MOBILNA APLIKACIJA ZA INTERNET TRGOVINU

Pepić, Robert

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

Sveučilišni studij

MOBILNA APLIKACIJA ZA INTERNET TRGOVINU

Završni rad

Robert Pepić

Osijek, 2021.

SADRŽAJ

1. UVOD

Tema ovog završnog rada je „Mobilna aplikacija za internet trgovinu”. Cilj je izraditi mobilnu aplikaciju za internet trgovinu korištenjem Flutter tehnologije. Flutter je paket za izradu višeplatformnih aplikacija kojeg je razvio Google, a služi za izradu aplikacija koje mogu biti namijenjene za Android, iOS, Linux, Mac i Windows operacijske sustave. Programski jezik u kojem se izrađuju Flutter aplikacije zove Dart. Dart je objektno orijentirani programski jezik zasnovan na klasama sa sintaksom u C stilu. U teorijskom dijelu rada detaljnije će se razmotriti način na koji Flutter radi, kako se pomoću njega izrađuju aplikacije, te najvažnije dijelove svake Flutter aplikacije. Također će se proučiti Firebase, baza podataka koja će biti korištena kao pozadinski sustav aplikacije. Firebase nudi velik broj usluga za razne potrebe, od kojih će se prvenstveno koristiti registriranje i prijava korisnika, te pohrana njihovih podataka. Na kraju će se pokazati kako se pomoću FlutterFirea, skupa Flutter dodataka, povezuje Flutter aplikacija s Firebaseom. U praktičnom dijelu rada cilj je pomoću Fluttera i Firebasea izraditi mobilnu aplikaciju za internet trgovinu koja će korisniku omogućiti kreiranje korisničkog računa, prijavu u svoj račun, pregled svih proizvoda, stavljanje odabranih proizvoda u košaricu i narudžbe proizvoda. Dodatne mogućnosti aplikacije su pretraga proizvoda po kategorijama, te ocjenjivanje i komentiranje proizvoda.

1.1 Zadatak završnog rada

U teorijskom dijelu rada potrebno je proučiti tehnologije potrebne za razvoj mobilne aplikacije i pozadinskog sustava za upravljanje internet trgovinom. U praktičnom dijelu rada cilj je izraditi mobilnu aplikaciju s Flutter tehnologijom koja će omogućiti sljedeće funkcionalnosti: kreiranje korisničkog računa i prijavu u aplikaciju, izbornik s kategorijama proizvoda, listu proizvoda za svaku kategoriju s ocjenom i komentarima, mogućnost stavljanja odabranih proizvoda u košaricu i narudžbe proizvoda, ocjenjivanje i komentiranje proizvoda, te pohranu korisničkih podataka u bazi podataka.

2. NATIVNE I VIŠEPLATFORMSKE MOBILNE APLIKACIJE

2.1 Nativne mobilne aplikacije

Kada se govori o nativnim aplikacijama misli se na aplikacije koje su razvijene specifično za jedan operacijski sustav odnosno platformu. Takve aplikacije razvijaju se korištenjem posebnih razvojnih okolina i pišu se u jezicima specifičnim za ciljanu platformu. Primjerice, za razvoj aplikacije namijenjene za iOS operacijski sustav koristilo bi se xCode razvojno okruženje, a kod aplikacije pisao bi se u Objective-C programskom jeziku, dok bi se za razvoj Android aplikacije koristio Android Studio i Java programski jezik. Budući da su one posebno dizajnirane za određeni uređaj, imaju potpunu slobodu korištenja značajki prisutnih na uređaju, kao što su kamera, GPS i Bluetooth. Nativne aplikacije komuniciraju direktno sa operacijskim sustavom i hardverom uređaja, pa su one po pitanju performansi nedvojbeno najsuperiornije. Iz tog razloga se aplikacije koje zahtijevaju veću procesorsku snagu u pravilu razvijaju nativno.

Unatoč svim navedenim prednostima, nativne aplikacije također imaju i svoje mane. Prva od kojih je visoka cijena razvoja i održavanja aplikacije. Prema [1] sama cijena održavanje nativnih aplikacija iznosi otprilike 20% novca uloženog u sam razvoj aplikacije. To znači što je skuplji razvoj aplikacije, skuplje je i održavanje. Druga velika mana nativnih aplikacija je vrijeme razvoja, jer kod napisan za jednu platformu nije moguće iskoristiti za razvoj aplikacije na drugoj platformi. Ako se primjerice aplikaciju nativnu Android operacijskom sustavu želi prebaciti na iOS operacijski sustav, mora ju se u potpunosti ispočetka razviti. To znači da se ista aplikacija ustvari razvija dva ili više puta, ovisno o tome na koliko platformi ju se želi implementirati, što uvelike povećava vrijeme, a time i cijenu razvoja. Upravo ovaj problem vremena i cijene razvoja pokušava se riješiti alatima za izradu višeplatformskih aplikacija.

2.2 Višeplatformske mobilne aplikacije

Razvoj višeplatformskih mobilnih aplikacije proces je stvaranja mobilnih aplikacija koje se mogu primijeniti ili objaviti na više platformi pomoću jednom napisanog koda, umjesto da se aplikacija mora razvijati više puta pomoću odgovarajućih nativnih tehnologija za svaku platformu. Prednost ovakvog pristupa razvoja je puno kraće vrijeme i puno manji troškovi razvoja aplikacije. Osim toga ovakve aplikacije brže su dostupne široj publici, što je još jedan dodatni plus ako je ovaj faktor bitan proizvođaču. Nedostatci ovakvih aplikacija su lošije performanse zbog problema

s integracijom s određenim operativnim sustavima. To nastaje zbog nedostatka kompatibilnosti između izvornih i stranih komponenti uređaja na kojima radi. Ove aplikacije imaju slabije performanse u usporedbi s nativnim verzijama. Postoje različiti alati, odnosno radni okviri (engl. *framework*) dizajnirani specifično za razvoj višeplatformskih aplikacija. Svaki od njih ima drugačiji pristup rješavanju ovog problema. Prema [2] najpopularniji alati za razvoj višeplatformskih aplikacija u 2021. godini su Flutter, React Native i Ionic.

Flutter je besplatni okvir otvorenog koda razvijen od strane Googlea i najmlađi je od navedena tri alata. Omogućuje razvoj mobilnih aplikacija pomoću jedne baze koda pisane u Dart programskom jeziku. Ionic omogućuje kreiranje nativnih aplikacija, ali to radi stvaranjem web aplikacije s HTML-om, CSS-om i JavaScript-om. Nakon što je web aplikacija spremna, omotana je okvirom Cordova, koji će zatim aplikaciju generirati u izvornom WebViewu. React Native je softverski okvir otvorenog koda kojeg je izradio Facebook i donedavno je bio najpopularniji alat za izradu višeplatformskih aplikacija namijenjenih za Android, Android TV, iOS, macOS, tvOS, Web i Windows operacijske sustave. No prema [3], mjesto najpopularnijeg alata za izradu višeplatformskih aplikacija u 2021. godini preuzeo je Flutter. Upravo zbog svoje popularnosti i jednostavnosti korištenja Flutter je odabran kao alat koji će se koristiti u izradi ovog rada.

2.3 Internet trgovine

Kupovina putem interneta posljednjih je godina doživjela velik porast u popularnosti i postala je omiljeni način kupnje mnogih kupaca. Postoje mnoge varijacije i konkretne implementacije internet trgovina. One se mogu razlikovati po raznim obilježjima kao što su veličina trgovine, područje djelovanja, vrsta proizvoda kojima se bave, usluge koje nude i slično. Za primjer će se uzeti dvije koje su trenutno među najvećim i najpopularnijim na svijetu, a to su eBay i Amazon. Dvije, na prvi pogled, vrlo slične internet trgovine koje se razlikuju u nekoliko bitnih pogleda. Obje rade na globalnoj razini i nude velik izbor različitih proizvoda, no Amazon funkcionira više kao tradicionalna trgovina, dok eBay funkcionira više kao aukcijska kuća. Dodatna razlika među njima je i njihov način poslovanja. Kada kupci posjećuju Amazon web stranicu ili se služe njihovom aplikacijom oni ustvari pregledavaju proizvode koje Amazon održava kao zalihe u svojoj velikoj mreži skladišta. S druge strane eBay funkcionira kao posrednik koji jednostavno olakšava prodaju robe između kupaca i prodavača. Obje trgovine također nude korisnicima mogućnost prodaje vlastitih proizvoda.

Za razliku od ovih velikih, globalnih internet trgovina koje imaju širok spektar različitih proizvoda, postoje naravno i manje, specijalizirane internet trgovine. One se obično bave određenom kategorijom proizvoda i djeluju na manjem području, primjerice unutar države ili grada. Primjeri ovakvih domaćih internet trgovina bili bi Links, Instar i Vacom. Sve tri specijalizirane su u prodaji informatičkih proizvoda i opreme te nude dostavu svojih proizvoda na ograničenom području, odnosno unutar države. Zbog jednostavnosti i lakše realizacije ovim radom prikazati će se razvoj jedne mobilne aplikacije za internet trgovinu specijalizirane u prodaji informatičkih proizvoda. Svih pet navedenih i opisanih internet trgovina, unatoč razlikama, imaju nekolicinu zajedničkih značajki koje će biti implementirane u aplikaciju koja se razvija. One su: Kreiranje korisničkog računa, pregled i uređivanje korisničkih podataka, pregled proizvoda po kategorijama i filtriranje proizvoda, prikaz detalja o proizvodu, stavljanje odabranih proizvoda u košaricu, narudžba proizvoda, pregled narudžbi te komentiranje i ocjenjivanje proizvoda. Funkcionalnosti kao što su transakcije i dostava proizvoda, naravno, neće biti implementirane.

3. FLUTTER

Alat korišten za izradu ovog rada, odnosno aplikacije za internet trgovinu je Googleov Flutter SDK (engl. *software development kit*). U početku zamišljen kao alat za izradu mobilnih aplikacija podržanih i na Androidu i na iOS-u, slično Facebookovom React Nativeu ili Microsoftovom Xamarinu, rastom popularnosti postepeno je i raslo njegovo područje primjene. Prvo proširenje u njegovoj primjeni odnosilo se na mogućnost korištenja gotovo istog koda za izradu web aplikacija, a nedavnim izlaskom njegove najnovije inačice Flutter 2.0, ta primjena se proširila i na izradu Linux, Mac i Windows desktop aplikacija. Upravo taj rast u popularnosti, te sve veće područje primjene neki su od glavnih razloga zašto je baš Flutter odabran za izradu ovog rada.

3.1 Arhitektura Fluttera

Arhitektura Fluttera, detaljnije opisana u [4], ima slojevitu strukturu koja se sastoji od tri glavna dijela, a oni su okvir (engl. *framework*), motor (engl. *engine*) i ugrađivač (eng. *embedder*). Okvir je najviši sloj Flutter arhitekture. U njemu se nalaze sve klase i biblioteke potrebno za kreiranje korisničkog sučelja (engl. user interface), kao što su biblioteke za vizualno uređivanje, biblioteke za animacije i biblioteke za prepoznavanje gesti. Ovo je sloj pomoću kojeg Flutter developeri grade Flutter aplikacije i to u Dart programskom jeziku.

Motor je srednji sloj i on predstavlja jezgru Fluttera. Zadužen je za razne zadatke kao što su prikaz i raspored elemenata na ekranu, način na koji se oni prikazuju, rad sa ulazno izlaznim tokovima, rad sa datotekama i rad sa mrežom. Ukratko, zadužen je za ispravno izvođenje Flutter aplikacija. Pisan je uglavnom u C++ programskom jeziku, a sa okvirom komunicira preko dart:ui biblioteke koja kod pisan u C++ „umota" u dart klase.Najniži sloj Flutter paketa je ugrađivač. On predstavlja sučelje između Flutter aplikacije i operacijskog sustava na kojem se ona izvodi i to na način da ju prikaže kao paket razumljiv tom operacijskom sustavu. Za svaku platformu koju Flutter podržava postoji poseban ugrađivač napisan u jeziku koji odgovara toj platformi: trenutno Java i C ++ za Android, Objective-C /Objective-C++ za iOS i macOS, te C ++ za Windows i Linux.

3.2 Izrada i glavni dijelovi Flutter aplikacija

Za izradu Flutter aplikacije, prije svega, potrebno ga je naravno preuzeti i instalirati. Paket sa instalacijom, zajedno sa svim uputama potrebnim za uspješnu instalaciju i uspostavljanje Flutter SDK-a dostupni su na [5]. Nadalje, potreban nam je i uređivač izvornog koda u kojem će se pisati izvorni kod aplikacije. Za izradu ovog rada koristiti će se *Visual Studio Code,* dostupan na [6]. Ako su svi uvjeti zadovoljeni, može se započeti sa izradom aplikacije, a to se radi na slijedeći način. U tetminal uređivača koda upisuje se sljedeća naredba: *„flutter create moja_aplikacija"*. Kao rezultat ove naredbe Flutter stvara novi projekt pod imenom *„moja_aplikacija",* te u njega generira sve potrebne mape i datoteke. Dvije najvažnije stvari u svakom Flutter projektu su *lib* mapa i *pubspec.yaml* datoteka. U *lib* mapu se pohranjuje sav izvorni kod aplikacije i to u obliku Dart datoteka, odnosno datoteka sa *.dart* ekstenzijom (*primjer.dart*). Unutar *lib* mape, datoteke se dodatno mogu organizirati u proizvoljno imenovane mape radi bolje preglednosti i lakšeg snalaženja u kodu. Unutar *pubspec.yamal* datoteke nalaze se sve ovisnosti i vanjski paketi (*engl. dependencies and external packages*), te meta – podaci vezani za aplikaciju kao što su korištena verzija Flutter i za koje je sve verzije ciljanih operacijskih sustava aplikacija podržana. Rad s njima biti će detaljnije prikazan u praktičnom dijelu rada.

4. FIREBASE

Kao pozadinski sustav (engl. *Backend*) aplikacije, koji će služiti kao baza podataka i biti zadužen za stvaranje korisničkih računa i autentikaciju korisnika, koristit će se Firebase. Firebase je, isto kao i Flutter, razvijen od strane Googlea, no oni su u potpunosti samostalni proizvodi neovisni jedan o drugome. Osim navedenih mogućnosti koje će se koristiti u izradi aplikacije, Firebase nudi još niz raznih usluga poput *push* notifikacija i analitike performansi aplikacije.

4.1 Uspostavljanje Firebase projekta i registracija aplikacije

Za korištenje Firebase-a prije svega potrebno je imati Google korisnički račun, te se s njim prijaviti na službenoj stranici Firebase-a. Nakon uspješne prijave odabire se opcija stvaranja novog projekta. Sam postupak kreiranja projekta vrlo je jednostavan i sastoji se od nekoliko koraka. Nakon što je projekt kreiran potrebno je u njemu registrirati našu aplikaciju. Prvo odabiremo operacijski sustav za koji je naša aplikacija namijenjena. Odabiremo android operacijski sustav. Nakon toga traži se da unesemo id aplikacije, koji je u našem slučaju *„com.example.techstreme“*. Zatim Firebase generira jednu *google-services.json* datoteku koju je potrebno preuzeti i premjestiti u *android/app* mapu našeg Flutter projekta. Za kraj, potrebno je dodati nekoliko paketa u *build.gradle* datoteku projekta. Cijeli ovaj postupak detaljno je objašnjen na [7] i s njim smo završili registraciju aplikacije.

4.2 Stvaranje pristupne točke

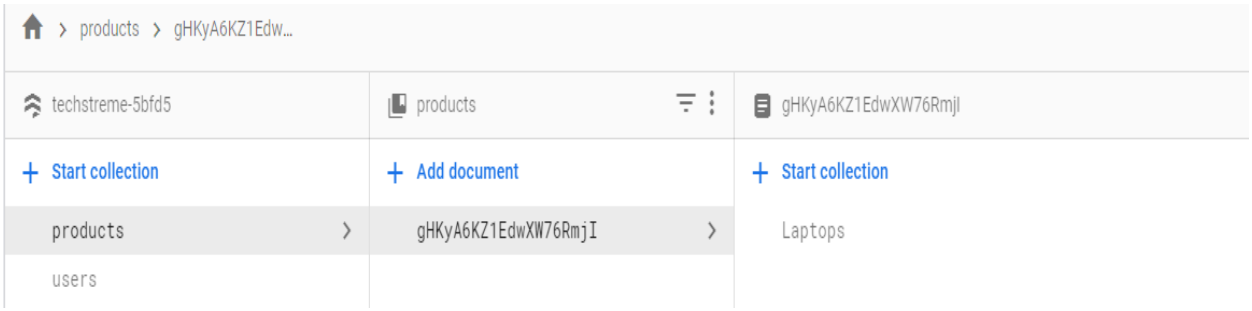
Kako bi se moglo služiti svim mogućnostima koje Firebase nudi potrebno je prvo dodati nekoliko paketa unutar *pubspec.yamal* datoteke Flutter projekta. Prvi i najvažniji paket koji se dodaje je *firebase_core* paket [8]. On je zadužen za stvaranje pristupne točke preko koje se odvija sva komunikacija između aplikacije i Firebase projekta. Ta pristupna točka se inicijalizira prije samog izvođenja aplikacije i to na način prikazan na slici ispod. Funkcija *„Firebase.initializeApp()“* zadužena je za samo stvaranje pristupne točke, dok će se funkcija *„WidgetsFlutterBinding.ensureInitialized();“* pobrinuti za to da se aplikacija ne pokrene prijevremeno, odnosno prije nego što je uspostavljena veza s Firebaseom.

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}
```

Slika 4.1 Inicijalizacija pristupne točke

4.3 Uspostavljanje baze podataka

Unutar Firebase projekta odabere se opcija *create database*. Odabirom ove opcije Firebase stvara novu bazu podataka. Baza podataka organizirana je kao niz kolekcija dokumenata. Svaka kolekcija unutar sebe može imati proizvoljan broj dokumenata, unutar kojih se mogu pohranjivati razni tipovi podataka. Unutar samih dokumenata mogu se stvarati i nove kolekcije u kojima se opet mogu pohranjivati dokumenti. Na slici 4.2 može se vidjeti primjer strukture Firebase baze podataka. Stvorena je kolekciju „products“, unutar koje se nalazi jedan dokument u kojem je pohranjena kolekcija „laptops“. U nju će se kasnije dodavati dokumenti koji će predstavljati laptope, a podaci unutar njih predstavljati će specifikacije pojedinog laptopa. Međutim, kako bi se moglo služiti bazom, kao i drugim mogućnostima Firebase-a unutar aplikacije, one se prvo moraju omogućiti dodavanjem određenih paketa unutar *pubspec.yamal* datoteke Flutter projekta. Za komuniciranje s bazom podataka potrebno je dodati *cloud_firestore* paket [9].



Slika 4.2 Struktura Firebase baze podataka

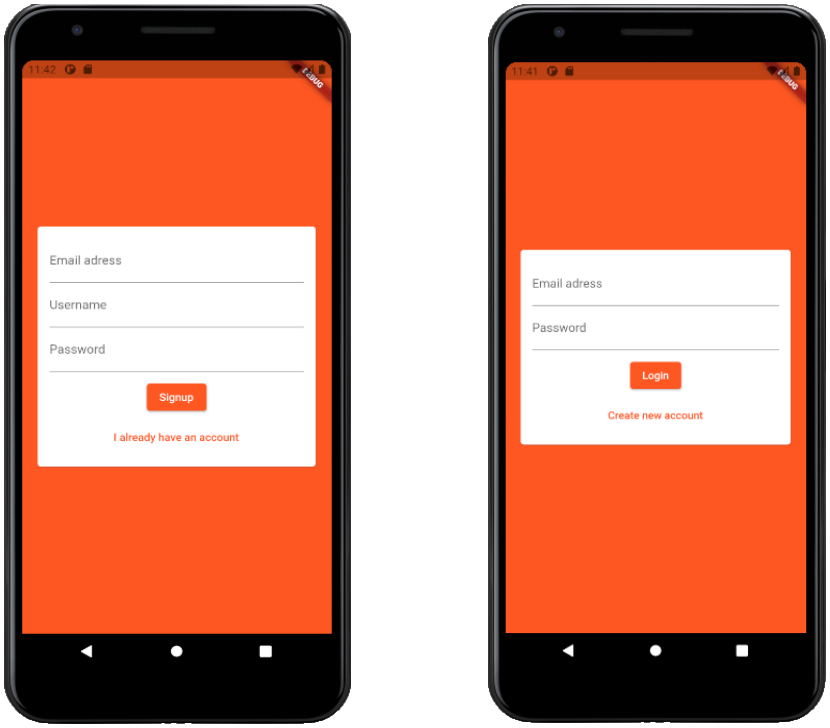
4.4 Firebase autentikacija

Sam proces autentikacije i kreiranja novih računa biti će u potpunosti obavljen od strane *Firebase Authentication* usluge. . Prvo je tu uslugu potrebno omogućiti u konzoli našeg Firebase projekta, te odabrati način autentikacije koji želimo. Kako bi se ona mogla koristiti, unutar *pubspec.yaml* datoteke projekta potrebno je dodat *firebase_auth* paket [10]. Implementacija i korištenje ovog paketa biti će detaljnije prikazana u praktičnom dijelu rada.

5. IZRADA APLIKACIJE

5.1 Implementacija autentikacije

Za prijavu u aplikaciju, odnosno stvaranje novog korisničkog računa koristit će se jednostavan *Form* widget. On omogućuje unos i pohranu korisničkih podataka unutar sebe korištenjem *TextFormField* widgeta, te validaciju istih. Za prijavu u aplikaciju od korisnika se traži da unese svoju e-mail adresu i zaporku, dok se za kreiranje novog korisničkog računa dodatno traži i odabiranje korisničkog imena. Pritiskom na tipku za prijavu/registraciju provjerava se valjanost korisničkih podataka, te ako je provjera uspješna stvoren je novi korisnički račun, donosno korisnik je prijavljen u aplikaciju.



Slika 5.1 Izgledi ekrana za registraciju i prijavu korisnika

Sam proces autentikacije i kreiranja novih računa biti će obavljen od strane *Firebase Authentication* usluge. Prvo je tu uslugu potrebno omogućiti u konzoli Firebase projekta, te odabrati način autentikacije koji želimo. U okviru ovog rada koristit će se jednostavna autentikacija sa e-mailom i lozinkom. Nadalje, kako bi se ova usluga mogla koristiti unutar aplikacije, unutar *pubspec.yaml* datoteke projekta potrebno je dodat *firebase_auth* paket, te ga je potrebno importirati na mjestu gdje ga se namjerava koristiti.

```
dependencies:
  flutter:
    sdk: flutter
  firebase_core : ^1.3.0
  firebase_auth: ^1.4.1
```

Slika 5.2 Dodavanje firebase_core i firebase_auth paketa

```
import 'package:firebase_auth/firebase_auth.dart';
```

Slika 5.3 Importiranje firebase_auth paketa

Funkcija koja se poziva prilikom pritiska na tipku za prijavu/registraciju, odnosno funkcija koja će koristiti Firebase uslugu za autentikaciju je *_submitAuthForm*. Ona je zadužena za validaciju podataka koje je korisnik unio u *Form* widget na ekranu za autentikaciju. Dio koda funkcije u kojem se vrši validacija prikazan je na slici ispod. *FirebaseAuth.instance* vraća instancu pristupne točke preko koje se može pristupiti uslugama autentikacije. Preko nje može se pristupiti funkcijama kao što su *signInWithEmailAndPassword* i *createUserWithEmailAndPassword* kojima se kao parametri predaju e-mail adresa i lozinka korisnika. One kao rezultat vraćaju *UserCredentials* objekt u kojemu će biti pohranjeni korisnički podaci ako je autentikacija uspješna, odnosno *null* ako je došlo do pogreške ili su uneseni neispravni podaci. Ovisno o njemu korisnik je ili vraćen na ekran za prijavu/registraciju ili uspješno prijavljen u aplikaciju i odveden na glavni izbornik aplikacije.

```
class _AuthScreenState extends State<AuthScreen> {
  final _auth = FirebaseAuth.instance;
  var _isLoading = false;

  void _submitAuthForm(
    String email,
    String password,
    String username,
    bool isLogin,
    BuildContext ctx,
  ) async {
    UserCredential userCredential;

    try {
      setState(() {
        _isLoading = true;
      });
      if (isLogin) {
        userCredential = await _auth.signInWithEmailAndPassword(
            email: email, password: password);
      } else {
        userCredential = await _auth.createUserWithEmailAndPassword(
            email: email, password: password);
```

Slika 5.4 Primjena Firebase usluge za autentikaciju

5.2 Definiranje izgleda glavnog izbornika

Nakon uspješne prijave korisnik je odveden na glavni izbornik aplikacije. On će se sastoji od dva ekrana između kojih će se kretati pomoću trake za navigaciju na donjem dijelu ekrana. Na prvom ekranu nalaziti će se lista proizvoda. Korisnik će moći pregledavati sve proizvode, filtrirati ih po proizvoljnim kriterijima, odabrati određeni proizvod kako bi vidio detalje o njemu, te odabrane proizvode staviti u košaricu. Drugi ekran biti će košarica u koju će korisnik stavljati one proizvode koje namjerava kupiti. Proizvode koje je stavio u košaricu korisnik će moći naručiti , odnosno ukloniti iz košarice ako se predomisli. Za stvaranje glavnog izbornika prvo se kreira nova datoteka pod imenom *tabs_screen.dart*. U njoj se definira widget *TabsScreen*. On će biti zadužen za kretanje između ekrana s listom proizvoda i ekrana košarice. Iste je, naravno, prvo potrebno definirati. Za ekran sa listom proizvoda stvaramo *products_overview_screen.dart* datoteku i njoj definiramo *ProductsOverviewScreen* widget. Za ekran s košaricom stvaramo *cart_screen.dart* datoteku i u njoj definiramo *CartScreen* widget. Za sada se u njih stavlja samo jedan *Scaffold* widget [11]. *Scaffold* je klasa iz Flutterovog *material.dart* paketa i predstavlja jedan od osnovnih elemenata od kojih se grade Flutter aplikacije. Detaljan rad s njim prikazan je i objašnjen u [12]. On zauzima cijelu površinu ekran i unutar njega se ugnježđuju drugi widgeti, od kojih se prvi predaje kao *body* parametar. Svi ostali widgeti koje će se prikazati na ekranu *djeca* su body widgeta. Međutim *Scaffold* nudi još niz korisnih mogućnosti, neke od kojih su postavljanje donje navigacijske trake (engl. *Bottom navigation bar*), naslovne trake (engl. *App bar*), plutajući akcijski gumb (engl. *Floating action button*) i ladica (engl. *Drawer*). Sve navedene mogućnosti biti će korištene u ovom radu.

Prva od njih biti će donja navigacijska traka. *Scaffold* kao jedan od mogućih parametara može primiti jedan *BottomNavigationBar* widget. Unutar njega definira se broj ekrana između kojih će se moći kretati, te koji su to ekrani. Ekrane između kojih će se kretati stavljaju se u listu koja se postavi kao *body* parametar *Scaffold* widgeta. Kao rezultat, na donjem dijelu ekrana dobijemo navigacijsku traku. Sama traka i elementi na traci mogu se proizvoljno urediti. Pojedini element trake može biti bilo koji drugi widget, najčešće tekst ili ikona. Drugi parametar koji se može predati *Scaffold-u* je *AppBarr* widget. Njime se može definirati naslov koji se prikazuje na naslovnoj traci, te se u nju mogu dodati još razne opcije kao što su padajući izbornik ili gumbovi s različitim funkcijama. Za sada će se koristiti samo za prikaz imena aplikacije. Na slici 5.5 može se vidjeti kod glavnog izbornika, te kako on u konačnici izgleda. Može se primijetiti kako je unutar *Scaffolda* dodan i jedan Drawer widget unutar kojega će se kasnije dodati opcije aplikacije.

```dart
class _TabsScreenState extends State<TabsScreen> {
  final List<Widget> _pages = [
    ProductsOverviewScreen(),
    CartScreen(),
  ];
  int _selectedPageIndex = 0;
  void _selectPage(int index) {
    setState(() {
      _selectedPageIndex = index;
    });
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(
          'Techstreme',
          style: TextStyle(fontWeight: FontWeight.bold),
        ), // Text
      ), // AppBar
      drawer: MainDrawer(),
      body: _pages[_selectedPageIndex],
      bottomNavigationBar: BottomNavigationBar(
        onTap: _selectPage,
        backgroundColor: Colors.black,
        unselectedItemColor: Colors.white,
        selectedItemColor: Colors.deepOrange,
        currentIndex: _selectedPageIndex,
        items: [
          BottomNavigationBarItem(
            icon: Icon(Icons.laptop_sharp),
            label: 'Products',
          ), // BottomNavigationBarItem
          BottomNavigationBarItem(
            icon: Icon(Icons.shopping_cart_sharp),
            label: 'Cart',
          ), // BottomNavigationBarItem
        ],
      ), // BottomNavigationBar
    ); // Scaffold
  }
```

Slika 5.5 Izvorni kod glavnog izbornika i njegov grafički prikaz

5.3 Definiranje modela proizvoda

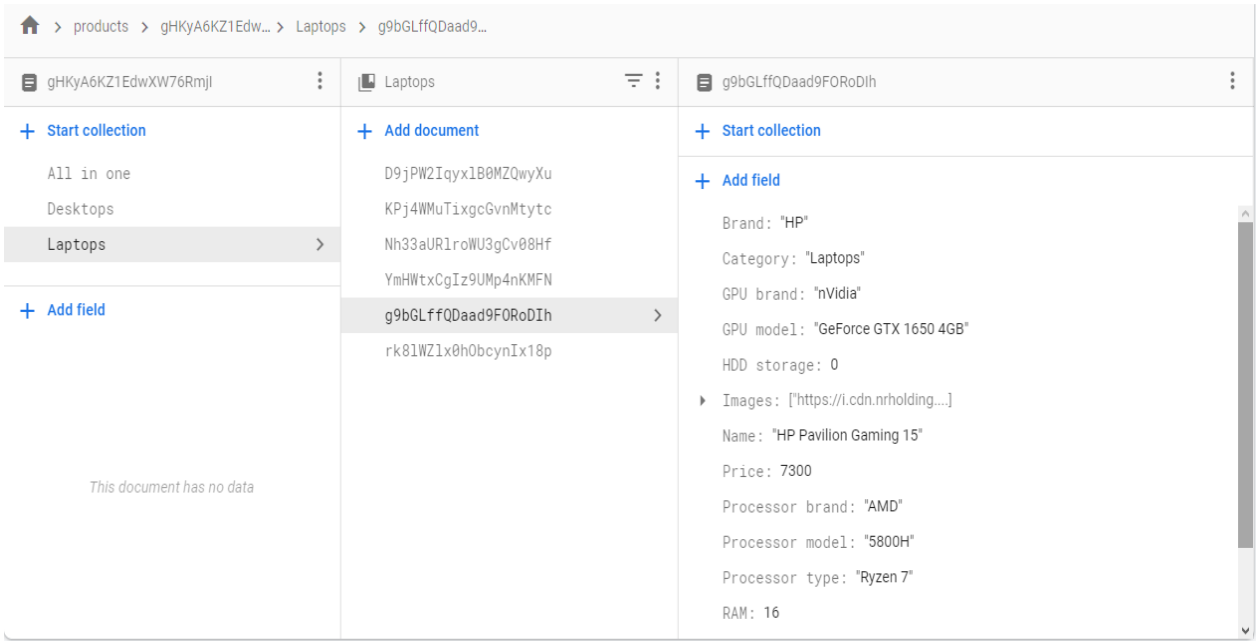
Definirana je klasa *Product* koja sadrži sljedeće atribute: id proizvoda, kategorija proizvoda, ime, marka proizvođača, cijena, slike, marka procesora, vrsta procesora, model procesora, marka grafičke kartice, model grafičke kartice, količina RAM memorije, količina SSD memorije i količina HDD memorije.

```dart
class Product {
  final String category;
  final String id;
  final String name;
  final String brand;
  final int price;
  final List<dynamic> images;
  final String processorBrand;
  final String processorModel;
  final String processorType;
  final String gpuBrand;
  final String gpuModel;
  final int ram;
  final int ssdStorage;
  final int hddStorage;
```

Slika 5.6 Atributi klase Product

5.4 Popunjavanje baze podataka

Sada kada je definiran model proizvoda i svi njegovi atributi, može se započeti sa popunjavanjem baze podataka. Kao što je već objašnjeno u teorijskom dijelu rada, Firebase baza podataka organizirana je kao kolekcija dokumenata, gdje svaki dokument može sadržavati razne tipove podataka, te unutar sebe može stvarati nove kolekcije dokumenata. U slučaju ove aplikacije stvorena je kolekcija *products* koja sadrži jedan dokument. Unutar tog dokumenta stvara se nova kolekciju koja će se zvati *Laptops.* Konačno, ona se popunjava dokumentima koji predstavljaju laptope, te se oni popunjavaju vrijednostima koje opisuju pojedini laptop. Prilikom popunjavanja moramo paziti da se tipovi podataka u dokumentu podudaraju s tipovima podataka u našem modelu proizvoda, kako ne bi došlo do pogrešaka prilikom dohvaćanja podataka iz baze. Na slici ispod prikazano je kako izgleda jedan unos u bazi podataka.



Slika 5.7 Primjer jednog unosa u bazi podataka

5.5 Dohvaćanje podataka iz baze

Za dohvaćanje podataka iz baze potrebno je, kao što je i prije navedeno, dodati *cloud_firestore* paket unutar *pubspec.yaml* datoteke aplikacije. Isti paket potrebno je importirati na mjestu u kodu gdje ga želimo koristiti. Nakon što je ovo napravljeno potrebno je stvoriti *Stream* koji će služiti kao izvor podataka. Način na koji se to radi prikazan je na slici 5.8. Pozivanjem gettera *instance* na klasi *FirebaseFirestore* pristupamo bazi podataka. Metodi *collection* kao

parametar predajemo putanju do kolekcije kojoj želimo pristupiti. Metoda *snapshots* vraća nam *Stream* preko kojeg možemo pristupiti svim dokumentima u toj kolekciji. Korištenjem metode *where* moguće je postavljati upite kako bi dobili samo točno određene dokumente, što će biti detaljnije prikazano kada se budu dodavali filteri.

```
Stream dataSource = FirebaseFirestore.instance
    .collection('products/gHKyA6KZ1EdwXW76RmjI/Laptops')
    .snapshots();
```

Slika 5.8 Dohvaćanje podataka iz baze

5.6 Prikaz proizvoda

Kako bi se prikazali proizvodi koji su dohvaćeni iz baze podataka, *Scaffold* widgetu ekrana sa svim proizvodima, odnosno *ProductsOverviewScreen* widgetu koji je dosad bio prazan, kao *body* postavljamo *StreamBuilder* [13]. On kao parametar prima dataSource *Stream* koji je definiran u prošlom koraku, te *builder* funkciju koja određuje što se radi sa dobivenim podacima. Podaci iz *dataSource Stream*-a pohranjuju su *streamSnapshot* varijabli. Prije samog korištenja podacima prvo se provjerava jesu li uspješno preuzeti ili je došlo do nekakve pogreške. Ako je sve u redu iz *streamSnapshota* izvlači se lista dokumenata te se ona pohranjuje u *documents* varijablu. U njoj se sada nalazi lista u kojoj svaki element predstavlja jedan laptop zajedno s njegovim specifikacijama.Kao povratnu vrijednost *StreamBuildera* vraćamo *ListView.builder* [14]. On služi za izgradnju *ListView*-a s fiksnim brojem elemenata. Ti elementi mogu biti bilo koji proizvoljni widget. Kao parametre prima broj elemenata kojih treba izgraditi, te *itemBuilder* funkciju koja definira kako će se elementi graditi. Broj elemenata lagano se dobije pomoću *documents.length* metode. Svaki element koji se gradi ima svoj indeks što je vrlo korisno jer i svaki dokument u listi dokumenata također ima svoj indeks, preko kojeg se pristupa njemu i njegovim podacima. Pojedinom podatku unutar dokumenta pristupa se tako što se u uglatim zagradama, pod jednostruke navodnike, napiše ime polja. Kao povratnu vrijednost *ListView.builder* vraća korisnički definiran widget pod nazivom *ProductCard.* On kao parametre prima sve informacije o proizvodu, te nam na ekranu generira karticu sa slikom, nazivom i cijenom proizvoda. Rezultat se može vidjeti na slici 5.9. Razlog tome što mu se predaju svi podaci o proizvodu je taj što će korisnik, odabirom jedne od kartica, biti odveden na ekran sa detaljima o proizvodu. Tom ekranu su, naravno, te informacije potrebne kako bi ih mogao prikazati. Izvorni kod *ProductCard* widgeta može se pronaći u dokumentaciji rada.

```
return Scaffold(
  backgroundColor: Colors.grey[900],
  body: StreamBuilder(
    stream: dataSource,
    builder: (ctx, streamSnapshot) {
      if (streamSnapshot.connectionState == ConnectionState.waiting) {
        return Center(
          child: CircularProgressIndicator(),
        ); // Center
      } else if (streamSnapshot.hasError) {
        return Center(
          child: Text('There was an error!'),
        ); // Center
      }
      final documents = streamSnapshot.data.docs;
      return ListView.builder(
        itemCount: documents.length,
        itemBuilder: (ctx, index) {

          return ProductCard(
            id: documents[index]['id'],
            name: documents[index]['Name'],
            price: documents[index]['Price'],
            brand: documents[index]['Brand'],
            images: documents[index]['Images'],
            gpuBrand: documents[index]['GPU brand'],
            gpuModel: documents[index]['GPU model'],
            processorBrand: documents[index]['Processor brand'],
            processorModel: documents[index]['Processor model'],
            processorType: documents[index]['Processor type'],
            ram: documents[index]['RAM'],
            ssdStorage: documents[index]['SSD storage'],
            hddStorage: documents[index]['HDD storage'],
          ); // ProductCard
```

Slika 5.9 Generiranje liste proizvoda i njen prikaz

5.7 Ekran s detaljima proizvoda

Kada je korisnik na ekranu s listom svih proizvoda odabrao jedan proizvod, odveden je na novi ekran na kojem se nalaze slike proizvoda i njegove specifikacije. Za kretanje s jednog na drugi ekran zadužena je funkcija *selectProduct* koja je predana kao *onTap* parametar *ProductCard* widgeta. Ona se poziva svaki put kada korisnik odabere neki proizvod. Njoj se predaju svi parametri koji su prethodno predani *ProductCard*-u. Nadalje unutar funkcije poziva se *Navigator.push* metoda. Ona kao parametar prima *MaterialPageRoute* widget. Njemu se predaje konstruktor ekrana na koji želimo otići, popunjen sa parametrima koje zahtjeva. Cijeli kod funkcije *selectProduct* može se vidjeti na slici 5.10. Kao rezultat *Navigator.push* metode korisnik je odveden na ekran sa detaljima o proizvodu, te je taj ekran stavljen na vrh navigacijskog stoga. U ovom slučaju to je *ProductsDetailScreen* koji je korisnički definiran i dostupan je u dokumentaciji rada. Korištenjem *Navigator.push* metode Flutter nam automatski generira gumb za povratak u naslovnoj traci novog ekrana. Pritiskom na njega poziva se *Navigator.pop* metoda koja sa vrha

navigacijskog stoga uklanja ekran na kojem se trenutno nalazimo i vraća nas na prethodni. Više o Navigator klasi može se pročitati na [15] a o tome kako se ona konkretno primjenjuje u [16]. Konačni rezultat, odnosno izgled ekrana s detaljima proizvoda može se vidjeti na slici 5.11. Može se primijetiti kao na ekranu postoje i dva plutajuća akcijska gumba za dodavanje proizvoda u košaricu i komentiranje, no njihova funkcionalnost implementirati će se kasnije.

```
void selectProduct(BuildContext context) {
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) => ProductDetailScreen(
        id: id,
        name: name,
        brand: brand,
        price: price,
        processorBrand: processorBrand,
        processorModel: processorModel,
        processorType: processorType,
        gpuBrand: gpuBrand,
        gpuModel: gpuModel,
        ram: ram,
        ssdStorage: ssdStorage,
        hddStorage: hddStorage,
        images: images,
      ), // ProductDetailScreen
    ), // MaterialPageRoute
  );
}
```

Slika 5.10 Funkcija za odabiranje proizvoda

Slika 5.11 Ekran s detaljima proizvoda

5.8 Filtriranje proizvoda

Filtriranje proizvoda biti će realizirano na način da će korisnik na ekranu s opcijama za filtriranje odabrati opcije koje želi, na primjer vrstu procesora i raspon cijena, te će se prilikom dohvaćanja proizvoda iz baze dohvatiti samo oni koji zadovoljavaju zadane kriterije. Kako se dohvaćanje proizvoda odvija na drugom mjestu u aplikaciji, to jest u *widgetu* koji je zadužen za dohvaćanje i prikaz proizvoda, odabrane opcije filtriranja moraju mu biti na raspolaganju. Za realizaciju navedenog koristit će se *provider* paket [17]. Kao i svaki paket dosad, isti je potrebno dodati u *pubspec.yaml* datoteku aplikacije kako bi se mogao koristiti. *Provider* paket omogućava

stvaranje *providera,* posebnih klasa unutar kojih se mogu pohranjivati podaci te definirati metode za dohvaćanje i rad s istima. Funkcionira na način da proizvoljan *widget* omotamo s *ChangeNotifierProvider widgetom* koji kao parametar prima jedan *provider.* Taj *provider* će tada biti dostupan svim *widgetima* koji su djeca omotanog *widgeta.* Kako bi *provider* s filterima bi bio dostupan bilo gdje u aplikaciji omotati ćemo korijenski *widget* aplikacije sa *ChangeNotifierProvider widgetom* te mu kao parametar predati prethodno definirani *Filters provider,* kao što je to prikazano na slici 5.12. Važno je napomenuti da to što je *provider* sada dostupan svim widgetima u aplikacija ne znači da ga svi i koriste. Kako bi se neki *widget* koristio *providerom* i bio obavješten o promjenama unutar istog, unutar samog *widgeta* se to mora posebno naglasiti linijom koda koja je prikazana na slici 5.13.

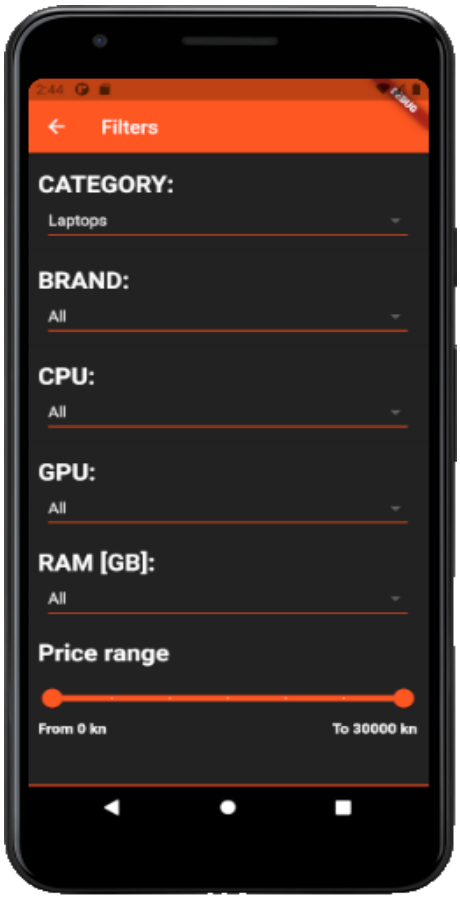
```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return
        ChangeNotifierProvider(
          create: (context) => Filters(),
      child: MaterialApp(
        title: 'MyShop',
```

Slika 5.12 Uspostavljanje providera

```
Filters filtersData = Provider.of<Filters>(context);
```

Slika 5.13 Spajanje widgeta s providerom

U okviru ove aplikacije to će biti unutar dva *widgeta,* od kojih je prvi ekran s opcijama za filtriranje, a drugi ekran s prikazom proizvoda. Po zadanom na ekranu s prikazom proizvoda prikazani su svi proizvodi jedne kategorije, no kada korisnik na ekranu s filterima odabere kategoriju proizvoda i određene opcije filtriranja, one će biti pohranjene u *Filters provideru*. Nakon što su one pohranjene poziva se funkcija *notifyListeners()* koja obavještava ekran s proizvodima o promjeni opcija filtriranja te će on na temelju istih iz baze podataka dohvatiti one proizvode koji zadovoljavaju zadane kriterije filtriranja i prikazati ih na ekranu. Izvorni kod ekrana za filtriranje i *providera* s filterima dostupan je u dokumentaciji rada, a na slici 5.14 može se vidjeti konačan izgled ekrana za filtriranje.



Slika 5.14 Ekran s filterima

5.9 Dodavanje proizvoda u košaricu i narudžba proizvoda

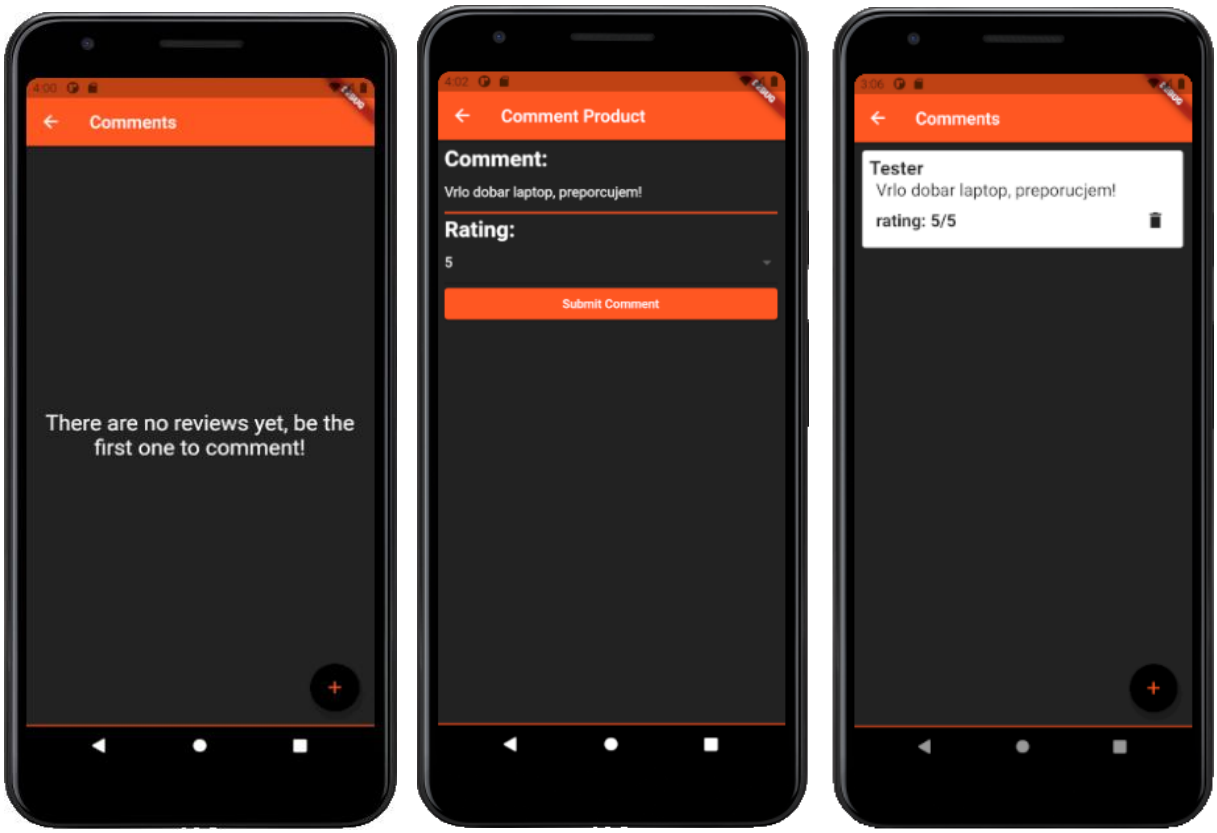
Dodavanje proizvoda u košaricu realizirati će se također pomoću *providera.* Na ekranu s detaljima o proizvodu pritiskom na plutajuća akcijska gumba za dodavanje proizvoda u košaricu, odabrani proizvodu biti će pohranjeni u prethodno definirani *Cart provider* koji predstavlja košaricu. Ekran koji prikazuje proizvode u košaricu obavješten je o promjenama unutar *Cart providera,* dohvaća proizvode iz njega te ih prikazuje na korisničkom sučelju. Na ekranu košarice, osim proizvoda prikazana je i ukupna cijena svih proizvoda u košarici. Korisnik ima mogućnost uklanjanja proizvoda iz košarice ili ako je zadovoljan svojim odabirom, potvrditi narudžbu. Potvrdom narudžbe ista se pohranjuje u bazi podataka, a košarica se prazni. Korisnik sve svoje narudžbe može vidjeti na ekranu s narudžbama. Izvorni kod *Cart providera,* ekrana s košaricom i ekrana sa svim narudžbama dostupan je u dokumentaciji, a na slikama ispod prikazani su ekran s košaricom i ekran sa svim narudžbama.

Slika 5.15 Ekran s košaricom

Slika 5.16 Ekran s narudžbama

5.10 Komentiranje i ocjenjivanje proizvoda

Posljednja funkcionalnost koja će se dodati u aplikaciju je mogućnost komentiranja i ocjenjivanja proizvoda. Na ekranu s detaljima o proizvodu pritiskom na plutajuća akcijska gumba sa znakom za komentiranje korisnik je prebačen na ekran sa svim komentarima i ocjenama tog proizvoda. Na njemu se nalazi još jedan plutajući akcijski gumb sa znakom plus. Pritiskom njega korisnik je odveden na novi ekran na kojem može unijeti svoj komentar i dati ocjenu proizvodu od jedan do pet. Svaki korisnik ima pravo ostavit jedan komentar s ocjenom po proizvodu. Pritiskom na gumb *Submit Comment* komentar i ocjena pohranjeni su u bazu podataka i korisnik je vraćen na prethodni ekran gdje sada može vidjeti svoj komentar i ocjenu. Ako korisnik ocjeni isti proizvod dva puta, nova ocjena zamijeniti će staru. Korisnik, naravno, može i obrisati komentar i ocjenu koju je dao određenom proizvodu. Komentari i ocjene pohranjeni su bazi podataka na način da svaki proizvod unutar sebe ima mapu sa svim komentarima i ocjenama koji su vezani za njega. Na slikama ispod može se vidjeti cijeli postupak komentiranje i ocjenjivanja proizvoda. Izvorni kod ekrana sa pregledom svih komentara i ocjena te ekrana za pisanje komentara i ocjenjivanje proizvoda dostupni su u dokumentaciji.



Slika 5.17 Komentiranje i ocjenjivanje proizvoda

6. ZAKLJUČAK

Završnim radom proučile su se dvije tehnologije koje su se koristile za njegovu izradu, Flutter i Firebase. Objašnjeno je kako Flutter funkcionira, kako se pomoću njega izrađuju aplikacije i koji su njeni glavni dijelovi. Također je objašnjeno kako Firebase funkcionira i kako se pomoću njega uspostavlja baza podataka i autentikacija korisnika. Korištenjem ovih tehnologija razvijena je aplikacija za internet trgovinu. Flutter se pokazao kao dobar izbor tehnologije za razvoj aplikacije zbog svoje jednostavnosti i intuitivnog načina korištenja, dok se Firebase pokazao kao dobra alternativa standardnim SQL bazama podataka. Sama aplikacija ima izgled i većinu funkcionalnosti pravih internet trgovina, no mogla bi se proširiti i nadogradit u nekim aspektima. Moguća proširenja bila bi dodavanje dodatnih kategorija proizvoda, naprednijih opcija filtriranja te pravih transakcija.

7. LITERATURA

[1] U. Khan, „The Pros and Cons of Native Apps“ [Online]
Dostupno na: https://clutch.co/app-developers/resources/pros-cons-native-apps
[18. rujna 2021.]

[2] P. Borrelli „ Comparing Mobile Develpoment Platforms in 2021“ [Online]
Dostupno na: https://strapi.io/blog/comparing-mobile-development-platforms-in-2021
[18.rujna 2021.]

[3] S. Liu „Cross-platform mobile frameworks used by developers worldwide 2019- 2021“
[Online] Dostupno na:
https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/ [18.rujna 2021.]

[4] Flutter, „Flutter architectual overview“ [Online].
Dostupno na: https://flutter.dev/docs/resources/architectural-overview [5. rujna 2021.]

[5] Flutter, „Instal“ [Online].
Dostupno na: https://flutter.dev/docs/get-started/install [5. rujna 2021.]

[6] Visual Studio Code, „Code editing. Redefined.“ [Online].
Dostupno na: https://code.visualstudio.com/ [5. rujna 2021.]

[7] Firebase, „Get started“ [Online].
Dostupno na: https://firebase.google.com/ [5. rujna 2021.]

[8] pub.dev, „Firebase Core for Flutter“ [Online].
Dostupno na: https://pub.dev/packages/firebase_core [5. rujna 2021.]

[9] pub.dev „Cloud Firestore Plugin for Flutter“ [Online].
Dostupno na: https://pub.dev/packages/cloud_firestore [5. rujna 2021.]

[10] pub.dev „Firebase Auth for Flutter“ [Online].
Dostupno na: https://pub.dev/packages/firebase_auth [5. rujna 2021.]

[11] Flutter, „Scaffold class“ [Online].
Dostupno na: https://api.flutter.dev/flutter/material/Scaffold-class.html [5. rujna 2021.]

[12] R. Payne, Beginning App Development with Flutter: Create Cross-Platform Mobile Apps, Apress, 2019.

[13] Flutter, „StreamBuilder class“ [Online].
Dostupno na: https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html [5. rujna 2021.]

[14] Flutter, „ListView class“ [Online].

Dostupno na: https://api.flutter.dev/flutter/widgets/ListView-class.html [5. rujna 2021.]

[15] Flutter, „Navigator class“ [Online].
Dostupno na: https://api.flutter.dev/flutter/widgets/Navigator-class.html [5. rujna 2021.]

[16] M. Katz, K. D. Moore, V. Ngo, Flutter Apprentice (First Edition): Learn to Build Cross-Platform Apps, Razeware LLC, 2021.

[17] pub.dev, „provider 6.0.0“ [Online].
Dostupno na: https://pub.dev/packages/provider [5. rujna 2021.]

SAŽETAK

Završnim radom prikazan je razvoj mobilne aplikacije za internet trgovinu korištenjem Flutter i Firebase tehnologija. Proučena je arhitektura Fluttera te je ukratko objašnjen način na koji se izrađuju Flutter aplikacije i koji su njeni glavni dijelovi. Objašnjeno je kako se pomoću Firebasea uspostavlja baza podataka, kako je ona strukturirana te kako se popunjava. Prikazano je i kako se pomoću istog implementira autentikacija korisnika. Razvijena je Flutter aplikacija i detaljno je objašnjeno kako ju povezati s Firebase bazom podataka i uslugom autentikacije korisnika. Demonstriran je način na koji se razmjenjuju podaci između aplikacije i baze podataka te način na koji to rade dijelovi aplikacije međusobno. Prikazan je razvoj ključnih dijelova aplikacije kao što su prikaz proizvoda, filtriranje proizvoda, dodavanje proizvoda u košaricu, narudžba proizvoda, komentiranje i ocjenjivanje proizvoda te kako su svi oni povezani u funkcionalnu cjelinu.

Ključne riječi: Firebase, Flutter, Internet trgovina, Mobilna aplikacija

ABSTRACT

The final paper shows the development of a webshop application using the Flutter and Firebase technologies. It studies te architecture of Flutter, the way in which Flutter applications are made and briefly explains what its main parts are. It explains how Firebase is used to establish a database, how the database is structured and how to populate it. It is also shown how Firebase is used to implement user authentication. A Flutter application was developed and it is explained in detail how to connect it to the Firebase database and authentication service. It demonstrates the way in which data is exchanged between the application and the database and how parts of the application exchange it between each other. Finally, it shows the development of key parts of the application, such as product display, product filtering, adding products to the cart, ordering products, commenting and rating products and shows how they are all connected into a functional whole.

Keywords: Firebase, Flutter, Mobile application, Webshop

ŽIVOTOPIS

Robert Pepić, rođen je 6. prosinca 1997. u Osijeku. Pohađao je i završio III. Gimnaziju u Osijeku. Godine 2016. upisuje preddiplomski sveučilišni studij elektrotehnike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek na Sveučilištu Josipa Jurja Strossmayera u Osijeku. Godine 2017. na istom fakultetu prebacuje se na sveučilišni studij računarstva.

Robert Pepić