



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MERVI LEPPÄKORPI
BUDJETOINTISOVELLUKSEN ASIAKASPÄÄ PROTOTYYPISTÄ
TUOTTEEKSI

Diplomityö

Tarkastaja: professori Kari Systä
Tarkastaja ja aihe hyväksytty
3. tammikuuta 2018

TIIVISTELMÄ

MERVI LEPPÄKORPI: Budjetointisovelluksen asiakaspää prototyypistä tuotteenksi

Tampereen teknillinen yliopisto

Diplomityö, 42 sivua

Maaliskuu 2018

Tietotekniikan diplomi-insinöörin tutkinto-ohjelma

Pääaine: Ohjelmistotuotanto

Tarkastaja: professori Kari Systä

Avainsanat: React.js, Angular 2, web-ohjelmointi, asiakaspää

Tämän diplomityön aiheena on Wapice Oy:n asiakkaalleen tekemän budjetointisovelluksen asiakaspään päivitys. Työssä vertaillaan kahta eri teknologiaa, React ja Angular 2, asiakaspään toteutusteknologiaksi toteuttamalla kaksi prototyyppiä. Lisäksi käsitellään syyt uuden asiakaspään toteutukseen ja asiakaspään toteutus prototyypistä eteenpäin.

Alkuperäinen budjetointisovellus toteutettiin vuosina 2009-2012. Sovellus tarjosi laskentataulukko-ohjelmistoa paremman työkalun budjettien hallinnoimiseksi. Sovelluksen teknologiana käytettiin Adobe Flex -ohjelmistokehystä, joka käyttää Adobe Flash Playeriä. Viime vuosina Adobe Flash Playeristä on löytynyt tietoturva-aukkoja, ja selaimet ovat hankaloittaneet tai estäneet sen käyttöä. Tästä johtuen asiakaspää haluttiin uudistaa modernilla teknologialla. Samalla uudistettaisiin asiakaspään ulkoasu.

Toteutus aloitettiin tekemällä kaksi prototyyppiä kahdella eri JavaScript-ohjelmointikehyksellä: Reactilla ja Angular 2:lla. Näistä Angular 2 oli aivan uusi ohjelmointikehys, josta kaivattiin kokemusta, ja React hyväksi ja toimivaksi todettu ohjelmistokehys. Prototyyppien toteutuksen jälkeen todettiin, että Angular 2 oli vielä liian keskeneräinen, ja päädyttiin valitsemaan React tuotteen toteutuksen teknologiaksi. Prototyypin koodia hyödynnettiin tuotteen toteutuksessa koodikatselmoinnin jälkeen. Tuotteessa käytettiin paljon erilaisia kirjastoja, jotka tarjosivat erilaisia perustoiminnallisuuksia kuten kansainvälistämisen ja ilmoitusten näyttämisen. Suurien taulukoiden esittämiseen etsittiin myös kirjastoa, mutta sopivaa ei löytynyt. Tuotetta testattiin manuaalisesti, ja uuteen asiakaspäähän lisättiin joitain uusia ominaisuuksia.

Projektin lopputuloksena valmistui uusi, vaatimukset täyttävä ja toimiva asiakaspää. Diplomityön kirjoitushetkellä asiakaspää odottaa käyttöönottoa tuotannossa.

ABSTRACT

MERVI LEPPÄKORPI: Budgeting application frontend from prototype to product
Tampere University of Technology
Master of Science Thesis, 42 pages
March 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Kari Systä

Keywords: React.js, Angular 2, web programming, frontend

The subject of this master's thesis is a renewal of a frontend in a budgeting application that Wapice Oy has done for a customer. A comparison between two technologies, React and Angular 2, was done by implementing two prototypes to decide which technology is better for this application. Thesis will also address the reasons for creating a new front end and describe the implementation of the new front end after the prototypes.

The original budgeting application was created between years 2009 and 2012 to replace the use of spreadsheet application. The frontend technology was Adobe Flex that uses Adobe Flash Player. In recent years, some security issues have been found in Adobe Flash Player and browsers have started to block or complicate the use of Adobe Flash Player. Because of this, the customer wanted to renew the frontend with a modern technology. At the same time, the layout of the application was to be refreshed.

Implementation was started by doing two prototypes with two different JavaScript frameworks: React and Angular 2. Angular 2 was a brand-new framework of which Wapice wanted some experience. React was older and already in use in Wapice projects. It was decided that Angular 2 was too incomplete to be used in products after implementing the prototypes and React was chosen as the technology. A code review was held to check the code of the prototype so that it could be used in the product. Many different libraries that offered basic functionalities like internationalization and notifications were used in the implementation of the product. The product was tested manually, and some new features were added to it.

As a result of the project a new front end was finished that filled the requirements. The frontend is waiting to be deployed to production during the writing of this thesis.

ALKUSANAT

Tämä opinnäyte on kirjoitettu Wapice Oy:lle, jota haluan myös kiittää diplomityöaiheen tarjoamisesta. Työn tarkoituksena oli suunnitella ja toteuttaa budjetointiohjelmalle uusi asiakaspää uudella teknologialla vanhan tilalle. Haluan kiittää työn tarkastajaa professori Kari Systää ja työn ohjaajaa Tommi Aittokalliota ohjauksesta ja neuvoista työn aikana. Haluan kiittää myös Pasi Leppästä avusta ohjelman ymmärtämisessä sekä Mika Kulmaa palvelinpään päivittämisestä ja testauksessa auttamisesta.

Tampereella, 19.3.2018

Mervi Leppäkorpi

SISÄLLYSLUETTELO

1.	JOHDANTO	1
2.	LÄHTÖKOHDAT	3
	2.1 Vanha järjestelmä.....	3
	2.2 Vaatimukset.....	5
3.	PROTOTYYPIT	8
	3.1 Angular 2 -prototyyppi.....	8
	3.1.1 Angular 2.....	8
	3.1.2 Prototyypin toteutus	11
	3.2 React-prototyyppi.....	14
	3.2.1 React.js.....	14
	3.2.2 Redux.js.....	18
	3.2.3 Prototyypin toteutus	21
	3.3 Teknologian valinta.....	25
4.	TOTEUTUS	27
	4.1 Prototyyppiin tehdyt muutokset	27
	4.2 Lopullinen tuote	30
	4.3 Tuotteen testaus.....	34
	4.4 Tuotteen jatkokehitys	36
5.	TYÖN ARVIOINTI.....	38
	5.1 Vaatimusten täytyminen.....	38
	5.2 Tehdyt teknologiapäätökset.....	39
6.	YHTEENVETO	41
	LÄHTEET.....	43

LYHENTEET JA MERKINNÄT

Ajax	Tekniikka, jolla tehdään asynkronisia HTTP-pyyntöjä yleensä asiakaspäästä palvelinpäähän
DOM	Document Object Model eli selaimen käyttämä puumainen malli HTML-sivun rakenteen jäsentämiseen. Puun olioita voi hakea, tutkia ja manipuloida esimerkiksi JavaScriptin avulla.
ES6	ECMAScript 6, vuonna 2015 julkaistu JavaScript standardi, joka tuo uusia ominaisuuksia JavaScriptiin esim. oletusparametrit.
HTML	Hypertext Modeling Language on standardisoitu kieli kuvata hyperlinkkejä sisältävää tekstiä. Sen avulla voidaan myös merkitä tekstin rakennetta esim. kappalejakoja ja otsikoita.
MVC-malli	Ohjelmistoarkkitehtuuri, jonka nimi tulee sanoista Model View Controller (malli, näkymä, käsittelijä). Nimensä mukaan arkkitehtuuri jakaa ohjelmistojn kolmeen osaan.
MVVM-malli	Ohjelmistoarkkitehtuuri, joka on muunnos tavallisesta MVC-arkkitehtuurista. MVVM tulee sanoista Model View View Model (malli, näkymä, näkymä, malli) eli ohjelmisto on jaettu neljään osaan.
REST	Representational State Transfer eli HTTP-protokollaan perustuva arkkitehtuurimalli web-rajapintojen toteutukseen.
XML	Extensible Markup Language on standardi, jolla tiedon merkitys kuvataan tiedon sekaan. XML-kieliä käytetään tiedonvälitykseen sekä formaattina tiedostojen tallentamiseen.

1. JOHDANTO

Ohjelmistokehitys on edistynyt hyvin paljon viime vuosien aikana, ja samaa tahtia on noussut ohjelmistojen kysyntä ja tarjonta. Monille asioille, joita ennen on tehty paperin ja kynän, laskentataulukko-ohjelmiston tai muun yleispätevän ohjelman avulla, on nykyään saatavilla juuri kyseiseen tarkoitukseen luotu sovellus. Paljon sovelluksia luodaan lisäksi vastaamaan asiakkaiden yksilöllisiä tarpeita yleispätevän ratkaisun tarjoamisen sijaan.

Nopean kehityksen seurauksena teknologiat kuitenkin vanhenevat, ja uusia teknologioita kehitetään ja julkaistaan koko ajan. Etenkin web-sovellusten tekniikat kehittyvät nopeasti, ja valittavana on satoja erilaisia ohjelmistokehyksiä ja ohjelmointikieliä, joilla sovelluksen voi toteuttaa. Sopivan teknologian löytäminen kullekin sovellukselle on hankalaa, kun valinnanvaraa on paljon.

Lisäksi on vaikea ennustaa, mitkä teknologiat vanhenevat seuraavan viiden tai kymmenen vuoden aikana, ja mitkä teknologiat pysyvät käytössä ja suosittuina. Uusimmat, vain vähän aikaa sitten julkaistut teknologiat eivät välttämättä ole vielä valmiita, tai niitä ei välttämättä ole ehditty testaamaan toimiviksi ja sopiviksi suuriin ohjelmistoihin. Vanhemmista teknologioista löytyy enemmän kokemusta ja tietoa, mutta toisaalta teknologia voi olla jo vanhentunut. Uusien ja vanhojen teknologioiden välillä tasapainoilu on haastavaa.

Nykyään web-ohjelmia käytetään paljon myös mobiililaitteilla, jolloin ulkoasun responsiivisuus eli skaalautuvuus erilaisille näytöille on tärkeässä asemassa. Mobiililaitteiden myötä myös ulkoasujen tyyli on tarvinnut muuttua yksinkertaisemmaksi ja selkeämmäksi, ja tällaista ulkoasua käyttäjät ovat tottuneet myös käyttämään.

Vanhentuneen web-sovelluksen päivittäminen voidaan tehdä monella eri tavalla. Web-sovellus on tyypillisesti jaettavissa kahteen eri osaan: asiakaspäähän ja palvelinpäähän. Näistä jompaakumpaa tai molempia voidaan pystyä hyödyntämään tai käyttämään pienillä muutoksilla ohjelman uudistamisen jälkeen. Vanha ohjelma voidaan myös toteuttaa alusta asti uudelleen, mikä mahdollistaa parannusten tekemisen, mutta myös vaatii enemmän aikaa ja työtä.

Asiakaspään suosituimpina teknologioina ovat tällä hetkellä erilaiset JavaScript-ohjelmistokehykset, joita on tarjolla useita. Suosituimpia ohjelmistokehyksiä ovat Angular.js ja React, jotka ovat käytössä monilla isoillakin sivuilla, mutta myös uusia ohjelmistokehyksiä on kehitteillä ja tarjolla. Ohjelmistokehykset ovat kaikki luonnollisesti erilaisia, ja se

ohjelmistokehys, joka sopii yhteen sovellukseen ei välttämättä sovi yhtä hyvin seuraavaan.

Tämän työn lähtökohtana on laskentataulukko-ohjelmiston käyttämisen korvannut web-ohjelma, jonka asiakaspään teknologia on vanhentunut ja jonka ulkoasu kaipaa päivitystä. Luvussa kaksi esitellään ohjelman rakennetta ja syitä ohjelman päivittämispäätökselle, sekä eritellään uuden asiakaspään vaatimukset. Ohjelman päivitysprosessi käynnistettiin kahden prototyypin avulla, joita käsitellään luvussa kolme. Teknologioiksi prototyyppeihin valittiin kaksi JavaScript-ohjelmistokehystä nimeltään Angular 2 ja React. Prototyypin avulla pyrittiin arvioimaan, kumpi valituista teknologioista sopisi parhaiten asiakaspään toteuttamiseen.

Luvussa neljä käsitellään, miten ohjelman toteutusta jatkettiin, miten ohjelmaa testattiin ja miten ohjelmaa on uudistettu teknologian ja ulkoasun lisäksi. Viidennessä luvussa arvioidaan, kuinka hyvin projektin vaatimukset tavoitettiin ja miten teknologiapäätöksissä onnistuttiin. Lopuksi on yhteenveto projektista.

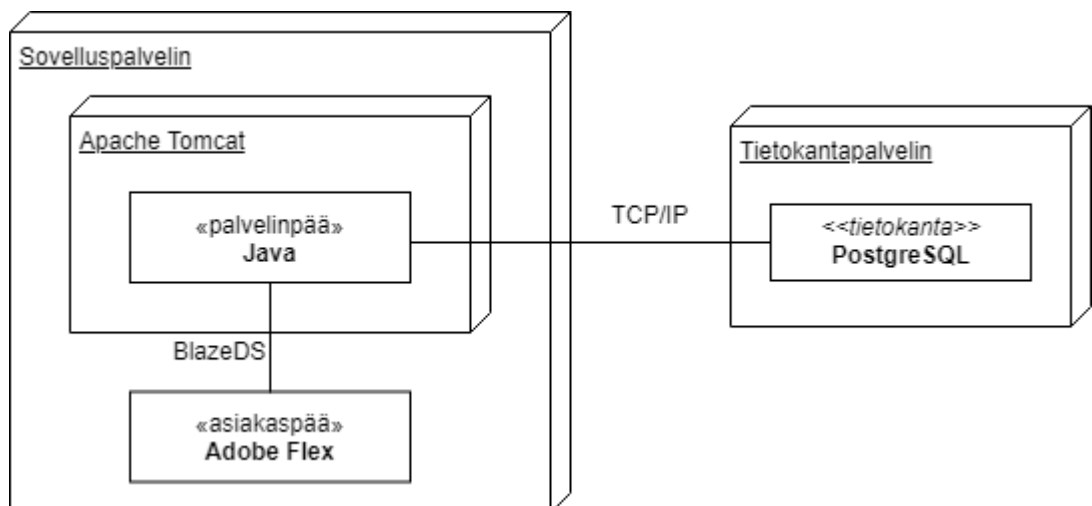
2. LÄHTÖKOHDAT

Tässä luvussa eritellään syyt, joiden takia asiakas on tilannut toimivasta järjestelmästä uuden asiakaspään ja mitä vaatimuksia asiakkaalla on uudelle versiolle järjestelmästä. Tavoitteena uuden version kehittämisessä oli toteuttaa modernimmalla teknologialla ohjelman asiakaspäästä uusi versio, joka sisältäisi kaikki samat toiminnallisuudet kuin vanha versio. Samalla uudistettaisiin asiakaspään ulkoasu.

2.1 Vanha järjestelmä

Wapice Ltd. teki asiakkaalleen vuosina 2009-2012 sovelluksen, jolla asiakas pystyy hallinnoimaan budjetteja ja kuluja sekä luomaan raportteja. Aiemmin asiakas oli käyttänyt tähän tarkoitukseen laskentataulukko-ohjelmistoa, jossa tietojen hallinnointi oli haasteellista tiedon suuren määrän sekä mutkikkaiden operaatioiden takia. Lisäksi laskentataulukko-ohjelmistoa käyttäessä laskukaavojen muutos tarvitsi päivittää erikseen jokaiseen tiedostoon tehden budjeteista hankalasti ylläpidettäviä. Käytettävyys laskentataulukko-ohjelmistossa ei myöskään ole kovin hyvä, kun käsitellään kymmeniä ellei jopa satoja tiedostoja.

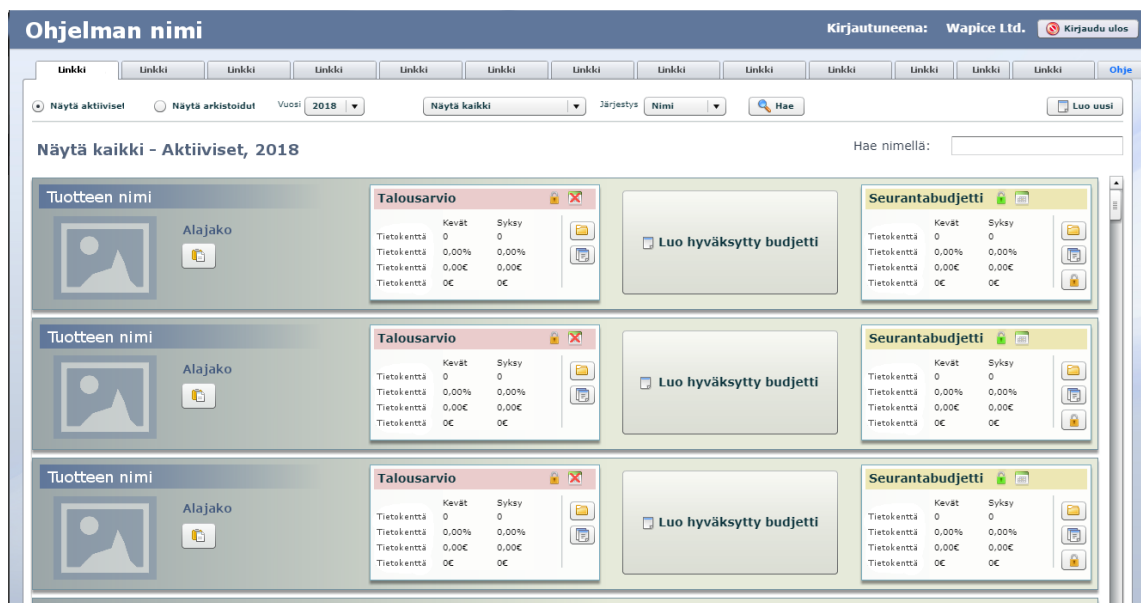
Wapice lähti toteuttamaan asiakkaalle järjestelmää web-sovelluksena kolmikerrosarkkitehtuurilla asiakaspään, palvelinpään ja tietokannan avulla. Sovelluksen palvelinpää toteutettiin Javalla ja tietokanta PostgreSQL-tietokannalla. Palvelinpään ja tietokannan väliseen tiedonsiirtoon käytettiin Hibernate-ohjelmistokehystä. Sovelluksen arkkitehtuuri on esitetty kuvassa 1.



Kuva 1. Vanhan sovelluksen arkkitehtuuri

Sovelluksen asiakaspää toteutettiin Adobe Flex –ohjelmointirajapinnalla [1], joka on Adobe Flash Playeriä käyttävä ohjelmistokehys. Ohjelmaa kehitettäessä selainten tuki muun muassa JavaScriptille ja CSS-muotoiluille ei ollut yhtä hyvä kuin nykypäivänä, ja Adobe Flexiä käyttämällä kehittäjiä ei tarvinnut huolehtia selainten rajoituksista ja eroavaisuuksista. Palvelinpään ja asiakaspään välinen tiedonsiirto toteutettiin BlazeDS-tekniologialla, joka [2] on tarkoitettu Java-palvelinpään ja Adobe Flex -asiakaspään väliseen tiedonsiirtoon.

Järjestelmän ulkoasulla haettiin samaa näköä kuin laskentataulukko-ohjelmistossa, jotta järjestelmä muistuttaisi asiakkaiden aikaisempaa tapaa tehdä asioita, ja olisi siten asiakkaalle helpompi ottaa käyttöön. Vanhan asiakaspään ulkoasu on esitetty alla olevassa kuvassa 2.



Kuva 2. Vanhan asiakaspään ulkoasu

Asiakaspää vaati päätelaitteen resoluution olevan leveydeltään vähintään 1300px. Mikäli näyttö on pienempi, ilmestyy käyttöliittymään vaakasuuntainen vieritys, joka aiheuttaa ohjelman käytettävyyden laskun, kun kaikkea sisältöä ei näe ilman vierittämistä. Pystysuuntainen vieritys käyttöliittymässä on myös erilainen kuin mihin käyttäjät ovat yleensä tottuneet, sillä yleensä vierittäminen vie sivua alaspäin vain osan sivua, kun taas ohjelman käyttöliittymässä vieritys vie kokonaisen näkymän verran alaspäin. Eli esimerkiksi kuvassa 2 olevassa tilanteessa vieritys vie kolme riviä tuotteita alas- tai ylöspäin.

Monet toiminnot vaativat käyttöliittymässä lisäksi useita klikkauksia. Esimerkiksi suoritettavat haut vaativat lähes kaikissa tilanteissa erillisen Hae-nappulan painalluksen sen sijaan, että kriteerin muutos vaikuttaisi heti sisältöön. Nykyteknologialla tilanteissa, joissa palvelinpäästä tiedon hakeminen on nopeaa, ei ylimääräiselle Hae-nappulan painallukselle ole syytä.

Teknologiavalinnan takia järjestelmää oli hankala käyttää mobiililaitteilla, kuten esimerkiksi tabletilla, koska jokaiselle laitteelle tarvitsisi asentaa erikseen Adobe Flash Player sekä budjetointiohjelman vaatima tietoturvarvarmenne. Tämä johtui siitä, että asiakkaalle ei ollut tärkeää ohjelman käyttäminen mobiililaitteella järjestelmän toteutuksen aikana. Nykyään kuitenkin on yleisempää, että web-ohjelmaa halutaan käyttää ajasta, paikasta ja välineestä riippumatta, jolloin tietoturvarvarmenne ja erillisen ohjelman asentaminen laitteelle vaikeuttavat ohjelman käyttämistä.

Vuonna 2015 [3] Adobe Flash Playerista löytyi tietoturva-aukkoja, joita kaikkia Adobe ei ole pystynyt paikkaamaan. Vakava haavoittuvuus mahdollisti muun muassa Adobe Flash Playeriä käyttävien tietokoneiden haltuunoton. Adobe [4] onnistui paikkaamaan tietokoneen haltuun ottamisen mahdollistavan tietoturva-aukon, mutta uusia [5] tietoturva-aukkoja löydetään edelleen.

Koska Adobe [6] ei ole onnistunut paikkaamaan kaikkia tietoturva-aukkoja, useat selaimet ovat estäneet tai hankaloittaneet Flash Playerin käyttöä. Nykyään käyttäjän tulee esimerkiksi google Chrome -selaimessa sallia Flash Playerin käyttö erikseen jokaisen sivun kohdalla tai muuttaa selaimen asetuksissa, että Flash Playerin käyttö sallitaan kaikilla sivuilla. Vuoden 2017 kesällä Adobe ilmoitti lopettavansa Adobe Flashin tukemisen vuonna 2020.

Kun ongelmia alkoi ilmetä Adobe Flashin käytön kanssa, Wapice aloitti keskustelemaan asiakkaan kanssa uuden HTML5-käyttöliittymän tekemisestä, jolla ratkaistaisiin teknologiaongelma sekä päivitetäisiin järjestelmän ulkoasu.

2.2 Vaatimukset

Asiakas asetti uudelle järjestelmälle toiminnalliseksi vaatimukseksi sen, että uudella järjestelmällä tulee pystyä tekemään kaikki samat toiminnot kuin vanhalla järjestelmällä. Vanhasta käyttöliittymästä tulisi siis löytää kaikki toiminnallisuudet ja toteuttaa vastaavat toiminnallisuudet uuteen käyttöliittymään. Tarkempaa, sivukohtaista ja toiminnallista vaatimuslistaa ei saatu asiakkaalta johtuen osaksi siitä, että suuri osa toteutustyöstä toteutettiin asiakkaan kesälomien aikana.

Apua toiminnallisuuksien ja suurempien, taustalla tapahtuvien syyseuraussuhteiden avaamisessa tarjosivat ohjelman ensimmäisen version tehneet kehittäjät, jotta nykyisten kehittäjien ymmärrys ohjelman toiminnoista kasvoi ja toimintojen testaaminen mahdollistui. Koska ensimmäisen version toteutuksesta oli jo useampia vuosia, vaati joidenkin asioiden selvittäminen myös vanhan koodin tutkimista.

Asiakaspään tuetuiksi selaimiksi valittiin Google Chrome sekä Safari sillä perusteella, että monet loppukäyttäjistä käyttävät Applen kannettavia, joissa on oletuksena Safari-selain, ja suurin osa muista käyttäjistä käyttää Chrome-selainta. Rajallisen testausajan takia useamman selaimen tukeminen olisi ollut vaikeaa.

Asiakkaan ei-toiminnallisina vaatimuksina uudelle järjestelmälle oli toteuttaa käyttöliittymä modernimmalla teknologialla, jotta Adobe Flash –ongelmasta päästäisiin eroon. Teknologian tulisi toivottavasti olla lisäksi riittävän moderni, jotta uusia ongelmia ei ilmeneisi ainakaan lähivuosina. Lisäksi asiakas halusi, että käyttöliittymän ulkoasu uudistettaisiin.

Wapice asetti teknologiavaihtoehtoiksi kaksi eri JavaScript-ohjelmistokehystä, joista toisella asiakaspää toteutettaisiin. JavaScript-ohjelmistokehysten käyttäminen tarkoitti sitä, että kommunikointi palvelinpään kanssa ei onnistuisi enää BlazeDS-teknologialla, koska BlazeDS on tarkoitettu vain Adobe Flex –ohjelmistokehysten kanssa käytettäväksi. Palvelinpäähän päätettiin toteuttaa nykyään monissa web-ohjelmissä käytössä oleva REST-arkkitehtuurin mukainen rajapinta, jotta uusi asiakaspää ja palvelinpää pystyisivät kommunikoidaan. Muita muutoksia palvelinpäähän ei tarvinnut tehdä, ja REST-rajapinnan funktiot pystyivät käyttämään valmiina olevia funktioita esim. erilaisten listojen tai olioiden hakemiseen.

Käyttöliittymän ulkoasulle ei asetettu sen tarkempia vaatimuksia, kuin että ulkoasun tulisi olla modernimpi ja selkeä, jotta järjestelmän myyminen muille asiakkaille helpottuisi. Alun perin asiakas olisi myös halunnut, että uutta asiakaspäätä pystyisi käyttämään myös mobiililaitteella. Käytännössä tämä olisi tarkoittanut, että käyttöliittymän tulisi olla responsiivinen ja mukautua toimivaksi vähintään tabletilla, mutta mahdollisesti myös matkapuhelimella. Tämä vaatimus olisi vaikeuttanut toteutustyötä huomattavasti, koska ohjelma sisältää erittäin suuria taulukoita, joiden sovittaminen mobiililaitteen näytölle olisi ollut lähes mahdotonta. Myöhemmin asiakas totesi, ettei sovelluksen tarvitse olla käytettävissä mobiililaitteella.

Uuden asiakaspään käytettävyyteen liittyen asiakas halusi käyttöliittymään ilmoituksen sivulta poistuttaessa, mikäli sivulla olisi tallentamattomia muutoksia. Monia asioita pystyy muokkaamaan käyttöliittymässä ilman, että muutoksen tallentuvat heti ja käyttäjän tarvitsee vasta lopuksi tallentaa muutoksensa. Käyttäjä voi siis helposti unohtaa tallentaa tekemänsä muutokset, mikäli käyttäjää ei muistuteta asiasta. Onnistuneesta ja epäonnistuneesta toiminnosta tulee myös ilmoittaa käyttäjälle jollain tavalla.

Asiakas asetti lisäksi tietoturvalle vanhasta järjestelmästä poikkeavan vaatimuksen. Vanha järjestelmä vaati, että päätelaitteelle tulee asettaa varmenne, ennen kuin järjestelmää pystyi käyttämään. Asiakas koki tämän tavan hankalaksi erityisesti uusien käyttäjien aloittaessa järjestelmän käyttämisen ja toivoi, että järjestelmän tietoturva päivitetäisiin sellaiseksi, että varmennetta ei enää tarvittaisi. Tämä vaatimus toteutettiin palvelinpäässä

luomalla uudet taulukot tietokantaan ja määrittämällä kaikkiin REST-rajapinnan kutsuihin, mikä rooli käyttäjällä tulee olla, jotta kutsun saa tehdä.

Koska asiakaspäätä lähdettiin uudistamaan alusta asti, ja on hyvinkin mahdollista, että ohjelman ottaa joskus käyttöön muukin asiakas, Wapice päätti mahdollistaa asiakaspään kansainvälistämisen. Kansainvälistämisen toteuttaminen käyttöliittymää luodessa on helpompaa, kuin kansainvälistäminen jälkikäteen, joten vaikka asiakas ei halunnutkaan lokalisoida käyttöliittymää, asetettiin Wapicen puolesta kansainvälistäminen vaatimukseksi.

3. PROTOTYYPIT

Kesällä 2016 tehtiin kaksi prototyyppiä kahdella eri JavaScript-ohjelmistokehyksellä. Ohjelmistokehyksiksi valittiin Angular 2 ja React. Angular 2 valittiin teknologiaksi, koska se oli aivan uusi ohjelmistokehys, ja siten siitä haluttiin saada kokemusta. React valittiin toiseksi teknologiaksi, koska se oli Wapicella ennestään toimivaksi ja hyväksi todettu ohjelmistokehys.

Ensimmäinen prototyyppi toteutettiin Angular 2 –ohjelmistokehyksellä alkukesästä ja toinen React-ohjelmistokehyksellä Redux.js-kirjaston kanssa loppukesästä. Molempien prototyyppien tavoitteena oli luoda uuteen ulkoasuun kaksi toiminnallisuutta: listaus tuotteista sekä kaavio, johon voi valita mitä tietosarjoja näytetään. Tuotelistauksessa tulisi lisäksi olla mahdollista rajata listaa sekä luoda ja muokata tuotteita, vaikka tieto ei tallentuisikaan tässä vaiheessa mihinkään.

Molemmat prototyypit ovat teknologiavalinnan ansiosta yhden sivun sovelluksia, eli ne vaativat vain yhden sivulatauksen koko istunnon aikana. Tämä olisi yleensä kirjautumissivu, mutta prototyypeissä ei vielä toteutettu käyttäjän kirjautumista. Ensimmäisellä sivulatauksella näytetään siten tuotelista-näyttö.

3.1 Angular 2 -prototyyppi

Ensimmäinen prototyyppi tehtiin touko- ja kesäkuussa 2016 Angular 2 –ohjelmistokehyksellä, josta oli tuolloin julkaistu julkaisuehdokas eli niin kutsuttu beeta-versio, joka on valmis julkaistavaksi, mikäli suuria ongelmia ei ilmene [7]. Syynä Angular 2 :n koikelemiseen prototyypin teknologiana oli se, että Wapicella ei ollut ohjelmistokehyksestä vielä kokemusta. Wapice halusi saada ohjelmistokehyksestä lisää tietoa sekä selvittää sen käyttömahdollisuuksia tuotantoon menevissä tuotteissa.

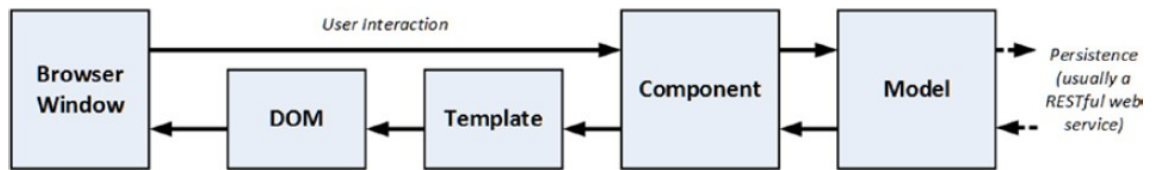
3.1.1 Angular 2

Angular 2, josta käytetään myös nimeä Angular, [8] on uusi versio AngularJS-ohjelmistokehyksestä. AngularJS [9] on erittäin suosittu ohjelmistokehys, joka kuitenkin sisältää vaikeasti ymmärrettäviä ja monien kehittäjien mielestä kummallisia ominaisuuksia, jotka ovat tehneet web-sovellusten kehityksestä vaikeampaa kuin sen tarvitsisi olla. Angular [8] on kokonaan uudelleenkirjoitettu versio AngularJS-ohjelmistokehyksestä. Angularin tarkoitus on olla helpommin opittava ja johdonmukaisempi verrattuna AngularJS-kehukseen, mutta silti toimia samalla ajatuksella kuin AngularJS.

Angular toteuttaa rakenteeltaan MVC-arkkitehtuurimallin, joka on esitetty kuvassa 3. Angularissa toteutetaan MVC-malli niin, että näkymä (view) tehdään template-tekniikalla,

jota käytetään käsitteenä näkymän tilaan. Käsittelijä (controller) on lisäksi Angularissa nimellä komponentti (component).

Angularissa mallit (model) sisältävät usein palvelinpäähän tehtävät REST-kutsut, joilla otetaan yhteys palvelinpäähän, joka sisältää varsinaisen logiikan, mutta myös malli voi sisältää ohjelman logiikkaa. Komponentit huolehtivat näkymän tilan alustamisesta ja päivittämisestä. Komponentit [10] ovat Angularissa käytännössä luokkia, jotka huolehtivat tiedon välittämisestä näkymälle ja sisältävät logiikan käyttäjän toimintoihin reagoimiseen.



Kuva 3. MVC-malli Angular 2:ssa [8]

Näkymät [10] sisältävät HTML-elementit, joilla kuvaillaan käyttöliittymä. Näkymä on tärkeä osa komponenttia, sillä ilman näkymää, DOM-puuhun ei lisätä mitään eli käyttäjälle ei näy mitään. Kun komponentteja halutaan laittaa sisäkkäin, tehdään se käyttämällä komponentin selector-parametria kulmasulkeiden sisällä kuten HTML-elementtiä näkymän koodissa.

Näkymän voi liittää komponenttiin joko antamalla polun ulkoiseen tiedostoon, jossa näkymä on kuvattu, tai määrittelemällä näkymän komponentin kanssa samassa tiedostossa. Näkymän sijoittaminen omaan tiedostoonsa on järkevää, kun näkymä on monirivinen ja pitkä. Samassa tiedostossa näkymän määrittely helpottaa kuitenkin näkymän ja komponentin koodin muokkaamista ja tiedon sidontaa. Näkymässä käytettävät tyylit voidaan kirjoittaa komponentin kodiin tai omaan tiedostoonsa samaan tapaan kuin näkymän koodi. Yksinkertainen komponentti on kuvattu ohjelmassa 1.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <h1>
      {{title}}
    </h1>
  `,
  styles: [`
    h1 { color: darkblue }
  `]
})
export class AppComponent {
  title = 'Basic application';
}
```

Ohjelma 1. Yksinkertainen Angular-komponentti [10]

Ohjelmassa 1 on sekä näkymä että tyylit kirjoitettu komponentin kanssa samaan tiedostoon. Ohjelmassa näkyy myös yksisuuntainen tiedon sidonta <h1>-elementin sisällä olevassa title-parametrissa, jonka arvo on määritelty AppComponent-luokassa. Ohjelmassa käytetään string-tyyppistä muuttujaa tiedonsidonnassa, mutta Angularissa voi sitoa minkä tyyppisen muuttujan hyvänsä mihin tarkoitukseen haluaa. Lisäksi on mahdollista sitoa tapahtuma suorittamaan komponentissa määritelty funktio esim. käyttäjän klikatessa napia, joka on esitetty ohjelmassa 2.

Toinen tapa [10] tiedonsidontaan on nimeltään kaksisuuntainen tiedonsidonta, joka on esitetty ohjelmassa 2. Kaksisuuntaiseen tiedonsidontaan käytetään ngModel-direktiiviä, joka lisää toiminnallisuutta HTML-elementille. Direktiivejä voi määritellä itse, mutta lisäksi on olemassa valmiita direktiivejä, joilla saa lisättyä elementeille perustoiminnallisuksia. Yksinkertainen esimerkki direktiivistä on ngIf, jolla saa jonkin komponentin piilotettua tai olemaan näkyvässä riippuen muuttujan arvosta, jonka direktiiville antaa. HTML-attribuutti ngIf-direktiivillä voisi näyttää esimerkiksi tältä: <p *ngIf="clicked">Klikattu</p>. Tällöin Klikattu-teksti näytetään, mikäli clicked-muuttuja on tosi. Ennen ngIf-tekstiä käytetään tähtimerkkiä, joka piilottaa template-tagin käytön.

```

import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <input type="text" [(ngModel)]="user.username"> // käyttäjänimi
    <input type="password" [(ngModel)]="user.password"> // salasana

    <button click="sendUser()">Send</button> // Lähetä-nappi

  `
})
export class AppComponent {
  private user = {
    username = '',
    password = ''
  }

  sendUser() {
    console.log(this.user);
  }
}

```

Ohjelma 2. *Kaksisuuntainen tiedonsiirto Angular-komponentissa [10]*

Ohjelmassa 2 on esitetty, miten voi toteuttaa kaksisuuntaisella tiedonsidonnalla tiedon lukeminen lomakkeesta komponenttiin. Tähän käytetään direktiiviä ngModel, joka on syntaksiltaan hieman erikoinen, koska se yhdistää ominaisuuden ja tapahtuman sitomisen. Ohjelmassa käyttäjän syöttämä käyttäjänimi ja salasana on sidottu komponentin yksityiseen muuttujaan nimeltä user. Kun käyttäjänimeä tai salasanaa muutetaan, ngModel-direktiivin ansiosta muutos päivittyy myös komponentin muuttujaan. Lisäksi ohjelmassa on esimerkki tapahtuman sitomisesta: kun käyttäjä klikkaa nappia ”Send”, suoritetaan funktio sendUser.

3.1.2 Prototyypin toteutus

Angularin omilla sivuilla [11] oli jo kesällä 2016 tutoriaali, jossa opastettiin yksinkertaisen Angular-sovelluksen luonti. Angular CLI [12], jolla pystyy nykyään muun muassa luomaan uuden Angular-projektin komentoriviltä, oli tuolloin vielä beta-vaiheessa. Siksi projektin luomiseen käytettiin tutoriaalın tiedostoja ja kansiorakennetta. Tutoriaali ja suurin osa muustakin materiaalista ja ohjeista oli olemassa vain TypeScriptille, joten prototyypissäkin päädyttiin käyttämään TypeScriptiä Angularin kanssa.

TypeScript [13] mahdollistaa vahvan tyyppityksen JavaScript-ohjelmointikielelle, jossa ei normaalisti ole ollenkaan tyyppitystä. Tyyppityksellä koodista saadaan selkeämpää ja järjestelmällisempää. Tyyppityksen ansiosta [14] pelkästään koodia katsomalla näkee, mitä kukin muuttuja sisältää, ja muuttujalle ei pystytä tekemään toimintoja, jotka eivät muuttujan tyyppille sovellu. TypeScriptillä [13] kirjoitettu koodi tarvitsee kuitenkin kääntää

normaaliksi JavaScript-koodiksi, jotta selaimet pystyvät ymmärtämään ja ajamaan koodia.

Angularin [11] sivulla olevien ohjeiden avulla prototyypille saatiin luotua yhden sivun sovelluksen tarvitsemat perustoiminnallisuudet kuten reititys, eli sivujen välillä siirtyminen niin, että osoiterivillä muuttuu polku, mutta sovellusta ei ladata uudestaan. Tutoriaalissa esiteltiin myös käsite palvelut (services), jotka huolehtivat muun muassa tiedon hakemisesta ja tallentamisesta. Prototyypin tässä vaiheessa palvelut huolehtivat esimerkkidatan lukemisesta paikallisesta tiedostosta, koska käytössä ei vielä ollut palvelinpäätä. Esimerkkipalvelu on kuvattu ohjelmassa 3.

```
import { Injectable } from '@angular/core';
import { MockProducts } from './mock-products';

@Injectable()
export class ProductService {

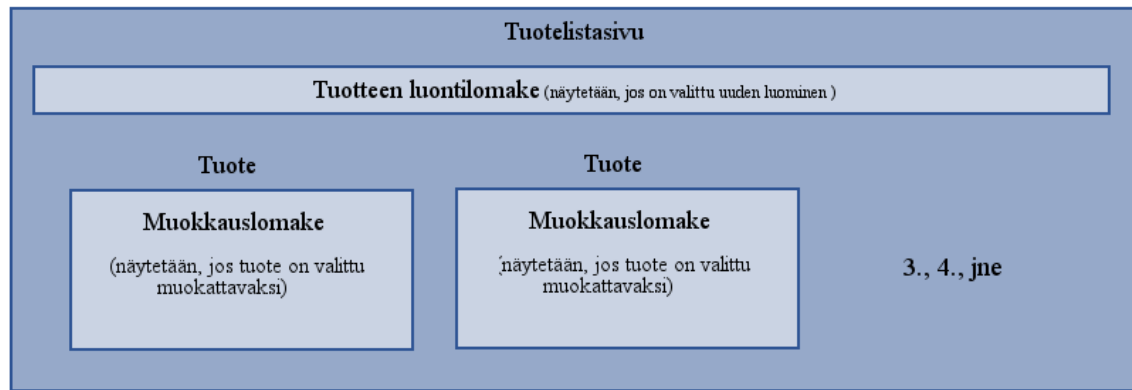
  // getting all products
  getProducts() {
    return Promise.resolve(MockProducts);
  }

  // getting products of specific year
  getProductsWithYear(year:number) {
    return Promise.resolve(MockProducts)
      .then(products => products.filter((product) => product.startYear <= year
        && (product.endYear >= year || product.endYear == 0)));
  }
}
```

Ohjelma 3. *Angular-palvelu, joka palauttaa tuotteita*

Ohjelmassa 3 on tuotepalvelu, joka sisältää kaksi funktiota. Ennen palvelua on kirjoitettu `Injectable()`, jotta koodia käännettäessä palveluun osataan yhdistää tarvittavat ominaisuudet, joiden ansiosta luotu palvelu käyttäytyy oikein. Toinen funktio palauttaa tiedostossa *mock-products* olevan tuotelistan. Toinen funktio palauttaa listan tuotteista, jotka ovat käynnissä annettuna vuonna. Tässä vaiheessa prototyyppiä oltaisiin voitu myös olla käyttämättä palveluita, koska käytännössä palveluissa vain luetaan tiedostosta lista tai muu muuttuja, ja palautetaan se. Käyttämällä palveluita mahdollistettiin kuitenkin palvelinpäähän tehtävien kutsujen helppo lisäys. Kun kutsut lisättäisiin, palvelua kutsuttaisiin edelleen samalla tavalla ja palvelu palauttaisi samalla tapaa listan.

Seuraavaksi tuli huolehtia jossakin komponentissa palvelun kutsumisesta ja tuotelistan näyttämisestä. Rakennetta eriteltiin niin, että listassa käytettiin kolmea eri komponenttia. Varsinainen tuotelistan sisältävä komponentti sisältää tuotteita katselutilassa. Mikäli jokin tuote valittaisiin muokattavaksi, tuotteen tietojen tilalla näytettäisiin tuotteen muokauskomponentti. Lisäksi mikäli valittaisiin uuden tuotteen lisääminen, näytettäisiin uuden tuotteen luomiseen tarkoitettu komponentti. Komponentit on esitetty kuvassa 4.



Kuva 4. Prototyypin komponentit

Tässä vaiheessa prototyypin toteutusta tuli eteen ensimmäinen ongelma, joka tulisi esiintymään monta kertaa kehityksen aikana, nimittäin tiedon välitys komponentilta toiselle. Kun tuotelistakomponentissa valitaan muokattavaksi jokin tuote, tulee tuotteen muokauskomponentille välittää kyseinen tuote tietoineen. Lisäksi tuotteen muokauskomponentista tulee välittää tieto sovelluksen juurikomponentille siitä, että komponentissa on tallentamattomia muutoksia, jotta pystytään estämään siirtyminen sivulta pois kesken muokkauksen. Muokauskomponentilta tulee myös välittää tieto tallennuksesta, jolloin muokauskomponentti tulisi poistaa näkymästä ja siirtyä näyttämään tuotteen tietoja.

Ylemmältä komponentilta sisemmälle komponentille (esim. tuotelistakomponentilta tuotteen muokauskomponentille) tiedon välitys onnistui yksinkertaisesti antamalla komponentille parametrin nimi hakasuluissa ja arvoksi välitettävä muuttuja. Välitettävä muuttuja tulee olla ulommassa komponentissa merkitty julkiseksi, jotta sen voi välittää sisemmälle komponentille. Sisemmän komponentin luokassa `@Input()`-määreellä muuttujan nimen edessä ilmoitetaan, että muuttuja on annettu komponentille parametrina.

Tiedon kulku toiseen suuntaan (tuotteen muokauskomponentilta tuotelistakomponentille) toimi antamalla komponentille hakasuluissa komponentissa käytettävä parametrin nimi ja arvoksi ulommassa komponentissa suoritettava funktio. Sisemmän komponentin luokassa laitetaan `@Output()`-määre muuttujan nimen eteen ja annetaan sen arvoksi `new EventEmitter()`, jolloin muuttujaa voidaan kutsua sisemmässä funktiossa tarpeellisessa kohtaa `this.muuttuja.emit()`, jolloin ulommassa funktiossa suoritetaan määritelty funktio. Emit-funktiolle pystyy lisäksi antamaan parametreja, jotka välittyvät ulommassa komponentissa suoritettavalle funktiolle.

Tiedon kulkemisen syntaksi on selkeä, mutta tuottaa ongelmia esimerkiksi tilanteessa, jossa halutaan pyytää varmistus sivulta poistumiseen, kun on tallentamattomia muutoksia. Tallentamattomia muutoksia voi olla monissa eri komponentissa, ja monissa eri komponenteissa voidaan tehdä jokin muutos, jonka seurauksena tallentamattomat tiedot häviävät. Esimerkiksi kun ollaan muokkaamassa jotakin tuotetta, muutokset häviävät sekä listasivulta pois siirryttäessä, että valitettaessa jokin muu tuote muokattavaksi, kun vain

yksi tuote voi olla kerrallaan muokattavana. Tällöin tieto siitä, että tuotteen tietoja ei ole tallennettu tulee olla sekä siinä komponentissa, joka huolehtii eri sivuille siirtymisestä, että listasivulla. Navigointikomponentti tulisi luultavasti olemaan yksi uloimmista komponenteista, joten tuotteen muokkauskomponentin ja navigointikomponentin välissä tulisi luultavasti olemaan useita komponentteja, joiden läpi tieto tulisi kuljettaa. Tämä tekisi toteutuksesta hankalaa, kun prototyypin pohjalta lähdetäisiin kehittämään tuotetta.

Toinen osuus prototyypissä oli toteuttaa jokin kaavio, jolla voidaan esittää yhden tai useamman tuotteen tietoja graafisessa muodossa. Tähän valittiin toteutettavaksi viivadiagrammi vanhasta käyttöliittymästä. Kaaviokirjastoksi valittiin ng2-charts, joka tekee Chart.js-kirjaston käytettäväksi Angular-projektissa. Viivakaavion toteutus oli melko suoraviivaista ng2-charts ohjeiden avulla. Lisäksi kaavion ominaisuuksista löytyi paljon lisätietoa Chart.js-kirjaston sivuilta.

3.2 React-prototyyppi

Toinen prototyyppi tehtiin React.js- ja Redux.js-kirjastojen avulla. React tuo joitain olio-ohjelmoinnin käsitteitä JavaScriptiin, ja huolehtii käytännössä siitä, miten asiat saadaan näkymään selaimessa. Redux määrittelee ohjelmalle tilakäsitteen eli paikan, jonne säilötään ohjelmassa näytettävä tieto ja status siitä, mitä ollaan tehty tai tekemässä. Esimerkiksi lista näytettävistä asioista tai lomakkeen kenttien sisältö säilötään Reduxin tilaan.

Wapicella oli näistä kirjastoista jo kokemusta, joten toinen prototyyppi auttoi lähinnä päättämään, kumpi ohjelmistokehys sopii paremmin tämän tuotteen toteutukseen. Lisäksi toisella ohjelmistokehyksellä prototyypin toteuttaminen auttoi kehittäjää saamaan paremman kuvan web-sovellusten asiakaspään kehityksestä.

3.2.1 React.js

React.js [15] on JavaScript-ohjelmistokehys, joka kehitettiin Facebookin tarpeisiin ratkaisemaan monimutkaisen ja vaihtuvaa tietoa sisältävän käyttöliittymän ongelmia. Toisin kuin monet muut JavaScript-ohjelmistokehukset, kuten esimerkiksi Angular.js ja Ember.js, React ei toteuta MVC-mallia kokonaisuudessaan. React toteuttaa MVC-mallista vain yhden osan, näkymän, eli kuvaa millainen ohjelman käyttöliittymä on ja millaisia muutoksia siihen tulee tehdä, kun tiedot muuttuvat.

Ydinkäsite Reactissa ovat komponentit, joilla kuvataan ohjelman käyttöliittymä. Komponenttien avulla React-sovellusta on helppo muokata, koska komponentteja pystyy luomaan ja yhdistelemään haluamallaan tavalla. Ohjelmassa 4 on kuvattu esimerkki ES6-tyylisestä komponentin luomisesta Reactilla.

```

class MyClass extends React.Component {
  render() {
    return (
      <div>
        Hello World!
      </div>
    )
  }
}

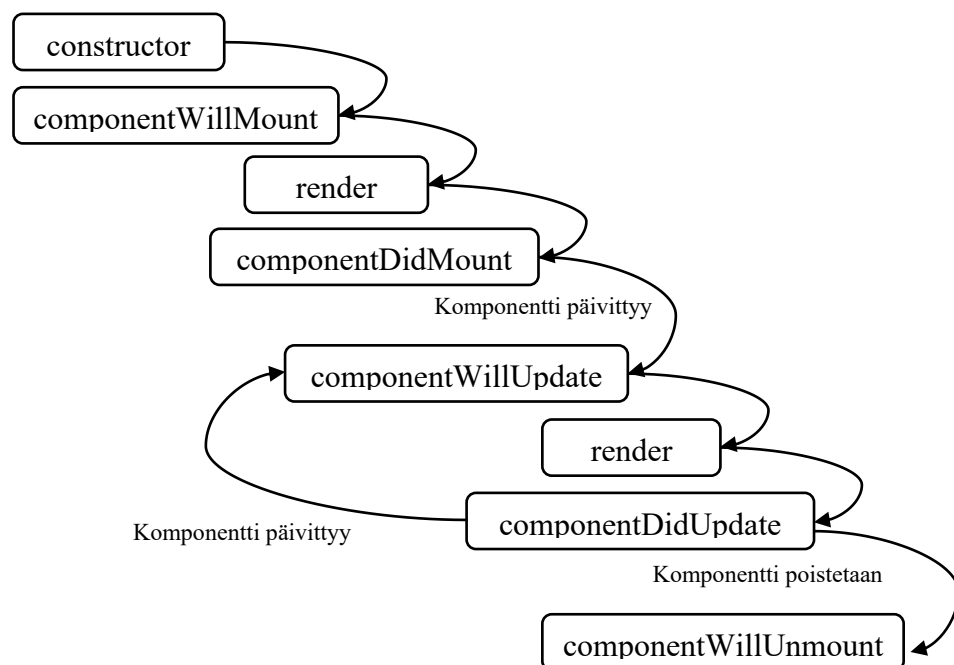
```

Ohjelma 4. Yksinkertainen React-komponentti [15]

Komponenteille [15] on määritelty erilaisia elinkaarifunktioita, joista yksi on render-funktio, joka esiintyy myös ohjelmassa 4. Muita elinkaarifunktioita ovat esimerkiksi:

- Rakentaja (constructor), joka suoritetaan ennen kuin komponentti lisätään DOM-puuhun.
- Komponentin päivittymisen alussa (componentWillUpdate) ja päivittymisen jälkeen (componentDidUpdate) suoritettavat funktiot.
- Ennen komponentin poistamista DOM-puusta (componentWillUnmount) ja komponentin DOM-puuhun lisäystä ennen (componentWillMount) ja lisäyksen jälkeen (componentDidMount) suoritettavat funktiot.

Mainitut elinkaarifunktiot tapahtuvat kuvassa 5 esitettyssä järjestyksessä, kun komponentti lisätään DOM-puuhun, komponentissa tapahtuu päivitys, tai komponentti poistetaan näkymästä.



Kuva 5. Yksinkertaistettu React-komponentin elinkaari

Render-funktio [15] palauttaa JSX-syntaksilla, mitä komponentti kuvantaa käyttöliittymään. JSX-syntaksi on staattisesti tyyipitetty olio-ohjelmointikieli, joka koostuu DOM-elementeistä sekä React-komponenteista XML-syntaksissa. JSX-syntaksia ei ole pakko käyttää Reactin kanssa, mutta se tekee komponenttien luomisesta helpompaa ja syntaksi on yleisesti hyväksytty tapa kirjoittaa Reactia. JSX-syntaksi on suosittu muun muassa sen takia, että se kääntyy JavaScriptiksi ja että se näyttää XML-/HTML-syntaksilta ja on siten kehittäjille tutun oloinen.

Komponenteilla on lisäksi ominaisuus-käsite (`React.Properties`), johon voidaan viitata koodissa *this.props*. Tämä palauttaa JavaScript-objektin, joka sisältää komponentille komponentin luonnin aikana annetut muuttujat ja funktiot. Komponenttien ominaisuudet käyttäytyvät samaan tapaan kuin funktioiden vakioparametrit eli niitä ei voi muokata komponentin sisällä. Komponentin luovassa komponentissa, jossa välitetään komponentille ominaisuudet, voidaan kuitenkin päivittää välitettävien ominaisuuksien arvot. Myös ominaisuuksien muuttumista pystyy tarkastelemaan elinkaarifunktiolla `componentWillReceiveProps`, jonka jälkeen suoritetaan myös `componentWillUpdate`-elinkaarifunktio ja sitä seuraavat elinkaarifunktiot [16].

Mikäli [15] halutaan muokata jotakin komponentin sisällä, voidaan muokata komponentin tilaa (*this.state*). Tila määritellään komponentille rakentajassa ja sitä voidaan muokata komponentin sisällä sen elinkaaren ajan. Tilaa tulee muokata vain `setState`-funktion avulla, jotta tulee suoritettua oikeat komponentin elinkaarifunktiot (komponentin päivittämiseen liittyvät funktiot) tilan muuttamisen jälkeen. On kuitenkin hyvä tapa olla käyttämättä komponentin tilaa mikäli mahdollista, sillä tilan käyttäminen lisää komponenttien kompleksisuutta.

Elinkaarifunktiota, ominaisuuksia ja tilaa käyttävä esimerkkiohjelma on kuvattu alla ohjelmassa 5. Ohjelmassa on käytössä elinkaarifunktioista rakentaja (constructor), jossa määritellään ohjelman alkutilaksi, että ei ole klikattu tekstiä, jolloin näkyy ”Click me” – teksti käyttöliittymässä. Mikäli käyttäjä klikkaa tekstiä, suoritetaan funktio `onClick`, joka vaihtaa komponentin tilan `clicked`-muuttujan todeksi. Muutos aiheuttaisi päivittämiseen liittyvien elinkaarifunktioiden suorittamisen. Ohjelmassa näitä ei ole, joten suoritetaan vain `render`-funktio uudestaan, joka päivittää komponentin niin, että näytetään teksti, jossa tervehditään käyttäjää.

```

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

    this.state = {
      clicked: false
    }
  }

  onTextClicked() {
    this.setState({ clicked: true });
  }

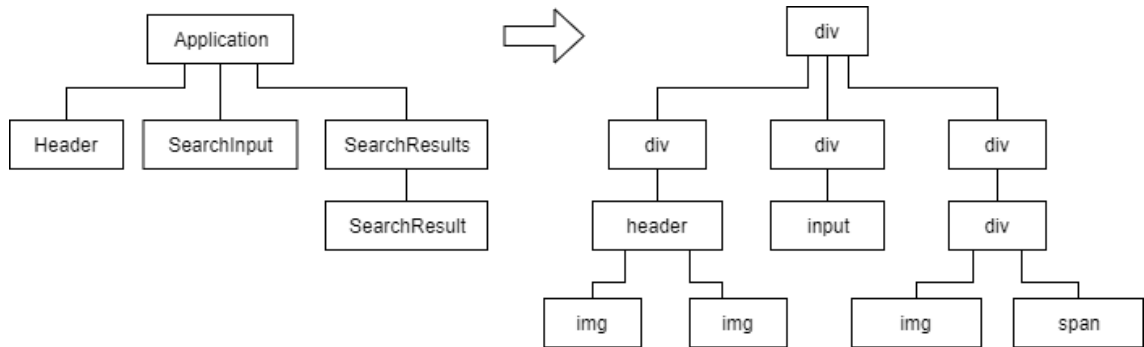
  render() {
    return (
      <div>
        {this.state.clicked &&
          <p>
            Hello {this.props.user}!
          </p>
        }
        {!this.state.clicked &&
          <p onClick={this.onTextClicked.bind(this)}>
            Click me
          </p>
        }
      </div>
    )
  }
}

```

Ohjelma 5. *React-komponentti, jossa käytössä elinkaarifunktio, ominaisuudet ja tila*

Tärkeä [17] käsite Reactissa on myös virtuaalinen DOM. Kaikki web-sivut esitetään HTML-koodin avulla niin, että HTML-koodi muodostaa HTML DOM –puun, jossa HTML-elementit ovat puun solmuja. Nykyään kuitenkin web-sivut ja siten myös DOM-puut ovat hyvin suuria ja yhden sivun sovelluksen arkkitehtuuri yleistä, minkä takia puuta tarvitsee päivittää usein ja paljon. Tämä aiheuttaa ongelmia suorituskyvylle sekä tekee kehityksestä monimutkaista ja vaikeaselkoista.

React ratkaisee DOM-puun kehittämisiongelmat ilmoittamalla, miltä komponentin tulisi näyttää sen sijaan, että DOM-puuta muokattaisiin manuaalisesti. Suorituskykyongelma ollaan Reactissa ratkaistu hyödyntämällä virtuaalista DOMia. Virtuaalinen DOM on HTML DOM –puun abstraktio, tai täsmällisemmin sanottuna virtuaalinen DOM on Reactin lokaali ja yksinkertaistettu kopio HTML DOM -puusta. Alla olevassa kuvassa 5 on esimerkki yksinkertaisen React-ohjelman muodostamasta puurakenteesta.



Kuva 6. Yksinkertaisen React-ohjelman puurakenne

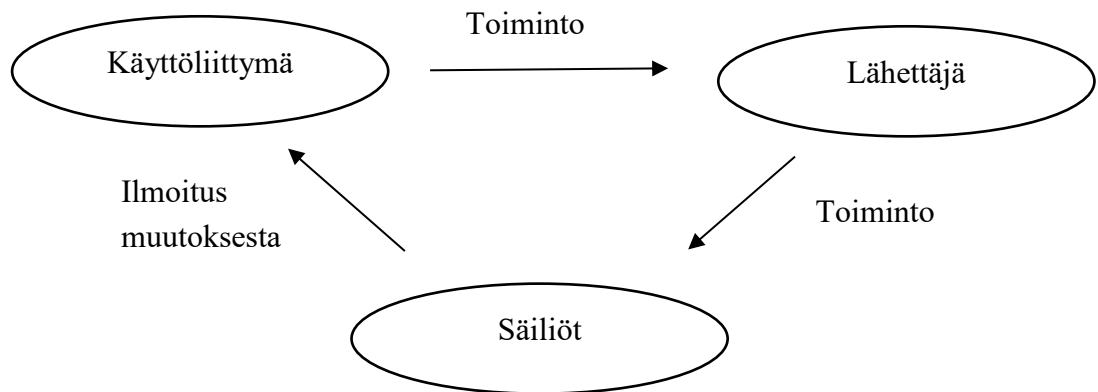
Kuvassa 6 on esitetty vasemmalla puolella ohjelman puurakenne React-komponenttien avulla ja oikealla puolella vastaava rakenne React-elementtien avulla. React-elementit ovat varsinaisia HTML-elementtejä, ja React-komponentit aikaisemmin kuvattuja komponentteja. Kun React-komponentin tila tai ominaisuus muuttuu ja näkymä tarvitsee päivittää, pyritään tekemään mahdollisimman vähän muutoksia DOM-puuhun. Tämä saavutetaan niin, että React-komponentit muunnetaan React-elementeiksi, joita käytetään virtuaalisessa DOMissa. Virtuaalinen DOM pystyy vertailemaan React-elementtejä nopeasti ja tehokkaasti, ja päivittää vain ne elementit, joihin on tullut muutoksia. [17]

3.2.2 Redux.js

Redux.js [18] on JavaScript-kirjasto, joka toteuttaa ennustettavalla tavalla käyttäytyvän tilasäiliön JavaScript-sovelluksille. Käytännössä ennustettavuus tarkoittaa sitä, että tiedetään miten tila muuttuu kunkin toiminnon seurauksena. Redux kehittyi vuonna 2015 Flux-arkkitehtuurista, joka uudelleenmääritteli MVC- ja MVVM-mallit ja esitteli asiakaspään toimintojen käsittelyyn kaksisuuntaisen tiedon sidonnan sijaan uuden käsitteen nimeltä yksisuuntainen tietovirta.

Flux-arkkitehtuurissa käyttöliittymässä tapahtuvat toiminnot käsitellään yksi kerrallaan erilaisten käsitteiden avulla. Käsitteet ovat:

- toiminto (action): mikä tahansa muutos ohjelmassa, esimerkiksi hiiren klikkaus tai Ajax-kutsu
- lähettäjä (dispatcher): yksittäinen kohta ohjelmassa, jonne voi lähettää toimintoja käsiteltäväksi ja joka huolehtii niiden lähetyksestä oikeaan paikkaan
- säiliö (store): paikka, jossa ylläpidetään ohjelman tilaa sekä käskyjä lähettäjältä.



Kuva 7. Toimintojen ketju Flux-arkkitehtuurissa [18]

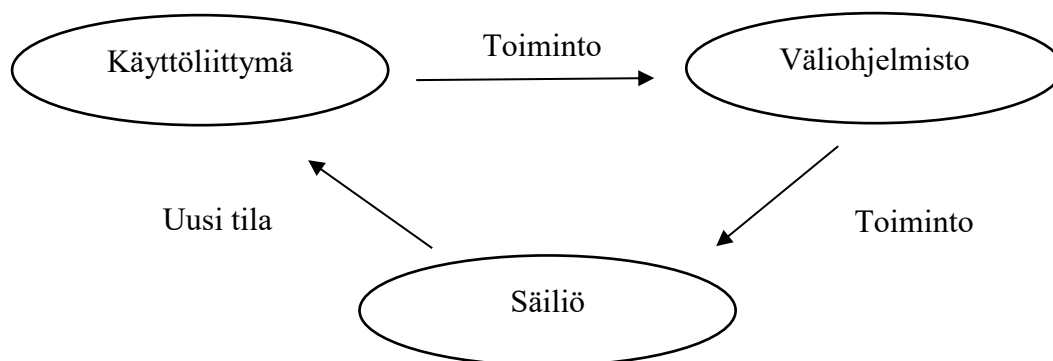
Yksinkertaisin [18] toimintojen ketju Flux-arkkitehtuurissa on seuraava:

1. Säiliöt kuuntelevat tietyn tyyppisiä toimintoja
2. Toiminto lähetetään käyttöliittymästä
3. Lähetäjä ilmoittaa säiliöitä toiminnosta
4. Säiliöt päivittävät tilansa toiminnosta riippuen
5. Näkymä päivittyy vastaamaan säiliöiden uutta tilaa
6. Seuraava toiminto voidaan käsitellä

Yllä kuvattu toimintojen ketju varmistaa sen, että on mahdollista päätellä: miten toiminnot liikkuvat systeemissä, mikä aiheuttaa tilamuutoksen ja miten säiliön tila muuttuu. Yksi klikkaus Flux-arkkitehtuurissa luo siten vain yhden toiminnon, joka muuttaa säiliön tilaa ja päivittää näkymän. Muut mahdolliset toiminnot, jotka luodaan prosessin aikana, laitetaan jonoon odottamaan käsittelyvuoroa.

Redux eroaa Flux-arkkitehtuurin konsepteista muutamalla tavalla. Ensimmäinen Reduxissa on vain yksi säiliö. Säiliö lähettää ja käsittelee toiminnot itse eli erillisille lähettäjäille ei ole tarvetta. Toinen eroavaisuus on se, että säiliö välittää toiminnot tilaa muuttaville funktioille (reducers), joita ei Flux-arkkitehtuurissa ole.

Lisäksi Reduxissa on mahdollista lisätä väliohjelmistoja, joiden läpi toiminnot kulkevat. Väliohjelmistot voivat muokata, pysäyttää tai lisätä muita toimintoja. Väliohjelmisto voi esimerkiksi tarkistaa, onko käyttäjällä oikeus suorittaa kyseinen toiminto tai lähettää API-kutsun palvelimelle. Väliohjelmistot pääsevät lisäksi käsiksi säiliön tilaan sekä Reduxin dispatch()-funktioon, minkä ansiosta väliohjelmistot ovat erittäin monikäyttöisiä sekä tehokkaita kokonaisuuksia. Toimintojen kulku Reduxissa on kuvattu kuvassa 7.



Kuva 8. Toimintojen ketju Reduxissa [18]

Käytännössä Redux-toiminto, joilla muokataan ohjelman tilaa, on yksinkertainen JavaScript-objekti, joka välitetään säiliölle. Esimerkkitoiminto on kuvattu ohjelmassa 6. Usein objektit palautetaan funktioista, jotta niitä voidaan käyttää useassa paikassa ja muokata lähetettävää toimintoa yhden tai useamman parametrin avulla. Näitä funktioita kutsutaan nimellä toimintojen luoajat (action creators).

```

{
  type: 'INCREMENT',           // toiminnon tyyppi
  payload: {                  // toiminnon lisätietoa
    counterId: 'main',
    amount: -10
  }
}

```

Ohjelma 6. Yksinkertainen Redux-toiminto [18]

Redux-toiminnolla eli objektilla on aina ominaisuus tyyppi (type), joka määrittää sen, miten Reduxin tila muuttuu. Ohjelmassa 6 olevan toiminnon tyyppi on INCREMENT. Tyyppien nimet on hyvä valita niin, että on selkeää, mitä toiminto tekee. Tässä tapauksessa toiminto kasvattaa laskurin arvoa. Ohjelman 6 toiminnolla on lisäksi payload-niminen ominaisuus, joka antaa lisätietoa siihen, mitä toiminto tekee. Tässä tapauksessa lisätietona on counterId ja amount, joilla kerrotaan, mille laskurille toiminto kohdistuu ja kuinka suurella määrällä laskuria kasvatetaan. Koska amount on negatiivinen (-10), laskurin arvoa oikeasti vähennetään kymmenellä eikä kasvateta.

Reducer-funktiot [18], joilla muokataan säiliön tilaa, ottavat parametreikseen säiliön nykyisen tilan sekä säiliölle lähetetyn Redux-toiminnon. Esimerkki tällaisesta funktiosta on esitetty ohjelmassa 7. Reducer-funktioita on ohjelmassa yleensä lukuisia, ja toiminto lähetetään kaikille näille funktioille. Näin on mahdollista varautua johonkin toimintoon ja toteuttaa tarvittavat muutokset useassa eri paikassa.

```
function calculateNextState(state, action) {
  switch(type) {
    // palauta muutettu säiliö, jos toiminto on INCREMENT
    case 'INCREMENT':
      return {
        ...state,
        counter: state.counter + action.payload.amount
      };
    // palauta tila muuttumattomana muissa tilanteissa
    default:
      return state;
  }
}
```

Ohjelma 7. Yksinkertainen reducer-funktio [18]

Ohjelmissa 6 ja 7 on kuvattu toiminto ja sen käsittelevä funktio, jotka yhdessä muokkaavat säiliössä pidettävää laskurin tilaa vähentämällä laskurin nykyisestä arvosta arvon kymmenen. Toiminto lisäksi välittää counterId-arvon, jolla pystyisi identifioimaan, mitä laskuria kyseinen muutos koskee.

3.2.3 Prototyypin toteutus

Ensimmäisen prototyypin toteutuksen jälkeen oli selvää, että ohjelmassa olisi haasteena tiedon säilöminen ja välitys. Lopullisessa tuotteessa tulisi olemaan paljon tietoa, jota useampi kuin yksi komponentti tulisi tarvitsemaan, joten tiedon säilöminen vain komponenteissa ei toimisi, kun toteutus viettäisiin prototyypistä pidemmälle. Tätä varten prototyypissä otettiin React.js-ohjelmistokehityksen kanssa käyttöön Redux.js-kirjasto. Valinnassa auttoi lisäksi Wapicen aiempi kokemus Reactin kanssa, sillä muissakin projekteissa oli ollut käytössä React ja Redux yhdessä.

Kehittäjälle kumpikaan kirjastoista ei ollut aiemmin tuttu, joten alkuun pääseminen oli haastavaa. Kehittäjän tarvitsi opetella ja ottaa käyttöön samanaikaisesti sekä React että Redux, koska molemmat muuttavat suuresti koko ohjelman rakennetta. Lisäksi siinä missä Angular-kirjastoon sisältyi lähes kaikki välttämättömät kirjastot, Reactin kanssa kehittäjä saa itse valita, mitä kirjastoja käyttää.

Reduxin Github-sivulta löytyi erilaisia esimerkkiprojekteja, joiden pohjalta lähdettiin rakentamaan prototyyppejä. Prototyypin alkuvaiheen kansiorakenne on esitetty kuvassa 8. Actions-kansiossa sijaitsevat funktiot, jotka palauttavat Redux-toimintoja, eli niin kutsutut toimintojen luojat. Reducers-kansiossa määritellään Reduxin tila sekä tilaa muuttavat funktiot. Reducers-kansion index.js-tiedosto yhdistää eri tiedostoissa tai kirjastoissa määritellyt tilasäiliöt yhdeksi tilasäiliöksi eli tässä vaiheessa prototyyppejä basic.js-tiedostossa määritellyn säiliön ja eri kirjaston käyttämät säiliöt.

```

- actions
  - index.js
- components
  - App.js
- containers
  - App.js
- reducers
  - basic.js
  - index.js
- static
  - index.html
  - index.js
- package.json
- webpack.config.js

```

Kuva 9. React-prototyypin kansiorakenne

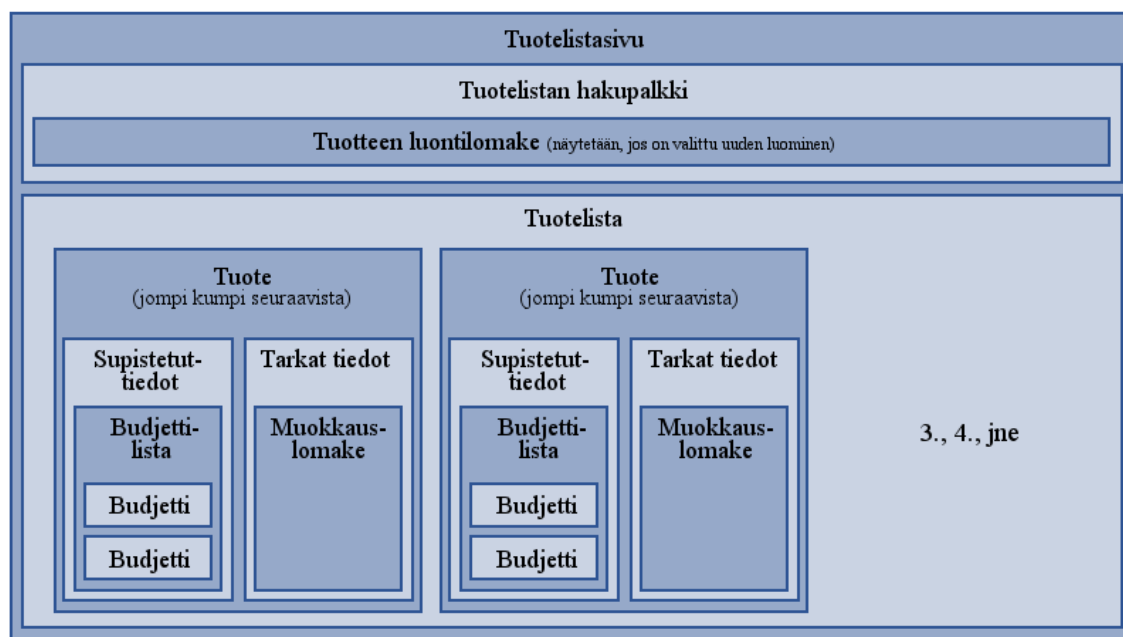
React-komponentit sijaitsevat components-kansiossa ja sen tulevissa alikansioissa. Mikäli jollekin komponentille halutaan välittää parametrina muuttuja, joka on säilötty Reduxin tilaan, tai funktio, jolla voidaan luoda Redux-toiminto, luodaan tiedosto myös containers-kansioon. Containers-kansiossa olevassa tiedostossa käytetään react-redux-kirjastossa määriteltyjä funktioita, joiden avulla saadaan välitettyä React-komponentille alkuperäisten parametrien lisäksi haluttuja Reduxiin liittyviä parametreja.

Kun ymmärrettiin, miten Reactia ja Reduxia käytetään yhdessä, oli prototyypin pidemmälle kehittäminen selkeää. Prototyypille saatiin käyttöön palvelinpää melko aikaisessa vaiheessa, jolloin käyttöön otettiin redux-thunk-kirjasto. Redux-thunk on väliohjelmisto, jolla mahdollistetaan kutsujen lähettäminen palvelinpäähän Redux-toimintoja luovissa funktioissa. Väliohjelmisto otetaan käyttöön projektin juurihakemistossa olevassa index.js-tiedostossa, jossa määritellään myös muut projektin asetukset kuten reititys.

Kun luotiin prototyyppiin tuotelista, tallennettiin listan hakuparametrit, tuotelista sekä tieto siitä, onko tallentamattomia muutoksia, Reduxin säiliöön. Komponentille, joka sisälsi listan rajausvalikot, välitettiin nykyiset rajausvalinnat sekä funktiot, joilla välitettyjä ominaisuuksia muokataan Redux-toimintojen avulla. Kun Redux-toiminto on lähetetty ja Reduxin tila muuttunut, päivittyvät kaikki komponentit, joille on välitetty Reduxissa päivittynyt muuttuja, automaattisesti.

Tuotelistan luontiin käytettiin kymmentä eri komponenttia, koska yksinkertaiset komponentit selkeyttävät ohjelman rakennetta ja koska Reduxin ansiosta prototyypissä ei tarvinnut murehtia tiedon välityksessä komponenttien välillä. Prototyypin komponentit olivat tässä vaiheessa: tuotenäkymä, tuotelistan hakupalkki, uuden tuotteen luontilomake, tuotelista, yksittäinen tuote, tuotteen supistetut tiedot, tuotteen tarkat tiedot, tuotteen

muokkauslomake, tuotteen budjettilista ja tuotteen budjetin budjettitiedot. Näistä komponenteista yhteenkään ei tarvinnut säilöä tietoa komponentin omaan tilaan, vaan kaikki tieto säilöttiin Reduxin tilassa. Komponentit on esitetty sisäkkäin kuvassa 9.



Kuva 10. Tuotelistan komponentit

Prototyypin toiseen osuuteen, kaavion luontiin, käytettiin react-chartjs-kirjastoa. Kirjasto tekee Charts.js-kirjastosta käytettävän React-projektissa samalla tavalla kuin ng2-charts tekee kirjaston käytettäväksi Angular-projektissa. React-chartjs oli siten hyvin samantyylinen käyttää kuin ng2-charts, jota käytettiin ensimmäisessä prototyypissä.

Prototyypin perustoiminnallisuuksiin hyödynnettiin erilaisia kirjastoja. Reactille ja Reduxille on olemassa paljon erilaisia suosittuja ja testattuja kirjastoja joista valita projektiin sopivat kirjastot, koska React ja Redux ovat olleet pitkään käytössä monissa erilaisissa projekteissa. Prototyypissä otettiin käyttöön heti alusta kirjastot Redux Form, react-redux-toastr ja react-redux-i18n.

Redux Form [19] sitoo HTML-lomakkeen arvot Reduxin tilaan. Lisäksi sen avulla pystyy muun muassa alustamaan lomakkeen JavaScript-taulun avulla sekä tyhjentämään lomakkeen alkuperäisiin arvoihin. Budjetointisovelluksesta tulisi olemaan useita suurehkoja, lähemmäs neljäkymmenen kentän lomakkeita, joiden alustaminen ja käsittely helpottuivat kirjaston avulla.

React-redux-toastr [20] tarjoaa valmiin toteutuksen ilmoitusten näyttämiseen käyttäjälle. Prototyypissä käytettiin virhe- ja onnistumisilmoituksia kertomaan käyttäjälle, onnistuiko jokin toiminto vai ei. Ilmoitus näytetään näytön vasemmassa alakulmassa muutaman sekunnin ajan, mikäli tapahtuma onnistui, tai useamman minuutin ajan, mikäli asia epäonnistui. Tällä pyrittiin varmistamaan, että käyttäjä ehtii huomaamaan tapahtuneen virheen.

Käyttäjä pystyy myös sulkemaan ilmoituksen itse, mikäli haluaa ilmoituksen nopeammin pois näkyvistä. Virheilmoituksia käytettiin ilmoittamaan sekä palvelinpäässä tapahtuneesta virheestä, että puuttuvista tai väärin täytetyistä kentistä lomakkeessa. Onnistumisilmoituksia käytettiin, kun asian tallentaminen tai muokkaaminen onnistui.

React-redux-in kirjaston [21] avulla sovelluksen saa kansainvälistettyä ja lokalisoitua. Vaikka asiakasvaatimuksena ei ollutkaan mahdollisuus kansainvälistää sovellusta, todettiin, että sovellus kannattaa kuitenkin toteuttaa niin, että myöhemmin mahdollinen kansainvälistäminen olisi helppoa. Lisäksi kirjaston avulla saatiin esitettyä numerot oikeiden välimerkkien avulla esim. normaali JavaScript-numero 12345.66 saatiin esitettyä muodossa 12 345,66.

Prototyyppeihin etsittiin lisäksi kirjastoa, jolla voisi toteuttaa sovelluksen ulkoasun päälle ilmestyvät ikkunat esimerkiksi tilanteissa, joissa luodaan tai muokataan jotakin artikkelia. Sopivaa kirjastoa ei kuitenkaan löytynyt sillä hetkellä, joten toiminnallisuus päätettiin toteuttaa itse. Ikkuna toteutettiin Reduxin avulla niin, että React-komponentti näytetään vain, kun Reduxin tilassa on määritelty jonkin niminen ikkuna näkyväksi, eli on luotu Redux-toiminto, joka asettaa ikkunan tyyppin. Toiminnossa määritellään ikkunan tyyppin lisäksi mahdolliset muut määreet, jotka asetetaan Reduxin tilan modalProps-objektiin, joka välitetään ikkunakomponentille. Toteutettu tilasäiliö ja sitä muokkaava funktio on kuvattu ohjelmassa 8.

```
const initialState = {
  modalType: null,
  modalProps: {}
}

function modal(state = initialState, action) {
  switch(action.type) {
    case 'OPEN_MODAL':
      return {
        modalType: action.modalType,
        modalProps: action.modalProps
      };
    case 'CLOSE_MODAL':
      return initialState
    default:
      return state;
  }
}

export default modal;
```

Ohjelma 8. *Käyttöliittymän päälle aukeavan ikkunan reducer-funktio*

Yllä olevassa funktiossa käytetään ES6-tyylistä oletusparametrin määrittelyä, jolla saadaan alustettua tila ensimmäisellä kerralla halutunlaiseksi. Funktiossa käsitellään kaksi Redux-toimintoa, eli toiminnot joilla avataan tai suljetaan ikkuna. Mikäli Redux-toiminto

on jokin muu, eli jokin toiminto, joka muuttaa eri tiedostossa määriteltyä tilaa, palaute-
taan tila muuttumattomana.

3.3 Teknologian valinta

Angular tuntui ohjelmistokehyksenä toimivalta prototyypin toteutuksen aikana. Asiaa auttoi se, että prototyyppi oli vielä suhteellisen yksinkertainen ohjelma, ja Angularin sivuilla oli hyvä tutoriaali yksinkertaisen ohjelman tekemiseksi. Lisäksi kehittäjän kokemuksen puute JavaScript-ohjelmistokehyksistä aiheutti sen, ettei kehittäjällä ollut tarvittavaa vertailupohjaa.

Prototyypin toteutuksen aikana tuli kuitenkin eteen ongelmia tiedon välityksen kanssa eri komponenttien välillä, vaikka komponentteja oli prototyypissä vain muutama. Lopullisessa tuotteessa ongelma olisi suurempi, koska komponentteja olisi hyvin paljon. Lisäksi kävi ilmi, että vaikka Angular oli jo julkaisuprosessissa vaiheessa julkaisuehdokas, oli ohjelmistokehyksessä vielä paljon keskeneräisiä asioita.

Alun perin Angular valittiin ensimmäisen prototyypin teknologiaksi siksi, että sen oletettiin olevan lähes valmis ja käytettävissä pian tuotantoon menevissä ohjelmissa. Angulariin tuli kuitenkin vielä paljon muutoksia prototyypin teon aikana sekä sen jälkeen. Prototyyppiä työstettiin touko- ja kesäkuussa, ja lopullinen versio Angularista julkaistiin syyskuussa 2016 [22]. Julkaisuehdokkaasta julkaistiin seitsemän versiota touko- ja syyskuun välillä, jotka muuttivat mm. angular/router-osuutta huomattavasti. Tämän takia prototyyppi oli vanhentunut syksyyn mennessä, ja prototyypissä oli käytössä paljon käytöstä poistuneita syntakseja. Angularin keskeneräisyydestä johtuen tuotteen toteuttaminen olisi tarvinnut aloittaa tyhjästä, mikäli päädyttäisiin käyttämään teknologiana Angularia.

Myöhemmin kävi ilmi, että React-prototyypin kanssa käytetty Redux olisi ollut käytettävissä myös Angularin kanssa. Reduxin avulla oltaisiin voitu välttää ongelmat tiedon välityksessä eri komponenttien välillä, ja se tai jokin vastaava kirjasto olisi pakko ottaa käyttöön, mikäli lopullinen tuote tehtäisiin Angularilla. Uuden kirjaston käyttöönotto tarkoittaisi sitä, että ohjelman toimintamalli muuttuisi paljon eli myös tästä syystä prototyypin koodia ei pystyittäisi hyödyntämään lopullisessa tuotteessa, vaan ohjelma tulisi aloittaa puhtaalta pöydältä.

React-prototyypin kanssa ei tullut eteen yhtä paljon ongelmia kuin Angular-prototyypin kanssa, ja eteen tulleet ongelmat pystyttiin ratkaisemaan prototyyppiä tehdessä. React on ollut käytössä useampia vuosia monilla isoilla sivustoilla esim. Netflix, Yahoo!, Facebook ja Atllassian [23], minkä ansiosta se on ohjelmistokehyksenä valmiimpi kuin Angular. Reactista on olemassa myös paljon erilaisia blogitekstejä ja ohjeita, ja keskustelufoorumeilla on paljon Reactiin liittyviä kysymyksiä vastauksineen, joista on usein hyötyä.

Reactille on lisäksi toteutettu paljon enemmän erilaisia kirjastoja, joita pystyttiin hyödyntämään prototyypin teossa. Angularille tehdyt kirjastot olivat usein vanhentuneita, koska Angularista julkaistiin niin taajaan uusia versioita.

Kesän lopulla molempien prototyyppien teon jälkeen päädyttiin siihen tulokseen, että lopullisessa tuotteessa käytettäisiin Reactia. Angular todettiin liian keskeneräiseksi, ja lisäksi mikäli oltaisiin valittu Angular asiakaspään teknologiaksi, Angular-prototyyppiä ei oltaisi pystytty hyödyntämään lopullisessa tuotteessa, koska prototyypissä ei ollut käytössä Reduxin kaltaista tilasäiliötä ja kirjasto kehittyi niin nopeasti. React-prototyyppiä sen sijaan pystyttäisiin hyödyntämään tuotteessa. Tyhjistä aloittaminen ei välttämättä olisi ollut huono asia, sillä toista kertaa tehdessä olisi tiedetty tulevat ongelmat ja pystytty varautumaan niihin etukäteen. Asioita olisi luultavasti saatu toteutettua paremmin, mutta tyhjistä aloittaminen olisi vienyt myös paljon enemmän aikaa.

Prototyyppien tekemiseen oli käytetty useampia kuukausia, joten se, ettei prototyyppiä pystyttäisi käyttämään lopullisessa tuotteessa olisi käytännössä tarkoittanut, että kuukausien työstä olisi ollut vähemmän hyötyä. React-prototyypissä oli lisäksi jo käytössä hyödyllisiä kirjastoja, joiden ansiosta prototyyppi oli toteutuksessa pidemmällä kuin Angular-prototyyppi.

Teknologiavalinnan jälkeen React-prototyyppiä työstettiin pidemmälle vielä jonkin aikaa, koska kehittäjällä oli ylimääräistä aikaa. Prototyyppiin toteutettiin tuotteiden näkymät ja tuotteen budjetin ja budjetin kulujen muokkaamiset, johon sisältyi paljon erilaisten lomakkeiden käyttämistä. Kulujen lomakkeet olivat laajimmat koko tuotteessa, ja niiden toteutuksessa oli erityisen paljon hyötyä Redux Form -kirjastosta.

Prototyyppiä käytettiin ohjelman uuden version esittelyssä asiakkaalle, mistä oli huomattavasti hyötyä myyntiprosessissa, koska asiakkaalla ei ole paljoa teknistä taustaa. Konkreettinen esimerkki siitä, miltä uudistettu ohjelma voisi näyttää ja miten se voisi toimia, auttoi asiakasta ymmärtämään paremmin uuden version tekemisen hyödyt. Syksyllä tehtiin myös työmääräarvio asiakaspään uudistamisesta, mikä oli helpompaa, kun vanhaan järjestelmään oli ehditty tutustumaan kunnolla prototyyppiä tehdessä.

4. TOTEUTUS

Keväällä 2017 aloitettiin varsinainen toteutus React.js-ohjelmistokehyksellä käyttäen keuhällä 2016 tehtyä React-prototyyppiä toteutuksen pohjana. Prototyypin koodiin tarvitsi tehdä jonkin verran muutoksia ennen kuin toteutusta jatkettiin pidemmälle.

4.1 Prototyyppiin tehdyt muutokset

Ennen tuotteen kehityksen jatkamista järjestettiin prototyypin koodin katselmointi. Katselmoijana toimi vanhempi kehittäjä, jolla oli enemmän kokemusta Reactin ja Reduxin käytöstä. Katselmoinnilla haluttiin varmistaa perusrakenteiden olevan kunnossa, jotta kehitystyö sujuisi mahdollisimman ongelmitta. Katselmointi toteutettiin palaverina, jossa kehittäjä esitteli ja keskusteli katselmoijan kanssa koodin eri osista. Katselmoinnin perusteella päätettiin muokata useampia osioita koodissa.

Tärkeimpänä, ja myös kehittäjälle myöhemmin mieleen tulleen, asiana projektissa otettiin käyttöön konfigurointitiedosto, josta luetaan muun muassa palvelinpään osoite. Prototyypissä oltiin kirjoitettu palvelinpään osoite jokaiseen kutsuun erikseen, jolloin esimerkiksi lokaalista kehitysympäristöstä siirtyminen tuotanto- tai demoympäristöön vaati osoitteen muokkaamisen moneen eri paikkaan. Ohjelmaa pidemmälle kehitettäessä kutsuja tulisi olemaan kymmeniä, jolloin myös osoite olisi tarvinnut vaihtaa kymmeniin eri paikkoihin. Palvelinpään osoitteen hakeminen yhdestä paikasta mahdollisti sen, että osoite tarvitsisi muuttaa vain yhteen paikkaan.

Toinen huomattava muutosehdotus, joka tuli eteen katselmoinnissa, oli väliohjelmiston kirjoittaminen palvelinpäähän tehtäville kutsuille. Prototyypissä palvelinpäähän tehdyt kutsut ja kutsujen vastauksiin varautuminen oli tehty erikseen jokaisessa funktiossa, jossa lähetettiin kutsu. Tämä tarkoitti sitä, että jokaisen kutsun yhteyteen oli kirjoitettu virheisiin varautuminen erikseen, vaikka usein huolimatta siitä, mikä kutsu on kyseessä, virhetilanteissa toimitaan samalla tavalla. Esimerkiksi mikäli käyttäjällä ei ole oikeuksia, näytetään ilmoitus tästä, tai mikäli käyttäjä ei ole kirjautunut, ohjataan käyttäjä kirjautumisivulle ja tyhjennetään sovelluksen muisti eli Reduxin tila. Ohjelmassa 9 on esitetty luotu väliohjelmisto yksinkertaistettuna niin, että palvelinpäähän tehdyn kutsun vastausten käsittely on kommentoitu pois.

```

function callAPIMiddleware({ dispatch, getState }) {
  return next => action => {
    const {
      types,
      callAPI,
      payload = {}
    } = action

    // Tavallinen toiminto, ohjaa eteenpäin
    if (!types) return next(action)

    if ( !Array.isArray(types) || types.length !== 3 ||
      !types.every(type => typeof type === 'string') )
      throw new Error('Expected an array of three string types.')

    if (typeof callAPI !== 'function')
      throw new Error('Expected callAPI to be a function.')

    const [ requestType, successType, failureType ] = types

    dispatch( { ...payload, type: requestType } )

    return callAPI()
      .then(
        response => {
          if(response.status === 401)
            // Tyhjä tila ja ohjaa kirjautumissivulle

          else if(response.status === 403)
            // Ilmoita, ettei ole oikeuksia

          else if(response.ok)
            // Käsittele onnistunut kutsu

          else
            // Käsittele epäonnistunut kutsu
        },
        error => {
          // Käsittele epäonnistunut kutsu
        }
      )
      .then(json => {
        // Käsittele onnistuneen kutsun sisältö
      })
    }
  }
}

```

Ohjelma 9. *Väliohjelmisto palvelinpäähän tehtäville kutsuille*

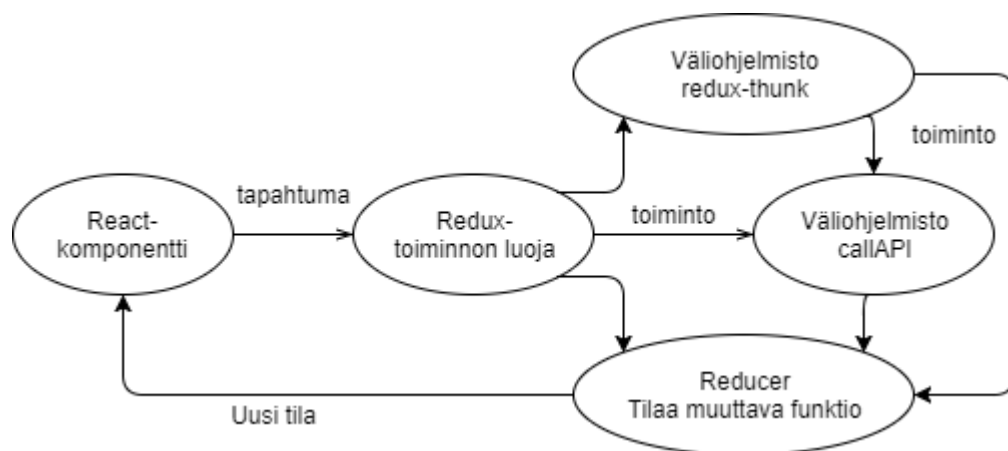
Yllä oleva väliohjelmisto odottaa Redux-toiminto-objektin sisältävän ominaisuudet `types`, `callAPI` ja `payload`. Mikäli toiminto ei sisällä `types`-ominaisuutta, on kyseessä toiminto, jota väliohjelmisto ei käsittele eli toiminto, joka ei lähetä kutsua palvelinpäähän. `Types`-ominaisuus on JavaScript-taulu, joka sisältää kolmen Redux-toiminnon nimet. Näistä ensimmäinen toiminto lähetetään ennen kutsun lähettämistä eli merkitsee Redux-tilaan, että kutsu on kesken. Käyttöliittymässä tämä näytetään yleensä latausikonina. Toi-

nen toiminto lähetetään, mikäli kutsu onnistuu, jolloin merkitään tilaan, että kutsu on valmis ja mahdollisesti tallennetaan tilaan palvelimen lähettämää tietoa. Kolmas toiminto lähetetään, mikäli kutsu epäonnistuu, jolloin tilaan merkitään, että kutsu on suoritettu, mutta se on epäonnistunut, jolloin voidaan näyttää käyttöliittymässä tieto tästä.

Väliohjelmistolla mahdollistetaan lisäksi tieto siitä, onko kutsu kesken tai epäonnistunut, mikä puuttui prototyypistä. Siksi prototyypissä ei pystytty myöskään näyttämään latausikonkia, tai esimerkiksi listan tilalla tekstiä, että tietoa ei pystytty hakemaan palvelimelta. Käytettävyyden kannalta nämä tiedot on hyvä näyttää, ja väliohjelmiston avulla toiminnallisuus saatiin lisättyä tuotteeseen melko vaivattomasti. Toimintoja luoviin funktioihin tuli paljon muutoksia väliohjelmiston lisäämisen jälkeen, ja Reduxin tilaan tuli lisäksi lisätä ominaisuudet, jotka kertovat keskeneräisestä latauksesta tai epäonnistuneesta latauksesta.

Väliohjelmiston käsittelemillä kutsuilla voi olla myös payload-parametri, jolla pystyy välittämään tarvittavia muuttujia väliohjelmistolle tai tilaa muuttaville funktioille. Väliojelmisto esimerkiksi voi tarkistaa, onko toiminnon mukana payload-objektissa funktiota, joka suoritetaan kutsun onnistuttua tai funktiota, joka suoritetaan kutsun epäonnistuttua. Lisäksi objektissa voi olla määriteltynä muuttuja sen varalle, että ei näytetä ilmoitusta onnistumisesta tai epäonnistumisesta, näytetään jokin erityinen onnistumis- tai epäonnistumisviesti yleisen viestin sijaan tai kerrotaan, ettei kutsun vastausta tarvitse käsitellä vastauskoodia pidemmälle eli yrittää lukea JSON-dataa. Näiden muuttujien avulla väliohjelmiston käyttäytyminen pystytään erikoistamaan sopivaksi kaikille ohjelman kutsuille. Muuttujien käsittely on kommentoitu pois ohjelmasta 9 väliohjelmiston toiminnan yksinkertaistamiseksi.

Luodun väliohjelmiston kanssa tapahtumien käsittely sovelluksessa toimii kuvassa 10 esitetyllä tavalla. Riippuen siitä, minkä tyyppinen Redux-toiminto on, se menee Redux-thunk-väliohjelmiston ja/tai itse toteutetun callAPI-väliohjelmiston kautta.



Kuva 11. Toimintojen kulku tuotteessa

Prototyypin Reduxin tilaa muuttavia funktioita muokattiin jonkin verran. Tilaa muuttava funktio palauttaa uuden tilan muokattuna. Prototyypissä uusi tila palautettiin JavaScript-objektina luetellen kaikki kyseisessä tiedostossa määritellyt objektin ominaisuudet. Kaikki ominaisuuksia ei tarvitse kuitenkaan luetella erikseen, jos käytetään JavaScript ES6-versiossa esiteltyä operaattoria, kolmea pistettä, kuten esimerkiksi ohjelmassa 10.

```
const initialState = {
  name: '',
  type: 1,
  parent: null,
  year: new Date().getFullYear()
}

function example(state = initialState, action) {
  switch(action.type) {
    case 'CHANGE_YEAR':
      return {
        ...state,
        year: action.year
      };
    default:
      return state;
  }
}

export default example;
```

Ohjelma 10. Esimerkki reducer-funktiosta, kun käytetään kolmea pistettä

Yllä olevassa tilaa muuttavassa funktiossa tila sisältää neljä eri ominaisuutta. Näitä ei kuitenkaan luetella kaikkia, kun palautetaan uusi tila, vaan voidaan ilmoittaa kolmen pisteen avulla, että objekti pysyy muuten samana, mutta vuosi muuttuu annetuksi vuodeksi. Tällöin myös uusien ominaisuuksien lisääminen säiliöön on helpompaa, kun ei tarvitse merkitä ominaisuutta palautettavaksi jokaiseen toimintovaihtoehtoon.

4.2 Lopullinen tuote

Tuotteen toteutuksessa käytettiin projektinhallinnan apuvälineenä Mantis-tehtävähallintatyökalua, jonne kirjattiin kaikki toteutuksen tehtävät. Aluksi asiakaspään tehtävät olivat melko suuria, kuten esim. yhden osa-alueen toteuttaminen. Myöhemmin, kun osa-alueita oli saatu jo tehtyä ja testattua, tehtävät pienenevät esimerkiksi testauksessa löytyneen virheen korjaamiseksi tai asiakkaan pyytämän korjauksen tekemiseksi.

Prototyypin katselmoinnin avulla oltiin saatu ratkaistua ohjelman korkean tason rakenne ja sovittua tapa toteuttaa näkymiä ja toiminnallisuuksia. Toteutuksen eteenpäin vieminen oli sen jälkeen melko suoraviivaista ja selkeää. Varsinaisia arkkitehtuurillisia ongelmia ei tarvinnut tehdä jatkossa, vaan pystyttiin keskittymään näkymien toteutukseen ja palve-

linpäästä tiedon hakemiseen ja sen näyttämiseen. Ohjelmaa kehitettiin eteenpäin toteuttamalla aina yksi osa-alue kerrallaan siirtyen navigointipalkissa yhdeltä sivualueelta toiselle.

Selkeyttämisen vuoksi luotiin erillinen reducers-funktio jokaiselle sivualueelle. Lisäksi siirrettiin aiemmin actions-kansiossa olleet toimintoja luovat funktiot reducers-kansion vastaaviin tiedostoihin niin, että toiminnon luova funktio on samassa tiedostossa, kuin toiminnon käsittelevä funktio. Tällöin yhdessä tiedostossa sijaitsee aina kaikki samaan Reduxin tilan osa-alueeseen liittyvät funktiot sekä tilan alustaminen ja muokkaaminen.

Ainoa suurempi ongelma tuotteen toteutuksessa oli sovelluksessa esitettävien isojen ja monimutkaisten taulukoiden toteuttaminen. Taulukkoja on monella eri sivulla, ja niillä on erilaisia vaatimuksia. Vaatimukset olivat:

- taulukoissa pitää pystyä esittämään tietoa dynaamisella määrällä sarakkeita ja rivejä
- taulukon leveyden tulee pystyä skaalautumaan ikkunan koon mukaan
- soluissa olevia tietoja pitää pystyä muokkaamaan
- alimmalla rivillä pitää pystyä esittämään taulukon yhteensä-rivi
- riviä napsauttamalla pitää pystyä suorittamaan tietty funktio
- osa tai kaikki soluista pitää pystyä kirjoitussuojaamaan, vaikka joitain soluja tulee pystyä muokkaamaan
- suuria taulukoita pitää pystyä vierittämään pystysuunnassa niin, että sarakeotsikot ja summarivi pysyvät näkyvissä
- rivejä tulee pystyä ryhmittelemään niin, että riveistä näytetään yhteenvetorivi ja rivit saa avattua ja suljettua.

Taulukkojen toteutusta varten etsittiin sopivaa kirjastoa, joka toteuttaisi annetut vaatimukset. Reactille on olemassa paljon kirjastoja, mutta kävi ilmi, että ilmaista kirjastoa, joka toteuttaisi kaikki halutut toiminnallisuudet, ei ole olemassa tai ainakaan sellaista ei löydetty. Käytettäväksi harkittiin ag-Grid-kirjastoa [24], joka tarjoaa maksullisella lisenssillä halutut toiminnallisuudet.

Asiakas ei kuitenkaan ollut kovin halukas maksamaan lisenssiä, ja lisäksi muissa Wapicen projekteissa ag-Gridiä käyttäneet kehittäjät kertoivat, että kirjasto on hieman hankalasti käytettävä Reactin ja Reduxin kanssa. Ag-Grid tarjoaa React-komponentin käytettäväksi Reactin kanssa, ja kirjastossa on otettu joissain kohdissa huomioon, että ohjelmassa voi olla käytössä Redux, jonne tallennetaan kaikki muutokset. Wapicella on kuitenkin muissa projekteissa, joissa on ollut käytössä React ja Redux, jouduttu toteuttamaan useampia asioita kiertoteitse ja melko vaikeasti. Taulukot päädyttiin siten toteuttamaan itse ilman kirjastojen apua.

Ilman kirjastojen apua taulukoiden toteutus vei enemmän aikaa, kuin valmiin kirjaston käyttäminen. Itse toteuttamalla taulukoista saatiin kuitenkin juuri sellaiset kuin haluttiin, ja aikaa ei kulunut dokumentaatioiden tutkimiseen ja kirjaston käytön opetteluun. Usein

myös kirjastoja käytettäessä joitakin asioita tarvitsee toteuttaa kiertotein muokaten kirjaston käyttäytymistä, miltä säästyttiin toteuttamalla taulukot itse.

Kaavioiden kirjastoksi valittiin jo prototyypin tehdessä react-chartjs-kirjasto, jolla saatiin toteutettua muutkin tuotteen kuvaajat, ja pitkien lomakkeiden tekoon Redux-Form. Muita erityisiä ominaisuuksia, joihin olisi voinut harkita kirjaston käyttöä, tuotteeseen ei tarvittu.

Alusta asti oli tiedossa, että ohjelmaan tehdään roolien avulla pääsynhallinta, mutta tarkemmat määrittelyt rooleille saatiin vasta siinä vaiheessa, kun ohjelma oli jo muuten toteutettu. Rooleilla tulisi rajata pääsyoikeus navigaatiopalkin sivuille niin, että navigaatiopalkissa ei näytettäisi sivuja, joille käyttäjällä ei ole oikeutta. Lisäksi käyttäjällä voisi joko olla katselu-oikeus sivulle tai myös muokkausoikeus.

Pääsyoikeuden rajaaminen eri sivuille asiakaspäässä tehtiin niin, että käyttäjän roolista riippuen näytettiin vain osa navigointipalkin elementeistä. Lisäksi, jotta osoitepalkille kirjoittamalla käyttäjä ei pääsisi sivuille, joilla hänellä ei ole oikeutta, lisättiin kunkin sivualueen pääkomponenttiin roolitarkastelu. Pääkomponentissa tarkastellaan komponentin `componentWillMount`-elinkaarifunktiossa, löytyykö käyttäjältä tarvittava rooli, että sivu voidaan näyttää. Mikäli käyttäjältä ei löydy tarvittua roolia, ohjataan käyttäjä roolista riippuen oikealle sivulle.

Palvelinpäässä huolehdittiin lisäksi siitä, että käyttäjällä ei ole oikeuksia tehdä kutsuja, joihin hänellä ei ole oikeutta. Lisäksi joidenkin kutsujen kohdalla, jotka palauttivat suuren määrän tietoa, karsittiin osa tiedosta pois, mikäli käyttäjällä ei ole riittävästi oikeuksia.

Katselu- ja muokkausoikeuden erottaminen toisistaan vaati useampaan komponenttiin roolioikeuksien tarkistuksia. Isoimpana muutoksena oli tehdä kaikista jonkin roolioikeuden tarvitsemista painikkeista oma komponenttinsa. Kyseisen komponentin `componentWillMount`-funktio, joka huolehtii näytetäänkö painike eli onko käyttäjällä muokkausoikeus, on esitetty ohjelmassa 11.

```
componentWillMount() {
  var roles = this.props.roles.map(role => { return role.authority});
  var userHasRole = false;

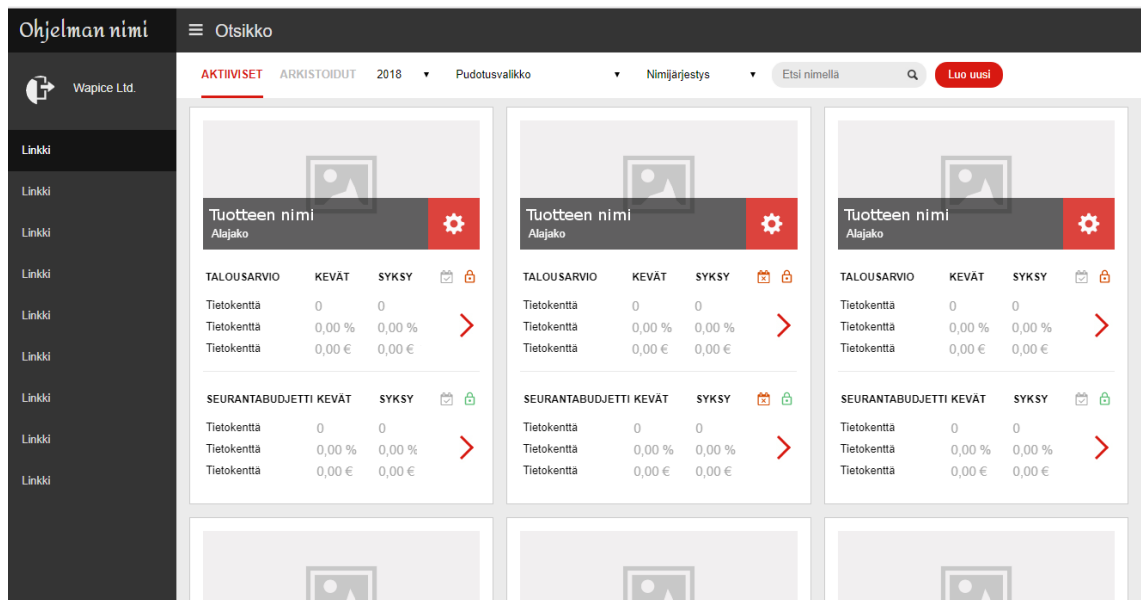
  this.props.roleRequirement.map(required => {
    if(roles.indexOf(required) !== -1)
      userHasRole = true;
  })

  this.setState({ visible: userHasRole });
}
```

Ohjelma 11. Käyttöoikeudet huomioivan painikekomponentin roolitarkastelu

Painikekomponentti saa parametreinaan listan rooleista, joista jokin tulee olla käyttäjällä, jotta käyttäjällä on muokkausoikeus. Komponentti saa myös parametrinaan käyttäjän oikeudet. Ohjelmassa 11 vertaillaan, onko käyttäjällä jokin tarvittavista rooleista, ja mikäli on, muutetaan komponentin tilaan, että painike voidaan näyttää. Palvelinpäässä on tietenkin myös käyttöoikeustarkastelut, jotka palauttavat virheen, mikäli käyttäjä yrittää lähettää kutsua, johon hänellä ei ole oikeutta. Painikkeiden piilottamisella estetään sellaisten kutsujen lähettäminen, jotka aiheuttavat käyttöoikeusvirheen ja siten parannetaan käytettävyyttä. Myös muihin komponentteihin tehtiin käyttöoikeustarkasteluja mm. kenttiin, joihin voidaan syöttää tekstiä.

Asiakaspään ulkoasu uusittiin myös toteutuksen aikana kokonaan. Ulkoasulla haettiin yksinkertaisempaa ja selkeämpää yleisvaikutelmaa kuin aiemmassa käyttöliittymässä niin, että uudessa käyttöliittymässä pystyy silti tekemään kaikki samat toiminnot kuin vanhasakin käyttöliittymässä. Kuva uudesta ulkoasusta on alla olevassa kuvassa 11.



Kuva 12. Uuden asiakaspään ulkoasu

Käyttöliittymän väreiksi valittiin neutraaleja sävyjä ja lisäksi yksi kirkkaampi väri tehosteväri. Vasemmalla oleva navigointipalkki on oletuksena piilossa pienemmillä näytöillä, ja sen saa piilotettua tai tuotua esiin painamalla otsikon vieressä olevaa kolmen viivan ikonia. Tämän avulla myös pienemmällä näytöllä, esimerkiksi tableteilla, pystyy käyttämään suurinta osaa asiakaspään sivuista, koska navigointipalkki ei vie tilaa näytöltä koko aikaa.

Sivuilla, joilla on useampia erillisiä sisältölaatikoita, otetaan näytön koko lisäksi huomioon järjestämällä laatikot yhteen tai useampaan riviin niin, että vaakasuuntainen vierittäminen ei ole pakollista. Esimerkiksi kuvassa 10 on kolme sisältölaatikkoa vierekkäin useammassa rivissä. Pienemmällä näytöllä sisältölaatikkoja on vain kaksi tai yksi yhdellä rivillä riippuen näytön koosta. Niillä sivuilla, joilla on kuitenkin vain yksi sisältölaatikkoa

ja sisältö on esimerkiksi suuri taulukko, ei lähdetty toteuttamaan responsiivista ulkoasua, koska se ei ollut varsinaisena asiakasvaatimuksena.

Verrattuna vanhaan käyttöliittymään (kuva 2, luku 2.1) uudessa käyttöliittymässä näytetään lisäksi vähemmän tietoja budjeteista. Näytettävät tiedot sovittiin asiakkaan kanssa, ja kävi ilmi, että yhtä riviä asiakas ei tarvitse budjetin esikatselutiedoissa. Lisäksi tuotteen kolmas budjetti on piilotettu niin, että siihen pääsee käsiksi painamalla punavalkoista ratasta. Kolmatta budjettia asiakas käyttää huomattavasti harvemmin kuin muita kahta, joten yksinkertaistamiseksi se päätettiin piilottaa.

4.3 Tuotteen testaus

Reactille [25] on olemassa paljon erilaisia testikirjastoja, joiden avulla voidaan kirjoittaa asiakaspään koodille kattavat testit. Ohjelmille kirjoitetaan yleensä yksikkö-, integraatio- ja päästä päähän –testejä. Ohjelman toimintaa laaja-alaisesti pystyy parhaiten testaamaan päästä päähän –testeillä, joihin yleensä valitaan teknologiaksi Selenium. Näillä testeillä testataan ohjelman käyttäjän näkökulmasta ohjelman käyttäytymistä käyttöliittymän kautta.

Reactin yksikkötesteihin [26] käytetään usein Jest- tai Mocha-kirjastoja. Näistä Jest on helposti käyttöön otettava ja mahdollistaa kuvakaappausten avulla testaamisen, mutta Jest on myös uudempi kuin Mocha ja siten vähemmän käytetty. Mocha vaatii enemmän konfigurointia, mutta on myös joustavampi ja jättää asioita kehittäjän päätettäväksi. Mocha on vanhempi testikirjasto, joten se on laajemmalti käytössä, ja siitä löytyy enemmän neuvovia blogitekstejä ja videoita.

Tässä projektissa suunniteltiin käytettäväksi päästä päähän –testeissä Seleniumia ja yksikkötesteissä Mochaa käyttävää Enzyme-kirjastoa. Asiakas ei kuitenkaan halunnut toteutettavan yksikkö- tai muita testejä asiakaspäähän, joten tuotteen testaaminen tehtiin manuaalisesti. Palvelinpäähän tehtyyn REST-rajapintaan sen sijaan oli työmääräarviossa tilaa kirjoittaa yksikkötestit. Asiakaspään toiminnallisuuksista testattiin yksinkertaisimmat käyttötapaukset osana toteuttamista, ja lisäksi käytettiin testausdokumenttia eri käyttötapauksista.

Testaamista varten Wapicella oli olemassa jo valmiiksi erillinen testipalvelin, jolla pystyttiin ajamaan uutta asiakaspäätä ja paranneltua palvelinpäätä. Ympäristö oli käytännössä samanlainen kuin lopullinen tuotantoympäristö tulisi olemaan, vain tietokannan sisällössä oli eroavaisuuksia. Testipalvelimella pystyttiin siten varmistumaan tuotteen toimivuudesta myös tuotantoympäristössä.

Tuotteen testaamisesta tehtiin projektin alkuvaiheilla testaussuunnitelma, johon kerättiin erilaisia testitapauksia, jotka kattoivat kaikki testattavat ominaisuudet. Dokumentti jaoteltiin navigaattiorakenteen perusteella eri osa-alueisiin, ja testaus suoritettiin aluekohtaisesti

aina alueen valmistuttua. Testaajana toimi eri henkilö kuin asiakaspään toteuttaja, jotta pystyttiin samalla arvioimaan asiakaspään käytettävyyttä.

Aluksi kaikki testitapaukset olivat tilassa epäonnistunut, ja tuotteen toteutuksen ja testausten edetessä pystyttiin seuraamaan testidokumentin etusivulta, kuinka suuri osuus testeistä milläkin hetkellä on hyväksyttyjä eli kuinka suuri osuus tuotteesta on valmis. Testidokumentti toimi siis myös toteutuksen edistymisen seuraamisen apuvälineenä.

Testitapauksiin kirjattiin testitapauksen kuvauksen lisäksi odotettu lopputulos, varsinainen lopputulos, testin tila, testaaja ja päivämäärä. Päivämäärän avulla pystyttiin selvittämään helpommin, missä vaiheessa jokin toiminnallisuus on mennyt rikki tilanteissa, joissa testi on aiemmin onnistunut mutta myöhemmin epäonnistuu. Testaajan nimen kirjaaminen helpotti lisätietojen saamista epäselvissä tilanteissa, kun testi on merkitty epäonnistuneeksi.

Testit suoritettiin vähintään kerran osa-alueen valmistuttua ja lisäksi uudestaan, mikäli osa-alueen testistä jokin epäonnistui ja osa-aluetta jouduttiin korjaamaan. Lisäksi testit suoritettiin vielä kertaalleen tuotteen valmistuttua tuotantotietokannan tiedoilla, jotta varmistuttiin toiminnan oikeellisuudesta. Tietokannan vaihdolla oli yllättävän paljon vaikutusta näkymiin ja testien onnistumiseen, koska testipalvelimen tietokanta sisälsi huomattavasti vähemmän tietoa. Testitietokannalla saatiin testattua toiminnallisuudet hyvin, mutta tiedon esittämistavassa ja laskentakaavoissa löytyi vielä virheitä tietokannan vaihdon jälkeen, kun nähtiin, millaista tietoa tuotteessa oikeasti käytetään.

Testeissä käytettiin sopimuksen mukaisesti Safari- ja Google Chrome -selaimia. Google Chrome-selainta käytettiin Windows-käyttöjärjestelmällä ja Safari-selainta macOS-käyttöjärjestelmällä, kuten myös loppukäyttäjät tulevat tuotetta käyttämään. Pääsääntöisesti testit ajettiin Google Chrome -selaimella, ja vasta lopuksi Safari-selaimella.

Lopullinen hyväksymistestaus oli asiakkaan vastuulla. Asiakkaalle toimitettiin tunnukset testipalvelimelle, ja testipalvelimelle otettiin kopio tuotantopalvelimen tietokannasta, jotta vanhaa asiakaspäätä ja uutta asiakaspäätä pystyttäisiin vertailemaan. Asiakas suoritti hyväksymistestejä syksystä 2017 helmikuuhun 2018 asti löytäen tuotteesta jonkin verran korjattavaa.

Suurin osa asiakkaan pyytämistä korjauksista olivat virheellisiä toiminnallisuuksia, mutta osa korjauspyynnöistä oli pikemminkin jatkokehityspyynnöjä. Näiden erottelu oli onneksi selkeää, koska vaatimuksena oli toteuttaa vanhassa käyttöliittymässä olleet asiat uuteen käyttöliittymään. Uudet asiat, joita ei löytynyt vanhastakaan käyttöliittymästä, luokiteltiin jatkokehityspyynnöiksi. Löydetyt virheet olivat usein sen tyylisiä, että piti olla tietyt asiat tehtynä tietyssä järjestyksessä, että virhe esiintyi, minkä takia manuaalisella testauksella näitä virheitä ei oltu löydetty. Automaatti- ja yksikkötesteillä useimmat asiakkaan löytämät virheet olisi sen sijaan voinut huomata.

4.4 Tuotteen jatkokehitys

Tuotetta lähdettiin jatkokehittämään jo syksyllä 2017, vaikka tuotteen hyväksymistestaus oli vielä kesken. Suurimpana ja kriittisimpänä tehtävänä oli ottaa tuotteessa käyttöön toinen integraation tarjoava palvelu, koska asiakas aikoi siirtyä käyttämään eri palveluntarjoajaa toiminnallisuudessa, jolla asiakas saa automaattisesti toisesta palvelusta tietoja, joita käytetään budjetoinnissa esim. kappalemääriä sekä päivämääriä, jolloin tuotteita on myyty. Toiminto tuli toteuttaa niin, että molempia integraatiopalveluita pystyisi käyttämään samaan aikaan käyttöliittymässä.

Uuden integraatiopalvelun käyttöönotossa oli joitain ongelmia siinä, että millaisessa muodossa integraatiosovellus tarjoaa tietoa. Wapicella oli kuitenkin mahdollisuus vaikuttaa integraatiopalvelun toimintaan, koska olimme ensimmäisiä palvelun käyttöönottajia. Palveluntarjoajan kanssa käytiin jonkin verran keskustelua siitä, miten palvelun on luvattu toimivan, minkä lopputulemana integraatiopalvelu saatiin melko pienellä työmäärällä otettua käyttöön. Kun asiakas oli testannut sen toimivuuden, poistettiin vanhaan integraatiopalveluun liittyvät käyttöliittymäkomponentit asiakaspäästä.

Syksyn ja talven aikana tehtiin lisäksi jonkin verran pienempiä muutospyyntöjä asiakaspäähän. Suurimpana muutoksena oli tarve muuttaa roolien oikeuksia sekä lisätä ylimääräinen rooli hallinnointia varten. Roolimuutoksia varten tarvitsi tehdä muutoksia sekä palvelin- että asiakaspäähän.

Lisäksi asiakaspäähän lisättiin erilaisia kenttiä budjetteihin. Kenttien laskukaavat olivat sen verran yksinkertaisia, että laskut päätettiin toteuttaa asiakaspäässä. Tiedonkulussa asiakkaan kanssa oli kuitenkin hieman ongelmia liittyen laskukaavoihin, minkä takia kenttiä ei aluksi saatu toimimaan siten kuin asiakas olisi halunnut. Ongelma saatiin kuitenkin esimerkkien kautta ratkaistua.

Asiakas on lisäksi ilmaissut kiinnostuksesta lisätä sellainen toiminnallisuus tuotteeseen, joka vaatisi muutoksia tietokantatasolle asti. Tästä keskustellaan asiakkaan kanssa vielä myöhemmin lisää. Wapice on lisäksi antanut työmääräarvion sille, että asiakkaan hallintakäyttäjät pystyisivät hallinnoimaan sovelluksen käyttäjiä, sillä tällä hetkellä Wapicen kehittäjät tekevät uusien käyttäjien lisäämisen, käyttäjien poiston ja käyttöoikeuksien muutoksen.

Wapicen kehitysehdotuksena on lisäksi muuttaa palvelintason arkkitehtuuria niin, että tietokanta ja sovelluspalvelin yhdistettäisiin. Tätä harkitaan siksi, että tällä hetkellä palvelinten uudelleenkäynnistyessä saattaa sovelluspalvelin käynnistyä ennen tietokantapalvelinta, minkä takia palvelinpää ei onnistu ottamaan yhteyttä tietokantaan käynnistymisen yhteydessä. Virheestä tarvitsisi joko pystyä ilmoittamaan Wapicen sisäiselle tekniselle tuelle tai sitten pyrkiä estämään virheen syntyminen, mikä onnistuisi palvelinten yhdistämisellä.

Asiakaspään responsiivisuuden parantaminen niin, että kaikkia sivua pystyttäisiin käyttämään myös vähintään tabletilla, voisi olla yksi kehitysmahdollisuus. Tällä hetkellä moni sivuista toimii myös mobiililaitteella, mutta suurien taulukoiden tai kuvaajien kanssa tämän toteuttaminen ei ollut mahdollista. Paremman responsiivisuuden toteuttamiseen tarvittaisiin tarkemmat määrittelyt, ja lisäksi tulisi esimerkiksi päättää, voiko joitain taulukon sarakkeita piilottaa, kun ohjelmaa käytetään pienellä näytöllä.

Tuote on ollut aikaisemmin käytössä muillakin asiakkailta kuin nykyisellä asiakkaalla, joten myös tulevaisuudessa tuotteen saattaa ottaa käyttöön yksi tai useampi muu asiakas. Näillä asiakkailta saattaa olla paljonkin erilaisia kehitystoiveita sovellukselle johtuen siitä, että kahdella asiakkaalla on harvoin tarkalleen samat tarpeet. Tähän liittyen tuotetta saatetaan jatkokehittää hyvin paljon tavoilla, joita tällä hetkellä ei pystytä ennakoimaan.

5. TYÖN ARVIOINTI

Projekti oli tyypiltään melko normaali päivittämisprojekti. Projektiryhmä koostui projektipäälliköstä ja kahdesta kehittäjästä, joista toinen teki tässä työssä käsitellyn asiakaspään. Asiakkaan vaatimukset saatiin täytettyä ja asiakas on vaikuttanut tyytyväiseltä uudistettuun asiakaspäähän. Projektissa tehdyt teknologiavalinnat ovat osoittautuneet työtä tehdessä hyviksi ratkaisuiksi.

5.1 Vaatimusten täytyminen

Projektin tärkeimpänä vaatimuksena oli toteuttaa vanhassa käyttöliittymässä olevat toiminnallisuudet uuteen käyttöliittymään. Vaatimus täyttyi melko hyvin, joskin asiakas löysi vielä hyväksymistestauksessa joitakin virheitä, jotka tuli vielä korjata. Kattavammilla testeillä olisi saatu varmistuttua paremmin toiminnallisuuksien toimivuudesta, mutta asiakkaan päätöksestä testit jätettiin kirjoittamatta. Testien kirjoittaminen olisi ollut kuitenkin huomattava parannus ja tae tuotteen toimivuudesta.

Asiakkaan ei-toiminnallinen vaatimus toteuttaa uusi asiakaspää modernilla teknologialla täyttyi myös valinnalla käyttää React-ohjelmistokehystä toteutuksessa. Käyttöliittymän ulkoasu uudistui sekä väreiltään ja tyyliältään että responsiivisuudeltaan. Uusi asiakaspää on myös osaksi käytettävissä mobiililaitteilla, vaikka kaikkien näyttöjen suuria taulukoita ei pystytäkään esittämään mobiililaitteiden kokoisilla näytöillä.

Kaikista tallentamattomista muutoksista varoitetaan käyttäjää ennen kuin käyttäjä siirtyy pois sivulta, ja käyttäjällä on mahdollisuus valita jääkö sivulle tallentamaan muutokset vai poistuuko sivulta menettäen muutokset. Lisäksi kaikista luonneista, muokkauksista ja poistoista tulee ilmoitus onnistumisesta tai epäonnistumisesta. Toiminnon epäonnistuessa tietoja ei myöskään hukata, vaan toimintoa pystytään yrittämään uudelleen. Myös käytettävyyden puolesta asiakkaan antamat vaatimukset täyttyivät.

Tallentamattomista muutoksista pidetään sovelluksessa kirjaa yhdessä muuttujassa muutamaa poikkeusta lukuun ottamatta, joten tallentamien muutosten tarkkailu toteutettiin sopivan yksinkertaisesti ja selkeästi. Ilmoitusten näyttämiseen käytettiin ulkopuolista kirjastoa, jonka avulla säästettiin aikaa. Kirjasto myös tarjoaa kaikki tarvittavat ilmoitustyytit ja lisäksi varmistusilmoituksen, jolla voidaan varmistaa toiminto, jota ei voi peruuttaa. Kirjaston valinta onnistui siis hyvin.

Wapicen asettama kansainvälistämisen vaatimus toteutettiin ulkoisella kirjastolla, joka tarjosi sekä tekstien että numeroiden lokalisoinnin. Toteutuksen aikana todettiin, että lokalisointi on hyvä olla olemassa etenkin numeroiden kanssa jo pelkästään sen takia, että

suuret luvut pystytään esittämään luettavassa muodossa. Lisäksi sanamuotojen vaihtaminen on helpompaa, kun muutokset voi tehdä yhteen tiedostoon.

Projektissa pysyttiin hyvin tehdyn työmääräarvion puitteissa varsinkin, kun työmääräarviota tehdessä vanhaan asiakaspäähän ei oltu pystytty tutustumaan läpikotaisin. Joihinkin osa-alueisiin kulutettiin suunniteltua enemmän työtunteja, mutta vastaavasti jotkin asiat saatiin tehtyä nopeammin, ja pystyttiin hyödyntämään aiemmin tehtyjä komponentteja ja funktioita myöhemmin toteutetuissa osa-alueissa.

5.2 Tehdyt teknologiapäätökset

Tärkeimpänä teknologiapäätöksenä projektissa oli valita asiakaspään teknologia. Teknologiaksi harkittiin kahta eri JavaScript-ohjelmistokehystä, Reactia ja Angularia. Näiden välillä tehtiin päätös toteuttamalla prototyyppi kummallakin teknologialla. Prototyyppien teon jälkeen päädyttiin käyttämään Reactia.

Päätös olisi ollut luultavasti erilainen, mikäli Reactilla tehdyssä prototyyppissä ei oltaisi käytetty Redux-kirjastoa, sillä ohjelmassa oli keskeisessä asemassa palvelinpäästä haetun tiedon säilöminen. Reduxin käyttöönottamisesta voi kiittää Wapicen aikaisempaan kokemuksesta Reactista.

Angular on kehittynyt paljon syksyn 2016 jälkeen, ja siitä on julkaistu jo useampi uusi versio [27]. Uudet Angularin versiot [28] eivät kuitenkaan eroa yhtä paljon toisistaan kuin AngularJS eroaa Angularista, vaan kaikista Angularin versioista kahdesta ylöspäin puhutaan nimellä Angular ja versiosta yksi AngularJS. Uudet Angularin versiot parantavat versiota kaksi, ja tuovat siihen lisäominaisuuksia. Versiosta ylöspäin siirtyminen on myös melko helppoa.

Teknologiavalinnan jälkeen, kun prototyypit olivat valmiita, projekti oli toteutuksen kannalta tauolla yli puoli vuotta. Tauko olisi voinut aiheuttaa teknologiapäätöksen vanhentumisen, etenkin Angularin kehityksen ansiosta, tai kenties uusi ohjelmistokehys, joka olisi sopinut projektiin paremmin, oltaisiin voitu julkaista. Näyttää [29] kuitenkin siltä, että edelleen vuonna 2018 React on suosituin JavaScript-ohjelmistokehys. Vue.js on uusi melko suosittu tulokas, mutta se ei ole saanut yhtä paljon suosiota, kuin React ja edelleen suosiossa oleva AngularJS. Angular-ohjelmistokehittäjät eivät ole lisäksi olleet kovin tyytyväisiä ohjelmistokehukseen, sillä käyttäjätyytyväisyys on noin 49 %, kun sama luku React-ohjelmistokehittäjien keskuudessa on 93 %.

Toinen suurempi teknologiapäätös oli valita sopiva kirjasto taulukoiden toteuttamiseen. Tähän harkittiin ag-Grid-kirjastoa, mutta siitä oli Wapicella huonoa kokemusta, joten päädyttiin toteuttamaan itse tarvittavat komponentit taulukkojen toteuttamiseen. Kehittäjä on nyttemmin saanut myös omakohtaista kokemusta siitä, että ag-Gridin käyttö voi

olla hankalaa Reduxin kanssa ja tilanteissa, joissa kirjasto ei tarjoa tarkalleen oikeanlaista käyttäytymistä. Ag-Gridin käyttämättä jättäminen tuntuu siis olleen hyvä päätös.

Itse toteutetut taulukkokomponentit olisi sen sijaan voinut toteuttaa paremminkin. Taulukkokomponenteista ei lähdetty projektissa tekemään yleiskäyttöisiä komponentteja, vaan jokaiselle eriävällä näytölle tehtiin muokattu taulukkokomponentti alakomponentteineen. Yleiskäyttöinen taulukkokomponentti olisi voinut olla selkeämpi kuin useampi vain hieman toisistaan eroava komponentti, ja testauksessa olisi pystytty varmistumaan komponenttien toimivuudesta paremmin.

6. YHTEENVETO

Projektin tavoitteena oli toteuttaa olemassa olevaan budjetointisovellukseen uusi asiakaspää modernilla teknologialla ja päivittää ohjelman käyttöliittymä. Vanhan asiakaspään teknologiana oli käytetty Adobe Flex –ohjelmistokehystä, joka hyödyntää Adobe Flash Playeriä. Nykyään Adobe Flash Playerin tuki selaimissa on rajallisempi löytyneiden tietoturva-aukkojen takia, ja sen tuki ollaan lakkauttamassa lähivuosina, minkä takia asiakas halusi uudistaa ohjelman asiakaspään.

Asiakas asetti ohjelmalle vaatimukseksi, että sillä tulee pystyä tekemään samat asiat, kuin vanhalla asiakaspäällä. Teknologian tulisi olla moderni, jotta asiakaspäätä ei tarvitsisi uusia lähivuosina uudestaan. Asiakas myös halusi, että käyttöliittymän ulkoasu uudistetaan samalla.

Projekti aloitettiin kesällä 2016 toteuttamalla prototyyppi uudella JavaScript-ohjelmistokehyksellä nimeltä Angular 2. Ohjelmistokehys oli aivan uusi, ja haluttiin selvittää sen soveltuvuutta tuotantokäyttöön uutena teknologiana. Kesällä toteutettiin myös toinen prototyyppi React-ohjelmistokehyksellä. Kahden prototyypin totutuksen jälkeen pystyttiin vertailemaan, kumpi olisi parempi teknologiavalinta budjetointisovelluksen uudeksi asiakaspään teknologiaksi.

Prototyyppien toteutuksen ja toteutusten vertailun jälkeen päädyttiin käyttämään Reactia uuden asiakaspään teknologiana. Angular 2 osoittautui vielä keskeneräiseksi, ja Angular onkin kehittynyt todella paljon kesän 2016 jälkeen. React on vanhempana ohjelmistokehyksenä valmiimpi ja sekä Wapicen projekteissa, että suurien sivustojen teknologiana hyväksi ja toimivaksi todettu ohjelmistokehys. React-prototyyppissä oli lisäksi käytössä Redux-kirjasto, jonka avulla tiedon säilöminen, välittäminen ja muuttaminen ovat suoraviivaisempaa.

React-prototyyppiä käytettiin myynnin apuna syksyn ja talven aikana, ja keväällä jatkettiin tuotteen toteuttamista prototyypin pohjalta. Koodikatselmoinnin avulla prototyypin koodirakenne saatiin muokattua paremmaksi, ja tuotteen pidemmälle työstäminen helpotui. Asiakaspään toteutus sujui koodikatselmoinnin jälkeen melko ongelmattomasti, mutta erilaisten taulukkojen toteutusta varten ei löydetty sopivaa kirjastoa. Taulukot päädyttiin lopulta toteuttamaan alusta asti itse, mihin meni enemmän aikaa kuin valmiin kirjaston käyttämisessä olisi mennyt.

Asiakaspään ulkoasu päivitettiin nykyaikaisemmaksi, yksinkertaisemmaksi ja responsiivisemmaksi. Kaikkia sivuja ei kuitenkaan pystytty toteuttamaan mobiililaitteille sopiviksi taulukoiden hyvin suurten kokojen takia.

Tuotetta testattiin ensin Wapicen toimesta manuaalisesti, ja sittemmin asiakas on suorittanut hyväksymistestejä. Asiakas on lisäksi pyytänyt jo jonkin verran uusia ominaisuuksia toteutettavaksi uuteen asiakaspäähän. Asiakas vaikuttaa tyytyväiseltä uuden asiakaspään ulkoasuun ja toteutukseen, ja asiakkaan asettamat vaatimukset täyttyivät.

LÄHTEET

- [1] Adobe Flex White Paper, Adobe, web page. Available (accessed 1.11.2017): <http://www.adobe.com/devnet/flex/whitepapers/roadmap.html>.
- [2] BlazeDS Overview, Sourceforge, web page. Available (accessed 2.11.2017): <https://sourceforge.net/adobe/blazeds/wiki/Overview/>.
- [3] V. Woollaston Google and Mozilla pull the plug on Adobe Flash, Mail Online, web page. Available (accessed 2.11.2017): <http://www.dailymail.co.uk/sciencetech/article-3160644/Google-Mozilla-pull-plug-Adobe-Flash-Tech-giants-disable-program-browsers-following-critical-security-flaw.html>.
- [4] D. Hardawar Adobe patches that horrible Flash vulnerability, Engadget, web page. Available (accessed 2.11.2017): <https://www.engadget.com/2015/10/16/adobe-flash-patch/>.
- [5] M. Locklear Latest Adobe Flash vulnerability allowed hackers to plant malware, Engadget, web page. Available (accessed 2.11.2017): <https://www.engadget.com/2017/10/16/adobe-flash-vulnerability-hackers-plant-malware/>.
- [6] T. Warren Adobe will finally kill Flash in 2020, The Verge, web page. Available (accessed 2.11.2017): <https://www.theverge.com/2017/7/25/16026236/adobe-flash-end-of-support-2020>.
- [7] Software release life cycle, Wikipedia, web page. Available (accessed 7.12.2017): https://en.wikipedia.org/w/index.php?title=Software_release_life_cycle&oldid=812315719.
- [8] A. Freeman, Pro Angular, Second Edition ed. Apress, 2017, 788 p.
- [9] L. Eidnes AngularJS: The Bad Parts, web page. Available (accessed 27.2.2018): <https://larseidnes.com/2014/11/05/angularjs-the-bad-parts/>.
- [10] N. Kaufman, T. Templier, Angular 2 Components, Packt Publishing, 2016, 107 p. Available: <https://ebookcentral.proquest.com/lib/tut/detail.action?docID=4755339>.
- [11] Angular Tutorial, Google, web page. Available (accessed 15.12.2017): <https://angular.io/tutorial>.
- [12] Angular CLI, web page. Available (accessed 15.12.2017): <https://github.com/angular/angular-cli/releases?after=v1.0.0-beta.11-webpack.6>.
- [13] S. Motazavi What is TypeScript and Why Should You Use it? web page. Available (accessed 27.2.2018): <https://www.wakefly.com/blog/what-is-typescript-and-why-should-you-use-it/>.

- [14] Strong versus weak typing, web page. Available (accessed 15.3.2018): <http://www.cs.cornell.edu/courses/cs1130/2012sp/1130selfpaced/module1/module1part4/strongtyping.html>.
- [15] C. Gackenheimer, Introduction to React, 1st ed. Apress, Berkeley, CA, 2015, .
- [16] React.Component, web page. Available (accessed 27.2.2018): <https://reactjs.org/docs/react-component.html>.
- [17] B. Krajka The difference between Virtual DOM and DOM, web page. Available (accessed 27.2.2018): <http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/>.
- [18] B. Dinkevich, I. Gelman, The Complete Redux Book, Leanpub, 2015, 179 p.
- [19] Redux Form, web page. Available (accessed 29.12.2017): <https://github.com/erikras/redux-form>.
- [20] React-redux-toastr, web page. Available (accessed 29.12.2017): <https://github.com/diegoddox/react-redux-toastr>.
- [21] React-redux-i18n, web page. Available (accessed 29.12.2017): <https://github.com/artisavotins/react-redux-i18n>.
- [22] Angular releases, web page. Available (accessed 6.1.2018): <https://github.com/angular/angular/releases>.
- [23] W. Mociun Big names using React.js, web page. Available (accessed 27.2.2018): <http://reactkungfu.com/2015/07/big-names-using-react-js/>.
- [24] Ag-Grid, web page. Available (accessed 25.2.2018): <https://www.ag-grid.com/>.
- [25] T. Connelly Good Practices for Testing React Apps, web page. Available (accessed 3.3.2018): <https://medium.com/@TuckerConnelly/good-practices-for-testing-react-apps-3a64154fa3b1>.
- [26] M. Nedrich Jest vs. Mocha for React.js Testing – Strengths & Weaknesses, web page. Available (accessed 3.3.2018): <https://spin.atomicobject.com/2017/05/02/react-testing-jest-vs-mocha/>.
- [27] What is the difference between angular 2 , 4 and 5? web page. Available (accessed 4.3.2018): <http://www.codekul.com/blog/difference-between-angular-2-4-5/>.
- [28] N. Gupta What is the difference between Angular 2 and Angular 4? web page. Available (accessed 4.3.2018): <https://www.quora.com/What-is-the-difference-between-Angular-2-and-Angular-4>.
- [29] E. Elliott Top JavaScript Libraries & Tech to Learn in 2018, web page. Available (accessed 4.3.2018): <https://medium.com/javascript-scene/top-javascript-libraries-tech-to-learn-in-2018-c38028e028e6>.