Višeplatformski razvoj mobilnih aplikacija sa sinkronizacijom u stvarnom vremenu

Stjepanek, Marko

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:058102

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-21**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Sveučilišni studij

VIŠEPLATFORMSKI RAZVOJ MOBILNIH APLIKACIJA SA SINKRONIZACIJOM U STVARNOM VREMENU

Diplomski rad

Marko Stjepanek

Osijek, 2021.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 19.02.2021.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	Marko Stjepanek
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1009R, 09.10.2018.
OIB studenta:	06165983415
Mentor:	Prof.dr.sc. Goran Martinović
Sumentor:	
Sumentor iz tvrtke:	Vlatko Vlahek
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Alfonzo Baumgartner
Član Povjerenstva 1:	Prof.dr.sc. Goran Martinović
Član Povjerenstva 2:	Doc.dr.sc. Mirko Köhler
Naslov diplomskog rada:	Višeplatformski razvoj mobilnih aplikacija sa sinkronizacijom u stvarnom vremenu
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U diplomskom radu potrebno je analizirati i opisati izazove višeplatformskog razvoja mobilnih aplikacija za iOS i Android platforme s posebnim osvrtom na tehnologiju React-Native. Također, treba analizirati i opisati načine pretvorbe programskog koda u druge programske jezike, mogućnosti i načine korištenja mikrousluga u oblaku računala (Google Firebase), rad s korisničkim računima, korištenje socketa (Web socket listener na Firebase u oblaku računala), kao i postupke lokalnog skladištenja podataka pri nedostatku mrežne veze. Navedena načela, metodologiju, okoline, alate i tehnologije potrebno je primijeniti u praktičnom dijelu rada za razvoj mobilne aplikacije za pretragu i prijavu na događaje uz zahtjev za
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	19.02.2021.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 08.03.2021.

Ime i prezime studenta:	Marko Stjepanek
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1009R, 09.10.2018.
Turnitin podudaranje [%]:	9

Ovom izjavom izjavljujem da je rad pod nazivom**: Višeplatformski razvoj mobilnih aplikacija sa sinkronizacijom u stvarnom vremenu**

izrađen pod vodstvom mentora Prof.dr.sc. Goran Martinović

i sumentora

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

Potpis studenta:

SADRŽAJ

1. UVOD 1

2. PREGLED STANJA U PODRUČJU VIŠEPLATFORMSKOG RAZVOJA MOBILNIH APLIKACIJA S MOGUČNOSTIMA SINKRONIZACIJE U STVARNOM VREMENU 2

2.1. Višeplatformski razvoj mobilnih aplikacija 2

2.1.1. Prednosti i nedostatci višeplatformskih mobilnih aplikacija 2

2.1.2. Arhitektura programskih okvira višeplatformskih mobilnih aplikacija 3

2.1.3. Transpiling koda 5

2.1.4. Rad s mikrouslugama 8

2.1.5. Websocket 10

2.2. Sinkronizacija u stvarnom vremenu 13

2.3. Prikaz sličnih postojećih rješenja aplikacija i sustava 14

3. IDEJNO RJEŠENJE I ARHITEKTURA MOBILNE APLIKACIJE TICKETNATOR 17

3.1. Funkcionalni i nefunkcionalni zahtjevi za rad mobilne aplikacije 17

3.2. Idejno rješenje i model mobilne aplikacije 18

3.3. Arhitektura mobilne aplikacije i osvrt na višeplatformski razvoj 21

4. PROGRAMSKO RJEŠENJE TICKETNATORA 24

4.1. Programski jezici i tehnologije 24

4.1.1. Operacijski sustav Android 24

4.1.2. Operacijski sustav iOS 25

4.1.3. Programski okvir React Native 27

4.1.4. Uređivač koda Visual Studio Code 28

4.1.5. Platforma Firebase i skladištenje podataka pri nedostatku mrežne veze 30

4.1.6. Sustav za upravljanje programskim kodom Git, model grananja GitFlow i alat za dijeljenje programskog koda Bitbucket 32

4.1.7. Spremnik stanja Redux 33

4.1.8. Sustav kontinuirane mobilne integracije i isporuke Bitrise 34
4.1.9. Alat za uređivanje i izradu prototipova za digitalne projekte Figma 35
4.2. Prikaz programskog rješenja na strani poslužitelja 37
4.2.1. Kreiranje korisničkog računa 38
4.2.2. Povezivanje korisničkog računa s Google i Facebook računom 42
4.2.3. Izmjena lozinke korisničkog računa 44
4.2.4. Dohvaćanje korisničkog profila 44
4.2.5. Dohvaćanje svih događaja iz baze podataka 46
4.2.6. Dohvaćanje vrsta ulaznica događaja za kupovinu 46
4.2.7. Kupovina ulaznice za određeni događaj 47
4.2.8. Dohvaćanje detalja kupljene ulaznice 49
4.2.9. Odjava korisničkog računa iz aplikacije 50
4.3. Prikaz programskog rješenja na strani klijenta 51
4.3.1. Spremanje stanja pomoću Reduxa 51
4.3.2. Navigacija aplikacije 55
4.3.3. Prikaz svih događaja 60
4.3.4. Filtriranje događaja 62
4.3.5. QR kod 64
4.3.6. Prikaz poruke kod pogreške 65
4.3.7. Resursi 65
4.3.8. Globalni stilovi 66
5. PRIKAZ NAČINA KORIŠTENJA I ISPITIVANJE APLIKACIJE 69
5.1. Način korištenja aplikacije 69
5.2. Ispitivanje rada aplikacije 73
5.2.1. Korisnički slučaj 1. 73
5.2.2. Korisnički slučaj 2. 80

5.3. Rezultati ispitivanja rada mobilne aplikacije Ticketnator 87

6. ZAKLJUČAK 89

LITERATURA 90

SAŽETAK 94

ABSTRACT 95

ŽIVOTOPIS 96

PRILOZI (NA CD-U): 97

1. UVOD

Mobilna komunikacija ušla je u ljudske živote tako da se više ne može zamisliti svakodnevni život bez pametnog telefona. Velikom primjenom pametnih telefona dolazi se do razvoja mobilne industrije te time i veće potražnje mobilnih aplikacija. Kako bi se obuhvatilo što veće tržište dolazi do razvoja višeplatformskih mobilnih aplikacija. Iako je razvoj višeplatformskih mobilnih aplikacija napredovao i dalje ima probleme s kojima se programeri i korisnici aplikacija prilikom korištenja suočavaju. Jedan od njih je sinkronizacija u stvarnom vremenu za neprekidno dijeljenje podataka između baze podataka i aplikacije.

Cilj ovog diplomskog rada je analizirati i opisati izazove višeplatformskog razvoja mobilnih aplikacija za iOS i Android platforme s posebnim osvrtom na tehnologiju React Native. Također, potrebno je analizirati i opisati načine pretvorbe programskog koda u druge programske jezike, mogućnosti i načine korištenja mikrousluga u oblaku računala, rad s korisničkim računima, korištenje socketa, kao i postupke lokalnog skladištenja podataka pri nedostatku mrežne veze. U praktičnom dijelu rada je potrebno realizirati navedena načela, metodologiju, okoline, alate i tehnologije mobilnom aplikacijom za pretragu i prijavu na događaje uz zahtjev za sinkronizacijom u stvarnom vremenu. Programsko rješenje izrađeno u React Native-u treba ispitati i analizirati za prikladan skup događaja i načina korištenja, te napraviti usporedbu i analizu višeplatformskih i aplikacija programiranih u nativnim okruženjima.

Rad je strukturiran na sljedeći način. U drugom poglavlju nalazi se pregled stanja u području višeplatformskog razvoja, sinkronizacija u stvarnom vremenu i analiza već postojećih rješenja za zadatak rada. U trećem poglavlju navedeni su zahtjevi mobilnih aplikacija i prema zahtjevima izrađena arhitektura, idejno rješenje i modeli mobilne aplikacije Ticketnator. U četvrtom poglavlju opisani su programski jezici, tehnologije i programsko rješenje za mobilnu aplikaciju Ticketnator. U petom poglavlju nalaze se upute, načini za rad i analiza mobilne aplikacije Ticketnator.

2. PREGLED STANJA U PODRUČJU VIŠEPLATFORMSKOG RAZVOJA MOBILNIH APLIKACIJA S MOGUČNOSTIMA SINKRONIZACIJE U STVARNOM VREMENU

U ovom poglavlju navedene su i opisane novosti i nove tehnologije o temama koje se obrađuju u ovom radu te prikaz sličnih postojećih rješenja aplikacija i sustava.

2.1. Višeplatformski razvoj mobilnih aplikacija

Prenosivost programa na različite sustave cilj je mnogim programskim jezicima pa tako mobilne aplikacije nisu prve koje koriste višeplatformski razvoj. Višeplatformski razvoj mobilnih aplikacija omogućuje programerima kreiranje mobilne aplikacije koja je kompatibilna s više operacijskih sustava, u ovom slučaju su to iOS i Android.

2.1.1. Prednosti i nedostatci višeplatformskih mobilnih aplikacija

Prilikom analize, često se uspoređuje višeplatformski razvoj s nativnim razvojem zbog njegovih prednosti i nedostataka kao što se vidi u tablici 2.1.

Tab. 2.1. *Usporedba višeplatformskog i nativnog razvoja* [1]

Parametri	**Višeplatformski razvoj**	**Nativni razvoj**
Cijena	Relativno mala cijena razvoja	Velika cijena razvoja
Korištenje koda	Isti kod se može koristit na različitim platformama	Radi na jednoj platformi
Korištenje uređaja	Nije sigurno hoće li radit svi API-ji	Svi API-ji koje ima uređaj se mogu koristit
Korisničko sučelje	Limitirano ovisno o uređaju	Konzistentno s komponentama uređaja
Performanse	Sporija i veća potrošnja resursa	Brže i bolje performanse
Sigurnost	Manja sigurnost	Veća sigurnost

Iz tablice se dolazi do zaključka kako će nativnim razvojem aplikacija biti brža i sigurnija, dok će se korištenjem višeplatformskog razvoja brže i jeftinije napraviti razvoj aplikacije. Još jedan od nedostataka je što Apple koristi Xcode kao integrirano razvojno okruženje, dok Android koristi Android Studio. Tu je prvi problem jer Xcode nije kompatibilan s Windowsima već se može koristit samo na macOS, operacijskom sustavu za Mac. Kako bi se koristio React Native za razvoj mobilnih aplikacija za Android i iOS, potrebno je investirati u Mac za testiranje

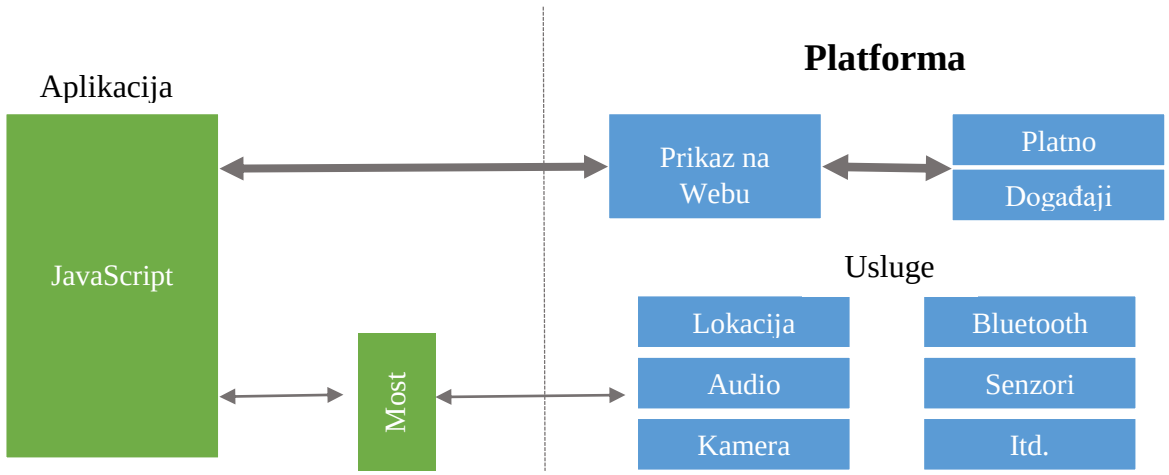
mobilne aplikacije na oba sustava. Isto tako kod višeplatformskog razvoja potrebno je odabrati programski okvir u kojemu će se razvijati aplikacija.

2.1.2. **Arhitektura programskih okvira višeplatformskih mobilnih aplikacija**

Programski okviri za višeplatformski razvoj mogu se podijeliti prema njihovoj arhitekturi na :

- Arhitekturu okvira baziranih na web tehnologijama
- Arhitekturu okvira baziranih na preslikavanju
- Arhitekturu okvira baziranih na vlastitom iscrtavanju

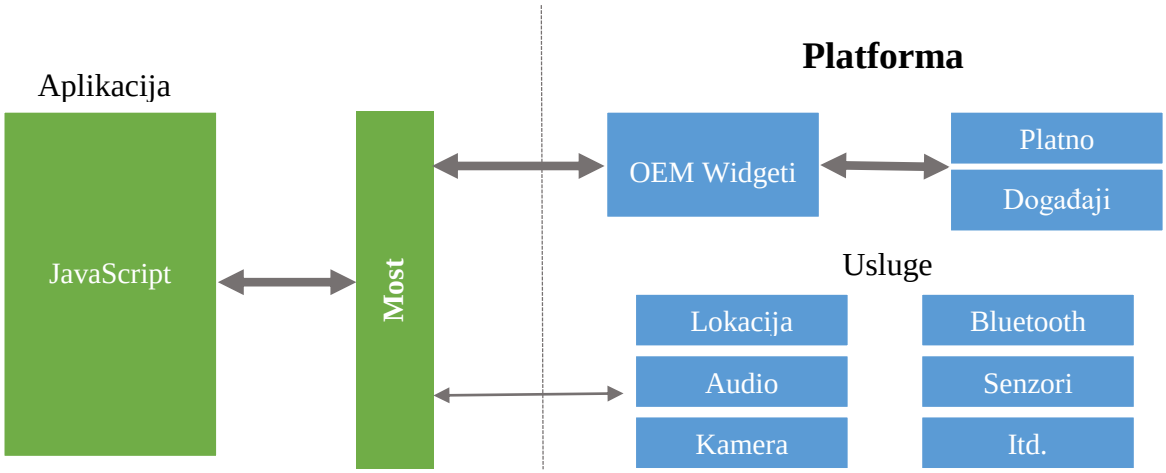
Arhitektura okvira baziranog na web tehnologijama sa slike 2.1 je prvi programski okvir za razvoj višeplatformskih mobilnih aplikacija. Baziran je na web tehnologijama gdje pomoću HTML-a i JavaScripta definira web stranice koje se prikazuju korištenjem komponente za prikaz weba u mobilnim aplikacijama – *WebView* [2]. Nedostatak okvira baziranih na web tehnologijama je otežao korištenje nativnih usluga kao što su kamera, GPS, senzori... Za korištenje tih usluga potrebno je koristiti most (engl. *bridge*). To može dovesti do sporijih performansi mobilne aplikacije. Neki od okvira baziranih na web tehnologijama su: PhoneGap, Apache Cordova i Ionic.



Sl. 2.1. *Arhitektura okvira baziranog na web tehnologijama prema [2]*

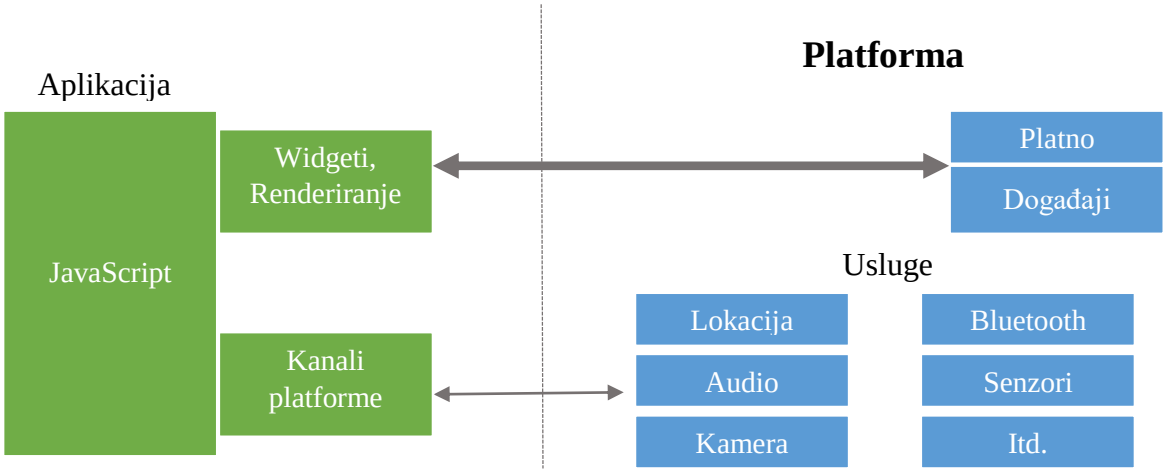
Arhitektura okvira baziranog na preslikavanju sa slike 2.2 je arhitektura koja preslikava svoje komponente u nativne komponente mobilnog operacijskog sustava. Te komponente se preko mosta između okvira i nativnog sustava preslikavaju u nativne komponente prikaza iOS ili Android operacijskog sustava [2]. Za razliku od arhitekture okvira baziranih na web tehnologijama to stvara izgled i osjećaj kao da je nativna mobilna aplikacija, ali ima isti

nedostatak korištenja mosta što utječe negativno na performanse. Neki od okvira baziranih na preslikavanju su: React Native i Xamarin.



Sl. 2.2. *Arhitektura okvira baziranog na preslikavanju prema [2]*

Arhitektura okvira baziranog na vlastitom iscrtavanju sa slike 2.3 je arhitektura gdje okvir sam iscrtava grafičke elemente korištenjem nativne mobilne komponente platna za iscrtavanje (engl. *canvas*) [2]. Okvir ima potpunu kontrolu nad grafičkim elementima čime se može postići poboljšan izgled i osjećaj nativnih mobilnih aplikacija. Koristi optimiziranu implementaciju mosta tako da se aplikacija prevodi u kod niske razine te zbog toga most ne utječe toliko na performanse. Okvir baziran na vlastitom iscrtavanju je Flutter.



Sl. 2.3. *Arhitektura okvira baziranog na vlastitom iscrtavanju prema [2]*

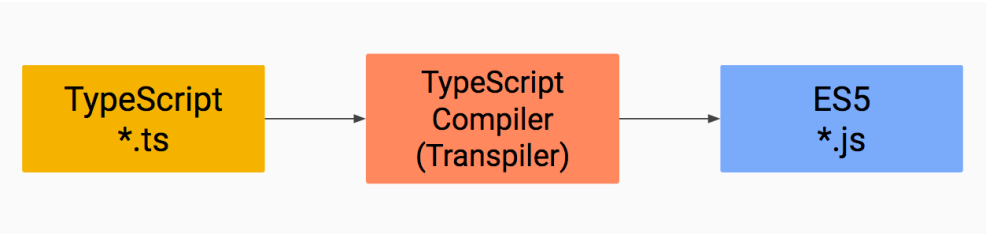
Trenutno najkorišteniji programski okviri za višeplatformski razvoj su [3] :

- React Native
- Flutter
- Ionic
- Xamarin
- PhoneGap

U ovom radu bit će korišten React Native.

2.1.3. **Transpiling koda**

Program napisan u programskom jeziku mora se pretvoriti u strojni jezik, tj. jezik niže razine kako bi bio razumljiv računalu, taj proces naziva se *compiling*. Za razliku od *compilinga*, *transpiling* je proces koji kod napisan u jednom programskom jeziku konvertira u ekvivalent koda u drugom programskom jeziku koji ima sličnu razinu apstrakcije [4]. U ovom radu koristi se transpiling programskog jezika TypeScript u programski jezik JavaScript kao na slici 2.4 React Native koristi Babel koji je besplatan *open-source transpiler* za konvertiranje novijih *ECMA Scripts* funkcionalnosti u ES5 kompatibilne funkcije koje ne bi bile drugačije podržane. *Ecma Scripts* ili ES je programski jezik opće namjene, koji je Ecma International standardizirao prema dokumentu ECMA-262 [5]. To je JavaScript standard koji treba osigurati optimalan rad web stranica na različitim web preglednicima. ES se obično koristi za skriptiranje na klijentskoj strani na W*orld Wide Webu*, a počinje se sve više koristiti za pisanje poslužiteljskih aplikacija i usluga pomoću Node.js.



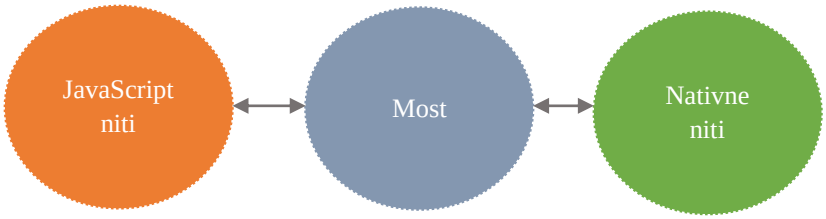
Sl. 2.4. *Prikaz Transpilinga TypeScripta [6]*

U osnovi Babel raščlanjuje kod, a zatim ponovo generira isti kod kako bi se mogao koristiti za konvertiranje. Nakon pisanja programa, potrebno je integrirati dodatke kako bi Babel mogao učiniti svoj dio. Dodaci i unaprijed postavljeni programi mogu se instalirati kao paketi pomoću

yarn ili npm upravitelja paketa za JavaScript, deklarirajući ih u datoteci naziva package.json [7]. Paketi su dostupni kao *dependencies* ili *devDependencies*. Razlika je u njihovoj upotrebi, gdje su *dependencies* potrebni za vrijeme izvođenja, dok su *devDependencies* potrebni tijekom razvoja.

Izvodeći naredbu yarn install, yarn preuzima sve potrebne *dependencies* u mapu koja se zove node_modules. Slijedeći korak je konfigurirat Babel kako bi mogao koristiti instalirane dodatke. Potrebno je kreirati datoteku .babelrc u kojoj će pisati da se trebaju koristit upravo instalirani dodaci. Babel zapravo čita .babelrc datoteku tražeći dodatke dostupne u node_modules mapi, zatim se poziva transformacija dodataka. U transformaciji se mogu dodati određene opcije koje su potrebne. Nakon toga je sve spremno za rad.

React Native se izvodi tako da postoje dva dijela: JavaScript i nativni. Oba imaju mogućnost razmjene informacija. JavaScript i nativni dio komuniciraju koristeći "most" kao na slici 2.5, koji se smatra jezgrom React Native arhitekture, dio koji daje React Native-u toliku fleksibilnost [8]. Most je koncept koji pruža dvosmjernu i asinkronu komunikaciju izmedu ova dva dijela. Važno je napomenuti da su oni u potpunosti napisani u različitim tehnologijama, ali su sposobni komunicirati.



Sl. 2.5. *Komunikacija između JavasScripta i nativnog dijela preko mosta prema [8]*

JavaScript šalje asinkrone JSON poruke koje opisuju radnju koju nativni dio treba izvršiti. Na primjer JavaScript će poslati podatke koje zaslone nativni dio treba stvoriti. Kad je nativna strana spremna, uspješno stvara zaslone. U React Native-u, most podržava ulogu posrednika za poruke, upravljajući asinkronim naredbama između dva različita dijela. Nudi više mogućnosti:

- Budući da je asinkrono, nije blokirajuće, pa omogućava glatko upravljanje prikazom na zaslonu (~ 60 sličica po sekundi je cilj React Native-a)

- S obzirom da je razdvojen i zasnovan na interoperabilnosti tj., ima mogućnost rada s različitim sustavima bez ikakvih ograničenja, otvoren je za ostale programske okvire i sustave prikazivanja pod uvjetom da poštuju komandno sučelje mosta React Native

React Native most je napravljen u C/C++ i zbog toga ima mogućnost pokretanja na različitim platformama i operacijskim sustavima. On ugrađuje Apple JavaScriptCore programski okvir u kojemu se nalaze API-ji za pristup stvarnim mogućnostima JavaScriptCore virtualnog stroja. Zbog toga se JavaScript može pokrenuti unutar C/C++ programa. On može ubrizgati varijable, funkcije i deklarirati globalne varijable kako bi poboljšao postojeći JavaScript kod. React Native se oslanja na ovu vrstu rada kako bi JavaScript komunicirao s nativnim dijelom i na taj način pokrenuo akcije u C/C++ dijelu.

S obzirom na to da je Objective-C produžetak jezika C, na iOS platformi komunikacija između ta dva jezika se događa nativno. Na taj su način razmjene između mosta i dijela Swift / Objective-C jednostavne i prirodne. Kao što se vidi na slici 2.6 JavaScript može komunicirati s JavaScriptCore okvirom, koji može komunicirat s mostom, koji komunicira sa Swiftom / Objective-C koji na kraju komunicira s nativnim dijelom.

Sl. 2.6. *Komunikacija na iOS platformi prema [8]*

Na Androidu, potrebno se oslanjati na Java nativno sučelje za komunikaciju s mostom. Prema slici 2.7 JavaScript može komunicirat s JavaScriptCore okvirom, koji može komunicirati s mostom, koji komunicira s Java nativnim sučeljem koji na kraju komunicira s nativnim dijelom.

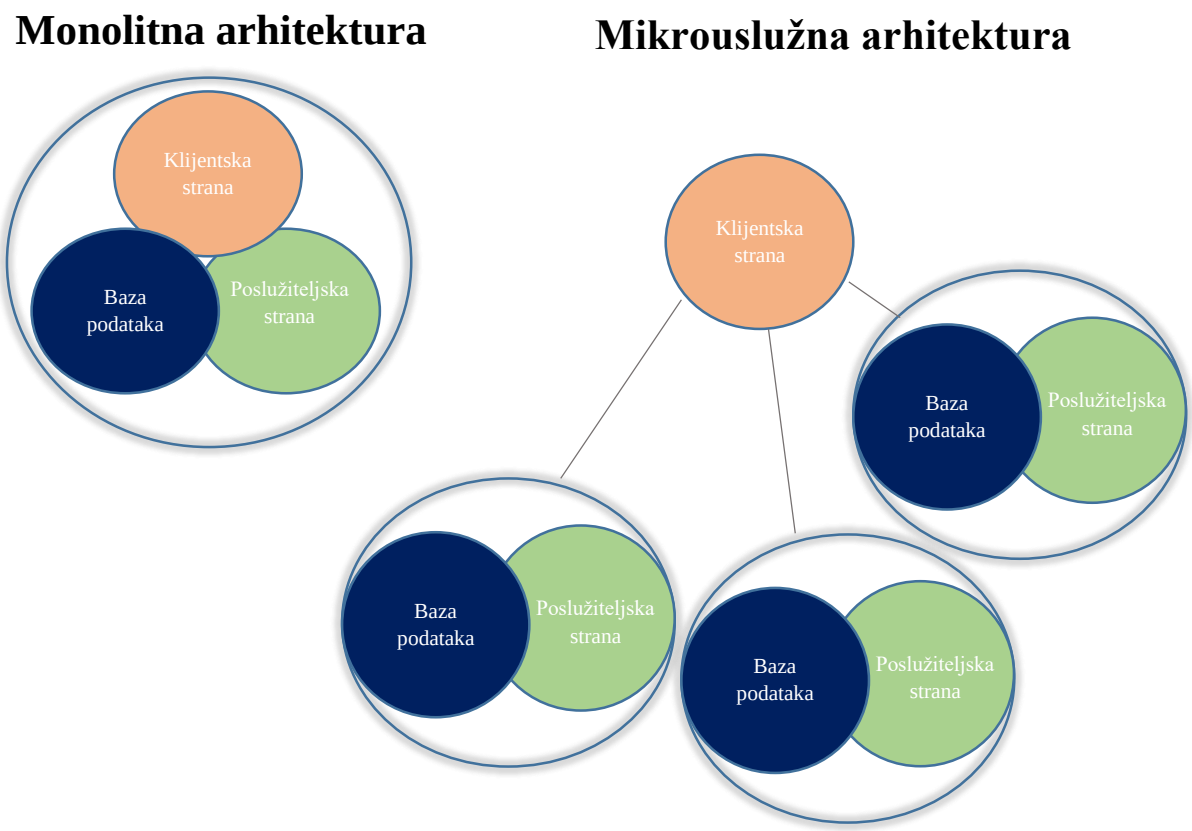
Sl. 2.7 *Komunikacija na Android platformi [8]*

Može se primijetiti da jedinu razliku ovisno o platformi čini krajnji korak prije komunikacije s nativnim dijelom.

2.1.4. **Rad s mikrouslugama**

Mikrousluge tj. arhitektura mikrousluga kao na slici 2.8 je jedinstvena metoda razvoja programskih sustava kao paketa samostalno raspoloživih, malih, modularnih usluga u kojima svaka usluga pokreće svoj proces i komunicira kroz dobro definiran i lagan mehanizam koji služi poslovnom cilju programskog sustava [9]. Kao što ne postoji formalna definicija mikrousluga, tako ne postoji ni standardni model u sustavu, temeljen na arhitekturi mikrousluga. Prema [10], glavne karakteristike i prednosti mikrousluga su:

- Sustav se može razdijeliti na više mikrousluga, kako bi se mikrousluge mogle implementirati, modificirati i objaviti bez ugrožavanja sustava.
- Stil mikrousluga je napravljen tako da su organizirane prema poslovnim zahtjevima i prioritetima. Za razliku od monolitnog razvoja, ovdje svaki tim treba napraviti specifičan proizvod baziran na temelju jedne ili više pojedinačnih usluga.
- Za razliku od monolitnih aplikacija gdje su napredni mehanizmi koordinacije i usmjeravanja, mikrousluge zahtjev primaju, procesiraju te generiraju odgovor.
- Kako mikrousluge koriste različite tehnologije i platforme, centralizirano upravljanje i pohrana nije moguća, zato se koristi decentralizirano upravljanje i pohrana te je za svaku mikrouslugu odgovoran tim koji ju je izradio.
- Jednostavnije otkrivanje te otklanjanje pogrešaka, kako se radi o mikrouslugama, jednostavnije se otkriva točno kojoj mikrousluzi pripada pogreška
- Raznovrsnost različitih platformi, jer je jednostavnije prilagoditi arhitekturu mikrousluga za rad na različitim platformama, nego razvijanje monolitnih sustava za različite platforme.



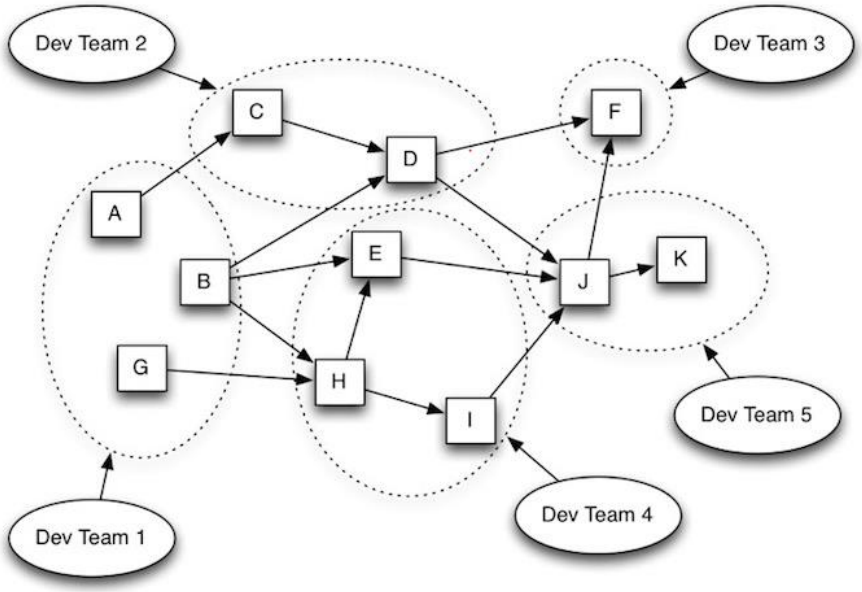
Sl. 2.8. *Monolitna i mikrouslužna arhitektura prema [11]*

S druge strane neki od nedostataka su:

- Testiranje može postati preveliko i prenaporno
- Iziskuje detaljno planiranje i točne procjene
- Arhitektura se brzo može previše zakomplicirati s prevelikim proširenjem sustava te može doći do težeg održavanja sustava
- Programeri moraju uložiti vrijeme za povezivanje mehanizama komunikacije između sustava kao na slici 2.9.
- Timovi moraju zajednički surađivati kako bi spojili sustave koje su izrađivali pojedinačno
- Mikrousluge su izvrsne za velike tvrtke, ali mogu biti sporije za primjenu i prekomplicirane za manje tvrtke.

Jasno je da postoje mnoge prednosti i nedostaci arhitekture mikrousluga, ali važno je uzeti u obzir sve prednosti i nedostatke kako bi se odabrala odgovarajuća arhitektura. Odluči li se za mikrouslužnu arhitekturu dobit će se brži, lakši, pokretljiviji razvoj aplikacije na način koji

monolitna arhitektura jednostavno ne može omogućiti. Poznate aplikacije koje koriste mikrousluge su Netflix, Amazon, eBay...



Sl. 2.9. *Prikaz sustava konstruiranog od više mikrousluga [12]*

S obzirom na to da će u ovom radu biti korišten Google Firebase, neke od mogućnosti i načini korištenja mikrousluga u Firebasu su:

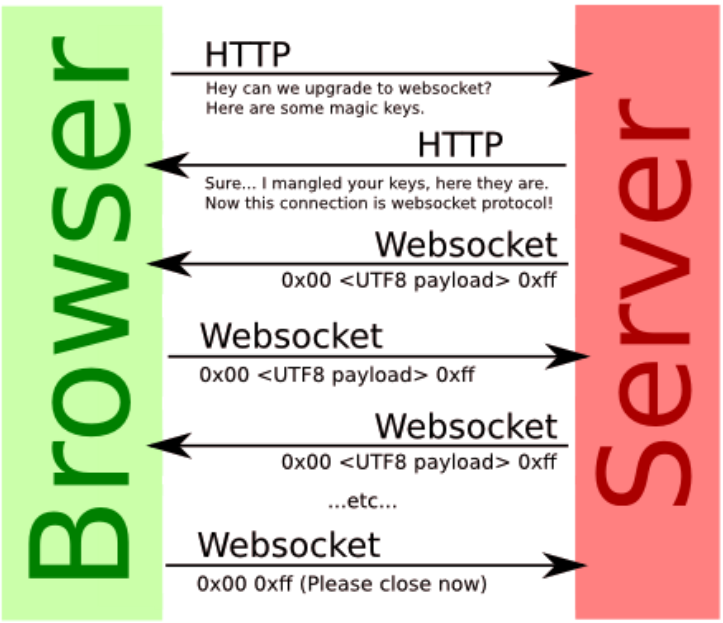
- *Cloud Firestore*
- Funkcije u oblaku
- Autentikacija korisnika
- Usluge poslužitelja
- Pohrana u oblaku
- Baza podataka u stvarnom vremenu
- Firebase strojno učenje

U potpoglavlju Firebase bit će detaljnije objašnjenje ovih nabrojanih mikrousluga.

2.1.5. **Websocket**

WebSocket protokol je računalni komunikacijski protokol koji omogućava dvosmjernu komunikaciju poslužitelja i klijenta preko jedne TCP veze kao što je prikazano na slici 2.10. Uvedeni su zbog problema obavijesti kada poslužitelj želi obavijestiti klijenta da se dogodio

neki događaj. Oni održavaju konstantnu konekciju između klijenta i poslužitelja, te ih obje strane mogu koristiti bilo kada. Početak komunikacije počinje „rukovanjem“ (eng. *handshake*). Rukovanje je zapravo HTTP upit kojeg bilo koja strana može odbiti. Nakon rukovanja započinje komunikacija. Poslužitelj sada može slati obavijesti o događajima. Okviri i biblioteke za rad s *WebSocketima* obično daju razinu apstrakcije – razvojni programeri rade sa „sobama“ u koje se korisnik „pretplati“, a zatim u određenim sobama puštaju „događaje“ koje „slušaju“ svi pretplaćeni na tu sobu [13]. Uz događaje mogu se slati i podatkovni paketi. *WebSocketi* su omogućili nove mogućnosti i sinkronizaciju u stvarnom vremenu koje do tada nisu prije bile moguće.



Sl. 2.10. *Prikaz rada WebSocketa [14]*

U ovom radu koristit će se *WebSocketi* za sinkronizaciju u stvarnom vremenu s Google Cloud Firestore bazom. Moguće je „slušanje“ izmjena na dokumentu s onSnapshot() metodom što je vidljivo na slici 2.11. Inicijalni poziv kreira snimak dokumenta na početku svakog korištenja dokumenta, zatim svaki put kad dođe do izmijene sadržaja dokumenta, drugi poziv radi izmjene snimke dokumenta [15].

```
db.collection("cities").doc("SF")
    .onSnapshot({
        // Listen for document metadata changes
        includeMetadataChanges: true
    }, function(doc) {
        // ...
    });
```

Sl. 2.11. *“Slušanje“ izmjena na dokumentu SF u zbirci cities [15]*

Lokalni zapisi u aplikaciji će pozvati *listenere* snimke istog trenutka. To se događa zbog svojstva zvanog nadoknade kašnjenja. Kada se izvrši zapis, bit će obaviješten *listener* o novom podatku prije nego što je podatak poslan u bazu. Vraćeni dokumenti imaju svojstvo metadata.hasPendingWrites koje pokazuje ima li dokument lokalne izmjene koje još nisu zapisane u bazi. Ovo svojstvo se može koristiti kako bi se odredio izvor događaja koju je primio *listener*.

Isto tako moguće je „slušati“ više dokumenata u zbirci. Koristi se onSnapshot() umjesto metode get() kako bi se „slušalo“ rezultate upita. To će zatim kreirati snimku upita. Snimka će primiti novu snimku upita svaki put kad se upit promjeni. To se događa kad je dokument dodan, obrisan ili promijenjen. Prilikom izmjene dokumenta ili ako je dodan dokument to se naplaćuje kao jedno čitanje dokumenta. Na Firebase-u postoji besplatna kvota koju se može iskoristiti prije daljnje naplate prema tablici 2.2.

Tab. 2.2. *Besplatna kvota pri korištenju Firestore-a* [16]

Besplatno	**Kvota**
Spremanje podataka	1 GiB
Čitanje dokumenata	50,000 čitanja po danu
Pisanje dokumenata	20,000 pisanja po danu
Brisanje dokumenata	20,000 brisanja po danu

Nakon iskorištavanja besplatne kvote izvodi se daljnja naplata prema tablici 2.3.

Tab. 2.3. *Naplata za čitanje, pisanje, brisanje i spremanje podataka na Firestore-u* [17]

	Naplata nakon besplatne kvote
Spremanje podataka	$0.18/GiB/mjesec
Čitanje dokumenata	$0.06 za 100,000 dokumenata
Pisanje dokumenata	$0.18 za 100,000 dokumenata
Brisanje dokumenata	$0.02 za 100,000 dokumenata

Kada je gotovo „slušanje“ podataka na nekom dokumentu, potrebno je onemogućiti *listener* kako se više ne bi pozivao. To omogućuje klijentu da prestane koristiti mrežnu propusnost kako bi primao izmjene. Za to se koristi metoda unsubscribe().

2.2. Sinkronizacija u stvarnom vremenu

U računalnoj znanosti sinkronizacija se odnosi na jedan od dva različita, ali povezana koncepta: sinkronizacija procesa i sinkronizacija podataka. Sinkronizacija procesa odnosi se na ideju da se više procesa treba spojiti ili rukovati u određenoj točki, kako bi se postigao dogovor ili obvezali na određeni slijed radnje. Sinkronizacija podataka odnosi se na ideju održavanja više kopija skupa podataka u međusobnoj koherentnosti ili održavanja integriteta podataka [18]. Kod novijih tehnologija, pogotovo u automobilskoj industriji uvode se nove sheme vremenskog razdvajanja za učinkovitiju sinkronizaciju. Prema [19], dvije glavne sheme vremenskog razdvajanja su statički i dinamički kvant. Ako su izdvedbe simulacije najvažnije i točnost do određenog stupnja zanemariva, statički kvant je prihvatljiva opcija. Ipak kvantnu veličinu programer treba postaviti i zahtijeva postupak finog podešavanja kako bi se pronašao optimalni kompromis. Dinamički kvant je alternativa statičkom kvantu, dajući savršenu točnost postavljanjem kvantne granie u vrijeme sljedećeg obavještavanja o događaju. Tako se ni jedan događaj ne može propustiti ili riješiti prekasno.

Kako bi sustav bio sinkroniziran u stvarnom vremenu ponašanje sustava mora biti vremenski usklađeno. Nije dovoljna samo logička ispravnost (ispravan rezultat nekog proračuna), već je potrebna i vremenska ispravnost. Zadatak je potpuni logički slijed izvršenja posla. Mogu se razvrstati prema zahtjevima za rok, prevencijom i karakteristikom dolaska [20]. Prema zahtjevima za rok zadaci u sustavima u stvarnom vremenu mogu biti strogi i ublaženi (eng. hard, soft). Zadatak je strog ako njegovo vrijeme ne zadovolji vremenski rok te time izazove fatalnu pogrešku [21]. Primjerice, kasna zapovijed zaustavljanja vlaka može izazvati sudar. Za razliku od strogog zadatka, ne zadovoljavanje vremenskog roka kod ublaženog zadatka bit će nepoželjno, ali ne fatalno. Međutim, nekoliko promašaja ublaženih rokova ne donosi ozbiljnu štetu. Ukupna izvedba sustava postajat će sve lošija i lošija kad sve više zadataka s ublaženim rokovima završi kasno. Prema prevenciji zadaci se smatraju međusobno preventivnim. Što se tiče karakteristike dolaska, zadaci mogu biti periodični, što znači da se pojavljuju u jednakim intervalima, tj, jednakom periodu [20]. Ako se zadaci pojavljuju u nepravilnim razmacima, nazivaju se aperiodičnim i da bi bili raspoređeni moraju imati poznato vrijeme dolaska.

Prilikom definiranja dizajna i arhitekture te razvoja programskih proizvoda potrebno je obratiti pozornost na parametre budućeg programskog proizvoda o kojima će odabir sinkronizacijskog mehanizma i implementacija sinkronizacije ovisiti. U obzir treba uzeti parametre [22]:

- Smjer sinkronizacije podataka – u različitim scenarijima nije uvijek potrebno vršiti sinkronizaciju u oba smjera
- Učestalost sinkronizacije podataka - potreba za učestalošću sinkronizacije u pojedinim poslovnim sustavima
- Brzina sinkronizacije podataka – različiti mehanizmi imaju značajnu razliku u trajanju sinkronizacije. Na primjer sinkronizacija koja je implementirana na standardnim web uslugama traje nekoliko puta duže nego sinkronizacija koja je implementirana direktnom TCP vezom između dva uređaja.
- Zahtjevi za sigurnošću podataka – mogu rezultirati zabranom sinkronizacije ako je uređaj spojen na nesigurnu vezu ili se mogu uključiti po potrebi dodatni sigurnosni mehanizmi
- Uključeni uređaji u sinkronizaciji – ovisnost o drugim infrastrukturnim uređajima kao što su web ili mobilni poslužitelji

U ovoj aplikaciji kako bi se sinkroniziralo u stvarnom vremenu koristit će se Websocket listeneri s Firebase bazom podataka koji su objašnjeni u prošlom poglavlju. Jedan od primjera zašto je potrebna sinkronizacija u stvarnom vremenu za ovu aplikaciju bit će kupovina ulaznice (zapis u bazi podataka) i korištenje ulaznice odmah nakon toga.

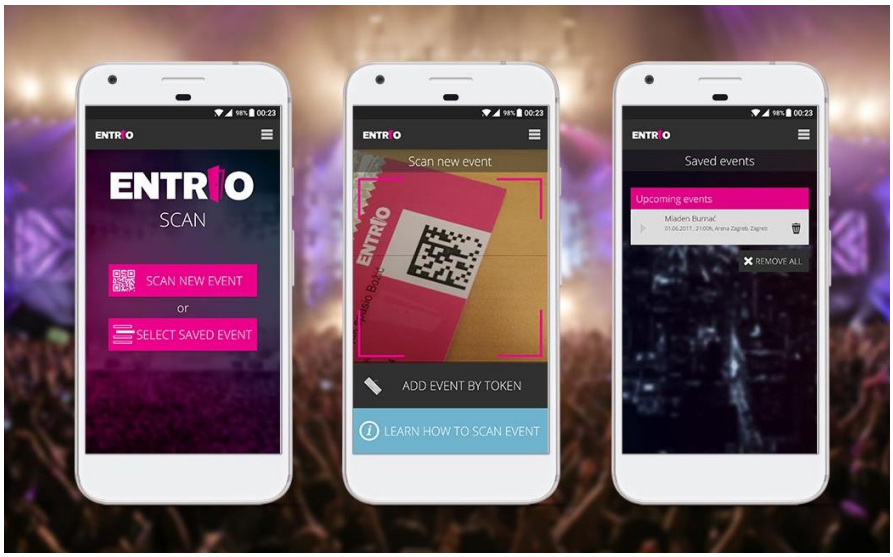
2.3. Prikaz sličnih postojećih rješenja aplikacija i sustava

U današnje vrijeme sve je veća potreba za online kupovinom ulaznica, pretrage te prijave događaja. Problem je što većina davatelja takvih usluga ne nude kompletno rješenje putem Web i mobilnih aplikacija. Poznate tvrtke koje nude ovakve usluge su :

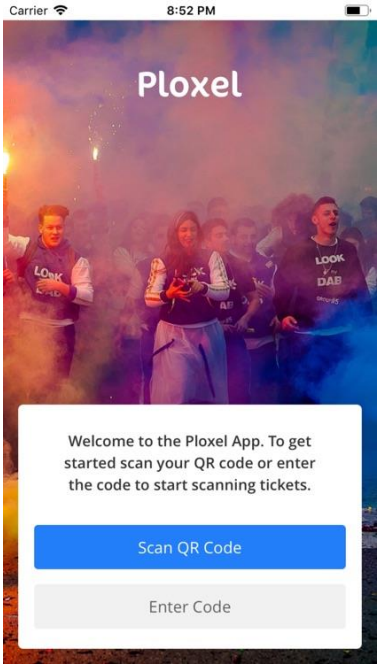
- Entrio – mogućnost prodaje i distribucije
- Eventim – sustav za registraciju i prodaju ulaznica te registracije sudionika na konferencijama, korporativnim događajima i sl. Radi lakšeg korištenja i pretrage događaja postoji mogućnost prijave i korištenja korisničkog računa. Isto tako nudi mogućnost bezgotovinskog plaćanja na događajima te svoj Entrio stream za virtualne i hibridne događaje s prodajom ulaznica [23]. Putem Entrio web stranice moguće je napraviti događaj dok mobilna aplikacija Entrio scan sa slike 2.12 omogućuje jednostavnu i brzu provjeru ulaznica i registraciju posjetitelja na ulazu događaja putem QR koda.
- Eventim – sustav za rezervaciju i prodaju ulaznica, osim jednostavnog i funkcionalnog prikaza prodaje i rezervacije ulaznica za razne događaje nudi mogućnost izvještaja prodaje

po svakom kanalu prodaje [24]. Postoji Eventim mobilna aplikacija, ali je napravljena samo za njemačko tržište.

- Ticket Tailor – sustav prilagođen kreatorima događaja. Nudi različite mogućnosti prodaje ulaznica i organiziranje događaja, sve do izrade popisa i mjesta za sjedenje [25]. Jednostavna i inovativna rješenja koja nude i Wordpress dodatke za direktnu prodaju sa svoje web stranice. Nedostatak je što nemaju mobilnu aplikaciju za kompletno povezivanje.
- Bizzabo – platforma za virtualna, hibridna događanja uživo. Omogućuje jednostavno upravljanje ulaznicama i registracijama, virtualizaciju, integrirana rješenja i bogatu analitiku događaja [26]. Bizzabo ima odličnu mobilnu aplikaciju koja omogućuje push notifikacije, razgovore, ankete te interaktivno sučelje. Isto tako ima mogućnost prijave na događaj putem mobilne aplikacije te zbog toga nije potrebno ispisivanje ulaznice.
- Ploxel – kompletna platforma za prodaju ulaznica. Omogućuje dizajniranje košarice, korisničku podršku, grupnu kupovinu ulaznica i mogućnost integracije košarice na vlastitu web stranicu, Wordpress ili Facebook stranicu [27]. Ploxer platforma ne postoji kao mobilna aplikacija, ali je web aplikacija potpuno responzivna i može radit na mobilnim i tablet uređajima. Ploxer ima mobilnu aplikaciju Ploxel Check In sa slike 2.13 za provjeru ulaznica posjetitelja putem QR koda.



Sl. 2.12. *Mobilna aplikacija za skeniranje ulaznice Entrio [23]*



Sl. 2.13. *Mobilna aplikacija za skeniranje ulaznice Ploxel [27]*

3. IDEJNO RJEŠENJE I ARHITEKTURA MOBILNE APLIKACIJE TICKETNATOR

U ovom poglavlju bit će opisani funkcionalni i nefunkcionalni zahtjevi za rad mobilne aplikacije i zahtjevi za izradu mobilne aplikacije Ticketnator. Prema zahtjevima su izrađeni idejno rješenje i model mobilne aplikacije. Za kraj poglavlja opisana je arhitektura mobilnih aplikacija s osvrtom na višeplatformski razvoj koji će se koristit za izradu mobilne aplikacije Ticketnator.

3.1. Funkcionalni i nefunkcionalni zahtjevi za rad mobilne aplikacije

Kako bi aplikacija bila što uspješnija potrebno je odvojiti vrijeme za definiranje zahtjeva. Zahtjevi se mogu podijeliti na funkcionalne i nefunkcionalne zahtjeve. Definiranje funkcionalnih i nefunkcionalnih zahtjeva može pomoći pri [28]:

- Određivanju pravila i uloga
- Smanjenju vremena potrebnom za komunikaciju tijekom razvoja
- Preciznijoj estimaciji projekta
- Uočavanju pogrešaka na početku
- Kreiranju preciznijeg projekta

Funkcionalni zahtjevi u aplikaciji definiraju sustav ili njegove komponente, tj. opisuju funkcionalnosti koje aplikacija mora činiti. Nefunkcionalni zahtjevi su zahtjevi koji definiraju način kako sustav tj. aplikacija izvodi određenu funkciju. Tablicom 3.1 navedeni su zahtjevi koje bi mobilna aplikacija Ticketnator trebala ispuniti.

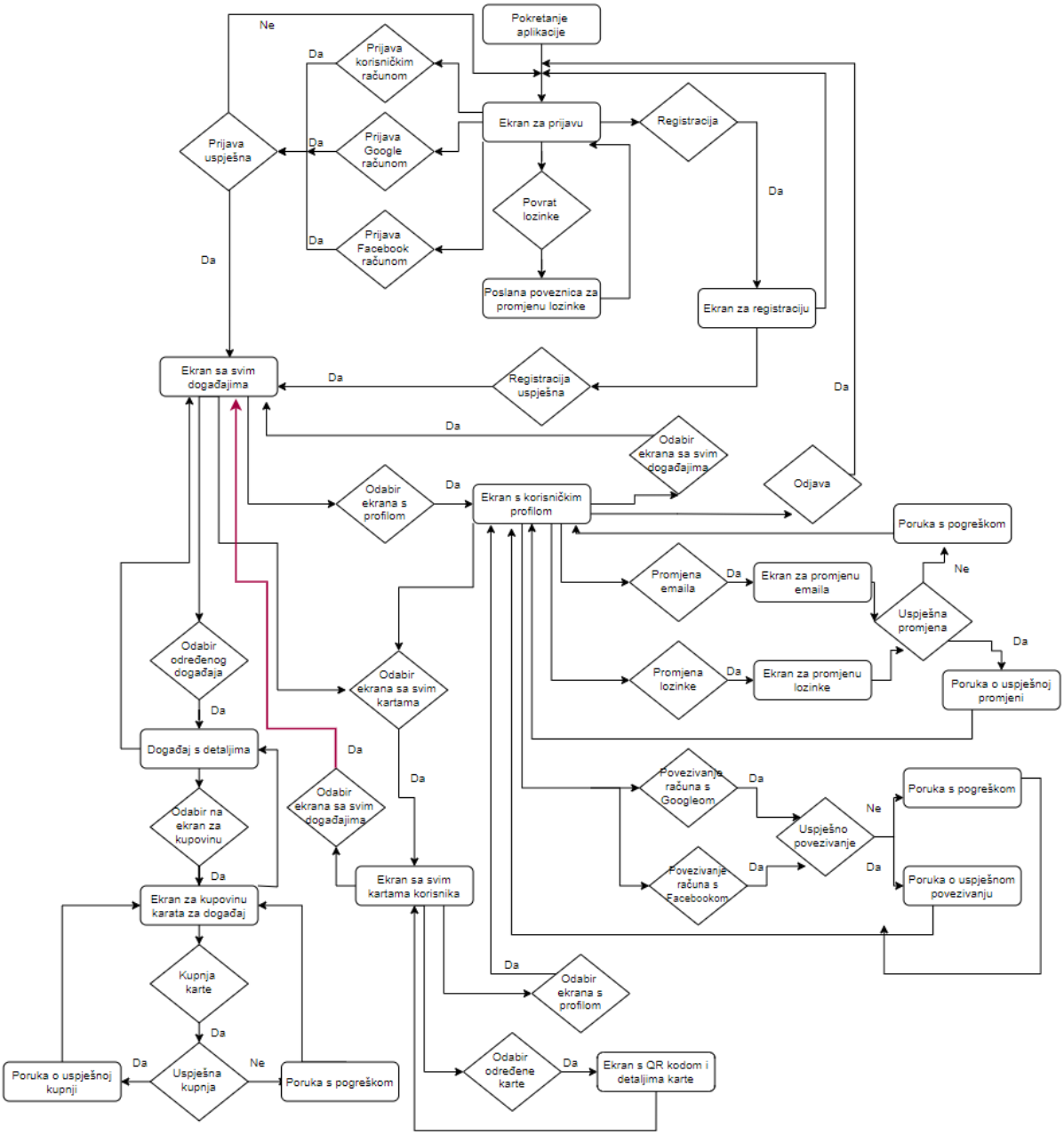
Tab. 3.1. *Funkcionalni i nefunkcionalni zahtjevi mobilne aplikacije Ticketnator*

Funkcionalni zahtjevi	**Nefunkcionalni zahtjevi**
Kreiranje korisničkog računa	Brzi odgovor aplikacije za određenu akciju
Prijava korisničkim računom	Sigurnost
Povrat lozinke korisničkog računa	Prenosivost
Prijava Google računom	Mogućnost održavanja
Prijava Facebook računom	Skalabilnost
Prikaz pogreške prilikom prijave	Učinkovitost
Prikaz učitavanja prilikom prijave	Upotrebljivost
Prikaz svih događaja	Pouzdanost
Filtrer događaje prema tipu događaja	Mogućnost suradnje u radu
Prikaz detalje događaja	Mogućnost testiranja
Različite vrste opcija ulaznica za događaj	Ponovna upotreba

Kupovina ulaznice za određeni događaj i smanjivanje broja dostupnih ulaznica	
Prikaz određene poruke ovisno o uspješnosti akcije kupovine ulaznice	
Ako je ulaznica uspješno kupljena treba ju prikazati u svim ulaznicama	
Prikaz QR kod i detalje određene ulaznice	
Promjena e-pošte	
Prikaz određene poruke ovisno o uspješnosti akcije promjene e-pošte	
Promjena lozinke	
Prikaz određene poruke ovisno o uspješnosti akcije promjene e-pošte	
Povezivanje korisničkog računa s Google računom	
Prikaz određene poruke ovisno o uspješnosti akcije povezivanja korisničkog računa s Google računom	
Povezivanje korisničkog računa s Facebook računom	
Prikaz određene poruke ovisno o uspješnosti akcije povezivanja korisničkog računa s Facebook računom	
Odjava iz aplikacije	

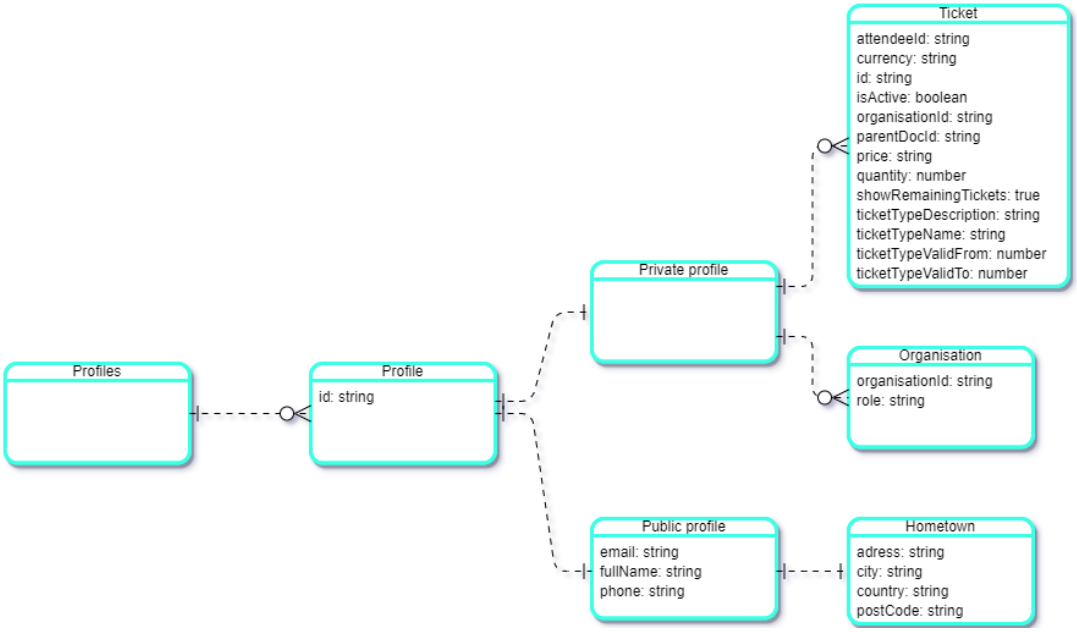
3.2. Idejno rješenje i model mobilne aplikacije

Idejno rješenje mobilne aplikacije Ticketnator zadano je dijagramom toka na slici 3.1. Pomoću dijagrama toka razrađeni su zasloni i akcije aplikacije Ticketnator što će smanjiti greške prilikom programiranja aplikacije



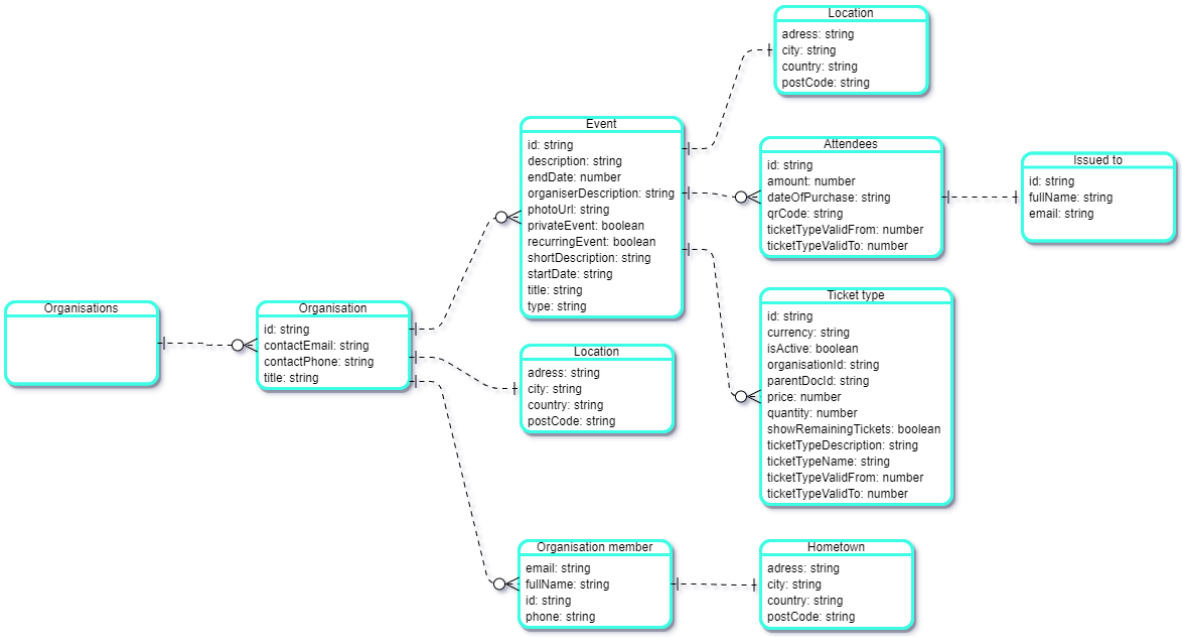
Sl. 3.1. *Dijagram toka mobilne aplikacije Ticketnator*

Za prikaz modela baze podataka u Firebase-u koristit će se ER dijagram. ER dijagram prikazuje međusobne odnose između entiteta. Slika 3.2 pokazuje zbirku Profila u bazi podataka. Prema slici zbirka Profiles može imat 0 ili više dokumenata Profile. Dokument profile ima jednu zbirku Public profile i jednu zbirku Private profile. Zbirka Public profile ima jednu podzbirku Hometown. Zbirka Private profile može imat 0 ili više podzbirki Ticket i 0 ili više podzbirki Organisation.



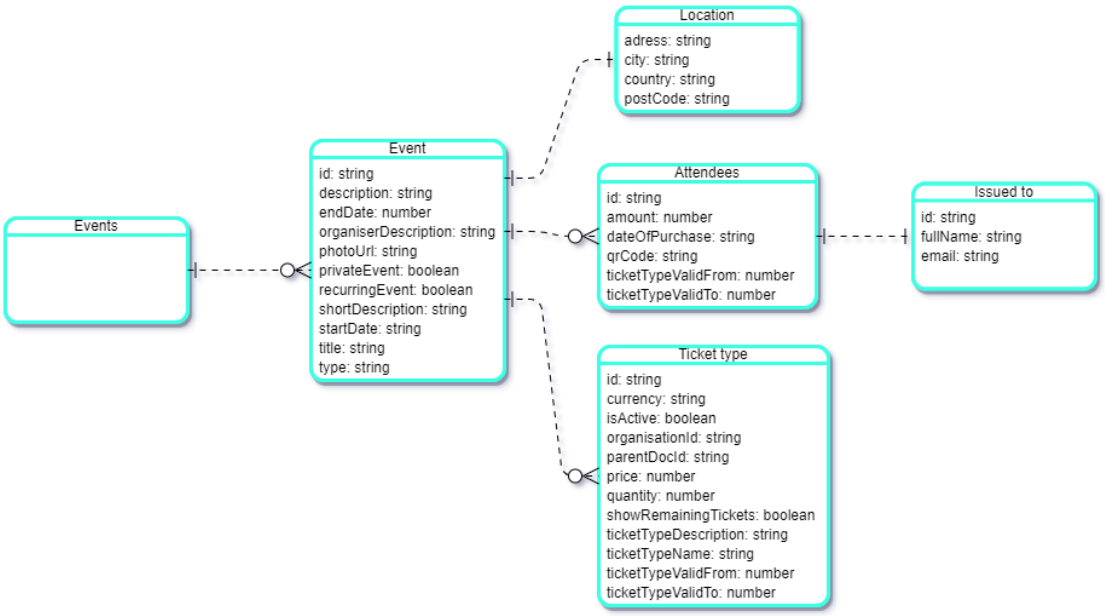
Sl. 3.2. *ER dijagram Profila za bazu podataka*

Slika 3.3 pokazuje zbirku Organizacija u bazi podataka. Prema slici zbirka Organisations može imat 0 ili više dokumenata Organisation.Dokument Organisation može imat 0 ili više zbirki Event, jednu podzbirku Location i 0 ili više zbirki Organisation member. Zbirka Event može imati jednu podzbirku Location, 0 ili više podzbirki Attendees i 0 ili više podzbirki Ticket type. Podzbirka Attendees ima jednu podzbirku Issued to. Zbirka Organisation member ima jednu podzbirku Hometown.



Sl. 3.3. *ER dijagram Organizacija za bazu podataka*

Slika 3.4 pokazuje zbirku Događaja u bazi podataka. Prema slici zbirka Events može imati 0 ili više dokumenata Event. Dokument Event može imati jednu podzbirku Location, 0 ili više podzbirki Attendees i 0 ili više podzbirki Ticket type. Podzbirka Attendees ima jednu podzbirku Issued to.



Sl. 3.4. *ER dijagram Događaja za bazu podataka*

3.3. Arhitektura mobilne aplikacije i osvrt na višeplatformski razvoj

Arhitektura aplikacije je skup tehnologija i modela za razvoj potpuno strukturiranih mobilnih programa zasnovanih na industrijskim standardima i standardima klijenta [29]. Određeni elementi arhitekture odabiru se na temelju značajki i zahtjeva aplikacije. Ispunjavanje uvjeta arhitektura omogućuje ubrzavanje razvoja i znatno olakšavanje budućeg održavanja. Na taj način štedi se vrijeme i novac. Isto tako dobrim odabirom arhitekture dobit će se najbolje rješenje složenih poslovnih problema na najučinkovitiji način za mobilne aplikacije. Tako će se lakše zadovoljiti različiti nefunkcionalni zahtjevi koji su ranije već spomenuti.

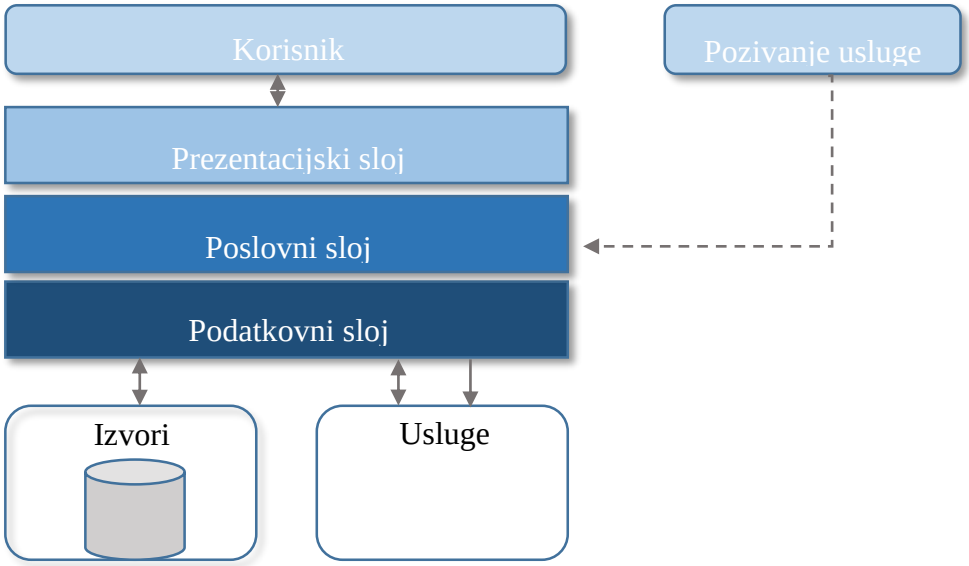
Dizajn arhitekture mobilne aplikacije obično se sastoji od više slojeva što je prikazano slikom 3.5, uključujući [30]:

- Prezentacijski sloj
- Poslovni sloj
- Podatkovni sloj

Prezentacijski sloj istražuje kako predstaviti aplikaciju krajnjim korisnicima. Prilikom dizajniranja ovog sloja, programeri moraju odrediti ispravnu vrstu klijenta za predviđenu infrastrukturu [29]. Obuhvaća komponente korisničkog sučelja i procese korisničkog sučelja. U ovoj fazi također treba odlučiti o važnim stvarima poput tema, fontova, boja itd.

Poslovni sloj predstavlja jezgru mobilne aplikacija sa svojim funkcionalnostima [29]. Mobilna aplikacija sloj poslovne logike može na daljinu postaviti na poslužitelj korisnika kako bi se smanjilo opterećenje. Smanjuje se tako opterećenje zbog ograničenih resursa dostupnih na mobilnim uređajima. Sloj uključuje poslovne komponente, tijek rada i entiteta , model domene i usluga.

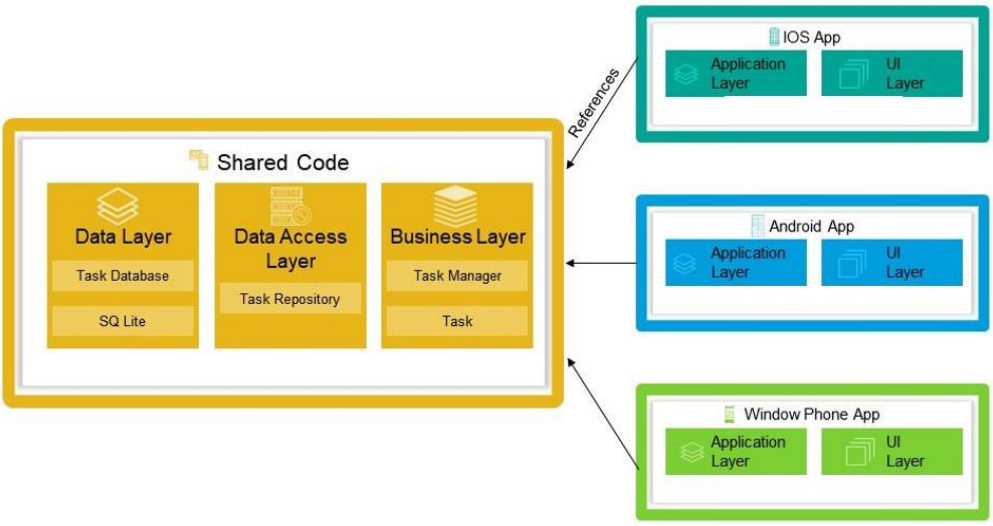
Podatkovni sloj mora udovoljavati zahtjevima aplikacije i treba pomoći u pružanju učinkovitih i sigurnih podatkovnih transakcija [31]. Programeri mobilnih aplikacija trebaju razmotriti i stranu održavanja podataka, istovremeno osiguravajući da se podatkovni sloj može lako mijenjati s promjenjivim poslovnim zahtjevima. Ovaj se sloj sastoji od komponenti specifičnih za podatke kao što su pristupne komponente i uslužne usluge. Isto tako bitno je za dizajniranje ovog sloja odabir ispravnog formata podataka i uspostavljanje snažne provjere valjanosti unosa podataka.



Sl. 3.5. *Arhitektura nativne mobilne aplikacije prema [29]*

Razlika između arhitekture nativnih i višeplatformskih aplikacija je što se dodaje aplikacijski sloj i korisničko sučelje što se može vidjeti na slici 3.6. Programski jezik kojim će bit napisan aplikacijski sloj i korisničko sučelje ovisi o operacijskom sustavu mobilne aplikacije za koji se

radi arhitektura. Složenost i stalni razvoj arhitekture mobilnih operacijskih sustava kao što su Android ili iOS i velika raznolikost mobilnih uređaja dovodi do neprestanog razvoja nove arhitekture programskih okvira [32]. Programeri se moraju prilagođavati novoj arhitekturi jer s novom arhitekturom dolazi i potencijalno stvaranje novih pogrešaka.



Sl. 3.6. *Arhitektura višeplatformskih aplikacija [33]*

U aplikaciji Ticketnator sloj korisničkog sučelja bit će dizajniran prema Redux obrascu razvoja aplikacije. Redux se koristi kao spremnik stanja za lakše dijeljenje podataka kroz aplikaciju. Njegovo korištenje bit će objašnjeno detaljnije u slijedećem poglavlju u potpoglavlju 4.7.

4. PROGRAMSKO RJEŠENJE TICKETNATORA

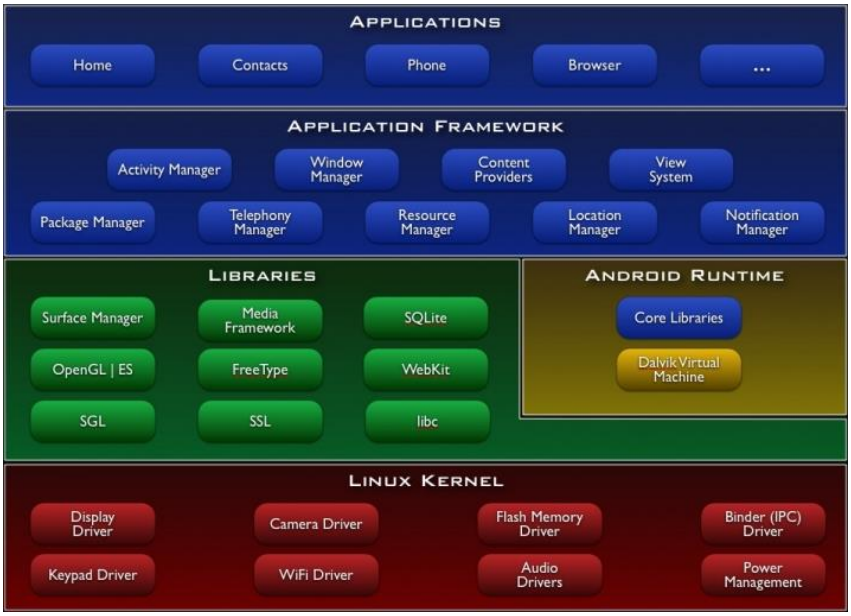
U ovom poglavlju bit će opisani programski jezici i tehnologije koje se koriste u aplikaciji za pretragu i prijavu događaja. Isto tako opisano je programsko rješenje koje je razdvojeno na dva dijela: programsko rješenje na strani poslužitelja i programsko rješenje na strani klijenta.

4.1. Programski jezici i tehnologije

4.1.1. Operacijski sustav Android

Android je operacijski sustav napravljen 2003. godine u Kaliforniji temeljen na Linuxu, napravljen prvenstveno za mobilne uređaje sa zaslonom osjetljivim na dodir [34]. Jedan je od najčešće korištenih mobilnih operacijskih sustava. Android ima otvoren kod što znači da je besplatan i da ga svatko može koristiti. Android arhitektura prema slici 4.1 je podijeljena na pet dijelova:

- Linux jezgra – srž operativnog sustava koji upravlja ulaznim i izlaznim zahtjevima programske podrške. To omogućuje osnovne funkcionalnosti kao što su upravljanje procesima, upravljanje memorijom, upravljanje uređajima poput kamere, tipkovnice, zaslona...
- Biblioteke – nakon Linux jezgre ide skup biblioteka. Te biblioteke koriste se za različite usluge, primjerice, SQLite je baza za pohranu i razmjenu podataka o aplikacijama, SSL biblioteke su zadužene za internet sigurnost itd.
- Android runtime – procesni virtualni stroj u Android operacijskom sustavu koji pokreće aplikacije na Android uređajima. Koristi Linux osnovne značajke kao što su upravljanje memorijom i višenitnost. Omogućava da svaka Android aplikacija pokrene svoj proces.
- Aplikacijski okvir – okvirni sloj aplikacija koji pruža brojne usluge viših razina aplikacijama kao što su upravitelj procesima, upravitelj paketa, upravitelj resursa... Programeri aplikacija mogu koristiti ove usluge u njihovim aplikacijama.
- Aplikacije i značajke – Sve aplikacije se instaliraju na ovaj sloj i nalaze na njemu. Primjeri takvih aplikacija su kontakti, glazba, poruke... Svaka aplikacija ima različitu ulogu u cjelokupnom skupu aplikacija.



Sl. 4.1. *Arhitektura Android operacijskog sustava [35]*

Za razvoj Android nativnih aplikacija najviše se koriste programski jezici: Java, Kotlin, C++, C#. Trenutna inačica Androida je Android 10 te neke od njegovih glavnih značajki su [36]:

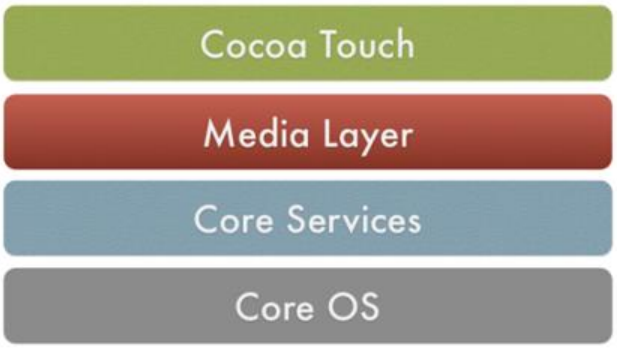
- Zapis uživo – automatski na zaslonu zapisuje što je rečeno na videu, glazbi ili podcastu bez potrebe za korištenjem interneta
- Pametni odgovor – pametni odgovor nudi preporučenu reakciju na dobivenu poruku, primjerice, ako je adresa u poruci, ponudit će otvaranje uputa na Google kartama.
- Pojačivač zvuka – mobitel može pojačati zvuk, filtrirati pozadinski šum i poboljšati zvuk koji se čuje. Potrebno je samo staviti slušalice i čuje se jasnije.
- Navigacija gestama – Brže i inovativnije geste za kretanje sa zaslona na zaslon
- Tamna tema – nova tamna tema koja smanjuje naprezanje očiju i pomaže trajanju baterije.
- Veća privatnost – nove pametnije kontrole koje omogućuju da korisnik odluči kako i kada se podaci na njegovom uređaju dijele.
- Obiteljska veza – postavljanje pravila i stvaranje zdravih navika kod djece tako da se može postaviti vremenska ograničenja zaslona i ograničit određeni sadržaj.

4.1.2. **Operacijski sustav iOS**

iOS je operacijski sustav napravljen u tvrtki Apple 2007. godine u Kaliforniji. Ova se inačica operacijskog sustava u vrijeme kad je napravljena nije zvala iOS [37]. Apple je od verzije 1-3 iOS nazivao Iphone OS. Tek dolaskom četvrte verzije naziva se iOS. Za razliku od

Androida iOS nema otvoren kod i napravljen je samo za Apple-ove mobilne uređaje tj. iPhone. iOS arhitektura prema slici 4.2 je podijeljena na četiri dijela [38]:

- Core OS – svi iOS izgrađeni su na značajkama niskih razina koje pruža Core OS sloj. Te tehnologije uključuju Core Bluetooth okvir, Accelerate okvir, vanjski dodatni okvir, okvir sigurnosnih usluga, okvir za lokalnu autentikaciju
- Core Services – neki od najbitnijih okvira koji se tu nalaze su : okvir s kontaktima, okvir za oblak, okvir za lokaciju, socijalni okvir...
- Media – grafička, audio i video tehnologija su omogućene pomoću Media sloja.
- Cocoa Touch – krajnji sloj koji sadrži okvir za događaje, okvir za igre, okvir za karte...



Sl. 4.2. *Arhitektura iOS operacijskog sustava [39]*

Za razvoj iOS nativnih aplikacija najviše se koristi Objective-C, Swift, C++, C#. Trenutna inačica iOS-a je iOS 13 te neke od njegovih glavnih značajki su [40]:

- Tamni način rada – Način rada koji se može uključiti u Kontrolnom centru ili postaviti da se automatski aktivira kada padne mrak.
- Poboljšanje fotografije i kamera – Novi organizirani pregled najboljih fotografskih trenutaka iz svakog dana, mjeseca i godine. Novi alati za kontrolu filtera i precizno uređivanje fotografija. Gotovo svaki alat i efekt koji se primjenjuje na fotografijama može se sada primijeniti i na videozapisima.
- Zaštita privatnosti i sigurnosti – Jednostavan način prijave u aplikacijama i na internetskim stranicama koji poštuju korisnikovu privatnost te nov i pouzdan način arhiviranja snimki video nadzora.
- Karte – 3D prikaz gradova koje se mogu rotirati za 360 stupnjeva. Mogućnost pravljenja popisa mjesta koje korisnik želi posjetiti na slijedećem putovanju i dijeljenje tog popisa s prijateljima.

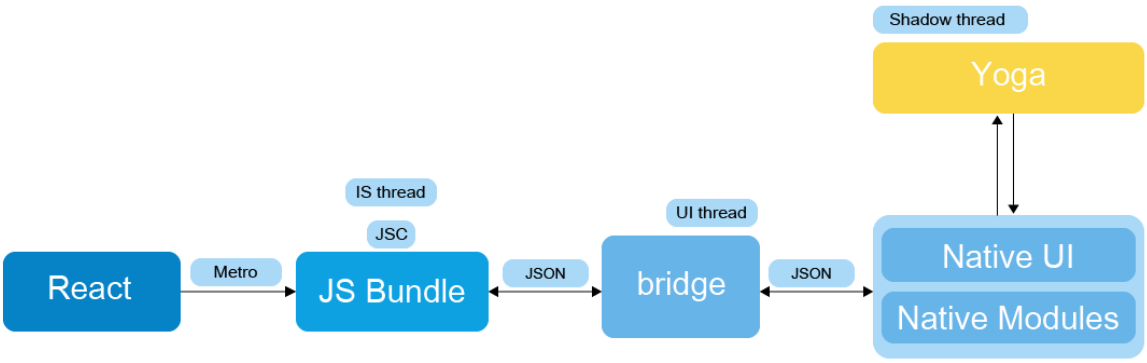
- Performanse – Brži od iOS-a 12. Na primjer Face ID otključava uređaj do 30 posto brže te se aplikacije pokreću do dva puta brže i lakše su za preuzimanje.
- Proširena stvarnost – AR aplikacije mogu smjestiti virtualne objekte prirodno ispred ili iza ljudi što iskustvo proširene stvarnosti čini još impresivnijim.

4.1.3. **Programski okvir React Native**

React Native je programski okvir otvorenog koda kojeg je napravio Facebook 2015. godine [41]. Prema arhitekturi koja se nalazi na slici 4.3 spada u arhitekturu okvira baziranog na preslikavanju. S obzirom na to da je sustav otvorenog koda, daje korisniku mogućnosti izmjene, dodavanje značajki te otklanjanje grešaka besplatno. JavaScript okvir koji radi s iOS i Android operacijskim sustavima, dok nativni razvoj za Android i iOS koriste tehnologije kao što su Kotlin, Java, Objective-C, Switft... Glavna biblioteka React Native mobilnih aplikacija je prilično mala i ovisi o bibliotekama trećih strana. Mnoge biblioteke zahtjevaju ručno povezivanje jer sadrže nativni kod. Zbog toga programeri moraju poznavati i Javu te Swift/Objective-C.

Nadalje React Native ne podržava sve API-je. Podržava najkorištenije API-je, ali za neke specifične kao što je rečeno ranije, potrebno je koristiti nativni most za povezivanje nativnog i JavaScript koda. Također možda najveći problem što React Native ne podržava višestruku obradu ili paralelne niti. Sastoji se samo od jedne JavaScript niti zbog čega može doći do sporijih performansi. Isto tako moguće je nativno napraviti dijelove kojima je potrebno da imaju paralelne niti.

Kao što je spomenuto ranije, najveća prednost React Native-a tj. glavna značajka višeplatformskih mobilnih aplikacija je što se većinu koda može dijeliti između Androida i iOS-a, što smanjuje kompleksnost. Omogućuje i Web programerima bržu tranziciju s web razvoja na razvoj mobilnih aplikacija zbog JavaScript programskog jezika. Zbog toga, znanje React Native-a se može primijeniti u Reactu (Web aplikacije) te u Protonu (PC aplikacije). Isto tako ponovna upotreba koda između gore navedenih tehnologija je velika značajka. Velika prednost React Native-a je što ima mogućnost *Hot reloadinga* koja omogućuje osvježavanje samo onih datoteka koje su se promijenile, bez gubitka ostalih stanja u aplikaciji, što štedi vrijeme i povećava produktivnost.

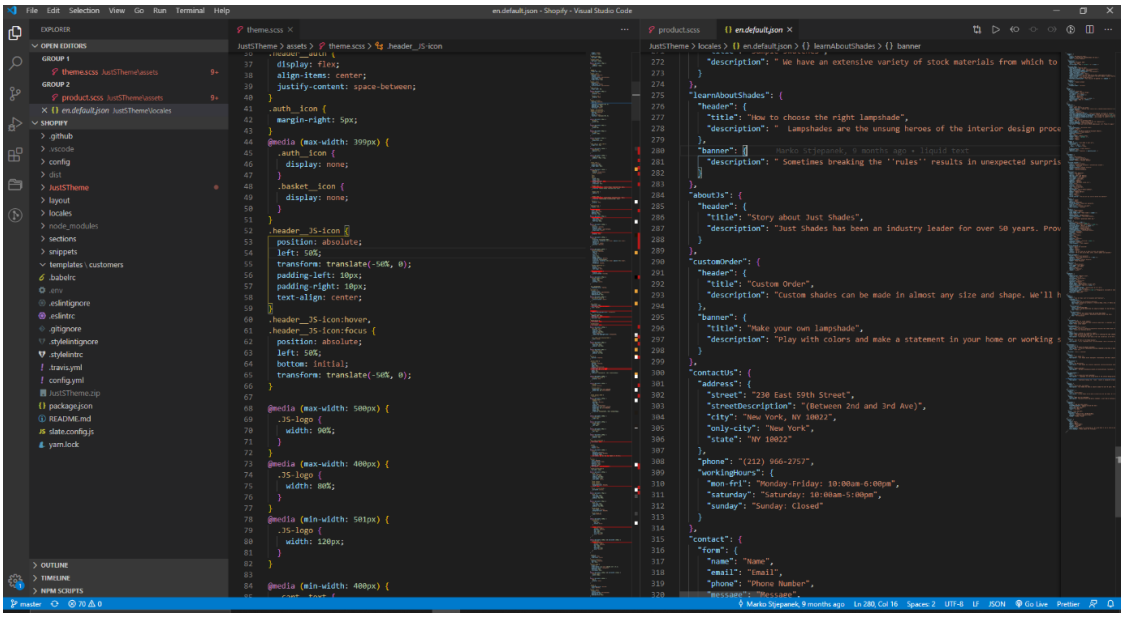


Sl. 4.3. *Arhitektura React Native programskog okvira [42]*

Neke od aplikacija koje su potpuno ili djelomično napisane u React Native-u su Instagram, Pinterest, Airbnb, Facebook, Uber...

4.1.4. Uređivač koda Visual Studio Code

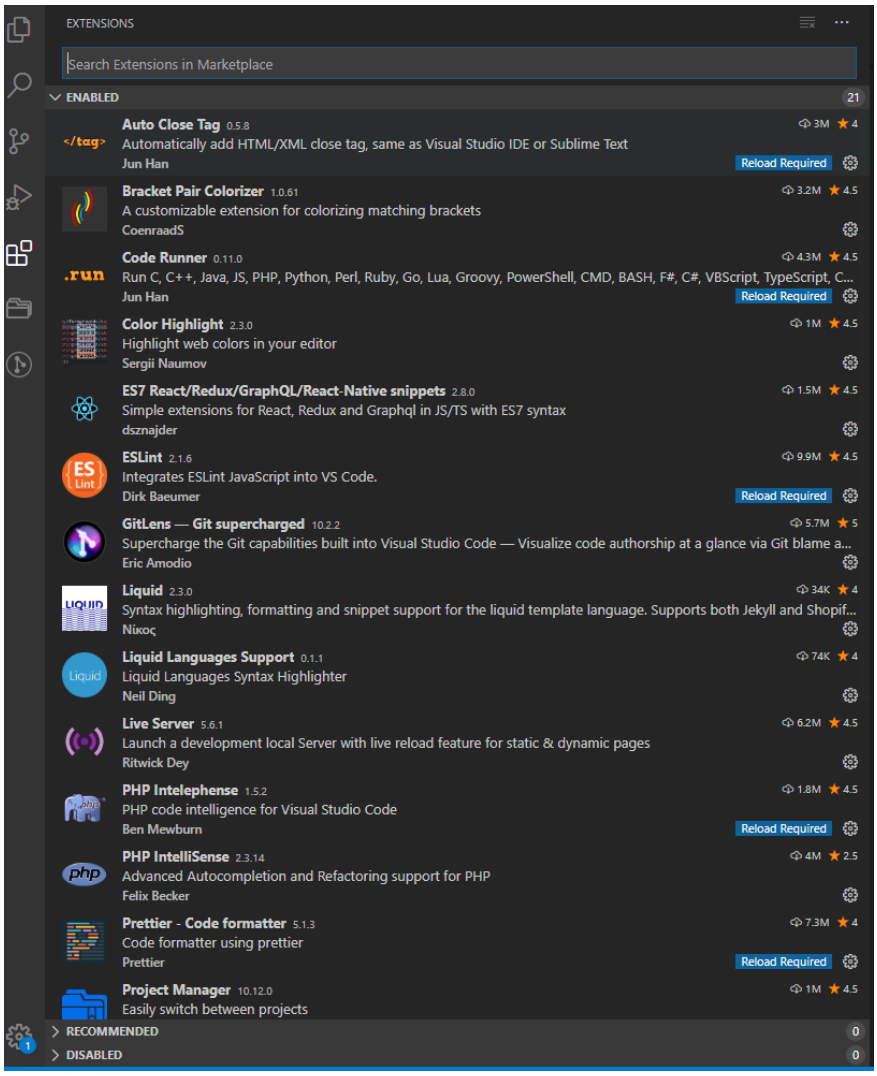
Visual Studio Code ili skraćeno VS Code, besplatni je uređivač koda sa slike 4.4 koji je izradio Microsoft 2005. godine za Windows, Linux i macOS. VS Code ima ugrađenu podršku za JavaScript, TypeScript i Node.js. Isto tako ima mogućnost dodavanja bogatog ekosistema proširenja za druge jezike kao što su C++, C#, Java, Python, PHP, GO...



Sl. 4.4. *Projekt otvoren u Visual Studio Code uređivaču*

Najvažnije funkcionalnosti VS Code-a su IntelliSense, debugiranje, integrirano Git sučelje te ekstenzije [43]. IntelliSense omogućuje čitljiviju sintaksu i automatsko pametno dovršavanje koda bazirano na prijašnjim varijablama, tipovima varijabli, funkcijama i modulima. Omogućuje jednostavnije debugiranje koda direktno iz uređivača s kontrolnim točkama i interaktivnom konzolom. VS Code ima mogućnost rad s Gitom i s pružateljima usluga verzioniranja izravno iz uređivača gdje se mogu vidjeti razlike u kodu, *commitat*, *pushat*, *pullat*....

Ekstenzije su jedna od najbitnijih funkcionalnosti koje omogućuju brži i efikasniji rad programera. Ekstenzije se mogu pronaći na Visual Studio Code Marketplace-u i potrebno ih je instalirati. Većina ekstenzija koje se mogu pronaći na Marketplace-u su besplatne. Neke od ekstenzija koje su korištene u izradi aplikacije ovog rada su prikazane na slici 4.5.



Sl. 4.5. *Visual Studio Code ekstenzije*

U Stack Overflow anketi 2019. godine za programere, Visual Studio Code rangiran je kao najpopularniji alat za razvojno okruženje, a 50.7% ispitanika tvrdi da ga koristi [44].

4.1.5. Platforma Firebase i skladištenje podataka pri nedostatku mrežne veze

Firebase je platforma za kreiranje mobilnih i web aplikacija. Izvorno je bila neovisna tvrtka osnovana 2011. godine, a 2014. godine Google je kupio platformu. Firebase je zapravo skup alata za izgradnju, poboljšanje i razvoj aplikacije, a alati koje daje pokrivaju velik dio usluga koje bi programeri obično morali sami napraviti, ali zapravo ne žele, jer se žele usredotočiti na sami proizvod [45].

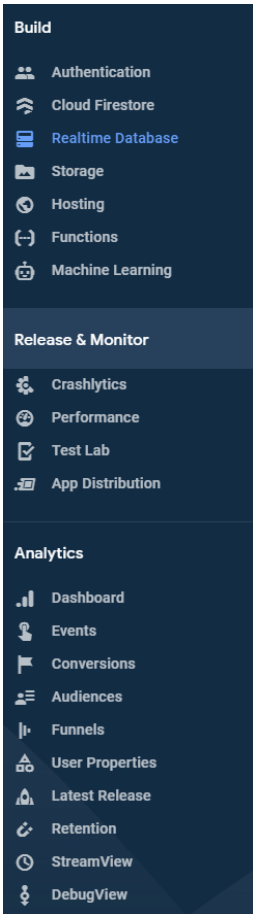
Neke od mogućnosti sa slike 4.6 koje Firebase pruža su [46]:

- *Cloud Firestore* – Spremanje i sinkronizacija podataka između korisnika i uređaja na globalnoj razini pomoću NoSQL baze koja se nalazi u oblaku. Omogućuje sinkronizaciju u stvarnom vremenu. Njegova integracija s drugim Firebase proizvodima omogućuje izradu aplikacija bez poslužitelja
- Funkcije u oblaku – funkcije koje nude mogućnost proširivanja aplikacije s prilagođenom poslužiteljskom stranom bez potreba za vlastitim poslužiteljem. Funkcije se mogu pokrenuti događajima koje su emitirane od Firebase proizvoda, Google oblaka ili treće strane.
- Autentikacija korisnika – Upravljanje korisnicima na jednostavan i siguran način. Nudi više načina za autentikaciju, uključujući e-poštu i lozinku, davatelje usluga trećih strana poput Google-a ili Facebooka.
- Usluge poslužitelja – pojednostavljene usluge poslužitelja napravljene posebno za moderne web aplikacije. Nakon što su datoteke prenesene, automatski se stavljaju na mrežu za dostavljanje podataka i daje im se besplatni SSL certifikat kako bi korisnici stekli sigurno i pouzdano iskustvo s malim kašnjenjem, neovisno o lokaciji gdje se korisnik nalazi.
- Pohrana u oblaku – pohrana i dijeljenje sadržaja koju stvori korisnik, kao što su slike, audio i video snimke. Firebase-ov paket za razvoj programa pohrane u oblaku dodaje dodatnu razinu sigurnosti u datoteke za prijenos i preuzimanje Firebase aplikacija, bez obzira na kvalitetu mreže.
- Baza podataka u stvarnom vremenu – Prva baza podataka u stvarnom vremenu koju je Firebase imao. Pomoću nje je moguće dobiti učinkovita rješenja s malim kašnjenjem za mobilne i web aplikacije kojima je potrebna sinkronizacija stanja aplikacije u stvarnom

vremenu. Ipak, Firebase preporučuje *Cloud Firestore* umjesto baze podataka u stvarnom vremenu za većinu programera.

- Firebase strojno učenje – strojno učenje koje pomaže u raspoređivanju optimiziranih prilagođenih modela za lakše korištenje u aplikaciji. Isto tako može se koristiti *AutoML Vision Edge* za treniranje vlastitih slika klasifikacije modela.

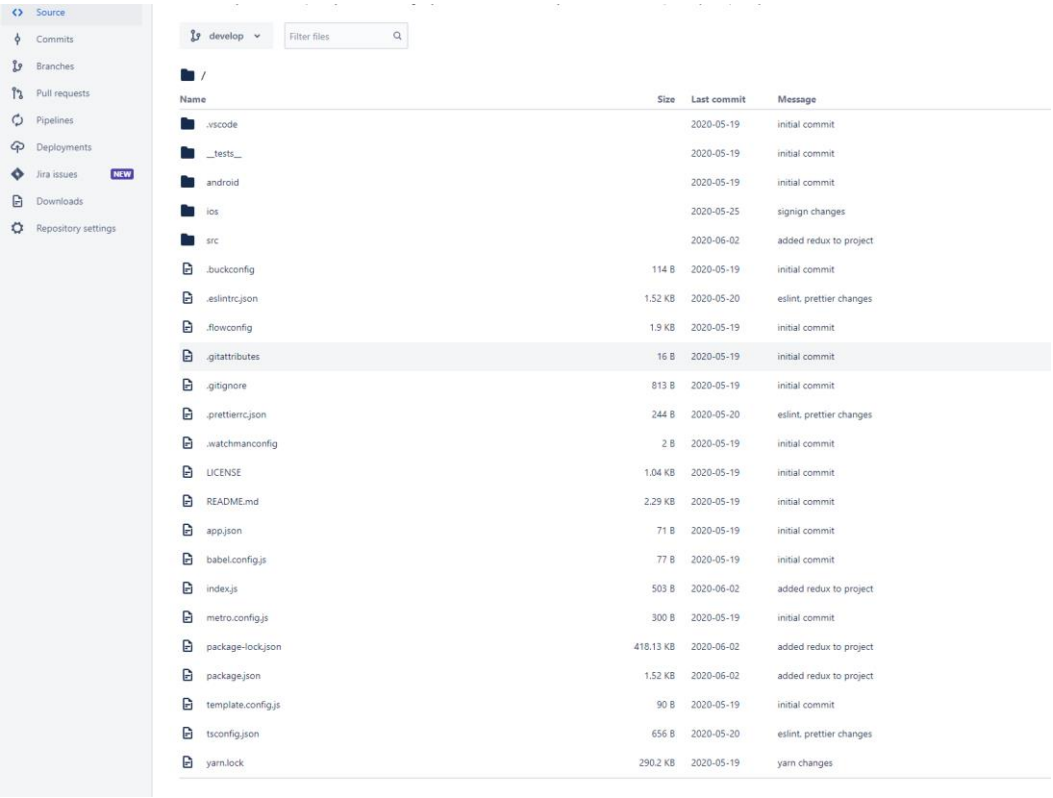
Kao što je rečeno u poglavlju s *Websocketima,* u ovom radu koristit će se *Websocketi* za sinkronizaciju u stvarnom vremenu s Firestore bazom. Isto tako kako bi se omogućilo skladištenje podataka u nedostatku mrežne veze baza podataka Cloud Firestore omogućuje podršku izvan mreže. Ova značajka sprema kopiju podataka koje aplikacija aktivno koristi, tako da aplikacija može pristupiti podacima kada je uređaj izvan mreže [47]. Aplikacija može zapisivati, čitati, „slušati i filtrirati predmemorirane podatke. Kad se uređaj vrati na mrežu baza podataka sinkronizira sve lokalne promjene koje je izvršila aplikacija u nedostatku mreže. Da bi se koristila ova značajka Firebase-a nije potrebno podešavati postavke jer je takav način rada zadan na početku.



Sl. 4.6. *Različite Firebase mogućnosti*

4.1.6. Sustav za upravljanje programskim kodom Git, model grananja GitFlow i alat za dijeljenje programskog koda Bitbucket

Git je besplatni i distribuirani sustav otvorenog koda dizajniran za verzioniranje koda koji pomaže programerskom timu za lakše upravljanje izmjenama koda tijekom vremena. Kako bi se koristio Git potrebno je napraviti lokalni Git repozitorij. Kada je cjelina koda gotova potrebno je taj kod commitat tj. dodati ga u lokalni repozitorij. Kako taj kod ne bi ostao samo u lokalnom repozitoriju potrebno je na nekom od davatelja usluga za verzioniranje koda napraviti udaljeni repozitorij te povezati lokalni repozitorij s udaljenim. Najpoznatiji hosting davatelji usluga za verzioniranje koda su Github, Bitbucket i Gitlab. U ovom radu koristi se Bitbucketov privatni repozitorij što se vidi na slici 4.7.



Sl. 4.7. *Bitbucket repozitorij za aplikaciju ovog rada*

Kako bi se poslao kod na udaljeni repozitorij potrebno je izvršiti push akciju. Akcijama pull i fetch dohvaćaju se promjene s udaljenog repozitorija i spremaju u lokalni repozitorij. Kako bi više programera moglo raditi na istom projektu koriste se grane, gdje svaki programer može raditi na svojoj grani, a kada su dijelovi na kojima programeri rade gotovi, spajaju se u cjelinu.

Postoje različiti pristupi koje Git koristi za spajanje grana, u ovom radu koristi se Git-flow pristup. Temelji se na Git rebase-u koji je strogog pristupa i potrebno je vrijeme navikavanja tome načinu rada. Glavni ciljevi Git-flow pristupa su [48]:

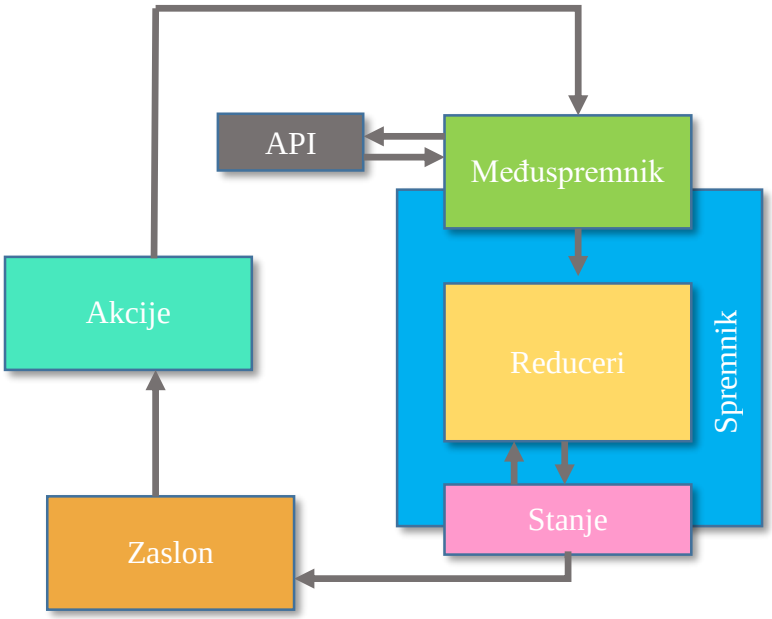
- Preglednija Git povijest
- Manje konflikata
- Nužnost pregledavanja koda
- Povećanje stabilnosti grana

Razlika između merge akcije i Git-flow-a je što merge spoji sve promjene odjednom, što dovodi do ogromne količine potencijalno konfliktnih datoteka, težeg razumijevanja tko je napravio promjene i zašto je došlo do sukoba. Git-flow automatski spaja promjene ako nije došlo do konflikta, a ako je došlo do konflikta, svaki commit rješava pojedinačno što pojednostavljuje spajanje različitog koda.

4.1.7. **Spremnik stanja Redux**

Redux je predvidljivi spremnik stanja dizajniran da pomogne u pisanju JavaScript aplikacija koje dijele različite podatke u klijentskom i poslužiteljskom okruženju [49]. Iako se uglavnom koristi kao alat za upravljanje stanjem s Reactom i React Native-om, može se koristiti s bilo kojim drugim JavaScript okvirom ili bibliotekom. Pomoću Redux-a se stanje aplikacije čuva u spremniku u kojem svaka komponenta može pristupiti te uzeti podatke koji su joj potrebni kao što je prikazano arhitekturom na slici 4.8. Prednosti koje Redux pruža su [50]:

- Predvidljivost – pomaže u pisanju aplikacija koje se dosljedno ponašaju, rade u različitim okruženjima (klijentska strana, poslužiteljska i nativna) i lako ih je testirati
- Centraliziranost - centralizacija stanja i logike aplikacije omogućuje dijeljenje podataka i postojanost stanja
- Redux DevTools – alat koji olakšava praćenje kada, gdje, zašto i kako se promijenilo stanje aplikacije. Reduxova arhitektura omogućuje bilježenje promjena, otklanjanje pogrešaka i slanje cjelovitih izvještaja o pogreškama
- Fleksibilnost – radi s bilo kojim slojem korisničkog sučelja i ima veliki ekosustav dodataka koji odgovara različitim potrebama

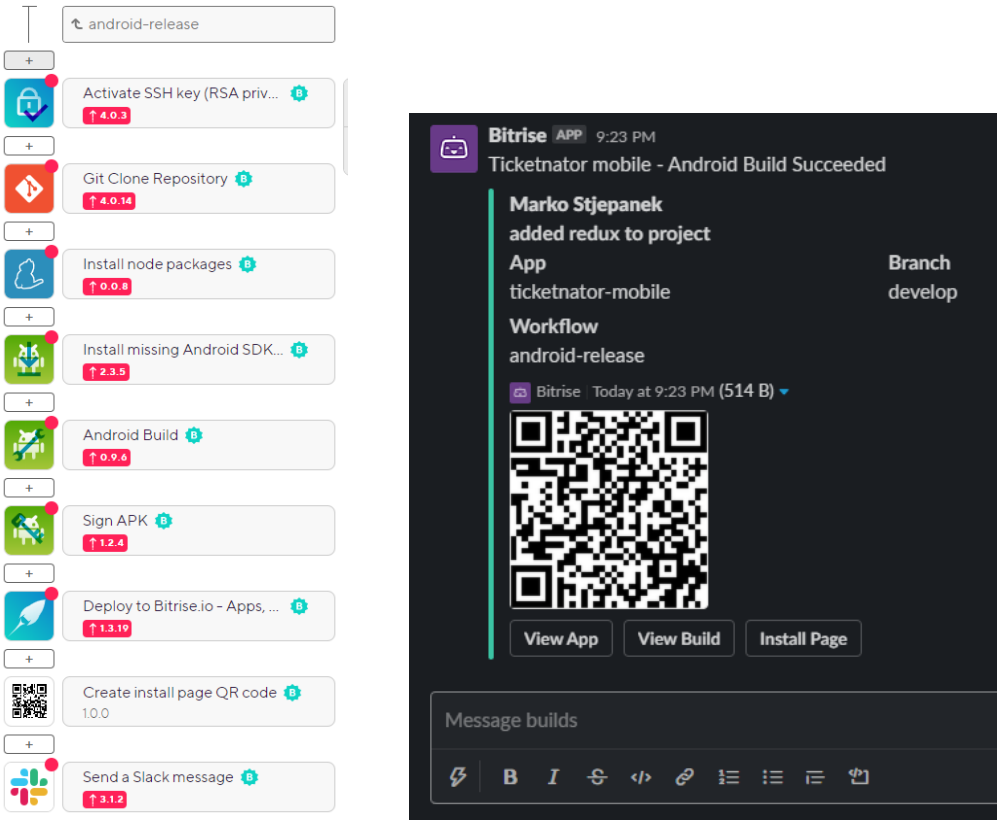


Sl. 4.8. *Redux arhitektura za sloj korisničkog sučelja prema [51]*

4.1.8. Sustav kontinuirane mobilne integracije i isporuke Bitrise

Bitrise je kontinuirana integracija i isporuka (CI / CD) platforme kao usluge s naglaskom na razvoj mobilnih aplikacija (iOS, Android, React Native...) [52]. To je zbirka alata i usluga koji pomažu pri razvoju i automatizaciji programske podrške. Automatizacija obuhvaća cjelokupni proces automatiziranja razvoja, testiranja i implementacije programske podrške. Time se štedi na vremenu i smanjuju se moguće pogreške. Kako bi se koristio Bitrise potrebno je aplikaciju postaviti na udaljeni repozitorij i postaviti zadane licence i ključeve za Android i iOS sustave. Moguće je postaviti okidače, tj. vrijeme kada će Bitrise napraviti novi test i pripremit aplikaciju za korištenje. Najčešći okidač je na akciju push na udaljeni repozitorij.

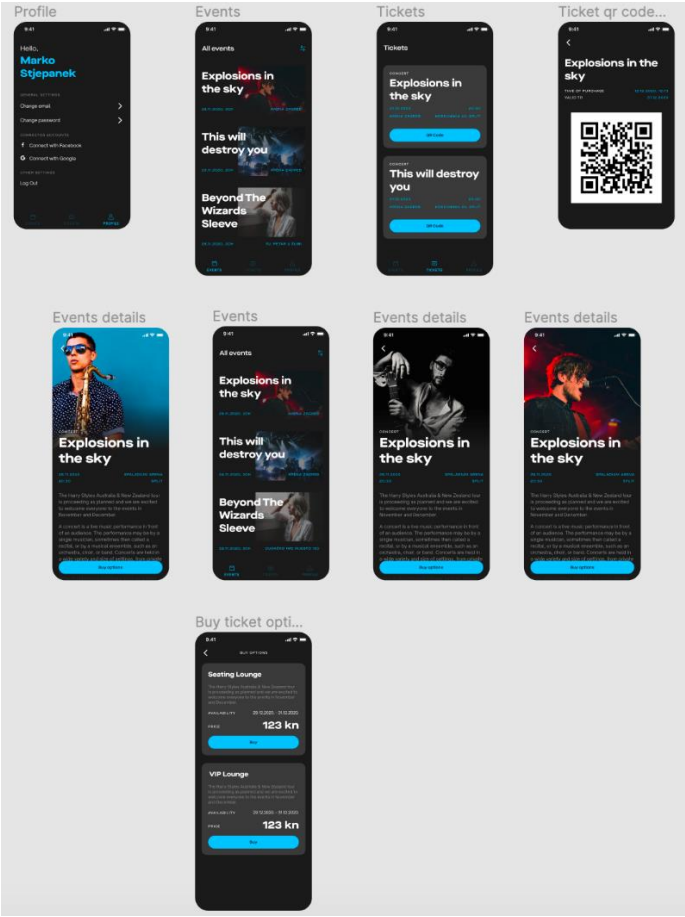
Na slici 4.9 je postupak što Bitrise radi prilikom izvršavanja i kreiranja aplikacije spremne za korištenje. Isto tako na slici 4.9 vidi se Slack prozor, gdje je poslan QR kod za instalaciju aplikacije nakon uspješnog builda na Bitrise-u.



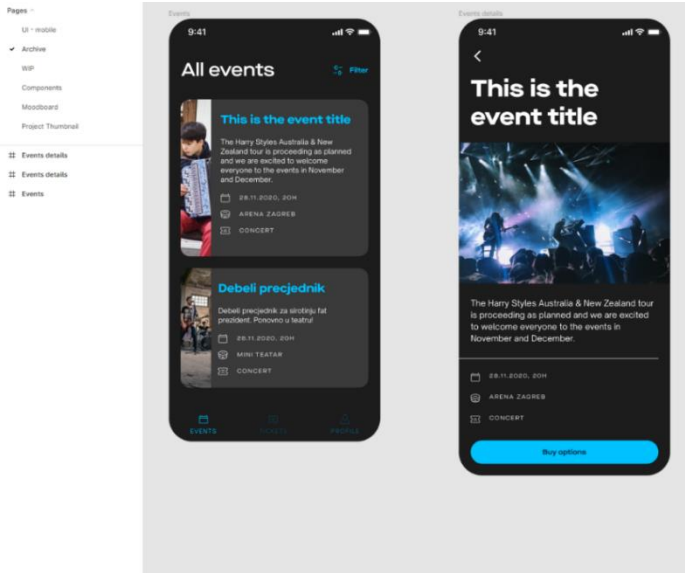
Sl. 4.9. *Tijek rada Bitrise-a i Slack kanal s QR kodom za instalaciju aplikacije*

4.1.9. Alat za uređivanje i izradu prototipova za digitalne projekte Figma

Figma je online alat za dizajn i izradu prototipa za digitalne projekte. Napravljena je tako da korisnici mogu surađivati na projektima i raditi s bilo kojeg mjesta [53]. Figma omogućuje suradnju u stvarnom vremenu. Više članova tima može se prijaviti odjednom u dizajn te istovremeno mijenjati dizajn. S obzirom na to da je Figma online alat, nema potrebe za brigom o gubitku projekta. Omogućuje lakšu komunikaciju s klijentom, gdje klijent može davati prijedloge gledajući u dizajn, a programeri to onda jednostavno implementiraju u projekt. Pomoću Figme dizajnirani su glavni zasloni za kasniju lakšu izradu mobilne aplikacije. Na slici 4.10 su zasloni prema kojima će biti izrađena mobilna aplikacija Ticketnator. Na slici 4.11 su prikazani različiti dizajni korisničkog sučelja kako bi se došlo do željenog izgleda.



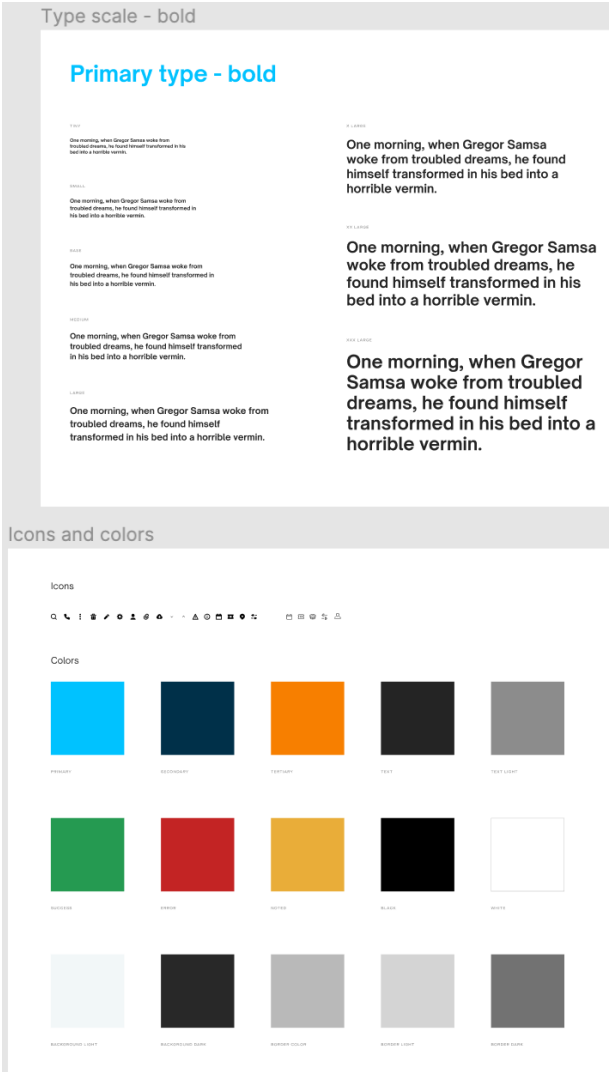
Sl. 4.10. *Dizajnirani zasloni u Figmi*



Sl. 4.11. *Arhivirani zasloni koji se neće koristiti*

Od samog početka projekta važno je postaviti osnovne elemente aplikacije kao na slici 4.12 koji će se koristiti kroz cijelu aplikaciju:

- Tipografska skala
- Različite vrste gumbova
- Forme i polja
- Ikone
- Paginacija
- Validacijske poruke
- Boje



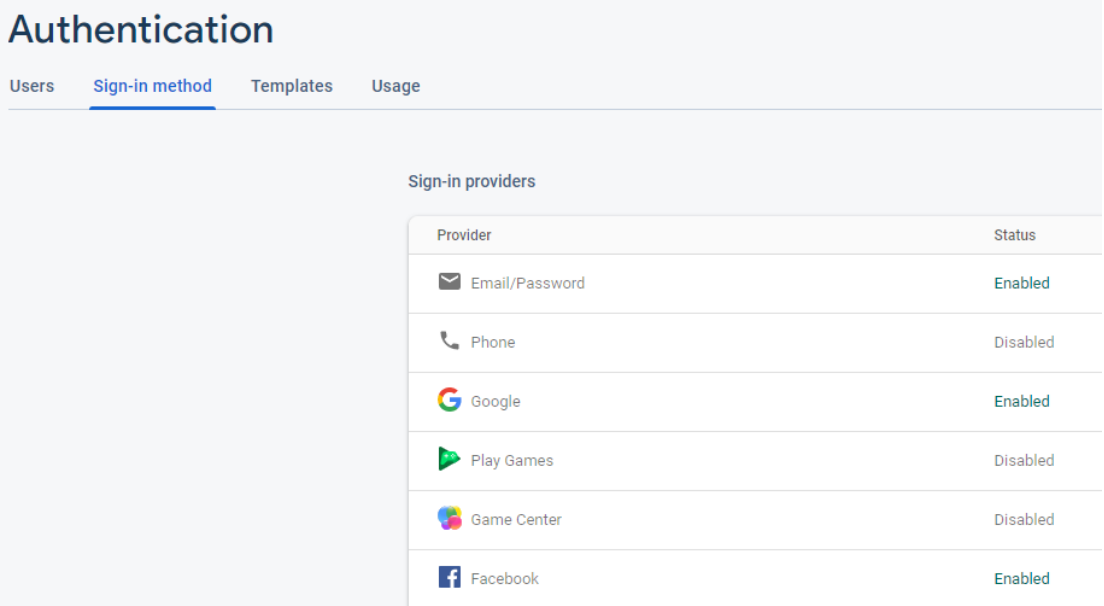
Sl. 4.12. *Tipografska skala i boje koje će se koristit kroz aplikaciju*

4.2. Prikaz programskog rješenja na strani poslužitelja

Kao što je ranije rečeno, u mobilnoj aplikaciji Ticketnator, za bazu podataka koristit će se Firebase s funkcionalnostima koje posjeduje.

4.2.1. **Kreiranje korisničkog računa**

Za kreiranje korisničkog računa u aplikaciji, potrebno je prvo postaviti mogućnosti prijave i kreiranja računa na Firebase-u. Sa slike 4.13 vidljivo je da je omogućena prijava i korištenje aplikacije putem korisničkog računa s e-poštom i lozinkom te, Google i Facebook prijavom.



Sl. 4.13 . *Različite opcije autentikacije na Firebase-u*

Za kreiranje korisničkog računa putem e-pošte i lozinke kao na slici 4.14 koristit će se Firebase-ova ugrađena funkcija *createUserWithEmailAndPassword.*

```
/** Returns user credential if successful, error string if failed. */
async function registerWithEmailAndPasswordAsync(
  email: string,
  password: string,
  fullName: string,
  phone: string
): Promise<RNFirebase.UserCredential | string> {
  return auth()
    .createUserWithEmailAndPassword(email, password)
    .then(async credentials => {
      const userId = credentials.user?.uid;
      if (!userId) {
        return 'Signup operation failed, user could not be created';
      }
      await createAuthUserWithFirestoreProfile(userId, email, fullName, phone);
      return credentials
    })
    .catch((error) => error.message);
}
```

Sl. 4.14 . *Funkcija za kreiranje korisničkog računa s e-poštom i lozinkom*

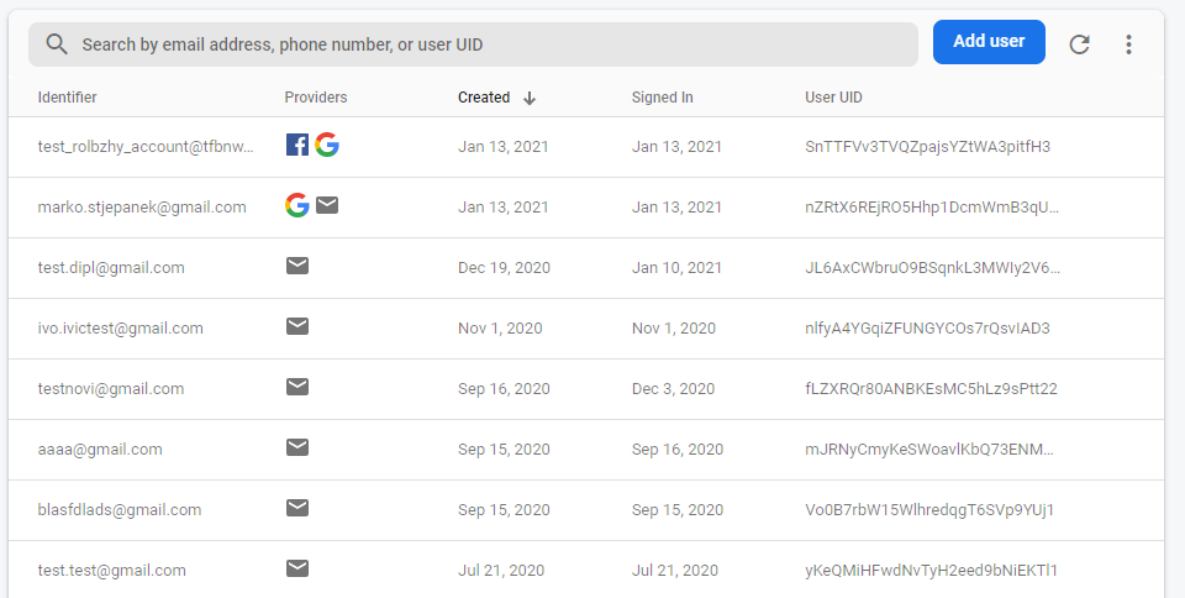
Kod izrade korisničkog računa kreirat će se zapis u bazi s podatcima o korisničkom profilu. Za to će se koristiti funkcija createAuthUserWithFireStoreProfile na slici 4.15 koja će primati

parametre: ID korisnika, e-poštu, puno ime i prezime i broj telefona. U bazi će se spremiti javno dostupni podaci korisnika i privatni podaci u različitim Firebase zbirkama.

```
async function createAuthUserWithFirestoreProfile(        You, 4 months ago
  userId: string,
  email: string,
  fullName: string,
  phone: string | null

) {

  // create public profile
  await profileService.update(
    new PublicProfile({ email, fullName, phone }),
    userId,
  );
  // create private profile
  await profileService
    .getDocRef(userId)
    .withSubcollection<PrivateProfile>(Subcollection.PrivateProfile)
    .add(new PrivateProfile(), 'profile');

  return;
}
```

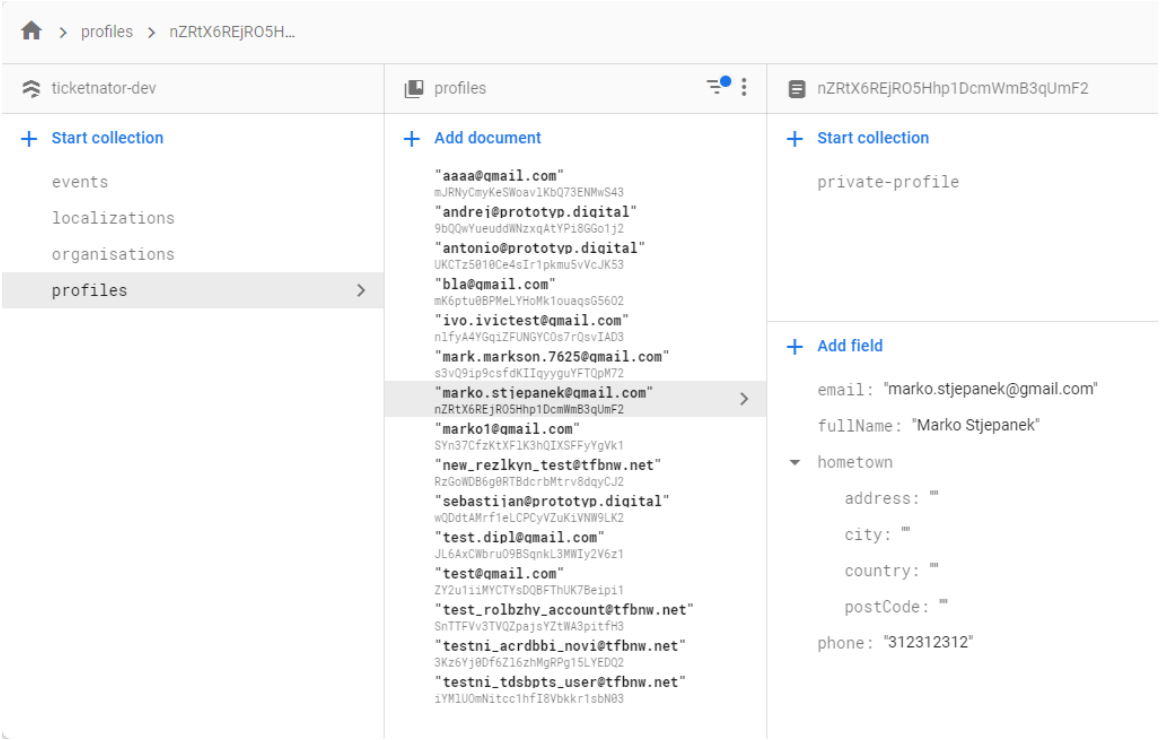
Sl. 4.15 . *Funkcija za spremanje korisničkog računa u bazu podataka*

U navigaciji Firebase-a poput prikaza na slici 4.16 nalazi se “Authentication“ tj. sučelje za upravljanje autentikacijom korisnika gdje je moguće dodavanje, brisanje i uređivanje korisnika.



Sl. 4.16 . *Firebase sučelje za upravljanje autentikacijom korisnicima aplikacije*

U zbirku “profiles“ sprema se novi kreirani korisnički račun s unesenim javnim i privatnim podacima kao na slici 4.17.



Sl. 4.17 . *Firebase baza podataka*

Za kreiranje korisničkog računa pomoću Google računa potrebno je na web stranici Google API Console registrirati aplikaciju i dobiti webClientId kako bi aplikacija dobila tražene ovlasti. Koristit će se paket @react-native-community/google-signin za lakše kreiranje računa. Funkcija na slici 4.18 prijavit će korisnika s Google računom te ako je korisnik prvi put prijavljen, ima e-poštu i korisničko ime, podaci o korisničkom računu će se spremiti u bazu podataka. Ako dođe do pogreške ili neki uvjet nije zadovoljen, funkcija će vratiti određenu poruku.

```
async function googleLogin() {
  // Configures the library for login. MUST be called before attempting login
  GoogleSignin.configure({
    webClientId: '843720144851-[redacted]d.apps.googleusercontent.com',
  });

  // Get the users ID token
  const { idToken } = await GoogleSignin.signIn();

  // Create a Google credential with the token
  const googleCredential = auth.GoogleAuthProvider.credential(idToken);

  // Sign-in the user with the credential
  return auth().signInWithCredential(googleCredential)
    .then(userCredential => {
    const userId = userCredential.user?.uid;
    if (!userId) {
      return 'Signup operation failed, user could not be created';
    }
    if (userCredential.additionalUserInfo?.isNewUser && userCredential.user.email
      && userCredential.user.displayName) {      You, seconds ago • Uncommitted changes
      AuthService().createAuthUserWithFirestoreProfile(userId, userCredential.user.email,
          userCredential.user.displayName, userCredential.user.phoneNumber)
    }
    return userCredential
  })
  .catch((error)=> {
    dispatch(AuthActions.error(error.message));
  })
}
```

Sl. 4.18 . *Funkcija za kreiranje korisničkog računa s Google računom*

Za kreiranje korisničkog računa pomoću Facebook računa potrebno je na web stranici Facebook for Developers registrirati aplikaciju, postaviti određene parametre i uloge korisnika koji mogu pristupiti i testirati aplikaciju dok je u razvoju. Koristit će se paket react-native-fbsdk za lakše kreiranje računa. Funkcija za kreiranje korisničkog računa s Facebook računom sa slike 4.19 je logikom slična funkciji za kreiranje korisničkog računa s Google računom. U funkciji se pomoću tokena dobivaju potrebni Facebook podaci. Prijavljuje li se korisnik prvi put, ima korisničku e-poštu i korisničko ime, spremit će se podatci o korisničkom računu u bazu podataka. Ako dođe do pogreške ili neki uvjet nije zadovoljen funkcija će vratiti određenu poruku.

```
async function facebookLogin() {
  // Attempt login with permissions
  const result = await LoginManager.logInWithPermissions([
    'public_profile',
    'email',
  ]);

  if (result.isCancelled) {
    throw 'User cancelled the login process';
  }

  // Once signed in, get the users AccesToken
  const data = await AccessToken.getCurrentAccessToken();

  if (!data) {
    throw 'Something went wrong obtaining access token';
  }

  // Create a Firebase credential with the AccessToken
  const facebookCredential = auth.FacebookAuthProvider.credential(
    data.accessToken,
  );

  // Sign-in the user with the credential and if the user is first signup, create user Firestore profile
  auth().signInWithCredential(facebookCredential)
    .then(userCredential => {
      const userId = userCredential.user?.uid;
      if (!userId) {
        return 'Signup operation failed, user could not be created';
      }
      if (userCredential.additionalUserInfo?.isNewUser && userCredential.user.email
        && userCredential.user.displayName) {      You, seconds ago • Uncommitted changes
        AuthService().createAuthUserWithFirestoreProfile(userId, userCredential.user.email,
          userCredential.user.displayName, userCredential.user.phoneNumber)
      }
      return userCredential
    })
    .catch((error)=> {
      dispatch(AuthActions.error(error.message));
    })
  return
}
```

Sl. 4.19 . *Funkcija za kreiranje korisničkog računa s Facebook računom*

4.2.2. Povezivanje korisničkog računa s Google i Facebook računom

Kako bi se korisnički račun s e-poštom i lozinkom povezao s Google računom ili Facebook računom koristit će se Firebase ugrađena funkcija linkWithCredential. S obzirom na to da je aplikacija već registrirana na Google API Console-i Facebook for Developers moguće je povezati korisnički račun, samo je potrebno pozvati funkcije sa slike 4.20 za povezivanje korisničkog računa s Google računom i funkciju sa slike 4.21 za povezivanje korisničkog računa s Facebook računom. U funkcijama će se predati Facebook ili Google token pomoću kojih će se dobiti traženi podaci koji su potrebni za funkciju linkWithCredential. Dođe li do pogreške, funkcija će vratiti određenu poruku za pogrešku do koje je došlo.

```
const connectGoogleProvider = () => async (dispatch: AppDispatch) => {
  dispatch(AuthActions.googleConnectChange());

  GoogleSignin.configure({
    webClientId: '843720144851-[redacted].apps.googleusercontent.com',
  });

  // Get the users ID token
  GoogleSignin.signIn()
    .then((result) => {
      if (!result) {
        return Promise.reject(new Error('The user cancelled the request'))
      }
      // Retrieve the access token
      return result
    })
    .then((data) => {
      // Create a new Firebase credential with the token
      if (!data) {
        return
      }
      const credential = auth.GoogleAuthProvider.credential(data.idToken)

      if (!auth().currentUser) {
        return
      }

      // Link using the credential
      return auth().currentUser?.linkWithCredential(credential)
    })
    .then(() => dispatch(AuthActions.googleConnectSuccess()))
    .catch((error) => {      You, 2 months ago • Connect facebook,google provider with acc in th
      const { message } = error
      dispatch(AuthActions.googleConnectError(message));
    })
};
```

Sl. 4.20. *Funkcija za povezivanje korisničkog računa s e-poštom i lozinkom s Google računom*

```
const connectFacebookProvider = () => async (dispatch: AppDispatch) => {
  dispatch(AuthActions.facebookConnectChange());

  LoginManager.logInWithPermissions(['public_profile', 'email'])
    .then((result) => {
      if (result.isCancelled) {
        return Promise.reject(new Error('The user cancelled the request'))
      }
      // Retrieve the access token
      return AccessToken.getCurrentAccessToken()
    })
    .then((data) => {
      // Create a new Firebase credential with the token
      if (!data) {
        return
      }      You, 2 months ago • Connect facebook,google provider with acc in th
      const credential = auth.FacebookAuthProvider.credential(data.accessToken)

      if (!auth().currentUser) {
        return
      }

      // Link using the credential
      return auth().currentUser?.linkWithCredential(credential)
    })
    .then(() => dispatch(AuthActions.facebookConnectSuccess()))
    .catch((error) => {
      const { message } = error
      dispatch(AuthActions.facebookConnectError(message));
    })
};
```

Sl. 4.21 . *Funkcija za povezivanje korisničkog računa s e-poštom i lozinkom s Facebook računom*

4.2.3. **Izmjena lozinke korisničkog računa**

Kako bi se izmijenila lozinka korisničkog računa napravljena je funkcija updatePassword sa slike 4.22. Funkcija prima kao parametar novu lozinku koja se želi postaviti te dohvaća trenutnog korisnika. Ako ne postoji trenutni korisnik ili nema nove lozinke funkcija se završava. Nakon provjere uvjeta, ako su uvjeti zadovoljeni mijenja se lozinka korisničkog računa te se vraća određena poruka ako je uspješno izmijenjena lozinka ili poruka s pogreškom u ovisnosti o kojoj se pogrešci radi.

```
const updatePassword = (newPassword: string) => async (dispatch: AppDispatch) => {
  const user = auth().currentUser;

  dispatch(AuthActions.changeInfoChange());

  if (!user || !newPassword) {      You, 2 months ago • Change firebase email, passwo
    return;
  }

  user
    .updatePassword(newPassword)
    .then(() => {
      dispatch(AuthActions.changeInfoSuccessPassword());
    })
    .catch(error => {
      const { message } = error
      dispatch(AuthActions.changeInfoError(message));
    });
};
```

Sl. 4.22. *Funkcija za izmjenu lozinke korisničkog računa*

4.2.4. **Dohvaćanje korisničkog profila**

Kod kreiranja korisničkog računa prikazano je da se u bazi kreiraju javno dostupni i privatni podaci korisnika. Kako bi se dohvatili ti podaci potrebno je koristiti funkciju za javno dostupne podatke i funkciju za privatne podatke korisnika jer se ti podaci nalaze u različitim zbirkama tj. podzbirkama. Za dohvaćanje javnih podataka korisnika koristit će se funkcija getPublicProfile sa slike 4.23. Prije korištenja funkcije stavljeno je da je konstanta profileService zbirka *Profiles* u bazi Firebase, koja će biti potrebna u funkciji za dohvaćanje profila. Funkciji se predaje ID korisničkog računa gdje će *listener* „slušati“ i pretražiti u zbirci *Profiles* ID korisničkog računa. Ako su uspješno nađeni javni podaci profila proslijedit će se za daljnje korištenje. U slučaju da dođe do pogreške, poslat će se poruka o utvrđenoj pogrešci.

```
const profileService = FirestoreService<PublicProfile>(Collection.Profiles);

const getPublicProfile = (profileId: string) => (
  dispatch: AppDispatch,
) => {

  function onSuccess(profile?: PublicProfile) {
    dispatch(ProfileActions.publicSuccessWithPayload(profile));
  }

  function onError(error?: string) {
    dispatch(ProfileActions.publicError(error));
  }

  dispatch(ProfileActions.publicLoading());

  const listener = profileService
    .findAndListen(
      profileId,
      onSuccess,
      onError
    );

  dispatch(FirestoreListenerActions.setListener('public-profile', listener));
};
```

Sl. 4.23 . *Funkcija za dohvaćanje javnih podataka korisničkog računa*

Za dohvaćanje privatnih podataka korisnika koristit će se funkcija getPrivateProfile sa slike 4.24. Funkciji se predaje ID korisničkog računa gdje će se u zbirci *Profiles* dohvatiti dokument s ID-em korisničkog računa te će se u tom dokumentu u podzbirci *PrivateProfile* „slušati“ izmjene. Ako su prethodni koraci uspješno obavljeni, dobit će se privatni podaci korisnika te će se proslijediti za daljnje korištenje. U slučaju da dođe do pogreške, poslat će se poruka u ovisnosti o pogrešci.

```
const getPrivateProfile = (profileId: string) => (
  dispatch: AppDispatch,
) => {

  function onSuccess(profile?: PrivateProfile) {
    dispatch(ProfileActions.privateSuccessWithPayload(profile));
  }

  function onError(error?: string) {
    dispatch(ProfileActions.privateError(error));
  }

  dispatch(ProfileActions.privateLoading());

  const listener = profileService
    .getDocRef(profileId)
    .withSubcollection(Subcollection.PrivateProfile)
    .findAndListen(
      'profile',
      onSuccess,
      onError,
    );

  dispatch(FirestoreListenerActions.setListener('private-profile', listener));
};
```

Sl. 4.24 . *Funkcija za dohvaćanje privatnih podataka korisničkog računa*

4.2.5. **Dohvaćanje svih događaja iz baze podataka**

Za prikazivanje svih događaja u aplikaciji potrebno je prvo dohvatiti sve događaje iz baze podataka. Za to se koristi funkcija getEventsAsync sa slike 4.25. Prije korištenja funkcije stavljeno je da je konstanta eventCollectionService zbirka *Events* u Firebase bazi koja će biti potrebna u funkciji za dohvaćanje događaja. U funkciji se provjerava postoji li već *listener* koji „sluša“ izmjene u zbirci *Events* te ako postoji funkcija se završava. Ako ne postoji kreira se *listener* koji će filtrirati i „slušati“ događaje u zbirci. Ako je uspješno povezano dobit će se događaji za rad koji slijedi, a ako je došlo do pogreške poslat će se poruka ovisno o određenoj pogrešci.

```
const eventCollectionService = FirestoreService<Event>(Collection.Events);

const getEventsAsync = () => async (
  dispatch: AppDispatch,
  getState: () => AppState,
) => {
  const eventListener = getState().listeners['events'];
  if (eventListener) {
    return;
  }

  dispatch(EventActions.addRequest());

  const successFunction = (data: Event[]) => {
   dispatch(EventActions.addEvents(data));
  };

  const errorFunction = (error: string) =>
    dispatch(EventActions.addError(error));

  const listener = eventCollectionService
    .filterAndListen(successFunction, errorFunction);
  dispatch(FirestoreListenerActions.setListener('events', listener));

  return;
};
```

Sl. 4.25 . *Funkcija za dohvaćanje svih događaja*

4.2.6. **Dohvaćanje vrsta ulaznica događaja za kupovinu**

Za dohvaćanje vrsta ulaznica za kupovinu nekog događaja koristit će se funkcija getTicketTypesAsync sa slike 4.26. Prije korištenja funkcije stavljeno je da je konstanta eventCollectionService zbirka *Events* u Firebase bazi koja bit će potrebna u funkciji za dohvaćanje događaja. Funkciji se predaje ID događaja te ako se ne preda funkcija završava. U zbirci *Events* traži se dokument s ID događaja te se u tom dokumentu u podzbirci *TicketTypes* „slušaju“ izmjene. Ako su prethodni koraci uspješno obavljeni, dobit će se vrste ulaznica

događaja te se proslijeđuju za daljnje korištenje. U slučaju da dođe do pogreške, poslat će se poruka ovisno o pogrešci.

```
const eventCollectionService = FirestoreService<Event>(Collection.Events);

const getTicketTypesAsync = (idEvent?: string) => async (
  dispatch: AppDispatch,
) => {

  if (!idEvent) {
    return;
  }

  dispatch(TicketTypesActions.addRequest());

  const successFunction = (data: TicketType[]) => {
    dispatch(TicketTypesActions.addEventDetails(data));
  };

  const errorFunction = (error: string) =>
    dispatch(TicketTypesActions.addError(error));

  const listener = eventCollectionService
    .getDocRef(idEvent)
    .withSubcollection(Subcollection.TicketTypes)
    .filterAndListen(successFunction, errorFunction)
  dispatch(FirestoreListenerActions.setListener('ticket-types', listener));

  return;
};
```

Sl. 4.26 . *Funkcija za dohvaćanje vrsta ulaznica događaja za kupovinu*

4.2.7. **Kupovina ulaznice za određeni događaj**

Za kupovinu ulaznice za određeni događaj koristit će se funkcija buyTicket sa slike 4.27 i 4.28 Prije korištenja funkcije stavljeno je da je konstanta orgsService zbirka *Organisations.* Funkciji se predaje vrsta ulaznice i broj ulaznica koji se želi kupiti. Ako u objektu vrsta ulaznice ne postoji ID organizatora, ID roditelja ulaznice ili korisnik nije prijavljen funkcija se završava. Zatim se postavlja konstanta ticketTypesService koja će iz zbirke *Organisations* u njezinoj podzbirci *Events* pronaći dokument s ID ulaznice te u njoj odabrati podzbirku *TicketTypes.* Pomoću konstante ticketTypesService pozvat će se funkcija incrementFieldValue gdje će se smanjit ukupan broj ulaznica za broj ulaznica koja je stavljena u ulazu funkcije. Dođe li do pogreške prilikom smanjivanja broja ulaznica vratit će se određena pogreška. Isto tako iz zbirke *Events* u tom događaju smanjit će se ukupan broj ulaznica na isti način. Zatim će se postaviti konstanta purchasedTicketService tako da ima vrijednost Podzbirke *Attendees* koja se nalazi u dokumentu koji je u podzbirci *Events,* u dokumentu ID-a organizatora koji je u zbirci *Organisations.*

```
const orgsService = FirestoreService<Organisation>(Collection.Organisations);
const buyTicket = (ticket: TicketType, amount: number, ) => async (
    dispatch: AppDispatch,
    getState: () => AppState,
  ) => {
    dispatch(TicketTypesActions.addRequest());

    const currentUser = getState().profile.publicProfile;

    if (!ticket.organisationId || !ticket.parentDocId || !currentUser) return;

    const id = uuid.v4();

    const ticketTypesService = orgsService
      .getDocRef(ticket.organisationId)
      .withSubcollection<Event>(Subcollection.Events)
      .getDocRef(ticket.parentDocId)
      .withSubcollection<TicketType>(Subcollection.TicketTypes);

    // deduct ticket amount from total quantity
    const decrementError = await ticketTypesService.incrementFieldValue(
      ticket.id,
      'quantity',
      -amount,
    );
    if (decrementError) {
      dispatch(TicketTypesActions.addError(decrementError));
      return decrementError;
    }

    // deduct ticket amount from event collection
    eventCollectionService
      .getDocRef(ticket.parentDocId)
      .withSubcollection<TicketType>(Subcollection.TicketTypes)
      .incrementFieldValue(ticket.id, 'quantity', -amount);

    const purchasedTicketService = orgsService
      .getDocRef(ticket.organisationId)
      .withSubcollection<Event>(Subcollection.Events)
      .getDocRef(ticket.parentDocId)
      .withSubcollection<Attendee>(Subcollection.Attendees);
```

Sl. 4.27 . *Funkcija za kupovinu ulaznice*

Zatim će se dodati korisnik kao polaznik događaja u podzbirku *Attendee* u dokumentu događaja. Prilikom dodavanja korisnika postavit će se podaci koji će biti prikazani kao QR kod. Podaci koji će biti upisani su: vlasnik ulaznice, broj kupljenih ulaznica i vrijeme valjanosti ulaznice. Ovisno o uspješnosti akcije vratit će se poruka o uspješnosti ili poruka o pogrešci. Kupljena ulaznica bit će spremljena u privatne podatke korisničkog računa.

```
// add user to attendee subcollection inside event
const userWithPurchasedTicket = {
  ...new Attendee({
    ...ticket,
    qrCode: `Ticket owner: ${currentUser.item?.fullName}
    \nAmount: ${amount} \nTicket valid to: ${ticket.ticketTypeValidTo}`,
    amount,
    issuedTo: {
      email: currentUser.item?.email || '',
      fullName: currentUser.item?.fullName || '',
      id: currentUser.item?.id || ''
    },
  }),
};

// Added attendee to event collection
eventCollectionService
  .getDocRef(ticket.parentDocId)
  .withSubcollection<Attendee>(Subcollection.Attendees)
  .add(userWithPurchasedTicket, id);

const attendeeError = await purchasedTicketService.add(
  userWithPurchasedTicket,
  id,
);
if (typeof (attendeeError) === 'string') {
  dispatch(TicketTypesActions.addError(attendeeError));
  return attendeeError;
}

if(typeof(attendeeError) !=='string') {
  dispatch(TicketTypesActions.successBuy())
}
// save ticket in users private profile
const privateSelectedTicketType: TicketType = {
  ...ticket,
  organisationId: ticket.organisationId,
  attendeeId: id,
};

dispatch(
  ProfileThunks.saveTicketToPrivateProfile(privateSelectedTicketType),
);

return;
};
```

Sl. 4.28 . *Nastavak funkcije za kupovinu ulaznice*

4.2.8. **Dohvaćanje detalja kupljene ulaznice**

Za dohvaćanje detalja kupljene ulaznice koristit će se funkcija getEventDetailsAsync sa slike 4.29. Funkciji se predaje ID događaja i ID korisnika, tj. polaznika događaja. Ako jedan od ta dva ulazna parametra ne postoji funkcija se završava. U zbirci *Events* traži se dokument s ID-em događaja te će se u tom dokumentu u podzbirci *Attendees* pronaći dokument s ID-em korisnika. Za taj dokument napravit će se *listener* i time „slušati“ izmjene dokumenta. Ako su

prethodni koraci uspješno obavljeni, dobivaju se detalji ulaznice za kupljeni događaj te se prosljeđuju za daljnje korištenje. U slučaju da dođe do pogreške, poslat će se poruka ovisno o pogrešci.

```
const getEventDetailsAsync = (idEvent?: string, idAttendee?: string) => async (
  dispatch: AppDispatch,
) => {
  if(!idAttendee || !idEvent) {
    return
  }

  dispatch(EventDetailsActions.addRequest());

  const successFunction = (data: Attendee) => {
   dispatch(EventDetailsActions.addEventDetails(data));
  };

  const errorFunction = (error: string) =>
    dispatch(EventDetailsActions.addError(error));

  const listener = eventCollectionService
    .getDocRef(idEvent)
    .withSubcollection(Subcollection.Attendees)
    .findAndListen(idAttendee, successFunction, errorFunction);
  dispatch(FirestoreListenerActions.setListener('event-details', listener));

  return;
};
```

Sl. 4.29 . *Funkcija za dohvaćanje detalja kupljene ulaznice*

4.2.9. **Odjava korisničkog računa iz aplikacije**

Kako bi se odjavio korisnik iz aplikacije potrebno je pozvati funkciju logoutAsync sa slike 4.30. Funkcija će odjaviti korisnika preko Firebase ugrađene funkcije logoutAsync te će pozvati funkciju unsubscribeListeners za prestanak „slušanja“ Firebase *listenera* tj WebSocketa.

```
const logoutAsync = () => async (dispatch: AppDispatch) => {
  dispatch(AuthActions.loading());
  await firebaseAuth.logoutAsync();

  dispatch(FirestoreListenerThunks.unsubscribeListeners());
};
```

Sl. 4.30. *Funkcija za odjavu korisnika iz aplikacije*

Funkcija unsubscribeListeners sa slike 4.31 u sebi sadržava petlju kroz koju se prolazi i zatim se brišu svi *listeneri* koji su stalno „slušali“ promjene u aplikaciji.

```
const unsubscribeListeners = () => async (
  dispatch: AppDispatch,
  getState: () => AppState,
) => {
  const listeners = getState().listeners;

  Object.values(listeners).forEach((listener) => {
    if (listener) {
      listener();
    }
  });

  dispatch(FirestoreListenerActions.removeAllListeners());
};
```

Sl. 4.31 . *Funkcija za brisanje svih postojećih listenera u aplikaciji*

U slijedećem poglavlju pokazat će se korištenje podataka koji su dobiveni funkcijama iz ovog poglavlja.

4.3. Prikaz programskog rješenja na strani klijenta

Kako bi se prikazali svi potrebni podaci koji su dobiveni funkcijama objašnjenim u prošlom poglavlju potrebno je programsko rješenje na strani klijenta.

4.3.1. Spremanje stanja pomoću Reduxa

Za korištenje Reduxa potrebno je prvo obuhvatit sa spremnikom (eng. *store*) cijelu navigaciju koja se zove RootNavigator. U njoj se nalaze sve komponente kao na slici 4.32 kako bi cijela aplikacija mogla imati pristup spremniku.

```
import { configureStore } from './src/modules/redux-store';
import { RootNavigator } from './src/modules/navigation';

const App = () => {
    const store = configureStore();
    return (
        <Provider store={store}>
            <RootNavigator />
        </Provider>
    );
};
```

Sl. 4.32 . *Root navigacija obgrljena spremnikom*

Zatim je potrebno imat AppState kao na slici 4.33 gdje se nalazi popis svih stanja aplikacije koji se nalaze u spremniku i koje može dohvatit bilo koja komponenta u aplikaciji.

```
You, 2 months ago | 1 author (You)
import { AuthState } from 'modules/authentication';
import { EventDetailsState } from 'modules/event-details';
import { EventsState } from 'modules/events';
import { FirestoreListenerState } from 'modules/firebase';
import { ProfileState } from 'modules/profile';
import { TicketTypesState } from 'modules/ticket-types';
You, 2 months ago | 1 author (You)
export interface AppState {
  auth: AuthState;
  listeners: FirestoreListenerState;
  profile: ProfileState;
  events: EventsState;
  eventDetails: EventDetailsState;
  ticketTypes: TicketTypesState;
}
```

Sl. 4.33. *Popis dijeljenih stanja aplikacije u spremniku AppState*

Svako stanje u aplikaciji prema arhitekturi Reduxa sastoji se od: akcija, međuspremnika (eng. middleware) i *reducer*-a. Na primjeru stanja svih događaja (events u spremniku AppState) pokazat će se korištenje Reduxa. U akcijama se nalazi zahtjev pokretanja akcije, dodavanje događaja, dodavanje pogreške u stanje i ponovno postavljanje događaja tj. pražnjenje niza događaja kao na slici 4.34.

```
import { createAction, ActionUnion } from 'modules/redux-store';
import { Event } from 'models';

import { EventTypes } from './types';

const AddActions = {
  addRequest: () => createAction(EventTypes.AddRequest),
  addEvents: (events: Event[]) =>
    createAction(EventTypes.AddEvents, { events }),
  addError: (error: string) => createAction(EventTypes.AddError, { error }),
  resetEvents: () => createAction(EventTypes.ResetEvents),
};

export const EventActions = {
  ...AddActions,
};

export type EventActions = ActionUnion<typeof EventActions>;
```

Sl. 4.34. *Akcije svih događaja*

U međuspremniku kao na slici 4.35 će se koristiti prethodno navedene akcije. Prilikom ovog koraka dohvaćaju se podaci iz baze podataka koristeći *listener* koji „sluša“ izmjene na bazi tako što se prvo šalje zahtjev za pokretanje akcije, zatim se dobiveni događaji pohranjuju u spremnik, a ako dođe do pogreške u spremnik se šalje odgovarajuća pogreška.

```
You, 3 months ago • get all events
const eventCollectionService = FirestoreService<Event>(Collection.Events);

const getEventsAsync = () => async (
  dispatch: AppDispatch,
  getState: () => AppState,
) => {
  const eventListener = getState().listeners['events'];
  if (eventListener) {
    return;
  }

  dispatch(EventActions.addRequest());

  const successFunction = (data: Event[]) => {
   dispatch(EventActions.addEvents(data));
  };

  const errorFunction = (error: string) =>
    dispatch(EventActions.addError(error));

  const listener = eventCollectionService
    .filterAndListen(successFunction, errorFunction);
  dispatch(FirestoreListenerActions.setListener('events', listener));

  return;
};

export const EventThunks = {
  getEventsAsync,
};
```

Sl. 4.35. *Međuspremnik svih događaja*

U *reduceru* se postavljaju atributi stanja. Nalaze se atributi eventsAreChanging, error i events što je vidljivo na slici 4.36. Atribut eventsAreChanging je potreban jer on pokazuje događa li se neka akcija ili se možda učitavaju podaci. U aplikaciji kad je eventsAreChanging istinit tj. (eng. true) pokazat će se animacija za učitavanje. Error prima odgovarajuću poruku o pogrešci ako dođe do pogreške. Atribut events je niz koji se puni kad se dohvate svi događaji. Početno stanje atributa je takvo da je eventsAreChanging istinit, error je nedefiniran, a events je prazan niz. Ovisno o tipu akcije koja se događa switch naredba grananja postavit će vrijednosti u atribute stanja.

```
export interface EventsState {
  eventsAreChanging: boolean;
  error?: string;
  events: Event[];
}

const INITIAL_STATE: EventsState = {
  eventsAreChanging: true,
  error: undefined,
  events: [],
};

export const EventReducer = (state: EventsState = INITIAL_STATE,
   action: EventActions) => {
  switch (action.type) {
    case EventTypes.AddRequest:
      return {
        ...state,
        eventsAreChanging: true,
        error: undefined,
      };

    case EventTypes.AddEvents:
      return {
        ...state,
        eventsAreChanging: false,
        error: undefined,
        events: action.payload.events,
      };

    case EventTypes.AddError:
      return {
        ...state,
        eventsAreChanging: false,
        error: action.payload.error,
      };

    case EventTypes.ResetEvents:
      return INITIAL_STATE;

    default:
      return state || INITIAL_STATE;
  }
};
```

Sl. 4.36. *Reducer svih događaja*

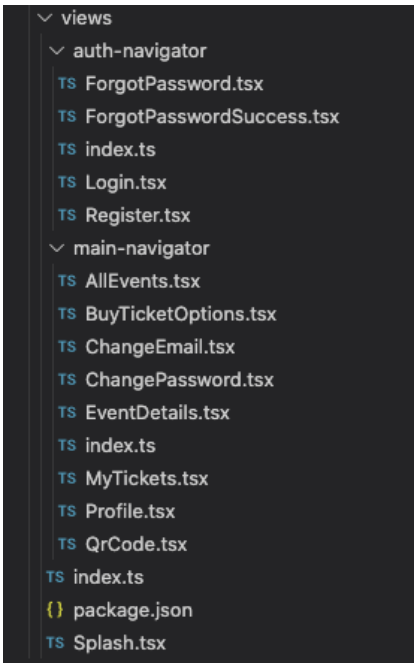
Za korištenje stanja svih događaja koji su pohranjeni u spremnik potrebno je u komponenti navesti iz spremnika AppState naziv stanja koji se želi koristit (events) kao na slici 4.37. Kako bi se lakše koristili atributi events, error, eventsAreChanging iz stanja svih događaja napravljeno je destrukturiranje objekta.

```
const { events, error, eventsAreChanging} = useSelector(
  (state: AppState) => state.events,
);
```

Sl. 4.37. *Korištenje stanja svih događaja u komponenti*

4.3.2. **Navigacija aplikacije**

Za kretanje kroz aplikaciju napravljena je navigacija. Prema funkcionalnostima u aplikaciji zasloni tj. komponente su podijeljene na dvije grupe: auth-navigator i main-navigator te Splash komponentu kao na slici 4.38.



Sl. 4.38. *Raspodjela zaslona na grupe auth-navigator i main-navigator*

U prošlom potpoglavlju prikazano je kako je Redux spremnik obgrlio RootNavigator gdje se nalaze sve komponente. U RootNavigatoru nalaze se Splash, AuthNavigator i MainNavigator komponente kao na slici 4.39. Prilikom pokretanja aplikacija prvo će se pokrenut komponenta Splash.

```
export const RootNavigator: React.FC = () => {
  const Stack = createStackNavigator<RoutesRecord>();

  return (
    <NavigationContainer>
      <Stack.Navigator
        screenOptions={{ headerShown: false, gestureEnabled: false }}
        initialRouteName="splash"
      >
        <Stack.Screen name="splash" component={views.Splash} />
        <Stack.Screen name="auth" component={AuthNavigator} />
        <Stack.Screen name="main" component={MainNavigator} />

      </Stack.Navigator>
    </NavigationContainer>
  );
};
```

Sl. 4.39. *RootNavigator komponenta za navigaciju*

U Splash komponenti kao na slici 4.40 dohvaćaju se iz Redux spremnika stanja o prijavi korisnika i korisnikovim podacima. Ako su se učitali svi podaci i korisnik nije prijavljen, navigacija aplikacije će korisnika odvesti na zaslon auth tj AuthNavigator komponentu. Ako je korisnik prijavljen, navigacija aplikacije će korisnika odvesti na zaslon main tj MainNavigator komponentu.

```
export const Splash: React.FC<NavigationParams> = ({ navigation }) => {        You, 6 months ago
  const dispatch = useDispatch();
  const { loading, isLoggedIn, user} = useSelector((state: AppState) => state.auth);
  const finishedLoading = !loading;

  /** Initialize firebase auth listener */
  function onMount() {
    dispatch(AuthThunks.initAuthListener());
  }

  /** When firebase auth listener does it's thing, redirect to appropriate navigator stack */
  function onAuthChanged() {
    const finishedLoading = !loading;

    if (finishedLoading) {
      if (!isLoggedIn) {
        navigation.replace('auth');
        return;
      }

      navigation.replace(user ? 'main' : 'auth');
    }
  }
```

Sl. 4.40. *Splash komponenta*

U AuthNavigator komponenti na slici 4.41 nalaze se komponente: Login, Register, ForgotPassword i ForgotPasswordSuccess. Prilikom poziva AuthNavigator komponente prvo će se pokrenuti Login komponenta. Funkcija useEffect na promjenu stanja isLoggedIn i user se okida te poziva funkciju onAuthStateChanged. Ako je korisnik prijavljen i podaci korisnika postoje, funkcija onAuthStateChanged pomoću navigacije vodi na zaslon main tj. MainNavigator komponentu.

```
export const AuthNavigator: React.FC<NavigationParams> = ({ navigation }) => {
  const { isLoggedIn, user } = useSelector((state: AppState) => state.auth);

  const Stack = createStackNavigator<RoutesRecord>();

  // If user has been logged in navigate to main navigator
  function onAuthStateChanged() {
    if (isLoggedIn && user)  {
      navigation.navigate('main');
      return;
    }
  }

  useEffect(onAuthStateChanged, [isLoggedIn, user]);

  return (
    <Stack.Navigator
      screenOptions={{ headerShown: false, gestureEnabled: false }}
    >
      <Stack.Screen name="login" component={views.Login} />
      <Stack.Screen name="register" component={views.Register} />
      <Stack.Screen name="forgot-password" component={views.ForgotPassword} />
      <Stack.Screen
        name="forgot-password-success"
        component={views.ForgotPasswordSuccess}      You, 6 months ago • Root, au
      />
    </Stack.Navigator>
  );
};
```

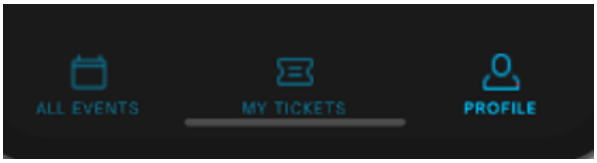
Sl. 4.41. *AuthNavigator komponenta*

U MainNavigator komponenti na slici 4.42 nalaze se komponente: AllEvents, Profile, MyTickets, EventDetails, QrCode, BuyTicketOptions, ChangeEmail, ChangePassword. Prilikom poziva MainNavigator komponente, prvo će se pokrenut AllEvents komponenta. Funkcija useEffect na promjenu stanja isLoggedIn se okida te poziva funkciju onAuthStateChanged. Ako korisnik nije prijavljen, funkcija onAuthStateChanged pomoću navigacije vodi na zaslon auth tj. AuthNavigator komponentu.

```
export const MainNavigator: React.FC<NavigationParams> = ({ navigation }) => {
  const { isLoggedIn } = useSelector((state: AppState) => state.auth);
  const Stack = createStackNavigator<RoutesRecord>();

  /** If user has logged out, go back to landing */
  function onAuthStateChanged() {
    if (!isLoggedIn) {
      navigation.replace('auth', undefined);
    }
  }

  useEffect(onAuthStateChanged, [isLoggedIn]);

  return (
    <Stack.Navigator
      screenOptions={{
        headerShown: false,
        gestureEnabled: false,
      }}
    >
      <Stack.Screen name="allEvents" component={views.AllEvents} />
      <Stack.Screen name="profile" component={views.Profile} />
      <Stack.Screen name="myTickets" component={views.MyTickets} />
      <Stack.Screen name="event-details" component={views.EventDetails} />
      <Stack.Screen name="qr-code" component={views.QrCode} />
      <Stack.Screen name="buy-ticket-options" component={views.BuyTicketOptions} />
      <Stack.Screen name="change-email" component={views.ChangeEmail} />
      <Stack.Screen name="change-password" component={views.ChangePassword} />
    </Stack.Navigator>
  );
};      You, 6 months ago • Root, auth and main navigator added
```

Sl. 4.42. *MainNavigator komponenta*

Za lakšu promjenu zaslona, u aplikaciji je napravljen izbornik tj. TabNavigator komponenta, kao na slici 4.43. U TabNavigator izborniku nalaze se rute tj komponente AllEvents, MyTickets i Profile. Prvo je postavljena konstanta icons s ikonicama za svaku komponentu koja će biti u izborniku. Funkcija renderTab vratit će klikabilnu ikonicu s imenom zaslona. Ako je trenutno aktivan zaslon, ikonica i ime zaslona bit će potpuno vidljive i podebljane, dok će zaslonima koji nisu aktivni vidljivost biti 60% kao na slici 4.44. Navigacija klikom na ikonicu ili ime zaslona vodi na rutu tog zaslona.



Sl. 4.43. *TabNavigator komponenta*

```
const icons = {
  allEvents: require('assets/icons/events.png'),
  myTickets: require('assets/icons/tickets.png'),
  profile: require('assets/icons/profile.png'),
};
export const TabNavigator: React.FC<Props> = ({ backgroundColor }) => {
  const navigation = useNavigation<NavigationProps>();

  function isActiveRoute(route: Routes) {
    const state = navigation.dangerouslyGetState();
    if (!state) {
      return false;
    }

    return state.routes[state.index].name === route;
  }

  function navigateToRoute(route: Routes) {
    navigation.replace(route);
  }

  function renderTab(route: Routes, title: string) {
    const isActive = isActiveRoute(route);

    return (
      <TouchableOpacity
        onPress={navigateToRoute.bind(null, route)}
        disabled={isActive}
        style={styles.tab}
      >
        <_Image
          height={24}
          width={24}
          source={icons[route]}
          resizeMode="contain"
          style={{ opacity: isActive ? 1 : 0.6 }}
        />
        <_Text
          size="tny"
          color={Color.Primary}
          opacity={isActive ? 1 : 0.6}
          textTransform="uppercase"
          margins={{ marginTop: Spacing.Tny }}
          style={[styles.letterSpacing,
            {fontWeight: isActive? 'bold' : 'normal'}]}>
          {title}
        </_Text>
      </TouchableOpacity>
    );
  }

  return (
    <View
      style={{
        ...styles.container,
        backgroundColor,
      }}
    >
      {renderTab('allEvents', 'All events')}
      {renderTab('myTickets', 'My tickets')}
      {renderTab('profile', 'Profile')}
    </View>
  );
};
```

Sl. 4.44. *TabNavigator komponenta*

4.3.3. **Prikaz svih događaja**

Kako bi se prikazali svi događaji prvo je potrebno iz Redux spremnika dohvatit događaje kao na slici 4.45.

```
const { events } = useSelector(
  (state: AppState) => state.events,
);
```

Sl. 4.45. *Dohvat događaja iz Redux spremnika*

Za korištenje događaja na zaslonu napravljena je konstanta allEvents kao na slici 4.46 kojoj se preko funkcije setAllEvents postavljaju vrijednosti. Na svaku promjenu događaja u Redux spremniku prazni se niz allEvents. Petljom forEach za svaki događaj, ako događaj nije privatnog karaktera tj. javan je, dodaje se događaj funkcijom setAllEvents u allEvents.

```
const [allEvents, setAllEvents] = useState<Event[]>([]);
useEffect(() => {
  setAllEvents([])
  events.forEach(event => {
    if (!event.privateEvent) {
      setAllEvents((events) => [...events, event])
    }
  })
}, [events]);
```

Sl. 4.46. *Popunjavanje allEvents konstante događajima*

Kako bi se događaji mogli filtrirati napravljena je konstanta showAllEvents kojoj se preko funkcije setShowAllEvents postavljaju vrijednosti. Na svaku promjenu konstante allEvents, funkcijom setShowAllEvents postavlja se vrijednost showAllEvents na vrijednost allEvents, što je vidljivo na slici 4.47. Pomoću showAllEvents varijable određivat će se koji se događaji žele prikazati pomoću filtera u slijedećem potpoglavlju.

```
const [showAllEvents, setShowAllEvents] = useState<Event[]>([]);
useEffect(() => {
  setShowAllEvents(allEvents)
}, [allEvents]);
```

Sl. 4.47. *Popunjavanje showAllEvents konstante*

Za prikaz događaja koristit će se komponenta FlatList koja je dio react-native paketa. Koristi se FlatList komponenta jer ona učitava samo one stavke popisa koje se mogu prikazati na zaslonu. Primjerice, ima sto događaja, ali na zaslonu se može prikazati u jednom trenutku samo pet događaja, učitat će se samo tih pet događaja i narednim listanjem učitavat će se još toliko

koliko se prikazuje na zaslonu. Takvim učitavanjem povećava se optimizacija aplikacije. Predaje se konstanta showAllEvents FlatListi kao podatak koji će prikazivati, a način na koji će se prikazivati svaka stavka u listi je zadana funkcijom renderEvents kao na slici 4.48.

```
<FlatList
  contentContainerStyle={styles.wrapper}
  data={showAllEvents}
  renderItem={renderEvents}
  showsVerticalScrollIndicator={false}
  showsHorizontalScrollIndicator={false}
/>
```

Sl. 4.48. *FlatList komponenta*

U funkciji renderEvents sa slike 4.50 ulazni parametar je event tj. u FlatListi funkciji se predaje showAllEvents. Funkcija vraća spremnik koji klikom na njega pokreće funkciju getEventDetails tj vodi na zaslon s detaljima događaja. U spremniku se nalazi naslov događaja, slika i pojedinosti događaja. Ako ne postoji slika tj. photoUrl stavlja se zadana slika za događaje koji nemaju sliku. Zadana slika nalazi se u direktoriju s resursima koji bit će objašnjeni u potpoglavlju 4.3.7. U pojedinostima događaja kao na slici 4.49 nalazi se početak događaja i adresa događaja. Početak događaja formatiran je pomoću node paketa date-fns.

Sl. 4.49. *Stavka u FlatListi prikazana pomoću funcije renderEvents*

```
function renderEvents(event: ListRenderItemInfo<Event>) {
  return (
    <TouchableOpacity
      onPress={() => getEventDetails(event.item, event.item.eventId)}
      style={styles.listItem}
      key={event.item.eventId}>
      <_Text
        size="xlrg"
        font="MaisonNeueExtended-ExtraBold"
        style={styles.listItemTitle}
        color={Color.White}
      >
        {event.item.title}
      </_Text>

      {Boolean(event.item.photoUrl) && (
        <_Image
          style={styles.itemImage}
          source={{ uri: event.item.photoUrl }}
        />
      )}

      {Boolean(!event.item.photoUrl) && (
        <_Image
          style={styles.itemImage}
          source={require('assets/images/party.jpg')}
        />
      )}

      <View style={styles.listDetails}>
        <_Text
          size="tny"
          font="MaisonNeueExtended-Medium"
          color={Color.Primary}
          style={styles.listDetailsDate}
        >
          {format(event.item.startDate, 'dd.MM.yyyy H')}H
        </_Text>

        <_Text
          size="tny"
          font="MaisonNeueExtended-Medium"
          color={Color.Primary}
          style={styles.listDetailsLocation}
          textTransform={'uppercase'}
        >
          {event.item.location.address}
        </_Text>
      </View>

    </TouchableOpacity>
  )
}
```

Sl. 4.50. *Funkcija renderEvents*

4.3.4. **Filtriranje događaja**

Za filtriranje događaja napravljena je konstanta eventTypeFilter sa slike 4.51 kojoj se preko funkcije setEventTypeFilter postavljaju vrijednosti. Konstanta eventTypeFilter primat će vrijednost koja je odabrana za filter. Isto tako napravljena je konstanta showPicker kojoj se preko funkcije setShowPicker postavljaju vrijednosti boolean. Konstanta showPicker određivat će treba li padajući izbornik za filter događaja biti prikazan.

```
const [eventTypeFilter, setEventTypeFilter] = useState<React.ReactText>();
const [showPicker, setShowPicker] = useState<boolean>(false);
```

Sl. 4.51. *Konstante eventTypeFilter, showPicker i funkcije za postavljanje njihovih vrijednosti*

Za odabir filtriranja napravljena je klikabilna ikonica filtera. Klikom na ikonicu poziva se funkcija setShowPicker koja mijenja vrijednost showPicker konstanti s istine na laž i obrnuto kao na slici 4.52.

```
<TouchableOpacity
  onPress={() => setShowPicker(!showPicker)}
>
  <_Image
    height={20}
    width={20}
    resizeMode="contain"
    source={require('assets/icons/filter.png')}
  />
</TouchableOpacity>
```

Sl. 4.52. *Ikonica za otvaranje ili zatvaranje filter padajućeg izbornika*

Ako je vrijednost konstante istina pokazat će se komponenta Picker koja je padajući izbornik. Komponenta Picker uvezena je iz node paketa @react-native-picker/picker. U komponenti Picker odabrana vrijednost padajućeg izbornika je konstanta eventTypeFilter. Na promjenu vrijednosti padajućeg izbornika poziva se funkcija filterEvents kojoj se predaje odabrana vrijednost izbornika. Prva stavka padajućeg izbornika je *Choose a value* s praznom vrijednosti. Ostale stavke padajućeg izbornika bit će generirane pomoću metode za niz map. Za svaki element u nizu eventTypeOptions napravit će se stavka s nazivom, vrijednosti i ključem u padajućem izborniku što se vidi na slici 4.53.

```
{showPicker && (
  <Picker
    itemStyle={styles.pickerItem}
    style={{color: Color.Primary}}
    dropdownIconColor={Color.Primary}
    mode="dropdown"
    selectedValue={eventTypeFilter}
    onValueChange={(itemValue) => { filterEvents(itemValue) }}>
    <Picker.Item label="Choose a value" value="" />
    {eventTypeOptions.map((item, index) => {
      return (<Picker.Item label={item.value}
        value={item.value} key={index} />)
    })}
  </Picker>
)}
```

Sl. 4.53. *Padajući izbornik pomoću komponente Picker*

Funkcija filterEvents sa slike 4.54 prima odabranu vrijednost izbornika. Postavlja primljenu vrijednost u eventTypeFilter pomoću funkcije setEventTypeFilter. Zatim niz koji se koristi u FlatListi showAllEvents ostavlja praznim kako bi se mogao napunit s novim vrijednostima. Metodom forEach prolazi se kroz niz allEvents po svakom elementu. Ako element tog niza tj. događaj ima tip događaja kao vrijednost koja je predana funkciji, taj događaj se dodaje u niz showAllEvents pomoću funkcije setShowAllEvents. Ako je vrijednost koju je primila funkcija "" tj. prazan string, svaki događaj se sprema u showAllEvents pomoću funkcije setShowAllEvents. Funkcija će primiti prazan string ako je u padajućem izborniku odabrana stavka s imenom „*Choose a value*".

```
function filterEvents(value: React.ReactText) {
  setEventTypeFilter(value);

  setShowAllEvents([])
  allEvents.forEach(event => {
    if (event.type === value) {
      setShowAllEvents((events) => [...events, event]);
      return
    }
    if (value === '') {
      setShowAllEvents((events) => [...events, event]);
      return
    }
    return
  })
}
```

Sl. 4.54. *Funkcija za filtriranje događaja*

4.3.5. **QR kod**

Za prikaz QR koda kupljene ulaznice za događaj koristit će se komponenta QRCode. Komponenta QRCode uvezena je iz node paketa react-native-qrcode-svg. Prije korištenja komponente QRCode postavljena je širina zaslona u konstantu windowWidth kao na slici 4.55.

```
const windowWidth = Dimensions.get('window').width;
```

Sl. 4.55. *Konstanta windowWidth*

Vrijednost koja je predana komponenti QRCode je string u kojem piše tko je vlasnik ulaznice, broj ulaznica i do kojeg datuma ulaznica vrijedi što je vidljivo na slici 4.56. Veličina QR koda bit će konstanta windowWidth – 100dp. Pozadina QR koda bit će transparentna.

```
<QRCode
  value={`Ticket owner: ${eventDetails?.issuedTo.fullName} \nAmount:
  ${eventDetails.amount} \nTicket valid to:
  ${format(eventDetails.ticketTypeValidTo, 'dd.MM.yyyy')}`}
  size={windowWidth - 100}
  logoBackgroundColor='transparent'
/>
```

Sl. 4.56. *QRCode komponenta*

4.3.6. **Prikaz poruke kod pogreške**

Kako bi se korisniku aplikacije pokazalo da neka akcija nije uspješno izvršena tj. da je došlo do pogreške, korisniku se prikaže poruka s pogreškom. Poruke s pogreškom su moguće na svakom zaslonu gdje se događa neka akcija. Na primjeru zaslona za prijavu korisnika pokazat će se prikaz poruke kod pogreške. Potrebno je prvo iz Redux spremnika dohvatiti atribut error iz stanja auth koje služi za prijavu kao na slici 4.57.

```
const { loading, error } = useSelector((state: AppState) => state.auth);
```

Sl. 4.57. *Dohvat pogreške iz Redux spremnika*

Ako postoji pogreška koja je dohvaćena iz Redux spremnika pokazat će se pogreška bojom vrijednosti enum Color.Danger, tny veličinom i imat će donju marginu vrijednosti enum Spacing.Sml kao na slici 4.58. Ove vrijednosti bit će detaljnije objašnjene u potpoglavlju 4.3.8.

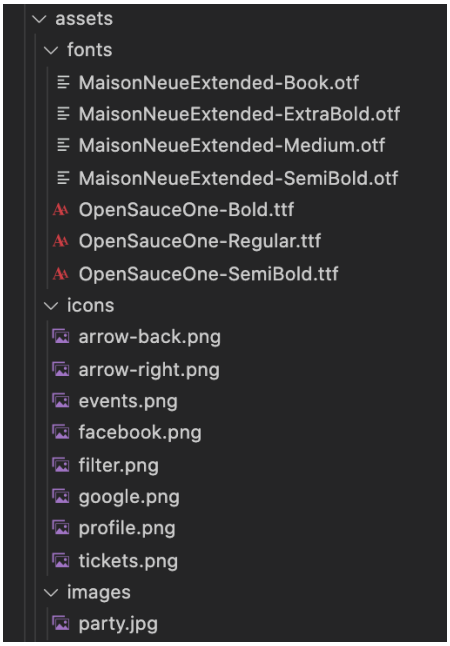
```
{Boolean(error) && (
  <_Text
    color={Color.Danger}
    size="tny"
    margins={{ marginBottom: Spacing.Sml }}
  >
    {error}
  </_Text>
)}
```

Sl. 4.58. *Prikaz pogreške ako pogreška postoji*

4.3.7. **Resursi**

U direktoriju resursa sa slike 4.59 nalaze se fontovi, ikonice i slike koje se koriste u aplikaciji. Od fontova koriste se MaisonNeueExtended i OpenSauceOne s njihovim različitim inačicama.

.



Sl. 4.59. *Direktorij resursi*

Kako bi iOS operacijski sustav prepoznao fontove potrebno je dodati u info.plist datoteku programski kod sa slike 4.60.

```
<array>
    <string>OpenSauceOne-Regular.ttf</string>
    <string>OpenSauceOne-Bold.ttf</string>
    <string>OpenSauceOne-SemiBold.ttf</string>
    <string>MaisonNeueExtended-Book.otf</string>
    <string>MaisonNeueExtended-ExtraBold.otf</string>
    <string>MaisonNeueExtended-Medium.otf</string>
    <string>MaisonNeueExtended-SemiBold.otf</string>
</array>
```

Sl. 4.60. *Datoteka info.plist s kodom za korištenje fontova*

4.3.8. **Globalni stilovi**

Za lakše korištenje stilova koji se često pojavljuju kroz aplikaciju napravljeni su određeni globalni stilovi kao što su gumbi, boje i razmaci. Komponenta Globalnog gumba sa slike 4.61 koji se može koristiti kroz cijelu aplikaciju može primiti četiri dodatna parametra koje zadani gumb React Native-a prima. Dodatni parametri su naziv, tip, učitavanje i sakrij onemogućeni stil. Primary tip gumba bit će kao primary boja koja će biti objašnjena u nastavku, dok će negative biti proziran gumb. Sve dok je parametar učitavanja istinit pokazivat će se komponenta ActivityIndicator koja je animacija učitavanja. Ako parametar učitavanja nije istinit pokazat će se gumb u kojemu se nalazi naziv gumba, fonta MaisonNeueExtendet-SemiBold i crne boje.

```
interface OwnProps extends TouchableOpacityProps {
  title: string;
  type?: 'primary' | 'negative';
  loading?: boolean;
  hideDisabledStyle?: boolean;
}

type Props = OwnProps & SpacingProps;

export const Button: React.FC<Props> = ({
  title,
  type = 'primary',
  style,
  disabled,
  hideDisabledStyle,
  margins,
  paddings,
  loading,
  ...rest
}) => {
  if (loading) {
    return (
      <ActivityIndicator
        size="small"
        color={type === 'primary' ? Color.Primary : Color.TextNegative}
        style={styles.loading}
      />
    );
  }

  return (
    <View style={disabled && !hideDisabledStyle && styles.disabled}>
      <TouchableOpacity
        style={[styles.base, styles[type], margins, paddings, style]}
        disabled={disabled}
        {...rest}
      >
        <_Text
          font="MaisonNeueExtended-SemiBold"
          size="sml"
          color="#000"
        >
          {title}
        </_Text>
      </TouchableOpacity>
    </View>
  );
};
```

Sl. 4.61. *Komponenta za globalni gumb*

Primjer korištenja globalnog gumba na slici 4.62 gdje prima parametre title, disabled, loading i onPress.

```
<Button
  title='Change Email'
  disabled={!isFormValid()}
  loading={infoChanging}
  onPress={updateEmail}
>
</Button>
```

Sl. 4.62. *Komponenta globalnog gumba prilikom korištenja na zaslonu*

Globalne boje zadane su kao enum zbog lakšeg korištenja *autocomplete* funkcionalnosti VS Code-a i s manjom mogućnosti za pogrešnim upisom vrijednosti. Boje koje se nalaze u enum Color sa slike 4.63 su boje koje se koriste kroz aplikaciju i koje su bile zadane u dizajnu na Figmi u paletama boja aplikacije. Zadane su heksadecimalnim i RGB-a zapisom.

```
export enum Color {
  Primary = '#00C2FF',
  Secondary = '#003049',
  White = '#FFFFFF',
  Danger = '#C32424',
  Success = '#259A51',
  Text = '#F9F7F4',
  TextLight = '#8C8C8C',
  TextNegative = '#F9F7F4',
  Border = '#515151',
  BorderTransparent = 'rgba(119, 119, 119, .2)',
  BorderUltraLight = '#F9F7F4',
  Background = '#1C1C1C',
  Placeholder = 'rgba(255,255,255, 0.7)'
}
```

Sl. 4.63. *Enum Color za boje korištene u aplikaciji*

Globalni razmaci zadani su iz istog razloga kao enum na slici 4.64. Nalaze se najčešće korišteni razmaci u aplikaciji. Vrijednosti koje su dane su u zadanoj React Native jedinici dp (eng. *layout size*).

```
export enum Spacing {
  Tny = 4,
  Sml = 12,
  Med = 20,
  Lrg = 28,
  LrgXlrg = 32,
  Xlrg = 40,
}
```

Sl. 4.64. *Enum Spacing za najčešće korištene razmake u aplikaciji*

U slijedećem poglavlju bit će prikazano korištenje aplikacije te izgled nakon spajanja programskog koda na strani poslužitelja i klijenta.

5. PRIKAZ NAČINA KORIŠTENJA I ISPITIVANJE APLIKACIJE

U ovom poglavlju opisane su upute za korištenje aplikacije, ispitivanje rada aplikacije pomoću dva korisnička slučaja i dani su rezultati ostvarenosti svih zahtjeva aplikacije.

5.1. Način korištenja aplikacije

Registracija i prijava obavlja se kroz šest sljedećih koraka u kojima je potrebno:

1. Odlučiti način na koji će se korisnik prijaviti u aplikaciju:
    a) Prijava korisničkim računom
    b) Prijava Google računom
    c) Prijava Facebook računom
2. Ukoliko ne postoji korisnički račun otići na zaslon za registraciju
3. Na zaslonu za registraciju unijeti podatke za:
    a) ime i prezime
    b) e-poštu
    c) lozinku
    d) broj telefona
4. Pritisnuti gumb “Sign Up“ za kreiranje računa i prijavu
5. Ako korisnik ima korisnički račun, za prijavu treba unijeti podatke za:
    a) e-poštu
    b) lozinku
6. Pritisnuti gumb “Login“ za prijavu s korisničkim računom

Ponovno postavljanlje lozinke obavlja se kroz šest sljedećih koraka u kojima je potrebno:

1. Pritisnuti gumb “Forgot password?“ na zaslonu za prijavu
2. Unijeti svoju e-poštu
3. Pritisnuti gumb “Reset password“
4. Ako postoji korisnički račun aplikacija će odvesti korisnika na zaslon s potvrdom, a ako ne postoji račun pokazat će poruku s odgovarajućom pogreškom.
5. Otići na svoju e-poštu i kliknuti na poveznicu koju je aplikacija poslala
6. Unijeti novu lozinku i potvrditi

Pretraga događaja s filterom prema tipu događaja obavlja se kroz pet sljedećih koraka u kojima je potrebno:

1. Biti prijavljen u aplikaciju
2. U izborniku aplikacije odabrati “ALL EVENTS“ zaslon
3. Odabrati gumb s ikonicom filter na vrhu zaslona
4. Odabrati tip događaja od ponuđenih opcija:
    a) “Choose a value“ – zadana opcija koja prikazuje sve tipove događaja
    b) “Seminar“
    c) “Workshop“
    d) “Conference“
    e) “Trade show“
    f) “Party“
    g) “Concert“
    h) „Ceremonie
5. Od filtriranih događaja odabrati željeni događaj

Kupovina ulaznice za događaj obavlja se kroz pet sljedećih koraka u kojima je potrebno:

1. Odabrati “ALL EVENTS“ zaslon u izborniku aplikacije
2. Odabrati željeni događaj
3. Kliknuti na gumb “Buy options“
4. Kliknuti na gumb “Buy“ u željenoj vrsti ponuđenih opcija ulaznica
5. Ako je ulaznica uspješno kupljena prikazat će se poruka “Ticket is successfully bought“, u protivnom prikazat će se poruka s određenom pogreškom

Pregled i korištenje ulaznice obavlja se kroz dva sljedeća koraka u kojima je potrebno:

1. Odabrati “MY TICKETS“ zaslon u izborniku aplikacije
2. Kliknuti na gumb “QR Code“ na željenu ulaznicu

Promjena e-pošte i lozinke korisničkog računa obavlja se kroz deset sljedećih koraka u kojima je potrebno:

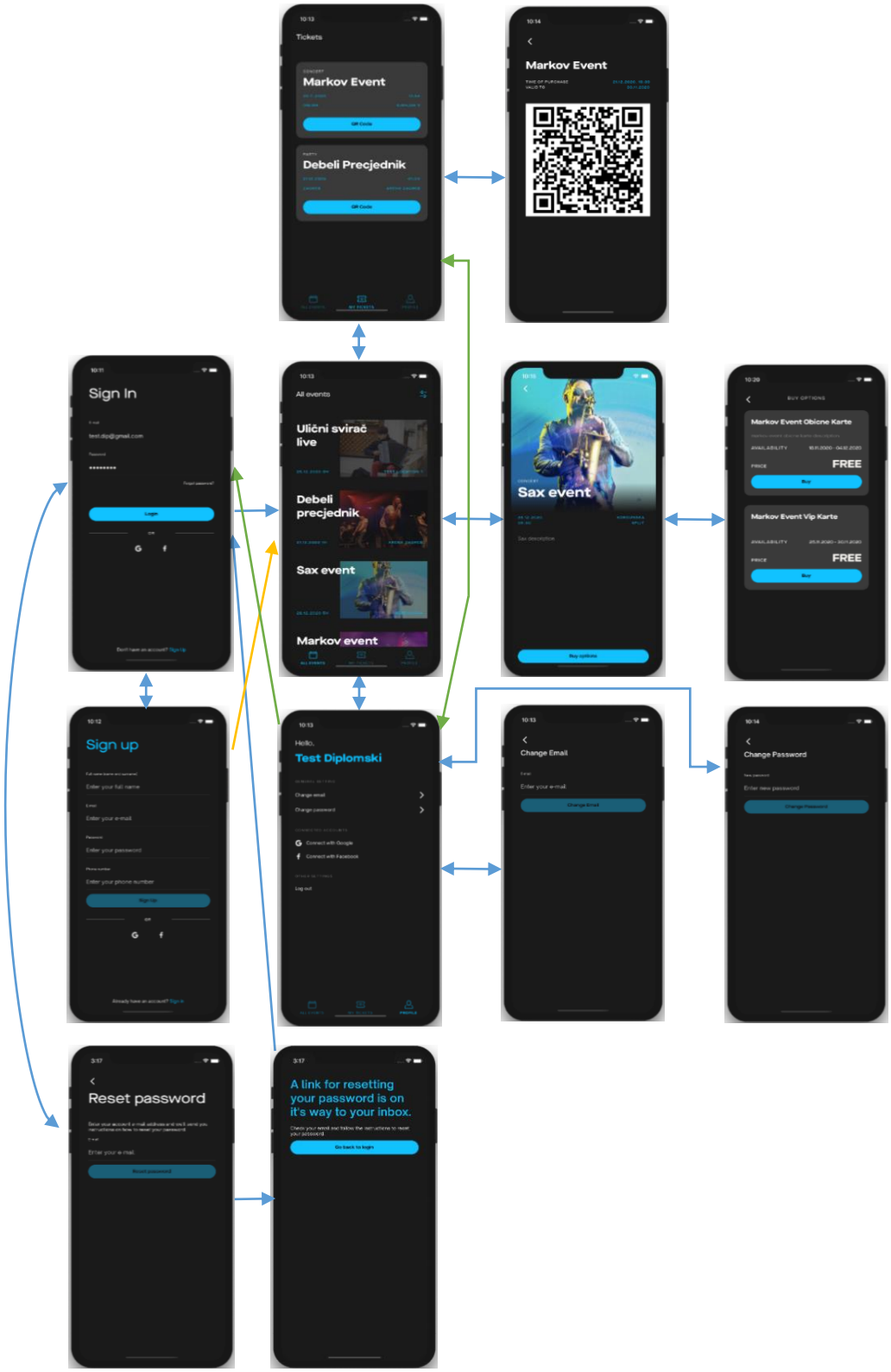
1. Odabrati “PROFILE“ zaslon u izborniku aplikacije
2. Kliknuti na gumb “Change email“ za otvaranje zaslona za promjenu e-pošte
3. Unijeti novu željenu e-poštu
4. Kliknuti na gumb “Change Email“ za promjenu e-pošte
5. Ako je e-pošta uspješno promijenjena prikazat će se poruka “Email is successfully changed“, u protivnom prikazat će se poruka s određenom pogreškom

6. Kliknuti na gumb za povratak na “PROFILE“ zaslon
7. Kliknuti na gumb “Change password“ za otvaranje zaslona za promjenu lozinke
8. Unijeti novu željenu lozinku
9. Kliknuti na gumb “Change Password“ za promjenu lozinke
10. Ako je lozinka uspješno promijenjena prikazat će se poruka “Password is successfully changed“, u protivnom prikazat će se poruka s određenom pogreškom

Povezivanje korisničkog računa s Gmail i Facebook računom obavlja se kroz jedanaest sljedećih koraka u kojima je potrebno:

1. Odabrat “PROFILE“ zaslon u izborniku aplikacije
2. Kliknuti na gumb “Connect with Google“
3. Kliknuti na gumb “Continue“ za potvrdu da aplikacija želi koristit google.com za prijavu
4. Odabrati željeni korisnički Google račun za povezivanje
5. Unijeti lozinku za odabrani korisnički Google račun
6. Kliknuti na gumb “Continue“ za povezivanje računa
7. Ako je korisnički račun uspješno povezan s gmail računom prikazat će se poruka “Account is successfully connected with your Google account“, u protivnom prikazat će se poruka s određenom pogreškom
8. Kliknuti na gumb “Connect with Facebook“
9. Kliknuti na gumb “Continue“ za potvrdu da aplikacija želi koristit facebook.com za prijavu
10. Kliknuti na gumb “Nastavi“ na facebook.com za potvdu prijave
11. Ako je korisnički račun uspješno povezan s Facebook računom prikazat će se poruka “Account is successfully connected with your Facebook account“, u protivnom prikazat će se poruka s određenom pogreškom

Prema funkcionalnostima iz načina korištenja aplikacije i dijagramu toka iz potpoglavlja 3.2 nastao je slijed zaslona aplikacije Ticketnator na slici 5.1.



Sl. 5.1 *Slijed zaslona aplikacije Ticketnator*

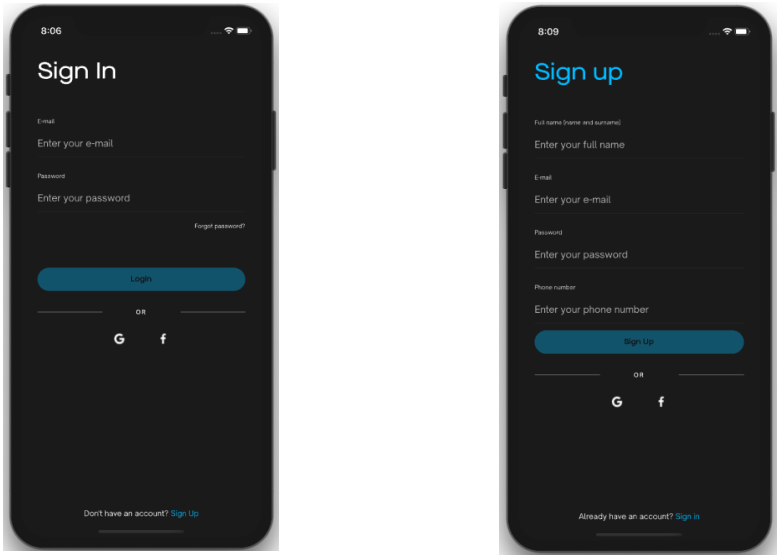
U slijedećem potpoglavlju prikazat će se korištenje aplikacije Ticketnator kroz dva korisnička slučaja.

5.2. Ispitivanje rada aplikacije

Kako bi se ispitao rad aplikacije napravit će se prikaz dva korisnička slučaja za prijavu u aplikaciju, upravljanja korisničkim računom, pregled događaja, kupovine ulaznice za događaj, korištenje ulaznice i sinkronizacija svih podataka. Rad aplikacije je ispitan na virtualnim uređajima iPhone 11, LG Nexus 5X i stvarnom uređaju Samsung Galaxy S9+.

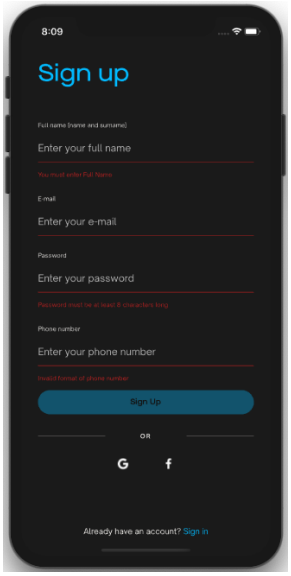
5.2.1. **Korisnički slučaj 1.**

Pokrenuta je aplikacija te je odabrana mogućnost izrade korisničkog računa kao na slici 5.2.



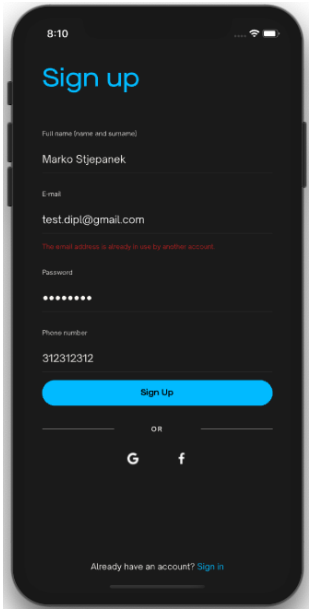
Sl. 5.2. *Pokretanje aplikacije – zaslon za prijavu korisnika i zaslon za kreiranje korisničkog računa*

Kako bi se korisniku olakšalo kreiranje korisničkog računa, korisnik će biti obaviješten ako ostavi prazno polje kao na slici 5.3.



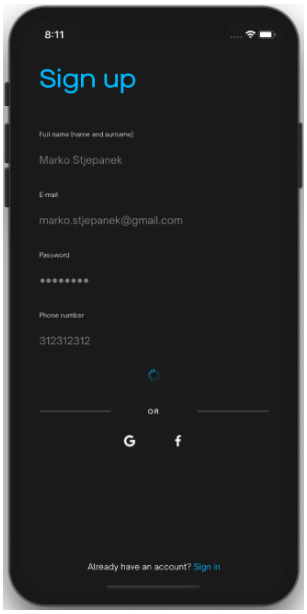
Sl. 5.3. *Kreiranje korisničkog računa – pogreška praznog polja*

Upiše li korisnik prilikom izrade korisničkog računa već korištenu e-poštu, aplikacija će ga obavijestiti o tome kao na slici 5.4.



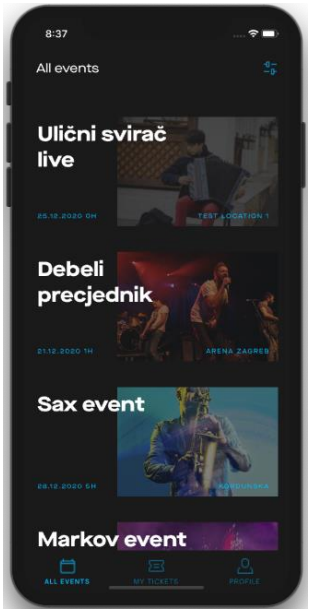
Sl. 5.4. *Unos već korištene e-pošte prilikom izrade korisničkog računa dat će za posljedicu poruku o pogrešci*

Kako bi se korisniku dalo do znanja da se njegovi podaci obrađuju i da mu se olakša čekanje na korištenje aplikacije, na zaslonu se prikazuje *loader* sa slike 5.5



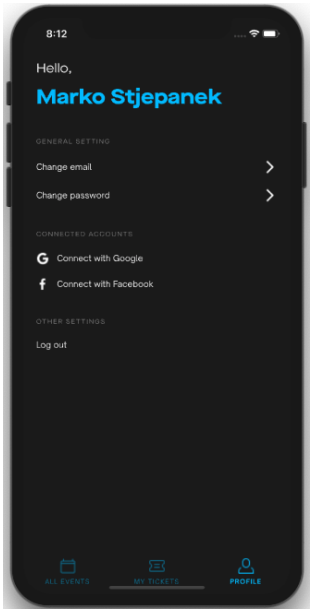
Sl. 5.5. *Loader prilikom učitavanja aplikacije*

Nakon što je korisnički račun kreiran, aplikacija će otvoriti zaslon sa svim događajima kao na slike 5.6 jer je to osnovna značajka aplikacije i korisniku treba olakšati dolazak do tog zaslona.



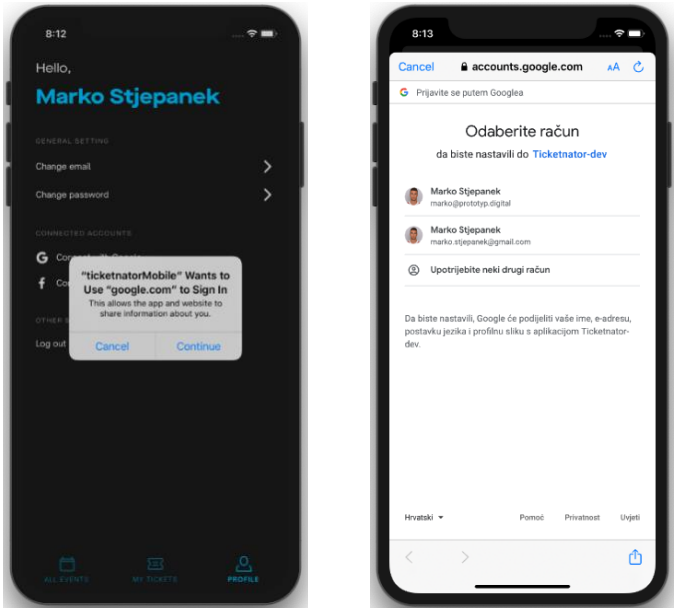
Sl. 5.6. *Početni zaslon nakon prijave/registracije – zaslon sa svim događajima*

Kako bi korisnik vidio postavke svog profila i mogućnosti sa slike 5.7 potrebno je kliknuti na gumb “PROFILE“ u navigaciji.



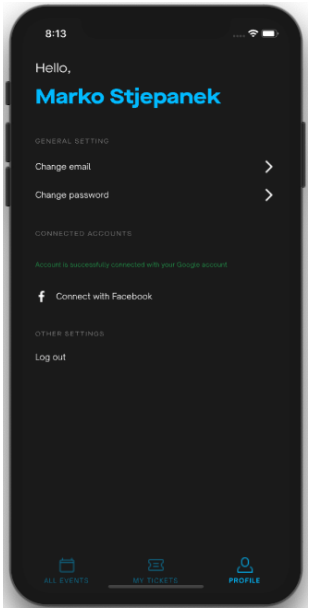
Sl. 5.7. *Zaslon za korisnički račun*

Korisnički račun povezat će se u ovom primjeru s Google računom radi lakših narednih prijava u aplikaciju. Potrebno je potvrditi akciju povezivanja korisničkog računa s Google računom te odabrati Google račun s kojim će se povezati (ako ih ima više dostupnih) kao što je ponuđeno na slici 5.8.



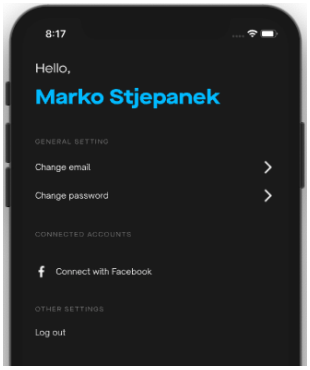
Sl. 5.8. *Povezivanje korisničkog računa s Google računom*

Kako bi korisnik znao da je uspješno povezan korisnički račun s Google računom prikazat će mu se poruka sa slike 5.9, u suprotnom ako je došlo do pogreške prikazat će se određena poruka s pogreškom.



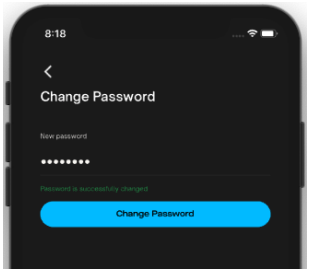
Sl. 5.9. *Poruka o uspješnoj povezanosti s Google računom*

Nakon što se korisnik ponovno vrati na zaslon za korisnički račun više mu se neće prikazivati opcija za povezivanje s Google računom kao na slici 5.10 (za povezivanje s Facebook računom je identični postupak).



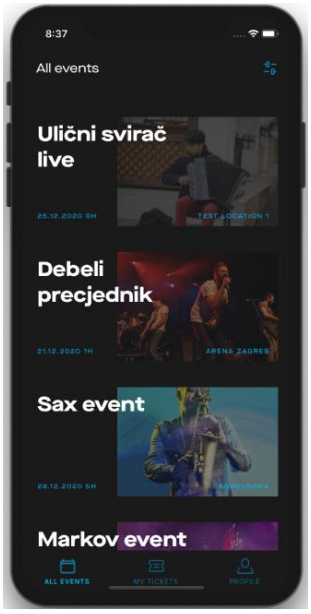
Sl. 5.10. *Na korisničkom profilu nema opcije za povezivanje s Google računom*

Korisnik ima mogućnost promjene e-pošte ili lozinke. Na slici 5.11 promjenila se lozinka kao te će aplikacija prikazati poruku o uspješno obavljenoj akciji.



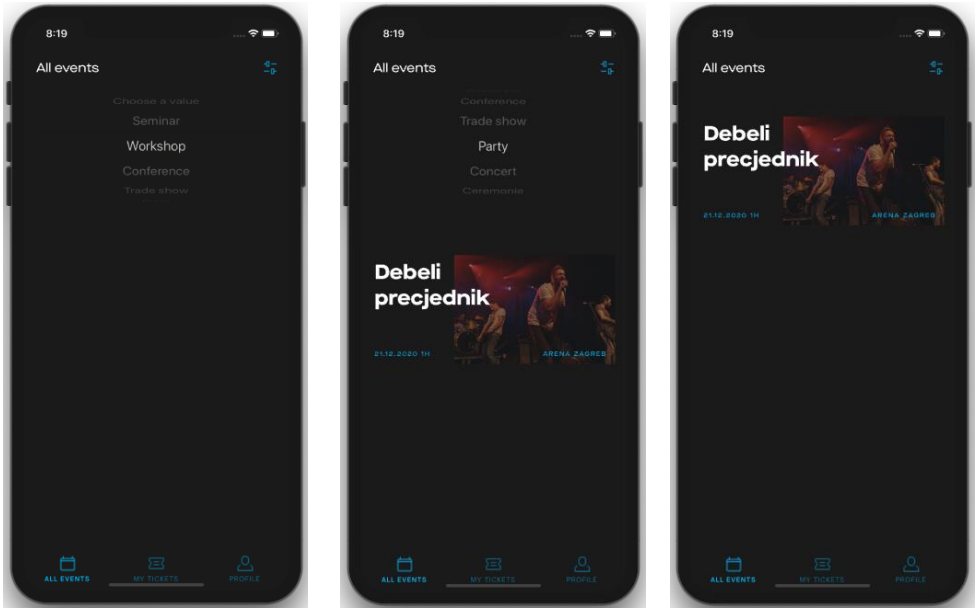
Sl. 5.11. *Uspješna promjena lozinke*

Nakon promjene lozinke potrebno se vratiti nazad na korisnički račun te kliknuti na gumb “ALL EVENTS“ u navigaciji kako bi se dobio zaslon sa svim događajima kao na slici 5.12.



Sl. 5.12. *Zaslon sa svim događajima*

Za lakšu pretragu događaja korisnik može pritisnuti gumb za filter događaja. Događaji će se filtrirati prema određenom tipu događaja kao na slici 5.13. Filter “Choose a value“ prikazat će sve događaje. Ponovnim klikom na gumb za filter zatvorit će se ponuđene filter opcije.

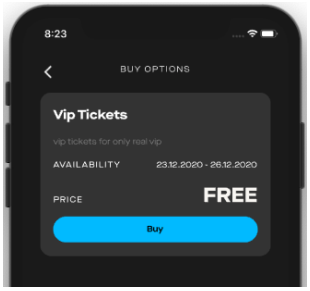


Sl. 5.13. *Filtriranje događaja prema tipu događaja “Party“*

Potrebno je kliknuti na određeni događaj kako bi saznali dodatne detalje o tom događaju. Na detaljima događaja bit će datum, vrijeme, lokacija i opis događaja, što je vidljivo slici 5.14.

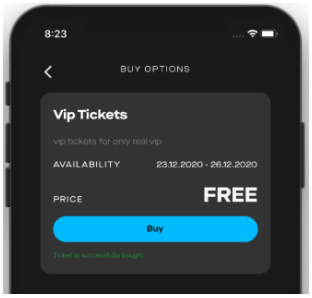
Sl. 5.14. *Detalji događaja*

Za kupovinu ulaznice za određeni događaj potrebno je kliknuti na gumb “Buy options“ gdje će se prikazati ponuđene vrste ulaznica kao na slici 5.15. Svi događaji su beplatni jer je za kupovinu s plaćanjem potrebno imati ugovor s tvrkom koja pruža usluge plaćanja.



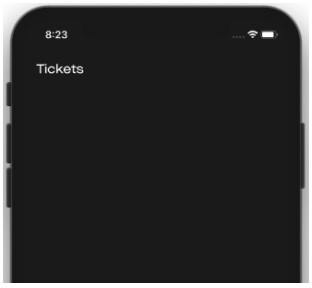
Sl. 5.15. *Vrste ulaznica za događaj*

Nakon kupovine ulaznice, prikazat će se poruka o uspješnoj kupovini kao na slici 5.16, tj. poruka s pogreškom ako nije kupljena ulaznica.



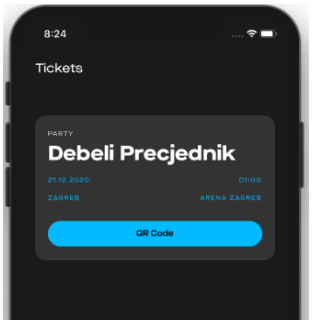
Sl. 5.16. *Uspješna kupovina ulaznice*

Kako bi se koristila kupljena ulaznica potrebno se vratiti natrag na zaslon sa svim događajima te kliknuti gumb u navigaciji “My tickets“. Ako nema kupljenih ulaznica zaslon s ulaznicama bit će prazan kao na slici 5.17.



Sl. 5.17. *Zaslon koji prikazuje da ulaznice još nisu kupljene*

Ako je ulaznica za događaj kupljena, prikazat će se među ulaznicama kao na slici 5.18.



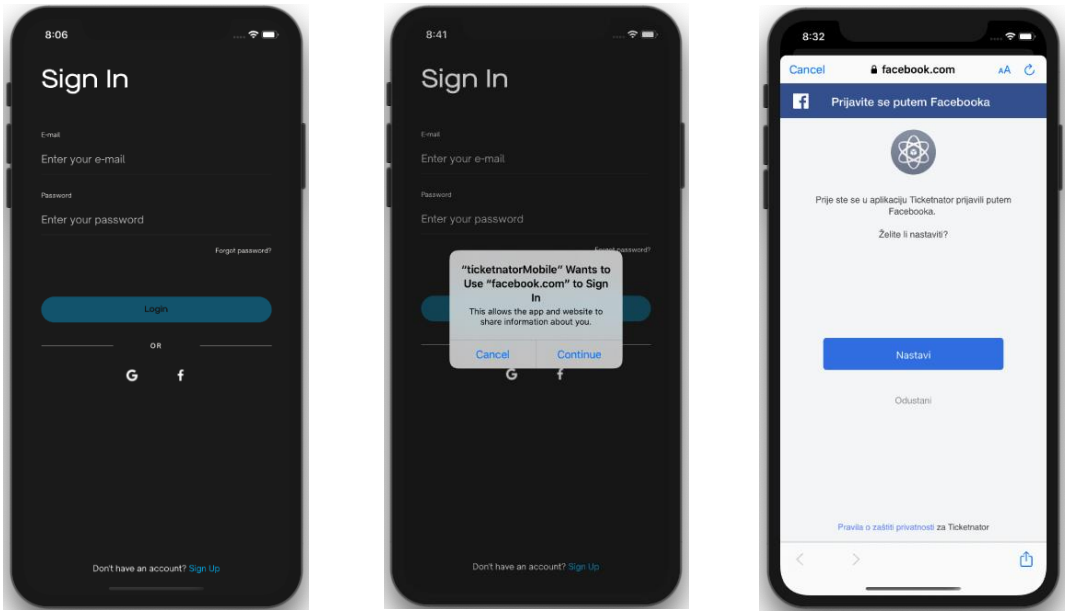
Sl. 5.18. *Zaslon s kupljenim ulaznicama*

Za korištenje ulaznice i dobivanje njenih detalja s QR kodom sa slike 5.19 potrebno je kliknuti na gumb “QR Code“.

Sl. 5.19. *Detalji ulaznice s QR kodom*

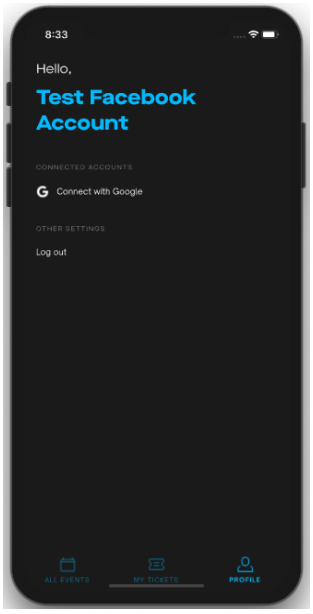
5.2.2. **Korisnički slučaj 2.**

Da bi pokazali registraciju s Facebook računom, pokrenut će se aplikacija i pritisnuti na gumb za prijavu s Facebook računom kao na slici 5.20.



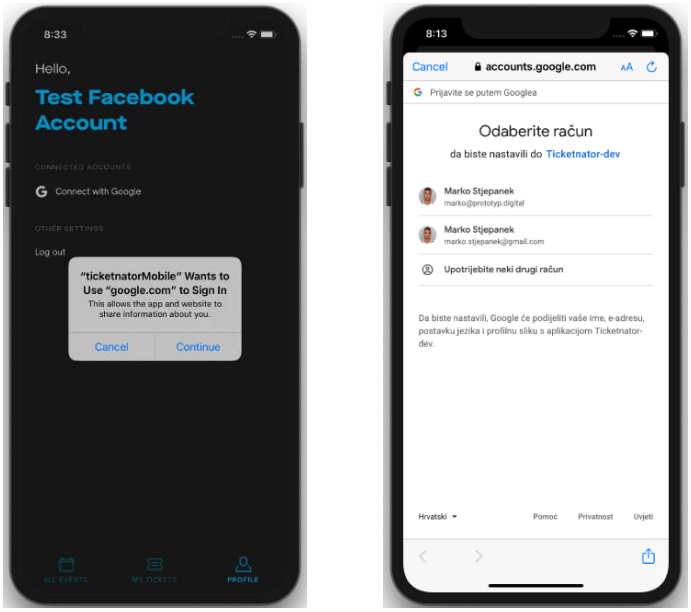
Sl. 5.20. *Pokretanje aplikacije i prijava Facebook računom*

Nakon uspješne prijave, aplikacija će otvoriti zaslon sa svim događajima. Kako bi korisnik vidio postavke svog profila i mogućnosti sa slike 5.21 potrebno je kliknuti na gumb "PROFILE" u navigaciji.



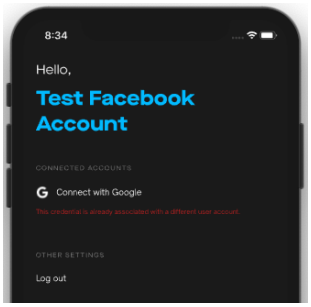
Sl. 5.21. *Zaslon za korisnički račun*

S obzirom na to da je korisnik prijavljen s Facebook računom, nema opcije za promjenu e-pošte i lozinke korisničkog računa. Isto tako nema opciju za povezivanje korisničkog računa s Facebook računom jer se korisnik prijavio s Facebook računom, već samo opciju povezivanja s Google računom kao što se vidi na slici 5.22.



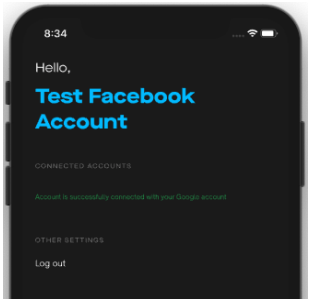
Sl. 5.22. *Mogućnost povezivanja korisničkog računa s Google računom*

Za povezivanje korisničkog računa s Google računom odabran je Google račun koji je već prije povezan s aplikacijom. Aplikacija će pokazati poruku s pogreškom sa slike 5.23 gdje piše da je račun već povezan.



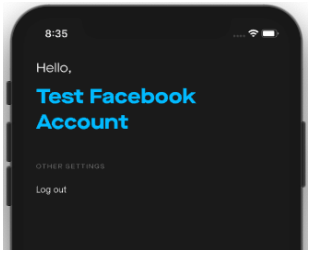
Sl. 5.23. *Poruka o pogrešci povezivanja korisničkog računa s Google računom*

Korisnik će ponovno odabrati gumb za povezivanje s Google računom te odabrati Google račun koji nije povezan s aplikacijom. Korisniku će se prikazat poruka o uspješnom povezivanju korisničkog računa s njegovim odabranim Google računom kao na slici 5.24.



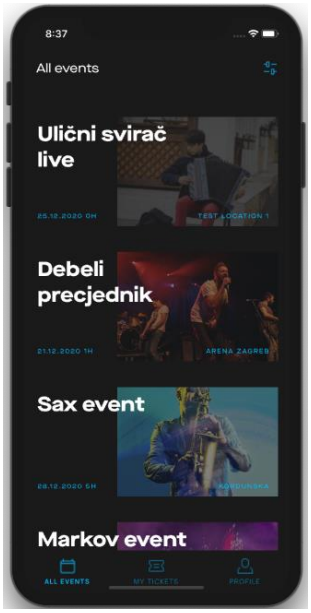
Sl. 5.24. *Poruka o uspješnoj povezanosti Google računom*

Kada bi korisnik ponovno došao na zaslon za korisnički račun imao bi samo opciju odjave iz aplikacije sa slike 5.25.



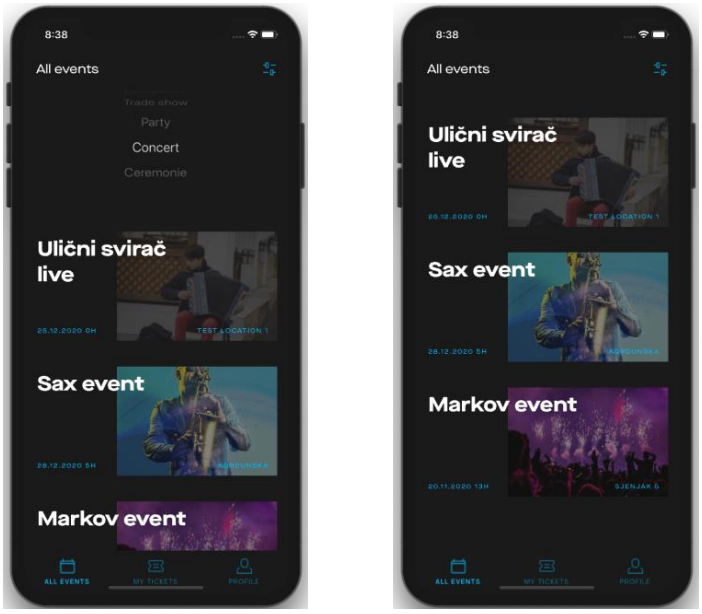
Sl. 5.25. *Zaslon za korisnički račun bez mogućnosti promjene e-pošte i lozinke te povezivanja s Google i Facebook računom*

Kako bi korisnik došao do glavne funkcionalnosti aplikacije – zaslona sa svim događajima sa slike 5.26 potrebno je kliknut na gumb “ALL EVENTS“ u navigaciji.



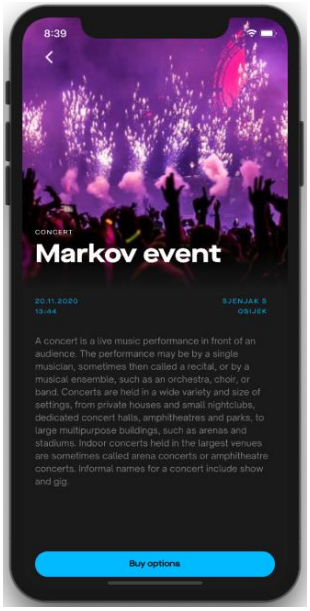
Sl. 5.26. *Zaslon sa svim događajima*

Za lakšu pretragu događaja korisnik pritišće gumb za filter događaja. Odabire filter prema tipu događaja “Concert“ kao na slici 5.27 te ponovno pritišće gumb za filter događaja za zatvaranje ponuđenih opcija.



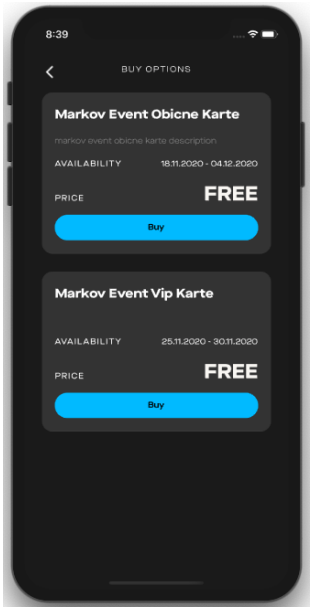
Sl. 5.27. *Filtriranje događaja prema tipu događaja “Concert“*

Potrebno je kliknuti na određeni događaj kako bi se vidjeli dodatni detalji o tom događaju. Na detaljima događaja nalaze se: datum, vrijeme, lokacija i opis događaja kao na slici 5.28.



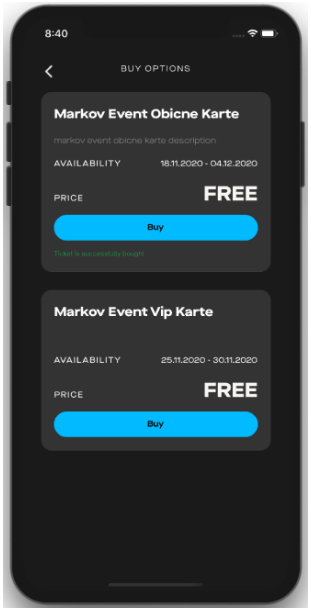
Sl. 5.28. *Detalji događaja “Markov event“*

Za kupovinu ulaznice za događaj potrebno je kliknuti gumb “Buy options“ gdje će se prikazati ponuđene vrste ulaznica za događaj kao na slici 5.29.



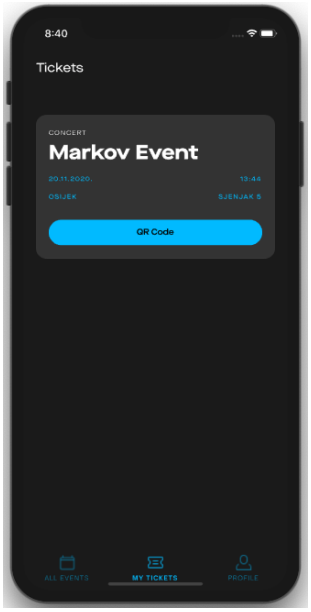
Sl. 5.29. *Vrste ulaznica za događaj "Markov event"*

Korisnik odabire koju vrstu ulaznice želi kupiti. Nakon kupovine ulaznice, pokazat će se poruka o uspješnoj kupovini kao na slici 5.30, tj. poruka s pogreškom ako nije kupljena ulaznica. Poruka će biti ispod vrste ulaznice za koju je obavljena akcija kupovine.



Sl. 5.30. *Uspješna kupovina ulaznice "Markov Event Obične karte"*

Kako bi se koristila kupljena ulaznica potrebno se vratiti natrag na zaslon sa svim događajima te kliknuti gumb u navigaciji "My tickets". Kupljena ulaznica prikazat će se među ulaznicama kao na slici 5.31.



Sl. 5.31. *Zaslon s kupljenim ulaznicama*

Za korištenje ulaznice i dobivanje njenih detalja s QR kodom sa slike 5.32 potrebno je kliknuti na gumb “QR Code“.

Sl. 5.32. *Detalji ulaznice s QR kodom*

5.3. Rezultati ispitivanja rada mobilne aplikacije Ticketnator

U prošlom poglavlju je kroz dva različita korisnička slučaja ispitan rad višeplatformske mobilne aplikacije Ticketnator. Rad aplikacije je ispitan pomoću virtualnog uređaja iPhone 11 za operacijski sustav iOS , virtualnog uređaja LG Nexus 5x i stvarnog uređaja Samsung Galaxy s9+ za operacijski sustav Android. Ispitivanje rada je bilo potrebno kako bi se vrednovala aplikacija prema osnovnim zahtjevima kao što su sinkronizacija, vremensko ponašanje i uspješnost rukovanja događajima. Aplikacija ima sinkronizacijske i vremenske zahtjeve jer je sustav strogo događajima orijentiran. Uspješno je postignuta sinkronizacija što je vidljivo gledajući usporedno bazu podataka s testnim uređajem na kojemu se određena akcija dogodi te se baza podataka sinkronizira prema toj akciji. S time je dokazano i zadovoljavanje vremenskih zahtjeva tj. vremenska usklađenost prilikom sinkronizacije u stvarnom vremenu. Za uspješno rukovanje događajima, tj. rukovanje svim akcijama u mobilnoj aplikaciji Ticketnator, zaduženi su Firebase *WebSocket listeneri* koji će obavijestiti klijenta da se dogodio neki događaj. Detaljnom analizom ispitivanja rada aplikacije pokazalo se kako su zadovoljeni svi funkcionalni i nefunkcionalni zahtjevi koji su navedeni u potpoglavlju 3.1. Nefunkcionalni zahtjevi su ostvareni korištenjem višeplatformske arhitekture okvira baziranih na preslikavanju, pomoću mikrousluga i Websocket listenerima. Postignućem svih zadanih funkcionalnosti može se vidjeti kako je programski okvir React Native uspješan za višeplatformski razvoj aplikacija u kojima je bitna sinkronizacija podataka u stvarnom vremenu i rukovanje događajima. Višeplatformski razvoj je na početku teži od nativnog razvoja jer je potrebno puno različitog znanja i alata za razvoj, ali nakon što se postavi sve potrebno za rad, krivulja učenja i razvoja je eksponencijalna. Nativnim razvojem mobilne aplikacije Ticketnator za operacijski sustav iOS i Android, aplikacija bi dobila poboljšanje performansi, što prosječni korisnik vjerojatno ne bi primjetio, ali razvoj dvije nativne aplikacije bi bilo puno skuplje i vremenski duže.

Mobilnom aplikacijom Ticketnator korisnik dobiva moderno sučelje aplikacije gdje brzom prijavom putem Google računa, Facebook računa ili korisničkog računa putem e-pošte može pregledati dostupne događaje i kupiti ulaznicu za željeni događaj. U aplikaciji korisnik ima pretragu događaja filtriranjem prema vrsti događaja. Kada korisnik želi koristiti kupljenu ulaznicu za određeni događaj, aplikacija stvara QR kod s detaljima ulaznice za njezino korištenje. Za uspješnije korištenje aplikacije i bolji korisnički doživljaj, korisnik za svaku izvršenu akciju prima poruku s potvrdom tj. poruku s pogreškom kod neuspjele akcije.

Korisniku je dostupno korištenje ulaznica i lokalno skladištenje podataka u trenutku kad je nedostupna internetska veza, ako je korisnik već bio prijavljen u aplikaciju. Sve nove akcije koje korisnik odradi u trenutku kad nema internetsku vezu odradit će se kad internetska veza ponovno postane dostupna.

Iako aplikacija koristi najnovije tehnologije današnjice s modernim dizajnom kako bi postala konkurentnija trenutnim popularnim aplikacijama potrebna bi bila nadogradnja aplikacije s dodatnim značajkama. Neke od tih značajki su: dodavanje kontakata i pregled njihovih javno dostupnih ulaznica, poklon bon za kupovinu ulaznica, prebacivanje ulaznice s jednog korisničkog računa na drugi, sniženja ulaznica, prijava s Apple računom itd. Isto tako aplikacija se trenutno može koristiti samo za besplatne događaje jer bi za usluge plaćanja potrebno bilo potpisati ugovor s određenim pružateljem usluga plaćanja. Od analiziranih sličnih već postojećih rješenja mobilna aplikacija Ticketnator najsličnija je mobilnoj aplikaciji Bizzabo.

6. ZAKLJUČAK

U radu se ustanovilo da se višeplatformski razvoj koristi kako bi se brže i jeftinije napravila mobilna aplikacija, dok će nativnim razvojem aplikacija biti brža i sigurnija, ali će razvoj biti skuplji i sporiji. Kako bi se dobila višeplatformska aplikacija događa se *transpiling* koda napisanog u programskom jeziku Typescript u kod napisan u programskom jeziku JavaScript. Zatim se koristi most koji pruža dvosmjernu komunikaciju između JavaScript i nativnog dijela aplikacije. U praktičnom dijelu rada za izradu višeplatformske aplikacije koristi se programski okvir React Native, koji koristi arhitekturu okvira baziranog na preslikavanju što je aplikaciji dalo izgled i osjećaj nativne mobilne aplikacije. Izrađena mobilna aplikacija Ticketnator s modernim sučeljem ima mogućnosti upravljanja korisničkim računom, pregled i pretragu događaja, kupovinu ulaznice za događaj i korištenje ulaznice za kupljeni događaj. Za ispunjavanje nefunkcionalnih zahtjeva aplikacije korištena je mikrouslužna arhitektura Firebase platforme. Firebase platforma je aplikaciji omogućila korištenje samostalnih paketa usluga kao što su baza podataka u stvarnom vremenu, autentikacija korisnika i pohrana u oblaku. U aplikaciji za sinkronizaciju podataka u stvarnom vremenu koriste se *WebSocket listeneri* s Firebase bazom podataka. Kako bi se omogućilo skladištenje podataka u nedostatku mrežne veze postavljen je spremnik stanja Redux koji uz Firebase-ovu podršku lokalno skladišti podatke.

Analiziranjem rezultata ispitivanja rada mobilne aplikacije Ticketnator pomoću virtualnih i stvarnih uređaja zaključeno je da je aplikacija funkcionalna te zadovoljava sve zadane funkcionalne i nefunkcionalne zahtjeve. Isto tako ispunjava sinkronizacijske, vremenske zahtjeve i uspješno rukuje događajima. Svojim dizajnom i performansama može konkurirati nativnim aplikacijama slične tematike. Kako bi se aplikacija Ticketnator koristila u stvarnom svijetu potrebno je u budućnosti dodati pružatelja usluga plaćanja jer su trenutno svi događaji u aplikaciji besplatni. Za daljnje poboljšanje aplikaciju moguće je napraviti nadogradnju s novim značajkama za poboljšano korisničko iskustvo u aplikaciji.

LITERATURA

[1] Net Solutions, „*Where Do Cross-Platform App Frameworks Stand in 2020?*“, https://www.netsolutions.com/insights/cross-platform-app-frameworks-in-2019/, 06.12.2019., pristup ostvaren 30.06.2020.

[2] Hackernoon, „*What's revolutionary about flutter*“, https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514, pristup ostvaren 17.1.2021.

[3] Litslink, „*Best Cross-Platform Mobile Development Tools in 2020*“, https://litslink.com/blog/best-cross-platform-mobile-development-tools-in-2020/, 26.03.2020., pristup ostvaren 30.06.2020.

[4] HowToDoInJava, „*Transpiler vs Compiler*“, https://howtodoinjava.com/typescript/transpiler-vs-compiler/, 26.03.2020., pristup ostvaren 1.07.2020.

[5] Stefanov, Stoyan, “*JavaScript Patterns: Build Better Applications with Coding and Design Patterns* ”, O'Reilly Media, Inc., 2010.

[6] Anthony Giretti's .NET blog, *„No, TypesScript is not compiled into JavaScript* , https://anthonygiretti.com/2017/06/04/no-typescript-is-not-compiled-into-javascript/, pristup ostvaren 1.07.2020.

[7] Medium, *„Understanding Babel as a native developer working with React Native*“, https://medium.com/@flexaddicted/understanding-babel-as-a-native-developer-dived-into-react-native-8cbf632318af/, 23.9.2018., pristup ostvaren 9.07.2020.

[8] Hackernoon, *„Understanding the React Native bridge concept*“, https://hackernoon.com/understanding-react-native-bridge-concept-e9526066ddb8/, 10.11.2017., pristup ostvaren 9.07.2020.

[9] Dudjak, M., Martinović, G. (2020). „*An API-first Methodology for Designing a Microservice-based Backend as a Service Platform*.“ Information Technology and Control, 49(2), 206-223. https://doi.org/10.5755/j01.itc.49.2.23757

[10] Smartbear, *„What is Microservices?“* , https://smartbear.com/solutions/microservices/, pristup ostvaren 5.07.2020.

[11] Clockwise, „*Monolithic architecture vs microservices comparison*“, https://clockwise.software/blog/monolithic-architecture-vs-microservices-comparison/, pristup ostvaren 22.1.2021.

[12] Newrelic, *„Microservices Architectures: What they are and why you should use them“* , https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/, pristup ostvaren 6.07.2020.

[13] Ivan Jakab, “*Sustav komunikacije u stvarnom vremenu* ”, Sveučilište Josipa Jurja Strossmayera u Osijeku Fakultet elektrotehnike, računarstva i informacijskih tehnologija, 2019.

[14] Ollyxar, *„Websockets“* , https://ollyxar.com/websockets/, pristup ostvaren 1.07.2020.

[15] Firebase, *„Get realtime updates with Cloud Firestore“,* , https://firebase.google.com/docs/firestore/query-data/listen/, pristup ostvaren 5.07.2020.

[16] Firebase, *„Usage and limits“* , https://firebase.google.com/docs/firestore/quotas/, pristup ostvaren 5.07.2020.

[17] Firebase, *„Understand Cloud Firestore billing“*, https://firebase.google.com/docs/firestore/pricing#europe/, pristup ostvaren 5.07.2020.

[18] Gramoli, V., „*More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorythms“,* ACM SIGPLAN Notices, 2015.

[19] L. Jünger, C. Bianco, K. Niederholtmeyer, D. Petras, R. Leupers, „*Optimizing Temporal Decoupling using Event Relevance*“, Proc. of the 26th Asia and South Pacific Design Automation Conference, January 2021, pp 331–337.

[20] G. Martinović, L. Budin, Ž. Hocenski, Željko, „*Undergraduate Teaching of Real-Time Scheduling Algorithms by Developed Software Tool*“, IEEE Transactions on Education, 46 (2003), 1; 185-196.

[21] Liu, Jane W. , *Real-Time Systems,* Integre Technical Publishing Co., Inc., 2000.

[22] Z. Stapić, N. Vrček, „*Izazov sinkronizacije podataka između mobilnih i standardnih baza podataka“,* Rijeka: Case d.o.o., 2010., 107-112.

[23] Entrio, *„O Entriu“*, https://www.entrio.hr/, pristup ostvaren 7.07.2020.

[24] Eventim, *„O Eventim“*, https://www.eventim.hr/hr/about_oet/, pristup ostvaren 7.07.2020.

[25] TicketTailor, *„Product“*, https://www.tickettailor.com/product/, pristup ostvaren 7.07.2020.

[26] Bizzabo, *„How Bizzabo compares to legacy event softwaret“*, https://www.bizzabo.com/compare/, pristup ostvaren 7.07.2020.

[27] Ploxel, *„About“*, https://www.ploxel.com/, pristup ostvaren 10.7.2020.

[28] Mobindustry, „*Writing Clear Functional and Non-functional Requirements: Examples and Best Practices*“, https://www.mobindustry.net/writing-clear-functional-and-non-functional-requirements-examples-and-best-practices/, pristup ostvaren 10.1.2021.

[29] Magora Systems, „*Mobile app development architecture*“, https://magora-systems.com/mobile-app-development-architecture/, pristup ostvaren 19.1.2021.

[30] AppVelocity, „*Guide mobile application architecture*“, https://www.appvelocity.ca/blog/guide-mobile-application-architecture, pristup ostvaren 20.1.2021.

[31] AppInventiv, „*Mobile app architecture explained*“, https://appinventiv.com/blog/mobile-app-architecture-explained/, pristup ostvaren 20.1.2021.

[32] M. Martinez, S. Lecomte, „*Towards the Quality Improvement of Cross-platform Mobile Applications*“, Proc. of the 4th International Conference on Mobile Software Engineering and Systems, May 2017, pp 184–188.

[33] Microsoft, „*Cross-Platform App Case Study*“, https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/case-study-tasky, pristup ostvaren 20.1.2021.

[34] Elprocus, *„What everybody ought to know about Android: introduction, features & applications“*,https://www.elprocus.com/what-is-android-introduction-features-applications/, pristup ostvaren 12.07.2020.

[35] Developer Android, „*System architecture*“, https://developer.android.com/images/system-architecture.jpg, pristup ostvaren 17.1.2021.

[36] Android, *„Android 10 Highlights“*, https://www.android.com/android-10/ , pristup ostvaren 12.07.2020.

[37] Lifewire, *„The History of iOS, from Version 1.0 to 13.0“*, https://www.android.com/android-10/ , 11.3.2020. pristup ostvaren 12.07.2020.

[38] Tutorialspoint, *„Apple iOS Architecture“*, https://www.tutorialspoint.com/apple-ios-architecture/ . pristup ostvaren 12.07.2020.

[39] Intellipaat, „*iOS Architecture*“, https://intellipaat.com/blog/tutorial/ios-tutorial/ios-architecture/, pristup ostvaren 17.1.2021.

[40] Apple, *„iOS 13“*, https://www.apple.com/hr/ios/ios-13/ . pristup ostvaren 12.07.2020.

[41] Engineering, „*React Native bringing modern web techniques to mobile*“, https://engineering.fb.com/2015/03/26/android/react-native-bringing-modern-web-techniques-to-mobile/, pristup ostvaren 18.1.2021.

[42] Litslink, „*New React Native architecture*“, https://litslink.com/blog/new-react-native-architecture, pristup ostvaren 19.1.2021.

[43] Visual Studio Code, „*About Visual Studio Code*“, https://code.visualstudio.com/ . pristup ostvaren 13.07.2020.

[44] Stack Overflow, „*Developer Survey Results 2019*“, https://insights.stackoverflow.com/survey/2019#development-environments-and-tools/ . pristup ostvaren 13.07.2020.

[45] Medium, „*What is firebase – the complete story*“, https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0, pristup ostvaren 23.1.2021.

[46] Firebase,“ *Products*“, https://firebase.google.com/products/, pristup ostvaren 6.07.2020.

[47] Firebase, „*Access data offline*“, https://firebase.google.com/docs/firestore/manage-data/enable-offline, pristup ostvaren 27.1.2021.

[48] Prototyp blog, „*Avoding the messy git history*“, https://blog.prototyp.digital/avoiding-the-messy-git-history/ , 22.3.2019. pristup ostvaren 13.07.2020.

[49] LogRocket, „*Why use redux? Reasons with clear examples*“, https://blog.logrocket.com/why-use-redux-reasons-with-clear-examples-d21bffd5835/, pristup ostvaren 28.1.2021.

[50] ReduxJS, „*About*“, https://redux.js.org/, pristup ostvaren 28.1.2021.

[51] Medium, „*App Architecture and Folder setup for React native App development*“, https://sandeeprajbhartechgeek.medium.com/app-architecture-setup-for-react-native-app-development-a2bb4c4fcbde, pristup ostvaren 21.1.2021.

[52] Bitrise, „*Bitrise Docs*“, https://devcenter.bitrise.io/ , pristup ostvaren 14.07.2020.

[53] Figma, „*Design*“, https://www.figma.com/design/ , pristup ostvaren 13.07.2020.

SAŽETAK

VIŠEPLATFORMSKI RAZVOJ MOBILNIH APLIKACIJA SA SINKRONIZACIJOM U STVARNOM VREMENU

U ovom diplomskog rada analizirani su izazovi višeplatformskog razvoja mobilnih aplikacija za iOS i Android platforme s posebnim osvrtom na tehnologiju React Native. Opisani su načini pretvorbe programskog koda, mikrousluge, rad s korisničkim računima, korištenje socketa i lokalno skladištenje podataka pri nedostatku mreže. U praktičnom dijelu rada izrađena je višeplatformska mobilna aplikacija za pretragu i prijavu na događaje uz zahtjev za sinkronizacijom u stvarnom vremenu Ticketnator. Aplikacija je ostvarena korištenjem programskog okvira React Native i razvojnog okruženja Visual Studio Code. Za potrebe sustava na strani poslužitelja koristi se platforma Firebase kao baza podataka, sustav za autentikaciju i korištenje aplikacije u nedostatku internetske veze. Za lokalno skladištenje podataka koristi se spremnik stanja Redux. Ispitivanjem aplikacije pokazano je da aplikacija zadovoljava sve funkcionalne i nefunkcionalne zahtjeve s naglaskom na sinkronizacijske i vremenske zahtjeve te uspješno upravljanje događajima. Aplikacija Ticketnator može konkurirati trenutno dostupnim aplikacijama za pretragu i prijavu na događaje.

Ključne riječi: događaji, mikrousluge, React Native, sinkronizacija u stvarnom vremenu, višeplatformski razvoj, websocket listener

ABSTRACT

CROSS-PLATFORM DEVELOPMENT OF MOBILE APPLICATIONS WITH REAL-TIME SYNCHRONIZATION

The master thesis analysis the challenges of cross-platform development of mobile applications for iOS and Android platforms with special reference to React Native technology. Methods of program code conversion, microservices, working with user accounts, using sockets, and local data storage in the absence of a network are described. In the practical part of the paper, the cross-platform mobile application for searching and event registration with a request for real-time synchronization called Ticketnator was accomplished. The application was created using the React Native application framework and the Visual Studio Code source-code editor. For the needs of the server-side system, the Firebase platform is used as a database, authentication, and accessing data offline. A Redux state container is used for local data storage. Testing the application has shown that the application meets all functional and non-functional requirements with an emphasis on synchronization, time requirements, and successful event handling. The Ticketnator application can compete with currently available applications for searching and event registration.

Keywords: events, microservices, React Native, real-time synchronization, cross-platform development, websocket listener

ŽIVOTOPIS

Marko Stjepanek rođen je 16. listopada 1996. godine u Osijeku. Nakon završene III. Gimnazije u Osijeku, 2015. godine ostvaruje upis na preddiplomski sveučilišni studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija, tadašnji Elektrotehnički fakultet u Osijeku. 2018. godine stječe akademski naziv sveučilišni prvostupnik (lat. *baccalaureus*) inženjer računarstva. Iste godine upisuje diplomski sveučilišni studij Programsko inženjerstvo. Od 2019. godine radi u tvrtki PROTOTYP d.o.o. kao programer web i mobilnih aplikacija.

Potpis:

_____

PRILOZI (NA CD-U):

Prilog 1. Dokument i pdf diplomskog rada

Prilog 2. Aplikacija za operacijski sustav iOS u .ipa formatu

Prilog 3. Aplikacija za operacijski sustav Android u .apk formatu