jamk.fi

MDO-järjestelmän kehittäminen

Case: Versine Oy

Eerik Anttila

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

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Anttila, Eerik	Opinnäytetyö, AMK	Marraskuu 2016
	Sivumäärä 62	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi

MDO-järjestelmän kehittäminen
Case: Versine Oy

Tutkinto-ohjelma
Tietojenkäsittelyn koulutusohjelma

Työn ohjaaja(t)
Tommi Tuikka

Toimeksiantaja(t)
Kari Aho

Tiivistelmä

Työelämän automatisoituessa tarvitaan yhä enemmän automatisointia helpottavia sovelluksia. Yritykset vaativat tänä päivänä yhä enemmän omiin tarpeisiinsa kehitettyjä sovelluksia, joissa on huomioitu yrityskohtaisesti yrityksen sisäiset käytänteet. Tähän liittyy paljon tutkimustyötä vaativia ongelmia.

Tutkimuksen viitekehys oli tapaustutkimus. Itse tutkija sekä toteuttaja eli ohjelmoija olivat sama henkilö. Toimeksiantajana tälle tutkimus- ja kehittämisprojektille oli IT-yritys Versine Oy, joka halusi laajentaa MDO-nimistä sovellustaan asiakkaiden toiveiden mukaan.

Yrityksen kehittämä MDO on pilvi- ja mobiilipohjainen sovellus infrasktuurin, dokumentaation ja työnkulun seurantaan. Sovellukseen toteutettiin ominaisuus, jolla sovelluksen käyttäjät eli yritykset pystyivät lisäämään työntekijöilleen muistutuksia liittyen määrättyyn kohteeseen. Kohde saattoi olla esimerkiksi teleliikennemasto tai muu etenkin teleoperaattorien alihankkijoiden työnkuvaan oleellisesti kuuluva paikka.

Kehitystyössä seurattiin selviä ja jo kehitystyössä sovellettuja suunnittelumalleja ja -käytänteitä. Ominaisuus pidettiin linjassa jo aiemmin toteutettujen ratkaisujen kanssa niin arkkitehtuuriltaan kuin käyttöliittymän toiminnaltaankin. Joissakin tapauksissa aiemmin toteutetuista tavoista saatettiin poiketa, esimerkiksi jos haluttiin tehdä tietyt asiat aiempaa oikeaoppisemmin ohjelmoinnin näkökulmasta.

Pääasialliset tutkimus- ja kehittämistyön aikana käytetyt tekniikat ovat mm. Backbone.js (JavaScript-sovelluskehys) sovelluksen Front endissä eli asiakaspuolella, PHP sovelluksen Back endissä eli palvelinpuolella sekä SQL (MariaDB) tietokannan toiminnassa. Tutkimus- ja kehityös tuloksena uudet toiminallisuudet ja lopulta toteutettu ominaisuus kokonaisuudessaan siirrettiin sovelluksen tuotantoympäristöön asiakkaiden käytettäväksi.

Avainsanat (asiasanat)
Design pattern, PHP, HTML5, Web development

Muut tiedot

jamk.fi

Description

Author(s) Anttila, Eerik	Type of publication Bachelor's thesis	Date November 2016
		Language of publication: Finnish
	Number of pages 62	Permission for web publication: x

Title of publication
Development of the MDO system
Case: Versine Oy

Degree programme
Business Information Systems

Supervisor(s)
Tuikka, Tommi

Assigned by
Aho, Kari

Abstract

Today the working life is getting more and more automatized. This will create a continuously growing need for different applications, and their features need to respond to the clients', such as companies' and their employees', needs. Especially, company-specific applications and features will get more and more important. Problems like these are worth researching and developing.

The chosen method for the research was an action research in which the researcher and the programmer were the same person. The client for this action research was an IT-company Versine Oy and their application called MDO.

The MDO application is a cloud and mobile based tool for infrastructure, documentation and work force management. During this action research this application received new a feature for answering to its clients' demands. This feature was made to mimic adding reminders for the employee's workflow. Every new reminder was added under a key point in the application, called "sites".

A site can be everything from a radio mast or tower to a telecommunications closet.
These reminders will ease the maintenance of these sites by reminding the user(s) or usergroup(s) of the certain tasks added for the sites.

The key technologies used in this thesis will include Backbone.js (JavaScript framework) for the application's Front end, PHP for the application's Back end and SQL (MariaDB) for the database functionalities. These completed features were merged in the master branch of the application and then deployed in the production, where the final users (clients) are able to use the new features like any other feature.

Keywords/tags (subjects)
Design pattern, PHP, HTML5, Web development

Miscellaneous

Sisältö

Kuviot

Termit ja lyhenteet

AngularJS	Googlen kehittämä JavaScript-pohjainen ohjelmistokehys
RDBMS	Relational database management system, relaatiotietokantatyyppi
Backbone.js	JavaScript-kehysympäristö käyttäjäpuolen kehitykseen
Back end	Nimitys palvelinpuolesta
Collection	Ohjelmistoarkkitehtuurissa käytettävä nimitys "kokoelmasta".
DOM	Document Object Model, eli dokumenttioliomalli, jonka tehtävä on määrittää miten sivun eri oliot kuten HTML-elementit määräytyvät sivulla ja miten niihin voi viitata ja näin vaikuttaa
Framework	Kehysympäristö/sovelluskehys
Front end	Nimitys selain-/käyttäjäpuolesta
Grunt	JavaScript-pohjainen tehtävänsuorittaja
JavaScript	Pääasiassa web-ympäristössä käytettävä dynaaminen ohjelmointikieli
MDO	Kehitystyössä kehitettävä sovellus
Model	Ohjelmistoarkkitehtuurissa käytettävä nimitys "mallista"
MVC	Model-View-Controller, malli–näkymä–käsittelijä-arkkitehtuurimalli
MVP	Model-View-Presenter, malli-näkymä-esittäjä-arkkitehtuurimalli
PHP	Palvelinpuolen ohjelmointikieli
React	Facebookin kehittämä JavaScript-pohjainen ohjelmistokehys
SQL	Structured Query Language, kyselykieli relaatiotietokannoille
Vagrant	Virtuaalinen ympäristö erilaisten ajoympäristöjen rakentamiseen
View	Ohjelmistoarkkitehtuurissa käytettävä nimitys "näkymästä"

1 Johdanto/kuvaus

Viime aikoina mediassa on puhuttu yhä enemmän työelämän automatisoitumisesta isommassa kuvassa käytännössä kaikilla työelämän aloilla. Yksi automatisoimista vaativa asia työelämässä on työnkulun seuranta, jolla työntekijöiden työnkulkua voidaan hallita, mutta ennen kaikkea helpottaa.

Eräs työnkulun seurannasta huomattavasti hyötyvä ammattikunta on urakoitsijat ja aliurakoitsijat. Näihin kuuluu mm. teleliikenteen huoltohenkilöstö. Heitä varten IT-yritys Versine Oy kehitti johtamis- ja dokumentointiratkaisun urakoitsijoille, rakennuttajille, kiinteistönhuoltoon, isännöitsijöille ja verkko-operaattoreille. Käytännössä tämä ratkaisu on erilaisilla päätelaitteilla toimiva sovellus nimeltään MDO.

Tässä työssä edellä mainittua sovellusta laajennettiin uudella ominaisuudella, jotta työnkulun seuranta ja tehtävien yleinen hallinta olisi entistä toimivampaa. Ominaisuus jäljittelee muistutuksia, joiden tarkoitus MDO-sovelluksessa on helpottaa tulevia huoltotehtäviä.

Kehitystyössä käytetään jo aiemmin sovelluksessa käytettäviä tekniikoita ja sovellusmalleja, eli pidetään uusi toiminallisuus linjassa vanhempien toteutusten kanssa. Yhtenäinen linja tulee näkymään sekä ohjelmistoarkkitehtuurissa kuin sovelluksen käyttöliittymässäkin. Työssä käytettäviä tekniikoita on mm. HTML5/JavaScript, PHP ja SQL. Lopullinen toteutus tullaan julkaisemaan MDO-sovelluksen tuotantoversiossa lopullisille käyttäjille.

Tutkimusosuudessa myös tutkitaan, minkälaista hyötyä React-kirjastosta voisi olla yhdessä Backbone.js-tekniikan kanssa käytettynä. Koska Backbone.js on yksi kehitystyön tekniikoista, tästä tutkimuksesta on hyötyä myös MDO-sovelluksen näkökulmasta.

2 Tutkimusasetelma

2.1 Tavoitteet ja rajaukset sekä tutkimuskysymykset

Tämä opinnäytetyö on Versine Oy:lle tehtävä kehittämistyö. Versine Oy:n on kehittänyt teleliikenneyritykselle ja niiden alihankkijoille MDO-järjestelmää, jonka yhdeksi ominaisuudeksi tämä kehittämistyö tehdään. MDO-järjestelmän tarkoitus on olla joustava johtamis- ja dokumentointiratkaisu, joka tukee rakentamisen, kiinteistöjen sekä laitteiden koko elinkaaren hallintaa aina suunnittelusta ylläpitoon ja huoltoon asti.

Pilvi- ja mobiilipohjainen palvelu mahdollistaa töiden reaaliaikaisen johtamisen, dokumentoinnin suoraan kentältä sekä turvallisen tiedon jakamisen asiakkaiden ja toimittajien kesken. MDO on käytössä urakoitsijoilla, rakennuttajilla, isännöitsijöillä, kiinteistönhuollossa ja verkko-operaattoreilla.

Tässä kehittämistyössä tehdään MDO-sovellukseen ominaisuus muistutuksille, joita voidaan luoda jokaiselle sovelluksen niin sanotulle kohteelle. Sovelluksessa "kohde" on esimerkiksi fyysisessä sijainnissa sijaitseva teleliikennemasto tai täysin abstrakti-asia, riippuen täysin halutunlaisesta käyttötavasta. Muistutuksia voi lisätä jokainen käyttäjä, jolla on käyttöoikeudet kohteelle, jolle muistutus lisätään. Muistutuksia voi lisätä yhdelle tai useammalle käyttäjälle. Muistutuksille tehdään oma hallintasivu kohde-osion välilehdeksi, missä muistutuksia voi lisätä, muokata ja poistaa. Muistutuksille määrätään muistutusaika, minkä tullessa ajankohtaiseksi muistutuksista lähtee ilmoitus määrätyille käyttäjille. Muistutusaika voi tapahtua kerran, päivittäin, viikoittain, kuukausittain tai vuosittain. Ominaisuuden valmistuessa se myös näkyy kohde-osion kalenteri-välilehdessä oikein.

Käytetyt tekniikat ovat palvelinpuolella PHP ja SQL (MariaDB) ja käyttöliittymän puolella Backbone.js (HTML5). Yrityksen käyttämä PHP-palvelin ei ole rakennettu minkään varsinaisen PHP-sovelluskehyksen päälle, mikä tuo omat haasteensa siihen perehtymisen kanssa. Asiakaspuolen toteutus on hyvin suuri Backbone.js-sovellus, mikä on jaettu Model-, View- ja Controller-osiin.

Opinnäytetyössä on tarkoitus käyttää Backbone.js:ää sen periaatteiden mukaisesti ja palvelinpuolta tavalla, miten yrityksessä sitä on suunniteltu käytettäväksi. Kehitystyössä oli sekä asiakas- että palvelinpuoli lokaalissa ympäristössä, eli omalla tietokoneella. Opinnäytetyön tutkimuskysymykset ovat seuraavat:

Miten ominaisuus voidaan kehittää Full stack -mittakaavassa?

- Miten tämä tehdään tietokantaan palvelinpuolelle sekä asiakaspuolelle?

Miten pidetään toteutus linjassa sovelluksen aiempien toteutusten kanssa?

- Miten arkkitehtuurinen toteutus saadaan pysymään linjassa aiempien toteutusten kanssa?
- Miten käyttöliittymän toiminallisuus saadaan pysymään linjassa muiden sovelluksen toimintojen kanssa?

Voidaanko Backbone.js:n View eli näkymä korvata React-kirjastolla?

- Tuoko React-kirjaston käyttö lisäetua renderöintinopeuteen sovelluksessa?
- Mitä muutoksia tämä toisi sovelluksen arkkitehtuuriin?

2.2 Tutkimusmenetelmät

Tämän opinnäytetyön tutkimusmenetelmien pääroolissa oli kehittämistutkimus. Kehittämistutkimus on yksi kehittämistyön julkituomismuoto ja siinä yhdistyvätkin kehittäminen ja tutkimus. (Kananen 2015, 33) Tarve muutokselle parempaan suuntaan on se lähtökohta, mistä kehittämistyö lähtee. (Barabi & Squire 2004) Koska muutoksia halutaan vielä parempaan suuntaan, eivät kaikki muutokset ole kehittämistutkimusta. Kehittämistutkimus vaatiikin tutkimuksellista otetta ja tutkimusosion, jotta tutkimustyöstä päästäisiin lähemmäs oikeaa kehittämistutkimusta. Tutkimustyön sisältäessä ongelman poistamisen, on kehittämistyö laajempi tutkimustyö kuin perinteinen laadullinen ja määrällinen tutkimus. (Kananen 2015, 40)

Kehittämistutkimuksessa, josta opinnäytetyö tehdään, aloitetaan johdannosta, jatketaan menetelmiin, jonka jälkeen tulee tutkimus ja tulokset. Lopussa on pohdinta. Kehittämistutkimuksen raportointimalli onkin sama kuin opinnäytetyön ja väitöskirjan yleinen raportointimalli. (Kananen 2015, 17)

Johdanto kuvaa tutkimusaihetta yleisellä tasolla. Johdannossa valotetaan tutkimuksen taustoja ja tarpeellisuutta. Tutkimusasetelma tulee johdannon jälkeen, missä

tuodaan esiin tutkimusongelma, rajauksia sekä ongelmasta johdettuja tutkimuskysymyksiä. Tutkimusasetelmassa kerrotaan myös aineistonkeruutavasta ja tutkimusotteesta. Johdannon ja tutkimusasetelman jälkeen tuodaan ilmi teoreettinen viitekehys eli tietoperusta. Tähän teoreettiseen viitekehykseen esitellään ongelman kannalta oleellista aineistoa, kuten tutkimuksia ja kirjallisuutta sekä työssä käytettävät käsitteet ja mittarit. Neljäs kohta on tutkimustulokset ja niiden analysointi. Tuloksista tehdään tulkinta, mutta varsinaiset johtopäätökset tehdään tutkimustulokset-luvun jälkeen. Johtopäätöksissä pohditaan mitä tutkimustulokset merkitsevät ongelman kannalta. Viimeinen luku on pohdinta, missä pohditaan oman tutkimuksen ja omien tutkimustulosten suhdetta teoreettiseen viitekehykseen. Näiden jälkeen esitetään mahdolliset jatkotutkimusaiheet sekä tehdään työn luotettavuusarviointi. (Kananen 2015, 16)

Nykytilan kartoituksessa mietittiin kehitystyön pääroolissa olevan MDO-sovelluksen nykyisiä ominaisuuksia ja mitä siihen oltiin suunniteltu tehtävän. Ensimmäinen tutkimustyön ongelma oli vielä toteuttamaton ominaisuus, joka sovellukseen haluttiin. Ongelmaan vastattiin tutkimuskysymyksillä, jotka kysyivät tämän ongelman poistamiseen oleellisesti liittyvien osa-alueiden toteuttamisesta. Ensimmäiset, samaan aiheeseen liittyvät tutkimuskysymykset olivat:

Miten ominaisuus voidaan kehittää Full stack -mittakaavassa?

- Miten tämä tehdään tietokantaan palvelinpuolelle sekä asiakaspuolelle?

Miten pidetään toteutus linjassa sovelluksen aiempien toteutusten kanssa?

- Miten arkkitehtuurinen toteutus saadaan pysymään linjassa aiempien toteutusten kanssa?
- Miten käyttöliittymän toiminallisuus saadaan pysymään linjassa muiden sovelluksen toimintojen kanssa?

Tutkimusosan tutkimusongelma on puuttuva ominaisuus, jonka tutkimuskysymyksenä kysytään sitä, miten kehitystyö tehdään asiakas- ja palvelinpuolelle sekä sitä, miten nämä asiat tehdään siten, että ne pysyvät sovelluksen aiempien toteutusten kanssa linjassa. Nämä raamit on toteutettava sekä sovellusarkkitehtuurissa että käyttöliittymässä. Tämän ongelman määrittelyssä vaihtoehtona ei ollut käytännössä muuta sellaista vaihtoehtoa, mikä olisi myös ratkaissut ongelman. Ratkaisu oli toteut-

taa tämä uusi ominaisuus, kun jo nykytilan kartoitus antoi viitteet siitä, että se pystyttiin toteuttamaan. Samalla tutkimusongelman tavoite oli ongelman poistaminen toteuttamalla ominaisuus. Materiaalin hankkimisessa tutkimusotteena on kvalitatiivinen tutkimus. Kvalitatiivinen tutkimus on toimivin keino hankkia materiaalia tai aineistoa näiden tutkimuskysymysten ratkaisemiseksi.

Tutkimusaineistona tässä toimii käytännössä sovellus itsessään ja sen toiminallisuuden tutkiminen käyttöliittymässä. Tähän perehtymällä päästään käyttöliittymässä yhteisen linjan säilyttämisen jäljille. Toinen osa aineistoa on ohjelmistollinen osuus, eli palvelin- sekä asiakaspuolen järjestelmän tutkiminen ja sisäistäminen jatkokehitystä ajatellen. Jatkokehityksellä tarkoitetaan kehitystyössä toteutettavan ominaisuuden toteuttamista.

Luotettavuuden varmistaa muiden kehittäjien palaute, jonka toivotaan vastaavan tutkimuskysymyksissä siihen, onko toteutus linjassa sovelluksen muihin osa-alueisiin niin sovellusarkkitehtuurillisesti kuin käyttöliittymässäkin. Sama palaute vastaa myös siihen, onko toteutus toimiva, eli onko ominaisuus kehitetty toimivasti niin asiakaspäähän, palvelinpuolella kuin tietokantaan. Tutkimusotteena tälle tutkimuskysymykselle on laadullinen tutkimus, kun varsinaisia lukuja ei tuloksina saada.

Aihe rajataan jo käytettyihin tekniikkoihin, eikä uusia tekniikoita valita käytettäväksi. Tämä haittaisi lähtökohtaisesti jo tutkimuskysymystä, missä halutaan säilyttää yhtenäinen linja. Uuden tekniikan valitseminen haittaisi tätä linjaa todennäköisesti sekä käyttöliittymässä, mutta etenkin sovellusarkkitehtuurissa. Sovellusarkkitehtuurin yhtenäisyys kärsisi sekä asiakas- kuin palvelinpuolellakin, jolloin tutkimuskysymyksen yhtenäisestä linjasta voisi katsoa epäonnistuneen.

Toinen tutkimustyön ongelmista oli kysymys siitä, olisiko vaihtoehtoisen piirtotekniikan valitseminen sovelluksen renderöintinopeutta parantava valinta. Kysymyksessä kysyttiin myös sitä, mitä muutoksia vaihtoehtoisen tekniikan valinta sovelluksen arkkitehtuuriin toisi. Tutkimuskysymykset olivat:

Voidaanko Backbone.js:n View eli näkymä korvata React-kirjastolla?

- Tuoko React-kirjaston käyttö lisäetua renderöintinopeuteen sovelluksessa?
- Mitä muutoksia tämä toisi sovelluksen arkkitehtuuriin?

Toinen alakysymys kysyy, voisiko React-näkymäkirjastosta olla hyötyä sovelluksen renderöintinopeudelle. Tähän alakysymykseen ratkaisu on tehdä testi, joka asettaa testattavat tekniikat samalle lähtöviivalle ja analysoi lopulta tulokset sekä antaa johtopäätökset. Tässä tutkimustyössä tuloksia on helpompi varmentaa, sillä myös konkreettisia testausaikoja saadaan. Tutkimusotteena voidaankin käyttää määrällistä tutkimusta, sillä testi tuottaa numeraalista dataa. Testin reliabiliteetti on suuressa roolissa etenkin siksi, että etuja antavia tai haittaavia muutoksia tekniikoilla testiympäristössä ei tule ilmetä. Tässä tutkimuskysymyksessä rajaus tehdään valitsemalla nykyinen tekniikka, potentiaalinen tekniikka, sekä tekniikka, jota ei tulla valitsemaan mutta mikä otetaan laajemman tutkimuksellisen näkökulman vuoksi käyttöön.

3 Työn taustaa

3.1 Vaatimukset ja aikataulu

Toimeksiantajan haluamalla ominaisuudella oli selvät vaatimukset. Kokonaiskuva vaatimuksista oli selvä, ja vain joissakin asioissa poikettiin suunnitelmasta sekä käyttöliittymän että palvelinpuolen ja tietokannan arkkitehtuurin osalta. Käytännössä pakollisten, toiminnan mahdollistavien ominaisuuksien jälkeen saatettiin huomata, että jossain kohtaa voisi vielä lisätä käytettävyyttä esimerkiksi selventämällä sen esitysasua.

Yleinen ohje oli, että ominaisuuden oli toimittava linjassa sovelluksen muiden ominaisuuksien ja komponenttien kanssa. Tämä toi omat haasteensa etenkin, jos komponentin käyttäminen ei muuten ollut selvää. Nämäkin haasteet tietysti toivat oman harjaannuttavan aspektinsa, mikä oli omiaan kartoittamaan omaa osaamista. Vapaasti päätettävien tekniikoiden ja toteutustapojen valitseminen olisi todennäköisesti tehnyt työstä huomattavasti helpomman toteuttaa, mutta koska kehittämistyö oli vain yksi osa sovellusta ja näin määritelty vaatimukseksi, ei tällainen vapaampi toteuttamistapa ollut mahdollinen.

Yleisesti ottaen vaatimukset selvisivät vasta toteutuksen aikana, eikä niinkään etukäteen, ennen kehitystyön aloittamista. Aikatauluna toimi käytännössä oma harjoitteluaikani yrityksessä. Toteutin ominaisuutta aina silloin, kun muuta kiireellisempää kehitystyötä tai muiden toteutuksien katselmointia tai testausta ei tarvinnut tehdä. Eri

vaiheiden toteutus oli kuin mitä tahansa taskeja eli tehtäviä backlogin näkökulmasta katsottuna.

3.2 Projektin- ja versionhallinta

Projektinhallinnassa käytimme muun muassa kehittämämme websovelluksen tuotantoversiota eli sitä, mitä "oikeatkin" asiakkaat käyttävät. Sinne oli kirjattu asiakaspuolella eli käyttöliittymässä, palvelinpuolella sekä tietokantaan toteutettavat tehtävät ominaisuuden toimintakuntoiseksi saamiseksi. Sovelluksen käyttäminen projektinhallintavälineenä tässä oli hyvää testiä sille, minkälaiseen käyttöön sitä voitiin tarvittaessa käyttää. Varsinainen niin sanottu viitekehys projektinhallinnassa oli ketterä ohjelmistokehitys eli Agile Scrum. Ketterään ohjelmistokehitykseen ajanjaksojen, niin sanottujen sprinttien välillä pystyin demoamaan valmiita vaiheita siinä samalla, kun esittelimme muita kyseisen sprintin aikaansaannoksia asiakkaalle.

Versionhallinnassa käytimme Git-työkalua, jonka varastot olivat Bitbucket.com:ssa. Täällä varastot oli jaettu palvelinpuolen sekä asiakaspuolen varastoihin, ja joita kumpiakin käytin riippuen ominaisuuden vaiheesta. Eri vaiheiden hyväksyntä tapahtui luomalla ns. Pull request, eli testaus- ja katselmointipyyntö yhdelle tai useammalle tiimin jäsenelle.

3.3 Vaiheet toteutuksessa

Projektin toteutuksessa edettiin vaiheittain. Kun tietty kohta oli valmis, oli usein sen hyödyntäminenkin mahdollista tavalla tai toisella. Esimerkiksi kun kehitettävien muistutusten ei-toistuvat, eli käytännössä vain kerran hälyttävät muistutukset olivat valmiit toteutukseltaan, voitiin ne siirtää dev- eli kehitys-/testauspalvelimelle sekä myöhemmin tuotantoon. Tässä vaiheessa niitä voitiin jo hyödyntää ainakin periaatteessa.

Esimerkiksi eräässä tilanteessa seuraava vaihe oli tehdä toteutus useammin kuin kerran, eli päivittäin, viikoittain, kuukausittain tai vuosittain toistuville muistutuksille. Tässä ei tarvinnut tehdä muutoksia enää asiakaspuolelle, mutta sekä tietokantatau-

lua että palvelinpuolta oli muutettava tiettyjen ominaisuuksien säilyttämiseksi samalla kun uudet toiminnot haluttiin tulevan voimaan. Tämä oli selvästi yksi niistä vaiheista, kun tuli vastaan jotain, mitä ei alun perin otettu huomioon.

Tietyn vaiheen valmistumisen jälkeen ne esiteltiin Scrum-kehitykselle tutuissa demoissa. Demoissa valmistunut työ esiteltiin asiakkaalle (tai sitä yrityksen sisällä edustavalle henkilöstölle kuten ns. myyntiporukalle). Asiakas tai asiakasta edustava taho kertokin useasti niistä seikoista, mihin haluttiin tai ainakin toivottiin parannuksia. Parannukset liittyivätkin usein sovelluksen käytön selkeyttämiseen asiakkaan näkökulmasta.

4 Tekniikat

Luvussa kuvataan kehitystyössä käytettyjä tekniikoita. Tekniikat esitellään etenkin niiden yleismaailmallisesta näkökulmasta sekä siitä sellaisella lähestymistavalla, mikä on oleellinen juuri tälle kehittämistyölle.

4.1 JavaScript

JavaScript (standardisoidulta nimeltään EcmaScript) on tänä päivänä hyvin oleellinen ohjelmointikieli sen ollessa käytössä niin käyttöliittymäpuolella (HTML5:n JS-frameworkit), kuin nykyään palvelinpuolellakin (Node.js).

JavaScriptin standardisoinnista vastaa Ecma International -standardisointijärjestö. JavaScriptin hyvinä voidaan pitää funktioita, heikkoa tyypitystä, dynaamisia olioita sekä olion luomiseen liittyvien arvojen antamisen "ilmeikkyyttä". JavaScriptin heikkouksien syynä JavaScript: The Good Parts -kirjan kirjoittaja, Douglas Crockford, pitää lähinnä selaimen esittämän dokumenttioliomallin eli DOM:n (eng. Document Object Model) heikkoa toteutusta, jonka kanssa mikä tahansa ohjelmointikieli olisi vaikeuksissa. (Crockford 2008.)

JavaScript on niin sanottu sanottuna skriptikielenä pidetty dynaaminen kieli, jota ei käännetä tavukieleksi eli ns. konekieleksi vaan sen suorittaminen annetaan käyttäjän selaimen tehtäväksi.

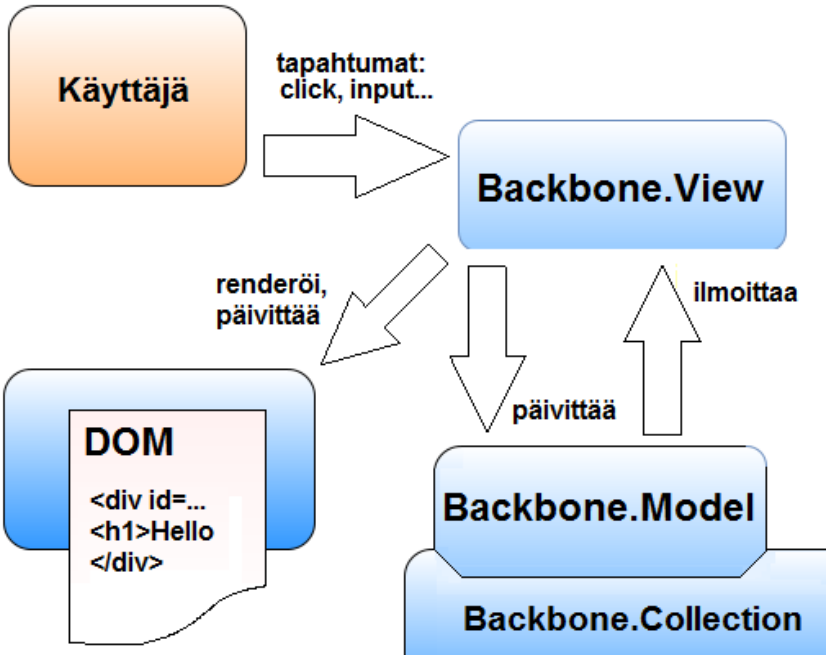
JavaScriptin minifioiminen on tapa, jolla JavaScriptin lähettämisestä aiheuttavaa suorituskyvyn laskua voidaan vähentää. Minifioimalla JavaScript sen pakkauskoko pienenee ja sitä voidaan suorittaakin nopeammin. Kääntöpuolena tälle siitä tulee ihmiselle hankalasti luettava, mutta minifioitua JavaScript-koodia ei lähtökohtaisesti ole tarvettakaan lukea. (Seibel 2009)

4.1.1 Backbone.js

Backbone.js on yksi vanhimmista vieläkin käytetyistä websovelluksen toiminnan mahdollistamista JavaScript-frameworkeista. Frameworkit eli kehysympäristöt mahdollistavan websovellusten toiminnan tarjoamalle näille websovelluksen jakamisen eri osiin, kuten Backbone.js:n tapauksessa malleihin (Model), kokoelmiin (Collection) sekä näkymiin (View). Mallit ja kokoelmat voivat molemmat ottaa yhteyden RESTful API-rajapintoihin. (Sayar 2013.)

Backbone.js:ää pidetään usein yhtenä vaihtoehtona MVC-frameworkille kuten Angular.js:lle, vaikkakin teknillisesti, tarkemmin tarkasteltuna Backbone.js on MVP- tai MVVM-framework. Täydellisesti Backbone.js:ää on mahdoton luokitella. (Bailey 2011.)

Backbone.js pitää useita funktioita View:ssä eli näkymässä, kun useassa MVC:ssä idea on pitää kaikki toiminallisuus poissa näkymästä. Backbonen Viewin tarkoitus on auttaa asynkronisesti toimivan käyttöliittymän rakentamisessa ja tiedon esittämisessä selaimessa. Vaikka Backbone.js pitää toiminallisuutta Viewissä, on sillä kuitenkin myös periaatteita, joiden mukaan esimerkiksi palvelinyhteydet on pidettävä poissa Viewistä eli näkymästä ja niitä suositellaan laitettavaksi Modeliin (esim. yksittäinen itemi) tai Collectioniin (kaikki itemit).



Kuvio 1. Backbone.js:lle tyypillinen arkkitehtuurimalli ja sen toiminta

Backbone.js on esimerkiksi MVC-framework AngularJS:ää vanhempi työkalu. Siinä, missä AngularJS:n suosion käyrä on jyrkkä sen räjähdysmäisestä suosiosta johtuen, on Backbone.js:n suosio pysynyt suhteellisen vakaana. (Usage of JavaScript libraries for websites, 2016)

Mallin reaaliaikaisen päivittämisen näkymässä mahdollistava two-way databinding ei ole Backbone.js:n tarjoama ominaisuus. Tämä ominaisuus on uusimmissa sovelluskehityksen frameworkeissä, mikä tekeekin siitä huomattavan heikkouden Backbone.js:lle. (Hannah 2015.)

4.1.2 jQuery

Oleellinen, myös Backbone.js:n Viewin eli näkymän DOM-rajapinnan manipuloimiseen liittyvä kirjasto on jQuery. Se on ominaisuuksiltaan rikas, pieni ja nopea JavaScript-kirjasto. Sen avulla voidaan tehdä HTML-elementtien manipulointia, tapahtumien käsittelyä, animaatioita ja Ajax-toimenpiteitä helpommin. Myös selaimien tuki jQuerylle on hyvin laaja. (What is jQuery? 2013.)

JavaScriptin yleistyessä selaimissa haluttiin sen käyttöä helpottaa ja jQueryn modulaarisen rakenteensa ansiosta sillä voitiin rakentaa dynaamisia nettisivuja. Esimerkiksi klikkauksesta laajentuvat tekstilaatikot voitiin toteuttaa todella vähällä koodilla

jQuery:n ansiosta. Esimerkissä toggle()-funktio, joka näyttää tai piilottaa elementin sen sitä ajettaessa esimerkiksi napin painalluksella:

```
$(document).ready(function(){
    $("button").click(function(){
        $("p").toggle();
    });
});
```

Se toimii käytännössä kuin mikä tahansa muukin JavaScript-kirjasto: linkityksellä sen-JavaScript-tiedostoon hakemistossa tai linkittämällä siihen jossain muualla internetissä, kuten Googlen tai Microsoftin mirror-palveluissa, saatin jQuery toimimaan selaimessa. Se toimii yhdessä Ajax-toimintojenkin kanssa ilman sivun uudelleenlataamista. Ylimääräiset nettisivun uudelleenlataukset ovat yksi niistä asioista, joihin jQueryn kehityksessä haluttiin vastata internetin käyttökokemuksen parantamiseksi. (Presentation of JQuery 2013.)

Suhteellisen pitkäikäinen jQuery-kirjasto julkaistiin jo 2006 ja on tänä päivänäkin suosituin JavaScript-kirjasto, jättäen taakseen mm. Bootstrap- ja Modernizr-kirjastot. Maailmanlaajuisesti jopa yli 70% kaikista nettisivuista käyttää jQuery-kirjastoa jollain tapaa. (Usage of JavaScript libraries for websites 2016.)

4.2 DOM ja Virtual DOM

DOM on yleisesti selaimissa käytetty ohjelmointirajapinta HTML- sekä XML-dokumenteille. DOM määrittää loogisen rakenteen näille dokumenteille sekä sen, miten näitä voidaan käsitellä ja manipuloida. Dokumenttia DOM:ssa voi kuvata mallilla, joka muistuttaa melkein puumallia, mutta alkaa ylhäältä ja jakaantuu yhä useammin alas mentäessä. Nämä malli kuvaa sitä, missä DOM-solmut (eng. nodes) sijaitsevat tässä mallissa. Nämä solmut toimivat olioina, joiden manipuloimisen selaimissa käytetty JavaScript mahdollistaa. (What is the Document Object Model?, 2000.)

HTML DOM mahdollistaa JavaScriptin käytön HTML-elementtien manipuloimiseen.JavaScript pystyy mm. manipuloimaan HTML-elementtejä ja -attribuutteja, sivustolle

määriteltyjä tyylejä sekä luomaan uusia HTML-tapahtumia sivulle. (JavaScript HTML DOM, 2010.)

Verkkosivulla, jolla käytetään JavaScriptiä sisällön esittämiseen ja toimintojen mahdollistamiseen, JavaScript-koodi linkitetään usein heti dokumentin alkuun - aivan kuten tyylitiedostotkin. Jotta JavaScript ei alkaisi vielä viittaamaan HTML-elementteihin joita olla vielä luotu, on syytä käyttää onload-funktiota, tai muita sen vaihtoehtoisia funktioita kirjastosta riippuen. (Blaze, 2008.)

Virtual DOM tekee perinteisestä DOM:sta abstraktion, joka on käytännössä kevyt kopio "oikeasta" DOM:sta. Virtuaalinen DOM vertaa muutoksia virtuaaliseen DOM:iin tehtyjen muutosten sekä oikean DOM:n välillä ja renderöi muuttuneet solmut uudelleen. Tämä tekee virtuaalisesta DOM:sta huomattavasti perinteistä DOM:a tehokkaamman. Virtuaalinen DOM tarkastaa erot joko pollaamalla eli tekemällä tarkastuksen tietyn aikavälin välein tai muussa tapauksessa havaitsemalla muutoksen tilassa esimerkiksi käyttäjän toimintojen aiheuttamana. (Freed, 2011)

Esimerkiksi virtuaalinen DOM React-kirjastossa käyttää tehokkaita ero-algoritmeja muutoksien havaitsemiseen, päivittää HTML:n alielementtejä samanaikaisesti sekä antaa muutokset erissä DOM:iin. Syitä, miksi perinteistä DOM-rajapintaa ei vastaavasti päivitetä, on mm. se, ettei tilojen seurannasta voida pitää kirjaa ja myös se, että etenkin yhden sivun sovelluksissa suorituskyky kärsii perinteisessä DOM:ssa. Tämä johtuu käytännössä siitä, ettei SPA-sovelluksia (Single Page Application) ollut käytössä nykyiseen malliin aikana, jolloin DOM-rajapintaa määriteltiin. (The Virtual DOM. 2016)

4.3 PHP yleisesti

PHP oli alkujaan internetin toiminnalle oleellisen CGI-tekniikan skriptikokoelma, josta se kehittyi dynaamiseksi komentosarjakieleksi saaden lopulta myös tuen olio-ohjelmoinnille 5-versionsa myötä. PHP tunnetaan pitkästä historiastaan ja etenkin siitä, että sille voi olettaa saavan tuen usealta palveluntarjoajalta, aina virtuaalisista pilvipalveluista jopa ilmaistarjoajiin.

Kielen kehitys alkoi noin vuotta ennen sen ensimmäistä julkaisuvuotta, eli vuonna 1994. Rasmus Lerdorfin kehittämä ohjelmointikieli oltiin nimettä alunperin nimellä

Personal Home Page Tools ja sen nimi vaihtui julkaisversioiden myötä. Vuonna 1998 julkaistussa versiossa 3 sen nimi muuttui nykyiseen muotoonsa "PHP: Hypertext Preprocessor". (PHP: Releases 2016.)

PHP on heikosti tyypitetty. Samoin kuin JavaScriptissä, PHP:ssa muuttujien määrittely ei tarvitse tyypitystä kuten vaikkapa Javassa, jossa esimerkiksi kokonaislukutyyppi (int) tai merkkijonotyyppi (String) on määriteltävä muuttujalle. Muita heikon tyypityksen piirteitä on ajonaikaiset toimenpiteet muuttujille, kuten niiden luonti tai poistaminen muistista. Vaikutteita PHP:n on ilmoitettu saaneen mm. C:stä, Javasta ja Perlistä. (Rantala, 2005)

Erityinen piirre PHP:lle on sen kyky käyttää staattista HTML-kuvauskieltä tiedostoissaan. PHP tulkitsee suoritettavat osat tiedostoissaan ja tulostaa näiden sekä HTML-elementtien yhteistuloksen käyttäjälle. Tämä yhteensopivuus myös hyvin siitä ajasta, kun palvelinohjelmointi ei ollut vielä jakaantunut yhtä selvästi eri osiin, kuten MVC- tai muihin malleihin, jotka tulivat myös PHP-maailmaan myöhemmin PHP-kehysympäristöjen, kuten Laravel tai Codeigniter, myötä.

PHP-kehittäjälle on hyötyä sen virheenraportoinnista, joka kirjaa php-error-logiin virheet palvelimella. Jos virheenrapotointi on päällä, se näyttää oleelliset virheet myös selaimelle, mistä on hyötyä kehityspalvelimella, mutta mitä ei kannata jättää päälle tuotantopalvelimelle sen haitatessa oleellisesti tietoturvaa. PHP:ssa koodin kääntämisen suorittaa palvelimen PHP-moduuli. Tämän ominaisuuden ansiosta PHP:lla skriptien ajaminen on huomattavan helppoa. (PHP: Preface n.d.)

4.3.1 PHP5 ja PHP7

Pitkään odotettu PHP:n viimeisin versio, PHP 7, julkaistiin 3. joulukuuta 2015. PHP:n versiota 7 voidaan pitää tärkeimpänä muutoksena PHP:lle sitten PHP 5:n julkaisun vuonna 2004. Hyvänä huomiona mainittakoon, että versioiden 5 ja 7 välillä ei julkaistu versiota 6, vaan kehityksessä siirryttiin suoraan versionumerosta 5.6 (tarkalleen 5.6.26) versionumeroon 7.0.0. (PHP: Releases 2016.)

PHP 6:aa oltiin kehittämässä, mutta kun sen sisältämästä PHP Unicode string –tuesta päätettiin luopua, päätettiin koko PHP 6-projekti olla julkaisematta vuonna 2010. (PHP 7: Will developers adopt the new version? 2016.)

Uusina ominaisuuksina PHP:en versionumero 7 tuo:

- parannetun suorituskyvyn
- pienemmän muistinkulutuksen
- deklaroinnin skaalarin tyypeille (parametrien/return typen pakotus)
- johdonmukaisen 64-bittisyyden tuen
- parannellun poikkeusten hierarkian
- monien fatal errorien konvertoinnin poikkeuksiksi
- uudenlaisten turvallisten satunnaislukujen generaattorin
- monen vanhentuneen SAPI:n (server-API:n) poiston
- uudenlaisen operaattorin nullin vertaamiseksi
- tuen anonyymeille luokille
- tuen nolla-kustannuksellisille asserteille.

PHP 7:n taustalla on avoimen koodin skriptimoottori Zend Engine 3. Sen avulla PHP 7:n luvataan olevan lähes kaksinkertaisesti niin tehokas ja 50% paremmin muistia käyttävä kuin PHP 5.6. PHP 7:n luvataan myös pystyvän palvelemaan useampia samanaikaisia käyttäjiä samalla laitteistolla, ja sen kerrotaankin olevan kehitetty silmällä pitäen tämän päivän verkon työmääriä. (PHP 7 - Performance 2016.)

Ohjelmontikielien sekä niiden kehysympäristöjen päivityksissä suuri huolenaihe on taaksepäin yhteensopivuus. Myös PHP 7:aan siirtymistä voi haitata yhteensopimattomuus etenkin siksi, että kun viime vuosien päivitystahti on ollut PHP:n osalta suhteellisen hidas, on suuri osa palvelin-alustaa voinut olla toteutettu tavalla, joka ei suoraan toimi PHP 7-ympäristössä. Esimerkkeinä uudistuksista monet PHP:n fatal errorit ollaan konvertoitu poikkeuksiksi PHP 7:ssa. Nämä virheen poikkeukset perityvät Error-luokasta, mikä implementoi Throwable-rajapinnan. Tämä on uusi käyttöliittymä, jonka kaikki poikkeukset perivät. Se, missä uudistus näkyy, on mm. kustomoiduissa error handlereissä. Nämä eivät välttämättä laukea, koska poikkeus voidaan heittää error handlerin tuloksen sijasta uudistuksen myötä. Muina mahdollisina ongelmina mm. se, ettei aiemmin toiminut PHP:n set_exception_handler()-metodi ei välttämättä enää saa Exception-olioita. (PHP 7: Will developers adopt the new version? 2016)

Muitakin poikkeuksiin/erroreihin liittyviä yhteensopimattomuuksia on PHP:n manuaalin listaamana useapia. Yhteensopimattomuudet koskevat myös olioita ja niiden funktioita kuten list():a, avainsanaa "global", jota voidaan käyttää enää vain yksinkertaisiin muuttujiin sekä sulkeiden käyttö, mikä ei vaikuta funktion käytökseen. Erroreiden/poikkeusten ja muuttujien/muuttujatyyppien lisäksi myös iteroinnit kokevat muutoksia. Esimerkiksi foreach ei enää muuta sisäistä taulukon pointeria, mikä muuttaa odotettua tulosta melkoisesti.

4.3.2 PHP vs. Node.js

PHP on vuosikausia - itseasiassa jopa vuosikymmeniensä - jälkeen saanut varsinaisen haastajansa niiden tekniikoiden joukossa, jotka tarjoavat ratkaisun niille, joille palvelimen nopea toiminta on tärkeää. Vuonna 1995 julkaistu PHP on jäämässä jalkoihin vuonna 2009 julkaistulle Node.js:lle, joka JavaScriptillä kirjoitettava, Googlen V8-JavaScript-moottoriin perustuva ajoympäristö. Node.js tuli saataville aikaan, jolloin etenkin asynkronisen ohjelmoinnin tarve kasvoi palvelinpuolella. Node.js pystyy suorittamaan tehtäviä samanikaisesti, kun PHP taas joutuu odottamaan edellisen valmistumista uuden aloittamiseksi. Asynkronisuus on monille kehittäjille ehkä tärkein Node.js:n tuoma etu. (Dausing 2015)

Asynkronisuus näkyy myös tavalla, missä Node.js tekee asiat tapahtumapohjaisesti, eli tulkki voi tehdä muita tapahtumia silloin kun muut prosessit ovat kesken. Tämä auttaa esimerkiksi tilanteessa, jossa joudutaan hakemaan suuri määrä dataa tietokannasta, mutta ei haluta sen olevan ainut sillä hetkellä tapahtuva asia samassa tapahtumaketjussa. Perinteisen PHP:n kohdalla tämän tietokantahaun valmistumisen odottaminen on käytännössä välttämätön vaihe.

Vaikka PHP:n kanssa harva kehittäjä kohtaa ongelmia suorituskyvyn kanssa, on Node.js:n suorituskyky vielä jotain, mikä voi yllättää kokeneemmankin PHP-kehittäjän. Nopea suorituskyky vähäisten laitteistovaatimusten lisäksi tekee Node.js:stä erinomaisen alustan etenkin pilvipalveluissa, missä laitteiston ominaisuudet voivat olla hyvin rajalliset. Node.js onkin omiaan reaaliaikaisten sovellusten kehittämisessä.

Asynkronisuuden ja pakettienhallinnan lisäksi eroina on Node.js:n tapa olla jatkuvasti päällä. Siinä, missä PHP "herää" ja tekee kaiken käyttäjäkohtaisesti pyynnön tullessa

palvelimelle, Node.js on jatkuvasti päällä ja renderöi tuloksen käyttäjälle. (Buckler 2015.)

Yksi Node.js:n vahvuuksista on NPM-pakettienhallintajärjestelmä. Yksi Node.js:n tärkeimmistä ideoista oli pitää sen ydin mahdollisimman kevyenä, ja tuoda kaikki muu siihen ulkoisten pakettien avulla käytettäväksi. Tähän PHP vastasi omalla Composer-pakettienhallintajärjestelmään, josta ei koskaan kasvanut yhtä suurta hittiä kuin NPM:stä, joka on tällä hetkellä maailman suurin pakettivarasto, ohittaen mm. Rubyn, Mavenin ja Pythonin varastot. Kuitenkin molemmat tulevat kuulumaan palvelinmaailman tekniikoihin samanaikaisesti vielä pitkään. Suurin vaikuttava asia näiden tekniikoiden valintaan lieneekin kehittäjän oma mieltymys, johon voi vaikuttaa niin selainpäässä kuin tietokannassakin käytettävä tekniikka. (Node.js vs PHP - The Workshape Smackdown! 2015)

4.4 SQL

Tietokantakielistä mahdollisesti tunnetuin, mutta toisaalta tänä päivänä merkitystään menettänyt kieli on SQL, eli Structured Query Language. SQL on tarkoitettu tiedon määrittelyyn, muokkaukseen ja hakemiseen relaatiotietokannoissa. SQL on pitkäaikainen kieli, jonka eri kehittäjien (mm. Microsoft, Oracle) muokkaamat versiot eivät eroa huomattavasti toisistaan ja niiden pääperiaatteet ovat pysyneetkin samoina. (Moilanen 2002.)

Relaatiotietokanta-ympäristö tunnetaan myös lyhenteellä "RDBMS", eli Relational database management system. SQL:n historia alkaa jo 1970-luvulta, kun englantilainen IBM:llä työskentelevä tietojenkäsittelytieteen tutkija, Edgar F. Codd, loi pohjan relaatiotietomallille tietokannoissa. Jo 1974 SQL sen alkuperäisellä nimellä julkaistiin. 1978 IBM julkaisi maailman ensimmäisenä relaatiotietokantaohjelmana pidetyn System R:n, josta kehittyi myöhemmin relaatiotietokantatuotteiden perhe DB2, eli Database 2. Vuonna 1986 SQL sai standardoinnin ANSI:lta, eli American National Standards Institutelta, jonka jälkeen se julkaistiin ensimmäisen kerran standardisoituna tietokannan kielenä Relational Software:n toimesta, josta tuli myöhemmin Oracle. (System R 2011)

SQL:n ominaisuudet tekivät siitä ylivoimaisen relaatiotietokakielen (relaatio itsessään tarkoittaa yhteyttä/suhdetta tietokantataulujen välillä). SQL:n ominaisuuksia ovat kattavat mahdollisuudet tiedon käsittelyyn relaatiotietokannoissa; mm. tiedon kuvaaminen ja muokkaaminen, mahdollisuus sulauttaa erilaisia SQL:aa tukevia moduuleja yhteen, mahdollisuus luoda ja tuhota tietokantoja ja tietokantatauluja, sekä mahdollisuus luoda käyttöoikeuksia tietokannan eri alueille/ominaisuuksille. SQL:n peruskomennot ovat CREATE, SELECT, INSERT, UPDATE, DELETE ja DROP. SQL:n komentoja voidaan myös ryhmitellä seuraavasti:

Tiedon määrittelemiseen:

CREATE: Luo taulun tai näkymän taululle tai muun olion tietokannassa
ALTER: Muokkaa olemassaolevaa tietokantaoliota kuten esimerkiksi taulua
DROP: Poistaa taulun, näkymän taulusta tai muun olion tietokannasta

Tiedon manipulointiin:

SELECT: Palauttaa tietyn tietueen/tulosjoukon tietokannan taulusta tai tauluista
INSERT: Luo tietueen
UPDATE: Päivittää tietueen
DELETE: Poistaa tietueen

Tiedon kontrollointiin:

GRANT: Luo käyttöoikeudet käyttäjälle
REVOKE: Poistaa käyttöoikeudet käyttäjältä

Esimerkissä eräs opinnäytetyön kehitystyössä käytetty PHP-koodiin sisällytetty SQL-kysely, jossa käytettiin SELECT-komentoa tiedon hakemiseen:

```
$workers = db_fetch_all('SELECT target_id, is_group FROM reminder_users_groups WHERE reminder_id = '.$reminder['id']);
```

4.5 NoSQL vastineena

Tietokannoissa on tapahtumassa muutosta pitkään aikaan. Relaatiotietokannat ovat olleet tietokantojen johtava malli jo 20 vuotta, mutta nykyään puhutaan NoSQL-kielistä, käytännössä siis ei-relaatiotietokantakielistä. Kehittäjien tarpeet datan säilyttämiselle ovat muuttuneet ja NoSQL antaa mahdollisuuden pitää data hyvin rakenteettomana ja heterogeenisenä. Useimmiten NoSQL tarkoittaa dataa JSON-muodossa. (Gentz 2016.)

NoSQL:n voidaan sanoa kattavan suuren osan niistä omanisuuksista, joita tämän päivän sovelluksen kehittäminen vaatii. Kehitetyt sovellukset voivat luoda suuria määriä nopeasti tietotyyppinsä vaihtavaa dataa, niin rakenteellista, "puolirakenteellista" tai rakenteetonta, kuin jopa polymorfista eli muotoaan muuttavaa dataa. Relaatiotietokanta tarvitsee niin sanotun skeeman, eli tietotyypin määrityksen ennen kuin tietoa voidaan tietueeseen asettaa. NoSQL-tietokannassa tietueen tyyppi voi vaihdella jatkuvasti, eikä esimerkiksi tietotyypin muuttaminen näin toimi rajoitteena.

Vapaan lähdekoodin dokumenttitietokannan kehittäjä, MongoDB, lupaa NoSQL:n vastaavan myös niihin tarpeisiin, joita monialustaiset sovellukset edellyttävät tietokannaltaan. Yksi syy tälle on tiedon varastoiminen pilvipalveluihin sen sijaan, että se olisi säilötty keskitetysti kuten aiemmin on totuttu tekemään etenkin relaatiotietokanta-maailmassa. Monet toiminnot relaatiotietokannoissa vaativat datan keskittämistä, kuten taulujen liittäminen sekä transaktioiden tekemisen. Nämä estävät esimerkiksi automaattisen tietokannan pirstaloinnin (eng. "auto-sharding"), jolla tietoa voidaan jakaa eri paikkoihin. Tässä NoSQL voikin olla RDBMS:ää parempi vaihtoehto. Muita MongoDB:n listaamia ominaisuuksia NoSQL-tietokannalle ovat replikointi ja integroitu cachen eli välimuistin käyttö. NoSQL:ää pidetään relaatiotietokakieliä paremmin skaalautuvana sekä suorituskykyisempänä, minkä todetaan myös nopeuttavan työskentelyä tämän päivän ketterässä ohjelmistonkehityksessä. (What is NoSQL?, 2013)

4.6 Cron

Cron on Unixin kaltaisissa ympäristöissä toimiva apuohjelma, joka on tarkoitettu ajamaan ajastettuja ja toistettuja tapahtumia. Cron huolehtii ajastettujen skriptien ajamisesta silloin, kun tietokone on päällä ja kun muitakaan sen toimintaa haittaavia tekijöitä ei ole. Cron voi ajaa komentoja tai shell-skriptejä esimerkiksi tiettyinä aikoina vuorokaudesta tai tiettyinä vuorokausina viikkossa. Cron on erinomainen myös hoitamaan usein toistuvat tapahtumat, kuten minuutin välein tehtävän tarkistuksen sille, halutaanko käyttäjälle ilmoittaa jostakin muutoksesta ilmoituksella. Ajastettu haku voi toimia turhaa dataliikennettä vähentävänä tekijänä verrattuna siihen, että ajastetun haun lisäksi toiminnot itsessään lähettäisivät viestin serveriltä). (Intro to Cron 1999.)

Esimerkki Cronilla kommenon ajamisesta:

5 * * * * /home/aspera/my_script.sh

5 tarkoittaa, että komento ajetaan jokaisen tunnin 5. minuutti. (Corliss 2016.) Viiden minuutin välein ajettavan syntaksimuoto olisi "*/5". Seuraava on .sh-tiedoston polku, missä voi olla esimerkiksi ajettava PHP-skripti. Sh-tiedoston tarkka sisältö voi olla esimerkiksi "*cd /var/www/server-env/shell/; php cron-handler.php*". Siinä mennään ihmisellekin tutuilla kommennoilla ensin oikeaan hakemistoon, minkä jälkeen ajetaan php:lla php-tiedosto. Php-tiedostossa voi olla pelkästään vain funktio ja funktiokutsu sille.

Crontab on esimerkiksi komentokehotteessa näytettävä lista komennoista, jotka ympäristössä on ajastettu ajettavaksi. Siinä näkyvät ajettavan skriptin toisto *-merkeillä sekä osoite (esimerkiksi shell-hakemistossa). Crontab viittaa "cron table":en, ja sen esitystapaa voidaankin pitää tietynlaisena tauluna (vrt. tietokantataulu).

Anacron on vastine Cronille siinä tapauksessa, että tietokone ei ole päällä vuorokauden ympäri. Sen käyttäminen sopii päivittäistä, viikottaisten ja kuukausittaisten ajojen ajamiseen. Sen on nimelläänsäkin viitaten asynkroninen, eli se hoitaa ns. velvollisuutensa konfiguraation mukaan viimeistään aina silloin kun tietokone tulee päälle.

Tavallinen Cron suorittaa kaiken synkronisesti eli tässä tapauksessa se jättää ajot suorittamatta, jos sillä ei ole sillä hetkellä mahdollisuutta siihen tietokoneen ollessa pois päältä tai muussa ei-toimintavalmiissa tilassa. (Anacron 2015.)

5 Arkkitehtuurimallien tarkoitus

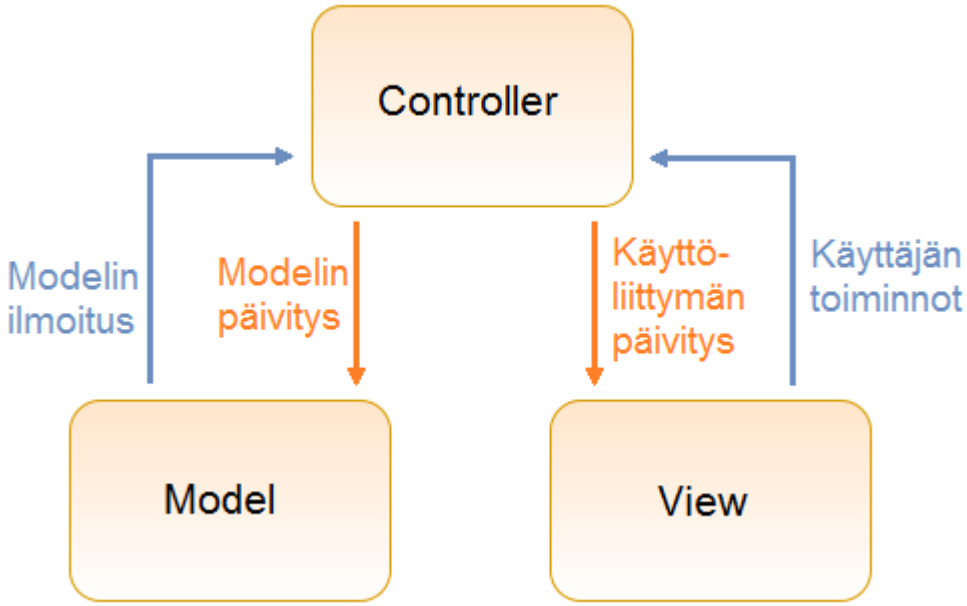
Arkkitehtuurimalli/suunnittelumalli on laaja käsite, jonka tarkoitus on luoda optimoitu, uudelleenkäyttettävä ratkaisu ohjelmoinnissa ilmeneviin ongelmiin. Käsitteenä se on suurempi kuin esimerkiksi valittu framework tai kirjasto, joka vain valitaan ja liitetään projektiin. Arkkitehtuurimalli/suunnittelumalli on käytännössä mallinne (eng. template), joka täytäntöönpannaan sille oikealla tavalla ja joka on myös riippumaton ohjelmointikielestä. (Bautista 2010.)

Suunnittelumalleista puhuttiin jo 1970- ja 1980-luvuilla. 1970-luvulla julkaistiin kirja "A Pattern Language", jossa ei tosin kuvattu ohjelmistokehityksen malleja, vaan kaupunkien suunnittelua. 1980-luvulla suunnittelumalleista puhuttiin jossain määrin olio-ohjelmoinnin yhteydessä, mutta varsinainen läpilyönti tietoisuudessa suunnittelumalleihin liittyen tapahtui vasta 1994, kun kirja "Design Patterns: Elements of Reusable Object-Oriented Software" julkaistiin. Kirja kuvasi 23 suunnittelumallia sekä tarkasteli yleisiä ongelmia, joita saattoi ohjelman suunnitteluvaiheessa ilmetä. (Fulcher 2014.)

Oikein käytettynä suunnittelumallit voivat nopeuttaa kehitystyötä tarjoamalla testattuja ja varmoja kehityksen paradigmoja eli yleisesti hyväksyttyjä oppirakennelmia ja suuntauksia ohjelmistokehityksessä. Tehokas, järkevä kehitystyö ottaa huomioon myös sen, että tietyt ongelmat tai viat voivat tulla ilmi vasta myöhemmässä vaiheessa kehitystyötä. Suunnittelumallien tehtävä on osaltaan estää näitä "konflikteja", mutta myös tehdä niiden korjaamisen helpommaksi. Merkittävä hyöty ohjelmiston jakamisessa osiin tietyn mallin mukaan on etenkin sen luettavuuden selkeytyminen kehittäjälle. (Design Patterns 2013)

5.1 MVC-malli

Tämän päivän todennäköisesti tunnetuin arkkitehtuurimalli, MVC-malli, jakaa sovelluksen logiikan kolmeen osaan edistäen näin sovelluksen modulaarisuutta, näiden yhteistyön helpottamista sekä uudelleenkäyttöä. Arkkitehtuurimallissa "M" eli Model määrittää, mitä dataa sovelluksen tulisi sisältää. Useimmiten jos data muuttuu, myös "V" eli View päivittyy esittäen uuden datan, joka modelissa on. Joissakin tapauksissa Model voi myös ilmoittaa "C":lle eli Controllerille muutoksesta datassa. (MVC architecture 2014.)



Kuvio 2. MVC-mallin toiminta

Viewin eli näkymän tehtävä on esittää modelin sisältö käyttäjälle. Viewsissä on kaikki, mitä käyttäjä näkee päätelaitteen näytöltä; graafinen sisältö sekä modelin eli varastoidun (tai staattisen) datan yksityiskohdat.

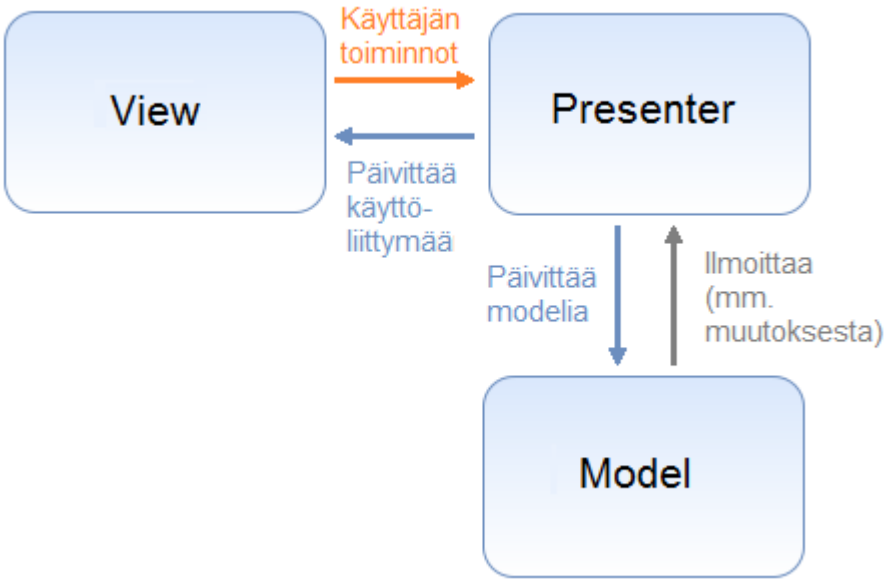
Controller sisältää logiikan, jolla Model, View tai molemmat päivitetään sen mukaan, mitä muutoksia niihin tarvitaan. Jos kyseessä on esimerkiksi ostoskori-sovellus, controller eli ohjain päivittää tuotteen lukumäärän molempiin näistä silloin, kun asiakas määrittää tuotteen uuden lukumäärän sovellusta käyttäessään. Käyttäjä tekee muutokset klikkaamalla painikkeita viewissä, joista lähtee tieto controllille, joka tekee päivityksen. Jos käyttäjä haluaa esimerkiksi muuttaa ostoskorinsa sisällön

esitysjärjestystä, ei kontrollerin tarvitse muuttaa modelia mitenkään. Muutokset näkyvät ainoastaan väliaikaisesti pelkässä viewissä käyttäjälle, eikä modelin ole tarpeen "tietää" tästä. (MVC architecture 2014.)

Tiivistettynä Model pitää sisällään datan ja bisneslogiikan, View esittää datan käyttäjälle halutussa formaatissa ja ulkoasussa, Controller vastaanottaa käyttäjältä pyyntöjä ja kutsuu tarvittavia resursseja ohjataakseen ne eteenpäin. Vaikka Mode-View-Controller onkin tämän hetken todennäköisesti suosituin malli sovellukselle, voi joissain tapauksissa erilaisesta mallista olla enemmän hyötyä sovellukselle riippuen sen ominaispiirteistä. (Pastor 2010.)

5.2 MVP-malli

MVP- eli Mode-View-Presenter-mallia voidaan pitää MVC-mallin pitemmälle kehittyneenä versiona. MVP-mallissa "View" eli näkymä kutsuu "Presenteriä" eli esittäjää tarpeen mukaan, eli esimerkiksi silloin, kun käyttäjä on tehnyt toimintoja käyttöliittymässä (esimerkiksi painanut nappia).



Kuvio 3. MVP-mallin toiminta

Model eli malli toimii rajapintana datalle. Jokaisen osan sovelluksesta, joka tarvitsee tiettyä dataa toimintaansa, on mentävä mentävä tämän rajapinnan kautta. Esimerkiksi kaikki validaatio-data on modelin hallussa. MPV-mallissa käyttäjä

vuorovaikuttaa "Viewiin", eli näkymään. Näkymiä voi olla lukuisia täysin sen mukaan, miten sovellus on haluttu rakentaa. MVP-mallille yleistä on näkymien paisuminen huomattavan suureksi, ja joissakin tapauksissa näkymä voi olla keskitetty enemmän tai vähemmän pelkästään yhdelle kehittäjälle projektissa.

Presenter eli esittäjä toimii tavallaan "välittäjänä" MVP-mallissa. Käyttäjän vuorovaikutuksen aiheuttaman "eventin" bisneslogiikka on esittäjässä. Tyypillisesti Viewillä eli näkymällä on ainoastaan "eventin" eli tapahtuman "handler" eli käsittelijä, jolla se voi kutsua esittäjän funktioita. Näin näkymän kehittäjän ei tarvitse käyttää aikaansa muuhun kuin näkymän kehittämiseen.

Esittäjän tehtävä on siis välittää data "Modelista" eli mallista "Viewille" eli näkymälle mahdollisesti formatoimalla sitä sen siten, että näkymä voisi sen esittää ongelmitta. MVP-malli edistää myös yksikkötestausta, mikä tekeekin siitä toimivan suunnittelumallin erittäin vakaan ohjelmiston kehittämiseksi.

5.3 MVC vs. MVP

Tänä päivänä keskustelua käydään siitä, onko MVC-malli jo vanhentunut. MVC-malli tuli tutuksi etenkin aikana, jolloin nettisivun lomake täytettiin, lähetettiin palvelimelle validoitavaksi ja saatiin vastaus siitä, oliko esimerkiksi rekisteröitymislomakkeen kentät oikein täytetty. Tämä oli ajalta, jolloin JavaScriptiä ei juurikaan ajettu selaimessa vaan vastaavat toiminallisuudet tehtiin PHP:lla. Kun rekisteröityjälle kerrotaan validaation tulokset, renderöidään view eli näkymä uudestaan.

MVP-malli vastaa sen ajan tarkoituksiin, kun näkymää ei tarvitse enää renderöidä jatkuvasti kuten MVC-aikakaudella. Yksi MVP:n tarkoituksista on tehdä Viewit eli näkymät uudelleenkäytettäviksi. Loppujen lopuksi MVC ja MVP eivät ole erityisen kaukana toisistaan ja niiden perimmäiset ongelmatkin arkkitehtuurissa ovat samoja. (Papou 2013.)

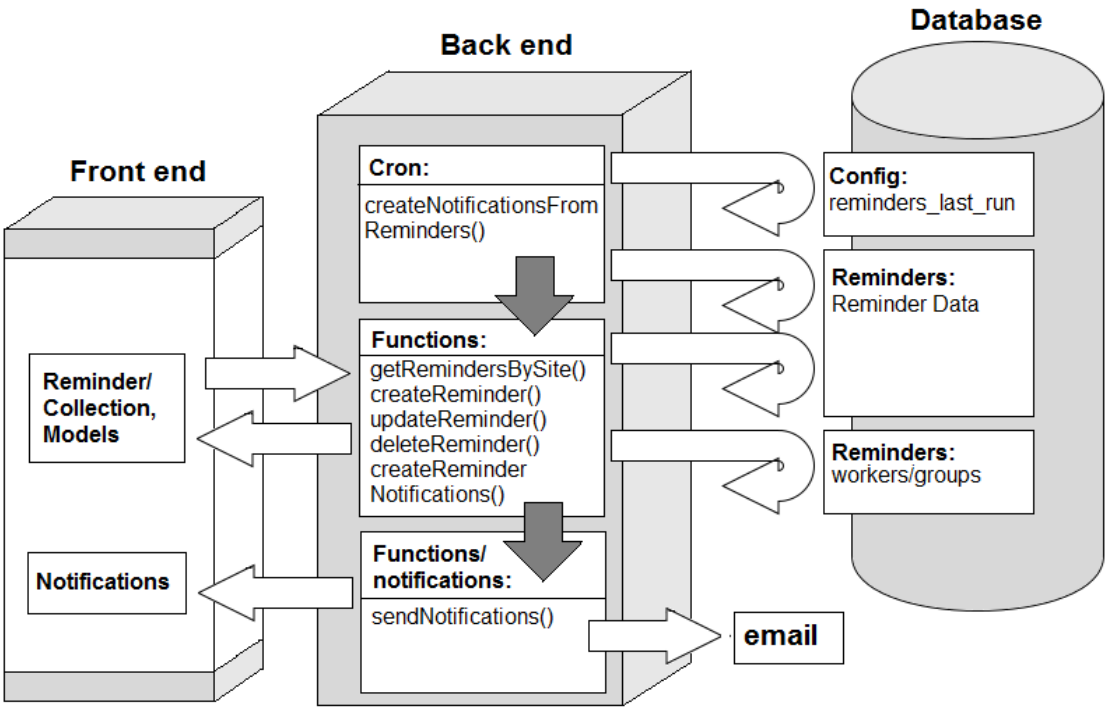
Erot tiivistettynä: MVP - MVC

MVP	**MVC**

***MVC:n kehittyneempi malli**	**Yksi vanhimmista suunnittelumalleista yleisimmille ongelmille**
***Näkymä käsittelee käyttäjän vuorovaikutuksen ja kutsuu esittäjää tarvittaessa**	**Ohjain (controller) käsittelee vuorovaikutuksen ja komentaa mallia**
Näkymä on täysin passiivinen ja kaiken toiminnon mentävä esittäjän läpi	**Näkymällä on jossain määrin logiikkaa, ja se voi vaikuttaa malliin suoraan**
Tukee yksikkötestausta hyvin	**Rajallinen tuki yksikkötestaukselle**

6 Palvelinpuolen toteutus kokonaisuutena

Palvelimelle rakennettiin itsenäinen minuutin välein muistutusten muistutusaikaa tarkistava Cron-funktio sekä normaalit luku-, lisäys-, muokkaus- ja poisto-funktiot, niin sanotut CRUD-funktiot. Cron-tarkistus tehdään aina palvelimen ollessa päällä ja CRUD-funktiot tehtiin ainoastaan muistutusten hallintaa varten asiakaspuolella.



Kuvio 4. Palvelintoteutuksen toiminta

Kokonaiskuvan saamisen helpottamiseksi toimintalogiikkaa palvelimen ja tietokannan sekä Front endin eli asiakaspuolen välillä kuvaa kuvio 4, joka kuvataan yhteyttä näiden välillä suhteellisen tarkasti. Back endin eli palvelinpuolen Cron-ajo toimii poikkeuksetta minuutin välein jotta tarkistus olisi aktiivinen ja jotta muistutussaika saataisiin tarkistettua ja ilmoitettua siitä tarvittaessa käyttäjälle.

Jos muistutuksen muistutusaika on tarkistusajan sisällä, eli siitä pitää ilmoittaa käyttäjälle, tehdään siitä ilmoitus. Muistutuksista tehtävien ilmoitusten funktio on reminder.functions.php-tiedostossa, jossa ilmoitus muistutuksesta valmistellaan, ja joka lopulta annetaan parametrina varsinaiselle ilmoitusfunktiolle. Tämä funktio lähettää ilmoituksen asiakaspuolelle ja mahdollisesti myös käyttäjän sähköpostiin. Ilmoituksia voi tulla muistakin asioista kuin muistutuksista, joten aiemmin tehtyä funktiota oli loogista yrittää hyödyntää tässäkin kohtaa.

Cron-ajo tarkistaa sekä config-taulun että muistutusten taulun, muistutusten funktiot tarkistavat muistutusten taulun sekä muistutusten workers/groups-taulun. Asiakaspuolesta lähetetyt pyynnöt muistutuksille menevät aina reminder.functions.php-tiedoston funktioihin, eikä Cron-funktioon ole tarkoitus vaikuttaa asiakaspuolella mitenkään. Asiakaspuolella sijaitseva Backbone.js:n Collection ajaa fetch()-funktion aina getReminderBySite()-funktioon, Backbone.js:n Model lähettää ja vastaanottaa pyynnöt create-, update- ja delete-funktioihin.

createReminderNotifications() on pelkästään Cron-ajon kutsuma funktio, jota asiakaspuolelta ei voida kutsua. Jos createReminderNotifications()-funktio ajetaan Cron-ketjussa, ajetaan myös sendNotifications()-funktio palvelimella, joka lähettää ilmoituksen asiakaspuolen ilmoitusten komponenteille ja ilmoitus luodaan käyttäjälle kuten mikä tahansa ilmoitus.

6.1 actionHandler

Sovelluksessa actionHandler.php toimii ensimmäisenä vaiheena, kun palvelimelle lähetetään pyyntö. Pyynnöstä poimitaan PHP:lla 'action', joka laitetaan switch-caseen. Pyydetty action voi olla mitä tahansa kirjautumisesta pienemmän tulosjoukon hakemiseen tietyssä näkymässä. Useimmiten actionHandlerin casen

ensimmäisinä vaiheina on pyynnöstä parametrien sijoittaminen omiin muuttujiinsa. Esimerkiksi pyynnön 'name'-parametrista tulee $name-muuttuja $_REQUEST:lla. Jos tietoa haetaan, actionHandlerin case kutsuu haluttua funktiota, joka yleisesti functions-hakemistossa. Tässä vaiheessa muuttujat ovat sellaisia, minkälaisiksi PHP on ne $_REQUEST:lla muodostanut, mutta actionHandlerin kutsumien funktioiden alussa ne alustetaan halutunlaisiksi.

Esimerkiksi String-muuttujat escapetetaan, jotta SQL-injektioiden riski pienenisi. Jos funktio palauttaa jotain onnistuneesti, actionHandler palauttaa sen (useimmiten taulukkona) takaisin asiakaspuolelle. Jos tietuetta ei löydykään, eli actionHandlerin kutsuma funktio epäonnistuu, palauttaa actionHandler ajaxErrorin ja sopivan viestin.

Esimerkki muistutuksen luomisesta actionHandlerissa, jossa tehdään kerrotut asiat:

```
case "createReminder":
$job_id = $_REQUEST['job_id'];
$name = $_REQUEST['name'];
$description = $_REQUEST['description'];
$time = $_REQUEST['time'];
$repeat = $_REQUEST['repeat'];
$workers = $_REQUEST['workers'];
$is_group = $_REQUEST['is_group'];
$id = createReminder($job_id, $name, $description, $companyusers, $time,
$repeat, $workers, $is_group);
if($id) {
  die(ajaxSuccess("Created reminder with id $id",array('reminderid' => $id)));
} else {
  die(ajaxFail("Failed to create reminder"));
}
```

6.2 Functions

Seuraavaksi on listattuna functions-hakemistoon lisätyt funktiot muistutus-toiminallisuuksien mahdollistamiseksi.

6.2.1 Create

Actionhandlerin esimerkissä oli funktiokutsu uuden muistutuksen luomiselle. Varsinainen luonti-funktio funktio-tiedostossa on seuraavanlainen:

```
function createReminder($job_id, $name, $description, $userid, $time, $repeat, $workers, $is_group) {
	$job_id = intval($job_id);
	$name = escape($name);
	$description = escape($description);
	$userid = intval($userid);
	$time = escape($time);
	$gmtime = date("Y-m-d\TH:i:s\Z", $time);
	$nonUnixDate = (string)$gmtime;
	$repeat = escape($repeat);

	if(!$userid) {
		return false;
	}

	$id = db_insert("INSERT INTO reminders (`job_id`, `creator`, `last_modified_by`,
	`name`, `description`, `time`, `nextReminder`, `repeat`) VALUES ($job_id, '".$userid."', '', '".$name."',
	'".$description."', '".$gmtime."', '".$gmtime."', '".$repeat."');");

	foreach($workers as $v) {
		$v = intval($v);
		if ($v) {
```

```
                $u = db_insert('INSERT INTO reminder_users_groups (reminder_id, target_id, is_group) VALUES ("'.$id.'", "'.$v.'", "'.$is_group.'")');
            }
        }

        return $id;
}
```

Koska actionhandlerissa ei tehty escape-funktion ajoa muuttujille, tehdään se tässä. Escape()-funktion lisäksi funktion alussa käytetään intval()-funktiota SQL-lausetta varten. Sillä saadaan lauseeseen varmasti kokonaisluku, mikä on tietoturvan kannalta positiivinen asia.

Asia, missä koettiin vaikeuksia, oli oikeanlaisen aikaformaatin saaminen kantaan ja etenkin sieltä ulospäin. Yleinen ohje oli, että aikaformaateissa käytettäisiin unix-timestampeja. Näin voitiin varmistua, että aikaformaatti on tietokannassa poikkeuksetta sama.

Tietoturvan kannalta haluttiin myös varmistaa, että käyttäjä on varmasti kirjautunut. Jos $userid-muuttujaa ei olla actionhandlerissa saatu laitettua parametriksi tämän funktion funktiokutsuun, tulee funktio epäonnistumaan välittömästi ennen tietokantatoimintoja. Uuden muistutuksen kantaan lisäämisen jälkeen oli myös lisättävä valitut työntekijät reminder_users_groups-tauluun. Ne lisättiin taulukkoa iteroimalla. Taulukossa oli tietty määrä työntekijöitä, jotka lisättiin yksittellen uusi rivin tauluun luoden.

Lopuksi funktio palautta tietokantamoottorin asettaman ID:n. Miksi tämä oli tärkeää saada takaisinpäin tietokannasta, oli siksi, että se haluttiin saada myös asiakaspuoleen asti modelin ID:ksi. Ilman ID:n asettamista Backbone.js:n Model ei olisi toiminut halutusti, eikä esimerkiksi muistutusta olisi voinut muokata välittömästi sivua/näkymää uudelleen lataamatta.

6.2.2 Read

Alunperin read eli lukeminen oli tarkoitus tehdä muistutuksen yksittäiselle hakemiselle ID:n perusteella. Tätä kuitenkin muutettiin siinä vaiheessa, kun huomattiin että komponentti asiakaspuolella lataa näkymään muistutukset "kohteen" ID:n perusteella, joten isompi tulosjoukko oli mahdollista saada kerralla. Muistutusten taulussa on kohteen ID, jolle muistutus on lisätty. Koodissa kohteesta puhutaan "site":na:

```
function getRemindersBySite($site, $companyusers) {
    $companyusers = escape($companyusers);
    if(!$companyusers) {
        return false;
    }
    $site = intval($site);
$reminders = db_fetch_all('SELECT UNIX_TIMESTAMP(time) AS unixtime, reminders.* FROM reminders WHERE job_id = "'.$site.'" AND is_deleted = 0 AND creator in ('.$companyusers.')');

foreach($reminders as $key => $reminder) {
$workers = db_fetch_all('SELECT target_id, is_group FROM reminder_users_groups WHERE reminder_id = '.$reminder['id']);
        $parsed_workers = array();
        if (!empty($workers)) {
            $is_group = $workers[0][is_group];
            foreach ($workers as $worker) {
                $parsed_workers[] = intval($worker['target_id']);
            }
        }
        if (!empty($parsed_workers)) {
            $reminders[$key]['workers'] = $parsed_workers;
            $reminders[$key]['is_group'] = $is_group;
        }
```

```
        }

return $reminders;
}
```

Funktio tarvitsee parametreikseen ainoastaan kohteen ID:n sekä hakijan ID:n, jottei yrityksen ulkopuolinen voisi onnistua toisen yrityksen tiedon hakemisessa. $companyusers on taulukko, joka menee SQL-lauseeseen kyselyssä.

Create-osassa mainittu haaste unix-timestampien kanssa ilmaantui varsinaisesti tässä vaiheessa. Haaste johtui siitä, että vaikka timestamp oltiin laitettu unix-timestampina tietokantaan, ei se antanut sitä oletuksen unix-timestampin ulospäin. Ratkaisu oli lopuksi suhteellisen looginen: kannasta haettiin SELECT:llä time-sarake UNIX_TIMESTAMP-muodossa sekä kaikki muu *-operantilla ja WHERE-kohtaan laitettiin kohteen ID, määritys ettei valita poistetuiksi merkittyjä muistutuksia (käytännössä mitään sovelluksessa ei poistettu kannasta asti) sekä määritys, että creator-sarakkeen arvo oli oltava yksi $companyusers-taulukon sisältämistä kokonaislukuarvoista.

Seuraavassa vaiheessa lisättiin muistutukselle määritetyt työntekijät toisesta kannasta. Tässä oli iteroitava melkoisesti. Työntekijöiden lisäksi oli otettava is_group-arvo, joka määritteli sen, oliko kyseessä työntekijöiden ID:t vai työryhmien ID:t. Tämä tehtiin käytännössä asiakaspuolella UserSelector-komponenttia varten.

6.2.3 Update

Muistutukselle oli toteutettava myös päivittämismahdollisuus. Siinä se otti actionHandlerista vastaan useamman parametrin, käytännössä siis muistutuksen oleellisimmat tiedot nimestä ja kuvauksesta hälytysaikaan ja määrättyihin työntekijöihin.

```
function updateReminder($id, $name, $description, $companyusers, $time,
$repeat,
$workers, $is_group) {
    $id = intval($id);
    $name = escape($name);
    $description = escape($description);
    ...
```

Funktion alussa tehtiin muutokset muuttujiin, kuten intval()- ja escape()- funktioiden ajot, joilla niistä saatiin turvallisempia SQL-lauseeseen asetetusta varten. Käytännössä sama muuttujan arvo vain "vaihdettiin" siihen, miksi funktion ajo sen vaihtoi. Esimerkiksi $id = intval($id); ei välttämättä tehnyt arvolle käytännössä mitään, mutta teki varmaksi sen, ettei se olisi voinut olla muuta kuin kokonaisluku. Muuttujaan ei siis olisi voinut tunkea SQL:n ajoa muuttavaa dataa. Näiden funktioiden jälkeen haluttiin tarkastaa, ettei $companyusers ollut jäänyt asettumatta. Ilman aktiivista kirjautumista omaan yritykseen näin olisi käynyt. Updaten ajaminen reminders-tauluun asetti uudet arvoille oikeille paikoille sillä rivillä, jonka ID oli funktiolle annettu muistutuksen ID. Samalla tarkastettiin, että päivittäjä oli varmasti saman yrityksen käyttäjä $companyusers-muuttujasta, joka oli käytännössä käyttäjä-ID:t sisältävä taulukko.

Seuraava vaihe oli aluksi haastava. Tarkoitus oli päivittää uudet määrätyt työntekijät reminder_users_groups-tauluun. Toisin kuin työntekijöiden asettamisessa, ei tässä kohtaa voitu iteroida vaihdettuja tai poistettuja työntekijötä samalla tapaa kuin käyttäjien lisäämisessä. Järkevintä oli siis poistaa aluksi kaikki työntekijät, jonka jälkeen iteroida ne takaisin $workers-taulukosta. Lopuksi palautettiin $update-olio jos päivitys onnistui.

6.2.4 Delete

Myös "poisto" haluttiin tietysti tehdä. Poistaminen oli suhteellisen yksinkertainen toiminto, jossa ei tarvittu kuin muistutuksen ID sekä actionHandlerista saatava $companyusers, eli käyttäjä- ID:t sisältävä taulukko-olio. Kuten muuallakin, tämän

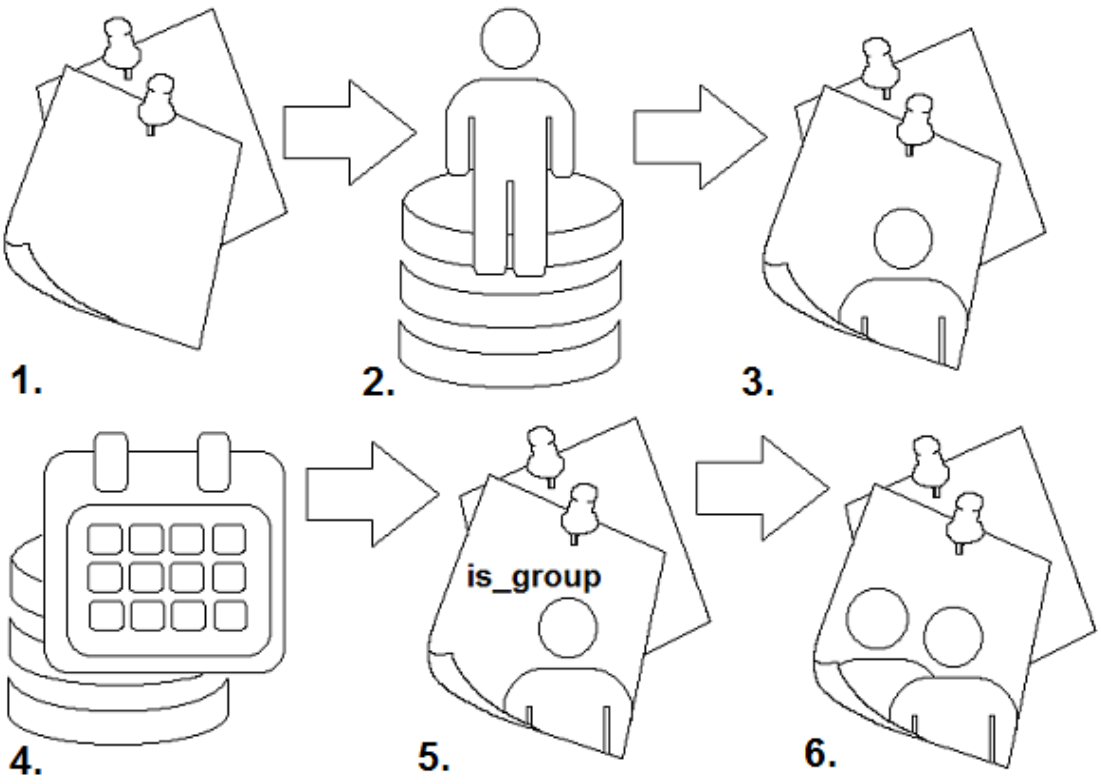
olion olemassaolo tarkastettiin ja jos se oli olemassa, edettiin koodissa $delete-olion luomiseen eli käytännössä tietokantafunktion ajamiseen.

Funktiossa muistutuksen is_deleted-arvo muutettiin arvoon "1", eli se merkittiin poistetuksi siltä riviltä, minkä ID oli muistutuksen ID jos $companyusers-taulukosta löytyi kannan taulun rivillä oleva creator-arvo. Tämä arvo varmisti sen, että käyttöoikeudet sen poistamiseen olivat olemassa.

Lopuksi palautettiin $delete-olio, joka ilmoitti, poistettiinko muistutus annetulla ID:llä onnistuneesti vai epäonnistuiko sen poisto ja mahdollisesti syy epäonnistumiselle (tämä tuli db_update-funktion asettamasta virheilmoituksesta, johon ei tässä kohtaa enää puututtu).

6.2.5 Ilmoituksen luonti

Jos muistutuksia saatiin minuutin välein ajetun Cron-skriptin seurauksena, eli niiden hälytysaika oli asetettujen aikavälien välissä, tehtiin muistutuksesta notifikaatio eli ilmoitus käyttäjälle. Sovelluksessa oli useita asioita jotka laukaisevat muistutuksen lähettämisen käyttöliittymään. Muistutus rakennetaan sekä palvelinpuolella mutta myös käyttäjäpäässä.



Kuvio 5. Ilmoituksen luonti vaiheittain

Funktiokutsu createRemindersNotifications-funktiolle tehtiin Shell-hakemistossa Cronin ajamassa funktiossa, jos hälytysaikaa täsmääviä muistutuksia löytyi. Kuvion 5 kohdassa "1" ollaan tilanteessa, missä muistutuksia on löytynyt:

Ilmoitusten rakentaminen tarvitsee muistutukselle/muistutuksille määrättyjen työntekijöiden hakemisen kannasta. Kuvion 5 kohdassa "2" haetaan työntekijöiden taulusta muistutukselle määrättyjen työntekijöiden ID:t. Ne asetetaan 'workers'-attribuutiin oliossa, ja samalla laitetaan myös 'is_group'-attribuutti paikoilleen. Kuvion 5 kohdassa "3" muistutus-oliolle on nyt määrätty työntekijä(t)/työntekijäryhmä(t).

Jos muistutuksella oli toistuva muistutusaika eli siitä ei ilmoitettu ainoastaan yhtä kertaa, piti sille laskea seuraava hälytysaika aina päivällä, viikolla, kuukaudella tai vuodella nykyisestä muistutusajasta eteenpäin. Sitä kuvaa kuvion 5 kohta "4". Aika tallennettiin muistutusten tauluun tietokannassa. Koodissa selvitettiin tässä vaiheessa is_group-attribuutin arvo. Jos se oli "1", tehtiin ylimääräinen if-lohko, joka selvitti ryhmien käyttäjät käyttäjäryhmistä. Tämä tehdään kuvion 5 kohdassa "5". Muistutuksia voitiin asettaa sekä käyttäjille että käyttäjäryhmille, mutta ilmoituksia varten oli välttämätöntä saada pelkät käyttäjä-ID:t.

Käyttäjien hakeminen käyttäjäryhmistä palautti harmillisesti moniulotteisen taulukon, kun tarkoitus oli saada yksiulotteinen taulukko, joka annettaisiin lopuksi muistutus-olion attribuutiksi kuten muutkin arvot. Ratkaisu oli tehdä funktio, joka litistäisi moniulotteisen taulukon. Tämän lisäksi piti yksiulotteisesta taulukosta myös poistaa duplikaatti-arvot, joita saattoi ilmetä käyttäjän kuuluessa samaan aikaan useampaan käyttäjäryhmään. Muuten muistutus oli lähtenyt samalle käyttäjälle useampaan kertaan.

Kuvion 5 viimeisessä kohdassa (6) annetaan valmis muistutus-olio varsinaiselle ilmoitusfuktiolle sovelluksessa (sendNotifications), joka hoitaa ilmoittamisen asiakaspuolelle saakka ja lähettää sähköpostia käyttäjille, jotka ovat valinneet vastaanottavansa ilmoituksia myös sähköpostiinsa.

6.3 Shell

Luvussa käydään läpi palvelinpuolen shell-hakemistossa olevaa Cron-skriptiä sekä koko Linux-ympäristön shell-hakemistossa olevan Cronin asetuksia.

6.3.1 Server/Shell

Palvelinpuolen Shell-hakemistossa on Cron minuutin välein ajama PHP-skripti, jossa on createNotificationsFromReminders-funktio ja sen alla funktiokutsu siihen. Tämä ajaa skriptin välittömästi kun skriptitiedosto on asetettu Crontabiin.

Funktio tekee nykyisestä hetkestä $timestamp-muuttujan ja poimii edellisen funktio ajoajan config-taulusta tietokannassa $lastRun-olioon. Jos kyseinen olio saadaan luotua (eli aika saadaan kannasta), ajetaan functions-hakemistossa oleva createReminderNotifications-funktio. Tälle funktiolle annetaan parametreiksi juuri luodut $timestamp- ja $lastRun-muuttujat.

Tämä funktio ajetaan tekemällä siitä $createNotifications-olio, jonka onnistuessa config-tauluun asetetaan viimeisin ajoaika, joka nyt on funktion ajossa luodun $timestamp-muuttujan aika. Virheiden välttämiseksi tämänhetkinen timestamp-tehtiin jo tässä kohtaa, eikä esimerkiksi createNotifications-funktiossa, joka olisi kuitenkin saattanut tehdä koodin luettavuutta selvemmäksi, kun parametrin kuljetukselta oltaisiin vältytty.

Tiedostoon alkuun on liitetty shell.php require_once-funktiolla, joka on PHP:n sisäinen funktio. Lopullisesta funktiosta saatiin suhteellisen selvä ja helppolukuinen:

```
<?php // Creates notifications from site reminders using Cron
require_once('shell.php');
function createNotificationsFromReminders() {
    $timestamp = date('Y-m-d G:i:s');
    $lastRun = db_fetch_one("SELECT value FROM mdo_config WHERE item =
'reminder_handler_last_run'");
    if ($lastRun) {
```

```
            $createNotifications = createReminderNotifications($lastRun,
$timestamp);
            if ($createNotifications) {
                $update = db_update("UPDATE mdo_config SET value =
'".$timestamp."' WHERE item = 'reminder_handler_last_run'");
            }
        }
}

createNotificationsFromReminders();
```

6.3.2 Cron

Muistutusten ”pollaamiseksi” eli tarkistamiseksi tarvittiin minuutin välein ajettava Cron-ajo. Cron-ajo tapahtui pitkälti samalla periaatteella, kuin millä ilmoitusten tarkistaminen sovelluksessa tapahtui. Myös muistutukset lisättiin niiden hälytyksen tapahtuessa ilmoitusten tauluun, josta ne tuotiin asiakaspuolelle ilmoituksiin kuin mitkä tahansa ilmoitukset. Cron toimi sovelluksessa - niin tuotannossa kuin kehitysympäristössäkin - virtuaalisessa Linux-ympäristössä, missä käytettiin Cronin käytölle tavalliseen tapaan Crontabia.

Crontabiin asetettiin virtuaalikoneessa minuutin välein tapahtuva ajo crons-per-minute.sh-tiedostoon, jossa oli rivi "*cd /var/www/mdo-hub-server/shell/; php reminder-handler.php*". Tällä ajettiin palvelimen ”shell”-hakemistossa oleva reminder-handler.php, jossa oli lyhyt tarkistusfunktio ajankohtaisista muistutuksista. Tämä funktio teki jatkotoimenpiteet, jos hälyttäviä muistutuksia löytyi.

6.4 Tietokanta

Muistutuksia varten tehtiin kaksi taulua tietokantaan. Näistä ensimmäinen (reminders) piti sisällään muistutusten tiedot ja toinen (reminder_user_groups) sisälsi muistutukselle valitut työntekijät. Molempiin näistä tauluista otettiin yhteys jokaisessa haussa, luonnissa, päivityksessä ja poistossa. Ne olivatkin palvelinpuolen koodissa peräkkäin, ja jos ensimmäinen tietokantafunktio onnistui (esim. muistutusten hakeminen), ajettiin heti perään seuraava tietokantafunktio, eli

käyttäjien/käyttäjäryhmien hakeminen reminder_user_groups-taulusta samaiselle muistutusten oliolle, joka lähetettiin palvelimelta asiakaspuolen Backbone.js:n Collectioniin.

Muistutuksen taulun sisältö oli tiivistettynä seuraavanlainen:

id - muistutuksen ID, tietokantamoottorin lisäämä
job_id - kohteen ID, eli se osio, mihin muistutus on lisätty
creator - lisääjän eli käyttäjän ID
last_modified - viimeisin muokkausaika (tietokantamoottorin asettama)
last_modified_by - viimeisimmän muokkaajan ID
name - muistutuksen nimi
description - muistutuksen kuvaus
time - muistutus- eli hälytysaika muistutukselle
nextReminder - seuraava muistutusaika (luonnin yhteydessä sama kuin hälytysaika)
repeat - toistuvuus (string)
is_group - tekijöiden/ryhmien määrittävä arvo (int)

Osa arvoista asetettiin suoraan parametreista määritellyillä muuttujilla, osa oli tuli tietokannan asettamina, kuten viimeisin muokkausaika. Työntekijät tai työntekijäryhmät sisältävä taulu, reminder_users_groups, piti sisällään kolme kohtaa:

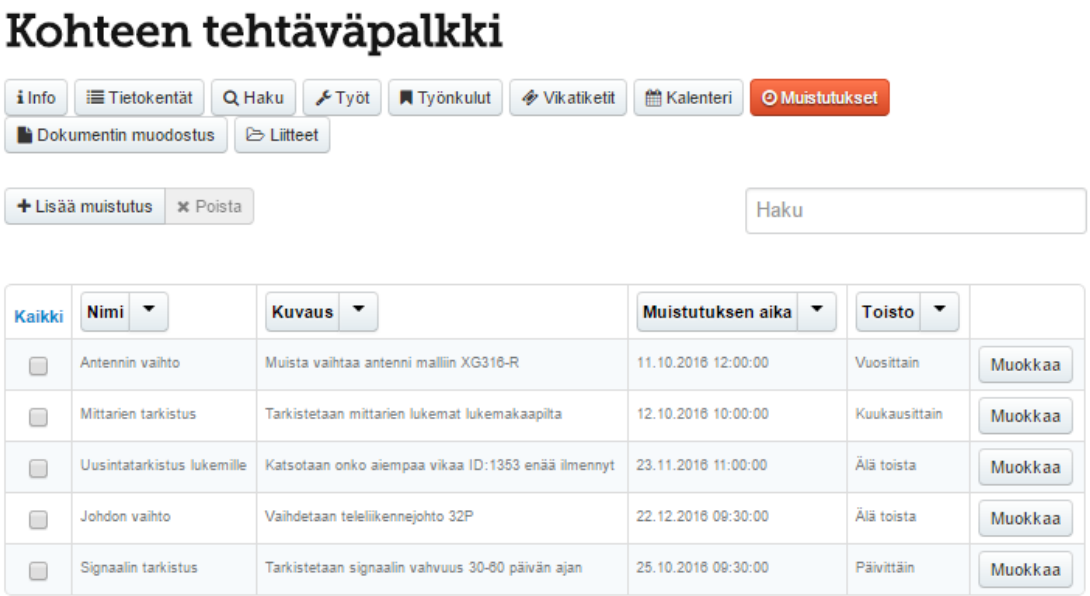
reminder_id - reminders-tauluun tietokantamoottorin antaman ID:n muistutukselle
target_id - käyttäjän tai käyttäjäryhmän ID
is_group - tekijöiden/ryhmien määrittävä arvo

Esimerkiksi jokaisen työntekijän lisääminen loi uuden rivin reminder_users_groups-tauluun, joten oma taulu muistutuksille nähtiin järkeväksi ratkaisuiksi muiden syiden ohella. Suuresta iterointimäärästä huolimatta suorituskyky- tai virhealttiutta ei myöskään ollut havaittavissa.

7 Asiakaspuolen toteutus kokonaisuutena

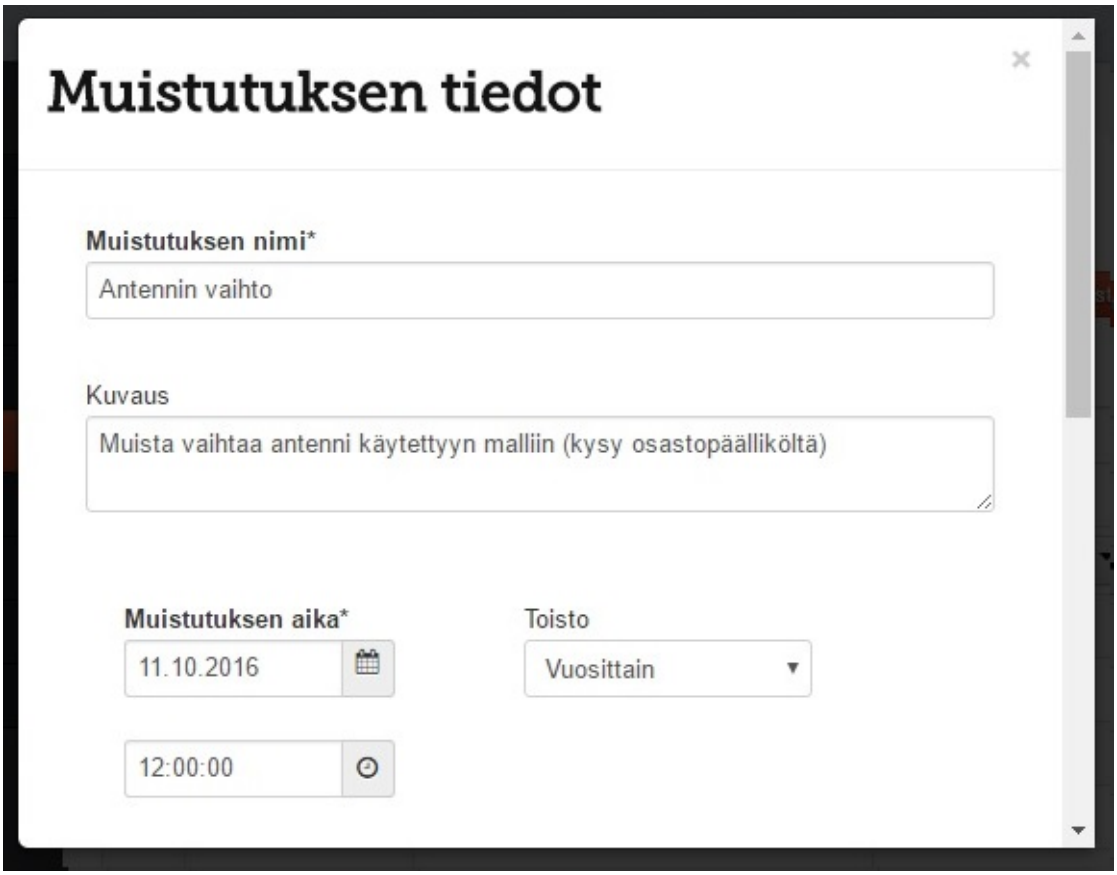
7.1 Hallintasivu

Sovelluksessa luotiin Kohteet-osion alle uusi ominaisuus, muistutukset. Sille luotiin oma välilehtensä, "Muistutukset". Muistutukset vaikuttivat oleellisesti koko sovellukseen ja ne näkyivät lisäksi mm. kohteen kalenterissa. Varsinaisella muistutukset-välilehdellä oli hallintasivu muistutuksille. Siellä niitä voitiin lisätä, muokata, päivittää ja poistaa. Myös niiden yksitellen tapahtuva katselmointi oli helpompaa aukeavan modaalin ansiosta.



Kuvio 6. Muistutusten hallintasivu selainpuolella

Muistutuksia pystyi avaamaan joko riviä klikkaamalla tai Muokkaa-nappia painamalla. Tämän seurauksena avautui modaali-ikkuna, joka näytti yksittäisen muistutuksen tiedot ja antoi muokata sitä.



Kuvio 7. Muistutuksen modaalinäkymä

Kuviossa 7 näkyvän modaalin alaosa näkyy myöhemm tässä luvussa.

7.1.1 Collection

Muistutusten Collection eli kokelma on suhteellisen lyhyt, kuten sovelluksessa mikä tahansa muukin Collection. Siinä keskitytään lähinnä hakemaan kohde-kohtaiset muistutukset kohteen ID:n perusteella. Tämä tehdään fetch-funktiolla, joka saa parametrikseen siteId-muuttujan. Tätä käytettiin siis näyttämään, kaikki kohteelle asetetut muistutukset, eikä vain niitä, joiden hälytysaika olisi ajankohtainen.

Fetch-funktio ajoi $.post-funktion, joka palautti haluttujen muistutusten taulukon. Sovellus käytti yleisesti $.post-metodia myös tiedon hakemiseen, jota sitä ei sopinut muuttaa.

Funktion palauttamat muistutukset laitettiin yksitellen .each:lla olioikseen asiakaspuolella. Niiden arvoihin oli kuitenkin tehtävä muutoksia ja/tai niistä oli

otettava duplikaatti-arvo jotta asiakaspuolen komponentit olisivat toimineet halutusti. Esimerkiksi unix-timestamp oli säilytettävä sellaisenaan, mutta myös ihmisen luettava aikaleima oikeassa aikaformaatissa ja aikavyöhykkeellä piti tehdä. Toinen muutos liittyi käännöksiin, kun sovellusta voitiin käyttää sekä suomeksi että englanniksi. Lopuksi Collection palautti muistutukset Backbone.je:lle tyypillisesti Collectionin lopussa olevalla returnilla.

7.1.2 Model

Muistutusten Collectionissa oli palvelimen kutsuina ainoastaan fetch:n ajaminen, eli kaikkien tarvittujen muistutusten hakeminen. Muistutuksen Model-mallissa on create-funktio uuden muistutuksen luomiseen niin malliin kuin tietokantaankin asti sekä funktio muistutuksen päivittämiseen ja poistamiseen. Save-funktio päivittää datan muistutuksen muuttamisen yhteydessä sekä malliin että tietokantaan ja remove poistaa muistutuksen sekä kannasta että mallista.

Uuden muistutuksen lisäämisen yhteydessä otetaan .then:llä luodun muistutuksen ID, mikä tulee palvelimelta onnistuneen päivityksen yhteydessä. Tämä ID on tietokantamoottorin asettama, eikä sitä voi jättää asettamatta myös malliin asiakaspuolella. Päivittävän save-funktion suurin ero on ehkä se, että se saa $.post-funktion parametriksi muistutuksen ID:n suoraan mallista helposti this.get('id'):llä.

Kuten palvelimen puolella, myös Frontissa muistutuksen poistaminen on suhteellisen yksinkertainen funktio, joka tarvitsee vain muistutuksen ID:n session-avaimen ja actionin ('deleteReminder') lisäksi.

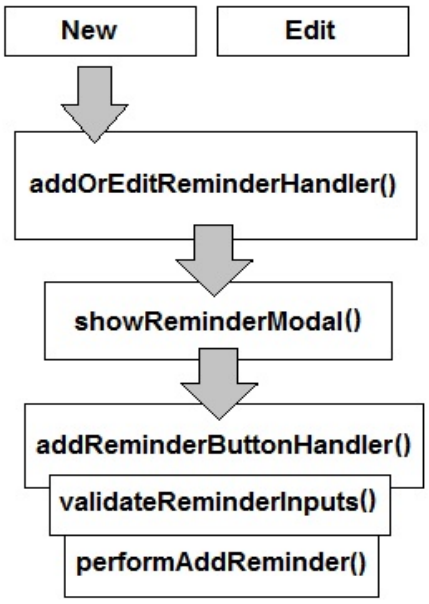
7.1.3 View

Sovelluksessa View on (myös MVP-frameworkeille tyypillisesti) suhteellisen massiivinen myös tämän projektin yhteydessä. Se alkaa määrittelemällä moduulit, kuten Collectionit ja Modelit ja muut komponentit sekä templaatit. Varsinaisen function-alustuksen jälkeen (johon laitetaan edellä mainitut komponentit

parametreiksi halutuilla nimillään), tulee näkymän eli View-olion luonti. Siinä määritellään käyttöliittymän elementit, kuten template (RemindersTemplate), Modaali-template sekä vahvistusdialogi-modaali. Nämä ovat pitkälti Bootstrapin muokattuja komponentteja. Näiden jälkeen tehdään eventtien eli tapahtumien määritykset. Niitä ovat jokainen click-, change- ja keyup-tapahtuma. Tapahtuma yhdistetään elementtiin (esimerkiksi tiettyyn painikkeeseen tai kenttään) sekä funktioon, jotka tästä View-tiedostosta löytyvät.

Koska sovellus toimii kahdella eri kielellä, tarvitaan käännökset templaatteihin. Ne palautetaan getModalTemplateData-funktiossa suoraan Localization-tiedostoista. Backbone.js:lle tyypilliseen tapaan näkymä alkaa initialize-funktiolla. Siinä määritellään oleelliset muuttujat halutuiksi sekä taulukko-muuttujat tyhjiksi. Render-funktio renderöi näkymän. Siinä haetaan käyttöliittymän templaattien data (mm. kielikäännökset ja oleelliset muuttujat kuten "admin") sekä varmistetaan, ettei aiemmin valittuja painikkeita ole enää valittuina.

Kuvassa muistutuksen lisääminen funktioissa:



Kuvio 8. Lisäysfunktion rakenne

Koska muistutuksen lisääminen sekä päivittäminen tehtiin samalla modaalilla, oli niiden toiminallisuus myös mahdollista tehdä lähtökohtaisesti samoissa funktioissa. Tämän funktio tosin jaettiin selkeydenkin lisäämiseksi useampaan kohtaan, joista ensimmäinen on addOrEditReminderHandler-funktio. Siinä on vain event-oliolle tehtävät preventDefault- ja stopPropagation-metodit. Näiden jälkeen tulee funktio-kutsu showReminderModal-funktioon.

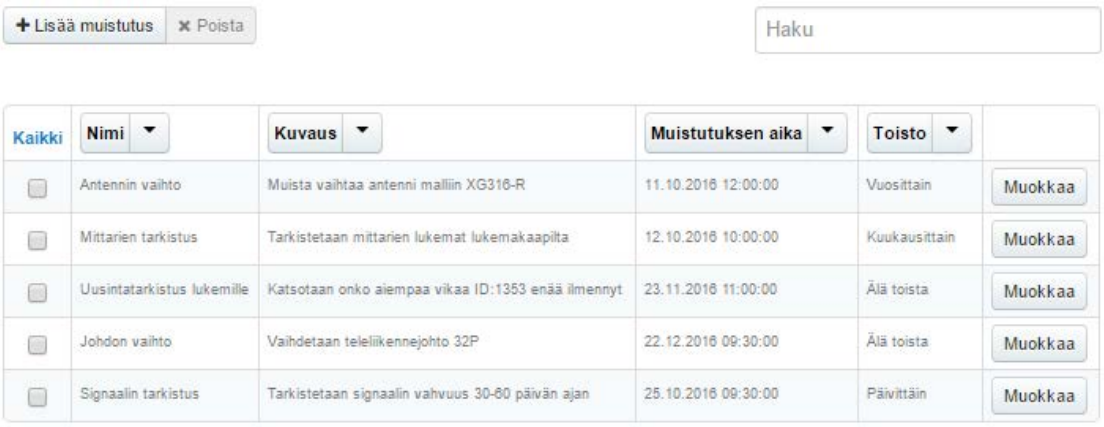
showReminderModal-funktio asettaa tai on asettamatta muuttujat kenttiin riippuen siitä, onko kyseessä uuden muistutuksen luonti vai muistutuksen päivittäminen. Funktiossa myös käytetään näkymään liitettyjä komponentteja kuten DataTimePicker-komponenttia, sekä tehdään validaatio kentille, sekä UserGroupSelector-komponenttia, jolla työntekijät/työryhmät voidaan valita muistutukselle.

ValidateReminderInputs-funktio ajetaan keyup-eventtien jälkeen ja se tarkistaa, onko muistutuksen nimi ja muistutusaika asetettu. Puuttuva tieto aiheuttaa Bootstrapille tyypillisen punaisena palavan kentän eikä aktivoi lisäys-painiketta modaalissa. Varsinainen lisääjä-funktio on performAddReminder. Se käsittelee suuren määrän attribuutteja vaikuttaa mm. modeliin, joka lähettää pyynnöt palvelimelle. Tämän funktion käsittelijänä addReminderButtonHandler-funktio. Loput funktiot ovat mm. selectCheckboxChanged-funktio checkboxien tarkistamiseen, getResponsiveListData responsiveList-komponentin käyttämiseen sekä muita clickHander-funktioita. Näkymässä oli 400 riviä, mikä suhteellisen suuri määrä, mutta ei suurimmasta päästä sovelluksen isoimpiin näkymiin verrattuna.

7.2 Käytetyt Backbone.js:n komponentit

7.2.1 ResponsiveList

Responsivelist oli Viewin eli näkymän osa, joka osasi "populoida" listan Collectionin avulla.



Kuvio 9. ResponsiveList

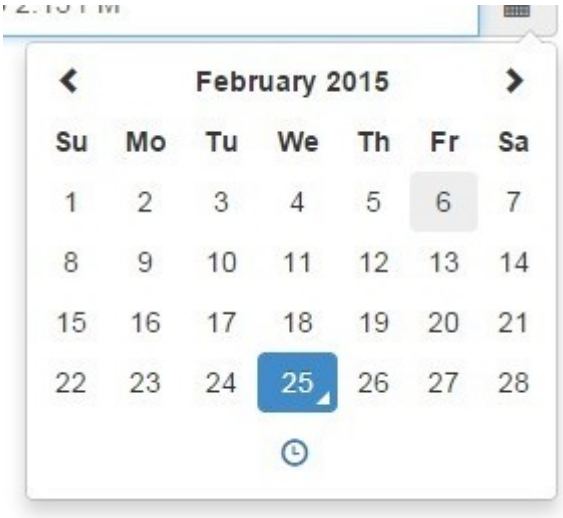
Responsivelist oli käytössä muissakin sovelluksen näkymissä sellaisenaan, mutta muistutuksia varten tähän lisättiin jokaiselta riviltä Muokkaa-painikkeesta aukeava modaali-ikkuna, jossa muistutusta voitiin muokata. Tässä jouduttiin muokkaamaan template-tiedostojen lisäksi myös Responsivelistin View-tiedostoa, mikä oli huomattavan mitäänsanomaton ja vaikeastiluettava.

Hakukenttä on oleellinen osa responsivelist-komponenttia. Se oli altis hajoamaan ja jättämään suodattamatta tuloksia tiettyjen muutosten jälkeen, mutta toimi lopullisessa versiossa täydellisesti.

Responsivelistissa oli myös lajittelufiltereiden valinta. Siinä voitiin järjestää lista esimerkiksi hälytysajan mukaan.

7.2.2 DateTimePicker

Datepicker on Twitter Bootstrapin suosittu päivämäärän ja kellonajan valitseva komponentti. Sovelluksessa Datepicker käytti Moment.js-kirjastoa, minkä toiminta edellytti oikeanlaisen aikaformaatin käyttöä. Datepicker toimi parhaiten niin, että unix-timestamp muutettiin muutettiin Moment.js-kirjastolla nykyiselle aikavyöhykkeelle ja aikaformaattiin.



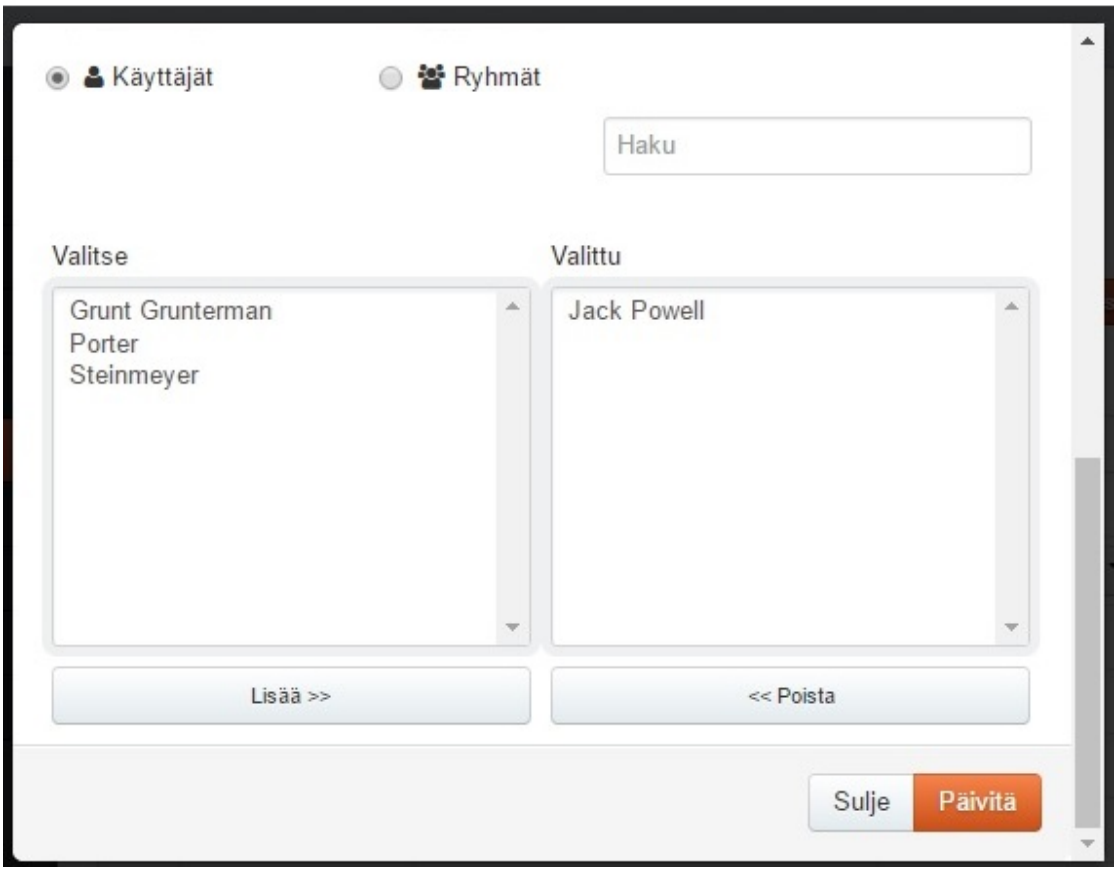
Kuvio 10. DateTimePicker

Tässä oli eroja Chromen ja Firefoxin välillä, kun Firefox ei toiminut vaikka Chromen toimikin oikein. Tämä korjaantui kertomalla unix-timestamp tuhannella millisekunnilla, jolloin molemmat selaimet tulkitsivat sitä samalla tavalla. Aika oli yksi niistä attribuuteista, joita modeliin oli tarpeen laittaa kaksi eri muodoissa Collectionin fetch-vaiheessa, jotta ne toimisivat eri kaikissa komponenteissa.

Aikaformaatti päiväyksessä oli sovelluksessa Suomessa käytetty DD.MM.YYYY, eli päivät, kuukaudet, vuodet pisteellä erotettuna. Aikaformaatti kellonajoissa oli HH:mm:ss eli tunnit, sekunnit minuutit kaksoispistein erotetulta.

7.2.3 GroupSelector

Muistutukset haluttiin niin sanotusti allokoida eri käyttäjille tai käyttäjäryhmille. Muistutukset oli yksi nistä asioista sovelluksessa, joille voitiin määrätä käyttäjät eli työntekijät.



Kuvio 11. GroupSelector

Groupselector-komponentin pystyi ottamaan muualta sovelluksesta suhteellisen helposti. Nimestään huolimatta se valitsi käyttäjien tai käyttäjäryhmien perusteella työntekijät, jotka se osasi käyttäjien Collectionista listata valittaviksi.

Komponentin esivalitut käyttäjät oli mahdollista määrittää Viewin koodissa. Siinä taulukkomuuttujaan "preSetUsers" voitiin lisätä jo valmiiksi esimerkiksi oma ID, minkä pystyi määrittämään koko näkymän alussa globaalilla metodilla sovelluksessa. Näin oma nimi oli jo valittujen puolella automaattisesti. isGroup-muuttujan arvo määritti sen, valitaanko käyttäjiä vai käyttäjäryhmiä muuttaen activeUsersType-muuttujaa. Myös tässä komponentissa oli toimiva hakukenttä niille tapauksille, että jos käyttäjiä oliskin ollut suurempi määrä.

8 Toteutustavan hyödyt

Luvussa pohditaan niin palvelinpuolella kuin selain-/asiakaspuolella tehtyjen toteutusten hyötyjä etenkin sen kannalta, mitä sovelluksen jatkokehitys tulee olemaan. Yksi tutkimuskysymyksistä otti huomioon ominaisuuden integroimisen sovellukseen,

ja sen onnistuminen on tärkeää tämän kysymyksen ja sen ratkaisun onnistumisen kannalta.

8.1 Toteutus palvelinpuolella

Sovelluksella oli melko omintakeinen palvelin-toteutus. Sovelluksen PHP-palvelin ei käyttänyt esimerkiksi mitään tunnettua PHP-frameworkia, joten niiden osaamisesta ei ollut merkittävää hyötyä. Kehitystyössä oli perehdyttävä niihin tapoihin, joilla palvelinpuoli oltiin toteutettu. Esimerkiksi siinä käytettiin omia funktioita tietokantakyselyihin. Yksinkertainen, esimerkiksi yhden henkilön lisääminen voitiin toteuttaa db_update_one()-funktiolla, jonka funktioparametriksi annettiin haluttu SQL-lause. Tämä palautti esimerkiksi lisätyn uuden työntekijän ID:n vastauksena, joka voitiin sitten laittaa asiakaspuolella käyttäjän ID:ksi modelissa, kuten muistutusten lisäämisen ja modeliin päivittämisen kanssa tehtiin. Pyrin toteuttamaan palvelinpuolen osuuden niin, ettei uusia funktioita olisi tarvinnut lisätä Backendiin esimerkiksi tietokantatoimintoja varten vaan vanhoja yritettiin hyödyntää parhaan mukaan.

Sovelluksen palvelinosuus on suhteellisen yksinkertainen, ja yhtenä tavoitteena ja tutkimuskysymyksenä olikin hyödyntää sitä mahdollisimman minimalistisesti. Monimutkaisempi logiikka liittyikin enemmän Frontendiin ja siinä toimivan Backbone.js:n hyödyntämiseen sen tarkoitusperiaatteiden mukaisesti. Palvelinpuolen toteutuksena hyötyinä voidaan pitää yksinkertaisuuden kunnioittamista ja säilyttämistä, eli aiempaa yksinkertaista toteutusta palvelinpuolella ei tarvinnut tässäkään tapauksessa muuttaa. Tämä etenkin siksi, että sovelluksen asiakaspuolella riitää haasteita sen laajuuden ja jossain määrin vaikean hallittavuudenkin vuoksi.

8.2 Toteutus asiakaspuolella

Sovelluksen asiakaspuolella on monellakin tapaa melkoinen vastakohta palvelinpuolelle. Asiakaspuoli on laajuudeltaan huomattavan suuri. Se jakautuu Model-, Collection- ja View-kansioihin ja niiden sisältämät JavaScript-tiedostot myös oltiin nimetty harmillisen samankaltaisesti, joten tiettyjen IDE-ohjelmien kanssa niiden käytössä oli hankaluuksia. Pelkkä "reminders.js"-tiedosto editorin viimeaikaisten tiedostojen listauksessa ei kertonut, että oli tiedosto osa Modelia,

Collectionia vai Viewiä, vaan pahimmassa tapauksessa asian joutui selvittämään katsomalla, mihin kansioon sen polku viittasi. Keskustelua käytiinkin siitä, voisiko kaikkien tiedostojen nimeen mahdollisesti laittaa Model-, Collection- tai View-pääte tai muu tunniste jossain vaiheessa tulevaisuutta.

Sovellusta oli aikojen saatossa ollut kehittämässä useampi kehittäjä ja kukin oli tehnyt sitä omalla tavallaan. Joissain tapauksissa oltiin selvästi käytetty järkeviä, enemmän Backbone.js:lle tyypillisiä toteutustapoja, mutta joissain osa-alueissa asiat oltiin tehty enemmän "spagettikooodimaisesti". Pahimmassa tapauksessa tämä toi ilmi sen tosiasian, että tietyn aluksi yksinkertaiselta vaikuttaneen asian korjaaminen oliskin ollut huomattavan suuri projekti, minkä tekemiseen olisi voinut kulua tarpeettoman paljon aikaa. Nämä asiat oli usein vaan yritettävä kiertää sekoittamatta koodia yhtään sen enempää.

Asiakaspuoli toimii Gruntilla Vagrant-sovellusäiliön sisällä, joten muutokset koodiin näkyivät kun selain päivitetään, eli sovelluksen uudelleenkäynnistystä ei tarvita. Tämä helpotti etenkin debuggaamista ja esimerkiksi sitä, kun oli tarpeen käyttää console.log()-funktiota sen selvittämiseen, mikä kohta koodissa ajettiin tietyllä sivunlatauksella. Tätä saikin tehdä huomattavan usein, oli kehittäjänä sitten kauemmin projektin parissa työskennellyt kehittäjä tai uudempi, sovellukseen vasta perehtyvä henkilö.

Koska palvelinpuolella ei oltu toteutettu MVC-mallia, voitiin sitä toteuttaa Asiakaspuolella sillä tavalla, millä Backend.js on sitä suunniteltu toteuttamaan sovelluksessa. Sovellus oltiin jaettu Model- Collection- ja View-kansioihin asiakaspuolella. Asiakaspuolen toteutuksen hyötyinä voidaan pitää sitä, että sovellukselle tietynlaista säännönmukaisuutta voitiin tässäkin tapauksessa ylläpitää. Asiakaspuolelta löytyy muistutusten ominaisuudet suhteellisen samanlaisena kuin muutkin sovelluksen ominaisuudet, kuten esimerkiksi muistutuksen "isäntä-osio", eli kohteet tai toinen osa sovellusta; tilaukset ja laskut. Kehittäjä voi olettaa löytävänsä halutut asiat niille tarkoituista hakemistoista.

Jos Asiakaspuolen lisäksi myös palvelinpuolelle oltaisiin toteutettu esimerkiksi MVC- tai muu malli, olisi näiden kokonaisuus voinut kasaantua liian monimutkaiseksi. Toisaalta, jos kumpaankaan ei oltaisi mitään järjestyksellistä rakennetta toteutettu, olisi se voinut olla vielä sovelluksen ylläpitoa huomattavasti haittaava asia.

Asiakaspuolen toteutuksen hyötyinä on sovellukselle tyypillisen arkkitehtuurin vaaliminen. Lähtökohtaisesti asiakaspuolen kehittämisen periaatteet työssä olivat järkevämpiä kuin vanhemmat asiakaspuolen toteutukset sovelluksessa. Tämä johtui etenkin siitä, että tiimissä haluttiin asiakaspuolta kehitettävän parempien periaatteiden mukaisesti. Myös huomattava määrä sovelluksen vikatikettien aiheuttamista bugeista näytti sijaitsevan niillä osa-alueilla, joilla arkkitehtuurinen malli oli heikompaa.

9 Ongelmat ja haasteet

Kehittämistyössä ongelmailta ei vältytty. Ongelmia ja oikeita pulmatilanteita esiintyi kaikilla osa-alueilla; niin palvelinpuolella, asiakaspuolella kuin tietokannassakin. Monet näistä liittyivät puhtaasti siihen, ettei tietyn asian toiminnasta, kuten Backbone.js:n komponentin toiminnasta, ollut tarpeeksi taustatietoa. Se, että tietty komponentti näytti tietyn kellonajan ja päivämäärän oikein, saattoi tarkoittaa että toinen komponentti ei ymmärtänyt sitä käytännössä lainkaan. Tällaisissa tilanteissa piti miettiä tarkkaan, käyttäisikö toista lähestymistapaa samassa kokonaisuudessa (valitut komponentit) vai joutuisiko luopumaan jo tehdystä toteutuksesta menettäen siihen käytetyn siihen astisen tuloksen.

Vastaavissa tilanteissa saattoi tuntea oivaltaneensakin. Jos toinen komponentti ei ymmärtänyt aikaleimaa tietyssä formaatissa, kannatti samaan olioon tehdä aikaleima myös toisessa formaatissa olioiden alustusvaiheessa. Tällöin saattoi joutua päivittämään kahta eri attribuuttia ja pitämään huolta että molemmat olivat oikeassa, mutta sovelluksen toiminta ainakin oli varma. Sama ratkaisutapa toimi käännösten kanssa. Tietyissä tapauksissa jouduttiin sovelluksen "kaksikielisyyden" takia käyttämän attribuutti-arvoa, joka ei olisi merkkijono "Älä toista" tai "Don't repeat", eli se saattoi olla "DontRepeat". Tällä pystyi saamaan käännökset toimimaan sen lisäksi, että arvoi pystyttiin tuollaisena asettamaan suoraan tietokantaan. Tämä oli esimerkki ongelmista, joita asiakaspuolella saattoi kohdata.

Myös pitkään pyöritelty ongelma palvelinpuoleen ja tietokantaan liittyen oli se, ettei tietokannassa olevan unix-aikaleiman aikavyöhykkeestä ollut varmuutta. Käytännössä saattoi luulla, että Unix-aikaleima olisi väärässä vain siksi, ettei olettanut tietokantamoottorin jo kääntäneen sitä omalle aikavyöhykkeelle.

Unix-aikaleimat ovat aina kolme tuntia jäljessä Suomen aikaan verrattuna, mutta koska SQL-moottori käänsi sen mitään sanomatta, sekoitti se pakkaa melkoisesti. Tämä yhdistettynä siihen, ettei tietokanta antanut aikaleimaa Unix-timestampina ilman oikeanlaista SELECT-lausetta, junnasi homma tässä kohtaa ikävän kauan. Lopulta asiat käsitti ja projekti eteni.

Kolmas mieleen jäänyt ongelma tuli palvelimelta liittyen siihen, ettei Cron saanut haettua oikeita muistutuksia tietyn aikavälin sisältä. Tämän selvittäminen sisälsi huomattavan määrän lokitusta Linux-ympäristön debug-konsoliin, kunnes lopulta ymmärsi että aikaväli ei ollut loogisesti toimiva. Huomionarvoista oli myös se, ettei lokaali ympäristö saanut lähetettyä sähköpostia. Tämä tarkoitti sitä, että funktioketju saattoi katketa hiljaisesti epäonnistuneeseen sähköpostinlähetykseen. Tätä asiaa sai selvitellä huomattavan kauan ennen kuin asian ymmärsi. Lopuksi korjasin try-catchin sähköpostinlähetysfunktioon muiden pyynnöstä. Näin koettiin haasteita niin asiakaspuolella, aina palvelinpuoleen sekä tietokantaankin asti. Jokaisen näistä voisi sanoa olleen hyvinkin opettavainen.

10 Tutkimustulokset

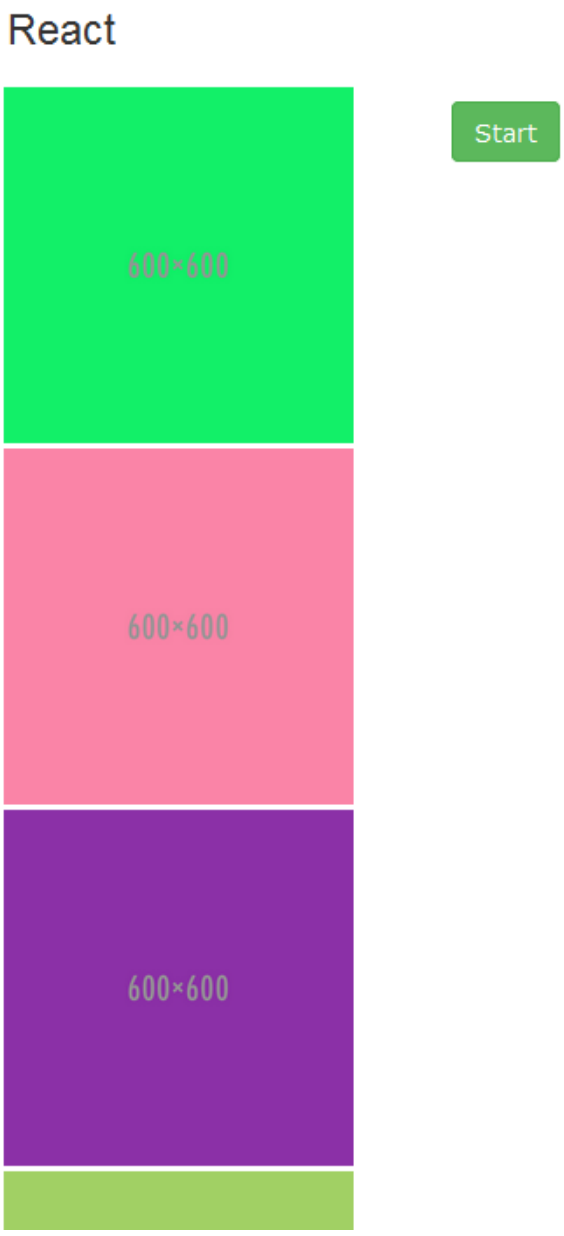
Luvussa käydään läpi ne tutkimuskysymykset, joita varten tehtiin tutkimustyötä ilman varsinaista kehitystyötä. Näissä tutkimuskysymyksissä tutkittiin muun muassa mahdollisuutta korvata Backbone.js-kehyksen View- eli näkymä-osa React-kirjastolla, eli rakentaa virtuaalinen DOM Reactin avulla. Tätä tutkittiin etenkin suorituskyvyn näkökulmasta, sillä Backbone.js:n View on varsin tavanomainen, perinteisellä DOM:lla toteutettu näkymä. Myös React-kirjaston käytön tuomia sovellusarkkitehtuurin muutoksia tutkittiin.

10.1 React-kirjaston nopeuden testaaminen renderöinnissä

Koska perinteistä DOMia haluttiin verrata virtuaaliseen DOMiin, tehtiin testi, jossa testattiin virtuaalista DOMia ja Backbone.js:n perinteistä DOMia sekä Angular.js:n omaa, ei-virtuaalista DOMia. Testissä renderöitiin sama lista Reactilla, jossa virtuaalinen DOM sekä Backbone.js:llä ja Angular.js:llä, joissa on perinteinen DOM. Testissä renderöitiin 6 eriväristä 600*600 pikselistä ruutua satunnaisessa

järjestetyksessä allekkain. Kuvien koko vaihdettiin 200*200 pikseliin tilan säästämiseksi.

Lista aukesi nappia painamalla ja aika laskettiin painalluksesta millisekunneissa. Aika otettiin renderöinnin alusta sen päättymiseen. Ensimmäisessä testissä rivejä renderöitiin 250, toisessa 500 ja viimeisessä 5000. Selaimina käytettiin Firefox- sekä Chrome-selainta, joilla otettiin keskimääräinen aika 10 latauksen perusteella.

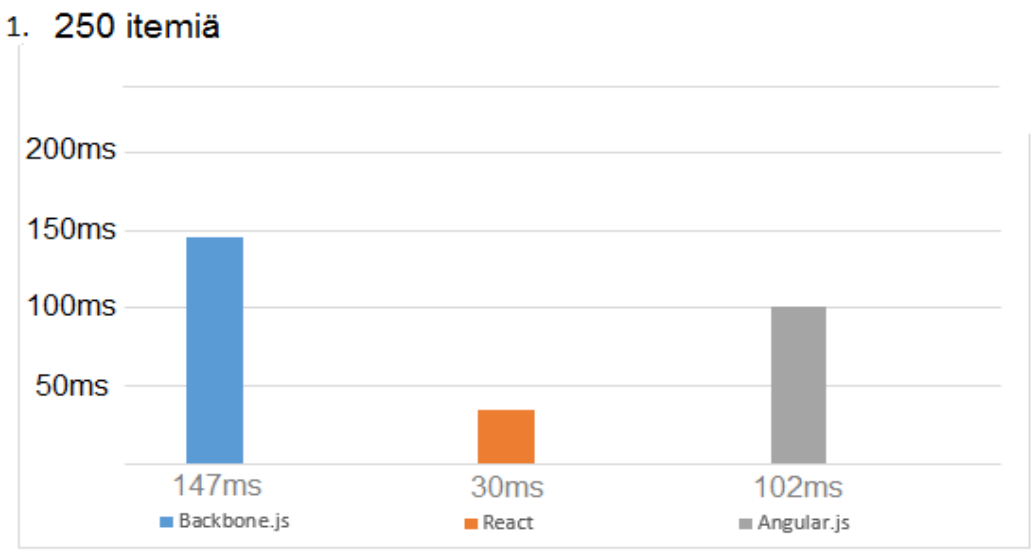


Kuvio 12. Renderöity näkymä

Aluksi itemeitä renderöitiin 250, sitten 500 ja lopuksi 5000. Näistä otettiin keskimääräinen aika 0 latauksen perusteell. Koska Googlen Chrome ja Mozillan

Firefox olivat kehitystyössä ne selaimet, joilla MDO-sovelluksen selainversiota testattiin, päätettiin tämäkin testi suorittaa näillä selaimilla.

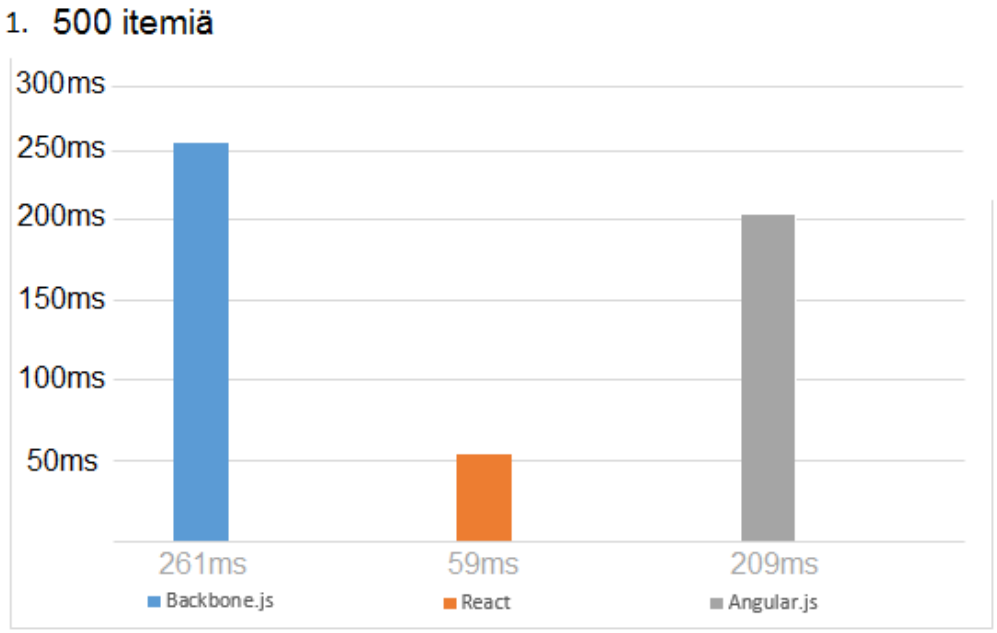
10.1.1 250 itemiä



Kuvio 13. 250 itemin renderöinti

250 on sovelluksessa lähtökohtaisesti suhteettoman suuri renderöitäväksi yhdelle sivulle. Joissakin tapauksissa sovelluksen käyttäjä voi olla tilanteessa, missä hän haluaa renderöitäväksi esimerkiksi isomman tulosjoukon haun perusteella.

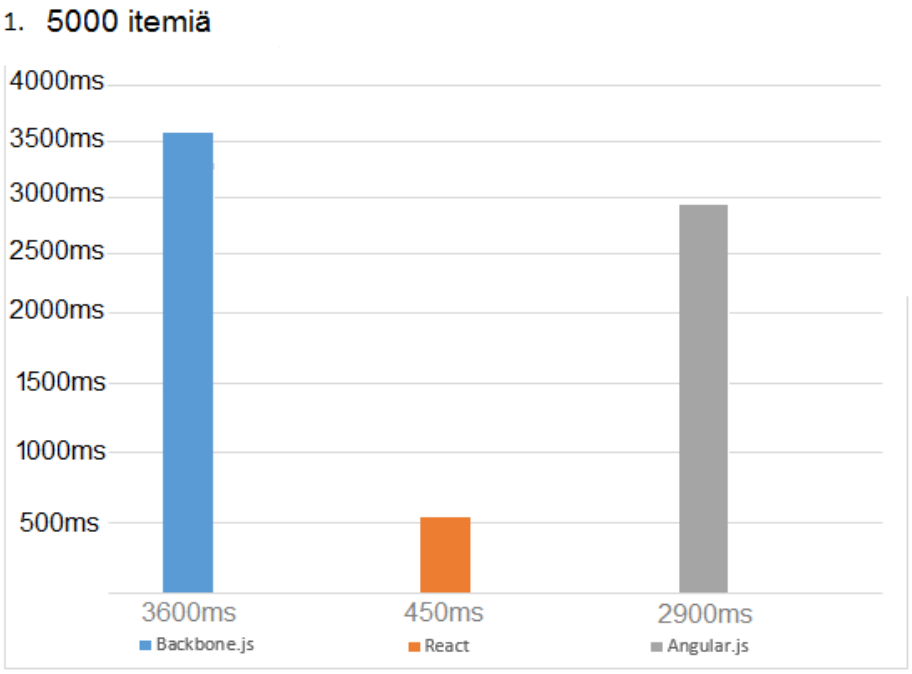
10.1.2 500 itemiä



Kuvio 14. 500 itemin renderöinti

500 oli odotetusti sitä, mitä saattoikin odottaa. Kaksinkertainen renderöintimääräkesti kaksinkertaisesti ajan. Viisisataa itemiä yhdessä näkymässä on kuitenkin jo jotain, mitä ei tänä päivänä useassa sovelluksessa näe renderöitävän.

10.1.3 5000 itemiä



Kuvio 15. 5000 itemin renderöinti

Suhteettoman suuressa määrässä selvisi se, että oikeilla tekniikoilla voidaan onnistua käytännössä missä vaan. React:lla renderöitäessä aikaa kului vain alle puoli sekuntia, mikä tekee viiveestä vain juuri ja juuri silmin nähtävää. Muilla tekniikoilla lataus kestäisi turhauttavan kauan ainakin tilanteessa, jossa latausta pitäisi tehdä useamman kerran.

10.2 Suorituskykytestin lopputulosten analysoiminen

Testissä tuli esille hyvin etenkin Reactin kyky käyttää virtuaalista DOMia. PerinteisenDOM:n kanssa renderöinti olisi suhteettoman raskasta jos datamäärä olisi tavanomaista suurempi yhdelle sivulle. Jos kehitettävä sovellus listaa edes jossakin vaiheessa paljon dataa ja etenkin jos sitä halutaan voida esittää rajaton määrä kerralla, on React varteenotettava valinta näkymäkirjastoksi.

10.3 Reactin tuomat muutokset Backbone.js:n kanssa käytettynä

Reactin tuomat muutokset Backbone.js:n kanssa käytettynä Backbone.js on yhteensopiva Reactin kanssa käytettäväksi, jolloin Backbone.js:stä jätetään pois muutama sen näkymällä oleellinen tekniikka. Oleellinen DOM-elementtien manipuloimiseen Backbone.js:ssä käytetty tekniikka - ja jo aiemmin työssä esitelty - jQuery jätetään kokonaan pois, kuten myös vaihtoehtoisesti käytetty Zepto.js. Reactin virtuaalinen DOM tekee näiden käyttämistä periaatteiden vastaista, ellei niillä tehdä ei-funktionaalisia toimintoja, kuten animaatioita. (Burget, 2013)

Backbone.js:n näkymän korvaaminen Reactilla korvaa myös Handlebars.js-templaattikirjaston, jolloin React sisältää kaikki ne attribuutit, joita ennen käytettiin Handlebars-templaateissa. (Lewis, 2015)

11 Yhteenveto ja pohdinta

Tutkimuskysymykset olivat seuraavat:

Miten ominaisuus voidaan kehittää Full stack -mittakaavassa?

- **Miten tämä tehdään tietokantaan palvelinpuolelle sekä asiakaspuolelle?**

Ratkaisuus päädyttiin jäljittelemään aiempia toteutuksia asiakaspuolelta palvelinpuolelle ja kantaan asti siinä tapauksessa, että niissä oltiin tehty järkeviä ratkaisuja. Palvelinpuolella tämä onnistui, johtuen palvelinpuolen jo aiemman toteutuksen yksinkertaisuudesta, jota ei haluttu muuttaa.

Tietokannan kohdalla muutoksia tehtiin niin, että lopputulos näytti perinteisemmältä relaatiotietokannan toteutukselta. Aiemmissa tietokantatauluissa sovelluksessa oltiin saatettu käyttää JSON-dataa yhdessä SQL-tietokannan taulun sarakkeista. Kehitystyössä tehtiin tämän sijasta kaksi eri taulua, joten em. menetelmää ei tarvittu. Asiakaspuolella haluttiin keskittyä tekniikan (Backbone.js) oikeaoppiseen käyttämiseen. Sovelluksessa oltiin käytetty Backbone.js:ää monella eri tavalla useamman eri kehittäjän toimesta, mikä teki hajotti sovelluksen yhtenäisyyttä myös toiminnan eri osa-alueilla. Tässä kehityössä Backbone.js siis sai erityishuomiota ja lopputulos näyttääkin Backbone.js-kehityksen näkökulmasta hyvin loogiselta ja oikeaoppiselta.

Kehitystyön onnistuminen jo itsessään kertoo siitä, että tutkimuskysymykseen saadun tuloksen validius on toimiva. Koska kehitystyön lopputulos on yrityksen hyväksymä kaikilla kehitystyön osa-alueilla, on syytä olettaa, että tulkinnan sisäinen validiteetti toteutui eikä ristiriitaisuutta ilmennyt. Suurin valideetin varmistaja on siis ominaisuuden siirtyminen sovelluksen tuotantoversioon täysin suunnitelmien mukaisesti.

Miten pidetään toteutus linjassa sovelluksen aiempien toteutusten kanssa?

- **Miten arkkitehtuurinen toteutus saadaan pysymään linjassa aiempien toteutusten kanssa?**

Tähän vastaus saatiin yksinkertaisesti perehtymällä sovelluksen aiempiin, hyväksi todettuihin osa-alueisiin. Niistä mallia ottamalla saatiin sovelluksen toiminnan periaatteita avattua erinomaisesti. Kun aiemmista toteutusmalleista ei poikettu - eikä

niitä varsinkaan muutettu - saatiin uusi ominaisuus loogiseksi osaksi sovellusta. Tietyissä vaiheissa oli selvää, että tiettyä asiaa ei tehdä kuten aiemmin oltiin tehty, ja tarvittaessa jopa korjattiin aiempaa toteutusta sellaiseksi, että kehitystyötä voitiin jatkaa järkevästi. Tutkimuskysmyksen validiutta vahvistaa omalla tapaa se, että muuta sovellusta saatettiin jopa muokata tämän ominaisuuden yhteydessä, mikä tekee lopullisesta yhteisestä kokonaisuudesta samanlaisen.

- **Miten käyttöliittymän toiminallisuus saadaan pysymään linjassa muiden sovelluksen toimintojen kanssa?**

Loogisin ratkaisu tähän oli yrittää käyttää niitä käyttöliittymän komponentteja, joita oltiin käytetty muuallakin sovelluksessa. Tosin tämä toi melkoisia haasteita. Koska aiemmin sovelluksessa oltiin käytetty JSON-dataa, piti kehitystyön ei-JSONia yrittää muokata komponenteissa toimivaksi. Vaihtoehto olisi ollut poiketa sovelluksen yhtenäisestä linjasta, mitä ei haluttu. Haasteet ohitettua nämäkin komponentit saatiin toimimaan sovelluksessa aivan kuten haluttiinkin.

Tässä tutkimuskysymyksen valideetti selviää suurelta osin tutkimalla uuden sovelluksen toimintaa ja pohtimalla, onko lopputulos linjassa muun sovelluksen kanssa. Kehitystiimin testien perusteella ominaisuus oli koko sen kehityskaaren ajan haluttujen raamien sisällä, mikä vahvistaa valideettia selvästi.

Voidaanko Backbone.js:n View eli näkymä korvata React-kirjastolla?

- Mikä on React-näkymän suorituskyky verrattuna Backbone.js-näkymään?

Tutkimuskysymyksessä pohdittiin, toisiko React-näkymä sovelluksessa parannusta näkymän renderöintiin. Tutkimuskysymystä päätetiin testata käytännössä suorituskyky testillä, johon valittiin oleelliset tekniikat. Testissä React-näkymä sai odotettua paremmat tulokset Backbone.js-näkymään verrattuna. Tämä testitulos onkin selvä vastaus siihen, että React-näkymä suoriutuu reilusti Backbone.js-näkymää paremmin renderöinnissä.

Testin reliabiliteettia vahvistaa lähtökohdat, jossa kaikki testisovellukset olivat samassa lähtökohdassa. Testejä tehtiin useampia mahdollisimman neutraaleissa olosuhteissa. Reliabiliteetti vahvistuu myös käytettyjen testiselainten oleellisuudella sekä lukumäärällä (ei pelkästään yhdellä selaimella suoritettu testi) sekä tietysti keskiarvon ottamisella testitulosten ajoista.

- Mitä muutoksia tämä toisi sovelluksen arkkitehtuuriin?

Tutkimuskysymys selvitti, miten asiakaspuolen Backbone.js-sovelluskehyksen View- eli näkymäosuus muuttuisi muutoksen myötä. Jos Backbone.js käytti esimerkiksi Handlebars-templaattikehystä, sen korvaaminen Reactilla oli todennäköisesti edessä. Toinen Reactin alta pois siirtyvä oli DOMia manipuloiva jQuery tai sen korvike Zepto.js. Koska kysymykseen haluttiin käytännössä vain esimerkkejä mahdollisista muutoksista, ei tässä tarvinnut reliabiliteettia erityisemmin vahvistaa.

Tutkimustyö React-kirjaston kanssa antoi viitteet siitä, että jatkokehitys React-kirjaston käytöllä sovelluksen näkymänä voisi olla varteenotettava vaihtoehto. Tekniikka on tämän hetken edistyneimpiä tekniikoita sovelluskehityksessä, joten sen käyttö voisi tehdä nykyistä sovellusta modernimmaksi. Jatkotutkimuksen kannalta kokeilemisen arvoista voisikin olla jonkin varsinaisen sovelluksen osan vaihtaminen React-kirjastolla käytettäväksi varsinkin nyt, kun viitteet sen mahdollisista eduista on saatu.

Lähteet

Anacron. 2015. Tutorialspoint. Viitattu 18.10.2016. http://www.tutorialspoint.com/unix_commands/anacron.htm.

Bailey, D. 2011. Backbone.js is not an MVC framework. Lost-Techies.com. https://los-techies.com/derickbailey/2011/12/23/backbone-js-is-not-an-mvc-framework/. Viitattu 23.9.2016.

Barab, S. & Squire, K. 2004. Design-based research: putting a stake in the ground. The Journal of the Learning Sciences 13, 1, 1 - 14.

Bautista, N. 2010. A beginner's guide to design patterns. Viitattu 19.10.2016. https://code.tutsplus.com/articles/a-beginners-guide-to-design-patterns--net-12752.

Blaze, X. 2008. DOM JavaScript-webkehityksessä. Viitattu 15.11.2016. http://www.ohjelmointiputka.net/oppaat/opas.php?tunnus=dom.

Buckler, C. 2015. SitePoint. Viitattu 13.10.2016. https://www.sitepoint.com/sitepoint-smackdown-php-vs-node-js/.

Burget, J. 2013. Backbone to React. Viitattu 13.11.2016. http://joelburget.com/backbone-to-react/.

Crockford, D. 2008. JavaScript. The good parts. California: O'Reilly Media.

Corliss, D. 2016. How to run a cron job every 5 minutes. Viitattu 14.10.2016. https://support.asperasoft.com/hc/en-us/articles/216127358-How-to-run-a-cron-job-every-5-minutes.

Dausinger, M. 2015. 10 weeks of node.js after 10 years of PHP. Viitattu 12.10.2016. https://medium.com/unexpected-token/10-weeks-of-node-js-after-10-years-of-php-a352042c0c11#.tyvlqw18j.

Design patterns. 2013. Viitattu 23.10.2016. https://sourcemaking.com/design_patterns.

Fulcher, S. 2014. Redweb. Viitattu 23.10.2016. https://www.redweb.com/agency/blog/2014/january/design-patterns-part-one-brief-introduction.

Freed, R. 2011. Virtual DOM. Viitattu 15.11.2016. https://medium.com/tony-freed-consulting/what-is-virtual-dom-c0ec6d6a925c#.6wryb2o7t.

Gentz, M. 2016. NoSQL vs SQL. Viitattu 16.10.2016. https://azure.microsoft.com/en-us/documentation/articles/documentdb-nosql-vs-sql/#nosql-vs-sql-comparison.

Hannah, J. 2015. Choosing the right JavaScript Framework for the Job. Lullabot. Viitattu 23.9.2016. https://www.lullabot.com/articles/choosing-the-right-javascript-fra-mework-for-the-job.

Intro to Cron. 1999. Unixgeeks.org. Viitattu 29.10.2016. http://www.unixgeeks.org/security/newbie/unix/cron-1.html.

JavaScript HTML DOM. 2010. Viitattu 13.11.2016. http://www.w3schools.com/js/js_htmldom.asp.

Kananen, J. 2016. Kehittämistutkimuksen kirjoittamisen käytännön opas. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Lewis, B. 2015. Integrating React with Backbone . Viitattu 12.11.2016. https://blog.engineyard.com/2015/integrating-react-with-backbone.

Moilanen, J. 2002. Enemmän irti SQL-tietokannoista. MikroPC 7.12.2002. Viitattu 13.10.2016. http://mikropc.net/nettilehti/pdf/pc1010200258.pdf.

MVC architecture. 2014. Viitattu 25.10.2016. https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture.

Node.js vs PHP - The Workshape Smackdown!. 2015. Viitattu 15.10.2016. http://blog.workshape.io/node-js-vs-php-the-workshape-smackdown/.

NoSQL databases explained. 2013. Viitattu 29.10.2016. https://www.mongodb.com/nosql-explained.

Papou, A. 2013. What is the difference between MVC and MVP?. Viitattu 27.9.2016. https://dzone.com/articles/how-not-fall-trap-using-mvc-or.

Pastor, P. 2010. MVC for Noobs. Viitattu 29.9.2016. https://code.tutsplus.com/tutorials/mvc-for-noobs--net-10488.

PHP 7 - Performance. 2016. Viitattu 10.10.2016. https://www.tutorialspoint.com/php7/php7_performance.htm.

PHP 7: Will developers adopt the new version?. 2016. Nestify. Viitattu 9.10.2016. https://nestify.io/blog/php-7-will-developers-adopt-the-new-version/.

PHP: Releases. 2016. Viitattu 7.10.2016. https://secure.php.net/releases/.

PHP: Preface. N.d. Viitattu 7.10.2016. http://fi2.php.net/manual/en/preface.php.

Presentation of JQuery. 2013. JScripters.com. Viitattu 1.10.2016. http://www.jscripters.com/what-is-jquery-presentation/.

Rantala, A. 2005. Web-ohjelmointi. Jyväskylä: Docendo.

Sayar, R. 2013. Top JavaScript frameworks, libraries and tools and when to use them. Sitepoint. Viitattu 20.9.2016. https://www.sitepoint.com/top-javascript-frameworks-libraries-tools-use/.

Seibel, P. 2009. Coders at work. New York: Apress.

System R. 2011. Webopedia. Viitattu 19.10.2016. http://www.webopedia.com/TERM/S/System_R.html.

The virtual DOM. 2016. Fullstackreact.com. Viitattu 17.11.2016. The Virtual DOM. 2016) https://www.fullstackreact.com/p/jsx-and-the-virtual-dom/.

Usage of JavaScript libraries for websites. 2016. W3Techs. Viitattu 5.10.2016. https://w3techs.com/.

What is jQuery?. 2013. jQuery. Viitattu 1.10.2016. https://jquery.com/.

What is the Document Object Model?. 2000. W3Schools. Viitattu 13.11.2016. https://www.w3.org/TR/DOM-Level-2-Core/introduction.html.