TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JANI PIIPPO

NOSTURIHUOLLON RAPORTOINNIN MODERNISOINTI

Diplomityö

Tarkastaja:
Prof. Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty
08.08.2018

TIIVISTELMÄ

JANI PIIPPO: Nosturihuollon raportoinnin modernisointi
Tampereen teknillinen yliopisto
Diplomityö, 49 sivua
Lokakuu 2018
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Hannu-Matti Järvinen
Avainsanat: nosturitarkastus, raportointi, modernisointi, digitaalinen muutos, mobiilisovellus, selainsovellus

Digitaalinen muutos ajaa yrityksiä ja yhteisöjä hyödyntämään digitaalisia teknologioita yhä enemmän ja enemmän liike- ja muussa toiminnassaan. Digitaalisten teknologioiden myötä on mahdollista saavuttaa tuottavuuden kasvua ja taloudellista hyötyä.

Digitaalisen muutoksen eräs ilmentymä on sähköisen tietojenkäsittelyn lisääntyminen. Tästä esimerkkinä voi olla esimerkiksi paperin korvaaminen yrityksen liiketoiminnan prosesseissa. Monien yritysten liiketoimintaan liittyy raportointi eri muodoissa, myös paperisena.

Hyvin toteutettu raportointi lisää tuottavuutta, työntekijöiden tyytyväisyyttä, parantaa päätöksentekokykä ja organisaation sisäistä ja ulkoista kommunikaatiota. Siksi mahdollisimman tehokkaasti toimiva raportointi on usein hyödyksi monella liiketoiminnan alueella.

Tässä diplomityössä modernisoidaan Konecranes Oyj:n satamanosturien huoltotarkastuksiin liittyvä raportointijärjestelmä. Työssä keskitytään erityisesti itse raporttien muodostamiseen liittyvään tekniseen toiminnallisuuteen osana selainsovellusta. Raportteja muodostetaan määrätyn muotoisesta lähtödatasta muutaman välivaiheen kautta. Lopputuloksena syntyy sähköinen PDF-muotoinen raportti.

Diplomityössä on tunnistettu toimintaympäristön vaatimukset, jonka pohjalta on suunniteltu PDF-raporttien luontiin tarkoitetun komponentin arkkitehtuuri. Arkkitehtuurin pohjalta toteutettiin toiminnallisuus ja tunnistettiin alkuperäiseen suunnitelmaan liittyneitä epäkohtia ja mahdollisia jatkokehityskohteita.

ABSTRACT

JANI PIIPPO: Modernizing reporting in crane inspections
Tampere University of Technology
Diplomityö, 49 pages
October 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Hannu-Matti Järvinen
Keywords: crane inspections, reporting, modernization, digital transformation, mobile application, browser application

Digital transformation drives the society and organizations to adapt digital technologies and seek benefit from them in business and other operations. With the help of digital technologies, it is possible to increase productivity and gain economic benefit.

One form of digital transformation is the increase in automated and digital information processing. As an example, paper is being substituted in business processes. Different kinds of reporting, also in paper format, are an integral part of many businesses' operations.

Well implemented reporting increases productivity, and employee happiness, as well as improves decision-making abilities, and intra- and interorganizational communication. Therefore efficient reporting can be beneficial in many fields.

In this Master's thesis, a port crane inspection and reporting system is modernized. The focus is mainly on the technical planning and implementation of the functionality responsible for the report creation from input data as a part of a browser application. A PDF report is created as a result of several steps, further described in the thesis.

Firstly, the requiments for the functionality has been recognized, upon which the architecture of the component creating the PDF reports has been planned. The functionality was implemented according to the architecture, although need for revision, as well as aspects for further development were recognized and partially implemented.

ESIPUHE

Tämä diplomityö on kirjoitettu kesän ja syksyn 2018 aikana työskennellessäni Wapice Oy:n Tampereen toimistolla. Olin kuulostellut sopivaa diplomityön aihetta työsuhteeni alusta lähtien. Palaset loksahtelivat erinomaisesti kohdilleen, kun ensimmäinen asiakasprojektini mahdollisti diplomityön tekemisen teknologioiden parissa, joista olin kiinnostunut ja minulla oli kokemusta.

Haluan kiittää työni ohjaajaa Tommi Aittokalliota mielekkäästä projektista sekä toteutukseen annetusta vapaudesta ja vastuusta. Kiitos myös Juuso Virnekselle erinomaisesta yhteistyöstä. Ammattilaisten kanssa on ilo työskennellä ja oppia uutta.

Haluan kiittää myös Tuomas Kivistöä ja Konecranesia sujuvasta yhteistyöstä ja työn mahdollistamisesta. Kiitos myös työn tarkastajalle Hannu-Matti Järviselle ohjauksesta ja neuvoista työn aikana.

Lopuksi haluan kiittää vanhempiani kaikesta tuesta opiskeluaikanani. Kiitos myös kaikille ystävilleni ja opiskelutovereilleni, joiden kanssa olen saanut kasvaa. Kiitos myös Olgalle, joka jaksaa muistuttaa siitä, mikä on tärkeintä.

Tampereella 27.9.2018

Jani Piippo

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

doc	Microsoft Word-tekstinkäsittelyohjelman tiedostomuoto
FOP	Formatted Objects Processor
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MMP	Main Man Ports
Oyj	Julkinen osakeyhtiö
PDF	Portable Document Format
REST	Representational State Transfer
SQL	Structured Query Language
TTY	Tampereen teknillinen yliopisto
XML	Extensible Markup Language
XPath	XML Path Language
XSL	Extensible Stylesheet Language
XSL-FO	Extensible Stylesheet Language Formatting Objects
XSLT	Extensible Stylesheet Language Transformations

1. JOHDANTO

Digitaalinen muutos (engl. digital transformation) on digitaalisten teknologioiden alati lisääntyvää hyödyntämistä liike- ja muussa ihmisen toiminnassa tuottavuuden ja uuden arvon lisäämiseksi [1]. Yhteiskunta ja yhteisöt, erityisesti yritykset, ovat tutkineet viime vuosina mahdollisuuksia muuntaa avainliiketoimintaoperaatioitaan ja prosessejaan digitaalisten teknologioiden avulla saavuttaakseen lisähyötyä [2]. Digitaalisen muutoksen ja digitalisoinnin potentiaaliset hyödyt ovat merkittäviä, ja digitaalisen muutoksen tavoitteena onkin yleensä taloudellinen hyöty. Digitaalisen muutoksen keskiössä ovat erilaiset ohjelmistoratkaisut ja -teknologiat. [1]

Prosesseja voidaan uudistaa ohjelmistoteknologian avulla esimerkiksi korvaamalla paperin käyttö erilaisilla ohjelmistoilla. Sähköinen tiedonkäsittely on usein edullisempaa vähentyneen paperinkäytön takia. Toisaalta sähköisten järjestelmien käyttö sallii nopean tiedonvälityksen tietoverkkojen yli nopeuttaen näin prosesseja ja lisäten tuottavuutta. Lisäksi, mikäli prosessissa käytetään paperia ja sähköisiä järjestelmiä sekaisin, voidaan sähköistämällä koko prosessi edelleen suoraviivaistaa prosessia ja mahdollisesti karsia ylimääräisiä työvaiheita kasvattaen edelleen tuottavuutta.

Eräs esimerkki tällaisesta sähköistettävästä prosessista on Konecranes Oyj:n nosturihuolto-organisaation huollon suunnittelu- ja raportointiprosessi. Nykyisessä ratkaisussa huoltoja suunnitellaan sähköisessä järjestelmässä, mutta prosessin aikana huoltosuunnitelma tulostetaan paperille, tulokset täytetään kyseiselle paperilomakkeelle, jonka jälkeen paperilomakkeelle kirjatut tulokset kirjataan uudelleen käsin sähköiseen järjestelmään, jossa suunnittelu tapahtui. Järjestelmässä voidaan tulostaa raportteja PDF-formaatissa, jotka edelleen lähetetään eteenpäin sidosryhmille joko sähköisesti tai paperille tulostettuna.

Raportointi on kriittinen osa mitä tahansa liiketoimintaa. Hyvin toteutetun raportoinnin hyötyihin lukeutuvat lisääntynyt tuottavuus, työntekijöiden tyytyväisyys, parantunut päätöksentekokyky sekä parempi organisaation sisäinen ja organisaatioidenvälinen kommunikaatio ja yhteistyö. Toisaalta huonosti toimiva raportointi johtaa heikompaan päätöksentekoon, epäluottamukseen tietoa kohtaan, vähentyneeseen tuottavuuteen ja vähentyneeseen työntekijöiden tyytyväisyyteen. [3]

Tähän liittyen totetutetaan nosturihuollon suunnittelu- ja raporointijärjestelmän modernisointi siten, että paperilomakkeet korvataan mobiili- ja selainsovelluksin. Modernisoituun järjestelmään on toteutettava soveltuvilta osin vanhaa vastaava raportointien tulostustoiminnallisuus, kuitenkin päivittäen sitä asiakkaan vaatimusten pohjalta.

Tämän työn päätavoitteena on suunnitella ja toteuttaa mahdollisimman yleiskäyttöinen ja laajennettava palvelu PDF-raporttien luontia varten. Työn alitavoitteena on selvittää ja punnita erilaisia jo olemassa olevia ja muita mahdollisia toteutusvaihtoehtoja. Työn toteuttamisen yhteydessä toteutetaan myös käyttöliittymä toteutettavan raportointikokonaisuuden käyttöä varten, mutta käyttöliittymän toteutus on työn kannalta pienemmällä prioriteetilla.

Tämä työ on suunnittelu- ja toteutusprojekti. Työssä kartoitetaan asiakasvaatimukset ja muut rajoitteet, jonka pohjalta suunnitellaan ja toteutetaan raportointikokonaisuus. Työ on rajattu siten, että toteutettu kokonaisuus kuvataan laajemmin korkeammalla tasolla, ja työhön on valittu lähempään tarkasteluun toteutuksen kannalta merkittävimmät havainnot ja huomiot.

Työ on jaettu seitsemään lukuun. Johdannon jälkeen toisessa luvussa esitellään asiakkaan toimintaympäristö, nykyinen järjestelmä ja sen puutteet sekä asiakkaan vaatimukset uudelle järjestelmälle. Kolmannessa luvussa käsitellään toteutettavan uuden järjestelmän erityispiirteet sekä vaatimukset ja rajoitteet uudessa järjestelmässä tuotettaville raporteille. Neljännessä luvussa määritellään PDF-raporttien luontiin tarkoitetun kokonaisuuden arkkitehtuuri perustuen sekä asiakkaan vaatimuksiin että toteutettavan järjestelmän teknologiavalintojen aiheuttamiin rajoituksiin. Viidennessä luvussa käsitellään tarkemmin komponentin toteutuksen yksityiskohdat, ja kuudennessa luvussa arvioidaan lopputuloksen sopivuutta. Viimeisessä seitsemännessä luvussa vedetään työn sisältö yhteen ja esitellään ideoita komponentin jatkokehitystä varten.

2. SATAMANOSTUREIDEN HUOLLON RAPORTOINTISOVELLUS

Tässä työssä toteutettava raportinluontipalvelu liittyy isompaan ohjelmistokokonaisuuteen. Kirjoittaja on ollut mukana toteuttamassa Konecranesille heidän tuotteidensa huollossa käytettävää järjestelmää ja on ollut pääasiallisessa vastuussa järjestelmän mobiili- ja selainsovelluksista. Toimitettava järjestelmä on satamanostureiden huollossa käytetty huoltotarkastusten suunnittelu- ja raportointisovellus. Järjestelmän mobiili- ja selainsovelluksia käytetään erilaisten huoltotarkastusraporttien luomiseen, täyttämiseen ja tuottamiseen. Järjestelmä tuottaa lopputuotteenaan PDF-muotoisia loppuraportteja, joten tässä työssä käsiteltävä PDF-raporttien muodostukseen ja tulostukseen tarkoitettu palvelu on järjestelmässä keskeisessä roolissa.

2.1 Taustaa

Konecranes Oyj on vuonna 1994 perustettu ja vuonna 1996 pörssiin listautunut suomalainen pörssiyhtiö, joka valmistaa erilaisia nostolaitteita. Konecranes toimittaa nostolaitteita ja palveluita muun muassa konepaja- ja prosessiteollisuudelle, telakoille, terminaaleihin ja satamille [4]. Yritys työllistää noin 17000 työntekijää 600 paikassa 50 maassa. [5]

Konecranesin liiketoiminta koostuu kolmesta liiketoiminta-alueesta: teollisuuslaitteista, satamaratkaisuista sekä palveluista. Teollisuuslaiteliiketoiminta-alue tarjoaa nostolaitteita ja materiaalinkäsittelyratkaisuja muun muassa puu- ja paperi-, auto-, metallintuotanto- ja jätteenpolttoteollisuuden tarpeisiin. Satamaratkaisuliiketoiminta-alue tarjoaa laajan valikoiman kontinhallinta- ja satama-alueenhallintalaitteita, liikkuvia satamanostureita sekä raskaita nosturikuorma-autoja. Satamaratkaisuliiketoiminta-alue tarjoaa myös valikoiman tuotteita tukevia palveluita. Palveluliiketoiminta-alue tarjoaa erikoistuneita huoltopalveluja ja varaosia kaikentyyppisiin ja -merkkisiin nosto- ja satamalaitteisiin. Palveluliiketoiminta-alueen huoltopalvelut ovat tässä työssä tuotettavan palvelun ja siihen liittyvän järjestelmän pääasiallisia käyttäjiä. [6]

Kuten kaikki mekaaniset laitteet, myös nostolaitteet vaativat huoltamista. Satamanostolaitteiden huolto on kokonaisvaltainen prosessi aina huollon suunnittelusta sen raportointiin huoltopalveluorganisaation asiakkaille.

2.1.1 Huoltotarkastusprosessi

Konecranesin huoltopalveluorganisaation vastuulla on yrityksen sekä muiden valmistajien valmistamien satamanostolaitteiden huoltotarkastusten suunnittelu, toteutus ja raportointi loppuasiakkaalle maailmanlaajuisesti useissa eri kohteissa. Huolto-organisaatiolla on huollettavanaan laaja valikoima erilaisia nostolaitteita ja niiden osakokonaisuuksia.

Kullakin nostolaitteella on omat huoltotarkastusaikataulunsa ja -tarpeensa. Kunkin nostolaitteen osakokonaisuus voidaan tarkastaa niin kutsuttuna kertatarkastuksena (engl. one time) tai niistä voidaan tarkastaa sähköosien (engl. electrical), mekaniikkaan (engl. mechanics), rakenteiden (engl. structural) tai turvallisuuden (engl. safety) näkökulmasta.

Huolto-organisaation huoltopäälliköt suunnittelevat huoltotarkastuskäyntejä (engl. inspection). Huoltotarkastuskäynneillä tarkastetaan nostolaitteen kuntoon ja turvallisuuteen liittyviä asioita. Huoltopäällikkö suunnittelee ja valitsee etukäteen, mitkä nostolaitteen osakokonaisuudet tarkastetaan sekä mistä näkökulmasta huoltotarkastus suoritetaan. Kuhunkin nostolaitteen osakokonaisuuteen liittyvä näkökulma muodostaa niin kutsutun huoltotarkastustyön (engl. inspection job). Esimerkiksi nostolaitteen puomin turvallisuuteen liittyvä tarkastus on yksi huoltotarkastustyö. Nostolaitteen puomille voitaisiin suorittaa myös esimerkiksi rakenteisiin liittyvä tarkastus, joka muodostaisi oman huoltotarkastustyönsä. Huoltopäällikkö voi määritellä kohteen huoltotarkastusjärjestyksen, eli toisin sanoen järjestyksen, jossa erilaiset huoltotarkastustyöt käydään läpi. Huoltopäällikkö voi tämän jälkeen osoittaa huoltotarkastuskäynnin jollekin huolto-organisaation huoltajalle hänen vastuulleen suoritettavaksi. Huolto-organisaation huoltajat menevät paikalle, suorittavat tarkastuksen ja koostavat tämän jälkeen huoltotarkastuskäynnin tuloksista raportin toimitettavaksi loppuasiakkaalle, joka käyttää tarkastuksen kohteena ollutta nostolaitetta.

2.1.2 Vanha järjestelmä

Konecranesilla (myöhemmin työssä yritykseen viitataan asiakkaana) on käytössään viime vuosikymmenen puolivälissä valmistunut järjestelmä satamanostolaitteiden huollon suunnitteluun ja raportointiin. Järjestelmä koostuu tietokannasta, Java-palvelimesta sekä Java-asiakassovelluksesta.

Java on laajalti käytetty yleiskäyttöinen oliopohjainen ohjelmointikieli. Java on nykyisin Oraclen omistaman Sun Microsystemsin kehittämä, ja sen kantava perusajatus on, että toteutuksessa on mahdollisimman vähän ulkoisia riippuvuuksia, ja lisäksi sovelluskehittäjät voivat kirjoittaa sovelluksen kerran ja ajaa missä vain. Tämä tarkoittaa sitä, että kerran käännetty Java-sovellus on mahdollista ajaa kaikilla alustoilla ilman uudelleenkääntämistä. Tämä tapahtuu siten, että käännetty Java-sovellus ajetaan Javan virtuaalikoneella (JVM, engl. Java Virtual Machine), joka puolestaan toimii riippumatta laitteistosta. Java on maailman suosituin ohjelmointikieli noin 15 prosentin osuudellaan [7].

Käytössä on Java-palvelinohjelmisto, joka välittää dataa tietokannasta asiakassovellukselle. Palvelinohjelmisto on ajossa virtuaalipalvelimella, jossa on Linux-pohjainen käyttöjärjestelmä. Vanha järjestelmä käyttää tietokannassaan Microsoftin SQL Server-tietokannan hallintajärjestelmää.

MMP, eli MainManPorts, on nykyinen asiakkaan käytössä oleva asiakassovellus. MMP on vuonna 2006 valmistunut ohjelmisto satamanostureiden huollon- ja palveluasiakkuuden hallintaan. Kyseisessä sovelluksessa käyttäjä tekee ensin huoltotarkastussuunnitelman, johon hän valitsee aiemmin kuvatun huoltotarkastusprosessin mukaisesti tarkastuskäyntiin liittyviä asioita, kuten mitkä huoltotarkastuksen vaiheet ja toimenpiteet suoritetaan ja mihin nosturin osaan kukin huoltotarkastus kohdistuu. Suunnitelman valmistuttua vastuussa oleva huoltaja tulostaa suunnitelman paperille, ja lähtee paperin kanssa huoltotarkastuskäynnille kohteeseen, jossa nostolaite sijaitsee.

Huoltaessaan nostolaitetta käyttäjä täyttää huoltotarkastuksen tulokset tulostamilleen paperilomakkeille. Tarkastukseen liittyen huoltaja voi ottaa valokuvia. Huoltotarkastuskäynnin päätteeksi käyttäjä palaa toimistolle ja kirjaa tarkastuskäynnin tulokset käsin asiakassovellukseen. Käyttäjä voi liittää mahdolliset valokuvat osaksi tuloksia. Tämän päätteeksi käyttäjä tulostaa sovelluksesta loppuraportin, joka toimitetaan prosessin mukaisesti loppuasiakkaalle sekä arkistoidaan tarpeen mukaisesti myös asiakkaan arkistoihin.

Teknisesti huoltotarkastuskäynti voi olla erilaisissa vaiheissa. Kun tarkastusta suunnitellaan sovelluksessa, se on luonnostilassa (engl. draft). Kun suunnitelma on saatu valmiiksi ja tulostettu paperille, tarkastuskäynti siirtyy käsittelyssä-tilaan (engl. in progress). Kun tarkastuskäynti on tehty, se siirtyy valmis-tilaan (engl. finished), jolloin tarkastuskäynnin tuloksia voidaan vielä muokata, ennen kuin tulokset lähetetään tarkastettavaksi (engl. ready for review). Kun huoltotarkastuskäynnin tulokset on tarkastettu, siirtyy tarkastuskäynti tarkastettu-tilaan (engl. reviewed) ja tällöin

yleensä tulostetaan loppuraportti lähetettäväksi asiakkaalle.

2.2 Raportoinnin merkitys ja raportointi ohjelmistoissa

Yleisellä tasolla raportti on asiatyylinen, muodollinen, järjestetty ja tiivis esitys jostain lähtödatasta [8][9]. Raportti voi olla selonteko tai tilannekatsaus, ja sen tehtävänä on dokumentoida tapahtumia ja välittää tietoa [8]. Raportit tyydyttävät tiedontarvetta monenlaisissa yleisöissä. Raportteja voidaan käyttää tiedon seuraamiseen, strategian arvioimiseen tai päätöksentekoon.

Raportit usein hyödyntävät elementtejä, kuten tekstiä, kuvia, taulukoita, kuvaajia ja muuta grafiikkaa sekä tiettyä sanastoa [10]. Raportin tavoitteena on usein vakuuttaa lukijakuntansa, ja tiedottaa lukijaansa ja saada heidät toimimaan toimitetun tiedon perusteella [11].

Hyvin suunniteltu ja toteutettu raportointi voi edesauttaa raporttien lukijaa kuluttamaan tietoa tehokkaasti, käyttämään tietoa ennakoivaan päätöksentekoon ja saavuttamaan tätä kautta kilpailuetua. Hienostuneen ja innovatiivisen raportoinnnin tuottaminen on kriittistä kaikille organisaatioille. Riittävä raportointi ja ajantasaisen tiedon käsittely auttaa tekemään paremmin perusteltuja ja todisteisiin perustuvia päätöksiä. [3]

Hyvin toteutettu raportointi vaatii luotettavan datan keräämistä mielellään useista lähteistä ja kokoamista luotettavaan ja ymmärrettävään muotoon. Raportoinnissa on päätettävä, minkätyyppistä dataa käytetään, missä muodossa sitä esitetään, millä aikajänteellä raportointi suoritetaan ja mitä sisältöä raportin kuluttajalle esitetään. [3]

Hyvin toteutetun raportoinnin hyötyjä ovat hyvin kohdennettu datan ja tiedon välitys, lisääntynyt tuottavuus, työntekijöiden tyytyväisyys, parantunut päätöksentekokyky sekä parempi organisaation sisäinen ja organisaatioidenvälinen kommunkaatio ja yhteistyö. Esimerkiksi tuottavuus lisääntyy, kun oikeaa tietoa saadaan toimitettua oikeassa muodossa kuluttajan haluamalla tavalla oikeaan aikaan. Sujuva tiedon kuluttaminen lisää tyytyväisyyttä, kun kuluttajan ei tarvitse käyttää aikaa tai energiaa tiedon keräämiseen, vaan he voivat keskittyä päätöksentekoon. Helposti saatavilla oleva oikea-aikainen -ja muotoinen data on hyvä päätöksenteon pohja. Lisäksi tehokas raportointi tehostaa ja helpottaa tiedonvaihtoa organisaation sisällä ja niiden välillä. Puolestaan huonosti toimiva raportointi aiheuttaa päinvastaisen lopputuloksen: kuluttaja kokee informaatioähkyä huonosti saatavilla olevasta ja järjestellystä datasta, huono raportointi johtaa heikompaan päätöksentekoon, epäluottamukseen

tietoa kohtaan, vähentyneeseen tuottavuuteen ja työntekijöiden tyytyväisyyteen. [3]

Toimiva huollon suunnittelu, seuranta ja raportointi on tärkeää myös satamanostureiden huoltoliiketoiminnassa. Ilman toimivaa raportointia huollon suunnittelu ja seuranta on vaikeaa ja jopa mahdotonta. Puutteellinen suunnittelu johtaa laiminlyönteihin ja virheisiin huollossa. Laitteisto, jonka huolto on laiminlyöty, menee todennäköisemmin rikki kuin huollettu. Huonosti huollettu ja rikkinäinen satamanosturikalusto johtaa alentuneeseen tuottavuuteen, viivästyksiin, kohonneisiin huolto- ja korjauskustannuksiin kaluston käyttäjän osalta sekä imago-ongelmiin, lisääntyneisiin sanktioihin ja alentuneeseen luottamukseen kaluston huoltajan osalta.

Erilaisia tapoja tuottaa raportteja ohjelmistoissa

Kuten mainittu, raportti on muodollinen ja järjestetty esitys jostain lähtödatasta. Jotta raportin muodostaminen ongelmana olisi selkeämpi hahmottaa, siihen liittyvät vaiheet on hyvä tunnistaa ja eritellä. Raportoinnin tekninen toteuttaminen voidaan abstrahoida esimerkiksi seuraavasti:

1. lähtödatan kerääminen, koostaminen ja muotoilu
2. lähtödatan sovittaminen ennalta määrättyyn muotoon ja rakenteeseen
3. muotoillun ja jäsennellyn tiedon levitysformaatin valinta
4. valmiin tiedon jakelu.

Esimerkiksi koulussa kirjoitettu kirjallisuusessee voi olla yksi esimerkki raportista. Kirjoittaja lukee kirjan (lähtödata), kirjoittaa esseen, jossa on alku, keskikohta ja loppu (muoto) konseptipaperille (levitysformaatti) ja palauttaa esseen opettajalle (jakelu).

Tämän työn tapauksessa luodaan huoltotarkastusraportteja ohjelmistossa. Erilaisilla ohjelmistoilla luotavat raportit voivat olla kaikkea lokitulosteista ja -tiedostoista erillisiin asiakirjoihin ja interaktiivisiin multimediaa hyödyntäviin kokonaisuuksiin. Tämän työn tapauksessa lokitulosteet eivät ole mielenkiintoinen käyttötapaus, mutta asiakirjat ja muut kokonaisuudet ovat.

Luotaessa raportteja ohjelmallisesti aiemmin kuvatut vaiheet ovat läsnä raportin valmistelu- ja luomisprosessissa. Ohjelmallisessa raporttien luonnissa valitaan raportin sisältämä data, jolle tehdään ohjelmallisesti haluttuja muunnoksia, järjestelyä ja

muuta manipulointia. Käsitelty tieto paketoidaan johonkin tietojärjestelmien ja tietokoneiden ymmärtämään formaattiin. Esimerkkejä tällaisista formaateista voivat olla Microsoft Wordin `.doc`, muun muassa verkkosivuissa käytetty `.html` ja varsin yleinen Portable Document Format, eli `.pdf`.

2.3 Raportointi vanhassa järjestelmässä ja tarve uudelle

Vanhassa järjestelmässä käyttäjä tuottaa MMP-asiakassovelluksen avulla PDF-muotoisia raportteja. Raporttien luontiin liittyvä toiminnallisuus on rakennettu monoliittisesti Java-asiakassovelluksen sisään. Asiakas käyttää raportointitoiminnallisuutta aktiivisesti, mutta järjestelmä alkaa olla vanhentunut, ja kaipaa päivittämistä.

2.3.1 Raportin tuottaminen vanhassa järjestelmässä

Huoltotarkastusraportin luominen ja muodostaminen tapahtuu siten, että aluksi käyttäjä avaa sovelluksessa täyttämänsä huoltotarkastuksen. Huoltotarkastuksen muokkausnäkymässä käyttäjä voi käynnistää käyttöliittymän yläreunassa raportinhallinnan (engl. Report Manager). Raportinhallinnassa käyttäjä voi valita, minkä muotoisia PDF-raportteja hän haluaa täytetyn huoltotarkastuksen pohjalta luotavan.

Mahdollisia raporttityyppejä on kymmenen. Näihin sisältyvät muun muassa ennen huoltotarkastuskäynnille lähtemistä tulostettava huoltotarkastuslista, loppuraportin kansilehti, loppuraportin selitesivu ja erilaisia nosturin huoltotarkastuskäynnin tulosten tulkintaan käytettyjä raportteja.

Käyttäjä voi raportteja valitessaan täyttää tarkastuksen tulostuspäivämäärän sekä tarkastuksen tekijän nimen, jotka tulostuvat valittuihin PDF-asiakirjoihin. Valittuaan haluamansa raporttityypit ja täydennettyään haluamansa tiedot, käyttäjä aloittaa PDF-raporttien tulostusprosessin. Lopputuloksena käyttäjän ruudulle aukeaa PDF-asiakirjojen lukuohjelmassa kaikki hänen valitsemansa raportit erillisinä tiedostoina.

Teknisesti raportin luonti tapahtuu siten, että Java-sovelluksen resursseissa on tallennettuna XSLT-muotoisia tyylitiedostoja, jotka määrittelevät kunkin raporttityypin ulkoasun. XSL, eli eXtensible Stylesheet Language, on World Wide Web Consortiumin kehittämä kieliperhe, jota käytetään XML-pohjaisten asiakirjojen ulkoasujen määrittelyyn tai rakennemuutosten tekemiseen. XSLT, eli XSL Transformations on

kieli XML-asiakirjojen muuntamiseen. XSL-kieliperheeseen kuuluu myös XSL-FO (XSL Formatting Objects), XML-sanakirja muotoilun semantiikkaan, sekä XPath, eli XML Path Language, XSLT:n ja useiden muiden kielien käyttämä kuvauskieli XML-asiakirjan osiin viittamiseen. [12]

Nämä XSLT-tiedostot ovat suhteellisen laajoja ja hankalasti luettavia tiedostoja, joiden ylläpito on hankalaa, etenkin kun alkuperäiset tyylitiedostojen laatijat eivät ole saatavilla. Tyylitiedostoissa on määriteltynä millin tarkkuudella kunkin elementin paikat, leveydet ja korkeudet, sekä niiden asettelu. Muutosten tekeminen näihin määrittelyihin on riskialtista, sillä ei voida taata, pysyvätkö asiakirjojen lopullisen muotoilut halutunlaisina kaikissa tilanteissa muutoksen jälkeen. Esimerkiksi nykyisessä järjestelmässä on erilaisia kieliversioita, ja kussakin kieliversiossa on erimittaisia merkkijonoja, joilla asiakirjan elementtejä populoidaan. Näiden elementtien koot ja sijainnit on jo kertaalleen viilattu toimiviksi tiukkojen vaatimusten mukaan, ja asiakirjat on todettu toimiviksi yli kymmenen vuoden käytön kokemuksen perusteella.

Java-sovellus käynnistää käyttöliittymässä edellä mainitulla tavalla määritellyn prosessin, jossa kutakin raporttia vastaava XSL-tyylitiedosto populoidaan XML-muotoisella datalla. Tämä data on haettu sovelluksen muistiin palvelimelta. XSL-tyylitiedosto populoidaan XML-datalla Javan XML-transformaatikirjaston avulla ja tuloksena syntyy uusi XML-asiakirja. Tämä asiakirja sisältää populoidun data muotoiluineen. Tämä XML-dokumentti sisältää kaiken tiedon A4-muotoisen raportin muodostamiseen joka sivulla toistuvine ylä- ja alatunnisteineen. Seuraavaksi tästä XML-asiakirjasta muodostetaan PDF-dokumentti, joka puolestaan on valmis esitettäväksi asiakkaalle. Tämä prosessi toistetaan kullekin raportille, jonka käyttäjä on halunnut tulostaa järjestelmästä.

2.3.2 Tarve ja vaatimukset uudelle järjestelmälle

Asiakkaan käytössä oleva MMP-järjestelmä on edelleen käytössä, ja sitä käyttää aktiivisesti noin 50 käyttäjää Euroopassa ja Aasiassa. Asiakas kuitenkin kokee, että nykyinen järjestelmä on vanhentunut, ja vain osaa sen toiminnoista käytetään. Asiakas käyttää nykyisen MMP:n toiminnallisuudesta vain lähinnä huoltotarkastuskäyntien suunnitteluun liittyvää toiminnallisuutta, ja uudessa järjestelmässä onkin tarkoituksena jättää pois toiminnallisuutta, jota ei enää käytetä.

Asiakas haluaa nykyaikaistaa ja suoraviivaistaa tarkastuskäyntien suunnittelu-, tarkastus- ja raportointiprosessia päivittämällä sovelluksiaan. Asiakas haluaa päästä eroon paperisista huoltolomakkeista. Asiakas haluaa helpon keinon kirjata muistiin

huollon tulokset ja tulostaa raportin huoltotoimenpiteistä. Lisäksi aiemmin työlääksi koetun kuvien lisäämisen tulisi olla helppoa. Raportin kuuluu olla yksittäinen PDF-raportti, joka sisältää useamman osaraportin. Nämä osaraportit ovat aiemmin lueteltuja raporttityyppejä. Asiakas on pääosin tyytyväinen nykyisiin raportteihinsa ja niiden sisältöön, mutta on esittänyt toiveita pienistä visuaalisista muutoksista.

Asiakkaan tavoitteena on, että uutta modernia järjestelmää käyttäisi noin 350 käyttäjää maailmanlaajuisesti. Asiakas haluaa, että paperilomakkeiden tilalle toteutetaan mobiililaitteilla toimiva sovellus, jonka kautta huoltajat voivat kirjata MMP:llä suunnitellun huoltotarkastuskäynnin tuloksia suoraan järjestelmään sähköisesti. Näin tulosten kahdenkertainen kirjaaminen – ensin paperille, sitten paperilta sovellukseen – saadaan karsittua pois.

Asiakas haluaa myös, että vanhaa MMP:tä päivitetään ja käyttämätöntä toiminnallisuutta karsitaan pois. Vaihtoehtoina on myös Windows 10 -sovelluksen tai uuden selainpohjaisen sovelluksen toteuttaminen. Vanhan järjestelmän päivittämisen hyviä puolia olisivat jo valmis toiminnallisuus eikä tarvetta suurille muutoksille olisi. Huonoja puolia ovat vanhat teknologiavalinnat sekä uuden toiminnallisuuden lisäämisen hankaluus tulevaisuudessa. Windows 10 -sovellus olisi moderni valinta, mutta vaatisi erillisen koodipohjan toteutettavaan mobiilisovellukseen nähden. Selainpohjainen ratkaisu olisi myös moderni, ja toteutus voitaisiin järjestää siten, että se hyödyntäisi samaa koodipohjaa toteutettavan mobiilisovelluksen kanssa, mikä vähentää työmäärää ja lisää houkuttelevuutta. Lisäksi selainsovellusta voi käyttää lähestulkoon kaikilla laitteilla, joissa on selain. Huono puoli on, että offline-tilassa käyttö on hankalaa tai liki mahdotonta.

3. UUSI JÄRJESTELMÄ

Asiakkaan vaatimuksiin vastataan luomalla alustariippumaton mobiilisovellus, selainsovellus, ja näitä tukeva infrastruktuuri. Asiakasvaatimusten mukaisesti mobiilisovellus toimii sekä iOS- että Android-käyttöjärjestelmillä. Uuteen järjestelmään totetutetaan keskeisimmät osat vanhasta asiakassovelluksesta sekä lisätään asiakkaan vaatimusten mukaisesti tuki esimerkiksi huoltotarkastusten tulosten sähköiselle kirjaamiselle sekä uudistetulle PDF-raportin luomiselle.

3.1 Projektin tavoitteet

Sekä asiakas että projektin toteuttava ohjelmistoinsinööritiimi asettivat projektille muutamia tavotteita. Asiakkaalta kerättyjen asiakasvaatimusten ja niistä muodostettujen toiminnallisuusmäärittelyjen lisäksi kumpikin osapuoli tiivisti projektin pääasialliset tavoitteet muutamaan ranskalaiseen viivaan.

Asiakkaan puolesta tavoitteet uuden järjestelmän toteutusprojetissa ovat:

- Vanha MMP täytyy korvata uudella järjestelmällä.
- Käyttäjä haluaa helpon tavan syöttää ja tulostaa tietoa.
- Uuden järjestelmän käytön pitäisi olla helpompaa kuin vanhan.
- Uuden järjestelmän pitäisi olla yhtä luotettava kuin vanha.

Näistä erityisesti tämän työn kannalta oleellisia tavotteita ovat järjestelmän luotettavuus, helppokäyttöisyys ja helppo tapa tulostaa raportteja. Toisaalta vanhan järjestelmän korvaaminen uudella teknologialla mahdollista modernien toimintamallien ja ratkaisujen käyttämisen.

Toteutustiimin asettamat sisäiset tavoitteet ja tärkeimmät arkkitehtoniset suuntaviivat projektissa ovat:

- Kaikesta teknisestä velasta on päästävä eroon.
- Uusi järjestelmä on oltava helposti laajennettava ja ylläpidettävä.
- Alustariippumattomalle mobiilikehitykselle on luovata yleinen toimintamalli.
- Mobiili- ja selainsovelluksille yhteistä koodipohjaa on hyödynnettävä mahdollisimman paljon.
- Mobiilisovellus on toteutettava React Native -sovelluskehystä käyttäen.

Näistä oleellisin tavoite on helposti laajennettava ja ylläpidettävä kokonaisuus. Raportointiin erikoistunut osakokonaisuus voidaan suunnitella ja toteuttaa monella tavalla. Laajennettavuus ja ylläpidettävyys otetaan nyt erityisesti huomioon suunnittelupäätöksiä tehtäessä. Lisäksi toteutustiimi halusi kerätä kokemusta React Native -sovelluskehyksestä [13] erityisesti tuotantokäytössä. React Native esitellään tarkemmin alikohdassa 3.2.2. Tämä tavoite asettaa heti oman rajoitteensa uuden järjestelmän asiakassovellusten teknologioita valittaessa.

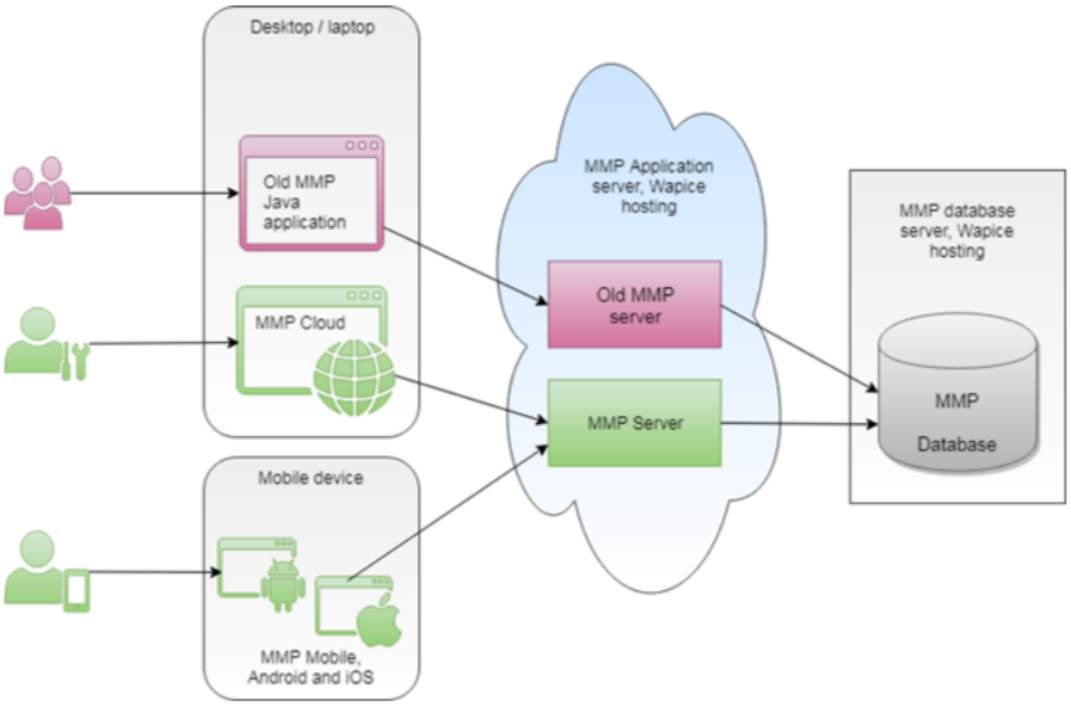
3.2 Uuden järjestelmän yleiskuvaus

Uusi järjestelmä koostuu tietokantapalvelimesta, sovelluspalvelimesta, sekä mobiili- ja selainasiakassovelluksesta. Kuvassa 3.1 on kuvattu vanhan MMP-järjestelmän rakenne punaisella, sekä toteutettavan uuden MMP-järjestelmän osat vihreällä.

Mobiili- ja selainsovellukset kommunikoivat palvelimen kanssa, joka noutaa, tallentaa ja ylläpitää tietoa vanhassa MMP-järjestelmässä käytössä olevaa tietokantaa käyttäen. Mobiili- ja selainsovellukset esittävät tietoa ja mahdollistavat tiedon muokkauksen. Muokattu tieto lähetetään palvelimelle tietokantaan tallennusta varten. Älypuhelimien kamerat mahdollistavat sovelluksen suunnittelemisen siten, että kuvien liittäminen on luontevaa.

3.2.1 Palvelinsovellus

Uuden MMP-järjestelmän palvelinsovellus toteutetaan Java-ohjelmointikielellä käyttäen Spring Boot -ohjelmointikehystä [14]. Palvelinsovellus on suunniteltu laitettavaksi ajoon samaan ympäristöön kuin nykyinen toteutus, eli virtuaalipalvelimelle, jossa on Linux-pohjainen käyttöjärjestelmä.



Kuva 3.1 *MMP-järjestelmän rakenne: uusi ja vanha sovellus, palvelimet ja yhteinen tietokanta*

Asiakassovellusten ja palvelimen välinen kommunikaatio tapahtuu REST-ohjelmointirajapinnan [15] välityksellä. REST-rajapinnassa käytetty tietojen esittämismuoto on JSON (engl. JavaScript Object Notation, yleisesti käytetty tiedon esittämismuoto) [16].

3.2.2 Selain- ja mobiilisovellus

Projektin toteutustiimin sisäiset tavoitteet projektille määrittivät, että projektista kerätään kokemusta mobiilisovellusten kehittämisessä käytetystä ohjelmointikehys React Nativesta, mikä lukitsee mobiilisovellukseen valittavan teknologiaperheen tietenkin React Nativeen.

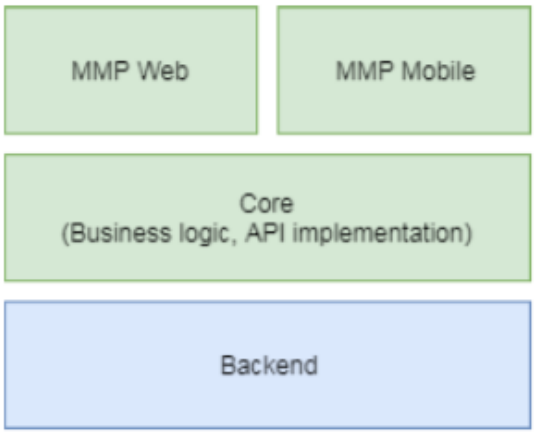
React Native on Facebookin ja avoimen lähdekoodin yhteisön kehittämä ohjelmistokehys alustariippumattomaan mobiilisovelluskehittämiseen. React Native hyödyntää JavaScriptiä ja React-komponenttikirjastoa sovellusten käyttöliittymien luomiseen. Tämä mahdollistaa sen, että sekä Androidille että iOS-käyttöjärjestelmälle voidaan kehittää suoraan natiivisovelluksia JavaScriptillä, ja yksi toteutus kääntyy kummallekin alustalle. React Native käyttää samoja käyttöliittymäkomponentteja kuin Javalla tai Objective-C:llä kirjoitetut natiiviapplikaatiot, mutta ne kootaan vain yhteen JavaScriptin ja React-komponenttikirjaston avulla. React Native -sovellukset

mahdollistavat myös natiivin sovelluskoodin käytön projekteissa, mikäli tarve vaatii. [13] React Nativen käyttö karsii siis tarpeen kirjoittaa sama toiminnallisuus Javalla tai Kotlinilla Androidille ja toisaalta Swiftillä tai Objective-C:llä iOS:lle. Tämä taas säästää kehittämiseen kuluvaa aikaa ja vaivaa.

React Nativen lopputuotteena syntyy puhdas natiivisovellus. Muita tapoja tehdä alustariippumattomia mobiilisovelluksia ovat niin kutsutut hybridiapplikaatiot, mobiili-web-applikaatiot sekä HTML5-sovellukset. Tällaisia sovelluskirjastoja- ja ohjelmointikehyksiä ovat muun muassa Googlen Progressive Web Appit ja PhoneGap. Myös muita ohjelmointikehyksiä ja teknologioita kuin React Native harkittiin mobiilisovellusten perustaksi, mutta projektille asetetut tavoitteet - lähinnä kokemuksen kerääminen React Nativesta tuotantokäytössä - rajasivat valinnan React Nativeen.

Mobiili- ja selainsovellusten haluttiin hyödyntävän lisäksi yhteistä ydintä eli koodipohjaa yhteisille toiminnallisuuksille. Ytimen päälle on toteutettu käyttöliittymäsovellukset kullekin alustalle erikseen. Yhteisen koodipohjan avulla voidaan varmistaa kummankin sovelluksen yhdenmukainen toiminta, ja samalla toisteisen koodin määrä vähenee ja koodi pysyy "kuivana" (engl. DRY - Don't Repeat Yourself). Samalla jatketaan React Nativen hengenperintöä kehitystyössä: yksi toteutus toimii useassa kohteessa, ja kehittämiseen kuluvaa aikaa säästyy, kun samankaltaista toteutusta ei tarvitse kirjoittaa useaan kertaan esimerkiksi eri kielellä.

Mobiili -ja selainsovellusten ydin sisältää yhteisen toiminnallisuuden sovelluksen tilanhallinnalle ja ohjelmointirajapinta-asiakaskomponentit palvelimen kanssa kommunikointiin. Kuvassa 3.2 on kuvattu asiakasohjelmistojen kerrosarkkitehtuuri.



Kuva 3.2 *Uusi MMP: käyttöliittymäsovellukset, ydinkerros sekä palvelin [17]*

Ydin perustuu Reduxin ja Redux-sagan ympärille. Redux on JavaScriptillä toteutettu kirjasto JavaScript-sovellusten tilanhallintaan. Redux perustuu ennaltaarvattaviin tilan muutoksiin [18]. Redux-saga puolestaan on suunniteltu helpotta-

maan asynkronisten toimintojen, kuten internetin yli tapahtuvien rajapintakutsujen hallinnan helpoksi [19].

Selaimessa ajettavien sovellusten de facto -ohjelmointikieli on JavaScript. Selainsovellusten puolesta teknologiavalintojen vaihtoehdot keskittyivät työn kirjoitushetkellä lähinnä kolmen käyttöliittymäohjelmointikehyksen – Angularin, Vuen ja Reactin – välille.

Koska mobiilisovelluksen ohjelmointikehykseksi lukittiin jo aikaisessa vaiheessa React Native, on luontevaa, että selainsovellus toteutetaan Reactilla, jota käytetään myös React Nativessa keskeisenä osana. React on Facebookin ja yhteisön kehittämä avoimen lähdekoodin JavaScript-kirjasto selainsovellusten käyttöliittymien kehittämiseen.

Käyttämällä Reactia selainsovelluksessa ja React Nativea mobiilisovelluksessa saavutetaan synergiaetua, koska React-kirjaston komponentit toimivat suoraan React Nativessa. Tällöin koodin toisteisuutta tai saman toiminnallisuuden toteuttamista kahdella eri tavalla voidaan edelleen välttää. Koska molemmat ovat Javascript-kirjastoja, ne voivat käyttää yhteistä JavaScriptillä totetutettua ydintä, jolloin myös vaatimus yhteisestä ytimestä täyttyy.

3.3 Muut teknologiarajoitteet

Kehitystyö itsessään tehtiin käyttäen TypeScript-ohjelmointikieltä. TypeScript lisää tyypityksen JavaScriptiin, ja se kääntyy puhtaaksi JavaScriptiksi. Tyypitys helpottaa kehitystyötä ja lisää sovelluksen toimintavarmuutta: kehitysympäristö voidaan pystyttää siten, että ympäristö valvoo ja varoittaa muun muassa erilaisista tyyppiyhteensopivuusvirheistä, puuttuvista tiedoista ja muista ohjelmoijan virheistä, mikä sekä nopeuttaa kehittämistä, kun virheet saadaan selville jo ennen käännöstä tai ajoa, ja toisaalta parantaa laatua, kun esimerkiksi tyyppiyhteensopivuudesta pidetään huolta.

3.4 Tarve ja vaatimukset raporttien tuottamiselle

Asiakkaan keskeisin tavoite ja suurin toive projektille on helppo tietojen kirjaus sekä helppo raporttien tulostus. Raporttien luominen on keskeisessä osassa tätä tavoitetta ja sen tarkempi suunnittelu päätettiin jättää erilliseksi osaksi varsinaisen projektin vaatimusmäärittelystä ja suunnittelusta.

Raporttien luonnin keskeisimmät toiminnallisuuteen ja tuloksena syntyvään raporttiin liittyvät tavoitteet ovat

- Käyttäjä voi valita, mitkä raportit hän sisällyttää valmiiseen PDF-asiakirjaan.
- Käyttäjä voi muokata lopulliseen PDF-asiakirjaan tulevia tiettyjä kenttiä, kuten alaotsikot
- Tuloksena on yksi yhtenäinen raportti, joka sisältää yhden tai useamman aliraportin
- Raporttien ulkoasu ja sisältö täytyy olla pääosin nykyisten raporttien kaltainen
- Raporttien ulkoasun päivittäminen voi olla tulevaisuudessa tarpeen
- PDF-raportoinnin rinnalle voidaan nostaa HTML-raportit, joita voi lukea selaimessa

Lyhyesti sanottuna toiminnallisuus on keskeisiltä osiltaan varsin samankaltainen kuin vanhassa MMP-järjetelmässä. Asiakas on pääosin tyytyväinen nykyiseen raportointiin. Raportointi vaatii kuitenkin hieman päivittämistä ja toteuttamisen kokonaan puhtaalta pöydältä uusin teknologioin. Lisäksi tulevaisuuden laajennettavuus on keskeisessä roolissa uutta raportointitoiminnallisuutta toteutettaessa.

Projektiin valittujen teknologioiden asettamien rajoitteiden takia myös seuraavat vaatimukset oli otettava huomioon:

- Komponentin pitää sisältää tai siihen pitää olla muutoin saatavilla TypeScript-tyypitykset.
- Komponentin pitää toimia ensisijaisesti palvelimella.
- Komponentissa pitää olla valmius toimia myös asiakassovelluksen osana.

Raportit luodaan ensisijaisesti palvelimella. Lopullisessa toteutuksessa loppukäyttäjä pyytää asiakassovelluksen kautta palvelinta luomaan loppukäyttäjän valitsemasta datasta raportin. Uuden järjestelmän tapauksessa palvelinpäässä toimiva raportinluontikomponentti olisi loogista toteuttaa Java-ohjelmointikielellä ja sulauttaa komponentti osaksi palvelinohjelmistoa.

Asiakas on kuitenkin esittänyt ajatuksia siitä, että loppuraportteja voisi hyvä olla mahdollista tuottaa myös silloin, kun internet-yhteyttä ei ole saatavilla. Tällöin

asiakasohjelmat, joita loppukäyttäjä käyttää, eivät tietenkään saa yhteyttä palvelimelle, joten raportinluontikomponentin olisi tällöin oltava osana asiakassovellusta, eikä komponentin sulauttaminen osaksi palvelinohjelmistoa ole järkevää. Koska asiakassovellukset ovat React- ja React Native -sovelluksia, ja ne ovat toteutettu tässä projektissa TypeScriptillä, Javan käyttäminen kehitystyössä ei ole mahdollista. Toisaalta JavaScriptin tai TypeScriptin käyttäminen mahdollistaa komponentin käyttämisen sekä palvelimella että asiakassovelluksissa. Komponentti voidaan ottaa käyttöön sellaisenaan kirjastona asiakassovelluksessa tai laittaa ajoon palvelimella esimerkiksi Node-ajoympäristössä. Node-ajoympäristön käyttöä pohditaan tarkemmin kohdassa 4.3

Koska asiakassovellusten kehitystyö tehdään pääasiassa TypeScriptillä, on hyvä varautua siihen, että sen täytyy sopia yhteen muiden TypeScript-totetutuksen kanssa. Mikäli komponenttia jossain tilanteessa täytyy käyttää esimerkiksi TypeScriptillä toteutetun asiakasohjelmiston osana, on komponentin sisällytettävä TypeScript-tyypitykset. Asiakasohjelmistojen kehitystyön aikana on noussut toisinaan rajoitteeksi se, että jotkin kirjastot eivät sisällä TypeScript-tyypityksiä, mikä puolestaan estää kirjaston käytön sovelluskehityksessä, kun kehitysympäristö ja kääntäjä eivät salli niiden käyttöä. TypeScriptin kääntäjä ei osaa tulkita oikein koodia kirjastossa, josta puuttuu tyypitykset.

Suunnittelu- ja kehitystyössä on hyvä ottaa huomioon myös mahdollinen React-yhteensopivuus ja siihen liittyvät mahdolliset haasteet. Järjestelmän tuottamat PDF-raportit tullaan toimittamaan loppukäyttäjän kulutettavaksi nimenomaan React- ja React Native -sovellusten kautta.

Asiakassovellukset käyttävät sisäisenä datan esitysmuotonaan ja kommunikointiin palvelimen kanssa JSON-notaatiota. Tästä syystä myös raportointikomponentin – jotta se tukisi käyttöä sekä palvelimella että asiakassovelluksen osana – on todennäköisesti järkevää tukea JSON-muotoista lähtödataa. Myös XML-formaatti on yleinen palvelin-asiakas-sovelluksissa ja REST-rajapinnoissa käytetty datan esitysmuoto. Siksi XML on toissijainen tuettava datan esitysmuoto.

3.5 Valmiit ratkaisut

Asiakkaan vaatimusten ja uuden järjestelmän teknisten vaatimusten pohjalta tutkittiin, onko PDF-raporttien tuottamiseen lähtödatasta olemassa valmiita ratkaisuja, joita pystyttäisiin käyttämään suoraan ongelman ratkaisemiseen. Lyhyesti sanottuna ratkaisun pitäisi ottaa sisäänsä JSON- tai XML-muotoista dataa ja muodostaa muutaman välivaiheen jälkeen PDF-raportti.

Tärkeimpiä valintakriteerejä olivat XSLT-kielen tuki, sillä toteutuksen ensimmäisessä vaiheessa tavoitteena on hyödyntää vanhoja asiakirjapohjia mahdollisimman paljon, TypeScript-yhteensopivuus sekä laajennettavuus esimerkiksi HTML- tai XML-muotoisiin asiakirjapohjiin. Myös React-tuen mahdollisuutta tutkittiin, mutta sen painoarvo oli vähäisempi.

Havaittiin, että saatavilla on laaja valikoima erilaisia avoimen lähdekoodin kirjastoja, jotka tukevat muun muassa HTML- tai XML-muotoisen datan muuttamista PDF-dokumentiksi. Hakemalla internetin hakukoneilla hakusanoilla kuten "react generate pdf", "typescript generate pdf", "react pdf", "javascript pdf", "typescript xml pdf", avoimen lähdekoodin säilytyspaikoista (engl. repository), kuten GitHub, löytyi muutamia mielenkiintoisia kirjastoja aiheeseen liittyen.

React-pdfs on Lang.ai:n ylläpitämä React-komponenttikirjastoon yhteensopivaksi tarkoitettu PDF-dokumentteja muodostava kirjasto. [20] Lang.ai on espanjalainen yritys, joka kehittää kielen ymmärtämiseen tarkoitettua valvomatonta keinoälyä (engl. AI - Artificial Intelligence). [21] Lang.ai:n React-pdfs tarjoaa hyvin niukasti esimerkkejä ja tietoa siitä, kuinka erilaisiin käyttötapauksiin kirjasto on laajennettavissa. Esimerkkien ja muun annetun tiedon perusteella vaikuttaa siltä, että kirjastolla on mahdollista muodostaa vain tietyn muotoisia PDF-dokumentteja, tarkemmin sanottuna laskuja. Kirjasto ei siten sovellu työn käyttötarkoitukseen.

Diego Muracciolen GitHub-säilytyspaikassa julkaisema React-pdf on PDF-raporttien tekijä ja esittäjä Reactille. Muracciolen React-pdf käyttää omia sisäänrakennettuja React-tageja sisällön jäsentämiseen dokumenttimuotoon, jonka jälkeen sisältö voidaan esittää dokumenttioliomallina (engl. DOM - Document Object Model, yleinen tapa kuvata esimerkiksi HTML-dokumentin rakenne) ja tarjota se selaimen kautta käyttäjälle kulutettavaksi, tai sisällön perusteella on mahdollista tallentaa dokumenttioliomalli PDF-tiedostoksi. Muracciolen React-pdf tarjoaa myös erilaisia keinoja muotoilla dokumentin sisältöä, sillä kirjasto käyttää Reactin tyylioliota sisällön muodon määrittelyyn. Reactin tyylioliot ovat syntaksiltaan hyvin samankaltaisia kuin CSS-tyylitiedostojen syntaksi. [22] Muracciolen React-pdf sopii myös käytettäväksi sekä selaimessa että palvelimella, ja tuki mobiilikäyttöön on kehityksessä. [23]

Wojciech Majn kehittämä Muracciolen kirjaston kanssa samanniminen React-pdf puolestaan on vain PDF-dokumenttien esittämiseen tarkoitettu kirjasto React-sovelluksille [24]. Koska kirjastosta puuttuu PDF-raportin luontiin liittyvä toiminnallisuus, ei kyseinen kirjasto ole työn käyttötarkoituksessa hyödyllinen. Kirjasto voi olla tarpeellinen siinä vaiheessa, kun uudessa järjestelmässä luotuja PDF-raportteja

aiotaan esittää käyttäjälle.

PDF.JS on Mozilla Labsin kehittämä PDF-dokumenttien esittäjä [25]. Kuten Wojciech Majn kirjaston tapauksessa, myös Mozillan kirjasto on vain PDF-dokumenttien esittämistä varten. Siitä puuttuu PDF-dokumenttien luomiseen tarkoitettu toiminnallisuus, eikä siten ole hyödynnettävissä suoraan työn puitteissa.

Generate-pdf on RMA Consulting-nimisen yrityksen kehittämä avoimen lähdekoodin kirjasto [26]. RMA Consulting on isobritannialainen Lontoossa toimiva ohjelmistokonsulttiyritys. [27] Generate-pdf on kirjasto, joka luo PDF-dokumentteja verkkosivuista lupauksiin (engl. Promise) perustuvan rajapinnan välityksellä. Generate-pdf:n tapauksessa on mahdollista käyttää HTML-muotoista lähtödataa, jonka pohjalta PDF-dokumentteja muodostetaan. Kirjastoa käytetään siten, että se käynnistää Express-palvelimen, ja itse palvelua käytetään Promise-rajapinnan kautta. Kirjasto käyttää Marc Bachmannin Node-html-pdf -kirjastoa itse PDF-tiedoston muodostamiseen HTML-datasta.

Marc Bachmannin Node-html-pdf muuttaa HTML-muotoisia asiakirjoja PDF-asiakirjoiksi. [28] Node-hmtl-pdf poikkeaa edellä esitellyistä siten, että React-tukea tai -pakkoa ei ole, ja siksi kirjasto on hyvin yksinkertainen ja vaikuttaa kevyeltä ratkaisulta. Node-html-pdf hyödyntää PhantomJS:ää itse PDF-asiakirjan muodostamiseen.

PhantomJS on Ariya Hidayatin kehittämä headless-tyyppinen selain. Headless-tyyppisissä selaimissa ei ole graafista käyttöliittymää. PhantomJS on QtWebKitin [29] päälle rakennettu selain, jolle pystyy kirjoittamaan skriptejä JavaScriptillä. Headless-tyyppisiä selaimia pystyy käyttämään monenlaiseen verkkosivujen automatisoituun hallintaan, kuten muun muassa testiautomaatiossa, käyttäjän interaktion automatisointiin sekä kuvakaappausten ottamiseen verkkosivusta. Esimerkiksi Marc Bachmannin Node-html-pdf -kirjaston tapauksessa PhantomJS:ää hyödynnetään juuri siten, että HTML-asiakirjan määrittelemästä verkkosivusta otetaan kuvakaappaus, ja se muunnetaan PDF-tiedostomuotoon. PhantomJS:n kehitystyö on kuitenkin keskeytetty 3. maaliskuuta 2018. Tämä on hyvä ottaa huomioon, mikäli PhantomJS-kirjastoa aikoo hyödyntää osana omaa projektiaan. Tuen jatkuminen ja päivitysten saatavuus lienee heikkoa. [30]

Puppeteer on Node-kirjasto, joka tarjoaa korkean tason ohjelmointirajapinnan Googlen Chromen tai sen headless-versio Chromiumin käyttämiseksi. Puppeteeriä voi käyttää PhantomJS:n tavoin verkkosivuautomaation järjestelmiseen, testiautomaatioon sekä kuvakaappausten ja PDF-tiedostojen muodostamiseen verkkosivuista. [31] PhantomJS:n tavoin Puppeteer soveltuu kirjaston ytimeksi, jos sille syötettä-

vä data on HTML-muodossa. Puppeteer on Googlen kehittämä avoimen lähdekoodin projekti, jonka kehitystyö on jatkuvaa, mikä tekee Puppeteeristä houkuttelevamman vaihtoehdon käytettäväksi omassa projektissa kuin PhantomJS.

Joseph Hasselin XML-PDF muuntaa korkean tason XML-tiedostoja PDF-tiedostoiksi. [32] Kirjasto on hyvin yksinkertainen ja kevyt, se itsessään koostuu 141 rivistä koodia, mutta käyttää riippuvuuksinaan muun muassa Marc Bachmannin Node-html-pdf-kirjastoa sekä Marek Kubican xml2js-kirjastoa XML-muotoisen datan muuntamiseen JavaScript-olioiksi. Xml-pdf voi olla toteutustyössä hyödyllinen, mutta ei suoraan ratkaise ongelmaa.

PDFKit on PDF-asiakirjojen muodostamiseen tarkoitettu kirjasto Node-ajoympäristöön ja selaimeen. [33] Se on kirjoitettu CoffeeScriptillä, mutta sen rajapintaa voi käyttää JavaScript-projekteissa. PDFKit sopii kirjaston moottoriksi, mutta ei vastaa toteutettavan toiminnallisuuden tarpeisiin suoraan.

Mikään tutkituista valmiista ratkaisuista ei tyydytä suoraan vaatimuksia. Tutkittuja kirjastoja voidaan käyttää joko inspiraation lähteenä tai uuden toiminnallisuuden osana. Esimerkiksi muuntamalla datan esitysmuotoa sopivassa kohtaa, esimerkiksi XSLT-asiakirjapohjasta ja datasta luodusta XML:stä HTML:ään, voidaan edellä mainittuja kirjastoja mahdollisesti hyödyntää toteutettavan komponentin osana.

4. RAPORTOINTIPALVELUN ARKKITEHTUURI

Raporttien luonnista vastaavasta kokonaisuudesta halutaan etenkin projektitiimin sisäisten tavotteiden ja toisaalta yleisten arkkitehtonisten hyvien toimintamallien tapaan tehdä mahdollisimman yleiskäyttöinen, laajennettava ja helposti ylläpidettävä. Raportointitoiminnallsuuden olisi hyvä tukea useita ajoympäristöjä, useita lähtödatan esitysmuotoja, useita asiakirjojen muotoa ja rakennetta määrittelevien asiakirjapohjien formaatteja sekä useita raporttien ulostuloformaatteja.

4.1 Raporttien luonti osana uutta järjestelmää

Toteutuksen ensimmäisessä vaiheessa raportteja tuottava kokonaisuus tuottaa PDF-muotoisia raportteja, joita asiakas kutsuu loppuraporteiksi (engl. Final Report). Tuotettavien loppuraporttien muoto ja sisältö ovat keskeisiltä osin samat, kuin vanhalla järjestelmällä tuotetuissa raporteissa. Erona vanhaan järjestelmään on se, että uusi raportteja tuottava toiminnallisuus tuottaa yhden PDF-asiakirjan, joka sisältää yhden tai useamman aliraportin. Vanha järjestelmä tulosti kunkin aliraportin erilliseen PDF-asiakirjaan. Aliraportteihin kuuluuvat muun muassa kansilehti, kuvaraportti, nosturin kuntoraportti, nosturin turvallisuusyhteenveto sekä raportin selitesivu. Uutena sisältönä toteutetaan mahdollisuus määrittää käyttöliittymässä niin kutsuttu loppukatselmus (engl. Executive report), jossa käyttäjä saa kirjoittaa vapaata tekstiä loppuraportin viimeiselle sivulle. Lisäksi täytyy olla mahdollista tulostaa erikseen paperinen huoltotarkastuslomake, jonka uuden järjestelmän mobiilisovellus korvaa. Tukea paperilomakkeelle halutaan kuitenkin jatkaa.

Raportteja tuottava kokonaisuus tuottaa PDF-muotoisen loppuraportin palvelimella ja toimittaa sen asiakasohjelmistolle loppukäyttäjän käytettäväksi.

Uudesta toteutuksesta jätetään muutamia osuuksia pois verrattuna vanhaan järjestelmään. Uuden toteutuksen raportit ovat vain englanninkielisiä, kun taas vanhassa järjestelmässä oli tuki kahdeksalle kielelle. Myös muutamat aliraporttityypit karsiutuvat uudesta toteutuksesta.

Osana toiminnallisuutta totetutetaan selainsovellukseen käyttöliittymä, jonka avulla käyttäjä voi valita, mitkä aliraportit loppuraporttiin sisällytetään, muokata raportin luojan nimeä ja luontipäivämäärää sekä määrittää edellä mainitun loppukatselmuksen tiedot. Kyseinen käyttöliittymä liittyy oleellisesti toteutukseen ja kokonaisuuteen, mutta on tässä työssä toissijaisella painoarvolla.

Käyttöliittymän suunnittelussa painotetaan sekä käyttöliittymän että käyttökokemuksen selkeyttämistä ja modernisointia. Esimerkiksi vanhassa järjestelmässä voidaan huoltotarkastuksen tilasta riippumatta tulostaa kaikenlaisia raporttityyppejä, kuten huoltotarkastuksen valmistumisen jälkeen käyttöliittymä mahdollistaa huoltotarkastuslistan tulostamisen ja vastaavasti valmistumattomalle tarkastukselle voidaan tulostaa raportteja. Tämänkaltaisia yksityiskohtia aiotaan selkeyttää siten, että ennen tarkastuksen valmistumista on mahdollista tulostaa vain huoltotarkastuslista kullekin tarkastukselle, ja vastaavasti tarkastuksen valmistumisen jälkeen tarjolla on vain kyseiseen kontekstiin sopivia osaraportteja, kuten nosturin kunnon tilaan liittyviä raporteja.

4.2 Kehitystyön kulku ja näkymät

Kehitystyön ensimmäisessä vaiheessa tavoitteena on, että uuden toiminnallisuuden tuottamat raportit ovat englanninkielisiä PDF-asiakirjoja, ja ne tuotetaan vanhassa MMP-järjestelmässä käytettyjen XSLT-tyylitiedostojen avulla, joten tuloksena syntyvät raportit ovat samanmuotoisia ja -näköisiä kuin vanhassa MMP-järjestelmässä. Tämä otettiin toteutuksen lähtökohdaksi, jotta toteutuksessa olisi jokin hyväksi todettu vertailukohta. Uuden toiminnallisuuden tuottamien raporttien oikeellisuutta on helpompi verrata vanhan järjestelmän samalla lähtödatalla tuottamiin raportteihin, ja toisaalta asiakas on pääosin tyytyväinen nykyisin raportteihin, joten lähtökohtaisesti niitä ei tarvitse merkittävästi muokata muotoilullisesti tai sisällöllisesti. Kehitystyössä säästyy näin merkittävästi aikaa, kun uusia tyylitiedostoja tai muita vastaavia asiakirjapohjia ei tarvitse alkaa työstämään nollasta. Aiempi kokemus on osoittanut, että esimerkiksi nyt käytössä olevien XSLT-tyylitiedostojen työstäminen ja viimeistely on ollut varsin pitkä ja hidas projekti. Lisäksi olemassa olevien monimutkaisten tyylitiedostojen laajempi muokkaus on riskialtista, sillä siihen vaadittava syvempi osaaminen puuttuu: tyylitiedostojen alkuperäiset toteuttajat eivät ole enää saatavilla, eikä projektiryhmästä löydy laajempaa kokemusta XSLT-tyylitiedostojen työstämisestä.

Kokonaisuus toteutetaan siten, että myöhemmissä vaiheissa sitä voidaan laajentaa. Raportointiin voidaan lisätä esimerkiksi uutta asiakirjan muotoilua, tyylitystä ja

rakennetta määrittäviä asiakirjapohjia, ja siinä voidaan käyttää myös muille teknologioilla kuin XSL-kielillä toteutettuja tyylitiedostoja tai asiakirjapohjia. Muut, mahdollisesti helppokäyttöisemmät ja kevyemmin ylläpidettävät teknologiat voivat olla varteenotettava vaihtoehto, varsinkin, jos niillä voidaan saavuttaa asiakasta tyydyttävä lopputulos.

Eräs raportteja tuottavaan kokonaisuuteen lisättävistä merkittävistä mahdollisista ominaisuuksista on HTML-muotoisten raporttien luontimahdollisuus. Totetutettavassa uudessa järjestelmässä toinen asiakasohjelmista on internet-selaimella käytettävä sovellus, joten voisi olla luontevaa, että sovelluksen avulla luodut raportit olisivat natiivisti selaimella luettavissa. Lisäksi HTML-muotoiset raportit mahdollistavat paljon joustavamman tavan esittää tietoa, kun raportti voidaan muotoilla ja järjestää rakenteellisesti mielikuvituksellisillakin tavoilla.

Eräs mahdollisuus hyödyntää raportointikokonaisuutta tulevaisuudessa on myös sen jonkinlainen integraatio mobiiliasiakassovellukseen. Koska raportteja tuottava toiminnallisuus on itsenäinen kokonaisuus, se voidaan sisällyttää joustavasti erilaisiin sovellutuksiin. Asiakas on esittänyt toiveen muun muassa siitä, että mobiiliasiakassovelluksessa olisi hyvä pystyä selaamaan huoltotarkastuksen loppuraportin jonkinlaista vedosta. Nyt TypeScriptillä toteutettu itsenäinen komponentti voidaan lisätä mobiiliasiakassovellukseen kirjastona, jolloin sen toiminnallisuus on käytettävissä mobiilisovelluksessa myös silloin, kun mobiililaitteella ei ole yhteyttä internetiin eikä se pysty pyytämään palvelimelta dataa tai valmista raporttia. Mobiilisovellus voi käyttää laitteen muistiin tallennettua dataa, joka on täysin samanmuotoista kuin palvelimella tai selainsovelluksessa käsiteltävä JSON-data, ja tulostaa kirjaston avulla raportteja. Laitteen muistiin pitää vain tallentaa tarvittavat tyylitiedostot ja asiakirjapohjat sekä muut resurssit. Tämä voidaan hoitaa sovelluksen asennuksen mukana.

4.3 Raportointipalvelu osana järjestelmän arkkitehtuuria

Asiakkaan vaatimukset, uudet teknologiavalinnat sekä uuden järjestelmän erityispiirteet vaikuttavat raportin luomisprosessiin. Projektin tavoitteena oli toteuttaa helposti laajennettava ja ylläpidettävä modernein teknologioin toteutettu järjestelmä.

Nykyhetkellä suosittu ja nouseva trendi toteuttaa laajennettavia ja helposti ylläpidettäviä järjestelmiä on mikropalveluiden käyttäminen keskeisenä osana järjestelmän arkkitehtuuria. Mikropalvelut ovat löysästi paritettuja ja tarkkaan määriteltyjä palveluita, jotka keskustelevat toistensa kanssa keveiden protokollien välityksellä.

Mikropalveluiden perimmäisenä ajatuksena on "tehdä yhtä asiaa, ja tehdä se hyvin". [34]

Mikropalvelut ohjaavat luontaisesti järjestelmän modulaariseen rakenteeseen, mikä helpottaa ohjelman jakamista itsenäisesti ja eristetysti kehitettäviin kokonaisuuksiin. Eristetyt ja itsenäiset komponentit helpottavat vikojen eristämistä, sillä esimerkiksi muistivuodot vaikuttavat vain mikropalveluun eikä koko järjestelmään. Mikropalvelut vähentävät riippuvuutta teknologiavalinnoista, sillä kukin palvelu voidaan tarvittaessa toteuttaa siihen parhaiten soveltuvilla teknologioilla, ja toisaalta olemassa olevia palveluita voidaan toteuttaa uudelleen päivitetyillä teknologiavalinnoilla, eikä muihin järjestelmän osiin tarvitse tehdä muutoksia. [35]

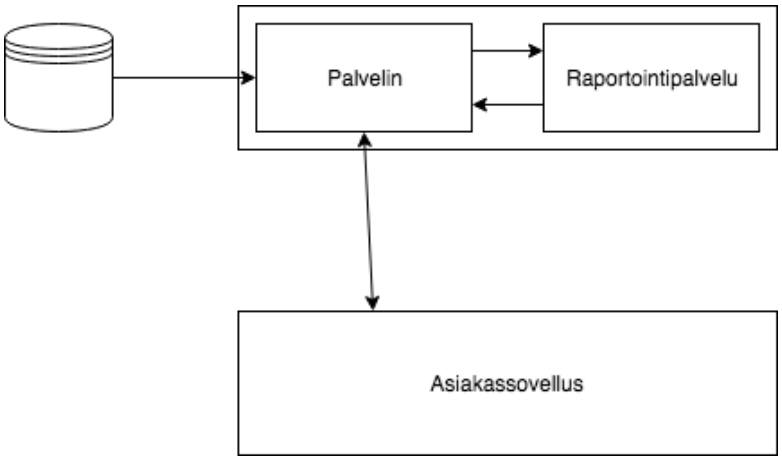
Mikropalveluiden haittapuolia voivat olla järjestelmän kompleksisuuden tarpeeton kasvaminen sekä resurssien lisääntynyt kulutus. Esimerkiksi N monoliittisen sovelluksen korvaaminen N kertaa M mikropalvelulla, joita ajetaan esimerkiksi virtuaalikoneilla tai muilla palvelininstansseilla, johtaa väistämättä siihen, että järjestelmän ajamiseen käytettyjen resurssien hukka (engl. overhead) kasvavat. Esimerkiksi yksi monoliittinen Java-sovellus toimii yhdellä Java-virtuaalikoneinstanssilla, jolla on oma hukkansa. M mikropalvelusta koostuva sovellus vaatii puolestaan M kappaletta Java-virtuaalikoneinstanssia, jolloin hukan määrä lisääntyy, vaikkei välttämättä lineaarisesti. [35]

Toisaalta mikropalveluiden verkon yli tapahtuva kommunikaatio on hitaampaa kuin monoliittisen ohjelman sisällä tapahtuva muistin lukeminen ja kirjoittaminen. Etäkutsut voivat myös epäonnistua, jolloin virheherkkyys kasvaa.[36]

Toteutettava järjestelmä on toistaiseksi varsin yksinkertainen, jolloin voidaan olettaa, että mikropalveluarkkitehtuurin suurimmat sudenkuopat, eli kasvanut kompleksisuus, lisääntynyt resurssien käyttö ja palveluiden välisen kommunikoinnin hallinnan vaikeus, on toistaiseksi hallittavissa. Toisaalta uuden järjestelmän kokonaiskäyttäjämääräksi odotetaan noin 350 käyttäjää. Tämä viittaa siihen, että uusi järjestelmä ei ole suorituskykyherkkä, eikä massiivisia optimointitoimia suorituskyvyn takaamiseksi ole vielä tässä vaiheessa tarpeellista toteuttaa.

Toteutettava raportointikokonaisuus voitaisiin toteuttaa myös uuden järjestelmän palvelinohjelmistoon perinteiseen tapaan monoliittisesti. Tämä lähestymistapa pitäisi järjestelmän kokonaisarkkitehtuurin yksinkertaisempana ja todennäköisesti myös suorituskyky voisi olla hieman nopeampi, koska hitaita etäkutsuja ei tarvitse tehdä. Kuitenkin mikropalveluajattelun tuomat edut, kuten modulaarisuus sekä helppo ylläpidettävyys ja uudelleentoteutus uusin teknologioin linjautuvat projektin tavotteisiin ja tahtotiloihin erinomaisesti.

Palvelun ydin toteutetaan itsenäisenä komponenttinaan, mikä mahdollistaa sen sijoittamisen sekä palvelimelle että asiakassovellukseen. Komponentti on TypeScript-kirjasto, mikä sallii puolestaan laajan kirjon erilaisia sovelluskohteita. Tässä työssä pohditaan toteutettavan palvelun sisällyttämistä palvelimelle ja asiakassovellukseen. Kuvassa 4.1 on kuvattu, miten toteutettava palvelu sijoittuu suhteessa palvelimeen ja asiakasohjelmistoon.



Kuva 4.1 *Palvelu osana palvelinta*

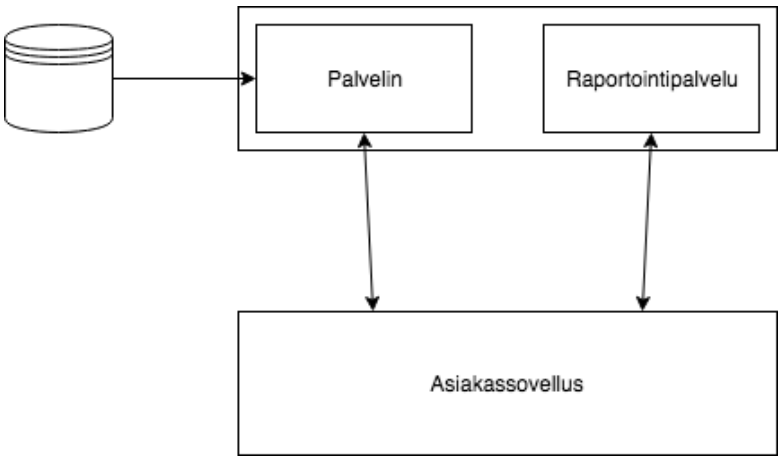
Raportointikomponentin ensisijainen toimintaympäristö on palvelimella, joten on päätettävä, miten komponentti laitetaan ajoon siten, että se toimii järjestelmän palvelinsovelluksen kanssa samalla laitteistolla, ja että palvelinsovellus voi kommunikoida raportointikomponentin kanssa. JavaScriptillä tehdyn komponentin voi laittaa palvelimella muutamalla tapaa ajoon.

Rhino on Mozillan kehittämä Javalla toteutettu avoimen lähdekoodin JavaScript-toteutus, jota tyypillisesti käytetään JavaScript-koodin ajamiseen Java-ympäristössä. Tällöin Rhino mahdollistaa kehittäjiensä mukaan skriptauksen. [37]

Rhinon dokumentaatio ja referenssimateriaali on kuitenkin vähäistä, eikä Rhino muutenkaan vaikuta soveltuvan täysimittaisen JavaScript-kirjaston ajamiseen palvelinympäristössä.

Rhinoa mielenkiintoisempi vaihtoehto on Node.js. Node.js on avoimen lähdekoodin JavaScriptillä toteutettu ajoympäristö (engl. runtime environment), joka mahdollistaa JavaScript- ja käännetyn TypeScript-koodin suorittamisen palvelimella alustasta riippumatta. Node.js:ää voidaan laajentaa esimerkiksi Express-kirjastolla, jolloin reititys ja ohjelmointirajapinta palvelukomponentin ja ulkomaailman välillä voidaan määrittää. Express on Node.js:lle tehty ohjelmistokehys asiakkaan pyyntöjen ja sovelluksen päätepisteiden reitittämiseen. [38]

Kun palvelu on osana palvelinta, ei ole yksiselitteistä, miten pyynnöt järjestelmän eri osien välillä kuuluu toteuttaa. Olisi mahdollista, että asiakasohjelmisto kutsuisi erikseen MMP-palvelinta, kun se haluaa esimerkiksi huoltotarkastuksiin liittyvää dataa ja erikseen raporttipalvelua, kun se haluaa vastaanottaa loppuraportteja. Tällainen tilanne on kuvattu kuvassa 4.2.



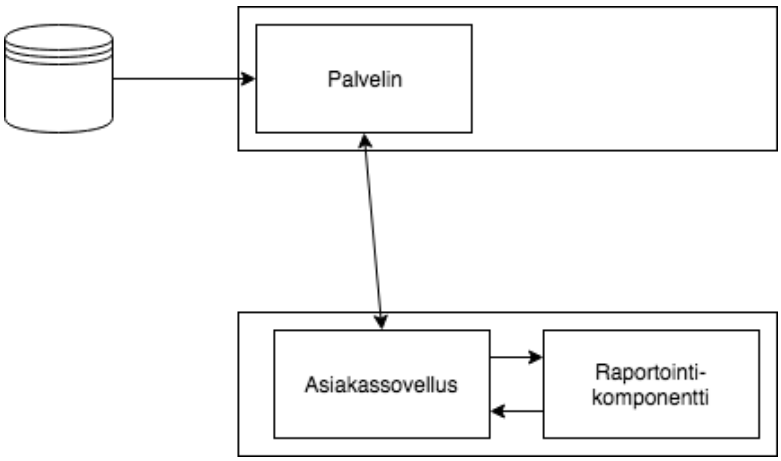
Kuva 4.2 *Ei-toivottu kommunikaatio asiakassovelluksen ja palvelimen välillä*

Kuvan 4.2 kaltainen tilanne mahdollistaisi sen, että asiakassovellus lähettää raportointipalvelulle datan ja palvelun käyttöön tarvittavat muut tiedot. Data voisi olla samaa dataa, joka on asiakassovelluksen saatavilla ja jota asiakassovellus muutenkin käsittelee. Kuitenkin kuvattu järjestely monimutkaistaisi asiakassovelluksen toimintaa ja rakennetta tarpeettomasti, etenkin, jos ja kun datalle täytyy tehdä jotain muunto-operaatioita palvelun käyttämistä varten. Olemassa olevat XSLT-tyylitiedostot vaativat tietyn muotoista dataa, joka poikkeaa jonkin verran asiakassovelluksen ydintoiminnassa käytetystä datasta. Lisäksi tarpeetonta riippuvuutta on hyvä välttää (engl. decoupling).

Suunnitelmassa päädyttiin kuvassa 4.1 kuvattuun malliin, jossa asiakassovellus kutsuu palvelinta, joka puolestaan kutsuu raporttipalvelua. Palvelin valmistelee palvelun tarvitseman datan, jonka se syöttää palveluun. Palvelu koostaa syötetyn datan perusteella loppuraportin, jonka se palauttaa palvelimelle. Palvelin voi halutessaan tallentaa tai muutoin työstää luotua raporttia ja lopulta lähettää sen asiakassovellukselle käytettäväksi.

Koska toteutettava palvelu on TypeScript-kirjasto, joka kääntyy JavaScriptiksi, sitä voidaan käyttää missä vain JavaScript- tai TypeScript-projektissa. Toteutettavassa järjestelmässä asiakassovellukset ovat React- ja React Native -sovelluksia, jotka puolestaan ovat pohjimmiltaan JavaScript-projekteja. Siksi totetutettava palvelu voidaan ottaa käyttöön myös asiakassovelluksen osana käyttämällä palvelua kirjas-

tona. Kuva 4.3 kuvaa, miten palvelu sijoittuu suhteessa palvelimeen ja asiakasohjelmistoon.



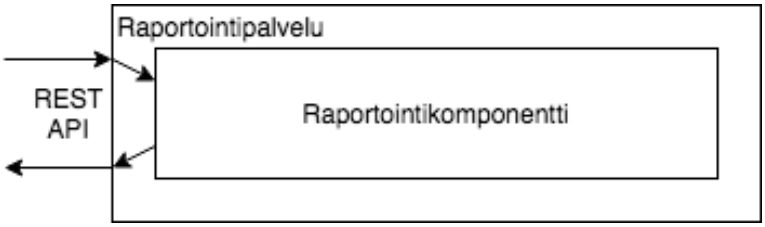
Kuva 4.3 *Palvelu osana asiakassovellusta*

Tässä käyttötapauksessa asiakassovellus on pyytänyt raporteissa käytettävän datan palvelimelta ja tallentanut sen laitteen muistiin. Käytännössä tämä tapahtuu automaattisesti aina, kun käyttäjä kirjautuu sovellukseen sisään. Raportit koostetaan huoltotarkastukset ja niiden tulokset määrittelevästä datasta. Palvelun toiminnallisuus on samanlainen kuin palvelimen osana käytettäessä. Asiakas lähettää datan palvelulle, joka puolestaan palauttaa valmiin loppuraportin. Erona on se, että tässä tapauksessa asiakas käyttää palvelun rajapintaa suoraan, kun taas osana palvelinta palvelua käytetään Node.js-palvelimen REST-ohjelmointirajapinnan kautta.

4.4 Palvelun yleisarkkitehtuuri

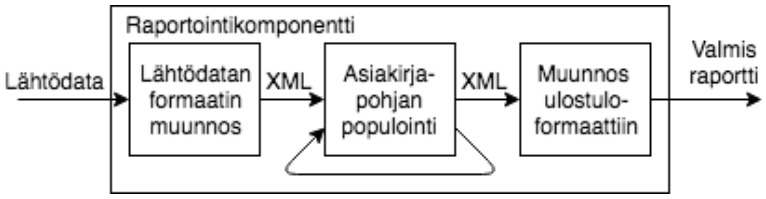
Raportointikomponentin kääriminen Node.js-ajoympäristöön mahdollistaa komponentin käyttämisen palvelinlaitteistolla rinnan järjestelmän Java-palvelinsovelluksen kanssa. Lisäämällä Node.js-ajoympäristöön laajalti käytetty Express-ohjelmointikehys, voidaan komponentin ja ulkomaailman välille määritellä REST-ohjelmointirajapinta, jonka välityksellä komponentille voidaan välittää dataa ja muita parametreja, ja komponentilta voidaan palauttaa valmiita PDF-asiakirjoja. Kuvassa 4.4 on kuvattu korkealla tasolla Node.js- ja Express-kääreen (engl. wrapper) ja raportointikomponentin suhtautuminen toisiinsa.

Raportointikomponentti, joka on käärityt Node-ajoympäristöön ja Express-ohjelmointikehykseen muodostaa siis käytännössä eräänlaisen mikropalvelun. Laajemmalla tasolla uuden MMP-järjestelmän palvelin toimii eräänlaisena orkestraattorina, joka koordinoi järjestelmän eri osien välistä kommunikointia. Puhdas mikropalvelupohjainen järjestelmä ei tässä kuitenkaan ole kyseessä.



Kuva 4.4 *Raportointipalvelu*

Raportointipalvelun sisällä toimiva raportointikomponentti muodostaa putken, jossa on kolme askelta. Kukin kolmesta askeleesta muuntaa ja käsittelee komponenttikirjaston syötettyä dataa ja lähettää sen seuraavalle askeleelle. Kuvassa 4.5 on kuvattu korkealla tasolla komponentin kolme askelta ja kunkin askeleen päävastuualue.



Kuva 4.5 *Raportointikirjasto*

Ensimmäinen askel vastaa datan muuntamisesta muotoon, joka on yhdenmukainen komponentin putken sisällä. Komponentin sisäiseksi datan esitysmuodoksi harkittiin joko JSONia tai XML:ää. JSON ja XML ovat kumpikin yleisesti käytettyjä datan esitysmuotoja datan siirtämisessä palvelimen ja asiakasohjelmistojen välillä. JSON olisi hyvä vaihtoehto, sillä raportteja luova komponentti toteutetaan JavaScriptin laajennoksella TypeScriptillä, jolloin JSON-oliot ovat luettavissa suoraan JavaScript-funktioissa [39]. XML täytyy jäsentää erikseen. JSON on myös tiiviimpi ja nopeammin luettavissa. [39] Lisäksi komponentin vastaanottama lähtödata on suurella todennäköisyydellä JSON-muotoista, sillä uusi MMP-järjestelmä käyttää jo JSONia datan esitysmuotona datan siirtämisessä palvelimen ja asiakassovellusten välillä.

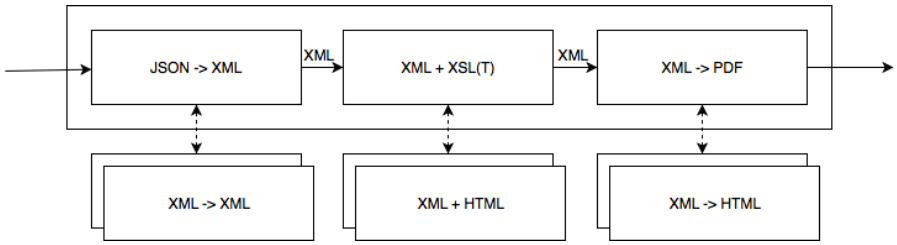
Komponentin sisäiseksi datan esitysmuodoksi valittiin kuitenkin XML, sillä alustavan tutkimuksen perusteella XML-formaattiin perustuvia valmiita ratkaisuja ja kirjastoja on runsaasti; XML:ää käytetään vanhassa MMP-järjestelmässä XSLT-tyylitiedostojen populointiin, jolloin olemassa olevasta tiedosta saadaan synergiaetua uutta toiminnallisuutta toteutettaessa; XML tukee attribuutteja, skeemoja ja nimiavaruuksia [40]. XML voi validoida itsensä XML-skeemojen avulla, manipuloida itseään XSL-tyylitiedostojen avulla ja siitä voi hakea XPathin avulla tietoa. [40]

Toinen askel ottaa XML-muotoisen datan ja muodostaa siitä asiakirjapohjien ja tyylitiedostojen avulla uusia datakokonaisuuksia, jotka kuvaavat tuotettavan loppura-

portin osia. Datasta poimitaan sopivat tiedot ja niillä populoidaan tiedoille osoitetut kentät asiakirjapohjassa. Toinen askel kokoaa kaikki loppuraportin osat yhdeksi yhtenäiseksi kokonaisuudeksi ja lähettää sen kolmannelle askeleelle.

Kolmas askel ottaa loppuraporttia kuvaavan XML-muotoisen kokonaisuuden ja muuntaa sen haluttuun loppuformaattiin. Tämä voi esimerkiksi olla PDF- tai HTML-dokumentti.

Toteutuksen ensimmäisessä vaiheessa komponentti rakennetaan siten, että se tukee JSON- tai XML-muotoisen datan syöttämistä. XML-muotoinen lähtödata syötetään ensimmäisen askeleen läpi sellaisenaan, mutta JSON-muotoinen data muunnetaan ensimmäisessä askeleessa XML-muotoon. Toisessa askeleessa XML-muotoisesta datasta muodostetaan olemassa olevien tyylitiedostojen avulla uusi XML-kokonaisuus. Tämä XML-kokonaisuus sisältää yhdessä paketissa kaikki loppuraportin sisältämät osat. Lopulta kolmannessa vaiheessa XML-muotoinen data muunnetaan PDF-raportiksi. Kuvassa 4.6 on kuvattu datan muunnosvaiheet sekä datan komponentin sisäinen esitysmuoto.



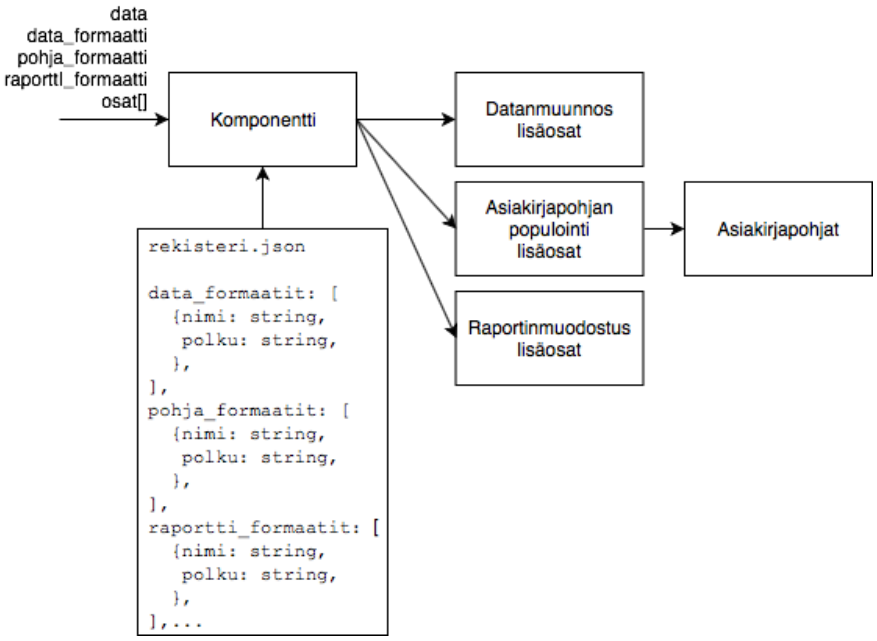
Kuva 4.6 *Komponentin laajennettavuus*

Kuvassa 4.6 on kuvattu myös, miten komponenttia voidaan jatkossa laajentaa tulevaisuuden tarpeisiin. Esimerkiksi komponentin hyväksymiä lähtödatan esitysmuotoja voidaan lisätä toteuttamalla uusia ensimmäistä askelta vastaavia lisäosia. Ensimmäinen vaihe ottaa sisäänsä haluttua lähtödataa ja tuottaa XML-muotoista dataa. Toiseen vaiheeseen voidaan toteuttaa tuki erilaisille asiakirjapohjille. Asiakirjoja voidaan määrittää esimerkiksi HTML:n avulla. Tästä nähtiin esimerkkejä muun muassa kohdassa 3.5. HTML-pohjiin voidaan suhteellisen vaivatta osoittaa kullekin tiedolle oma paikkansa esimerkiksi JavaScriptille tehtyjen Mustache- tai Handlebars -asiakirjapohjamoottoreiden avulla [41][42]. HTML-asiakirjojen tyyliä voidaan puolestaan muotoilla CSS-tyylitiedostojen avulla verkkosivujen tapaan.

Eri vaiheiden yhteensopivuudessa on tärkeää, että vaiheet kommunikoivat yhteisellä kielellä ja että yksittäisen vaiheen rajapinta on kaikissa versioissa sama. Tästä syystä ensimmäisen vaiheen rajapinnaksi sovitaan alustavasti funktiokutsu `convertData(data)`, jossa parametri data on muunnettava data. Nyt siis kaikki

ensimmäisessä vaiheessa käytettävissä olevat laajennusosat on oltava kutsuttavissa edellä olevalla funktiokutsulla. Toinen vaihe kutsutaan `populateTemplate(source, data)`, jossa parametri source on populoitava asiakirjapohja ja parametri data sisältää datan, joka injektoidaan asiakirjapohjaan. Kolmannessa vaiheessa rajapinta on `createReport(source, callback)`, jossa parametri source sisältää PDF-asiakirjan luomiseen tarvittavan datan ja callback-funktio palauttaa valmiin PDF-asiakirjan JavaScript-bufferissa.

Raportointikomponentti on siis laajennettavissa lisäosilla. Kuvassa 4.7 on kuvattu suunnitelma siitä, miten lisäosien rekisteröinti ja yhteistoiminta komponentin ydintoiminnallisuuden kanssa voidaan järjestää.



Kuva 4.7 *Lisäosien rekisteröinti komponentille*

Komponentille syötetään parametreja, joiden perusteella valitaan kullekin askeleelle sopiva lisäosa. Ydintoiminnallisuus tietää sille osoitetun rekisterin perusteella, mitkä lisäosat ovat sen käytettävissä, joten se osaa varautua myös mahdollisiin laittomiin kutsuihin. Ydintoiminnallisuus lataa kullekin askeleelle valitun lisäosan dynaamisesti ohjelmakutsun perusteella.

Asiakirjapohjien populoinnista vastaava toinen osa vaatii populointilogiikan lisäksi myös tiedon itse asiakirjapohjista ja niihin liittyvistä tyylitiedostoista. Nämä tyylitiedostot kulkevat käsi kädessä logiikan kanssa, ja ne rekisteröidään lisäosaan samaan tapaan kuin lisäosat ydintoiminnallisuuteen. Nämä asiakirjapohjat siis kuvaavat kukin yhtä osaa loppuraportista. Mikäli kutsussa yritetään pyytää sellaista osaa, jolle ei ole rekisteröity asiakirjapohjaa, osaa komponentti hylätä kutsun jo kutsua

tehtäessä, eikä vasta ajon aikana.

Raporttienluontipalvelun ja ulkomaailman välinen REST-ohjelmointirajapinta on hyvin yksinkertainen, sillä se koostuu vain yhdestä yhteyspisteestä. Palvelulle lähetetään POST-pyynnön URL-parametreina komponentille välitettävät parametrit ja POST-pyynnön body-osassa itse komponentille syötettävä data.

Raportointipalvelu ottaa parametreina vastaan suoraan komponentin vaatimat tiedot, toisin sanoen määrittelyn siitä, mikä on sisään tulevan datan formaatti, mikä on asiakirjapohjien formaatti, mitä asiakirjapohjia käytetään, ja missä formaatissa raportti lopulta tulostetaan. Käyttäjä valitsee rajapintakutsussa, mitä komponenttiin rekisteröityjä suotimia käytetään missäkin raportointikomponentin putken kolmesta vaiheesta.

5. TOTEUTUKSEN YKSITYISKOHDAT

Komponentin ja palvelun toteutus tehtiin alhaalta ylöspäin-menetelmällä (engl. bottom-up), eli ensin keskityttiin itse raportointikomponentin toiminnallisuuteen, jonka jälkeen komponentti käärittiin itsenäiseksi palveluksi. Komponentin pohjana käytettiin Alex Joverin Typescript-library-starter-aloituspaketilla [43], joka sisältää hyödyllisiä kirjastoja valmiiksi konfiguroituina TypeScriptillä toteutettavien kirjastojen kehittämistyötä silmällä pitäen. Koodin muodon yhdenmukaisuudesta ja oikeellisuudesta huolehtivat Prettier ja TSLint [44][45]. Prettier pakottaa noudattamaan hyvää tyyliä koodattaessa. TSLint tarkkailee tyyppien oikeellisuutta, koodin ylläpidettävyyttä ja mahdollisia toiminnallisia virheitä.

Toiminnallisuuden testaaminen tapahtuu käyttämällä Jest-yksikkötestikirjastoa [46]. Jokaiselle toiminnallisuudelle kirjoitettiin testit, joka olikin käytännössä ainoa mielekäs tapa testata ohjelmaa. Komponentti on itsenäinen kokonaisuus, eikä sitä siksi voi testata kokeilevalla käytännön testauksella ohjelman osana käyttöliittymän kautta, kuten usein ohjelmistokehityksessä voi. Typescript-library-starterissa Jest on lisäksi konfiguroitu siten, että se pitää jatkuvasti kirjaa testien kattavuudesta, ja siten kannustaa mahdollisimman korkeaan testikattavuuteen.

5.1 XSL-teknologian vaihto HTML:ään ja JSON:iin

Kehitystyö suunniteltiin tapahtuvan komponentin sisäisen putken järjestyksessä, lähtien datan muuntamisesta palvelinpäässä sopivaan muotoon ja JSON-muotoisen datan muuntamisesta XML-muotoon ensimmäisessä vaiheessa, siirtyen XSL-asiakirjapohjien populointiin XML-muotoisella datalla toisessa vaiheessa, syntyneen XML-dokumentin muuntamiseen PDF-asiakirjaksi kolmannessa vaiheessa ja päättyen PDF-raportin palauttamiseen palvelimen kautta asiakassovellukselle.

Komponenttia kehittäessä havaittiin, että olemassa olevat XSL-pohjaisia asiakirjapohjia ei voida hyödyntää komponentin kaikkien kolmen vaiheen läpi. Palvelinpäässä oikeaan muotoon muokattu JSON-data saatiin komponentin ensimmäisessä vaiheessa muunnettua sopivaksi XML-dataksi. Tähän käytettiin XML-JS -kirjastoa [47].

XML-JS -kirjasto käyttää joko niin sanottua kompaktia tai ei-kompaktia JSON-dataa XML-muunnoksen toteuttamiseksi. Merkittävin ero on se, että ei-kompakti JSON-data sisältää tarkempaa dataa muun muassa tulokseksi halutun XML- datan attribuuteista ja esimerkiksi siitä, miten taulukoiden muunnos tapahtuu. [47]

Näistä ei-kompakti JSON toimi paremmin tämän työn käyttötapauksessa, sillä käytetty data on hyvin monitasoista. Dataan liittyvät yksityiskohdat, kuten tieto erilaisista vanhempi-lapsi-suhteista on siirrettävä ja säilytettävä mahdollisimman tarkasti muunnosprosessissa. Muokkaamalla palvelimen tuottama JSON ei-kompaktiksi lisäämällä siihen vaadittuja kenttiä ja muuntamalla rakennetta sopivaksi saatiin JSON muunnettua vaaditunlaiseksi XML-dataksi.

Toisessa vaiheessa XSL-tyylitiedostot saatiin populoitua onnistuneesti XML-datalla. Tähän käyttötarkoitukseen löytyi kolme ratkaisua. Mozillan Developer Network tarjosi ratkaisuksi selaimissa sisäänrakennettuna olevan XSLTProcessorin. Tätä ei kuitenkaan saatu toimimaan. Toisena ratkaisuna kokeiltiin Libxslt, joka kuitenkaan ei kääntynyt kehitysympäristössä, joten se päätettiin hylätä. Käytetty ratkaisu oli XSLT-processor-kirjasto, joka on XSL-tyylitiedostojen prosessointiin tarkoitettu kirjasto. Kyseinen kirjasto ei kuitenkaan sisällä TypeScript-tyypityksiä, mistä johtuen kehitystyössä käytetty TSLint huomauttaa tästä varoituksella, joista lopulta päätettiin olla välittämättä.

Prosessin kolmannessa vaiheessa, XML-dokumentin muuntamisessa PDF-dokumentiksi, törmättiin suureen ongelmaan. Käytettävissä olleet XSL-tyylitiedostot määrittelevät populoinnin lopputulokseksi XSL-FO-muotoisen dokumentin. XSL-FO-dokumenttien manipulointiin ei kuitenkaan onnistuttu löytämään ainuttakaan JavaScriptille tai TypeScriptille sopivaa vaihtoehtoa.

Ainoa löydetty vaihtoehto oli Apachen FOP (Formatted Objects Processor), joka on Java-kehitykseen soveltuva XSL-manipuloitiin tarkoitettu kirjasto [48]. Ratkaisua, jossa raportointikomponentti laitettaisiin kutsumaan Java-toteutusta, joka manipuloisi XSL-FO-dataa harkittiin. Se kuitenkin lisäisi liiaksi järjestelmän monimutkaisuutta, ja pilaisi raportointikomponentin alkuperäisen ajatuksen yksinkertaisesta, itsenäisestä ja helposti ympäristöstä toiseen siirrettävästä komponentista.

Toinen ratkaisuvaihtoehto oli, että XSLT-tyylitiedostot ja tuloksena syntyvät XSL-FO-tiedostot korvattaisiin jollain muulla teknologialla. Tarkasti määriteltyjen tyylitiedostojen korvaaminen ja kirjoittaminen alusta tyhjästä vaikutti myös riskialttiilta, sillä luotavien uusien asiakirjapohjien luomiseen ja sen manipulointiin käytettävän infrastruktuurin tekemiseen vaadittavaa aikaa ei pystytty täysin varmasti arvioimaan. Tämä puolestaan lisäsi riskiä siihen, että asiakkaan kanssa jo sovitusta työ-

määräarviosta joudutaan poikkeamaan - toisin sanoen työmääräarviota lisäämään, mikä ei projektiorganisaation puolesta ollut toivottava lopputulos.

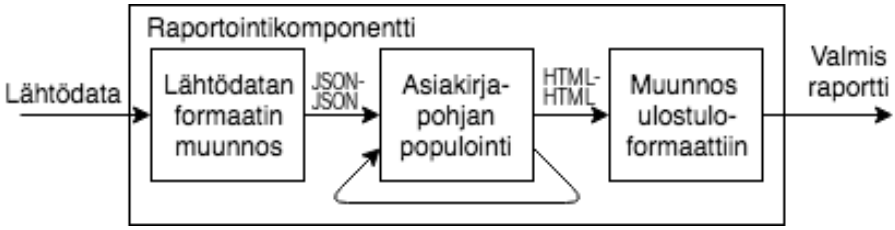
Todettiin kuitenkin, että ainoa järkevä vaihtoehto on kokeilla uusien asiakirjapohjien tekemistä. Kuten kohdassa 3.5 nähtiin, useat kirjastot käyttävät HTML-pohjaisia asiakirjapohjia PDF-asiakirjojen luontiin, joten uusien asiakirjapohjien kuvauskieleksi valittiin HTML. Jotta HTML-asiakirjapohjat voidaan populoida halutulla datalla, täytyy ottaa käyttöön jokin asiakirjapohjamoottori (engl. template engine), joiden avulla voidaan datan kullekin kentälle osoittaa niille tarkoitettu paikka.

JavaScriptille toteutetuista asiakirjapohjamoottoreista harkittiin Mustachea [41], Handlebarsia [42] sekä EJS:ää [49]. Kaikki kolme ovat suhteellisen suosittuja asiakirjapohjamoottoreita, ja niitä tarjotaan esimerkiksi Express-verkkosovelluskehyksen mukana [50]. Näistä moottoreista kehittäjällä oli ennestään kokemusta. Pikaisen vertailun jälkeen moottoriksi valikoitui Handlebars. Handlebarsin dokumentaatio oli kattavin ja lisäksi mahdollisuus kirjoittaa apurifunktioita (engl. helpers) koettiin hyödylliseksi. Handlebarsin ja Mustachen syntaksi miellyttivät enemmän kuin EJS. Handlebars ja Mustache ovat toistensa läheisiä sukulaisia, mutta Handlebars laajentaa Mustachea lisäämällä siihen toiminnallisuutta, joka helpottaa asiakirjapohjien kirjoittamista [51].

HTML-pohjaisten asiakirjapohjien valintaa tukee myös se, että asiakkaan ajatuksissa on ollut myös interaktiiviset HTML-raportit. HTML-kuvauskielellä tehtyjä asiakirjapohjia voidaan hyödyntää helposti myös HTML-muotoisten loppuraporttien muodostamisessa.

Asiakirjapohjien kuvauskielen vaihtuminen XML:stä HTML:ään johti myös siihen, että komponentin sisäinen datan esitysmuoto vaiheiden välillä oli järkevää vaihtaa. JavaScript tukee oletuksena JSON-esitysmuotoa, ja Handlebars toimii tällöin suoraan ilman ylimääräisiä välivaiheita. Ei ole järkevää muuntaa esimerkiksi JSONia XML:ksi, jos heti seuraavassa vaiheessa XML olisi muunnettava JSON:ksi.

Komponentin yhtenäisestä sisäisestä esitysmuodosta luovuttiin, ja päädyttiin ratkaisuun, jossa vaiheiden sisään- ja ulostulojen datan esitysmuotojen on sovittava yhteen. Tästä pidetään huoli komponentin rekisterin avulla. Käyttäjä määrittelee kutsussaan, mitä vaiheita hän haluaa käyttää. Kunkin vaiheen sisään- ja ulostulevan datan esitysmuodot tallennetaan, ja niitä vertaillaan ajonaikaisesti. Mikäli esitysmuodot eivät täsmää, tai esitysmuodoille ei ole toteutusta, käyttäjää varoitetaan ja prosessi keskeytetään. Kuvassa 5.1 on kuvattu esimerkki vaiheiden välisestä kommunikaatiosta.



Kuva 5.1 *Komponentin uusi sisäinen järjestys*

Nyt siis ensimmäinen vaihe tuottaa JSON-muotoista dataa, ja toisen vaiheen on otettava sisäänsä JSON-muotoista dataa. Vastaavasti toinen vaihe tuottaa HTML-asiakirjan, ja kolmannen vaiheen on otettava sisäänsä HTML-dataa. Huonona puolena on se, että toisin kuin alkuperäisessä ajatuksessa, komponentit eivät enää ole täysin vaihdettavissa keskenään, vaan joudutaan kiinnittämään ylimääräistä huomiota komponenttien välisiin suhteisiin. Saavutetut edut, eli vähentynyt monimutkaisuus, kun ylimääräisiä muunnosvälivaiheita saadaan karsittua, ja toisaalta JavaScriptin suoraan tukemat datan esitysmuodot, kuten JSON, helpottavat ja nopeuttavat kehitystyötä sekä yksinkertaistavat ohjelmaa.

Vaikka joka vaiheen välisestä yhteisestä kielestä luovuttiin, on edelleen tärkeää, että vaiheen kaikki laajennusosat toteuttavat yhtenäisen rajapinnan, jotta ne ovat edelleen vaihdettavissa keskenään.

5.2 Uudet asiakirjapohjat ja muunnos PDF-tiedostoksi

Tässä työssä mielenkiintoisimmat ja hyödyllisimmät Handlebarsin ominaisuudet ovat tagit, toistorakenteet ja apurifunktiot. HTML-asiakirjapohjiin sijoitettavat tagit kertovat, mikä JSON-datan kenttä sijoitetaan mihinkin kohtaan tuloksena syntyvää HTML-dokumenttia. Esimerkiksi tagin `{{reportedBy}}` tilalle sijoitetaan JSON-datassa esiintyvän `reportedBy`-kentän sisältö. JSON-datan taulukoita voidaan iteroida käyttämällä `{{#taulukko}}...{{/taulukko}}`-rakennetta, jolloin esimerkiksi `machineries`-taulukon kaikki nimikentät voidaan iteroida käyttäen `{{#machineries}} <span>{{machineryName}}</span> {{/machineries}}`-rakennetta.[42]

Lisäksi Handlebars tukee apurifunktioita, jolloin kehittäjä voi luoda omia yksilöllisiä tagejaan ja lisätä niihin tarvitsemaansa logiikkaa. Näin esimerkiksi voidaan tarvittaessa vertailla tagiin syötettyä arvoa ja palauttaa tagin määrittelemässä rakenteessa haluttuja HTML-elementtejä if-else-rakennetta muistuttavasti. [42] Tämä mahdollistaa varsin monimutkaistenkin asiakirjapohjien toteuttamisen.

HTML-asiakirjapohjien muuntamiseen PDF-asiakirjoiksi päätettiin käyttää kohdas-

sa 3.5 esiintynyttä Marc Bachmannin Node-html-pdf-kirjastoa, jota on käytetty monissa vastaavanlaisissa kirjastoissa ja projekteissa. Node-html-pdf tukee juoksevaa sivunumerointia sekä sivukohtaisia ylä- ja alatunnisteita. Nämä koettiin erityisen hyödyllisiksi, koska tuotettavat loppuraportit koostuvat useista osaraporteista. Ylä- ja alatunnisteen tarkka määrittely vaikutti houkuttelevalta.

5.2.1 Uusien asiakirjapohjien haasteet

Uusien HTML-muotoisten asiakirjapohjien luonnissa sekä yhden yhtenäisen PDF-tiedoston muodostamisessa eri osaraporteista oli muutamia haasteita. Ensimmäinen kohdatuista ongelmista oli, miten osaraportit käytännössä saadaan koostettua yhdeksi kokonaisuudeksi. Useamman valmiin PDF-asiakirjan yhdistäminen JavaScript-ympäristössä todettiin joko vaikeaksi tai mahdottomaksi. Node-html-pdf ottaa sisäänsä yhden HTML-merkkijonon ja muodostaa siitä PDF:n. Ongelman ratkaisemiseksi toteutettiin logiikka, jossa palvelulle syötetyissä parametreissä määritellään lista haluttavista osaraporteista, jonka perusteella luettiin HTML-asiakirjapohjatiedostoja merkkijonoina muuttujiin. Muuttujien sisältämä data myöhemmin yhdistettiin +-operaatiolla toisiinsa yhden pitkän HTML-merkkijonon muodostamiseksi. Tuloksena saatu yksittäinen merkkijono syötetään Node-html-pdf-kirjastolle.

Käytettävät osaraportit sisältävät taulukoita, jotka voivat olla enemmän kuin yhden sivun mittaisia. Näiden taulukoiden otsikkorivien täytyy toistua jokaisen sivun yläreunassa. Mikäli osaraportit tulostettaisiin yksittäisinä PDF-tiedostoina, kuten vanhassa MMP-järjestelmässä, voitaisiin HTML-PDF-muunnoksesta vastaavalle kirjastolle todeta, että jokaisen sivun yläreunassa ylätunnisteessa toistuu taulukon otsikkorivi. Tämä ei ole kuitenkaan mahdollista, sillä yhteen PDF-tiedostoon oli tulostettava useita erimuotoisia osaraportteja, joissa taulukoiden otsikkorivit vaihtelevat.

Ongelma ratkaistiin käyttäen Handlebarsin apurifunktioita. Osaraportin taulukkoa populoitaessa lasketaan, montako elementtiä on tulostettu, ja kun tietty raja-arvo ylittyy, päätetään taulukko, tehdään sivunvaihto ja aloitetaan uusi taulukko, jonka data jatkuu siitä, mihin edellisellä sivulla päädyttiin. Sivunvaihto voidaan pakottaa lisäämällä HTML-asiakirjaan `<div>`, jonka tyylimäärittelyissä on määritelty `page-break-after: always`. Kunkin osaraportin ensimmäinen sivu on erikoistapaus, sillä ensimmäiselle sivulle mahtuva taulukko on pienempi johtuen otsikkokentistä ensimmäisen sivun yläreunassa. Tämä otettiin huomioon apurifunktiota toteutettaessa.

Eräs mielenkiintoinen yksityiskohta osaraportissa, jossa kuvataan nosturin kuntoa

(engl. Crane condition), on värilliset palkit, joiden pituus vaihtelee sen mukaan, kuinka vakava jonkin komponentin korjaustarve on, ja väri puolestaan kuvaa huollon tarpeen kiireellisyyttä. Oli keksittävä, miten tämä palkki voidaan muodostaa HTML-kuvauskieltä ja CSS-tyylimäärittelyjä käyttäen. Lähtödatassa on kaksi kenttää, kiireellisyys (engl. priority) ja kunto (engl. condition). Kiireellisyysarvo vastaa siis jotain väriä, ja sillä voi olla arvoja nollasta kolmeen. Kuntoarvo kertoo, kuinka pitkä palkki on, ja se voi saada arvoja yhdestä yhdeksään.

Koska Handlebars on ensisijaisesti logiikaton (engl. logic-less) asiakirjapohjamoottori, oli kehitettävä keino, miten kahta arvoa voidaan vertailla keskenään, jotta voitaisiin määrittää, muodostetaanko komponentin korjaustarpeen kiireellisyyttä kuvaava vihreä, keltainen vai punainen palkki. Toteutettiin jälleen apurifunktio, joka vertailee kahta Handlebars-tagille annettua arvoa, eli lähtödatasta saatavaa kiireellisyysarvoa numeroon yksi. Mikäli arvot ovat yhtäsuuret, palautetaan Handlebars-tagin aloitus- ja lopetustagin välinen elementti, esimerkin tapauksessa punainen texttt -elementti. Näin vertailemalla kiireellisyysarvoa arvoihin 0...3 voitiin määrittää muodostettavan elementin väri.

Muodostettavan elementin pituus määritettiin siten, että käyttäen Handlebarsin datan injektointitagia syötettiin muodostettavan elementin tyylimääritysattribuuttiin elementin pituutta määrittelevään kenttään CSS:n calc-funktion sisään kuntoarvo, joka kerrottiin funktion sisällä arvolla 5 millimetriä. Näin muodostettavalle elementille saatiin haluttu pituus viiden millimetrin portaissa.

5.2.2 Pysty- ja vaakasuuntaisten raporttien käsittely

Osa loppuraporttiin sisällytettävistä osaraporteista on pystysuuntaisia, kun taas valtaosa on vaakasuuntaisia. Esimerkiksi kansilehti on orientaatioltaan pystysuuntainen, kun taas nosturin kuntoa kuvaava raportti vaakasuuntainen taulukko.

Oli selvitettävä, miten samassa asiakirjassa voidaan esittää sekä pysty- että vaakasuuntaisia osaraportteja siten, että lopputulos on sekä tulostettaessa että tietokoneen näytöllä järkevä. Tunnistettiin kolme vaihtoehtoa:

1. Lopullisessa PDF-asiakirjassa on sekä pysty- että vaakasuuntaisia sivuja
2. Lopullisessa PDF-asiakirjassa on joko pysty- tai vaakasuuntaisia sivuja
3. Tulostetaan erillisiä PDF-tiedostoja kustakin osaraportista, jolloin ongelmaa ei tarvitse käsitellä.

Kolmas vaihtoehto oli välittömästi poissuljettu, sillä asiakas haluaa, että kaikki osaraportit muodostavat yhden yhtenäisen kokonaisuuden. Tutkittiin, pystytäänkö Node-html-pdf -kirjastolla muodostamaan PDF-tiedostoja, jossa osa sivuista on pysty- ja osa vaakasuuntaisia, mutta mitään tähän viittaavaa mahdollisuutta ei löytynyt dokumentaatiosta eikä kokeiluista. Jäi siis jäljelle mahdollisuus pohtia, miten pysty- ja vaakasuuntaiset osaraportit voidaan määritellä siten, että ne muodostavat järkevän lopputuloksen.

Todettiin, että käyttäen CSS-tyylimäärittelyn kierto-ominaisuutta voidaan pystysuuntaiset HTML-asiakirjapohjat kääntää vaakasuuntaisiksi. Alunperin pystysuuntaiset osaraportit olivat tietosisällöltään onneksi kaikkein yksinkertaisimpia, jolloin oli järkevämpää kiertää ne vaakasuuntaisiksi kuin vaakasuuntaiset pystysuuntaisiksi.

Näin kaikki sivut voidaan muodostaa oletuksena vaakasuuntaisiksi ja paperille tulostamisen jälkeen lopputulos on järkevä: kansilehti on pystysuutainen ja taulukoita voi lukea kääntämällä paperin vaakaan.

5.3 Komponentista itsenäinen palvelu

Toteutettu raportointikomponentti käännettiin lopulliseksi tuotantokokonaisuudeksi, joka oli tarkoitus laittaa Node-ajoympäristöön ajoon raportointipalveluna. Käännöksen ja uuden Node-Express-sovelluksen luonnin yhteydessä kuitenkin havaitiin, että jostain tuntemattomasta syystä Node-ajoympäristö nosti virheen ja esti ajamisen, huomauttaen, että dynaamiset moduulien esittelyt (engl. import) eivät ole tuettu ominaisuus käytetyssä kääntäjäpaketoijassa. Mitään dynaamisia esittelyitä ei oltu kuitenkaan käytetty, joten jatkokehityksen estänyt virhe päätettiin kiertää lykkäämällä ajatus itsenäisestä kirjastosta ja lisäämällä kirjaston lähdekoodi suoraan palvelun lähdekoodin osaksi.

Palvelun ja itse komponentin ytimenä toimivan Node-html-pdf-kirjaston rajapintafunktiot ovat asynkronisia, joten havaittiin, että lienee tarpeen muuttaa kaikki PDF-asiakirjan luontiin liittyvää funktiota ympäröivät funktiot myös asynkronisiksi. Tämä toteutettiin lupauksia (engl. promise) hyödyntämällä siten, että Node-html-pdf-kirjaston PDF-asiakirjan luovan funktion vastakutsufunktio (engl. callback function) palauttaa virhetilanteissa hylkäävän signaalin (engl. reject) ja onnistuneissa tapauksissa hyväksyntäsignaalin (engl. resolve). Tätä ketjua jatkettiin aina palvelun REST-rajapinnan määrittelevään funktioon, jolloin virhetilanteet saadaan johdettua aina rajapintaan asti ja sen kautta käyttäjille.

Java-palvelinohjelmiston rajapinta määriteltiin siten, että asiakassovellukselle

avoin POST-rajapinta lähettää pyynnöstä kyseisenmuotoista dataa POST-pyynnöllä raportointipalvelulle, josta vastauksena saatu tavutaulukkona (engl. byte array) välitettävä valmis PDF-asiakirja välitetään suoraan vastauksena alkuperäiseen POST-pyyntöön. POST-kutsu koostuu perusosoitteesta, `inspections`-määritteestä, kyseisen nosturihuoltotulostietueen tunnistenumerosta (engl. id, identifier), `report`-määritteestä sekä joukosta kyselyparametreja. Kyselyparametreissa voidaan määritellä raporttiin valikoidut osaraportit. Esimerkki lopullisesta POST-pyynnön URL-osoitteesta voisi olla `http://localhost:9999/inspections/1234/report?parts=cover&parts=key &parts=condition`. Tällä pyynnöllä tulostuisi loppuraportti, joka koostuisi kansilehdestä (cover), selitesivusta (key) sekä nosturin kuntoraportista (condition).

Vastaavasti käyttöliittymään, eli selainsovellukseen, luotiin näkymä, jossa raportoijan nimi ja alaotsikot voidaan määritellä sekä valita, mitkä osaraportit tulostetaan. Lisäksi käyttöliittymään lisättiin uutena ominaisuutena loppukatselumuksen tietojen täyttäminen. Asiakassovelluksen palvelimelle lähettämän POST-pyynnön body-osassa lähetetään loppukatselmuksen sisältö, josta se välitetään edelleen raportointipalvelulle. Lisäksi bodyn mukana lähetetään erilaisia alaotsikoita ja muita raportissa näkyviä, käyttäjän muokattavissa olevia kenttiä.

Tässä vaiheessa otettiin huomioon myös tavoitteet siitä, että käyttäjälle on tarjolla vain kontekstiin sopivia raporttityyppejä, toisin sanoen esimerkiksi ennen tarkastuskäynnin valmistumista käyttöliittymässä ei nyt voi tulostaa raportteja, jotka riippuvat tarkastuskäynnin tuloksista.

Ajoympäristön yhteensopivuusongelmat

Siirrettäessä Node-ajoympäristö ajoon tuotantotestausympäristöön havaitiin, että tuloksena syntyvissä PDF-asiakirjoissa sisällön mittasuhteet ovat virheellisiä. Ilmeni, että PhantomJS - jota Node-html-pdf-kirjasto käyttää keskeisenä riippuvuutena PDF-asiakirjojen luomisessa HTML-dokumenteista - käyttäytyy eri tavoin Windows- ja UNIX-ympäristöissä. PhantomJS käyttää eri pikselitiheysarvoa (engl. dots per inch, DPI) Windowsissa, jossa pääasiallinen kehitystyö tapahtuu, ja UNIX-pohjaisissa järjestelmissä, joihin tuotanto- ja tuotantotestauspalvelimilla käytetty käyttöjärjestelmä kuuluu. Windowsissa käytetään DPI-arvoa 72, kun taas UNIX-järjestelmissä arvoa 96.

Selvisi, että ongelma on havaittu ensimmäisen kerran jo vuonna 2014, eikä PhantomJS- tai Node-html-pdf-kirjastoihin ole saatu tähän päivään mennessä korjausta toteutettua [52]. Lisäksi, kuten kohdassa 3.5 todettiin, PhantomJS:n kehitys-

työ on keskeytetty, joten selvää korjausta tähän lienee turha odottaa.

Tunnistettiin muutamia ratkaisuvaihtoehtoja. Joko

1. määritellään kaikki luodut HTML-asiakirjapohjat uusiksi siten, että ne toimivat UNIX-ympäristössä ajettaessa
2. kääritään HTML-merkkijono, jota käytetään PDF-asiakirjojen luonnissa, yhteisen HTML-tagin alle ja määritellään CSS-tyylimäärittelyssä sen zoom-arvoksi 0.75 (72/96)
3. vaihdetaan Node-html-pdf-kirjasto ja siinä käytettävä PhantomJS pois ja otetaan tilalle Puppeteer, tai
4. määritellään Node-html-pdf-kirjaston asetuksissa, että sivun koko on (96/72)-kertaa suurempi, kuin standardi A4-arkki

Ensimmäinen vaihtoehto tulkittiin liian raskaaksi ja aikaa vieväksi. Lisäksi todetiin, että se vaikeuttaisi liikaa sekä kehitystyötä että ylläpidettävyyttä, sillä ensisijainen kehitysympäristö on Windows.

Toinen vaihtoehto sai onnistuneesti skaalattua sisältöä alaspäin siten, että kaikki sisältö mahtui omille tarkoituksenmukaisille sivuilleen ja paikoilleen. Zoom-operaatio kuitenkin sekoitti tuloksena saatua HTML-asiakirjaa siten, että joissain kohdissa kirjainten väliset etäisyydet olivat joko liian pieniä tai suuria. Vaikka lopputulos oli luettava, johti ratkaisu levottomaan ja rumaan ulkoasuun.

Kokeiltiin myös vaihtaa PhantomJS-riippuvuus pois ja ottaa tilalle Puppeteer. Muunnostyö eteni lupaavasti, ja kaikki tarvittavat ominaisuudet saatiin toimimaan myös Puppeteeria käyttäen. Ongelmat alkoivat jälleen, kun palvelu piti laittaa tuotantotestausympäristöön ajoon. Selvisi, että Puppeteer vaatii tiettyjä riippuvuuksia, jotka on asennettava suoraan palvelimelle. Kuitenkaan juuri käytössä olevalle käyttöjärjestelmälle ei löytynyt kaikkia tarvittavia riippuvuuksia, eikä Puppeteeria kehittävä Googlen Chromium-tiimi ole jatkanut käytössä olevan käyttöjärjestelmäversion tukemista [53]. Ratkaisuvaihtoehtoina olisi joko käyttöjärjestelmän päivittäminen tuoreempaan, raportointipalvelun laittaminen ajoon Docker-säiliönä tai palvelimen käyttöjärjestelmän vaihtaminen toiseen, esimerkiksi Ubuntuun [53].

Docker on ohjelmisto, joka suorittaa virtualisointia käyttöjärjestelmän tasolla siten, että samaa Docker-pakettia, niin kutsuttua säiliötä (engl. container) voidaan ajaa

lähes missä ympäristössä tahansa, ja kaikki säiliön sisällä olevan ohjelman riippuvuudet kulkevat säiliön sisällä. Tämä helpottaa ohjelman kehitystyötä ja saattamista tuotantoon, sillä kaikki ohjelman tarvitsemat komponentit, työkalut ja asetukset kulkevat säiliössä, ja ne ajetaan eristetysti käyttöjärjestelmätason virtuaalipalvelimilla. [54]

Mahdollisuutta ajaa raportointipalvelua Docker-säiliön sisällä tutkittiin tarkemmin. Selvisi kuitenkin, että käytössä oleva käyttöjärjestelmä ei ole tuettu, ja Dockerin käyttöönotto olisi vaatinut joko käyttöjärjestelmän tai käyttöjärjestelmän ytimen (engl. kernel) vaihtamista tai päivittämistä versioon, jota palvelinympäristöstä vastaava tukiorganisaatio ei puolestaan tue. Siten Dockerin käyttöönotto olisi vaatinut käytännössä yhtä merkittäviä muutoksia palvelinympäristön infrastruktuuriin kuin Puppeteer-kirjaston käyttöönotto.

Päätettiin kokeilla vielä eräässä toisessa projektissa käytettyä ratkaisua, eli muuttaa PhantomJS-kirjaston ajoasetuksissa sivun mittoja A4-arkista (96/72)-kertaisiksi. Effektiivisesti tämän pitäisi johtaa samankaltaiseen lopputulokseen kuin CSS zoom-toiminnallisuuden käyttäminen vaihtoehdossa numero 2. Lopputuloksena saatiin oikein skaalautunut ja aseteltu PDF-asiakirja, jossa toisesta vaihtoehdosta tuttuja kirjaintenvälivirheitä ei näkynyt.

Näin keskeinen raportointitoiminnallisuus saatiin toimimaan myös tuotantotestausympäristössä. Jäljellä oli vielä huoltotarkastuslomakkeen toiminnallisuuden toteuttaminen. Raportointipalvelun päässä tämä vaati vain uuden HTML-asiakirjapohjan luomisen sekä uuden huoltotarkastuslomakkeelle räätälöidyn rajapinnan toteuttamisen palveluun. Huoltotarkastuslomakkeeseeen ei tarvitse muun muassa valita eri osaraportteja tai liittää loppukatselmusosaa. Toiminnallisuuden laajentaminen oli tässä vaiheessa varsin suoraviivainen projekti, ja haasteellisin osuus oli toteuttaa Java-palvelimelle rajapinta, joka muokkaa datan huoltotarkastuslomakkeen haluamaan muotoon.

6. TOTEUTUKSEN ARVIOINTI

Työssä toteutetun raportointikomponentin ja -palvelun sopivuuden ja toimivuuden arviointi tapahtuu luontevimmin peilaamalla toteutuksen ominaisuuksia ja toiminnallisuutta projektin ja raportointikomponentin tavotteisiin sekä asiakas vaatimuksiin. Lisäksi toimivuutta ja sopivuutta voidaan arvioida erilaisten käyttökokemustapausten avulla. Tässä merkittävässä roolissa ovat asiakkaan edustajan kokemukset ja kommentit.

Kerrataan ensin työlle asetetut tärkeimmät tavoitteet ja vaatimukset.

- Käyttäjä voi valita, mitkä raportit hän sisällyttää valmiiseen PDF-asiakirjaan.
- Käyttäjä voi muokata lopulliseen PDF-asiakirjaan tulevia tiettyjä kenttiä, kuten alaotsikot.
- Tuloksena on yksi yhtenäinen raportti, joka sisältää yhden tai useamman aliraportin.
- Raporttien ulkoasu ja sisältö täytyy olla pääosin nykyisten raporttien kaltainen.
- Raporttien ulkoasun päivittäminen voi olla tulevaisuudessa tarpeen.
- Varataan mahdollisuus nostaa PDF-raportoinnin rinnalle HTML-raportit, joita voi lukea selaimessa.
- Komponentin pitää sisältää tai siihen pitää olla muutoin saatavilla TypeScript-tyypitykset.
- Komponentin pitää toimia ensisijaisesti palvelimella.
- Komponentissa pitää olla valmius toimia myös asiakassovelluksen osana.

Raportointikokonaisuuden toiminnallisuutta rajoittivat eniten projektin tavoitteet

ja teknologiavalinnat. Asiakassovelluksissa käytetyt React ja React Native sekä tietoinen päätös käyttää TypeScriptiä kehitystyössä JavaScriptin sijaan antoivat selkeät suuntaviivat toteutuksen aloittamiselle. Myös komponentti oli toteutettava TypeScriptillä, jotta TypeScript-projektien vaatimat tyypitykset luotaisiin automaattisesti ja toisaalta komponentti olisi suoraan yhteensopiva TypeScriptillä toteutettujen asiakassovellusten kanssa tarpeen vaatiessa.

Toteuttaminen aloitettiin JavaScript-kielessä käytettyyn NPM-pakettikirjastoon soveltuvan kirjaston luomisella, mutta myöhemmät esteet (kirjaston käyttökelvottomuus paikallisesti) kehitystyössä hidastivat niin paljon, että täydellisestä NPM-kirjastosta luovuttiin. Sen sijaan kirjastossa käytetty lähdekoodi vain otettiin omaksi moduulikseen uuteen raportointipalveluun. Tämä toki vähentää ratkaisun eleganssia ja tekee kirjastosta ja itse palvelusta asteen enemmän riippuvaisia toisistaan. Kun raportointikomponentin lähdekoodi on kiinteänä osana palvelua, on sen jatkokehitys vähemmän hienostunutta, sillä komponenttia ei nyt voi kehittää yhtä eristetysti ja itsenäisesti, täysin erillisenä kokonaisuutena, mikä alunperin oli ideana.

Koska komponentin toteuttaminen täysin itsenäisenä NPM-kirjastona toistaiseksi epäonnistui, ei komponenttia tai sen lähdekoodia voi käyttää sellaisenaan esimerkiksi puhtaissa JavaScript-projekteissa. Mikäli toteutus olisi onnistunut, komponentti kääntyisi JavaScriptiksi, jolloin sitä voisi käyttää sekä JavaScript- että TypeScript-projekteissa. Nyt komponentin lähdekoodille pitää löytää jokin vaihtoehtoinen keino kääntää se JavaScriptiksi. Komponentin kehittäminen loppuun itsenäisenä NPM-kirjastona on eräs hyvä jatkokehityskohde.

Alunperin raportointikokonaisuuden kantavana ideana oli käyttää vanhassa MMP-järjestelmässä käytettyjä XSLT-tyylitiedostoja. Arkkitehtuurin ja yleisten suuntaviivojen suunnittelussa painotettiin ja otettin huomioon myös se, että asiakirjapohjien ja asiakirjapohjissa käytettyjen teknologioiden vaihtamisen täytyy olla mahdollista. Kuten huomattiin, olemassa olevien XSLT-tyylitiedostojen käyttö osottautui mahdottomaksi JavaScript- tai TypeScript-maailmassa, sillä tuloksena syntyneiden XSL-FO-tiedostojen manipulointiin ei löydetty mitään ratkaisua.

Varhaisessa vaiheessa huomioon otettu teknologioiden vaihtamisen mahdollisuus oli eduksi, kun kehitystyössä havaittiin, että XSLT-pohjaisten teknologioiden hyödyntäminen kokonaisuudessaan on mahdotonta. Komponentin arkkitehtuuriin ei tarvinnut koskea, jolloin pystyttiin keskittymään uusien asiakirjapohjien luontiin. Asiakirjapohjissa käytettyjen teknologioiden vaihdon myötä kokonaisuuden ylläpidettävyys saatiin paremmalle tasolle, sillä uudet asiakirjapohjat ovat selkeämpiä ja helpommin luettavia.

Käyttöliittymän ja käyttökokemuksen puolesta pyrittiin jäljittelemään mahdollisimman pitkälle vanhan järjestelmän toiminnallisuutta. Käyttöliittymään kopioitiin vanhan järjestelmän syötekentät, jolloin oppimiskynnyksen pitäisi olla erittäin matala. Käyttöliittymää ja käyttökokemusta järkeistettiin muun muassa yksinkertaistamalla työvirtaa siten, että esimerkiksi useimmin käytetyt raporttityypit ovat valmiiksi valittuna, ja toisaalta käyttäjä ei voi tulostaa raportteja, joiden vaatimaa tietoa ei ole vielä saatavilla.

Asiakkaan ehdottomana vaatimuksena oli, että järjestelmästä tulostuu vain yksi PDF-asiakirja, joka koostuu yhdestä tai useasta osaraportista. Tässä onnistuttiin, ja käyttäjä voikin muiden valintojen ohessa valita käyttöliittymässä haluamansa osat, ja ne tulostuvat loppuraporttiin. Tästä asiakasvaatimuksesta seurasi kuitenkin tarve muodostaa kompromisseja sekä asiakirjapohjia luotaessa että tulostuslogiikkaa miettiessä.

Kaiken kaikkiaan uusien asiakirjapohjien luonnissa onnistuttiin hyvin, sillä osaraporttien ja huoltotarkastuslomakkeiden sivujen sisältö ja asettelu saatiin vastaamaan vanhassa järjestelmässä tuotettuja raportteja. Asiakirjapohjien toteutuksessa jouduttiin kuitenkin tekemään muutamia kompromisseja, lähinnä sen takia, että yhteen PDF-asiakirjaan haluttiin sisällyttää useampi osaraportti, ja PDF-asiakirjojen sisältö voi niiden osalta vaihdella.

Ensinnäkin uudessa järjestelmässä raportit tulostetaan aina siten, että sivut ovat vaakasuuntaisia, vaikka osassa osaraportteja sivun asettelu on pystysuuntainen. Tämä aiheuttaa sen, että tietokoneen ruudulla raporttia on käännettävä aina sen mukaisesti, onko ruudulla pysty- vai vaaka-aseteltu sivu. Tämä haitta on lähinnä kosmeettinen, ja esimerkiksi sen jälkeen, kun raportti on tulostettu paperille, raporttia pystyy lukemaan täysin luontevasti.

Toinen kompromissi liittyy siihen, että nosturin kuntoraportissa ja turvallisuusyhteenvedossa näytetään kullekin nosturin komponentille tehdylle työlle kunto- ja kiireellisyysarvion lisäksi kuvauskenttä. Kuvauskenttä on vapaamuotoinen tekstikenttä, johon huoltotarkastuksen tekijä voi kirjata lisätietoja tehdystä työstä. Tämä kuvaus voi ääritapauksissa olla hyvin pitkä, jolloin vaarana on, että se levittyy sivulla useammalle riville. Tämä saattaa johtaa siihen, että raportin sivutuksessa käytetty logiikka ei toimi, ja sivujen sivutus ja asettelu menee rikki, mikäli yhdestä rivistä tuleekin pitkän kuvauskentän takia useamman rivin mittainen. Tämä ratkaistiin siten, että yksi rivi näissä osaraporteissa on aina yhden rivin mittainen, ja kuvauskentissä ylimenevä osa katkaistaan, ja siitä merkiksi tilalle tulostetaan pistekolmikko (...). Tämä ei ole paras ratkaisu, koska vaarana on, että kuvauskentästä leikkautuu pois

mielenkiintoista informaatiota. Tämä saattaa olla kuitenkin ainoa ratkaisu, mutta sen selvittäminen loppuun asti on toisaalta hyvä jatkokehitysidea.

Raportointikokonaisuuden mikropalvelutyyppinen luonne havaittiin hyödylliseksi muun muassa järjestelmää pystyttäessä tuotantotestauspalvelimelle hyväksymistestausta ja käyttöönottoa varten. Raportointikomponenttia pystyttiin tutkimaan, tarvittavia viimeistelyjä tekemään ja löydettyjä epäkohtia korjaamaan itsenäisesti ja eristetysti muusta järjestelmästä. Tällöin raportointikomponentissa havaitut ongelmat eivät - mikropalveluille tyypilliseen tapaan - heijastuneet muun järjestelmän toimintaan, vaan muu tiimi saattoi jatkaa keskeytyksettä työskentelyä ja testaamista muun järjestelmän parissa samalla, kun raportointikokonaisuutta kehitettiin. Mikäli raportointikomponentti olisi ollut monoliittinen osa järjestelmää, olisi koko tiimin työskentely hidastunut, kun uutta versiota koko järjestelmästä olisi pitänyt odottaa valmiiksi palvelimelle pystyttämistä varten.

Jo toteutetusta järjestelmästä voisi olla mielekästä jatkokehittää ja jatkotutkia edelleen esimerkiksi aiemmin kohdissa 3.4 ja 4.3 esiteltyyn tapaan muun muassa erilaisten käyttöympäristöjen parissa. Erityisesti raportointikomponentin sijoittaminen suoraan mobiilisovellukseen ja raporttien tuottaminen laitteelle tallennetusta datasta ilman internetyhteyttä voisi olla mielenkiintoinen kokeilu. Mobiilisovellusta saatetaan käyttää kohteissa, joissa ei ole riittävää kuuluvuutta ja pääsyä internetiin edes matkapuhelinverkon yli, jolloin palvelimella tapahtuvaa raporttien tulostusta ei voida hyödyntää.

Kirjoittamishetkellä komponentin sijoittaminen mobiilisovellukseen ja toiminnallisuuden testaaminen ei ole mielekästä, sillä kokonaisprojektissa on keskeisempiä kehittämiskohteita, joihin on syytä keskittyä ensin. Näihin lukeutuvat raporttien tulostusmahdollisuus palvelimen kautta mobiilisovelluksessa ja siihen liittyvän käyttöliittymän toteutus. Käytännössä selainsovelluksen toiminnallisuus toistetaan mobiilisovelluksen puolella.

Jotta komponenttia voitaisiin käyttää mobiilisovelluksessa ilman yhteyttä palvelimeen, olisi sovellukseen joko tallennettava jatkuvasti valmiiksi dataa raporttien asiakirjapohjien sallimassa muodossa tai toteutettava suhteellisen monimutkainen logiikka nykyisellään sovelluksen muistiin tallennettavan datan muuntamiseksi asiakirjapohjien hyväksymään muotoon. Ensimmäinen vaihtoehto vaatisi, että sovelluksen muistissa olisi käytännössä sama tieto kahteen kertaan, mutta eri muodossa, mikä ei välttämättä ole toivottavaa, sillä sovellusten tilankäyttö ja paikallisen muistin koko halutaan pitää kohtuullisena. Toinen vaihtoehto puolestaan vaatisi merkittävästi lisää työtä, mikä ei välttämättä projektin kannalta ole mielekästä.

Raportointikomponenttin laajennettavuutta ja laajennettavuuden helpottamista voitaisiin kehittää edelleen viemällä toteutusta entistä hienostuneempaan ja toimivampaan lisäosa-arkkitehtuurin (engl. plugin architecture) suuntaan. Komponentin sisäistä putkea voitaisiin varmasti edelleen jalostaa ja optimoida, etenkin siinä tapauksessa, että komponenttia laajennetaan edelleen tukemaan entistä enemmän erilaisia välivaiheita.

Olisi myös mielenkiintoista kehittää komponenttia ja järjestelmää edelleen siten, että raportointikokonaisuus muodostaisikin HTML-sivuja, jotka toimisivat interaktiivisina raportteina. HTML-raportit voisivat sisältää linkkejä toisiin raportteihin ja raportin osiin. Tämä ajatus tultaneen toteuttamaan projektin myöhemmissä vaiheissa.

7. YHTEENVETO

Tässä työssä toteutettiin ja modernisoitiin nosturihuollon suunnittelu- ja raportointisovellusta. Työssä keskittyttiin erityisesti modernisoitavan sovelluksen erilaisia raportteja tulostavan kokonaisuuden vaatimusten keräämiseen, suunnitteluun, toteutukseen ja toteutuksen arviointiin.

Työssä tunnistettiin ensin sovelluksen toimintaympäristö, sovelluksen käyttöön liittyvä huoltoprosessi, modernisoitavan sovelluksen lähtökohdat ja historia sekä tarve ja vaatimukset uudelle järjestelmälle. Aluksi pohdittiin myös raportteja yleisellä tasolla sekä raporttien merkitystä liiketoiminnassa sekä yleisellä tasolla että asiakkaan tapauksessa. Modernisoitava sovellus oli Konecranes Oyj:n nosturihuolto-organisaation käyttämä MMP-sovellus, jota käytetään nosturihuollon suunnitteluun, tulosten raportointiin ja huollon hallintaan.

Kyseinen monoliittinen työpöytäsovellus kaipasi sekä teknisesti että ominaisuuksien puolesta päivittämistä, jolloin todettiin järkeväksi kehittää uusi sovellus modernein teknologioin. Työpöytäsovelluksen tilalle luotaisiin selainsovellus, joka on käytettävissä laitteistosta riippumatta. Modernisoinnissa paperilomakkeen rooli haluttiin poistaa, ja tilalle kehittää mobiilisovellus, jonka avulla huoltotarkastuksen tekijä voisi kirjata tulokset suoraan järjestelmään sähköisesti.

Tämän jälkeen esiteltiin uuden järjestelmän suunnitteluratkaisut ja teknologiavalinnat sekä uuden järjestelmän toteutusprojektin rajaamat ehdot pohjustamaan tässä työssä kohteena olleen raportointikokonaisuuden suunnittelua. Merkittävimmät suuntaviivat olivat lopputuloksen modernius, helppokäyttöisyys ja laajennettavuus, vanhaa järjestelmää pääosin vastaava toiminnallisuus ja raporttien sisältö sekä lisäksi yhteensopivuus sekä palvelin- että TypeScriptillä toteutettujen asiakassovellusten kanssa.

Teknologiarajotteiden ja asiakasvaatimusten pohjalta suunniteltiin uuteen järjestelmään integroitava mikropalvelutyyppinen raportointipalvelu. Raportointipalvelu toimisi itsenäisenä osana järjestelmää, jolloin se olisi itsenäisesti kehitettävissä ja laajennettavissa tulevaisuudessa. Palveluun otettaisiin yhteyttä palvelinohjelmiston

välityksellä, joka valmistelisi raportteja varten tarvittavan datan asiakasohjelmiston pyynnöstä.

Palvelun ytimessä toimisi TypeScriptillä toteutettu raportointikomponentti, joka muodostaisi putken, joka koostuisi kolmesta käsittelyvälivaiheesta. Ajatuksena oli, että näitä käsittelyvälivaiheita voisi vaihdella dynaamisesti, ja siten voitaisiin tukea erilaisia lähtödatan esitysmuotoja, asiakirjapohjaformaatteja sekä raportin ulostuloformaatteja. Ensimmäisessä vaiheessa komponentin haluttiin tukevan JSON-muotoista lähtödataa ja tulostavan PDF-muotoisia asiakirjoja lopputuloksena.

Ylimääräisen työn määrä haluttiin pitää mahdollisimman pienenä, joten toteutusta lähdettin lähestymään siten, että uudessa järjestelmässä käytettäisiin vanhassa järjestelmässä käytettyjä XML-merkintäkieliperheeseen pohjautuvia asiakirjapohjia. Komponentti suunniteltiin toimimaan siten, että komponentin välivaiheiden välisenä sisäisenä datan esitysmuotona käytetään XML-merkintäkieltä.

Myöhemmin selvisi, että XML-pohjaisten asiakirjojen ja erityisesti XSL-FO-asiakirjojen käsittely on käytännössä mahdotonta JavaScript-pohjaisilla teknologioilla, joten XML/XSL-pohjaiset asiakirjat päätettiin hylätä ja toteuttaa uudet asiakirjapohjat käyttäen HTML-merkintäkieltä ja Handlebars-asiakirjapohjamoottoria. Komponentin sisäisestä vaiheiden välisestä yhtenäisestä kommunikaatiokielestä luovuttiin. Uudistetut asiakirjapohjat täyttivät paremmin vaatimusta moderneista teknologioista. Asiakirjapohjien ylläpito ja jatkokehitys helpottuivat aiempaan verrattuna.

Vaatimus siitä, että yhteen PDF-asiakirjaan piti olla mahdollista valita dynaamisesti eri osaraportteja aiheutti muutamia ratkaistavia ongelmia sekä kompromisseja komponentin ja asiakirjapohjien suunnittelussa ja toteutuksessa. Näistä huolimatta loppujen lopuksi lopputuloksen sisältö ja toiminnallisuus saatiin kiitettävälle tasolle ja ne vastaavat vaatimuksia ja vanhasta järjestelmästä tuttua toiminnallisuutta.

Tuloksena saatiin asiakkaan vaatimukset täyttävä ja jatkossa laajennettavissa oleva, teknisesti modernein ratkaisuin toteutettu raportointikokonaisuus, jonka käyttökokemus joko vastaa vanhaa tai on loogisempi kuin vanhassa MMP-järjestelmässä. Toisaalta käyttöliittymän sisällön suunnittelussa ja toteutuksessa otettiin mallia vanhasta MMP-järjestelmästä, jolloin voidaan olettaa, että vanhaan järjestelmään tottuneilla käyttäjillä ei ole vaikeuksia tunnistaa, miten uutta järjestelmän toiminnallisuutta käytetään.

Jatkokehitysideoina tunnistettiin komponentin jalostaminen hienostuneemmalle tasolle laajennettavuuden ja arkkitehtuurin saralla. Lisäksi komponentin laajentami-

nen tukemaan aidosti useampia erilaisia välivaiheiden formaatteja todistaisi tällä hetkellä vain teoriassa todetun ominaisuuden. Komponentin sijoittaminen myös muihin toimintaympäristöihin, kuten mobiilisovellukseen suoraan, voisi olla mielenkiintoinen kehityskohde.

LÄHTEET

[1] Ebert C., and Duarte C. H. C., "Digital transformation," *IEEE Software*, vol. 35, no. 4, pp. 16–21, July 2018.

[2] Matt C., Hess T., and Benlian A., "Digital transformation strategies," *Business & Information Systems Engineering*, vol. 57, no. 5, pp. 339–343, Oct 2015. Saatavissa: https://doi.org/10.1007/s12599-015-0401-5

[3] RSM US LLP, "The importance of the right reporting, analytics and information delivery," Tech. Rep., 2018.

[4] Konecranes. About Konecranes, verkkosivu. Viitattu: 13.6.2018. Saatavissa: http://www.konecranes.com/about-konecranes

[5] Konecranes. Business Areas, verkkosivu. Viitattu: 13.6.2018. Saatavissa: http://www.konecranes.com/about/business-areas

[6] Konecranes. General Description, verkkosivu. Viitattu: 13.6.2018. Saatavissa: http://www.konecranes.com/about/general-description

[7] TIOBE. TIOBE Index for June 2018. Viitattu: 13.6.2018. Saatavissa: https://www.tiobe.com/tiobe-index//

[8] Juntunen T. Raportti - Oulun seudun ammattikorkeakoulu, tekniikan yksikkö. Viitattu: 11.8.2018. Saatavissa: http://www.tekniikka.oamk.fi/~tuijaj/suomen_kieli/raportti.htm

[9] Webopedia. Report. Viitattu: 11.8.2018. Saatavissa: https://www.webopedia.com/TERM/R/report.html

[10] Business Dictionary. Report. Viitattu: 9.9.2018. Saatavissa: http://www.businessdictionary.com/definition/report.html

[11] Hopeavuori T. Raportoinnin tarkoitus. Viitattu: 9.9.2018. Saatavissa: http://www.oamk.fi/~thopeavu/materiaalit/raportoinnin_tarkoitus.html

[12] W3C. The Extensible Stylesheet Language Family (XSL). Viitattu: 3.9.2018. Saatavissa: https://www.w3.org/Style/XSL/

[13] React Native. React Native - A framework for building native mobile apps using JavaScript and React. Viitattu: 27.7.2018. Saatavissa: https://facebook.github.io/react-native/

[14] Pivotal Software. Spring Boot. Viitattu: 18.10.2018. Saatavissa: https://spring.io/projects/spring-boot

[15] Fielding R. T. Representational State Transfer. Viitattu: 18.10.2018. Saatavissa: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[16] JSON. Introducing JSON. Viitattu: 18.10.2018. Saatavissa: https://www.json.org/

[17] Haapanen J., "MMP Mobile and Cloud - UI Architecture," Wapice Oy.

[18] Redux. Three principles. Viitattu: 14.8.2018. Saatavissa: https://redux.js.org/introduction/threeprinciples

[19] Redux-Saga. Redux-saga. Viitattu: 14.8.2018. Saatavissa: https://redux-saga.js.org

[20] GitHub. Lang.ai / react-pdfs. Viitattu: 20.7.2018. Saatavissa: https://github.com/lang-ai/react-pdfs

[21] Lang.ai. About us. Viitattu: 20.7.2018. Saatavissa: https://lang.ai/company

[22] GitHub. Diego Muracciole / react-pdf. Viitattu: 20.7.2018. Saatavissa: https://github.com/diegomura/react-pdf

[23] Muracciole D. React-pdf. Viitattu: 20.7.2018. Saatavissa: http://react-pdf.diegomura.com

[24] GitHub. Wojciech Maj / react-pdf. Viitattu: 20.7.2018. Saatavissa: https://github.com/wojtekmaj/react-pdf

[25] GitHub. Mozilla Labs / pdf.js. Viitattu: 20.7.2018. Saatavissa: https://github.com/mozilla/pdf.js

[26] GitHub. RMA Consulting / generate-pdf. Viitattu: 20.7.2018. Saatavissa: https://github.com/rma-consulting/generate-pdf

[27] RMA Consulting. About us - We are RMA. Viitattu: 20.7.2018. Saatavissa: http://rma-consulting.com/about-us/

[28] GitHub. Marc Bachmann / node-html-pdf. Viitattu: 23.7.2018. Saatavissa: https://github.com/marcbachmann/node-html-pdf

[29] QT Company. Qt WebKit. Viitattu 18.10.2018. Saatavissa: https://wiki.qt.io/Qt_WebKit

[30] PhantomJS. PhantomJS - Scriptable Headless Browser. Viitattu: 23.7.2018. Saatavissa: http://phantomjs.org/

[31] GitHub. Puppeteer - Headless Chrome Node API. Viitattu 18.10.2018. Saatavissa: https://github.com/GoogleChrome/puppeteer

[32] GitLab. Joseph Hassell / XML-PDF. Viitattu: 23.7.2018. Saatavissa: https://gitlab.com/poster983/XML-PDF/tree/master

[33] PDFKit. Home. Viitattu: 23.7.2018. Saatavissa: http://pdfkit.org/

[34] Järvinen H-M., "Software architectures - architectural styles," Spring 2018.

[35] Richardson C. Pattern: Microservice Architecture. Viitattu: 3.9.2018. Saatavissa: https://microservices.io/patterns/microservices.html

[36] Fowler M. Microservice Trade-Offs. Viitattu: 3.9.2018. Saatavissa: https://martinfowler.com/articles/microservice-trade-offs.html

[37] Mozilla. Mozilla Developer Network - Rhino. Viitattu: 1.8.2018. Saatavissa: https://developer.mozilla.org/fi/docs/Mozilla/Projects/Rhino

[38] Node.js. About Node.js. Viitattu: 1.8.2018. Saatavissa: https://nodejs.org/en/about/

[39] W3 Schools. JSON vs XML. Viitattu: 8.8.2018. Saatavissa: https://www.w3schools.com/js/js_json_xml.asp

[40] Bugayeko Y. Stop Comparing JSON and XML. Viitattu: 8.8.2018. Saatavissa: https://www.yegor256.com/2015/11/16/json-vs-xml.html

[41] Mustache.js. Logic-less mustache templates with JavaScript. Viitattu: 8.8.2018. Saatavissa: https://github.com/janl/mustache.js

[42] Handlebars. About Handlebars.js. Viitattu: 8.8.2018. Saatavissa: https://handlebarsjs.com

[43] GitHub. alexjoverm/typescript-library-starter. Viitattu: 18.8.2018. Saatavissa: https://github.com/alexjoverm/typescript-library-starter

[44] Prettier. Prettier - Opionated code formatter. Viitattu: 18.8.2018. Saatavissa: https://prettier.io

[45] TSLint. TSLint - An extensible linter for the TypeScript language. Viitattu: 18.8.2018. Saatavissa: https://palantir.github.io/tslint/

[46] Jest. Jest - Delightful JavaScript Testing. Viitattu: 18.8.2018. Saatavissa: https://jestjs.io

[47] GitHub. nashwaan/xml-js. Viitattu: 18.8.2018. Saatavissa: https://github.com/nashwaan/xml-js

[48] The Apache Software Foundation. Apache FOP. Viitattu 18.10.2018. Saatavissa: https://xmlgraphics.apache.org/fop/

[49] Bitomi. EJS - Embedded JS. Viitattu 18.10.2018. Saatavissa: http://www.embeddedjs.com

[50] Express. Using template engines with Express. Viitattu 20.10.2018. Saatavissa: https://expressjs.com/en/guide/using-template-engines.html

[51] HandlebarsJS. Differences between HandlebarsJS and Mustache. Viitattu: 21.8.2018. Saatavissa: https://github.com/wycats/handlebars.js#differences-between-handlebarsjs-and-mustache

[52] GitHub. PhantomJS 2: PDF rendering too large, page.zoomFactor doesn't work. Viitattu: 27.8.2018. Saatavissa: https://github.com/ariya/phantomjs/issues/12685

[53] GitHub. Puppeteer: Can't run Puppeteer in Centos7. Viitattu: 27.8.2018. Saatavissa: https://github.com/GoogleChrome/puppeteer/issues/391

[54] Docker. What is a Container - A standardized unit of software. Viitattu: 27.8.2018. Saatavissa: https://www.docker.com/resources/what-container