Tampereen ammattikorkeakoulu

Frontend-teknologiamigraatio AngularJS:stä Reactiin

Nino Penttinen

OPINNÄYTETYÖ
Toukokuu 2022

Tieto- ja viestintätekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tieto- ja viestintätekniikka
Ohjelmistotekniikka

PENTTINEN, NINO:
Frontend-teknologiamigraatio AngularJS:stä Reactiin

Opinnäytetyö 36 sivua, joista liitteitä 1 sivu
Toukokuu 2022

Web-teknologiat kehittyvät nopeassa tahdissa. Jokin uusi, entistä lupaavampi teknologia siintää aina horisontissa. Samaan aikaan kun sovelluskehittäjät siirtyvät uudempien teknologioiden ja työkalujen käyttämiseen uusissa sovelluksissa, alkavat vanhemmat teknologiat jäämään yhä vain vähemmälle käytölle.

Opinnäytetyön tarkoituksena oli suorittaa AngularJS-React-sovelluskehysmigraatio loppuun VALTSU-nimiseen, keskisuureen tuotantokäytössä olevaan verkkosovellukseen. Työssä selvitettiin myös migraation syyt ja hyödyt sekä kuvattiin migraatioprosessi alusta loppuun.

Migraation pääasiallisena syynä oli sovelluksen elinkaaren pidentäminen. Toissijaisena syynä oli jatkokehityksen ja ylläpidon tehostaminen uudempien kehitystyökalujen ja teknologioiden avulla.

Migraatioprosessi itsessään oli monivaiheinen. Prosessi aloitettiin ottamalla käyttöön React2Angular-kirjasto, joka mahdollisti React-komponenttien upottamisen AngularJS-sovellukseen. Kun kaikki sovelluksen näkymät ja komponentit oli saatu uudelleenkirjoitettua Reactilla, luotiin sovellukselle uusi sovellusrunko Create React App -työkalun avulla. Lopuksi sovellus uudelleenrakennettiin liittämällä jo tehdyt React-komponentit osaksi sovellusta. Navigointi komponenttien välillä toteutettiin React Router -kirjaston avulla.

Asiasanat: web-kehitys, frontend, sovelluskehysmigraatio, AngularJS, React, TypeScript

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

PENTTINEN, NINO:
Frontend technology migration from AngularJS to React

Bachelor's thesis 36 pages, appendices 1 page
May 2022

Web technologies are advancing at a rapid rate. Some promising new technology is always waiting on the horizon. While the developers are moving to use newer technologies and tools with their new applications, the older technologies are becoming obsolete.

The purpose of this thesis was to migrate a midsized, in-production web application from the AngularJS framework to React. Moreover, the aim was to cover the reasons and benefits of doing so.

The main reason for the migration was to lengthen the application lifecycle. The secondary reason was to make development and maintenance easier and faster, with more modern development tools and technologies.

The migration process itself was multistaged. The process began with installation of the React2Angular library, which allowed React components to be embedded into an AngularJS application. Once all of the AngularJS components were rewritten with React, a base for the new React application was created with the Create React App tool. Finally, the application was recreated by attaching the finished React components to the application. Navigation between the components was implemented using the React Router library.

Key words: web development, frontend, framework migration, AngularJS, React, TypeScript

SISÄLLYS

LYHENTEET JA TERMIT

Frontend	Ohjelmistokoodi, jota suoritetaan käyttäjän selaimessa.
Backend	Ohjelmistokoodi, jota suoritetaan palvelimella.
Sovelluskehys	Sovelluskehitystä helpottamaan suunniteltu abstraktiokerros.
JavaScript	Komentokieli (engl. ”scripting language”), jota käytetään selaimissa tuomaan interaktiivisuutta verkkosivuihin.
HTML	Lyhenne sanoista ”HyperText Markup Language”. HTML on selaimessa käytettävä dokumenttipohja, jolla verkkosivut määrittelevät näytettävän sisällön.
DOM	Lyhenne sanoista “Document Object Model”. DOM on selaimessa HTML-dokumentin pohjalta rakentuva malli, jolla on rajapinta, jonka avulla dokumenttia voidaan manipuloida esimerkiksi JavaScriptillä.
SPA	Lyhenne sanoista ”Single Page Application”. SPA tarkoittaa verkkosovellusta, joka rakentuu kokonaan yhden HTML-dokumentin päälle.
Templaatti	Templaatilla tarkoitetaan HTML-mallipohjaa, johon lisätään ajonaikaisesti esimerkiksi muuttujien arvoja.
CSS	Lyhenne sanoista “Cascading Style Sheets”. CSS määrittelee sen, miten HTML-elementit tyylitellään.
NodeJS	JavaScript-ajoympäristö.
CoffeeScript	JavaScript-murre, joka käännetään ennen suoritusta JavaScriptiksi.
TypeScript	JavaScript-laajennus, joka tuo kieleen mm. tyypityksen. Käännetään ennen suoritusta JavaScriptiksi.
React	JavaScript-sovelluskehys/-kirjasto.
Angular	JavaScript-sovelluskehys.
AngularJS	AngularJS tarkoittaa Angularin 1.x versioita. JS-loppuliite jäi pois Angular versiosta 2 eteenpäin.

1 JOHDANTO

Web-teknologiat kehittyvät nopeassa tahdissa. Jokin uusi, entistä lupaavampi teknologia siintää aina horisontissa. Samaan aikaan kun sovelluskehittäjät siirtyvät uudempien teknologioiden ja työkalujen käyttämiseen uusissa sovelluksissa, alkavat vanhemmat teknologiat jäämään yhä vain vähemmälle käytölle. Seurauksena näiden teknologioiden ylläpidolliset resurssit laskevat.

Pitkän elinkaaret omaavat verkkosovellukset tulevat ennemmin tai myöhemmin törmäämään tilanteeseen, jossa sovelluksen käyttämät teknologiat alkavat vanhentua. Toisinaan vanheneminen näkyy siinä, että teknologiat eivät vain enää pysty vastaamaan sovelluksen kasvaviin tarpeisiin, ja niillä kehittäminen on hidasta. Mutta joskus teknologian tuki saatetaan myös lakkauttaa kokonaan, kuten kävi esimerkiksi aikoinaan suositun AngularJS:n kanssa. Tuen loppuminen tarkoittaa sitä, että mikäli teknologiasta löydetään tulevaisuudessa haavoittuvuuksia, ei niitä enää korjata. Tällöin kyseistä teknologiaa käyttävät sovellukset voivat jäädä haavoittuviksi. Riippuen sovelluksen koosta, voi teknologiamigraatioprosessi olla pitkä ja kallis, joten on hyvä toimia ennakoivasti.

Tässä opinnäytetyössä käydään suurpiirteittäin läpi, miten yhdessä tuotantokäytössä olevassa verkkosovelluksessa vaihdettiin selainpuolen sovelluskehys AngularJS:stä Reactiin. Työssä käydään myös läpi migraation syyt, sekä mitä hyötyjä, elinkaaren pidentämisen lisäksi, migraatiolla pyrittiin saavuttamaan. Toisena osiona työssä käydään läpi migraatioprosessin eri työvaiheita, käytännön toteutusta ja sitä, miten uusi sovelluskehys saatiin pienissä osissa ketterästi siirrettyä tuotantokäyttöön.

Tämän työn tavoitteena on tarjota yleishyödyllinen katsaus toimintamalleihin, joita voidaan käyttää selainpuolen sovelluskehysmigraation toteuttamisessa.

2 TYÖN TARKOITUS

Opinnäytetyön tarkoituksena oli suorittaa AngularJS-React-migraatio loppuun keskisuureen tuotantokäytössä olevaan verkkosovellukseen, sekä kuvata prosessin kulku alusta loppuun. Lisäksi käydään läpi syyt migraation aloittamiselle sekä uusille teknologiavalinnoille.

2.1 Sovellus

Sovellus, johon migraatio tehtiin, on VALTSU-niminen verkkosovellus, jota käytetään raideliikenteen kaluston valvontaan (Väylävirasto 2021). Sovellus on otettu ensimmäistä kertaa tuotantokäyttöön vuonna 2015, ja on ollut siitä lähtien aktiivisessa jatkokehityksessä.

VALTSU:lla on noin 5–10 aktiivista käyttäjää, sekä satunnaisia käyttäjiä arviolta noin 10–20. VALTSU:n tarjoamaa dataa käytetään myös analytiikka tarpeisiin. Vaikka sovelluksen käyttäjäkunta onkin suhteellisen pieni, on VALTSU kuitenkin keskeinen työkalu tiettyjen käyttäjäryhmien työssä.

VALTSU on raideliikenteen turvallisuuden, ja ratainfrastruktuurin kunnon kannalta merkityksellinen työkalu. Migraation aikana erityisesti VALTSU:n keskeisimpien toimintojen osalta huolellisuus ja vikatilanteisiin varautuminen olivat ensisijaisen tärkeitä.

2.2 Migraation syyt

Migraatiota aloitettiin alun perin tekemään jo marraskuussa 2019 sovelluksen toimintakyvyn kehittämisen/ylläpidon budjetissa pienellä resurssilla. Lisäresurssien tarpeellisuudesta tehtiin päätös elokuussa 2020, jolloin migraatio muuttui toimintakyvyn kehityksestä omaksi epicikseen. (Sorje 2022.) Päätöksen keskeisimpänä syynä oli Googlen ilmoitus AngularJS:n tuen loppumisesta 30.6.2021 (Darwin

2018). Tuen loppuminen tarkoittaa sitä, että mikäli sovelluskehyksestä löydettäisiin tulevaisuudessa haavoittuvuuksia, ei niitä tultaisi enää korjaamaan, jolloin sitä käyttävät sovellukset altistuisivat tietoturvauhille. Google ilmoitti 27.7.2020 AngularJS:n tuen jatkumisesta vielä vuoden 2021 loppuun asti, joka tarjosi lisäaikaa migraation loppuun saattamiselle (Thompson 2020).

Muita syitä migraation toteuttamiselle olivat sovelluksen käyttämän NodeJS 8:n päivittäminen, vanhentuneista teknologioista syntyneen teknisen velan kuittaaminen, sekä ylläpidon helpottuminen uudempien työkalujen ansiosta. NodeJS 8:n tuki päättyi 31.12.2019 (Red Hat 2019), tämä huomattiin samassa yhteydessä kuin AngularJS:n vanhentuminen. Liian vanha Angular versio oli kuitenkin tuolloin estänyt NodeJS:n version nostamisen.

2.3 Migraation lähtötilanne ja tavoitteet

Migraation alussa sovelluksessa oli 137 Jade-templaattia, joka tarkoitti suurin piirtein samaa määrää AngularJS-komponentteja, ja 216 CoffeeScript-tiedostoa. Alkuperäinen optimistinen työarvio oli, että yhden AngularJS-komponentin uudelleenkirjoittaminen Reactilla kestäisi keskimäärin noin yhden henkilötyöpäivän. Pessimistisessä arviossa yhdessä komponentissa kestäisi noin kolme henkilötyöpäivää. Kaiken kaikkiaan optimistinen työarvio oli siis noin 137 henkilötyöpäivää, ja pessimistinen 411 henkilötyöpäivää. (Liite 1.)

Migraation päällimmäisenä tavoitteena oli sovelluksen elinkaaren pidentäminen, ja toissijaisena sovelluksen ylläpidon helpottaminen. Ylläpito tulisi helpottumaan, kun vanhalla teknologialla vuosien aikana tehty koodi saadaan korvattua uusilla, varttuneemmilla teknologioilla kirjoitetulla koodilla. Lisäksi uutena teknologiana käyttöönotettu TypeScript tulisi tarjoamaan koodiin parempaa luettavuutta, sekä vähentämään virheherkkyyttä tyypityksen ansiosta. Samalla migraatioprosessin aikana pystyttiin arvioimaan eri komponenttien tarpeellisuutta. Vuosien aikana turhaksi jääneitä tai koettuja ominaisuuksia karsittiin, ja monia tarpeettoman monimutkaiseksi tehtyjä komponentteja virtaviivaistettiin tekemällä niistä yksinkertaisempia ja geneerisempiä.

Migraation kanssa samoihin aikoihin tehtiin haastatteluja eri käyttäjäryhmille, joissa kartoitettiin heidän tyytyväisyyttään sekä toiveitaan heidän käyttämiensä sovellusten osalta. Migraatio tarjosi hyvän tilaisuuden toteuttaa pieniä käyttöliittymäparannuksia käyttäjien palautteiden perusteella, koska sovelluksen eri näkymät oltiin tekemässä uusiksi joka tapauksessa.

3 TEKNOLOGIAT

Tässä luvussa käydään läpi migraation yhteydessä poistuneet teknologiat, sekä niiden korvaajat. Kerrotaan myös migraation aikana käytetystä React2Angular-kirjastosta.

3.1 Poistuvat teknologiat

Migraation yhteydessä päätetiin uusia samalla kertaa koko selainpuolen teknologiapaletti. CoffeeScript-murteen käytöstä luovuttiin, ja sen tilalla alettiin käyttämään TypeScriptiä. Lisäksi luovuttiin Jade-templaattien käytöstä, Stylus CSS-esiprosessori vaihdettiin SCSS:ään, http-client vaihdettiin RestAngularista Axiokseen, ja projektin selainpuolen rakentamiseen käytetty Gulp vaihtui Webpackiin.

Syynä AngularJS:stä luopumiseen, uudempaan versioon päivittämisen sijasta oli se, että entinen koodi olisi jouduttu joka tapauksessa kirjoittamaan hyvin pitkälti uusiksi. Korvaajaksi valikoitui lopulta React hyvin yksinkertaisin perusteluin – sitä on miellyttävämpää kirjoittaa, ja se on ollut jo vuosia yksi ylivoimaisesti suosituimmista selainpuolen sovelluskehyksistä/kirjastoista (Stack Overflow 2021).

CoffeeScript ja Jade-templaatit poistuivat käytöstä johtuen niiden huonosta yhteensopivuudesta Reactin ja TypeScriptin kanssa. Lisäksi kumpikaan em. teknologioista ei ole nauttinut enää kovin suurta suosiota vuosiin, ja niiden kehittäminen on jäänyt taka-alalle. Niiden käyttö myös tuottaa ylimääräisen kerroksen kompleksisuutta ja opeteltavaa niille, jotka eivät ole kyseisiä teknologioita aiemmin käyttäneet. Näistä syistä johtuen CoffeeScript ja Jade eivät välttämättä ole ideaaleja vaihtoehtoja pitkän elinkaaren omaavaan sovellukseen, jossa kehittäjät vaihtuvat aika ajoin, vaan sopivat paremmin esimerkiksi omiin harrasteprojekteihin.

Myös muissa teknologia valinnoissa perimmäisenä syynä oli siirtyä suositumpiin ja laajemmin käytettyihin teknologioihin, joita kehitetään aktiivisemmin, ja joiden

käyttämiseen löytyy paremmin dokumentaatiota ja tukea internetistä. Myös tulevaisuutta ajatellen on edullisempaa käyttää teknologioita, joihin löytyy helpommin osaavia tekijöitä siinä vaiheessa, kun projektitiimin jäsenet vaihtuvat.

3.1.1 AngularJS

AngularJS on Googlen vuonna 2010 julkaisema JavaScript-sovelluskehys, joka suunniteltiin helpottamaan ja nopeuttamaan dynaamisten verkkosovellusten kehittämistä. AngularJS:llä tarkoitetaan nimenomaan Angularin versioita 1.x. JS-loppuliite jäi nimestä pois Angular versiosta 2 eteenpäin. Nimenmuutoksen syynä oli Angularin täysi uudelleenkirjoitus, ja siitä seuranneet huomattavat muutokset versioiden 1 ja 2 välillä. (AngularJS 4U 2017.)

AngularJS on sen kehittäjien omin sanoin vapaasti suomennettuna ”se mitä HTML olisi, jos se olisi suunniteltu sovelluksille” (Google n.d.). Tällä viitataan siihen, että HTML oli alun perin suunniteltu käytettäväksi staattisiin verkkosivuihin, joissa palvelin vastaa käyttäjän syötteisiin. Dynaamiset verkkosivut, eli verkkosovellukset, puolestaan mahdollistavat käyttäjien aktiivisen vuorovaikuttamisen sivuston kanssa JavaScriptin avulla.

AngularJS mahdollistaa HTML:n käyttämisen mallikielenä, samalla laajentaen HTML:n syntaksia Angularin ”direktiiveillä”, jotka mm. auttavat käsittelemään sovelluksen komponentteja dynaamisesti. Direktiivit ovat merkkejä DOM-elementeissä, kuten elementin nimi, attribuutti, kommentti tai CSS-luokka, jotka kertovat Angularin HTML-kääntäjälle, millainen toiminto siihen kohtaan DOM:ia täytyy lisätä. Lisättäviä toimintoja voivat olla esimerkiksi tapahtumaseuraimet (engl. ”event listener”), tai vaikka täysin uudet DOM-elementit. (Google n.d.) Tämä säästää kehittäjien aikaa, kun monet toiminnallisuudet, jotka normaalisti pitäisi lisätä manuaalisesti ja vaatisivat useita rivejä ns. ”boilerplate” koodia, saadaankin nyt otettua helposti käyttöön vain lisäämällä uusi direktiivi HTML-templaattiin.

3.1.2 React2Angular

React2Angular on kirjasto, jonka avulla React-komponentteja voi upottaa AngularJS-sovellukseen. React2Angular ei käännä React-komponentteja Angulariksi, vaan auttaa AngularJS-riippuvuuksien injektoimisessa React-komponenteille, jolloin niitä voi käyttää helposti AngularJS:n kanssa rinnakkain (coatue-oss 2019). Esimerkiksi AngularJS:n navigointia voidaan käyttää React-komponenttien renderöintiin. Lopputuloksenahan syntyy vain HTML:ää ja JavaScriptiä, josta osa on AngularJS:n luomaa, ja osa React-komponenttien.

3.2 React

React on Metan (aiemmin Facebook) vuonna 2013 julkaisema JavaScript-kirjasto. Julkaisunsa jälkeen React on noussut suureen suosioon web-kehittäjien keskuudessa, ja se onkin monien mittausten perusteella ollut käytetyin verkkosovelluskehys viime vuosina (Stack Overflow 2021).

React on suunniteltu erityisesti käyttöliittymien tekemiseen. Vaikka Reactista usein puhutaankin sovelluskehyksenä, se ei todellisuudessa kuitenkaan ole täysiverinen sovelluskehys. React huolehtii ainoastaan käyttöliittymän renderöimisestä, ja mm. sovelluksen reititys mahdollisuudet puuttuvat Reactista täysin. (Baer 2018.) Tämä antaa kehittäjille vapauden valita itse omat työkalunsa, ja mahdollistaa Reactin käyttämisen sovelluksessa vain osittain.

React-koodi on näennäisesti sekoitus JavaScriptiä ja HTML:ää. Tämän mahdollistaa Reactin tarjoama JSX-formaatti. JSX tulee sanoista ”JavaScript Syntax Extension”, se on nimensä mukaisesti syntaktinen laajennus JavaScript-kieleen, joka tulkitsee HTML:n näköisen koodin HTML:ää luoviksi React-metodeiksi (Meta 2022). JSX:n käyttäminen React-komponenteissa ei ole pakollista, mutta luettavuuden kannalta suotavaa, mikäli sen käyttäminen on mahdollista. React-projekteissa, joissa on TypeScript mukana käytetään usein TSX-formaattia, joka yhdistää sekä TS-, että JSX-formaatit.

```
return (
  <div>
    Seconds: {this.state.seconds}
  </div>
);
```

```
return React.createElement(
  "div",
  null,
  "Seconds: ",
  this.state.seconds
);
```

KUVA 1. Sama paluuarvo JSX:llä (vasen) ja perinteisellä JS:llä (oikea).

Käyttöliittymä luodaan Reactissa komponenttien avulla ns. ”puumalilla”. React-komponentit saadaan otettua käyttöön lisäämällä ensin juurikomponentti sovellukseen React-DOM-luokan render-metodin avulla. Metodille annetaan argumenteiksi haluttu juurikomponentti, sekä HTML-elementti, jonka sisälle juurikomponentti halutaan liittää ajonaikana DOM:issa. Kaikki muut komponentit rakentuvat tämän juurikomponentin päälle lapsikomponentteina.

Komponentit itsessään ovat uudelleenkäytettäviä ja huolehtivat lähtökohtaisesti itse omasta tilastaan. Jokaisella komponentilla on siis oma tilansa, sekä rajapinta, jolla tilaa voidaan hallita. Ennen Reactin versiota 16.8, kaikki React-komponentit olivat luokkapohjaisia. Luokkapohjaisissa komponenteissa tilaa voidaan hallita manuaalisesti kutsumalla React-kantaluokan (engl. ”base class”) setState-metodia. Luokkapohjaisissa komponenteissa on myös käytettävissä niin sanotut elinkaarimetodit, jotka suoritetaan objektien elinkaaren eri vaiheissa. (Meta n.d.)

Versiossa 16.8 tulivat React Hookien myötä funktionaaliset React-komponentit. Funktionaaliset React-komponentit ovat JavaScript-funktioita, jotka palauttavat React-komponentin. Funktionaalisissa React-komponenteissa tilaa hallitaan hookien avulla. Hookit mahdollistavat Reactin toimintojen käyttämisen ilman React-kantaluokkaa. (Edpresso Team n.d.)

Komponenttien tilan muuttuminen on välttämätöntä sovelluksen vuorovaikutteisuuden kannalta. Koska koko käyttöliittymän uudelleenrenderöiminen jokaisen tilan muutoksen yhteydessä olisi laskennallisesti kuitenkin kallista, käytetään Reactissa suorituskyvyn parantamiseksi virtuaalista DOM:ia. Virtuaali-DOM on Reactin muistiin luoma datarakenne, joka kuvaa sovelluksen sen hetkistä DOM:ia. Reactin tilan muuttuessa React tarkastaa, onko virtuaali-DOM:issa tapahtunut muutoksia, ja uudelleenrenderöi vain sen osan DOM:ia, mikä on muuttunut edelliseen tilaan verrattuna. (Meta n.d.)

3.2.1 Create React App

Create React App, eli lyhyesti CRA on helppo ja nopea tapa aloittaa Reactin käyttö, se asentaa ja konfiguroi yhdellä komennolla erinomaisen moderniin web-kehitykseen suunnitellun React-kehitysympäristön. CRA tarjoaa myös skriptit, joilla koodin saa käännettyä helposti tuotanto kelpoiseen kuntoon.

CRA käyttää ns. ”pellin alla” Babelia ja Webpackia. Babel kääntää ECMAScript 2015+ mukaisen JavaScript-koodin vanhempaan JavaScriptiin, joka toimii myös joidenkin selainversioiden käyttämissä vanhemmissa JavaScript-moottoreissa. Webpack puolestaan hoitaa JavaScript-moduulien niputtamisen (engl. "bundling”) yhdeksi, optimoiduksi tiedostoksi. (Meta n.d.)

Erityisesti valmis Webpack-konfiguraatio on suuri etu, mikäli tiimissä ei ole ennalta Webpack osaamista. Haittapuolena CRA:n käytössä on se, että konfiguraatioiden muuttaminen on silloin hankalampaa. Esimerkiksi Webpack-konfiguraatio ei ole suoraan saatavilla, vaan jos siihen haluaa tehdä muutoksia, täytyy konfiguraatiot purkaa CRA:n tarjoamalla react-scripts eject -komennolla, jolloin CRA lakkaa ylläpitämästä konfiguraatioita. Purkamisen voi tosin myös kiertää käyttämällä esimerkiksi Craco-nimistä kirjastoa, joka osaa ujuttaa käyttäjän omia konfiguraatioita CRA:n ylläpitämien sekaan.

3.2.2 React Router

React Router on Reactille luotu kirjasto, joka on suunniteltu erityisesti selainpuolen reitittämiseen. React Router auttaa pitämään sovelluksen käyttöliittymän synkronoituna URL:än kanssa. (freeCodeCamp 2016.)

3.3 TypeScript

TypeScript on Microsoftin vuonna 2012 julkaisema JavaScript-laajennus, joka lisää kieleen staattisen tyypityksen. TypeScript käännetään ohjelman käännösvaiheessa JavaScriptiksi ja ei siis täten tarjoa ajonaikaista dynaamista tyyppien tarkastusta.

Tarve JavaScriptin tyypittämiselle syntyi, kun verkkosivuista alkoi kehittymään yhä isompia ja monimutkaisempia sovelluksia. Alun perin pelkäksi komentokieleksi suunniteltu JavaScript ei enää riittänyt isojen sovellusten tarpeisiin, vaan kieleen tarvittiin enemmän tyyppivarmuutta, jotta voitiin paremmin välttyä aiheuttamasta bugeja sovellukseen. (Microsoft n.d.)

TypeScript koostuu kahdesta eri osasta, itse TypeScript-kielestä, ja TypeScript-kääntäjästä (engl. ”compiler”), jota kutsutaan TSC:ksi. TSC on kääntäjä, joka tarkastaa tyyppisidokset, ja kääntää TypeScript-tiedostot JavaScriptiksi. (Microsoft n.d.)

Koska TypeScript on vain JavaScriptiä laajennetulla syntaksilla, voidaan TypeScript ottaa käyttöön jo olemassa olevissa JavaScript-koodeissa. TypeScript ei pakota käyttäjäänsä noudattamaan kaikkia TypeScriptin sääntöjä, vaan käyttäjä voi itse määrittää, miten haluaa TypeScriptin toimivan. Määrittelyt tehdään TypeScriptin konfiguraatiotiedostossa tsconfig.json.

Konfiguraatiotiedostossa määritetään asetukset TSC:lle. Määriteltäviä asetuksia ovat mm. lähdekoodien hakemisto, käännettyjen tiedostojen kohdehakemisto, ja muita itse kääntämiseen liittyviä asetuksia, kuten poistetaanko kommentit käännetyistä tiedostoista. Lisäksi voidaan määrittää joitakin tyypitykseen liittyviä sääntöjä, kuten hyväksytäänkö any-tyyppiä. (Microsoft n.d.)

3.4 SCSS

SCSS (”Sassy CSS”) on CSS-esiprosessori. Esiprosessorilla tarkoitetaan ohjelmaa, joka laajentaa CSS:n syntaksia ja kääntää SCSS-koodin CSS:ksi. SCSS

pyrkii tekemään CSS:n kirjoittamisesta ja ylläpitämisestä helpompaa, tuomalla siihen hyödyllisiä uusia toimintoja, kuten muuttujat, sisäkkäiset luokat, sekä uudelleenkäytettävät mixin-luokat. (Sass team n.d.)

4 MIGRAATION TOTEUTUS

4.1 Aloitus

Migraation tiedettiin alusta alkaen olevan pitkä ja aikaa vievä prosessi, joten koko selainpuolen koodin uudelleenkirjoittaminen kerralla valmiiksi ei ollut realistinen tavoite. Koodin uudelleenkirjoittaminen omassa haarassaan aiheuttaisi lisäksi sen, että se tulisi eittämättä olemaan vanhentunutta siinä vaiheessa, kun se lopulta mergettäisiin primääriseen kehityshaaraan. Riskienhallinnan näkökulmasta oli myös parempi, että uudet Reactilla tehdyt näkymät ja komponentit saataisiin vietyä tuotantokäyttöön pienemmissä osissa.

Vanhaa koodia haluttiin siis alkaa uusimaan pienissä palasissa, ja uudet toiminnallisuudet haluttiin alkaa kirjoittamaan suoraan Reactilla. Tätä varten otettiin käyttöön React2Angular-kirjasto, joka mahdollisti uusien React-komponenttien upottamisen AngularJS-sovellukseen. React2Angular-kirjaston käyttöönoton myötä kehitystä siirryttiin tekemään Reactilla ja TypeScriptilla.

4.1.1 TypeScript käyttöönotto

TypeScriptin käyttöönottamiseksi täytyi sovelluksen rakentamisesta vastaavaan Gulp-työkaluun tehdä muutama konfiguraatiomuutos. Gulpin käyttämä, riippuvuuksien ajonaikaiseen ympäristöön niputtamisesta vastaava Browserify-liitännäinen muokattiin hyväksymään coffee-päätteisten tiedostojen lisäksi myös js-, ts- ja tsx-päätteisiä tiedostoja. Lisäksi Browserify-liitännäiseen otettiin käyttöön toinen Tsify-niminen liitännäinen. Tsify käyttää Browserifyn API:a lisätäkseen siihen toiminnallisuuden, joka kääntää TypeScript-moduulit JavaScriptiksi ennen niputtamista.

```
initBrowserify = ->
  options = {}
  options.debug = true
  options.extensions = ['.coffee', '.js', '.jsx', '.ts', '.tsx']
  options.cache = {} # For watchify
  options.packageCache = {} # For watchify
  return browserify(paths.app.source, options).plugin(tsify, { noImplicitAny: true })
```

KUVA 2. Browserify konfiguroitu käyttämään Tsify-liitännäistä.

TypeScriptin konfigurointia varten luotiin tiedosto tsconfig.json. Tässä JSON-muotoisessa tiedostossa määritetään mm. TypeScriptille säännöt, joiden mukaan TypeScriptin halutaan toimivan.

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "allowJs": true,
    "checkJs": false,
    "jsx": "react",
    "outDir": "./build",
    "rootDir": "./src/react/",
    "removeComments": true,
    "forceConsistentCasingInFileNames": true,
    "noImplicitReturns": true,
    "noUnusedLocals": true,
    "strictNullChecks": true,
    "noImplicitAny": false,
    "baseUrl": "./",
    "preserveConstEnums": true,
    "sourceMap": true,
    "lib":[
      "es2015.promise", "es5", "dom","es2015","es2017.object", "scripthost"
    ]
  },
  "include": [
    "./lib/**/*"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

KUVA 3. TypeScriptin konfiguraatiotiedosto migraation alkuvaiheessa.

4.2 Näkymien migrointi

Näkymien ja komponenttien migraatio aloitettiin tekemällä ensin kokeiluksi jokin pieni ja sovelluksen toiminnan kannalta vähemmän kriittinen komponentti. Tämän jälkeen seuraavat migroitavat näkymät/komponentit päätettiin sen perusteella, mitä uusia toiminnallisuuksia sovellukseen oli tulossa, tai mitkä olivat työmäärä arvioltaan sopivan kokoisia tehtäväksi muun kehitystyön välissä. Rajapintakutsuista vastaavien palvelujen (engl. ”service”), sovelluksen navigaation ja käännöskirjaston migroinnit jätettiin tehtäväksi viimeisinä.

Seuraavassa luvussa katsotaan esimerkki siitä, miten yksi näkymä migroidaan AngularJS:stä Reactiin.

4.2.1 Esimerkki yhden näkymän migroinnista

Näkymän migrointi aloitetaan lisäämällä AngularJS-koodikantaan uusi moduuli. Uudelle moduulille annettiin nimeksi passingr, joka kuvaa näkymän tehtävää. Seuraavaksi moduuliin lisätään komponenttidirektiivi passingrListing, joka saa argumenttinaan React-komponentin, johon injektoidaan sen vaatimat riippuvuudet React2Angular-kirjaston avulla. Tämän komponentin osalta vaadittavia riippuvuuksia ovat eri rajapintakutsuja tekevät palveluluokat, sovelluksen navigointiin vaadittava state-objekti, sekä Angular Translate -kirjaston tarjoama translate-objekti.

```
react2angular = require 'react2angular'
passingrListing = (require '../../react/reports/passing/PassingListing').default

module = angular.module 'passingr', []

module.component 'passingrListing', react2angular.react2angular(passingrListing, ['store'], ['$translate',
  '$state', 'TrainPassingService', 'StationsService', 'OperatorService', 'VekuService'])
```

KUVA 4. React-komponentti liitetty AngularJS-moduuliin.

Lopuksi moduuli konfiguroidaan käyttämään Angular-ui-router-kirjaston tarjoamaa tilanhallintaa stateProvider-objektin avulla. Tilanhallintaobjektille lisätään kaksi uutta mahdollista tilaa, passing_beta ja passing_beta.listing. Tila pas-

sing_beta otetaan käyttöön käyttäjän siirtyessä ”/passing_beta”-URL-osoitteeseen, tämä tila ainoastaan uudelleenohjaa sovelluksen käyttämään passing_beta.listing-tilaa. Tämä tehdään siksi, että sovellus voi siirtyä ”/passing_beta”-osoitteeseen myös ilman URL-parametrejä. Sovelluksen siirtyessä passing_beta.listing-tilaan, renderöidään passingrListing-komponentti Angular-ui-router-kirjaston tarjoamaan ui-view-näkymään, joka on upotettuna sovelluksen HTML-templaattiin.

```
react2angular = require 'react2angular'
passingrListing = (require '../../react/reports/passing/PassingListing').default

module = angular.module 'passingr', []

module.component 'passingrListing', react2angular.react2angular(passingrListing, ['store'], ['$translate',
  '$state', 'TrainPassingService', 'StationsService', 'OperatorService', 'VekuService'])

module.config ['$stateProvider', ($stateProvider) ->

  $stateProvider.state 'passing_beta',
    url:          '/passing_beta'
    redirectTo:   'passing_beta.listing'

  $stateProvider.state 'passing_beta.listing',
    url: '/?page&sortOrder&orderBy'
    template: '<passingr-listing store="store" />'
    controller: ($scope, $ngRedux) ->
      $scope.store = $ngRedux

]
```

KUVA 5. Valmiiksi määritelty AngularJS-moduuli.

Uusi moduuli otetaan käyttöön lisäämällä se sovelluksen käyttämään päämoduuliin. Päämoduuli toimii kaikkien muiden moduulien juurena, ja pitää mm. huolen näkymien välisestä navigoinnista.

Nyt kun React-komponentin vastaanottava AngularJS-moduuli on määritelty, voidaan näkymä uudelleenkirjoittaa Reactilla. React2Angularin injektoimat riippuvuudet voidaan ottaa React-komponentille sisään normaaliin tapaan argumentteina.

```
interface Props {
  $state: any
  $translate: Translate
  TrainPassingService: TrainPassingService
  StationsService: StationsService
  OperatorService: OperatorService
  VekuService: VekuService
}

const PassingListing = (props:Props) => {
  const { $state, TrainPassingService, StationsService, OperatorService, VekuService } = props
  const translate = props.$translate
```

KUVA 6. AngularJS:stä injektoitujen riippuvuuksien käyttö React-komponentissa.

4.2.2 Uusien näkymien tuotantoon vienti

Riskienhallitsemiseksi uudet Reactilla kirjoitetut näkymät otettiin tuotantokäyttöön limittäin vanhojen näkymien kanssa. Ensiksi uudet näkymät otettiin käyttöön beta-versioina, jota käyttäjät saivat halutessaan kokeilla. Tietyn ajan kuluttua, uudet näkymät tulivat käyttöön oletuksina, mutta niissä oli edelleen linkki vanhaan versioon, joka oli nyt siirretty eri URL:än taakse. Näkymät olivat useimmiten limittäin käytössä muutamista viikoista kuukausiin, riippuen näkymien tärkeydestä ja tuotantoasennuksien aikatauluista.

Syynä tälle menettelylle oli se, että erityisesti sovelluksen tärkeimmille näkymille ja toiminnoille saataisiin ensin käytännön testausta ennen vanhoista näkymistä luopumista siltä varalta, että niistä löytyisi käyttöä estäviä bugeja. Tämä oli erityisesti tärkeää, sillä sovelluksessa ei ollut minkäänlaisia UI-testejä.

4.3 Sovellusrungon vaihto

Sovellusrungon vaihto aloitettiin, kun kaikki näkymät oli saatu uudelleenkirjoitettua Reactilla. Sovellusrungon vaihto tehtiin omassa haarassaan.

4.3.1 Sovellusrungon luonti

Ensimmäiseksi luotiin uusi React-projekti CRA:lla, komennolla npx create-react-app valtsu-ui-react, jossa valtsu-ui-react oli hakemiston nimi, johon uusi React-projekti luotiin. Vanha AngularJS-projekti säilytettiin toistaiseksi hakemistossa valtsu-ui. Komennon ajamisen jälkeen poistettiin kaikki CRA:n luomat projektille turhat kirjastot ja tiedostot. CRA:n luomasta .gitignore-tiedostosta kopioitiin tarpeelliset rivit jo olemassa olevaan .gitignore-tiedostoon.

KUVA 7. CRA:lla luotu hakemisto ylimääräisten tiedostojen poiston jälkeen.

React-projektin luomisen jälkeen asennettiin siihen TypeScript komennolla yarn add typescript --save. TypeScriptin konfiguraatiotiedosto haettiin vanhalta AngularJS puolelta. Seuraavaksi React puolelle kopioitiin kaikki AngularJS puolelle tehdyt React- ja SCSS-lähdekoodit, sekä assets-hakemisto, joka sisältää sovelluksen käyttämät fontit ja kuvatiedostot.

Lähdekoodien siirron jälkeen luotiin sovelluksen runko kääntämällä vanha index.jade-templaattitiedosto HTML:ksi, ja tekemällä siihen tarvittavat muutokset. Uusi index.html-tiedosto siirrettiin valtsu-ui-react/public-hakemistoon, josta CRA odottaa sen löytyvän. Koko sovellus rakentuu tämän HTML-pohjan päälle. React saadaan sovellukseen käyttöön index.js-tiedostosta löytyvän ReactDOM-luokan render-metodin avulla.

Käynnistämällä sovellus komennolla yarn start, alettiin saamaan virheviestejä puuttuvista riippuvuuksista, määrittelemättömistä TypeScript-luokista ja muista TypeScript-virheistä, joita ei aiemmin tullut, kun projekti käännettiin Gulpilla. Virheitä aloitettiin korjaamaan yksi kerrallaan ja puuttuvat riippuvuudet asennettiin sitä mukaan, kun niitä löydettiin. TypeScript-tyyppivirheet johtuivat myöskin

useimmiten puuttuvista riippuvuuksista. Useat kirjastot nimittäin tarjoavat TypeScript-tyypit erillisinä riippuvuuksina, jotta niitä ei tarvitse turhaan ladata projekteihin, jotka eivät käytä TypeScriptiä. Tyyppikirjastot ovat useimmiten nimetty seuraavalla tavalla ”@types/kirjaston_nimi”.

SCSS lisättiin projektiin yksinkertaisesti importoimalla SCSS-tyylien indeksitiedosto style.scss projektin index.js-tiedostossa. Koska projektissa on käytössä myös Bootstrap, on tärkeää importoida SCSS-tiedosto vasta Bootstrapin jälkeen, jotta projektin omat SCSS-tyyliluokat ylikirjoittavat Bootstrapin tyyliluokat.

4.3.2 React-i18next käännöskirjaston käyttöönotto

React puolella sovelluksen käännöksistä vastaamaan valittiin React-i18next-kirjasto, joka on React versio suositusta i18next-kirjastosta. Kirjasto asennettiin komennolla yarn add react-i18next i18next –save. Asennuksen jälkeen kirjasto alustettiin dokumentaation ohjeiden mukaisesti.

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import { fi } from './fi';

const resources = {
  fi: {
    translation: fi
  }
}

i18n
  .use(initReactI18next)
  .init({
    resources,
    lng: 'fi',
    interpolation: {
      escapeValue: false // not needed for react as it escapes by default
    },
  })

export default i18n;
```

KUVA 8. Käännösobjektin alustus käyttöönottovaiheessa.

AngularJS puolella käytetty Angular-translate-käännöskirjasto käyttää käännöksiinsä JavaScript-objektia, jossa jäsenmuuttujien nimet toimivat ikään kuin avain-arvo-parin avainsanoina itse käännösteksteille. React-i18next toimii samalla periaatteella, joten vanha käännöstiedosto fi.coffee käännettiin vanhasta CoffeeScript-murteesta JavaScriptiksi käyttämällä js2coffee-käännöstyökalua.

4.3.3 Näkymien lisääminen ja navigointi

Nyt kun sovellukselle oli saatu jonkinlainen toimiva React-runko valmiiksi, oli aika uudelleenrakentaa sovellus toistaiseksi vielä irtonaisista React-komponenteista. Koska sovellus on SPA, joka sisältää useita eri näkymiä, on sovellukseen luotava dynaaminen navigointi. Tähän otettiin avuksi React Router, joka on de facto -standardi kirjasto React-sovellusten reititys tarpeisiin.

React Routerin käyttöönotto aloitettiin asentamalla se komennolla yarn add react-router-dom --save. Asentamisen jälkeen tuotiin kirjasto sovelluksen App-juurikomponenttiin. Aluksi reitit luotiin vain sovelluksen ylätason näkymille, ja reitit laitettiin väliaikaisesti palauttamaan vain yksinkertainen div-elementti. Tämä tehtiin siitä syystä, että näkymiä oli silloin helpompi alkaa lisäämään yksi kerrallaan, sillä tiedettiin, että niiden lisääminen ei tulisi onnistumaan ilman käännösvirheitä.

```
return (
  <Router basename="valtsu/">
    <div className="page-container">
      <div className="topbar">
        <Navbar user={{username: 'PLACEHOLDER'} as User}/>
      </div>
      <div className="alert-fixed">
        <div className="alert alert-success fade in">
          <div className="alert-content"></div>
        </div>
      </div>
      <div className="content-container" ui-view=''>
        <Routes>
          <Route path={''} element={<div>tilanne</div>} />
          <Route path={'incident/'} element={<div>tehtävät</div>}/>
          <Route path={'alarms/'} element={<div>hälytykset</div>}/>

          <Route path={'passing/'} element={<div>ohitukset</div>}/>
          <Route path={'rfid/'} element={<div>RFID</div>}/>
          <Route path={'laku/'} element={<div>LAKU</div>}/>
          <Route path={'veku/'} element={<div>VEKU</div>}/>
          <Route path={'apms/'} element={<div>APMS</div>}/>
          <Route path={'proku/'} element={<div>PROKU</div>}/>
          <Route path={'tbogi/'} element={<div>TBOGI</div>}/>
          <Route path={'cvision/'} element={<div>CVISION</div>}/>

          <Route path={'sensors/'} element={<div>sensorit</div>}/>
          <Route path={'vehicles/'} element={<div>kalustot</div>}/>
          <Route path={'weight/'} element={<div>kalibrointi</div>}/>
          <Route path={'applianceroom/'} element={<div>laitetilahälytykset</div>}/>
          <Route path={'admin/'} element={<div>ylläpito</div>}/>
        </Routes>
      </div>
      {/*modal*/}
    </div>

    <AppFooter/>

  </Router>
```

KUVA 9. React Router otettu käyttöön App-komponentissa.

Ennen kuin varsinaisia näkymiä alettiin lisäämään sovellukseen, täytyi ensin kuitenkin refaktoroida käyttöliittymän navigoinnista huolehtiva Navbar-komponentti tukemaan React Routerin tarjoamaan navigointia, aiemmin käytetyn Angular-ui-routerin sijasta. Kun uudistettu navigaatiokomponentti oli valmis, alettiin lisäämään näkymiä reittien taakse. Tämä tehtiin lisäämällä näkymän tarjoava komponentti Routerin Route-komponentin element-jäsenmuuttujaan.

Komponentin lisäämisen jälkeen, alkaa ajoympäristöstä tulemaan virheviestejä. Useimmat virheet liittyivät komponenttien käyttämiin AngularJS-riippuvuuksiin, kuten translate- tai state-objektin puuttumiseen. Nämä vanhat riippuvuudet korvattiin uusilla React-i18next- ja React Router -kirjastojen tarjoamilla vastaavilla

toiminnallisuuksilla. Näkymiä lisätessä ilmeni myöskin, että joissain komponenteissa oli käytössä vielä vanhoja CoffeeScriptillä kirjoitettuja utility-tiedostoja, nämä tiedostot käännettiin JavaScriptiksi ja niihin otettiin tyypitys mukaan. Myös aiemmin Angular-sovelluksesta puuttunut React Strict Mode lisäsi virheilmoitusten määrää.

```
interface Props {
  $state: any
  $translate: Translate
}

const LakuReportsListing = (props: Props) => {
  const { $state } = props
  const translate = props.$translate
const LakuReportsListing = () => {
  const navigate = useNavigate()
  const { t } = useTranslation()
```

KUVA 10. Vanhat AngularJS-riippuvuudet korvattu uusien kirjastojen tarjoamilla vastaavilla toiminnoilla.

Kun ylätason näkymät saatiin reititettyä, alettiin reitittämään näkymät, joihin normaalisti navigoidaan ylätason näkymien kautta. Sisäkkäiset näkymät saadaan reititettyä lisäämällä Route-komponentin sisälle toinen Route-komponentti, jolloin ulomman komponentin URL toimii kontekstina sisäkkäiseen komponenttiin määritetylle URL:lle.

```
<Route path='/incident'>
  <Route index element={<IncidentListing />} />
  <Route path='laku/:id' element={<LakuIncidentDetails />} />
  <Route path='veku/:id' element={<VekuIncidentDetails />} />
  <Route path='apms/:id' element={<ApmsIncidentDetails />} />
  <Route path='rfid/:id' element={<RfidIncidentDetails />} />
  <Route path='tbogi/:id' element={<TbogiIncidentDetails />} />
  <Route path='proku/:id' element={<ProkuIncidentDetails />} />
</Route>
```

KUVA 11. Sisäkkäisten näkymien reititys.

4.3.4 Rajapintoihin yhdistäminen

Näkymien reitittämisen jälkeen tarvitsi näkymiin saada dataa sisään. Koska sovelluksen rajapinnat vaativat käyttäjän autentikoinnin ja autorisaation, tarvitsi ensin uudelleenkirjoittaa sovelluksen kirjautumissivu ja -logiikka. Kirjautuminen tehtiin aluksi hyvin yksinkertaisesti, tavoitteena vain saada käyttäjälle sessio auki palvelimella, jotta rajapintakutsut onnistuisivat. Kirjautumisen yhteydessä täytyi hoitaa myös CSRF-tokenin hakeminen palvelimelta, ja sen lisääminen palvelinkutsuihin.

Jotta rajapintakutsut toimisivat lokaalissa kehitysympäristössä, täytyy selaimesta lähtevät kutsut myös ohjata oikeaan osoitteeseen ja porttiin. Tämä saatiin tehtyä lisäämällä package.json-tiedostoon proxy-asetus. Tämä asetus uudelleenohjaa selaimesta tulevat pyynnöt siinä määriteltyyn osoitteeseen ja porttiin. Tässä tapauksessa sille annettiin arvoksi ”http://localhost:8000”, eli osoite ja portti, jossa sovelluksen palvelinpuolta ajetaan lokaalisti.

4.3.5 Käyttöliittymätyylien korjaukset

Sovelluskehysmigraation yhteydessä päätettiin samalla luopua vanhasta Stylus CSS-esiprosessorista, sekä nostaa Bootstrap-kirjaston versio 3:sta 5:een. Nämä muutokset aiheuttivat käyttöliittymän tyylien hajoamisen useammassa paikkaa.

Nyt kun käyttöliittymään saatiin dataa, oli sopiva aika alkaa tekemään korjauksia käyttöliittymän tyyleihin. Ensimmäiseksi korjattiin elementit, jotka ovat läsnä jokaisessa näkymässä, kuten navigointi- tai alapalkki. Näiden jälkeen näkymät käytiin läpi yksi kerrallaan.

Suurin osa ongelmista johtui puuttuvista CSS-luokista, joita oli kadonnut Styluksen poistamisen yhteydessä. Migraation aikataulun vuoksi nämä tyyliluokat yksinkertaisesti käännettiin Styluksesta SCSS:ään verkosta löytyvän Stylus-SCSS-käännöstyökalun avulla.

Bootstrap-version nosto muutti sitä miltä jotkin tyyleistä näyttävät. Tyylejä korjattiin joko mukauttamalla olemassa olevia tyyliluokkia sopimaan uusien Bootstrap-luokkien kanssa, tai ylikirjoittamalla Bootstrapin-luokat omilla tyyliluokilla.

Jotta komponentteihin importoitavat kuvatiedostot saatiin toimimaan TypeScriptin kanssa, täytyi ne ensin tyypittää. Kuvatiedostojen tyypitys tehtiin luomalla images.d.ts-tiedosto, jossa määritetään mitä tyyppiä importoitavat png- ja svg-tiedostot ovat.

```
/* eslint-disable @typescript-eslint/no-explicit-any */
declare module '*.png' {
  const value: any
  export default value
}

declare module '*.svg' {
  const content: any
  export default content
}
```

KUVA 12. Png- ja svg-päätteisten tiedostojen sisältö määritetty any-tyypiksi.

4.3.6 Suojatut reitit

Sovelluksen kirjautumista ja reititystä refaktoroitiin ja paranneltiin aina migraation loppuun saakka. Tässä kappaleessa käydään läpi ensimmäisiä vedoksia, jolla saavutettiin pitkälti halutut tulokset, ja joiden pohjalta mekanismia alettiin jatkokehittämään.

Koska sovellusta käyttävät erilaiset käyttäjäryhmät, on eri ryhmille myöskin eri käyttäjärooleja, jotka rajoittavat käyttäjien oikeuksia. Toistaiseksi mitään näkymiä ei oltu vielä asetettu autorisaation, tai edes autentikaation taakse. Vaikka sovelluksen palvelinpuolella pyörivä Spring security pitääkin jo huolen siitä, että autorisoimaton käyttäjä ei saa dataa, on selainpuolella silti hyvä käytäntö estää autorisoimattomattomia käyttäjiä pääsemästä sivuille, joihin heillä ei ole pääsyoikeutta.

Strategiana oli, että kaikki näkymät laitetaan ensiksi ainakin autentikoinnin taakse. Koska käyttäjäroolien oikeudet kasvavat hierarkisesti, riittää matalimman käyttöoikeuden vaativiin näkymiin ja toimintoihin pelkkä autentikoituminen.

Aluksi autentikoinnin tarkistamiseksi luotiin yksinkertainen PrivateOutlet-komponentti, joka saa argumenttina tiedon siitä, onko käyttäjä autentikoitunut. Autentikoimaton käyttäjä ohjataan sovelluksen kirjautumissivulle. Autentikoituneelle käyttäjälle puolestaan ladataan haluttu näkymä React Routerin Outlet-komponenttia hyödyntäen.

```
const PrivateOutlet = ({ isAuthenticated }: Props) => {
  const location = useLocation()
  const navigateState = { redirectURL: location.pathname }
  return isAuthenticated
    ? <Outlet />
    : <Navigate to='login' state={navigateState} />
}
```

KUVA 13. Yksinkertainen komponentti, joka ohjaa kirjautumattoman käyttäjän kirjautumissivulle.

Outlet-komponentti renderöi sisempien reittien komponentin siihen kohtaan DOM:ia, johon Outlet on siinä määritelty. Kuvista 13 ja 14 nähdään, että PrivateOutlet renderöi ValtsuPage-komponentin. ValtsuPage tarjoaa kaikki yleiset käyttöliittymä elementit, kuten navigaatiopalkin, ja pitää sisällään toisen Outlet-komponentin, jonka sisälle puolestaan renderöidään eri näkymien sisältö.

```
<Route path='/login' element={<Login fetchUser={fetchUser} isAuthenticated={isAuthenticated} />} />
<Route path='*' element={<PageNotFound />} />
<Route element={<PrivateOutlet isAuthenticated={isAuthenticated} />}>
  <Route element={<ValtsuPage user={user} appInfo={appInfo} />}>
    <Route path='/' element={<Navigate to='/dashboard' />} />
    <Route path={'/dashboard'} element={<Dashboard />} />

    <Route path='/incident' >
      <Route path='laku/:id' element={<LakuIncidentDetails />} />
      <Route path='veku/:id' element={<VekuIncidentDetails />} />
      <Route path='apms/:id' element={<ApmsIncidentDetails />} />
      <Route path='rfid/:id' element={<RfidIncidentDetails />} />
      <Route path='tbogi/:id' element={<TbogiIncidentDetails />} />
      <Route path='proku/:id' element={<ProkuIncidentDetails />} />
      <Route path='' element={<IncidentListing />} />
    </Route>
```

KUVA 14. React Routerin reittihierarkia.

Kun käyttäjän autentikointi oli huomioitu, tarvitsi vielä varmistaa autorisointi niihin näkymiin, jotka sen vaativat. Ylemmän käyttöoikeuden vaativille näkymille tehtiin uusi RestrictedRoute-komponentti, joka saa argumentteinaan vaadittavan roolin, sekä elementin, joka halutaan mahdollisesti renderöidä. Komponentti tarkastaa mikä käyttäjän rooli on, vertaa sitä vaadittuun rooliin, ja joko ilmoittaa, että sivua ei löydy, tai renderöi näkymän riippuen siitä, riittääkö käyttäjän rooli kyseisen näkymän avaamiseen.

```
const RestrictedRoute = (props: Props) => {
  const { element, roles } = props
  const { user } = useAuth()

  return (roles.includes('admin') && AuthService.isAdmin(user))
    || (roles.includes('valvomo') && AuthService.isValvomo(user))
    || (roles.includes('operator') && AuthService.isOperator(user))
      ? element
      : <PageNotFound />
}
```

KUVA 15. RestrictedRoute-komponentti.

4.3.7 Sovelluksen tilan säilyttäminen ja muita haasteita

Reitityksen, käännöksen ja tyylikorjausten lisäksi sovellusrungon vaihto vaati lisäksi muitakin toimenpiteitä, joista ei aiemmin vielä ole mainittu. Näitä toimenpiteitä olivat mm. sovelluksen tilan säilyttäminen sivun uudelleenlatauksissa, URL-hakuparametrien käyttöönotto ja Redux-kirjaston uudelleen konfigurointi. Tässä luvussa perehdytään erityisesti siihen, miten sovelluksen tilan säilyttäminen hoidettiin.

Sovelluksen tilan säilyttämiseen otettiin avuksi selaimen Session Storage- ja Local Storage -toiminnot, jotka tallentavat avain-arvo-pareina dataa selaimen muistiin, jolloin data selviää sivun uudelleenlatauksista. Joitain oleellisimpia tietoja säilyttää selaimen muistissa ovat mm. käyttäjän tunnistamiseen liittyvät tiedot, roolit, CSRF-token, URL-parametrit, ja muut käyttökokemusta kohentavat tiedot. Tunnistautumiseen liittyvät tiedot ja roolit estävät sen, että sovellus palauttaisi käyttäjän aina kirjautumissivulle uudelleenlatauksen jälkeen. CSRF-token vaaditaan

mukaan jokaiseen palvelinkutsuun, joten sen säilyttämisellä vältytään sen uudelleen hakemiselta. URL-parametreja ei säilytetä pelkästään uudelleenlatausten varalta, vaan myös siksi, että ne eivät nollaudu näkymää vaihtaessa. Osa sovelluksen käyttöön vaadittavista tiedoista, kuten sovelluksen koontiversio, käyttäjätiedot ja debug feature -lista, haetaan automaattisesti palvelimelta jokaisen sivunlatauksen yhteydessä.

Lopuista tehtävistä asioista, ja muista sovellusrungon vaihdon aikana löytyneistä bugeista pidettiin yllä Confluence-sivua, jossa asiat olivat listattuna kolmeen kategoriaan niiden tärkeyden perusteella. Sivu toimi sekä muistilistana, että auttoi työn jakamisessa.

4.3.8 Migraation viimeistely

Lopuksi kun kaikki löydetyt bugit ja hajonneet tyylit ym. oli saatu korjattua, tarvitsi vielä korjata sovelluksen rakentaminen. Sovellus rakennetaan Gradle-työkalulla, ja paketoidaan WAR-paketiksi, jossa Spring Boot tarjoaa sovelluksen selainkoodin.

React-sovellus rakennetaan yksinkertaisesti ensin asentamalla package.json-tiedostossa määritellyt riippuvuudet, ja sitten rakentamalla projekti käyttämällä CRA:n tarjoamaa build-skriptiä. Skripti mm. muuttaa modernin ECMAScript 2015+ mukaiset JavaScript-koodit perinteiseksi JavaScriptiksi käyttäen Babelia, sekä niputtaa riippuvuudet ja minifioi, eli typistää JavaScript-koodit yhdeksi tiedostoksi. Lisäksi myös CSS- ja HTML-tiedostot minifioidaan. Valmiit koodit kootaan build-hakemistoon.

Jotta sovelluksen rakentaminen onnistuisi myös Jenkins-palvelimella, käytetään sovelluksen rakentamiseen Gradlea. Gradle automatisoi vaadittavien työkalujen ja riippuvuuksien asentamisen, sekä rakennus skriptin ajamisen. Edellä mainitut rakennus ohjeet annetaan Gradlelle build.gradle-tiedostossa.

```
plugins {
  id "com.github.node-gradle.node" version "3.1.1"
}

node {
  version = '16.13.1'
  yarnVersion = '1.22.5'
  download = true
}

yarn_build.dependsOn(yarn_install)

task wrapper(type: Wrapper) {
  gradleVersion = '6.4'
}
```

KUVA 16. Selainpuolen rakennus Gradlella.

Palvelinpuolelta löytyy toinen build.gradle-tiedosto. Tässä tiedostossa Gradle ohjeistetaan ensin ajamaan selainpuolelta löytyvä yarn_build-komento, joka näkyy kuvassa 16. Tämän jälkeen build-hakemistoon kasattu selainpuolen koodi viedään palvelinpuolelta löytyvään static-hakemistoon, josta Spring Boot osaa tarjota sen käyttäen Tomcat-webpalvelinta. Palvelinpuolen koodit, sekä em. static-hakemisto paketoidaan lopuksi yhdeksi WAR-paketiksi.

Viimeiseksi, kun migraatio oli valmis, otettiin vielä projektiin käyttöön ESLint- ja Prettier-työkalut. ESLint on koodin analysointityökalu, joka pitää huolen siitä, että koodissa noudatetaan sääntöjä, jotka sille on asetettu eslintrc.js-tiedostossa. Prettier puolestaan on koodin formatointityökalu, joka formatoi koodin automaattisesti prettierrc.json-tiedostossa määriteltyjen sääntöjen mukaisesti, esimerkiksi aina tallennuksen yhteydessä.

5 POHDINTA

Tässä opinnäytetyössä käytiin läpi, miten selainpuolen sovelluskehysmigraatio toteutettiin VALTSU-nimiseen keskisuureen dynaamiseen verkkosovellukseen. Lisäksi käytiin läpi migraation taustoja, syitä ja tavoitteita.

Sovelluskehysmigraatio on pitkä ja työläs prosessi. Kuten ohjelmistokehityksessä yleensäkin, työmäärän arviointi on hankalaa, ja useimmiten tuntuukin, että alkuperäiset arviot osoittautuvat aina paljon optimistisemmiksi kuin mitä todellisuus lopulta on. Vaikka tarkkaa dataa ei ole saatavilla, niin karkeasti laskettuna voisi sanoa, että myös tässä migraatiossa alkuperäinen pessimistinen työmääräarvio ylittyi huomattavasti.

Menikö migraatiossa sitten jotakin merkittävästi pieleen? Mielestäni ei mennyt, työmäärä arvio oli loppujen lopuksi vain nimensä mukaisesti arvio, ja se oli arvioitu alakanttiin. Arvio oli tehty todella karkeasti ja siinä ei oltu osattu ottaa huomioon sitä, miten työläitä osa komponenteista oli oikeasti uudelleenkirjoittaa (Sorje 2022). Lisäksi se, että itse tulin projektiin mukaan pienellä työelämäkokemuksella, ja toteutin valtaosan migraatiosta itse, niin luonnollisesti kokeneempi tekijä olisi pystynyt hoitamaan saman työmäärän lyhyemmässä ajassa.

Mitä migraatiosta sitten opittiin? Aivan migraation alkuvaiheessa, jo kauan ennen kuin itse aloitin, oli useita komponentteja tehty hyvin pitkälti vain kopioimalla AngularJS-komponenteissa käytettyä logiikkaa uuteen React-komponenttiin. Tästä tavasta oli onneksi luovuttu hyvin nopeasti, kun oli huomattu miten monimutkaisia ja hankalasti ylläpidettäviä komponenteista tuli.

Lopputuloksena migraatio saatiin valmiiksi, ja sovellus on nyt teknologioidensa puolesta täysin ajan tasalla. Mahdolliset AngularJS:stä tai NodeJS 8:sta tulevaisuudessa löytyvät haavoittuvuudet eivät tule uhkaamaan sovellusta. Lisäksi selainpuolella sovelluskehitys on nyt merkittävästi nopeampaa ja motivoivampaa, kun käytössä ovat modernit teknologiat ja kehitystyökalut.

LÄHTEET

AngularJS 4U. 2017. Branding Guidelines for Angular and AngularJS. Luettu 7.5.2022.
https://www.angularjs4u.com/angularjs2/branding-guidelines-angular-angularjs/

Baer E. 2018. What React Is and Why It Matters. O'Reilly Media, Inc. Luettu 7.5.2022.
https://www.oreilly.com/library/view/what-react-is/9781491996744/ch01.html

coatue-oss. 2019. react2angular. Luettu 7.5.2022.
https://github.com/coatue-oss/react2angular/blob/master/README.md

Darwin P. 2018. Stable AngularJS and Long Term Support. Luettu 7.5.2022.
https://blog.angular.io/stable-angularjs-and-long-term-support-7e077635ee9c

Edpresso Team. n.d. What are React functional components. Luettu 7.5.2022.
https://www.educative.io/edpresso/what-are-react-functional-components

freeCodeCamp. 2016. Beginner's Guide to React Router. Luettu 7.5.2022.
https://www.freecodecamp.org/news/beginner-s-guide-to-react-router-53094349669/

Google. n.d. Creating Custom Directives. Luettu 7.5.2022.
https://docs.angularjs.org/guide/directive

Google. n.d. What is AngularJS. Luettu 7.5.2022.
https://docs.angularjs.org/guide/introduction

Meta. 2022. JSX. Luettu 7.5.2022.
https://facebook.github.io/jsx/

Meta. n.d. Create a New React App. Luettu 7.5.2022.
https://reactjs.org/docs/create-a-new-react-app.html

Meta. n.d. React.Component. Luettu 7.5.2022.
https://reactjs.org/docs/react-component.html

Meta. n.d. Virtual DOM and Internals. Luettu 7.5.2022.
https://reactjs.org/docs/faq-internals.html

Microsoft. n.d. tsc, the TypeScript compiler. Luettu 7.5.2022.
https://www.typescriptlang.org/docs/handbook/2/basic-types.html#tsc-the-typescript-compiler

Microsoft. n.d. What is tsconfig.json. Luettu 7.5.2022.
https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

Microsoft. n.d. What Problems Can TypeScript Solve. Luettu 7.5.2022.
https://www.typescriptlang.org/why-create-typescript

Red Hat. 2019. End of Support for Node.js 8.x on December 31, 2019. Luettu 7.5.2022.
https://access.redhat.com/announcements/4636741

Sorje O. Senior Software Designer. 2022. Haastattelu 14.2. Haastattelija Penttinen N. Puhelu.

Thompson M. 2020. AngularJS LTS Extended in response to COVID-19. Luettu 7.5.2022.
https://blog.angular.io/angularjs-lts-extended-in-response-to-covid-19-321b037212f5

Sass team. n.d. Sass Basics. Luettu 7.5.2022.
https://sass-lang.com/guide

Stack Overflow. 2021. 2021 Developer Survey. Luettu 16.5.2022.
https://insights.stackoverflow.com/survey/2021

Väylävirasto. 2021. Rautateiden verkkoselostus. Luettu 7.5.2022.
https://julkaisut.vayla.fi/pdf12/vj_2019-46_vs2021_web.pdf

LIITTEET

Liite 1. VALTSU AngularJS-React-migraatiosuunnitelma

Tausta

- AngularJS tuki loppuu 30.6.2021. Tämä on tietoturvaongelma.
- NodeJS 8 tuki loppui 31.12.2019. AngularJS päivitys mahdollistaa myös NodeJS:n päivityksen.
- Helpompi ja tehokkaampi ylläpito uudemmilla työkaluilla

Scope

Must have

- Kaikki näkymät kirjoitetaan Reactilla, AngularJS, CoffeeScript ja Jade pois
- Lopuksi react2angular pois
- Stylus pois, SCSS tilalle
- Node päivitys

Nice to have (ei mukana työmääräarviossa)

- Gulp pois, webpack tilalle
- Restangular pois, Axios tilalle

Sekalaisia ajatuksia toteutuksesta

- Jade templatet kannattanee ensin kääntää automaattisesti HTML:ksi, jotta React tekeminen helpottuu
- Servicet konvertoidaan viimeisenä
- Näkymä kerrallaan kuntoon. Viedään tuotantoon sitä mukaa, kun näkymiä on valmistunut.
- Kaikki uudet toiminnallisuudet rakennetaan jo nyt Reactilla
- Syytä käydä kriittisesti läpi toiminnot, jos niissä olisi jotain turhia, jotka olisi syytä poistaa eikä migroida.

Työmäärä

- 137 Jade templatea (näkymää/komponenttia)
- 218 CoffeeScript tiedostoa

Optimistinen
137 htp ~puolen vuoden projekti (1 htp per template)

Pessimistinen
411 htp ~ 1,5 vuoden projekti(3 htp per template)