



TAMPEREEN TEKNILLINEN YLIOPISTO

TOMMI LEINAMO
REAKTIIVISEN WEB-SOVELLUKSEN TOTEUTTAMINEN
Diplomityö

Tarkastaja: Professori Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston kokouksessa
9. huhtikuuta 2014

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

LEINAMO, TOMMI: Reaktiivisen web-sovelluksen toteuttaminen

Diplomityö, 44 sivua

Huhtikuu 2015

Pääaine: Ohjelmistotuotanto

Tarkastaja: Professori Tommi Mikkonen

Avainsanat: web-sovellukset, reaktiivisuus, reaktiivinen ohjelmointi, Meteor

Nykyaikaisissa web-sovelluksissa pyritään usein välttämään sivun uudelleenlatauksia sovelluksen tilan päivittymisen yhteydessä. Koko sivun lataaminen uudelleen on hidasta ja se keskeyttää web-sovelluksen käytön. Niinpä web-sovelluksissa suositetaan nykyisen näkymän dynaamista päivittämistä koko sivun uudelleenlataamisen sijaan.

Web-sovellukseen liittyy dynaamisuudesta huolimatta datan päivittymisen ongelma. Jos näkymässä oleva data muuttuu näkymän luomisen jälkeen, ei datan muutos vaikuta sovelluksen tilaan. Tämä ongelma voidaan ratkaista tekemällä sovelluksesta reaktiivinen. Reaktiivisessa web-sovelluksessa käyttöliittymä päivittyy automaattisesti, kun sen sisältämä data muuttuu.

Tässä diplomityössä arvioidaan reaktiivisen web-sovelluksen kehitysprosessin kautta, mitä ongelmia reaktiivisuuden hyödyntämiseen web-sovelluksissa liittyy, ja mitä hyötyjä sillä voidaan saavuttaa. Työssä esitellään arvioinnin pohjana oleva Vincit Oy:ssä kehitetty web-sovellus sekä siinä hyödynnetty reaktiivisuutta tukeva Meteor-sovelluskehys.

Työssä todetaan, että vaikka reaktiivisuuden hyödyntämisellä voidaan saavuttaa huomattavia hyötyjä kuten sovelluksen käyttökokemuksen parantaminen ja sovelluskehityksen helpottaminen, hyötyjen suuruus riippuu paljolti sovelluksen käyttökohteesta. Reaktiivisuus voi myös aiheuttaa ongelmia muun muassa ohjelman suorituskyvyn kanssa. Vaikka Meteor-sovelluskehityksen reaktiivinen malli on toimiva, se asettaa sovelluksen arkkitehtuurille rajoitteita tietokannan suhteen. Reaktiivisuuden hyödyntämisen tarpeellisuutta tulisi arvioida ottamalla sovelluksen loppukäyttäjien tarpeet huomioon.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Master's Degree Programme in Information Technology

LEINAMO, TOMMI: Developing a reactive web application

Master of Science Thesis, 44 pages

April 2015

Major: Software engineering

Examiner: Professor Tommi Mikkonen

Keywords: web applications, reactivity, reactive programming, Meteor

In modern web applications it is usually preferable to avoid full page reloads when the state of the application is updated. Reloading the page is slow and it interrupts the usage of the application. Because of this updating the view dynamically is preferred to reloading the page.

Despite the dynamic updating of the view web applications have a problem with data updates. If the data that the view contains changes after the view has been initialized, the change does not affect application's state. This problem can be solved by making the web application reactive. In a reactive web application the view is updated automatically when the data displayed in it changes.

In this thesis the development process of a reactive web application will be used to evaluate the problems caused by applying reactive model to a web application and what advantages it might achieve. The thesis outlines the web application which was developed in Vincit Ltd and which is used as basis for the evaluation in this thesis. Also, the Meteor framework used in the web application will be outlined.

The conclusion of this thesis is that even if utilizing reactivity in a web application can achieve considerable advantages such as user experience improvement and facilitation of application's development, the extent of these advantages is largely dependent on how the application is used. Reactivity can also cause problems with e.g. the application's performance. While a working solution, the reactivity model used by Meteor places some constraints on the application's architecture in terms of the used database. It is recommended that the necessity for using reactivity should be evaluated based on the needs of the end users of the application.

ALKUSANAT

Tämä diplomityö on syntynyt Vincit Oy:n Summer@Vincity-hankkeessa toteutetun web-sovelluksen pohjalta. Hankkeessa toteutettu web-sovellus sekä sen kehitysprosessin aikana kerätyt kokemukset toimivat pohjana tässä diplomityössä tehdyille arvioinnille.

Haluan kiittää diplomityön tarkastajaa professori Tommi Mikkosta hänen rakentavasta palautteestaan, joka ohjasi diplomityötä oikeaan suuntaan ja kannusti sen saattamisessa loppuun. Kiitokset työtovereilleni Anssi Kuutille, joka toimi diplomityön ohjaajana ja oli valmis avustamaan aina tarvittaessa, Jari Huillalle avusta sovelluksen toteutuksen aikana, sekä Olli Sallille diplomityön tekemiseen liittyneistä neuvoista. Kiitokset myös Vincitille joka loistavana työnantajana tarjosi mahdollisuudet tämän diplomityön tekemisen. Lisäksi erittäin suuret kiitokset myös muille Summer@Vincity-hankkeeseen osallistuneille sovelluskehittäjille – Lasse Liehu, Janne Koski ja Aleks Grön – sekä kaikille muille hankkeeseen vaikuttaneille.

Tampereella 22.3.2015

Tommi Leinamo

SISÄLLYS

1. Johdanto	1
2. Reaktiivisuus	3
2.1 Määritelmä	3
2.2 Reaktiivinen ohjelmointi	4
2.2.1 Muutoksen levittäminen	5
2.2.2 Toteutustapoja	5
2.2.3 Funktionaalinen reaktiivinen ohjelmointi	6
2.3 Sovelluskohteita	6
2.3.1 Reaaliaikajärjestelmät	6
2.3.2 Taulukkolaskenta	7
2.3.3 Käyttöliittymät	7
2.3.4 Web-sovellukset	7
3. Taustaa	8
3.1 Web-sovellus	8
3.2 Arkkitehtuureista	8
3.2.1 Asiakas-palvelin-malli	9
3.2.2 MVC-arkkitehtuuri	9
3.3 Yhden sivun sovellukset	10
3.4 Responsiivisuus	11
3.5 Koostepalvelut ja joukkoistaminen	11
3.6 Reaktiivisuus web-sovelluksissa	12
4. Meteor-sovelluskehys	13
4.1 Taustaa	13
4.1.1 MongoDB	13
4.1.2 Node.js	14
4.1.3 WebSocket	14
4.1.4 Handlebars	15
4.2 Yleiskuvaus	15
4.2.1 Arkkitehtuurista	16
4.2.2 Palvelin ja tietokanta	17
4.2.3 Asiakassovelluksen näkymät	17
4.2.4 Minimongo	18
4.2.5 Asiakkaan ja palvelimen välinen dataliikenne	18
4.3 Reaktiiviset ominaisuudet	19
4.3.1 Meteor Tracker	19
4.3.2 Näkymien reaktiivisuus	21
4.3.3 DDP-protokolla	22

4.3.4	Livequery	23
5.	Sovelluksen esittely	24
5.1	Käyttötarkoitus ja ominaisuudet	24
5.1.1	Projektin tausta	24
5.1.2	Käyttöliittymä	25
5.1.3	Kohteiden tiedot	25
5.1.4	Kohteiden hakumahdollisuudet	26
5.1.5	Kohdedatan kerääminen	27
5.1.6	Ylläpitäjien työkalut	28
5.2	Sovelluksen arkkitehtuuri	28
5.2.1	Tietomalli	29
5.2.2	Kartta ja kohteiden esittäminen	30
5.2.3	Näkymät ja sivupohjat	31
5.2.4	Reititys	32
5.2.5	Kuvat	33
5.2.6	Avoimet datalähteet	33
6.	Toteutusprosessi	35
6.1	Projektin eteneminen	35
6.2	Testaus	37
6.3	Työtavat	37
7.	Arviointi	39
7.1	Projektin onnistuminen	39
7.2	Ongelmia	39
7.2.1	Suurten datamäärien hallinta	40
7.2.2	Sirrettävän datan hallinta	41
7.2.3	Asiakassovelluksen suorituskyky	41
7.2.4	Meteorin reaktiivisen mallin rajoitteet	42
7.3	Reaktiivisuudella saavutetut hyödyt	42
7.3.1	Sovelluksen käyttökokemus	42
7.3.2	Sovelluskehityksen helpottuminen	43
8.	Yhteenveto	44
	Lähteet	45

1. JOHDANTO

Web-sivujen luonne on muuttunut internetin alkua ajoista merkittävästi. Aikaisemmin web-sivut olivat tyypillisesti yksinkertaisia ja staattisia: käyttäjän navigoidessa web-sivun URL-osoitteeseen selain teki palvelimelle HTTP-pyyynnön, johon saatiin vastauksena HTML-sivu, jonka selain näytti käyttäjälle sellaisenaan. Tässä mallissa on yleistä, että koko sivu joudutaan lataamaan uudestaan aina, kun käyttäjä navigoi verkkopalvelussa uuteen näkymään esimerkiksi seuraamalla sivulla näkyvää linkkiä. Käyttäjälle tämä näkyy pitkinä latausaikoina ja jatkuvana selauskokemuksen keskeytymisenä.

Viime vuosina ohjelmoinnin hyödyntäminen web-sivujen toteutuksessa on kuitenkin yleistynyt, ja dynaamiset web-sovellukset ovat kasvattaneet osuuttaan web-sivuista. Web-sovellukset hyödyntävät selaimen rajapintoja ja web-sivuihin liitettäviä JavaScript-ohjelmia, joiden avulla osa sivusta pystytään generoimaan vasta selaimessa. Näin voidaan jopa kokonaan välttää sivun uudelleenlatauksen tarve siirryttäessä näkymästä toiseen.

Vaikka web-sovelluksesta olisi toteutettu dynaaminen eikä sivua tarvitsisi ladata uudelleen näkymän vaihtuessa, siinä esiintyy yleensä siitä huolimatta toinen ongelma. Kun web-sovellus generoi näkymän, se käyttää siihen sillä hetkellä saatavilla olevaa dataa. Yleensä tämä data haetaan palvelimelta. Jos data muuttuu palvelimella sen jälkeen kun näkymä on generoitu, ei web-sovellus saa muutoksesta tietoa ennen kuin data haetaan uudelleen, mitä ei yleensä tehdä kuin näkymää generoitaessa. Tiedon välitystä datan muuttumisesta vaikeuttaa monesti myös se, että palvelin yleensä lähettää asiakkaalle dataa ainoastaan kun asiakas tekee pyynnön eikä omasta aloitteestaan.

Käyttäjän kannalta olisi yksinkertaisempaa, jos uusi data tulisi sivulle näkyviin automaattisesti. Tieto olisi käyttäjän saatavilla heti, eikä sivua tai näkymää tarvitsisi ladata uudelleen muutoksen havaitsemiseksi. Tällaista ominaisuutta nimitetään reaktiivisuudeksi, ja web-sovellusten yhteydessä se tarkoittaa yleensä näkymän automaattista päivittymistä siinä esitetyn datan muuttuessa.

Reaktiivisuuden hyödyllisyydestä ja tarpeesta kertoo myös sen toteuttamiseen keskittyvien sovelluskehysten kasvava määrä. Yksi tällainen sovelluskehys on Meteor, joka toteuttaa web-sovelluksessa rungon sekä asiakas- että palvelinsovelluksen toteutukselle ja painottaa reaktiivisuuden tukemista vahvasti. Sovelluskehystä

hyödyntämällä voidaan mahdollisesti säästää huomattavasti aikaa toteutuksessa, ja reaktiivisuudesta voi olla esimerkiksi sovelluksen käyttökokemuksen kannalta merkittävää hyötyä.

Tässä diplomityössä käsitellään reaktiivisen web-sovelluksen kehittämistä, sekä web-sovellusten reaktiivisuuteen ja reaktiiviseen ohjelmointiin liittyviä ilmiöitä ja haasteita. Tarkastelun pohjana käytetään yhdessä Vincit Oy:n projektissa toteutettua web-sovellusta, joka toteutettiin Meteor-sovelluskehysellä. Työssä pyritään tunnistamaan mahdollisia reaktiivisuuden käyttöön liittyviä ongelmia ja arvioidaan reaktiivisuudella saavutettuja hyötyjä.

Tämän diplomityön rakenne on seuraavanlainen. Luvussa 2 käsitellään reaktiivisuutta ja reaktiivisen ohjelmoinnin paradigmaa. Luvussa 3 käydään läpi diplomityön kannalta oleellinen web-sovelluksiin liittyvä teoria. Luvussa 4 esitellään Meteor-sovelluskehysten taustateknologiat, arkkitehtuuri ja reaktiiviset ominaisuudet. Luvussa 5 kuvataan projektissa toteutettu web-sovellus, joka toimii arvioinnin pohjana. Luvussa 6 käsitellään projektin etenemistä sekä siinä sovellettuja työmenetelmiä. Luvussa 7 arvioidaan projektin onnistumista, projektissa havaittuja ongelmia ja niihin sovellettuja ratkaisuja, sekä reaktiivisuudella saavutettuja hyötyjä. Luku 8 sisältää yhteenvedon arvioinnista sekä siitä vedettävät johtopäätökset.

2. REAKTIIVISUUS

Macmillan-sanakirjan määritelmän mukaan sana *reaktiivinen* (reactive) kuvaa jotakin joka reagoi tapahtuviin asioihin sen sijaan että tekisi asioita oma-aloitteisesti [32]. Merriam-Webster-sanakirja sisällyttää adjektiivin ”reaktiivinen” tarkoittamaan myös nopeasti ärsykkeisiin reagoivaa [34]. Vastakohtana tälle on proaktiivinen eli ennakkoiva: asioita ja muutoksia tehdään ennen kuin niitä välttämättä tarvitsee tehdä, sen sijaan että odotettaisiin asioiden tai ongelmien tapahtuvan.

Ohjelmistoissa reaktiivisuudella viitataan tyypillisesti ohjelman tapaan reagoida syötteisiin ja datan muutoksiin. Tässä luvussa esitellään reaktiivisuus ohjelmistojen näkökulmasta sekä siihen läheisesti liittyvä reaktiivisen ohjelmoinnin paradigma. Reaktiivisuuden käsite määritellään kohdassa 2.1, reaktiivista ohjelmointia käsitellään kohdassa 2.2, ja kohdassa 2.3 annetaan esimerkkejä sovellusalueista jossa reaktiivisuutta on hyödynnetty.

2.1 Määritelmä

Tietotekniikan tieteellisessä kirjallisuudessa termi ”reaktiivisuus” esiintyi ensimmäisen kerran vuonna 1985. Harel ja Pnueli jakoivat järjestelmät *muuttaviin* (transformational) ja reaktiivisiin järjestelmiin. Tässä yhteydessä järjestelmällä ei tarkoiteta ainoastaan ohjelmistoja, vaan myös laitteistopohjaisia- ja sulautettuja järjestelmiä. Heidän jaossaan muuttavat järjestelmät ottavat vastaan syötteitä, tekevät niille muutoksia eli suorittavat operaatioita, ja tuottavat tulosteen. Reaktiiviset järjestelmät sen sijaan vastaanottavat toistuvasti pyyntöjä ympäristöltään ja niiden tehtävänä on vastata jatkuvasti ulkoisiin syötteisiin. [24]

Ohjelmistoihin keskittyneen määritelmän esitti Berry, joka jakoi ohjelmat muuttaviin, interaktiivisiin ja reaktiivisiin. Hänen mukaansa reaktiiviset ohjelmat suorittavat laskentaa vastatakseen ulkoisiin pyyntöihin ja ovat jatkuvassa yhteydessä ympäristöönsä. Vuorovaikutuksen nopeuden määrää kuitenkin ympäristö, toisin kuin interaktiivisilla ohjelmilla jotka ohjaavat itse suorituksen nopeutta. Tämä on seurausta siitä, että reaktiiviset ohjelmat pääsääntöisesti aloittavat toimintansa vain reaktionä ulkopuolisiin syötteisiin. Reaktiivisten ohjelmien käsittelemien syötteiden ja tulosteiden voidaan myös katsoa olevan jatkuvia virtoja. [5; 59]

Reaktiivisten- ja reaaliaikaohjelmien välillä on paljon yhtäläisyyksiä. Monet reaaliaikaohjelmat voidaan lukea reaktiivisiksi [5], sillä ne ovat tyypillisesti jatkuvassa

yhteydessä ympäristöönsä. Berryn mukaan erona reaktiivisten- ja reaaliaikaohjelmien välillä on reaaliaikaohjelmille asetetut aikarajoitteet [3].

Edellä mainituista määritelmistä voidaan reaktiivisille ohjelmille tunnistaa seuraavat tunnusomaiset piirteet:

- jatkuva vuorovaikutus ympäristönsä kanssa
- kommunikointi ympäristön määräämällä tahdilla
- ei tiukkoja reaaliaikavaatimuksia
- toiminta muodostuu reagoinnista ympäristön antamiin syötteisiin.

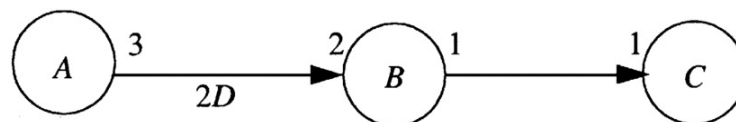
2.2 Reaktiivinen ohjelmointi

Reaktiivinen ohjelmointi on ohjelmointiparadigma, jota käyttämällä saadaan ohjelmiin tuotua reaktiivisia piirteitä. Se perustuu *synkronisen tietovuo-ohjelmoinnin* (synchronous data flow, SDF) paradigmaan, joka nimensä mukaisesti yhdistelee synkronista ja tietovuo-ohjelmointia [2]. SDF-paradigma on alun perin kehitetty digitaalisen signaalinkäsittelyn algoritmien kuvaamiseen [30].

Synkronisissa kielissä aika on jaettu diskreetteihin hetkiin, joissa tapahtuvien reaktioiden oletetaan yksinkertaistetusti olevan atomisia ja tapahtuvan välittömästi [2; 4]. Tietovuo-ohjelmoinnissa ohjelma mallinnetaan suunnattuna graafina jossa solmut ovat ohjelman operaatioita ja kaaret kuvaavat datan liikkumista [58]. Data liikkuu operaatioiden välillä *datanäytteinä* (data token, data sample). Operaatiot voivat tapahtua aina kun datanäytteitä on saatavilla, mistä johtuen tietovuo-ohjelmoinnin malli on pohjimmiltaan rinnakkainen [30; 58].

Leen ja Messerschmittin SDF-mallissa operaatioiden kuluttamien ja tuottamien datanäytteiden määrä voi vaihdella operaatioiden välillä ja on ennalta tiedossa. Tämän ansiosta mallin aikataulutusta voidaan tehdä staattisesti eikä tarvitse dynaamisuutta, toisin kuin ei-synkronisessa mallissa jossa aikataulutusta hallitsee sisääntuleva data. [30]

Kuvassa 2.1 on esitetty yksinkertainen SDF-graafi, jossa kunkin operaation kuluttamien ja tuottamien datayksiköiden määrä on ilmaistu kaarien päällä olevin numeroin. Kaaren (A, B) alapuolelle merkitty $2D$ kuvaa viivettä.



Kuva 2.1: Yksinkertainen SDF-graafi [6].

2.2.1 Muutoksen levittäminen

Reaktiivisen ohjelmoinnin tärkeimpiä käsitteitä ovat aikariippuvaiset muuttujat ja *muutoksen levittäminen* (propagation of change). Aikariippuvaisten muuttujien arvot muuttuvat ohjelman suorituksen aikana. Arvojen muuttuessa kaikki niistä riippuvat laskennat suoritetaan uudelleen automaattisesti. Muutokset siis levitetään muualle järjestelmään. Tästä seuraa reaktiivisen ohjelmoinnin tärkein luonteenpiirre eli muuttujien automaattinen päivityminen ohjelman suorituksen aikana. Muutoksen levittämisessä voidaan nähdä yhtymäkohtia SDF-paradigmaan, jossa data etenee ohjelmassa tietovuon rakenteen mukaisesti. [2]

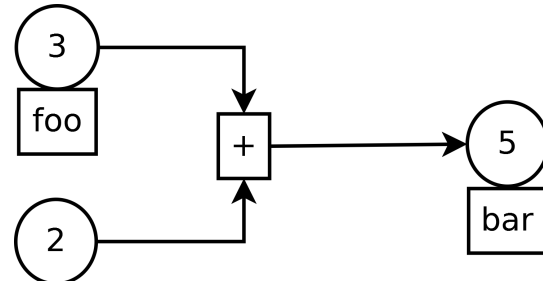
Tarkastellaan muutoksen levittämistä ohjelman 2.1 avulla. Imperatiivisessa ohjelmoinnissa sijoituslausekkeet tallentavat muuttujiin operaation tuloksen, joten rivillä neljä (4) tulostuisi arvo 5, joka tallennettiin muuttujaan `bar` rivillä kaksi (2). Reaktiivisessa ohjelmoinnissa sen sijaan sijoituslauseketta voidaan pitää riippuvuussuhteen luomisena. Rivillä kaksi (2) luodaan siis riippuvuussuhde: muuttujan `bar` arvo riippuu muuttujasta `foo`. Ohjelman riippuvuussuhteet on esitetty kuvassa 2.2. Riippuvuussuhteen luomisen jälkeen muuttujan `foo` arvoa muutetaan rivillä kolme (3). Koska muuttujan `bar` arvo riippuu muuttujasta `foo`, täytyy tämä muutos levittää myös muuttujaan `bar`. Käytännössä muuttujan `bar` arvo lasketaan siis uudelleen. Tästä syystä rivillä neljä (4) tulostuu arvo 6 käytettäessä reaktiivista ohjelmointia.

```

1 foo = 3
2 bar = foo + 2
3 foo = 4
4 print(bar) // 6

```

Ohjelma 2.1: Esimerkki reaktiivisesta ohjelmoinnista.



Kuva 2.2: Ohjelman 2.1 riippuvuudet graafisesti kuvattuna.

2.2.2 Toteutustapoja

Reaktiivisen ohjelmoinnin mahdollistavat ympäristöt voidaan jakaa muutamaan ryhmään sen perusteella kuinka ne toteuttavat muutoksen levittämisen. Edellä annettu esimerkki edustaa muutokset *työntävää* mallia (push-based), jossa muutosten levittäminen tehdään työntämällä muutokset muuttujista eteenpäin niiden riippuvuuksille [2]. Kaikki muuttujan riippuvuudet siis lasketaan aina uudestaan heti kun muuttujan arvo muuttuu.

Toinen yleinen menetelmä muutoksen levittämiseen on muutokset *vetävä* malli (pull-based), jota ensimmäiset reaktiivisen ohjelmoinnin kielet käyttivät [2; 16].

Siinä jokaista muutosta ei levitetä välittömästi, vaan muuttujien arvot ratkaistaan vasta kun niitä tarvitaan. Jos ohjelma 2.1 käyttäisi muutokset vetävää mallia, muuttujaa `bar` tulostettaessa sen arvo ratkaistaisiin vasta tulostushetkellä käymällä riippuvuusketjua läpi vastakkaiseen suuntaan.

Kumpikaan malli ei ole selvästi parempi vaan niihin molempiin liittyy omat etunsa ja haittansa. Muutokset työntävässä mallissa muutosten päivittyminen järjestelmään on nopeaa. Suorituskyvyn kannalta tämä ei kuitenkaan välttämättä ole paras ratkaisu, sillä muuttujien arvot saattavat tulla lasketuksi usein turhaan jos päivitetään muuttujia joita ei sillä hetkellä käytetä. Muutokset vetävässä mallissa muuttujia ei päivitetä turhaan koska muutokset levitetään vasta kun niitä tarvitaan. Turhaa uudelleenlaskentaa saattaa silti esiintyä, koska muuttujan käyttöhetkellä koko sen riippuvuusketju on käytävä läpi ja arvot laskettava uudelleen. Lisäksi vetävän mallin ongelmana on muutosten ja niiden päivittämisen välille syntyvä viive, joka saattaa olla huomattava varsinkin jos riippuvuusketju on pitkä. Työntävää ja vetävää mallia on pyritty myös yhdistelemään jotta niiden kummankin vahvuuksia saataisiin hyödynnettyä. [2; 17]

2.2.3 Funktionaalinen reaktiivinen ohjelmointi

Reaktiiviseen ohjelmointiin yhdistetään usein funktionaalisen ohjelmoinnin periaatteita, jolloin paradigmaa nimitetään *funktionaaliseksi reaktiiviseksi ohjelmoinniksi* (functional reactive programming, FRP [16]). Funktionaalisisessa ohjelmoinnissa pyritään välttämään funktioiden sivuvaikutuksia ja ohjelman tilan käsitettä, jolloin funktioiden tulokset riippuvat ainoastaan funktion argumenteista [26, s. 235]. Tämän tavoitteena on tehdä ohjelmista helpommin ymmärrettäviä ja tukea ohjelmien deklarativista kehitystä. FRP kehitettiin alun perin animaatioiden toteutukseen [16], mutta sitä on käytetty myös muun muassa robotiikassa ja käyttöliittymissä.

2.3 Sovelluskohteita

Tässä kohdassa tarkastellaan sovellusalueita joissa reaktiivisuutta tai reaktiivista ohjelmointia on hyödynnetty.

2.3.1 Reaaliaikajärjestelmät

Vaikka reaktiivisen ohjelmoinnin edeltäjiä SDF-paradigmaa ja synkronisia kieliä on sovellettu laajalti reaktiivisten reaaliaikaohjelmien kehittämisessä, on reaktiivisen ohjelmoinnin käyttö niissä ollut vähäisempää. FRP-paradigmaa on kuitenkin onnistuttu soveltamaan myös reaaliaikaohjelmointiin [56].

2.3.2 Taulukkolaskenta

Taulukkolaskentaohjelmat kuten Microsoft Excel ovat ehkä arkipäiväisin esimerkki reaktiivisuuden hyödyntämisestä. Taulukkolaskentaohjelmassa käyttäjä voi tallentaa taulukon soluihin arvoja tai laskukaavoja. Jos laskukaavassa käytetään toisen solun arvoa, syntyy siihen implisiittinen riippuvuussuhde. Kun jonkin solun arvoa muutetaan, myös kaikki siitä riippuvat solut päivittyvät automaattisesti samaan tapaan kuin muuttujien arvot reaktiivisessa ohjelmassa.

2.3.3 Käyttöliittymät

Käyttöliittymissä käytetään usein tapahtumapohjaista ohjelmointia esimerkiksi hiiritapahtumiin reagointiin: aina hiiren koordinaattien tai painikkeiden asentojen muuttuessa luodaan tapahtuma johon ohjelmassa reagoidaan tarvittaessa. Reaktiivisen ohjelmoinnin avulla tapahtumien toteutus ja hallinta voidaan piilottaa ohjelmoijalta ja automatisoida, mikä yksinkertaistaa ohjelman toteutusta.

Reaktiivisia piirteitä löytyy esimerkiksi QML-kuvauskielestä¹, jota käytetään Qt-ohjelmistokehityksen yhteydessä. QML-skripteissä ominaisuuksia voidaan sitoa toisiinsa (property binding). Esimerkiksi elementin koko voidaan sitoa sen vanhemman kokoon:

```
1 Rectangle {
2     width: parent.width / 2
3     height: parent.height
4 }
```

Tämän jälkeen elementin koko päivittyy automaattisesti sen vanhemman koon muuttuessa.

2.3.4 Web-sovellukset

Web-sovellusten käyttöliittymien kehittämiseen liittyy samoja piirteitä kuin natiivisovellustenkin käyttöliittymiin. Reaktiivisuudesta voi siis olla hyötyä myös web-sovellusten yhteydessä, ja se onkin kasvattanut suosiotaan viime vuosina. Reaktiivisuuden hyödyntämistä web-käyttöliittymissä käsitellään tarkemmin seuraavan luvun kohdassa 3.6.

¹<http://qt-project.org/doc/qt-5/qtqml-index.html>

3. TAUSTAA

Reaktiivisuuden lisäksi tämä diplomityö liittyy läheisesti myös web-sovelluksiin. Tässä luvussa esitellään tämän diplomityön kannalta oleelliset web-sovelluksiin liittyvät taustatiedot. Seuraavissa kohdissa käydään läpi web-sovelluksen ja yhden sivun sovellusten käsitteet, esitellään web-sivuissa yleisesti hyödynnetyt arkkitehtuurityylit sekä responsiivisuuden ja koostepalvelujen käsitteet.

3.1 Web-sovellus

Web-sovellukset ovat selaimessa suoritettavia sovelluksia, jotka on toteutettu perinteisten web-teknologioiden HTML:n ja CSS:n sekä jonkin ohjelmointikielen avulla. Yleensä ohjelmointikielenä käytetään JavaScriptiä, sillä se on tuettu käytännössä kaikissa selaimissa.

Ero web-sovelluksen ja tavallisen web-sivun välillä on häilyvä, sillä nykyään lähes kaikissa web-sivuissa käytetään jonkin verran JavaScript-ohjelmointia. Web-sovellusten voidaan kuitenkin ajatella painottavan sovelluksen toiminnallisuutta, kun taas web-sivujen pääasiallisena tarkoituksena on sisällön tarjoaminen käyttäjälle. Esimerkkejä web-sivuista ovat uutissivustot ja Wikipedia¹. Tyypillisiä web-sovelluksia puolestaan ovat sähköpostisovellukset kuten Googlen Gmail².

Web-sovelluksia voidaan käyttää vaihtoehtona natiivisovelluksille. Natiivisovellukset on yleensä ainakin osittain toteutettava jokaiselle alustalle erikseen, joskin tätä voidaan helpottaa käyttämällä ohjelmointikieltä tai sovelluskehystä joka tarjoaa ohjelman siirrettävyyttä parantavia abstraktioita. Eri selainten välillä sen sijaan on vain vähän toiminnallisuuseroja, joten sama sovellus on käytettävissä eri alustoilla ja selaimilla. Suurempi haaste on saada sovelluksesta käytettävä työpöytä- ja mobiiliselaimilla, mutta vastaavia ongelmia tulisi vastaan myös natiivisovelluksilla. Natiivisovellukset ovat kuitenkin yleensä tehokkaampia sillä selain tuo mukanaan aina jonkin verran ylimääräistä laskentaa joka hidastaa sovelluksen toimintaa.

3.2 Arkkitehtuureista

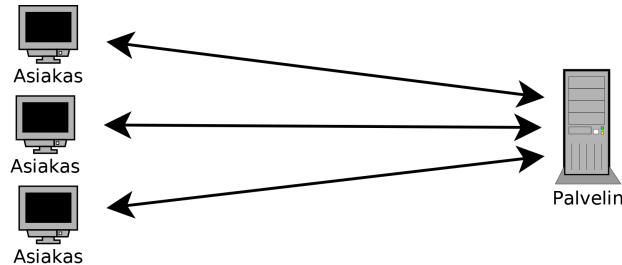
Verkköjärjestelmissä ja web-sovelluksissa käytetään toistuvasti muutamia yleisiä arkkitehtuurityylejä. Tässä kohdassa esitellään näistä kaksi oleellista.

¹<http://www.wikipedia.org/>

²<https://mail.google.com>

3.2.1 Asiakas-palvelin-malli

Yleisin verkkopohjaisten järjestelmien arkkitehtuurityyli on *asiakas-palvelin-malli* (client-server model). Se on hajautettu arkkitehtuuri, jossa asiakas tekee pyyntöjä palvelimelle ja palvelin reagoi pyyntöihin. Tyypillisesti palvelin palvelee useaa asiakasta. Havainnekuva mallista on esitetty kuvassa 3.1. [19, s. 45-46]

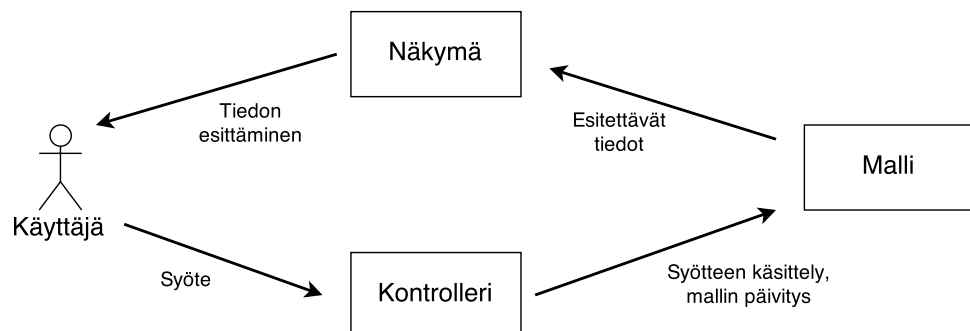


Kuva 3.1: Asiakas-palvelin-malli.

Web-järjestelmien yhteydessä asiakas on selain, joka suorittaa palvelimen palauttamaa verkkosivua. Asiakas kommunikoi palvelimen kanssa HTTP-protokollan avulla internetin välityksellä. Yleensä palvelin on yhteydessä tietokantaan, joka voidaan nähdä myös asiakas-palvelin-mallin kolmantena kerroksena. Asiakas vastaa siis käyttäjästä ja palvelin datan hallinnasta ja prosessoinnista, joskin tämä vastuunjakoa ei aina ole yksiselitteinen.

3.2.2 MVC-arkkitehtuuri

Web-sovelluksissa hyödynnetään usein myös *malli-näkymä-kontrolleri* eli *MVC*-arkkitehtuuria (model-view-controller). MVC-arkkitehtuurissa näkymä esittää informaation käyttäjälle, kontrolleri vastaa käyttäjän syötteen käsittelystä ja malli sisältää näkymässä näytettävän datan sekä logiikan sen manipulointiin. Mallin eri osien välisiä interaktioita on havainnollistettu kuvassa 3.2. Arkkitehtuuri jakaa sovelluksen selkeisiin vastuualueisiin, minkä lisäksi sen etuihin kuuluu näkymän helppo muokattavuus ilman muutoksia tietomalliin tai sovelluslogiikkaan. [31]



Kuva 3.2: MVC-malli.

Web-sovelluksissa näkymä on käytännössä aina asiakassovelluksen vastuulla, mutta kontrollerin ja mallin sijoittaminen vaihtelee. Yhden sivun sovelluksissa kontrolleri on usein asiakkaalla, joka saattaa sisältää myös osan mallin toiminnallisuudesta. Kontrolleri ja malli saattavat kuitenkin sijaita myös kokonaan palvelimella. [31]

3.3 Yhden sivun sovellukset

Tavallisissa web-sivuissa ja web-sovelluksissa eri näkymät on toteutettu omina sivuina, jotka ladataan palvelimelta käyttäjän navigoidessa sivujen välillä. *Yhden sivun sovellukset* (single-page application, SPA) ovat web-sovelluksia, jossa sivu ladataan sovelluksen käynnistyessä ja näkymää päivitetään dynaamisesti tarpeen mukaan. Tämä vähentää sivun uudelleenlatauksen tarvetta. Monesti koko sovelluksen JavaScript-ohjelma ja sivupohjat ladataan sovelluksen käynnistyessä ja näkymien vaihtuessa palvelimelta ladataan vain uutta dataa.

Sivujen dynaamisen päivityksen apuna käytetään yleensä Ajax-tekniikkaa ja DOM-manipulaatiota. *Ajax* (Asynchronous JavaScript and XML) on joukko web-teknologioita, joiden avulla sovellus voi kommunikoida palvelimen kanssa asynkronisesti ja päivittää sivua keskeyttämättä asiakassovelluksen toimintaa [20]. Palvelimelle lähetettävät HTTP-pyyntö tehdään sovelluksen JavaScript-ohjelmassa XMLHttpRequest-rajapinnan avulla. Sivun päivitys tehdään näiden pyyntöjen vastauksissa saatavan datan perusteella. Vaikka Ajax:n nimi viittaa XML-muotoisen datan käyttämiseen, nykyään sen sijasta käytetään usein JSON-formaattia. *JSON*-formaatti (JavaScript Object Notation [9]) vastaa JavaScriptin Object-tyypin esitystapaa, joten sen käyttö JavaScriptin kanssa on luontevaa.

Varsinainen näkymän päivitys tehdään DOM-manipulaation avulla. *DOM* (Document Object Model) on alusta- ja kieliriippumaton rajapinta HTML-sivujen rakenteen esittämiseen, ja se tarjoaa myös rajapinnat niiden dynaamiseen päivittämiseen [55]. Sivua päivitetään poistamalla, lisäämällä ja muokkaamalla elementtejä DOM:n tarjoamia rajapintoja hyödyntämällä, ja tätä toimintaa nimitetään DOM-manipulaatioksi.

Yhden sivun sovelluksissa asiakaspäässä on enemmän sovelluslogiikkaa kuin ei-SPA -sovelluksissa. Tämä vähentää palvelimen kuormaa, koska suurempi osa laskennasta voidaan tehdä asiakaspäässä. Myös käytettävyys voi parantua, sillä sivujen uudelleenlatausten eliminointi nopeuttaa siirtymistä näkymien välillä: sisältö on nopeammin käytettävissä kun palvelimelta ladattavan sisällön määrä on pienempi.

3.4 Responsiivisuus

Erilaisia päätelaitteita, joilla web-sovelluksia voidaan käyttää, on nykyään valtava määrä. Tämä aiheuttaa haasteita web-sovellusten suunnitteluun, sillä eri laitteet asettavat erilaisia rajoitteita web-sivujen käyttöön. Erityisesti vaihtelevat näyttökoot vaikeuttavat käyttöliittymän suunnittelua, sillä sama ulkoasu ei välttämättä tarjoa hyvää käytettävyyttä sekä työpöytäkoneen näytöllä ja älypuhelimella.

Usein vaihtelevien näyttökokojen ongelma ratkaistaan toteuttamalla mobiililaitteille oma verkkosivu, jonka ulkoasu on suunniteltu mobiililaitteiden rajoitteet huomioiden. Tämä ratkaisu voi olla liian rajoittunut, sillä myös mobiililaitteiden näyttökokojen välillä on paljon eroja. Mobiililaitteille suunniteltu verkkosivu ei myöskään välttämättä ole ihanteellinen vaihtoehto esimerkiksi tablettikoneilla käytettäväksi.

Vaihtoehtona erilliselle mobiilisivulle on tehdä web-sovelluksesta *responsiivinen* (responsive web design), jolloin sama käyttöliittymä mukautuu käytössä olevaan laitteeseen – pääsääntöisesti näyttökokoon. Tyypillistä on mukauttaa ulkoasua kaapeammaksi niin, että verkkosivu pysyy päätelaitteen levyisenä mutta kasvaa korkeutta tarpeen mukaan. Palstojen määrää ja sitä kautta myös verkkosivun leveyttä saadaan vähennettyä siirtämällä rinnakkain olevia elementtejä päällekkäin. Yleensä tämä toteutetaan käyttämällä verkkosivussa *joustavaa ruudukkorakennetta* (fluid grid) ja CSS:n media query -ominaisuuksia. Näyttökoon lisäksi responsiivinen sovellys voi mukautua esimerkiksi kosketusnäytön tai hiiren puuttumiseen. [33]

Responsiivisuuden etuna on, että voidaan käyttää vain yhtä sivua usean sijaan. Se myös mahdollistaa laajemman laitevalikoiman tukemisen helpommin. Responsiivisen ulkoasun toteuttamiseen on nykyään olemassa useita kirjastoja, joista yksi suosituimmista on Twitterin Bootstrap [53].

3.5 Koostepalvelut ja joukkoistaminen

Monissa verkkopalveluissa palvelun sisältö on kriittinen tekijä palvelun menestymisen kannalta. Ongelmaksi voi kuitenkin muodostua riittävän sisällön saaminen palveluun, sillä sisällön tuottaminen tai kerääminen itse voi olla haastavaa ja kallista. Usein onkin järkevämpää käyttää jo olemassa olevaa sisältöä tai hyödyntää sisällön luomisessa ulkopuolisia tahoja.

Monet verkkopalvelut tarjoavat julkisen, avoimen rajapinnan jonka kautta muut sovellukset voivat käyttää niiden dataa. Erityisesti näitä rajapintoja hyödyntävät *koostepalvelut* (mashup). Koostepalvelut ovat web-sovelluksia, jotka käyttävät useista lähteistä saatua sisältöä tai dataa ja esittävät sen käyttäjälle hyödyllisellä tavalla [57]. Tyypillinen koostepalvelu saattaa noutaa jonkin asian paikkatiedot yhdestä palvelusta, ja sijoittaa ne esimerkiksi Google Maps -kartalle³.

³<https://maps.google.com/>

Sisällön tuottamisessa voidaan hyödyntää palvelun käyttäjiä, jolloin puhutaan *joukkoistamisesta* (crowdsourcing). Joukkoistamista hyödyntävässä web-palvelussa käyttäjien osallistuminen sisällön tuottamiseen mahdollistetaan tyypillisesti tarjoamalla käyttöliittymät sisällön lisäämistä ja muokkausta varten. Esimerkkejä joukkoistamista hyödyntävistä verkkopalveluista ovat Wikipedia-tietosanakirja ja OpenStreetMap-karttapalvelu [49], joiden sisältö on käyttäjien luomaa ja ylläpitämää.

3.6 Reaktiivisuus web-sovelluksissa

Web-sovelluksissa halutaan usein päivittää osia sivusta automaattisesti kun sivun käyttämä data muuttuu. Tavoitteena voi olla kriittisen tiedon saattaminen mahdollisimman nopeasti käyttäjän nähtäväksi tai käyttökokemuksen parantaminen. Puhuttaessa reaktiivisuudesta web-sovelluksissa viitataan pääsääntöisesti juuri tähän toimintaan: sivua päivitetään automaattisesti datan muuttuessa. Reaktiivisuus voi rajoittua web-sovelluksen asiakasovelluksen sisälle, tai ulottua myös palvelimelta saatuun dataan.

Reaktiivisuutta on hyödynnetty muun muassa Facebookissa⁴ ja Googlen Gmail-sovelluksessa. Facebookissa sisäänkirjautuneen käyttäjän etusivulla näytetään muiden käyttäjien tilapäivityksiä, jotka päivittyvät automaattisesti kun uusia päivityksiä lisätään. Gmailissa uudet sähköpostit tulevat automaattisesti näkyville käyttäjän postilaatikkoon ilman sivun uudelleenlatausta. Myös molempien palvelujen chat-toiminnallisuudet ovat reaktiivisia: sisäänkirjautuneet kaverit ja vastaanotetut viestit pysyvät ajan tasalla.

Reaktiivisen ohjelmoinnin tukemiseen web-sovelluksissa on kehitetty useita kirjastoja. Avoimena lähdekoodina julkaistu Facebookissa ja Instagramissa käytetty React⁵ on kirjasto reaktiivisten käyttöliittymäkomponenttien toteuttamiseen. React auttaa toteuttamaan sovellukseen yhdensuuntaisen tietovuon, jossa sovelluksessa oleva data päivittyy automaattisesti Reactilla toteutettuihin käyttöliittymäkomponentteihin. Koska React rajoittuu ainoastaan käyttöliittymään, se ei yksinään riitä toteuttamaan myös palvelimen käsittävää reaktiivisuutta.

Muita esimerkkejä reaktiivisista web-kehitykseen soveltuvista kirjastoista ovat RxJS⁶, Bacon.js⁷ ja Elm⁸, jotka tarjoavat vaihtelevan tasoista tukea reaktiivisuudelle. Näiden lisäksi mainittakoon tässä diplomityössä käsiteltävässä sovelluksessa käytetty Meteor-sovelluskehys, joka tarjoaa työkalut kokonaisvaltaisen reaktiivisen sovelluskehityksen kehittämiseen käyttöliittymää, palvelinta ja tietokantaa myöten.

⁴<https://www.facebook.com/>

⁵<http://facebook.github.io/react/>

⁶<https://github.com/Reactive-Extensions/RxJS>

⁷<https://github.com/baconjs/bacon.js>

⁸<http://elm-lang.org/>

4. METEOR-SOVELLUSKEHYS

Meteor on web-sovellusten toteutukseen soveltuva *full stack* -sovelluskehys. Monet sovelluskehukset keskittyvät vain tietylle järjestelmän osa-alueelle, kuten käyttöliittymään tai palvelinpuolen toteutukseen. Full stack -sovelluskehukset sen sijaan tarjoavat sovellukselle koko ohjelmistopinon — Meteorin tapauksessa pohjan asiakas- ja palvelinpään toteutukselle sekä tietokantaratkaisun — ja sitä kautta rajaavat sovelluksen kokonaisarkkitehtuuria.

Tässä luvussa esitellään Meteorin arkkitehtuuria, ominaisuuksia ja käyttöä. Kohdassa 4.1 esitellään Meteorissa käytetyt keskeiset taustateknologiat. Meteorin arkkitehtuurista ja ominaisuuksista annetaan yleiskuva kohdassa 4.2, ja sovelluskehysten reaktiivisia ominaisuuksia ja niiden toteutusta käydään läpi kohdassa 4.3.

4.1 Taustaa

Meteor rakentuu usean yleisesti käytössä olevan web-tekniikan ja sovelluskehysten päälle. Tässä kohdassa esitellään keskeisimmät näistä teknologioista.

4.1.1 MongoDB

MongoDB on tämän hetken suosituin NoSQL-tietokanta [10; 40]. Se on avoimen lähdekoodin dokumenttipohjainen tietokanta, jossa tietoalkiot tallennetaan BSON-muodossa [42]. *BSON* (Binary JSON) on JSON-formaatin laajennos, joka sallii JSON-standardin tukemien ominaisuuksien lisäksi muun muassa binääri- ja päivämäärätyyppisen datan tallentamisen [8]. MongoDB on myös tehokas sekä kyselyettä kirjoitusoperaatioissa verrattuna esimerkiksi MySQL-relaatiotietokantaan [48].

Dokumentit jaetaan MongoDB:ssä *kokoelmiin* (collection), jotka vastaavat relaatiotietokantojen taulu-käsitettä. Yhden kokoelman dokumenteilla voi kuitenkin olla erilaisia kenttiä eikä dokumentteja sidota skeemaan. MongoDB:en tehtävät kyselyt eivät palauta dokumentteja suoraan. Sen sijaan ne palauttavat *kursorin* (cursor), jota iteroimalla saadaan luettua varsinaiset dokumentit. [43; 44]

Yksi oleellisista MongoDB:n rajoitteista on relaatiotietokantojen JOIN-lauseiden tapaisten liitosoperaatioiden puute. Jos halutaan yhdistellä dataa eri kokoelmista on välttämätöntä tehdä useita kyselyitä. Tämä heikentää suorituskykyä ja voi olla vaikeasti hallittavaa. Erillisten kokoelmien sijaan onkin monesti järkevämpää käyttää sisäkkäisiä kokoelmia tai duplikoida tarvittavat kentät useampaan kokoelmaan.

4.1.2 Node.js

Node.js on Google Chromessakin käytetyn tehokkaan JavaScript-moottori V8:n päälle rakennettu sovelluskehys [46]. Se soveltuu JavaScript-ohjelmien suorittamiseen esimerkiksi palvelimella, ja siten osaksi web-sovellusten palvelinpuolen toteutusta. Node.js:n tavoitteena on olla tehokas ja skaalautuva ratkaisu sovellusten palvelinpuolelle, ja monet suuret verkkopalvelut — muun muassa Wikipedia, Yahoo ja Paypal [50] — ovatkin päätyneet käyttämään sitä näiden ominaisuuksien ansiosta.

Node.js:n tehokkuus pohjautuu paljolti sen ratkaisuun hyödyntää asynkronisia tapahtumia ja tapahtumasilmukkaa. Monet muut palvelinpuolen sovelluskehukset luovat jokaista HTTP-yhteyttä varten oman säikeen, mikä saattaa olla tehotonta. Nodessa puolestaan käytetään yksisäikeistä mallia. HTTP-pyyntöjä ja muita tapahtumia varten käytetään takaisinkutsufunktioita, joita tapahtumasilmukka käsittelee automaattisesti. Tämän ansiosta suurenkin yhteysmäärän käsittely on mahdollista tehokkaasti ja pitkäkestoisetkin tapahtumat voidaan suorittaa ei-estävästi.

Tehokkuuden lisäksi Node.js:n houkuttelevia puolia ovat sen vahva ekosysteemi ja suuri kolmansien osapuolien julkaisemien pakettien valikoima. Pakettien jakeluun käytetään *npm*-työkalua (Node Packaged Modules)¹, jonka kautta on kirjoitushetkellä ladattavissa yli 130 000 pakettia esimerkiksi yleisten toiminnallisuuksien toteuttamiseen ja muiden kirjastojen integrointiin.

Vaikka Node.js ei tarjoakaan tukea varsinaisille säikeille, voidaan siinä käyttää niin sanottuja *kevyitä säikeitä* (fiber). Kevyet säikeet eroavat normaaleista säikeistä siinä, että ohjelmoija voi päättää itse missä kohtaa kevyt säie luovuttaa suoritusvuoron. Tämä yksinkertaistaa kevyiden säikeiden hallintaa, mutta ei takaa yhtä tasaisesti jakautunutta suoritusaikaa kuin normaaleilla säikeillä. Kevyet säikeet eivät kuulu Node.js:n ydintoiminnallisuuteen, vaan ne ovat saatavilla erillisenä npm-pakettina. [27; 47]

4.1.3 WebSocket

Interaktiivisissa web-sovelluksissa – kuten peleissä – tulee usein vastaan tilanteita, jossa verkkosivulle pitäisi saada päivitettyä jokin resurssi sen muuttuessa palvelimella. Tällainen tilanne soveltuu huonosti webissä yleisesti käytettyyn asiakas-palvelinmalliin, jossa palvelin lähettää vastauksia ainoastaan asiakkaan tekemiin pyyntöihin. HTTP-protokollaa käytettäessä ongelma ratkaistaan usein tekemällä toistuvia kyselyitä palvelimelle, mistä seuraa paljon ylimääräistä verkkoliikennettä erityisesti jos resurssi muuttuu harvoin.

Yksinkertaisempaan ratkaisuun tällaisia ongelmia varten on kehitetty WebSocket-teknologia. Se koostuu WebSocket-protokollasta ja sitä käyttävästä rajapinnasta,

¹<https://www.npmjs.org/>

jonka selaimet toteuttavat. WebSocket-teknologian avulla voidaan asiakkaan ja palvelimen välille muodostaa kaksisuuntainen yhteys, jonka kautta kumpikin osapuoli voi vapaasti lähettää dataa toiselle. [18; 25]

Kaksisuuntaisen kommunikation lisäksi WebSocket-teknologian etuihin kuuluu protokollan keveys. HTTP-protokollaa käytettäessä sekä asiakas että palvelin joutuvat usein pitämään yllä useita samanaikaisia TCP-yhteyksiä, ja jokainen viesti sisältää varsinaisen viestin lisäksi paljon ylimääräistä otsikkokenttädataa. WebSocket käyttää asiakkaan ja palvelimen välillä yhtä TCP-yhteyttä, joka pidetään jatkuvasti avoinna. Näin kaikki dataliikenne saadaan tehtyä saman yhteyden yli eikä yhteyksiä tarvitse jatkuvasti avata ja sulkea. Lisäksi WebSocket-protokollan viesteissä otsikkokentät ovat huomattavasti pienempiä kuin HTTP-protokollassa, jolloin hyötykuorman osuus viestistä on paljon suurempi. [18]

4.1.4 Handlebars

Handlebars on web-sovellusten toteutuksen apuna käytettävä sivupohjatyökalu. Sivupohjatyökalujen tarkoituksena on helpottaa dynaamisten web-sivujen toteutusta tarjoamalla työkaluja dynaamisen sisällön määrittelyyn ja käsittelyyn. Handlebarsissa dynaamisen sisällön merkitsemiseen käytetyt lausekkeet erotetaan muusta merkkauksesta kaksinkertaisin aaltosulkein, esimerkiksi `<h1>{{title}}</h1>`. Nämä lausekkeet esiprosessoidaan HTML-merkkauksesta, jolloin niiden sisältö korvataan muuttujien sen hetkisten arvojen mukaisesti. Lausekkeissa voidaan käyttää myös yksinkertaista logiikkaa kuten ehtolauseita ja toistorakenteita. [23]

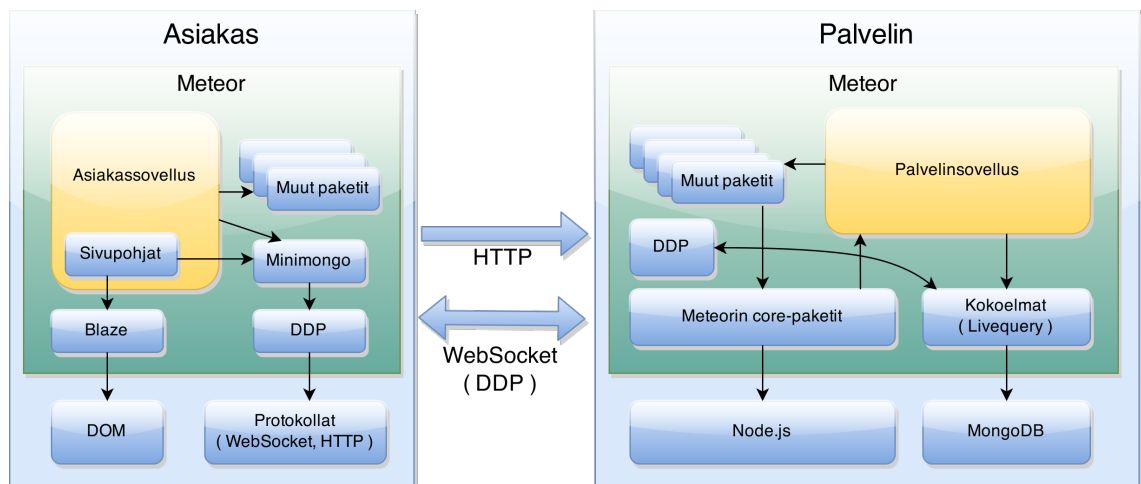
4.2 Yleiskuvaus

Vaikka reaktiivisuuden tukemiseen onkin olemassa useita sovelluskehyskiä, harva ratkaisu tukee sitä yhtä kokonaisvaltaisesti kuin Meteor. Reaktiivisuus ja datan ajantasaisuus ovat oleellisessa asemassa Meteorissa, joka pyrkiikin tukemaan niitä toteuttamalla reaktiivisuutta järjestelmän kaikilla osa-alueilla tietokannasta näkymiin.

Meteor on avointa lähdekoodia ja sen GitHub-projektiin on tähän mennessä osallistunut lähes 200 tekijää [35]. Sitä on kehitetty vuodesta 2011 lähtien, mutta tuotantovalmis 1.0 versio julkaistiin vasta 28. lokakuuta 2014 [13]. Jo ennen 1.0 version julkaisemista Meteor ja sen kehitys oli kuitenkin herättänyt paljon kiinnostusta. Meteor oli yksi seuratuimmista projekteista GitHub-palvelussa vuonna 2012 [15], ja vuoden 2012 heinäkuussa se sai kerättyä 11,2 miljoonaa dollaria rahoitusta [51].

4.2.1 Arkkitehtuurista

Meteor on suunniteltu ensisijaisesti käytettäväksi yhden sivun sovellusten toteuttamiseen. Sillä toteutetut sovellukset edustavat asiakas-palvelin-mallia, jossa palvelin on yhteydessä tietokantaan ja tarjoaa asiakassovelluksille dataa näkymien luomista varten. Yleiskuva sovelluskehysten arkkitehtuurissa on esitetty kuvassa 4.1. Meteorin komponentteja kuvataan tarkemmin seuraavissa alakohdissa.



Kuva 4.1: Meteor-sovelluskehysten arkkitehtuuri.

Meteorin full stack -mallissa on käytössä yksi yhteinen ohjelmointikieli, sillä sekä asiakas- että palvelinpuoli ovat JavaScript-pohjaisia. Yhden ohjelmointikielen ansiosta suuri osa lähdekoodista voidaan tarvittaessa ottaa käyttöön sekä asiakassovelluksessa että palvelimella. Yleensä kuitenkin osa lähdekoodista rajoitetaan pelkästään asiakassovelluksen tai palvelimen käyttöön.

Suuri osa Meteorin toiminnallisuudesta on jaettu itsenäisiin paketteihin, jotka toimivat sekä asiakassovelluksessa että palvelimella. Meteorin ydintoiminnallisuus on jaettu hieman yli kymmeneen ydinpakettiin, joihin kuuluvat muun muassa tietokantayhteyden, palvelimen ja asiakassovelluksen välisen kommunikoinnin ja asiakassovelluksen näkymien hallinnan toteuttavat paketit. Tärkeimpiä näistä käsitellään seuraavissa alikohdissa. Paketteja on myös mahdollista jättää pois, mikäli kaikkea toiminnallisuutta ei tarvita. [37]

Virallisten pakettien lisäksi Meteorissa on mahdollista käyttää yhteisön jäsenten tuottamia paketteja, joita jaetaan Atmosphere-palvelun kautta. Yhteisön tuottamia paketteja on tällä hetkellä yli 4 000, ja niiden joukossa on esimerkiksi runsaasti sovittimia erilaisiin JavaScript-kirjastoihin. [1]

4.2.2 Palvelin ja tietokanta

Meteorin palvelinpuolen toteutuksen pohjana on Node.js-sovellusalusta. Tämän ansiosta Meteorissa voidaan hyödyntää myös npm:n runsasta pakettivalikoimaa. Vaikka Node.js-sovelluksissa yleensä käytetään asynkronista takaisinkutsufunktioihin perustuvaa ohjelmointityyliä, Meteor käyttää jokaista pyyntöä kohden yhtä kevyttä säiettä. Tämä vähentää tarvittavien takaisinkutsufunktioiden määrää ja Meteorin kehittäjiä mukaan myös soveltuu paremmin tyyppillisen Meteorin palvelinsovelluksen ohjelmointiperiaatteisiin. Oletuksena jokaista asiakassovellusta kohti palvelimella on aktiivisena enintään yksi kevyt säie kerrallaan. [37]

Meteorissa käytetään oletusarvoisesti MongoDB-tietokantaa. Se soveltuu hyvin Meteorin JavaScript-ympäristöön, sillä MongoDB:n BSON-dokumentteja on helppo käsitellä JavaScript-ohjelmassa. Tällä hetkellä tukea muille esimerkiksi SQL-pohjaisille tietokannoille ei ole saatavilla, mutta kehittäjät ovat luvanneet tukea myös muita tietokantoja tulevien Meteor versioiden myötä [37].

4.2.3 Asiakassovelluksen näkymät

Meteor-sovellukset ovat yhden sivun sovelluksia, joten Meteorin palvelin ei lähetä asiakassovellukselle staattista HTML-merkkausta vaan asiakassovellus generoi näkymät palvelimelta saadun datan perusteella. Näkymät ovat reaktiivisia ja niiden luomisessa käytetään *sivupohjia* (template). Suuria kolmannen osapuolen sovelluskehelyksiä ei ole käytetty, vaan asiakassovellus on lähinnä Meteorin oman kehitystyön tulosta.

Asiakassovellus käyttää näkymien toteutuksessa Meteorin Blaze kirjastoa. Blaze mahdollistaa reaktiivisten näkymien luomisen yksinkertaisten HTML-sivupohjien perusteella, joten sovelluskehittäjä voi määrittää näkymät deklaratiiivisesti ilman näkymien päivittämiseen vaadittavan logiikan määrittelyä. HTML-merkkauksen lisäksi sivupohjiin liittyvät läheisesti niihin JavaScript-ohjelmassa liitetyt tapahtumankäsittelijät ja apufunktiot. Niiden avulla muun muassa syötetään sivupohjalle sen käyttämä data ja reagoidaan käyttäjän syötteisiin. [7; 36]

Oletuksena Blaze käyttää Spacebars-sivupohjatyökalua. Se on Meteorin versio Handlebars-sivupohjatyökalusta ja käyttää pääosin samaa syntaksia. Spacebarsin lähtökohtana on kuitenkin reaktiivisuuden tukeminen. Vaihtoehtona Spacebarsille on mahdollista käyttää esimerkiksi Jade-sivupohjatyökalua. [36; 52]

Sivupohjia ei lueta HTML-tiedostoista sovelluksen suorituksen aikana, vaan Blaze kääntää sivupohjat JavaScript-lähdekoodiksi ennen niiden lähettämistä asiakassovellukselle. Blaze käyttää käännettyjä sivupohjia ja huolehtii DOM:in päivityksestä sovelluksen suorituksen aikana. [7]

4.2.4 Minimongo

Tärkeä ominaisuus Meteorissa on asiakassovelluksen käytettävissä oleva MongoDB-emulaattori Minimongo. Minimongo tarjoaa asiakassovelluksen käyttöön MongoDB:n rajapinnan, joten sen kautta voidaan tehdä kyselyitä sekä muita tietokantaoperaatioita samaan tapaan kuin palvelimella. MVC-arkkitehtuurimallista se toteuttaa siis mallin toiminnallisuutta. Minimongo toimii myös asiakassovelluksen välimuistina, mikä vähentää asiakkaan ja palvelimen välisen dataliikenteen tarvetta. Blaze toimii yhdessä Minimongon kanssa toteuttaakseen näkymien reaktiivisen päivittymisen. [14; 37]

Minimongon kolmantena tehtävänä on *latenssin kompensointi* (latency compensation). Kun asiakas tekee kirjoitusoperaation tietokantaan, näkymät halutaan päivittää vastaamaan uutta tilannetta mahdollisimman nopeasti. Jotta asiakassovelluksen ei tarvitsisi odottaa palvelimen *kiertoviivettä* (round trip time, RTT), kirjoitusoperaatiot tehdään samanaikaisesti asiakkaan paikalliseen Minimongoon. Näin muuttunut data on heti asiakassovelluksen käytettävissä ja näkymät voidaan päivittää. Kun palvelimelta saapuu vastaus kirjoitusoperaatioon, Minimongo päivittyy vastaamaan oikeaa tilannetta. [14; 37]

4.2.5 Asiakkaan ja palvelimen välinen dataliikenne

Datan välitys palvelimelta asiakassovellukselle tehdään Meteorissa *julkaisu/tilaus-mallin* (publish/subscribe -pattern) mukaisesti. Palvelin määrittää julkaistavat *tietueet* (record set), joita asiakassovellukset voivat tilata. Tietueeseen kuuluva data voidaan vapaasti määritellä palvelimella, mutta tyypillisesti ne sisältävät tietokannan yhden kokoelman dokumenteista rajatun osan.

Kun asiakassovellus tilaa tietueen, Meteor lähettää tietueen sisältämät dokumentit asiakassovelluksen Minimongo-tietokantaan ja huolehtii niiden reaktiivisesta päivittämisestä. Tietueiden avulla voidaan myös rajoittaa asiakassovellukselle toimitettavaa dataa, mikä voi olla tarpeen esimerkiksi käyttöoikeus- tai tehokkuussyistä. Tyypillisesti asiakassovellus tilaa useita tietueita, ja tilauksia lisätään ja poistetaan kun asiakassovelluksen tila ja sitä kautta myös vaaditun datan tarve muuttuu.

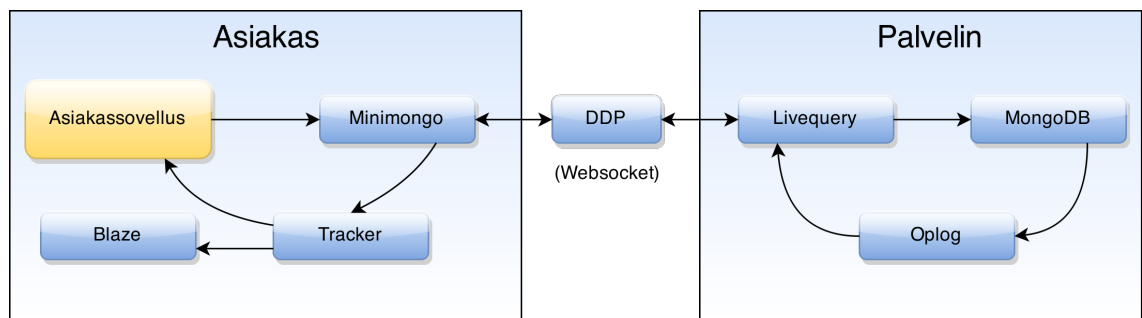
Tietueisiin liittyvä kommunikointi asiakassovelluksen ja palvelimen välillä tehdään WebSocket-protokollan yli. WebSocket soveltuu Meteoriin hyvin, sillä sen avulla palvelimen on helppo myös lähettää asiakkaalle dataa. Tämä on tarpeen esimerkiksi tietueiden sisältämien dokumenttien päivittyessä. WebSocketin lisäksi Meteorissa käytetään myös HTTP-protokollaa muun muassa staattisten resurssien kuten kuvien siirtämiseen.

4.3 Reaktiiviset ominaisuudet

Reaktiivisuuden tukeminen on yksi Meteor-sovelluskehysten tärkeimmistä periaatteista. Koska reaktiivisuus on hyvin vahvasti tuettuna Meteorissa, se lienee myös Meteorin tunnusomaisin piirre. Reaktiivinen ohjelmointityyli on oletuksena käytössä lähes kaikkialla Meteorissa, ja JavaScriptin funktionaalisten ominaisuuksien hyödyntämisen ansiosta Meteorissa on myös selkeitä funktionaalisen reaktiivisen ohjelmoinnin piirteitä.

Tässä kohdassa käydään läpi, kuinka kukin Meteor-sovelluskehysten osa toteuttaa osaltaan reaktiivisuutta ja kuinka näiden yhteistoiminnalla saavutetaan Meteorin tietokannasta näkymiin ulottuva reaktiivisuus. Lisäksi kuvataan näiden reaktiivisten ominaisuuksien käytännön toteutuksia.

Meteorin reaktiivisuuden toteuttavat komponentit ja datan liikkuminen niiden välillä on esitetty kuvassa 4.2. Nämä komponentit käydään tarkemmin läpi seuraavissa alakohdissa.



Kuva 4.2: Datan liikkuminen Meteorin reaktiivisten komponenttien välillä.

4.3.1 Meteor Tracker

Reaktiiviset ominaisuudet JavaScript-ohjelmassa toteutetaan Meteorissa Tracker-kirjaston avulla. Sen tehtävänä on käytännössä tarjota kehys reaktiivisille muuttujille ja niiden riippuvuuksien seurantaan. Trackerissa reaktiivisten muuttujien roolia toteuttavat *reaktiiviset datalähteet* (reactive data source), jotka ovat datan tuottajia. Esimerkiksi Meteorin asiakassovelluksen Minimongo-tietokanta ja siihen tehtävien kyselyjen palauttamat cursorit toimivat reaktiivisina datalähteinä. Niiden riippuvuuksia taas vastaavat *reaktiiviset laskennat* (reactive computation), jotka käyttävät datalähteiden dataa. Reaktiiviset laskennat sisältävät mielivaltaisen JavaScript-funktion, joka suoritetaan uudelleen kun laskennan käyttämä data muuttuu. Muun muassa Meteorin näkymät ja suuri osa Meteorin käyttävän sovelluksen omasta toteutuksesta suoritetaan reaktiivisten laskentojen sisällä. [39]

Ohjelmassa 4.1 käytetään reaktiivista metodia `Tracker.autorun`, joka luo sille annetusta funktiosta reaktiivisen laskennan. Tässä tapauksessa funktiossa on käytetty reaktiivista datalähdettä `Session.get`, joka palauttaa sille parametrina annetun nimen mukaisen sessiomuuttujan. Kun tämän sessiomuuttujan – tässä tapauksessa `currentSiteId` – sisältämä arvo muuttuu, myös `Session.get` metodin arvo muuttuu. Tämän seurauksena `Tracker.autorun` metodin sisältämä funktio suoritetaan uudelleen.

```
1 Tracker.autorun(function() {
2   Meteor.subscribe('siteComments', Session.get('currentSiteId'));
3 }
```

Ohjelma 4.1: Esimerkki reaktiivisesta laskennasta.

Trackerin yhtenä tavoitteena on, että sen reaktiivisia ominaisuuksia voisi hyödyntää muuttamatta muuta ohjelmaa. Meteor nimittää tätä lähestymistapaa *transparentiksi reaktiiviseksi ohjelmoinniksi* (transparent reactive programming) – reaktiivista ohjelmointia ilman muutoksia lähdekoodiin. Tämän idean tukemiseksi Tracker tunnistaa reaktiivisten laskentojen riippuvuudet datalähteisiin automaattisesti eikä ohjelmoijan tarvitse määritellä niitä. Näin tapahtuu myös ohjelmassa 4.1, jossa ei eksplisiittisesti määritetty että reaktiivinen laskenta riippuu `Session.get` datalähteestä. [39]

Riippuvuuksien automaattisen tunnistamisen mahdollistamiseksi Tracker pitää yllä tietoa kulloinkin suoritettavasta reaktiivisesta laskennasta. Viite aktiiviseen laskentaan tallennetaan globaaliin muuttujaan `Tracker.currentComputation`. Kun reaktiivisen datalähteen metodeita kutsutaan, Tracker tarkistaa tämän muuttujan avulla onko reaktiivista laskentaa käynnissä. Jos muuttuja on asetettu eli jos datalähdettä käytettiin reaktiivisen laskennan sisältä, datalähde tallentaa tämän laskennan itselleen muistiin. Reaktiivinen datalähde siis tietää mitkä laskennat riippuvat siitä. Riippuvuudet rekisteröityvät vaikka reaktiivisia datalähteitä ei käytettäisi suoraan laskennan omassa funktiossa vaan esimerkiksi jossakin sen käyttämässä funktiossa. [39]

Reaktiivisten laskentojen reaktiivinen päivittäminen tehdään muutokset työntävän reaktiivisen ohjelmoinnin mukaisesti. Kun reaktiivinen datalähde muuttuu, se ilmoittaa muutoksesta kaikille siitä riippuville laskennoille. Käytännössä datalähde invalidoi kaikki nämä laskennat eli merkitsee ne suoritettavaksi uudelleen. Invalidoituja laskentoja ei suoriteta uudelleen välittömästi, vaan vasta seuraavan *päivityskierroksen* (flush cycle) yhteydessä. Tämän tarkoituksena on parantaa laskentojen päivittämisen tehokkuutta ja selkeyttää Trackerin toimintaa. [39]

4.3.2 Näkymien reaktiivisuus

Näkymien toteuttamiseen käytettävä Blaze-kirjasto hyödyntää reaktiivisuuden toteuttamiseen Trackerin toiminnallisuutta. Meteor piirtää sivupohjat käyttämällä reaktiivista `Blaze.render`-metodia, joka tekee sivupohjasta reaktiivisen laskennan Trackerin avulla [37]. Koska sivupohjat piirretään reaktiivisen laskennan sisällä, ne reagoivat kaikkien niissä käytettyjen reaktiivisten datalähteiden muutoksiin. Näin muun muassa asiakassovelluksen Minimongo-tietokannassa tapahtuvat muutokset aiheuttavat myös näkymien päivittymisen.

Ohjelmassa 4.2 on esitetty `siteComments`-sivupohja, joka luo yksinkertaistetun listan kohteeseen liittyvistä kommentteista. Sivupohjassa on yksi reaktiivinen apufunktio `comments`, joka palauttaa kohteeseen liittyviin kommentteihin osoittavan kursorin. Kursorin osoittamaa dataa hyödynnetään sivupohjan `#each`-lausekkeessa, joka iteroi kursorin datan läpi ja luo jokaiselle kommentille sitä vastaavan DOM-elementin. Aina kun tämä data muuttuu, Meteor päivittää näkymän vastaamaan uutta tilannetta.

```
1 // siteComments.html
2 <template name="siteComments">
3   {{#each comments}}
4     <p>{{text}}</p>
5   {{/each}}
6 </template>
7
8 // siteComments.js
9 Template.siteComments.helpers({
10   comments: function() {
11     var currentSite = Session.get('siteDetailsId');
12     return Comments.find(
13       {site: currentSite},
14       {sort: {dateTime: -1}}
15     );
16   }
17 });
```

Ohjelma 4.2: Esimerkki reaktiivisesta sivupohjasta.

Meteorissa on kiinnitetty huomiota näkymien reaktiivisen päivittämisen tehokkuuteen. Päivittämiseen tarvittavien DOM-operaatioiden määrän minimoimiseksi Blaze yrittää päivittää näkymät mahdollisimman vähillä muutoksilla. Sivupohjien datan lähteinä usein käytettävät tietokannan kursorit ovat tässä hyödyllisiä, sillä niiden palauttavat dokumentit voidaan yksilöidä id-tunnisteen perusteella ja muutokset on helppo tunnistaa. Kursoria on käytetty myös ohjelmassa 4.2. Jos jokin kommentteista tämän ohjelman mukaisessa näkymässä muuttuu, koko näkymää ei siis piirretä uudestaan vaan Blaze päivittää ainoastaan muuttuneet elementit. [52]

4.3.3 DDP-protokolla

Tietueiden välitys ja niiden päivittäminen asiakkaan ja palvelimen välillä tehdään Meteorissa *DDP-protokollan* (Distributed Data Protocol) avulla. DDP on yksinkertainen protokolla, joka on kehitetty Meteoria varten. Protokolla on kieli- ja tietokantariippumaton, joten sitä voidaan käyttää myös muiden ympäristöjen kuin Meteorin kanssa. DDP-protokollaa käytetään Meteorissa myös *etäproseduurikutsujen* (Remote Procedure Call) suorittamiseen. [11; 12]

DDP-protokollan mukaisessa yhteydessä asiakas ja palvelin kommunikoivat keskenään WebSocket-kanavan yli lähetettävillä määrätyn muotoisilla JSON-viesteillä. Jokaisessa viestissä on `msg`-kenttä, joka määrittää viestin tyyppin ja rakenteen. Yhteys asiakkaan ja palvelimen välille muodostetaan asiakkaan lähettämän `connect`-viestin aloittamalla proseduurilla, jossa sovitaan käytettävä protokollan versio. Yhteydenmuodostuksen päätteeksi asiakas saa palvelimelta istuntotunnuksen, jolla se voi tarvittaessa yhdistää uudelleen olemassa olevaan DDP-istuntoon. [11]

Yksi esimerkki DDP:n käyttökohteesta Meteorissa on palvelimen julkaisemien tietueiden tilaaminen asiakkaalle. Kun asiakassovellus kutsuu `Meteor.subscribe`-metodia, Meteor lähettää palvelimelle tietueen tilaavan `sub`-viestin. Kun palvelin vastaanottaa tilausviestin, se alkaa ylläpitämään asiakasta tietueen datan tilasta. Tämä liikennöinti tapahtuu viestien `added`, `changed` ja `removed` avulla, joita käytetään kun tietueeseen lisätään, muutetaan tai poistetaan dataa. Alla on esimerkiksi `added`-viestistä, jossa palvelin lähettää asiakkaalle uuden `comments`-kokoelman dokumentin:

```

1  {
2    "msg": "added",
3    "collection": "comments",
4    "id": "ZctZ45pK659d8yKK6",
5    "fields": {
6      "author": "RBd3GTCmGMcu7DvyF",
7      "text": "Uusi kommentti!",
8      "dateTime": {
9        "$date": 1422389209541
10     }
11   }
12 }
```

Useat palvelimen julkaisemat tietueet saattavat sisältää samaan kokoelmaan kuuluvaa dataa. Tästä johtuen on mahdollista, että saman aikaisesti tilatut tietueet sisältävät päällekkäistä tai ristiriitaista dataa. Vaikka useampi asiakkaan tilaama tietue käsittelee samaa kokoelmaa, asiakas ylläpitää vain yhtä datajoukkoa jokaista kokoelmaa kohti. Viestien määrän pienentämiseksi eri tilauksista saadut dokumentit yhdistetään samaan viestiin ennen datan lähettämistä asiakkaalle. Ristiriitalanteissa — esimerkiksi jos eri tilaukset antavat saman dokumentin jollekin kentälle eri arvot — palautetaan jokin mahdollisista arvoista. [11]

4.3.4 Livequery

Koska Meteorin tarkoituksena on olla mahdollisimman reaaliaikainen sovelluskehys, on tietokantaan tehdyistä muutoksista saatava tieto, jotta ne voidaan lähettää myös asiakassovelluksille. Tämän muutosten seurannan Meteorissa toteuttaa Livequery-toiminnallisuus. Livequeryn tehtävänä on mahdollistaa kyselyiden tekeminen tietokantaan ja ilmoittaa kun näiden kyselyiden tulokset muuttuvat. Tieto muutoksista voidaan saada tietokannasta riippuen esimerkiksi tietokannan lähettämien tapahtumien avulla. Jos käytetty tietokanta ei tarjoa tarvittavia mekanismeja tai palvelinsovelluksella ei ole riittäviä käyttöoikeuksia, voidaan muutoksia seurata myös tekemällä toistuvia kyselyitä eli pollaamalla. Tämä on tosin melko tehoton vaihtoehto eikä sovellu tuotantokäyttöön. [38]

Meteor ja Livequery käyttävät tietokantaa sille toteutetun sovittimen kautta. Sovitin muun muassa toteuttaa muutosten seurannan tietokannalle sopivalta menetelmällä ja kommunikoi muutokset asiakassovelluksille DDP:n avulla. Livequeryn MongoDB-sovitin käyttää muutosten seurantaan MongoDB:n *oplog-toiminnallisuutta* (Operations Log). Oplog on erillinen MongoDB-kokoelma, johon MongoDB tallentaa tiedot kaikista tietokantaan tehdyistä muutoksista. Meteor käyttää tietokannan oplog-logia ja tunnistaa siitä sovelluksen tarvitsemat muutokset. Tämä lähestymistapa päivittää Meteor-sovelluksiin myös muutokset, jotka tehdään tietokantaan Meteor-sovelluksen ulkopuolelta, esimerkiksi MongoDB:n komentoriviltä. [38; 45]

5. SOVELLUKSEN ESITTELY

Tässä luvussa esitellään diplomityössä käsiteltävä SLURP-sovellus. Sovelluksen käyttötarkoitus ja ominaisuudet käydään läpi kohdassa 5.1, ja kohdassa 5.2 annetaan tarkempi kuvaus sovelluksen arkkitehtuurista ja toteutuksen yksityiskohdista.

5.1 Käyttötarkoitus ja ominaisuudet

SLURP (Suomen Liikunta- ja UlkoiluReittiPalvelu) on web-sovellus, jonka tärkein tarkoitus on auttaa sen käyttäjiä löytämään uusia ulkoilu- ja liikuntamahdollisuuksia. Tämä pyritään saavuttamaan esittämällä liikuntapaikat käyttäjälle kartalla, tarjoamalla työkaluja niiden hakemiseen sekä antamalla niihin liittyvää hyödyllistä tietoa. Tässä kohdassa annetaan yleiskuva sovelluksesta ja sen ominaisuuksista.

5.1.1 Projektin tausta

Sovellus toteutettiin Vincit Oy:n kesäprojektina Summer@Vincity-hankkeessa. Vincit on Tamperelainen ohjelmistotalo, joka on keskittynyt asiakkailleen räätälöityjen ohjelmistojen ketterään kehitykseen. Suurin osa projekteista liittyy web- tai mobiilisovelluksiin tai sulautettuihin järjestelmiin. Näiden lisäksi Vincit tarjoaa myös konseptointia, käyttöliittymäsuunnittelua ja konsultointia. Kirjoitushetkellä Vincitillä on noin 130 työntekijää, joista suurin osa työskentelee Tampereella ja loput Helsingissä. Vincit on saanut runsaasti tunnustusta lupaavana yrityksenä ja hyvänä työnantajana, ja se on muun muassa valittu Suomen parhaaksi työpaikaksi Great Place to Work -kisassa vuosina 2014 ja 2015. [22; 54]

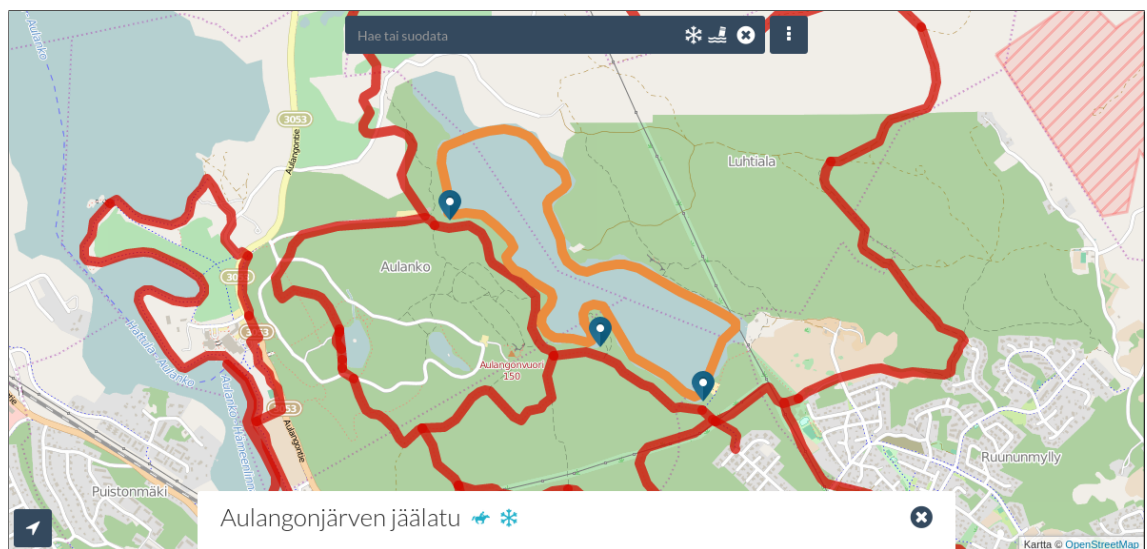
Kesäprojektin toteutustiimiin kuului projektinvetäjä ja neljä opiskelijaa, jonka lisäksi projektissa avusti Vincitin graafikko. Projektissa tehtiin yhteistyötä Avoimen Tampereen, Valomon, Aamulehden ja Demolan kanssa. Sovelluksen perusidea oli päätetty jo ennen projektia Aamulehden järjestämän kyselyn perusteella, mutta tarkemmasta suunnittelusta ja toteutuksesta tiimi vastasi itse. Varsinaista asiakasta projektilla ei ollut, vaan sovellus oli tarkoitus julkaista kesän päätteeksi avoimena lähdekoodina. Tämä suunnitelma myös toteutettiin, ja sovelluksen lähdekoodit ovat saatavilla GitHub-palvelussa Vincit-organisaation alla¹.

¹<https://github.com/Vincit/slurp>

5.1.2 Käyttöliittymä

Erilaiset liikuntamahdollisuudet esitetään sovelluksessa kohteina, jotka on jaettu kahteen tyyppiin: reitteihin ja paikkoihin. Reittejä ovat esimerkiksi luontopolut. Ne kulkevat kahden pisteen välillä ja ovat pääsääntöisesti yksihaaraisia. Paikat edustavat pistemäisiä kohteita tai pienehköjä alueita, ja niihin kuuluvat esimerkiksi kuntosalit ja uimarannat.

Koska kohteiden sijainnin selvittäminen on käyttäjille oleellista, päätettiin karttanäkymä ottaa sovelluksen käyttöliittymän pohjaksi. Kartta peittää koko selaimen ikkunan alueen ja muut käyttöliittymän elementit on aseteltu sen päälle. Ruutu-kaappaus sovelluksen käyttöliittymästä on esitetty kuvassa 5.1. Kuvassa sovelluksessa valittuna oleva reitti on korostettu oranssilla värillä ja sen tiedot ovat avattuna käyttöliittymän alalaidassa.



Kuva 5.1: SLURP-sovelluksen käyttöliittymä.

Sovelluksen käyttöliittymä on responsiivinen, jotta sen käyttö olisi sujuvaa riippumatta ikkunan koosta. Erityistä huomiota kiinnitettiin sovelluksen käyttöön mobiililaitteilla. Käyttöliittymän elementit on pyritty asettelemaan siten, että ne ovat helposti käytettävissä niin työpöytä- kuin mobiiliselaimilla. Lisäksi reittien ja paikkojen editointitilassa on mobiililaitteilla käytössä erillinen editori, jota on helpompi käyttää kosketusnäytöllä.

5.1.3 Kohteiden tiedot

Kohteiden sijainti selviää kartalta, mutta niiden tarkempia tietoja varten sovelluksessa on kaksi näkymää. Kuvan 5.1 alalaidassa näkyy kohteen kurkistusnäky, jossa näytetään ainoastaan kohteen nimi ja kategoriat. Kurkistusnäky näytetään

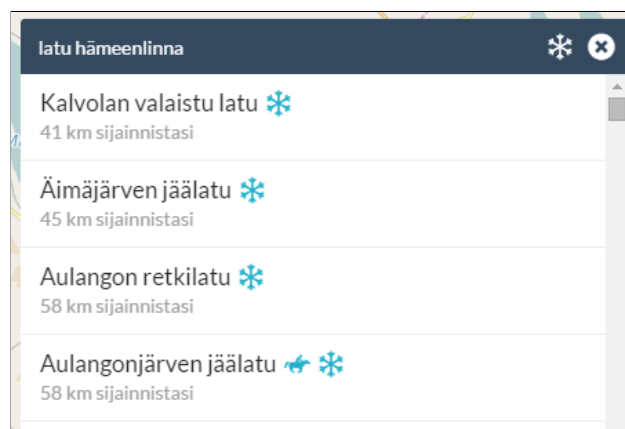
kun käyttäjä painaa kohdetta kartalla, ja sitä edelleen painamalla käyttäjä voi siirtyä kohteen tarkemmat tiedot sisältävään näkymään. Nimen ja kategorioiden lisäksi kohteen tietoihin kuuluu muun muassa reitin pituus, tiedot kohdedatan lähteestä ja datan muutoshistoria, kohteen arvostelutiedot sekä käyttäjien lisäämät kommentit ja kuvat.

5.1.4 Kohteiden hakumahdollisuudet

Kohteiden luokittelua varten sovelluksessa on kymmenen ennalta määrättyä kategoriaa. Kategorioita ovat muun muassa sisäliikunta, talvilajit ja eläinurheilu, ja kukin kohde voi kuulua useaan kategoriaan. Näitä kategorioita käytetään kohteiden etsimisen ja rajaamisen tukena. Tavallisten kategorioiden lisäksi kohteiden rajaamisessa on käytössä ”muut”-kategoria, joka käsittää kaikki kohteet joille ei ole määritetty yhtään kategoriaa.

Hakupalkki kohteiden etsimiseen sijaitsee käyttöliittymän ylälaudassa. Hakupalkkia painamalla sen alapuolelle avautuu lista kategorioista, joista käyttäjä voi valita kategoriat joihin kuuluvat kohteet kartalla näytetään. Oletuksena yhtään kategoriaa ei ole valittu, jolloin kartalla näkyviä kohteita ei rajata vaan kaikki kohteet ovat näkyvissä.

Käyttäjä voi myös syöttää hakupalkkiin hakusanan, jolloin hänelle näytetään lista hakusanaa vastaavista kohteista. Listassa olevaa kohteen nimeä painamalla sovellus siirtää kartan kohteen sijaintiin ja avaa kohteen tiedot. Hakusanojen vertailu kohdistetaan kohteen nimeen ja sijaintiin. Tulosten järjestelyn perusteena käytetään hakusanojen osuvuutta, etäisyyttä käyttäjän nykyisestä sijainnista sekä kohteiden suosiota. Kuvassa 5.2 on esimerkki avatusta hakupalkista käyttäjän syötettyä hakusanan.

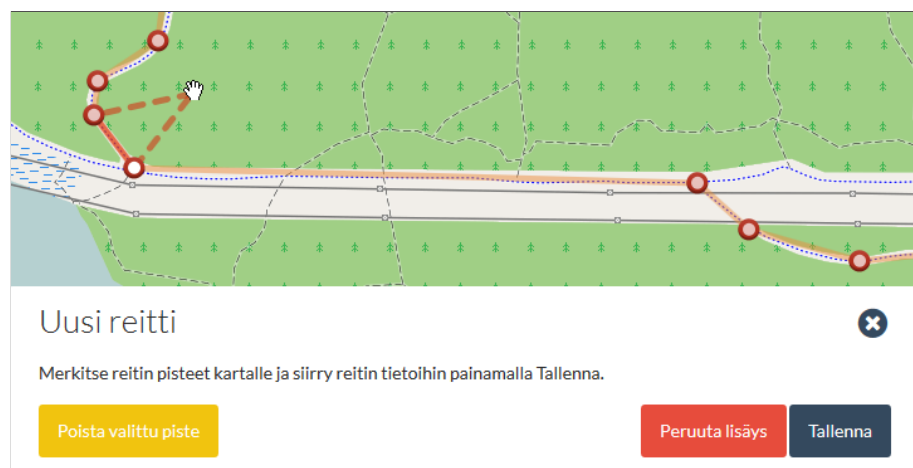


Kuva 5.2: Hakutoiminnon käyttöliittymä.

5.1.5 Kohdedatan kerääminen

Jotta palvelusta olisi hyötyä sen käyttäjille, on oleellista että siinä on riittävästi korkealaatuisia sisältöä. Erityisesti tämä tarkoittaa reitti- ja paikkatietoja, mutta myös muut kohteisiin liittyvät tiedot kuten käyttäjien julkaisemat kuvat ja kommentit ovat tärkeitä.

Jotta palveluun saataisiin mahdollisimman kattavasti kohdetietoa, on tiedon keräämiseen käytössä kaksi eri tapaa: joukkoistaminen ja avoimien data-lähteiden käyttö. Joukkoistamisella pyritään saavuttamaan kohdetietojen ajantasaisuus sekä myös harvinaisempien kohteiden kertyminen palveluun, ja sitä tuetaan tarjoamalla käyttäjille mahdollisuus luoda palveluun itse lisää kohteita. Myös vanhojen kohteiden muokkaaminen on mahdollista, joten esimerkiksi reittejä voidaan tarkentaa. Kohteita voi joko luoda suoraan sovelluksen kartalla tai antamalla sovellukselle GPS-dataa sisältävän GPX-tiedoston² josta reitti generoidaan. Vaikka sovellus tukee myös monihaaraisia reittejä, käyttöliittymän yksinkertaistamiseksi käyttäjien lisäämät reitit voivat olla ainoastaan yksihaaraisia. Ruutukaappaus reitin muokkauskäyttöliittymästä on esitetty kuvassa 5.3.



Kuva 5.3: Reitit muokkauskäyttöliittymä.

Joukkoistamiseen liittyy kuitenkin omat riskinsä. Koska käyttäjät voivat muokata kaikkia kohteita, saattaisivat pahansuovat käyttäjät pyrkiä kumoamaan muiden tekemiä muutoksia tai lisäämään palveluun olemattomia tai muilla tavoin asiattomia kohteita. Tämän ehkäisemiseksi kohteita on mahdollista poistaa ja kaikki muutokset ja kohteiden poistot arkistoidaan. Näistä arkistointimerkinnöistä muodostetaan kohteen muutoshistoria, jonka käyttöliittymän kautta muutoksia voi tarkastella ja kohteen tilan voi palauttaa aiempaan versioon. Myös poistetut kohteet on mahdollista palauttaa. Kohteen muutoshistorian avulla käyttäjät voivat myös vertailla

²<http://www.topografix.com/gpx.asp>

kohteeseen tehtyjä muutoksia suoraan kartalla, jolloin esimerkiksi reitin pisteisiin tehdyt muutokset on helppo hahmottaa.

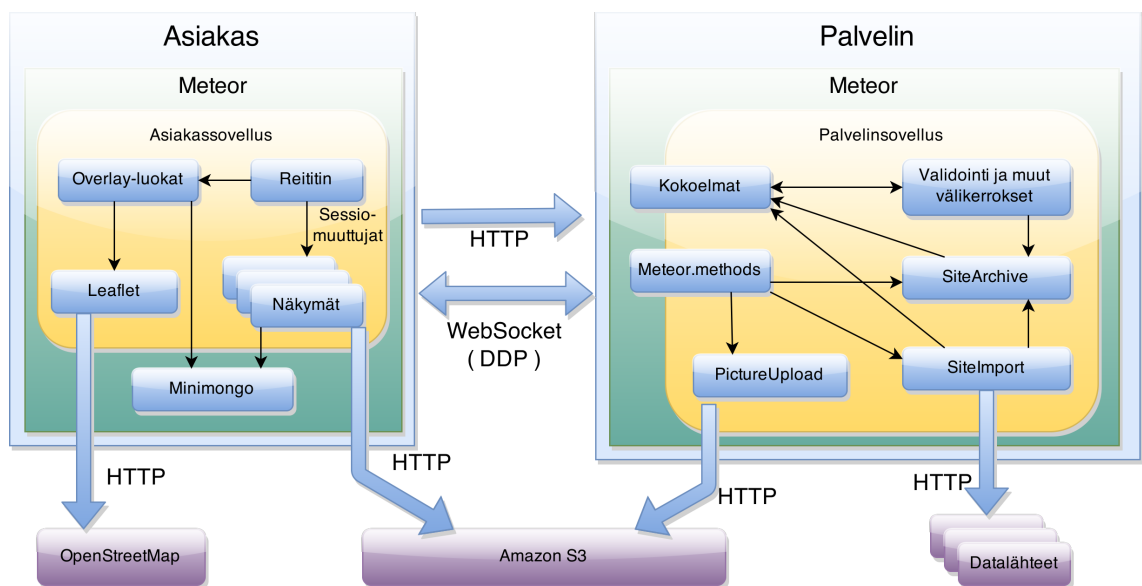
Pelkällä joukkoistamisella datan kertyminen palveluun olisi hidasta erityisesti sovelluksen käyttöönoton alkuvaiheessa. Käyttäjää on vaikeampi houkutella palveluun, jos siinä ei ole riittävästi sisältöä. Tästä syystä palvelussa hyödynnetään myös avoimia datanlähteitä joista noudetaan sovellukseen reitti- ja paikkatietoja. Sovellus on siis näitä valmiita datalähteitä hyödyntävä koostepalvelu. Näin palveluun saatiin jo alkuvaiheessa riittävästi dataa jotta sen käyttö olisi mielekästä.

5.1.6 Ylläpitäjien työkalut

Sovellukseen toteutettiin myös yksinkertainen käyttöliittymä ylläpitäjiä varten. Koska sovellukseen voidaan lisätä päivässä runsaasti uusia kohteita ja uusimpien kohteiden löytäminen kartalta on hankalaa, on ylläpitökäyttöliittymässä listattuna uusimmat kohteet. Näin asiattomat kohteet voidaan huomata nopeammin ja ne voidaan poistaa. Vastaava listaus on tehty myös käyttäjien lisäämille kommentteille, joiden poisto on myös mahdollistettu ylläpitäjille. Ylläpitäjät voivat myös tarvittaessa evätä käyttäjältä pääsyn sovellukseen mielivaltaiseksi ajaksi.

5.2 Sovelluksen arkkitehtuuri

SLURP-sovellus on toteutettu Meteor-sovelluskehityksellä, josta on tällä hetkellä käytössä versio 0.6.4. Yleiskuva sovelluksen arkkitehtuurista on esitetty kuvassa 5.4. Seuraavissa alakohdissa käydään läpi sovelluksen tietomalli sekä tärkeimpiä osia sovelluksen moduuleista.



Kuva 5.4: SLURP-sovelluksen arkkitehtuuri.

Kokoelmat `Localities` ja `Categories` toimivat lähinnä sovelluksen hakutoimintojen apuna. Niihin kuuluvan tiedon avulla kohteita voidaan etsiä ja rajata paikannimien tai ennalta määrättyjen kategorioiden perusteella. Paikkatiedot on eriytetty kohteista jotta niitä voitaisiin helposti käyttää esittämään kohteiden määrä tietyllä alueella.

Käyttäjien tekemät kohteiden arvostelut tallennetaan `SiteRatings`-kokoelmaan. Näitä suosiopisteitä käytetään muun muassa hakutulosten järjestyksessä.

Kohteiden arkistointi on toteutettu `ArchivedSites`- ja `ArchivedSiteData`-kokoelmien avulla. `ArchivedSite`-dokumentit sisältävät muutokseen liittyvän metadatan. Varsinainen kohdedata on tallennettu `ArchivedSiteData`-kokoelman dokumentteihin, joiden rakenne on identtinen `Site`-dokumenttien kanssa. Metadata tallennetaan erilliseen kokoelmaan koska sama kohdedata voi liittyä useaan arkistointimerkintään. Jos kohde esimerkiksi poistetaan ja sen jälkeen palautetaan, kohdedata ei muutu mutta molemmista tapahtumista tehdään arkistointimerkintä, joilla on eri metadata.

Sovelluksen suorituskyvyn parantamiseksi tietomalli sisältää jonkin verran kenttiä joiden arvo voitaisiin laskea ohjelman suorituksen aikana. Tällaisia ovat esimerkiksi reitin pituus ja paikkatietoon liittyvien kohteiden määrä. Koska nämä tiedot vievät hyvin vähän tilaa ja niitä tarvitaan usein, on niiden tallentamisesta selvää hyötyä sovelluksen suorituskyvyn kannalta.

5.2.2 Kartta ja kohteiden esittäminen

Sovelluksessa karttojen hallintaan käytetään `Leaflet`-kirjastoa. `Leaflet` on avoimen lähdekoodin `JavaScript`-kirjasto, joka on suunniteltu toimimaan hyvin myös mobiiliselaimilla [29]. Projektin aikana osoittautui, että sen käyttö ja integroitavuus järjestelmään oli helppoa. Kirjastoa käytetään karttojen piirtämiseen sekä niillä navigointiin, kuten liikkumiseen ja zoomaukseen. Sitä hyödynnetään apuna myös kohteiden piirtämisessä ja editoinnissa.

`Leaflet`in avulla käytetään `OpenStreetMap`-projektin karttoja. `OpenStreetMap` on joukkoistamiseen perustuva suosittu karttakirjasto, jonka kartat ovat avointa dataa [49]. Sen karttoja päätettiin käyttää projektissa koska ne ovat laadukkaita ja ajantasaisia ja niiden käyttö `Leaflet`in kanssa oli helppoa. `Leaflet`in avulla olisi ollut mahdollista käyttää myös muita karttakirjastoja, jolloin käyttäjille olisi voitu tarjota käyttöliittymä karttakirjaston valintaan. Tälle ei kuitenkaan projektin aikana nähty tarvetta.

Kohteet lisätään asiakassovelluksessa kartalle `PlaceOverlay`- ja `RouteOverlay`-luokkien kautta, jotka vastaavat paikkojen ja reittien hallinnasta kartalla. Näillä luokilla on paljon yhteistä toiminnallisuutta, joten ne periytyvät yhteisestä

SiteOverlay-kantaluokasta. Kumpikin luokka etsii tietokannasta hallinnoimansa kohteet ja lisää ne kartalle Leafletin rajapintojen avulla. Lisäksi luokat muun muassa korostavat käyttäjän valitseman kohteen kartalla ja reagoivat zoomaustason muutoksiin.

Kartalla näkyvien kohteiden rajausta tehdään jo tilattaessa kohdetietueet palvelimelta. Tämän ansiosta asiakassovelluksen tietokannassa on ainoastaan kartalla näkyvät kohteet, mikä pienentää siirrettävän datan määrää ja yksinkertaistaa asiakassovelluksessa tehtäviä tietokantakyselyjä. Tilatessaan kohdetietueita asiakassovellus antaa palvelimelle parametrina käyttäjälle näkyvän kartta-alueen koordinaatit. Palvelin etsii tietokannasta alueella olevat kohteet käyttämällä MongoDB:n \$geoWithin-operaattoria ja lähettää asiakassovellukselle nämä kohteet. Rajausta päivitetään aina zoomaustason muuttuessa tai kun käyttäjä liikuttaa karttaa, jolloin kartalla näkyvät kohteet ovat aina tilattuna.

5.2.3 Näkymät ja sivupohjat

Meteorissa näkymiä on helppo sisällyttää toisten näkymien sisään. Sovelluksessa näkymät muodostavat hierarkian, jonka ylimmällä tasolla on main, jonka sivupohja on esitetty ohjelmassa 5.1. Tässä sivupohjassa luodaan sivun ylimmän tason head- ja body-elementit ja sisällytetään body-elementtiin muut sivupohjat. Sivupohjassa ei ole määritelty alisivupohjien sijoittelua päänäkymässä, vaan ne vastaavat itse asettelustaan CSS-tyyliin avulla.

```
1 <head>
2   <meta charset="utf-8">
3   <title>Liikunta- ja ulkoilureitit</title>
4 </head>
5
6 <body>
7   {{> map}}
8   {{> topView}}
9   {{> detailView}}
10  {{> dialogContainer}}
11  {{> mainMenu}}
12  {{> lowerLeftToolbar}}
13  {{> loadIndicator}}
14 </body>
```

Ohjelma 5.1: Ote main.html-sivupohjasta.

Ylimmän tason sivupohjasta huomataan että kaikki siihen kuuluvat alisivupohjat sisällytetään aina sivuun. Kaikkia elementtejä ei kuitenkaan käytetä joka näkymässä, vaan sivupohjan piirtämisen tarpeellisuuden vaatimat tarkistukset on sisällytetty niihin itseensä. Tämä tehdään Meteorin reaktiivisten sessiomuuttujien avulla.

Esimerkki yhdestä päänäkymässä esiintyvistä sivupohjasta on esitetty ohjelmassa 5.2. Vaikka `dialogContainer`-sivupohja sisällytetään aina päänäkymään, sen sisältö luodaan vain jos `showDialog`-sessiomuuttuja on asetettu todeksi. Myös dialogi-ikkunan sisällä esitettävän sisällön valinta tehdään samaan tapaan sessiomuuttujan `currentPrimaryView`-avulla.

```
1 <template name="dialogContainer">
2   {{#if sessionValueTrue "showDialog"}}
3     <div class="dialog-container">
4       <button class="close-button icon-remove-sign" />
5       {{#if sessionValueEquals "currentPrimaryView" "login"}}
6         {{> login}}
7       {{/if}}
8       {{#if sessionValueEquals "currentPrimaryView" "register"}}
9         {{> register}}
10      {{/if}}
11    {{/if}}
12 </template>
```

Ohjelma 5.2: Dialogi-ikkunan sivupohja `dialogContainer.html`.

Piirtojen ehdollisuustarkistukset sisällytetään alisivupohjiin tehokkuussyistä. Jos ohjelman 5.2 tarkastelu riviltä 2 siirrettäisiin ohjelman 5.1 pääsivupohjaan, sessiomuuttujan `showDialog` muuttuminen aiheuttaisi koko pääsivupohjan uudelleenpiirtämisen. Tällöin myös kaikki muut alisivupohjat luotaisiin tarpeettomasti uudelleen.

Näkymien responsiivisuuden toteutuksessa hyödynnetään Twitterin Bootstrap-kirjastoa [53]. Bootstrapin avulla saatiin toteutettua käyttöliittymän elementtien skaalaus ja asetelu ikkunan koon muuttuessa. Tämän lisäksi joidenkin näkymien toiminnallisuutta muokataan ohjelmallisesti näyttökoon mukaan. Esimerkiksi kohteiden editointitilan mobiilikäyttöliittymä aktivoidaan ohjelmallisesti jos käyttäjän havaitaan käyttävän sovellusta mobiililaitteella.

5.2.4 Reititys

Tavallisesti Meteoria käytetään yhden sivun sovellusten luomiseen, eikä se täten tarjoa työkaluja URL-osoitteiden hallintaan. Sovelluksessa oli kuitenkin tärkeää pystyä viittaamaan esimerkiksi kohteisiin URL-osoitteiden avulla. Tätä varten sovellukseen toteutettiin yksinkertainen reititin `page.js`-kirjaston⁴ avulla.

Reitittimen tehtävänä on reagoida URL-osoitteiden muutoksiin ja alustaa näkymät niiden perusteella. Tähän kuuluu edellisen näkymän tilan purkaminen tarvittavilta osin sekä uuden näkymän tilan alustaminen. Tilan määrittämisessä käytetään

⁴<https://github.com/tmeasday/page.js>

paljon sessiomuuttujia, joiden avulla ohjataan näkymiin kuuluvien sivupohjien piirtymistä.

Ohjelmassa 5.3 on yksinkertaistettu esimerkki, jossa on salasanan unohtus-näkymään reitittämiseen tarvittavat osat. Kun käyttäjä siirtyy sovelluksen sisällä URL-osoitteeseen `/forgot-password`, reititin kutsuu ensin funktiota `reset` jossa resetoidaan sovelluksen tilaan vaikuttavat sessiomuuttujat. Kun tila on resetoitu, `reset`-funktioista kutsutaan `forgotPassword`-funktioita joka alustaa tarvittavat sessiomuuttujat salasanan unohtusnäkyä varten.

```
1 Meteor.startup(function() {
2   pagejs.base('');
3   pagejs('/forgot-password', reset, forgotPassword);
4 });
5
6 function reset(ctx, next) {
7   Session.set('currentPrimaryView', false);
8   Session.set('showDialog', false);
9
10  next();
11 }
12
13 function forgotPassword(ctx) {
14   Session.set('showDialog', true);
15   Session.set('currentPrimaryView', 'forgotPassword');
16 }
```

Ohjelma 5.3: Salasanan palautus -näkyman reititys.

5.2.5 Kuvat

Käyttäjien lataamien kuvien tallentamiseen käytetään Amazonin S3-pilvipalvelua⁵. Käyttäjä lataa kuvan asiakassovelluksessa, joka lähettää ne palvelimelle. Palvelin tarkistaa kuvan koon ja tyyppin ja tarvittaessa pienentää sitä. Kuvat lähetetään S3:en, josta asiakassovellus lataa ne tarvittaessa. Pilvipalvelun käyttö kuvien tallentukseen vähentää sovelluksen oman palvelimen kuormaa ja skaalautuu hyvin myös suurillekin käyttäjämäärille.

5.2.6 Avoimet datalähteet

Asiakassovellus ei käytä avoimia datalähteitä suoraan. Sen sijaan palvelinsovellus lataa kohdedatan avoimista datalähteistä ja luo sen perusteella kohteet sovelluksen omaan tietokantaan. Näin datalähteitä ei tarvitse käyttää jatkuvasti, mikä vähentää datalähteiden kuormitusta ja pienentää viivettä. Kun kohteet ovat sovelluksen

⁵<http://aws.amazon.com/s3>

omassa tietokannassa, myös Meteorin reaktiivisten ominaisuuksien hyödyntäminen niiden kanssa on huomattavasti helpompaa.

Sovelluksessa hyödynnetään kolmea avointa datalähdettä, jotka kaikki julkaisevat datansa JSON-muodossa. Kaikki datalähteet tarjoavat REST-rajapinnan jonka kautta tiedot pystytään hakemaan. Kaksi datalähteistä käyttää GeoServer-palvelinta⁶, joten niiden rajapinta on melko lähellä toisiaan.

Datalähteille on määritetty yksi konfiguraatiotiedosto, joka sisältää datalähteisiin liittyvät yleiset tiedot kuten niiden URL-osoitteet ja säännöt datalähteistä saatujen kohteiden kategorioiden määrittämiseen. Kaikkea käsittelyä ei kuitenkaan ole määritelty konfiguraatiotiedostossa, vaan jokaiselle datalähteelle on toteutettu oma JavaScript-käsittelijänsä. Tämä nähtiin helpompana ratkaisuna kuin monimutkaisemman konfiguraation kehittäminen, koska datalähteitä on vähän ja niiden JSON-datan formaatit eroavat toisistaan huomattavasti.

Kohdedatan luominen datalähteistä tehdään kahdessa vaiheessa. Kaikki datalähteet käsitellään rinnakkain ja niistä luodaan kohteet, minkä jälkeen kohteisiin linkitetään paikkatiedot. Paikkatiedot päivitetään erikseen, koska ne noudetaan erilliseltä rajapinnalta jolta paikkatietoja voidaan kysyä ainoastaan yksitellen. Jos paikkatiedot päivitetäisiin kohdedatan luonnin yhteydessä, hidastuisi kohdedatan käsittely huomattavasti. Kohteet toimivat sovelluksessa hyvin myös ilman paikkatietoja, joten tästä ei ole suurta haittaa. Suurimmat ongelmat paikkatietojen puuttumisesta ilmenevät hakutoiminnossa, jossa kohdetta ei voi etsiä paikannimien perusteella ennen kuin sen paikkatiedot on asetettu.

Kohteet noudetaan datalähteistä palvelinsovelluksen käynnistyessä jos sen tietokanta on tyhjä, sekä ajastetusti viikoittain jotta myös uudet kohteet päivittyisivät sovellukseen. Lisäksi ylläpitäjillä on mahdollisuus käynnistää päivitys manuaalisesti. Tälle ei ole toteutettu erillistä käyttöliittymää, mutta noutaminen voidaan käynnistää kirjautumalla sovellukseen ylläpitäjänä ja suorittamalla selaimen komentoriviltä noutamisen käynnistävä etäproseduurikutsu.

⁶<http://geoserver.org/>

6. TOTEUTUSPROSESSI

Tässä luvussa käydään läpi sovelluksen toteutusprosessia sekä projektiryhmän työmenetelmiä. Projektin etenemistä käsitellään kohdassa 6.1, kohdassa 6.2 käydään läpi sovelluksen testatusta projektin aikana ja kohdassa 6.3 esitellään projektissa hyödynnettyjä työmenetelmiä.

6.1 Projektin eteneminen

Vincit on ketterään kehitykseen erikoistunut ohjelmistotalo, joten sovelluksen kehittäminen iteratiivisesti ketterän kehityksen periaatteiden mukaisesti oli luonteva lähestymistapa. Koska sovellukselle ei oltu asetettu tiukkoja asiakasvaatimuksia vaan lähtökohtana oli ainoastaan pääpiirteittäinen sovellusidea, olisi sovelluksen konseptointi ja idean jalostaminen välttämätöntä myös kehityksen aikana. Myös tämä puolisi iteratiivisuutta. Näin ollen oli järkevää toteuttaa ensin oleelliset ominaisuudet joita varmasti tullaan käyttämään lopullisessa sovelluksessa. Uusia ominaisuuksia lisättiin sovellukseen kehityksen edetessä, ja vanhoja paranneltiin palautteen ja sisäisestä testauksesta saatujen kokemusten perusteella.

Iteratiivisuutta toteutettiin projektissa julkaisemalla sovelluksesta väliversioita. Versiot toteutettiin vaihtelevan pituisten *sprinttien* (sprint) eli kehitysjaksojen aikana, jotka toimivat myös projektinhallinnan pidemmän aikavälin työkaluna. Sprinttien idea projektissa oli mukailtu scrumista, jossa sprintit ovat kiinteän mittaisia ja kestävät yleensä kaksi tai kolme viikkoa [28, s. 109-111]. Kunkin sprintin aluksi sovittiin aloitettavaan versioon toteutettavat ominaisuudet ja asetettiin aikataulutavoite jota päivitettiin kehityksen edetessä. Sprintin lopuksi sovellus päivitettiin julkiselle testipalvelimelle kokeiltavaksi. Julkaisemalla väliversioita sovelluksesta saatiin siitä kerättyä palautetta jo kehitysvaiheen aikana, mikä antoi ideoita uusille ominaisuuksille ja vanhojen parantamiseen.

Projektin aikana sovelluksesta julkaistiin kuusi versioita. Versiot, niiden sprinttien kestot ja kuhunkin versioon toteutetut tärkeimmät ominaisuudet on listattu taulukossa 6.1.

Ensimmäisen version kehitys aloitettiin sovellusidean tarkentamisella ja konseptoinnilla. Lähtökohtana oli idea liikuntapaikkojen löytämisen mahdollistavasta sovelluksesta. Koska liikuntapaikkojen sijoittaminen kartalle nähtiin oleellisena toiminnallisuutena, päätettiin karttanäkymä ottaa koko sovelluksen pohjaksi ja sijoittaa

Versio	Aloitus	Kesto	Tärkeimmät ominaisuudet
Versio 1	27.5	23 työpäivää	Projektipohja, kohteet kartalla, rekisteröityminen ja sisäänkirjautuminen, kohteiden yleistiedot, yksinkertainen hakutoiminto, kohteiden lisäys ja muokkaus.
Versio 2	28.6	9 työpäivää	Facebook ja Google -kirjautuminen, kohteiden luonti GPX-datasta, kohteiden editointi mobiililaitteilla.
Versio 3	11.7	16 työpäivää	Kohteiden arkistointi tehtäessä muutoksia, arkistoidun version palauttaminen, kuvien liittäminen kohteisiin, ylläpitäjän työkalut, kohteiden luonti avoimesta datasta.
Versio 4	2.8	9 työpäivää	Haku paikannimien perusteella, kohteiden arvostelu, reaktiivisuuden optimointia.
Versio 5	15.8	3 työpäivää	Gravatar-kuvat käyttäjille, avoimen datan päivitys viikoittain.
Versio 6	20.8	8 työpäivää	Porttikiellon asettaminen käyttäjälle, käyttäjien ja kohteiden top-listat, sovelluksen julkaisu avoimena lähdekoodina.

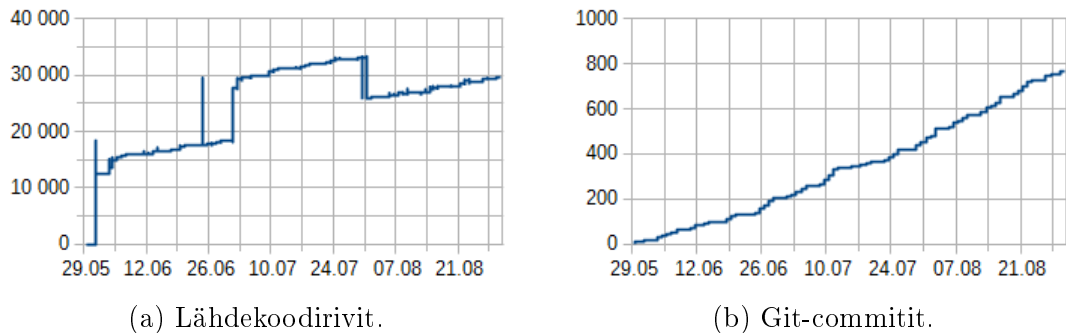
Taulukko 6.1: Ohjelmasta julkaistut versiot ja niissä lisätyt ominaisuudet.

käyttöliittymä sen päälle. Oleellisena osana sovellusta olisi siis toimiva karttanäkymä, sekä kohteiden sijoittaminen siihen. Ensimmäiseen versioon toteutettiin nämä ominaisuudet ja niiden lisäksi muita oleellisia toimintoja, kuten käyttäjien rekisteröinti ja sisäänkirjautuminen, sekä ensimmäiset versiot kohteiden hakutoiminnosta ja kohteiden tietojen tarkastelusta.

Vaikka kaikilla projektin jäsenillä oli kokemusta web-ohjelmoinnista, ei Meteor-sovelluskehys ollut tuttu kenellekään projektinvetäjää lukuun ottamatta. Tämä luonnollisesti hidasti sovelluksen kehitystä projektin alkupuolella, ja ensimmäisestä sprintistä tulikin projektin pitkäkestoisin. Sovelluskehityksen tuntemattomuus aiheutti myös pidemmän aikavälin seurauksia projektissa, sillä sovelluksen rakennetta ja tietomallia muokattiin useaan kertaan kun hyvät käytännöt alkoivat hahmottua. Varsinkin toisen ja kolmannen sprintin aikana sovellusta refaktoroiitiin runsaasti.

Meteor-sovelluskehys ehti projektin aikana päivittyä useampaan kertaan. Alussa versio oli 0.6.3.1 ja tästä päivitettiin vielä versioon 0.6.4, jossa pysyttiin projektin loppuun asti. Päivityksiä tuli vielä tämän jälkeenkin, mutta versiossa 0.6.5 Meteorin pakettijärjestelmä uudistui. Tämä aiheutti hieman ongelmia, sillä tuolloin ei vielä ollut saatavilla uusia versioita kaikista projektissa käytetyistä kirjastoista. Näin ollen oli perusteltua olla ottamatta uusinta päivitystä käyttöön projektin aikana.

Sovelluksen konseptointia jatkettiin myös ensimmäisten versioiden jälkeen. Vaikka tärkeimmät ominaisuudet sovellukseen toteutettiin projektin alkupuolella, oli sovelluksen koon kasvu ja kehityksen eteneminen melko tasaista koko projektin ajan. Tämä voidaan havaita sovelluksen lähdekoodirivien ja tehtyjen Git-commitien määrästä projektin edetessä, jotka on esitetty kuvassa 6.1. Muutamat suuret vaihtelut kuvassa 6.1a näkyvässä lähdekoodirivien määrässä johtuivat projektiin liittyneiden kolmannen osapuolen kirjastojen lisäämisestä tai poistamisesta.



Kuva 6.1: Sovelluksen lähdekoodirivien ja Git-commitien lukumäärät.

6.2 Testaus

Projektin alussa Meteor oli vielä varhaisessa kehitysversiossa eikä sille ollut vielä tuolloin saatavilla monipuolista testauskehystä. Automaattisten testien toteuttaminen olisi vaatinut oman testikehyksen toteuttamista, mikä olisi vaatinut suunnittelua ja toteutusaikaa. Tästä johtuen sovelluksen testaus oli manuaalista, eikä automaattisia testejä ollut. Manuaalinen testaus ei kuitenkaan ollut millään tavalla formalisoitua, eikä testattavia asioita oltu listattu tai muuten dokumentoitu. Lähtökohtaisesti kukin kehittäjä testasi oman lähdekoodinsa sekä ne ohjelman osat, joihin hänen tekemänsä muutokset mahdollisesti vaikuttavat.

6.3 Työtavat

Projektissa hyödynnettiin Kanban-menetelmää. Kanban on lean-ajattelun mukainen lähestymistapa ketterään kehitykseen, ja sen olennaisia ideoita ovat työnkulun visualisointi Kanban-aulun ja -korttien avulla, työn alla olevien tehtävien rajoittaminen, sekä tehtävien keston seuranta ja arviointi [28, s. 112-116]. Kanban-taulu on jaettu sarakkeisiin, jotka kuvaavat tehtävän etenemistä. Tehtävät kirjataan taululle sijoitettaviin kortteihin ja niitä siirretään sarakkeissa eteenpäin tehtävän etenemisen mukaan. Kussakin sarakkeessa olevien tehtävien määrää voidaan rajoittaa, mikä auttaa hallitsemaan työn alla olevien tehtävien määrää. Projektissa esimerkiksi toteutuksessa olevien ominaisuuksien määrä oli rajoitettu yhteen jokaista kehittäjää

kohden. Yhteen ominaisuuteen keskittyi kerrallaan enintään kaksi kehittäjää, mutta tämä oli sovittu erikseen eikä sitä oltu rajoitettu Kanbanin avulla. Kuvassa 6.2 on esitetty projektin Kanban-taulu ensimmäisen sprintin aikana.



Kuva 6.2: Projektin Kanban-taulu ensimmäisen sprintin aikana.

Työn etenemisen seuranta Kanban-taulun avulla oli projektin hallinnassa tärkeässä asemassa. Toinen olennainen käytäntö olivat päivittäin työpäivän aluksi pidettävät palaverit, joissa käytiin läpi, mitä kukin oli tehnyt, parhaillaan tekemässä ja aloittamassa seuraavaksi. Samalla käytiin läpi mahdollisia ongelmia sekä päivitettiin Kanban-taulu mikäli se ei ollut ajan tasalla.

Lähdekoodin laadun varmistamiseksi ja virheiden ennaltaehkäisemiseksi projektissa käytettiin lähdekoodin katselmointia. Katselmoinnissa hyödynnettiin Git-pohjaista Gerrit-järjestelmää, joka on Googlen kehittämä avoimena lähdekoodina julkaistu katselmointityökalu [21]. Gerritissä lähdekoodin katselmointi tapahtuu Git-commit kohtaisesti ja kommentit voi kohdentaa suoraan asianomaisiin riveihin. Projektissa jokainen Git-commit katselmoitiin kahden muun kehittäjän toimesta. Koska projektiryhmäläisten työpisteet olivat toistensa vieressä, oli katselmoinnissa kertyneistä kommentteista helppo keskustella myös kasvotusten.

7. ARVIOINTI

Tässä luvussa arvioidaan reaktiivisuuden hyödyllisyyttä ja vaikutuksia projektiin. Projektin onnistumista arvioidaan kohdassa 7.1, sovelluksen kehityksen aikana kohdattuja ongelmia kohdassa 7.2 ja reaktiivisuuden hyötyjä kohdassa 7.3.

7.1 Projektin onnistuminen

Kokonaisuudessaan projekti onnistui hyvin. Sovelluksesta saatiin tavoitteiden mukainen toimiva kokonaisuus projektin aikataulun puitteissa. Perustoiminnallisuuden lisäksi sovellukseen ehdittiin toteuttamaan myös useita lisäominaisuuksia, jotka paransivat sovelluksen käyttöarvoa. Projektiryhmä oli myös itse tyytyväinen lopputulokseen.

Testauksen kannalta projektissa oli kuitenkin kehittämisen varaa. Manuaalinen testaus oli useimmiten riittävää, mutta testattavien asioiden tarkempi määrittely olisi ollut hyödyllistä erityisesti jatkokehitettäessä jonkun toisen kehittäjän toteuttamaa ominaisuutta. Vaikka sovelluksesta löydettiin ja korjattiin kehityksen aikana useita virheitä, mihinkään tiettyyn ohjelman osaan liittyviä toistuvia virheitä ei kuitenkaan tullut vastaan. Osa löydetyistä virheistä olisi kuitenkin varmasti voitu huomata aiemmin paremman testauksen avulla. Lisäksi automaattiset testit olisivat varmasti helpottaneet myös sovelluksen tulevaa kehitystä.

Projektissa sovelletut työmenetelmät soveltuivat projektiin hyvin. Kanbanmenetelmän ja päivittäisten palaverien yhdistelmä todettiin toimivaksi, sillä niiden avulla pystyttiin keskittymään tärkeiden ominaisuuksien toteuttamiseen ja koko kehitystiimi tiesi jatkuvasti miten projekti eteni. Myös lähdekoodin katselmointi oli toimiva käytäntö. Sen avulla sovelluksesta saatiin poistettua monta potentiaalista virhettä ennen kuin ne edes päätyivät testipalvelimelle. Lisäksi lähdekoodin katselmointi auttoi yhtenäiseen ohjelmointityyliin ja paransi kehittäjien käsitystä myös toistensa toteuttamista sovelluksen osista.

7.2 Ongelmia

Sovelluksen toteutuksen aikana tuli vastaan useita ongelmia, joista suuri osa liittyy reaktiivisuuteen tai Meteorin reaktiiviseen malliin. Näitä ongelmia käydään läpi tässä kohdassa.

7.2.1 Suurten datamäärien hallinta

Sovellusta kehitettäessä suurimmaksi ongelmaksi osoittautui sovelluksen suuret datamäärät. Erityisesti kohteet sisältävät paljon dataa. Kartalla voisi helposti näkyä satoja kohteita, ja yhdessä reitissä voi usein olla satoja pisteitä. Ongelmaa pahentaa myös se, että kartalla näkyvät kohteet saattavat vaihtua nopeasti jos käyttäjä esimerkiksi siirtyy toisen kaupungin alueelle. Näin ollen kohteiden siirtäminen asiakassovellukselle ja niiden reaktiivinen hallinta on raskasta.

Tämä ongelma tuli erityisen hyvin esiin projektin puolivälin tienoilla, kun reittien pisteet olivat vielä tallennettu omaan kokoelmaansa tietokannassa. Erillisen piste-kokoelman hallinta oli hankalaa, koska reitin pisteitä varten piti olla oma tilauksensa asiakassovelluksessa. Useassa kohdassa sovelluksen suoritusta oli myös varmistettava, että sekä reitin että sen pisteiden tilaukset olivat tulleet valmiiksi. Ongelma saatiin ratkaistua poistamalla erillinen piste-kokoelma ja siirtämällä reitin pisteet samaan kokoelmaan reitin kanssa. Tämä tietysti kasvatti reitti-dokumenttien kokoa, mutta yksinkertaisti tietomallia ja sovelluslogiikkaa huomattavasti.

Kun tietokannassa olevien kohteiden määrä kasvoi, kävi nopeasti ilmi myös että asiakassovellukselle ei voitu aina toimittaa kaikkia sovelluksen kohteita, vaan niitä oli pystyttävä rajaamaan. Käyttäjä pystyy itse rajoittamaan näkyviä kohteita haku- ja suodatustoimintojen avulla, mutta niiden lisäksi vain kartalla näkyvien kohteiden siirtäminen asiakassovellukselle oli luonnollinen vaihtoehto. Ongelmana oli rajauksen toteutus palvelimella. Näkyvällä alueella olevien kohteiden etsiminen jokaisen kartalla liikkumisen yhteydessä olisi ollut yksinkertaista, mutta liian raskasta ollakseen järkevä vaihtoehto.

Kehityksen alkuvaiheessa reittien pisteet oli tallennettu tietokantaan yksinkertaisina koordinaattipareina. Rajausongelmaa selvitettyä huomattiin, että MongoDB tukee geospaatialisia kyselyjä ja -indeksointia. Näiden ominaisuuksien hyödyntämistä varten reittien sijaintitiedot oli tallennettava GeoJSON-muodossa, mikä jälleen kerran lisäsi hieman dokumenttien kokoa. MongoDB:n geospaatialiset ominaisuudet mahdollistivat kuitenkin kohteiden tehokkaan hakemisen tietokannasta alueen perusteella, joten niitä päätettiin hyödyntää.

Vaikka rajauskyselyistä saatiinkin tehtyä melko tehokkaita, tehtiin niitä silti liian usein. Käyttäjät saattavat liikuttaa kartan sijaintia hyvin usein, joten rajauksen parametrit muuttuivat jatkuvasti. Jatkuva rajauksen päivittyminen oli ongelmallinen myös mobiilikäytön ja hitaampien nettiyhteyksien kannalta. Ennen kuin yhden rajauksen kohteet oli saatu kokonaan siirrettyksi asiakassovellukselle, saattoi rajaus olla jo vaihtunut. Tätä ongelmaa pyrittiin ehkäisemään noutamalla kohteet suuremmalta alueelta kuin mitä kartalla näkyi. Näin rajausparametreja ei tarvinnut aina päivittää, jos käyttäjä siirsi karttaa hieman.

7.2.2 Siirrettävän datan hallinta

Asiakassovellukselle siirrettävien kohteiden rajoittaminen aiheutti kuitenkin uusia ongelmia. Koska Meteorin asiakassovellus noutaa tietueet palvelimelta dynaamisesti, ei kaikkia tarvittavia tietoja ole välttämättä heti saatavilla esimerkiksi sovelluksen käynnistyessä. Tämä ilmiö alkoi yleistyä myös sovelluksen käytön aikana, kun kaikki kohteet eivät enää olleetkaan aina saatavilla. Ongelmia tuli esimerkiksi hakutoiminnon tulosten käsittelyssä sekä kohteen tietojen tarkastelussa.

Käytettäessä hakutoimintoa saatiin usein hakutuloksia jotka eivät vielä näkyneet kartalla. Koska itse haku tehtiin palvelimella, eivät kaikki tuloksena saadut kohteet siis vielä välttämättä olleet asiakassovelluksen Minimongo-tietokannassa. Järkevin ratkaisu oli palauttaa hakukyselyssä kaikki tarpeelliset tiedot, jotta hakutulokset voitiin esittää ja niistä pystyttiin siirtymään kohteiden sijaintiin kartalla. Hakutulokset eivät tämän jälkeen enää olleet reaktiivisia, vaan haku oli tehtävä uudestaan jotta mahdolliset uudet kohteet ilmestyivät tuloksiin.

Käyttäjä saattoi myös tarkastella tietoja kohteesta, joka oli nykyisen karttanäkymän ulkopuolella. Tarkasteltavan kohteen tiedot eivät välttämättä siirtyneet normaalin kohdetilauksen mukana, joten sitä varten lisättiin oma tilaus joka pidettiin voimassa niin kauan kuin kohde oli avattuna käyttöliittymässä. Ylimääräisiä tilauksia jouduttiin käyttämään muissakin tilanteissa, kun oli varmistettava että jokin data oli saatavilla asiakassovelluksessa. Niiden käyttäminen Meteorin mallissa oli välttämätöntä, mutta niiden hallinta monimutkaisti ohjelmaa hieman.

7.2.3 Asiakassovelluksen suorituskyky

Vaikka kaikki kohteet saataisiinkin siirrettyä asiakkaalle tehokkaasti, ei karttakirjaston suorituskyky välttämättä riitä tuhansien samanaikaisten kohteiden piirtämiseen. Liian monen kohteen yhtäaikainen näyttäminen on myös sekavaa, eikä kovin hyödyllistä käyttäjälle.

Eryityisesti reittien näyttäminen on raskasta. Tästä syystä ensimmäinen ratkaisu oli muuttaa reittien esitystapaa. Kun käyttäjä siirtyi kartassa riittävän kaukaiselle zoomaustasolle, reitin muotoa ei juuri enää hahmottanut. Kaukaisilla zoomaustasoilla reitit päätettiin korvata paikkoja vastaavilla pistemäisillä merkeillä, jotka ovat kevyitä piirtää.

Ongelmana olivat kuitenkin edelleen tilanteet, joissa kartalla näkyi esimerkiksi koko Suomen alue. Hieman samaan tapaan kuin reittienkin kohdalla päätettiin myös kaikkien kohteiden esitystapaa muuttaa riittävän kaukaisella zoomaustasolla. Yksittäiset kohteet poistettiin kartalta ja korvattiin lukuarvolla, joka kertoi kuinka monta kohdetta esimerkiksi tietyn kaupungin alueella oli. Tämä auttoi lisäksi käyttäjiä hahmottamaan millä alueilla oli paljon kohteita listattuna.

7.2.4 Meteorin reaktiivisen mallin rajoitteet

Yksi merkittävä rajoite Meteorin reaktiivisuuden toteutustavassa on sen läheinen kytkeytyminen MongoDB-tietokantaan. Vaikka Meteorin kehittäjät ovatkin luvanneet tukea myös muita tietokantoja tulevaisuudessa ja tietokantayhteyden toteutustavan pitäisi tämä periaatteessa myös mahdollistaa, vielä toistaiseksi tuki on hyvin rajoittunutta. MongoDB:n lisäksi tällä hetkellä on saatavilla ainoastaan kokeiluasteella oleva tuki Redis-palvelimelle [38].

Meteorin Livequery-toiminnallisuuden reaktiiviset ominaisuudet ovat myös paljon riippuvaisia MongoDB:n ominaisuuksista. Vastaavia ei löydy kaikista muista tietokannoista, jolloin niitä käytettäessä olisi turvaututtava vähemmän tehokkaisiin ratkaisuihin. Ainakin toistaiseksi Meteorin arkkitehtuuri siis rajoittaa tietokannan vaihtamisen mahdollisuuksia, eikä Meteorin käyttäjällä juuri ole vaihtoehtoja MongoDB:n käytölle. Muita tietokantavaihtoehtoja harkittiin myös tämän projektin alkupuolella, mutta ne jouduttiin hylkäämään koska muun kuin MongoDB:n käyttö Meteorin kanssa olisi ollut hyvin hankalaa.

7.3 Reaktiivisuudella saavutetut hyödyt

Projektissa havaitut reaktiivisuuden hyödyt voidaan jakaa sovelluksen loppukäyttäjille näkyviin ominaisuuksiin ja sovelluksen kehittäjille hyödyllisiin asioihin. Näitä käsitellään seuraavissa alakohdissa.

7.3.1 Sovelluksen käyttökokemus

Reaktiivisuudella saavutettavat käyttäjäkokemushyödyt tulevat selkeästi esiin Meteorilla toteutetussa sovelluksessa. Sivun uudelleenlatauksen tarve saadaan eliminoitua kun kaikki data päivittyy käyttäjän näkymään automaattisesti. Näin uusi tieto tulee välittömästi käyttäjän näkyville ja sovellus tuntuu dynaamisemmalta ja elävämmältä.

SLURP-sovelluksen tapauksessa reaktiivisuudesta ei kuitenkaan ollut käyttäjän näkökulmasta kovin paljon olennaista hyötyä. Kohteiden ja muun sisällön automaattinen päivittyminen käyttöliittymään on lähinnä mukava lisäominaisuus, mutta ei ole sovelluksen käyttötarkoituksen kannalta oleellista tai tarjoa käyttäjälle merkittävää lisäarvoa. Jos sovellus ei olisi reaktiivinen, se olisi edelleen käyttökelpoinen sillä reaktiivisuus ei suoraan ole osa mitään sovelluksen käyttötarkoituksen kannalta kriittistä ominaisuutta. Tällöin osa käyttöliittymän päivityksistä pitäisi tosin hoitaa muulla tavalla, mutta tämä liittyy sovelluksen toteutustekniikkaan. Sovelluksen data ei myöskään pääsääntöisesti päivity kovin usein, joten sivun uudelleenlatauksen välttämällä ei ole kovin suurta merkitystä.

7.3.2 Sovelluskehityksen helpottuminen

Kun asiakassovellus on reaktiivinen, moni toiminto joka muuten olisi toteutettava erikseen tapahtuu automaattisesti. Esimerkiksi kohteen kommentointinäkyvässä käyttäjän lisätessä kommentin olisi se lisäyksen jälkeen lisättävä myös samassa näkyvässä olevaan kommenttilistaan, tai kommenttilistan sisältämät kommentit tulisi ladata palvelimelta uudelleen. Reaktiivisuuden ansiosta kommentti ilmestyy kommenttilistaan ilman lisätyötä ohjelmoijalta.

Myös Meteorin malli toi mukanaan omat hyötynsä. Yleensä web-sovellusta kehitettäessä sovelluskehittäjän täytyy suunnitella asiakassovelluksen käyttämä palvelimen rajapinta. Meteorilla tätä ei tarvitse tehdä, sillä tietorakenteen suunnittelu ja tarvittavien tietueiden määrittely ja tilaus asiakassovelluksesta riittää. Tietueiden tilausten hallinta ei kuitenkaan aina ole triviaalia ja vaatii myös suunnittelua, mutta tämä on yleensä helpompaa ja nopeampaa kuin esimerkiksi REST-rajapinnan suunnittelu ja toteuttaminen.

Reaktiivista sovellusta toteutettaessa ohjelmoija joutuu tyypillisesti näkemään vaivaa joko reaktiivisuuden toteuttamisessa tai sen toteuttavan kirjaston integroinnissa muuhun järjestelmään. Koska Meteorissa reaktiivisuus on kiinteästi mukana sovelluskehityksen ytimessä, on reaktiivisuuden ja muun ohjelman yhteistoiminta valmiiksi suunniteltu ja toteutettu. Tämän ansiosta reaktiivisten ominaisuuksien toteuttamiseen ei tarvitse nähdä ylimääräistä vaivaa. Jos siis sovelluksen reaktiivisuus on tärkeä ominaisuus, voi koko sovelluskehityksen reaktiivisuudesta olla hyötyä.

8. YHTEENVETO

Reaktiivisuus tarjoaa paljon potentiaalisia hyötyjä sovelluksen käyttökokemuksen kannalta. Käyttäjän kannalta olennaisin reaktiivisuuden ominaisuus on uuden tiedon välitön päivittyminen käyttöliittymään, mikä saattaa tiedon nopeasti käyttäjän nähtäville ja voi tehdä sivun uudelleenlatauksesta tarpeetonta.

Reaktiivisen toiminnallisuuden toteuttaminen ei kuitenkaan ole yksinkertaista. Niinpä reaktiivisen sovelluksen kehityksessä on suositeltavaa käyttää reaktiivisuuden toteuttavaa kirjastoa tai sovelluskehystä, joita onkin saatavilla useita. Meteor-sovelluskehys onnistuu tässä kohtuullisen hyvin, sillä sen reaktiivista mallia on helppo käyttää ja reaktiiviset muutokset ulottuvat tietokannasta aina näkymään asti. Reaktiivisuuden hyödyntäminen vaatii ohjelmoijalta paikoitellen jonkin verran suunnittelua, mutta pääsääntöisesti Meteor huolehtii datan päivittämisestä itsenäisesti. Täysin ongelmaton Meteorin malli ei kuitenkaan ole, vaan Meteorin arkkitehtuuri asettaa sovellukselle omat rajoitteensa. Koska Meteorin reaktiivinen malli riippuu sen eri komponenttien yhteistoiminnasta, esimerkiksi tietokannan vaihto on erittäin hankalaa. Tämä voi olla merkittävä rajoite joidenkin sovellusten kannalta.

Sovelluskehittäjälle reaktiivinen ohjelmointi on hyödyllistä, sillä se muun muassa vähentää datan päivittämiseen liittyvän manuaalisen käsittelyn tarvetta. Ohjelmoijan ei tarvitse dataa päivittävien ohjelman osien toteutuksen yhteydessä miettiä, mitkä muut ohjelman osat riippuvat muokatusta datasta, sillä reaktiivisuuden toteuttava sovelluskehys tai kirjasto heijastaa muutokset automaattisesti muualle ohjelmaan. Reaktiivisuuden hyödyntäminen ei kuitenkaan ole täysin ongelmaton, sillä datan muutosten levittäminen saattaa sen automatisoinnista huolimatta vaatia ohjelmoijalta lisätyötä. Myös sovelluksen suorituskyky saattaa kärsiä.

Vaikka reaktiivisuudesta voi selvästi olla paljon hyötyä, hyödyllisyys riippuu pitkälti sovelluskohteesta. Jos sovelluksessa oleva tieto päivittyy harvoin tai päivitetyn tiedon näyttäminen käyttäjälle välittömästi ei ole tärkeää, ei reaktiivisuudesta ole sovelluksen loppukäyttäjän kannalta paljon hyötyä koska sen vaikutukset ovat vähäiset. Reaktiivinen ohjelmointi voi tästä huolimatta olla hyödyllistä jo pelkästään sovelluskehityksen kannalta, mutta sen hyödyntäminen saattaa muuttaa sovelluksen arkkitehtuuria huomattavastikin. Reaktiivisuuden hyödyntämisen järkevyyttä olisi-kin syytä pyrkiä arvioimaan sovelluksen käyttötarkoitus ja sen käyttäjien tarpeet huomioiden.

LÄHTEET

- [1] Atmosphere [WWW]. [viitattu 5.1.2015].
Saataavissa: <https://atmosphere.meteor.com/>
- [2] Bainomugisha, E., Carreton, A. L., Van Cutsem, T., Mostinckx, S. & De Meuter, W. A Survey On Reactive Programming. ACM Computing Surveys 45(2013)4, Article 52, 34 p.
- [3] Benveniste, A. & Berry, G. The Synchronous Approach to Reactive and Real-time Systems. Proceedings of the IEEE 79(1991)9, pp. 1270-1282.
- [4] Benveniste, A., Caspi, P., Edwards, S.A. & Halbwachs, N. The Synchronous Languages 12 Years Later. Proceedings of the IEEE 91(2003)1, pp. 64-83.
- [5] Berry, G. Real Time Programming: Special Purpose or General Purpose Languages. IFIP World Computer Congress, San Francisco, USA 1989. North-Holland/IFIP. pp. 11-17.
- [6] Bhattacharya, B. & Bhattacharyya, S.S. Parameterized Dataflow Modeling for DSP Systems. IEEE Transactions on Signal Processing 49(2001)10, pp. 2408-2421.
- [7] Blaze – Meteor Reactive Templating Library: Overview [WWW]. [viitattu 4.1.2015]. Saataavissa: <https://atmospherejs.com/meteor/blaze>
- [8] BSON Specification [WWW]. [viitattu 19.2.2014].
Saataavissa: <http://bsonspec.org/>
- [9] Crockford, D. The application/json Media Type for JavaScript Object Notation (JSON) – IETF RFC 4627 [WWW]. 7/2006 [viitattu 7.3.2015].
Saataavissa: <http://www.ietf.org/rfc/rfc4627>
- [10] DB-Engines Ranking [WWW]. [viitattu 19.2.2014].
Saataavissa: <http://db-engines.com/en/ranking>
- [11] DDP Specification [WWW]. [viitattu 10.1.2014]. Saataavissa:
<https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>
- [12] DeBergalis, M. Meteor Blog: Introducing DDP [WWW]. 21.3.2012 [viitattu 10.1.2014]. Saataavissa: <http://www.meteor.com/blog/2012/03/21/introducing-ddp>

- [13] DeBergalis, M. Meteor Blog: Meteor 1.0 [WWW]. 28.10.2014 [viitattu 3.1.2015]. Saatavissa: <https://www.meteor.com/blog/2014/10/28/meteor-1-0>
- [14] DeBergalis, M. MongoDB and Meteor: an Architecture for Realtime Web Apps [WWW]. 10.5.2013 [viitattu 19.2.2014]. Saatavissa: <http://www.mongodb.com/presentations/mongodb-and-meteor-architecture-realtime-web-apps>
- [15] Doll, B. GitHub - The Octoverse in 2012 [WWW]. 19.12.2012 [viitattu 19.2.2014]. Saatavissa: <https://github.com/blog/1359-the-octoverse-in-2012>
- [16] Elliott, C. & Hudak, P. Functional Reactive Animation. International Conference on Functional Programming, Amsterdam, Netherlands 9.–11.6.1997. New York, NY, USA, ACM. pp. 263-273.
- [17] Elliott, C. Push-Pull Functional Reactive Programming. The 14th ACM SIGPLAN International Conference on Functional Programming, Edinburgh, Scotland, 31.8.–2.9.2009. New York, NY, USA, 2009, ACM Press. pp. 25-36.
- [18] Fette, I. & Melnikov, A. The WebSocket Protocol – IETF RFC 6455 [WWW]. 12/2011 [viitattu 18.1.2015]. Saatavissa: <http://tools.ietf.org/html/rfc6455>
- [19] Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures. Dissertation. Irvine, California, Usa 2000. University of California, Irvine. 162 p. Saatavissa: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- [20] Garret, J. J. Ajax: A New Approach to Web Applications [WWW]. 18.2.2005 [viitattu 30.12.2014]. Saatavissa: <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications>
- [21] Gerrit Code Review [WWW]. [viitattu 22.2.2015]. Saatavissa: <https://code.google.com/p/gerrit/>
- [22] Great Place to Work – Suomen parhaat työpaikat [WWW]. [viitattu 18.3.2015]. Saatavissa: <http://www.greatplacetowork.fi/best-companies/suomen-parhaat-tyoepaikat-listat>
- [23] Handlebars.js Homepage [WWW]. [viitattu 10.8.2014]. Saatavissa: <http://handlebarsjs.com/>

- [24] Harel, D. & Pnueli, A. On the Development of Reactive Systems. In: Apt, K. R. (ed.). Logics and Models of Concurrent Systems. NATO ASI Series, Vol. F-13. New York 1985. Springer-Verlag. pp. 477-498.
- [25] Hickson, I. The WebSocket API [WWW]. 20.9.2012 [viitattu 18.1.2015]. Saatavissa: <http://www.w3.org/TR/websockets/>
- [26] Igarashi, Y., Altman, T., Funada, M. & Kamiyama B. Computing: A Historical and Technical Perspective. Boca Raton 2014, Chapman & Hall/CRC Press. 350 p.
- [27] Jouhier, B. Fibers and Threads in Node.js – What For? [WWW]. 11.3.2012 [viitattu 4.1.2015]. Saatavissa: <https://bjouhier.wordpress.com/2012/03/11/fibers-and-threads-in-node-js-what-for/>
- [28] Kniberg, H. Lean From The Trenches: Managing Large-Scale Projects With Kanban. USA 2011, The Pragmatic Bookshelf. 157 p.
- [29] Leaflet Homepage [WWW]. [viitattu 21.2.2015]. Saatavissa: <http://leafletjs.com>
- [30] Lee, E.A. & Messerschmitt, D.G. Synchronous Data Flow. Proceedings of the IEEE 75(1987)9, pp. 1235-1245.
- [31] Leff, A. & Rayfield, J.T. Web-Application Development Using the Model/View/Controller Design Pattern. Fifth IEEE International Enterprise Distributed Object Computing Conference, Seattle, WA, Usa 4.–7.9.2001. Seattle, WA, USA, IEEE. pp. 188-127.
- [32] Macmillan Dictionary - Reactive [WWW]. [viitattu 22.10.2014]. Saatavissa: <http://www.macmillandictionary.com/dictionary/british/reactive>
- [33] Marcotte, E. Responsive Web Design [WWW]. 25.5.2010 [viitattu 30.12.2014]. Saatavissa: <http://alistapart.com/article/responsive-web-design>
- [34] Merriam-Webster Online Dictionary - Reactive [WWW]. [viitattu 22.10.2014]. Saatavissa: <http://www.merriam-webster.com/dictionary/reactive>
- [35] Meteor - GitHub [WWW]. [viitattu 5.1.2015]. Saatavissa: <https://github.com/meteor/meteor>
- [36] Meteor Blaze [WWW]. [viitattu 4.1.2015]. Saatavissa: <https://www.meteor.com/blaze>

- [37] Meteor Documentation [WWW]. [viitattu 5.1.2015].
Saatavissa: <http://docs.meteor.com>
- [38] Meteor Livequery [WWW]. [viitattu 4.1.2015].
Saatavissa: <https://www.meteor.com/livequery>
- [39] Meteor Tracker - Manual [WWW]. 27.10.2014 [viitattu 6.1.2015]. Saatavissa:
<https://github.com/meteor/meteor/wiki/Tracker-Manual>
- [40] MongoDB – The Leading NoSQL Database [WWW]. [viitattu 19.2.2014].
Saatavissa: <http://www.mongodb.com/leading-nosql-database>
- [41] MongoDB Geospatial Indexes and Queries [WWW]. [viitattu 7.2.2014].
Saatavissa: <http://docs.mongodb.org/manual/applications/geospatial-indexes/>
- [42] MongoDB Homepage [WWW]. [viitattu 19.2.2014].
Saatavissa: <http://www.mongodb.org/>
- [43] MongoDB Manual - Cursors [WWW]. [viitattu 23.2.2014].
Saatavissa: <http://docs.mongodb.org/manual/core/cursors>
- [44] MongoDB Manual - Glossary [WWW]. [viitattu 23.2.2014].
Saatavissa: <http://docs.mongodb.org/manual/reference/glossary>
- [45] MongoDB Manual - Replica Set Oplog [WWW]. [viitattu 4.1.2015]. Saatavissa:
<http://docs.mongodb.org/manual/core/replica-set-oplog/>
- [46] Node.js Homepage [WWW]. [viitattu 7.3.2015].
Saatavissa: <http://nodejs.org/>
- [47] Node Fibers - GitHub [WWW]. [viitattu 4.1.2015].
Saatavissa: <https://github.com/laverdet/node-Fibers>
- [48] Nyati, S.S., Pawar, S. & Ingle, R. Performance Evaluation of Unstructured NoSQL Data Over Distributed Framework. 2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Mysore, India, 22.–25.8.2013. pp. 1623-1627.
- [49] OpenStreetMap - About [WWW]. [viitattu 21.2.2015].
Saatavissa: <http://www.openstreetmap.org/about>
- [50] Projects, Applications, and Companies Using Node [WWW]. 25.7.2014 [viitattu 10.8.2014]. Saatavissa: <https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node>

- [51] Schmidt, G. Meteor Blog: Meteor's new \$11.2 million development budget [WWW]. 25.7.2012 [viitattu 19.2.2014]. Saatavissa: <https://www.meteor.com/blog/2012/07/25/meteors-new-112-million-development-budget>
- [52] Spacebars – Handlebars-like template language for Meteor: Overview [WWW]. [viitattu 4.1.2015]. Saatavissa: <https://atmospherejs.com/meteor/spacebars>
- [53] Twitter Bootstrap Homepage [WWW]. [viitattu 7.3.2015]. Saatavissa: <http://getbootstrap.com>
- [54] Vincit Oy [WWW]. [viitattu 18.3.2015]. Saatavissa: <http://www.vincit.fi/>
- [55] W3C Document Object Model [WWW]. [viitattu 7.8.2014]. Saatavissa: <http://www.w3.org/DOM/>
- [56] Wan, Z., Taha, W. & Hudak, P. Real-time FRP. International Conference on Functional Programming, Florence, Italy 3.–5.9.2001. New York, NY, USA, 2001 ACM. pp. 146-156.
- [57] What Is a Mashup? Fichter, D. In: Engard, N. C. (ed.). Library Mashups: Exploring New Ways to Deliver Library Data. Medford, New Jersey, USA 2009, Information Today, Inc. pp. 3-17.
- [58] Whiting, P.G. & Pascoe, R.S.V. A History of Data-Flow Languages. IEEE Annals of the History of Computing 16(1994)4, pp. 38-59.
- [59] Wirth, C., Prähofer, H. & Schatz, R. A Multi-level Approach for Visualization and Exploration of Reactive Program Behavior. 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis, Williamsburg, Virginia, USA 29.–30.9.2011. 2011, IEEE. 4p.