Izrada aplikacije za planiranje letova bespilotnih letjelica korištenjem softverskog okvira React Native

Frlić, Filip

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Frlić

IZRADA APLIKACIJE ZA PLANIRANJE LETOVA BESPILOTNIH LETJELICA KORIŠTENJEM SOFTVERSKOG OKVIRA REACT NATIVE

ZAVRŠNI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Frlić

Matični broj: 45176/16–R

Studij: Informacijski sustavi

IZRADA APLIKACIJE ZA PLANIRANJE LETOVA BESPILOTNIH LETJELICA KORIŠTENJEM SOFTVERSKOG OKVIRA REACT NATIVE

ZAVRŠNI RAD

Mentor/Mentorica:

Dr. sc. Marko Mijač

Varaždin, studeni 2021.

Filip Frlić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Rad je podijeljen na teorijski i praktični dio. U teorijskom dijelu su objašnjeni React, React Native softverski okviri te Redux, paket za upravljanjem stanja Javascript aplikacija. Također su opisani neki koncepti Javascript jezika koju su bitni u React Native okruženju i kratka usporedba React Native okvira sa sličnim višeplatformskim tehnologijama kao što su Ionic i Flutter okvirima. U praktičnom dijelu je opisan razvoj aplikacija za planiranje letova bespilotnih letjelica, također je opisana arhitektura pakiranja po značajkama, te neki od najpopularnijih paketa koji olakšavaju rad sa spomenutim tehnologijama.

Ključne riječi: React, React Native, Redux, Javascript, mobilni razvoj, višeplatormski razvoj, pakiranje po značajka

Sadržaj

1. Uvod

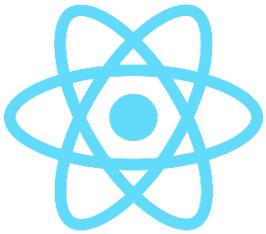
Svake godine mobilne aplikacija postaju sve popularnije te sve više aplikacija koje su isprva bile pisane za web također dobivaju i svoju mobilnu verziju. S toga je izrazito važno poduzećima da što brže i jeftinije razviju i mobilne aplikacije svojih proizvoda.

Upravo ovdje višeplatformski razvoj dolazi do izražaja. On omogućava razvoj aplikacija za više platformi, koje su najčešće iOS i Android, ali tu ulaze i manje poznate tehnologije kao AndroidTv, Apple Watch i slične. Kod odabira višeplatformskog okvira imamo nekoliko mogućnosti, među kojima su React Native, koji je proizvod Mete, Flutter koji je proizvod Googla, te još neke manje poznate tehnologije kao Ionic, Xamarin, Nativescript itd.

U ovome radu opisati ću razvoj mobilne aplikacije pomoću okvira React Native. U prvome dijelu ćemo se upoznati s teoretskim osnovama React-a i React Native-a, te dijela Javascript jezika koji je posebno bitan za razvoj React Native aplikacija. Također ćemo se upoznati sa Redux i Redux Toolkit paketom, koji će nam služiti kao spremnik stanja aplikacije. Na kraju teoretskog dijela kratko ću usporediti React Native sa ostalim višeplatformskim okvirima kao što su Flutter i Ionic.

U praktičnom dijelu izraditi ću aplikaciju pomoću okvira React Native, uz pomoć Redux-a i Redux Toolkita kao spremnika stanja. Glavni cilj aplikacije je simulacija i planiranje letova bespilotnih letjelica, uz koje ću ukratko opisati sve vanjske pakete uključene u projekt, te opisati pakiranje po značajkama i arhitekturu kojom ćemo pisati aplikaciju.

2. React i React Native



Slika 1: React Logo

React je besplatan i open-source programski okvir kreiran za ubrzani razvoj klijentskih (engl. *frontend*) aplikacija baziran na Javascript-u. Kreiran je od strane zaposlenika *Mete* (tadašnjeg *Facebooka*) Jordana Walkea kako bi olakšao izgradnju korisničkih sučelja web aplikacija. Prva verzija React-a je izdana 29.05.2013. Održava se od strane *Mete*, ali i velike zajednice drugih kompanija i samostalnih programera. (React - A javascript library for building user interfaces, 2021)

Za malo manje od 2 godine, dijelom zbog uspješnosti Reacta, izdana je i prva inačica React Native-a. React Native je, umjesto za web, namijenjen „native“ razvoju, kojem pripadaju ponajprije aplikacije za Android i iOS sustave, ali također omogućuje i razvoj za platforme kao što su Windows, macOS, Android TV, tvOS, pa čak i na Oculusu. Kako su sami React i React Native napravljeni od strane Mete, nije potrebno posebno naglašavati da ga oni koriste za sve svoje aplikacije kao što su *Facebook*, *Instagram*, *WhatsApp* i ostale, javnosti manje poznate aplikacije. Dočarati kako izgleda jednostavna „Hello World“ aplikacija najbolje je prikazati primjerom koda :

```
const root = ReactDOM.createRoot(document.getElementById('root'))

root.render(<h1>Hello, world!</h1>)
```

Iz gornjeg primjera je vidljivo da je koristimo dvije funkcije iz klase ReactDOM. Prva funkcija stvara korijen (*engl. root*) elementa, koji ćemo kasnije manipulirati pomoću našeg koda. Kada smo uspješno deklarirali korijen, sve što je onda potrebno je pozvati *render* funkciju, koja kao parametar prima html ili JSX kod koji će se prikazati u pregledniku.

2.1. Razvojno okruženje

Na samome početku svakoga React-Native projekta moramo odlučiti koji ćemo CLI (*command line interface*) koristiti. Ovdje nam se nameću 2 mogućnosti : Expo ili React Native CLI. Započeti razvoj je najlakše pomoću Expo-a. Expo je set alata koji su izgrađeni isključivo za React Native, i iako ima mnoštvo značajki, najvažnija je jednostavnost korištenja i brzina početka razvoja. Uz Expo nam nije potreban niti virtualni uređaj, već se aplikacija može testirati na postojećem fizičkom uređaju koji je spojen na WiFi mrežu ili putem USB-a. Međutim, uz prednosti uvijek dolaze i mane. Expo je odličan alat za početak i učenje, ali ako su nam za našu mobilnu aplikaciju potrebne neke napredne značajke, kao korištenje nativnih modula, tada ćemo morati posegnuti za React Native CLI-em.

Također, jedna stvar koju moramo imati na računalu prije početka je i NPM (node package manager) koji nam omogućuje jednostavno instaliranje potrebnih JavaScript paketa koje ćemo koristiti u projektu. Prvi od tih paketa je Expo ili React Native CLI, ovisno za koji pristup smo se odlučili. Prva komanda će izgledati :

```
npm install -g expo-cli
```

Komanda *install* radi upravno ono što i označava, zastava *-g* označava da paket koji preuzimamo želimo koristiti globalno, dok je *expo-cli* ime paketa koji preuzimamo. Ukoliko koristimo expo, potrebno je još generirati naš projekt komandom:

```
expo init AwesomeProject
```

Prva komanda će instalirati sve potrebne pakete, a druga komanda će inicijalizirati predložak projekta i nakon toga možemo odmah pokrenuti našu aplikaciju na fizičkom uređaju.

Ukoliko se pak odlučimo koristiti React Native CLI, potreban nam je veći broj koraka za postavljanje projekta. Prvo moramo instalirati Node i Java Development Kit, a nakon toga preuzeti i instalirati Android Studio i Xcode (ukoliko koristimo Mac OS). Kada je sva potrebna konfiguracija gotova, projekt generiramo pomoću slijedeće linije :

```
npx react-native init AwesomeProject
```

2.2. Osnove JavaScripta

Kako bi mogli razumjeti React, prvo moramo zavladati osnovama tehnologija koje ga pogone. React je temeljen na Javascriptu, jeziku kojega je 1995. u tjedan dana napisao Brendan Eich kao skriptni jezik za korištenje u pregledniku Netscape Navigator. Nakon nekoliko iteracija i poboljšanja, sve veće tadašnje kompanije su ga prihvatile i počele koristit, i tako je Javascript postao nezamjenjivi dio današnjeg Interneta.

Prvi pojam s kojim ćemo se pobliže upoznati u Javascriptu su varijable. Javascript je Labavo tipizirani jezik *(engl. Loosely typed language)*, što znači da varijabla može mijenjati svoj tip nakon što je deklarirana.

Drugi pojam koji ćemo uvesti je destrukturiranje. Ovaj jednostavan koncept se uvelike koristi kod zakački, posebice s *useState* zakačkom, s kojom ćemo se pobliže upoznati kasnije u radu. Destrukturiranje je pojam kojim možemo „izvaditi“ referencu varijable unutar liste ili objekta izvan nje, bez da mijenjamo originalnu vrijednost.

Recimo da imamo listu u koja sadrži dva elementa:

```
const list = [1, 2];
```

Vrijednosti unutar te liste možemo dohvatiti na tradicionalan način kao što je:

```
const first = list[0];

const second = list[1];
```

međutim postoji i jedan „ljepši“ način, pogotovo kada moramo dohvatiti dvije ili varijable unutar liste ili objekta. Linija koda ispod radi potpuno istu stvar kao i ona iznad :

```
const [first, second] = list;
```

Važno je spomenuti da istu stvar možemo ostvariti i sa objektima, ali tu nam redoslijed nije bitan :

```
const object = { first : 1, second : 2 };

const { second, first } = object;
```

2.3. Osnovni koncepti Reacta

Glavni koncept samoga React razvojnog okvira je *JSX (JavaScript Syntax Extension)*. Na prvu će nam on izgledati kao još jedan generator predložaka *(engl. Templating engine),* sličan koji koriste ostali okviri koji manipuliraju HTML-om, međutim JSX ima punu moć JavaScripta unutar sebe. Recimo da imamo jedan element koji izgleda ovako:

```
const element = <h1>Hello, world</h1>
```

Na prvu, varijabla element izgleda pomalo neuobičajeno. To nije string, jer između vrijednosti nema navodnika, definitivno nije niti broj, nego je jsx element. Ako se osvrnemo na prvi primjer, element možemo zamijeniti kao 1. parametar render funkcije, tada bi ona izgledala ovako:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(element);
```

No sve ovo je i dalje statičan html. Poanta JSX-a je da nam omogući dinamičnost na najjednostavniji mogući način. JSX sintaksu je lako primijetiti u kodu, ona se uvijek nalazi unutar vitičastih zagrada. Ako nadogradimo nas prijašnji primjer JSX kodom, on bi izgledao ovako :

```
const helloMessage = 'Hello'

const element = <h1>{helloMessage}, world</h1>


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(element);
```

Vidimo da smo prvo definirali varijablu *helloMessage*, i nju kasnije koristimo u varijabli *element* unutar vitičastih zagrada.

Slijedećim konceptom ćemo definirati način pisanja komponenti. Tu imamo dvije mogućnosti: funkcijske komponente te komponente bazirane na klasama. Funkcijske komponente su noviji način pisanja komponenti, te također nam omogućuju korištenje zakački, koje uvelike olakšavaju rad na komponentama.

Prije funkcijskih komponenti, komponente su se definirale pomoću klasa. Iako slične sintakse, komponente bazirane na klasama su zastarjeli način pisanja React i React Native koda, međutim mogu se miješati unutar projekta, ali se preporučuje sve nove komponente pisati isključivo pomoću funkcijskih komponenti. Kako bi se upoznali s komponentama baziranim na klasama, slijedi jednostavni primjer kako bi razlikovali pristupe.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Na primjeru iznad prvo vidimo funkcijski pristup, a ispod njega i pristup pomoću klasa.

Treći koncept govori o načinu korištenja podataka u React i React Native aplikacijama su stanje (*engl. state*) te parametri (*engl. props*).

Stanje opisuje komponentu u kojem se ona trenutno nalazi. Za razliku od parametara, ono se može mijenjati, a ovisno o stanju se mijenja i izgled aplikacije. Stanje mijenjamo i čitamo ga pomoću *useState* zakačke, a ukoliko koristimo komponente bazirane na klasama, stanje ćemo definirati kao objekt unutar *this.state* varijable. Za razliku od stanja, parametre šaljemo komponentama kada ih definiramo u JSX-u. Kao primjer parametra možemo se poslužiti primjerom iznad gdje smo pokazali razliku između funkcijskih i komponenti baziranih na klasama. U primjeru s funkcijskim komponentama, funkcija *Welcome* prima varijablu *props*, koja sadrži parametre s kojima je funkcija pozvana. Kako bi pristupili parametrima komponente definirane pomoću klasa, potrebno je koristiti *this.props* za pristup parametrima. Što se tiče pozivanja komponenti, pristup je jednak za oba slučaja, a izgleda ovako:

```
<Welcome name="World" />
```

2.4. React zakačke

Jedna od najvažnijih novih osobina Reacta su zakačke – funkcije koje proširuju ponašanje funkcijskih komponenti. Umjesto oslanjanja na klase kako bi gradili komponente koje sadrže stanje, sada možemo koristiti API zakački kako bi mogli koristiti stanje u funkcijskim komponentama. (Derks & Boduch, 2020)

2.4.1. useState

Početi ćemo sa najjednostavnijom zakačkom, koji je ***useState***.

```
const [data, setData] = useState('');
```

Kod iznad možda izgleda zastrašujuće na prvu, međutim on je zaista jednostavan. Svaka *useState* funkcija (ili zakačka) nam vraća listu u kojoj su dvije stvari : taj podatak i funkciju kojom mijenjamo taj podatak. Također vidimo da useState funkcija prima jedan parametar, a to je početna vrijednost varijable. Ovdje dolazi pitanje: zašto nam treba komplicirana funkcija koja samo sprema podatke kao i obična varijabla? Odgovor leži u tome da će React ponovo generirati DOM *(engl. : Document Object Model)* svaki puta kada se stanje promjeni. Ukoliko napravimo običnu varijablu u JavaScriptu i promijenimo joj vrijednost, ta promjena se neće vidjeti sve dok se DOM ne generira ponovo.

2.4.2. useEffect

Kako korištenje jedne zakačke nije previše korisno, uvedimo i drugu zakačku, koja se zove ***useEffect***, a izgleda ovako:

```
useEffect(() => {

   fetch(URI).then((data) => setData(data))

},[URI])
```

useEffect zakačka će se izvršiti svaki puta kada se stranica generira, koje se može dogoditi ili na prvom prikazu stranice, ili kod ponovnog generiranja stranice zbog promjene stanja. Kao prvi parametar prima funkciju koja će se izvršiti prilikom generiranja, a kao drugi prima listu ovisnosti. Lista ovisnosti je lista stanja (statea), što znači kada se promjeni neko stanje o kojem ovisi useEffect zakački, stranica, tj. komponenta će se ponovo generirati. Ukoliko je lista ovisnosti prazna, funkcija će se izvršiti samo prilikom prvog generiranja.

2.4.3. useContext

State nam je koristan kada ga koristimo u komponenti u kojoj ga i deklariramo. Naravno da postoje načini kako to stanje poslati s jedne komponente u drugu, ali postoji i bolji način za korištenje „globalnih“ varijabli.

Tu dolazimo do *useContext* zakačke. Recimo da koristimo samo jednu globalnu varijablu koja označava koristi li naša aplikacija tamnu temu, tada će naš kontekst izgledati ovako:

```
const context = { isDark : true}

const ThemeContext = createContext(context)
```

U liniji iznad kontekst spremamo u varijablu ThemeContext, koja sadrži jednu vrijednost, varijablu *isDark.* Samo to nam nije dovoljno za korištenje konteksta, također je potrebno „omotati“ sav kod u kojem želimo koristiti kontekst u poseban element „dobavljač“ *(engl. provider).* Najjednostavniji primjer korištenja konteksta izgleda ovako :

```
const context = { isDark : true};
const ThemeContext = createContext(context);
function App(){
   return (
      <ThemeContext.Provider value={context}>
            <Button />
      </ThemeContext.Provider>
    );
}
function Button() {
  const { isDark } = useContext(ThemeContext);
  const buttonColor : isDark ? 'white' : 'black';

  return (
      <button style={{ background: theme.background}}>
          Button!
      <button>
   );
}
```

U primjeru iznad vidimo da u prve dvije linije deklariramo kontekst. U komponenti App smo omotali komponentu Button u context provider, i kasnije u komponenti *Button* možemo pristupiti varijabli *isDark,* te ovisno o njoj mijenjamo pozadinsku boju gumba.

2.4.4. useReducer

Kao što smo već vidjeli, useState zakačka nam odlično služi kao spremnik jednostavnih vrijednosti. Do problema dolazi kada moramo manipulirati većim brojem međuzavisnih vrijednosti. Taj problem rješava reducer zakačka. On osigurava da će iste promjene uvijek rezultirati jednakim izračunom novoga stanja. Ovaj koncept je još poznat kao čiste funkcije *(engl. pure functions)* koji u širem smislu objašnjava kako, uz jednake ulazne parametre neke funkcije, njezin rezultat uvijek mora biti isti.

Reducer zakačka prima 2 parametra, funkciju reducer, kojom mijenjamo stanje, i početno stanje, kao drugi parametar. U slijedećem primjeru ćemo vidjeti jednostavni brojač pomoću reducera. Prvo ćemo definirati reducer funkciju:

```
function reducer(state, action) {
    switch (action.type) {
        case 'increment':
            return {count: state.count + 1};
        case 'decrement':
            return {count: state.count - 1};
        default:
            return state;
        }
}
```

Vidimo da reducer funkcija prima 2 parametra, trenutno stanje i akciju kojom želimo promijeniti stanje. Pomoću switch case-a definirao akcije koje su moguće, te tako imamo 2 akcije : increment i decrement, koje povećavaju, odnosno smanjuje vrijednost count koja se nalazi u stanju.

Kada smo definirali reducer funkciju, možemo je početi koristiti u našim komponentama:

```
function Counter() {
  const [state, dispatch] = useReducer(reducer, {count: 0});
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

Za početak definiramo reducer zakačku, koji prima reducer funkciju koju smo ranije definirali, i početno stanje, koji je na početku 0. Kao rezultat zakačke dobivamo state i dispatch funkciju. State sadrži nove, ažurirane vrijednosti, dok je dispatch funkcija kojom manipuliramo tim stanjem. Na kraju vidimo vrijednost count koju dohvaćamo iz stanja, i 2 gumba kojima pokrećemo različite akcije, a to su increment i decrement, koje smo definirali u reducer funkciji.

2.5. Kompozicija i nasljeđivanje

Često čujemo kako je nasljeđivanje jako važan dio objektno orijentiranog programiranja. Iako *Javascript* podržava klase, on daje veću prednost funkcijskom programiranju, a prema tome i konceptu kompozicije.

Najjednostavnije rečeno, kompozicija i nasljeđivanje su dva različita pristupa pisanju koda. Oni rješavaju isti problem na dva različita načina. U slučaju nasljeđivanja, u običnom jeziku ćemo reći: „Auto je vozilo“, međutim u slučaju kompozicije, mogli bi reći: „Auto sadrži gume“. Vidimo da su ovo dva različita stajališta, gdje kod kompozicije gradimo stvari od nule, od manjih dijelova prema većima, dok je sa strane nasljeđivanja obrnuta stvar, od nekih općenitih i generalnih stvari gradimo manje. Također možemo reći da kod nasljeđivanja gradimo klase po tome što one jesu, a kod kompozicije dizajniramo stvari po onome što one rade (Composition vs Inheritance - React, 2021).

Recimo da imamo primjer toga već spomenutog vozila i auta, te želimo dodati novo vozilo, recimo bicikl koje također sadrži gume. Pošto su gume sadržane i na autu i biciklu, praksa i logika nalažu da gume prebacimo u klasu vozilo , pošto ga i bicikl i auto sadrže. Međutim, što ćemo napraviti ako želimo dodati i brod? On je također vozilo, ali nema gume.

Sada moramo ili raditi neku međuklasu, recimo vozila sa gumama, ili opet vratiti gume na auto i bicikl. Tu dolazimo do problema kojega kompozicija rješava na bolji način. Kod nje ćemo krenuti obrnutim putem, i reći od početka da auto sadrži gume, prozore, motor itd, dok bicikl sadrži gume, pedale, kočnice itd.

Kako je *React* građen uz pomoć *Javascipta*, prirodno je da on preferira metode funkcijskog programiranja, koje sadrže i daju prednost kompoziciji nad nasljeđivanjem. Svaki puta kada šaljemo parametre nekoj komponenti, koristimo kompoziciju, tj. gradimo komponentu od malih dijelova. Na primjeru gumba koji otvara neku stranicu, možemo primijetiti kompoziciju:

```
<Button label='Google' color='blue' url='https://google.com'
textColor='red' />
```

Na primjeru koda iznad vidimo da jedan gumb slažemo od mnoštva malih dijelova kojima definiramo što taj gumb radi, pa tako definiramo boju gumba, tekst i boju koja se prikazuje na gumbu, i na kraju URL na koji nas gumb vodi.

2.6. Testiranje

Kao i kod svakoga poznatoga programskog jezika ili okvira, i React ima svoj način provođenja testiranja. React ima alate za testiranje svake komponente zasebno, ali također i za testiranje cjelokupne aplikacije, međutim, najpopularniji su upravo *end-to-end* testovi, tj. testovi koji provjeravaju cjelokupni sustav. Također je važno napomenuti da su testovi za React i React Native aplikacije gotovo jednaki, tako da isti kod možemo gotovo uvijek koristiti na obje platforme.

Najpopularniji paketi za testiranje aplikacija su *Jest* i *React Testing Library,* a mi ćemo se upoznati sa *Jest-om,* pošto je on ipak popularniji, i on se može korisiti na bilo kojoj *Javascript* aplikaciji, dok je *React Testing Library* vezan uz *React*-ov ekosustav.

Svaka datoteka koja sadrži test koji napišemo će završavati sa „test.js“ ekstenzijom, to je još jedan standard kojega se drže programeri kako bi odmah prepoznali razlikovati testove od ostalog koda. U nastavku slijedi primjer jednog jednostavnog testa.

```
import renderer from 'react-test-renderer';
import Link from '../Link';

it('changes the class when hovered', () => {
  const component = renderer.create(
    <Link page="http://www.facebook.com">Facebook</Link>,
  );
  let tree = component.toJSON();
  expect(tree).toMatchSnapshot();

  renderer.act(() => {
    tree.props.onMouseEnter();
  });

  tree = component.toJSON();
  expect(tree).toMatchSnapshot();


  renderer.act(() => {
    tree.props.onMouseLeave();
  });

  tree = component.toJSON();
  expect(tree).toMatchSnapshot();
});
```

Na početku datoteke uvodimo potrebne pakete za testiranje. Svaki test započinje funkcijom *it*. Ta funkcija, kao i sve ostale u *Jest* paketu, je napravljena da što više sliči pravom jeziku, kako bi se još olakšalo pisanje i čitanje testova.

Funkcija „it“ prima 2 parametra, prva je opis testa, dok je drugi parametar funkcija kojoj kreiramo potrebni element za testiranje. Tada spremamo element *DOM-a* u zasebnu varijablu, i pretvaramo je u JSON format kako bi mogli iz nje lakše čitati podatke.

Tada nam dolazi nova funkcija, naziva *expect*. Njome provjeravamo da li je došlo do nekih nepredviđenih promjena na zadanom elementu.

Pošto u našem testu provjeravamo da li će se promijeniti CSS klasa kada pređemo mišem preko linka, moramo taj prijelaz generirati i na testu. To radimo *act* funkcijom, koja će

pozvati funkciju *onMouseEnter.* Kada je naša radnja izvršena, ponovo ćemo element pretvoriti u *JSON* format i provjeriti da li je došlo do nepredviđenih promjena na elementu. Ukoliko ih nema, tj, klasa je promijenjena, test uspješno prolazi.

Ostaje nam još samo provjeriti da li se CSS klasa vraća na originalno stanje kada se miš makne sa elementa, a to provjeravamo na isti način kao i ranije, samo sa *onMouseLeave* funkcijom.

Ukoliko želimo usporediti prijašnji test sa AAA (arrange, act, assert) principom, vidimo da u početku testa definiramo komponente i potrebne varijable, kao što su *component* i *tree*. Kada je postavljanje gotovo, možemo prijeći na pokretanje testova (act). Njih ćemo prepoznati pomoću ključne riječi *act*, koji je funkcija koja kao parametar prima drugu funkciju kojom definiramo radnju koju izvodimo nad testom. Na kraju testa slijedi provjera rezultata testa (assert), koju u primjeru možemo prepoznati pomoću funkcije *expect*.

2.7. Najpopularnije biblioteke

Kako su i React i React Native u sustavu otvorenog koda, postoje mnogobrojne biblioteke koje olakšavaju rad sa navedenim tehnologijama. Kako ima mnoštvo biblioteka zasebnih namjena, podijelih ćemo ih u nekoliko kategorija.

2.7.1. Korisničko sučelje

Glavna zadaća Reacta i React Native-a je upravo izgradnja korisničkih sučelja, pa ćemo nabrojati neke od najpopularnijih biblioteka za izgradnju istih.

- **Styled Components** – vrlo popularna biblioteka i u Reactu i u React Native-u, takozvano CSS in JS rješenje. Nema već definirane komponente kojima izgrađujemo sučelje, već jednostavno olakšava pisanje standardnog CSS-a. To uvelike olakšava pisanje logike u CSS-u, npr. vraćanjem različitih boja elemenata ovisno o tamnoj ili svijetloj temi sustava (styled-components, 2022).
- **Emotion** – još jedno popularno CSS in JS rješenje, sintakse vrlo sličnoj onoj koja koristi Styled Components. Međutim, paket je manji pa s toga može doprinijeti performansama sustava (Emotion, 2022).
- **Tailwind** - možda i najpopularnija biblioteka za rad s Reactom. Sadrži već napisane CSS klase i komponente, tako da ih korisnih može samo definirati i promijeniti boje

svoje aplikacije i uz minimalno truda izraditi vrlo lijepa korisnička sučelja (Tailwind CSS, 2022).

2.7.2. Forme

Izgradnja formi je jedna od osnovnih i najčešće rađenih vrsti sustava, prema tome već imamo veliki izbor paketa kojima ih možemo izgraditi.

- **Formik** – jedna on naj popularnih biblioteka za izradu formi. Omogućava jednostavnu izgradnju kompleksnih formi, a sadrži sustav za upravljanjem stanja, podnošenja i validacijom stanja.
- **React Hook Form** – biblioteka napravljena specifično za zakačke, pa prema tome iskorištava sve mogućnosti modernog Reacta.
- **React Final Form** – nadograđena na popularnu biblioteku Final Form, napisana i optimizirana za React.

2.7.3. Animacije

Animacije su važan dio svake mobilne aplikacije, pa tako i u React Nativeu imamo nekoliko mogućnosti za postizanje istih:

- **React Native Animated** - Biblioteka koja dolazi uz React Native i nije ju potrebno posebno instalirati pošto je uključena u glavni paket. Kako React Native komunicira sa nativnim kodom preko mosta *(engl. Bridge)***,** a ta komunikacija može biti dosta spora, Animated biblioteka izračuna sve potrebno za animacije prije pokretanja aplikacije, te ih pošalje nativnim dijelovima aplikacije. Tako se minimizira potreba za komunikacijom između *Javascripa* i nativnih dijelova dok aplikacija radi.
- **React Native Reanimated** – Reanimated je nadograđena verzija osnovnog Animated paketa, te tako i nudi više mogućnosti. Sadrži posebne zakačke koji omogućuju komunikaciju između Javascript i nativnog koda te je optimiziran za rad s *React Native-om.*
- **React Native Skia** – Velika biblioteka koja osim animacija sadrži razne mogućnosti crtanja različitih grafika. Najveća prednost Skie je ta što sav svoj kod prevodi u C i C++ jezik koji koriste Android i iOS aplikacije kako bi omogućio najbolje performanse.

2.7.4. Ostale biblioteke

Važno je spomenuti još neke biblioteke koje ne spadaju u kategorije ispred, a vrlo često se koriste u razvoju *React* i *React Native* aplikacija.

- **React Query** – izuzetno popularna biblioteka za rad s web zahtjevima. Omogućava dohvaćanje, spremanje, i sinkronizaciju podataka dohvaćenog sa servera. Vrlo je jednostavna za korištenje i uz pravilnu primjenu može znatno poboljšati performanse aplikacija tako da se izbace nepotrebni web zahtjevi za one podatke koji su već u aplikaciji.
- **Victory** i **Victory Native** – vrlo često u razvoju aplikacija je potrebno generiranje raznih vizualnih podataka kao što su grafovi. Victory je odličan alat za crtanje istih, a sadrži identičan API za *React* i *React Native* aplikacije, s tim da je *Victory Native* optimiziran za mobilne aplikacije.
- **Socket.io** – biblioteka napravljena za rad sa utičnicama (*engl. socket*). Ukoliko je u aplikaciji potreba konstantna komunikacija između klijenta i poslužitelja, a ne samo da poslužitelj odgovara na zahtjeve klijenta, preporučeno je koristiti ovu biblioteku.

3. Redux

Redux je predvidljiv spremnik stanja za Javascript aplikacije. (Redux, 2020)

Nastao je 2015. godine po uzoru na Flux, koji je još jedna biblioteka za upravljanje stanjem. Flux je usko vezan uz React i Metu, i Meta ga je napravila kako bi njoj služio za upravljanjem stanjem aplikacija pisanih u React-u. Redux je izrazito mala biblioteka koja se zapravo može rekreirati u par desetaka linija koda, međutim on donosi ideju spremanja cijelog stanja aplikacije na jedno mjesto kako bi mijenjanje tog stanja bilo što jednostavnije i bezbolnije.

Važno je napomenuti da Redux nije potpuna zamjena za Reactovo stanje sa useState zakačkom, nego ovi žive zajedno i upotpunjuju se. Na Redux možemo gledati kao neko globalno stanje aplikacije, i u Redux nećemo spremati svaku sitnicu. Na primjer, ako imamo element koji nam služi za pretragu neke liste, vrijednost te pretrage nećemo upisivati u Redux-ovo stanje, već nam je dovoljno ono koje dolazi uz React. Međutim, rezultat te pretrage, npr. proizvodi koji sadrže taj upisani tekst, možemo spremiti u Redux.

Također, cijelo stanje zapisano u Reduxu živi samo u memoriji računala, te gašenjem te aplikacije ćemo izgubiti cijelo stanje Redux-a. Tu naravno postoje različita rješenja kao što su biblioteke poput *redux-persist*, koje omogućavaju spremanje tih podataka u neku vrstu trajne memorije.

3.1. Osnovni koncepti

Glavni dijelovi Redux-a su: akcije (*engl. actions*), reduktori (*engl. reducer*) i pohranitelj (*engl. store*). Pohranitelj je namjerno napisan u jednini zbog uobičajenosti aplikacija da sadrže samo jedan pohranitelj, što naravno ne mora biti istina za svaku aplikaciju, međutim akcija i reduktora ćemo sigurno imati nekoliko u svakoj većoj aplikaciji.

3.1.1. Akcije

Započeti ćemo sa akcijama. Akcije su objekti kojima opisujemo što želimo promijeniti u našem stanju i osnovni su način promjene stanja u svakoj aplikaciji koja sadrži Redux. Akcije nisu ništa drugo nego objekt koji mora imati jedan parametar, tip. Najjednostavnija Redux akcija nam može izgledati ovako:

```
{ type: 'ADD_NUMBER'}
```

Vidimo da ta akcija sadrži tip : „Add number“. I dok ovo može raditi tako da kada pozovemo akciju svaki puta povećamo brojač za jedan, puno nam je bolje dodati još jedan parametar u objekt akcije, koji se naziva teret (*engl. payload*). Tada u teret možemo upisati bilo koji broj koji želimo, a brojač će se povećati za onoliko koliko je definirano u teretu. Tada će naša akcija izgledati ovako:

```
{ type: 'ADD_NUMBER', payload: 3}
```

Često u programiranju ne želimo ostavljati takozvane „magične stringove“ kako bi sveli pogreške programera na minimum. Ovaj problem je puno bolje riješen u tvrdo pisanim jezicima, međutim u Javascriptu ćemo se koristiti trikom konstanti. Svaku akciju koju definiramo ćemo zapisati kao konstantu:

```
const ADD_NUMBER = 'ADD_NUMBER'
```

te onda u akciju možemo upisati tu konstantu umjesto stringa:

```
{ type: ADD_NUMBER, payload: 3}
```

Samo to nije dovoljno za korištenje akcija. Moramo napraviti još jednu stvar, a to je definirati i kreatora te akcije. Kreator je funkcija koja kao argument prima teret, ukoliko ga ima, te vraća objekt kojim smo definirali akciju, stoga naziv kreator akcije. Za naš slučaj ćemo definirati jednog kreatora akcije kojeg ćemo nazvati *addNumber.*

```
const addNumber = (payload) => {
      return {
            type: "ADD_NUMBER",
            payload: { payload },
            }
      }
```

3.1.2. Reduktori

Reduktori su funkcije koje kao argument primaju trenutno stanje i akciju, te kao rezultat vraćaju novo stanje nakon promjena uzrokovanih navedenom akcijom. Reduktor sadrži sve

moguće promjene koje su moguće u stanju aplikacije, te kada ga akcija pozove, proći će po svim svojim slučajevima dok ne nađe slučaj koji se naziva isto kao i tip akcije koja ga je pozvala. Već vidimo da ćemo u rješenju koristiti switch grananje. Za sada, naš reduktor će izgledati ovako:

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'ADD_NUMBER':
      return state + action.payload
    default:
      return state
  }
}
```

U gornjem primjeru možemo primijetiti nekoliko ključnih stvari. Prvo, vidimo da se stanje, ukoliko ne postoji postavlja na 0. To znači da kada ćemo prvi puta koristiti reduktor, dok još nismo dirali njegovo stanje, ono biti postavljeno na 0. Tada vidimo da switch grananje traži onaj tip koje je poslan sa akcijom, kako bi znao što treba učiniti. Kako za sada naš reduktor ima samo jedan slučaj, njega nije teško prepoznati. U toj grani također možemo vidjeti što se događa sa tom akcijom, a rezultat koji će reduktor vratiti će biti zbroj starog stanja (ukoliko postoji) i tereta dobivenog s akcijom. Na kraju, ukoliko reduktor ne može pronaći tip akcije, vratiti će isto stanje s kojim je i počeo.

Iako smo definirali reduktor množinom, što znači da ćemo ih napraviti više u aplikaciji, ovisno o domeni aplikacije, svi reduktori će se spojiti u jedan, što Redux radi sam kada kreira pohranitelja.

3.1.3. Pohranitelj

Pohranitelj, kao što mu i samo ime kaže, pohranjuje sve podatke Redux-a u memoriju. Kada dobijemo rezultate promjena pomoću akcija i reduktora, pohranitelj će se pobrinuti da se ti podaci spreme u memoriju. Jedini način za promjenu stanja u pohranitelju je tako da otpremimo (*engl. dispatch*) akciju na njemu. Pohranitelj sadrži nekoliko metoda za upravljanje stanjem, kao što su *getState()* i *dispatch(action) i subscribe(listener).*

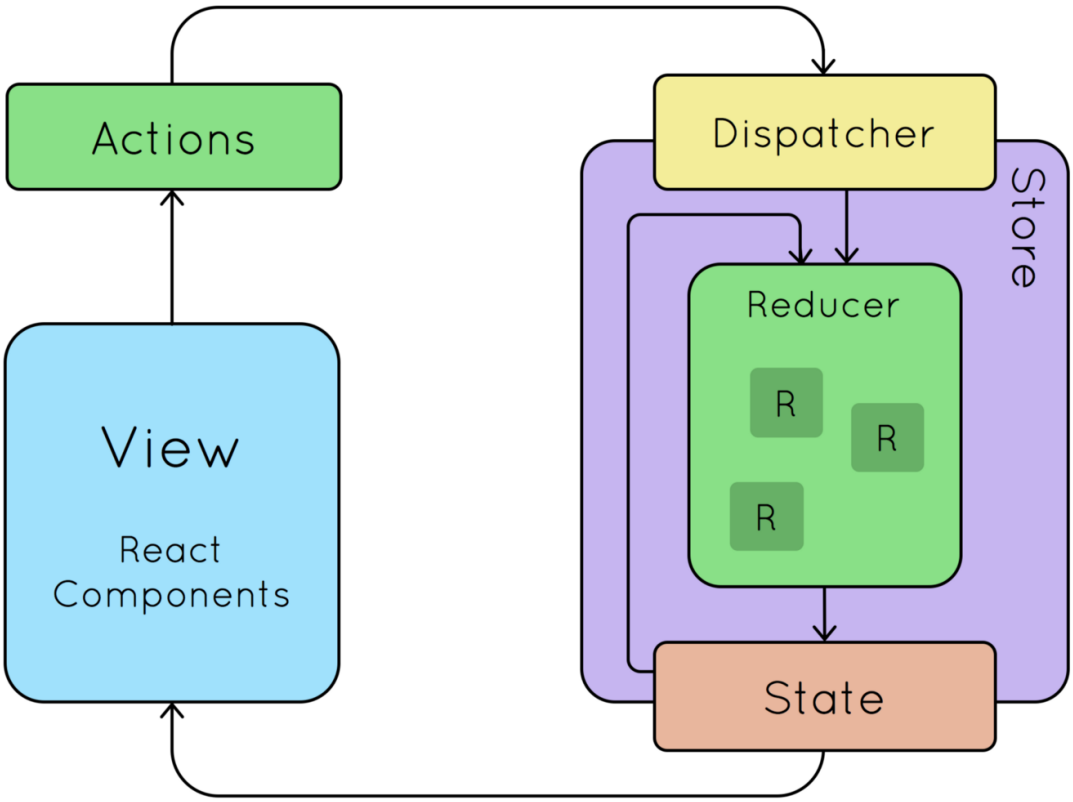
- ***getState*** metoda je najjednostavnija od navedenih. Ona ne prima nikakve parametre, a kao rezultat vraća trenutno stanje, jednako onome koje je bilo vraćeno kao rezultat zadnje akcije nad reduktorom.
- ***dispatch*** metoda kao parametar prima objekt akcije. To je jedini način na koji možemo pokrenuti promjenu u stanju. Kao rezultat on vraća vrijednost koja se smatra novim stanjem, tj. vraća vrijednost koju bi dobili ako pozovemo *getState* metodu nakon promjene.
- ***subscribe(listener)*** metodom dodajemo slušača (*engl. listener*) na naše stanje. Slušač će se pozvati svaki puta kada se pozove otpremitelj, tj. svaki puta kada se stanje naše aplikacije promjeni. Njime obavještavamo React da se stanje promijenilo, i da se komponenta ili cijela stranica moraju ponovo generirati sa novim stanjem.

3.2. Tok podataka

Kada smo se upoznali sa osnovnim dijelovima svake Redux aplikacije, važno je dobro razumjeti i tok podataka koji se događa svaki puta kada otpremimo neku akciju, Na slici ispod možemo detaljno vidjeti kružni tijek događanja u Redux-u. Krenuti ćemo od React komponente koja je obojana plavom bojom.

Svaki događaj koji se dogodi u komponenti može otpremiti neku akciju. Kada reduktor primi tu akciju, proći će kroz sve moguće slučajeve na koje ta akcija može utjecati, i sukladno njoj će promijeniti stanje aplikacije.

Kada izračunamo novo stanje aplikacije, pohranitelj će obavijestiti sve svoje slušače o novonastaloj promjeni, što znači da će se sve komponente koje ovise o tom stanju biti ponovo generirane s novim stanjem. Također, moguće je da pohranitelj sam pozove neke druge reduktore dok se događa promjena, to se dešava kada imamo više međuzavisnih varijabli u stanju.



Slika 2: Veza između komponenti Redux-a i React-a.
(https://miro.medium.com/max/1400/1*gZgzQTPqgS9opZBvjB9tUg.png)

3.3. Redux Toolkit

Kada konfiguriramo Redux za našu aplikaciju, imamo osjećaj da je previše toga potrebno napraviti kako bi ga počeli koristiti. Također, Redux sam po sebi je izrazito mala biblioteka, i za ugodan rad s njim je potrebno je uključiti mnoge druge pakete. Na primjer, Redux sam po sebi ne podržava asinkroni rad, što znači da nećemo moći raditi HTTP zahtjeve unutar akcija Reduxa.

Kao odgovor na ove probleme dolazimo do Redux Toolkit-a. On je napravljen kao rješenje problema koje smo naveli iznad. Znatno pojednostavljuje početnu konfiguraciju Redux aplikacija, olakšava njegovo korištenje i omogućava asinkroni rad. Za početak, pokazati ćemo razlike između definiranja akcija, reduktora i pohranitelja.

Ako se prisjetimo primjera definiranja neke akcije u standardnom Reduxu, on je izgledao ovako:

```
const ADD_NUMBER = 'ADD_NUMBER'

{ type: ADD_NUMBER, payload: 3}
```

Jednostavne akcije ćemo definirati funkcijom ***createAction.*** Ukoliko akcija ne prima teret, nju ćemo definirati na slijedeći način:

```
createAction('ADD_NUMBER')
```

Možemo primijetiti da više nije potrebno prvo definirati ime akcije kao konstantu, te onda tu konstantu koristiti u samom objektu akcije, već Redux Toolkit to odrađuje sam. Također, nije potrebno niti definirati kreatora akcije. Ukoliko želimo da naša akcija prima teret, proširiti ćemo je funkcijom kao drugim parametrom *createAction* funkcije. Tada će akcija izgledati ovako:

```
const addNumber = (payload) => {

    return { payload }

}
```

Ili korištenjem modernog Javascripta:

```
const addNumber = (payload) => ({ payload })
```

Slijedi prikaz pojednostavljene definicije reduktora pomoću Redux Toolkita:

```
createReducer(0, {

  addNumber: (state, action) => state + action.payload,

});
```

Primjećujemo da je reduktor funkcija koja prima dva parametra. Prvi je početno stanje reduktora, a drugo je objekt u kojemu se definirane radnje koje će dogoditi kada se otpremi akcija. Svaka radnja je definirana funkcijom, a ime funkcije mora biti jednako imenu kreatora funkcije. Rada reduktor naiđe na ime funkcije koje je jednako onome koje je otpremljeno, on ju izvrši. Ta funkcija prima 2 parametra, trenutno stanje i objekt akcije koja ga je pozvala, a kao rezultat, u našem slučaju, vraća zbroj stanja i tereta akcije.

Slijedi nam kreator trampolina (*engl. thunk*). On nam izrazito olakšava kreiranje akcija za koje nam je potreban neki asinkroni poziv, bilo to na mrežu, na disk sustava ili bazu podataka. Svaki trampolin ćemo definirati createAsyncThunk funkcijom koja prima dva

parametra, ista ona kao i kreator akcija *createAction.* Prvi parametar je ime trampolina kojega definiramo, a drugi je funkcija u kojoj izvršavamo potrebnu radnju. U slijedećem primjeru možemo vidjeti jedan HTTP poziv:

```
const getUserById = createAsyncThunk("getUserById", async (id) => {
  const response = await fetch(`https://example.com/${id}`);
  const data = await response.json();


  return data;
});
```

U primjeru iznad možemo vidjeti kako je ime akcije *getUserById,* a u funkciji koja se izvršava vidimo da se obavlja mrežni poziv sa id-em koji je dobiven kao teret akcije, odgovor poziva se pretvara u JSON format, i taj podatak se vraća kao rezultat funkcije.

Iako asinkroni trampolini rješavaju problem asinkronog koda u Reduxu, u Redux Toolkit sadrži još moćniji alat, specifičan za HTTP zahtjeve i upite, naziva RTK Query. RTK Query je alat sličan React Query-u, međutim dok React Query kao memoriju koristi kontekst koji dolazi uz React, RTK Query koristi Redux kao spremište podataka. RTK Query kao bazu koristi createApi funkciju, pomoću koje definiramo sve moguće mrežne pozive prema određenom poslužitelju. Ukoliko je u aplikaciji potrebno raditi pozive na više poslužitelja, definirati ćemo ih sa više API-ja.

Jednostavan primjer definiranja poziva nekom poslužitelju izgleda ovako:

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

const api = createApi({
  reducerPath: "api",
  baseQuery: fetchBaseQuery({ baseUrl: "https://example.com/" }),
  endpoints: (builder) => ({
    getUserById: builder.query({
      query: (id) => `user/${id}`,
```

```
    }),
  }),
});

export const { useGetUserByIdQuery } = api
```

Na početku uvozimo potrebne funkcije koje su nam potrebne za početak rada, a to su već spomenuti *createApi* te *fetchBaseQuery*. Funkcijom *createApi* ćemo definirati sve potrebne parametre koji su potrebni. Vidimo iz primjera da ona kao parametar prihvaća objekt koji i sam prima nekoliko različitih parametara. Prvi od njih je *reducerPath,* koji određuje putanju u kojoj će se spremati odgovori poslužitelja. *baseQuery* ćemo postaviti na vrijednost koju nam vraća funkcija *fetchBaseQuery*, a u njoj definiramo glavni URL nad kojim ćemo vršiti HTTP pozive. Na kraju vidimo parametar *endpoints*, u kojem pak definiramo naše pozive. U slučaju GET zahtjeva koristiti ćemo *query* funkciju, koja prima id korisnika o kojemu želimo dohvatiti podatke. Ta funkcija kao rezultat vraća string koji će se dodati na glavni URL na koji vršimo pozive, što znači da će potpuni url izgledati ovako:

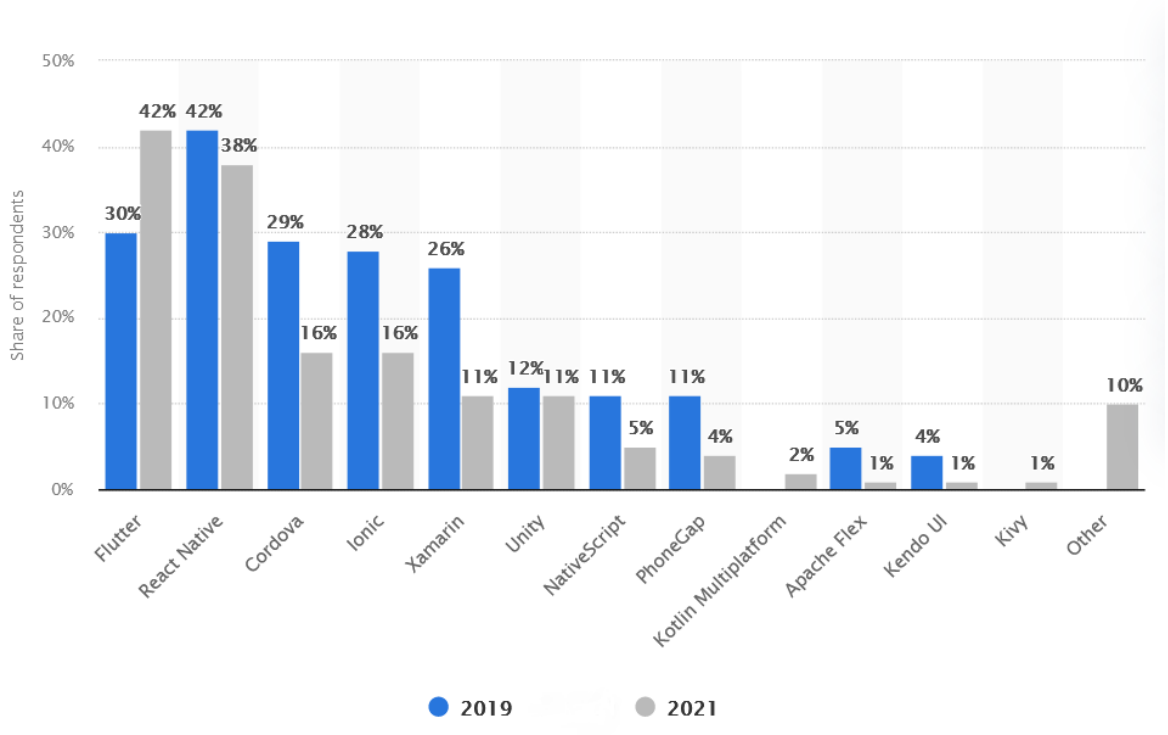
`https://example.com/user/${id}`

A vrijednost id-a se mijenja ovisno o ulaznom parametru.

4. Slične tehnologije

Zadnjih godina postoji sve više različitih rješenja za izradu višeplatformskih aplikacija. Tu naravno, kao i u drugim granama razvoja softvera, ne postoji najbolje rješenje, nego je svako rješenje bolje u jednoj, a gore u drugoj stvari. Najveća prednost React Nativea je lagan prelazak iz svijeta Reacta u svijet React Native-a, jer je kod gotovo isti. Druge poznate tehnologije koje ćemo spomenuti su Ionic i Flutter, koje također imaju respektabilan status u svijetu razvoja višeplatformskih aplikacija. Najvažnija stavka je dakako mogućnost razvoja iOS i Android aplikacija, dok različiti okviri omogućuju razvoj i na drugim platformama kao što su web, AndroidTV, TvOS itd…

Jedan od najboljih pokazatelja uspješnosti nekog razvojnog okvira je broj ljudi koji koriste određeni programski okvir. Tako da vidimo da React Native trenutno drži drugo mjesto po popularnosti, a bio je najpopularniji 2019. godine, kada ga je pretekao Flutter. Također možemo vidjeti da Ionic brzo gubi na popularnosti, što je i očekivano pošto je to jedna od starijih tehnologija.



Slika 3: Popularnost okvira među programerima višeplatformskih aplikacija

(https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours)

4.1. Flutter

Flutter je tehnologija za razvoj višeplatformskih aplikacija koja je nastala u Googlu (Flutter, 2022). Predstavlja izravnog suparnika React Nativeu kada gledamo hibridni razvoj aplikacija. Prva verzija izdana je 2017. godine.

Za razliku od React Nativea, za početak rada sa Flutterom potrebno je naučiti posve novi programski jezik Dart. To može biti velika mana programerima, za razliku od React Nativea, gdje se koristi već poznati Javascript.

Izgled „Hello World“ aplikacije u Flutteru izgleda ovako:

```
import 'package:flutter/material.dart';


void main() {
  runApp(
    const Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

Iako im je sintaksa generiranja elemenata na ekranu slična, velika razlika između React Nativea i Fluttera je u tome što Flutter svaki element na ekranu crta posebno, dok React Native maksimalno iskorištava nativne mogućnosti svakoga elementa. To znači da možemo dobiti različite izglede gumba i drugih elemenata na Android i iOS sustavima, što poboljšava performanse zato što se koriste native komponente.

4.2. Ionic

Ionic je prva poznatija višeplatformska tehnologija. Nastao je još 2013. godine te za razliku od React Nativea i Fluttera, nema veliku tehnološku firmu iza sebe, već tim programera koji održavaju ovaj softverski okvir (Ionic, 2022). U početku je podržavao samo softverski okvir Angular, međutim danas podržava rad sa svim bitnim alatima za klijentsko programiranje, što podrazumijeva React i Vue.

```
<ion-navbar *navbar>

  <ion-title>

    Home

  </ion-title>

</ion-navbar>



<ion-content class="home">

  <ion-card>

    <ion-card-content>

      Hello World

    </ion-card-content>

  </ion-card>

</ion-content>
```

U primjeru iznad vidimo „Hello World“ aplikaciju napisanu u Ionic okviru. Također i kod njega možemo prepoznati gniježđenje elemenata, koje smo već vidjeli u React Nativeu i Flutteru, ideja koja izvorno dolazi iz HTML-a, odnosno razvoja web aplikacija.

5. Aplikacija

U ovome poglavlju slijedi opis aplikacije za planiranje letova dronova. Aplikacija je generirana pomoću React Native CLI.

5.1. Opis ekrana aplikacije

Osnovni problem koji aplikacija rješava je simuliranje letova bespilotnih letjelica. Korisnik aplikacije unosi sve potrebne podatke, na različitim ekranima aplikacije, te kada su svi potrebni podaci uneseni, odabire jedan od definiranih planova te pokreće simulaciju. Svi podaci ostaju spremljeni u trajnoj memoriji uređaja, što znači da ih nije potrebno ponovo unositi ukoliko ugasimo aplikaciju.

Na prvom ekranu korisnik ima mogućnost pregleda spremljenih letjelica. Na dnu ekrana se nalazi gumb „Nova letjelica“, koja dovodi korisnika na novi ekran gdje korisnik upisuje ime letjelice te ju sprema u aplikaciju. Korisnik također na prvom ekranu ima mogućost uređivanja ili brisanja letjelica klikom na jednu od njih.

Na slijedećem ekranu korisnik ima mogućnost dodavanja točki interesa do kojih će letjelice letjeti. Na prvome ekranu su izlistane sve točke interesa. Klikom na gumb „Nova Točka“ moguće je dodati novu točku interesa. Kao potrebni podaci, traži se ime točke, te koordinate, odnosno geografska širina i visina točke, koju korisnik odabire pomoću interaktivne karte koju pogone Google Maps na Android uređajima, odnosno Apple Maps na iOS uređajima. Korisnik također može urediti postojeće točke klikom na jednu od njih ili izbrisati istu.

Na trećem ekranu se nalazi pregled svih planova leta. Klikom na jedan od njih je moguće urediti ili izbrisati plan, a klikom na gumb „Novi plan“ korisnik upisuje podatke za spremanje novoga plana. Kao potrebni podaci traže se ime plana, letjelica koja se odabire iz padajućeg izbornika, te točke do kojih će letjelica letjeti. Točke se u plan dodaju tako da se klikom na njih u izborniku „Sve točke“ prebace u izbornik „Točke plana“. Kada je korisnik zadovoljan unesenim podacima, klikom na gumb „Dodaj“ sprema podatke.

Zadnji ekran aplikacije predstavlja simulaciju leta letjelice. Odabirnom plana iz padajućeg izbornika na karti se pojavljuju točke interesa u obliku markera. Također se pojavljuje ikona letjelice kod početne točke plana. Klikom „Pokreni“ pokrećemo simulaciju tako

da animiramo poziciju ikone letjelice na karti. Letjelica uvijek leti istom brzinom, bez obzira na udaljenost točki, a udaljenost točaka se računa pomoću haversinus formule.

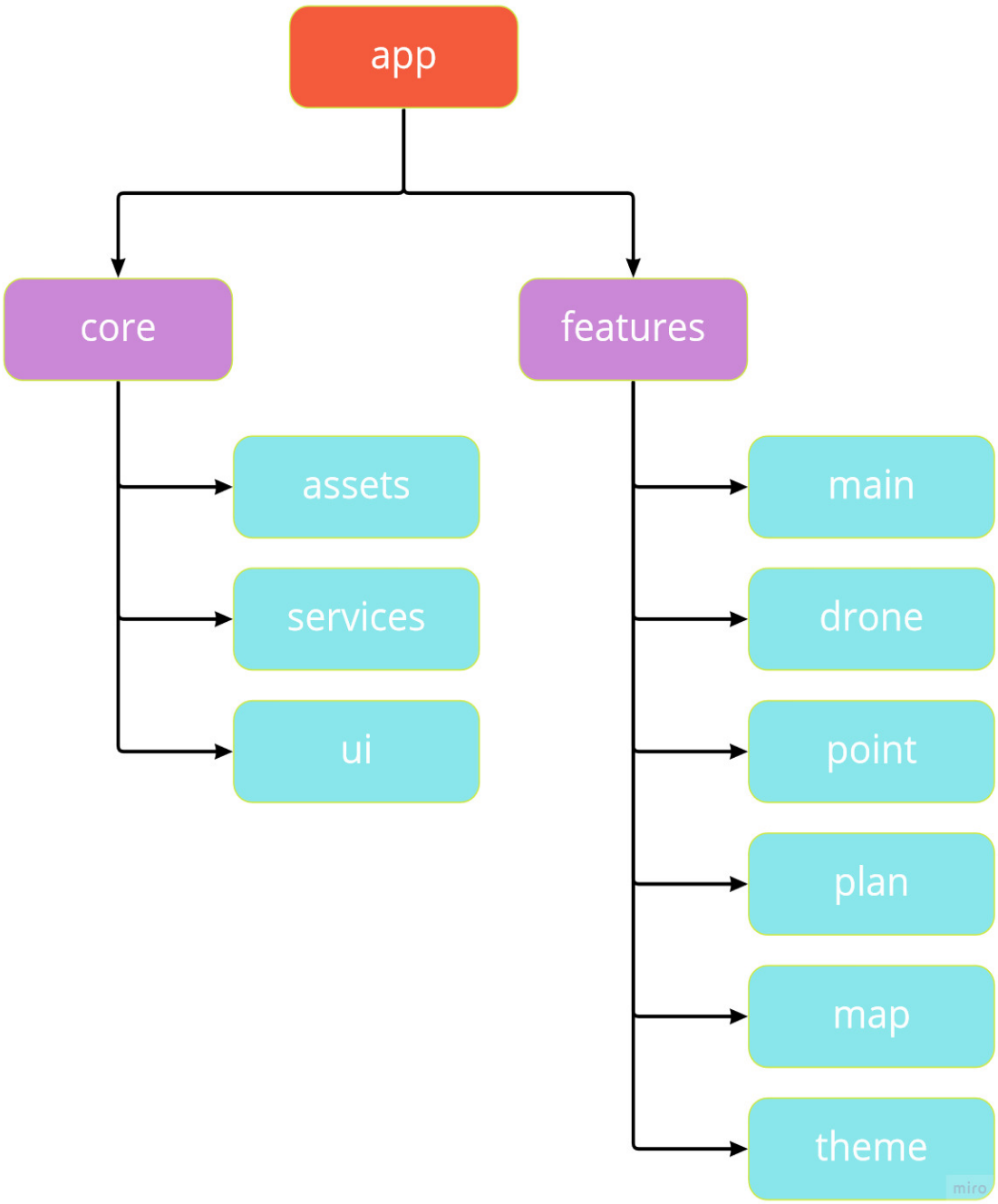
5.2. Arhitektura

Za arhitekturu aplikacije odabrano je pakiranje po značajkama (engl. package by feature). Glavno obilježje ove arhitekture je pakiranje koda i funkcionalnosti aplikacije u male cjeline, koje mogu djelovati same za sebe. S ovom arhitekturom je vrlo lagano uključivati i isključivati određene funkcionalnosti aplikacije, što također omogućava lagano mijenjanje, ili pak dodavanje novih značajki, odnosno funkcionalnosti u aplikaciju. S druge strane, ovaj pristup koristi puno predloženog (*engl. boilerplate*) koda, pošto pišemo više manjih aplikacija koje kasnije slažemo u cjeline.

Glavna podjela naše aplikacije je na temeljne i ostale značajke. Temeljne značajke sadrže funkcionalnosti, glavne komponente te slike, ikone, fontove i slične stvari koje se koriste posvuda u aplikaciji, odnosno u gotovo svim značajkama. U slučaju naše aplikacije, temeljne značajke se dijele na direktorije *assets* koji sadrži ikone, *services* koji sadrži pomoćne funkcije te *ui* koji sadrži komponente kao što su *Button*, *Text*, *TextInput* i *Separator*.

Međutim, ova arhitektura više dolazi do izražaja kod ostalih značajki naše aplikacije. U konkretnom slučaju, imamo 6 značajki koje se redom zovu : *main, drone, map, plan, point i theme.* Svaka od ovih značajki ima definiranu strukturu direktorija, koja najčešće sadrži:

- assets – slike, ikone, fontovi
- fragments – komponente koje sadrže stanje
- components – komponente koje ne sadrže stanje, odnosno samo generiraju komponente ovisno o parametrima koje dobiju.
- hooks – korisnički definirane zakačke
- redux – sve stvari vezane uz redux, odnosno memoriju značajke



Slika 4: Značajke aplikacije

5.3. Popis vanjskih paketa

Kao što je već napomenuto, sami React i React Native nam ne nude puno mogućnosti osim stiliziranja i definiranja komponenti. Ne možemo implementirati čak niti navigaciju aplikacije bez nekog vanjskog paketa. Stoga je važno napomenuti koji paketi se koriste u aplikaciji te im ukratko opisati funkcionalnosti. Sve pakete dodane u projekt možemo vidjeti u *package.json* datoteci koja se nalazi u korijenskom direktoriju projekta.

Ime paketa	Opis
@react-native-async-storage/async-storage	Omogućava spremanje podataka u trajnu memoriju uređaja.
@react-navigation/bottom-tabs	Navigacija u obliku gumba na dnu ekrana aplikacije.
@react-navigation/native	Implementacija react-navigation navigacije za nativni razvoj.
@react-navigation/stack	Navigacija između ekrana pomoću stoga.
@reduxjs/toolkit	Redux Toolkit pojednostavljuje pisanje redux koda.
haversine-distance	Paket za izračun duljine dvije točke na zemlji.
prop-types	Definira tip i zadane vrijednosti parametara komponenti.
react	Glavni React paket.
react-native	Glavni React Native paket.
react-native-gesture-handler	Paket koji omogućava korištenje gesti.

react-native-maps	Implementacija Google odnosno Apple Maps za React Native.
react-native-safe-area-context	Paket koji ograničava komponentama da se prikazuju preko zuba kamere (*engl. notch*) na iOS uređajima
react-native-screens	Paket o kojemu ovisi @react-navigation/native za navigaciju između ekrana.
react-native-select-dropdown	Jednostavan paket za padajuće izbornike
react-native-svg	Omogućava korištenje SVG ikona u aplikaciji.
react-native-uuid	Generira *uuid* identifikatore.
react-redux	Paket za upravljanje stanjem aplikacija.
redux-persist	Paket koji pomoću async-storage sprema redux stanje u trajnu memoriju.

Tablica 1: Opis paketa korištenih u projektu

5.4. Temeljne značajke

Na početku ćemo pokazati temeljenje značajke, odnosno one koje se koriste posvuda u aplikaciji. Fokusirati ćemo se na definiciju komponenti pošto nam je taj dio najbitniji. Krenuti ćemo od definicije komponente *Text:*

```
import React from 'react';

import { Text as NativeText } from 'react-native';

import PropTypes from 'prop-types';


import { colors } from 'features/theme';
```

```
export const Text = ({ children, size, bold, color }) => {
  const style = {
    fontSize: size,
    fontWeight: bold ? 'bold' : 'normal',
    color,
  };

  return <NativeText style={style}>{children}</NativeText>;
};

Text.propTypes = {
  children: PropTypes.node.isRequired,
  size: PropTypes.number,
  bold: PropTypes.bool,
  color: PropTypes.string,
};

Text.defaultProps = {
  size: 16,
  bold: false,
  color: colors.black,
};
```

Vidimo da *Text* komponenta kao parametre prima 4 vrijednosti, koje su *children, size, bold i color*. Children je ključna riječ u Reactu i React Nativeu i označava djecu komponente,

odnosno sve ono što ta komponenta okružuje. Svom JSX-u, ili u ovome slučaju tekstu ćemo pristupiti pomoću toga parametra. Sljedeći parametar je *size*, koji kao broj prima veličinu fonta kojim će tekst biti ispisan. *Bold* je zastavica koja može biti *true* ili *false*, a označava hoće li tekst biti podebljan. Zadnji parametar koji *Text* komponenta prima je *color*, koji označava boju kojom će tekst biti ispisan.

Sljedeće što možemo vidjeti je objekt style kojim definiramo stil teksta pomoću ranije spomenutih parametara. Varijabli *fontSize* se pridružuje vrijednost parametra *size.* Ternarnim operatorom provjeravamo da li je parametar *bold true* ili *false*, te ovisno o njemu vraćamo vrijednost *bold* ili *normal*, dok vrijednosti *color* samo pridružujemo vrijednost koju smo dobili iz parametara.

Na kraju definicije komponente vraćamo komponentu *NativeText* sa pridruženim stilom i djecom, odnosno tekstom. Na početku definicije smo preimenovali *Text* komponentu koja dolazi iz React Native-a u *NativeText*, kako bi svoju komponentu mogli nazvati *Text* i koristiti je posvuda u aplikaciji.

Još možemo primijetiti dvije stvari koje smo definirali u komponenti, a to je tip varijabli koji se očekuje za svaki parametar i ukoliko parametar nije obavezan, njegovu zadanu vrijednost. Tako je tip *children* definiran kao *node*, *size* kao *number*, *bold* kao *bool* itd.

Sljedeća komponenta koju ćemo definirati je *Button.* Struktura definicije je slična kao i kod *Text* komponente, međutim *Button* prima druge parametre i vraća drugu komponentu.

```
import React from 'react';

import { Pressable, StyleSheet, Text } from 'react-native';

import PropTypes from 'prop-types';


import { colors } from 'features/theme';


const Button = ({ title, onPress, disabled }) => (

  <Pressable

    style={[styles.button, disabled && styles.disabled]}
```

```
    onPress={onPress}
    disabled={disabled}
  >
    <Text style={styles.text}>{title}</Text>
  </Pressable>
);

Button.propTypes = {
  title: PropTypes.string.isRequired,
  onPress: PropTypes.func.isRequired,
  disabled: PropTypes.bool,
};

Button.defaultProps = {
  disabled: false,
};

const styles = StyleSheet.create({
  button: {
    backgroundColor: colors.lightGray,
    borderRadius: 8,
    borderWidth: 1,
    borderColor: colors.gray,
    paddingVertical: 8,
```

```
  },
  text: {
    fontSize: 22,
    textAlign: 'center',
    fontWeight: 'bold',
    color: colors.white,
  },
  disabled: {
    backgroundColor: colors.gray,
    opacity: 0.5,
  },
});

export { Button };
```

Button prima 3 parametra, *title* koji predstavlja tekst koji će se ispisati na gumbu, *onPress* funkciju koja će se pokrenuti pritiskom na gumb, te zastavicu *isDisabled* koja označava može li se gumb pritisnuti ili ne. Ovisno o tim parametrima opet definiramo stilove koje ćemo pridružiti gumbu, pa će tako gumb promijeniti pozadinsku boju i prozirnost ovisno o zastavici *isDisabled.* Kao i na primjeru *Text* komponente, definirani su tipovi parametara koje komponenta očekuje, kao i njihove zadane vrijednosti.

5.5. Ostale značajke

5.5.1. Drone

Drone značajka definira funkcionalnosti i ponašanje svih stvari vezanih uz letjelice. U njoj je sadržan sav kod koji se tiče letjelica, kao što je pregled i brisanje postojećih ili dodavanje novih. Sadrži dva ekrana, *Drones* koji prikazuje listu letjelica, te *NewDrone* pomoću kojeg možemo dodavati, brisati ili uređivati letjelice. Započeti ćemo sa *Drones* komponentom.

```
import React from 'react';
import { FlatList, Pressable, StyleSheet, View } from 'react-native';
import { useSelector } from 'react-redux';
import { useNavigation } from '@react-navigation/native';

import { Button, Separator, Text } from 'core/ui';

import { DroneSelectors } from '../redux';

export const DronesScreen = () => {
  const navigation = useNavigation();
  const drones = useSelector(DroneSelectors.getAll);

  const handleItemPress = (item) => {
    navigation.navigate('Nova letjelica', { id: item.id, title: item.name
});
  };

  const renderItem = ({ item }) => (
    <Pressable style={styles.itemWrapper} onPress={() =>
handleItemPress(item)}>
      <Text bold align>
        {item.name}
      </Text>
```

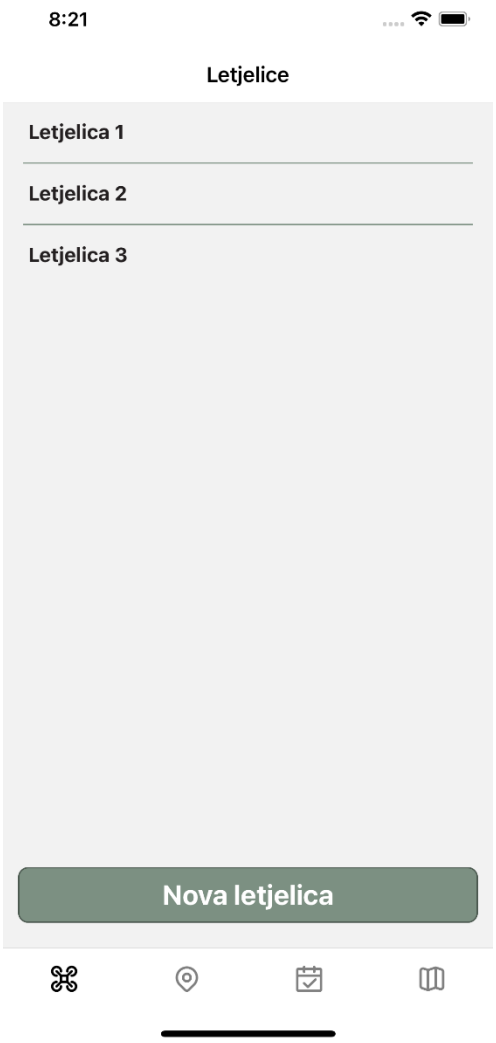
```
    </Pressable>
  );

  return (
    <View style={styles.wrapper}>
      <FlatList
        data={drones}
        renderItem={renderItem}
        keyExtractor={(item) => item.id}
        ItemSeparatorComponent={Separator}
      />
      <Button
        title="Nova letjelica"
        onPress={() => {
          navigation.navigate('Nova letjelica');
        }}
      />
    </View>
  );
};

const styles = StyleSheet.create({
  itemWrapper: {
    margin: 8,
```

```
    height: 32,
    justifyContent: 'center',
  },
  wrapper: {
    flex: 1,
    paddingHorizontal: 12,
    paddingBottom: 20,
  },
  separator: {
    height: 1,
    backgroundColor: 'black',
  },
});
```

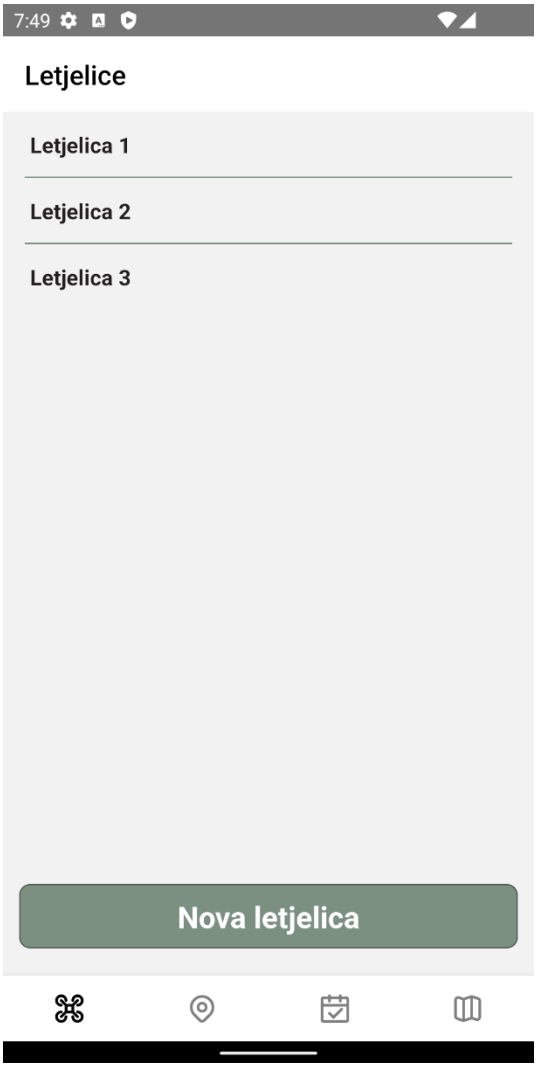
Iznad vidimo kompletni kod komponente *Drones.* Na početku uvozimo sve pakete koje se koriste u komponenti. Iz paketa *react-redux* i *@react-navigation/native* uvozimo funkcije *useSelector* i *useNavigation,* koje su zapravo zakačke pošto sadrže *use* u imenu.

Slijedi definicija same *Drones* komponente. Komponenta sadrži 2 pomoćne funkcije, *handleItemPress* koja prebacuje korisnika na ekran „Nova Letjelica“, uz objekt koji sadrži podatke i letjelici na koju je korisnik kliknuo i *renderItem* koji definira izgled svakog retka unutar *FlatList* komponente. *FlatList* nam dolazi iz glavnog paketa React Nativea i omogućava prikaz listi. Tu listu ćemo definirati pomoću *data* parametra, a lista je dobivena pomoću *DroneSelectors.getAll* redux selektora. Parametar *renderItem* sadrži izgled komponente koja će se pridružiti svakom elementu liste. U našem primjeru *FlatList* komponenta sadrži još 2 parametra, *keyExtractor* koji služi za optimizaciju, te *ItemSeparatorComponent* koji služi za definiciju komponente koja će se prikazati između elemenata liste. Na kraju ekrana možemo vidjeti gumb koji vodi korisnika na novi ekran za upisivanje nove letjelice, ovaj put bez dodatnih parametara. Na slikama 5 i 6, prikazanim ispod, možemo vidjeti izgled komponente *Drones* na iOS i Android uređaju. Kako React Native koristi nativne komponente kada god može, vidimo

da izgled ekrana nije u potpunosti jednak. Naslov trenutnog ekrana je centriran na iOS uređaju, dok je on lijevo poravnat na Android uređaju.



Slika 5: Ekran Letjelice na iOS uređaju



Slika 6: Ekran Letjelice na Android uređaju

Slijedi opis komponente *NewDrone.* Već je navedeno iznad da na ovaj ekran možemo doći na 2 načina: klikom na gumb „Nova letjelica“ ili klikom na jednu od postojećih letjelica u listi, a ovisno o tome na koji smo način došli na ekran, izgled će biti drugačiji. Ukoliko smo kliknuli na gumb „Nova letjelica“, forma neće biti već ispunjena imenom letjelice, na gumbu na dnu će pisati „Dodaj“ umjesto „Ažuriraj“, te gumb za brisanje u gornjem desnom uglu neće biti prikazan.

```
import React, { useCallback, useEffect, useState } from 'react';
```

```
import { Pressable, StyleSheet, View } from 'react-native';
import { useDispatch } from 'react-redux';
import { useNavigation } from '@react-navigation/native';

import { DeleteIcon } from 'core/assets';
import { generateId } from 'core/services';
import { Button, TextInput } from 'core/ui';

import { DroneActions } from '../redux';

export const NewDrone = ({ route }) => {
  const navigation = useNavigation();
  const dispatch = useDispatch();
  const { id, title } = route.params || {};

  const [name, setName] = useState(id ? title : '');
  const newId = generateId();

  const handleDelete = useCallback(() => {
    dispatch(DroneActions.remove(id));
    return navigation.goBack();
  }, [dispatch, id, navigation]);

  const RightIcon = useCallback(
```

```
  () => (
    <Pressable onPress={handleDelete}>
      <DeleteIcon width={24} height={24} style={styles.icon} />
    </Pressable>
  ),
  [handleDelete],
);

useEffect(() => {
  if (title) {
    navigation.setOptions({
      title,
      headerRight: RightIcon,
    });
  }
}, [title, navigation, RightIcon]);

const handleTextChange = (text) => {
  setName(text);
};

const handleNewPress = () => {
  dispatch(DroneActions.add({ id: newId, name }));
  return navigation.goBack();
```

```
  };

  const handleUpdatePress = () => {
    dispatch(DroneActions.update({ id, name }));
    return navigation.goBack();
  };

  return (
    <View style={styles.wrapper}>
      <TextInput onChangeText={handleTextChange} value={name}
placeholder="Ime" />
      {id ? (
        <Button title="Azuriraj" onPress={handleUpdatePress} />
      ) : (
        <Button title="Dodaj" onPress={handleNewPress} />
      )}
    </View>
  );
};

const styles = StyleSheet.create({
  wrapper: {
    flex: 1,
    padding: 20,
    justifyContent: 'space-between',
```

```
  },
  icon: {
    marginRight: 12,
    color: 'black',
  },
});
```

Iako u JSX dijelu imamo svega par linija koda, puno toga se dešava prije njega. Kao parametar komponente primamo *route,* koji sadrži objekt letjelice koji smo poslali sa prošlog ekrana pomoću *navigation.navigate* funkcije. Ovisno o njemu, postavljamo ime letjelice na ime koje smo dobili iz *route* objekta ili prazan string. Malo kasnije u kodu vidimo definiciju RightIcon komponente, koja vraća ikonu smeća. Klikom na tu ikonu brišemo trenutnu letjelicu pomoću akcije *DroneActions.remove(id),* a ikonu prikazujemo ovisno o imenu letjelice, tj. ako smo ime dobili iz *route* objekta onda je prikazana, u protivnome ne. Također vrijednost *TextInputa* postavljamo samo ako smo dobili ime iz već spomenutog objekta. Ovisno na koji gumb kliknemo, izvršiti će se različite akcije. Klikom na „Dodaj“ gumb, izvršiti će se akcija *DroneActions.add*, odnosno *DroneActions.update* klikom na „Ažuriraj“ gumb. Kada smo već spomenuli sve akcije u drone značajki, vrijedi i prikazati reduktor iste:

```
import { createReducer } from '@reduxjs/toolkit';

import { DroneActions } from './actions';

const INITIAL_STATE = [];

export const reducer = createReducer(INITIAL_STATE, (builder) => {
  builder.addCase(DroneActions.add, (state, { payload }) => [...state,
payload]);
  builder.addCase(DroneActions.update, (state, { payload }) => {
    const index = state.findIndex((d) => d.id === payload.id);
```

```
    const newState = [...state];
    newState[index] = payload;

    return newState;
  });
  builder.addCase(DroneActions.remove, (state, { payload }) =>
    state.filter((d) => d.id !== payload),
  );
});
```

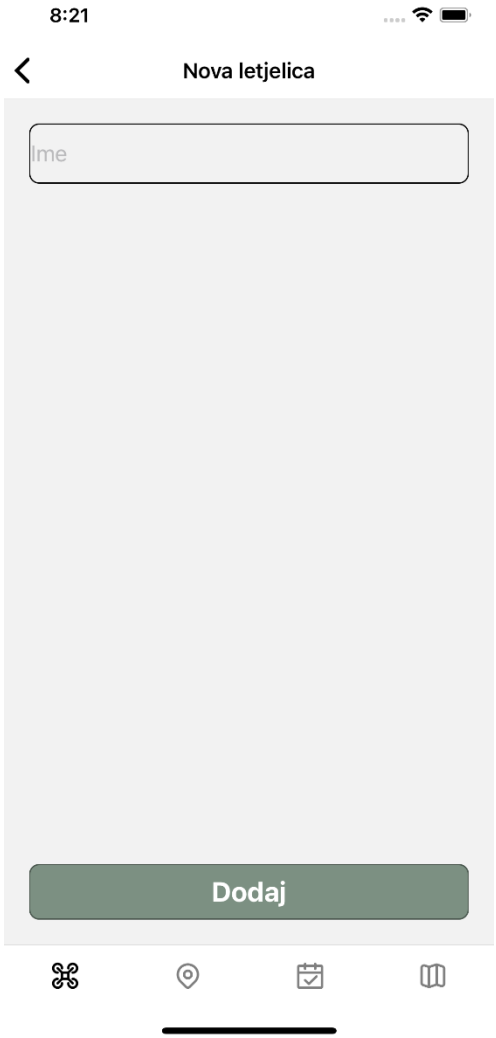
Reduktor sadrži 3 akcije. Akcijom *add* dodaje objekt s podacima letjelica na kraj liste. Akcijom *update* pronalazi letjelicu pomoću identifikatora u listi, te ju ažurira novim podacima, dok akcijom *remove* briše letjelicu iz liste.

Još ćemo prikazati 2 selektora koja su sadržana u drone značajki:

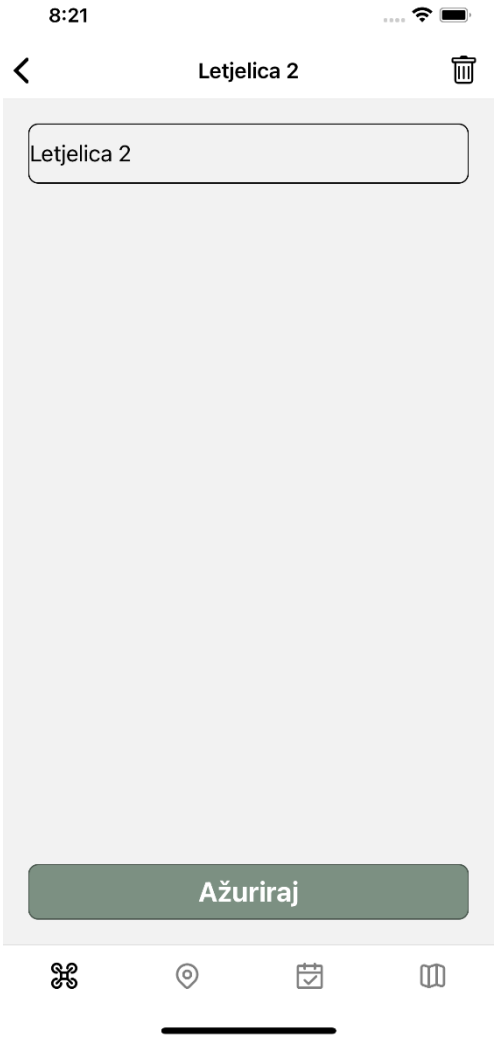
```
const rootSelector = (state) => state.drones;

const getAll = (state) => rootSelector(state);
const getById = (state) => (id) => rootSelector(state).find((d) => d.id ===
id);

export const DroneSelectors = {
  getAll,
  getById,
};
```

Na početku datoteke označavamo korijenski selektor, koji je u našem slučaju *drones*. Slijedi selektor *getAll*, koji vraća sve letjelice u obliku liste, te *getById*, koji pronalazi u listi letjelicu na danim identifikatorom te ju vraća kao rezultat.

Slijedi prikaz ekrana nove letjelice i ekrana uređivanja letjelice.



Slika 7: Nova letjelica



Slika 8: Uređivanje letjelice

5.5.2. Point

Slijedi opis point značajke. Kako je ova značajka vrlo slična drone značajki, nećemo je previše opisivati niti prikazivati kod. Prikazati ćemo samo dio komponente *NewPoint*, kako bi prikazali kartu i odabir točke interesa. Kod NewPoint komponente:

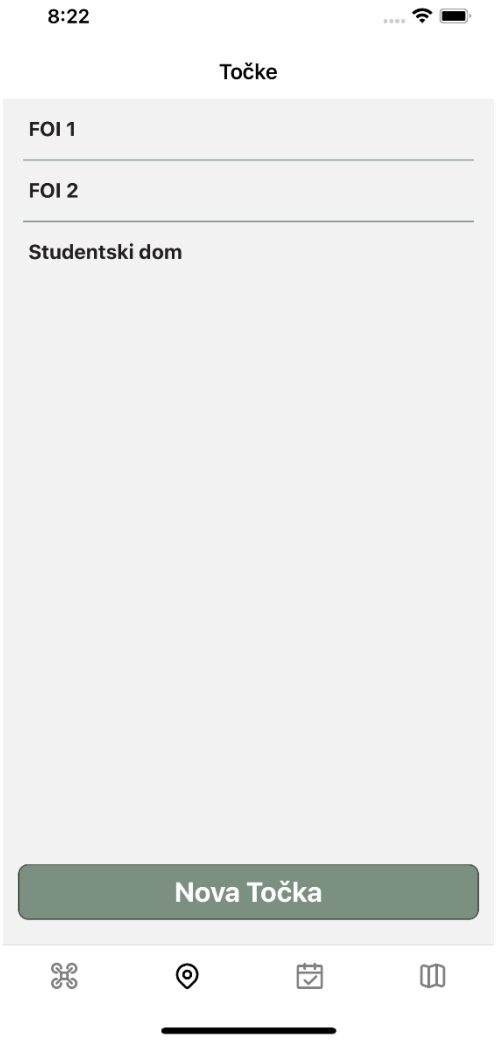
```
const handleMapPress = (e) => {

  setMarkerPosition(e.nativeEvent.coordinate);
```

```
  };


return (
    <View style={styles.wrapper}>
      <TextInput onChangeText={handleTextChange} value={name}
placeholder="Ime" />
      <MapView
        onPress={handleMapPress}
        style={styles.map}
        initialRegion={{
          latitude: 45.5911677,
          longitude: 16.2285311,
          latitudeDelta: 0.0922,
          longitudeDelta: 0.0421,
        }}
      >
        {markerPosition && <Marker coordinate={markerPosition} />}
      </MapView>
      {id ? (
        <View>
          <Button title="Ažuriraj" onPress={handleUpdatePress} />
        </View>
      ) : (
        <Button title="Dodaj" onPress={handleNewPress}
disabled={isDisabled} />
```

```
      )}
    </View>
  );
};
```

Najvažniji dio ove komponente je *MapView* komponenta koja dolazi iz paketa *react-native-maps*, koja prima parametre *style*, *initialRegion* i *onPress.* Ovisno gdje kliknemo na mapu, *onPress* funkcijom dobijemo događaj (*engl. event*) koji, između ostaloga, sadrži i koordinate točke, koje tada spremamo u stanje. Klikom na kartu se na njoj također prikazuje marker, koji izgleda drugačije ovisno o platformi, tj, ovisi o tome koristimo li Google ili Apple maps.

Slika 7: Nova točka



Slika 8: Točke

5.5.3. Plan

Značajka plan je nešto kompleksnija od dvije već opisane značajke. Ona također sadrži dvije komponente, *Plan* i *NewPlan.* Plan komponenta je gotovo ista kao komponente *Drone* i *Point*, ona sadrži listu upisanih planova, te sadrži mogućnost dodavanja novih te uređivanje postojećih planova. Međutim, komponenta *NewPlan* je nešto drugačija. Ona se sastoji od 4 dijela, gdje u prvom dijelu upisujemo ime plana, u drugom iz padajućeg izbornika izabiremo

letjelicu koja će letjeti na tom planu, te 2 liste, jedna u kojoj su sadržane sve točke interesa, a u drugoj su sadržane točke do kojih letjelica leti u trenutnom planu.

```
export const NewPlan = ({ route }) => {
  const navigation = useNavigation();
  const dispatch = useDispatch();
  const { id, name: planTitle, selectedPoints: sPoints, drone } =
route.params || {};

  const [name, setName] = useState(id ? planTitle : '');
  const [selectedPoints, setSelectedPoints] = useState(id ? sPoints : []);
  const drones = useSelector(DroneSelectors.getAll);

  const [selectedDrone, setSelectedDrone] = useState(drone ?? null);
  const points = useSelector(PointSelectors.getAll);

  const handleDelete = useCallback(() => {
    dispatch(PlanActions.remove(id));
    return navigation.goBack();
  }, [dispatch, id, navigation]);

  const RightIcon = useCallback(
    () => (
      <Pressable onPress={handleDelete}>
        <DeleteIcon width={24} height={24} style={styles.icon} />
      </Pressable>
```

```
    ),
    [handleDelete],
  );

  useEffect(() => {
    if (planTitle) {
      navigation.setOptions({
        title: planTitle,
        headerRight: RightIcon,
      });
    }
  }, [planTitle, navigation, RightIcon]);

  const handleTextChange = (text) => {
    setName(text);
  };

  const handleNewPress = () => {
    const newId = generateId();

    dispatch(PlanActions.add({ id: newId, name, selectedPoints, drone:
selectedDrone }));
    return navigation.goBack();
  };
```

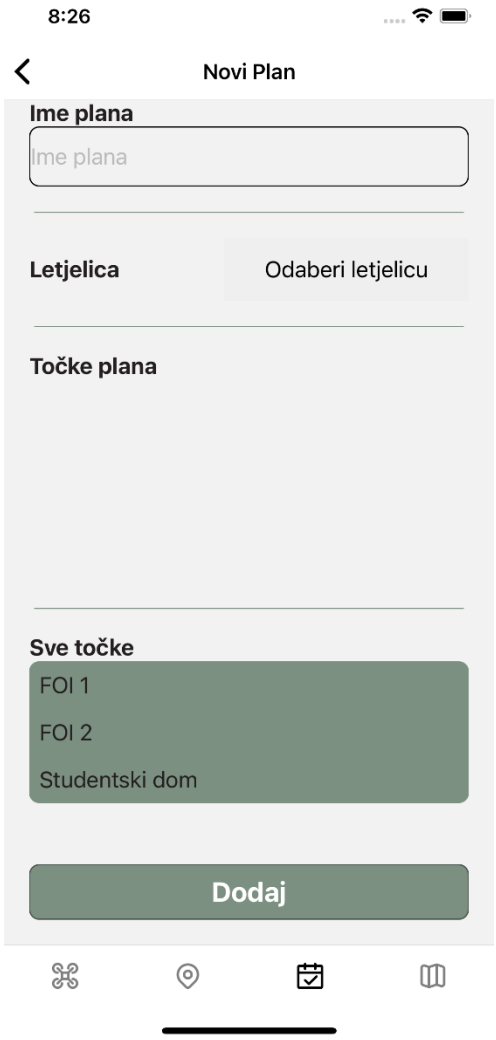
```
  const handleUpdatePress = () => {
    dispatch(PlanActions.update({ id, name, selectedPoints, drone:
selectedDrone }));
    return navigation.goBack();
  };

  const renderPointItem = ({ item }) => (
    <Pressable
      onPress={() => {
        setSelectedPoints([...selectedPoints, item]);
      }}
      style={styles.buttonPadding}
    >
      <Text size={18}>{item.name}</Text>
    </Pressable>
  );

  const renderSelectedPointItem = ({ item }) => (
    <Pressable
      onPress={() => {
        setSelectedPoints(selectedPoints.filter((d) => d.id !== item.id));
      }}
      style={styles.buttonPadding}
    >
      <Text size={18}>{item.name}</Text>
```

```
    </Pressable>
  );

  return (
    <View style={styles.wrapper}>
      <View style={styles.contentWrapper}>
        <Text bold size={18}>
          Ime plana
        </Text>
        <TextInput onChangeText={handleTextChange} value={name}
placeholder="Ime plana" />
        <Separator style={styles.separator} />
        <View style={styles.selectWrapper}>
          <Text bold size={18}>
            Letjelica
          </Text>
          <SelectDropdown
            data={drones}
            defaultButtonText="Odaberi letjelicu"
            onSelect={(selectedItem) => {
              setSelectedDrone(selectedItem);
            }}
            rowTextForSelection={(item) => item.name}
            buttonTextAfterSelection={(item) => item.name}
            defaultValue={selectedDrone}
```

```
      />
    </View>
    <Separator style={styles.separator} />
    <Text bold size={18}>
      Točke plana
    </Text>
    <FlatList
      style={styles.flex}
      data={selectedPoints}
      renderItem={renderSelectedPointItem}
      contentContainerStyle={styles.flatListContainerStyle}
    />
    <Separator style={styles.separator} />
    <Text bold size={18}>
      Sve točke
    </Text>
    <FlatList
      style={styles.flex}
      data={points}
      renderItem={renderPointItem}
      contentContainerStyle={styles.flatListContainerStyle}
    />
  </View>
  {id ? (
```

```
        <View>
          <Button title="Azuriraj" onPress={handleUpdatePress} />
        </View>
      ) : (
        <Button title="Dodaj" onPress={handleNewPress} />
      )}
    </View>
  );
};
```

Na početku komponente provjeravamo kako smo došli do *NewPlan* ekrana, tj. da li *route* objekt sadrži neke podatke. Ovisno o njima postavljamo već upisane vrijednosti na njima određena mjesta. Također dohvaćamo pomoću selektora sve letjelice i točka, kako bi ih mogli postaviti u padajući izbornik, odnosno listu. Sami JSX-u komponente sadrži *TextInput* u koji se upisuje ime plana, *SelectDropdown* komponenta pomoću koje se odabire letjelica, te 2 *FlatList* komponente koje sadrže točke. Jedna sadrži sve točke, dok druga sadrži samo točke u trenutnom planu. Ova komponenta je prikazana na slici ispod.



Slika 9: Novi plan

Slika 10: Uređivanje plana

5.5.4. Map

Map je najvažnija značajka aplikacije. U njoj simuliramo određeni plan pomoću animacija i karte. Kada korisnik definira plan, na ekranu *MapScreen* iz padajućeg izbornika bira koji plan želi simulirati. Kada je plan odabran, pritiskom na gumb „Pokreni“ pokreće simulaciju tako da se ikona drona animira po karti i putuje od točke do točke dok nije došao do svojeg odredišta. Prvo ćemo opisati komponentu Map, a poslije nje i komponentu *MapScreen*.

Na početku komponente uvozimo pakete i funkcije koje komponenta koristi. Također definiramo brzinu letjelice, koja je postavljenja na 200 km/h. Posebno je važna funkcija *startAnimation* gdje je zapravo definirana animacija letjelice. U njoj prolazimo kroz sve točke odabranog plana te računamo prvo udaljenost točaka pomoću haversinus formule, a kada imamo brzinu i udaljenost, lako nam je izračunati i vrijeme puta letjelice između točaka. Kada smo izračunali sve udaljenosti i vremena puta svih točaka, pokrećemo animaciju.

JSX dio nam sadrži *MapView* komponentu na kojoj označavamo sve točke interesa, te također prikazujemo i ikonu letjelice, koja se animira. Na kraju komponente je prikazan gumb „Pokreni“ kojim pokrećemo simulaciju.

```
import React, { useEffect, useState } from 'react';

import { Animated, SafeAreaView, StyleSheet, View } from 'react-native';

import MapView, { AnimatedRegion, Marker } from 'react-native-maps';

import haversine from 'haversine-distance';

import PropTypes from 'prop-types';


import { Button } from 'core/ui';


import { DroneIcon } from '../assets';

import { calculateTime } from '../services';


const droneSpeed = 200 * 0.277777778; // 200 km/h to m/s


export const Map = ({ points, isAnimated }) => {

  const [animRegion, setAnimRegion] = useState(new
AnimatedRegion(points[0]));


  const markerPoints = points.map(({ latitude, longitude }) => (
```

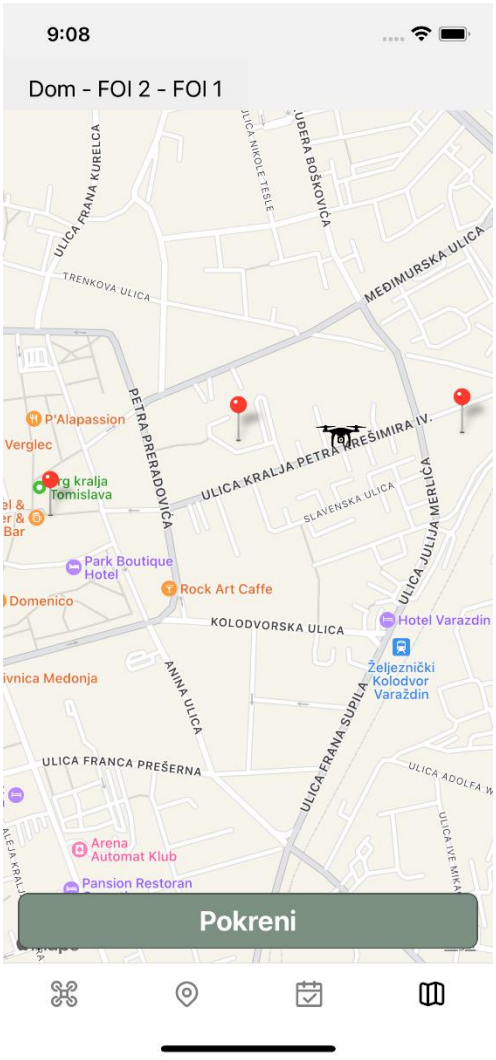
```
    <Marker key={latitude + longitude} coordinate={{ latitude, longitude }}
/>
  ));

  useEffect(() => {
    setAnimRegion(new AnimatedRegion(points[0]));
  }, [points]);

  const startAnimation = () => {
    const data = points.map((nextPoint, index) => {
      if (!points[index + 1]) {
        return null;
      }
      const distance = haversine(nextPoint, points[index + 1]);

      const duration = calculateTime(distance, droneSpeed);

      return {
        ...points[index + 1],
        duration,
      };
    });

    const animatedList = data.filter((d) => d).map((d) =>
animRegion.timing(d));
```

```
  Animated.sequence(animatedList).start();
};

return (
  <View style={styles.wrapper}>
    <SafeAreaView style={styles.container}>
      <MapView
        style={styles.map}
        initialRegion={{
          latitude: 45.5911677,
          longitude: 16.2285311,
          latitudeDelta: 0.0922,
          longitudeDelta: 0.0421,
        }}
      >
        {markerPoints}
        <Marker.Animated coordinate={animRegion}>
          <DroneIcon style={styles.droneColor} width={40} height={40} />
        </Marker.Animated>
      </MapView>
    </SafeAreaView>
    {isAnimated && (
      <View style={styles.buttonWrapper}>
```

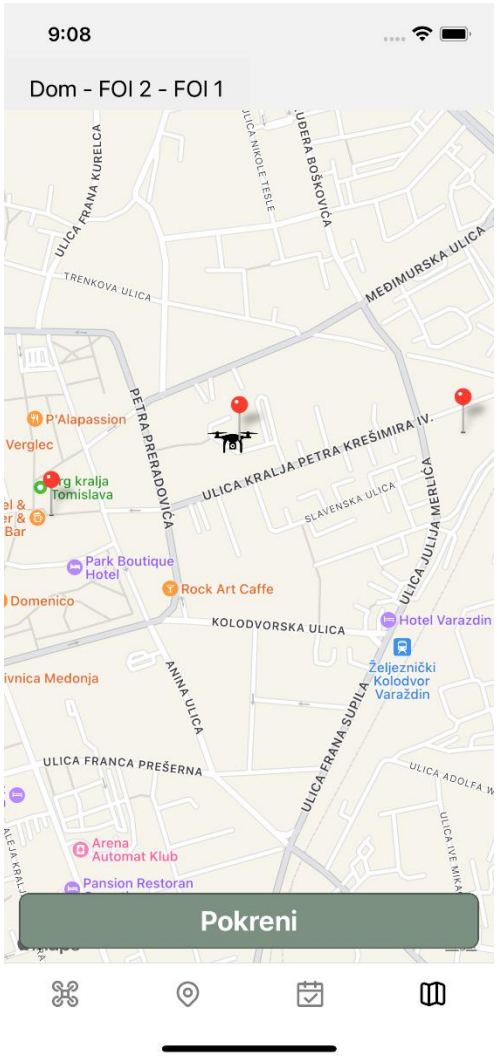
```
          <Button title="Pokreni" onPress={startAnimation} />
        </View>
      )}
    </View>
  );
};
```

Slijedi prikaz 4 slike koji dočaravaju put letjelice. Ime plana vidimo u gornjem dijelu ekrana, naziva Dom – FOI 2 – FOI 1, a na kartama vidimo letjelicu na različitim pozicijama na karti.

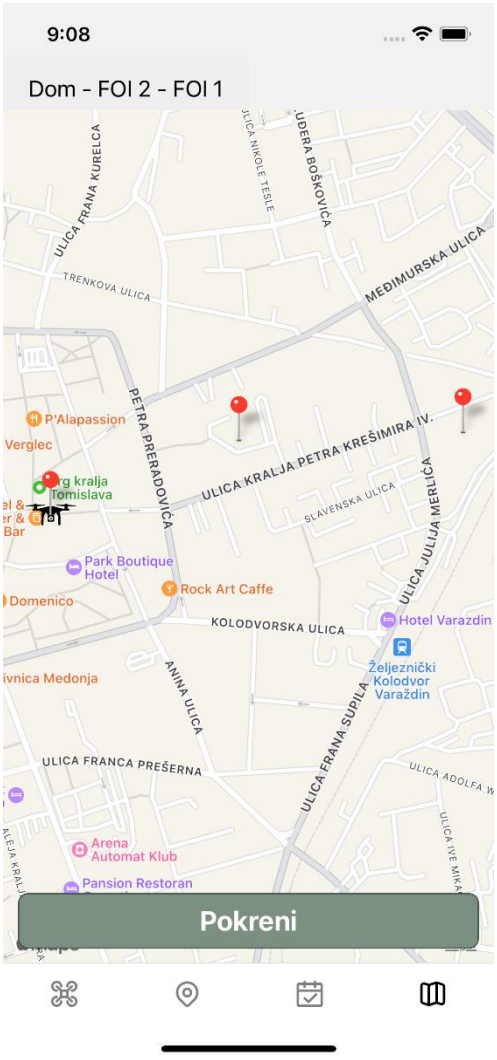
Slika 11: Početni položaj letjelice



Slika 12: Položaj letjelice između Doma i FOI 2



Slika 13: Položaj letjelice između FOI 2 i FOI 1



Slika 14: Završni položaj letjelice

5.5.5. Main i theme

Main značajka aplikacija služi za spajanje svih ostalih značajki u jednu cjelinu. Ona ne sadrži nikakve funkcionalnosti osim implementacije navigacije kako bi mogli doći do svih ekrana aplikacije. Kod nije previše interesantan, te ga nećemo ni prikazati, međutim valja spomenuti da nam paket *@react-navigation/bottom-tabs* služi kao glavna navigacija koju vidimo na dnu svakog ekrana, dok paket *@react-navigation/stack* omogućava navigaciju između ekrana pregleda i kreiranja letjelica, točaka i planova. *Main* značajka također ne prikazuje nikakve dodatne ekrane, svi ekrani su već prikazani u prijašnjim značajkama.

Na kraju nam ostaje još *theme* značajka, koja definira temu aplikacije. U većim aplikacijama ona može sadržavati mnoštvo parametara kao što su veličine margina, veličine komponenti, sjena itd. Međutim u našem slučaju definirane su samo boje. Boje su definirane kao jednostavan objekt sa imenom i heksadecimalnom vrijednošću boje.

Izgled colors.js direktorija je slijedeći:

```
export const colors = {

  white: '#FCFCFC',

  black: '#231F20',

  lightGray: '#7C9082',

  gray: '#4B584F',

};
```

6. Zaključak

Zadnjih godina sve je popularniji višeplatformski razvoj aplikacija. Gotovo sva poduzeća koja žele izraditi mobilnu aplikaciju odlučuju se za razvoj i za Android i za iOS sustav. Tu činjenicu podržava i trenutno stanje na tržištu gdje je Android i iOS uređaji pokrivaju daleko najveći dio tržišta. Kada se odluče za razvoj, moraju se odlučiti hoće li svaku verziju raditi nativno ili će koristiti jedan od višeplatformskih softverskih okvira.

Pošto najčešće žele da ta aplikacija izgleda vizualno jednako na oba sustava, često se okreću ili React Native-u ili Flutteru. Toj činjenici pridonosi kraće vrijeme izrade i manja cijena razvoja pošto je za razvoj potreban jedan tim koji pokriva i Android i iOS uređaje, te istu aplikaciju nije potrebno razvijati dva puta.

S druge strane, razvoj na nekima od višeplaformskih alata krije i neke nedostatke. Otklanjanje pogrešaka je puno teže na React Native-u nego na nativnim alatima, međutim i tu se vide značajni pomaci. Uvelike pomaže i razvoj pomoću Redux-a, koji omogućava pregled cijele memorije aplikacije u svakome trenutku.

Ako ste izabrali višeplatformski razvoj, još slijedi pitanje koji okvir koristiti. Najviše se nameću React Native, koji je opisan u ovome radu, te Flutter, koji sustiže React Native po popularnosti ovih godina. Velika prednost React Native-a leži u tome što znanje React-a uvelike pomaže u razvoju React Native aplikacija, te se programeri bez većih problema mogu prebaciti s jedne tehnologije na drugu, dok je za Flutter potrebno naučiti novi jezik. Međutim, odlučili se ili na Ionic, Flutter ili React Native, sigurno nećete pogriješiti jer oni omogućavaju brzi razvoj za više platformi, te im je budućnost svijetla.

Iz osobnog iskustva React Native okvir je najugodniji za rad. Podržava i ažurira ga *Meta*, koristi se u velikom broju aplikacija, postojeći web programeri bez problema mogu naučiti raditi u React Native-u te zadnjih godina drži respektabilnu poziciju na tržištu. Naravno, ima i nekih nedostatka. Tehnologija je relativno nova te se često mijenja, i još uvijek postoje određeni optimizacijski problemi. Međutim, za izradu aplikacija koje nisu hardverski zahtjevne, smatram da je najbolji izbor upravo React Native.

Popis literature

Composition vs Inheritance - React. (27. 12 2021). Dohvaćeno iz React: https://reactjs.org/docs/composition-vs-inheritance.html

Derks, R., & Boduch, A. (2020). *React and React Native, Third Edition.* Birmingham: Packt Publishing Ltd.

Emotion. (7. 2 2022). Dohvaćeno iz Emotion: https://emotion.sh/docs/introduction

Flutter. (14. 8 2022). Dohvaćeno iz Flutter: https://flutter.dev/

Ionic. (16. 8 2022). Dohvaćeno iz Ionic: https://ionicframework.com/docs/

React - A javascript library for building user interfaces. (13. 12 2021). Dohvaćeno iz https://reactjs.org/

Redux. (18. 8 2020). Dohvaćeno iz Redux: https://redux.js.org/

styled-components. (7. 2 2022). Dohvaćeno iz styled-components: https://styled-components.com/

Tailwind CSS. (7. 2 2022). Dohvaćeno iz Tailwind: https://tailwindcss.com/

Popis slika

Popis tablica