OAMK
OULUN AMMATTIKORKEAKOULU

Valtteri Impola

METSÄNHOIDOLLISTEN TÖIDEN TOTEUTUS WEB-KÄYTTÖLIITTYMÄSSÄ

METSÄNHOIDOLLISTEN TÖIDEN TOTEUTUS WEB-KÄYTTÖLIITTYMÄSSÄ

Valtteri Impola
Opinnäytetyö
Kevät 2020
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, Ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä: Valtteri Impola
Opinnäytetyön nimi suomeksi: Metsänhoidollisten töiden toteutus web-käyttöliittymässä
Opinnäytetyön nimi englanniksi:
Työn ohjaaja: Juha Valkola
Työn valmistumislukukausi ja -vuosi: Kevät 2020
Sivumäärä: 48

Tässä opinnäytetyössä tavoitteena oli luoda web-pohjainen ohjelma metsäteollisuuden tarpeisiin. Ohjelmalla käyttäjän tulisi pystyä tekemään sopimuksia ja lisäämään näille sopimuksille työohjeita, sekä myyntinimikkeitä. Luodut sopimukset pitää pystyä listaamaan luetteloon ja näille sopimuksille tulee voida lisätä liitetiedostoja. Näiden toimintojen suorittaminen on ollut mahdollista PiiMegan ForestPro-tuotteen avulla, mutta silloin käyttäjä on ollut sidottuna työpisteeseensä, johon ohjelma on asennettu. Opinnäytetyön tavoitteena oli myös mahdollistaa ohjelman käyttö suoraan selaimesta, jolloin käyttäjä pystyy käyttämään ohjelmaa millä tahansa laitteella, jossa on selain.

Työ toteutettiin uudeksi osaksi olemassa olevaan ohjelmistoon. Käytettäviä menetelmiä olivat React, GraphQL, Apollo client, HotChocolate ja Google Maps API. Toteutustapana oli full stack -kehittäminen, mikä tarkoittaa sitä, että kehittäjä on itse vastuussa kaikista ohjelman osa-alueista.

Opinnäytetyön lopputuloksena syntyi ohjelma, jolla pystytään hallitsemaan ja suunnittelemaan metsänhoidollisia töitä aina sopimuksen luomisesta työohjeiden tarkastelemiseen. Ohjelma mahdollistaa myös työskentelyn millä tahansa laitteella, jossa on selain, jolloin ohjelman käyttäjä ei ole sidoksissa mihinkään tiettyyn työpisteeseen, vaan voi työskennellä myös kentällä.

Asiasanat: React, käyttöliittymä, ohjelmointirajapinta

ABSTRACT

Oulu University of Applied Sciences
Degree Programme of Information Technology, Software Development

---

Author: Valtteri Impola
Title of thesis: Implementation of forestry maintenance in a web user interface
Supervisor: Juha Valkola
Term and year when the thesis was submitted: Spring 2020
Pages: 48

The aim for this thesis work is to create web-based program what user can use to make all planning actions regarding forestry maintenance and supervise ongoing forestry maintenance work. These functions can be done by PiiMega Oy's ForestPro program, but then user is tied to the workstation wherever program is installed. This thesis work aim is to allow user to do all those functions from browser and therefore work with any device and wherever user wants if there is internet connection.

Thesis work was implemented as new part into existing program. The methods used where React, GraphQL, Apollo, HotChocolate and Google maps API. The implementation method was full stack development, which means that developer develops both client and server software.

As a result of this thesis work program was created which allows user to make all planning actions regarding supervise forestry maintenance work and plan forestry maintenance work from generating contracts to adding work instructions to that contract. Program allows user to work with any device with browser and internet connection.

Keywords: React, user interface, API, GraphQL

ALKULAUSE

Tämän opinnäytetyön toimeksiantajana oli PiiMega Oy. Työn ohjaavana opettajana toimi Oulun ammattikorkeakoulun tietotekniikan lehtori Lasse Haverinen. Tilaajan puolesta ohjaajana toimi Project Manager Juha Valkola.

Ohjaajien lisäksi haluan kiittää ohjelmistosuunnittelijoita Arttu Heikkalaa ja Heikki Laulajaista, jotka auttoivat ohjelmointi ongelmissa ja antoivat lisää näkökulmia ratkaisujen muodostamiseen.

Oulussa 6.4.2020

Valtteri Impola

SISÄLLYS

TIIVISTELMÄ 3
ABSTRACT 4
ALKULAUSE 5
SISÄLLYS 6
SANASTO 8
1 JOHDANTO 9
2 KÄYTTÖLIITTYMÄ JA KÄYTTÄJÄKOKEMUS 10
2.1 Käyttöliittymä 10
2.2 Käyttäjäkokemus 11
3 KÄSITTEET, MENETELMÄT JA TYÖKALUT 12
3.1 GraphQL 12
3.1.1 Query 12
3.1.2 Mutaatio 13
3.1.3 Tyypitys ja Skeema 14
3.1.4 Query- ja Mutation-tyypit 15
3.1.5 Interface ja Union tyypit 15
3.2 React 15
3.2.1 Elinkaarimetodit 16
3.2.2 React Hooks 19
3.2.3 UseState Hook 20
3.2.4 Effect Hook 21
3.2.5 Contex Hook 21
3.3 Apollo Client 21
3.3.1 Kyselyiden suorittaminen 21
3.3.2 Mutaatioiden suorittaminen 22
3.3.3 Datan tallennus 22
3.4 Google Maps JavaScript API 23
3.5 IIS 23
3.6 Windows Communication Foundation 24
3.6.1 Perusteet 24
3.6.2 Sopimukset 24

3.6.3 Päätepisteet 25
3.7 Hot Chocolate 26
4 TYÖN TOTEUTUS 27
4.1 Työn tausta 27
4.2 Vaatimukset ja määrittely 28
4.3 Tekniset valinnat 29
4.4 Projektin aloitus 29
4.5 Toteutuneet näkymät 30
4.5.1 Sopimuslistaus 30
4.5.2 Perustiedot 31
4.5.3 Lohkot 32
4.6 KarttaKomponentti 36
4.7 Datakerros 37
4.8 Logiikkakerros 38
4.9 Käyttöliittymä 39
4.10 Datan hakeminen 40
4.11 Jatkokehitys 42
5 YHTEENVETO 44
LÄHTEET 46

SANASTO

API	Application Programming Interface, ohjelmointirajapinta
Client	Client on se osa ohjelmaa, jolla on pääsy serverin ylläpitämään palveluun
DOM	Document Object Model on API HTML- ja XML-dokumentteja varten
GraphQL	Datan haku ja manipulointi kieli ohjelmointirajapinnoille
HTML	Hypertext Markup Language, Internet-sivujen luomiseen käytetty kieli
Hook	Funktio, joka mahdollistaa React staten ja elinkaarimetodien ominaisuuksien käyttämisen funktion komponentista
React	React on JavaScript-kirjasto käyttöliittymien rakentamista varten
Skeema	Skeemalla tarkoitetaan tietokannan rakennetta.
State	Funktio, joka mahdollistaa React staten ja elinkaarime todien ominaisuuksien käyttämisen funktiokomponentista
UI	User Interface eli käyttöliittymä on ohjelmiston osa, missä käyttäjä käyttää ohjelmaa
UX	User experience, käyttäjäkokemus
Virtual DOM	Ohjelmointikonsepti, jossa ideaali kuvaus UI:sta on tallennettu muistiin ja synkronoidaan DOMin kanssa

1 JOHDANTO

Opinnäytetyö tehtiin syksyn 2019 ja kevään 2020 aikana PiiMega Oy:lle osaksi Piimegan olemassa olevaa ForestPortal-web-portaalia. Työ tehtiin muiden yritykselle tekemieni harjoitteluiden jälkeen, ja aiempien projektien aikana saamistani kokemuksista oli apua ohjelman toteutuksessa.

PiiMega Oy on oululainen vuonna 1998 perustettu ohjelmistotalo, jonka tuotteita ovat yritysten toiminnanohjausjärjestelmät, saha- ja metsäteollisuuden toiminnanohjausjärjestelmät sekä verkkokaupparatkaisut. Toteutettu ohjelma kuuluu ForestPortal-web-portaaliin, joka on osa metsäteollisuuden toiminnanohjausjärjestelmiä. ForestPortal on verkkopohjainen järjestelmä, jonka avulla voidaan tehdä muun muassa metsäteollisuuden hakkuiden suunnittelu, puunhankinta ja logistiikan hallinnointi. Järjestelmässä voidaan luoda myös asiakirjoja yllä mainittuihin prosesseihin.

Opinnäytetyön aiheena oli luoda ohjelma, jolla käyttäjä pystyisi tekemään kaikki tarvittavat toiminnot metsänhoidollisten töiden suunnitteluun aina tarjouksen luomisesta leimikolle tehtävien töiden suunnitteluun ja niiden edistymisen seurantaan.

Aikaisemmin metsänhoidollisten töiden suunnittelu ja seuranta on hoidettu työpöytäsovelluksella, joka on asennettu johonkin tiettyyn työpisteeseen. Kevyempi web-sovellus mahdollistaa tietojen reaaliaikaisen päivittämisen suoraan kohteena olevalta työmaalta, mikä johtaa reaaliaikaisempaan tilannekuvaan ja siten mahdollistaa nopeamman reagoinnin.

Pohjustuksena työlle sain tarvittavaa tietoa ja kokemusta kehittämällä toista web-sovellusta PiiMegalle 2019 kevään ja kesän aikana. Tämä auttoi ymmärtämään käytössä olevia työkaluja ja menetelmiä.

2 KÄYTTÖLIITTYMÄ JA KÄYTTÄJÄKOKEMUS

Digitaalisten tuotteiden, kuten sovellusten ja laitteiden, tuotannossa on tärkeänä osana käyttöliittymän ja käyttäjäkokemuksen suunnittelu. Niillä pyritään ohjaamaan käyttäjää toimimaan ja käyttämään tuotetta siten, miten tuote on suunniteltu käytettäväksi. Hyvä käyttöliittymäsuunnittelu helpottaa tuotteen käytettävyyttä ja näin ollen parantaa tuotteen käyttäjäkokemusta. (1.)

Käyttöliittymistä ja käyttäjäkokemuksesta on yleisesti käytetty englanninkielisiä termejä user interface, lyhemmin UI, sekä user experience eli UX. Nämä termit sekoitetaan usein keskenään ja niistä puhutaan samana asiana. (1.)

2.1 Käyttöliittymä

Tässä opinnäytetyössä tuotetaan web-ohjelma, joten tässä luvussa keskitytään ohjelmistojen käyttöliittymien tutkimiseen.

Käyttöliittymällä tarkoitetaan sitä osaa ohjelmistosta, jota käyttäjä käyttää ohjelman kanssa kommunikointiin. Käyttöliittymällä voidaan ohjata ohjelman toimintaa, esimerkiksi millä tavalla käyttäjä siirtyy näkymien välillä. (2, s. 6.)

Käyttöliittymä koostuu yleensä näytöstä, jossa näytetään ohjelman graafinen osa, jota käyttäjä hallitsee, sekä laitteista, joilla näitä laitteita hallitaan. Näitä laitteita ovat hiiri ja näppäimistö. Mobiililaitteita käytettäessä käyttäjän omat kädet tai erilaiset piirtotyökalut ovat korvanneet erilaiset fyysiset painikkeet, joilla voidaan käyttää kosketusnäyttöä. (3, s. 11.)

Ohjelman käytettävyyttä voidaan mitata siten, kuinka nopeasti käyttäjä oppii käyttämään tuotetta tehokkaasti ja oikealla tavalla. Ohjelman käytettävyys on tärkeä osa hyvää käyttäjäkokemusta. (3, s. 11.)

2.2 Käyttäjäkokemus

Käyttäjäkokemus voidaan ajatella eräänlaisena sateenvarjoterminä, jonka alle myös käyttöliittymä ja siihen vaikuttavat tekijät voidaan sijoittaa. Sillä kuvataan käyttäjän tai käyttäjäryhmän kokonaisvaltaista tunnetta, jonka tuotteen käyttäminen tuottaa. Käyttäjäkokemuksen suunnittelu on tärkeä osa sovellusten kehitysprosessia, jotta käyttäjät käyttäisivät tuotetta mielellään ja sen käyttäminen olisi mahdollisimman miellyttävää. (2, s. 12.)

Hyvän käyttäjäkokemuksen toteuttamiseksi kehittäjien täytyy toteuttaa esitutkimusta tuotteen suunnitellusta kohderyhmästä ja ottaa kehityksessä huomioon todellisten käyttäjien antama palaute. Näillä tutkimuksilla saadaan tietoa esimerkiksi käyttöliittymän helppokäyttöisyydestä. Käyttöliittymän helppokäyttöisyyttä voidaan tutkia esimerkiksi pyytämällä käyttäjää suorittamaan jokin tietty toiminto ja samaan aikaan kertomaan ääneen ajatuksiaan ja tuntemuksiaan ohjelman käytöstä. Käytettävyystutkimuksessa on viime aikoina otettu käyttöön katseenseurantateknologia, jolla voidaan seurata käyttäjän silmän liikkeitä käyttöliittymän käytön aikana. Sillä saadaan selville, mihin elementteihin käyttäjä kohdistaa katseensa ja mitkä elementit jäävät huomiotta. Tutkimuksen tulosten perusteella kehittäjät saavat arvokasta tietoa tuotteen käytettävyydestä sekä tietoa siitä, millaisia muutoksia ohjelmaan tulisi tehdä käyttöliittymän käyttäjäkokemuksen parantamiseksi. (2, s. 12.)

3 KÄSITTEET, MENETELMÄT JA TYÖKALUT

Tässä luvussa tutustutaan web-pohjaisen ohjelmiston kehittämiseen liittyviin käsitteisiin sekä käytettyihin menetelmiin ja työkaluihin.

3.1 GraphQL

GraphQL on avoimen lähdekoodin datan haku- ja manipulointikieli API:lle ja kyselyiden täyttämiselle jo olemassa olevalla datalla ajon aikana. Alun perin Facebook kehitti sitä sisäisesti vuosina 2012–2015 mobiili- ja web-tuotteidensa tarpeisiin, kunnes julkaisi sen avoimena lähdekoodina vuonna 2015. Facebook tarvitsi API:n, joka pystyisi kuvaamaan kaikki Facebookin datakyselyt ja olisi samaan aikaan helppo käyttää ja oppia. (4.)

GraphQL:ssä on yksi älykäs päätepiste, joka voi vastaanottaa monimutkaisia operaatioita ja palauttaa vain sen datan, minkä client-puoli tarvitsee. Koska kieli tarjoaa mahdollisuuden clientille määritellä tarkasti, millaista dataa se haluaa API:sta. Tällöin kyselyn tulos on juuri sellainen, kuin client pyytää. (4.)

3.1.1 Query

GraphQL:ssä tietoa hakevia kyselyitä kutsutaan queryiksi. Ne ovat tehokas tapa hakea tietoa, sillä GraphQL:ssä query kysyy tiettyjä oliokenttiä, missä kenttä voidaan ajatella funktiona, joka palauttaa joko primitiivisen arvon, olion tai taulukon olioita. Koska query palauttaa näitä kenttiä, client-puolella voidaan tarkasti määritellä, millaista dataa halutaan vastaanottaa ohjelmointirajapinnalta ja rajapinta taas tietää tällöin tarkasti, mitä dataa client-puoli haluaa. Tällöin vältytään tilanteilta, joissa kysely tuottaa joko liikaa dataa, josta osa on tarpeetonta, tai kyselyitä täytyisi tehdä useita tarvittavan datan saamiseksi. Queryt alkavat niin sanotulla root query-oliolla, joka toimii sisääntulokohtana dataan, joka taulukon 1 queryssä on opiskelija-olio. (5.)

Kuten taulukosta 1 näkee, GraphQL:ssä query ja sen lopputulos ovat samanmuotoisia. Tämä on tärkeää, koska tällöin kyselyn palauttavan datan muoto voidaan ennakoida. Kyselyssä haetaan vain opiskelija-olion nimitieto, jolloin serveri taas

osaa palauttaa juuri client-päädyn haluamat tiedot ja palauttaa ainoastaan nimi-tiedon. Tällöin vältytään kuljettamasta turhaa tietoa rajapintojen ylitse. (5)

TAULUKKO 1. GraphQ -kysely

Query	Lopputulos

{
 	
  opiskelija {
 	
        nimi
 	
  }
 	
}
	{
	
	  "data": {
	
	    "opiskelija": {
	
	      "nimi": "Matti"
	
	    }
	
	  }
	
	}

3.1.2 Mutaatio

GraphQL:ssä kyselyitä, jotka muokkaavat tietokannassa olevaa dataa, kutsutaan mutaatioiksi. Mutaatiot seuraavat samaa rakennetta kuin queryt, mutta niiden tulee aina alkaa avainsanalla mutation. (6.) Taulukossa 2 näkyy mutaatio, jossa luodaan uusi opiskelija, jolle annetaan argumenteiksi ikä ja nimi. Mutaatiolla pystytään myös tekemään useita kyselyitä samalla kerralla. Esimerkiksi mutaation luodessa uuden opiskelijan se voi samalla palauttaa uuden opiskelijan tiedot. Tällöin vältytään ylimääräiseltä queryltä. Taulukossa 2 haetaan juuri luodun opiskelijan id-tunniste. (5.)

TAULUKKO 2. GraphQL Mutation

Mutaatio

mutation {
 	
  createStudent(name: "Matti", age: 22) {
 	
   id
 	
  }
 	
}

3.1.3 Tyypitys ja skeema

GrapQL:ssä voidaan ennustaa kyselyn tuottama tulos, sillä GrapQL-kysely vastaa suuressa määrin kyselyn tuottamaa vastausta. Tämä johtuu siitä, että GraphQL-palvelu määrittelee tyypit, jotka kuvaavat, millaista dataa kyseisestä palvelusta voidaan hakea, ja jokainen kysely, joka GrapQL:n tarjoaman palvelun avulla suoritetaan, valikoidaan ja testataan skeemaa eli tietokantamallia vastaan, jolloin kyselyt, joita ei ole kuvattu skeemassa, eivät pääse läpi ja aiheuttavat virheilmoituksen. (7.)

Skeeman peruselementti on oliotyypit, jotka kuvaavat sitä, millaista olioita palvelusta voidaan hakea ja millaisia kenttiä nuo oliot sisältävät (7.)

Kuvassa 1 on esimerkki GraphQL-tyypistä Ajoneuvo. Ajoneuvo on oliotyyppi, mikä tarkoittaa sitä, että se on tyyppi, jolla on kenttiä, kuten kuvassa 1 olevat id, rekisterinumero ja painokentät. Kentät kuvaavat sitä, mitä tietoja voidaan hakea GraphQL-kyselyllä, joka käyttää Ajoneuvo-tyyppiä. Kentät ovat myös tyypitettyjä, mikä kertoo, millaista dataa kyselyn voidaan olettaa palauttavan. Esimerkiksi kenttä id on tyypitetty number-tyypillä, mikä kertoo sen, että id ei voi sisältää mitään muuta kuin numeroita, eikä siten voi sisältää esimerkiksi kirjaimia tai erikoismerkkejä. Id:n tyypin perässä oleva ! kertoo sen, että kenttä on määritelty

GraphQL-palvelussa siten, että sille täytyy antaa aina jonkin arvo ja se ei voi olla tyyppiä null.

GraphQL-oliotyypeille voidaan myös määrittää n kappaletta argumentteja, jotka voivat olla joko pakollisia tai vapaaehtoisia. Esimerkiksi kuvassa 1 painokentällä on pakollinen argumentti, joka on yksikkö. (7.)

```
type Ajoneuvo {
    id: Number!
    rekisterinumero: String!
    paino(yksikko: Kilo): Float
}
```

KUVA 1. GraphQL-tyyppi

3.1.4 Query- ja Mutation-tyypit

Jokaisessa GraphQL-palvelussa on vähintään yksi query-tyyppi ja mahdollisesti mutation-tyyppejä. Nämä tyypit ovat tärkeitä, sillä ne määrittelevät sisääntulokohdan jokaiselle GraphQL-kyselylle. (8.)

3.1.5 Interface ja Union tyypit

Interface on abstrakti tyyppi, joka sisältää joitain kenttiä, jotka tyypin täytyy sisältää, jotta se voi käyttää interfacea. Jos tyyppi sisältää yleisiä kenttiä, interfacea voidaan käyttää palauttamaan useita tyyppejä kerralla ilman, että jokainen tyyppi määritellään haettavaksi erikseen. Interface määrittelee kentät, jotka toteutuksen täytyy sisältää ja näin takaa sen, että kyseiset kentät ovat aina tuettuja. (8.)

Unioneja voidaan käyttää tyyppien, joilla ei ole yleisiä tyyppejä, yhdistämiseen. Nämä tyypit eivät voi olla toisia unioneja tai interfaceja. (8.)

3.2 React

React on Facebookin kehittämä JavaScript-kirjasto käyttöliittymien kehitystä varten. Sovellusnäkymä voidaan jakaa useisiin komponentteihin. Esimerkiksi kirjautumisikkuna voidaan jakaa tekstikenttään, tekstinsyöttökenttään ja painikkeisiin. Näitä kutsutaan React-ohjelmoinnissa komponenteiksi ja niiden voidaan ajatella

olevan Reactin avulla kehitettyjen sovellusten perusta, joita yhdistelemällä voidaan luoda monimutkaisempia kokonaisuuksia. Komponentit ovat itsenäisiä moduuleita, jotka palauttavat jonkin tuotteen ja niillä tulisi olla vain yksi tehtävä. Tämä yhden toiminnon periaate mahdollistaa komponenttien tehokkaamman ja helpomman uudelleen käyttämisen toisilla sivuilla. Jos komponentti alkaa kasvaa ja sille tulee useita toimintoja, tulisi se jakaa pienempiin komponentteihin. Komponentit ovat yhteen sovitettavia, jolloin komponentti saattaa käyttää lähtöarvoinaan yhden tai useamman muun komponentin tuottamia arvoja. Komponentit mahdollistavat käyttöliittymän jakamisen itsenäisiin, uudelleenkäytettäviin palasiin. Käytännössä React-komponentit ovat JavaScript-funktioita, jotka hyväksyvät parametreiksi mielivaltaisia arvoja, joita kutsutaan propseiksi, sekä palauttavat React-elementtejä, jotka kuvaavat sitä, mitä tulisi näkyä näytöllä. (9.)

3.2.1 Elinkaarimetodit

Reactissa jokainen komponentti käy elinkaaren, joka koostuu tapahtumista, joiden voidaan ajatella olevan komponentin luominen eli mounting, päivittäminen eli update ja kuolema eli unmount. (10.)

Render

Kaikista elinkaarimetodeista eniten käytetty on render()-metodi. Render-metodi löytyy kaikista React-luokista. Se piirtää komponentin käyttöliittymään. Metodin täytyy olla puhdas eli siinä ei saa olla sivuvaikutuksia ja sen täytyy aina palauttaa sama lopputulos, kun samat parametrit syötetään metodille. Se, että komponentilla ei ole sivuvaikutuksia tarkoittaa sitä, että itse metodissa ei tehdä mitään muutoksia komponentin tilaan. Jos render-metodissa tehdään muutoksia komponentin tilaan, siitä aiheutuu ikuinen luuppi, missä jokainen tilan muutos aiheuttaa uuden komponentin piirtymisen. Kuvassa 2 on esimerkki render()-metodista. (10.)

```
class tervehdys extends Comment{
  render(){
    return <div> terve {this.props.name}</div>
  }
 }
}
```

KUVA 2. Esimerkki render()-metodista

ComponentDidMount()

Kun komponentti on luotu ja se on kiinnittynyt virtuaali DOMiin, kutsutaan componentDidMountia. Sen sisällä pystytään kutsumaan setStatea ja näin muuttamaan komponentin tilaa. Kaikki metodissa tapahtuvat muutokset tapahtuvat, ennen kuin selain päivittää käyttöliittymän, jolloin komponenttien piirtämiseltä kahteen kertaan, mutta komponentin tilan muuttamien voi tässä elinkaarimetodissa johtaa erilaisiin suorituskykyongelmiin. (10.)

ComponentDidUpdate()

ComponentDidUpdate-metodia kutsutaan, kun jokin arvo päivittyy ja aiheuttaa uuden piirtämisen, mikä päivittää DOMin. Tässä komponentissa voidaan myös kutsua setState()-metodia, mutta silloin sen täytyy olla sidottuna jonkin tilan tai propsin muutokseen. Propsi tarkoittaa jotakin olion ominaisuutta, jota voidaan muuttaa. Muulloin setStaten() kutsuminen voi johtaa ikiluuppiin ja ohjelman kaatumiseen. (10.)

ComponentWillUnMount()

ComponentWillUnMount-metodia kutsutaan juuri ennen kuin komponentti irrotetaan DOMsta ja tuhotaan. Koska tämän metodin sisällä ei tapahdu uudelleen piirtämistä, niin komponentin statea ei voida muuttaa. Yleensä tämän metodin sisällä suoritetaan kaikki siivoustoiminnot, kuten välimuistissa olevan datan poistaminen. (10.)

DOM

DOM tulee sanoista ”Document Object Model” ja se on ohjelmistorajapinta HTML- ja XML-dokumentteja varten sekä edustaa sovelluksen käyttöliittymää. Sen avulla selaimessa ajettava JavaScript-ohjelma voi sen rajapintojen kautta muuttaa dokumentin rakennetta, sisältöä ja tyyliä. DOM edustaa dokumenttia solmuina ja olioina. (11.)

DOMissa kaikki voidaan ajatella solmuina. Dokumentti itsessään on pääsolmu, joka voidaan myös ajatella dokumentti solmuna. Kaikki HTML-elementit taas ovat elementtisolmuja ja kaikki HTML-attribuutit ovat attribuuttisolmuja. Tekstit HTML-elementtien sisällä taas ovat text nodeja. (11.)

Nettisivu on tässä mallissa dokumentti. Dokumentti voidaan esittää nettisivuna tai HTML-lähdekoodina, mutta molemmissa tapauksissa se on sama dokumentti. Koska DOM on oliopohjainen malli tästä samasta dokumentista, niin sitä voidaan manipuloida ohjelmointikielillä kuten JavaScriptillä. (11.)

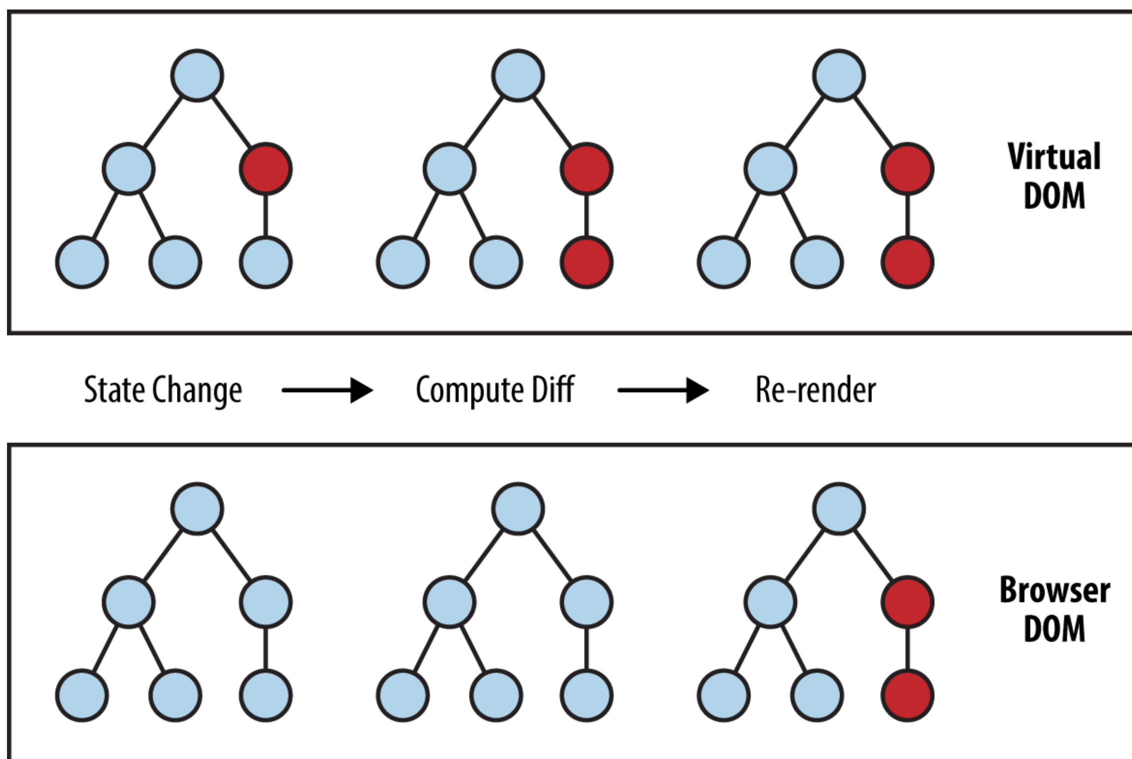
Kaikki muutokset sovelluksen käyttöliittymän tilassa saavat DOMin päivittymään, mikä kuluttaa paljon resursseja ja näin ollen hidastaa ohjelman toimintaa. DOM voidaan esittää puumallina, minkä ansiosta yksittäiset muutokset DOMissa ovat nopeita. Muutosten jälkeen päivitetty elementti ja sen lapsielementit täytyy piirtää uudelleen, jotta muutokset tulisivat näkyviin käyttöliittymässä. Tämän takia mitä useampia käyttöliittymäkomponentteja ohjelmassa on, sitä enemmän resursseja DOMin päivitys vie, sillä jokainen komponentti täytyy piirtää uudelleen. (12.)

React Virtual DOM

React käyttää kahta oletusta vähentämään vertailujen määrää olettamalla, että kaksi erityyppistä elementtiä tuottavat erilaiset puut ja ohjelmoija voi avain-propertyjen avulla kertoa, mitkä lapsielementit ovat vakaita uudelleen piirtämisten välillä. (13.)

Virtual DOM edustaa virtuaalisesti DOMia ja siihen on tallennettu käyttöliittymän ideaalitila. Aina kuin sovelluksen tila muuttuu, niin ”oikeaa” DOMia ei tarvitse päi-

vittää jokaisen muutoksen jälkeen, vaan virtuaalinen DOM päivittyy oikean DOMin sijasta. Tämä ansiosta virtuaalista DOMia voidaan vertailla edelliseen virtuaaliseen DOMiin ja laskea, mikä päivityspolku on kaikista vähiten resursseja vievä suorittaa. Kun vertailu on tehty, React päivittää ainoastaan ne elementit, joihin on tullut muutoksia ja näiden lapsielementit, kuten kuvassa 3 oikeaan DOMiin. Tällöin muutokset vaativat vähemmän resursseja, kuin suora DOMin muokkaaminen jokaisen muutoksen jälkeen. (12.)



KUVA 3. Laskennan suorittaminen virtuaali-DOMissa vähentää uudelleen piirtämistä. (14.)

3.2.2 React Hooks

Hookit lisättiin React 16.8 -versiossa korvaamaan luokkarakenteiset komponentit. Hookit ovat funktioita, jotka mahdollistavat Reactin ominaisuuksien käyttämisen ilman luokan kirjoittamista. React tarjoaa muutamia sisäänrakennettuja hookeja kuten useState, useEffect ja useContext. Käyttäjä voi myös luoda omia hookeja tilallisen logiikan uudelleenkäyttämiseksi eri komponenteissa. (15.)

3.2.3 UseState Hook

UseState Hookia käytetään komponentin tilamuuttujien esittelyyn. Kuvassa 4 tilamuuttujaa kutsutaan nameksi. UseState mahdollistaa muuttujan tilan tallentamisen funktioiden kutsumisen ja siitä seuraavien komponentin uudelleen piirtämisien välillä. Hook voidaan myös alustaa jollain arvolla, kuten kuvassa 4, mutta se ei ole pakollista. (16.)

UseState palautta aina kaksi arvoa, jotka ovat muuttujan nykyinen tila sekä funktio, jolla sitä voidaan muuttaa. Esimerkiksi kuvassa 4 namen kutsuminen palauttaa namen tämänhetkisen arvon, kun taas setName(Matti) kutsuminen asettaisi komponentin tilaksi Matti. (16.)

```
import React, { useState } from "react";

export const student = () => {
  const [name, setName] = useState([]);

  return (
    <input
      type="text"
      id="studentName"
      value={name}
      onChange={event => {
        setName(event.target.value);
      }}
    />
  );
};
```

KUVA 4. React useState Hook

3.2.4 Effect Hook

Effect Hookilla voidaan lisätä sivuvaikutuksia funktionaaliseen komponenttiin. Kaksi tyypillisintä sivuvaikutusta React-komponenteilla ovat sellaiset, jotka vaativat siistimistä jälkeenpäin ja sellaiset, jotka eivät vaadi. UseEffectin voidaan ajatella olevan yhdistelmä componentDidMounttia, componentDidUpdatea sekä componentWillUnmounttia. UseEffectiä käyttämällä kerrotaan Reactille, että komponentin tulee suorittaa jokin toiminto piirtämisen jälkeen. React muistaa funktion, johon viitattiin useEffection sisällä ja kutsuu sitä uudelleen piirtämisen jälkeen. (17.)

UseEffectille voidaan myös asettaa ehtoja, joiden täyttymisen jälkeen React kutsuu effectin sisällä olevaa funktiota. Tällainen tilanne voi esimerkiksi olla tilanne, jossa kysely tulee suorittaa vasta jonkin toisen kyselyn jälkeen. (17.)

3.2.5 Contex Hook

Context Hook tarjoaa mahdollisuuden lähettää dataa komponenttipuun läpi ilman että propsit tulisi kuljettaa käsin jokaisen kerroksen läpi. Normaalisti React-applikaatioissa data kuljetetaan vanhemmalta lapsikomponentille propsien avulla, mutta tämä voi olla hieman hankala ratkaisu joidenkin erityistyyppisten propsien käytössä, kuten esimerkiksi lokaalit asetukset, kieli ja käytettävä UI-teema, joita käytetään useissa komponenteissa läpi koko ohjelman. (18.)

3.3 Apollo Client

Apollo Client on client-kirjasto GraphQL:n kanssa työskentelyä varten. Sillä voidaan tehdä kyselyitä ja mutaatioita ja tallentaa kyselyiden tuottamaa dataa välimuistiin. Apollo Clientin käyttö ei myöskään vaadi mitään tiettyä GraphQL API:a vaan toimii niiden kaikkien kanssa. (19.)

3.3.1 Kyselyiden suorittaminen

Apollo tarjoaa kahta erillistä tapaa suorittaa GraphQL-kyselyitä: useQuery ja useLazyQuery.

UseQuery on React hook ja se on Apollo-sovellusten pääasiallinen ohjelmointirajapinta kyselyiden suorittamiseen. Se laukeaa, kun komponentti, jonka sisällä se on, piirretään uudelleen. Hookille voidaan myös antaa ehtoja, jolloin sen tulee piirtyä uudelleen käyttämällä vapaaehtoista skip-valintaa, joka estää kyselyn laukeamisen ehdon ollessa tosi. UseQuery palauttaa Apollo Clientille olion, joka sisältää loading, error ja data propertyt. Näiden propertyjen arvot muuttuvat kyselyn suorittamisen aikana, mikä mahdollistaa erilaisten käyttöliittymäkomponenttien piirtämisen kyselyn suorittamisen aikana. Loading propertyn arvon ollessa tosi, on kyselyn suorittaminen vielä kesken ja käyttöliittymään voidaan piirtää uudelleen esimerkiksi latauskuvakkeen eli spinnerin indikoimaan käyttäjälle latauksen olevan vielä käynnissä. Jos loading on false ja virhettä ei tapahdu, kysely on valmis ja sen palautusarvo asetetaan data propertyyn. Virheen sattuessa mahdollinen virhesanoma asetetaan error propertyyn. (20.)

UseLazyQuery on React hook, joka laukaisee kyselyn, jos tietyt ehdot täyttyvät. UseLazyQuery käyttäytyy lähes samalla tavalla kuin useQuery, mutta toisin kun useQuery se ei heti kutsuttaessa laukaise siihen sidottua kyselyä vaan palauttaa funktion, jota kutsumalla voidaan laukaista kysely halutun tapahtuman jälkeen. (20.)

3.3.2 Mutaatioiden suorittaminen

Apollo-sovellukset käytävät mutaatioiden suorittamiseen useMutation React-hookia. Komponentin piirtyessä hook palauttaa mutaatio funktion, jota kutsumalla voidaan laukaista itse mutaatio. Hook palauttaa myös olion, joka kuvastaa mutaation laukaisemisen tilaa, jotka ovat propertyt loading, error ja data. Näiden Propertyjen toiminta kuvattiin edellisessä kappaleessa ”Kyselyiden suorittaminen”. UseMutationille voidaan antaa valinnaisia parametrejä, kuten variables parametriä, jolla voidaan määritellä GraphQL-muuttujat, joiden arvoilla mutaatio suoritetaan. (21.)

3.3.3 Datan tallennus

Apollo Clientin avulla dataa voidaan tallentaa välimuistiin, jolloin jo haettua dataa voidaan käyttää toisissa queryissä, jos niiden id-tiedot täsmäävät. Uniikki tunniste

luodaan yhdistämällä kyselyn tyyppinimi ja mahdollinen id-kenttä. Jos Apollo Client tunnistaa kyselyn, jonka tunniste on sama, kun jo muistissa olevan datan, tehdään kysely muistissa olevasta datasta, eikä kysely mene serverille asti, jolloin säästetään aikaa. (22.)

Esimerkiksi jos on haettu ensin kaikki henkilöt listaan ja sitten halutaan näyttää vain yhden henkilön tiedot, Apollo Client mahdollistaa näiden tietojen hakemisen välimuistista. Siten kyselyä ei tarvitse suorittaa serverille asti, vaan se voidaan tehdä jo haetusta datasta, jolloin säästetään aikaa.

3.4 Google Maps JavaScript API

Google Maps JavaScript API on ohjelmointirajapinta, joka tarjoaa mahdollisuuden luoda karttoja ja kustomoida niitä. API:n avulla voidaan myös suunnitella reittejä ja esittää sijaintiin sidottua dataa. API:n avulla luotuja karttoja voidaan muokata käyttämällä erilaisia kerroksia kartan päällä ja käyttämällä erilaisia tyylittelyjä. Google Maps Javascript API tarjoaa myös mahdollisuuden vuorovaikuttaa kartan kanssa erinäisien tapahtumien, kuten hiirellä raahauksen, painamisen ja leijumisen avulla. Näihin tapahtumiin voidaan sitoa erinäisiä funktioita, jotka laukeavat kyseisen tapahtuman jälkeen. (23.)

3.5 IIS

IIS on lyhenne sanoista Internet Information Services. Se on web-serveriohjelmistopaketti, ja sen on kehittänyt Windows Server. Sitä käytetään isännöimään nettisivuja ja muuta sisältöä webissä. IIS käyttää HTTP-protokollaa viestimiseen nettiserverien ja käyttäjien välillä sekä tietoturvaprotokollana Socket Layer eli SSL protokollaa kommunikaation salaamiseen. (24.)

IIS voi palvella tavallisia HTML-nettisivuja ja dynaamisia nettisivuja, kuten ASP.NET-applikaatioita. Kun käyttäjä navigoi staattiselle nettisivulle, IIS lähettää HTML:n ja sivustolle kuuluvat kuvat käyttäjän selaimelle. Kun taas dynaamisten nettisivujen tapauksessa käyttäjän navigoidessa sivustolle, IIS ajaa kaikki applikaatiot ja prosessoi sivustoon kuuluvat koodit, jonka jälkeen se lähettää synty-

neen datan käyttäjän selaimeen. Vaikka IIS sisältää kaikki tarvittavat ominaisuudet nettisivun isännöintiin, se tukee myös laajennusosien lisäämistä uusien ominaisuuksien lisäämiseksi serveriin. (24.)

3.6 Windows Communication Foundation

3.6.1 Perusteet

Windows Communication Foundation eli WCF on ohjelmointimalli serveripainotteisten ohjelmistojen kehitystä varten käyttäen .NET ohjelmistokehystä. WCF:n Palvelu osa mahdollistaa helpon ohjelman eri osien välisen kommunikoinnin tai helpon kommunikoinnin ohjelmistojen välillä käyttäen viestien välitykseen erilaisia palvelun päätepisteitä.

WCF mahdollistaa datan keräämisen ja jakamisen useiden clientien välillä samaan aikaan vaikuttamatta käyttäjäkokemukseen. Data voidaan antaa palvelulle, esimerkiksi. Tietokantaan tallentamista varten. Kun palvelu on vastaanottanut datan, käyttäjän toiminta käyttöliittymässä ei voi enää vaikuttaa datan tallennukseen. Tällöin esimerkiksi jos käyttäjä tappaa ohjelman, se ei enää vaikuta tiedon tallennukseen, koska se on jo annettu palvelulle. Tämmöisiä metodeita kutsutaan asynkronisiksi metodeiksi. (25.)

3.6.2 Sopimukset

.NET-ohjelmointikehyksessä sopimuksilla tarkoitetaan erilasia ehtoja ohjelman eri osien koodeille. Sopimus voi esimerkiksi määritellä, mitkä ennen ja jälkeen ehdot tulee täyttyä, että ohjelman koodi suorittaa, jonkin tietyn metodin. Sopimuksilla voidaan, myös määrittää invariantti muuttuja, jolla tarkastetaan ohjelman oikea toiminta. Yhdelle muuttujalle voidaan määritellä useampia muuttujia. Koska sopimuksia on useita erilaisia ja niillä kaikilla on omat merkityksensä ja vaikutuksensa, niitä voidaan yhdistellä ja näin saavuttaa suuri määrä erilaisia ehtoja yhdelle alkiolle. Esimerkiksi, Browsable-sopimusta voidaan käyttää määrittelemään, kuinka ominaisuus näytetään käyttäjälle, kun Browsable asetetaan Booleaan muuttujalla tilaan false, sitä ei näytetä käyttäjälle vaan se piilotetaan. Sopimus voi myös määritellä kuinka eri luokat ovat vuorovaikutuksessa keskenään. (26.)

WCF:ssä yleisimmät sopimustyypit ovat data-, palvelu- ja operaatiosopimukset. Datasopimus tarkoittaa sopimusta palvelun ja clientin välillä, joka kuvaa, kuinka dataa vaihdetaan niiden välillä. Clientin ja palvelun ei tarvitse jakaa samoja tyyppejä vaan sama sopimus, jossa kuvataan tarkasti jokaisen parametrin, palautusarvon tyyppi ja mitä dataa sarjallistetaan eli muutetaan XML:ksi. Näitä datasopimuksia käytetään useissa paikoissa WCF-palveluissa. Esimerkiksi niitä käytetään määrittelemään, mitä metodeja palvelu voi käyttää. Ne auttavat ymmärtämään millaista dataa metodit voivat käyttää. (26.)

Kun dataa kulkee palveluun ja siitä ulos olioiden mukana, on tärkeää, että oliot on määritelty käyttäen palvelusopimusta. Siinä määritellään kaikki muuttujat, joita välitetään clientiltä palveluun tai toisinpäin. Jos ominaisuudella ei ole sopimusta, joka määrittäisi sen datasopimuksen jäseneksi, se lähetetään joko tyhjänä tai oletus arvona. Palvelusopimuksia käytetään määrittelemään palvelu, kuinka se käyttäytyy ja kommunikoi clientin kanssa. (26.)

Operaatiosopimusta käytetään: kuin datasopimusta ja ne määrittelevät, kuinka erilaiset palvelulle kuuluvat metodit ja funktiot käyttäytyvät. Näitä sopimuksia käytetään yleensä palvelusopimuksen määrittelemän palvelun sisällä. (26.)

3.6.3 Päätepisteet

Kaikki WCF-palvelut vaativat päätepisteitä toimiakseen. Ne määrittelevät yleensä, millä tasolla client voi ottaa yhteyttä palveluun ja kuinka yhteys luodaan. Jos palvelulla ei ole päätepisteitä, client ei voi ottaa siihen yhteyttä. Päätepisteillä on kolme tärkeää osaa: osoite, sidonta ja sopimus. (27.)

Kaikilla päätepisteillä on osoite, jolla tarkoitetaan päätepisteen sijaintia. Se voidaan antaa joko koodissa tai erillisessä asetukset tiedostossa. Usein osoite kuitenkin määritellään asetukset tiedostossa, koska osoite voi vaihtua ja sen muuttaminen itse koodiin on vaikeampaa. Tähän osoitteeseen client ottaa yhteyttä kommunikoidessaan palvelun kanssa. (27.)

Sidonnassa määritellään, kuinka päätepisteen kanssa voidaan kommunikoida. Se sisältää tiedon kuljetus protokollasta ja salauksesta. Yleisimpiä kuljetus protokollia ovat TCP ja HTTPS, jotka määrittelevät kuinka lähtevät ja tulevat viestit käyttäytyvät. (27.)

Päätepiste tarvitsee, myös sopimuksen, jossa määritellään mitä ominaisuuksia se paljastaa clientille. Näitä ominaisuuksia ovat operaatiot, joita client voi kutsua, viestien muoto, millaisia parametrejä käytetään ja millaista vastausta client voi odottaa. (27.)

3.7 Hot Chocolate

Hot Chocolate on .NET-GraphQL-alusta, joka mahdollistaa GraphQL-kerroksen lisäämisen järjestelmään. Sen avulla voidaan määritellä skeemaa käyttäen .NET-tyyppejä. Hot Chocolatessa skeema luodaan SchemaBuilder-metodia käyttäen. Sen avulla määritetään, mitä tyyppejä skeemassa käytetään. Nämä tyypit määrittelevät, millaista dataa palvelusta voidaan hakea. Kaikki palvelun kyselyt validoidaan ja toteutetaan tätä skeemaa vastaan. Hot Chocolate mahdollistaa skeeman tyyppien sitomista, joihinkin tiettyihin .NET-tyyppeihin. Tällöin voidaan määrittää, että .NET-tyyppi string sidotaan skeemassa määriteltyyn tyyppiin StringType, jolloin skeema käsittelee sen tänä tyyppinä. Tämä toiminto mahdollistaa esimerkiksi front-endin ja back-endin date tyyppien määrittelyn samaan muotoon, jolloin ohjelma käsittelee molempia tyyppejä samalla tavalla. (28)

4 TYÖN TOTEUTUS

Työn toteutusta varten sain esitietona omakohtaista kokemusta web-ohjelmoinnista työskentelemällä PiiMega Oy:ssä useissa projekteissa full stack -ohjelmistokehittäjänä. Tämä käytännön kokemus auttoi käytettävien työkalujen kuten Reactin, TypeScriptin sekä C#:n käytössä, sillä olin käyttänyt samoja työkaluja jo aikaisemman projektin aikana. Tuotettavan ohjelman vaatimuksia määriteltiin PiiMegan sisällä pidetyissä ohjauspalavereissa Product Owner Juha Valkolan ja Heikki Laulajaisen kanssa, sekä asiakkaan kanssa kahden viikon välein pidetyissä katselmointipalavereissa. Juha Valkola toimi työnaikaisena ohjaajana. Työn aikana pidettiin kerran viikossa tiimin sisäinen palaveri, jossa katsottiin yleisempiä kaikkiin projekteihin liittyviä asioita ja noin kerran kahdessa viikossa pidettiin ohjauspalavereita, joissa tarkasteltiin tarkemmin projektin etenemistä. Palavereissa katsottiin saadut tulokset ja asiakkaalta saadut palautteet tuotteen etenemisestä ja tarkennettiin vaatimuksia tai sovittiin jostain muutoksista.

Käyttöliittymän ohjelmointiin käytettiin Microsoftin Visual Studio Codea, Node.js:ää, Typescript-tulkkia sekä npm-paketinhallintaa. Ohjelma suunniteltiin mahdollistamaan sopimusten luomisen koskien erinäisiä metsänhoidollisia töitä, kuten taimien istuttamista, harvennushakkuita ja raivausta. Ohjelmalla pystytään myös näyttämään sopimusta koskevat alueet kartalla, tekemään erinäisiä työohjeita urakoitsijoille ja sitomaan nämä työohjeet tiettyyn alueeseen kartalla. Käyttäjänäkymän ja back-endin väliseen kommunikointiin käytettiin HotChocolatea sekä Apolloa.

4.1 Työn tausta

PiiMega Oy:n kehittämällä ForestPro-tuotteella voidaan suunnitella ja hallinnoida metsänhoidollisia töitä, mutta tällöin käyttäjä on sidottuna työskentelemään PC-tietokoneella, johon on asennettuna ForestPro-työpöytäsovellus.

Ohjelma sai alkunsa tarpeesta tarjota asiakkaille mahdollisuus suunnitella ja hallinnoida metsänhoidollisia töitä web-selaimella ja näin ollen mahdollistaa työskentely myös kentällä sekä ohjelman käyttö myös mobiililaitteella.

4.2 Vaatimukset ja määrittely

Opinnäytetyön vaatimuksena oli toteuttaa ohjelma, jonka avulla pystyttäisiin tarjoamaan asiakkaille ja heidän urakoitsijoilleen mahdollisuus suunnitella ja hallinnoida metsänhoidollisia töitä web-selaimesta. Tekniset vaatimukset olivat, että tuotteen tulisi olla web-pohjainen ja siinä tulisi hyödyntää mahdollisimman suuressa määrin jo olemassa olevia tietokantarakenteita, back-endiä ja GraphQL API-kerrosta. Käyttöliittymän vaatimuksina oli ensisijaisesti tukea käytettävyyttä PC-tietokoneilla, mutta toimia myös mobiililaitteilla.

Projekti määriteltiin kaksivaiheiseksi, jonka ensimmäisessä vaiheessa tavoitteena oli luoda ohjelmasta versio, jossa käyttöliittymä olisi valmis ja sitä pystyttäisiin esittelemään asiakkaille. Sen tulisi sisältää näkymät sopimusten tarkasteluun taulukossa, sopimuksen tarkemman tarkastelun mahdollistava näkymä, jossa esiteltäisiin sopimukseen kuluvat alueet kartalla ja alueisiin kuuluvat työohjeet ja sopimukselle tarkoitettujen liitetiedostojen tarkastelun mahdollistava näkymä.

Toisessa vaiheessa tavoitteena oli tuottaa ohjelmasta versio, jota pystyttäisiin testaamaan oikealla datalla asiakkaan testiympäristössä. Tässä vaiheessa tulisi rakentaa ohjelman dataliikenteen mahdollistaminen.

Projektin etenemistä seurattiin Jira-tiketeillä, jotka oli jaoteltu joko ensimmäiseen tai toiseen vaiheeseen. Näin pystyttiin seuraamaan projektin etenemistä ja kuhinkin tehtävään kuluvaa aikaa. Sovelluksen määritelmä tarkennettiin projektin edetessä. Projektin kulku pyrittiin toteuttamaan ketteriä menetelmiä käyttäen tekemällä tiivistä yhteistyötä asiakkaan kanssa käymällä noin kerran kahdessa viikossa projektin kulkua läpi katselmointipalaverissa yhdessä asiakkaan kanssa. Näissä palavereissa katsottiin jo luotuja ominaisuuksia ja keskusteltiin mahdollisista tarkennuksista määrittelyyn tai mihin suuntaan mitäkin ominaisuutta tulisi kehittää.

4.3 Tekniset valinnat

Projektin isot tekniset valinnat tehtiin heti projektin alussa. Ensimmäinen päätös oli toteuttaa ohjelma jo kehitteillä olleelle PiiMegan ForestPortal-alustalle yhtenä osana ohjelmaa. Käyttöliittymä päätettiin toteuttaa React-käyttöliittymäkirjastoa käyttäen, koska React tarjoaa komponenttipohjaisen rakenteen, mikä tekee ohjelman laajentamisesta helppoa komponenttien karttuessa. React mahdollistaa nopean uudelleen piirtämisen Virtuaali DOMin avulla, sillä DOMin päivitys on yleensä esteenä sulavalle web-applikaation käytölle varsinkin suuremmissa ohjelmissa. React-kirjaston avulla voidaan tehdä käyttöliittymiä, jotka toimivat sekä PC-tietokoneissa että mobiililaiteissa. Myös se, että Reactia käyttävät suuret teknologiayritykset kuten Facebook, Instagram ja Netflix. Tämän ansiosta voidaan olettaa, että kiinnostus Reactiin ei lopu yllättäen ja sen kehittäminen jatkuu. Tämä auttaa ennakoimaan teknologian vanhenemista. Käyttöliittymässä ohjelmointikielenä päätettiin käyttää TypeScriptiä JavaScriptin sijasta, koska TypeScriptin avulla koodia voidaan tyypittää, jolloin koodi on helpompaa lukea ja ymmärtää ja tällöin vältytään koodausvirheiltä, joita voi syntyä, kun väärää tietotyyppiä oleva arvo asetetaan muuttujaan.

Back-endin ja tietokannan tapauksissa ei tarvinnut tehdä teknisiä valintoja, sillä niissä käytettiin jo olemassa olevia ratkaisuja. Päätös käyttää jo olemassa olevaa back-endiä pohjautui siihen, että tällöin pääsimme käsiksi jo olemassa oleviin funktioihin ja tietokantakyselyihin, joita pystyttiin hyödyntämään ohjelmiston kehityksessä. Back-end on toteutettu käyttäen ASP.NET-, C#- ja VisualBasic-tekniikoita. Tietokantana käytettiin jo olemassa olevaa SQL-tietokantaa, koska siellä oli jo valmiina suurin osa tarvittavista tietokantarakenteista.

4.4 Projektin aloitus

Projekti aloitettiin erinäisillä suunnitelupalavereilla, joissa käytiin asiakkaan toiveita läpi ja suunniteltiin käyttöliittymän ulkoasua ja sitä, millaisia toimintoja sovelluksessa tulisi olla. Käyttöliittymän suunnitellussa päätettiin noudattaa Material Design-periaatteita, jolloin voitaisiin käyttää hyväksi jo käyttäjille tuttuja mekanismeja, jolloin ohjelma näyttäisi jo valmiiksi tutulta ja käyttäjä tietäisi jo automaattisesti, mistä tietyt toiminnot löytyisivät.

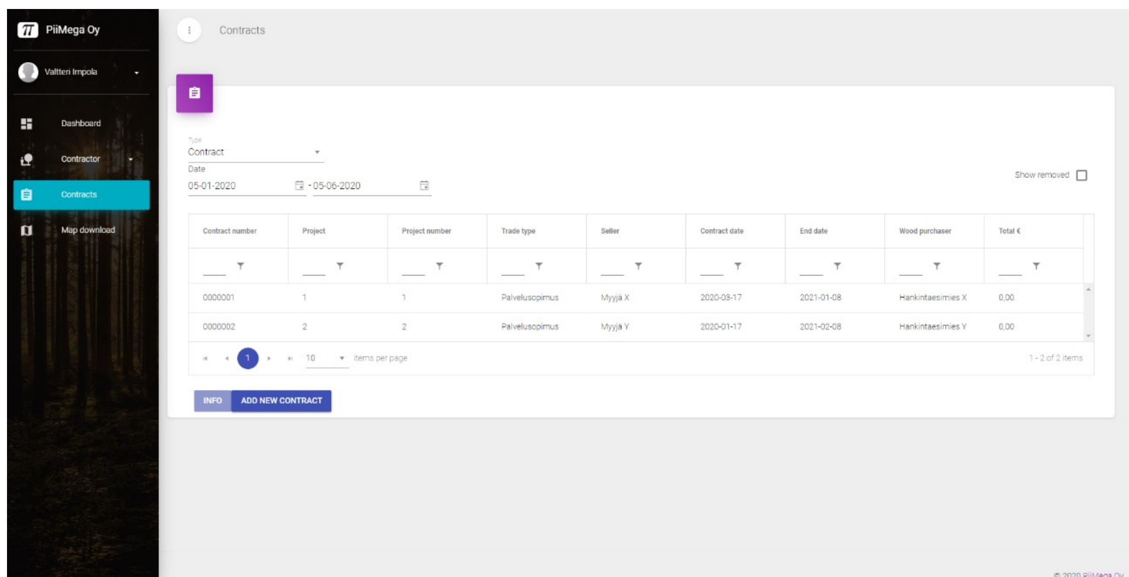
4.5 Toteutuneet näkymät

Sovellus on jaettu sopimuslistaus-, perustiedot-, lohkot- ja liitetiedostot-näkymiin.

4.5.1 Sopimuslistaus

Kuvassa 5 näkyy sopimuslistaus, jossa näytetään kaikki käyttäjälle linkitetyt sopimukset sekä tarjoukset. Tarjouksien ja sopimuksien välillä voidaan liikkua ”Sopimuksen tila”-pudotusvalikon avulla, jolloin taulukko päivittyy valinnan mukaisesti. Taulukkoa voidaan suodattaa erilaisin filtterein, kuten päivämäärärajauksella, ja jokaista kolumnia voidaan suodattaa erikseen, jolloin taulukko päivittyy käyttäjän antamien parametrien mukaan. Kun käyttäjä valitsee jonkin sopimuksen, infopainike tulee aktiiviseksi ja sitä painamalla valitun sopimuksen tiedot avautuvat uuteen näkymään.

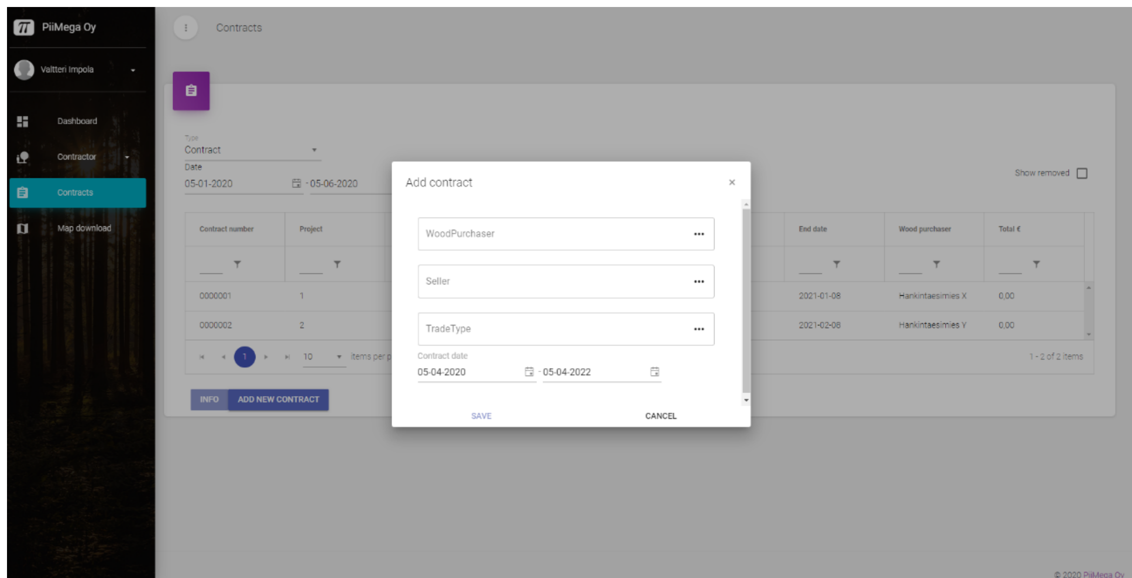
Taulukosta voidaan myös tarkastella poistettuja sopimuksia ja tarjouksia valitsemalla ”Näytä poistetut”-valintaruudun, jolloin poistetuksi merkittyjä sopimuksia voidaan tarkastella ja suodattaa samoilla valinnoilla, kuin olemassa olevia sopimuksia. Tästä valikosta valittuja sopimuksia voidaan myös palauttaa voimassa oleviksi sopimuksiksi.



KUVA 5. Sopimuslistaus

Tältä sivulta voidaan myös luoda uusia sopimuksia sekä tarjouksia. Sopimuksen voi luoda sopimuslistauksesta painamalla ”ADD NEW CONTRACT”-painiketta,

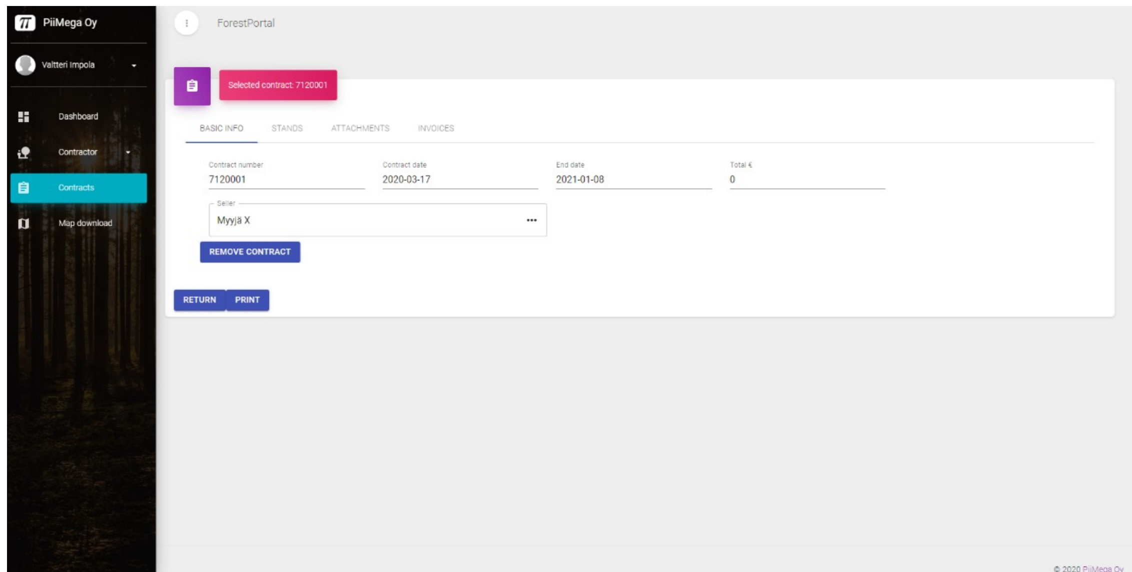
jolloin avautuu kuvan 6 mukainen sopimuksen luontinäkymä. Jos taulukko on suodatettu näyttämään tarjoukset, niin ”ADD NEW QUOTATION”-painikkeella voidaan luoda uusi tarjous. Käyttäjä valitsee hankintaesimiehen, myyjän ja sopimustyypin sekä aloitus- ja lopetuspäivämäärät, jonka jälkeen ohjelma luo uuden sopimuksen ja generoi sille sopimusnumeron, jonka jälkeen ohjelma avaa luodun sopimuksen.



KUVA 6. Sopimuksen lisäysnäkymä

4.5.2 Perustiedot

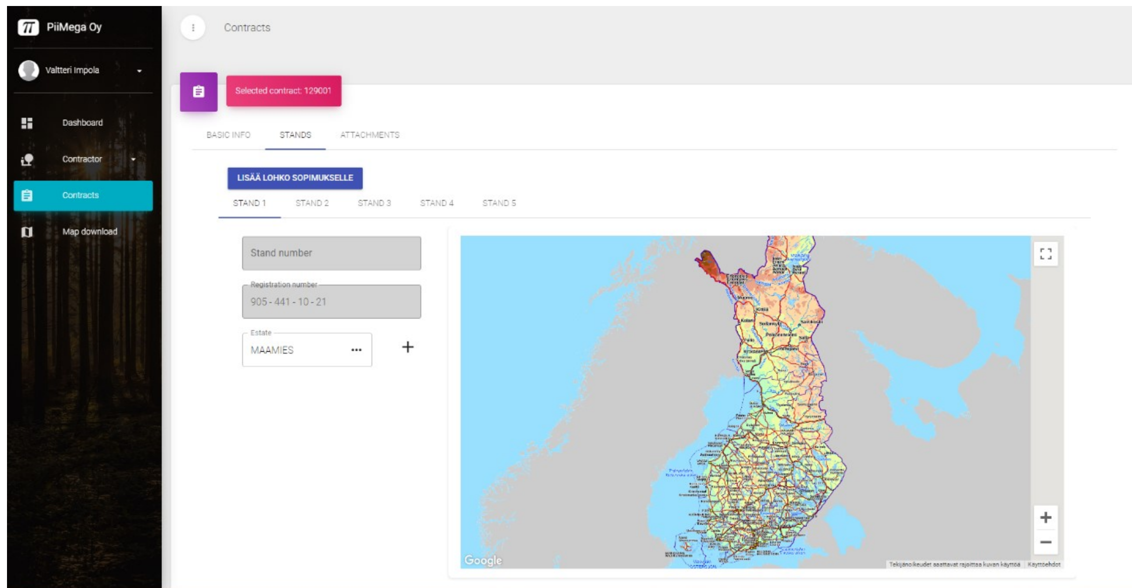
Kuvassa 7 näkyvässä perustiedot-näkymässä voidaan tarkastella sopimuksen tietoja, kuten sopimuksen aloitus- ja lopetuspäiviä. Sivulla voidaan myös vaihtaa sopimuksen luonnissa siihen liitettyä myyjää, sekä tarvittaessa poistaa sopimus, jolloin sopimus siirtyy sopimuslistauksessa ”Näytä poistetut”-filtterin taakse. Jos tarkastellaan tarjousta, voidaan siitä luoda sopimus, jolloin ohjelma luo tarjouksesta uuden sopimuksen ja aukaisee sen tarkastelua varten.



KUVA 7. Perustiedot-näkymä

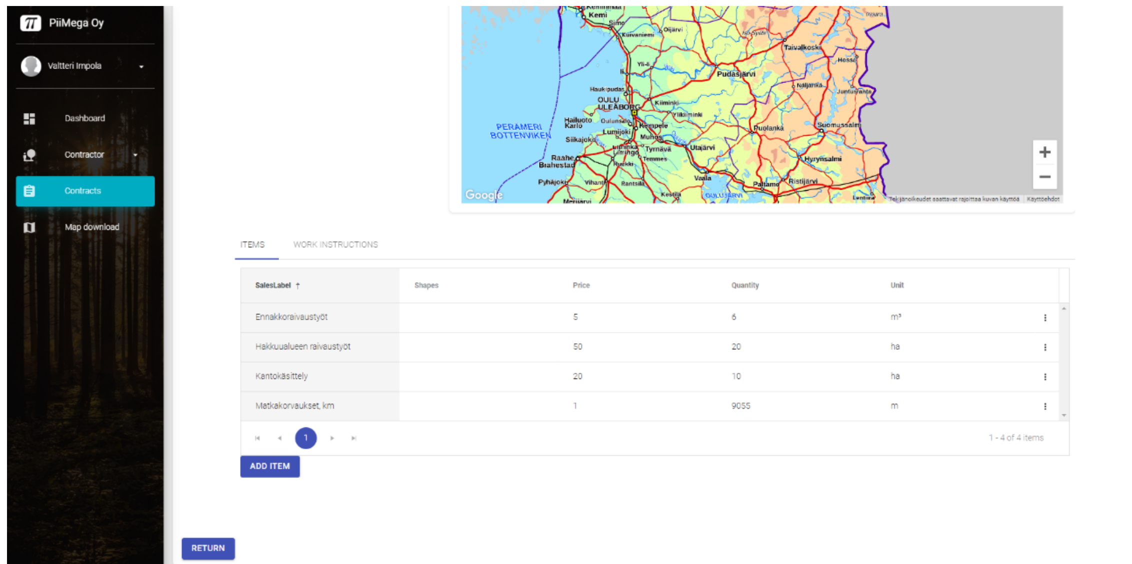
4.5.3 Lohkot

Kuvassa 8 näkyvässä lohkot-näkymässä voidaan tarkastella ja hallinnoida sopimukselle kuuluvien lohkojen tietoja. Lohkolla tarkoitetaan maa-aluetta, jota sopimus koskee. Sivulla sopimuksen jokaiselle lohkolle luodaan automaattisesti oma välilehti, jonka sisällä voidaan hallinnoida kyseisen lohkon tietoja ja sille kuuluvia työohjeita ja myyntinimikkeitä. Jokaisella välilehdellä näytetään lohkon numero sekä tilarekisterinumero, joka kertoo missä lohko sijaitsee ja se muodostuu yhdistämällä tila-, kunta- ja kylänumerot. Sivulla voidaan myös vaihtaa lohkon tilaa. Se voidaan tehdä kahdella eri tavalla, joko valitsemalla jo olemassa oleva tila tilalistauksesta tai luomalla kokonaan uusi tila. Jos lohkolle on piirretty kuvioita, kysyy ohjelma, haluatko varmasti vaihtaa lohkon tilaa, sillä tällöin tilan vaihtaminen johtaa lohkon kuvioiden poistamiseen.



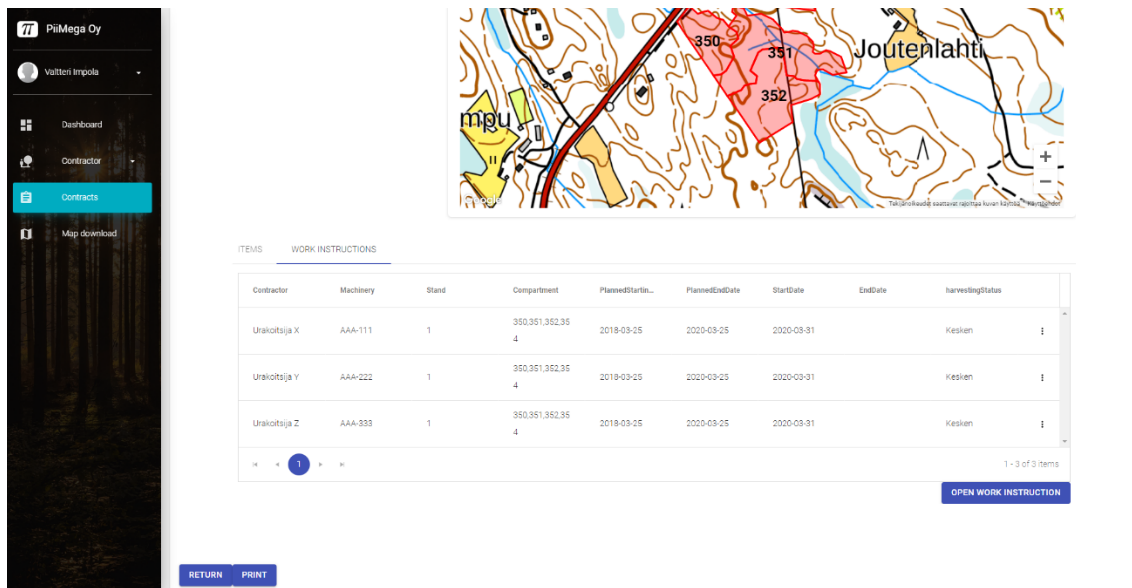
KUVA 8. Lohkot-näkymä

Lohkot-näkymän alaosassa on kuvan 9 mukainen myyntinimiketaulukko, josta voidaan tarkastella kyseiselle lohkolle lisättyjä myyntinimikkeitä. Taulukosta voidaan myös lisätä uusia myyntinimikkeitä lohkolla olevalle leimikoille klikkaamalla ensin haluttua leimikkoa kartalta ja sitten painamalla ”ADD ITEM” -painiketta, jolloin avautuu myyntinimikkeen lisäysdialogi. Kun uusi myyntinimike on luotu, lisätään se automaattisesti myyntinimiketaulukkoon. Myyntinimikkeet kertovat, mitä kyseiselle leimikolle aiotaan tehdä ja kuinka paljon, esimerkiksi Tienteko, 1km, hinta 1000/km. Lohkolla voi olla n kappaletta erilaisia myyntinimikkeitä. Myyntinimiketaulukossa voidaan myös poistaa myyntinimikkeitä painamalla taulukon reunassa olevaa Menu-näppäintä, jos niille ei ole linkitetty yhtään työohjetta. Myyntinimike-riviä klikkaamalla voidaan valita kyseinen rivi, jolloin ohjelma myös valitsee riviin linkitetyn leimikon kartalta ja tarkentaa siihen.



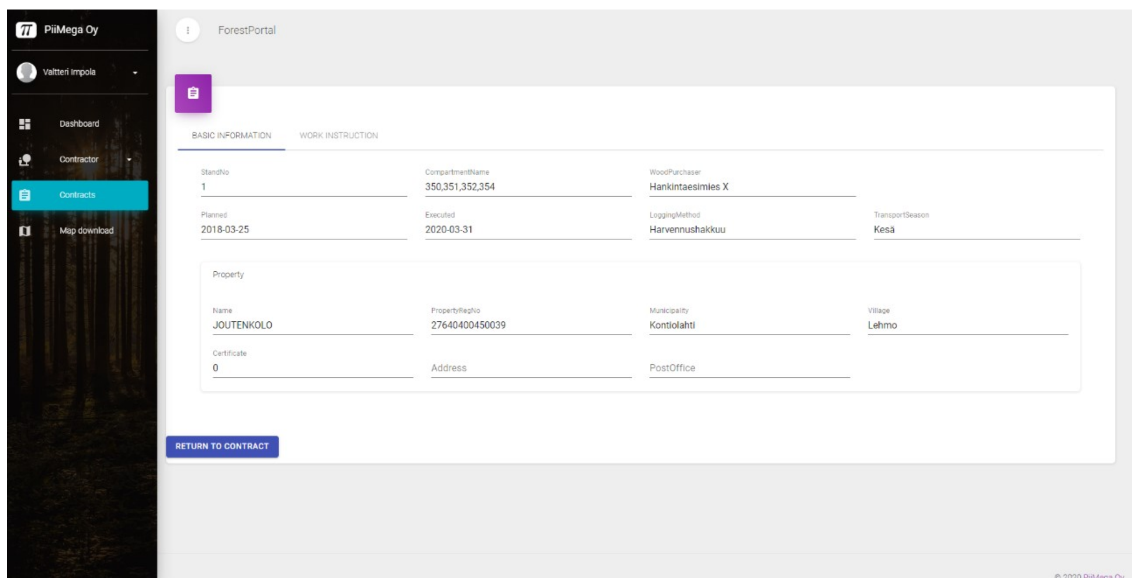
KUVA 9. Myyntinimikelistaus

Jokaiselle myyntinimikkeelle voidaan luoda työohjeita, joissa kerrotaan suunnitellut aloitus- ja lopetuspäivät, kuka urakoitsija tekee työn, millä kalustolla ja kuinka paljon, sillä samalle myyntinimikkeelle kuuluvat työt voidaan jakaa useammille urakoitsijoille. Kuvassa 10 näkyvässä työohjetaulukossa voidaan myös kopioida työohjeita, jos halutaan tehdä nopeasti uusi työohje samoilla valinnoilla. Työohje voidaan myös poistaa, jos sille ei olla vielä tehty suoritteita. Valitsemalla jonkin työohjeen ja painamalla sen jälkeen ”OPEN WORK INSTRUCTION”-painiketta avautuu työohjenäkymä, missä voidaan tarkastella valittua työohjetta tarkemmin.



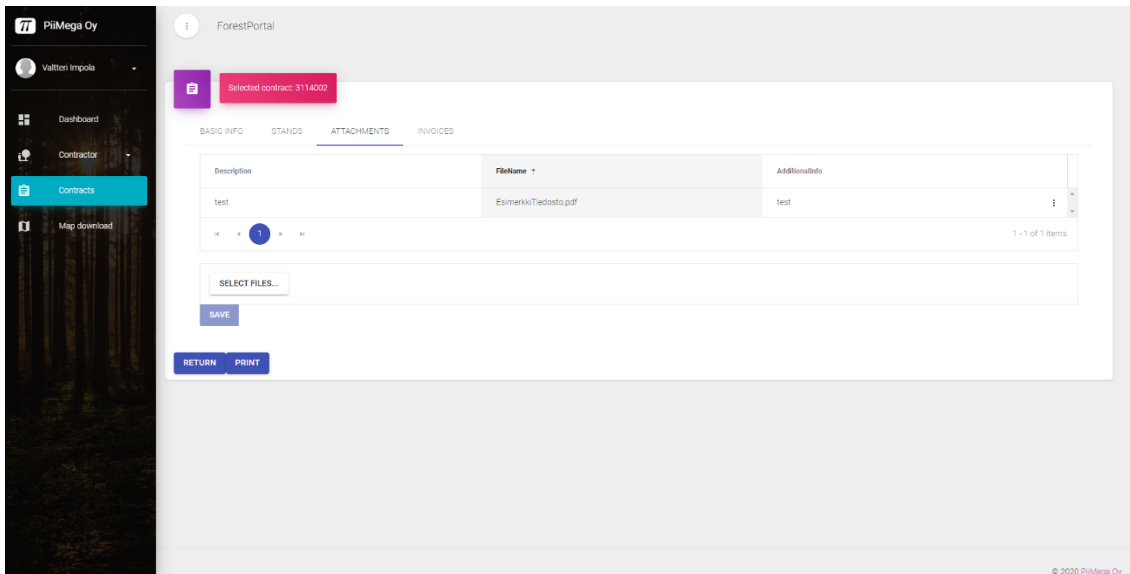
KUVA 10. Työohjelistaus

Kuvassa 11 näkyvässä näkymässä on kaksi eri välilehteä, perustiedot ja työohje. Perustiedot-välilehdellä voidaan tarkastella valitun työohjeen tilatietoja sekä myyjän tietoja.



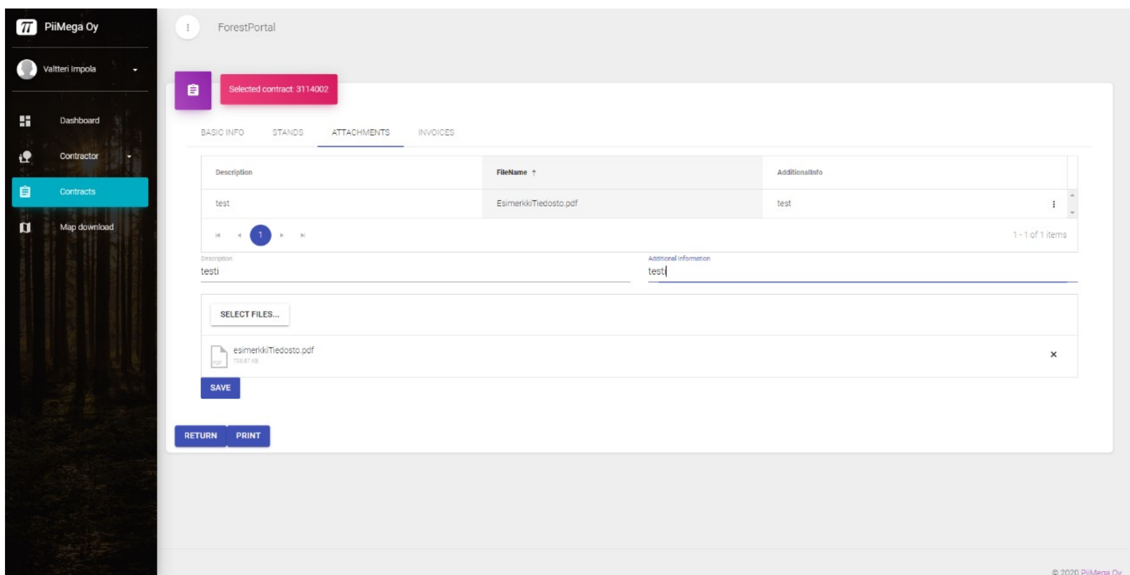
KUVA 11. Työohjeen tiedot

Kuvassa 12 kuvatussa attachments-näkymässä näytetään kaikki sopimukselle kuuluvat liitetiedostot. Sivulla voidaan lisätä uusia liitetiedostoja raahaamalla ne ”SELECT FILES”-kenttään, jolloin tiedostolle voidaan antaa nimi sekä mahdollinen kuvaus tiedostosta, kuten kuvassa 13 näkyy.



KUVA 12. Liitetiedostot-näkymä

Liitetiedostoja voidaan myös poistaa valitsemalla Kuvassa 13 näkyvästä Menu-painikkeesta poista-painike, jolloin liitetiedosto poistetaan sopimukselta.



KUVA 13. Lisää liitetiedosto -näkymä

4.6 Karttakomponentti

Sovelluksessa keskeisimpänä ominaisuutena on kuvassa 8 näkyvä karttakomponentti, jota käytetään havainnollistamaan kulloisenkin työmaan sijaintia ja tarjoamaan käyttäjälle paremman käsityksen itse työmaa-alueesta.

Kartan toteuttamisessa päätettiin käyttää Google Maps API:a, sillä se tarjoaa paljon jo valmiina olevia ominaisuuksia, kuten erilaisten ikonien lisäämisen kartalle ja mahdollisuuden piirtää erinäisiä kerroksia kartan päälle, kuten tämän sovelluksen tapauksessa teitä ja erilaisia työmaa-alueita, joista on esimerkkejä kuvassa 8. Google Maps API tarjoaa myös kattavan dokumentaation, joka helpottaa suuresti kartan kanssa työskentelyä.

Itse karttakomponentti toimii siten, että tietokantaan tallennetusta kuviotiedostosta haetaan back-endissä kaikki tarvittava tieto, kuten itse koordinaatin ja kuviolle annettu nimi. Tämä tiedosto muutetaan back-endissä bittitaulukoksi ja lähetetään API-kerrokseen, jossa data muutetaan Google Maps API:n tukemaan GeoJson muotoon, joka sitten palautetaan front-endille, jossa karttakomponentille annetaan parametriksi tämä GeoJson, jonka jälkeen komponentti käsittelee datan ja käyttää Google Maps API:n loadGeoJson-metodia, jolla halutut kuviot saadaan piirrettyä kartalle. Karttakomponentti on myös sidottu toimimaan yhdessä taulukon kanssa. Tämä auttaa havainnollistamaan dataa, koska karttakomponentissa voi olla nähtävissä useita kuviota yhtä aikaa, jolloin on helpompi hahmottaa, mitkä taulukon rivit kuuluvat millekin kuviolle kartalla. Nyt kartan kuviota painamalla voidaan myös valita sille kuuluvat rivit, jolloin rivit tulevat myös valituksi. Tämä toiminto toimii myös toisinpäin, taulukon riviä painamalla valitaan automaattisesti kartalta riviin linkitetty kuvio.

4.7 Datakerros

Projekissa käytettiin tietokantana PiiMega Oy:n konesalissa pyörivää jo aiemmin luotua SQL Serveriä. Projektin aikana käytettiin mahdollisuuksien mukaan jo olemassa olevia tauluja, joihin lisättiin tarpeen vaatiessa uusia kenttiä. Sovellus vaati myös uusien taulujen luomista, jotta halutut tiedot pystyttiin linkittämään ilman suuria muutoksia jo olemassa olevaan tietokantarakenteeseen.

Suurimmaksi ongelmaksi tietokannan luomisessa osoittautui jo olemassa olevien kartta-aineistotaulujen muokkaaminen, jotta ne voitiin linkittää alueen sijasta yksittäiselle kuviolle. Tämä muutos vaati uusien kenttien lisäämistä tauluihin, sekä jo olemassa olevien SQL-kyselyiden muokkaamista. Haastavuutta näihin muu-

toksiin toi myös se, että tauluja käytettiin myös muissa projekteissa. Tällöin jouduttiin tarkkaan pohtimaan muutoksien vaikutusta jo olemassa oleviin rakenteisiin. Kartta-aineiston datan luomiseenkin tuli tehdä muutoksia, jotta tarvittavat uudet tiedot saatiin datan mukana.

4.8 Logiikkakerros

Solvelluksen logiikkakerros koostuu kahdesta osasta, ensimmäinen osa on API-kerros, mikä koostuu HotChocolate GraphQL-serveristä ja toinen osa itse varsinainen back-end, mikä koostuu ASP.NET-kehikosta, WCF:stä ja IIS-palvelusta ja kielenä käytetään C# -ohjelmointikieltä

API-kerrosta käytetään yhdistämään front-end ja back-end toisiinsa, sekä välittämään dataa näiden välillä. API-kerroksessa määritellään front-endin käyttämät QraphQL queryt ja mutaatiot. Tässä osassa luodaan myös GraphQL-skeema, jota vasten kaikki front-endin tuottamat kyselyt validoidaan. Skeemassa on listattuna kaikki sallitut kyselyt ja mutaatiot. Front-endistä tulevat GraphQL-oliot mapataan back-endistä rajapinnan kautta tuotuihin olioihin. Tämän jälkeen API-kerros ottaa yhteyttä back-endin WCF-päätepisteeseen ja lähettää oliot back-endille. API-kerroksessa on myös oma bisneslogiikkansa, jota käytettään esimerkiksi back-endistä tulevan spatial datan muuttamiseen front-endissä olevan kartta-komponentin tarvitsemaan GeoJson-muotoon. Tämä toiminto suoritetaan API-kerroksessa eikä front-endissä sen takia, että muutkin sivut voivat helposti käyttää kyseistä funktiota ja samalla vältytään front-endin kasvamiselta liian suureksi ja näin hidastumiselta.

Back-endiä käytetään suorittamaan datan hakeminen tietokannasta ja sen käsittelemiseen API-kerroksen haluamaan muotoon. Back-endissä tunnistetaan sinne lähetetty olio ja se lähetään oikealle käsittelijä metodille. Käsittelijästä olio lähetään oikeaan businesslogiikkaan, jossa suoritetaan itse kysely. Kyselyn suorittamisen jälkeen WCF-palvelu palauttaa haetun datan API-kerrokselle, missä sille tehdään vielä tarvittavat muutokset, kuten laskentaa vaativat toimet. Tämän jälkeen data validoidaan taas skeemaa vastaan, jotta vain oikean muotoinen vastaus pääsee takaisin front-edille. Back-endinä päätettiin käyttää jo olemassa olevaa back-endiä, koska silloin päästiin käsiksi jo olemassa oleviin funktioihin ja

kyselyihin, jolloin niitä ei tarvinnut lähteä rakentamaan alusta asti. Tällöin saatiin hyödynnettyä työtä.

Logiikkakerroksen luomisessa suurimmaksi haasteeksi osoittautui SQL-kyselyiden muodostaminen, jotka kasvoivat varsin suuriksi, kun oikean datan saamiseksi linkitettävien taulujen määrä kasvoi.

4.9 Käyttöliittymä

Ohjelman käyttöliittymää lähdettiin rakentamaan osaksi jo olemassa olevaan ForestPoral web-applikaatiota. Käyttöliittymä on toteutettu React-kirjastoa käyttäen, jolloin se koostuu erilaisista komponenteista. Alussa komponentteja lähdettiin rakentamaan KendoReact-komponenttikirjaston päälle. Tällöin Kendon komponentit toimivat eräänlaisena pohjana, jota alettiin tarpeen mukaan laajentamaan haluttujen ominaisuuksien aikaansaamiseksi. Tämä toimi jonkin aikaa hyvin ja sillä saatiin aikaa toimivia näkymiä. Myöhemmässä vaiheessa komponentteja alettiin myös rakentamaan alusta asti itse, jolloin saatiin hallittua komponenttien rakennetta paremmin ja saavutettiin geneerisempiä komponentteja, jolloin niiden uudelleenkäyttäminen tuli helpommaksi.

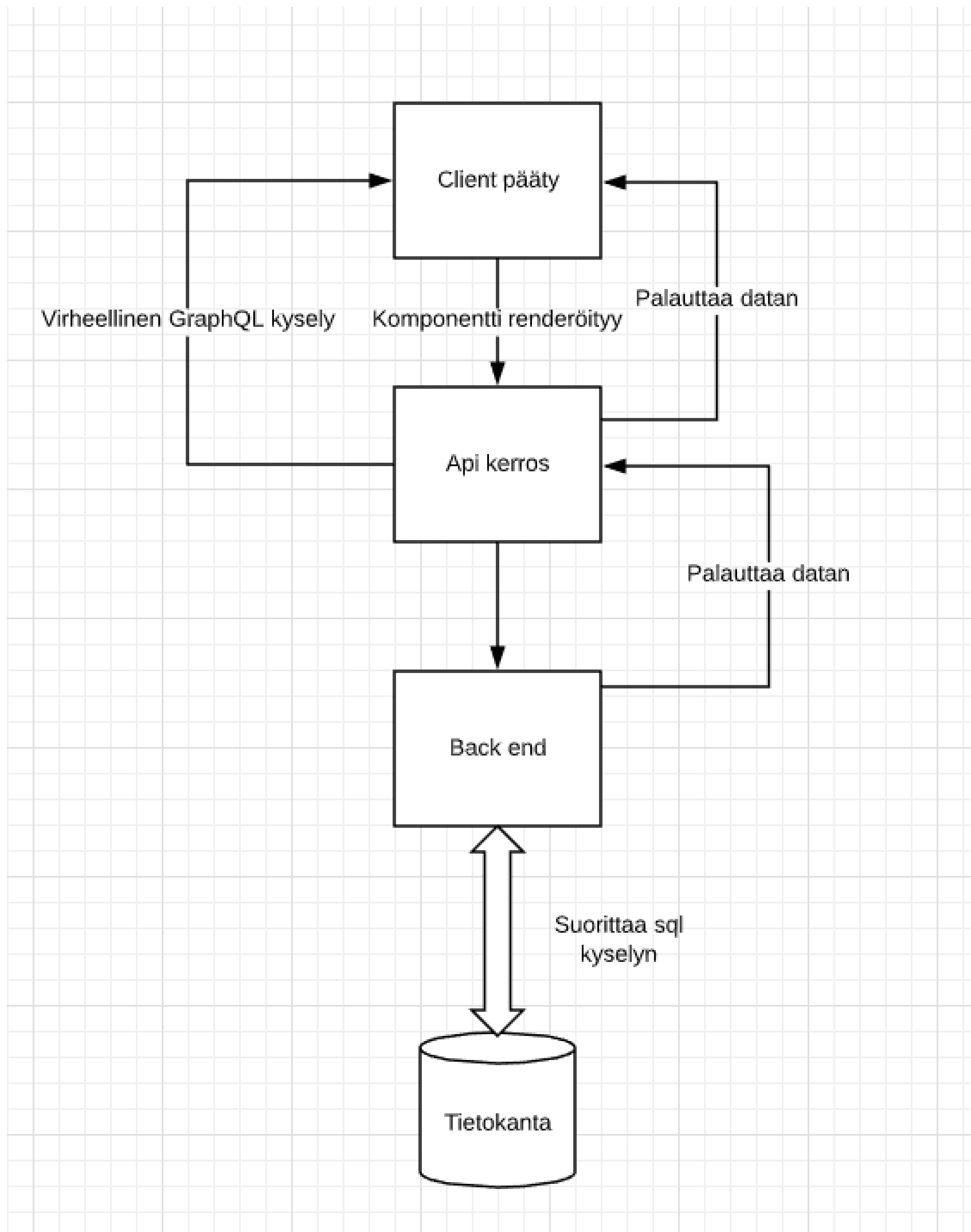
Käyttöliittymän tyylit tulevat kaikkiin näkymiin erikseen määritellyistä CSS-tiedostoista, joissa määritellään kaikki tyylilliset asiat, kuten minkä värisiä nappeja käytetään missäkin tilanteissa ja kuinka suuret marginaalit komponentilla on. Näkymien tyylittelyssä käytettiin pääosin Googlen Material Desing -periaatteita, jolloin saavutettiin erityisesti mobiililaitteilla ohjelmaa käytettäessä jo suurimmalle osalle käyttäjistä tuttuja elementtejä ja toimintoja, kuten kuvassa 12 näkyvä taulukossa käytetty Menu-painike, jota käytetään avaamaan erilaiset komponenttia koskevat toiminnot, kuten rivin poistamien tai uuden rivin lisääminen.

Material Desing -periaatteiden seuraaminen auttaa myös tekemään sivustosta luonnollisemman ja parantaa näin käyttäjäkokemusta. Komponenttien suunnittelussa on otettu huomioon, miten kunkin komponentin käytettävyyttä saataisiin mahdollisimman luonnolliseksi eli vältytään painelemasta useita painikkeita jonkin tietyn toiminnan saavuttamiseksi. Esimerkiksi kuvassa 5 olevassa sopimuksen lisäysnäkymässä sopimus avataan automaattisesti luomisen jälkeen, jolloin

vältytään turhilta painikkeiden painamisilta, koska voidaan olettaa, että käyttäjä haluaa avata luodun sopimuksen.

4.10 Datan hakeminen

Jokaisessa ohjelman näkymässä esitetään tietokannasta haettua dataa ja datan hakemiseen käytetään GraphQL API:a. Prosessi alkaa, kun jokin dataa hakeva komponentti piirtyy uudelleen, jolloin sen sisällä olevat Apollon useQuery-hookit laukeavat ja suorittavat niihin linkitetyt GraphQL-kyselyt, jolloin GraphQL-kysely etenee kohti API-kerrosta, jossa kysely tarkastetaan vertaamalla sitä skeemassa oleviin sallittuihin kyselyihin. Jos kysely menee läpi, API-kerros lähettää kyselyn eteenpäin back-endille, jossa suoritetaan itse kysely tietokannasta. Back-end palauttaa halutut tiedot API-kerrokseen, josta tiedot palautetaan takaisin client päätyyn, missä haettu data-olio tarkastetaan vertaamalla sitä client-päädyssä olevaan interfaceen. Jos data on oikean muotoista, se palautetaan kyselyn suorittaneelle komponentille ja esitetään halutulla tavalla käyttöliittymässä. Kuvassa 14 esitetään datan hakeminen kaaviona.



KUVA 14 Datan kulku ohjelmassa

4.11 Jatkokehitys

Jatkokehityksessä tätä opinnäytetyötä tullaan laajentamaan lisäämällä siihen uusia toiminnallisuuksia, kuten mahdollisuus piirtää kartalle, erilaisten raporttien tulostaminen ja rahaliikenteen lisääminen.

Karttakomponenttia tullaan laajentamaan siten, että käyttäjä pystyy piirtämään haluamiaan karttamerkkejä ja kuvioita, kuten teitä ja leimikoita kartalle. Käyttäjän tulisi myös itse pystyä muokkaamaan jo piirrettyjä kuvioita sekä lisäämään erilaisia käyttäjälle itselleen tärkeitä tietoja kartalle, kuten varoituksia jyrkänteistä ja alhaalla olevista sähköjohdoista. Käyttäjän tulisi myös itse pystyä määrittelemään, millä väreillä mikäkin karttamerkki piirrettäisiin. Kartan tulisi myös pystyä esittämään käyttäjän reitti työmaa-alueelle tämän nykyisestä sijainnista.

Sivustolta pitäisi myös pystyä tuottamaan erilaisia raportteja jo tehdyistä töistä kullekin lohkolle. Nämä raportit tuli pystyä tulostamaan ja näyttämään Excelissä tai PDF-muodossa. Nämä tulosteet helpottaisivat entisestään metsänhoidollisten töiden edistymisen seurantaa, koska silloin asiakkaalla olisi käytössään reaaliaikaiset raportit kustakin työmaasta.

Sivustolle tullaan myös lisäämään mahdollisuus tehdä laskuja jo tehdystä työstä. Kun urakoitsija on saanut jonkin työohjeen kuvaaman työn valmiiksi, ohjelma osaa tehdä siitä laskun ja lähettää sen laskutuspalveluun. Tämä helpottaisi varsinkin urakoitsijoiden työtä, koska heidän ei itse tarvitsisi tehdä laskuja vaan ohjelma hoitaisi sen automaattisesti.

Opinnäytetyönä syntynyttä ohjelmaa voidaan tulevaisuudessa hyödyntää käyttämällä käyttöliittymän komponentteja uudelleen tulevissa projekteissa, jolloin tulevien ohjelmistojen kehittäminen nopeutuu. Samojen komponenttien käyttäminen myös muissa yrityksen tuottamissa ohjelmistoissa parantaa asiakkaan käyttäjäkokemusta, jos asiakkaalla on käytössä useampia tuotteita, koska silloin tuotteiden keskeiset toiminnot, kuten rivien poisto ja lisääminen löytyvät samoista paikoista ja toimivat samoilla periaatteilla. Tällöin asiakkaan ei tarvitse opetella jokaisen tuotteen käyttämistä aivan alusta alkaen. Ehkä parhaiten hyödynnettä-

vissä on projektin aikana luotu karttakomponentti, jota tullaan käyttämään pohjana kaikissa tulevissa sivuissa, joissa on tarve esittää dataa kartalla. Komponenttia hyödynnetään jo sivun urakoitsijoille suunnatussa mobiiliversiossa. Projektin aikana syntyneitä back-end-funktioita voidaan myös hyödyntää tulevaisuudessa, sillä näitä funktioita käytetään jo osittain ForestPro-tuotteessa. Tietokantaan rakennettuja uusia tauluja voidaan myös hyödyntää tulevissa ohjelmistoprojekteissa. Opinnäytetyössä kerättyä tietoa voidaan hyödyntää erityisesti React-kirjastoa käyttävissä web-käyttöliittymissä, joissa käytetään hookeja kehittämiseen.

5 YHTEENVETO

Opinnäytetyön tavoitteena oli luoda web-pohjainen ohjelma, jolla käyttäjä pystyisi tekemään sopimuksia ja lisäämään näille sopimuksille kuvioita. Kuviolle tuli pystyä lisäämään työohjeita ja myyntinimikkeitä. Myyntinimikkeiden ja työohjeiden havainnollistamisessa tulisi myös käyttää karttaa, jolla pystyttäisiin esittämään niille kuuluvaa dataa. Sopimuksille tuli myös voida lisätä ja poistaa liitetiedostoja. Näiden toimintojen suorittaminen oli mahdollista PiiMegan ForestPro-työpöytäsovelluksella, mutta projektin tavoitteena oli mahdollistaa nämä toiminnot web-sovelluksessa, joka mahdollistaisi ohjelman käytön suoraan selaimesta, jolloin käyttäjä pystyy käyttämään ohjelmaa millä tahansa laitteella, jossa on selain ja nettiyhteys.

Tehtävänä ohjelman toteuttaminen oli haastava, sillä käytössä olevat tekniikat olivat uusia ja haasteellisia, jolloin niiden opettelussa meni aikaa. Myös eri tekniikoiden yhteensovittaminen toimivaksi kokonaisuudeksi oli haastavaa. Vaikein vaatimus oli yhdistää Google Maps API, karttadata ja erilaiset dataa havainnollistavat komponentit, kuten karttaan sidotut taulukot ja listat ja saada nämä komponentit toimimaan myös mobiililaitteissa. Tällöin vältyttiin ylimääräiseltä koodilta, kun samat komponentit toimivat ja skaalautuvat erikokoisiin näyttöihin.

Työn alkuvaiheessa luoduista komponenteista, joista ohjelman käyttöliittymä koostuu, olisi voinut tehdä vielä enemmän geneerisiä, jolloin ne todellakin olisivat olleet helposti uudelleen käytettäviä, niiden käyttäminen olisi ollut helpompaa ja niitä olisi voitu hyödyntää tehokkaammin tulevissa projekteissa. Tämä ongelma tosin korjaantui työn myöhäisemmässä vaiheessa, jolloin komponenttien luomisesta tuli entistä suunnitelmallisempaa, kun toisetkin projektit alkoivat käyttää samoja komponentteja ja oma taitotasoni ja ymmärrykseni komponenttien käytöstä kasvoi. Monikäyttöisten ja mahdollisimman geneeristen komponenttien luominen on toki työläämpää ja se vie enemmän aikaa kuin erittäin spesifin yhteen tarkoitukseen sopivan komponentin luominen, mutta opinnäytetyön aikana huomattiin, että tällaiset monikäyttöiset komponentit mahdollistavat nopeamman kehityksen tulevaisuudessa, jolloin pitkällä aikavälillä niiden tekeminen on kannattavampaa. Monikäyttöiset komponentit myös yhdenmukaistavat koodia, sillä silloin vältytään

tekemästä samankaltaisia komponentteja erikseen jokaiselle sivulle. Tällöin kehittäjien on helpompi ja nopeampi ymmärtää toistensa koodeja, koska samoja asioita tehdään vain yhdellä johdonmukaisella tavalla. Tämä myös keventää koodia, sillä siellä ei ole useita samaa asiaa tekeviä komponentteja.

Käytettävien työkalujen nopea kehittyminen toi myös oman haasteensa, sillä jo käytössä olevia menetelmiä jouduttiin muuttamaan, kun niiden tukeminen loppui uudemmassa Apollon versiossa. Tällöin jouduttiin luopumaan suurelta osin luokkapohjaisten komponenttien käyttämisestä ja muuttamaan jo olemassa olevat komponentit käyttämään hookeja, koska Apollon kyselyt siirtyivät käyttämään hookeja ja Apollo lopetti luokkakomponenttien tukemisen. Tosin React itsessäänkin on liikkumassa poispäin luokkapohjaisten komponenttien käytöstä ja näin ollen samat muutokset olisivat olleet edessä tulevaisuudessa. Näillä muutoksilla saavutettiin myös koodin yhdenmukaisuus tulevaisuudessa, jolloin vältytään saman asian tekemiseltä monilla eri tavoilla ja sen aiheuttamalta sekaannukselta.

Työssä saavutettu lopputulos on mielestäni onnistunut, sillä se täyttää sille asetetut vaatimukset. Ohjelman ensimmäisessä vaiheessa saatiin ohjelman käyttöliittymät valmiiksi, jolloin niiden avulla pystyttiin havainnollistamaan asiakkaalle ohjelman rakennetta ja kuinka ohjelmaa tultaisiin käyttämään. Tässä vaiheessa saatiin valmiiksi sopimustenlistaus-, perustiedot-, lohkot- ja liitetiedostot-näkymät. Projektin toisessa vaiheessa onnistuttiin pääsemään tavoitteeseen ja näin tuottamaan ohjelmasta versio, jota asiakas pystyi itse testaamaan omassa ympäristössään oikealla datalla. Tässä versiossa asiakas pystyi tekemään sopimuksen ja lisäämään sille lohkoja. Näille lohkoille pystyttiin, myös tekemään työohjeita ja myyntinimikkeitä. Myyntinimikkeitä pystyttiin, myös havainnollistamaan kartan avulla, jolla esitettiin leimikko, jolle kyseinen myyntinimike oli linkitetty. Toisessa vaiheessa keskityttiin data- ja logiikkakerrosten rakentamiseen. Projektin molemmat vaiheet pysyivät myös suunnitelluissa aikatauluissaan, joista oli sovittu asiakkaan kanssa.

LÄHTEET

1. Turunen, Saana 2017. Design termistö tutuksi: Näin UI- UX ja visuaalinen suunnittelu eroavat toisistaan. Blogi. Saatavissa: https://lamia.fi/blog/design-termisto-tutuksi. Hakupäivä 11.3.2020.

2. Salovaara, Ilmari 2011. Käyttöliittymäsuunnitelu. Opinnäytetyö. Forssa: Hämeen ammattikorkeakoulu, Tietotekniikan koulutusohjelma. Saatavissa: http://urn.fi/URN:NBN:fi:amk-201105168101. Hakupäivä 22.3.2020.

3. Haukipuro, Henri 2020. Urheilukellon käyttöliittymän mallinnustyökalu. Opinnäytetyö. Oulu: Oulun ammattikorkeakoulu, Tietotekniikan tutkinto-ohjelma. Luettavissa Oamkin sisäverkossa. Hakupäivä 20.3.2020.

4. HowToGraphQL (Fundamentals) – Introduction (1/4). HowToGraphQL. Saatavissa: https://www.howtographql.com/basics/0-introduction/. Hakupäivä 7.12.2019.

5. Queryes and Mutations. 2020. GrapQL Foundation. Saatavissa: https://graphql.org/learn/queries/. Hakupäivä 1.12.2019.

6. Introduction to GraphQl. 2020. GrapQL Foundation Saatavissa: https://graphql.org/learn/ . Hakupäivä 2.12.2019.

7. Schemas and Types. 2020. GraphQL Foundation. Saatavissa: https://graphql.org/learn/schema/. Hakupäivä 20.2.2020.

8. Core Consepts. HowToGraphQL Saatavissa: https://www.howtographql.com/basics/2-core-concepts/. Hakupäivä 8.12.2019.

9. Lerner, Ari 2018. What is React?. Blogi Saatavissa: https://www.fullstackreact.com/30-days-of-react/day-1/. Hakupäivä 3.12.2019.

10. React Lifecycle Methods – A Deep Dive. 2018. Programming with Mosh. Saatavissa: https://programmingwithmosh.com/javascript/react-lifecycle-methods/. Hakupäivä 20.2.2020-

11. Introduction to the DOM. 2020. MDN Contributors. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. Hakupäivä 8.3.2020.

12. Hamedani, Mosh 2018. React Virtual DOM Explained in Simple English. Programming with Mosh. Saatavissa: https://programmingwithmosh.com/react/react-virtual-dom-explained/ . Hakupäivä 13.12.2019

13. Reconciliation. 2020. React. Saatavissa: https://reactjs.org/docs/reconciliation.html. Hakupäivä 17.3.2020.

14. Eisenman, Bonnie 2020.Learning React Native by Bonnie Eisenman. O'Reilly Media. Saatavissa: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch02.html. Hakupäivä 17.3.2020.

15. Introducing Hooks. 2020. React. Saatavissa: https://reactjs.org/docs/hooks-intro.html. Hakupäivä 31.3.2020.

16. Using the State Hook. 2020. React. Saatavissa: https://reactjs.org/docs/hooks-state.html. Hakupäivä 31.3.2020.

17. Using the Effect Hook. 2020. React. Saatavissa: https://reactjs.org/docs/hooks-effect.html. Hakupäivä 22.2.2020.

18. Hooks API Reference. 2020. React. Saatavissa: https://reactjs.org/docs/hooks-reference.html#usecontext. Hakupäivä 28.1.2020.

19. Why Apollo Client. 2020. Apollo. Saatavissa: https://www.apollographql.com/docs/react/why-apollo/. Hakupäivä 17.3.2020.

20. Queries. 2020. Apollo. Saatavissa: https://www.apollographql.com/docs/react/data/queries/. Hakupäivä 17.3.2020.

21. Mutations. 2020. Apollo. Saatavissa: https://www.apollographql.com/docs/react/data/mutations/. Hakupäivä 17.3.2020.

22. Local state management. 2020. Apollo. Saatavissa: https://www.apollographql.com/docs/react/data/local-state/#managing-the-cache. Hakupäivä 17.3.2020.

23. Overview. 2020. Google. Saatavissa: https://developers.google.com/maps/documentation/javascript/tutorial. Hakupäivä 26.3.2020.

24. IIS. 2013. Tech Terms. Saatavissa: https://techterms.com/definition/iis. Hakupäivä 26.3.2020.

25. What Is Windows Communication Foundation. 2020. Microsoft. Saatavissa: https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf. Hakupäivä 21.3.2020.

26. Using Data Contracts. 2017. Microsoft. Saatavissa: https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/using-data-contracts. Hakupäivä 25.3.2020.

27. Endpoints: Address, Bindings, and Contracts. 2017. Microsoft. Saatavissa: https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/endpoints-addresses-bindings-and-contracts. Hakupäivä: 30.3.2020.

28. Staib, Michael 2019. Introduction. Hot Chocolate. Saatavissa: https://hotchocolate.io/docs/introduction. Hakupäivä 31.3.2020.