LAHDEN AMMATTIKORKEAKOULU
Lahti University of Applied Sciences

MODERNIT WEB-SOVELLUKSET

LAHDEN
AMMATTIKORKEAKOULU
Tekniikan ala
Tietotekniikka
Ohjelmistotekniikka
Opinnäytetyö
Syksy 2013
Joonas Teurokoski

Lahden ammattikorkeakoulu
Tietotekniikan koulutusohjelma

TEUROKOSKI, JOONAS: Modernit web-sovellukset

Ohjelmistotekniikka opinnäytetyö, 45 sivua

Syksy 2013

TIIVISTELMÄ

Tässä työssä käsitellään modernien ja skaalautuvien web-sovellusten kehittämistä erilaisten sovelluskehysten avulla. Työssä tutkittiin mahdollisuutta siirtää perinteisten web-sovellusten esityslogiikka ja liiketoimintalogiikka palvelimelta selaimelle. Logiikan siirtäminen selaimelle vähentäisi palvelimelle kohdistuvaa kuormaa ja parantaisi sovelluksen suorituskykyä. Työssä tutustuttiin asiakaspuolen Single-Page Application sovelluskehitysmalliin, sekä siihen liittyviin tekniikoihin ja toteutusmalleihin.

Single-Page Applicationit (SPA) ovat HTML5 web sovelluksia tai sivuja, jotka suoritetaan asiakaspään selaimessa. SPA-sovelluksissa kaikki ohjelmakoodi suoritetaan selaimessa tai haetaan palvelimelta erikseen tarvittaessa. Perusajatuksena selainpuolen sovelluksissa on se, että mitä tahansa käyttäjä tekeekin sivuilla, se ei vaadi sivuston uudelleenlatausta. SPA-sovellukset poikkeavat merkittävästi perinteisestä monisivuisesta toteutustavasta, jossa selain hakee uutta sisältöä palvelimelta jokaisesta käyttäjän interaktiosta.

Ongelma perinteisissä palvelinpuolen web-toteutuksissa on se, että jokaisen painalluksen aiheuttama sivulataus voi merkittävästi häiritä käyttökokemusta. Perinteisissä web-sovelluksissa käyttäjä joutuu odottamaan joka painalluksella, että koko sivusto on latautunut. Tämä tarkoittaa myös sitä, että mahdollisesti suuri määrä samaa dataa ladataan useita kertoja. Single-Page Applikaatioissa data haetaan käyttämällä XHR Ajax kutsuja ja päivittämällä ainoastaan tiettyjä osioita sivusta, täten tehden käyttökokemuksesta sulavamman.

Tutkittujen tekniikoiden demonstroimiseksi toteutettiin yksinkertainen asiakashallintajärjestelmä. Sovellus toteutettiin JavaScript-ohjelmointikielellä ja käyttäen asiakaspuolen Backbone.js- ja palvelinpuolen Node.js-sovelluskehyksiä.

Asiasanat: Single-Page Application, JavaScript, Node.js, Ajax, web-sovellus, selainpuolen sovellus, HTML5

Lahti University of Applied Sciences
Degree Programme in Information Technology

TEUROKOSKI, JOONAS: Modern Web-Applications

Bachelor's Thesis in software engineering 45 pages

Autumn 2013

ABSTRACT

This Bachelor's Thesis deals with building modern and scalable web applications by using certain application frameworks. The possibility to move most of the presentation logic and business logic of the application's to the clients browser was studied thus reducing the server load and improving performance of the application. This work concentrates on the Single-Page Application programming model and on the different technologies related to it.

Single-Page Applications (SPA) are web applications or websites which run in client's browser. In SPA applications all the program code exists in clients browser's or are fetched separetly from the server as required. The basic idea behind an SPA is that regardless of what interactions users make in the application, the page does not get reloaded. SPAs differ radically from the classic multipage applications where the browser requests data from the server with every single user interaction.

The problem with the classical server side web-applications is that it tends to disrupt the user experience. In classical web applications the user will have to wait for entire page to finish loading. This often means requesting the same data content over and over again. In SPA applications, on the other hand, requests are made using XHR Ajax calls and updating only necessary partions of the page, thus making the user experience more fluid.

To demonstrate the researched methods, a simple CRM (customer relationship manager) application was made. The application was created using the JavaScript programming language and Backbone.js and Node.js application frameworks for both server and client side implementations.

Keywords: Single-Page Application, JavaScript, Node.js, Ajax, web-application, client-side application, HTML5

SISÄLLYS

1 JOHDANTO

11,8 sekuntia on arvioitu web-sivun keskimääräinen latausaika mobiililaitteilla. Vaikkakin selaimet, tietokoneet ja tietoverkot ovatkin kehittyneet yhä nopeammiksi, niin sivujen datamäärät ovat kasvaneet samassa tahdissa. Näin ollen käyttökokemukset eivät ole juurikaan parantuneet, ja käyttäjät joutuvat yhä tänäkinpäivänä odottamaan kohtuuttomia aikoja päästäkseen haluamalleen sivulle. (Strangeloops 2012.)

Sivustojen latausajat ovat merkittävä osa sivuston käyttäjäkokemusta. Mikäli käyttäjien interaktioihin kyetään reagoimaan 0,1 sekunnissa, eli välittömästi, luo se kuvan sivuston reaaliaikaisuudesta ja mahdollistaa käyttäjälle vastaavanlaisen käytettävyyden kokemuksen, kuin perinteisissä työpöytäsovelluksissa on totuttu. Mikäli interaktio kestää sekunnin, käyttäjä alkaa huomata sivuston toiminnassa viivettä ja alkaa erkaannuttaa itseään käyttökokemuksesta. Mikäli sivulatauksessa kestää yli kymmenen sekuntia, alkaa käyttäjän keskittymiskyky herpaantua. Käyttäjä saattaa mahdollisesti unohtaa, mitä oli tullut sivulle tekemään. Latausajan kasvaessa yli kymmeneen sekuntiin käyttäjä todennäköisesti poistuu sivulta. (Strangeloops 2012.)

Sivustojen latausaikojen pidentymisellä on myös taloudellisia vaikutuksia. Amazon arvioi, että yhden sekunnin viive sivulatauksessa voisi maksaa sille 1,6 miljardia dollaria vuodessa. Samoin Google on arvioinut, että mikäli Google-hakukoneen hakutulokset hidastuisivat vain 400 millisekunnilla he voisivat menettää 8 miljoonaa hakua päivässä. (Fastcompany 2012.)

Web on kehittynyt pitkälle sen varsin nöyristä lähtökohdista, jossa sen oli tarkoitus tarjota käyttäjilleen staattisia tekstipohjaisia HTML-dokumentteja. Nykypäivänä web-sivustojen tulee kyetä tarjoamaan yhä monipuolisempaa ja interaktiivisempaa sisältöä kasvavalle määrälle kävijöitä sekä suunnattomalle määrälle erilaisia laitteita. Vaatimusten kasvaessa perinteiset web-tekniikat ja ratkaisut eivät välttämättä ole tehokkain ratkaisu sisällön esittämiseksi.

Perinteisesti web-sovellukset jättivät kaiken raskaan työn palvelinten huoleksi. Palvelinten tehtävänä oli vastata sovelluksen esitys- ja liiketoimintalogiikasta sekä tuottaa staattista sisältöä käyttäjien selaimille. Asiakaspuolen ohjelmointiratkaisut

olivat usein rajoitettuja yksittäisten komponenttien toteuttamiseen ja ainoastaan niiden käyttökokemuksen parantamiseen. (Osmani 2013, 1.)

Modernien ja interaktiivisten web-sovellusten toteuttamiseksi tämä suhde on muutettava päinvastaiseksi. Asiakasohjelmien tehtävänä olisi nyt vastata sovelluksen liiketoiminta- ja esityslogiikasta. Asiakasohjelmat voisivat hakea raakaa sisältöä palvelimilta ja esittää sitä, missä ja milloin se on tarpeellista. Palvelinten tehtäväksi jäisi täten muodostaa rajapinta persistenttien resurssien tarjoamiseksi asiakasohjelmille. (Osmani 2013, 1.)

Tässä työssä käsitellään modernien ja skaalautuvien web-sovellusten kehittämistä erilaisten sovelluskehysten avulla. Työssä tutustutaan asiakaspuolen Single-Page Application sovelluskehitysmalliin sekä siihen liittyviin tekniikoihin ja toteutusmalleihin.

Työ toteutettiin Restbyte Oy:lle eräänlaisena proof-of-concept-ratkaisuna, jonka pohjalta voidaan lähteä toteuttamaan tulevia projekteja. Tarkoituksena oli toteuttaa työ pitkälti tutkimustyönä sekä kyetä demonstroimaan tutkittuja tekniikoita yksinkertaisen web-sovelluksen avulla.

Restbyte Oy on vuonna 2013 perustettu suomalainen IT-alan startup-yritys. Restbyte tarjoaa asiakkailleen rikkaita web- ja mobiilisovellusratkaisuja.

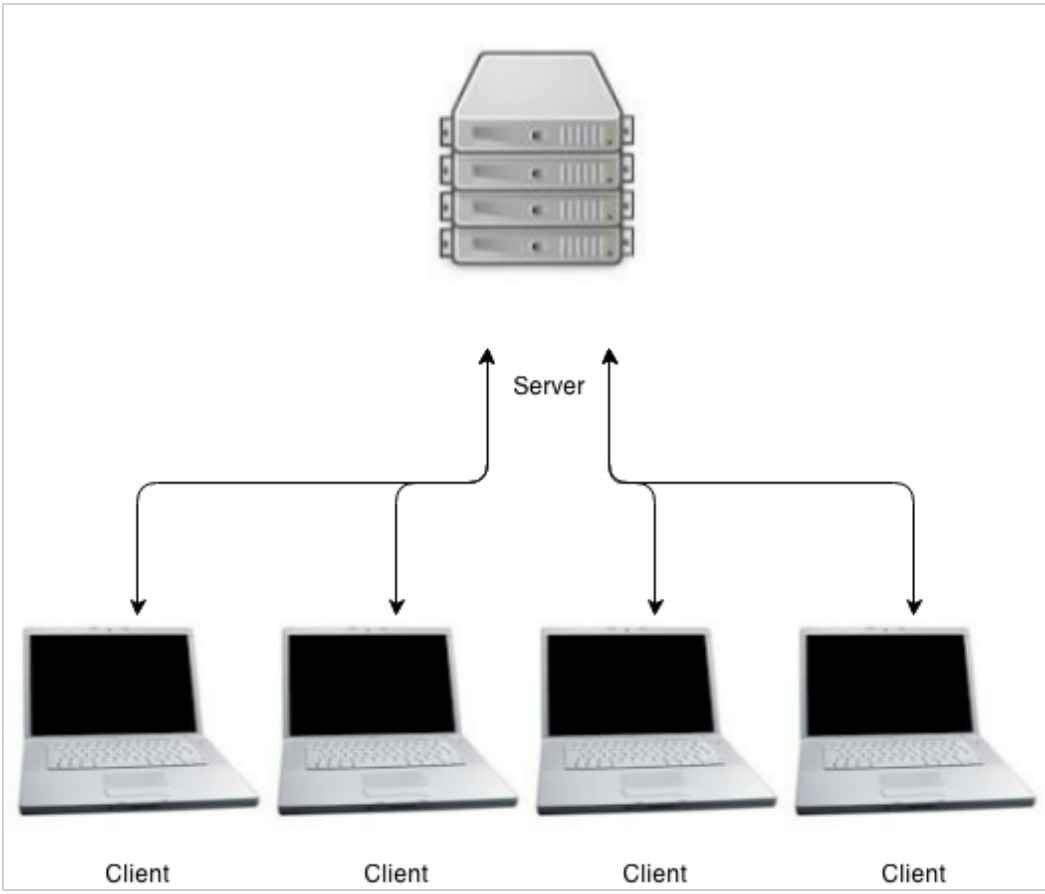
Tutkimustyö alkoi keväällä 2013. Tukimustyön tarkoituksena oli selvittää tapoja web-sovellusten käyttökokemuksen, suorituskyvyn sekä kustannustehokkuuden parantamiseksi. varsinkin laajemmissa sovelluskokonaisuuksissa. Tämä opinnäytetyö käsittelee edellämainitussa tutkimustyössä tutkittuja menetelmiä sekä niiden pohjalta luotua asiakasrekisteri-sovellusta. Proof-of-concept työ toteutettiin kesällä 2013, ja siinä hyödynnettyjä menetelmiä käytetään yrityksen web-sovelluksissa.

Toisessa luvussa käsitellään web-sovellusten historiaa, sekä perinteisten web-toteutusten ja toimintatapojen luomia rajoitteita nykypäiväisissä interaktiivissa web-sovelluksissa. Kolmannessa luvussa käsitellään menetelmiä sovellusten suorituskykyongelmien ratkaisemiseksi ohjelmistokehyksien, kehitysmallien sekä tietokantojen avulla. Neljännessä luvussa käsitellään tutkittujen menetelmien

hyödyntämistä asiakasrekisteri-sovelluksessa. Viides luku sisältää yhteenvedon, pohdintaa tutkittujen menetelmien ongelmista ja niiden korjaamisesta sekä johtopäätöksen.

2 WEB OHJELMISTOKEHITYS YMPÄRISTÖ

Webin toiminta perustuu asiakas-palvelin-malliin. Asiakas-palvelin-malli koostuu kolmesta osasta: asiakasohjelmasta, palvelinohjelmasta ja yhteyskäytännöstä. Asiakasohjelman tehtävänä on lähettää palvelimelle pyyntöjä käyttäjän toimesta. Webissä asiakasohjelmana toimii tyypillisesti selain. Palvelinohjelma käsittelee asiakasohjelmien pyynnöt ja vastaa niihin palauttaen HTML-merkkikielisiä staattisia dokumentteja. Yhteyskäytäntö, eli protokolla määrittelee, miten asiakasohjelma ja palvelinohjelma viestivät keskenään. Asiakas-palvelin-arkkitehtuurimallia on havainnollistettu kuviossa 1. (Osmani 2013, 13.)



KUVIO 1. Asiakas-palvelin arkkitehtuuri

Web-sovellusten asiakaspään toteutus koostuu perinteisesti kolmesta eri teknologiasta; HTML:stä sivujen sisältöjen määrittelemiseksi, CSS:stä sivujen ulkoasujen määrittelemiseksi ja JavaScriptistä sivujen toiminnallisuuksien määrittelemiseksi. Näiden lisäksi web-sivustoilla voidaan mahdollisesti käyttää erilaisia serveri-puolen ohjelmistoratkaisuja, joiden avulla voidaan generoida staattista HTML-sisältöä käyttäjän tarpeiden mukaisesti. Palvelinpuolen sovelluksien tarkoitus on suorittaa skriptejä, joiden pohjalta luodaan HTML-merkkikielistä staattista sisältöä. Skriptit ovat perinteisesti olleet ohjelmointikieliltään ja toteutustavoiltaan agnostisia, mikä tarkoittaa sitä, että mitä tahansa ohjelmointikieltä, jossa on mahdollisuudet HTTP-yhteyksien hallitsemiseen sekä HTML-merkkikielen luontiin, voidaan käyttää palvelinpuolen toteutuksessa. (Osmani 2013, 13.)

2.1 HTML-merkkikieli

HTML (HyperText Markup Language) on hyperlinkkejä sisältävän tekstin eli hypertekstin merkitsemiseen kehitetty kuvauskieli, joka mahdollistaa monimuotoisen sisällön esittämisen web-sivuilla. Internet-sovelluksissa HTML-merkkikieli muodostaa sisällön perusrungon. (W3 2013.)

HTML-merkkikieli suunniteltiin ensisijaisesti hyperlinkkejä sisältävien tieteellisten dokumenttien kuvauskieleksi. Ensimmäinen versio HTML:sta julkaistiin 1990-luvun alkupuolella CERN:ssa. Myöhemmin HTML:n kehittämiseksi ja standroimiseksi perustettiin W3C eli World Wide Web Consortium. HTML-merkkikieli tuli monelle tutuksi 1990-luvulla suositun Mosaic-selaimen myötä, johon se oli sisäänrakennettuna. (W3 2013.)

HTML5 on yleisnimitys työlle, jonka tavoitteena on HTML-kielen laajentaminen ja sen toimintojen määrittely. HTML5:n kehitystyön periaatteena on, että selaimien ja HTML:n kehitys tapahtuu rinnakkain yhteistyössä selainvalmistajien kanssa. HTML5:n määrittelyä tekevät yhteistyössä W3C ja WHATWG. (W3 2013.)

HTML5-standardi mahdollistaa sisältörikkaiden web-sovellusten toteutuksen. HTML5:ssä on ominaisuuksia verkkosivujen grafiikan, liikkuvan kuvan ja vuorovaikutuksen toteuttamiseen ilman riippuvuutta ulkoisista liitännäisistä. HTML5 helpottaa myös sivustoilla tapahtuvaa tiedon tarkastamista ja muokkaamista sekä selkeyttää sivustojen rakennetta. Lisäksi HTML5:n avulla voidaan toteuttaa JavaScript-pohjaisia web-sovelluksia joko paikallisesti tai palvelimella. HTML5 vähentää myös toimintojen selainkohtaista koodaamista, sillä useat selainvalmistajat ovat mukana kehitystyössä. Selainvalmistajat ottavat ominaisuuksia käyttöön kuitenkin eri tahtia. (W3 2013.)

2.2 CSS-tyylikieli

Cascading Style Sheets (CSS) on yksinkertainen tyylikieli, jonka tarkoituksena on mahdollistaa HTML-dokumenttien ulkoasun ja esitystapojen määrittäminen. Sen tehtävänä on muotoilla HTML-dokumenteissä esiityviä elementtejä ominaisuuksiensa mukaan. CSS3 on CSS-merkkikielen uusin versio, johon on lisätty paljon uusia ominaisuuksia. CSS3 kehitettiin osittain korvaamaan vanhoja Flash-, ja Java-applet toteutuksia. CSS3-kieltä kehitetään samaan aikaan HTML5:n rinnalla. Kyseistä tekniikkaa kehittää W3C (World Wide Web Consortium). (W3 2013.)

2.3 JavaScript-ohjelmointikieli

JavaScript on web-sovelluskehitykseen suunniteltu funktionaalinen ohjelmointikieli. Ylivoimaisesti suurin osa moderneista web-sivustoista käyttää Javascriptiä, sekä kaikki modernit web-selaimet tietokoneilla, pelikonsoleilla, tableteilla, sekä älypuhelimilla tukevat sitä tehden siitä yhden monikäyttöisimmistä ohjelmointikielistä. (Haverbecke 2007.)

JavaScipt syntyi tarpeesta luoda webistä dynaaminen. Kielen kehitti Brendan Eich, ja se julkaistiin ensimmäisen kerran Netscape 2-selaimen mukana vuonna 1995. (Haverbecke 2007.)

JavaScript-nimi on varsin harhaanjohtava. Ohjelmointikielen syntaksin yhteneväisyyksistä Java-ohjelmointikieleen lukuunottamatta JavaScript on täysin

eri kieli kuin Java. Java-etuliite kielelle otettiin puhtaasti markkinointitarkoituksiin. Vuonna 1995, kun JavaScript julkaistiin, Java-ohjelmointikieli oli saavuttamassa suurta suosiota. Täten ilman parempaa avostelukykyä Netscapen markkinointivastaavat päättivät yrittää hyötyä Javan saavuttamasta suosiosta. Nimestään huolimatta JavaScriptistä on kehittynyt täysiveroinen yleiskäytännöllinen itsenäinen ohjelmointikieli. (Haverbecke 2007.)

ECMAScript on skriptikielen standardi, jota myös JavaScriptin ydinkomponentit noudattavat. ECMAScript-standardi syntyi tarpeesta standardisoida JavaScript-ohjelmointikieli. ECMAScriptin standardisoinnista vastaa European Computer Manufacturer's Association (ECMA), jonka mukaan myös standardi on nimetty. Opinnäytetyön kirjoitushetkellä ECMAScript 5 on standardin tuorein iteraatio. (Haverbecke 2007.)

Suurimmat erot Javascriptin ja esimerkiksi Javan ja C-kielen kanssa on se, että JavaScript on löyhästi tyypitetty. Löyhällä tyypityksellä tarkoittetaaan sitä, ettei muuttujia tarvitse määritellä erikseen kokonaisluvuksi, liukuluvuksi tai merkkijonoksi. JavaScriptissä selaimet pitävät huolen muistin käytöstä, eli huolehtivat niin sanotusta roskien keruusta, jossa tarpeettomat objektit pyritään tuhoamaan automaattisesti muistista. Tämä helpottaa ja nopeuttaa Javascriptillä ohjelmointia. (Haverbecke 2007.)

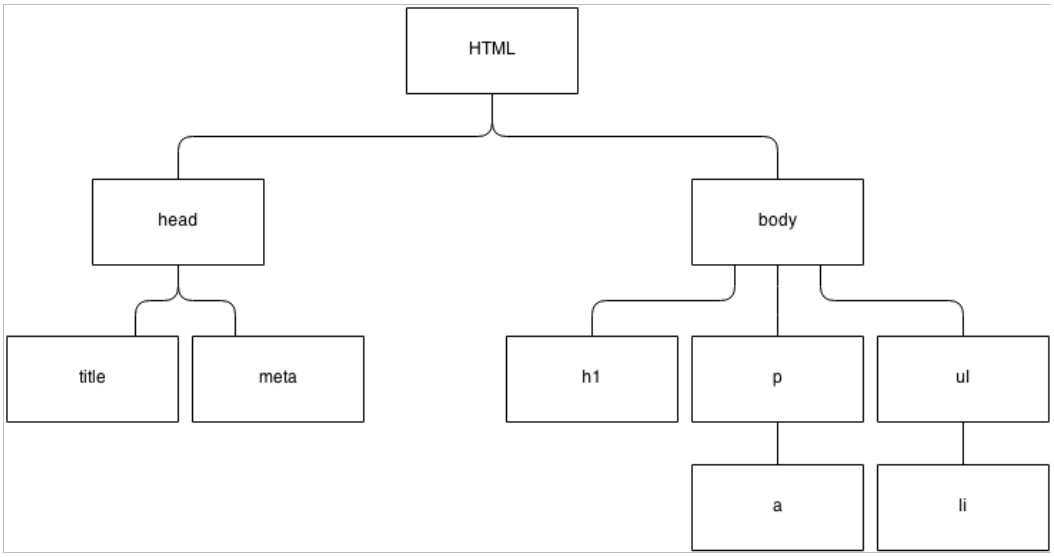
Vaikka JavaScript onkin funktionaalinen ohjelmointikieli, tukee se myös olio-ohjelmointiin perustuvaa sovelluskehitystä. Toisin kuin perinteiset olio-ohjelmoitivat ohjelmointikielet, JavaScriptissä ei ole luokkia. Olioiden sijaan JavaScriptissä käytetään olioprototyyppejä ja funktioita, joita voidaan käyttää olioina. (Mozilla 2013.)

JavaScriptin tulevaisuus näyttää kirjoitushetkellä poikkeuksellisen valoisalta. JavaScriptistä on tullut varteenotettava ohjelmointikieli niin selain-, työpöytä- kuin palvelinohjelmistoissa. Yritykset ja organisaatiot, kuten Mozilla, Google ja Microsoft kehittävät aktiivisesti omia JavaScript-implementaatioitaan. Selainten välinen kilpailutilanne vauhdittaa merkittävästi JavaScriptin kehitystä parantaen sen ominaisuuksia ja suorituskykyä julkaisusta toiseen.

2.4 Document-Object Model (DOM)

Jokainen elementti HTML-dokumentissä kuuluu osaksi rakennetta, jota kutsutaan Document-Object Model (DOM) -rakenteeksi. DOM on tapa, jolla JavaScript ymmärtää HTML-dokumentin ja jolla se voi tarkastella ja muokata dynaamisesti dokumenttia. Jokainen elementti voidaan kuvata tällä mallilla, ja niiden kanssa voi tapahtua interaktioita. Asiakaspuolen JavaScript-sovelluksissa voidaan helposti käsitellä, muokata, tai luoda uusia DOM-elementtejä, mikä puolestaan heijastuu perinteisesti staattiseen HTML-näkymään, mahdollistaen dynaamisen sisällön luomisen. (Haverbecke 2007.)

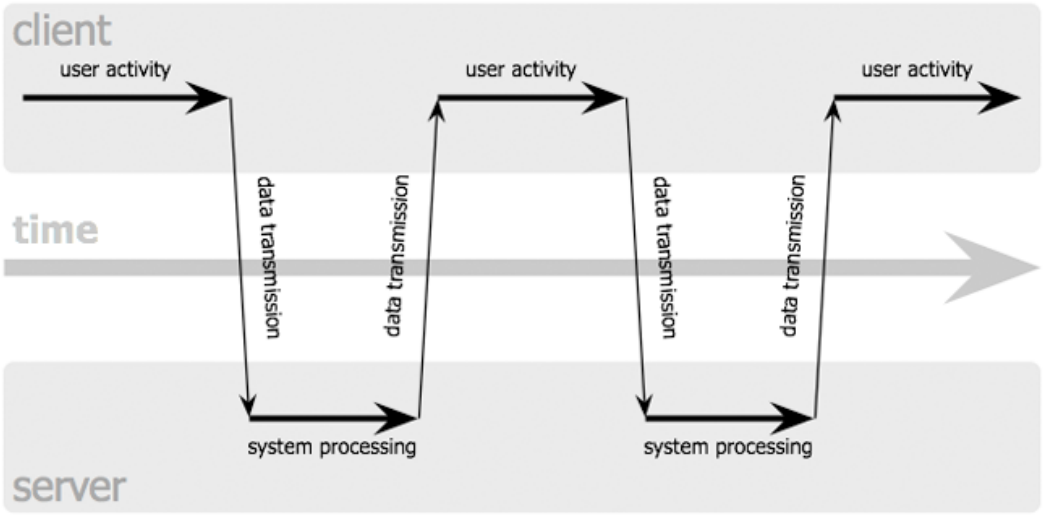
HTML-dokumenteilla on siis niin kutsuttu hierarkinen rakennne; jokainen elementti muodostaa solmun, joka kuuluu jonkin toisen elementin sisään, joka on sen isäntä-elementti. Jokainen elementti voi sisältää lapsi-elementtejä, täten muodostaen eräänlaisen puurakenteen. Kuviossa 2 esitetään HTML-dokumentti DOM-puurakenteena. (Haverbecke 2007.)



KUVIO 2. DOM-elementtien puurakenne

3 UUDET WEB-SOVELLUS TEKNIIKAT

Perinteisessä web-sovellusten toimintamallissa selain tekee palvelupyynnön palvelimelle käyttäen HTTP-yhteyskäytäntöä, jolloin palvelin palauttaa käyttäjälle HTML-sisältöä. HTML sisältää määrittelyt CSS-tyylitiedostojen, kuvien, videoiden ja muun datan lataamiseksi, jonka jälkeen selain lähettää uuden palvelupyynnön palvelimelle näiden lataamiseksi. Tätä on havainnollistettu kuviossa 3. Yksittäinen sivulataus voi pahimmillaan aiheuttaa jopa satoja palvelupyyntöjä palvelimelle. Käyttäjän interaktioidessa sivuston kanssa esimerkiksi linkkiä painamalla tämä tapahtuu jälleen uudelleen, aiheuttaen valtavan määrän turhaa datan prosessointia. Tämä on pitkälti se toimintamalli, miten web on toiminut lähes koko historiansa ajan. (Garrett 2005).



KUVIO 3. Perinteisen web-sovelluksen toimintamalli (Garrett 2005).

Ensimmäinen pullonkaula sivuston suorituskyvylle varsinkin mobiiliverkoissa on viive. Kuten edellä mainittiin reaaliaikaisen käyttäjäkokemuksen saavuttamiseksi sivun tulisi kyetä käsittelemään sisältö noin sadassa millisekunnissa. Palvelimella on siis vain sata millisekunttia aikaa prosessoida data ja palauttaa se takaisin käyttäjälle, tietoverkkojen viiveet voivat kuitenkin helposti viedä koko tuon käytettävissä olevan ajan.

Palvelimen prosessoidessa käyttäjän palvelupyyntöjä joutuu se usein tekemään kyselyitä tietokantaan. Data-intensiivisissä sovelluksissa tietokantakyselyt voivat

helposti viedä loputkin käytettävissä olevasta ajasta. Tämän lisäksi palvelin joutuu prosessoimaan tietokannasta saadun datan useimmiten HTML-, XML- tai JSON-muotoon ja palauttamaan sen takaisin käyttäjälle selaimen tulkittavaksi, tällöin törmätään jälleen viiveeseen.

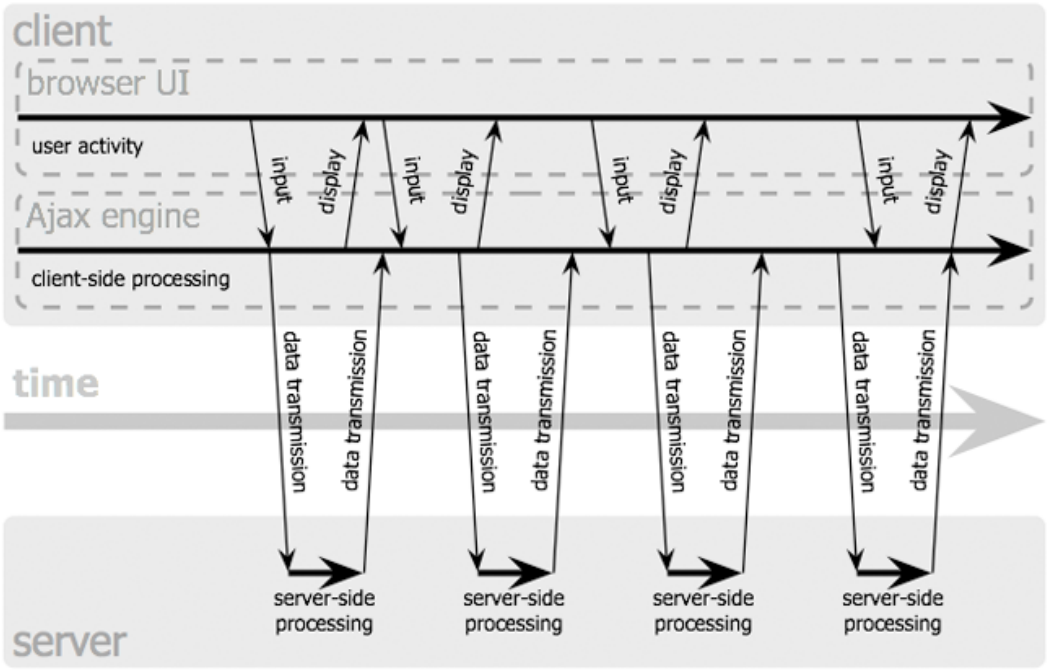
Ratkaisuna perinteisten web-sovellusten käyttäjäkokemusten ja suorituskyvyn parantamiseksi tutkittiin mahdollisuutta muuntaa perinteinen asiakas-palvelin-toimintamalli päinvastaiseksi. Asiakasohjelman tehtäväksi tulisi reagoida käyttäjän interaktoihin ja esittää käyttäjän vaatimaa sisältöä, sen sijaan, että palvelin tekisi kaiken raskaan prosessoinnin ja palauttaisi asiakasohjelmalle jo käsitellyn datan valmiiksi esitetyssä muodossa.

Ongelmaan ratkaisuna haluttiin tutkia natiivien JavaScript web -sovellusten näkökulmasta, sen mahdollistamien suorituskykyhyötyjen vuoksi. Sen lisäksi haluttiin tutkia mahdollisuutta käyttää samaa kieltä, niin asiakas- kuin palvelinohjelmistojen sekä tietokantojen kanssa. Täten toteuttaen koko web-ohjelmiston pino, joka on perinteisesti ollut sekoitus eri ohjelmointikieliä, käyttämällä pelkästään JavaScript-ohjelmointikieltä. Kuviossa 4 on havainnollistettu eri sovelluskerroksissa käytettäviä sovelluskehyksiä.

KUVIO 4. Sovelluksen eri kerroksissa käytettävät teknologiat

3.1 Asynchronous Javascript and XML (AJAX)

Ajax on lyhenne termistä Asynchronous Javascript and XML, joka tarkoittaa JSON-, XML-, tai HTML-tyyppisen sisällön lataamista verkkolähteestä. Ajax mahdollistaa reaaliaikaisen interaktiivisuuden web-sivuilla. Ajaxin avulla asiakaspään ohjelma voi syöttää tai hakea tietoja palvelimelta. Ajax-kutsujen asynkroninen luonne mahdollistaa web-sivun eri osioiden päivityksen ilman, että koko sivu haettaisiin uudelleen. (Garrett 2005.)



KUVIO 5. Ajax-toimintamalli (Garrett 2005)

Perinteisissä web-sovelluksissa jokainen käyttäjän interaktio aiheuttaisi normaalisti HTTP–pyynnön palvelimelle. Jokainen tapahtuma, joka ei välttämättä vaadi kommunikointia palvelimen kanssa, kuten yksinkertainen datan validointi voidaan suorittaa puhtaasti JavaScript-koodissa. Mikäli ohjelma tarvitsee dataa palvelimelta vastatakseen, tällöin ohjelma voi suorittaa kyseisen datan haun palvelimelta asynkronisesti keskeyttämättä käyttäjän interaktioita sivulla. Kuviossa 5 kuvataan Ajax-ohjelmointimallin asynkronista toimintaa.

Ajax ei ole yksittäinen teknologia, vaan se koostuu useammasta teknologiasta. Ajax rakentuu olemassa olevien tekniikoiden varaan. Sen takana ovat standardeihin perustuvat verkkosivun esitysteknologiat HTML, CSS, DOM, datan vaihdon hoitava XML sekä asynkronisen tiedon haun hoitava XMLHttpRequest. Ajaxin keskeisin komponentti on JavaScript, joka sitoo kaikki nämä teknologiat yhteen. Ajaxin etuna on myös se, että se ei vaadi selaimeen asennettavia erillisiä lisäosia. (Garrett 2005.)

XMLHttpRequest (XHR) on tärkeä osa Ajax-tekniikkaa, ja kaikki yleisimmät selaimet tukevat sitä. XHR on DOM-ohjelmointirajapinta, jota voidaan käyttää selainohjelmissa. XHR mahdollistaa asynkronisen kommunikoinnin selaimen ja palvelimen välillä. Sen avulla voidaan lähettää HTTP-pyyntö suoraan palvelimelle ja saada vastaus teksti- tai XML-muodossa ilman koko verkkosivun uudelleen lataamista. (Garrett 2005.)

Asynkronisissa selainohjelmissa ohjausrakenteena käytetään perinteisesti callback-metodeja. Callback-metodilla tarkoitetaan asynkronisesti suoritettavalle funktiolle välitettävää metodia, joka ajetaan asynkronisen toiminnon valmistuttua. Callback-menetelmät monimutkaistavat sovelluksen ohjausrakennetta, sen poikkeusehtojen hallintaa sekä funktioiden semantiikka verrattuna siihen, mihin on totuttu synkronisissa sovelluksissa. Asynkronisten sovellusten ohjausrakenteen helpottamiseksi on kuitenkin olemassa menetelmä nimeltä Promiset. Promise on abstraktio, joka mahdollistaa synkronisesta ohjelmointimallista tutun ohjausrakenteen käytön asynkronisissa sovelluksissa. Promise-objektin tehtävänä on toimia välittäjänä asynkronisten metodien paluuarvoille tai poikkeuksille. (Strongloop. 2012.)

3.2 Selainpohjaiset sovellukset

Selainpohjaisilla sovelluksilla perinteisesti tarkoitetaan web-sovellusta, joka suoritetaan käyttäjän selaimessa. Selainpohjaisissa sovelluksissa ohjelmalogiikka on siirretty palvelimelta suoraan käyttäjän selaimeen prosessoitavaksi. Tämän ansiosta sovellus kykenee työpöytäsovellusten kaltaisesti reagoimaan välittömästi käyttäjän syötteisiin. Selainpohjaisissa sovelluksissa palvelinohjeman rooliksi jää

toimiminen ainoastaan rajapintana asiakasohjelman palvelupyynnöille. (Osmani, 2013.)

Idea puhtaasti selaimessa toimivista ohjelmista ei sinäänsä ole uusi ja vallankumouksellinen. Vastaavanlaisia toteutuksia on ollut olemassa jo aikaisemminkin. Suosittuina toteutusmalleina ovat aiemmin olleet mm Flash, Silverlight ja Java-appletit. Aiemmat toteutusmallit ovat kuitenkin idealtaan vastanneet ainoastaan yksittäisten komponenttien toiminnasta eivätkä niinkään monimutkaisten sivukokonaisuuksien hallinnasta. (Osmani, 2013.)

Ratkaisuksi laajojen sivukokonaisuuksien hallintaan selainpuolella syntyi niin kutsutut yhden sivun sovellukset, eli Single Page Applicationit (”SPA”). SPA-toteutusmalli mahdollistaa sen, että kun sivu on kertaalleen ladattu, kaikki myöhemmät tapahtumat sivuilla sekä navigaatio voidaan suorittaa lataamatta koko sivua uudelleen. Palvelimelta saatava sisältö ladattaisiin tarvittaessa käyttäen ennaltamäärättyjä kommunikointimenetelmiä ja rajapintoja. Sovelluskehitysmallin keskeisiin toimintamalleihin kuuluu asynkroninen kommunikointi selain- ja palvelinpuolen ohjelmistojen välillä käyttäen AJAX-menetelmiä. Sivuston asynkroninen toiminta mahdollistaa käyttäjälle työpöytäsovellusmaisen reaaliaikaisen käyttökokemuksen. (Osmani, 2013.)

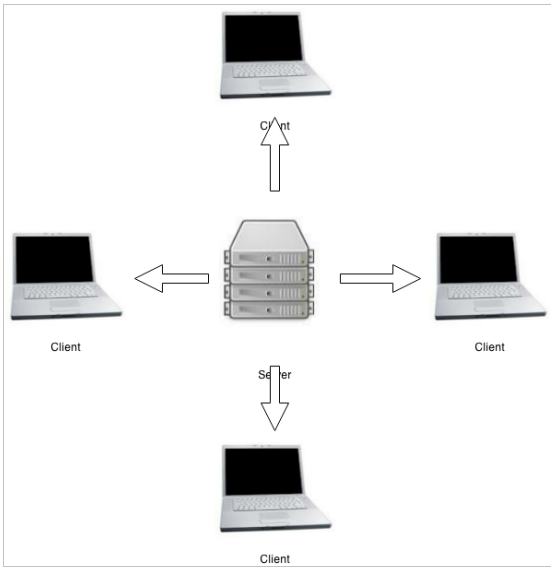
SPA-sovellukset saivat alkunsa 2000-kuvun puolessavälissä sovellusten, kuten Gmail ja Google Mapsin yhteydessä. Tänäpäivänä menetelmää käytetään useissa korkean profiilin web-palveluissa, kuten Twitterissä, Facebookissa, GitHubissa, sekä Flickr:ssä. (Osmani, 2013.)

Yhden sivun sovelluksissa asiakas- ja palvelinohjelmistojen välinen interaktio tapahtuu vain sovitun rajapinnan kautta. Tällöin ohjelmalla ei tarvitse olla tarkempia tietoja toistensa sisäisistä toteutuksista, tietokannoista tai ohjelmointikielistä. Toteutusmallissa sovelluksen eri kerrokset olisivat näin ollen löyhästi sidottuja toisiinsa. Palvelinohjelma voi siis kätkeä sisäänsä toiminnallisuuden tietokannan kanssa interaktoimiseen sekä niin kutsutun liiketoimintalogiikan. Näin ollen palvelimen tehtäväksi jää toimia lähinnä persistenttinä tietovarastona. Ohjelmien ollen löyhästi sidottuja selainpuolen varsinaisen toteutuksen vaihtuessa yhteensopivuus palvelinpuolen ohjelmaan

säilyy. Sama pätee myös palvelinohjelmiston muuttuessa, mikäli vain sovitut yhteyskäytännöt ovat yhä samat. (Osmani 2013.)

SPA-sovelluksissa palvelimilta haettava data noudetaan ja esitetään vasta, kun sille on tarvetta. Palvelinohjelma voi tarvittaessa muokata tai käsitellä tietokantaan serialisoitua dataa. Pääsääntöisesti data palautetaan asiakasohjelmalle raa'assa JSON-formaatissa. Tällä toteutusmallilla verkon yli siirrettävän datan määrä pienenee, minkä ansiosta voidaan helpottaa verkkoviiveistä johtuvia ongelmia. (Osmani, 2013, 14.)

Toteutusmalli keventää merkittävästi palvelimelle kohdistuvaa kuormaa, kun tavanomaiset tehtävät, kuten HTML-merkkijonojen ketjutus siirretään pois palvelimelta viemästä arvokasta prosessointiaikaa. Tämänkaltaisella hajautetulla järjestelmällä mahdollistetaan yhä paremmin skaalautuvien ja reaaliaikaisten verkkosovellusten sekä kustannustehokkaampien web-palveluiden toteuttaminen. Hajautettua arkkitehtuuria on havainnollistettu kuviossa 6.



KUVIO 6. Hajautettu arkkitehtuuri

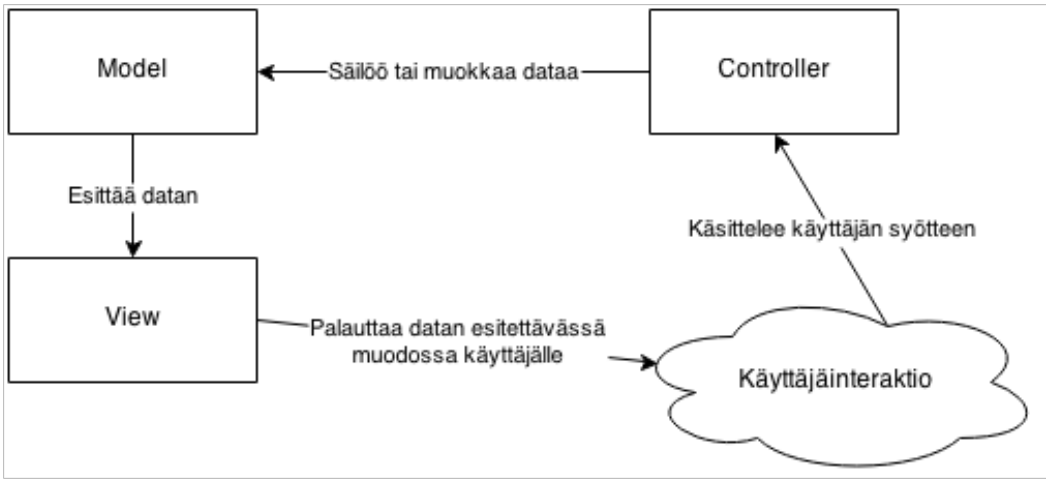
SPA-sovelluksien on myös mahdollista käyttää hyväkseen HTML5- tekniikan tuomia etuja, kuten selainpohjaisia local storage- tietokantoja sekä History API:a, joka mahdollistaa perinteisen sivunavigaation toiminnallisuudet historioineen toimimaan puhtaasti selaimessa. (Osmani 2013.)

3.3 Model-View-Controller (MVC) –arkkitehtuuri

Tarve nopeiden ja interaktiivisten, mutta monimutkaisten Ajax-pohjaisten sovellusten kehittämiselle vaatii usein paljon ohjelmakoodin duplikointia. Koodin duplikointi lisää merkittävästi sovellusten kokoa ja monimutkaisuutta. Tämä on luonut tarpeen tehokkaampien ohjelmistojen suunnittelmullien käytölle selainpuolen ohjelmissa.

Model-View-Controller (malli-näkymä-käsittelijä) on arkkitehtuurillinen suunnittelumalli, joka pyrkii helpottamaan sovellusten organisointia erottamalla sovelluksen eri vastuualueet toisistaan. Se pakottaa liiketoimintalogiikan erottamisen käyttöliittymästä käyttämällä käsittelijää, jonka tehtävänä on logiikan ja käyttäjäsyötteiden hallinta sekä mallien ja näkymien ohjaus. (Osmani, 2013, 13.) Kuviossa 7 kuvataan MVC-suunnittelumallin toimintaa.

MVC-mallin kehitti alunperin Trygbe Reenskaud työskennellessään Smalltalk-80:n parissa. Tänäpäivänä MVC-suunnittelumallia käytetään hyväksi laajassa kirjossa eri ohjelmointikieliä sekä sovelluskehyksiä. (Osmani, 2013, 13.)



KUVIO 7. Model-view-controller arkkitehtuuri

3.4 Representational State Transfer (REST)

Representational State Transfer (REST) on HTTP-protokollaan perustuva arkkitehtuurimalli. REST-arkkitehtuurimallin tyypillisiä ominaisuuksia ovat resurssipohjaisuus sekä yhteneväiset rajapinnat asiakkaan ja palvelimen välillä. REST-arkkitehtuurimallin kehitti alunperin W3C Technical Architecture Group (TAG) rinnakkaisesti HTTP 1.1-standardin kanssa, joka perustuu olemassaolevaan HTTP 1.0-standardiin. REST-arkkitehtuurimallin ydin koostuu kuudesta eri säännöstä, joita se asettaa ohjelmistoarkkitehtuurille: yhteneväiset rajapinnat, asiakas-palvelin-malli, välimuisti, oletus kerroksittaisesta järjestelmästä ja mahdollisuudesta code on demand -toteutuksille. Mikäli sovellus täyttää kaikki REST-arkkitehtuurimallin säännöt, voidaan sen katsoa olevan REST-arkkitehtuurimallin mukainen. (Fielding 2000.)

REST-arkkitehtuurimallia käyttävän sovelluksen tila ja toiminnallisuus tulee olla abstraktoituna resursseiksi. REST:ssa resurssilla tarkoitetaan mitä tahansa informaatiota, jonka voi nimetä. (Fielding 2000.)

Suurin ero muihin rajapinta- arkkitehtuurityyleihin on komponenttien välisten rajapintojen yhteneväisyys. REST:issa resurssien tulee jakaa yhtenäinen rajapinta sovelluksen tilan siirtämiseksi palvelimen ja asiakkaan välillä. (Fielding 2000.)

Rajapinnan tilattomuudella tarkoitetaan sitä, että palvelinohjelma ei sisällä tietoa asiakasohjelmien tiloista. REST:issa jokainen viesti kuvastaa itseään sisälten tarpeeksi tietoa, jotta palvelin kykenee ymmärtämään asiakasohjelman viestit pelkästään sen sisällön perusteella. Näin ollen palvelimen skaalautuvuus paranee, sillä sen ei tarvitse pitää muistissaan tietoa aikaisemmista pyynnöistä käsitelläkseen uusia pyyntöjä. (Fielding 2000.)

REST-rajapintojen tulee noudattaa asiakas-palvelinmallia. Resurssien tulee olla yksilöllisesti nimettyjä, ja niiden tulee olla saatavissa käyttäen URI-osoitteita. Rajapinnan asiakas-palvelin-arkkitehtuuri mahdollistaa eri puolten komponenttien eriyttämisen toisistaan. Tämä mahdollistaa löyhän sidonnan periaatteen asiakkaan ja palvelimen välillä, täten parantaen ohjelmien siirrettävyyttä eri alustoille. (Fielding 2000.)

Välimuistin käytöllä RES-rajapinnoissa tarkoitetaan palvelinohjelman mahdollisuutta asettaa asiakasohjelmalle lähetettäviä vastauksia välimuistiin tallennettavaksi. Tällä tehostetaan verkon ja palvelinresurssien käyttöä, kun staattinen ja usein tarvittu data saadaan haettua välittömästi välimuistista, mahdollisesti jopa välttäen käynnin kokonaan palvelimelle. (Fielding 2000.)

REST-rajapintojen kerroksittaisuudella tarkoitetaan, että palvelinrajapinnoilla voi olla useita eri ohjelmistokerroksia eikä asiakasohjelma aina voi tietää, kenen kanssa se keskustelee. Asiakasohjelma ei siis tiedä, eikä sen tarvitse tietää, mikäli haettava resurssi palautetaan välimuistista, tai mikäli se haetaan suoraan tietokannasta. Kerroksittaisuus edesauttaa ohjelmien eriyttämistä toisistaan ja parantaa niiden skaalautuvuutta. (Fielding 2000.)

Code on demand tarkoittaa toimintatapaa, jossa palvelinohjelma voi väliaikaisesti laajentaa asiakasohjelmaa siirtämällä logiikkaa palvelimelta asiakkaalle. Code on demand-kehitysmallissa asiakasohjelmalla on pääsy tiettyihin resursseihin, mutta sillä ei ole tietoa, kuinka käsitellä niitä. Käsitellääkseen kyseisiä resursseja asiakasohjelma voi hakea ohjeet niiden suorittamiseksi palvelimelta. Asiakasohjelma lähettää pyynnön palvelimelle, joka puolestaan palauttaa takaisin ohjelmakoodin näiden resurssien suorittamiseksi. Tarvittaessa ohjelmointi on REST-arkkitehtuurimallin ainut valinnainen sääntö. (Fielding 2000.)

REST-arkkitehtuurimallin hyödyntämisessä on useita hyötyjä. REST mahdollistaa tehokkaamman verkon suorituskyvyn, skaalautuvuuden, rajapintojen yksinkertaisuuden, muokattavuuden, siirrettävyyden ja luotettavuuden. (Fielding 2000.)

3.5 Backbone.js

Backbone.js on Model-view-presenter-suunnittelumallia käyttävä pieni JavaScript-kirjasto, jonka tehtävänä on luoda selkeämpi rakenne asiakaspuolen ohjelmiin. Sen tarkoituksena on parantaa ohjelmiston ylläpidettävyyttä tekemällä ohjelmistosta helpommin hallittava, hajauttamalla monoliittiset ja triviaalit kokonaisuudet erillisiksi osikseen. (Osmani 2013.)

Backbonea käytetään usein Yhden sivun applikaatioiden (”Single-page application”) luomiseen. Backbone.js on yksi monipuolisimmista ja suosituimmista SPA-sovelluskehyksistä. Useat suuret yritykset, kuten Disqus, Walmart, SoundCloud ja LinkedIn käyttävät ohjelmistoratkaisuissaan Backbonea. (Osmani, 2013, 2.)

Backbone tarjoaa minimalistisen kokoelman työkaluja datan jäsentelyyn, käyttöliittymien luontiin sekä apuvälineitä, jotka ovat hyödyllisiä suurten ja dynaamisten ohjelmistojen luonnissa. Backbone ei ole dogmaattinen toteutustapojen suhteen, vaan tarjoaa kehittäjälle vapauden ja joustavuuden kehittää sovelluksesta juuri sellaisen, kuin kehittäjä itse näkee parhaakseen.

Kehittäjä voi vaihtoehtoisesti käyttää mukana tulevaa ohjelmistoarkkitehtuuria, tai laajentaa sitä sopimaan tarpeisiinsa. Backbone.js pyrkii olemaan modulaarinen, eikä se sido kehittäjää käyttämään mitään tiettyä kirjastoa näkymien DOM:in tai templatejen hallintaan, toisin kuin useimmat MVC-sovelluskehykset. Sen sijaan Backbone tarjoaa vapauden kehittäjälle käyttää mieluisia kirjastoaan. (Osmani, 2013, 4.)

Backbonen ydinkomponentteihin kuuluvat malli (model), näkymä (view), kokoelma (collection) ja reititin (router). Backbone tukee tapahtumapohjaista kommunikaatiota näkymien ja mallien välillä, mahdollistaen löyhän sidonnan eri komponenttien välille. Komponenttien välinen löyhä sidonnaisuus ja Backbonen

tapahtumavetoisuus antaa kehittäjille mahdollisuuden hienosäätää näkymien tilojen muutoksia. (Osmani 2013.)

Backbone tarjoaa myös tuen mallien ja kokoelmien suoraan liittämiseen REST-rajapintoihin. Tämä mahdollistaa asiakaspuolen datan helpon sitomisen palvelinpuolelle. (Osmani, 2013, 22.)

Yhden sivun applikaatioissa sovellus ladataan selaimeen normaalilla HTTP-käytännöllä. Sivu voi olla yksinkertaisesti staattinen HTML-sivu. Kun SPA-sovellus on ladattu, asiakaspuolen reititin kaappaa selaimen URL-muutokset sekä alustaa ohjelmalogiikan, sen sijaan että palvelimelle lähetettäisiin uutta palvelupyyntöä.

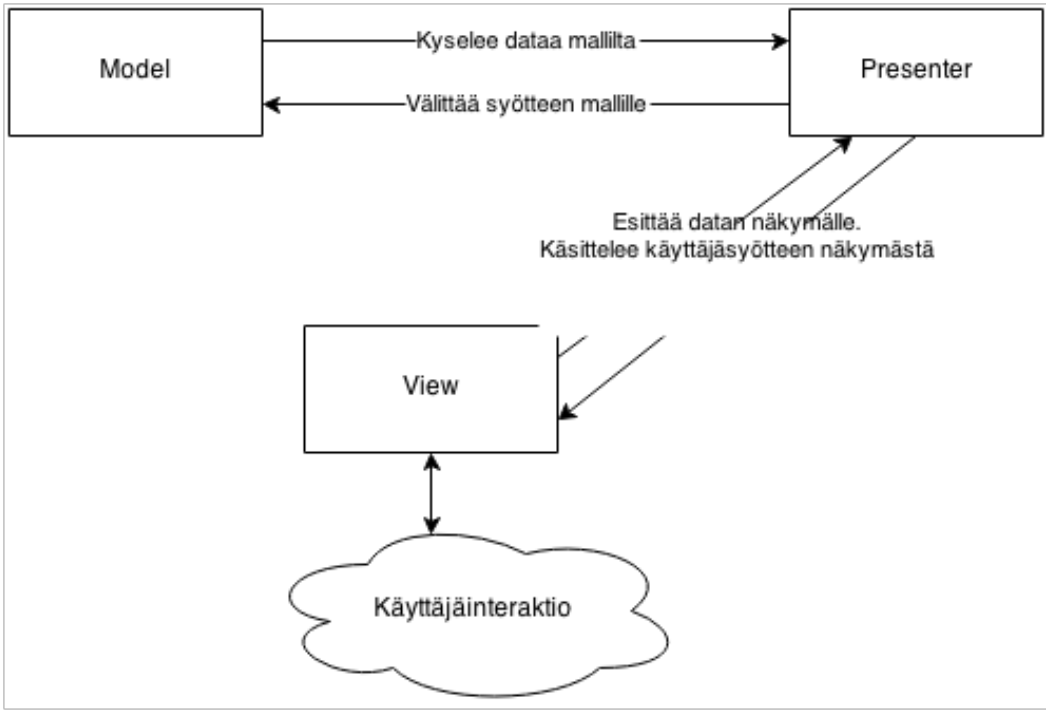
URL-reititys, DOM-tapahtumat sekä mallien muutokset ja tapahtumat laukaisevat tapahtumia suoritettavaksi näkymän hallintalogiikassa. Näkymä tekee tarvittavat muutokset DOM-elementeille sekä malleille, jotka puolestaan voivat laukaista uusia tapahtumia. Mallit synkronisoidaan automaattisesti datalähteeseen, mikä useimmissa tapauksissa tarkoittaa kommunikointia palvelinrajapinnan kanssa. (Osmani, 2013, 18.)

3.5.1 Model-View-Presenter (MVP)

Model-View-Presenter (MVP) on MVC-suunnittelumalliin pohjautuva suunnittelumalli, jonka tehtävänä on parantaa ohjelmistojen esityslogiikkaa. Presenter-komponentin tehtävänä on pitää huolta käyttöliittymän liiketoimintalogiikasta näkymälle. Toisin kuin MVC:ssä, näkymän kutsut välitetään presenter-komponentille, joka on eriytetty erilleen näkymästä. Presenter toimii välittäjänä, jonka tehtävänä on keskustella näkymän ja mallin kanssa eriyttäen näkymän ja mallin toisistaan. Presenterit sitovat tehokkaasti mallit näkymiin, korvaten pitkälti MVC-mallin käsittelijän (controller) roolin. (Osmani, 2013, 324.)

Näkymän pyynnöstä presenterit käsittelevät kaiken käyttäjäinteraktioihin liittyvän työn, muokkaavat sen esitettävään muotoon ja palauttavat käsitellyn datan takaisin näkymälle. Mallit voivat laukaista tapahtumia näkymälle, mutta on presenterin

tehtävä huolehtia niistä ja vastata näkymien päivittämisestä. MVP-suunnittelumallin toimintaa on havainnollistettu kuviossa 8. (Osmani, 324, 2013.)



KUVIO 8. Model-View-Presenter-arkkitehtuurimalli

MVP-suunnittelumallin suurimpina hyötyinä voidaan pitää sitä, että se lisää sovellusten testattavuutta, sekä tarjoaa siistimmän eriyttämisen näkymän ja mallin välille. (Osmani, 324, 2013.)

3.5.2 Malli

Backbonen mallit (model) ovat sovelluksen ydin. Mallit sisältävät sovelluksen dataa sekä siihen liittyvää logiikkaa, kuten muunnokset, validoinnit sekä tapahtumien hallinnan. Kuviossa 9 on luotu esimerkki Backbone mallin toteuttamiseksi. (Osmani, 2013, 27.)

Backbonessa malleja käytetään kuvastamaan palvelinpuolen dataa. Muutokset, joita niihin tehdään, heijastuvat suoraan palvelimelle, käyttäen REST-rajapintoja. (Osmani, 2013, 53.)

Malleille on mahdollista asettaa kuuntelijoita, joiden tehtävänä on reagoida mallejen CRUD (create, read, update, delete)-operaatioihin. Kuuntelijoita voidaan asettaa joko suoraan mallille tai mille tahansa sen attribuuteille. (Osmani, 2013, 55.)

Backbone tukee myös mallien attribuuttien validointia validate-metodinsa avulla. Validointi mahdollistaa mallien attribuuttien oikeellisuuden tarkastamisen ennen niiden tallentamista persistenttiin tietovarastoon. (Osmani, 2013, 28.)

```
// Esimerkki Backbone Malli
var Book = Backbone.Model.extend({
    // Mallin attribuutit, ja niiden vakioarvot
    defaults: {
        title: '',
        isbn: '',
        author: 'Tuntematon',
        releaseDate: 'Tuntematon',
        keywords: ''
    },

    // Initialize metodia kutsutaan,
    // kun uusi instanssi mallista luodaan
    initialize: function () {
        console.log('Minusta luotiin uusi instanssi!');

        // Modellin CRUD-operaatioiden kuuntelijat

        // Mikäli mikä tahansa attribuutti on muuttunut
        this.on('change', function () {
            // Tee jotakin!
        });

        // Mikäli title attribuutti on muuttunut
        this.on('change:title', function () {
            // Tee jotakin!
        });

        // Mikäli mallin validoinnissa tapahtui virhe
        this.on('invalid', function (model, error) {
            console.log(error);
        });
    },

    // Mallin validointi
    validate: function () {
        if (attribs.title === undefined) {
            return "Unohdit asettaa titlen!";
        }
    }
});
```

KUVIO 9. Backbone-mallin koodiesimerkki

3.5.3 Kokoelma

Moderneissa MVC-pohjaisissa sovellusratkaisuissa on tyypillistä tarjota ratkaisuja mallien instanssien ryhmittämiseksi. Backbonessa näitä ryhmiä kutsutaan kokoelmiksi (collection).

Mallien instanssien hallitseminen kokoelmissa mahdollistaa kuuntelijoiden asettamisen tietylle ryhmälle malleja. Tämä mahdollistaa ohjelmalogiikan luomisen tapahtumille, mikäli minkä tahansa kokoelmaan kuuluvan instanssin tila muuttuu. Tämän avulla vältytään tarpeelta manuaalisesti seurata yksittäisten instanssien tapahtumia. (Osmani, 2013, 19.)

Joka kerta kun data muuttuu asiakas- ja palvelinohjelmien välillä, muutokset tulee synkronisoida palvelimelle. Backbonessa kokoelmien yksittäiset mallien instanssit voidaan erottaa toisistaan uniikeilla id-, cid- ja idAttribute-muuttujilla. Id-attribuutti on käyttäjän asettama instanssin muuttuja, jonka avulla instanssit tulee kyetä erottelemaan toisistaan paikallisesti. Cid (client ID) on Backbonen automaattisesti luoma mallin identifioija. Cid on hyödyllinen tapauksissa, milloin mallille ei ole vielä asetettu id-arvoa, esimerkiksi mikäli instanssia ei ole vielä tallennettu palvelimelle tai sitä ei tallennetakaan. IdAttribute-arvon tehtävänä on identifioida instanssi palvelinpäässä. Arvo asetetaan automaattisesti instanssille, kun se haetaan pavelimelta. Tyypillisesti idAttribute on suoraan tietokantaan serialisoidun instanssin uniikki id-arvo. Kuviossa 10 on esimerkkimääritys Backbone kokoelmalle. (Osmani, 2013, 42.)

```
// Esimerkki Backbone Kokoelma
var LibraryCollection = Backbone.Collection.extend({
    defaults: {
        model: Book
    }
});
```

KUVIO 10. Backbone-kokoelman koodiesimerkki

3.5.4 Näkymä

Näkymät Backbonessa sisältävät logiikan mallien esittämiseksi käyttöliittymässä sekä käyttäjäinteraktioihin reagoimiseen. Ne eivät sisällä itsessään ominaisuuksia datan HTML-merkkikielelle muuntamiseksi, vaan käyttävät ulkoisia JavaScript template -kirjastoja siihen (Osmani, 2013, 35.). Backbonessa näkymät voidaan asettaa kuuntelemaan niihin liitettyjen mallien muutoksia. Kuuntelijat päivittävät näkymän välittömästi, kun muutoksia tapahtuu mallille, ilman ilman että koko sivu päivitettäisiin uudelleen. (Osmani, 60, 2013.)

Keskeinen osa näkymän määrittelyä on el-attribuutti. El on periaatteessa referenssi tiettyyn sivun DOM-elementtiin. Jokaisella näkymällä tulee olla oma elementtinsä. Backbone:ssa on kaksi tapaa liittää näkymä DOM-elementtiin: uusi elementti voidaan luoda dynaamisesti näkymälle ja myöhemmin lisätä DOM:iin, tai referenssi voidaan luoda jo olemassa olevaan elementtiin sivulla. El-attribuutti toisin sanoen kuvastaa HTML-merkkikielistä osiota näkymästä, joka esitetään sivulla. (Osmani, 2013, 35.)

Luodakseen HTML-merkkikielistä sisältöä näkymistä Backbone vaatii käyttämään jotakin kolmannen osapuolen JavaScript template-kirjastoa. Näitä ovat muun muassa Underscore sekä Handlebars.js. Template-kirjastojen pohjimmaisena tehtävä on kääntää merkkikielisen template.tiedoston sisältö JavaScript-funktioiksi. Käännettyä template-tiedostoa voidaan myöhemmin käyttää näkymien esittämisessä. (Osmani, 2013, 35.)

Backbonen näkymien render-metodin tehtävä on määrittää logiikka näkymän templaten esittämiseksi. Näkymän render-metodi käyttää template-tiedostoa välittämällä sille JSON-enkoodatun mallin attribuutit ja kutsumalla template-kirjaston kääntömetodeita. Template palauttaa takaisin näkymälle attribuuteista luodun HTML-merkkikielisen sisällön. Käännetty merkkikielinen sisältö lopulta asetetaan näkymän DOM-elementin sisällöksi, jolloin näkymä tulee osaksi käyttöliittymää. (Osmani, 2013, 35.)

Ohjelman käyttöliittymätapahtumiin reagoimiseen, näkymälle voidaan määrittää tapahtumien kuuntelijoita. Kuuntelijat voidaan asettaa kuuntelemaan näkymän el-

attribuutin määrittelemän DOM-elementin tapahtumia. Kuviossa 11 on esimerkkimääritys näkymälle ja sen tapahtumienkäsittelylle. (Osmani, 2013, 35.)

```
// Esimerkki Backbone Näkymä
var BookView = Backbone.View.extend({
    // Näkymää tulee osaksi listaa
    el: 'li',

    // Underscore template
    template: _.template(templateMerkkijono),

    // Näkymän DOM tapahtumat
    events: {
        'dblclick label': 'edit'
    },

    // Näkymä kuuntelee mallinsa muutoksia,
    // uudelleen esittäen näkymän
    initialize: function () {
        this.listenTo(this.model, 'change', this.render);
        this.listenTo(this.model, 'destroy', this.remove);
    },

    // Esitä yksittäisen kirjan tiedot näkymässä
    render: function () {
        this.$el.html(this.template(this.model.toJSON()));
        return this;
    },

    // Aseta näkymä muokkaustilaan
    edit: function () {
        this.$el.addClass('editing');
        this.$input.focus();
    }
});
```

KUVIO 11. Backbone-näkymän koodiesimerkki

3.5.5 Tapahtumat

Tapahtumat ovat Backbonen keino hallita laajoja kokonaisuuksia sekä välttää riippuvuuksia ohjelmakoodissa. Sen sijaan, että funktio kutsuisi toista funktiota

nimellään, se voidaan rekisteröidä käsittelijäksi tapahtumalle. Kuunneltu funktio suoritetaan tapahtuman ilmaantuessa. Backbone sallii tapahtumakäsittelijät liitettäväksi mihin tahansa olioon antaen sille mahdollisuuden kuunnella tai laukaista tapahtumia. (Osmani, 2013, 55.)

Backbonen tapahtumien merkityksellisin rooli on sovellusten komponenttien erottaminen toisistaan. Löyhä sidonta mahdollistaa mm. sen, että mallien koodin ei tarvitse tietää esimerkiksi, miten käyttöliittymä toimii. Välttämällä suoria referenssejä eri sovelluksen osien kesken voidaan helpottaa virheiden hallintaa tai jopa välttää kokonaan sovelluksen kaatuminen, mikäli jokin yksittäinen komponentti lakkaa toimimasta. (Osmani, 2013, 55.)

3.5.6 Reititys

Web-sivu, jonka voi olla mahdollista kirjainmerkitä, jakaa tai johon voidaan siirtyä selainhistoriassa, vaatii aina yksilöllisen URL-osoitteen. Backbonen reititys mahdollistaa osoiterivin URL-osoitteiden ohjauksen suoraan tiettyihin näkymiin sekä toiminnallisuudet selainhistoriassa taaksepäin palaamiseen. (Osmani, 2013, 55.)

Backbonen reititys tukee ainoastaan hash- tai hashbang-tyyppisiä osoitteita, joita on perinteisesti käytetty kuvaamaan sijaintia asiakaspuolen sovelluksissa. Käyttämällä URL fragment -tyyppisiä osoitteita vältetään selaimen uudelleenlataukset sivunaavigaation vaihtuessa. Hakukoneoptimoinnin sekä käyttäjäystävällisyyden kannalta olisi kuitenkin parempi, mikäli sovelluksen reititys tapahtuisi käyttämällä niin kutsuttuja pretty url -osoitteita. Ongelmaan on luotu ratkaisu käyttäen HTML5-standardin pushState-tukea, joille Backbone tarjoaa automaattisesti tuen. Mikäli selain ei tue HTML5-pushState ominaisuutta, Backbone osaa automaattisesti siirtyä takaisin käyttämään perinteisiä hash-osoitteita. (Osmani, 2013, 63.)

Backbonessa reitittimet hoitavat pitkälti perinteisen MVC-mallin Controllerin asemaa. Reittimen vastuulla on URL-ohjauksen lisäksi alustaa sivuun liittyvät näkymät, niiden mallit sekä kokoelmat. Backbonessa tulee sisäänrakennettuna

niin kutsuttu History API, joka mahdollistaa perinteisen selainhistorian käytön yhden sivun applikaatioissa. (Osmani, 2013, 62.)

3.5.7 RESTful persistenttisyys

Asiakas-palvelin-sovelluksissa kokoelmat sisältävät mallejen instansseja, jotka on perinteisesti noudettu palvelimelta. Backbone helpottaa merkittävästi palvelimen ja asiakkaan välistä REST-pohjaista kommunikaatiota. Se pyrkii abstraktoimaan perinteiset AJAX-kutsut yksinkertaiseen rajapintaansa, jota voidaan hyödyntää malleissa ja kokoelmissa. (Osmani, 2013, 53.)

Kokoelmien fetch-metodi mahdollistaa kokoelman noutamisen suoraan palvelimelta JSON-formaatissa. Fetch-metodi lähettää perinteisen HTTP GET-kutsun kokoelmassa määriteltyyn URL-osoitteeseen. Kun kokoelman data on haettu palvelimelta, se päivitetään asiakaspuolelle automaattisesti. (Osmani, 2013, 53.)

Vaikka Backbone kykeneekin hakemaan kokonaisia kokoelmia palvelimelta kerrallaan, niin muutokset malleihin suoritetaan yksittäin käyttäen save-metodia. Mallin instanssin save-metodia kutsuttaessa Backbone luo kutsun palvelimelle käyttäen kokoelmassa määritettyä URL-osoitetta. URL-osoitteeseen lisätään muutetun mallin tunnus. Mallin muutetut attribuutit välitetään palvelimelle HTTP PUT-kutsulla. Mallien instanssien poisto tapahtuu samalla logiikalla, kuten lisääminenkin, mutta palvelimelle lähetetään HTTP DELETE-kutsu. (Osmani, 2013, 54.)

Jokainen REST-rajapintakutsu sallii useita eri vaihtoehtoja määrittelemisekseen. Olennaisimpana kaikista on se, että jokaisen rajapintakutsun tulee sallia takaisinkutsumetodeita palvelimen virhe- ja onnistumisviesteihin reagoimiseen. Täten mahdollistaen selkeän virheidenhallinnan ja asynkronisen toiminnan. (Osmani, 2013, 55.)

3.6 Thorax

Thorax on Backbone.js:n liitännäinen, jonka tarkoituksena on luoda selkeämpi rakenne Backbone-sovelluksille. Thorax sai alkunsa alunperin Walmartin mobiili web-sovelluksesta, jossa sen oli tarkoitus luoda selkeä ja helpommin ylläpidettävä sovellusarkkitehtuuri sekä korvata joitakin Backbonen puutteita datan esittämisessä. (Osmani, 2013, 145.)

Osa Backbonen viehätysvoimasta koostuu siitä, että se antaa kehittäjälle vapaat kädet ohjelmiston arkkitehtuurin suunnitteluun. Laajemmissa projekteissa voi kuitenkin olla tärkeää tehdä selviä arkkitehtuurillisia valintoja alusta alkaen. Thorax pyrkii tuomaan Backbone-sovelluksille selkeän rakenteen, jonka päälle voidaan luoda monipuolisia sovelluskokonaisuuksia. (Osmani, 2013, 145.)

Thorax pyrkii tekemään tarkkoja valintoja käytettävän arkkitehtuurimallin sekä kolmannen osapuolten kirjastojen suhteen. Thorax käyttää Handlebars-kirjastoa templateissaan ja pyrkii tuomaan laajan kirjon aputoiminnallisuuksia template-tiedostojen esittämiseen. Thorax mahdollistaa hankalasti implementoitavien käsitteiden, kuten lapsinäkymien ja kokoelmanäkymien luonnin mutkattomasti, niin näkymissä kuin templateissa. (Osmani, 2013, 151.)

3.7 MarionetteJS

Backbone tarjoaa tarvittavan valmiuden pienten ja keskisuurten JavaScript-sovellusten toteuttamiseksi. Backbone ei ole kuitenkaan täysiverinen sovelluskehys. Se on enemmänkin ryhmä työkaluja, joka jättää paljon tärkeitä ominaisuuksia kehittäjän harteille, kuten muun muassa muistin ja näkymien hallinnan. (Osmani, 2013, 125.)

MarionetteJS tarjoaa useita ominaisuuksia, joita sovelluskehittäjä saattaa tarvita Backbonen ominaisuuksien lisäksi. Marionette on laaja sovelluskirjasto, jonka tarkoituksena on yksinkertaistaa laajojen sovelluskokonaisuuksien kehittämistä. Se pyrkii ratkaisemaan näihin liittyviä ongelmia tarjoamalla kokoelman yleisiä suunnittelumalleja, joiden on katsottu helpottavan merkittävästi suurten kokonaisuuksien hallintaa. (Osmani, 2013, 125.)

Marionette mahdollistaa suurten tapahtumapohjaisten ja entistä modulaarisempien sovellusten kehityksen. Kuten Thorax, se pyrkii tekemään selkeämpää linjausta ohjelmistoarkkitehtuurin suhteen.

Marionetten perusajatukseen kuuluu sovelluksen pilkkominen entistä pienempiin osiin ja näiden osien vastuualueiden eriyttäminen käyttämällä hyväksi havaittuja suunnittelumalleja. Marionette tarjoaa kehittäjälle mahdollisuuden korvata monia Backbonen puutteellisuuksia. Thoraxin tavoin se tarjoaa mahdollisuuden monimutkaisten näkymien esittämiseksi. Kuitenkin Thoraxista poiketen, Marionette tarjoaa myös laajan kirjon erilaisia apuvälineitä muun muassa automaattisen muistinhallinnan, löyhän sidonnan ja modulipohjaisuuden luomiseksi. (Osmani, 2013, 125.)

3.8 Palvelinpuolen JavaScript ja Node.js

JavaScript suunniteltiin alunperin selainpohjaiseksi ohjelmointikieleksi. JavaScript on kuitenkin yleishyödyllinen ohjelmointikieli ja sitä voidaan käyttää monessa eri viitekehyksessä, kuten mitä tahansa muutakin ohjelmointikieltä.

Jotta JavaScriptiä voitaisiin suorittaa palvelinympäristöissä, sen ajamiseksi tarvitaan tulkki. Tähän tarpeeseen kehitettiin Node.js, joka käyttää hyväkseen Googlen Chrome-selaimesta tuttua tehokasta V8-virtuaalikonetta. Node.js on tapahtumavetoinen JavaScript-ympäristö. Node:n erikoisuutena voidaan pitää sen tapahtumavetoisuutta ja I/O:n asynkronista toimintaa. Node.js on suunniteltu erityisesti skaalautuvien palvelinsovellusten sekä reaaliaikaisten ohjelmien toteuttamiseksi. (Hughes-Croucher & Wilson, 2012, 3.)

Web-sovellukset ovat perinteisesti olleet I/O-sidonnaisia, mikä tarkoittaa sitä, että sovelluksen suorituskyky on ollut riippuvainen pitkälti verkon ja kiintolevyn nopeudesta. Verkkosovellukset kuluttavat suurimman osan ajastaan odottaen I/O:ta esimerkiksi verkon yli tapahtuvan vastauksen saapumista tai tietokantakyselyn valmistumista. Useimmissa web-toteutuksissa I/O-kutsu keskeyttää koko ohjelman suorittamisen odottamisen ajaksi. Täten heikentäen merkittävästi ohjelman suorituskykyä ja sen käyttökokemusta. (Hughes-Croucher & Wilson, 2012, 33.)

Node.js:n asynkronisuus mahdollistaa sen erinomaisen suorituskyvyn käsiteltäessä lukuisia yhtäaikaisia yhteyksiä. Asynkronisuus mahdollistaa sen, että palvelimen ei enää tarvitse kuluttaa aikaa I/O-operaatioiden odottamiseen. Poiketen kuitenkin periteisemmistä asynkronisista toteutuksista, Node.js ei luo omaa säiettä tai käynnistä omaa aliohjelmaa jokaiselle tapahtumalle, vaan käyttää hyväkseen kevyempää I/O-pohjaista mallia, josta on poistettu keskeytykset. Keskeyttämätön I/O-malli parantaa sovelluksen suorituskykyä, skaalautuvuutta ja muistinkäyttöä, koska sen ei tarvitse jäädä odottamaan I/O-kutsun vastausta, eikä käyttää perinteisesti varsin raskaita säikeitä. (Hughes-Croucher & Wilson, 2012, 35.)

Node.js:lle on olemassa useita eri web-sovellus kirjastoja, jotka mahdollistavat helpon ja selkeän toteutustavan niin REST-rajapintojen, kuin perinteisten web-palvelintoteutusten luomiseksi.

3.9 Modulaarinen JavaScript

Puhuttaessa modulaarisista ohjelmista, tarkoitetaan usein löyhästi toisiinsa sidottuja toiminnallisuuksia, jotka on rakennettu moduuleiksi. Moduleita tulee voida ladata ja poistaa ohjelmasta tarvittaessa, jolloin ainoastaan tarvittavat ominaisuudet vievät resursseja. (Osmani, 2012, 139.)

Moduulit parantavat ohjelmiston skaalautuvuutta. Jokainen moduuli on oma itsenäinen osa ohjelmaa, joka kykenee toimimaan itsenäisesti muista moduleista. Ohjelmiston modulaarisuus mahdollistaa modulien poistamisen ja lisäämiseen tarvittaessa. (Osmani, 2012, 139.)

Kuten on mainittu aikaisemmin, löyhä sidonnaisuus parantaa ohjelmiston ylläpidettävyyttä poistamalla siitä riippuvaisuuksia. Modulipohjainen arkkitehtuuri helpottaa laajojen sovelluskokonaisuuksien rakentamista sekä suurien tiimien toimintaa. Jokainen sovelluskehittäjä voi työskennellä tiettyjen moduleiden parissa välittämättä muiden osien kehityksestä. Tällöin eri ohjelmiston osien kehitys tapahtuu rinnakkain ilman suurempia ristiriitoja. (Osmani, 2012, 139.)

Modulit mahdollistavat muuttujien paikallisuuden. JavaScriptissä jokaisella modulilla on automaattisesti oma nimiavaruutensa, paikallistaen siihen kuuluvat muuttujat lokaaleiksi muuttujiksi. Tämän lisäksi moduleille voidaan välittää globaaleja muuttujia, jotka toimivat modulin sisäisessä nimiavaruudessa. Muuttujien paikkalistamisella on useita suorituskyvyllisiä hyötyjä JavaScriptissä; ne helpottavat virtuaalikoneen roskienkeruuta sekä nopeuttavat muuttujaulottuvuuden ketjun läpikäymistä. (Osmani, 2012, 140.)

Valitettavasti JavaScriptin nykyinen iteraatio (ECMA-262) ei tarjoa itsessään ominaisuuksia modulaarisen ohjelmiston rakentamiseksi. Ratkaisu ongelmaan on tulossa todennäköisesti seuraavan ECMAScript-iteraation, eli ES Harmonyn myötä. (Osmani, 2012, 158.)

Vaikka JavaScript ei nykyisellään sisällä toiminnallisuuksia modulaarisen rakenteen määrittämiseksi, niin erilaiset kolmannen osapuolen moduliformaatit pyrkivät paikkaamaan tätä puutetta. Yleisimpinä korvaavina moduliformaatteina voidaan pitää Asynchronous Module Pattern (AMD)- ja CommonJS-moduliformaatteja. (Osmani, 2012, 139.)

AMD sekä CommonJS ovat molemmat toimivia moduliformaatteja, joilla on eri tavoitteet. AMD pyrkii selainpohjaisiin ratkaisuihin sekä asynkroniseen toimintaan. CommonJS puolestaan vastaa serveripuolen ratkaisuista. CommonJS pyrkii vastaamaan koko ohjelmiston ekosysteemin tarpeisiin huolehtimalla moduleiden lisäksi muun muassa I/O:sta, tiedostojärjestelmästä ja promiseista. AMD puolestaan vastaa ainoastaan moduleiden ja niiden riippuvuuksien määrittelystä ja lataamisesta, eikä se sisällä alemman tason toiminnalisuuksia. (Osmani, 2012, 139.)

3.9.1 Asynchronous Module Pattern (AMD)

Ansynchronous Module Definition (AMD) tarkoituksena on tarjota kokonaisvaltainen ratkaisu modulaarisen JavaScriptin kirjoittamiseksi. AMD:n moduuliformaatti on ehdotus siitä, kuinka modulit sekä niiden riippuvuudet voidaan ladata asynkronisesti. AMD:llä on lukuisia etuja, se on sekä asynkroninen että joustava eri käyttötarkoituksiin, mikä mahdollistaa ohjelmiston eri osioiden

löyhän sidonnan. AMD-moduliformaatti on kirjoitushetkellä todennäköisesti kehittynein moduliformaatti selainpuolen ohjelmistoilla sekä luotettava ponnahduslauta tulevaisuuden ES Harmonyn moduleihin. (Osmani, 2012, 140.)

Perinteisesti JavaScript web -sovelluksissa jokainen yksittäinen skripti ilmoitettiin <script> tagein sivuilla. Skriptien piti olla lueteltuina siinä järjestyksessä jossa ne ladattaisiin, jotta ne täyttäisivät toistensa riippuvuudet. Jokainen skripti ladattaisiin yksitellen palvelimelta törmäten joka kerta verkkoviiveeseen. Vasta kun kaikki sivuilla ilmoitetut skriptit oltaisiin ladattu ja käsitelty virtuaalikoneessa, voitaisiin ohjelma suorittaa. Lataukset tapahtuivat ei-asynkronisesti, mikä aiheutti pitkiäkin viiveitä sivulatauksien yhteydessä. (Osmani, 2013, 179.)

Vähentääkseen skriptien latauksesta johtuvia viiveitä kehittäjät pyrkivät yhdistämään kaiken ohjelmakoodin yksittäiseen skripti-tiedostoon. Yhdistämällä koko ohjelma yksittäiseen tiedostoon ei kuitenkaan ratkaista ongelmaa, jossa selainpuolen ohjelma joutuu lataamaan mahdollisesti valtavan määrän tarpeetonta koodia käsiteltäväkseen. Ratkaisuna ongelmaan syntyi selainpuolen AMD-lataimet, kuten RequireJS. (Osmani, 2013, 179.)

RequireJS on JavaScript-pohjainen tiedosto ja modulilaataja. Se on optimoitu käytettäväksi selainpuolen ohjelmistoissa, mutta sitä voidaan käyttää myös muissa JavaScript ympäristöissä, kuten Rhino:ssa ja Node:ssa. RequireJS mahdollistaa moduleiksi jaettujen koodiosuuksien sekä niiden riippuvuuksien asynkronisen ja automaattisen latauksen. (Require.js, 2013.)

Kaksi huomionarvoista AMD-konseptia ovat define ja require. Define-metodin tehtävänä on helpottaa moduleiden määrittämistä ja require-metodin vastata moduleiden riippuvuuksien lataamisesta. (Osmani, 2012, 140.)

Define-metodissa määritetään modulin tunnus sekä sen riippuvuudet asettamalla sille argumenteiksi lista nimetyistä moduleista. Definelle välitetään myös funktio, joka suoritetaan määritetyn modulin alustamiseksi. Require-metodi puolestaan vastaa JavaScript-tiedostojen dynaamisen lataamisen. Kuviossa 12 esitellään yksinkertainen AMD-moduli. (Osmani, 2012, 141.)

```
define(function (require) {
    // modulin määritys funktio
    // riippuvuudet foo ja bar määritetään lokaaleiksi muuttujiksi
    var foo = require('foo'),
        bar = require('bar');

    // modulin sisäinen privaatti funktio
    function doSomethingElse () {
        console.log('Something else');
    }

    var myModule = {
        doSomething: function () {
            console.log('Do something');
        }
    }

    // palautetaan modulin ominaisuudet jotka halutaan asettaa julkisiksi
    return myModule;
});
```

KUVIO 12. AMD-moduulimääritys

Vaikka RequireJS:stä on tullutkin lähes standardi JavaScript-modulien lataamiseksi selaimessa, siitä huolimatta kaikki JavaScript-kirjastot eivät käytä modulimäärityksiä hyväkseen. Kirjastot, kuten Jquery ja Underscore, eivät tue AMD-modulimäärityksiä eivätkä siis täten ole määritetty käyttämällä define-metodeja. Tästä huolimatta RequireJS kykenee lataamaan myös perinteiset JavaScript-kirjastot käyttämällä niin kutsuttuja shim-määrityksiä. Shim-määrittely mahdollistaa yhteensopimattomien kirjastojen automaattisen paketoimisen define-syntaksilla. Shim-määrittelyissä asetetaan kirjastoille niiden riippuvuudet ja määritetään moduleissa käytettävä nimiavaruus. (Osmani, 2012, 139.)

```
require.config({
    shim: {
        'lib/underscore': {
          exports: '_'
        },
        'lib/backbone': {
            deps: ['lib/underscore', 'jquery'],
            exports: 'Backbone'
        }
    }
});
```

KUVIO 13. RequireJS shim -määrittely

AMD tarjoaa lukuisia hyötyjä moderneissa web-sovelluksissa. Se tarjoaa selkeän toteutustavan joustavien moduleiden rakentamiseksi. Se tarjoaa huomattavasti selkeämmän ja siistimmän tavan skriptien lataamiseksi perinteiseen verrattuna, sen sijaan, että käytettäisiin perinteisiä <script>-tageja ja globaaleja nimiavaruuksia. AMD mahdollistaa helpon tavan itsenäisten moduleiden luomiseksi ja niiden riippuvuuksien huolehtimiseksi. RequireJS tarjoaa työkalut modulikokonaisuuksien yhdistämiseksi yhteisiin tiedostoihin ja näiden minimointiin vähentäen verkkoliikenteen viiveitä. Se tarjoaa myös mahdollisuuden moduleiden niin kutsutulle laiskalle lataamiselle, jolloin halutut modulit ladataan palvelimelta vasta kuin niille on oikeasti tarvetta. (Osmani, 2012, 149.)

3.9.2 CommonJS

CommonJS-moduliehdotus määrittää yksinkertaisen rajapinnan moduleiden käyttämiseksi serveripuolen JavaScript-sovelluksissa. Toisin kuin AMD, se pyrkii käsittelemään laajempaa kokonaisuutta huolehtien muun muassa IO:sta, tiedostojärjestelmästä ja promiseista jne. (Osmani, 2012, 150.)

CommonJS-modulit ovat rakenteeltaan uudelleenkäytettäviä osia JavaScript-koodia jotka määrittelevät julkisia osia ohjelmasta käytettäväksi muissa

moduleissa. Toisin kuin AMD, CommonJS moduleita ei tyypillisesti pakata funktioiden sisään. (Osmani, 2012, 150.)

CommonJS-modulit sisältävät kaksi pääosaa. Muuttujan nimeltään exports, joka sisältää osiot, jotka moduli tahtoo tehdä julkisesti saatavaksi muille moduleille. Sekä require-metodin, jota voidaan käyttää muiden moduleiden export ominaisuuksien tuomiseksi ohjelmaan. Kuviossa 14 on määritetty yksinkertainen CommonJS moduli. (Osmani, 2012, 152.)

```
// modulin riippuvuuksien lataaminen
var lib = require( "package/lib" );

// modulin toiminnallisuudet

function foo(){
    lib.log("hello world!");
}

// modulin julkisen rajapinnan määrittäminen
exports.foo = foo;
```

KUVIO 14. CommonJS-modulin määrittely

3.9.3 ES Harmony Modulit

Yksi lukuisista ominaisuuksista, joita tuleva ES Harmony ECMAScript-standardi tulee sisältämään, ovat modulit. ES Harmonyssä moduleiden riippuvuuksien lataaminen ja niiden julkisten rajapintojen määritteleminen on yksinkertaistettu import- ja export-ominaisuuksiin (Osmani, 2012, 159.). Vaikka ES Harmony on opinnäytetyön kirjoitushetkellä vielä määrittelyvaiheessaan, niin joitakin sen ominaisuuksia, kuten moduleita voidaan käyttää käyttämällä Googlen Traceur-kääntäjää.

ES Harmony-modulien toiminta ei poikkea juuri AMD- tai CommonJS-moduliformaattien toiminnasta. Import-määrittelyt vastaavat moduleiden julkisten rajapintojen lataamisesta ja määrittelemisestä modulin lokaaliin nimiavaruuteen. Export-määrittelyt vastaavat muiden moduliformaattien tapaan moduleiden julkisten rajapintojen määrittelystä. ES Harmony mahdollistaa myös moduleiden lataamisen ulkoisista lähteistä, esimerkiksi kolmannen osapuolen kirjastoista. (Osmani, 2012, 159.)

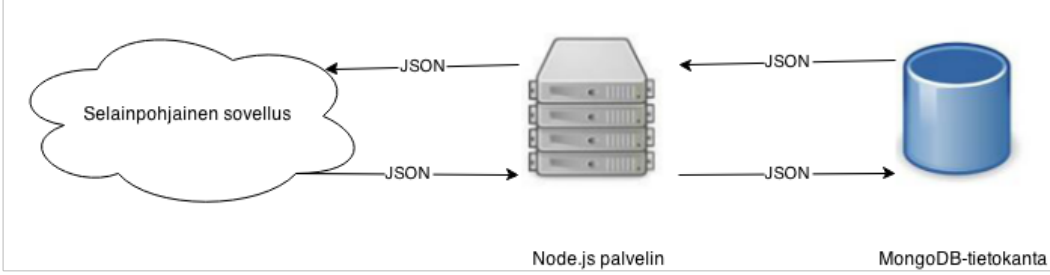
ES Harmony mahdollistaa moduleidensa avulla perinteisen olio-pohjaisen ohjelmointitavan, joka poikkeaa vahvasti JavaScriptin perinteisestä protyyppisestä kehitystavasta. ES Harmonyn olio-pohjaisuuden avulla voidaan selkeästi määrittää luokkia, konstruktoreita sekä määrittää niiden näkyvyyttä, ja näin voidaan selkeyttää modulin rakennetta ja helpottaa sen ylläpidettävyyttä. (Osmani, 2012, 159.)

3.10 MongoDB-tietokanta

MongoDB on tehokas, joustava ja skaalautuva yleiskäytännöllinen NoSQL-tietokanta. MongoDB on dokumentti-orientoitunut tietokanta. Toisin kuin perinteiset relaationalliset tietokannat, dokumentti-orientoituneet tietokannat korvaavat perinteisen käsitteen rivistä dokumentilla. MongoDB mahdollistaa kokoelmien sekä sisäisten dokumenttien määrittelyn, mahdollistaen monimutkaisten hierarkisten suhteiden esittämisen yksittäisellä tietueella. Tämä ajattelumalli sopii erinomaisesti yhteen modernien olio-pohjaisten sovellusten kanssa. (Hughes-Croucher & Wilson, 2012, 107.)

MongoDB on luonnostaan skeematon, mikä tarkoittaa että teoriassa tietokantaan voidaan tallentaa mitä tahansa dataa ilman aiempaa tietoa avaimista tai tietotyypeistä. Skeemoissa on kuitekin hyötynsä, sillä ne auttavat määrittämään, miltä datan kuuluisi näyttää. NoSQL-tietokannoissa skeemoista vastaaminen jätetään useimmiten sovelluksen huolehdittavaksi. Node.js-sovelluksissa on skeemojen luomiseen mahdollista käyttää erilaisia Object Document Mapping (ODM) -kirjastoja, kuten Mongoosea. (Wilson, 2012, 36.)

NoSQL-tietokantojen suurin hyöty moderneissa web-sovelluksissa on se että, tietokantaan tallennettu data on luonnostaan siinä muodossa ja rakenteessa, kuin sitä tarvitaan sovelluksessa. MongoDB esittää kokoelmansa JSON-formaatissa, joka on sama formaatti kuin asiakassovelluksen ja Node.js palvelimen välillä kommunikoinnissa. Yhtenäinen formaatti sovelluksen eri kerroksien välillä säästää aikaa, joka normaalisti kuluisi datan uudelleen jäsentelyyn eri formaatista toiseen. Kuviossa 15 esitetään eri sovelluskerrosten yhteensopivuutta.



KUVIO 15. Sovelluskerrosten välinen datan yhteneväisyys

MongoDB-tietokanta sopii erinomaisesti JavaScript-ohjelmistopino-malliin, sillä MongoDB:n komentotulkki on kirjoitettu JavaScript-ohjelmointikielellä. Tämä mahdollistaa JavaScript-skriptien luonnin esimerkiksi tietokannan datan manipuloinnille sekä ylläpidollisten toimintojen toteuttamiseksi.

4 ASIAKASREKISTERI-SOVELLUKSEN TOTEUTTAMINEN

Asiakasrekisteri-sovellus luotiin puhtaasti demonstroimaan selainpuolen ohjelmistojen toimintaa sekä sovelluskehitystä käyttäen teoriaosuudessa esiteltyjä tekniikoita.

Sovelluksessa tuli kyetä suorittamaan perus CRUD-operaatioita (lisää, lue, päivitä, poista), demonstroimaan relaatioiden toteutusta sovelluksen eri kerroksissa sekä vastaamaan datan validoinnista sekä sivunavigaatiosta.

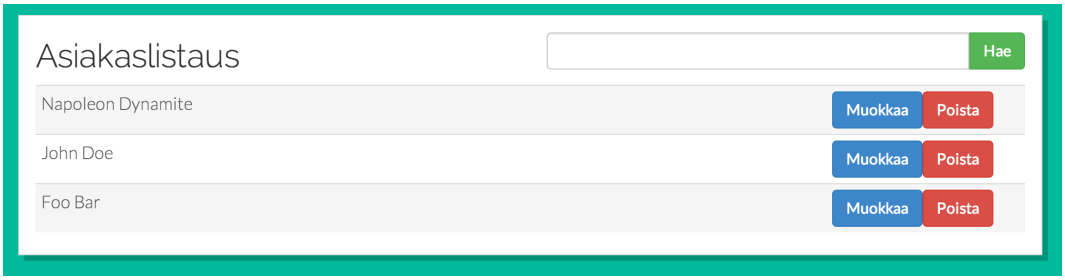
Selainpuolen sovellus toteutettiin käyttämällä Backbone.js-sovelluskehystä, sekä underscore template -kirjastoa. Selainpuolen sovelluksen toteuttamisessa pyrittiin käyttämään teoriaosuudessa hyväksitodettuja toteutustapoja. Sovelluksen arkkitehtuuri toteutettiin käyttäen modulaarista rakennetta ja löyhän sidonnan periaatteita. Sovelluksessa tuli kyetä esittämään kokoelmia käyttäjälle. Tämän vuoksi asiakasohjelman ominaisuuksia laajennettiin käyttämällä MarionetteJS-kirjastoa.

Palvelinpuolen REST-rajapinta toteutettiin käyttämällä Node.js-serveripään ajoympäristöä sekä Express -web-sovelluskehystä. Tietokantana käytettiin MongoDB-tietokantaa sekä Mongoose Object-Data Mapper (ODM):ää.

4.1 Selainpuolen sovelluksen toiminta

Asiakasrekisteri-sovelluksen käyttöliittymä oli jaettu kahteen eri näkymään: päänäkymään ja asiakasnäkymään.

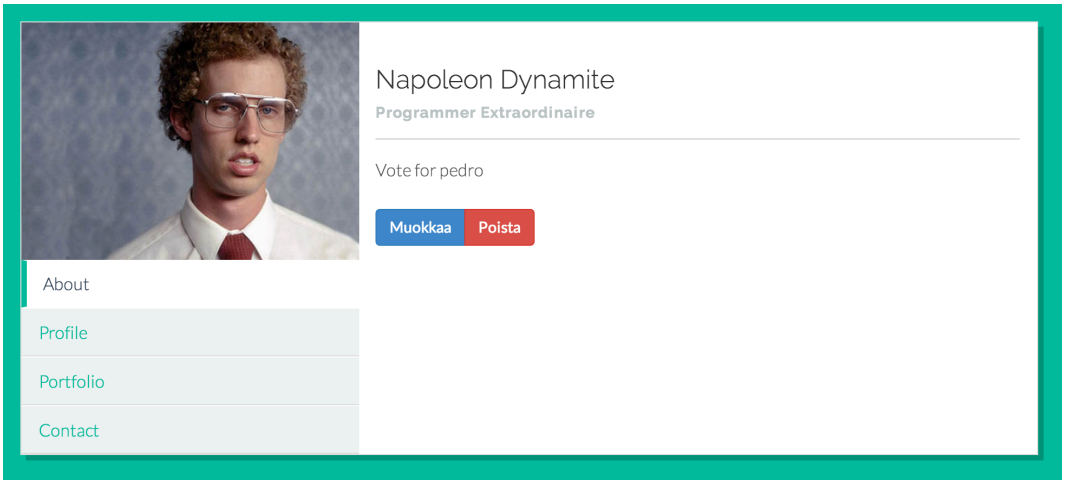
Päänäkymässä käyttäjän tuli kyetä listaamaan asiakasrekisteriin tallennetut asiakkaat, tarkastella niihin liittyviä tietoja sekä suodattamaan näytettävää dataa. Päänäkymässä näytettävä asiakaslistaus-kokoelma haettiin tietokannasta käyttämällä palvelinohjelmistoon siihen tarkoitukseen luotua REST-rajapintaa. Päänäkymä on havainnollistettu kuviossa 16.



KUVIO 16. Päänäkymä

Päänäkymä asetettiin kuuntelemaan kokoelmassa tapahtuvia muutoksia; mahdollistaen näkymän päivittämisen asiakkaita luotaessa, tai muokattaessa. Asiakaslistaus-kokoelmalle luotiin toiminnallisuudet yksittäisten mallien etsimiselle ja suodattamiselle kokoelmasta käyttäen RegExp-lausekkeita.

Asiakkaiden muokkaamiseen ja luomiseen oli olemassa omat näkymänsä. Asiakasnäkymässä käyttäjän tuli kyetä katsomaan ja muokkaamaan asiakkaisiin liittyviä perustietoja, sekä määrittelemään eri ryhmiä, joihin käyttäjä kuuluu. Kuviossa 17 on havainnollistettu asiakasnäkymä.



KUVIO 17. Yksittäisen asiakkaan näkymä

Ennen asiakkaan tallentamista kantaan siihen liittyvät kentät tuli kyetä validoimaan. Validointi suoritettiin käyttämällä Backbone-mallien validate-toiminnallisuutta. Asiakkaan tallentaminen aiheutti automaattisesti päivityksen kokoelmaan, johon asiakasmalli kuului. Kokoelman päivittäminen puolestaan päivitti päänäkymän, joka oli asetettu kuuntelemaan kokoelmassa tapahtuvia muutoksia. Mallien muutokset serialisoitiin automaattisesti kantaan käyttäen palvelinrajapinnan REST-rajapintoja.

Toteutuksessa käytettiin hyväksi Asynchronous Module Definition (AMD)-moduleita sekä RequireJS-modulilataajaa. Sovelluksessa käytettävien kirjastojen, kuten Backbonen ja Jqueryn, määrittelemiseksi luotiin oma konfiguraatiotiedosto. Tiedostossa määriteltiin avain-arvoparein modulien nimet sekä niihin liittyvät polut. Konfiguraatiossa määriteltiin myös AMD-yhteensopimattomien kirjastojen niinkutsutut shim-määrittelyt. Shim-määrittelyt mahdollistavat riippuvuuksien manuaalisen asettamisen sekä yhteensopimattomien kirjastojen automaattisen paketoimisen AMD-yhteensopivalla modulisyntaksilla.

4.2 Palvelinpuolen rajapinnan toteutus

REST-rajapinnan toteuttamiseksi luotiin yksinkertainen HTTP-palvelin käyttäen Express-web-sovelluskehystä. Toteutus tehtiin käyttäen Express-sovelluskehyksen vakiomäärittelyjä. Jokaiselle CRUD-operaatiolle, jota asiakaspuolen ohjelmasta kutsutaan, määritetiin oma rajapintatoiminnallisuutensa.

Tietokannan kanssa kommunikointia varten sekä tietokannan skeeman määrittelemiseksi palvelinpuolen sovelluksessa käytettiin Mongoose ODM-kirjastoa. Sovelluksessa määritettiin omat skeemansa niin asiakas-, kuin ryhmämalleille.

4.3 Huomioita sovelluksen suorituskyvystä

Yleisesti ottaen Single-Page Application -sovelluksia pidetään suorituskyvyiltään parempina kuin puhtaasti palvelinpäässä toimivat sovellukset. Todellisuus on kuitenkin monisyisempi.

Valtaosa ohjelmalogiikasta on siirretty selaimeen, täten mahdollistaen kevyemmän palvelinpuolen arkkitehtuurin. Palvelimen tehtäväksi on jäänyt lähinnä autentikointi sekä datan persistenttisyys, kun valtaosa liiketoimintalogikasta ja esityslogiikasta on siirretty asiakaspäähän. Siirtyminen palvelinpuolelta selainpuolelle ei aina kuitenkaan suju kitkattomasti.

Perinteisesti palvelimet ovat poikkeuksetta loistavia käsittelemään dataa ja selaimet esittämään sitä. Palvelinpuolen toteutusmalli voi kuitenkin suoriutua jopa paremmin, kuin selainpuolen sovellus, mikäli esitettävä data on pääsääntöisesti staattista sekä harvoin muuttuvaa. Palvelinarkkitehtuuri mahdollistaa nopean välimuistin käytön parantaen merkittävästi tietokanta-hauista johtuvia viiveitä data-intensiivisissä sovelluksissa. Välimuistin käyttö on kuitenkin käytännöllistä ainoastaan, kun data on staattista ja muuttuu harvoin. SPA-sovellusmallin käyttäminen on siis varsin riippuvainen toteutettavan palvelun tarpeista.

Opinnäytetyössä toteutetussa sovelluksessa SPA-kehitysmalli vaikuttaisi toimivan erinomaisesti, sillä sovellus oli tarkoitettu toimintatavoiltaan varsin dynaamiseksi.

4.4 SPA-sovelluskehitysmallissa havaitut puutteet

Opinnäytetyön toteutuksessa tutkituissa kehitystavoissa paljastui kolme ratkaisevanlaatuista, mutta korjattavaa ongelmaa: hakukoneoptimointi, sivunavigointi, sekä ensimmäisen sivulatauksen hitaus suurissa sovelluskokonaisuuksissa.

Tyypillisesti suurin este selainpohjaisten web-sovellusten implementoinnille varsinkin yritysten käytössä on ollut hakukoneoptimoinnin vaikeus käyttäessä pelkästään selainpuolen ratkaisuja. Perinteiset hakukonerobotit eivät kykene käsittelemään monimutkaisia JavaScript-toteutuksia sekä verkon yli tapahtuvia AJAX-kutsuja, täten tehden selainpuolen sovelluksien hakukoneoptimoinnista lähes mahdotonta. Toistaiseksi ainut mahdollinen ratkaisu on käyttää serveripuolen toteuksia sisällön staattiseksi esittämiseksi hakukoneille. Tämä toimintatapa aiheuttaa kuitenkin turhaa koodin duplikointia ja vaikeuttaa sovelluksen ylläpidettävyyttä, koska sama logiikka pitää kyetä luomaan myös esitettäväksi serverillä. Yksinkertaisimpana ratkaisuna ongelmaan voidaan pitää

päättömän Phantom.js selaimen käyttöä sivuston esittämiseksi. Hakukonerobotit voidaan automaattisesti uudelleenohjata palvelimelle, jossa sovellus esitetään Phantom.js:n kautta staattisessa muodossa, täten välttyen koodin duplikoinnilta.

Yhden sivun sovellusten käyttäessä HTML5-standardin history pushStaten mahdollistamia pretty urls -osoitteita, palvelinohjelmiston tulee olla tietoinen siitä, mitä URL-osoitteita selainpuolen sovellukset tukevat. Mikäli käyttäjä pyrkii suoraan osoiteriviltä muualle kuin etusivulle, selainpuolen sovellus ei alustu eikä täten kykene tulkitsemaan annettua osoitetta. Toistaiseksi ainut ratkaisu kyseiseen ongelmaan on duplikoida sovelluksen reititysominaisuudet myös palvelimelle (Osmani, 2013, 64.). Palvelinohjelmassa voidaan määrittää selainohjelmistossa käytetyt reitit ja antaa sen huolehtia osoitteiden uudelleenohjaamisesta asiakassovelluksen ymmärtämään muotoon, esimerkiksi käyttämällä hashtag -osoitteita. Siirtämällä sovelluksen reititys palvelimelle mahdollistettaisiin myös pretty urls-osoitteiden käyttö vanhemmilla selaimilla.

Selainpuolen sovellusten suorituskyky on lähes poikkeuksetta parempi, kuin perinteisissä palvelinpuolen toteutuksissa. Laajoissa sovelluskokonaisuuksissa selain voi joutua kuitenkin lataamaan suuren määrän JavaScript-tiedostoja. Vasta kun kaikki sovelluksen toimimiseen vaadittava ohjelmalogiikka on ladattu, voidaan sivusto esittää. Tämä voi aiheuttaa sivuston huomattavasti hitaamman ensilatauksen ja mahdollisesti moninkertaistaa verkkoviiveistä johtuvat suorituskyvylliset ongelmat. Osittaisena ratkaisuna ongelmalle voidaan pitää moduleiden yhdistämistä yksittäiseen tiedostoon sekä koodin minimointia tiedostokoon pienentämiseksi. Modulipohjaisissa sovelluksissa tämä voidaan helposti toteuttaa käyttämällä RequireJS optimizer -työkalua. RequireJS tarjoaa myös huomattavasti elegantimman toteutustavan, jossa require-lausekkeita käyttämällä voidaan sovelluksen ohjausrakenteissa laiskasti ladata ainoastaan tietyt modulit, jotka sovellus oikeasti vaatii toimiakseen, täten pienentäen ladattavan koodin määrää.

Edellä mainitut ongelmat vaativat poikkeuksetta kehittäjältä lisätyötä tai pahemmassa tapauksessa jopa koodin duplikointia asiakas- ja palvelinpuolelle, täten vaikeuttaen merkittävästi sovelluksen ylläpidettävyyttä. Yhtenä ratkaisuna edellämainittuihin ongelmiin olisi kyetä suorittamaan sama koodi niin asiakas-

kuin palvelinpuolellakin. Tällöin sivuston ensilataus voitaisiin toteuttaa suoraan palvelimelta, ladaten samalla tarvittavat JavaScript-modulit asiakaspäähän. Toteutustapa mahdollistaisi palvelin- ja asiakaspuolen ohjelmien parhaimpien puolien yhdistämisen samaan sovellukseen. Sivusto kyettäisiin esittämään käyttäjälle palvelinsovelluksista tutulla nopeudella, mutta myöhemmät interaktiot tapahtuisivat suoraan selainohjelmassa. Kirjastot, kuten backbone-serverside sekä Airbnb Rendr, pyrkivät toteuttamaan tämänkaltaista hybridi-rendeerausta selainpuolen sovelluksissa. Kirjastot eivät olleet kuitenkaan vielä opinnäytetyön kirjoitushetkellä tuotantokelpoisia.

5 YHTEENVETO

Opinnäytetyön tavoitteena oli tutkia mahdollisuutta parantaa web-sovellusten suorituskykyä ja käytettävyyttä sijoittamalla ohjelmalogiikka toimimaan suoraan käyttäjän selaimessa. Opinnäytetyössä tutkittujen menetelmien pohjalta oli tarkoituksena luoda yksinkertainen sovellus menetelmien demonstroimiseksi.

Tutkitut menetelmät osoittautuivat varsin onnistuneiksi. Niiden avulla voitiin ratkaista joitakin perinteistiin web-sovelluksiin liittyviä perustavanlaatuisia ongelmia, joita työn olikin tarkoituksena ratkaista.

Selainpohjaiset sovellukset tulevat näyttelemään suurta roolia tulevaisuudessa. Web-sovelluksilla on joitakin selkeitä hyötyjä verrattuna perinteisiin natiiveihin työpöytäsovelluksiin. Web-sovellukset ovat luonnostaan pilvi-pohjaisia, yhteisöllisiä sekä erittäin siirrettäviä, koska selainkehittäjät käyttävät nykyään yhä enemmän huomiota web-standardien noudattamiseen ja selainyhteensopivuuteen. Modernit web-sovellukset, jotka usein kulkevat termin HTML5:n alla, pääsevät käytettävyydessään jo hämmästyttävän lähelle niin kutsuttujen natiivien työpöytäsovellusten toimintaa.

Vaikka tutkitut sovellustavat kykenivätkin ratkaisemaan useita web-sovellusten perinteisiä ongelmia, niin uusien toteutustapojen käyttöönotto ei tullut ilman hintaa. Sijoittamalla koko sovelluksen ohjelmalogiikka selaimeen luotiin uusia ongelmia, joiden toteuttaminen voi olla haastavaa. Optimaalisin kehitysmalli moderneille web-sovelluksille voisikin olla malli, jossa yhdistettäisiin perinteisten palvelinohjelmien ja asiakasohjelmien parhaat puolet, häilyttäen asiakas- ja palvelinpuolten toimintojen selkeän erottelun toisistaan.

LÄHTEET

DiveintoHTML5. 2013. Manipulating history for fun and profit [viitattu 21.10.2013]. Saatavissa: http://diveintohtml5.info/history.html

Fastcompany. 2012. How one second could cost Amazon $1.6 billion in sales [viitattu 13.10.2013]. Saatavissa: http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

Fielding R. 2000. Architectural styles and design of network-based software architectures [viitattu 13.10.2013]. Saatavissa: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Garrett J. 2005. Ajax: A new approach to web applications [viitattu 15.10.2013]. Saatavissa: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/

Haverbecke M. 2007. Eloquent JavaScript [viitattu 14.10.2013]. Saatavissa: http://eloquentjavascript.net/

Hughes-Croucher T. & Wilson M. 2012. Node up and running. USA: O'Reilly Publishing.

Mozilla. 2013. A re-introduction to JavaScript [viitattu 14.10.2013]. Saatavissa. https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

Osmani, A. 2012. JavaScript Design Patterns. USA: O'Reilly Publishing.

Osmani, A. 2013. Developing Backbone.js Applications. USA: O'Reilly Publishing.

RequireJs. 2013. RequireJs Documentation [viitattu 11.10.2013]. Saatavilla: http://requirejs.org/

Strangeloops. 2012. State of Mobile eCommerce Performance [viitattu 13.10.2013]. Saatavissa:

http://www.strangeloopnetworks.com/resources/research/state-of-mobile-ecommerce-performance/

Strongloop. 2012. How to combine Node.js promises with with Q – an alternative to callbacks [viitattu 16.11.2013]. Saatavissa: http://strongloop.com/strongblog/promises-in-node-js-with-q-an-alternative-to-callbacks/

W3. 2012. About W3C [viitattu 13.10.2013]. Saatavissa: http://www.w3.org/Consortium/

W3. 2013. HTML5 [viitattu 20.10.2013]. Saatavissa: http://www.w3.org/TR/html5/

Wilson M. 2012. Building Node Applications With MongoDb and Backbone. USA: O'Reilly Publishing.