TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

OLLI SAARINEN

OFFLINE-TOIMINNALLISUUS LOGISTIIKAN WEB-SOVELLUKSESSA

Diplomityö

Tarkastaja:
Assistant Prof. David Hästbacka

Tarkastaja ja aihe hyväksytty
1. marraskuuta 2017

TIIVISTELMÄ

OLLI SAARINEN: Offline-toiminnallisuus logistiikan web-sovelluksessa
Tampereen teknillinen yliopisto
Diplomityö, 53 sivua, 2 liitesivua
Maaliskuu 2018
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Automaation tietotekniikka
Tarkastaja: Assistant Professor David Hästbacka

Avainsanat: logistiikka, offline-toiminnallisuus, web-sovelluskehitys, Redux

Digitalisaatio on yleistynyt valmistavassa teollisuudessa, mutta sen tuomat hyödyt on huomattu muilla myös toimialoilla, kuten esimerkiksi logistiikassa. Digitalisaatiolla on mahdollisuus vähentää yrityksen prosesseissa syntyvää hukkaa, jolloin ne itsessään tehostuvat.

Nykyään älypuhelimet ja tabletit ovat osa jokapäiväistä arkea, mikä mahdollistaa niiden hyödyntämisen myös tuotantoympäristössä. Pöytäkoneisiin verrattuna ne tuovat kuitenkin mukanaan uusia haasteita, kun verkkoyhteys ei tulekaan enää verkkojohtoa pitkin. Kun verkkoyhteys vaihtelee tai katkeaa kokonaan, joudutaan mobiililaitteella offline-tilaan. Web-sovelluksesta tulee käyttökelvoton, jos offline-toiminnallisuutta ei ole otettu suunnittelussa huomioon. Tunnistettua ongelmaa lähdettiin ratkaisemaan toimintatutkimuksen avulla, jossa haettiin teorian avulla ratkaisuja ongelman selvittämiseksi.

Aluksi tutustuttiin offline-toiminnallisuuden suunnitteluperiaatteisiin, ja vertailtiin vaihtoehtoja offline-toiminnallisuuden toteuttamiseksi web-sovelluksissa. Lisäksi pohdittiin offline-toiminnallisuuden toteutuksen mukanaan tuomia haasteita, jotka on huomioitava järjestelmän suunnittelussa ja tehdyissä ratkaisuissa. Koska tutkimukseen liittyvä asiakasprojekti oli jo aloitettu, olivat varsinaisen lopullisen toteutuksen raamit hyvin rajatut.

Redux-arkkitehtuuriin pohjautuva ratkaisu pystyi tästä huolimatta täyttämään sille asetetut offline-vaatimukset logistiikan toimintaympäristössä, vaikka lähtötilanne ei ollut suunnittelumielessä ideaalinen. Offline-toiminnallisuus päätettiin lopulta toteuttaa selaimen välimuistin ja service workereiden avulla. Nämä W3C:n (*The World Wide Web Consortium*) standardoimat ratkaisut osoittautuivat toimiviksi web-sovelluksen offline-tilan hallitsemisessa. Käytetty Redux-arkkitehtuuri mahdollisti vaivattomasti sovelluksen tilan synkronoimisen välimuistin kanssa ja web-käyttöliittymän suunnittelemisen käyttötapauskohtaisesti soveltuvilla suunnittelumalleilla.

ABSTRACT

Olli Saarinen: Offline functionality in a logistics web application
Tampere University of Technology
Master of Science Thesis, 53 pages, 2 Appendix pages
March 2018
Master's Degree Programme in Automation Technology
Major: Information Systems in Automation
Examiner: Assistant Professor David Hästbacka

Keywords: logistics, web application development, offline functionality, Redux

For the past few years digitalization has been trending in manufacturing. Benefits and potential of the digitalization have been discovered also beyond industrial applications, for example in logistics. By utilizing digitalization, examined processes can be improved when waste inside them is reduced.

Smartphones and tablets are part of our everyday life, which has made it possible to exploit them also in manufacturing environments. In comparison to desktop computers, mobile devices have brought about new challenges with wireless network connections. When network connection varies or is disconnected, offline state is confronted. Web applications will be useless if a possible offline state has not been taken into account during the development process. An action research based study was carried out to map solutions to offline state that were later used to solve the identified offline problem.

In the beginning of the research, offline design principles and patterns were explored in the scope of web applications. Furthermore challenges, raised by the possible implementation of offline functionality were introduced. The challenges discussed must be considered in order to successfully implement an offline-capable web application. However, the linked case project had already been started which narrowed down possible solutions. Offline functionality was implemented into the frontend relying on Redux architecture.

Although the baseline for offline the design process was far from ideal, the used solutions were able to satisfy the offline requirements set by the end users. Offline functionality was carried out by utilizing browser cache and service workers. These implementations are standardized by W3C (*The World Wide Web Consortium*) and they turned out qualified in offline state management. The Redux architecture enabled with easy to synchronize the application state with browser cache and to design the web user interface with use case specific design patterns.

ALKUSANAT

Haluan kiittää ohjaajiani David Hästbackaa ja Jani Timmerheidiä upeasta ohjauksesta ja avusta diplomityön tekemisprosessin aikana. Diplomityön aiheeksi sain kehiteltyä jotain modernia ja mielenkiintoista, josta olen Leanwarelle myös kiitollinen. Sitten päästään teihin läheisiin ja rakkaimpiini. Ihanampaa perhettä ja ystäviä en voisi toivoakaan.

”Smile, Laugh, Sleep, Repeat!”

Tampereella, 14.2.2018

Olli Saarinen

SISÄLLYSLUETTELO

KUVALUETTELO

OHJELMALUETTELO

LYHENTEET JA MERKINNÄT

AJAX	engl. Asynchronous JavaScript And XML, kokoelma web-tekniikoita asynkroniseen tiedonsiirtoon, myös jalkapalloseura Hollannissa
API	engl. Application Programming Interface, ohjelmointirajapinta
Backend	Web-sovelluskehyksen palvelinpuoli
CouchDB	Apachen HTTP-rajapintaan perustuva NoSQL-tietokanta
CRUD	engl. Create, Read, Update, Delete, perusfunktiot pysyvän tiedon muokkaamiseen
CSS	engl. Cascading Style Sheets, HTML-sivun tyylin määrittävä tyylitiedosto
DOM	engl. Document Object Model, ohjelmointirajapinta puumaisille tiedostoille
ERP	engl. Enterprise Resource Planning, toiminnanohjausjärjestelmä
Flux	Facebookin kehittämä arkkitehtuurimalli web-sovelluksille
Frontend	Web-sovelluskehityksen käyttäjälle näkyvä web-käyttöliittymä
HTML	engl. HyperText Markup Language, hypertekstin merkintäkieli
HTTP	engl. HyperText Transfer Protocol, sovellusprotokolla hypermedian toteutukseen
IndexedDB	Standardoitu selaimen paikallinen tietokanta
JS	engl. JavaScript, korkean tason dynaaminen ohjelmointikieli
localStorage	Selaimen paikallinen tallennustila avain-arvo-pareille
MES	engl. Manufacturing Execution System, tuotannonohjausjärjestelmä
MVC	engl. Model-View-Controller, web-arkkitehtuurimalli
PouchDB	Selaimessa toimiva JS-pohjainen tietokanta
PWA	engl. Progressive Web Application, progressiivinen web-sovellus
SOAP	engl. Simple Object Access Protocol, WS*-perheeseen kuuluva avoin tiedonsiirtoprotokolla
SPA	engl. Single Page Application, yhden sivun web-sovellus
RAIL-malli	Käyttäjäkeskeinen malli kuvaamaan web-käyttöliittymän suorituskykyä
React.js	engl. JavaScript-kirjasto web-käyttöliittymien rakentamiseen
Redux	Flux-arkkitehtuurimalliin pohjautuva funktionaalinen arkkitehtuurimalli web-käyttöliittymille
RFID	engl. Radio Frequency IDentification, radiotaajuinen etätunnistus menetelmä
REST	engl. REpresentional State Transfer, HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen
URI	engl. Uniform Resource Identifier, resurssien tunnistusmenetelmä
Webpack	Web-käyttöliittymän staattisten resurssien paketointityökalu
WS*-perhe	engl. Web Service, kokoelma SOAP-protokollaan pohjautuvia standardeita
WMS	engl. Warehouse Management System, varastonhallintajärjestelmä
W3C	engl. The World Wide Web Consortium, WWW:n standardeja kehittävä ja ylläpitävä järjestö
XML	engl. Extensible Markup Language, rakenteeltaan puumainen merkintäkieli

1. JOHDANTO

Digitalisaatio on levinnyt erityisesti valmistavaan tuotantoon, mutta alkanut jo jalkautua myös pienemmille toimialoille ja yrityksiin. Digitalisoimalla tehostetaan yritysten prosesseja, kun ylimääräiset toimet ja hukat, esimerkiksi paperisten asiakirjojen käsittely, minimoidaan. Suuressa roolissa ovat myös web-sovellukset, jotka usein liitetään moderniin tietojärjestelmään. Uusia web-tekniikoita syntyy jatkuvasti, ja hallitseva tekniikka voidaan nähdä lähes muoti-ilmiönä, mikä on syytä huomioida tietojärjestelmää suunnitellessa.

Useimmilla meistä varmasti on hyviä kokemuksia web-sovelluksista ja vielä todennäköisimmin niitä huonoja. Huonot käyttäjäkokemukset syntyvät, jos web-sovellus ei toimi oletetulla tavalla. Sovelluksesta voikin tulla täysin käyttökelvoton, kun yhteyttä verkkoon ei ole tai yhteyden laatu vaihtelee sattumanvaraisesti.

Sama lähtökohta on myös asiakasprojektissa, johon tämä diplomityö pohjautuu. Käsiteltävä asiakasprojekti on logistiikan tietojärjestelmä. Projektissa ongelman omistajia ovat työnjohtaja ja työkoneenkuljettajat, jotka käyttävät järjestelmää varastossa tai lastausalueella. Työn suorittamisen kannalta suuria ongelmia työkoneenkuljettajille aiheuttavat verkko-ongelmat, jotka johtuvat esimerkiksi toimipaikan maantieteellisestä sijainnista tai alueella sijaitsevista katvealueista.

Diplomityön tavoitteena on suunnitella ja toteuttaa toimiva offline-toiminnallisuus työkoneiden käyttämään web-sovellukseen, joka mahdollistaa työn tekemisen myös verkko-ongelmien ilmetessä. Ideaalitilassa web-sovellus hoitaa offline-toiminnallisuuden taustalla, eikä käyttäjä välttämättä edes tiedä verkkoyhteyden olevan poikki. Työssä selvitettävät tutkimuskysymykset ovat:

1. Mitä hyötyjä voidaan saavuttaa offline first -suunnittelumallin avulla?
2. Millaisia haasteita offline-toiminnallisuus luo web-sovellukselle?
3. Miten offline-toiminnallisuus voidaan toteuttaa Redux-pohjaiseen[1] logistiikan web-sovellukseen?

Esitettyjen tutkimuskysymysten ratkaisemiseen käytetään toimintatutkimusta (*action research*). Toimintatutkimuksen ideana on samanaikaisesti perehtyä tutkimuksen aihealueen teoriaan, käytännön ongelmiin ja ratkaisuihin. Tavoitteena on löytää teoreettisia malleja ja ratkaisuja, joita voidaan soveltaa käytännön ongelmien ratkaisussa. Jotta ite-

[1] Flux-arkkitehtuurimalliin pohjautuva funktionaalinen arkkitehtuurimalli web-käyttöliittymille. [50]

ratiivinen tutkimusprosessi olisi tehokas, on tiivis yhteistyö tutkimuksen sidosryhmien välillä välttämätöntä. [1]

McKay & Marshall [1] korostavat julkaisussaan, että toimintatutkimus koostuu kahdesta rinnakkaisesta mutta riippuvaisesta iteratiivisesta kehitysprosessista. Tutkijan tehtävänä on etsiä uusia teoreettisia ratkaisuja, joita voidaan soveltaa tavoitellun käytännön ratkaisun kehityksessä. Yhdessä tutkimuksen ongelman omistajien kanssa tutkija sitten perehtyy käytännön toteutukseen ja sille asetettuihin vaatimuksiin. Mainittu toinen iteratiivinen prosessi syntyy käytännön ratkaisun vaatimus- ja hyväksymismäärittelystä. Ongelman omistaja testaa ja validoi teorian pohjalta kehitettyä ratkaisua, mikä antaa lähtökohdan tutkijan seuraavalle iteraatiokierrokselle. [1]

Työssä sovelletaan toimintatutkimusta offline-toiminnallisuuden suunnitteluun ja toteutukseen ohjelmistokehityksen näkökulmasta. Tutkimuksen alussa esitellään yleisesti web-sovelluskehitystä ja web-sovelluksien arkkitehtuurimalleja. Luvussa 4 kartoitetaan sekä vertaillaan offline-toiminnallisuuden toteuttamiseksi erilaisia suunnittelumalleja ja ratkaisuja, jotka muodostavat tutkimuksen kehitysprosessin teoreettisen näkökulman. Käytännön ongelman ja ratkaisun vaatimusmäärittely tehdään yhdessä ongelman omistajien kanssa, jolloin käytännön toimintatapoihin ja ongelmiin on mahdollista perehtyä. Toteutettava ratkaisu testataan ja validoidaan ongelman omistajien toimesta, mistä kerätyllä palautteella voidaan tarvittaessa aloittaa uusi kehitysprosessin iteraatiokierros.

2. WEB-SOVELLUSKEHITYS

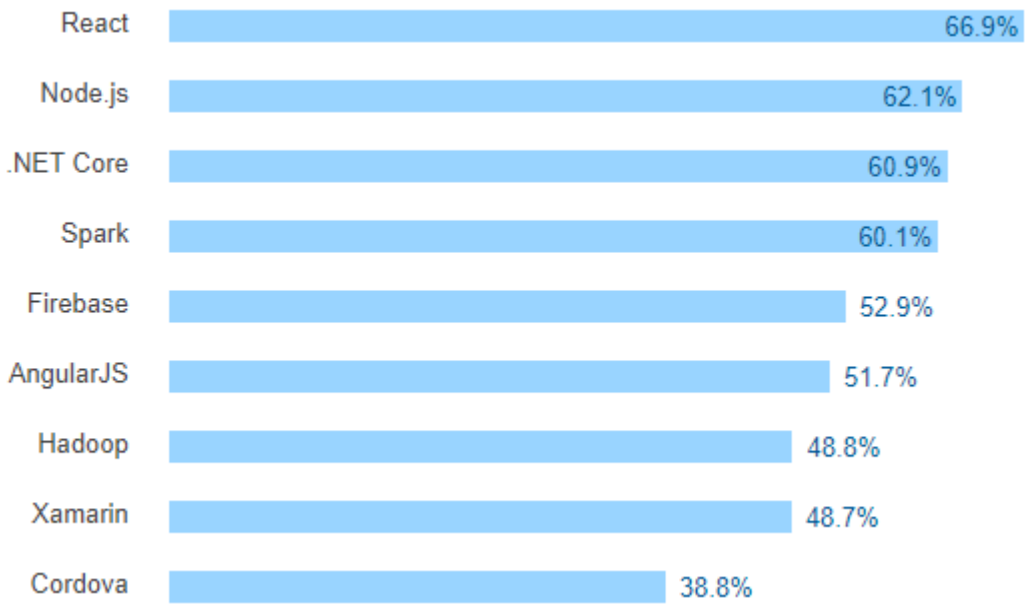
Tässä luvussa käsitellään web-sovelluskehitystä ja -kehystä, jotka muodostavat pohjan tutkittavalle offline-toiminnallisuudelle. Luvussa 2.1 käydään läpi yleinen web-sovelluskehys ja sen komponentteja (ks. kuva 2). Tämän jälkeen esitellään web-sovelluskehityksen viimeisimpiä suuntaviivoja. Luvussa 2.2 käydään läpi yhden sivun web-sovellus ja lopuksi mietitään funktionaalisen ohjelmoinnin roolia web-sovelluskehityksessä.

Web-sovelluksella on perinteisesti tarkoitettu palvelua tai resurssia, johon käyttäjä pääsee yksiselitteisellä Internet-osoitteella, URI:lla (*Uniform Resource Identifier).* Alkujaan web-sovellukset toimivat hyvinkin staattisesti, noudattaen tiukasti asiakas-palvelin-mallia. Kun käyttäjä teki pyynnön uudelle sivulle, haettiin valmis sivu verkon yli HTTP-protokollan avulla (*Hypertext Transfer Protocol*). Tällä hetkellä web-sovelluksilta vaaditaan myös dynaamisuutta, jossa yhden sivun (*Single Page Application, SPA*)- ja progressiiviset web-sovellukset (*Progressive Web Application, PWA*) vastaavat huutoon. Yksinkertaiset web-sivut koostuvat HTML- (*HyperText Markup Language*), CSS- (*Cascading Style Sheets)* ja JS-tiedostoista (*JavaScript*). Web-sivun runko rakentuu HTML-komponenteista, joiden tyyliä voidaan käsitellä CSS-tiedostossa ja toiminnallisuutta muokata JS-tiedostossa.

Web-sovelluskehitys ja sen merkitys ohjelmistotuotannossa on kasvanut suuresti viime vuosina. Nykyään yhä useammat selaavat Internetiä mobiililaitteilla perinteisen tietokoneen sijasta [2]. Suuri merkitys suosion kasvuun on myös älypuhelimien ja tablettien teknisellä kehityksellä sekä suosiolla. Web-sovelluskehityksen ehdottomana etuna on ketterä ohjelmistopäivitysten tekeminen, jolloin muutokset saadaan jokaiselle käyttäjälle vaivattomasti. Toisena selkeänä etuna voidaan pitää koodin monikäyttöisyyttä esimerkiksi natiiveihin mobiilisovelluksiin verrattuna. Innovatiivisten web-sovelluskehysten ja -kirjastojen ansiosta selainpohjainen ohjelma toimii moitteettomasti niin tietokoneella, tabletilla kuin älypuhelimessakin [3].

Uusien teknologioiden nopea kehitys ja teknologioiden murrokset luovat haasteita myös sovelluskehitykselle, kun saatavilla olevia tekniikoita on paljon. Tekniikoiden välinen paremmuus ei aina ole yksiselitteistä, vaan välillä se on enemmänkin muoti-ilmiö. Tämä näkyy myös Stack Overflow:n teettämässä vuotuisessa kyselytutkimuksessa [4], joka selvittää ohjelmistokehittäjien käyttämiä tekniikoita ja mieltymyksiä. Vaikka lähde ei täysin tieteellinen olekaan, antaa tunnetun yhteisön 64 000 kehittäjän kattava tutkimus hyvää näkökulmaa tilanteeseen. Tutkimuksen mukaan JavaScriptin käyttö on lisääntynyt ja se on edelleen käytetyin tekniikka sovelluskehityksessä. Suuri syy tähän on myös

suositut ja käyttäjäystävälliset JavaScript-kehykset sekä -kirjastot, jotka esiintyvät edukseen myös kuvan 1 tehdyssä tutkimuksessa.



Kuva 1. *Stack Overflow:n kyselytutkimuksen pidetyimmät web-tekniikat, joita kehittäjät haluavat käyttää tulevaisuudessa. [4]*

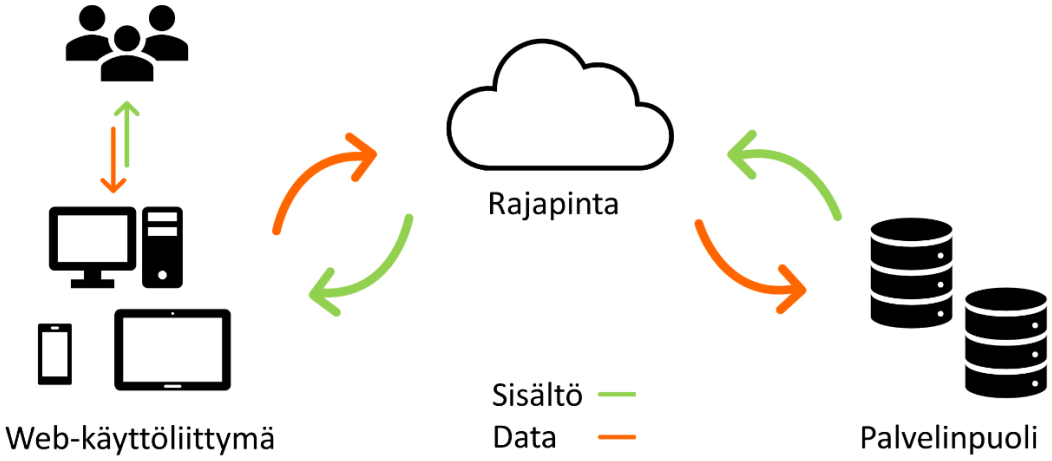
Kuvassa 1 on esitetty kyselytutkimuksen mukaan pidetyimmät web-tekniikat, joita ohjelmistokehittäjät haluavat tulevaisuudessakin käyttää. JavaScript-pohjaisista tekniikoista loistavasti pärjäsivät React, Node.js ja AngularJS, mikä tukee JavaScriptin käyttöä sekä suosiota. On kuitenkin muistettava, että aina tulee uusia tekniikoita ja JavaScriptillekin löytyy korvaaja ennen pitkään.

2.1 Web-sovelluskehys

Kun tarkastellaan web-sovelluskehitystä, voidaan karkean tason arkkitehtuurikehys jakaa kolmeen osaan:

- Web-käyttöliittymä, josta käytetään myös nimitystä *frontend.*
- Palvelinpuoli, josta käytetään myös nimitystä *backend*.
- Rajapinta.

Luetellut komponentit on kuvattu kuvassa 2, joka sisältää kehyksen osat ja esittää niiden välistä yhteistoimintaa.



Kuva 2. *Yleisen web-sovelluskehyksen komponentit.*

Kuvassa 2 vihreillä nuolilla esitetään tiedonsiirtoa sisällön esittämiseksi käyttäjälle. Oranssit nuolet puolestaan esittävät käyttäjältä saatua tietoa, jonka pohjalta web-sovellus toimii.

2.1.1 Web-käyttöliittymä

Web-sovelluksen ainoa loppukäyttäjälle näkyvä osa on käyttöliittymä, jota voidaan käyttää erilaisilla päätelaitteilla. Web-sovelluksen käyttöliittymää suoritetaan asiakkaan selaimen resursseilla, jolloin käyttäjältä ei vaadita erinäisten liitännäisten asentamista.

Web-käyttöliittymän tehtävänä on esittää käyttäjälle palvelinpuolelta välitettyä sisältöä ja toimintalogiikan mukaisesti reagoida käyttäjän syötteisiin. Koska käyttäjä näkee vain web-käyttöliittymän, on erittäin tärkeää suunnitella se käyttäjäystävälliseksi. Kuten Egger [5] artikkelissaan esittää, on käyttäjäkokemuksen vaikutus tuotteen tai palvelun arvon luonnissa merkittävä. Eggerin mukaan käyttäjän luottamus ja mielipide web-sovellusta rakentuu jo ensi kosketuksessa, esimerkiksi sovelluksen brändäyksestä ja käytettävyydestä. Mitä pidempään käyttäjä selaa sivustoa tai käyttöliittymää, sitä suurempi merkitys on myös sivuston kokonaisuuden pätevyydellä ja mahdollisten transaktioiden luotettavuudella.

2.1.2 Palvelinpuoli

Palvelinpuolella tarkoitetaan käyttäjältä piilossa olevia resursseja ja toimia, joita hallitsevat palveluntarjoajat, kehittäjät ja ylläpitäjät. Palvelinpuolen tehtävänä on toteuttaa web-käyttöliittymästä saadut pyynnöt ja palauttaa rakennettu uusi sisältö takaisin käyttäjälle. [6]

Palvelinpuoli koostuu kontrollereista, sovelluslogiikasta, tietokannoista ja muista mahdollisista palveluista, joita ovat esimerkiksi sanoma- sekä viestijonopalvelut. Kontrollerit huolehtivat web-käyttöliittymältä saapuvista pyynnöistä, jotka ne välittävät palvelinpuolen ohjelmistolle toteutettavaksi. Palvelinpuolen vastuulla ovat myös käytettyjen tietokantojen toiminta sovelluksen tiedon tallentamisessa ja sovelluksen tietoturvallisuudesta huolehtiminen, esimerkiksi hyökkääjiä vastaan.

2.1.3 Rajapinta

Web-käyttöliittymä ja palvelinpuolen varsinainen sovelluslogiikka sekä kontrollerit eivät kuitenkaan vaihda tietoja suoraan keskenään, vaan se tapahtuu erillisen rajapinnan, API:n (*Application Programming Interface*) kautta. Rajapinnalla tarkoitetaan yhdessä määriteltyjä toimintoja, metodeita ja objekteja, joita esimerkiksi web-käyttöliittymä ja palvelinpuoli voivat yhteistoiminnassaan hyödyntää. Rajapinnan avulla saadaan integroitua selain- ja palvelinsovellus toisiinsa.

Tällä hetkellä yhä useammin ratkaisuksi rajapinnan toteutukseen web-sovelluksissa nousee REST-arkkitehtuuri (*Representational State Transfer*), jonka etuina on keveys, moni- ja helppokäyttöisyys verrattuna raskaampaan ja standardoidumpaan WS*-perheen SOAP-protokollaan (*Simple Object Access Protocol*) [7]. REST-arkkitehtuurin esitteli ensi kertaa Fielding [8] tutkimuksessaan, jonka mukaisesti REST-rajapinnan avulla voidaan kohdistaa web-käyttöliittymästä pyyntöjä URI:en päässä oleviin palvelinpuolen resursseihin HTTP:n CRUD-metodeilla (*Create, Read, Update, Delete*).

Rajapinnan toteutus on muuten löyhä sallien suuren vapauden ohjelmistokehitykselle ja sovelluskomponenttien ja -kerrosten integroimiselle. REST-arkkitehtuuri tarjoaa tilattoman tiedonsiirron asiakas-palvelin-mallille, jolloin web-käyttöliittymä on riippumaton palvelinpuolen tilasta [8, 9].

Pautasso et al. [10] tuovat julkaisussaan selkeästi esille perusteita rajapinnan toteutuksen valintaan REST:in ja WS*-perheen välillä. REST-arkkitehtuurin valitseminen antaa kehittäjille vapautta, eikä pakota tekemään useampia arkkitehtuurivalintoja kuten WS*-perheen kanssa. Vapauden kanssa kulkevat myös hallinnalliset ja teknilliset riskit, jotka korostuvat ilman kunnollista ja yhteneväistä määrittelyä REST-rajapinnasta. Nyrkkisääntönä voikin pitää, että WS*-perheen valinta kannattaa suuren mittakaavan integraatiossa, joka vaatii erityisominaisuuksia esimerkiksi turvallisuudelta tai palvelun laadulta.

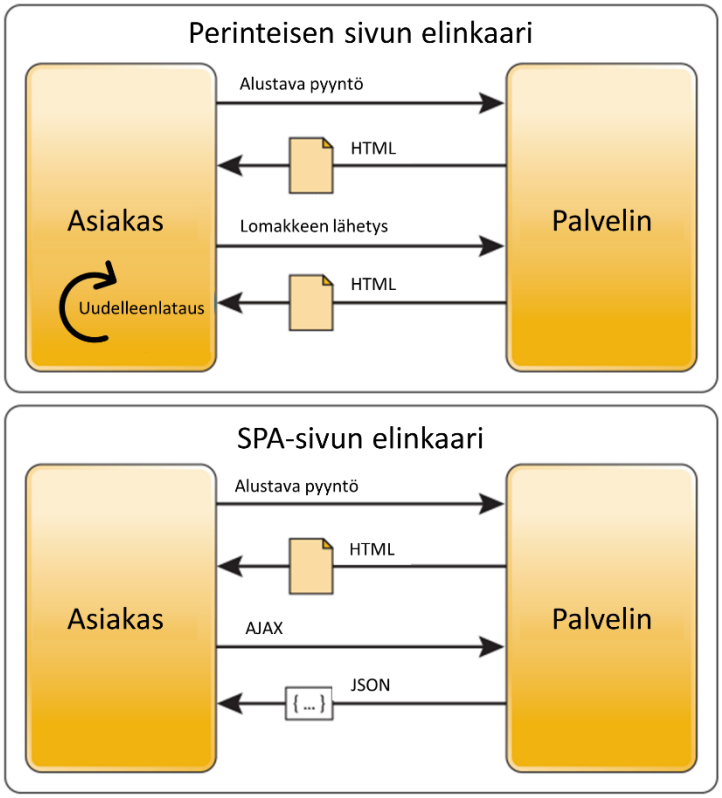
2.2 Yhden sivun web-sovellus

Yhden sivun web-sovelluksen (SPA) ideana on minimoida web-sivun uudelleenlataukset, jotka ovat totutusti perinteisten web-sovellusten toimintapa. Wassonin [11] mukaan

SPA-sovellus on web-sovellus, joka lataa vain yksittäisen HTML-sivun ja dynaamisesti päivittää sitä käyttäjän toimien mukaisesti.

Yksinkertaisuudessaan SPA-sovelluksen toimintamalli on seuraava, kuten kuva 3 esittää:

1. Sovellus lataa palvelimelta alustavalla pyynnöllä kaikki tarvitsemansa resurssit.
2. Kun käyttäjä tekee pyynnön web-käyttöliittymässä, lähetetään pyyntö AJAX-tekniikoiden (*Asynchronous JavaScript And XML*) avulla palvelimelle.
3. Palvelin palauttaa web-käyttöliittymälle vain tarvitun tai muuttuneen tiedon esimerkiksi JSON-objektina (*JavaScript Object Notation*).
4. Web-käyttöliittymä käsittelee uuden tiedon ja päivittää tarvittavat sivun komponentit. [12]



Kuva 3. *Perinteisen ja SPA-sovelluksen web-sivun elinkaaret, mukaillen lähteestä [11].*

Kuvassa 3 on myös esitetty perinteisen web-sovelluksen toimintatapa. Siinä palvelinpuoli on vastuussa päivitetyn HTML-sivun luonnista, kun SPA-sovelluksessa vastuu sisällön päivittämisestä on siirretty web-käyttöliittymän puolelle. [11, 12]

SPA-sovelluksen ehdoton hyöty on sen nopeus, kun raskaita resursseja, HTML-, CSS- ja JS-tiedostoja, ei ladata selaimeen kuin käynnistyksessä. Rakenteen keveys ja vähäinen tiedonsiirto palvelimen välillä parantavat käyttäjäkokemusta, kun web-sovelluksen toiminta sujuvoituu eikä koko sivun uudelleenlatauksia tarvitse odotella. [13] Tämä lisää merkittävästi käyttäjän tai asiakkaan kokemaa arvoa web-sovelluksen tarjoamasta

palvelusta [5]. Ohjelmistokehityksen näkökulmasta SPA-sovellus muuttaa perinteisiä palvelinpuolen ja web-käyttöliittymän vastuita siirtämällä toimintalogiikkaa selaimelle. Tällöin palvelinpuolen ja web-käyttöliittymän kehitys pysyvät erillään [12], mikä helpottaa esimerkiksi päivitysten tekemistä ja arkkitehtuurin hallintaa.

Vastuiden muutoksella on myös kääntöpuolensa. SPA-sovellukset vaativat toimiakseen raskaita kirjastoja ja kehyksiä, jotka on ladattava sovelluksen käynnistyessä selaimeen. SPA-sovellus ei kuitenkaan ole vastaus kaikkeen. Jos sivuilla on paljon raskasta sisältöä tai se jakautuu selkeisiin osa-alueisiin, perinteinen web-sovellus voi yhä olla parempi ratkaisu (ks. kuva 3). Ei myöskään ole poissuljettua yhdistää lähestymistapoja hybridityyliseksi sovellukseksi. [13]

2.3 Funktionaalinen ohjelmointi

Funktionaalinen ohjelmointi on yksi vanhoista ohjelmointiparadigmoista, mutta web-sovellusten suosio ja kehitys ovat antaneet sille paljon huomiota. Hudakin [14] mukaan funktionaalinen ohjelmointi perustuu puhtaisiin funktioihin. Puhtailla funktioilla ei ole sivuvaikutuksia, vaan ne tuottavat annetuille lähtöarvoille aina saman lopputuloksen. Sivuvaikutusten puutteesta seuraa, että käsiteltyjen objektien tilat pysyvät aina muuttumattomina (*immutable*).

Kaipiainen [15] toteaa tutkimuksessaan, että funktionaalinen ohjelmointi sopii hyvin web-ohjelmistokehitykseen. Se parantaa ohjelmoijien tuottavuutta sekä heidän koodinsa laatua. Syntyvä yksinkertainen tieto- tai tapahtumavirta (*stream*) sopii etenkin reaktiivisiin web-käyttöliittymiin, joissa reagoidaan käyttäjän syötteisiin tai palvelinpuolelta saatuun dataan.

Saadun suosion myötä funktionaalinen ohjelmointi on levinnyt myös ei-funktionaalisiin ohjelmointikieliin. Ohjelmassa 1 on esitetty Java:n Stream-API:n [16] avulla toteutettu metodi, joka hyödyntää rajapinnan tarjoamia funktionaalisen ohjelmoinnin ominaisuuksia.

```
List<String> getPersonsWithPet(List<Person> persons) {

// Haetaan annetuista henkilöistä ne, joilla on lemmikki. Palautetaan
// heidän koko nimet listassa sukunimen mukaan järjestettyinä.
  return persons.stream()
                .filter(person -> person.hasPet())
                .sorted(comparing(Person::getLastName))
                .map(person -> person.getFullName())
                .collect(toList());
}
```

Ohjelma 1. *Funktionaalisesti toteutettu metodi Java:n Stream-API:n avulla.*

Ohjelmassa annetaan `getPersonsWithPet()`-metodille lista `Person`-tyypin henkilöitä, joista palautetaan uudessa listassa lemmikkejä omistavien henkilöiden nimet. Funktionaalisen ohjelmoinnin periaatteen mukaisesti objektit pysyvät muuttumattomina. Esimerkiksi ohjelmassa 1 ei voida suoraan muuttaa annetun objektin tilaa, vaan muutokset voidaan tehdä vain luomalla uusi objekti.

3. MODERNIT WEB-SOVELLUSARKKITEHTUURIT

Web-sovelluskehitykseen liittyy aina lähes poikkeuksetta arkkitehtuurimallien käyttö, ja niin myös tässä tutkimuksessa. Arkkitehtuurimalli selkeyttää web-sovellusta, parantaa sen ylläpidettävyyttä ja tarjoaa ratkaisuja tunnettuihin ongelmatapauksiin. Tässä luvussa käydään läpi tunnettu ja yleisesti käytetty MVC-sovellusarkkitehtuurimalli (*Model-View-Controller*), jota verrataan ratkaisua hieman eri suunnasta lähestyvää Flux-arkkitehtuurimalliin. Tutkimuksen asiakasprojektissa käytetty web-sovelluksen arkkitehtuuri pohjautuu esiteltävään Flux-arkkitehtuurimalliin.

3.1 MVC-arkkitehtuurimalli

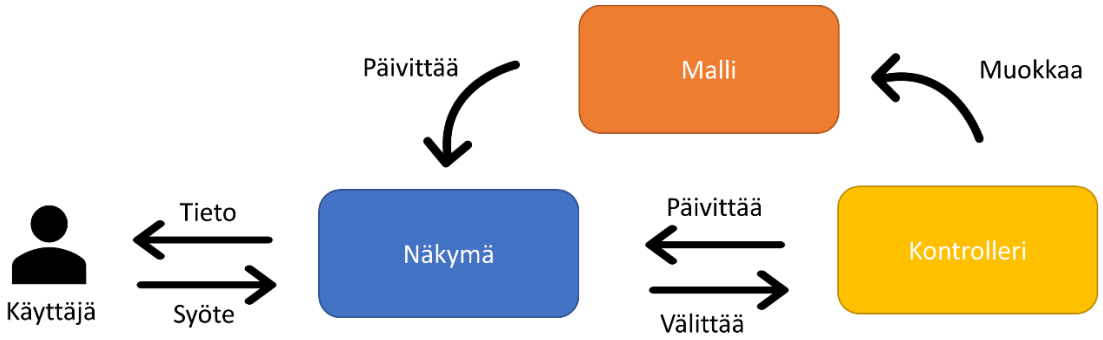
Ensi kerran MVC-arkkitehtuurimallin määrittelivät Krasner & Pope [17] jo 1980-luvun loppupuoliskolla. Hyväksi todettu arkkitehtuurimalli on yhä tähän päivään asti laajassa käytössä, etenkin kun puhutaan web-sovelluksista, jotka yleisesti perustuvat käyttäjän kanssa vuorovaikutukseen. MVC-arkkitehtuurimallia hyödyntäviä web-sovelluskehyksiä ovat muun muassa Spring, ASP.NET ja Ruby on Rails [18]. MVC-arkkitehtuurimallin avulla sovellus voidaan erottaa kolmeen toiminnallisuudeltaan erilliseen osaan:

- malliin (*model)*
- näkymään (*view*)
- kontrolleriin (*controller*). [19]

Koska kyseessä on vain arkkitehtuurimalli, se ei edellytä tiukkoja vaatimuksia toteutuksille, vaan eri alustojen ratkaisut voivat olla hyvinkin erilaisia. Ajan myötä MVC-arkkitehtuurimallin ympärille on syntynyt MV*-arkkitehtuuriperhe, kun on kehitetty uusia MVC-arkkitehtuurimalliin pohjautuvia ratkaisuja. Kaikilla näillä malleilla pohjalla oleva ideologia on sama, mutta esimerkiksi osien välinen yhteistyö ja vastuut ovat erilaisia. [18]

MVC-arkkitehtuurimallin toisistaan erotetut komponentit parantavat sovelluksien modulaarisuutta ja helpottavat sovelluksen osien uudelleenkäyttöä. Arkkitehtuurimalli soveltuu myös suuremmille kehittäjätiimeille mahdollistaen hyvin sovelluksen korkeantason toiminnallisuuden hallinnan. [20, 21] Mallin tehtävänä on kuvata sovelluksen tietomalli, jonka esittämisestä käyttäjälle huolehtii näkymä. Kontrollerin vastuulla on sovelluslogiikan toteutus, joka perustuu käyttäjän antamiin syötteisiin tai muihin määriteltyihin ominaisuuksiin. [19] Aiemmin mainitut arkkitehtuurimallin komponentit, malli,

näkymä ja kontrolleri on kuvattu kuvassa 4, jossa nuolilla on ilmaistu komponenttien vuorovaikutussuhteita ja vastuita.



Kuva 4. *MVC-arkkitehtuurimalli, perustuen lähteeseen [20].*

Kuvan 4 mukaisesti jokaisella komponentilla on oma roolinsa arkkitehtuurimallin yhteistoiminnassa. Näkymän tehtävänä on tiedon esittämisen lisäksi olla vuorovaikutuksessa käyttäjän kanssa ja välittää tehdyt pyynnöt sekä tapahtumat kontrollerille. Kontrolleri puolestaan reagoi näkymän välittämiin syötteisiin ja tarvittaessa muokkaa mallia. Kun mallin sisältämä tieto on päivittynyt, käskee se näkymää päivittymään. On myös mahdollista, että kontrolleri päivittää tiedon suoraan näkymälle. Tämä on hyödyllistä esimerkiksi tilanteissa, joissa näkymän esittämää tietoa halutaan järjestää tai suodattaa. [19–21]

Kuten Syromiatnikov & Weyns [18] esittävät, soveltuu MVC-arkkitehtuurimalli erittäin hyvin web-sovelluskehitykseen, koska näkymät ja kontrollerit ovat luontaisesti erotettuina toisistaan. MVC-arkkitehtuurimallissa on kuitenkin haasteellista hallita web-käyttöliittymän tilaa ja sen logiikkaa. Näkymien ja kontrollereiden riippuvuus lisää logiikkaa ja dataa palvelinpuolelle, jossa täytyy ylläpitää sekä sovelluksen että näkymän tilaa ja toteuttaa niiden toimintalogiikkaa. Sovellusten kasvaessa myös komponenttien väliset tieto- ja tapahtumavirrat monimutkaistuvat ja niiden hallinta vaikeutuu.

3.2 Flux-arkkitehtuurimalli

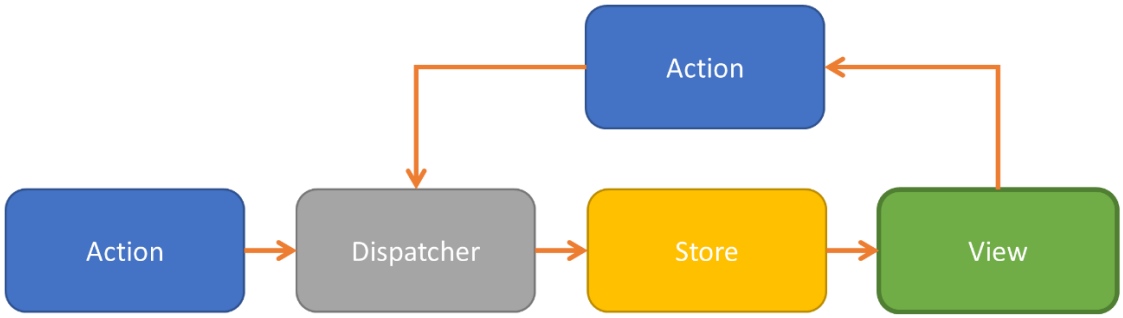
MVC-arkkitehtuurimallilla syntyviä haasteita pyrkii ratkaisemaan Facebookin kehittämä Flux-arkkitehtuurimalli, jonka ideana on yhdensuuntaistaa web-sovelluksen sisällä kulkeva tietovirta. Flux-arkkitehtuurimalli koostuu neljästä keskeisestä komponentista, joita ovat:

- action-komponentti
- dispatcher-komponentti
- store-komponentti
- view-komponentti.

Vaikka malli hieman muistuttaakin perinteistä MVC-arkkitehtuuria komponentteineen, ei sitä saa siihen sekoittaa. [22]

3.2.1 Yhdensuuntainen tietovirta

Flux-arkkitehtuurimallin lähtökohtana on yhdensuuntainen tietovirta, joka määrittää tarkat säännöt komponenttien väliseen tiedonvaihtoon. Yhdensuuntaisuuden vuoksi tapahtumaketju ja samalla myös tietovirta pysyy aina vakioituna, mikä helpottaa sovelluksen tilan hallintaa. Arkkitehtuurin rakenne sopiikin erinomaisesti funktionaaliseen ohjelmointiin, koska sovelluksen ja komponenttien tilaa päivitetään aina yksiselitteisesti DOM-mallin (*Document Object Model*) hierarkiaa noudattaen. [22] Kuvassa 5 on esitetty Flux-arkkitehtuurin tietovirran kulku.



Kuva 5. *Flux-arkkitehtuurimallin tietovirta, mukaillen lähteestä [22].*

Kuvan 5 mukaisesti tietovirta alkaa action-komponenteista, joita voidaan luoda tapahtumista (*events*) tai view-komponenttien toimesta. Dispatcher-komponentti jakaa action-komponenttien avulla päivitetyn tiedon halutuille store-komponenteille. Viimeisenä view-komponentit hakevat tarvitsemansa tilatiedon store-komponenteilta käsittelyyn ja esittävät sen käyttäjälle.

3.2.2 Komponentit

Ainoa keino välittää tietoa Flux-sovelluksessa ovat action-komponentit. Action-komponentin tulee sisältää tyyppi (*type*) ja vapaavalintaisesti muuta tietoa, jota halutaan välittää eteenpäin. [22] Ohjelmassa 2 on esitetty yksinkertainen action-komponentti, jonka tehtävänä on muuttaa tunnetun tuotteen tietoja web-sovelluksessa.

```
{
   type: 'MODIFY_PRODUCT',
   id: '23',
   price: '150',
   discount: '15'
}
```

Ohjelma 2. *Action-komponentin tietorakenne.*

Ohjelman 2 action-komponentti välittää signaalin muuttaa kyseiselle tuotteelle uuden hinnan sekä määritetyn alennuksen. Action-komponenttien käsittelyyn ja rakentamiseen käytetään usein action creator -apumetodeita, joiden tehtävänä on luoda tietyn tyypin omaava action-komponentti halutuilla parametreilla. [22] Ohjelmassa 3 on esitelty `createModifyProductAction()`-metodi, jota käytetään ohjelman 2 action-komponentin luomiseksi.

```
function createModifyProductAction(id, price, discount) {

  const action =    {
    type: 'MODIFY_PRODUCT',
     id,
    price,
    discount
  };
  Dispatcher.dispatch(action);
}
```

Ohjelma 3. *Action creator -apumetodi.*

Ohjelman 3 funktio luo uuden action-komponentin annetuista parametreista ja välittää sen suoraan dispatcher-komponentille.

Dispatcher-komponentti on arkkitehtuurin keskipiste, joka hallitsee kaikkea Flux-sovelluksen sisällä kulkevaa tietovirtaa. Sen kautta kulkevat kaikki sovelluksen action-komponentit, jotka välitetään eteenpäin oikeille store-komponenteille. Jotta tämä on mahdollista, täytyy jokaisen store-komponentin rekisteröidä itsensä dispathcher-komponentille. [22]

Store-komponentin tehtävänä on puolestaan ylläpitää ja hallita sovelluksen tilaa ja sovelluslogiikkaa. Toisin kuin dispatcher-komponentteja, store-komponentteja voi Flux-sovelluksessa olla useita, jolloin jokainen niistä vastaa omasta sovelluksen osasta. [22] Esimerkiksi yksi store-komponentti huolehtii sovelluksen käyttäjistä ja tunnistautumisesta, kun taas toinen voi olla vastuussa näkymässä näytettävistä tuotteista. MVC-arkkitehtuurin mallia (*model*) voidaan pitää jokseenkin vastaavana store-komponentille. Erona kuitenkin on, että store-komponentit eivät ole kytköksissä varsinaisiin entiteetteihin tai fyysisiin kokonaisuuksiin, joita esimerkiksi tietokannassa ja palvelinpuolella käsitellään [22].

Käyttäjän kanssa vuorovaikutuksessa ovat säiliötyyppiset view-komponentit eli näkymät. Korkean tason näkymiä kutsutaan controller view -komponenteiksi, jotka käsittelevät tietoa ja luovat action-komponentteja käyttäjän pyyntöjen seurauksena. Ne seuraavat store-komponenttien muutoksia ja hakevat niistä tarvitsemansa tiedot. Controller view -komponenteilla on useita lapsikomponentteja, joille ne välittävät seuraavan tilatiedon `props`-objektina. [22]

4. OFFLINE-TOIMINNALLISUUS WEB-SOVELLUKSISSA

Tässä luvussa käsitellään web-sovelluksien offline-toiminnallisuutta ja kuinka se eroaa niin kutsutusta perinteisestä web-sovelluksesta. Luvun tarkoituksena on esitellä vaihtoehtoisia sekä karkeamman tason että teknisempiä ratkaisuja offline-toiminnallisuuden toteuttamiseksi. Koska offline-toiminnallisuus välittyy web-käyttöliittymän kautta, täytyy se huomioida myös web-käyttöliittymän suunnitteluperiaatteissa, jotka on käsitelty luvussa 4.4. Offline-toiminnallisuus ei ole järjestelmälle itsestäänselvyys, vaan se aiheuttaa myös huomattavia haasteita järjestelmän eri komponenteille. Löydetyt ratkaisut muodostavat tutkimuksen teoreettisen lähestymistavan käytännön ongelmien ratkaisemiseksi.

Offline-tilalla tarkoitetaan tilannetta, kun käyttäjän päätelaitteen Internet-yhteyden laatu on heikko tai sitä ei ole ollenkaan. Perinteisen web-sovelluksen näkökulmasta kyseessä on virhetilanne, joka helposti ajaa sovelluksen käyttäjälle outoon tilaan. Tällöin selain ilmoittaa virheellä yhteyden puutteesta tai näkyviin ilmestyy loputtomuuksiin pyörivä spinner-elementti. Web-sovellus kohtaa offline-tilan esimerkiksi, kun:

- Käyttäjä ajautuu verkon katvealueelle.
- Käyttäjällä ei ole pääsyä verkkoon.
- Palveluntarjoajan verkolla on huono kattavuus.
- Epätavalliset sääolosuhteet huonontavat yhteyden laatua.

Offline-kykyisellä web-sovelluksella tarkoitetaan sovellusta, joka pystyy toiminaan myös vaihtelevissa olosuhteissa ja reagoimaan niihin. Tällaisia offline-kykyisiä web-sovelluksien ja applikaatioiden yhdistelmiä Google on alkanut nimittää PWA-sovelluksiksi (*Progressive Web Application*) [23]. Tarkemmin PWA-sovelluksen piirteet ja vaatimukset on esitellyt LePage [24] artikkelissaan. PWA-sovelluksen tavoitteena on tarjota käyttäjilleen käyttäjäkokemus, joka on:

- helppokäyttöinen
- luotettava
- suorituskykyinen
- riippumaton verkkoyhteydestä.

On kuitenkin huomioitava, että web-sovelluksen kutsuminen PWA-sovellukseksi edellyttää juuri LePagen artikkelin vaatimusten täyttymistä. Muutoin voidaan puhua offline-kykyisistä web-sovelluksista.

4.1 Offline first -suunnitteluperiaate

Kun suunnitellaan offline-kykyisiä web-sovelluksia, voidaan noudattaa offline first -suunnitteluperiaatetta. Offline first -suunnitteluperiaate tarkoittaa oletusta, että suunniteltava web-sovellus on lähtökohtaisesti offline-tilassa. Oletuksen seurauksena sovelluksen toiminnallisuus ja käyttäjälle välittyvä käyttäjäkokemus on suunniteltava uudesta perspektiivistä, kun offline-tila ei voi vain olla virhetilanne kuten aiemmin. [25–28]

Koska käyttäjät yhä useammin hyödyntävät mobiililaitteita web-sovellusten käytössä, on offline-toiminnallisuudella ja käyttäjäkokemuksella väliä. [28] On selvää, että esimerkiksi työkäytössä web-sovelluksien tulisi olla saatavissa, eikä aiheuttaa ylimääräisiä katkoksia työntekoon ja prosesseihin. Jos käyttäjien esimerkiksi täytyy valita kahden kilpailevan palvelun välillä, riippuu se lopulta palvelusta jääneestä vaikutelmasta, käyttäjäkokemuksesta.

Kasten [26] mainitsee offline first -suunnitteluperiaatteen luontaiseksi eduksi käyttäjäkokemuksen ja sovelluksen suorituskyvyn parantumisen. Vaikka verkkoyhteyden laatu vaihtelee, ei web-sovellus toimi yllätyksettömästi. Ideaalisessa tilanteessa web-sovellus voi paikallisesti tallentaa tietoa, mikä mahdollistaa monipuolisen sovelluksen käytön myös offline-tilassa [26]. Feyerke [27] ja Teixeira [28] puhuvat myös välimuistin käytön puolesta, jolloin voidaan parantaa sovelluksen suorituskykyä ja tarjota miellyttävämpi käyttäjäkokemus.

Offline-kykyisen web-sovelluksen käyttäjäkokemus koostuu useasta tekijästä, eikä vain palvelun koetusta nopeudesta ja saatavuudesta. Jotta offline-tila olisi hyödyllinen, sovelluksen tarjoamaa sisältöä tulisi pystyä käyttämään myös ilman verkkoyhteyttä. Ratkaisussa tulisi pystyä myös ennakoimaan käyttäjän mahdolliset toimet, mikä voi perustua esimerkiksi sovelluksen käyttöhistoriaan. Tällöin työ voidaan tehdä taustalla ja tarjota sisältö esimerkiksi suoraan sovelluksen välimuistista. Koska lopullisen vaikutelman luo aina käyttäjä, ei vuorovaikutustilanteiden suunnittelua voi liikaa korostaa. Käyttäjän tulisi aina olla tietoinen verkkoyhteyden tilasta, käytettävissä olevasta sisällöstä ja kuinka eri tilanteissa tulisi toimia. Voi olla hyödyllistä jopa piilottaa toiminnallisuudet, joita ei offline-tilassa ole tarkoitus käyttää. [29, 30]

Offline-kykyisten web-sovellusten tulemisen edellytyksenä on ollut teknologioiden ja erityisesti päätelaitteiden ja selainten kehitys. Kun selaimessa pyörivällä web-sovelluksella on resursseja tiedonvarastointiin ja käsittelyyn, voidaan sovelluslogiikkaa toteuttaa tehokkaasti myös päätelaitteen resursseilla, jolloin offline-toiminnallisuuden kehitys helpottuu. Pilvipalveluiden kehitys ja niiden tarjoamat rajapinnat tuovat myös uusia mahdollisuuksia offline-kykyisille sovelluksille.

4.2 Korkean tason toteutustavat

Lähestytään offline-toiminnallisuuden ratkaisua ensin yksinkertaisesti korkeammalla tasolla, jolloin ei huomioida vielä teknisempiä tekijöitä. Korkean tason ratkaisumalleja on kolme erilaista, jotka Andreu [31] julkaisussaan esittelee:

- lataa kaikki
- välimuistin käyttö
- offline-sisällön valikoiminen.

Kaikki ratkaisumallit ovat konkreettisia ja helposti ymmärrettäviä. Niitä hyödynnetään monissa arkisissakin sovelluksissa, joita useat käyttävät lähes päivittäin.

Lataa kaikki -suunnittelumalli nimensä mukaisesti perustuu siihen, että web-sovellus lataa kaikki tarvitsemansa tiedot kerralla varastoon. Tällöin ladatut tiedot ja resurssit ovat varmasti käytettävissä myös offline-tilassa. Kuitenkaan lataa kaikki -suunnittelumallia ei voida sellaisenaan pitää kovin käyttökelpoisena, vaan se on pikemminkin vältettävä antisuunnittelumalli (*anti pattern*). Ongelmallista suunnittelumallin kanssa on sen hitaus ja skaalautuvuus, kun tiedon ja tarvittavien resurssien tarve kasvaa. [31] Päinvastoin kuin offline first -suunnitteluperiaatteessa sovelluksen suorituskyky hidastuu ja käyttäjäkokemus huonontuu, kun ladataan turhia resursseja ilman tulevaa käyttöä. Erityisen huomattavia ongelmia ilmenee juuri sovelluksen alustuksessa, sekä tiedon päivittämisessä palvelimelle.

Huomattavasti hienostuneempi tapa on käyttää selaimen välimuistia. Tällöin voidaan hyödyntää esimerkiksi localStoragea tai IndexedDB:tä [32, 33], jotka tarjoavat selaimille W3C:n puolesta standardoidun API:n tiedon lokaaliin hallintaan. Yleisesti uusimmat selaimet tukevat näitä standardeja, mutta rajoitukset täytyvät ottaa suunnittelussa huomioon, jos laitekanta on vanhanaikaista. Lataa kaikki -suunnittelumallista poiketen välimuistin käyttö tarjoaa joustavuutta resurssien hallintaan, jolloin sovelluksen käynnistyessä ladataan vain välttämättömät resurssit [31]. Suunnittelumalli parantaa myös käyttäjäkokemusta, kun jo kertaalleen ladatut resurssit ovat nopeasti saatavilla suoraan välimuistista. Välimuistin käyttö on yleistynyt web-sovelluksissa huomattavasti, mutta se sisältää myös haasteita välimuistissa olevan tiedon hallintaan. Suurimmat näistä ovat välimuistin tyhjentäminen ja sen päivittäminen.

Viimeinen suunnittelumalli on offline-sisällön valitseminen, joka siirtää vastuun ja hallinnan offline-tilassa käytettävissä olevista resursseista käyttäjälle. [31] Suunnitteluperiaatetta on hyödynnetty etenkin suoratoisto- sekä karttapalveluissa. Ne tarjoavat usein käyttäjilleen mahdollisuuden valita laitteelle ladattavaa offline-sisältöä, joka on aina käyttäjän käytettävissä. Käyttäjäkokemuksen kannalta ratkaisumalli on erittäin selkeä ja informatiivinen käyttäjälle, joka on itse vastuussa sovelluksensa offline-toiminnallisuuden kattavuudesta. Samalla käyttäjä on tietoinen sisällön latauksen viemästä ajasta ja sen vaatimasta kaistasta.

Koska web-sovellukset voivat koostua hyvin erilaisista osista ja toiminnoista, on sanomattakin selvää, ettei yksittäinen korkean tason suunnittelumalli ratkaise ongelmia. Riippuu täysin sovelluksesta ja sille asetetuista vaatimuksista kuinka esitettyjä malleja kannattaa soveltaa ja yhdistellä. Voi olla täysin järkevää hyödyntää sekä välimuistin käyttöä että offline-sisällön valikoimista tai jopa ladata tietty osa sovelluksesta aina kokonaisuudessaan.

4.3 Matalan tason toteutustavat

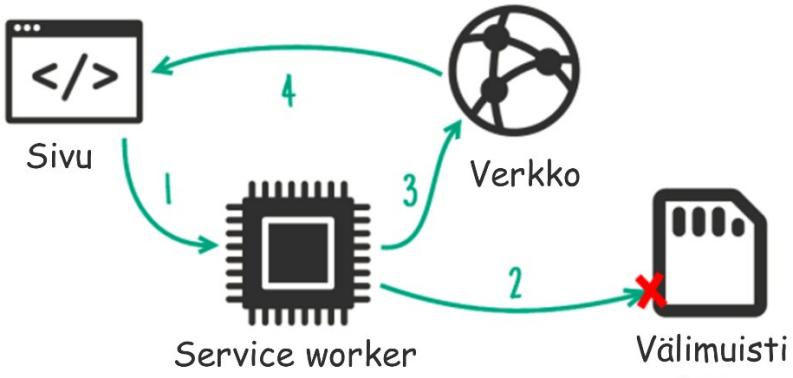
Seuraavaksi käsitellään offline-toiminnallisuuden matalan tason toteutustapoja. Ne tarjoavat teknisiä ratkaisuja, joiden avulla korkean tason toiminnallisuus sekä web-sovellukselta vaaditut offline-vaatimukset voidaan toteuttaa.

4.3.1 Service worker

Service worker on selaimessa taustalla suoritettava JavaScript-ohjelma, joka toimii erillään avoimista web-sivuista sovelluksen ja verkon välissä. Se voidaan vapaasti määritellä W3C:n [34] standardoimalla API:lla, joka tarjoaa työkalut service workerien alustamiseen, rekisteröimiseen ja palvelinpuolen pyyntöjen kaappaamiseen. Service workerit antavat mahdollisuuden määrittää välimuistin käyttöä web-sovelluksessa, toteuttaa push-ilmoituksia sekä hallita tietojen synkronointia standardoidun API:n välityksellä. [34, 35]

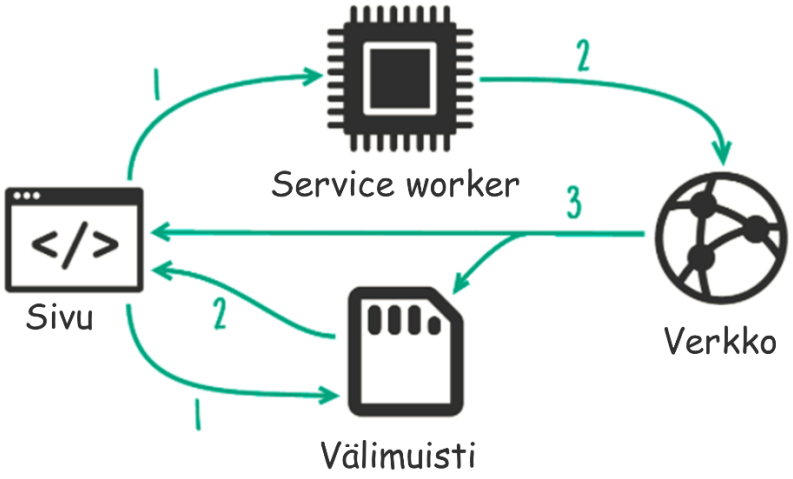
Service workerien avulla web-sovelluksen offline-toiminnallisuus ja käyttäjäkokemus parantuvat. Niillä voidaan määrittää staattisten resurssien tallentamista selaimen välimuistiin, jolloin web-sovellus on käyttäjien saatavilla myös verkkoyhteyden katketessa. Koska service workerit voivat kaapata myös pyyntöjä palvelinpuolelle, ei web-sovellus välttämättä tarvitse verkkoyhteyttä käyttäjän pyynnön toteuttamiseksi. [34, 35] Service workerien monikäyttöisyyden vuoksi voidaan web-sovelluksen offline-toiminnallisuus toteuttaa usealla eri mallilla. Archibald [36] korostaa kuitenkin, että service workerit eivät itsessään tee web-sovelluksista offline-kykyisiä. Ne tarjoavat kehittäjille ongelmien ratkaisemiseksi työkaluja, joita voidaan soveltaa tapauskohtaisesti. Koska usein web-sovelluksen eri toimintojen offline-vaatimukset vaihtelevat, voidaan koko web-sovelluksen offline-toiminnallisuus rakentaa useaa eri mallia hyödyntäen.

Kaksi esimerkillistä toteutustapaa on esitetty kuvissa 6 ja 7. Kuvassa 6 on esitelty malli, jossa service worker käyttää ensin välimuistia ja sitten verkkoyhteyttä.



Kuva 6. *Service workerin käyttö, kun hyödynnetään ensisijaisesti välimuistia. [36]*

Kuvan 6 malli perustuu välimuistin hyödyntämiseen ensisijaisena tietolähteenä, jolloin kaikki välimuistissa olevat resurssit ovat käyttäjien käytössä myös offline-tilassa. Jos pyyntöä ei voida täyttää välimuistin avulla, ohjataan pyyntö palvelinpuolelle. Esitelty ratkaisumalli toimii useasti pohjana muille web-sovellusten offline-ratkaisumalleille, jotka ovat monesti vain poikkeuksia kuvan 6 malliin. [36] Kuvassa 7 on esitelty malli, jossa hyödynnetään samanaikaisesti välimuistia ja verkkoyhteyttä.



Kuva 7. *Service workerin käyttö, kun hyödynnetään samanaikaisesti välimuistia ja verkkoyhteyttä. [36]*

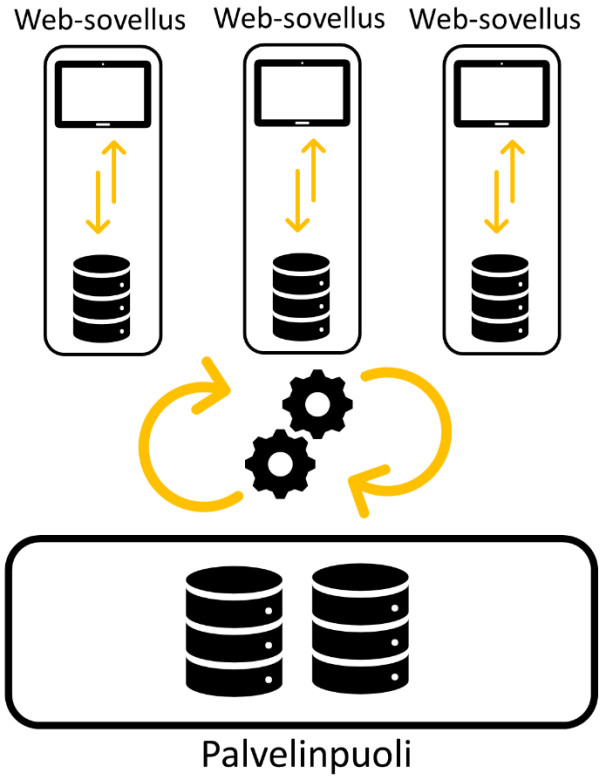
Kuvan 7 mukaisesti mallissa tehdään kaksi samanaikaista pyyntöä sekä välimuistiin että palvelinpuolelle. Ideaalitilanteessa voidaan näyttää ensin välimuistiin tallennettu sisältö ja päivittää sitä, kun palvelinpuolelta saadaan uutta tietoa. Ratkaisumalli on erityisen hyvä tilanteisiin, kun web-sivun sisältö päivittyy usein. [36]

4.3.2 Web-käyttöliittymän ja palvelimen välinen synkronointi

Offline-kykyisen web-sovelluksen edellytyksenä melkein poikkeuksetta on välimuistin käyttö sovelluksen tilan ja resurssien hallntaan. Selaimen tarjoaman localStoragen ja In-

dexedDB:n lisäksi tietojenvarastointiin on kuitenkin myös muita ratkaisuja. Teixeira [28] esittelee yhdeksi tavaksi web-sovelluksen ja palvelimen välisen tietojen synkronoinnin, joka perustuu selaimessa olevan tietokannan replikoimiseen palvelinpuolelle.

Esitellyssä arkkitehtuurissa jokaisella web-sovelluksella on oma tietokanta, joka sisältää kaikki käyttäjän tarvitsemat ja hallitsemat tiedot. Tällöin ensisijaisesti päätelaitteen tietokannassa olevaa tietoa pidetään oikeana. [28] Kuvassa 8 on esitetty arkkitehtuurikuva, jossa jokaisella web-sovelluksella on oma erillinen tietokanta. Kun ei olla offline-tilassa, synkronoidaan jokaisen käyttäjän muutokset palvelinpuolen tietokantaan.



Kuva 8. *Arkkitehtuurikuva web-sovellusten paikallisten tietokantojen synkronoinnista palvelimelle, mukaillen lähteestä [28].*

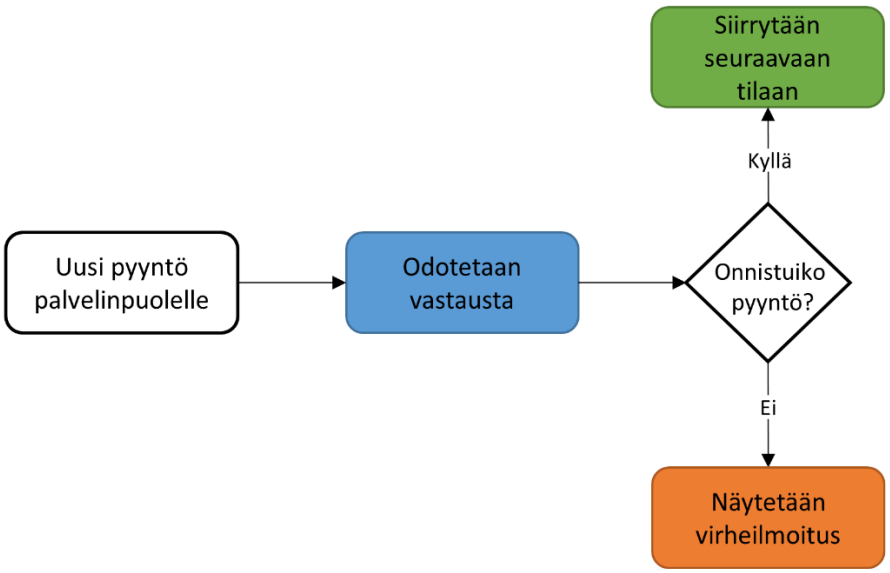
Kuvan 8 mukaisesti synkronointi tapahtuu palvelinpuolella, jossa huolehditaan mahdollisista konfliktitilanteista ja niiden ratkaisujen välittämisestä takaisin web-käyttöliittymän tietokantaan. Tällöin web-käyttöliittymän vastuulle jää oman tilansa päivitys, kun palvelinpuolelta on toimitettu uutta tietoa selaimen tietokantaan tai sovellus esimerkiksi alustetaan. [28]

Web-sovelluksen ja palvelinpuolen synkronoinnin toteuttamiseen voidaan hyödyntää esimerkiksi Pouch- ja CouchDB:tä. PouchDB tarjoaa selainkohtaisen paikallisen tietokannan, joka voidaan synkronoida CouchDB-palvelimelle tai muuhun yhteensopivaan tietokantaan. Tietokannan on kuitenkin tuettava CouchDB:n replikointiprotokollaa synkronoinnissa. [28]

4.4 Web-käyttöliittymän offline-suunnittelumallit

Koska offline-toiminnallisuus linkittyy vahvasti web-käyttöliittymän luomaan käyttäjäkokemukseen, on käyttöliittymän pystyttävä ohjaamaan käyttäjää luontevasti tilasta toiseen. Käyttäjäkokemukseen vaikuttaa vahvasti tilasiirron tai sisällön päivityksen nopeus. Tavoitearvoina voidaan pitää käyttäjäkeskeisen RAIL-mallin [37] viitearvoja, jolloin käyttäjän toimiin tulisi pystyä reagoimaan alle 100 millisekunnissa ja suorittamaan palvelinpuolelle tehtävä pyyntö alle sekunnissa. Kun mainitut rajat ylitetään, käyttäjä kokee web-käyttöliittymän hitaaksi ja turhautunee pyyntöjen suorituksen hitauteen [37].

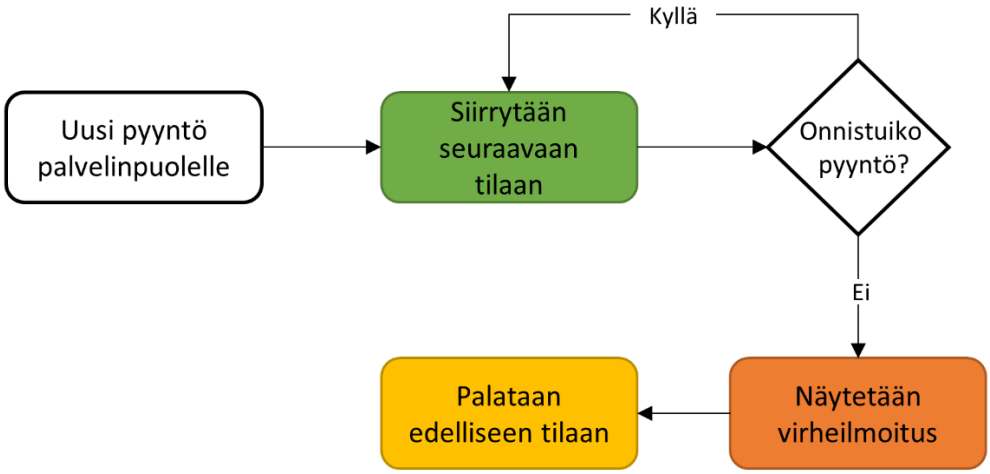
Perinteisesti web-käyttöliittymät on suunniteltu päivittyvän pessimistisesti käyttäjän pyyntöjen seurauksena. Web-käyttöliittymän pessimistinen päivitys on havainnollistettu kuvassa 9. Pessimistisen mallin lähtökohtana on odottaa, kunnes palvelinpuolelta saadaan vastaus käyttäjän tekemään pyyntöön.



Kuva 9. *Web-käyttöliittymän pessimistinen päivitys.*

Kuvan 9 mukaisesti onnistuneen pyynnön seurauksena voidaan siirtyä seuraavaan tilaan, kun taas pyynnön epäonnistuessa ilmoitetaan vain virheestä käyttäjälle. Pessimistinen päivitys hidastaa helposti käyttöliittymää, kun odotetaan vastauksia palvelinpuolelta.

Mallin vastakohtana on web-käyttöliittymän päivitys optimistisesti, kuten Mishunov [38] esittää. Optimistinen malli suhtautuu tehtäviin palvelinpuolenpyyntöihin optimistisesti, eikä oleta niiden epäonnistuvan. Optimistinen malli web-käyttöliittymän päivitykseen on esitetty kuvassa 10. Mallissa siirrytään suoraan seuraavaan tilaan, eikä odoteta palvelinpuolen vastausta tehtyyn pyyntöön. Jos kuitenkin pyyntö epäonnistuu, täytyy web-käyttöliittymän ilmoittaa siitä käyttäjälle ja palata edelliseen tilaansa.



Kuva 10. *Web-käyttöliittymän optimistinen päivitys.*

Optimistisen mallin etuina ovat nopea reagointi käyttäjän toimiin sekä pyynnöissä mahdollisesti tapahtuvien virheiden hallinta. [38] Malli sopii hyvin offline-kykyisten web-sovellusten toteutukseen, koska se mahdollistaa sovelluksen käyttämisen, vaikka verkkoyhteyttä ei olisi. Mallin käyttöön on myös selkeitä rajoituksia, sillä monimutkaiset toiminnallisuudet vaativat usein poikkeuksetta palvelinpuolen resursseja. Jotta optimistisen mallin edut saadaan hyödynnettyä, on käytetyn API:n vakaudella ja suorituskyvyllä iso merkitys lopputulokseen [38].

4.5 Offline-toiminnallisuuden aiheuttamat haasteet

Offline-kykyiset web-sovellukset auttavat parantamaan käyttäjäkokemusta sekä tehostamaan suorituskykyä, mutta ne tuovat kuitenkin mukanaan uusia haasteita. Näitä haasteita perinteisten web-sovelluksien kanssa ei tarvitse ottaa huomioon, kun käyttäjä luonnostaan on tietoinen, ettei sovellus toimi ennen verkkoyhteyden palautumista.

Koska kaikki offline-toiminnallisuuden ratkaisut perustuvat jotenkin selaimen tai päätelaitteen välimuistin käyttöön, ei sen aiheuttamilta ongelmilta voida koskaan täysin välttyä. Andreu [31] muistuttaa, että offline-toiminnallisuuden ratkaisu on pohjimmiltaan aina kompromissi muistin käytön ja verkkoyhteyden välillä. Tämä on syytä huomioida etenkin mobiililaitteiden kohdalla, joiden muisti on rajallinen ja verkkoyhteyden laatu vaihteleva. Rajallisten resurssien vuoksi tehokkaiden ja suorituskykyisten API:en käyttö on ehdotonta, jolloin turhaan sisältöön ei käytetä verkkoyhteyden kaistaa tai välimuistin kapasiteettia.

Bricklinin [39] ja Feyerken [27] mukaan tyypillisin ongelmatilanne syntyy, kun selaimen välimuistiin on tallennettu uutta sisältöä, joka täytyisi saada synkronoitua myös palvelinpuolelle. Haasteeksi tällöin voivat koitua:

- Käyttäjän tekemien paikallisten muutosten menettäminen.
- Muutosten välittäminen palvelinpuolen tietovarastoon.
- Konfliktitilanteiden syntyminen synkronoinnin aikana.

Pahimmassa tapauksessa käyttäjä menettää kaikki tekemänsä paikalliset muutokset, kun web-sovellus ei pysty pitämään välimuistissa tilaansa. Web-sovelluksen tulisikin pystyä palauttamaan tilansa paikallisesta välimuistista tarpeen vaatiessa. Eväkallio [40] kuitenkin tuo esille, että joustavan ja monipuolisen välimuistin hallinta on service workerien avulla vaikeaa. Välimuistia käytettäessä on muistettava myös tietoturvallisuus, jolloin esimerkiksi selaimen välimuistin tiedot eivät saa päätyä vääriin käsiin [39].

Muutosten välittäminen selaimen paikallisesta tietokannasta palvelinpuolelle kuulostaa helpolta vaatimukselta. Usein palvelinpuolen tietokannat ovat erityyppisiä kuin selaimessa tai pilvipalvelussa. Etenkin suuremmissa sovelluksissa ja ratkaisuissa ei välttämättä ole mahdollista edes käyttää välikerroksia tai tietokantoja, jotka huolehtivat selaimen ja palvelinpuolen synkronoinnista. [39, 40] Synkronointi on kyettävä toteuttamaan myös toiseen suuntaan, jolloin palvelinpuolelta voidaan päivittää uutta tietoa käyttäjille [39]. Tämä sopii SPA-sovellusten ajatusmaailmaan, jolloin voidaan päivittää vain välttämätön tieto käyttäjälle ilman sivun uudelleenlatausta.

Kun jokaisen käyttäjän muutokset on tallennettu paikallisesti selaimen välimuistiin, ovat synkronoinnissa tapahtuvat konfliktitilanteet todennäköisiä. Niiden ratkaiseminen voidaan tehdä automaattisesti, jos tietokantojen synkronointi sitä tukee, tai jättää se vastaavasti käyttäjän vastuulle. [27, 39] Eväkallion [40] mukaan on syytä muistaa, että synkronointi on yksinkertaisissa tapauksissa helppoa, mutta monimutkaisen toiminnallisuuden toteutus on hyvin haasteellista. Eväkallio korostaa myös, ettei erikoisratkaisuihin nojautuminen palvelinpuolella ole hyvä lähtökohta koko järjestelmän arkkitehtuurin pohjaksi.

Vähemmän tekninen mutta sitäkin monimutkaisempi haaste offline-kykyisille web-sovelluksille ovat itse käyttäjät. Vaikka web-sovellus hyödyntäisi välimuistia ja palvelinpuolelle synkronointia, on sekin hyödytöntä, jos käyttäjä toimii ennakoimattomasti. Kun web-sovellus toimii selaimessa, niin mikä estää käyttäjää sulkemasta selainta, tyhjentämästä välimuistia tai lataamasta sivua uudelleen kesken transaktion. Lopputuloksena syntyy tila, jota ei voida ennustaa tai jonka tiedon eheydestä ei voida varmistua. Tällöin vastuu web-sovelluksen käytöstä jää käyttäjälle.

5. ASIAKASPROJEKTIN TAUSTA

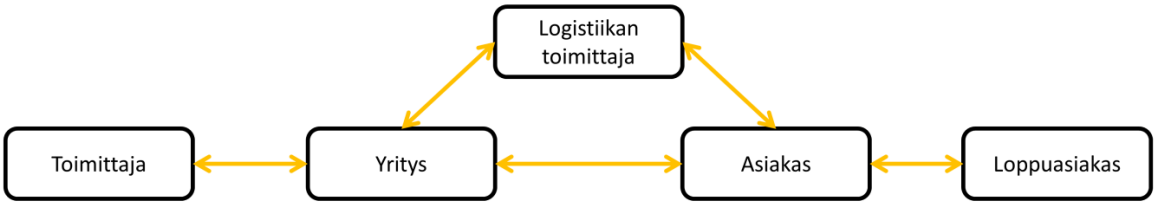
Tässä luvussa esitellään asiakasprojektin tausta, johon tämä tutkimus kohdistuu. Toimintatutkimuksen mukaisesti ongelman omistajat osallistuvat kehitysprosessiin määrittelyn, palautteenannon, testauksen ja validoinnin myötä. Yhdessä asiakkaan kanssa tunnistetuista verkkoyhteyteen liittyvistä ongelmista on muodostettu offline-vaatimusmäärittely. Tehtyä vaatimusmäärittelyä hyödynnetään teorian pohjalta sovellettujen ratkaisujen validoinnissa, joka on oleellista iteratiiviselle kehitysprosessille.

5.1 Logistiikka ja tilaus-toimitusketju

Ballou [41] määrittelee logistiikan joukoksi funktionaalisia toimintoja, jotka yhdessä muodostavat tilaus-toimitusketjun (*supply chain*) raaka-aineiden alkulähteiltä loppuasiakkaalle asti. Kahdensuuntaisen materiaalivirran lisäksi prosessissa liikkuu myös tietoa toimijoiden välillä. Mentzer et al. esittelevät yksinkertaisen tilaus-toimitusketjun, joka on kuvattu kuvassa 11. Se koostuu toimittajasta, loppuasiakkaasta ja yrityksestä näiden välissä.

Kuva 11. *Suora tilaus-toimitusketju, mukaillen lähteeseen [42].*

Kuvan 11 mukaisesti yritys tilaa toimittajalta raaka-aineita tai osatuotteita omaa liiketoimintaprosessiaan varten. Valmiita tuotteita tai palveluita ostavat asiakkaat, joille yritys niitä toimittaa tilausten mukaisesti. Aina prosessit eivät ole näin suoraviivaisia, vaan ne sisältävät kolmansia osapuolia [42]. Kuvassa 12 on esitetty hieman monimutkaisempi tilaus-toimitusketju, jossa yrityksen ja loppuasiakkaan välinen logistiikka on kolmannen osapuolen vastuulla.



Kuva 12. *Kolmansia osapuolia sisältävä tilaus-toimitusketju, perustuen lähteeseen [42].*

Logistiikan ainoana tehtävänä on luoda asiakkaalle arvoa, joka syntyy toimittamalla tilatut tuotteet oikeaan paikkaan sovittuna aikana. Balloun mukaan logistiikkaprosessin avaintoimintoja ovat:

- asiakaspalvelun- ja palvelutasonhallinta
- kuljetustenhallinta
- saldon- ja varastonhallinta
- tietovirran- ja tilaustenhallinta.

Avaintoimintoja tuetaan muilla organisaation tai kolmansien osapuolien tukipalveluilla, joita ovat esimerkiksi varastointi, materiaalinhallinta, osto sekä tiedonhallinta. Koska logistiikka liittyy lähes jokaisen yrityksen liiketoimintaan, ovat sen kustannukset huomattavia ja otettava suunnitelmissa huomioon. Lisäksi on tärkeää tietää asiakkaan asettamat toimituskohtaiset vaatimukset, joiden täyttyminen heijastuu nopeasti asiakastyytyväisyyteen. [41]

5.2 Tietojärjestelmän rooli logistiikassa

Tieto on valtaa myös logistiikassa ja tilaus-toimitusketjun hallinnassa, jossa painottuvat tiedonkeruu sekä käsittely ja jalostetun tiedon jakaminen yrityksen ja muiden sidosryhmien välillä. [43] Jotta tämä on mahdollista, vaaditaan toimivia tietojärjestelmiä, joita ovat esimerkiksi ERP- (*Enterprise Resource Planning),* MES- (*Manufacturing Execution System*) ja WMS-järjestelmät (*Warehouse Management System*).

Digitalisaation vaikutukset ovat merkittäviä myös logistiikan alalla, kun turha manuaalinen työ sekä paperien käsittely vähenevät ja prosesseista tulee käyttäjäystävällisempiä. Fawcett et al. [43] painottavat myös uusien teknologioiden tuomia mahdollisuuksia, joiden avulla yrityksen prosesseja voidaan tehostaa. Onnistuessaan kaikki voittavat. Yritys saavuttaa kilpailuetua muihin nähden, prosessien tehokkuus ja tarkkuus parantuvat sekä asiakkaat ovat tyytyväisempiä. [44]

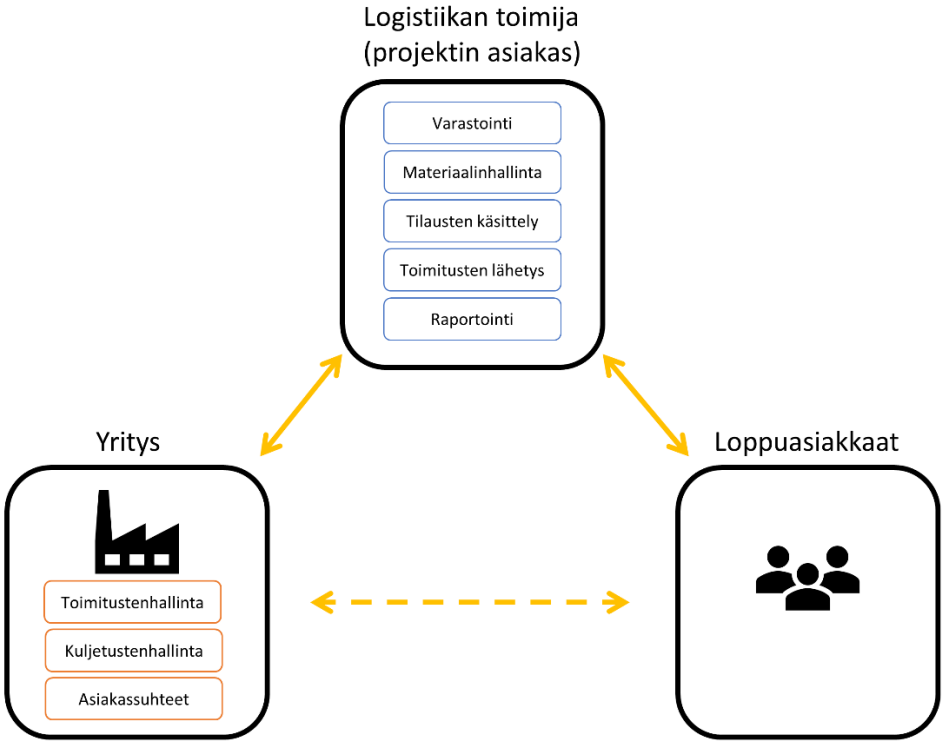
Laajalti hyödynnettyjä teknologioita materiaalinkäsittelyssä ovat RFID- (*Radio Frequency Identification*) ja viivakoodinlukijat, joita työntekijät käyttävät omilta päätelaitteiltaan. Ne mahdollistavat tuotteiden yksilöimisen ja läpinäkyvän sekä katkeamattoman jäljitys- ja seurantaketjun jokaiselle kappaleelle. Työntekijälle teknologiat tuovat helpo-

tusta ja apua työtehtävien suoritukseen. Esimerkiksi keräyksessä päätelaite voi opastaa ja neuvoa työntekijää keräysmäärissä ja varastopaikoissa. Päätelaite voi pyytää myös käyttäjää kuittaamaan kerättyjä tuotteita tai varastopaikkoja lukijan avulla, jolloin inhimilliset virheet voidaan minimoida jo keräystasolla. [43, 44]

5.3 Asiakkaan logistiikkaprosessi

Projektin asiakas on osa Metzner et al. [42] esittelemää monimutkaisempaa tilaustoimitusketjua, jossa se vastaa logistiikasta yrityksen ja loppuasiakkaiden välillä. Kolmannen osapuolen logistiikan toimijan vastuut on hyvin tarkasti rajattu projektin asiakkaan toimipaikan sisäpuolelle, eikä sen vastuulla ole esimerkiksi kuljetusten suunnittelua ja hallintaa. Yrityksellä on itsellään myös varastoja, mutta kaikki loppuasiakkaiden toimitukset hoidetaan logistiikan toimijan ja sen varastojen kautta.

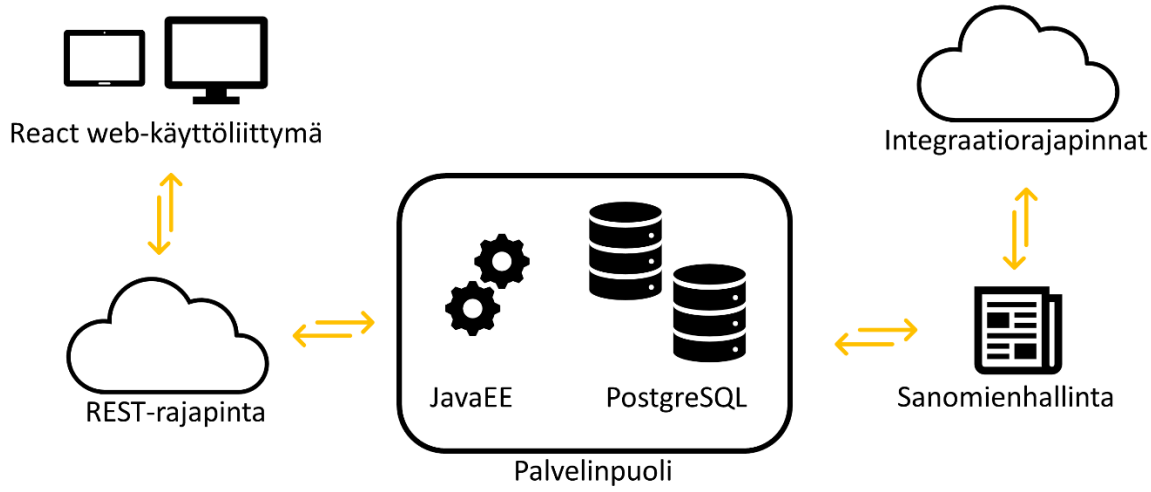
Kuvassa 13 on esitetty projektin asiakkaan logistiikkaprosessi ja toimijoiden kesken jaetut vastuut. Tuotteita valmistava yritys on vastuussa asiakassuhteista ja yhteydenpidosta loppuasiakkaiden kanssa. Saatujen tilausten perusteella yritys suunnittelee ja hallitsee kuljetuksia sekä toimituksia. Kolmannen osapuolen logistiikan toimijan vastuulle jää sille toimitettujen kappaleiden varastointi, materiaalinhallinta, yritykseltä saatujen tilausten käsittely, toimitusten lähetys ja raportointi.



Kuva 13. *Projektin asiakkaan logistiikkaprosessi ja toimijoiden vastuunjako. Yhdistetyt materiaali- ja tietovirrat on kuvattu yhtenäisillä viivoilla, kun taas pelkät tietovirrat on kuvattu katkoviivoin.*

5.4 Järjestelmän arkkitehtuuri

Järjestelmän arkkitehtuuri mukailee aiemmin esitettyä web-sovelluskehystä, johon on liitetty sanomapalvelut muiden järjestelmien integraatioiden vuoksi. Kuvassa 14 on esitetty karkean tason kuva järjestelmän arkkitehtuurista. Järjestelmän web-käyttöliittymä on toteutettu React.js- ja Redux-kirjastoilla, jotka esitellään tarkemmin luvussa 6. Se on yhteydessä REST-rajapinnan avulla JavaEE-pohjaiseen palvelinpuolen sovellukseen, joka hyödyntää avoimen lähdekoodin PostgreSQL-tietokantoja. Integraatiorajapinnat yrityksen muihin järjestelmiin, kuten toiminnanohjausjärjestelmään, on toteutettu sanomapalveluilla, jotka välittävät XML-sanomia (*Extensible Markup Language*) järjestelmien välillä.



Kuva 14. *Karkean tason arkkitehtuurikuva järjestelmästä.*

Toiminnanohjausjärjestelmän integraatiorajapinnan kautta logistiikan toimija saa tiedon uusista tilauksista, saapuvista sekä lähtevistä kuljetuksista. Sanomapalvelut hoitavat myös tilausten kuittauksen sekä muiden määriteltävien tapahtumien raportoimisen takaisin yrityksen toiminnanohjausjärjestelmään.

5.5 Web-käyttöliittymän käyttäjät

Asiakkaan järjestelmällä on kaksia erilaisia käyttäjiä: työnjohtajia ja työkoneenkuljettajia. Työnjohtajien tehtävänä on suunnitella tilausten sekä toimitusten toimenpiteet, jotka hän koordinoi työkoneenkuljettajille. Työnjohtajien vastuulla on myös raportointi ja yhteistoiminta muiden prosessin sidosryhmien kanssa. Työkoneenkuljettajat suorittavat materiaalinhallinnan työtehtäviä tilauksiin ja toimituksiin liittyen. He käyttävät työssään trukkeja tai pyöräkuormaajia, joille he lastaavat sekä purkavat kuljetuksia ja järjestävät varastoa. Huomionarvoista on, että jokainen käsiteltävä kappale on yksilöity RFID-tunnisteella.

Kaikilla käyttäjillä on käytössään sama web-käyttöliittymä, jossa on mahdollisuus suorittaa kirjautuneelle käyttäjälle sallittuja toimintoja. Esimerkiksi työkoneenkuljettaja ei pääse käsiksi tilausten hallintaan, vaan hän näkee vain heidän työjonon ja siihen liittyviä muita tukitoimenpiteitä.

Työkoneen web-käyttöliittymä on entuudestaan jo toteutettu SPA-sovelluksena, joka ei ole luonnostaan offline-kykyinen. Kuvassa 15 on esitelty hahmotelma työkoneen web-käyttöliittymästä. Aloitusnäkymässä esitetään saapuvien kuljetusten purkutehtävät ja lähtevien kuljetusten lastaustehtävät, joita työntekijöiden on mahdollista suorittaa.

Kuva 15. *Karkea kuva työkoneen web-käyttöliittymästä. Vasemmalla näkymä työtehtävän valitsemiselle ja oikealla näkymä valitun työtehtävän suorittamiselle.*

Kun valitaan web-käyttöliittymän listasta haluttu työtehtävä, vaihdetaan tehtäväkohtaiseen näkymään. Tehtäväkohtaisessa näkymässä luetaan RFID-tunnisteilla varustettuja kappaleita kyytiin. Koska jokaiselle käsiteltävälle kappaleelle on olemassa suunnitelma, näytetään työkoneen kuljettajalle suunnitelman mukainen kohdesijainti, jonne poimitut kappaleet tulee kuljettaa. Näkymän alalaidassa esitetään lisäksi työtehtävän nimikerivien tilanteet.

Työkoneenkuljettajat työskentelevät yrityksen suuressa varastossa sekä viereisellä lastaus- ja vastaanottoalueella. He käyttävät web-käyttöliittymää omilta tableteiltaan yrityksen yksityisessä WLAN-verkossa (*Wireless Local Area Network*). Työkoneenkuljettajien kaksi tärkeintä työtehtävää ovat saapuvan kuljetuksen purkaminen ja lähtevän kuljetuksen lastaaminen. Työtehtävät ovat hyvin samankaltaisia keskenään. Työprosessin kannalta ainoa eroavaisuus on saapuvista kuljetuksista ennakkoon saadut rahtikirjat, jotka sisältävät tiedot kuljetuksen RFID-tunnuksin yksilöidyistä kappaleista. Varastosta kappaleita noudettaessa on puolestaan varmistuttava, että kyseinen kappale todellakin löytyy kyseiseltä varastopaikalta.

5.6 Web-käyttöliittymän offline-vaatimukset

Verkon oletetaan olevan pääosin käytössä asiakkaan toimipisteen alueella, mutta pieniä katkoksia voi aina esiintyä. Koska työnjohtajat tekevät työtä toimistosta käsin perinteisillä pöytäkoneilla, ei heidän näkymillään ole erityisiä offline-vaatimuksia. Tästä syystä offline-toiminnallisuus ja sen vaatimukset kohdistuvat työkoneenkuljettajien työtehtäviin ja näkymiin.

Yhdessä asiakkaan kanssa on tunnistettu seuraavat mahdolliset verkkoyhteyteen liittyvät ongelmatapaukset:

- Suuret kappalemäärät ja korkeat varastopaikat aiheuttavat katvealueita varaston sisällä.
- Verkkoyhteys on heikompi ulkona sijaitsevalla lastaus- ja vastaanottoalueella.
- Verkkoyhteys voi pätkiä kuljetusyksikköjen sisällä, kun kappaleita lastataan tai puretaan.

Jotta havaitut ongelmat eivät häiritse työkoneenkuljettajien työntekoa tai laske logistisen prosessin tehokkuutta, on web-sovelluksen offline-toiminnallisuus tarpeen. Web-sovelluksen merkittävimmät toiminnalliset offline-vaatimukset ovat:

- Kuljetuksen purkaminen ilman verkkoyhteyttä.
- Kappaleiden keräys varaston katvealueella.
- Ilmoitus käyttäjälle mahdollisesta offline-tilasta.

Etenkin ensimmäistä vaatimusta voidaan pitää hyvin tärkeänä koko varaston toiminnan kannalta. Saapuvat kuljetukset on saatava purettua vaivatta, jotta jonoa ei lastausalueelle synny ja kuljetukset pääsevät suorittamaan seuraavia tehtäviään. Lisäksi käyttäjien on oltava aina tietoisia, jos web-sovellus on offline-tilassa. Web-käyttöliittymän on myös pysyttävä käyttökelpoisena, vaikka web-käyttöliittymä ladattaisiin uudelleen offline-tilassa.

6. ASIAKASPROJEKTISSA KÄYTETYT MODERNIT WEB-TEKNIIKAT

Koska varsinainen asiakasprojekti oli aloitettu jo ennen tutkimusta, oli siinä käytettävät web- ja ohjelmointitekniikat jo päätetty ja lukittu. Tässä luvussa perehdytään tarkemmin asiakasprojektissa käytettyihin web-tekniikoihin, jotka samalla ohjaavat sekä rajaavat tutkimuksen kehitysprosessia. Web-käyttöliittymät on rakennettu React.js-JavaScript-kirjastolla. Web-sovelluksen tietovirran hallinnassa on hyödynnetty Facebookin Flux-arkkitehtuuriin pohjautuvaa Reduxia.

6.1 React.js

React.js on JavaScript-kirjasto, joka tarjoaa työkaluja web-käyttöliittymien rakentamiseen. Reactin ajatus perustuu komponentteihin ja niiden hyödyntämiseen web-käyttöliittymissä. Komponenttipohjaisuuden ehdoton etu on saavutettavissa virtuaalisen DOM:in avulla. Se mahdollistaa päivitysten tekemisen varsinaiseen DOM:iin tehokkaasti, kun näkymä saadaan päivitettyä optimoidusti vain tarvittavien komponenttien osalta [45].

Reactissa käsitellään sekä elementtejä että komponentteja. Abramovin [46] mukaan Reactin elementit ovat yksinkertaisia objekteja, jotka kertovat DOM:in avulla kuinka ne tulisi käyttäjälle esittää. Kun elementti on luotu, ei sitä ole tarkoitus muuttaa kuin tekemällä kokonaan uusi. Reactin komponentti puolestaan on monimutkaisempi ja se voi sisältää toimintalogiikkaa. Yksinkertaisissa toteutuksissa komponenttikin voidaan toteuttaa funktionaalisesti, jos monimutkaista toimintalogiikkaa komponentilta ei vaadita.

6.1.1 React-komponentit

Web-käyttöliittymän jakaminen useisiin komponentteihin parantaa toteutuksen uudelleenkäytettävyyttä, kun komponenteilla on yksinkertaiset vastuut ja tehtävät. Yksi keino yksinkertaisuuden saavuttamiseksi on funktionaalisuus, jolloin voidaan hyödyntää funktionaalisen ohjelmoinnin tuomia etuja. Reactin komponenttien vähimmäisvaatimuksena on toteuttaa rajapinta `render()`-metodille, joka palauttaa näkymälle komponentista näytettävän osan.

React hyödyntää JSX-syntaksia, joka on XML-merkkikieltä muistuttava laajennus JavaScriptille. JSX-syntaksi mahdollistaa HTML-elementtien käsittelyn JavaScriptin sisällä. [47] Esimerkki JSX-syntaksista on esitetty ohjelmassa 4. Ohjelma luo funktionaa-

lisesti parametrina annetun `props`-objektin avulla `div`-elementin, jonka otsikon arvoksi annetaan tuotteen nimi.

```
const ProductHeader = props => (
  <div>
    <h1>
     {props.name}
    </h1>
  </div>
);
```

Ohjelma 4. *JSX-syntaksilla toteutettu funktionaalinen HTML-elementti.*

JSX-syntaksi voi kuitenkin palauttaa vain ainoastaan yhden HTML-elementin, jolloin moniosaiset komponentit on kapseloitava esimerkiksi `div`-elementtiin, kuten ohjelmassa 4. JSX-syntaksin käyttäminen on Reactin kanssa suositeltavaa muttei kuitenkaan pakotettua. [47]

Komponentit voivat olla ohjelmaa 4 muistuttavia funktionaalisia komponentteja tai ne voivat olla luokkia (*class*). Kaikki React-komponentit saavat tarvitsemansa tiedon vanhemmaltaan (*parent*) `props`-objektina, jota ne eivät pysty muokkaamaan. Komponentit toimivat puhtaina funktioina, joilla ei ole sivuvaikutuksia, kuten luvussa 2.3 on mainittu. [48] Jos komponentti vaatii monimutkaisempaa toimintalogiikkaa, kannattaa se toteuttaa luokkana. Luokka voi hallita omaa tilaansa (*state*), jota komponentin toimesta voidaan muuttaa sen elinkaaren aikana [49]. Esimerkki luokkamaisesta `ProductView`-komponentista on esitetty ohjelmassa 5.

```
class ProductView extends React.Component {

  constructor(props) {
    super(props);
    this.state = { showReviewModal: false };
  }

  showModal() {
   this.setState({ showReviewModal: true});
  }

  componentDidMount() {
    this.props.requestProductReviews();
  }

  sendReview(review) {
    this.setState({ showReviewModal: false});
    this.props.sendReview(review);
  }
```

```
20      render() {
          return (
22          <div>
              <ProductHeader
24              name={this.props.product.name}
              />
26            <ProductInfo
                product = {this.props.product}
28              openModal = {this.showModal()}
              />
30            <ProductReviewModal
                show = {this.state.showReviewModal}
32              sendReview = {review => this.sendReview(review)}
              />
34          </div>
          );
36      }
      }
```

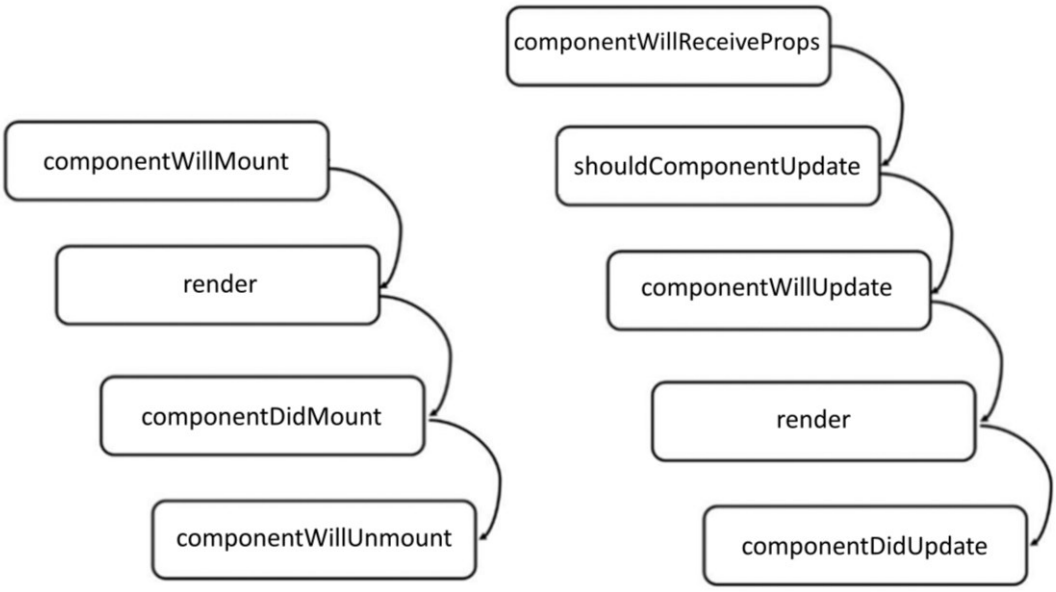
Ohjelma 5. *Luokkana toteutettu React-komponentti.*

Ohjelma 5 kuvaa React-komponenttia, joka luo näkymän tuotteelle rakentuen itse `ProductHeader`-, `ProductInfo`- ja `ProductReviewModal`-komponenteista. `ProductHeader`-komponentille välitetään tuotteen nimi `props`-objektina, kun taas `ProductInfo`-komponentille tuotteen tiedot. Tuotteelle voidaan myös antaa erillinen arvostelu ponnahdusikkunassa, joka voidaan kutsua näkyviin komponentin `showModal`-metodilla. Ponnahdusikkunan hallintaan luokka puolestaan hyödyntää paikallista tilaansa.

`ProductReviewModal`-komponentille välitetään `sendReview()`-funktio arvosteluiden tallentamiseen sekä komponentin näkyvyys. `ProductView`-komponentin `requestProductReviews()`-metodin tehtävänä on puolestaan hakea komponentille kyseisen tuotteen aiemmat arvostelut, jotta ne voidaan näyttää käyttäjälle. `ProductView`-komponentti alustetaan sen rakentajassa (*constructor*) luonnin yhteydessä. Ohjelmassa esiintyvä `componentDidMount()`-metodi esitellään tarkemmin luvussa 6.1.2, jossa käsitellään React-komponenttien elinkaaren hallintaa.

6.1.2 React-komponenttien elinkaari

React-komponenttien elinkaari koostuu niiden rakentamisesta, päivittämisestä ja poistamisesta. Komponenttipohjaisen ajattelutavan tukemiseksi React tarjoaa komponenteilleen elinkaarimetodeita komponenttien hallitsemiseen niiden elinkaaren eri vaiheissa. Kuvassa 16 on esitetty React-komponentin rajapinnan elinkaarimetodit ja niiden suoritusjärjestys suhteessa komponentin renderöimiseen (*render*). Kuvan vasen puoli kuvaa komponentin rakentamisen ja poistamisen elinkaarta, kun taas oikea puoli esittää komponentin päivityksen elinkaarta.



Kuva 16. *React-komponentin elinkaarimetodit. Vasemmalla on esitetty komponentin luominen ja poistaminen, kun taas oikealla komponentin päivitys. [45]*

Koska osa elinkaarimetodeista suoritetaan ennen komponentin renderöimistä ja osa vasta sen jälkeen, mahdollistavat ne esimerkiksi yksittäisen komponentin tilan tai useamman komponentin välisen yhteistoiminnan hallitsemisen. React-komponentin elinkaarimetodit sekä niiden kuvaukset on esitelty taulukossa 1.

Taulukko 1. *React-komponentin elinkaarimetodit ja niiden kuvaukset. [45, 50]*

Metodi	**Kuvaus**
componentWillMount()	Ensimmäinen elinkaarimetodi, joka suoritetaan ainoastaan kerran komponenttia alustettaessa ennen kuin se renderöidään käyttäjälle. Metodissa voidaan muokata komponentin tilaa, mutta se ei kuitenkaan aiheuta komponentin uudelleen renderöintiä.
componentDidMount()	Metodi suoritetaan, kun komponentti on renderöity osaksi sovelluksen DOM:ia. Esimerkiksi REST-rajapintakutsujen sekä DOM:in käsittely on hyvä tehdä tässä metodissa.

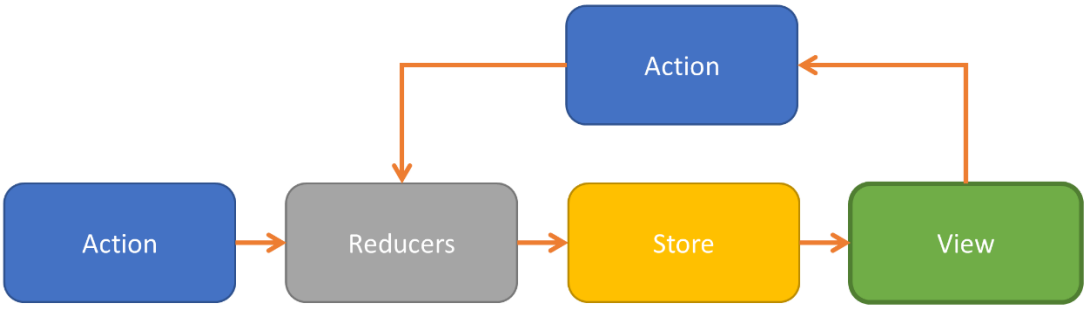
componentWillReceiveProps (nextProps)	Metodia kutsutaan pääasiassa, kun komponentille välitettään uusi `props`-objekti. React voi kutsua metodia, vaikkei `props`-objekti olisikaan muuttunut. Metodissa voidaan verrata uutta `props`-objektia ja päivittää sen pohjalta komponentin tilaa.
shouldComponentUpdate (nextProps, nextState)	Apumetodi, jolla voi tarvittaessa estää komponentin uudelleen renderöimisen. Metodi suoritetaan aina, kun komponentin saama `props`-objekti tai tila päivittyvät.
componentWillUpdate (nextProps, nextState)	Metodi suoritetaan juuri ennen komponentin renderöintiä, kun komponentin saama `props`-objekti tai tila päivittyvät. Metodi tarjoaa mahdollisuuden valmisteluiden tekemiseen ennen komponentin näkymistä web-käyttöliittymässä.
componentDidUpdate (previousProps, previousState)	Metodia kutsutaan heti, kun komponentti on päivitetty ja renderöity. Metodissa voidaan käsitellä vielä komponentin edellistä `props`-objektia tai tilaa.
componentWillUnmount()	Viimeinen elinkaarimetodi, jota kutsutaan juuri ennen komponentin poistamista DOM:ista. Tarvittavia siivoustoimenpiteitä voidaan tehdä tässä metodissa.

6.2 Redux

Sovelluksen tilan hallinta monimutkaistuu nopeasti SPA-sovelluksissa, joissa sovelluksen tila koostuu palvelimelta saaduista pyynnöistä, käyttäjän toimista sekä välimuistiin tallennetuista tiedoista. Kun tietovirta ei ole ennustettavaa, ei sitä voida useastikaan hallita. Tähän ongelmaan Redux tarjoaa ratkaisua tekemällä monimutkaisia asioita helpoksi kehittäjille.

Redux on Flux-arkkitehtuurimallista jalostettu toteutus web-sovelluksien tilan hallintaan. Reduxin on luonut Dan Abramov, jonka tavoitteena oli kehittää mahdollisimman pieni ja resursseja säästävä toteutus web-sovelluksen tilan hallintaan. Toteutuksen tavoitteena oli olla täysin ennustettava, joka laajentaisi sen sovellettavuutta sekä parantaisi kehitysympäristöä. Tämä tehostaa uusien ominaisuuksien kehittämistä ja sovelluksen

virheiden löytämistä, toistamista sekä korjaamista. Reduxin vahvuutena on sen toteutuksen yksinkertaisuus, mikä mahdollistaa sen nopean omaksumisen. [51] Kuvassa 17 on esitetty Reduxin arkkitehtuurimalli, joka muistuttaa nopeasti katsottuna aiemmin kuvassa 5 esiteltyä Flux-arkkitehtuurimallia.



Kuva 17. *Redux-arkkitehtuurimalli.*

Kuten kuvistakin voidaan nähdä, suurimpana erona arkkitehtuurimallien välillä on action-komponenttien käsittely. Flux-arkkitehtuurimallissa se on yksittäisen dispatcher-komponentin vastuulla, kun taas Reduxissa dispatcher-komponentti on korvattu useilla reducer-komponenteilla. Reduxin komponentit ja niiden erot Flux-arkkitehtuurimallin vastineisiin kuvataan tarkemmin luvussa 6.2.2.

6.2.1 Reduxin periaatteet

Reduxin ideologia koostuu kolmesta periaatteesta, jotka antavat rajat arkkitehtuurin toiminnoille:

- Sovelluksen tilat kootaan yhteen.
- Komponentit voivat vain lukea sovelluksen tilatietoa.
- Muutokset sovelluksen koottuun tilaan tehdään puhtailla funktioilla.

Kun sovelluksen tilat kootaan yhteen, lopputuloksena saadaan puumainen tietorakenne kuvaamaan koko sovelluksen tilaa. Yhden puumaisen tietorakenteen hallinta on helppoa, kun kaikki tarvittavat tiedot löytyvät aina samasta paikasta. [52]

Reduxin toinen periaate on, että sovelluksen tilaa voidaan muuttaa vain action-komponenttien välityksellä. Kuten Flux-arkkitehtuurimallissa, action-komponentit kuvaavat tapahtumia, joiden pohjalta ne viestivät halusta päivittää sovelluksen tilaa. Koska kaikki muutokset tilaan tehdään keskitetysti ja sääntöjen mukaisesti, ei hienoja tarkistuksia action-komponenttien vertailuun tai valintaan tarvita. [52]

Muutoksista sovelluksen tilaan huolehtivat reducer-komponentit, jotka määrittelevät kuinka sovelluksen tilaa tulee muuttaa saapuvien action-komponenttien seurauksena. Niiden on oltava puhtaita funktioita, joille annetaan parametreina edellinen tilatieto sekä

vastaanotettu action-komponentti. Lopputuloksena ne palauttavat uuden luodun objektin, joka kuvaa sovelluksen uutta tilaa. [52]

6.2.2 Redux-komponentit

Niin kuin Flux-arkkitehtuurimallissa, myös Reduxissa on neljä keskeistä komponenttia:

- action-komponentti
- reducer-komponentti
- store-komponentti
- view-komponentti.

Koska action- sekä view-komponentit eivät sisällä suurempia eroja arkkitehtuurien välillä, ei niitä tulla tässä kappaleessa käsittelemään. Sen sijaan keskitytään reducer- ja store-komponenttiin.

Reducer-komponentin tehtävänä on käsitellä näkymältä välitettyjä action-komponentteja ja suorittaa sovitut muutokset sovelluksen tilaan. Reducer-komponentit saavat parametreinaan edellisen hallitsemansa tilan ja vastaanotetun action-komponentin. Kuten edellä on jo mainittu, reducer-komponenttien on oltava puhtaita funktioita. Ne eivät saa muuttaa parametrejaan, suorittaa esimerkiksi API-kutsuja tai hyödyntää muita epäpuhtaita funktioita. Sovelluksen puumaisen tilan määrittävät siten reducer-komponentit, jotka voidaan aina sitoa yhden solmun alle `CombineReducers()`-metodilla. Samaa logiikkaa käytetään myös puun juuren luomiseksi. [53]

Ohjelmassa 6 on esitetty yksi reducer-komponentti, jonka vastuulla on reagoida käyttäjän valitsemaan tuotteeseen. `CombineReducers()`-metodi määrittää puumaiselle rakenteelle solmun, jonka nimeksi tässä tapauksessa tulee `selectedProduct`.

```
function selectProductReducer (state, action) {

  switch (action.type) {
    case REQUEST_PRODUCT:
      return action.product

    case REQUEST_PRODUCT_CATALOG:
      return {}

  default:
    return state
}

const selectedProduct = combineReducers({ selectProductReducer })
export default selectedProduct
```

Ohjelma 6. *Reducer-komponentti, joka hallitsee valitun tuotteen tilaa.*

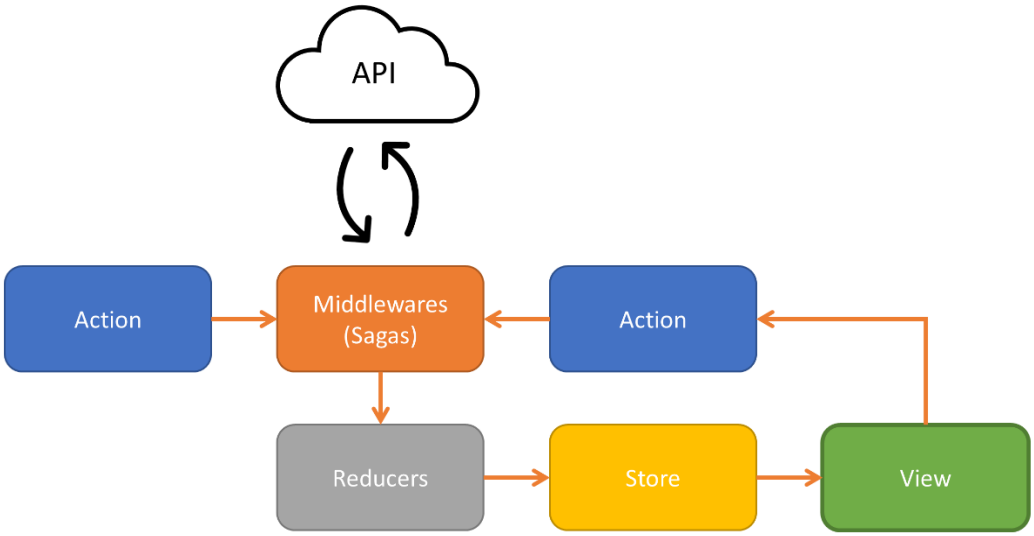
Ohjelmassa 6 käyttäjä voi valita tuotteen, jolloin tilaan tallennetaan valitun tuotteen tiedot, jotka otetaan parametrina saadusta action-komponentista. Jos käyttäjä esimerkiksi palaa takaisin luetteloituun valikkoon tarkastelemaan tuotteita, tyhjennetään hallittu tila vastaanotetun `REQUEST_PRODUCT_CATALOG`-action-komponentin seurauksena.

Toinen suuri ero Flux-arkkitehtuurimalliin verrattuna on, että Reduxissa on vain yksi store-komponentti usean sijasta. Se säilöö sovelluksen reducer-komponenteista koostettua tilaa. Store-komponentti rakennetaan ylimmän tason reducer-komponentista, joka tapahtuu `createStore()`-metodilla . Lisäksi store-komponentti tarjoaa rajapinnan tilan lukemiseen `getState()`-metodilla, action-komponenttien välittämisen `dispatch()`-metodilla ja kuuntelijoiden rekisteröimisen `subscribe()`-metodilla. [53]

6.3 Redux-saga

Garcia-Molena & Salem [54] esittelivät saagat ensimmäistä kertaa julkaisussaan jo vuonna 1987. Saagat ovat nimitys pitkille tapahtumille (l*ong-lived transaction),* jotka voidaan esittää ketjuna tai lomittaa toisten tapahtumien kanssa

Redux-saga puolestaan on JavaScript-kirjasto, joka mahdollistaa asynkronisten toimintojen hallitsemisen web-sovelluksessa. Redux-saga on Reduxin välikerros (*middleware*), jolloin sillä on pääsy Reduxin store-komponentin tilatietoon ja se voi käsitellä action-komponentteja. [55] Saagojen luonteen mukaisesti kirjasto mahdollistaa kutsujen ketjutuksen ja lomituksen. Kuvassa 18 on kuvattu tietovirta Reduxin ja välikerrosten välillä.



Kuva 18. *Redux-arkkitehtuurimalli middleware-välikerroksella, joka huolehtii asynkronisten toimintojen hallinnasta.*

Kun Redux-arkkitehtuurissa on käytössä välikerroksia, ei action-komponentteja välitetä suoraan reducer-komponenteille, vaan ne kulkevat myös välikerroksen läpi. Välikerroksella on helppoa suorittaa asynkronisia toimenpiteitä ja lähettää kutsuja esimerkiksi REST-rajapintaan.

6.3.1 Generaattorifunktiot

Redux-saga -kirjaston saagojen helppokäyttöisyys perustuu generaattorifunktioihin (*generator function*), joita käytetään tapahtumien ketjuttamiseen ja lomittamiseen saagoilla. Generaattorifunktiot ovat funktioita, joista voidaan poistua tai vastaavasti palata kesken suorituksen. Syntaksiltaan ne eroavat tavallisista funktioista vain *-merkillä, mutta niitä hallitaan poikkeavasti `yield`-lauseilla (*expression*). Generaattorifunktioista on mahdollista palauttaa arvoja `yield`-lauseella tai vastaavasti delegoida toista generaattori funktiota `yield*`-lauseella. [56] Ohjelmassa 7 on esitetty yksinkertainen `helloWorld()`-generaattorifunktio.

```
function* helloWorld() {

   yield console.log('Hello world!')
   yield console.log('Hello world again!')

};
```

Ohjelma 7. *Esimerkki generaattorifunktiosta.*

Kun luodaan ohjelman 7 mukainen `helloWorld()`-generaattorifunktio, generoi se kutsuttaessa kaksi konsoliin tulostavaa tervehdystä `yield`-lausekkeilla.

6.3.2 Watcher- ja worker-saagat

Redux-saga -sovelluksessa esiintyy kahta erityyppistä saagaa, jotka yhdessä huolehtivat asynkronisten toimintojen hallinnasta. Watcher-saagojen tehtävänä on tarkkailla vastaanotettuja action-komponentteja ja reagoida niihin. Kun watcher-saaga havaitsee sille määritetyn action-komponentin, alustaa se uuden worker-saagan tehtävän suorittamiseksi. Worker-saagan vastuulle jää varsinaisen logiikan toteutus, joka voi koostua API-kutsuista, uusien action-komponenttien luomisesta tai poikkeusten käsittelystä. [55]

Ohjelmassa 8 on kuvattu watcher- ja worker-saaga, jotka huolehtivat halutun kategorian tuotekuvaston käsittelystä.

```
function* fetchCatalog(payload) {

   const response = yield call(Api.fetchProducts, payload.category)
   yield put(FETCHED_CATALOG(response))
};

function* watchRequestProductCatalog() {

   yield takeEvery('REQUEST_PRODUCT_CATALOG', fetchCatalog)
};
```

Ohjelma 8. *Watcher- ja worker-saaga -generaattorifunktiot.*

`WatchRequestProductCatalog`-saagan `takeEvery()`-metodi kertoo saagalle, että sen täytyy reagoida jokaiseen tulevaan `REQUEST_PRODUCT_CATALOG`-action-komponenttiin. Toiminnallisuuden suorittava `fetchCatalog`-saaga suorittaa `call()`-metodilla kutsun REST-rajapintaan huomioiden kysytyn tuotekategorian. Pyynnön jälkeen se välittää saadun vastauksen Reduxin `FETCHED_CATALOG`-action-komponentille, joka toimitetaan edelleen reducer-komponenteille.

7. OFFLINE-TOIMINNALLISUUDEN TOTEUTUS TYÖKONEEN PÄÄTELAITTEELLE

Tutkimuksen mukaisesti lopuksi tavoitteena on soveltaa aiemmin etsittyjä teoreettisia ratkaisuja käytännön ongelmien ratkaisemiseksi. Asiakasprojektin tapauksessa toteutettavan ratkaisun tulee vastata sille asetettuihin offline-vaatimuksiin ja olla yhteensopiva järjestelmän jo olemassa olevaan toteutukseen. Toteutettu ratkaisu testataan ja validoidaan sidosryhmien toimesta. Jos ratkaisu ei vastaa sille asetettuihin tavoitteisiin tai siinä huomataan joitain muita puutteita, kehitysprosessissa aloitetaan uusi iteraatiokierros.

7.1 Valitut suunnitteluperiaatteet ja ratkaisut

Suunniteltavan offline-toiminnallisuuden tavoitteena on vastata web-käyttöliittymän offline-vaatimuksiin, jotka on esitetty luvussa 5.6. Lähtökohtaisesti offline-toiminnallisuuden suunnittelun kannalta on ongelmallista, että projektissa käytettävät teknologiat ja pohjalla olevat ratkaisut on lukittu. Tällöin ei voida noudattaa hyväksi todettua offline first -suunnitteluperiaatetta. Offline-toiminnallisuus on kuitenkin mahdollista suunnitella myös täysin SPA-sovelluksen web-käyttöliittymän vastuulle, mitä tukevat sovelluksen tilaa ylläpitävä Redux-arkkitehtuuri sekä selaimen tarjoamat resurssit.

Koska sovellus käyttää perinteistä SQL-tietokantaa ja palvelinpuolen toteutus on lukittu, ei luvussa 4.3.2 esitettyä päätelaitteiden ja palvelinpuolen synkronointia ole mahdollista toteuttaa. Karkean tason toteutustavoista ainoa järkevä vaihtoehto on välimuistin käyttö, sillä kyseessä on tuotannossa oleva dynaaminen järjestelmä, joka hyödyntää paljon palvelinpuolella toteutettua logiikkaa. Sovelluksen välimuistin käyttöä voidaan hallita service workerien avulla. Niiden avulla voidaan web-käyttöliittymän staattiset resurssit säilyttää selaimen välimuistissa ensimmäisen latauksen jälkeen. Kun staattiset resurssit haetaan selaimen välimuistista, avautuu web-käyttöliittymä uudelleen latauksen yhteydessä oikein myös offline-tilassa. Välimuistin käyttöä tukee myös käytössä oleva Redux-arkkitehtuuri, joka mahdollistaa store-komponentin sisältävän sovelluksen tilan synkronoinnin selaimen välimuistin kanssa. Tällöin sovelluksen tila ei nollaudu, vaikka selain suljettaisiin. Reduxin ja välimuistin avulla voidaan myös puskuroida palvelinpuolelle toimitettavia offline-tilan API-kutsuja.

Koska käyttötapaukset, saapuvan kuljetuksen purku ja lähtevän kuljetuksen lastaus, omaavat poikkeavat offline-vaatimukset, tulee ne suunnitella myös vastaavasti eri lähtökohdista. Suurin ero on työtehtävien lähtötilanne. Saapuvaa kuljetusta purettaessa tiedetään kuljetuksen kappaleiden tiedot. Varastosta lastatessa on puolestaan aina varmis-

tuttava palvelinpuolelta, että juuri kyseinen RFID-lukijalla tunnistettu kappale löytyy varastosta. Esitellyt käyttötapaukset on esitetty Redux-arkkitehtuurissa alustavasti neljällä action-komponentilla:

- `PICK_UP_UNITS`
- `PICK_UP_UNITS_FROM_WAREHOUSE`
- `UNDO_PICKED_UP_UNITS`
- `RELEASE_UNITS`,

joita Redux-saga -välikerros kuuntelee. Koska palvelinpuolelta vaaditaan aina vastaus ennen lastaamista, täytyy web-käyttöliittymä varastolastauksen osalta suunnitella toimivan pessimistisesti. Vastakohtaisesti saapuvista kuljetuksista voidaan tietoa ylläpitää Reduxin store-komponentissa, jolloin web-käyttöliittymä voidaan suunnitella optimistiseksi purkutehtävissä.

7.2 Web-sovelluksen staattisten resurssien hallinta offline-tilassa

Asiakasprojekti hyödyntää entuudestaan Webpackia [57] web-sovelluksen paketoimiseen. Olisi täysin mahdollista kirjoittaa oma W3C:n standardoiman API:n mukainen JavaScript-ohjelma, joka hoitaisi resurssien välimuistituksen selaimessa. Parempi vaihtoehto on hyödyntää Webpackille jo olemassa olevaa ja jatkuvasti kehitettävää offline plugin -liitännäistä [58], joka tarjoaa Webpackia käyttävälle web-käyttöliittymälle service workerin. Koska Reduxin store-komponentin avulla voidaan offline-tilassa hakea sovelluksen edellinen tila välimuistista, ei kyseisellä service workerilla ole tarkoitus huolehtia kuin staattisista resursseista välimuistissa.

Offline plugin -liitännäisen käyttöönotto sovelluksessa sisältää kaksi vaihetta:

- Liitännäisen käynnistäminen, kun web-sovellus käynnistetään.
- Välimuistin käytön määrittäminen.

Ensimmäisen vaiheen tehtävä on rekisteröidä selaimeen service worker, kun web-sovellus käynnistetään. Ohjelmassa 9 on esitetty liitännäisen käynnistäminen, joka on toteutettu projektin web-käyttöliittymän juurena toimivassa `index.jsx`-tiedostossa.

```
import * as OfflinePluginRuntime from 'offline-plugin/runtime';

OfflinePluginRuntime.install();
```

Ohjelma 9. *Offline plugin -liitännäisen käynnistäminen web-sovelluksessa.*

Liitännäisen avulla hallittavan välimuistin käyttö on määritetty puolestaan projektin `webpack.config.js`-tiedostossa, jolla hallitaan kaikkia Webpackin käyttöön liittyviä asetuksia. Offline-toiminnallisuuden vaatima asetusten määritys on esitetty ohjelmassa

10, jonka avulla voidaan valita välimuistiin tallennettavat resurssit ja noudatettavat säännöt.

```
plugins: [
  // Muut Webpackin liitännäiset
  new OfflinePlugin({
    publicPath: '/',
    caches: {
      main: ['*/app.css', '*bundle.js’],
      additional: [‘:rest:’]
    },
    ServiceWorker: {
       navigateFallbackURL: '/'
     }
  })
]
```

Ohjelma 10. *Webpack Offline plugin -liitännäisen asetusten määritys.*

Liitännäisen `main`-osassa ladataan tärkeimmät resurssit sovelluksen välimuistiin jo käynnistyessä, kun taas `additional`-osan resurssit ladataan vasta käynnistyksen jälkeen taustalla. Service workerille määritettiin myös `navigateFallbackURL`-polku, johon käyttäjä ohjataan, jos haluttua resurssia ei löydy välimuistista.

7.3 Web-käyttöliittymän toiminnallisuus offline-tilassa

Ideaaliratkaisu Redux-pohjaisen web-käyttöliittymän offline-toiminnallisuuteen olisi hyödyntää Eväkallion [40] kehittämää Redux Offline -kirjastoa. Se antaa mahdollisuuden suunnitella web-käyttöliittymän arkkitehtuuri offline-toiminnallisuus etusijalla. Toteutus perustuu Reduxiin lisättyyn välikerrokseen, joka huolehtii sovelluksen tilan välimuistituksesta sekä määriteltyjen action-komponenttien puskuroinnista offline-tilassa. Jotta verkkoyhteyttä vaativat action-komponentit puskuroitaisiin offline-välikerroksen toimesta, on niiden metatiedoissa määritettävä offline-kentät.

Ohjelmassa 11 on esitetty Eväkallion offline-action -komponentin luonti action creator -apumetodilla. Offline-action -komponentin metatiedot koostuvat verkkoyhteyttä vaativasta pyynnöstä (*effect*), onnistuessa lähetetystä action-komponentista (*commit*) sekä pyynnön täysin epäonnistuessa lähetettävästä action-komponentista (*rollback*).

```
const createOfflineProductAction = payload => ({
  type: 'FETCH_PRODUCTS’,
  payload: { payload },
  meta: {
    offline: {
      // Pyyntö REST-rajapintaan
      effect: { url: '/api/fetchProducts, method: ‘GET’ },
      commit: { type: ' FETCH_PRODUCTS_COMMIT' },
      rollback: { type: ' FETCH_PRODUCTS_ROLLBACK'}
    }
  }
});
```

Ohjelma 11. *Offline-action -komponentin luonti.*

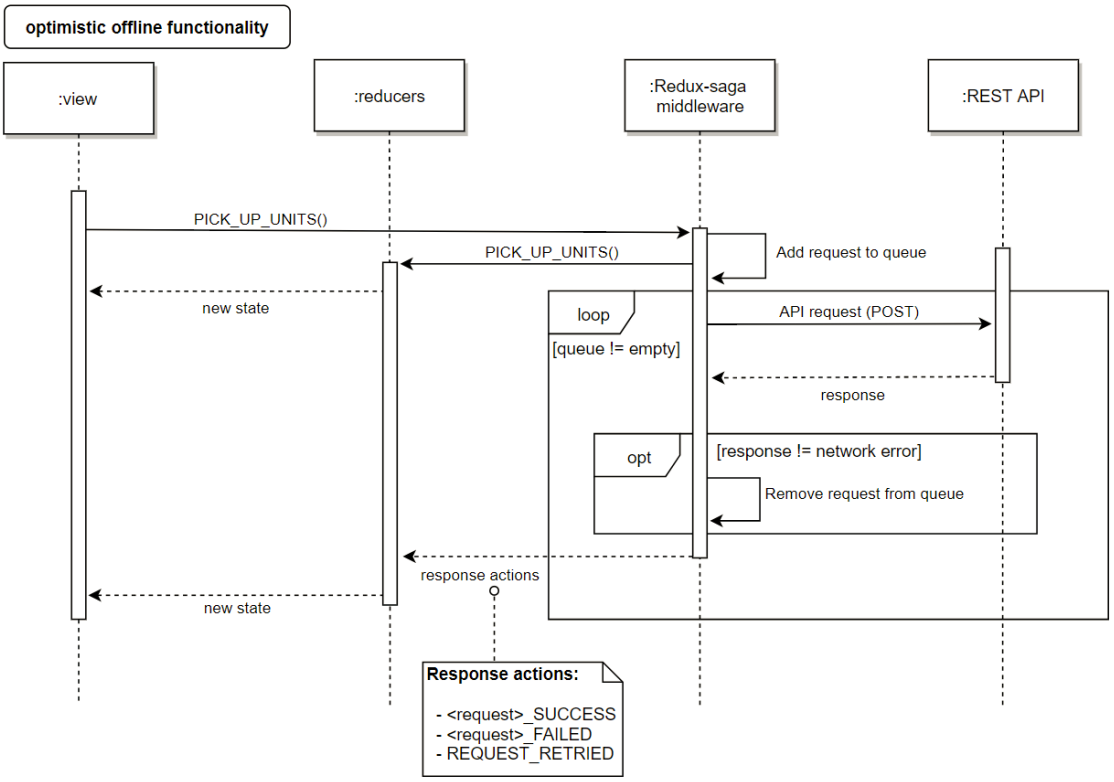
Koska asiakasprojektin web-käyttöliittymän arkkitehtuuri on pitkälle jo valmiiksi toteutettu, ei uuden Reduxin välikerroksen lisääminen ollut kuitenkaan mahdollista projektin nykyisessä vaiheessa. Web-käyttöliittymän offline-toiminnallisuus päätettiin toteuttaa olemassa olevan Redux-saga -välikerroksen tukemilla generaattorifunktioilla. Yleisesti toteutetut apugeneraattorifunktiot on esitelty liitteessä A. Se sisältää `createRetryingGenerator()`-funktion, joka luo uuden generaattorifunktion uudelle tapahtumalle ja lisää sen generaattorifunktioista koostuvan `requestQueue`-objektin jonoon. `RequestQueue`-objektin tehtävänä on huolehtia jonotettujen pyyntöjen suorituksesta, niiden mahdollisesta uudelleen yrityksestä ja hylkäyksestä, jos ne epäonnistuvat lopullisesti.

Redux-offline -kirjaston mukaisesti jokaiselle palvelinpuolelle vaikuttavalle action-komponentille luotiin toiset action-komponentit, jotka viestivät REST-rajapintaan kohdistuvien pyyntöjen onnistumisesta tai epäonnistumisesta. Nämä erilliset `SUCCESS`- ja `FAILED`-action-komponentit mahdollistavat ilmoitukset käyttäjälle pyyntöjen lopputuloksesta. Web-käyttöliittymän päivittäminen optimistisesti tai pessimistisesti voidaan toteuttaa reducer-komponenttien avulla tapauskohtaisesti. Jos päivityksen on määrä olla optimistinen, käsitellään reducer-komponenteissa alkuperäisiä action-komponentteja ja pyynnön virhetilanteen action-komponentteja. Pessimistisessä päivityksessä reducer-komponenteissa reagoidaan puolestaan vain palvelinpuolen vastauksen jälkeisiin action-komponentteihin.

7.3.1 Kappaleiden purkaminen kuljetuksesta

Lähtökohtana kappaleiden purkuun saapuvasta kuljetuksesta oli valitun kuljetuksen tietojen lataaminen Reduxin store-komponenttiin, kun tehtävää aloitetaan suorittamaan. Tällöin tiedetään entuudestaan kuljetuksessa sijaitsevien kappaleiden RFID-tunnisteet, joiden pohjalta kappaleita voidaan lukea työkoneen kyytiin. Työtehtävän vaatimuksena oli pystyä purkamaan koko kuljetus ilman verkkoyhteyttä, mikä tuli huomioida Redux-saga- välikerroksen generaattorifunktioissa ja Reduxin reducer-komponenteissa. Opti-

mistinen offline-toiminnallisuus on esitetty kuvan 19 sekvenssikaaviossa, joka kuvaa kappaleen nostamisen työkoneen kyytiin kuljetuksesta.



Kuva 19. *Optimistisen offline-toiminnallisuuden sekvenssikaavio.*

Kuvan 19 sekvenssikaviota vastaavat worker- ja watcher-saagat on esitetty puolestaan ohjelmassa 12.

```
const pickUp = createRetryingGenerator(
  Api.pickupUnit,
  PICK_UP_UNITS_SUCCESS,
  PICK_UP_UNITS_FAILED,
  payload => parseIdAndTimestamp(payload),
  REQUEST_RETRIED
);

export const watchPickUpUnit = takeEvery(PICK_UP_UNIT().type,
pickUp);
```

Ohjelma 12. *Yksittäisen toiminnallisuuden worker- ja watcher-saagat.*

Ohjelma 12 hyödyntää liitteen A `createRetryingGenerator()`-funktiota, joka huolehtii `Api.pickupUnit`-parametrin mukaisen pyynnön toimituksesta REST-rajapintaan. Jos verkkoyhteyttä ei ole, lähetetään `REQUEST_RETRIED`-action-komponentti. Kun lopuksi pyyntö onnistuu tai epäonnistuu, lähetetään joko `PICK_UP_UNITS_SUCCESS`- tai `PICK_UP_UNITS_FAILED`-action-komponentti. Alkuperäisen action-komponentin `payload`-objekti puolestaan käsitellään `par-`

`seIdAndTimestamp()`-funktiolla, joka karsii objektista vain id:n ja aikaleiman. `WatchPickUpUnit`-watcher-saaga huolehtii, että jokainen saapuva `PICK_UP_UNITS` -action-komponentti käsitellään `pickUp`-worker-saagalla. Yhtäläisesti myös `RELEASE_UNITS`- ja `UNDO_PICKED_UP_UNITS`-action-komponenttien käsittelyyn luotiin ohjelmaa 12 vastaavat toteutukset.

Varsinaisen web-käyttöliittymän optimistisen päivityksen ja ilmoitusten esittämisen käyttäjälle hoitavat Reduxin reducer-komponentit. Ohjelmassa 13 on esitetty muokatut reducer-komponentit, `carriedUnitsReducer` ja `selectedTransportReducer`, jotka huolehtivat web-käyttöliittymän optimistisesta päivittämisestä.

```
export const carriedUnitsReducer = handleActions({
  //*
     Muiden action-komponenttien käsittelyä
  *//
  [PICK_UP_UNITS]: (state, action) =>
     state.concat(action.payload),
  [UNDO_PICKED_UP_UNITS]: (state, action) =>
     removeFromState(state, action),
  [RELEASE_UNITS]: (state, action) =>
     removeFromState(state, action)
  }, Immutable.List());

export const selectedTransportReducer = handleActions({
  //*
     Muiden action-komponenttien käsittelyä
  *//
  [RELEASE_UNITS]: (state, action) => {
      return removeFromTransport(state, Action);
  }, Immutable.Map());
```

Ohjelma 13. *Optimistisesti toimivat reducer-komponentit käsiteltävän kuljetuksen ja työkoneen kyydissä olevien kappaleiden tilan hallintaan.*

Reducer-komponentit reagoivat `handleActions()`-funktiossa määritettyihin action-komponentteihin. `CarriedUnitsReducer`-komponentti huolehtii työkoneen kyydissä olevista kappaleista. Kun kappale poimitaan, `carriedUnitsReducer`-komponentti lisää sen hallitsemaansa tilaan optimistisesti `concat()`-metodilla. Kun kappale puolestaan kuitataan kohteeseen, poistetaan se molempien reducer-komponenttien tilasta määritettyjen `removeFromState()`- ja `removeFromTransport()`-funktioiden avulla. Reducer-komponentit on toteutettu muuttumattomilla `Immutable`-kirjaston objekteilla, jotka alustetaan oletusarvoisesti tyhjiksi, kun web-sovellus käynnistetään päätelaitteen selaimessa.

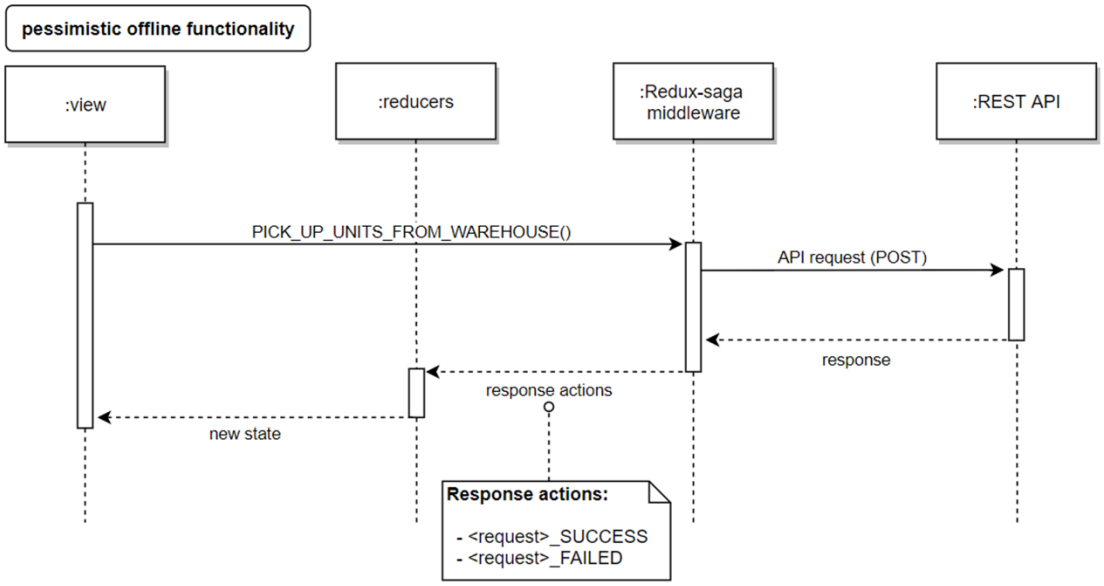
Kun joudutaan offline-tilaan, on siitä ilmoitettava käyttäjälle. Web-käyttöliittymässä ilmoituksista vastaa oma reducer-komponentti, joka luo erilaisia ilmoituksia saapuvien action-komponenttien pohjalta. Saagojen avulla generoitujen action-komponenttien

avulla käyttäjää informoidaan esimerkiksi kappaleiden kuittauksen yhteydessä seuraavasti:

- Ilmoita offline-tilasta, jos pyyntöä yritetään uudestaan (`REQUEST_RETRIED`).
- Ilmoita, kun pyyntö suoritetaan onnistuneesti palvelinpuolelle (`RELEASE_UNITS_SUCCESS`).
- Ilmoita, jos pyyntö palvelinpuolelle epäonnistuu (`RELEASE_UNITS_FAILED`).

7.3.2 Kappaleiden lastaaminen varastosta

Kun varastosta kerätään kappaleita, on varmistuttava niiden olemassa olosta. Tällöin web-käyttöliittymän täytyy päivitys suorittaa keräyksen osalta pessimistisesti. Pessimistinen offline-toiminnallisuus on kuvattu kuvan 20 sekvenssikaaviossa. Sekvenssikaavio esittää kappaleen keräämisen varastosta työkoneen kyytiin.



Kuva 20. *Pessimistisen offline-toiminnallisuuden sekvenssikaavio.*

Ohjelmassa 14 on esitetty varastolastauksessa käytetyt worker- ja watcher-saagat. Jokaisesta `PICK_UP_UNITS_FROM_WAREHOUSE`-action-komponentista `pickupFromWarehouse`-worker-saaga suorittaa ensin pyynnön palvelinpuolelle, joka kertoo löytyikö kappaleita varastosta. Jos pyyntö epäonnistuu esimerkiksi verkko-ongelmien seurauksena, lähetetään `PICK_UP_UNITS_FROM_WAREHOUSE_FAILED`-action-komponentti.

```
function* pickupFromWarehouse({ payload }) {
  try {
    const response = yield call(Api.pickUpFromWarehouse,
      payload);

    // Löytyykö kappaleet varastosta
    if (response.data && response.data.length !== 0) {
        yield put(PICK_UP_UNITS_FROM_WAREHOUSE_SUCCESS(
                  response.data));
    } else {
        yield put(UNITS_NOT_FOUND_FROM_WAREHOUSE());
    }
  } catch (err) {
    yield put(PICK_UP_UNITS_FROM_WAREHOUSE_FAILED(err));
  }
};

export const pickUpPulpFromWarehouse = takeEvery(
  PICK_UP_UNITS_FROM_WAREHOUSE().type, pickupFromWarehouse)
```

Ohjelma 14. *Varastosta lastauksen worker- ja watcher-saagat.*

Kappaleiden kuittaamiseen kohteeseen voidaan hyödyntää optimistista mallia jo toteutettujen ohjelmien 12 ja 13 mukaisesti. Varastokeräyksen pessimistisen päivityksen huomioimiseksi, `carriedUnitsReducer`-komponentissa reagoidaan vain onnistuneeseen pyyntöön palvelinpuolelle ohjelman 15 mukaisesti.

```
[PICK_UP_UNITS_FROM_WAREHOUSE_SUCCESS]: (state, action) =>
  state.concat(action.payload)
```

Ohjelma 15. *Varastolastauksen pessimistinen päivitys reducer-komponentissa.*

Varastosta keräys voi epäonnistua, kun ollaan joko offline-tilassa tai varastosta ei löydy luettuja kappaleita. Myös tällöin käyttäjälle ilmoitetaan virhetilanteesta ilmoituksista vastaavan reducer-komponentin avulla.

8. TULOKSET JA JOHTOPÄÄTÖKSET

Tämän työn tavoitteena oli perehtyä web-sovellusten offline-toiminnallisuuden suunnitteluun ja tuoda esiin haasteita, joita offline-kykyisyys aiheuttaa koko toteutettavalle järjestelmälle. Löydettyjen suunnittelu- ja ratkaisumallien pohjalta suunniteltiin ja toteutettiin offline-toiminnallisuus asiakasprojektin logistiikan tietojärjestelmään. Lopputuloksena saatiin toteutettua offline-toiminnallisuus Redux-arkkitehtuurimalliin pohjautuvaan web-sovellukseen, joka mahdollistaa ongelman omistajien työtehtävien suorittamisen vaihtelevallakin verkkoyhteydellä.

Nimensä mukaisesti offline first -suunnitteluperiaate on suositeltava lähtökohta offline-kykyisen web-sovelluksen suunnitteluun. Se auttaa ottamaan offline-vaatimukset huomioon ratkaisuissa, kun web-sovellus suunnitellaan ensisijaisesti offline-tilassa käytettäväksi. Offline-toiminnallisuuden suunnittelun tulisi olla järjestelmän kehitysprosessissa mukana alusta asti. Tällöin hyödynnettäviä ratkaisumalleja on tarjolla useampia ja vastuut offline-toiminnallisuudesta voidaan jakaa halutuille järjestelmän komponenteille. Offline-toiminnallisuuden toteutus liittyy joka tapauksessa web-käyttöliittymän kautta välittyvään käyttäjäkokemukseen. Käyttäjän opastus, informointi ja muut vuorovaikutustilanteet on otettava huomioon, jotta sovelluksen käyttö on luontevaa ja mieluista. Jos käyttäjä ei tiedä sovelluksen toimintaperiaatetta, on hänelle pyynnön kestäessä luonteenomaista ladata sivu uudelleen. Tällöin syntyy helposti uusia haasteita toteutettavalle offline-toiminnallisuudelle.

Suurin offline-toiminnallisuuden aiheuttama haaste liittyy tiedon eheyteen. Koska web-käyttöliittymän ja palvelinpuolen tilatieto on tavalla tai toisella synkronoitava, ovat konfliktitilanteiden syntyminen ja käyttäjän tekemän transaktion tietojen menettäminen mahdollisia. Jos offline first -suunnitteluperiaatetta on noudatettu, on ongelman ratkaisuun parhaimmillaan useampi vaihtoehto. Löydetyistä vaihtoehdoista ongelman ratkaisuun parhaiten sopivat selaimen välimuistin käyttö, service workerit tai Teixeiran [28] esittelemän web-käyttöliittymän ja palvelinpuolen erillisten tietokantojen synkronointi. Tärkeää on lisäksi muistaa, että offline-toiminnallisuuden ratkaisu on aina kompromissi. Voidaan joko luottaa enemmän verkkoyhteyteen ja hakea useasti uutta sisältöä tai käyttää selaimen resursseja sisällön ennakoivaan tallennukseen raskailla API-pyynnöillä.

Lähtökohta logistiikan web-sovelluksen offline-toiminnallisuuden suunnitteluun ja toteutukseen ei ollut ideaalinen. Järjestelmän arkkitehtuuri ja tekniikat oli jo valittu, jolloin vaihtoehdot offline-toiminnallisuuden toteutukseen olivat rajatut. Koska projekti nojasi jo SQL-tietokantaan, olisi NoSQL-välikerroksen lisääminen dynaamiseen järjestelmään ollut monimutkaista. Sen sijaan päätettiin hyödyntää service workereita ja se-

laimen välimuistia. Niiden avulla toteutettiin offline-kykyinen web-sovellus, joka täyttää sille luvussa 5 asetetut vaatimukset. Web-sovelluksen staattisten resurssien välimuistitukseen käytettiin service workeria Webpackille tarjotun liitännäisen avulla. Sovelluksen tilatiedon hallinnassa hyödynnettiin Reduxin tarjoamia työkaluja, Redux-saga- kirjastoa ja generaattorifunktioita.

Redux-arkkitehtuuri osoittautui yksinkertaiseksi sekä tehokkaaksi tavaksi hallita ja välimuistittaa sovelluksen tilatietoa keskitetyn store-komponentin avulla. Dynaamisen sovelluksen tilatiedon synkronoinnin hallinta oli helppoa, mikä Eväkallion [40] mukaan pelkällä service workerin API:lla voi osoittautua mahdottomaksi. Käyttäjäkokemuksesta huolehtiminen web-käyttöliittymässä tapauskohtaisesti optimistisesti tai pessimistisesti oli mahdollista Reduxin reducer-komponenteilla, jotka huolehtivat myös palvelinpuolelta saapuvan tilan synkronoinnista store-komponentille. Palvelinpuolen asynkronisten pyyntöjen jonotus osoittautui myös toimivaksi generaattorifunktioilla. Koska generaattorifunktioita ei kuitenkaan välimuistitettu, aiheuttaa esimerkiksi selaimen uudelleenlautaus jonon tyhjenemisen ja tietojen menettämisen. Löydetty ongelma voitaisiin välttää esimerkiksi käsitellyllä Redux Offline -kirjastolla, joka mahdollistaa myös offline-tilassa jonottavien pyyntöjen välimuistituksen. Tällöin uudelleenkäynnistäessä voitaisiin jatkaa taas jonon käsittelemättömistä pyynnöistä. Redux Offline -kirjasto ei kuitenkaan ollut suoraan yhteensopiva riippuvuuksien vuoksi asiakasprojektiin. Se olisi vaatinut suuren määrän työtä projektipohjan päivittämisessä, jota ei nähty toistaiseksi vaivan arvoiseksi.

Toteutuksessa käytetyt menetelmät vahvistivat selaimen välimuistin ja service workereiden soveltuvuutta web-sovellusten offline-toiminnallisuuden toteuttamiseksi. Havainnot ja lopputulos tukevat etenkin Kastenin [26], Feyerken [27] ja Teixeiran [28] näkemystä välimuistin käytön puolesta. Välimuistia hyödyntämällä sovelluksen suorituskykyä voidaan tehostaa ja käyttäjäkokemusta parantaa, vaikka web-sovellus ei joutuisikaan varsinaiseen offline-tilaan.

LÄHTEET

[1] J. McKay, P. Marshall, The dual imperatives of action research, Information Technology & People, Vol. 14, No. 1, 2001, pp. 46–59.

[2] M.H. Baturay, M. Birtane, Responsive Web Design: A New Type of Design for Web-based Instructional Content, 4th International Conference on New Horizons in Education, 10 December 2013, pp. 2275–2279.

[3] F. Shahzad, Modern and Responsive Mobile-enabled Web Applications, 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops, 2017, pp. 410–415.

[4] Developer Survey Results 2017, Stack Overflow, verkkosivu. Saatavissa (viitattu 8.10.2017): https://insights.stackoverflow.com/survey/2017#overview.

[5] F.N. Egger, Affective design of e-commerce user interfaces: How to maximise perceived trustworthiness, Proc. Intl. Conf. Affective Human Factors Design, pp. 317–324.

[6] G.R. Andrews, Paradigms for Process Interaction in Distributed Programs, ACM Comput.Surv., Vol. 23, No. 1, 1991, pp. 49–90.

[7] J. Tihomirovs, J. Grabis, Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics, Information Technology and Management Science, Vol. 19, No. 1, 2016, pp. 92–97.

[8] R.T. Fielding, REST: architectural styles and the design of network-based software architectures, Doctoral dissertation, University of California, 2000.

[9] R.T. Fielding, R.N. Taylor, Principled design of the modern Web architecture, ACM Transactions on Internet Technology (TOIT), Vol. 2, No. 2, 2002, pp. 115–150.

[10] C. Pautasso, O. Zimmermann, F. Leymann, Restful web services vs. "big" web services: making the right architectural decision, Proceedings of the 17th international conference on World Wide Web, ACM, pp. 805–814.

[11] M. Wasson, Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET, verkkosivu. Saatavissa (viitattu 8.10.2017): https://msdn.microsoft.com/en-us/magazine/dn463786.aspx.

[12] Why a Single Page Application, What are the Benefits ? What is a SPA ? Angular University, verkkosivu. Saatavissa (viitattu 9.10.2017): https://blog.angular-university.io/why-a-single-page-application-what-are-the-benefits-what-is-a-spa/;.

[13] P. Skólski, Single-Page Application vs. Multiple-Page Application, Neoteric, verkkosivu. Saatavissa (viitattu 9.10.2017): https://neoteric.eu/single-page-application-vs-multiple-page-application.

[14] P. Hudak, Conception, evolution, and application of functional programming languages, ACM Computing Surveys (CSUR), Vol. 21, No. 3, 1989, pp. 359–411.

[15] A. Kaipiainen, Funktionaalinen ohjelmointi web-ohjelmistokehityksessä, Diplomityö, 2017, 1–50 p. Saatavissa: https://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/24541/Kaipiainen.pdf.

[16] Stream (Java Platform SE 8), Oracle, verkkosivu. Saatavissa (viitattu 11.10.2017): https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.

[17] G.E. Krasner, S.T. Pope, A description of the model-view-controller user interface paradigm in the smalltalk-80 system, Journal of object oriented programming, Vol. 1, No. 3, 1988, pp. 26–49.

[18] A. Syromiatnikov, D. Weyns, A Journey through the Land of Model-View-Design Patterns, 2014 IEEE/IFIP Conference on Software Architecture, April, pp. 21–30.

[19] A. Leff, J. T. Rayfield, Web-application development using the Model/View/Controller design pattern, Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference, 2001, pp. 118–127.

[20] MVC architecture, Mozilla, verkkosivu. Saatavissa (viitattu 15.10.2017): https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture.

[21] ASP.NET MVC Overview, Microsoft, verkkosivu. Saatavissa (viitattu 15.10.2017): https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx.

[22] Flux documentation: In depth overview, Facebook Inc., verkkosivu. Saatavissa (viitattu 15.10.2017): http://facebook.github.io/flux/docs/in-depth-overview.html#content.

[23] Progressive Web Apps, Google Developers, verkkosivu. Saatavissa (viitattu 14.11.2017): https://developers.google.com/web/progressive-web-apps/.

[24] P. LePage, Your First Progressive Web App, Google Developers, verkkosivu. Saatavissa (viitattu 14.11.2017): https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp.

[25] Offline-first web and mobile apps: Top frameworks and components, Tech Beacon, verkkosivu. Saatavissa (viitattu 14.11.2017): https://techbeacon.com/offline-first-web-mobile-apps-top-frameworks-components.

[26] N. Kasten, Using React and Preact to Build My First Offline First Apps, Offline Camp, verkkosivu. Saatavissa (viitattu 14.11.2017): https://medium.com/offline-camp/using-react-and-preact-to-build-my-first-offline-first-apps-8df4a1e5471b.

[27] A. Feyerke, Designing Offline-First Web Apps, A List Apart, No. 386, 2013, Saatavissa (viitattu 14.11.2017): https://alistapart.com/article/offline-first.

[28] P. Teixeira, Build More Reliable Web Apps with Offline-First Principles, The New Stack, verkkosivu. Saatavissa (viitattu 14.11.2017): https://thenewstack.io/build-better-customer-experience-applications-using-offline-first-principles/.

[29] M. Kurtuldu, Offline UX Considerations, Google Developers, verkkosivu. Saatavissa (viitattu 14.11.2017): https://developers.google.com/web/fundamentals/instant-and-offline/offline-ux.

[30] D. Sauble, Offline: When Your Apps Can't Connect to the Internet, uxdesign.cc, verkkosivu. Saatavissa (viitattu 14.11.2017): https://uxdesign.cc/offline-93c2f8396124.

[31] C. Andreu, Offline Patterns, IBM, verkkosivu. Saatavissa (viitattu 14.11.2017): https://www.ibm.com/developerworks/community/blogs/worklight/entry/offline_patterns?lang=en.

[32] Web Storage, W3C, verkkosivu. Saatavissa (viitattu 22.1.2018): https://www.w3.org/TR/webstorage/.

[33] Indexed Database API 2.0, W3C, verkkosivu. Saatavissa (viitattu 22.1.2018): https://www.w3.org/TR/IndexedDB-2/.

[34] Service Workers Spesification, W3C, verkkosivu. Saatavissa (viitattu 22.11.2017): https://www.w3.org/TR/service-workers-1/.

[35] Using Service Workers, Mozilla, verkkosivu. Saatavissa (viitattu 14.11.2017): https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers.

[36] J. Archibald, The Offline Cookbook, Google Developers, verkkosivu. Saatavissa (viitattu 14.11.2017): https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook/.

[37] M. Kearney, Measure Performance with the RAIL model, Google Developers, verkkosivu. Saatavissa (viitattu 24.11.2017): https://developers.google.com/web/fundamentals/performance/rail.

[38] D. Mishunov, True Lies of Optimistic User Interfaces, Smashing Magazine, verkkosivu. Saatavissa (viitattu 24.11.2017): https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interface.

[39] D. Bricklin, Dealing with Disconnected Operation in a Mobile Business Application: Issues and Techniques for Supporting Offline Usage, verkkosivu. Saatavissa (viitattu 14.11.2017): http://bricklin.com/offline.htm.

[40] J. Eväkallio, Introducing Redux Offline: Offline-First Architecture for Progressive Web Applications and React Native, Hacker Noon, verkkosivu. Saatavissa (viitattu 14.11.2017): https://hackernoon.com/introducing-redux-offline-offline-first-architecture-for-progressive-web-applications-and-react-68c5167ecfe0.

[41] R.H. Ballou, Business logistics/supply chain management: planning, organizing, and controlling the supply chain, 5th ed. Pearson Prentice Hall, Upper Saddle River, NJ, 2004, 789 p.

[42] J.T. Mentzer, W. DeWitt, J.S. Keebler, S. Min, N.W. Nix, C.D. Smith, Z.G. Zacharia, Defining supply chain management, Journal of Business logistics, Vol. 22, No. 2, 2001, pp. 1–25.

[43] S.E. Fawcett, L.M. Ellram, J.A. Ogden, Supply chain management: from vision to implementation, Pearson new international ed. Pearson, Harlow, 2014, 600 p.

[44] D.C. Milić, B. Zorić, Trends in the use of information technology in logistics systems management, Ekonomski Vjesnik, Vol. 30, No. 1, 2017, pp. 221–236.

[45] C. Gackenheimer, Introduction to React, Apress, 2015, 129 p.

[46] D. Abramov, React blog: React Components, Elements and Instances, Facebook Inc., verkkosivu. Saatavissa (viitattu 25.10.2017): https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html.

[47] React documentation: Introducing JSX, Facebook Inc., verkkosivu. Saatavissa (viitattu 28.10.2017): https://reactjs.org/docs/introducing-jsx.html.

[48] React documentation: Components and props, Facebook Inc., verkkosivu. Saatavissa (viitattu 25.10.2017): https://reactjs.org/docs/components-and-props.html.

[49] React documentation: State and lifecycle, Facebook Inc., verkkosivu. Saatavissa (viitattu 25.10.2017): https://reactjs.org/docs/state-and-lifecycle.html.

[50] React documentation: React.Component, Facebook Inc., verkkosivu. Saatavissa (viitattu 3.11.2017): https://reactjs.org/docs/react-component.html.

[51] Redux documentation: Read me, verkkosivu. Saatavissa (viitattu 7.11.2017): https://redux.js.org.

[52] Redux documentation: Three Principles, verkkosivu. Saatavissa (viitattu 7.11.2017): https://redux.js.org/docs/introduction/ThreePrinciples.html.

[53] Redux documentation: Basics, verkkosivu. Saatavissa (viitattu 7.11.2017): https://redux.js.org/docs/basics/.

[54] H. Garcia-Molina, K. Salem, Sagas, Proceedings of the 1987 ACM SIGMOD international conference on Management of data, Vol. 16, No. 3, 1987, pp. 249–259.

[55] Redux-saga documentation: Introduction, verkkosivu. Saatavissa (viitattu 7.11.2017): https://redux-saga.js.org/docs/introduction.

[56] Function*, Mozilla, verkkosivu. Saatavissa (viitattu 7.11.2017): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*.

[57] Webpack module bundler, Webpack, verkkosivu. Saatavissa (viitattu 30.11.2017): https://webpack.js.org/.

[58] A. Stolyar, Webpack offline-plugin, Github, Inc., verkkosivu. Saatavissa (viitattu 23.1.2018): https://github.com/NekR/offline-plugin.

LIITE A: YLEISET GENERAATTORIFUNKTIOT

```
import { call, put } from 'redux-saga/effects';
import { delay } from 'redux-saga';

// Funktio pyynnön virhetyypin määrittämiseen
const isTimeout = (error) => {
  if (error.message) {
    return error.message === ('Network Error');
  }
  return false;
};

const requestQueue = {
  requests: [],
  requestInProgress: false,
  // Lisätään uusi pyyntö jonon viimeiseksi ja otetaan
  // ensimmäinen käsittelyyn
  * add(nextAction) {
    requestQueue.requests.push(nextAction);
    if (!requestQueue.requestInProgress) {
      yield requestQueue.makeNextRequest();
    }
  },
  // Erotetaan jonosta ensimmäinen pyyntö, josta lähdetään jonoa
  // rekursiivisesti purkamaan
  * makeNextRequest() {
    if (requestQueue.requests.length === 0) {
      requestQueue.requestInProgress = false;
    } else {
      requestQueue.requestInProgress = true;
      const first = requestQueue.requests.slice(0, 1)[0];
      const rest = requestQueue.requests.slice(1);
      try {
        yield requestQueue.tryToCall(first);
        // Onnistuessa poistetaan ensimmäinen pyyntö ja jatketaan
        // seuraavaan jonossa
        requestQueue.requests = rest;
        yield requestQueue.makeNextRequest();
      } catch (err) {
        // Verkkoyhteyden puuttuessa, yritetään viiveen jälkeen uudestaan
        if (isTimeout(err)) {
          yield call(delay, 7 * 1000);
        } else {
          // Hylätään muut virheelliset pyynnöt
          requestQueue.requests = rest;
        }
        // Virhetilanteessa jatketaan myös jonon tyhjentämistä
        yield requestQueue.makeNextRequest();
      }
    }},
```

```
  // Suoritetaan jonosta seuraava pyyntö
  * tryToCall({ apiEndpoint, successAction, retryAction, errorAction,
                payload }) {
    try {
      const response = yield call(apiEndpoint, payload);
      yield put(successAction(response.data));
      return;
    } catch (err) {
      // Ilmoitetaan mahdollisesti offline-tilasta käyttäjälle
      if (isTimeout(err) && retryAction) {
        yield put(retryAction());
      } else {
        // Ilmoitetaan, että pyyntö epäonnistui muusta syystä
        yield put(errorAction(err));
      }
      throw err;
    }
  }
};

// Luodaan uusi pyyntö jonoon
// Parametrit:
// apiEndpoint: pyyntö REST-rajapintaan
// successAction: action-komponentti pyynnön onnistuessa
// errorAction: action-komponentti pyynnön epäonnistuessa
// payloadTransformation: funktio payload-objektin käsittelyyn
// retryAction: action-komponentti pyynnön uudelleenyrityksessä
export const createRetryingGenerator = (apiEndpoint, successAction,
errorAction, payloadTransformation, retryAction) =>
  function* retryRequest({ payload }) {
    const request = {
      apiEndpoint,
      successAction,
      retryAction,
      errorAction,
      payload: payloadTransformation ? payloadTransformation(payload)
                : payload
    };
    yield requestQueue.add(request);
  };
```