Mirka Poikelin

MOBIILISOVELLUKSEN KEHITTÄMINEN MICRO FRONTEND -ARKKITEHTUURILLA

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittelyn koulutus

2023

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Tradenomi (AMK)
Tekijä	Mirka Poikelin
Työn nimi	Mobiilisovelluksen kehittäminen micro frontend -arkkitehtuurilla
Toimeksiantaja	Riversoft Oy
Vuosi	2023
Sivut	46 sivua
Työn ohjaaja	Janne Turunen

TIIVISTELMÄ

Tämän opinnäytetyön tavoitteena oli suunnitella ja toteuttaa olemassa olevaan toiminnanohjausjärjestelmään perustuva mobiilisovellus micro frontend -arkkitehtuurilla. Tarkoituksena oli tutkia arkkitehtuurin sovellettavuutta mobiilisovelluksissa toteuttamalla yksinkertainen tavaran- ja kalustonhallintasovellus React Native -ohjelmistokehyksen avulla iOS ja Android -alustoille.

Työn toimeksiantajana toimi Riversoft Oy, joka valmistaa siivous- ja kiinteistönhuoltoalan yrityksille suunnattua web-sovelluspohjaista toiminnanohjausjärjestelmää. Toimeksianto syntyi tarpeesta parantaa sovelluksen saavutettavuutta myös niille käyttäjäryhmille, jotka tarvitsevat vain tiettyjä järjestelmän osia päivittäisessä työssään tai joutuvat työskentelemään olosuhteissa, joissa ei ole mahdollisuutta käyttää tietokonetta tai toimivaa verkkoyhteyttä. Järjestelmän laajuus ja monimutkaisuus toi mukanaan myös kiinnostuksen selvittää samalla micro frontend -arkkitehtuurin käyttömahdollisuuksista mobiilisovelluksessa.

Opinnäytetyön tuloksena syntyi toimeksiannon mukainen mobiilisovellus. Sovelluksen arkkitehtuuri toteutettiin Re.Pack-paketointityökalun avulla, joka tukee Webpack 5:n Module Federation -lähestymistapaa. Toteutuksen pohjalta voitiin todeta micro frontend -arkkitehtuurin muistuttavan suurilta osin web-pohjaisten sovellusten toteutustapaa, joten hyödyntäminen oli vähäiselläkin kokemuksella yllättävän sujuvaa ja käytetyt työkalut vaikuttivat nuoresta iästään huolimatta vakailta. Micro frontend -arkkitehtuurin monimutkaisuuden ja toistuvien työvaiheiden vuoksi voitiin todeta sen soveltuvan paremmin laajoihin projekteihin tai suurille organisaatioille, jotka pystyvät paremmin hyödyntämään arkkitehtuurin tuomat edut.

Asiasanat: ohjelmistoarkkitehtuuri, mobiilisovellukset, toiminnanohjausjärjestelmä, micro frontend, React Native

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Business Administration
Author	Mirka Poikelin
Thesis title	Mobile application development with micro frontend architecture
Commissioned by	Riversoft Oy
Time	2023
Pages	46 pages
Supervisor	Janne Turunen

ABSTRACT

The goal of this thesis was to design and implement a mobile application based on an existing ERP system with a micro frontend architecture. The purpose was to investigate the applicability of the architecture in mobile applications by implementing a simple inventory management application for iOS and Android platforms using the React Native framework.

The commissioner of this project was Riversoft Oy, a manufacturer of a web-based enterprise resource planning system aimed at companies in the cleaning and property maintenance industry. The assignment arose from the need to improve the accessibility of the application also for those user groups who only needed certain features of the system in their daily work or must work in conditions without the possibility to use a computer or a working network connection. The scope and complexity of the system also brought an interest in exploring the possibilities of using the micro frontend architecture in a mobile application.

As a result of the thesis, a mobile application was created according to the assignment. The architecture of the application was implemented using the Re.Pack toolkit which supported the Webpack 5's module federation approach. Based on the implementation, it could be stated that the micro frontend architecture was largely a reminiscent of the way web-based applications were implemented, therefore the utilization was surprisingly smooth even with a little experience and the tools used seemed stable despite their young age. Due to the complexity and repetitive work steps of the micro frontend architecture, it could be concluded that it would be more suitable for large-scale projects or large organizations that are able to utilize the advantages brought by the architecture.

Keywords: software architecture, mobile applications, enterprise resource planning system, micro frontend, React Native

SISÄLLYS

1 JOHDANTO 5
2 MIKROPALVELUARKKITEHTUURI 6
2.1 Micro frontend -arkkitehtuuri 7
2.2 Mikropalvelut mobiilisovelluksissa 8
3 TYÖSSÄ KÄYTETTÄVÄT TEKNOLOGIAT 10
3.1 REST API 10
3.2 MongoDB Realm 12
3.3 React 13
3.4 Webpack 5 ja Module federation 14
3.5 React Native 15
3.6 Re.Pack-paketointityökalu 16
4 SUUNNITTELU 18
4.1 Toimeksiannon lähtökohdat 18
4.2 Toimeksiannon tavoitteet ja rajaus 19
4.3 Sovelluksen ulkoasu 20
4.4 Sovelluksen arkkitehtuuri ja rakenne 21
4.5 Navigointi ja kirjautuminen 23
4.6 Tavaroiden ja kaluston hallinta 24
5 TOTEUTUS 25
5.1 Projektin luominen 25
5.2 Käyttäjän todennus ja navigointi 29
5.3 Tavaran- ja kalustonhallinta 35
5.4 Seuranta 38
6 LOPPUTULOKSEN ARVIOINTI 40
7 PÄÄTÄNTÖ 43
LÄHTEET 45

1 JOHDANTO

Tämän opinnäytetyön tavoitteena on suunnitella ja toteuttaa olemassa olevaan toiminnanohjausjärjestelmään integroitava tavaroiden ja kaluston hallinta- ja seurantasovellus micro frontend -arkkitehtuurilla. Sovellus toteutetaan React Native -ohjelmistokehyksen avulla iOS- ja Android-alustoille, ja nykyisen järjestelmän kanssa kommunikoiminen toteutetaan REST API -ohjelmointirajapinnan kautta. Micro frontend -arkkitehtuurin soveltuvuuteen mobiilisovellusten kehittämisessä perehdytään toteuttamalla sovellus Webpack 5:n Module Federation -lähestymistapaa käyttäen.

Opinnäytetyö toteutetaan produktiivisena kehittämistyönä. Teoreettiset lähtökohdat koostuvat pääasiassa kansainvälisestä kirjallisuudesta ja virallisista dokumentaatioista. Opinnäytetyöraportti alkaa teoriaosalla, jossa käsitellään työn keskeisimmät käsitteet ja menetelmät. Luvussa 2 käsitellään micro frontend -arkkitehtuurin taustaa, lähestymistapaa sekä sovellettavuutta mobiilisovellusympäristöissä ja luvussa 3 käydään tarkemmin läpi tekniseen toteutukseen käytettävät työkalut ja menetelmät perusteluineen. Arkkitehtuuria käsitellään aiheen monimutkaisuuden ja laajuuden vuoksi tässä raportissa vain hyvin pintapuolisesti.

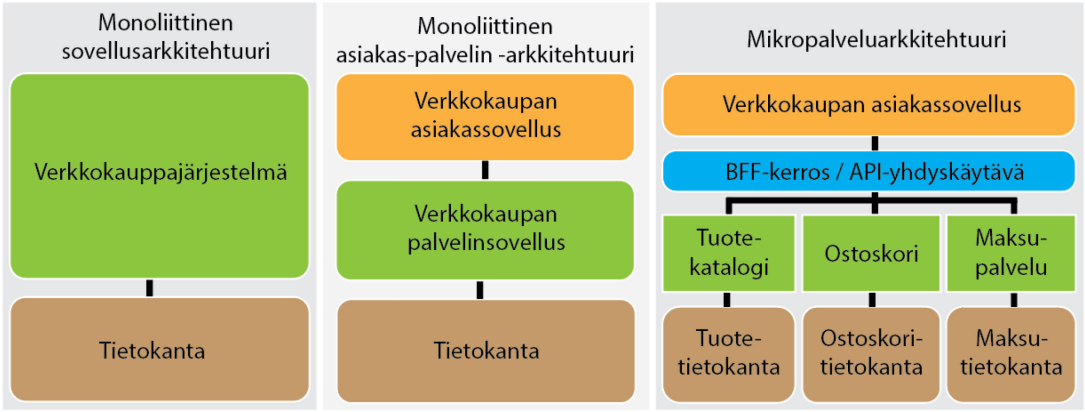
Teorian jälkeen jatketaan kehittämisosuudella, jossa käsitellään sovelluksen tekninen toteuttaminen teoriaosassa esiteltyjen menetelmien avulla. Neljännessä luvussa käsitellään käyttöliittymän ja sovelluksen rakenteen suunnittelu ja viidennessä luvussa käydään vaiheittain läpi varsinainen tekninen toteutus.

Luvussa 6 käydään läpi kehittämistyölle asetettuja tavoitteita ja arvioidaan niiden toteutumista peilaten niitä lopputuotoksen onnistumiseen. Lisäksi tarkastellaan mahdollisia jatkokehitystarpeita sekä arvioidaan lisäksi micro frontend -arkkitehtuurin vaikutuksia ja toimivuutta mobiilisovellusten kehittämisessä. Lopuksi käydään läpi yhteenveto projektin toteutuksesta ja siihen liittyvistä kokemuksista.

2 MIKROPALVELUARKKITEHTUURI

Modernin sovelluskehityksen myötä sovelluksista on tullut tehokkaampia, mutta sen seurauksena ne ovat myös monimutkaistuneet merkittävästi. Perinteisesti sovellukset on toteutettu ns. monoliittisina sovelluksina, jossa kaikki sovelluksen loogiset kerrokset ja toiminnot muodostavat yhtenäisen koodikannan, jota kehitetään ja hallitaan yhtenä kokonaisuutena. Nykyisin sovelluksia tulee voida ylläpitää ja kehittää yrittäen samalla vastata asiakkaiden tarpeisiin sekä liiketoiminnan ja asiakasmäärien muutoksista johtuviin olosuhteiden muutoksiin. Sovellusten tulisi pystyä skaalautumaan tarpeen mukaan eri suuruisille käyttäjämäärille, ja niiden tulisi samalla olla vikasietoisia sekä jatkuvasti saatavilla. Näiden haasteiden hallitseminen monoliittisessa koodikannassa on epäkäytännöllistä ja vaikeaa, minkä vuoksi alettiin kiinnostua tavoista jakaa sovelluksia pienempiin ja paremmin hallittaviin osiin. Tällaista mallia kutsutaan mikropalveluarkkitehtuuriksi. (Davis 2021, 1–2; Netkow s.a.)

Mikropalveluarkkitehtuuri noudattaa yhden vastuun periaatetta (Single Responsibility Principle), jonka perusteella samaan vastuualueeseen kuuluvat toiminnot rajataan erillisiksi, itsenäisesti toimiviksi palveluiksi, joita voidaan sitten käyttää rakennuspalikoina monimutkaisempien järjestelmien rakentamiseen. Jokaisella mikropalvelulla voi myös olla oma tietokantansa. Usean mikropalvelun järjestelmässä yksi mikropalvelu voi edustaa esimerkiksi tuotekatalogia, toinen ostoskoria ja kolmas maksupalvelua, jolloin niistä voidaan muodostaa yhdessä kokonainen verkkokauppajärjestelmä. Kuvassa 1 on havainnollistettu eräitä yksinkertaisia monoliittisena ja mikropalveluina toteutetun verkkokauppajärjestelmän arkkitehtuurimalleja. (Newman 2021, 3–14; Davis 2021, 6–8.)



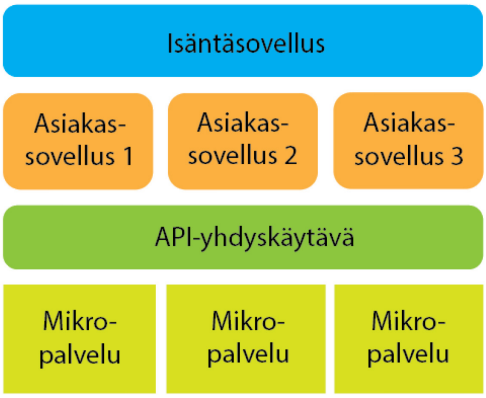
Kuva 1. Esimerkkejä verkkokauppajärjestelmän rakenteesta eräillä monoliittisilla arkkitehtuurimalleilla sekä mikropalveluarkkitehtuurilla

Mikropalvelut keskustelevat keskenään ainoastaan ohjelmointirajapintojen kautta, jolloin niiden sisäinen toiminta on piilossa muilta sitä käyttäviltä mikropalveluilta tai ohjelmistoilta. Tämä mahdollistaa yksittäisen mikropalvelun muuttamisen siten, että se ei vaikuta muihin järjestelmän osiin, jolloin yksittäisiä mikropalveluita voidaan kehittää, julkaista ja skaalata joustavasti omassa tahdissaan ja tarvittaessa jopa kokonaan erilaisilla teknologioilla. Tämän joustavuuden ansiosta mikropalveluista on tullut jatkuvasti suosiotaan kasvattava, lähes oletusarvoinen arkkitehtuurivalinta monelle organisaatiolle. Joustavuudella on kuitenkin hintansa, joista merkittävin on toteutuksen monimutkaisuus. Mikropalvelut eivät siis aina ole järkevin ratkaisu, vaan valinta on tehtävä tapauskohtaisesti. (Newman 2021, 3–14; Davis 2021, 6–8.)

2.1 Micro frontend -arkkitehtuuri

Suuresta suosiostaan huolimatta mikropalveluarkkitehtuuria on useimmiten käytetty lähinnä sovellusten palvelinohjelmistojen toteuttamiseen. Samalla kuitenkin kamppaillaan edelleen suurten ja monimutkaisten monoliittisten asiakassovellusten haasteiden parissa. Entistä enemmän on kuitenkin alettu kiinnostua mikropalveluiden konseptin tuomisesta myös asiakassovellusten puolelle. Sen inspiroimana on syntynyt micro frontend -arkkitehtuuri, jossa asiakassovellus jaetaan esimerkiksi sivujen tai ominaisuuksien mukaisiin pienempiin ja joustavampiin palvelukokonaisuuksiin, jotka toimivat yhdessä luoden käyttäjälle yhtenäiseltä näyttävän sovelluksen. (Jackson 2019; Netkow s.a.)

Micro frontend -arkkitehtuuri voidaan toteuttaa monilla eri lähestymistavoilla. Useimmiten ne noudattavat kuitenkin pääpiirteittäin samaa perusrakennetta, joka koostuu useasta pienestä asiakassovelluksesta ja niiden avulla sopivien näkymien koostamisesta vastaavasta isäntäsovelluksesta (kuva 2). Lähestymistavat poikkeavat toisistaan sen suhteen, kuinka osat integroidaan keskenään. Integraatio voidaan tehdä käyttäen esimerkiksi palvelinpuolen kokoonpanointegraatiota, rakennusaikaista integraatiota tai sovelluksen ajonaikaista integraatiota iframe-elementtien, Javascriptin tai Web-komponenttien avulla. (Jackson 2019; Mezzalira 2022, 80–113.)



Kuva 2. Esimerkki micro frontend -arkkitehtuurista, jossa asiakassovellus koostuu useista pienistä sovelluksista, jotka yhdistellään sopiviksi näkymiksi isäntäsovelluksen avulla

Micro frontend -arkkitehtuurin ominaisuudet etuineen ja haasteineen ovat suurilta osin samat kuin mikropalveluissa. Suunnitteluperiaatteet kuitenkin poikkeavat hieman toisistaan. Siinä missä mikropalveluiden lähtökohtaisena toimintaperiaatteena on, että mitään ei jaeta mikropalveluiden kesken, on micro frontend -arkkitehtuurissa kuitenkin toisinaan järkevää jakaa esimerkiksi tiettyjä uudelleenkäytettäviä komponentteja ulkoasun yhteneväisyyden varmistamiseksi ja toistuvan koodin vähentämiseksi. Uudelleenkäytettävät komponentitkin tulisi kuitenkin suunnitella siten, että ne eivät sisällä liiketoimintalogiikkaa ja olisivat mahdollisimman riippumattomia muista komponenteista. (Jackson 2019; Mezzalira 2022, 47.)

2.2 Mikropalvelut mobiilisovelluksissa

Micro frontend -arkkitehtuuri on vielä nykyisellään tunnetumpi konsepti lähinnä web-sovellusten parissa, sillä natiivien mobiilisovellusten jakelutavan vuoksi se ei sovellu yhtä hyvin niiden käyttöön. Siinä missä web-sovellukset voidaan eristää täysin itsenäisiksi ja sitä myöten kehittää ja päivittää sovelluksen osia muista osista erillisinä, edellyttää mobiilisovellusten jakelutapa niiden koodin kääntämisen yhdeksi binääriksi, jolloin kaikki sovelluksen päivitykset joudutaan lopulta ajamaan yhtenä kokonaisuutena tuotantoon. Näin ollen micro frontend -arkkitehtuurin tuomat hyödyt sovelluksen skaalautuvuuteen ja kehityksen joustavuuteen lopulta menettävät merkityksensä mobiilisovelluksissa. (Geers 2020 19–20; Netkow s.a.)

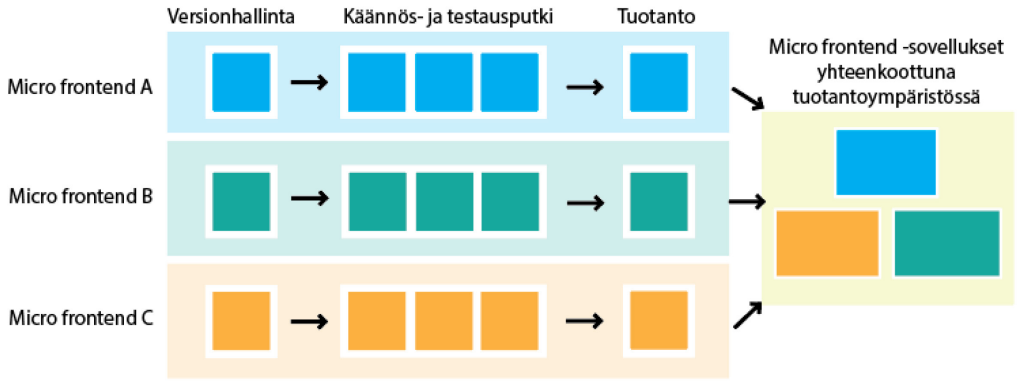
Alustaan liittyvien rajoitteiden lisäksi haasteena on sovelluksen monimutkaistuminen. Useisiin micro frontend -sovelluksiin sisältyy useita koodikantoja

sekä enemmän palveluita ja sisältöä, mikä voi olla erityisesti pienille organisaatioille liian hidas ja kallis hallittava. Jokainen tiimi voi käyttää omia ratkaisujaan oman vastuualueensa sovelluksessa, mikä johtaa herkästi koodin ja tekniikoiden toistamiseen sekä päällekkäisten verkkopyyntöjen kertymiseen sovellusten välisessä kommunikoinnissa. Lisäksi kaikkien yksittäisten sovelluksen osien päivittäminen ja julkaiseminen jakelukanavien kautta on erittäin hidasta, koska jokaisen tulisi käydä läpi samat sovelluskauppakohtaiset julkaisuprosessit. Arkkitehtuurin riskit ovat kuitenkin hallittavissa ja sen tuomien etujen on koettu voittavan haasteiden aiheuttamat kustannukset, minkä vuoksi mobiilikehitystiimit ovat kiinnostuneita arkkitehtuurin tuomisesta mobiiliympäristöihin. (Micro Frontends for Mobile 2022.)

Yhdysvaltain Chicagossa sijaitseva IT-alan konsulttiyritys Thoughtworks julkaisee muutaman kerran vuodessa Technology Radar -nimisen verkkojulkaisun ohjelmistoalan kiinnostavimmista teknologiatrendeistä. Micro frontend -arkkitehtuuri listattiin adopt-merkinnällä ensimmäisen kerran maaliskuun 2019 numerossa. Adopt-merkinnällä kehotetaan vahvasti arkkitehtuurin ottamista käyttöön sille soveltuvissa projekteissa. Vuoden 2020 maaliskuun numerossa arkkitehtuuri listattiin nousevaksi trendiksi myös mobiilisovellusten puolella trial-merkinnällä, joka kehottaa organisaatioita ottamaan arkkitehtuurin käyttöönsä laajemmissa ja monimutkaisemmissa projekteissa, jotka sietävät mahdolliset uuden ja kehittymättömän teknologian tuomat riskit. Vuoden 2021 lokakuun numerossa listausmerkintä nostettiin trendeihin uudelleen päivityksellä, jossa kerrottiin erityisesti React Nativea käyttävien tiimien pyrkivän parhaillaan kehittämään sovelluskehyksiä, joiden avulla arkkitehtuuria voitaisiin hyödyntää mobiilisovellusprojekteissa. (Micro frontends 2016; Micro frontends for mobile 2020.)

Mobiilialustojen rajoitteiden vuoksi micro frontend -arkkitehtuurin toteuttamisen mahdollistavien työkalujen kehittämisessä on keskitytty lähestymistapaan, jossa micro frontend -sovelluksiin sisällytetään erillinen isäntäsovellus, joka vastaa sovellusten kokoamisesta yhdeksi yhtenäiseksi kokonaisuudeksi. Näin ollen isäntäsovellus katsotaan alustojen jakelutapavaatimusten mukaiseksi yksittäiseksi sovellukseksi, joka voidaan julkaista sovelluskauppoihin muiden micro frontend -sovellusten toimiessa sovelluksen sisäisinä, itsenäisesti kehitettävinä kirjastoina (kuva 3). Isäntäsovellukseen ei suoraan sijoiteta ollenkaan

liiketoimintalogiikkaa, vaan kukin sovelluskehitystiimi vastaa oman micro frontend -sovelluksensa liittämisestä isäntäsovellukseen. Isäntäsovellus pysyy kevyenä ja selkeänä, mikä pienentää ristiriitaisuuksien riskiä kehitettäessä micro frontend -sovelluksia itsenäisinä ja toisistaan riippumattomina osina. (Micro Frontends for Mobile 2022.)



Kuva 3. Micro frontend -arkkitehtuurissa itsenäisesti toimivat palvelut kommunikoivat keskenään luoden käyttäjälle yhtenäiseltä näyttävän sovelluksen (mukaillen Jackson 2019)

Alustaan liittyvistä haasteista sekä toteuttamiseen tarvittavien työkalujen ja menetelmien kehittämisen aikaisesta kehitysasteesta huolimatta riskit ovat hallittavissa, ja saavutetut hyödyt voivat olla riskien arvoisia erityisesti isommissa organisaatioissa. Avainasemassa toteutuksen onnistumisessa ovat huolellinen suunnittelu ja selkeät palveluiden väliset rajaukset, joilla voidaan pienentää arkkitehtuurin riskejä. Micro frontend -sovellusten vastuualueiden selkeä rajaaminen pienentää tarvetta sovellusten väliseen kommunikointiin ja näin ollen pienentää koko sovelluskokonaisuuden monimutkaisuutta. (Micro Frontends for Mobile 2022.)

3 TYÖSSÄ KÄYTETTÄVÄT TEKNOLOGIAT

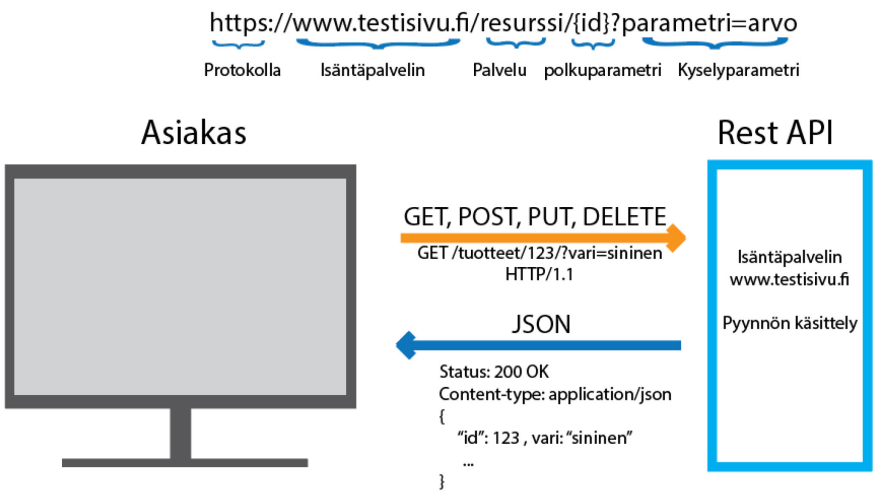
3.1 REST API

API (Application Programming Interface) on ohjelmointirajapinta, jota käytetään laitteiden tai laitteissa toimivien sovellusten väliseen kommunikointiin. Sovellukset voivat tehdä pyyntöjä toisilleen rajapintojen kautta määrittelemällä tarvitsemansa datan ja sen formaatin, jolloin pyynnön vastaanottava sovellus hakee tiedot, muuntaa ne pyynnön osoittamaan muotoon ja palauttaa tiedot vastauksena pyynnön tehneelle sovellukselle. (Sharma 2022, 239–240.)

API-rajapintoja on olemassa useampaa eri tyyppiä, mutta erityisesti modernien web API -rajapintojen yhteydessä tunnetumpia ovat REST- ja GraphQL-pohjaiset rajapinnat. Toimeksiannossa sovelluksen integrointiin käytettävän olemassa olevan toiminnanohjausjärjestelmän rajapinta on REST-tyypin rajapinta, joten tässä työssä perehdytään tarkemmin vain REST API -rajapintoihin.

REST (Representational State Transfer) on alustasta ja ohjelmointikielestä riippumaton arkkitehtuuri. Tarkemmin sanottuna se on enemmänkin sarja hyviä käytäntöjä ja toimintatapoja, jotka mahdollistavat rajapintojen kehittämisen tietyllä selkeällä esitystavalla. Vaikka REST toimii useiden protokollien kautta, sitä käytetään yleensä kuitenkin vain HTTP-protokollan yli. REST-arkkitehtuurissa käytetään yleensä yksinkertaisia datan esitysmuotoja, kuten JSON-formaattia, joita voidaan helposti lukea ja käyttää kaikenlaisissa sovelluksissa, mutta se tukee myös muita formaatteja. (Zammetti 2020, 170.)

REST API -rajapinnat koostuvat kolmesta komponentista: resursseista, toiminnoista ja esitysmuodoista. REST käyttää yksinkertaisesti URL (Uniform Resource Locator) -osoitetta osoittamaan haluttuun resurssiin ja suorittamaan sille halutut CRUD-toiminnot eli tietojen luominen (Create), lukeminen (Read), päivittäminen (Update) ja poistaminen (Delete). Esimerkki REST API:n käyttämästä URL-formaatista ja pyyntöjen käsittelystä rajapinnan ja asiakassovelluksen välisessä kommunikoinnissa esitetään kuvassa 4. (Chellammal ym. 2020, 120–121.)



Kuva 4. Esimerkki REST API -rajapinnan ja asiakassovelluksen välisestä kommunikoinnista

Parametrejä on neljää eri tyyppiä: otsikko-, polku- ja kyselyparametrit sekä pyynnön runkoparametrit. Otsikkoparametrit lähetetään pyynnön otsikossa. Kyselyparametrit ja polkuparametrit määritetään pyynnön polussa. Kyselyparametrit annetaan yleensä avain-arvo-pareina muodossa ”?muuttuja=arvo” eli esimerkiksi ”?vari=sininen” ja polkuparametrit ennen kysymysmerkkiä olevassa osassa, usein aaltosuluilla määritettynä (Esimerkiksi: {id}). Pyynnön runkoparametrit lähetetään pyynnön rungossa useimmiten JSON-formaatissa. (Chellammal ym. 2020, 120–121.)

3.2 MongoDB Realm

Vuonna 2011 kaksi entistä Nokian mobiili-insinööriä, Alexander Stigsen ja Bjarne Christiansen, kehittivät uuden tavan saada data käyttämään vähemmän puhelimen muistia, jotta sovellukset toimisivat paremmin myös resurssirajoitteisemmissa ympäristöissä, kuten mobiili- ja IoT-laitteissa. Tämän seurauksena syntyi avoimeen lähdekoodiin perustuva sulautettu, moderneja objektiperiaatteita käyttävä tietokanta nimeltään Realm. Se käyttää ns. ”nollakopioarkkitehtuuria” (Zero-copy architecture), joka pienentää muistin tarvetta ja takaa tietojen säilymisen myös virtakatkojen aikana. MongoDB osti Realmin vuonna 2019 osaksi omaa palvelutarjontaansa. (Alekseev 2022; Bort 2015; Horowitz 2019.)

Realm on suunniteltu erityisesti mobiili- ja verkkosovellusten kehityksen yksinkertaistamista silmällä pitäen. Se tarjoaa avoimeen lähdekoodiin perustuvia ohjelmistokehityspaketteja useille ohjelmointikielille, -kehyksille ja alustoille, ja sen natiivisti toimiva oliopohjainen tietomalli mahdollistaa työskentelyn ilman ORM-työkaluja (Object Relational Mapping). Tietokanta upotetaan suoraan sovellukseen, minkä vuoksi tietokantaa ei tarvitse ottaa erikseen käyttöön, jolloin prosessi yksinkertaistuu. Realmin live-objektit mahdollistavat datan muutostapahtumien tilaamisen, jolloin käyttöliittymä pysyy aina ajan tasalla ilman erillisen kyselyn tekemistä tietokantaan. Tämä vähentää tarvittavan koodin määrää, kun erillisiä tarkistusvaiheita muuttuneen datan varalta ei tarvita. Nämä ominaisuudet yksinkertaistavat ja nopeuttavat sovellusten kehittämistä. (Alekseev 2022, Bort 2015.)

Realm on integroitavissa MongoDB Atlas -palvelun kanssa. Atlas Device Sync -toiminnon avulla Realm-tietokantaa käyttävien sovellusten data voidaan synkronoida laitteiden välillä, jolloin tietokannan tiedot pysyvät aina ajan tasalla. Realm toimii kuitenkin aina ensisijaisesti kirjoittamalla ja lukemalla data suoraan laitteessa sijaitsevasta paikallisesta tietokannasta ja synkronointi tapahtuu taustalla, minkä vuoksi se soveltuu erityisen hyvin sovelluksiin, joiden tulee pystyä toimimaan myös offline-tilassa. Lisäksi Realm sisältää sisäänrakennetun käyttäjienhallinnan, jolloin tietokannan käyttöoikeudet ja käyttäjien todennus voidaan suoraan liittää lähes mihin tahansa ulkoiseen todennuspalveluntarjoajaan. (MongoDB docs s.a.)

3.3 React

React on Facebookin vuonna 2013 julkaisema modernien käyttöliittymien toteuttamiseen tarkoitettu kirjasto Javascriptille. Reactin ekosysteemi mahdollistaa sovellusten käyttöliittymien muodostamisen SPA-sovelluksina (Single-Page Appliation) eli ns. yksisivuisina sovelluksina. Se käyttää itsenäisiä, uudelleenkäytettäviä käyttöliittymäkomponentteja, jotka on helppo muuttaa tai vaihtaa toisiin paikkoihin tai jopa projekteihin sellaisenaan ja muodostaa niiden avulla uudenlaisia kokonaisuuksia. (Wieruch 2018, 2.)

React noudattaa ehkä yllättäenkin hyvin minimaalisia periaatteita. Siinä ei yritetä vaikuttaa toteutustapoihin tai rakenteisiin, eikä sen mukana tule valmiita palikoita, vaan kehittäjä voi itse muodostaa omiin käyttötarkoituksiinsa sopivat monikäyttöiset rakennuspalikat. React tarjoaa ainoastaan neljä ominaisuutta: komponentit, niiden ominaisuudet eli propsit, tilan sekä tyylin. React käyttää virtuaalista dokumenttioliomallia eli VDOM:ia (Virtual Document Object Model) näkymän muutoksien tallettamiseen. Tällöin varsinaiseen HTML-elementtien jäsentämiseen käytetyn DOM-puun muokkaaminen tapahtuu yhdellä kertaa virtuaalisen DOM:in kautta, mikä useimmiten parantaa käyttöliittymän suorituskykyä. Lisäksi React käyttää JSX-merkintäkieltä (Javascript XML), joka yhdistää renderöintilogiikan käyttöliittymälogiikan kanssa, jolloin tilan ja tapahtumien käsittely on helpompaa. (Zammetti 2020, 44–45.)

Komponentit ovat Javascript-funktioita, joihin on sekoitettu HTML-merkintäkieltä ja CSS-tyylejä. Käyttöliittymä voidaan rakentaa näiden komponenttien

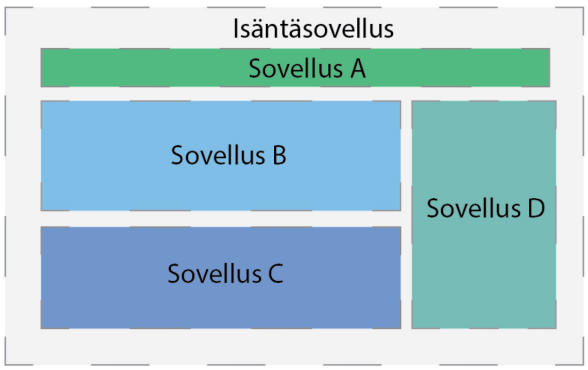
avulla kokoamalla, järjestelemällä ja sijoittelemalla niitä sisäkkäin halutun lopputuloksen aikaansaamiseksi. Komponentit kommunikoivat keskenään välittämällä tietoa toisilleen propsien avulla. Propsit ovat muuttumattomia ja niiden avulla voidaan välittää ennalta määrättyjä arvoja, kuten src- tai className-arvot, mutta propseina voidaan lisäksi välittää minkälaisia Javascript-arvoja tahansa, kuten esimerkiksi objekteja, taulukoita tai funktioita. (React Docs s.a.)

Sovelluksen koon kasvaessa sovelluksen tilanhallinta ja datan liikkuminen komponenttien välillä monimutkaistuvat, mikä vaikeuttaa koodin ylläpidettävyyttä ja kasvattaa sen virhealttiutta. Sovelluksen tilanhallinta tulisi organisoida siten, että päivityslogiikka on muokattavissa loogisesti yhdestä paikasta, ja tila voidaan jakaa toisistaan kaukana olevien komponenttien välillä aiheuttamatta ylimääräistä, toistuvaa koodia. Reactissa käyttöliittymää ei muokata suoraan koodista, vaan sen avulla kuvaillaan komponentin halutut visuaaliset tilat ja sitten aktivoidaan tilamuutokset käyttäjän syötteen perusteella. Tilan hallinta voidaan hoitaa nostamalla tila yhteiseen pääkomponenttiin ja välittämällä se propseilla alikomponenteille. Reactin reducer-metodilla on mahdollista yhdistää sovelluksen tilanhallinta yhteen paikkaan, ja jakaa siihen tallennettu tila kaikille halutuille komponenteille Reactin Context-objektin avulla. (React Docs s.a.)

3.4 Webpack 5 ja Module federation

Webpack on moderneille Javascript-sovelluksille kehitetty paketointityökalu, jonka avulla sovelluksen tarvitsemat moduulit voidaan koota yhdeksi tai useammaksi paketiksi. Webpack käyttää sille määritettyä sovelluksen aloituspistettä lähtökohtana sisäisen riippuvuuskaavion rakentamiseen, jossa se määrittelee kaikki sovelluksessa tarvittavat moduulit selvittämällä aloituspisteeseen suorasti tai epäsuorasti liittyvät riippuvuudet, ja muodostaa niistä tarvittavat paketit haluttuun sijaintiin. Webpack on erittäin laajasti konfiguroitavissa erilaisiin käyttötarpeisiin, ja sen avulla on mahdollista optimoida sovellukset suorituskykyisemmiksi ja yhteensopivammiksi eri selainversioiden kanssa. Webpack voidaan myös laajentaa tukemaan muitakin resursseja kuten kuvia, fontteja tai tyylitiedostoja. (Webpack s.a.)

Webpack toi version 5 julkaisun yhteydessä uuden Module Federation -laajennuksen, jonka tarkoitus on mahdollistaa moduulien yhdistäminen toisiinsa synkronisesti tai epäsynkronisesti sovelluksen ajoaikana. Module Federation -sovellus koostuu kahdesta osasta: isäntä- ja etäsovelluksista. Etäsovelluksilla tarkoitetaan micro frontend -sovelluksia tai kirjastoja, jotka paljastavat osia sovelluksesta ladattavaksi dynaamisesti isäntäsovellukseen. Isäntäsovellus toimii siis eräänlaisena säiliönä, johon voidaan koota näkymiä muista ladattavista kirjastoista tai etäsovelluksista halutussa muodossa (kuva 5). (Mezzalira 2022, 81.)



Kuva 5. Module Federation -menetelmässä asiakassovellus kootaan eri sovellusten välillä jaetuista komponenteista ja kirjastoista yhtenäiseltä näyttäväksi sovellukseksi

Module Federation -lähestymistavassa mikrosovelluksista SPA-sovelluksen sisältämät mikrosovellukset kootaan omiksi paketeikseen, jolloin niitä voidaan mikropalveluarkkitehtuurille tyypilliseen tapaan muokata ja julkaista itsenäisinä osina. Esimerkiksi kolmesta mikrosovelluksesta koostuvan sovelluksen yhteen osaan voidaan tehdä muutokset ja ajaa sen jälkeen tuotantoon ilman, että muita sovelluksen osia tarvitsee pakata ja julkaista uudelleen. Tätä voidaan hyödyntää myös erillisten jaettujen komponenttikirjastojen käytössä ja ylläpidossa. Sen sijaan, että komponenttikirjasto asennettaisiin riippuvuutena suoraan sovellukseen tai sen eri osiin, voidaan sitä ylläpitää itsenäisesti, jolloin kaikki sitä hyödyntävillä sovelluksilla on käytettävissään komponenttien uusin versio. (Webpack s.a.)

3.5 React Native

React Native on Facebookin kehittämä React-kirjastoon perustuva mobiilisovellusten kehittämiseen tarkoitettu sovelluskehys, jonka avulla voi kehittää

monialustaisia sovelluksia. Sen avulla voi kehittää mobiilisovelluksia useimmille suurille alustoille kuten Android, iOS, Universal Windows Platform (UWP), mutta se soveltuu myös web-sovellusten kehittämiseen. React Nativen ja Reactin eroavaisuus on niiden tavassa hahmontaa koodia. React käyttää komponenttien renderöintiin VDOM:ia ja React Native käyttää Native API -rajapintaa. React Native käyttää lisäksi HTML- ja CSS-ohjelmointikieliä eri tavoin mobiilissa verrattuna web-sovelluksiin. (Sharma 2022, 102–103.)

React Nativen vahvuuksiin kuuluvat sen nopea kehitysaika ja alhaisemmat kehityskustannukset, sillä yhdellä koodipohjalla on mahdollista toteuttaa useampi eri alustalle suunnattu sovellus, ja kehittämiseen tarvittava tiimikin voi olla natiivisovellusten kehittämiseen verrattuna pienempi. React mahdollistaa yksinkertaisen, responsiivisen ja nopean käyttöliittymän toteuttamisen sekä paremman käyttökokemuksen. Suorituskyvyltään React Native sovellus ei pärjää täysin natiivisovelluksille, mutta ero niiden välillä on hyvin pieni. (Budzinski 2022.)

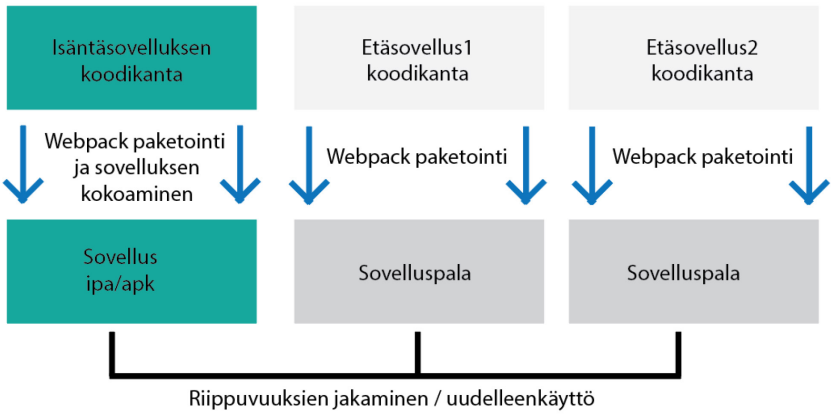
React Nativella on myös heikkoutensa. Se on vielä verrattain nuori teknologia, minkä vuoksi kustomoitujen moduulien saatavuus on huono ja olemassa olevatkin kaipaavat vielä kehittymistä. Tämän vuoksi se kärsii myös yhteensopivuusongelmista eri riippuvuuksien välillä ja edellyttää toisinaan silti natiivin koodin kirjoittamista. Lisäksi virheiden löytäminen on hankalaa puutteellisten työkalujen takia. Skaalautuvuuden on myös koettu olevan erityisesti aiemmin hieman ongelmallinen, joskin React Native on kehittynyt vauhdikkaasti ja pystyy mahdollisesti vastaamaan tähän ongelmaan nykyään hieman paremmin. (Budzinski 2022.)

3.6 Re.Pack-paketointityökalu

Re.Pack on avoimen lähdekoodin työkalu, joka mahdollistaa Webpackin käyttämisen React Native -sovelluksissa. Re.Pack tukee Webpack 5:n koko ekosysteemiä, mikä mahdollistaa hyvin edistyneiden ominaisuuksien hyödyntämisen myös mobiilisovellusympäristössä. Tähän sisältyvät myös Webpack 5:n Module Federation ja koodin jakamisen tuki. Lisäksi Re.Pack tukee mm.

Flipper-alustaa, joka helpottaa virheenkorjausta ja vian etsintää erilaisten lokien ja raporttien avulla sekä Hermes-moottoria, jonka avulla voidaan kääntää ja ajaa kehitys- ja tuotantoympäristön paketteja. (Trysła 2021.)

React Native -projekteissa oletusarvoisesti kehitysalustana ja paketointityökaluna käytetään Metro-nimistä työkalua. Re.Pack eroaa Metrosta siinä, että Metro on valmis räätälöity ratkaisu, mikä helpottaa ja nopeuttaa tiettyjä työvaiheita, mutta rajoittaa yksilöllisten ratkaisujen käyttöä. Re.Packin periaatteena on antaa kehittäjille mahdollisimman paljon valtaa vaikuttaa asetuksiin ja määrityksiin, jotta tekniikkaa on mahdollista soveltaa kunkin projektin käyttötarkoitukseen sopivimmalla tavalla. Laaja muokattavuus tosin monimutkaistaa ja hidastaa vastaavasti tiettyjä työvaiheita ja edellyttää React Native -sovellusten kehittäjiltä ymmärrystä myös Webpackin konfiguroinnista. Metro paketoi oletusarvoisesti kaiken koodin riippuvuuksineen aina yhdeksi Javascript-tiedostoksi, kun taas Re.Packin yksi vahvuuksista on mahdollisuus jakaa micro frontend -arkkitehtuurille ominaiseen tapaan sovellus erillisiksi paketeiksi (kuva 6). (About Re.Pack 2021; Metro 2018.)



Kuva 6. Mobiilisovelluksen rakenne Webpackin Module Federation -lähestymistavalla (mukaillen Trysła 2021)

Re.Pack tarjoaa kolme eri lähestymistapaa koodin jakamisen toteuttamiseksi: epäsynkroniset osat, skriptit sekä Webpack 5:n Module Federation. Epäsynkroniset osat ovat dynaamisesti ladattavia sovelluksen osia, jotka sijoitetaan sisällönjakeluverkkoon tai erilliselle palvelimelle, josta isäntäsovellus pääsee niihin käsiksi. Skriptit antavat paljon enemmän joustavuutta toteutukselle, mutta ne vaativat erittäin syvällistä ymmärrystä Webpackin lisäksi React Nativen toi-

minnasta. Niiden avulla voidaan ajaa mitä tahansa koodia dynaamisesti. Module Federation -lähestymistavan avulla voidaan toteuttaa sovellus itsenäisinä osina samasta tai erillisistä koodikannoista. (About Re.Pack 2021.)

4 SUUNNITTELU

4.1 Toimeksiannon lähtökohdat

Riversoft Oy on Seinäjoella vuonna 2021 perustettu ohjelmistoalan yritys, joka valmistaa siivous- ja kiinteistönhuoltoalan yrityksille suunnattua toiminnanohjausjärjestelmää SaaS (Software as a Service) -palveluna. Järjestelmän päätoiminnallisuuksiin kehityksen alkuvaiheessa kuuluvat varastojen ja kaluston hallinta- ja seurantaominaisuudet sekä niihin liittyvät tukitoiminnot, kuten asiakkaiden ja työkohteiden hallinta. Ohjelmistoa laajennetaan tulevaisuudessa lisämoduuleilla, jolloin kokonaisuudesta muodostuu kokonaisvaltainen toiminnanohjausjärjestelmä.

Järjestelmä on kokonaisuudessaan hyvin monimutkainen ja sen hyödyntäminen mahdollisimman tehokkaasti edellyttää yrityksen henkilöstön sitoutumista monessa eri portaassa ottamaan järjestelmän käytön osaksi päivittäisiä työrutiineja. Järjestelmän kohderyhmä koostuu suurilta osin henkilöistä, joille erilaisten laitteiden ja ohjelmistojen käyttäminen saattaa olla haasteellista, tai työtehtävät saattavat edellyttää toimimista työkohteissa, joissa ei ole mahdollisuutta käyttää tietokonetta tai välttämättä edes toimivaa nettiyhteyttä. Nämä tuovat haasteita järjestelmän käyttöönotettavuudelle ja käytettävyydelle.

Järjestelmän käyttäjistä suuren osan odotetaan käyttävän järjestelmää mobiililaitteilla. Vaikka web-sovelluksen voisi toteuttaa myös PWA (Progressive Web App) -sovelluksena, jolla voidaan jäljitellä natiivin mobiilisovelluksen käyttäjäkokemusta, on erillisessä mobiilisovelluksessa kuitenkin erityisesti järjestelmän myöhemmän vaiheen ominaisuuksien ja sen laajuuden kannalta tiettyjä etuja. Näitä ovat esimerkiksi suorituskyvyn parempi optimointi mobiililaitteille ja mahdollisuus siirtää osa järjestelmän resurssien käytöstä käyttäjien laitteille. Lisäksi työnkulkua järjestelmässä voitaisiin kehittää hyödyntämällä mobiililaitteiden sisäänrakennettuja ominaisuuksia kuten kameraa ja sijaintipalveluita. Tämän vuoksi web-sovelluksen rinnalle päätettiin toteuttaa mobiilisovellus, mistä syntyi myös tämän opinnäytetyön toimeksianto.

Micro frontend -arkkitehtuurin käyttäminen sovelluksen toteutuksessa tuli esille suunniteltaessa toimeksiannon sisältöä ja siinä käytettäviä teknologioita. Erityisesti esille nousi skaalattavuuden, ylläpidettävyyden ja selkeyden ylläpitäminen sovelluksessa, johon on tulevaisuudessa tarve lisätä uusia toimintoja ja jopa kokonaisia moduuleita. Tällöin toimeksiantoon päätettiin liittää toteutuksen yhteydessä tutkittavaksi kohteeksi myös mahdolliset työkalut ja keinot mikropalveluarkkitehtuurin soveltamiseksi mobiilisovelluksissa.

4.2 Toimeksiannon tavoitteet ja rajaus

Toimeksiantajan edellytys sovellukselle oli, että se toteutetaan iOS- ja Android-alustoille, sen on pystyttävä kommunikoimaan ja synkronoimaan dataa eri laitteiden välillä ja sitä on tarpeen mukaan pystyttävä käyttämään myös offline-tilassa. Eri käyttäjätasoilla on erilaiset käyttöoikeudet ja näkyvyysasetukset järjestelmässä, joten myös sovelluksessa on voitava erotella eri käyttäjätasoihin liittyvät näkymät ja toiminnot. Toiminnanohjausjärjestelmän jatkuva laajeneminen ja tarpeiden muuttuminen edellyttävät lisäksi sovelluksen arkkitehtuurin ja käyttöliittymän toteutustavan suunnittelua siten, että se mahdollistaa uusien toimintojen lisäämisen ja erottelun portaittain mahdollisimman sujuvasti.

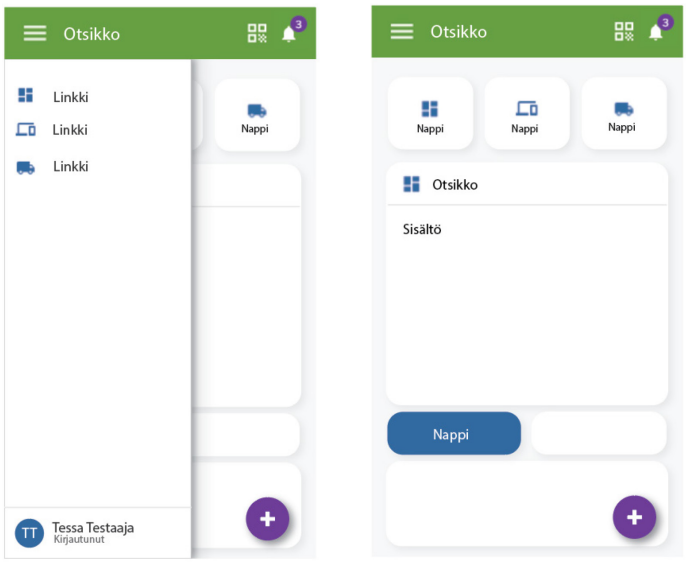
Sovellukseen toteutetaan tässä vaiheessa vain järjestelmän oleellisimmat, päivittäin käytettävät toiminnot, jotka käsittävät tavaroiden ja kaluston hallinnan ja seurannan. Hallintatoimet sisältävät tavaroiden ja kaluston CRUD-toiminnot. Tavaroiden seurantaan käytetään QR-koodin skannausta mobiililaitteen kameran avulla ja sijaintipalveluita, joilla tavaran sijaintia voidaan seurata ja muuttaa tarpeen mukaan. Kaluston seurantaan tullaan myöhemmässä vaiheessa ottamaan käyttöön ajonseurantalaitteisto, jonka avulla sijainnit kirjataan automaattisesti, mutta tässä vaiheessa kaluston sijainnit kirjataan vain manuaalisesti.

Työ rajataan siten, että siinä ei käsitellä tarkemmin sovelluksen taustalla toimivan web-sovelluksen rakennetta paitsi tarpeellisten rajapintakutsujen toteuttamiseen tarvittavilta osin. Mobiilisovellukseen toteutettaviin toimintoihin sisältyy

varsinaisessa toiminnanohjausjärjestelmässä lisäksi muita toimintoja ja riippuvuuksia, mutta mobiilisovelluksessa ne jätetään huomiotta osin käyttöliittymän yksinkertaistamiseksi ja osin laajuuden rajaamiseksi. Sovelluksesta ei tästä syystä tehdä vielä julkaistavaa versiota, joten julkaisuun liittyviä toimia ei myöskään käsitellä. Samoin tietoturvaan liittyvä syvempi tarkastelu jätetään tämän työn ulkopuolelle.

4.3 Sovelluksen ulkoasu

Sovelluksen ulkoasu toteutetaan siten, että lopputulos noudattaa olemassa olevan web-sovelluksen tyyliä. Näin käyttökokemus pysyisi mahdollisimman yhtenäisenä ja helposti omaksuttavana. Web-sovellus on saanut hyvää palautetta sen selkeästä käyttöliittymästä, joten myös mobiilisovelluksessa pyritään samaan. Mobiilisovelluksen käyttöliittymässä pyritään kuitenkin välttämään kaikkea ylimääräistä kognitiivista kuormaa myös sovelluksen jatkokehitys huomioiden. Sovelluksen ulkoasun toteuttamiseen käytetään Googlen kehittämää Material design -muotokieltä, jota on käytetty myös websovelluksessa. Sovelluksen alustavaa tyyliä ja ulkoasua sekä navigointivalikkoa hahmoteltiin Adobe XD -sovelluksen avulla (kuva 7).



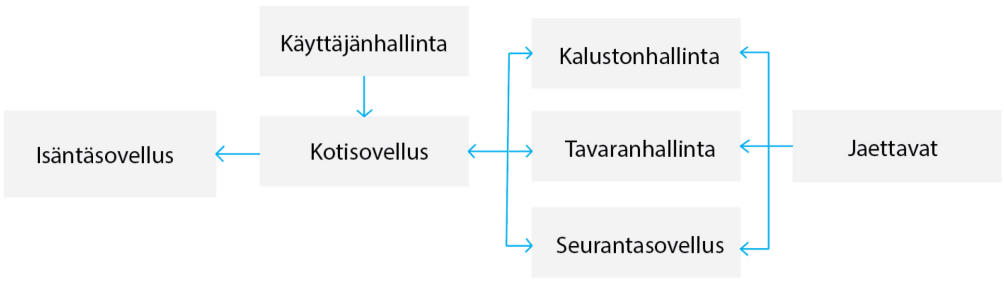
Kuva 7. Sovelluksen alustava ulkoasu ja tyyli

Sovelluksessa on käyttäjän näkökulmasta neljä päänäkymää, jotka ovat kotinäkymä, tavaralistaus, kalustolistaus sekä seurantanäkymä. Kotivalikko sisältää tämän työn puitteissa lähinnä vain navigointiin tarkoitetun valikon. Jatkossa kotivalikon on tarkoitus toimia käyttäjän henkilökohtaisena näkymänä,

joka voi sisältää tarpeellista statistiikkaa, ilmoituksia tai muuta oleellista informaatiota. Kotivalikosta voidaan siirtyä suoraan eri toimintoihin. Tavara- ja kalustonäkymät sisältävät listausnäkymän, jossa voi selata ja hakea tietoja, yksittäisen tavaran tietonäkymän sekä uuden tavaran luontinäkymän. Sijaintinäkymässä on tarkoitus myöhemmin näyttää karttapalvelun avulla eri tavaroiden sijainnit, mutta tässä vaiheessa sijainnit näytetään ainoastaan osoitteina. Vaikka sovelluksen kotivalikko tämän työn puitteissa toimiikin navigointitarkoituksissa, toteutetaan silti lisäksi yhteneväisyyden varmistamiseksi web-sovelluksen tyyliä noudattavat yläpalkki sekä sivunavigaatiopalkki, joka avataan ja suljetaan yläpalkissa sijaitsevasta napista.

4.4 Sovelluksen arkkitehtuuri ja rakenne

Sovellus toteutetaan jakamalla se micro frontend -arkkitehtuurin periaatteiden mukaisesti useampiin pienempiin sovelluksiin, joiden paljastamat komponentit kootaan isäntäsovelluksessa yhtenäiseksi kokonaisuudeksi. Etäsovellusten jako voidaan määrittää usealla eri tavalla. Tässä työssä jaon suunnittelun periaatteena päätettiin käyttää liiketoiminta-alueisiin ja käyttäjärooleihin perustuvaa jakoa, jonka seurauksena sovellus päätettiin jakaa isäntäsovelluksen lisäksi kuuteen etäsovellukseen. Kuvassa 8 esitetään micro frontend -sovellusten välistä rakennetta ja kommunikointia. Kuvan nuolet osoittavat tiedonkulun suuntaa siten, että nuolen alkupäässä olevasta sovelluksesta paljastetaan tarvittavat komponentit ja näkymät, jotka otetaan käyttöön nuolen osoittamassa sovelluksessa.



Kuva 8. Sovellusten väliset suhteet komponenttien ja näkymien paljastaminen

Vaikka Webpackin Module Federation -lähestymistapa ei edellytä varsinaisesti erillisen isäntäsovelluksen käyttöä, ovat Re.Packin avulla toteutetut etäsovellukset mobiiliympäristöjen luonteen vuoksi riippuvaisia isäntäsovelluksen olemassaolosta. Koska isäntäsovellus on välttämätön, edellyttävät kaikki suoraan

isäntäsovellukseen toteutetut muutokset koko sovelluskokonaisuuden uudelleenpaketointia ja -kääntämistä. Tämän vuoksi isäntäsovellukseen ei sijoiteta suoraan mitään erillisiä toimintoja, vaan tarvittavat komponentit jaetaan sille etäsovelluksilta. Tällaisia voivat olla esimerkiksi erilaiset tarjoajamallit (provider pattern), joiden avulla jaetaan koko sovelluskokonaisuudelle yhteisiä ominaisuuksia kuten globaalit tilat tai teemat. Isäntäsovellus toimii siis koontisovelluksena, johon jaetaan tarpeen mukaan muilta etäsovelluksilta tarvittavia komponentteja.

Tavaroiden ja kaluston hallintatoiminnot jaetaan omiksi mikrosovelluksikseen, jotta ne voidaan tarvittaessa myöhemmässä vaiheessa erottaa toisistaan kokonaan eri sovelluksiksi. Tällöin niiden käyttäjien, joilla ei ole oikeutta käyttää tiettyjä toimintoja, ei tarvitse myöskään tarpeettomasti ladata kyseisiä osia sovelluksesta ollenkaan ja rajatummat käyttöoikeudet sisältävän käyttäjän osuus sovelluksesta kevenee ja yksinkertaistuu. Vaikka seuranta liittyy hyvin vahvasti tavaroiden ja kaluston hallintaan, toteutetaan se kuitenkin erillisenä etäsovelluksena, sillä toimintojen toteuttamiseen tarvitaan hyvin erilaisia riippuvuuksia. Kartta-, ajonseuranta- ja sijaintiominaisuuksia voidaan tällöin kehittää kokonaan muista osista erillään, vähentäen erilaisten riippuvuuksien välisiä konflikteja ja turhaa resurssien käyttöä.

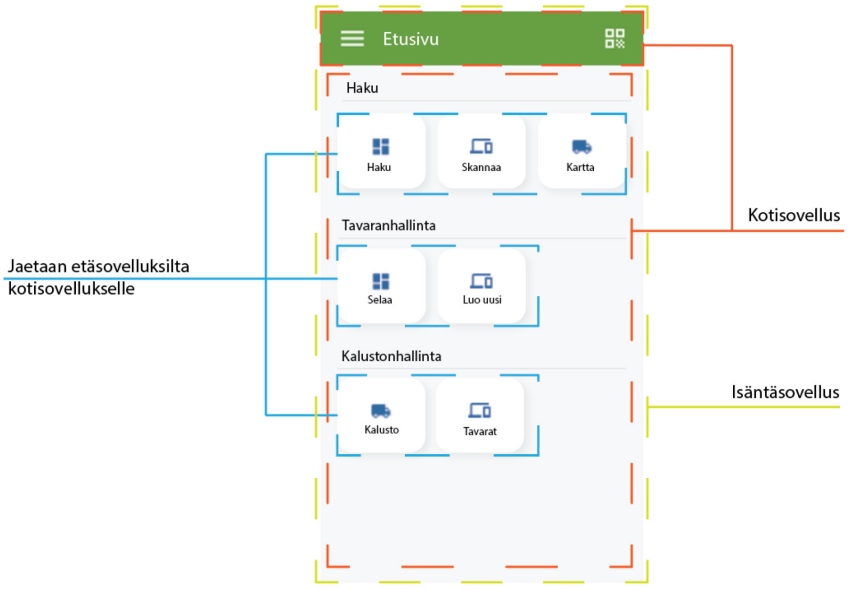
Vaikka opinnäytetyö tehdään lähinnä yksilötyönä, on suunnittelussa pyritty silti huomioimaan jatkokehitystä siten, että sovelluksen kehittäminen useamman kehitystiimin voimin olisi mahdollisimman sujuvaa ja vastuualueet selkeitä. Osin tämän vuoksi sovellukseen kirjautuminen ja siihen liittyvät istunnot ja käyttöoikeuksien hallinta toteutetaan myös erillisenä etäsovelluksena, josta toiminnot voidaan jakaa muihin mikrosovelluksiin tarpeen mukaan. Tällöin yksi tiimi voi keskittyä kehittämään ainoastaan käyttäjähallintaa eikä koko sovellusta tai sen osaa tarvitse kääntää ja paketoida aina uudelleen, jos jotain toimintoja muutetaan. Käyttäjillä ei toistaiseksi ole mahdollisuutta rekisteröityä itse sovellukseen, vaan käyttäjätilit luodaan aina organisaation pääkäyttäjän toimesta erikseen, joten erillistä rekisteröitymistä ei toteuteta.

Sovellukseen toteutetaan lisäksi vielä erillinen komponenttikirjastona toimiva sovellus, jossa on tarkoitus säilyttää kaikki eri mikrosovellusten välillä toistuvat

komponentit, joiden täytyy pysyä samanlaisina koko sovelluksen läpi. Tällä jaolla on lähtökohtaisesti tarkoituksena saavuttaa yhteneväisyyden säilyttämisen lisäksi mahdollisuus julkaista ja muuttaa komponentteja tyylillisesti yhdessä sijainnissa, jolloin jokaisessa niitä käyttävässä sovelluksessa kyseinen komponentti on aina samanlainen. Komponenttikirjaston uusien komponenttien julkaiseminen voidaan myös tehdä tällöin muista sovelluksista täysin erillisenä.

4.5 Navigointi ja kirjautuminen

Kotisovellukseen sijoitetaan kaikki sovelluksessa kiinteästi saatavilla olevat toiminnot eli valikot ja navigaatiotoiminnot sekä muut siirtymiseen tarvittavat komponentit, jotka paljastetaan isäntäsovellukselle. Tähän sisältyvät myös sovelluksen alkunäkymä ja kotisovelluksen päänäkymä, johon siirrytään kirjautumisen jälkeen automaattisesti (kuva 9). Alkunäkymä toteutetaan yksinkertaisena asetteluna, jossa näytetään sovelluksen logo sekä kirjautumispainike. Varsinainen kotisovelluksen valikkonäkymä koostetaan muista etäsovelluksista jaettavista komponenteista. Kotinäkymän valikkokomponentit voidaan liittää näkymään dynaamisesti, jolloin ainoastaan tarpeelliset komponentit ladataan näkymään saatavilla olevien palveluiden perusteella. Tällöin käytössä olevien palveluiden lista päivittyy näkymään automaattisesti ilman uudelleenpaketointia, mikäli palveluita esimerkiksi muutetaan tai poistetaan myöhemmin käytöstä.



Kuva 9. Luonnos kotisovellukseen sijoitettavien elementtien rakenteesta ja tyylistä

Sovelluksen käyttäjien todentamiseen, käyttöoikeuksiin ja muuhun käyttäjien hallintaan liittyvät toteutukset sijoitetaan erilliseen etäsovellukseen. Websovellus käyttää identiteetin- ja pääsynhallintajärjestelmänä Keycloak-järjestelmää. Käyttäjänhallintasovellus kommunikoi Keycloakin kanssa REST API -rajapinnan kautta, jolloin voimassa olevat käyttäjien tiedot ja käyttöoikeudet voidaan varmentaa sen avulla eri laitteiden välillä. Keycloakissa on mahdollisuus hyödyntää valmista offline-token-todennustapaa, jolla voidaan varmistaa kirjautuminen tarvittaessa myös tilanteissa, joissa verkkoa ei ole saatavilla.
Työn laajuuden rajaamiseksi tietoturvakriittiset toiminnot päätettiin toteuttaa kirjastojen ja työkalujen suosituksilla ja oletusmenetelmillä tietoturvan varmistamiseksi mahdollisimman yksinkertaisesti. Keycloakin virallinen suositus mobiilisovellusten tietoturvalliseen kommunikointiin Keycloakin kanssa ovat AppAuth-nimiset natiivikirjastot Androidille ja iOS:lle, joten käyttäjänhallintasovelluksessa käytetään näihin kirjastoihin perustuvaa React Nativelle soveltuvaa React Native App Auth -nimistä kirjastoa. Kirjaston noudattaman RFC8252-standardin mukaisesti sovelluksen kirjautuminen suoritetaan laitteen selaimen kautta.

Todennuksen tila ja sen apufunktiot tallennetaan Reactin Context-objektiin ja jaetaan mikrosovellusten kesken Provider-komponentin avulla isäntäsovelluksen kautta. Sensitiiviset tiedot, kuten tokenit tallennetaan Androidissa Secure Shared Preferences ja iOS-käyttöjärjestelmissä Keychain Services -palveluihin. Tämä toteutetaan React Native Keychain -kirjastolla, joka mahdollistaa pääsyn kummankin alustan palveluihin.

4.6 Tavaroiden ja kaluston hallinta

Tavaroiden ja kaluston hallintatoiminnallisuudet noudattavat hyvin läheisesti samaa kaavaa, joten myös näkymistä tehdään yhtenevät. Listausnäkymässä näytetään kaikki tietokantaan tallennetut tavarat tai kalusto perustietoineen. Listan rivit ovat aktivoitavissa painamalla halutun tavaran kohdalta, jolloin näkymän alareunaan avautuu lisätoimintovalikko, josta pääsee halutessaan suoraan tavaran lisätietonäkymään sekä tarkastelemaan tai muuttamaan sen sijaintia. Riittävillä käyttöoikeuksilla varustetulle käyttäjälle näkymässä on erillinen FAB (floating action button) -painike, josta pääsee siirtymään uuden tava-

ran luontilomakkeelle. Myöhemmässä vaiheessa samaan painikkeeseen voidaan sijoittaa myös muita korkeamman käyttöoikeustason käyttäjälle suunnattuja toimintoja.

Järjestelmään tallennettujen tietojen tulisi pysyä ajan tasalla web-sovelluksen ja mobiilisovelluksen käyttäjien välillä. Lisäksi verkkoyhteyden puuttuessa tieto pitäisi mobiilisovelluksissa pystyä tallentamaan laitteelle siten, että tapahtuneet muutokset voidaan synkronoida myöhemmin, kun verkkoyhteys on jälleen käytettävissä. Mobiilisovelluksessa tiedot tallennetaan MongoDB Realm –tietokantaan. MongoDB Realm sisältää automaattisen synkronointimahdollisuuden MongoDB Atlas -palvelun kanssa, jolloin tietoja voidaan päivittää taustalla eri laitteiden välillä. Muilta osin mobiilisovellus ja web-sovellus kommunikoivat keskenään web-sovelluksen REST API -rajapinnan kautta.

Tavaroiden ja kaluston seurantasovellukseen sisältyvät kaikki tavaroiden hakuun ja paikannukseen liittyvät toiminnot. Tämä sisältää QR-koodien lukemisen ja sen perusteella tavaran tietojen etsimisen ja uuden sijainnin tallentamisen. Koska kalustolla ei ole käytössä QR-koodeja, päästään kaluston sijaintitiedot tässä vaiheessa lisäämään manuaalisesti halutun kaluston tietonäkymän kautta.

5 TOTEUTUS

5.1 Projektin luominen

Micro frontend -arkkitehtuurissa sovellus koostuu useasta pienemmästä sovelluksesta, joten uusia projekteja eli koodipohjia tulee perustaa niin monta kuin isäntä- ja etäsovelluksia tarvitaan. Työvaiheiden nopeuttamiseksi luotiin projektien perustamista varten mallipohja, johon on valmiiksi asennettuna kaikki usein toistuvat kirjastot ja asetukset. Micro frontend -sovellukset voidaan toteuttaa versionhallinnassa kokonaan erillisinä projekteina ja erillisissä repositorioissa, mutta työnkulun sujuvoittamiseksi tässä työssä projektit sijoitettiin yhteiseen repositorioon.

Kehitysympäristö ja projektipohja asennettiin React Nativen virallisen dokumentaation iOS- ja Android-alustojen asennusohjeiden mukaisesti. Projektipohja asentamiseen käytettiin React Native CLI -komentorivityökalun avulla

asennettavaa valmista Typescript-projektipohjaa. Pohjaan asennettiin valmiiksi lisäksi Re.Pack-kirjasto tarvittavine riippuvuuksineen ja aloituskonfiguraatioineen kirjaston virallisen dokumentaation mukaisesti. Mallipohjan toimivuus varmistettiin vielä käynnistämällä sovellus iOS-emulaattorilla, jolloin voitiin todeta pohjan olevan valmis käyttöön.

Versionhallintaa varten luotiin uusi kansiorakenne, johon aluksi kopioitiin luotu malliprojekti nimellä “host”, jonka tarkoitus on edustaa isäntäsovellusta. Useamman micro frontend -sovelluksen kehitysaikaisen hallinnoimisen helpottamiseksi juurikansioon lisättiin oma erillinen package.json-tiedosto, johon määritettiin käynnistysskriptejä, joiden avulla sovellukset ja tarvittavat simulaattorit voitiin käynnistää suoraan sovellusten juurikansiosta. Tässä käytettiin apuna concurrently-kirjastoa, joka osaa yhdistää ja ajaa useita komentoja rinnakkain. Näin saatiin luotua käynnistysskripti siten, että kaikki sovellukset käynnistyvät yhtäaikaisesti yhdellä komennolla, ja kaikkien sovellusten kehityspalvelimen lokit voitiin lukea yhdestä konsolista.

Seuraavaksi tehtiin isäntäsovellukseen tarvittavat asetukset Module Federation -laajennuksen käyttöönottoa varten. Webpackin määritystiedostossa otettiin käyttöön Re.Packin Module Federation -laajennus ja määritettiin React ja React Native riippuvuudet jaetuiksi riippuvuuksiksi. Kaikki natiiveja moduuleja sisältävät kirjastot tuli dokumentaation mukaan asentaa niitä käyttävän etäsovelluksen lisäksi myös isäntäsovellukseen ja määrittää jakamisasetukseksi singleton- ja eager-tilat, jolloin kirjastosta sallitaan vain yksi ja sama versio kaikille sitä käyttäville sovelluksille konfliktien välttämiseksi. Poikkeuksena tästä ovat Reactin ja React Nativen riippuvuudet, joille tuli asettaa Re.Pack-kirjaston oma konfiguraatio. Määritykset toteutettiin Webpack.config.cjs tiedostoon käyttäen Re.Packin Module Federation -laajennusta (kuva 10).

```
new Repack.plugins.ModuleFederationPlugin({
  name: 'host',
  shared: {
    react: {
      ...Repack.Federated.SHARED_REACT,
      requiredVersion: '18.1.0',
    },
    'react-native': {
      ...Repack.Federated.SHARED_REACT_NATIVE,
      requiredVersion: '0.70.6',
    },
    'react-native-safe-area-context': {
      singleton: true,
      eager: true,
      requiredVersion: '4.4.1',
    },
```

Kuva 10. Natiiveja moduuleita sisältävien kirjastojen jakamisasetukset Module Federation -laajennuksessa

Re.Packin Module Federation -laajennuksen eroavaisuus web-pohjaiseen verrattuna on siinä, että isäntäsovelluksessa ei voida esimäärittää lisättäviä etäsovelluksia tai omia ulospäin jaettavia komponentteja laajennuksessa suoraan tavallisesti web-ympäristössä käytettävien remote- ja exposes-ominaisuuksien kautta. Sen sijaan tätä varten isäntäsovellukseen toteutettiin dokumentaation mukaisesti erillinen Re.Packin ScriptManager-toiminnon avulla toteutettava resolveri, jonka tarkoitus on selvittää dynaamisesti liitettyjen etäsovellusten URL-osoitteet sovellusten ajoaikana. Eri sovellusten kehityspalvelimien URL-osoitteet määritettiin Re.Packin createURLResolver-metodin containers-ominaisuuteen, jonka avulla isäntäsovellus pystyy yhdistämään tarvittavaan etäsovellukseen (Kuva 11).

```
const resolveURL = Federated.createURLResolver({
    containers: {
      account: 'http://localhost:9101/[name][ext]',
      asset: 'http://localhost:9103/[name][ext]',
      home: 'http://localhost:9102/[name][ext]',
      lib: 'http://localhost:9104/[name][ext]',
      tracking: 'http://localhost:9105/[name][ext]',
      vehicle: 'http://localhost:9106/[name][ext]',
    },
});
```

Kuva 11. Etäsovellusten URL-osoitteiden määrittäminen isäntäsovellukseen resolverimetodin avulla

Etäsovellukset tai niiden komponentit voidaan liittää toisiinsa Reactin lazy-metodin eli ns. laiskan latauksen avulla. Isäntäsovelluksessa etäsovelluksen liittäminen jouduttiin toteuttamaan tavallisesti käytettävän import-toiminnon sijaan Re.Packin importModule-metodin avulla, jossa määritetään liitettävän etäsovelluksen nimi ja etäsovellukselta paljastettu komponentti. Reactin Suspense-

komponentilla määritettiin etäsovelluksen lataamisesta ilmoittava teksti siltä varalta, että etäsovelluksen lataaminen on jostain syystä hidasta tai se epäonnistuu (kuva 12).

```
const HomeApp = React.lazy(() => Federated.importModule('home', './HomeApp'));

return (
  <React.Suspense
    fallback={
      <View style={styles.container}>
        <Text>Ladataan...</Text>
      </View>
    }
  >
    <HomeApp />
  </React.Suspense>
);
};
```

Kuva 12. Etäsovellukselta paljastetun komponentin liittäminen isäntäsovellukseen

Etäsovelluksissa toisten etäsovellusten tai niiden komponenttien liittäminen ja lataaminen voidaan tehdä normaalisti import-toimintoa käyttäen, eivätkä ne tarvitse myöskään erillistä resolveria, sillä isäntäsovellus hoitaa yhteydet niiden välillä. Näin ollen myös Module Federation -määritelmät on etäsovelluksissa mahdollista tehdä lähes samalla tavoin kuin web-pohjaisessa versiossa. Kuvassa 13 on havainnollistettu etäsovelluksissa käytettävää Webpack-konfiguraatiota komponenttien ja toimintojen lukemiseksi muilta etäsovelluksilta sekä komponenttien paljastamiseksi muille etäsovelluksille.

```
new Repack.plugins.ModuleFederationPlugin({
  name: 'home',
  exposes: {
    './HomeApp': './App.tsx',
  },
  remotes: {
    account: 'account@dynamic',
    asset: 'asset@dynamic',
    vehicle: 'vehicle@dynamic',
    tracking: 'tracking@dynamic',
  },
  shared: {
    react: {
      ...Repack.Federated.SHARED_REACT,
      requiredVersion: '18.1.0',
    },
```

Kuva 13. Etäsovelluksissa käytettävä Webpack-konfiguraatio tietojen lukemiseen ja välittämiseen etäsovellusten välillä

Webpack-konfiguraatiotiedostossa luettavat etäsovellukset määritetään kohdassa ”remotes” ja määrityksessä käytetään muotoa ”nimi@dynamic”, jossa

nimi tarkoittaa etäsovelluksen name-kenttään määritettyä sovelluksen nimeä. Kohdassa ”exposes” määritetään paljastettavat komponentit sillä periaatteella, että avain on nimi, jolla komponenttiin viitataan lukevassa päässä olevassa etäsovelluksessa ja arvo on komponentin paljastavassa etäsovelluksessa käytettävä polku ja nimi.

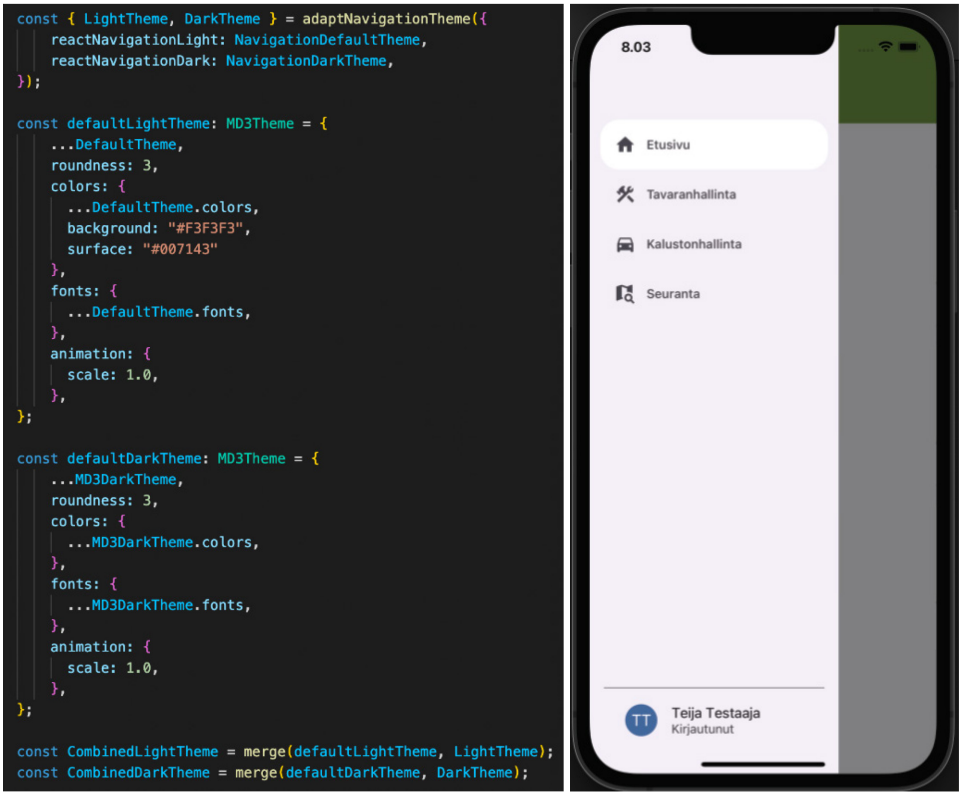
5.2 Käyttäjän todennus ja navigointi

Sovelluksen keskitetty tyyli toteutettiin luomalla mikrosovellusten välillä jaettavalla teemalla. Tätä varten asennettiin React Native Paper -kirjasto tarvittavine riippuvuuksineen kirjaston virallisen ohjeistuksen mukaisesti kotisovelluksen lisäksi isäntäsovellukseen. Navigointia varten asennettiin lisäksi myös React Navigation -kirjasto, joka toimii hyvin yhteen React Native Paper -kirjaston kanssa. Käyttäjänhallintasovellukseen sekä isäntäsovellukseen asennettiin käyttäjän todennusta sekä sovelluksen käyttöoikeuksien hallinnoimista varten React Native App Auth- ja React Native Keychain -kirjastot.

Alkuvaiheen asetusten ja valmistelujen jälkeen alkoivatkin dokumentaation puutteellisuudesta ja osin teknologioiden uutuudesta tai kehittymättömyydestä johtuvat ongelmat. Eri kirjastojen välinen riippuvuuksien yhteensopimattomuus aiheutti paljon erikoisia ongelmia, ja ne tuntuivat ilmenevän aivan satunnaisesti ja epäloogisesti, jolloin toimivan version löytäminen olikin välillä täysin hakuammuntaa. Lopulta suurimmaksi yhteensopivuusongelmien aiheuttajaksi osoittautui kehitysympäristön asennusohjeiden puutteellisuus M1-sirua käyttävien tietokoneiden osalta. Ongelma oli erityisesti iOS-riippuvuuksien hallintaan käytettävässä Cocoapods-työkalussa, joka React Nativen virallisten ohjeiden mukaan asennettuna epäonnistui joko kokonaan tai osittain tarvittavien riippuvuuksien asentamisessa, jolloin projekti ei enää kääntynyt ja syntyneet virheviestitkin viittasivat mitä erikoisimpiin paikkoihin. Kehitysympäristön asennus onnistui lopulta noudattamalla epävirallisilla keskustelupalstoilla annetuilla ohjeilla, jolloin myös yhteensopivuusongelmat viimein katosivat.

Kun kirjastot saatiin onnistuneesti asennetuksi ja projekti jälleen toimivaksi, luotiin React Native Paper -teematiedosto, johon määritettiin sovelluksen oletustyylit. Paper- ja Navigation-kirjastoilla on molemmilla omat oletusteemansa, jotka tuli yhdistää yhdeksi teemaksi muokattavien ominaisuuksien kanssa.

Teematiedosto syötettiin React Navigation ja React Native Paper -kirjastojen tarjoajakomponentteihin, jotka käärittiin kotisovelluksen pääkomponentin ympärille, jolloin teema saatiin jaettua navigaation kautta myös muille etäsovelluksille. Kotisovellukseen toteutettiin sen jälkeen yläpalkki- ja sivunavigaatiokomponentit sekä niiden alustavat reititykset React Navigation -kirjaston Drawer- ja Native Stack -navigointikomponenttien avulla. Sovelluksen teemat ja sivuvalikko on esitetty kuvassa 14.



Kuva 14. Teematiedostojen yhdistäminen ja sovelluksen sivuvalikko

Keycloak-järjestelmään integroinnin toteuttaminen aloitettiin luomalla järjestelmään valmis testiorganisaatio ja sille testikäyttäjä, jota voitiin käyttää kirjautumisen toimivuuden testaamiseen. Sovelluksen yhdistämiseksi Keycloakiin asetettiin yhteyden muodostamiseen tarvittavat API-osoitteet ja salausavaimet ympäristömuuttujiin, jotka liitettiin React Native App Auth -kirjaston määritysobjektiin. React Native App Auth -kirjasto tarjoaa todennusjärjestelmän kanssa kommunikointiin valmiita metodeita, kuten esimerkiksi authorize-metodin, jota kutsumalla sovellus ohjautuu erilliseen selainikkunaan avautuvalle kirjautumissivulle. React Native Keychain -kirjaston avulla toteutettiin tokenin hakemiseen ja tallentamiseen tarvittavat apufunktiot kirjaston valmiiden getGeneric-

Password- ja setGenericPassword-metodien avulla. Keychain tallentaa halutun tiedon avain-arvo-pareina laitteen sisäiseen salaisuuksienhallintasovellukseen. Kuvassa 15 on esitettynä esimerkit Keycloak-integraation ja Keychain-kirjaston toimintojen apufunktioista.

```
const config: AuthConfiguration = {
    issuer: `${ISSUER_URL}`,
    clientId: `${CLIENT_ID}`,
    redirectUrl: `${REDIRECT_URL}`,
    scopes: ['openid', 'profile'],
};

const login = async () => {
    try {
        const result = await authorize(config);
        return result;
    } catch (error) {
        console.log(error);
    }
};
```

```
const setToken = async (key: string, value: string): Promise<any> => {
    try {
        const response = await Keychain.setGenericPassword(key, value);

        if (response) {
            console.log('Token successfully set');
            return response;
        } else {
            console.log('Token was not stored');
        }
    } catch (err) {
        console.log(err);
    }
};
```

Kuva 15. Esimerkit keycloak-integraatioon (vasen) ja Keychain-toimintoihin (oikea) tarvittavista asetuksista ja apufunktioista

Käyttäjän todennuksen tilan tulee olla ajan tasalla kaikissa mikrosovelluksissa, joten tila tuli voida jakaa niiden kesken. Tilan tallettamista varten käytettiin apuna Reactin useReducer-metodia (kuva 16). Samaa globaalia tilaa voidaan myöhemmin laajentaa käyttäjän käyttöoikeuksien varmistamiseen rajoitettujen toimintojen osalta.

```
export const AuthProvider : React.FC<{
  children: React.ReactNode;
}> = ({ children }) => {
  const [state, dispatch] = React.useReducer(
    (prevState: AuthState, action: AuthAction): AuthState => {
      switch (action.type) {
        case 'LOG_IN':
          return {
            ...prevState,
            isLoggedIn: true,
            isLoading: action.isLoading
          };
        case 'LOG_OUT':
          return {
            ...prevState,
            isLoggedIn: false,
            isLoading: false
          };
        case 'LOADING_STATE_CHANGE':
          return {
            ...prevState,
            isLoading: action.isLoading
          };
        default:
          return prevState;
      }
    },
    initState
);
```

Kuva 16. Käyttäjän todennuksen tilanhallintaan käytettävä useReducer-metodin määritys

Varsinaiset sisään- ja uloskirjautumisfunktiot toteutettiin aiemmin luotujen apufunktioiden avulla ja päivittämällä sovelluksen tila vastaamaan käyttäjän todennuksen tilaa. Sisäänkirjautumisen yhteydessä käynnistetään kirjautumistoiminto Keycloakin kautta, jonka jälkeen kirjautumisen yhteydessä vastauksena saatu token tallennetaan Keychainin avulla laitteelle ja muutetaan sen jälkeen tila kirjautuneeksi. Uloskirjautumisen yhteydessä tila muutetaan uloskirjautuneeksi. Toiminnoista muodostettiin tarjoajakomponentti, johon toiminnoista muodostettu konteksti syötettiin propsina, josta se voitiin jakaa kaikille alikomponenteille. Kuvassa 17 on esitetty tarjoajakomponentin muodostaminen sekä siihen määritellyt sisään- ja uloskirjautumisfunktiot.

```
const logIn: AuthContextActions["logIn"] = async () => {
    try {
      const authenticate = await login();

      if(authenticate) {
        await setToken(
          "jojoba-accessToken",
          authenticate.accessToken
        );
        dispatch({
          type: 'LOG_IN',
          isLoggedIn: true,
          isLoading: false
        });
      };
    } catch(err) {
      console.log(err);
    }
};

const logOut: AuthContextActions["logOut"] = async () => {
    try {
      await logout();
      dispatch({ type: 'LOG_OUT' });
    } catch (err) {
      console.log(err);
    };
};

return (
    <AuthContext.Provider
      value={{logIn, logOut, ...state }}
    >
      {children}
    </AuthContext.Provider>
  )
};
```

Kuva 17. Todennuksen tilasta ja toiminnoista muodostettava tarjoajakomponentti

Tarjoajakomponentti käärittiin navigointi- ja tyylikirjastojen tarjoajakomponenttien tapaan kotisovelluksen ylimpään komponenttiin tietojen jakamiseksi myös kaikkiin etäsovelluksiin (Kuva 18). Tarvittaviin ominaisuuksiin voidaan alikomponenteissa viitata kirjastojen omilla siihen tarkoitetuilla metodeilla. Esimerkiksi teematiedoston ominaisuuksiin pääsee käsiksi useTheme-metodilla ja navigaation ominaisuuksiin useNavigation-metodilla. Kirjautuneen käyttäjän

tietojen lukemista varten käyttäjänhallintasovellukseen tehtiin oma vastaavalla tavalla toimiva useAuth-metodi.

```
const App : React.FC<{}> = () => {
  const theme = CombinedLightTheme;

  return (
    <AuthProvider>
      <PaperProvider theme={theme}>
        <NavigationContainer theme={theme}>
          <DrawerMenu />
        </NavigationContainer>
      </PaperProvider>
    </AuthProvider>
  );
};

export default App;
```

Kuva 18. Sovelluksen tila ja määrittelyt jaetaan muille etäsovelluksille ja alikomponenteille tarjoajakomponenttien kautta

Kotisovellukseen toteutettiin yksinkertainen aloitusnäkymä, johon sisällytettiin kirjautumisnappi ja lisäksi rekisteröitymisnappi valmiiksi odottamaan jatkokehitystä. Kirjautumisnappia painamalla avautuu laitteen selainikkunaan Keycloak-kirjautumislomake. Aloitus- ja kirjautumisnäkymät on esitetty kuvassa 19.

Kuva 19. Sovelluksen aloitus- ja kirjautumisnäkymä

Drawer-navigaatiokomponenttiin toteutettiin ehto, joka näyttää käyttäjän todentamisen tilan perusteella sopivan näkymän. Jos käyttäjä ei ole kirjautunut järjestelmään, näytetään käyttäjälle sovelluksen alkunäkymä. Kirjautumisen jälkeen käyttäjä ohjataan automaattisesti sovelluksen kotinäkymään, josta pääsee käsiksi sovelluksen muihin näkymiin. Alkunäkymän swipeEnabled-

asetus estää myös Drawer-sivupalkin käyttämisen ilman kirjautumista. Drawer-navigaatiokomponentti on esitetty kuvassa 20.

```
const SideDrawer : React.FC<{}> = () => {
  const { isLoggedIn, isLoading } = useAuth();

  return (
      <Drawer.Navigator
        useLegacyImplementation
        drawerContent={(props) => <CustomDrawerContent {...props} />}
      >
        {isLoggedIn && !isLoading
          ? <Drawer.Screen
              name="Root"
              component={Routes}
              options={{headerShown: false}}
            />
          : <Drawer.Screen
              name="Kirjaudu"
              component={LoginScreen}
              options={{headerShown: false, swipeEnabled: false}}
            />
        }
      </Drawer.Navigator>
  );
};
```

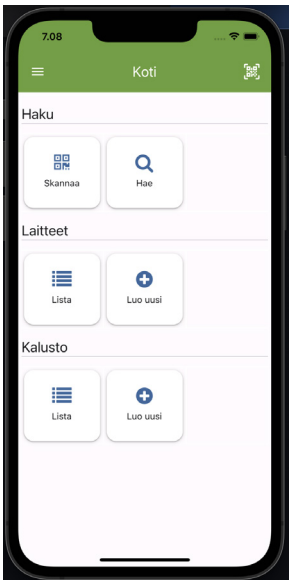
Kuva 20. Drawer-navigaatiokomponentti

Muut sovelluksen sisällä käytettävät reitit toteutettiin Drawer-komponentin reittien sisään sijoitetun Native Stack Navigator -komponentin avulla, joka mahdollistaa myöhemmin vastaavalla tavalla toteutettuna esimerkiksi käyttäjien roolien perusteella pääsyn rajoittamista eri osiin sovellusta. Kuvassa 21 on esitetty Drawer-navigaatiokomponentin sisäinen Stack-navigaatiokomponentti. Etäsovellukset ladataan laiskasti lazy-metodin avulla eli vasta siinä vaiheessa, kun käyttäjä siirtyy ensimmäisen kerran haluttuun etäsovelluksessa sijaitsevaan näkymään.

```
const AssetsScreen = React.lazy(() => import('asset/AssetsScreen'));
const DetailsScreen = React.lazy(() => import('asset/DetailsScreen'));
const VehiclesScreen = React.lazy(() => import('vehicle/VehiclesScreen'));
const TrackingScreen = React.lazy(() => import('tracking/Scanner'));
const NewLocation = React.lazy(() => import('tracking/LocationForm'));

const Navigation : React.FC<{}> = () => {
    const Stack = createNativeStackNavigator();

    return (
      <Stack.Navigator
        screenOptions={{
          header: (props) => <NavBar title={props.route.name} {...props} />
        }}
        initialRouteName="Koti"
      >
        <Stack.Screen name="Koti" component={HomeScreen} />
        <Stack.Screen name="Tavarat" component={AssetsScreen} />
        <Stack.Screen name="Tiedot" component={DetailsScreen} />
        <Stack.Screen name="Kalusto" component={VehiclesScreen} />
        <Stack.Screen name="Seuranta" component={TrackingScreen} />
        <Stack.Screen name="Siirrä" component={NewLocation} />
      </Stack.Navigator>
    )
};
```

Kuva 21. Sovelluksen sisäinen reititys

Seuraavaksi toteutettiin yksinkertainen kotivalikkonäkymä, johon käyttäjä ohjataan onnistuneen kirjautumisen jälkeen (kuva 22). Kotivalikon pikalinkkinapit toteutettiin myös vastaavasti laiskasti lataamalla ja hyödyntämällä Suspense-komponenttia, jotta pienempiä sovelluskokonaisuuksia ajettaessa linkit päivittyvät vastaamaan sen hetkistä tilannetta.



Kuva 22. Kotivalikkonäkymä

Etäsovellusten välillä jaettavia komponentteja varten toteutettuun erilliseen etäsovellukseen asennettiin samat tyyli- ja navigointikirjastot ja luotiin valmiiksi muutamia jaettavia komponentteja, kuten nappi-, kortti- ja lomakekomponentteja, joiden tuli olla tyyliltään yhteneväisiä eri osissa sovellusta.

5.3 Tavaran- ja kalustonhallinta

Tavaran- ja kalustonhallintasovelluksissa sekä seurantasovelluksessa käytettävää MongoDB Realm -tietokannan automaattisen synkronoinnin käyttöönottoa varten luotiin MongoDB Atlas -palveluun uusi App Service -palvelu MongoDB:n dokumentaation mukaisesti sekä asetettiin App sync ja Custom JWT -toiminnot päälle. Custom JWT -asetuksen avulla Realmin käyttäjän todentaminen voitiin toteuttaa Keycloakin kautta. Sovelluksessa tietokanta otettiin käyttöön asentamalla Realmin virallinen React Native SDK -paketti sekä Realmin käyttöä helpottava @realm/react-paketti MongoDB:n ja Re.Packin dokumentaatioita noudattaen.

Realmin asennus ja käyttöönotto ei onnistunut aivan suoraviivaisesti. Webpackin avulla ajettuna sovellus jumiutui aina yrittäessä avata näkymiä, joissa Realm oli käytössä. Virheilmoituksista ei ollut juurikaan apua, eikä niitä aina edes syntynyt. Vianetsintä voitiin toteuttaa lopulta ajamalla ainoastaan yksittäinen etäsovellus käyttämällä React Native -projektin mukana tullutta Metro-paketointityökalua sekä Realmin sisäänrakennettuja Debugger-metodeita, jotka tuottivat tulosteet koko prosessista virheilmoituksineen. Tämän jälkeen virhe kerrallaan korjattuna ja erilaisia webpack-konfiguraatioita kokeilemalla saatiin myös Realmia käyttävät osat toimimaan.

Seuraavaksi määritettiin tietokantamallit tavaroille ja kalustolle. Skeema syötettiin Realm konfiguraatio-objektiin, josta muodostettiin kontekstiobjekti, jotta sen jakaminen eri komponenttien välillä olisi helpompaa. Tavaranhallintasovelluksen listausnäkymäkomponentti käärittiin kontekstin tarjoajakomponentin lisäksi @realm/react-kirjaston edellyttämien UserProvider-, AppProvider- ja RealmProvider-tarjoajakomponenttien sisään. Autentikaatio tehtiin UserProvider-tarjoajakomponentin fallback-ominaisuuden sisällä olevassa komponentissa, jossa Reactin useEffect-toiminnon avulla todennettiin käyttäjä Custom JWT -menetelmällä MongoDB Atlas -palvelun kanssa (kuva 23).

```
const RealmFallback: React.FC<{}> = () => {
  const app = useApp();

  useEffect(() => {
    const getUser = async () => {
      const jwt = AuthStore.getToken();
      const credentials = Realm.Credentials.jwt(jwt);
      try {
        const user = await app.logIn(credentials);
        console.log("user", user);
        return user;
      } catch (e) {
          console.log('Error logging in', e);
      }
    };

    getUser();
  }, []);

  return (
    <Text>Ladataan...</Text>
  );
};
```

Kuva 23. Realm-tietokannan käyttäjän todennus

Realmin asennuksen ja konfiguroinnin jälkeen toteutettiin tavaroiden listausnäkymä. Listausnäkymän tiedot haettiin ensisijaisesti Realmin kautta, mutta

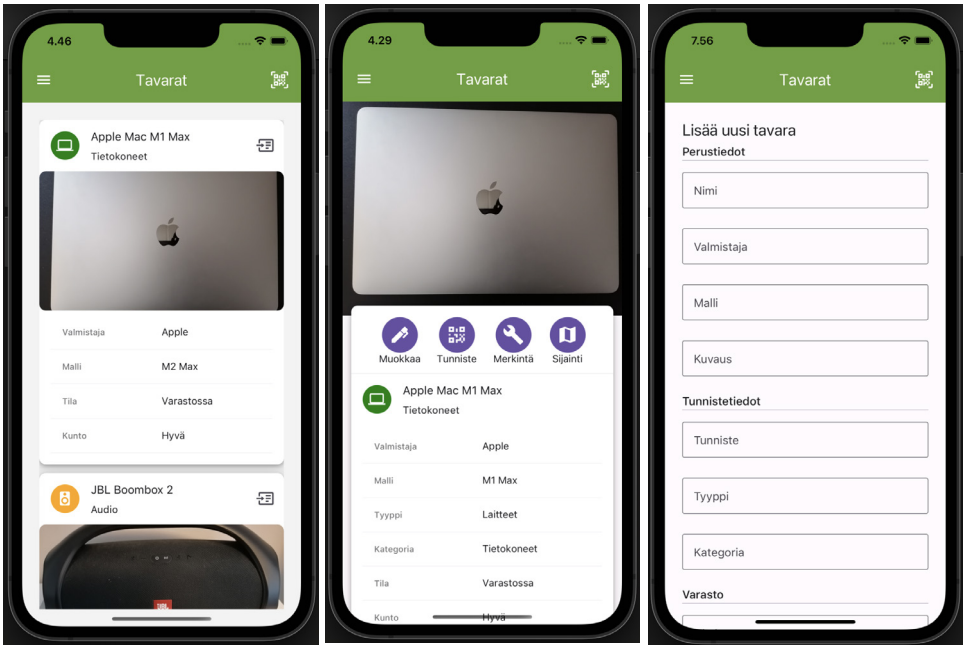
myöhemmin lisättävien toimintojen lisäksi Realmin App sync -toiminnon virhetilanteisiin varautumiseksi toteutettiin tietojen hakeminen ja tallentaminen lisäksi myös suoraan web-sovelluksen REST API -rajapinnan kautta. Tätä varten luotiin funktiot tietojen hakemiselle ja tietokantaan tallentamiselle, jotka määritettiin käytettäväksi silloin, jos Realmissa tapahtuu jokin virhe. Muutamia REST API -rajapinnan ja Realm-tietokannan CRUD-toimintoihin liittyviä funktioita on esitetty kuvassa 24.

```
const getApiData = async (fetchUrl: string) => {
  try {
    const response = await fetch(fetchUrl);
    const data = await response.json();
    setAssets({ data: data.products, loading: false });
  } catch (error) {
    console.error(error);
  }
};
```

```
const getAll = () => {
    const result = useQuery(Asset);
    return result;
};

const create = (asset: Asset) => {
    realm.write(() => {
        realm.create("Asset", asset);
    })
};
```

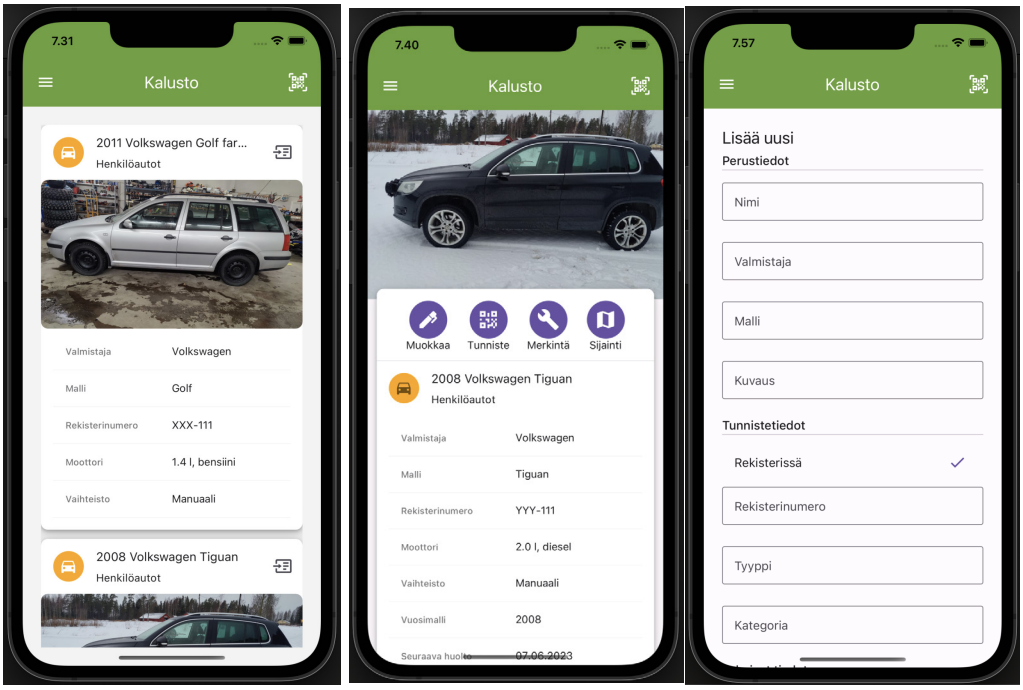
Kuva 24. REST API -rajapinnan (vasen) ja Realm-tietokannan (oikea) CRUD-operaatioita

Tavaranhallintaan luotiin tavaroiden listaamisnäkymä, tietonäkymä sekä uuden tavaran luonti- ja muokkausnäkymät, jotka on esitetty kuvassa 25. Listausnäkymän toimintonappia painamalla pääsee kyseisen tavaran tietonäkymään ja FAB-nappia (Floating Action Button) painamalla avautuu uuden tavaran luontinäkymä. Tietonäkymän toimintonapista pääsee muokkaamaan tavaran tietoja.



Kuva 25. Tavaranhallinnan listaus-, muokkaus- ja tietonäkymät.

Vastaavasti kalustolle toteutettiin samat näkymät suurimmaksi osaksi samanlaisella asettelulla ja ulkoasulla. Kaluston ominaisuudet poikkeavat jonkin verran tavaroista, joten niiltä osin tietonäkymä toteutettiin luonnollisesti hieman erilaisella asettelulla. Kalustonhallinnan näkymät on esitetty kuvassa 26.



Kuva 26. Kalustonhallinnan listaus-, luonti- ja tietonäkymät

Samoja jaettavia komponentteja käyttävissä näkymissä oli havaittavissa ajoittain ongelma, jossa jaetut komponentit eivät latautuneet näkymään navigoitaessa etäsovellusten välillä. Ongelman sai poistumaan poistamalla muihin samoja komponentteja käyttäviin etäsovelluksiin vievät reitit väliaikaisesti käytöstä ja säilyttämällä vain sen, mihin kulloinkin oli tarkoitus navigoida. Virheilmoituksia tilanteesta ei syntynyt ollenkaan, eikä ongelma esiintynyt muissa etäsovelluksissa tai näkymissä. Aiemmat vastaavat ongelmat osoittautuivat johtuvaksi jostain riippuvuuksien määrittelyn virheestä Module Federation -laajennuksessa, mutta tarkkaa syytä tähän ei löytynyt. Ongelma ei kuitenkaan estänyt kehitystyön jatkamista, joten ongelmasta tehtiin ilmoitus kirjastoa ylläpitävälle taholle ja jatkettiin kehitystä.

5.4 Seuranta

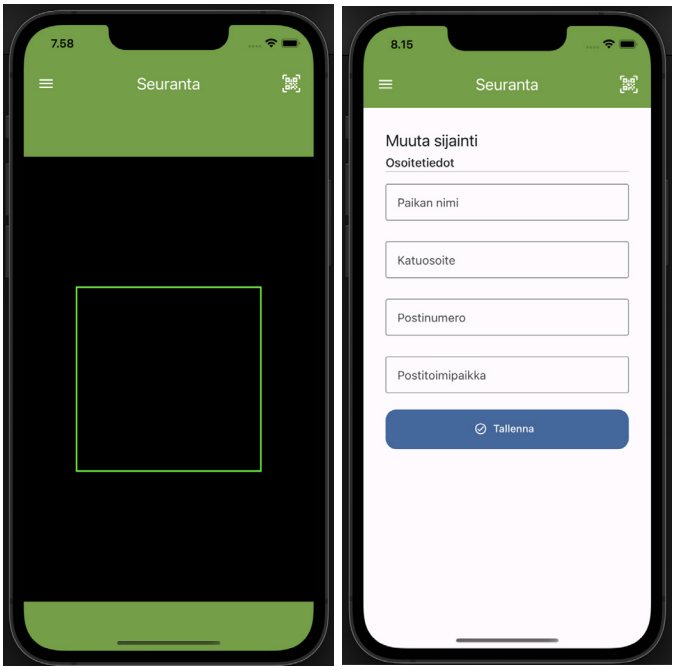
Seurantatoimintojen toteutusta varten asennettiin React Native Camera-, React Native QRcode Scanner- ja React Native Permissions -kirjastot seu-

ranta- ja isäntäsovelluksiin sekä toteutettiin niiden dokumentaatioiden mukaiset asetukset. QR-koodin skannausnäkymä toteutettiin pyytämällä ensin tarvittavat käyttöoikeudet kameran käyttöön React Native Permissions -kirjaston request-metodin avulla. Kun käyttäjä hyväksyy pyynnön, renderöidään varsinainen skannausnäkymä QRcodeScanner-komponentin avulla (kuva 27).

```
useEffect(() => {
  request(
    Platform.OS === 'ios'
    ? PERMISSIONS.IOS.CAMERA
    : PERMISSIONS.ANDROID.CAMERA
  ).then((result) => { setHasPermission(result) });
}, []);

if(hasPermission !== '' && hasPermission !== 'granted') {
  return <Text>Access denied</Text>;
}

const getData = (code: string) => {
  const item = assets.find(code);
  return item;
};

const handleBarCodeScanned = (event: BarCodeReadEvent) => {
  const item = getData(event.data);

  item
    ? navigation.navigate("Siirrä", { assetData: item })
    : navigation.navigate("Luo");
};

return (
    <QRCodeScanner
      onRead={(e) => handleBarCodeScanned(e)}
      flashMode={RNCamera.Constants.FlashMode.torch}
      bottomViewStyle={styles.bottom}
      topViewStyle={styles.bottom}
      cameraStyle={styles.camera}
      showMarker={true}
    />
);
```

Kuva 27. QR-koodin skannaamiseen käytettävä komponentti

QR-koodin skannauksen jälkeen sovellus hakee tietokannasta koodia vastaavan tavaran ja ohjautuu automaattisesti kyseisen tavaran tietonäkymään, josta voidaan siirtyä joko muokkaamaan tavaran tietoja tai sen sijaintia. Jos koodia vastaavaa tavaraa ei löydy, sovellus ohjautuu automaattisesti uuden tavaran luontilomakkeelle. Tavaran sijainnin muutos toteutetaan tässä vaiheessa erillisellä manuaalisesti täytettävällä lomakkeella. Kuvassa 28 on nähtävissä seurantasovelluksen skannaus- ja sijaintilomakenäkymät.



Kuva 28. Seurantasovelluksen sijainnin muutos- ja skannausnäkymät

Lopuksi sovellusta testattiin vielä manuaalisesti kokeilemalla, että sovelluksen kaikki osat toimivat hyvin yhdessä ja kaikki oleelliset toiminnot ovat saatavilla. Sovellusta testattiin emulaattoreiden lisäksi fyysisillä laitteilla. Lisäksi sovellusta testattiin myös käynnistämällä vain tietyistä etäsovelluksista koostettuja kokonaisuuksia ja testaamalla osien toimivuutta sellaisenaan.

6 LOPPUTULOKSEN ARVIOINTI

Projektin lopputuloksena syntyi toimeksiannossa määritellyt perusedellytykset täyttävä micro frontend -arkkitehtuurilla toteutettu tavaran- ja kalustonhallintasovellus iOS ja Android -laitteille. Sovellus sisälsi alkuvaiheessa määritellyt tavaroiden ja kaluston hallinta- ja seurantatoiminnot. Sovellus integroitiin nykyiseen web-sovellukseen REST API -rajapinnan sekä MongoDB Realm -tietokannan synkronointitoimintojen avulla. Realm-tietokannan avulla voitiin lisäksi mahdollistaa sovelluksen toiminta offline-tilassa. Micro frontend -arkkitehtuurin toteuttamiseen käytettiin Webpack-pohjaista React Native -sovelluskehykselle kehitettyä Re.Pack-paketointityökalua.

Kokonaisuutena projektin toteutus onnistui pääosin hyvin sujuvasti. Varsinaisten hallinta- ja seurantatoimintojen toteutuksessa haasteellisinta oli toimintojen rajaaminen, sillä alkuperäinen järjestelmä sisälsi paljon myös tämän työn yhteydessä toteutettuihin toimintoihin liittyviä lisätoimintoja ja riippuvuuksia, jotka

olivat oleellisia koko järjestelmän käyttötarkoituksen kannalta. Varsinaiset toiminnot toteutettiin tässä vaiheessa melko yksinkertaisina ja keskityttiin enemmän uudenlaisen arkkitehtuurin soveltamiseen, sillä se oli jatkokehityksen tarpeiden sekä soveltuvuuden arvioinnin kannalta merkityksellisin osuus. Toiminnot noudattivat samaa kaavaa olemassa olevan järjestelmän toimintojen kanssa, joten varsinaisia vaikeuksia ei kehitysympäristön ja Realmin aiheuttamien ongelmien lisäksi toteutuksessa tullut vastaan.

Re.Packin mahdollistama Module Federation -tuki teki micro frontend -arkkitehtuurin toteuttamisesta mobiiliympäristössä odotettua sujuvampaa. Webpack-konfiguraatiot toimivat myös muutamaa poikkeusta lukuun ottamatta suurilta osin samalla tavoin kuin web-sovelluksissa, joten aiemmasta kokemuksesta web-sovellusten parissa oli hyötyä toteutuksessa. Suurin eroavaisuus oli isäntäsovelluksen konfiguraatiossa, joka edellytti erillistä resolveria etäsovellusten lataamiseen, mutta siihenkin tarjottiin dokumentaatiossa ja esimerkeissä täysin käyttövalmiita ratkaisuja. Jatkossa sovelluksista on myös helppo muodostaa sopivia kokonaisuuksia esimerkiksi käyttäjän käyttöoikeuksien tai eri tilaustasojen perusteella jättämällä pois tarpeettomia etäsovelluksia kokonaisuudesta tai lisäämällä eri toimintoja käyttäjän tarpeen mukaan, jolloin käytettävä sovellus sisältää aina vain tarpeenmukaisen sisällön.

Arkkitehtuurin suunnittelussa monimutkaisinta oli micro frontend -sovellusten jakoperiaatteiden päättäminen. Toimeksiannossa määritellyt tavoitteet oli mahdollista toteuttaa useammalla eri jakoperiaatteella, joista jokaisella oli omat hyvät ja huonot puolensa, joten selkeän näkemyksen muodostaminen jonkin periaatteen sopivuudesta jonkin muun sijaan oli vaikea muodostaa. Haastetta lisäsi vielä se, että jatkossa kasvavan ja monimutkaistuvan sovelluksen osalta piti tarkoin rajata, miten pitkälle tulevaisuuteen on järkevää yrittää varautua. Toteutettu jako toimii hyvin pienessä kokonaisuudessa, mutta isommassa mittakaavassa se voi olla vaikeammin hallittavissa. Toisaalta Module Federation -lähestymistapa toimii niin joustavasti, että selkeällä koodirakenteella toteutettuna rajausten muuttaminen myöhemmin ei välttämättä ole suurikaan ongelma.

Arkkitehtuurin toteutus oli melko työlästä, koska jokaista etäsovellusta varten jouduttiin luomaan kokonaan uusi projekti, joista jokaiseen tuli toteuttaa omat

konfiguraatiot ja kirjastojen asennukset. Työläyttä kasvatti vielä tarve tehdä konfiguraatiot sekä Android että iOS -alustoille jokaisen sovelluksen osalta. Toisaalta pakollisiin vaiheisiin syntyi melko nopeasti hyvä työskentelyrutiini, mikä vauhditti toistuvia työvaiheita. Selkeästi haasteellisinta oli kuitenkin virheiden etsiminen, johon varmasti suurilta osin vaikutti kokemattomuus mobiiliympäristöistä ja käytetyistä teknologioista. Kaikista virhetilanteista ei syntynyt varsinaisesti mitään virheilmoituksia tai ne viittasivat vääriin osiin koodia. Lisäksi samoja virheitä oli harvoin raportoitu minkään kirjaston ylläpitäjille, joten edes keskustelua vastaavista ongelmista ei juurikaan löytynyt. Välillä vianetsinnässä auttoi ainoastaan yksittäisen etäsovelluksen ajaminen itsenäisenä sovelluksena, jolloin virheilmoitukset saattoivat näkyä oikein. Webpack-konfiguraatioon liittyviin ongelmiin auttoi toisinaan yksittäisen etäsovelluksen ajaminen Metron avulla, jolloin virheilmoitukset saattoivat olla kuvaavammat ja auttoivat ongelman alkulähteelle.

Re.Pack tuntui Module Federation -tuen puolesta jo melko vakaalta työkalulta ja soveltuvan melko hyvin mobiilisovelluksiin. Tulosta tosin saattaa vääristää työssä toteutetun sovelluksen pieni koko. Micro frontend -arkkitehtuurin avulla yksittäisten sovellusten koko saatiin pidettyä melko pienenä ja arkkitehtuurin mahdollistama sovelluksen kääntäminen ja paketoiminen vain tarpeellisilta osin kehityksen edetessä mahdollisti monien työvaiheiden nopeutumisen. Arkkitehtuurin mukanaan tuomat toistuvat työvaiheet ja toteutuksen monimutkaisuus selkeästi kuitenkin hidastivat kehitystyötä melko paljon, joten arkkitehtuurin hyödyntämistä ei voida suositella varsinaisesti pienille organisaatioille, joissa työvaiheita ei pystytä tehokkaasti toteuttamaan rinnakkain. Lisäksi edistyneemmät toiminnot todennäköisesti edellyttävät melko kokeneita kehitystiimejä, jotka tuntevat hyvin mobiilialustojen ominaisuudet ja Webpack-konfiguraatiot. Suurille organisaatioille ja monimutkaisille sovelluksille micro frontend -arkkitehtuurin toteuttaminen Re.Packin avulla sen sijaan voisi toimia hyvin.

Jatkokehitysvaiheessa toteutettuihin toimintoihin lisätään tarpeelliset riippuvaisuudet ja moduulit, jotta sovelluskokonaisuus kattaisi vielä laajemmin alkuperäisen järjestelmän oleellisimpia toimintoja ja vastaisi käyttötarkoitustaan paremmin. Näitä ovat esimerkiksi edistyneemmät hallinta- ja seurantatoiminnot, automaattinen ajopäiväkirja, työkohteiden hallinta sekä myöhemmin toteutet-

tava työajanseuranta. Lisäksi kotisovellukseen on hyvä lisätä käyttäjäkohtaisesti tarvittavat lisätiedot, kuten tapahtumat, ilmoitukset ja käyttäjälle lisätyt tehtävät. Samoin on syytä vielä selvittää jatkossa Realm-tietokannan kanssa esiintyneet satunnaiset ongelmat sekä selvittämättä jäänyt satunnaisesti esiintynyt reititysongelma. Jatkossa on myös syytä vielä arvioida uudelleen micro frontend -sovellusten jakojen ja rajausten toimivuutta tulevaisuudessa lisättävät toiminnot huomioiden.

7 PÄÄTÄNTÖ

Tämän opinnäytetyön aikana suunniteltiin ja toteutettiin micro frontend -arkkitehtuurilla toteutettava tavaran- ja kalustonhallintasovellus iOS ja Android -laitteille. Sovellukseen toteutettavat hallinta- ja seurantatoiminnot toteutettiin olemassa olevan web-sovelluspohjaisen toiminnanohjausjärjestelmän päätoimintojen tueksi. Sovellus integroitiin olemassa olevaan järjestelmään REST API -rajapinnan avulla sekä Realm-tietokannan App Sync -toiminnon avulla, jolla voitiin varmistaa tietojen synkronoituminen eri laitteiden välillä, sekä samalla varmistaa sovelluksen toimiminen myös offline-tilassa. Micro frontend -arkkitehtuuri toteutettiin Webpack 5:n module federation -laajennuksen avulla.

Toimeksianto oli kokonaisuudessaan hyvin mielenkiintoinen ja sopi erittäin hyvin tietojenkäsittelyn tradenomin tutkintoon kuuluvien opintojen sisältöön sekä opinnäytetyön työvaiheiden kuin myös käytettävien teknologioidenkin osalta. Toimeksiantajan web-pohjaisen toiminnanohjausjärjestelmän suunnittelussa ja kehittämisessä mukana oleminen on tehnyt niin järjestelmässä käytetyn rakenteen ja tekniikat kuin asiakkaan ja toimeksiantajan tavoitteetkin hyvin tutuiksi, mikä antoi hyvän pohjan etenemiselle. Aihe tuki hyvin myös henkilökohtaista kiinnostusta ja halua oppia mobiilisovellusten kehittämistä ja kokonaisvaltaista monialustaista järjestelmäarkkitehtuuria syvemmin. Lisäksi työssä oli mahdollista päästä osaltaan pilotoimaan mobiilisovelluksille uusia arkkitehtonisia ratkaisumalleja ja tutkia niiden hyötyjä ja haasteita, mikä voi olla merkityksellistä sekä alan että toimeksiantajan näkökulmasta.

Haasteena työn suorittamisessa oli koko projektin läpi erityisesti dokumentaation vähyys ja heikko laatu. Teknologioista oli myös verrattain vähän kolman-

sien osapuolien käyttökokemuksia ja julkaisuja, joten tiedon hakeminen ongelmatilanteissa oli ajoittain vaikeaa ja ongelmia jouduttiin ratkomaan osin sokkona yritys ja erehdys -taktiikalla. Haasteellisuutta lisäsi vielä henkilökohtainen kokemuksen puute mobiilisovellusten kehittämisestä ja työssä käytetyistä teknologioista. Toisaalta aiemmin kerrytetty kokemus micro frontend -arkkitehtuurista ja Webpackin konfiguroinnista web-sovelluksissa auttoivat hyvin pitkälle työn kokonaisuuden hahmottamisessa ja ongelmakohtien ratkaisemisessa. Kokonaisuutena toteutus onnistui sujuvasti ja antoi riittävästi eväitä oman osaamisen kehittämiseen jatkossakin.

LÄHTEET

About Re.Pack. 2021. WWW-dokumentti. Päivitetty 13.7.2022. Saatavissa: https://re-pack.netlify.app/docs/about/ [viitattu 25.9.2022].

Alekseev, I. 2022. Increase app responsiveness with MongoDB Realm mobile database and AWS Wavelength. AWS Open Source Blog. Blogi. Saatavissa: https://aws.amazon.com/blogs/opensource/increase-app-responsiveness-with-mongodb-realm-mobile-database-and-aws-wavelength/ [viitattu 28.1.2023].

Bort, J. 2015. A Startup launched a year ago by these former Nokia engineers is already powering half a billion smartphones. Business Insider. WWW-dokumentti. Saatavissa: https://finance.yahoo.com/news/startup-launched-ago-former-nokia-190000684.html [viitattu 28.1.2023].

Budziński, M. 2022. What Is React Native? Complex Guide for 2022. Blogi. Saatavissa: https://www.netguru.com/glossary/react-native#benefits-of-react-native [viitattu 16.8.2022].

Chellammal, S., Gopinath, G. & Pethuru, R. 2020. Essentials of Microservices Architecture – Paradigms, Applications and Techniques. Boca Raton: Taylor & Francis Group, LLC.

Davis, A. 2021. Bootstrapping Microservices with Docker, Kubernetes and Terraform. Shelter Island, NY: Manning Publications Co.

Geers, M. 2020. Micro frontends in Action. Shelter Island, NY: Manning Publications Co.

Horowitz, E. 2019. MongoDB and Realm make it easy to work with data, together. MongoDB. Blogi. Päivitetty 25.5.2019. Saatavissa: https://www.mongodb.com/blog/post/mongodb-and-realm-make-it-easy-to-work-with-data-together [viitattu 28.1.2023].

Jackson, C. 2019. Micro Frontends. Blogi. Saatavissa: https://martinfowler.com/articles/micro-frontends.html [viitattu 24.8.2022].

Metro. 2018. Facebook. WWW-dokumentti. Päivitetty 5.8.2019. Saatavissa: https://facebook.github.io/metro/docs/concepts [viitattu 25.9.2022].

Mezzalira, L. 2022. Building Micro-Frontends. Scaling teams and projects, empowering developers. 1. painos. Sebastopol, CA: O'Reilly Media, Inc.

Micro frontends. 2016. Thoughtworks Technology Radar. WWW-dokumentti. Päivitetty 19.3.2020. Saatavissa: https://www.thoughtworks.com/radar/techniques/micro-frontends [viitattu 25.8.2022].

Micro frontends for mobile. 2020. Thoughtworks Technology Radar. WWW-dokumentti. Päivitetty 27.10.2021. Saatavissa: https://www.thoughtworks.com/radar/techniques/micro-frontends-for-mobile [viitattu 25.8.2022].

Micro Frontends for Mobile. 2022. Zddhub.com. Blogi. Saatavissa: https://zddhub.com/article/2022/05/25/micro-frontends-for-mobile.html [viitattu 5.2.2023].

MongoDB docs. s.a. Introduction to Realm. WWW-dokumentti. Saatavissa: https://www.mongodb.com/docs/realm/introduction/ [viitattu 28.1.2023].

Netkow, M. s.a. Micro Frontends for Mobile with Ionic Portals. Blogi. Saatavissa: https://ionic.io/resources/articles/micro-frontends-for-mobile-with-ionic-portals#h-from-microservices-to-micro-frontends [viitattu 23.8.2022].

Newman, S. 2021. Building Microservices. 2. painos. Sebastopol, CA: O'Reilly Media, Inc.

Sharma, M. 2022. Full Stack Development with MongoDB. BPB Publications.

Trysła, P. 2021. Implementing Code Splitting in React Native with Re.Pack. Blogi. Päivitetty 7.9.2021. Saatavissa: https://medium.com/call-stack/implementing-code-splitting-in-react-native-with-re-pack-6398881ade6b [viitattu 16.10.2023].

Webpack. s.a. WWW-dokumentti. Saatavissa: https://webpack.js.org/concepts/ [viitattu 12.9.2022].

Wieruch, R. 2018. The Road to learn React. Berlin: Robin Wieruch.

Zammetti, F. 2020. Modern Full-Stack Development. California: Apress Media LLC.