Android aplikacija za promicanje i olakšavanje sudjelovanja javnosti u rješavanju problema sa zelenilom grada

Mijić, Matej

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

Sveučilišni diplomski studij

ANDROID APLIKACIJA ZA PROMICANJE I OLAKŠAVANJE SUDJELOVANJA JAVNOSTI U RJEŠAVANJU PROBLEMA SA ZELENILOM GRADA

Diplomski rad

Matej Mijić

Osijek, 2020.

SADRŽAJ

1. UVOD

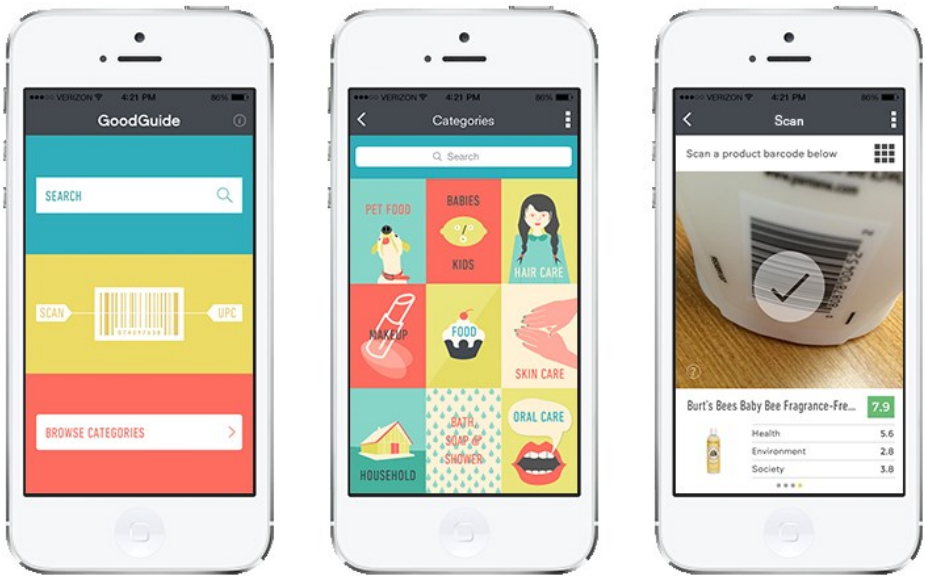
U današnje vrijeme pametni telefoni postali su svakodnevica. Trenutno samo na *Google Play* trgovini ima registrirano preko dva milijuna aplikacija [26]. Većina tih aplikacija nastoji olakšati rješavanje problema s kojima se čovjek svakodnevno susreće. Jedan od tih problema je očuvanje okoliša odnosno životnog prostora. Zadatak ovog diplomskog rada je izrada *Android* aplikacije koja omogućava građanima i zaposlenicima komunalnog poduzeća suradnju u rješavanju problema urednosti grada. Aplikacija je nazvana Zelenko. Građani putem aplikacije Zelenko mogu uslikati uočeni problem te poslati sliku komunalnom poduzeću s lokacijom i detaljnim opisom. Time se postiže lakše i brže informiranje komunalnog poduzeća o mogućim potrebama za intervencijom vezano za urednost grada i optimiziraju se aktivnosti komunalnog poduzeća što omogućava kvalitetnije i pravovremeno rješavanje problema. Cilj aplikacije Zelenko je i podizanje razine ekološke svijesti građana te motiviranje građana da se uključe u rješavanje lokalnih ekoloških problema. Ivan Cifrić u svojoj knjizi Socijalna Ekologija govori da "ekološka svijest pretpostavlja spoznaju o stanju društva i prirode i o uzrocima tog stanja; spoznaju da treba zaštititi prirodu od daljnjeg propadanja upotrebom preventivnih i kurativnih sredstava" [28]. Da bi se riješio ekološki problem društvo mora biti svjesno da taj problem uopće postoji i da ga se tiče [27]. Svaki pojedinac utječe na poboljšanje ili pogoršanje ekoloških prilika, a isto tako svaka ekološka promjena utječe na svakog pojedinca [27]. Stoga je bitno da je svaki pojedinac svjestan svog utjecaja na okoliš jer i najmanjim djelima utječe na njegovu promjenu. U prvom dijelu rada opisane su aplikacije iste ili slične svrhe. U drugom dijelu rada objašnjene su tehnologije koje su korištene prilikom izrade aplikacije te detaljan postupak primjene tih tehnologija prilikom razvoja. Na kraju, u trećem dijelu rada, opisan je način korištenja aplikacije.

2. APLIKACIJE SA SVRHOM ZAŠTITE OKOLIŠA

Razina ekološke svijesti ljudi je svakim danom sve veća i ljudi sve više shvaćaju kako je uloga čovjeka u zaštiti okoliša bitna. Svakodnevne aktivnosti čovjeka utječu na okoliš, a iz tog razloga programeri su stvorili niz aplikacija koje pomažu čovjeku da obrati pozornost na te aktivnosti. Cilj tih aplikacija je pomoći čovjeku da shvati na koji način može sačuvati okoliš i koje korake može poduzeti svaki dan kako bi njegove svakodnevne aktivnosti postale ekološki prihvatljive. U nastavku su uspoređene i opisane aplikacije koje spašavaju okoliš.

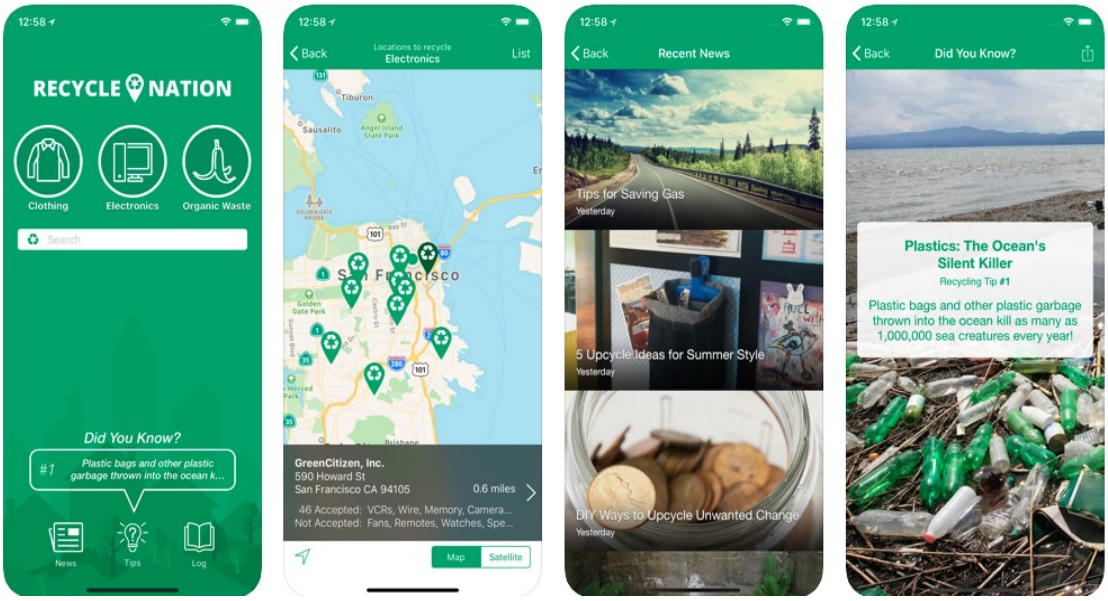
2.1. Primjer aplikacija koje spašavaju okoliš

GoodGuide je aplikacija koju je razvio tim znanstvenika s *Berkeley* sveučilišta [1]. Aplikacija sadrži ocjene za preko pet stotina tisuća različitih proizvoda. Ti proizvodi se svrstavaju u različite kategorije kao što su hrana, zdravlje, kozmetička pomagala, hrana za životinje i slično. Ocjena pojedinog proizvoda predstavlja utjecaj tog proizvoda na okoliš. Korisnik kamerom pametnog telefona skenira proizvod i zatim se prikazuje lista informacija o tom proizvodu i proizvođaču te ukupna ocjena. Korisnici također mogu davati ocjene proizvodima i pisati recenzije. Cilj aplikacije je pomoći kupcu da odabere proizvod koji je ekološki prihvatljiv. Ova aplikacija ne komunicira direktno s komunalnim poduzećem grada poput aplikacije Zelenko, ali svakako pomaže u zaštiti okoliša i nastoji motivirati ljude da sudjeluju u zaštiti. Na slici 2.1. prikazana je *GoodGuide* aplikacija.



Slika 2.1. *Prikaz GoodGuide aplikacije [1]*

Na slici 2.2. prikazana je *RecycleNation* aplikacija [2]. *RecycleNation* aplikacija svojim korisnicima omogućava lakše recikliranje smeća. Aplikacija sadrži bazu podataka lokacija na kojima se može reciklirati smeće. Pomaže korisniku pronaći zeleni otok na kojem se može reciklirati određeni predmet. Nakon recikliranja korisnik ima uvid u resurse koje je spasio prilikom recikliranja.

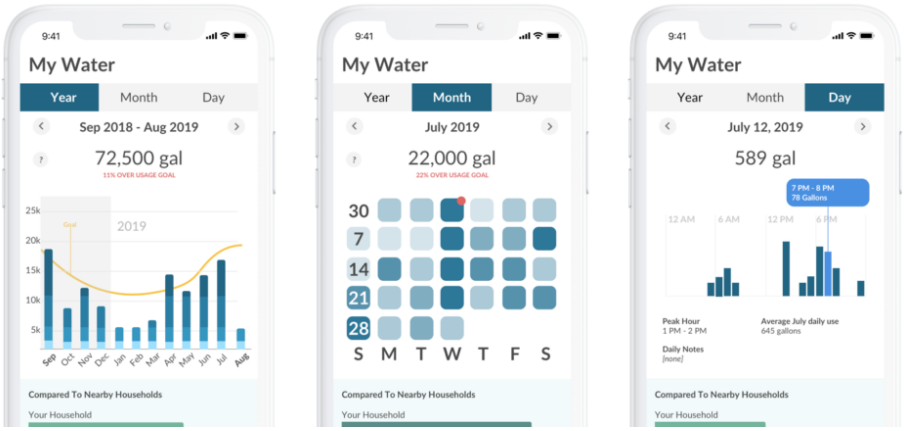


Slika 2.2. *Prikaz RecycleNation aplikacije [2]*

Oroeco [31] je aplikacija koja ima nastoji podići razinu svijesti o utjecaju jedne osobe na klimatske promjene, globalno zatopljenje i druge posljedične pojave. Aplikacija funkcionira kao precizni kalkulator koji prati različite čimbenike poput ugljičnog otiska pojedinca, utjecaja na klimu, konzumiranu hranu i slično. Aplikacija svakodnevno daje korisne savjete koji pomažu da se pojedinac promijeni na bolje.

Refill [32] je zanimljiva aplikacija koja pomaže riješiti problem s pretjeranim korištenjem plastike. *Refill* aplikacija potiče korištenje boca za višekratnu uporabu te pomaže pronaći najbližu lokaciju gdje se može napuniti voda.

Dropcountr je aplikacija koja je možda najsličnija aplikaciji Zelenko jer omogućava komunikaciju između vodovoda i građana [3]. Vodovod šalje korisnicima obavijesti o potrošnji vode ili ako je došlo do puknuća cijevi i slično. Korisnik može pratiti potrošnju vode tijekom dana, tjedna, mjeseca ili godine. Cilj ove aplikacije je pomoći korisniku da optimizira potrošnju vode, a omogućava i lakšu suradnju između građana i vodovoda. Korisnik također može usporediti svoju potrošnju s drugim korisnicima. Slika 2.3. prikazuje izgled *Dropcountr* aplikacije.



Slika 2.3. *Prikaz Dropcountr aplikacije [3]*

3. TEHNOLOGIJE KOJE SE KORISTE ZA RAZVOJ APLIKACIJE

Za izradu zadatka koristi se više tehnologija kako bi se ostvario željeni rezultat. Za razvoj same aplikacije koristi se *React Native* [4]. To je okvir otvorenog koda za izradu *Android*, *iOS* i Web aplikacija kreiran od strane *Facebook-a*. Aplikacija se piše u programskom jeziku *TypeScript*, a za uređivanje koda koristi se *Visual Studio Code* (kasnije u tekstu skraćeno *VSC*) razvojno okruženje [5, 6]. Verziranje koda vodi se pomoću *Git-a* na *GitHub-u*, a kao baza podataka koristi se *Firebase* [7, 8, 9]. Prilikom razvoja implementira se i kontinuirana integracija i isporuka aplikacije na *Bitrise-u*, što omogućava lakše testiranje i davanje aplikacije na uvid klijentima [11].

3.1. Programski jezik TypeScript

Aplikacijski kod piše se u programskom jeziku *TypeScript*. To je programski jezik otvorenog koda koji koristi istu sintaksu i semantiku kao i *JavaScript*, ali razlika je u tome što se kod *TypeScript-a* definiraju tipovi podataka. Koriste ga okviri za izradu aplikacija kao što su *Angular*, *React*, *Vue*, *Ruby on Rails*, *ASP.NET Core* i mnogi drugi. Prednost *TypeScript-a* je to što se prevodi u *JavaScript*, što znači da se može pokretati na *Node.js-u* ili bilo kojem pregledniku koji minimalno mora podržavati *ECMAScript 3* programski jezik. Na slici 3.1. prikazana je deklaracija tri iste varijable u *JavaScript-u* i *TypeScript-u*

```javascript
// Javascript

let godinaRodenja;
let ime;
let lista;
```

```typescript
// TypeScript

let godinaRodenja: number;
let ime: string;
let lista: string[];
```

Slika 3.1. *Usporedba deklaracija varijabli u JavaScript-u i TypeScript-u*

Nakon deklaracije varijabli u *TypeScript-u* zna se točno da je godina rođenja broj, ime niz znakova, a lista polje niza znakova. Kod deklaracija u *JavaScript-u* to nije definirano i ništa ne sprječava programera da za vrijednost varijable ime stavi broj koji je tipa *number*. Definiranje tipova

podataka omogućava izbjegavanje grešaka koje se često susreću ukoliko se kod piše u čistom *JavaScript-u*. *TypeScript* kod je čitak i mogu se brzo tražiti i otklanjati pogreške. Neki od osnovnih tipova podataka su *boolean*, *number*, *string*, *array*, *enum*, *any*, *null*, *object*. Podržava i statičku provjeru što znači da će se greška u kodu pronaći automatski prije nego što se program pokrene.

Jedna od funkcionalnosti *TypeScript-a* je i mogućnost korištenja sučelja. Na slici 3.2. prikazan je primjer sučelja.

```
interface Knjiga {
  id: number;
  brojStrana: number;
  autor: string;
  tema: string;
}
```

Slika 3.2. *Primjer sučelja*

Putem sučelja programer može definirati svoje tipove podataka. Na slici 3.2. definirano je sučelje Knjiga. To sučelje sadrži četiri svojstva: identifikacijski broj, broj strana, autora i temu. Ako se kreira novi objekt koji je tipa Knjiga, to znači da taj objekt mora imati definirana ova četiri svojstva. Na sučelja se može gledati kao na ugovore koji moraju biti zadovoljeni. Na slici 3.3. prikazana je inicijalizacija objekta tipa Knjiga kojem nisu definirana svojstva. *TypeScript* u tom slučaju odmah prijavljuje grešku jer takozvani ugovor koji je potpisao objekt Stranac nije ispunjen. Na slici 3.4. prikazana je pravilna inicijalizacija tog objekta.

```
const Stranac: Knjiga
Type '{}' is missing the following properties from type 'Knjiga': id, brojStrana, autor, tema ts(2739)
Peek Problem (Alt+F8)   No quick fixes available
const Stranac: Knjiga = {};
```

Slika 3.3. *Inicijalizacija objekta tipa Knjiga bez definiranja svojstava*

```
const Stranac: Knjiga = {
  id: 1,
  brojStrana: 136,
  autor: 'Albert Camus',
  tema: 'Svakodnevna monotonost',
};
```

Slika 3.4. *Pravilna inicijalizacija objekta tipa Knjiga*

Na slici 3.5. vidi se primjer definiranja i pozivanja *TypeScript* funkcije. Definirana je jednostavna funkcija koja uzima dva broja i kao rezultat vraća njihov zbroj.

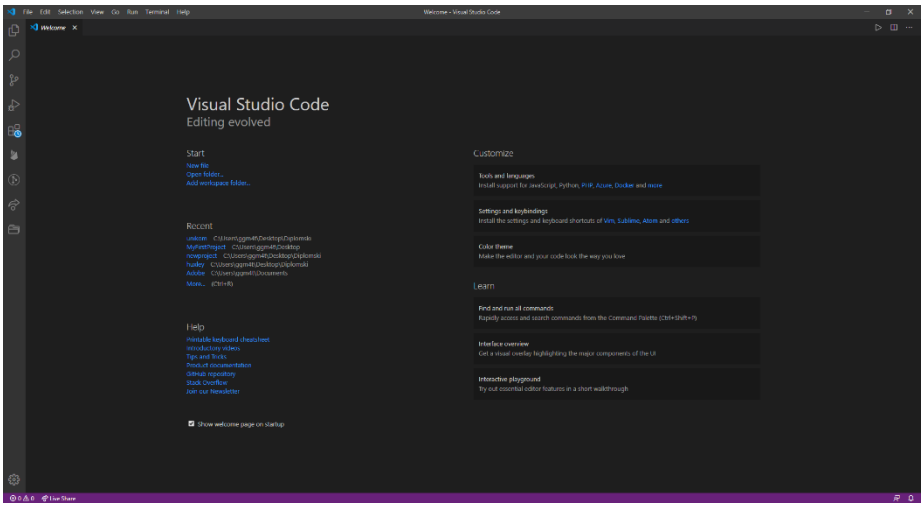
```
function zbrojiDvaBroja(prviBroj: number, drugiBroj: number) {
  return prviBroj + drugiBroj;
}
       const rezultat: number
const rezultat = zbrojiDvaBroja(2, 2);
```

Slika 3.5. *Primjer TypeScript funkcije*

Kod pisanja parametara funkcije se također moraju definirati tipovi podataka. No, kod funkcije na slici 3.5. nije definiran tip koji ta funkcija vraća. To je još jedno odlično svojstvo *TypeScript-a*. *TypeScript* sam prepoznaje da ako funkcija vraća zbroj dvije varijable tipa *number* rezultat mora biti isto tipa *number*. Isto tako ako se definira neka varijabla ili konstanta kojoj se odmah pridružuje vrijednost, ne mora se nužno definirati tip već *TypeScript* sam zaključuje i definira tip. [5, 12].

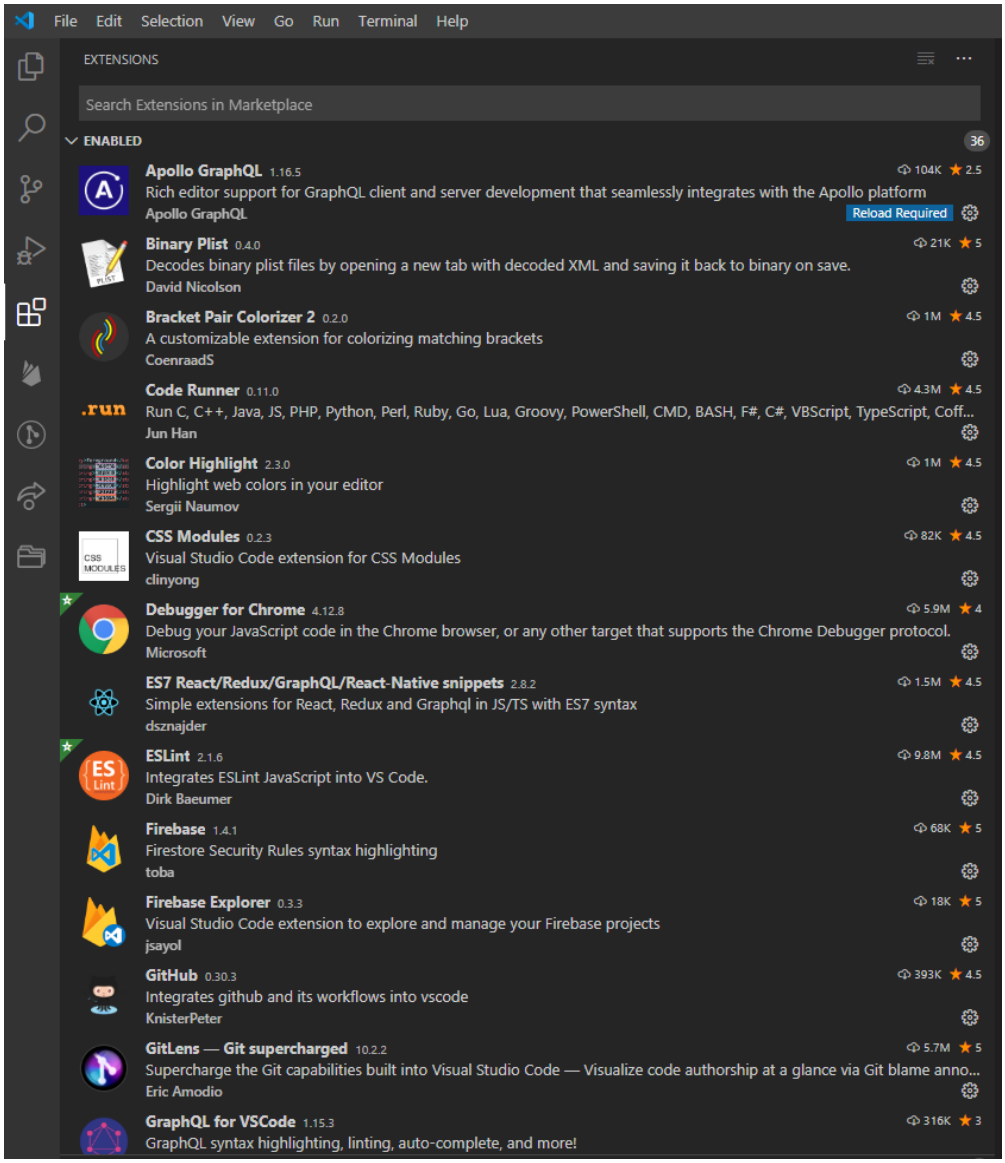
3.2. Razvojno okruženje Visual Studio Code

Za pisanje i uređivanje koda koristi se razvojno okruženje *Visual Studio Code* (kasnije u tekstu skraćeno *VSC*). To je besplatni, pojednostavljeni uređivač koda koji podržava sve funkcionalnosti koje su potrebne programeru prilikom razvijanja softvera. Najvažnije funkcionalnosti su otklanjanje pogrešaka, izvršavanje zadataka i kontrola nad verziranjem koda. *VSC* je podržan na *MacOS*, *Linux* i *Windows* operacijskim sustavima.



Slika 3.6. *Visual Studio Code stranica dobrodošlice*

VSC ima integriranu potporu za programske jezike *JavaScript*, *TypeScript* i *Node.js*. Ima i bogat ekosustav ekstenzija koje programer može dodati kao proširenja za druge jezike kao što su *C++, C#, Java*, *Python*, *PHP*, *Go* i drugi. No, ekstenzije ne služe samo kako bi omogućile podršku za druge jezike. *VSC* pruža programerima i niz drugih alata koji pridonose bržem razvoju softvera, pokretanju i ispravljanju koda. Postoje ekstenzije za različite alate koje podržavaju programerov tijek rada. Ekstenzije se jednostavno mogu pretraživati i instalirati putem *VSC Marketplace-a* gdje različiti autori objavljuju svoje ekstenzije. Većina tih ekstenzija su besplatne, no neke se plaćaju. *VSC* je fleksibilan i omogućava programeru potpunu slobodu da si složi razvojno okruženje kako to njemu odgovara, od prečaca na tipkovnici pa sve do velike liste ekstenzija. To je razlog zašto je *VSC* danas najpopularniji uređivač koda. Na slici 3.7. prikazana je lista nekoliko ekstenzija koje se koriste prilikom izrade aplikacije Zelenko [6].



Slika 3.7. *Ekstenzije unutar Visual Studio Code-a*

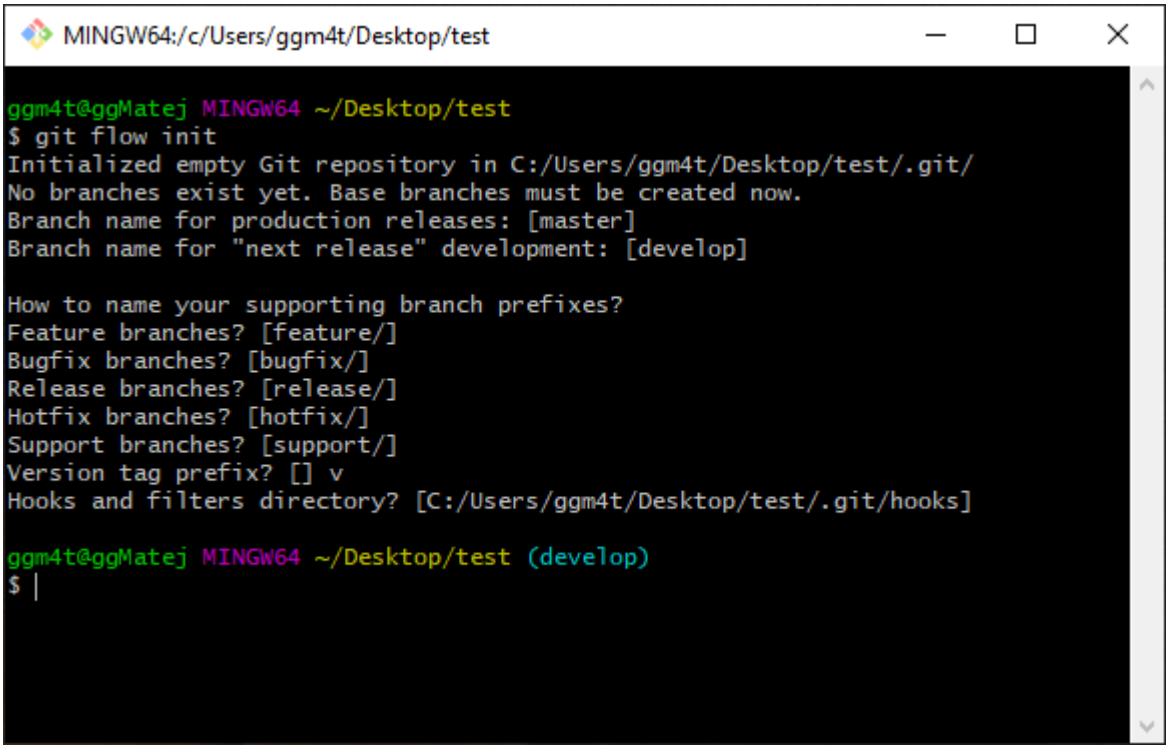
3.3. Sustav kontrole verzije koda

Sustavi kontrole verzija spadaju u kategoriju softverskih alata koji pomažu softverskom timu da upravljaju promjenama izvornog koda tijekom razvoja softvera. Softver za kontrolu verzija prati svaku izmjenu koda u nekoj vrsti baze podataka. Ako se prilikom izmjene koda dogodila pogreška, programeri mogu vratiti izvorni kod na neko prijašnje stanje koje je bilo stabilno.

Programeri koji rade u timovima neprestano mijenjaju postojeći izvorni kod. Taj kod obično ima arhitekturu i organiziran je u strukturi mapa. Jedan programer u timu može raditi na novoj funkcionalnosti, dok drugi radi na ispravljanju pogreške koja se dogodila prilikom izrade neke druge funkcionalnosti. Svaki programer u timu može raditi izmjene u bilo kojoj datoteci u strukturi mapa. Kod nekih manjih projekata na kojem radi manji tim programera se možda i mogu pratiti te promjene bez potrebnih alata, ali kod velikih projekata s velikim brojem različitih datoteka i timom od dvadesetak ljudi to može predstavljati problem. Sustavi kontrole verzija pomažu timovima u rješavanju tih problema. Prati se svaka pojedinačna promjena od strane svakog programera. Kod koji je izmijenjen u jednom dijelu softvera može biti nespojiv s kodom kojeg je neki drugi programer istodobno napisao. Sustav kontrole verzija pomaže otkriti taj problem na vrijeme i riješiti ga na uredan način tako da ne blokira rad ostatka tima. Prilikom razvoja softvera svaka promjena može uzrokovati nove pogreške. Sustav kontrole verzija u tom slučaju omogućava postojanje izvornog koda koji je stabilan, te koda na koji se dodaju izmijene i nove funkcionalnosti. Nakon što su nove funkcionalnosti testirane spaja se novi kod s izvornim kodom. Moguće je razviti softver bez korištenja kontrole verzija koda, no to ne bi preporučio niti jedan profesionalni tim.

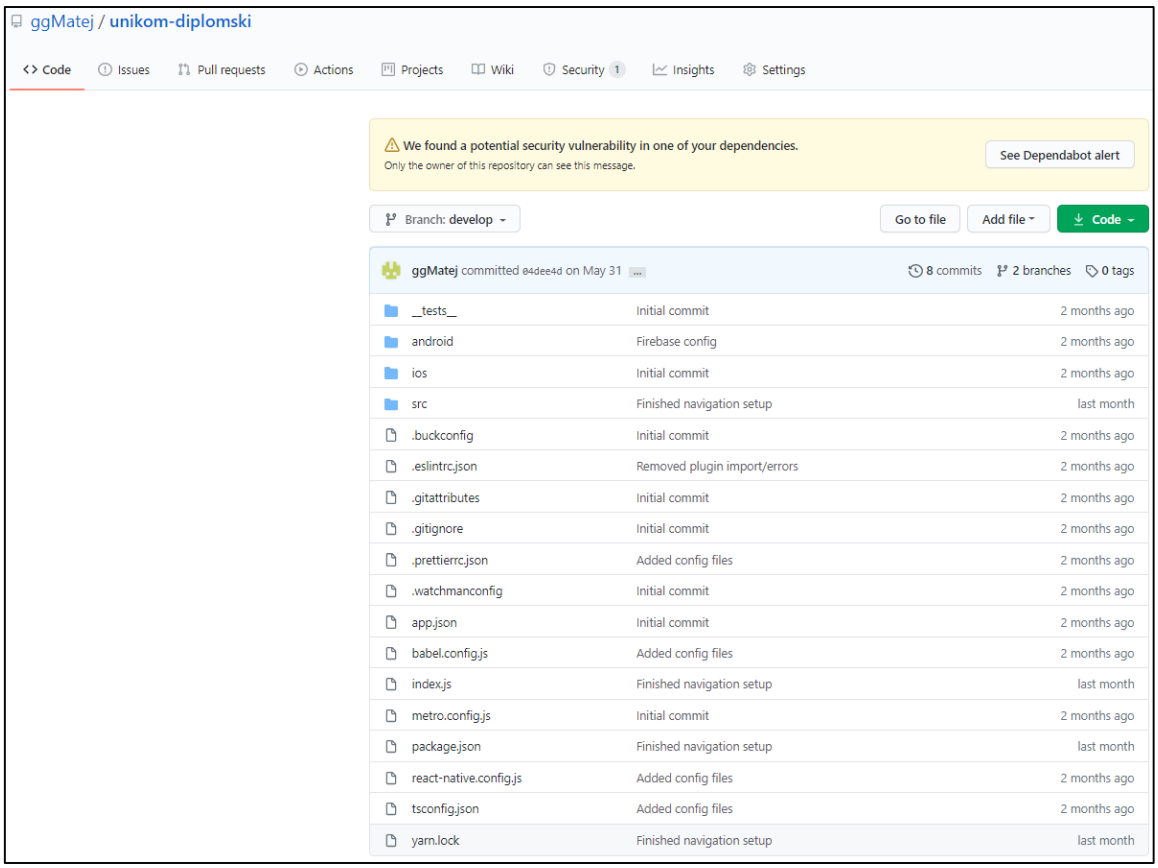
Tijekom razvoja aplikacije Zelenko kao alat za verziranje koda korišten je sustav za kontrolu verzija koda *Git. Git* je aktivno održavani projekt otvorenog koda kojeg je 2005. godine razvio Linus Torvalds, poznati tvorac jezgre *Linux* operativnog sustava [29]. *Git* ima funkcionalnosti, performanse, sigurnost i fleksibilnost koja je potrebna većini timova i pojedinačnih programera. U početku *Git* može biti težak za naučiti, ali u svijetu programera to je sigurno jedan od najbitnijih alata. Jedna od najvećih prednosti *Git-a* je mogućnost korištenja grana. Ukoliko programer želi razviti novu funkcionalnost može napraviti svoju granu na kojoj će neometano pisati svoj kod. Kada završi funkcionalnost može tu granu spojiti nazad na glavnu granu. *Git* omogućava različite pristupe korištenja koji ovise od tima do tima.

Prilikom izrade aplikacije Zelenko koristio se *git flow* tijek rada. Neke od prednosti ovog pristupa su čišća *Git* povijest, manje konflikata, nužni pregledi napisanog koda te veća stabilnost grana [13]. Način na koji se inicijalizira *git flow* tijek rada prikazan je na slici 3.8..



Slika 3.8. *Inicijalizacija gitflow tijeka rada*

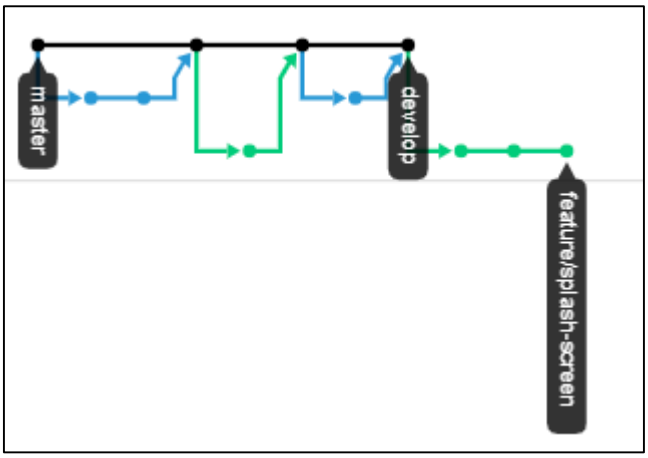
Kao pružatelj *Git* usluge koristi se *GitHub*. Na *GitHub-u* se nalazi repozitorij projekta gdje su spremljene sve datoteke odnosno izvorni kod projekta. Repozitorij ovog rada može se vidjeti na slici 3.9. Svaki projekt ima svoj repozitorij i odgovarajući URL putem kojeg se pristupa određenom repozitoriju. Naredbom *git flow init* inicijalizira se *git flow* tijek rada na repozitoriju. Nakon toga unosi se ime grane za produkcijsku verziju softvera i ime grane na kojoj se softver razvija. Zatim se mogu unijeti imena potpornih grana kao što su naprimjer grane za funkcionalnosti, otklanjanje pogrešaka, isporuku, podršku i hitni popravak. Također, može se navesti i prefiks za oznaku verzije. Prilikom izrade aplikacije Zelenko koristi se prefiks v, a ostala imena grana su ostavljena prema zadanim postavkama.



Slika 3.9. *GitHub repozitorij projekta*

Ukoliko se želi dodati nova funkcionalnost u aplikaciju mora se napraviti nova grana s imenom *feature/ime-funkcionalnosti* na kojoj se onda piše kod za novu funkcionalnost. Ta grana se mora napraviti s postojeće razvojne grane kako bi se zadržale sve postojeće funkcionalnosti aplikacije. Dok je prvi programer razvijao jednu funkcionalnost, drugi programer je u međuvremenu završio drugu funkcionalnost koju je spojio na postojeću razvojnu granu. Prvi programer je napravio svoju granu s razvojne grane i ima sve postojeće funkcionalnosti, no fali mu nova funkcionalnost koju je u međuvremenu razvio drugi programer i dodao na razvojnu granu. Iz tog razloga svaki put kada programer završi s razvijanjem funkcionalnosti mora napraviti takozvani *rebase* proces. Tim procesom programer povlači sve promjene koje su se u međuvremenu dogodile na razvojnoj grani i dodaje ih u svoju granu. Ako je programer dva radio promijene na istom dijelu koda kao i programer jedan, doći će do konflikta. U tom slučaju programer jedan mora odlučiti hoće li zadržati svoj kod, ili će ipak koristiti kod koji je napisao programer dva. Kada su svi konflikti riješeni i kad se završi *rebase* proces, funkcionalnost je spremna za spajanje na razvojnu granu. Da bi se nova funkcionalnost spojila na razvojnu granu programer mora napraviti *pull request*. Kada programer napravi *pull request*,

voditelji tima vide da je otvoren novi *pull request* na kojem se vide sve izmijene napravljene u kodu. Voditelji tima zatim mogu ostaviti komentare i pregledati novo napisani kod. Ako je sve uredu, voditelji će odobriti spajanje nove funkcionalnosti na razvojnu granu. Ovaj postupak se ne odnosi samo na grane za funkcionalnosti već i na ostale grane. Na slici 3.10. prikazan je graf grana na *GitHub-u*.



Slika 3.10. *Prikaz grafa grana na GitHub-u*

3.4. Firebase pozadinski servis

Firebase predstavlja primjer *Backend-as-a-Service (BaaS)* usluge koja se nalazi na *Google Cloud* platformi. To je model usluge u oblaku računala gdje programeri izdvajaju posao koji se bavi pozadinskim dijelom aplikacije kao što su baze podataka, autentifikacija korisnika i slično. *BaaS* pružatelji usluge pružaju unaprijed napisani softver za navedene aktivnosti koje se odvijaju na serveru tako da programerima preostaje samo pisanje i održavanje sučelja. Usluge koje pruža *Firebase* su *Cloud Firestore, Realtime Database, Authentication, Cloud Storage, Cloud Functions i Hosting*. U nastavku su opisane najčešće korištene usluge [9].

Cloud Firestore omogućava spremanje i sinkroniziranje podataka između korisnika i uređaja na globalnoj razini koristeći *NoSQL* bazu podataka koja se nalazi na oblaku. On omogućava sinkronizaciju u stvarnom vremenu te podršku u načinu rada bez mreže. *Cloud Firestore* ima predefinirane upite koje programer može koristiti za izvođenje *Create, Read, Update* i *Delete* (kasnije u tekstu skraćeno *CRUD*) operacija nad bazom podataka. Podaci se spremaju u bazu kao kolekcije i dokumenti. Jedna kolekcija može imati više dokumenata i pod kolekcija, koje zatim mogu imati svoje dokumente i pod kolekcije itd.

Realtime Database je slična usluga *kao Cloud Firestore*, no u tu bazu podaci se spremaju kao jedna velika *JavaScript Object Notation* datoteka [30]. To omogućava lakše spremanje jednostavnijih podataka, ali ako kod kompliciranijih struktura podataka preporučuje se korištenje *Cloud Firestore-a* zbog lakše organizacije tih podataka.

Authentication usluga omogućava upravljanje korisnicima. *Firebase* pruža više metoda za autentifikaciju korisnika kao što su e-mail i lozinka ili korištenje drugih vanjskih usluga poput *Google-a ili Facebooka*. Korisnik može razviti svoje sučelje za autentifikaciju, ili može koristiti postojeće sučelje koje nudi *Firebase*.

Na *Cloud Storage* se može pohraniti ili dijeliti sadržaj koji je stvorio korisnik poput slika, audio podataka, video snimaka i slično. *Firebase-ov SDK* za *Cloud Storage* dodaje *Google* sigurnost u datoteke koje se postavljaju i preuzimaju putem aplikacije bez obzira na kvalitetu interneta.

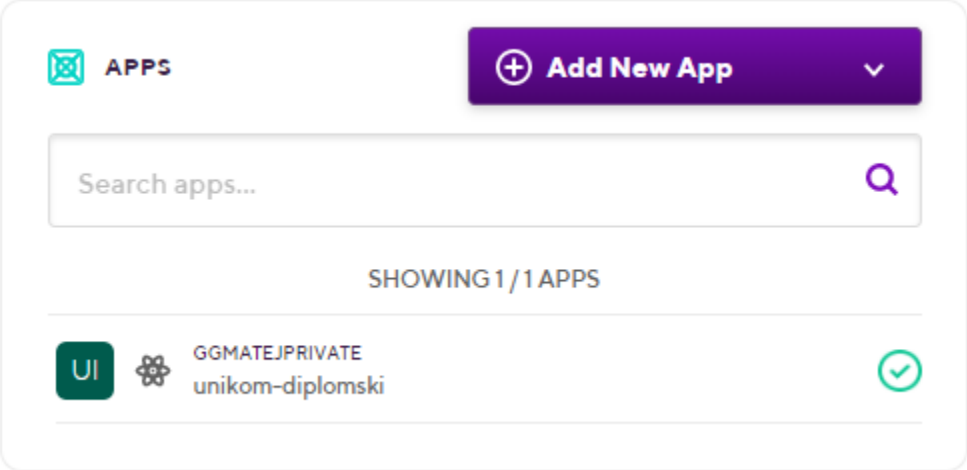
Cloud Functions pruža mogućnost proširivanja aplikacije gdje korisnik može sam pisati kod za pozadinski servis. Korisnik piše funkcije koje mogu biti automatski pozvane nakon što se dogodi događaj poput *CRUD* operacija nad kolekcijom.

3.5. Kontinuirana izgradnja i isporuka aplikacije pomoću Bitrise-a

Razvoj, testiranje i izgradnja aplikacije dugotrajan je proces koji traži puno pažnje i vremena od tima. Osobito ako se svaki taj korak radi ručno. U svakom koraku može doći do greške i mora se biti jako pažljiv. Kako bi se izbjegle greške prilikom razvoja, testiranja i izgradnje nastoji se što je više moguće procesa automatizirati. Automatiziranjem pojedinih koraka smanjuje se vjerojatnost pojavljivanja greške, povećava se kvaliteta softvera i štedi se puno vremena unutar tima. Postoji puno automatizacijskih alata kao što su *Jenkins, Travis, Circle, Buddybuild i Bitrise*. Za izgradnju i isporuku aplikacije Zelenko koristi se *Bitrise* jer je napravljen isključivo za mobilne aplikacije.

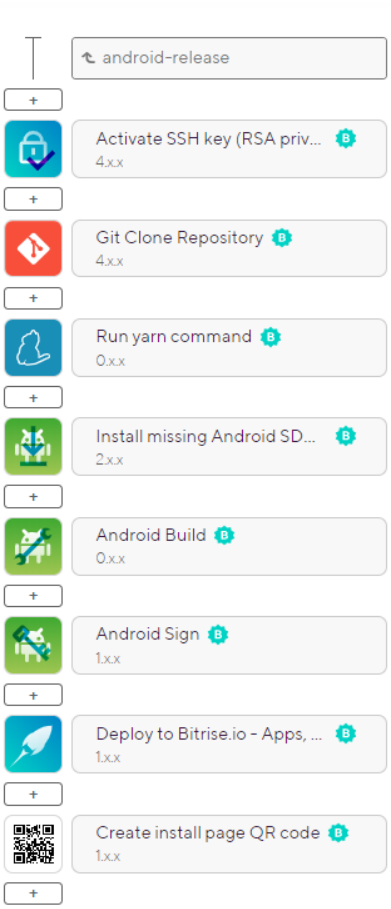
Za korištenje *Bitrise-a* može se napraviti korisnički račun ali najbolja praksa je prijaviti se s postojećim računom koji se već koristi na pružatelju *Git* usluge. U slučaju aplikacije Zelenko to je *GitHub* račun na kojem se nalazi i repozitorij aplikacije. Za dodavanje nove aplikacije na *Bitrise* moraju se dodati određeni podaci poput vlasnika, repozitorija, pristupa repozitoriju, imena grane na repozitoriju s koje će se aplikacija izgrađivati, konfiguracija izgrađivanja i drugo. Na slici 3.11. prikazani su koraci dodavanja aplikacije na *Bitrise,* a na slici 3.12. vidljiva je lista dodanih aplikacija.

Slika 3.11. *Dodavanje aplikacije na Bitrise*



Slika 3.12. *Popis aplikacija na Bitrise-u*

Kada programer završi s razvojem aplikacije, tu aplikaciju mora izgraditi kako bi ju mogao proslijediti klijentu ili testeru, ili pak kako bi je mogao dodati na *Google* trgovinu. Svaki puta kada programer doda novu funkcionalnost mora opet izgraditi aplikaciju i ponoviti postupak. Tijek izgradnje aplikacije na *Bitrise-u* to automatizira i uvelike olakšava programeru posao. Nakon što je aplikacija dodana, mora se napraviti konfiguracija tijeka rada, odnosno tijeka izgradnje aplikacije. Ona se sastoji od nekoliko uzastopnih koraka kroz koje izgradnja aplikacije mora proći. Obavezni koraci koje tijek izgradnje mora sadržavati su kloniranje *Git* repozitorija, instaliranje potrebnih aplikacijskih paketa, izgradnja aplikacije i potpisivanje aplikacije. Neki dodatni koraci koji se mogu dodati su automatsko slanje e-maila kada se izgradnja završi, slanje *QR* koda za instalaciju i slično. Koraci tijeka izgradnje koji se koriste pri izgradnji aplikacije Zelenko prikazani su na slici 3.13.



Slika 3.13. *Koraci tijeka izgradnje aplikacije na Bitrise-u*

Izgradnja aplikacije na *Bitrise-u* može se pokrenuti ručno, no moguće je staviti okidač koji će automatski pokrenuti izgradnju. Za aplikaciju Zelenko izgradnja će se pokrenuti svaki puta kada se nova funkcionalnost spoji na glavnu granu za razvoj na repozitoriju. Kada izgradnja završi, *QR* kod ili link za instalaciju bit će poslan na odabrane e-mail adrese. Aplikacija se putem tog linka može instalirati direktno na mobitel i testirati. Korištenjem *Bitrise-a* automatizira se dio koji se bavi izgradnjom aplikacije i olakšava se prosljeđivanje aplikacije klijentima.

3.6. React Native okvir za izradu mobilnih aplikacija

React Native je okvir koji služi za izradu višeplatformskih mobilnih aplikacija. Kombinira najbolje dijelove nativnog razvoja s *React-om*. Omogućava web developerima koji nemaju puno iskustva u mobilnom svijetu razvijanje mobilnih aplikacija koje izgledaju kao nativne [4].

3.6.1. Prednosti i nedostaci React Native razvojnog okvira

Prednost korištenja *React Native* okvira za razvoj u odnosu na nativni razvoj je ta što se kod može dijeliti između platformi. To znači da se može istovremeno razvijati aplikacija za *Android* i *iOS* platforme. Razvoj višeplatformskih aplikacija je jeftiniji jer umjesto dva razvojna tima potreban je samo jedan. Isto tako lakše je i efikasnije održavati jedan izvorni kod nego posebno za *Android* i *iOS* platforme. Naravno, *React Native* ima i svojih nedostataka. Višeplatformske aplikacije u odnosu na nativne znaju imati problema s performansama. Nisu optimizirane kao nativne aplikacije i tijekom korištenja zna doći do gubljenja okvira kod zahtjevnih animacija. Isto tako neke funkcionalonosti mogu se postići samo nativnim razvojem.

3.6.2. Osnove React Native razvojnog okvira

Za razvoj *Android* zaslona koriste se programski jezici *Kotlin* i *Java*. Za razvoj *iOS* zaslona koriste se *Swift* ili *Objective-C*. U *React Native-u* zasloni se stvaraju s *JavaScriptom* korištenjem *React* komponenti. Za vrijeme izvođenja *React Native* stvara odgovarajuće *Android* i *iOS* zaslone za te komponente. *React Native* sadrži komplet nativnih komponenti koje se mogu koristiti prilikom razvoja. U *React Native-u* te komponente su poznate kao *Core Components*. Naravno, *React Native* omogućava kreiranje vlastitih nativnih komponenata za *Android* i *iOS* ovisno o potrebama aplikacije koja se razvija. Neke od glavnih komponenata koje se koriste prilikom razvoja aplikacije Zelenko mogu se vidjeti u tablici 3.1. Također, navedeni su *Android*, *iOS* i web ekvivalenti [4].

React Native komponenta	**Android view**	**iOS view**	**Web**	**Opis**
<View>	<ViewGroup>	<UIView>		Okvir koji omogućava prikaz izgleda zaslona
<Text>	<TextView>	<UITextView	<p>	Prikaz stringova
s<Image>	<ImageView>	<UIImageView>	<img>	Prikaz slika
<ScrollView>	<ScrollView>	<UIScrollView>		Okvir koji se može pomicati
<TextInput>	<EditText>	<UITextField>	<input>	Unos stringova

Tablica 3.1. *React Native Core Components*

3.6.3. Komponente

Kombiniranjem komponenti koje *React Native* pruža stvaraju se vlastite komponente. Osnovne stavke jedne komponente su *JavaScript extensible markup language* (kasnije u tekstu skraćeno *JSX)*, svojstva, stanja i stilovi. Na slici 3.14. prikazan je primjer jedne komponente. *React* i *React Native* koriste *JSX* koji omogućava korištenje komponenti unutar *JavaScript-a* [4]. Unutar *return* izraza ubačene su *TouchableOpacity* i *Image* komponente. *HeaderInfoIcon* komponenta koristi *navigation* svojstvo. To znači da kada se koristi *HeaderInfoIcon* komponenta mora joj se proslijediti *navigation* svojstvo koje služi za navigaciju između zaslona. Također, *Image* komponenti, koja je dio *HeaderInfoIcon* komponente, su proslijeđena dva svojstva a to su *style* i *source.* Svojstva mogu biti obvezna ili opcionalna. *HeaderInfoIcon* komponenta je takozvana *stateless* komponenta odnosno komponenta bez stanja. Primjer komponente sa stanjem je brojač. Unutar te komponente prati se stanje brojača. Stilovi koje sadrži su desna margina i boja.

```
type Props = NavigationProps<'main'>;

export const HeaderInfoIcon: React.FC<Props> = ({ navigation }) => {
  function onInfoPressed() {
    navigation.replace('onboarding');
  }
  return (
    <TouchableOpacity onPress={onInfoPressed}>
      <Image
        style={styles.infoIcon}
        source={require('assets/images/info-icon.png')}
      />
    </TouchableOpacity>
  );
};

const styles = StyleSheet.create({
  infoIcon: {
    marginRight: 15,
    tintColor: Color.Secondary,
  },
});
```

Slika 3.14. *Primjer React Native komponente*

4. OPIS RAZVOJA APLIKACIJE

U ovom poglavlju opisan je tijek razvoja aplikacije. Detaljno su prikazani načini implementacije traženih funkcionalnosti odnosno sam proces razvoja aplikacije. Također, opisani su i svi alati koji se koriste prilikom razvoja aplikacije.

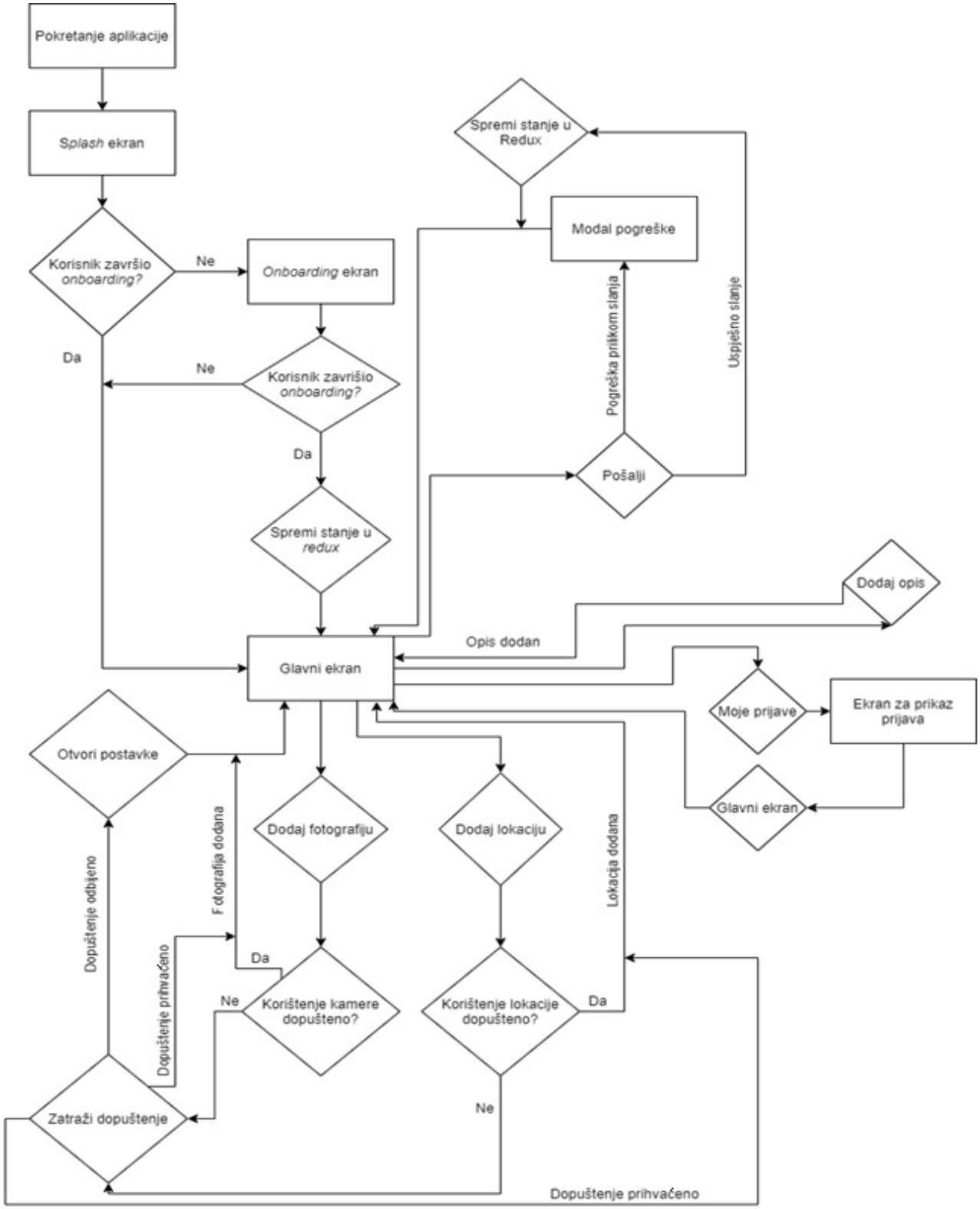
4.1. Dizajn aplikacije i korisničko iskustvo

Prije početka razvoja aplikacije bitno je napraviti dijagram toka. Dijagram toka koristi se za pokrivanje svih slučajeva s kojima se korisnik aplikacije može susresti. Detaljan i kvalitetan dijagram toka znači brži i kvalitetniji razvoj aplikacije. Također, dobra priprema prije samog razvoja aplikacije omogućava bolje strukturiranje koda i komponenata te će arhitektura aplikacije biti vrlo kvalitetna. Ako je arhitektura aplikacije kvalitetna, kod će biti lakše održavati u slučaju da dođe do promjena dizajna ili novih funkcionalnih zahtjeva. Na slici 4.1. prikazan je dijagram toka aplikacije. Iz dijagrama toka aplikacije vidi se kakve zaslone aplikacija koristi. Zasloni koji se koriste u izradi aplikacije su: *splash*, *onboarding,* glavni, prijave i skočni prozori.

Splash zaslon je zaslon koji se prikazuje prilikom svakog pokretanja aplikacije. On služi za obavljanje inicijalne logike prije nego se korisniku dopusti korištenje aplikacije. Na primjer, dok se učitavaju podaci potrebni za uspješno pokretanje i korištenje aplikacije korisniku se može prikazati logo aplikacije. Inicijalno učitavanje ne treba trajati dugo kako bi se izbjeglo loše korisničko iskustvo.

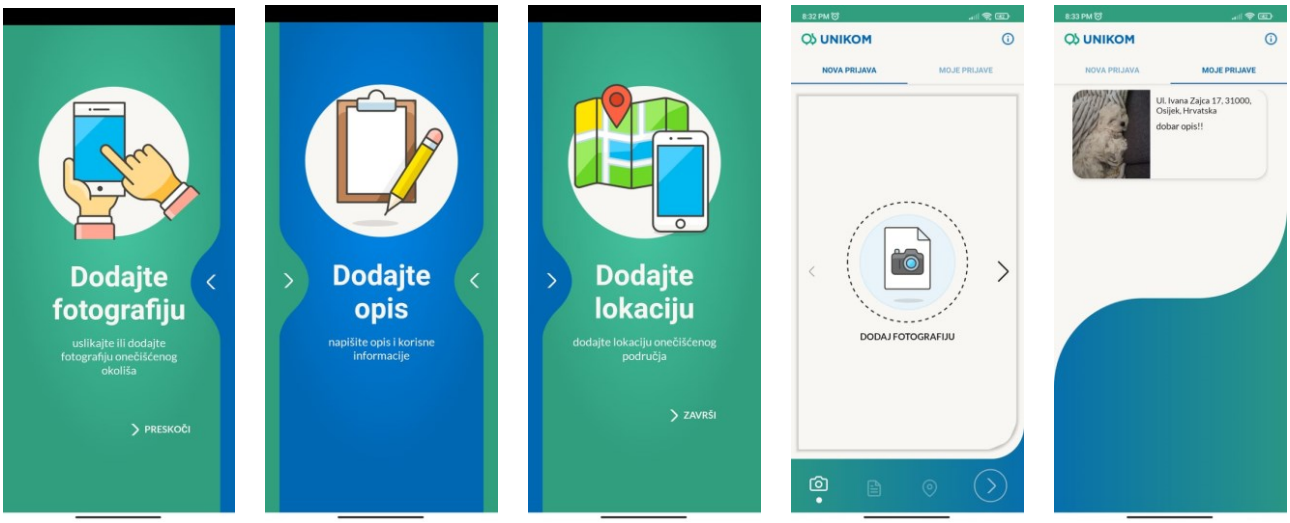
Onboarding zaslon služi da se korisniku objasni korištenje aplikacije. On se najčešće sastoji od niza zaslona koji korisniku nastoje što jednostavnije i slikovitije prikazati postupak korištenja aplikacije. *Onboarding* zasloni nisu potrebni ali pridonose korisničkom iskustvu.

Splash i *onboarding* zasloni su uobičajeni zasloni većine aplikacija. Preostali zasloni ovise o funkcionalnim zahtjevima te samom rješenju arhitekture aplikacije. U slučaju aplikacije Zelenko to su glavni zaslon na kojem su implementirane glavne funkcionalnosti dodavanja slike, opisa i lokacije te zaslon na kojem se može vidjeti povijest prijava korisnika.



Slika 4.1. *Dijagram toka aplikacije*

Nakon što se definira dijagram toka aplikacije i zasloni koji su potrebni, može se početi s dizajnom izgleda aplikacije. Dizajn aplikacije najbolje je raditi s alatima poput *Figme* ili *InVision-a* jer pružaju različite funkcionalnosti za detaljan dizajn zaslona. Za dizajn aplikacije Zelenko nije korišten niti jedan alat već su se zasloni dizajnirali prilikom samog razvoja. Razlog tome je nedostatak resursa i prakse u tom području. Definirane su osnovne varijable kao što su globalni stilovi. Globalni stilovi predstavljaju font i boje koje se koriste na razini cijele aplikacije. Također, ikone i ilustracije koje su korištene prilikom izrade aplikacije Zelenko preuzete su sa *Streamline* stranice [14]. Na slici 4.2. može se vidjeti finalni izgled zaslona aplikacije



Slika 4.2. *Finalni izgled zaslona aplikacije Zelenko*

4.2. Postavljanje alata i inicijalizacija projekta

Kada se definiraju funkcionalnosti, dizajn, slučaj uporabe i dijagram toka razvija se aplikacija. Za razvoj *React Native* aplikacije potrebni su *Node*, *Android Studio* i *Java SE Development Kit (JDK). Android* studio omogućava potrebne alate za izradu *React Native* aplikacije. Kao razvojno okruženje koristi se *Visual Studio Code*, a za verziranje koda koristi se *Git*. *Visual Studio Code* i *Git* objašnjeni su u potpoglavljima 3.2 i 3.3.

Kada su svi potrebni alati postavljeni može se inicijalizirati projekt. Projekt se inicijalizira naredbom *npx react-native init ime-aplikacije --template react-native-template-typescript.* Ta naredba će izgenerirati datoteke koje su potrebne za pokretanje *React Native* aplikacije. Zbog *react-native-template-typescript* opcije generirane su dodatne datoteke koje omogućavaju korištenje *TypeScript-a* unutar projekta. Zatim se može inicijalizirati *git flow* čiji postupak je objašnjen u potpoglavlju 3.3. Nakon što su generirane sve potrebne datoteke projekt se dodaje na *GitHub* repozitorij i projekt se pokreće radi provjere ispravnosti generiranih datoteka. Projekt se pokreće s naredbom *react-native run-android* za *Android* mobilne uređaje.

Kada je ustanovljeno da je aplikacija uspješno pokrenuta odnosno kada je projekt inicijaliziran, mogu se početi dodavati paketi potrebni za razvoj rješenja te postaviti određene konfiguracije. Jedna od tih konfiguracija je *TypeScirpt* konfiguracija. Ta se konfiguracija definira u *tsconfig.json* datoteci. Ona se sastoji od seta pravila koja se moraju poštivati prilikom razvoja našeg koda. Pravila se definiraju od strane developera i ovise od poduzeća do poduzeća. Ukoliko se prekrši jedno od pravila programski prevodilac će bacati grešku. Pravila se definiraju iz razloga kako bi se postigla konzistentnost u pisanju koda na razini cijele aplikacije te kako bi se izbjegle različite greške u kodu. Pravila korištena prilikom razvoja aplikacije Zelenko mogu se vidjeti na slici 4.3.

Slika 4.3. *Prikaz TypeScript konfiguracijske datoteke i korištenih pravila*

Kao upravitelj paketima koristi se *yarn* [15]. Uz *yarn* postoji još i *npm* upravitelj paketa [16] ali *yarn* instalira pakete istovremeno dok *npm* instalira jedan po jedan. Dakle, korištenje *yarn-a* ubrzava proces razvoja. Prvo su dodani osnovni paketi prikazani na slici 4.4.

yarn add [ime paketa] –D

- *eslint*
- *eslint-config-prettier*
- *eslint-loader*
- *eslint-plugin-prettier*
- *eslint-plugin-react*
- *eslint-plugin-react-hooks*
- *eslint-plugin-simple-import-sort*
- *@typescript-eslint/eslint-plugin*
- *@typescript-eslint/parses*
- *prettier*

Slika 4.4. *Dodavanje osnovnih paketa naredbom yarn*

Prema slici 4.4. dodani su *prettier* [18] i *eslint* [19] paketi. *Prettier* se koristi za formatiranje koda na razini cijele aplikacije. Pravila po kojima se kod formatira se također mogu definirati u konfiguracijskoj datoteci. Korištenjem *prettier* paketa kod će biti formatiran na isti način u svakoj datoteci a time i puno čitljiviji. *Eslint* paket ima sličan zadatak kao *tsconfig* pravila. To je alat za identificiranje i prijavljivanje uzoraka pronađenih u *ECMAScript/JavaScript* kodu s ciljem da kod bude konzistentniji i da se izbjegnu greške. Konfiguracijske datoteke *prettiera* i *eslinta* mogu se vidjeti na slikama 4.5. i 4.6.

```
{
  "tabWidth": 4,
  "singleQuote": true,
  "prettier.jsxBracketSameLine": true,
  "trailingComma": "all",
  "overrides": [
    {
      "files": ["*.tsx", "*.ts", "*.json", "*.yml"],
      "options": {
        "tabWidth": 2
      }
    }
  ]
}
```

Slika 4.5. *Konfigracijska datoteka prettier paketa*

Slika 4.6. *Prikaz pravila korištenih u eslint konfiguracijskoj datoteci*

Prettier i *eslint* paketi su uobičajeni paketi koji se dodaju na svaki projekt. Ostali paketi dodaju se po potrebi ovisno o funkcionalnim zahtjevima aplikacije. Na primjer, kod razvoja aplikacije Zelenko dodaje se paket za pozadinski servis *Firebase* što je objašnjeno u potpoglavlju 4.5.

4.3. Upravljanje stanjem unutar aplikacije

Za upravljanje stanjem u aplikaciji koristi se *Redux* [19]. Ukratko, *Redux* koristi spremnik s podacima kojima se može pristupiti iz bilo koje komponente aplikacije. On olakšava komunikaciju i razmjenu podataka među komponentama. *Redux* je koristan u slučaju da se stanje mora dijeliti između komponenti koje su daleko u stablu komponenti. U slučaju da se ne koristi *Redux* stanje se mora prosljeđivati gore po komponentama do najbliže roditeljske komponente, a zatim se prosljeđivati dolje do komponente koja treba koristiti to stanje. To je ne praktično zato jer postoji definirano stanje u komponentama u kojima to stanje nije potrebno. Također, to otežava održavanje i praćenje stanja i stanje je manje predvidivo. Kod manjih aplikacija *Redux* možda i nije potreban, ali kod većih aplikacija s kompleksnijim stanjima alat za upravljanjem stanja je obavezan ako se želi napraviti kvalitetan proizvod lagan za održavanje. Redux se sastoji od tri dijela. Akcija, *store-a* i *reducer-a.*

Akcije su događaji koji se koriste za slanje podataka iz aplikacije u *Redux store*. Akcije se šalju *dispatch* metodom. Akcije predstavljaju običan *JavaScript* objekt koji se sastoji od tipa akcije i *payload-a* kojemu se prosljeđuju podaci koji se šalju. *Payload* je opcionalan.

Reduceri su čiste funkcije kojima se prosljeđuje trenutno stanje aplikacije i akcija, a zatim vraćaju novo stanje aplikacije ovisno o poslanoj akciji. Unutar *reducera* definira se sučelje stanja aplikacije te inicijalne vrijednosti tih stanja. Aplikacija se može sastojati od više *reducera*. *Reduceri* se najčešće definiraju po modulima aplikacije odnosno po logičkim cjelinama.

Store je spremnik koji sadrži cjelokupno stanje aplikacije. Preporuka je da se koristi isključivo jedan *store* u aplikaciji.

4.3.1. Implementacija Redux paketa u aplikaciji Zelenko

Na slici 4.7. prikazani su potrebni *redux* paketi koji su dodani u projekt. Aplikacija Zelenko sastoji se od samo jednog *reducera*. To je *UserReducer*. Unutar tog *reducera* prati se stanje korisnika. To stanje sastoji se od sljedećih podataka: prijave koje je poslao korisnik te podatak koji govori je li korisnik prošao kroz *onboarding* zaslon ili ne. Prikaz i inicijalne vrijednosti stanja *UserReducer-a* vidljivi su na slici 4.8.

```
"react-redux": "^7.2.4",
"redux": "^4.1.1",
"redux-devtools-extension": "^2.13.9",
"redux-flipper": "^2.0.0",
"redux-persist": "^6.0.0",
"redux-thunk": "^2.3.0",
"reselect": "^4.0.0",
```

Slika 4.7. *Potrebni redux paketi*

```
export interface UserState {
  finishedOnboarding: boolean;
  reports: Report[];
}

const INITIAL_STATE: UserState = {
  finishedOnboarding: false,
  reports: [],
};
```

Slika 4.8. *Prikaz stanja UserReducer-a*

```
export const UserActions = {
  finishOnboarding: () => createAction(UserActionTypes.OnboardingSuccess),
  addReport: (userReport: Report) =>
    createAction(UserActionTypes.ReportSuccess, { userReport }),
};
```

Slika 4.9. *Prikaz korištenih akcija*

Prema slici 4.9. akcije se kreiraju *createAction* funkcijom. Akcije koje se koriste su *OnboardingSuccess* i *ReportSuccess*. Tip akcije *OnboardingSuccess* ne sadrži *payload* dok se kod akcije tipa *ReportSuccess* prosljeđuje *userReport* kao *payload*. Kako bi se pozvala promjena stanja akcija se mora proslijedit u *dispatch* funkciju. Na slici 4.10. prikazana je funkcija koja će se pozvati nakon što se klikne na završi tipku. Ta funkcija *dispatch-a* akciju *OnboardingSuccess* koja zatim u *reduceru* poziva slučaj uzrokovan tom akcijom te mijenja stanje svojstva *finishedOnboarding* na *true*. Slika 4.11. prikazuje *UserReducer*.

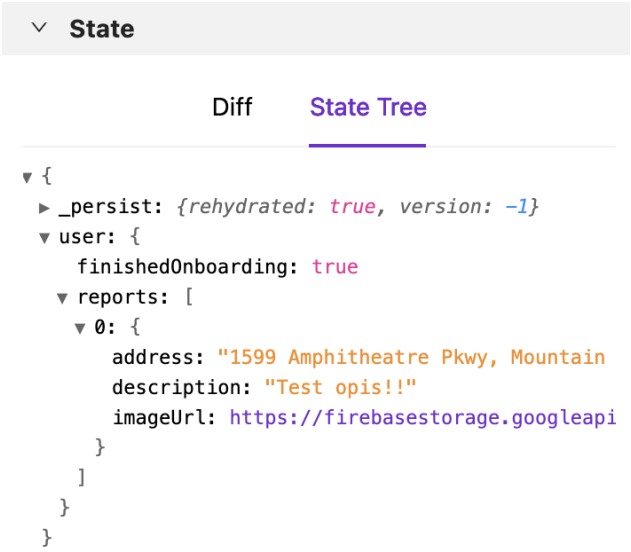
```
function handleOnDone() {
  dispatch(UserThunks.finishOnboarding());
  navigation.replace('main');
}
```

Slika 4.10. *Dispathcanje OnboardingSuccess akcije*

```
export const UserReducer = (
  state: UserState = INITIAL_STATE,
  action: UserAction,
) => {
  switch (action.type) {
    case UserActionTypes.OnboardingSuccess:
      return {
        ...state,
        finishedOnboarding: true,
      };
    case UserActionTypes.ReportSuccess:
      return {
        ...state,
        reports: [...state.reports, action.payload.userReport],
      };
    default:
      return state;
  }
```

Slika 4.11. *Prikaz user reducera*

Poput *OnboardingSuccess* slučaja slika 4.11. prikazuje i kakvo stanje *reducer* vraća u slučaju pozivanja *ReportSuccess* akcije. Nakon uspješnog slanja prijave *dispatch-a* se *ReportSuccess* akcija kojoj se kao *payload* mora proslijediti *Report* objekt. *Reducer* će zatim vratiti novo stanje u koje je dodana nova prijava. Na slici 4.12. vidi se prikaz stanja aplikacije nakon što je pregledan *onboarding* zaslon i dodana jedna prijava.



Slika 4.12. *Prikaz stanja aplikacije Zelenko*

Store spremnik se kreira s pomoćnom *configureStore* funkcijom (slika 4.13.). Ta funkciju se poziva u *index.js* datoteci. Nakon toga kreirani store se prosljeđuje *Provider* komponenti kojom se onda omeđuje cijela aplikacija. Način na koji je to implementiramo vidi se na slici 4.14. Također, može se primijetiti i korištenje *persistora*. *Perisitor* omogućava da se stanje aplikacije sprema i nakon što je aplikacija ugašena. Dakle, ako je korisnik pregledao *onboarding* zaslon svaki sljedeći put kada upali aplikaciju očitati će se prethodno spremljeno stanje i u slučaju da je korisnik pregledao *onboarding* zaslon automatski će ga se navigirati na glavni zaslon. U aplikaciji Zeleneko se sprema stanje cijelog *UserReducer-a.*

```
export const configureStore = () => {
  /**
   * Create the composing function for our middlewares
   * Include dev tools support
   */
  const composeEnhancers = composeWithDevTools({});

  /**
   * Merge all reducers into a single object
   */

  const middlewaresToApply = [thunk];

  if (__DEV__) {
    const createFlipperDubugger = require('redux-flipper').default;
    middlewaresToApply.push(createFlipperDubugger());
  }

  const persistConfig = {
    key: 'root',
    storage: AsyncStorage,
  };

  const reducer = persistCombineReducers(persistConfig, {
    user: UserReducer,
  });

  /**
   * We'll create our store with the combined reducers and all enchancers
   */
  return createStore(
    reducer,
    {},
    composeEnhancers(applyMiddleware(...middlewaresToApply)),
  );
};
```

Slika 4.13. *Prikaz configureStore pomoćne funckije za kreiranje store-a*

```
const App = () => {
    const store = configureStore();
    const persistor = persistStore(store);

    return (
        <Provider store={store}>
            <SafeAreaProvider>
                <PersistGate loading={null} persistor={persistor}>
                    <RootNavigator />
                </PersistGate>
            </SafeAreaProvider>
        </Provider>
    );
};
```

Slika 4.14. *Kreiranje store-a i korištenje Provider komponente*

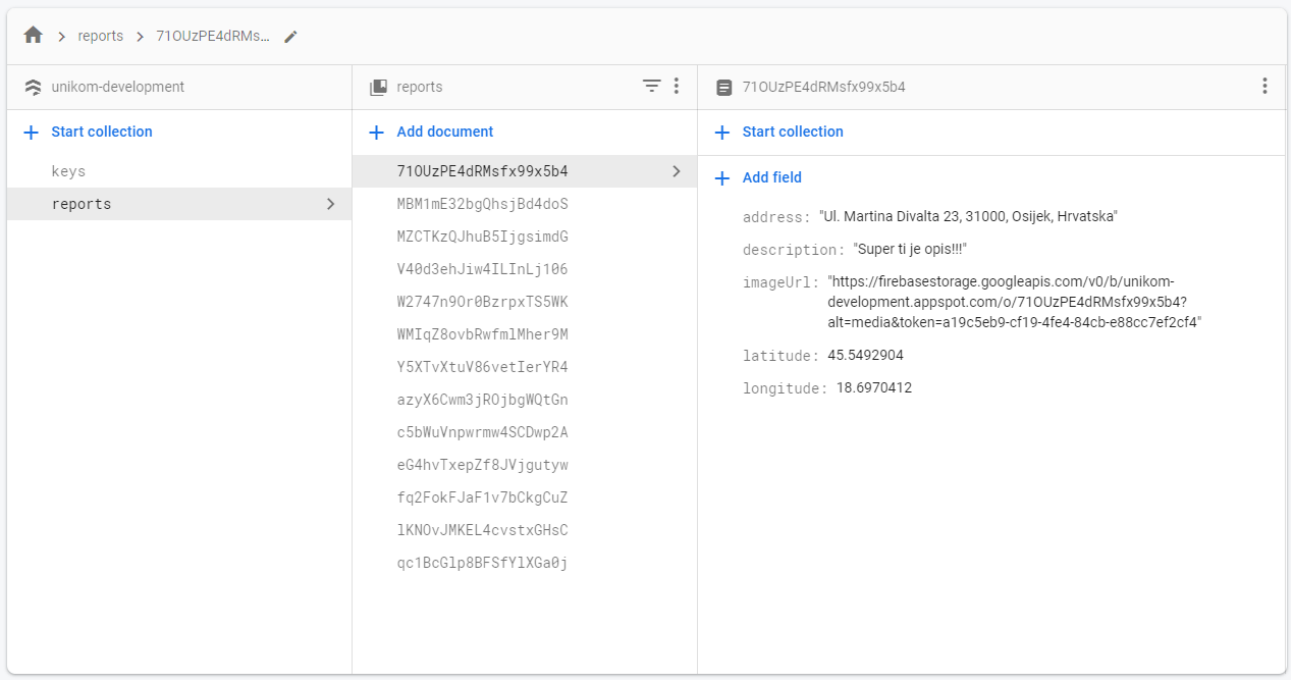
4.4. Dodavanje pozadinskog servisa

Kao pozadinski servis u aplikaciji Zelenko koristi se *Firebase*. *Firebase* funkcionalnosti objašnjene su u potpoglavlju 3.4. Kao i do sada, za korištenje *Firebase* funkcionalnosti u aplikaciji moraju se dodati određeni paketi koji su prikazani na slici 4.15.

```
"@react-native-firebase/app": "^12.6.1",
"@react-native-firebase/firestore": "^12.6.1",
"@react-native-firebase/storage": "^12.6.1",
```

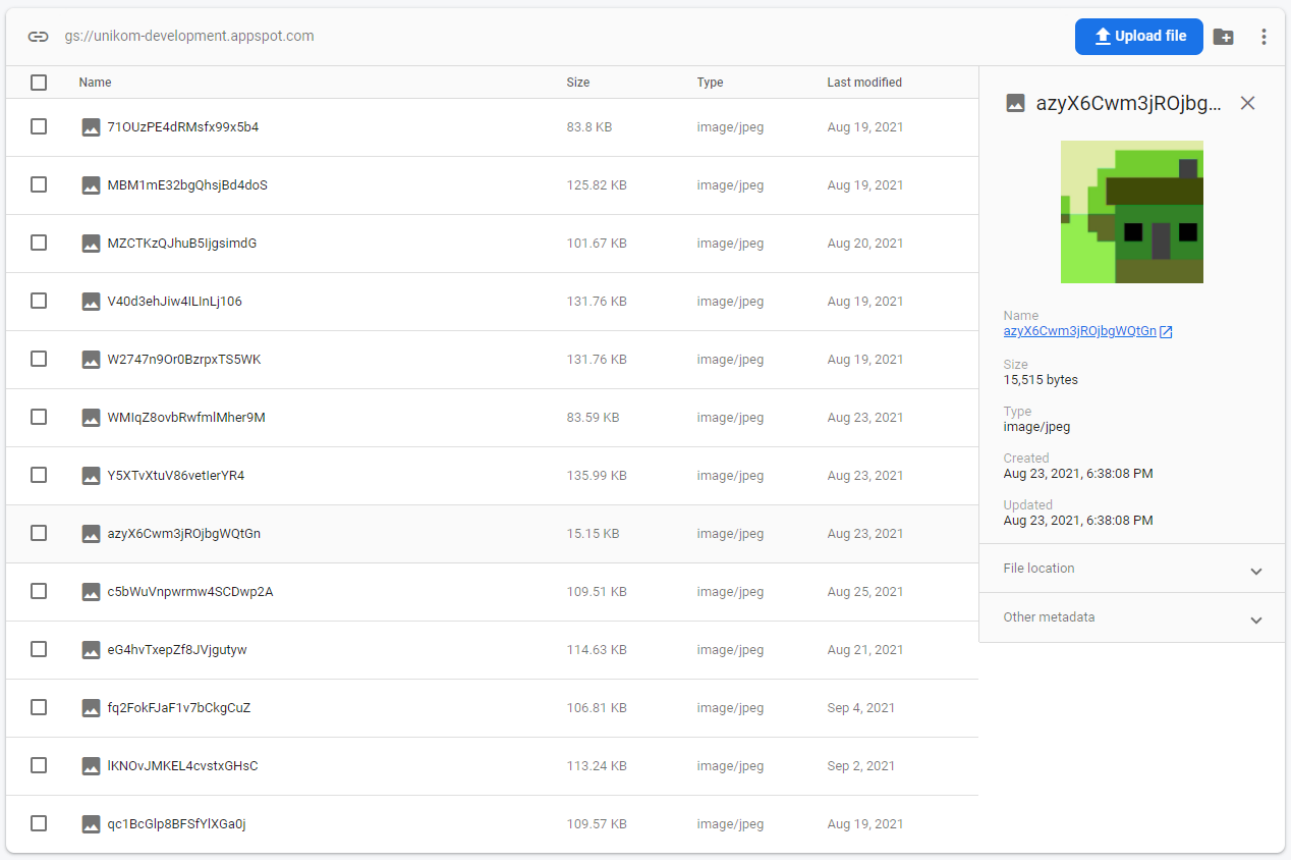
Slika 4.15. *Prikaz potrebnih firebase paketa*

Od navedenih *Firebase* funkcionalnosti u potpoglavlju 3.4. koriste se *Cloud Firestore* i *Cloud Storage*. U *Firestore* bazu podataka spremaju se prijave dok se u *Cloud Storage* spremaju dodane fotografije. Nakon što se doda projekt na *Firebase* mora se preuzeti konfiguracijska datoteka. Ona sadrži informacije o projektu bez kojih aplikacija ne može komunicirati s pozadinskim servisom. Konfiguracijska datoteka sadrži informacije poput imena i broja projekta, različite identifikacijski brojeve, API ključeve i slično. Kada se doda konfiguracijska datoteka u projekt mogu se početi koristiti *Firebase* funkcionalnosti. Na slici 4.16. prikazan je način na koji je složena struktura *Firestore* baze podataka aplikacije Zelenko.



Slika 4.16. *Prikaz Firestore baze podataka aplikacije Zelenko*

Firestore baza podataka sastoji se od kolekcija i dokumenata. Dokumenti mogu imati različita polja različitih tipova podataka. Svaki dokument može imati i svoju dodatnu kolekciju te se tako može složiti kompleksna hijerarhija podataka. Baza podataka aplikacije Zelenko sastoji se od dvije kolekcije: *keys* i *reports*. U kolekciju keys spremljeni su različiti API ključevi iz sigurnosnih razloga. U kolekciju *reports* spremaju se prijave. Spremljene prijave su zapravo dokumenti koji se sastoje od sljedećih polja: adresa, opis, url dodane fotografije, zemljopisna širina i dužina. Url fotografije je zapravo referenca na fotografiju koja je spremljena unutar *Cloud Storage-a*. Prikaz spremljenih fotografija unutar *Cloud Storage-a* može se vidjeti na slici 4.17.



Slika 4.17. *Prikaz spremljenih fotografija u Cloud Storage-u*

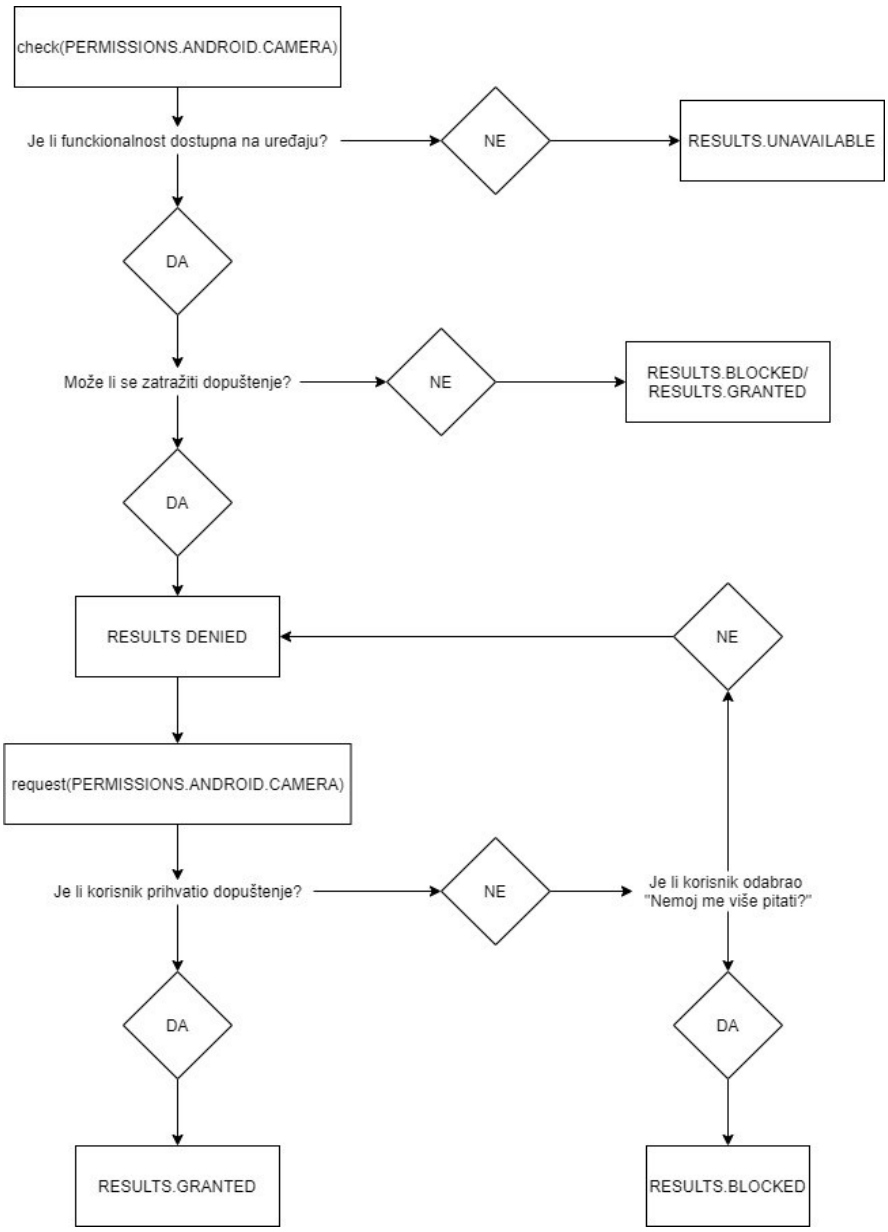
Kada korisnik pošalje tipku pošalji poziva se *onSubmit* funkcija. Funkcija *onSubmit* pristupa *reports* kolekciji i dodaje dokument koji sadrži adresu, zemljopisu dužinu i širinu te opis. Nakon što je dokument uspješno dodan automatski se poziva funkcija za dodavanje fotografije u *Cloud Storage*. Nakon izvršenja te funkcije poziva se *update* funkcija na navedenom dokumentu i dodaje se url fotografije kao novo polje. Kada je prijava uspješno dodana *dispatch-a* se akcija koja će prijavu spremiti i lokalno u *Redux store*. Naravno da prilikom komuniciranja s pozadinskim servisom može doći do pogreške. U tom slučaju prikazuje se pogreška i obavještava se korisnika. Na slici 4.18. prikazana je *onSubmit* funkcija.

```
function onSubmit() {
  setIsLoading(true);
  firestore()
    .collection('reports')
    .add({
      address: location?.address,
      latitude: location?.lat,
      longitude: location?.lng,
      description: description,
    })
    .then((response) => {
      if (!image) {
        return;
      }

      const ref = storage().ref(response.id);

      ref
        .putFile(image.path)
        .then(() => {
          ref
            .getDownloadURL()
            .then((url) => {
              response
                .update({
                  imageUrl: url,
                })
                .then(() => {
                  dispatch(
                    UserThunks.addReport({
                      imageUrl: url,
                      description: description,
                      address: location?.address,
                    }),
                  );
                  setIsSuccessModalVisible(true);
                  setIsLoading(false);
                  scrollViewRef?.current?.scrollTo({
                    x: FIRST_ACTION_POSITION,
                    y: 0,
                    animated: true,
                  });
                  setActionIndex(0);
                  setImage(undefined);
                  setDescription(undefined);
                  setLocation(undefined);
                  setPercentage(0);
                })
                .catch(() => {
                  setIsLoading(false);
                  ToastAndroid.show('Došlo je do pogreške', 2000);
                });
            })
            .catch(() => {
              setIsLoading(false);
              ToastAndroid.show('Došlo je do pogreške', 2000);
            });
        })
        .catch(() => {
          setIsLoading(false);
          ToastAndroid.show('Došlo je do pogreške', 2000);
        });
    })
    .catch(() => {
      setIsLoading(false);
      ToastAndroid.show('Došlo je do pogreške', 2000);
    });
}
```

Slika 4.18. *Prikaz onSubmit funckije*

4.5. Implementacija glavnih funkcionalnosti

Glavne funkcionalnosti aplikacije Zelenko su dodavanje fotografije, opisa i lokacije. Te funkcionalnosti implementirane su na glavnom zaslonu aplikacije. Za dodavanje fotografije i lokacije korisnik mora dozvoliti korištenje kamere i lokacije uređaja. Za traženje dopuštenja u aplikaciji korišten je *react-native-permissions* paket [20]. Dijagram toka traženja dopuštenja za *Android* uređaje korištenjem navedenog paketa prikazan je na slici 4.19. Na slici se traži dopuštenje korištenja kamere no postupak je isti za sva ostala dopuštenja.



Slika 4.19. *Dijagram toka traženja dopuštenja Android uređaja*

Prema dijagramu toka vidljivom na slici 4.2. napisan je *usePermissionRequest hook*. To je funkcija kojoj se prosljeđuju četiri argumenta: dopuštenje koje se želi dobiti, funkcija koja se poziva u slučaju da korisnik dopusti pristup odnosno odbije pristup i funkcija koja se poziva u slučaju da je dopuštenje u jednom trenutku bilo blokirano. U slučaju da je dopuštenje odbijeno korisniku se prikazuje skočni prozor na kojem piše da bez dopuštenja neće moći koristiti aplikaciju i nakon toga ga se opet pita za dopuštenje. U slučaju kada je dopuštenje blokirano korisniku se također prikazuje skočni prozor s istom porukom, ali ovaj put se otvaraju postavke gdje može omogućiti aplikaciji različite dozvole.

```
export function usePermissionRequest(
  permission: Permission,
  onAllow: () => void,
  onDeny: () => void,
  onBlocked: () => void,
): () => void {
  function requestPermission() {
    request(permission).then((permissionResult) => {
      switch (permissionResult) {
        case 'granted':
          onAllow();
          break;
        case 'denied':
          onDeny();
          break;
        case 'blocked':
          onBlocked();
          break;
      }
    });
  }

  return requestPermission;
}
```

Slika 4.20. *usePermissionRequest hook*

4.5.1. Dodavanje fotografije

Dodavanje fotografija omogućeno je korištenjem *react-native-image-crop-picker* paketa [21]. To je paket koji ima već gotove funkcije za dodavanje fotografija putem kamere ili iz memorije uređaja. Pri izradi aplikacije korištene su dvije funkcije iz paketa, a to su: *openPicker* i *openCamera.* Obje funkcije kao argument primaju konfiguracijski objekt. Neka od konfiguracijskih svojstva su visina i širina slike, tip medija, korištenje *base64* enkodiranog stringa itd. Popis svih svojstava može se vidjeti u dokumentaciji paketa. U svrhu korištenja *react-native-image-crop-picker* paketa napisan je *ImageService* servis (Slika 4.21.).

```
import ImagePicker, { Image } from 'react-native-image-crop-picker';

async function selectPhotoFromGalleryAsync(): Promise<Image> {
  const image = await ImagePicker.openPicker({
    includeBase64: true,
    mediaType: 'photo',
    width: 400,
    height: 400,
    cropping: true,
  });

  return image as Image;
}

async function takePhotoAsync(): Promise<Image> {
  const image = await ImagePicker.openCamera({
    includeBase64: true,
    mediaType: 'photo',
    width: 400,
    height: 400,
    cropping: true,
  });

  return image as Image;
}

export const ImageService = {
  selectPhotoFromGalleryAsync,
  takePhotoAsync,
};
```

Slika 4.21. *Prikaz ImageService servisa*

ImageService koristi funkcije paketa te omogućava dodavanje fotografije putem kamere ili iz memorije uređaja. Također na slici 4.21. mogu se vidjeti i konfiguracijski objekti koji su korišteni prilikom pozivanja funkcija *openCamera* i *openPicker*.

4.5.2. Dodavanje lokacije

Za dodavanje lokacije korišten je *Google Geocoding* API u kombinaciji s *react-native-geolocation-service* i *axios* paketom [22, 23, 24]. Za implementaciju rješenja potrebno je prijaviti se na *Google Maps* platformu i dodati projekt [25]. Geokodiranje je proces pretvaranja adresa u geografske koordinate. Također postoji i proces obrnutog geokodiranja gdje se uz pomoć geografske širine i dužine može dobiti adresa. Dovoljno je poslati zahtjev s određenim parametrima i *Google Geocoding* će vratiti odgovarajuću adresu. Na slici 4.22. prikazana je funkcija kojom se poziva *get* zahtjev pomoću *axios* paketa.

```
return axios.get<GeocodingResponse>(
  `https://maps.googleapis.com/maps/api/geocode/json?latlng=${coordinates.latitude},
  ${coordinates.longitude}&language=hr&result_type=street_address&location_type=ROOFTOP&key=${geoCodingApiKey}`,
);
```

Slika 4.22. *Prikaz get zatjeva Geocoding API-ja*

Parametri koji se moraju proslijediti su geografska širina i dužina te ključ. Ključ se može generirati unutar *Google Maps* platforme, dok je za geografsku širinu i dužinu korišten *react-native-geolocation-service* paket. Taj paket omogućava poziv *getCurrentPosition* funkcije koja vraća *position* objekt koji sadrži geografsku širinu i dužinu. Zatim se te vrijednosti mogu proslijediti kao parametri prilikom pozivanja *get* zahtjeva na *Geocoding* API. Funkcija za dohvaćanje adrese prikazana je na slici 4.23.

```
async function getAddress() {
  let coordinates: GeoCoordinates;
  let coordinatesError: string;
  let geoCodingApiKey: string;
  let firestoreError: string;

  await getCoordinates()
    .then((position) => {
      coordinates = position;
    })
    .catch((err: GeoError) => (coordinatesError = err.message));

  await getGeocodingApiKey()
    .then((key) => (geoCodingApiKey = key))
    .catch((err) => (firestoreError = err));

  if (coordinatesError) {
    return coordinatesError;
  }

  if (firestoreError) {
    return firestoreError;
  }

  return axios.get<GeocodingResponse>(
    `https://maps.googleapis.com/maps/api/geocode/json?latlng=${coordinates.latitude},
    ${coordinates.longitude}&language=hr&result_type=street_address&location_type=ROOFTOP&key=${geoCodingApiKey}`,
  );
}
```

Slika 4.23. *Funkcija za dohvaćanje adrese*

4.6. Način korištenja aplikacije i testiranje

U nastavku je opisan način na koji se koristi aplikacija te ispitivanje rada aplikacije.

1. Korisniku je prikazan *onboarding* zaslon. Korisnik ima mogućnost preskočiti *onboarding* ili prelistati sve zaslone i završiti *onboarding*
2. Klikom na *Info* gumb na vrhu zaslona korisnik može ponoviti *onboarding* proces
3. Na glavnom zaslonu korisnik može listati opcije na tri načina:
    I. Pomakom prsta
    II. Korištenjem navigacije na dnu zaslona
    III. Pritiskom na tipke za navigaciju
4. Za dodavanje fotografije potrebno je pritisnuti tipku *Dodaj fotografiju* nakon čega će se otvoriti skočni prozor s tri opcije (potrebna dozvola korištenja kamere):
    I. Otvori kameru (otvara se kamera mobitela)
    II. Izaberi iz galerije (otvara se galerija mobitela)
    III. Odustani (korisnika se vrača na glavni zaslon)
5. Za dodavanje opisa potrebno je pritisnuti tipku *Dodaj opis* nakon čega se otvara skočni prozor u koji se može upisati željeni opis. Nakon toga klikom na tipku *Dodaj* opis će biti dodan
6. Za dodavanje lokacije potrebno je pritisnuti tipku *Dodaj lokaciju* nakon čega će se lokacija automatski dodati (potrebna dozvola korištenja lokacije)
7. Za bolje rezultate lokacije potrebno je uključiti lokaciju u postavkama mobitela
8. Prijava se šalje klikom na gumb sa strelicom prema desno u navigaciji na dnu zaslona
9. Klikom na *Moje prijave* u gornjoj navigaciji korisnik može vidjeti svoje prijave

Nad aplikacijom je obavljeno funkcionalno testiranje na više virtualnih i stvarnih *Android* uređaja. Testirane su sve implementirane funkcionalnosti i nije pronađena ni jedna greška, odnosno sve funkcionalnosti su pokrivene i daju pozitivne rezultate.

5. ZAKLJUČAK

Svaki pojedinac ima ulogu u očuvanju okoliša. Stoga je bitno da je svaki pojedinac svjestan svog utjecaja na okoliš jer i najmanjim djelima utječe na njegovu promjenu. Cilj ovog rada je osnažiti ekološku svijest građana razvojem *Android* aplikacije za promicanje i olakšavanje sudjelovanja javnosti u rješavanju problema sa zelenilom grada. Građani putem aplikacije Zelenko mogu na brz i jednostavan način prijaviti uočeni problem komunalnom poduzeću. Aplikacija se razvija s *React Native* tehnologijom. *React Native* je odlično rješenje za razvoj aplikacije Zelenko jer podržava višeplatformski razvoj koji omogućava daljnji razvoj za iOS uređaje. Također, razvoj je brži i jeftiniji. Prilikom razvoja aplikacije koristi se i set različitih alata koji optimiziraju proces razvoja aplikacije. Alati se biraju proizvoljno, no bitno je da developer dobro poznaje odabrane alate i da zna njihove mogućnosti. Glavne funkcionalnosti se razvijaju uključivanjem biblioteka koje sadrže već gotova rješenja potrebna za aplikaciju Zelenko. To je još jedna od prednosti *React Native-a* jer se ne mora gubiti vrijeme na razvijanje nečega što već postoji. Testiranjem aplikacije na virtualnim i stvarnim uređajima ustanovljeno je da je aplikacija funkcionalna i da zadovoljava sve definirane funkcionalne zahtjeve. Kako bi se aplikacija Zelenko mogla koristiti, treba se riješiti problem pregledavanja prijava. To može biti web aplikacija koja se spaja na Firebase pozadinski servis aplikacije Zelenko i koja omogućava upravljanje prijavama korisnika. Također, može se uvesti skupljanje bodova i osvajanje nagrada za uspješne prijave što bi potaklo više građana na sudjelovanje u rješavanju problema sa zelenilom grada.

LITERATURA

[1] D. O'Rourke, GoodGuide, University of California-Berkley, dostpuno na: https://goodguide.com/ [25.5.2020.]

[2] Electronic Recyclers International, RecycleNation, dostpuno na: https://recyclenation.com/ [25.5.2020.]

[3] Dropcountr, dropcountr app, dostupno na: https://www.dropcountr.com/ [25.5.2020.]

[4] Facebook, Inc., React Native, dostupno na: https://reactnative.dev/docs/intro-react-native-components [15.9.2021.]

[5] Microsoft, TypeScript, Redmond, Boston, San Francisco, Dublin dostupno na: https://www.typescriptlang.org/docs/handbook/intro.html [15.9.2021.]

[6] Microsoft, Visual Studio Code, Seattle, dostupno na: https://code.visualstudio.com/docs [15.9.2021.]

[7] GNU General Public License, Git, dostupno na: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control [15.9.2021.]

[8] GitHub, Inc., GitHub, dostupno na: https://github.com/features [15.9.2021.]

[9] Google, Firebase, dostupno na: https://firebase.google.com/docs/build [15.9.2021.]

[10] Figma, dostupno na: https://www.figma.com/ [15.9.2021.]

[11] Bitrise Ltd., Bitrise, London, Budapest, dostupno na: https://devcenter.bitrise.io/getting-started/index/ [15.9.2021.]

[12] S. Baumgartner, Typescript in 50 lessons, Smashing Media AG, Freiburg, Njemačka, 2020.

[13] V. Vlahek, Avoiding the messy git history, Prototyp, Osijek, 2019., dostupno na: https://blog.prototyp.digital/avoiding-the-messy-git-history/ [15.9.2021.]

[14] Streamline, streamline ilustracije, dostupno na: https://streamlinehq.com/ [15.9.2021.]

[15] Yarn Contributors, Yarn upravitelj paketa, dostupno na: https://yarnpkg.com/ [15.9.2021.]

[16] npm, Inc., npm upravitelj paketa, dostupno na: https://www.npmjs.com/ [15.9.2021.]

[17] Prettier formater koda, dostupno na: https://prettier.io/docs/en/index.html [15.9.2021.]

[18] OpenJS Foundation, ESlint, dostupno na: https://eslint.org/docs/user-guide/getting-started [15.9.2021.]

[19] D. Abramov, Redux, dostpuno na: https://redux.js.org/introduction/getting-started [15.9.2021.]

[20] M. Acthernoene, React Native permissions paket, dostupno na: https://github.com/zoontek/react-native-permissions [15.9.2021.]

[21] I. Pušić, React Native image crop picker paket, dostupno na: https://github.com/ivpusic/react-native-image-crop-picker [15.9.2021.]

[22] Google Developers, Geocoding API, dostupno na: https://developers.google.com/maps/documentation/geocoding/overview [15.9.2021.]

[23] I. Rifat, React Native geolocation service, dostupno na: https://github.com/Agontuk/react-native-geolocation-service [15.9.2021.]

[24] Axios paket, dostupno na: https://github.com/axios/axios [15.9.2021.]

[25] Google Developers, Google maps konzola, dostupno na: https://console.cloud.google.com/projectselector2/home/dashboard?supportedpurview=project [15.9.2021.]

[26] Statista Research Department, *Number of available applications in the Google Play Store from December 2009 to July 2021,* Statista, 9.10.2021., dostupno na: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ [14.9.2021.]

[27] J. Laklija, Razvoj ekološke svijesti u Republici Hrvatskoj, Sveučilište u Rijeci, Filizofski fakultet, Rijeka, 2019

[28] I. Cifrić, Socijalna Ekologija, Zagreb: Globus, 1989.

[29] J. Loeliger, M. McCullough, Version control with Git, O'Reilly Media, Inc., 2012.

[30] H. Singh, M. Tanna, Serverless Web Applications with React and Firebase, Packt Publishing, 2018.

[31] Oroeco Inc., Oroeco App, dostupno na: https://www.oroeco.com/ [19.9.2021.]

[32] Refill aplikacija, dostupno na: https://www.refill.org.uk/about/ [19.9.2021.]

SAŽETAK

U diplomskom radu razvijena je *Android* aplikacija za promicanje i olakšavanje sudjelovanja javnosti u rješavanju problema sa zelenilom grada. Cilj aplikacije Zelenko je osnažiti ekološku svijest građana i omogućiti jednostavno i brzo prijavljivanje problema. Aplikacija je razvijena u *React Native-u*. Prilikom razvoja korišteni su različiti alati i tehnologije što je optimiziralo sam proces razvoja. Neki od korištenih alata su: pozadinski servis *Firebase*, sustav kontrole verzija koda *Git*, kontinuirana isporuka i izgradnja aplikacije na *Bitrise-u* i drugi.

Ključne riječi: Ekološka svijest, Firebase, Git, mobilna aplikacija, React Native

ABSTRACT

ANDROID APPLICATION FOR PROMOTION AND FACILITATION OF PUBLIC PARTICIPATION IN SOLVING THE PROBLEM WITH THE GREENERY OF THE CITY

In the thesis, an Android application was developed to promote and facilitate public participation in solving problems with the greenery of the city. The goal of the Zelenko application is to strengthen the environmental awareness of citizens and enable easy and fast reporting of problems. The application was developed in React Native. During development, various tools and technologies were used, which optimized the development process itself. Some of the tools used are: Firebase background service, Git code version control system, continuous delivery and application building on Bitrise and others.

Keywords: Environmental Awarness, Firebase, Git, Mobile Application, React Native