Mobiilisovelluksen kehittäminen React Native -sovelluskehyksellä

Aleksi Vuorela

Opinnäytetyö
Joulukuu 2018
Tekniikan ja liikenteen ala
Insinööri (AMK), ohjelmistotekniikan tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Vuorela, Aleksi	Opinnäytetyö, AMK	Joulukuu 2018
	Sivumäärä 72	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Mobiilisovelluksen kehittäminen React Native -sovelluskehyksellä

Tutkinto-ohjelma
Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)
Rantala, Ari

Toimeksiantaja(t)
Meiko Oy

Tiivistelmä

Opinnäytetyön tavoitteena oli toteuttaa mobiilisovellus Sopimustieto.fi-palvelulle. Sopimustieto.fi on sopimista helpottava verkkopalvelu, jossa voidaan laatia, allekirjoittaa ja arkistoida sopimuksia sähköisesti. Mobiilisovelluksen avulla palvelun käyttö on entistä helpompaa mobiililaitteilla ja sovellus tarjoaa palvelulle mahdollisuuden hyödyntää mobiililaitteiden ominaisuuksia, kuten push-ilmoituksia.

Mobiilisovelluksella palvelun käyttäjät voivat selata ja hallita omaa sopimusarkistoaan sekä allekirjoittaa sopimuksia sähköisesti piirtämällä allekirjoituksen sormella laitteen näytölle. Allekirjoituskutsu saapuu käyttäjän laitteeseen push-ilmoituksena. Sovelluksessa on chatti, jonka avulla osapuolet voivat keskustella sopimuksen sisällöstä sen luontivaiheessa. Sovelluksella voidaan myös ostaa erilaisia tietopalveluja, kuten luottotietojen tarkistus, sekä hallita käyttäjän ja yrityksen asetuksia.

Työssä toteutettiin iOS- ja Android-sovellukset käyttäen React Native -sovelluskehystä, joka mahdollistaa yhteisen koodikannan molemmille alustoille. Sovelluksen tilanhallintaan käytettiin Redux-kirjastoa ja erilaiset sivuvaikutukset, kuten asynkroniset kutsut, käsiteltiin Reduxin apukirjastolla Redux-Sagalla. Push-ilmoituksiin käytettiin Googlen Firebase Cloud Messaging -palvelua. Reaaliaikainen chatti toteutettiin WebSocketien avulla käyttäen Pusher-palvelua.

Työn tuloksena toteutettiin valmiit mobiilisovellukset iOS- ja Android-alustoille, jotka täyttivät asiakkaan vaatimukset. Sovellukset julkaistiin Applen ja Googlen sovelluskaupoissa. Sovelluksen kehitys jatkuu verkkopalvelun kehityksen rinnalla hyödyntämällä palvelussa yhä enemmän mobiililaitteiden tarjoamia mahdollisuuksia.

Avainsanat (asiasanat)
React Native, React, Redux, JavaScript, mobiilikehitys

Muut tiedot

Description

Author(s) Vuorela, Aleksi	Type of publication Bachelor’s thesis	Date December 2018
		Language of publication: Finnish
	Number of pages 72	Permission for web publication: x

Title of publication
Mobile application development with React Native

Degree programme
Software Engineering

Supervisor(s)
Rantala, Ari

Assigned by
Meiko Oy

Abstract

The objective of the thesis was to implement a mobile application for Sopimustieto.fi service. Sopimustieto.fi is a web service designed to help with contract management. Contracts can be created, signed and archived digitally in the service. The mobile application makes the service easier to use on mobile devices and it provides the service with an opportunity to utilize the features of mobile devices, such as push notifications.

With the mobile application, users can browse and manage their contract archive and sign contracts digitally by drawing their signature on the mobile device’s screen. The invitation to sign a contract arrives to the user’s mobile device as a push notification. The mobile application includes a chat where the parties can discuss the contents of the contract during its creation. Different information services can also be purchased with the mobile application, such as credit information and user’s and company’s settings can be managed with it.

During the writing of the thesis, iOS and Android applications were developed with React Native framework that allows shared codebase for both platforms. The application’s state management was implemented with Redux library and different side effects were handled with Redux-Saga library. Push-notifications were implemented with Google’s Firebase Cloud Messaging service. Chat was created with WebSockets using Pusher service.

As a result of the thesis, finished mobile applications were created for iOS and Android platforms thus meeting the defined requirements. The mobile applications were released in Apple’s and Google’s application stores. The mobile application’s development continues by utilizing more mobile devices’ features in the service.

Keywords/tags (subjects)
React Native, React, Redux, JavaScript, mobile application development

Miscellaneous

Sisältö

Sanasto **5**

1 Työn lähtökohdat **8**

1.1 Toimeksiantaja 8

1.2 Tavoitteet 8

2 Vaatimukset ja rajaukset **8**

2.1 Vaatimukset 8

2.2 Rajaukset 10

3 Tekniikat ja työkalut **11**

3.1 JavaScript 11

3.2 React 12

3.3 React Native 13

3.4 Redux 14

3.5 Redux-Saga 17

3.6 Firebase Cloud Messaging 20

4 Sovelluksen toteutus **21**

4.1 Kehitysympäristö 21

4.1.1 Lähtökohdat 21

4.1.2 iOS 22

4.1.3 Android 23

4.2 Projektin kansiorakenne 24

4.3 Komponentit 25

4.4 JSX ja tyylit 30

4.5 Komponenttikirjastot 32

4.6 Tilanhallinta 33

4.6.1 Yleistä ... 33

4.6.2 Storen luonti ... 33

4.6.3 Storen tallennus laitteen muistiin ... 35

4.6.4 Komponentin kytkeminen storeen React-Reduxilla ... 36

4.6.5 Tiedonhaku storesta selector-funktioilla ... 36

4.6.6 Actionit ... 37

4.6.7 Reducerit ... 39

4.7 Sivuvaikutukset ... 41

4.8 Navigointi ... 45

4.9 Push-ilmoitukset ... 48

4.10 WebView ... 53

4.11 Chatti ... 54

4.12 Työkalut ... 56

5 Tulokset ... 57

5.1 Vaatimusten täyttyminen ... 57

5.2 Julkaisu ... 60

6 Pohdinta ... 61

6.1 Kehitysvauhti ... 61

6.2 React ja teknologiavalinnat ... 61

6.3 Käyttöliittymät ... 62

6.4 Haasteet ... 63

6.5 Yhteenveto ja jatkokehitys ... 64

Lähteet ... 65

Liitteet ... 67

Liite 1. Kirjautumisnäkymä ... 67

Liite 2. Arkiston listausnäkymä ... 68

Liite 3. Yksittäisen sopimuksen näkymä ... 69
Liite 4. Tietopalveluiden näkymä ... 70
Liite 5. Yritystietopalveluiden näkymä ... 71
Liite 6. Asetusten päänäkymä ... 72

Kuviot

Kuvio 1. Imperatiivisesti rakennettu DOM ... 12
Kuvio 2. Deklaratiivisesti rakennettu DOM ... 13
Kuvio 3. Esimerkki action ... 15
Kuvio 4. Esimerkki reducer ... 16
Kuvio 5. Reduxin yhdensuuntainen datan kulku ... 16
Kuvio 6. Reduxin yhdensuuntainen datan kulku Redux-Sagan kanssa ... 17
Kuvio 7. Esimerkki generaattorifunktio ... 18
Kuvio 8. Esimerkki sagat ... 18
Kuvio 9. Esimerkki sagasta, joka hakee dataa asynkyronisesti ... 19
Kuvio 10. Esimerkki sagan testaamisesta ... 20
Kuvio 11. React Nativen generoima kansiorakenne ... 22
Kuvio 12. Koodin allekirjoittamisen konfigurointi ... 23
Kuvio 13. Projektin kansiorakenne ... 24
Kuvio 14. PDF-tiedoston hakemisen logiikan kapsuloiva säiliökomponentti ... 27
Kuvio 15. PDF-tiedoston renderöivä esityksellinen komponentti ... 28
Kuvio 16. Komponentin elinkaaren aikana suoritettavat metodit ... 29
Kuvio 17. Funktionaalinen komponentti tilin valitsemiseen ... 30
Kuvio 18. Esimerkki tyylien määrittelystä JavaScript-objektina ... 31
Kuvio 19. Sopimusten listauksessa käytetyt NativeBase-komponentit ... 32
Kuvio 20. Redux-storen luominen ... 33
Kuvio 21. Sovelluksen juurireducer ... 34
Kuvio 22. Määrittely laitteen muistiin tallennettavista storen osista ... 35
Kuvio 23. Selector-funktiot käyttäjän kaikkien sopimusten hakemiseen storesta ... 36
Kuvio 24. Action, jolla haetaan sopimuksen PDF-tiedosto ... 37

Kuvio 25. Flux Standrad Action 38
Kuvio 26. Sopimuksen PDF-tiedoston asettaminen storeen reducerissa 39
Kuvio 27. Oletustilan antaminen reducer-funktiolle.. 40
Kuvio 28. Contracts-reducerin oletustila 40
Kuvio 29. Juurisagan käynnistäminen 42
Kuvio 30. Sovelluksen juurisaga 42
Kuvio 31. Sopimuksen PDF-tiedoston hakeminen API:lta sagassa 43
Kuvio 32. API-kutsun suorittava funktio.. 44
Kuvio 33. Sovelluksen juurinavigaattori 45
Kuvio 34. Kirjautuneen käyttäjän navigaattori... 46
Kuvio 35. Viittaus sovelluksen juurinavigaattorista navigator-servicelle 47
Kuvio 36. Navigator-servicen navigointi-funktio.. 47
Kuvio 37. FCM:än alustaminen.. 49
Kuvio 38. Lukemattomien ilmoitusten hakeminen ja FCM-päivitysavainkuuntelija ... 49
Kuvio 39. Ilmoituksen avaaminen sovelluksen ollessa suljettuna 50
Kuvio 40. Ilmoitusten kuuntelu sovelluksen ollessa taustalla tai avoinna.................. 51
Kuvio 41. Ilmoitusten näkymä.. 52
Kuvio 42. Allekirjoitusnäkymä 53
Kuvio 43. Pusherin alustaminen 54
Kuvio 44. Chat-näkymä.. 55
Kuvio 45. React Native Debugger.. 56

Taulukot

Taulukko 1. React-elementtien React Native -vastineita.. 26

Sanasto

Android

Googlen kehittämä Linux-pohjainen käyttöjärjestelmä mobiililaitteille.

Android Studio

Ohjelmointiympäristö Android-käyttöjärjestelmälle.

API

Application Programming Interface. Ohjelmointirajapinta, jonka avulla eri ohjelmat voivat keskustella keskenään.

APK

Android Package Kit. Android-sovellusten tiedostotyyppi.

CSS

Cascading Style Sheets. Tyyliohjeiden laji, jolla määritellään ulkoasu HTML-kielellä kuvatulle rakenteelle.

Debuggaus

Ohjelman virheellisen toiminnan aiheuttaneen virheen paikallistamista ja korjaamista.

DOM

Document Object Model. Ohjelmointirajapinta HTML-dokumenteille, jonka avulla ohjelmat voivat muuttaa dokumentin rakennetta, tyyliä ja sisältöä.

ECMAScript

Ecma International -organisaation kehittämä JavaScript-standardi.

HTML

Hypertext Markup Language. Merkintäkieli, jolla kuvaillaan web-sivujen rakenne.

iOS

Applen kehittämä käyttöjärjestelmä mobiililaitteille.

Java

Yleiskäyttöinen ohjelmointikieli, joka on laitteistoriippumaton ja oliopohjainen.

JavaScript

Ohjelmointikieli, jota käytetään erityisesti web-sivuilla dynaamisen toiminnallisuuden toteuttamiseen. React Native käyttää JavaScriptiä ohjelmointikielenä.

JDK

Java Developement Kit. Ohjelmistokehityspaketti Javalle.

Mounttaus

Mounttaus tarkoittaa tässä yhteydessä React-komponentin asettamista DOMiin.

MVC-arkkitehtuuri

Model-View-Controller-arkkitehtuuri. Ohjelmistoarkkitehtuurityyli, jossa ohjelma jaetaan kolmeen osaan: malliin, näkymään ja käsittelijään.

Natiivisovellus

Sovellus, joka on toteutettu alustan omalla kehitysympäristöllä ja ohjelmointikielellä.

Node.js

Alustariippumaton ajoympäristö JavaScriptille, jolla JavaScript koodia voidaan suorittaa palvelimella.

NPM

Node Package Manager. Paketinhallintajärjestelmä Node.js:lle.

Objective-C

Applen käyttämä ohjelmointikieli, joka on oliolaajennus C-ohjelmointikieleen.

PDF

Portable Document Format. Tiedostomuoto, jota käytetään sähköiseen julkaisemiseen, tulostamiseen ja painamiseen.

Promise

JavaScript-olio, jota käytetään asynkronisten kutsujen yhteydessä.

React

Facebookin kehittämä JavaScript-kirjasto web-käyttöliittymien tekoon.

Refaktorointi

Ohjelmakoodin muuttamista siten, että sen toiminnallisuus säilyy ennallaan, mutta sisäinen rakenne muuttuu.

Renderöinti

Renderöinti tarkoittaa tässä yhteydessä elementin piirtämistä päätelaitteen näytölle.

Responsiivisuus

Verkkosivusto mukautuu kaikille päätelaitteille sopivaksi.

SDK

Software development kit. Kokoelma kehitystyökaluja tietylle ohjelmistopaketille, sovelluskehykselle tai vastaavalle kehitysalustalle.

WebSocket

Verkkotekniikka, jolla voidaan muodostaa asiakkaan ja palvelimen välille jatkuva kaksisuuntainen yhteys.

Xcode

Applen kehittämä ohjelmointiympäristö macOS:lle, joka tukee useita alustoja ja ohjelmointikieliä.

XML

Extensible Markup Language. Merkintäkieli, jolla datan joukkoon voidaan lisätä sen merkitystä kuvaavaa tietoa.

XMPP

Extensible Messaging and Presence Protocol. Avoin standardoitu pikaviestintäprotokolla.

1 Työn lähtökohdat

1.1 Toimeksiantaja

Työn toimeksiantajana toimi Meiko Oy. Meiko on vuonna 2011 perustettu jyväskyläläinen ohjelmistotalo, joka kehittää liiketoimintaa ja työntekoa tukevia järjestelmiä yritysten yksilöllisiin tarpeisiin. Meikon tavoitteena on tuottaa arvoa teknisille investoinneille, joita kasvu ja prosessien nykyaikaistaminen edellyttävät. (Me 2018.)

1.2 Tavoitteet

Opinnäytetyön tavoitteena oli toteuttaa mobiilisovellus Sopimustieto.fi-palvelulle, joka on Meikon asiakasprojekti. Sopimustieto.fi on sopimista helpottava verkkopalvelu. Palvelussa onnistuu sopimusten laatiminen, allekirjoittaminen, arkistoiminen ja sopimuskumppanin luotettavuuden tarkistaminen.

Nykyisin palvelua käytetään responsiiviselta verkkosivulta, mutta palvelulle haluttiin mobiilisovellus. Mobiilisovelluksen avulla palvelun käyttö on entistä helpompaa mobiililaitteilla, ja sovellus tarjoaa palvelulle mahdollisuuden hyödyntää mobiililaitteiden ominaisuuksia, kuten push-ilmoituksia.

Opinnäytetyön kirjallisen osuuden tavoitteena oli tarkastella mobiilisovelluksen kehitystä käytännönläheisesti projektin käynnistämisestä julkaisuun saakka sekä käydä läpi kehitystyössä vastaan tulleita ongelmia ja niiden ratkaisuja.

2 Vaatimukset ja rajaukset

2.1 Vaatimukset

Mobiilisovelluksesta tuli olla sekä iOS- että Android-versio ja se tuli toteuttaa React Native -sovelluskehyksellä. React Nativen valintaan oli kaksi syytä: ensinnäkin palvelun web-sovellus on toteutettu Reactilla, joten mobiilisovelluksessa voitiin hyödyntää aikaisempaa koodia. Toiseksi, React Native mahdollistaa yhteisen

koodikannan sekä iOSille että Androidille, minkä takia kehitys on nopeampaa ja vaatii vähemmän resursseja.

Mobiilisovellukseen tuli toteuttaa suurin osa web-sovelluksen toiminnoista, jotka esitellään seuraavaksi. Pois rajatut toiminnot esitellään luvussa 2.2.

Arkisto

Arkistoon tallentuvat palvelussa laaditut sopimukset sekä palveluun tuodut, muualla laaditut PDF-tiedostot. Arkistossa on kaksi osaa: sopimusten listaus ja yksittäisen sopimuksen avaaminen. Listaa piti pystyä suodattamaan ja järjestämään. Listassa tuli myös olla hakutoiminto. Yksittäisellä sopimuksella tuli olla mahdollisuus avata sopimuksen PDF-tiedosto. Yksittäisen sopimuksen arkistotietoja piti pystyä muokkaamaan, sekä kopioimaan sopimus toiselle tilille ja poistamaan se. Sopimus piti myös pystyä lähettämään sähköpostiin.

Allekirjoittaminen

Sovelluksessa tuli olla mahdollisuus allekirjoittaa sopimuksia sähköisesti, mikä tapahtuu piirtämällä allekirjoitus sormella mobiililaitteen näytölle. Ennen allekirjoitusta tuli olla mahdollista esikatsella sopimuksen PDF-tiedosto sekä valita tili, jolle allekirjoitettu sopimus tallennetaan.

Tietopalvelut

Tietopalvelut sisältävät erilaisia palveluja sopimuskumppanin luotettavuuden tarkistamiseen. Tietopalveluissa on kaksi osaa: yritystietopalvelut ja henkilötietopalvelut. Yritystietopalveluissa tuli olla yrityshaku sekä toiminnot hakea erilaisia yrityksen tietoja, kuten luottotiedot ja kaupparekisteriote. Henkilötietopalveluissa tuli olla toiminto hakea yksityishenkilön luottotiedot.

Chat

Sopimuksen luontivaiheessa on mahdollisuus keskustella sopimuksen sisällöstä osapuolten kanssa chatissa. Chatissa tuli olla mahdollisuus lähettää viestejä reaaliajassa muille osapuolille sekä samalla nähdä sopimuksen PDF-tiedosto, jonka sisältö päivittyy sitä mukaan kun sopimusta laaditaan.

Ilmoitukset

Ilmoituksia lähetetään esimerkiksi, kun kutsutaan osapuolia allekirjoittamaan sopimus. Ilmoituksista tuli saapua push-ilmoitus mobiililaitteeseen, ja ne tuli myös listata omassa näkymässään. Avaamalla ilmoituksen tuli tapahtua määritelty toiminto, kuten sopimuksen allekirjoitustilanteeseen siirtyminen.

Kirjautuminen

Sovellukseen piti pystyä kirjautumaan sisään ja vaihtamaan tiliä. Kirjautumisen tuli tallentua laitteen muistiin, jolloin sitä ei tarvitse tehdä joka kerta uudestaan, kun sovellus käynnistetään. Kirjautumisessa tuli olla kaksivaiheinen tunnistautuminen.

Asiakaspalvelu

Sovelluksessa tuli olla mahdollisuus keskustella asiakaspalvelun kanssa reaaliaikaisessa chatissa käyttäen Intercom-alustaa. Asiakaspalvelun viesteistä tuli saapua push-ilmoitus käyttäjälle.

Asetukset

Asetuksissa tuli olla seuraavat toiminnot:

- Perustietojen muokkaus
- Sähköpostin, puhelinnumeron ja salasanan vaihtaminen
- Allekirjoituksen vaihtaminen
- Kuittien näyttäminen
- Kansioiden ja ryhmien hallinta
- Yrityksen käyttäjien ja roolien hallinta
- Yrityksen tilauksen hallinta.

2.2 Rajaukset

Sopimusten laatiminen rajattiin pois mobiilisovelluksesta toiminnon laajuuden takia. Omien PDF-asiakirjojen tuonti jätettiin pois, mutta jatkokehitykseen suunniteltiin mobiililaitteen kameran hyödyntämistä paperisten sopimusten skannaamiseen sähköiseen muotoon. Rekisteröityminen palveluun jätettiin pois sovelluksesta toistaiseksi, sillä yritystilin rekisteröiminen palveluun on maksullista ja Apple asettaa

tiukat säännöt maksamisen suhteen. Apple vaatii, että sovelluksen sisäiset maksut tehdään heidän oman palvelunsa kautta, josta Apple ottaa itselleen 30 % palkkiota.

3 Tekniikat ja työkalut

3.1 JavaScript

JavaScript-ohjelmointikieli sai alkunsa vuonna 1995 keinona lisätä ohjelmia verkkosivuille Netscape Navigator -selaimessa. Kieli on sen jälkeen omaksuttu lähes kaikkiin web-selaimiin, ja se on mahdollistanut modernit web-sovellukset. Kun JavaScript omaksuttiin Netscapen ulkopuolelle, kirjoitettiin standardi kuvaamaan, kuinka Javascript-kielen kuuluu toimia. Kyseistä standardia kutsutaan ECMAScriptiksi, jonka Ecma International -organisaatio kehitti. (Haverbeke 2014, 6-7.)

Crockfordin (2008, 3) mukaan JavaScriptin hyviä ominaisuuksia ovat funktiot, löysä tyypitys, dynaamiset objektit sekä objekti-literaali. JavaScript antaa ohjelmoijalle paljon vapauksia, mikä mahdollistaa sellaisia tekniikoita, joiden toteutus olisi mahdotonta tiukemmissa ohjelmointikielissä. Tämä vapaus kuitenkin aiheuttaa myös sen, että virheiden löytäminen ohjelmasta vaikeutuu, sillä järjestelmä ei pysty aina osoittamaan niitä ohjelmoijalle. (Haverbeke 2014, 7.)

JavaScriptistä on kehitetty useita versioita. ECMAScriptin kolmas versio oli laajalti tuettu ajalla, kun JavaScript nousi valta-asemaansa vuosien 2000 ja 2010 välillä. Samoihin aikoihin kehitettiin neljättä versiota, johon suunniteltiin suuria parannuksia ja laajennuksia. Muutokset olivat kuitenkin liian radikaaleja, minkä vuoksi neljäs versio hylättiin vuonna 2008. Tämä johti vähemmän kunnianhimoisen viidennen version julkaisuun vuonna 2009. (Haverbeke 2014, 7-8.)

ECMAScriptin versio, jolla tämä työ on tehty, on vuonna 2015 julkaistu kuudes versio (ES6). Kyseisessä versiossa kieleen tuli paljon uusia ominaisuuksia, joita käytetään laajalti tässä työssä (mm. luokat, moduulit, nuoli- ja generaattorifunktiot). Web-selaimet eivät tue vielä kaikkia uusimpia ominaisuuksia, mutta työkalujen avulla lähdekoodi voidaan kääntää yhteensopivaksi aikaisemman ECMAScriptin version kanssa. Tähän tarkoitukseen työssä käytettiin Babel-nimistä työkalua.

3.2 React

React on suosittu JavaScript-kirjasto web-käyttöliittymien tekoon. Facebook kehitti sen ratkaisemaan haasteita, joita tulee vastaan monimutkaisien käyttöliittymien kehityksessä ja joissa esitettävä data muuttuu ajan myötä. React sai alkunsa Facebookin mainonnasta vastaavassa organisaatiossa, jossa hyödynnettiin perinteistä MVC-arkkitehtuuria. Kehittäessään Reactia Facebook ei luonut uutta MVC-arkkitehtuuriin perustuvaa sovelluskehystä korvaamaan edellisiä ratkaisuja. React luotiin ratkaisemaan vain yksittäisen ongelman: datan esittämisen käyttöliittymässä. (Gackenheimer 2015, 1-2.)

Tämän ongelman ratkaisemiseksi React rakennettiin käyttämään deklaratiivisia komponentteja. Deklaratiivisuus ohjelmoinnissa tarkoittaa sitä, että kuvaillaan, mitä ohjelmassa tulee tapahtua sen sijaan, että kuinka se tapahtuu. Tämän vastakohta on imperatiivinen ohjelmointi, jossa ollaan kiinnostuneita vain siitä, kuinka ratkaisuun päästään. (Banks & Porcello 2017, 34.)

Tarkastellaan aluksi tapaa, jossa DOM rakennetaan imperatiivisesti kuvion 1 mukaan.

```
var target = document.getElementById('target');
var wrapper = document.createElement('div');
var headline = document.createElement('h1');

wrapper.id = 'welcome';
headline.innerText = 'Hello World';

wrapper.appendChild(headline);
target.appendChild(wrapper);
```

Kuvio 1. Imperatiivisesti rakennettu DOM

Koodissa luodaan ja asetetaan elementtejä sekä lisätään niitä dokumenttiin. Mikäli koodirivejä olisi kymmeniä tuhansia, tulisi muutosten tekemisestä, toimintojen lisäämisestä ja skaalaamisesta hyvin hankalaa. Tarkastellaan seuraavaksi tapaa, kuinka DOM voidaan rakentaa deklaratiivisesti käyttäen React-komponenttia kuvion 2 mukaan.

```
const Welcome = () => (
  <div id='welcome'>
    <h1>Hello World</h1>
  </div>
)

ReactDOM.render(
  <Welcome />, document.getElementById('target')
)
```

Kuvio 2. Deklaratiivisesti rakennettu DOM

Tässä koodissa *Welcome*-komponentti kuvaa DOMia, joka tulee renderöidä. *Render*-funktio käyttää komponentissa esitettyjä ohjeita rakentaakseen DOMin ja abstraktoi pois kaikki yksityiskohdat siitä, kuinka DOM renderöidään. Koodista nähdään selvästi heti, että tässä halutaan renderöidä *Welcome*-komponentti *target*-elementin sisään.

DOMin renderöintiin React käyttää virtuaalista DOMia, joka on yksi Reactin tärkeimpiä ominaisuuksia. Virtuaalinen DOM rakentuu React elementeistä, jotka näyttävät samoilta kuin perinteiset HTML-elementit, mutta todellisuudessa ne ovat JavaScript-objekteja. JavaScript-objektien käsittely on huomattavasti nopeampaa kuin DOM API:n kanssa työskentely. Muutokset tehdään JavaScript-objekteihin eli virtuaaliseen DOMiin, ja React renderöi nämä muutokset käyttäen DOM API:ia tehokkaasti. Tehokkuus on sen ansiota, että React päivittää DOMia vain muutosten osalta. (Banks & Porcello 2017, 61-62.)

Reactin käyttämät komponentin mahdollistavat saman DOM-rakenteen käyttämisen erilaisten datakokoelmien esittämiseen. Tämän ansiosta koodi on uudelleen-käytettävää ja skaalautuvaa. Jos käyttöliittymässä esitettäisiin esimerkiksi reseptejä, voitaisiin samaa resepti-komponenttia käyttää yhden tai tuhannen eri reseptin esittämiseen.

3.3 React Native

React Native on JavaScript-sovelluskehys, jolla voidaan tehdä natiivisti renderöityjä mobiilisovelluksia iOSille ja Androidille. Se pohjautuu Reactiin, ja sen avulla web-

kehittäjät voivat tehdä tutulla JavaScript-kirjastolla mobiilisovelluksia, jotka näyttävät ja tuntuvat natiiveilta. Lisäksi se mahdollistaa lähes yhteisen koodikannan iOS- ja Android-alustojen välillä. (Eisenman 2017, 1.)

Kuten React web-sovellukset, myös React Native -mobiilisovellukset kirjoitetaan käyttäen JavaScriptin ja XML-tyylisen merkintätavan sekoitusta nimeltä JSX. Taustalla React Nativen käyttämä silta kutsuu natiiveja renderöinti API-rajapintoja käyttäen Objective-C:tä (iOS) tai Javaa (Android). Tämän ansiosta sovellus renderöidään käyttäen oikeita mobiilikäyttöliittymän komponentteja eikä *WebVie*w-säiliöitä. WebView-säiliö on sovellukseen upotettu web-selain, johon perustuvat monet muut mobiilisovellusten kehittämiseen käytetyt sovelluskehykset, kuten Ionic ja Apache Cordova. React Native tarjoaa myös JavaScript-rajapinnat alustojen API-rajapinnoille, joten React Native -sovellukset voivat käyttää alustojen ominaisuuksia, kuten kameraa tai sijaintia. (Mts. 1.)

React Native tukee virallisesti iOS- ja Android-alustoja, mutta yhteisön jäsenet ovat toteuttaneet tuen myös mm. Windowsille ja macOS:lle. React Nativella on toteutettu useita suurten yhtiöiden mobiilisovelluksia, kuten Facebook, Airbnb, Walmart ja Baidu. (Mts. 1.)

3.4 Redux

Redux on Flux-nimiseen suunnittelumalliin pohjautuva kirjasto. Flux on Facebookin kehittämä vaihtoehto MVC-arkkitehtuurille, jossa data kulkee vain yhteen suuntaan. Kun data kulkee sovelluksessa vain yhteen suuntaan, sovelluksen logiikka on helpompi ymmärtää ja ennakoida. Reduxissa koko sovelluksen tila tallennetaan yhteen paikkaan, jota kutsutaan *storeksi*. Tämän ansiosta suurissa sovelluksissa ollaan paremmin tietoisia koko sovelluksen tilasta verrattuna siihen, että jokainen komponentti säilyttäisi oman tilansa. Tila tallennetaan storessa yhteen muuttumattomaan objektiin, jota kutsutaan tilapuuksi. (Banks & Porcello 2017, 183-186.)

Koska tilaobjekti storessa on muuttumaton, päivittäminen tehdään korvaamalla se kokonaan uudella. Tämä päivittäminen tapahtuu *actionien* avulla: actionit sisältävät ohjeita siitä, mitä sovelluksen tilassa tulee muuttaa sekä tarvittavan datan näiden

muutosten tekemiseen. Actionit ovat ainut tapa muuttaa sovelluksen tilaa Reduxissa. Actionien avulla nähdään myös historia siitä, miten sovelluksen tila on muuttunut ajan myötä. Actionilla on aina tyyppi, joka kuvaa sitä, mitä tulee tapahtua. Lisäksi actionilla on usein myös dataa, jota tarvitaan tilan muutoksen tekemiseen. Tätä dataa kutsutaan actionionin *payloadiksi*. Kuviossa 3 on esimerkki actionista, jolla lisätään uusi väri. Tyyppinä actionilla on *ADD_COLOR* ja payloadina ovat ID, värikoodi ja titteli. (Mts. 187-190.)

```
{
  type: "ADD_COLOR",
  id: 'BZ35d0',
  color: '#FFFFFF',
  title: 'Titanium White'
}
```

Kuvio 3. Esimerkki action

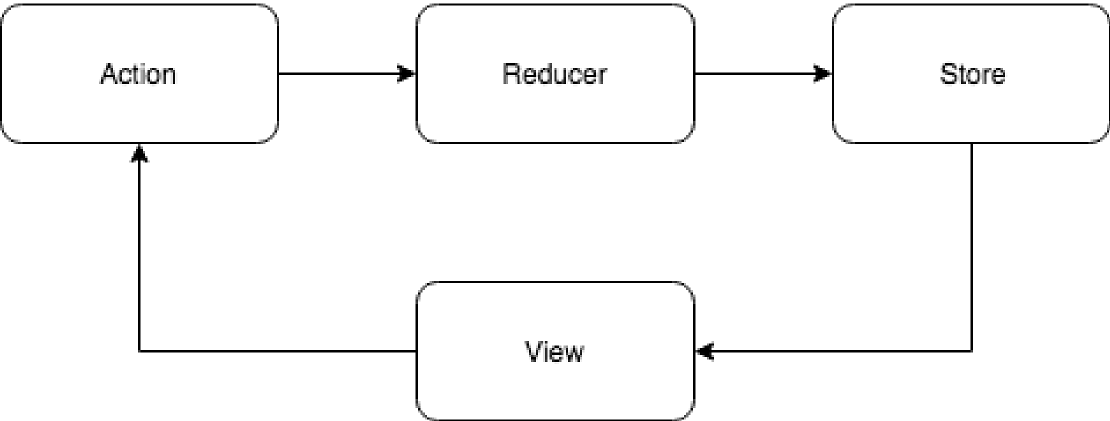
Actionit kuvaavat siis, miten sovelluksen tilan tulee muuttua, mutta muutoksen tekemiseen tarvitaan funktioita. Näitä funktioita kutsutaan *reducereiksi*. Reducerit ovat funktioita, jotka ottavat sovelluksen nykyisen tilan sekä actionin argumentteina ja käyttävät niitä palauttaakseen uuden tilan. Reducerit muuttavat tiettyjä osia tilapuusta, ja ne voidaan yhdistää yhdeksi reduceriksi, joka hallitsee koko sovelluksen tilanmuutoksia. Kuviossa 4 on esimerkki reducerista, joka testaa switch-lausekkeessa actionin tyypin, käsittelee jokaisen eri tyyppisen actionin ja palauttaa uuden tilan. Jos reduceria kutsutaan tuntemattomalla actionilla, palautetaan nykyinen tila. (Mts. 190-195.)

```
export const color = (state = [], action) => {
  switch (action.type) {
    case "ADD_COLOR":
      return [
        ...state,
        {
          id: action.id,
          color: action.color,
          title: action.title
        }
      ];
    default:
      return state;
  }
};
```

Kuvio 4. Esimerkki reducer

Koska tilaa tulee käsitellä muuttumattomana objektina, uutta väriä ei voida lisätä taulukon *push()*-metodilla, vaan luodaan uusi taulukko nykyisten arvojen sekä uuden arvon perusteella.

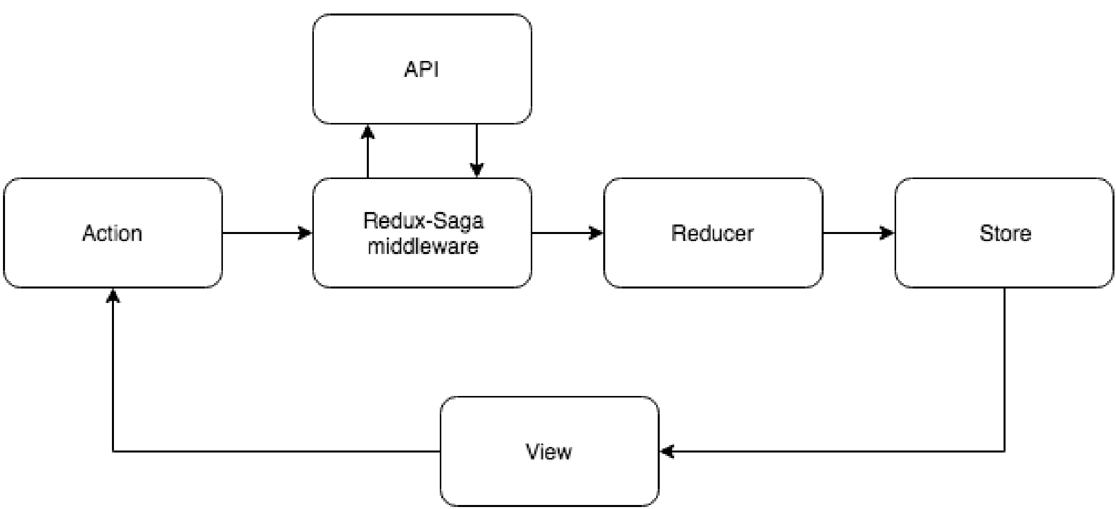
Action lähetetään Reduxin *dispatch(action)*-metodin avulla. Tämä lähetetty action napataan kiinni reducerissa, jossa tilanmuutos tehdään storeen. Nykyinen storen tilapuu saadaan kutsumalla storen *getState()*-metodia. Actionien ja reducereiden avulla data kulkee Reduxissa aina vain yhteen suuntaan, jota havainnollistetaan kuviossa 5.



Kuvio 5. Reduxin yhdensuuntainen datan kulku

3.5 Redux-Saga

Reducerit toimivat synkronisesti, ja niiden toiminnan on oltava ennakoitavissa. Tämän takia erilaisia sivuvaikutuksia, kuten asynkronisia kutsuja, ei voi tehdä niissä. Redux-Saga tarjoaa yhden ratkaisun näiden sivuvaikutusten käsittelyyn. Redux-Saga on apukirjasto Reduxille ja toimii sen *middlewarena* eli osien välisenä rajapintana. Reduxin middleware on funktio, jota kutsutaan actionin lähetyksen ja reducerissa tapahtuvan tilanmuutoksen välillä. (Redux-Saga Interview 2016.) Redux-Saga lisää Reduxin yhdensuuntaiseen datan kulkuun yhden vaiheen, jota havainnollistetaan kuviossa 6.



Kuvio 6. Reduxin yhdensuuntainen datan kulku Redux-Sagan kanssa

Saga on taustalla ajettava koodinpätkä, joka tarkkailee lähetettyjä actioneja, ja pystyy lähettämään uusia actioneja. Saga mahdollistaa asynkroniset toimenpiteet actionin lähetyksen yhteydessä. Sagat on toteutettu uusilla ES6:n generaattorifunktioilla. Generaattorifunktion suoritus voidaan pysäyttää, esimerkiksi kun odotetaan vastausta API:lta, ja suoritusta voidaan jatkaa vastauksen saavuttua. Generaattorifunktio voi myös palauttaa useita arvoja. (Mt.)

Kuviossa 7 on esimerkki yksinkertaisesta generaattorifunktiosta. Kun generaattorifunktiota kutsutaan, se palauttaa iteraattoriobjektin. Jokaisella iteraattorin *next()*-metodin kutsulla generaattorin ohjelmakoodia suoritetaan seuraavaan *yield*-lausekkeeseen saakka ja pysähdytään. Tämä helpottaa asynkronisen koodin kirjoittamista ja ymmärtämistä. (Herrera 2017.)

```
function* myGenerator() {
  let first = yield 'first yield value';
  let second = yield 'second yield value';
  return 'third returned value';
}

const it = myGenerator();
console.log(it.next()); // {value: 'first yield value', done: false}
console.log(it.next()); // {value: 'second yield value', done: false}
console.log(it.next()); // {value: 'third returned value', done: true}
console.log(it.next()); // {value: undefined, done: true}
```

Kuvio 7. Esimerkki generaattorifunktio

Redux-Sagassa määritellään saga, jonka tehtävänä on tarkkailla lähetettyjä actioneja. Varsinainen logiikka implementoidaan toiseen sagaan eikä tarkkailijaan. Tarkkailijasagassa voidaan käyttää erilaisia Redux-Sagan sisältämiä apufunktioita, kuten *takeEvery()*, jolla kutsutaan toista sagaa operaation suorittamiseen. Tästä nähdään esimerkki kuviossa 8.

```
// Watcher saga for spawning new tasks
function* watchFetchData() {
  yield takeEvery('FETCH_DATA', fetchDataAsync)
}

// Worker saga that performs the task
function* fetchDataAsync() {
  // ...
}
```

Kuvio 8. Esimerkki sagat

Jos tarkkailijasagaan saapuu useita pyyntöjä, *takeEvery()* aloittaa useita instansseja sagasta, joka suorittaa operaation. Toisin sanoen se huolehtii samanaikaisuudesta (engl. concurrency). Toinen esimerkki tällaisesta apufunktiosta on *takeLatest()*, joka suorittaa vain uusimman pyynnön ja peruuttaa aikaisemman operaation, mikäli se on vielä käynnissä. (Mt.) Varsinaisen logiikan sisältävä saga *fetchDataAsync()* voitaisiin toteuttaa esimerkiksi kuvion 9 mukaan.

```
function* fetchDataAsync() {
  try {
    const data = yield call(() => fetch(/* ... */))
    yield put(fetchDataSuccess(data))
  } catch (error) {
    yield put(fetchDataError())
  }
}
```

Kuvio 9. Esimerkki sagasta, joka hakee dataa asynkyronisesti

Esimerkissä nähdään kaksi uutta metodia: *call()* ja *put()*. Nämä ovat myös Redux-Sagan apufunktioita, ja ne palauttavat JavaScript-objektin, jota kutsutaan *efektiksi*. Efekti on yksinkertaisesti objekti, joka kuvaa toteutettavaa operaatiota. Sen sijaan, että kutsuttaisiin suoraan asynkronista operaatiota, *call()*-metodi palauttaa objektin, joka kuvaa kyseistä operaatiota. Redux-Saga huolehtii sen suorittamisesta ja palauttaa tuloksen generaattorille. (Mt.)

Sama tapahtuu *put()*-metodin kanssa, jota käytetään actionin lähettämiseen. Sen sijaan, että action lähetettäisiin suoraan *dispatch()*-metodilla, *put()*-metodi palauttaa objektin, joka sisältää ohjeet middlewarelle actionin lähettämiseen. (Mt.)

Efektien avulla sagat ovat deklaratiivisia. Tällä saavutettava etu on se, että funktio, joka palauttaa yksinkertaisen objektin, on helpompi testata kuin funktio, joka tekee asynkronisen kutsun. Testin ajamiseksi ei tarvita oikeaa API-rajapintaa eikä sitä tarvitse feikata. Testissä täytyy vain iteroida generaattorifunktion läpi ja testata palautettujen objektien yhdenvertaisuus. (Mt.) Esimerkki sagan testaamisesta nähdään kuviossa 10.

```
const iterator = fetchDataAsync();

// expects a call instruction
assert.deepEqual(
  iterator.next().value,
  call(fetch(/* ... */)),
  "fetchDataAsync should yield an Effect call(fetch)"
)
```

Kuvio 10. Esimerkki sagan testaamisesta

Toinen etu efekteistä on se, että useita efektejä voidaan yhdistää monimutkaisia toimenpiteitä varten. Redux-Saga tarjoaa useita apufunktioita efektien luomiseen esimerkiksi throttlausta tai tehtävien rinnakkaisajoa varten.

3.6 Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) on alustariippumaton ratkaisu viestien lähetykseen omalta palvelimelta Firebase-pilvipalvelun kautta iOS- ja Android-laitteille sekä web-sovelluksiin. Viestejä on mahdollista lähettää myös toisinpäin käyttäjän laitteelta palvelimelle. Firebase Cloud Messaging on ilmainen ja palvelun tarjoaa Google. (Firebase Cloud Messaging 2018.)

Viestejä voidaan lähettää käyttäen Admin SDK:ta tai HTTP ja XMPP API-rajapintoja. Näistä ainoastaan XMPP:tä käyttämällä voidaan lähettää viestejä käyttäjän laitteelta palvelimelle. (About FCM Server 2018.) Lähetettäviä viestejä on kahden tyyppisiä: notifikaatio-viestejä ja data-viestejä. Notifikaatio-viestit voivat sisältää vain ennalta määrättyjä avaimia sekä valinnaisen data-kentän, joka voi sisältää omia avain-arvo-pareja. FCM esittää notifikaatio-viestit käyttäjälle automaattisesti asiakassovelluksen puolesta. Data-viesteissä on vain omia avain-arvo-pareja, ja niiden käsittely on asiakassovelluksen vastuulla. Molempien viestien maksimi koko on 4KB. (About FCM Messages 2018.)

Virallista FCM SDK:ta ei ole saatavilla React Nativelle, mutta *react-native-fcm*-kirjasto on tehty iOS ja Android SDK:n päälle.

4 Sovelluksen toteutus

4.1 Kehitysympäristö

4.1.1 Lähtökohdat

Helpoin tapa aloittaa kehitys React Nativella on *Create React Native App* -niminen komentorivityökalu. Työkalun avulla kehitys voidaan aloittaa heti ilman konfigurointia tai kehitysympäristön pystytystä. Työkalu tukee vain JavaScript-sovelluksia, eli sen kanssa ei voi käyttää natiivia Javalla tai Objective-C:llä kirjoitettua koodia. Projekti voidaan kuitenkin muuttaa koska tahansa täydeksi React Native -projektiksi, mikäli natiiville koodille tulee tarve. Esimerkiksi useat React Native -kirjastot käyttävät natiivia koodia. Tässä työssä lähdettiin liikkeelle kyseisellä työkalulla ja projekti muutettiin myöhemmin täydeksi React Native -projektiksi, eikä toimenpiteessä tullut vastaan mitään ongelmia. Työkalun avulla on myös helppo testata sovellusta oikealla laitteella. Laitteeseen tulee vain asentaa *Expo*-niminen sovellus ja skannata komentoriville ilmestyvä QR-koodi. Koodimuutokset päivittyvät automaattisesti laitteella ajettavaan sovellukseen ”hot reload” -ominaisuuden ansiosta.

Seuraavaksi käydään läpi vaiheet, jotka tulee tehdä, kun aloitetaan täysi React Native -projekti. Kyseiset vaiheet tulee tehdä myös silloin, kun muutetaan Create React Native App -työkalulla tehty projekti täydeksi projektiksi. Sovellusten kehitykseen iOSille tarvitaan macOS. Windows ja Linux -käyttäjät voivat kuitenkin kehittää Android-sovelluksia. Tämä työ tehtiin macOS:llä, joten ohjeet keskittyvät sille. Kehitystä varten seuraavat ohjelmistot tulee olla asennettuna:

- Node.js
- Watchman
- React Native CLI
- iOS-kehitysympäristö (Xcode)
- Android-kehitysympäristö (JDK, Android SDK, Android Studio)

Node.js tarvitaan, koska React Native käyttää sille kirjoitettuja työkaluja. Noden paketinhallintajärjestelmää eli NPM:ää käytetään myös kirjastojen asentamiseen React Native -projektiin.

Watchman-työkalua käytetään koodimuutosten tarkkailuun. Kun Watchman huomaa muutoksen koodissa, JavaScript-paketti rakennetaan automaattisesti uudelleen. Tällä päästään eroon yhdestä natiivikoodauksen ikävästä ja hitaasta osasta.

Uusi sovellus voidaan luoda React Nativen komentorivityökaluilla, jotka asennetaan NPM-komennolla *npm install –g react-native-cli*. Projekti tarvittavine pohjineen React Nativelle, iOSille ja Androidille generoidaan komennolla *react-native init Projekti*. Generoitu kansiorakenne on kuvion 11 mukainen.

Kuvio 11. React Nativen generoima kansiorakenne

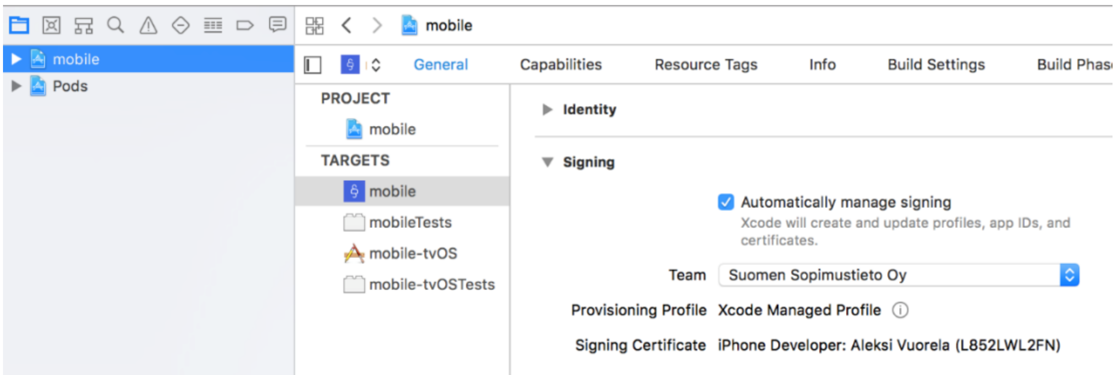
Kansiot *ios* ja *android* sisältävät tarvittavat pohjat kyseisille alustoille. Sovelluksen tulokohtana (engl. entry point) iOSille toimii *index.ios.js* ja Androidille *index.android.js*. Sovelluksen suoritus alkaa näistä tiedostoista. NPM-paketit löytyvät tuttuun tapaan *node_modules*-kansiosta.

4.1.2 iOS

Xcode voidaan asentaa helposti Macin App Storesta. Xcoden asennuksen mukana tulee iOS-simulaattori sekä tarvittavat työkalut iOS-sovelluksen rakentamiseen. Sovellus voidaan ajaa iOS-simulaattorilla komennolla *react-native run ios*. Vaihtoehtoisesti sovellus voidaan avata Xcodessa tiedostosta *ios/Projekti.xcodeproj* ja ajaa simulaattori sitä kautta. Xcodea tarvitaan myös, kun halutaan testata sovellusta oikealla laitteella. Tätä varten tarvitaan aluksi Applen kehittäjätunnukset.

Testaaminen oikealla laittella onnistuu ilmaisilla kehittäjätunnuksilla, mutta mikäli sovelluksen haluaa Applen sovelluskauppaan, tulee kehittäjätunnuksista maksaa 99 dollaria vuodessa.

Kehittäjätunnuksilla päästään konfiguroimaan koodin allekirjoittaminen. Koodin allekirjoitus vakuuttaa käyttäjille, että se on peräisin tunnetusta lähteestä ja sovellusta ei ole muokattu viimeisimmän allekirjoituksen jälkeen. Ennen kuin sovellus voidaan asentaa laittelle tai laittaa Applen sovelluskauppaan, sen tulee olla allekirjoitettu Applen myöntämällä sertifikaatilla. (Code Signing n.d.) Koodin allekirjoittaminen voidaan konfiguroida Xcodessa kuvion 12 mukaan.



Kuvio 12. Koodin allekirjoittamisen konfigurointi

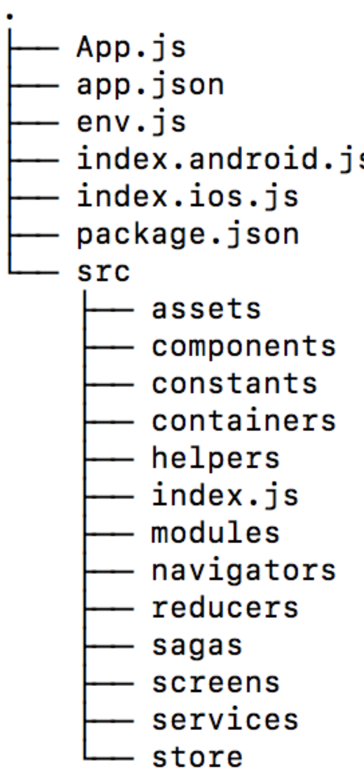
Avataan siis projekti ja valitaan *target*, joka on samanniminen kuin projekti. Tämän jälkeen *General*-välilehdeltä löytyy kohta *Signing*, jossa valitaan kehittäjätunnus *Team*-valikosta. Sama tulee tehdä myös *Targets*-listan kohdalle *Tests*. Kun sovellusta ajetaan ensimmäistä kertaa jollakin laitteella, Xcode pyytää kirjautumaan kehittäjätunnuksella ja rekisteröimään laitteen kehitystä varten.

4.1.3 Android

Android-kehitystä varten tarvitaan JDK, Android SDK ja Android Studio. Näiden asennusvaiheet ovat pidemmät, minkä vuoksi niitä ei käydä läpi tässä. Tarkat asennusohjeet löytyvät React Nativen *Getting Started* -oppaasta. Sovellus voidaan ajaa Androidilla komennolla *react-native run-android* tai avaamalla sovellus Android Studiossa, jossa se voidaan kääntää ja ajaa.

4.2 Projektin kansiorakenne

Projektissa käytetty kansiorakenne on kuvion 13 mukainen.



Kuvio 13. Projektin kansiorakenne

Sovelluksen tulokohtana iOSille toimii *index.ios.js* ja Androidille *index.android.js*. Näissä tiedostoissa rekisteröidään sovelluksen juurikomponentti (engl. root component), joka on *App.js*-tiedosto. App.js-tiedostossa tehdään sovelluksen alustavia toimenpiteitä, kuten API:n konfigurointi. Kun nämä toimenpiteet on tehty, renderöidään varsinainen sovellus *src/index.js*-tiedostosta. Index.js-tiedosto sisältää sovelluksen juurinavigaation, joka huolehtii tietyn näkymän renderöimisestä, kun kyseisen näkymän reitti on aktiivinen.

Sovelluksen ympäristömuuttujat löytyvät *env.js*-tiedostosta. Esimerkiksi API:n URL-osoite on määritelty tässä tiedostossa, sillä osoite on erilainen tuotanto- ja kehitysversiossa.

src sisältää seuraavat kansiot:

- **assets:** sovelluksen käyttämät kuvatiedostot
- **components:** Reactin esitykselliset komponentit
- **constants:** vakiomuuttujat kuten värit
- **containers:** Reactin säiliökomponentit
- **helpers:** useassa paikassa käytettävät apufunktiot
- **modules:** sisältää sovelluksen business-logiikkan, joka on jaettu useisiin kokonaisuuksiin omiksi moduuleikseen
- **navigators:** sovelluksen eri reitit
- **reducers:** sisältää juurireducerin, joka yhdistää kaikki reducerit
- **sagas:** sisältää juurisagan, joka yhdistää kaikki sagat
- **screens:** sovelluksen eri näkymät
- **services:** sisältää rajapinnat sovelluksen käyttämille palveluille, kuten hakukoneelle
- **store:** Redux-storen konfiguraatiotiedostot

4.3 Komponentit

Kaikki React-sovellukset koostuvat komponenteista, jotka kuvaavat renderöitävää käyttöliittymää. Komponenttien avulla koodi on uudelleenkäytettävää ja skaalautuvaa. React Native -komponentit ovat pitkälti samanlaisia kuin tavalliset React-komponentit, mutta niissä on eroavaisuuksia renderöinnin ja tyylien osalta. Reactissa renderöidään tavallisia HTML-elementtejä, mutta React Nativessa niiden sijaan käytetään alustalle ominaisia React-komponentteja. Yksinkertaisin tällainen komponentti on *View*, joka vastaa perus *div*-elementtiä. *View* renderöidään iOSilla käyttäen *UIView:tä* ja Androidilla *View:tä*. Taulukossa 1 on esitetty perus React-elementtien React Native -vastineita.

Taulukko 1. React-elementtien React Native -vastineita

React	**React Native**
<div>	<View>
<span>	<Text>
,	<FlatList>, lapsi elementit
<img>	<Image>

Jotkin komponentit toimivat vain yhdellä alustalla, esimerkiksi *DatePickerIOS* renderöi iOSin vakiopäivämäärävalitsimen. Koska nämä komponentit ovat erilaisia alustojen välillä, tulee React-komponenttien rakentamisesta järkevästi entistä tärkeämpää. Reactissa käytetään tavallisesti kahdentyyppisiä komponentteja: esityksellisiä komponentteja ja säiliökomponentteja. Esitykselliset komponentit eivät sisällä logiikka ja keskittyvät vain esittämään jonkin käyttöliittymän osan. Säiliökomponentit taas sisältävät logiikkaa, mutta ne eivät ota kantaa ulkoasuun, vaan koostavat muita komponentteja toimiviksi kokonaisuuksiksi. Jos koodia halutaan uudelleenkäyttää, tulee esityksellisten komponenttien ja säiliökomponenttien erottamisesta kriittistä.

Projektissa sopimuksen PDF-tiedoston esittäminen on hyvä esimerkki tästä. Androidissa PDF-tiedostoa ei voi esittää WebView-säiliössä kuten iOSissa, joten PDF-tiedoston esittävää komponenttia ei voi uudelleenkäyttää alustojen välillä. Komponentti, joka kapsuloi PDF-tiedoston hakemiseen liittyvän logiikan, taas voidaan uudelleenkäyttää alustojen välillä. Kyseinen säiliökomponentti on esitetty kuviossa 14.

```
class Container extends Component {
  componentDidMount() {
    this.props.load();
  }

  componentWillUnmount() {
    this.props.unload();
  }

  render() {
    return <ContractPreview {...this.props} />;
  }
}

function mapStateToProps(state, ownProps) {
  return {
    selectedContract: getSelectedContract(state),
    selectedContractPdf: getSelectedContractPdf(state)
  };
}

function mapDispatchToProps(dispatch, stateProps, ownProps) {
  const {selectedContract} = stateProps;

  return {
    load() {
      if (selectedContract) {
        dispatch(fetchContractPdf(selectedContract.id));
      }
    },
    unload() {
      dispatch(clearSelectedContractPdf());
    }
  };
}

export default connect(mapStateToProps, mapDispatchToProps)(Container);
```

Kuvio 14. PDF-tiedoston hakemisen logiikan kapsuloiva säiliökomponentti

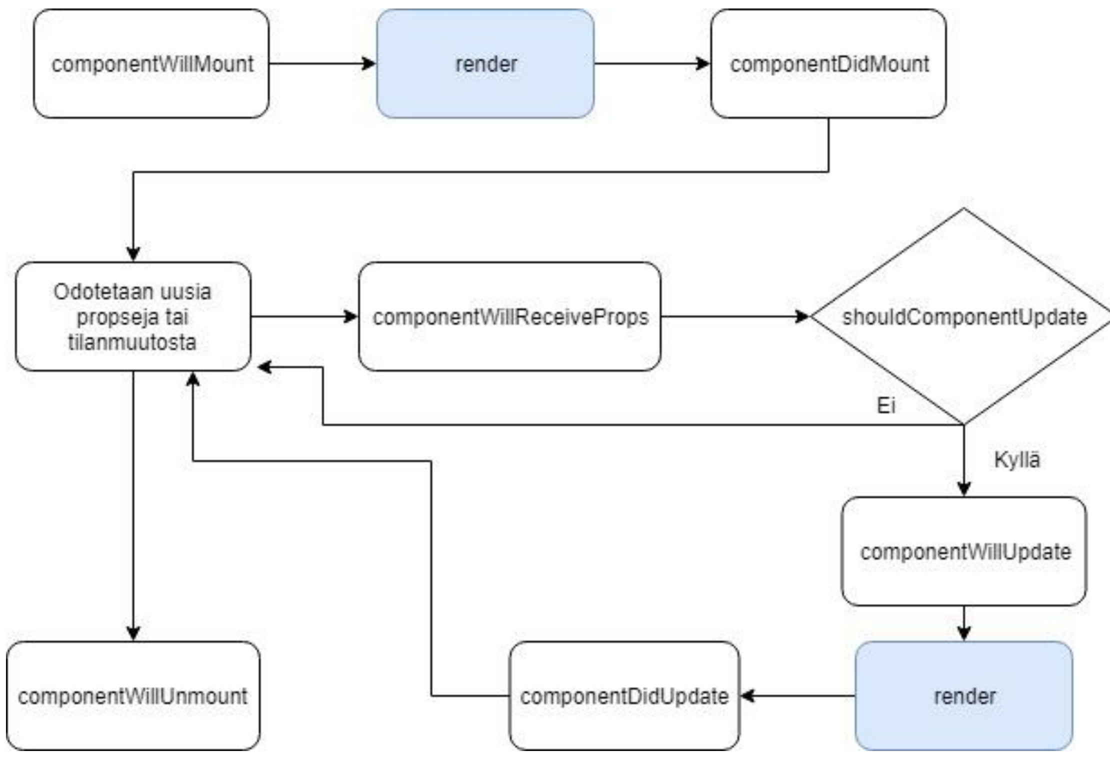
Ainoastaan visuaalinen komponentti täytyy vaihtaa alustan perusteella, joka on esitetty kuviossa 15. Androidissa PDF-tiedoston esittämiseen käytettiin *react-native-pdf*-kirjaston tarjoamaa komponenttia.

```
export default class ContractPreview extends Component {
  render() {
    const {selectedContractPdf} = this.props;

    return (
      <View style={styles.container}>
        {Platform.OS === 'android' &&
          <Pdf
            source={{uri: 'data:application/pdf;base64,' + selectedContractPdf}}
            style={styles.pdf} />
        }
        {Platform.OS === 'ios' &&
          <WebView
            scalesPageToFit
            source={{uri: 'data:application/pdf;base64,' + selectedContractPdf}}
            style={styles.pdf} />
        }
      </View>
    );
  }
}
```

Kuvio 15. PDF-tiedoston renderöivä esityksellinen komponentti

React-komponenteista saadaan dynaamisia *propseilla*, jotka ovat komponenteille annettavia parametreja. Esimerkiksi kuviossa 15 valitun sopimuksen PDF-tiedosto tulee ContractPreview-komponentille propsina. Komponenteilla voi olla oma sisäinen tila eli *state*, mutta tässä tapauksessa tilaa valitusta sopimuksesta säilytetään Redux-storessa. Aikaisemmassa kuviossa 14 valittu sopimus haetaan storesta Container-komponentissa ja annetaan propsina sen esittävälle ContractPreview-komponentille. Reduxia ja tilanhallintaa käsitellään tarkemmin luvussa 4.5.

Kuviossa 14 nähdään myös metodit *componentDidMount()* ja *componentWillUnmount()*. Nämä ovat React-komponentin metodeja, jotka suoritetaan tietyssä vaiheessa komponentin elinkaarta. Komponentin elinkaaren aikana suoritettavat metodit on esitettu kuviossa 16.



Kuvio 16. Komponentin elinkaaren aikana suoritettavat metodit

Ennen komponentin mounttausta, React kutsuu *componentWillMount*-metodia. Ensimmäisen renderöinnin jälkeen kutsutaan *componentDidMount*-metodia, joka on hyvä paikka tehdä API-kutsuja. Aikaisemmassa kuviossa 14 tässä kohtaa haetaan API:lta valitun sopimuksen PDF-tiedosto. Tämän jälkeen React jää odottamaan uusia propseja tai tilanmuutosta. *ComponentWillReceiveProps*-metodia kutsutaan, kun komponentille saapuu uusia proseja, ja niitä voidaan vertailla vanhojen kanssa.

ShouldComponentUpdate-metodia kutsutaan, kun propsit tai tila muuttuu. Metodi palauttaa totuusarvon (engl. boolean), jonka perusteella komponentti renderöidään uudelleen tai ei. *ComponentWillUpdate* ja *componentDidUpdate* metodeja kutsutaan ennen komponentin päivitystä ja sen jälkeen. Kun komponentti poistetaan DOMista, kutsutaan *componentWillUnmount*-metodia. Tämä on hyvä paikka siivota asioita, joita komponentti loi elinkaarensa aikana. Kuviossa 14 tässä kohtaa siivotaan pois storesta API:lta haettu sopimuksen PDF-tiedosto.

Yksinkertaiset komponentit, joilla ei ole omaa tilaa, kannattaa toteuttaa ns. funktionaalisina komponentteina. Nämä ovat funktioita, jotka ottavat vastaan propseja ja palauttavat renderöitäviä JSX-elementtejä. Funktionaalisilla

komponenteilla ei ole omaa tilaa eikä elinkaaren aikana suoritettavia metodeja. Niiden etuja ovat yksinkertaisuus, testaamisen helppous ja suorituskyky. Kuviossa 17 on esitetty komponentti tilin valitsemiseen, joka on toteutettu funktionaalisena komponenttina.

```
const AccountItemClient = ({account}) => {
  const active = isAccountActive(account);
  return (
    <ListItem onPress={_ => selectAccount(account)}>
      <MaterialInitials
        style={{alignSelf: 'center'}}
        backgroundColor={Colors.brandPrimary}
        color={'#fff'}
        size={40}
        text={account.attributes.name}
        single={false} />
      <Body>
        <Text>{account.attributes.name}</Text>
      </Body>
      <Right>
        {active
          ? <Icon ios="md-checkmark" android="md-checkmark" />
          : <Icon ios="ios-arrow-forward" android="md-arrow-forward" />}
      </Right>
    </ListItem>
  );
};
```

Kuvio 17. Funktionaalinen komponentti tilin valitsemiseen

4.4 JSX ja tyylit

React Nativessa näkymät tehdään JSX:llä kuten Reactissa. JSX yhdistää XML-tyylisen merkintätavan sekä sitä kontrolloivan JavaScriptin yhteen tiedostoon. JSX ei saanut aluksi kovin hyvää vastaanottoa, sillä web-kehittäjät olivat tottuneet erottamaan tiedostot teknologian perusteella: CSS-, HTML- ja JavaScript-tiedostot pidettiin erillään. Idea yhdistää merkintätapa, kontrolloiva logiikka ja jopa tyylit yhteen kieleen tuntui hämmentävältä. JSX keskittyy vastuiden jakamiseen (engl. separation of concerns) teknologioiden erottelun sijaan. React Nativessa ei tarvitse huolehtia selaimesta, jonka takia on entistä järkevämpää yhdistää tyylit, merkintätapa ja käyttäytyminen yhteen tiedostoon jokaiselle komponentille. Esimerkki JSX:n syntaksista nähdään aikaisemmassa kuviossa 17.

Reactissa komponenttien tyylit määritellään CSS:llä, mutta React Nativessa tyylit määritellään hieman eri tavalla. Osana React Nativen siltaa, joka toimii Reactin ja isäntäalustan välillä, on toteutettu raskaasti karsittu CSS:n osajoukko. Tämä suppea CSS:n toteutus perustuu pitkälti *flexboxiin* asemmoinnin suhteen ja keskittyy yksinkertaisuuteen. Flexbox on kohtuullisen uusi ja hyvin joustava tapa asemmoida elementtejä CSS:llä. Webissä CSS-tuki vaihtelee eri selainten välillä, mutta React Native pystyy noudattamaan yhtenäistä tukea tyylisäännöille.

React Nativessa tyylit kirjoitetaan JavaScript-objekteina. Yksi CSS:n suurimmista ongelmista on se, että kaikki tyylisäännöt ja luokkien nimet ovat globaaleja. Tämä voi johtaa siihen, että yhden komponentin tyylit hajottavat toisen, mikäli ei ole tarkkana. React Nativessa tätä ongelmaa ei ole, sillä tyylien näkyvyys on rajattu komponenttiin. Useiden eri komponenttien käyttämät tyylit voidaan määritellä yhteisiin tyyli-tiedostoihin, mistä ne voidaan tuoda (engl. import) komponentteihin. Esimerkki tyylien määrittelystä on esitetty kuviossa 18.

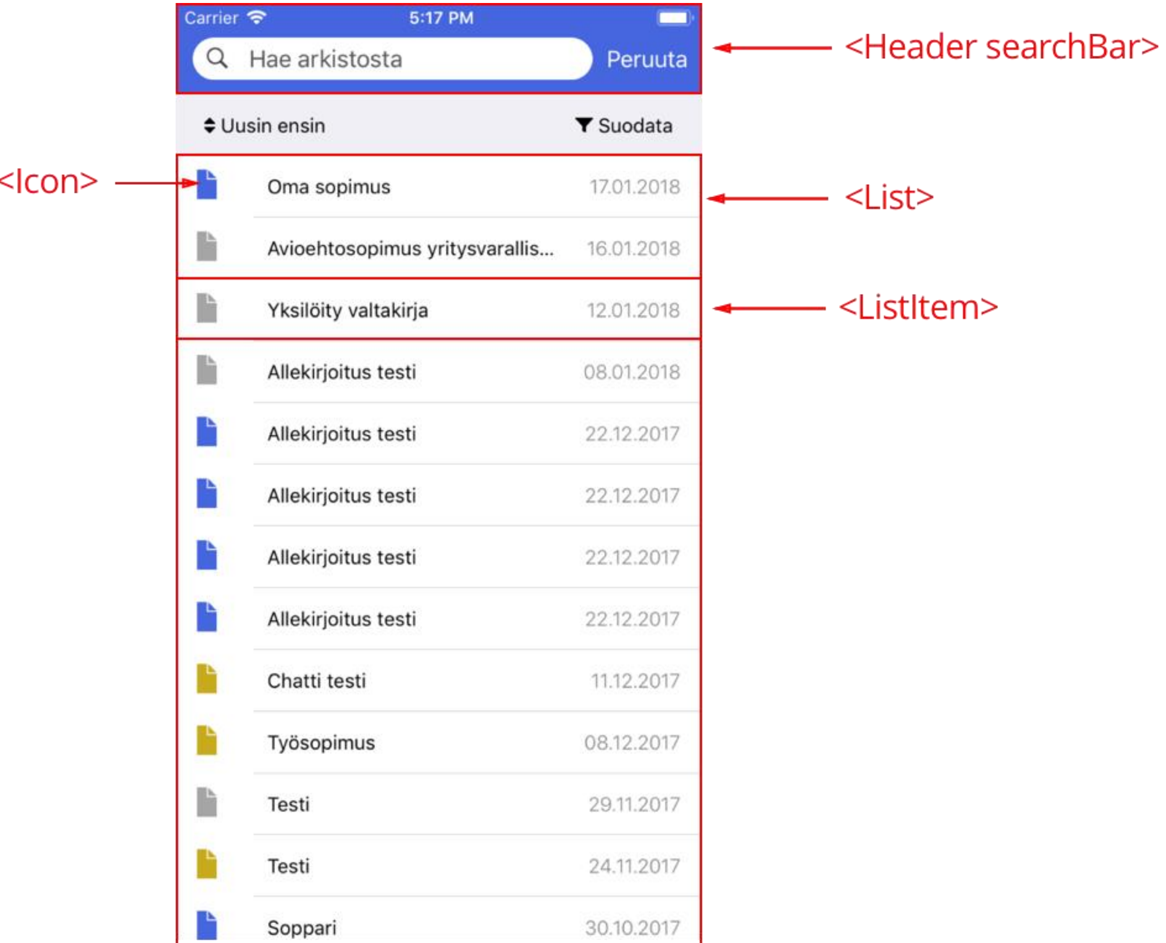
```
const styles = {
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center'
  },
  webViewContainer: {
    width: 400,
    height: 150,
    overflow: 'hidden'
  },
  footer: {
    position: 'absolute',
    bottom: 0,
    flexDirection: 'row'
  }
};
```

Kuvio 18. Esimerkki tyylien määrittelystä JavaScript-objektina

4.5 Komponenttikirjastot

Kaikissa käyttöliittymissä esiintyy samanlaisia, yleisiä elementtejä, kuten painikkeet ja lomakkeet. Näiden tekeminen tyhjästä vie aikaa, jonka takia usein käytetään apuna jotakin kirjastoa. Webissä suosittu kirjasto tähän tarkoitukseen on Bootstrap. React Nativessa komponenttien tulee lisäksi toimia ja näyttää hyvältä kahdella eri alustalla.

Suosittuja React Native komponenttikirjastoja ovat mm. React Native Elements sekä NativeBase. Näistä ensimmäinen on kevyt ja yksinkertainen, mutta tähän projektiin valikoitui NativeBase. Kyseinen kirjasto on hyvin monipuolinen ja sen komponentit näyttävät pitkälti natiiveilta iOS- ja Android-komponenteilta. Kaikkia komponentteja voidaan myös kustomoida muokkaamalla niiden teemoja ja kirjaston dokumentaatio on erinomainen. Kuviossa 19 on esitetty näkymä sovelluksesta, jossa listataan käyttäjän sopimukset, sekä siinä käytetyt NativeBase-komponentit.



Kuvio 19. Sopimusten listauksessa käytetyt NativeBase-komponentit

4.6 Tilanhallinta

4.6.1 Yleistä

Sovelluksen tilanhallinta toteutettiin Reduxilla sekä komponentin sisäisellä tilalla, kun tilatietoa ei tarvittu komponentin itsensä ulkopuolella. Projektissa käytettiin seuraavia Reduxiin liittyviä kirjastoja:

- **Redux:** peruskirjasto
- **React-Redux:** apukirjasto Reduxin käyttöön Reactin kanssa
- **Redux-Saga**: apukirjasto Reduxille sivuvaikutusten käsittelyyn
- **Redux-persist**: storen tallentaminen laitteen muistiin
- **Redux-persist-transform-filter:** apukirjasto Redux-persistille, jolla voidaan rajata tallennettavia asioita
- **Reselect:** apukirjasto Reduxille tiedon hakemiseen storesta, jolla tulokset voidaan tallentaa välimuistiin turhan uudelleenlaskennan välttämiseksi
- **Redux Form:** lomakkeiden tilan tallennus storeen

4.6.2 Storen luonti

Reduxissa ensimmäisenä tulee luoda store, joka on esitetty kuviossa 20.

```
import {compose, applyMiddleware, createStore} from 'redux';
import {autoRehydrate} from 'redux-persist';
import createSagaMiddleware from 'redux-saga';
import rootReducer from '../reducers';

const devTools = window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__();

// middleware
const sagaMiddleware = createSagaMiddleware();
const middleware = applyMiddleware(
  sagaMiddleware
);

// compose all enhancers
const enhancer = devTools
  ? compose(middleware, devTools, autoRehydrate())
  : compose(middleware, autoRehydrate());

export default function configureStore() {
  const store = createStore(rootReducer, undefined, enhancer);
  store.runSaga = sagaMiddleware.run;
  return store;
}
```

Kuvio 20. Redux-storen luominen

Esitetty konfiguraatio on sovelluksen kehitysversiolle, jossa käytetään Redux DevTools -lisäosaa. Lisäosa auttaa Reduxin debuggaamisessa, ja se esitellään tarkemmin luvussa 4.12. Tuotantoversion konfiguraatio on samanlainen, mutta siinä ei käytetä Redux DevToolsia. Konfiguraatiossa määritellään storeen Redux-Saga middleware sekä Redux-persist, jonka avulla store populoidaan laitteen muistiin tallennetusta datasta automaattisesti. Näin esimerkiksi käyttäjä pysyy kirjautuneena sovelluksessa, vaikka se suljetaan. Storeen määritellään myös juurireducer, joka yhdistää kaikki sovelluksen reducerit. Juurireducer on esitetty kuviossa 21.

```
import {combineReducers} from 'redux';
import {reducer as form} from 'redux-form';
import accounts from '../modules/accounts/reducer';
import auth from '../modules/auth/reducer';
import contracts from '../modules/contracts/reducer';
import search from '../modules/search/reducer';
import metas from '../modules/metas/reducer';
import infoServices from '../modules/info-services/reducer';
import notifications from '../modules/notifications/reducer';
import signature from '../modules/signature/reducer';
import chat from '../modules/chat/reducer';
import campaigns from '../modules/campaigns/reducer';

export default combineReducers({
  form,
  accounts,
  auth,
  contracts,
  search,
  metas,
  infoServices,
  notifications,
  signature,
  chat,
  campaigns
});
```

Kuvio 21. Sovelluksen juurireducer

4.6.3 Storen tallennus laitteen muistiin

Sovelluksen juurikomponentissa *App.js*-tiedostossa määritellään, mitä osia storesta tallennetaan laitteen muistiin. Tämä määrittely tapahtuu *init()*-funktiossa, joka on esitettu kuviossa 22.

```
init() {
  // blacklist filters (don't save specified keys)
  const authFilter = createBlacklistFilter('auth', ['isLoggingIn', 'isVerifying']);
  const accountsFilter = createBlacklistFilter('accounts', ['activeCompany']);

  // filters (save only specified keys)
  const notificationsFilter = createFilter('notifications', ['device']);

  persistStore(store, {
      storage: AsyncStorage,
      whitelist: ['auth', 'accounts', 'notifications'],
      transforms: [authFilter, accountsFilter, notificationsFilter]
    }, () => {
  });
}
```

Kuvio 22. Määrittely laitteen muistiin tallennettavista storen osista

Init()-funktiota kutsutaan komponentin mountatessa. Funktiossa määritellään aluksi filtterit *Redux-persist-transform-filter*-kirjaston avulla. Filttereiden avulla voidaan tallentaa vain tiettyjä avaimia reducerista. Filttereitä on kahden tyyppisiä: *blacklist-* ja *whitelist*-filttereitä. Blacklist-filtterissä reducerista tallennetaan kaikki muut avaimet, paitsi erikseen määritellyt avaimet. Whitelist-filtterissä taas tallentaan vain määritellyt avaimet.

Storen tallennus laitteen muistiin tapahtuu Redux-persistin *persistStore()*-funktiossa. Data määritellään tallentumaan *AsyncStorageen*, joka on React Nativen ominaisuus avain-arvo-parien tallentamiseen. AsyncStorage käyttää iOSissa natiivia koodia, joka tallentaa arvot serialisoituun sanakirjaan (engl. dictionary) ja suuremmat arvot erillisiin tiedostoihin. Androidissa AsyncStorage käyttää RocksDB:tä tai SQLiteä sen mukaan, kumpi on saatavilla. AsyncStorageen tallennettua dataa ei salata, jonka takia sinne ei tule tallentaa arkaluontoista dataa, kuten salasanoja. (AsyncStorage n.d.) Seuraavaksi funktiossa määritellään tallennettavat reducerit sovelluksen juurireducerista. Lopuksi annetaan filtterit, joilla tallennettavista reducereista valitaan tallennettavat avaimet.

4.6.4 Komponentin kytkeminen storeen React-Reduxilla

Aikaisemmassa kuviossa 14 esiteltiin säiliökomponenttia, joka haki sopimuksen PDF-tiedoston API:lta ja antoi haetun PDF-tiedoston propsina PDF:än esittävälle komponentille. Kyseinen säiliökomponentti on tietoinen Redux-storesta, sillä se on kytketty storeen React-Redux-apukirjaston *connect()*-funktiolla. Kyseistä funktiota varten komponenttiin tulee määritellä erityinen *mapStateToProps*-funktio. Funktiossa määritellään, kuinka nykyinen Redux-storen tila muutetaan propseiksi, jotka annetaan esittävälle komponentille. Tässä tapauksessa mapStateToProps-funktiossa storesta haettiin tieto siitä, mikä sopimus on tällä hetkellä valittuna, sekä mikä on kyseisen sopimuksen PDF-tiedosto.

Tiedonhaun lisäksi säiliökomponentit voivat lähettää actioneja. Tätä varten komponenttiin määritellään toinen erityinen funktio: *mapDispatchToProps*. Kyseiseen funktioon voidaan määritellä funktioita, jotka lähettävät actioneja *dispatch()*-metodilla. Määritellyt funktiot annetaan propseina esittävälle komponentille, josta niitä voidaan kutsua. Funktioita voidaan myös kutsua säiliökomponentista itsestään, kuten kuviossa 14 kutsutaan *load()*-funktiota komponentin mountattua, joka lähettää actionin PDF-tiedoston hakemiseen API:lta.

4.6.5 Tiedonhaku storesta selector-funktioilla

Tieto haetaan storesta *selectoreiksi* kutsutuilla funktioilla. Kuviossa 23 on esitetty kaksi selector-funktiota käyttäjän kaikkien sopimusten hakemiseen storesta.

```
export const getContracts = get('contracts.byId');
export const getContractsList = createSelector(getContracts, Object.values);
```

Kuvio 23. Selector-funktiot käyttäjän kaikkien sopimusten hakemiseen storesta

Käyttäjän sopimukset ovat tallennettuna storeen normalisoidussa muodossa, jossa sopimusten ID:t toimivat avaimina. Redux-storen tilan tulisi olla aina normalisoitu, aivan kuten relaatiotietokannan, jotta sama tieto ei toistu ja rakenne pysyy selkeänä. Selector-funktioiden suurin hyöty on se, että storeen voidaan tallentaa vain välttämätön tilatieto.

Sovelluksen käyttöliittymässä tarvitaan käyttäjän sopimukset listan muodossa, eikä normalisoidussa muodossa. Sopimuksia ei kuitenkaan ole järkeä tallentaa toiseen kertaan storeen listana. Selector-funktio *getContracts()* palauttaa sopimukset, kuten ne ovat tallennettuna storeen normalisoidussa muodossa. Toinen selector-funktio, *getContractsList()* taas käyttää hyväkseen ensimmäistä selector-funktiota ja tekee sille laskennan *Object.values()*-funktiolla, joka palauttaa sopimukset listan muodossa. Selector-funktio *getContractsList()* käyttää lisäksi Reselect-kirjaston *createSelector()*-funktiota, joka tallentaa tulokset välimuistiin. Näin vältytään turhalta uudelleenlaskennalta, jonka *Object.values()*-funktio tekisi aina tilan päivittyessä. Uudelleenlaskenta tehdään vain siinä tapauksessa, kun storeen tallennetut sopimukset päivittyvät. Kuviossa 23 esiintyvä *get()*-funktio on itse tehty apufunktio, joka palauttaa datan storesta määritellystä polusta. Selector-funktioiden etuja ovat siis myös niiden yhdistely sekä suorituskyky. Selector-funktioilla on kuitenkin vielä yksi etu: jos datan rakenne storessa muuttuu, ainoastaan selector-funktio täytyy muuttaa, eikä refaktoroida jokaista komponenttia hakemaan dataa uuden rakenteen mukaan.

4.6.6 Actionit

Actionit ovat yksinkertaisesti JavaScript-objekteja. Action-objekti sisältää aina tyypin, joka kertoo, mitä tulee tapahtua. Action-objekti voi sisältää myös dataa, jota kutsutaan actionin payloadiksi. Projektissa actionit määriteltiin jokaisen moduulin *actions.js*-tiedostoon. Esimerkiksi action, jolla haetaan sopimuksen PDF-tiedosto, näyttää kuvion 24 mukaiselta.

```
import {createNamespace, fsa} from '../../helpers/action';

const ns = createNamespace('contracts');

export const FETCH_CONTRACT_PDF = ns('FETCH_CONTRACT_PDF');
export const fetchContractPdf = id => fsa(FETCH_CONTRACT_PDF, id);
```

Kuvio 24. Action, jolla haetaan sopimuksen PDF-tiedosto

Kaikki sovelluksen action tyypit ovat määritelty nimiavaruuteen (engl. namespace), jona toimii moduulin nimi. Näin eri moduulien actionit eivät mene sekaisin, vaikka

niiden tyyppi olisi samanniminen. Actionit nimittäin ovat globaaleja, sillä sovelluksen kaikkien moduulien reducerit yhdistetään juurireducerissa *combineReducers()*-funktiolla.

Kuviossa 24 esiintyvän actionin tyyppi on siis *contracts/FETCH_CONTRACT_PDF*. Kyseinen action voidaan lähettää säiliökomponentista kutsumalla *dispatch(fetchContractPdf(id))*. Funktio *fetchContractPdf(id)* luo action-objektin ja se vastaanottaa parametrina actionin payloadin eli tässä tapauksessa sopimuksen ID:n. Tälläisiä funktioita, jotka luovat actioneja, kutsuaan Reduxissa termillä *action creator*. Action-objekti voitaisiin antaa myös suoraan *dispatch()*-funktiolle ilman action creatoria. Action creatorien käyttämisestä on kuitenkin useita hyötyjä:

1. Abstraktio: Sen sijaan, että kirjoitettaisiin action-objekti jokaiseen kyseistä actionia tarvitsevaan komponenttiin, logiikka actionin luontiin voidaan laittaa vain yhteen paikkaan. Action-objektin luontiin voi myös liittyä laajempaa logiikkaa sen sijaan, että se vain palautettaisiin suoraan.
2. Dokumentaatio: Funktion parametrit toimivat ohjeina siitä, mitä dataa action tarvitsee.
3. Kapsulointi ja yhtenäisyys: Komponentin ei tarvitse tietää yksityiskohtia siitä, kuinka action luodaan. Komponentti kutsuu aina vain funktiota, joka huolehtii yksityiskohdista. (Idiomatic Redux: Why use action creators? 2016.)

Kuviosta 24 nähdään myös, että *fetchContractPdf()*-funktio kutsuu sisällään *fsa()*-funktiota, jolle annetaan parametreina actionin tyyppi ja payload. Tämä *fsa()*-funktio on itse tehty apufunktio, joka on lyhenne sanoista Flux Standard Action. Kyseinen funktio pyöräyttää annetun actionin tyypin ja payloadin standardiin muotoon, joka on esitetty kuviossa 25.

```
{
  type: 'contracts/FETCH_CONTRACT_PDF',
  payload: {
    id: '123'
  }
}
```

Kuvio 25. Flux Standrad Action

4.6.7 Reducerit

Actionien tapaan myös reducerit määriteltiin jokaisen moduulin *reducer.js*-tiedostoon. Reducerit ovat funktioita, joissa napataan kiinni eri tyyppiset actionit sekä actionien sisältämä payload, joiden perusteella tehdään tilanmuutos. Reducer-funktio vastaanottaa actionin lisäksi sovelluksen nykyisen tilan ja se palauttaa uuden, muuttuneen tilan. Actionit kuvaavat siis mitä tapahtui, mutta varsinainen tilanmuutos tehdään reducerissa.

Käytännössä reducer on yksi suuri switch-lauseke, jossa käsitellään jokainen eri tyyppinen action ja palautetaan uusi tila. Kuviossa 26 on esitetty, kuinka reducerissa käsitellään *SET_SELECTED_CONTRACT_PDF*-tyyppinen action. Actionin payload sisältää sopimuksen PDF-tiedoston base64-muodossa. Se asetetaan storessa polkuun *contracts.selectedContractPdf*, jonka jälkeen palautetaan uusi tila.

```
case SET_SELECTED_CONTRACT_PDF: return set('selectedContractPdf', payload, state);
```

Kuvio 26. Sopimuksen PDF-tiedoston asettaminen storeen reducerissa

Kuviossa 26 esiintyvä *set()*-funktio on itse tehty apufunktio, joka vastaanottaa kolme parametria: polun objektissa (pistenotaatiolla määritelty merkkijono), polkuun asetettavan arvon sekä itse objektin. Funktio palauttaa uuden objektin, jossa annettu arvo on asetettu määriteltyyn polkuun.

Reducer-funktioiden tulee aina palauttaa uusi tilaobjekti eikä muuttaa vanhaa, sillä reducerit ovat puhtaita funktioita (engl. pure function). Puhdas funktio tarkoittaa funktiota, joka palauttaa samalla syötteellä (engl. input) aina saman tulosteen (engl. output), eikä se tuota sivuvaikutuksia. Toisin sanoen reducer-funktiossa ei saa koskaan muuttaa sille annettuja argumentteja, tehdä API-kutsuja tai kutsua epäpuhtaita funktioita kuten *Date.now()*. (Reducers 2018.)

Reducereissa voidaan myös määritellä oletustila (engl. initial state). Kun Redux alustetaan ensimmäistä kertaa, se lähettää “alustus-actionin” täyttääkseen storen. Tämä action kutsuu sovelluksen reducer-funktioita antamalla niille tilan, jonka arvo on *undefined,* mikäli oletustilaa ei ole määritelty. Näin ollen koko storen arvo olisi

undefined. (Initializing State 2018.) Oletustila voidaan määritellä antamalla se reducer-funktioille oletusparametrina, joka on esitetty kuviossa 27.

```
import initialState from './initial-state';

export default function contracts(state = initialState, {payload, type}) {
  switch (type) {⋯
  }

  return state;
}
```

Kuvio 27. Oletustilan antaminen reducer-funktiolle

Reducer-funktiolle annetaan siis oletusparametrina *state = initialState*, jonka reducer palauttaa kun sitä kutsutaan alustusvaiheessa ensimmäistä kertaa. Projektissa jokaisen moduulin reducerille määriteltiin oletustila omaan tiedostoon, esimerkiksi *contracts*-reducerin oletustila näyttää kuvion 28 mukaiselta.

```
export default {
  initialLoad: true,
  loaded: false,
  fetching: false,
  refreshing: false,
  searching: false,
  byId: {},
  filter: createFilter('lastUsed'),
  sortBy: SORT_BY_OPTIONS[0],
  paginator: {
    total: 1,
    count: 1,
    per_page: 1,
    current_page: 1,
    total_pages: 1
  },
  selectedContract: null,
  selectedContractPdf: '',
  selectedMetas: [],
  ui: {
    migrateOverlayOpen: false
  }
};
```

Kuvio 28. Contracts-reducerin oletustila

4.7 Sivuvaikutukset

Reducerit eivät voi tuottaa sivuvaikutuksia, kuten API-kutsuja, sillä ne ovat puhtaita funktioita. Reduxin middleware kuitenkin mahdollistaa actionien kiinni nappaamisen ja sitä kautta sivuvaikutusten lisäämisen niiden yhteyteen. Yksinkertaisin tapa sivuvaikutusten käsittelyyn on Redux Thunk -kirjasto. Sen avulla voidaan kirjoittaa action creatoreita, joissa actionin lähetystä voidaan viivästyttää, tai lähettää action vain, jos tietty ehto toteutuu. Toinen paljon käytetty kirjasto on Redux-Saga, jonka avulla voidaan kirjoittaa synkronisen näköistä koodia käyttäen generaattori-funktioita. Redux-Sagan koodi toimii ikään kuin taustalla ajettavana säikeenä. Vielä yksi suosittu vaihtoehto on Redux-Loop, jossa reducerit esittelevät sivuvaikutukset, ja ne ajetaan erikseen. (Actions 2018.)

Projektiin valittiin sivuvaikutusten käsittelyyn Redux-Saga. Suurin syy tähän oli se, että Redux-Saga on käytössä myös Sopimustiedon web-sovelluksessa, joten aiempaa koodia voitiin jälleen hyödyntää. Web-sovellusta tehdessä Redux-Saga todettiin erinomaiseksi ratkaisuksi, sillä generaattori-funktiot tekevät asynkronisesta koodista synkronisen näköistä, jota on helppo kirjoittaa, lukea ja testata. Redux-Sagalla voidaan myös tehdä hyvin monimutkaisia operaatioita sen tarjoamien apufunktioiden avulla. Ainoa huono puoli on sen korkea oppimiskynnys johtuen generaattori-funktioiden syntaksista ja useista opeteltavista konsepteista. Näiden teoriaa käsiteltiin luvussa 3.5.

Redux-Sagan käyttöönotto projektissa nähdään aikaisemmasta kuviosta 20, jossa luodaan Redux-store. Storea luodessa siihen kytketään kiinni Redux-Saga middleware ja käynnistetään se. Kuten sovelluksen kaikki reducerit yhdistettiin yhdeksi juurireduceriksi, myös sagat voidaan yhdistää yhdeksi juurisagaksi. Sovelluksen juurikomponentissa eli *App.js*-tiedostossa juurisaga käynnistetään komponentin mountatessa kuvion 29 mukaan.

```
import store from './store/configure-store';
import rootSaga from './sagas';

class App extends Component {
  componentWillMount() {
    store.runSaga(rootSaga);
  }
```

Kuvio 29. Juurisagan käynnistäminen

Juurisaga näyttää kuvion 30 mukaiselta.

```
import {fork, all} from 'redux-saga/effects';
import auth from '../modules/auth/saga';
import accounts from '../modules/accounts/saga';
import contracts from '../modules/contracts/saga';
import search from '../modules/search/saga';
import metas from '../modules/metas/saga';
import signature from '../modules/signature/saga';
import infoServices from '../modules/info-services/saga';
import notifications from '../modules/notifications/saga';
import chat from '../modules/chat/saga';
import lawAssignment from '../modules/law-assignment/saga';
import campaigns from '../modules/campaigns/saga';


export default function* root() {
  yield all ([
    fork(auth),
    fork(accounts),
    fork(contracts),
    fork(search),
    fork(metas),
    fork(signature),
    fork(infoServices),
    fork(notifications),
    fork(chat),
    fork(lawAssignment),
    fork(campaigns),
  ]);
}
```

Kuvio 30. Sovelluksen juurisaga

Juurisagassa käytetään Redux-sagan efektejä *all()* ja *fork()*, joiden avulla kaikkien moduulien sagat yhdistetään ja palautetaan vain yksi efekti.

Kuviossa 31 on esitetty sopimuksen PDF-tiedoston hakeminen API:lta sagassa.

```
function* handleFetchContractPdf({payload}) {
  try {
    const pdf = yield call(api.fetchContractPdf, payload);
    yield put(setSelectedContractPdf(pdf));
  } catch (e) {
    yield put(contractPdfFetchFailed());
  }
}

function* watchFetchContractPdf() {
  yield takeLatest(FETCH_CONTRACT_PDF, handleFetchContractPdf);
}

export default function* watch() {
  yield fork(watchFetchContracts);
  yield fork(watchFetchByPage);
  yield fork(watchSendContractToEmail);
  yield fork(watchUpdate);
  yield fork(watchRemove);
  yield fork(watchMigrate);
  yield fork(watchDuplicate);
  yield fork(watchFetchAndSetSelectedContract);
  yield fork(watchFetchContractPdf);
}
```

Kuvio 31. Sopimuksen PDF-tiedoston hakeminen API:lta sagassa

Kuten luvussa 3.5 kerrottiin, sagoja määritellään kaksi: tarkkailija ja käsittelijä. Tarkkailijasaga eli *watchFetchContractPdf()*-funktio nappaa kiinni actioneja, joiden tyyppi on *FETCH_CONTRACT_PDF*. Tarkkailijasagassa käytetään Redux-Sagan apufunktiota *takeLatest()*, joka nappaa kiinni viimeisimmän *FETCH_CONTRACT_PDF*-tyyppisen actionin ja kutsuu käsittelijäsagaa eli *handleFetchContractPdf()*-funktiota. Käsittelijäsagalle välitetään parametrina actionin payload eli tässä tapauksessa sopimuksen ID.

Käsittelijäsagassa haetaan sopimuksen PDF-tiedosto API:lta. Tähän käytetään Redux-Sagan apufunktiota *call()*. Se palauttaa sagalle asynkronista operaatiota kuvaavan objektin eli efektin, jonka suorituksesta middleware huolehtii. Kyseiselle funktiolle annetaan parametreina API-kutsun suorittava funktio (*api.fetchContractPdf*) sekä arvot, jotka halutaan antaa tälle funktiolle parametreina (payload eli sopimuksen ID). API-kutsun suorittava funktio on esitetty kuviossa 32.

```
fetchContractPdf(id) {
  return api.get(`/me/account/items/${id}/pdf`).send().then(res => res.data.data);
},
```

Kuvio 32. API-kutsun suorittava funktio

API-kutsuja suorittavat funktiot määriteltiin jokaiselle moduulille omaan *api.js*-tiedostoon. Ne käyttävät API-serviceä pyynnön tekemiseen, joka asettaa kaikille sovelluksen API-kutsuille tyypilliset asiat kohdalleen, kuten käyttäjän todennuksen pyynnön otsakkeeseen. API-service käyttää taustallla Axios-kirjastoa, joka pohjautuu JavaScriptin Promise-olioihin. Näin ollen *fetchContractPdf()*-funktio palauttaa Promisen.

Käsittelijäsagassa käytettiin *call()*-funktiota, jolle annettiin API-kutsun suorittava funktio. Mikäli API-kutsun suorittava funktio palauttaa Promisen, kuten tässä tapauksessa, *call()*-funktio pysäyttää sagan suorittamisen kunnes Promise on selvitetty. Mikäli Promisen tila on “täytetty” (engl. fulfilled) eli operaatio onnistui, jatketaan sagan suorittamista. Kuvion 31 tapauksessa lähetetään action (*setSelectedContractPdf*), joka asettaa API:lta haetun sopimuksen PDF-tiedoston storeen. Mikäli taas Promisen tila on “hylätty” (engl. rejected), sagan suoritus lopetetaan ja heitetään virhe. Virheidenkäsittely sagassa tapahtuu perus *try/catch*-lauseessa. Kuviossa 31 virhetilanteessa lähetetään virhettä kuvaava action (*contractPdfFetchFailed*), jonka perusteella näytetään ilmoitus käyttöliittymässä.

Kuvion 31 lopussa nähdään myös *watch()*-funktio. Tämä funktio yhdistää moduulin kaikki tarkkailijasagat ja yhdistettyä tarkkailijaa käytetään sovelluksen juurisagassa.

4.8 Navigointi

React Nativeen ei ole sisäänrakennettu globaalia historiapinoa kuten web-selaimeen. Suosituin kirjasto tähän tarkoitukseen on React Navigation, jonka avulla sovelluksessa voidaan siirtyä näkymien välillä ja hallita navigointi historiaa. Sen pinonavigaattori toimii kuten selaimen; sovellus lisää ja poistaa alkioita pinon huipulta käyttäjän vuorovaikutuksesta. Suurin ero selaimeen on siinä, että React Navigation tarjoaa eleet ja animaatiot, joita käyttäjä odottaa navigoidessaan Androidilla ja iOSilla. (Hello React Navigation n.d.)

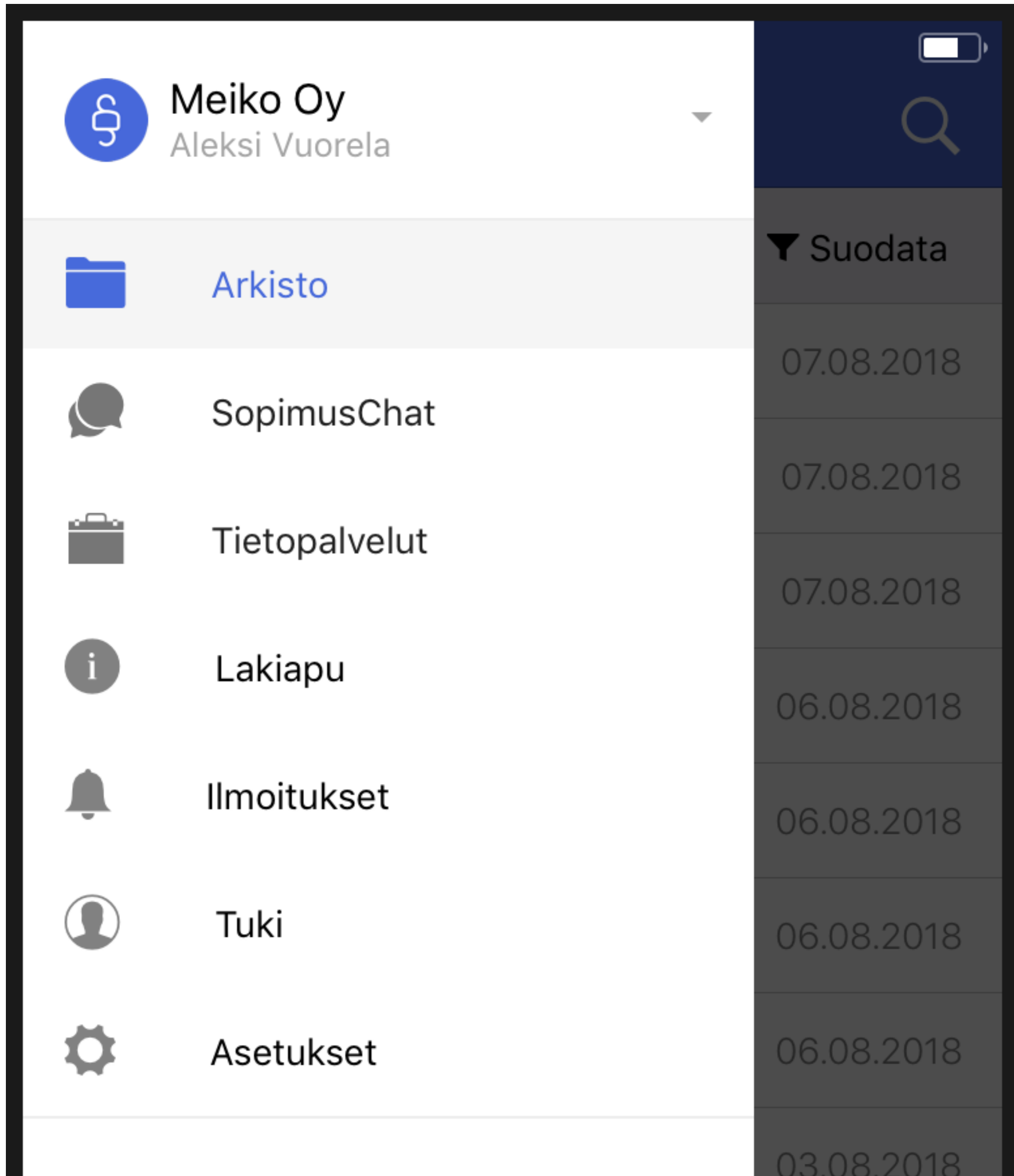
Kaikki sovelluksen navigaattorit määriteltiin *navigators/index.js*-tiedostoon. Sovelluksen juurinavigaattori on esitetty kuviossa 33.

```
// Root routes
export const createRootNavigator = (signedIn = false) => {
  return StackNavigator({
    SignedIn: {
      screen: SignedIn
    },
    SignedOut: {
      screen: SignedOut
    }
  },
  {
    headerMode: 'none',
    initialRouteName: signedIn ? 'SignedIn' : 'SignedOut'
  });
};
```

Kuvio 33. Sovelluksen juurinavigaattori

Sovelluksen juurinavigaattori luodaan funktiolla, jolle välitetään tieto siitä, onko käyttäjä kirjautunut sisään. Funktio palauttaa pinonavigaattorin ja asettaa sen oletus-reitin kirjautumisen perusteella. Navigaattoriin määritellään sen eri reittin nimet (*SignedIn* ja *SignedOut* tässä tapauksessa) sekä reittiä vastaava näkymä (*screen*). Näkymä voi olla myös toinen navigaattori, kuten se tässä tapauksessa onkin, jolloin puhutaan sisäkkäisistä navigaattoreista (engl. nested navigators). Juurinavigaattori luodaan ja renderöidään sovelluksen juurikomponentissa.

Kirjautuneen käyttäjän navigaattori käyttää React Navigationin tarjoamaa drawer-navigaattoria, joka näyttää käyttöliittymässä kuvion 34 mukaiselta.



Kuvio 34. Kirjautuneen käyttäjän navigaattori

Jokainen drawer-navigaattorin alkio on oma pinonavigaattorinsa. Esimerkiksi arkiston pinonavigaattorista löytyvät reitit sopimusten listaukselle, yksittäiselle sopimukselle ja sen esikatselulle sekä muokkaamiselle. React Navigation tarjoaa myös tab-navigaattorin, jossa navigointi tapahtuu ruudun ylä- tai alaosassa sijaitsevan palkin välilehdillä.

Siirtyminen näkymien välillä tapahtuu yksinkertaisesti kutsumalla *navigate('reitti')*-funktiota, joka tulee propsina jokaiselle näkymälle. Jotta siirtymisiä näkymien välillä

voidaan tehdä myös näkymien ulkopuolelta, kuten sagoista, toteutettiin sovellukseen navigator-service. Navigator-servicelle annetaan viittaus sovelluksen juurinavigaattoriin Reactin *ref*:in avulla, joka on esitetty kuviossa 35.

```
const RootNavigator = createRootNavigator(signedIn);

return (
  <View>
    <RootNavigator
      onNavigationStateChange={null}
      ref={navigatorRef => {
        NavigatorService.setContainer(navigatorRef);
      }} />
```

Kuvio 35. Viittaus sovelluksen juurinavigaattorista navigator-servicelle

Navigator-serviceen on määritelty funktioita, jotka lähettävät React Navigationin navigointi-actioneja. Funktioita voidaan kutsua mistä vain, kuten sagoista. Kuviossa 36 on esitetty navigator-servicen perus navigointi-funktio, jolle annetaan haluttu reitti sekä mahdolliset query-parametrit.

```
import {NavigationActions} from 'react-navigation';

let _container;

function setContainer(container) {
  _container = container;
}

function navigate(routeName, params) {
  _container.dispatch(
    NavigationActions.navigate({
      type: 'Navigation/NAVIGATE',
      routeName,
      params,
    }),
  );
}
```

Kuvio 36. Navigator-servicen navigointi-funktio

4.9 Push-ilmoitukset

Push-ilmoitukset olivat tärkeä osa mobiilisovellusta. Niiden avulla käyttäjä saa nopeasti tiedon esimerkiksi sopimuksen allekirjoituskutsusta ja pääsee suoraan allekirjoittamaan sen yhdellä painalluksella. Push-ilmoitukset päätettiin toteuttaa Googlen Firebase Cloud Messaging (FCM) -palvelulla. Tärkeimpinä kriteereinä push-ilmoituspalvelulle olivat tuki React Nativelle sekä alustariippumattomuus, bonusta oli palvelun ilmaisuus. Vaihtoehtoinen palvelu FCM:lle oli OneSignal, jota ei kuitenkaan valittu, sillä palvelu tekee rahaa myymällä tietoja käyttäjistä kolmansille osapuolille. Ainut huono puoli FCM:ssä oli se, ettei sillä ole virallista SDK:ta React Nativelle. Ratkaisuksi löytyi kuitenkin *react-native-fcm*-kirjasto, joka on tehty iOS ja Android SDK:n päälle.

FCM:än käyttöönotossa tulee aluksi luoda projekti Firebasen konsolilla. Luonnin jälkeen projektiin lisätään iOS- ja Android-applikaatiot. FCM käyttää iOSilla Apple Push Notification -palvelua (APNs) ilmoitusten lähettämiseen, jota varten tulee luoda todennusavain. Avain voidaan luoda Applen kehittäjätilin kautta, jonka jälkeen se asetetaan Firebasessa iOS-applikaation asetuksiin. Tämän jälkeen voidaan ladata Firebasesta konfiguraatiotiedostot: *google-services.json* Androidille sekä *GoogleService-Info.plist* iOSille. Kyseiset tiedostot asetetaan sovelluksen kansioihin *android/app* ja *ios/projektin-nimi*. Firebasessa voidaan myös luoda palvelinavain (engl. server key), jota tarvitaan viestien lähettämiseen backendistä sovellukseen Firebasen kautta.

Sovellukseen tehtiin ilmoituksia varten oma komponentti, joka renderöidään sovelluksen juurikomponentissa. Kyseisessä komponentissa käsitellään ilmoitusten kuuntelu kolmessa eri tilassa: kun sovellus on suljettuna, taustalla tai avoinna. Komponentissa käsitellään myös eri tyyppisten ilmoitusten avaaminen sekä FCM:än alustaminen. FCM:än alustaminen on esitetty kuviossa 37. Alustaminen tehdään sen

jälkeen, kun käyttäjä on kirjautunut sovellukseen.

```
init() {
  // Initialize FCM on login
  FCM.requestPermissions();
  FCM.getFCMToken().then(fcmToken => {
    const deviceType = Platform.OS;
    dispatch(addDevice({fcmToken, deviceType}));
  });
  dispatch(fcmInitialized());
},
```

Kuvio 37. FCM:än alustaminen

Aluksi käyttäjältä pyydetään lupa vastaanottaa ilmoituksia. Mikäli käyttäjä antaa luvan, haetaan FCM:än palvelimelta käyttäjän laitteen yksilöivä FCM-avain (FCM token). Kyseistä avainta käytetään ilmoitusten lähettämiseen käyttäjän laitteelle. Tämän jälkeen lähetetään action *ADD_DEVICE*, jolle annetaan payloadina avain sekä laitteen tyyppi (Android/iOS). Action otetaan kiinni sagassa ja lähetetään POST-tyyppinen kutsu backendille, jossa käyttäjälle lisätään tietokantaan uusi laite. Tietokannan laite-taulussa on sarake, johon FCM-avain tallennetaan. FCM-avainta käytetään ilmoitusten lähettämiseen backendistä käyttäjän laitteelle.

Ilmoituskomponentin mountattua kutstuaan *load()*-funktiota, jossa lisätään ilmoitusten kuuntelijat ja FCM-päivitysavainkuuntelija, sekä haetaan tieto lukemattomien ilmoitusten määrästä. Funktio on esitetty kuviossa 38.

```
load() {
  // Fetch unread notifications on load
  dispatch(fetchUnread());

  // Set new device token if it refreshes
  this.refreshTokenListener = FCM.on(FCMEvent.RefreshToken, (fcmToken) => {
    const deviceType = Platform.OS;
    dispatch(addDevice({fcmToken, deviceType}));
  });
```

Kuvio 38. Lukemattomien ilmoitusten hakeminen ja FCM-päivitysavainkuuntelija

Tieto lukemattomien ilmoitusten määrästä haetaan backendiltä, joka tapahtuu actionilla *FETCH_UNREAD* kuviossa 38. Backendissä jokainen lähetetty ilmoitus tallennetaan tietokantaan. Ilmoituksen rivillä on kannassa tieto siitä, onko kyseinen ilmoitus luettu. Lukemattomien ilmoitusten määrä näytetään sovelluksen ikonissa sekä navigaation Ilmoitukset-kohdassa. Kuviossa 38 lisätään myös FCM-päivitysavainkuuntelija (engl. refresh token listener). FCM-avain saattaa nimittäin päivittyä, jolloin käyttäjälle tulee lisätä uusi laite.

Seuraavaksi lisätään kuuntelijat ilmoituksille. Ilmoitusten kuuntelussa tulee huomioida kolme eri tilaa: sovellus on suljettuna, taustalla tai avoinna. Kun sovellus avataan painamalla push-ilmoitusta sen ollessa suljettuna, tulee määritellä *getInitialNotification()*-funktio, joka on esitetty kuviossa 39.

```
// App in killed state and opened by tapping notification
FCM.getInitialNotification().then(notif => {
  if (notif && notif.action) {
    switch (notif.action) {
      case 'goToContractSigning':
        dispatch(openSignNotification(notif));
        break;

      case 'goToAcceptChatInvite':
        dispatch(acceptChatInvite(notif));
        break;

      case 'goToAcceptCompanyInvite':
        dispatch(acceptCompanyInvite(notif));
        break;

      case 'goToChat':
        dispatch(openChat(notif));
        break;
    }
  }
});
```

Kuvio 39. Ilmoituksen avaaminen sovelluksen ollessa suljettuna

Kyseinen funktio palauttaa Promisen, jonka tuloksena saadaan ilmoitusobjekti. Ilmoitukset lähetetään backendistä sovellukseen, jossa tämä ilmoitusobjekti on

muodostettu. Ilmoitusobjektille on määritelty *action*, joka kuvaa sitä, mitä tulee tapahtua, kun ilmoitus avataan. Jokaiselle eri tyyppiselle actionille löytyy sovelluksesta vastaava Redux action. Eri tyyppiset actionit käsitellään switch-lausekkeessa, ja lähetetään vastaava Redux action. Ilmoitusobjektin data sisältää tarvittavat asiat ilmoituksen avaamiseen, esimerkiksi allekirjoituskutsun data sisältää avattavan sopimuksen ID:n.

Viimeisenä lisätään kuuntelija ilmoituksille, joka käsittelee ilmoitusten avaamisen sovelluksen ollessa avoinna tai taustalla. Kyseinen kuuntelija on esitetty kuviossa 40.

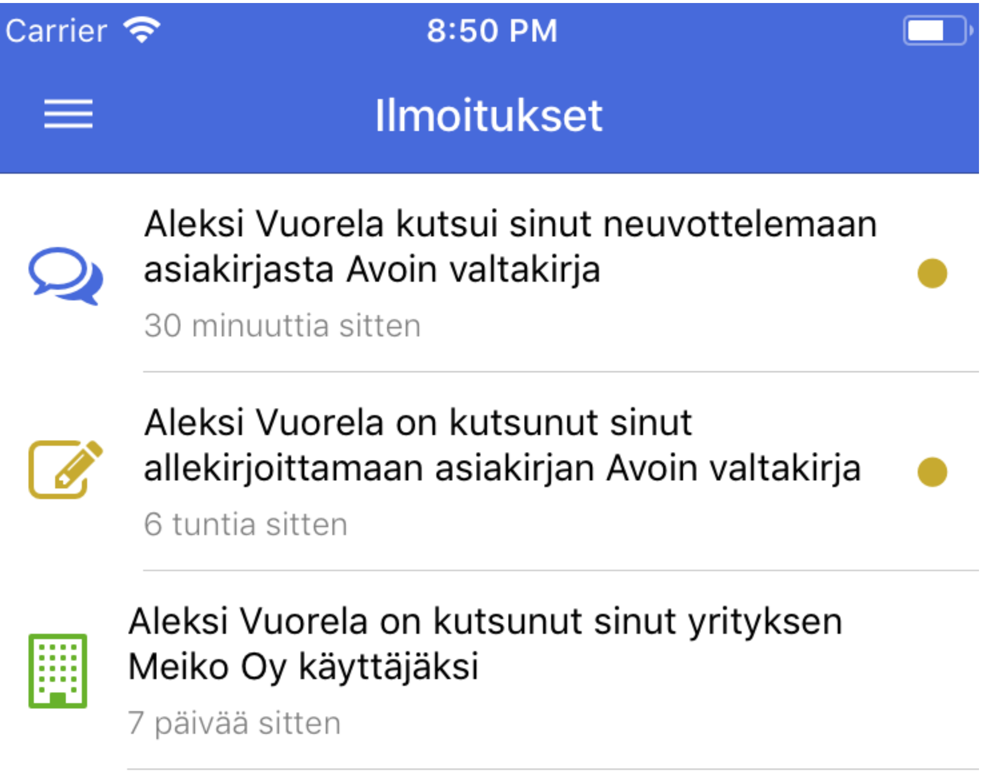
```
// Listen to notifications
this.notificationListener = FCM.on(FCMEvent.Notification, (notif) => {
  if (notif && notif.action && notif.opened_from_tray) {
    switch (notif.action) {
      case 'goToContractSigning':
        dispatch(openSignNotification(notif));
        break;

      case 'goToAcceptChatInvite':
        dispatch(acceptChatInvite(notif));
        break;

      case 'goToAcceptCompanyInvite':
        dispatch(acceptCompanyInvite(notif));
        break;

      case 'goToChat':
        dispatch(openChat(notif));
        break;
    }
  }
  // Foreground notification (Android)
  else if (Platform.OS === 'android' && notif && notif.action && !notif.opened_from_tray) {
    dispatch(setForegroundNotification(notif));
  }
  // Foreground notification <= iOS 9 (iOS 10+ foreground push supported)
  else if (Platform.OS === 'ios' && notif && notif.action && !notif.opened_from_tray) {
    const iosVersion = parseInt(Platform.Version, 10);
    if (iosVersion <= 9) {
      dispatch(setForegroundNotification(notif));
    }
  }
});
```

Kuvio 40. Ilmoitusten kuuntelu sovelluksen ollessa taustalla tai avoinna

React-native-fcm-kirjasto asettaa ilmoitusobjektiin *opened_from_tray*-lipun. Mikäli *opened_from_tray*-lipun arvo on tosi, tiedetään sovelluksen olevan taustalla. Tällöin eri tyyppiset actionit käsitellään samaan tapaan kuin aikaisemmassa switch-lausekkeessa. Mikäli taas *opened_from_tray*-lipun arvo on epätosi, tiedetään

sovelluksen olevan avoinna. Androidissa ei ole tukea esittää ilmoitusta automaattisesti sovelluksen ollessa avoinna, ja iOSissa tuki tälle tuli versiossa 10. Androidissa ja iOS 9:ssä tai vanhemmassa ohjelmoijan tulee siis itse käsitellä ilmoituksen esittäminen sovelluksen ollessa avoinna. Tämä toteutettiin lähettämällä Redux action *SET_FOREGROUND_NOTIFICATION*, joka avaa ilmoituksen sisällön modaalissa. Modaalin 'Jatka'-painikkeesta ajetaan funktio, jossa käsitellään eri tyyppiset actionit switch-lausekkeessa samalla tavalla kuin aikaisemmin.

Sovellukseen tehtiin ilmoituksille oma näkymä, jossa listataan tietokantaan tallennetut ilmoitukset. Sitä kautta ilmoitukset päästään avaamaan myös myöhemmin. Kyseinen näkymä on esitetty kuviossa 41.



Kuvio 41. Ilmoitusten näkymä

Ilmoitusten listaus toteutettiin *react-native-infinite-scroll-view*-kirjastolla. Ilmoituksia haetaan API-kutsulla sivu kerrallaan, kun lista ollaan vieritetty loppuun asti. Kirjastossa on myös tuki *pull-to-refresh*-eleelle, jolla ylöspäin pyyhkäisy listan alussa päivittää sen.

4.10 WebView

WebView tarkoittaa sovellukseen upotettua web-selainta, joka mahdollistaa web-teknologioiden käytön sovelluksessa. Monet mobiilisovellusten kehittämiseen käytetyt sovelluskehykset, kuten Ionic ja Apache Cordova, perustuvat täysin WebViewiin. Jopa työpöytäsovelluksia voidaan tehdä WebViewillä käyttäen työkaluja, kuten Electronia.

Sopimustiedon web-sovelluksessa asiakirjojen allekirjoittaminen tapahtuu *Signature Pad* -nimisellä JavaScript-kirjastolla, joka käyttää HTML5 canvasta. Mobiilisovellukseen haluttiin toteuttaa asiakirjojen allekirjoittaminen samalla kirjastolla, jotta allekirjoitukset näyttäisivät samanlaisilta. Tähän tarvittiin WebViewiä, johon upotetaan canvas ja injektoidaan WebViewiin Signature Pad -kirjaston JavaScript-koodi. Allekirjoitusnäkymä on esitetty kuviossa 42. Näkymässä kaikki muu on renderöity React Nativella, paitsi keskellä oleva piirtoalue, joka on WebViewiin upotettu HTML5 canvas.

Kuvio 42. Allekirjoitusnäkymä

Upottaminen tapahtuu luomalla string-tyyppinen muuttuja, johon tallennetaan kaikki tarvittava HTML-, CSS- ja JavaScript-koodi, ja annetaan se *source*-propsina WebView-komponentille. WebViewin ja sovelluksen tuli pystyä kommunikoimaan toistensa kanssa. Esimerkiksi kun piirtoliike loppuu, halutaan allekirjoituksen data tallentaa komponentin tilaan, tai ‘Tyhjennä’-painikkeella halutaan tyhjentää canvas. Tähän

löytyi ratkaisuksi *react-native-webview-bridge*-kirjasto. Sen avulla voidaan lähettää string-muotoisia viestejä sovelluksesta WebViewiin ja toisinpäin.

4.11 Chatti

Sovellukseen tuli tehdä reaaliaikainen chatti, jossa osapuolet voivat keskustella sopimuksen sisällöstä sen luontivaiheessa. Chatti toteutettiin Pusher-nimisellä palvelulla, sillä myös web-sovelluksen chatti on toteutettu kyseisellä palvelulla. Palvelun avulla mm. chatin toteuttaminen onnistuu paljon nopeammin verrattuna siihen, että tekisi kaiken itse. Käyttäjälle tuli lisäksi saapua push-ilmoitus chattikutsuista sekä uusista viesteistä chattiin.

Pusher on ilmainen palvelu, mikäli yhtäaikaisia yhteyksiä on enintään 100 ja 200 000 viestiä päivässä riittää. Palvelu käyttää WebSocketteja, jotka mahdollistavat reaaliaikaisen viestinnän. Pusherin JavaScript-kirjastosta löytyy tuki React Nativelle. Kirjaston alustaminen on esitetty kuviossa 43.

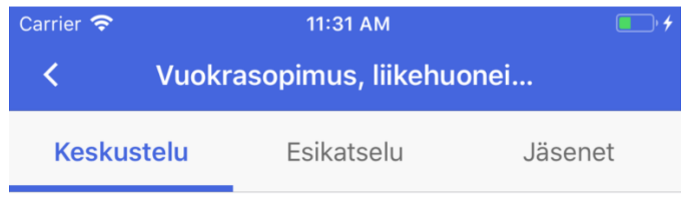
```
// Set Content-Type explicitly as otherwise subscribe doesn't work on real device
apiConfig.headers['Content-Type'] = 'application/x-www-form-urlencoded';

this.socket = new Pusher(appKey, {
  cluster: appCluster,
  encrypted: true,
  auth: {
    headers: {
      ...apiConfig.headers
    }
  },
  authEndpoint: apiConfig.baseURL + '/pusher/auth'
});

// Pusher events
this.channel = this.socket.subscribe(this.props.activeChat.channelName);
this.channel.bind('pusher:subscription_succeeded', (members) => {
  handleSuccesfulSubscription(members);
  this.channel.bind('message-created', (data) => handleIncomingMessage(data));
  this.channel.bind('pusher:member_added', (member) => handleMemberJoined(member));
  this.channel.bind('pusher:member_removed', (member) => handleMemberLeft(member));
});
};
```

Kuvio 43. Pusherin alustaminen

Kuvion 43 alustus tehdään chatin avaamisen yhteydessä. Pusherin konstruktorifunktio avaa yhteyden palveluun, ja se palauttaa socket-objektin.

Konfiguraatiossa on yksi huomio: pyynnön otsakkeisiin tulee asettaa *Content-Typeksi application/x-www-form-urlencoded*. Mikäli tätä ei oltu asetettu, kanavan tilaaminen ei toiminut oikealla laitteella, vaikka se toimikin simulaattorissa. Pusherin (ja WebSocketien) toiminta perustuu tapahtumien kuunteluun sekä callback-funktioihin, jotka suoritetaan tapahtuman saapuessa. Esimerkiksi kun uusi viesti saapuu, callback-funktiossa *handleIncomingMessage()* muutetaan saapuneen viestin data haluttuun formaattiin, jonka jälkeen viesti asetetaan Redux-storeen.

Chatin ulkoasuun käytettiin *react-native-gifted-chat*-kirjastoa, jonka avulla ulkoasu ja käyttökokemus saatiin vastaamaan tuttuja mobiililaitteiden pikaviestipalveluja. Chat-näkymä on esitetty kuviossa 44.

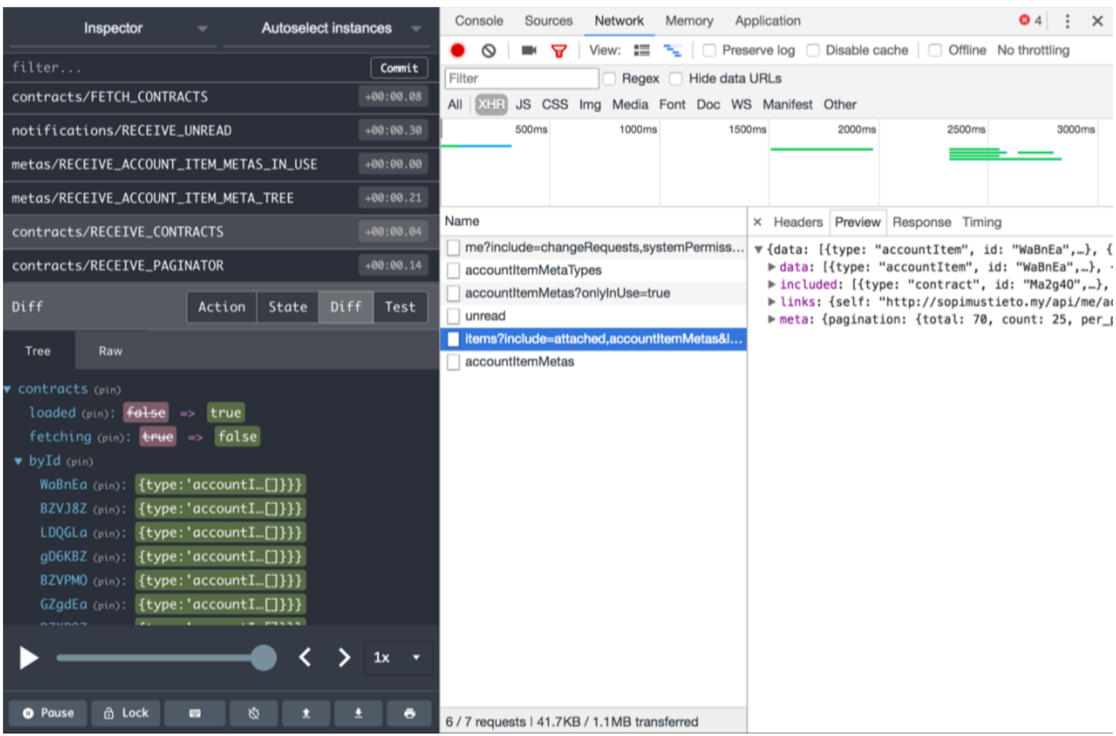


Kuvio 44. Chat-näkymä

Kuvion 44 yläosassa näkyvät välilehdet on toteutettu NativeBase-komponenttikirjaston *Tabs*-komponentilla. Kun esikatselu-välilehti avataan, haetaan API:lta uusin versio laadittavan sopimuksen PDF-tiedostosta. Jäsenet-välilehdeltä nähdään chatin osapuolet sekä tieto siitä, onko osapuoli paikalla tai milloin hän on viimeksi ollut paikalla.

4.12 Työkalut

Työssä käytettyjä työkaluja olivat React Native Debugger sekä Bugsnag. React Native Debugger oli erittäin hyödyllinen työkalu kehityksessä. Työkalu sisältää React Inspectorin, Redux DevToolsin sekä Chromen Network Inspectorin. Se yhdistyy automaattisesti simulaattorissa tai oikealla laittella ajettavaan sovellukseen. Kuvakaappaus React Native Debuggerista on esitetty kuviossa 45.



Kuvio 45. React Native Debugger

Redux DevToolsin avulla nähdään jokainen lähetetty action ja kuinka kyseinen action muutti sovelluksen tilaa. Redux DevTools vaatii sen kytkemisen sovellukseen storeen, joka esitettiin aikaisemmassa kuviossa 20. React Inspectorilla voidaan tarkastella renderöityjä komponentteja, niiden propseja sekä tilaa. Chromen Network

Inspectorilla nähdään sovelluksen lähettämät API-kutsut ja kutsujen vastaukset. Kun sovellus julkaistiin, siihen lisättiin Bugsnag-niminen työkalu, joka raportoi automaattisesti sovelluksen käyttäjillä tapahtuvat virheet. Bugsnag tukee laajalti eri alustoja ja sille löytyykin myös React Native integraatio.

5 Tulokset

5.1 Vaatimusten täyttyminen

Työn tuloksena toteutettiin valmiit mobiilisovellukset iOS- ja Android-alustoille, jotka täyttivät asiakkaan vaatimukset. Vaatimuksina oli toteuttaa suurin osa web-sovelluksen toiminnoista mobiilisovellukseen sekä tietyt uudet ominaisuudet, joiden toteuttamisen mobiilisovellus mahdollisti.

Arkisto

Yksi tärkeimmistä vaatimuksista oli arkisto, jossa käyttäjä voi selata ja hallita sopimusarkistoaan. Arkistossa listataan käyttäjän sopimukset käyttäen ”infinite scroll” -tekniikkaa, jossa listaa alaspäin vierittämällä voidaan käydä läpi kaikki käyttäjän sopimukset ilman sivunvaihtoja. Tällaista listaa on mielekkäämpi käyttää mobiililaitteella, verrattuna web-sovelluksessa käytettyyn listaan, jossa sivua tulee vaihtaa. Arkistosta voidaan hakea asiakirjoja Algolia-hakukoneen avulla. Arkiston listaa voidaan järjestää päivämäärän, muokkauspäivämäärän ja aakkosjärjestyksen mukaan. Listaa voidaan myös suodattaa eri asiakirjatyyppien perusteella sekä käyttäjän itse luomien kansioiden mukaan. Arkiston listausnäkymä on esitetty liitteessä 2.

Arkistossa voidaan avata yksittäinen sopimus omaan näkymäänsä, josta nähdään osapuolten allekirjoitusten tilanne. Sopimuksen PDF-tiedosto voidaan avata suoraan sovelluksessa, ilman että PDF-tiedostoa täytyy ladata laitteelle, kuten Android-käyttäjän täytyy tehdä web-sovelluksessa. Yksittäisen sopimuksen arkistotietoja voidaan muokata (nimi, päivämäärä, lajittelu). Yksittäinen sopimus voidaan myös lähettää sähköpostiin, kopioida toiselle tilille ja poistaa. Yksittäisen sopimuksen näkymä on esitetty liitteessä 3.

Allekirjoittaminen

Toinen tärkeimmistä vaatimuksista oli sopimuksen allekirjoittaminen mobiilisovelluksella. Mobiililaitteella sopimuksen allekirjoittaminen web-sovelluksessa vaatii seuraavat vaiheet:

1. Avaa allekirjoituskutsu sähköpostista tai tekstiviestistä
2. Avaa kutsun linkki selaimessa
3. Kirjaudu sisään tilille
4. Syötä vahvistuskoodi mikäli kaksivaiheinen kirjautuminen on käytössä
5. Syötä allekirjoituskutsun sisältämä PIN-koodi
6. Allekirjoita sopimus.

Mobiilisovelluksella allekirjoittaminen onnistuu huomattavasti helpommin. Allekirjoituskutsusta saapuu push-ilmoitus, jota painamalla päästään suoraan allekirjoittamaan sopimus. Käyttäjän tarvitsee kirjautua sisään ainoastaan kerran, jonka jälkeen hän pysyy kirjautuneena sovellukseen. Käyttäjän ei myöskään tarvitse syöttää PIN-koodia päästäkseen allekirjoittamaan sopimusta.

Mobiilisovelluksella piirretyt allekirjoitukset näyttävät täsmälleen samalta kuin verkkopalvelussa piirretyt allekirjoitukset, sillä ne on toteutettu samalla JavaScript-kirjastolla. Tämä on yksi React Nativen vahvuuksista: sen kanssa voidaan hyödyntää mobiilisovelluksissa valtavaa määrää kirjastoja, joita on tehty JavaScriptille ja Reactille. Mikäli mobiilisovellus olisi toteutettu täysin natiivina, ei allekirjoituksista olisi välttämättä saatu täsmälleen samanlaisia verkkopalvelun kanssa. Allekirjoitusnäkymä esitettiin aikaisemmassa kuviossa 42.

Tietopalvelut

Yhtenä vaatimuksena mobiilisovellukselle oli tietopalveluiden ostaminen. Yritystietopalveluista voidaan hakea yrityksen luottotiedot, nimenkirjoitusoikeudet, yhteisösäännöt ja kaupparekisteriote. Yrityksen laajennetut tiedot ja riskiluokitus nähdään ilmaiseksi. Yrityksiä voidaan hakea Algolia-hakukoneella ja käyttäjälle ehdotetaan hakutermiä vastaavia yrityksiä. Myös yksityishenkilön luottotiedot voidaan hakea henkilötunnuksella. Haetut tiedot avautuvat PDF-tiedostona suoraan

sovelluksessa. Tietopalveluiden näkymä on esitetty liitteessä 4 ja yritystietopalvelujen näkymä liitteessä 5.

Chatti

Mobiilisovellukseen tuli toteuttaa reaaliaikainen chatti, jossa osapuolet voivat keskustella sopimuksen sisällöstä sen luontivaiheessa. Verkkopalvelun chatin käyttäminen mobiililaitteella ei ole kovin mielekästä, mutta mobiilisovelluksessa chatin ulkoasu ja käyttökokemus saatiin vastaamaan tuttuja mobiililaitteiden pikaviestipalveluja. Chatissa nähdään laadittavan sopimuksen PDF-tiedosto, jonka sisältö päivittyy sitä mukaa kun sopimusta laaditaan. Chatissa nähdään myös lista osapuolista sekä tieto siitä, onko osapuoli paikalla tai milloin hän on viimeksi ollut paikalla. Chattikutsuista ja uusista viesteistä chattiin saapuu käyttäjän laitteeseen push-ilmoitus, jota painamalla päästään suoraan chattiin. Chat-näkymä esitettiin aikaisemmassa kuviossa 44.

Kirjautuminen

Sovellukseen kirjautuneen käyttäjän piti pysyä kirjautuneena sisään ikuisesti, ellei hän erikseen kirjaudu ulos. Kun käyttäjä kirjautuu sisään ensimmäistä kertaa laitteella, tallennetaan laitteen muistiin päivitysavain (engl. refresh token), joka on käytännössä ikuinen. Päivitysavaimen avulla voidaan hakea väliaikainen pääsyavain (engl. access token), jonka avulla käyttäjä tunnistetaan. Kyseiset avaimet tallennetaan laitteeseen salattuna turvalliseen paikkaan, eli iOSissa Keychainiin ja Androidissa Keystoreen. Lisäksi osa Redux-storesta tallennetaan laitteen muistiin AsyncStorageen, jonka avulla esimerkiksi viimeksi valittu käyttäjätili on heti aktiivisena kun sovellus avataan. Näin mobiilisovellukseen tarvitsee kirjautua vain kerran, toisin kuin web-sovelluksessa, jossa kirjautuminen vanhenee tietyn ajan kuluttua. Mobiilisovellukseen tehtiin myös kaksivaiheinen kirjautuminen, jossa käyttäjälle saapuu tekstiviestinä vahvistuskoodi salasanan syöttämisen jälkeen. Mobiilisovelluksessa voidaan vaihtaa tiliä sekä kirjautua ulos. Kirjautumisnäkymä on esitetty liitteessä 1.

Asiakaspalvelu

Mobiilisovellukseen tuli toteuttaa asiakaspalvelu samaan tapaan kuin verkkopalveluun käyttäen Intercom-alustaa. Mobiilisovellukseen laitettiin Intercomin React Native -kirjasto ja konfiguroitiin asiakaspalvelun lähettämistä viesteistä saapumaan push-ilmoitus käyttäjän laitteeseen.

Asetukset

Viimeisenä vaatimuksena mobiilisovellukselle oli käyttäjän ja yrityksen asetusten hallinta. Käyttäjä voi muokata sovelluksessa perustietojaan, sekä vaihtaa sähköpostia, puhelinnumeroa ja salasanaa. Käyttäjä voi piirtää ja tallentaa oman allekirjoituksensa asetuksissa, jolloin tallennettu allekirjoitus tulee automaattisesti sopimuksen allekirjoitustilanteessa. Käyttäjä voi luoda asetuksissa omia kansioita, joihin hän voi lajitella sopimuksiaan. Kansioita voidaan myös muokata ja poistaa. Lisäksi käyttäjä näkee asetuksista kuitit ostotapahtumistaan.

Yrityksen asetuksissa voidaan niin ikään muokata yrityksen perustietoja, nähdä kuitit ostotapahtumista ja hallita kansioita. Yrityksen asetuksissa voidaan kutsua käyttäjiä yritystilille, sekä muokata yrityksen käyttäjien rooleja ja poistaa käyttäjiä yrityksestä. Myös rooleja voidaan hallita yrityksen asetuksissa, eli luoda uusia rooleja, asettaa oikeuksia rooleille ja poistaa rooleja. Asetusten päänäkymä on esitetty liitteessä 6.

5.2 Julkaisu

Valmiit sovellukset julkaistiin Applen ja Googlen sovelluskaupoissa. Sovelluksen julkaiseminen Googlen sovelluskauppaan onnistui ensimmäisellä yrityksellä, mutta julkaisu Applen sovelluskauppaan vaati kolme yritysä. Googlella sovellusten tarkitusprosessi on pitkälti automatisoitu, mutta Applella jokainen sovellus tarkistetaan manuaalisesti ihmisen toimesta, minkä takia tarkistusprosessi on tiukempi.

Ensimmäisellä kerralla hylkäämisen syynä oli sovelluksen kirjautumisnäkymässä oleva linkki rekisteröitymiseen palvelun verkkosivun kautta. Rekisteröityminen jätettiin pois mobiilisovelluksesta, sillä yritystilin rekisteröiminen palveluun on maksullista, ja sovelluksen sisäiset maksut tulee tehdä Applen kautta, josta he ottavat itselleen 30 %

palkkiota. Tätä yritettiin kiertää laittamalla linkki rekisteröitymiseen verkkosivun kautta, mutta yritys ei mennyt tarkistusprosessin läpi. Toisella kerralla hylkäämisen syynä oli vajaavaiset metatiedot, mutta kolmannella yrityksellä sovellus läpäisi tarkistusprosessin ja se saatiin julkaistua. Jokainen tarkistusprosessi kesti yli viikon, joten mikäli sovellus tulee saada julkaistua tiettynä päivänä, se kannattaa laittaa tarkistukseen hyvissä ajoin.

6 Pohdinta

6.1 Kehitysvauhti

Minulla ei ollut lainkaan aikaisempaa kokemusta mobiilisovellusten kehittämisestä ennen tätä työtä. Työn valmistumiseen meni noin neljä kuukautta kokopäiväistä työtä. Tässä ajassa toteutettiin julkaisukuntoiset sovellukset iOSille ja Androidille. Työ sisälsi myös käyttöliittymien suunnittelun ja osan tarvittavista backend-muutoksista. Mielestäni tämä on hyvä esimerkki siitä, kuinka hurja kehitysvauhti React Nativella saavutetaan. React Nativen avulla voidaan toteuttaa mobiilisovellukset kummallekin alustalle lähes samalla koodikannalla, eikä kahta eri ohjelmointikieltä tarvitse opetella. React Nativessa voidaan myös hyödyntää valtavaa määrää Reactille ja JavaScriptille tehdyistä kirjastoista, jotka ratkaisevat yleisiä ongelmia ohjelmoijan puolesta.

Nopeuteen vaikutti tietysti myös se, että Sopimustiedon web-sovellus oltiin toteutettu Reactilla ja samoilla teknologiavalinnoilla, minkä takia aikaisempaa koodia voitiin hyödyntää työssä. Tämä taas antaa hyvän esimerkin siitä, miksi React on loistava valinta frontend-sovelluskehykseksi. Reactilla hyvin suunnitellusta web-sovelluksesta saadaan tarpeen tullen toteutettua myös mobiilisovellus nopeasti React Nativella.

6.2 React ja teknologiavalinnat

React ja siihen liittyvät teknologiavalinnat (Redux, Redux-Saga) olivat minulle entuudestaan tuttuja Sopimustiedon web-sovelluksen kehittämisestä. Aikaisemmin olin kuitenkin vain käyttänyt niitä, mutta työn aikana pääsin tutustumaan niihin

syvällisemmin. Erityisesti pääsin tarkastelemaan syitä, minkä takia niitä ylipäätään käytetään, eli mitä ongelmia ne ratkaisevat. En myöskään aikaisemmin ollut juurikaan perehtynyt niiden konfigurointiin, mutta työn aikana tämäkin osa-alue tuli tutuksi kun projektia lähdettiin tekemään nollasta.

Työssä pääsin myös miettimään ensimmäistä kertaa kunnolla Reactiin liityvää optimointia. Suorituskyvystä on harvoin aiheutunut mitään ongelmia web-kehityksessä, minkä takia se onkin jäänyt vähän taka-alalle. Mobiilisovelluksessa suorituskyvystä aiheutui käyttäjälle asti näkyviä ongelmia, kun esimerkiksi siirtymäanimaatiot tökkivät komponentin renderöinnin yhteydessä tapahtuvien operaatioiden takia. Näihin ongelmiin auttoi mm. turhien uudelleenlaskentojen välttäminen komponentin renderöinnissä. React Nativella ei päästä ihan yhtä hyvään suorituskykyyn kuin täysin natiivilla sovelluksella, mutta hyvin lähelle sitä.

6.3 Käyttöliittymät

Kaikki käyttöliittymät täytyi suunnitella ja toteuttaa uudestaan, sillä React Native ei käytä HTML:ää sovelluksen renderöintiin tai CSS:ää tyylittelyyn. Tämä oli minulle täysin uusi osa-alue, ja työtä hankaloitti myös se, että iOS- ja Android-käyttäjät olettavat käyttöliittymien näyttävän alustoille ominaisilta. Käytin suunnittelussa apuna web-sovelluksen responsiivista mobiiliversiota sekä ottamalla mallia muista mobiilisovelluksista. Käyttöliittymistä saatiin hyvin pitkälti natiivin näköisiä NativeBase-komponenttikirjaston avulla ja pyrinkin suunnittelemaan käyttöliittymät sen sisältämien valmiiden UI-komponenttien avulla. Suunnittelutyössä olisi kannattanut käyttää apuna jotain graafista työkalua, sillä suunnittelu tuli pitkälti tehtyä koodaamalla elementit paikoilleen. Siinä vaiheessa tuli usein huomattua, ettei tämä toiminutkaan, ja kaikki piti tehdä uudestaan.

Käyttöliittymiä tehdessä myös flexbox-tekniikan käyttö tuli hyvin tutuksi, sillä se on React Nativessa käytännössä ainut tapa asemmoida elementtejä. Tekniikka onkin hyvin tehokas ja joustava tähän tarkoitukseen, ja sen opettelusta on ollut valtavasti apua myös web-kehityksessä. Käyttöliittymien suunnittelu pelkästään mobiilille opetti myös paljon muita asioita, mitä ei ole tullut juurikaan aikaisemmin web-kehityksessä ajateltua. Esimerkiksi kaikkien painettavien elementtien, kuten

nappuloiden, tulisi Applen ohjeistuksen mukaan olla vähintään 44 pikseliä leveitä ja korkeita. Toinen esimerkki on lomakkeet, joissa mobiililaitteiden virtuaalinen näppäimistö vie paljon näyttötilaa. Käyttöliittymää on mukautettava näppäimistön avautuessa niin, ettei oleellinen tieto jää sen alle. Myös animaatiot, esimerkiksi siirtymien yhteydessä, ovat tärkeässä roolissa mobiilisovelluksissa. Onneksi React Navigation -kirjasto hoiti siirtymäanimaatiot ohjelmoijan puolesta, sillä ne muutamat animaatiot, jotka jouduin itse tekemään, olivat työläitä ja veivät paljon aikaa.

6.4 Haasteet

Työn haastavimpana osuutena olivat push-ilmoitukset. Suurin syy tähän oli työssä käytetyn react-native-fcm-kirjaston sekavuus ja puutteellinen dokumentaatio. Kirjasto vaati myös paljon konfigurointia natiivikoodin tasolla käyttäen Javaa ja Objective-C:tä. Kirjastosta aiheutui lisäksi konflikti työssä käytetyn Intercom-kirjaston push-ilmoitusten kanssa, minkä takia jouduin tekemään react-native-fcm-kirjastosta oman kopion (engl. fork) ja muokkaamaan sen lähdekoodia yhteensopivaksi natiivilla tasolla Androidin osalta. Firebase Cloud Messaging -palvelu päivittyi myös tiheään tahtiin kehityksen aikana, eikä react-native-fcm-kirjaston ylläpito pystynyt vastaamaan muutoksiin riittävän nopeasti. Ratkaisuja eri ongelmiin joutuikin usein etsimään Githubin ongelmanseurannasta (engl. issue tracker). Kirjaston Github-sivulla lukeekin nykyään, että he suosittelevat react-native-firebase-kirjaston käyttöä, jolla pystytään tekemään samat asiat ja sen kehitys on aktiivisempaa. Näillä kokemuksilla suosittelenkin vahvasti käyttämään ennemmin kyseistä kirjastoa.

Työn toinen haastava, mutta myös kaikista mielenkiintoisin osuus, oli kirjautumisen toteuttaminen. Backendin olemassa oleva järjestelmä oli toteutettu väliaikaisilla pääsyavaimilla (engl. access token). Pääsin tekemään järjestelmään tuen päivitys-avaimille (engl. refresh token) sekä hyödyntämään laitteen muistia avaimien ja Redux-storen tallentamiseen, jotta käyttäjä pysyy kirjautuunena sovellukseen myös sen sulkemisen jälkeen.

6.5 Yhteenveto ja jatkokehitys

Yhteenvetona olen hyvin tyytyväinen työn lopputulokseen ja se oli erittäin opettavainen kokemus. Työssä pääsin tutustumaan ensimmäistä kertaa mobiilikehityksen maailmaan ja vahvistamaan ammatillista osaamistani usealla eri osa-alueella. Työ opetti myös itsenäiseen työskentelyyn ja oma-aloitteisuuteen, sillä kukaan ei ollut kertomassa, mitä pitäisi tehdä seuraavaksi.

Työn lopputuloksena saatiin julkaistua valmiit iOS- ja Android-sovellukset, jotka täyttivät asiakkaan vaatimukset. Mobiilisovelluksen kehitys jatkuu web-sovelluksen kehityksen rinnalla hyödyntämällä palvelussa yhä enemmän mobiililaitteiden tarjoamia mahdollisuuksia. Yksi tällainen jatkokehitysidea oli hyödyntää mobiililaitteen kameraa paperisten sopimusten skannamiseen sovelluksessa, jonka jälkeen sopimuksen voisi allekirjoittaa ja arkistoida sähköisesti. Sovelluksen automaatiotestaus olisi myös hyvä toteuttaa tulevaisuudessa.

Lähteet

About FCM Messages. 2018. Dokumentaatio FCM-viesteistä. Viitattu 10.3.2018. https://firebase.google.com/docs/cloud-messaging/concept-options.

About FCM Server. 2018. Dokumentaatio FCM-palvelimesta. Viitattu 10.3.2018. https://firebase.google.com/docs/cloud-messaging/server.

Actions. 2018. Dokumentaatio Reduxin actionien usein kysytyistä kysymyksistä. Viitattu 3.8.2018. https://redux.js.org/faq/actions.

AsyncStorage. N.d. Dokumentaatio React Nativen AsyncStoragesta. Viitattu 10.6.2018. https://facebook.github.io/react-native/docs/asyncstorage.html.

Banks, A. & Porcello, E. 2017. Learning React. Sebastopol: O'Reilly Media.

Code Signing. N.d. Kuvaus koodin allekirjoittamisesta Applen kehittäjäsivustolla. Viitattu 22.4.2018. https://developer.apple.com/support/code-signing/.

Crockford, D. 2008. JavaScript: The Good Parts. Sebastopol: O'Reilly Media.

Eisenman, B. 2017. Learning React Native. 2. p. Sebastopol: O'Reilly Media.

Firebase Cloud Messaging. 2018. Johdanto Firebase Cloud Messaging dokumentaatiossa. Viitattu 17.3.2018. https://firebase.google.com/docs/cloud-messaging/.

Gackenheimer, C. 2015. Introduction to React. New York: Apress.

Haverbeke, M. 2014. Eloquent JavaScript. 2. p. San Francisco: No Starch Press.

Hello React Navigation. N.d. Johdanto React Navigation -kirjastoon. Viitattu 19.8.2018. https://reactnavigation.org/docs/en/hello-react-navigation.html.

Herrera, E. 2017. Understanding redux-saga: From action creators to sagas. Viitattu 10.3.2018. https://blog.logrocket.com/understanding-redux-saga-from-action-creators-to-sagas-2587298b5e71.

Me. 2018. Kuvaus Meiko-yrityksestä. Viitattu 20.11.2018. https://meiko.fi/me.

Idiomatic Redux: Why use action creators? 2016. Artikkeli action creatorien käytön hyödyistä. Viitattu 3.8.2018. https://blog.isquaredsoftware.com/2016/10/idiomatic-redux-why-use-action-creators/.

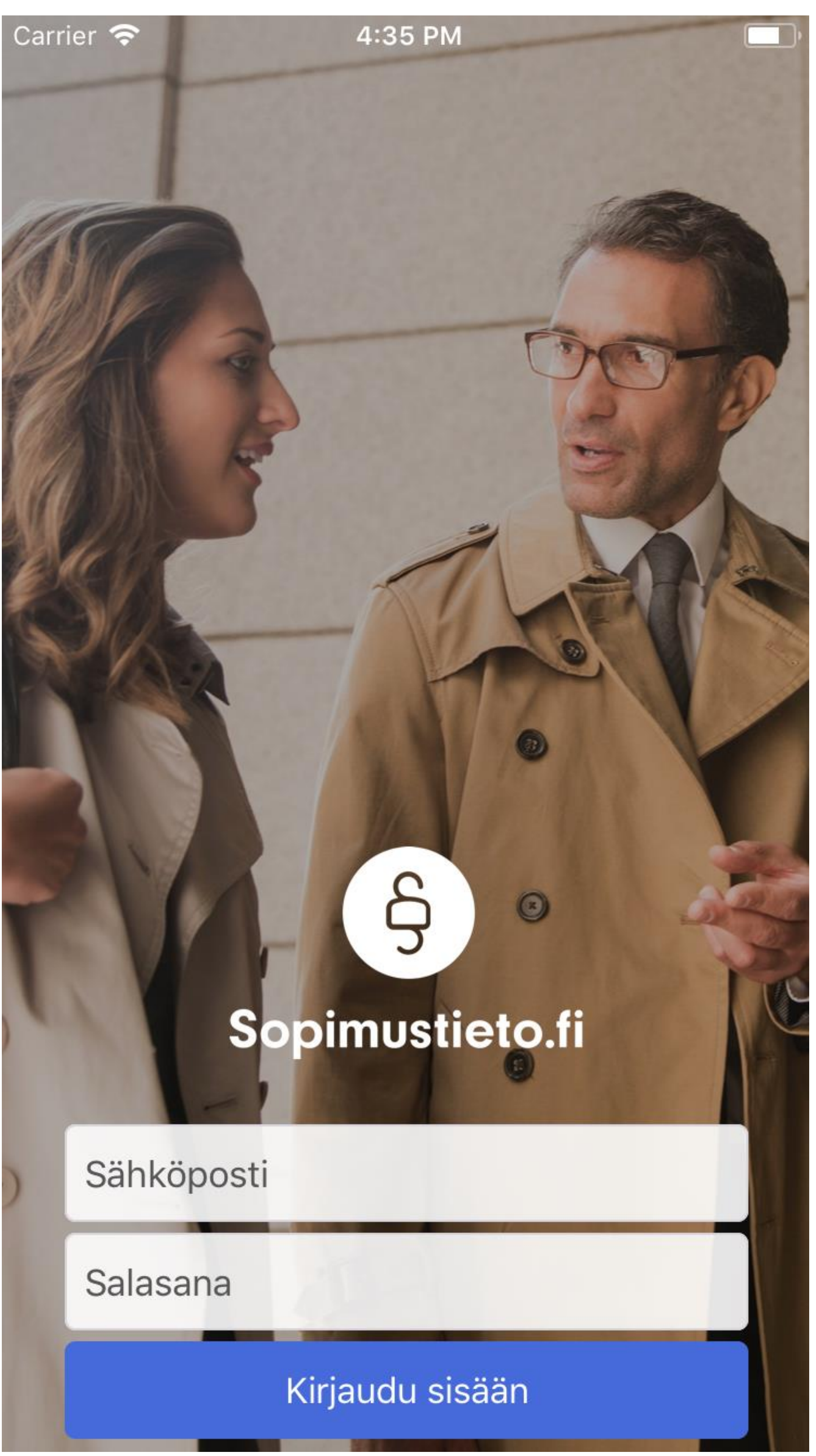
Initializing State. 2018. Dokumentaatio Reduxin tilan alustamisesta. Viitattu 2.8.2018. https://redux.js.org/recipes/structuringreducers/initializingstate.

Reducers. 2018. Dokumentaatio reducereista. Viitattu 3.8.2018. https://redux.js.org/basics/reducers.

Redux-Saga Interview. 2016. Haastattelu SurviveJS blogissa. Viitattu 11.2.2018. https://survivejs.com/blog/redux-saga-interview/.

Liitteet

Liite 1. Kirjautumisnäkymä



Liite 2. Arkiston listausnäkymä

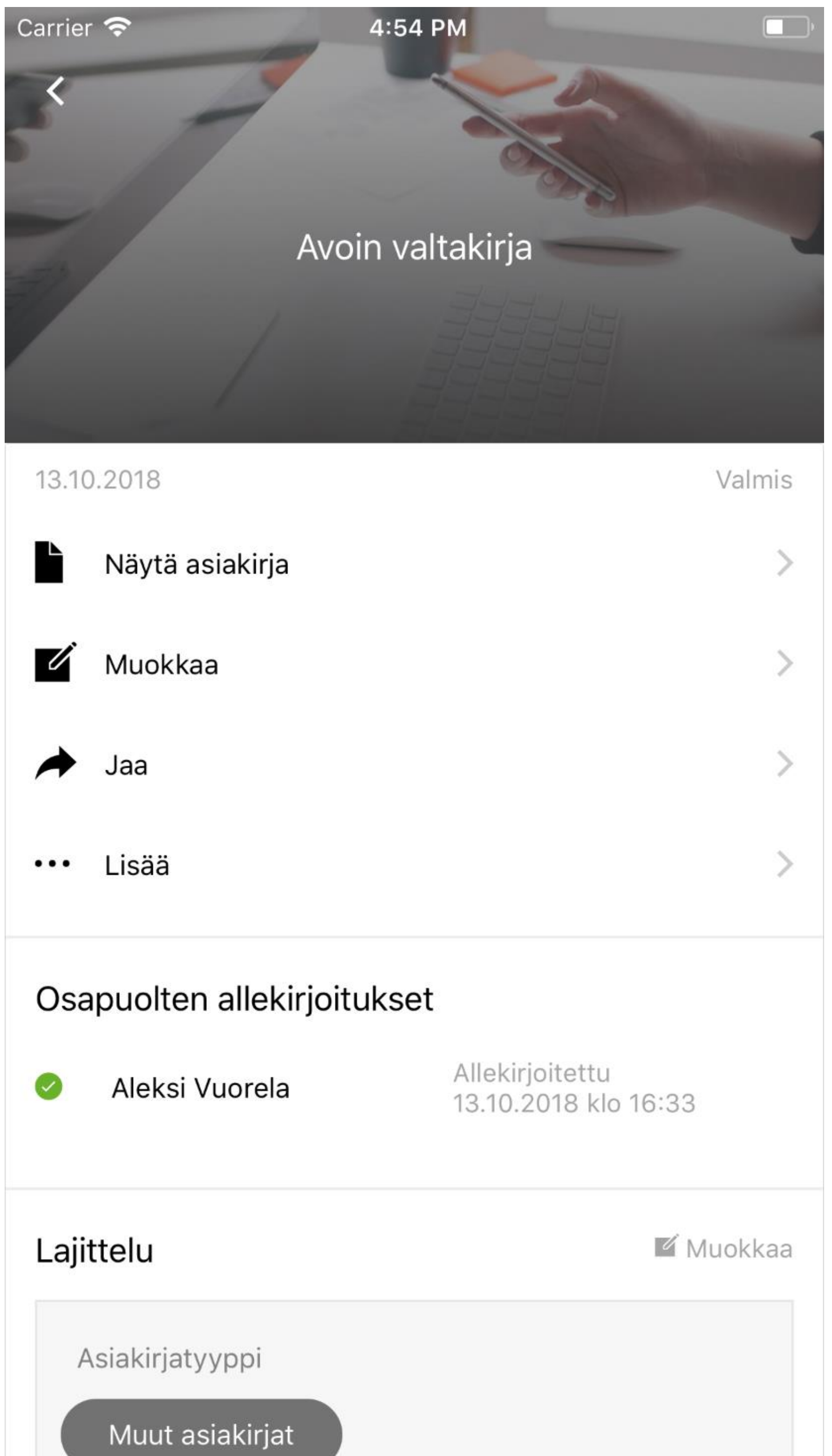
Carrier 5:22 PM

Arkisto

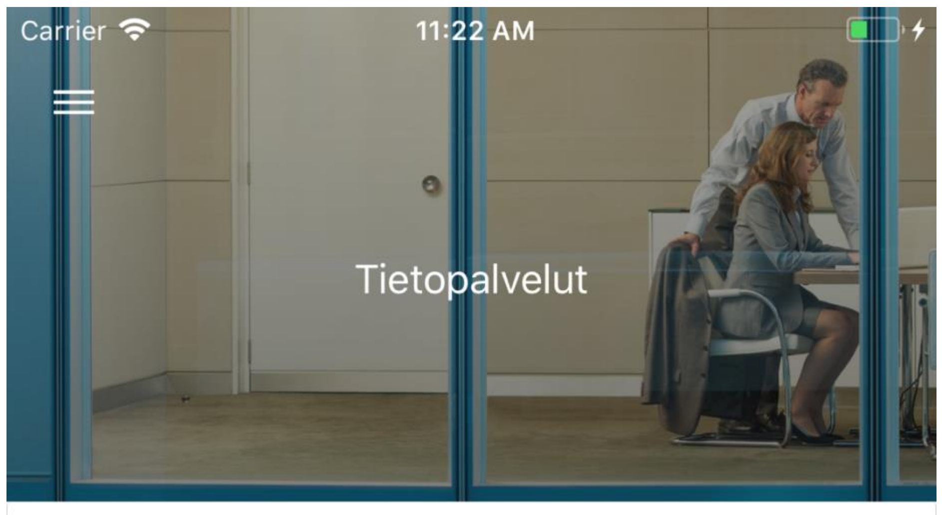
Uusin ensin Suodata

Avoin valtakirja	13.10.2018
Oma sopimus	12.10.2018
Avoin valtakirja	12.10.2018
Avoin valtakirja	18.09.2018
Lahjakirja, kiinteistö	17.09.2018
Avoin valtakirja	17.09.2018
Avoin valtakirja	14.09.2018
Avoin valtakirja	14.09.2018
Avoin valtakirja	13.09.2018
Avoin valtakirja	13.09.2018
Avioehtosopimus yritysvarallis...	13.09.2018
Avioerohakemus, I-vaihe	13.09.2018
Avoin valtakirja	13.09.2018

Liite 3. Yksittäisen sopimuksen näkymä



Liite 4. Tietopalveluiden näkymä



Yrityshaku ×

Yritystietopalvelut

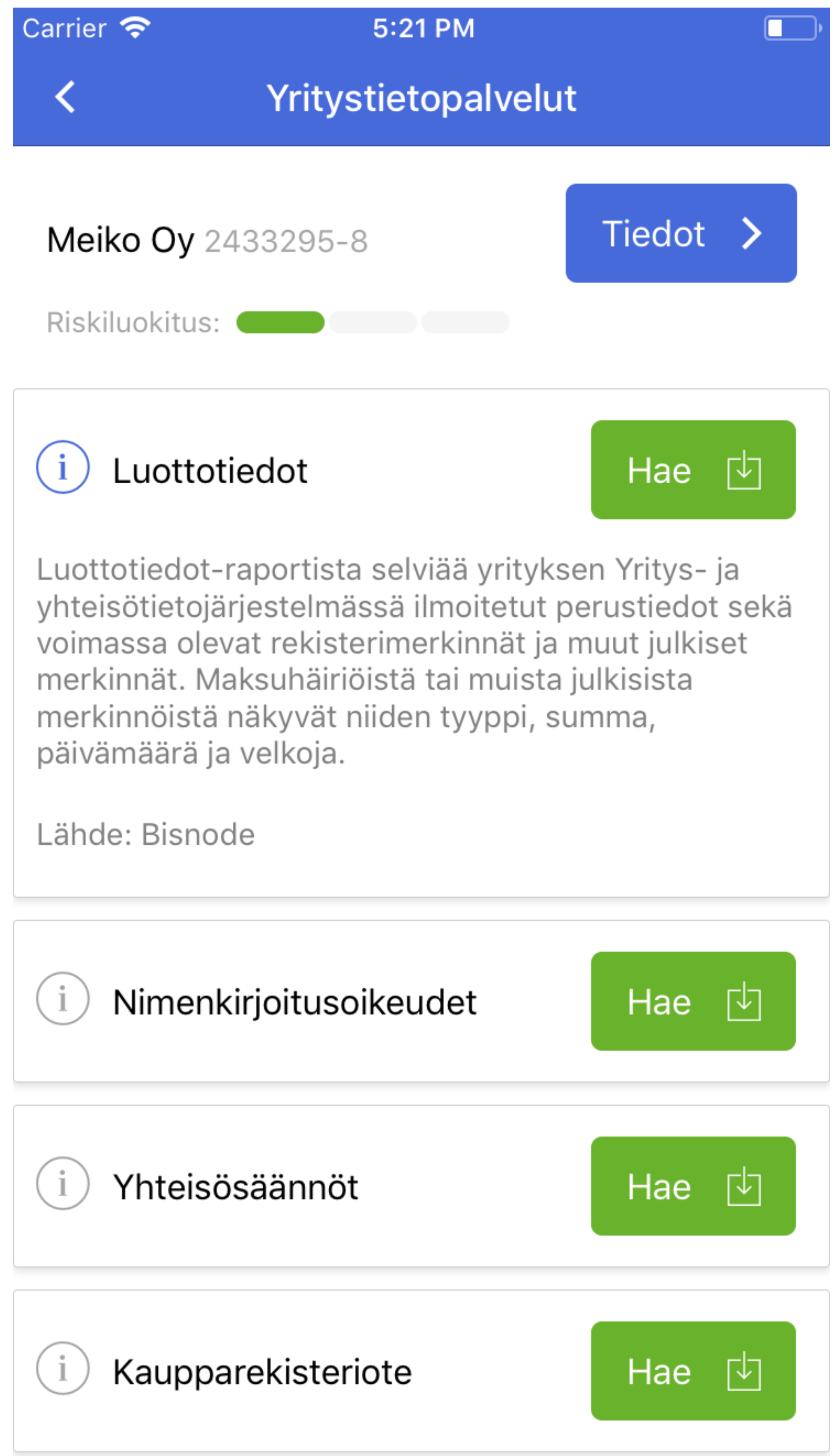
Tietopalvelumme avulla selvität yrityksen Patentti- ja rekisterihallituksen rekisteritiedot sekä luottotiedot.

Henkilötietopalvelut

Selvitä omat tai toisen osapuolen luottotiedot ja mahdolliset maksuhäiriömerkinnät sekä niiden vanhenemisajat.

Siirry luottotietohakuun >

Liite 5. Yritystietopalveluiden näkymä



Liite 6. Asetusten päänäkymä