Izrada aplikacije za pronalazak kućnog majstora korištenjem Flutter razvojnog okvira

Ladavac, Dino

Sveučilište u Rijeci, Fakultet informatike i digitalnih tehnologija

Sveučilišni prijediplomski studij Informatika

Dino Ladavac

Izrada aplikacije za pronalazak kućnog majstora korištenjem Flutter razvojnog okvira

Završni rad

Mentor: izv. prof. dr. sc. Marija Brkić Bakarić

Rijeka, 15. Rujan 2023

Sveučilište u Rijeci
FIDIT Fakultet informatike i digitalnih tehnologija
www.inf.uniri.hr

Rijeka, 26.5.2023.

Zadatak za završni rad

Pristupnik: Dino Ladavac

Naziv završnog rada: Izrada aplikacije za pronalazak kućnog majstora korištenjem Flutter razvojnog okvira

Naziv završnog rada na engleskom jeziku: Development of a Flutter application for searching for a local handyman

Sadržaj zadatka: Cilj rada je istražiti Flutter kao razvojni okvir, njegovu jednostavnost korištenja te konkurentnost na trenutnom tržištu višeplatformskih razvojnih okvira. U uvodnom djelu se uspoređuju karakteristike Flutter-a s ostalim popularnim razvojnim okvirima. Nakon toga detaljno se prikazuje i opisuje izrada aplikacije za pronalazak kućnog majstora unutar lokalnog područja. U istom dijelu prikazani su programski dijelovi koda kao i datoteke koje tvore samu aplikaciju. U zaključku su objedinjene glavne prednosti i nedostatci pri korištenju razvojnog okvira Flutter te su s obzirom na njih iznesene preporuke za njegovo korištenje.

Voditelj za završne radove

Doc. dr. sc. Miran Pobar

Mentor

Izv. prof. dr. sc. Marija Brkić Bakarić

Zadatak preuzet: 26.5.2023.

Dino Ladavac

(potpis pristupnika)

Adresa: Radmile Matejčić 2
51000 Rijeka, Hrvatska
Tel: +385(0)51 584 700
E-mail: ured@inf.uniri.hr
OIB: 64218323816
IBAN: HR1524020061400006966
uniri

Sažetak

U ovom završnom radu prikazana je izrada aplikacije za pronalazak kućnog majstora korištenjem razvojnog okvira *Flutter*. Kroz prikaz izrade aplikacije predočeni su glavni aspekti razvojnog okvira, korištene datoteke te su pojašnjeni pojedini dijelovi programskog koda potrebni za izgled i funkcionalnost konačne aplikacije. Aplikacija, uz dodatne *Flutter*-ove pakete, koristi *Cloud Firestore* - bazu podataka pohranjenu u oblaku za pohranu podataka unesenih u aplikaciji te *Firestore*-ov sustav za autentifikaciju korisnika. Aplikacija je predviđena za dva tipa korisnika – korisnici koji šalju zahtjeve za obradu željenog servisa te kućne majstore koji prihvaćaju zahtjev te obavljaju posao uz predloženu cijenu. Korisnicima se omogućuje registracija, prijava, stvaranje novih, prikaz i brisanje prethodnih zahtjeva te izmjena osobnih podataka, dok se kućnim majstorima omogućuje registracija, prijava, prikaz zahtjeva za servise u području kojem su specijalizirani te prihvaćanje postojećih zahtjeva uz navođenje cijene.

Ključne riječi

Aplikacija, Flutter razvojni okvir, Cloud Firestore, sustav za autentifikaciju korisnika, Flutter paketi, kućni majstor

Sadržaj

Uvod

U ovom završnom radom prikazan je proces izrade aplikacije za pronalazak kućnog majstora u *Flutter* razvojnom okviru kako bi se na kvalitetan način predočio potpun proces stvaranja aplikacija za svakodnevnu uporabu. Odabran je *Flutter* zbog svoje jednostavnosti kao i lakoće učenja *Dart* programskog jezika što omogućava brzu izradu gotovog proizvoda i lak prelazak sa druge programske podrške (Montaño, 2023). Na tržištu je trenutno moguće pronaći veliki broj razvojnih okvira kao što su *React Native, Cordova, Ionic, NativeScript, Xamarin, Kotlin, Unity* te ostali, manje popularni razvojni okviri s kojima *Flutter* konkurira.

Obzirom da se radi o višeplatformskom (*engl .Cross-platform*) razvojnom okviru, podržana je izrada aplikacija za korištenje na mobilnim operacijskim sustavima (poput *Androida* ili *iOS*-a) i web platformama korištenjem istog programskog koda, odnosno *Flutter* ne zahtjeva izradu različitih programskih kodova za različite platforme što mu daje prednost nad konkurentima poput *IOS SDK, Microsoft Windows SDK* i ostalih.

Iako, je kao razvojni okvir poprilično jednostavan, *Flutter* se ne preporuča za izradu kompliciranijih aplikacija koje pohranjuju i čitaju veliki broj podataka u bazi stoga je za izvođenje nekih dijelova programskog koda potrebno duže vrijeme od većine razvojnih okvira s kojima konkurira na tržištu.

Neovisno o tome, stekao je visoku popularnost te je trenutno na vrhu liste najkorištenijih okvira za razvoj softvera (Vailshery, 2023) . O popularnosti ovog razvojnog okvira pokazuje i veliki broj modernih aplikacija izrađenih u *Flutter-u* koje koriste *Google, SpaceX, eBay* i ostali (Khomutova, 2023).

U nastavku biti će prikazan proces postavljanja *Flutter* projekta, instalacija potrebnih paketa, spajanje sa bazom podataka te izrada i izgled same aplikacije u postepenim koracima.

1. Priprema radnog okruženja

1.1. Postavljanje radnog okruženja i stvaranje projekta

Kako bi mogli početi sa izradom aplikacije, najprije je potrebno postaviti radno okruženje i instalirati potrebne datoteke. Sa službene stranice *Flutter*[1]-a potrebno je preuzeti i instalirati službenu podršku za razvoj softvera (*engl. Software development kit, SDK*). Nakon instalacije, potrebno je pokrenuti razvojno okruženje. U ovom završnom radu aplikacija će se izrađivati u *Visual Studio Code*-u.

Na željenoj lokaciji, pomoću *Command Prompta* ili *terminala* upisujemo naredbu:

$$flutter\ create\ handyman\_finder\_app$$

Ova naredba stvara stablo datoteka koje tvori strukturu aplikacije naziva *handyman_finder_app.* Nakon pokretanja naredbe, stablo datoteka trebalo bi imati nekoliko datoteka i mapa.

1.2. Stablo mapa i datoteka

Mape *.dart_tool* i *.idea* su mape koje sadrže konfiguracijske datoteke koje pohranjuju informacije poput korištenih paketa u aplikaciji i konfiguracijskih datoteka za integrirano okruženje.

Nadalje, mape *android, ios, macos, linux, windows* i *web* sadrže konfiguracije za stvaranje i pokretanje aplikacije na zasebnoj platformi. Primjerice u mapi *web* nalazi se konfiguracija za pokretanje web aplikacije.

U mapi *build* nalaze se fragmenti aplikacije stvoreni u procesu izrade i finalni fragmenti stvoreni nakon izrade aplikacije. Ovdje je pohranjen kompajliran kod, resursi kao i međurezultati dobiveni prilikom sastavljanja aplikacije.

[1] Službena stranica nalazi se na poveznici u izvorima

Mapa *test* služi kao mjesto pohrane programskih datoteka koje pokreću automatske testove na gotovoj aplikaciji. Ovaj se dio uvelike koristi kod kreiranja većih aplikacija gdje je potrebno stvoriti veliki broj podataka te ih provesti kroz aplikaciju kako bi se utvrdilo dolazi li negdje do prekida rada aplikacije ili do greška pri izvođenju.

Programski kod aplikacije pisan u *dart*-u biti će pohranjen u mapi *lib*. Svaki pojedini zaslon aplikacije ima svoju *dart* datoteku u kojoj se nalazi njegova struktura i programska logika te se komunikacija između njih vrši unutar ove mape. Pri stvaranju nove aplikacije, u ovoj mapi nalazi se samo *main.dart* datoteka koja pokreće samu aplikaciju. Sve ostale programske datoteke stvara programer.

Izvan mapa nalazi se još nekoliko konfiguracijskih datoteka, no valja istaknuti *pubspec.yaml* datoteku u kojoj navodimo korištene pakete i zavisnosti sa njihovim verzijama.

1.3. Postavljanje i spajanje baze podataka

Kako aplikaciju koriste dvije vrste korisnika, ona treba sadržavati nekakvu prijavu u sustav da bi se moglo razlikovati običnog korisnika od kućnog majstora. Kod prijavljivanja u sustav, aplikacija pretražuje među pohranjenim podacima registrirane korisnike i provjerava ukoliko postoji njihova elektronička adresa kojom se prijavljuju te ukoliko je lozinka za tu adresu točna. Da bi se podaci mogli sigurno i logički pohraniti, potrebno je postaviti bazu podataka.

Uz sustave za bazu podataka, potrebno je odabrati i odgovarajući sustav za provjeru autentifikacije. Zbog svoje jednostavnosti prilikom postavljanja i korištenja te kompatibilnosti sa *Flutterom* i sustavom za autentifikaciju, odabrana je *Cloud Firestore* baza podataka. Ova baza podataka nerijetko se koristi u kombinaciji sa *Flutterom* jer sadrži gotove pakete za njeno korištenje, ali i zbog same fleksibilnosti baze podataka koja omogućuje spremanje više tipova podataka na kvalitetan i uređen način. Također, ova baza podataka omogućuje stvaranje kolekcija što je za ovaj tip aplikacije potrebno jer će se spremati tri skupine podataka u kolekcije. Prvu kolekciju činit će podaci stalnih korisnika, drugu kolekciju činit će podaci kućnih majstora te treću kolekciju podaci pojedinih zahtjeva koje postavljaju korisnici.

Kako bi mogli stvoriti novu bazu podataka, potrebno je posjetit službenu stranicu *Firebase*-a[2] te se registrirati ili prijaviti sa *google računom*. Nakon prijave, pojavljuje se stranica na kojoj možemo kreirati novi projekt kojem određujemo ime, identifikator te platforme na kojima se predviđa koristiti. Unutar projekta, na navigacijskoj traci odabire se odjeljak „*Firestore Database*“ koji stvara bazu podataka pohranjenu u oblaku. Kako bi se aplikacija mogla pokretati na *android* mobilnim uređajima potrebno je preuzeti *google-services.json* datoteku te je spremiti u projekt unutar *android/app* mape. Datoteku je moguće pronaći u postavkama baze podataka gdje se nalaze i ime projekta, identifikator i Web API ključ[3].

Unutar projekta potrebno je pronaći *pubspec.yaml* datoteku u koju se unose zavisnosti za bazu podataka uz njihove verzije korištenja. Zavisnosti se unose pod sekciju *dependencies*.

```
dependencies:
  flutter:
    sdk: flutter
  firebase_core: ^2.4.0
  cloud_firestore_platform_interface: ^5.9.1
  cloud_firestore: ^4.2.0
```

Slika 1 - Prikaz unosa zavisnosti za Cloud Firebase bazu podataka

Nakon unosa potrebnih zavisnosti pokreće se naredba *flutter pub get* koja priprema zavisnosti za korištenje.

Sada je u *main.dart* datoteci potrebno konfigurirati kojom bazom podataka upravljamo. Ovdje se postavljaju Web API ključ, identifikator projekta i ostali osjetljivi[4] podaci baze podataka koje je moguće pronaći u postavkama.

[2] Službena stranica nalazi se na poveznici u izvorima
[3] Web API ključ – „jedinstveni identifikator koji omogućuje izvođenje API poziva. API je skraćenica sučelje za programiranje aplikacija“ (Reichert, 2020)
[4] Ukoliko se projekt javno objavljuje, potrebno je obrisati ili sakriti ove podatke kako ne bi došlo do napada na bazu podataka.

Slika 2 - Konfiguracija baze podataka u main.dart datoteci

Prilikom rada s bazom podataka i dohvaćanjem podataka iz baze, potrebno je koristiti tip podatak *Future* koji sugerira programu kako će se nekad u budućnosti nakon izvršavanja zadatka u ovu varijablu pohraniti neka vrijednost. Ovo pri izvedbi, sprječava program da ispiše grešku zbog nepostojeće vrijednosti ili pohranjene *null* vrijednosti.

2. Izrada aplikacije

2.1. main.dart

Kao što je već napomenuto, *main.dart* datoteka služi za pokretanje aplikacije. Ukoliko se trenutno pokrene aplikacija, prikazat će se bijela pozadina sa svijetlo plavom navigacijskom trakom što je znak da aplikacija radi te da ju je potrebno strukturirati. Unutar *MyApp* klase moguće je stvoriti varijablu u koju će se pohraniti tema aplikacije. Dobra praksa je temu aplikacije u *Flutter-u* zadati odmah na početku kako bi se kasnije lakše usklađivali naknadno dodani elementi.

```
class MyApp extends StatelessWidget {
  final ThemeData themeM = ThemeData(
      primaryColor: const Color.fromARGB(255, 209, 24, 11),
      scaffoldBackgroundColor: Colors.white,
      appBarTheme:
          const AppBarTheme(backgroundColor: Color.fromARGB(255, 209, 24, 11)),
      textTheme: const TextTheme(bodyMedium: TextStyle(color: Colors.black))); // ThemeDa

  MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: const LoginPage(),
      theme: themeM,
      debugShowCheckedModeBanner: false,
    ); // MaterialApp
  }
}
```

Slika 3 - Programski kod main.dart datoteke

Temu se zadaje pomoću *ThemeData()* klase kojom je moguće definirati temu teksta, gumbova i boja aplikacije tako da se na primjer u parametar *primaryColor* postavlja primarna boja aplikacije. Boje aplikacije moguće je zadati pomoću dvije klase:

Klasa *Color* koja se koristi u kombinaciji predefiniranih funkcija. Ona očekuje unos brojčanih varijabli kao vrijednosti RGB spektra.

Klasa *Colors* koja ima predefinirane boje te očekuje samo njihov poziv.

Parametrom *ScaffoldBackgroundColor* postavlja se pozadinsku boju klase *Scaffold* koja će se naknadno koristiti za strukturiranje samih zaslona aplikacije. *appBarTheme* je parametar kojim se definira tema navigacijske trake aplikacije, a parametrom *textTheme* postavlja se tema tekstnih elemenata u aplikaciji.

Dart je objektno-orijentirani programski jezik stoga zahtjeva od svake klase da sadrži svoj konstruktor, pa ga je u nastavku klase potrebno definirati. Unutar definicije konstruktora poziva se *super.key* koji osigurava poziv klase roditelja.

Osim konstruktora, potrebno je definirati i *build* metodu koja stvara i vraća *widget* (izv. engl.) odgovoran za stvaranje korisničkog sučelja aplikacije. Postoje dvije vrste *widget*-a:

Stateless widget – koristi se kod kreiranja korisničkog sučelja koji se ne mijenja nakon njegovog stvaranja (Mike Katz, 2022).

Stateful widget – koristi se kod kreiranja korisničkog sučelja koji prilikom korištenja koristi i mijenja interaktivna stanja te tako stvara dinamičko sučelje (Mike Katz, 2022).

Main.dart datoteka ne sadrži nikakve promjene stanja stoga je praksa koristiti *Stateless widget.* Unutar *widget-a* pozvana je klasa *MaterialApp()*[5] koja služi za stvaranje aplikacija čiji je dizajn lako prilagodljiv i izmjenjiv. Ovdje se poziva parametar *theme* kojem se pridružuje varijabla u kojoj je prethodno pohranjena tema. Uz to definira se *home* koji služi za prikaz ekrana prilikom prvog pokretanja aplikacije. Tom parametru pridružuje se *widget LoginPage()* kojeg naknadno stvaramo i uređujemo.

2.2. login.dart

Prilikom pokretanja aplikacije treba se prikazati zaslon za prijavu kako bi se moglo razlikovati korisnike koji postavljaju zahtjev od kućnih majstora koji prihvaćaju zahtjev i obavljaju dogovoren posao. U *main.dart* datoteci već je pozvan *LoginPage() widget,* što znači da ga je sada samo potrebno definirati.

[5] Prilikom pokretanja aplikacije, u gornjem desno kutu pojavljuje se znak *„Debug“* kojeg je moguće ukloniti parametrom *debugShowCheckedModeBanner* kao gore na slici

Unutar *lib* mape potrebno je stvoriti datoteku *login.dart* koja će sadržavati spomenuti *widget.* Sama prijava koristi unos teksta i provjeru u bazi podataka što zahtjeva uporabu stanja *widget-a* i definiranje *StatefulWidget-a,* no ostatak *„stranice"* (poput navigacijske trake) ne zahtijeva nikakve promjene stanja. Zbog uštede prostora i povećanja brzine aplikacije, kreira se *StatelessWidget* koji predstavlja cijeli prikaz stranice te se njemu kao dijete dodjeljuje *StatefulWidget* samo za one dijelove koji koriste promjenu stanja.

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: const Text("Log in")),
        body: Padding(
          padding: const EdgeInsets.all(16.0),
          child: LoginForm(context: context),
        )); // Padding // Scaffold
  }
}
```

Slika 4 - LoginPage() widget

Ovaj *widget* poziva ranije spomenutu *Scaffold()* klasu koja stvara strukturu aplikacije. Tako se parametrom *appBar* može odrediti da se na trenutnom prikazu aplikacije prikazuje navigacijska traka s naslovom „*Log in*", a da će se u tijelu aplikacije prikazivati *widget LoginForm().* Klasom *Padding()* postavljamo poravnanja.

LoginForm() je *widget* koji zahtjeva promjenu stanja, a sastoji se od forma za unos elektroničke pošte i lozinke. Nakon što korisnik unese tražene podatke, ovaj *widget* treba provjeriti nalaze li se uneseni podaci u bazi podataka.

```dart
class LoginForm extends StatefulWidget {
  const LoginForm({required this.context, super.key});
  final BuildContext context;

  @override
  // ignore: library_private_types_in_public_api
  _LoginFormState createState() => _LoginFormState();
}

class _LoginFormState extends State<LoginForm> {
  final key = GlobalKey<FormState>();
  String email = '';
  String password = '';
```

Slika 5 - Stvaranje LoginForm() widget-a

StatefulWidget stvara se tako da se unutar njega pozivaju *„stanja“* koja vrše programski dio provjere, unosa i ostalih metoda koje zahtijeva neku izmjenu. Očekivano je da se kod prijave unose samo elektronička pošta i lozinka stoga se definiraju potrebne varijable. Varijabla *Key* omogućuje da forma spremi vrijednosti pri pritisku gumba.

```dart
@override
Widget build(BuildContext context) {
  return Form(
    key: key,
    child: Column(children: [
      TextFormField(
        decoration: const InputDecoration(labelText: "Enter Email"),
        validator: (value) {
          if (value!.isEmpty) {
            return "Email required!";
          }
          return null;
        },
        onChanged: (value) {
          email = value;
        },
      ), // TextFormField
```

Slika 6 - LoginForm() widget

Kako klasa *Form()* ne može sadržavati više od jednog djeteta (što bi značilo da se unutar forme može prikazati samo primjerice forma za unos lozinke), potrebno je kao njeno dijete dodati klasu *Column()*. Ova klasa sadrži atribut *children* kojim se u obliku polja omogućuje dodavanje više djeteta.

Kao jednog od djeteta, definira se klasa *TextFormField()* što u aplikaciji prikazuje polje za unos teksta. Kao dekoracija postavljen je tekst „*Enter Email*“ te se tako korisniku predočuje koja je uloga ovog polja za unos. Pomoću parametra *validator* omogućuje se provjera unosa teksta te ako on nije unesen ispisuje se odgovarajuća poruka. U suprotnom u

varijablu *email* sprema se unesen tekst. Sada se na isti način ispunjava klasa *TextFormField()* za unos lozinke.

Kako bi se potvrdio unos podataka, potrebno je dodati gumb za prijavu klasom *ElevatedButton()*. Ova klasa sadrži parametar *onPressed* u kojem je moguće definirati što se izvodi prilikom pritiska na gumb. Najprije se uz pomoć ključa forme provjerava ukoliko su uneseni svi podaci te se zatim poziva funkcija *loginUser* koju je naknadno potrebno definirati. Parametrima *style* definiramo stil gumba, a *child* služi kako bi gumbu dodali odgovarajući tekst.

```
const SizedBox(height: 20),
ElevatedButton(
  onPressed: () {
    if (key.currentState!.validate()) {
      loginUser(email, password, context);
    }
  },
  style: ElevatedButton.styleFrom(
      backgroundColor: const Color.fromARGB(255, 47, 45, 45)),
  child: const Text("Log in"),
), // ElevatedButton
```

Slika 7 - Dodavanje gumba za prijavu

Osim prijave, korisniku treba omogućiti mogućnost registracije. Kako bi se razlikovala registracija kućnog majstora od korisnika koji šalje zahtjev, dodaju se dva različita gumba za registraciju. Unutar *onPressed* parametra koristi se *Navigator* klasa kojom se uz pomoć *push* metode otvara novi prikaz. Ovdje će se pozvati dva različita *widget-a*, svaki za zasebni način registracije korisnika. Osim gumbova, treba dodati standardne tekstove koji opisuju funkciju gumba te se za to koriste klase *Text*. Valja napomenuti da *SizedBox* klasa stvara virtualnu kutiju. Ukoliko njoj ne dodamo nikakvo dijete, tada ta kutija služi samo kao separator između gumbova kako oni ne bi bili prilijepljeni.

```
const SizedBox(height: 20),
const Text("Not a member? Try registering first!"),
const SizedBox(height: 20),
ElevatedButton(
  onPressed: () {
    Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) => const RegistrationPage())); // Materi
  },
  style: ElevatedButton.styleFrom(
      backgroundColor: const Color.fromARGB(255, 47, 45, 45)),
  child: const Text("Register"),
), // ElevatedButton
const SizedBox(height: 20),
const Text("Want to register your workspace?"),
const SizedBox(height: 20),
ElevatedButton(
  onPressed: () {
    Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) => const HandymanRegistrationPage()));
  },
  style: ElevatedButton.styleFrom(
      backgroundColor: const Color.fromARGB(255, 47, 45, 45)),
  child: const Text("Register as Handyman"),
), // ElevatedButton
```

Slika 8 - Gumbovi za registraciju

2.2.1. Autentifikacija korisnika i *loginUser* metoda

Kako bi se mogla izvoditi autentifikacija korisnika, *Flutter* i *Firestore* razvili su paket koji omogućuje ovaj proces na pojednostavljeni način. Unutar *pubspec.yaml* pod zavisnosti, potrebno je nadodati *firebase_auth: ^4.2.1* te ponovno pokrenuti naredbu *flutter pub get*. Na ovaj način priprema se paket autentifikacije korisnika za korištenje. U *login.dart* sada je potrebno pozvati ovaj paket dodavanjem linije *import 'package: firebase_auth/ firebase_auth.dart'*.

Ranije se u kodu gumba za prijavu korisnika spominje metoda *loginUser* koja služi za prijavu korisnika u sustav koju je sada potrebno definirati. Funkcija direktno komunicira s bazom podataka stoga je ona tipa *Future<void>*. Unutar parametara metode navode se oni podaci koje korisnik unosi kod prijave dakle elektronička adresa i lozinka.

Prijava se izvodi metodom *signInWithEmailAndPassword* iz klase *FirebaseAuth* te se u parametre navodi elektronička pošta i lozinka korisnika. Nakon što je prijava izvršena, potrebno je odrediti koja je vrsta korisnika prijavljena. Ukoliko se prijava izvršila u kolekciji *users*, onda se radi o korisniku koji postavlja zahtjeve u aplikaciji, a ako je prijava izvršena u kolekciji *handyman* onda se radi o kućnom majstoru te obzirom na vrstu korisnika otvara se

odgovarajući prikaz naslovne stranice. U cijelom tom procesu, dohvaća se *korisničko ime* korisnika te se pohranjuje za daljnje korištenje. U nastavku je prikazan programski kod za prijavu korisnika te se na isti način izvodi prijava za kućnog majstora.

```
Future<void> loginUser(
    String email, String password, BuildContext context) async {
  try {
    UserCredential userCredential = await FirebaseAuth.instance
        .signInWithEmailAndPassword(email: email, password: password);

    if (userCredential.user != null) {
      String userId = userCredential.user!.uid;
      DocumentSnapshot userDoc = await FirebaseFirestore.instance
          .collection('users')
          .doc(userId)
          .get();
      if (userDoc.exists) {
        String username = userDoc.get("username");
        // ignore: use_build_context_synchronously
        Navigator.pushReplacement(
            context,
            MaterialPageRoute(
                builder: (context) => MainPage(username: username))); // Mat
      }
```

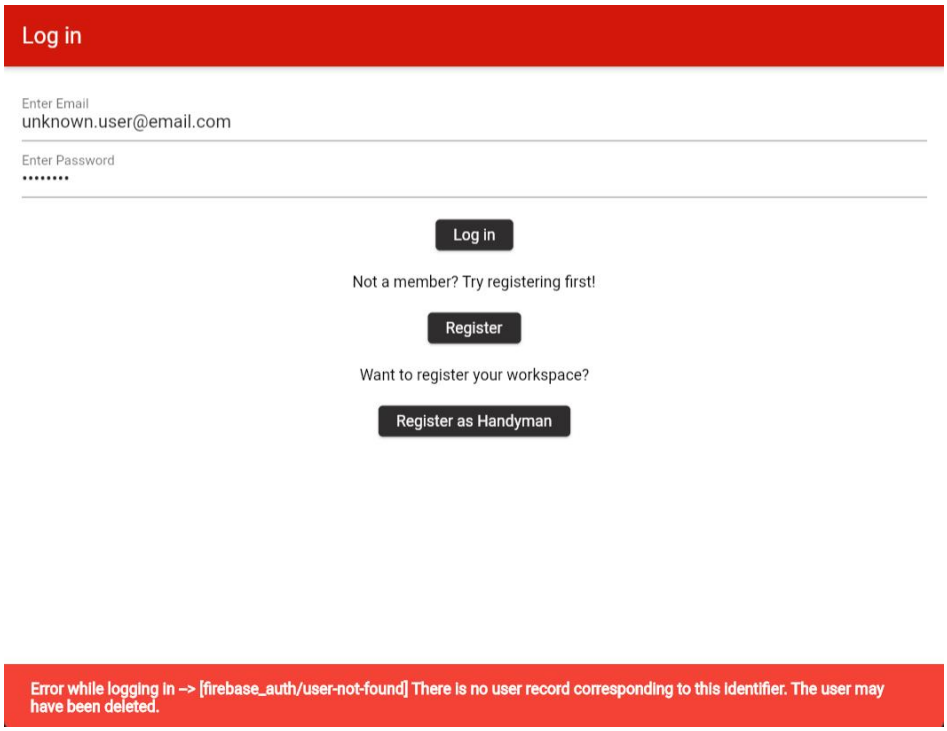
Slika 9 - Prijava korisnika

Izvede li se ovaj proces te metoda ne pronađe podudaranje unesenih podataka s podacima pohranjenima u bazi, prikazuje se odgovarajuća povratna poruka korištenjem metode *ShowSnackBar* iz klase *ScaffoldMessenger* koja služi za prikazivanje poruka i obavještavanje korisnika tokom rada aplikacije. Ova metoda stvara kratki prikaz obavještajne trake na dnu zaslona s navedenom porukom.

```
  } catch (err) {
    ScaffoldMessenger.of(context).showSnackBar(SnackBar(
      content: Text("Error while logging in --> $err",
          style: const TextStyle(
              color: Colors.white, fontWeight: FontWeight.bold)),
      backgroundColor: Colors.red,
    )); // SnackBar
  }
}
```

Slika 10 - Poruka o grešci pri prijavi

Kako bi se pri pokretanju aplikacije prikazao ovaj zaslon, u *main.dar*t datoteci potrebno je dodati import *'login.dart'*. Zaslon za prijavu izgleda ovako:



Slika 11 - Izgled zaslona za prijavu

2.3. registerUser.dart

2.3.1. Korištenje i prikaz mapa

U trenutnoj aplikaciji, pritiskom na gumb „*Register"* ne događa se ništa jer *widget* za prikaz zaslona za registraciju nije definiran. U zasebnoj datoteci naziva „*registerUser.dart"* potrebno je stvoriti strukturu tog zaslona. Kod registracije korisnika zahtijevat će se njegov unos korisničkog imena, elektroničke pošte, lozinke (uz potvrdu lozinke zbog sigurnosti), adrese i lokacije na mapi stoga je potrebno koristit *widget* s pohranom stanja.

Velik dio implementacije toga opisan je prethodno tekstu kod dodavanja *TextFormField*-ova, no novitet u ovom dijelu je dodavanje lokacije na mapi. Većina aplikacija današnjice koristi *Google Karte* za prikaz i postavljanje lokacija (Arellano, 2021), pošto su one široke distribuirane i lako upotrebljive. Međutim, pri izradi aplikacija pomoću *Flutter*-a, *Google Karte* zahtijevaju postavljanje za različite platforme. Primjerice, neophodno je zasebno konfigurirati mape za prikaz na *Android* i *IOS* sustavima, te je prije svega potrebno od *Google*-a zatražiti *API Ključ* kako bi programer mogao koristiti mape. Zbog ovih razloga, razvijena je alternativa *FlutterMap* koja nije široko distribuirana i nema detaljno razrađen prikaz karata kao *Google*, no za svrhu ove aplikacije je dovoljna.

Unutar *pubspec.yaml* konfiguracijske datoteke dodaje se zavisnost *flutter_map: ^0.13.0* te se pokreće naredba za ažuriranje zavisnosti u aplikaciji. Osim *flutter_map-a* u *registerUser.dart* datoteci poziva se *latlong2* ugrađeni paket koji služi za definiranje varijabli sa zemljopisnim dužinama i širinama. Pomoću ovog, klasom *LatLng* definiraju se vrijednosti koordinatne širine i visine na karti tako da se navode odgovarajuće brojčane vrijednosti u zagradi. Na slici je naznačen centar grada Rijeke u koordinatama. Osim lokacije na karti potrebno je definirati i varijable za spremanje ranije navedenih vrijednosti koje korisnik unosi pri registraciji.

```
class _RegisterFormState extends State<RegisterForm> {
  final formKey = GlobalKey<FormState>();
  String username = "";
  String email = "";
  String password = "";
  String confirmpass = "";
  String address = "";
  LatLng selectedLocation = LatLng(45.3271, 14.4422);
```

Slika 12 - Prikaz varijabli Registracijskog zaslona

Kako bi se karta mogla prikazati u *Column-u* ovog zaslona, za nju se definira *SizedBox* kojim se postavlja veličina i širina mape na zaslonu. Kao dijete ove klase postavlja se *Card* kao vizualni kontejner koji sadrži mapu. Njegovo dijete biti će klasa *FlutterMap* sa sljedećim parametrima:

- *options* parametar služi za postavljanje konfiguracije prikaza mape te se uz pomoć klase *MapOptions* mapu centrira, zumira te joj se dodaje akcija pri pritisku. Ovdje će se pozvati metoda *onMapTapped* koja će u varijablu za pohranu koordinata, pohraniti one koordinate na mapi gdje korisnik klikne.

- *layers* parametar naznačuje koji će se slojevi prikazivati na karti. Najniži sloj biti će sama mapa, a kako bi se korisniku povratno vizualno predočilo gdje je kliknuo na mapi, potrebno je dodati gornji sloj na kojem će se prikazati *marker*[6]. Klasom *MarkerLayerOptions* dodaje se *marker* i konfiguriraju njegove postavke.

[6] *Marker u ovom kontekstu se odnosi na engl. Pin - pribadača*

```
SizedBox(
    height: 400,
    width: 1400,
    child: Card(
        child: FlutterMap(
            options: MapOptions(
              center: selectedLocation,
              zoom: 14.0,
              onTap: onMapTapped,
            ), // MapOptions
            layers: [
          TileLayerOptions(
            urlTemplate:
                "https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",
            subdomains: ['a', 'b', 'c'],
          ), // TileLayerOptions
          MarkerLayerOptions(
            markers: [
              Marker(
                  width: 40.0,
                  height: 40.0,
                  point: selectedLocation,
                  builder: (context) => IconButton(
                      onPressed: () {},
                      icon: const Icon(Icons.location_on),
                      color: Colors.red)) // IconButton // Marker
            ],
          ) // MarkerLayerOptions
        ])), // FlutterMap // Card // SizedBox
```

Slika 13 - Definicija prikaza mape u widget-u

```
void onMapTapped(LatLng setLocation) {
  setState(() {
    selectedLocation = setLocation;
  });
}
```

Slika 14 - Metoda onMapTapped

2.3.2. Metoda *registerUser*

Uz definiranje polja za unos potrebnih informacija, neophodno je stvoriti i gumb za registraciju koji poziva metodu *registerUser* tipa *Future<void>*. Ova metoda koristi metodu *createUserWithEmailAndPassword* iz klase *FirebaseAuth* stvara novog korisnika aplikacije te mu sprema kriptirano lozinku i email u bazu podataka. Nakon toga u kolekciju *users* sprema korisničko ime, adresu, email i naznačenu lokaciju adrese korisnika. *Cloud Firebase*

omogućuje spremanje lokacija u obliku *GeoPoint* varijabli[7] stoga je potrebno pretvoriti koordinate prije spremanja.

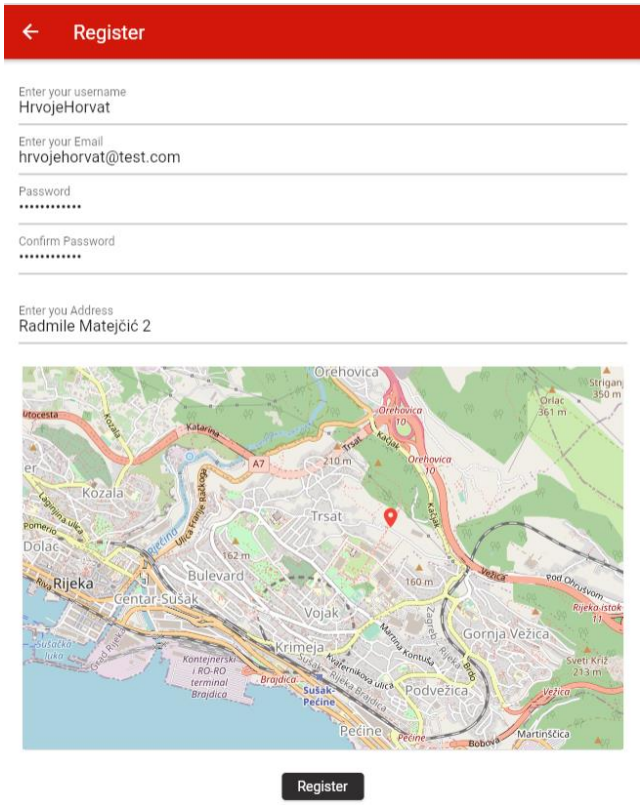
Dobra praksa kod izrade aplikacija je preusmjeravanje korisnika, odnosno automatska prijava nakon registracije, stoga je korisno implementirati ovu funkcionalnosti koristeći već prethodno spomenutu metodu *signInWithEmailAndPassword* te nakon toga koristeći *Navigator.push* preusmjeriti korisnika na naslovni zaslon aplikacije.

```
Future<void> registerUser(String username, String email, String password,
    String address, LatLng selectedLocation, BuildContext context) async {
  try {
    UserCredential userCredential = await FirebaseAuth.instance
        .createUserWithEmailAndPassword(email: email, password: password);

    String userId = userCredential.user!.uid;

    await FirebaseFirestore.instance.collection('users').doc(userId).set({
      'username': username,
      'email': email,
      'address': address,
      'address_location':
          GeoPoint(selectedLocation.latitude, selectedLocation.longitude)
    });
    await FirebaseAuth.instance
        .signInWithEmailAndPassword(email: email, password: password);
    // ignore: use_build_context_synchronously
    Navigator.pushReplacement(
        context,
        MaterialPageRoute(
            builder: (context) => MainPage(username: username))); // MaterialPageRoute
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(SnackBar(
      content: Text("Error while registrering --> $e",
          style: const TextStyle(
              color: Colors.white, fontWeight: FontWeight.bold)), // TextStyle // Text
      backgroundColor: Colors.red,
    )); // SnackBar
  }
}
```

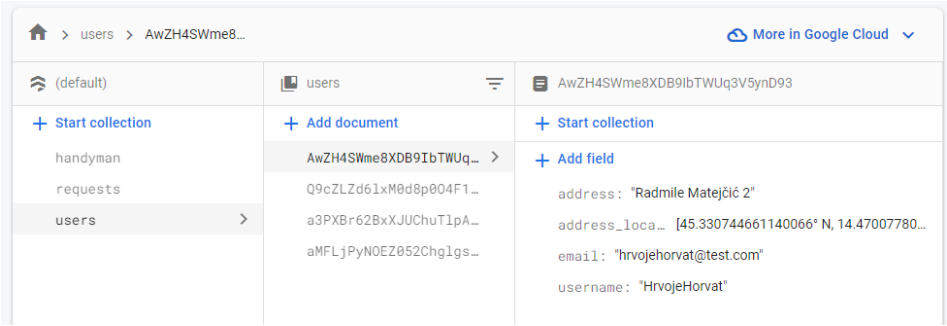
Slika 15 - metoda registerUser

Ukoliko se sada, unesu validni podaci za registraciju doći će do greške jer naslovni zaslon još nije stvoren, ali podaci će biti spremljeni u bazu podataka. Potrebno je pozvati *registerUser.dart* datoteku u *login.dart.*

[7] *GeoPoint* se koristi kod spremanja u bazu podataka zbog lakšeg zapisa dok je *LatLng* standard zapisa koordinata u *Dart*-u



Slika 16 - Izgled Zaslona za registraciju korisnika



Slika 17 - Slika zapisa u bazi podataka

2.4. registerHandyman.dart

Kod registracije kućnog majstora fokus nije na istim informacijama. Primjerice, adresa kućnog majstora nije potrebna u ovoj aplikaciji, dok s druge strane naziv tvrtke u kojoj radi mogao bi biti koristan. Temeljem toga, potreban je odvojen zaslon za registraciju kućnih majstora u aplikaciji. Od kućnih majstora tražit će se unos email adrese, lozinke (uz potvrdu lozinke zbog sigurnosti), korisničko ime, naziv tvrtke u kojoj je zaposlen te popis poslova za koje je specijaliziran. Pošto je moguće napisati veliki broj varijacija za nazive poslova,

napravljen je predefinirani popis te se od kućnog majstora očekuje da sam označi poslove. U tu svrhu koristiti će se *CheckBoxListTile* klasa koja omogućuje korisniku odabir jednog ili više od ponuđenih izbora. Ovdje se također koristi *widget* s promjenom stanja.

Za početak je, uz ostale tražene vrijednosti, potrebno definirati varijable za odabir poslova. U varijablu *availableServices* pohranit će se lista mogućih poslova za odabir. Za početak neka to bude par poslova, a kasnije razvojem aplikacije se dodaje više njih. Zatim, definira se prazna lista *selectedServices* u koju će se pohraniti oni poslovi koje je kućni majstor odabrao.

Također, kako ne bi došlo do prelamanja sadržaja zaslona sa prozorom u kojem se aplikacija otvara, dodaje se klasa *SingleChildScrollView* iznad *Column*-a koja omogućuje pomicanje sadržaja stranice povlačenjem po ekranu ili korištenjem kotačića na mišu.

```
class _HandymanRegisterFormState extends State<HandymanRegisterForm> {
  final formKey = GlobalKey<FormState>();
  String username = "";
  String email = "";
  String password = "";
  String confirmpass = "";
  String industry = "";
  List<String> availableServices = [
    "Plumber",
    "Electrician",
    "Ceramist",
    "Locksmith",
    "Painter",
    "Appliance Technician",
    "Carpenter"
  ];
  List<String> selectedServices = [];
```

Slika 18 - Varijable HandymanRegisterForm-e

Unutar glavne *Column* klase, definira se nova *Column* klasa koja će sadržavati sve odabire. Pomoću metode *map* za svaki element pohranjen u listi *availableServices* prikazuje se *CheckBoxListTile* takav da je njegov naslov pohranjena vrijednost. Ovime se stvara okvir za odabir te se klikom na njega u listu *selectedServices* dodaje označen odabir, a ponovnim klikom se taj odabir briše iz liste. Logika programskog koda izgleda ovako:

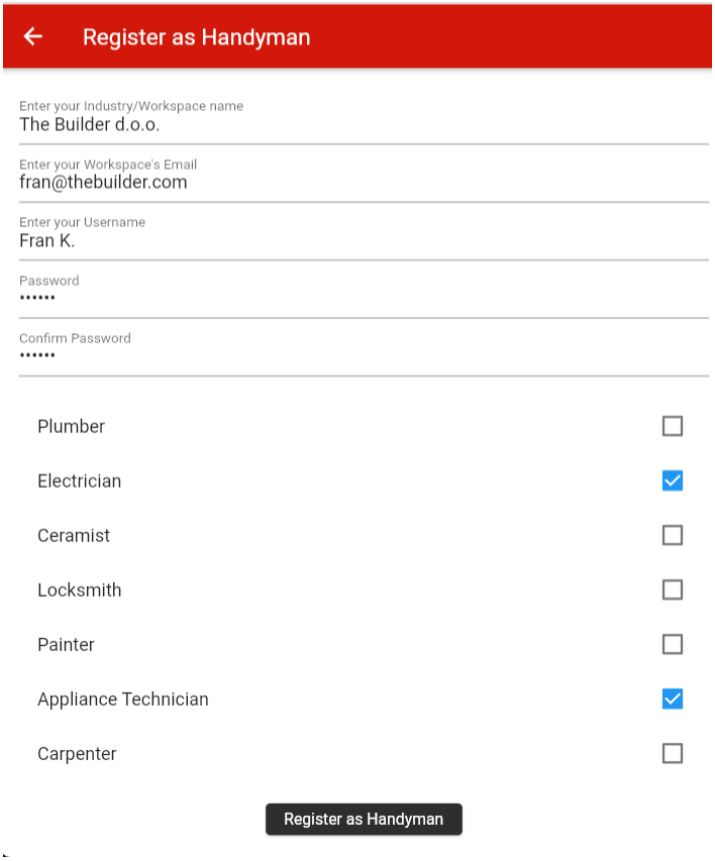
```
Column(
  children: availableServices.map((services) {
    return CheckboxListTile(
      title: Text(services),
      value: selectedServices.contains(services),
      onChanged: (bool? value) {
        setState(() {
          if (value!) {
            selectedServices.add(services);
          } else {
            selectedServices.remove(services);
          }
        });
      },
    ); // CheckboxListTile
  }).toList(),
), // Column
```

Slika 19 - Stvaranje okvira za odabir poslova

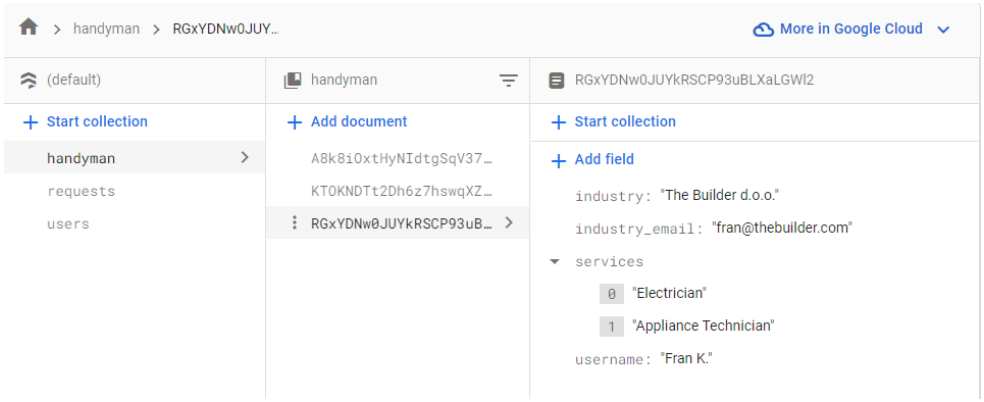
Temeljem prethodno objašnjenih procesa dodaju se okviri za unos teksta za ostale tražene informacije, te gumb za registraciju koji poziva metodu *registerHandyman.* Ova metoda slična je metodu *registerUser* uz razliku da se u bazu podataka zapisuje u kolekciju „*handyman*“ te se poziva drugi zaslon nakon registracije. Potrebno je pozvati *registerHandyman.dart* datoteku u *login.dart-u.*

```
  await FirebaseFirestore.instance
      .collection('handyman')
      .doc(handymanId)
      .set({
    'industry': industry,
    'industry_email': email,
    'username': username,
    'services': services,
  });
  await FirebaseAuth.instance
      .signInWithEmailAndPassword(email: email, password: password);
  // ignore: use_build_context_synchronously
  Navigator.pushReplacement(
      context,
      MaterialPageRoute(
          builder: (context) => HandymanMainPage(username: username)));
} catch (e) {
```

Slika 20 - Dio registerHandyman metode



Slika 21 - Zaslon za registraciju kućnog majstora



Slika 22 – Slika zapisa u bazi podataka

2.5. MainPage.dart

Dovršen je proces prijave i registracije te se sada korisnika preusmjerava na naslovni zaslon. Kod običnog korisnika[8] strukturira se zaslon koji sadrži navigacijsku traku gdje se korisniku omogućuje odjava iz sustava. Nadalje, glavni sadržaj stranice sastoji se od nekoliko gumbova zasebnih poslova i svaki od njih preusmjerava na stvaranje zahtjeva. Na dnu zaslona nalazi se traka gdje se korisniku nudi mogućnost odabira izmjene podataka profila ili pregled prethodno postavljenih zahtjeva. Pošto ovaj *widget* ne zahtjeva nikakvu pohranu vrijednosti iz unosa, ovdje se koristi *Stateless widget.*

Najprije se u *appBar* klasu definiranu unutar *Scaffold-a* dodaje parametar *actions* u kojem je moguće nizati gumbove ili ostale klase koje izvode neku akciju. Ovdje se poziva *Padding* klasa kako bi se pozicionirao gumb u gornji desni rub navigacijske trake, Zatim se pomoću *IconButton* klase definira gumb, postavlja odgovarajuća ikona te na pritisak gumba poziva metoda *_handleLogOut* koja poziva metodu *signOut* iz *FirebaseAuth* te se korisnika vraća na zaslon za prijavu.

[8] Običnim korisnikom smatra se korisnik koji nije kućni majstor već onaj koji šalje zahtjeve za potragom kućnog majstora.

```
appBar: AppBar(
  title: const Text("Handyman locator"),
  actions: [
    const Padding(
        padding: EdgeInsets.only(top: 16.0, right: 16.0),
        child: Text("Logged in as")), // Padding
    Padding(
        padding: const EdgeInsets.only(top: 16.0),
        child: Text(username,
            style: const TextStyle(fontWeight: FontWeight.bold))),
    IconButton(
        onPressed: () => _handleLogOut(context),
        icon: const Icon(Icons.logout)) // IconButton
  ],
), // AppBar
```

Slika 23 - Dodavanje odjave u AppBar

U metodi *_handleLogOut* koristi se *Navigator.pushReplacement* koja, za razliku od običnog *Navigator.push* ne sprema prethodni zaslon u stog. Korištenjem ove metode izbjegava se vraćanje u aplikaciju bez prijave jer aplikacija tada nema direktan pristup prethodnoj stranici.

```
void _handleLogOut(BuildContext context) async {
  await FirebaseAuth.instance.signOut();
  // ignore: use_build_context_synchronously
  Navigator.pushReplacement(
      context, MaterialPageRoute(builder: (context) => const LoginPage()));
}
```

Slika 24 - Odjava korisnika

Nakon što je implementirana mogućnost odjave, potrebno se fokusirati na sam izgled ovog zaslona. Na vrhu se dodaje običan tekst, pomoću klase *Text*, koji sugerira da korisnik treba odabrati jedan od ponuđenih poslova koji mu trenutno treba. Ispod teksta nalaze se gumbovi, svaki za pojedini posao koje aplikacija trenutno podržava. Ovi gumbovi vodit će na zaslon gdje će korisnik unositi više detalja o tome što mu treba. Kako bi se moglo razlikovati koji posao je korisnik odabrao, u pozvani *widget* potrebno je proslijediti i naziv posla.

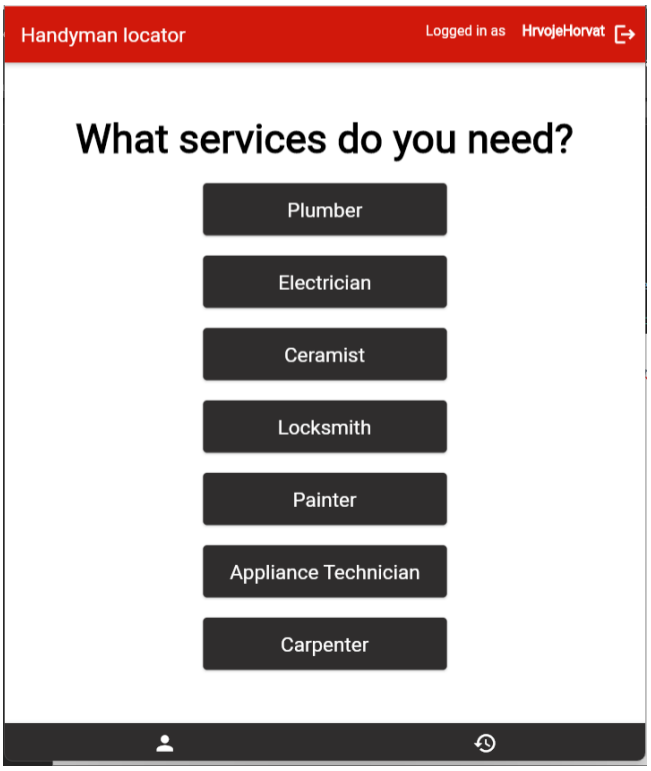
```
        ElevatedButton(
          onPressed: () =>
              navigateToRequest(context, "Electrician", username),
          style: ElevatedButton.styleFrom(
              minimumSize: const Size(250, 60),
              backgroundColor: const Color.fromARGB(255, 47, 45, 45)),
          child: const Text("Electrician", style: TextStyle(fontSize: 20)),
        ), // ElevatedButton
void navigateToRequest(BuildContext context, String service, String username
  {
    Navigator.pushReplacement(
        context,
        MaterialPageRoute(
            builder: (context) => RequestPage(
                  service: service,
                  username: username,
                ))); // RequestPage // MaterialPageRoute
  }
}
```

Slika 25 - Primjer gumba i metode navigateToRequest

Klasa *Scaffold* sadrži parametar *bottomNavigationBar* gdje je moguće definirati istoimenu metodu. Unutar ove klase definirat će se dva *IconButton-a* sa odgovarajućim ikonama od kojih jedan vodi u zaslon izmjene podataka korisnika, a drugi na povijest postavljenih zahtjeva. *MainPage.dart* potrebno je pozvati u svim datotekama koje pozivaju definiran *widget.* Izgled gotovog zaslona slijedi u nastavku.

```
bottomNavigationBar: BottomAppBar(
    color: const Color.fromARGB(255, 47, 45, 45),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceAround,
      children: [
        IconButton(
            onPressed: () {
              Navigator.pushReplacement(
                  context,
                  MaterialPageRoute(
                      builder: (context) =>
                          ProfilePage(username: username))); // MaterialPageRoute
            },
            icon: const Icon(Icons.person, color: Colors.white)), // IconButton
```

Slika 26 - Poziv klase BottomAppBar



Slika 27 - Prikaz naslovnog zaslona korisnika koji postavlja zahtjev

2.6. Request.dart

Odabirom jednih od ponuđenih poslova, korisnika treba zatražiti unos potrebnih detalja o zahtjevu kojeg postavlja. Korisnik treba unijeti naslov zahtjeva, detaljan opis problema ili posla kojeg treba obaviti te adresu na koju kućni majstor treba doći. Ovaj *widget* sadrži unose korisnika stoga on mora biti tipa *Stateful widget.* Pošto kod registracije korisnika već tražimo njegovu adresu, ovaj dio moguće je izvući iz baze podataka, no radi li se o nekoj drugoj lokaciji (primjerice apartmanu koji je u vlasništvu korisnika) onda korisnika treba zatražiti unos te lokacije. Ranije se već koristio okvir za odabir te je njegova modifikacija rješenje u ovom slučaju.

Najprije je potrebno učitati adresu prijavljenog korisnika iz baze podataka. Ovaj proces izvršava se odmah pri inicijalizaciji zaslona aplikacije, stoga je poziv metode implementiran u *initState* metodi. Iz kolekcije *„users“* metodom *where* dohvaćamo onaj dokument gdje je korisničko ime jednako korisničkom imenu prijavljenog korisnika pomoću parametra *isEqualTo* te spremamo taj dokument za daljnju uporabu. Ukoliko je taj dokument pronađen, u za to definirane varijable sprema se učitana adresu i koordinate lokacije na mapi.

Pošto je lokacija zapisana u obliku *GeoPoint* varijable, potrebno je obaviti pretvorbu u *LatLng* tip varijable.

```
void getAddress() async {
  DocumentSnapshot userSnapshot = await FirebaseFirestore.instance
      .collection('users')
      .where('username', isEqualTo: widget.author)
      .limit(1)
      .get()
      .then((querySnapshot) => querySnapshot.docs.first);

  if (userSnapshot.exists) {
    setState(() {
      address = userSnapshot.get('address');
      loadedLocation = userSnapshot.get('address_location');
      selectedLocation =
          LatLng(loadedLocation.latitude, loadedLocation.longitude);
    });
  }
}
```

Slika 28 - Dohvaćanje adrese iz baze podataka

Nakon toga, definira se klasa *CheckBoxListTitle* čiji je natpis učitana adresa. Vrijednost ovog okvira za odabir postavlja se kao nova varijabla tipa *bool NewAddress* čija je vrijednost *false*. Zbog toga je, pri otvaranju ovog zaslona, okvir za odabir već označen te sugerira da se koristi učitana adresa iz baze podataka. Ukoliko korisnik poništi ovaj odabir vrijednost varijable *NewAddress* postavlja se na *true* što omogućuje aplikaciji prikaz novog unosa za tekst nove adrese te mapu za označavanje nove lokacije.

```
    CheckboxListTile(
      title: const Text("Use new address?"),
      value: NewAddress,
      onChanged: (value) {
        setState(() {
          NewAddress = value!;
        });
      },
    ), // CheckboxListTile
```

Slika 29 - Odabir postojeće adrese

Naredbom za grananje *if* provjerava se vrijednost varijable *NewAddress* te se sukladno tome prikazuje novi tekstni okvir za unos adrese i mapa za označavanje lokacije. Zbog lakšeg snalaženja korisnika u aplikaciji, marker na mapi postavit će se na učitanu lokaciju iz baze podataka te će korisnik sam označiti novu lokaciju.

```
if (NewAddress)
  Column(
    children: [
      TextFormField(
          decoration:
              const InputDecoration(labelText: "Enter New Address"),
          validator: (value) {
            if (value!.isEmpty) {
              return "Please uncheck the Checkbox if you don't want to enter a New Address!";
            }
            return null;
          },
          onChanged: (value) {
            address = value;
          }), // TextFormField
```

Slika 30 - If grananje za provjeru unosa nove adrese

Na dnu zaslona, sada je potrebno samo dodati gumbove za potvrdu postavljanja zahtjeva te za povratak na prethodnu stranicu. Povratak se vrši metodom *Navigator.pop* koja učitava prethodnu stranicu iz stoga

Klikom na gumb potvrde, u bazu pod kolekcijom „*requests*“ spremaju se uneseni podaci uz dodatne podatke koji će biti potrebni dalje pri izradi aplikacije.

```
void storeRequest(String title, String description, LatLng location,
    String service, String author, String address) {
  FirebaseFirestore.instance.collection('requests').add({
    'author': author,
    'address': address,
    'title': title,
    'description': description,
    'service': service,
    'location': GeoPoint(location.latitude, location.longitude),
    'date_time': FieldValue.serverTimestamp(),
    'accepted': false,
    'price_per_hour': null,
    'fee': null,
    'description_of_fee': null,
    'handyman': null,
  }).then((value) {
    Navigator.pushReplacement(
        context,
        MaterialPageRoute(
            builder: (context) => MainPage(username: widget.author)));
```

Slika 31 - Spremanje zahtjeva u bazu podataka

U nastavku slijedi pojašnjenje parametara zapisa prilikom zapisivanja u bazu:

Author – korisničko ime korisnika koji je postavio zahtjev

Address – adresa na koju se očekuje dolazak kućnog majstora

Title – naslov / glavni opis problema ili posla koji se traži

Description – detaljan opis problema/posla koji treba obaviti

Service – vrsta odabranog posla za koji se postavlja zahtjev

Location – zemljopisne koordinate lokacije na koju se očekuje dolazak kućnog majstora zapisane u obliku *GeoPoint* varijable.

Date_time – vrijeme i datum postavljanja zahtjeva

Sljedeće varijable koristiti će se u nastavku razvoja aplikacije te se u njih trenutno pohranjuju *null* vrijednosti:

Accepted – *bool* varijabla koja pohranjuje vrijednost *true* ukoliko je netko od kućnog majstora ponudio cijenu obavljanja servisa koju je korisnik prihvatio

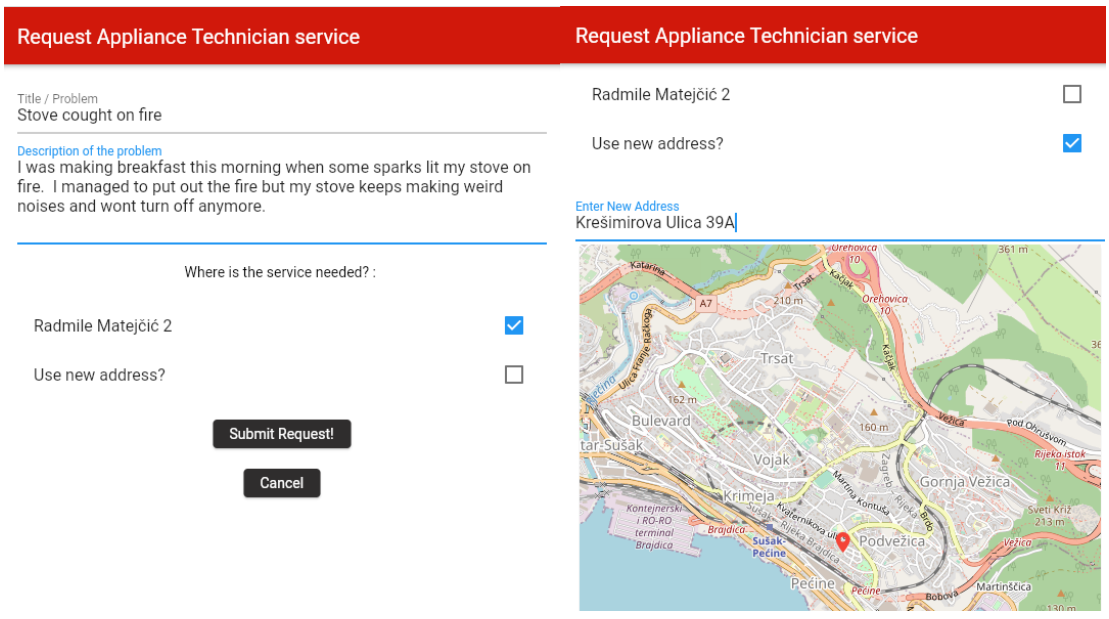
Price_per_hour – cijena posla po satu koju predlaže kućni majstor za obavljanje posla

Fee – dodatni troškovi koje kućni majstor navodi

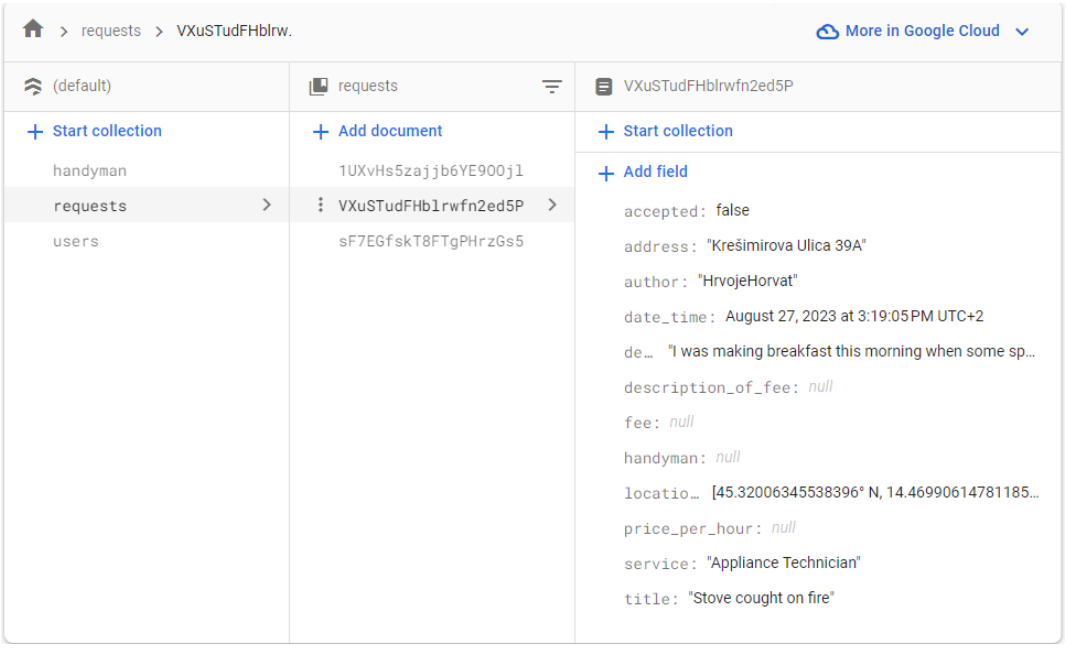
Description_of_fee – opis razloga navođenja dodatnih troškova

Handyman – korisničko ime kućnog majstora koji je predložio cijenu.

Request.dart potrebno je pozvati unutar *MainPage.dart* datoteke. Trenutni zaslon izgleda ovako:



Slika 32 - Prikaz zaslona aplikacije korištenjem stare i nove adrese



Slika 33 - Zapis u bazi podataka

2.7. HandymanMainPage.dart

Nakon što je zahtjev postavljen, on bi se trebao prikazat kućnim majstorima na naslovnom zaslonu nakon prijave. Međutim, aplikacije treba filtrirati samo one zahtjeve koji traže poslove za koje je kućni majstor specijaliziran. Na ovom zaslonu prikazuju se samo zahtjevi te se od kućnog majstora ne traži nikakav unos stoga je ovo *Stateless widget*.

Da bi aplikacija mogla filtrirati zahtjeve, najprije treba iz baze podataka izvući one poslove za koje je prijavljeni kućni majstor specijaliziran. Ovaj se postupak odvija metodom *getServicesForHandyman* koja je tipa *Future<List<String>>*. Pošto se radi o čitanju iz baze podataka, metoda mora biti tipa *Future* kako aplikacija ne bi nailazila na nepostojeće vrijednosti. Uz to metoda mora biti tipa *List<String>* jer će vratiti listu specijaliziranih poslova. Nakon što se dohvati dokument koji odgovara prijavljenom kućnom majstoru, u varijablu *services* tipa *List<String>* spremaju se poslovi.

```
Future<List<String>> getServiceForHandyman(
    String username, BuildContext context) async {
  List<String> services = [];
  try {
    var querySnapshot = await FirebaseFirestore.instance
        .collection('handyman')
        .where('username', isEqualTo: username)
        .get();

    if (querySnapshot.size > 0) {
      var handymanData = querySnapshot.docs.first;
      if (handymanData.exists) {
        services = List<String>.from(handymanData['services']);
      }
    }
  } catch (error) {
    ScaffoldMessenger.of(context).showSnackBar(SnackBar(
      content: Text("Error --> $error",
          style: const TextStyle(
              color: Colors.white, fontWeight: FontWeight.bold)),
      backgroundColor: Colors.red,
    )); // SnackBar
  }

  return services;
}
```

Slika 34 - Metoda getServiceForHandyman

U tijelu *widget*-a koristi se klasa *StreamBuilder* koja se sastoji od parametara *stream* i *builder*. Ova klasa uzima niz događaja definiranih u *stream*-u te prilikom svakog dodavanja novog elementa u niz pokreće funkciju definiranu u *builder*-u. Kao *stream* uzet će se zapisi u kolekciji *„request“* čiji su parametari *accepted* i *handyman* postavljen na *false* i vrijednost *null* te će se urediti silazno prema parametru *date_time* (novi zahtjevi prikazat će se na vrhu). Ukoliko u trenutku dohvaćanja podataka iz baze, nema takvih zapisa, ispisat će se odgovarajuća poruka, u suprotnom pozvat će se klasa *FutureBuilder*.

Klasa *FutureBuilder* funkcionira isto kao *StreamBuilder* osim što umjesto *stream* parametra sadrži *future* u koji se postavlja niz tipa *Future*.. Ovime će se prikazivati svi zahtjevi koji zadovoljavaju zadan uvjet.

```
body: StreamBuilder<QuerySnapshot>(
    stream: FirebaseFirestore.instance
        .collection('requests')
        .where('handyman', isNull: true)
        .where('accepted', isEqualTo: false)
        .orderBy('date_time', descending: true)
        .snapshots(),
    builder: (context, snapshot) {
      if (snapshot.hasError) {
        print(snapshot.error);
        return Center(child: Text('Error: ${snapshot.error}'));
      }
      if (!snapshot.hasData || snapshot.data!.docs.isEmpty) {
        return const Center(
            child: Text(
                'No requested services yet! Thank you for checking!'));
      }
      return FutureBuilder<List<String>>(
        future: getServiceForHandyman(username, context),
        builder: (context, servicesSnapshot) {
          if (servicesSnapshot.connectionState ==
              ConnectionState.waiting) {
            return const Center(child: CircularProgressIndicator());
          }
          if (servicesSnapshot.hasError) {
            return Center(
                child: Text("Error: ${servicesSnapshot.error}")); // Ce
          }
```

Slika 35 - StreamBuilder i FutureBuilder

Sada je još potrebno filtrirati one zahtjeve za čije je poslove kućni majstor specijaliziran. Ovo se obavlja metodama *where* i *any* slično kao što je prethodno objašnjeno u tekstu.

```
List<String> services = servicesSnapshot.data ?? [];
var filterRequest = snapshot.data!.docs
    .where((requestData) => services
        .any((service) => service == requestData["service"]))
    .toList();
```

Slika 36 - Filtriranje zahtjeva

Na kraju je podatke o zahtjevima potrebno prikazati u obliku kartica te se za to koristi klasa *ListView* s metodom *builder*. U parametar *itemCount* navodi se broj kartica, a u *itemBuilder* funkcija koja će definirati izgled kartice. *Widget InkWell* omogućuje interakciju sa karticom, odnosno klikom na karticu otvorit će se zaslon sa više detalja o zahtjevu.

Parametar *date_time* u bazi podataka pohranjuje proteklo vrijeme u nanosekundama od prvog siječnja 1970. godine, a zbog prirodnijeg izgleda, to bi trebalo pretvoriti u datum. Najprije u *TimeStamp* varijablu pohranimo spremljeno vrijeme te klasom *DateTime*

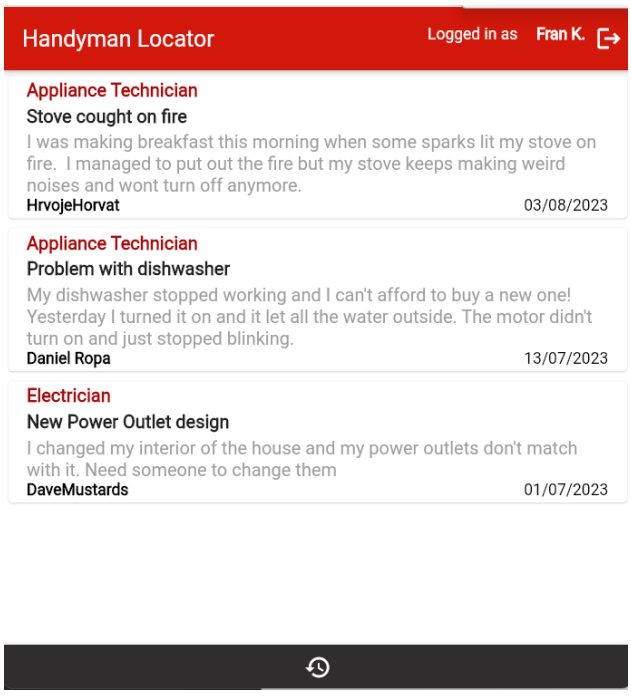
pretvorimo u datum. Na kraju, taj datum pretvorimo u *String* kako bi ga mogli prikazat u aplikaciji.

```
return ListView.builder(
  itemCount: filterRequest.length,
  itemBuilder: (context, index) {
    var requestData = filterRequest[index];
    Timestamp timestamp = requestData['date_time'];
    DateTime dateTime = timestamp.toDate();
    String formattedDate =
        DateFormat('dd/MM/yyyy').format(dateTime);
    return InkWell(
        onTap: () {
          Navigator.push(
              context,
              MaterialPageRoute(
                  builder: (context) => RequestDetailPage(
                      requestData: requestData,
                      username: username))); // RequestDetai
        },
```

Slika 37 - Definicija kartica zahtjeva u prikazu

U *ListTile* navode se podaci koji se žele prikazati u kartici. Ovdje se često koristi i *Row* koji omogućuje prikaz više elemenata u jednom redu. Potrebno je pozvati ovu datoteku u svim datotekama koje sadrže njene *widgete*. Izgled ovog zaslona prikazan je u nastavku.

Uz ovaj prikaz, na dnu je potrebno implementirati *bottomNavigationBar* u kojem se dodaje gumb koji vodi na prikaz prethodnih zahtjeva na koje je kućni majstor ponudio cijenu.



Slika 38 - Izgled naslovnog zaslona za kućne majstore

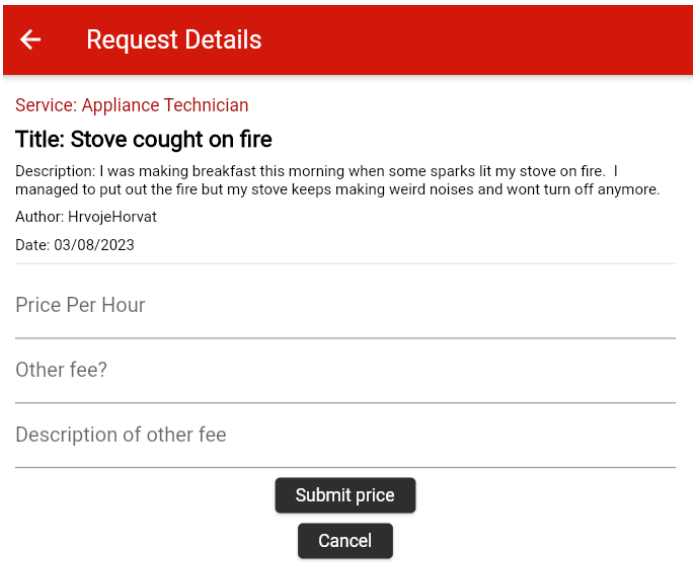
2.8. requestDetailPage.dart

Klikom na jednu od kartica omogućit će se kućnom majstoru detaljniji prikaz navedenog zahtjeva. Primjerice, biti će prikazani naslov, opis problema, traženi servis te autor i vrijeme zahtjeva. U ovom dijelu još nema prikaza adrese i lokacije zbog sigurnosti samih korisnika već će se oni prikazati tek kada kućni majstor prihvati posao. Kako bi prihvatio posao, on mora unijeti cijenu po satu, dodatne troškove te razlog dodatnih troškova zbog čega je ovo *Stateful widget*.

Cijena po satu, troškovi i opis troškova su standardni *TextFormField* koji je prethodno objašnjen, no ovoga se puta, pritiskom gumba ne stvara novi zapis već se izmjenjuje postojeći. Kako bi se to mogli izvesti, koristi se metoda *update* iz *FirebaseFirestore*-a. U ovom slučaju korišteni su *TextEditingControlleri* koji u spremaju vrijednost te se njoj pristupa pomoću *.text* varijable. U bazu se pohranjuju prethodno navedeni parametri u vrijednosti koje unosi kućni majstor te parametar *handyman* u korisničko ime kućnog majstora koji je prihvatio zahtjev. Nakon ovog, popunjena su sva polja u zapisu unutar baze podataka, osim parametra *accepted* koji će se izmijeniti tek kada korisnik prihvati ponuđenu cijenu. Datoteku *requestDetailPage.dart* potrebno je pozvati unutar *HandymanMainPage.dart* datoteke.

```
void updateRequest(
    BuildContext context,
    TextEditingController pph,
    TextEditingController fee,
    TextEditingController desc,
    String username) async {
  try {
    await FirebaseFirestore.instance
        .collection('requests')
        .doc(widget.requestData.id)
        .update({
      'price_per_hour': pph.text,
      'fee': fee.text,
      'description_of_fee': desc.text,
      'handyman': username
    });
```

Slika 39 - metoda updateRequest



Slika 40 - Izgled zaslona za unos i prikaz detalja zahtjeva

2.9. ConfirmationPage.dart

Nakon što kućni majstor unese cijenu i prihvati zahtjev, korisniku koji je postavio zahtjev treba se prikazati zaslon sa ponuđenim cijenama obavljanja servisa. Taj zaslon sadrži sve informacije o zahtjevu uz adresu i lokaciju, informacije o cijeni, troškovima i korisničkom imenu kućnog majstora. Isti taj zaslon potrebno je prikazati i kućnom majstoru kao potvrdu. Ovo je jednostavni zasloni koji samo prikazuju podatke i ne zahtijevaju nikakav unos stoga se koristi *Stateless widget.*

Obje strane ovog zahtjeva mogu u bilo kojem trenutku odbiti zahtjev te za to treba na dnu dodati gumb. Odbijanje zahtjeva vrši se tako da se zapis u bazi podataka izmjeni i postave parametri *handyman, price_per_hour, fee* i *description_of_fee* natrag na vrijednost *null* te se *accepted* postavlja na *false.*

```
Future<void> updateRequestStatus(
    String requestID, BuildContext context) async {
  try {
    await FirebaseFirestore.instance
        .collection("requests")
        .doc(requestID)
        .update({
      "accepted": false,
      "price_per_hour": null,
      "fee": null,
      "description_of_fee": null,
      "handyman": null
    });
```

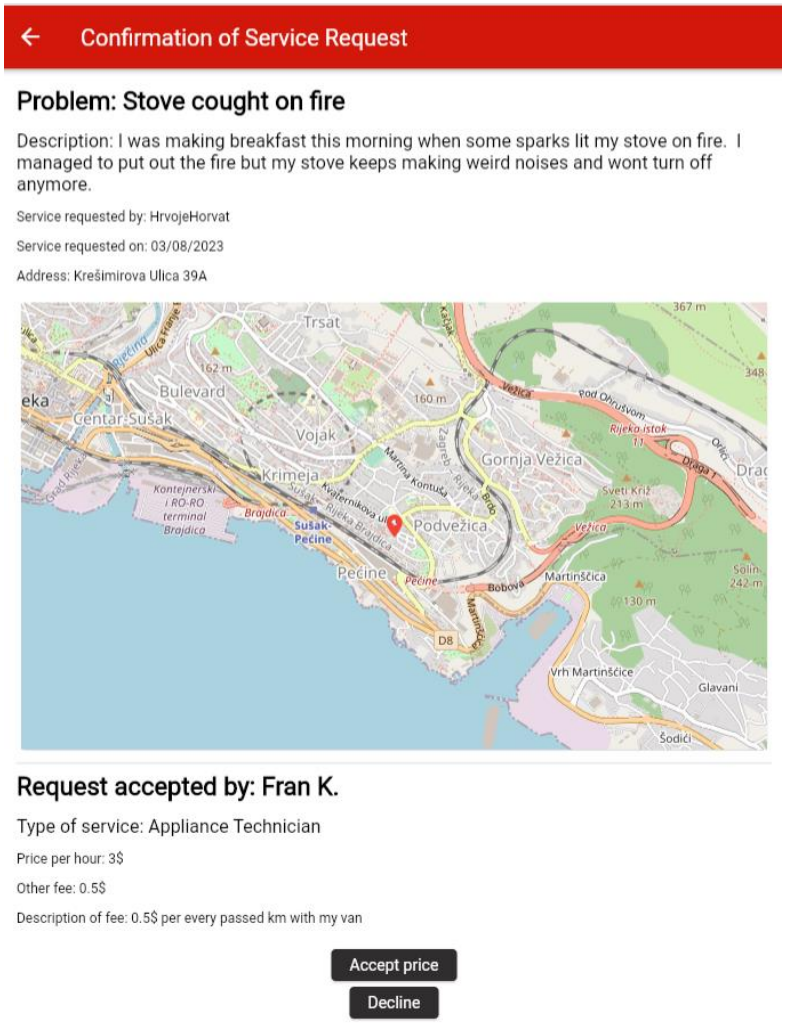
Slika 41 - Izmjena zapisa zahtjeva nakon odbijanja

Ukoliko je kućni majstor predložio cijenu, korisniku se u tom trenutku prikazuje i gumb za potvrdu cijene. To se vrši tako da se naredbom grananja provjerava ako je parametar *author* jednak korisničkom imenu korisnika (kako bi se izbjeglo prikazivanje gumba kućnom majstoru), parametar *handyman* različit od vrijednosti *null* te ako je *accepted false*. U tom slučaju, kućni majstor je predložio cijenu, no korisnik je još nije prihvatio stoga mu se za to nudi opcija. Klikom na gumb poziva se metoda *AcceptRequest* koja u bazi podataka za ovaj zahtjev postavlja parametar *accept* u vrijednost *true*.

```
Future<void> AcceptRequest(String requestID, BuildContext context) async {
  try {
    await FirebaseFirestore.instance
        .collection("requests")
        .doc(requestID)
        .update({
      "accepted": true,
    });
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(SnackBar(
      content: Text("Error --> $e",
          style: const TextStyle(
              color: Colors.white, fontWeight: FontWeight.bold)), // Text
      backgroundColor: Colors.red,
    )); // SnackBar
  }
}
```

Slika 42 - Metoda AcceptRequest

Ove datoteke sada je potrebno pozvati unutar datoteka koje pozivaju ove *widgete*. Izgled zaslona prikazani su u nastavku.



Slika 43 - Prikaz zaslona gdje korisnik može prihvatiti cijenu usluge kućnog majstora

2.10. HistoryRequestPage.dart

Kako bi korisnik imao sve prethodne zahtjeve na jednom mjestu, pritiskom na gumb s ikonom prošlih zapisa na alatnoj traci, oni se prikazuju u obliku listu baš kao što se to izvodi na naslovnom zaslonu kućnih majstora. Razlika je u tome što se ovdje ne filtriraju prema poslovima za koje su naznačeni već prema autoru zahtjeva.

```
body: StreamBuilder<QuerySnapshot>(
    stream: FirebaseFirestore.instance
        .collection('requests')
        .where('author', isEqualTo: widget.username)
        .orderBy('date_time', descending: true)
        .snapshots(),
    builder: ((context, snapshot) {
      if (snapshot.hasError) {
        return Center(child: Text('Error: ${snapshot.error}'));
      }
      if (!snapshot.hasData) {
        return const Center(
            child: Text("You haven't requested any service yet!"));
      }
```

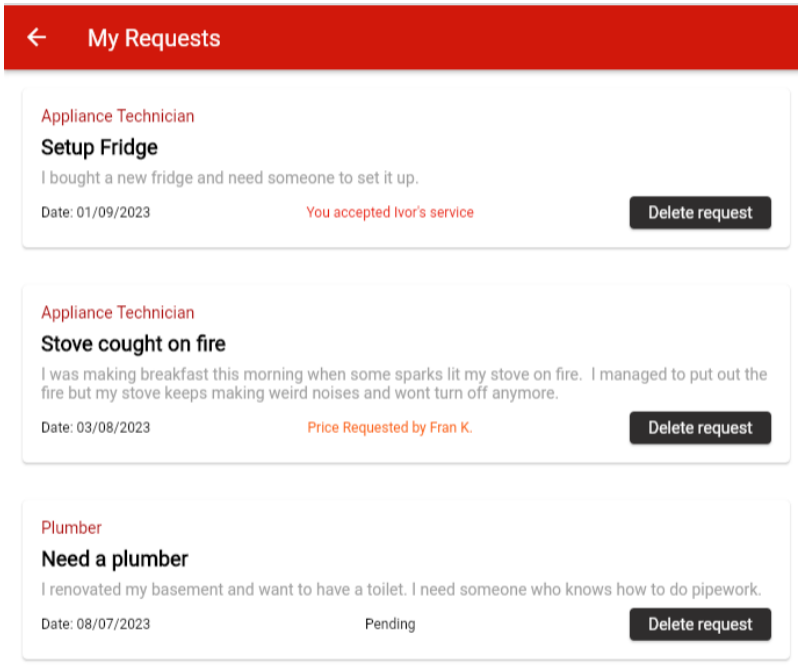
Slika 44 - Dohvaćanje zahtjeva autora iz baze podataka

Koristeći *InkWell* korisniku se klikom na karticu omogućuje i prikaz potvrde zahtjeva koji je ranije izrađen. Osim standardnih podataka, ukoliko je parametar *accepted* postavljen na *true*, na kartici će se pojaviti informacija kako je korisnik prihvatio ponuđenu cijenu kućnog majstora što naznačuje da je sada potrebno čekati majstora da dođe na adresu. Ukoliko je *accepted* postavljen na *false*, a *handyman* je različit od *null* vrijednosti, tada je kućni majstor ponudio cijenu obavljanja servisa te čeka korisnika da potvrdi ili odbije tu cijenu. Tada se na kartici prikazuje poruka koja savjetuje kako je cijena ponuđena te se ispisuje korisničko ime kućnog majstora. U suprotnom, ispisuje se poruka kako je zahtjev na čekanju. Uz ove podatke, na kartici se treba nalaziti i gumb za brisanje zahtjeva koji koristi *FirebaseFirestore* metodu *delete* koja brise zapis iz baze podataka.

```
if (request['accepted'])
  Text(
      "You accepted ${request["handyman"]}'s service",
      style: const TextStyle(
          fontSize: 12,
          color: Color.fromARGB(
              255, 232, 20, 5))) // Color.fromARGB // Text
else if (request['handyman'] == null)
  const Text("Pending",
      style: TextStyle(
        fontSize: 12,
      )) // TextStyle // Text
else
  Text(
      "Price Requested by ${request["handyman"]}",
      style: const TextStyle(
          fontSize: 12,
          color: Color.fromARGB(
              255, 251, 92, 18))), // Color.fromARGB // Te
ElevatedButton(
    onPressed: () {
      FirebaseFirestore.instance
          .collection('requests')
          .doc(request.id)
          .delete()
          .then((_) {});
    },
```

Slika 45 - Oznaka o prihvaćanju zahtjeva i brisanje

Ovu datoteku potrebno je pozvati unutar *MainPage.dart* datoteke. Konačan izgled zaslona sa svim vrstama zahtjeva izgleda ovako:



Slika 46 - Izgled zaslona povijesti zahtjeva korisnika

2.11. HandymanHistoryPage.dart

Na isti način kao korisniku, kućnom majstoru treba se prikazati popis prethodnih zahtjeva i to onih koji su prihvaćeni te onih kojima se još čeka odobrenje cijene. Do ovog prikaza dolazi se putem gumba na navigacijskoj traci na naslovnom ekranu. Princip je isti kao kod zaslona za prikaz prethodnih zahtjeva kod korisnika stoga se i tu koristi *Statless widget.*

Klikom na zasebnu karticu zahtjeva, kućnom majstoru treba se prikazati zaslon potvrde, odnosno potrebno je pozvati *ConfirmationPage.dart.* Iz baze podataka povlače se oni zahtjevi čiji je *handyman* parametar jednak korisničkom imenu prijavljenog kućnog majstora te se sortira silazno prema datumu.

```
stream: FirebaseFirestore.instance
    .collection('requests')
    .where('handyman', isEqualTo: username)
    .orderBy('date_time', descending: true)
    .snapshots(),
```

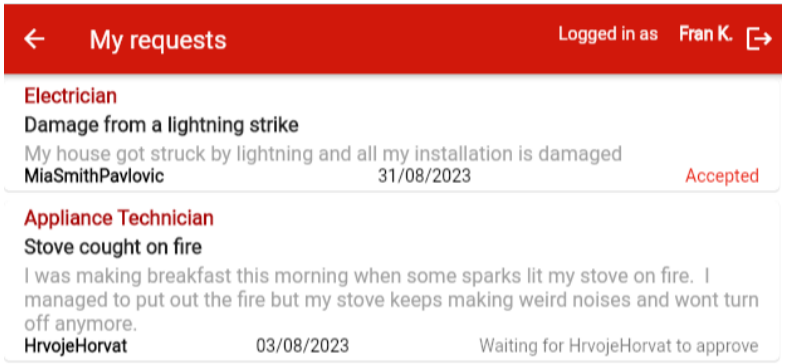
Slika 47 - Povlačenje iz baze zahtjeve po kućnom majstoru

Uz standardne podatke koji se prikazuju na kartici, potrebno je prikazati i tekst o stanju zahtjeva. Ako je parametar *accepted* postavljen na *true* prikazuje se potvrdna poruka o prihvaćanju ponuđene cijene, a u suprotnom ispisuje se korisničko ime autora zahtjeva te poruka o čekanju prihvata cijena.

```
if (requestData['accepted'])
  const Text("Accepted",
      style: TextStyle(
          color: Color.fromARGB(255, 232, 20, 5))) // TextStyle // Text
else
  Text(
    "Waiting for ${requestData['author']} to approve",
  ) // Text
```

Slika 48- Prikaz odgovarajućih poruka zahtjeva

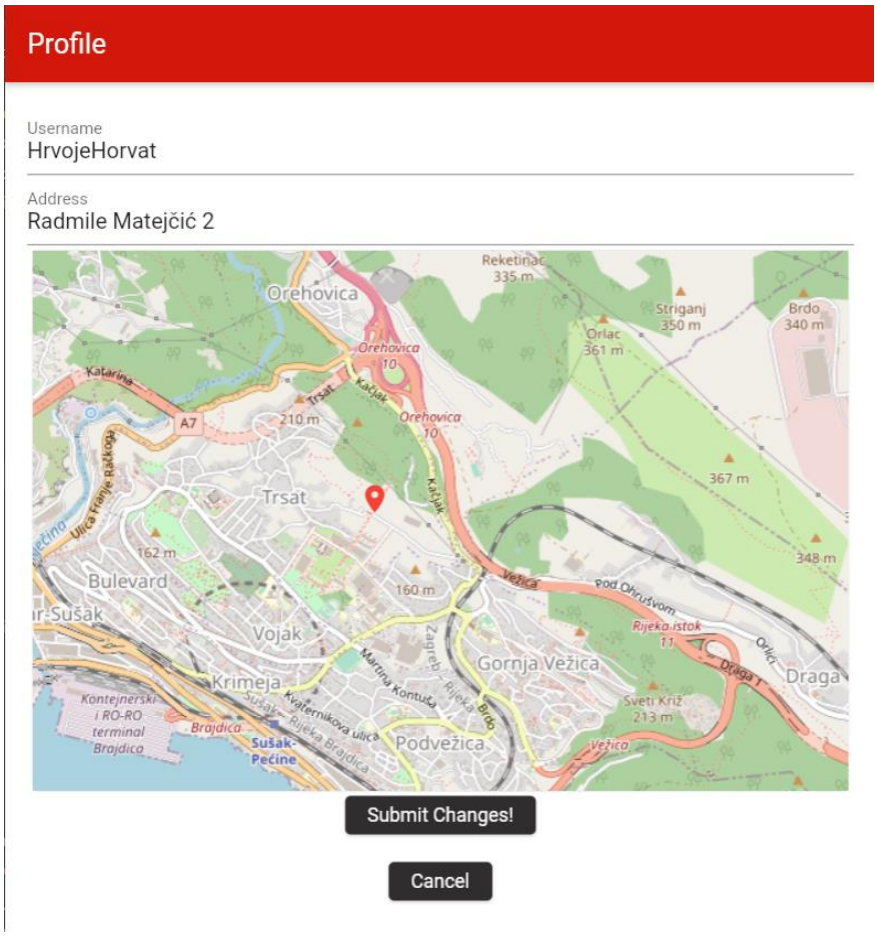
Ovu datoteku potrebno je pozvati svugdje gdje se spominju njegovi *widgeti*. Izgled zaslona prikazan je u nastavku:



Slika 49 - Izgled zaslona povijesti zahtjeva kućnog majstora

2.12. ProfilePage.dart

Za kraj, potrebno je još stvoriti zaslon na kojem korisnik može izmijeniti svoje podatke. Do tog zaslona korisnik dolazi pritiskom na gumb sa ikonom osobe na alatnoj traci. Na njemu se nalaze prethodno ispunjeni podaci o korisničkom imenu, adresi i karta sa lokacijom adrese te se omogućuje njihovo izmjenjivanje. Kako bi okviri za unos teksta bili popunjeni vrijednostima dosegnutim iz baze, oni se pohranjuju u *TextEditingController* te se u parametar *controller* klase *TextEditingForm* postavlja ta varijabla. Nakon izmjene podataka poziva se metoda *update* te se nove vrijednosti zamjenjuju starima u bazi. *Widget* koristi forme stoga je on tipa *Stateful widget.* Ostatak programskog koda ovog zaslona sličan je ostatku koda opisanog u pojedinim datotekama. *ProfilePage.dart* potrebno je pozvati u datoteci *MainPage.dart*. Slijedi izgled zaslona.



Slika 50 - Zaslon za izmjenu podataka korisnika

3. Zaključak

Izradom aplikacije prikazuju se glavne prednosti i nedostatci koje ovaj razvojni okvir nudi. Već je ranije spomenuta njegova popularnost, no valja napomenuti kako je ovo relativno novi razvojni okvir zbog čega trenutno manjka broj ugrađenih paketa koje on nudi. Velik dio funkcionalnosti potrebno je samostalno implementirati što može dovesti do komplikacija prilikom izrade opširnijih aplikacija.

Ipak, razvijanje aplikacije u *Flutter*- u je brzo i efikasno što je omogućeno izmjenom koda aplikacije dok je ona pokrenuta kako bi se u stvarnom vremenu predočile obavljene promjene. Ova praksa korisna je kod testiranja i ispravljanja grešaka.

Jednostavnosti ovog razvojnog okvira pridonosi i ugrađen dizajn korisničkog sučelja koji je lako izmjenjiv i prilagodiv, a prati standarde aplikacija današnjice. Što se tiče *backened*[9]-a, kompatibilan je sa raznim sustavima poput *SQLite, Firebase, Cloud Firestore, Hive* i ostalih[10],

Flutter se preporučuje pri izradi aplikacija standardnih funkcionalnosti jer nudi brzinu u razvoju te jednostavnost pri izradi, no kod izrade kompleksnijih aplikacija može dovesti do male brzine izvodivosti aplikacije kao i kompleksnosti samog koda.

[9] *Beckend* – pozadinski dio aplikacije koji se bavi pohranom i uređivanjem podataka. Dio aplikacije koji nema izravan kontakt s korisnikom (Fitzgibbons, 2019).
[10] Kompletan popis kompatibilnih sustava za *backend* moguće je pronaći na priloženom izvoru

4. Popis priloga

Programski kod aplikacije:

https://github.com/DinoLadavac/HandymanFinderApp

5. Popis slika

6. Popis literature

Arellano, K. (23. Travanj 2021). *The Top 10 Mapping & Maps APIs (for Developers in 2018)*. Preuzeto Kolovoz 2023 iz rapidapi.com: https://rapidapi.com/blog/top-map-apis/

Fitzgibbons, L. (Svibanj 2019). *Frontend and backend*. Preuzeto Kolovoz 2023 iz techtarget.com: https://www.techtarget.com/whatis/definition/fronend#:~:text=The%20back%20end%20refers%20to,end%20of%20a%20computer%20system.

Flutter. (n.d.). *Flutter documentation*. Preuzeto Srpanj 2023 iz docs.flutter.dev: https://docs.flutter.dev/

Flutter. (n.d.). Flutter Youtube channel. Preuzeto Srpanj 2023 iz https://www.youtube.com/@flutterdev

Google. (n.d.). *Službena stranica Firebase*. Preuzeto Srpanj 2023 iz firebase.google.com: https://firebase.google.com/

Khomutova, S. (3. Kolovoz 2023). *Top 24 famous apps built with Flutter Framework*. Preuzeto 11. Kolovoz 2023 iz apexive.com: https://apexive.com/post/top-apps-built-with-flutter-framework

Lab, I. M. (21. Srpanj 2023). *The Best Database for Flutter: A Comprehensive Guide*. Preuzeto 11. Kolovoz 2023 iz fluttertalk.com: https://fluttertalk.com/best-database-for-flutter/

Mike Katz, K. D. (2022). *Flutter Apprentice.*

Montaño, D. (20. Srpanj 2023). *Why use flutter: pros and cons of flutter app development*. Preuzeto 10. Kolovoz 2023 iz waverleysoftware.com: https://waverleysoftware.com/blog/why-use-flutter-pros-and-cons/#2

Rall, R. (18. Svibanj 2022). *Flutter Database Options: How to Choose the Right Database for Your Flutter App?* Preuzeto Srpanj 2023 iz medium.com: https://medium.com/dhiwise/how-to-choose-right-flutter-database-a1f35237a7f9

Reichert, A. (Travanj 2020). *API key*. Preuzeto Kolovoz 2023 iz techtarget.com: https://www.techtarget.com/whatis/definition/API-key

Službena dokumentacija Cloud Firestore - Flutter. (n.d.). Preuzeto Srpanj 2023. iz firebase.flutter.dev: https://firebase.flutter.dev/docs/firestore/usage/

Vailshery, L. S. (1. Lipanj 2023). *Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2022*. Preuzeto 10. Kolovoz 2023 iz statista.com: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/