TAMPEREEN
AMMATTIKORKEAKOULU

Meteor verkkosovelluskehityksessä

Sami Joutsijoki

Opinnäytetyö
Joulukuu 2015
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

JOUTSIJOKI, SAMI:
Meteor verkkosovelluskehityksessä

Opinnäytetyö 56 sivua, joista liitteitä 4 sivua
Joulukuu 2015

Opinnäytetyön tavoitteena oli tutkia Meteorin soveltuvuutta verkkosovelluksien kehitykseen. Toimeksiantaja oli Tampereen ammattikorkeakoululla toimiva kehitysyhteisö Floworks Living Lab, jonka jatkuvana tavoitteena on etsiä sovelluskehitysympäristöjä tulevia ohjelmistoprojekteja varten. Kehitysyhteisöltä projektin tilannut Tampereen ammattikorkeakoulun sairaanhoitajien koulutusohjelma halusi digitalisoida sairaanhoitajien ohjatun harjoittelun arviointiprosessin. Tämä oli oiva tilaisuus testata Meteorin soveltuvuutta tällaisen projektin toteutukseen ja raportoida kokemukset ja tulokset Floworksille opinnäytetyön muodossa. Opinnäytetyön tiedonkeruumenetelmänä toimivat pääosin omat kokemukset Meteorista suhteessa aikaisempiin verkkosovelluskehityskokemuksiin sekä internetistä löytyvät alan ammattilaisten blogikirjoitukset.

Opinnäytetyön aikana sovellus nimeltä Digiharkka saatiin valmiiksi. Sovellus täytti kaikki asiakkaan vaatimukset, ja se saatiin noin 30 henkilön pilottikäyttöön syksyllä 2015. Vuoden 2016 puolella sovellus on tarkoitus ottaa käyttöön kaikkiin tuleviin sairaanhoitajien ohjattuihin harjoitteluihin Tampereen ammattikorkeakoulussa.

Sovelluksen teko oli erityisen vaivatonta Meteorilla verrattuna aikaisempiin kokemuksiini vastaavista työkaluista. Meteorin suunnittelumalli ja sisäänrakennetut ominaisuudet mahdollistivat erittäin vahvan ja selkeän koodin luomisen hyvin pienellä rivimäärällä. Meteorin erinomaisen ekosysteemin ansiosta moniin sovelluskohtaisiin ongelmiin löytyi kolmannen osapuolen paketti, jolla ongelma saatiin ratkaistua muutamalla rivillä koodia.

Jatkon osalta Meteoria voisi kokeilla vielä monipuolisempien ongelmien ratkomisessa, jotta sen kaikkia ominaisuuksia testattaisiin toden teolla. Digiharkka-sovelluksessa joitain komponentteja voisi refaktoroida vielä enemmän Meteormaisiksi, sillä opinnäytetyön loppuvaiheilla huomasin, miten sovellukseen tekemiäni ensimmäisiä ominaisuuksia olisi voinut toteuttaa vieläkin paremmin.

Asiasanat: meteor, digitalisointi, digiharkka, moderni verkkosovelluskehitys

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

JOUTSIJOKI, SAMI:
Meteor in Web Application Development

Bachelor's thesis 56 pages, appendices 4 pages
December 2015

The purpose of this thesis was to study the viability of Meteor in web application development. The thesis was commissioned by Floworks Living Lab which operates at Tampere University of Applied Sciences (TAMK). Their ongoing goal is to find the best development environments possible for future projects.

The Degree Programme in Health Care at TAMK commissioned a project from Floworks, where they wanted to digitalize the evaluation process of their mandatory internship periods. As a part of this study, an application called Digiharkka was programmed. It met all the client's expectations and it was taken into use in a 30-person pilot during the autumn of 2015. In the beginning of the year 2016 the application is expected to be going into full scale production.

Developing the application with Meteor was very seamless compared to the author's previous experiences in web development. Meteor's design patterns and built-in features make it possible to write powerful and clear code in a few lines.

It is recommended here that Floworks should continue working with Meteor in their future projects. More thorough testing of all the features of Meteor in different problem domains would be advisable. In terms of Digiharkka, some of the application's components could be refactored to be even more Meteor-like, as the author discovered ways to improve the older features of Digiharkka even further.

Key words: meteor, digitalization, digiharkka, modern web development

SISÄLLYS

1 JOHDANTO

Floworks Living Lab on Tampereen ammattikorkeakoululla toimiva kehitysyhteisö, jonka yhtenä tavoitteena ovat oppimisprojektit, joissa opiskelijat pääsevät kehittämään TAMKin eri yksikköjen toimintaa monenlaisten projektien parissa. Floworksin sovelluskehitysprojektit ovat pääosin hyvin käyttöliittymäpainotteisia, joten sillä on jatkuva tarve etsiä teknologioita, jotka mahdollistavat rikkaiden käyttäjäkokemuksien luomisen mahdollisimman vaivattomasti.

Floworks oli saanut alkuvuodesta 2015 tilauksen sairaanhoitoalan koulutusohjelmalta, jossa haluttiin digitalisoida sairaanhoitajien ohjattuun harjoitteluun liittyvä arviointiprosessi. Projekti alkoi kahden hengen kehittäjätiimin voimin maaliskuun 2015 aikoihin verkkokehitysopintojakson myötä. Itse liityin projektiin kesällä 2015 toisen opiskelijan kanssa ja jatkoimme kehitystä syyskuuhun asti. Tästä eteenpäin jatkoin yksin projektissa opinnäytetyön merkeissä ja jo syksyllä 2015 sovellus oli pilottikäytössä oikeissa harjoitteluissa. Vuoden 2016 puolella sovelluksen on tarkoitus tulla käyttöön kaikille harjoitteluun osallistuville.

Ongelma

Sairaanhoitoalan koulutusohjelman ohjatun harjoittelun aikana opiskelijoiden pitää määritellä itselleen oppimistavoitteita, joita he harjoittelun aikana pyrkivät toteuttamaan. Nämä tavoitteet auttavat harjoittelupaikalla olevaa harjoittelun ohjaajaa paremmin arvioimaan väliarviossa ja loppuarviossa opiskelijaa. Tavoitteet kuuluvat joihinkin yhdeksästä osaamisalueesta, jotka ovat määritelty seuraaviksi:

1. asiakaslähtöisyys
2. hoitotyön eettisyys ja ammatillisuus
3. johtaminen ja yrittäjyys
4. sosiaali- ja terveydenhuollon toimintaympäristö
5. kliininen hoitotyö
6. näyttöön perustuva toiminta ja päätöksenteko

7. ohjaus- ja opetusosaaminen
8. terveyden ja toimintakyvyn edistäminen
9. sosiaali- ja terveyspalvelujen laatu ja turvallisuus

Jokaisen osaamisalueen alle opiskelijan pitää siis määritellä henkilökohtaisia tavoitteita, joita voi olla useita. Opiskelijalle näiden keksiminen ja ymmärtäminen voi olla aikamoisen työn takana ja tällä hetkellä käytössä oleva ohjatun harjoittelun lomake koetaan epäselvänä ja vaikeana käyttää. Yleensä käykin niin, että opettaja joutuu opastamaan tavoitteiden määrittelyssä sähköpostien välityksellä, mikä vie aikaa ja resursseja turhaan. Nykyisellä lomakkeella opiskelija ei myöskään saa jatkuvaa kirjallista arviointia harjoittelun aikana, eikä ohjaajan väliarviointi ole helposti opettajan luettavissa. Ongelman ydin on siis se, että harjoitteluprosessissa tieto ei kulje tarpeeksi hyvin osapuolien välillä ja että paperityötä kertyy liian paljon.

Ratkaisu

Ratkaisu edellä kuvattuun ongelmaan oli siis verkkosovellus, joka täyttää minimissään seuraavat kriteerit:

- Opiskelija pystyy tekemään kaikki harjoitteluun liittyvät asiat (tavoitteiden luonti, väliarviointi, loppuarviointi).
- Ohjaaja näkee ohjattavan opiskelijansa itsearvioinnit ja voi itse antaa omat arviointinsa.
- Opettaja näkee oman ryhmänsä opiskelijat ja heidän harjoittelunsa tilan sekä pystyy hyväksymään/hylkäämään opiskelijan tavoitteet.
- Kaikki pystyvät kommunikoimaan keskenään reaaliajassa ja näkevät toistensa kirjoitukset.
- Opiskelijalle on mahdollista antaa harjoittelun aikana jatkuvaa palautetta harjoittelun etenemisestä.

Sovelluksen pitää myös olla modernin verkkokehityksen vaatimuksien tasolla, sillä siihen haluttiin myös mahdollisuus tehdä tavoitteiden jatkuvaa arviointia mobiililaitteilla.

Moderni verkkokehitys

Tämän päivän verkkokehityksessä yksi tärkeimmistä asioista on ottaa huomioon erilaisten verkkoon pääsevien laitteiden valtava määrä. Vuonna 2008 internetiin yhdistyneiden laitteiden määrä ohitti niitä käyttävien ihmisten määrän (Gasston 2013). Laitteita, joilla nykyään pääsee internetiin, ovat muun muassa pöytäkoneet, kannettavat tietokoneet, tabletit, älypuhelimet, pelikonsolit, TV:t jne. Käytännössä tämä tarkoittaa sitä, että käyttöliittymien tulisi venyä/kutistua näytön resoluution mukaan ja mobiililaitteilla sivua ladattaessa tulisi ottaa huomioon siirrettävän datan määrä (mitä huonompi internet–yhteys, sitä vähemmän dataa pitäisi kerralla lähettää).

Nykyään käyttäjien odotukset verkkosovelluksien toimivuudesta ja laadusta ovat paljon korkeammat kuin jokunen vuosi sitten. Esimerkiksi Google, Twitter ja Facebook ovat päässeet niin vaikuttaviin tuloksiin sovelluksiensa kanssa, että ne tuntuvat enemmänkin työpöytäsovelluksilta kuin verkkosovelluksilta (Hochhaus & Schoebel 2015). Reaktiivisuus, skaalautuvuus ja responsiivisuus ovat nykyään enemmän normeja, kuin poikkeuksia.

Sovelluskehittäjän valitessa työkaluja ja teknologioita kannattaa tutkia, kuinka hyvin eri vaihtoehdot ovat ottaneet huomioon näitä nykyajan vaatimuksia.

Miksi Meteor?

Meteor on melko uusi, vuonna 2012 julkaistu avoimen lähdekoodin isomorfinen JavaScript–verkkosovelluskehys, jolla on paljon tuulta allaan. Meteor Development Group (MDG) on saanut 11.2 miljoonaa dollaria rahoitusta ja Meteor on kirjoitushetkellä GitHubin kymmenenneksi eniten tähtiä saanut projekti. Seuraavaksi suosituimmalla full stack JavaScript-kehyksellä Derby.js:llä on kymmenesosa Meteorin näkyvyydestä StackOverflow:ssa ja Quorassa (Meteorpedia Why Meteor 2015).

Meteor on myös erittäin helppo oppia verrattuna muihin full stack –vaihtoehtoihin, kuten LAMP (Linux Apache MySQL PHP) tai MEAN (MongoDB Express Angular Node) -stackiin. Jos JavaScript on tuttua, ei Meteorissa lisäosaamista vaadita kovinkaan paljoa suhteessa siihen lisäarvoon, mitä kehys antaa. Meteorista luvataankin, että yksinkertaisen

reaaliaikaisen chat–sovelluksen pystyy rakentamaan noin tunnissa. (Meteorpedia Why Meteor 2015.)

Erityisesti Meteorin sisäänrakennettu reaktiivisuus ja yhden latauksen sivu (Single Page Application, SPA)-suunnittelumalli tuntuivat sopivan hyvin tehtävänannon vaatimuksien toteuttamiseen. Reaktiivisuus mahdollisti reaaliaikaisen chatin toteuttamisen varsin vaivattomasti ja yhden latauksen malli mahdollistaa kevyen datan siirron ensimmäisen latauksen jälkeisillä käyttökerroilla, mikä oli tärkeää mobiililaitteilla. Sovelluksen julkaiseminen Meteorin palvelimille yhdellä komennolla oli myös todella tärkeää kehityksen kannalta.

Edellä mainittujen syiden takia päädyimme kokeilemaan Meteoria. Varsinkin lupaus helppoudesta oppia ja tuottavuuden kasvusta olivat sellaisia tekijöitä, jotka painoivat paljon vaakakupissa teknologiapäätöksiä tehtäessä.

Opinnäytetyön rakenne

Käyn opinnäytetyössä läpi Meteorin rakennetta, Digiharkka-sovellusta ja lopuksi pohdin Meteorin soveltuvuutta verkkosovelluskehitykseen. Meteorin rakenne –luvussa käyn läpi niitä osia, joista Meteor koostuu ja niitä ominaisuuksia, joita Meteor tarjoaa. Demonstroin koodinpätkillä joitakin Meteorin tärkeimmistä ominaisuuksista. Digiharkka-sovellus -luvussa kerron sovelluksen teosta, sekä sen ominaisuuksista ja esittelen kuvia sen käyttöliittymästä. Pohdinta-luvussa pohdin Meteorin soveltuvuutta verkkosovelluskehitykseen tietoturvallisuuden, skaalautuvuuden, sekä tuottavuuden näkökulmista, ja pohdinnan yhteenvedossa kerron sovelluksen tulevaisuudesta. Opinnäytetyössä käytettävät termit ja lyhenteet ovat selitettynä liitteinä työn lopussa.

2 METEORIN RAKENNE

Meteor on lopulta kokoelma avoimen lähdekoodin projekteja, jotka on sidottu yhteen oletusasetuksilla siten, että sen kanssa alkuun pääseminen on muihin vastaaviin työkaluihin verrattuna erityisen nopeaa. Tässä luvussa pyrin selventämään, mistä kaikista osasista ja kirjastoista Meteor koostuu. Meteorin tarjotessa työkaluja aina käyttäjän ruudulta tietokannan manipulointiin asti on syytä jakaa Meteor sekä asiakas-, että palvelinpuolen osiin.

2.1 Asiakaspuoli

Meteor, kuten SPA–sovellukset yleensä, lataa kaiken asiakaspuolen koodin ja resurssit (HTML:n, CSS:n, JavaScriptin, kuvat jne.) sivun ensimmäisellä latauksella HTTP–protokollalla. Tämän jälkeen vain raaka data liikkuu asiakkaan ja palvelimen välillä. Tässä alaluvussa käyn läpi niitä komponentteja, joita asiakkaan koneella suoritetaan.

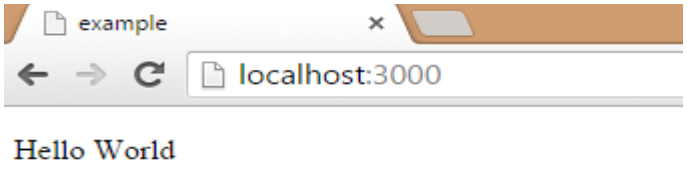
2.1.1 HTML

HTML (**H**yper**T**ext **M**arkup **L**anguage) on World Wide Webin keksijäksi yleisesti ajatellun Tim Berners-Leen kehittämä alustariippumaton dokumenttien kuvauskieli. Hän kehitti HTML:n ja HTTP–protokollan, koska hänen tarkastellessaan satojen muiden tutkijoiden löydöksiä CERNissä ollessaan, ongelmaksi muodostuivat monet eri ohjelmointikielet, dokumenttien formaatit ja yhteensopimattomat rajapinnat. Nämä hidastivat löydöksien jakamista muiden tutkijoiden kesken. (Computermagazine 2015.)

Meteor käyttää HTML:ää standardina templaattikielenään, mutta myös Jade–niminen kieli on tuettu. Ainoa Meteorin lisäämä HTML-tagi on ”template”, eikä Meteor näinollen vaadi yhtään uuden oppimista HTML:n puolelta (Meteorpedia Why Meteor 2015). Kuvissa 1 ja 2 on klassinen ”Hello World”–esimerkki Meteorilla.

```
<head>
  <title>example</title>
</head>

<body>
  {{> hello}}
</body>

<template name="hello">
  <p> Hello World </p>
</template>

```

KUVA 1. Hello World Meteorissa (koodi)



KUVA 2. Hello World Meteorissa (näkymä)

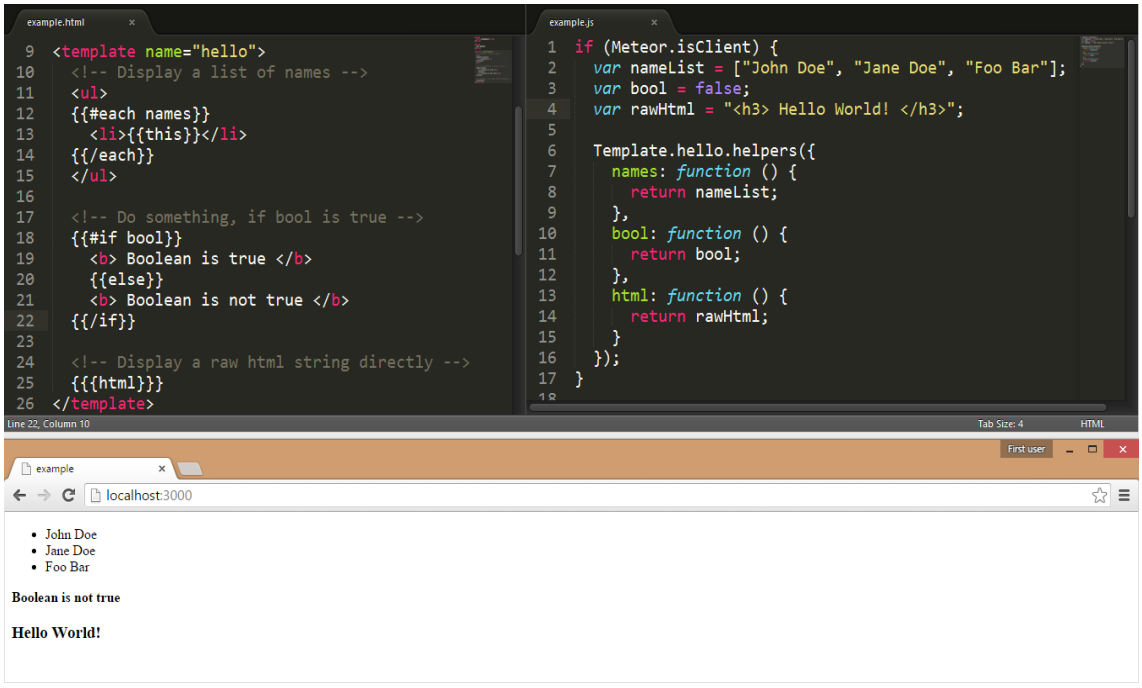
2.1.2 Spacebars

Spacebars on Meteorin oma Handlebarsin pohjalta kehitetty HTML:n seassa käytettävä templaattikieli, jolla sidotaan näkymiä ja sovelluslogiikkaa deklaratiivisesti toisiinsa niin, että kun pohjalla oleva data muuttuu, näkymää päivitetään automaattisesti vastaamaan uutta dataa (Meteor docs 2015).

Kuvassa 3 tehdään yhdensuuntainen datan sitominen näkymän ja datan välillä. Kun nappia painetaan, kasvatetaan laskuria yhdellä ja näytetään luku näkymässä. {{counter}} on Spacebars–syntaksia ja siinä sidotaan sen nimisen ”helperin” palauttama data kyseiseen kohtaan näkymässä. Helpereihin ja eventteihin palataan tarkemmin seuraavassa luvussa, mutta käytän niitä jo nyt Spacebarsin käytön demonstroimiseksi.

KUVA 3. Spacebars-syntaksin demonstrointia

Spacebarsilla voi myös suorittaa yksinkertaista logiikkaa näkymän seassa, mutta tämä ei ole kovin suositeltavaa, sillä logiikka ja näkymä olisi hyvä pitää mahdollisimman paljon erosssa toisistaan koodin ylläpidettävyyden kannalta. Kuvassa 4 on kuitenkin muutamia edistyneempiä asioita, joita Spacebarsin avulla voi tehdä.



KUVA 4. Spacebarsin logiikan demonstrointia

Kuvassa 4 ensimmäisenä iteroidaan taulukko (array), jossa on 3 jäsentä ja näytetään jokainen jäsen järjestämättömän listan alkiona (list item). Seuraavaksi demonstroidaan ehdollista lauseketta, jos ”bool”–niminen helper palauttaa arvon true, tehdään jotain. Muuten tehdään jotain muuta. Kolmannessa kohdassa näytetään, kuinka Spacebarsilla voidaan esittää raakaa HTML–dataa suoraan merkkijonosta kolminkertaisilla aaltosuluilla.

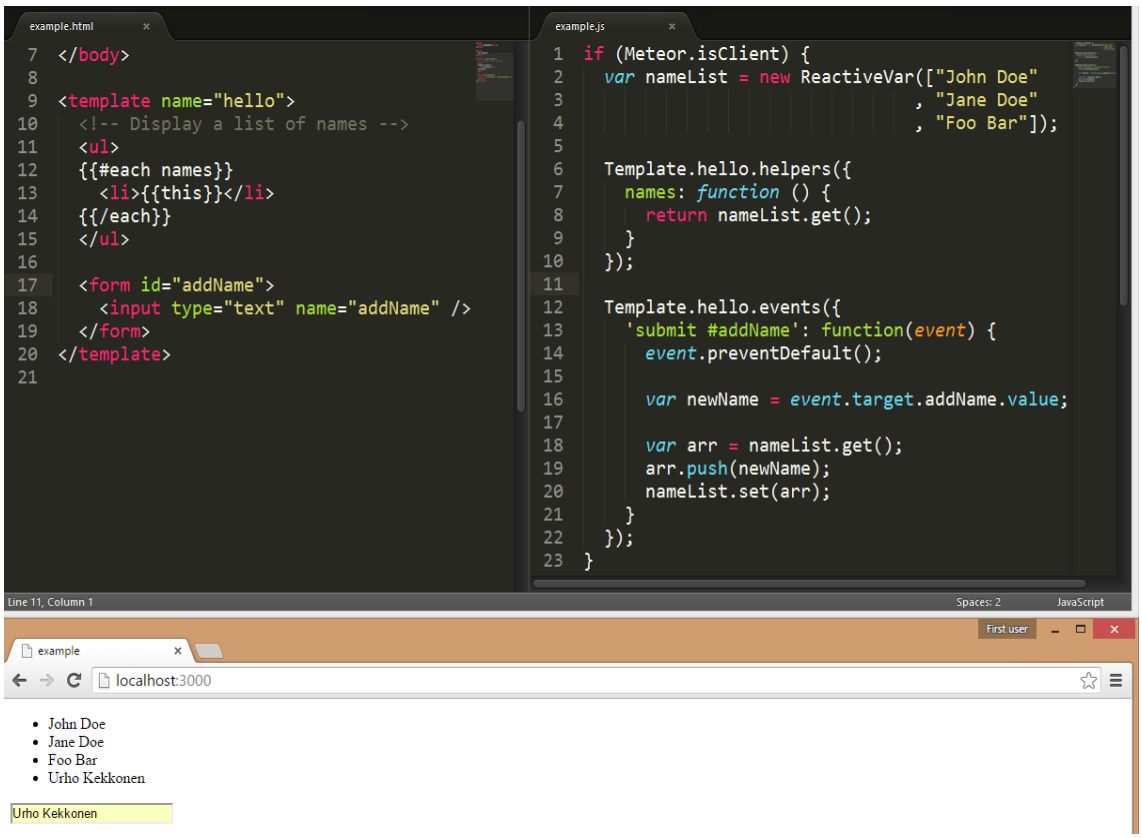
Spacebarsilla voi myös tehdä paljon muutakin erikoista, kuten lähettää parametrejä helppereille, asettaa datakonteksteja ja uudelleennimetä lausekkeita (expressions) (Atmosphere Spacebars 2015).

2.1.3 JavaScript

JavaScript on Meteorin sydän. Meteor käyttää JavaScriptiä sekä asiakaspuolella että palvelimen puolella ja MongoDB–tietokannan rajapinnassa. Myös DDP–protokolla, johon palaan myöhemmin, on pohjimmiltaan JSONia (**J**ava**S**cript **O**bject **N**otation). CoffeeScript on myös tuettuna Meteorissa, mutta sitä käytettäessä kehys ei olisi enää isomorfinen (sama kieli asiakas- ja palvelinpuolella).

Meteor ei lisää kieleen kovin paljoa Meteor–spesifistä syntaksia. Asiakkaan puolella Meteor lisää käsitteet ”helpers” ja ”events”. Helper on avain-arvo-pari, jossa avain on helperin nimi ja arvo on funktio, joka voi palauttaa joko staattista tai dynaamista dataa. Dynaaminen data voi olla esimerkiksi tietokantakyselyn tulos. Tämä data on automaattisesti reaktiivista, eli aina kun palautettava data muuttuu, Meteor päivittää tämän datan myös näkymässä. Eventit taas ovat Meteorin tapa käsitellä käyttäjän syötettä sitomalla tapahtumankäsittelijöitä tiettyihin templaatin elementteihin käyttämällä CSS (**C**ascading **S**tyle **S**heets) –valitsijoita (selector).

Templaatille lähetetään siis sekä helpereitä että eventtejä helpers- ja events-nimisiin funktioihin JSON–muodossa, jossa helperien kohdalla avain on helperin nimi ja eventtien kohdalla avain on CSS–valitsija. Helpereiden arvona on palautettava data ja eventtien arvona tapahtumankäsittelijä (event handler). Kuvassa 5 demonstroidaan helperien ja eventtien käyttöä.



KUVA 5. Helperien ja eventtien käyttöä

Kuvan 5 koodissa sidotaan ”names”–niminen helper järjestämättömän listan datan lähteeksi. Tämä helper palauttaa nimilistan, joka on luotu reaktiivisena muuttujana (muuttujat eivät ole automaattisesti reaktiivisia, tätä varten pitää lisätä reactive-var –niminen paketti). Tämän jälkeen alempana HTML:ssä on luotu lomake. JavaScriptin puolella tähän on sidottu tapahtuma CSS–valitsijan ja tapahtumankäsittelijän avulla. Kun lomake lähetetään, laitetaan uusi parametrinä saatu arvo nimilistan jatkoksi. Tämän jälkeen uusi nimi näkyy samantien näkymässä ilman tarvetta ladata sivua uudelleen.

Kirjasto, joka hoitaa kaiken tämän käyttöliittymän päivittämisen käyttäjältä piilossa, on nimeltään Blaze.

2.1.4 Blaze

Blaze on MDG:n oma front end –kirjasto, joka on verrattavissa Angulariin, Backboneen, Emberiin, Reactiin, Polymeriin tai Knockoutiin, mutta on paljon helppokäyttöisempi. Blazea voi ajatella reaktiivisena jQuerynä, joka automaattisesti pitää DOMin (**D**ocument **O**bject **M**odel) synkronissa määriteltyjen datan sidosten kanssa. (Meteor Blaze 2015.)

Meteorissa ohjelmoijan itse ei yleensä tarvitse Blazea suoraan käyttää. Ohjelmoija vain kirjoittaa templaattinsa normaaliin tapaan ja Blaze toimii taustalla. Templaattikielellä ei myöskään ole väliä: se voi olla joko Spacebarsia tai Jadea tai ohjelmoija voi kirjoittaa omansa.

Blazen ansiosta ohjelmoijan ei tarvitse kirjoittaa valtavasti turhaa itseään toistavaa (boilerplate) koodia, mikä tekee koodista selkeämpää ja ylläpidettävämpää. (Meteor Blaze 2015.)

2.1.5 Tracker

Tracker on 1 kilobitin kokoinen JavaScript–kirjasto, joka mahdollistaa läpinäkyvän reaktiivisen ohjelmoinnin (transparent reactive programming). Tällä tarkoitetaan sitä, että ohjelmoijan ei tarvitse tietää miten reaktiivisuus on Meteorissa toteutettu ja hän silti saa siitä kaiken hyödyn. (Meteor Tracker 2015.)

Tracker on vastaus seuraavaan ongelmaan: sovelluksessa monitoroidaan jotain arvoa, esimerkiksi tietokantakyselyn tulosta tai senhetkistä aikaa. Kun kyseinen arvo muuttuu halutaan tehdä jotain, esimerkiksi päivittää käyttöliittymää. Tämän ongelman ratkaisemiseksi on olemassa muutamia yleisesti käytettyjä suunnittelumalleja:

- Kysely ja vertailu -malli (Poll and diff) -> Tietyin väliajoin (esimerkiksi 1 sekunnin välein) haetaan uusi arvo ja verrataan sitä vanhaan. Jos arvo on muuttunut, päivitetään datan käyttäjää (käyttöliittymää).
- Tapahtuma/kuuntelijamalli (Event/listener) -> Datan lähde ”ilmoittaa” tapahtuman datan muuttuessa. Jossain muualla sovelluksessa määritelty kuuntelija hoitaa tapahtuman käsittelyn (esimerkiksi päivittämällä käyttöliittymää vastaamaan uutta dataa).
- Sitominen (Bindings) -> Arvot ovat esitetty objekteina, jotka toteuttavat jonkin rajapinnan ja jossain muualla sovelluksessa nämä kaksi sidotaan toisiinsa siten, että kun toinen muuttuu niin toista muutetaan samantien.

Näissä kaikissa tyyleissä on kuitenkin omat ongelmansa. Kysely ja vertailumallissa käyttöliittymä ei ole reaaliaikainen, sillä käyttäjän muuttaessa dataa käyttöliittymä

muuttuu vasta seuraavan kyselykierroksen jälkeen. Tapahtuma/kuuntelijamallissa ohjelmoijan pitää kirjoittaa melko monimutkaista kontrollerikoodia, jossa manuaalisesti kerrotaan miten tapahtuma pitää käsitellä. Tämä taas johtaa lopulta vaikeasti ylläpidettävään ja rumaan koodiin. Sitomisessa taas joudutaan käyttämään liian spesifistä kieltä monimutkaisten arvojen suhteiden esittämiseen (Tracker Manual 2015.)

Trackerin idea on hyvin yksinkertainen ja selkeä. Imperatiivisen (ohjelmoija kertoo tarkalleen, mitä sovelluksen tulisi tehdä) tyylin sijasta se mahdollistaa deklaratiivisen (ohjelmoija ilmaisee ongelman juuri niin kuin sitä ajattelee) tyylin ohjelmistoissa. Kuvassa 6 demonstroidaan, miten Trackerin avulla mistä tahansa arvosta voidaan luoda reaktiivinen.

KUVA 6. Trackerin demonstraatiota

Kuvassa 6 muuttuja ”name” on täysin reaktiivinen arvo. Kun kyseisen arvon aksessorimetodia (getName) kutsutaan, kutsutaan myös Dependency–instanssin depend–metodia. Arvon mutaattorimetodia (setName) kutsuttaessa kutsutaan Dependency–instanssin changed–metodia. Näin luodaan riippuvuussuhde, jota voidaan käyttää Tracker.autorun –metodissa. Autorunin sisälle voisi esimerkiksi laittaa jQuery–koodia, joka sitten päivittäisi DOMia datan muuttuessa. Juuri näin tekee Blaze, joten ohjelmoijan tehtäväksi jää vain templaattien kirjoittaminen.

Mistä tahansa kirjastosta voidaan tehdä Trackerin huomioiva (Tracker aware), kuten aikaisemmin mainittu Blaze on. Näiden kirjastojen avulla voidaan tehdä monimutkaisia tapahtumapohjaisia ohjelmia ilman turhaa itseään toistavaa koodia. (Meteor Tracker 2015.)

2.1.6 MiniMongo

MiniMongo on lähes koko MongoDB–tietokannan rajapinnan uudelleentoteutus, joka toimii selaimen muistissa. Sitä voi ajatella MongoDB-emulaattorina, jota vasten voi suoraan käyttää samoja kyselyitä kuin palvelimella (Meteor mini-databases 2015.)

MiniMongo kehitettiin, koska yleinen ongelma dataa tallentavissa verkkosovelluksissa oli datan käsittelyn skaalautumattomuus. Sovelluksien pitää tallentaa palvelimelta haettu data jotenkin väliaikaisesti muistiin, jotta datan pohjalta voidaan renderöidä käyttöliittymä. Yleinen tapa luoda malliluokkia (model classes) on hyvä idea pienessä mittakaavassa, mutta hyvin äkkiä nousee tarve filtteröidä, järjestää ja muutenkin käsitellä dataa. Lopuksi päädytään siihen, että ohjelmoija on kehittänyt omanlaisensa tietokantarajapinnan datan käsittelyyn. Yleensä tällainen rajapinta joko ei ole tarpeeksi hyvä tai sen teossa kestää liian kauan aikaa. Kaikki tämä aika on pois itse sovelluksen ominaisuuksista (Meteor mini-databases 2015.)

MiniMongo mahdollistaa yhteensopivuuden asiakaspuolen ja palvelinpuolen tietokantaa käsittelevän koodin kanssa, jolloin tällainen koodi tarvitsee kirjoittaa vain kerran. MiniMongo mahdollistaa Meteorissa myös ominaisuuden nimeltä viiveen kompensaatio (latency compensation).

2.1.7 Viiveen kompensaatio

Viiveen kompensaatio tarkoittaa uuden datan ja päivittyneen näkymän simulointia asiakkaan koneella heti asiakkaan syöttäessä uutta dataa tietokantaan. Asiakkaan kone siis optimistisesti olettaa, että uusi data pääsee palvelimen tarkistuksista läpi ja näyttää tämän datan samantien käyttöliittymässä. Jos tieto ei tallennukaan tietokantaan (esimerkiksi käyttäjällä ei ollut riittäviä oikeuksia), päivitetään käyttöliittymä vastaamaan

oikeaa dataa datan saapuessa palvelimelta. Suuressa osassa käyttötapauksia tämä ominaisuus antaa illuusion siitä, että tieto liikkuu salamannopeasti asiakkaan koneen ja palvelimen välillä.

Jotta viiveen kompensaatio olisi mahdollista, tulee asiakkaan koneella olla näkyvissä se koodi, jossa tieto tallennetaan tietokantaan. MiniMongoon kohdistuvat tietokantaoperaatiot ovat automaattisesti viive-kompensoituja, mutta jos tieto lähtee parametreinä palvelimen metodeille (näistä myöhemmin lisää), joissa tieto sitten tallennetaan suoraan MongoDB-tietokantaan, tulee itse tietokantakyselyn olla näkyvissä asiakkaan koneella ollakseen viive-kompensoitu. Tällaiset metodit tulisi laittaa tiedostoon, joka suoritetaan sekä asiakkaan koneella että palvelimen puolella.

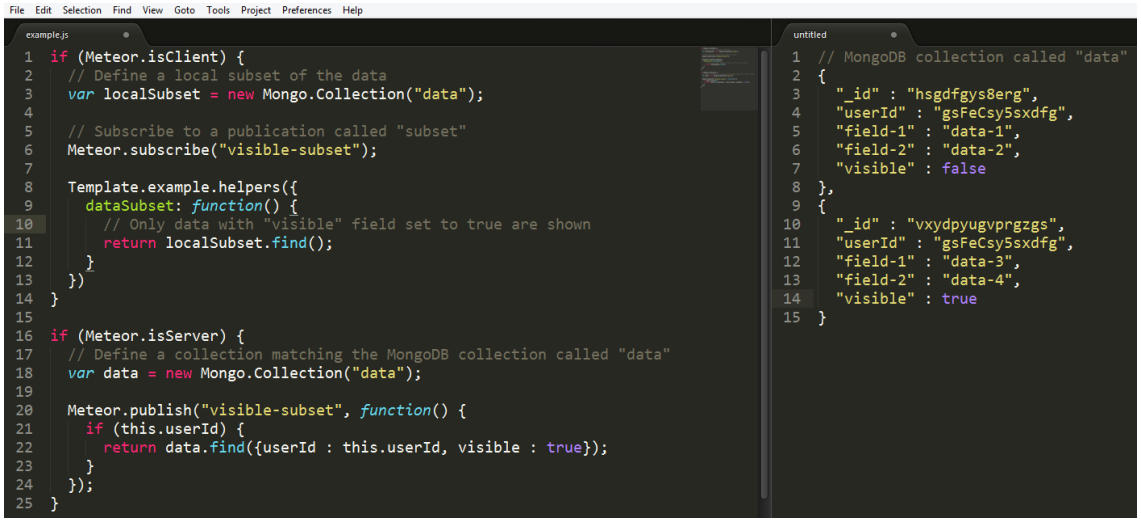
2.2 Palvelimen puoli

Meteorin palvelinpuoli on rakennettu NodeJS:n päälle. Tämä mahdollistaa Meteorissa myös natiivien NodeJS-työkalujen, kuten npm:n (paketinhallintatyökalu) käytön. Tässä alaluvussa käyn läpi Meteorin palvelimen puolen komponentteja.

2.2.1 Julkaisija/tilaaja -malli

Julkaisija/tilaaja –malli (Publish/subscribe) on Meteorin tapa käsitellä datan siirto asiakaskoneen ja palvelimen välillä. Tässä suunnittelumallissa datan lähde (palvelin) lähettää tiedon dataan liittyvistä muokkauksista datan tilanneille asiakkaille. Asiakkaan ei siis tarvitse sokeasti kysellä datan lähteeltä onko muutoksia tapahtunut.

Asiakkaan ei kuitenkaan ole pakko tilata kaikkea dataa tietystä kokoelmasta. Asiakkaan puolella tilataan *osakokoelma* suuremmasta kokoelmasta (collection) rajattuna parametreilla. Kuvassa 7 näytetään, miten tämä käytännössä toimii.



KUVA 7. Julkaisija/tilaaja –malli

Kuvan 7 tapauksessa asiakas tilaa julkaisun ”visible-subset”, jonka sisällä sanotaan data –nimiselle kokoelmalle ”etsi kaikki dokumentit, joissa kentän userId arvo on kutsujan userId ja kentän visible arvo on true”. Tämä osakokoelma lähetetään asiakkaalle, joka vastaa asiakkaan ”localSubset”–osakokoelmaa. Tämän jälkeen asiakkaan kutsuessa ”find”–metodia osakokoelmalle palautuvat vain ne dokumentit, jotka asiakas oli tilannut.

2.2.2 Metodit

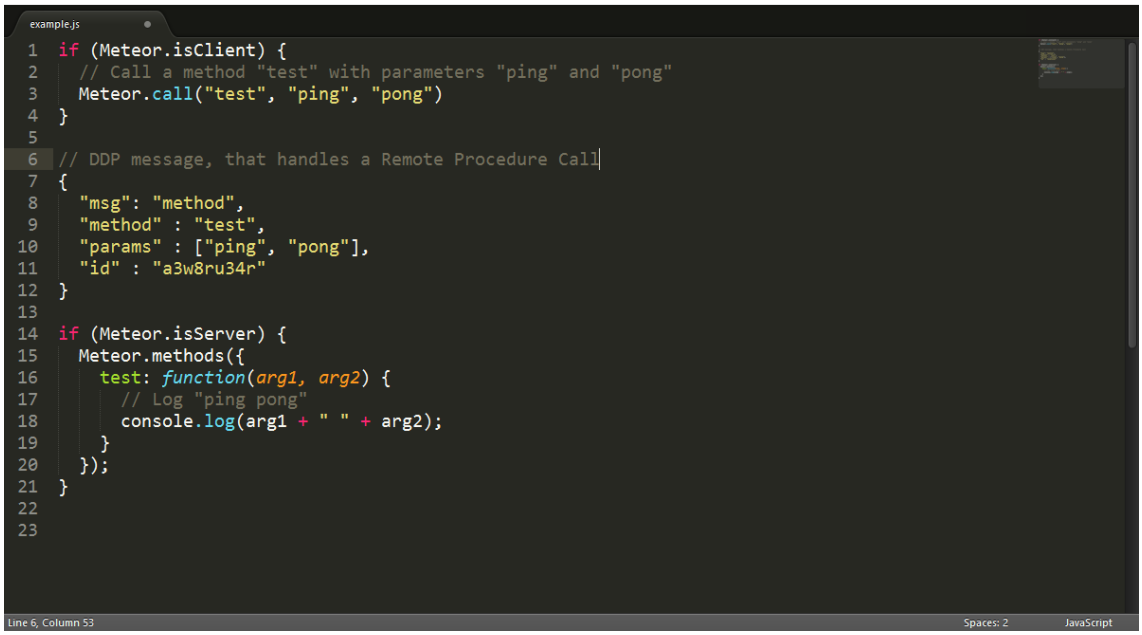
Vaikka käyttäjä pystyy antamaan syötettä suoraan tietokantaoperaatioina MiniMongon avulla, ei se aina ole paras tyyli toteuttaa syötteen ottaminen. Tietokantakyselyitä voi nimittäin tehdä suoraan selaimen JavaScript–konsolilla, joten tiedon varmentamisessa voi tulla suuriakin ongelmia. Tällaisia ongelmia varten kehitettiin vain palvelimen puolella olevat metodit, joita asiakas voi kutsua normaaliin tapaan parametrein ja joissa ohjelmoija voi määritellä tarkalleen, mitä tekee kyseisellä datalla.

2.2.3 DDP

DDP (**D**istributed **D**ata **P**rotocol) on MDG:n kehittämä protokolla, jota voi ajatella websocketien REST–protokollana. DDP on pohjimmiltaan JSON-dataa, joka määrittelee standardit nimet viesteille asiakkaan ja palvelimen välillä. Jotkut kentät on määritelty olevan EJSON–muodossa, jolla voidaan lähettää binääridataa verkon yli. DDP:n rakenne

on yksinkertainen: jokaisella viestillä on ”msg”–kenttä, joka määrittelee viestin tyypin. Muut kentät sitten riippuvat viestin tyypistä.

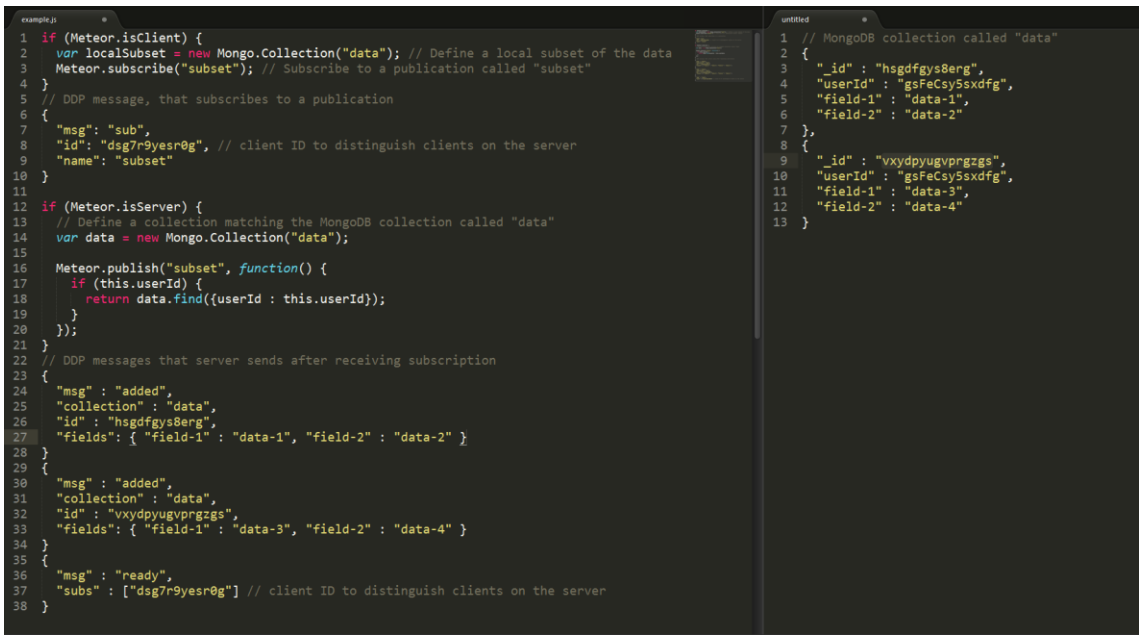
DDP:llä on pääasiallisesti kaksi tehtävää: kutsua palvelimen funktioita (Remote Procedure Call) ja hoitaa datan käsittely (Meteorhacks DDP 2015). Jos selain tukee WebSocket–teknologiaa viestien kuljettamiseen, käyttää DDP sitä. Muutoin se käyttää SockJS–kirjastoa, joka emuloi WebSocketia. Kuvassa 8 on esimerkki DDP–viestistä, joka lähtee palvelimelle asiakkaan kutsuessa palvelimen metodia.



KUVA 8. Esimerkki DDP-viestistä

Jos käyttäjä kutsuu tietokannan dataa muokkaavaa metodia, lähettää palvelin viestin takaisin käyttäjälle muokkauksen ollessa valmis.

DDP:n datan käsittelyllä taas tarkoitetaan viestejä, jotka kulkevat asiakkaan ja palvelimen välillä, kun asiakas on tilannut jonkin julkaisun palvelimelta. Asiakkaan tilattua datansa voi palvelin lähettää milloin tahansa DDP–viestejä, jotka indikoivat muutoksia asiakkaan tilaamaan dataan. Kuvassa 9 on esimerkki DDP–viesteistä asiakkaan lähettäessä tilausviestin palvelimelle.



KUVA 9. Lisää DDP-viestejä

Kun kuvan mukainen tilaus on valmis, on yhteys muodostettu asiakkaan ja palvelimen datan välillä ja näinollen palvelin voi milloin tahansa lähettää asiakkaalle uusia DDP–viestejä ilman että asiakkaan pitäisi itse niitä pyytää.

Vaikka DDP on kehitetty osana Meteoria, ei se kuitenkaan ole Meteor–spesifinen. Meteor–asiakas ei tiedä puhuvansa Meteor–palvelimen kanssa, eikä Meteor–palvelin vastaavasti tiedä puhuvansa Meteor–asiakkaalle. Ne vain tietävät toistensa puhuvan DDP:tä. Tämä mahdollistaa muilla ohjelmointikielillä kehitettyjen asiakaslaitteiden yhdistämisen Meteor–palvelimeen tai vastaavasti Meteor–asiakkaan yhdistämisen millä tahansa kielellä kirjoitettuun DDP-palvelimeen. (Meteor DDP 2015.)

2.2.4 Livequery

Meteor Livequery on työkalu, jonka avulla voi tehdä ”eläviä” kyselyitä tietokantaan. Alkuperäisen kyselyn tuloksen lisäksi Livequery tarjoaa virran (stream) muokkausilmoituksia aina kun kyselyn tulos muuttuu. Livequeryn sisäinen toteutus riippuu tietokannasta, johon sillä otetaan yhteys. Livequeryllä on muutama strategia tämän muokkausilmoitusvirran luomiseen. Se voi joko yhdistää tietokannan kopioon (replication slave), asettaa erinäisiä tietokannan tapahtumia (trigger), käyttää tietokannan

natiivia julkaisu/tilaus–toimintoa tai pahimmassa tapauksessa turvautua kyselyihin (polling). (Meteor Livequery 2015.)

Livequerystä on kirjoitushetkellä olemassa tuotantovalmis toteutus MongoDB:lle ja kehitysvaiheessa oleva toteutus Redis–tietokannalle. MongoDB:n kanssa Livequery kytkeytyy tietokannan kopioon ja käyttää MongoDB:n tarjoamaa tapahtumalokia (replication log). (Meteor Livequery 2015.)

Livequery ei kuitenkaan ole täydellinen ratkaisu, sillä sitä ei olla rakennettu tietokantaan sisälle. Ensinnäkin Livequery vaatii admin–oikeudet palvelimella, jotta se voi lukea tapahtumalokia tai asettaa erilaisia tietokannan tapahtumia. Toisekseen Livequery ei ole tällä hetkellä kovin hyvin skaalautuva, jos käyttäjämäärät ovat isoja. Tapahtumalokin lukemiseen vaaditaan sellaista prosessoritehoa, jota ei muissa järjestelmissä vaadita. Livequery on kuitenkin vielä niin nuori, että sitä optimoidaan varmasti lähitulevaisuudessa. (Meteor Livequery 2015.)

2.2.5 MongoDB

MongoDB on alustariippumaton avoimen lähdekoodin NoSQL-tietokantajärjestelmä, joka pohjautuu dokumenttien tallentamiseen taulukoiden sijasta. Erona SQL–pohjaisiin järjestelmiin on muun muassa se, että NoSQL-tietokannassa kehittäjän ei tarvitse ennalta määritellä tiettyä skeemaa datalle ja että NoSQL–ajattelumalli sopii erittäin hyvin yhteen oliopohjaisen ohjelmoinnin kanssa. (MongoDB what is NoSQL 2015.) MongoDB on myös erittäin hyvin horisontaalisesti skaalautuva tietokantajärjestelmä ja sitä käyttävätkin monet varsin tunnetut yritykset, kuten esimerkiksi Electronic Arts, ebay ja Adobe (MongoDB scale 2015).

MDG valitsi MongoDB:n ensimmäiseksi tuetuksi tietokannaksi, koska sen avulla MDG:n ei tarvinnut luoda ORM–kerrosta (**O**bject **R**elational **M**apping) ja MongoDB sopi erittäin hyvin JavaScriptin kanssa yhteen (Quora why MongoDB 2014). Tuki muille tietokannoille on kuitenkin tulossa ja kolmannen osapuolen kirjastot tukevat jo nyt MySQL:ää sekä PostgreSQL:ää (Dascalescu Why Meteor 2015).

Meteorin tarjoama rajapinta MongoDB–kyselyille on hyvin samankaltainen MongoDB:n natiivin rajapinnan kanssa. Tämä auttaa Meteorin oppimisessa huomattavasti, jos MongoDB on jo ennalta tuttu. Kuvassa 10 vertaillaan Meteorin ja MongoDB:n kyselyitä keskenään.

```
40  // Meteor mongo
41  data.insert({"testdata" : data})
42  data.find({}, {fields : {"testdata" : 1}})
43  data.upsert({"_id" : "ecEASqBEusE9fCb8y"}, {"testdata" : "more testdata"})
44  data.remove({"fieldTwo" : "more testdata"})
45
46
47  // Plain MongoDB
48  db.data.insert({"testdata" : data})
49  db.data.find({}, {"testdata" : 1})
50  db.data.update({"_id" : "ecEASqBEusE9fCb8y"}, {$set : {"testdata" : "newData"}})
51  db.data.remove({"fieldTwo" : "more testdata"})
52
53
```

KUVA 10. Meteorin ja MongoDB:n tietokantakyselyiden vertailua

Kuten kuvasta ilmenee, syntaksissa ei juurikaan ole eroja. Kirjoitushetkellä Meteorin MongoDB–rajapinta ei kuitenkaan tue aivan jokaista natiivia MongoDB-operaatiota, kuten esimerkiksi taulukon tietyn indeksin muokkaamista. Vaikka joitain operaatioita joutuu tekemään sovelluksen tasolla, on datan käsittely silti erittäin kitkatonta MongoDB:n dokumenttien ollessa valmiiksi JSON–muodossa.

2.3 Paketit

Meteorin kehittäjien itse ylläpitämillä paketeilla pääsee jo hyvin pitkälle sovelluksen kehittämisessä, mutta ennen pitkää kehittäjällä tulee tarve toteuttaa jokin sovelluskohtainen ominaisuus, jonka toteuttamiseen eivät pelkät ydinpaketit riitä. Tähän ongelmaan Meteorin ekosysteemi tarjoaa monenlaisia kolmannen osapuolen paketteja, sillä Meteor mahdollistaa pakettien tekemisen ja jakamisen standardoidulla tyylillä.

Atmosphere on Meteorin oletuspalvelu paketinhallinnalle. Sen avulla voi etsiä paketteja kaikenlaisiin ongelmiin. Myös jo olemassa olevia JavaScript–kirjastoja on helppo paketoida Meteor–paketeiksi ja niitä onkin olemassa jo melkoinen määrä (esimerkiksi Bootstrap, jQuery, Moment.js...).

Tässä alaluvussa käyn läpi joitain Meteorin tärkeimmistä ja hyödyllisimmistä paketeista, joita myös itse sovelluksen teossa käytin.

2.3.1 Accounts

Hyvin yleinen ominaisuus verkkosovelluksissa on jonkinlainen tilinhallinta. Tämän avulla voidaan esimerkiksi erotella sivun sisältöä henkilön perusteella, voidaan hallita käyttäjien oikeuksia ja voidaan asettaa erilaisia rooleja. Meteor tarjoaa paketin nimeltä accounts, jonka avulla kehittäjä saa käytettäväkseen kokonaisen käyttäjätilijärjestelmän. Kuvassa 11 näytetään tämän paketin käyttöä.

example.js

```
Meteor.methods({
  registerUser: function(userName, userEmail, userPassword) {
    Accounts.createUser({
        email: userEmail,
        password: userPassword,
        profile: {
            userName: userName
        }
    });
  }
});

if (Meteor.isClient) {
  Template.example.events({
    'submit #registerUser': function(event) {
      event.preventDefault();

      var userName = $('#userName').val();
      var userEmail = $('#userEmail').val();
      var userPassword = $('#userPassword').val();

      Meteor.call("registerUser",
                  userName,
                  userEmail,
                  userPassword,
                  onUserRegistered);
    }
  });

  function onUserRegistered (error) {
    if (error) {
      // Handle error
    }
  }
}

if (Meteor.isServer) {
  // ...
}
```

example.html

```
<head>
  <title>example</title>
</head>

<body>
  {{> example}}
</body>

<template name="example">

  <form id="registerUser">
    <input type="text" id="userName">
    <input type="text" id="userEmail">
    <input type="password" id="userPassword">
    <input type="submit">
  </form>
</template>
```

KUVA 11. Accounts-paketin avulla luodaan käyttäjä

Kun accounts–paketti on lisätty projektiin, saa käyttöönsä paketin tarjoamat funktiot. Oikealla puolella kuvassa on käyttäjälle näkyvä lomake, jonka avulla tiedot otetaan vastaan. Vasemmalla näkyy lomakkeen ”submit”–tapahtumalle määritelty tapahtumankäsittelijäfunktio, jossa data haetaan ja registerUser–funktiota kutsutaan. Tämän funktion sisällä kutsutaan Accounts.createUser()–funktiota, jolle annetaan parametrinä JSON–muotoinen objekti, jossa tietyille avaimille laitetaan parametreinä

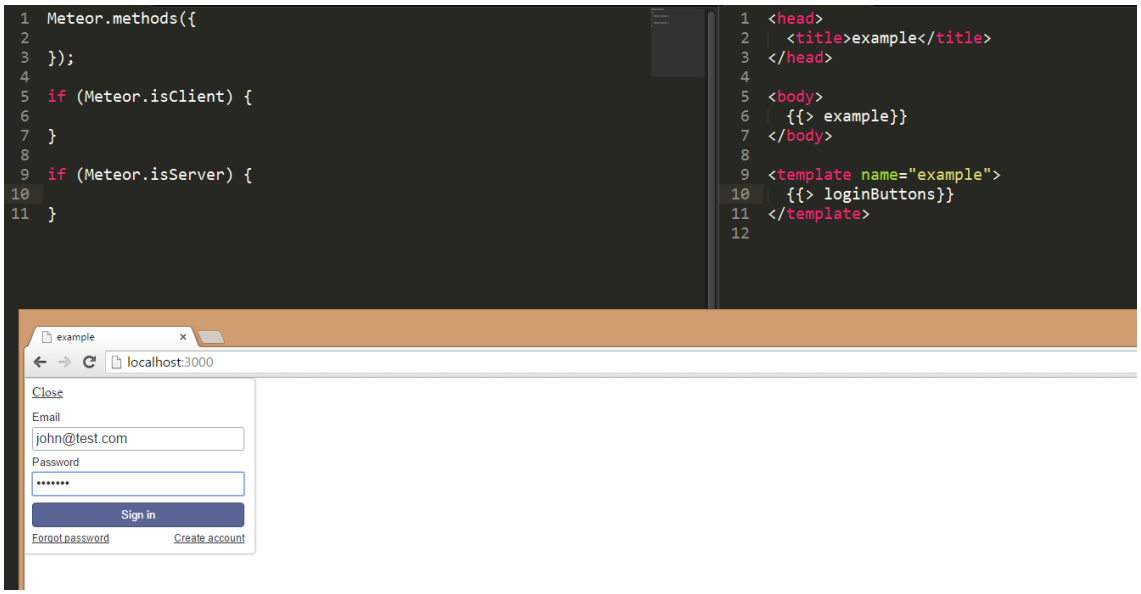
saadut käyttäjän tiedot. Tämän jälkeen data on tallennettu MongoDB–tietokantaan ”users”–nimiseen kokoelmaan. Kuvassa 12 näytetään, miten tieto näkyy tietokannassa.


```
{
  "_id": "w7nm9meqPQsArdAqz",
  "createdAt": ISODate("2015-10-29T10:58:56.485Z"),
  "services": {
    "password": {
      "bcrypt": "$2a$10$I1G2p4FsVVQ9jsCL1IJ.v.uT2tzfPUfvW6.NQOVq/kx1G.AwRfkjm"
    },
    "emails": [
      {
        "address": "john@test.com",
        "verified": false
      }
    ],
    "profile": {
      "userName": "John"
    }
  }
```


KUVA 12. Yhden käyttäjän tiedot MongoDB:ssä

Kenttä ”profile” on se paikka, johon kaikki itse luotu data laitetaan. Salasanat on tallennettu bcrypt-algoritmilla.

Kehittäjän ei myöskään tarvitse itse ohjelmoida käyttöliittymää tilinhallintaa varten, jos hän lataa ”accounts-ui”–paketin. Kuvassa 13 näytetään, kuinka kehittäjä voi saada täyden tilinhallintajärjestelmän käyttöliittymineen yhdellä rivillä koodia.



KUVA 13. Accounts ja accounts-ui yhdessä

Pakettien lisäämisen jälkeen kehittäjän tarvitsee siis vain sisällyttää loginButtons–template, jonka ”accounts-ui”–paketti tarjoaa.

Accounts–paketti on myös Trackerin huomioiva, eli sen avulla on mahdollista esimerkiksi näyttää latausikonia käyttäjän kirjautuessa sisään. Tämä tapahtuu kutsumalla Meteor.loggingIn()–funktiota. On myös mahdollista näyttää kirjautuneen henkilön profiilidata Meteor.user()–funktiolla. Näiden funktioiden palautusarvo muuttuu reaktiivisesti, eikä niitä tarvitse kehittäjän manuaalisesti kutsua tiettyinä ajan hetkinä. (Meteor accounts 2015.)

Meteorin oman tilinhallinnan lisäksi Meteorissa on myös mahdollista kirjautua sisään kolmannen osapuolen palveluntarjoajien, kuten Googlen, Facebookin tai Twitterin tileillä. Itseasiassa minkä tahansa OAuth–palvelun tarjoajan tili on mahdollista integroida Meteor–sovellukseen. (Meteor accounts 2015.)

2.3.2 Iron Router

Iron Router on kirjasto, jonka tehtävänä on määritellä mikä template renderöidään missäkin URLissa. Tämän lisäksi sillä voi määritellä, mikä data tilataan missäkin templatessa ja mitä tehdään ennen kuin data on saatu palvelimelta asiakkaan koneelle (esimerkiksi näytä lataustemplaatti).

Iron Router on ehkä tärkein (tai ainakin käytetyin) Meteorin paketeista. Ennen Iron Routeria käytettiin pääasiallisesti kahta pakettia: Meteor Router ja Mini Pages. Pakettien tekijät kuitenkin liittyivät yhteen ja näin Iron Router syntyi. Kuvassa 14 demonstroidaan joitain Iron Routerin perusominaisuuksista.

```
Router.route('/', function() {
  this.render('home');
});

Router.route('/page-one', {
  template: 'pageOne',
  waitOn: function () {
    return Meteor.subscribe("pageOneData", "<userId>");
  }
});

Router.route('/page-two/:testdata', {
  template: 'pageTwo',
  waitOn: function () {
    return [Meteor.subscribe("pageTwoData", "<userId>", this.params.testdata)
          , Meteor.subscribe("pageOneData", "<userId>")];
  }
});

if (Meteor.isClient) {

}

if (Meteor.isServer) {
  Meteor.publish("pageOneData", function(userId) {
    // return some data...
    return Meteor.users.find();
  });

  Meteor.publish("pageTwoData", function(userId, testdata) {
    console.log(testdata);
    // return some data...
    return Meteor.users.find();
  });
}
```

```
<head>
  <title>example</title>
</head>

<template name="home">
  <nav>
    <a href="/page-one"><button> Page One </button></a>
    <a href="/page-two/thisIsTestData"><button> Page Two </button></a>
  </nav>
</template>

<template name="pageOne">
  <h1> Page One </h1>
</template>

<template name="pageTwo">
  <h1> Page Two </h1>
</template>
```

KUVA 14. Iron Routerin perusominaisuuksia

Ensimmäisellä rivillä vasemmalla puolella on yksinkertaisin mahdollinen Iron Router -komento. Siinä kuvataan, että ”kun käyttäjä navigoi juurikansioon, renderöi home -niminen template”. Home–templaten sisällä on kaksi nappia, joita painamalla linkki vie joko URLiin ”/page-one” tai ”/page-two/thisIsTestData”. Näille URLeille on määritetty Iron Routerilla tiettyjä komentoja. Ensimmäisellä sivulla ilmoitetaan, että renderöidään template nimeltä ”pageOne” ja ennen kuin tämä tehdään, odotetaan ”pageOneData”–julkaisun datan saapumista. Toisella sivulla renderöidään template ”pageTwo” ja odotetaan kahden tilauksen datan saapumista. Toisella sivulla lähetetään myös ”pageTwoData”–tilaukselle parametrina ”this.params.testdata”, joka on se tekstinpätkä, joka reitissä on määritetty muuttujaksi ”:testdata”. Näin voidaan lähettää parametreja URLien kautta julkaisuille. Jos esimerkiksi halutaan keskustelupalstan ensimmäiset 20 viestiä, niin reitti voisi olla ”/posts/20”.

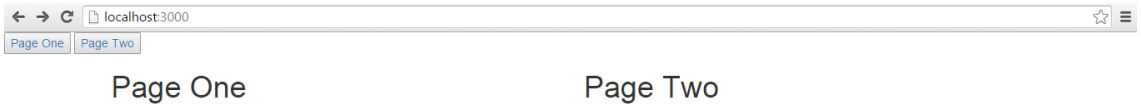
Iron Routerilla voi myös renderöidä templateja tiettyyn osaan sivustosta, jolloin osa sivusta pysyy samana kuin edellisellä sivulla. Kuvassa 15 on esimerkki sivun osaan renderöimisestä.

```
Router.route('/', function() {
  this.render('home');
});

Router.route('/page-one', function () {
  this.render('pageOne', {to: 'left'});
});

Router.route('/page-two/:testdata', function () {
  this.render('pageTwo', {to: 'right'});
});

if (Meteor.isClient) {

}

if (Meteor.isServer) {

}
```

```
<head>
  <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.
  com/bootstrap/3.3.5/css/bootstrap.min.css">
  <title>example</title>
</head>

<template name="home">
  <nav>
    <a href="/page-one"><button> Page One </button></a>
    <a href="/page-two/thisIsTestData"><button> Page Two </button></a>
  </nav>

  <div class="col-xs-4 col-xs-offset-1">
    {{> yield "left"}}
  </div>

  <div class="col-xs-4 col-xs-offset-1">
    {{> yield "right"}}
  </div>
</template>

<template name="pageOne">
  <h1> Page One </h1>
</template>

<template name="pageTwo">
  <h1> Page Two </h1>
</template>
```

KUVA 15. Sivun tiettyyn osaan renderöinti Iron Routerilla

Tässä HTML:n puolella oikealla on määritelty tiettyjä sivun osioita, joihin voi myöhemmin renderöidä templaten. {{> yield ’templaten-nimi’}} määrittää osion, johon voi Iron Routerin puolella viitata, kuten kuvan vasemmalla puolella näkyy. Kuvassa 16 on edellä mainitun koodin näkymä.



KUVA 16. Iron Routerilla monen sivun näkymät samalla sivulla

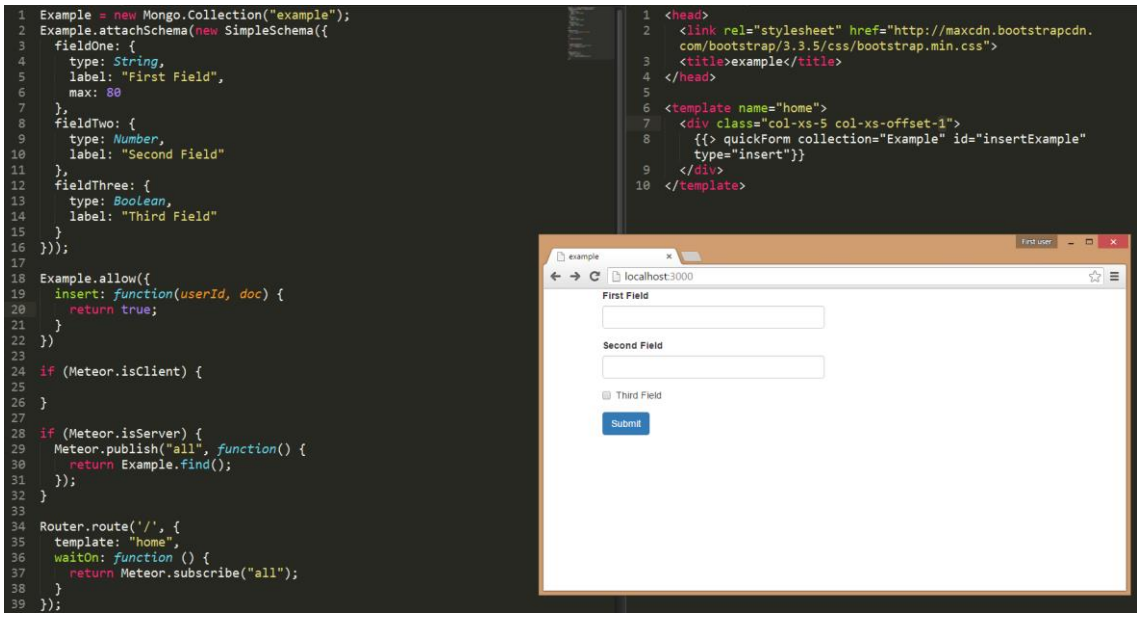
Iron Routerilla pystyy myös tekemään paljon muutakin, kuten määrittelemään toimintoja tiettyihin sivun renderöinnin vaiheisiin (hooks) ja määrittelemään kontrollereita reiteille (RouteController). Näitä kannattaa käyttää varsinkin silloin, jos reititykseen liittyy paljon logiikkaa. Muutoin pelkillä edellä mainituilla ominaisuuksilla pääsee jo varsin pitkälle.

2.3.3 Autoform

Päätin ottaa myös Autoform-nimisen paketin tähän alalukuun mukaan. Autoform ei ole MDG:n ylläpitämä paketti, mutta se on erittäin käytetty (kirjoitushetkellä asennuksia 111492 kappaletta) ja erittäin geneerinen paketti, jonka avulla kehittäjän ei tarvitse

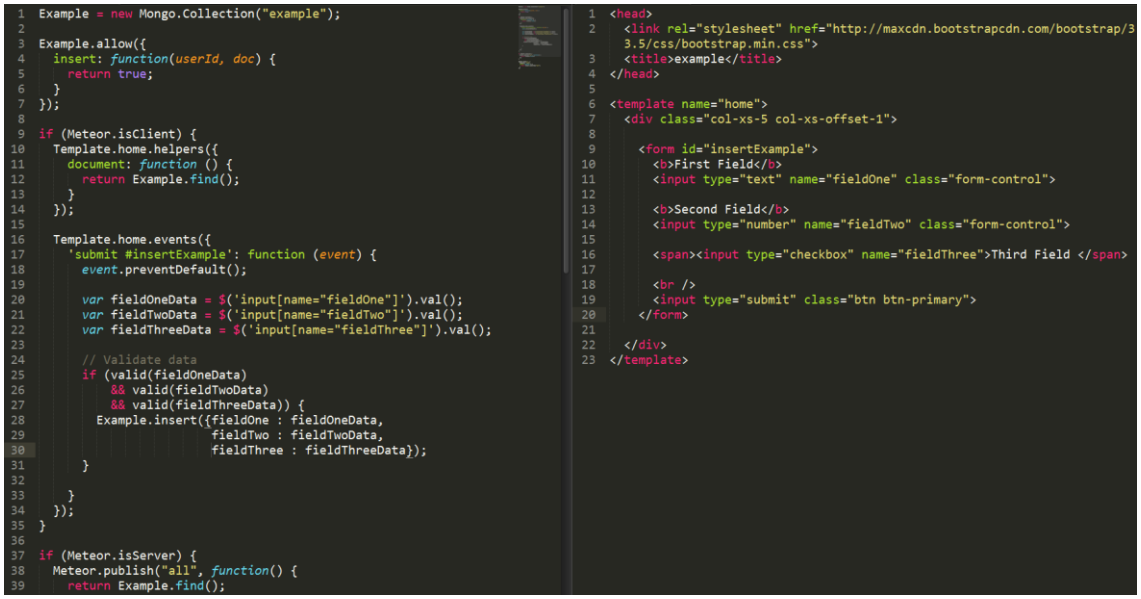
kirjoittaa lomakkeisiin liittyvää itseään toistavaa koodia. Tämän vuoksi Autoform on erittäin hyödyllinen paketti kaikenlaisiin sellaisiin sovelluksiin, joilla tallennetaan, muokataan ja poistetaan dataa tietokannasta ja tämän ansiosta se on mainitsemisen arvoinen.

Autoformilla kehittäjä pystyy yhdellä rivillä koodia luomaan lomake-käyttöliittymän parametrina annetulle kokoelmalle. Lomakkeessa on automaattisesti luonti- ja päivitystapahtumat (Atmosphere Autoform 2015). Autoformia kannattaa käyttää ”collection2”-paketin kanssa, jolloin datalle voi määritellä skeeman. Tällöin lomakkeelle tulee automaattisesti reaktiivinen varmennus (validation). Kuvassa 17 havainnollistetaan hieman autoformin käyttöä.



KUVA 17. Autoformin käyttöä

Kuvan 17 esimerkissä määritellään ensin kokoelma, sitten sen datalle asetetaan skeema, joka määrittelee esimerkiksi datan tyyppiä ja maksimipituutta. Tämän jälkeen HTML:n seassa yksi rivi koodia riittää lomakkeen luomiseen. Tässä lomakkeessa siis on automaattisesti datan eheyden tarkistus ja tiedon tallennus. Kuvassa 18 näytetään, miltä edellä mainittu ominaisuus saattaisi näyttää ilman autoformia.



```
Example = new Mongo.Collection("example");

Example.allow({
  insert: function(userId, doc) {
    return true;
  }
});

if (Meteor.isClient) {
  Template.home.helpers({
    document: function () {
      return Example.find();
    }
  });

  Template.home.events({
    'submit #insertExample': function (event) {
      event.preventDefault();

      var fieldOneData = $('input[name="fieldOne"]').val();
      var fieldTwoData = $('input[name="fieldTwo"]').val();
      var fieldThreeData = $('input[name="fieldThree"]').val();

      // Validate data
      if (valid(fieldOneData)
          && valid(fieldTwoData)
          && valid(fieldThreeData)) {
        Example.insert({fieldOne : fieldOneData,
                        fieldTwo : fieldTwoData,
                        fieldThree : fieldThreeData});
      }

    }
  });
}

if (Meteor.isServer) {
  Meteor.publish("all", function() {
    return Example.find();
```

```
<head>
  <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3
  3.5/css/bootstrap.min.css">
  <title>example</title>
</head>

<template name="home">
  <div class="col-xs-5 col-xs-offset-1">

    <form id="insertExample">
      <b>First Field</b>
      <input type="text" name="fieldOne" class="form-control">

      <b>Second Field</b>
      <input type="number" name="fieldTwo" class="form-control">

      <span><input type="checkbox" name="fieldThree">Third Field </span>

      <br />
      <input type="submit" class="btn btn-primary">
    </form>

  </div>
</template>
```

KUVA 18. Lomake-käyttöliittymä tapahtumineen ilman autoformia

Kuten nähdään, autoform säästää huomattavan määrän koodia varsinkin muokattavien kenttien määrän kasvaessa.

3 DIGIHARKKA-SOVELLUS

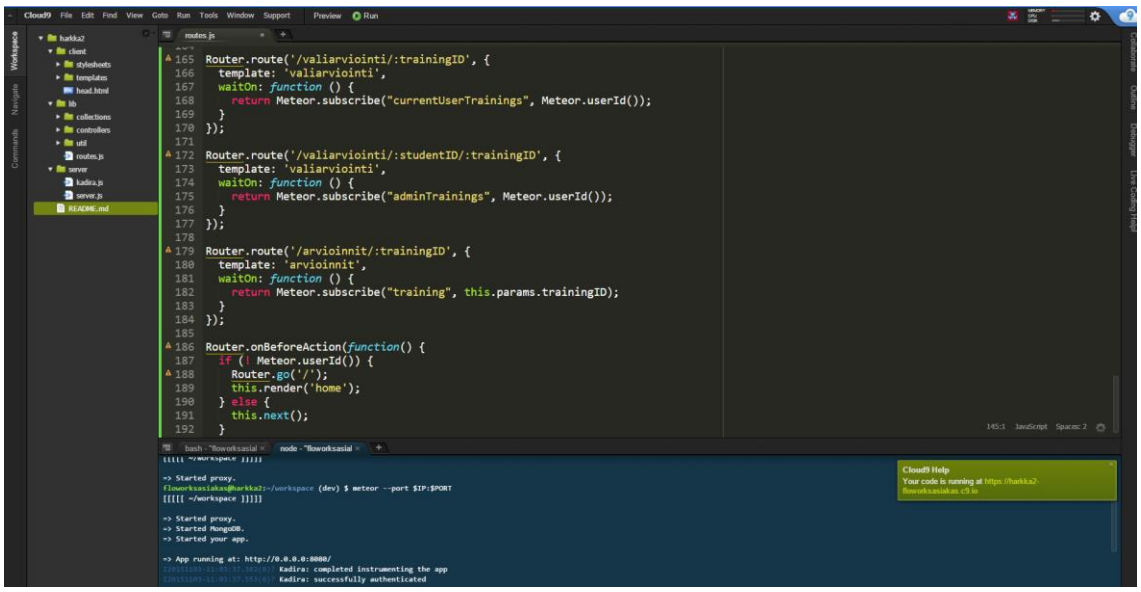
Tässä luvussa käyn läpi sovellusta, jonka Meteorilla toteutin. Kyseessä siis oli Tampereen ammattikorkeakoulun sairaanhoitajien koulutusohjelman ohjatun harjoittelun arviointiprosessin digitalisointi. Kerron ensin hieman työkaluista ja työmetodeista, joita käytettiin sovelluksen teon aikana. Tämän jälkeen kerron ominaisuuskohtaisesti, kuinka suunnittelimme ja toteutimme kaikki ne asiakkaan vaatimukset, joita meille annettiin.

3.1 Työkalut ja työmetodit

Tässä alaluvussa kerron projektin aikana käytetyistä työkaluista ja työmetodeista, alkaen kehitysympäristöstä.

3.1.1 Cloud9

Kehitysympäristönä käytettiin verkossa sijaitsevaa integroitua kehitysympäristöä (Integrated Development Environment) nimeltä Cloud9. Päädyimme tähän, sillä tämän ansiosta omalle koneelle ei tarvinnut asentaa Meteor-kehitysympäristöä tai itse Meteoria. Minulla oli aiemmista projekteista kokemusta Cloud9:n käytöstä, ja se myös tarjosi valmiin templaatin Meteor-projekteille. Mielestäni Cloud9:ssä oli myös erittäin miellyttävän näköinen käyttöliittymä, jota jaksoi katsoa monia tunteja päivässä. Kuvassa 19 on Cloud9:n käyttöliittymä.

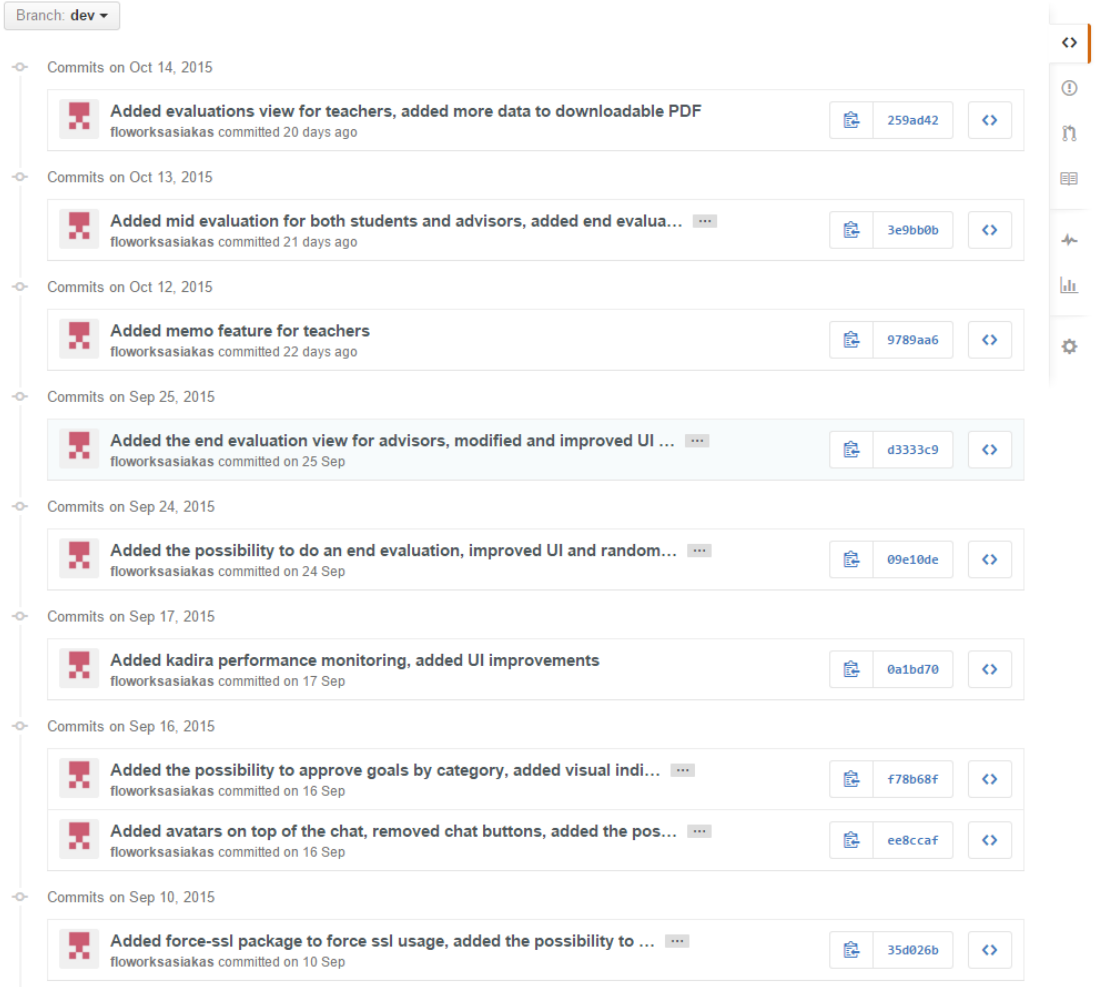


KUVA 19. Cloud9:n käyttöliittymä

Cloud9:n tarjoaa myös palvelimia kehitysvaiheessa sovelluksen testaamisessa. Cloud9 myös mahdollisti sen, että versionhallintaa ei ollut pakko käyttää. Projektin alkupäässä emme käyttäneetkään minkäänlaista versionhallintaa, sillä pilvessä oleva kehitysympäristö takasi sen, että koodi oli aina uusimmassa versiossa. Projektin koodimäärän kasvaessa ja toisen kehittäjän lähdettyä projektista päätin kuitenkin siirtyä käyttämään versionhallintaa. Versionhallintajärjestelmäksi valitsin Gitin, sillä siitä oli kokemusta aiemmista projekteista.

3.1.2 Git

Git on Linus Torvaldsin alunperin kehittämä hajautettu versionhallintajärjestelmä. Käyttäessäni Gitiä ja Githubia en kuitenkaan käyttänyt sitä ”oikeaoppisesti” tekemällä uusia haaroja (branch) jokaista uutta ominaisuutta varten. Koska tässä vaiheessa projektissa oli ohjelmoimassa vain yksi henkilö, päätin yksinkertaistaa asioita ja tehdä kaikki koodin ”commitit” dev-haaraan. Kuvassa 20 näytetään joitain projektin committeja GitHubissa.



KUVA 20. Git-committeja GitHubissa

Gitin käyttö osoittautui erinomaiseksi ideaksi, sillä yhdessä kohtaa projektia päivitin useita kolmannen osapuolen paketteja kerralla ja eräs paketeista rikkoi taaksepäin yhteensopivuuden (backwards compatibility). Gitin committeja tarkastelemalla sain selville kyseisen paketin, peruutin tämän paketin päivityksen ja kaikki taas toimi.

3.1.3 Tapaamiset

Pidimme kerran viikossa Floworksilla sisäisen tapaamisen, jossa kävimme läpi viikon aikana tekemiämme ominaisuuksia ja kohtaamiamme haasteita. Kirjasimme ylös tämän kaiken ja arvioimme myös edistymistä asteikolla 1-5. Tämä auttoi kärryillä pysymisessä, sillä projekti oli melko pitkä ja ominaisuuksiakin tuli todella paljon. Edistymisen näkeminen myös toi lisämotivaatiota ja positiivista painetta jatkaa eteenpäin.

Viikottaisten tapaamisten lisäksi tapasimme sairaanhoitajien koulutusalan yhteyshenkilöitä aina silloin tällöin. Näissä tapaamisissa demonstroimme sovellusta ja saimme palautetta, mikä usein oli erittäin positiivista. Uusia kehitysehdotuksiakin tuli paljon, mutta ihan kaikkea ei ehditty toteuttamaan.

Mitään varsinaista sovelluskehitysmetodologiaa emme käyttäneet, emmekä mielestäni sellaista tarvinneet. Sovelluskehitystiimi oli niin pieni, että päiväkirjamaiset arvioinnit ja viikottaiset tapaamiset olivat riittäviä pitämään kaikki osapuolet ajantasalla ja sovelluksen kehityksen raiteillaan.

3.2 Ominaisuudet

Tässä alaluvussa kerron ominaisuuskohtaisesti sovelluksen toteutuksesta: mihin ratkaisuun päädyttiin ja miksi.

3.2.1 Tilijärjestelmä

Ensimmäinen ongelma, joka sovelluksella tuli ratkaista, oli erilaisten näkymien luominen eri käyttäjille. Tätä varten meidän piti erotella käyttäjien roolit jotenkin toisistaan, joten päädyimme käyttämään MDG:n ylläpitämää Accounts-pakettia. Kuten alaluvussa ”2.3.1 Accounts” mainitsin, paketin avulla luoduille käyttäjille luodaan ”profile”-kenttä tietokantaan. Tähän kenttään pystyy laittamaan kaiken käyttäjäspesifisen datan, joten sen avulla pystyimme erottelemaan opiskelijat, ohjaajat ja opettajat toisistaan. Kuvassa 21 on käyttäjän kirjautumis/rekisteröinti-ikkuna, joka sijaitsee sovelluksen aloitussivulla.

KUVA 21. Käyttäjän kirjautumis/rekisteröintinäkymä

Käyttäjän luodessa tilin hänen sähköpostiinsa lähetetään varmistusviesti, jolla varmennetaan käyttäjä. Tämänkin ominaisuuden Accounts-paketti tarjoaa valmiina. Kehitysvaihetta varten piti kuitenkin tehdä monia testitilejä, joten käyttöliittymän kautta on myös mahdollista rekisteröityä ilman sähköpostivarmennusta.

Salasanan unohdus on melko yleistä sovelluksissa, joissa tilejä tehdään. Tätä varten oli hyvä luoda mekanismi, jolla salasanan voi asettaa uudelleen. Tämäkin oli Accounts-paketissa erittäin helppoa: yhden funktion kutsu, jossa käyttäjä antaa sähköpostiosoitteensa. Sähköpostiin sitten lähetetään linkki, jonka avulla käyttäjä pääsee sovellukseen vaihtamaan salasanansa. Kuvissa 22 ja 23 demonstroidaan Accounts-paketin ominaisuuksia koodissa.

```
// Create the user.
Accounts.createUser({
    email: email,
    password: password,
    profile: {
        userName: username,
        accountType: accountType
    }
});

// Send a verification link to the email of the given user.
Accounts.sendVerificationEmail(userId);

// Handle the event of user clicking the verification link.
Accounts.onEmailVerificationLink(function(token, onEmailVerificationDone) {
    Accounts.verifyEmail(token, onEmailVerificationDone);
});
```

KUVA 22. Tilin luominen ja sähköpostin varmennus

```
// Email templates used on the specified events (email verification, password reset)
// Configured on the server code
Accounts.emailTemplates.verifyEmail.subject = function(user) {
    return "Sähköpostiosoitteen varmennus käyttäjälle " + user.profile.userName;
};

Accounts.emailTemplates.verifyEmail.html = function(user, url) {
    var html = "<p>Klikkaa linkkiä varmentaaksesi sähköpostiosoite</p>"
             + "<a href='" + url + "' >Varmennus</a>";
    return html;
};

Accounts.emailTemplates.resetPassword.subject = function(user) {
    return "Salasanan nollaus käyttäjälle " + user.profile.userName;
};

Accounts.emailTemplates.resetPassword.html = function(user, url) {
    var html = "<p>Klikkaa linkkiä vaihtaaksesi salasanasi</p>"
             + "<a href='" + url + "' >Vaihda salasana</a>";
    return html;
};

/**
 * token = the String token received from the reset password link
 * onPasswordResetDone = callback function after password has reset
 */
Accounts.onResetPasswordLink(function(token, onPasswordResetDone) {
  // newPassword = Take new password as user input
  Accounts.resetPassword(token, newPassword, onPasswordResetDone);
});
```

KUVA 23. Accounts-paketin sähköpostien konfigurointia ja salasanan resetointi

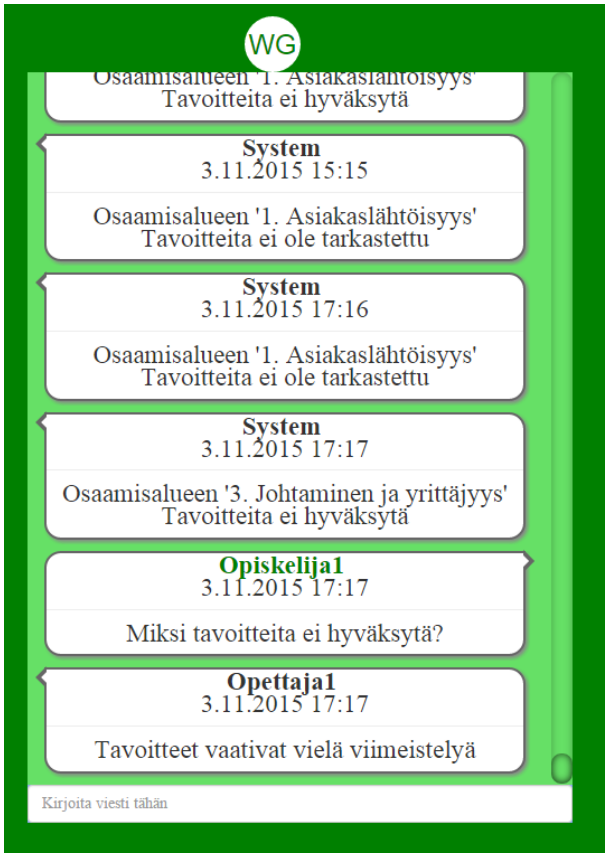
Accounts-paketti mahdollisti kokonaisen tilinhallintajärjestelmän luonnin erittäin vähällä vaivalla. Saimme tehtyä tämän ominaisuuden todella nopeasti, mikä vapautti aikaa muihin ominaisuuksiin.

3.2.2 Chat

Seuraava suuri ominaisuus listalla oli chat, jonka avulla kaikki kolme käyttäjätyyppiä pystyisivät kommunikoimaan mahdollisimman vaivattomasti keskenään. Chat on melko yleinen ominaisuus, joten siitä olettaisi olevan olemassa valmis paketti. Atmospherea kuitenkin selattaessa ei paketteja kovin montaa löytynyt ja niitä joita löytyi oli asennettu todella vähän. Tämän johdosta päädyimme rakentamaan chatin itse, sillä olihan Meteorista luvattu, että yksinkertaisen chatin pystyy luomaan jopa tunnissa.

Meteoriin sisäänrakennettu reaktiivisuus ja julkaisija/tilaajamalli olivat otollisia alustoja chatin kehittämiselle. Idea oli siis erittäin suoraviivainen: käyttäjälle tarjotaan syötekenttä, jonka avulla viestin voi lähettää. Tämän päällä näytetään lista viestejä, jotka

käyttäjä on tilannut palvelimelta. Kun uusi viesti lisätään, tulee se näkyvien viestien alle. Kuvassa 24 on chatin näkymä.

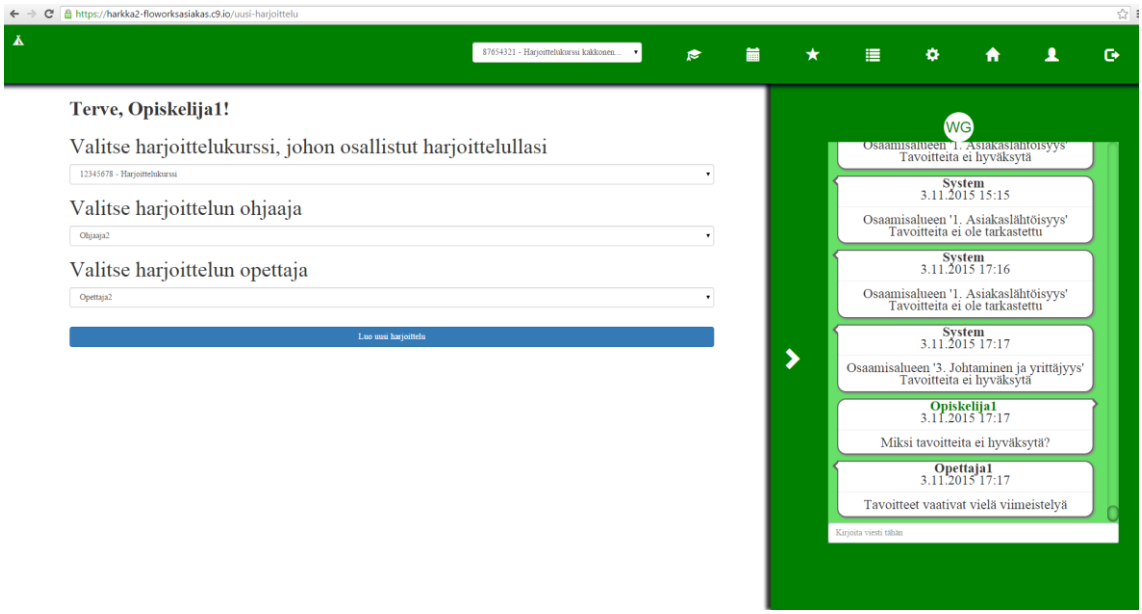


KUVA 24. Chatin näkymä

Tektsikenttä on lomake (form), jonka submit-tapahtumaan on asetettu tapahtumankäsittelijäfunktio Meteorin events-syntaksilla. Tämän funktion sisällä kutsutaan palvelimen metodia tekstin ollessa parametrinä (Remote Procedure Call). Palvelimen puolella tieto tallennetaan tietokantaan. Käyttäjän omissa viesteissä viestin lähettäjän nimi näkyy vihreällä. Puhekuplan ”kolmio” on toisella puolella puhekuplaa, kuin muilla viestittelijöillä.

3.2.3 Sivun asemointi

Sivun asemointia voi ajatella yhtenä ominaisuutena, sillä alusta alkaen haluttiin, että chat pysyisi kokoajan näkyvissä, kun sovelluksessa navigoidaan sivulta toiselle. Iron Routerilla tämä oli onneksi erittäin helppo tehdä. Kuvassa 25 nähdään opiskelijan näkymästä sivun yleinen asetelma.

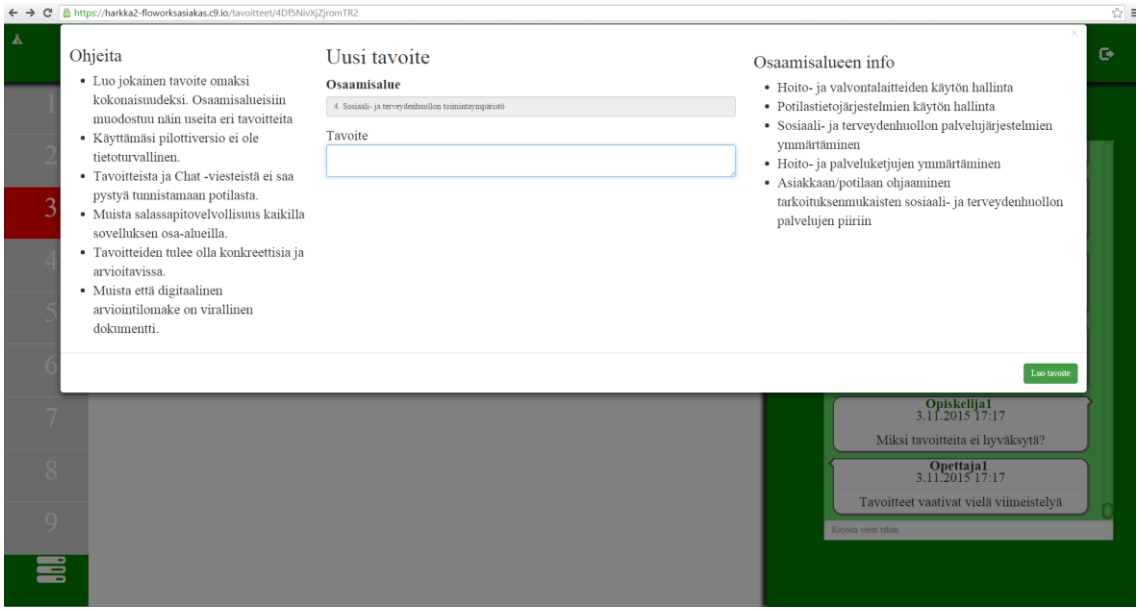


KUVA 25. Opiskelijan yleisnäkymä

Iron Routerin yield-templaatin avulla voitiin renderöidä chat-template sivun oikeaan laitaan. Sivun vasen puoli (valkoinen tausta) määritellään pääalueeksi, johon kaikki muut templatet oletuksena renderöidään. Sivun ylälaidan navigaatiopalkki saatiin pysymään paikallaan Bootstrapin luokkamäärityksillä.

3.2.4 Profiilin, harjoittelun ja tavoitteiden luominen

Opiskelijan harjoitteluun liittyvän datan (profiilitiedot, mikä harjoittelu kyseessä, tavoitteet) luominen oli melko vaivatonta. Autoformin avulla saatiin kaikki data tallennettua, muokattua ja poistettua pienellä määrällä koodia. Profiilin ja harjoittelun luonti oli erittäin helppoa, mutta tavoitteiden luomisen käyttöliittymän kanssa oli pieniä haasteita. Käyttöliittymän haluttiin olevan dialogi, jonka sisällä näkyisivät kyseisen tavoitteen osaamisalueen kuvaukset. Kuvassa 26 on tavoitteiden luonti-ikkuna.



KUVA 26. Tavoitteiden luonti-ikkuna

Itse dialogi oli helppo saada aikaan Bootbox-paketilla, mutta templaten renderöinti dialogin sisälle ei ollut ihan yksinkertainen tehtävä. Tämän ominaisuuden kanssa piti ensimmäistä ja ainoaa kertaa sovelluksessa käyttää suoraan Blazen funktioita. Kuvassa 27 näytetään koodi, jolla sain templaten renderöityä Bootbox-dialogin sisälle.

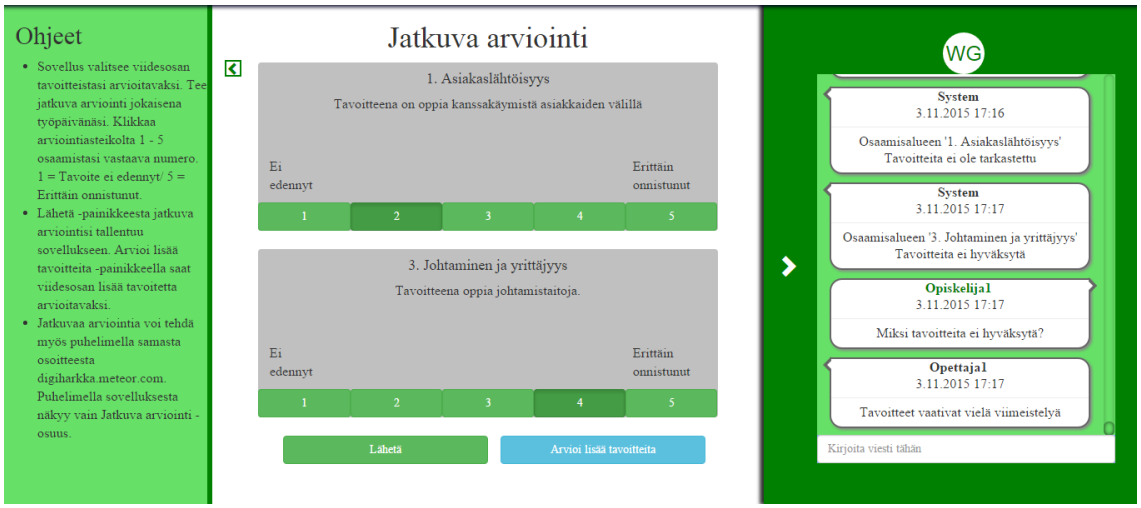
```
'click #newTavoite': function(event) {
  bootbox.dialog({
    message: renderTmp(Template.uusiTavoite, {"category" : category.get(), "dialog" : true}),
    className: "bootboxLarge",
    buttons: {
      success: {
        label: "Luo tavoite",
        className: "btn-success",
        callback: function () {
          var category = $('#category').val();
          var goal = $('#goal').val();
          var state = $('#state').val();

          Tavoitteet.insert({"category" : category
                             , "goal" : goal
                             , "state" : state
                             , "trainingID" : Router.current().params.trainingID});
        }
      }
    }
  });
}

function renderTmp (template, data) {
    var node = document.createElement("div");
    document.body.appendChild(node);
    Blaze.renderWithData(template, data, node);
    return node;
}
```

KUVA 27. Templaten renderöinti bootbox-dialogiin Blazella

Tavoitteiden luonnin käyttöliittymän lisäksi ei tietojen luomisessa ollut yhtään ongelmia pääosin autoformin ansiosta.

3.2.5 Jatkuva arviointi

Jatkuva arviointi tarkoittaa opiskelijan omien tavoitteiden toteutumisen/onnistumisen arviointia päivittäin/viikoittain. Jatkuvaa arviointia opiskelijat eivät olleet tehneet aiemmalla paperilomakkeella. Tämän vuoksi oli tärkeää, että jatkuvan arvioinnin sivulla näkyisivät ohjeet arvioinnin tekoon. Tämä ohje-alue toteutettiin puhtaalla CSS:llä, jotta sivu ei monimutkaistuisi liian monella sisäkkäisellä templatella. Kuvassa 28 näytetään jatkuvan arvioinnin näkymä pöytäkoneella.



KUVA 28. Jatkuva arviointi pöytäkoneella

Koska jatkuvaa arviointia käytetään sovelluksessa jopa päivittäin, haluttiin, että se pystytään tekemään puhelimella. Meteorilla tämäkin oli erittäin helppoa, sillä nopean tiedonhaun jälkeen löytyi paketti nimeltä device-detection. Tätä pakettia käyttäen laitteen tunnistaminen oli kuvan 29 kaltaista.

```
{{#if isPhone}}
  {{> mobileTemplate}}
{{else}}
  {{> desktopTemplate}}
{{/if}}
```

KUVA 29. Laitteen tunnistaminen device-detection paketilla

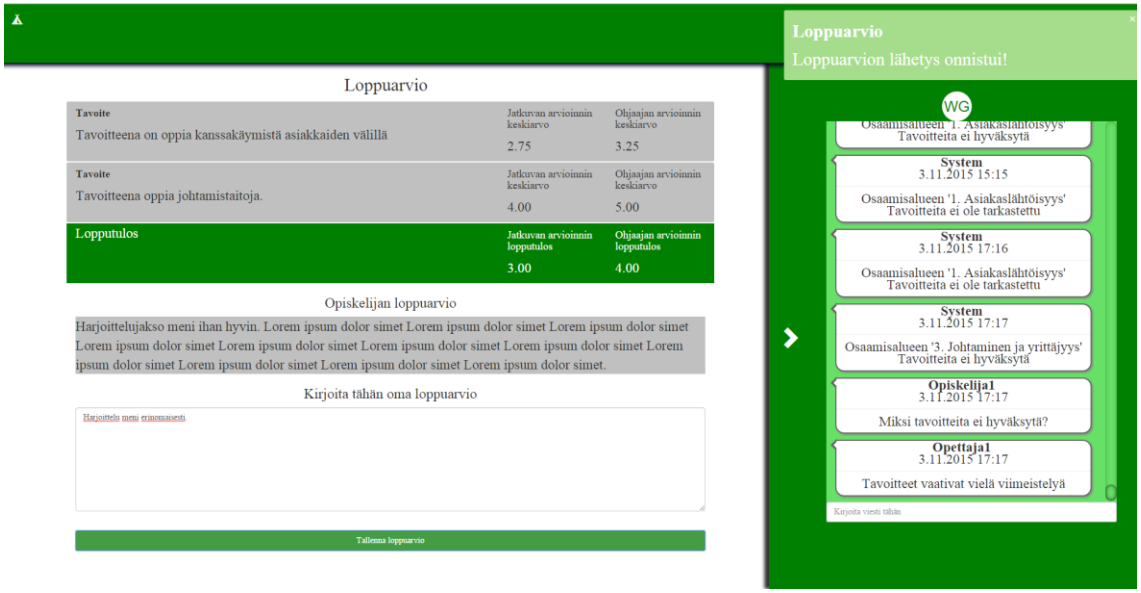
Tämä mahdollisti oman käyttöliittymän tekemistä puhelimille. Puhelimen ruudulla haluttiin näkyvän pelkkä arviointiosa ilman ohjeita. Kuvassa 30 näytetään puhelimen jatkuva arviointi –käyttöliittymä.

KUVA 30. Jatkuva arviointi puhelimella

3.2.6 Väli- ja loppuarviointi

Väli- ja loppuarviointien spesifikaatiot olivat hyvin selkeät. Opiskelija tekee väliaikaisen itsearvioinnin harjoittelun keskivaiheilla. Tämän jälkeen ohjaaja kirjoittaa omansa, samalla nähden opiskelijan itsearvioinnin. Molemmat arvioinnit ovat pelkkää tekstiä, joten näiden ominaisuuksien kanssa ei ollut ongelmia.

Loppuarviointi on muuten tismalleen sama kuin väliarviointi, mutta näkymään haluttiin harjoittelun jatkuvan arvioinnin tavoitteiden keskiarvot, jotta ohjaaja voisi muodostaa tarkemman mielipiteen harjoittelun arvioinnista. Kuvassa 31 nähdään ohjaajan loppuarvionäkymä.



KUVA 31. Ohjaajan loppuarvionäkymä

Lopputuloksen keskiarvo on painotettu keskiarvo jokaisesta arvioinnista: jos yhtä tavoitetta arvioi monta kertaa ja toista vain kerran, painaa ensimmäisen tavoitteen arvio lopputuloksessa enemmän.

3.2.7 Lataaminen PDF:nä

Sovellukseen piti myös saada mahdollisuus tulostaa lomake jollain tavalla, jotta siitä olisi olemassa paperiversio. Päätin toteuttaa tämän siten, että opiskelija voi ladata tietonsa PDF-muodossa ja sitten tulostaa myöhemmin tämän PDF-dokumentin. Pikaisen haun jälkeen löytyi muutamia vaihtoehtoisia paketteja, joilla voisi konvertoida HTML:ää PDF:ksi. Valitsin paketin nimeltä pdfmake, sillä sen avulla saisi tyylejä aikaan ja se vaikutti varsin helppokäyttöiseltä. Kuvassa 32 demonstroidaan koodia, jolla PDF saadaan aikaan.

```
function saveAsPdf() {
  // Function returns an array of key-value pairs that make the columns of a table
  var pdfTableBody = getPdfTableBody();

  // Function returns an array of objects, which have a "text"-field
  // and optional options fields (eg. "fontSize" or "bold")
  var pdfGoalsText = getPdfGoalsText();

  // Define the document with the following contents and styles
  var pdfDocument = {
      content: [
      {
        table: {
          widths: [ 'auto', 'auto'],
          body: pdfTableBody
        }
      },

      { text: pdfGoalsText, margin: [0, 20] }
    ]
  };

  var currentUserName = Meteor.user().profile.userName;

  // Create the PDF and download it, with a name of the given parameter
  pdfMake.createPdf(pdfDocument).download("tavoitteet-" + currentUserName);
}
```

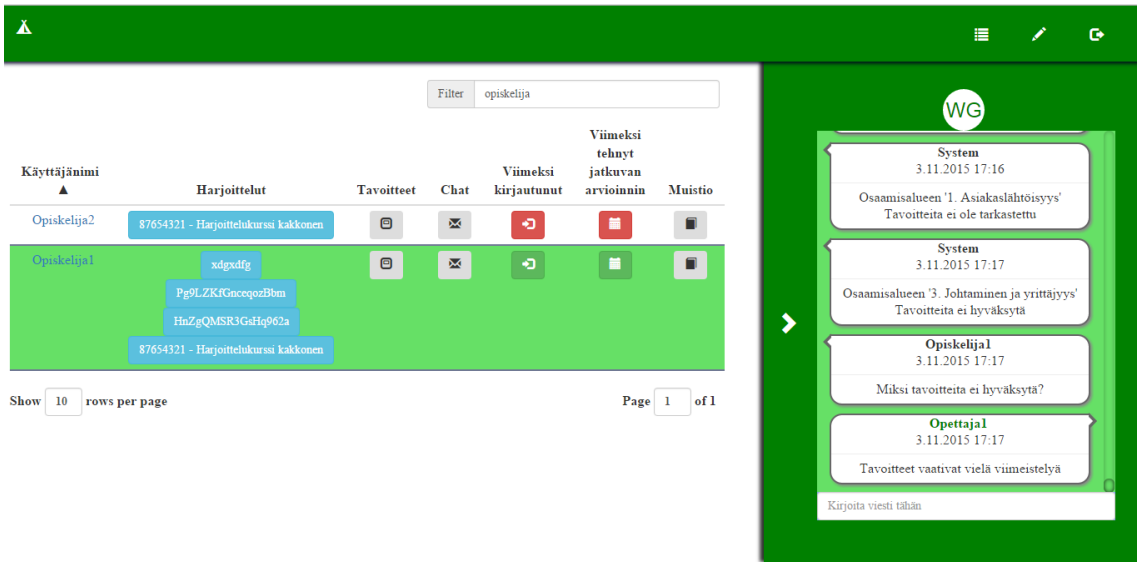
KUVA 32. Lomakkeen tallennus ja lataus PDF:nä

Kuvassa 32 määritellään kaksi sisältö-osiota PDF-lomakkeelle: taulukko-osio ja teksti-osio. Taulukko-osioon haetaan runko getPdfTableBody-funktiolla. Tämä funktio palauttaa taulukon avain-arvo pareja, jotka muodostavat PDF:ssä näkyvän taulukon sarakkeet. Teksti-osioon haetaan funktiolla getPdfGoalsText taulukko objekteja, joilla on pakollisena kenttänä ”text” ja valinnaisina kenttinä esimerkiksi ”fontSize” tai ”bold”. ”text”-kenttään tulee näkyvä teksti ja valinnaisiin kenttiin tekstin tyylimäärittelyjä. Näiden tietojen avulla saadaan PDF-lomakkeeseen ohjelmallisesti näkymään tarvittavat asiat.

3.2.8 Opettajan ja ohjaajan päänäkymä

Opettajan ja ohjaajan näkymissä oli tärkeää, että mahdollisimman paljon dataa saatiin mahtumaan yhteen näkymään. Päädyimme ratkaisuun, jossa näytämme taulukon opiskelijoista ja heihin liittyvistä asioista. Aikani etsiskeltyäni sopivaa pakettia tätä ominaisuutta varten päädyin käyttämään ”reactive-table” –nimistä pakettia. Tämä paketti mahdollisti HTML:n renderöinnin taulukon soluihin, filtteröinnin käyttäjän syötteen

perusteella ja järjestämisen sarakkeen perusteella. Kuvassa 33 esitetään opettajan päänäkymää.

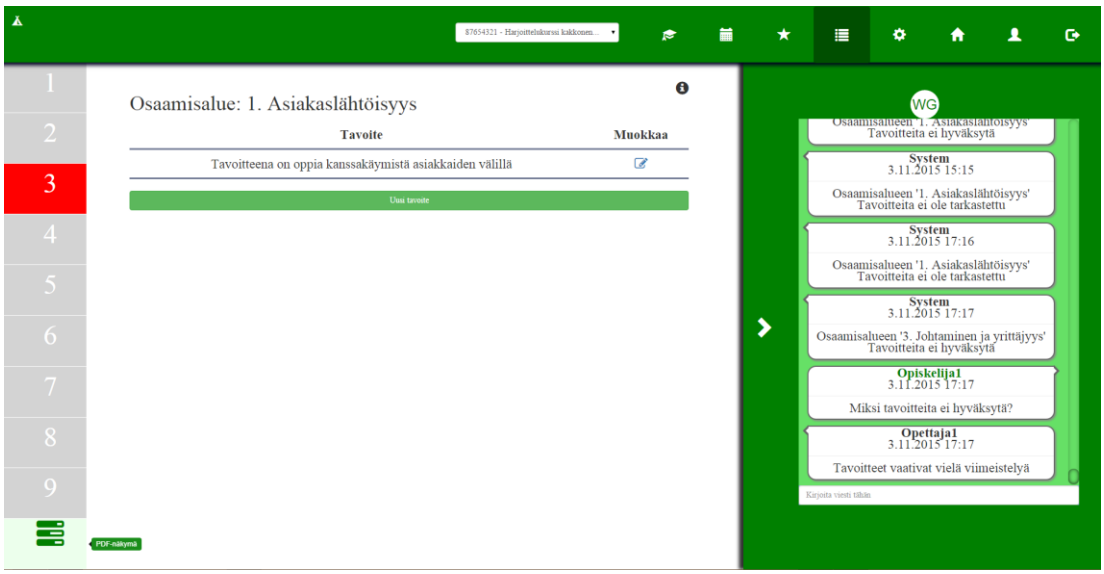


KUVA 33. Opettajan päänäkymä

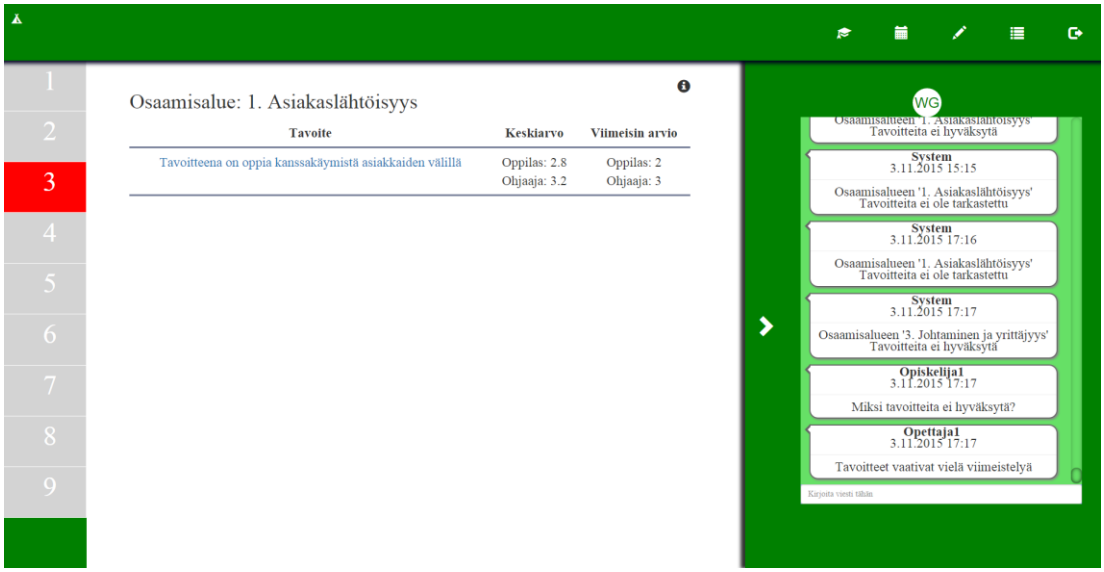
Tästä näkymästä opettaja näkee opiskelijan profiilidatan opiskelijan nimeä klikkaamalla, tavoitteet tavoite-nappia painamalla, sekä ajan milloin opiskelija on viimeksi kirjautunut ja tehnyt jatkuvaa arviointia. Muistio-napista opettaja voi kirjoittaa opiskelijakohtaisia muistiinpanoja, kuten milloin arviointikeskustelu on käytävä. Chat-napista opettaja voi vaihtaa chat-huoneeseen, missä kyseinen opiskelija on. Harjoittelu-nappia painamalla pääsee kyseisen harjoittelun tavoitenäkymään.

3.2.9 Opettajan ja ohjaajan tavoitenäkymä

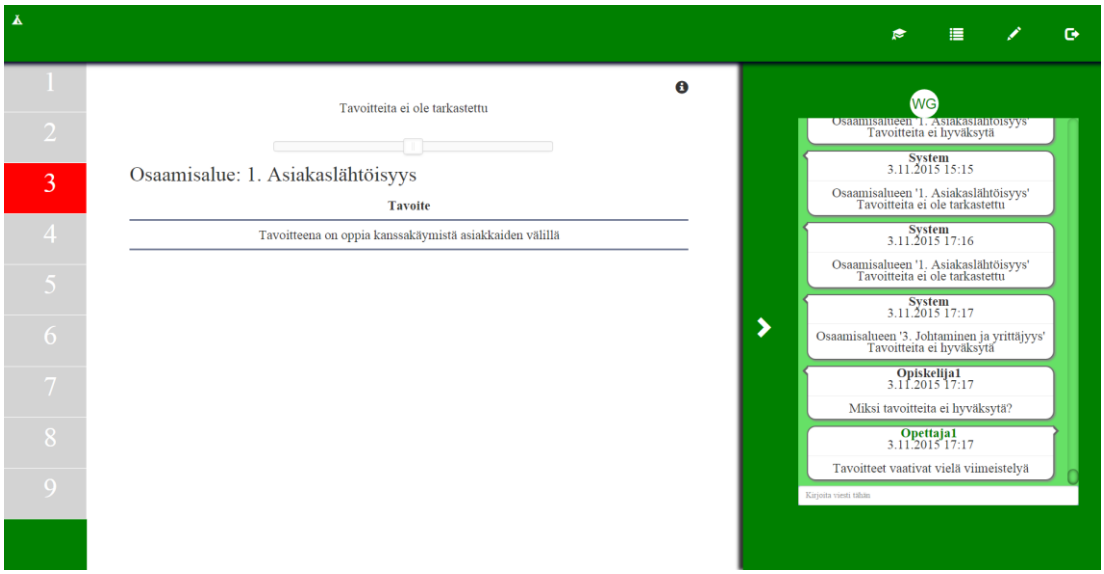
Opettajalle ja ohjaajalle oli molemmille tehtävä näkymä opiskelijan tavoitteisiin. Opettajalle, jotta hän voisi hyväksyä/hylätä ja kommentoida niitä, ja ohjaajalle, jotta hän voisi arvioida tavoitteiden onnistumista numeerisesti. Kaikille kolmelle roolille näytetään sama template pienin muutoksin. Kuvissa 34, 35 ja 36 näkyvät kaikkien roolien tavoite-näkymät.



KUVA 34. Opiskelijan tavoite-näkymä



KUVA 35. Ohjaajan tavoite-näkymä



KUVA 36. Opettajan tavoite-näkymä

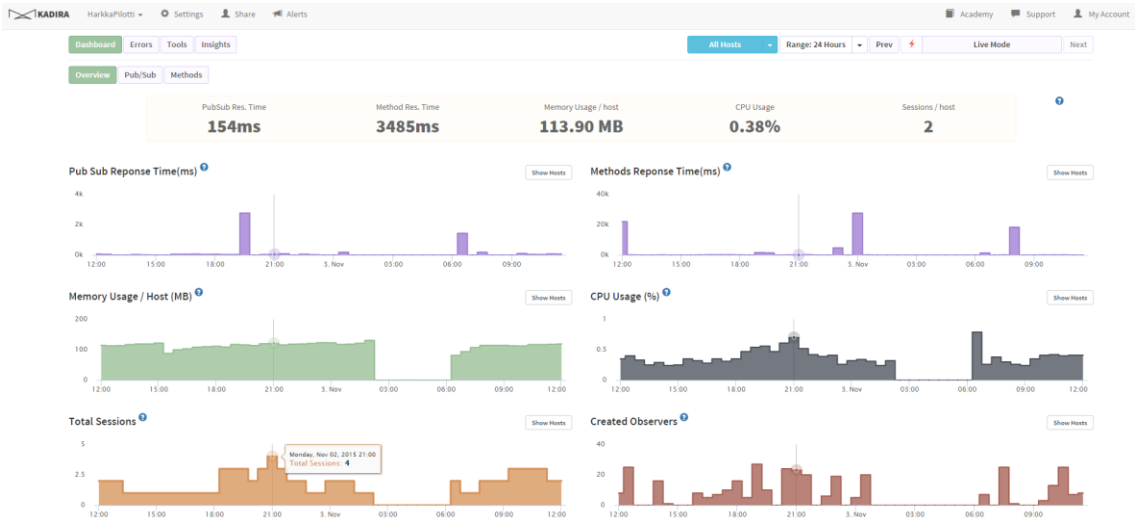
Opettajan tavoite-näkymässä opettaja voi siis joko hyväksyä tai hylätä osaamisalueen tavoitteet vierittämällä liukusäädintä (slider). Tämän ominaisuuden tarjosi paketti nimeltä nouislider. Kun osaamisalueen tavoitteet hyväksytään tai hylätään, muuttuu vasemmalla olevan osaamisalueindikaattorinumeron väri punaiseksi (hylätty), harmaaksi (ei tarkastettu) tai vihreäksi (hyväksytty). Tästä tapahtumasta lähtee myös viesti kyseisen opiskelijan chat-huoneeseen.

Ohjaajan tavoite-näkymässä ohjaaja voi klikata tiettyä tavoitetta arvioidakseen sen, tai navigaatiopalkissa olevaa kynää arvioidakseen kaikki opiskelijan tavoitteet. Ohjaaja näkee myös opiskelijan tavoitteiden jatkuvan arvioinnin keskiarvon, sekä oman arviointinsa keskiarvon.

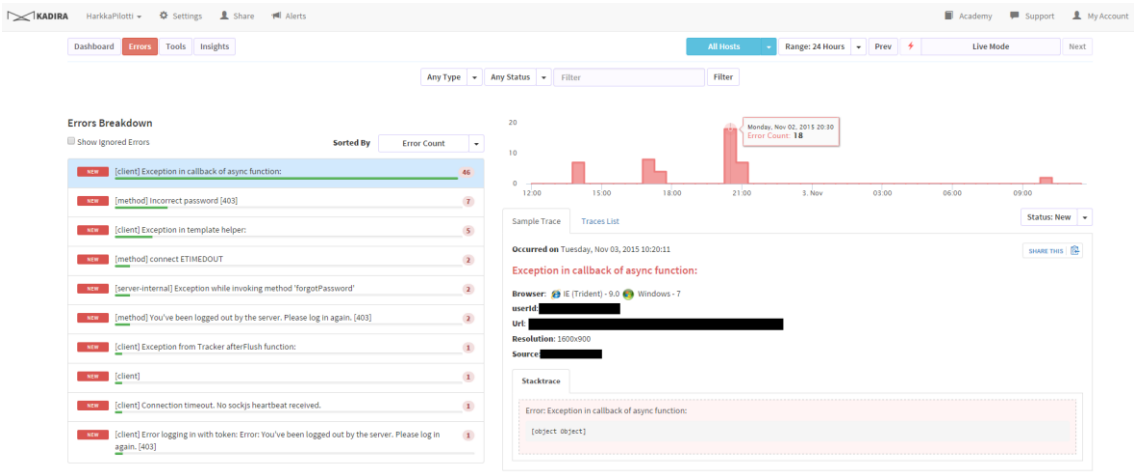
3.2.10 Muut ominaisuudet

Sovelluksen muista ominaisuuksista mainitsemisen arvoisia ovat muun muassa SSL (**S**ecure **S**ockets **L**ayer) protokollan pakotus ”force-ssl”–paketin avulla, notifikaatioiden näyttäminen notifications-paketilla tiettyjen tapahtumien jälkeen, aikaleimojen luonti momentjs-paketilla ja pyörivän latausikonin näyttäminen spin-paketilla.

Suorituskykyä ja virheiden monitorointia seuraan Kadira–nimisellä työkalulla (entinen Meteor APM). Kadiralla kehittäjä näkee palvelimen suorituskyvystä dataa, kuten muistin ja prosessorin käyttöprosentin tai julkaisujen vasteajan. Kadira myös tallentaa ja lähettää tiedon kaikista virheistä sen omille palvelimille, jotka sitten näkee Kadiran työpöydältä. Kuvissa 37 ja 38 näytetään Kadiran käyttöliittymää.



KUVA 37. Kadiran päänäkymä



KUVA 38. Kadiran virhenäkymä

Virhenäkymästä kehittäjä näkee muun muassa virheen yksityiskohtaisen kuvauksen, selaimen ja käyttöjärjestelmän, jolla virhe tapahtui, käyttäjän tunnisteen, virheen tapahtumahetken URLin, käyttäjän resoluution ja käyttäjän IP-osoitteen. Näiden tietojen avulla virheiden etsiminen ja niiden korjaaminen helpottuu huomattavasti.

4 POHDINTA

Tässä luvussa pohdin Meteoria kolmen monia askarruttavan kysymyksen tiimoilta: tietoturvallisuus, skaalautuvuus ja tuottavuus. Pohdin myös kokemuksiani Digiharkan toteutuksesta käyttäen Meteoria ja yhteenvedossa pohdin tuloksia sekä jatkokehitysideoita.

4.1 Tietoturvallisuus

Kun Meteorilla aloittaa ensimmäistä kertaa työskentelyn, on sillä erittäin helppo tehdä tietoturvatonta koodia. Jokainen Meteorilla luotu projekti alkaa oletusarvoisesti autopublish- ja insecure–paketeilla. Autopublish julkaisee koko tietokannan asiakaskoneille ja insecure mahdollistaa kaikki tietokantakyselyt jokaiselle asiakkaalle. Nämä paketit mahdollistavat nopean prototypoinnin, mutta tuotantoon mennessä tulisi nämä paketit poistaa. Jos pakettien poistaminen unohtuu, on jokaisella käyttäjällä admin-tason oikeudet sovellukseen.

Asiakkaan selaimessa sijaitseva JavaScript-konsoli mahdollistaa tietokantakyselyiden ja metodeiden kutsun suoraan. Tämä voi olla kehittäjän kannalta erittäin ikävä ominaisuus, sillä kokoelmien muokkausoikeuksien ja metodien parametrien tarkistuksen kanssa täytyy olla todella tarkkana, jotta vihamieliset komennot voidaan torjua. Tällaisia tilanteita varten kokoelmille voi määritellä allow/deny-takaisinkutsufunktioita (callback), joissa määritellään kriteerit muokkausoperaatioiden onnistumisille/hylkäämisille. Metodien sisällä kehittäjän on tarkastettava parametrien eheys. Tämä onnistuu esimerkiksi ”audit-argument-checks”-paketilla, joka pakottaa parametrien tarkistuksen metodeissa ja julkaisufunktioissa.

Hyvänä puolena Meteorin tietoturvallisuudessa voidaan pitää sitä, että se käyttää WebSocket-teknologiaa tiedonsiirrossa ensimmäisen latauksen jälkeen. Näin ollen Meteorin ei tarvitse käyttää evästeitä (cookie), joten XSRF-hyökkäykset (Cross-site request forgery) ovat mahdottomia. WebSocketien käytön myötä Meteor on kuitenkin haavoittuvainen DOS-hyökkäyksiä (Denial of Service) kohtaan. Tätä varten on kehitetty sovelluksen tasolla toimiva palomuuri nimeltä Sikka, joka estää käyttäjän IP-osoitteen ja

pyytää ihmisvarmennusta, jos kyseisestä IP-osoitteesta huomataan tulevan liian paljon liikennettä. (Meteorhacks Sikka 2015.)

Meteorissa on siis helppo tehdä virheitä tietoturvan kanssa, mutta niitä on myös helppo korjata. Yleisesti Meteorin tietoturva on mielestäni hyvä, kunhan nämä edellämainitut toimenpiteet muistaa tehdä.

4.2 Skaalautuvuus

Skaalautuvuus on toinen monia mietityttävä kysymys, kun Meteoriin alkaa tutustua. Idea siitä, että jokaisella sovellukseen yhteyden ottavalla asiakaskoneella on kopio palvelimella määritellystä kokoelmasta ja kaikki kokoelmat on pidettävä synkronissa keskenään aina yhden tietokantaoperaation jälkeen saattaa kuulostaa hyvin raskaalta. Näin ei kuitenkaan ole, sillä pelkästään tietokantaan tapahtuvat muutokset suhteessa asiakaskoneiden MiniMongo-kantoihin lähtevät asiakkaiden koneelle ja tieto siirtyy erittäin kevyessä JSON-formaatissa.

Meteorista on tehty stressitesti halvimmille Digital Oceanista saataville, 512MB RAM-muistia omaaville palvelimille. Testissä tutkittiin, kuinka monta tilausta palvelin pystyi käsittelemään minuutin aikana vasteajan pysyessä alle 8 millisekunnissa. Yksi palvelin pystyi hoitamaan 2500 yhtäaikaista tilausta ja horisontaalisesti skaalattaessa turhaan laskentaan (overhead) kului alle 5 % laskentatehoa. (StackOverflow Meteor Scale 2015.)

Ottaen huomioon Meteorin tarjoamat sisäänrakennetut modernin verkkokehityksen vaatimat ominaisuudet, Meteor on mielestäni yllättävän hyvin skaalautuva. Meteor on myös vielä melko uusi sovelluskehys, joten optimointia on vielä paljon tulossa.

4.3 Tuottavuus

Vaikka Meteorin syntaksi ja ajattelumalli eroaa huomattavasti muista kehitysympäristöistä, ei sen oppimiseen silti mene muutamaa päivää kauempaa. Meteorissa on nähty huomattavasti vaivaa, jotta kaikesta on tehty mahdollisimman yksinkertaista. Hyvin vähällä määrällä koodia saa aikaan hyvin paljon asioita.

Deklaratiivinen syntaksi auttaa siihen, että kun palaa koodin ääreen muutaman päivän jälkeen, on kaiken ymmärtäminen uudelleen huomattavasti helpompaa. Koodi myös samalla dokumentoi itsensä, kun ohjelmoijan ilmaisukyky paranee huomattavasti.

Reaktiivisuuden sisäänrakentaminen Meteoriin antaa myös ohjelmoijalle automaattisesti erittäin vahvoja ominaisuuksia, joilla hyvän modernin verkkosovelluksen voi tuottaa ilman mitään lisävaivoja. Reaktiivisuuden ollessa normi Meteorissa, ei sitä varten myöskään tarvitse kirjoittaa turhaa koodia. Tämä lisää ylläpidettävyyttä ja samalla tuottavuutta, kun tietyn kohdan löytäminen koodin seasta on nopeampaa.

Meteorin ekosysteemi on erinomainen. Projektin aikana lähes aina löytyi hyvin dokumentoitu Meteor-paketti, joka ratkaisi tietyn ongelman. Paketin lataus ja asennus hoitui yhdellä komennolla terminaalissa ja paketin ominaisuuksien käyttö oli lähes aina erittäin helppoa. Ominaisuuksien toteuttaminen oli näinollen erittäin nopeaa. Huonona puolena tässä projektin aikanakin huomasin, että taaksepäin yhteensopivuuden rikkominen on aina mahdollista päivityksien myötä. Harvoin näin kuitenkin tapahtuu ja palaaminen tilaan ennen päivitystä on yhden komennon päässä.

Meteor on erittäin korkean abstraktiotason kehys, mikä piilottaa käyttäjältä erittäin tehokkaasti asioiden toteutuksien yksityiskohtia. Tämä on erittäin hyvä ominaisuus koodin ylläpidettävyyden ja selkeyden vuoksi, mutta jos Meteor on ensimmäinen ohjelmointikehys, jolla kehittäjä aloittaa, voi siirtyminen muihin ympäristöihin olla vaikeaa.

Henkilökohtaisesti itselläni tuottavuus nousi erittäin paljon Meteorin myötä. Ennen Meteoria minulla oli verkon puolelta jonkinverran kokemusta MySQL:stä, PHP:sta, JavaScriptistä ja Wordpressillä työskentelystä, mutta en oikeastaan ollut saanut valmiiksi yhtään suurempaa sovellusta. Meteorin voimalla sain kuitenkin tämän melko suuren projektin valmiiksi nopeasti ja kaikki tuntui erittäin helpolta suhteessa aiempiin kokemuksiin verkkokehityksestä. Samanlaisia kokemuksia löytyi myös paljon internetin keskustelupalstoilta.

4.4 Yhteenveto

Opinnäytetyön tavoitteena oli tutkia Meteorin soveltuvuutta verkkosovelluskehitykseen. Tätä varten Tampereen ammattikorkeakoulun sairaanhoitajien koulutusohjelman ohjatun harjoittelun digitalisointi toteutettiin Meteorilla ja samalla pohdittiin, miten kehitys nimenomaan tällä työkalulla sujui.

Meteor on kaikinpuolin erinomainen verkkosovelluskehys. Sen oppiminen oli hyvin nopeaa, vaikka se vaatikin suurehkoa ajattelumallin muutosta. Meteorilla omasta koodistani tuli paljon ylläpidettävämpää ja selkeämpää, sillä yksi kieli riitti asiakkaan koneella ja palvelimella, ja lähes jokaiseen ongelmaan löytyi Meteor-paketti, jonka avulla koodin määrä pysyi kurissa. Meteoria ei kuitenkaan aina kannata käyttää. Jos kyseessä on staattinen verkko*sivu*, ei Meteorin ominaisuuksista ole suurta hyötyä.

Digiharkka saatiin valmiiksi opinnäytetyön aikana ja se otettiin noin 30 henkilön pilottikäyttöön syksyllä 2015. Pilotin aikana ja sen jälkeenkin Digiharkkaa ylläpidetään ja jatkokehitetään, ja vuoden 2016 aikana sovellus on tarkoitus saada kaikkien harjoitteluun osallistuvien käyttöön. Asiakaspalaute on ollut enimmäkseen positiivista ja sovelluksen käyttäjiltä on tullut jatkokehitysideoita.

Jatkossa Digiharkan suhteen sovelluksen joitain ominaisuuksia kannattaisi entisestään parantaa ja tehdä vielä enemmän ”Meteormaiseksi”. Esimerkiksi palomuurin (Sikka) käyttöönotto, tietoturvallisuuden parantaminen entisestään ja viiveen kompensaation parempi hyödyntäminen voisivat olla hyviä toimenpiteitä. Muutamia käyttökokemusta parantavia ominaisuuksia en ehtinyt syksyn aikana tekemään, joten nekin voisi vielä toteuttaa.

Kirjoitushetkellä Floworksilla on jo alkanut toinen projekti, jossa sovelluksen chat-ominaisuutta parannellaan entisestään. Sen projektin avuksi opinnäytetyöni tuottaa arvokasta tietoa Meteorin toiminnasta. Floworksille opinnäytetyö tuottaa lisätietoa kehyksestä, jotta se voi paremmin arvioida kehyksen soveltuvuutta tulevaisuuden projekteihin.

LÄHTEET

Atmosphere Autoform. https://atmospherejs.com/aldeed/autoform. Luettu 31.10.2015

Atmosphere Spacebars. https://atmospherejs.com/meteor/spacebars. Luettu 22.10.2015.

Computermagazine. http://computemagazine.com/man-who-invented-world-wide-web-gives-new-definition/. Luettu 22.10.2015.

Dascalescu Why Meteor. http://wiki.dandascalescu.com/essays/why_meteor. Luettu 28.10.2015.

Meteor accounts. https://www.meteor.com/accounts. Luettu 29.10.2015.

Meteor Blaze. https://www.meteor.com/blaze. Luettu 22.10.2015.

Meteor DDP. https://atmospherejs.com/meteor/ddp. Luettu 25.10.2015.

Meteor docs. http://docs.meteor.com/#/full/livehtmltemplates. Luettu 22.10.2015.

Meteor Livequery. https://www.meteor.com/livequery. Luettu 27.10.2015.

Meteor mini-databases. https://www.meteor.com/mini-databases. Luettu 23.10.2015.

Meteor Tracker. https://www.meteor.com/tracker. Luettu 23.10.2015.

Meteorhacks DDP. https://meteorhacks.com/introduction-to-ddp. Luettu 26.10.2015.

Meteorhacks Sikka. https://meteorhacks.com/introducing-sikka-a-firewall-for-meteor-apps. Luettu 6.11.2015

Meteorpedia Why Meteor. http://meteorpedia.com/read/Why_Meteor. Luettu 16.10.2015.

MongoDB what is NoSQL. https://www.mongodb.com/nosql-explained?jmp=dotorg-form&_ga=1.1111217.1095380313.1446028113. Luettu 28.10.2015.

MongoDB scale. https://www.mongodb.com/mongodb-scale. Luettu 28.10.2015.

Quora why MongoDB. https://www.quora.com/Why-did-the-Meteor-team-pick-MongoDB-as-the-database-for-the-framework. Luettu 28.10.2015.

StackOverflow Meteor Scale. http://stackoverflow.com/questions/18083835/how-many-concurrent-users-can-a-web-app-built-in-meteor-js-handle/28653419#28653419. Luettu 19.11.2015

Tracker Manual. https://github.com/meteor/meteor/wiki/Tracker-Manual. Luettu 23.10.2015.

LIITTEET

Liite 1. Lyhenteet ja termit

1 (4)

Blaze	MDG:n kehittämä reaktiivinen DOM:n muokkauskirjasto.
Bootstrap	Twitterin työntekijöiden alulle panema front end –kehys, joka tarjoaa mm. valmiita määrityksiä käyttöliittymää varten.
CoffeeScript	Pieni ohjelmointikieli, joka käänetään JavaScriptiksi.
CSS	Cascading Style Sheets. Verkkosivun tyylejä kuvaava kieli.
DDP	Distributed Data Protocol. MDG:n kehittämä protokolla, joka määrittelee standardit nimet tietokannan kysely- ja muokkausoperaatioiden ja päivityksien synkronoinnin viesteille.
Deklaratiivinen	Ohjelmointityyli, jossa haluttu lopputulos kuvataan suoraan sellaisenaan, eikä kuvata ongelman ratkaisuun tarvittavia primitiivejä eksplisiittisesti.
Derby.js	Meteorin kaltainen sovelluskehys.
DOM	Document Object Model. Alusta- ja kieliriippumaton tapa kuvata ja käsitellä objekteja HTML, XHTML ja XML dokumenteissa.
Event	Tapahtumankäsittelijäfunktio, joka sidotaan tietyn elementin tiettyyn tapahtumaan.
Front End	Käyttäjälle näkyvä osa sovelluksesta.

Full Stack	Kaikki verkon ”kerrokset” asiakastietokoneen ruudulta palvelimelle ja tietokantaan.
Git	Hajautettu versionhallintajärjestelmä.
GitHub	Verkossa sijaitseva Git-projektien ylläpitopalvelu.
Helper	Funktio, joka palauttaa dataa, johon templaatista viitataan ja templaatissa käytetään.
HTML	HyperText Markup Language. Verkon standardi dokumenttien kuvauskieli.
HTTP	HyperText Transfer Protocol. Sovelluksen tasolla oleva protokolla, joka määrittelee viestien formaatin ja siirtotavan.
Imperatiivinen	Ohjelmointityyli, jossa ohjelmoija määrittelee lopputuloksen saamiseksi joukon ”käskeviä” komentoja (esim. lähde ulos -> nouse tuolilta, laita kengät jalkaan, laita takki päälle, avaa ovi).
Isomorfinen	Sama ohjelmointikieli sekä asiakaskoneella, että palvelimella.
Jade	HTML:n kaltainen templaattikieli.
JavaScript	Moniparadigmainen ohjelmointikieli, jota käytetään paljon verkkosovelluksissa.
jQuery	JavaScript kirjasto, joka helpottaa DOM:n manipulointia.
JSON	JavaScript Object Notation. Kevyt datan kuljetusformaatti.

LAMP	Linux, Apache, MySQL, PHP. Avoimen lähdekoodin teknologiakokoelma, jolla voidaan toteuttaa dynaamisia verkkosovelluksia.
Latency Compensation	Viiveen kompensointi. Tietokantakyselyiden onnistumisen simulointia asiakkaan koneella ilman odottelua tiedon saapumiselle kyselyn tilasta palvelimelta.
MDG	Meteor Development Group, Meteorin kehittäjäryhmä.
MEAN	MongoDB, Express.js, Angular.js, Node.js. Avoimen lähdekoodin JavaScript-teknologiakokoelma.
Meteor	Isomorfinen JavaScript-verkkosovelluskehys.
MiniMongo	Meteor-asiakkaan selaimen muistissa sijaitseva MongoDB-rajapinta, ts. MongoDB-emulaattori.
MongoDB	Dokumenttipohjainen NoSQL-tietokanta.
ORM	Object Relational Mapping. Abstraktiokerros, jossa esim. tietokannan data esitetään objekteina.
Protokolla	Ennalta sovitut säännöt viestien tulkitsemiseen.
Quora	StackOverflow:n kaltainen sivu muillekin kuin ohjelmoijille.
Reaktiivinen ohjelmointi	Ohjelmointityyli, jossa ilmaisun tulos muuttuu, kun ilmaisun muodostavat alkiot muuttuvat (esim. a = b + c. Kun b:lle tai c:lle sijoitetaan uusi arvo, muuttuu a:n arvo automaattisesti).
Remote Procedure Call	Palvelimella sijaitsevan funktion kutsuminen asiakaskoneelta.

Responsiivinen	Näytön kokoon mukautuva.
Skaalautuvuus	Systeemin mahdollisuus käsitellä kasvavaa määrää työtä.
SockJS	JavaScript-kirjasto, joka tarjoaa WebSocketin kaltaisen rajapinnan myös selaimille, jotka eivät tue WebSocketia.
SPA	Single Page Application. Verkkosovellus, joka lataa kaikki sivustolla navigoimiseen tarvittavat resurssit yhdellä kerralla palvelimelta.
Spacebars	Handlebarsiin pohjautuva Meteorin oma templaattikieli, jolla sidotaan sovelluslogiikka dokumentin kuvauskielen kanssa.
StackOverflow	Keskustelu- ja kysymys/vastaussivu ohjelmoijille.
URL	Uniform Resource Locator. Referenssi verkossa sijaitsevaan resurssiin, joka samalla ottaa kantaa resurssin yhteydenottotapaan.
WebSocket	Protokolla, joka mahdollistaa jatkuvan kaksisuuntaisen kommunikaation asiakaskoneen ja palvelimen välillä.