jamk.fi

Kuntotestisovelluksen kehittäminen

Case: Trainer4You

Lauri Saarela

Opinnäytetyö
Marraskuu 2016
Luonnontieteiden ala
Tradenomi (AMK), tietojenkäsittelyn tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Saarela, Lauri	Opinnäytetyö, AMK	Marraskuu 2016
	Sivumäärä 39	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Kuntotestisovelluksen kehittäminen
Case: Trainer4You

Tutkinto-ohjelma
Tietojenkäsittelyn tutkinto-ohjelma

Työn ohjaaja(t)
Tommi Tuikka

Toimeksiantaja(t)
Trainer4You Oy

Tiivistelmä

Personal trainer -palvelut ja koulutukset ovat olleet viime vuosina kovassa kysynnässä. Opinnäytetyön toimeksiantaja Trainer4You Oy on Suomessa toimiva personal training -palveluita tarjoava yritys. Yrityksen palveluihin sisältyy mm. personal trainer-, kuntosalivalmentaja- sekä ravintovalmentajakoulutukset.

Opinnäytetyön tavoitteena oli suunnitella ja kehittää toimeksiantajan määrityksiin perustuva web-sovellus, minkä pääasialliset toiminnot olisivat kuntotestien mittaukset ja raportointi. Sovelluksen olisi tarkoitus palvella personal trainer -valmentajia heidän tekemässään työssään. Sen tulisi olla yksinkertainen järjestelmä, jonka avulla valmentajat voisivat lisätä itselleen asiakkaita, merkitä heille kuntotestien tuloksia sekä koostaa tuloksista kuntotestiraportteja. Sovelluksen tulisi toimia tyypillisimmillä selaimilla niin työpöytäkäytössä kuin tablettitietokoneillakin.

Opinnäytetyö toteutettiin sovelluksen kehittämistutkimuksena. Tutkimuksen viitekehyksenä toimi modernit web-teknologiat ja niihin liittyvät sovelluskehykset. Tutkimuksen aikana sovellukselle selvitettiin sopivat toteutusvälineet, joita hyödyntäen se myös lopulta kehitettiin. Sovelluksen toiminallisuus oli mietitty jo hyvin pitkälle kehitystyön alussa, joten valtaosa kehittämistyöstä liittyi sen tekniseen toteutukseen. Teoriaosuus käsittelee valittujen tekniikoiden taustaa ja toteutusosio kuvaa sovelluksen pääominaisuuksien kehitystä.

Projektin tuloksena syntyi toimiva ja alkumäärityksiin hyvin pitkälti perustuva sovellus, joka on tätä nykyä myös valmentajien hyödynnettävissä. Sovelluksen jatkokehityksestä ei ole varmaa tietoa, mutta se ei ole myöskään poissuljettu asia.

Avainsanat (asiasanat)

web-sovellus, web-kehitys, JavaScript, Meteor

Muut tiedot

jamk.fi

Description

Author(s)	Type of publication	Date
Saarela, Lauri	Bachelor's thesis	November 2016
		Language of publication: Finnish
	Number of pages: 39	Permission for web publication: x

Title of publication

Development of fitness test application

Case: Trainer4You

Degree programme

Business Information Systems

Supervisor(s)

Tuikka, Tommi

Assigned by

Trainer4You Oy

Abstract

There has been a high demand for personal trainer services in the recent years. Trainer4You Ltd is a personal training services provider in Finland. The company also offers educational services nationwide involving health, fitness, and wellness courses.

The objective of the thesis was to design and develop a web based application for the company and their trainers. The idea behind the application was to be a helpful tool for the trainers in their everyday work. The main requirements and features of the application were that it could be used to organize customers and their fitness test results, and to generate reports based on these results. The application was supposed to be used in most common web browsers in desktop and tablet environments.

The thesis was carried out as an action research. The focus was on modern web technologies and frameworks. The research consisted of the search for suitable tools to implement the application, and then the actual development phase. The functionality of the application was already largely thought of at the start of the project, so the major part of the development focused on executing its technical aspects. The theoretical framework of the research explains the background of the technologies and tools used in the project. The implementation part shows how the main features were developed.

The project resulted in a functional web-application that met the objectives and requirements that were specified for it. Further development plans for the application are not known as of now but they are not out of the question.

Keywords/tags (subjects)

web-application, web-development, JavaScript, Meteor

Miscellaneous

Sisältö

Kuviot

Käsitteet

JavaScript	Pääasiassa Web-ympäristössä käytettävä ohjelmointikieli.
Asynkroninen ohjelmointi	Ohjelman erillisiä toimintoja voidaan suorittaa rinnakkain eli yhtä aikaa.
Callback-mekanismi	Callback on funktio, joka annetaan parametrina toiselle funktiolle, jonka sisällä se myös suoritetaan.
I/O	Input/output. Tiedon siirräntämenetelmä tietokoneissa ja ohjelmissa.
Apache Cordova	Alusta mobiilisovellusten kehitykseen.
React Native	Sovelluskehys natiivisovellusten kehitykseen Reactilla.
JSON	JavaScript Object Notation. Tiedostomuoto tiedonvälitykseen ja varastointiin. Käyttää JavaScriptia muistuttavaa syntaksia.
DOM / HTML DOM	Document Object Model. Web-sivujen puumainen rakennemalli, jota voidaan manipuloida esimerkiksi JavaScriptin avulla.
WebSocket	Asiakkaan (yleensä selaimen) ja palvelimen välinen kommunikaatioprotokolla.
API	Application Programming Interface. Ohjelmointirajapinta, eli määritelmä miten eri ohjelmat voivat keskustella keskenään.
HTTP	Hyper Text Transfer Protocol**.** Selainten ja www-palvelimien käyttämä tiedonsiirtoprotokolla.
REST	Representational State Transfer. HTTP-protokollaan perustuvan tiedonvälityksen ohjelmointirajapinta.
CSV	Comma Separated Values. Yksinkertaisen taulukkomuotoisen tiedon tallennusmuoto.
HTML	Hypertext Markup Language. Web-sivujen määrittelemiseen käytetty merkintäkieli.

CSS	Cascading Style Sheets. Tyylikieli web-sivujen ulkoasulliseen tyylittelyyn.
Bootstrap	CSS-ulkoasukirjasto.
Base64 enkoodaus	Binääridatan muunnos tekstimuotoon. Esittää binääridataa ASCII merkkijonona.
MEAN-pino	MongoDB, Express, Angular & Node. Kokoelma avoimen lähdekoodin ohjelmia dynaamisten web-sivujen ja web-sovellusten kehittämiseen.

1 Johdanto

Nykypäivänä sekä varmasti myös tulevaisuudessa on yhä enemmän kysyntää personal trainer -palveluille ja koulutuksille, niin Suomessa kuin ulkomaillakin. Suomessa tähän kysyntään vastaavat alalla toimivat yritykset ja valmentajat, jotka tarjoavat laaja-alaisesti koulutuksia ja palveluita näihin tarpeisiin. Niin myös tämän opinnäytetyön toimeksiantaja, Trainer4You Oy, joka tarjoaa koulutuksia ja asiantuntijapalveluita personal trainer-, ravintovalmentaja- sekä kuntosalivalmentajasektoreilla.

Toimiessani harjoittelussa kyseisessä yrityksessä toimeksiantajan edustaja ehdotti, että tutkisin heidän ideoimansa kuntotestisovelluksen kehitysmahdollisuutta. Ideoitu sovellus liittyisi asiakkaille tehtyjen kuntotestien raportointiin. Se myös varastoisi asiakkaisiin ja heidän testituloksiin liittyviä tietoja, joita sekä valmentajat että yritys voisivat hyödyntää myös muilla tavoin toiminnassaan. Sovellusta ei oltu toistaiseksi voitu toteuttaa ajan ja resurssien puutteiden vuoksi. Kehitystyö oli itselleni oiva mahdollisuus oppia jotain uutta, mihin myös toimeksiantajan edustaja kannusti.

2 Tutkimusasetelma

2.1 Tavoitteet ja rajaukset

Opinnäytetyön tavoitteena oli kehittää toimeksiantajayrityksen jo ennalta määrittelemän kuntotestisovellusidean pohjalta toimiva web-sovellus. Toimeksiantajalla oli melko tarkka kuvaus sovelluksesta, mistä kävi ilmi sen päätoiminnot ja ominaisuudet. Kehittäjänä minun tulisi tutkia ja valita sopivat teknologiaratkaisut, joilla sovellus voitaisiin kehittää.

Web-sovelluksien toteutustapoja on nykypäivänä valtama määrä. Oikeiden teknologiaratkaisujen valinta voi olla haastavaa. Toimeksiantajan hyväksynnällä päätin lähestyä asiaa modernin web-kehityksen tekniikoita tutkimalla, sillä niillä saataisiin luultavimmin sovelluksenomaisin lopputulos aikaiseksi. Sovellus nähtiin hyvänä lisänä toiminnalle, mutta ei välttämättömänä. Näin ollen kustannusten

kannalta oli selvää, että sovelluksen kehittämiseen käytettäisiin avoimen lähdekoodin alustoja ja osia.

2.2 Tutkimuskysymykset

Pystyin erottelemaan kehitystyöprosessin tärkeimmät seikat sovelluksen kuvauksesta. Nämä kysymykset tulisi ottaa huomioon toteutusvälineiden valintoja tehdessä.

1. Mitkä ovat toimeksiantajan ideoiman sovelluksen vaatimukset?
    - Mitkä ovat tärkeimmät ominaisuudet?
    - Miten sovelluksen tulisi toimia?
2. Mitkä toteutusvälineet sopivat parhaiten sovelluksen vaatimuksiin?
    - Millä välineillä voidaan valmistaa nopeasti prototyyppi demoa varten?
    - Millä välineillä pystytään toteuttamaan sovelluksen tärkeimmät ominaisuudet?
    - Millä välineillä pystytään toteuttamaan määrityksiin perustuva, kokonaisvaltaisen sovellus rajallisessa ajassa?

2.3 Tutkimusmenetelmä

Opinnäytetyö on tutkimusmenetelmältään kehittämistutkimusta. Kehittämisen kohteena on toimeksiantajan ideoima ja pääkohtineen määritelty sovellus. Teoriapohja tulee koostumaan sovelluksessa käytettävien teknologioiden taustoista ja niiden soveltuvuudesta kehitystehtävään. Tehtyjen valintojen pohjalta voidaan siirtyä kehitysvaiheeseen.

Kananen (2012, 20) mainitsee, että kehittämistutkimuksen työelämän kohteita voivat olla esimerkiksi prosessit, toiminnot, tuotteet, palvelut ja asiantilat. Kehittämistutkimuksen lopputuotoksena syntyvä sovellus voidaan kategorioida tuotteeksi, mutta sen voi ajatella olevan yhtä lailla palvelukin – se palvelee sovellusta käyttäviä valmentajia ja heidän asiakkaitaan. Kehittämistutkimus koostuu kehittämistyöstä ja tutkimuksesta, jossa kehittämistyö noudattaa ilmiölle tyypillistä prosessia (Kananen 2012, 45). Tulokset koskettavat yksittäistapauksia, joista ei voida

tehdä yleistyksiä, sillä muutos koskee vain kehittämisen kohteena ollutta ilmiötä (Kananen 2012, 43). Nämä näkökohdat pätevät myös kehitettyyn sovellukseen.

3 Taustaa JavaScript web-kehityksestä

3.1 JavaScript

JavaScriptistä on muodostunut eräänlainen standardi dynaamisten web-käyttöliittymien tekoon. JavaScript toimii taustalla lähes joka kerta nettisivuilla, missä käytetään esimerkiksi animaatioita, muuttuvaa live-dataa, nappeja, pudotusvalikkoja sekä muita dynaamisia elementtejä. JavaScriptiä voidaan ajaa kaikissa verkkoselaimissa, ja tätä nykyä se on suosituin ja välttämättömin taito, jonka modernin web-kehittäjän tulisi hallita. (Minnick & Holland 2015, Why JavaScript?)

JavaScript esiteltiin vuonna 1995 keinona lisätä ohjelmia nettisivuille Netscape Navigator -selainympäristössä. Sittemmin sen ovat omaksuneet kaikki muut tänäkin päivänä käytetyt verkkoselaimet. Se on tehnyt modernien web-sovellusten kehityksestä mahdollista – sovelluksien, joiden kanssa voidaan olla suorassa vuorovaikutuksessa ilman, että jokainen toiminto käynnistäisi sivun uudelleenlatauksen. Sitä käytetään kuitenkin yhtä lailla myös perinteisimmissä websivuissa dynaamisten elementtien luontiin ja manipulointiin. (Haverbeke 2015, What is JavaScript?)

Nykypäivänä verkkoselain ei ole ainoa ympäristö, missä JavaScriptiä käytetään. Jotkin tietokannat, kuten esimerkiksi MongoDB ja CouchDB, käyttävät JavaScriptiä datan käsittelyyn ja kyselyyn. Myös useat työpöytäsovellukset ja sovellusalustat palvelinpuolen ohjelmoinnissa, ehkä tunnetuimmin Node.js, tarjoavat tehokkaan ympäristön käyttää JavaScriptiä selainten ulkopuolella. (Haverbeke 2015, What is JavaScript?)

3.2 Node.js

Node.js on JavaScript-ajoympäristö, jota käytetään palvelinpään ohjelmoinnissa. Se on tapahtumavetoinen, kevyt ja tehokas. Sen toiminta perustuu pääosin asynkronisiin callback-mekanismeihin. (Salonen 2012.)

Node on suunniteltu rakennustyökaluksi skaalautuviin, yhteyttä ylläpitäviin sovelluksiin. Nodessa monia yhteyksiä voidaan käsitellä samanaikaisesti. Noden toimintamalli on päinvastainen tyypillisimpiin toimintamalleihin verrattuna, joissa käytetään säikeitä. Säie-pohjainen toiminta on suhteellisen tehotonta ja yleensä vaikeaa käyttää. Tämän lisäksi Noden käyttäjien ei tarvitse huolehtia prosessien lukkiutumisesta, sillä lähes mikään funktio Nodessa ei käynnistä suoraan I/O -yhteyttä (non-blocking I/O). Prosessit eivät siis koskaan jumita toisiaan. (About Node.js® n.d.)

npm

Npm on Node.js:n pakettienhallintajärjestelmä (Salonen 2012). Tätä nykyä sitä voidaan kuitenkin pitää yleisesti koko JavaScript yhteisön järjestelmänä.

Npm:n avulla kehittäjät voivat helposti jakaa omia koodejaan muiden kehittäjien kesken ratkaisemaan erilaisia ongelmia erilaisissa ympäristöissä. Hallintajärjestelmän avulla voidaan tarkistaa, jos koodin omistaja on päivittänyt koodejaan. Tällöin ne on myös muiden kehittäjien päivitettävissä. Tämäntyyppisiä uudelleenkäytettäviä, yhteisön kesken jaettuja koodinpätkiä kutsutaan paketeiksi, tai joskus moduuleiksi. Tyypilliset sovellukset, kuten web-sivut, ovat riippuvaisia kymmenistä tai jopa sadoista paketeista. Idea on, että nämä paketit muodostavat rakenneosia, jotka ratkaisevat tiettyjä ongelmia. Tämä mahdollistaa isompien, kustomoitujen ratkaisujen koostamista pienemmistä, jaetuista rakenneosista. (What is npm? n.d.)

3.3 MongoDB

MongoDB on tyypillisistä relaatiokannoista eli SQL-kannoista poiketen niin kutsuttu NoSQL-kanta. Mongossa ja yleisesti NoSQL-kannoissa on relaatiokantoja parempi skaalautuvuus kun tietomassat kasvavat. Ne ovat myös hyviä suorituskyvyltään, ja datan käsittely on yleensä helppoa ja mutkatonta. Siinä missä relaatiokannoissa rivejä tallenetaan tauluihin, Mongossa tallennetaan dokumentteja kokoelmiin. Datan rakenteessa käytetään JSON-muotoa. MongoDB:n etuna voidaan pitää mm. sitä, että datan rakenetta ei tarvitse muuttaa relaatiomuotoon kun sitä ollaan tallentamassa kantaan. Voidaan siis käyttää samaa oliomaista rakennemuotoa, jota on

todennäköisesti käytetty tallennettavan datan koostamisessa muutenkin. Huonoina puolina voitaisiin pitää esimerkiksi sitä, että kannasta hakeminen voi olla ainakin alkuun monimutkaisempaa, sillä käytössä ei ole SQL-kieltä vastaavaa käsittelykieltä helpottamaan hakuja. (Hovi 2015.)

3.4 Meteor

Meteor on kokonaisvaltainen (full-stack) JavaScript sovelluskehys/sovellusalusta modernien web-sovellusten ja mobiilisovellusten kehitykseen. Meteor pitää sisällään avainasemassa olevia teknologioita reaktiivisten sovellusten rakentamiseen. Se sisältää koontityökalun (build tool), sekä toiminalleen tarpeelliset Node.js ja JavaScript yhteisön moduulit ja paketit. (What is Meteor? n.d.)

Robinson, Gray ja Titarenco (2015, The Seven Principles of Meteor) esittelevät Meteorin seitsemän periaatetta, joihin sen toiminta perustuu:

- Data langalla (Data on the Wire). Selainten kehittyessä sivujen yksittäisten osien päivittäminen mahdollistui ilman, että palvelin palauttaisi kokonaisen HTML-sivun selaimeen. Tämä parantaa käyttöystävällisyyttä nopeuttaen sivujen käyttöä. Se vähentää kutsujen määrää palvelimille, vähentäen niiden rasitusta.
- Yksi kieli. Voit kirjoittaa sekä palvelinpuolen että asiakaspuolen koodia JavaScriptillä. Kehitystyö sujuvoittuu, kun kieltä ei tarvitse vaihtaa näiden kahden välillä.
- Tietokantayhteys kaikkialla. Samaa kannan rajapintaa voidaan käsitellä niin palvelin- kuin asiakaspuolellakin.
- Viiveen kompensointi. Nykyään ihmiset odottavat sovelluksilta nopeutta. Yleensä kaikissa etäyhteyksissä on kuitenkin odotettavissa jonkun verran viivettä. Meteorin viiveen kompensoinnilla saadaan aikaan illuusio, että asiat tapahtuisivat välittömästi.
- Kokonaisvaltainen reaktiivisuus. Datan muutoksen hallinta tulee pitää yksinkertaisena. Heti kun data muuttuu kannassa, on muutos havaittavissa myös jokaisessa selaimessa ja jokaisella asiakkaalla, jolla on näköyhteys tähän dataan, reaaliajassa.

- Ekosysteemin hyödynnettävyys. Miksi kehittää pyörää uudelleen, kun voi hyödyntää olemassa olevia avoimen lähdekoodin osia. Meteorin kehityksen nopeuttamisen lisäksi se nopeuttaa kehittäjiä omaksumaan sovelluskehyksen.
- Yksinkertaisuus tuottavuuteen johtamassa. Sovelluskehyksen tehokkuus ja ominaisuuksien määrä ei takaa käyttökelpoisuutta. Sen pitää olla myös yksinkertainen käyttää.

3.4.1 Atmosphere ja npm

Atmosphere on Meteorin oma pakettienhallintajärjestelmä. Se muistuttaa toiminnaltaan hyvin paljon npm:ää, mutta se on suljettu järjestelmä vain Meteorin ympäristöä varten. Atmospheren paketit ovat pelkästään Meteorille tarkoitettuja paketteja. (Atmosphere vs. npm n.d.)

Kehittäjät voivat kirjoittaa omia Atmosphere paketteja noudattaen Meteorin pakettien julkaisuohjeita. Aiemmin npm paketteja ei voitu suoraan käyttää Meteorissa, vaan se vaati erään ”säliö-paketin” asentamista Meteorin omasta pakettienhallintajärjestelmästä. Tämän avulla myös npm:n paketteja voitiin käyttää Meteorin ympäristössä. Kuitenkin versiosta 1.3 lähtien Meteorissa on ollut täysi tuki npm paketeille (Atmosphere vs. npm n.d.)

3.4.2 ES2015 standardi ja moduulit

Meteor on tukenut ES2015 moduuleita versiosta 1.3 lähtien. Tämän standardin avulla muuttujia voidaan siirrellä tiedostosta toiseen *export* ja *import* avainsanoja käyttäen. Arkkitehtuurillisesti ajateltuna tiedostot, joista viedään muuttujia muualla kutsutaan *moduuleiksi*, koska ne edustavat uudelleenkäyttävää koodia. Kun moduuleita ja paketteja *viedään* ja *tuodaan* (export, import) tarkoin määritellysti, koodin rakenteesta tulee modulaarista, jolloin koodi on helpommin jäsenneltävissä ja hallinoitavissa. Modulaarisen koodin kirjoittaminen on tätä nykyä JavaScript-yhteisön trendi modernissa web-kehityksessä. (ES2015 modules n.d.) ES2015-standardin ominaisuudet tulevat jokaiseen Meteorin projektiin oletuksena *ecmascript* Atmosphere-paketin myötä. Paketti on myös poistettavissa, mutta se ei ole suositeltavaa. (ES2015+ (recommended) n.d.)

ES eli ECMAScript on JavaScriptin kielistandardi. Kun sovellus kootaan ja ajetaan, Meteorin *ecmascript*-paketti kääntää kaiken ES2015-standardin mukaisen syntaksin (kuvio 1) kaikkien selainten ymmärtämäksi ”normaaliksi JavaScriptiksi”. Se tarkoittaa myös sitä, että kyseisen standardin mukaista syntaksia ei ole pakko käyttää. (Use the ecmascript package n.d.)

```
import React, {Component} from 'react';
import TrackerReact from 'meteor/ultimatejs:tracker-react';

import { Customers } from '../../api/collections.js';
import { TestResults } from '../../api/collections.js';

import AdminPanelUserSingle from './AdminPanelUserSingle.jsx';

export default class AdminPanel extends TrackerReact(Component) {
```

Kuvio 1. Esimerkki ES2015 syntaksin ja moduulien käsittelytavasta

Kuviossa 1 on esitetty esimerkki ES2015 syntaksista kehitetyn sovelluksen koodista. Tiedostoon tuodaan (import) tarvittavat moduulit ja komponentit, sekä viedään (export) yksi React-komponentti muualle käytettäväksi. Komponentti noudattaa ES2015 JavaScript-luokan syntaksia.

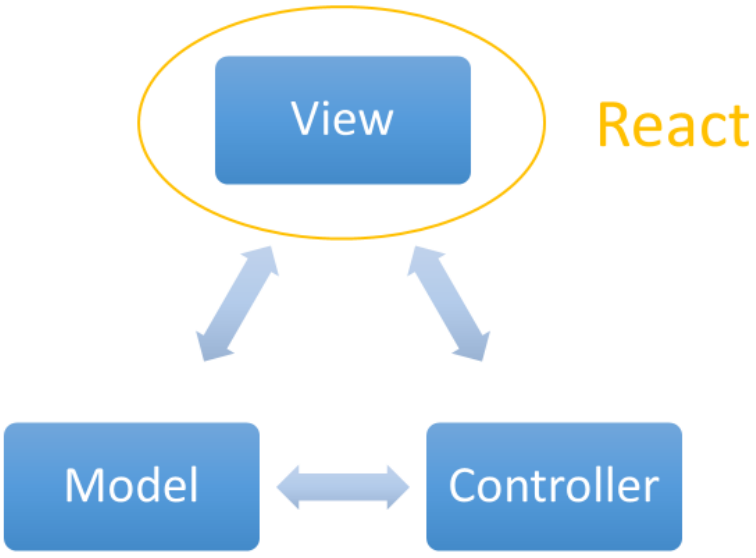
3.4.3 Näkymäkerroksen vaihtoehdot

Meteor tukee kolmea eri käyttöliittymäkirjastoa: Blaze, React ja Angular. Blaze kehitettiin osana Meteoria, React on Facebookin luoma UI-kirjasto, ja Angular on Googlen kehittämä front-end-sovelluskehys. Blaze on yleisesti ottaen helpoiten opittavissa, ja osana Meteorin kehitystä sillä on laajin lisäosavalikoima Atmosphere-pakettijärjestelmässä. Reactin ja Angularin parempina puolina voidaan pitää niiden laajempia yhteisöjä ja dokumentaation määrää. (View layers n.d.)

Mobiilipuolen kehityksessä on tuki Apache Cordovalle, minkä kanssa voidaan käyttää myös kaikkia yllä mainittuja kirjastoja. Natiivin sovelluksen kehitykseen (Android, iOS) voidaan käyttää React Nativea, tai iOS:ää varten voidaan käyttää Meteor iOS -sovelluskehystä. (Mobile n.d.)

3.5 React

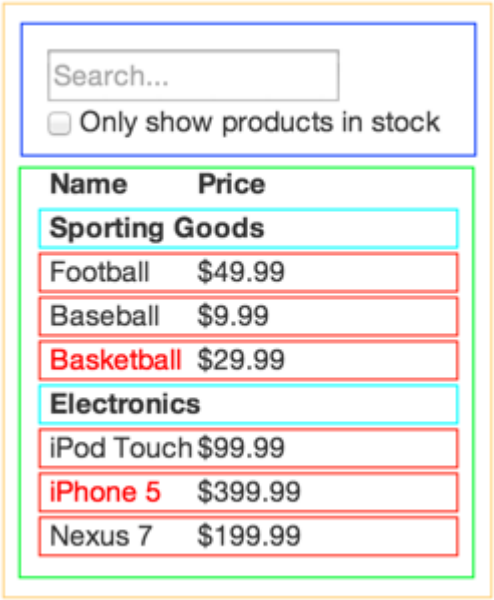
React on Facebookin vuonna 2013 julkaisema JavaScript-sovelluskehys. Se suunniteltiin ratkaisemaan ongelmia monimutkaisissa käyttöliittymissä, joissa data muuttuu ajan myötä. Se poikkeaa monista muista sovelluskehyksistä, jotka perustuvat MVC-arkkitehtuurimalliin (kuvio 2). Monesti sitä ei pidetä edes varsinaisena sovelluskehyksenä. MVC-mallin sovelluskehykset hyödyntävät monesti kaksisuuntaista datavirtaa (two-way data binding) ja templaatteja renderöintiin. React muutti melko radikaalisti web-käyttöliittymien kehitystyön lähestymistapaa. Se poikkeaa paljon muista sovelluskehyksistä ja JavaScriptistä totutuista käytännöistä. (Gackenheimer 2015, Defining React.)



Kuvio 2. MVC-malli ja React

React poikkeaa paljon käytetystä MVC (Model-View-Controller) -arkkitehtuurimallista (kuvio 2). Tyypilliseti MVC-mallin M eli *malli* on vastuussa sovelluksen tilasta, ja tilan muuttuessa se lähettää V osaan eli *näkymään* sovelluksen tilaa muuttavia tapahtumia. Näkymä on loppukäyttäjälle näkyvä osa sovelluksesta ja samalla rajapinta, jonka kautta käyttäjä voi käyttää sovellusta. Kun sovelluksen eri toimintoja käytetään, näkymästä lähetetään näitä ohjaavia tapahtumia C osalle eli *kontrollerille*,

ja joskus myös mallille. Kontrolleri on MVC-mallin suurin koneisto, joka vastaanottaa sovelluksen käyttäjältä tulevat käskyt muuttaen samalla mallia ja näkymää käskyjen mukaan. MVC-mallista on olemassa monia variaatioita ja niihin viitataankin monesti MV* -malleina. Reactin voidaan ajatella olevan pelkkä V eli näkymäosa tästä arkkitehtuurimallista. Sillä määritellään sovelluksen käyttöliittymä ja mekanismit käyttöliittymän muuttamiseen silloin, kun data vaihtuu ajan myötä. React-käyttöliittymä rakentuu deklaratiivisista komponenteista (kuvio 3), jotka määrittelevät käyttöliityymää kokonaisuutena. (Gackenheimer 2015, React Is Not Just Another Framework.)



Kuvio 3. Reactin komponenttijaottelu (Thinking in React n.d.)

React ohjaa ajattelemaan sovelluksen käyttöliittymän muodostuvan hierarkisista komponenteista. Perusidea on se, että komponenteista ja niiden alikomponenteista koostuu lopulta kokonainen käyttöliittymä. Ideaalitilanteessa yksi komponentti tekee ainoastaan yhtä asiaa. Jos komponentti alkaa paisua, se kannattaa hajottaa pienemmiksi alikomponenteiksi. Komponentit tulisi suunnitella uudelleenkäytettäviksi rakennusosiksi. (Thinking in React n.d.) Kuvion 3 samanväriset komponentit ovat sama (eli uudellenkäytettävä) komponentti, jota on monistettu vaihtuvalla datalla.

3.5.1 Virtuaalinen DOM

Reactin ehkä tärkein konsepti on sen *virtuaalinen DOM.* Virtuaalinen DOM on vastuussa siitä, että selain voidaan päivittää mahdollisimman nopeasti, kun käyttöliittymässä tapahtuu muutoksia.

Virtuaalinen DOM on abstraktio oikeasta HTML DOM:ista. Selaimen päivittäminen on nopeampaa, sillä nyt ei tarvitse muokata oikeaa DOM:ia. Se on yleensä hidasta ja siihen liittyy usein selainriippuvaisia seikkoja. Virtuaalinen DOM muistuttaa hyvin pitkälti oikeaa HTML DOM:ia. Tämä mahdollistaa JSX syntaksin käytön, missä voidaan hyödyntää HTML-tageja. (Krajka 2015.)

3.5.2 JSX

JSX on Reactissa käytettävä syntaksi (kuvio 4). Se muistuttaa XML- ja HTML merkintäkieliä. JSX:ää ei ole pakko käyttää, mutta se on suositeltavaa, sillä se on nopea ja tutun tyylinen tapa määritellä puumaisia rakenteita. JSX kääntyy lopulta tavalliseksi JavaScriptiksi. (Gackenheimer 2015, Why Use JSX Instead of Conventional JavaScript?) JSX syntaksi ymmärtää kustomoitujen React-luokkien (eli komponenttien) lisäksi tavallisia HTML-tageja (Gackenheimer 2015, JSX).

```
// JSX versio

React.render(
    <div>
        <h1 className="header">Otsikko</h1>
    </div>
);

// ..käännettynä React elementeiksi ilman JSX syntaksia

React.render(
    React.createElement('div', null,
        React.createElement('h1', {className: 'header'}, 'Otsikko')
    );
);
```

Kuvio 4. JSX syntaksi

Reactin koodia voi siis kirjoittaa joko JSX syntaksilla tai ilman kuvion 4 osoittamalla tavalla. Kärjistetysti voidaan sanoa, että JSX on kuin JavaScriptiä, joka sisältää HTML:ää. Esimerkistä voidaan huomata, että otsikkoelementin attribuuttina on *className* eikä *class*, jota käytetään HTML-kielessä. Tämä johtuu siitä, että JSX (ja Reactin ”normaali” JavaScript) on pohjimmiltaan tavallista JavaScriptiä. JavaScriptin nimiavaruudessa class on varattuna määrittelemään luokkia ES2015-standardista lähtien.

4 Tarkempi katsaus Meteorin teoriaan

Kehittäjänä sain vapaat kädet valita välineet sovelluksen toteuttamiseen, mikäli niillä saavutettaisiin määriteltyihin ominaisuuksiin ja toiminnallisuuteen perustuva lopputulos. Meteor valikoitui kehitettävän sovelluksen sovellusalustaksi. Perusteluihin paneudutaan tarkemmin opinnäytetyön loppuosassa, jossa niitä peilataan tutkimuskysymyksiin.

Ennen toteutusosiota on syytä paneutua hetkeksi Meteorin teoriaan, jolla havainnollistetaan sovellusalustan ominaispiirteitä tarkemmin. Teoriakatsauksen jälkeen sovelluksen toteutusosio on helpommin ymmärrettävissä, ja se selittää osaltaan, miksi kyseinen sovellusalusta valikoitui kehitetyn sovelluksen tekoon. Nopean teoriakatsauksen jälkeen siirrytään varsinaiseen toteutusosioon luvussa 5.

4.1 Datan käsittely

Meteorissa dataa käsitellään kokoelmien (collection) kautta, ja data tallentuu MongoDB kantaan. Kokoelma on tietokannassa joukko toisiinsa yhteydessä olevia tietoja. Meteorin ohjelmakoodissa kokoelman voidaan myös ajatella olevan datan pysyvyysmekanismi (persistence mechanism). (MongoDB collection in Meteor n.d.)

Meteorissa on mahdollista kirjoittaa joko palvelinpuolen- tai asiakaspuolen koodia, tai koodia joka toimii kummallakin puolella. Tähän liittyy erinäisiä sääntöjä kansiorakenteista ja kansioiden nimistä. Meteorissa kaikki koodi on samassa koodikannassa.

Esimerkiksi *server*- tai *private*-kansioissa oleva koodi toimii ainoastaan palvelinpuolella. *Client*-kansiossa oleva koodi suoritetaan puolestaa asiakaspuolella jne. Meteorin funktioilla ehtolauseisiin yhdistettyinä *if Meteor.isClient* ja *if Meteor.isServer* voidaan myös ohjailla sitä, millä puolen koodi suoritetaan. (Special Directories n.d.)

Kun palvelinpuolen koodissa määritellään uusi kokoelma:

```
Asiakkaat = new Mongo.Collection('asiakkaat');
```

…samalla luodaan kokoelma MongoDB kantaan. Määritelmä luo myös rajapinnan kyseiseen kokoelmaan kannassa, eli käytössä on toimiva kantayhteys ja kokoelmaa voidaan käsitellä näin koodista käsin. Asiakaspuolen koodissa samalla tavalla määritelty kokoelma toimii myös rajapintana, mutta tällöin kokoelmalla ei ole yhteyttä oikeaan tietokantaan. Sen sijaan tämä kokoelma on yhteydessä asiakaspuolen Minimongo ”välimuistikantaan”. Minimongo-kirjasto on muistissa toimiva implementaatio MongoDB:n API:sta. (MongoDB collection in Meteor n.d.) Minimongon toiminta liittyy oleellisesti siihen, miten data päivittyy välittömästi käyttöliittymässä kantaoperaatioita tehdessä.

4.2 DDP-protokolla

Distributed Data Protocol eli lyhyemmin DDP on Meteoria varten kehitetty tiedonvälitysprotokolla, jonka avulla tieto liikkuu palvelimen ja asiakkaan välillä. DDP perustuu WebSocket -protokollaan ja se noudattaa JSON-tietorakennetta. DDP tekee pääasiassa kahta asiaa: se käsittelee etäyhteydellisiä proseduurikutsuja (Remote Procedure Call - RPC) sekä se vastaa datan käsittelystä. (Introduction to DDP 2014.)

Datan julkaiseminen ja tilaaminen

Web-sovellusten ja web-sivujen dataliikenne kulkee palvelimen ja asiakkaan välillä tyypillisesti http-protokollan välityksellä. Nykyään käytetään monesti REST-rajapintoja, joissa palvelimelle lähetetään http-pyyntöjä. Palvelin vastaa palauttamalla dataa joko HTML- tai JSON-muodossa. Palvelin ei kuitenkaan kykene

”työntämään” dataa asiakkaalle automaattisesti, jos palvelinpuolella tapahtuu muutoksia. (Publications and subscriptions n.d.)

Meteorin DDP-protokollan ansiosta data liikkuu molempiin suuntiin. Meteor-sovelluksessa ei tarvitse käyttää REST-rajapintapäätteitä sarjallistaakseen ja lähettääkseen dataa. Sen sijaan kannan dataa *julkaistaan* (publication) palvelinpuolelta. Asiakaspuolella määritetään, mitä julkaisuja *tilataan* (subscribe). Tiedot, joita *tilataan* on aina sen hetkinen joukko dataa kannassa – kannan tietojen muuttuessa *julkaisut* päivittyvät ja täten niitä *tilaavat* päätteet päivittyvät myös uudella tiedolla. Datan *julkaiseminen* ja *tilaaminen* (kuviot 5 ja 6) luovat myös ”sillan” palvelinpuolen MongoDB-kannan ja asiakaspuolen Minimongo-välimuistikannan välille. (Publications and subscriptions n.d.)

```
Meteor.publish("allCustomers", function() {
    return Customers.find();
});

Meteor.publish("allTestresults", function() {
    return TestResults.find();
});

Meteor.publish("allUsers", function() {
    if (Roles.userIsInRole(this.userId, ['admin'])) {
        return Meteor.users.find({});
    }
});
```

Kuvio 5. Esimerkki datan julkaisusta.

```
this.state = {
    subscribtion: {
        allCustomers: Meteor.subscribe("allCustomers"),
        allResults: Meteor.subscribe("allTestresults"),
        allUsers: Meteor.subscribe("allUsers")
    }
}
```

Kuvio 6. Esimerkki datan tilaamisesta

Kuviot 5 ja 6 esittävät esimerkin datan *julkaisusta* ja *tilaamisesta*. Ensiksi dataa julkaistaan palvelinpuolen koodissa. Tämän jälkeen julkaisut tilataan asiakaspuolen koodissa, tässä tapauksessa React-komponentin state -objektissa.

Metodit

Meteorin kontekstissa metodilla viitataan aikaisemmin mainitun proseduurikutsujärjestelmän (RPC) työkaluun, jolla voidaan suorittaa käyttäjän ”laukaisemia” tapahtumia ja tietojen tallennuksia. Kaikessa yksinkertaisuudessaan metodi toimii siten, että se määritellään oikealla tavalla, jonka jälkeen siihen voidaan olla yhteydessä metodikutsulla (kuviot 7 ja 8). (What is a Method? n.d.)

```
Meteor.methods({
    'lisaaAsiakas'(asiakas) {
        Asiakkaat.insert({
            etunimi: asiakas.etunimi,
            sukunimi: asiakas.sukunimi,
        }, function(error, result) {
            ....
        });
    }
});
```

Kuvio 7. Esimerkki Meteorin metodista

```
const asiakas = {etunimi: 'Aku', sukunimi: 'Ankka'};

Meteor.call('lisaaAsiakas', asiakas, (error) => {
    if (error) {
        Alert('ERROR!');
    } else {
        Alert('Asiakas lisätty.');
    }
});
```

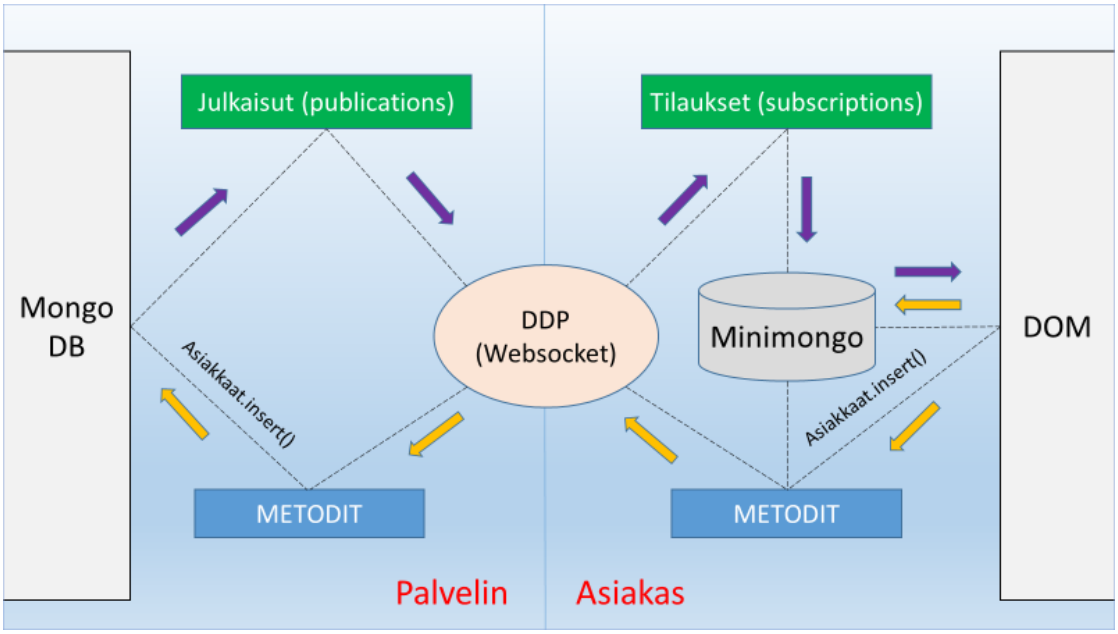
Kuvio 8. Esimerkki metodikutsusta

Kuviot 7 ja 8 esittävät yksinkertaisen tietokantaoperaation, jossa lisätään uusi asiakas tietokantaan. Ensiksi määritellään Meteorin metodi asiakkaan tallennusta varten. Tämän jälkeen luodaan uusi kuvitteellinen ja kovakoodattu asiakas, jota käytetään

metodikutsun parametrina. Todellisessa tilanteessa asiakkaan tiedot napattaisiin käyttäjän syötteistä.

Mikäli Meteor-sovelluksessa halutaan hyödyntää niin kutsuttua *viiveen kompensointia* ja *optimistista käyttöliittymää*, tulisi metodit määritellä koodissa siten, että niihin on pääsy sekä palvelin- että asiakaspuolelta. Tämä mahdollistaa sen, että metodeita kutsuttaessa asiakaspuolella toimiva Minimongo *pystyy simuloimaan samoja metodeja.* Tällöin metodeissa suoritettavat kantaoperaatiot suoritetaan myös muistissa toimivaan Minimongo-”kantaan” (katso kuvio 9). (Optimistic UI n.d.) Viiveen kompensointimekanismia ei ole pakko käyttää. Joissain tilanteissa sen käyttö ei ole edes järkevää. Sen käyttö riippuu tilanteesta ja kehitettävästä sovelluksesta.

Meteorin dataliikenne



Kuvio 9. Datan virtaus Meteorissa (alkuperäinen kuvio Goldshtein 2015)

Kuviossa 9 on pyritty hahmottamaan Meteorin datan virtausta nojaten edellä käytyyn teoriaan. Uutta asiakasta lisättäessä kutsutaan sitä varten määriteltyä *metodia*, joka tekee lopulta tietokantatallennuksen. Kantaoperaatio ajetaan samanaikaisesti myös Minimongoon, mikäli metodi on määritelty sekä palvelin- että asiakaspuolella toimivaksi. Käyttöliittymä reagoi heti tallennuksen aiheuttamaan

muutokseen, jolloin syntyy illuusio siitä, että data päivittyisi oikeassa kannassa välittömästi. Todellisuudessa siihen menee pieni hetki. Kun lisätty asiakas on saavuttanut oikean kannan tietoineen, asiakas-kokoelman *julkaisu* päivittyy. Tieto kulkee sitä *tilaaviin* päätteisiin ja tällöin ”oikea” asiakas synkronoidaan myös Minimongoon.

Minimongon ominaisuuksia hyödynnettäessä tieto päivittyy asiakkaan käyttöliittymässä salamannopeasti. Tämän mahdollistavaa mekanismia kutsutaan Meteorissa *optimistiseksi käyttöliittymäksi* tai *viiveen kompensoinniksi*. (Stubailo 2015.)

5 Sovelluksen toteutus

Sovelluksen ominaisuuksista oli olemassa jo melko tarkat määritelmät kehitystyön alkaessa. Kehitystyön eri vaiheissa määritelmät saattoivat kuitenkin hieman muuttua. Uusia ominaisuuksia lisättiin, jos ne nähtiin tarpeellisiksi. Siinä mielessä kehitys muistuttikin hieman ketterää ohjelmistokehitystä, vaikka mitään ketterän kehityksen menetelmää ei tosiasiassa käytettykään.

Tässä luvussa esitetään sovelluksen tärkeimmät toteutusosiot alkaen sen määrittelyistä. Osiot sisältävät esimerkkejä sovelluksen koodista sekä kuvankaappauksia sovelluksen käyttöliittymästä, joiden tarkoitus on toimia tekstin tukena.

5.1 Sovelluksen määrittelyt

Käyttäjät ja asiakkaat

Asiakkailla tarkoitetaan valmentajan palvelemia henkilöitä, joiden kuntotestituloksia mitataan. Asiakkaita varten ei tarvita omaa käyttäjätasoa, vaan he ovat osa sovelluksen käyttäjiin liittyviä henkilöitä. Sovelluksen varsinaisia käyttäjatasoja on kaksi.

- Sovelluksen peruskäyttäjät ovat valmentajia, jotka mittaavaat kuntotestejä asiakkaidensa hyväksi. Käyttäjien tulisi pystyä lisäämään itselleen asiakkaita.

Heidän tulisi pystyä merkitsemään kuntotestien tulokset järjestelmään, jolla voidaan koostaa raportteja asiakkaille.

- Admin-käyttäjän määritelmät ja toteutus tehtiin kehitystyön loppupuolella. Admin-käyttäjä näkee kokonaisuudessaan kaikki sovelluksen peruskäyttäjät, ja pystyy tarpeen vaatiessa poistamaan niitä. Hän hallinnoi sovelluksen peruskäyttäjiä kuponkijärjestelmän avulla. Hän pystyy latamaan tiedot kaikista käyttäjistä, käyttäjien asiakkaista ja asiakkaiden kuntotestien tuloksista. Kupongit ovat tarkoitettu erittelemään sovelluksen käyttäjäryhmiä toisistaan. Ne toimivat samalla pääsylippuina sovellukseen – ilman toimivaa kuponkia sovellukseen ei voi tehdä käyttäjätiliä. Admin-käyttäjää varten tulisi kehittää omat näkymät, mihin peruskäyttäjillä ei olisi pääsyä.

Kuntotestien tulokset ja raportit

Yksi tärkeimmistä ominaisuuksista oli kuntotestitulosten syöttäminen sovellukseen, joista voidaan koostaa kuntotestiraportteja. Kun testien tulokset syötetään, ne tallentuvat kantaan. Kannasta haetuista tuloksista tulisi pystyä koostamaan raportteja. Raporteissa näkyy kuntotestien lopulliset tulokset. Lopulliset tulokset perustuvat todellisiin, ennalta määriteltyihin viitearvoihin, joita kuhunkin kuntotestiin liittyy. Tätä varten tulisi kehittää toimintatapa, joka laskee lopulliset tulokset syötettyjä tuloksia ja viitearvoja vertailemalla. Tästä lisää luvussa 5.5.

5.2 Käyttäjätilien toteutus

Käyttäjätilit on toteutettu muutamia Atmosphere-paketteja käyttämällä. Sovelluksessa on käytetty *accounts-password*-pakettia, johon sisältyy myös *accounts-base*-paketin ominaisuudet. *Alanning:roles*-paketti vastaa käyttäjätasoista, ja *ian:accounts-ui-bootsrap-3*-paketti luo yksinkertaisen käyttöliittymäosan käyttäjätilien hallintaan.

Accounts-base-paketin myötä käyttäjien lisäämiseen voidaan hyödyntää käyttäjät-kokoelmaa (users). Kokoelma noudattaa Meteorin valmiiksi määrittelemää skeemaa (schema). Kokoelman tietokantatoimintojen ohjaamiseen käytetään *Meteor.users*-

rajapintaa. Paketti sisältää ison määrän toimintoja, joilla voidaan ohjata käyttäjien liikennettä ja toimintaa sovelluksessa (kuvio 10). (accounts-base n.d.)

Accounts-password paketti luo turvallisen salasanapohjaisen sisäänkirjautumisen osaksi käyttäjätilejä (accounts-password n.d.). Paketti sisältää kokonaisvaltaisen systeemin salasanapohjaiseen autentikaatioon. Kirjautumiseen voidaan käyttää käyttäjänimen lisäksi sähköpostiosoitetta (kuvio 11). Paketin avulla käyttäjälle voidaan myös lähettää sähköpostin verifiointi- ja salasanan palautusviestejä. (Passwords n.d.)

```
Meteor.call('checkCoupon', coupon, (error, result) => {
    if (error) {
        Alert.warning("Kuponki ei ole käytössä");
    } else {
        let user = {
            email: email,
            password: password,
            fname: fname,
            lname: lname,
            coupon: result
        };
        Accounts.createUser(user, (error) => {
            if (error) {
                if (error.reason === "Username already exists.") {
                    Alert.warning("Käyttäjätunnus on jo käytössä!");
                } else if (error.reason === "Email already exists.") {
                    Alert.warning("Tunnus on jo käytössä!");
                }
            } else {
                this.refs.email.value = "";
                this.refs.password.value = "";
                this.refs.passwordcheck.value = "";
                this.refs.fname.value = "";
                this.refs.lname.value = "";
            }
        });
    }
});
```

Kuvio 10. Uuden käyttäjän rekisteröinti

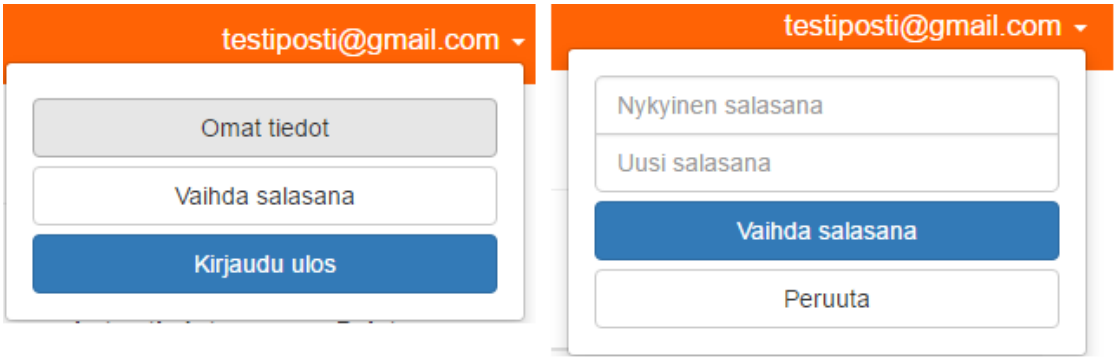
```
loginUser(event) {
    event.preventDefault();

    const email = this.refs.email.value.trim();
    const password = this.refs.password.value.trim();

    Meteor.loginWithPassword(email,password, function(error, result) {
        if (error) {
            Alert.warning("Tarkista tunnus ja salasana");
        }
    });

    this.refs.email.value = "";
    this.refs.password.value = "";
}
```

Kuvio 11. Käyttäjän sisään kirjaaminen

Kuvioissa 10 rekisteröidään uusi käyttäjä sovellukseen. Kuviossa 11 kirjataan käyttäjä sisään. Näitä ohjaavat operaatiot tapahtuvat niitä varten määritellyissä React-komponenteissa. Rekisteröinnissä kutsutaan Meteorin metodia, joka tarkistaa kannasta syötetyn kupongin todenperäisyyden. Jos kuponki on ok, uusi käyttäjä rekisteröidään. Sisäänkirjaaminen on tätäkin yksinkertaisempi operaatio.

Käyttäjätasojen erottelu on tehty *alanning:roles*-paketilla. Paketin avulla jokaiseen luotuun käyttäjään voidaan liittää rooli eli käyttäjätaso. Paketti tarjoaa yksinkertaisen tavan tarkistaa, mikä rooli käyttäjällä on. Sen perusteella voidaan määritellä esimerkiksi mihin metodeihin ja julkaisuihin kullakin käyttäjällä on oikeus. (alanning:roles n.d.) Sovelluksessa käyttäjätaso tarkistetaan esimerkiksi Admin-näkymiä varten tehdyissä React-komponenteissa. Jos peruskäyttäjä yrittää jostain syystä siirtyä admin-näkymiin, pääsy evätään ja käyttäjä ohjataan peruskäyttäjän näkymiin reitittimen avulla.

Accounts-ui-paketti luo yksinkertaisen käyttöliittymäosan käyttäjätilin hallintaan (kuvio 12). Se tarvitsee toimiakseen vähintään *accounts-password*-paketin asennuksen. Myös esimerkiksi *accounts-facebook*- tai *accounts-github*-paketteja voidaan käyttää, jotka mahdollistavat sisäänkirjautumisen näiden palveluiden kredentiaaleilla. (accounts-ui n.d.)



Kuvio 12. Yksinkertainen käyttöliittymäosa käyttäjätilien hallintaan.

Accounts-ui-pakettia käytetään useasti vain sovellusten prototyypeissä. Kehitetyssä sovelluksessa käytettiin *ian:accounts-ui-bootsrap-3*-pakettia (kuvio 12). Se on periaatteessa sama paketti, mutta se on toteutettu Bootstrapilla. Bootsrapia käytettin sovelluksen ulkoasun rakentamisessa, joten paketin UI-osa sopi sovellukseen ulkonäöllisesti paremmin. Paketti oli käännetty usealle eri kielelle, ja sen sai toimimaan myös suomenkielisenä. Yllä olevasta kuviosta on huomioitava se, että “Omat tiedot” -valinta vie sovellusta varten tehtyyn omaan profiilinäkymäänsä, mikä ei sidoksissa kyseiseen pakettiin.

5.3 Tietokanta

Tietokanta koostuu kokonaisuudessaan viidestä eri kokoelmasta. Yksi oletuskokoelma toimii oletusmallilla, mutta muut kokoelmat ovat itse määriteltyjä niiden malleja myöten.

- users – Meteorin oletuskokoelma (oletusmallilla), kun käytetään Meteorin accounts-järjestelmää. Sisältää kaikki sovelluksen käyttäjät.
- customers – kokoelma, joka sisältää kaikki asiakkaat. Jokainen dokumentti kokoelmassa on sidoksissa yhteen käyttäjään, eli jokainen asiakas kuuluu jollekin käyttäjälle.
- testresults – sisältää kaikki tallennetut testitulokset. Yhdessä dokumentissa voi olla ja monesti onkin useita eri testien tuloksia. Esimerkiksi cooperin testin tulos ja vatsalihastestin tulos. Testit kuuluvat aina jollekin tietylle asiakkaalle.

- coupons – yksinkertainen kokoelma, mikä sisältää adminkäyttäjän lisäämät kupongit.
- images – käyttäjät voivat lisätä halutessaan kuvan, joka korvaa raportteihin tulevan oletuskuvan (Trainer4You logon). Kuvat tallentuvat tähän kokoelmaan.

Skeemat ja datan validointi

Skeema (schema) eli malli on tapa määritellä säännöstö, jota tulee noudattaa kun dataa tallenetaan kantaan. MongoDB toimii ilman säännöstöäkin, mutta ilman sitä kantaan voidaan tallentaa periaatteessa mitä tahansa tietoa, mikä koituu tietenkin ongelmaksi. Tallennettavaa dataa verrataan määriteltyyn malliin. Uutta dokumenttia ei tallenneta kokoelmaan, jos data poikkeaa mallista. Loppukäyttäjälle olisi aina syytä ilmoittaa, mitä väärää tietoa on yritetty syöttää.

Meteorissa yleisemmin käytetty paketti skeeman määrittelyyn on *aldeed:simple-schema.* Paketin avulla voidaan määritellä kullekin kokoelmalle skeema, joka sisältää tiettyjä sääntöjä. Kantaan menevää dataa eli tallennettavia dokumentteja voidaan tällöin validoida skeemoja kohden. (Defining a schema n.d.) Validointi voidaan tehdä manuaalisesti, mutta helpoin tapa siihen on käyttää *aldeed:collection2* pakettia. Käyttämällä paketin *attachSchema* metodia, kokoelmiin voidaan liittää skeemat, jotka tarkastetaan aina automaattisesti, kun kokoelmiin tehdään jatkossa tallennusoperaatiota (kuvio 13). (Using schemas on write n.d.)

```
// tuodaan kuponkien kokoelma tiedostosta, jossa se on määritelty
import { Coupons } from '../imports/api/collections.js';

// luodaan kupongeille schema (aldeed:simple-schema)
var Schemas = {};

Schemas.Coupons = new SimpleSchema({
    code: {type: String},
    definition: {type: String},
    createdAt: {type: Date, defaultValue: new Date()}
});

// yhdistetään schema kokoelmaan validiointia varten (aldeed:collection2)
Coupons.attachSchema(Schemas.Coupons);
```

Kuvio 13. Kuponkikokoelman skeeman määrittely

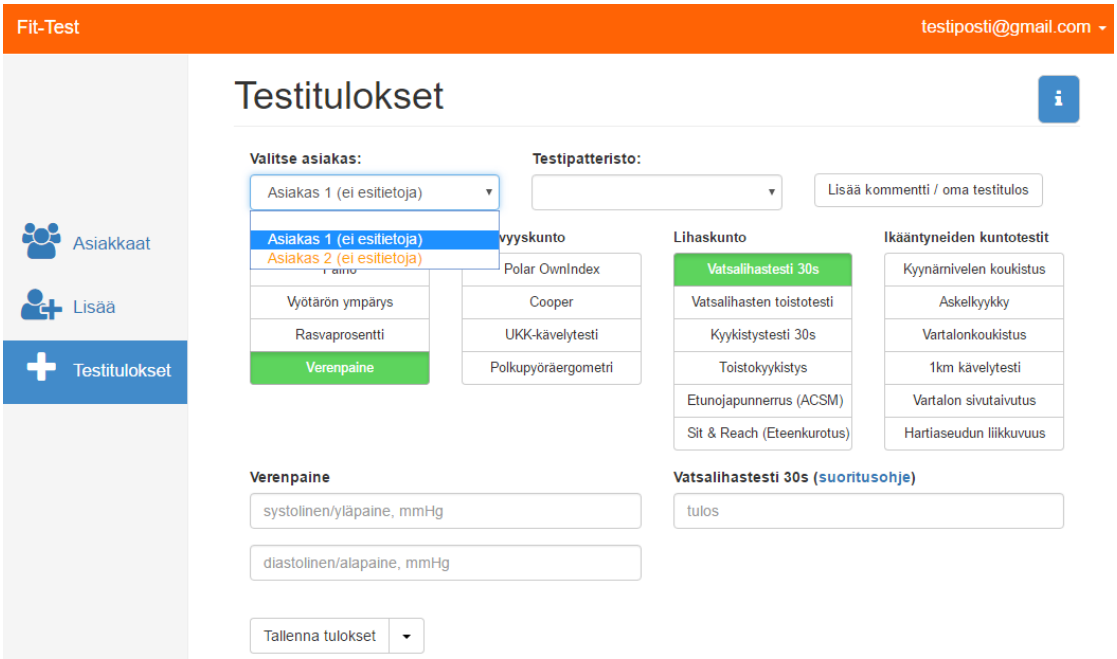
Kuviossa 13 tehdyn määrittelyn jälkeen kaikki kuponkikokoelman talennusoperaatiot validoidaan aina automaattisesti. Enempää koodia ei tarvinnut kirjoittaa kuponkikokoelman validointia varten.

Skeemojen määrittely oli ehdottoman tärkeää kehitetyssä sovelluksessa. Esimerkiksi testituloskokoelmassa (testresults) oli tärkeää määritellä testikohtaisesti, käsitelläänkö tulosta esimerkiksi desimaalilukuna tai koostuuko tulos kenties useasta eri arvosta. Esimerkiksi verenpainetestin ylä- ja alapaine. Tulosten muodot tuli pitää yhteisenä senkin takia, että niiden pohjalta laskettaisiin lopulliset tulokset raportteihin viitearvojen avulla. Nämä paketit toimivat yhdessä mainiosti siltäkin osin, että kehitysvaiheessa ne antoivat aina tarkan virheilmoituksen kehityskonsoliin, mikäli väärää tietoa yritettiin tallentaa. Tämän ansiosta erilaiset virheilmoitukset oli helppo eritellä toisistaan. Virheilmoitusten informatiivisuuden ansiosta loppukäyttäjää voitiin ohjailla tarkoin ilmoituksin käyttöliittymässä.

5.4 Käyttöliittymät

Peruskäyttäjän käyttöliittyymän (kuvio 14) toteutukseen kului eniten aikaa. Luonnollisestikin, sillä se sisälsi eniten toimintoja. Se toteutettiin ensimmäisenä sen tärkeyden vuoksi. Mutta myös siksi, että idea admin-tason käyttäjästä omine näkymineen syntyi vasta myöhemmällä osalla kehitystyötä. Admin-käyttäjän käyttöliittymä (kuvio 15) oli nopeampi tehdä, sillä se oli yksinkertaisempi, eikä sitä tarvinnut sen kummemmin koristella. Kaikille käyttätasoille yhteisiä näkymiä ovat erilliset sisäänkirjautumis- ja rekisteröintinäkymät. Monia peruskäyttäjän näkymiä yhdistää infonappi, jota painamalla käyttäjä voi avata näkymäkohtaiset ohjeet sovelluksen käyttöön.

Pääkäyttöliittymä

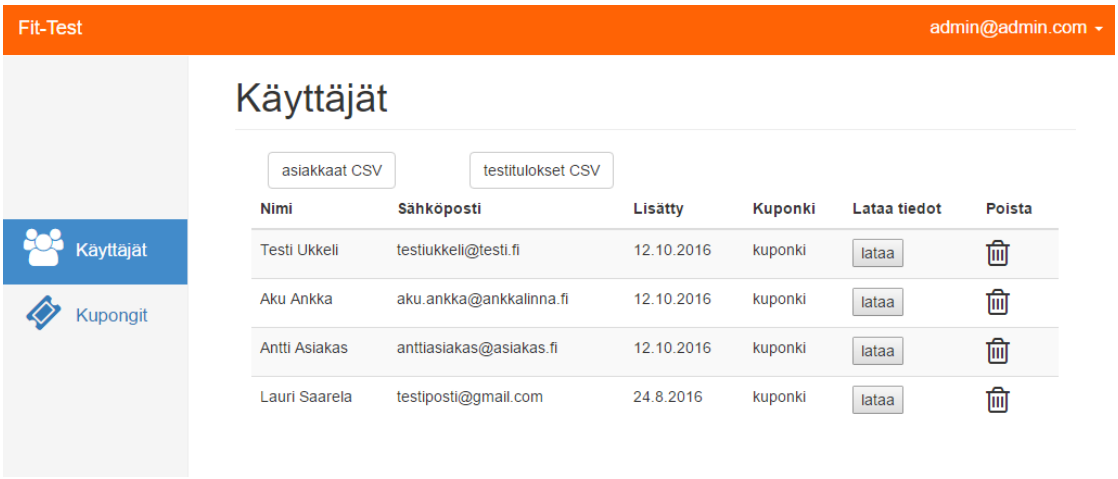


Kuvio 14. Pääkäyttöliittymä

Peruskäyttäjän käyttöliittymässä (kuvio 14) on kaikkiaan kuusi näkymää.

- ‘Asiakkaat’ -näkymä. Oletusnäkymä sisään kirjautuessa. Näkymässä on listattuna kaikki asiakkaat. Näkymästä voi hakea asiakkaita nimellä, poistaa niitä, sekä siirtyä tutkimaan asiakkaan täyttämiä esitietoja. Näkymästä voi myös siirtyä tutkimaan asiakkaan henkilökohtaisia tietoja ja tuloshistoriaa omiin näkymiinsä. Näkymä sisältää mahdollisuuden ladata asiakkaiden tiedot csv-tiedostoina.
- ‘Lisää asiakkaita ’ -näkymä. Uuden asiakkaan lisäämiseen.
- ‘Testitulokset’ -näkymä (kuvio 14). Asiakkaiden testitulosten lisääminen, ja lisätyistä tuloksista raporttien koostaminen.
- ‘Asiakkaan tiedot / muokkaus’ -näkymä.
- ‘Tuloshistoria’ -näkymä. Näyttää asiakkaan tuloshistorian kokonaisuudessaan. Näkymästä voidaan koostaa laajempia vertailevia raportteja, jossa mukaan voidaan ottaa tuloksia kautta koko asiakkaan tuloshistorian.
- ‘Omat tiedot’ -profiilinäkymä. Käyttäjätietojen muuttamiisen ja oman raporttikuvan tallentamiseen.

Admin-käyttöliittymä



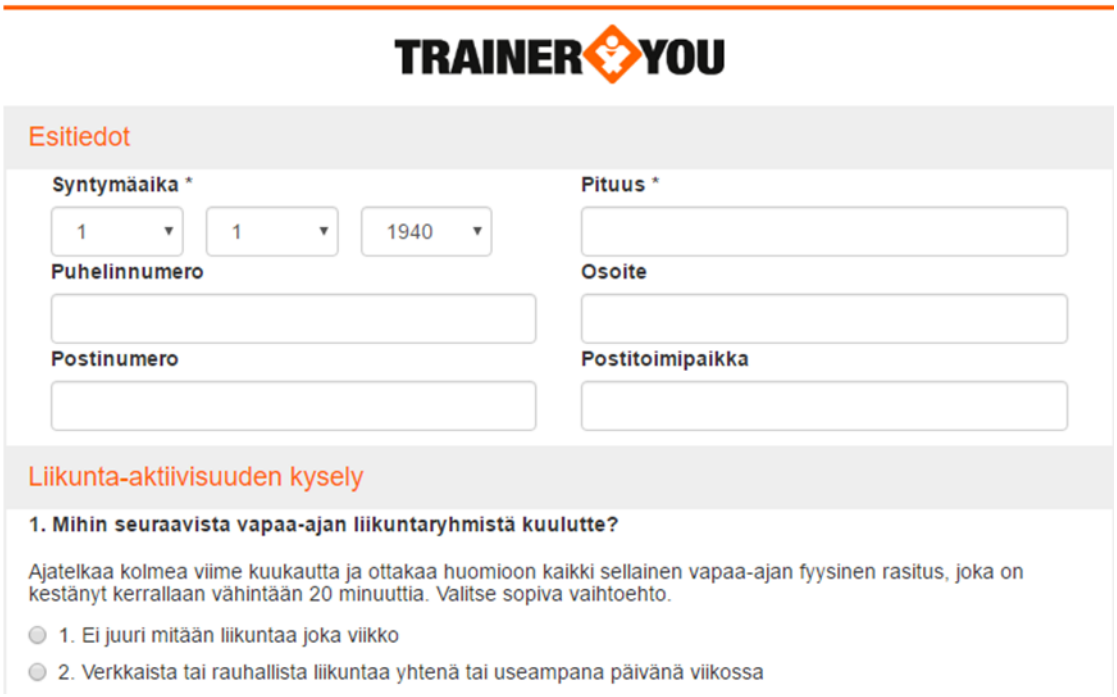
Kuvio 15. Adminkäyttöliittymä

Admin-käyttäjällä on yksinkertaisempi, kahdesta näkymästä koostuva käyttöliittymä (kuvio 15).

- ‘Käyttäjät’ -näkymä. Lista kaikista sovelluksen käyttäjistä. Mahdollisuus poistaa käyttäjiä ja ladata käyttäjien tiedot (csv). Mahdollisuus kaikkien asiakkaiden tietojen ja testitulosten lataukseen.
- ‘Kupongit’ -näkymä. Yksinkertainen näkymä kuponkien lisäämiseen.

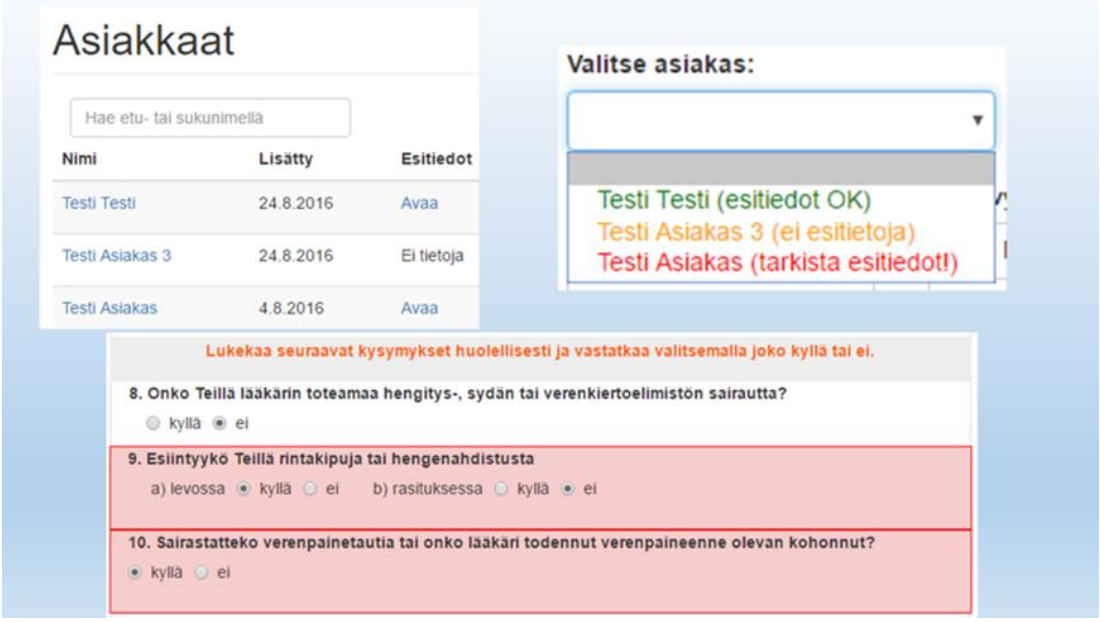
Asiakkaan rajapinta sovellukseen

Kehitystyön loppupuolella sovellukseen tehtiin muutamia lisäyksiä. Yksi niistä oli kehittää keino, joilla asiakkaat voisivat täyttää esitietonsa järjestelmään. Asiakkaat eivät ole sovelluksen varsinaisia käyttäjiä, joten tätä operaatiota varten asiakkaille piti kehittää oma rajapinta sovellukseen. Kuvion 16 esittämä esitietokysely oli ratkaisu tähän.



Kuvio 16. Ote asiakkaan esitietokyselystä

Kun valmentaja lisää uuden asiakkaan, hän voi lähettää asiakkaalle esitietokyselyn sähköpostitse (kuvio 16). Asiakkaalle lähetetty sähköposti sisältää linkin esitietokyselylomakkeeseen. Kun asiakas on täyttänyt lomakkeen tiedoillaan, ne tallentuvat kantaan. Käyttäjä näkee sovelluksesta, onko tiedot täytetty (kuvio 17).



Kuvio 17. Varoitus asiakkaan terveysongelmista

Sovellus varoittaa valmentajaa, mikäli asiakkaan esitiedoissa on joitain seikkoja, joita tulisi ottaa huomioon kuntotestejä tehdessä. Onko asiakkaalla esimerkiksi sydänvaivoja, tai jotain muita ongelmia terveyden kanssa. Asiakkaan täyttämistä esitiedoista voidaan käydä tarkistamassa, mikä kohta on aiheuttanut varoituksen kuvion 17 osoittamalla tavalla.

5.5 Kuntotestiraportit

PDF-raporttien toteutus perustuu *Webshot* npm-paketin käyttöön. Webshotin avulla voidaan ottaa kuvankaappauksia web-sivuista. Se käyttää tähän toimintoon *PhantomJS-kirjastoa* (node-webshot n.d.). PhantomJS käyttää WebKit -selainmoottoria sivujen renderöintiin ja kuvankaappauksiin (Screen Capture n.d.). PhantomJS:n voi ajatella olevan “virtuaaliselain”, jota voidaan ajaa palvelinympäristössä. *Meteorhacks:ssr* paketin ansiosta palvelinpuolella voidaan koostaa ja renderöidä HTML-sivuja käyttämällä Meteorin blaze-templaatteja (meteorhacks:ssr n.d.). Näiden ominaisuuksien avulla sovelluksen palvelinpuolen koodissa on kirjoitettu valmis raporttipohja, josta koostetaan HTML-sivu kuvankaappausta varten. Sovelluksen PDF-raportit ovat siis palvelimella dynaamisesti koostettujen HTML-sivujen kuvankaappauksia, jotka on muotoiltu raporttien näköisiksi yhdessä CSS:n ja Webshotin asetuksien avulla.

Raportteihin tulevat kuntotestien “lopulliset tulokset” perustuvat todellisiin kuntotestien viitearvoihin. Sitä varten oli kehitettävä systeemi, mikä laskee tulokset lopulliseen muotoon raporttien generoinnin yhteydessä. Sovelluksen koodissa on yksi tiedosto, mikä sisältää yksinkertaisen, joskin laajan JavaScript-luokan metodeineen (~1600 riviä koodia). Jokainen luokan metodi vastaa yhtä kuntotestin viitearvoihin perustuvaa “laskuria”. Eri kuntotesteja sovelluksessa on kaikkiaan 20.

Esimerkiksi, jos halutaan laskea lopullinen tulos cooper-testin tulokselle, kannasta haetaan asiakas ja hänen tuloksensa. Cooper-testin “laskuri” laskee lopullisen tuloksen näitä tietoja käyttäen. Laskeminen tarkoittaa tässä tapauksessa tallennettujen tulosten vertailua viitearvoihin. Muuttujina käytetään testistä riippuen yleensä myös asiakkaan sukupuolta ja ikää. Iso koodimäärä kyseisessä tiedostossa selittyykin sillä, että vertailevia ehtolauseita piti kirjoittaa hyvin paljon. Jokainen

“laskuri”, eli luokan metodi palauttaa arvona ns. lopullisen tuloksen. Esimerkiksi “terveysluokka 2”, “erittäin hyvä”, “keskinkertainen”.

Raportin prosessoiminen etenee päävaiheineen seuraavasti: käyttäjä valitsee käyttöliittymässä asiakkaan ja tälle mitattavan tuloksen. Tämä voidaan tehdä kahdella tapaa: Ensimmäinen tapa on syöttää asiakkeelle uusi tulos tai useita tuloksia eri testeihin. Juuri syötetystä tuloksista voidaan välittömästi koostaa uusi raportti. Toinen tapa on koostaa laajempia raportteja asiakkaan tuloshistorianäkymästä, jossa vertailun vuoksi mukaan voidaan ottaa useita testien mittauskertoja. Kummassakin tapauksessa käyttäjä painaa raporttinappia, joka käynnistää PDF-raporttien generoinnista vastaavan Meteorin metodin.

Metodin käynnistyessä kannasta haetaan asiakkaan tiedot ja tulokset. Nämä tiedot käytetään laskentavaiheen läpi viitearvolaskureissa. Laskureista palautuvat tulokset otetaan talteen raporttia varten. Raportti koostetaan dynaamisesti tuloksia käyttäen, eli tulokset asetellaan raporttiin niille kuuluville paikoille. Kun koostaminen on valmis, Webshot (ja sen sisällä toimiva PhantomJS) renderöi koostetun sivun ja ottaa siitä asetuksineen kuvankaappauksen. Ja näin PDF-raportti on valmis. PDF enkoodataan base64-merkkijonoksi, jotta se voidaan lähettää palvelimelta asiakaspuolelle. Asiakaspuolella eli käyttäjän päässä se puretaan alkuperäiseen muotoonsa, eli käsitellään base64-merkkijonosta takaisin PDF-muotoon. Tämän jälkeen PDF joko ladataan suoraan käyttäjän laitteelle, tai se avataan selaimessa uudessa ikkunassa. Tämä riippuu laitteesta, käyttöjärjestelmästä ja käytetystä selaimesta. Liitteet 1 ja 2 ovat esimerkkejä kuntotestiraporteista.

5.6 Reactin käyttö

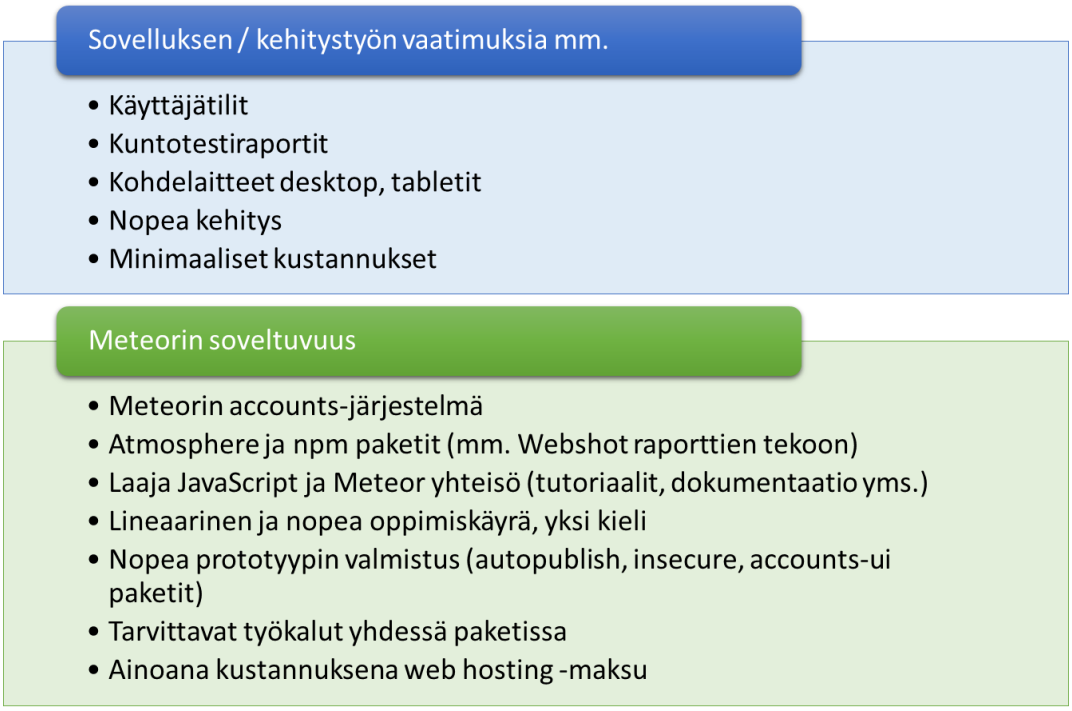
Pääsyy Reactin valinnalle käyttöliittymäkerroksessa oli ennen kaikkea kiinnostus kirjastoa kohtaan ja siitä kokemuksen saaminen. Angular oli tuttu peruskäsitteiltään, mutta se on kokonaisvaltaisempi sovelluskehys, joka tuntui olevan ehkä turhankin laaja käytettäväksi kehitettyyn sovellukseen. Blazeakin kokeililtiin demovaiheessa, mutta React päätyi kuitenkin lopulta valinnaksi käyttöliittymän koodaukseen. Blazea tosin hyödynnettiin palvelinpuolella raportin koostamisessa.

React ei ollut kovin tuttu entuudestaan, mutta demovaiheessa totesin sen olevan hyvin mielenkiintoinen ja toimiva lähestymistapa käyttöliittymien koodaukseen. Pidin eritoten sen komponenttimaisuudesta, joka sopi myös hyvin yhteen Meteor-ympäristön modulaariseen koodaustapaan. JSX syntaksi oli aluksi outo, mutta se osoittautui hyvin nopeasti erinomaiseksi tavaksi tehdä siistiä ja selvää koodia komponenteissa, joissa pystyttiin määrittelmään samalla myös HTML-rakenteita. React-komponenttien tilan ohjailu osoittautui hyväksi tavaksi käsitellä käyttäjän valintoja sovelluksen käyttöliittymässä, mikä koostuu napeistä, valintaruuduista ja muista interaktiivisista osista.

6 Pohdinta

6.1 Johtopäätökset

Kehitystyön alku oli siinä mielessä kysymerkkejä täynnä, että itselläni ei ollut aikaisempaa kokoemusta kokonaisen sovelluksen tekemisestä. Olin kuitenkin jokseenkin tietoinen erilaisista toteutustavoista. Valinta käytetyistä tekniikoista piti tehdä melko nopeasti, sillä sovelluksen toteutukselle oli varattu rajallinen aika. Alusta asti pidin selvänä sitä, että sovellus olisi hyvä toteuttaa moderneja JavaScript-tekniikoita käyttäen, joihin web-sovellusten kehitys nojaa hyvin paljon tänä päivänä. Olin tietoinen JavaScript-yhteisön laajuudesta ja sen tarjoamista paketeista ja moduuleista, jotka tulisivat varmasti hyödyllisiksi sovellusta kehittäessä. Oma kiinnostukseni modernia web-kehitystä kohtaan toimi myös hyvänä motivaattorina. Alussa tutkin mm. MEAN-pinoon pohjautuvia toteutusvälineitä, kuten Mean.io ja MeanJS. Kuitenkin lyhyen perehtymisen jälkeen totesin, että niihin perehtyminen syvemmin olisi vienyt enemmän aikaa, jota ei käytössä kuitenkaan ollut. Meteorin tutoriaaleja tehdessäni ja sen dokumentaatiota tutkiessani totesin hyvin nopeasti, että se soveltuisi sovelluksen tekoon moneltakin osin. Kuvio 18 esittää lyhyen listauksen projektin vaatimuksista ja Meteorin soveltuvuudesta kehitettyyn sovellukseen.



Kuvio 18. Meteorin soveltuvuus

Meteorin todella yksinkertaisen ekosysteemin ansiosta kehitystyö lähti todella helposti käyntiin vain lyhyen perehtymisen jälkeen. Prototyypin valmistaminen oli äärimmäisen nopeaa, joten pystyin esittelämään ideaani hyvin nopeasti demovaiheessa. Vaatimuksiin perustuvien ominaisuuksien (käyttäjätilit, raportit yms, kuvio 18) toteuttaminen nollapisteestä olisi vienyt huomattavan paljon aikaa, varsinkin kun aikaisempaa kokemusta ei ollut. Erinomainen Atmosphere-pakettijärjestelmä yhdessä npm-moduulien kanssa tarjosivat valtavan määrän rakenneosia, joilla pystyttiin nopeuttamaan kehitystyötä merkittävästi. Meteorin palvelinpuolen ja asiakaspuolen koodaus on nidottu erinomaisella tavalla yhteen, eikä kehittäjän tarvitse olla mitenkään erityisen back-end- tai front-end-orientoitunut pystyäkseen tekemään molempia puolia. Etenkin, kun kieltä ei tarvitse vaihtaa. DDP:n konsepti oli nopeasti ymmärrettävissä, ja dataliikenteen käsittely kantaoperaatioineen onnistui käden käänteessä. Lähes 36 000 tähteä Githubissa (25.10.2016), valtava ja aktiivinen yhteisö tutoriaalineen, dokumentaatioineen ja paketteineen puhuvat paljon Meteorin puolesta. Suurimpina argumenttina Meteorin valikoitumiseen olikin sen pieni oppimiskynnys ja mahdollisuus nopeampaan kehitykseen, sillä lähes kaikki työkalut löytyivät samasta paketista. Kokonaisvaltaisen

sovelluksen kehittäminen onnistui yhdellä kielellä, yksinkertaisesti, ja ennen kaikkea nopeasti.

Vaikka Meteor onkin tehokas alusta, joka tarjoaa ennennäkemättömän ja samalla hauskan kehityskokemuksen, ei se silti ole välttämättä oikea valinta. Se sopi kehitettyyn sovellukseen hyvin, mutta se ei ole aina paras mahdollinen väline. Meteor ei välttämättä sovellu monien erilaisten tietokantojen kanssa käytettäväksi. Tällä hetkellä varsinkin SQL-kannat eivät oikein sovellu Meteorin kanssa yhteen, vaikka sitäkin onglemaa varten on kirjoitettu joitain yhteisön paketteja. Meteor ei siis ole välttämättä se soveltuvin vaihtoehto, jos esimerkiksi projektin tietokantapuoli on toteutettu relaatiokantana. Meteor-sovelluksen palvelinpuoli voi myös käydä valtavilla käyttäjämäärillä raskaammaksi moneen muuhun back-end -tekniikkaan verrattuna. Meteor on kokonaisvaltainen alusta kehittämiseen, mutta samalle se voi asettaa myös joitain rajoitteita kehittäjälle. Kehitettävä kohde voi olla esimerkiksi hyvin omanlaatuinen, mikä vaatii erilaisen lähestymistavan kehitystyöhön. Tällöin enemmän minimalistiset sovelluskehykset, kuten esimerkiksi Express tai Koa vosivat sopia paremmin toteutusvälineiksi. Niillä kehittäjä pystyy itse vaikuttamaan paremmin siihen, miten sovellus rakentuu sen alkuhetkistä lähtien. Mainitut esimerkit ovat Meteorin tavoin Noden päälle rakennettuja sovelluskehyksiä, jotka eivät kuitenkaan tarjoa täysin käyttövalmista kokonaisuutta Meteoriin verrattuna. Meteorin koodi on myös lähes poikkeuksetta synkronista, joten asynkroniseen koodaustapaan tottuneet huomaavat, että kaikkien toimintojen etenemistä ei voidakaan tarkoin ohjata esim. callback-funktioita käyttämällä.

JavaScriptin ja Noden ympärillä on valtava määrä erilaisia toteutusvälineitä sovellusten tekoon. Toteutusvälineitä valittaessa tulisikin miettiä aina tarkkaan määritelmiä ja mahdollisia rajoitteita joita projekteihin liittyy. Kaikki välineet eivät sovellu kaikkiin tilanteisiin.

6.2 Kehitysmahdollisuudet ja tulevaisuus

Sovelluksessa on toki parantamisen varaa ja kehitysmahdollisuuksia. Sovelluksen responsiivisuus ulottuu tableteille (iPad ja Android-tabletilla testattu), mutta puhelimen näytöille se ei sovellu. Responsiivisuus kaikilla laitteilla ei tosin

kuulunutkaan sovelluksen määrittelyihin. Sovelluksen käyttöliittymä on muutamissa näkymissä suhteellisen monimutkainen monine valintamahdollisuuksineen ja interaktiivisin toimintoineen, joten sen mahduttaminen tiiviimpään muotoon voisi olla haastavaa. Yksi ratkaisu olisi kehittää puhelimia varten kokonaan oma, irrallinen mobiilikäyttöliittymä, mikä olisi yksinkertaistetumpi muoto nykyisestä. Mobiiliversiota voisi käyttää vaikka asiakkaiden ja tulosten nopeaan lisäämiseen. Sovelluksen ulkonäköäkin voisi myös varmasti kohentaa. Käyttöliittymä on nopeasti suunniteltu ja sen ulkonäöllisiin seikkoihin on käytetty melko tavanomaisia Bootsrap kirjaston ominaisuuksia. Yksinkertaisuuteen kuitenkin tähdättiin ja käyttöliittymä hyväksyttiin sellaisenaan kun se nyt on.

Aikaa laajemmalle testausvaiheelle ei ollut nopean kehitystahdin johdosta, joten mahdolliset ongelmat tulivat vastaan aika sattumanavaraisesti kehitystyötä tehdessä. Toki kaikki vastaan tulleet ongelmat korjattiin, mutta mahdollisia bugeja voi vielä sovelluksessa piiletä. Pienen “betatestauksen” jälkeen saatiin kuitenkin jotain palautetta muutamilta valmentajilta, ja heidän osoittamat ongelmat ja muutosehdotukset katsottiin kyllä kuntoon.

Sovelluksen tulevaisuudelta toivon, että sen käyttöaste ei jäisi pelkän testaamisen tasolle. Sovellus on melko uniikki, enkä usko että Suomesta löytyy ainakaan tällä hetkellä mitään vastaavanlaista. Oma tieni sovelluksen kehityksen parissa on päättynyt, vaikkakin olen luvannut olla konsultoitavissa tarpeen tullen. Kuitenkin se, mitä rajallisessa aikataulussa oli saavutettavissa, saavutettiin. Käytössä on toimiva web-sovellus, joka vastaa hyvin pitkälti sitä kuvaavia määrittelyitä.

Lähteet

About Node.js®. N.d. Viitattu 29.9.2016. https://nodejs.org/en/about/.

accounts-base. N.d. Viitattu 11.10.2016. https://guide.meteor.com/accounts.html#accounts-base.

accounts-password. N.d. Viitattu 11.10.2016. https://atmospherejs.com/meteor/accounts-password.

alanning:roles. N.d. Viitattu 11.10.2016. https://atmospherejs.com/alanning/roles.

Atmosphere vs. npm. N.d. Viitattu 3.10.2016. https://guide.meteor.com/atmosphere-vs-npm.html.

Defining a schema. N.d. Viitattu 11.10.2016. https://guide.meteor.com/collections.html#schemas.

ES2015 modules. N.d. Viitattu 3.10.2016. https://guide.meteor.com/structure.html#es2015-modules.

ES2015+ (recommended). N.d. Viitattu 3.10.2016. https://guide.meteor.com/build-tool.html#es2015.

Gackenheimer, C. 2015. Introduction to React. Appress. Viitattu 6.10.2016. https://janet.finna.fi/, Books24x7.

Goldshtein, U. 2015. Creating realtime apps with Angular 2 and Meteor. Video. Viitattu 5.10.2016. https://www.youtube.com/watch?v=3FT0BqYASCo.

Haverbeke, M. 2015. Eloquent JavaScript: A Modern Introduction to Programming, 2nd Edition. No Starch Press. Viitattu 29.9.2016. https://janet.finna.fi/, Books24x7.

Hovi, A. 2015. MongoDB haastaa relaatiokantoja. Artikkeli. Viitattu 3.10.2016. http://www.arihovi.com/mongodb-haastaa-relaatiokantoja/.

Introduction to DDP. 2014. Viitattu 4.10.2016. https://meteorhacks.com/introduction-to-ddp/.

Kananen, J. 2012. Kehittämistutkimus opinnäytetyönä. Kehittämistutkimuksen kirjoittamisen käytönnön opas. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Krajka, B. 2015. The Difference between Virtual DOM and DOM. Viitattu 6.10.2016. http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/.

Minnick, C. & Holland, E. 2015. Coding with JavaScript For Dummies. John Wiley & Sons. Viitattu 29.9.2016. https://janet.finna.fi/, Books24x7.

Mobile. N.d. Viitattu 5.10.2016. https://guide.meteor.com/ui-ux.html#mobile.

MongoDB collections in Meteor. N.d. Viitattu 4.10.2016. https://guide.meteor.com/collections.html#mongo-collections.

meteorhacks:ssr. N.d. Viitattu 20.10.2016. https://github.com/meteorhacks/meteor-ssr.

node-webshot. N.d. Viitattu 20.10.2016. https://github.com/brenden/node-webshot.

Optimistic UI. N.d. Viitattu 4.10.2016. https://guide.meteor.com/ui-ux.html#optimistic-ui.

Passwords. N.d. Viitattu 11.10.2016. http://docs.meteor.com/api/passwords.html.

Publications and subscriptions. N.d. Viitattu 4.10.2016. https://guide.meteor.com/data-loading.html#publications-and-subscriptions.

Robinson, J., Gray, A. & Titarenco, D. 2015. Introducing Meteor. Appress. Viitattu 30.9.2016. https://janet.finna.fi/, Books24x7.

Salonen, J. 2012. Johdanto JavaScript-sovellusten kehitykseen Node.js:llä. Päivitetty 17.9.2012. Viitattu 29.9.2016. http://blite.iki.fi/artikkelit/javascript-nodejs-johdanto/.

Screen Capture. N.d. Viitattu 20.10.2016. http://phantomjs.org/screen-capture.html.

Special Directories. N.d. Viitattu 4.10.2016. https://guide.meteor.com/structure.html#special-directories.

Stubailo, S. 2015. Optimistic UI with Meteor. Viitattu 4.10.2016. http://info.meteor.com/blog/optimistic-ui-with-meteor-latency-compensation.

Thinking in React. N.d. Viitattu 7.10.2016. https://facebook.github.io/react/docs/thinking-in-react.html.

Use the ecmascript package. N.d. Viitattu 3.10.2016. https://guide.meteor.com/code-style.html#ecmascript.

Using schemas on write. N.d. Viitattu 11.10.2016. https://guide.meteor.com/collections.html#schemas-on-write.

View layers. N.d. Viitattu 5.10.2016. https://guide.meteor.com/ui-ux.html#view-layers.

What is a Method? N.d. Viitattu 4.10.2016. https://guide.meteor.com/methods.html#what-is-a-method.

What is Meteor? N.d. Viitattu 29.9.2016. https://docs.meteor.com/#what-is-meteor.

What is npm? N.d. Päivitetty 18.7.2016. Viitattu 29.9.2016. https://docs.npmjs.com/getting-started/what-is-npm.

Liitteet

Liite 1. Esimerkki 1 kuntotestiraportista.

TRAINER YOU

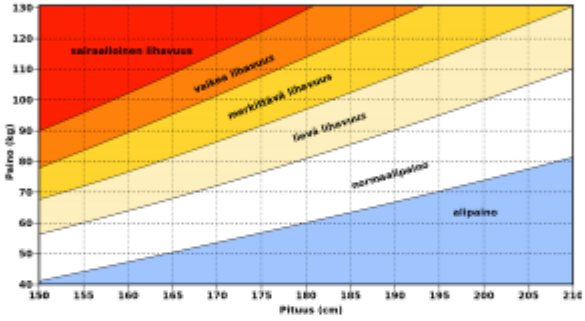
Antti Asiakas 3.11.2016

Painoindeksi

Painoindeksi eli BMI (Body Mass Index) kertoo pituuden ja painon suhteesta. Tulos saadaan kaavalla: paino (kiloina) / pituus x pituus (metreinä). Painoindeksiä 18,5 - 24,99 pidetään normaalina

Mittauspäivämäärä: 3.11.2016 Paino: 75 kg - BMI: 26.3 (lievä lihavuus)

Mittauspäivämäärä: 3.11.2016 Paino: 65 kg - BMI: 22.8 (normaali paino)



Viitearvojen lähde: WHO

Verenpaine

Verenpaine kuvaa sitä työtaakkaa, jonka sydän joutuu tekemään ympäri vuorokauden. Normaalin lepoverenpaineen ylärajana pidetään aikuisilla ihmisillä 129 / 84 mmHg.

Mittauspäivämäärä: 3.11.2016 Tulos: 150/40 mmHg (lievästi kohonnut/alhainen paine)

Mittauspäivämäärä: 3.11.2016 Tulos: 125/75 mmHg (normaali/ihannearvo)

Jaottelu	Systolinen		Diastolinen
Optimaalinen	< 120	ja	< 80
Normaali	120-129	ja/tai	80-84
Tyydyttävä *(high normal)*	130-139	ja/tai	85-89
Asteen 1 hypertensio *(lievä)*	140-159	ja/tai	90–99
Asteen 2 hypertensio *(keskivaikea)*	160-179	ja/tai	100-109
Asteen 3 hypertensio *(vaikea)*	≥ 180	ja/tai	≥ 110

Liite 2. Esimerkki 2 kuntotestiraportista.

TRAINER YOU

Antti Asiakas 3.11.2016

Cooper

Cooper-testi on testi, jolla mitataan maksimikestävyyttä. Sen kehitti yhdysvaltalainen tohtori Kenneth H. Cooper 1968 Yhdysvaltain armeijalle. Ideana testissä on edetä 12 minuutissa niin pitkälle kuin pääsee. Cooper-testi korreloi väestötasolla hyvin maksimaalisen hapenottokyvyn (VO2max) kanssa.

Mittauspäivämäärä: 3.11.2016 Tulos: 3000 metriä - erittäin hyvä

erittäin huono	huono	keskiverto	hyvä	erittäin hyvä
-1600 m	1600 - 2199 m	2200 - 2399 m	2400 - 2800 m	2800+ m

Viitearvojen lähde: Cooper, K. H. (1968) A means of assessing maximal oxygen uptake. Journal of the American Medical Association 203:201-204.

Vatsalihakset: 30 sekunnin toistotesti

Testin tarkoitus on testata vartalon koukistajalihasten dynaamista kestävyyttä. Testattava asettuu selin makuulle siten, että polvi- ja nilkkakulma on 90 astetta. Kädet ovat niskan takana sormet lomittain ja testaaja tukee nilkoista. Ylävartalo nostetaan pystysuoraan mahdollisimman monta kertaa 30 sekunnin aikana siten, että kyynärpäät koskettavat polvia ja alas tullessa hartiat ja pää koskettavat alustaa. Suoritus alkaa testaajan "nyt" komennolla ja päättyy testaajan ilmoittaessa suoritusajan päättymisestä.

Mittauspäivämäärä: 3.11.2016 Tulos: 20 - keskinkertainen

heikko	välttävä	keskinkertainen	hyvä	erinomainen
≤ 16	17-18	19-23	24-28	≥29

Viitearvojen lähde: Liite ry, 1991

Vatsalihasten toistotesti

Testin tarkoitus on testata vartalon koukistajalihasten dynaamista kestävyyttä. Testattava asettuu selin makuulle, polvet 90° kulmassa ja jalkapohjat alustalla. Kädet ovat kämmenet alaspäin reisien päällä. Testattava kurkottaa käsillään kohti polviaan ja nousee istumaan niin pitkälle, että ranteiden