Arttu Saarimaa

Sovelluskehitys micro frontend -arkkitehtuurilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

8.10.2021

Tiivistelmä

Tekijä:	Arttu Saarimaa
Otsikko:	Sovelluskehitys micro frontend -arkkitehtuurilla
Sivumäärä:	34 sivua
Aika:	8.10.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mediatekniikka
Ohjaaja:	Lehtori Ilkka Kylmäniemi

Insinöörityössä oli tarkoituksena muuttaa asiakasyrityksen olemassa oleva sovellus käyttämään micro frontend -sovellusarkkitehtuuria sekä tuoda uusia micro frontend -toimintoja sovellukseen. Sovellusarkkitehtuuri valittiin sovelluksen kehityksessä työskentelevien tiimien nopean vaihtumisen vuoksi. Vanha monoliittinen arkkitehtuuri vei lyhyen kehitysjakson omaavilta tiimeiltä aikaa, sillä tiimien oli opiskeltava koko sovelluksen toiminta, ennen kuin se pääsi sovelluskehitysvaiheeseen.

Tavoitteena oli suunnitella sovellukselle uusi rakenne uutta sovellusarkkitehtuuria käyttäen ja muuttaa sovellus suunniteltuun muotoon. Sovelluksen uudelleenrakentamisen jälkeen sovellukseen lisättäisiin testinä uusia toimintoja micro frontendeinä, joita vanhassa sovelluksessa ei ollut.

Uuden sovellusarkkitehtuurin toteuttamiseen käytettiin Webpackin Module Federationia, jonka implementointi oli konversion alkuvaiheissa hyvin yksinkertaista vähäisestä dokumentaatiosta huolimatta. Konversion edetessä Redux-toimintojen implementointiin dokumentaation vähyys jarrutti sovelluksen kehitystä huomattavasti.

Uusien toimintojen lisäys sovellukseen ei onnistunut, koska toimintoja tekevien tiimien aika ei riittänyt tuottamaan lisäyskelpoisia toimintoja ennen määräaikaa. Uusien toimintojen sijaan sovelluksen rinnalle kehitettiin uusi sovellus toimimaan admin-sovelluksena vanhalle sovellukselle.

Työn lopputuloksena saatiin aikaan micro frontend -arkkitehtuuria käyttävä monoliittisesta sovelluksesta konvertoitu progressiivinen verkkosovellus. Sovelluskehityksen ohessa monien vanhan sovelluksen toimintojen suorituskykyä parannettiin koodia refaktoroimalla, ja uuden admin-sovelluksen pohja kehitettiin uutta sovellusarkkitehtuuria käyttäen.

Avainsanat: micro frontend, sovellusarkkitehtuuri

Abstract

Author:	Arttu Saarimaa
Title:	Software development using micro frontend architecture
Number of Pages:	34 pages
Date:	8 October 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Media Technology
Supervisors:	Ilkka Kylmäniemi, Senior Lecturer

The aim of the final year project was to transform the customer's existing application to use micro frontend architecture, and to bring new micro frontend functionalities to the application. The architecture was chosen because of the fast changing of teams that work on the application. The old monolithic architecture caused development delays, because the teams already short on time had to learn the inner workings of the application before they could start the development process.

The goal of the project was to design the new structure of the application using the new architecture, and to transform the application to the new structure. After the rebuilding of the application there would be the addition of new functionalities as a test as micro frontends.

To create the new architecture, Webpack's Module Federation was used as a base for the new technology, implementation of which was rather simple at the beginning of the conversion process regardless of lackluster documentation. As the conversion process advanced to adding Redux functionalities though, the lack of documentation caused the development to slow down significantly.

The addition of new functionalities to the application had to be disregarded, as the teams building the new functionalities did not have the time to make them usable before the deadline. Instead of new functionalities, a new application was developed to be used alongside the transformed application as an admin client.

As a result of the study an application using the micro frontend architecture was created by transforming the old monolithic progressive web application. During the development process, many of the existing functionalities' performance was improved by refactoring the old code, and the base for the new admin client application was created using the new architecture.

Keywords: Micro frontend, software architecture

Sisällys

Lyhenteet

PWA: Progressive web application. Natiivin ja verkkopohjaisen sovelluksen välimuoto, joka on asennettavissa mutta toimii myös selaimessa.

MFE: Micro frontend. Micro frontend -sovellusarkkitehtuuria käyttävässä sovelluksessa käytettävä autonominen selainpuolen komponentti.

MFEA: Micro frontend -arkkitehtuuri. Mikropalveluarkkitehtuuriin perustuva sovellusarkkitehtuurityyli, joka tuo edellä mainittuun tyyliin myös selainpuolen toiminnallisuuden.

SCS: Self-contained systems. Mikropalveluarkkitehtuuriin perustuva sovellusarkkitehtuurimalli toisistaan riippumattomien sivukokonaisuuksien tekemiseen.

FIVS: Frontend integration for verticalized systems. Mikropalveluarkkitehtuuriin perustuva sovellusarkkitehtuurimalli sivun sisäisten autonomisten toimintojen kehittämiseen.

SPA: Single page application. Sovellus, jonka kaikki sisältä on yhden jatkuvasti päivittyvän sivun alla usean navigoitavan sivun sijaan.

1 Johdanto

Modernien sovellusten palvelinpuolen sovellusarkkitehtuuri on ottanut mikropalveluiden yleistyttyä askeleen modulaarisempaan suuntaan ja pois monoliittisesta sovellusarkkitehtuurista. Tällainen modulaarisuus ei kuitenkaan ole vielä yleistynyt sovellusten selainpuolella useista yrityksistä huolimatta, mutta micro frontendit saattavat olla ratkaisu tähän.

Micro frontend -arkkitehtuuri on mikropalveluarkkitehtuuria mukaileva sovellusarkkitehtuurityyli. Arkkitehtuuri yrittää tuoda selainpuolelle mikropalveluiden hyödyllisiä toimintoja, kuten jatkuva käyttöönotto ja komponenttien autonomisuus sovelluksen ylläpitämisen ja sen osien käyttöönoton helpottamiseksi. Arkkitehtuurin tiimijako myös vähentää yksittäisen kehittäjän tarvetta opiskella sovellusta projektiin liittyessä, sillä komponenttien autonomisuuden vuoksi muut sovelluksen osat eivät vaikuta uuteen komponenttiin. Poikkeuksena tähän on isäntäsovellus, johon komponentit tuodaan ja joka on täten jaettu kaikkien komponenttien kesken.

Insinöörityössä oli tarkoituksena muuttaa vanha monoliittinen progressiivinen verkkosovellus (PWA) käyttämään micro frontend -arkkitehtuuria. Uusi arkkitehtuuri muuttaisi vanhan sovelluksen yksittäiset sivut micro frontendeiksi, mikä mahdollistaisi yksittäisten sivujen käyttöönoton tai käytöstä poistamisen ilman koko sovelluksen sammuttamista. Jos aika riittäisi, oli myös suunnitelmissa rakentaa toinen sovellus alusta asti uudella arkkitehtuurilla toimimaan vanhan sovelluksen kanssa.

Insinöörityö tehtiin Nokialle, ja työn tavoite oli helpottaa sovelluksen jatkokehitystä. Sovellusta oli tähän asti ollut kehittämässä useita tiimejä, ja sovellus oli laajentunut huomattavasti alkuperäisestä. Tiimien jatkuvan vaihtumisen vuoksi uusien kehittävien tiimien olisi opiskeltava sovelluksen toiminta ja sovelluksen osien väliset ajonaikaiset toiminnot. Tämä prosessi pitkittyy sovelluksen laajentuessa, joten micro frontendit nähtiin mahdollisuutena vähentää tätä

opiskelutaakkaa. Uudella arkkitehtuurilla toimittaessa tiimien tarvitsee opiskella vain isäntäsovelluksen toiminta sekä micro frontendin rakentamiseen tarvittavat tiedot.

2 Micro frontend -arkkitehtuuri

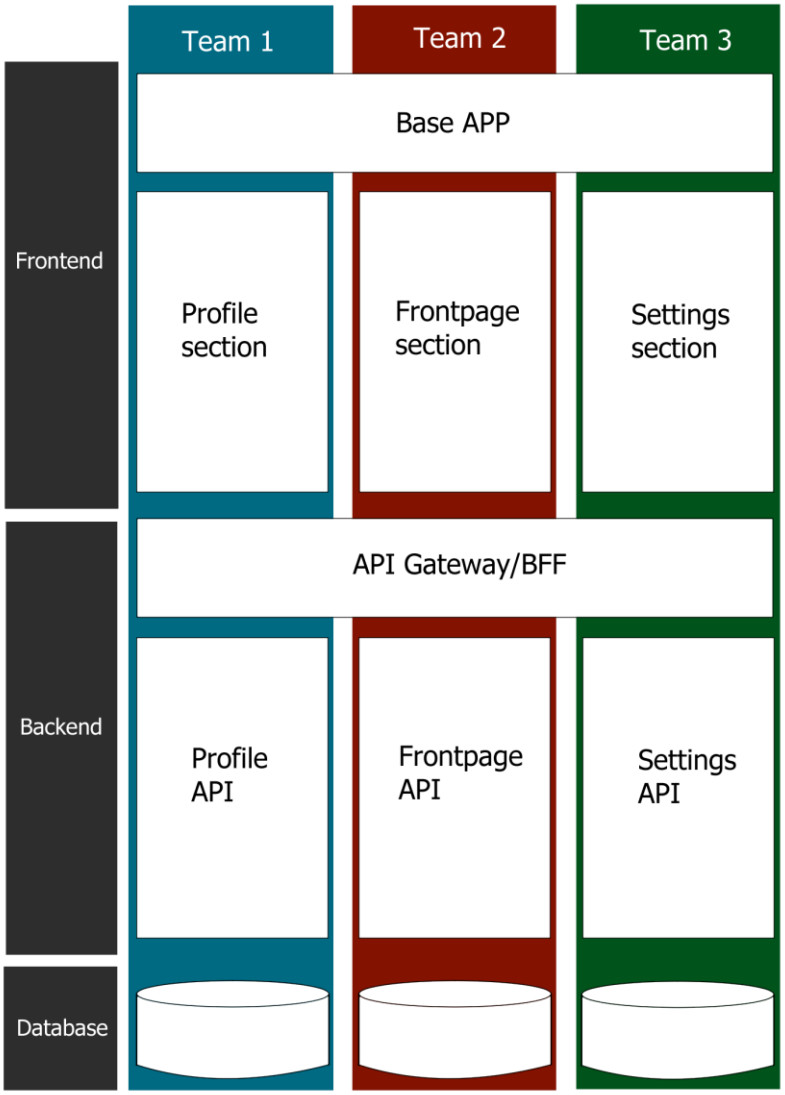
2.1 Määritelmä

Micro frontend -arkkitehtuuri (MFEA) on mikropalveluarkkitehtuurin inspiroima sovellusarkkitehtuurityyli [1]. Sovellusarkkitehtuuri määrittelee sovelluksen sisäisten osien kokonaisuuden rakenteen ja organisoinnin [2]. MFEA pyrkii luomaan hajautetun rakenteen, mikä tarkoittaa, että sovellus koostuu itsenäisistä toisistaan riippumattomista komponenteista, jotka sisältävät sekä selainpuolen että palvelinpuolen toiminnot.

MFEA:n määritelmä on hieman epäselvä, sillä mikropalveluarkkitehtuuri on inspiroinut useita selainpuolen sisältäviä sovellusarkkitehtuurityylejä, joista MFEA on vain yksi uusimmista variaatioista. MFEA:n erottaa muista vastaavanlaisista tyyleistä sen joustavuus sovelluksen komponenttien laajuuden suhteen.

2.2 Rakenne

MFEA:ta käyttäen rakennettu sovellus koostuu autonomisista modulaarisista komponenteista, jotka tuodaan isäntäsovellukseen micro frontendeinä (MFE). Yksittäinen MFE koostuu selainpuolen komponentista, jolla optimaalisessa tilanteessa on oma palvelinsovellus ja tietokanta, joita se ei jaa muiden sovellusten kanssa. Ainoat asiat, joita MFE-komponenttien tulisi mahdollisesti jakaa keskenään, ovat isäntäsovelluksessa luotu data, sillä isäntäsovellus on ainoa osa, jonka tulee aina olla toiminnallinen, sekä API Gateway, joka ohjaa selainpuolen pyynnöt palvelinsovellukseen (kuva 1).



Kuva 1. Yksinkertainen micro frontend -sovellusarkkitehtuurimalli, jossa selainpuoli on jaettu micro frontendeiksi, palvelinpuoli on jaettu mikropalveluiksi ja jokaisella toiminnolla on oma tietokantansa.

MFEA on vertikaalinen sovellusarkkitehtuurimalli, eli sovellusta kehittävä tiimi rakentuu selainpuolen ja palvelinpuolen kehittäjistä sekä tietokantakehittäjistä. Osat, joihin sovellus jaetaan, vaihtelevat projektin mukaan, mutta ne voivat esimerkiksi olla kokonaisia verkkosivuja, jotka yhdistetään toisiinsa sovelluksen sisäisillä poluilla, tai ne voivat olla yksittäisiä toimintoja, jotka järjestellään yhdelle verkkosivunäkymälle. MFEA:ssa MFE:t lisätään sovelluspohjaan eli isäntäsovellukseen, joka yhdistää erilliset MFE:t kokonaisuudeksi. Isäntäsovellus toimii myös pohjana, jolla näyttää joko sivuja tai yksittäisiä toimintoja pitäen samalla

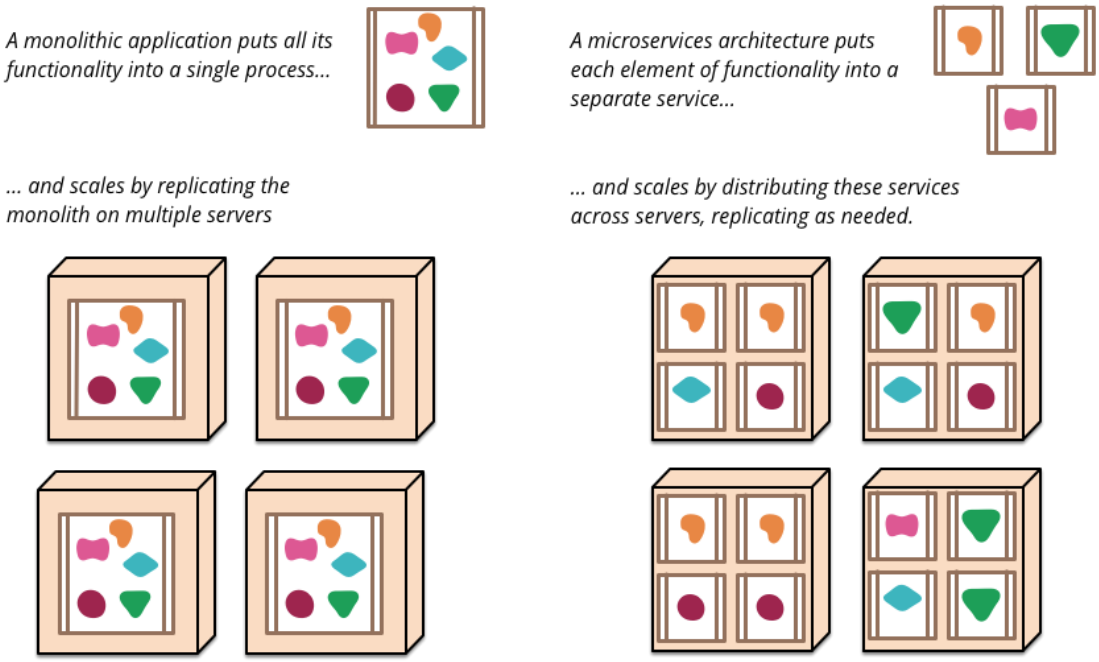
sivun tyylin yhtenäisenä. Monimutkaisemmissa projektirakenteissa MFE:t saattavat sisältää pienempiä MFE:itä ja toimia itse isäntäsovelluksina, luoden tietynlaisen pyramidimaisen sovellushierarkian.

2.3 Micro frontendit ennen micro frontendejä

Micro frontendit eivät ole ensimmäinen teknologia, joka on yrittänyt parantaa laajojen sovellusten skaalautuvuutta ja ylläpitoa, tai edes ensimmäinen arkkitehtuurityyli, joka on yrittänyt tuoda tätä toiminnallisuutta selainpuolelle. Aikaisempia teknologioita tutkimalla on helppo huomata, mistä micro frontendit ovat saaneet mallia toiminnallisuuksiinsa. MFEA:ta edeltävän mikropalveluarkkitehtuurin periaatteita suunniteltiin jo 1990-luvulla [4], mutta ne saivat nimensä vasta vuonna 2011, jolloin esimerkiksi Netflix ja Amazon olivat jo alkaneet testata arkkitehtuurityyliä [3].

Mikropalveluarkkitehtuuri perustuu monimutkaisten sovellusten rakentamiselle pienemmistä yksinkertaisemmista sovelluksista, jotka kommunikoivat toistensa kanssa ohjelmointirajapintojen kautta [3]. Toimintojen eristäminen mahdollistaa sovelluksen pitämisen helposti ylläpidettävänä ja skaalattavana [4] ja sovelluksen osien sulkemisen ja käynnistämisen ilman muun sovelluksen tai sen osien sulkemista [5]. Tämä tekee uusien tai päivitettyjen toimintojen käyttöönotosta helppoa, ja se on mikropalveluarkkitehtuurin suurin vahvuus ja syy sen suosiolle.

Mikropalveluarkkitehtuuri eroaa monoliittisista sovelluksista eniten juuri sen skaalautuvuuden vuoksi (kuva 2). Siinä missä monoliittisen sovelluksen skaalaaminen vaatii koko sovelluksen skaalaamista, voivat mikropalvelut skaalata vain haluttua osaa sovelluksesta lisäämällä tarvittavia mikropalveluita [6].



Kuva 2. Monoliittisen sovelluksen ja mikropalvelun skaalaaminen. Kuvassa näkyy, kuinka monoliittisessa sovelluksessa kaikki toiminnot on skaalattava, kun taas mikropalveluarkkitehtuurissa voidaan skaalata vain haluttuja toimintoja. [6.]

Self-contained systems (SCS) ovat askeleen lähempänä micro frontend -arkkitehtuuria kuin puhdas mikropalveluarkkitehtuuri, sillä SCS pyrkii palvelinpuolen lisäksi eristämään myös sovelluksen selainpuolen toiminnot toisistaan.

SCS pyrkii sovellukseen, jonka osat pystyvät toimimaan toisistaan riippumatta, mutta kykenevät silti kommunikoimaan toistensa kanssa. Kommunikoinnin tulisi olla epäsynkronista aina kun mahdollista, sillä tämä pitää sovelluksen osat autonomisina ja estää toimintojen kaatumisen muiden toimintojen ollessa pois käytöstä [7]. SCS-malli myös mahdollistaa eri ohjelmointikielien ja runkojen käytön samaan tapaan kuin micro frontendit [8]. SCS-malli eroaa MFEA-mallista, sillä SCS sulkee pois yhden sivun sovellus- eli single page application (SPA) -ratkaisut, koska SCS-komponentit eivät saa jakaa käyttöliittymää toistensa kanssa [7].

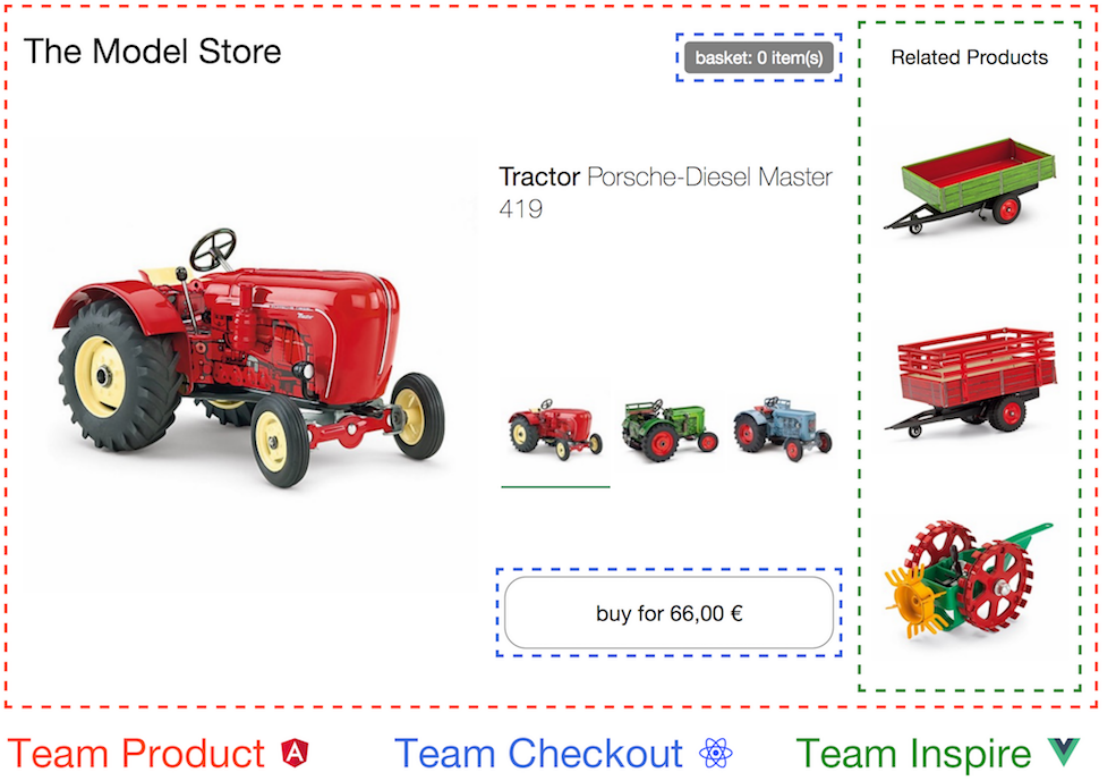
Frontend integration for verticalized systems (FIVS) on mikropalveluarkkitehtuuriin perustuva sovellusarkkitehtuurityyli, joka muistuttaa yhden sivun micro

frontend -sovellusta. FIVS jakaa mikropalveluarkkitehtuuriin perustuvien arkkitehtuurimallien yleisimmät ominaisuudet, eli skaalautuvuuden, ylläpidon helppouden ja komponenttien autonomisen toiminnan. FIVS:in erottaa esimerkiksi SCS-mallista se, että siinä missä yksittäinen SCS-komponentti on koko sivu, yksittäinen FIVS-komponentti eli vertikaali on sivun sisäinen toiminto. Erona micro frontendeihin on se, että yksittäinen MFE voi olla SPA, mikä ei onnistu FIVS:n rakenteella.

2.4 Erot monoliittiseen sovellukseen

MFEA:ssa sovellusta rakentavat tiimit jaetaan vertikaalisesti eli toimintopohjaisesti. Yksi tiimi on vastuussa yhdestä sivusta tai toiminnosta (kuva 3, s. 7). Tämän lähestymistavan hyöty verrattuna yleisempään horisontaaliseen tiimirakenteeseen on se, että tiimit koostuvat sekä selainpuolen, palvelinpuolen että tietokantakehittäjistä. Tämän vuoksi tiimin kehittäjät saavat paremman käsityksen oman komponenttinsa kaikista osista ja tarpeista, kun taas horisontaalisessa rakenteessa esimerkiksi selainpuolen kehittäjät ovat pääasiassa yhteydessä vain muihin selainpuolen kehittäjiin. Tämä on myös hyödyllinen tiimirakenne tilanteissa, joissa yritys esimerkiksi palkkaa ulkopuolisia sovelluskehittäjiä luomaan uusia toimintoja sovellukseensa. Vertikaalinen rakenne ei pakota ulkopuolisia kehittäjiä opettelemaan koko sovelluksen selainpuolta tai palvelinsovellusta yhteensopivuuden takaamiseksi, vaan he voivat keskittyä omaan toimintoonsa.

MFEA pyrkii olemaan agnostinen teknologian suhteen, mikä tarkoittaa, että arkkitehtuurissa pyritään käyttämään oikeaa työkalua oikeassa tilanteessa [9]. Tämä tarkoittaa micro frontendien kontekstissa kykyä käyttää eri JavaScript-pohjaisia ohjelmointikirjastoja ja runkoja sovelluksen eri osien tekemiseen. Eri JavaScript-pohjaisilla kirjastoilla tehdyt komponentit pystyvät silti kommunikoimaan keskenään MFE-teknologian ansiosta, eli eri tiimien tekemien komponenttien ei tarvitse olla tehty samalla rungolla edes yksittäisen sivun sisällä.



Kuva 3. Esimerkkiprojektissa luodun traktorikaupan eri JavaScript-kirjastoilla tehdyt MFE:t, jotka pystyvät kommunikoimaan keskenään [1].

Teknologia-agnostisuuden vuoksi MFEA:n on oltava modulaarinen ja eristettävä sovelluksen sisäiset osat toisistaan. Modulaarisuus antaa kehittäville tiimeille vapauden käyttää haluamaansa tai parhaaksi näkemäänsä teknologiaa omaan osuuteensa ilman, että niiden tarvitsee huolehtia yhteensopivuusongelmista. Tämän vuoksi MFE on myös hyvä vaihtoehto tilanteissa, joissa vanhaan sovellukseen, jota ei enää aktiivistesti ylläpidetä, halutaan lisätä toimintoja [10].

Sovelluksen osien autonomisuudesta on hyötyä sovelluksen ylläpidon kannalta, sillä toisistaan riippumattomuus mahdollistaa sovelluksen jatkuvan toiminnon jopa yksittäisten komponenttien pettäessä. Autonomia myös mahdollistaa sovellukseen uusien toimintojen jatkuvan käyttöönoton sekä vanhojen toimintojen helpon ylläpitämisen ja päivittämisen. Monoliittisessa sovelluksessa koko sovellus olisi otettava pois käytöstä aina toimintoa päivitettäessä tai uutta lisättäessä, sillä sivut jakavat ajonaikaisia toimintoja ja niitä ajetaan samasta lähteestä.

MFEA sisältää kuitenkin myös haittoja, jotka on otettava huomioon arkkitehtuurimallin valitsemisessa. MFEA tuo sovellukseen joustavuutta käytettyjen teknologioiden ja käyttöönoton suhteen, mutta nämä hyödyt tulevat ohjelmistorakenteen yksinkertaisuuden kustannuksella [11]. Monoliittista ohjelmistorakennetta on huomattavasti helpompi ymmärtää kuin MFEA:ta, kun projekti on laaja, ja huolimattomasti suoritettu MFEA voi johtaa jopa melkein monoliittiseen sovellukseen, jos komponentteja ei pidetä autonomisina. Tämän monimutkaisen sovellusrakenteen vuoksi MFEA ei ole myöskään tarpeellinen sovelluksiin, joiden tiedetään pysyvän pieninä ja jotka eivät täten tarvitse MFEA:n mahdollistamaa skaalautuvuutta. MFEA:n vertikaalisuus saattaa myös koitua ongelmaksi, jos sovelluksen rakennukseen ei ole kohdennettu tarpeeksi työntekijöitä. Tämä ei välttämättä ole ongelma suurille yhtiöille, mutta pienemmissä projekteissa saattaa olla vaikeaa muodostaa useita tiimejä, jotka koostuvat sekä selainpuolen että palvelinpuolen osaajista [11].

3 Sovelluskehityksen työkalut

Insinöörityön sovellus toteutettiin vanhan sovelluksen tapaan Reactilla, mutta työ vaati myös useita muita kirjastoja haluttujen toimintojen kehittämisen mahdollistamiseksi. Tässä luvussa selitetään keskeisimpien sovelluksen kehitykseen käytettyjen kirjastojen ja työkalujen toiminta ja tarkoitus sovelluksessa.

3.1 Material UI -kirjasto

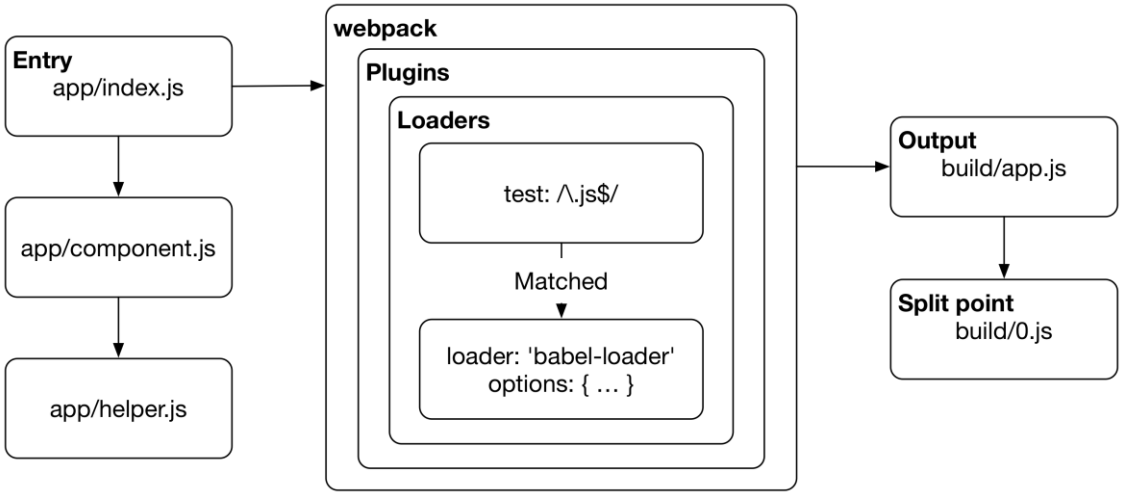
Material UI on Googlen vuonna 2014 julkaisema Material Designiin perustuva kirjasto. Material UI kehitettiin helpottamaan modernin näköisen minimalistisen käyttöliittymän luomista käyttämällä paremmin sivun vapaata tilaa, jolloin mobiili ja verkkosovelluksen käyttö saadaan pidettyä yhtenäisempänä. [12.]

Material UI antaa sovelluskehittäjälle mahdollisuuden luoda modernin näköistä käyttöliittymää tuomalla kirjaston laajasta elementtivalikoimasta tarvitsemansa valmiit elementit ja kuvakkeet eli material-komponentit. Jos oikeanlaisen sovelluksen sisällä laajasti käytettävän Material-komponentin puuttuu, Material UI

antaa mahdollisuuden myös luoda omia komponentteja käytettäväksi sovelluksen sisällä. Material-komponentit helpottavat kehitystä, sillä sovelluskehittäjän ei tarvitse itse luoda modernissa web-kehityksessä usein käytettyjä elementtejä tyylittelemällä niitä itse vakioelementeistä useaan kertaan, tai parhaassa tapauksessa ollenkaan. Tämä myös selventää tyylitiedostojen rakennetta ja helpottaa niiden lukemista, sillä tyylejä tarvitaan lähinnä vain komponenttien hienosäätöön.

3.2 Webpack-kirjasto

Webpack on JavaScript-pohjainen paketointityökalu ja toimintojen ajaja. Webpack ottaa sille annetut moduulit ja paketoi ne sovelluksen käytettäväksi. se luo paketointivaiheessa tiedostojen vaatimista muiden tiedostojen osista graafin, jonka avulla se luo pienempään kokoon paketoidun sovelluksen (kuva 4). [13.]



Kuva 4. Webpackin toimintaprosessi. Vasemmalla Entry-kohdassa webpackille annettava sovellus, jonka alla ovat sen riippuvuudet, ja oikealla ulos tuleva paketti. [13.]

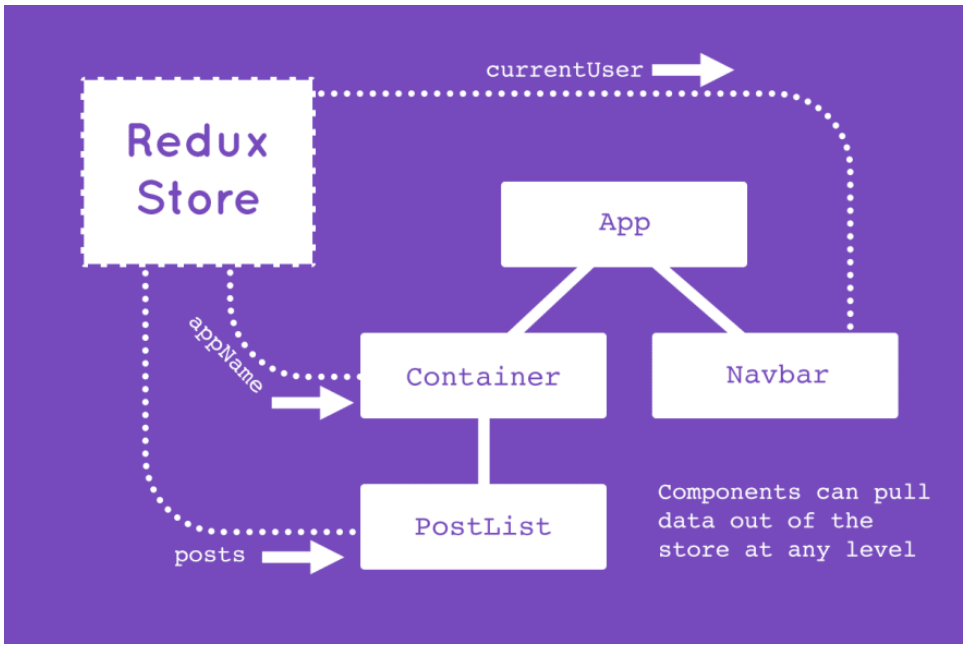
Webpack antaa sovelluskehittäjälle paljon vapauksia pakettien muodostamisen suhteen, ja tukee myös liitännäisiä. Sen muokattavuus tulee sen intuitiivisuuden kustannuksella, sillä Webpack ei ole kovin aloittelijaystävällinen ja vaatii paljon opiskelua.

Moduulien paketoijia (module bundler) käytetään koodin rakenteen parantamiseksi suurissa projekteissa, sillä koodin lisääntyessä sitä on usein jaettava useisiin tiedostoihin luettavuuden ja ylläpidon helpottamiseksi. Koodin jakaminen useisiin tiedostoihin lisää importtien määrää ja vaatii html-tiedoston puolella script-tagien järjestelyä, sillä tiedostojen on oltava siinä järjestyksessä, missä ne ajetaan, mikä käy vaikeaksi tagien lisääntyessä [14]. Tämän järjestyksen tärkeys vaikeuttaa myös sovelluksen laajentamista, sillä sovelluskehittäjä joutuu nyt ajattelemaan, mihin uudet toiminnot on lisättävä tiedostohierarkiassa, jotta uusi toiminto suoritetaan oikeaan aikaan.

Tiedostorakenteen laajentuessa sovelluksen toimintoja oli muutettava taas kehityksen ja ylläpidon helpottamiseksi, ja tämän mahdollistamiseksi moduulien paketoijat kehitettiin. Paketoija ottaa kaikki sille annetut tiedostot ja tekee niistä yhden suuren tiedoston, mikä poistaa usean script-tagin tarpeen. Paketoija tarkistaa myös koodien riippuvuudet ja joko tuo moduulille sen tarvitseman koodin toisesta moduulista, tai jos toista moduulia ei ole vielä ladattu, laittaa ensimmäisen moduulin odottamaan koodin suorittamista. Tämä ratkaisee tagien järjestelyongelman. [14.]

3.3 Redux-kirjasto

Redux on JavaScript-pohjainen kirjasto sovelluksen tilanhallintaan. Tilat (state) ovat sovelluksen sisäisiä muuttujia, joita hallinnoidaan Redux storella. Redux store on näiden muuttujien varasto, josta eri osat sovellusta voivat hakea niitä tarvittaessa. Tämän tilan varastoinnin vuoksi sovellus voi välttää tarpeetonta tiedon hakemista, sillä sovelluksen osat voivat hakea tarvitsemansa tiedot varastosta suoraan datan lähteestä hakemisen sijaan (kuva 5). Redux mahdollistaa myös datan helpon jakamisen samalla tasolla olevien komponenttien kesken.



Kuva 5. Reduxin toiminta sovelluksen eri tasoilla [15].

Redux oli lisätty projektiin alun perin työkaluna service workerien päivittämiseen. Service workerit ovat skriptejä, jotka toimivat selaimen taustalla erillään verkkosivusta. [16.] Niitä käytettiin sovelluksessa PWA-toimintojen mahdollistamiseksi. Redux otettiin myöhemmin laajempaan käyttöön etusivun piensovellusten tallentamisen sekä muiden saman tason toimintojen kommunikoinnin mahdollistamiseksi.

4 Arkkitehtuurin suunnittelu ja prototyypit

4.1 Sovelluksen kehityksen ennuste

Työn suunnitteluvaiheessa oli vaikea arvioida työn vaikeutta, sillä MFE-teknologia on hyvin uusi konsepti eikä sen implementaatiosta ole juuri esimerkkejä tai dokumentaatiota suuressa mittakaavassa. Suurimmat huolet olivat MFE-teknologian opiskelun vaikeus vähillä materiaaleilla sekä PWA-toiminnallisuuksien lisääminen sovellukseen, jonka osat ovat MFE-komponentteja.

Teknologian opiskelu vähillä materiaaleilla oli tietysti itsessään oletettu ongelma työn toteutuksen kannalta, mutta huolta opiskelun viemästä ajasta lisäsi, että teknologia tulisi pystyä lisäämään uusiin komponentteihin neljän kuukauden kuluttua. Sovellukseen kehitettiin insinöörityön ohella myös uusia toimintoja, jotka tulisi pystyä muuttamaan MFE-komponenteiksi niiden valmistuttua. Vanha sovellus oli siis muutettava MFEA:han olemassa olevine toimintoineen noin neljän kuukauden aikana.

Syy PWA-toiminnallisuuksien implementoinnista huolehtimiseen oli niiden toimintaan saamisen vaikeus sovelluksessa ennen MFE-teknologian lisäämistä sekä service workerien tuoma rasite sovellukselle. Työn alkaessa ennen tarkempaa MFE-teknologian opiskelua ei ollut varmaa, tuodaanko MFE:stä vain tarvittava koodi isäntäsovellukseen ajettavaksi vai paljastaako MFE sovelluksesta tuotavan toiminnan ikään kuin heijastamalla sen isäntäsovellukselle. Jos sovelluksen koodi suoritettaisiin isäntäsovelluksessa sellaisenaan, ei PWA:n kanssa olisi suurempia ongelmia, sillä silloin yksi service worker onnistuisi varastoimaan kaiken tarvittavan tiedon. Jos sovellus kuitenkin heijastettaisiin, ei olisi ollut varmaa tietoa siitä, kuinka isäntäsovelluksen service worker toimisi, jolloin komponentteihin tulisi lisätä omat service workerit, mikä madaltaisi sovelluksen toimintanopeutta ja tekisi sovelluksesta entistä raskaamman.

4.2 Tiedostorakenteen suunnittelu

MFEA:n tuottamiseen vaadittu tiedostorakenne on erittäin joustava, ja se, kuinka hankalaa vanhan koodin muuttaminen tähän rakenteeseen on, vaihtelee riippuen siitä, kuinka paljon vaivaa muutoksia tekevä tiimi on valmis näkemään.

Koska yksittäisen MFE:n koolle ei ole varsinaisesti ylä- tai alarajaa, on mahdollista tehdä vanhasta sovelluksesta itsestään yksi suuri MFE, jos tiimillä ei ole aikaa jakaa sovellusta osiin. Tällöin sovellukseen lisättäisiin isäntäsovellus ja vanhaan sovellukseen lisättäisiin tarpeelliset osat, jolla se voidaan näyttää isäntäsovelluksessa. Uudet osat lisättäisiin todennäköisesti paljon pienempinä ja helpommin ylläpidettävinä MFE:inä. Tämä tiedostorakenne pidettiin projektissa

mahdollisuutena sen varalta, että kunnolliselle uudelleenjärjestelylle ei olisi aikaa.

Toinen tapa muodostaa projektin tiedostorakenne on sivupohjaisesti, jos sovellus koostuu useista sivuista. Olemassa olevasta sovelluksesta erotetaan eri sivuilla käytettävät osat omiin kansioihinsa, jotka siirretään isäntäsovelluksen ulkopuolelle ja muutetaan omiksi sovelluksikseen. Tämä uudelleenjärjestelytapa voi olla huomattavasti työläämpi riippuen siitä, kuinka paljon ajossa jaettavaa tietoa sivuilla on, sillä yksittäisistä sivuista on tehtävä autonomisia sovelluksia. Tämä lähestymistapa valittiin projektiin jatkokehityksen ja ylläpidon helpottamiseksi.

Kolmas tapa järjestellä sovelluksen tiedostorakenne on toimintopohjaisesti, jolloin sovellus jaetaan pienimpiin mahdollisiin osiin. Tällä tavalla jokaisella sivulla on todennäköisesti useita micro frontendejä ja osa toiminnoista saattaa olla usealla eri sivulla. Tämä on hyvä lähestymistapa projektissa, jossa samaa toimintoa täytyy pystyä käyttämään usealla eri sivulla, sillä rakenne mahdollistaa toimintojen paljastamisen kaikille sovelluksen osille. Toimintojen testaus on myös erittäin helppoa, sillä testattavat komponentit ovat todella pieniä, mutta olemassa olevan sovelluksen muuttaminen tähän rakenteeseen on todella työlästä, etenkin jos sovellus on suuri. Tiedostorakennetta ei käytetty projektissa, sillä se olisi tehnyt uudelleenjärjestelyprosessista todella pitkän ja työlään, koska vanhan sovelluksen komponentit jakoivat paljon ajonaikaisia toimintoja.

4.3 Teknologiaprototyypit

Oikean työkalun löytämiseksi projektille sekä MFEA-sovelluksen rakentamisen testaamiseksi oli rakennettava useita eri prototyyppejä. Työkaluja testatessa oli myös pidettävä mielessä, kuinka helppo uuden kehittäjän olisi oppia käyttämään sitä, sillä uusia komponentteja tulisivat tekemään tiimit, joilla ei todennäköisesti olisi kokemusta micro frontendeistä.

Yhdellä työkalulla tehtävän prototyypin testauksen aikamääreeksi asetettiin yksi viikko rakentamisen aloituksesta. Valmiin prototyypin tuli tukea sekä Reactilla että Vuella rakennettua komponenttia, ja komponenttien tuli pystyä kommunikoimaan keskenään ja vaikuttamaan toisiinsa. Komponenttien kommunikointia testattiin lisäämällä molempiin komponentteihin numero, joka suureni yhdellä nappia painamalla. React-komponentin nappi lisäsi siis Vue-komponentin lukua ja toisinpäin.

Ensimmäinen prototyyppi rakennettiin Single SPA:lla, koska siitä löytyi enemmän esimerkkejä ja dokumentaatiota kuin muista työkaluista. Prototyypin kehitys alkoi Single SPA:lla tehtyjen esimerkkisovellusten etsimisellä ja sovellusrungon dokumentaation lukemisella. Prototyypille tehtiin yksinkertainen kolmen sovelluksen pohja, jossa komponentit luotiin Reactilla ja Vuella ja isäntäsovellus puhtaalla JavaScriptillä. Prototyypin rakennusvaiheessa ilmeni paljon ongelmia, jotka johtuivat pääasiassa Single SPA:n monimutkaisesta implementaatiosta. Single SPA:n käyttämiseen vaadittava koodi on hajanaista, eli sitä on lisättävä useaan eri tiedostoon, jotta sovellus toimi. Vaadittavan koodin syntaksi on myös hankalasti luettavaa, ja esimerkeistä oli vaikea ymmärtää, mitkä osat koodista olivat toiminnan kannalta tarpeellisia. Sovellusta ei toteutettu Single SPA:lla, sillä prototyyppiä ei saatu toiminnalliseksi sovitussa aikamääreessä ja teknologian vaikeus olisi ollut suuri kynnys uusille kehittäjille.

Toinen prototyyppi rakennettiin Web Componenteilla. Ne olivat hyvin yksinkertaisia oppia ja pääasiassa helppoja implementoida prototyyppiympäristössä. Esimerkkien komponentit eivät vaatineet suurta määrää hajanaista koodia lisättävän useaan eri tiedostoon, joten Web Componentit vaikuttivat hyvältä vaihtoehdolta sovelluksen muuttamiseen.

Ongelmana Web Componentien kanssa oli React-komponenttien luomisen vaikeus verrattaessa Vue-komponentteihin. React-komponentit vaativat paljon pohjustavaa koodia toimiakseen, vaikkakin vähemmän kuin vastaavassa Single SPA-sovelluksessa. Tämä oli ongelma, sillä vaikka uudet toiminnot voitaisiin tehdä Vuella kehityksen helpottamiseksi, vanha sovellus oli jo kehitetty

Reactilla. Web Componentit todettiin mahdollisesti hyväksi tavaksi toteuttaa projekti, sillä uusien kehittäjien olisi helppo oppia käyttämään työkalua kehittäessään toimintoja sovellukseen.

Kolmas prototyyppi tehtiin Module Federationilla, joka vaikutti sopivan projektiin täydellisesti. Module Federationin yhteensopivuus sovelluksen kanssa johtui sen toiminnasta Dockerin kanssa, joka on sovellusalusta, jota tultaisiin käyttämään projektin käyttöönottoon. Jokainen Module Federation -komponentti on autonomisesti omassa portissaan ajettava sovellus, joka paljastetaan komponentin asetustiedostosta eli Webpack.config.js-tiedostosta muille sovelluksille. Module Federation toimi helpommin Reactin kanssa kuin Web Components, eikä se vaatinut tiedostoihin lisättävää koodia toimiakseen. Module Federation vaati tosin toimiakseen muutaman tiedoston sisällön uudelleenjärjestelyä, sillä sovelluksen index.js ei saanut sisältää muita toimintoja kuin kutsua tiedostoa, joka sisältää React.DOMin.

Module Federation vaatii toimiakseen Webpackin, mikä on suuri etu pitkäaikaisessa kehityksessä, mutta kynnys uusille sovelluskehittäjille, jotka liittyvät projektiin väliaikaisesti. Webpackin hyöty on, että se tekee Module Federationin implementoinnista hyvin helppoa keskittämällä melkein kaiken MFE-toimintaan tarvittavan koodin yhteen tiedostoon. Tämä tekee MFE:hen liittyvien ongelmien ratkomisesta helppoa, sillä ongelmat ovat melkein aina samassa paikassa. Webpackista aiheutuva haitta on sen käytön opettelu, sillä Webpackin käyttö ja syntaksi ovat vaikeita oppia ja dokumentaatio on usein huonoa.

Vanhan sovelluksen kanssa hyvän yhteensopivuuden ja tulevaisuudessa lisättävien Docker-toimintojen vuoksi Module Federation valittiin micro frontendien kehitykseen Webpackin vaikeuksista huolimatta.

4.4 Toimintoprototyyppi

Oikean työkalun löydyttyä oli kehitettävä prototyyppejä testaamaan toimintoja, joita alkuperäinen sovellus vaati toimiakseen. Testattavia toimintoja olivat reititys hookrouter-kirjastolla ja PWA:n toiminto MFE-teknologian kanssa.

Hookrouter-reitityksessä ilmeni nopeasti ongelma, sillä sovellus ei löytänyt reitittimelle annetusta web-osoitteesta Vue-sovellusta, jonka olisi tullut siellä näkyä. Testattava komponentti, joka ei näkynyt isäntäsovelluksessa, oli kuitenkin päällä ja toiminnassa omassa portissaan, joten ongelman oli oltava isäntäsovelluksen puolella eikä itse komponentissa. Tämä ongelma ilmeni toistakin komponenttia testattaessa, joten yhteensopivuusongelma React.js, isäntäsovelluksen ja Vue-komponentin välillä ei vaikuttanut syyltä ongelmiin. Ongelman selvittämiseksi komponentit tuotiin omilta sivuiltaan isäntäsovelluksen päänäkymään, mikä sai komponentit toimimaan odotetusti. Tämä tarkoitti, että itse MFE-teknologia kyllä toimi, mutta Webpackilla tai Module Federationilla saattoi olla yhteensopivuusongelmia hookrouterin kanssa.

Ongelma johtui prototyypin navigaation puutteesta, minkä vuoksi sovellusta navigoitiin suoraan web-osoitetta muuttamalla. Tämän vuoksi sovellus ei saanut ilmoitusta käyttäjältä navigoinnin tapahtumisesta ja ohitti index.html-tiedoston, joka vaaditaan sovelluksen toimintaan, eikä siis kyennyt lataamaan sivua [17]. Vastaus ongelmaan oli historyApiFallback: true -parametrin lisääminen asetustiedoston dev-server-muuttujaan, joka määrittää, mistä Webpack ajaa paketoidut tiedot. Tämä korjasi ongelman uudelleenohjaamalla käyttäjän pyynnöt index.html:n kautta uuteen web-osoitteeseen.

Vanhan sovelluksen tärkeimmän ominaisuuden eli PWA:n käyttäminen Webpackin kanssa oli alusta asti ongelmallista. Koska vanha sovellus oli luotu Create React Application (CRA) -työkalukokoelmalla, oli myös sovelluksen PWA-toiminnot tehty työkalukokoelman mukana tulleella service workerillä. Vanhan sovelluksen service workeria ei kuitenkaan saatu toimimaan webpackin

kanssa, joten ajan säästämiseksi vanha service worker hylättiin, jotta paremmin webpackin kanssa toimivaa tapaa voitaisiin alkaa tutkimaan.

Webpackissa on mahdollisuus luoda sovellukseen service worker Workbox-lisäosaa käyttäen, joka vaati vain kirjaston lataamisen ja lisäosan kutsumisen asetuksissa. Workboxin generateSW-toiminto, joka luo sovellukseen service workerin vaikutti hyvin yksinkertaiselta, mutta esimerkkejä sen käytöstä oli vähän ja esimerkit olivat usein yhteensopimattomia sovelluksen kanssa. Lisäosan muuttujista oli dokumentaatiota, mutta dokumentaatio ei avannut muuttujille annettavia parametrejä tai edes niiden syntaksia millään merkityksellisellä tavalla. Oikeiden muuttujien löytäminen oli todella vaikeaa, ja niiden oikeat parametrit löytyivät pääasiassa arvaamalla. Dokumentaation vähyyden vuoksi parametrien syntaksin löytämisessä kului päiviä, mutta niiden löydyttyä sovelluksen uusi service worker toimi ilman enempää hienosäätöä.

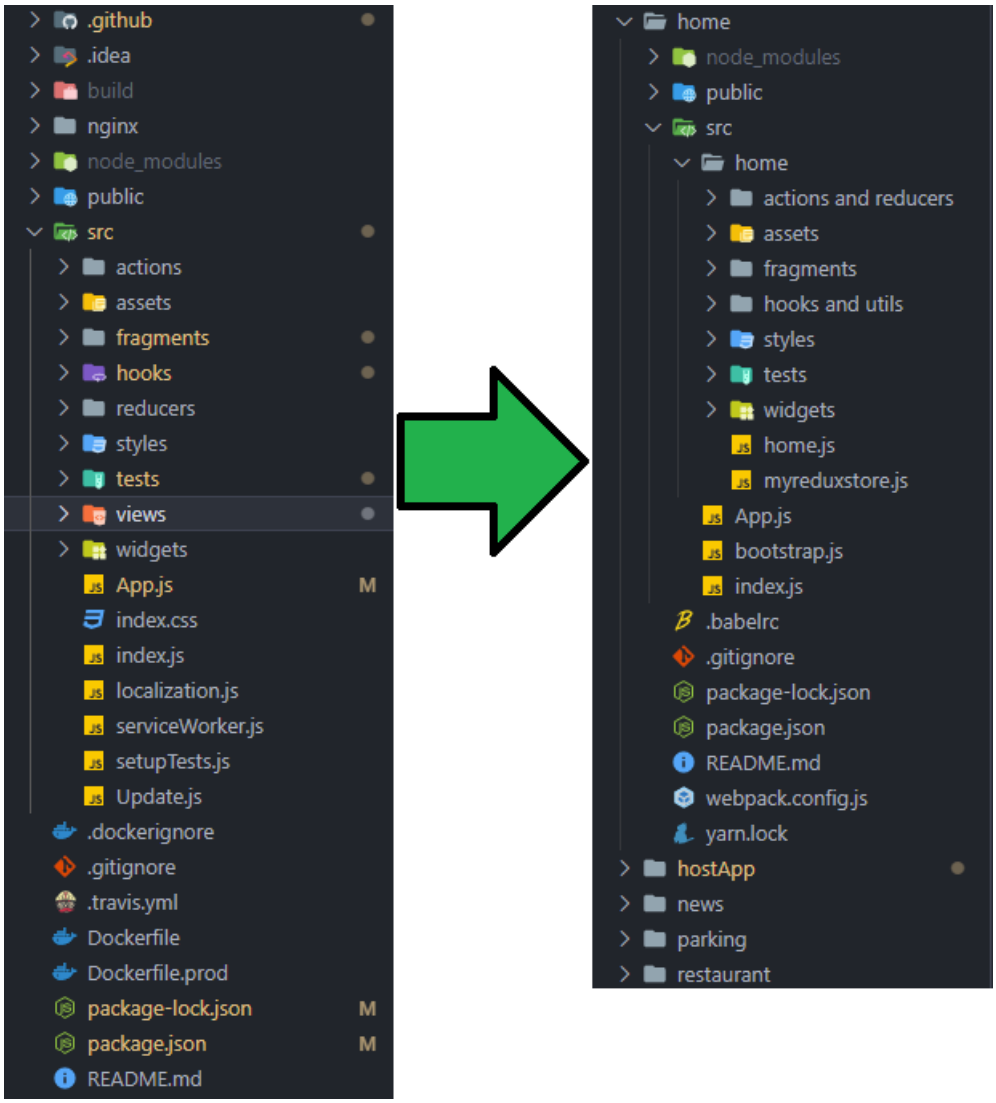
Redux-persistille oli tehtävä prototyyppi paljon muita prototyyppejä myöhäisemmässä vaiheessa, sillä teknologiaa ei käytetty vanhassa sovelluksessa. Kirjaston tarve ilmeni vasta sovellettaessa R+eduxia MFE-komponentteihin, koska Redux-persist mahdollisti tilojen tallentamisen siten, että sivun päivittäminen ei tyhjentäisi tilan sisältöä.

5 Uuden arkkitehtuurin toteutus

5.1 Tiedostorakenteen uudelleenjärjestely

Jotta tilaajayrityksen sovellus voitaisiin muuttaa käyttämään MFEA:ta, oli ensiksi jaettava sovelluksen osat autonomisiksi vanhan sovelluksen sisällä. Sovellus oli aikaisemmin jaettu osiin sen mukaan, minkätyyppisen toiminnon tiedosto sisälsi, mikä oli toimiva tapa järjestellä sovellus vielä, kun se oli kooltaan paljon pienempi. Tämä järjestelytapa oli kuitenkin yhteensopimaton MFE-teknologian kanssa ja hyvin epäselvä sovelluksen suuren koon vuoksi. Sovelluksen osat jaettiin sivupohjaisiin kansioihin, ja sivujen tarvitsemat ajonaikaiset toiminnot joko kopioitiin tai toiminnoille luotiin uudet autonomiset tiedostot, jos koodi oli

hajotettavissa. Näiden lisäksi sovellukseen luotiin global-kansio, joka koostuu tiedostoista, joita suurin osa komponenteista käyttää, kuten tyylit ja Redux-toiminnot. (Kuva 6.)

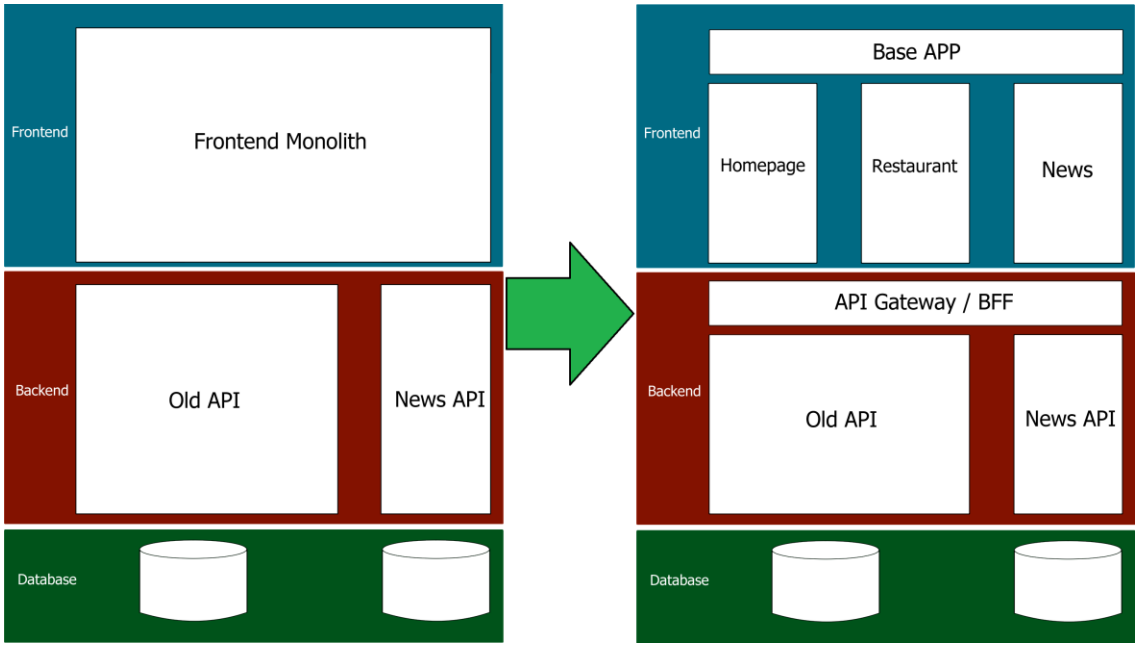


Kuva 6. Sovelluksen vanha tiedostorakenne ja uusi MFEA-tiedostorakenne.

5.2 Konversio

MFE-konversio oli todella yksinkertaista, kun sovellus oli uudelleenjärjestelty alkuperäisestää toimintopohjaisesta tiedostorakenteesta uuteen sivupohjaiseen

rakenteeseen. Jokaiselle tulevalle MFE:lle luotiin oma tyhjä React.js-pohja, jonka src-kansioon rakennettavan sovelluksen toimintokansio siirrettiin sellaisenaan. Kaikista toiminnoista ei kuitenkaan tehty omaa MFE:tä, vaan esimerkiksi sovellusten reititys ja navigointiin tarkoitettu yläpalkki jätettiin isäntäsovelluksen sisään. Navigointipalkista ei tehty MFE:tä, sillä sen tulisi näkyä jokaisella sivulla, joten oli hyödyllisempää jättää se ajettavaksi samaan instanssiin isäntäsovelluksen kanssa. (Kuva 7.)



Kuva 7. Vasemmalla visualisointi sovelluksen vanhasta arkkitehtuurista, ja oikealla MFEA:ta käyttämään muutettu sovellus.

Jokaiseen komponenttiin luotiin oma Babel.rc- ja Webpack.config.js-tiedosto. Komponentin asetuksissa paljastettujen tiedostojen sijainnit merkitään isäntäsovelluksessa nimellä, jota tiedoston kutsumiseen halutaan käyttää. Webpackin tiedostojen paljastamisen (expose) vuoksi isäntäsovelluksen navigaation importit oli muutettava webpackin kanssa yhteensopivaan muotoon (esimerkkikoodi 1). Komponentit eivät kuitenkaan välittömästi toimineet, sillä sovelluksen Redux-toimintojen vuoksi navigaatiossa oli käytettävä React.lazya. React.lazy oli tarpeellinen, jotta isäntäsovellus ei kaatuisi käynnistettäessä siihen, että se ei löydä haluttuja komponentteja.

```
Import App from 'component/App';
const App = React.lazy(() => import('component/App'));
```

Esimerkkikoodi 1. Normaalin paljastetun komponentin import ylempänä ja reduxia käyttävän komponentin import alempana.

5.3 Redux-komponentit

Redux toimi vanhassa sovelluksessa luomalla Redux storen heti sovelluksen käynnistyttyä käyttämällä staattisia eli etukäteen määriteltyjä sovitteita (reducer). Luomisen jälkeen varastolle ilmoitettiin subscribe-toiminnolla tilojen muutoksista, jotka sitten tallennettiin varastoon. Tämä oli toimiva ratkaisu monoliittisemmassa sovellusarkkitehtuurissa, sillä kaikki sovelluksen osat toimivat yhtäaikaisesti, mutta MFEA:han siirryttäessä ilmeni ongelma komponenttien asynkronisuuden kanssa. Ongelma tässä lähestymistavassa MFEA:ssa oli, että sovitteiden oli oltava autonomisuuden vuoksi komponentin sisällä, mutta komponentit eivät pysty käynnistymään ilman varastoa, eivätkä täten pysty paljastamaan sovitteita, jolloin varastoa ei pystytä luomaan.

Ensimmäinen idea ongelmien ratkomiseksi oli luoda uusi asynkroninen varasto, joka ei vaadi sovitteita heti luomisvaiheessa, vaan pystyy ottamaan ne asynkronisesti vastaan. Uuden varaston luominen oli nopeaa, mutta sen asynkronisten toimintojen ymmärtäminen oli paljon pidempi prosessi. Komponentteihin lisättiin tällä välin omat varastot niiden toimintojen testaamiseksi isäntäsovelluksen ulkopuolella. Tämä johti ratkaisuun, jossa komponentin App.js sai varaston muuttujana, kun App.js-funktiota kutsuttiin. Varasto annettiin funktioon bootstrap.js:n puolella tai isäntäsovelluksen reitittimessä. Tämä mahdollisti sekä autonomisen sovelluksen että MFE:n toimivuuden testauksen isäntäsovelluksessa yhtäaikaisesti, sillä se esti komponentin kaatumisen, jos isäntäsovellus ei ollut päällä.

Suurin ongelma injektoinnissa, joka mahdollistaa sovitteiden lisäämisen asynkronisesti, oli sen syntaksin ymmärtäminen pelkistä esimerkkiprojekteista, sillä dokumentaatiossa ei ollut esimerkkiä sen käytöstä. Syntaksin ja toiminnon

sisällön selvittyä sovitteiden injektoiminen oli helppoa, mutta sovellus ei vieläkään toiminut odotetusti.

Sovelluksen etusivun pienoisohjelmat katosivat sivua päivitettäessä, mikä tarkoitti, että sovelluksen työmuistin toiminnot eivät toimineet odotetusti ja sovitteiden tilat eivät tallentuneet, eli ne tyhjenivät sivua päivitettäessä. Ongelma vaikutti johtuvan injektion käytöstä etusivun sovitteiden saamiseen isäntäsovellukseen, sillä työmuisti vaati toimiakseen staattisen sovitteen. Tämä johtuu siitä, kuinka sivut toimivat Reduxin kanssa. Kun varasto luodaan, se saa listan sovitteista, joiden tiedot jäävät varastoon. Näiden sovitteet tiloista sivu voi tarvitessaan ottaa dataa sivun luomiseksi, jonka jälkeen sivu voi lähettää omat sovitteensa varastoon injektoimalla, ja niitä varasto voi käyttää tilojen hallintaan. Ongelman aiheutti, että WidgetReducer pääsi varastoon vasta, kun varasto oli jo luotu, ja varasto ei tämän vuoksi osannut hakea WidgetReducerin tallennettua tilaa, vaan loi sivun ilman tietoa valituista pienoisohjelmista. Väliaikaisena ratkaisuna sovitteista tehtiin kopio isäntäsovellukseen, ja se tuotiin staattisena varastoon, jolloin sovellus toimi odotetusti.

Varasto sai nyt tarvitsemansa sovitteet staattisina, mutta piensovellukset eivät silti toimineet. Tämän oletettiin johtuvan jostakin App.js:n sisällä olevista varastoa käyttävistä funktioista. Koodin vertaamien siihen, kuinka edellinen sovellus oli tehty, paljasti, että store.subscribe-toiminto oli jäänyt forEach-silmukan sisälle, mikä aiheutti vain viimeisimmän subscriben vaikuttavan sovellukseen. Tämän vuoksi WidgetReduceria ei koskaan kutsuttu store.subscriben sisäisellä funktiolla. Funktion siirtäminen ei vielä korjannut sovellusta, sillä funktioon syötettävien sovitteiden tilat olivat aina tyhjiä, sillä sovellus tallensi vain ensimmäiset tiedot sovitteen tiloista listaa joka syötettiin store.subscribeen. Tämän listan tiedot eivät koskaan muuttuneet, sillä se luotiin ennen store.subscribea, mutta listaa ei voitu luoda funktion sisälläkään, joten sovitteiden tilojen haut oli kirjoitettava funktioon sellaisinaan. Nämä funktiot siirrettiin myöhemmin Redux storeen tapahtumaan asynkronisten injektioiden kanssa samaan aikaan, jotta kaikki tarpeelliset tilat varmasti päivittyvät.

Väliaikaisena ratkaisuna sovitteiden tiedostojen tuominen isäntäsovellukseen ei kuitenkaan ollut hyvä tapa rakentaa sovellusta jatkokehityksen kannalta, joten sovelluksessa päätettiin testinä käyttää redux-persistiä, josta lyhyen testauksen jälkeen luotiin prototyyppiympäristö ilman MFEA:ta kirjaston toiminnan ymmärtämiseksi.

Redux-persist-prototyyppi oli yksinkertainen yhden sivun CRA-sovellus, joka käytti Redux storea tallentamaan käyttäjän antaman tekstin sovitteen tilaan, jonka redux-persist tallentaa työmuistiin. Varasto kopioitiin kehitettävästä sovelluksesta ja sovitettiin toimimaan prototyypin sovitteiden kanssa. Oikean rakenteen löytäminen redux-persistin käytölle vei muutaman päivän, sillä esimerkit oli yleensä hajautettu useampaan tiedostoon, ja vaikka kaikki käyttivät suunnilleen samanlaista rakennetta, oli niissä silti toimintoon vaikuttavia eroja. Oikean rakenteen löydyttyä sovellus saatiin toimimaan samalla tavalla kuin aikaisempi versio, eli vain staattisia sovitteita käyttämällä, joten prototyypissä alettiin testata dynaamisia sovitteita.

Dynaamiset sovitteet toimivat siten, että varastoa luodessa varastolle annetaan sovite ilman tilaa, ja sillä olisi sama nimi kuin komponentista lisättävällä sovitteella. Varaston valmistuttua sovellus injektoi asynkronisesti oikean sovitteen, joka korvaa tyhjän sovitteen tilan ja tallentaa tämän uuden sovitteen tilan persist-tilassa työmuistiin. Nyt ohjelma osaa hakea oikean sovitteen tilan työmuistista, sillä varastolla on oikeanniminen tyhjä sovite, jonka nimellä hakea tallennettu tila, ja sovelluksen sovite päivittää tallennetun tilan aina sivun sisällön muuttuessa.

Prototyypin valmistuttua uusi varasto yritettiin implementoida kehitettävään sovellukseen, mutta uusi varasto tallensi haluttujen sovitteiden lisäksi myös sovitteet, joita ei haluttu tallentaa työmuistiin. Sovitteet myös menettivät tilansa sivua päivitettäessä, vaikka sovitteet itse jäivätkin työmuistiin. Ongelman syyksi paljastui vanhojen injektiofunktioiden käyttö, ja tilat jäivät ongelmitta työmuistiin, kun tämä oli korjattu, mutta ylimääräisetkin sovitteet tallentuivat niiden mukana.

5.4 Tyylit

Alkuperäisen sovelluksen tyylittely oli tehty CSS:ää ja Material-UI:ta (MUI) hyväksikäyttäen, mutta uudessa sovelluksessa nähtiin järkeväksi siirtyä kokonaan yhteen tapaan tyylitellä sovellusta. Koska suurempi osa sovelluksen tyylittelystä oli tehty MUI:lla, päätettiin sitä käyttää myös jatkokehitykseen ja vanhat CSS-tyylit muutettiin MUI:ksi.

Vaikka tyylittelyn muuttaminen MUI:ksi olikin helppoa, tyylien ongelmien selvitys uudessa sovellusarkkitehtuurissa oli huomattavasti vaikeampaa. Koska sivun osat eivät enää olleet saman sovelluksen osia vaan autonomisia komponentteja, oli komponenteille luotava omat tyylit. Ongelma komponenttien omissa tyyleissä oli niiden yhteensopimattomuus isäntäsovelluksen itsenäisesti kehitettyjen tyylien kanssa. Tämä johti tilanteisiin, joissa sivu ottaisi tyylit joihinkin osiin komponentilta, toisiin isäntäsovellukselta ja joskus se ei käyttäisi tyylejä kummaltakaan. Tähän ongelmaan ei ollut mitään helppoa ratkaisua, vaan kaikki komponenttien ja isäntäsovelluksen tyylit oli käytävä läpi ja muutettava muotoon, jossa ne eivät häiritsisi toisiaan. Tämä vaati joskus myös generoitujen HTML-elementtien muuttamista, jotta molemmat tyylit eivät yrittäisi tyylitellä samaa elementtiä. Työtä vaikeutti myös se, että jos yhden sivun tyylien muuttamiseksi oli muokattava isäntäsovellusta, se saattoi rikkoa toisen sivun tyylit.

5.5 Admin-sovellus

Admin-sovellus oli uusi MFEA-sovellus, jonka kehitys aloitettiin alkuperäisen sovelluksen MFE-konversion jälkeen. Sovelluksen tarkoituksena oli pystyä valvomaan sovelluksen dataa, ja editoimaan sekä lisäämään sovellukseen uutisia. Admin-sovelluksen kehitys rinnakkain alkuperäisen sovelluksen konversion kanssa antoi näkymän siihen, miten MFEA-sovelluksen kehittäminen erosi olemassa olevan sovelluksen muuttamisesta arkkitehtuuriin.

Admin-sovelluksen MFE-toimintojen kehitys ei juuri eronnut konvertoitavan sovelluksen MFE-kehityksestä. Suurin ero alkuperäiseen sovellukseen verrattuna

oli tyylittelyn helppous uutta sovellusta tehtäessä, sillä monet ongelmat tyylittelyn kanssa olivat johtuneet niiden kehityksestä ilman MFEA:ta. Uuden sovelluksen kehitys alusta asti MFEA:lla mahdollisti tällaisten tyylivirheiden löytämisen aikaisessa vaiheessa, eivätkä ne päässeet kasautumaan.

Admin-sovelluksen kehitykseen tuli kuitenkin hidasteita, sillä sovelluksen selainpuolen tarpeiden vuoksi sen kehitystä ei ollut mahdollista jatkaa ennen palvelinsovelluksen toimintojen lisäämistä. Konversiossa palvelinsovellusta pystyttiin käyttämään sellaisenaan, sillä sovellukset olivat samoja kuin ennen, mutta autonomisia ja modulaarisempia. Uusia toimintoja kehittäessä tiimin pieni koko osoittautui kuitenkin riittämättömäksi, sillä yksi kehittäjä jouduttiin siirtämään sovelluksen selainpuolelta käytännössä kokonaan palvelinpuolelle, ja vain yksi kehittäjä jäi korjaamaan vanhaa sovellusta.

6 Insinöörityön tulokset

6.1 Ennusteiden osuvuus

Uuden teknologian opiskelusta tehtävät oletukset päätyvät usein olemaan vääriä, koska teknologian toiminnasta ei ole tarkkaa tietoa. MFEA:sta tehdyt ennusteet päätyivät kuitenkin olemaan noin puoliksi oikein, mutta teknologiat, joiden toimimaan saaminen päätyi viemään eniten aikaa, eivät olleet edes harkinnassa työn alkuvaiheessa.

Kuten työn alkuvaiheessa jo oletettiin, tulivat teknologian uutuus ja dokumentaation vähyys ongelmiksi, mutta tämä ilmeni vasta myöhemmässä vaiheessa, kun sovelluksen sivut oli jo muutettu MFE-komponenteiksi.

PWA-toiminnallisuus oli vanhan sovelluksen keskeisin toiminto, ja se oli saatava toimimaan myös MFEA:ksi muutetussa sovelluksessa. Kuten jo projektin alussa oletettiin, oli PWA-toiminnallisuus ongelma sovelluksen kehityksessä, mutta ongelmat johtuivat Webpackin PWA-lisäosasta, eivätkä MFE-teknologiasta. Aluksi oletettu huoli tarpeesta lisätä service worker kaikkiin komponentteihin erikseen

oli turha, sillä MFE-komponentit jakavat koodia isäntäsovellukselle renderöitäväksi eivätkä heijasta jo renderöityä toimintoa.

Suurin ongelma sovelluksen komponenttien muuttamisessa MFE-komponenteiksi oli Redux, jonka yhteensopivuutta MFEA:n kanssa ei edes harkittu kehityksen alussa. Redux sellaisenaan ei aiheuta ongelmia MFE-komponenteissa, ja varaston jakaminen komponentteihin on todella yksinkertaista. Ongelma Reduxin kanssa tulee, jos sovellus vaatii persisted-tiloja toimiakseen ja sovellusta kehittävä tiimi haluaa pitää kiinni MFEA:n autonomisuuden säännöistä. Helppo tapa lisätä persisted-tila on antaa tilan vaatima sovite staattisena varastoon, mutta jos varasto sijaitsee isäntäsovelluksessa, on koko sovite siirrettävä isäntäsovelluksen puolelle. Tässä tapauksessa samaa sovitteita on joko kaksi tai MFE-komponentti ei toimi autonomisesti ilman isäntäsovellusta. Redux persist korjasi tämän ongelman luomalla tyhjät sovitteet samoilla nimillä kuin syötettävät sovitteet, mikä antoi varastolle sen tarvitsemat tyhjät sovitteet, joihin syöttää persisted-tilat. Redux persistin käyttämisen oppimisessa kesti kuitenkin kauan aikaa, koska esimerkki-implementaatioita oli monenlaisia ja MFE-teknologian kanssa yhteensopivan esimerkin löytäminen oli vaikeaa

Odottamaton ongelma, jota ei otettu työn alussa lainkaan huomioon, oli se, kuinka raskas sovelluksesta tuli MFEA-implementaation myötä. Sovellus ei ollut toimintakyvyltään erityisen hyvä ennen konversiota, mutta PWA-auditoinnissa ilmeni, että sovelluksen toimintakyky oli laskenut entisestään. Jopa paketoituna MFE-komponenttien lataamisessa kestää huomattavan kauan sovellusten raskauden vuoksi. Vaikka sovellus toimiikin ilman kaikkien komponenttien käynnistämistä, siitä tulee vielä hitaampi, sillä sovellus jää useaksi hetkeksi etsimään puuttuvia komponentteja. Sovellusten käynnissä pitäminen vie jo itsessään huomattavasti enemmän resursseja, koska niiden on oltava autonomisia ja täten niitä on ajettava erikseen.

6.2 Kehitys

Sovellus, joka on alusta asti suunniteltu MFEA:ta käyttäen, tulee aina olemaan lopputulokseltaan parempi ja helpompi kehittää, mutta valmiin sovelluksen muuttaminen MFEA:ksi ei ole niin vaikeaa kuin sen luulisi olevan. Module Federationia käyttämällä uuden sovelluksen muuttaminen MFE-komponentiksi oli kahden tiedoston lisäämisen ja niiden vaatimien kirjastojen asentamisen päässä. Näistä tiedostoista Babel.rc on samanlainen jokaisessa sovelluksessa, ja Webpack.config.js vaatii vain portin numeron muutoksen ja paljastettavien tiedostojen sijantien ja nimien muuttamisen. Nämä muutokset tosin tekivät sovelluksesta vain toimivan MFE:n, mutta eivät vielä toimivaa sovellusta.

Sovelluksen kehittäjälle MFEA:n suurin kynnys on sen vaatima opiskelu ja materiaalin vähyys, kun se on kerran opiskeltu ainakin Module Federationilla uusien toimintojen muuttaminen MFE-komponenteiksi on erittäin helppoa. Korkean oppimiskynnyksen vuoksi olisi järkevintä, että sovellusta kehittävissä tiimeissä kaikille selainpuolen kehittäjille ei välttämättä opetettaisi teknologiaa. Useimmiten kehittäjä pystyy teknologian opittuaan selviytymään yksin MFE-teknologiaan liittyvistä ongelmista.

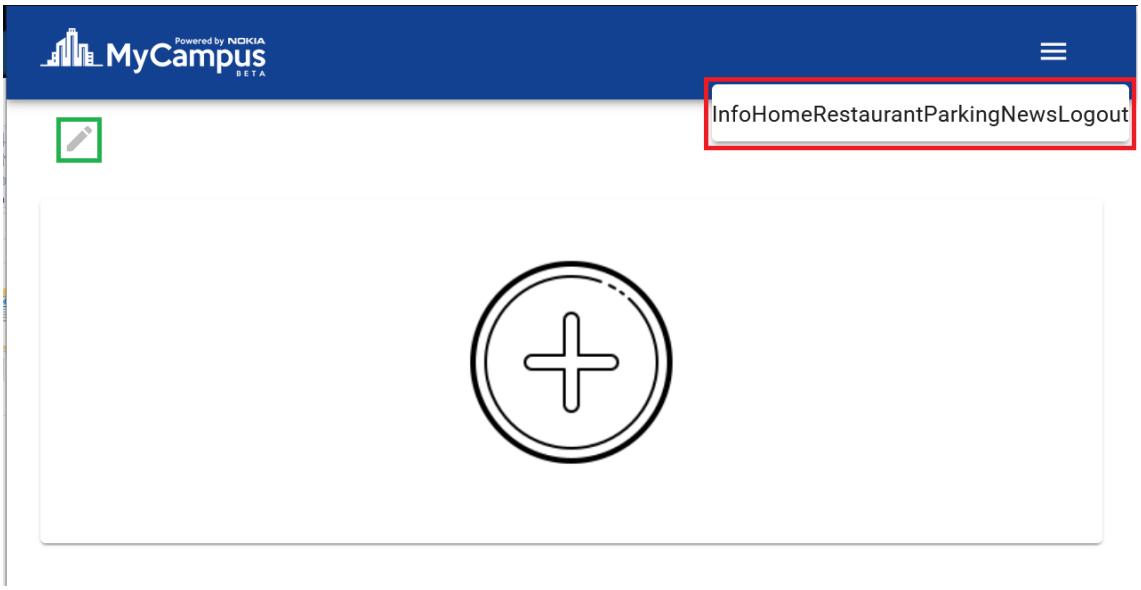
Vaikka teknologian vaatima opiskelun määrä on suuri, ovat materiaalit sen opiskeluun hyvin vähäiset tai huonot. Konkreettisten esimerkkien vähyyden ja huonon dokumentaation vuoksi ”shotgun debugging” -menetelmä tulee kehittäjälle hyvin nopeasti tutuksi, sillä useimmat ongelmat ratkeavat vain kokeilemalla ja hyvällä onnella. Tämän vuoksi on myös suositeltavaa, että kehittäjä dokumentoi omat ratkaisunsa hyvin, jos ei julkiseen käyttöön, niin ainakin itselleen, sillä tiedon unohduttua sen uudelleenlöytäminen on erittäin aikaa vievää.

Useammalla tiimillä suurempaa sovellusta kehittävän yrityksen kannalta MFEA:n käyttämisessä on enemmän esteitä. Koska MFEA-kehitys on vertikaalista, vaaditaan suuremman sovelluksen tekemiseen useita tiimejä kehittämään komponentteja, sillä yhden tiimin on mahdotonta saada useaa MFE-komponenttia aikaiseksi järkevässä ajassa. Pienempää sovellusta kehittäessä ei MFEA:n

tuoma kompleksisuus ole sen tuomien hyötyjen arvoista, ellei sovelluksen tiedetä jo etukäteen laajenevan. Yksittäisten komponenttien kehityksessä tiimien ei välttämättä tarvitse olla suuria, mutta tämän projektin kaltaista konversiota tehtäessä olisi hyvä, että tiimikoko olisi vähintään kaksinkertaisesti oman nelihenkisen tiimimme kokoinen ja sisältäisi alusta asti sekä selainpuolta että palvelinpuolta osaavia kehittäjiä.

MFEA on hyvä projekteihin, joissa on paljon vaihtuvia tiimejä luomassa eri komponentteja, mutta tämä vaatii yhtä tai useampaa ylläpitäjää, jotka ymmärtävät MFE-teknologiaa. Ylläpitäjät voivat ohjata MFE-komponenttien rakennuksessa tai muuttaa tiimien rakentamat tavalliset komponentit MFE-komponenteiksi niiden valmistuttua. Ylläpitäjät olisivat suositeltava lisäys tällaisen sovelluksen kehittämiseen aikaisemmin mainitun opiskelukynnyksen madaltamiseksi, sillä nopeasti vaihtuvilla tiimeillä harvoin on kuukausia aikaa opiskella uusi teknologia yhtä komponenttia varten.

Ongelmana kehityksessä saattaa ilmetä kehitettävien komponenttien tyylittely, sillä usean toisistaan riippumattoman tiimin on vaikea saada aikaiseksi yhtenäistä tyyliä sovellukselle, jos kaikki työskentelevät eri sivun parissa. Kaikkien tiimien olisi myös kehitettävä sovellusta isäntäsovelluksen kanssa ja koskematta isäntäsovelluksen tyyleihin, sillä muuten sovelluksen komponenttien tyylit saattavat rikkoutua, ja näiden ongelmien korjaaminen saattaa hyvinkin muodostua isommaksi urakaksi kuin komponentin kehitys. Esimerkkinä tästä tämän työn aikana tapahtunut tyylittelyongelma, jonka vuoksi sovelluksen yläpalkin navigaatio rikkoutui, kun kotisivun MFE tuotiin sovellukseen. (Kuva 8.)



Kuva 8. Vihreällä korostettuna home MFE:n pienoisohjelman edit-painike ja punaisella korostettuna isäntäsovelluksen MFE:n yläpalkin navigaatio.

Tämän konfliktin aiheuttivat piensovellusten editointinapin ja navigaation listan MUI-komponentit, minkä vuoksi navigaation listan sisältö muuttui riviksi. Tämän korjasi väliaikaisesti parkkihalli-piensovelluksen lisääminen sivulle, sillä kyseinen piensovellus sisälsi ListItem-elementin, joka kumosi aikaisemman tyylin päällekirjoittamisen. On edelleen epäselvää, mikä kohta MUI:sta tarkalleen aiheuttaa tämän tyylien rikkoutumisen.

Testauksen puolesta MFEA:ssa on hyvät ja huonot puolensa. Yksittäisten komponenttien testaus on helppoa ja paljon nopeampaa kuin monoliittisen sovelluksen testaus, sillä jos yhteen osaan koodia on tehty muutoksia, ei MFEA:ssa ole tarvetta testata koko sovellusta. Tämä johtuu siitä, että komponenttien ei tulisi jakaa ajonaikaisia toimintoja, mikä tarkoittaa, että komponenttien sisäisten muutoksien ei tulisi vaikuttaa muihin komponentteihin tai isäntäsovellukseen. Yksittäisten komponenttien testauksen helppouden ohella tulee myös haitta, sillä koko sovelluksen testaaminen kerralla vaikeutuu sovelluksen rakenteen vuoksi. Tilanteita, joissa kaikki sovelluksen testit haluttaisiin käydä läpi kerralla, ei pitäisi olla usein, jos sovelluksen tiimirakenne on muodostettu oikein, mutta tilanne saattaa tulla esiin jatkokehityksen tai ylläpidon puitteissa. Testien kirjoittaminen muuttuu myös joillakin osa-alueilla. Funktioiden testaus ei ole muuttunut

aikaisemmasta, mutta visuaalisten komponenttien testaus ei enää toiminut, sillä aikaisemmin käytetyt testauskirjastot eivät toimineet webpackin kanssa.

6.3 Jatkokehitys

MFEA on jatkokehityksen kannalta erinomainen arkkitehtuurimalli, jos siihen sitoutuu kunnolla ja pystyy pitämään sovelluksen kehityksessä vähintään yhden asiaa tuntevan kehittäjän. MFE-komponenttien kehittämisessä on kaksi tapaa kehittää sovellusta, suunnittelu ja konversio.

Kehitystavoista lopputulokseltaan parempi on suunnittelu. Tässä kehitystavassa MFE-kehitystä ymmärtävä kehittäjä on enemmän ohjaavassa asemassa, sillä kaikkien tiimien tulisi opetella, kuinka MFE luodaan ja kuinka arkkitehtuuri toimii. Tämä on ongelma pääasiassa vain silloin, kun kehittävät tiimit vaihtuvat jatkuvasti. Jos sovellusta kehittää muutama hyvin opiskellut tiimi, ei implementaatiolla ole tätä ongelmaa enää tiimin ensimmäisen komponentin jälkeen.

Alusta asti MFE:ksi suunniteltu komponentti on helpompi kehittää ilman, että sovelluksen tyylit rikkoutuvat. Jos sovelluksessa käytetään Reduxia ja PWA-teknologiaa, kuten tämän insinöörityön projektissa, voidaan niiden MFE-kohtaiset erilaisuudet ottaa huomioon jo työn alkuvaiheessa. Sovelluksessa käytettävä Reduxin storen kutsuminen voidaan tehdä useissa kohdissa monoliittisessa sovelluksessa, mutta MFE-komponentissa kutsu ja sovitteiden rekisteröinti on tehtävä tietyssä kohdassa koodia, jotta varasto saadaan isäntäsovelluksesta komponenttiin ja sovitteet isäntäsovellukseen. Jos sovellus on PWA, service workeria ei tarvita MFE-komponenteissa lainkaan, mutta testaus on tehtävä tämän vuoksi isäntäsovelluksessa. Jos komponenttia kehitetään omana sovelluksenaan, on PWA-toiminnot joko rakennettava komponenttiin, jotta toimivuutta voidaan testata ja poistettava jälkeenpäin, tai sovellus on rakennettava ilman toiminnallisuuksia, jolloin testaus ja kehitys jätetään konversion tekevälle kehittäjälle.

Kehitystavoista huomattavasti nopeampi on konversio, jota tässäkin työssä käytettiin. Konversio on tapa, jolla monoliittisesta sovelluksesta saadaan tehtyä MFE, tai isäntäsovellus ja useita MFE:tä, jos monoliittinen sovellus oli monisivuinen. Konversiolla on mahdollista ohittaa MFE-teknologian oppiminen kokonaan ja antaa tiimille kehitettäväksi tavanomainen sovellus, jolle suoritetaan konversio jälkeenpäin. Tällaista lähestymistapaa MFE-komponenttien luomiseen tulisi kuitenkin välttää.

Konversio on huomattavasti nopeampaa kuin suunnittelu ja vaatii vähemmän teknologiaa osaavia kehittäjiä. Kun sovellus koostuu vain yhdestä sivusta voi kehittäjä muuttaa sen MFE:ksi yksin. Konversio on nopeampaa, koska kehittäjien ei tarvitse opiskella MFE-teknologiaa ja tiimi voi kehittää tavanomaisen yhden sivun sovelluksen ja jättää konversion MFEA:ta ymmärtävälle kehittäjälle.

Aikaisemmin mainitun suunnitellun MFE-komponentin ratkaisemat ongelmat ilmenevät konversiossa, minkä vuoksi alusta asti suunnittelun tulisi olla ensimmäinen vaihtoehto kehityksessä ja konversiota tulisi tehdä pääasiassa vanhoja sovelluksia muutettaessa. Yksisivuisen sovelluksen tyylit rikkoutuvat usein, kun se muutetaan komponentiksi ja tuodaan isäntäsovellukseen, koska isäntäsovelluksen tyylejä ei ole otettu huomioon kehitysvaiheessa. Sovellukseen kehitetyt Redux-ratkaisut on kehitettävä uudestaan MFEA:han sopiviksi, ja Reduxia käyttäviä funktioita voidaan joutua siirtämään alemmille tasoille, jolloin sovelluksen logiikkaa saatetaan joutua uudelleenkirjoittamaan. On myös mahdollista, että isäntäsovelluksessa käytetyt PWA-toiminnot eivät toimi komponentissa, jolloin konversiota tekevän kehittäjän on käytettävä aikaa korjatakseen itselleen tuntemattoman koodin sisältöä.

MFEA:n mahdollistama teknologia-agnostisuus on sovellusta kehitettäessä erittäin hyödyllistä ja helpottaa tiimien kehitysprosessia huomattavasti. Teknologia-agnostisuuden hyödyt ilmenevät kaikkein näkyvimmin, jos kehittävät tiimit vaihtuvat aina komponentin valmistuttua. Tällaisen kehityksen hyödyt päätyvät tosin usein olemaan lyhytkestoisia, koska sovellusta on silti ylläpidettävä. Parhaassa tapauksessa komponentteja kehittävät tiimit olisivat vielä komponentin

valmistuttua kutsuttavissa ylläpitämään sovellusta ongelmien ilmetessä. Kehityksen rakenne ei tätä välttämättä kuitenkaan mahdollista, jolloin sovelluksella on todennäköisesti vain yksi tai muutama ylläpitäjä, joiden on nyt osattava kaikki sovelluksessa käytetyt teknologiat.

7 Yhteenveto

Insinöörityössä toteutettiin MFE-konversio olemassa olevaan sovellukseen ja selvitettiin, onko micro frontend -arkkitehtuuri hyvä vaihtoehto tämän ja muiden vastaavanlaisten sovellusten kehitykseen. Insinöörityön aihe valittiin, koska työn pohjasovellus tarvitsi tavan helpottaa jatkokehitystä, kun sovelluksen parissa työskentelevät tiimit vaihtuvat nopeasti. MFEA vaikutti vastaukselta tähän ongelmaan, sillä uusien tiimien tarvitsisi opiskella vain MFE-teknologia kaikkien sovelluksen osien opiskelun sijaan, mikä nopeuttaisi sovelluskehitystä.

MFEA on mikropalveluarkkitehtuuriin perustuva uusi sovellusarkkitehtuurimalli, jonka on tarkoitus tuoda mikropalveluarkkitehtuurille tyypillinen modulaarisuus selainpuoleen. MFEA koostuu yksittäisistä micro frontendeistä, jotka ovat teknologia-agnostisia autonomisia komponentteja. MFE:n autonomisuus johtuu siitä, että se ei jaa muiden sovellusten osien kanssa ajonaikaisia toimintoja ja mahdollistaa sovelluksen osien yksittäisen kehityksen. MFE:n autonomisuuden luoma modulaarisuus taas mahdollistaa komponenttien poistamisen tai sammuttamisen ilman, että koko sovellus on suljettava. Teknologia-agnostisuus tarkoittaa sitä, että komponentti voidaan kirjoittaa millä tahansa JavaScript-pohjaisella sovellusrungolla ja se silti toimii sovelluskokonaisuudessa. Tämä mahdollistaa sovellusta kehittävien tiimien autonomisen toiminnan.

Insinöörityön pohjasovellus oli Reactilla toteutettu PWA, joka käytti Reduxia tilanhallintaan, ja MFEA toteutettiin sovellukseen Webpackin Module Federationilla. Module Federation vaatii komponenteilta tiettyä rakennetta toimiakseen, mutta keskittää kaiken MFEA:han liittyvän yhteen Webpackin tiedostoon. Ongelma Module Federationissa on jo valmiiksi monimutkaisen sovellusarkkitehtuurin lisäksi Webpack, jota on vaikea ymmärtää ja käyttää. Webpack tuo myös

mukanaan mahdollisuuden käyttää sille kehitettyjä kirjastoja erilaisten toiminnallisuuksien mahdollistamiseksi. Ongelma näissä kirjastoissa on se, että ne ovat usein tiettyyn käyttötarkoitukseen tehtyjä valmiita pakkauksia, joiden muokkaaminen tämän vuoksi on joko hankalaa tai mahdotonta. Micro frontend -PWA:n kehittäminen Webpackilla ei ole vaikeaa, kunhan löytää oikean esimerkin Webpackin PWA-moduulin syntaksista.

Alkuperäisessä sovelluksessa käytetty Redux oli hyvin vähäisessä roolissa, ja konfiguroiminen halutulla tavalla oli helpohkoa. MFEA-konversio vaikeutti Reduxin käyttöä huomattavasti, sillä sen vähäisen roolin vuoksi alkuperäisessä sovelluksessa suurin osa Redux-reducereista oli staattisia. Toiminnallisesti sovitteet olisi voinut jättää staattisiksi, mutta se olisi vaikeuttanut jatkokehitystä, jonka parantaminen oli työn tarkoitus. Paras tapa jatkokehityksen kannalta oli jättää sovitteet komponentteihin, joissa niitä käytettiin, ja poistaa komponenttien sovitteet isäntäsovelluksesta. Tämä myös vähentäisi isäntäsovelluksen kompleksisuutta. Jotta sovitteet saataisiin isäntäsovelluksen varastoon komponenteista, oli ne injektoitava varastoon komponentin käynnistyttyä. Tämä osoittautui vaikeaksi, sillä komponentit käynnistyivät varaston kanssa samaan aikaan ja sovitteet oli saatava varastoon ennen isäntäsovelluksen käynnistymistä. Tämä onnistui redux persistillä, joka antoi isäntäsovellukselle tyhjän sovitteen ja joka päivitettäisiin oikeaksi sovitteeksi komponentin käynnistyttyä.

MFEA on monessa mielessä hyvin kätevä arkkitehtuuri, mutta sen kompleksisuus sekä dokumentaation vähyys luovat kehittäjälle kynnyksen sen opiskeluun. MFEA-sovellusta kehittävällä taholla ongelmia voi olla enemmän, kuten sovelluksen raskaus ja ylläpitäjien sekä useamman tiimin tarve. MFEA pohjainen sovellus ei sovi pieniin projekteihin tai projekteihin, joissa on hyvin rajallinen määrä työntekijöitä. MFEA ei kuitenkaan sisällä vain haittoja vaan on erittäin hyvä siinä, mitä yrittää tehdä. Yksittäisten komponenttien ylläpito ja testaus ovat erittäin helppoja oikealla tiimirakenteella, ja jatkuva käyttöönotto mahdollistaa sovelluksen jatkuvan toiminnan komponenttien huollosta tai poistamisesta huolimatta. Teknologiavapaus antaa yksittäisille tiimeille paremmat mahdollisuudet tuottaa parhaita mahdollisia tuloksia, ja tarkka roolijako komponenttien ja

komponenttien sisäisten osien kehityksessä mahdollistaa täyden keskittymisen tiimin tai kehittäjän omaan sovelluksen osaan.

Lähteet

1 Geers, Michael. Micro Frontends - extending microservice idea to frontend development. Verkkoaineisto. Micro-frontends. < https://micro-frontends.org/>. Luettu 9.1.2021.

2 Software Architecture & Software Security Design. Verkkoaineisto. Synopsys. < https://www.synopsys.com/glossary/what-is-software-architecture.html>. Luettu 26.2.2021.

3 Mauersberger, Laura. 2017. Microservices: What They Are and Why Use Them. Verkkoaineisto. LeanIX. < https://www.leanix.net/en/blog/a-brief-history-of-microservices>. 14.8.2017. Luettu 11.03.2021.

4 Richardson, Chris. What are microservices? Verkkoaineisto. Microservice Architecture. < https://microservices.io/>. Luettu 26.2.2021

5 Stenberg, Jan. 2016. The Long History of Microservices. Verkkoaineisto. InfoQ. < https://www.infoq.com/news/2016/11/microservices-history/>. 25.11.2016. Luettu 11.3.2021.

6 Lewis, James & Fowler, Martin. 2014 Microservices. Verkkoaineisto. Martinfowler. < https://martinfowler.com/articles/microservices.html >. 25.3.2014. Luettu 21.4.2021.

7 Wolff, Eberhard. 2017. Self Contained Systems (SCS): Microservices Done Right. Verkkoaineisto. InfoQ < https://www.infoq.com/articles/scs-microservices-done-right/>. 3.5.2017. Luettu 11.3.2021.

8 Wolff, Eberhard. 2016. Self-contained systems: A different approach to microservices. Verkkoaineisto. Jaxenter. < https://jaxenter.com/self-contained-systems-a-different-approach-to-microservices-130466.html>. 29.11.2016. Luettu 11.3.2021.

9 Are You Technology Agnostic? 2018. Verkkoaineisto. Commonplaces interactive. < https://www.commonplaces.com/blog/are-you-technology-agnostic/>. 6.9.2018. Luettu 5.3.2021.

10 Kofler, Jürgen. 2020. How Microfrontends Can Help to Focus on Business Needs. Verkkoaineisto. InfoQ. <https://www.infoq.com/articles/microfrontends-business-needs/?itm_source=articles_about_micro-frontends&itm_medium=link&itm_campaign=micro-frontends>. 13.6.2020. Luettu 11.3.2021.

11 Eberhardt, Colin. 2021. You probably don’t need a micro-frontend. Verkkoaineisto. Scott Logic. < https://blog.scottlogic.com/2021/02/17/probably-dont-need-microfrontends.html >. 17.2.2021. Luettu 11.3.2021.

12 Brian, Matt. 2014. Google’s new ‘Material Design’ UI coming to Android, Chrome OS and the web. Verkkoaineisto. Engadget. < https://www.engadget.com/2014-06-25-googles-new-design-language-is-called-material-design.html >. 25.6.2014. Luettu. 28.4.2021.

13 What is Webpack. Verkkoaineisto. survivejs. < https://survivejs.com/webpack/what-is-webpack/ >. Luettu. 28.4.2021.

14 Why would I use Webpack? 2018. Verkkoaineisto. Tinselcity. < http://tinselcity.net/whys/packers >. 3.4.2018. Luettu 28.4.2021.

15 Archard, Chris. 2019. Redux Crash Course with Hooks. Verkkoaineisto. DEV. < https://dev.to/chrisachard/redux-crash-course-with-hooks-a54 >. 2.10.2019. Luettu 5.5.2021.

16 Gaunt, Matt. Service Workers: and Introduction. Verkkoaineisto. Web | Google Developers. < https://developers.google.com/web/fundamentals/primers/service-workers >. Luettu. 23.7.2021

17 Connect.history-api-fallback. 2019. Verkkoaineisto. Github. < https://github.com/bripkens/connect-history-api-fallback >. 30.12.2019. Luettu 1.4.2021.