Juhani Lammi

Moderni mobiilisovelluskehitys ja serverless-arkkitehtuuri

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Ohjelmistotekniikka

Insinöörityö

7.5.2018

Tekijä Otsikko Sivumäärä Aika	Juhani Lammi Moderni mobiilisovelluskehitys ja serverless-arkkitehtuuri 37 sivua + 5 liitettä 7.5.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikka
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaajat	Lehtori Juha Kämäri Lehtori Jussi Alhorinne

Insinöörityön tarkoituksena oli perehtyä mobiilisovelluskehityksen erilaisiin menetelmiin ja toteuttaa sovelluskonsepti käyttämällä yhtä näistä teknologioista sekä tutkia sovelluksen taustapalveluiden vaihtoehtoisia toteutusmenetelmiä. Tarkoitus oli myös arvioida näitä menetelmien ja teknologioiden vaikutus sovelluskehityksen resurssivaatimuksiin.

Mobiilisovelluksella voidaan tavoittaa suuria määriä ihmisiä ja ihmisryhmiä. Tämän vuoksi mobiilisovellusten tarve on kasvanut ja johtanut erilaisten kehysten ja menetelmien syntyyn. Näillä kehyksillä pyritään pienentämään eri alustoille toteutettavan sovelluskehityksen resurssivaatimuksia ja nopeuttamaan kehitykseen vaadittavaa aikaa.

Mobiilisovelluksia voidaan toteuttaa web-pohjaisesti, erilaisilla monialustaisilla sovelluskehyksillä tai alustan omilla menetelmillä. Näistä tarkempaan tutkailuun valittiin uusi sovelluskehys React Native. React Native on Facebookin kehittämä mobiilisovelluskehys, joka mahdollistaa kehityksen molemmille suosituimmille alustoille, iOS:lle ja Androidille.

Nykypäiväiset mobiilisovellukset myös monesti hyödyntävät taustapalveluita, jonka avulla voidaan toteuttaa monenlaisia verkkoyhteyttä vaativia toiminnallisuuksia. Erilaiset pilvipalvelut ovat nousseet perinteisten palvelimien rinnalle ja tarjoavat kilpailukykyisiä palveluita. Yksi tällainen on Firebase, joka tarjoaa laajat palvelut mobiilisovellusten käyttöön aina analytiikasta tietokantaan.

Lopuksi työssä toteutetaan prototyyppiversio sovelluksesta käyttäen edellä mainittuja teknologioita. Itse sovellus on paikannukseen ja yhteisöllisyyteen perustuva sovellus, jossa käyttäjät voivat merkitä ja arvioida paikkoja kartalla. Tätä ideaa voidaan hyödyntää monissa erilaisissa käyttötapauksissa kuten teattereissa, myymälöissa tai vaikka prototyyppiin valituissa yleisissä käymälöissä.

Avainsanat	React Native, React, Firebase, serverless, mobiilikehitys, JavaScript, MobX, Android, iOS, mobiilisovelluskehys

Author Title Number of Pages Date	Juhani Lammi Modern mobile app development and serverless architecture 37 pages + 5 appendices 7 May 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Technology
Instructors	Juha Kämäri, Lecturer

The purpose of this theses to research current mobile app development methods, to implement an application with one of the said methods, and to research various backend services available for mobile app development. The intent was also to evaluate these technologies and their impact in resourcing of app development.

The modern day mobile application can reach out to large amounts of people or groups of people. This has led to increased demand for mobile applications and has led to invention multiple new frameworks and development methods. These new innovations try to lower the resource requirements of mobile app development and to lower the time required to develop a working mobile app.

Mobile applications can be developed in multiple different ways which include web-based application, cross-platform frameworks and by using the technologies and methods intended by the platform itself. React Native was selected for a closer look at for this thesis. React Native is a cross-platform mobile development framework created by Facebook. It allows developing for apps for the two most used platforms, iOS and Android.

Often time modern mobile apps also take advantage of a backend service, to allow more advanced features that require connection to the internet. Recently the traditional server and application has gotten a competitor in the form of cloud services and they offer very competitive features. One of these cloud service providers is Firebase that offers a large variety of features from analytics to real-time database.

In the end of the thesis we implement a prototype of an application using these technologies. The application itself is location and social based application in which users can add places to a map and review them. This application idea can be used in various cases like movie theaters, shops or as was selected for the prototype, public restrooms.

Keywords	React Native, React, Firebase, serverless, mobile development, JavaScript, MobX, Android, iOS, mobile development framework

Sisällys

1 Johdanto 1

2. Mobiilisovellusten kehitysalustat 2

2.1 Natiivi 2

2.1.1 Android 2

2.1.2 iOS 3

2.2 Progressiiviset web-sovellukset 3

2.3 Monialustaiset sovelluskehykset 4

2.3.1 HTML5 -pohjaiset sovelluskehykset 4

2.3.2 Natiivit sovelluskehykset 4

3. Mobiilisovelluksen taustapalvelut 5

3.1 Palvelinarkkitehtuuri 5

3.2 Serverless -arkkitehtuuri 6

4. React Native -sovelluskehitys 8

4.1 React 8

4.2 Tilan hallinta 11

4.3 React Native 14

4.4 Työkalut ja menetelmät 16

5. Esimerkkisovellus: Hotelli helpotus 18

5.1 Projektin arkkitehtuuri 19

5.1.1 Tietovarastot ja tilanhallinta 21

5.1.2 Sovellus 24

6. Tulokset 32

Lähteet 34

Liite 1: Säiliökaavio 37

Liite 2: Tilavarastokavio 39

Liite 3: Käyttöliittymän rakenne 39

Liite 4: PlaceStoren paikkojenhakufunktio 40

Liite 5: MergeAndSave -funktio 41

Lyhenteet

RN	React Native. Facebookin kehittämä mobiilisovellusten sovelluskehys.
JS	JavaScript. Suosittu, alun perin web-kehitykseen tarkoitettu ohjelmointikieli.
SDK	Software development kit. Lajitelma kirjastoja, jotka mahdollistavat sovelluskehityksen jollekin alustalle.
IDE	Intelligent development environment. Ohjelma tai joukko ohjelmia, jotka mahdollistavat ohjelmistokehityksen.
PWA	Progressive web app. Progressiivinen web sovellus. Web-tekniikoita hyödyntävä, mobiilioptimoitu sivu.
REST	Representational state transfer. Arkkitehtuurityyli hajautettujen järjestelmien toteuttamiseen.
CRUD	Create, read, update, delete. Lyhenne jolla kuvataan tietokantaan tehtäviä operaatioita.
JSON	JavaScript object notation. Yksinkertainen tiedostomuoto tiedonvälitykseen verkkoprotokollien yli.
SQL	Structured Query Language. Tietokantojen yleisin kyselykieli.
NPM	Node package manager. Kirjastojen jakeluun tarkoitettu ekosysteemi.
UI	User interface. Käyttöliittymä.
RTDB	Real-time database. Reaaliaikainen tietokanta, jota Firebase käyttää.

1 Johdanto

Tämän insinöörityön tarkoituksen on tutustua mobiilisovellusten kehityksen teknologioihin ja tutustua lähemmin sovelluskehys React Nativeen ja pilvipalvelu Firebaseen. Aluksi käydään läpi, mitä eri tapoja on tarjolla sovelluksen kehittämiseen ja minkälaisia taustapalveluita mobiilisovelluksille on tarjolla.

Mobiililaitteet ja niihin liittyvät teknologiat kehittyvät kovaa vauhtia, ja ihmiset kuluttavat paljon enemmän aikaa älylaitteillaan kuin ennen. Matkapuhelimien maailmanlaajuisen määrän on ennustettu olevan 5,07 miljardia vuonna 2019 [1]. Tämä on johtanut siihen, että yhä useampi yritys haluaa päästä osaksi mobiilisovellusten markkinoita sen kattavuuden vuoksi. Älypuhelimet ovat niin läsnä arjessa, että sen kautta on mahdollista tavoittaa monia eri ihmisryhmiä. Älylaitteet ovat kytkettyjä verkkoon ja niitä käytetään kaikenlaiseen toimintaan aina viestittelystä verkko-ostamiseen. Mobiilimarkkinoita hallitsee tällä hetkellä Applen iPhone ja erilaiset Googlen Androidiin perustuvat laitteet. Nämä myös muodostavat käytetyimmät käyttöjärjestelmät: iOS ja Android, joilla on maaliskuun 2018 tilaston mukaan noin 95%:n markkinaosuus [2]. Näille alustoille on tarjolla omat kehitysympäristönsä ja ohjelmointikielensä; iOS:llä kehitetään Objective-C:llä tai Swiftillä ja Androidilla Javalla tai Kotlinilla. Tämä siis tarkoittaa, että halutakseen tavoittaa suurimman osan mobiilisovellusten käyttäjistä pitää kehittää sovellus kahdelle alustalle ja kahdella eri ohjelmointikielellä. Tällaisten osaajien onnistunut hankkiminen saattaa olla kallista ja työlästä. Tämä ja mobiilisovellusten kasvava tarve on johtanut erilaisten sovelluskehysten nousuun, jotka koittavat vähentää mobiilisovellusten kehitykseen kuluvia resursseja ja nopeuttaa mobiilisovellusprojektin toteutumista.

Web-kehitykseen käytetyt teknologiat ovat kehittyneet ottamaan paremmin pienemmät älypuhelimien ruudun koot sekä mahdolliset internetyhteyksien vaihtelut huomioon. Näiden lisäksi on jo kehitetty useampia sovelluskehyksiä. Näistä yksi tämän hetken suosituimpia sovelluskehyksiä on React Native. RN on Facebookin kehittämä, ja se perustuu React -nimiseen käyttöliittymien toteuttamiseen tarkoitettuun JavaScript -kirjastoon. [3;4;5.] Sen ohjelmointikielenä toimii JavaScript ja se mahdollistaa sovelluskehityksen molemmille alustoille. Toisin kuin monet muut kehykset, se ei perustu web-näkymien käyttämiseen. Sillä on mahdollista toteuttaa mobiilisovelluksia ilman suurempaa natiivikehityksen osaamista, vaikkakin JavaScriptin ja Reactin osaaminen on

tarpeen. Suosion syykin saattaa johtua juuri käytetyistä teknologista, sillä React on tämän hetken käytetyimpiä JavaScript -kirjastoja [6].

Nykyajan älylaite on suurimman osan ajasta kytkettynä verkkoon. Tämän vuoksi ne myös hyödyntävät kaikenlaisia verkkopalveluita ja monesti ne myös tarvitsevat omat taustapalvelunsa toimintoihinsa. Taustapalvelut saattavat olla hyvinkin resurssiraskaita toteuttaa. Tämä on johtanut erilaisten valmiiden taustapalveluiden suosioon. Tällaisia pilvipalveluihin perustuvia ratkaisuja voidaan myös nimittää serverless-arkkitehtuureiksi. Ne mahdollistavat kehitysresurssien vähentämisen, tai jopa kokonaan poistamisen taustapalveluista. Mobiilisovellusten keskuudessa suosittu palvelu on Firebase. Se on nykyisin Googlen omistama pilvipalvelu, joka tarjoaa laajan kirjon toiminnallisuuksia ja mahdollistaa monien sovellusideoiden toteuttamisen ilman panostamista taustapalveluiden kehittämiseen. Se tarjoaa mm. autentikoinnin, reaaliaikaisen tietokannan, analytiikkaa, mainontaa, notifikaatioita ja virheenraportointia. [7.]

2. Mobiilisovellusten kehitysalustat

2.1 Natiivi

Natiivilla sovelluskehityksellä tarkoitetaan sovelluskehitystä, joka on kehitetty tietylle alustalle käyttäen alustan kehittäjän määrittelemiä kehitystapoja. Natiivikehitys tarjoaa pääsyn lähelle alustaa ja näin ollen mahdollistaa alustan uusimpien ja tehokkaimpien ominaisuuksien käytön. Yleisesti ottaen natiivikehitys on resurssoinniltaan raskaampaa, sillä monesti se edellyttää usean kehittäjän osaamista, mikäli mobiilisovellus halutaan julkaista usealle alustalle.

2.1.1 Android

Android on Googlen kehittämä avoimen lähdekoodin Linux-pohjainen käyttöjärjestelmä [8]. Se on tämän hetken käytetyin mobiilikäyttöjärjestelmä, valtaamalla jopa 70 % markkinaosuudesta. Androidin kohdalla tämä tarkoittaa Android SDK:ta, Gradle-koontityökalua, Java-ohjelmointikieltä sekä Android Studio IDE:tä. Vuonna 2017 Google julkaisi myös virallisen tuen Kotlin-ohjelmointikielelle, joka on yleistynyt Android-

kehityksessä ja alkaa vakiinnuttaa paikkaansa Androidin kehityskielenä [9]. Androidin SDK sisältää itse alustan ja rajapinnat kehittämiseen, työkaluja virheiden etsimiseen, koontiin ja testaukseen sekä emulaattorin, jolla voidaan ajaa Android-sovelluksia virtuaaliympäristössä. Androidin virtuaaliympäristöt pyrkivät emuloimaan oikeaa Android-laitetta mahdollisimman tarkasti.

2.1.2 iOS

Applen kehittämän suljetun lähdekoodin käyttöjärjestelmän, iOS:n kehittämiseen tarvitaan X-Code -ohjelmointiympäristö, iOS SDK ja käytettävänä ohjelmointikielenä Objective-C tai sen korvannut uudempi Swift. [10.] SDK:n mukana tulevat tarvittavat kirjastot ja työkalut sekä iPhone Simulaattori. Toisin kuin Androidin emulaattori, ei simulaattori pyri täydellisesti imitoimaan oikean laitteen toiminnallisuutta. tämän vuoksi ei sillä pysty kaikkea laitteiden toiminnallisuuksia testaamaan tai käyttämään. Androidia huomattavasti suljetummasta ekosysteemistä johtuen iOS-sovellusten koonti vaatii Apple-kehittäjätilin ja macOS-käyttöjärjestelmän.

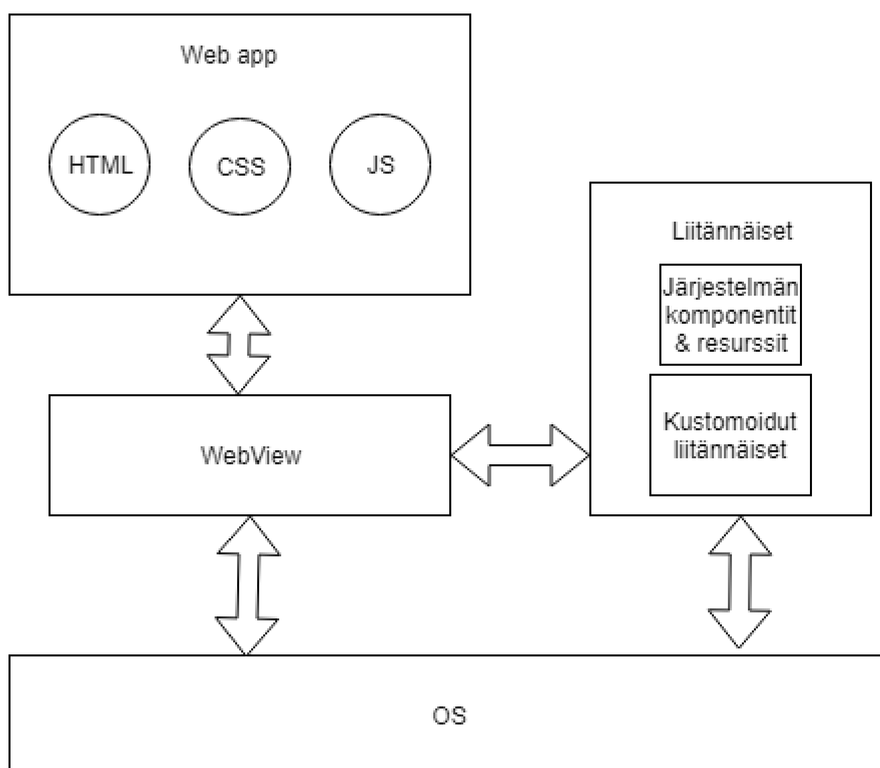
2.2 Progressiiviset web-sovellukset

Mobiilialustojen kehityksen myötä ovat myös alustojen web-selaimet kehittyneet ja pystyvät hyödyntämään yhä enemmän laitteiden resursseja. Tästä kehityksestä on syntynyt mm. Service Worker, jonka avulla pystytään selaimessa suorittamaan tausta-ajoa, joka mahdollistaa esimerkiksi ilmoitusten vastaanottamisen asynkronisesti tai verkottoman tilan käytön. [11.] Tämä verkoton tila voidaan toteuttaa toteuttamalla välimuistitus service workerillä. Tämän myötä on yleistyneet PWA:t eli progressiiviset web-sovellukset. PWA:n tarkoitus on toimia sovelluksen tavoin ja tämän vuoksi mobiiliympäristölle tyypillinen verkon tilan katoaminen on otettava huomioon, jotta käyttäjäkokemus pysyy siedettävällä tasolla. [12.] Service workereiden avulla on mahdollista toteuttaa hyvinkin pitkälle viety verkoton toiminnallisuus. Lisäksi PWA:n kanssa on mahdollista käyttää joitain laitteen komponentteja kuten kameraa, bluetoothia, paikannusta tai gyroskooppia. [13.] Näillä yhdistelmillä saadaan jo suhteellisen pitkälle viety mobiilisovellus toiminnallisuuksien osalta.

2.3 Monialustaiset sovelluskehykset

2.3.1 HTML5 -pohjaiset sovelluskehykset

HTML5-pohjaisella kehyksellä tarkoitetaan sovelluskehystä, joka toteuttaa HTML5- sekä JavaScript-pohjaisia komponentteja, joita ajetaan alustan web-näkymissä. Näiden komponenttien ja käyttöjärjestelmän välissä on joukko liitännäisiä, jolla saadaan kytkettyä komponentit kiinni järjestelmän tapahtumiin ja laiteresursseihin. Näitä teknologioita hyödyntäviä sovelluskehyksiä on jo olemassa useampiakin. Yksi tunnetuimmista on Apachen kehittämä Cordova-sovelluskehys. [14.]



Kuva 1: Cordovan arkkitehtuuri.

2.3.2 Natiivit sovelluskehykset

Vaikkakin otsikoinnissa sanotaan natiivit sovelluskehykset ei kyseessä ole täysin natiivi kehitysympäristö. Tässä yhteydessä käytettiin sanaa natiivi kuvaamaan sovelluskehyksen tapaa toteuttaa monialustaisuus. Tämä toteutetaan kääntämällä sovelluskehyksen omalla kääntäjällä käytetystä kielestä natiiveja komponentteja, joista koostetaan itse sovellus. Loppukädessä näillä kehyksillä toteutetut sekä natiivilla kehityksellä toteutetut sovellukset, sekä niiden paketoitu lähdekoodi, näyttävät siis hyvin samalta. Tämä ei kuitenkaan tarkoita, että kyseessä olisi täysin natiivi sovellus. Tällaisia

sovelluskehyksiä ovat muun muassa Xamarin ja React Native: Xamarin käyttää ohjelmointikielenään C#-ohjelmointikieltä ja React Native JavaScriptiä. Tässä insinöörityössä otetaan tarkasteluun React Native.

3. Mobiilisovelluksen taustapalvelut

Nykypäivän mobiililaitteen käyttäjä on mitä todennäköisemmin yhdistettynä internetiin. Joidenkin arvioiden mukaan vuonna 2017 olisi 8 miljardia laitetta yhdistettynä verkkoon [15]. Tämän myötä on myös käyttäjien odotukset sovelluksia kohtaan kasvaneet ja yhä useampi haluaa sulavan käyttökokemuksen ja ajankohtaisimman tiedon. Vaatimusten täyttäminen vaatiikin useimmiten sovellukselle erilaisia taustapalveluita. Näiltä taustapalveluilta tarvitaan mm. tietojen varastointia, käyttäjien tunnistamista ja tietojen noutamista.

3.1 Palvelinarkkitehtuuri

Palvelimella tarkoitetaan tässä verkkosovellusta, jota ajetaan palvelimella. Useimmiten verkkosovelluksen kehityksen sekä palvelimen ylläpidon ja pystyttämisen hoitaa sama taho, kuin mobiilikehityksen. Palvelimen voi toteuttaa hankitulle fyysiselle palvelinkoneelle, jolloin myös tietokoneen fyysiset ylläpitotoimet ovat myös sovelluksen kehittäjän vastuulla. Näistä kuitenkin käytännöllisemmät tavat on vuokrata palvelin joltain palveluntarjoajalta. Palvelimella ajettava sovellus myös yleensä hyödyntää tietokantaa toimintojensa toteuttamiseen. Palvelin tarvitsee rajapinnan, jotta mobiilisovellus pääsee sen tarjoamiin resursseihin käsiksi. Tällä hetkellä yksi erittäin suosittu tapa on REST-rajapinta. Rest perustuu HTTP:n yli kutsuttaviin tilattomiin päätepisteisiin. [16.] Rajapinnan yli voidaan tehdä CRUD-operaatioita tai ohjata liikennettä eteenpäin. Rest-rajapinnan yleisimmät http-metodit ovat GET ja POST, joiden avulla voidaan siirtää dataa molempiin suuntiin. Yleisin tietomuoto tällaisessa rajapinnassa on JSON -muotoinen data, sillä sitä on helppo lukea, kirjoittaa, generoida ja parsia.

```
{
    "data": {
        "variable": 1,
        "timeStamp": 10001293938,
        "name": "Esimerkki"
    }
}
```

Kuva 2: Yksinkertainen JSON-tietomalli

3.2 Serverless-arkkitehtuuri

Johtuen mobiililaitteiden [17] kasvavasta suosiosta sekä niiden kautta tavoitettavien ihmisten määrästä yhä useampi yritys ja yhteisö haluaa mukaan tähän markkinaosuuteen. [18.] Yritysjohtajat ovat huomanneet, että mobiilisovellusten mukaan on päästävä ennemmin kuin myöhemmin. Tästä johtuen on mobiiliosaajien tarve kasvanut huimaa vauhtia. Projekteja laitetaan käyntiin nopealla tahdilla ja usein käytettävä lopputuote halutaan käsille mahdollisimman nopeasti. Lisäksi mobiilisovelluksiin lähtevät mukaan monet startupit ja muut pienyritykset, joiden käytettävissä olevat resurssit ovat huomattavasti pienemmät kuin suuremmilla toimijoilla. Monien startupien tulonlähde on itse sovellus, joten on tärkeä saada nopeasti sovellus, jota voidaan esittää ja käyttää rahoituksen tai jalansijan saamiseksi.

Helpotusta edellä mainittuihin ongelmiin tuo sovellusten taustapalveluiden arkkitehtuuriksi serverless-arkkitehtuurin valitseminen. Serverlessillä eli palvelimettomalla arkkitehtuurilla tarkoitetaan pilvipalveluiden nojaan rakennettuja palveluita [19]. Vaikka nimi toisin sanoo, ei kyseessä periaatteessa ole palvelimeton palvelu, vaan palvelin on abstrahoitu palveluntarjoajan omiin järjestelmiin, jolloin sovelluksen kehittäjän ei tarvitse huolehtia sen ylläpidosta ja kehittämisestä. Tämä keventää resurssipaineita huomattavasti, joita taustapalveluiden kehittäminen ja ylläpito aiheuttaa. Yksi tunnettu serverless-arkkitehtuurin perustuva palvelu on vuonna 2011 perustettu Firebase, jonka nykyään omistaa hakukonejätti Google. Firebase on erityisesti mobiilisovelluksille kohdennettu palvelu, joka sisältää reaaliaikaisen tietokannan, virheenraportointipalvelun, erilaisia analytiikka- ja monitorointipalveluita, autentikointipalvelun, tiedostojen talletuksen, ilmoituspalvelun ja pilvifunktiot, joihin paneudutaan myöhemmin lisää. Näillä ominaisuuksilla voidaan viedä jo hyvinkin pitkälle lähes mikä vaan sovellusidea. Firebasea voidaan käyttää mobiilisovelluksessa joko Firebasen tarjoaman JavaScript-liitännäisen kautta tai mobiililaitteille tarkoitetuilla kehityspaketeilla.

Firebasen tietokanta on NoSQL-pohjainen, dokumenttivarastoksikin kutsuttu, tietokanta. NoSQL ei pyri relaatioiden toteuttamiseen perinteisen SQL-kyselykielen tavoin, vaan tietokantojen rakenne koostuu objekteista. Objektien lisäksi NoSQL-kannan rakenne voi myös hyödyntää dokumentteja, graafeja tai avain-arvo-pareja. [20.] Objektipohjaisen

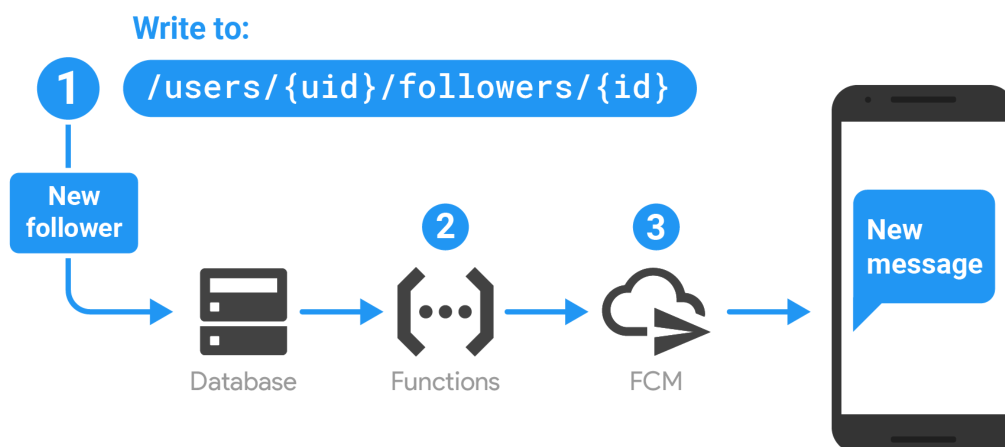
tietokannan muuntaminen on hyvin helppoa lisäämällä kenttiä objekteihin. Firebasen tietokannan kaavaa on helppo manipuloida häiritsemättä liikaa tiedon eheyttä muokkaamalla itse objekteja. Kantaan luodaan solmuja, johon voidaan lisätä helposti JSON-muodossa olevaa dataa.

Kuva 3: Esimerkki Firebasen tietokannasta.

Firebase tarjoaa valmiin ratkaisun käyttäjien todentamiseen, mikä helpottaa sovellukseen kohdistuvia tietoturvapaineita ja mahdollistaa luotettavien sovellusten toteuttamisen nopealla aikataululla. Vaihtoehtoina on perinteinen sähköposti-salasana-yhdistelmä, kirjautuminen toisen palvelun kautta kuten Facebook-tilillä tai Google-tilillä tai tunnistautuminen puhelinnumerolla. Lisäksi on myös mahdollista asettaa anonyymi tunnistautuminen. Tunnistautumisen lisäksi palvelu tarjoaa myös tunnusten palauttamisen, vahvistustekstiviestien lähettämisen ja sähköpostin varmentamisen. [21.]

Firebase tarjoaa myös pilvifunktioita. Pilvifunktioiksi kutsutaan funktioita, jotka sijaitsevat palvelinympäristössä, ja ne suoritetaan tiettyjen liipaisimien käynnistäessä ne. [22.] Niiden avulla voidaan toteuttaa logiikkaa ja pieniä ohjelmia, jotka normaalisti olisivat palvelimella. Toisin kuin täydet palvelinsovellukset, eivät funktiot ole käynnissä koko ajan, vaan ne ajetaan vain kutsusta. Funktiot siis reagoivat tapahtumiin, mihin ne on kytketty, ja näin voidaan toteuttaa logiikkaa kuten tietokantaan tulevaan muutokseen

reagointi tai vaikkapa käyttäjän kirjautuessa tehtävä käyttäjätilin muokkaus. Yksi hyvä käyttötarkoitus näille funktioille on notifikaatioiden lähetys käyttäjille tiettyjen ehtojen perusteella. Näiden funktioiden avulla voidaan toteuttaa erittäin kehittynyt kokonaisuus, joka soveltuu monien sovellusten taustalogiikoiden toteuttamiseen.



Kuva 4: Pilvifunktion käyttö notifikaation lähettämiseen.

4. React Native -sovelluskehitys

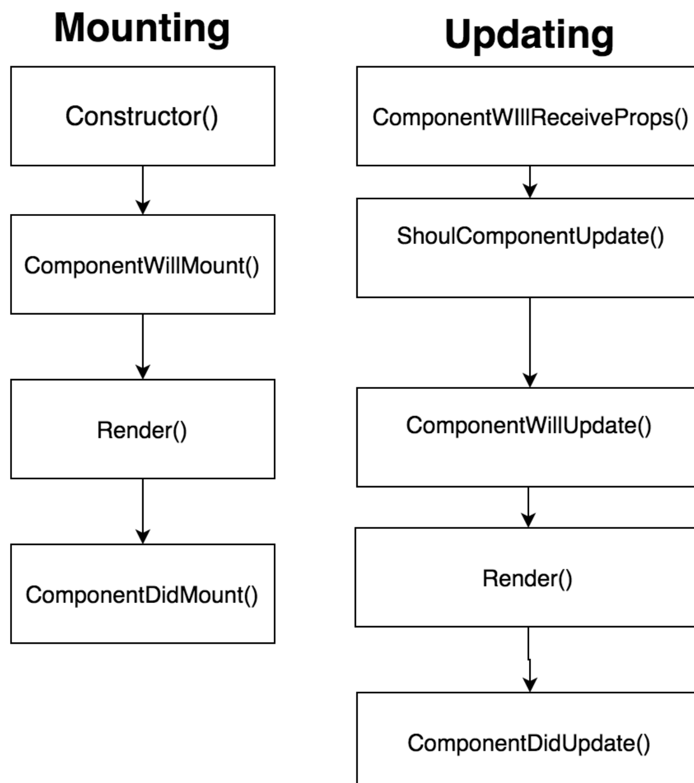
4.1 React

Kuten aiemmin havaittiin, on mobiilisovelluksien toteuttamiseen monia vaihtoehtoja alustojen omien tapojen vastapainoksi. Tässä työssä otetaan tarkempaan analyysiin tämän hetken kuumimpiin sovelluskehyksiin kuuluva React Native [23], lyhyemmin RN. RN on Facebookin kehittämä, vuonna 2013 alkunsa saanut, Reactiin perustuva mobiilisovellusten kehittämiseen tarkoitettu sovelluskehys. Ennen RN:een perehtymistä on kuitenkin syytä tutustua Reactiin. React on myös Facebookin kehittämä käyttöliittymien rakentamiseen tarkoitettu JavaScript-kirjasto. [24.] Reactin ideana on toteuttaa kapseloituja komponentteja jotka hallinnoivat omaa tilaansa. Näiden komponenttien avulla voidaan rakentaa monimutkaisia käyttöliittymiä ja jakaa sekä uudelleen käyttää komponentteja.

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

Kuva 5: Esimerkki React-komponentista.

Komponenttien vähimmäisvaatimuksena niiden tulee laajentaa perusluokkaa Component ja niiden tulee toteuttaa funktio render. [25.] Toteutettuna render tarkastelee komponentin ominaisuuksia (props) ja tilaa (state) ja palauttaa React-elementtejä, jotka voivat olla perinteisiä HTML -elementtejä kuten <div> tai toisia komponentteja, ja jotka lopulta lisätään dokumenttioliomalliin eli dom:iin [26]. Render-funktio voi myös sisältää funktiokutsuja, kunhan kutsutut funktiot myös palauttavat React-elementtejä. Komponenteilla on elinkaari, jonka vaiheet ovat kiinnitys, päivitys, irrotus ja virhe. Kiinnitys vaiheessa ollaan, kun komponenttia alustetaan ja kiinnitetään dom:iin. Kiinnityksen vaiheisiin pääsee käsiksi kutsumalla siihen liittyviä tapahtumankäsittelijöitä.



Kuva 6: React-komponentin mounting ja updating elinkaaren vaiheet.

Kiinnitysvaiheen ensimmäinen kohta on komponentin konstruktori, ja se vaatii yläluokan konstruktorin kutsumista ominaisuuksilla eli propseilla, jotta itse komponentti saa ominaisuudet käyttöön aliluokkaan. Jotta mahdollisilta virheiltä vältytään, ei tulisi konstruktorissa käyttää mitään logiikkaa, jolla on sivuvaikutuksia vaan sillä tulisi vain alustaa komponentin tila ja mahdolliset tilanteiden käsittelijät (Event handlers) [27]. Seuraava vaihe on ComponentWillMount, jota kutsutaan juuri ennen komponentin kiinnittymistä. Seuraavaksi kutsutaan render, jolloin komponentti piirretään ja lopuksi kutsutaan ComponentDidMount. Tässä viimeisessä tilassa komponentti on täysin alustettu ja valmis toimimaan, joten tämän funktion kutsussa voidaan suorittaa kutsuja, joilla on sivuvaikutuksia tai tilauksia, kuten palvelinpyyntö noutaa dataa näytettäväksi.

Päivitysvaihe käynnistyy, kun komponentti saa uudet ominaisuudet. Tähän vaiheeseen voidaan kytkeytyä ComponentWillReceiveProps funktiolla, joka kertoo, että komponentti on saamassa uudet ominaisuudet. Tässä funktiossa on mahdollista vertailla nykyisiä ja uusia ominaisuuksia. ShouldComponentUpdate kertoo, onko komponentilla tarve piirtyä uudelleen. Toteuttamalla itse kyseinen funktio voidaan tarkemmin kontrolloida piirtämisen tarvetta, sillä vakioarvoilla komponentti piirretään uusiksi kaikista muutoksista. CompontentWillUpdate- ja ComponentDidUpdate-funktioilla voidaan reagoida komponentin päivittymiseen ennen ja jälkeen. Näiden lisäksi voidaan tehdä logiikkaa, kun komponentti on poistumassa kirjoittamalla se ComponentWillUnmount-funktioon ja virheet renderöinnissä ja elämänkaaren toiminnoissa voidaan napata ComponentDidCatch-funktiossa.

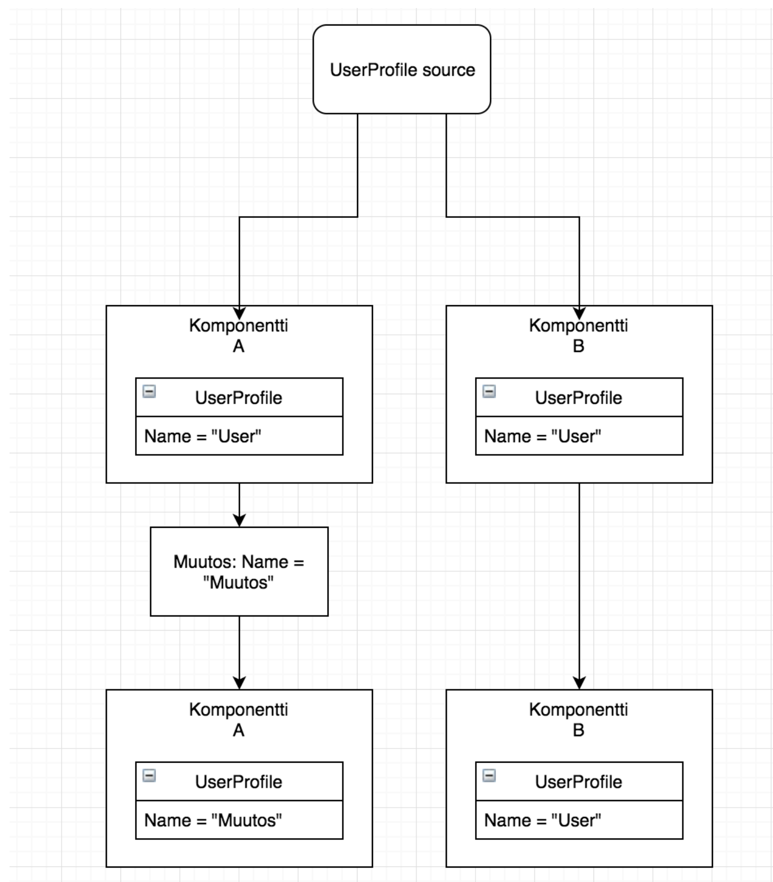
Komponentille on myös tärkeää tila ja ominaisuudet (state & props), jotka muistuttavat toisiaan, mutta joilla on eri käyttötarkoitus. Propsit eli ominaisuudet ovat arvoja, jotka annetaan komponentille sen alustusvaiheessa, ja ne ovat vain lukutyyppisiä. Propseissa voidaan välittää esimerkiksi komponentin asetuksiin liittyviä ominaisuuksia. Tila taas on komponentilla oleva objekti, jolla kuvataan komponentin tilaa. Kutsumalla komponentilla this.state päästään käsiksi yhteen tilaan. Tilaa voidaan myös muokata, mutta se on suoritettava kutsumalla this.setState()-funktiota, jolloin komponentti tietää tilan muuttuneen ja osaa näin pyytää uudelleenpiirtoa.

```
render() {
    // Tulostaa tilassa olevan muuttujan 'text'
    <div> {this.state.text} </>
}

changeText(newText) {
    //asettaa tilassa olevalle muutujalle 'text' uuden arvon
    this.setState({
        text: newText
    })
}
```

Kuva 7: Esimerkki tilan käytöstä.

4.2 Tilan hallinta

React-komponentin tila yleensä sisältää komponentin tarvitsemia tietoja kuten näytettävää tekstiä tai totuusarvoja, joilla komponentin toimintaa voidaan säädellä. Todellisuudessa kuitenkin tilassa oleva tieto on laajempaa ja komponentin tilassa on kokonaisia objekteja. Esimerkiksi komponentti, joka näyttää käyttäjätietoja, saattaa sisältää kokonaisen käyttäjäobjektin, joka sisältää monta kenttää. Jotta tämä käyttäjä saadaan komponenttiin, on se välitettävä sille ominaisuutena tai se on ladattava jostain. Jälkimmäisessä esimerkissä voisi luoda kutsun komponentin käynnistyksessä, joka hakee käyttäjätiedot-olion. Tämän jälkeen voidaan asettaa käyttäjäobjekti tilaan talteen kutsumalla setState(käyttäjä). Nyt käytössä on käyttäjäobjektin sisältämät tiedot. Jos joku toinen komponentti haluaa myös käyttöönsä käyttäjän tiedot, on kyseisen komponentin myös hankittava käyttäjän tiedot. Miten sitten varmistetaan tietojen oikeellisuus tilanteessa, jossa molemmat komponentit potentiaalisesti muokkaavat kyseistä dataa?



Kuva 8: Tilanne jossa kaksi komponenttia näyttävät samaa dataa.

Kuten kuvasta 8 huomataan, eivät kahden eri komponentin tiedot ole samat, vaikkakin ne semanttisesti ovat. Tilanne voitaisiin korjata siirtämällä tila toiseen komponenttiin ominaisuuksien avulla. Tämä kuitenkin lisäisi tahattomien tiedonmuutosten riskiä ja loisi hyvin vaikealukuisen verkon tietojen välillä datamäärien kasvaessa sovelluksessa. Toinen, järkevämpi tapa korjata tilanne, olisi kaiuttaa komponentin A muutos komponenttiin B. Jotta tämä onnistuisi, tulisi niiden käsitellä yhteistä käyttäjäobjektia. Tämä tilanne voidaan luoda yhteisellä, globaalitilalla, jota molemmat komponentit käyttävät. Näin ollen komponentti A muuttaa globaalia tietoa, ja se on tarjolla kaikille muille halukkaille. Tämän lisäksi tarvitaan myös keino ilmoittaa kaikille kiinnostuneille, että muutos tilaan on tapahtunut. Tähän tilan hallintaan on olemassa useampikin kirjasto, joista kaksi suosittua kirjastoa ovat Redux ja MobX. Ne molemmat säilyttävät tilaa varastoissa ja toteuttavat kokonaisvaltaisen tilanhallinnan, mutta toteuttavat sen eri tavalla.

Redux perustuu kolmeen ydinperiaatteeseen: on olemassa vain yksi totuuden lähde, tila on vain lukutyyppinen ja muutokset tehdään puhtailla funktioilla. [28.] Yksi totuuden lähde tarkoittaa, että kaikki tieto haetaan yhdestä paikasta. Tämä mahdollistaa

esimerkiksi palvelimelta haettavan datan koostamisen yhdeksi objektiksi, jota käytetään tilana. Luomalla tila uudelleen muutoksen tapahduttua voidaan tieto pitää vain luku-tyyppisenä ja välttyä ei toivotuilta sivuvaikutuksilta. Nämä muunnokset käynnistetään toiminnoilla (action), jolloin muutoksien hallinta pysyy kontrollissa. Tässä tapauksessa hyödynnetään puhtaita funktioita, eli funktioita, jotka palauttavat aina saman, ennakoitavan tuloksen. [29.] Esimerkiksi funktio, jolle annetaan parametrina luku 2, kertoo sen luvulla 5 ja palauttaa arvon 10, on puhdas funktio niin kauan, kuin sen paluuarvo on aina sama eli tässä tapauksessa 10. Jos sille annetaan parametriksi luku 5, on sen paluuarvo oltava ennakoitavissa, ja se on tässä tapakusessa 25.

Puhtailla funktioilla voidaan taata, että mitään muuta tilaa ei ylläpidetä, kuin yksi, jolloin myös yksi totuuden lähde toteutuu. Muutos-funktiot ottavat siis parametreinaan nykyisen tilan sekä muunnoksen ja palauttavat uuden tilan. Puhtaan funktion vuoksi on saavutettava tila ennakoitavissa.

```
//Tilavarastolle lähetetään  pyyntö muuttaa tilaa
store.dispatch({ type: 'SET_NAME', name: "Matti Meikäläinen" })}
store.dispatch({ type: 'CLEAR_NAME' })}

// reducer vastaanottaa pyynnon, tekee
export reducer (state , action) => {
  switch (action.type) {
    case 'SET_NAME':
       //uusi tila vanhaan perustuen
      return [...state, name: action.name]
    case 'CLEAR_NAME':
        //uusi tila vanhaan perustuen
      return [...state, name: ""]
    default:
      return state
  }
}
```

Kuva 9: Lyhyt esimerkki reduxin periaatteista.

MobX taas lähestyy asiaa eri näkökulmasta [30]. Se ei pyri saavuttamaan tilan eheyttä luomalla muuttumattoman tilan, vaan tilan eheys taataan tekemällä tilan poikkeamat mahdottomiksi. Tämä toteutetaan takaamalla, että kaikki, mikä tilasta on johdettavista, johdetaan. MobX käyttää apunaan tarkkailijoita saavuttaakseen tämän johdettavuuden. Tilasta kiinnostuneet komponentit rekisteröivät tarkkailijan, joka kuuntelee jotain tarkkailtavaa. Tarkkailija saa välittömästi tiedon, kun tarkkailtavassa tapahtuu muutos. Näin saavutetaan reaktiivisia tietomalleja, jonka muutoksiin voidaan reagoida

automaattisesti. Tämän lisäksi voidaan ottaa tarkkailtavasta tilasta johdettuja arvoja ulos ilman tilaan koskemista.

```
//Kuunneltava tietomalli
var person = mobx.observable({
    id: "10349912"
    firstName: '',
    lastName: '',
    get fullName () {
        return this.firstName + ' ' + this.lastName;
    }
});
//mobx:n autorun funktio ajetaan aina, kun person muuttujassa tapahtuu muutos
mobx.autorun(function fullName () {
    console.log('name: ' + person.fullName);
});

//person muuttujan muuttaminen käynnistää vaikutusten ketjun
person.firstName = 'Matti'; //yllä oleva autorun funktio tulostaa 'name: Matti'
person.lastName = 'Meikäläinen'; //yllä oleva autorun funktio tulostaa 'name: Matti Meikäläinen'
```

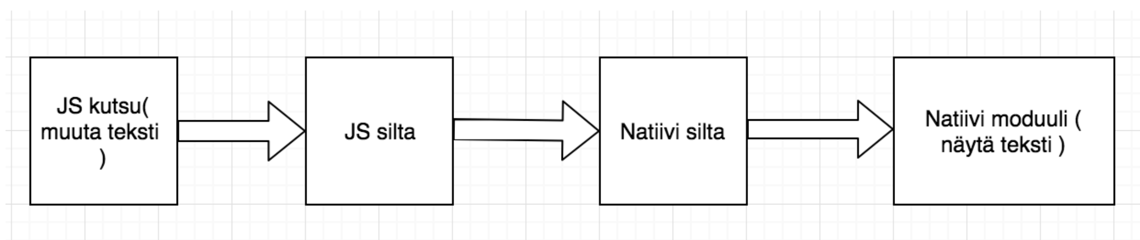
Kuva 10: Lyhyt esimerkki MobX:n periaatteista

4.3 React Native

React Native, lyhyemmin RN, on sovelluskehys, jonka avulla voidaan kehittää mobiilisovelluksia iOS- ja Android-alustoille käyttämällä Reactia. Sillä kirjoitettu koodi on n. 70 % jaettavaa alustojen välillä, ja se tuottaa loppukädessä natiivia koodia. Se on suosittu sovelluskehys mobiilikehityksessä johtuen Reactin suosiosta ja avoimesta lähdekoodista. RN mahdollistaa myös natiivien moduulien kirjoittamisen ja näiden liittämisen React Native -kerrokseen natiivi JavaScript-silloilla. [31.] Tämä tarkoittaa teoriassa sitä, että kaikki toiminnallisuudet, jotka voidaan toteuttaa alustojen omilla kehityspaketeilla, voidaan toteuttaa RN:lla. RN vaatii Reactin lisäksi Node.js:n toimiakseen. Node [32] on JavaScript-moottori, joka on rakennettu Chromen V8 JavaScript -moottorin päälle, ja se käyttää tapahtumapohjaista toimintamallia, jonka vuoksi se on kevyt ja tehokas. RN:ssä Noden avulla on toteutettu silta, jonka avulla JavaScript RN -projekti kääntää natiiviin muotoon. Lisäksi Node sisältää npm:n eli node package managerin, joka on maailman suurin avoimen lähdekoodin kirjastojen ekosysteemi. Vaihtoehtoisesti on tarjolla myös npm:ää uudempi sovellus Yarn, joka on npm:ää hieman nopeampi, mutta tarjoaa samat kirjastot. [33] Tämän lisäksi on hyvä olla myös Watchman [34], jonka käyttötarkoitus on seurata tiedostomuutoksia projektissa. Näiden muutosten perusteella voidaan kääntää vain muutokset, eikä koko projektia, ja nopeuttaa latausaikoja huomattavasti. RN sisältää myös oman komentorivirajapinnan, react-native-cli:n, jonka avulla RN:n käsitteleminen on helpompaa.

RN sisältää myös paketoijan (packager). Se on Node-palvelin, johon kohdelaite (tai simulaattori) ottaa yhteyden. Paketoija käynnistetään npm:n avulla, kun RN-projekti on luotu kutsumalla projektin kansiossa npm start. Kun palvelin on käynnissä, voidaan sen kautta koostaa ja välittää paketti kohdejärjestelmään. Se mahdollistaa myös kohdejärjestelmässä olevan RN-sovelluksen monitoroinnin ja debuggauksen.

RN-projekti koostuu moduuleista, jotka tulevat itse RN:n myötä ja käyttäjän lisäämistä moduuleista, alustojen natiiveista moduuleista ja lähdekoodista. RN luo yhteyden natiiviin puoleen silloilla. Se sisältää kaksi siltaa: JavaScript-sillan ja natiivin sillan. [35.]



Kuva 11: React Native natiivin ja JavaScriptin silta.

JavaScript-sillan tehtävä on toimia välittäjänä kehittäjän kirjoittaman JavaScript-koodin ja natiivien moduulien välillä. JavaScript-silta on sidottu laitteessa olevaan säikeeseen, joka on varattu JavaScriptin suorittamiseen. Tätä säiettä kutsutaan myös js-säikeeksi, ja se sisältää koostetun JavaScript-pakettin. Kun natiiviin komponenttiin kohdistettu kutsu luodaan, lisätään se nativiin siltaan menevään jonoon, jota kutsutaan varjojonoksi (shadow queue). Jokaisella sovelluksen moduulilla on oma jononsa. Tätä jonoa puretaan pääsäikeessä eli UI-säikeessä. Natiivi silta pitää huolen, että välitetty kutsu ohjautuu oikeaan moduuliin ja se suoritetaan pääsäikeessä. Lyhyesti sanottuna RN muuntaa JavaScript-koodia natiiviksi koodiksi siltojen ja natiivien moduulien avulla. Luomalla natiivi moduuli, vaikkapa MapModule, joka tarjoaa järjestelmän karttapalvelun (Google Maps tai Apple Maps), ja paljastamalla se natiiville sillalle, voidaan sitä kutsua JavaScript-koodissa tyylillä MapModule ja käyttää järjestelmän karttaa hyväksemme. Tämä prosessi on kuitenkin työläs, sillä molemmille alustoille (iOS ja Android) on luotava omat moduulit. Onneksi ei kaikkea kuitenkaan tarvitse tehdä, vaan RN-sovelluskehys sisältää eri moduuleja tavallisen mobiilisovelluksen tarpeisiin. Tämän lisäksi RN:n laaja yhteisö tarjoaa monia avoimen lähdekoodin moduuleja.

4.4 Työkalut ja menetelmät

Itse RN:n ja Noden sekä npm:n lisäksi kehitykseen tarvitaan editori. RN: n kehittämiseen käy mikä tahansa tekstieditori, mutta tietyt editorit ovat suositumpia johtuen niiden liitännäisistä ja JavaScriptin kehitykseen soveltuvuudesta. Suosittuja editoreita RN-yhteisön keskuudessa on mm. Atom, Sublime text ja Visual Studio Code.

RN:n ja Reactin kanssa käytetään usein myös Babel-kääntäjää [36]. Babel on uudemman sukupolven JavaScript-kääntäjä, joka mahdollistaa uusimpien JavaScript-versioiden käyttämisen vanhemmissa ympäristöissä ilman ristiriitoja. Tämän myötä voidaan kehityksessä käyttää uusia ja kehitystä helpottavia toiminnallisuuksia kuten nuolifunktioita, luokkia ja destrukturointia. Nuolifunktion avulla voidaan korvata JavaScriptistä tuttu function(x) {} syntaksilla x => {}. Destrukturoinnilla taas tarkoitetaan objektin muuttujien avaamista, jonka avulla tehdään koodista luettavampaa ja nopeampaa kirjoittaa.

```
var { firstName, lastName } = user
console.log(firstName + " " + lastName)
```

Kuva 12: Esimerkki destrukturoinnista.

Koodin laadun varmistamiseksi on myös syytä tehdä toimenpiteitä. Tehokas työkalu tähän on ESLint [37], joka mahdollistaa mallien tunnistamisen ja raportoinnin JavaScriptillä kirjoitetusta lähdekoodista. Tämä mahdollistaa myös ei-haluttujen mallien tunnistamisen, kuten syntaksivirheet tai ei-haluttu formatointi. Myös suurimpaan osaan JavaScriptiin kohdennetuista editoreista on saatavilla liitännäinen, joka lukee eslintin syötettä ja näyttää suoraan editorissa nämä havaitut poikkeamat.

```
const InputWithIcon = props => {
  return <View />
};
```

Kuva 13: Visual studio coden näyttämä syntaksivirhe.

Kuvassa 13 on nähtävissä Visual Studio Coden eslint-liitännäisen näyttämä virhetilanne, jossa se on tunnistanut koodista puuttuvan puolipisteen. ESlint on lähes pakollinen työkalu sen tuottaman hyödyn vuoksi. Kuvan 13 tilanne voi helposti jäädä kehittäjältä huomaamatta ja aiheuttaa ongelmia koonnin ja ajon aikana. JavaScript on luonteeltaan ekspressiivinen eikä tunne muuttujien tyyppejä ja sen vuoksi voi aiheutua harmillisia virhetilanteita, jotka voitaisiin ennaltaehkäistä staattisella tyypityksellä. Facebook on kehittänyt tähän työkalun nimeltä Flow. Flow mahdollistaa muuttujien tyypittämisen ja editorien liitännäisten avulla näistä varoittamisen jo kehitysvaiheessa kuten vahvasti tyypitetyissä ohjelmointikielissä.

Itse koodin testaamiseen on tarjolla JavaScript-testikirjasto jest [38]. Jest on monipuolinen testikirjasto, jolla voidaan yksikkötestien lisäksi tehdä tilannekuviin perustuvia testejä. Tilannekuviin perustuvilla testeillä voidaan testata käyttöliittymien toimintaa, etenkin odottamattomien tilanteiden varalta. Se perustuu odotetun tuloksen ja toteutuneen tuloksen vertailuun. Testattavasta komponentista luodaan kuva, tai oikeastaan näkymähierarkian kuvaus, testirenderöijän avulla, jossa on komponentin toivottu tulos annetulla tilalla. Testiä ajettaessa luodaan uusi kuva, jota verrataan tähän kuvaan. Mikäli testi epäonnistuu, mutta kyseessä on tarkoituksellisesta muutoksesta johtunut epäonnistuminen, voidaan tilannekuva päivittää tähän uuteen kuvaan.

```
//testattava komponentti
class UserInfo extends Component {
  render() {
    return (
      <View>
        <Text>
          {this.props.user.fullName}
        </Text>
      </View>
    );
  }
}
//testi
test('renders correctly', () => {
  const tree = renderer.create(
        <UserInfo user={fullName= "Matti Meikäläinen"}/>
      ).toJSON();
  expect(tree).toMatchSnapshot();
});

//Ensimmäisellä ajokerralla luodaan tilannekuva, johon myöhemmät testit vertaavat.
//Mikäli syntyneet tilannekuvat eivät täsmää, testi on epäonnistunut.
<View>
    <Text>
        Matti Meikäläinen
    </Text>
</View>
```

Kuva 14: Esimerkkitesti

5. Esimerkkisovellus: Hotelli helpotus

Tässä työssä luodaan alustava prototyyppi sovelluksesta. Sovelluksen teknologiat valikoituivat seuraavin periaattein: nopeus, monialustaisuus ja pienet resurssit. Lisäksi prototyyppiin yksi tavoite oli kokeilla RN:n soveltuvuus sovellusprojektin toteuttamiseen, ja tämän vuoksi kehykseksi valikoitui siis RN. Prototyypin ja rajallisten resurssien puitteissa todettiin, ettei sovellukselle ollut perusteltua alkaa toteuttamaan omaa taustapalveluaan johtuen Firebasen tarjoamista toiminnallisuuksista, joilla pystyttiin sovelluksen vähimmäisvaatimukset toteuttamaan. Prototyypin toivottu tulos olisi sovelluspohja, joka voitaisiin räätälöidä ja myydä. Sovellus toteutettiin Punos Mobile Oy:lle. Sovelluksen idea oli yhteisöllisyyttä hyödyntävä sovellus, jonka käyttäjät voivat jakaa ja arvostella erilaisia asioita.

Prototyypin ideaksi valikoituivat yleiset käymälät. Monella ihmisellä on terveydellisiä vaivoja, jonka vuoksi he joutuvat käymään käymälöissä normaalia useammin tai tarpeen tullessa hyvin nopeasti. Tämä saattaa aiheuttaa suurta vaivaa kyseisille ihmisille ja

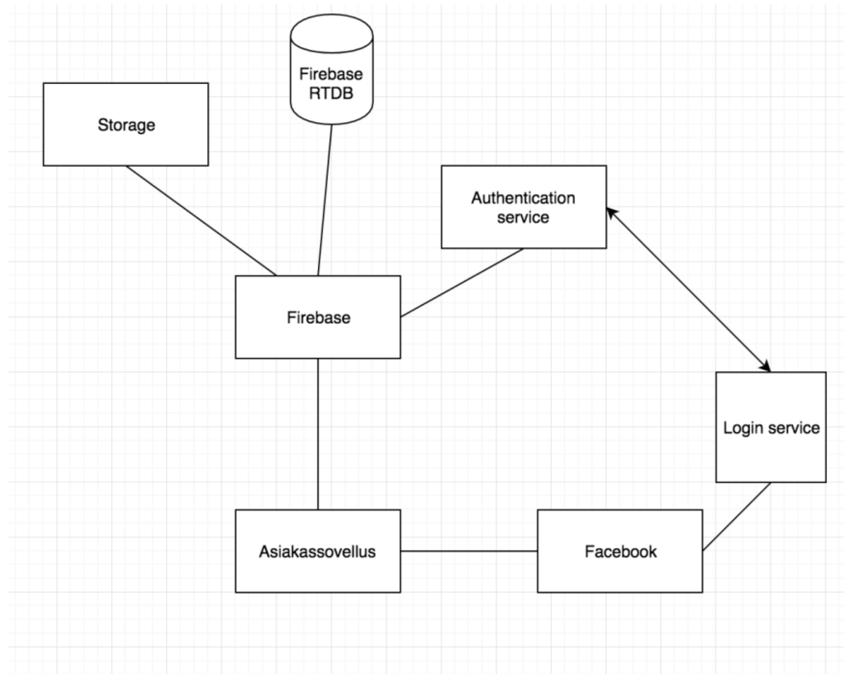
heikentää elämänlaatua ja siksi näille ihmisille ja ihmisille, jotka arvostavat hyvää käymäläkokemusta, haluttiin tarjota sovellus. Sovellus on kuitenkin soveltuva myös kenelle tahansa, joka esimerkiksi matkustaa uuteen kaupunkiin ja haluaa löytää hyvän käymälän. Käyttäjien tuli pystyä lisäämään oikeita paikkoja ja niitä piti pystyä arvioimaan. Käyttäjiä vaadittiin myös kirjautumaan, jotta sisältö pysyisi mahdollisimman asiallisena.

Taulukko 1: Syntyneitä käyttäjätarinoita

Käyttäjänä haluan lisätä uuden paikan arvosteltavaksi.
Käyttäjänä haluan lisätä kuvan paikasta.
Käyttäjänä haluan nähdä paikat lähelläni.
Käyttäjänä haluan lisätä arvion paikalle.
Käyttäjänä haluan nähdä paikan arviot ja keskiarvon.
Ylläpitäjänä haluan hallita käyttäjiä ja sisältöä.

5.1 Projektin arkkitehtuuri

Projektin kokonaisarkkitehtuuri muodostui RN-sovelluksesta, Firebasen palveluista ja Facebookin kirjautumispalvelusta.



Kuva 15: Sovelluksen kokonaisarkkitehtuuri

Sovellukseen haluttiin luotettava kirjautumispalvelu ja sosiaalisen median yhteisöllisyyden kautta kirjautumisen vaihtoehdoksi valittiin kirjautuminen Facebook-tunnuksilla. Muut taustapalvelut kuten tietokanta saatiin Firebasen toimesta. Sovelluksen arkkitehtuuri muodostui säiliöistä [liite 1], komponenteista, tilavarastoista [liite 2], tietomalleista sekä muista apuluokista. Säiliöt (container) ovat komponenttikokonaisuuksia, jotka käytännössä muodostavat yhden mobiilisovelluksen sivun. Sivujen kokonaisuus muodostuu kirjautumisnäkymästä, päänäkymästä, paikkanäkymästä, lisäysnäkymästä ja profiilinäkymästä [liite 3]. Kommunikaatio Firebaseen hoidetaan react-native-firebase-kirjaston avulla, joka sisältää natiivit moduulit molemmille alustoille. Kirjaston avulla luodaan Firebase-instanssi, jonka avulla voidaan kutsua tietokantaa, tietovarastoja ja muita palveluita. Firebase toteuttaa myös paikallisen tietokannan, joka mahdollistaa sovelluksen verkkoyhteydettömän käytön ja tietojen synkronoinnin yhteyden palatessa. Itse tietokanta muodostuu käyttäjäkohtaisista tietueista sekä yhteisistä paikkatietueista. Sovelluksen navigointiin käytettiin react-navigation-kirjastoa, joka sisältää natiivit moduulit navigoinnin toteuttamiseen. Sovelluksen tilanhallintaan valittiin MobX.

Taulukko 2: Projektissa käytettyjä tekniikoita

Sovelluskehys	React Native
Kehityskieli	Javascript (ES6)
Kääntäjä	babel
Editori	Visual Studio Code
Tyypitys	Flow
Tilanhallinta	MobX
Versionhallinta	Git
Projektinhallinta	Pivotal tracker
Testaus	jest
Koodin siistin	prettier
Taustapalvelut	Firebase
Autentikointipalvelu	Facebook

Taulukko 3: Käytettyjä kirjastoja

geofire	Lisää helppokäyttöisen rajapinnan Firebasen tietokannan käyttämisen lokalisaation avulla. Mahdollistaa paikkatietojen lisäämisen tietorakenteisiin sekä niiden käyttämisen tietokantakyselyiden hakuehtoina.
react-native-firebase	Sisältää natiivit moduulit Firebasen SDK:n käyttöön RN:ssä.
react-navigation	Mobiilisovelluksen navigaation sisältävä kirjasto.
react-native-maps	Natiivit karttamoduulit.
ramda	Funktionaalinen apukirjasto erilaisiin loogisiin operaatioihin.
react-native-permissions	Natiivit moduulit käyttöoikeuksien kyselyyn.
react-native-geocoder	Natiivit moduulit jotka muuntavat GPS-koordinaatteja luettavaksi osoitteeksi.
react-native-vector-icons	Apukirjasto, joka sisältää laajan kirjon erilaisia vapaasti käytettäviä kuvakkeita.
react-native-image-picker	Tarjoaa rajapinnan kameran käyttöön.

5.1.1 Tietovarastot ja tilanhallinta

Kuten luvussa 5 mainittiin, on tilanhallintaan useampi vaihtoehto. Tässä projektissa valittiin käyttöön MobX, joka todettiin hieman helppokäyttöisemmäksi. MobX ottaa suurinta kilpailijaansa, Reduxia enemmän tilanhallinnan kirjaston sisuksien hoidettavaksi. Tämä mahdollistaa nopeaa sovelluskehitystä, kun kehitysaikaa ei tarvitse kuluttaa itse tilanhallinnan toteuttamiseen. Se taas vastakohtaisesti aiheuttaa sen, ettei kehittäjillä ole niin paljoa kontrollia tilanhallinnan sisäiseen toimintaan. Tämä ei kuitenkaan ole suurimmassa osaa sovelluksista ongelma ja tilanhallintakirjaston valinta lähinnä ratkeaa kehittäjien mieltymysten perusteella. MobX:n tarkkailtavat kentät kapseloitiin tietovarastoihin [liite 2] ja niiden tarkkailijat olivat sovelluksen säiliöt. Varastot ovat LoginStore, PlaceStore ja RootStore.

LoginStore sisältää kaksi tarkailtavaa kenttää, niille kaksi toimintoa sekä joukon apufunktioita.

```
@observable fbLogin = null;
@observable firebaseUser = null;

@action
logout: () => void = () => {
  this.fbLogin = null;
  this.firebaseUser = null;
};

@action
authWithFb: (response: any) => void = response => {
  this.fbLogin = response;
  if (this.firebaseUser === null) {
    this.firebaseLoginWithAccessToken();
  }
};
```

Kuva 16: LoginStoren tarkkailtavat kentät ja toiminnot.

LoginStoren tehtävä on hallita käyttäjän kirjatumista, sen tilaa sekä kirjautuneen käyttäjän tietoja. Sovellukseen kirjautuessa käyttäjän kirjautuminen hoidetaan Facebookin sovelluksessa. Tästä saadaan paluuarvona FacebookLogin-komponentilta kirjautumistiedot, jotka välitetään LoginStoreen fbLogin-muuttujaan. Tämä muuttuja on merkitty tarkkailtavaksi, jotta Facebook-kirjautumisen muutoksiin voidaan reagoida silloin, kun käyttäjä kirjautuu tai joissain tapauksissa kirjataan ulos. Kirjautumistiedot välitetään Firebaseen authWithFb-funktiolla, jossa varmistetaan kirjautumisen oikeellisuus. Onnistuneesta tapahtumasta sovellus saa Firebaselta itse Firebasen kirjautumistiedon. Nämä tiedot myös tallennetaan tarkkailtavaan kenttään. Tässä kohtaa myös LoginStore luo kirjautuneelle käyttäjälle Firebasen generoimaan käyttäjätunnisteeseen perustuvan käyttäjätietotietueen tietokantaan. Jos käyttäjällä on jo tietue, päivitetään se uuden kirjautumisen tiedoilla.

PlaceStore on sovelluksen laajin tilavarasto, sillä se käsittelee suurinta osaa sovellukseen liittyvästä tilasta. Sen tarkkailtavat muuttujat ovat currentPlace, shownPlaces ja isCreatingReview. CurrentPlace-muuttuja sisältää senhetkisen paikkaobjektin, joka on käyttäjällä jollain tavalla käsittelyssä. Tämä tarkoittaa käytännössä sitä, että joko käyttäjä on lisäämässä uutta paikkaa sovellukseen tai käyttäjä on selaamassa paikan tietoja. ShownPlaces sisältää kaikki käyttäjälle näkyvät paikat. Se on listatyyppinen muuttuja, joka elää jatkuvasti käyttäjän liikuttaessa karttaa [liite 4]. Hakualgoritmi muodostaa pisteestä x (latitudi, longitudi) ympyrän, jonka säde on

10 km. Tämän jälkeen luodaan rajaus, joka sisältää kaikki ympyrän sisään osuvat koordinaatit. Näin tiedämme, mitkä pisteet kuuluvat rajauksen sisään ja pystymme tarkkailemaan rajauksen sisään tulevia tai siitä lähteviä paikkoja. Tällaisessa toiminnallisuudessa MobX:n tehokkuus korostuu, sillä tarkkailevan käyttöliittymän on helppo reagoida muuttuvaan listaan ja piirtää karttamerkit uudelleen. IsCreatingReview-muuttujan avulla hallinnoidaan arvioiden luomiseen käytettävän ikkunan näkyvyyttä, kun arvio on talletettu onnistuneesti Firebasen tietokantaan. Tallentaminen tapahtuu mergeAndSave-funktion kautta. Muuttujien lisäksi varasto hallinnoi arviot, tietojen tallentamisen tietokantaan sekä tietojen päivittämisen.

RootStore toimii sovelluksessa varastojen varastona. Sen tehtävä on tarjota yksi liityntäpiste sovelluksen varastoihin. RootStore voisi toimia yhtenä isona varastona, jossa kaikki sovelluksen tilanhallinta hoidetaan. Sovelluksen luettavuuden ja ylläpidettävyyden vuoksi on kuitenkin järkevää kapseloida samankaltaiset tai samaan asiaan liittyvät toiminnallisuudet omissa varastoissaan ja antaa niihin yhteinen liityntäpiste. RootStoren avulla voidaan tarvittaessa myös mahdollistaa kommunikaatio varastojen välillä, kuten kirjautumisvarastosta kirjautumisen tilan hakeminen. RootStore on rivien määrässä varastoista pienin.

```
import LoginStore from './LoginStore';
import PlaceStore from './PlaceStore';

export type RootStoreType = {
  loginStore: LoginStore,
  placeStore: PlaceStore
};

class RootStore {
  constructor() {
    this.loginStore = new LoginStore(this);
    this.placeStore = new PlaceStore(this);
  }
}
export default RootStore;
```

Kuva 17: RootStore

5.1.2 Sovellus

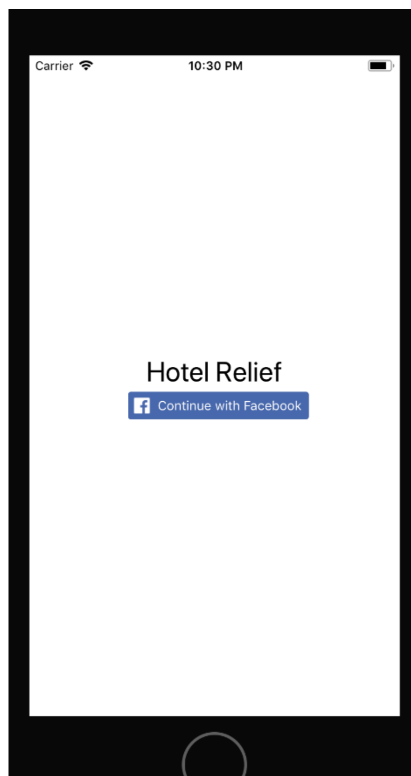
Sovelluksen ajo alkaa alustakohtaisesta index.js-tiedostosta, jossa rekisteröidään sovellus RN-sovelluskehykseen. Tiedostot ovat alustakohtaisia, jotta tarvittaessa voidaan erotella alustojen koodipohjat. Rekisteröinnin yhteydessä määritellään JavaScript-tiedosto, joka käynnistetään. Tämä tiedosto nimettiin App.js:ksi, ja se sisältää sovelluksen navigoinnin ja tietovarastojen alustuksen. App.js on ylimmän tason React-komponentti, jonka render-funktio palauttaa sovelluksen ensisijaisen navigointipinon tai kirjautumisnäkymän. Navigointi perustuu pinotyyppiseen navigointiin, jossa näkymiä lisätään ja poistetaan pinosta. Päänavigoinnin pino koostuu päänäkymästä, uuden paikan luontinäkymästä, profiilinäkymästä sekä paikan tietonäkymästä.

```
const AppStack = StackNavigator(
  {
    Main: {
      screen: MainContainer,
      navigationOptions: {
        title: 'Hotel Relief'
      }
    },
    NewPlace: {
      screen: NewPlaceContainer,
      navigationOptions: {
        title: 'Add new Place'
      }
    },
    Profile: {
      screen: ProfileContainer,
      navigationOptions: {
        title: 'Profile'
      }
    },
    PlaceDetails: {
      screen: PlaceContainer,
      navigationOptions: {
        title: 'Details'
      }
    }
  },
  { initialRouteName: 'Main' }
);

export default AppStack;
```

Kuva 18: AppStack.js. Sovelluksen päänavigointi.

Navigointi on toteutettu react-native-navigation-kirjastolla. Kirjasto toteuttaa navigation-objektin, jota välitetään näkymien välillä propseina erilaisilla navigaattoreilla. Navigatorin mukana voidaan välittää myös muuttujia, samoin kuin normaalien komponenttien kanssa. Tämä mahdollistaa tietovarastojen siirron navigaatiotapahtumien mukana. Tapahtumaan voidaan myös liittää asetuksia, kuten näytettävän ikkunan otsikko.

Kirjautumisnäkymä (LoginContainer) on kytketty Facebookin autentikointipalveluun 'react-native-facebook-login'-kirjaston avulla. Kirjasto perustuu Facebookin oman natiivin sdk:n hyödyntämiseen, jotta lopputulos on tehokas ja mobiiliystävällinen. Autentikoinnin voi myös toteuttaa hyödyntämällä Facebookin web-käyttöliittymiin tarkoitettua JavaScript-kirjastoa, mutta natiiviin tekniikkaan perustuvalla kirjastolla toiminta oli varmempaa. Itse kirjautuminen toteutetaan Facebookin SDK:n avulla, jolloin sen toiminnallisuus on sovelluksen ulottumattomissa. Kirjautumisen tulos saadaan kuitenkin paluuarvona talteen ja välitetään tilavarastoon. Näkymä hyödyntää Reactin componentDidUpdate-funktiota, jossa LoginStoren muuttujista tarkastetaan kirjautumisen tila ja jatketaan sovelluksen sisään, mikäli tila on oikea.



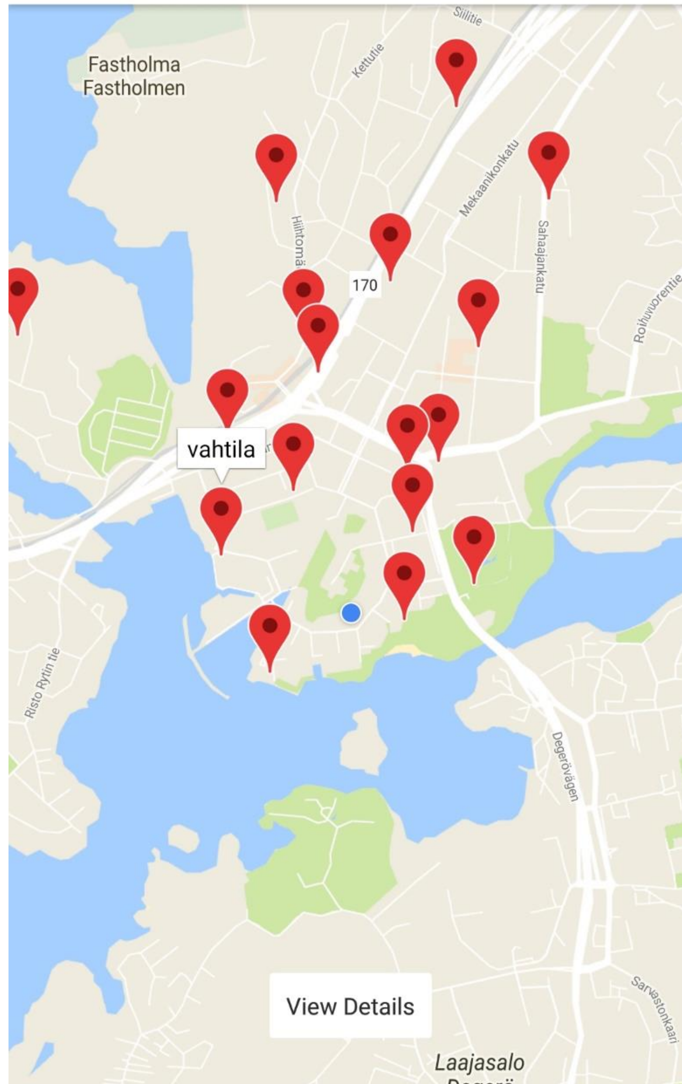
Kuva 19: Kirjautumisnäkymä

Kirjautumisnäkymästä siirrytään sovelluksen päänäkymään. Päänäkymä on karttanäkymä, jossa käyttäjä näkee lisättyjä paikkoja ja voi lisätä uusia paikkoja. Kartta on toteutettu natiiveilla moduuleilla, jossa Android perustuu Google Mapsiin ja iOS Apple Mapsiin. Näkymä pyytää ensimmäisenä luvan käyttää käyttäjän paikannusta. Käyttäjälle näytetään kehote, jossa kerrotaan sovelluksen perustuvan vahvasti paikannukseen, mikäli lupaa ei annettu. Paikannuksen kieltäminen ei estä sovelluksen käyttöä, mutta sen käyttäminen ilman paikannusta on epäkäytännöllistä. Paikkatietoihin rekisteröidään kuuntelija, mikäli lupa paikannukseen annettiin. Kuuntelijan avulla voidaan piirtää käyttäjän sijainti ja lähellä olevat paikat paikkatiedon muutosten perusteella. Luvan puuttuessa pitää paikkoja hakea kartan liikuttelun perusteella. Sovellus lataa 10 kilometrin säteeltä paikat käyttäen joko paikannusta tai kartan sijaintia. PlaceStoren kautta lisätään kuuntelija tähän tietoryhmään, jonka avulla voidaan lisätä tai poistaa näytettäviä paikkoja. Paikkoja lisätään ja poistetaan joko silloin, kun paikkatiedot ovat muuttuneet niin, ettei paikka enää kuulu muodostuneeseen ryhmään tai tietokannan tietue on muuttunut. Lisää PlaceStoren toiminnasta luvussa 6.1.1. Päänäkymä hyödyntää ehdollisia funkitoita, joita kutsutaan render-funktiossa. Tämän avulla voidaan piirtää ehdollisia tilasta riippuvia komponentteja.

```
render() {
  return (
    <View style={styles.container}>
      <MapView
        ref={ref => (this.mapRef = ref)}
        style={styles.map}
        scrollEnabled={true}
        zoomEnabled={true}
        mapType={mapSettings.mapType}
        showsUserLocation={mapSettings.showUserLocation}
        region={this.state.region}
        onLongPress={this.onLongPress}
        onPress={() => {
          this.setState({
            showAddNewLocation: false,
            currentSelection: null,
            currentNewMarker: null
          });
        }}
        onRegionChange={this.onRegionChanged}
      >
        {this.addNewMarker()}
        {this.addMarkers()}
      </MapView>
      {this.conditionalAddLocationButton()}
      {this.conditionalShowDetailsButton()}
    </View>
  );
}
```

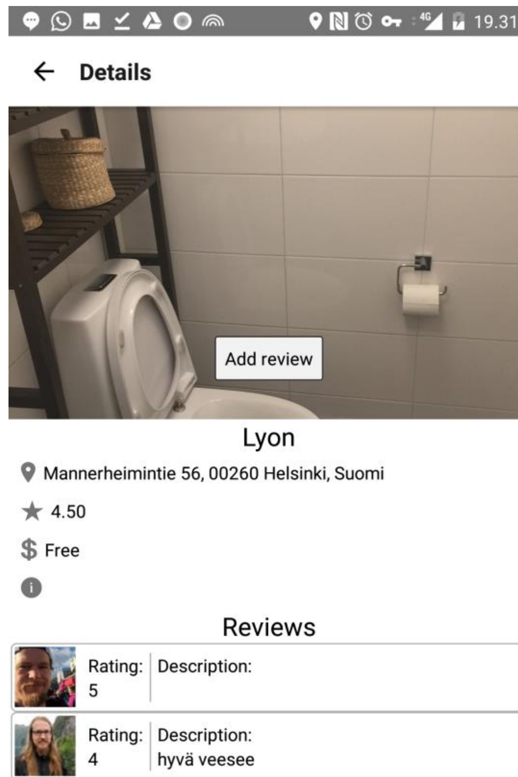
Kuva 20: Päänäkymän render-funktio

Kuvan 20 renderissä kutsutut conditionalAddLocationButton ja conditionalShowDetailsButton palauttavat JSX-elementtejä riippuen sovelluksen tilasta. Funktiot addNewMarker ja addMarkers vastaavat karttamerkkien piirtämisen. Ne palauttavat merkki-komponentteja, jotka annetaan karttanäkymälle.



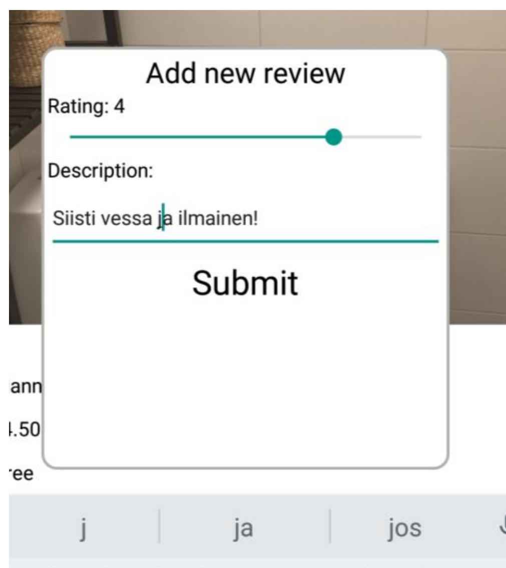
Kuva 21: Päänäkymä

Päänäkymästä voidaan navigoida sovelluksen muihin osiin. Yksi tällainen on paikan lisätietonäkymä. Lisätietonäkymässä nähdään paikan tiedot kuten osoite, hinta, keskiarvo arvioista ja yksittäiset arviot. Paikan tiedot -näkymän saa PlaceStoresta, ja ne populoidaan PlaceInformation -nimiseen komponenttiin. Arvioiden tiedot populoidaan RN:n omaan FlatList-komponenttiin. FlatList on ListView-komponenttia suorituskykyisempi mm. vähentämällä käytettävän muistin määrää virtualisoimalla näkymättömät listan osiot. [39.]



Kuva 22: Lisätietonäkymä

Lisätietonäkymä renderöi myös ehdollisesti arvion luontia varten tarkoitetun ikkunan (NewReviewModal), jonka avulla käyttäjä voi lisätä arvion kyseiselle paikalle. Ikkuna näytetään PlaceStoren isCreatingReview-tarkkailtavan kentän avulla. Logiikka on yksinkertainen if-lause, joka tarkastaa isCreatingReview-kentän totuusarvon. Tarkkailtavan kentän avulla render-funktio kutsutaan heti muutoksen tapahtuessa ja näin saadaan myös ikkuna heti näkyviin.



Kuva 23: Ikkuna arvioiden lisäämiseen.

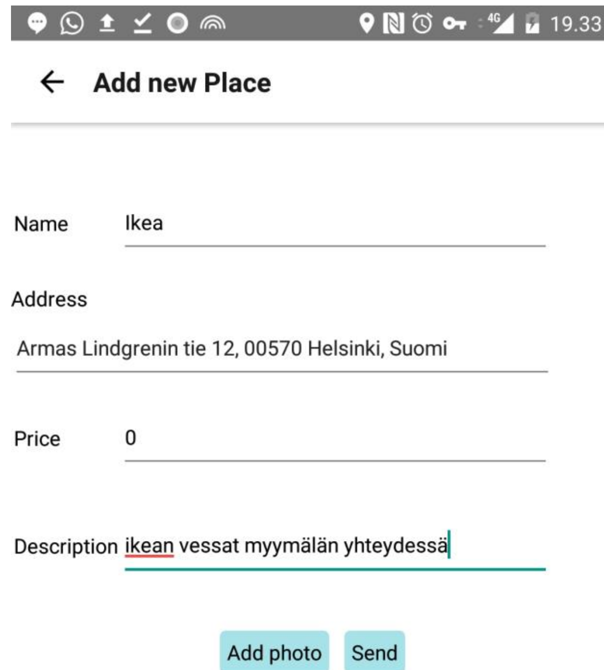
Arvioikkuna on kytketty PlaceStoreen. Komponentin tilan tiedoilla kutsutaan funktiota addReview käyttäjän painaessa 'Lähetä'. AddReview luo uuden arvio-objektin, jonka tiedot täytetään syötetyllä arviolla ja kuvauksella. Arvioon lisätään kirjautuneen käyttäjän käyttäjätunnus, nimi sekä mahdollinen kuva. Lisäksi arvio saa aikaleiman ja uuden keskiarvon. Arvio lisätään tarkkailtavaan currentPlace-kenttään 'reviews'-listaan. Lopuksi currentPlace päivitetään Firebasen tietokantaan.

```
@action
addReview: (reviewBase: Object) => void = reviewBase => {
    const review = new Review();
    review.description = reviewBase.description;
    review.rating = reviewBase.rating;
    review.user = {
      uid: this.rootStore.loginStore.firebaseUser.uid,
      displayName: this.rootStore.loginStore.firebaseUser.displayName,
      profilePicture: this.rootStore.loginStore.firebaseUser.photoURL
     };
     review.createdAt = Date.now();
     if (this.currentPlace.reviews === undefined) {
        this.currentPlace.reviews = [];
     }
     this.currentPlace.reviews.push(review);

     let ratingCount = 0;
     let totalRatingCounter = 0;
     this.currentPlace.reviews.map(item => {
       ratingCount += 1;
       totalRatingCounter += item.rating;
     });
     if (ratingCount > 0) {
     this.currentPlace.rating = totalRatingCounter / ratingCount;
         this.currentPlace.ratingCount = ratingCount;
    }

    firebase
      .database()
      .ref(`locations/${this.currentPlace.id}`)
      .update(this.currentPlace)
    this.isCreatingReview = false;
};
```

Kuva 24: addReview-funktio

Uusia paikkoja voidaan lisätä karttanäkymällä. Karttanäkymä rekisteröi normaalien painallusten lisäksi pitkät painallukset. Kartalle lisätään uusi merkki sen saadessa pitkän painalluksen tapahtuma. Samalla näytetään painike, josta käyttäjä voi aloittaa uuden paikan lisäämisen prosessin. Tämä alkaa navigoinnilla paikan lisäyksen näkymään

(NewPlaceContainer) ja alustamalla PlaceStoren currentPlace kentän uutta paikkaa varten ja päivittämällä paikkatiedot tähän objektiin. Paikkatiedot saadaan karttaan lisätystä merkistä. Näkymä hakee componentDidMount-funktiossaan PlaceStoresta currentPlace-muuttujan, jonka paikkatiedot annetaan Geokooderille. Geokooderi muuntaa koordinaattitietoja luettavaksi osoitetiedoiksi.



Kuva 25: Uuden paikan lisäysnäkymä

Uudelle paikalle syötetään nimi, osoite, hinta ja kuvaus paikasta, mutta yllämainitun geokooderin avulla voidaan osoitekenttä täyttää ennalta saatujen koordinaattien avulla eikä käyttäjän tarvitse sitä itse hakea. Käyttäjä voi myös lisätä kuvan paikasta, joka on toteutettu natiiveilla moduuleilla. Käyttäjät voivat ottaa kuvan kamerallaan tai lisätä kuvan galleriasta. Ennen kuvan ottamista pitää kuitenkin käyttäjien antaa lupa käyttää laitteen kameraa ja galleriaa. Tämän jälkeen voidaan siirtyä kuvan lisäämiseen. Sovelluksessa oli kuvan lisäämisen apuna react-native-image-picker-kirjasto, joka tekee kuvien lisäämisestä helppoa tarjoamalla valmiit moduulit.

```
showImagePicker: () => void = () => {
    ImagePicker.showImagePicker(photoOptions, response => {
      if (response.error) {
        console.error('error taking photo', response.error);
         this.setState({
             photoUri: "",
             photoFileName: ""
         });
      }
       if (response.didCancel) {
        this.setState({
         userCanceled: true,
         photoUri: "",
         photoFileName: ""
      });

      }
      this.setState({
        photoUri: response.uri,
        photoFileName: response.fileName
      });
    });
};
```

Kuva 26: Funktio kuvan ottamiseen

Uusi paikka tallennetaan save-funktiolla, joka välittää tilassa olevat tiedot PlaceStoreen kutsumalla mergeAndSave -funktiota [liite 5]. PlaceStoressa currentPlace-muuttujaan lisätään saadut tiedot ja lisätään lisääjän tiedot ja aikaleima. Lopuksi currentPlace päivitetään Firebasen tietokantaan, ja jos paikkaan lisättiin kuva, lisätään se Firebasen tietovarastoon.

6. Tulokset

Projektin tavoitteena oli saada ymmärrys sovelluskehittämiseen React Nativen avulla, tutkia serverless-arkkitehtuurien ja pilvipalveluiden hyödyntämistä mobiilisovelluskehityksessä ja saada laajempaa näkemystä mobiilisovellusten kehitystavoista. Näihin tavoitteisiin päästiin ja projektin jälkeen pystytään antamaan oma arvio kustakin menetelmästä. Itse sovellus jäi vielä prototyypiksi, mutta sen jatkokehitystä on suunniteltu.

Minulla oli projektia aloittaessa hyvä kokemus natiivista sovelluskehityksestä Android-alustalle ja hieman kokemusta React Nativesta. Kuten alussa todettiin, mobiilisovellusten kehittämiseen on erilaisia työkaluja ja sovelluskehyksiä todella paljon. Ne nousevat pinnalle ja putoavat kelkasta nopeaan tahtiin. RN:kin haastaja varmasti tulee jossain vaiheessa, mutta sen käytön täydellinen lakkaaminen on epätodennäköistä johtuen muutamasta seikasta. Ensinnäkin RN:n ympärillä on erittäin laaja avoimeen lähdekoodiin perustuva yhteisö, jotka tuottavat paljon käyttökelpoisia kirjastoja ja parannuksia RN:ään. Tämän lisäksi RN:n taustalla on sosiaalisen median jätti Facebook, ja se perustuu suosittuun React.js-kirjastoon. StackOverflow:n 2018 kehittäjille suunnatun kyselyn mukaan JavaScript oli käytetyin teknologia ja React toiseksi suosituin kirjasto [40]. Kyselyyn oli vastannut yli 100 000 sovelluskehittäjää. React Nativen suosion taustalla on varmastikin suurimpana tekijänä se, kuinka vaivattomasti web-kehityksen ja Reactin taitaja pystyy siirtymään mobiilisovelluskehitykseen opettelematta Javaa, Objective-c:tä, Swiftiä tai Kotlinia.

Kehittäjälle, joka on jo osaava natiivien sovellusten kehittäjä, ei RN välttämättä tuo kuitenkaan niin suoraa lisäarvoa, joten kehyksen valitseminen kannattaa suunnitella tarkoin. RN:llä voidaan toteuttaa sovellus molemmille alustoille, joka saattaa näkyä projektin kustannuksissa tai vaadittavassa henkilötyöpäivissä. Tämä varmastikin on teknologian viehätys liiketoiminnallisessa mielessä. Projektin aikana kuitenkin havaittiin, että RN ei tule ilman ongelmia ja niiden ymmärtäminen on tärkeää teknologiaa valitessa. RN on sovelluskehys, jolla sovellusten tekeminen onnistuu Android- ja iOS-alustoille, mutta vaikkakin koodipohja on jaettavaa alustojen välillä, joutuu yllättävänkin paljon tekemään alustakohtaista hienosäätöä tekemään lähdekoodiin, joten työmäärällisesti ei voi määritellä kahdelle alustalle toteutetun RN-sovelluksen vievän tuplasti vähemmän kuin kahden natiivisovelluksen. RN on vielä uusi kehys ja nojautuu paljon sen yhteisöön, jotta samat toiminnallisuudet ovat tarjolla kuin natiivissa kehityksessä. Tämän vuoksi RN-sovellus saattaa vaatia paljonkin kolmansien osapuolten kirjastoja, ja tästä saattaa aiheutua ylimääräistä ylläpitotyötä. RN päivittyy suhteellisen tiheään tahtiin, ja tämän vuoksi päivityksen yhteydessä jotkut kolmannen osapuolen kirjastot saattavat lakata toimimasta halutulla tavalla. RN vaatii myös väistämättä natiivin puolen osaamista, mikäli ei kirjastoja halua käyttää, sillä RN ei tarjoa kaikkia toiminnallisuuksia kuin natiivit SDK:t. Samasta syystä myös uusimmat toiminnallisuudet tulevat natiivia myöhemmin RN:ään. Tilanteesta ja tarjolla olevista resursseista, kuten kehittäjistä tai budjetista, riippuen voi kuitenkin RN:än tuoma lisäarvo olla suuri ja sen valitseminen sovelluksen alustaksi on täysin perusteltua, mutta sen tuomat mahdolliset ongelmat tulee tiedostaa.

Serverless-arkkitehtuuri mahdollistaa nopean sovelluskehityksen ja sovelluskehityksen ilman hyvää palvelinsovellusten kehitystaitoa. Firebase on erittäin hyvä ratkaisu prototyyppien ja muiden pienten nopean aikataulun mobiilisovellusten taustapalveluksi. Se tarjoaa kaiken tarvittavan suurelle osalle sovellusideoista aina autentikointi-palvelusta tietokantaan. Firebasen taustapalvelut soveltuvat hyvin esimerkiksi startup-yritykselle, joka haluaa sovelluksen ensimmäisen version nopeasti, jotta voivat kerätä rahoitusta jatkokehitykseen. Firebase tuo myös isoihin it-ratkaisuihin lisäarvoa mm. analytiikan, virheraportoinnin ja notifikaatioiden kautta. Isot it-hankkeet törmäävät kuitenkin väistämättä Firebasen rajoitteisiin, jonka vuoksi ei voi pelkästään siihen nojautua. Isoimpana ja ensimmäisenä rajoitteena tulee vastaan Firebasen tietokanta. Vaikkakin se on nopea ottaa käyttöön, niin se on vain dokumenttitietokanta, jossa ei onnistu SQL-tietokannoille tyypilliset toiminnot kuten relaatiot. Lisäksi suurien tietomäärien käsittely sen kanssa on liian työlästä vapaiden kyselyjen puuttuessa.

RN yhdistettynä Firebasen tyyliseen palveluun mahdollistaa sovellusten kehittämisen pienellä budjetilla. Se mahdollistaa yksittäisten kehittäjien toteuttaa helpommin omia ideoitaan ja mahdollistaa uusien ideoiden nopean realisoinnin kokeiltavaan muotoon. Yrityksille se mahdollistaa helpommat resurssit toteuttavat pieniä sisäisiä sovelluksia tai kokeilla uusia konsepteja ilman suurta investointia.

Lähteet

1. Number of mobile phone user worldwide from 2013 to 2019 (in billions), Statista. Verkkoaineisto <https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/> luettu 22.1.2018.
2. Global mobile OS market share in sales to end users from 1st quarter 2019 to 2nd quarter 2017, Statista. Verkkoaineisto. < https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/>.
3. Exploring the state of mobile development with Stack Overflow trends, David Robinson. Verkkoaineisto <https://stackoverflow.blog/2017/05/16/exploring-state-mobile-development-stack-overflow-trends/> luettu 15.4.2018.

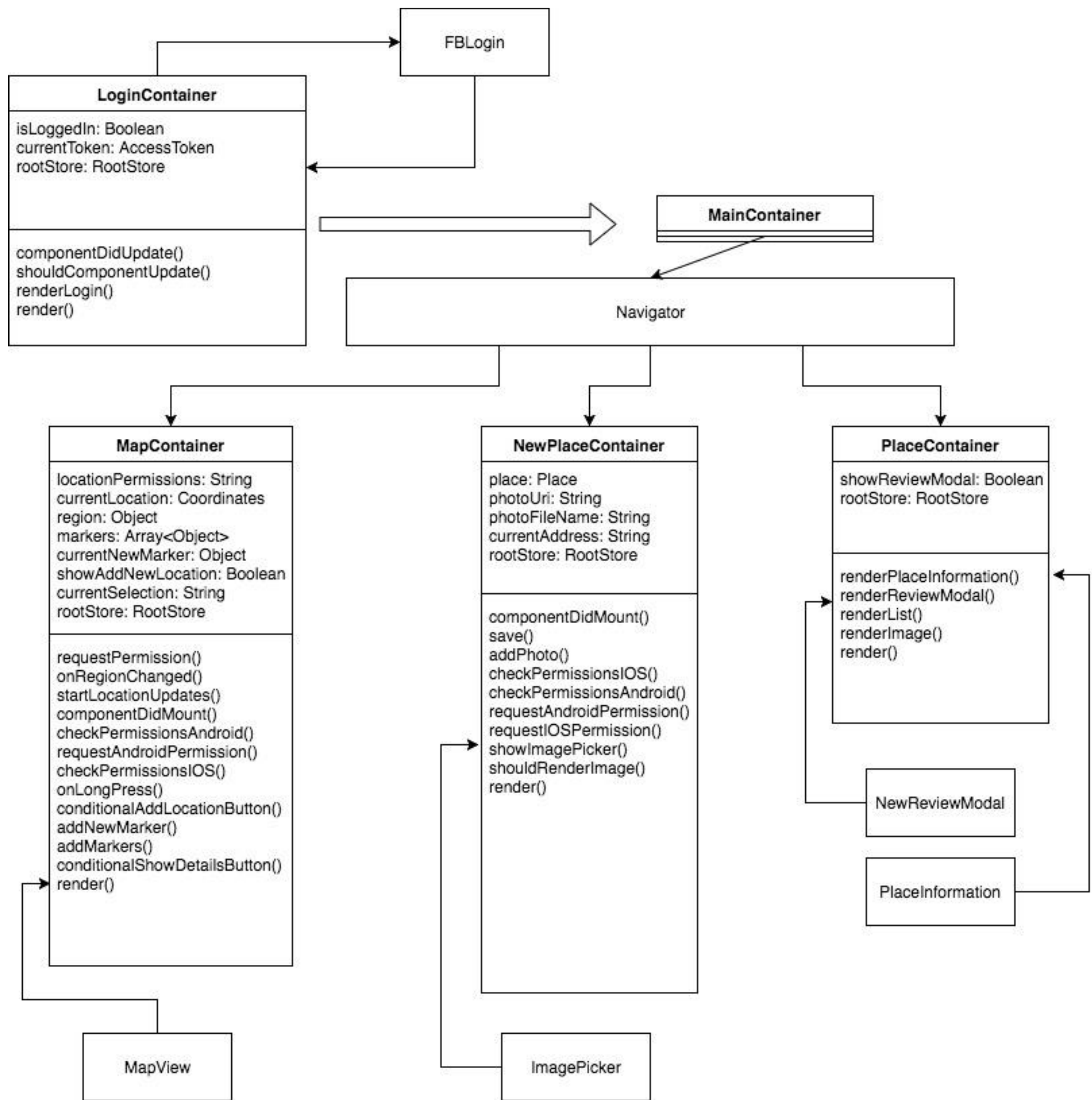
4. What is React Native?, Christina Mercer & Margi Murphy. Verkkoaineisto https://www.techworld.com/apps-wearables/what-is-react-native-3625529/ Kirjoitettu 12.10.2017. luettu 22.1.2018.
5. React Native – Building Mobile Apps with JavaScript, Vladimir Novick. Packt Publishing Ltd. Julkaistu 24.8.2017 luettu 12.3.2018.
6. Web framework rankings. Verkkoaineisto. <https://hotframeworks.com/> luettu 15.4.2018.
7. Firebase palvelun tarjoamat palvelut <https://firebase.google.com/products/> luettu 24.1.2018.
8. Android, Prasanna Kumar Dixit. Kirjoitettu 2014. luettu 25.1.2018.
9. Android announces support for Kotlin, Mike Cleron. Verkkoaineisto. <https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html> 22.1.2018.
10. Swift. Verkkoaineisto <https://developer.apple.com/swift/> Kirjoitettu 1.3.luettu 22.1.2018.
11. Building Progressive web apps: Bringing the Power of Native to the Browser, Tel Ater, sivu 7. Kirjoitettu: 31.8.2017, luettu 22.1.2018.
12. Progressive web apps. Verkkoaineisto. <https://developers.google.com/web/progressive-web-apps/> luettu 22.1.2018
13. Why ”Progressive Web App vs. native” is the wrong question to ask, Dan Dascalescu. Verkkoaineisto <https://medium.com/dev-channel/why-progressive-web-apps-vs-native-is-the-wrong-question-to-ask-fb8555addcbb> Kirjoitettu 24.8.2017. Päivitetty helmikuu 2018. Luettu 22.2.2018.
14. Apache Cordova 3 Programming, John M. Wargo, Addison-Wesley Professional. Julkaistu 2.12.2013. Luettu 22.1.2018.
15. Cisco visual network index: Global mobile data traffic forecast update, Cisco Mobile VNI, 2017. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html Päivitetty 28.3.2017. Luettu 22.1.2018.
16. Understanding REST, Pivotal software. Verkkoaineisto <https://spring.io/understanding/REST> luettu 22.1.2018.
17. Number of smartphone users worldwide from 2015 to 2020 (in billions), Statistia. Verkkoaineisto. <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/> luettu 22.1.2018.
18. Number of available apps in the iTunes App Store from 2008 to 2017 (in 1,000s), Statistia. Verkkoaineisto. <https://www.statista.com/statistics/268251/number-of-apps-in-the-itunes-app-store-since-2008/> luettu 22.1.2018.

19. Serverless Architectures, Mike Roberts. Verkkoaineisto. <https://martinfowler.com/articles/serverless.html> Kirjoitettu 04.08.2016. Luettu 2.2.2018.
20. NoSQL Databases Explained, MongoDB. Verkkoaineisto. <https://www.mongodb.com/nosql-explained/>. Luettu 24.1.2018.
21. Firebase authentication, Google Developers. Verkkoaineisto <https://firebase.google.com/docs/auth/> Päivitetty 13.4.2018. Luettu 15.4.2018.
22. Cloud Functions for Firebase, Google Developers. Verkkoaineisto <https://firebase.google.com/docs/functions> Päivitetty 3.4.2018. Luettu 24.1.2018.
23. Who’s using React Native?, Facebook Open Source. Verkkoaineisto <http://facebook.github.io/react-native/showcase.html>. Luettu 24.1.2018.
24. React.js, Facebook Open Source. Verkkoaineisto <https://reactjs.org/>. Luettu 24.1.2018.
25. React.Component, Facebook Open Source. Verkkoaineisto. <https://reactjs.org/docs/react-component.html>. Luettu 24.1.2018.
26. Dokumenttioliomalli eli DOM, 2kmediat.com. Verkkoaineisto. http://www.2kmediat.com/dhtml/dokumenttimalli.asp. Luettu 12.4.2018.
27. React.Component constructor(), Facebook Open Source. Verkkoaineisto. <https://reactjs.org/docs/react-component.html#constructor > Luettu 12.3.2018.
28. Three principles, Redux.js. Verkkoaineisto https://redux.js.org/docs/introduction/ThreePrinciples.html. Luettu 12.1.2018.
29. Beginning functional JavaScript – Functional Programming with JavaScript using EcmaScript 6, Anto Aravinth, Apress. Julkaistu: 7.3.2017, Luettu 13.4.2018.
30. MobX, mweststrate. Verkkoaineisto <https://mobx.js.org/getting-started.html>. Luettu 31.1.2018.
31. Native Modules, Facebook Open Source. Verkkoaineisto <https://facebook.github.io/react-native/docs/native-modules-ios.html>. Luettu 29.1.2018.
32. About Node.js, Joyent. Verkkoaineisto. <https://nodejs.org/en/>. Luettu 28.1.2018
33. Yarn: A new package manager for JavaScript, Sebastian McKenzie, Christoph Nakazawa, Jamie Kyle. Verkkoaineisto. https://code.facebook.com/posts/1840075619545360. Kirjoitettu 11.10.2016. Luettu 13.4.2018.
34. Watchman, Facebook Open Source. Verkkoaineisto. https://facebook.github.io/watchman/ luettu 28.1.2018.

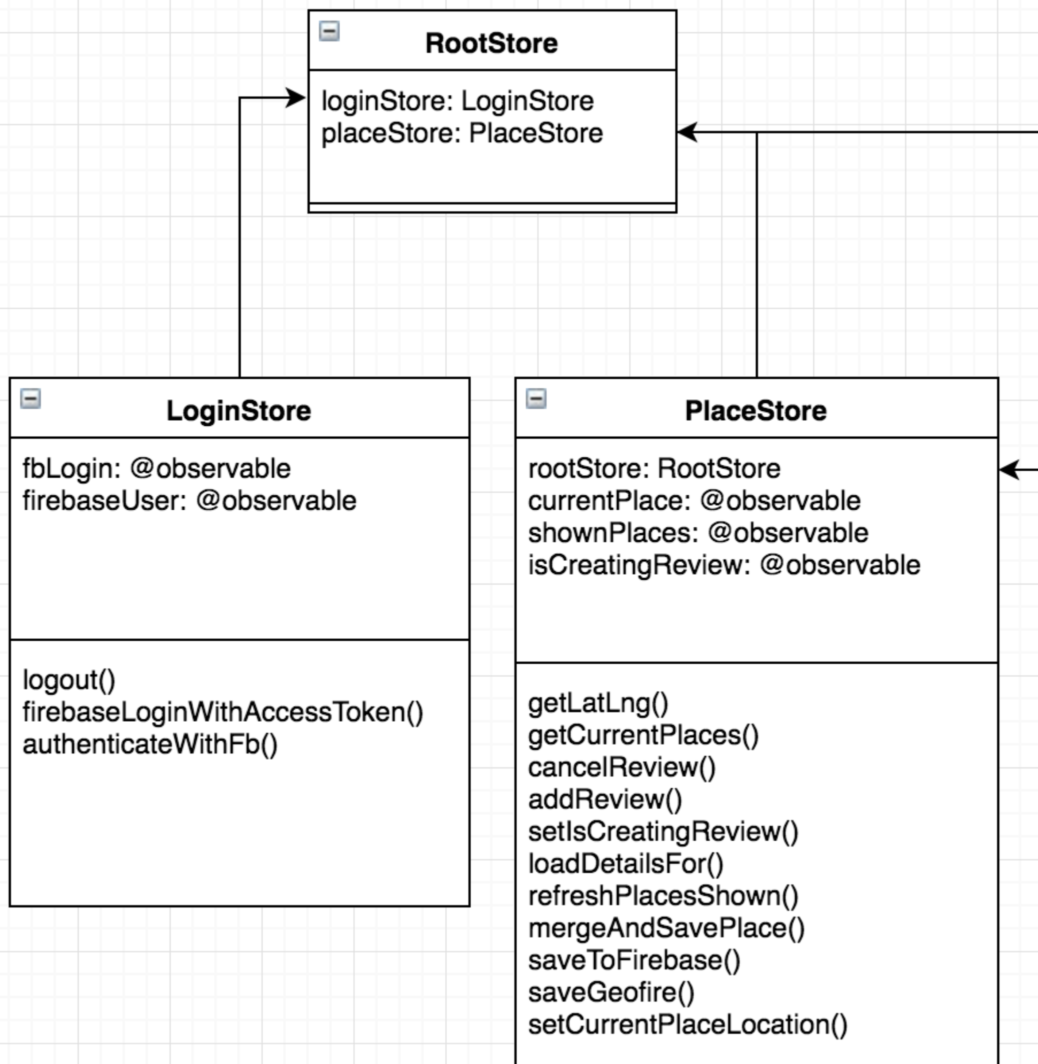
35. Bridging in React Native, Tadeu Zagallo. Verkkoaineisto. <https://tadeuzagallo.com/blog/react-native-bridge/> Kirjoitettu 14.10.2015, luettu 28.1.2018.
36. What is Babel, and how will it help you write JavaScript, Nicholas Johnson. Verkkoaineisto http://nicholasjohnson.com/blog/what-is-babel/. Luettu 15.4.2018.
37. ESLint https://eslint.org/ luettu 28.1.2018.
38. Snapshot testing, Jest. Facebook Open Source. Verkkoaineisto https://facebook.github.io/jest/docs/en/snapshot-testing.html luettu 15.4.2018.
39. Better List Views in React Native, Spencer Ahrens. Verkkoaineisto. https://facebook.github.io/react-native/blog/2017/03/13/better-list-views.html Kirjoitettu 13.3.2017. Luettu 15.2.2018.
40. Most loved, dreaded, and wanted, Stack Overflow. Verkkoaineisto. https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted Luettu 15.3.2018.

Liite 1: Säiliökaavio

Kaaviossa kuvataan säiliöt, niiden tilamuuttujat ja kutsutut funktiot sekä säiliöiden suhteet.

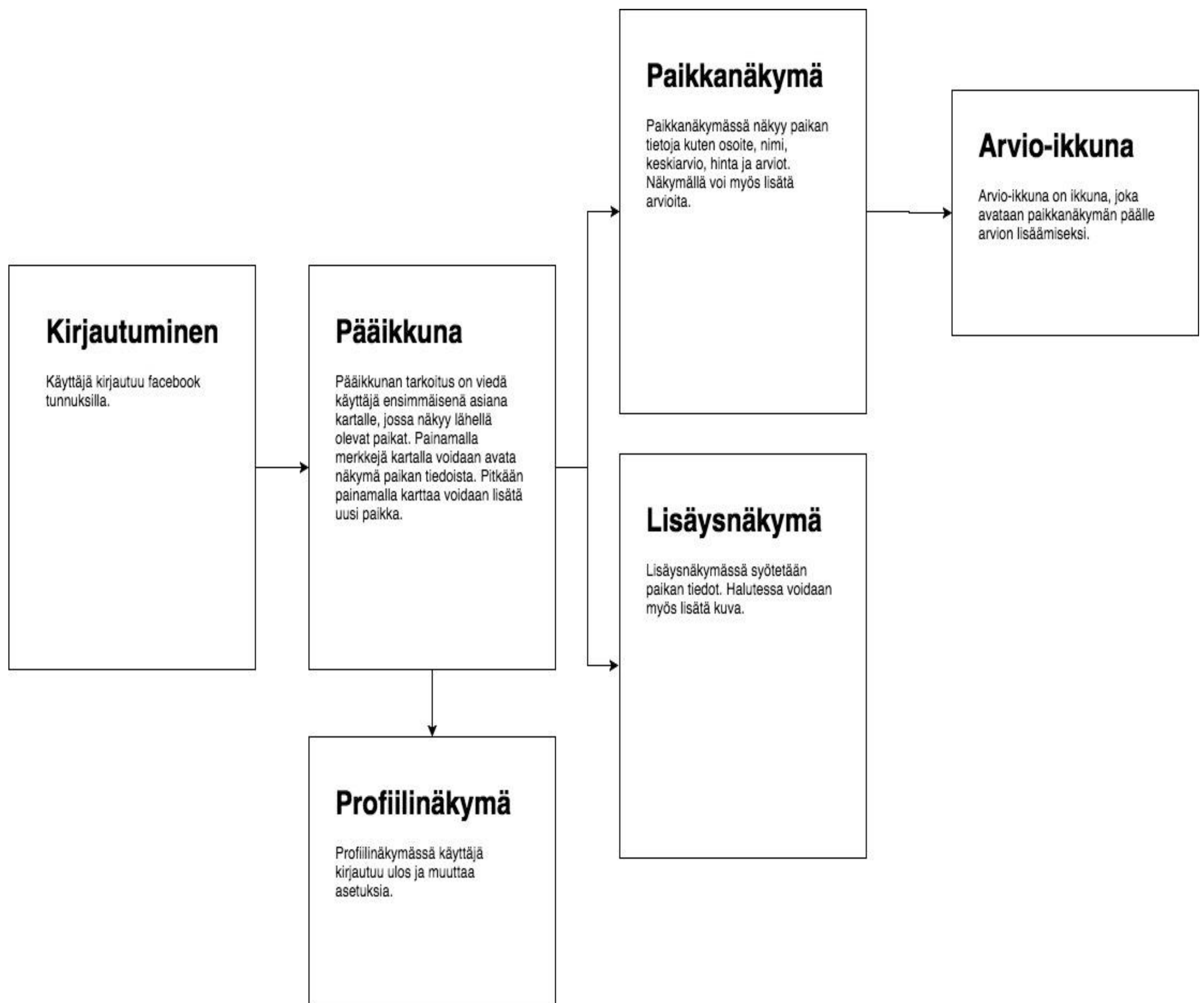

FBLogin
LoginContainer
isLoggedIn: Boolean
currentToken: AccessToken
rootStore: RootStore
componentDidUpdate()
shouldComponentUpdate()
renderLogin()
render()
MainContainer
Navigator
MapContainer
locationPermissions: String
currentLocation: Coordinates
region: Object
markers: Array<Object>
currentNewMarker: Object
showAddNewLocation: Boolean
currentSelection: String
rootStore: RootStore
requestPermission()
onRegionChanged()
startLocationUpdates()
componentDidMount()
checkPermissionsAndroid()
requestAndroidPermission()
checkPermissionsIOS()
onLongPress()
conditionalAddLocationButton()
addNewMarker()
addMarkers()
conditionalShowDetailsButton()
render()
NewPlaceContainer
place: Place
photoUri: String
photoFileName: String
currentAddress: String
rootStore: RootStore
componentDidMount()
save()
addPhoto()
checkPermissionsIOS()
checkPermissionsAndroid()
requestAndroidPermission()
requestIOSPermission()
showImagePicker()
shouldRenderImage()
render()
PlaceContainer
showReviewModal: Boolean
rootStore: RootStore
renderPlaceInformation()
renderReviewModal()
renderList()
renderImage()
render()
NewReviewModal
PlaceInformation
MapView
ImagePicker

Liite 2: Tilavarastokavio



Liite 3: Käyttöliittymän rakenne

Käyttöliittymän rakennekuvaaja



Liite 4: PlaceStoren paikkojenhakufunktio

Funktio, jonka avulla haetaan paikkoja kartalle. haku hakee pisteestä x 10 kilometrin säteellä olevat paikat. Geofiren avulla voidaan suorittaa paikkatietoihin perustuvia hakuja ja luoda tietueisiin paikkatietoparametreja, joilla haku onnistuu. Tulokset lisätään shownPlaces -muuttujaan mikäli se ei vielä sisällä kyseistä paikkaa. Lisäksi, jos paikka on poistunut säteeltä, poistetaan se myös shownPlaces -muuttujasta.

```
@action
refreshPlacesToShow: (latlng: any) => void = latlng => {
  const tempData = [];
  const geofire = new Geofire(firebase.database().ref('geofire'));
  const geoquery = geofire.query({
    center: latlng,
    radius: 10
  });
  const entered = geoquery.on('key_entered', (key, location, distance) => {
    firebase
      .database()
      .ref(`locations/${key}`)
      .on('value', snapshot => {
        const value = snapshot.val();
        value.id = snapshot.key;
        if (!R.contains(value, this.shownPlaces)) {
          this.shownPlaces.push({ id: value.id, value });
        }
      });
  });
  const ready = geoquery.on('ready', () => {});
  const onKeyExitedRegistration = geoquery.on('key_exited', function(
    key,
    location,
    distance
  ) {
    this.shownPlaces = R.reject(n => n.id === key, this.shownPlaces);
  });

  const onKeyMovedRegistration = geoquery.on('key_moved', function(
    key,
    location,
    distance
  ) {
     let index = this.shownPlaces.findIndex((n => n.id == key))
     this.shownPlaces[index].latitude = location[0]
     this.shownPlaces[index].longitude = location[1]
  });
};
```

Liite 5: MergeAndSave -funktio

Funktio ottaa parametreina paikan, kuvatiedoston polun, kuvatiedoston nimen sekä paikan osoitteen. Paikkaan yhdistetään koordinaattitiedot, lisätään käyttäjän tiedot sekä muut tarpeelliset tiedot. Tiedot lähetetään taustapalvelulle kahdessa osassa, sillä

kuvatiedosto lähetetään tiedostojen talletukseen tarkoitettuun cloud storageen ja paikan tiedot talletetaan tietokantaan.

```
@action
mergeAndSavePlace: (
  place: Place,
  imageUri: string,
  fileName: string,
  address: string
) => void = (place, imageUri, fileName, address) => {
  if (place) {
    place.latitude = this.currentPlace ? this.currentPlace.latitude : 0;
    place.longitude = this.currentPlace ? this.currentPlace.longitude : 0;
  }

  this.currentPlace = place;
  this.currentPlace.address = address;
  this.currentPlace.addedBy = {
    uid: this.rootStore.loginStore.firebaseUser.uid,
    displayName: this.rootStore.loginStore.firebaseUser.displayName,
    profilePicture: this.rootStore.loginStore.firebaseUser.photoURL
  };

  this.currentPlace.rating = 0;

  if (!this.currentPlace.price) this.currentPlace.price = 0;
  this.currentPlace.ratingCount = 0;
  this.currentPlace.reviews = [];
  this.currentPlace.addedAt = Date.now();
  if (imageUri) {
    firebase
      .storage()
      .ref(`images/${fileName}`)
      .putFile(imageUri, { contentType: 'image/jpeg' })
      .then(uploadedFile => {
        this.currentPlace.imageUrl = uploadedFile.downloadUrl;
        this.saveToFirebase();
      });
  } else {
    this.saveToFirebase();
  }
};
```