TAMPEREEN
AMMATTIKORKEAKOULU

MODERNIN JAVASCRIPT-POHJAISEN WEB-SOVELLUKSEN TOTEUTTAMINEN

Case: LUMATE Välineistöpankki

Esa Parkkila

Opinnäytetyö
Joulukuu 2016
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

PARKKILA ESA:
Modernin JavaScript-pohjaisen web-sovelluksen toteuttaminen
Case: LUMATE Välineistöpankki

Opinnäytetyö 68 sivua, joista liitteitä 3 sivua
Joulukuu 2016

Tässä opinnäytetyössä rakennettiin laboratoriovälineistöä oppilaitoksille lainaavalle LUMATE-keskukselle sähköinen lainanhallintajärjestelmä. Järjestelmä toteutettiin JavaScript-pohjaisena web-sovelluksena. Työssä käytettiin alan uusinta ja vielä osittain kokeellistakin tekniikkaa. Työn tuloksena syntynyt sovellus on aktiivisessa käytössä LUMATE-keskuksen asiakkailla ja henkilökunnalla.

Työ toteutettiin rakentamalla Meteor-sovellusalustan ja React-käyttöliittymäkirjaston päälle dynaaminen nk. yhden sivun sovellus. Sovelluksen palvelinosuus asetettiin toimimaan Docker-sovellussäiliön sisälle, jota puolestaan ajetaan pilvipalvelinyhtiö Linodelta vuokratun virtuaalikoneen sisällä. Käyttöliittymän responsiivisuuteen kiinnitettiin erityistä huomiota, ja sovelluksen asiakas- ja henkilökuntanäkymät toimivat sekä mobiiliettä työpöytäympäristössä. Kehitysympäristön työnkulkua pyrittiin tehostamaan esimerkiksi ottamalla käyttöön jatkuva integraatio ja automatisoimalla sovelluksen toimitus palvelimelle.

Tietolähteinä käytettiin pääosin internetissä saatavilla olevaa materiaalia. Web-teknologioiden nopean kehityksen vuoksi ajantasaisia kirjallisia teoksia oli heikosti saatavilla, ja valtaosa tiedosta kerättiin teknisestä dokumentaatiosta, tietotekniikka-aiheisista blogeista ja YouTubesta löytyneiltä konferenssivideoilta.

Asiasanat: web-kehitys, javascript, react, meteor

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
ICT Engineering
Software Engineering

PARKKILA ESA:
Developing a Modern JavaScript-based Web Application
Case: LUMATE Equipment Bank

Bachelor's thesis 68 pages, appendices 3 pages
December 2016

This thesis describes the development of a loan management system for Tampere's LUMATE Centre, an organization that lends laboratory equipment for educational institutions. The system was implemented as a JavaScript-based web application. Development was done using the latest, and partly experimental, technology. The produced application is in active use among LUMATE Centre's customers and staff.

The system was implemented by building a dynamic single page application on Meteor framework and React user interface library. The server side of the application was set to run inside Docker application container, which in turn runs on a virtual machine leased from a cloud server provider company called Linode. Special attention was paid to the responsivity of the user interface, for the application is used in both mobile and desktop environments by the customers and staff. The workflow of the development environment was enhanced by implementing practices like continuous integration and by automating the deployment of the application to the server.

Materials found in the internet were used as the main information sources The fast development of web technologies has made it difficult to find up-to-date printed sources, and most of the information was gathered from technical documentation, IT-blogs, and conference videos found on YouTube.

Key words: web development, javascript, react, meteor

SISÄLLYS

LYHENTEET JA TERMIT

Asiakaspää	Engl. front-end. Käyttäjän selaimessa toimiva web-sovelluksen osa. Synonyymejä: client, front-end, frontti.
Atomisuus	Tietokannoista puhuttaessa atominen tietokantaoperaatio tarkoittaa sitä, että operaatio joko suorittaa kaikki halutut toiminnot tai ei mitään niistä.
Botti	Engl. bot. Web-kehityksen ja tietoturvan kontekstissa usein pahantahtoinen ohjelma, joka esimerkiksi luo väärennettyjä profiileja sivustoille, levittää niiden avulla mainoksia ja etsii muiden käyttäjien sähköpostiosoitteita.
Blaze	Meteorin alkuperäinen käyttöliittymäkomponentti.
CRUD	Create Read Update Delete. Luo, lue, päivitä ja poista, tietokantojen neljä perustoimintoa.
DDP	Engl. Distributed Data Protocol, Meteor-sovelluskehyksen käyttämä kaksisuuntainen yhteysprotokolla asiakas- ja palvelinpään välillä.
Deklaratiivinen	Deklaratiiviset ohjelmakoodilauseet kertovat mikä on haluttu lopputulos, mutta eivät sitä, miten lopputulokseen päästään.
ECMAScript	Ecma International -organisaation kehittämä JavaScript-standardi.
Full-stack	Asiakaspään ja palvelinpään muodostama kokonaisuus.
Floodaus	Engl. flooding. Palvelunestohyökkäyksen tyyppi, jossa palvelua kuormitetaan kutsumalla sen rajapinnan toimintoja useita kertoja hyvin lyhyessä ajassa.
Hybridisovellus	Web-sovellus, jolla on yhteys mobiililaitteen natiiveihin ohjelmointirajapintoihin.
I/O	Engl. Input/Output. Ohjelmoinnissa termillä viitataan ohjelman ulkopuolisten resurssien, kuten tiedostojen tai verkkoyhteyksien, luku-, kirjoitus-, vastaanotto- ja lähetysoperaatioihin.
JavaScript	Web-ohjelmoinnissa suosittu ohjelmointikieli.

Linttaus	Engl. linting. Ohjelmakoodin tyylivirheiden etsiminen lintteri-työkalun avulla.
Modulaarinen	Löysästi pakattu, hajautettu järjestelmä joka koostuu yksittäisistä tietyn osa-alueen hyvin hoitavista komponenteista. Vastakohtana monoliittinen.
Mongo	MongoDB-tietokanta.
Monoliittinen	Tiukasti pakattu, yhtenäinen järjestelmä. Vastakohtana modulaarinen.
NoSQL	Lyhenne sanoista ”Not only SQL”, tarkoittaen perinteisistä SQL-relaatiotietokannoista poikkeavia tietokantoja, kuten MongoDB.
Npm	Engl. Node Package Manager. Paketinhallintajärjestelmä JavaScriptille.
Lavastus-ympäristö	Engl. staging environment. Palvelinympäristö, jolla suoritetaan laadunvalvontaa ja erilaisia testejä web-palvelun seuraavalle vielä julkaisemattomalle versiolle.
Palvelinpää	Engl. back-end. Verkkopalvelun palvelimella toimiva osa-alue.
Tuotantoympäristö	Engl. production environment. Palvelinympäristö, jolla varsinainen verkkopalvelun käyttö tapahtuu.
React	Facebookin kehittämä JavaScript-pohjainen käyttöliittymäkomponentti.
Refaktorointi	Ohjelmakoodin sisäisen rakenteen parantaminen ulkoisen toiminnallisuuden pysyessä muuttumattomana.
Repositorio	Engl. repository. Tietovarasto, tuttavallisemmin repo. Esimerkiksi Ubuntu-käyttöjärjestelmän pakettivarasto tai Git-versionhallintaohjelman tietovarasto.
Web-sivusto	Kokoelma perinteisiä web-sivuja, joiden esittämä sisältö on pääosin staattista. Esim. Wikipedia.
Web-sovellus	Web-selaimessa ajettava yhden sivun sovellus (SPA, Single-page application). Sisältö on pääosin dynaamista. Esim. Gmail.

1 JOHDANTO

Aikaisemmin valtaosa kuluttajien käyttöön suunnatuista tietokoneohjelmista tuotettiin PC-tietokoneille. Älypuhelinten ja tablettien yleistyminen on kuitenkin tuonut mukanaan monenlaiset käyttöjärjestelmät ja laiterajapinnat, jotka saattavat poiketa ohjelmointikielien ja muiden ominaisuuksien osalta merkittävästi. Tämä on aiheuttanut sen, että haluttaessa tavoittaa suuri yleisö on usein tarpeen tehdä samasta ohjelmasta useita versioita eri alustoille, minkä seurauksena ohjelman kehitykseen vaaditut resurssit voivat kasvaa merkittävästi.

Yksi mahdollinen ratkaisu tähän ongelmaan on toteuttaa ohjelma selaimessa ajettavana web-sovelluksena. Teknologian kehitys on tehnyt mahdolliseksi toteuttaa internet-selaimessa toimivia ohjelmia, jotka käyttäjäkokemukseltaan ja ominaisuuksiltaan ovat natiiviohjelmien tasoa. Nykytekniikka mahdollistaa jo pääsyn joihinkin mobiililaitteiden natiiveihin rajapintoihin web-sovelluksen sisältä.

Kun ohjelma tuotetaan web-sovelluksena, samaa ohjelmaa voidaan käyttää millä tahansa päätelaitteella, joka tukee modernia internet-selainta. Ohjelman julkaisu ja ylläpito helpottuvat, kun mobiililaitteiden sovelluskauppojen hitaat julkaisuprosessit jäävät pois. Lisäksi ohjelman käyttöönotto asiakkaan puolella on vaivatonta, kun tarvetta erillisille asennuksille ja päivityksille ei ole.

Web-sovelluksen suurimmat ongelmat ovat rajoitettu pääsy käyttöjärjestelmän natiiveihin ohjelmointirajapintoihin, sekä toimivan käyttöliittymän toteuttaminen eri kokoisille näyttölaitteille. Nämä ongelmat ovat kuitenkin hiljalleen poistumassa, kun teknologiat nk. hybridisovellusten ympärillä kehittyvät.

Tämän opinnäytetyön tarkoitus on toteuttaa toimiva lainanhallintajärjestelmä asiakkaan käyttöön. Tavoitteena on helpottaa asiakkaan toimintaa siirtämällä lainattavien tuotteiden kirjanpito pois kömpelöistä Excel-taulukoista ja muistioista helppokäyttöisen tietojärjestelmän sisälle. Tavoitteen saavuttamiseksi perehdytään JavaScript-kieleen ja sitä ympäröivään ekosysteemiin sekä web-kehityksessä pinnalle nousseisiin työkaluihin kuten esimerkiksi Docker ja jatkuva integraatio.

2 JAVASCRIPT

Tässä luvussa käydään läpi lyhyesti JavaScript-kielen historiaa, ominaisuuksia ja kieleen vahvasti liittyviä aihepiirejä kuten ECMAScript ja Node.js. JavaScript-ekosysteemi on vuosien saatossa kasvanut hyvin monimutkaiseksi, ja aloittelevan ohjelmoijan voi olla hyvin vaikea sisäistää mitä kaikkea kieli pitää sisällään. Luvun tarkoituksena on tarjota lukijalle pintaraapaisu kieleen liittyvien aihealueiden monimutkaisesta viidakosta.

2.1 Taustaa

JavaScript sai alkunsa vuonna 1995, kun Brendan Eich sai Netscapelta tehtävän luoda uusi ohjelmointikieli web-selainta varten. Kielen syntaksin tulisi muistuttaa Javaa, mutta olla samalla aloittelijaystävällinen ja kevyt. Eich sai 10 päivää aikaa toimivan prototyypin tuottamiseen, ja lopputuloksena syntyi JavaScript. (Severance 2012.)

JSON-tiedostomuodon luojan Douglas Crockfordin (2008, 3) mukaan JavaScriptin hyviin ominaisuuksiin lukeutuvat funktiot, löysä tyypitys, dynaamiset objektit ja ilmaisuvoimainen objektien literaalinotaatio. Isommissa ohjelmissa nämä mahdollistavat nopean prototyypityksen ja antavat ohjelmoijalle huomattavia vapauksia, mutta tuovat samalla vastuun noudattaa hyvää koodityyliä virheiden välttämiseksi ja koodin ymmärrettävyyden parantamiseksi.

JavaScriptin suurimmaksi ongelmaksi Crockford (2008, 3) mainitsee erityisesti globaaleihin muuttujiin perustuvan ohjelmointityylin, joka oli pitkään yleisin tapa toteuttaa JavaScript-kirjastoja virallisen moduulituen puuttumisen vuoksi. Ongelman helpottamiseksi on jo pitkään ollut olemassa erilaisia epävirallisia moduulijärjestelmiä, mutta vasta vuonna 2015 julkaistu ECMAScript 6 toi kieleen virallisen tuen moduuleille tehden globaalien muuttujien käyttämisestä lopullisesti tarpeetonta (Crockford 2008, 40; Ward 2014).

2000-luvun alussa ennen web-sovellusten yleistymistä web-selainten kehittäjät eivät juurikaan olleet kiinnostuneita JavaScriptin suorituskyvyn parantamisesta. Tilanteeseen tuli muutos vuonna 2008 Googlen julkaistua Chrome-selaimensa, jonka sisältämä V8-nimellä

lanseerattu JavaScript-moottori oli rakennettu alusta alkaen erityisesti raskasta JavaScript-pohjaista sovelluskäyttöä ajatellen. Googlen insinöörien arvioiden mukaan JavaScriptin suorituskyky kasvoi yli 100 kertaiseksi vuosien 2006 – 2013 välisenä aikana. Vuonna 2012 suoritetussa testissä havaittiin, että tietyllä osa-alueella optimoitu JavaScript-koodi suoriutui laskentatehtävästä vain 17% hitaammin kuin vastaava optimoitu C++-koodi. (Clifford 2012; Lund & Bak 2013.)

Vaikka JavaScript on ollut pitkään ohjelmistokehittäjien keskuudessa väheksytty ohjelmointikieli, on sen arvo ohjelmistokehityksessä kasvanut valtavasti web-teknologioiden noustua hallitsevaan asemaan kevyiden ohjelmistojen tuottamisessa. JavaScript on edelleen web-selaimen ainoa natiivi ohjelmointikieli, ja Node.js-alusta on nostanut JavaScriptin varteenotettavaksi kieleksi myös palvelimella. JavaScript-pohjaiset teknologiat kuten Cordova ja React Native puolestaan mahdollistavat natiivinkaltaisten sovellusten tuottamisen eri mobiilialustoille vain yhdellä lähdekoodilla.

2.2 ECMAScript

ECMAScript (ES) on Ecma International -organisaation kehittämä ja ylläpitämä standardi JavaScriptille. Kaikki nykyaikaiset JavaScript-moottorit pohjautuvat ES-standardiin, minkä ansiosta JavaScriptiä voidaan ajaa eri alustoilla ja web-selaimilla ilman pelkoa yhteensopivuusongelmista. (Mozilla Developer Network n.d.)

Standardi eli 2000 luvun alussa pitkään hiljaiseloa, kunnes vuonna 2009 julkaistiin sen viides editio (ES5), joka toi syntaksiin suuren määrän uusia ominaisuuksia. Seuraava iso julkaisu tapahtui vuonna 2015, kun ES6 julkaistiin, ja samalla päätettiin siirtyä noudattamaan vuosittaista julkaisusykliä. Vuodesta 2015 alkaen uusi ES-standardin versio julkaistaan joka vuosi kesäkuun aikana (Branscombe 2016).

Mikään nykyinen web-selain ei täysin tue uusimpien ES julkaisujen kaikkia ominaisuuksia. On kuitenkin olemassa työkaluja, joiden avulla lähdekoodin voi muuttaa aikaisemman standardin kanssa yhteensopivaksi. Tällöin on mahdollista hyödyntää jopa vielä julkaisemattomia, tuleville ES-julkaisuille suunnitteilla olevia ominaisuuksia. Yksi tällainen työkalu on nimeltään Babel, joka kääntää lähdekoodin ES5-yhteensopivaksi. ES5-tuki kattaa kaikki yleisessä käytössä olevat modernit selaimet. (Rauschmayer 2016.)

Tässä työssä hyödynnetään laajasti ES6:n ominaisuuksia (mm. luokat, moduulit, nuolija generaattorifunktiot), minkä lisäksi käytössä on muutama vielä julkaisematon ES- ominaisuus. Kaikki kehitysvaiheessa olevat ominaisuudet eivät välttämättä koskaan päädy viralliseen standardiin saakka. Tästä syystä projektin ylläpitovaiheessa on tarpeen pitää silmällä standardin kehittymistä siltä varalta, että jokin käytössä oleva kehitysvaiheen ominaisuus hylätään.

2.3 Node.js

Node.js on palvelimella toimiva ajoympäristö JavaScriptille. Se sai alkunsa vuonna 2009 kun web-kehityksestä kiinnostunut matematiikan opiskelija Ryan Dahl huomasi, miten hankalaa Ruby-kielellä on luoda tehokas palvelinohjelma web-sovellusta varten. Hän ymmärsi JavaScript-kielen sopivan hyvin tarkoituksiinsa, ja alkoi kehittää JavaScript-ajoympäristöä Linuxille. (Dahl 2011.)

Yksinkertaisen web-palvelimen luominen esimerkiksi Rubyllä, Pythonilla tai muilla perinteisillä palvelinpään kielillä on suhteellisen helppoa. Ongelmaksi muodostuu, kun halutaan palvelimen kykenemään käsittelemään useita yhtäaikaisia yhteyksiä, jolloin palvelinohjelma täytyy säikeistää tai muuten hitaat operaatiot kuten tiedoston lukeminen pysäyttävät pääsäikeen jokaisen yhteyden ajaksi. Säikeet tuovat palvelimen suoritukseen ylimääräistä kuormaa, ja liian monen yhtäaikaisen yhteyden avaaminen saattaa aiheuttaa huomattavaa suorituskyvyn heikkenemistä. (Dahl 2009.)

Tapahtumasilmukka (event loop) on esimerkiksi käyttöliittymäohjelmoinnissa käytetty tekniikka, mikä mahdollistaa takaisinkutsufunktioiden (callback functions) kutsumisen aina tapahtuman, kuten näppäinpainalluksen, yhteydessä. Takaisinkutsufunktio rekisteröidään tapahtumasilmukalle, joka suorittaa sen aina havaitessaan tapahtuman. Tällöin tapahtuman odottaminen ei pysäytä ohjelman suoritusta, ja sitä voidaan ajaa yhden säikeen sisällä. Selaimessa ajettavan JavaScriptin moottorit ovat perinteisesti hyödyntäneet tapahtumasilmukkaa. (Roberts 2014.)

Node.js rakennettiin alusta asti hyödyntämään tapahtumasilmukkaa I/O-operaatioiden suoritukseen, mikä tekee siitä erittäin suorituskykyisen I/O-intensiivisissä sovellutuksissa

kuten web-palvelimet. Node.js kykenee palvelemaan tuhansia yhtäaikaisia yhteyksiä ilman suorituskyvyn heikkenemistä. (Dahl 2009.)

3 TYÖKALUT JA TEKNIIKAT

Tässä luvussa käydään läpi opinnäytetyössä käytettyjä työkaluja ja tekniikoita. Luvun tarkoitus on tarjota lukijalle tietoa web-kehityksessä hyödyllisistä työkaluista, sekä pohjustaa luvussa 4 esiteltävän sovelluksen toteutuksessa käytettyjä tekniikoita. Luvussa käydään läpi vain erityisen tärkeiksi tai mielenkiintoisiksi koettuja nimikkeitä. Esimerkiksi JavaScript-kirjastoja oli projektin aikana käytössä yli sata kappaletta, joten niiden kaikkien läpi käyminen ei ole opinnäytetyön skaalan kannalta ajateltuna järkevää.

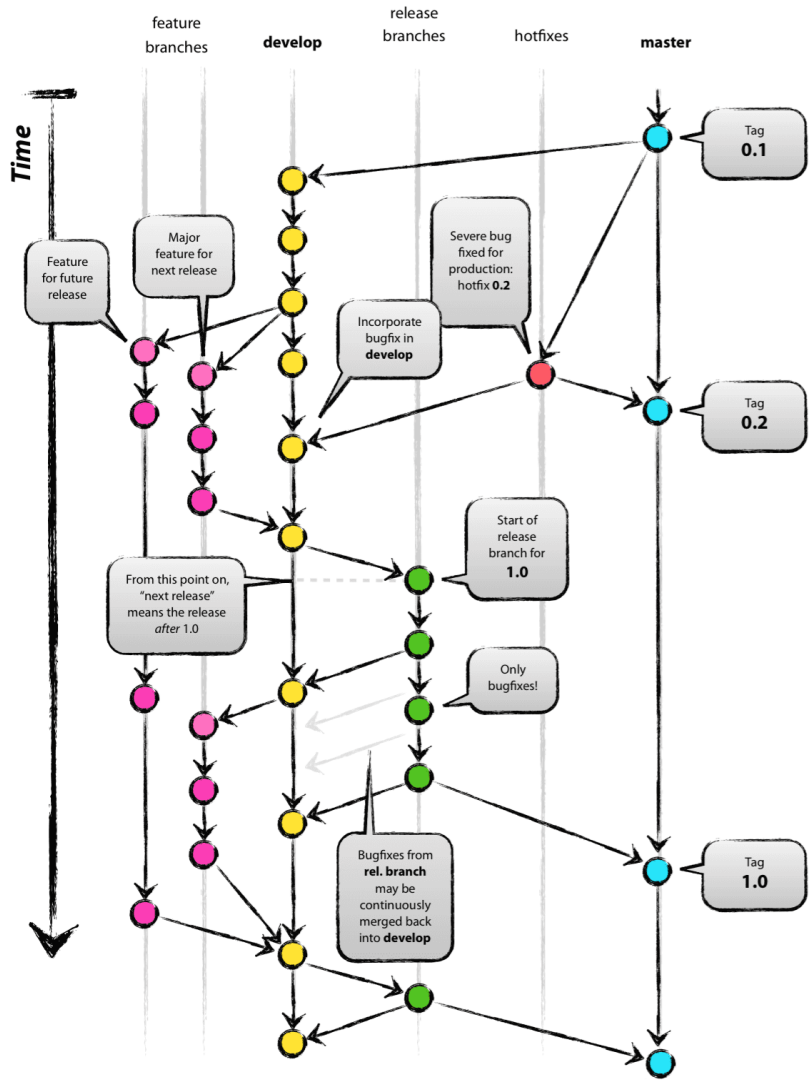
3.1 Ohjelmointiympäristö

Projektin ohjelmointiympäristö asennettiin Linuxille, jonka komentorivityökalut ja komentorivin skriptattavuus ovat eduksi web-kehityksessä. Seuraavaksi esitellään joitain ohjelmointiympäristössä hyödylliseksi osoittautuneita työkaluja.

3.1.1 Git

Git on Linus Torvaldsin kehittämä versionhallintaohjelmisto. Git sai alkunsa vuonna 2005, kun Linux-yhteisön suhteet Linux-ytimen silloisen versionhallintaohjelmiston BitKeeperin omistavan yhtiön kanssa murenivat. Torvalds kehitti oman järjestelmänsä ja nimesi sen Gitiksi, jonka tärkeimmät ominaisuudet ovat nopeus, luotettavuus ja hajautettu työskentelymalli. (Chacon & Straub 2014, 31.)

Git flow on Vincent Driessenin kehittämä versionhallintamalli Gitille. Malli määrittelee projektissa käytettävät Git-haarat ja niiden keskinäiset suhteet (kuvio 1). Se sisältää kaksi pysyvää päähaaraa ohjelman kehitysversiolle (develop) sekä vakaalle versiolle (master). Niiden lisäksi on myös erityyppisiä väliaikaisia haaroja. Malli mahdollistaa kehittäjäryhmälle tehokkaan yhtäaikaisen työskentelyn sovelluksen eri osissa. (Driessen 2010.)



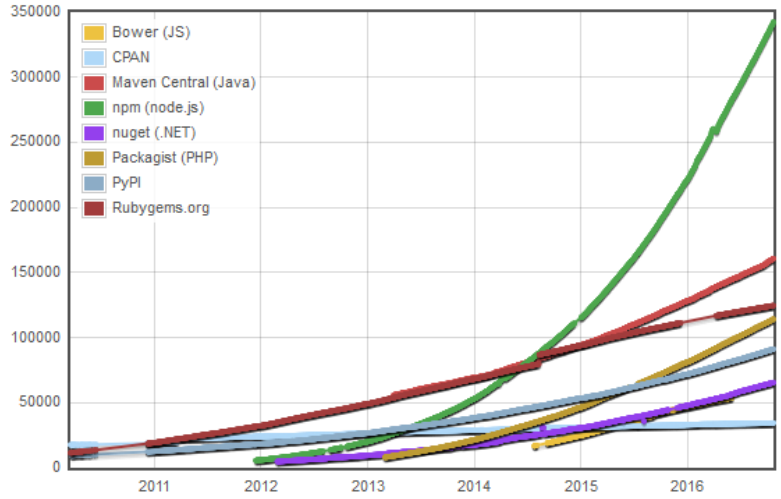
KUVIO 1. Git flow -malli (Driessen 2010)

Mallin rakenteen noudattamisen helpottamiseksi Gitin rinnalle asennettiin komentorivityökalu Git-flow (AVH Edition) (liite 1). Työkalu abstrahoi haarojen yhdistämisoperaatiot (merge operations) yksinkertaisen käyttöliittymän taakse, mikä yksinkertaistaa työnkulkua ja auttaa pitämään Git-historian rakenteen yhtenäisenä.

3.1.2 Npm

Npm (Node Package Manager) on JavaScript-paketinhallintaohjelmisto. Nimestään huolimatta npm ei ole suunnattu pelkästään Node.js-paketteja varten, vaan sitä käytetään yleisesti myös selainpään JavaScript-kirjastojen asennukseen (Voss 2014). Viime vuosina sen suosio on lähtenyt voimakkaaseen kasvuun ja nykyään sen repositorio sisältää jo yli

kaksi kertaa enemmän paketteja kuin seuraavaksi suurin Java-kirjastoille tarkoitettu Maven Central -repositorio (kuvio 2).

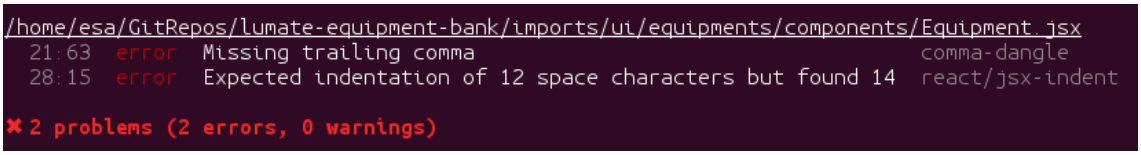


KUVIO 2. Paketinhallintaohjelmien pakettien määrän kehitys (Modulecounts 2016)

Npm tallentaa projektin riippuvuudet package.json-nimiseen tiedostoon. Kun projektille asennetaan uusi kehitysympäristö, kaikki sen riippuvuudet voidaan asentaa samalla kertaa komennolla npm install. Tiedosto säilyttää riippuvuuksien versiotiedot, eikä npm asenna taaksepäin epäyhteensopivia päivityksiä päivityskomentoa ajattaessa. Ajoittain pakettien julkaisijat kuitenkin epähuomiossa julkaisevat taaksepäin epäyhteensopivan päivityksen ja jättävät sen mainitsematta, jolloin pahimmassa tapauksessa laadunvalvonnan ohi pääsevä muutos voi rikkoa tuotantoympäristön. Tämän vuoksi npm update –päivityskomennon kanssa tulee olla aina erityisen huolellinen, ja varmistaa kaikkien päivitettyjen pakettien toimivuus myös päivityksen jälkeen.

3.1.3 ESLint

ESLint on JavaScriptille tarkoitettu ohjelmakoodin tarkistusohjelma (linting utility), joka vertaa ohjelmakoodin muotoilua asetuksissa määriteltyihin sääntöihin ja ilmoittaa tarkistuksessa esiintyneistä poikkeuksista. Tämä helpottaa tyylioppaiden seurantaa sekä mahdollistaa luettavamman ja virhevapaamman ohjelmakoodin. Kuvassa 1 on esitelty tarkistuksessa ilmenneitä tyylivirheitä. (ESLint n.d.)



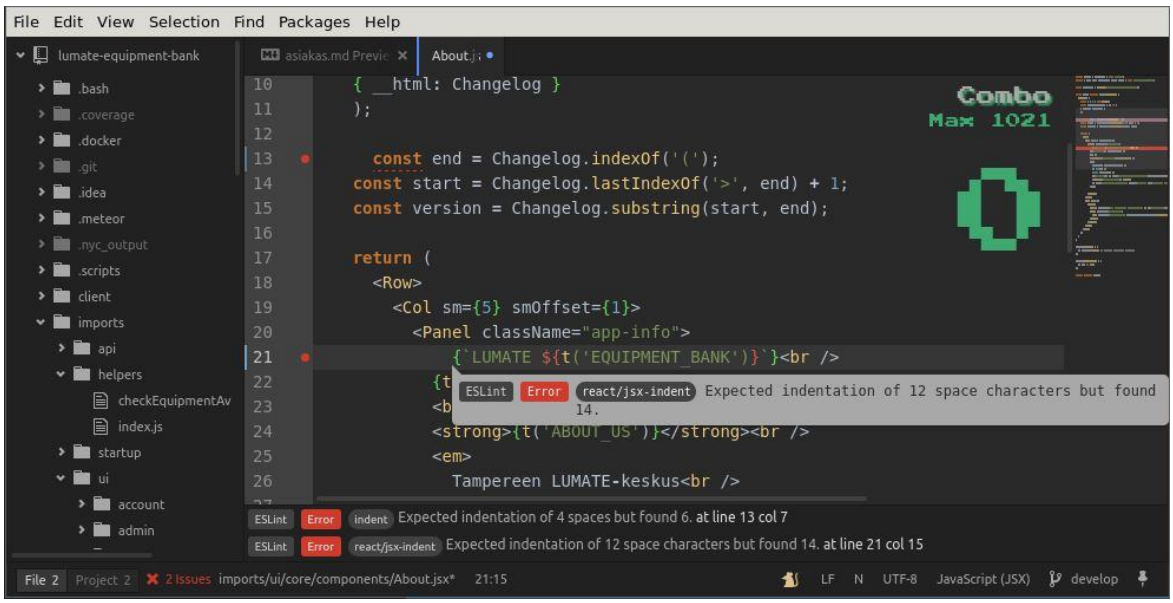
KUVA 1. ESLint-tarkistuksessa ilmenneitä tyylivirheitä

Projektissa otettiin käyttöön AirBnB-tyyliopas, jolle löytyy valmis ESLint-lisäosa. Pieni joukko lisäosan asettamia sääntöjä koettiin liian tiukoiksi, joten ne kytkettiin pois päältä ESLint-asetustiedostossa.

3.1.4 Atom

Atom on moderni avoimen lähdekoodin tekstieditori, joka on saatavilla ilmaiseksi Windowsille, Macille ja Linuxille. Se on toteutukseltaan Chromium-selaimen päälle rakennettu JavaScript-sovellus, minkä ansiosta sille on erittäin helppo luoda lisäosia ja tyylejä web-tekniikoiden avulla. (Atom n.d.)

Atomin suurin etu on sen monipuolinen lisäosavalikoima ja aktiivinen kehittäjäyhteisö. Kuvassa 2 on esitelty sen käyttöliittymä. Näkyvillä useita lisäosien tuomia ominaisuuksia, kuten oikean reunan minimap-lisäosan tarjoama tiivistetty näkymä nykyisestä tiedostosta sekä ESLint-lisäosan tuottamia tyylivirheiden ilmoituksia. Lisäksi näkyvillä on ”activate-power-mode”-lisäosan kombomittari, joka tuo ohjelmointiin kevyttä pelillistä tunnelmaa.



KUVA 2. Atom-tekstieditorin käyttöliittymä

3.2 Ajoympäristö

Projektin ajoympäristö asennettiin Saksassa sijaitsevalle pilvipalvelimelle, jonka tarjoaa pilvipalvelinyhtiö Linode. Palvelimelle asennettiin käyttöjärjestelmäksi Ubuntu 16.04 LTS, jonka käyttöjärjestelmätuki ulottuu vuodelle 2021 asti. Seuraavaksi käydään läpi ajoympäristölle asennettuja ohjelmistoja.

3.2.1 Docker

Docker on työkalu sovellussäiliöiden eli konttien (containers) hallintaan. Kontti muistuttaa periaatteeltaan virtuaalikonetta olemalla eristetty ajoympäristö eli hiekkalaatikko. Toisin kuin virtuaalikone, se ei kuitenkaan emuloi tietokoneen toimintoja laitteistotasolla, vaan kytkeytyy suoraan emokäyttöjärjestelmän käyttöjärjestelmäytimeen (kernel) aiheuttaen erittäin vähän ylimääräistä resurssien käyttöä (overhead). (Felter, Ferreira, Rajamony & Rubio 2014.)

Docker-compose on aputyökalu, joka helpottaa sellaisten kokoonpanojen ajamista, mitkä vaativat useamman kuin yhden kontin toimiakseen. Sen asetustiedostossa määritetyt kontit jakavat saman sisäisen verkon, joka mahdollistaa esimerkiksi sovelluskontin ja tietokantakontin välisen yhteyden. (Goasguen 2016.)

Projektia varten kehitysympäristölle ja palvelimelle asennettiin Ubuntun paketinhallinnasta paketit docker-engine ja docker-compose. Palvelinympäristössä Linoden Ubuntu 16.04 -käyttöjärjestelmällä Dockerin ohjeiden mukaisesti tehty asennus jäi jumiin. Ongelma ratkesi asentamalla dmsetup-paketti ennen docker-engine-pakettia (Brentjanderson 2016).

Koska Linode-pilvipalvelimen muokattu käyttöjärjestelmäydin ei tue AUFS-tiedostojärjestelmää, mikä on yleensä Dockerin oletusasetus, Docker ottaa oletuksena käyttöön joidenkin lähteiden mukaan epäluotettavan devicemapper-tiedostojärjestelmän (Mathus 2015). Sen tilalle vaihdettiin overlay2-tiedostojärjestemä, jonka ennustetaan tulevaisuudessa korvaavan AUFS-tiedostojärjestelmän Dockerin oletustiedostojärjestelmänä (Estesp 2016). Tiedostojärjestelmä saatiin käyttöön luomalla sijaintiin /etc/docker/daemon.json kuvan 3 mukainen asetustiedosto.

```
{
    "storage-driver": "overlay2"
}
```

KUVA 3. Daemon.json-asetustiedoston sisältö

3.2.2 MongoDB

MongoDB (jatkossa Mongo) on dokumentteihin pohjautuva NoSQL (Not only SQL) -tietokanta. NoSQL-tietokannat poikkeavat perinteisistä relaatiotietokannoista kuten MySQL. Ne tarjoavat suorituskykyetuja etenkin Big Data -tyyppisissä skenaarioissa, joissa dataa on paljon ja lukuoperaatiot muodostavat valtaosan tietokannan käytöstä. Lisäksi NoSQL-tietokannat mahdollistavat löysempien rajoitustensa vuoksi käyttötapoja, jotka eivät olisi mahdollisia perinteisissä relaatiotietokannoissa. (Rapa 2016.)

Mongossa data tallennetaan kokoelmiin, jotka sisältävät dokumentteja. Dokumentit koostuvat JSON-tyyppisistä avain-arvo-pareista. Dokumenttien sisältöä ei oletuksena rajoiteta millään tavoin, vaan jokainen dokumentti voi sisältää erityyppistä dataa, mukaan luettuna muita dokumentteja. (MongoDB n.d.)

Mongossa riippuvuuksia voidaan mallintaa pääsääntöisesti kahdella eri tavalla. Ensimmäinen tapa on käyttää alidokumentteja, esimerkiksi lisäämällä kerrostalo-dokumentin

sisälle taulukko, joka sisältää asunto-dokumentteja. Tämä on yleensä paras ratkaisu edellä kuvatun kaltaisissa yhden suhde moneen -tyyppisissä riippuvuuksissa, joissa alidokumenttien määrän voidaan olettaa olevan rajallinen. Alidokumenttien noutaminen kannasta on nopeaa, koska tarvitaan vain yksi kysely. Lisäksi kirjoitusoperaatiot saman dokumentin sisällä ovat atomisia, eikä alidokumenttien kanssa ole vaaraa siitä, että useita kohteita käsittelevä kirjoitusoperaatio jättäisi keskeytyessään tietokannan rikkinäiseen tilaan. (MongoDB n.d.)

Tilanteissa joissa alidokumenttien määrä voi kasvaa rajatta, kuten esimerkiksi viestiketju ja sen sisältämät viestit, alidokumenttien suuri määrä voi aiheuttaa ongelmia. Muistinkäytön rajoittamiseksi yksittäisen dokumentin koko on rajoitettu 16 megatavuun ja jokainen alidokumentti kasvattaa emodokumentin kokoa. 16 megatavua on kuitenkin riittävästi valtaosaan sovellutuksista. Todennäköisempi este alidokumenttien käytölle onkin tilanne, jossa on kyse monen suhde moneen -riippuvuudesta, tai jossa alidokumentit eivät saa olla riippuvaisia emodokumentin olemassaolosta. (MongoDB n.d.)

Toinen perinteisempi relaatiotietokannoista tuttu tapa mallintaa riippuvuuksia on käyttää apuna dokumenttien referenssejä eli vierasavaimia. Tämä on pakollista haluttaessa kuvata monen suhde moneen -tyyppisiä riippuvuuksia. Mongossa monen suhde moneen -riippuvuuden vierasavaimet voidaan säilyttää saman dokumentin sisällä taulukossa, jolloin tarvetta erilliselle avaintaululle ei ole. Näin on toimittu esimerkiksi tässä opinnäytetyössä käytetyssä tietokantamallissa. Projektissa käytetyt tietokantaratkaisut käydään läpi tarkemmin luvussa 4.5.2. (MongoDB n.d.)

3.3 JavaScript-kirjastot

Sovellusta varten asennettiin npm-paketinhallintaohjelmalla yhteensä noin sata kappaletta JavaScript-kirjastoja. Tässä alaluvussa esitellään niistä tärkeimmät ja hyödyllisimmät. Täydellinen lista projektissa käytetyistä kirjastoista on nähtävillä liitteessä 2.

3.3.1 Meteor

Meteor on avoimen lähdekoodin täyden pinon -sovellusalusta, joka julkaistiin tammikuussa 2012. Lähdekoodi on vapaasti saatavilla GitHub-palvelussa, ja kuka tahansa voi

halutessaan osallistua Meteorin kehitykseen, josta päävastuussa on Meteor Development Group (MDG) -niminen startup-yritys.

Meteorin erityispiirteisiin kuuluu JavaScriptin käyttö ainoana ohjelmointikielenä. Tämä on mahdollista myös palvelinpään koodissa. Aloittelijalla sen käyttö madaltaa kynnystä päästä tuottamaan web-sisältöä, sillä JavaScriptin lisäksi ei tarvitse käyttää aikaa palvelinpään kielien, kuten Pythonin tai Rubyn opiskeluun (Yu 2013).

DDP (Distributed Data Protocol) on MDG:n kehittämä protokolla, joka määrittelee WebSocket-protokollan päällä toimivan kaksisuuntaisen yhteyden. DDP-yhteyden avulla on mahdollista muodostaa reaaliaikainen yhteys asiakas- ja palvelinpään välille, mikä soveltuu erityisen hyvin esimerkiksi reaktiivisen datayhteyden ylläpitoon. DDP-protokollaa ei ole sidottu Meteor-sovellusalustaan, vaan sitä voidaan hyödyntää myös muilla alustoilla ja sovelluskehyksillä. (Ćwirko 2016.)

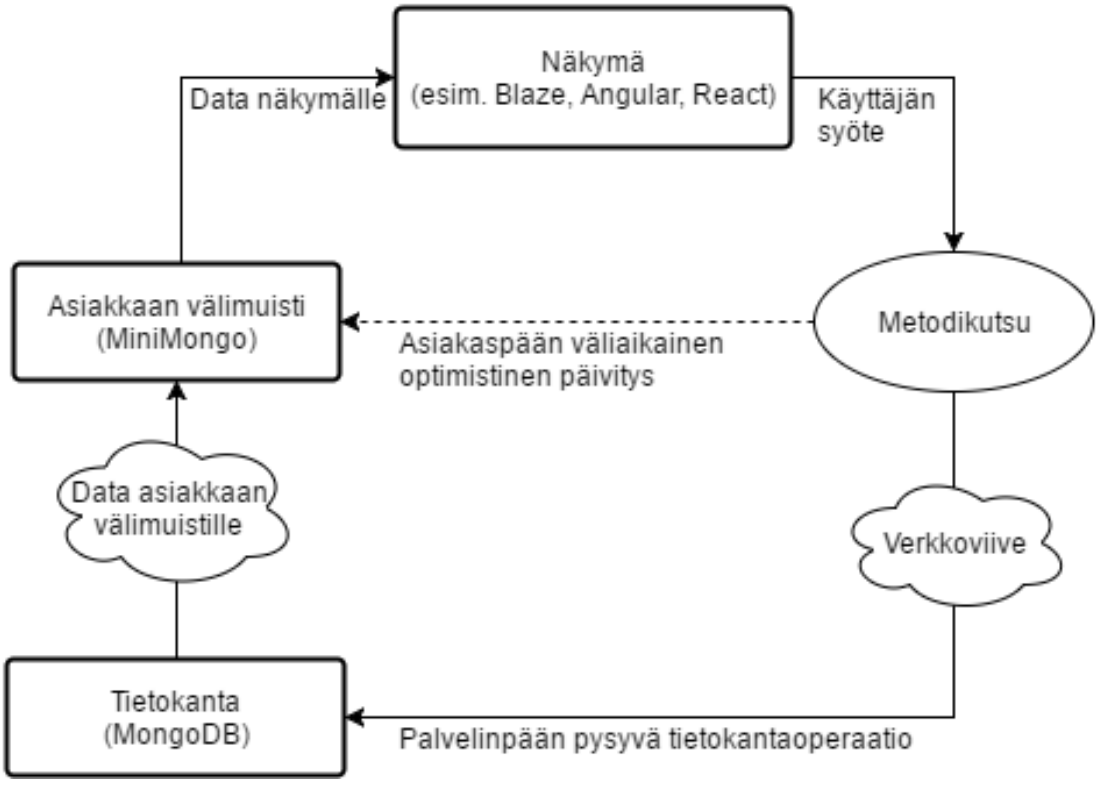
Meteorin ominaisuuksiin lukeutuvat isomorfisuus ja optimistiset päivitykset. Isomorfisuudella tarkoitetaan sitä, että samaa ohjelmakoodia voidaan ajaa sekä selaimen että palvelimen puolella. Tämä tulee kyseeseen esimerkiksi metodien kohdalla. Kuvassa 4 on esitelty isomorfinen metodi.

```
82  export const remove = new ValidatedMethod({
83    name: 'Methods.equipments.remove',
84    mixins: [CallPromiseMixin],
85    validate: new SimpleSchema({
86      targetId: { type: String },
87    }).validator(),
88    run({ targetId }) {
89      // Check that the user is authorized
90      if (Roles.userIsInRole(this.userId, ['admin']) === false) {
91        throw new Meteor.Error(403, 'Not authorized');
92      }
93
94      // Get the item
95      const equipment = Equipment.findOne({ _id: targetId });
96
97      // Delete from DB
98      equipment.remove();
99
100     // Update system log
101     logger.info({ user: this.userId }, `DELETE EQUIPMENT ${equipment._id} name:${equipment.name}`);
102   },
103 });
```

KUVA 4. Isomorfinen metodi

Periaatteena on, että asiakaspään data saattaa milloin hyvänsä olla vanhentunutta, eikä siihen siis voi luottaa. Asiakaspäässä ei ole myöskään välttämättä käytettävissä kaikkea metodin suorittamisen kannalta oleellista dataa. Voidaan kuitenkin optimistisesti ajatella, että asiakaspään data on valtaosan ajasta oikeellista ja riittävää. Tällöin on mahdollista tehdä käyttöliittymässä optimistinen päivitys, joka näkyy käyttäjälle käyttöliittymän välittömänä palautteena ja saa sovelluksen vaikuttamaan reaaliaikaiselta ilman internetin aikaansaamaa latenssia. (Stubailo 2015.)

Metodia kutsuttaessa sen koodi ajetaan ensin asiakkaan puolella selaimessa. Asiakaspään metodin aiheuttamat tietokantamuutokset päivittyvät MiniMongon paikalliseen tietokantaan, josta ne edelleen siirtyvät käyttöliittymä tilasäiliöön ja sitä kautta lopulta käyttäjän näkymään. Jos asiakaspään metodissa tapahtuu virhe, metodin suoritus päättyy ja asiakaspään tila palautuu entiselleen. Kuviossa 3 havainnollistetaan tapahtumien kulkua käyttäjän syötteestä näkymän päivittymiseen.



KUVIO 3. Meteorin optimistiset päivitykset (Stubailo 2015)

Mikäli asiakaspään metodi suoritetaan ilman ongelmia, kutsutaan seuraavaksi palvelinpään metodia DDP-yhteyden kautta. Palvelinpäässä metodi ajetaan samoilla argumenteillä kuin asiakaspäässä. Mikäli metodi ja sen data ovat keskenään identtiset, metodi tekee samat muutokset palvelimen MongoDB-tietokantaan ja asiakaspään MiniMongo-tietokantaan. Jos näin ei kuitenkaan ole, palvelinpään tekemät muutokset siirtyvät MongoDB-tietokannasta DDP-yhteyden välityksellä MiniMongoon joka päivittää käyttöliittymän tilan vastaamaan todellista lopputulosta.

Saman metodin sisällä voi olla myös ohjelmakoodia, joka ajetaan vain joko asiakas- tai palvelinpäässä. Tämä tulee tarpeeseen esimerkiksi silloin, kun osa metodin suorittamisen kannalta välttämättömästä datasta on saatavilla vain palvelinpäässä. Yksinkertaisin keino on rajoittaa ohjelmakoodin osan suoritus Meteor.isClient tai Meteor.isServer -ehtolauseella.

Meteorissa data välitetään palvelimelta asiakkaalle julkaisija-tilaaja-mallin avulla. DDP-yhteyden välityksellä asiakas pystyy tilaamaan dataa palvelinpään julkaisuilta. Julkaisut ovat reaktiivisia datalähteitä, eli palvelinpään datan muuttuessa muutokset lähetetään automaattisesti asiakkaalle. (Meteor n.d.)

Asiakkaan tilattua julkaisun kaikki sen sisältämä data lähetetään asiakkaalle ja tallennetaan paikalliseen MiniMongoksi kutsuttuun säiliöön. Julkaisun sisältämä data voi riippua tilaajan antamista parametreista ja tilaajan käyttöoikeuksista järjestelmässä. Julkaisun sisällä tilaajan käyttäjätietoihin päästään käsiksi this.userId-muuttujan avulla, jonka Meteor takaa kuuluvan sen hetkisen DDP-yhteyden välityksellä kirjautuneelle käyttäjälle. Käyttöoikeuksien validointiin ei tule ikinä käyttää asiakkaan puolelta lähetettyä id-tunnusta, koska tällöin kuka tahansa id-tunnuksen tietoonsa saanut voi käyttää sitä ja huijata olevansa kyseinen käyttäjä. (Meteor n.d.)

3.3.2 React

React on Facebookin kehittämä JavaScript-käyttöliittymäkirjasto. Reactin erityispiirteisiin kuuluu käyttöliittymän ohjelmointi puhtaasti JavaScript-kielen avulla. React tarjoaa Flux-arkkitehtuurin kanssa yksinkertaisen ja tehokkaan työkalun toteuttaa monimutkaisia datavirtoja sisältäviä käyttöliittymiä web-sovelluksille.

Reactin ydinominaisuuksiin kuuluu virtuaalinen dokumenttioliomalli (VDOM), joka on kevyt JavaScript-pohjainen abstraktio web-selaimen sisällä toimivasta dokumenttiobjektimallista (DOM). Perinteinen tapa muuttaa selaimen näkymää JavaScriptillä on tehdä suuri määrä yksittäisiä pieniä DOM-mutaatioita, jolloin selain joutuu vastaavasti tekemään paljon raskaita piirto-operaatioita pitääkseen näkymän ajan tasalla. Reactissa kaikki näkymän päivitykseen tarvittavat muutokset tehdään ensin JavaScriptin sisällä VDOM:iin. Kun jonkin yksittäisen komponentin sisäinen tila tai parametri muuttuu, muodostetaan uusi VDOM, jota verrataan vanhaan versioon. Tarvittavat DOM-muutokset voidaan tehdä yhdellä operaatiolla, mikä vähentää selaimen kuormaa. (Minnick 2016.)

React-käyttöliittymä muodostuu komponenteista, joiden rakentamiseen on olemassa kaksi eri vaihtoehtoa. Ensimmäinen ja vähemmän käytetty tapa on luoda komponentit perinteisellä JavaScript-syntaksilla createElement-kutsuja hyödyntäen. Suositellumpi tapa on käyttää Reactin tarjoamaa JSX-syntaksilaajennusta, joka lisää JavaScriptiin mahdollisuuden käyttää XML:n kaltaista merkintäkieltä. Se muistuttaa todella paljon perinteistä HTML-syntaksia. (React n.d.)

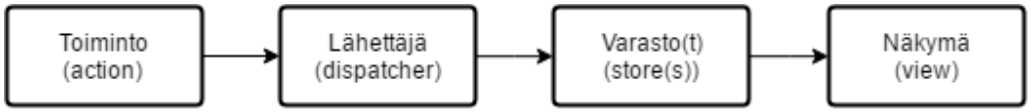
Reactin komponenttipohjaisen ajattelun ansiosta koodin uudelleenkäytettävyys paranee. Komponenteista kannattaa pyrkiä tekemään toisistaan riippumattomia, jotta niitä olisi helppo käyttää uudelleen sovelluksen eri osissa ja mahdollisesti jopa eri projektien välillä. Lisäksi komponenttien tarvitsema sovelluksen tila tulisi säilyttää mahdollisimman ylhäällä komponenttipuussa, ja siirtää se alaspäin proppien välityksellä. Näin komponentit pysyvät puhtaina ja helposti ymmärrettävinä. Jos komponentilla ei ole ollenkaan omaa sisäistä tilaa eikä sen tarvitse käyttää elämänkaarimetodeja (lifecycle methods), voidaan se esittää tilattomana funktionaalisena komponenttina (stateless functional component) ja React pystyy tekemään sille suorituskykyä parantavia optimointeja. (React n.d.)

Reactin ehkä suurin etu kilpailijoihinsa kuten Angular verrattuna on se, että kehittäjän ei tarvitse opiskella erillistä syntaksia voidakseen olla tuottava, vaan pelkkä JavaScriptin osaaminen riittää. Sen lisäksi React ei sisällä mitään ylimääräistä, vaan ainoastaan sen mikä on tarpeen toimivan käyttöliittymän rakentamiseksi. Tämä sopii hyvin yhteen Unix-filosofian ”tee yksi asia, ja tee se hyvin”-ajattelumallin kanssa, minkä ansiosta kehittäjä on vapaa valitsemaan haluamansa kirjaston sovelluksen muihin osiin.

3.3.3 Redux

Redux on Dan Abramovin vuonna 2015 kehittämä tilasäiliökirjasto JavaScript sovelluksille. Nimi Redux on yhdistelmä sanoista reduce ja Flux, joilla viitataan JavaScript-kielen Array.prototype.reduce-funktioon ja Facebookin Flux-sovellusarkkitehtuuriin. Redux sai alkunsa, kun Abramov pohti keinoja yksinkertaistaa Fluxin toteutusta. Hän sai idean funktionaalisen ohjelmoinnin kautta tutun reducer-funktion pohjalta: funktio, jolle syötetään argumentteina nykyinen tila ja muuttuja, ja joka palauttaa niiden perusteella uuden tilan. Ideasta syntyi Redux, jonka yksi peruskomponentti on funktio, joka ottaa argumenteikseen sovelluksen nykyisen tilan ja Flux-toiminnon, ja palauttaa sovelluksen uuden tilan. (Abramov 2015.)

Flux on Facebookin kehittämä sovellusarkkitehtuuri. Se pyrkii yksinkertaistamaan perinteistä MVC arkkitehtuuria pakottamalla yksisuuntaisen datavirran käyttäjän toiminnosta näkymään. Kuviossa 4 on esitelty Flux-arkkitehtuurin datavirtamalli. Käyttäjä tekee toiminnon (action), joka välitetään lähettäjälle (dispatcher). Lähettäjä kutsuu varaston (store) metodia, mikä päivittää varaston tilan. Uusi tila päivitetään näkymään (view). Varastoja voi olla yksi tai useampia. (Li 2016.)

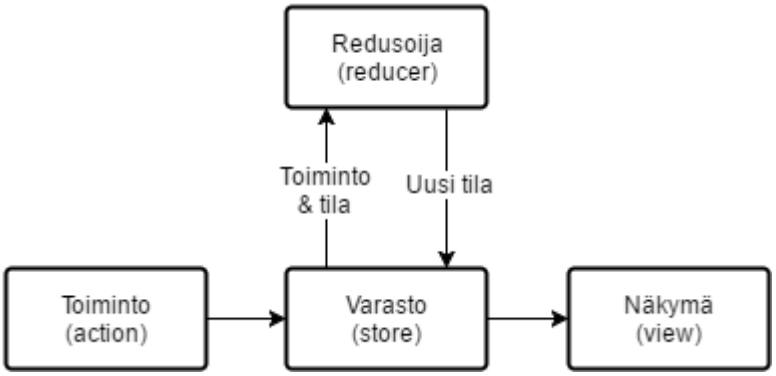


KUVIO 4. Alkuperäinen Flux-arkkitehtuurin datavirta

Redux on yksinkertaistettu, funktionaalinen toteutus Flux-sovellusarkkitehtuurista. Redux ei ole täysin puhdas Flux-toteutus, mutta pääperiaatteet ovat samat. Se säilyttää Reactin ja Fluxin yksisuuntaisen datavirran periaatteen, mutta Fluxista poiketen sisältää ainoastaan yhden varaston (single source of truth). (Redux n.d.)

Varaston tila on kirjoitussuojattu, mikä tarkoittaa, että varastoa ei ikinä muokata suoraan. Tarvittavat muutokset tehdään redusoijafunktiolla, joka palauttaa aina kokonaan uuden varaston. Funktio voidaan koota pienemmistä tietyn osa-alueen hallitsevista redusoijista, jolloin sovelluslogiikka saadaan jäsenneltyä toisistaan riippumattomiksi komponenteiksi, kuten perinteisessä Fluxissa käytettäessä useampaa varastoa. (Redux n.d.)

Kuviossa 5 on esitelty Reduxin muokattu datavirtamalli. Fluxista poiketen myöskään erillistä lähettäjä-komponenttia ei tarvita, vaan toiminnot lähetetään suoraan varasto-olion dispatch-metodilla. Toiminnot välitetään redusoijalle (reducer), joka sisältää sovelluksen käyttöliittymälogiikan. Varaston tila korvataan redusoijan palauttamalla uudella tilalla, joka päivitetään näkymään.



KUVIO 5. Redux-datavirta

Redusoija on puhdas funktio, joka palauttaa samoilla argumenteilla aina saman tuloksen, eikä aiheuta sivuvaikutuksia. Varaston tila pysyy helposti ennustettavana, koska sama tila on aina saatavilla syöttämällä varastolle identtiset toiminnot. Tämä mahdollistaa esimerkiksi Reduxin kehittäjätyökalujen aikamatkustustoiminnon, jonka avulla yksittäisiä jo tehtyjä toimintoja voi kytkeä pois päältä. (Redux n.d.)

3.3.4 Redux Saga

Reduxin toimintalogiikka on perusperiaatteeltaan synkroninen, ja erilaiset sivuvaikutukset kuten asynkroniset kutsut eivät istu sen malliin kovin hyvin. Sovelluksen käyttöliittymä tarvitsee kuitenkin asynkronisia kutsuja hakeakseen dataa palvelimilla sijaitsevilta tietolähteiltä. Asynkroniset kutsut toteutetaan oletusarvoisesti redux-thunk-kirjastolla, jota käytettäessä Reduxille lähetetyt viestit voivat sisältää funktioita, ja näihin funktioihin on mahdollista sisällyttää asynkronista toimintalogiikkaa (Redux n.d). Ratkaisu tuottaa kuitenkin vaikeasti ymmärrettävää koodia, jota on myös hankala testata.

Redux Saga on apukirjasto Reduxille, jonka tehtävä on helpottaa Reduxia käyttävän sovelluksen sivuvaikutusten hallintaa. Sen avulla sivuvaikutukset voidaan eristää muusta

toimintalogiikasta generaattorifunktioiden sisälle. Lisäksi se abstrahoi asynkroniset kutsut helposti testattavan deklaratiivisen ohjelmointirajapinnan sisälle. (Fairbank 2016.)

3.3.5 Later.js

Later.js-kirjaston avulla voidaan määrittää aikatauluja tietyn väliajoin toistuville tehtäville. Sillä voidaan toteuttaa helposti esimerkiksi päivittäisiä tai viikoittaisia tehtäviä, kuten sähköpostimuistutuksia tai tietokannan siivousoperaatioita.

Projektin vaatimuksiin sisältyi sähköpostimuistutuksen lähettäminen asiakkaalle lainan alkamista ja erääntymistä edeltävinä päivinä. Palvelinpään ohjelmakoodiin luotiin tiedosto cron.js, johon asetettiin muistutuksen lähettävät funktiot ajettavaksi joka päivä kello 9 aamulla suomen aikaa (kuva 5).

```
135 const testSchedule = later.parse.text('every 30 seconds');
136 const dailySchedule = later.parse.text('at 6:00am'); // Server is GMT so 9:00am in Finland
137 const weeklySchedule = later.parse.text('on the last day of the week');
138 later.date.localTime();
139
140 const testCron = Meteor.bindEnvironment(() => {
141   logger.debug('Running test cron operations..');
142 });
143
144 const dailyCron = Meteor.bindEnvironment(() => {
145   logger.info('Running daily cron operations..');
146   sendBeginningWarning();
147   sendExpirationWarning();
148 });
```

KUVA 5. Later.js-kirjaston avulla aikataulutettuja tehtäviä

3.4 GitLab

GitLab.com on GitLab Inc. -yhtiön ylläpitämä SaaS (Software as a Service) -pilvipalvelu, joka tarjoaa säilytystilaa ja hallintatyökaluja Git-repositorioille. Toisin kuin suurimmat kilpailijansa GitHub ja Atlassian Bitbucket, GitLab.com tarjoaa ilmaiseksi rajattoman määrän yksityisiä repositorioita, projektin osallistujia ja jatkuvan integraation (continuous integration, CI) ajamiseen tarvittavat työkalut. (Sijbrandij 2016a). GitLab pyrkii tarjoamaan kaikki ohjelmakoodin työnkulun kannalta tarpeelliset työkalut yhdessä paketissa (Sijbrandij 2016b).

Projektin alussa ohjelmakoodia säilytettiin Atlassian-yhtiön Bitbucket-palvelussa. Lokakuussa 2016 projektissa otettiin käyttöön Bitbucket Pipelines -toiminnallisuus, mikä mahdollisti jatkuvan integraation käyttöönoton. Sen ansiosta ohjelmakoodin testaus, rakennus ja toimitus palvelimelle saatiin automatisoitua kokonaan, jolloin integraatioprosessi ajetaan joka kerta kun kehittäjä lähettää uudet ohjelmakoodin muutokset Git-etäpalvelimelle. Pipelines oli kuitenkin käyttöönotettaessa vielä testausvaiheessa, ja se tulisi säilymään ilmaisena ainoastaan vuoden 2016 loppuun (Atlassian n.d).

Jatkuva integraatio osoittautui kuitenkin erittäin hyödylliseksi ja aikaa säästäväksi työkaluksi, eikä siitä haluttu luopua. Asiaa tutkittaessa kävi kuitenkin ilmi, että on olemassa kilpaileva palvelu GitLab.com, joka tarjoaa täysin ilmaisen jatkuvan integraation rajattomalle määrälle projekteja (Sijbrandij 2016a). Palvelun muutkin ominaisuudet vaikuttivat lupaavilta, minkä vuoksi päätös projektin siirtämisestä GitLabille oli helppo. GitLab tarjosi myös helppokäyttöisen import-työkalun, jonka avulla projekti saatiin kopioitua Bitbucketista muutamassa minuutissa.

GitLabin varjopuoleksi osoittautui GitLab.com-palvelun ajoittainen hidastuminen, sillä välillä selainkäyttöliittymän sivulataukset kestivät jopa 2-5 sekuntia. Myös ohjelmakoodimuutosten lähettäminen palvelimelle kesti ajoittain useamman sekunnin. GitLab kertoo kuitenkin tekevänsä töitä infrastruktuurinsa parantamiseksi, joten tilanteen parantuminen tulevaisuudessa on todennäköistä (Carranza 2016). Lisäksi palvelun suorituskyky on jo nyt riittävän hyvällä tasolla siihen nähden, että se on täysin ilmainen ja sisältää työnkulkua huomattavasti nopeuttavia ominaisuuksia.

4 VÄLINEISTÖPANKKI-SOVELLUS

Tässä luvussa tutustutaan opinnäytetyön käytännön osuutena toteutettuun LUMATE Välineistöpankki -lainanhallintajärjestelmään. Järjestelmä toteutettiin selaimessa ajettavana web-sovelluksena, ja se sisältää mm. täysin reaktiivisen käyttöliittymädatan ja mobiililaitteille skaalautuvan, responsiivisen käyttöliittymän.

4.1 Alkutilanne

Välineistöpankki-sovellus sai alkunsa, kun alkuvuodesta 2016 Tampereen yliopistojen alaisuudessa toimivalta LUMATE-keskukselta tuli Tampereen Ammattikorkeakoululle pyyntö toteuttaa internetissä toimiva lainausjärjestelmä-tyyppinen ohjelma organisaation ja heidän asiakkaidensa käyttöön. Sovelluksen kehittäjäksi valikoitui allekirjoittanut, ja projekti vaikutti erittäin hyvältä opinnäytetyön aiheeksi. Kokemusta web-kehittämisestä oli kertynyt kesätöiden kautta, ja ison projektin toteuttaminen yksin vaikutti realistiselta, joskin alusta asti oli selvää, että työtä on paljon ja aikaa tulee kulumaan runsaasti.

4.1.1 Vaatimukset ja määrittely

Projektin asiakkaan vaatimuksena oli toteuttaa lainanhallintajärjestelmä, jonka avulla asiakkaan ydintoiminta eli laboratoriovälineistön lainaaminen erilaisille kerhoille ja kouluille helpottuisi. Tekniset vaatimukset rajoittuivat lähinnä tuotteen käyttöön, jonka tulisi tapahtua pääasiassa web-selaimella. Käyttöliittymän tulisi toimia sekä PC-tietokoneilla että mobiililaitteilla, joista erikseen mainittiin vielä tabletti.

Varsinaisia määrittelydokumentteja ei projektissa käytetty, ja sovelluksen määritelmää tarkennettiin projektin edetessä. Koska lainausjärjestelmä on melko tyypillinen ohjelmistoprojekti, spesifisten määritelmädokumenttien puuttuminen ei ollut ylitsepääsemätön ongelma. Projektin kulku pyrittiin toteuttamaan ketterien menetelmien mukaisesti tekemällä tiivistä yhteistyötä asiakkaan kanssa, ja kysymällä tarkentavia kysymyksiä epäselvyyksien ilmaantuessa.

4.1.2 Tekniset valinnat

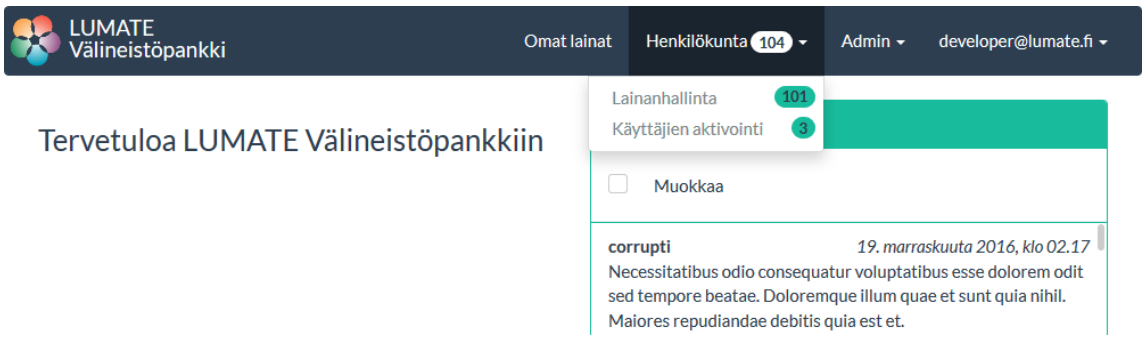
Projektin isot tekniset valinnat tehtiin jo heti projektin alussa. Ensimmäinen päätös oli toteuttaa sovellus Meteor-sovellusalustalla, josta oli jo aikaisempaa kokemusta pienemmän projektin parista. Meteorin uusi 1.3-versio oli juuri edennyt beta-testaukseen ja sen tuomat isot muutokset kuten npm-integraatio vaikuttivat mielenkiintoisilta. Lisäksi Meteoriin oli jo jonkin aikaa sitten tullut tuki React-käyttöliittymäkirjastolle, jonka opiskelu houkutteli erityisen paljon sen saaman suuren huomion takia. Sen lisäksi kyseiset tekniikat vaikuttivat hyvältä keinolta päästä oppimaan lisää JavaScript-kielestä ja sen ekosysteemistä.

Muut tekniset valinnat kietoutuivat Meteorin ja Reactin ympärille. Meteor toi mukanaan pakollisena MongoDB-tietokannan, joka ei ehkä muuten olisi ollut päävaihtoehto tämän tyyppiselle projektille. Meteorin uuden 1.3-version tuoma tuki npm-paketeille oli suuri parannus ja se toi Meteor-kehittäjän saataville paljon hyödyllisiä JavaScript-kirjastoja. Lisäksi React-yhteisön sisällä suureen suosioon noussut Redux-tilasäiliö ja sen erilaiset lisäosat toivat mukanaan uusia mielenkiintoisia tapoja toteuttaa käyttöliittymän tilanhallintaa.

4.2 Käyttöliittymä

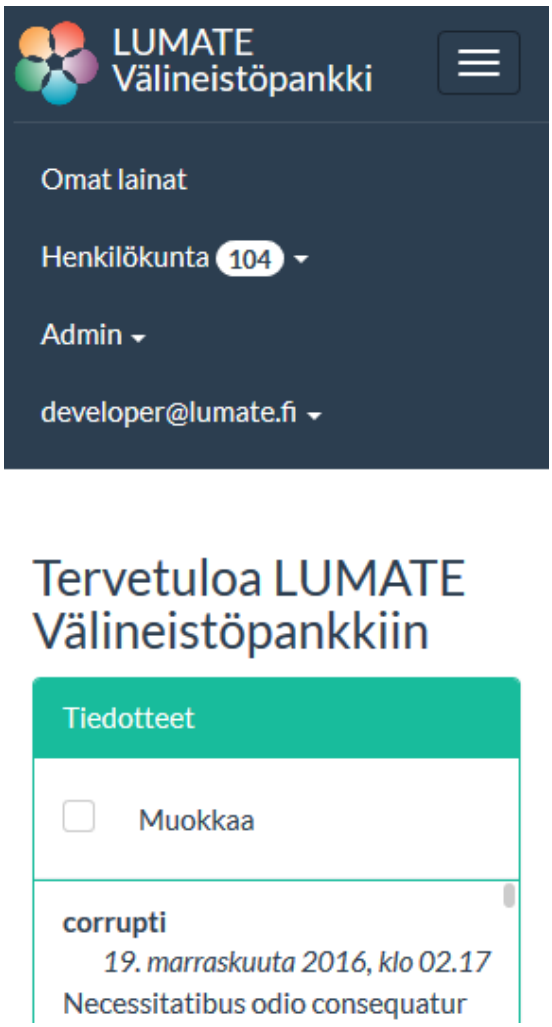
Sovelluksen käyttöliittymää rakentaessa tavoitteeksi otettiin helppokäyttöisyys ja toimivuus sekä työpöydällä että mobiililaitteilla. Koska sovelluksen on tarkoitus toimia etupäässä työkaluna, ei graafisten elementtien ulkonäön suunnitteluun haluttu käyttää ylimääräistä aikaa. Käyttöön otettiin suosittu Bootstrap-sovelluskehys, jonka päälle asennettiin ilmainen Bootswatch Flatly -teema. Näin saatiin aikaiseksi modernin litteä ulkoasu, joka skaalautuu automaattisesti mobiilinäytöille Bootstrapin grid-ominaisuuden ansiosta.

Kuvassa 6 on esitelty sovelluksen yläpalkki. Sovelluksen eri osien välillä navigointi tapahtuu sovelluksen yläreunassa sijaitsevan yläpalkin avulla. Yläpalkin linkit on ryhmitelty eri käyttäjätasojen mukaisesti.



KUVA 6. Sovelluksen yläpalkki

Mobiililaitteilla sovelluksen elementit mukautuvat Bootstrap-sovelluskehyksen ansiosta automaattisesti selaimen leveyden mukaan. Kuvassa 7 on esitelty käyttöliittymän mobiilitila, joka aktivoituu selainikkunan leveyden ollessa alle 768 pikseliä. Mobiilitilassa elementit kasautuvat päällekkäin, jolloin niistä saadaan riittävän isoja ja ne erottuvat selkeästi pieneltäkin näytöltä.



KUVA 7. Mobiilikäyttöliittymä

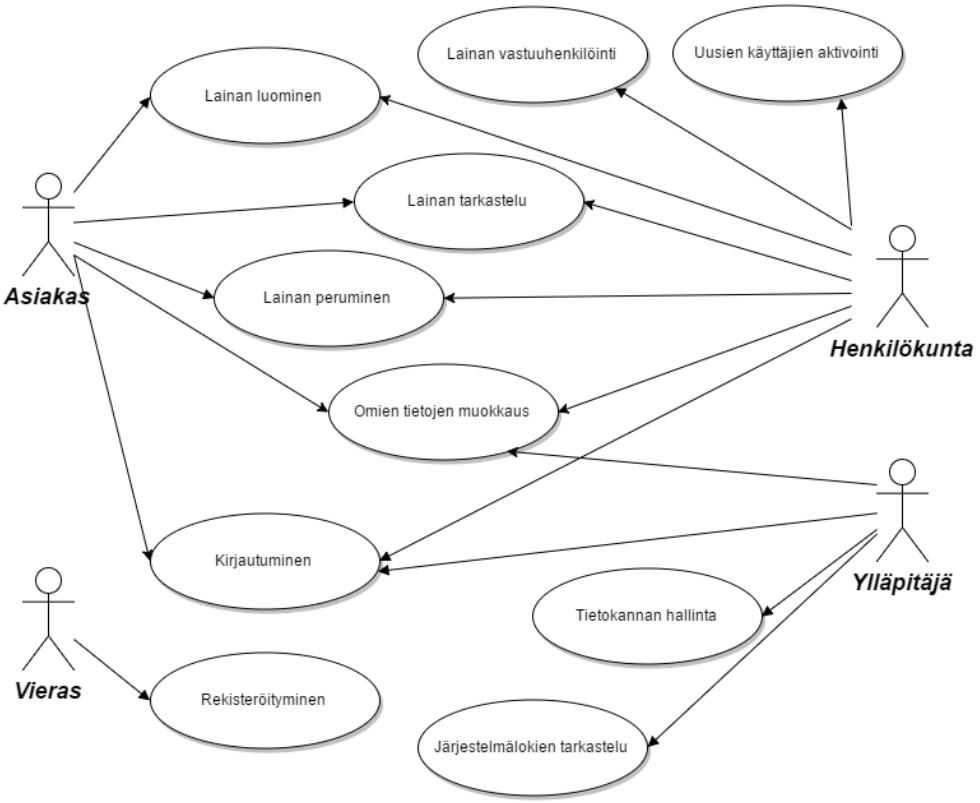
Käyttöliittymän kieleksi on valittavissa suomi ja englanti. Oletuskieli on suomi, mutta kieltä voi vaihtaa omat asetukset -näkymän kautta. Muutamat ylläpitäjän näkymät ovat saatavilla vain englanniksi. Monikielisyys on toteutettu i18next-kirjastolla. Kuvassa 8 on esitelty osa JSON-tyyppistä kielitiedostoa, joka sisältää käyttöliittymän tekstisisällön. Uuden kielen lisäämiseksi tarvitsee vain lisätä uusi JSON-tiedosto joka sisältää käännetyt tekstit.

```
90    "ROLES": "Roolit",
91    "SAVE": "Tallenna",
92    "SELECT": "Valitse",
93    "SEND": "Lähetä",
```

KUVA 8. Kielitiedostoa

4.3 Roolit

Palvelussa käyttäjät on jaettu neljään eri rooliin: vieras, asiakas, henkilökunta ja ylläpitäjä. Kuviossa 6 on esitelty palvelun käyttötapauskaavio, joka havainnollistaa roolien yhteyttä palvelun eri toimintoihin eli käyttötapauksiin.



KUVIO 6. Käyttötapauskaavio

Palvelun tärkein tehtävä on mahdollistaa tuotteiden lainaaminen asiakkaalle, ja toiminnallisuus rakentuu sen tehtävän ympärille. Henkilökunnan tehtävä on antaa tukea asiakkaalle toimimalla asiakkaan lainojen vastuuhenkilönä. Henkilökunnan täytyy myös kyetä luomaan lainoja organisaation sisäiseen käyttöön. Ylläpitäjän pääasiallinen tehtävä on tietokannan hallinta ja ongelmatilanteissa järjestelmälokien tutkiminen.

Järjestelmän sisällä käyttöoikeudet määräytyvät viiden eri käyttäjätason mukaan. Kuvion 6 käyttötapauskaaviossa esiteltyjen neljän roolin lisäksi järjestelmä tuntee uusi käyttäjä -roolin, jonka tarkoitus on toimia suojamuurina palveluun rekisteröityvien haitallisten käyttäjien varalta. Taulukossa 1 on esitelty käyttäjätasot ja niiden sallitut toiminnot.

Ylemmällä käyttäjätasolla on käytettävissään myös kaikki alempien tasojen toiminnallisuus.

TAULUKKO 1. Käyttäjätasot

Käyttäjätaso	Rooli	Sallitut toiminnot ja tehtävät
3	Ylläpitäjä	Koko järjestelmän ylläpito, mm. ylläpito, järjestelmäloki ja debug -näkymät, sekä tietokannan hallinta-operaatiot.
2	Henkilökunta	Uusien käyttäjätunnusten aktivoiminen. Lainojen manageröinti (merkitseminen asiakkaalle luovutetuksi, merkitseminen palautetuksi).
1	Asiakas	Omien lainojen luominen ja seuranta.
0	Uusi käyttäjä	Oman käyttäjätunnuksen asetusten muokkaus.
-1	Vieras	Järjestelmään rekisteröityminen ja kirjautuminen sekä unohtuneen salasanan palautus sähköpostin kautta.

Uusi käyttäjä kykenee ainoastaan muokkaamaan omia asetuksiaan. Henkilökunnan tehtäviin kuuluu aktivoida uuden käyttäjän tunnus asiakas-tasolle. Tätä ennen tulee varmistua uuden käyttäjän tietojen oikeellisuudesta. Tarkoituksena on näin estää botteja ja muita tunkeutujia pääsemästä käsiksi järjestelmän toimintoihin.

Roolit tallennetaan tietokantaan Meteorin paketinhallintajärjestelmällä asennettavan alanning:roles-lisäosan avulla. Lisäosa tarjoaa yksinkertaisen ja turvallisen tavan hallita sovelluksen käyttöoikeuksia ja sen asentamista suositellaan Meteorin oppaassa (Meteor n.d).

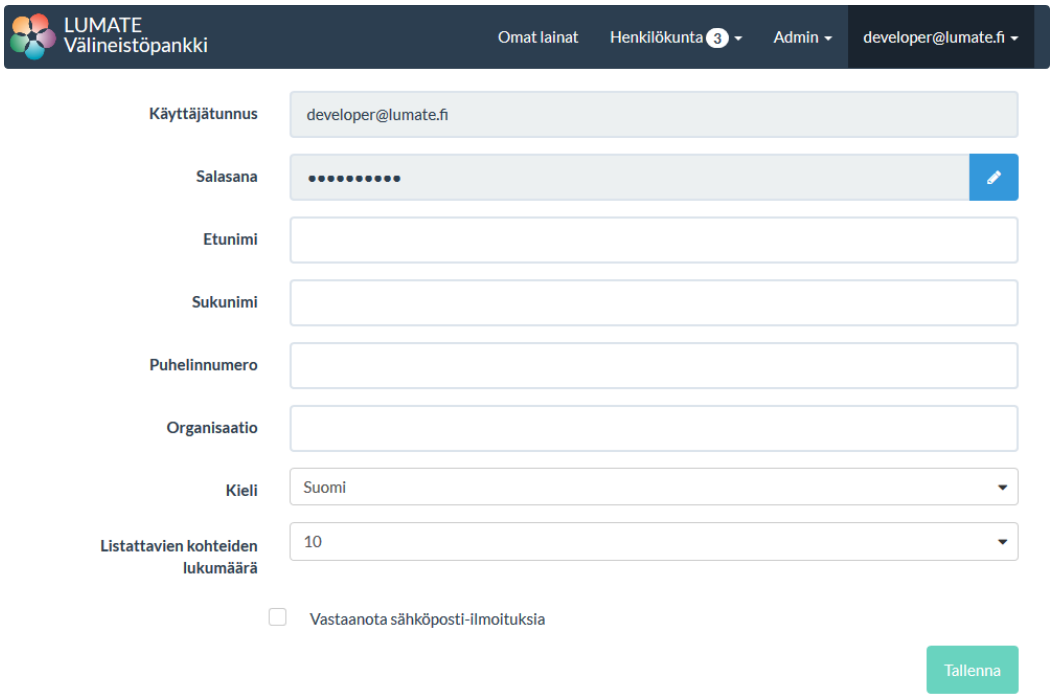
4.4 Ominaisuudet

LUMATE Välineistöpankki on lainanhallintajärjestelmä, ja sen toiminnallisuus keskittyy lainojen luomisen ja niiden hallitsemisen ympärille. Tärkein ydinominaisuus on siis asiakkaan kyky luoda uusia lainoja ja seurata niiden edistymistä. Lisäksi sovelluksesta löytyy esimerkiksi henkilökunnalle tarkoitettu asiakkaiden lainojen hallintatyökalu, sekä kattavat työkalut ylläpitäjälle järjestelmän hallitsemiseen. Tässä alaluvussa käydään läpi

sovelluksen tärkeimmät ominaisuudet. Ominaisuudet on otsikoitu seuraavasti: ominaisuus (kohderyhmä).

4.4.1 Käyttäjätilin asetukset (kaikki)

Kuvassa 9 on esitelty käyttäjätilin asetukset -näkymä, jonka avulla käyttäjä pääsee muokkaamaan käyttäjätilin tietoja ja asetuksia. Henkilökohtaisten tietojen antaminen ei ole pakollista. Tiedot ovat saatavilla käyttäjän itsensä lisäksi järjestelmän ylläpitäjälle ja henkilökunnalle. Asiakkailla ei ole pääsyä toisten asiakkaiden henkilökohtaisiin tietoihin.

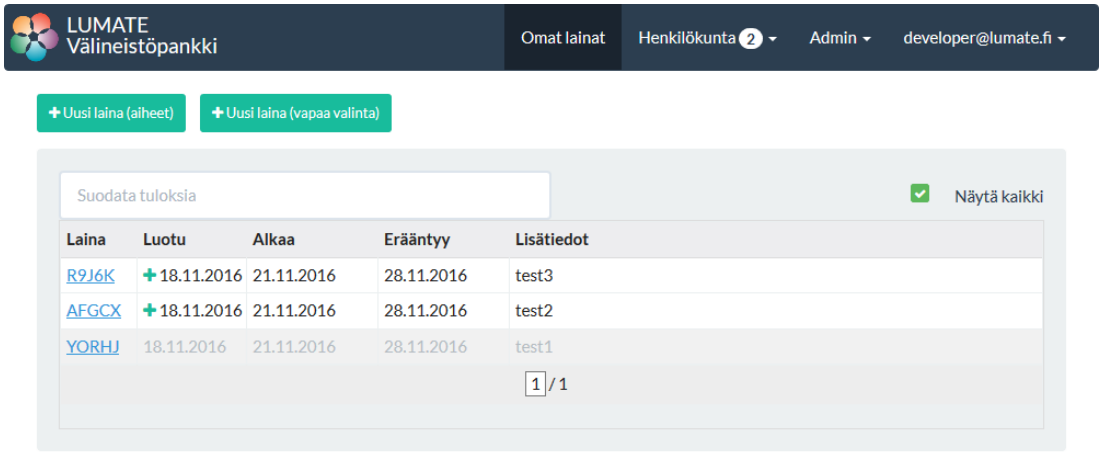


KUVA 9. Käyttäjätilin asetukset -näkymä

4.4.2 Omien lainojen hallinta (asiakas, henkilökunta)

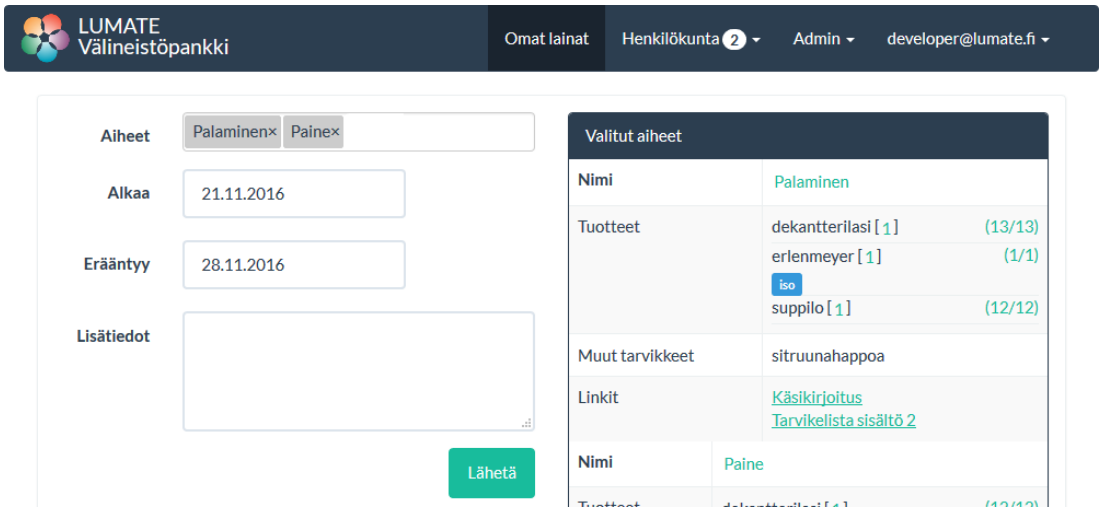
Kuvassa 10 on esitelty omat lainat -näkymä, jonka avulla käyttäjä pääsee hallitsemaan omia lainojaan. Näkymässä on listattu kaikki kirjautuneen käyttäjän luomat lainat. Vanhentuneet eli palautetut ja peruutetut lainat on piilotettu näytä kaikki -valinnan taakse, jotta ajankohtaiset lainat olisivat aina helposti näkyvillä. Lainan tarkempiin tietoihin pää-

see käsiksi klikkaamalla lainan neljän merkin pituista tunnusta, joka on lainan id-tunnuksen alkuosa. Yläreunassa sijaitsevien linkkien avulla käyttäjä pääsee luomaan uusia lainoja.



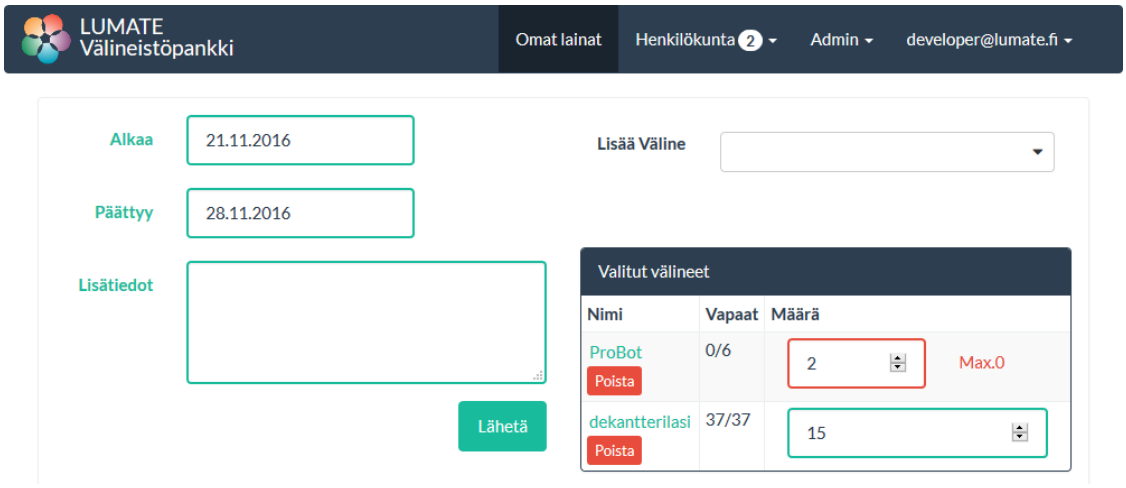
KUVA 10. Omat lainat -näkymä

Kuvassa 11 on esitelty uusi laina (aiheet) -näkymä, jonka avulla käyttäjä pääsee luomaan lainoja, joiden tuotteet sisältyvät aihekokonaisuuksiin. Aiheeseen voi kuulua yksi tai useampi tuote sekä muita tarvikkeita, jotka eivät liity välineistöpankkiin ja joita ei ole tallennettu välinetietokantaan. Tämän lisäksi aihe voi sisältää linkkejä aiheeseen liittyviin välineistöpankin ulkopuolisiin lähteisiin. Tilanteessa missä aiheita on valittuna useita ja ne sisältävät päällekkäisiä tuotteita, laina sisältää vain alhaisimman vaaditun määrän tuotteita.



KUVA 11. Uusi laina (aiheet) -näkymä

Kuvassa 12 on esitelty uusi laina (vapaa valinta) -näkymä, jonka avulla käyttäjä voi luoda lainoja vapaasti haluamistaan tuotteista. Kun käyttäjä on lisännyt tuotteen lainauslistaan, ilmaistaan sen saatavuus lainan aikavälillä ja yksittäisen lainattavan tuotteen määrälle asetetaan yläraja saatavuuden mukaisesti.

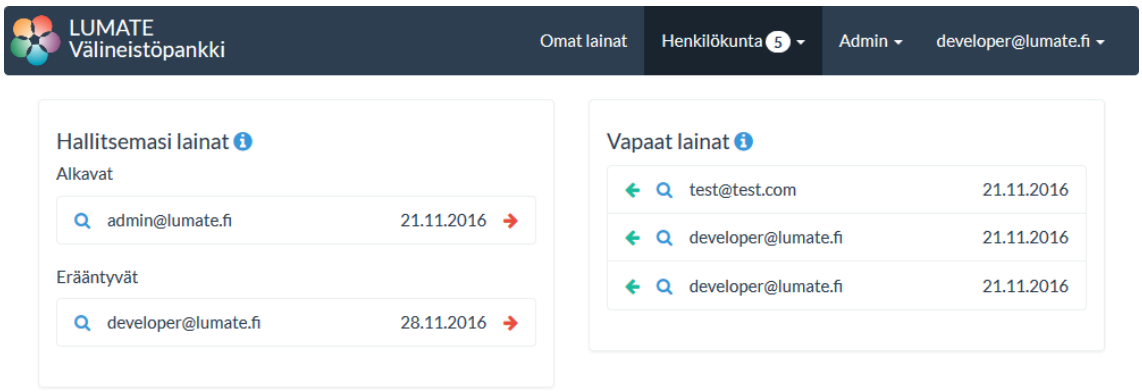


KUVA 12. Uusi laina (vapaa valinta) –näkymä

Molempien lainatyyppien ajankohdan käyttäjä voi itse määrittää, mutta laina saa alkaa aikaisintaan seuraavana päivänä ja sen sallittu kesto on 1-14 vuorokautta. Lainan täytyy alkaa ja päättyä arkipäivänä. Lainan luontihetkellä järjestelmä varaa valitut tuotteet valitulle ajankohdalle ja tallentaa yksittäisten tuotteiden varaustilan muutokset tietokantaan. Kun seuraava käyttäjä yrittää luoda lainaa, joka sisältää päällekkäisiä tuotteita, niiden varaustilat tarkistetaan ja varmistetaan ettei yhdenkään tuotteen varausten lukumäärä ikinä ylitä sen kokonaismäärää inventaariossa.

4.4.3 Asiakkaiden lainojen hallinta (henkilökunta)

Henkilökunnan tärkein tehtävä on valvoa, että tuotteet luovutetaan asiakkaalle lainan alkaessa, ja että asiakas palauttaa ne lainan päättyessä. Tämä on mahdollista kuvassa 13 esitellyn lainanhallinta-näkymän avulla. Henkilökunnan jäsen voi ottaa lainoja vastuulleen, ja vastuulle otetun lainan voi merkitä asiakkaalle luovutetuksi ja lopulta palautetuksi.

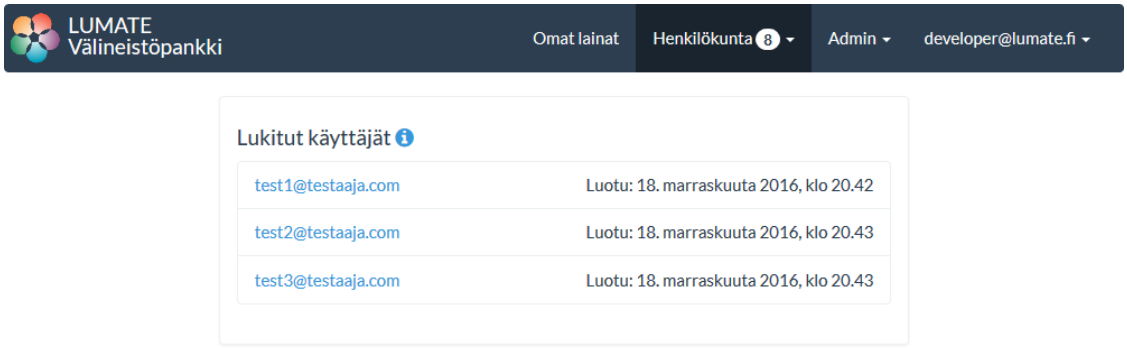


KUVA 13. Lainanhallinta-näkymä

Suurennuslasi-ikonin kautta käyttäjä pääsee lainan omaan laina-näkymään. Laina-näkymässä lainan vastuuhenkilö kykenee merkitsemään sen asiakkaan noutamaksi, jolloin lainan noutoaika tallentuu järjestelmään. Tämän jälkeen lainan voi merkitä palautetuksi, jolloin palautusaika tallentuu ja järjestelmä vapauttaa lainalle varatut tuotteet uusien lainojen käytettäväksi.

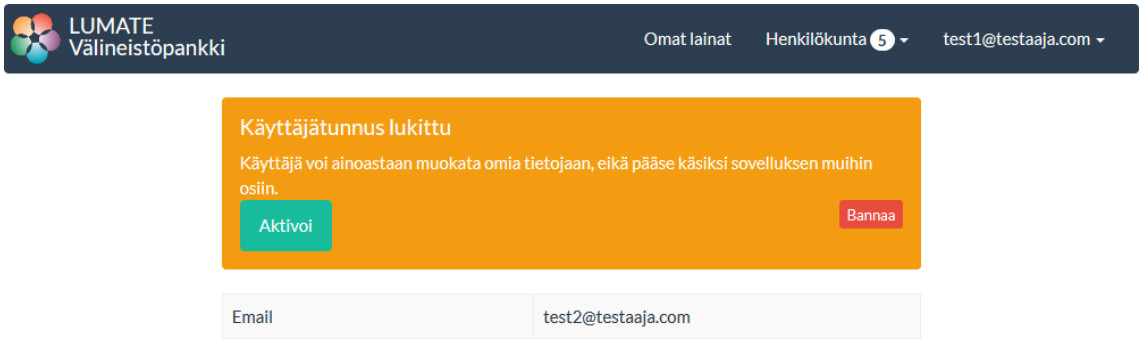
4.4.4 Uusien käyttäjien aktivointi (henkilökunta)

Kun uusi käyttäjä on rekisteröitynyt järjestelmään, käyttäjän pääsy kaikkiin muihin näkymiin lukuun ottamatta käyttäjätilin asetukset -näkymää on estetty. Pääsy sallitaan vasta, kun henkilökunnan jäsen verifioi käyttäjän tiedot ja aktivoi tunnuksen, mikä estää tehokkaasti botteja ja muita haitallisia käyttäjiä pääsemästä käsiksi järjestelmän toimintoihin. Kuvassa 14 on esitelty käyttäjien aktivointi –näkymä, joka listaa kaikki uudet aktivoimattomat käyttäjätunnukset.



KUVA 14. Käyttäjien aktivointi –näkymä

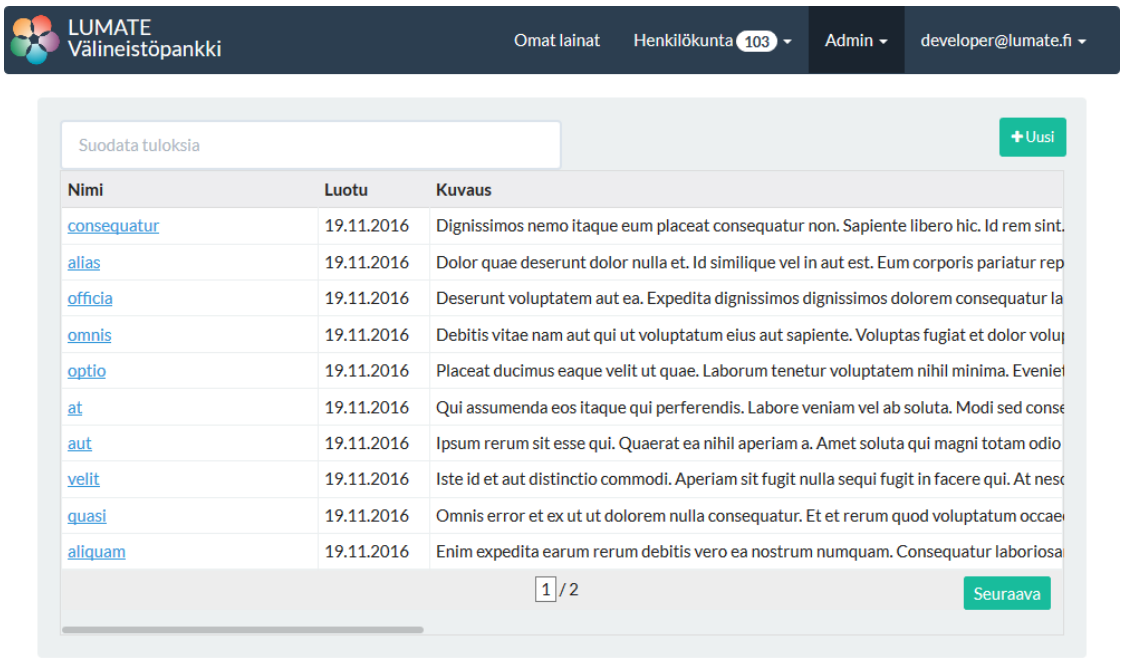
Kuvassa 15 on esitelty lukitun käyttäjän profiilinäkymä. Sen yläreunassa näkyy ikkuna, jonka painikkeiden avulla henkilökunnan jäsen voi joko aktivoida tai bannata käyttäjän. Kun käyttäjä aktivoidaan, järjestelmä lähettää sähköpostitse ilmoituksen käyttäjätunnuksen omistajalle.



KUVA 15. Käyttäjätunnuksen aktivointi-ikkuna profiilinäkymässä

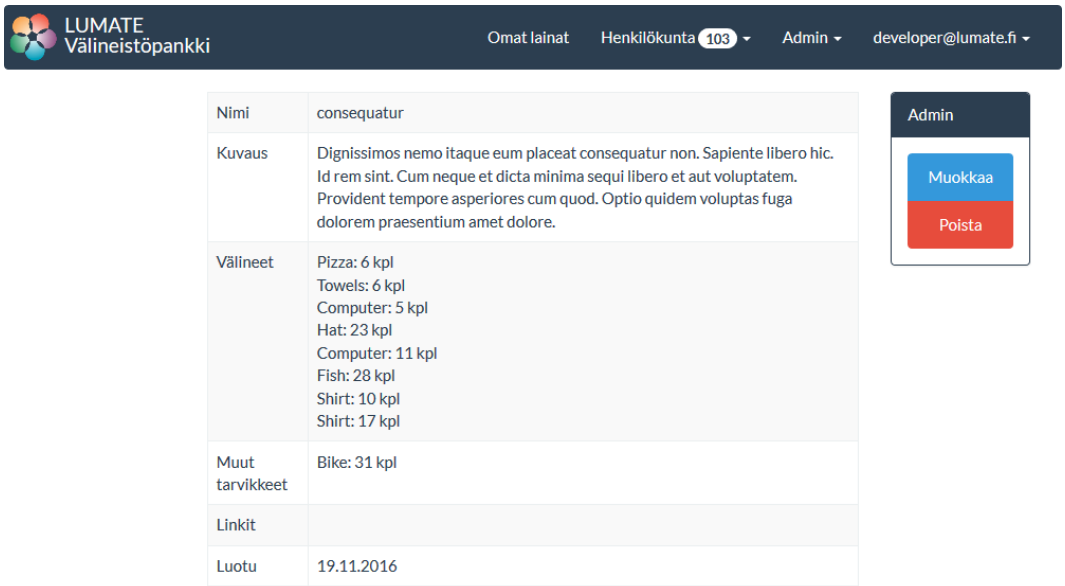
4.4.5 Tietokannan hallinta (ylläpitäjä)

Järjestelmän ylläpitäjä pääsee käsiksi tietokannan hallintanäkymiin yläpalkin admin-pudotusvalikon kautta. Kuvassa 16 on esitelty aiheet-kokoelman hallintanäkymä, jossa on listattuna kaikki kokoelman sisältämät dokumentit. Oikean yläkulman uusi-painikkeen kautta käyttäjä pääsee luomaan uusia dokumentteja.



KUVA 16. Aiheet-kokoelman hallintanäkymä

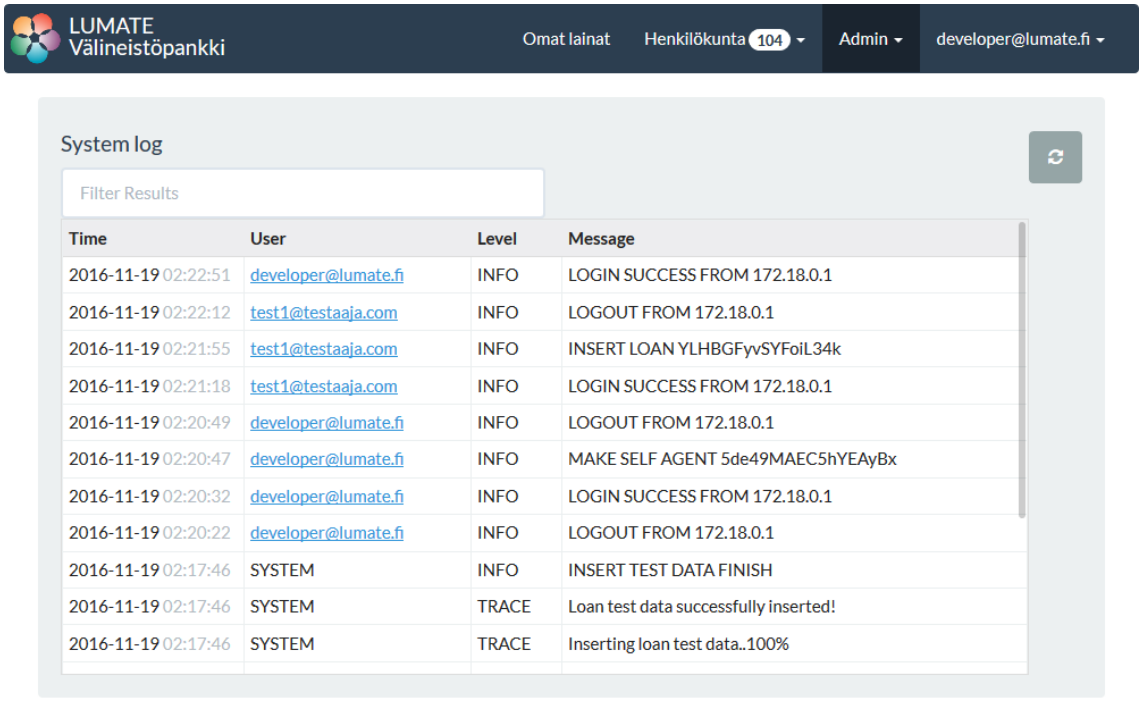
Dokumentin nimeä klikkaamalla käyttäjä ohjataan dokumenttinäkymään, jossa ylläpitäjä voi muokata tai poistaa dokumentin (kuva 17). Muokkaa-painikkeen kautta käyttäjä ohjataan dokumentin muokkausnäkymään. Poista-painike käynnistää poisto-operaation, joka kysyy käyttäjältä ensin varmistuksen vahinkopoistojen välttämiseksi.



KUVA 17. Yksittäisen tietokantadokumentin näkymä ylläpitäjänä

4.4.6 Järjestelmälokit (ylläpitäjä)

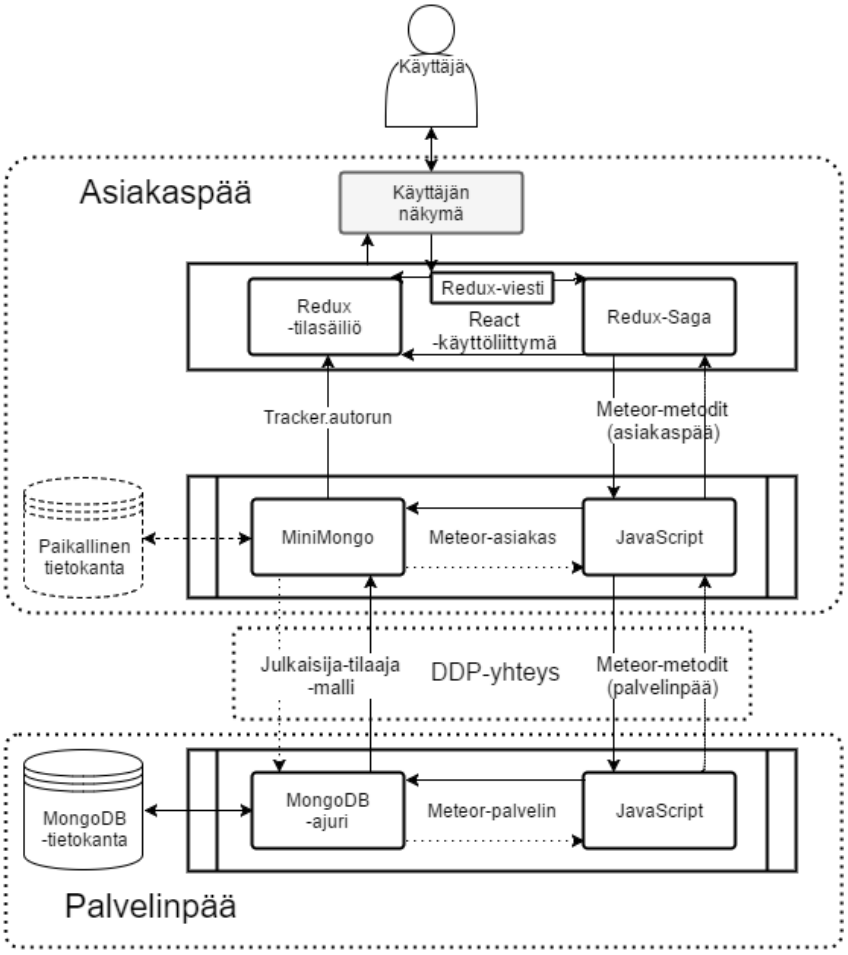
Kuvassa 18 on esitelty järjestelmälokit-näkymä, jonka avulla ylläpitäjä pääsee tarkastelemaan järjestelmän tallentamia lokitietoja. Lokitietoja tallennetaan monista järjestelmän sisäisistä ja käyttäjien aikaansaamista toiminnoista ja virhetilanteista. Tietoja säilytetään palvelimella noin puoli vuotta.



KUVA 18. Järjestelmälokit-näkymä

4.5 Datakerros

Sovelluksen perustana toimivan Meteor-sovellusalustan datakerros koostuu MongoDB-tietokannasta, reaktiivisesta julkaisija-tilaaja-mallista ja asiakaspään paikallisesta Mini-Mongo-tietovarastosta. Lisäksi React-käyttöliittymän tarvitsema data on tallessa Redux-tilasäiliössä. Kuviossa 7 on esitelty sovelluksen datakerros, joka on täysin reaktiivinen, eli tietokannan datan muutokset päivittyvät automaattisesti kaikille asiakkaille. Data kulkee palvelin- ja asiakaspäiden välillä perinteisen Meteor-sovelluksen tapaan, mutta sovelluksen käyttöliittymässä käytetty Redux-tilasäiliö tuo prosessiin ylimääräisen vaiheen, koska data täytyy ensin ladata Meteorilla ja sen jälkeen siirtää tilasäiliön sisälle.



KUVIO 7. Datan kulku välineistöpankissa

4.5.1 Käyttöliittymä

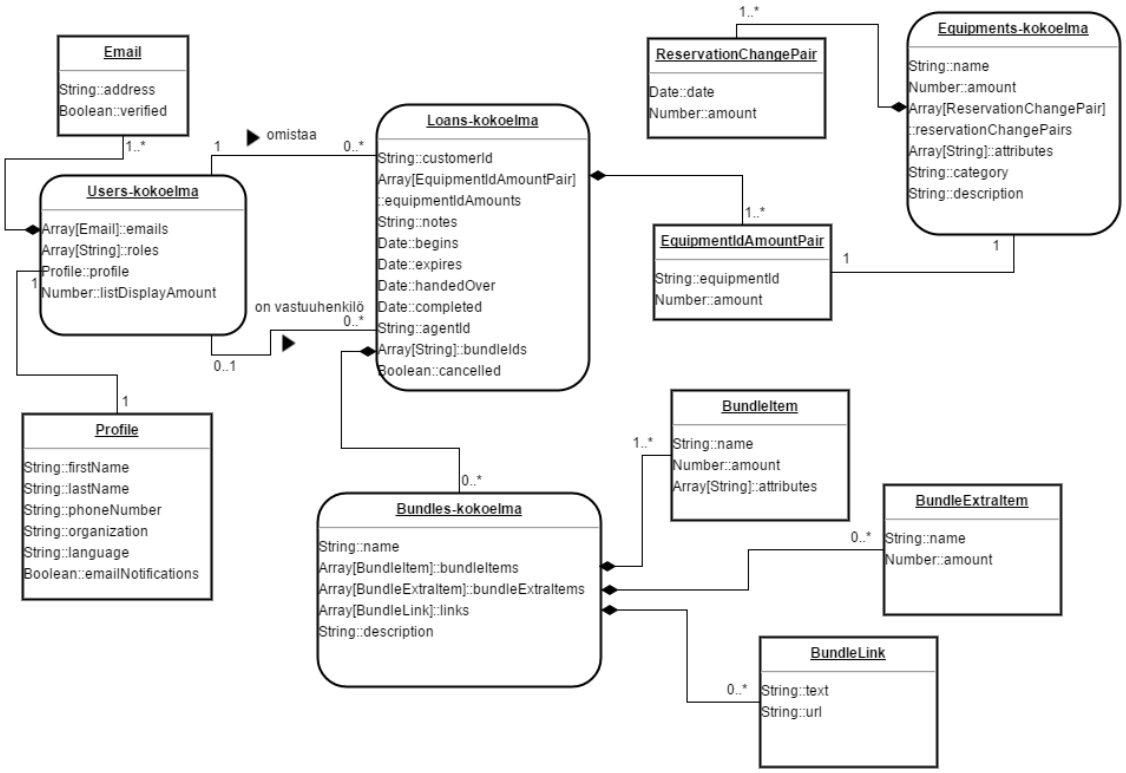
Käyttöliittymädatan käsittelyn selkeyttämiseksi projektissa otettiin käyttöön Redux-tila-säiliö, joka esiteltiin jo aikaisemmin luvussa 3.3.3. Reduxin käyttö Meteor-sovelluksessa vaatii rajapinnan, jonka kautta data saadaan kulkemaan Meteorilta Reduxille. Kuvassa 19 on esitelty rajapinnan ohjelmakoodia. Meteorin tracker.autorun-toiminto varmistaa, että Reduxin tila päivitetään reaktiivisesti aina kun Meteorin datalähteisiin tulee muutoksia.

```
96          // Get announcements
97          Tracker.autorun(() => {
98            // Subscribe to Meteor data
99            const sub = Meteor.subscribe(ANNOUNCEMENTS);
100           if (sub.ready()) {
101             // Get the data from MiniMongo
102             const announcements = Schemas.Announcement.find({}, {
103               transform: null,
104               sort: { createdAt: -1 },
105               limit: 50,
106             }).fetch();
107
108             // Send the data to Redux as ImmutableJS object
109             store.dispatch(announcementsDataLoaded(Immutable.fromJS(announcements)));
110           }
111         });
```

KUVA 19. Datan reaktiivinen päivitys Redux-tilasäiliölle

4.5.2 Palvelinpää ja tietokanta

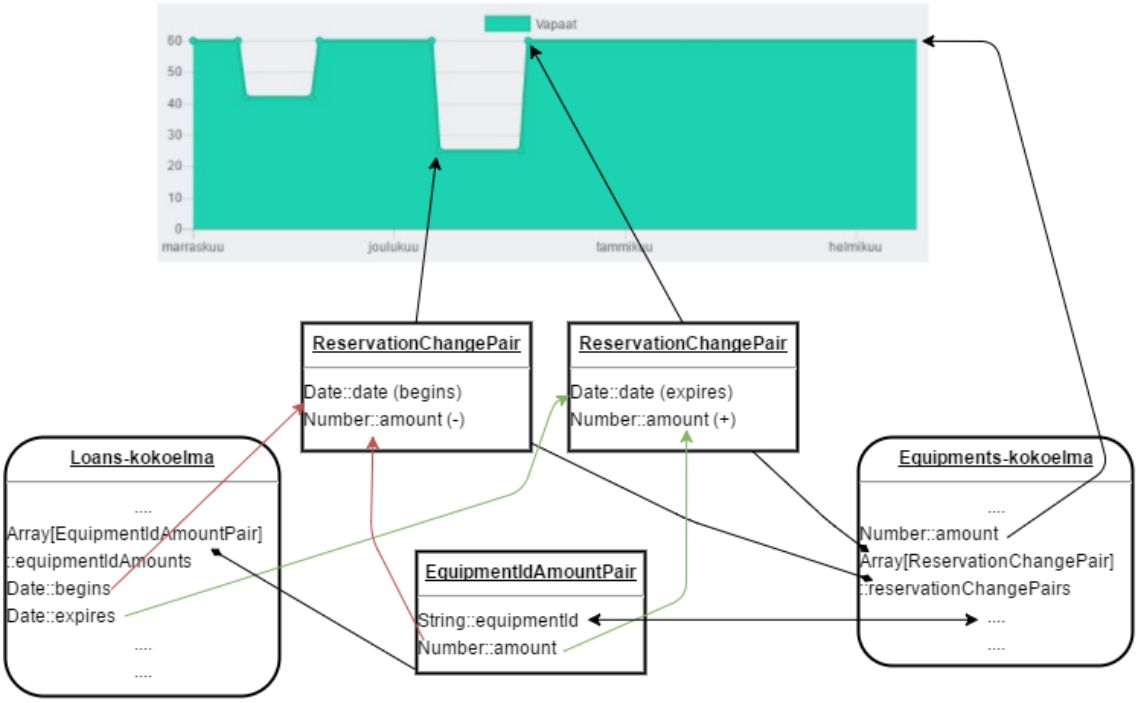
Projekti päätettiin toteuttaa Meteor-sovellusalustalla, jonka ainoa vaihtoehto tietokannaksi projektin aloitushetkellä oli Mongo. Kuviossa 8 on esitelty sovelluksen tietokantamalli. Mallissa on 4 päätaulua eli kokoelmaa, jotka sisältävät erityyppisiä kenttiä. Suorituskykysyistä taulujen välillä esiintyy päällekkäistä dataa, eli tietokanta on normalisoimaton. Ratkaisuun päädyttiin, kun oltiin ensin kokeiltu normalisoitua mallia, jonka suorituskyky osoittautui liian huonoksi.



KUVIO 8. Tietokantamalli

Projektin datassa on useita eri olioiden välisiä relaatioita eli riippuvuuksia, joiden mallintaminen Mongossa voi tuottaa ongelmia. Mongo ei ole relaatiotietokanta ja sen kirjoitusoperaatiot eivät ole atomisia kokoelmatasolla. Tästä seuraa, että datan eheyttä kokoelmien välillä ei voida taata. Sen vuoksi jouduttiin tekemään jonkin verran ylimääräistä logiikkaa ongelmien välttämiseksi, joilta oltaisiin vältytty täysin, jos käytössä olisi ollut esimeriksi SQL- tai graafitietokanta.

Isoimmaksi ongelmaksi osoittautui laina- ja välinekokoelmien välisen suhteen mallintaminen. Järjestelmän toiminnan kannalta on oleellista tuntea yksittäisen välineen varaustila ajan funktiona, eli kuinka monta kappaletta välinettä on varastossa vapaana tietyllä ajanhetkellä. Kuviossa 9 havainnollistetaan varaustilan kuvaajan muodostumista. Siinä näkyy, kuinka yksittäisellä lainalla on alku- ja päättymispäivä-kentät sekä lainatut välineet ja niiden lainamäärät sisältävä equipmentIdAmounts-taulukko. Näistä tiedoista muodostetaan yksittäisen välineen reservationChangePairs-taulukko, jonka alkiot sisältävät suoraan välineen varaustilan kuvaajan muodostamiseen vaadittavat tiedot.



KUVIO 9. Välineen varaustilan muodostuminen

4.6 Tietoturva

Tietoturvasta huolehtiminen on oleellinen osa järjestelmää, jossa käsitellään asiakkaiden henkilökohtaisia tietoja. Tietoturva täytyy ottaa huomioon sekä palvelin- että asiakaspään puolella. Tietoturvaan sisältyy myös palvelun suojaaminen palvelunestohyökkäyksiltä.

Meteorissa palvelimelta välitetään dataa asiakkaalle metodeilla (methods) ja julkaisuilla (publications). Kuka tahansa pystyy ottamaan omalta Meteor-päätteeltään yhteyden palvelimeen ja kutsumaan sen metodeja sekä tilaamaan sen julkaisuja, kunhan vain onnistuu arvaamaan niille annetut nimet. Onkin tärkeää, että niiden tietoturvaan kiinnitetään erityistä huomiota.

4.6.1 Käyttöoikeudet

Käyttöoikeuksien tarkistus suoritetaan jokaisessa metodissa ja julkaisussa, jonka käyttö halutaan rajoittaa vain tietyille rooleille. Kuvassa 20 on esitelty metodi, jonka käyttö on sallittu ainoastaan ylläpitäjä (admin) -roolille.

```
24    run({ name, attributes, amount, category, description }) {
25      // Check that the user is authorized
26      if (Roles.userIsInRole(this.userId, ['admin']) === false) {
27        throw new Meteor.Error(403, 'Not authorized');
28      }
```

KUVA 20. Käyttöoikeuksien tarkistus alanning:roles-lisäosan avulla

Tarkistuksen turvaamiseksi siinä käytettävää id-tunnusta ei tule ikinä lähettää asiakassovellukselta palvelimelle, koska tällöin pahantahtoinen käyttäjä voi toisen käyttäjän id-tunnuksen vakoiltuaan huijata olevansa kyseinen käyttäjä. Id-tunnus on saatavilla metodeissa ja julkaisuissa this.userId-muttujan avulla, jonka Meteor takaa aina kuuluvan avoimen yhteyden kautta kirjautuneelle käyttäjälle (Meteor n.d).

4.6.2 Injektiot

Hyvin yleinen tapa tunkeutua perinteisiä SQL-tietokantoja käyttäviin järjestelmiin ovat nk. SQL-injektiot. Vaikka Meteorissa käytetäänkin MongoDB-tietokantaa, injektioita hyödyntäen myös MongoDB:n kyselyihin on mahdollista ujuttaa haitallisia käskyjä. Niiden avulla on pahimmillaan mahdollista vaikkapa tyhjentää koko tietokanta yhdellä komennolla. Metodien ja julkaisujen parametrien validointi on tehokas suoja injektioita vastaan. Kuvassa 21 on esitelty Meteorin mdg:validated-method-lisäosan avulla luotu metodi, joka validoi parametrit ja palauttaa virheen mikäli niiden tyypit eivät vastaa validate-kentässä määriteltyjä tyyppejä.

```
14  export const insert = new ValidatedMethod({
15    name: 'Methods.equipments.insert',
16    mixins: [CallPromiseMixin],
17    validate: new SimpleSchema({
18      name: { type: String },
19      attributes: { type: [String], optional: true },
20      amount: { type: Number },
21      category: { type: String, optional: true },
22      description: { type: String, optional: true },
23    }).validator(),
24    run({ name, attributes, amount, category, description }) {
```

KUVA 21. Metodin parametrien validointi

4.6.3 DDP-floodaus

Yksi tapa tehdä palvelunestohyökkäys Meteor-sovellusta vastaan on muodostaa DDP yhteys palvelimeen, skannata sen metodien ja julkaisujen nimet asiakaspään koodista ja ylikuormittaa palvelin DDP-kutsuilla. DDP-floodaukselta suojautumiseen on saatavilla Meteor-paketti ddp-rate-limiter, jolla kutsujen määrää voidaan rajoittaa. Kuvassa 22 on esitelty koodia, joka rajoittaa metodit viiteen ja julkaisut kymmeneen kutsuun per sekunti.

```
19  // Set rate limiting on methods and publications
20  let methodNames = Immutable.fromJS([]);
21  let pubNames = Immutable.fromJS([]);
22  // Map method names
23  Object.keys(Methods).forEach(superKey => {
24    Object.keys(Methods[superKey]).forEach(key => {
25      const methodName = Methods[superKey][key].name;
26      if (methodName.length > 0) {
27        methodNames = methodNames.push(methodName);
28      }
29    });
30  });
31  // Map publication names
32  Object.keys(PUBLICATION_NAMES).forEach(key => {
33    pubNames = pubNames.push(PUBLICATION_NAMES[key]);
34  });
35  // Only allow 5 method calls per second
36  DDPRateLimiter.addRule({
37    type: 'method',
38    name(name) {
39      return _.includes(methodNames.toJS(), name);
40    },
41    // Rate limit per connection ID
42    connectionId() { return true; },
43  }, 5, 1000);
44  // Only allow 10 publication subscriptions per second
45  DDPRateLimiter.addRule({
46    type: 'subscription',
47    name(name) {
48      return _.includes(pubNames.toJS(), name);
49    },
50    // Rate limit per connection ID
51    connectionId() { return true; },
52  }, 10, 1000);
```

KUVA 22. DDP Rate limiting -koodia

Menetelmä ei kuitenkaan yksistään täysin riitä suojaamaan DDP-floodaukselta, koska sillä ei voi rajoittaa uusien DDP-yhteyksien avaamista palvelimelle (Corey, 2016). Hyökkääjä voi avata uusia DDP-yhteyksiä nopeaan tahtiin ja ylikuormittaa palvelimen. Jotta palvelu olisi täysin suojattu, täytyy myös http-kutsujen määrää rajoittaa, mikä on helpointa toteuttaa web-palvelimen asetuksissa.

4.7 Palvelininfrastruktuuri

LUMATE-keskuksen verkkopalvelut muodostuvat kahdesta osasta. Lumate.fi on Wordpress-sisällönhallintaohjelmistolla toteutettu staattinen web-sivusto, kun taas lumate.fi/valineistopankki on dynaaminen JavaScript web-sovellus. Tämä opinnäytetyö keskittyy näistä jälkimmäiseen, mutta palvelininfrastruktuurin kannalta palvelua on ajateltava kokonaisuutena.

Projektin alussa välineistöpankki-sovellus asetettiin toimimaan Tampereen teknillisen yliopiston ylläpitämälle palvelimelle, missä myös Wordpress-kotisivua ylläpidettiin. Palvelimella ilmeni kuitenkin ajoittain epävakautta ja projektin tilaajan kanssa päädyttiin lopulta siirtämään kaikki LUMATE-keskuksen verkkopalvelut kaupalliselle palveluntarjoajalle. Siirto päätettiin toteuttaa kahdessa vaiheessa: kehitysvaiheessa oleva välineistöpankki-sovellus siirrettäisiin ensin ja Wordpress sivusto myöhemmin loppuvuodesta.

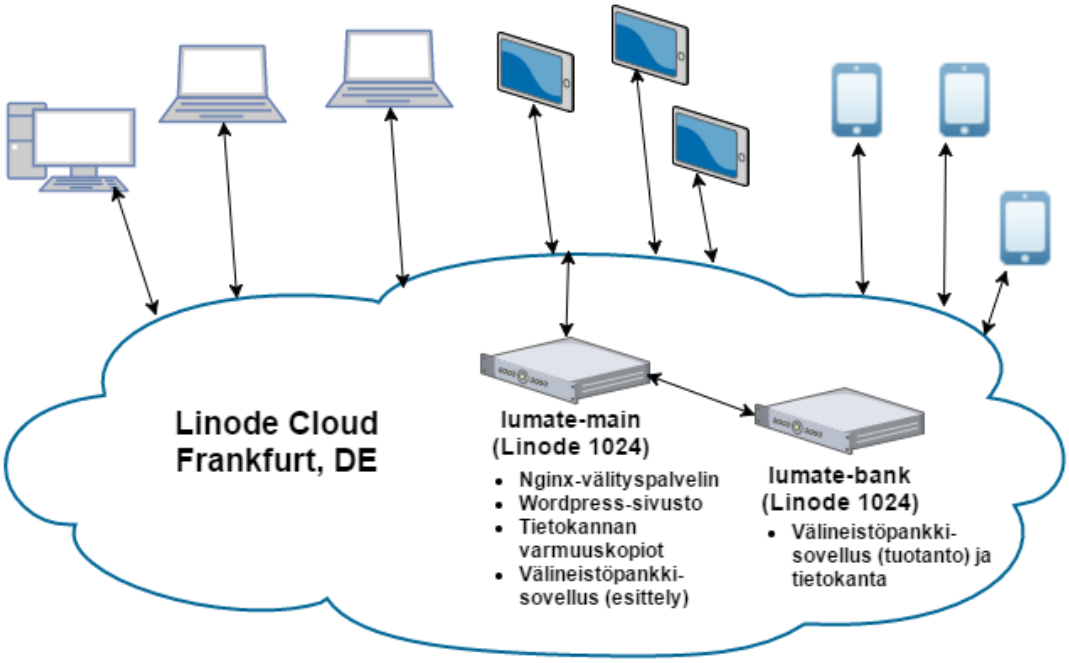
Palveluntarjoajaa valittaessa tärkeimmäksi kriteeriksi asetettiin edullisuus. Palvelun käyttäjämääräksi arvioitiin ainakin alussa enintään muutamia kymmeniä asiakkaita, mistä johtuen palvelun suorituskykyvaatimukset eivät olleet erityisen suuret. Palveluntarjoajaa etsittiin internetistä ja tuloksista nousi esille varsin edulliselta vaikuttanut DigitalOcean, jonka edullisimman pilvipalvelimen hinta oli 5 euroa kuukaudessa (taulukko 2).

TAULUKKO 2. DigitalOcean ja Linode, pilvipalvelinratkaisut hintaluokassa 20$/kk

Tarjoaja	DO	DO	DO	Linode	Linode
Hinta	5$/kk	10$/kk	20$/kk	10$/kk	20$/kk
RAM	512MB	1024MB	2048MB	2048MB	4096MB
Prosessori	1 ydin	1 ydin	2 ydintä	1 ydin	2 ydintä
Levytila	20GB SSD	30GB SSD	40GB SSD	24GB SSD	48GB SSD
Datakatto	1TB	2TB	3TB	2TB	3TB

Välineistöpankki-sovellus siirrettiin aluksi DigitalOceanille, kunnes huomattiin, että kilpaileva palveluntarjoaja Linode oli joitakin kuukausia aiemmin tuplannut edullisimpien palveluidensa RAM-kapasiteetin (Linode 2016). Päätös Linodelle vaihtamisesta oli helppo, sillä molempien ratkaisut olivat muuten varsin samankaltaiset.

Linodelle luotiin kaksi pilvipalvelinta (kuvio 10). Ensimmäisen (lumate-main) tehtäviksi asetettiin välityspalvelimena toimiminen, Wordpress-sivuston palvelu, välineistöpankki-sovelluksen tietokannan varmuuskopioiden säilyttäminen sekä välineistöpankki-sovelluksen lavastusympäristön (staging environment) palvelu. Toisen (lumate-bank) ainoa tehtävä on palvella välineistöpankki-sovelluksen tuotantoympäristöä (production environment).



KUVIO 10. Verkkopalvelujen infrastruktuuri

Nykyaikana tärkeäksi osaksi monia verkkopalveluita on noussut korkea käytettävyys (High Availability), joka tarkoittaa palvelua, jonka toimivuus ei ole riippuvainen yksittäisen komponentin vikaantumisesta (Hakala 2014). Korkean käytettävyyden toteuttaminen vaatisi kuitenkin lisää virtuaalipalvelimia, mikä moninkertaistaisi kustannukset. LUMATE-keskus on pienen budjetin toimija, ja palvelun käyttäjämäärä tulee odotusten mukaan olemaan alhainen. Tästä syystä päädyttiin kompromissiratkaisuun: välineistöpankille tehtiin oma erillinen palvelin, mikä vähentää riskiä koko sivuston kaatumiselle esimerkiksi palvelun resurssit vievän suuren hetkellisen kuormituksen aikana. Lisäksi Linoden palvelimet itsessään ovat korkean käytettävyyden mukaisesti käytännössä aina saatavilla, joten riski koko verkkopalvelun kaatumiselle ilman tarkoituksellista palvelunestohyökkäystä on pieni.

5 TESTAUS

Tässä luvussa tutustutaan luvussa 4 esitellyn sovelluksen testaukseen. Sovelluksen tekniset valinnat toivat omat haasteensa testien toteuttamiseen, ja aikarajoitteiden vuoksi testauta ei ehditty toteuttaa täysin tavoitteiden mukaisesti. Luvun tarkoitus on tarjota lukijalle tietoa JavaScript-pohjaisen web-sovelluksen testauskäytännöistä ja testien automatisoinnista.

5.1 Haasteet

Meteor-sovellusalusta teki projektin testaamisesta haastavaa puutteellisten testausominaisuuksiensa vuoksi. Meteorissa ei ennen keväällä 2016 julkaistua 1.3-versiota ollut lainkaan virallista tukea yksikkötestaukselle. Versio 1.3 toi mukanaan meteor test -komennon, jota ei yrityksistä huolimatta onnistuttu ajamaan onnistuneesti projektin ohjelmakoodilla. Lisäksi komennon käyttäminen vaatisi jokaisen ohjelmakoodimuutoksen jälkeen usean sekunnin uudelleenlatausajan, mikä tekee testien jatkuvasta ajamisesta kömpelöä.

Meteorin tuomien haasteiden ja rajallisten resurssien vuoksi ns. testivetoista kehitystä (test-driven development) ei otettu käyttöön. Valtaosa testeistä päätettiin jättää tehtäväksi projektin opinnäytetyö-osuuden valmistumisen jälkeen, kun aikaa olisi käytettävissä enemmän. Yksikkötestejä tehtiin jo heti projektin alussa muutama, jotta niiden lisääminen projektin myöhemmässä vaiheessa olisi helppoa esimerkkien pohjalta. Monimutkaisempien integraatiotestien sisällyttämistä päätettiin tutkia vasta opinnäytetyön valmistuttua.

5.2 Yksikkötestit

Koska Meteorin oma testauskomento oli vaikeakäyttöinen, päädyttiin yksikkötestit toteuttamaan yleisillä npm-työkaluilla, jotka ovat Mocha, Chai ja Sinon. Mocha on JavaScript-testikehys, jolla itse testejä ajetaan, ja jonka avulla tulokset saadaan tulostumaan konsoli-ikkunaan reaaliaikaisesti jokaisen ohjelmakoodimuutoksen jälkeen. Chain avulla suoritetaan testiarvojen oikeellisuustarkistukset (assertions). Sinonin avulla puolestaan

pystytään luomaan ns. vakoojafunktioita, joiden avulla tutkitaan testattavien funktioiden tilaa ja käyttäytymistä testin aikana. Lisäksi käyttöön otettiin erityisesti React-komponenttien testaukseen suunnattu Enzyme-työkalu.

Nykyaikaiset JavaScript-testityökalut nojaavat vahvasti ECMAScript 6:ssa julkaistuun moduulit-syntaksiin, jonka avulla yksittäisiä tiedostoja ja muuttujia voidaan sisällyttää import-käskyn avulla. Yksikkötestauksessa yksittäisten komponenttien eristäminen on tärkeää, ja moduulien avulla se on helppoa.

Ongelmaksi muodostuu kuitenkin Meteorin käyttämät globaalit muuttujat, joiden esiintyminen import-komennolla sisällytetyissä tiedostoissa aiheuttaa testien kaatumisen. Syynä tähän on se, että Meteorin globaalit muuttujat vaativat taustalle toimivan Meteor-instanssin, joka taasen vaatisi Meteorin oman testauskomennon käyttöä. Ongelman kiertäminen on mahdollista Testdouble-testaustyökalun avulla. Työkalu mahdollistaa sisällytetyissä tiedostoissa käytettyjen muuttujien ylikirjoittamisen omalla muokatulla versiolla.

Kuvassa 23 on esitelty käyttöliittymän yksikkötesti. Rivillä 18 nähdään, kuinka Account-tiedostossa käytetty Roles-muuttuja muuttuja kirjoitetaan yli tyhjällä oliolla. Rivillä 31 Account-komponentista muodostetaan testattava komponentti käärimällä se tarvittavan Redux-tilasäiliön Provider-komponentin sisälle. Rivillä 36 määritellään yksikkötesti, joka testaa komponentin yhtä osaa.

```
// Import libraries
import React from 'react';
import chai, { expect } from 'chai';
import chaiEnzyme from 'chai-enzyme';
import { shallow, render, mount } from 'enzyme';
import sinon from 'sinon';
import td from 'testdouble';
import { Provider } from 'react-redux';
import configureStore from 'redux-mock-store';
import Immutable from 'immutable';
import _ from 'lodash';

const { describe, it, before, after } = global;
chai.use(chaiEnzyme());

// Apply testdoubles
const Roles = td.object();
td.replace('meteor/alanning:roles', { Roles });

// Import the component
const Account = require('./Account').default;

// Create default settings
const mockStore = configureStore([]);
const defaultStore = mockStore({ form: {} });
const defaultProps = {
  handleSubmit: () => {},
  handleEditPassword: () => {},
  handleUpdatePassword: () => {},
};
const TestComponent = args => <Provider store={args.store || defaultStore}>
  <Account {...defaultProps} {...args} /></Provider>;


describe('ui.account.components.Account', function () {
  it('renders 7 input fields', () => {
    const node = render(<TestComponent />);
    expect(node.find('input')).to.have.length(7);
  });
```

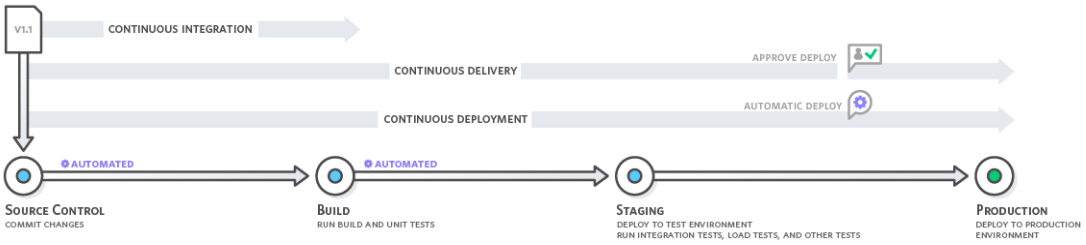
KUVA 23. Käyttöliittymäkomponentin yksikkötestausta

Ohjelmoitaessa on suositeltavaa asettaa testit ajettaviksi automaattisesti jokaisen ohjelmakoodimuutoksen jälkeen, jolloin kehittäjä saa reaaliaikaisesti palautteen, mikäli tehty muutos rikkoo ohjelman. Lisäksi on suositeltavaa tarkistaa ohjelmakoodi myös ESLint-työkalulla ennen muutosten tallentamista Git-repositorioon.

5.3 Jatkuva integraatio

Jatkuvalla integraatiolla tarkoitetaan käytäntöä, missä kehitystiimin jäsenet lähettävät valmiit ohjelmakoodimuutokset yhteiselle integraatiopalvelimelle, joka suorittaa niille automaattisia testejä. Tämä tapahtuu yleensä useita kertoja päivän aikana. Mikäli tiimin jäsen lähettää testit rikkovan muutoksen, vika etsitään ja korjataan heti ennen muiden töiden jatkamista. Jatkuvaan integraatioon liittyvät läheisesti myös käsitteet jatkuva toimitus (continuous delivery) sekä jatkuva julkaisu (continuous deployment). (Amazon Web Services n.d.)

Jatkuvan integraation tarkoitus on nopeuttaa ohjelmiston virheiden löytymistä, parantaa sen laatua, ja lyhentää sen julkaisujen välillä kuluvaa aikaa. Kuviossa 11 on esitelty tyypillinen jatkuvan integraation malli. Ohjelmiston uusi versio (kuviossa V1.1) siirretään ensin keskitettyyn versionhallintaan (source control), mistä se edelleen automatisoidusti toimitetaan rakennettavaksi ja testattavaksi (build). Jos käytössä on lisäksi jatkuva toimitus tai jatkuva julkaisu, ohjelmisto voidaan lisäksi testien onnistuessa pystyttää lavastus- ja tuotantopalvelimelle automaattisesti. (Amazon Web Services n.d.)



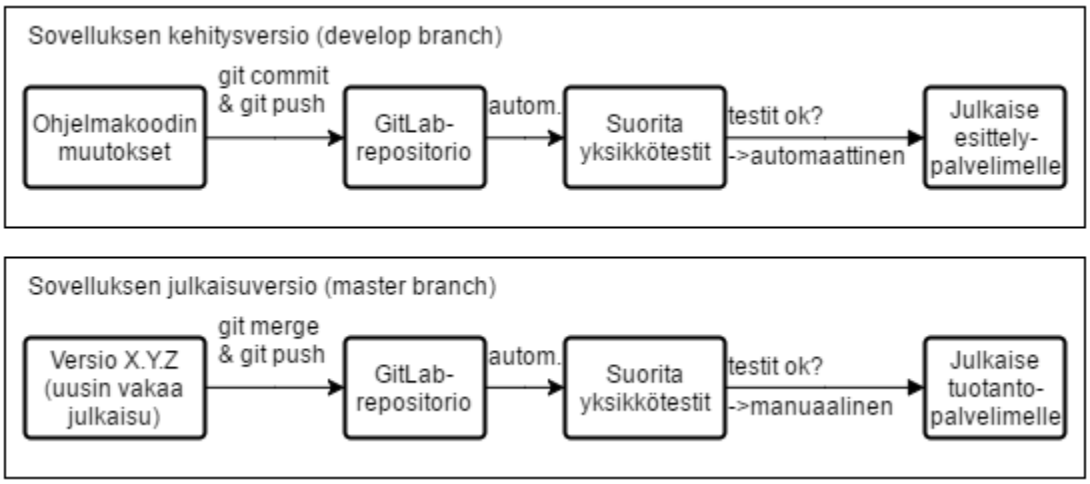
KUVIO 11. Jatkuvan integraation malli (Amazon Web Services n.d)

Jatkuvan integraation käyttöönottoa projektissa sivuttiin jo aikaisemmin luvussa 3.4. GitLabissa se tapahtuu lisäämällä projektiin gitlab-ci.yml-asetustiedosto, joka sisältää vaadittavat asetukset ja shell-komentosarjat. Kuvassa 24 on esitelty asetustiedoston deploy_to_staging-komentosarja, jonka tehtävä on julkaista sovellus esittelypalvelimelle. Komentosarja ajetaan tilanteessa, jossa GitLab-repositorio on vastaanottanut muutoksia projektin kehityspuulle (develop branch) ja niiden testit ovat menneet läpi virheettä.

```
deploy_to_staging:
  stage: deploy
  only:
    - /^develop$/
  environment:
    name: staging
    url: https://www.lumate.fi/staging
  script:
    # Install rsync
    - apt-get update
    - apt-get install -y rsync
    # Get SSH key
    - mkdir -p ~/.ssh
    - cat my_known_hosts >> ~/.ssh/known_hosts
    - (umask  077 ; echo $LUMATE_MAIN_SSH_KEY | base64 --decode > ~/.ssh/id_rsa)
    # Build meteor
    - meteor build --directory ../build --allow-superuser
    # Copy files to build
    - cp .docker/Dockerfile ../build/bundle/Dockerfile
    - cp .docker/docker-compose.yml ../build/
    - cp .docker/docker-compose.stag.yml ../build/
    - cp .bash/fixMongoRS.sh ../build/
    - cp .docker/startup.sh ../build/
    # Compress build & send to server
    - rsync -azh --delete ../build $LUMATE_MAIN_HOST:./
    # Start docker
    - ssh $LUMATE_MAIN_HOST bash ./build/startup.sh stag
```

KUVA 24. Gitlab-ci.yml-asetustiedostoa

Kuviossa 12 on esitelty projektissa käytetty jatkuvan integraation työnkulku. Sovelluksen kehitysversio testataan ja julkaistaan automaattisesti, ja sen julkaisuversio testataan automaattisesti mutta julkaistaan manuaalisesti. Projekti siis täyttää jatkuvan toimituksen määritelmän (Ramos 2016).



KUVIO 12. Jatkuvan integraatio työnkulku

5.4 Käyttöliittymän suorituskykymittaukset

Käyttöliittymän suorituskykyä mitattiin suurilla datamäärillä. Satunnaisgeneraattorilla luotiin tietokantaan tuhansia kohteita, jonka jälkeen käyttöliittymän eri näkymien käytettävyyttä tarkkailtiin. Suorituskykyä arvioitiin silmämääräisesti ja käyttöä haitanneet käyttöliittymän hidastumiset mitattiin noin sekunnin tarkkuudella. Testit ajettiin sovelluksen versiolla 1.0.12.

Taulukossa 3 on esitelty tietokantakokoelmien dokumenttien määrät testeissä. Testi 1 aikana lainat olivat aktiivisia, ja testi 2 aikana epäaktiivisia. Testien ulkopuolelle jätettiin tiedotekokoelma, jonka sisältämien dokumenttien lukumäärä järjestelmässä on rajoitettu 50 kappaleeseen.

TAULUKKO 3. Tietokannan alkioiden lukumäärät testien aikana

	Käyttäjät	Välineet	Aiheet	Lainat
Testi 1	500	2000	200	5000
Testi 2	10	500	200	10000

Mobiililaitteiden suorituskyvyn selvittämiseksi testit ajettiin PC-kokoonpanon lisäksi myös Oneplus One -älypuhelimella. Taulukossa 4 on esitelty testikokoonpanojen tärkeimmät spesifikaatiot.

TAULUKKO 4. Testikokoonpanot

	Prosessori	Keskusmuisti	Web-selain
PC	Intel i5 4690K @4GHz x86-64	16GB	Google Chrome
Älypuhelin	Qualcomm Snapdragon 801 @2.5GHz ARMv7	3GB	Mozilla Firefox

5.4.1 Testi 1

Ensimmäistä testiä varten tietokantaan luotiin 500 käyttäjää, 2000 välinettä, 200 aihetta ja 5000 lainaa. Pyrkimyksenä oli selvittää, kuinka hyvin käyttöliittymän käytettävyys säilyy tilanteessa, missä välineitä ja aktiivisia lainoja on tietokannassa epätavallisen paljon. Taulukossa 5 on esitelty ensimmäisen testin tulokset.

TAULUKKO 5. Testi 1, näkymien ylimääräiset viiveet

	PC	Älypuhelin
Omat lainat	Ei havaittavaa	Ei havaittavaa
Uusi laina (aiheet)	3s	17s
Uusi laina (vapaa valinta)	3s	28s
Lainanhallinta	45s, optimoituna 3s	57s, optimoituna 12s
Käyttäjien aktivointi	2s	4s
Kaikki välineet	1s	4s
Kaikki aiheet	Ei havaittavaa	5s
Kaikki lainat	1s	10s
Kaikki käyttäjät	1s	10s
Muut näkymät	Ei havaittavaa	Ei havaittavaa

Sovelluksen toimintaa testattiin aluksi asiakaskäyttäjällä. Uusi laina -näkymissä ilmaantui PC-kokoonpanolla 3 sekunnin ja älypuhelimella 17 ja 28 sekunnin viiveet, minkä selittää välineiden suuri määrä (2000). Muissa asiakkaan näkymissä ei havaittu huomattavaa hidastumista.

Seuraavaksi testattiin henkilökunnan näkymiä. Lainanhallintanäkymässä havaittiin merkittävää hidastumista: 5000 aktiivista lainaa aiheutti PC-kokoonpanolla 45 sekunnin ja älypuhelimella 57 sekunnin viiveet, minkä lisäksi älypuhelimen Firefox-selain antoi monta kertaa varoituksen sovelluksen jumiutumisesta. Näkymän vapaat lainat -sarake optimoitiin näyttämään yhtäaikaisesti enintään 50 lainaa, jonka jälkeen latausviive saatiin tiputettua PC-kokoonpanolla 3 sekuntiin ja älypuhelimella 12 sekuntiin.

Henkilökunnan ja ylläpitäjän näkymissä esiintyi 4 – 10 sekunnin viiveitä, minkä selittää osittain yläpalkissa sijaitseva lainanhallinta-näkymän laskuri. Laskuri aiheuttaa ylimääräistä kuormaa, koska sen vuoksi joudutaan lataamaan osittaiset dokumentit kaikista 5000 aktiivisesta lainasta.

5.4.2 Testi 2

Toista testiä varten tietokantaan luotiin 10 käyttäjää, 500 välinettä, 200 aihetta ja 10 000 lainaa. Lainat ovat ainoa kokoelma missä dokumenttien määrä tulee tuotantokäytössä kasvamaan pitkän ajan kuluessa rajatta. Pyrkimyksenä oli selvittää, kuinka hyvin käyttöliittymän käytettävyys säilyy tilanteessa, missä tietokantaan on ehtinyt kertyä huomattavan suuri määrä epäaktiivisia lainoja. Taulukossa 6 on esitelty toisen testin tulokset.

TAULUKKO 6. Testi 2, näkymien ylimääräiset viiveet

	PC	Älypuhelin
Omat lainat	Ei havaittavaa, epäaktiiviset lainat näkyville kytkettäessä 3s viive	Ei havaittavaa, epäaktiiviset lainat näkyville kytkettäessä 3s viive
Uusi laina (aiheet)	Ei havaittavaa	3s
Uusi laina (vapaa valinta)	Ei havaittavaa	7s
Kaikki välineet	Ei havaittavaa	3s
Kaikki aiheet	Ei havaittavaa	1s
Kaikki lainat	3s	16s
Kaikki käyttäjät	Ei havaittavaa	3s
Muut näkymät	Ei havaittavaa	Ei havaittavaa

Omat lainat -näkymän testissä vaikuttavana tekijänä oli asiakkaan n. 750 epäaktiivista lainaa. Näkymä piilottaa oletuksena epäaktiiviset lainat, jolloin ne eivät vaikuta näkymän suorituskykyyn lainkaan. Epäaktiivisten lainojen asettaminen näkyville aiheutti sekä PC-kokoonpanolla että älypuhelimella 3 sekunnin viiveen.

Muissa näkymissä esiintyneet viiveet olivat yhdenmukaisia ensimmäisen testin kanssa. Testin aikana kaikki lainat olivat epäaktiivisia, jolloin ensimmäisessä testissä ilmaantunut ylimääräinen viive henkilökunnan ja ylläpitäjän näkymissä jäi pois. 10 000 lainaa aiheutti kaikki lainat -näkymässä PC-kokoonpanolla 3 sekunnin ja älypuhelimella 16 sekunnin viiveet.

5.4.3 Yhteenveto

Pitkän ajan kuluessa tuotantopalvelimelle kertynee suuri määrä epäaktiivisia lainoja. Testi 2 osoitti, että 10 000 epäaktiivista lainaa aiheutti näkyvissä vain marginaalisia viiveitä. Kaikki lainat -näkymän mobiilikäyttöliittymässä ilmaantunut 16 sekunnin viive oli merkittävä, mutta koska ylläpitäjän näkymiä on tarkoitus käyttää pääasiassa työpöytäkoneilla, ei sillä ole merkittävää vaikutusta kokonaisuuden kannalta.

Tuotantokäyttöä ajatellen tuloksista suurimpana riskitekijänä esille nousee testi 2 aikana mobiililaitteella ilmaantunut 7 sekunnin viive uusi laina (vapaa valinta) -näkymässä. Ainoa näkymässä viiveeseen vaikuttava välineet-kokoelma sisälsi vain 500 dokumenttia, jonka saavuttaminen lienee realistista palvelun kasvaessa. Huomioitavaa on kuitenkin, että testeissä satunnaisgeneraattorilla luodut dokumentit sisältävät suhteellisen ison määrän dataa, ja tuotantokäytössä vastaava määrä dokumentteja aiheuttanee huomattavasti vähemmän rasitusta käyttöliittymälle.

Kaiken kaikkiaan testit osoittivat, että käyttöliittymä selviytyy hyvin myös tilanteissa, joissa tietokanta sisältää erittäin suuren määrän dokumentteja. Tilanteet joissa merkittäviä viiveitä syntyi, sisälsivät yleensä moninkertaisen määrän dokumentteja tuotantokäytön odotuksiin verrattuna. Todennäköisesti tulee kestämään vuosia, ennen kuin varteenotettavaa hidastumista on odotettavissa. Tilanteen kehitystä tuotantopalvelimella on tarpeen seurata erityisesti asiakkaan näkymien kohdalla, joiden tulisi toimia kaikilla laiteympäristöillä moitteettomasti. Tietokantakyselyjä optimoimalla lienee mahdollista saavuttaa merkittäviä parannuksia käyttöliittymän suorituskyvyn osalta.

6 POHDINTA

Opinnäytetyön päätavoite eli käyttökelpoisen lainanhallintajärjestelmän toteuttaminen asiakkaalle onnistui hyvin. Vaikka alkuperäisessä liian tiukaksi määritellyssä aikataulussa ei pysytty, toimivan järjestelmän toteuttaminen reilussa puolessa vuodessa vain yhden kehittäjän voimin oli silti positiivinen suoritus. Oppimistavoitteen osalta onnistuttiin erinomaisesti, ja opinnäytetyön aikana tutuksi tulleet teknologiat antoivat paljon uutta tietoa ja osaamista työelämässä hyödynnettäväksi.

Sovelluksen toteuttaminen web-pohjaisilla teknologioilla eikä esimerkiksi natiivina mobiilisovelluksena osoittautui toimivaksi ratkaisuksi. Rautatason API-kutsuja ei tarvittu, ja koska paikallista pysyvää dataa ei ole, selaimen suurin rajoite eli kovalevylle kirjoittamisen puuttuminen ei ollut ongelma. Lisäksi Meteor-sovellusalustan Cordova-integraatio mahdollistaa sovelluksen pakkaamisen mobiililaitteille, ja vaikka sovelluksen jatkokehityksessä Meteorista tultaisiin jossain vaiheessa luopumaan, on sovelluksen käyttöliittymä mahdollista portata mobiililaitteille React Native -kirjaston avulla.

JavaScript oli tullut jo pintapuolisesti tutuksi aikaisempien projektien parissa, ja opinnäytetyö oli hyvä tilaisuus syventää osaamista. Uusien ECMAScript-standardin ominaisuuksien käyttöä oli mielenkiintoista opetella, ja esimerkiksi moduulit, nuolifunktiot ja merkkijonotemplaatit mahdollistivat entistä ositellumman, tiiviimmän ja selkeämmän ohjelmakoodin kirjoittamisen. Kielen monipuolisuus mahdollistaa myös esimerkiksi funktionaalisen ohjelmointityylin käyttämisen, mitä tuli harjoiteltua Redux-kirjaston parissa.

Valtaosa projektin ongelmista ratkesi asentamalla kirjasto npm-paketinhallinnasta, ja niitä kertyikin yhteensä yli sata kappaletta, joista tosin vain noin puolet varsinaisia tuotantosovellukseen päätyneitä käyttökirjastoja. Kirjastojen suuri määrä selittänee osaltaan sovelluksen tuotantoversion 5,2 megatavun kokoa. Tämä saattaa aiheuttaa ongelmia varsinkin hitailla mobiiliyhteyksillä, ja sovelluksesta olisikin kenties järkevää pakata mobiililaitteille erilliset Android ja iOS -versiot, joka onnistuu Meteorin avulla ilman ylimääräistä työtä Cordova-integraation ansiosta. Mobiiliversion pakkaaminen ja toimittaminen sovelluskauppoihin ladattavaksi on yksi mahdollinen toimenpide jatkokehitystä ajatellen.

Kenties projektin suurimmaksi ongelmaksi osoittautui toimivan tietokantamallin rakentaminen. Järjestelmän data sisältää kohtuullisen paljon relaatioita, joiden esittäminen Mongon dokumenttimallin avulla oli haasteellista. Ensimmäisenä testattu normalisoitu tietokantamalli oli järjestelmän suorituskyvyn kannalta liian raskas. Lopulta päädyttiin kompromissiratkaisuun, jonka avulla suorituskyky saatiin nostettua halutulle tasolle. Käytetty ratkaisu toi omat ongelmansa datan yhtenäisyyden kannalta, koska dataa joudutaan monistamaan dokumenttien kesken ja Mongon kirjoitusoperaatiot ovat atomisia vain saman dokumentin sisällä. Sen vuoksi palvelimen ohjelmakoodiin jouduttiin tekemään korjaavia algoritmeja, jotka tarkistavat tietokannan tilan viikoittain ja suorittavat samalla korjaavia toimenpiteitä mahdollisten, joskin epätodennäköisten, ongelmatilanteiden ilmetessä.

Projektin aikana kävi selväksi, että Meteor sovellusalustan alun perin monoliittiseksi suunniteltu rakenne ja puutteellinen tuki testaukselle muodostuivat rasitteeksi. Meteor on perinteisesti ollut erittäin aloittelijaystävällinen kokonaisuus, ja ensimmäistä web-sovellusprojektia aloittavalle sitä voikin edelleen lämpimästi suositella. Meteor tarjoaa monia hyödyllisiä asioita, kuten reaktiivinen data ja käyttäjätunnusten hallinta, valmiiksi toteutettuna ja testattuna. Mikäli tarkoituksena on toteuttaa pienehkö sovellus nopeasti esimerkiksi esittelykäyttöön, on Meteor edelleen varteenotettava vaihtoehto.

Meteorin viimeaikaiset päivitykset ovat ajaneet sitä yhä modulaarisempaan suuntaan, mikä on yleisesti ottaen hyvä asia, mutta Meteorin itsensä kannalta saattaa jopa koitua sen kohtaloksi. React-tuen lisääminen on aiheuttanut Meteorin oman Blaze-käyttöliittymäkomponentin jäämisen taka-alalle, minkä lisäksi 1.3-versiossa lisätty npm-tuki on muutamia poikkeuksia lukuun ottamatta tehnyt Meteorin omasta Atmosphere-paketinhallintajärjestelmästä käytännössä tarpeettoman.

Vaikka opinnäytetyön alussa lähdettiin rakentamaan nimenomaan Meteor-sovellusta, loppuvaiheessa Meteorista on jäljellä käytännössä enää vain datakerros, käyttäjätunnusten hallinta ja rakennustyökalu. Kun ottaa vielä huomioon, että MDG suunnittelee seuraavassa 1.5-julkaisussa tuovansa Meteorin nykyisen datakerroksen rinnalle vaihtoehtoisen kaikkia sovelluskehyksiä ja tietokantoja tukevan Apollo-datakerroksen, on enää hyvin hankala perustella Meteorin tuomaa lisäkuormaa testauksen hankaloitumisen muo-

dossa. Edistyneempi kehittäjä tekee käyttäjätunnusten hallinnasta tarvittaessa oman toteutuksensa, ja rakennustyökalun tilalle on tarjolla paljon monipuolisempi ja tehokkaampi Webpack.

Sovelluksen jatkokehityksen ja ylläpidon kannalta tehtäväksi jää vielä ainakin kunnollisen testikattavuuden toteuttaminen ja lähdekoodin luettavuuden parantaminen refaktoroimalla. Mahdollisina lisäominaisuuksina tulee mieleen esimerkiksi henkilökunnan kyky merkitä lainalle noutopaikka, mahdollisuus kommentoida lainoja sekä reaaliaikainen chat-keskustelu. Lisäksi opinnäytetyön loppuvaiheessa sovellukselle luotiin vielä keskeneräinen käyttöopas, joka tulisi tehdä valmiiksi.

Lähdemateriaalin hankinnassa haastavinta oli löytää ajantasaista ja relevanttia kirjallisuutta. Opinnäytetyön teknologioista valtaosa oli enimmillään muutaman vuoden ikäisiä, ja esimerkiksi Redux-kirjasto oli luotu alle vuosi ennen projektin alkua. Kirjatoista löytyi lähinnä yleisimpiin aiheisiin kuten JavaScript liittyvää materiaalia. Lisäksi fyysiset teokset olivat auttamatta vanhentuneita, ja vaikka e-kirjastosta löytyi muutamia suhteellisen tuoreita painoksia, niidenkin sisältämä tieto oli osittain vanhentunutta jo kirjan julkaisuhetkellä. Suurin osa ajantasaisesta tiedosta löytyi teknologioiden dokumentaatiosta, tietotekniikka-aiheisista blogeista sekä YouTube-videopalvelun avulla löytyneistä konferenssivideoista. Luotettavina lähteinä pidettiin erityisesti teknologian alkuperäistä kehittäjää tai kehittäjäryhmän jäseniä, joita pyrittiin suosimaan aina kun mahdollista.

Tässä opinnäytetyössä kerättyä tietoa voidaan hyödyntää erityisesti JavaScript-pohjaisten web-sovellusten toteutuksessa, mutta valtaosaa tiedosta voidaan soveltaa yleisesti kaikenlaiseen web-sovelluskehitykseen. Osa käytetyistä menetelmistä kuten ohjelmakoodin linttaus ja jatkuva integraatio ovat käytössä yleisesti kaikkialla ohjelmistotuotannossa.

LÄHTEET

Abramov, D. 2015. Live React: Hot Reloading with Time Travel. Konferenssivideo. Katsottu 24.10.2016. https://www.youtube.com/watch?v=xsSnOQynTHs

Amazon Web Services. N.d. What is Continuous Integration? Tietosivu. Luettu 11.11.2016. https://aws.amazon.com/devops/continuous-integration/

Atlassian. N.d. Limitations of Bitbucket Pipelines. Käyttöopas. Luettu 7.11.2016. https://confluence.atlassian.com/bitbucket/limitations-of-bitbucket-pipelines-827106051.html

Atom. N.d. Atom Flight Manual. Käyttöopas. Luettu 25.12.2016. http://flight-manual.atom.io/

Branscombe, M. 2016. JavaScript Standard Moves to Yearly Release Schedule; Here is What's New for ES16. Uutisartikkeli. Luettu 14.11.2016. http://thenewstack.io/whats-new-es2016/

Brentjanderson. 2016. Vastaus Dockerin Github-ongelmaan nro. 23347. Luettu 24.10.2016. https://github.com/docker/docker/issues/23347

Carranza, P. 2016. GitLab Infrastructure Update. Blogikirjoitus. Luettu 8.11.2016. https://about.gitlab.com/2016/09/26/infrastructure-update/

Chacon, S., Straub, B. 2014. Pro Git. 2. painos. New York City: Apress.

Clifford, D. 2012. Google I/O 2012 - Breaking the JavaScript Speed Limit with V8. Konferenssivideo. Katsottu 22.11.2016. https://www.youtube.com/watch?v=UJPdhx5zTaw

Corey, P. 2016. The Missing Link in Meteor's Rate Limiter. Blogikirjoitus. Luettu 9.9.2016. http://www.east5th.co/blog/2016/05/16/the-missing-link-in-meteors-rate-limiter/

Crockford, D. 2008. JavaScript: The Good Parts. 1. Painos. Sebastopol, CA: O'Reilly Media.

Ćwirko, J. 2016. Live Chat App Using Meteor and the DDP Protocol. Blogikirjoitus. Luettu 14.11.2016. http://julian.io/live-chat-app-using-meteor-and-the-ddp-protocol/

Driessen, V. 2010. A successful Git branching model. Blogikirjoitus. Luettu 19.10.2016. http://nvie.com/posts/a-successful-git-branching-model/

ESLint. N.d. ESLint-dokumentaatio. Luettu 22.10.2016. http://eslint.org/docs/about/

Estesp. 2016. Storage Drivers in Docker: A Deep Dive. Blogikirjoitus. Luettu 24.10.2016. https://integratedcode.us/2016/08/30/storage-drivers-in-docker-a-deep-dive/

Fairbank, J. 2016. Managing Side Effects With Redux Saga: A Primer. Blogikirjoitus. Luettu 13.11.2016.
http://www.sigient.com/blog/managing-side-effects-with-redux-saga-a-primer-1

Felter, W., Ferreira, A., Rajamony, R., Rubio, J. 2014. An Updated Performance Comparison of Virtual Machines and Linux Containers. IBM Research. Luettu 23.10.2016.
http://domino.watson.ibm.com/library/CyberDig.nsf/1e4115aea78b6e7c85256b360066f0d4/0929052195dd819c85257d2300681e7b

Sijbrandij, S. 2016a. The future of SaaS hosted Git repository pricing. Blogikirjoitus. Luettu 7.11.2016. https://about.gitlab.com/2016/05/11/git-repository-pricing/

Sijbrandij, S. 2016b. The GitLab Master Plan. Webcast-video. Katsottu 7.11.2016.
https://www.youtube.com/watch?v=iV7mVGPXrxU

Goasguen, S. 2016. Introduction to Docker Compose Tool for Multi-Container Applications. Blogikirjoitus. Luettu 23.10.2016.
https://www.linux.com/learn/introduction-docker-compose-tool-multi-container-applications

Hakala, A. 2014. Korkeasti käytettävän pilvipalvelun toteuttaminen. Tietojenkäsittelyn koulutusohjelma. Tampereen ammattikorkeakoulu. Opinnäytetyö.

Li, D. 2016. Understanding Flux, Facebook's Application Architecture. Blogikirjoitus. Luettu 29.10.2016.
http://blog.brew.com.hk/understanding-flux/

Linode. 2016. Linode’s 13th Birthday – Gifts for All! Blogikirjoitus. Luettu 21.10.2016.
https://blog.linode.com/2016/06/16/linodes-13th-birthday-gifts-for-all/

Losh, S. 2010. My Extravagant Zsh Prompt. Blogikirjoitus. Luettu 18.10.2016.
http://stevelosh.com/blog/2010/02/my-extravagant-zsh-prompt/

Lund, K., Bak, L. 2013. Web Languages and VMs: Fast Code is Always in Fashion. (V8, Dart) - Google I/O 2013. Konferenssivideo. Katsottu 22.11.2016.
https://www.youtube.com/watch?v=huawCRlo9H4

Mathus, B. 2015. Docker Storage Driver: Don't Use Devicemapper. Blogikirjoitus. Luettu 24.10.2016.
https://batmat.net/2015/08/26/docker-storage-driver-dont-use-devicemapper/

Meteor. N.d. Meteor-dokumentaatio. Luettu 6.11.2016. https://guide.meteor.com

Minnick, C. 2016. The Real Benefits of the Virtual DOM in React.js. Blogikirjoitus. Luettu 24.10.2016.
https://www.accelebrate.com/blog/the-real-benefits-of-the-virtual-dom-in-react-js/

Modulecounts. 2016. Module Counts. Statistiikkasivu. Luettu 22.10.2016.
http://www.modulecounts.com/

MongoDB. N.d. MongoDB-dokumentaatio. Luettu 8.11.2016.
https://docs.mongodb.com/

Mozilla Developer Network. N.d. JavaScript Guide: Introduction. JavaScript-opas. Luettu 14.11.2016.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction

Ramos, M. 2016. Continuous Integration, Delivery, and Deployment with GitLab. Blogikirjoitus. Luettu 9.11.2016.
https://about.gitlab.com/2016/08/05/continuous-integration-delivery-and-deployment-with-gitlab/

Rapa, A. 2016. Relaatio- ja epärelaatiotietokantojen suorituskykyvertailu: MySQL ja MongoDB. Jyväskylän yliopisto. Tietotekniikan laitos. Pro gradu -tutkielma.

Rauschmayer, A. 2015. Deploying ECMAScript 6. Blogikirjoitus. Luettu 15.7.2016.
http://www.2ality.com/2015/04/deploying-es6.html

React. N.d. React-dokumentaatio. Luettu 25.11.2016.
https://facebook.github.io/react/docs

Redux. N.d. Redux-dokumentaatio. Luettu 29.10.2016. http://redux.js.org/

Roberts, P. 2014. Philip Roberts: What the heck is the event loop anyway? Konferenssivideo. Katsottu 20.11.2016.
https://www.youtube.com/watch?v=8aGhZQkoFbQ

Dahl, R. 2009. Ryan Dahl: Original Node.js presentation. Konferenssivideo. Katsottu 18.11.2016. https://www.youtube.com/watch?v=ztspvPYybIY

Dahl, R. 2011. Ryan Dahl – History of Node.js. Video. Katsottu 19.11.2016.
https://www.youtube.com/watch?v=SAc0vQCC6UQ

Severance, C. 2012. JavaScript: Designing a Language in 10 Days. Computer 45 (2), 7-8.

Stubailo, S. 2015. Optimistic UI with Meteor. Blogikirjoitus. Luettu 13.10.2016.
http://info.meteor.com/blog/optimistic-ui-with-meteor-latency-compensation

Voss, L. 2014. npm and front-end packaging. Blogikirjoitus. Luettu 18.11.2016.
http://blog.npmjs.org/post/101775448305/npm-and-front-end-packaging

Ward, R. 2014. Getting Started with ES6 Modules. Blogikirjoitus. Luettu 22.11.2016.
https://eviltrout.com/2014/05/03/getting-started-with-es6.html

Yu, A. 2013. Why web beginners should start with Meteor. Blogikirjoitus. Luettu 24.7.2016. http://info.meteor.com/blog/why-web-beginners-should-start-with-meteor

LIITTEET

Liite 1. Linux-komentoja (Ubuntu 16.04)

Luvussa 3.1.1 esitellyn GitFlow (AVH Edition) -komentorivityökalun käyttö.

```
# Asennus
sudo apt install git-flow
cd ~/projektin-kansio
git flow init

# Luo uusi toimintohaara
git flow feature start newFeatureName
# Commitoi uusi toiminto
git commit newFeatureFiles.js -m ‘Added a new feature
# Sulauta valmis toimintohaara
git flow feature finish newFeatureName

# Luo uusi julkaisuhaara (versio X.Y.Z)
git flow release start X.Y.Z
# Commitoi mahdolliset bugikorjaukset ja lopuksi päivitä versionume-
rot
git commit package.json CHANGELOG.md -m 'Bumped versions for release
X.Y.Z'
# Sulauta valmis julkaisuhaara
git flow release finish X.Y.Z
```

Liite 2. Projektissa käytetyt JavaScript-kirjastot

1 (2)

Liitteen sisältö on osa npm-paketinhallinnan package.json-tiedostoa. Se määrittelee npm-paketinhallinnalla asennettavat kirjastot ja niiden versiot.

```
"dependencies": {
    "awesome-bootstrap-checkbox": "^0.3.7",
    "babel-polyfill": "6.x.x",
    "bcrypt": "^0.8.7",
    "bunyan": "^1.8.1",
    "bunyan-format": "^0.2.1",
    "chart.js": "^2.2.0",
    "events": "^1.1.0",
    "font-awesome": "^4.6.3",
    "griddle-react": "^0.7.0",
    "history": "^2.0.0",
    "i18next": "^3.1.0",
    "i18next-node-fs-backend": "^0.1.2",
    "i18next-sync-fs-backend": "^0.1.0",
    "i18next-xhr-backend": "^1.0.0",
    "immutable": "^3.7.6",
    "immutable-props": "^1.1.0",
    "later": "^1.2.0",
    "lodash": "^4.9.0",
    "moment": "^2.14.1",
    "moment-business": "^3.0.0",
    "react": "15.3.x",
    "react-addons-css-transition-group": "^15.0.0",
    "react-bootstrap": "^0.30.2",
    "react-chartjs-2": "^1.1.5",
    "react-datepicker": "^0.34.0",
    "react-dom": "15.x.x",
    "react-html-email": "^1.1.2",
    "react-i18next": "^1.4.1",
    "react-moment-proptypes": "^1.2.0",
    "react-redux": "^4.4.0",
    "react-router": "^2.0.0",
    "react-router-bootstrap": "^0.23.0",
    "react-router-redux": "^4.0.0",
    "react-widgets": "^3.3.2",
    "redux": "^3.3.1",
    "redux-actions": "^1.0.0",
    "redux-auth-wrapper": "^0.9.0",
    "redux-form": "^6.0.1",
    "redux-immutablejs": "0.0.8",
    "redux-logger": "^2.6.0",
    "redux-notifications": "^3.0.0",
    "redux-saga": "0.13.x",
    "redux-storage": "^4.0.0",
    "redux-storage-decorator-filter": "^1.1.1",
    "redux-storage-decorator-immutablejs": "^1.0.0",
    "redux-storage-merger-immutablejs": "^1.0.1"
  },
```

(jatkuu)

2 (2)

```
  "devDependencies": {
    "babel": "^6.3.26",
    "babel-core": "6.x.x",
    "babel-eslint": "^7.0.0",
    "babel-loader": "^6.2.0",
    "babel-plugin-__coverage__": "^11.0.0",
    "babel-plugin-add-module-exports": "^0.2.0",
    "babel-plugin-react-transform": "^2.0.0",
    "babel-plugin-transform-decorators-legacy": "^1.3.2",
    "babel-preset-airbnb": "^2.0.0",
    "babel-preset-es2015": "6.x.x",
    "babel-preset-react": "6.x.x",
    "babel-preset-stage-0": "^6.3.13",
    "babel-root-import": "^4.0.0",
    "chai": "^3.5.0",
    "chai-enzyme": "^0.6.0",
    "cheerio": "^0.22.0",
    "css-loader": "^0.26.0",
    "enzyme": "^2.1.0",
    "eslint": "^3.0.0",
    "eslint-config-airbnb": "^13.0.0",
    "eslint-import-resolver-webpack": "^0.7.0",
    "eslint-plugin-import": "^2.0.0",
    "eslint-plugin-jsx-a11y": "^2.0.0",
    "eslint-plugin-meteor": "^4.0.0",
    "eslint-plugin-react": "^6.0.0",
    "expose-loader": "^0.7.1",
    "extract-text-webpack-plugin": "^1.0.0",
    "faker": "^3.1.0",
    "file-loader": "^0.9.0",
    "html-loader": "^0.4.0",
    "jsdom": "^9.0.0",
    "json-loader": "^0.5.4",
    "markdown-loader": "^0.1.7",
    "mocha": "3.x.x",
    "node-sass": "^3.4.2",
    "nyc": "^9.0.0",
    "react-transform-catch-errors": "^1.0.0",
    "react-transform-hmr": "^1.0.1",
    "redbox-react": "^1.2.0",
    "redux-devtools": "^3.1.1",
    "redux-devtools-diff-monitor": "^5.0.0",
    "redux-devtools-dispatch": "^2.0.1",
    "redux-devtools-dock-monitor": "^1.1.0",
    "redux-devtools-filter-actions": "^1.1.0",
    "redux-devtools-filterable-log-monitor": "^0.6.1",
    "redux-devtools-log-monitor": "^1.0.5",
    "redux-mock-store": "^1.0.2",
    "redux-slider-monitor": "^1.0.2",
    "source-map-support": "0.4.3",
    "sass-loader": "^4.0.0",
    "sinon": "1.17.x",
    "style-loader": "^0.13.0",
    "testdouble": "^1.4.3",
    "url-loader": "^0.5.7",
    "webpack": "^1.13.0",
    "webpack-hot-middleware": "^2.4.1"
  }
```