Izrada aplikacije Restoran korištenjem Flutter razvojnog okvira

Smojver, Filip

Sveučilište u Rijeci, Fakultet informatike i digitalnih tehnologija

Sveučilišni prijediplomski studij Informatika

Filip Smojver

Izrada aplikacije Restoran korištenjem Flutter razvojnog okvira

Završni rad

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

Rijeka, srpanj 2023.

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

Rijeka, 26.5.2023.

Zadatak za završni rad

Pristupnik: Filip Smojver

Naziv završnog rada: Izrada aplikacije Restoran korištenjem Flutter razvojnog okvira

Naziv završnog rada na engleskom jeziku: Development of the Flutter application Restaurant

Sadržaj zadatka: Cilj ovog završnog rada je istražiti Flutter okvir za razvoj mobilnih/desktop aplikacija. U sklopu rada će biti osmišljena, razvijena i testirana aplikacija Restoran u kojoj su implementirane četiri razine uloga korisnika. Dok administrator ima sve ovlasti u aplikaciji, uloga klijenta pruža uvid u jelovnik i trenutni broj slobodnih stolova, a uloge konobar i kuhar podržavaju rad s narudžbama te međusobnu komunikaciju.

Mentor

Izv. prof. dr. sc. Marija Brkić Bakarić

Voditelj za završne radove

Doc. dr. sc. Miran Pobar

Zadatak preuzet: 26.5.2023.

(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

Cilj ovog završnog rada je istražiti Flutter okvir za razvoj mobilnih/desktop aplikacija. U sklopu rada će biti osmišljena, razvijena i testirana aplikacija Restoran u kojoj su implementirane četiri razine uloga korisnika. Dok administrator ima sve ovlasti u aplikaciji, uloga klijenta pruža uvid u jelovnik i trenutni broj slobodnih stolova, a uloge konobar i kuhar podržavaju rad s narudžbama te međusobnu komunikaciju. U prvom dijelu rada opisane su korištene tehnologije i funkcionalnosti aplikacije, a u središnjem dijelu detaljnije je prikazan razvoj aplikacije u Flutteru. Autentifikacija korisnika i baza podataka implementirana je pomoću usluga koje pruža Firebase.

Ključne riječi

Flutter, widget, Firebase, Firestore, aplikacija

Sadržaj

1. Uvod

Zbog velike potražnje višeplatformskih (*cross-platform*) mobilnih aplikacija na tržištu i potrebe pojednostavljenja njihove izrade Googleov tim razvio je Flutter, razvojni okvir otvorenog koda (*open-source framework*). Kao glavni programski jezik Flutter koristi Dart, koji je također Google razvio, te pruža programerima veliki izbor alata i widgeta za izgradnju korisničkih sučelja i time pojednostavljuje razvoj aplikacija.

Svrha ovog završnog rada je opisati aplikaciju Restoran i njezin razvoj pomoću razvojnog okvira Flutter. Za izradu aplikacije korišten je uređivač koda Visual Studio Code, a za autentifikaciju i uporabu baze podataka koriste se usluge koje pruža Firebase. Funkcionalni cilj aplikacije je pružiti pregled glavnih stavki Restorana koje bi mogle zanimati posjetitelje i koristiti radnom osoblju.

Unutar aplikacije korisnik koji nije prijavljen ima mogućnost saznati broj trenutno slobodnih stolova, telefonski broj restorana i omogućen mu je pregled jelovnika hrane i pića. Radno osoblje za svoje potrebe mora biti prijavljeno i ima dodatne opcije ovisno o njihovoj ulozi koja može biti konobar, kuhar i administrator.

Konobar ima funkciju rezerviranje i dodavanja narudžbi za određeni stol, može dodati hranu ili piće koje se nalazi u jelovniku i u slučaju zauzetog stola dobiva povratnu informaciju zauzeća. Narudžba hrane koju unosi konobar automatski se prosljeđuje na prikaz kuharu. Uloga kuhara omogućava označavanje napravljene narudžbe te automatsko dodavanje označene narudžbe na listu gotovih jela kojoj mogu pristupiti korisnici u ulogama konobara i kuhara. Dodatno, konobar ima funkciju oslobođenja stola i brisanja narudžbi stola. Uloga administratora omogućava dodavanje, uređivanje i brisanje korisničkih računa zaposlenika iz baze podataka, te ima opcije prikaza i korištenja svih funkcionalnosti implementiranih za ostale uloge.

2. Korištene tehnologije

2.1. Flutter

Razvojni okvir Flutter izrazito je koristan za razvoj modernih aplikacija koje su uporabljive na različitim platformama, kao što su iOS, Android, stolno računalo i različiti moderni web preglednici. Ta mogućnost je olakšana jer Flutter koristi jedan izvorni kod za sve platforme te time omogućuje efikasnije kreiranje aplikacije. Kažemo da je Flutter *natively compiled* jer koristi vlastiti kompajler koji prevodi izvorni kod u izvršni i time omogućuje puno bolje performanse i brže izvršavanje aplikacije.

Flutter se koristi *Widget-based* arhitekturom koja omogućuje izgradnju aplikacije pomoću *widgeta* čija je svrha izrada korisničkog sučelja, tj. *widgeti* se koriste kao temeljni gradivni elementi vizualnog i funkcionalnog dijela Fluttera. Postoje dvije vrste *widgeta* u Flutteru - *stateless widget* i *stateful widget*. Temeljna razlika između te dvije vrste je promjenjivost, *stateful widget* ima mogućnost promjene stanja ako korisnik odradi neku radnju unutar njega, dok stateless widget uvijek ostaje isti i nema promjena stanja. Za organizaciju *widgeta* koristi se stablo *widgeta*, gdje svaki element stabla ima svog roditelja osim korijenskog *widgeta* u kojemu započinje izvođenje aplikacije. Na slici 1 prikazan je kod s označenim nasljednicima *widgeta* ElevatedButton koji predstavlja povišeni gumb. a pored koda na slici 2 se može vidjeti prikaz *widgeta* ElevatedButton u sučelju aplikacije.

```
ElevatedButton(
  style: ElevatedButton.styleFrom(···
  onPressed: () {
    Navigator.push(···
      MaterialPageRoute( // MaterialPageRoute ···
  },
  child: const Column(
    children: [
      Icon(
        Icons.food_bank,
        size: 100,
      ), // Icon
      SizedBox(
        height: 5,
      ), // SizedBox
      Text(
        'Hrana',
        style: TextStyle( // TextStyle ···
      ), // Text
    ],
  )), // Column // ElevatedButton
```

Slika 1: Kod povišenog gumba s označenim nasljednicima

Slika 2: Prikaz povišenog gumba

Widgeti također doprinose boljim performansama aplikacije jer imaju mogućnost nepromjenjivosti korištenjem ključne riječi *const* i mogu se ponovno koristiti za optimiziranje *rendering* procesa.

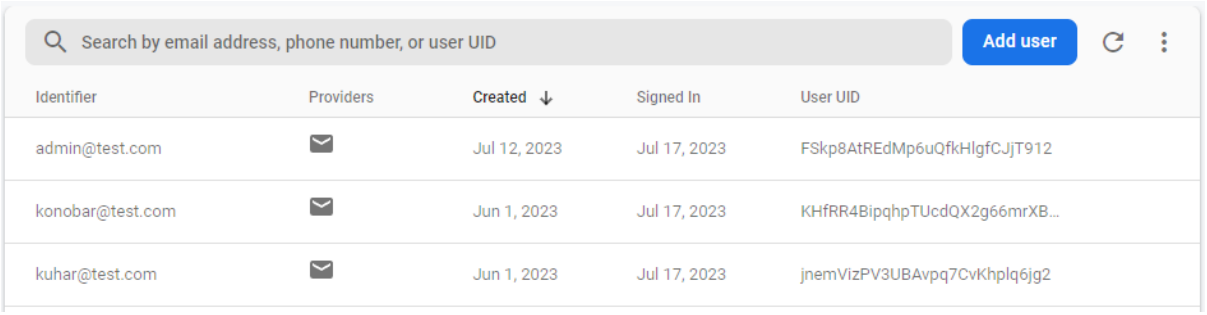
Važne značajke u Flutter-u su funkcije *hot reload* i *hot restart* koje omogućavaju brzi prikaz rezultat promijenjenog koda tokom izrađivanja aplikacija. *Hot reload* koristi se za brzu ponovnu izgradnju korisničkog sučelja s ažuriranim kodom i njegovom izvedbom stanje (*state*) aplikacije ostaje isto. *Hot restart* traje duže od *hot reloada* jer se ponovno pokreće aplikacija s ažuriranim kodom, a stanje stanje aplikacije se resetira.

2.2. Dart

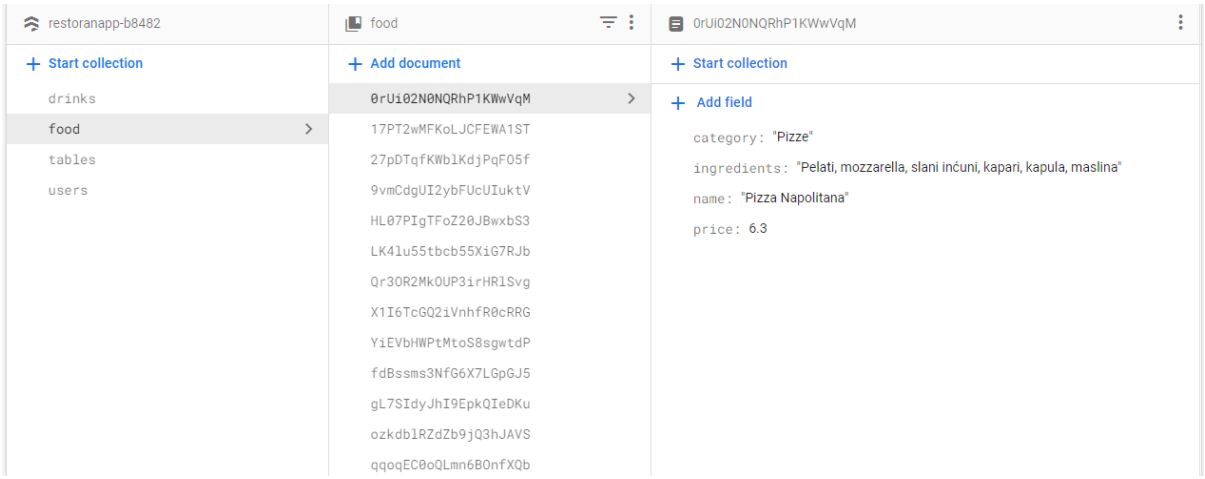
Objektno orijentirani jezik Dart koristi se za razvoj aplikacija u Flutteru. Inspiriran je programskim jezicima Javom, JavaScriptom, Pythonom i C#, te je kreiran uzimajući u obzir najkorisnije osobine tih programskih jezika. Dart je optimiziran za izradu korisničkih sučelja i ima mogućnost prevođenja u JavaScript za web preglednike. Koristi dvije vrste kompajlera ovisno o fazi u kojoj je aplikacija. Tijekom razvoja koristi se *just-in-time* kompajler koji omogućava lakše ponovno učitavanje aplikacije i otklanjanje pogrešaka. Kada je aplikacije gotova i spremna za korištenje upotrebljava se *ahead-of-time* kompajler za brže performanse tijekom korištenja i bolje održavanje memorije. *Garbage collector* ili sakupljač smeća važna je stavka kod upravljanja memorijom, njegova svrha je oslobađanje memorije koja se ne koristi.

2.3. Firebase

Firebase je razvojna platforma za izradu aplikacija u vlasništvu Googlea. Omogućava razvojnim programerima pristup veoma korisnim alatima i funkcijama za izgradnju moderne aplikacije koja konkurira na tržištu. Osim za razvoj aplikacije, Firebase daje usluge praćenja performansi i stabilnosti, te analitičke podatke korištenja aplikacije. Usluge korištene u razvoju aplikacije ovog završnog rada su baza podataka Cloud Firestore i usluga autentifikacije korisnika Authentication. Prikaz pregleda registriranih računa na Firebase usluzi Authentication je vidljiv na slici 3. Na slici 4 je prikaz baze podataka i dokumenata kolekcije hrane (*food*).



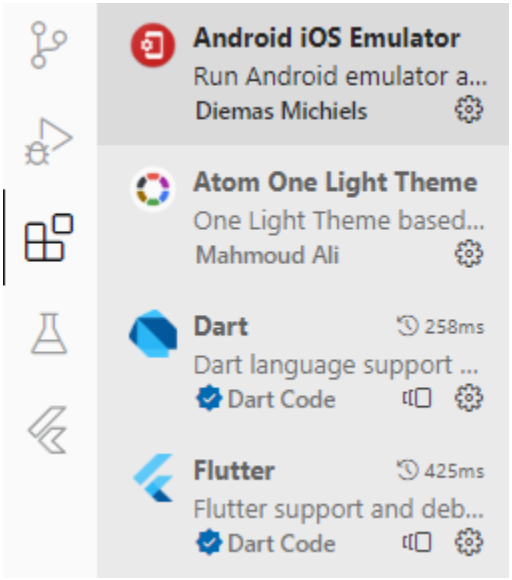
Slika 3: Prikaz registriranih korisnika na Firebaseu



Slika 4: Prikaz Cloud Firestore baze podataka na Firebaseu

2.4. Visual Studio Code

Code editor ili uređivač koda Visual Studio Code koji je razvio Microsofta je jako popularan izbor za razvojne programere. Omogućava jednostavno uređivanje koda uz mogućnosti automatskog ispravljanja koda, otkrivanja i otklanjanja pogrešaka unutar editora bez potrebe za pokretanjem programa, rada s Git-om i dodavanja proširenja za lakši razvoj aplikacije. Za izradu aplikacije završnog rada korištena su glavna proširenja: podrška i debugger za Dart jezik i Flutter, te podrška za pokretanje Android iOS Emulatora.



Slika 5: Prikaz korištenih proširenja u Visual Studio Codeu

2.5. Android Studio

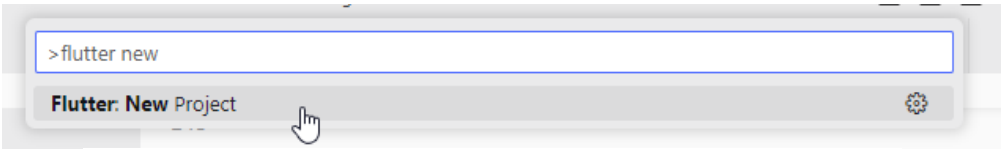
Za potrebe korištenje Android Emulator tokom razvoj aplikacije primijenjeno je integrirano razvojno okruženje Android Studio koje je kreirao Google za izgradnju Android aplikacija. Iako Android Studio nije bio korišten za uređivanje koda, valja spomenuti da je on odličan izbor za tu svrhu i omogućuje jednostavan razvoj Android mobilnih aplikacija jer je specifično dizajniran za njihovu izradu. Android Studio ima mogućnost emuliranja raznih Android uređaja i time je omogućeno lakše testiranje prilagodljivosti aplikacije. Na slici 6 prikazan je Android Emulator korišten tokom razvoja.

Slika 6: Android emulator "Pixel 2 API 29"

3. Razvoj aplikacije

3.1. Priprema i stvaranje projekta

Prije same izrade aplikacije potrebno je instalirati Flutter SDK (*Software development kit*) i pripremiti kodni uređivač Visual Studio Code za razvoj Flutter aplikacije. To zahtjeva instalaciju potrebnih ekstenzija za Flutter i Dart unutar programskog okruženja Visual Studio Code. Nakon instalacije potrebnih stavki, Flutter projekt stvara se naredbom prikazom na slici 7.



Slika 7: Stvaranje novog Flutter projekta za Visual Studio Code

Za prikaz emulatora Android uređaja tijekom razvoj aplikacije potrebno je instalirati Android Studio i ekstenziju Android iOS Emulator.

Postavljanje Firebase-a nalaže instalaciju Firebase CLI-a (*Command-line interaface*), prijave unutar terminala računom korištenim na Firebaseu, aktiviranja i podešavajanja FlutterFire CLI-a za spajanje aplikacija na Firebase. Nakon svih obavljenih koraka potrebno je inicirati Firebase u novostvoreni projekt i postaviti sve potrebne pakete za uspješno izvođenje Firebasea u aplikaciji. Završni korak je dodavanje linije koda u glavnoj funkciji programa (Slika 8) i imamo aplikaciju povezanu na Firebase koja je spremna za razvoj.

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform);

  runApp(const MyApp());
}
```

Slika 8: Prikaz linije koda za iniciranje Firebasea

Pokretanje aplikacije izvodimo naredbom „flutter run“ unutar terminala glavne datoteke projekta.

```
Using hardware rendering with device Android SDK built for x86. If you
notice graphics artifacts, consider enabling software rendering with
"--enable-software-rendering".
Launching lib\main.dart on Android SDK built for x86 in debug mode...
Building with Flutter multidex support enabled.
Running Gradle task 'assembleDebug'...
```

Slika 9: Prikaz izvođenja naredbe "flutter run"

3.2. Prvi widget i provjera prijave

Razvoj aplikacije počinje pozivanjem prvog *widgeta* zvanog „MyApp“. Njegova funkcija je stvoriti bazni dio aplikacije u kojem se poziva *widget* „MaterialApp“ unutar kojeg imamo definirane tematske boje korištene za osnovne elemente aplikacije kao što su gumbi, pozadina, izbornik (*drawer*) i slično. Na kraju koda poziva se *widget „LoggedIn“* čija je svrha provjera prijave korisnika. *Widget* „MyApp“ prikazan je na slici 10.

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        debugShowCheckedModeBanner: false,
        title: 'Restoran App',
        theme: ThemeData(
            progressIndicatorTheme: ProgressIndicatorThemeData(
                color: Colors.lightGreen,
                circularTrackColor: Colors.lightGreen.shade200,
                refreshBackgroundColor: Colors.lightGreen.shade200), // Progres
            drawerTheme: const DrawerThemeData(
                backgroundColor: Color.fromARGB(255, 248, 252, 218)), // Drawer
            scaffoldBackgroundColor: const Color.fromARGB(255, 248, 252, 218),
            appBarTheme:
                const AppBarTheme(color: Color.fromARGB(255, 40, 110, 19)),
            floatingActionButtonTheme: const FloatingActionButtonThemeData(
                backgroundColor: Color.fromARGB(255, 48, 133, 23)), // Floating
            elevatedButtonTheme: ElevatedButtonThemeData(
                style: ElevatedButton.styleFrom(
                    backgroundColor: const Color.fromARGB(255, 48, 133, 23))),
            buttonTheme: const ButtonThemeData(alignedDropdown: true)), // ThemeD
        home: const LoggedIn()); // MaterialApp
  }
}
```

Slika 10: Prikaz koda widgeta "MyApp"

Provjera prijave odvija se u *widgetu* „LoggedIn“ koji obnaša tu zadaću pomoću *widgeta* „StreamBuilder“ kojim se dolazi do informacije promjene autentifikacijskog stanja trenutačnog korisnika.

„StreamBuilder“ *widget* je veoma bitan element razvoja ove aplikacije zbog potrebe dobivanja informacija koje se mijenjaju ovisno o interakciji korisnika s aplikacijom i promjenama unutar baze podataka. Funkcioniranje „StreamBuilder“ *widgeta* ovisi o *streamu* koji konstanto „sluša“ sve moguće promjene sa zadanim podatkovnim izvorom, kad se dogodi promjena *builder* funkcija se ponovo poziva i taj dio *widgeta* koristi nove informacije od *streama.*

U ovom slučaju „StreamBuilder“ radi na dobivanju informacije promjena autentifikacijskog stanja korisnika preko Firebasea. Kada dođe do promjene builder se iznova poziva i podatak izvora se sprema u varijablu *snapshot*. U slučaju da se čeka *stream* i dohvaćanje konekcije na zaslonu nam se prikazuje *widget* „Loading“. Kada je podataka dohvaćen započinje provjera tog podataka. Ako podatak ima neki sadržaj odnosno nije *null*, to označuje stanje prijavljenog korisnika i poziva se *widget* „Role“ koji provjerava ulogu korisnika, a u slučaju gdje nema sadržaja za podataka koji smo dohvatili, korisnik nije prijavljen i poziva se *widget* „Home“. Na slici je prikaz koda *widgeta* „LoggedIn“.

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: StreamBuilder<User?>(
      stream: FirebaseAuth.instance.authStateChanges(),
      builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
          return const Loading();
        }
        if (snapshot.hasData) {
          return const Role();
        } else {
          return const Home(0);
        }
      },
    )); // StreamBuilder // Scaffold
  }
}
```

Slika 11: Prikaz koda widgeta "LoggedIn"

3.3. Provjera uloge korisnika

Uloga korisnika provjerava se u *widgetu* „Role“ gdje koristimo *widget* „FutureBuilder“ za dohvaćanje uloge trenutno prijavljenog korisnika preko baze podataka Cloud Firestore.

„FutureBuilder“ ima sličnu funkciju kao i „StreamBuilder“, ali za razliku od „StreamBuildera“ njegova svrha je dohvatiti podatak koristeći *future* samo jedanput i spremit ga u *snapshot*, promjenom tog podatka „FutureBuilder“ ne poziva funkciju *builder* opet već je potrebno iskoristiti funkciju *setState* za ponovno postavljanje stanja *widgeta*. Za korištenje *setState* funkcije bitna je stavka da *widget* u kojem se funkcija koristi bude *Stateful* inače funkcija nije uporabljiva.

Provjera uloge korisnika ne zahtjeva konstantni dotok promjena jer se poziva kod promjene autentifikacijskog stanja korisnika.

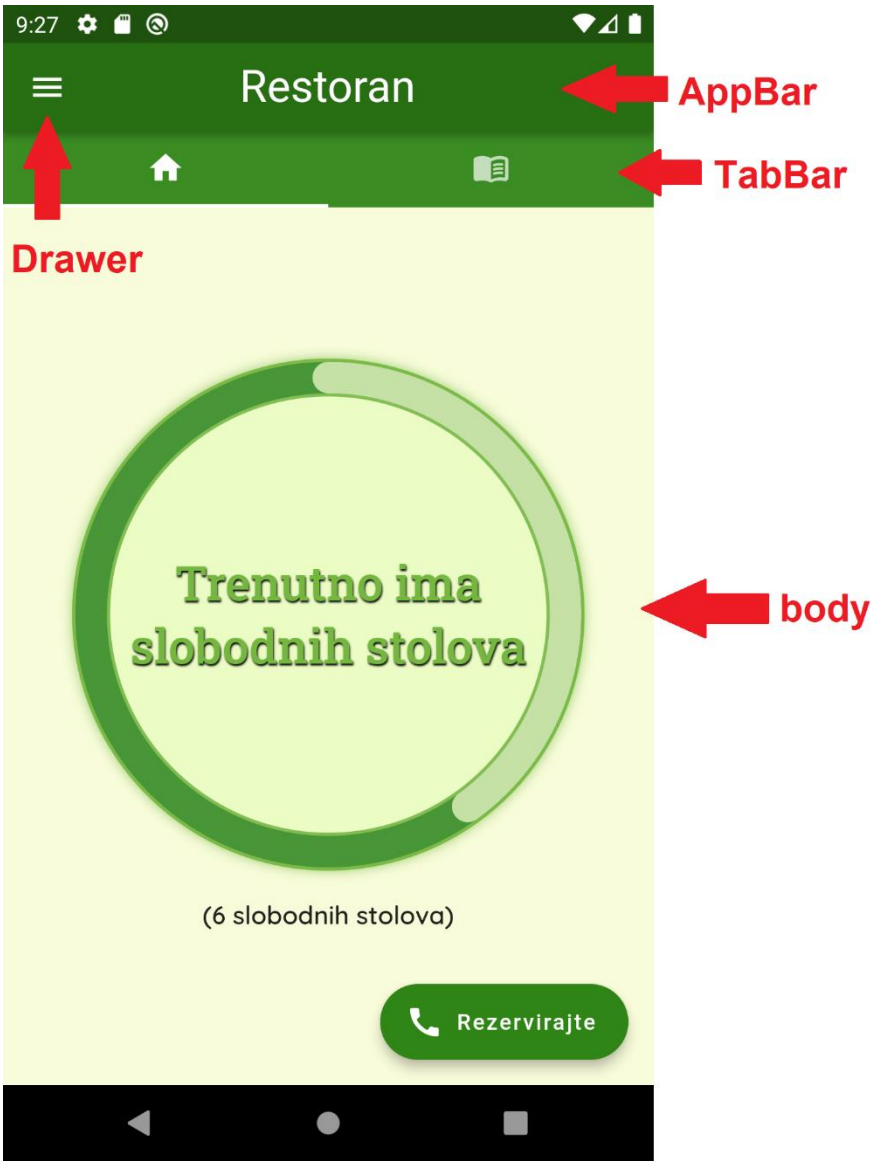
Nakon dohvaćanja uloge korisnika, a ovisno o vrijednosti uloge, pozivaju se specifični *widgeti* o kojima ovisi daljnji prikaz aplikacije. U slučaju gdje uloga nema vrijednost unutar baze podataka poziva se *widget* „Home“ koji vraća prikaz za neprijavljene korisnike. Razlog tomu je što bi svaki prijavljeni korisnik trebao imati dodijeljenu ulogu. Ako nema uloga, a zaposlenik je restorana aplikacije, potrebno mu je dodijeliti ulogu preko Cloud Firebasea.

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

  @override
  Widget build(BuildContext context) {
    final FirebaseAuth auth = FirebaseAuth.instance;
    final db = FirebaseFirestore.instance;
    final User? user = auth.currentUser;
    final userEmail = user!.email;

    return FutureBuilder(
        future: db
            .collection('users')
            .doc(userEmail)
            .get()
            .then((DocumentSnapshot documentSnapshot) {
          if (documentSnapshot.exists) {
            var role = documentSnapshot.get('role') as String;
            return role;
          } else {
            return 'user';
          }
        }),
        builder: (context, snapshot) {
          if (snapshot.connectionState == ConnectionState.waiting) {
            return const Loading();
          }
          if (snapshot.data == 'Admin') {
            return const AdminPage(0);
          } else if (snapshot.data == 'Konobar') {
            return KonobarPage(0);
          } else if (snapshot.data == 'Kuhar') {
            return KuharPage(0);
          } else {
            return const Home(0);
          }
        }); // FutureBuilder
  }
}
```

Slika 12: Prikaz koda widgeta "Role"

3.4. Neprijavljeni korisnik

Neprijavljenom korisniku ili gostu restorana u aplikaciji se prikazuje jednostavno korisničko sučelje u kojemu ima pristup prikazu informacija o slobodnim stolovima u restoranu, te jelovniku hrane i pića. Također ima mogućnost prijave ako ima potrebne podatke za pristup računu koji je za njega kreirao administrator. Korisničko sučelje sastavljeno je od *widgeta* „Scaffold“ koji je obuhvaćen *widgetom* za kontroliranje kartica sučelja zvanim „DefaultTabController“ koji ima svojstvo *length* postavljeno na dva zbog zadanog broja izbora između dvije kartice. „Scaffold“ je *widget* koji nam omogućuje stvaranje okvira koji je temelj za prikaz različitih elemenata modernih aplikacija. Glavna svojstva unutar „Scaffolda“ su „appBar“, „drawer“ i „body“ koji predstavljaju gornju traku, izbornik i tijelo aplikacije. Kartice i njihov sadržaj nalaze se u tijelu „Scaffolda“. Prikaz sučelja (Slika 13) i koda aplikacije (Slika 14) prikazani su u nastavku.



Slika 13: Prikaz sučelja neprijavljenog korisnika

```dart
class Home extends StatefulWidget {
  final int selectedPage;
  const Home(this.selectedPage, {super.key});

  @override
  State<Home> createState() => _HomeState();
}

class _HomeState extends State<Home> {
  @override
  Widget build(BuildContext context) {
    return DefaultTabController(
        initialIndex: widget.selectedPage,
        length: 2,
        child: Scaffold(
          appBar: AppBar(
            title: const Text('Restoran'),
            centerTitle: true,
            titleTextStyle: const TextStyle(
              fontSize: 28,
              color: Color.fromARGB(255, 255, 255, 255),
            ), // TextStyle
          ), // AppBar
          drawer: const UserDrawer(),
          body: Column(children: [
            const Material(
                color: Color.fromARGB(255, 59, 141, 35),
                child: TabBar(indicatorColor: Colors.white, tabs: [
                  Tab( // Tab ···
                  Tab( // Tab ···
                ])), // TabBar // Material
            Expanded(
              child: TabBarView(children: [
                Scaffold( // Scaffold ···
                const MenuDisplay()
              ]), // TabBarView
            ) // Expanded
          ]), // Column
        )); // Scaffold // DefaultTabController
```

Slika 14: Prikaz koda sučelja neprijavljenog korisnika

Važno je spomenut da je „Home“ *stateful widget* jer kad dođe do potrebe za navigaciju na njega iskoristiva je mogućnost selekcije prikazane kartice. Na prikazu primjera linije koda iz „LoggedIn“ *widgeta* (Slika 15) broj nula označava prvu karticu koja se prikazuje nakon pozivanja *widgeta.* Broj nula se stavlja u „selectedPage“ varijabli i koristi se za kao inicijalni indeks unutar *widget*a „DefaultTabController“.

```dart
return const Home(0);
```

Slika 15: Prikaz primjera pozivanje widgeta "Home"

Prva kartica korisničkog sučelja sastavljena je od „Scaffolda“ koji ima dva svojstva: tijelo i lebdeći gumb (*floatingActionButton*). Lebdeći gumb ima funkciju otvaranja Android aplikacije za telefonski poziv gdje je unesen telefonski broj restorana. Tijelo poziva novi *widget* „CircleDisplay“ koji nam daje obojeni okrugli prikaz i broj slobodnih stolova u sredini tijela kartice. On unutar sebe koristi paket „Percent Indicator“. Prikaz „CircleDisplay“ *widgeta* se mijenja ovisno o broju stolova u bazi podataka, te se za dohvaćanje tog podatka koristi „StreamBuilder“. Prikaz dohvaćanja sa „StreamBuilder“ *widgetom* i mogućih prikaza sučelja dani su na slikama 16-20.

```
child: StreamBuilder<QuerySnapshot>(
    stream:
        FirebaseFirestore.instance.collection('tables').snapshots(),
    builder: (context, snapshots) {
      if (snapshots.connectionState == ConnectionState.waiting) {
        return const Column(
          children: [
            SizedBox(height: 140),
            CircularProgressIndicator(),
          ],
        ); // Column
      } else {
        final tables = snapshots.data!.docs.length;
        final double tPercent = ((tables / 10));
```

Slika 16: Prikaz dohvaćanja broja stolova

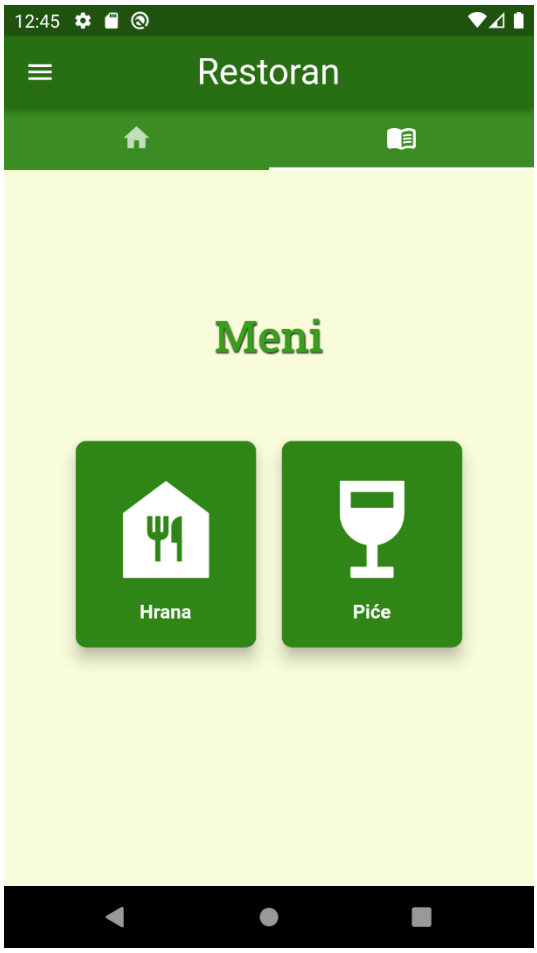
Slika 17: Prikaz sučelja 1

Slika 18: Prikaz sučelja 2

Slika 19: Prikaz sučelja 3

Slika 20: Prikaz sučelja 4

Druga kartica sučelja koristi *widget* „MenuDisplay“ koji prikazuje izbor između dva menija (*Slika 21*). Oba menija imaju isti način prikaza stavki, ali dohvaćaju različite dokumente unutar baze podataka. Samo funkcioniranje prikaza stavki za oba menija je isto, pa će u svrhu demonstracije biti objašnjeno prikazivanje jelovnika hrane.



Slika 21: Prikaz izbora menija

Jelovnik hrane svoj prikaz započinje *widgetom* „Foods“. Unutar njega kreirana je prazna lista „categoryFoods“ s modelom „CategoryFoods“ koji je sastavljen od kategorije i liste stavki koji pripadaju kategoriji. Stavke imaju svoj model koji sadrži naslov, cijenu i sastojke za hranu te kategoriju (Slika 22). Za dohvaćanje podataka i spremanje tih podataka o različitim kategorijama u listu modela, dodana je vanjska funkcija „getStream“ koja se izvodi kada je *widget* prvi put izgrađen (Slika 23). U „getStream“ funkciji izvodi se i kreiranje prijelomne točke koja označava kada će se kartica promijeniti ovisno o tome gdje je korisnik pozicioniran tokom prikaza svih stavki (Slika 24). Zadnja stavka je iniciranje *scroll controllera* koji „sluša“ sve moguće promjenu tokom listanja i ažurira selekciju kartica.

```
class Food {
  final String title;
  final num price;
  final String ingredients;

  Food({required this.title, required this.price, required this.ingredients});
}

class CategoryFoods {
  final String category;
  final List<Food> items;

  CategoryFoods({required this.category, required this.items});
}
```

Slika 22: Prikaz modela za hranu

```
class _FoodsState extends State<Foods> {
  List<CategoryFoods> categoryFoods = [];

  final CollectionReference foodsRef =
      FirebaseFirestore.instance.collection('food');

  getStream() async {
    for (var i = 0; i < categories.length; i++) {
      List dataList = [];
      var data =
          await foodsRef.where('category', isEqualTo: categories[i]).get();
      dataList = data.docs;
      List<Food> items = [];
      for (var i = 0; i < dataList.length; i++) {
        items.add(Food(
            title: dataList[i]['name'],
            price: dataList[i]['price'],
            ingredients: dataList[i]['ingredients']));
      }
      categoryFoods.add(CategoryFoods(category: categories[i], items: items));
    }
```

Slika 23: Prikaz funkcije "getStream"

```
List<int> breakPoints = [];
void createBreakPoints() {
  int firstBreakPoint = (112 * categoryFoods[0].items.length);
  breakPoints.add(firstBreakPoint);
  for (var i = 1; i < categoryFoods.length; i++) {
    int breakPoint =
        breakPoints.last + (64 + 58 * categoryFoods[i].items.length);
    breakPoints.add(breakPoint);
  }
}
```

Slika 24: Prikaz stvaranja prijelomnih točka

Widget Foods koristi „slivers“ svojstvo *widgeta* „CustomScrollView“ za prikazivanje svih kategorija i njihovih stavki (Slika 25). Za prikaz stavki unutar liste kategorija korišten je vlastiti *widget* „FoodCard“, a za prikaz kategorija *widget* „RestoranCategories“.

```
return Scaffold(
    body: CustomScrollView(
  controller: scrollController,
  slivers: [
    const SliverAppBar(title: Text('Hrana'), pinned: true),
    SliverPersistentHeader(
      delegate: RestoranCategories(
          onChanged: scrollToCategory,
          selectedIndex: selectedCategoryIndex), // RestoranCategories
      pinned: true,
    ), // SliverPersistentHeader
    SliverPadding(
      padding: const EdgeInsets.symmetric(horizontal: 16),
      sliver: SliverList(
        delegate: SliverChildBuilderDelegate(
          (context, categoryIndex) {
            List<Food> items = categoryFoods[categoryIndex].items;
            return FoodsCategory(
                title: categoryFoods[categoryIndex].category,
                items: List.generate(
                    items.length,
                    (index) => Padding(
                          padding: const EdgeInsets.only(bottom: 14.0),
                          child: FoodCard(
                            title: items[index].title,
                            price: items[index].price,
                            ingredients: items[index].ingredients,
                          ), // FoodCard
                        ))); // Padding // List.generate // FoodsCategory
          },
          childCount: categoryFoods.length,
```

Slika 25: Prikaz koda widgeta Foods

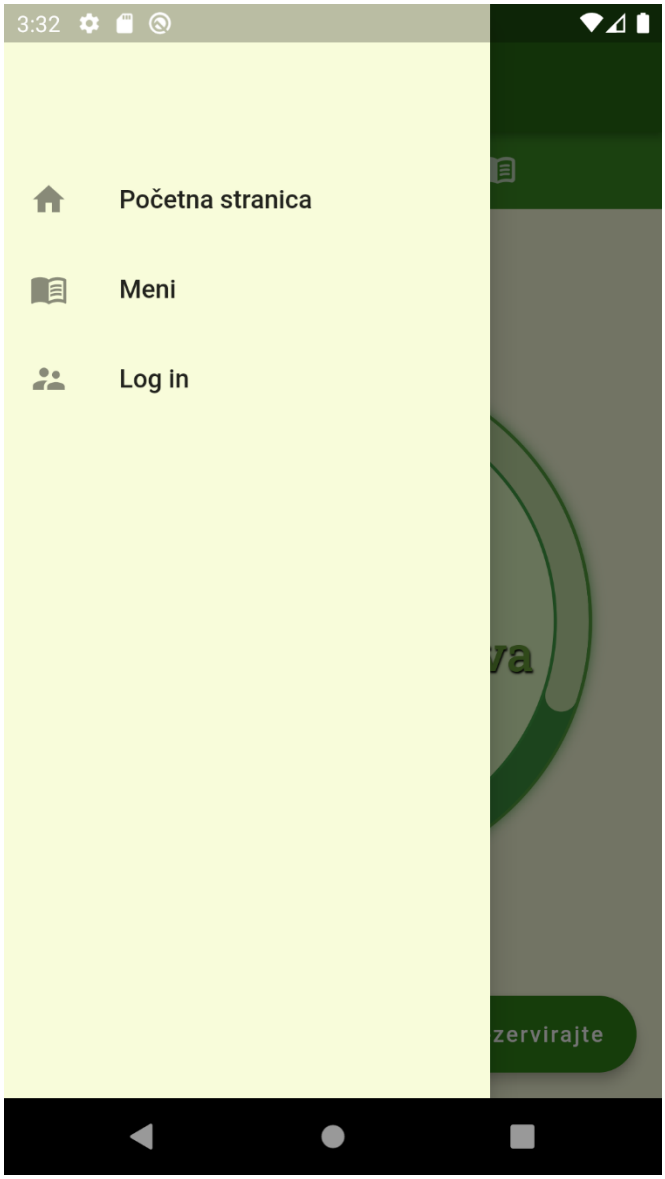
Kategorije su fiksno zadane u programu u listi „categories“ (Slika 26), potreba za promjenom ili dodavanjem novih kategorija zahtjevala bi promjene u samoj aplikacije budući da taj dio nije povezan s bazom podataka. Baza podataka se služi navedenom listom za pretragu hrane prema kategoriji. Prikaz jelovnika hrane u sučelju je vidljiv na slici 27.

```
List categories = ['Mesna jela', 'Grill', 'Riblja jela', 'Pizze', 'Deserti'];
```

Slika 26: Prikaz liste "categories"

Slika 27: Prikaz menija hrane u sučelju

Izbornik ili *drawer* prikazuje se pozivom widgeta „Drawer“ te korisniku daje mogućnost navigacije prema početnoj stranici, stranici menija i stranici za prijavu. Njegov prikaz se sastoji od *widgeta* „ListView“ u kojemu su „ListTileovi“ za svaki odabir te liste. Prikaz izbornika može se vidjeti na slici 28.



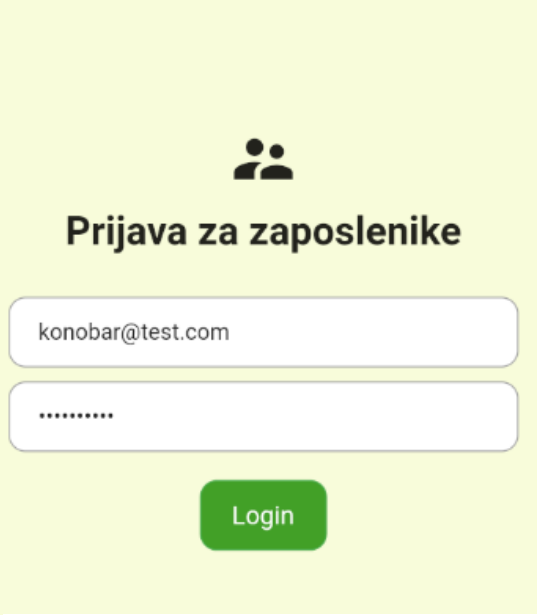
Slika 28: Prikaz drawera

3.5. Prijava korisnika

Sama izvedba prijave korisnika zahtjeva uporabu autentifikacije Firebasea i prihvaćanje upisanog teksta na polju za email i password. Za prihvaćanje upisa koristi se „TexEditingtController“ koji sprema upis u varijablu te je šalje na provjeru Firebaseu. U slučaju gdje dođe do greške prilikom upisa ili su upisani prijavni podaci netočni, prikazuje se obavijest na ekranu i korisnik ima mogućnost ponovnog upisa. Ako je prijava uspješno obavljena, poziva se početni *widget* „MyApp“ i korisnika se šalje na prikaz sukladno njegovoj ulozi. Na slikama 29 i 30 prikazan je kod i izgled prijavne stranice.

```dart
Future signIn() async {
  showDialog(
    useRootNavigator: false,
    context: context,
    builder: (context) {
      return const Center(child: CircularProgressIndicator());
    },
  );
  try {
    await FirebaseAuth.instance.signInWithEmailAndPassword(
        email: _emailController.text.trim(),
        password: _passwordController.text.trim());
    if (context.mounted) {
      Navigator.pop(context);
      Navigator.pushReplacement(
          context,
          MaterialPageRoute(
            builder: (context) => const MyApp(),
          )); // MaterialPageRoute
    }
  } catch (err) {
    Navigator.pop(context);
    showDialog(
        context: context,
        builder: (context) {
          Future.delayed(const Duration(milliseconds: 1500), () {
            Navigator.of(context).pop(true);
          }); // Future.delayed
          return const AlertDialog(
            title: Text('Invalid login information.'),
          ); // AlertDialog
```

Slika 29: Prikaz koda za prijavu korisnika



Slika 30: Prikaz sučelja za prijavu

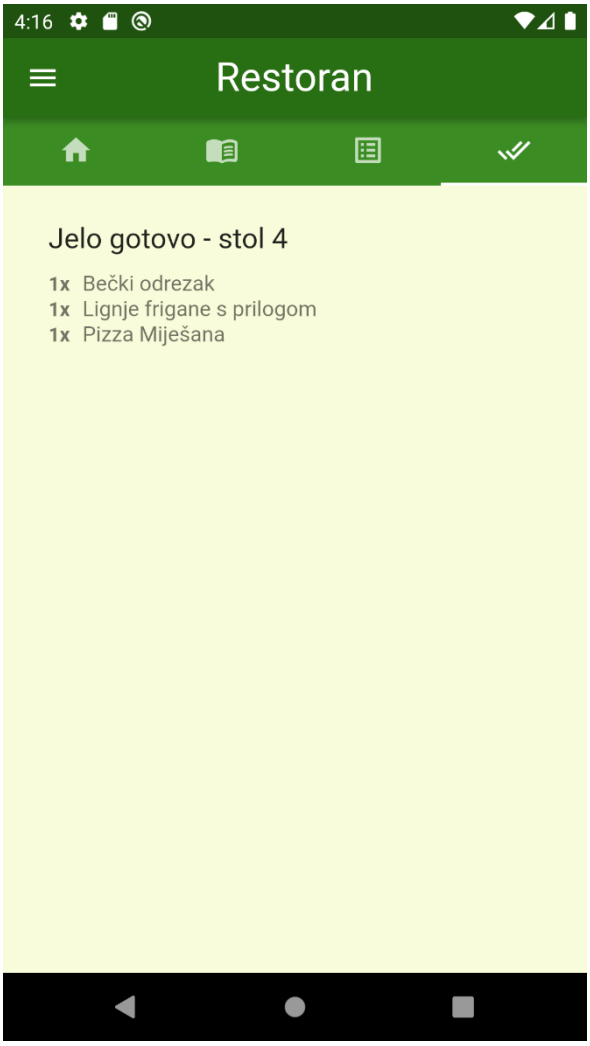
3.5.1. Uloga konobar

Uloga konobar ima dvije dodatne kartice u svom prikazu, a one su stolovi i gotova jela. Kartica stolova prikazuje deset stolova koji su fiksno određeni unutar programa. Stolovi su označeni brojevima i njihova boja označava jesu li slobodni ili zauzeti.

Odabirom stola konobar može rezervirati stol čime se stol prikazuje kao zauzet. Konobar ima i mogućnost unijeti narudžbu za stol i na taj način se stol automatski prikazuje kao zauzet. Omogućeno je dodavanje količine za stavke narudžbi i njihovo brisanje, a apsolutno brisanje svih narudžbi označenog stola moguće je gumbom za oslobađanje stola koje se potom pretvara u gumb za rezerviranje stola.

Pregled gotovih jela prima sva jela koja imaju svojstvo „finished“ postavljeno na *true*. Samo dodavanje nove narudžbe navedeno svojstvo postavlja na *false* i ono se mijenja tek kad kuhar označi narudžbu gotovom.

Slika 31: Prikaz sučelja konobar - stolovi

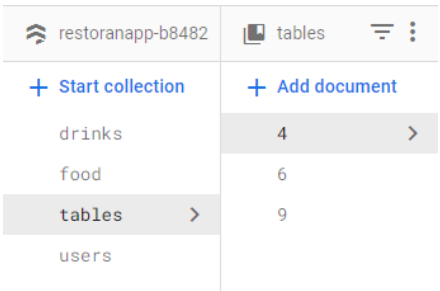


Slika 32: Prikaz sučelja konobar - gotove narudžbe

Izrada prikaza svih stolova i njihovih boja zahtjeva korištenje *widgeta* „GridView“ koji za svaki od deset stolova u Cloud Firestoreu provjerava postojanje dokumenta pod njihovim brojem, ako dokument postoji gumb za stol postaje crven. Sljedeće slike prikazuju kod za sučelje svih stolova (Slika 33) i prikaz baze podataka u kojoj su dodana tri stola (Slika 34).

```
return FutureBuilder<bool>(
    future: docExists('$count'),
    builder: (context, snapshot) {
      if (snapshot.connectionState == ConnectionState.waiting) {
        return ElevatedButton(
            style: ElevatedButton.styleFrom(
                backgroundColor: Colors.grey.shade400),
            onPressed: () {
              Navigator.push(
                  context,
                  MaterialPageRoute(
                      builder: (context) => KonobarTable('$count',
                          notifyParent: refresh))); // KonobarTable // MaterialPageRoute
            },
            child: const Stack(
              alignment: Alignment.center,
              children: <Widget>[
                Icon(
                  Icons.table_bar,
                  size: 90,
                ), // Icon
                Center(
                  child: CircularProgressIndicator(
                      valueColor: AlwaysStoppedAnimation<Color>(
                        Colors.grey,
                      ), // AlwaysStoppedAnimation
                      color: Colors.grey,
                      backgroundColor: Colors.white), // CircularProgressIndicator
                ) // Center
              ], // <Widget>[]
            )); // Stack // ElevatedButton
      }
      if (snapshot.data == true) {
        tableColor = Colors.red.shade700;
      }
      return ElevatedButton(
          style: ElevatedButton.styleFrom(
              backgroundColor: tableColor),
          onPressed: () {
            Navigator.push(
                context,
                MaterialPageRoute(
                    builder: (context) => KonobarTable('$count',
                        notifyParent: refresh))); // KonobarTable // MaterialPageRoute
          },
```

Slika 34: Prikaz koda sučelja - stolovi



Slika 33: Prikaz stolova na Cloud Firestoreu

Dio za prikaz, dodavanje novih narudžbi i brisanje starih implementiran je koristeći „CheckboxListTile“ *widget*. Budući da dodavanje narudžbi za hranu i piće funkcionira na isti način, u ovom primjeru biti će opisan rad s narudžbom pića. Ovisno o *bool* vrijednosti, koja je određena ako lista koja poprima elemente *streama* „selectedItemsDrinksStream“ sadrži vrijednost identifikacijskog broja dokumenta za određenu narudžbu, prikazuje se označena stavka narudžbe.

Funkcije za dodavanje i brisanje narudžbi izvršavaju se korištenjem lebdećih gumbova. Dodavanje nove narudžbe započinje prikazom privremenog prozora pomoću funkcije „showDialog“ koja unutar sebe poziva *widget* „DropdownSearch“. Uporaba „DropdownSearch“ *widgeta* zahtijeva instalaciju paketa pod istim nazivom, a njegova svrha je omogućiti odabir željenih stavki iz prikaza liste pića koja se nalaze u kolekciji baze podataka. Za sve označene stavke putem „DropdownSearch“ *widgeta*, njihovo ime i količina dodana je u novu kreiranu kolekciju *drinks* unutar dokumenta trenutačnog stola.

Označavanje stavke narudžbe i potom pritisak na lebdeći gumb znaka plus povećava količinu za označeni element, a pritisak na gumb znaka minus smanjuje količinu. Ako je količina 1, gumb minus će izbrisati tu stavku narudžbe. Lebdeći gumb koji prikazuje ikonu kante za smeće trenutno izbriše sve označene narudžbe.

Prilikom zauzimanja ili brisanja stola poziva se funkcija koja ponovo postavlja stanje prethodnog *widgeta* koji je zaslužan za prikaz gumba stolova. Razlog tome je što „FutureBuilder“ koji provjerava postoji li stol zahtjeva ponovno postavljanje stanja kako bi mogao prikazat ažurirano stanje stolova. Neki od glavnih dijelova koda i primjena u aplikaciji prikazani su na slikama u nastavku.

```
class _KonobarTableState extends State<KonobarTable> {
  late String name;

  List<String> selecteditemsDrinks = [];
  List<String> selecteditemsFood = [];

  final StreamController<List<String>> _selecteditemsDrinks =
      StreamController<List<String>>.broadcast();

  Stream<List<String>> get selectedItemsDrinksStream =>
      _selecteditemsDrinks.stream;

  final StreamController<List<String>> _selecteditemsFood =
      StreamController<List<String>>.broadcast();

  Stream<List<String>> get selectedItemsFoodStream => _selecteditemsFood.stream;

  IconData? icon;
  String? iconlabel;
  bool isChecked = false;
  final CollectionReference tableCollection =
      FirebaseFirestore.instance.collection('tables');
  final _searchTextController = TextEditingController();
```

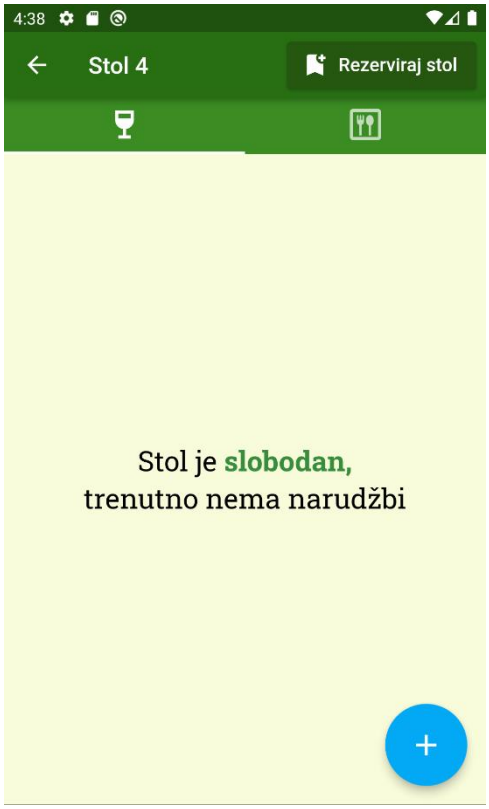
Slika 35: Prikaz listi i Streamova označenih stavki narudžbi

```
child: TabBarView(
  children: [
    TableDrinks(
        name,
        tableCollection,
        drinksCollection,
        selecteditemsDrinks,
        _selecteditemsDrinks,
        selectedItemsDrinksStream,
        _searchTextController,
        widget.notifyParent), // TableDrinks
    TableFood(
        name,
        tableCollection,
        foodCollection,
        selecteditemsFood,
        _selecteditemsFood,
        selectedItemsFoodStream,
        _searchTextController,
        widget.notifyParent) // TableFood
  ],
), // TabBarView
```

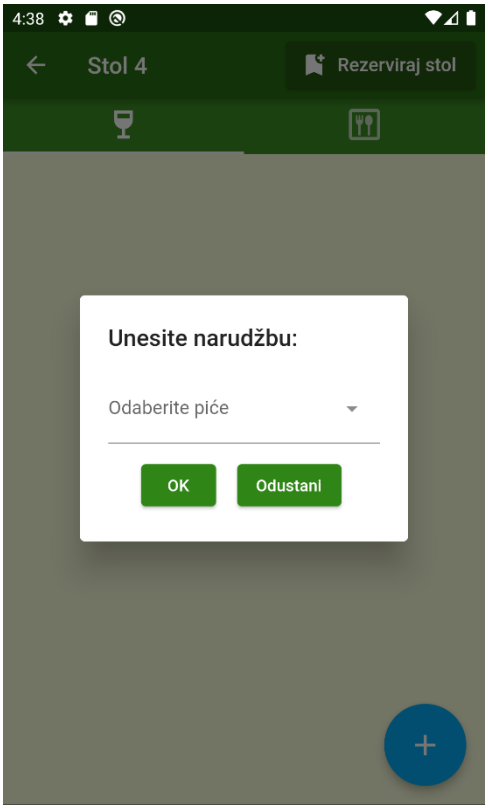
Slika 36: Prikaz koda pozivanja widgeta i njihovih argumenata potrebnih za prikaz i označivanje stavki

```
return StreamBuilder<List<String>>(
    initialData: const [],
    stream: selectedItemsDrinksStream,
    builder: (BuildContext context,
        AsyncSnapshot<List<String>> snapshots) {
      selecteditemsDrinks =
          snapshots.data?.toList() ?? [];
      return ListView.builder(
          padding: const EdgeInsets.only( // EdgeInsets.or
          shrinkWrap: true,
          key: UniqueKey(),
          itemCount: orderList.length,
          itemBuilder: (context, index) {
            return Column(
              children: [
                CheckboxListTile(
                    value: selecteditemsDrinks
                        .contains(orderList[index].id),
                    title: Text( // Text ···
                    subtitle: Text( // Text ···
                    onChanged: (isChecked) {
                      if (isChecked == true) {
                        selecteditemsDrinks
                            .add(orderList[index].id);
                        _selecteditemsDrinks.sink
                            .add((selecteditemsDrinks));
                      } else {
                        selecteditemsDrinks
                            .remove(orderList[index].id);
                        _selecteditemsDrinks.sink
                            .add((selecteditemsDrinks));
                      }
                    }), // CheckboxListTile
                const Divider()
```

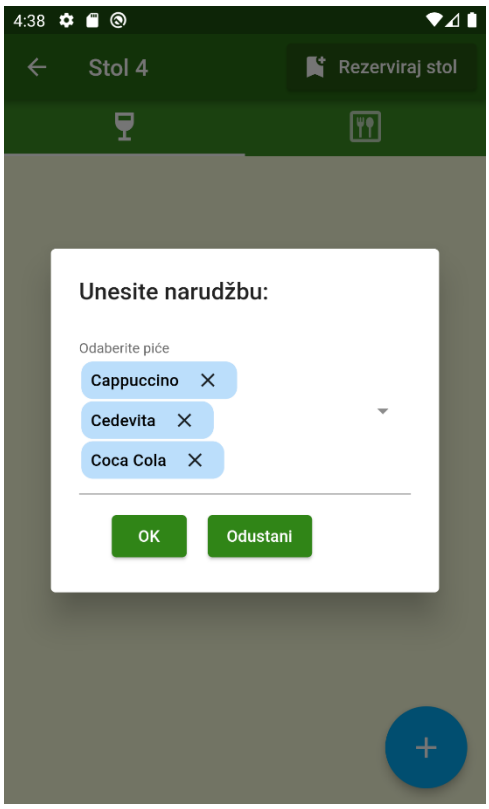
Slika 37: Prikaz koda widgeta "StreamBuilder" koji uzima stream označenih stavki



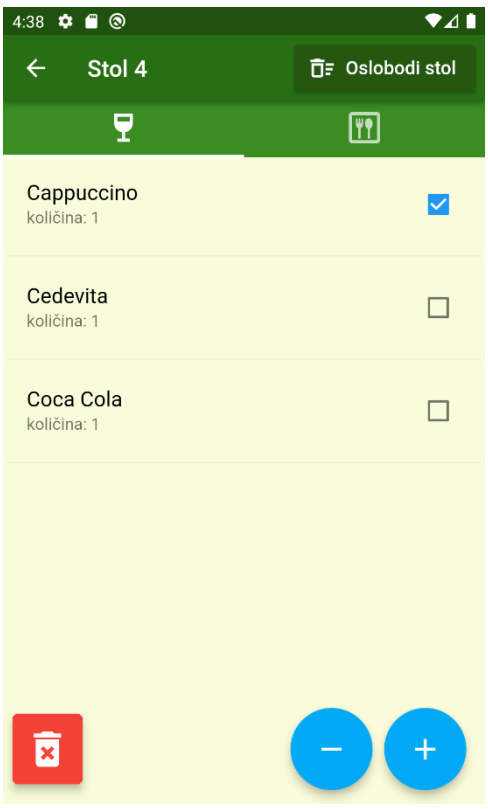
Slika 39: Prikaz slobodnog stola



Slika 38: Prikaz dodavanja narudžbi stola



Slika 41: Prikaz prihvaćanja narudžbi



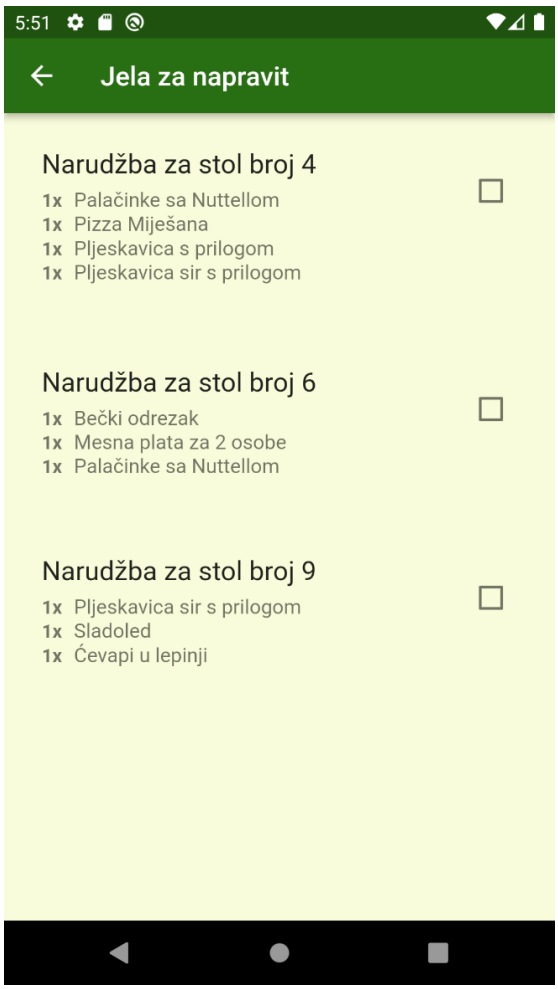
Slika 40: Prikaz rada s označenim narudžbama

3.5.2. Uloga kuhar

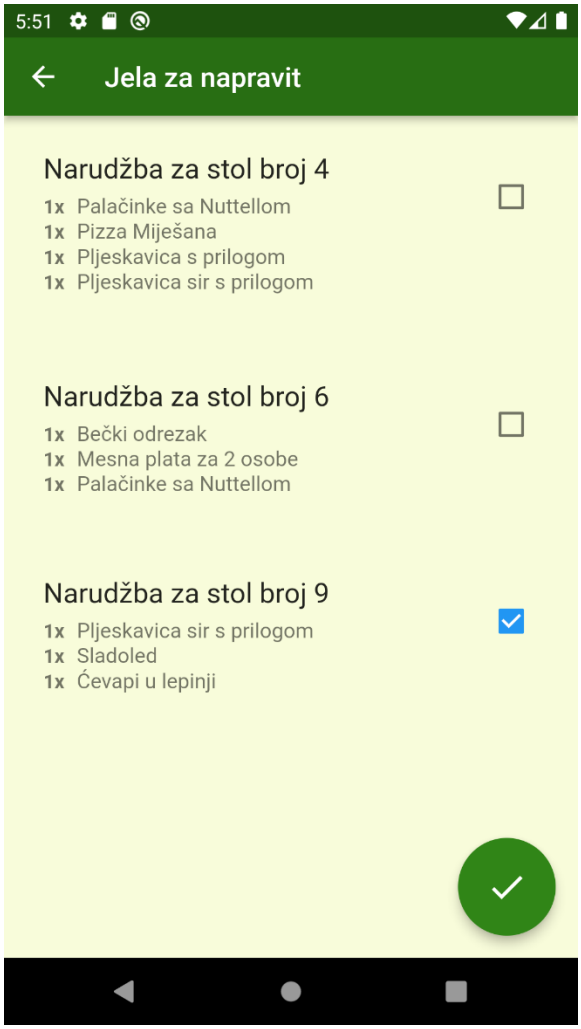
Uloga kuhara daje pregled jela koja je potrebno pripremiti i povijest gotovih jela. Kuhar prima sva naručena jela koja nisu označena kao gotova. Kad završi s pripremom jela, kuhar označi narudžbu koju je završio i u Cloud Firestore-u za dokumente te narudžbe atribut *finished* postavlja se na *true*. Takve narudžbe se ne prikazuju u prikazu jela za pripremu već se njihov prikaz pojavljuje u povijesti gotovih jela.

Implementacija prikaza jela za pripremu slična je prikazu narudžbi za stol konobar, koristi *widget* „CheckboxListTile“ i *stream* za dohvaćanje označenih jela. Slike u nastavku prikazuju interakciju sa sučeljem kuhara.

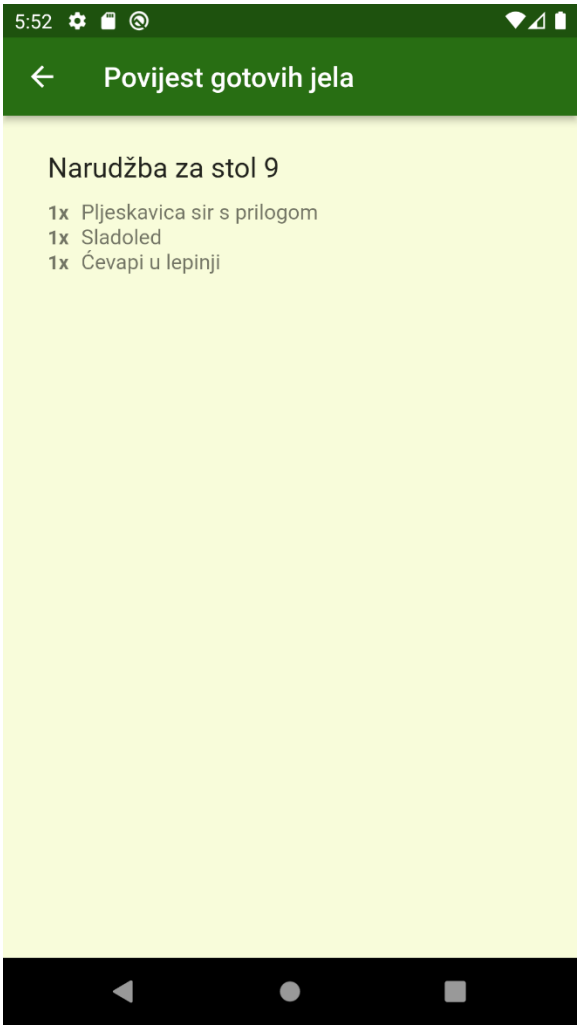
Slika 42: Prikaz sučelja kuhara - 1



Slika 43: Prikaz sučelja kuhara - 2



Slika 44: Prikaz potvrde završenog jela



Slika 45: Prikaz završenog jela u sekciji povijest gotovih jela

3.5.3. Uloga administrator

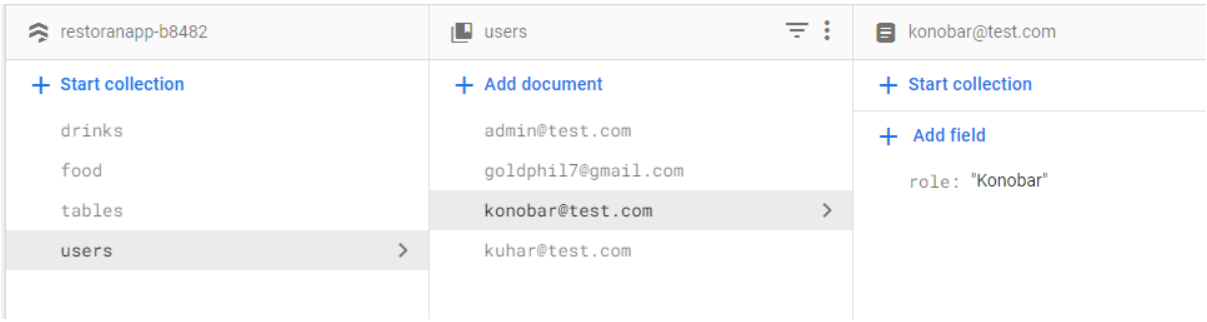
Administrator ima pristup svim prikazima, može vidjeti, dodavati, brisati narudžbe, i označavati gotova jela te vidjet njihovu povijest. Uz navedeno, administrator ima funkciju dodavanja, uređivanja i brisanje postojećih računa iz baze podataka, što uključuje mijenjanje njihovih uloga. Brisanje računa nije sasvim moguće, već je samo izbrisan korisnik iz baze podataka. Tim načinom ako se korisnik želi prijaviti s obrisanim računom on će imati isti prikaz kao i neprijavljeni korisnik. Za potpuno brisanje računa potrebno je koristiti uslugu Firebase-a.

Implementacija dodavanje novih korisničkih računa slična je prijavljivanju korisnika, uz dva nova polja, „Confirm Password“ i „Uloga“. Nakon što su uneseni podaci u sva polja, provjeravaju se moguće greške. Ako je registracija korisnika uspješna u Cloud Firestore bazi podataka dodaje se novi dokument u kolekciji *users* čiji je naziv jednak e-mailu korisnika i unutar njega je polje *role* koje označava ulogu.

Prikaz koda za registraciju, baze podataka i sučelje dodavanja korisnika prikaza je na slikama u nastavku.

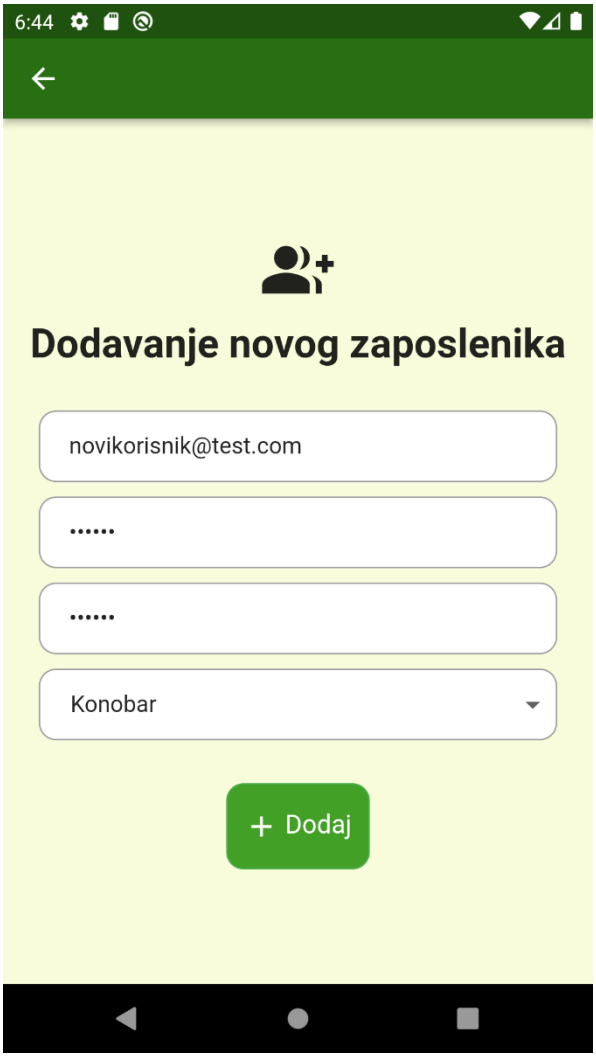
```
Future register() async {
  if (passwordConfirmed()) {
    showDialog(
      useRootNavigator: false,
      barrierDismissible: false,
      context: context,
      builder: (context) {
        return const Center(child: CircularProgressIndicator());
      },
    );
    if (_role != null && context.mounted) {
      FirebaseApp app = await Firebase.initializeApp(
          name: 'Secondary', options: Firebase.app().options);
      try {
        await FirebaseAuth.instanceFor(app: app)
            .createUserWithEmailAndPassword(
                email: _emailController.text.trim(),
                password: _passwordController.text.trim());
        await app.delete();
        addUserDetails(_emailController.text.trim(), _role!);
        if (context.mounted) {
          Navigator.of(context).pop(true);
          showDialog(
              barrierDismissible: false,
              context: context,
              builder: (context) {
                Future.delayed(const Duration(milliseconds: 1200), () {
                  Navigator.of(context).pop(true);
                  _emailController.clear();
                  _passwordController.clear();
                  _confirmpasswordController.clear();
                  setState(() {
                    _role = null;
                  });
                }); // Future.delayed
                return const AlertDialog(
                  content: Text('User has been added succesfully.'),
                ); // AlertDialog
              });
        }
      } on FirebaseAuthException catch (e) {
```

Slika 46: Prikaz isječka koda za registraciju novog korisnika



Slika 47: Prikaz kolekcije users unutar Cloud Firestorea

Slika 49: Prikaz sučelja administatora

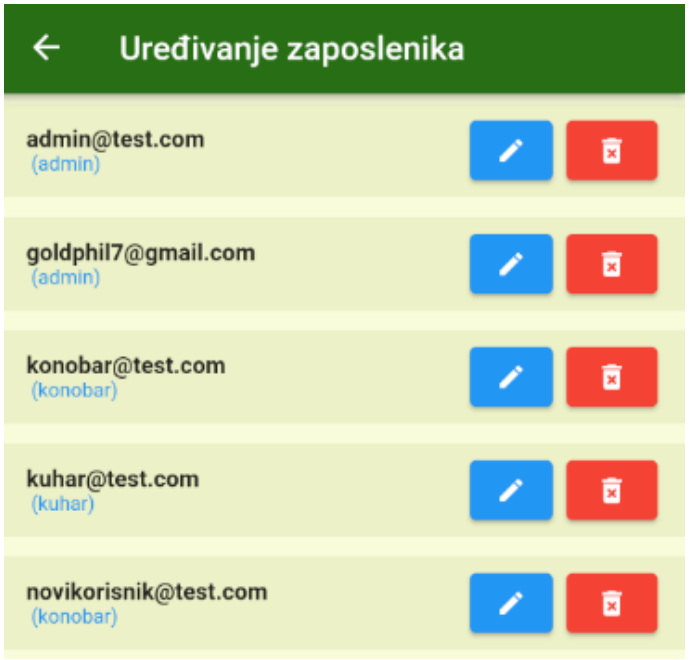


Slika 48: Prikaz sučelja dodavanja novog korisnika

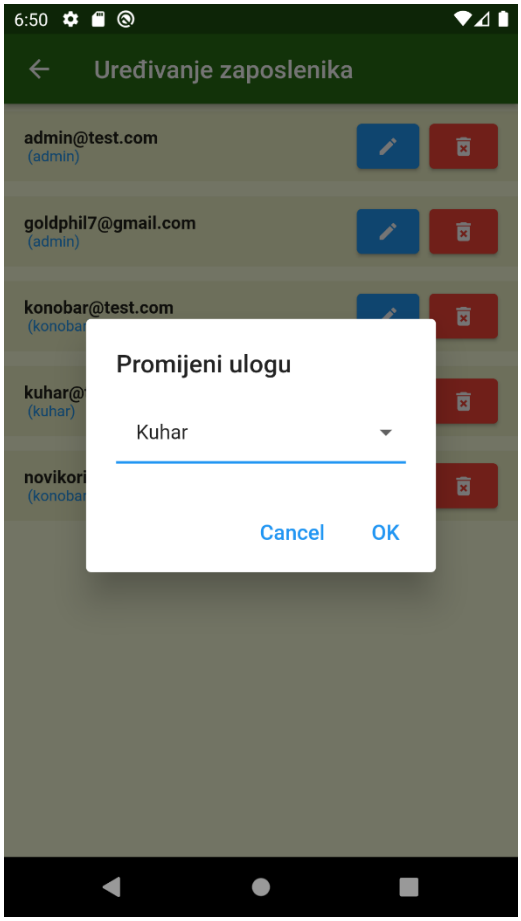
Uređivanje postojećih korisnika implementirano je prikazom „ListView“ *widgeta* koji uzima podatke od „FutureBuilder“ *snapshota* kojim je dohvaćeno identifikacijsko ime dokumenta koje je u ovom slučaju označeno e-mailom od pojedinog računa. Ovisno o potrebi, radnje brisanja i uređivanja uloge odvijaju se pozivanjem funkcija „editUserDetails“ i „deleteUser“ koji su tipa *future*. Svaki put kad dođe do korištenja tih funkcija odvija se ponovni postupak postavljanja stanja funkcijom „setState“ zbog potrebe prikaza ažuriranih podataka. U sljedećim slikama prikazani je isječak koda za funkciju za uređivanje i rad u sučelju.

```
class _EditUsersPageState extends State<EditUsersPage> {
  List<String> docIDs = [];
  String? _role;

  Future getDocId() async {
    docIDs.clear();
    await FirebaseFirestore.instance
        .collection('users')
        .get()
        .then((snapshot) => snapshot.docs.forEach((element) {
              docIDs.add(element.reference.id);
            }));
  }

  Future deleteUser(String email) async {
    await FirebaseFirestore.instance.collection('users').doc(email).delete();
  }

  Future editUserDetails(String email, String role) async {
    await FirebaseFirestore.instance.collection('users').doc(email).set({
      'role': role,
    });
  }
```

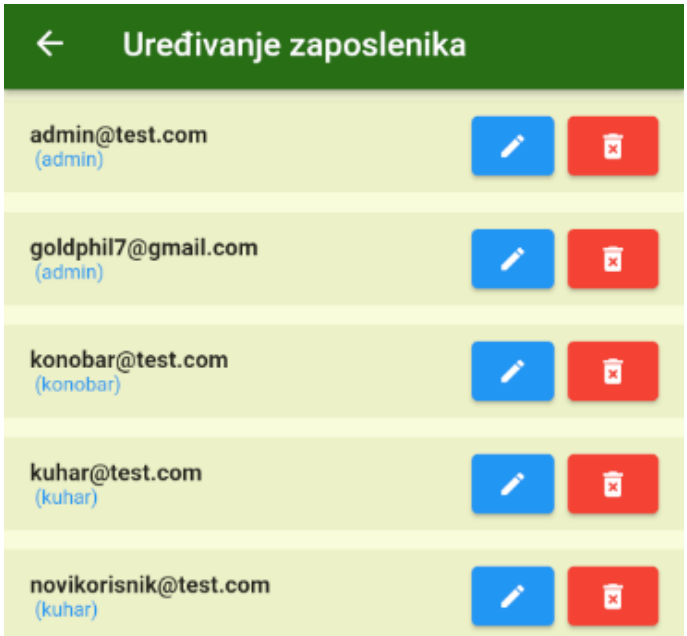
Slika 50: Isječak koda korištenih funkcija za uređivanje korisnika



Slika 51: Sučelje uređivanje zaposlenika



Slika 52: Mijenjanje uloge za računa novikorisnik@test.com



Slika 53: Uloga za račun novikorisnik@test.com je promijenjena

4. Zaključak

Razvojni okvir Flutter i usluge Firebase-a su izvrsno poslužili pri izradi aplikacije. Sama aplikacija bi mogla pojednostaviti rad zaposlenika restorana, ali i poslužiti kao jednostavni pregled slobodnih stolova, telefonskog broja za rezervaciju, te jelovnika hrane i piće za gosta. Međutim, moguće je i unaprijediti aplikaciju tako da se ne koriste fiksni elementi iz koda programa već bi sve trebalo biti unutar baze podataka na Firebaseu. Također, administrator ne može obrisati korisnika direktno iz aplikacije.

Flutter ima prednost kao što su jednostavnost, fleksibilnost i brzina razvoja, ali potrebno je navesti i njegove nedostatke. Veličina aplikacije tokom razvoja nije optimalna, a podaci koji zauzimaju puno prostora mogu stvorit probleme za neke razvojne programere. Iako je popularan, Flutter sam po sebi je nov i potreban mu je još veći broj biblioteka za razvojne funkcije. Valja napomenut da su mogući problemi s iOS sustavima pošto je Flutter razvijen od Googlea koji pridaje veću važnost Android uređajima.[1]

[1] **Montaño, Daniela.** Why Use Flutter: Pros and Cons of Flutter App Development. [Mrežno] 7. srpanj 2023. [Pokušaj pristupa: 19. srpanj 2023.] https://waverleysoftware.com/blog/why-use-flutter-pros-and-cons/; **Serrano, David.** The dark side of Flutter: 4 main drawbacks to take into account. [Mrežno] 22. siječanj 2022. [Pokušaj pristupa: 20. srpanj 2023.] https://davidserrano.io/the-dark-side-of-flutter-4-inconveniences-that-every-flutter-developer-should-know.

5. Popis slika

6. Literatura

1. **V, Tejashwini.** Introduction to Flutter Widgets - Edureka. [Mrežno] [Pokušaj pristupa: 17. srpnja 2023.] https://www.edureka.co/blog/flutter-widgets/.

2. **Ford, Star.** A Dart Language Guide for C# and Java Developers. [Mrežno] [Pokušaj pristupa: 18. srpanj 2023.] https://www.toptal.com/dart/dartlang-guide-for-csharp-java-devs.

3. **Sullivan, Matt.** Flutter: Don't Fear the Garbage Collector. [Mrežno] 4. siječanj 2019. [Pokušaj pristupa: 17. srpanj 2023.] https://medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30.

4. **nepoznat autor.** Dart programming language. [Mrežno] [Pokušaj pristupa: 18. srpanj 2023.] https://dart.dev/.

5. **nepoznat autor**. Firebase | Google's Mobile and Web App Development Platform. [Mrežno] [Pokušaj pristupa: 18. srpanj 2023.] https://firebase.google.com/.

6. **nepoznat autor**. Meet Android Studio. [Mrežno] [Pokušaj pristupa: 18. srpanj 2023.] https://developer.android.com/studio/intro.

7. **nepoznat autor**. Dart overview. [Mrežno] [Pokušaj pristupa: 18. srpanj 2023.] https://dart.dev/overview.

8. **nepoznat autor** Visual Studio Code - Code Editing. Redefined. [Mrežno] [Pokušaj pristupa: 18. srpanj 2023.] https://code.visualstudio.com/.

9. **Montaño, Daniela.** Why Use Flutter: Pros and Cons of Flutter App Development. [Mrežno] 7. srpanj 2023. [Pokušaj pristupa: 19. srpanj 2023.] https://waverleysoftware.com/blog/why-use-flutter-pros-and-cons/.

10. **nepoznat autor.** StreamBuilder class - widgets library - Dart API. [Mrežno] [Pokušaj pristupa: 18. srpanj 2023.] https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html.

11. **nepoznat autor**. FutureBuilder class - widgets library - Dart API. [Mrežno] [Pokušaj pristupa: 18. srpanj 2023.] https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html.

12. **Serrano, David.** The dark side of Flutter: 4 main drawbacks to take into account. [Mrežno] 22. siječanj 2022. [Pokušaj pristupa: 20. srpanj 2023.] https://davidserrano.io/the-dark-side-of-flutter-4-inconveniences-that-every-flutter-developer-should-know.