Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra systémového inženýrství

Diplomová práce

Vývoj multiplatformní mobilní aplikace pomocí React Native

Bc. Jakub Mužík

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Jakub Mužík

Systémové inženýrství a informatika
Informatika

Název práce

Vývoj multiplatformní mobilní aplikace pomocí React Native

Název anglicky

Multiplatform mobile application development with React Native

---

Cíle práce

Diplomová práce je tématicky zaměřena na problematiku vývoje mobilní aplikace pro platformy iOS a Android. Cílem této práce je tedy seznámit čtenáře s multiplatformním frameworkem React Native, postupem vývoje mobilní aplikace a obecně s technologiemi týkající se této problematiky.
Hlavním cílem práce poté bude analýza, návrh a následná implementace konkrétní aplikace ve zmíněném frameworku React Native pomocí programovacího jazyku JavaScript.

Metodika

Práce sestává z teoretické a praktické části. Teoretická část práce vychází obecně ze studia odborných informačních zdrojů a na základě takto získaných poznatků budou popsána všechna východiska pro následující praktickou část a to konkrétně programovací jazyky, technologie a knihovny používané pro vývoj mobilní aplikace ve frameworku React Native.

Praktická část bude spočívat v samotné tvorbě mobilní aplikace zaměřené na tvorbu a zobrazování událostí a to na bázi mapy a aktuální polohy uživatele. Bude provedena analýza aplikace a definice základních funkcionalit, které budou v rámci této diplomvé práce implementovány. Na základě této analýzy bude následně provedeno rozvržení aplikace z hlediska navigace a také výběr potřebných technologií, knihoven a vhodné databáze pro zprovoznění této aplikace. Dále bude následovat samotná implementace.

Nejdříve bude provedena základní konfigurace React Native projektu, databáze Firestore a jejich vzájemné propojení. Následně bude zprovozněna autentizace uživatelů s možností vytvoření nového účtu pomocí klasické registrace či přihlášení přes sociální síť Facebook a dále už bude implementace pokračovat samotnou tvorbou klientské části aplikace za použití zmíněné databáze.

Výsledná aplikace bude otestována a závěrem budou shrnuty poznatky vycházející ze samotné implementace a také budou definovány následné možnosti rozvoje aplikace.

Doporučený rozsah práce

60-80 stran

Klíčová slova

React Native, JavaScript, Expo, Redux, Firestore, iOS, Android

---

Doporučené zdroje informací

HAVERBEKE, Marijn, 2019. Eloquent JavaScript, third edition. San Francisco: no starch press. ISBN 978-1-59327-950-9

Introduction to Expo [online]. [cit. 2022-03-02]. Dostupné z WWW: https://docs.expo.dev/

Redux Fundamentals, Part 1: Redux Overview [online]. 2021-06-25 [cit. 2022-02-17]. Dostupné z WWW: https://redux.js.org/tutorials/fundamentals/part-1-overview

What is Firebase? [online]. [cit. 2022-03-04]. Dostupné z WWW: https://firebasetutorials.com/what-is-firebase/

Předběžný termín obhajoby

2021/22 LS – PEF

Vedoucí práce

Ing. Martin Pelikán, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 30. 1. 2022

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 30. 1. 2022

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 30. 03. 2022

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Vývoj multiplatformní mobilní aplikace pomocí React Native" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 30.3.2022 _____

Poděkování

Rád bych touto cestou poděkoval panu Ing. Martinu Pelikánovi, Ph.D. za odbornou pomoc ve všech aspektech ohledně tvorby mé diplomové práce a za ochotu a rychlou domluvu při plánování konzultací.

Vývoj multiplatformní mobilní aplikace pomocí React Native

Abstrakt

V teoretické části práce popisuji nejdůležitější pojmy týkající se vývoje mobilních aplikací pomocí React Native spolu s technologiemi použitými pro rozšiřování funkcionalit tvořené aplikace. Dále je zde uveden popis Firestore databáze, která byla v rámci implementace použita.

Praktická část této práce obsahuje celý proces tvorby multiplatformní mobilní aplikace pomocí React Native a aplikace třetí strany Expo. Proces začíná analýzou aplikace, pokračuje konfigurací všech potřebných technologií spolu s vytvořením React Native projektu a jeho propojením s databází a končí samotnou implementací aplikace pomocí jazyku JavaScript.

Klíčová slova: React Native, JavaScript, Expo, Redux, Firestore, Firebase, iOS, Android

Multiplatform mobile application development with React Native

Abstract

In the theoretical part of this thesis, I describe the most important elements of mobile application development with React Native, together with the technologies used to extend functionalities of created application. The following is a description of the Firestore database that was used during the implementation.

The practical part of this thesis contains the whole process of creating a multiplatform mobile application with React Native and third-party application Expo. The process begins with the analysis of the application, continues with the configuration of all necessary technologies together with the creation of React Native project and its connection to the database and ends with the actual implementation of the application using JavaScript.

Keywords: React Native, JavaScript, Expo, Redux, Firestore, Firebase, iOS, Android

Obsah

Seznam obrázků

Seznam použitých zkratek

CLI – Command-line Interface

SQL – Structured Query Language

API – Application Programming Interface

URL – Uniform Resource Locator

URI – Uniform Resource Identifier

SDK – Software Development Kit

LAN – Local Area Network

HTTP – Hypertext Transfer Protocol

UI – User Interface

DOM – Document Object Model

1 Úvod

Téměř každý člověk v dnešní době disponuje chytrým telefonem a používá tak nějakou mobilní aplikaci ať už pro jakýkoliv účel. V současné době existuje takové množství mobilních aplikací, až by se dalo říct, že neexistuje oblast či požadavek, který by již nebyl nějakou dostupnou aplikací pokryt. Aktuálně je v této oblasti tedy velice těžké prorazit, a proto při výběru zaměření nové aplikace hledáme sebemenší maličkosti, které považujeme za doposud neimplementované, a které by tak uživateli mohli ještě více usnadnit jeho každodenní život a my tak měli šanci, že se naší aplikaci dostane zalíbení.

Téma diplomové práce bylo autorem vybráno jak z důvodu jeho zkušeností a znalostí v oblasti vývoje aplikací, které při její tvorbě mohl uplatnit, tak nepochybně z důvodu jeho zájmu o tuto oblast a neustálé se zdokonalování v ní. Autor tak při tvorbě diplomové práce tedy mohl paralelně pracovat na posunu k úspěšnému zakončení jeho magisterského studia a zároveň se zdokonalovat a získávat nové znalosti v oblasti, o kterou má zájem. Takto získané znalosti poté autor bude moci zužitkovat jak v pracovním, tak v osobním životě.

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je tématicky zaměřena na problematiku vývoje mobilní aplikace pro platformy iOS a Android. Cílem této práce je tedy seznámit čtenáře s multiplatformním frameworkem React Native, postupem při vývoji mobilní aplikace a obecně s technologiemi týkající se této problematiky.

Hlavním cílem práce poté bude analýza, návrh a následná implementace konkrétní aplikace ve zmíněném frameworku React Native pomocí jazyku JavaScript.

2.2 Metodika

Práce sestává z teoretické a praktické části. Teoretická část práce vychází obecně ze studia odborných informačních zdrojů a na základě takto získaných poznatků budou popsána všechna východiska pro následující praktickou část, a to konkrétně programovací jazyky, technologie a knihovny používané pro vývoj mobilních aplikací ve frameworku React Native.

Praktická část bude spočívat v samotné tvorbě mobilní aplikace zaměřené na tvorbu událostí a jejich následné zobrazení za použití mapy a polohy uživatele. Nejdříve bude provedena analýza a definice základních funkcionalit aplikace, které budou v rámci této diplomové práce implementovány. Na základě této analýzy bude následně proveden výběr a konfigurace všech technologií, knihoven a databáze potřebné pro zprovoznění aplikace. Dále už bude následovat samotná implementace. Nejdříve bude zprovozněna autentizační část s možností registrace a přihlášení pomocí emailu a hesla či účtu na sociální síti Facebook a dále už bude implementace pokračovat hlavní částí aplikace a jejích dílčích obrazovek.

Výsledná aplikace bude na obou platformách otestována a závěrem budou shrnuty poznatky vycházející ze samotné implementace spolu s dalšími možnostmi případného rozšíření aplikace.

3 Teoretická východiska

3.1 Vymezení pojmů

3.1.1 HTTP

HTTP, v celém znění HyperText Transfer Protocol, je v současnosti nejpoužívanější protokol pro komunikaci na internetu. Tento protokol představuje řadu pravidel, které poskytují interakci mezi klientem a serverem, kde klientem se rozumí náš webový prohlížeč a serverem cílová webová stránka. Interakce zpravidla začíná na straně klienta, kde uživatel vytvoří požadavek na server – cílovou webovou stránku. Tento požadavek poté HTTP přebere a vytvoří spojení mezi klientem a serverem pomocí TCP protokolu. Jakmile je toto spojení úspěšně vytvořeno, bude klientův požadavek pomocí HTTP odeslán na sever, který požadavek zpravuje a následně odešle jeho výsledek zpět klientovi. Při odesílání tohoto zpracovaného požadavku zpět klientovi, první věc, která se posílá, je stavový kód. Stavový kód pomocí číselné kombinace popisuje výsledek zpracování požadavku, kde jeho první číslo indikuje výsledek operace a následná čísla poté specifické události týkající se tohoto výsledku. (1)

3.1.2 API

Application Programming Interface, v doslovném překladu rozhraní pro programování aplikací, je sada pravidel popisující, jakým způsobem spolu mají jednotlivé systémy nebo aplikace komunikovat. API tak umožňuje společnostem zpřístupnit data a funkce svých aplikací skrze webové rozhraní a umožnit tak dvěma odděleným systémům spolu vzájemně komunikovat a vyměňovat si potřebná data. Toto rozhraní je vždy zdokumentováno a vývojáři ho tak mohou jednoduše použít, aniž by věděli, jak byly jeho jednotlivé služby implementovány. Využití API za poslední desetiletí vzrostlo do také míry, že by bez něj mnoho současných nejpopulárnějších webových aplikací ani nemohlo existovat. Pokaždé tedy když se v nějaké webové aplikace přihlašujeme, ukládáme nějaké data a nebo jen zobrazujeme část jejího obsahu, využíváme API. (2)

3.1.3 JavaScript

JavaScript je interpretovaný programovací jazyk, který vývojáři používají především pro implementaci dynamičnosti jejich webových stránek a v kombinaci se značkovacím jazykem HTML a kaskádovými styly tvoří základ pro tvorbu moderních webových stránek. Javascript kód je do stránky vkládán buď na přímo pomocí tagu *script* nebo pomocí jeho

reference na externí soubor s příponou .js a po jejím načtení je tento kód spolu s HTML a CSS stáhnut a uložen ve webovém prohlížeči u klienta. Toto je jeden z hlavních rozdílů oproti server-side programovacím jazykům, které jsou uloženy a běží pouze na straně serveru. V dnešní době už lze ale JavaScript pomocí knihovny Node.js spouštět i na serveru a nebo také na jakémkoliv jiném zařízení, které disponuje speciálním programem nazvaným JavaScript engine. Navíc je zde v současnosti díky popularitě tohoto jazyka také několik JavaScript frameworků, které zjednodušují implementaci složitých projektů, jako například Angular, jQuery nebo React. Mezi největší výhody JavaScriptu patří mimo jiné rychlost, a to z důvodu jeho běhu přímo v prohlížeči a oproštěním se tak od potřeby provádět požadavek na server pokaždé, když uživatel se stránkou provede nějakou interakci. Zviditelnění a přístup k JavaScript kódu u klienta sebou ale nese také pár nevýhod, a to například jeho možné zneužití ke škodlivým účelům. Toto je mimochodem jeden z důvodů, proč se někteří provozovatelé na jejich webových stránkách rozhodli JavaScript úplně zakázat. (3)

3.1.4 MVC

Model View Controller představuje způsob pro uspořádávání zdrojového kódu. Hlavní myšlenka MVC struktury je taková, že každý kus zdrojového kódu má nějaký svůj účel a tyto účely mohou být různé. Jedna část kódu může být zodpovědná za ukládání a manipulaci s daty, druhá za vizuální stránku aplikace a jiná zas třeba za její funkcionality. MVC je tedy organizace částí kódu z hlediska jejich účelu do 3 skupin – Model, View, Controller. (6)

Model

Model představuje logiku týkající se dat, se kterými uživatel pracuje. Pokud máme tedy například jednoduchou aplikaci pro správu a výpis denních úkolů, Model bude obsahovat definici co jednotlivé úkoly představují a definici jakým způsobem se mají úkoly vypisovat. (6)

View

Tato část kódu představuje čistě vizuální stránku aplikace, respektive koncové funkce a prvky, se kterými bude uživatel interagovat a jakým způsobem budou zobrazovány. (6)

Controller

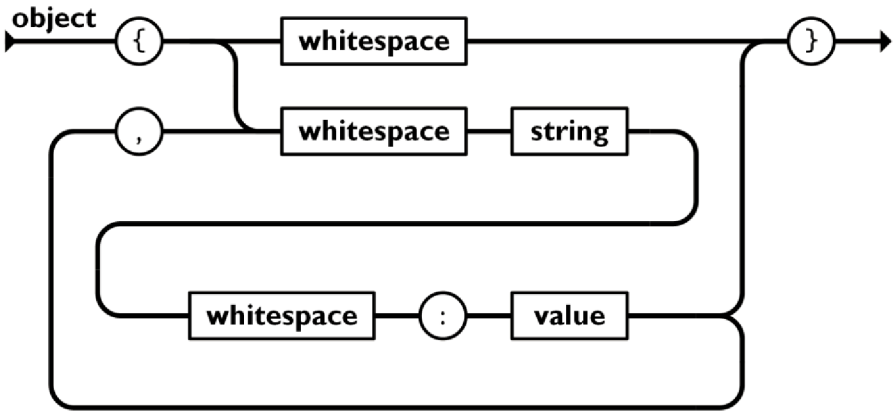
Controller leží mezi View a Modelem, a tedy přijímá vstupy od uživatele a následně rozhoduje, jak s nimi naloží. Controller se považuje za mozek aplikace, jelikož zprostředkovává veškerou bussinesovou logiku. Pomocí modelu manipuluje s daty a takto zpracovaná data poté zobrazuje ve View. (6)

3.1.5 JSON

JSON je zkratka pro JavaScript Object Notation a je to způsob uskupení dat nezávislého na programovacích jazycích určený pro jejich přenos a ukládání. Výhoda tohoto způsobu zápisu dat je mimo jiné jeho člověku čitelný formát a jednoduchost orientovat se v něm. Tento formát, jak už z názvu vyplývá, je založen na struktuře objektů v jazyce JavaScript a je složen ze dvou základních struktur – Objekt a Pole, které je možné libovolně kombinovat a vkládat do sebe sama a tvořit tak velice komplexní datovou strukturu.

Objekt

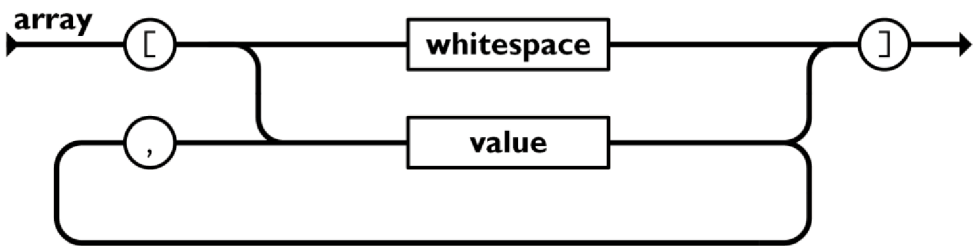
Objekt představuje neuspořádanou množinu klíč-hodnota párů obalených do složených závorek. Mezi každým klíčem a jeho hodnotou je dvojtečka a jednotlivé páry se oddělují pomocí čárky. Klíčem se zde rozumí textová hodnota reprezentující název daného páru a hodnota poté může obsahovat data následujících datových typů: text, číselná hodnota, boolean, null, objekt a pole. (16)



Obrázek 1 Struktura JSON objektu (Internetový zdroj dle (16))

Pole

Pole je uspořádaná množina hodnot obalených do hranatých závorek, které stejně jako v předchozím případě mohou nabývat hodnot několika datových typů včetně dalších objektů a polí. Jednotlivé hodnoty lze poté získat pomocí indexu jejich umístění v poli. (16)

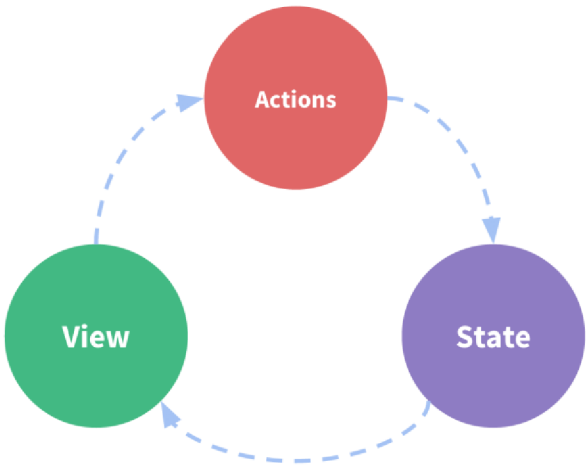


Obrázek 2 Struktura JSON pole (Internetový zdroj dle (16))

3.1.6 Redux

Redux je nástroj pro správu stavu aplikace, který funguje na principu centralizovaného úložiště, respektive samostatného místa obsahující stav aplikace, známého také jako jediný zdroj pravdy. Do tohoto místa má poté každá jednotlivá komponenta aplikace přístup a odpadá tak nutnost si data, respektive stavy komponent předávat pouze mezi příbuznými komponentami ve vztahu parent-child nebo naopak. Tento nástroj je nejčastěji používat pro tvorbu React aplikací, nicméně je možné jej použít s jakýmkoliv jiným JavaScript frameworkem. Redux se skládá z následujících prvků: (8)

- Store – objekt sloužící jako centralizované úložiště aplikace
- Action – pomocí Action se provádí změny v úložišti, skládá se z parametru určující typ změny a případně také jejích dat
- Reducer – obsahuje pro každou z typů akcí funkci, která změna na základě předem dané logiky zpropaguje do úložiště
- Dispatch – pomocí dispatch se zavolají změny v úložišti, je to centralizovaná možnost, jak změny provádět
- Selector – funkce pro selekci námi požadovaných dat z úložiště

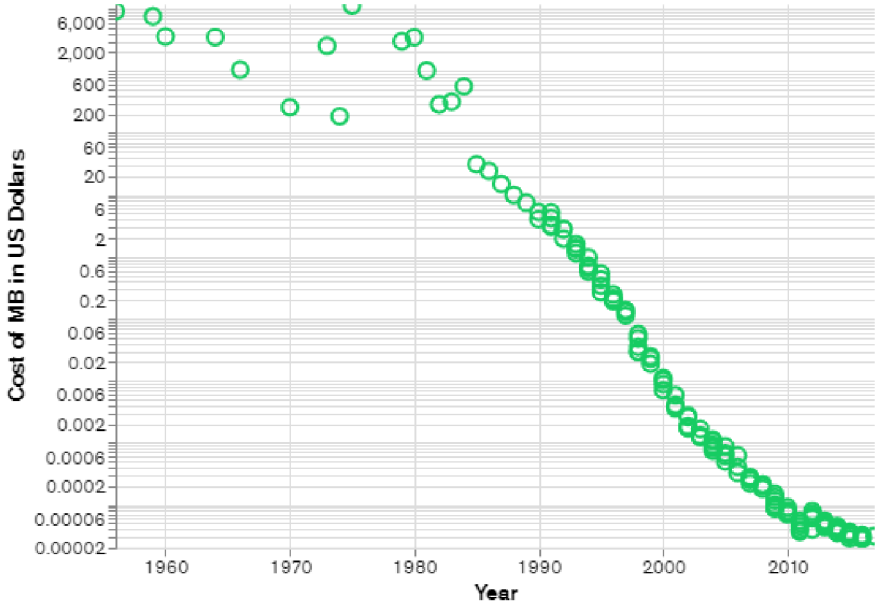


Obrázek 3 Jednosměrný tok dat v Redux (Internetový zdroj dle (8))

3.1.7 NoSQL databáze

NoSQL databáze jsou obecně ty, které ukládají data v jiném formátu než pomocí relačních tabulek. Takto schematicky založené databáze se začali objevovat na konci dvacátého století, kdy cena za množství uložených dat začala rapidně klesat. Spolu s klesající cenou totiž začalo nepřímo úměrně stoupat množství dat, které aplikace potřebovali ukládat či získávat z databáze. Tyto data přicházeli ve všech různých formátech – strukturovaných, polo strukturovaných nebo i polymorfních, a tudíž tvorba databáze s nějakým předem definovaným schématem se projevila jako téměř nemožný úkol. Tento problém byl tedy vyřešen pomocí NoSQL databází, které vývojářům umožňují ukládat veliké množství dat v předem nestrukturovaném formátu a dávají tak aplikacím větší flexibilitu. Ve stejnou chvíli se začalo dostávat popularitě i Cloud computingu, kdy vývojáři začali čím dál více používat cloudové servery pro ukládání dat či běh jejich aplikací a nebylo tomu pochopitelně jinak ani u NoSQL databází. Současně třemi nejčastějšími typy NoSQL databází jsou: (17)

- Dokumentové databáze – data jsou ukládána do dokumentů podobající se JSON objektům
- Klíč-hodnota databáze – jednodušší typ databáze kde její jednotlivé položky představují klíč-hodnota páry
- Graph databáze – ukládá data to uzlů a hran, kde uzly obvykle obsahují informace o lidech, místech a věcech, zatímco hrany obsahují informace o vztazích mezi uzly



Obrázek 4 Cena za MB dat v průběhu času (Internetový zdroj dle (17))

3.2 React

React je v současnosti jedna z nejpopulárnějších JavaScript knihoven pro tvorbu uživatelského rozhraní a to především pro single-page aplikace. Má na starosti *View* vrstvu z modelu Model-View-Controller, která je stavěna za pomoci znovupoužitelných UI kontejnerů nazvaných komponenty. React byl vytvořen Jordanem Walkem zaměstnaného na pozici softwarového inženýra ve společnosti Facebook a v roce 2011 byla tato knihovna poprvé nasazena na Facebook newsfeed aby vyřešila nastávající problémy týkající se údržby kódu. React vývojářům umožňuje tvořit komplexní webové aplikace, které mohou měnit obsah svých stránek bez nutnosti jejich kompletního znovu načítaní. Hlavními účely Reactu jsou rychlost, škálovatelnost a jednoduchost. Vzhledem k tomu, že React sám o sobě zajišťuje pouze *View* vrstvu, respektive správu stavu komponent a jejich vykreslení do DOM, je pro tvorbu React aplikací obvykle potřeba také použití doplňujících knihoven, a to například pro správu dat nebo implementaci navigace. (4)

3.2.1.1 Virtual DOM

Manipulace s DOM je obecně jádrem moderního interaktivního webu. Bohužel je ale také mnohem pomalejší než většina JavaScript operací. Tato pomalost je způsobena především tím, že většina JavaScript frameworků aktualizují DOM mnohem častěji, než je ve skutečnosti potřeba. Představme si například situaci, kdy máme k dispozici seznam deseti zaškrtávacích políček a zaškrtneme první z nich. Většina JavaScript frameworků by v tuto chvíli aktualizovalo a znovu vykreslilo celý tento seznam. A jelikož jsme změnili pouze jeden prvek seznamu a zbylých devět zůstalo nezměněných, provedlo by desetkrát více práce, než bylo ve skutečnosti třeba. Znovu vykreslení jednoho seznamu nepředstavuje pro výkonnost zásadní problém. Nicméně komplexní aplikace mohou disponovat velikým množstvím takové nadbytečné DOM manipulace, což může vést k vážnému výkonnostní poklesu. Pro vyřešení toho problému, React přišel s novým konceptem nazvaným Virtual DOM. (5)

V Reactu je pro každý objekt v DOM vytvořena jeho další kopie představující *Virtual DOM object*. Tato kopie disponuje všemi atributy jako jeho originál až na možnost reálně změnit co je na obrazovce, z čehož také vyplývá rychlost její manipulace. Manipulace Virtual DOM objektu je oproti klasickému DOM objektu mnohem rychlejší, protože se jeho obsah nevykresluje na obrazovku. Pokaždé když v uživatelském rozhraní vykreslíme nebo nějakým způsobem pozměníme obsah React komponenty, všechny její příslušné Virtual DOM objekty se aktualizují. Jakmile se aktualizují, React je porovná s verzemi virtual objektů před touto

změnou a zjistí tak, které jednotlivé objekty byly změněny. A na základě tohoto zjištění a pouze a jedině pro změněné objekty poté React změny propíše do odpovídajících reálných DOM objektů a změnu tak zviditelní na obrazovce. Pokud bychom tedy tento koncept aplikovali na předchozí příklad se seznamem deseti prvků, kde jsme změnili pouze jeden z nich, změna z hlediska aktualizace DOM objektů by proběhla pouze pro jeden DOM objekt odpovídající změněnému prvku seznamu a zbytek by zůstal nedotčen. (5)

3.2.1.2 JSX

JSX neboli JavaScript Syntax Extension, je rozšíření syntaxe JavaScript, které je doporučováno použít pro tvorbu React komponent. Toto rozšíření poskytuje možnost kombinace HTML/XML značek spolu s JavaScript kódem v jednom souboru. Oproti dřívější možnosti vkládat JavaScript kód do HTML souborů, nám toto rozšíření naopak umožňuje vkládat HTML kód do JavaScript souborů, kde takto vložený HTML kód je posléze při spuštění aplikace transpilátorem nalezen a přetransformován do standartních JavaScript objektů. (7)

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

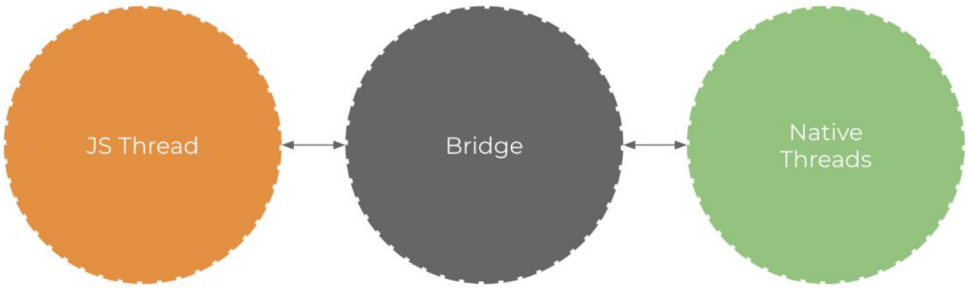
Obrázek 5 Ukázka syntaxe JSX (Internetový zdroj dle (7))

3.3 React Native

React Native je JavaScript framework pro tvorbu nativně vykreslovaných mobilních aplikací pro platformy iOS a Android. Je postaven na knihovně React a umožňuje vývoj pro obě zmíněné platformy pomocí stejné kódové základny. React Native byl poprvé zveřejněn společností Facebook jako open source projekt v roce 2015. Za pouhých pár let tento projekt získal velikou oblíbenost a postupem času se tak stal jedním z nejpoužívanějších řešení pro tvorbu mobilních aplikací. V současnosti je tento framework použit pro několik předních světových mobilních aplikací jako je například Facebook, Instagram nebo Skype. (9)

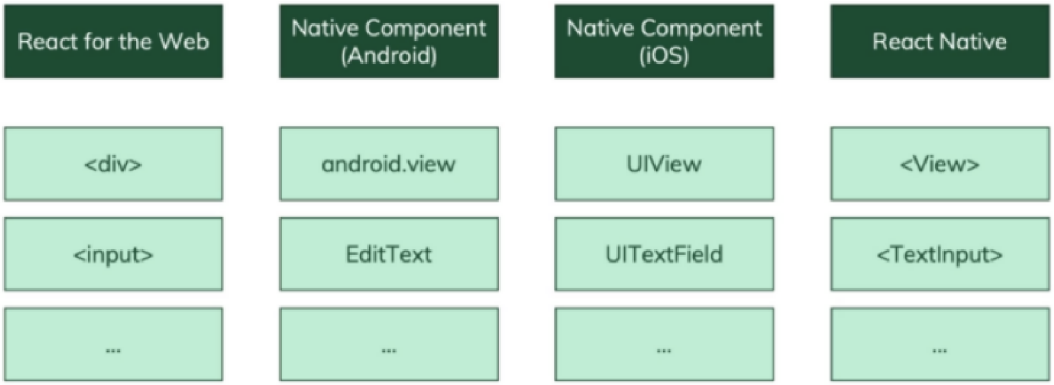
Jak již bylo zmíněno, React Native používá stejný JavaScript kód pro tvorbu uživatelských rozhraní obou platforem, ale každá z platforem výslednou aplikaci poté vykreslí nativně za pomocí svých elementů a komponent. Toto je dosaženo pomocí React Native konceptu nazvaného *bridge*, který disponuje schopností komunikovat s oběma stranami – JavaScript a Nativním vláknem, a to pomocí interoperabilní notace JSON. Pokud tedy chceme

něco zobrazit, JavaScript odešle JSON objekt obsahující požadavky na zobrazení prostředníku *bridge,* který požadavek přijme, přeloží do jazyka dané platformy a platformu ohledně požadavku následně kontaktuje. (9)



Obrázek 6 Komunikace JS a nativního vlákna skrze bridge (Internetový zdroj dle (9))

Uživatelské rozhraní se v React Native, oproti standartním HTML tagům jako je tomu u webových aplikací, tvoří pomocí speciálních React komponent, které jsou následně kompilovány do nativních widgetů dané platformy. Mezi tyto komponenty patří například *View*, která přestavuje základní stavební prvek uživatelského rozhraní a v klasickém webovém rozhraní by představovala element *div*. Na obrázku níže lze vidět jakým způsobem jsou komponenty View a TextInput interpretovány v různých prostředích. (10)



Obrázek 7 Porovnání React Native, Android a iOS komponent (Internetový zdroj dle (10))

3.3.1 React Native CLI

React Native CLI, jakožto jeden ze dvou možných způsobů, jakým tvořit React Native aplikace, je rozhraní zaštiťované oficiálním React Native týmem, které poskytuje základní příkazy pro inicializaci, vývoj a následnou manipulaci s aplikací. Při tomto způsobu vývoje je potřeba mít ještě nainstalovaná vývojová prostředí, jako je Android Studio pro platformu

Android a Xcode pro iOS. Pokud je zde potřeba použít některou z nativních funkcionalit telefonu, jako například kamera, bude zapotřebí použít některý z balíčku třetích stran, kde jejich konfigurace může být velice komplexní a obtížná. Mimo instalace a využívání balíčků třetích stran, je zde i druhá možnost, která je mimochodem jednou z největších výhod použití React Native CLI, a to napsání vlastního nativního kódu pro obě platformy zvlášť a jeho napojení do JavaScriptu. Tato možnost tedy přináší kompletní kontrolu nad aplikací, a proto pokud jsou požadavky na mobilní aplikaci velice komplexní a vyžadují spoustu manuálních úprav, je React Native CLI vývojáři často vybírán pro implementaci jejich aplikací. (11)

3.3.2 Expo

Expo je sada nástrojů a služeb vytvořené okolo React Native, které odebírají komplexitu obsaženou při vývoji pomocí React Native CLI a usnadňují tak vývoj, build a testování React Native aplikací. Pro vývoj pomocí Expo není potřeba specifických vývojových prostředí jako je Android Studio nebo Xcode, protože zde odpadá nutnost psaní nativního kódu a vývoj tak probíhá pouze pomocí jednoho jazyku – JavaScript. Je zde k dispozici vývojářský balíček obsahující řadu dodatečných API, které poskytují nadstandartní přístup k nativním funkcionalitám telefonu jako například kamera, mapa nebo souborový systém, které navíc není zapotřebí nijak zvlášť konfigurovat či upravovat. Další výhodou je možnost testování aplikací na reálných mobilních zařízeních pomocí aplikace Expo GO a naskenováním vygenerovaného QR kódu. Jednou z nevýhod použití Expo je limitace použití pouze takových funkcionalit, které Expo nabízí. Pokud zde tedy bude požadavek na implementaci něčeho, co není obsaženo v Expo vývojářském balíčku, nebude to kvůli nemožnosti psaní vlastního nativního kódu možné. (12)

3.4 Firebase

Firebase je cloudová platforma pro tvorbu webových a mobilních aplikací, která poskytuje širokou škálu nástrojů a služeb pomáhajících k tvorbě vysoce kvalitních aplikací a rozšiřování uživatelské základny. Tyto služby, které by si vývojáři za jiných okolností museli vytvářet sami, pokrývají typické služby a požadavky pro implementaci moderních aplikací, a vývojáři se tak při využití Firebase mohou více zaměřit a soustředit na vývoj aplikace jako takové. Firebase v roce 2011 původně vznikl jako startup pod názvem Envolve, který sloužil jako poskytovatel API pro implementaci chatu na webových stránkách pomocí posílání a zobrazování dat v reálném čase. Postupem času si ale vývojáři aplikací uvědomili potenciál této

služby a začali ji tak využívat pro posílání i jiných dat než jenom chatovací zprávy. Začaly se zde posílat různá data jako například stavy herních aplikací a zakladatelé Envolve se tak rozhodli oddělit jejich služby do samostatných systémů – chat a real-time architekturu. Chvíli poté v roce 2012 vznikl Firebase jako oddělená společnost poskytující Backend-as-a-Service s real-time funkcionalitami a o 2 roky později byl už Firebase odkoupen společností Google a začal se tak rapidně vyvíjet co se týče funkcionalit a také popularity. (13)

3.4.1 Autentizace

Firebase Authentication je backend služba poskytující sadu vývojářských nástrojů pro autentizaci uživatelů do našich aplikací. V normálním případě by tato implementace mohla trvat i několik měsíců, ale s Firebase je možné autentizaci zprovoznit pomocí méně něž deseti řádků kódu, které za nás vyřeší standartní problémy typu připojení do databáze, hashování hesla anebo také single sign on funkcionalitu. Mezi nejvíce používané metody autentizace poskytované Firebase patří: (14)

- Email a heslo
- Telefonní číslo
- Google
- Facebook
- Twitter

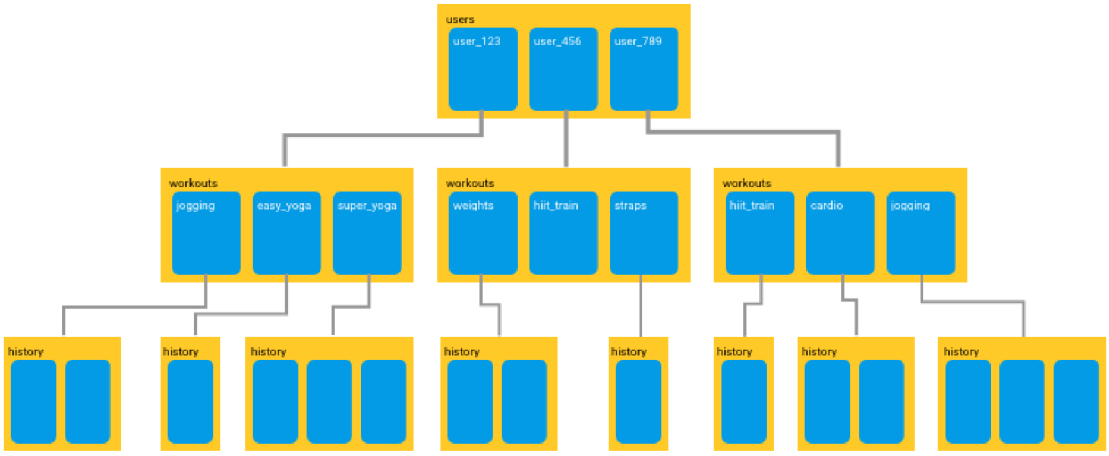
3.4.2 Realtime databáze

Firebase Realtime Database, jakožto první ze dvou databází poskytovaných platformou Firebase, je cloudová NoSQL databáze, která umožňuje ukládat data a synchronizovat je mezi uživateli v reálném čase. Z pohledu struktury tato databáze představuje jeden modifikovatelný JSON objekt, který je zpřístupněn skrze jediné API poskytující jak aktuální hodnotu tohoto objektu, tak jakékoliv změny v něm provedené. Další výhodou, která tato databáze přináší, je ošetření výpadků internetového připojení. Ve chvíli, kdy uživatelé ztratí přistup k internetu, vývojářský balíček Realtime Database použije lokální cache úložiště daného zařízení a uloží do něj uživatelovi nezpracované požadavky. Jakmile bude uživatel znovu připojen, budou tyto lokálně uložená data a požadavky automaticky zpracovány. Realtime Database také umožňuje propojení s Firebase autentizačním systémem a poskytuje tak kompletní prostředky pro implementaci uceleného a intuitivního autentizačního procesu pro vyvíjené aplikace. (14)

Obrázek 8 JSON data v Realtime Database (Internetový zdroj dle (14))

3.4.3 Firestore

Druhou a současně poslední databází, která platforma Firebase nabízí je Firestore. Firestore je opět Cloudová NoSQL databáze poskytující jednoduchý zápis, čtení a synchronizaci dat pro mobilní a webové aplikace. Byť se zdá, že mezi Firestore a předchozí databází není žádný rozdíl, Firestore ve skutečnosti přichází s řadou věcí, kterými se od Realtime Database zásadně liší. Data jsou ve Firestore namísto jednoho JSON objektu strukturovány do takzvaných kolekcí a dokumentů, kde kolekce, identifikovatelná pomocí jména, představuje datový kontejner skládající se z jednotlivých dokumentů. Každý dokument poté obsahuje jednoznačný identifikátor a několik klíč-hodnota dvojic, kde hodnota může obsahovat data libovolného typu – od textových řetězců po JSON objekty či dokonce ukazatele na další vnořené kolekce. Jedna kolekce se tedy skládá z dokumentů, které mohou ukazovat na další vnořené kolekce, kde tyto vnořené kolekce mohou obsahovat opět dokumenty ukazující na další vnořené kolekce. Tento proces vnořování můžeme opakovat dle potřeby a tvořit tak komplexní strukturu databáze. (14)



Obrázek 9 Kolekce a Dokumenty ve Firestore (Internetový zdroj dle (14))

3.4.4 Firebase Storage

Firebase Storage je cloudové úložiště, do kterého je možné ukládat binární soubory jako jsou nejčastěji obrázky, videa nebo pdf soubory. Po úspěšném nahrání těchto souborů je vrácena URL adresa, kterou je možné použít pro opětovné stáhnutí daného souboru. Pomocí těchto URL adres tak může daná aplikace na vyžádání stahovat například různé dočasně potřebné soubory a ve výsledku tak dosáhnout snížení její konečné velikosti. Velikou výhodou je zde opět podpora výpadků internetového připojení. Pokud je proces nahrávání z nějakého důvodu přerušen, je možné se k němu později vrátit a v nahrávání pokračovat. Toto je automaticky ošetřeno pomocí vývojářského balíčku Firebase Cloud Storage SDK. Dále by se mělo brát na paměti to, že kdokoliv s přístupem ke stáhnutelné URL adrese souboru, je ve výchozím nastavení oprávněn tento soubor stáhnout. A vzhledem k tomu, že cena Firebase je přímo úměrná velikosti užití platformy včetněmnožství stáhnutých dat, nemuselo být to být z hlediska nákladů výhodné. Pro tento případ slouží funkcionalita Firebase Cloud Storage Rules, která poskytuje definici několika pravidel nutných pro splnění před přístupem do úložiště. V případě zákazu neoprávněným uživatelům stahovat soubory z úložiště, by se zde mohla definovat kontrola, zda-li je daný uživatel s Firebase autentizován a pokud ne, nebude mu umožněno soubor stáhnout. (15)

4 Vlastní práce

4.1 Analýza požadavků

Před začátkem tvorby jakékoliv aplikace je vždy dobré, si definovat základní funkcionality, jakými budeme chtít, aby naše aplikace disponovala. Je nutné si tedy určit minimální životaschopný produkt, který v rámci této diplomové práce implementujeme a vytvoříme tak funkční soběstačnou aplikaci. Nutno podotknout, že můj záměr je tuto aplikaci v budoucnu skutečně vydat v podobě produkční verze dostupné v App store a Google play, a tak výsledek této diplomové poté využiji jako základní stavební kámen, který budu dále zdokonalovat a rozšiřovat o další doplňující funkcionality na což ale ještě také navážu v závěrečné části této práce.

Aplikace bude fungovat na mobilních platformách iOS a Android a bude sloužit především jako online platforma pro vyhledávání neoficiálních událostí v okolí uživatele pomocí polohových služeb a mapy. Uživatel tak například bude moci okamžitě získat přehled o tom, co se v jeho okolí děje, nehledě na tom, zdali v daném místě někoho zná nebo zdali se v něm vůbec orientuje. Události budou rozděleny do několika kategorií, které budou filtrovatelné a uživatel tak bude moci najít události dle svých preferencí. Označení událostí jako neoficiální v tomto kontextu znamená, že budou tvořeny fyzickými osobami, respektive uživateli aplikace, a ne skrze oficiální organizace. Uživatelé tak tedy kromě vyhledávání budou moci události také tvořit a zvát na ně své přátelé. Může se například jednat o fotbalový zápas nebo skupinovou jógu v parku, společné učení na určitý předmět nebo zkoušku, oslava narozenin nebo cokoliv jiného. Uživatelé se v rámci aplikace budou moci "sledovat", podobně jako je tomu například u aplikace Instagram a mohou tak být například notifikováni o vytvoření nových událostí jejich přátel. K tomuto bude užitečná další funkcionalita této aplikace, která umožní přihlášení pomocí účtu z platformy Facebook a natáhnutí tak již existujících přátel do této aplikace. Bude se ale jednat pouze o přátele, které se v naší aplikaci již zaregistrovali. Události budou dále dvojího typu – veřejné a soukromé. Soukromé události budou dostupné pouze pozvaným uživatelům, a naopak veřejných událostí se budou moci účastnit všichni uživatelé nehledě na tom jestli byli pozváni či nikoliv. Každá událost bude mimo jiné obsahovat základní informace jako je místo konání, datum a čas konání, název a popis.

Na základně analýzy požadavků budu po aplikaci v rámci minimálního životaschopného produktu požadovat následující základní funkcionality:

- Funkčnost aplikace na obou mobilních platformách – iOS, Android

- Registrace a přihlášení pomocí e-mailu/hesla nebo platformy Facebook
- Zobrazit událostí na mapě s využitím polohových služeb telefonu
- Navigace ke konkrétní události za použití Google Maps
- Možnost vyhledávat a sledovat ostatní uživatele aplikace
- Tvorba událostí s možností poslat pozvánku přátelům
- Zobrazení notifikací
- Zobrazení a editace profilu

4.2 Analýza technologií

Mobilní aplikace je obecně možné vyvíjet v několika různých frameworkách, za použití různých technologií, databází, nástrojů a postupů. V této části mé práce bych si na základě předchozí analýzy požadavků chtěl definovat, jakým způsobem a za pomocí jakých nástrojů a technologií naši aplikaci budeme implementovat.

Jelikož chceme, aby aplikace fungovala na obou mobilních platformách zároveň, tak jak už i z názvu vyplývá, aplikaci budeme vyvíjet ve framework React Native. Díky tomu nám pro obě platformy postačí pouze jedna zdrojová základna, což nám značně usnadní vývoj. Při vytváření React Native aplikací máme obecně 2 možnosti. A to vývoj čistě pomocí React Native CLI nebo za využití aplikace třetí strany Expo. První možnost, vývoj pomocí React Native CLI, je značně komplikovanější a používá se zpravidla pro velmi komplexní aplikace kde je zapotřebí řada dolaďování co se týče přístupu k funkcím telefonu či jejich vlastní implementace spolu s psaním nativního kódu pro každou z platforem. Pro vývoj naší aplikace zvolíme druhou možnost, aplikaci třetí strany Expo, která vývoj aplikace značně zjednoduší a pro splnění našich požadavků je zcela dostačující. Expo nám totiž mimo jiné také například umožní aplikaci testovat na reálných zařízeních pomocí mobilní aplikace Expo Go a naskenováním QR kódu, který Expo aplikace vygeneruje.

Dále bude potřeba se nějakým způsobem vypořádat i se správou a tokem dat v aplikaci. Vzhledem k její komplexnosti je už teď jasné, že se bude skládat z desítek různých komponent, které do sebe budou vnořovány do několika různých úrovní a tak pouhé předávání dat mezi komponentami ve vztahu rodič-dítě by po čase bylo velice komplikované a tedy nepřipadá v úvahu. Správu dat tedy vyřešíme knihovnou React Redux, která představuje jeden globální kontejner, do kterého budeme ukládat data a ke kterému bude mít každá komponenta v aplikaci přístup. React Redux ještě rozšíříme o knihovny Redux Persist, která nám umožní data uchovat

i po zavření aplikace a Redux Thunk, pomocí které budeme moci změnu dat provádět na základě nějakého asynchronního procesu – v našem případě například zavolání API.

Dále je jasné, že budeme potřebovat nějakou databázi, kam budeme ukládat jednotlivé události, uživatele samotné a obecně veškerá data, která by měla být dostupná například i ostatním uživatelům. Je také nutno vzít v potaz náš požadavek z předchozí analýzy na notifikace o změnách dat v reálném čase. Databáze, kterou zvolíme bude cloudová No-SQL databáze Firestore od společnosti Google. Tato databáze poskytuje notifikace v reálném čase, které nám například umožní uživatele informovat o nových událostech v čase jejich vytvoření, nebo také knihovnu Firebase JS SDK poskytující velice užitečné funkcionality jako je například registrace, nebo přihlášení.

Aplikace Expo dále poskytuje jak Single Sign On funkcionalitu, pomocí které se budeme moct do aplikace přihlásit za využití účtu z platformy Facebook, tak přístup k funkcím telefonu jako jsou polohové služby, kamera nebo navigace čímž splňujeme zbytek námi požadovaných funkcionalit z předchozí analýzy.

Po analýze technologií jsme zjistili, že pro tvorbu aplikace a splnění našich funkčních požadavků budeme potřebovat následující technologie:

- React Native
- Expo
- React Redux
- Firestore

4.3 Analýza uživatelského rozhraní

Poslední krok před tím, než se pustíme do samotného vývoje aplikace, bude analýza uživatelského rozraní. V této analýze si na základě našich předešlých požadavků navrhneme, jak by naše aplikace měla po vizuální stránce vypadat. Konkrétně se bude jednat o prvky jako jsou logo, splash obrazovka a wireframy jednotlivých obrazovek aplikace. Nutno podotknout, že tato fáze projektu by se rozhodně neměla podcenit, jelikož vizuální stránka aplikace je z hlediska úspěšnosti klíčová a rozhodně stojí za to se zamyslet, jestli není lepší tuto část přenechat profesionálním designerům. Nicméně v rámci této práce provedeme analýzu uživatelského prostředí sami.

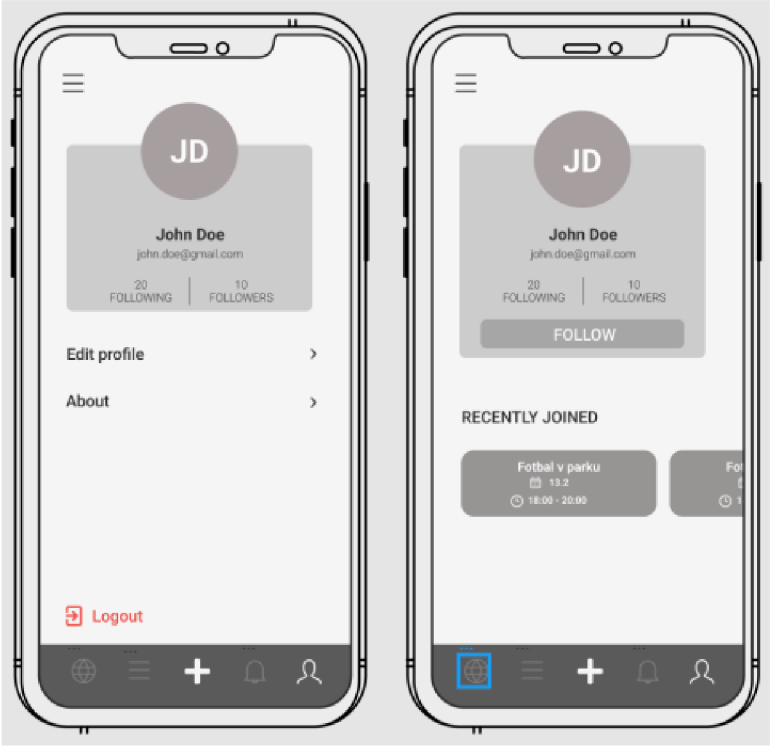
4.3.1 Wireframy

Wireframy jsou velice důležitou součástí plánovací fáze vývoje mobilní aplikace, jelikož nám před samotným vývojem poskytnou prvotní představu, jak by aplikace měla

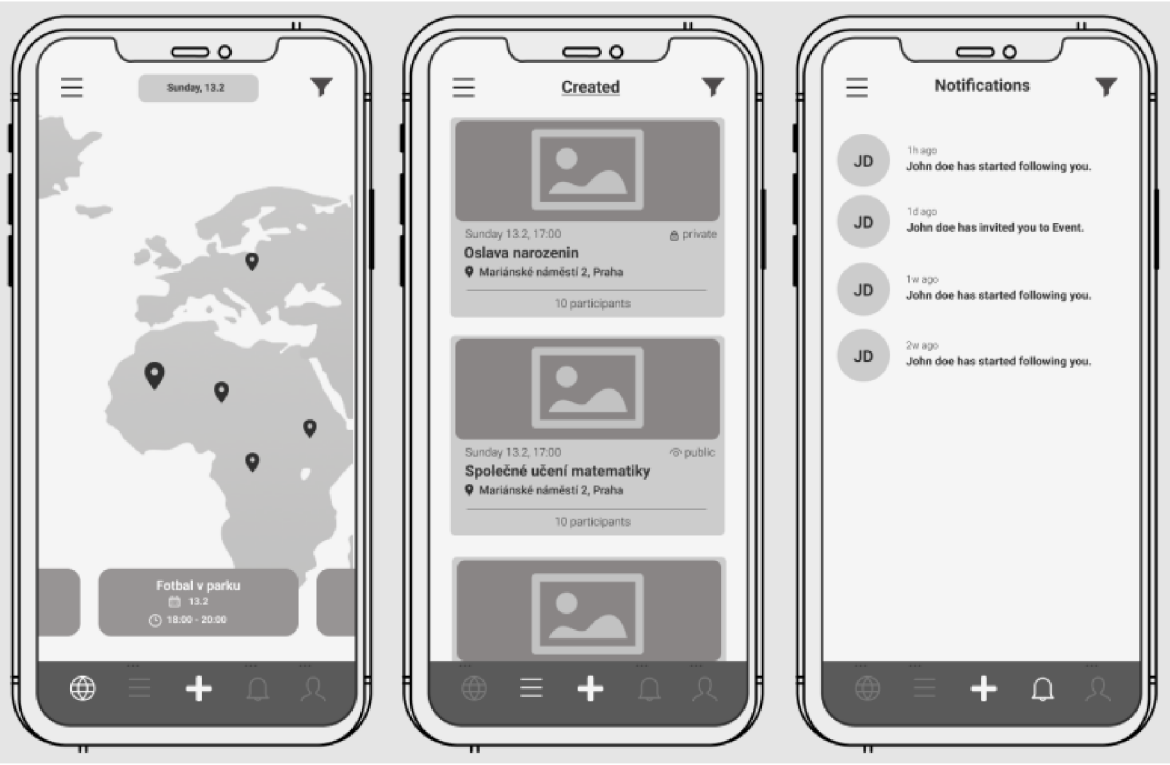
vypadat. Wireframy budou sloužit také jako takzvaná předloha při následném vývoji uživatelského rozhraní.

Pro tvorbu Wireframů využijeme webovou aplikaci Figma, která je velice intuitivní a která poskytuje řadu velice užitečných šablon a prvků týkajících se mobilních aplikací. Wireframy vytvoříme pro následující obrazovky: Přihlašovací a registrační stránka, zobrazení profilu – svého a cizího, stránka s mapou zobrazující události, výpis událostí do seznamu, seznam notifikací, vytvoření události a její zobrazení.

Obrázek 10 Přihlašovací a registrační obrazovka



Obrázek 11 Profily uživatelů

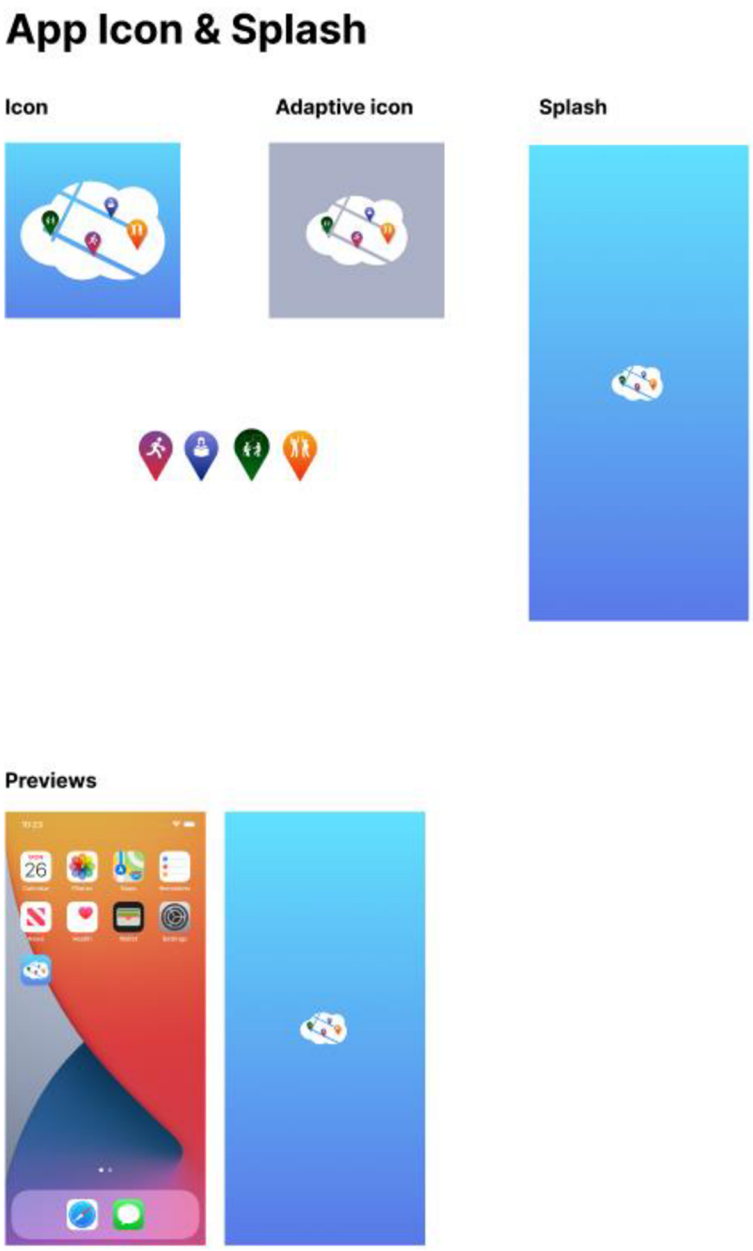


Obrázek 12 Obrazovky spodní navigace

Obrázek 13 Vytvoření a zobrazení události

4.3.2 Logo a Splash obrazovka aplikace

Logo a Splash obrazovku vytvoříme opět pomocí webového nástroje Figma. Splash obrazovka je obrazovka, kterou uživatel uvidí při načítání aplikace. Pro jejich tvorbu využijeme šablonu App Icon & Splash, kterou Figma zdarma nabízí. Tato šablona mimo jiné obsahuje pomocné prvky jako je například vycentrovaný kontejner pro umístění loga na splash obrazovku či náhled, jak logo aplikace bude vypadat na hlavní obrazovce telefonu.



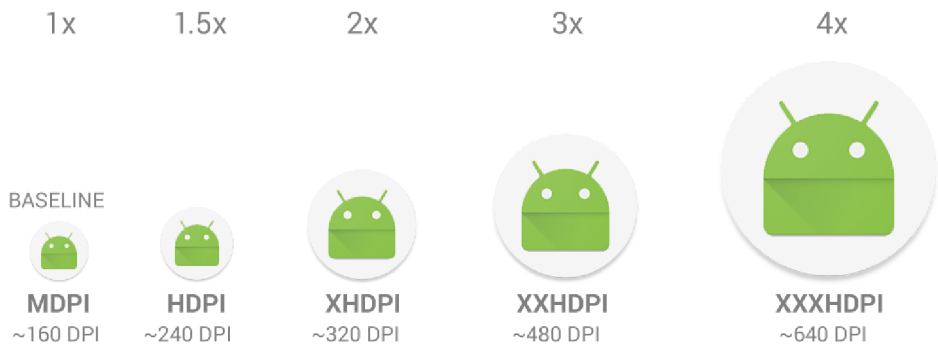
Obrázek 14 Logo a splash obrazovka v nástroji Figma

Na obrázku výše je vidět, že logo jsme vytvořili pro každou z platforem zvlášť. Logo s názvem "Icon", ve velikosti 1024 x 1024px, je určeno pro platformu iOS. Tato velikost odpovídá velikosti loga aplikace běžící na iOS zařízení s aktuálně největší dostupnou obrazovkou. Expo toto logo poté automaticky transformuje na optimální velikost odpovídající obrazovce aktuálního zařízení na kterém aplikaci spouštíme. Níže na obrázku, v sekci "Previews", je vidět jak bude aplikace zobrazena na domovské stránce zařízení.

V případě platformy Android, respektive loga s názvem "Adaptive Icon", je tomu trochu jinak. Logo pro platformu Android se skládá ze 2 částí – foregroundImage a

backgroundColor. V našem případě foregroundImage představuje transparentní logo z obrázku výše a backgroundColor barvu pozadí, která bude nakonfigurovaná později ve vývojovém prostředí, jakmile vytvoříme náš React Native projekt.

Co se týče Splash obrazovky a její velikosti, tak tam to nebude tak jednoduché tomu bylo s logy. U Splash obrazovky se budeme muset vypořádat s rozdíly v chování mezi systémy iOS a Android spolu s rozdílnými velikostmi obrazovek, jelikož naše Splash obrazovka pokrývá celý obsah obrazovky. iOS, oproti systému Android, dokáže Splash obrazovku roztáhnout nebo naopak zmenšit v závislosti na rozlišení displeje. Pro platformu iOS tedy, podobně jako tomu bylo u loga, vytvoříme pouze jednu Splash obrazovku s největším možným rozlišením. Android touto funkcionalitou bohužel nedisponuje, a tak budeme muset Splash obrazovku vytvořit v několika různých rozlišeních. Pro toto úskalí vývojářský tým zaštiťující Android rozdělil jejich zařízení do několika skupin dle hustoty pixelů obrazovek.



Obrázek 15 https://developer.android.com/training/multiscreen/screendensities

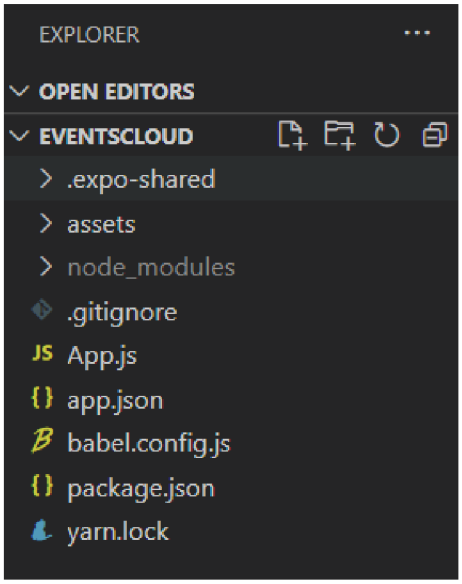
Pro tento případ tedy budeme muset Splash obrazovku vytvořit v několika různých rozlišeních odpovídajících těmto skupinám obrazovek. Konfiguraci Splash obrazovky a loga provedeme v následující kapitole, jakmile vytvoříme React Native projekt ve vývojovém prostředí.

Vzhledem k logu a tématu naší aplikace jsem se rozhodl aplikaci pojmenovat EventsCloud. Tento název vyjadřuje spojení 2 slov, kde první z nich – Events, vychází vůbec ze základního tématu naší aplikace a druhé – Cloud odpovídá jak logu aplikace, tak dostupnosti událostí odkudkoliv z internetu. Logo obsahuje 4 mapové značky, odpovídají čtyřem základním typům událostí, kterými naše aplikace bude disponovat.

4.4 Příprava a základní konfigurace

V tuto chvíli, abychom mohli začít, nám zbývá už jen nainstalovat 2 základní knihovny, které budeme potřebovat po celou dobu vývoje – NodeJS a Expo CLI. Knihovny nainstalujeme

pomocí příkazové řádky a nyní již máme veškeré potřebné podklady k tomu, abychom mohli začít s tvorbou aplikace. Přesuneme se tedy v příkazové řádce do adresáře, ve kterém chceme projekt vytvořit a zadáme příkaz „expo init EventsCloud“. Následně budeme vyzváni abychom zvolili, jaké Expo Workflow chceme použít. Pro naší aplikaci tedy vybereme Managed Workflow, které přináší řadu výhod zmíněných v předchozích částích mé práce a následně pro nás Expo knihovna vytvoří odpovídající projekt spolu se základní strukturu složek, před vytvořenými konfiguračními soubory, nainstalovanými knihovnami React a React Native a jejich závislostmi. Vytvořený projekt následně otevřeme v našem vývojovém prostředím VS Code.



Obrázek 16 Základní struktura projektu

V tuto chvíli je náš projekt/aplikaci možné spustit pomocí příkazu “expo start” zadaného z adresáře našeho projektu. Po zadaní příkazu bude aplikace spuštěna na vzdáleném Expo serveru a spolu s tím vygenerován QR kód, který po jeho naskenování umožní aplikaci spustit a testovat na reálném mobilním zařízení. Pro tento účel slouží mobilní aplikace Expo GO, která je zdarma dostupná jak z Google Play, tak z App Store. Jakékoliv změny provedené ve zdrojovém kódu budou poté automaticky reflektovány v této spuštěné aplikaci. Nutno podotknout, že abychom aplikaci mohli spustit a testovat na našem mobilním zařízení, je zapotřebí být telefonem připojený ke stejné LAN síti jako je počítač, ze které byla aplikace spuštěna. Pokud bychom chtěli umožnit aplikaci spustit na vzdáleném mobilním zařízení, je potřeba k příkazu přidat parametr “—tunnel”. Toto je velice užitečné například v situaci, kdy

vyvíjíme aplikaci pro zákazníka a chceme mu v průběhu vývoje dovolit se na aplikaci podívat a získat tak jeho zpětnou vazbu.

Obrázek 17 Spuštěná aplikace a vygenerovaný QR kód

V následujících podkapitolách připravím a provedu základní konfiguraci všech prostředí a technologií, které vychází z naší předchozí analýzy a které budeme pro tvorbu naší aplikace používat.

4.4.1 Konfigurace Loga a Splash obrazovky

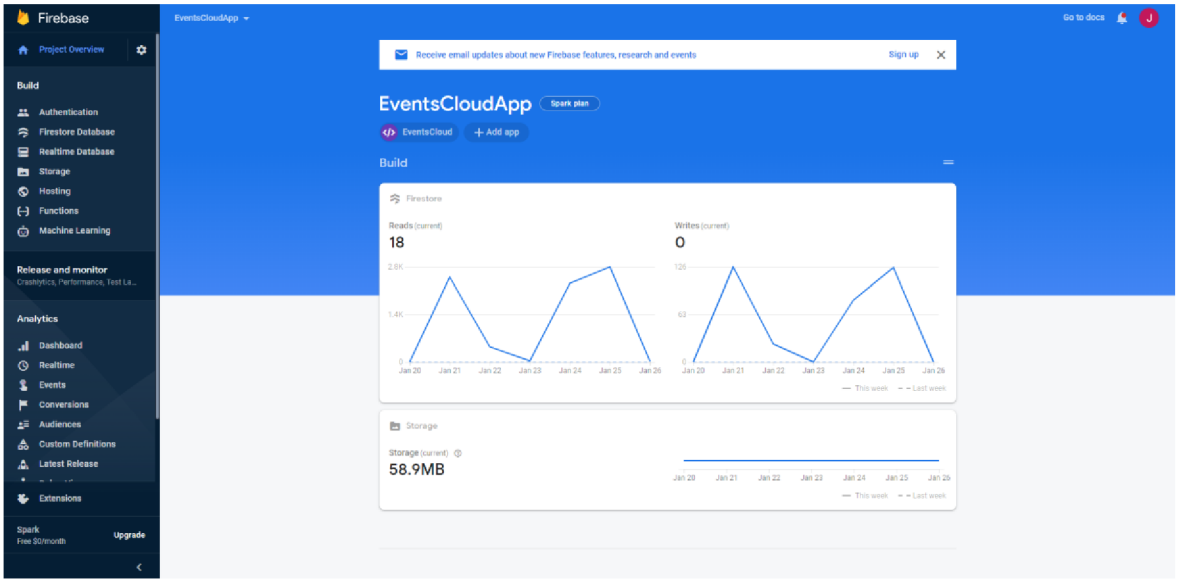
Základní konfigurace aplikace se provádí v souboru app.json, jehož parametry jsou mimo poté dostupné z JavaScript kódu pomocí Constans API dostupné z balíčku Expo SDK. Právě tento soubor použijeme ke konfiguraci našeho loga a Splash obrazovky pomocí definice relativních cest jejich umístění v projektu.

Na obrázku níže je nutno si všimnout specifického nastavení loga a Splash obrazovky pro platformu Android. Logo, jak bylo zmíněno výše, se pro Android konfiguruje pomocí 2 parametrů a Splash obrazovka zvlášť pro každou ze skupin hustot pixelů obrazovek.

Obrázek 18 Konfigurační soubor app.json

4.4.2 Konfigurace databáze Firestore a její propojení s aplikací

Jako dalším nezbytně nutným krokem bude konfigurace, příprava databáze a její propojení s mobilní aplikací. Firestore je cloudová databáze, z čehož tkví řada výhod jako například její dostupnost odkudkoliv z internetu a odpadnutí nutnosti pořizování a údržby vlastního serveru. Začneme tedy vytvořením projektu na stránce https://console.firebase.google.com/, po kterém se dostaneme do konfigurační konzole Firebase platformy. Tato platforma nabízí spoustu užitečných funkcionalit pro analýzu, monitorování a konfiguraci databáze jako například Google analytics, hosting, indexování, programovatelné funkce a tak podobně.



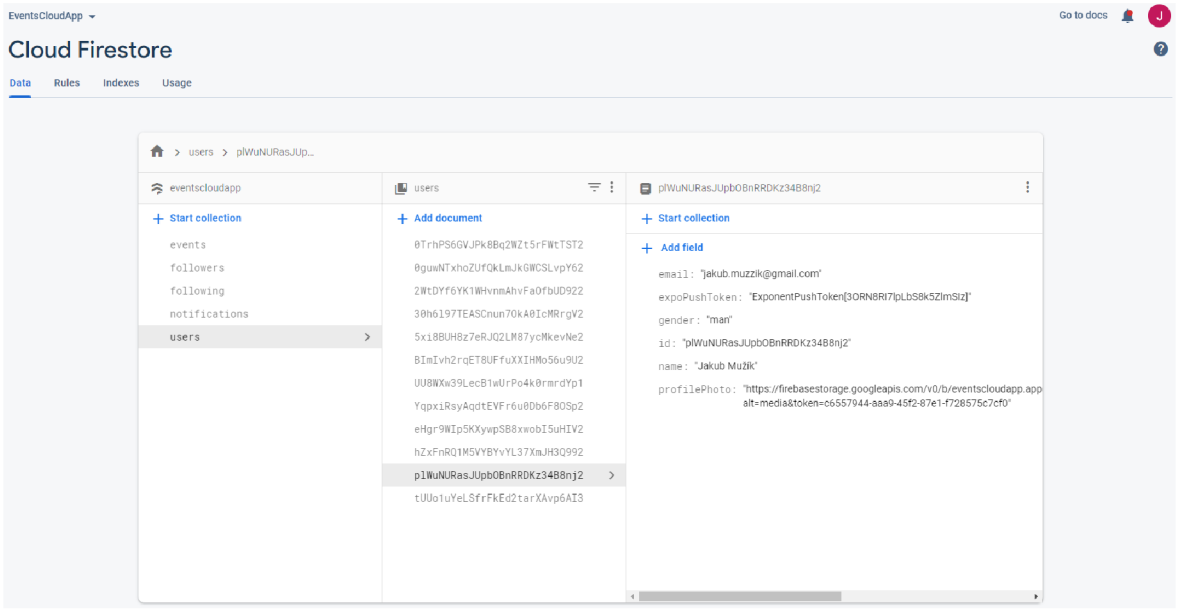
Obrázek 19 Firebase konfigurační konzole

4.4.2.1 Vytvoření Firestore databáze

Jakmile v konzoli inicializujeme Firestore databázi pomocí intuitivního uživatelského rozhraní můžeme začít s její konfigurací. Jak už bylo zmíněno výše, Firestore je No-SQL databáze a data se v ní ukládají ve formátu klíč-hodnota. Z hlediska Firestore terminologie se data strukturují a ukládají do takzvaných kolekcí a jednotlivé záznamy v těchto kolekcích poté představují dokumenty. Každý dokument se skládá z jednotlivých atributů – fieldů, nebo z dalších kolekcí a každý dokument je také identifikován a vytvořen pod jeho unikátním identifikátorem. Tento identifikátor je buď automaticky vygenerován nebo manuálně definován při ukládání do databáze.

Pro účely naší aplikace v databázi vytvoříme následující kolekce:

- Users
- Events
- Notifications
- Following
- Followers



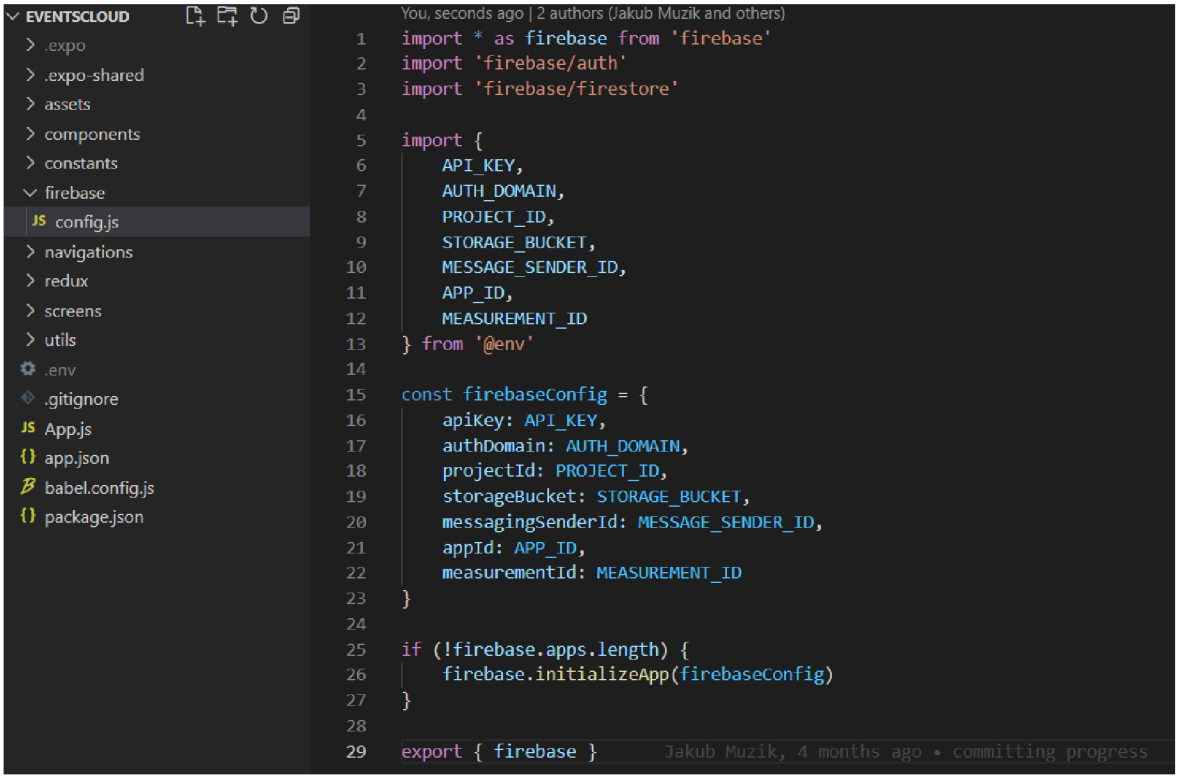
Obrázek 20 Kolekce Firestore databáze

4.4.2.2 Propojení s mobilní aplikací

Dále je potřeba naší aplikaci s touto databází umožnit komunikovat. K tomuto účelu využijeme JavaScript knihovnu Firebase JS SDK, která mimo jiné poskytuje funkce jako jsou registrace, přihlášení, čtení a zápis do databáze. Pro její instalaci v příkazovém řádku v adresáři našeho projektu spustíme příkaz *expo install firebase*.

Aby naše aplikace měla do databáze přístup a mohla s ní komunikovat je potřeba ji při spuštění aplikace inicializovat za pomocí specifických parametrů databáze jako je například API klíč, URL adresa nebo identifikátor projektu. Vzhledem k tomu, že tyto hodnoty poskytují přístup a manipulaci s databází tak jsou považovány za citlivé údaje a měli bychom je tak ve zdrojovém kódu aplikace uložit na bezpečné místo. Pro tento účel využijeme systémových proměnných, kam právě tyto hodnoty uložíme.

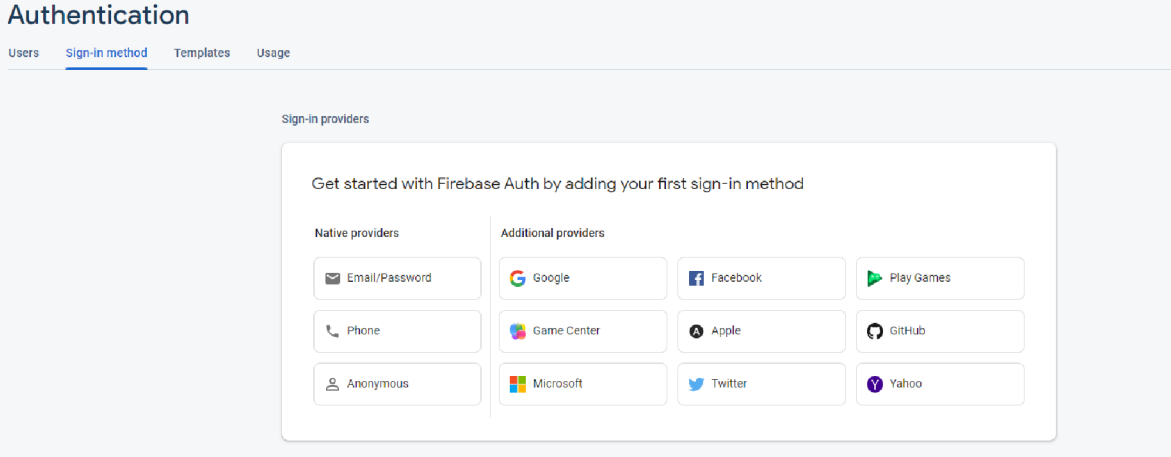
Inicializaci provedeme v separátním JavaScript souboru, který uložíme do nové složky pojmenované firebase. Do tohoto souboru naimportujeme knihovnu Firebase JS SDK, moduly pro autentizaci a Firestore databázi a také parametry databáze ze systémových proměnných. Následně provedeme inicializaci a poté už jen knihovnu exportujeme čímž ji zpřístupníme pro kteroukoliv komponentu z naší aplikace, ve které bude potřeba komunikace s databází.



Obrázek 21 Inicializace Firebase

4.4.2.3 Autentizace

Pro to, aby uživatel mohl pomocí výše uvedené Firebase knihovny komunikovat s databází, musí být přihlášený. Přihlášení, respektive autentizaci budeme implementovat dvojího typu: pomocí dvojice email/heslo a pomocí single sign on za použití Facebookového účtu. Obě tyto přihlašovací metody budeme muset první ve Firebase konzoli povolit. Metodu email/heslo bude stačit v konzoli pouze jednoduše zapnout bez jakékoliv další konfigurace a pro její následné použití už poté jen stačí zavolat příslušné funkce pro registraci a přihlášení dostupné z Firebase knihovny, které implementujeme později v kapitole Implementace. Co se ale týče metody přihlášení pomocí Facebookového účtu, tady budeme při jejím povolení muset také specifikovat parametry Facebookové aplikace splňující funkci Identity providera. Předtím, než tuto metodu tedy povolíme, musíme Facebookovou aplikaci, jakožto Identity providera, vytvořit a naší mobilní aplikaci v ní zaregistrovat.

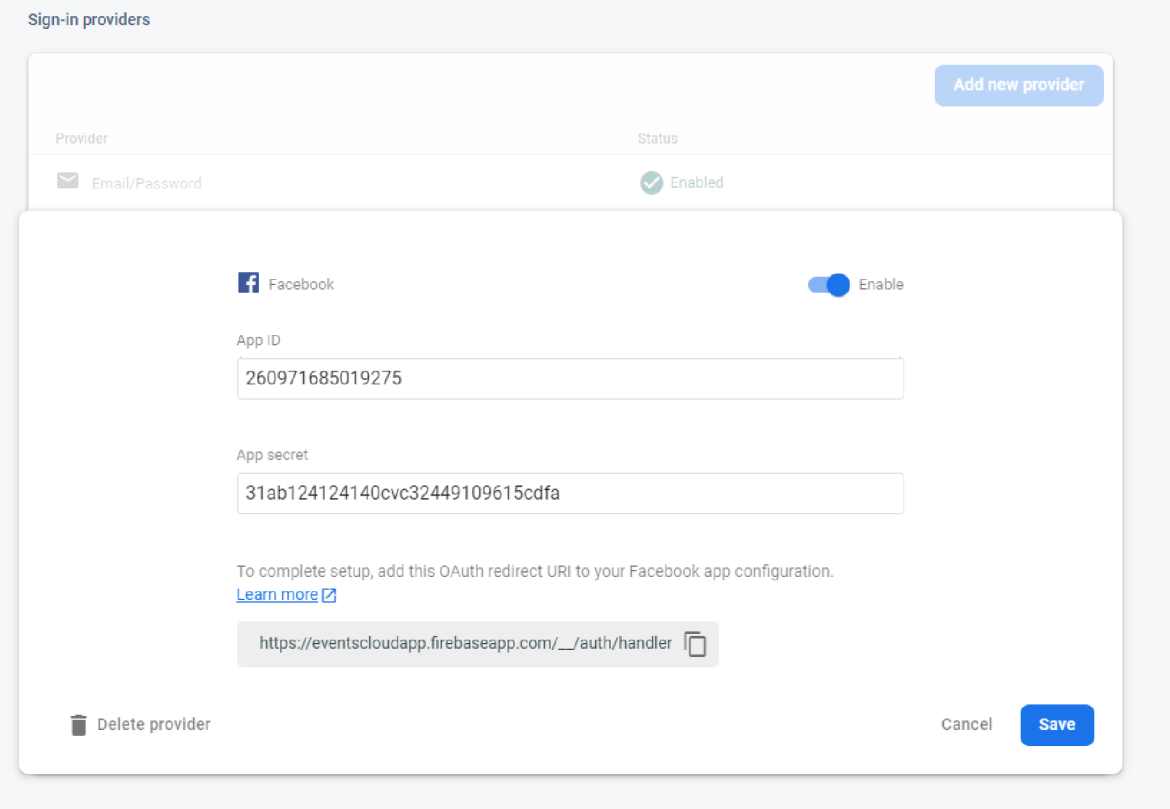


Obrázek 22 Přihlašovací metody do Firebase

4.4.3 Konfigurace Facebook účtu pro SSO

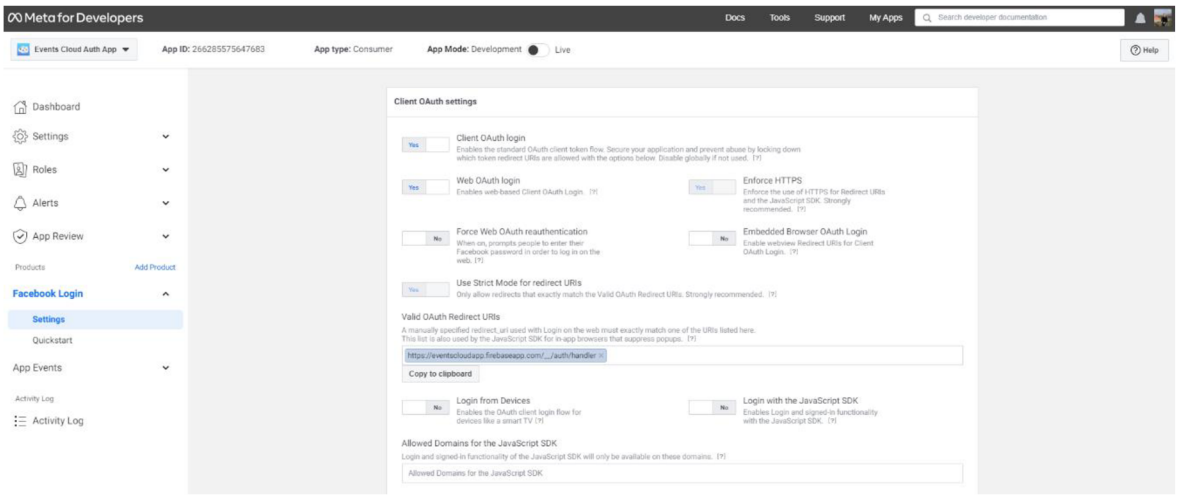
Jako první si na stránce https://developers.facebook.com/ musíme vytvořit vývojářský účet, který nám poskytne přístup k Facebookovým vývojářským balíčkům, API, dokumentaci a tak podobně. K registraci je možné využít běžný, už existující Facebook účet, nicméně pro naše účely bych volil spíše variantu vytvoření zcela nového účtu.

Po registraci, podobně jako tomu bylo u Firebase platformy, vytvoříme novou aplikaci, která nám poskytne parametry jako jsou App ID a App Secret a právě tyto parametry potřebujeme zadat při povolování metody přihlášení pomocí Facebooku ve Firebase konzoli. Hodnotu App ID uložíme také jako systémovou proměnnou v aplikaci, kterou vyžijeme později při implementaci přihlášení. Vrátíme se tedy zpět do Firebase konzole a toto povolení provedeme.



Obrázek 23 Povolení Facebook přihlášení

Pro dokončení této konfigurace už poté ve Facebookové aplikaci potřebujeme jen nakonfigurovat Facebook Login funkcionalitu specifikováním URL adresy, kam má být uživatel po úspěšném přihlášení spolu s přístupovými tokeny přesměrován. Následně je konfigurace hotová a pro využití těchto služeb z mobilní aplikace využijeme expo-facebook knihovny dostupné v Expo aplikaci. Knihovnu nainstalujeme v adresáři našeho projektu pomocí příkazu *expo install expo-facebook*.



Obrázek 24 Konfigurace Facebook Login

Na závěr této konfigurace je nutno ještě podotknout, že aplikace Expo GO, ve které budeme našim mobilní aplikaci spouštět a testovat nepodporuje integraci Facebookového přihlášení se službou zajišťující Firebase autentizaci. Abychom tak tuto funkcionalitu byli schopni otestovat, budeme pro tento případ naší aplikaci muset spustit v mobilním emulátoru vývojového prostředí Android studio.

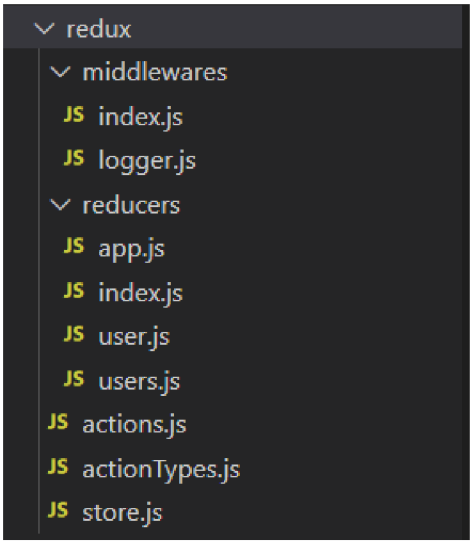
4.4.4 Konfigurace technologie Redux pro správu dat

Předtím, než začneme se základní implementací a přípravou Reduxu ve zdrojovém kódu, budeme muset nainstalovat potřebné knihovny. V naší aplikaci budeme mimo základního Reduxu využívat také pár jeho rozšíření, které budou velice užitečné pro implementaci hned několika funkcionalit. Bude se jednat o rozšíření:

- **React Redux** – velice nám zjednoduší komunikaci mezi komponentami a Redux úložištěm a také zvýší výkon aplikace pomocí optimalizace znovu vykreslování komponent při změnách dat, bez React Redux bychom tyto prvky museli implementovat manuálně
- **Redux persist** – dovolí nám část dat uložit do interní paměti telefonu a zachovat tyto data i po zavření aplikace či vypnutí telefonu, využijeme například před zobrazováním onboarding informací pro kontrolu, zdali uživatel už na svém telefonu aplikaci dříve spustil či nikoliv
- **Redux thunk** – budeme využívat například ve chvíli kdy změna dat v Redux úložišti bude záviset na nějakém asynchronním procesu, například datech získaných pomocí API požadavku

Pro instalaci Reduxu a těchto 3 rozšíření zadáme v příkazové řádce příkaz: *npm install redux react-redux redux-persist redux-thunk*.

Nyní můžeme začít s přípravou Redux struktury ve zdrojovém kódu. Pro tyto účely vytvoříme v kořenovém adresáři novou složku s názvem redux, kde budeme implementovat veškeré související funkcionality.



Obrázek 25 Struktura Reduxu

Middlewares

V našem případě budeme middleware využívat jen při vývoji aplikace, a to pro logovací účely. Tedy soubor logger.js nám bude podávat informace jako je jako jakou akci, respektive změnu dat provádíme, jaký je stav úložiště před změnou a jaký je stav po změně. Tyto informace nám budou užitečně především pro lepší debuggování chyb, které v rámci vývoje budou nastávat.

Reducers

Složka reducers obsahuje 3 reducery, rozdělené podle oblasti jejich použití. Reducer *app.js* se bude starat o data týkající aplikace jako takové. Bude se jednat o různá nastavení aplikace, její stavy anebo například také Boolean hodnotu, pomocí které dokážeme zjistit zdali uživatel aplikaci spouští poprvé nebo už ji spustil dříve. Tento reducer bude také jediný, který pomocí knihovny redux-persist budeme zachovávat.

Dále tu máme reducery *user.js* a *users.js*, které budou obsahovat data týkající se aktuálního uživatele aplikace či uživatelů s kterými bude v rámci aplikace nějakým způsobem interagovat.

Actions a Action Types

Actions budou spolu s Action types obsahovat znovupoužitelné specifické datové změny, které budeme v aplikaci používat. Bude se jednat například o získání informací o uživateli z databáze a jejich uložení do Redux úložiště, nebo naopak propsání změn jeho profilu

do databáze a následná aktualizace Redux úložiště a tak podobně. Pro většinu těchto akcí budeme používat výše zmíněnou knihovnu Redux-thunk.

Store

Na konec ten nejdůležitější soubor *store.js*, ve kterém provedeme inicializaci Redux úložiště spolu s implementací funkcionalit zmíněných výše.

```
redux > JS store.js > default
      Jakub Muzik, 4 months ago | 1 author (Jakub Muzik)
1     import { createStore, applyMiddleware } from "redux"
2     import thunk from 'redux-thunk'
3     import { persistStore, persistReducer } from 'redux-persist'
4     import AsyncStorage from '@react-native-async-storage/async-storage'
5
6     import rootReducer from './reducers'
7
8     import middleWares from "./middlewares"
9
10    const persistConfig = {
11        key: 'root',
12        storage: AsyncStorage,
13        whitelist: ['appState'],
14        blacklist: ['userState']
15    }
16
17    const persistedReducer = persistReducer(persistConfig, rootReducer)
18
19    export default () => {
20        let store = createStore(persistedReducer, applyMiddleware(thunk, ...middleWares))
21        let persistor = persistStore(store)
22        return { store, persistor }
23    }       Jakub Muzik, 4 months ago • committing progress
```

Obrázek 26 Konfigurace Redux úložiště

Ve zdrojovém kódu výše lze vidět, že jako první jsme vytvořili objekt persistConfig, ve kterém pomocí parametrů whitelist a blacklist definujeme, které reducery budeme chtít zachovat a které nikoliv a dále také kam tyto data budeme ukládat – Async Storage. Async Storage poskytuje přístup k nativnímu persistentnímu úložišti telefonu, ze kterého se při znovu otevření aplikace automaticky natáhnout námi uložená a zachovaná data. Tento konfigurační objekt poté použijeme při vytvoření instance persistentního reduceru pomocí metody persistReducer z námi nainstalované knihovny. Následně už jen exportujeme funkci, která po zavolání inicializuje a vrátí Redux úložiště. Tuto funkci poté zavoláme z hlavního souboru aplikace.

```
import initStore from './redux/store'
const { store, persistor } = initStore()
```

Obrázek 27 Inicializace Redux

4.5 Implementace

4.5.1 Onboarding, registrace a přihlášení

Aplikaci se budeme snažit tvořit postupně dle předpokládaného postupu uživatele uživatelským rozhraní. Implementaci tedy začneme tvorbou úvodní části aplikace, respektive jejím onboardingem, registrací a přihlášením. V této podkapitole vytvoříme 3 odpovídající obrazovky, které následně propojíme, abychom uživateli umožnili se mezi nimi pohybovat.

4.5.1.1 Onboarding

Onboarding obecně oproti registraci a přihlášení slouží spíše jako dodatečná funkcionalita pro zvýšení uživatelského prožitku a určitě stojí za to se zamyslet, jestli jeho implementace bude pro aplikaci dávat smysl či nikoliv. Vzhledem ke komplexnosti a řadou funkcionalit, kterými naše aplikace bude disponovat, implementace Onboardingu určitě dává smysl. Onboarding se uživateli zobrazí pouze jednou, a to při prvním spuštění aplikace a bude mít za úkol uživatele ve stručných bodech informovat o obsahu a základních funkcích aplikace. Při zobrazení této obrazovky budeme tedy muset v persistentním *app* reduceru nastavit *isFirstLaunch* parametr, který bude využívám pro určení, jaká obrazovka se má zobrazit jako první.

Onboarding implementujeme pomocí 3 horizontálně listovatelných stránek popisující tyto funkcionality:

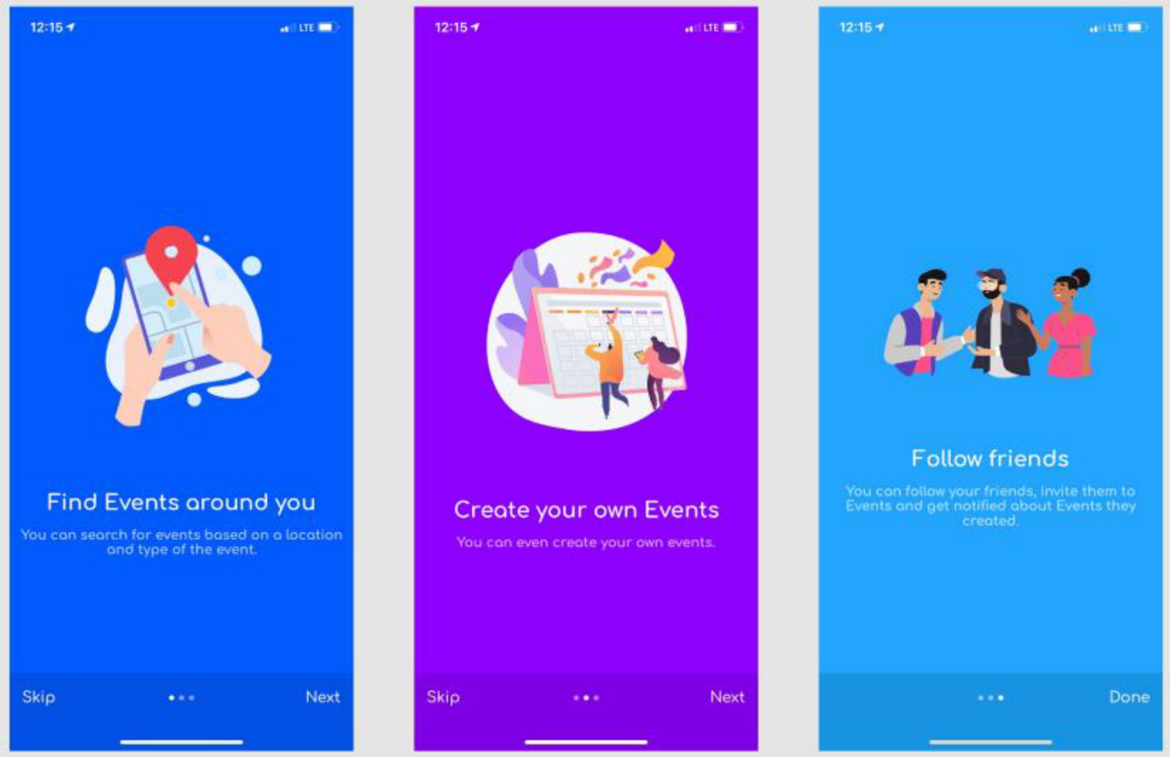
- Nalezení událostí v okolí uživatele
- Vytvoření svých vlastních událostí
- Možnost přidání přátel, pozvání je na události a získávání notifikací

Všechny 3 obrazovky budou obsahovat stručný titulek a jeho odpovídající animaci. Pro zobrazení animací v tomto případě vyžijeme skvělé knihovny Lottie vytvořené společností AirBnB, která na svých stránkách nabízí mnoho zdarma animací dostupných pro použití v mobilních aplikacích. Na stránkách https://lottiefiles.com/ tedy najdeme relevantní a použitelné animace, stáhneme je a uložíme do podsložky animations ve složce assets. Do assets budeme obecně ukládat veškeré statické soubory aplikace. Následně tyto animace už jen

pomocí komponenty LottieView na jednotlivých stránkách zobrazíme a nastavíme tak aby se automaticky spustila a běžela ve smyčce.

```
<LottieView
    style={{width: windowWidth * 0.7}}
    autoPlay
    loop
    source={require('../assets/animations/event.json')}
/>,
```

Obrázek 28 Použití LottieView pro animaci



Obrázek 29 Onboarding

4.5.1.2 Registrace

Registrační stránka bude sloužit pro registraci pomocí emailové adresy. Mimo emailové adresy bude po uživateli také vyžadováno jeho pohlaví, celé jméno a heslo s minimální délkou 8 znaků. Celá obrazovka bude skrolovatelná a budeme muset také ošetřit možnou situaci, která může nastat když uživatel klikne na vstupní pole ve spodní části obrazovky. Po kliknutí na vstupní pole se totiž uživateli zobrazí klávesnice, která ho může překrýt. Pro ošetření použijeme KeyboardAwareScrollView komponentu, která stisknuté pole automaticky posune nad horní hranici klávesnice. Dále bude stránka obsahovat potvrzovací tlačítko, které po stisknutí ověří, zdali se zadaná hesla shodují, obsahují minimální počet znaků, emailová adresa je správném

formátu a nebo už nebyla použita. Pokud bylo vše zadané správně, uživatele pomocí funkce *createUserWithEmailAndPassword* z firebase knihovny zaregistrujeme a pokud vše proběhlo v pořádku, data uživatele uložíme do naší databáze do už dříve vytvořené kolekce *users*. Tyto data budeme ukládat pod unikátním identifikátorem uživatele vráceném registrační funkcí, čímž mimo jiné umožníme jejich pozdější selekci z databáze. Po registraci můžeme informaci o uživateli vidět ve Firebase konzoli jak v sekci *Authentication*, která zobrazuje metodu použitou pro registraci (email/heslo nebo Facebook), emailovou adresu, datum vytvoření a unikátní identifikátor, tak v sekci Firestore databáze v kolekci *users*.

```
firebase.auth().createUserWithEmailAndPassword(email, password)
    .then((response) => {
        firebase.firestore().collection('users')
            .doc(response.user.uid)
            .set({
                id: response.user.uid,
                name,
                email,
                gender,
                createdDate: new Date(),
                createdEvents: [],
                likedEvents: [],
                attendedEvents: []
            })
            .catch((error) => {
                Alert.alert(error)
            });
    })
    .catch(error => {
        if (error.code === 'auth/email-already-in-use') {
            Alert.alert('That email address is already in use!')
        }

        if (error.code === 'auth/invalid-email') {
            Alert.alert('That email address is invalid!')
        }
    })
```

Obrázek 30 Registrace pomocí emailu a hesla

4.5.1.3 Přihlášení

Přihlašovací stránka bude poskytovat přihlášení buď pomocí emailu a hesla, nebo pomocí Facebookového účtu. Začneme s implementací pomocí emailu a hesla, která se bude z hlediska implementace podobat předchozímu případu. Na obrazovce vytvoříme 2 vstupní pole pro email a heslo, přihlašovací tlačítko a ošetříme zobrazování klávesnice stejným způsobem jako na předchozí stránce. Po stisknutí tohoto tlačítka už jen zavoláme

signInWithEmailAndPassword funkci z firebase knihovny a pokud uživatel zadal správná data, bude přesměrován do hlavní části aplikace.

Pro přihlášení pomocí Facebooku vytvoříme ve spodu obrazovky druhé tlačítko, které už z hlediska implementace bude značně složitější než u předchozích případů. Po stisknutí tlačítka budeme jako první muset uživatele přesměrovat do Facebookové aplikace, ve kterém po něm po úspěšném přihlášení vyžádáme povolení k přístupu k jeho emailové adrese, informacím na profilu a jeho kamarádům. Toho docílíme pomocí *logInWithReadPermissionsAsyc* funkce z Facebook knihovny, které jako parametr předáme pole obsahující oprávnění, které budeme po uživateli požadovat.

```
await Facebook.initializeAsync({appId: FACEBOOK_APP_ID})

const { type, token, permissions, declinedPermissions } = await Facebook.logInWithReadPermissionsAsync({
  permissions: ['public_profile', 'email', 'user_friends'],
})
```

Obrázek 31 Facebook přihlášení

Na úryvku kódu výše je možno vidět, že jako první jsme museli Facebookovou aplikaci také inicializovat pomocí parametru App ID, který jsme dříve, při konfiguraci aplikace, uložili jako systémovou proměnnou. Funkce poté vrátí proměnné – type, token, permissions a declinedPermissions. Proměnná type obsahuje informaci o úspěchu naší operace, token představuje přístupový token, který můžeme použít například pro volání Facebook API a permissions spolu s declinedPermissions obsahují oprávnění, které uživatel povolil či zamítl.

Pokud tedy ve Facebook aplikaci došlo k úspěšnému přihlášení a povolení k přístupu k osobním informacím, musíme uživatele přihlásit také do naší databáze. K tomu nám poslouží přístupový token, vrácený z předešlé funkce a opět relevantní funkce ve firebase knihovně.

```
const credential = firebase.auth.FacebookAuthProvider.credential(token)
const user = await firebase.auth().signInWithCredential(credential)
```

Obrázek 32 Facebook přístupový token

Funkce *signInWithCredential* mimo přihlášení do databáze také vrátí objekt obsahující řadu informací o samotném uživateli, použité metody přihlášení a podobně. Na základě těchto informací tedy uložíme uživatele, stejně jako tomu bylo u registrace pomocí emailu, do kolekce users v naší databázi. Dále budeme chtít, aby se uživateli ve chvíli, kdy se přihlásil poprvé, zobrazil seznam jeho Facebookových přátel, který se do naší aplikace již dříve zaregistrovali.

Informaci, zdali se uživatel přihlásil poprvé, obsahuje parametr *isNewUser* dostupném v právě vráceném objektu. Pokud tomu tak tedy je a pokud zároveň uživatel ve Facebookové aplikaci povolil přístup k informacím o jeho přátelích, zavoláme za použití přístupového tokenu Facebook API pro získání seznamu těchto přátel a později, až bude uživatel přesměrován do domovské stránky aplikace, je zobrazíme v modálním oknu.

```
const response
    = await fetch(`https://graph.facebook.com/me/friends?fields=id,name,picture&access_token=${currentUser.token}`)
const { data } = await response.json()
```

Obrázek 33 Požadavek pro získání Facebookových přátel

4.5.1.4 Propojení obrazovek pomocí navigace

Nyní tyto 3 vytvořené obrazovky musíme propojit takovým způsobem, aby se mezi nimi uživatel mohl pohybovat. Uživatel bude po konci Onboardingu automaticky přesměrován do přihlašovací stránky, ze které se poté bude moci přesouvat do registrační stránky a zpátky. Pro tento případ nám bude sloužit Stack navigátor, který vytvoříme v novém souboru *AuthStack.js*, ve složce navigations. Při definici navigace a jejích obrazovek musíme také pomocí parametru *initialRouteName* určit, jaká obrazovka se má zobrazit jako první. To bude záležet na tom, zdali uživatel spouští aplikaci poprvé, či nikoliv. Pokud ano, jako první zobrazíme obrazovku pro Onboarding a pokud ne zobrazíme rovnou obrazovku pro přihlášení. Jak už bylo zmíněno výše, tato informace bude dostupná z persistentního *app* reduceru, kterou z úložiště dostaneme pomocí Redux funkce connect a poté předáme jako parametr komponenty.

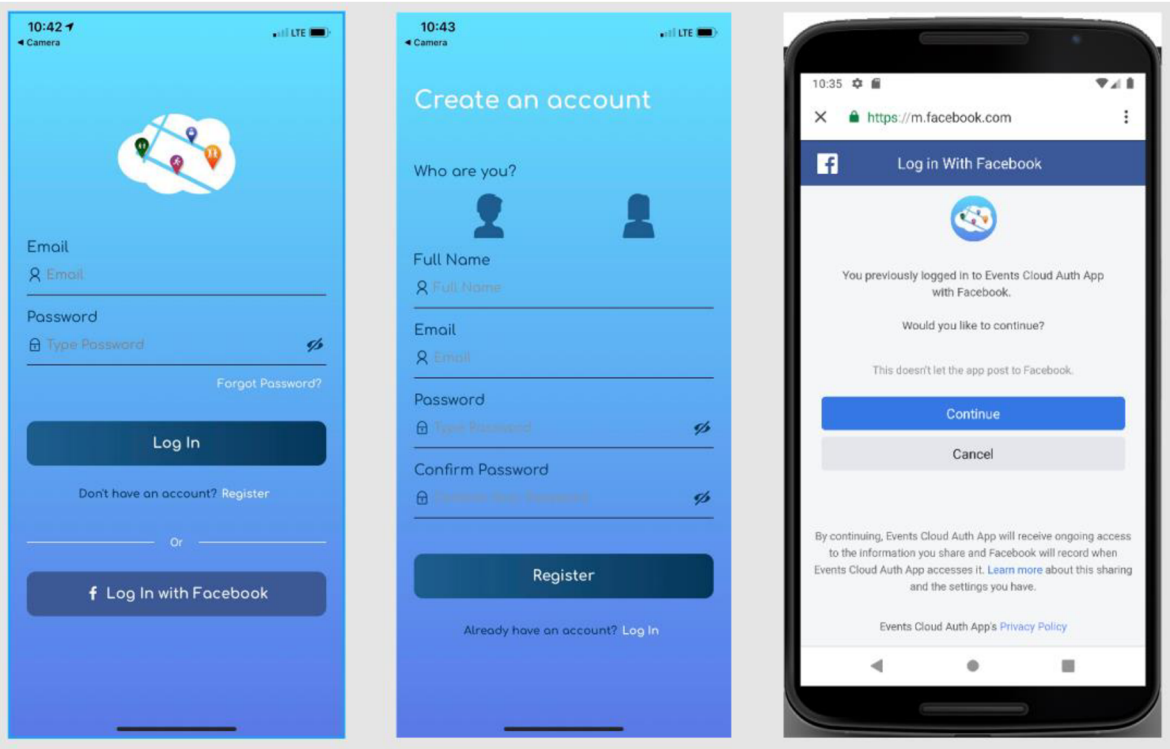
```
const AuthStack = ({ isFirstLaunch }) => {
    return (
        <Stack.Navigator initialRouteName={isFirstLaunch ? 'OnboardingScreen' : 'SignInScreen'}>
            <Stack.Screen
                name="OnboardingScreen"
                component={OnboardingScreen}
                options={{ headerShown: false }}
            />
            <Stack.Screen
                name="SignInScreen"
                component={SignInScreen}
                options={{ headerShown: false }}
            />
            <Stack.Screen
                name="SignUpScreen"
                component={SignUpScreen}
                options={{ headerShown: false }}
            />
        </Stack.Navigator>
    )
}

const mapStateToProps = (store) => ({
    isFirstLaunch: store.appState.isFirstLaunch
})

export default connect(mapStateToProps)(AuthStack)
```

Obrázek 34 Autentizační Stack navigace

V tuto chvíli máme hotové 3 úvodní obrazovky a jejich propojení pomocí navigace. Nyní tuto celou Stack navigaci vložíme do další, nadřazené a hlavní Stack navigace aplikace umístěnou v kořenovém *App.js* souboru, která bude propojovat tento autentizační Stack se zbytkem aplikace a na základě toho, jestli je uživatel přihlášen či nikoliv, budeme zobrazovat její příslušnou část. K tomuto účelu nám poslouží *onAuthStateChanged* event listener, který pokaždé, změní-li se stav přihlášení uživatele v databázi, zavolá funkci, kterou jsme mu předali jako parametr. Pokud se tedy uživatel přihlásí, funkce pomocí reference navigačního kontejneru uživatele automaticky přesměruje z přihlašovací stránky do hlavní části aplikace. Naopak, pokud se uživatel odhlásí, bude automaticky zpět přesměrován do autentizační části.

```
useEffect(() => {
  const unsubscribe = firebase.auth().onAuthStateChanged(user => {
    if (!user) {
      navigationRef.current?.dispatch(StackActions.replace('AuthStack'))
    } else {      Jakub Muzik, 4 months ago • committing progress
      navigationRef.current?.dispatch(StackActions.replace('Main'))
    }
  })

  return () => unsubscribe()
}, [])

return (
  <NavigationContainer ref={navigationRef}>
    <Stack.Navigator
      initialRouteName={loggedInOnInit ? 'Main' : 'AuthStack'}
      screenOptions={{
        headerShown: false
      }}
    >
      <Stack.Screen
        name="AuthStack"
        component={AuthStack}
      />
      <Stack.Screen
        name="Main"
        component={Main}
      />
    </Stack.Navigator>
  </NavigationContainer>
)
```

Obrázek 35 Autentizační listener



Obrázek 36 Přihlašovací a registrační stránka

4.5.2 Hlavní část aplikace

V tuto chvíli máme hotovou autentizační část aplikace, respektive uživatel se může registrovat a přihlásit a nyní tedy budeme pokračovat s implementací hlavní a obsahové části aplikace. Opět se budu snažit implementaci provádět postupně dle posloupnosti průchodu uživatele uživatelským rozhraním.

4.5.2.1 Inicializace

Jak jsme viděli v předchozí podkapitole, jakmile se uživatel přihlásí, funkce v autentizačním event listeneru přesune uživatele do komponenty *Main*. Tuto komponentu tedy budeme považovat jako kořenovou komponentu hlavní části aplikace, a právě zde budeme pokračovat s dalším vývojem. Toto místo také obsahuje kód, který se bude spouštět jako první po otevření hlavní části aplikace, čehož využijeme a implementujeme zde náležité následující funkcionality.

Jako první budeme muset získat z databáze data o aktuálně přihlášeném uživateli a uložit je do Redux úložiště. Data z databáze získáme pomocí unikátního identifikátoru přihlášeného uživatele a bude se jednat o data jako je například jméno, profilová fotka nebo pohlaví. Pro tento účel vytvoříme *fetchEvents* funkci v námi již připravené Redux složce *actions*. Dále jelikož budeme manipulovat s Redux úložištěm, budeme také muset vytvořit konstantu určující typ této manipulace, pomocí které se bude později *reducer* rozhodovat jakým způsobem změnu do Redux úložiště propagovat. Konstantu pojmenujeme USER_STATE_CHANGE a použijeme ji tedy jak při volání změny dat o aktuálním uživateli ze složky *actions*, tak v reduceru *user* obsahující logiku a propagaci těchto dat do úložiště. Reducer poté uživatelská data uloží do parametru *currentUser* a pomocí JavaScript spread operátoru zkopíruje a zachová zbytek úložiště.

```
export const fetchUser = () => async (dispatch) => (
    firebase.firestore()
        .collection('users')
        .doc(firebase.auth().currentUser.uid)
        .get()
        .then((snapshot) => {
            if (snapshot.exists) {
                dispatch({ type: USER_STATE_CHANGE, currentUser: snapshot.data() })
            } else {
                dispatch({ type: USER_STATE_CHANGE, currentUser: null })
            }
        })
)
```

Obrázek 37 Získání dat uživatele z databáze

```
export const user = (state = INITIAL_STATE, action) => {
    switch (action.type) {
        case USER_STATE_CHANGE:
            return {
                ...state,
                currentUser: action.currentUser ? action.currentUser : INITIAL_STATE
            }
        default:
            return state;
    }
}
```

Obrázek 38 Propagace uživatelských dat do Redux

Dále budeme potřebovat po otevření aplikace z databáze také dostat data jako jsou koho přihlášený uživatel sleduje, kdo sleduje jeho a také jeho notifikace. Zároveň pokud v těchto datech, respektive v kolekcích *notifications, following a followers* dojde k nějakým změnám, budeme chtít abychom o nich byly v aplikaci v reálném čase notifikováni. Toto je jedna z funkcionalit, která databáze Firestore nabízí a také jeden z důvodů, proč jsme si pro náš projekt Firestore vybrali. Tato funkcionalita se provádí pomocí takzvané subskripce na danou databázovou kolekci a my tyto subskripce provedeme právě v *Main* komponentě po otevření aplikace. Subskripce se provádí pomocí onSnapshot listeneru a vložením funkce, která se bude volat pokaždé, pokud v dané kolekci došlo k nějaké datové změně. Zároveň bude tato funkce přijímat parametr obsahující změněná data z dané kolekce. V našem případě tedy provedeme tuto subskripci pro každou ze 3 zmíněných kolekcí a uvnitř funkce, která se bude volat při každé změně, implementujeme propagaci změněných dat do Redux úložiště. Pro všechny 3 kolekce ale subskripce budeme muset vytvořit na jejich další, vnořené kolekce patřící příslušnému uživateli. Datová struktura je totiž v těchto kolekcích následovná (uvedeno na příkladu pro kolekci followers): Kolekce *followers -> dokument vytvořený pod unikátním identifikátorem uživatele -> vnořená kolekce userFollowers -> seznam uživatelů, které uživatele sledují.* Pomocí této struktury poté dokážeme získat námi požadovaná data pro konkrétního uživatele pomocí jeho unikátního identifikátoru.

```
export const fetchUserFollowers = () => (dispatch) => {
    return firebase.firestore()
        .collection("followers")
        .doc(firebase.auth().currentUser.uid)
        .collection("userFollowers")
        .onSnapshot((snapshot) => {
            let followers = snapshot.docs.map(doc => {
                const id = doc.id
                return id
            })

            dispatch({ type: USER_FOLLOWERS_STATE_CHANGE, followers })
        })
}
```

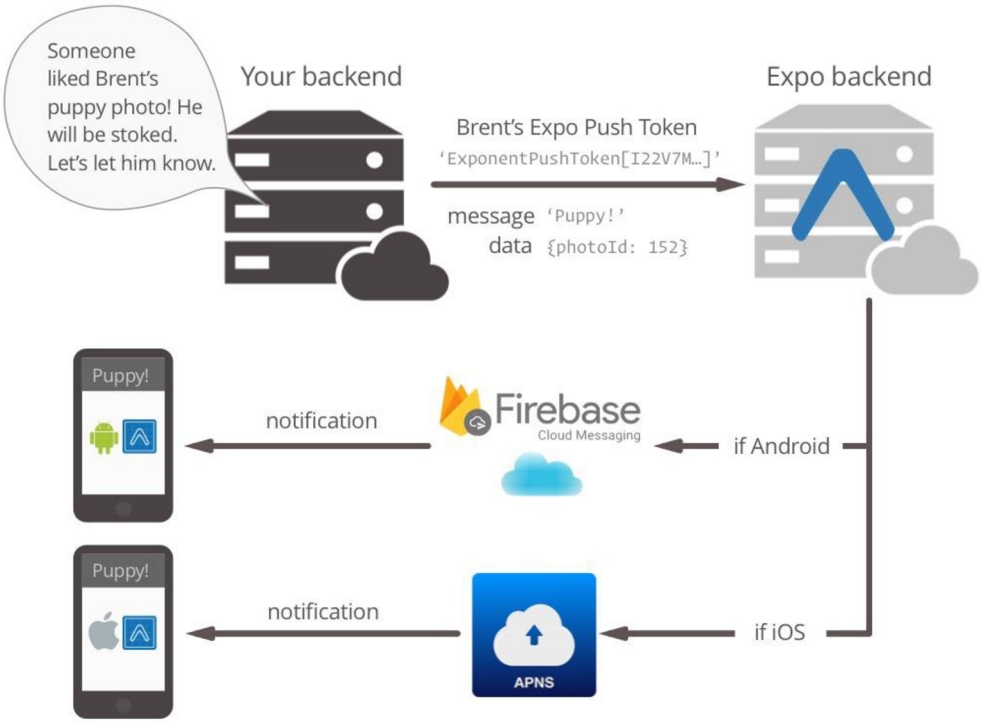
Obrázek 39 Vytvoření subskripce na databázovou kolekci

Dalšími daty, které budeme chtít po otevření aplikace dostat z databáze, jsou události. V tomto případě budeme chtít získat nejen události, které jsme sami vytvořili ale i události od ostatních uživatelů. Námi vytvořené události budeme chtít všechny. V případě událostí od ostatních uživatelů budeme chtít pouze ty, které mají začátek konání v rozmezí 14 dní. Dále také pro soukromé události budeme ještě muset navíc zkontrolovat, jestli jsme na ně byli pozvaní. Tyto získaná data poté opět uložíme do Redux úložiště – námi vytvořené události do proměnné *createdEvents* a zbytek do proměnné *events*.

4.5.2.2 Implementace notifikací

Jako poslední věc, kterou musíme v této inicializační části aplikace implementovat, je získání Expo push tokenu a vytvoření listenerů pro notifikace. Push token je unikátní klíč identifikující nainstalovanou aplikaci v konkrétním mobilním zařízení, do kterého budeme notifikace chtít posílat. Při standartním vývoji mobilních aplikací jsou tyto notifikace doručovány skrze notifikační brány: *Firebase Cloud Messaging* pro Android a *Apple Push Notification service* pro iOS zařízení, kde obě tyto brány poskytují webové API pro poslání požadovaných notifikací skrze HTTP požadavku a za použití nativního Push tokenu. V tomto případě bychom se tedy pro posílání notifikací museli vypořádat s rozdílností mezi iOS a Android tokeny a vytvořit 2 různé integrace pro každou z notifikačních brán v závislosti na použitém tokenu. Implementace notifikací pomocí přímé komunikace s notifikačními bránami je zcela jistě správný a smysluplný způsob, nicméně pro naši aplikaci použijeme trochu jiný postup. V našem případě pro konfiguraci a posílání notifikací použijeme Expo modul. Tento modul poskytuje vlastní notifikační bránu – *Expo push notifications service*, která se za nás

automaticky vypořádá s rozdílností platforem, a to automatickým přeposláním notifikace do příslušené nativní notifikační brány.



Obrázek 40 Expo notifikační brána

Jak je vidět na obrázku výše, notifikace budeme místo nativních iOS a Android tokenů posílat za použití Expo Push Tokenu. Tento token tedy budeme muset pro každého našeho uživatele vygenerovat a následně uložit do databáze, aby k němu ostatní uživatelé měli přístup a při posílání notifikací využít. Tento token se může pro obě mobilní platformy lišit, a tak token vygeneruje a uložíme pokaždé když se uživatel přihlásí. Tím docílíme toho, že v databázi budeme mít pokaždé token odpovídající aktuálnímu používanému zařízení cílového uživatele. Pro generování tokenu využijeme metody *getExpoPushTokenAsync* z Expo knihovny *Notifications*, ale ještě před tím, než token vygenerujeme a uložíme do databáze, budeme muset zkontrolovat jestli uživatel aplikaci povolil zasílání notifikací a pokud ne, požádáme ho o povolení. Pro tento případ použijeme metody *getPermissionsAsync* a *requestPermissionAsync* dostupné ze stejné knihovny. Jakmile máme od uživatele souhlas, vygenerujeme token a uložíme ho do databáze. Tokeny budeme ukládat do dokumentů uživatelů v kolekci *users* do pole *expoPushToken*.

```
const { status: existingStatus } = await Notifications.getPermissionsAsync()
let finalStatus = existingStatus
if (existingStatus !== 'granted') {
    const { status } = await Notifications.requestPermissionsAsync()
    finalStatus = status
}
if (finalStatus !== 'granted') {
    return
}
token = (await Notifications.getExpoPushTokenAsync()).data

firebase.firestore()
    .collection('users')
    .doc(firebase.auth().currentUser.uid)
    .update({
        expoPushToken: token
    })
```

Obrázek 41 Získání Expo push tokenu

V tuto chvíli máme v databázi k dispozici tokeny jednotlivých uživatelů a nyní tedy vytvoříme funkci pro posílání notifikací. V této funkci notifikaci nejdříve uložíme do databáze do kolekce *notifications* pod každého cílového uživatele. A jelikož jsme právě na tuto kolekci v přechozí části vytvořili subskripci, uživatelům se tato notifikace v aplikaci v reálném čase uloží a zobrazí jako nová položka seznamu na obrazovce *Notifications*. Z hlediska dat budeme pro notifikace ukládat: datum vytvoření, text, obrázek, typ, id záznamu (kam uživatele přesměrovat po kliknutí na notifikaci) a pokud se jedná o notifikaci týkající se události, tak také její typ.

Dále už nám jen zbývá tyto notifikace odeslat do Expo notifikační brány pomocí HTTP post požadavku. Jako tělo tohoto požadavku budeme posílat pole, ve kterém každá z jeho položek bude obsahovat následující parametry: body (text notifikace), data (přenášená data) a Expo push token. Po odeslání naše data Expo notifikační brána zpracuje, pomocí tokenu přepošle do odpovídající notifikační brány platformy a pokud vše proběhne v pořádku, notifikace se uživateli automaticky zobrazí v jeho telefonu.

```
const expoPushMessages = data.map(d => ({
  body: d.body,
  data: JSON.stringify({
    screen: d.type === 'follow' ? 'UserProfile' : 'EventScreen',
    id: d.redirect
  }),
  to: d.token
}))

fetch('https://exp.host/--/api/v2/push/send', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(expoPushMessages)
})
```

Obrázek 42 Posílání notifikací

Poslední, s čím se zde musíme vypořádat je jak se má aplikace zachovat, pokud uživatel notifikaci stiskne. V tomto případě budeme chtít, aby se uživatel přesměroval na konkrétní záznam týkající se notifikace. Konkrétně se bude jednat buď o událost nebo profil uživatele, který nás začal sledovat. K tomuto nám poslouží opět *Notifications* knihovna a její *addNotificationResponseReceivedListener* listener. Pomocí tohoto listeneru totiž můžeme definovat funkci, který se spustí pokaždé když uživatel stiskne notifikaci. Navíc tato funkce obdrží parametr, ze kterého můžeme získat data přenášená v notifikaci. Z předchozího úryvku kódu si lze všimnout, že tyto data obsahují právě jméno obrazovky, kam se má uživatel přesměrovat spolu s identifikátorem zobrazovaného záznamu. Tento listener vytvoříme opět, jako v předchozích případech, v kořenové komponentě *Main*.

```
responseListener.current = Notifications.addNotificationResponseReceivedListener(response => {
    const { notification: { request: { content: { data: { screen, id } } } } } = response
    switch (screen) {
        case "UserProfile":
            return navigation.navigate(screen, { user: {id} })
        case "EventScreen":
            return navigation.push(screen, { eventId: id })
        default:
            return
    }
})
```

Obrázek 43 Notifikační listener

Tímto bychom měli hotovou inicializační část aplikace, připravené všechny data, zprovozněné notifikace a nyní se tedy můžeme pustit do implementace dílčích obrazovek aplikace a jejich funkcionalit.

4.5.2.3 Profil

Jako první obrazovku, kterou budeme implementovat bude Profil. Profil nám bude sloužit jak k zobrazení a editaci našeho profilu, tak k zobrazení profilu ostatních uživatelů. Základní struktura stránky bude tedy pro oba případy stejná, nicméně zde bude také několik odlišností v závislosti na tom, koho profil si prohlížíme. Pokud si budeme prohlížet profil jiného uživatele, bude zde navíc tlačítko pro možnost začít uživatele "sledovat" a také seznam událostí, které uživatel navštívil naposledy. V případě zobrazení našeho vlastního profilu, bude místo možnosti pro sledování a seznamu událostí k dispozici tlačítko k jeho editaci. V obou případech bude profil obsahovat profilovou fotku, jméno uživatele a počet sledujících a sledovaných. Abychom mohli takto podmíněně zobrazovat určité prvky na obrazovce, musíme vědět čí profil prohlížíme. Toto zjistíme v inicializační části životního cyklu komponenty, kde zkontrolujeme, jestli při navigaci do této obrazovky nebyl předám parametr obsahující data jiného uživatele. Pokud ano, budeme vědět, že prohlížíme profil někoho jiného a na základě toho, budeme zobrazovat anebo schovávat některé prvky na obrazovce.

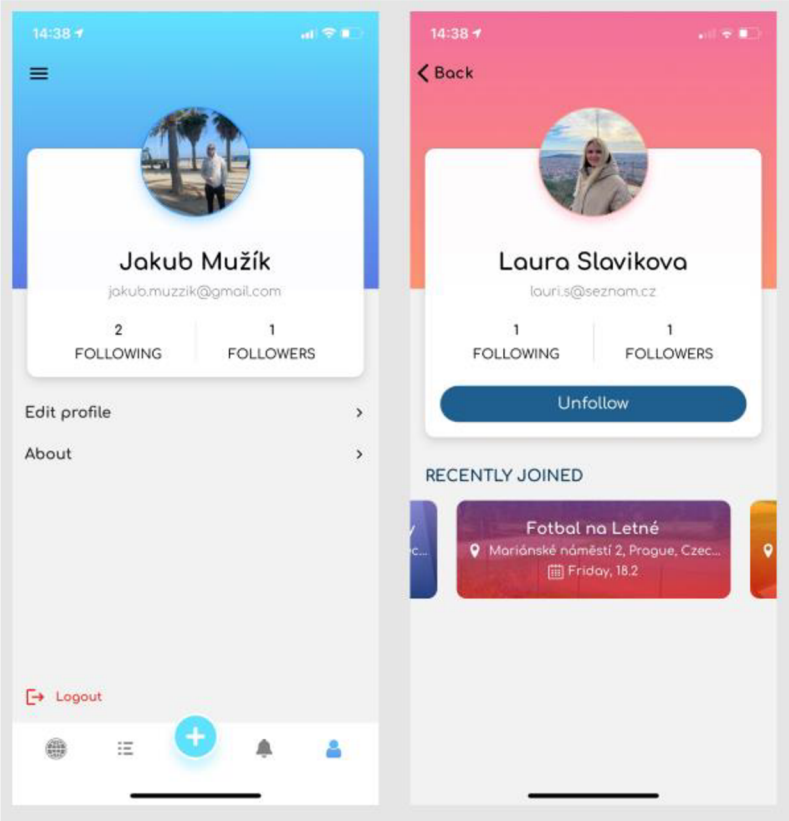
Mimo standartní tvorbu a podmíněné zobrazování prvků uživatelského rozhraní zde budeme muset také umožnit editaci profilové fotky, a to buď pomocí výběru již existujícího obrázku z galerie telefonu nebo vyfocením nové fotky. Tato editace bude umožněna po stisknutí profilové fotky v konfigurační obrazovce profilu a následné volby způsobu změny fotky. Pro oba případy, respektive otevření kamery nebo galerie telefonu využijeme knihovnu *expo-image-picker*. V případě volby vyfocení nové fotky, budeme muset nejdříve, pokud jsme tak již neprovedli, požádat o přístup ke kameře telefonu. Toto provedeme pomocí funkce *requestCameraPermissionsAsync* dostupné ve zmíněné knihovně. Pokud uživatel přístup povolil, otevřeme kameru telefonu pomocí funkce *launchCameraAync* a vyfocenou fotku v podobě URI uložíme do proměnné pro následné potvrzení a uložení změn v profilu. Podobně tomu tak bude i při výběru fotky z galerie telefonu. Nejdříve uživatele pomocí funkce *requestMediaLibraryPermissionsAsync* požádáme o přístup ke galerii a pokud přístup povolil, otevřeme galerii funkcí *launchImageLibraryAsync* a opět uložíme URI zvolené fotky. Poměr stran vybrané fotky v obou případech nastavíme na 1:1 a snížíme kvalitu o 40 %. Poměr stran 1:1 se bude lépe zobrazovat v kruhovém avataru zobrazující profilovou fotku a snížení kvality

fotografie nám sníží velikost použitého úložného prostoru v databázi, a tedy i náklady na její použití.

```
const pickImage = async () => {
    let permissionResult = await ImagePicker.requestMediaLibraryPermissionsAsync()

    if (permissionResult.granted === false) {
        Alert.alert("Permission to access your gallery is required!")
        return
    }

    let result = await ImagePicker.launchImageLibraryAsync({
        mediaTypes: ImagePicker.MediaTypeOptions.Images,
        allowsEditing: true,
        aspect: [1, 1],
        quality: 0.6
    })

    if (!result.cancelled) {
        try {
            setUser({
                ...user,
                profilePhotoUri: result.uri
            })
            editPhotoSheetRef.current.close()
        } catch (e) {
            console.error(e)
        }
    }
}
```

Obrázek 44 Výběr fotky z galerie telefonu

Jakmile uživatel pomocí jednoho ze zmíněných způsobů vybere novou fotku a potvrdí tuto změnu, budeme ji muset uložit do databáze a zpřístupnit tak ostatním uživatelům. Fotku tedy nejdříve budeme muset nahrát do Firebase cloud úložiště do podsložky pojmenované unikátním identifikátorem uživatele a získat tak její stáhnutelnou URL adresu. Tuto adresu už následně jen uložíme do databáze do záznamu uživatele a příště, až si někdo zobrazí náš profil, nová fotka se pomocí URL adresy z úložiště stáhne a zobrazí.



Obrázek 45 Obrazovky profilů

4.5.2.4 Tvorba události

Pro tvorbu událostí vytvoříme Modal obrazovku, která bude dostupná po stisknutí "plus" tlačítka uprostřed spodního menu aplikace. Tato obrazovku bude sloužit jak pro tvorbu, tak i pro následnou úpravu událostí. Obrazovka bude představovat formulář, kam uživatel zadá veškerá data popisující událost a poté uloží do databáze a zpřístupní tak ostatním uživatelům. V rámci této práce a minimálního životaschopného produktu se bude jednat o data:

- Typ události
- Jméno
- Datum a čas začátku
- Datum a čas konce
- Místo konání
- Popis
- Zvolení viditelnosti události (veřejná / soukromá)
- Výběr seznamu přátel pro odeslání pozvánky
- Přidání fotky

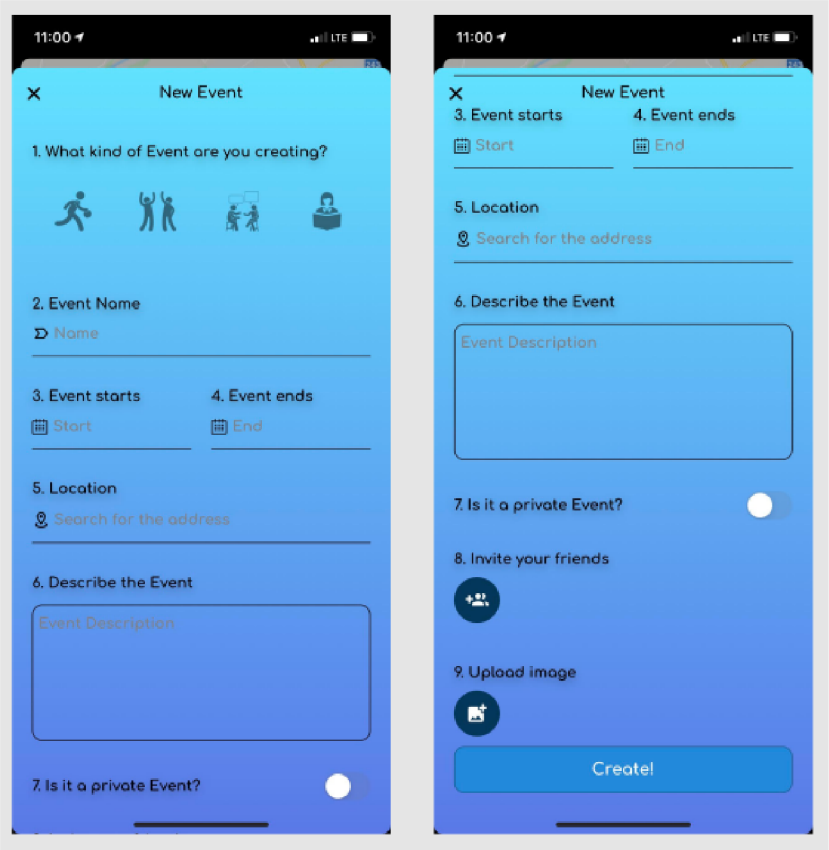
Stěžejním bodem při tvorbě události bude především určení místa konání. Při jejím určení totiž budeme muset znát její zeměpisné souřadnice abychom ji později mohli zobrazit na mapě. Pro tento případ nám poslouží knihovna *expo-location*, která poskytuje 2 zásadní funkce:

- *geocodeAsync* – na základě textového vstupu nalezne adresy a vrátí jejich zeměpisné souřadnice
- *reverseGeocodeAsync* –na základě zeměpisných souřadnic vrátí odpovídající adresu

Pro implementaci tohoto požadavku tedy vytvoříme vstupní pole, kam uživatel první zadá adresu konání jeho události. Při tomto zadávání pomocí funkce *geocodeAsync* budeme hledat adresy odpovídající vstupu uživatele a získáme tak jejich zeměpisné souřadnice. Současně pomocí těchto souřadnic a funkce *reveseGeocodeAsync* zjistíme jejich plné znění a nabídneme je jako výsledek hledání. Každý tento nabídnutý výsledek bude tedy obsahovat jak plné znění adresy, tak její zeměpisné souřadnice a jakmile uživatel jeden z výsledků vybere, obě tyto hodnoty uložíme.

```
const onSearchLocation = async (search) => {
    if (!search || search.length < 2) {
        setResults([])
        return
    }

    let geoLocations = await Location.geocodeAsync(search)

    const addresses = []
    for (let geoLocation of geoLocations) {
        const { latitude, longitude } = geoLocation
        let address = await Location.reverseGeocodeAsync({ latitude, longitude })
        addresses.push({ geoLocation: { latitude, longitude }, address: address[0] })
    }

    setResults(addresses)
}
```

Obrázek 46 Vyhledávání adres



Obrázek 47 Obrazovka pro tvorbu událostí

4.5.2.5 Mapa událostí

Obrazovka s mapou událostí bude představovat hlavní a stěžejní obrazovku naší aplikace. Tato mapa bude velice komplexní, a proto její implementaci rozdělíme do několika bodů dle funkcionalit, kterými bude disponovat:

- Zobrazení událostí na mapě v podobě mapových značek
- Navigace do aktuální polohy uživatele
- Filtrování dle datumu a typu události
- Zobrazení horizontálně skrolovatelných detailů aktuálně zobrazených událostí na mapě

Zobrazení událostí na mapě v podobě mapových značek

Jako první budeme muset vůbec nainstalovat *react-native-maps* knihovnu, která poskytuje konfigurovatelnou mapu od Google Maps. Z této knihovny poté budeme importovat 2 stěžejní komponenty – MapView pro zobrazení a interakce s mapou a Marker pro zobrazení mapových značek. Pro zobrazení mapových značek na mapě musíme komponentu Marker

vložit jako child do komponenty MapView a pomocí parametrů longitude (zeměpisná délka) a latitude (zeměpisná šířka) definovat její umístění. Pro tento případ nám tedy bude stačit uvnitř komponenty MapView iterovat přes pole obsahující události a pro každé z nich vrátit komponentu Marker s hodnotami příslušné události.

```
<MapView
    ref={mapRef}
    style={styles.map}
    showsUserLocation={true}
    initialRegion={INITIAL_REGION}
    animationEnabled={true}
    provider={MapView.PROVIDER_GOOGLE}
>
    {events.map((event, index) => (
        <Marker
            key={index}
            coordinate={{
                latitude: event.latitude,
                longitude: event.longitude
            }}
        />
    ))}
</MapView>
```

Obrázek 48 Zobrazení mapových značek

Mohlo by se zdát, že pro tento požadavek bychom měli hotovo. Nicméně pokud by pole *events* obsahovalo příliš velké množství událostí, respektive pokud bychom vykreslovali příliš mnoho mapových značek najednou, mohlo by to mít negativní dopad na výkonnost a mapa by se mohla začít sekat. Abychom tomuto předešli, budeme muset v danou chvíli vykreslovat pouze ty mapové značky, které jsou v aktuálně, uživatelem zobrazovaném prostoru. Pro splnění tohoto požadavku budeme tedy muset po každém pohybu na mapě přepočítat, které mapové značky máme zobrazit. K tomu využijeme callback *onRegionChangeComplete* na MapView komponentě, který se zavolá po každém ukončení pohybu na mapě. Tento callback totiž obsahuje hodnoty popisující aktuálně zobrazovaný prostor na mapě. A to pomocí parametrů: longitude, latitude, longitudeDelta a latitudeDelta, kde longitudeDelta představuje délku mezi levým a pravým okrajem a latitudeDelta zase délku mezi horním a spodním okrajem mapy. Skrze tyto parametry už poté jednoduchým výpočtem dokážeme vypočítat souřadnice každého z rohů zobrazovaného prostoru na mapě.

```
const onRegionChangeComplete = (region) => {
    boundingBox.current = {
        westLng: region.longitude - region.longitudeDelta / 2, // westLng - min lng
        southLat: region.latitude - region.latitudeDelta / 2, // southLat - min lat
        eastLng: region.longitude + region.longitudeDelta / 2, // eastLng - max lng
        northLat: region.latitude + region.latitudeDelta / 2 // northLat - max lat
    }

    let filteredEvents = events.filter(e => isInBoudingBox(e.geoLocation))

    setVisibleEvents(filteredEvents)
}

const isInBoudingBox = (geoLocation) => {
    if (geoLocation.latitude > boundingBox.current.southLat && geoLocation.latitude < boundingBox.current.northLat &&
        geoLocation.longitude > boundingBox.current.westLng && geoLocation.longitude < boundingBox.current.eastLng) {
        return true
    }

    return false
}
```

Obrázek 49 Filtrace viditelných mapových značek

Jak je vidět na úryvku kódu výše, vypočítané souřadnice rohů uložíme do proměnné boundingBox a poté už jen pomocí JavaScript funkce *filter* vyfiltrujeme události, které události se vyskytují v námi zobrazovaném prostoru. Tyto události uložíme do nové proměnné *visibleEvents*, kterou posléze použijeme při vykreslování mapových značek.

Navigace do aktuální polohy uživatele

Obecně pro využívaní lokalizačních služeb telefonu budeme využívat *expo-location* knihovnu a její Location komponentu. Jako první, než je ale začneme využívat, musíme zkontrolovat, jestli jsou tyto služby v telefonu uživatele vůbec povoleny. Pokud nejsou, místo mapy zobrazíme hlášku, která požádá o jejich povolení. Kontrolu provedeme v inicializační části životního cyklu komponenty pomocí funkce *hasServicesEnabled* dostupné z *expo-location* knihovny.

Pro navigaci uživatele do jeho aktuální polohy bude sloužit absolutně umístěné tlačítko ve spodní části obrazovky. Po jeho stisknutí budeme muset opět zkontrolovat povolení polohových služeb. Tentokrát se ale bude jednat o polohové služby konkrétně pro naši aplikaci, nikoliv pro mobilní zařízení obecně. Tato kontrola se provede pomocí funkce *requestForegroundPermissionsAsync* volané ze stejné knihovny jako v přechozím případě a pokud byla aplikace v telefonu otevřena poprvé, uživateli se zobrazí nativní modální okno žádající ho o jejich povolení. Pokud uživatel služby nepovolil, nic se nestane a uživatel nikam přesměrován nebude. Pokud ano, nejdříve pomocí funkce *getCurrentPositionAsync* získáme souřadnice uživatelovi polohy a poté je využijeme jako parametry při volání funkce

animateToRegion dostupné na MapView komponentě. Tato funkce uživatele automaticky přesměruje do jeho aktuální lokace.

```
const navigateToCurrentLocation = async () => {
    let permissionResult = await Location.requestForegroundPermissionsAsync()

    if (permissionResult.granted === false) {
        return
    }

    let { coords } = await Location.getCurrentPositionAsync()
    if (coords) {
        const { latitude, longitude } = coords
        mapRef.current.animateToRegion({
            latitude,
            longitude,
            latitudeDelta: LOCATION_LATITUDE_DELTA,
            longitudeDelta: LOCATION_LONGITUDE_DELTA
        }, 500)
    }
}
```

Obrázek 50 Navigace do polohy uživatele

Filtrování dle datumu a typu události

V neposlední řadě také rozhodně budeme chtít umožnit uživateli události filtrovat, a to podle jejich typu a datumu konání. Filtrování dle datumu bude umožněno pomocí *Chip* komponenty v hlavičce obrazovky, kdy po jejím stisknutí se uživateli zobrazí modální okno nabízející volbu datumu. Toto datum bude možné zvolit v rozmezí 14 dnů a bude se uplatňovat na celou dobu konání události. Modální okno pro výběr datumu implementujeme pomocí komponenty *DateTimePickerModal* z knihovny *react-native-modal-datetime-picker* a jejího *isVisible* atributu pro podmíněné zobrazení. Filtrování dle typu událostí bude přístupné po stisknutí ikony v pravé horní části obrazovky. Její stisknutí vysune Drawer navigaci umístěnou na pravé straně obrazovky a poskytne zvolení typů událostí, které chceme zobrazit. Pro vytvoření Drawer navigace budeme muset vytvořit novou komponentu, do které obrazovku s mapou vložíme jako child komponentu. Naše obrazovka poté automaticky obdrží *navigation* atribut poskytující řadu funkcí pro ovládání nadřazené navigace. Pro otevření Drawer navigace budeme používat funkci *openDrawer* a pro její umístění *drawerPosition* atribut s hodnotou nastavenou na „right“.

```
return (
    <Drawer.Navigator
        screenOptions={{drawerPosition: "right", headerShown: false}}
        drawerContent={props => <ExploreFilters {...props} />}
    >
        <Drawer.Screen name="ExploreScreenStack" component={ExploreScreenStack} />
    </Drawer.Navigator>
)
```

Obrázek 51 Definice pravé Drawer navigace

Zobrazení horizontálně skrolovatelných detailů aktuálně zobrazených událostí na mapě

Aby se uživatel mohl dozvědět, jakou událost daná mapová značka představuje, tak po jejím stisknutí zobrazíme ve spodní části obrazovky kartu, obsahující stručný popis odpovídající události. Tato karta bude součástí horizontálně skrolovatelného listu, který bude navíc obsahovat také detaily všech na mapě viditelných událostí. Při jeho skrolováním budou mapové značky aktuálně prohlížených karet automaticky zvýrazňovány pomocí jejich zvětšení.

Jako první tedy budeme muset vytvořit list, který bude ve výchozím stavu schovaný a jakmile na mapě stiskneme nějakou mapovou značku, list se zobrazí a automaticky vycentruje na kartu právě stisknuté mapové značky. List opět schováme ve chvíli, kdy se uživatel na mapě buď posune nebo stiskne prostor mimo mapové značky. List budeme zobrazovat a schovávat standartní React Native knihovnou – *Animated,* která pomocí animace poskytuje možnost přesouvat prvky z bodu a do bodu b. Náš list bude tedy absolutně umístěn a pomocí parametru *translateY* bude přesouván po ose y z hodnoty 500 (bod dostatečně nízko pod obrazovkou kde bude list schovaný) do hodnoty -20 (místo nad spodní hranou obrazovky). Pokud bychom ale pro přesun listu používali pouze tyto 2 body, list by mezi nimi pouze přeskakoval bez jakékoliv plynulosti. Abychom list mohli přesouvat plynule, budeme muset nějakým způsobem určit trasu, respektive všechny body, po kterých se má list přesouvat. Tyto body budeme vypočítávat pomocí funkce *interpolate*, která použitím lineární interpolace vypočítá zbylé hodnoty mezi body v rozsahu 500 a -20. Funkce obecně přijímá 2 parametry – rozsah vstupních a výstupních hodnot. Rozsah vstupních hodnot bude v našem případě 0 a 1 (0 – list je schovaný, 1 – list je zobrazený) a rozsah výstupních hodnot bude 500 a -20 (odpovídající umístění listu na ose y). Tuto funkci poté budeme volat na animační proměnné, jejíž hodnotu budeme přepínat mezi hodnotami 0 na 1. A právě při tomto přepínání, bude funkce *interpolate* v reálném čase vracet odpovídající hodnoty z definovaného výstupního rozsahu, které budeme ukládat do proměnné *scrollViewY* a využijeme pro přesun listu po ose y.

```
const scrollViewMode = useRef(new Animated.Value(0)).current

const showScrollViewAnim = () => {
    Animated.spring(scrollViewMode, {
        toValue: 1,
        useNativeDriver: false
    }).start()
}

const scrollViewY = scrollViewMode.interpolate({
    inputRange: [0, 1],
    outputRange: [500, normalize(-20)]
})

return (
    <Animated.ScrollView
        horizontal
        style={[styles.scrollView, { transform: [{ translateY: scrollViewY }] }]}
    ></Animated.ScrollView>
)
```

Obrázek 52 Animace horizontálního listu

Zobrazování a schovávání listu bychom tedy měli hotové a nyní už nám jen zbývá vyřešit automatické posouvání příslušných karet v listu na střed poté, co uživatel stiskne některou z mapových značek. Pro toto posouvání využijeme metodu *scrollTo,* která komponenta *ScrollView* nabízí. Tato metoda umožňuje posunutí se v listu do námi určeného bodu. Poté, co tedy uživatel stiskne nějakou mapovou značku, budeme muset vypočítat, kde na ose x se její příslušná karta v listu vyskytuje a toto číslo poté zadat jako parametr x při volání zmíněné *scrollTo* funkce. Tuto pozici budeme moci vypočítat po uvědomění, jakým způsobem jsou na ose x položky seznamu usazeny. Zpravidla první položka se na ose x nachází v bodě 0 a následující položka je vždy v bodě položky předchozí + její šířka. Abychom tedy mohli pro stisknuté mapové značky vypočítat jejich pozici na ose x, budeme muset znát jejich index, respektive pozici v listu a také mít definovanou pevnou šířku karty. Pro definici pozice mapových značek využijeme standartní *index* atribut dostupný při iteraci jednotlivých událostí JavaScript metodou *map*, která následně předáme jako parametr při volání *onPress* funkce.

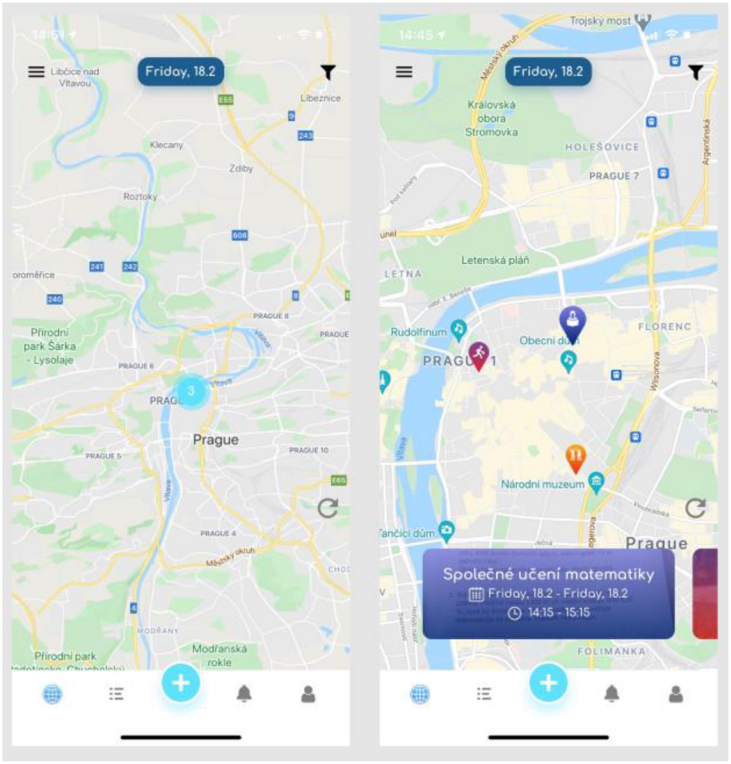
```
{visibleEvents.map((event, index) => (
    <Marker
        key={index}
        coordinate={{
            latitude: event.latitude,
            longitude: event.longitude
        }}
        onMarkerPress={() => onMarkerPress(index)}
    />
))}
```

Obrázek 53 Indexování mapových značek

Funkce *onMarkerPress* už poté bude jen obsahovat logiku popsanou výše pro výpočet pozice karty a její zobrazení v horizontálně skrolovatelném listu.

```
const onMarkerPress = (index) => {
    showScrollViewAnim()

    let x = index * (CARD_WIDTH)
    scrollViewRef.current.scrollTo({ x: x, y: 0, animated: true })
}
```

Obrázek 54 Výpočet pozice zobrazované karty



Obrázek 55 Zobrazení událostí na mapě

5 Výsledky a diskuse

Při různých diskusích s ostatními vývojáři mobilních aplikací, ať už na webu nebo s kolegy v práci, velmi často narážím na jednu z hlavních otázek – jakým způsobem React Native aplikace tvořit. Konkrétně se tedy jedná o rozpor mezi vývojem pomocí React Native CLI anebo pomocí aplikace třetí strany Expo. Osobně si myslím, že nelze jednoznačně určit, který z těchto dvou způsobů je lepší a vždy bude záležet na tom, co po naší aplikaci požadujeme a jestli nás omezení vycházející ze zvoleného způsobu nijak zásadně nelimitují. Jestliže tomu tak je, rozhodně bych se přikláněl pro volbu aplikace Expo, a to už jen z důvodu, že i kdybychom v průběhu vývoje narazili na nějaký zásadní problém, je vždy možné se z této aplikace třetí strany odpojit a pokračovat se standartním React Native CLI.

Jinak tomu ani nebylo v našem případě, respektive jelikož všechny naše požadované funkcionality Expo umožňovalo, rozhodli jsme se pro jeho volbu. A pokud se nyní zpětně ohlédnu na průběh implementace, rozhodně toho nelituji. Nejen z důvodu, že jsem, co se týče obecného vývoje mobilních aplikací spíše začátečník a ulehčení vývoje díky Expo jsem velice ocenil, ale také hlavně z důvodu skvělé možnosti aplikaci testovat na reálných mobilních zařízeních prostřednictvím nainstalované Expo GO aplikace. Pro Windows uživatele je toto mimochodem jedna z největších výhod oproti vývoji pomocí React Native CLI, jelikož při jeho použití je možné aplikaci testovat pouze v mobilních emulátorech a vývojové prostředí poskytující iOS emulátor je dostupné pouze na Macu. Pokud jsme se tedy rozhodli pro React Native CLI a nedisponujeme Macem, nemáme možnost aplikaci na iOS zařízení otestovat. Mimo tyto získané výhody pro nás ale volba aplikace Expo nese také jednu nevýhodu, a to o trochu větší konečná velikost aplikace z důvodu několika nadstandartních API a služeb, kterými aplikace vyvinuté pomocí Expo disponují.

Kompletní zdrojový kód aplikace je dostupný ve veřejném gitovém repositáři na adrese https://github.com/jakubmuzzik/EventsCloud.

6 Závěr

Jedním z cílů této práce bylo seznámit čtenáře s multiplatformním frameworkem React Native, dvěma možnými způsoby pro jeho použití a také se souvisejícími technologiemi týkající se vývoje mobilních aplikací za použití tohoto frameworku. Všechny tyto technologie a způsoby byly popsány v teoretické části práce, a tak můžeme tento cíl požadovat za splněný.

Druhým a současně hlavním cílem práce bylo vytvořit mobilní aplikaci fungující na platformách iOS a Android za použití frameworku React Native a zároveň tak čtenáře seznámit s kompletním procesem vývoje mobilní aplikace od analýzy po její konečnou implementaci. Tento cíl byl splněn v hlavní části práce, kde jsme si na jejím začátku pomocí výpisu několika funkcionalit definovali minimální životaschopný produkt, který byl následně dle logické posloupnosti průchodu uživatele uživatelským prostředím implementován. Pro celý vývoj a konfiguraci prostředí a technologií spojených s tvorbou aplikace pomocí React Native jsem využíval znalostí získaných jak samostudiem odborné literatury, tak zkušeností získaných mým zaměstnáním na pozici Salesforce Developera. Definice minimálního životaschopného produktu a jeho implementace také odpovídá vizi autora – vytvořit v rámci této diplomové práce funkční základ mobilní aplikace, na které bude později dále pracovat a rozšiřovat o doplňující funkcionality až do doby, kdy bude aplikace hodná jejímu nahrání do obchodů Google Play a App Store. Jako možné další rozšíření aplikace by mohlo být například přidání možnosti události více konkretizovat z hlediska jejich zaměření. Pro například sportovně zaměřené události by tak mohla být přidána možnost výběru o jaký sport se bude jednat a později, kdy si uživatel bude události filtrovat, tak bude moci lépe a přesněji uplatňovat jeho preference. Zároveň by mohlo být uživateli umožněno si tyto preference nějakým způsobem uložit do profilu a později, až budou vytvořeny události splňující tyto preference, o nich být notifikován.

Na závěr bych chtěl také zmínit vděčnost pro umožnění volby takto tematicky zaměřené diplomové práce, jelikož mě tato problematika baví a její tvorbou jsem získal mnoho zkušeností a poznatků, které bez pochyby uplatním jak v profesním, tak osobním životě.

7 Seznam použitých zdrojů

1. Smarty. What is HTTP? [online]. [cit. 2022-02-10]. Dostupné z WWW: https://smartystreets.com/articles/what-is-http
2. IBM. Application Progamming Interface (API) [online]. 2020-08-19 [cit. 2022-02-16]. Dostupné z WWW: https://www.ibm.com/cloud/learn/api
3. HAVERBEKE, Marijn, 2019. Eloquent JavaScript, third edition. San Francisco: no starch press. ISBN 978-1-59327-950-9.
4. C-sharpcorner. What and Why React.js [online]. 2021-02-10 [cit. 2022-02-17]. Dostupné z WWW: https://www.c-sharpcorner.com/article/what-and-why-reactjs/
5. React: The Virtual DOM [online]. [cit. 2022-02-17]. Dostupné z WWW: https://www.codecademy.com/article/react-virtual-dom
6. MVC: Mode, View, Controller [online]. [cit. 2022-02-17]. Dostupné z WWW: https://www.codecademy.com/article/mvc
7. Introducing JSX [online]. [cit. 2022-02-17]. Dostupné z WWW: https://reactjs.org/docs/introducing-jsx.html
8. Redux Fundamentals, Part 1: Redux Overview [online]. 2021-06-25 [cit. 2022-02-17]. Dostupné z WWW: https://redux.js.org/tutorials/fundamentals/part-1-overview
9. What is React Native? [online]. [cit. 2022-02-17]. Dostupné z WWW: https://www.netguru.com/glossary/react-native
10. React Native – Základy React Native [online]. [cit. 2022-02-17]. Dostupné z WWW: https://www.itnetwork.cz/javascript/react/native/react-native-zaklady-react-native
11. Expo Vs React Native CLI [online]. 2020-07-03 [cit. 2022-03-02]. Dostupné z WWW: https://www.linkedin.com/pulse/expo-vs-react-native-cli-prakhar-nagpal
12. Introduction to Expo [online]. [cit. 2022-03-02]. Dostupné z WWW: https://docs.expo.dev/
13. What is Firebase? [online]. [cit. 2022-03-04]. Dostupné z WWW: https://firebasetutorials.com/what-is-firebase/
14. Introduction to Firebase [online]. 2017-12-27 [cit. 2022-03-04] Dostupné z WWW: https://hackernoon.com/introduction-to-firebase-218a23186cd7
15. How to use Firebase Storage [online]. 2020-02-20 [cit. 2022-03-08]. Dostupné z WWW: https://firebasetutorials.com/use-firebase-storage/
16. Introducing JSON [online]. [cit. 2022-03-08]. Dostupné z WWW: https://www.json.org/json-en.html

17. What is NoSQL? [online]. [cit. 2022-03-08]. Dostupné z WWW: https://www.mongodb.com/nosql-explained