurrentUserStream().listen((user) {
    add(AuthEvent.onCurrentUserChanged(user));
  });
}

void _onCurrentUserChanged(
  _OnCurrentUserChanged event,
  Emitter<AuthState> emit,
) {
  event.user != null
      ? emit(AuthState.authenticated(event.user!))
      : emit(const AuthState.unauthenticated());
}

@override
Future<void> close() {
  _authSubscription?.cancel();
  return super.close();
}
```

Programski kod 4.8. *Događaji za Bloc autentifikacije*

Također postoji funkcija koja se koristi prilikom navigacije kako bi se za vrijeme pokretanja aplikacije utvrdilo postoji li već prijašnja sesija. U sljedećem poglavlju će biti objašnjeno kako rute i navigacija aplikacije funkcioniraju. Bitno je naglasiti da se preko klijenta tj. korisnika da Supabase može vidjeti postoji li već prijašnja sesija. Sesija se stvara nakon što se korisnik uspješno

prijavi ili registrira. Način korištenja te sesije gdje se definira inicijalna ruta aplikacije se može vidjeti u programskom kodu 4.9.

```
String getInitialRoute() {
  final Session? session = _supabaseClient.auth.currentSession;
  if (session == null || session.isExpired) {
    return RoutePaths.login;
  }
  if (session.user.userMetadata!['role'] == UserRoles.admin) {
    return RoutePaths.adminHomeScreen;
  }
  return RoutePaths.home;
}
```

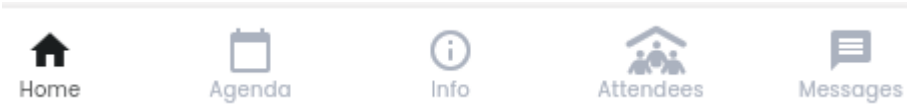
***Programski kod 4.9**. Definiranje inicijalne rute aplikacije.*

Duljina trajanje sesije se definira preko web stranica na Supabase projektu. Prilikom korištenja aplikacije može se dogoditi da korisnik ugasi aplikaciju. Nakon gašenja aplikacije sesija se čuva, tako da kada korisnik sljedeći put otvori aplikaciju može se automatski prijaviti nazad i postaviti inicijalnu rutu ovisno o tome je li sesija istekla te ili je administrator ili ne.

4.6. Navigacija

Navigiranje unutar mobilnih aplikacija funkcionira tako što se koristi stog gdje se na vrhu dodaju ili brišu zasloni aplikacije. U Flutter okruženju se to postiže korištenjem *Navigator widgeta*, ali također postoji *GoRouter* paket koji olakšava implementaciju i upravljanje logikom navigacije unutar aplikacije. Paket se koristi zbog toga što pruža mogućnost implementacije nekoliko različitih stogova za navigaciju, pojedinačnih ruta i upravljanje zaslonima pomoću navigacijske trake. *GoRouter* paket funkcionira tako što uz već postojeću logiku Flutter navigacije pomoću stoga dodaje logiku navigacije koristeći URL poveznice [10].

Aplikacija treba podržavati nekoliko različitih navigacijskih stogova jer nakon zaslona za registraciju treba biti, ovisno o vrsti korisnika, na dnu zaslona navigacijska traka. Na navigacijskoj traci se nalaze tipke koje preusmjeravaju korisnika na različite segmente aplikacije. Izgled navigacijske trake se može vidjeti na slici 4.6.



***Slika 4.6**. Navigacijska traka na dnu zaslona*

Svaki od segmenata na navigacijskoj traci treba imati svoj zaseban navigacijski stog. Na svakom od segmenata moguće je otići na novi zaslon koji je specifičan za taj segment i predstavlja definiranu rutu do tog zaslona. Kada korisnik pritisne tipku za navigaciju unazad, očekuje se da iz stoga makne zaslon na kojem se nalazi te da se vrati nazad na zaslon koji je prethodio u tom segmentu. S tim pristupom rješava se problem vračanja zaslona unazad bez obzira je li korisnik promijenio segment aplikacije preko donje navigacijske trake.

Kako bi se ovakav stil navigacije realizirao potrebno je napraviti klasu *AppRouter* koja upravlja rutama cijele aplikacije. Klasa morawh sadržavati konstruktor za *GoRouter* u kojemu je definiran svaki zaslon aplikacije te koji zasloni su ugniježđeni jedan unutar drugog. Dijelovi navedene klase se mogu vidjeti u programskom kodu 4.9.

```
), // GoRoute
// Admin routes
StatefulShellRoute.indexedStack(
  pageBuilder: (context, state, child) => NoTransitionPage(
    child: AdminScaffoldWtihNavBar(navigationShell: child),
  ), // NoTransitionPage
  branches: [
    StatefulShellBranch(
      navigatorKey: _navigatorKeyManager.shellNavigatorAdminHomeKey,
      routes: [
        GoRoute(
          path: RoutePaths.adminHomeScreen,
          pageBuilder: (context, state) => const MaterialPage(
            key: ValueKey('AdminHomeScreen'),
            child: AdminHomeScreen(),
          ), // MaterialPage
        ), // GoRoute
      ],
    ), // StatefulShellBranch
    StatefulShellBranch(
      navigatorKey:
          _navigatorKeyManager.shellNavigatorAdminEventsManagmentKey,
      routes: [
        GoRoute(
          path: RoutePaths.adminEventsMangment,
          pageBuilder: (context, state) => const MaterialPage(
            key: ValueKey('AdminEventsManagmentScreen'),
            child: EventManagmentScreen(),
          ), // MaterialPage
          routes: [
            GoRoute(
              path: RoutePaths.subRouteCreateEventName,
```

Programski kod 4.9. *Prikaz definiranja ruta unutar aplikacije.*

Potrebno je koristiti *StatefullShellRoute* i *StatefullShellBranch* klase unutar kojih se definiraju pojedinačne rute pomoću *GoRouter* paketa. *StatefullShellRoute* definira pojedinačni stog tj. segment aplikacije unutar kojeg se nalazi zasloni i rute specifične za taj segment. Svaki od segmenata je označen indeksom gdje svaki od indeksa predstavlja pojedinačni stog zaslona. Kako bi bilo moguće uopće rukovati rutama i indeksima potrebno je definirati za svaki stog navigacijski

ključ koji najčešće je predstavljen imenom toga što sadrži. Svaka od ruta također ima svoj jedinstveni ključ kojim se pristupa prilikom upravljanja i mijenjanjem zaslona prilikom korištenja aplikacije. Zadnji korak za definiranje ruta je postavljanje putanje rute i koji zaslon ta putanja treba prikazati korisniku.

Kao što je prikazano na u programskom kodu 4.10. prilikom prikazivanja zaslona kojih korisnik izabere, prvo se definira *pageBuilder* element koji za svaki zaslon izgrađuje prikladnu navigacijsku traku.

```
StatefulShellRoute.indexedStack(
  pageBuilder: (context, state, child) => NoTransitionPage(
    child: AdminScaffoldWtihNavBar(navigationShell: child),
  ), // NoTransitionPage
```

***Programski kod 4.10**. Prikaz sučelja za navigaciju ovisno o segmentima i rutama*

Taj element služi kao glavni gradivni element korisničkog sučelja na kojeg se onda tek nadovezuju zasloni kojima se korisnik služi. Postoje dvije vrste indeksiranih stogova u ovoj aplikaciji, to su administratorska i korisnička. Prilikom prijave, kada aplikacija preusmjerava korisnika na pravilnu putanju, *GoRouter* paket provjerava unutar kojeg stoga se nalazi ta putanja te izgrađuje prikladne elemente i korisničko sučelje.

Zadnji korak pri definiranju ovakvog stila navigacije je stvaranje globalnih ključeva za navigacijska stanja. Za svaki od segmenata je potrebno imati svoj navigacijski ključ. Taj ključ pruža Flutter razvojno okruženje, te kada se definira neki segment mora se kreirati prikladan objekt klase koja upravlja ključevima. Ti ključevi se prosljeđuju kasnije u *GoRouter* paket, te se definiranje tih ključeva može vidjeti u programskom kodu 4.11.

```
class NavigatorKeyManager {
  final GlobalKey<NavigatorState> rootNavigatorKey =
      GlobalKey<NavigatorState>();

  final GlobalKey<NavigatorState> shellNavigatorHomeKey =
      GlobalKey<NavigatorState>(debugLabel: 'ShellNavigatorHomeKey');

  final GlobalKey<NavigatorState> shellNavigatorAgendaKey =
      GlobalKey<NavigatorState>(debugLabel: 'ShellNavigatorAgendaKey');
  ...
```

***Programski kod 4.11.** Prikaz klase za upravljanje ključevima za navigaciju*

4.6.1. Scaffhold

Kako bi Flutter razvojni okvir uopće mogao izgraditi sučelje koristi se *Scaffhold* widget koji pruža mogućnost izgradnje za elemente poput navigacijskih traka, elemente samog korisničkog sučelja. Na slici 4.11 mogu se vidjeti primjeri za korisnički i administratorski *scaffhold*.

```dart
class AdminScaffoldWtihNavBar extends StatelessWidget {
  const AdminScaffoldWtihNavBar({super.key, required this.navigationShell});

  final StatefulNavigationShell navigationShell;

  void _goBranch(int index) {
    navigationShell.goBranch(
      index,
      initialLocation: index == navigationShell.currentIndex,
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: BlocProvider(
        create: (context) => getIt<EventFormBloc>(),
        child: navigationShell,
      ), // BlocProvider
      resizeToAvoidBottomInset: true,
      bottomNavigationBar: BottomNavBar(
        key: const ValueKey('BottomNavBar'),
        tabs: const ['Home', 'Event', 'Sponsors', 'Announcements'],
        icons: const [
          Icons.home,
          Icons.event,
          Icons.handshake,
          Icons.announcement,
        ],
        onDestinationSelected: _goBranch,
      ), // BottomNavBar
    ); // Scaffold
  }
}
```

```dart
class ScaffoldWithNavBar extends StatelessWidget {
  const ScaffoldWithNavBar({
    super.key,
    required this.navigationShell,
  });

  final StatefulNavigationShell navigationShell;

  void _goBranch(int index) {
    navigationShell.goBranch(
      index,
      initialLocation: index == navigationShell.currentIndex,
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: navigationShell,
      resizeToAvoidBottomInset: true,
      bottomNavigationBar: BottomNavBar(
        key: const ValueKey('BottomNavBar'),
        tabs: const ['Home', 'Agenda', 'Info', 'Attendees', 'Messages'],
        icons: const [
          Icons.home,
          Icons.calendar_today,
          Icons.info_outline_rounded,
          FontAwesomeIcons.peopleRoof,
          Icons.message,
        ],
        onDestinationSelected: _goBranch,
      ), // BottomNavBar
      extendBody: true,
    ); // Scaffold
  }
}
```

Programski kod 4.11. *Scaffhold widget koji pruža izgradnju potrebnih elemenata*

Također je implementirano svojstvo povratka na inicijalnu rutu stoga ako je pritisnuta tipka za segment koji je već aktivan. Kako bi se to postiglo potrebno je provjeriti inicijalnu lokaciju i trenutni indeks, što se može vidjeti u programskom kodu 4.12. To je gesta koja je implementirana na mnogo drugih novijih aplikacija, te prilikom korištenja novih aplikacija, korisnici očekuju da mogu koristiti iste takve geste.

```dart
void _goBranch(int index) {
  navigationShell.goBranch(
    index,
    initialLocation: index == navigationShell.currentIndex,
  );
}
```

Programski kod 4.12. *Funkcija za promjenu indeksa i aktivnog segmenta*

4.6.2. Navigacijska traka

Unatoč tome što je bilo potrebno za izgradnju dva različita *scaffhold widgeta*, za navigacijsku traku je potrebno imati samo jedan element koji je postavljen tako da prima parametre koji predstavljaju ikone i imena zaslona koje će navigacijska traka prikazivati. To je prikazano u programskom kodu 4.13.

```
Widget build(BuildContext context) {
  return BlocProvider(
    create: (context) => ScreenIndexCubit(),
    child: BlocBuilder<ScreenIndexCubit, int>(
      builder: (context, state) {
        return BottomNavigationBar(
          type: BottomNavigationBarType.fixed,
          backgroundColor: AppColors.white,
          useLegacyColorScheme: false,
          selectedItemColor: AppColors.black,
          unselectedItemColor: AppColors.gray,
          enableFeedback: true,
          selectedLabelStyle: const TextStyle(
            color: AppColors.black,
            fontSize: 10,
          ), // TextStyle
          unselectedLabelStyle: const TextStyle(
            color: AppColors.gray,
            fontSize: 10,
          ), // TextStyle
          currentIndex: state,
          items: List.generate(
            tabs.length,
            (index) => BottomNavigationBarItem(
              icon: Icon(icons[index]),
              label: tabs[index],
            ), // BottomNavigationBarItem
          ), // List.generate
          onTap: (index) {
            BlocProvider.of<ScreenIndexCubit>(context).setIndex(index);
            onDestinationSelected(index);
          },
        ); // BottomNavigationBar
      },
    ), // BlocBuilder
  ); // BlocProvider
```

Programski kod 4.13. *Kod za navigacijsku traku*

Kako bi se pratilo stanje i promjena između segmenata koristi se *ScreenIndexCubit.* Taj *cubit*, u programskom kodu 4.14., je vrlo jednostavan i sadrži samo jedno stanje tipa *int* kojim se emitira tj. segment na kojem se nalazi.

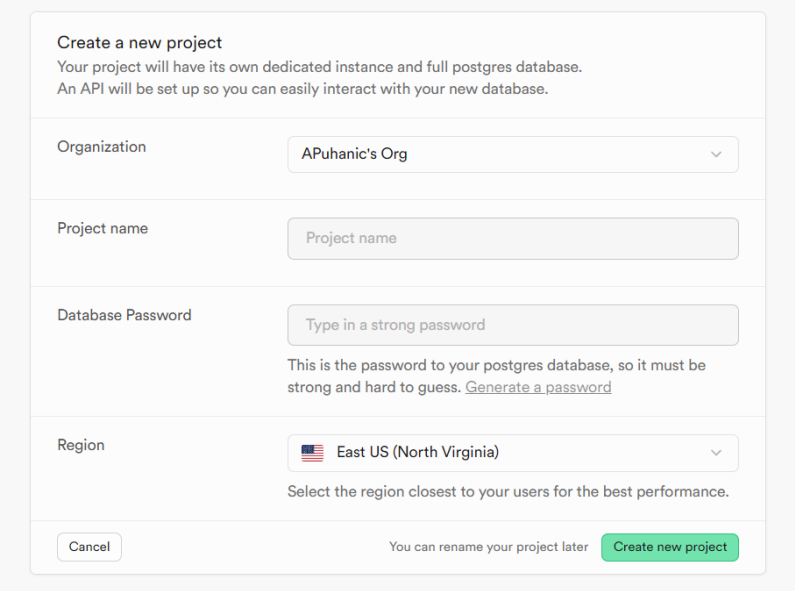
```
class ScreenIndexCubit extends Cubit<int> {
  ScreenIndexCubit() : super(0);

  void setIndex(int index) => emit(index);
}
```

Programski kod 4.14. *Kod za emitiranje stanja indeksa navigacijske trake*

Kada se stanje promjeni tj. emitira, preko *BlocBuilder* widget koji ga sluša, gradi se ponovno te iz svog novog stanja mijenja *currentIndex* svojstvo navigacijske trake što uzrokuje izgradnju novog zaslona.

4.7. Supabase

Proces postavljanja novog Supabase projekta je jednostavan te zahtjeva samo nekoliko početnih podataka. Nakon nekoliko minuta, projekt je generiran i spreman za korištenje što uključuje stvaranje tablica i stvaranje API poziva na istu.



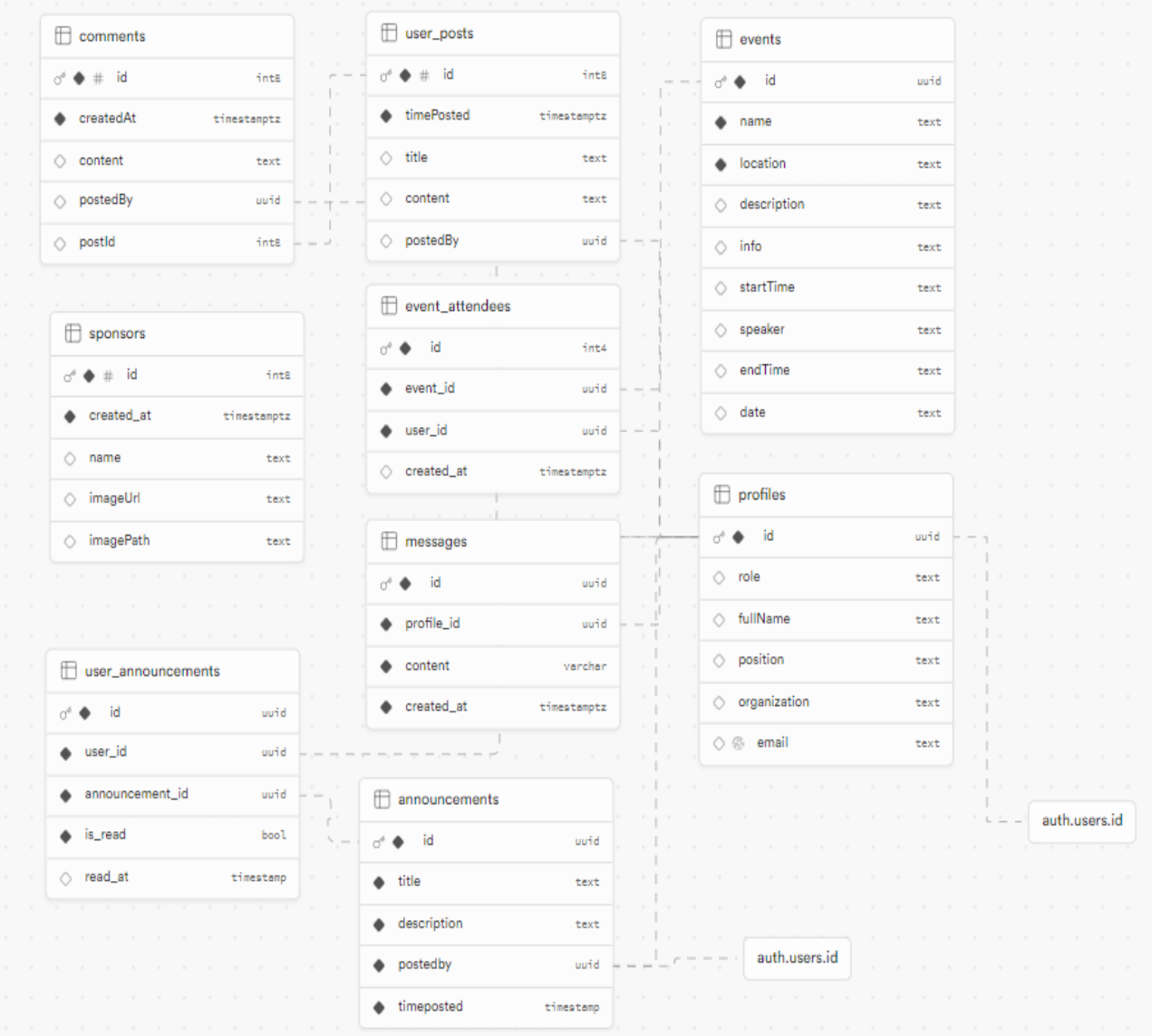
***Slika 4.7**. Sučelje za stvaranje Supabase projekta*

Za stvaranje tablica i RPC poziva unutar projekta koristi se *„SQL editor“* funkcionalnost na web stranici. Prva tablica koju je potrebno napraviti je tablica za popis registriranih korisnika. Iako prilikom registracije Supabase već sadržava internu tablicu s kojom automatski stvara popis korisnika, biti će i dalje potrebna nova tablica koja sadržava dodatne podatke o korisniku kao što su organizacija, pozicija i ime. Primarni ključ ove tablice je ujedno i strani ključ koji referencira već postojeću integriranu tablicu za korisnike. U programskom kodu 4.15 se može vidjeti SQL funkcija za kreiranje potrebne tablice.

```sql
create table profiles (
  id uuid references auth.users on delete cascade,
  role text default 'user',
  primary key (id)
);
```

Programski kod 4.15. *SQL kod za stvaranje tablice za profile korisnika*

Često je potrebno imati vizualizaciju strukture baze podataka. Supabase nakon što se stvore tablice automatski stvara shemu svih tablica te vizualizira kako su povezane i koji su strani ključevi. Izgled sheme baze podataka za ovu aplikaciju se može vidjeti na slici 4.8.



Slika 4.8. *Shema baze podataka*

4.8. Izgled aplikacije

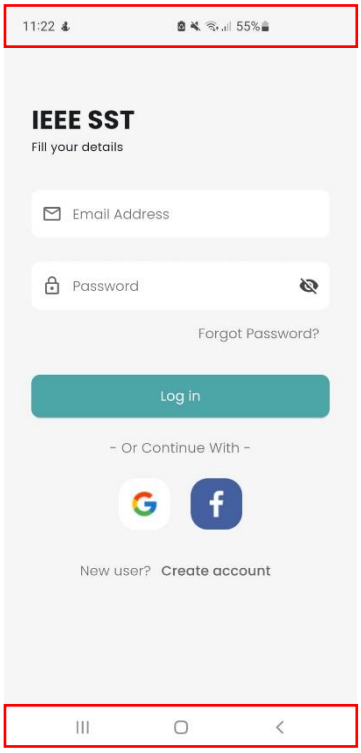
Kako bi se postigao izgled aplikacije na osnovu prethodnog dizajna potrebno je definirati boje i font za cijelu aplikaciju. Dobra je praksa imati apstraktne klase koje drže te podatke, što se može vidjeti u programskom kodu 4.16.

```
abstract class AppColors {
  static const primary = Color(0xFF4CA6A8);
  static const background = Color.fromARGB(255, 247, 246, 246);
  static const gray = Color(0xFFABB2BF);
  static const white = Color(0xFFFFFFFF);
  static const black = Color(0xFF1A1D1E);
  static const grayText = Color(0xFF6A6A6A);
  static const warning = Color(0xFFFF4141);
}
```

Programski kod 4.16. *Paleta boja koju aplikacija koristi*

Prilikom pisanja koda za sučelje i definiranje boja elemenata aplikacije poziva se klasa sa prethodne slike, prednost ovog načina korištenja je što olakšava promjenu izgleda aplikacije u budućnosti. U slučaju da je potrebno promijeniti boju, treba samo promijeniti vrijednost pojedine varijable te će se u cijeloj aplikaciji ta boja promijeniti. Na isti način su definirani stilovi teksta koji se u aplikaciji.

Potrebno je modificirati boje navigacijske trake i statusne trake samog mobitela kao što je na slici 4.9. To se postiže korištenjem *SystemChrome widgeta*, prije izgradnje samog sučelja aplikacije.



Slika 4.9. *Definiranje boja statusne trake i donje navigacijske trake*

Za definiranje fonta potrebno je definirati *„assets“* direktorij u kojem se stavljaju resursi poput slika i datoteka za vanjske fontove koji nisu podržani unutar aplikacije. U ovom slučaju je potrebno imati definirano vanjski font *„Poppins“*, kao što se može vidjeti u programskom kodu 4.17. Kako bi aplikacija mogla učitati te podatke potrebno ih je staviti pod listu vanjskih resursa unutar *pubspec.yaml* datoteke.

```yaml
flutter:
  uses-material-design: true

  assets:
    - assets/images/
    - .env

  fonts:
    - family: Poppins
      fonts:
        - asset: assets/fonts/Poppins-Regular.ttf
        - asset: assets/fonts/Poppins-Bold.ttf
        - asset: assets/fonts/Poppins-Italic.ttf
        - asset: assets/fonts/Poppins-BoldItalic.ttf
        - asset: assets/fonts/Poppins-Light.ttf
        - asset: assets/fonts/Poppins-LightItalic.ttf
        - asset: assets/fonts/Poppins-Medium.ttf

flutter_launcher_icons:
  android: true
  ios: true
  image_path: "assets/images/ieee-sst-logo.png"
```

Programski kod 4.17. *Popis vanjskih resursa za font, slike i ikone*

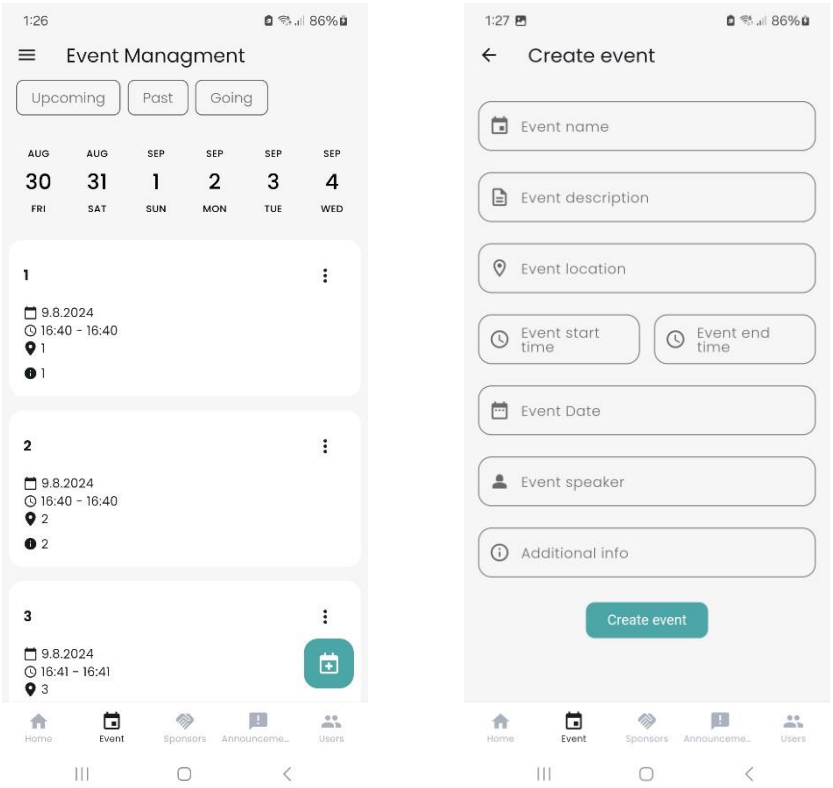
U ovu datoteku su uključene i ikone za aplikaciju koje su postavljene automatski pomoću paketa za generiranje ikona aplikacije i prikladnog početnog zaslona koji se prikazuje dok se aplikacija pokreće. U programskom kodu 4.18. se može vidjeti kako se objedinjuju boje i fontovi prije pokretanja aplikacije.

```dart
child: MaterialApp.router(
  debugShowCheckedModeBanner: false,
  title: 'IEEE SST',
  theme: ThemeData(
    useMaterial3: true,
    scaffoldBackgroundColor: AppColors.background,
    fontFamily: 'Poppins',
  ), // ThemeData
  routerConfig: getIt<AppRouter>().router,
  builder: (context, child) => SafeArea(child: child!),
), // MaterialApp.router
```

Programski kod 4.18. *Pokretanje aplikacije s postavljenim fontom i korištenjem definiranih boja.*

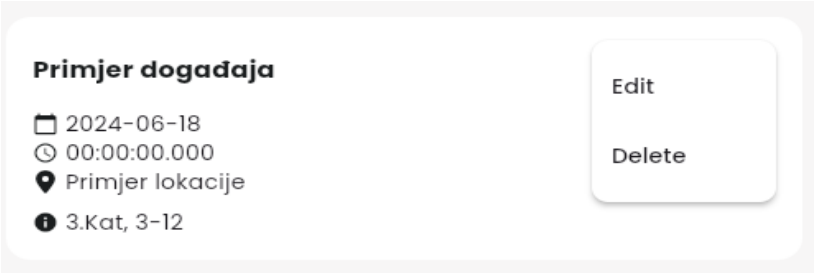
4.9. Dodavanje događaja

Za ovu i buduće konferencije koje će se događati potrebno je imati način na koji će se dodavati i brisati planirani događaji za konferenciju. Administratori imaju pravo kreiranja i brisanja događaja, dok korisnici imaju samo pravo pregledavati listu događaja i označiti na koje će događaje ići. Prilikom prijave na zaslonu agende, koji se može vidjeti na slici 4.10., postoji *floatingActionButton* tipka koja preusmjerava administratora na zaslon za upisivanje podataka o novom događaju.



Slika 4.10. *Prikaz sučelja za upravljanje događajima*

Na slici 4.11 se vidi na desnoj strani kartice koja prikazuje detalje pojedinog događaja da se nalazi meni koji kada se pritisne prikazuje dvije dodatne opcije za uređivanje i brisanje.



Slika 4.11*. Opcije za brisanje i uređivanje događaja*

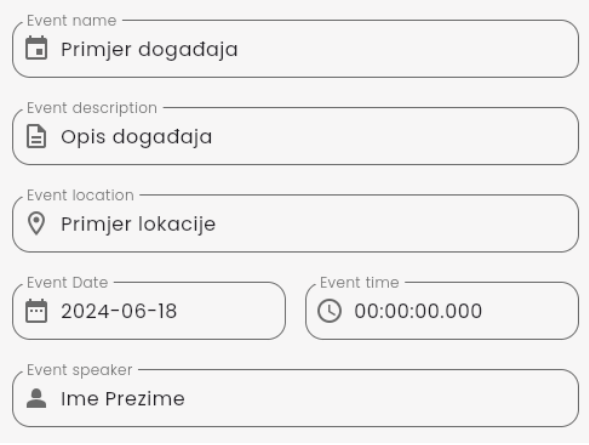
Iznad liste događaja se *filterChip* elementi i traka s datumima kojima se može filtrirati popis događaja. Na vrhu zaslona kod korisnika također se nalazi ikona koja pretraživanje događaja po imenu.

Logika ovog sučelja je podijeljena u dva dijela, a to su dio za prikazivanje podatka i unos podataka. Za svaki od dijelova koristiti će se poseban *Bloc*. Za unos detalja o događaju i spremanje podataka na bazu podataka potrebno je imati stanje koje sadrži vrijednosti svih formi za unos podataka o događaju te vrijednost za praćenje statusa API poziva dok se izvršava spremanje u bazu podataka. Klasa stanja u programskom kodu 4.19. je definirana na sličan način kao i stanja za prijavu i registraciju, ali u ovom slučaju je razlika u tome što se ista forma treba koristiti i za uređivanje događaja.

```
@freezed
class EventFormState with _$EventFormState {
  const factory EventFormState({
    String? id,
    @Default(FormzSubmissionStatus.initial) FormzSubmissionStatus status,
    @Default('') String name,
    @Default('') String description,
    @Default('') String location,
    @Default('') String errorMessage,
    @Default('') String speaker,
    DateTime? time,
    String? info,
  }) = _CreateEventState;
}
```

Programski kod 4.19. *Stanje Bloca za unos događaja*

Kada se pritisne opcija za uređivanje već postojećeg događaja, otvara se sučelje istog izgleda kao i za unos podataka. Kada se otvori sučelje za uređivanje događaja, prikazano na slici 4.10., forme već moraju biti ispunjene postojećim podatcima tih događaja, stoga se koristi ista *Bloc* klasa za dodavanje i uređivanje.



Slika 4.10. *Početne vrijednosti događaja prilikom uređivanja*

Kako bi uopće bilo moguće ažurirati događaj, prilikom poziva funkcije za promjenu podataka potrebno je proslijediti ID događaja koji se uređuje. Zato je varijabla *id* označena sa simbolom „?“, što omogućuje da varijabla bude prazna tj. *null,* jer prilikom kreiranja događaja podataka ta varijabla nije potrebna jer Supabase prilikom spremanja u bazu podataka automatski generira ID.

Za mijenjanje u bazi podataka koristi se *Upsert* koja ja radi kao funkcije *Update* i *Insert* zajedno. Pojednostavljuje proces ažuriranja tako što omogućuje da se pošalju novi podatci te na osnovu tih podataka traži postojeći redak koji je najsličniji tome i ažurira ga. Ako taj objekt u bazi podataka ne postoji onda se stvara i zapisuje novi objekt.

4.10. Filtriranje događaja

Kako bi se filtrirali podatci koristite se elementi koji su jednostavni koji jednim klikom aktiviraju filter i prikazuju filtrirane događaje. Najvažniji način filtriranja je po kategorijama i datumu. Za filtriranje po kategorijama koristi se *FilterChip widget*. Za upravljanje stanjem aktivnih filtera umjesto *Bloc* klase potrebno je koristiti *Cubit* klasu jer stanje koje se prati nije kompleksno. Potrebno je samo pratiti koji od filtera je aktivan te prilikom pritiska na filter promijeniti stanje aktivnog filtera na način kako je prikazano u programskom kodu 4.20.

```
class FilterChipCubit extends Cubit<FilterChipState> {
  FilterChipCubit() : super(const FilterChipState.initial());

  void selectFilterChip(FilterType filterType) =>
      emit(FilterChipState.selected(filterType));
}

class FilterChipItem {
  final String label;
  final FilterType filterType;

  FilterChipItem({
    required this.filterType,
  }) : label = filterType.toString().split('.').last[0].toUpperCase() +
          filterType.toString().split('.').last.substring(1);
}
```

Programski kod 4.20. *Cubit klasa za aktiviranje filtera*

Za realizaciju ovog widgeta bez obzira što nije *Bloc* klase i dalje je potrebno koristiti *BlocBuilder klasu*. Svaki put kada korisnik promjeni filter *BlocBuilder* widget je zadužen za izgradnju sučelja ovisno o tome što je korisnik pritisnuo. Implementacija filtriranja je prikazana na u programskom kodu 4.21.

```dart
return BlocBuilder<FilterChipCubit, FilterChipState>(
  builder: (context, state) {
    final filterType = FilterType.values[index];
    final isSelected = state.maybeWhen(
      selected: (selectedFilterType) =>
          selectedFilterType == filterType,
      orElse: () => false,
    );
    return FilterChip(
      label: Text(
        FilterChipItem(filterType: FilterType.values[index])
            .label,
        style: isSelected
            ? AppTextStyle.lightText
                .copyWith(color: AppColors.white)
            : AppTextStyle.lightText,
      ), // Text
```

***Programski kod 4.21**. Izgradnja filtera na korisničkom sučelju*

Funkcija *maybeWhen* je automatski generirana funkcija koji stvori paket *freezed.* Omogućuje da ovisno o stanju *Bloc* ili *Cubit* klase da se grade drugačiji elementi sučelja. Može se koristiti unutar *BlocListener widgeta* kako bi se obavljale funkcije kada se promjeni stanje, a nije potrebno mijenjati elemente sučelja.

Za prikazivanje datuma koristi se vanjski paket koji pruža izgradnju horizontalne trake u kojoj se nalaze datumi. Izgled paketa prije modificiranja se može vidjeti na slici 4.11. Paket pruža mnogo mogućnosti uređivanja i proširivanja, te u ovom slučaju se koristiti tako da prikazuje datume ovisno o tome koji je trenutni datum u vrijeme korištenja aplikacije.

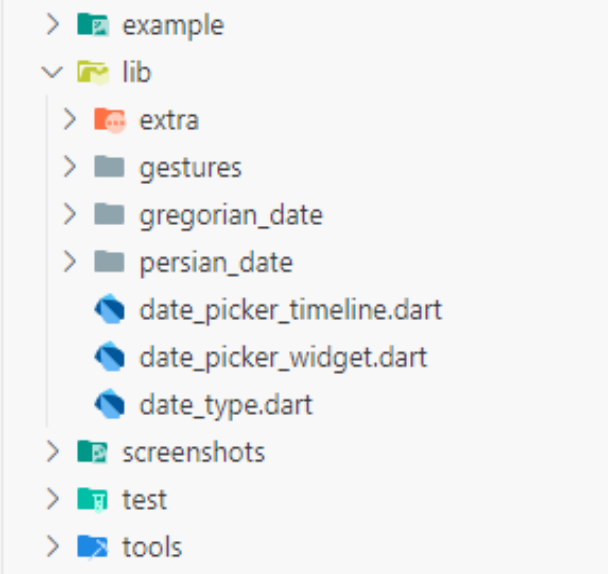
***Slika 4.11.** Prikaz datuma u vrijeme izrade aplikacije*

Kada budu događaji aktivni te kada se datum približi, moguće je pritisnuti na datum, te se ovisno o datumu filtriraju i prikazuju prikladni događaji. Prednost ovog načina prikazivanja datuma je što za buduće konferencije neće biti potrebno mijenjati funkcionalnost i način filtriranja događaja, funkcionirati će uvijek isto te će biti samo potrebno dodati nove događaje i automatski će biti moguće ih filtrirati.

4.10.1. Modificiranje paketa

Svi paketi sa *pub.dev* stranice koji se stave u projekt su otvorenog koda i slobodni za modificiranje. U ovome slučaju, paket za datume nije pružao mogućnost da nijedan datum bude izabran. Stoga je bilo potrebno klonirati taj paket, napraviti izmjene i postaviti ga na vlastiti GitHub račun. Za svaki paket postoji poveznica na originalni kod.

Iako je izvorni kod paketa uvijek napisan koristeći Flutter i Dart, paketi se razlikuju po tome što ne sadrže *main* datoteke, nego samo datoteke potrebne za izgradnju i logiku tog elementa što se može vidjeti na slici 4.12. Kako bi bilo moguće napraviti modifikacije nad paketom, potrebno je napraviti novi direktorij u projektu koji će sadržavati kod za pokretanje tog paketa.



Slika 4.12. *Prikaz direktorija paketa*

Za ovaj paket postoji *example* direktorij koji sadrži aplikaciju u kojemu je moguće implementirati element paketa, te uz pomoć toga modificirati izvorni kod po volji. Nakon što je paket izmijenjen da podržava prazan odabir datuma, promjene su postavljene na GitHub.

Budući da je potrebno koristiti modificirani paket, a ne originalni, potrebno je promijeniti u glavnoj aplikaciji da paket se instalira preko poveznice kloniranog paketa. To je potrebno napraviti u *„pubspec.yaml“* datoteci kako je prikazano u programskom kodu 4.22.

```
dependencies:
  date_picker_timeline:
    git:
      url: https://github.com/APuhanic/DatePickerTimelineFlutter.git
```

Programski kod 4.22. *Postavljanje izvora za instalaciju modificiranog paketa*

S ovime je postavljen samo *widget* koji vizualno pruža popis i interakciju sa datumima, ali još je potrebno postaviti funkcionalnost samog filtriranja. To je realizirano koristeći *Bloc* klase za događaje jer glavna svrha te klase je dohvaćanje događaja. Kada korisnik odabere pojedini datum, ta klasa će već se nalaziti u stanju gdje su učitani događaji, stoga je samo potrebno napraviti metodu koja će vratiti događaje ovisno u datumu ili filteru koji je izabran.

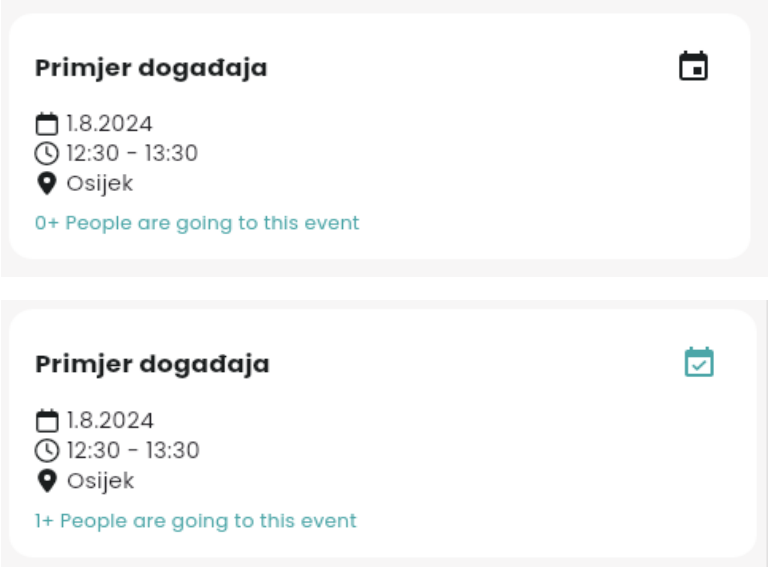
U programskom kodu 4.23. se nalazi kod za filtriranje događaja. Potrebno je prilikom pritiska na filter provjeriti je su li podatci u uopće učitani. Ako nisu neće se ništa dogoditi, ako jesu onda se nastavlja s izvođenjem koda, te se primjenjuju filteri. Nakon što se filteri primjenjuju, samo se ponovno emitira učitano stanje, ali sa drugačijim popisom događaja. Kada se to emitiranje dogodi, već postojeći element koji prikazuje popis događaja će reagirati na tu promjenu, dobiti će novi set podataka, te će se zaslon promijeniti i prikazati filter.

```
FutureOr<void> _onFilterEvents(
    _FilterEvents event, Emitter<EventsState> emit) {
  if (state is! _Loaded) return null;

  final isSameFilter = _isSameFilterApplied(event);

  List<Event> filteredEvents = _filterEventsByDate(event.date);

  if (isSameFilter) {
    emit(_Loaded(filteredEvents));
    return null;
  }
  filteredEvents = _filterEventsByType(filteredEvents, event.filter);

  emit(_Loaded(filteredEvents));
}
```

Programski kod 4.23. *Prikaz koda za filtriranje događaja*

4.11. Označavanje dolaznosti

Korisnik za svaki događaj može označiti da će ići na taj događaj. Kako bi se postigla jednostavnost prilikom korištenja, gumb za označavanje dolaznosti će se implementirati tako što kada se pritisne promijeniti će se ikona i boja ikone kako bi se označilo da je korisnik uspješno označen za dolaznost. Izgled označavanja prije i nakon se može vidjeti na slici 4.13.



Slika 4.13. *Prikaz označavanja dolaznosti*

Za spremanje dolaznosti, u bazi podataka je napravljena nova tablica koja sadržava popis ljudi i događaja na koje idu koristeći strane ključeve. U programskom kodu 4.24. se vidi da svaki novi korisnik koji pritisne tipku za dolaznost povećava broj koji se prikazuje na kartici za pregled događaja.

```
Future<FutureOr<void>> _onMarkGoing(
    _MarkGoing event, Emitter<EventsState> emit) async {
  await supabaseEventRepository.markGoing(event.event.id!);
  if (state is _Loaded) {
    final events = (state as _Loaded).events;
    final updatedEvents = events.map((e) {
      if (e.id == event.event.id) {
        return e.copyWith(isGoing: true, attendeeCount: e.attendeeCount + 1);
      }
      return e;
    }).toList();
    originalEvents.clear();
    originalEvents.addAll(updatedEvents);
    emit(_Loaded(updatedEvents));
  }
}
```

Programski kod 4.24. *Prikaz koda za označavanje dolaznosti*

Za implementaciju ove funkcionalnosti koriste se već postojeće klase i repozitorij gdje će se samo dodati nove metode kako je prikazano u programskom kodu 4.25.

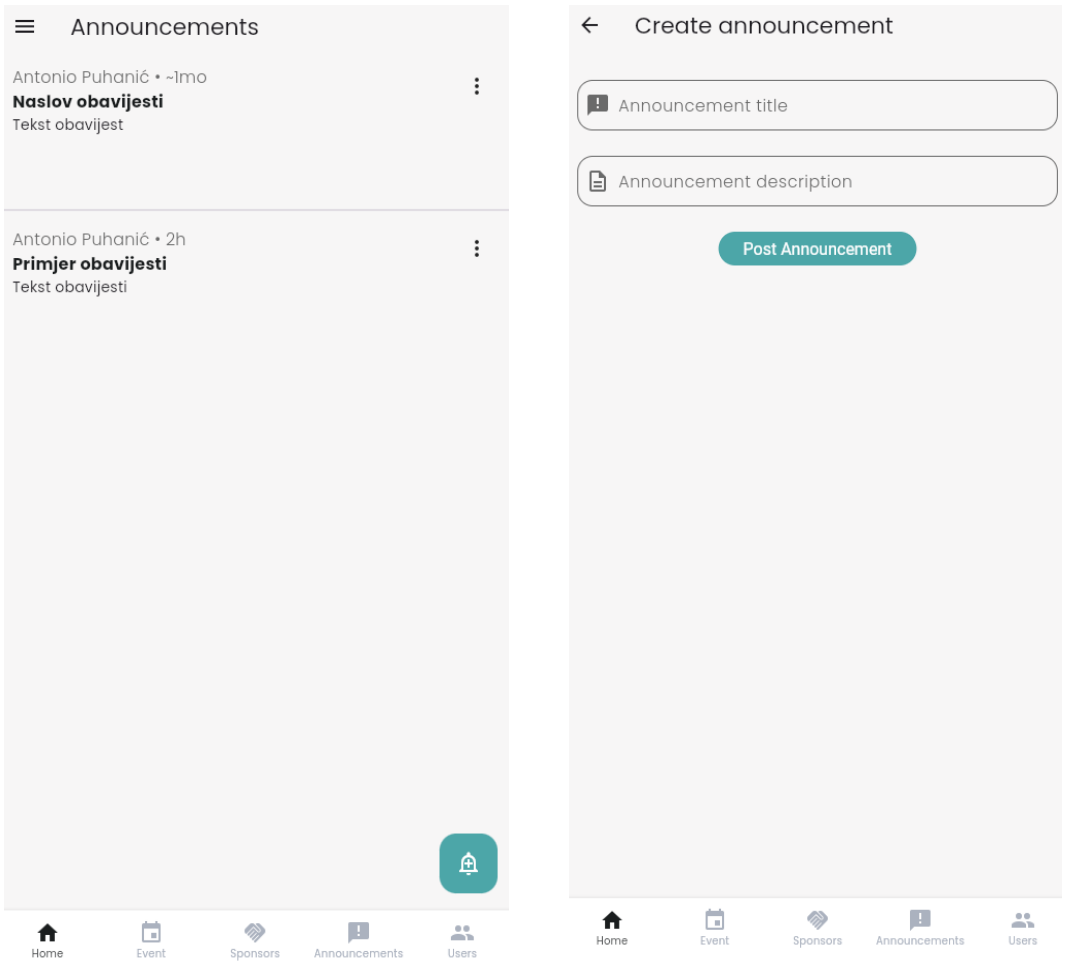
```
@override
Widget build(BuildContext context) {
  return BlocBuilder<IsGoingCubit, bool>(
    builder: (context, state) {
      return IconButton(
        onPressed: _onPressed,
        icon: isLoading
            ? const CircularProgressIndicator(color: AppColors.primary)
            : _buildIcon(),
        color: AppColors.primary,
      ); // IconButton
    },
  ); // BlocBuilder
}

void _onPressed() {
  setState(() => isLoading = true);
  debouncer.run(() {
    context.read<IsGoingCubit>().toggleIsGoing();
    final eventAction = widget.event.isGoing
        ? EventsEvent.markNotGoing(widget.event)
        : EventsEvent.markGoing(widget.event);

    context.read<EventsManagmentBloc>().add(eventAction);
    setState(() => isLoading = false);
  });
}
```

***Programski kod 4.25**. Kod za implementaciju tipke za dolaznost korisnika*

U ovom slučaju se koristi *Statefull Widget* uz već postojeće praćenje stanja pomoću *Bloc* paketa. Iako je standard da se koristi *Bloc* za upravljanje stanjima, nekada je to jednostavnije implementirati ovakve pomoću već postojećih funkcionalnosti poput *Statefull Widgeta* jer stanje *widgeta* za dolaznost je jedinstveno i neće se nigdje drugdje u aplikaciji koristiti.

4.12. Obavijesti

U slučaju promjena u konferenciji ili ako je potrebno objaviti nove i dodatne informacije događajima potrebno je imati sustav obavijesti. Izgled sučelja koji to sadrži se nalazi na slici 4.13. Kao i za događaje, samo administratori mogu objavljivati nove obavijesti koji onda korisnici mogu pregledavati. Ovo sučelje funkcionira na vrlo sličan način poput sučelja za događaje te je implementirano koristeći istu logiku za unošenje podataka i prikazivanje istih.



Slika 4.13. *Prikaz sučelja za upravljanje i kreiranje obavijesti s administratorske strance*

Prilikom prikaza pojedinačne obavijesti iznad naslova se nalazi ime korisnika koji je poslao obavijest. Budući da tablica za obavijesti sadrži samo podatke o obavijesti i strani ključ koji pokazuje na korisnika koji je postavio obavijest. Potrebno je napraviti da prilikom dohvaćanja podataka umjesto stranog ključa, koji je nasumično generiran ID, se dohvati ime korisnika. To se može postići kreiranjem SQL funkcija na Supabase web stranici. Potrebno je imati funkciju koja će vratiti podatke koji odgovaraju modelu klase koja se koristi za prikaz na obavijesti na zaslonu.

U programskom kodu 4.26. se vidi klasa koja je generirana koristeći *freezed* paket. Ova klasa sadrži sve podatke potrebne kako bi se obavijest prikazala na zaslonu. Najvažnija funkcionalnost koju ovaj paket pruža je generiranje koda koji pretvara JSON podatke u objekte klase *Announcement.*

```
@freezed
class Announcement with _$Announcement {
  const factory Announcement({
    String? id,
    required String title,
    required String description,
    required String timeposted,
    required String fullName,
  }) = _Announcement;

  factory Announcement.fromJson(Map<String, dynamic> json) =>
      _$AnnouncementFromJson(json);
}
```

Programski kod 4.26. *Prikaz modela za podatke obavijesti.*

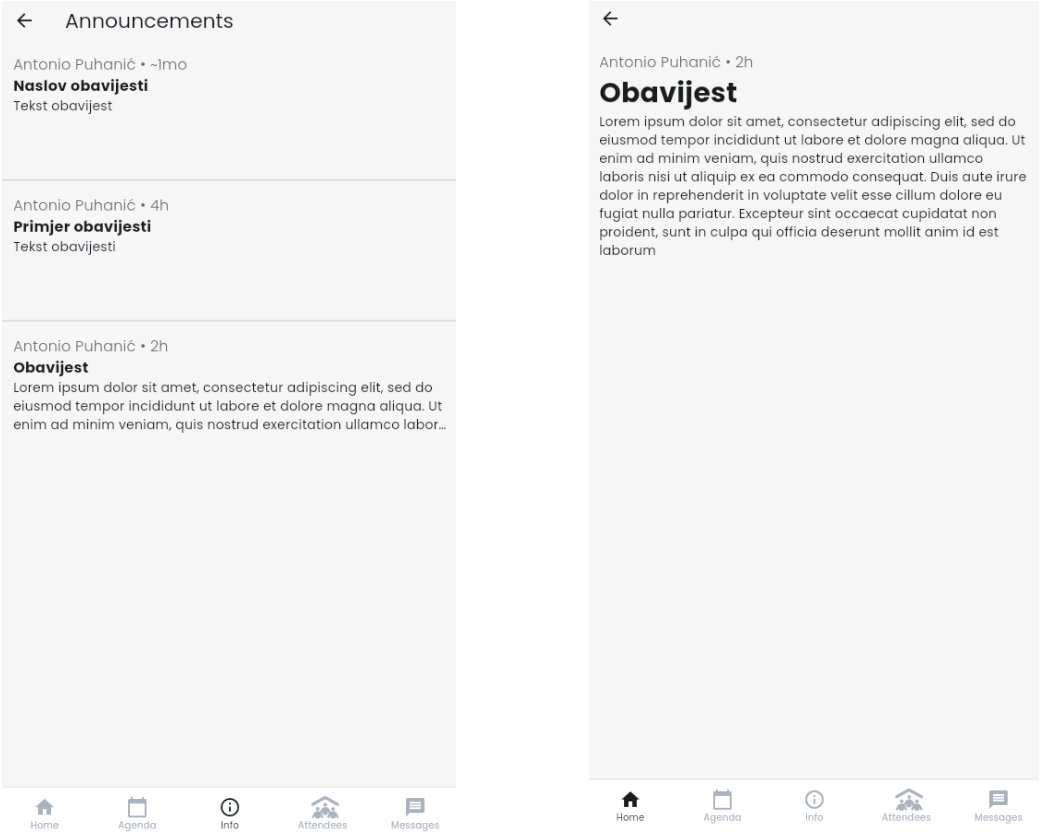
Nakon što je kreiran model, potrebno je napraviti funkciju koja će vratiti prikladne podatke te onda iste podatke propagirati kroz aplikaciju. U programskom kodu 4.27 se može vidjeti SQL kod navedene funkcije. Da bi se pozvala ova funkcija u aplikaciji je potrebno samo napravit RPC poziv s imenom funkcije.

```
CREATE OR REPLACE FUNCTION get_announcements_with_profiles()
RETURNS json AS $$
    SELECT json_agg(row_to_json(t))
    FROM (
        SELECT
            a.id,
            a.title,
            a.description,
            a.timeposted,
            p."fullName"
        FROM
            public.announcements a
        JOIN
            public.profiles p ON a.postedby = p.id
    ) t;
$$ LANGUAGE sql;
```

Programski kod 4.27*. SQL funkcija za vraćanje podataka o obavijesti*

Koristeći *fromJson* funkciju podatci se pretvaraju u prikladan objekt te se prikazuju na zaslon aplikacije na isti način kako je već objašnjeno za prikazivanje događaja.

Na strani korisnika obavijesti su postavljene tako samo tako da može vidjeti sadržaj obavijesti. Na slici 4.14 se može vidjeti, ako je obavijest dugačka, korisnik može kliknuti na obavijest koje će ga preusmjeriti na zaslon gdje može vidjeti cijelu obavijest.



Slika 4.14. *Prikaz obavijest sa strane korisnika*

Prilikom kreiranja obavijesti u bazi podataka automatski kada se obavijest spremi, također se nalazi podatak kada je obavijest spremljena. Za prikaz prije koliko vremena je obavijest objavljena koristi se paket *time_ago*.

4.13. Sponzori

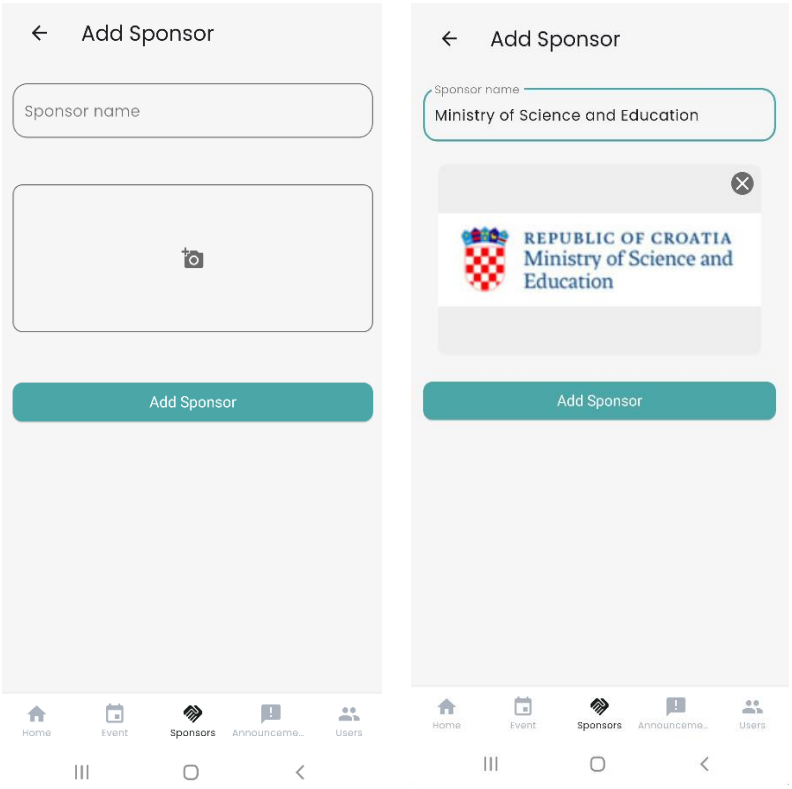
IEEE SST konferencija ima sponzore koji se mogu mijenjati kroz godine. Zbog toga je potrebno također imati sučelje preko kojeg se mogu dodati sponzori. Prilikom kreiranja sponzora potrebno je imati formu s kojom je omogućeno dodavanje slika. Kako bi se ta funkcionalnost postigla potrebno je imati nekoliko stvari; omogućeno spremanje datoteka preko *bucketa* u bazi podataka, te sučelje koje omogućuje pristup galeriji kako bi se izabrala prikladna slika.

Bucket predstavlja određeni spremnik unutar baze podataka koji je odvojen od same tablice. Taj spremnik može sadržavati bilo kakve vrste datoteka koje Supabase podržava. Stoga je prilikom spremanja sponzora u bazu podataka prvo bitno prenijeti sliku u *bucket*, sačekati da se slika prenese i spremiti poveznicu na sliku unutar tablice za sponzor. U programskom kodu 4.28 se može vidjeti implementacija funkcionalnosti za prenošenje slike u bazu podataka.

```
uploadImage(File image) async {
  try {
    await _supabaseClient.storage.from('images').upload(image.path, image);
    return _supabaseClient.storage.from('images').getPublicUrl(image.path);
  } catch (e) {
    throw Exception(e);
  }
}
```

Programski kod 4.28. *Prijenos slika na bazu podataka*

Bitno je uzeti u obzir da slike su uvijek velike količine podataka koje se trebaju prenijeti, stoga zahtijevaju više vremena. Dok se slika prenosi, sve ostale tipke na zaslonu sa slike 4.15 su onemogućene i prilikom izlaska iz tog zaslona, prijenos se zaustavlja.



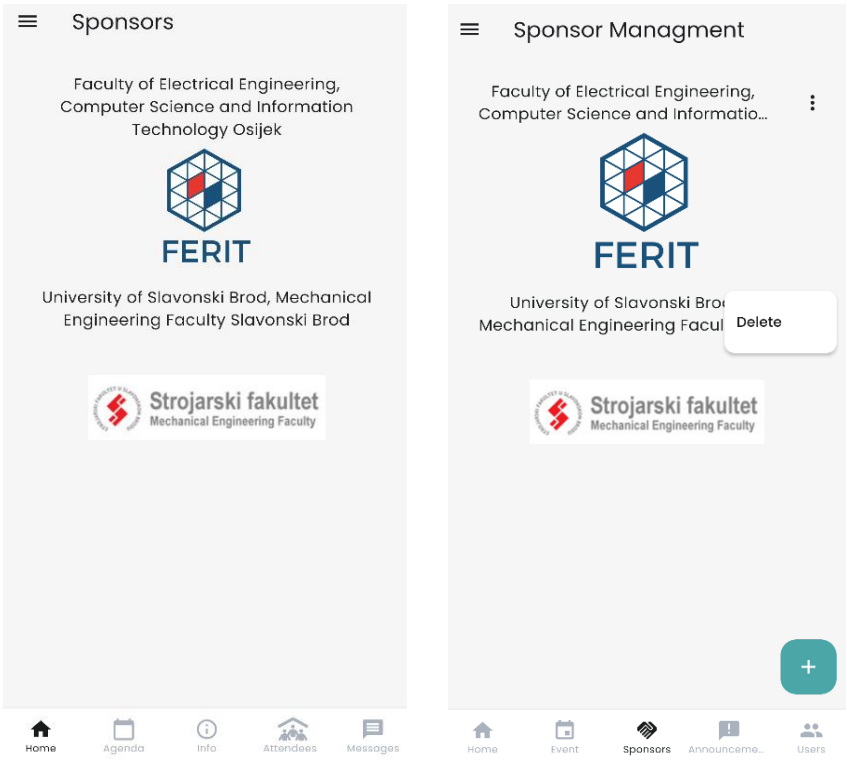
Slika 4.15. *Prikaz sučelja za dodavanje sponzora*

Za izabiranje slika koristiti će se *image_picker* paket koji pruža mogućnost izabiranja slika. Za sliku je potrebno postaviti logo sponzora, stoga će se koristiti način izabiranja slika iz galerije. Također je bitno imati mogućnost brisanja trenutnog odabira slike u slučaju da administrator krivo izabere sliku. Sve što je potrebno je samo pozvati funkciju *pickImage()* kao što je prikazano u programskom kodu 4.29.

```
onPressed: () async {
  final image =
      await ImagePicker().pickImage(source: ImageSource.gallery);
  if (image == null) return;
  if (context.mounted) {
    context
        .read<SponsorFormBloc>()
        .add(SponsorFormEvent.imageChanged(image));
  }
},
```

Programski kod 4.29. *Prikaz koda za biranje slike sponzora*

Zaslon za prikaz na strani administratora sadržava izbornik koji omogućuje brisanje sponzora sa liste i tipku koja preusmjerava administratora na zaslon za dodavanje sponzora kako je prikazano na slici 4.16. Korisnici imaju samo pristup popisu sponzora.



Slika 4.16. *Prikaz sučelja za listu sponzora korisnika i administratora*

4.14. Lokalna pohrana

Podatci se primarno dohvaćaju iz baze podataka, ali to nije uvijek moguće ili potrebno. Korisnik prilikom korištenja aplikacije može izgubiti konekciju s internetom te zbog toga ne može dobiti pristup informacijama. Stoga je potrebno implementirati lokalnu pohranu podataka. Prilikom dohvaćanja podataka, provjerava se postoji li aktivna internet konekcija te se odabire je li potrebno dohvatiti podatke sa poslužitelja ili lokalne pohrane. Kako bi se to postiglo potrebno je koristiti se paket za provjeru internet konekcije i*nternet_connection_cheker* i *Hive* koji omogućuje lokalnu pohranu [11].

Paket *Hive* sprema podatke u pomoću setova ključeva i vrijednosti. Podatci se spremaju i organiziraju pomoću kutija (engl. *Boxes).* Kako bi se podatak zapisao, kutije se moraju asinkrono otvoriti te zapisati podatke unutra. Primjer se može vidjeti u programskom kodu 4.30.

```
// Open your boxes. Optional: Give it a type.
final catsBox = collection.openBox<Map>('cats');

// Put something in
await catsBox.put('fluffy', {'name': 'Fluffy', 'age': 4});
await catsBox.put('loki', {'name': 'Loki', 'age': 2});
```

Programski kod 4.30. *Primjer korištenja lokalne pohrane paketa Hive*

Ova aplikacija ne koristi klasične tipove varijabli za spremanje podataka, već posebne klase. Zbog toga je potrebno napraviti adapter koji će omogućiti da se takvi podatci mogu spremiti u lokalnu pohranu. *Hive* taj proces olakšava s posebnim anotacijama, s kojima generator koda automatski stvara potrebne adaptere. Na već postojeće kako je prikazano u programskom kodu 4.31., klase potrebno je dodati *HiveType i HiveField* anotacije s kojima se identificiraju klase i njene varijable kako bi generator mogao stvoriti potreban adapter.

```
@HiveType(typeId: 4)
@freezed
class Post with _$Post {
  const factory Post({
    @HiveField(0) int? id,
    @HiveField(1) required String title,
    @HiveField(2) required String content,
    @HiveField(3) required String timePosted,
    @HiveField(4) required String fullName,
  }) = _Post;

  factory Post.fromJson(Map<String, dynamic> json) => _$PostFromJson(json);
}
```

Programski kod 4.31. *Prikaz korištenja anotacije za generiranje adaptera*

Nakon toga je potrebno napraviti klasu koja će registrirati potrebne adaptere te upravljati kutijama za svaki od modela podataka. Metode prikazane u programskom kodu 4.32. se pozivaju unutar prikladnog repozitorija za svaki od modela.

```
KeyValueStorage(this._hive) {
  try {
    _hive
      ..registerAdapter(EventAdapter())
      ..registerAdapter(AnnouncementAdapter())
      ..registerAdapter(CommentAdapter())
      ..registerAdapter(PostAdapter())
      ..registerAdapter(ProfileAdapter());
  } catch (e) {
    throw Exception(
        'You shouldn\'t have more than one [KeyValueStorage] instance in your
        'project');
  }
}

final HiveInterface _hive;

Future<Box<Event>> get eventsBox async {
  return _hive.openBox<Event>(_eventsBoxKey);
}
```

Programski kod 4.32. *Prikaz koda za upravljanje pristupom lokalnoj pohrani*

Kada *Bloc* sloj poziva funkciju za dohvaćanje događaja, očekuje listu objekata tog modela. Zadatak repozitorija je da se brine o tome na koji način će dostaviti te podatke. Najčešće se brine o tome koji će se izvor podataka koristiti te o parsiranju podataka na način kako je prikazano u programskom kodu 4.33. Stoga je prvo potrebno napraviti provjeru postoji li konekcija na internet, ako postoji podatci će se dohvaćati s poslužitelja, ako ne postoji podatci će se dohvatiti iz lokalne pohrane.

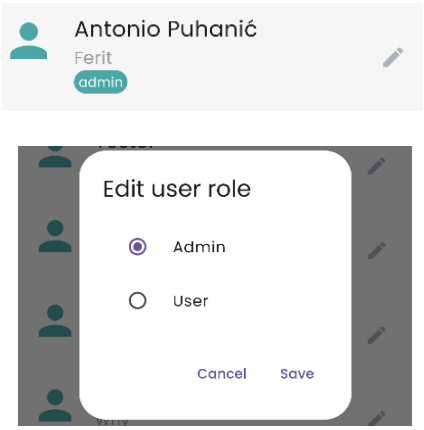
```
Future<List<Event>> getAllEvents() async {
  if (await _connectionChecker.hasConnection) {
    return await _getAllEventsFromClient();
  } else {
    return await _eventLocalStorage.getEvents();
  }
}
```

Programski kod 4.33. *Dohvaćanje podataka ovisno o konekciji na internet*

Kako bi uopće bilo moguće dohvatiti podatke lokalno, potrebno je popuniti lokalnu memoriju. To se postiže tako što se prilikom svakog poziva sa poslužitelja, lokalna pohrana se popunjava ili ažurira s novima ako već postoje.

4.15. Upravljanje korisnicima

Na administratorskoj strani sučelja, potrebno je imati način definiranja uloga korisnika. Za ovu aplikaciju je samo potrebno imati sučelje da se određeni korisnik postavi kao administrator ili da se nekom korisniku maknu administratorska prava. Prikaz tog sučelja se može vidjeti na slici 4.17



Slika 4.16. *Prikaz sučelja za mijenjanje uloga korisnika*

Zbog toga što Supabase paket ima svoju internu logiku upravljanja autentifikacijom, potrebno je promijeniti ulogu na dva mjesta, tablica s popisom korisnika i interna tablica autentifikacije koju Supabase automatski stvara. Interna tablica se ne može mijenjati direktno putem poziva funkcija sa strane klijenta, zato je potrebna posebna funkcija koja će se svaki puta izvršiti kada se promjeni uloga u tablici s popisom korisnika. U programskom kodu 4.34 se može vidjeti implementacije te funkcije koristeći SQL okidače.

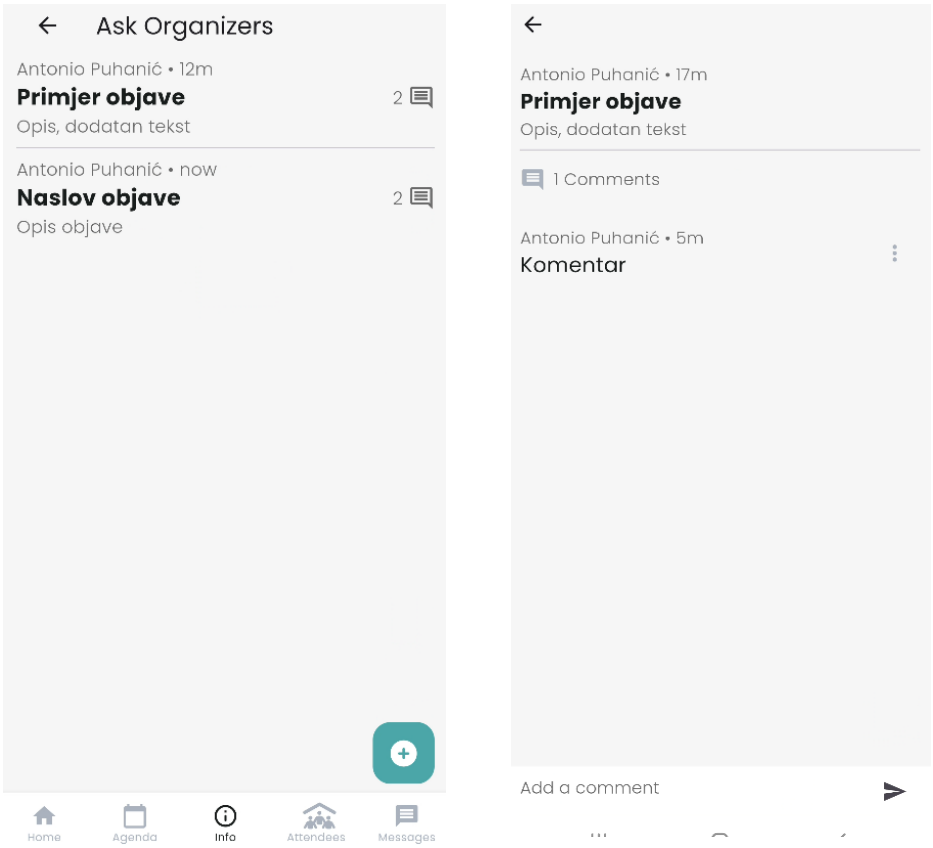
```
CREATE OR REPLACE FUNCTION update_auth_metadata()
RETURNS TRIGGER
SECURITY DEFINER
AS $$
BEGIN

  UPDATE auth.users
  SET raw_user_meta_data = jsonb_set(
    COALESCE(raw_user_meta_data, '{}'::jsonb),
    '{role}',
    to_jsonb(NEW.role)
  )
  WHERE id = NEW.id;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER trigger_update_role
AFTER UPDATE OF role
ON profiles
FOR EACH ROW
EXECUTE FUNCTION update_auth_metadata();
```

Programski kod 4.34. *Funkcija za mijenjanje uloga*

Kada se izvrši funkcija sa prijašnje slike i podatci se ažuriraju, prilikom sljedeće prijave korisnik kojemu su promijenjena prava će biti preusmjeren na prikladan zaslon, jer se iz tablice za autentifikaciju može pravilno dobiti podatak o ulozi korisnika.

4.16. Objave korisnika

Za ovu aplikaciju i konferenciju je potrebno imati način na koji korisnici mogu međusobno komunicirati. Za to je napravljeno sučelje u obliku objava na koje korisnici mogu odgovoriti. Izgled sučelja se može vidjeti na slici 4.18. Klikom na pojedinu objavu korisnik je preusmjeren na zaslon te objave na kojemu se nalazi njen sadržaj i popis svih komentara tj. odgovora na tu objavu. Oba dva zaslona su implementirani na sličan način kao i ostali zasloni uz nekoliko manjih razlika. Glavna razlika je što je za zaslon na kojemu se pišu komentari potrebno maknuti donju navigacijsku traku, te postaviti formu za unos komentara.

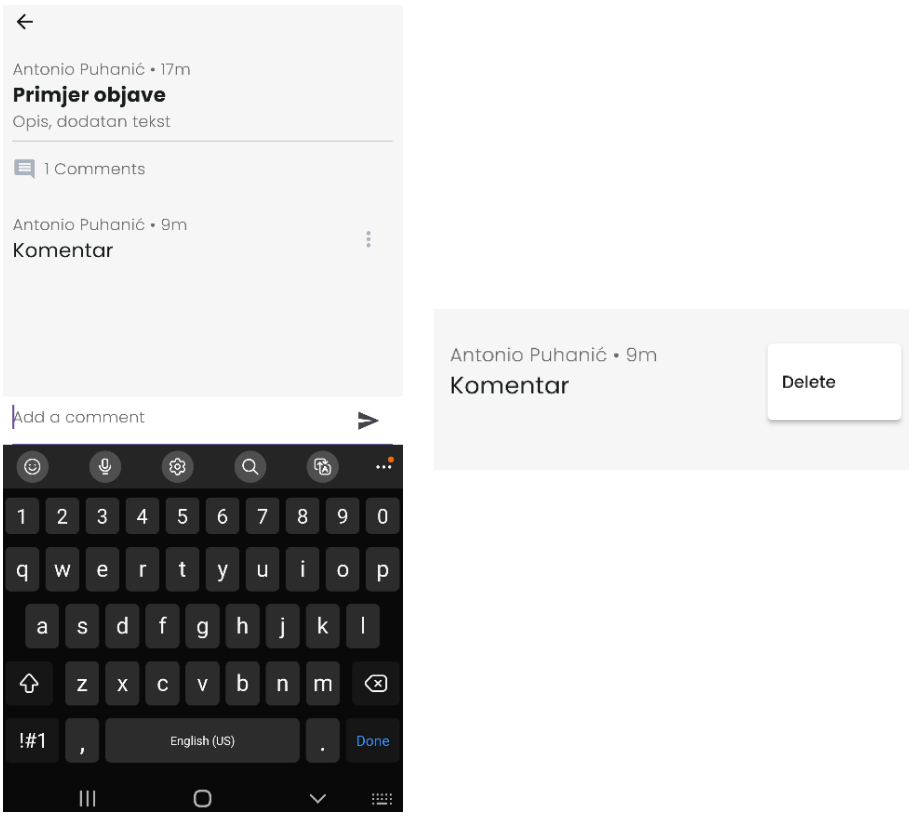


Slika 4.18. *Izgled sučelja za objave i komentare korisnika*

Na lijevom dijelu slike 4.18. može se vidjeti popis objava gdje svaka objava sadržava naslov, opis, ime korisnika i vrijeme objave. Na desnoj strani se nalazi sučelje pojedinačne objave, gdje se na vrhu zaslona nalazi objava, a ispod objave komentari. Prilikom klika na formu za dodavanje

komentara, otvara se tipkovnica gdje korisnik upisuje komentar. Kada se klikne na tipku za slanje, tipkovnica se zatvara i forma za komentar se resetira, te se u bazu podataka sprema komentar.

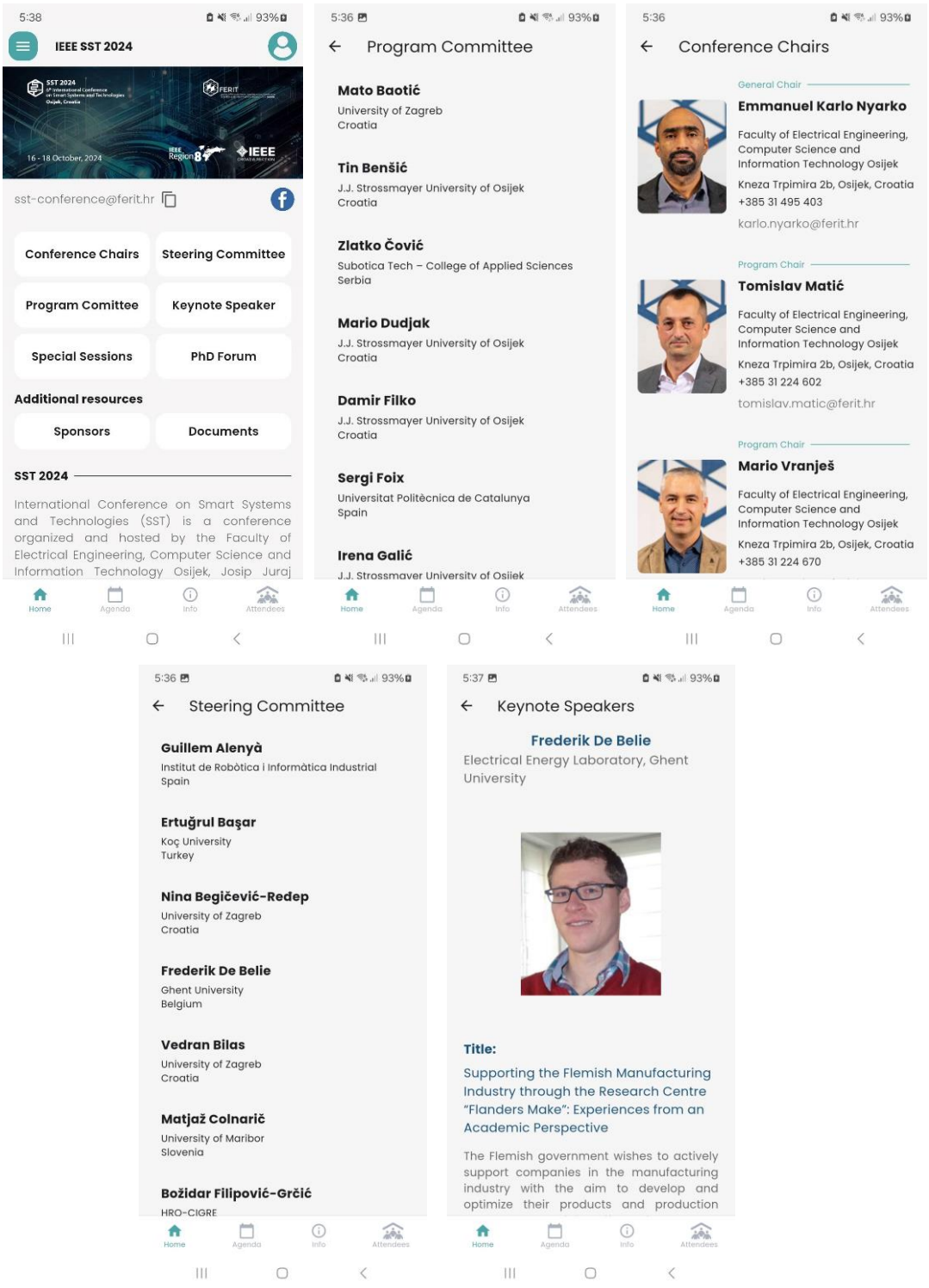
Kako bi sučelje bilo jednostavno za koristiti, forma za unos komentara se automatski podiže iznad tipkovnice. Prijavljeni korisnici mogu u bilo kojem trenutku obrisati samo svoj komentar klikom na meni desno od sadržaja komentara kako je prikazano na slici 4.19.



***Slika 4.19**. Prikaz sučelja za unos i brisanje komentara*

4.17. Zasloni za informacije o konferenciji

IEEE SST konferencija ima svoju web stranicu na kojoj se nalaze sve informacije o samoj konferenciji. Kako bi se olakšalo pristup nekim informacijama, sa te web stranice su uzete informacije i napravljeno je nekoliko različitih zaslona za prikazivanje potrebnog sadržaja. Zasloni koji će se dodati su „*Conference chairs*“, „*Steering committee*“, „*Program committe“, „Keynote speakers“, „Special sessions“* i *„PhD Forum“*. Kako bi se to postiglo na početnom dijelu aplikacije postaviti će se tipke za svaki od zaslona što je potrebno napraviti. Izgled ovih zaslona se može vidjeti na slici 4.20.



Slika 4.20. *Prikaz početnog zaslona i ostalih zaslona za informacije*

Budući da su ovo zasloni koji imaju statične informacije koje se neće mijenjati s vremenom, nije ih potrebno spremati u bazu podataka. Dovoljno je imati *„.json"* datoteke koje će držati potreban sadržaj. Za potrebe čitanja podataka treba imati način na koji će se podatci pretvoriti u prikladni objekt za prikazivanje sadržaja na zaslon mobitela. Implementacija za čitanje tih podataka se može

vidjeti u programskom kodu 4.35. Nakon toga je samo potrebno postaviti klasične *widgete* koji su već dosad korišteni kako bi se podatci prikazali na zaslon.

```
Future<List<ConferenceChair>> loadConferenceChairs() async {
  try {
    final jsonString =
        await rootBundle.loadString('assets/json/conference_chairs.json');
    final Map<String, dynamic> jsonMap = jsonDecode(jsonString);
    final List<dynamic> jsonList =
        jsonMap['conference_chairs'] as List<dynamic>;
    return jsonList.map((e) {
      return ConferenceChair.fromJson(e as Map<String, dynamic>);
    }).toList();
  } catch (e) {
    debugPrint(
        'Error loading conference chairs: $e');
    return [];
  }
}
```

Programski kod 4.35 *Kod za pretvaranje „.json“ podataka u prikladne modele*

5. Zaključak

Izradom ovog projekta postignuta je funkcionalna mobilna aplikacija za upravljanje događajima za IEEE SST konferencije. Cilj je bio napraviti aplikaciju koja olakšava korisnicima pristup informacijama o konferenciji kroz sistem agende, objava i obavijesti. Problem koji je riješen ovom aplikacijom je jednostavnost pregleda događaja i izmjene informacija između korisnika. Moderni elementi korisničkog sučelja olakšavaju korisnicima navigaciju i korištenje same aplikacije.

Implementacija je provedena koristeći Flutter razvojno okruženje i Supabase platformu, čime je omogućena visoka razina skalabilnost za buduće korištenje i proširivanje aplikacije. Flutter se pokazao kao izvrsno razvojno okruženje u kojemu je bilo lagano realizirati aplikaciju. Pružilo je mnoge pakete koje su smanjili vrijeme potrebno da bi se različiti dijelovi aplikacije izradili.

Supabase se također pokazao kao izvrstan alat za razvoj baze podataka i ostalih usluga za poslužitelje. Uvijek je dostupan te omogućava vrlo lako povezivanje s klijentskim uređajima i razmjenu podataka. Već postojeći sistem Supabase autentifikacije je pomogao prilikom izrade prijave zaslona za prijavu i registraciju.

Razvoj ove aplikacije je pokazao kako korištenje modernih alata i paketa mogu olakšati cjelokupan proces izrade aplikacije koje su skalabilne, brze i prilagodljive. U budućnosti se mogu izraditi dodatne funkcionalnosti i servisa poput višejezičnosti, kalendara te dodatne opcije za personalizaciju profila i aplikacije kako bi se povećala korisnost i atraktivnost aplikacije.

LITERATURA

[1] Whova, "Whova" [Mrežno]. Dostupno: https://whova.com/. [lipanj 2024].

[2] Yapp, "Yapp" [Mrežno]. Dostupno: yapp.us. [svibanj 2024].

[3] Perfect Venue, "Perfect Venue" [Mrežno]. Dostupno: https://www.perfectvenue.com/. [lipanj 2024].

[4] Google, "Flutter" [Mrežno]. Dostupno: flutter.dev. [svibanj 2024].

[5] Google, "Dart" [Mrežno]. Dostupno: https://dart.dev/. [lipanj 2024].

[6] Supabase,"Supabase"[Mrežno]. Dostupno: https://supabase.com/docs. [svibanj 2024].

[7] P. Tyagi, "Pragmatic Flutter", CRC Press, 2021.

[8] BLoC, "BLoC" [Mrežno]. Dostupno: https://bloclibrary.dev/ [svibanj 2024].

[9] R. Rose, "Flutter and Dart Cookbook", O'Reilly Media, 2023.

[10] R. T. Team, Real-World Flutter by Tutorial, Razeware LLC, 2022.

[11] Hive, "Hive", [Mrežno]. Dostupno: https://docs.hivedb.dev/. [kolovoz 2024].

SAŽETAK

Bez obzira na već veliki broj postojećih rješenja za upravljanje događajima, nekada je potrebno imati aplikaciju koja će biti specifično namijenjena potrebama određenih događaja. Stoga je u okviru ovog rada napravljena mobilna aplikacija za upravljanje događajima IEEE SST konferencije. Aplikacija je realizirana koristeći dvije glavne tehnologije, Flutter i Supabase. U ovom radu je objašnjeno kako navedene tehnologije funkcioniraju i na koji način je aplikacija realizirana. Aplikacija omogućuje autentifikaciju korisnika, stvaranje događaja, pregled događaja, pregled obavijesti, objavljivanje i komentiranje. Najvažnija značajka je pregled i filtriranje događaja koje korisnik može označiti da će ići. Svi podatci se spremaju u Supabase bazu podataka.

Ključne riječi: Bloc, Dart, događaj, Flutter, objava, Supabase

ABSTRACT

Regardless of the already large number of existing event management solutions, sometimes it is necessary to have an application that will be specifically intended for the needs of certain events. Therefore, in this project, a mobile application was created to manage the events of the IEEE SST conference. The application was realized using two main technologies, Flutter and Supabase. This paper explains how the aforementioned technologies work and how the application was implemented. The application allows user to login and register, event creation, event viewing, notifications, posting and commenting. The most important feature is the event managment. All data is saved in the Supabase database.

Keywords: Bloc, Dart, event, Flutter, post, Supabase

PRILOZI

Prilog P.1

Kod mobilne aplikacije nalazi se na poveznici: https://github.com/APuhanic/IEEE-SST