Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Robert Ruigendijk

React-sovelluksen lokalisointi ja kustomointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

20.11.2020

Tekijä Otsikko Sivumäärä Aika	Robert Ruigendijk React-sovelluksen lokalisointi ja kustomointi 37 sivua 20.11.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaajat	Tuotepäällikkö Mika Koivupalo Lehtori Antti Laiho

Insinöörityön tarkoituksena oli tutkia kolmea eri lokalisoinnin toteutustapaa sähköiselle allekirjoituspalvelulle. Näistä kolmesta toteutustavasta oli tavoitteena löytää parhaiten soveltuva tapa kyseiselle palvelulle ja toteuttaa lokalisointi niin, että palvelu toimii englanniksi, suomeksi ja ruotsiksi. Tavoitteena oli myös toteuttaa palveluun kustomointi, eli toiminto, joka muuttaa palvelun käyttöliittymän visuaalisen näkymän eri yrityksien käyttäjille. Työ tehtiin yritykselle, joka on yksi maailman johtavista IT-palveluiden tarjoajista.

Sähköinen allekirjoituspalvelu oli toteutettu JavaScriptillä käyttäen React-ohjelmistokehystä, joten lokalisoinnin toteutustapojen piti tukea Reactia.

Insinöörityössä onnistuttiin löytämään hyvin tietoa eri lokalisointitavoista ja oikea toteutustapa löydetyn tiedon perusteella. Toteutustavoista tutkittiin niiden ominaisuuksia ja käyttötapoja esimerkkien avulla. Näin niistä huomattiin, kuinka paljon toiminnallisuuksia toteutustavoissa oli ja kuinka helppoa niitä oli käyttää. Tutkittiin myös niiden vaikutusta olemassa olevan koodin ”kauneuteen” ja helppolukuisuuteen.

Insinöörityön lopputuloksena syntyi toimiva toteutus palveluun sekä lokalisoinnin että kustomoinnin osalta: palvelua on mahdollista käyttää englannin-, suomen- ja ruotsinkielisenä, ja palvelun käyttöliittymä näyttää erilaiselta riippuen siitä, minkä yrityksen edustaja käyttäjä on.

Avainsanat	lokalisointi, kustomointi, React

Author Title Number of Pages Date	Robert Ruigendijk Localization and customization of a React application 37 pages 20 November 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Game Applications
Instructors	Mika Koivupalo, Product owner Antti Laiho, Senior lecturer

The purpose of this thesis was to find information on three different ways to implement localization to an electronic signature service, choose the best implementation to the service and implement it into the service in such a way that it is possible to use it in English, Finnish and Swedish. Another goal was to implement customization to the service to make the service's user interface look different depending on the user's company.

The electronic signature service was implemented with JavaScript using the React-framework. In other words the three different ways to implement localization also had to support React.

Plenty of information on the different methods of localization was found during this thesis and also the best method based on the information. These methods' properties and ways of usage were examined by using examples. By doing this it was possible to determine how many functions the methods have and how easy they were to use. Special attention was also given to the effects of the methods on previously written code's "beauty" and readability.

As a result of this final year project, a working solution was made to the service: the service now works in English, Finnish and Swedish and the visual look of the user interface will change depending on the user company.

Keywords	localization, customization, React

Sisällys

Lyhenteet

SPA	Single-page application. Verkkosovellus, joka toimii koko ajan samalla html-sivulla.
API	Application programming interface. Ohjelmointirajapinta.
DOM	Document Object Model. Ohjelmointirajapinta HTML- ja XML-dokumenteille.
I18N	Internationalization. Kansainvälistäminen. Lyhenne tulee siitä, että sanan ensimmäisen ja viimeisen kirjaimen välissä on 18 kirjainta.
HOC	Higher Order Component. Edistynyt React-komponentti.

1 Johdanto

Insinöörityön tarkoituksena oli toteuttaa lokalisointi ja kustomointi Fujitsu Finland Oy:n kehitteillä olevaan sähköiseen allekirjoituspalveluun, joka rakennettiin React-sovelluksena. Lokalisoinnilla tarkoitetaan ohjelmistokehityksen yhteydessä tuotteen muuttamista niin, että se vastaa muun muassa kohdeyleisön kielellisiä ja kulttuurisia vaatimuksia. Lokalisointi koostuu käännöstyöstä ja sovelluksen ohjelmoimisesta niin, että se osaa näyttää käännetyt tekstit. Tässä insinöörityössä perehdyttiin vain ohjelmointiin, koska käännöstyö ulkoistettiin ammattilaisille.

Lokalisointi on kasvanut viime vuosikymmenten aikana miljardien arvoisiksi markkinoiksi maailman globalisaation seurauksena. Lokalisoimalla ja kustomoimalla tuotteita maksimoidaan alue, jossa tuotetta voidaan myydä. Nostamalla potentiaalisten asiakkaiden määrää saadaan itse tuottoakin nostettua.

Insinöörityön tavoitteena oli löytää tietoa kolmesta eri lokalisoinnin toteutustavasta, esittää niiden vahvuudet ja heikkoudet esimerkkien avulla, valita parhaiten soveltuva toteutustapa kyseiselle palvelulle ja toteuttaa se palveluun niin, että palvelu toimii englannin-, suomen- ja ruotsinkielisenä. Lisäksi tavoitteena oli toteuttaa kustomointi palveluun niin, että palvelu saadaan näkymään erilaisena riippuen siitä, mistä yrityksestä palvelun käyttäjä on.

Fujitsu on yksi maailman johtavista IT-palveluiden tuottajista, ja varsinkin korona-aikaan on sähköiselle allekirjoitusjärjestelmälle kysyntää yrityksen sisällä ja sen ulkopuolella. Palvelun kehittää Fujitsu Finland Oy, ja ottaen huomioon yrityksen työntekijät ja asiakkaat olisi suurin osa palvelun käyttäjistä joko englantia, suomea tai ruotsia puhuvia henkilöitä. Tämän takia oli tärkeää, että palvelu lokalisoitiin näille kielille. Lisäksi oli tärkeää, että palvelu saatiin kustomoinnin avulla näkymään erilaisena riippuen siitä, minkä yrityksen työntekijä käyttäjä on, jotta palvelu näkyy käyttäjille heidän edustamiensa yrityksien brändistandardien mukaisesti.

2 Verkkosovelluksen globalisointi

1990-luvun alusta lähtien termi lokalisointi on liitetty varsinkin sovelluksien, tuotedokumentaatioiden ja sähköisten markkinoiden alueilla menestyksekkäimpiin yrityksiin. On yleinen harhakäsitys, että lokalisointi tarkoittaa vain jonkin kielen kääntämistä halutulle kielelle. Kääntäminen on vain osa lokalisointia. Jos ajatellaan nykyisiä verkkosivustoja, myös lokalisointi on osa isompaa aihealuetta, globalisointia, ja globalisointi pitää sisällään sekä lokalisoinnin että kansainvälistämisen (engl. internationalization). (1, s. 30.)

Globalisointi voidaan tietenkin myös määritellä eri lailla, riippuen siitä, missä kontekstissa siitä puhutaan. Jos globalisoinnista puhutaan yleisesti, on se melko hankalaa pukea sanoiksi niin, että se kattaisi kaiken mitä siihen oikeasti kuuluu, varsinkin siitä syystä, että termi on todella monimutkainen ja kehittyy koko ajan. Jos se olisi pakko pukea sanoiksi, voitaisiin sanoa, että globalisointi on kulttuurien, teknologioiden, markkinoiden ja hallintojen globaalia integraatiota. (2, s. 134.)

Yhden ehdotuksen mukaan globalisointia voitaisiin mitata globalisointi-indeksillä, jolla mitattaisiin, kuinka yhdistynyt jokin asia globaalisti on. Tämä indeksi koostuu viidestä eri osa-alueesta: taloudellinen integraatio, poliittinen sitoumus, tekninen yhteys, henkilökohtaiset kontaktit ja elämänlaatu. (2, s. 134.) Tarkastellaan tarkemmin, mitä nämä käsitteet tarkoittavat:

- Taloudellinen integraatio: alueiden välinen kaupankäynti.
- Poliittinen sitoumus: niiden instituutioiden esiintyminen ja vahvistuminen, jotka edistävät ja hallinnoivat globalisointia.
- Tekninen yhteys: kyky yhdistää kuluttajia teknologioiden, kuten internetin avulla.
- Henkilökohtaiset kontaktit: ihmiskontaktien määrä, joka ihmisillä kasvaa matkustamisen ja teknologioiden, kuten puhelimen ja internetin, seurauksena.
- Elämänlaatu: elinajanodotteen kasvaminen paremman ja helpommin saatavilla olevan terveydenhuollon, koulutuksen ja ravinnon seurauksena. (2, s. 134.)

Kuten tästä nähdään, kattaa globalisointi kulttuurisia, teknisiä, taloudellisia, poliittisia ja maantieteellisiä käsitteitä, joten ei ole ihme, että näin monimutkaista ja laajaa käsitettä on vaikea määritellä.

Kun globalisointia tarkastellaan ohjelmistokehityksen kannalta, ei se onneksi kata näin laajaa kokonaisuutta. Se ei kuitenkaan tarkoita sitä, etteikö se olisi monimutkainen kokonaisuus. Verkkosivun globalisointi pitää sisällään kaiken sen, että sitä pystyy myymään maailmanlaajuisesti, ja se koostuu verkkosivun lokalisoinnista ja verkkosivun kansainvälistämisestä. Jotta tämä on mahdollista, pitää yrityksellä olla kyky tehdä seuraavat asiat:

- tutkia, mitä vaatimuksia globaaleilla verkkosivuilla on.
- luoda sellainen teknillinen ja rakenteellinen infrastruktuuri, jolla lokalisointi ja kansainvälistäminen on mahdollista.
- integroida käytössä olevat kansainväliset sovellukset ja palvelut omaan sovellukseen.
- luoda prosessit, jotka hallinnoivat kansainvälistettyjen ja lokalisoitujen verkkosivujen laatua, tukea ja markkinointia (2, s. 135).

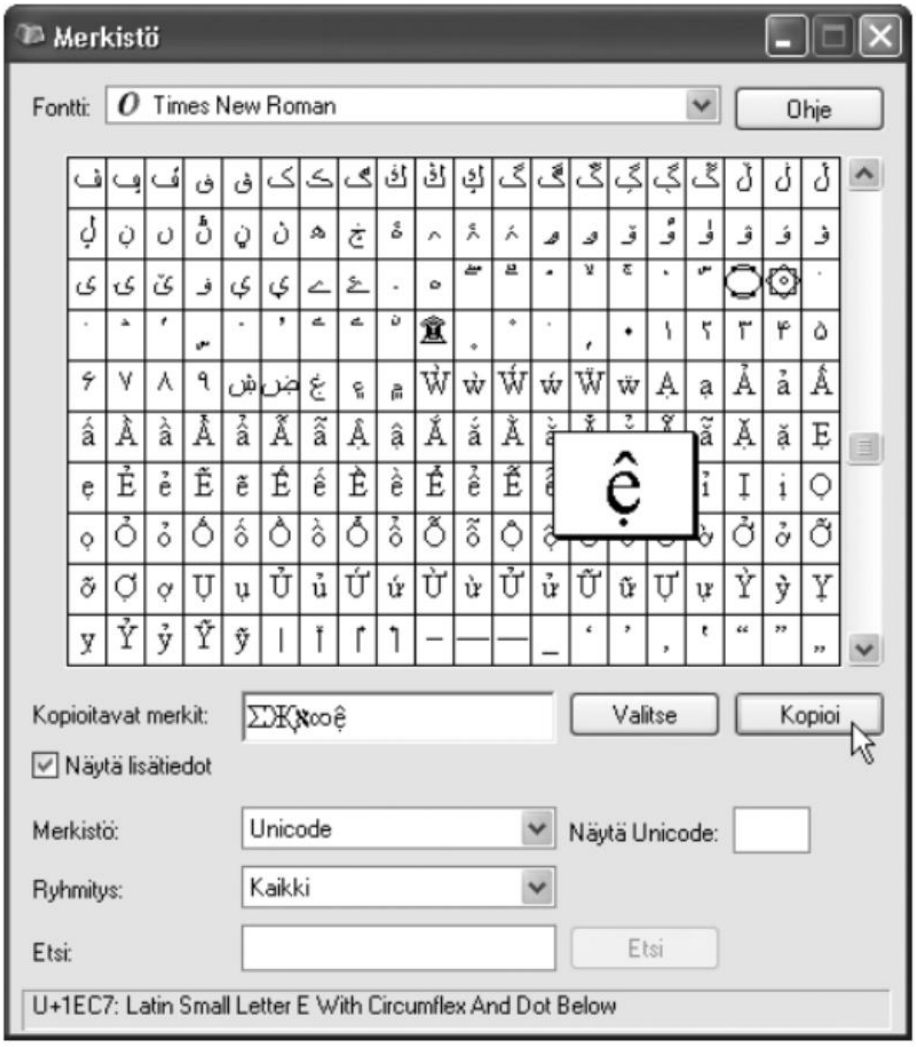
2.1 Verkkosovelluksen kansainvälistäminen

Verkkosovelluksen kansainvälistämisen voi laajasti määritellä tuotteen yleistämiseksi niin, että se tukee useita kieliä ja kulttuurisia tapoja merkitä asioita. Jotta tuote tukee jotakin kieltä, sen pitää pystyä koodaamaan kielen kirjaimet niin, että tietokone ymmärtää ne. Nykyään kirjaimien ja merkkien koodaamiseen lokalisointia varten käytetään Unicode-merkistöstandardia, mutta esimerkiksi silloin, kun tietokonejärjestelmät olivat alkutekijöissään ja englanti oli oikeastaan ainoa kieli, mitä käytettiin tietokoneiden kanssa, oli 7-bittinen ASCII-koodausjärjestelmä, joka tuki sataakahtakymmentäkahdeksaa merkkiä, riittävä sen ajan vaatimuksille Kun IBM aloitti tuotteidensa kansainvälisen jakelun, sen oli kehitettävä 8-bittinen koodausjärjestelmä kaikkien romaanisten kielten aksentteja ja merkkejä varten. Tämä järjestelmä tuki kahtasataaviittäkymmentäkuutta merkkiä. Tämä ei kuitenkaan sallinut useamman kuin kahden kielikirjaimiston käyttöä samaan aikaan yhdessä järjestelmässä. (3, s. 31.)

Tämän ongelman ratkaisemiseksi kehitettiin 1990-luvun alkupuolella Unicode-merkistöstandardi. Nykyajan tietokoneet käyttävät sisäisesti Unicodea, mutta käyttäjät ja ohjelmat saattavat silti käyttää vielä aikaisempia merkistökoodausjärjestelmiä. Merkin käsite on yksi vaikeimmista peruskäsitteistä informaatioteknologiassa, mutta todella tärkeä tekstin

prosessointiin, tietokantoihin, lokalisointiin ja muihin IT-alan alueisiin (4, preface). Tietokoneet toimivat numeroilla, mika tarkoittaa sitä, että merkit, kuten kirjaimet, pitää pystyä koodaamaan niin, että tietokone ymmärtää ne.

Kaikkien maailman kielien kirjaimet ja merkit eivät mahtuisi mitenkään 8-bittisten järjestelmien 256 merkkiin, joten ratkaisuna bittien määrä nostettiin 16 bittiin. Näin ollen se tukee 65 536 numeroa, ja näin jokaiselle kirjaimelle ja merkille voidaan määrittää oma uniikki numero. Käyttäjät voivat siis saada minkä vain merkin syöttämällä vain sen merkin uniikin numeron, ja kaikki järjestelmät, jotka käyttävät Unicodea, ymmärtävät, minkä merkin käyttäjä haluaa (kuva 1).



Kuva 1. Unicode-merkistö, jonka avulla voi syöttää minkä vain merkin (3, s. 6).

Nykyajan yleisesti käytetyt ohjelmat, kuten Windows, MacOS ja Linux, ovat tukeneet Unicodea jo pitkän aikaa, mutta on myös pidettävä huolta, että niissä käytetyissä ohjelmissa ja niihin liittyvissä komponenteissa Unicode on aktivoitu. Vaikka esimerkiksi Windows tuntee Unicodea, kaikki sille alustalle tehdyt ohjelmat eivät. Lisäksi fontit tuottavat ongelmia; kaikki fontit eivät tue kaikkia Unicoden merkkejä. Tämä on menossa koko ajan parempaan suuntaan, kun fontteja kehitetään, mutta siitä on hyvä olla tietoinen.

Jotain ohjelmaa lokalisoidessa on siis hyvä pitää huolta, että kaikki ohjelmaa käyttävät komponentit tukevat Unicodea. Jos näin ei ole, se tuottaa ison haasteen, sillä eri merkkien koodausstandardit eivät ole keskenään yhteensopivia. Jonkin koodausstandardin koodi jollekin merkille voi tarkoittaa jotain ihan muuta merkkiä toisessa standardissa. Fujitsun sähköisen allekirjoitusjärjestelmän kaikki komponentit onneksi jo tukivat Unicodea, joten palvelun kansainvälistämisestä ei tarvinnut huolehtia.

2.2 Verkkosovelluksen lokalisointi

Jotta yritys pystyy myymään tuotteitaan nykyajan globaaleilla markkinoilla, on sen pidettävä huolta, että myydyt tuotteet täyttävät kulttuuriset, kielelliset ja muut vaatimukset niin, että käyttäjät maailmanlaajuisesti pystyvät käyttämään niitä. Kun on pidetty huolta, että ohjelma on kansainvälistetty niin, että se pystyy tukemaan lokalisointia, voidaan itse lokalisointi aloittaa. Lokalisoinnilla tarkoitetaan ohjelman muuttamista niin, että se täyttää kohdemarkkinan kielelliset, kulttuuriset ja tekniset vaatimukset. Kun jotain tuotetta lokalisoidaan, on otettava huomioon kohdevaltion yleiset tavat käyttää aikaa, päivämääriä, valuuttoja, mittayksiköitä, merkkejä, osoitteita, puhelinnumeroita, kieltä ja juridisia asioita (9, s. 137). Lokalisointi tulee sanasta ”locale”, joka tarkoittaa jotain tiettyä aluetta tai ympäristöä. Valtioista ei puhuta, koska valtiossa voidaan puhua useita eri kieliä. Tätä sanaa käytetään nykyään pääosin vain teknisessä kontekstissa, jossa sana edustaa jonkin alueen kieltä tai kieliä, sen ympäristöä ja merkkien koodausta. Esimerkiksi Suomessa puhuttu ruotsi on eri ”locale” kuin Ruotsissa puhuttu ruotsi. (6, s. 1.)

Lokalisointi on viime vuosikymmenten aikana kasvanut pienestä käännöksiin liitetystä ammatista omaksi teollisuudenalaksi, jonka arvo on miljardeja dollareja. Näin ollen se on mennyt siihen pisteeseen, että se oletusarvoisesti aina toteutetaan varsinkin isoihin ja

kansainvälisiin ohjelmistoihin. Tuotetta on paljon helpompi myydä, jos sitä pystytään tarjoamaan mahdollisimman useilla markkinoilla. Toinen syy lokalisoinnin yleistymiselle ovat lainsäädäntöön liittyvät vaatimukset. Usein paikalliset lait vaativat, että myydyn tuotteen mukana tulee ohjekirja paikallisella kielellä, ja joissain maissa tuotetta ei saa edes myydä, ellei se ole paikallisella kielellä (5, s. 5).

2.3 Lokalisoinnin haasteet

Lokalisoinnin ohjelmistokehityksessä voi jakaa kahteen osaan: ohjelmistokehitys, eli se, miten ohjelmisto koodataan, jotta se lokalisoi ohjelman sisällön, ja itse käännöstyö. Käännöstyö kannattaa ulkoistaa ammattilaisille, koska silloin vältytään käännösvirheiltä. Käännösvirheet voivat aiheuttaa sen, että yritys vaikuttaa epäammattilaiselta ja saattaa myös joissain tapauksissa loukata paikallisia käyttäjiä (6). Samalla ohjelmistokehittäjien aikaa ei kulu työhön, joka ei ole heidän vahvuutensa. Myös tässä insinöörityössä käännöstyö ulkoistettiin ammattilaisille ja keskityttiin vain ohjelmistokehitykseen.

Monikkosäännöt ja -muodot (engl. pluralization)

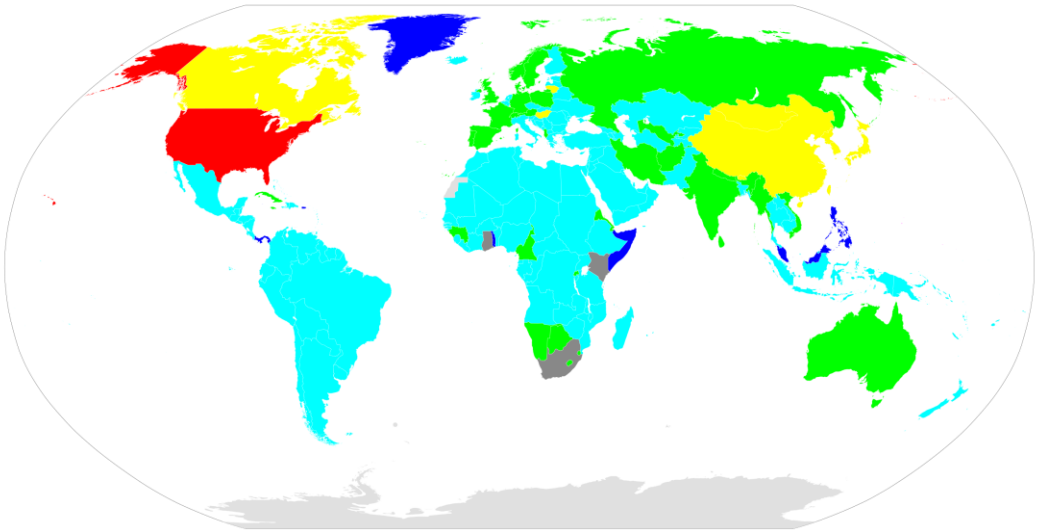
Eri kielillä on erilaiset säännöt, jotka määräävät, miten sanat esitetään riippuen siitä, kuinka monesta asiasta puhutaan (7). Tässä insinöörityössä isossa osassa oli sana dokumentti. Riippuen siitä, kuinka monesta dokumentista puhutaan, kirjoitetaan se eri tavalla: yksi dokumentti tai kaksi dokumenttia. Voisi helposti ajatella, että tämä on vain yksinkertainen ongelma: lisätään vain sanan perään pääte.

Kaikki kielet eivät kuitenkaan käsittele monikkoja tällä tavalla: esimerkiksi japanissa ja kiinassa ei ole monikkoja ollenkaan ja joillain arabikielillä on jopa kuusi monikkomuotoa (8). Useimmilla kielillä on myös poikkeuksia, kuten englannin kielessä: jos sana loppuu muotoon -s, -sh, -ch, -x tai -z, ei sanan perään lisätäkään vain s-kirjainta vaan -es (9). Lisäksi on otettava huomioon, että vaikka joillain kielillä olisi sama määrä monikkomuotoja, voivat säännöt olla erilaiset. Esimerkiksi englannin kielessä käytetään sanan yksikkömuotoa, jos määrä on yksi, mutta esimerkiksi ranskassa yksikkömuotoa käytetään yhden lisäksi myös nollan kohdalla. Monikkomuotoja voi olla kielen mukaan nollasta kuuteen, mutta eri sääntöjä on yhteensä 19 (7).

Yksi tapa ratkaista näitä ongelmia olisi käyttää Unicoden kehittämää Common Locale Data Repositorya (CLDR), joka pitää sisällään todella suuren määrän dataa lokalisointia varten. Data sisältää eri kielille tietoa luku-, aika- ja määräformatointiin sekä monikkosääntöihin ja -muotoihin liittyen. Monet yritysjätit, kuten Apple, Google ja Microsoft käyttävät tietokantaa omien tuotteidensa lokalisointiin. (8.)

Päivämääräformaatit

Lainsäädäntöön liittyvät ja kulttuuriset odotukset siitä, miten aika ja päivämäärät esitetään, vaihtelevat valtioiden välillä (kuva 2), ja tämä pitää ottaa huomioon, kun toteutetaan lokalisointia. Yleisin tapa esittää päivämäärä on DMY, missä D on päivä, M on kuukausi ja Y on vuosi. Muualla Euroopassa on käytössä myös muoto YMD, mutta esimerkiksi Yhdysvalloissa on yleisin tapa MDY (10). On myös hyvä tietää, että vaikka alueilla olisi yleinen päivämääräformaatti sama, voivat niissä käytetyt merkit vaihdella. Joillain alueilla päivät, kuukaudet ja vuodet erotetaan pisteillä, mutta joillain alueilla ne erotetaan vinoviivalla.



Kuva 2. Valtioiden käyttämät päivämääräformaatit. Turkoosi DMY, keltainen YMD. vihreä DMY ja YMD, sininen DMY ja MDY, punainen MDY ja YMD, harmaa MDY, YMD ja DMY (10).

Päivämääräformaattien lokalisointi oli jo toteutettu sähköiseen allekirjoitusjärjestelmään käyttämällä JavaScriptin ECMAScript Internationalization API-rajapintaa, joten myöskään tästä asiasta ei tässä insinöörityössä tarvinnut pitää huolta.

3 React-ohjelmistokehys

React (https://reactjs.org) on JavaScript-kirjasto käyttöliittymien rakentamiseen. Se on Facebookin kehittämä ja käytössä useissa suurissa palveluissa, kuten Facebookissa, Instagramissa ja Netflixissä (11). React on laajalti käytetty ohjelmistokehys verkkosivujen ja -sovelluksien rakentamiseen. Reactin komponentteja käyttämällä on helppo luoda monimutkaisiakin käyttöliittymiä.

Yleensä verkkosivu koostuu useista sivuista. Käytetään esimerkkinä vaikka jonkin urheiluseuran kotisivuja; tämäntapaiselta verkkosivulta voisi löytyä

- pääsivu
- sivu, jossa esitellään pelaajat
- sivu, jossa luetellaan otteluiden tulokset ja tulevat ottelut
- sivu hakutuloksille
- galleriasivu
- muita sivuja.

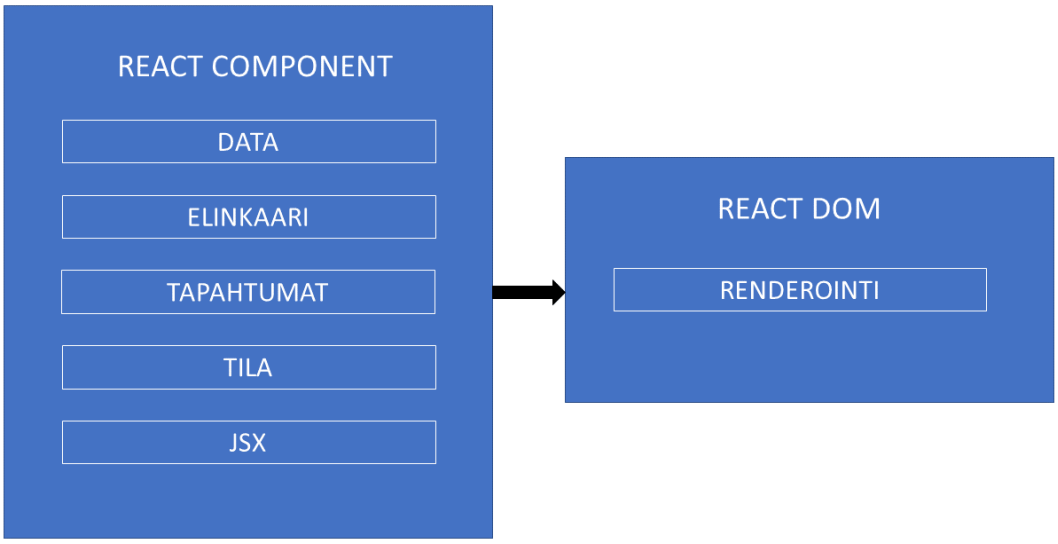
Jos tämäntapainen sivu olisi pitänyt luoda noin kymmenen vuotta sitten, olisi luultavasti käytetty toteutustapaa, jossa jokainen sivu luodaan erikseen, ja aina kun sivulla painettaisiin jotain linkkiä, veisi ohjelma käyttäjän kokonaan uudelle sivulle. Tämä ei luo hyvää käyttäjäkokemusta, koska käyttäjä näkee, kun sivu latautuu kokonaan uudestaan. Nykyajan sovellukset käyttävät Single-Page Application -mallia (SPA). Käyttäjä pysyy koko ajan samalla sivulla, mutta sivun data muuttuu käyttäjän toimintojen seurauksena. Tämä mahdollistaa erilaisten visuaalisten tekniikoiden käyttämisen, joiden ansiosta muutokset sivulla vaikuttavat sulavammilta ja joiden avulla saadaan käyttäjälle paljon parempi käyttäjäkokemus. Yleisesti SPA-sivun käyttöliittymän tilan hallinnointi on vaikeaa, mutta React helpompaa. (12, Introducing React.)

Reactin pääpiirteinen idea on antaa käyttäjille tapa luoda jokaiselle sovelluksen toiminnallisuudelle oma komponentti ja rakentaa käyttöliittymä kokonaan näistä komponenteista. Koko käyttöliittymä olisi siis komponentti, joka pitää sisällään pienempiä komponentteja. React rohkaisee komponenttien luomista, varsinkin sellaisten, joita voi käyttää uudestaan ja useissa eri tilanteissa. (12.)

3.1 Reactin ominaisuudet

React koostuu kahdesta ohjelmointirajapinnasta: React DOM ja React Component API (kuva 3). React DOM vastaa renderoinnistä, eli siitä, mitä itse sivulla näkyy. React Componentissa on sivun osat, jotka React DOM renderoi sivulle:

- Data: se data, joka komponentille syötetään. Tätä dataa kutsutaan propseiksi (tulee sanasta properties).
- Elinkaari (engl. lifecycle): komponentin koodi, joka reagoi komponentin elinkaaren muutoksiin. Jokin koodi voi esimerkiksi suoriutua aina, kun komponentti päivittyy.
- Tapahtumat (engl events): komponentin koodi, joka reagoi käyttäjän toimintoihin, kuten hiiren painalluksiin.
- Tila (engl. state): data, jolla hallitaan käyttöliittymän päivitystä. Aina kun data muuttuu, käyttöliittymä päivittyy.
- JSX: syntaksi, jota käytetään React-komponenteissa, joissa kuvaillaan käyttöliittymän osia. (13, Simplicity is good.)



Kuva 3. Reactin ohjelmointirajapinnat (13, Simplicity is good).

React siis perustuu komponentteihin, joista jokainen hallinnoi omaa tilaa ja näkymää. Rakentamalla ja yhdistelemällä näitä komponentteja voidaan luoda monimutkaisia käyttöliittymiä. Reactissa voi luoda kahdenlaisia komponentteja: funktiokomponentteja (engl.

function component) ja luokkakomponentteja (engl. class component). Funktiokomponenteille voi antaa propseja antamalla ne parametrinä, ja komponentti palauttaa käyttöliittymän. Luokkakomponenteilla on render()-funktio, joka palauttaa käyttöliittymän. Käyttöliittymät määritetään kummassakin komponentissa JSX-syntaksin avulla.

React käyttää myös niin sanottua deklaratiivista (engl. declarative) koodia (esimerkkikoodi 1), joka on imperatiivisen (engl. imperative) koodin (esimerkkikoodi 2) vastakohta. Tämän avulla voidaan Reactille kertoa, mitä sen halutaan tekevän sen sijaan, että kerrottaisiin sille tarkkaan, mitä tehdään ja miten jokin asia tehdään. (11, Principles of React.)

```
const userInput = [1, 2, 3];
let result = [];
for (var i = 0; i < userInput.length; i++) {
     result.push(userInput[i] * userInput[i])
}
console.log(result);
```

Esimerkkikoodi 1. Imperatiivista koodia, jossa kerrotaan tarkkaan, mitä ja miten halutaan tehtävän. Tulostuu [1, 4, 9].

```
const userInput = [1, 2, 3];
let result = userInput.map(num => num * num);
console.log(result);
```

Esimerkkikoodi 2. Deklaratiivista koodia, jossa kerrotaan, että halutaan lista, jonka alkiot ovat userInput-listan alkiot kerrottuna itsellään. Tulostuu myös [1, 4, 9].

Näin ollen React osaa itse optimoida ohjelman suorituskykyä ja päivittää oikeat komponentit. Deklaratiivinen koodi on myös paljon helpommin luettavaa (11, Principles of React.).

3.2 Yksinkertainen React-komponentti

Esimerkkikoodissa 3 on toteutettu komponentti, joka tulostaa verkkosivulle ”Hello Robert”. Tämä on yksinkertaisin tapa luoda komponentteja Reactissa, ja näitä komponentteja kutsutaan funktiokomponenteiksi. Funktiokomponentit ovat myös uudempi lisäys Reactiin verrattuna luokkakomponetteihin, ja ne vaikuttavat helppokäyttöisemmiltä ja monipuolisemmilta. Tästä syystä työssä pyrittiin käyttämään funktiokomponentteja aina, kun mahdollista.

```
const Hello = props => {
      return (
            <div>
                  Hello {props.name}
            </div>
      );
}

ReactDOM.render(
      <Hello name=”Robert” />,
      document.getElementById(‘root’)
);
```

Esimerkkikoodi 3. Yksinkertainen React-komponentti.

Komponentille välitetään data propsien avulla. Komponentti siis luo div-elementin, jossa on teksti ”Hello” ja propsien läpi saatu data. Lopulta React DOM renderoi komponentin verkkosivulle. Propsien avulla voidaan komponenteille tuoda kaikenlaista dataa, kuten numeroita, merkkijonoja ja taulukoita. (14.)

3.3 Tilan hallinta React-komponenteissa

Yksi SPA-sovelluksen suurimmista haasteista on käyttöliittymän pitäminen ajan tasalla. Reactissa tämä onnistuu, kun komponenttiin lisätään yksi tai useampi tila (engl. state).

Esimerkkikoodissa 4 toteutettu React-komponentti on luokkakomponentti (engl. class component). Ennen, jos React-komponentissa halusi käyttää Reactin tilaa tai elinkaari-funktioita, oli pakko käyttää luokkakomponentteja. Tämä on muuttunut React Hooks -ominaisuuksien saapumisen myötä, joita tutkitaan luvussa 3.4.

```
import React from ‘react’;

class DisplayTime extends React.Component {
      constructor(props) {
            super(props);
            this.state = {date: new Date()};
      }

      componentDidMount() {
            this.timerID = setInterval(() => this.tick(), 1000);
      }

      componentWillUnmount() {
            clearInterval(this.timerID);
      }


      tick() {
            this.setState({date: new Date()});
```

```
      }

      render() {
            return (
                  <div>
                        <h1>Hello, world!</h1>
                        <h2>It is {this.state.date.toLocaleString()}.</h2>
                  </div>
            );
      }
}

ReactDOM.render(
    <DisplayDate />,
  document.getElementById('root')
);
```

Esimerkkikoodi 4. Tilallinen React-luokkakomponentti.

Esimerkkikoodin 4 komponentissa on kaksi funktiota, jotka reagoivat komponentin elinkaaren muutoksiin, componentDidMount() ja componentDidUnmount(). Ensimmäinen suoriutuu aina, kun komponentti kiinnitetään DOM:iin, ja toinen juuri ennen, kuin se poistuu siitä. Näiden avulla on luotu laskuri, joka muuttaa joka sekunti tilan arvoksi uuden tämänhetkisen päivämäärän ja kellonajan. Koska sivu päivittyy aina, kun komponentin tila muuttuu, saadaan sivulle tulostumaan toimiva kello, joka päivittyy joka sekunti. (15.)

3.4 React Hook -toiminnallisuudet

React Hookit ovat joukko Reactin uusi toiminnallisuuksia (lisättiin React 16.8:aan), jotka mahdollistavat muun muassa tilan ja elinkaaritoimintojen käyttämisen Reactin funktiokomponenteissa. Niiden avulla Reactia voi käyttää käyttämällä pelkästään funktiokomponentteja luokkakomponenttien sijaan, jotka ovat vaikeammin ymmärrettäviä ja käytettäviä. Hookeja ei myöskään voi käyttää luokkakomponenteissa. Esimerkkikoodissa 5 nähdään esimerkkikoodin 4 kello toteutettuna funktiokomponentilla, joka käyttää State- ja Effect-hookeja. (11.)

State-hookia käytetään useState()-funktiolla. Määritetään date-muuttuja, johon tila tallennetaan ja setDate(), joka on funktio, jolla tilaa voi muuttaa. Antamalla useState()-funktiolle parametri voidaan määrittää tilan alkuperäinen arvo. Effect-hookia käytetään useEffect()-funktiolla, joka suoriutuu joka kerta, kun komponentti renderoidaan. Tämä

vastaa luokkakomponentin componentDidMount()-funktiota. Jos jonkin funktion palauttaa useEffect()-funktion sisällä, se vastaa luokkakomponentin componentWillUnmount()-funktiota.

```
import React, {useEffect, useState} from 'react';

const DisplayTime = props => {
      const [date, setDate] = useState(new Date())
      let timerID = 0;

      useEffect(() => {
            timerID = setInterval(() => tick(), 1000);
            return function cleanUp() {
                  clearInterval(timerID);
            }
      }, [])

      const tick = () => {
            setDate(new Date());
      }

      return (
            <div>
                  <h1>Hello, world!</h1>
                  <h2>It is {date.toLocaleString()}.</h2>
            </div>
      );

};
```

Esimerkkikoodi 5. Tilallinen React-funktiokomponentti, joka myös käyttää elinkaaritoimintoja Effect hookin avulla.

Käyttämällä Reactin hookeja voidaan siis käyttää tilaa ja elinkaaritoimintoja myös funktiokomponenteissa, mikä ei ennen ollut mahdollista. Tässä insinöörityössä funktiokomponentit ovat myös isommassa osassa kuin luokkakomponentit, ja näin ollen myös React hookeja käytetään paljon.

4 Lokalisoinnin toteutustavat

Lokalisointi toteutettiin ennen perinteisesti vain manuaalisesti. Käännettävät tekstit syötettiin taulukkoihin merkkijonoina, minkä jälkeen taulukot lähetettiin kääntäjille. Kun taulukot saatiin takaisin kääntäjiltä käännettyinä, lisäsivät kehittäjät käännetyt tekstit koodiin. Automatisoidut prosessit ovat sekä tehokkaampia että vähemmän alttiita virheille kuin manuaaliset toimintatavat, mutta ne ovat myös kustannustehokkaampia. Nykyajan teknologiat tarjoavat yrityksille paljon edistyneempiä toteutustapoja lokalisoinnille, ja niitä käytettiin myös tässä insinöörityössä. (6.)

Insinöörityössä tutkitaan kolmea erilaista React-ohjelmistokehystä, joilla lokalisointi voidaan toteuttaa: React Localize Redux, React Intl ja React i18next. Lisäksi selvitetään esimerkkien avulla jokaisen ohjelmistokehityksen vahvuudet ja heikkoudet sekä tavat, miten ne voidaan rakentaa osaksi valmista React-sovellusta.

4.1 React Localize Redux -ohjelmistokehys

React Localize Redux on kevyt ohjelmistokehys, joka on helppo asentaa ja helppokäyttöinen. Pääosin tämä ohjelmistokehys on tarkoitettu käytettäväksi React Reduxin kanssa, mutta se toimii myös ilman Reduxia. Koska Localize Redux on niin kevyt, se soveltuu parhaiten käytettäväksi pieniin ja yksinkertaisiin sovelluksiin, joissa tarvitaan vain käännöksiä. Se ei tue monimutkaisempia toimintoja, kuten päivämäärien tai monikkojen formatointia, joten se ei sovellu sellaisiin sovelluksiin, jotka sitä vaativat.

React Localize Redux asennetaan React-sovellukseen npm-komennolla esimerkkikoodin 6 mukaisesti.

```
npm install react-localize-redux –save
```

Esimerkkikoodi 6. React Localize Reduxin asennuskomento.

Käyttöönotto

Jotta päästään käsiksi ohjelmistokehyksen toiminnallisuuksiin, on React Localize Redux ensin otettava käyttöön. Tämä tapahtuu käärimällä React-sovellus LocalizeProvider-elementin sisään, kuten nähdään esimerkkikoodista 7. Näin ollen on sovellus LocalizeProviderin lapsi ja Reactin toiminnallisuuden mukaan on sillä pääsy sen toimintoihin.

```
import React from ‘react’;
import { LocalizeProvider } from ‘react-localize-redux’;
import App from ‘./App’;

ReactDom.render(
      <LocalizeProvider>
            <App />
      </LocalizeProvider>
);
```

Esimerkkikoodi 7. LocalizeProvider määritetään sovelluksen hierarkian ylimmälle tasolle.

```
import React from 'react';
import { renderToStaticMarkup } from 'react-dom/server';
import { withLocalize } from 'react-localize-redux';
import Translations from './translations/translations.json';

class App extends React.Component {
      constructor(props) {
      super(props);

      this.props.initialize({
            languages: [
                  { name: "English", code: "en" },
                  { name: "Finnish", code: "fi" },
                  { name: "Swedish", code: "se" }
       ],
       translation: Translations,
       options: {
         renderToStaticMarkup,
         defaultLanguage: "en"
       }
     });
  }

  render() {
    // Tähän määritetään komponentin näkymä
  }
}

export default withLocalize(App);
```

Esimerkkikoodi 8. React Localize Reduxin alustus.

Kuten esimerkkikoodissa 8 nähdään, on React Localize Redux alustettava niillä kielillä, joita sovellus tulee käyttämään. Kielet alustetaan taulukkona niin, että jokaisella kielellä on nimi ja koodi. Käännökset laitetaan json-tiedoston sisään esimerkkikoodin 9 mukaisesti, ja tiedosto tuodaan sovellukseen import-komennon avulla (esimerkkikoodi 8).

```
{
      "welcome": ["Welcome to Fujitsu eSignature", "Tervetuloa Fujitsun eSigna-
      ture -palveluun", "Välkommen till Fujitsu eSignature"]
}
```

Esimerkkikoodi 9. translations.json-tiedosto, johon lisätään tekstikäännökset.

Kielen valinta ja käännöksien käyttö

Kun React Localize Redux on asennettu ja alustettu ja käännökset on tuotu sovellukseen, voidaan alkaa toteuttaa itse sovellusta. Kielen valitsemiseen voidaan käyttää useita eri toteutustapoja, mutta se tapahtuu aina React Localize Reduxin setActiveLanguage()-funktiolla. Kieli voitaisiin valita esimerkiksi luomalla sovellukseen alasvetova-

likko, jossa on lista sovelluksen tukemista kielistä ja valintaa muuttamalla vaihtuu aktiivinen kieli. Vaihtoehtoisesti voitaisiin luoda jokaiselle kielelle omat napit, ja nappeja painamalla vaihtuu aktiivinen kieli kyseisen napin kieleksi, kuten nähdään esimerkkikoodissa 10.

```
const ToggleLanguage = ({ languages, activeLanguage, setActiveLanguage }) => (
      <ul className=”languageSelector”>
            {languages.map(language => (
                  <li key={language.code}>
                        <button onClick={() => setActiveLanguage(language.code)}>
                              {language.name}
                        </button>
                  </li>
            ))}
      </ul>
);
```

Esimerkkikoodi 10. React-komponentti, jossa nappi jokaiselle kielelle aktiivisen kielen vaihtamista varten.

Kuten esimerkkikoodissa 11 nähdään, on käännöksien käyttäminen yksinkertaista. Translate()-funktiolle annetaan parametrina arvo riippuen siitä, mikä teksti halutaan näyttää, ja translate()-funktio palauttaa json-tiedostosta kyseisen parametrin arvon aktiivisen kielen mukaisesti.

```
<h1>{ translate(“welcome”) }</h1>
```

Esimerkkikoodi 11. Käännöksien käyttäminen translate()-funktion avulla.

Kuten esimerkeistä nähdään, on React Localize Redux kevyt ja helppokäyttöinen. Lisäksi se pitää koodin helppolukuisena. Sen vähäisien toiminnallisuuksien takia se ei kuitenkaan soveltunut käytettäväksi Fujitsun sähköiseen allekirjoituspalveluun.

4.2 FormatJS/React Intl -ohjelmistokehys

FormatJS on kokoelma JavaScript-kirjastoja, joiden tarkoitus on tarjota kehittäjille keinoja sovelluksien lokalisointiin. Sen avulla tekstien, päivämäärien ja merkkijonojen formatointi on helppoa, mikä tekee siitä kattavan paketin sovelluksien lokalisointiin. Siinä missä React Localize Redux oli kevyt ja tarkoitettu vain käännöksien toteuttamiseen, on

FormatJS:ssä kaikki tarvittava lokalisointiin, mukaan lukien päivämäärien ja monikkomuotojen formatointi. React Intl on FormatJS:n integraatio Reactiin, joten insinöörityössä puhutaan tästä eteenpäin React Intl:stä.

Kuten React Localize Redux, on React Intl yhtä helppo asentaa npm-komennolla (esimerkkikoodi 12).

```
npm i -S react react-intl
```

Esimerkkikoodi 12. React Intl:n asennus.

React Intl otetaan käyttöön määrittämällä IntlProvider-elementti koko sovelluksen hierarkian päätasolle. Näin se antaa pääsyn toiminnallisuuksiinsa kaikille lapsillensa (esimerkkikoodi 13).

```
ReactDom.render(
      <IntlProvider locale={locale} messages={messages}>
            <App />
      </IntlProvider>
);
```

Esimerkkikoodi 13. IntlProvider on määritettävä hierarkian ylimmälle tasolle, jotta sen toiminnallisuudet olisivat muiden komponenttien käytettävissä.

React Intl:ää ei tarvitse erikseen alustaa kuten React Localize Redux.

Kielen valinta ja valitun kielen käännöksien syöttö sovellukseen

Koska kaikkien kielien käännöksiä ei React Intl:ssä voi antaa yhdessä json-tiedostossa kuten React Localize Reduxissa, on hyvä tehdä jonkinlaisen toteutus kielen valitsemiselle, joka sitten valitun kielen mukaan syöttää käännökset sovellukselle propsien avulla.

Esimerkkikoodissa 14 on toteutettu React-komponentti ChooseLanguage, joka ensin hakee käyttäjän selaimen kielen navigator.language-komennolla. Jos käyttäjän selaimen kieli on suomi tai ruotsi, käytetään suomen- tai ruotsinkielisiä käännöksiä, muissa tapauksissa englanninkielisiä. Kun oikea kieli on selvillä, syötetään IntlProvider-elementille oikea lokaali ja käännökset propsien avulla.

```
import React, { useState } from 'react';
import { IntlProvider } from 'react-intl';
import Finnish from '../languages/fi-FI.json';
import English from '../languages/en-EN.json';
import Swedish from '../languages/se-SE.json';

export const Context = React.createContext();
const userLocale = navigator.language;

let lang;
if (userLocale === "fi-FI") {
     lang = Finnish;
} else if (userLocale === "se-SE") {
     lang = Swedish;
} else {
     lang = English;
}

const ChooseLanguage = (props) => {
     const [locale, setLocale] = useState(userLocale);
     const [messages, setMessages] = useState(lang);

     function selectLang(e) {
          const newLocale = e.target.value;
          setLocale(newLocale);
          if (newLocale === "fi-FI") {
               setMessages(Finnish);
          } else if (newLocale === "se-SE") {
               setMessages(Swedish);
          } else {
               setMessages(English);
          }
     }

     return (
          <Context.Provider value={{ locale, selectLang }}>
               <IntlProvider locale={locale} messages={messages}>
                    {props.children}
               </IntlProvider>
          </Context.Provider>
     );
}
```

Esimerkkikoodi 14. React-komponentti ChooseLanguage, joka valitun kielen mukaisesti syöttää sovellukselle oikeat käännökset propsien avulla.

Kieli on myös mahdollista vaihtaa jälkikäteen käyttämällä Reactin Context-toiminnallisuutta: sen avulla voidaan muille React-komponenteille antaa pääsy tämän komponentin muuttujiin tai funktioihin. Tässä tapauksessa pääsy annetaan locale-muuttujaan ja selectLang()-funktioon, joka asettaa uuden ”localen” ja samalla uudet käännökset IntlProvider-elementille. Näin ollen esimerkiksi alasvetovalikkoa, jolla kieli valitaan, ei tarvitse määritellä tässä komponentissa ja näin koodi pysyy siistinä ja jäsenneltynä, kuten nähdään esimerkkikoodista 15.

```
import React, { useContext } from 'react';
import { Context } from './components/ChooseLanguage';

const App = (props) => [
      const context = useContext(Context);

      return (
            <div>
                  <select value={context.locale} onChange={context.selectLang}>
                        <option value="en-EN">English</option>
                        <option value="fi-FI">Finnish</option>
                        <option value="se-SE">Swedish</option>
                  </select>
            </div>
      );
}
```

Esimerkkikoodi 15. Kielen voi vaihtaa missä vain muussa komponentissa, koska Context-toiminnallisuuden avulla saadaan pääsy selectLang()-funktioon.

Tämä mahdollistaa myös sen, että kielen valinnan komponentti saadaan näkymään eri puolelle verkkosivua.

Formatointi ja käännöksien käyttö

React Intl:ssä on tarjolla kaksi eri kehitysrajapintaa: Imperative API ja sen omat React-komponentit. Imperative API:a suositellaan käytettäväksi vain kun

- React-komponentteja ei voida käyttää, esimerkiksi kun halutaan kääntää tekstien attribuutteja
- muussa kuin React-ympäristössä, kuten Node.js
- suorituskyky on huono liiallisten React-komponenttien vuoksi.

Seuraavaksi tutkitaan React Intl:n omia React-komponentteja ja miten niitä voidaan käyttää hyväksi lukujen, päivämäärien ja tekstien formatoinnissa sekä käännöksien käytössä.

Päivämäärien formatointi onnistuu React Intl:llä hyvin FormattedDate-komponentilla. Koska React Intl:ään on sisäänrakennetut käännökset joka kielelle, esimerkiksi viikonpäiville ja kuukausille, ei käännöksiä tarvitse itse syöttää mihinkään. Muuttamalla FormattedDate-komponentin asetuksia voidaan muuttaa päivämäärän formatointia. Näillä asetuksilla viikonpäivät ja kuukaudet näkyvät tekstinä, päivämäärät ja vuodet lukuina (esimerkkikoodi 16).

```
<FormattedDate
      value={new Date(Date.now())}
      year="numeric"
      month="long"
      day="numeric"
      weekday="long"
/>
```

Esimerkkikoodi 16. FormattedDate-komponentti.

Esimerkkikoodilla 16 tulostuu

- suomeksi: maanantaina 26. lokakuuta 2020
- ruotsiksi: måndag 26 oktober 2020
- englanniksi: Monday, October 26, 2020.

Kuten tuloksista nähdään, osaa React Intl useimpien kielien kuukaudet ja viikonpäivät, mutta se osaa myös formatoida päivämäärän englannin kohdalla oikein niin, että kuukausi tulee päivämäärää ennen. Jos kaikki olisi määritetty näkymään lukuina ja ilman viikonpäivää, tulostuisi

- suomeksi: 26.10.2020
- ruotsiksi: 2020-10-26
- englanniksi: 10/26/2020.

Oikeantapainen formatointi ei siis rajoitu pelkästään järjestykseen, vaan myös merkkeihin.

Tekstien formatointiin ja käännöksien käyttöön voidaan käyttää React Intl:n FormattedMessage-komponenttia (esimerkkikoodi 17).

```
<FormattedMessage
      id="Welcome"
      values={{
            strong: (serviceName) => <strong>{serviceName}</strong>
      }}
/>
<FormattedMessage
      id="Pending"
      values={{
            count: {count},
            a: (link) => <a href="#">{link}</a>
      }}
/>
```

Esimerkkikoodi 17. Kaksi FormattedMessage-komponenttia.

React Intl mahdollistaa tekstin kääntämisen ja formatoinnin sekä html-elementtien käytön käännöksissä. Luvussa 4.1 tutkitulla React Localize Reduxilla ei ollut mitään keinoa esimerkiksi tehdä linkkiä yhdestä käännetyn tekstin sanasta, koska käännökset määritettiin json-tiedostoissa eikä html-syntaksia voi niissä käyttää. React Intl:ssäkin käännökset määritetään json-tiedostoon (jokaiselle kielelle oma), mutta FormattedMessage-elementin values-propsille arvoja antamalla voidaan määritellä, miten json-tiedoston käännöstekstit näkyvät verkkosovelluksessa.

React Intl tunnistaa json-tiedostojen sisään määritetyt html-elementit (esimerkkikoodi 18), ja osaa näiden määritysten perusteella näyttää käännökset oikein verkkosivulla. Myös monikkomuotojen formatointi onnistuu hyvin näin: jos FormattedMessage-komponentille annetun count-muuttujan arvo on yksi, tulostuu ”uusi dokumentti”, ja jos arvo on mikä vain muu, tulostuu ”uutta dokumenttia”.

```
{
      “Welcome”: “Tervetuloa <strong>Fujitsu eSignature</strong>-palveluun!”,
      “Pending”: “Sinulla on <a>{count, plural, one {# uusi dokumentti} other
      {# uutta dokumenttia}}</a> allekirjoitettavana.”
}
```

Esimerkkikoodi 18. Json-tiedosto suomenkielisille käännöksille.

React Intl on erittäin kattava tapa toteuttaa lokalisointi, ja se varmasti soveltuisi käytettäväksi useimpiin React-sovelluksiin. Insinöörityössä tutkitut komponentit FormattedDate ja FormattedMessage olivat vain pintaraapaisu kaikista React Intl:n toiminnallisuuksista; tarjolla olisi omat komponentit muun muassa ajan ja numeroiden formatointia varten. Vastakohtana React Localize Reduxille, se olisi jo liiankin kattava Fujitsun sähköistä allekirjoituspalvelua varten. Lisäksi se tekee koodista hieman rumempaa ja vaikeammin luettavaa. Näistä syistä sitä ei päätetty toteuttaa palveluun.

4.3 React i18next -ohjelmistokehys

React i18next on i18next-ohjelmistokehyksen integraatio Reactiin. I18next on JavaScript-ohjelmistokehys, mutta siitä on tehty integraatiot useimpiin suosittuihin frontend-ohjelmistokehyksiin, kuten AngularJS:ään ja Vue.js:ään.

Verrattuna edellä tässä insinöörityössä tutkittuihin ohjelmistokehyksiin se sijoittuu aika lailla keskiväliin ominaisuuksien määrän ja helppokäyttöisyyden perusteella. Se ei ole yhtä kevyt kuin React Localize Redux, mutta melkein yhtä helppokäyttöinen, ja siitä löytyy melkein kaikki samat ominaisuudet kuin React Intl:stä. Tästä syystä se valittiin toteutettavaksi Fujitsun sähköiseen allekirjoituspalveluun.

Myös React i18next asennetaan sovellukseen npm-komennon avulla, kuten nähdään esimerkkikoodissa 19.

```
npm install react-i18next i18next –save
npm install i18next-http-backend i18next-browser-languagedetector --save
```

Esimerkkikoodi 19. React i18next:n asennuskomento.

Koska käännökset halutaan pitää ulkoisissa json-tiedostoissa eikä itse sovelluksen koodissa, pitää asentaa myös i18next-http-backend-lisäosa. Se mahdollistaa käännöksien latauksen muualta. Muuten käännökset olisi pitänyt määrittää ohjelmistokehyksen alustavassa i18n.js-tiedostossa, mikä olisi tehnyt koodista rumaa ja liian täyttä. Jos haluttaisiin, että i18next tunnistaisi automaattisesti käyttäjän ”localen”, olisi voitu asentaa i18next-browser-languagedetector-lisäosa, mutta sitä ei kuitenkaan eSignature-palveluun haluttu.

Käyttöönotto

Ohjelmistokehys otetaan käyttöön luomalla i18n.js-tiedosto samalle tasolle sovelluksen index.js-tiedoston kanssa (esimerkkikoodi 20).

```
import i18n from ‘i18next’;
import [ initReactI18next } from ‘react-i18next’;
import Backend from ‘i18next-http-backend’;

const languages = [ “en”, “fi”, “se” ];

i18n
      .use(Backend)
      .use(initReactI18next)
      .init({
            lng: “en”,
            keySeparator: false,
            fallbackLng: “en”,
            debug: false,
            whitelist: languages,
            backend: {
                  loadPath: “./locales/{{lng}}/{{ns}}.json”
            },
```

```
            react: {
                  useSuspense: true
            }
      });

export default i18n;
```

Esimerkkikoodi 20. i18n.js-tiedosto, joka alustaa react-i18next:n.

Alustuksessa ohjeistetaan ohjelmistokehys käyttämään vain tiettyjä kieliä ja määritetään polku jokaisen kielen json-tiedostolle, johon käännökset oli kääntäjien toimesta tehty. Lisäksi määritetään, että kieli on aina aluksi englanti, ja käyttäjä voi sitten itse muuttaa kieltä joko suomeksi tai ruotsiksi. Lisäksi on tärkeää merkitä keySeparator-arvo epätodeksi, jotta ohjelmistokehys osaa hakea käännökset oikein ”tasaisista” json-tiedostoista, joita palvelussa on. Tasainen json-tiedosto tarkoittaa sellaista json-tiedostoa, jossa kaikki avaimet ovat ylimmällä tasolla eikä esimerkiksi niin, että jotkut avaimet ovat toisen avaimen sisällä. Koska palvelussa käytetään backend-lisäosaa, pitää ohjelmistokehystä ohjeistaa, että sovellus tulee käyttämään Reactin Suspense-ominaisuutta.

Tämän jälkeen i18n tuodaan (engl. import) sovellukseen index.js-tiedostossa (esimerkkikoodi 21) ja määritetään sovellus käyttämään Reactin Suspense-ominaisuutta. Suspense-ominaisuutta pitää käyttää, jos käytetään backend-lisäosaa. Suspensen avulla sovellus odottaa, että käännökset latautuvat sovellukseen json-tiedostosta ennen renderointia. Muuten sivusto renderoisi vain tyhjää, koska käännökset eivät ole vielä latautuneet palvelimelta. Kuten esimerkeistä nähdään, on react-i18next:n alustus ja käyttöönotto myös yksinkertaista.

```
import ‘./i18n’;

const rootElement = document.getElementById(‘root’);

ReactDOM.render(
      <BrowserRouter>
            <Suspense fallback=”Loading…”>
                  <App />
            </Suspense>
      </BrowserRouter>,
rootElement);
```

Esimerkkikoodi 21. Sähköisen allekirjoituspalvelun index-js-tiedosto. Tiedostosta näkyy vain osa koodista, joka on oleellista lokalisointia varten. Muu koodi on luottamuksellista.

Toimintojen käyttö

React-i18next:n toimintojen käyttö vaihtelee hieman riippuen siitä, käytetäänkö niitä Reactin funktio- vai luokkakomponenteissa. Seuraavaksi tutkitaan ohjelmistokehyksen toimintojen käyttöä kummassakin tapauksessa.

Funktiokomponenteissa voi käyttää ohjelmistokehyksen useTranslation-hookia. UseTranslation tuodaan ensin komponenttiin import-komennolla, minkä jälkeen sitä käyttämällä saadaan komponentille käyttöön t()-funktio ja i18n-instanssi (esimerkkikoodi 22).

```
import React from 'react';
import { useTranslation, Trans } from 'react-i18next';
import Customize from './Customize'
import custom from './esigncustom.json'

const Home = props => {
      const [customTexts, setCustomTexts] = useState(custom);
      const { t, i18n } = useTranslation();

      useEffect(() => {
            fetch(Customize.CustomTextsPath(), { method: "get" })
            .then(response => response.json())
            .then(data => setCustomTexts(data))
            .catch(error => {
                  console.log("Fetching customtexts error.")
            });
      }, [])

      return (
            <div>
                  <Customize.Style />
                  <Customize.MainLogo />
                  <h1>{t("Welcome", {customTexts})}</h1>
                  <h3>
                        <Trans i18nKey="You have new documents to sign"
                        count={count}>You have <a href=#>{{count}} new docu-
                        ments</a> to sign</Trans>
                  </h3>
                  {renderCustomTexts()}
            </div>
      );
}
```

Esimerkkikoodi 22. Otos palvelun Home.js-tiedostosta. Vain lokalisointiin ja kustomointiin liittyvä koodi on näkyvillä, koska muu koodi on luottamuksellista.

I18n-instanssi antaa pääsyn yleisiin toiminnallisuuksiin, kuten nykyisen kielen hakuun ja kielen valitsemiseen. T()-funktio on ohjelmistokehyksen yleisin ja helpoin tapa käyttää käännöksiä, ja sitä on hyvä käyttää aina paitsi silloin, jos esimerkiksi käännösten sisään halutaan linkkejä tai muita html-elementtejä. T()-funktiolle annetaan parametrina avain,

jolla se hakee aktiivisen kielen json-tiedostosta avainta vastaavan käännöksen. T()-funktiolle voi myös antaa toisen parametrin. Tätä tutkitaan tarkemmin insinöörityön luvussa 5.

Jos käännöksissä halutaan käyttää tekstiformatointia tai html-elementtejä, kuten linkkejä, pitää käyttää ohjelmistokehyksen Trans-komponenttia. Sille annetaan i18nKey-propsiin käännösavain ja count-propsiin arvo, joka määrittelee monikkoformatoinnin. Esimerkkikoodin 22 Trans-komponentilla halutaan saada osa käännöstekstistä näkymään linkillä; itse linkki määritetään komponentin sisään (lapseksi) ja json-tiedostossa määritetään, minkä sanojen halutaan näkyvän linkkinä (esimerkkikoodi 23).

```
{
      "Welcome": "Tervetuloa {{customTexts.serviceName}} -palveluun.",
      "You have new documents to sign": "Sinulla on <1>{{count}} uusi doku-
      mentti</1> allekirjoitettavana.",
      "You have new documents to sign_plural": "Sinulla on <1>{{count}} uutta
      dokumenttia</1> allekirjoitettavana."
}
```

Esimerkkikoodi 23. Otos suomenkielisestä translation.json-tiedostosta.

Linkkinä näkyvät sanat kääritään tässä tapauksessa <1></1>-merkkien sisään, linkki vastaa Trans-komponentin toista alkiota (alkiot alkavat nollasta):

- alkio 0: ”You have ” -merkkijono
- alkio 1: a-elementti (linkki)
- alkio 2: count-arvo
- alkio 3: ”new documents to sign.” -merkkijono.

Jos count-arvo on 1, käyttää Trans-komponentti käännösavainta ”You have new documents to sign”, muussa tapauksessa käännösavainta ”You have new documents to sign_plural”. Tämä on hyvä tapa ratkaista monikkosääntöongelma.

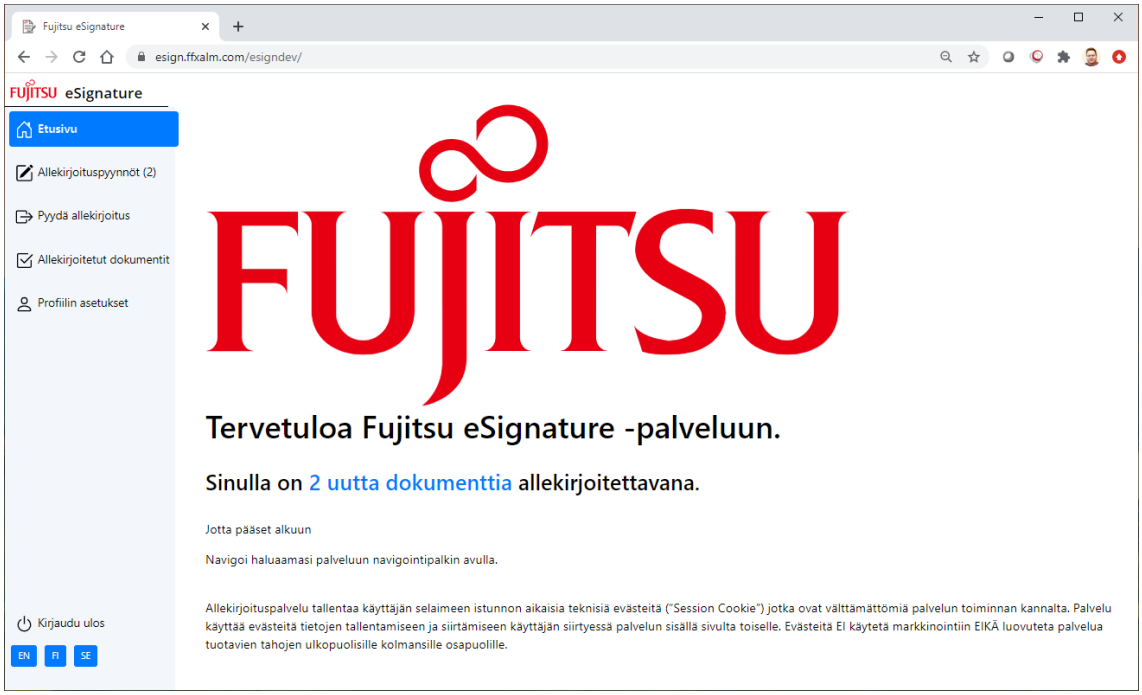
Luokkakomponenteissa pitää käyttää withTranslation-komponenttia. Se on HOC (Higher Order Component), joka on funktio, jolle annetaan parametrina luotu luokkakomponentti, joka sen jälkeen palauttaa uuden komponentin. Tämä komponentti pääsee sitten käsiksi t()-funktioon ja i18n-instanssiin propsien avulla, kuten nähdään esimerkkikoodissa 24.

```
import React, { Component } from 'react';
import { withTranslation } from 'react-i18next';

class Pending extends Component {
      constructor(props} {
            super(props)
      }

render() {
      const { t, i18n } = this.props;
      return (
            <h1>{t("Signature Requests")}</h1>
      )
}

export default withTranslation()(Pending);
```

Esimerkkikoodi 24. Otos Pending.js-tiedostosta, joka on luokkakomponentti.

Kielen valinta

Allekirjoituspalvelun kielen valinta päätettiin toteuttaa sovelluksen navigointipalkkiin. Näin kielen voi muuttaa riippumatta siitä, missä tilassa sovellus on, koska navigointipalkki on aina näkyvissä sovelluksen vasemmassa reunassa (kuva 4).



Kuva 4. Fujitsu eSignature -palvelun käyttöliittymä suomenkielisenä. Vasemmalla näkyy navigointipalkki, jonka alaosassa ovat näppäimet, jotka vaihtavat aktiivisen kielen.

Koska oli päätetty, että palvelussa ei automaattista kielentunnistusta käytetä, oli haasteena saada kieli pysymään valittuna kielenä silloin, kun verkkosivusto ladattiin uudestaan. Koska i18n:n alustus tapahtuu aina sivun uudelleen latautuessa, se muuttaa kielen takaisin englanniksi, koska niin oli 18n:n alustuksessa määritelty. Tämän korjaamiseksi käytetiin ratkaisua, joka luo (jos sellaista ei ole) tai kirjoittaa ”language”-evästeen yli, johon tallennetaan valittu kieli aina, kun kielenvalintanäppäintä painetaan. Sivun uudelleenlatautumisen jälkeen tarkistetaan valittu kieli kyseisestä evästeestä ja vaihdetaan kieli takaisin aikaisemmin valituksi kieleksi (esimerkkikoodi 25).

```
import { Cookies } from 'react-cookie';

const cookies = new Cookies();

const updateLanguage = (language) => {
      var languageCookie = cookies.get("language");
      if (languageCookie !== language) {
            cookies.set("language", language);
      }
      if (language !== i18next.language) {
            i18next.changeLanguage(language);
      }
}
```

Esimerkkikoodi 25. Otos App.js-tiedostosta, jossa määritelty updateLanguage()-takaisinkutsufunktio, jota kutsutaan aina, kun kielenvalintanäppäintä painetaan.

Kielen vaihtamiseksi sivun uudelleenlatauksen jälkeen oli hyvä käyttää Reactin Effect-hookia, joka on hyvä tapa tehdä sivutoimenpiteitä, kuten datan noutoa palvelimelta. Tavallisesti Effect-hookit suoriutuvat aina kun sovellus päivittyy tai kun sovellus renderoituu uudestaan. Tässä tapauksessa Effect-hookin ei tarvinnut suoriutua, kun silloin, jos sivusto uudelleen latautuu. Antamalla Effect-hookille parametrina tyhjä taulukko vältyttiin sen suoriutumiselta sivuston renderoituessa uudestaan ja näin ollen parannettiin sovelluksen suorituskykyä (esimerkkikoodi 26).

```
useEffect(() => {
      var language = cookies.get("language");
      if (language !== undefined && language !== i18next.language) {
            i18next.changeLanguage(language);
      }
}, []);
```

Esimerkkikoodi 26. Otos App.js-tiedostosta, jossa määritelty Reactin Effect-hook, jossa kieli vaihdetaan takaisin valituksi kieleksi sivun uudelleenlataamisen jälkeen.

5 Verkkosovelluksen kustomointi

Kustomoinnilla tarkoitetaan yleisesti sovelluksen muuttamista niin, että sen toiminnot ja visuaalinen näkymä ovat erilaiset riippuen siitä, kuka sovellusta käyttää. Kustomointi on varsinkin nykyajan globaaleilla markkinoilla tärkeä osa sovelluskehitystä. Ensinnäkin sen avulla palvelu saadaan näkymään yrityksen brändin standardien mukaisesti. Lisäksi se parantaa tuotteen käyttäjäkokemusta, koska käyttäjä voi itse muokata tuotetta omanlaisekseen ja -näköisekseen.

Sähköiseen allekirjoituspalveluun ei tarvinnut muuttaa toimintoja, sillä niiden haluttiin olevan samat jokaisella käyttäjällä. Visuaalinen näkymä haluttiin kuitenkin saada erilaiseksi riippuen käyttäjän yrityksestä. Palveluun haluttiin kustomoida

- palvelun nimi
- palvelussa näkyvät logot
- fontit
- värit
- etusivun tekstit.

Palvelussa on käytössä tietokanta, jossa on jokaisen käyttäjän tiedot. Jokaisella käyttäjällä on tietokannassa monta tietuetta, kuten nimi ja sähköpostiosoite, mutta myös tieto siitä, mistä yrityksestä hän on. Palveluun oli tehty toteutus, joka käyttäjän kirjautuessa sisään luo evästeen, josta löytyy käyttäjän yritys. Palveluun oli myös tehtävä client/public/custom-polkuun jokaiselle yritykselle oma kansio, jossa on tiedostot kustomointia varten. Kansiossa ovat seuraavat tiedostot:

- Esigncustom.json, johon lisätään yrityksen haluama palvelun nimi ja etusivun tekstit. Tekstit lisättiin kaikilla palvelun kielillä.
- Esigncustom.css, johon lisätään css-koodia, jonka avulla saadaan palveluun yrityksen haluamat värit ja fontit.
- Mainlogo.svg-kuvatiedosto, joka näkyy etusivulla.
- Navmenulogo.svg-kuvatiedosto, joka näkyy navigointipalkin yläosassa.

Kustomointi päätettiin toteuttaa palveluun Customize.js-tiedoston (esimerkkikoodi 24), johon määriteltiin omat komponentit jokaiselle kustomoinnin osa-alueelle. Tiedostosta viedään (engl. export) jokainen komponentti ulos, jotta muilla komponenteilla on niihin pääsy. Idea on siis se, että käyttäjän kirjautuessa luodun customPath-evästeen data on sama kuin yrityksen kansion nimi. Näin oli tiedostojen polku helppo määrittää koodiin.

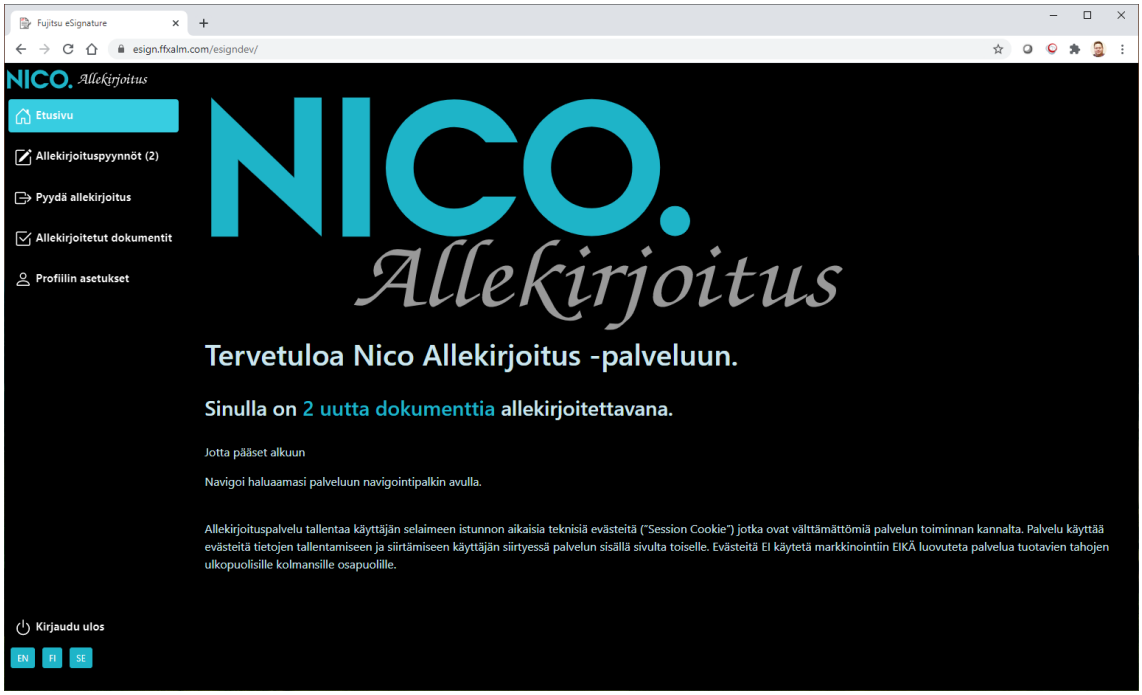
```
import React from 'react';
import { Cookies } from 'react-cookie';

const cookies = new Cookies();

const baseUrl = `custom/${cookies.get('customPath')}`;

const MainLogo = () = {
      let logoUrl = `${baseUrl}/mainlogo.svg`;
      return (
            <img src={logoUrl} alt="logo" width="70%"></img>
      );
};

const NavMenuLogo = () => {
      let logoUrl = `${baseUrl}/navmenulogo.svg`;
      return (
            <img src={logoUrl} alt="logo" height="30%"></img>
      );
};

const Style = () => {
      let styleUrl = `${baseUrl}/esigncustom.css`;
      return (
            <link rel="stylesheet" type="text/css" href={styleUrl} />
      );
};
const CustomTextsPath = () => (`${baseUrl}/esigncustom.json`);

export default { MainLogo, NavMenuLogo, Style, CustomTextsPath };
```

Esimerkkikoodi 27. Customize.js, jossa komponentit jokaiselle kustomoinnin osa-alueelle. Lopuksi komponentit viedään ulos (engl. export), jotta muut komponentit pääsevät käsiksi niihin.

Kuten esimerkkikoodin 22 (s. 26) Home-komponentista nähdään, on jokaisella komponentilla pääsy Customize-komponentteihin, kunhan Customize-komponentti importoidaan. Customize.Style-komponentti piti asettaa jokaiseen komponenttiin, jotta css-koodit suoriutuivat joka sivulle. Esigncustom.json-tiedoston data haettiin Effect-hookilla Customize.CustomTextsPath-komponentin palauttamalla polulla. Tämän jälkeen data annettiin t()-funktiolle toisena parametrina, jolloin palvelun nimeä voitiin käyttää myös käännöksissä (esimerkkikoodi 22). Home-komponentti käyttää Customize.MainLogo-kompo-

nenttia, ja NavMenu-komponentti käyttää Customize.NavMenuLogo-komponenttia. Esimerkkikoodin 22 RenderCustomTexts()-funktio on käytännössä vain if-lause, joka tulostaa yrityksen kustomoidut tekstit riippuen valitusta kielestä.

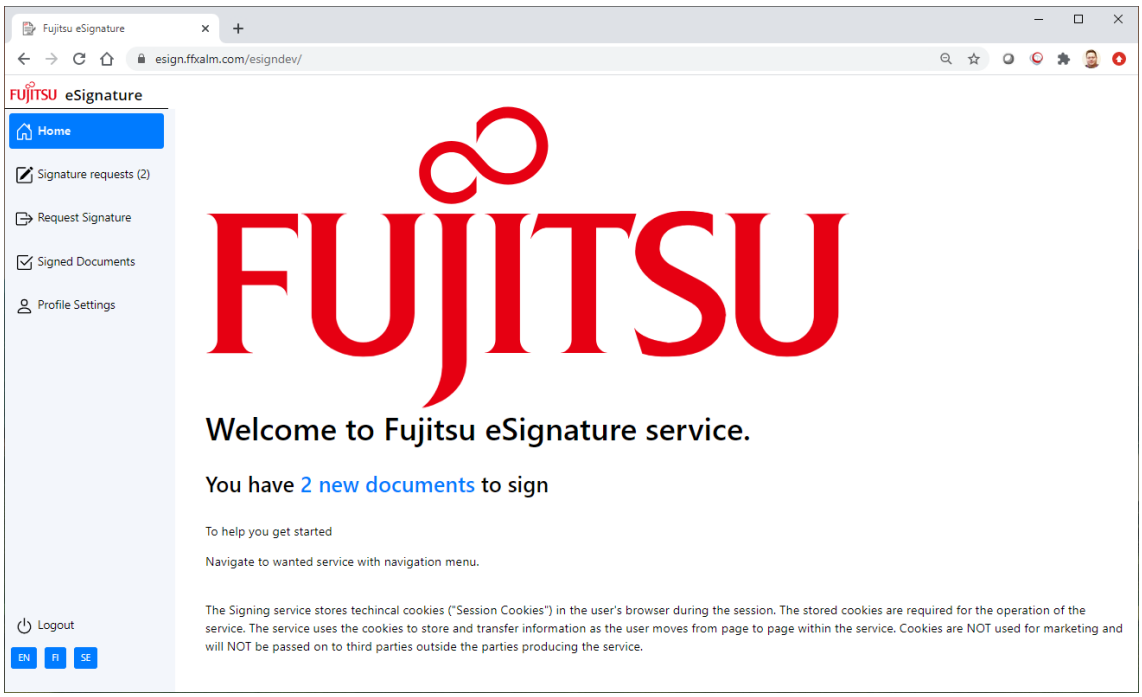
Kuten kuvasta 7 nähdään, saatiin allekirjoituspalvelun käyttöliittymän näkymä tehtyä erilaiseksi kustomoinnin avulla: koska jonkin muun yrityksen käyttäjä on kirjautunut palveluun, ovat palvelun logot, värit ja fontit erilaiset ja käytössä on myös eri palvelun nimi. Etusivun kustomoidut tekstit ovat samat, koska tälle yritykselle oli esigncustom.json-tiedostoon määritelty samat tekstit kuin Fujitsulle.



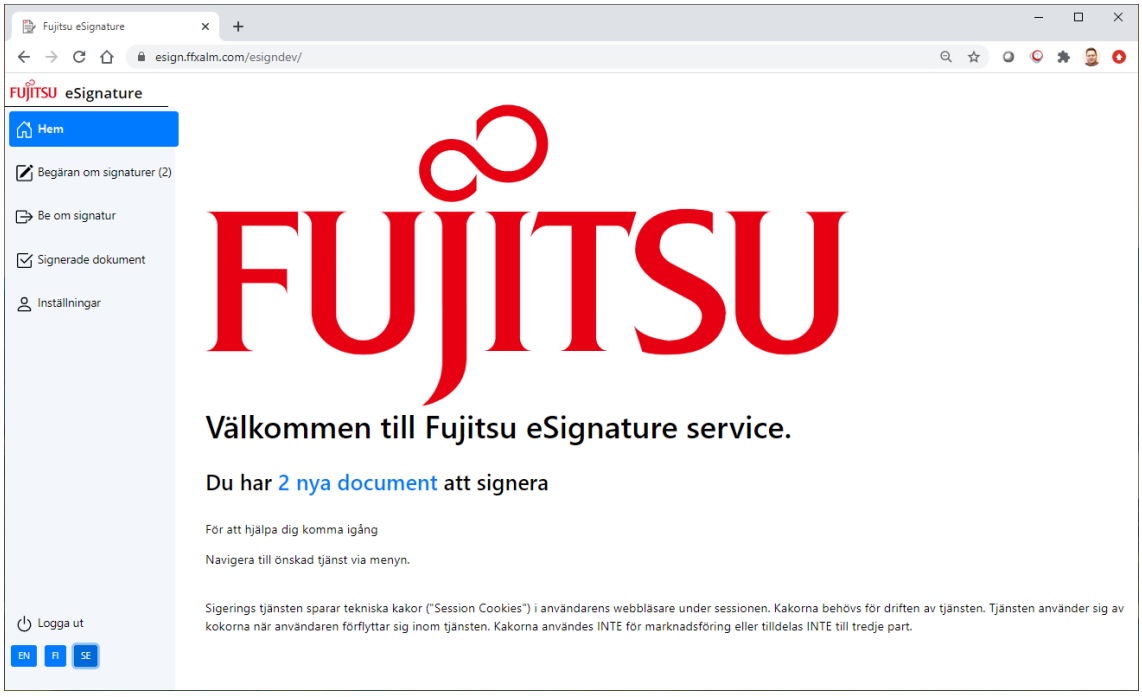
Kuva 5. Allekirjoituspalvelu kustomoituna.

6 Insinöörityön tulokset

Insinöörityössä käytettiin sähköisen allekirjoituspalvelun lokalisointiin React i18next -ohjelmistokehystä, joka on FormatJS:n (JavaScript-ohjelmistokehys) integraatio Reactia varten. Ohjelmistokehyksen avulla kaikki sähköisen allekirjoituspalvelun tekstit saatiin näkymään englannin-, suomen- ja ruotsinkielisenä (kuvat 4 (s. 28), 6 ja 7). Itse käännöstyötä ei tässä insinöörityössä tehty, vaan käännöstyö ulkoistettiin ammattilaisille, jotta mahdollisilta käännösvirheiltä vältyttäisiin.



Kuva 6. Fujitsu eSignature-palvelu englanninkielisenä.



Kuva 7. Fujitsu eSignature-palvelu ruotsinkielisenä.

Palvelu siis toimii kolmella kielellä, ja jos palveluun haluttaisiin tuoda lisää kieliä, olisi suurin työ jo tehty. Jos lisää kieliä haluttaisiin, pitäisi tehdä seuraavat toimenpiteet:

- uusien kielien alustus ohjelmistokehyksen alustustiedostoon (i18n.js)
- uusien kielien käännöksien määrittäminen omiin translation.json-tiedostoihin ja tiedostojen siirtäminen oikeisiin polkuihin
- uusien kielenvalintanäppäinten luominen käyttöliittymään ja näppäinten toiminnallisuuksien määrittäminen.

Jos kielten kokonaismäärä alkaisi nousta suureksi, olisi elegantimpi ratkaisu esimerkiksi siirtyä kielen valitsemisessa näppäimistä alasvetovalikkoon, koska näppäimet vievät melko paljon tilaa. Muuten palveluun määritetty koodi toimii niin kuin pitää: se hakee nykyisen kielen ohjelmistokehykseltä ja hakee sen kielen json-tiedostosta avainta vastaavan käännöksen.

Kustomointi toteutettiin käyttämällä omaa toteutustapaa, ja sen avulla käyttöliittymän visuaalinen näkymä muuttuu: palvelulla on eri nimi, eri tekstit etusivulla ja eri logot, värit ja fontit. Jos palveluun halutaan tuoda lisää yrityksiä se ei vaadi kuin yritysten määrittämisen palvelun tietokantaan ja kustomointitiedostojen määrittämisen ja niiden siirtämisen oikeaan polkuun.

7 Yhteenveto

Insinöörityössä selvitettiin, mitä kaikkea pitää ottaa huomioon, kun sovelluksiin tehdään lokalisointia. Lisäksi tarkoituksena oli löytää tietoa eri tavoista toteuttaa lokalisointi React-sovellukseen, tapojen hyvistä ja huonoista puolista, valita oikea tapa niiden perusteella ja lopuksi toteuttaa sekä lokalisointi että kustomointi sähköiseen allekirjoituspalveluun.

Ennen kuin lokalisointia lähdetään toteuttamaan, on pidettävä huolta, että sovellus on tarpeen mukaisesti kansainvälistetty. Tämä tarkoittaa sitä, että kaikki sovellusta käyttävät komponentit ja teknologiat tukevat niitä merkkejä ja kirjaimia, joita sovelluksessa tulee olemaan. Lokalisointia tehdessä on otettava huomioon kohdelokaalin kielelliset, tekniset ja kulttuuriset käytännöt, jotta täytetään kohdelokaalin vaatimukset esimerkiksi merkkien, ajan ja päivämäärien käytön kohdalla.

Eri React-sovelluksen tavoista löydettiin hyvin tietoa, ja näiden tietojen perusteella osattiin valita oikea tapa toteuttaa lokalisointi. Palvelussa päätettiin käyttää React i18next -

ohjelmistokehystä, joka oli muiden insinöörityössä tutkittujen lokalisointitapojen hyvien puolien yhdistelmä: se oli helppokäyttöinen ja melko kevyt, mutta piti silti sisällään paljon toiminnallisuuksia. Se myös piti koodin siistinä ja helppolukuisena.

Kustomointi toteutettiin palveluun käyttämällä itse suunniteltua toteutustapaa, jonka avulla sähköisen allekirjoituspalvelun käyttöliittymä saatiin näkymään erilaisena riippuen siitä, mistä yrityksestä palvelun käyttäjä on.

Lokalisointi ja kustomointi saatiin toteutettua sähköiseen allekirjoituspalveluun ilman suurempia ongelmia. Insinöörityötä tehdessä opittiin paljon lokalisoinnista yleisesti, mutta myös Reactin toimintatavoista ja ohjelmointitekniikoista.

Lähteet

1 Pym, Anthony. 2004. The Moving Text: Localization, Translation, and Distribution. E-kirja. Amsterdam/Phildelphia: John Benjamins Publishing Company.

2 Nitish, Singh. 2012. Localization Strategies for Global E-Business. E-kirja. Cambridge University Press.

3 Korpela, Jukka. 2006. Unicode Explained. E-kirja. O'Reilly Media.

4 Esselink, Bert ; de Vries, Arjen-Sjoerd & O'Brien, Shiera. 2000. A Practical Guide to Localization. E-kirja. John Benjamins Publishing Company.

5 Dunne, Keiran J. 2006. Perspectives on Localization. E-kirja. John Benjamins Publishing Company.

6 Xu, Lucy. The 6 Biggest Challenges of Localization. 2018. Verkkoaineisto. Transifex. <https://www.transifex.com/blog/2018/6-biggest-challenges-of-localization-and-translation-management>. Luettu 10.2020.

7 Localization and Plurals. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Mozilla/Localization/Localization_and_Plurals>. Luettu 10.2020.

8 Pluralization (p11n) - the many of plurals. 2020. Verkkoaineisto. LingoHub. <https://lingohub.com/blog/2019/02/pluralization>. 13.1.2020. Luettu 10.2020.

9 Spelling Plurals with -s or -es. Verkkoaineisto. Grammarly. < https://www.grammarly.com/blog/spelling-plurals-with-s-es/>. Luettu 10.2020.

10 Date format by country. Verkkoaineisto. Calendar Wiki. https://calendars.wikia.org/wiki/Date_format_by_country. Luettu 10.2020.

11 Bugl, Daniel. 2019. Learn React Hooks. E-kirja. Packt Publishing.

12 Chinnathambi, Kirupa. 2016. Learning React. E-kirja. Addison-Wesley Professional.

13 Boduch, Adam. 2017. React and React Native. E-kirja. Packt Publishing.

14 Components and Props. Verkkoaineisto. React. <https://reactjs.org/docs/components-and-props.html>. Luettu 10.2020.

15 State and Lifecycle. Verkkoaineisto. React. <https://reactjs.org/docs/state-and-lifecycle.html>. Luettu 10.2020.

16 React Localize Redux Documentation. Ryan Johnson. Verkkoaineisto. <https://ryandrewjohnson.github.io/react-localize-redux-docs>. Luettu 10.2020.

17 FormatJS Documentation. Verkkoaineisto. <https://formatjs.io/>. Luettu 10.2020.

18 React i18next Documentation. Verkkoaineisto. < https://react.i18next.com/>. Luettu 10.2020.

19 O'Hagan, Minako & Mangiron, Carmen. 2013. Game Localization: Translating for the Global Digital Entertainment Industry. E-kirja. John Benjamins Publishing Company.

20 Dunne, Keiran J. & Dunne, Elena S. 2011. Translation and Localization Project Management: The Art of the Possible. E-kirja. John Benjamins Publishing Company.