Jari Räisänen

MVC-mallin mukainen web-kehitys JavaScriptillä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

6.9.2013

Metropolia

Tekijä Otsikko Sivumäärä Aika	Jari Räisänen MVC-mallin mukainen web-kehitys JavaScriptillä 41 sivua + 14 sivua liitteitä 6.9.2013
Tutkinto	insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaajat	CTO, Aki Lehtinen FT, Lehtori Vesa Ollikainen

Insinöörityössä käsitellään moderneja JavaScript-tekniikoita sekä MVC-arkkitehtuurin käyttöä web-kehityksessä. Alussa käydään läpi yleisesti web-sovelluksen rakennetta, jotta loppuosan asiat olisivat helpommin omaksuttavissa. Tavoite on lähestyä aihetta yhdistämällä teoriaa ja käytännön esimerkkejä sopivassa suhteessa. Esimerkkejä pyritään sitomaan myös liitteenä olevaan kokonaiseen sovellukseen.

MVC-arkkitehtuurin lisäksi työssä pyritään selventämään modulaarisen ohjelmoinnin etuja. Modulaarinen ohjelmointi ei ole ollut itsestäänselvyys JavaScript-sovelluksissa, joten sitäkin voidaan pitää uutena asiana JavaScriptin osalta.

Yksi iso osa tätä työtä on web-sovelluksen suorituskyky. Suorituskykyä parantavia tekijöitä pyritään tuomaan esiin useissa luvuissa, mutta erityisesti luvussa kahdeksan, joka on omistettu kokonaan suorituskyvylle.

JavaScriptin ympärille rakennettujen MVC-kirjastojen määrä on tällä hetkellä erittäin suuri, eikä kaikkia kirjastoja tai tekniikoita ollut mahdollista esitellä yhdessä työssä. Siksi tässä työssä keskitytään niihin tekniikoihin, joita olen itse päässyt käyttämään työelämässä. Monet kirjastot noudattavat hyvin samankaltaisia perusperiaatteita, joten suurin osa teoriasta pätee muihinkin kirjastoihin.

Avainsanat	JavaScript, Backbone, MVC-arkkitehtuuri, Web-kehitys, RequireJS, AMD, jQuery, modulaarinen ohjelmointi, DOM

Author	Jari Räisänen
Title	Web Development with MVC and JavaScript
Number of Pages	41 pages + 14 appendices
Date	6 September 2013
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructors	Aki Lehtinen, CTO Vesa Ollikainen, PhD

The object of this final year project was to introduce modern JavaScript technologies. The project also examined how well JavaScript and MVC-pattern work together. There are a lot of code examples in this project along with the theory. Some of the examples are also related to the application found in the appendices.

One chapter of this project tells about modular programming. Modular programming overall is quite a new technique in JavaScript.

One big part of the whole project was finding ways to improve performance in JavaScript applications.

At the moment the variety of JavaScript MVC-libraries is huge. It is impossible to cover all the libraries in one project so limiting the scope was necessary. All the libraries are still quite similar so most things written apply to other libraries too.

Keywords	JavaScript, Backbone, MVC-pattern, Web development, RequireJS, AMD, jQuery, modular programming, DOM

Sisällys

Lyhenteet

AJAX	AJAX on lyhenne sanoista Asynchronous JavaScript And XML. AJAX on termi, joka tarkoittaa tekniikoita, joiden avulla sivuston osia voidaan päivittää päivittämättä koko sivua.
AMD	AMD on lyhenne sanoista Asynchronous Module Definition. AMD on ohjelmointirajapinta, jonka avulla ohjelmiston osa eli moduuli ja sen riippuvuudet voidaan ladata yhtaikaisesti.
CDN	CDN on lyhenne sanoista Content Delivery Network. CDN on datakeskusten kokonaisuus, jonka avulla Internetin sisältöä pyritään tarjoamaan käyttäjille mahdollisimman nopeasti.
CSS	CSS on lyhenne sanoista Cascading Style Sheets. CSS on WWW-sivujen tyylin kuvaukseen kehitetty kieli.
DNS	DNS on lyhenne sanoista Domain Name System. DNS on nimipalvelujärjestelmä, jonka avulla verkkotunnukset muutetaan IP-osoitteiksi.
DOM	DOM on lyhenne sanoista Document Object Model. DOM on ohjelmointirajapinta, jonka avulla HTML-dokumenttien sisältöä päästään muokkaamaan esimerkiksi JavaScript-kielen avulla.
HTML	HTML on lyhenne sanoista Hypertext Markup Language. HTML on erityisesti verkkosivujen kuvauksessa käytetty kuvauskieli, joka sisältää hypertekstiä eli tekstiä, jossa on hyperlinkkejä.
HTTP	HTTP on lyhenne sanoista Hypertext Transfer Protocol. HTTP on siirtoprotokolla, jota selaimet ja palvelimet käyttävät tiedonsiirrossa.
jQuery	jQuery on avoimen lähdekoodin JavaScript-kirjasto, joka sopii muun muassa DOM-elementtien valitsemiseen ja animaatioiden tekemiseen.
JSON	JSON on lyhenne sanoista JavaScript Object Notation. JSON on JavaScriptistä riippumaton ja ulkoasultaan yksinkertainen tiedonsiirtomuoto.

MVC	MVC on lyhenne sanoista Model-View-Controller. MVC-arkkitehtuuri on ohjelmistoarkkitehtuuri, joka erottaa käyttöliittymän ja tiedon toisistaan.
MySQL	MySQL on yksi suosituimmista relaatiotietokantaohjelmistoista.
NoSQL	NoSQL on lyhenne sanoista Not only SQL. NoSQL tarkoittaa tietokantoja, jotka poikkeavat tavallisesta relaatiotietokannasta.
PHP	PHP tulee sanoista Hypertext Preprocessor. PHP on ohjelmointikieli, jota käytetään muun muassa Web-palvelimilla.
REST	REST on lyhenne sanoista Representational State Transfer. REST on arkkitehtuurimalli, joka on rakennettu HTTP-protokollan ympärille. REST:n avulla voidaan toteuttaa ohjelmointirajapintoja.
SPA	SPA on lyhenne sanoista Single Page Application. SPA on verkkosovellus tai WWW-sivu, joka sisältää vain yhden sivun ja muistuttaa toiminnaltaan paljon normaalia työpöytäohjelmaa.
SQL	SQL on lyhenne sanoista Structured Query Language. SQL on IBM:n kehittämä kyselykieli, jonka avulla voidaan käsitellä relaatiotietokantaa.
URL	URL on lyhenne sanoista Uniform Resource Locator. URL on merkkijono, jonka avulla kerrotaan tietyn verkkosivun paikka Internetissä.

1 Johdanto

JavaScript on yksi tämän hetken suosituimmista kielistä web-kehityksessä, koska se löytyy jokaisesta selaimesta. JavaScriptin avulla voidaan myös toteuttaa verkkosivuja, joiden sisältö ladataan dynaamisesti yksi osa kerrallaan ilman koko sivun uudelleenlatausta. Nämä niin kutsutut SPA-ohjelmat (engl. Single-page application) ovat suosittuja tällä hetkellä, koska niiden avulla sivuston käyttökokemuksesta saadaan erittäin sujuva.

JavaScriptin kasvava suosio on tuonut ongelmia etenkin suurissa sovelluksissa, joissa koodia kertyy huomattavia määriä. Koodin uudelleenkäytettävyys ja ylläpidettävyys olivat hyvin vaikeita tehtäviä vanhan tyylin JavaScript-koodauksessa, jossa koko sovellus kirjoitettiin yleensä yhteen tiedostoon.

Tämän insinöörityön on tarkoitus selvittää, miten modernin JavaScript-sovelluksen ylläpidettävyyttä saadaan parannettua MVC-arkkitehtuurin avulla. Toisaalta työn on tarkoitus olla yleisopas MVC-mallin ja JavaScript-kielen käyttöön web-kehityksessä. Työssä keskitytään erityisesti selaimessa suoritettavaan asiakaspään JavaScriptiin, mutta esitellään pintapuolisesti myös palvelinpään JavaScript-tekniikoita.

Työ on tehty Bublaa Oy:lle, joten siinä on painotettu erityisesti kyseisessä yrityksessä käytettäviä tekniikoita. Teorian ohessa on tavoitteena rakentaa toimiva esimerkkisovellus hyödyntäen JavaScriptin MVC-kirjastoja.

Insinöörityössä arvioidaan JavaScriptin ominaisuuksia, heikkouksia, vahvuuksia ja sen ympärille rakennettuja kirjastoja. Työssä käsitellään myös web-sovellusten suorituskyvyn parantamista, koska se on hyvin tärkeä osa kaupallista sovelluskehitystä. Nopeammat latausajat pitävät käyttäjät tyytyväisenä ja pienemmät tiedonsiirtokustannukset ovat yritykselle eduksi.

Koska työn on tarkoitus olla käytännönläheinen opas, se sisältää runsaasti koodiesimerkkejä käsitellyistä tekniikoista ja teorioista. Koodiesimerkit on eroteltu omiksi kappaleikseen ja ne on pyritty pitämään mahdollisimman lyhyinä ja selkeinä. Tämän lisäksi liitteenä on yhtenäinen sovellus, johon monet käytännön esimerkit pyritään sitomaan.

Ensiksi työssä kerrotaan hieman JavaScriptin alkutaipaleesta ja kehityksestä kohti nykypäivää. Tämän jälkeen käydään läpi JavaScriptin erikoispiirteitä verrattuna muihin ohjelmointikieliin. Kolmannessa kappaleessa perehdytään yleisesti MVC-arkkitehtuuriin sekä JavaScriptin MVC-kirjastoihin.

Neljännessä luvussa käydään läpi DOM:a (engl. Document Object Model), joka sitoo verkkosivun eri osat yhteen sekä jQuery-kirjastoa, jonka avulla DOM:a käsitellään. Viidennessä luvussa palataan taas MVC:n pariin ja tutustutaan Backboneen, joka on yksi JavaScriptin MVC-kirjastoista.

Kuudes luku kertoo modulaarisesta ohjelmoinnista ja siitä, millaisiin osiin sovellus kannattaa jakaa. Samalla kerrotaan myös JavaScriptin moduulinlatauskirjastosta RequireJS:stä. Työn loppupuolella käydään läpi suorituskyvyn parantamista web-kehityksessä. Viimeisenä uutena aiheena on palvelinpään JavaScript. Kuten edellä mainitsin, palvelinpään JavaScriptiin ei perehdytä kovin syvällisesti, vaan tarkoituksena on vain kertoa, millaisia tekniikoita palvelinpäässä nykyään käytetään. Aivan viimeisenä on luonnollisesti yhteenveto ja sen jälkeen liitteenä esimerkkiohjelma. Esimerkkiohjelman lähdekoodikin kannattaa lukea läpi, koska se on kokonaisuudessaan kommentoitu.

2 JavaScript-kieli

JavaScript-kieltä alettiin kehittää 90-luvulla, kun verkkosivuista haluttiin dynaamisempia. JavaScriptin kehitystyön aloitti 90-luvun alkupuolella Brendan Eich, joka kehitti kielen yhtiölle nimeltä Netscape. JavaScriptistä tuli osa Netscape 2 -selainta vuonna 1995 ja sen haluttiin olevan Javaa helposti lähestyttävämpi kieli amatööriohjelmoijille. Netscape otti mallia Microsoftista, joka oli tuolloin kehittänyt Visual Basic -kielen C++-kielen helpommaksi versioksi.

JavaScript ei liity suoraan mitenkään Javaan, eikä sen alkuperäiseen kehittäjään Sun Microsystemsiin, vaikka sen nimessä sana Java esiintyykin. Nykyisin Javaa kehittää Oracle. Kehityksen alkuvaiheissa JavaScriptiä kutsuttiinkin Mochaksi ja LiveScriptiksi. [1.]

Ensimmäinen standardoitu versio JavaScriptistä saatiin kesäkuussa 1997, kun Netscape lähetti JavaScriptin standardointiorganisaatio Ecma Internationalsille. Standardoidun

version nimi on ECMAScript ja Netscapen JavaScript on vain yksi sen murteista. Myös Microsoft kehitti Internet Explorer -selaimeensa oman murteensa ECMAScriptistä, ja se kulkee nimellä JScript. [1.]

JavaScriptiä kirjoitetaan tavallisesti joko suoraan HTML-koodin sekaan tai erilliseen tiedostoonsa, joka liitetään HTML-sivulle. Seuraavana on yksinkertainen esimerkki HTML-sivusta, jossa käytetään JavaScriptiä avaamaan huomioikkuna:

```
<html>
    <head>
        <title>Ensimmäinen esimerkki</title>
        <script>
            function tervehdi() {
                alert('Hei maailma!');
            }
        </script>
    </head>
    <body>
        <button onclick="tervehdi()">hei</button>
    </body>
</html>
```

Esimerkin ensimmäisellä rivillä kerrotaan, että dokumentti on verkkosivu. Itse verkkosivu alkaa ja loppuu html-tageihin. Tagilla tarkoitetaan tässä yhteydessä merkintää, joka alkaa ja loppuu kulmasulkeeseen. Lopettava tagi erotetaan aloittavasta vinoviivalla.

Head-tagit pitävät sisällään sivun otsikon ja JavaScript-funktion. Tässä tapauksessa funktio avaa huomioikkunan, kun sitä kutsutaan. Sivun body-osassa piirretään ruudulle painike, jota painamalla aiemmin määritelty funktio suoritetaan.

Käytännön esimerkin jälkeen on hyvä perehtyä hieman tarkemmin JavaScriptiin. Seuraavissa aliluvuissa käydään läpi muutamia yksinkertaisia asioita, jotka on JavaScriptissä toteutettu hieman toisin kuin muissa ohjelmointikielissä.

2.1 Yksi numerotyyppi

JavaScriptissä on vain yhden tyyppisiä numeroita, jotka ovat 64-bittisiä liukulukuja ja vastaavat Javan double-tyyppiä. Erillistä kokonaislukutyyppiä ei JavaScriptissä ole olemassa. Näin vältytään muun muassa lyhyiden kokonaislukujen (Javassa short-tyyppi) ylivuoto-ongelmilta.

2.2 Olioita ei kopioida

JavaScriptissä kopioidaan ainoastaan viittaus olioon. Itse olioita ei kopioida lainkaan. Sama käytäntö on voimassa myös Java-ohjelmointikielessä. Tämän pinnallisen kopioinnin (engl. shallow copy) etu on sen nopeus, eikä se hidastu tietomäärän kasvaessa. Myös muistivuodoilta vältytään, koska JavaScriptissä on automaattinen roskienkeruu. [2.]

2.3 Prototyyppiperintä

JavaScriptissä oliot voivat periä ominaisuuksia suoraan toisilta olioilta eikä erillisiä luokkia ole lainkaan [3, s. 29]. JavaScriptin oliomalli perustuu luokkien sijasta prototyyppeihin, mikä tarkoittaa käytännössä sitä, että uusia olioita luodaan laajentamalla ja kloonaamalla jo olemassa olevia olioita [4]. Jokainen olio sisältää viittauksen omaan prototyyppiinsä sekä taulukon ominaisuuksista, jotka eroavat prototyypistä [5].

2.4 Globaalit muuttujat

Yksi JavaScriptin huonoimmista puolista on sen riippuvuus globaaleista eli kaikille näkyvistä muuttujista. Mikä tahansa ohjelman osa pystyy muokkaamaan globaaleita muuttujia, ja tämä voi aiheuttaa arvaamattomia seurauksia ohjelman toiminnassa.

Ongelma ei niinkään ole se, että JavaScript sallii globaalit muuttujat, koska ovathan ne esimerkiksi Javassakin sallittuja. JavaScript sen sijaan vaatii globaaleita muuttujia, koska se ei käytä lainkaan linkittäjää (engl. linker) eri koodiosioiden väliseen tiedonsiirtoon. JavaScript sitoo kaikki globaalit muuttujat ja funktiot globaaliin olioon, joka on selaimessa nimeltään window. [3, s. 101.] Paikalliset muuttujat esitellään JavaScriptissä var-sanan avulla ja globaalit muuttujat ilman etuliitettä.

```
<script>
    // Paikallinen muuttuja
    var paikallinen = 1;
    // Globaali muuttuja
    globaali = 2;
</script>
```

2.5 Näkyvyysalue

Näkyvyysalue (engl. scope) tarkoittaa sitä osaa koodista, joka pystyy käsittelemään tiettyä muuttujaa. Kaikissa muissa C:n sukuisissa kielissä on tapana, että aaltosulut luovat aina uuden näkyvyysalueen, ja tämän alueen sisällä esitellyt muuttujat eivät näy alueen ulkopuolelle.

JavaScriptissä näin ei kuitenkaan ole, vaan esimerkiksi if-lauseessa esitellyt muuttujat näkyvät koko funktion näkyvyysalueella. Tästä syystä JavaScriptissä kaikki muuttujat on syytä esitellä jokaisen funktion alussa väärinkäsitysten välttämiseksi. [3, s. 102.]

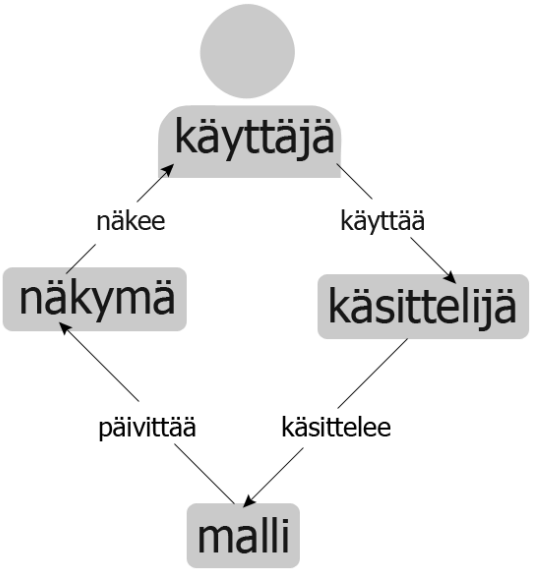
3 MVC-arkkitehtuuri

Kuva 1 esittää MVC-mallin rakennetta. MVC tulee sanoista model-view-controller eli malli-näkymä-käsittelijä ja sen tarkoitus on pitää ohjelmiston käyttöliittymä ja itse tieto erillään toisistaan. Käyttäjä siis näkee ainoastaan näkymiä ja käyttää käsittelijää muokatakseen malleja.

Malli kertoo, kuinka ohjelma tallentaa ja käsittelee tietoa. Näkymä taas määrittää sen, mitä käyttäjän ruudulle tai muulle päätelaitteelle piirretään näkyviin. Käsittelijän eli ohjaimen on tarkoitus toimia tiedonvälittäjänä käyttäjän ja mallin välillä. Käsittelijä siis ottaa käyttäjän syötteet vastaan ja muokkaa niiden perusteella mallia. [6.]

Malli voi olla joko passiivinen tai aktiivinen. Jos malli on passiivinen, sen ei tarvitse kertoa näkymälle milloinkaan tilansa muutoksista. Passiivinen malli voi esiintyä esimerkiksi yksinkertaisessa tekstinkäsittelyohjelmassa, jossa näkymä päivittyy ainoastaan käyttäjän syöttäessä tietoa. Tällöin käsittelijä huolehtii näkymän päivityksestä ja käskee näkymän hakea uudet tiedot mallilta.

Aktiivinen malli ilmoittaa sitä kuunteleville näkymille ja käsittelijöille tilansa muutoksista, jolloin näkymä pystyy piirtämään käyttäjälle uudet tiedot näkyviin ja käsittelijä pystyy esittämään mahdolliset uudet komentopainikkeet. [7.]



Kuva 1. MVC-arkkitehtuuri, jossa aktiivinen malli

3.1 Kirjastot ja kehykset

JavaScriptin ympärille on rakennettu runsas määrä kirjastoja (engl. library) ja kehyksiä (engl. framework), jotka helpottavat koodin kirjoittamista, selkeyttävät rakennetta tai tuovat uutta toiminnallisuutta. Kirjastot tuovat yleensä mukanaan vain tietyn toiminnallisuuden, eivätkä ne määrittele esimerkiksi hakemistorakennetta, kun taas kehykset määräävät koko projektin rakenteen tarkasti. Kehykset siis sitovat kehittäjän tiettyyn ohjelmistonkehitystyyliin ja antavat vähemmän vapauksia kuin kirjastot, joita voidaan yleensä käyttää hieman vapaammin. [8.]

3.2 MVC-arkkitehtuuri ja JavaScript

Modernit JavaScript-kehykset ja -kirjastot auttavat ohjelmoijia järjestämään koodiaan MVC-arkkitehtuurin mukaisesti. Monet kehykset ja kirjastot eivät kuitenkaan noudata MVC-arkkitehtuuria kirjaimellisesti, vaan usein ne käyttävät niin sanottua MV*-arkkitehtuuria. MV*-arkkitehtuurissa malli ja näkymä ovat aina mukana, mutta käsittelijä korvataan tilanteeseen sopivalla osalla. [6.] Kirjastojen ja kehysten kehittäjät ilmeisesti havaitsivat, että perinteinen MVC ei sovellu täydellisesti asiakaspään JavaScript-ohjelmiin.

Esimerkiksi Backbone-kirjasto, joka on yksi tämän hetken suosituimmista MVC-kirjastoista, ei sisällä käsittelijää. Backbonessa näkymä sisältää käsittelijän loogiset operaatiot ja reititin auttaa ohjelmaa pitämään kirjaa tilastaan. Kumpaakaan näistä ei kuitenkaan voida pitää käsittelijänä perinteisen MVC:n mukaan. [9.] Backbonen reitittimestä puhutaan enemmän luvussa 5.5.

3.3 MVC:n hyödyt web-kehityksessä

MVC:n hyödyt tulevat esiin, kun rakennetaan modernia web-sovellusta, joka sisältää paljon dynaamisia osia ja käyttöliittymäkomponentteja. MVC-kehykset nopeuttavat sovelluksen kehitystä, koska niiden avulla koodista saadaan uudelleenkäytettävää. [6.] MVC-arkkitehtuuri mahdollistaa sen, että sovelluksen näkymiä päivitetään sitä mukaa, kun niihin liittyvien mallien data muuttuu. Tällöin ei tarvitse ladata koko sivua uudelleen, vaan päivittää vain tarvittavat näkymät.

Ilman MVC-kehystä web-sovelluksen koodi sidotaan yleensä suoraan ruudulle piirrettyihin DOM-elementteihin, joita päivitetään JavaScriptillä. Tällöin koodin ylläpidettävyys ja uudelleenkäytettävyys muuttuu vaikeaksi varsinkin isommissa projekteissa. DOM eli Document-Object-Model on rajapinta, joka mahdollistaa pääsyn sivun sisältöön ja sen päivittämisen dynaamisesti komentosarjojen eli skriptien avulla [10]. DOM:a käsitellään tarkemmin luvussa neljä.

Hyvä apuväline DOM:n käsittelyyn on jQuery-kirjasto, joka sisältää suuren määrän valmiita DOM-funktioita. JQueryä ei ole kuitenkaan tarkoitettu sovelluksen rakenteen määrittämiseen, vaan DOM-elementtien manipulointiin, animointiin, tapahtumien hallintaan ja Ajax-kutsujen käsittelyyn. JQuery-kirjastosta puhutaan tarkemmin myöhemmin luvussa 4.3.

4 DOM

DOM (engl. Document Object Model) on malli, joka määrittää, kuinka kaikki verkkosivulla olevat elementit liittyvät toisiinsa ja itse verkkodokumenttiin. Valittuamme DOM-elementin voimme tehdä siihen muutoksia. [11.]

JavaScript sisältää erilaisia funktioita DOM-elementtien valintaan esimerkiksi id:n tai elementin tyypin perusteella [12]. Edellisessä luvussa mainittu jQuery-kirjasto helpottaa myös DOM-elementtien valintaa. Seuraava esimerkki havainnollistaa DOM-elementin valintaa sekä perinteisen JavaScriptin että jQueryn avulla.

```
<p id=”kappale”>DOM</p>

<script>
    // Elementin valinta JS:n omalla metodilla
    var muuttuja = document.getElementById(’kappale’);

    // Saman elementin valinta jQueryn avulla
    var muuttuja2 = $(’#kappale’);
</script>
```

JQuery tarjoaa siis lyhyemmän kirjoitusasun samalle operaatiolle. $-merkki tarkoittaa, että haemme koko DOM:sta, ja #-merkki tarkoittaa, että haemme elementtiä id:n perusteella. Esimerkin suorituksen jälkeen molempien muuttujien (muuttuja ja muutuja2) arvona on alussa esitelty p-elementti.

4.1 Puurakenne ja solmut

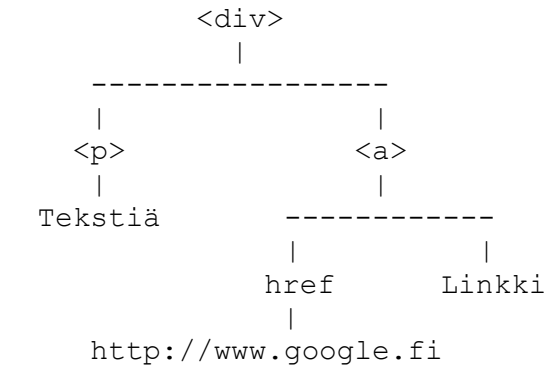
DOM kuvataan puurakenteen avulla ja jokainen elementti DOM:ssa on solmu (engl. node). Ensinnäkin kaikki HTML-elementit ovat solmuja, mutta myös elementin sisältämä teksti on oma tekstisolmunsa. Seuraavassa esimerkissä on siis kaksi solmua: elementti-solmu ja tekstisolmu. Tekstisolmu on elementti-solmun sisällä, joten sitä kutsutaan lapsisolmuksi. Vastaavasti elementtisolmu on tekstisolmun vanhempisolmu. [11.]

```
<span>Olipa kerran DOM.</span>
```

Myös elementtien ominaisuudet ovat omia ominaisuussolmujaan. On olemassa myös muita solmutyyppejä, mutta elementtisolmut, tekstisolmut ja ominaisuussolmut käsittävät valtaosan HTML-dokumentin sisällöstä.

```
<div>
    <p>Tekstiä</p>
    <a href=’http://www.google.fi’>
        Linkki
    </a>
</div>
```

Edellisen HTML-koodin puurakenne näyttäisi seuraavalta:



4.2 DOM:n läpikäynti

DOM:a voidaan käydä läpi eri tavoin. Perus JavaScriptin avulla DOM:n läpikäynti ei ole yhtä sulavaa kuin jQueryllä, mutta se on kuitenkin mahdollista. Luvussa 4.3.2 puhutaan tarkemmin DOM:n läpikäynnistä jQueryn avulla.

Otetaan aluksi käytännön esimerkki, joka koostuu HTML- ja JavaScript-osasta:

```
<html>
    <body>
        <p class=’kappale’ id=’kappale1’>Tekstiä</p>
    </body>

    // JavaScript-koodi
    <script>
        var kappale =
            document.firstChild.firstChild;
        kappale =
            document.getElementByTagName('p');
        kappale =
            document.getElementById('kappale1');

        var luokka = kappale.childNodes[0];
        var teksti = kappale.childNodes[1];
    </script>
</html>
```

Edellisessä esimerkissä on yksinkertainen HTML-sivu, jossa p-elementti on body-elementin sisällä. P-elementtiin voidaan päästä käsiksi eri tavoin. Ensimmäisessä tapauksessa p-elementti haetaan firstChild-metodin avulla. Tämä ei ole kovin kätevä tapa, koska DOM:n rakenne on tiedettävä tarkalleen, jotta päästään käsiksi oikeaan

elementtiin. Tässä tapauksessa p-elementin täytyy olla body-elementin ensimmäinen lapsisolmu ja body:n täytyy olla koko dokumentin ensimmäinen lapsisolmu.

Toisessa tapauksessa sivulta haetaan kaikki p-elementit getElementByTagName-metodin avulla. Tässä tapauksessa meidän ei tarvitse tietää DOM:n rakenteesta mitään, vaan saamme kaikki p-elementit riippumatta siitä, missä kohtaa DOM:a ne sijaitsevat.

Kolmannessa tapauksessa haemme p-elementin id:n avulla. Tämä on kätevä tapa, jos DOM:sta on tarkoitus löytää yksittäinen elementti. Viimeiset kaksi muuttujaa näyttävät, miten p-elementin luokka-attribuuttiin ja sisältöön päästään käsiksi. [11.]

4.3 jQuery-kirjasto

jQuery on JavaScript-kirjasto, joka helpottaa HTML-dokumentin läpikäyntiä. jQuery yksinkertaistaa myös tapahtumienkäsittelyä, animointia sekä AJAX:n käyttöä. AJAX on termi, joka tarkoittaa tekniikoita, joiden avulla sivuston osia voidaan päivittää päivittämättä koko sivua. AJAX ei siis ole yksittäinen tekniikka, vaan se koostuu useista tekniikoista (HTML, CSS, JavaScript, DOM, XML, XSLT ja XMLHttpRequest-olio). AJAX:a ei käydä tässä työssä sen tarkemmin läpi, mutta on hyvä tietää, mitä kyseisellä termillä tarkoitetaan. [13.]

jQuery:n avulla päästään käsiksi JavaScriptin monimutkaisiin ominaisuuksiin hyvin pienellä määrällä koodia. jQuery on siis työkalu kehitystyön nopeuttamiseen. jQueryn käyttöönotto tapahtuu samaan tapaan kuin minkä tahansa muunkin JavaScript-kirjaston eli kirjoittamalla script-elementti HTML-sivun head-osioon. Src-määreellä kerrotaan tiedoston paikallinen sijainti, joka on alla olevassa esimerkissä sama kuin itse HTML-sivun sijainti.

```
<html>
    <head>
        <script type="text/javascript"
            src="jQuery-1.9.1-min.js"></script>
    </head>
    <body>
        <p id='teksti'>jQuery ei toimi!</p>
        jQuery(document).ready(function() {
            jQuery('#teksti').html('jQuery toimii!');
        });
```

```
    </body>
</html>
```

Edellisessä esimerkissä jQuery:a käytetään sivun body-osassa p-elementin tekstin muuttamiseen. Ensiksi tarkastetaan, että dokumentti on kokonaan latautunut. Tämä tehdään jQueryn ready-funktiolla, josta puhutaan tarkemmin luvussa 4.3.4. Sitten etsitään elementti id:n perusteella ja korvataan sen sisältämä teksti uudella html-funktion avulla. [14.]

4.3.1 $-merkki

Edellisen luvun esimerkissä jQuery:a käytettiin tarkoituksella ilman $-merkkiä, koska halusin ensimmäisen esimerkin pysyvän mahdollisimman selkeänä. $-merkki jQuery:ssä tarkoittaa samaa kuin sana jQuery. Edellisen luvun esimerkki voitaisiin siis kirjoittaa seuraavasti:

```
<html>
    <head>
        <script type="text/javascript"
            src="jQuery-1.9.1-min.js"></script>
    </head>
    <body>
        <p id='teksti'>jQuery ei toimi!</p>
        $(document).ready(function() {
            $('#teksti').html('jQuery toimii!');
        });
    </body>
</html>
```

Tämä onkin suositeltava tapa käyttää jQuery:a, koska se tekee koodista lyhyempää ja selkeämpää etenkin, jos jQuery-funktioita on runsaasti koodissa. Kaikki tulevat esimerkit tässä työssä tulevat myös käyttämään pelkästään $-merkintää.

4.3.2 Valitsimet

Yksinkertaisin käyttötarkoitus jQuerylle on HTML-elementtien valitseminen ja niiden käsittely. jQuery käyttää CSS-kielen valitsimia sekä muutamia erityisiä valitsimia. Useimmiten HTML-elementtejä valitaan joko luokan tai id:n perusteella, mutta jQuery mahdollistaa myös huomattavasti monimutkaisemmat valitsimet. [15.]

```
$("#idTesti");          // id-valitsin
```

```
$(”.luokkaTesti”);      // luokka-valitsin
$(”div.luokkaTesti”);  // tarkennettu luokka-valitsin
$(”#idTesti .luokkaTesti”); // yhdistetty valitsin
```

Edellä on esitelty muutamia yleisimpiä tapauksia valitsimista. Ensimmäisessä kohdassa haetaan id:tä, jonka nimi on ”idTesti”. Tämä viittaa vain yhteen elementtiin, koska samaa id:tä ei saa olla kahdella elementillä.

Toisessa kohdassa elementtejä haetaan luokan perusteella. Luokka määrittää HTML-elementille nimen, mutta id:stä poiketen monella elementillä voi olla sama luokan nimi. Luokan avulla voimme siis käsitellä montaa elementtiä kerralla.

```
// Kahta samaa id:tä ei saa olla
<div id=”yksi”></div>
<div id=”yksi”></div>

// Samaa luokkaa käyttäviä elementtejä voi olla useita
<div class=”monta”></div>
<div class=”monta”></div>
```

Kolmannella rivillä valitsinta on tarkennettu, jolloin jQuery etsii ”luokkaTestiä” ainoastaan div-elementeistä. Tämä parantaa suorituskykyä, koska jQueryn tarvitsee käydä pelkästään div-elementit läpi [15].

Viimeisessä kohdassa kaksi valitsinta on yhdistetty, jolloin jQuery hakee ”luokkaTesti” nimistä luokkaa ”idTesti” -nimisen id:n sisältä. Length-ominaisuudella voidaan tarkistaa, palauttiko valinta mitään.

```
if($(”.luokkaTesti”).length) {
    console.log(”valinta ei ollut tyhjä.”);
}
```

4.3.3 Valintojen käyttäminen

Kun valinta on määritetty, voidaan sille kutsua funktioita. jQueryssä funktiot on kirjoitettu siten, että samaa funktiota voidaan käyttää arvojen lukemiseen sekä asettamiseen.

```
$(”.luokkaTesti”).html(”asetetaan teksti”);
var teksti = $(”.luokkaTesti”).html();
$(”.luokkaTesti”).find(”p”).html();
```

Edellisessä esimerkissä html-funktiota käytetään ensiksi arvon asettamiseen ja sitten arvon hakemiseen. Kolmannelta riviltä nähdään, että funktioita voidaan myös kirjoittaa peräkkäin eli ketjuttaa silloin, kunhan edeltävä funktio palauttaa jQuery-olion. [16.] Ketjuttaminen parantaa myös suorituskykyä, koska samaa elementtiä ei tarvitse hakea useaan kertaan DOM:sta [17, s. 5].

4.3.4 Koodin suorittaminen DOM:n latauduttua

Nykyiset JavaScript-sovellukset suorittavat JavaScriptiä usein vasta, kun DOM on kokonaan latautunut. Perus-JavaScriptistä löytyy tapahtuma (engl. event) nimeltä window.onload, joka laukeaa silloin, kun kaikki dokumentin elementit ovat DOM:ssa ja kaikki kuvat sekä muut resurssit ovat latautuneet. [18.] Tämä ei kuitenkaan ole suositeltava tapa, koska esimerkiksi kuvien latautuminen saattaa hyvinkin kauan ja sillä aikaa voitaisiin käsitellä DOM:ia ja suorittaa JavaScriptiä [17, s. 11].

jQuery tarjoaa tähän ongelmaan ratkaisuksi ready-funktion, jolle voidaan antaa parametrina koodi, joka suoritetaan sitten, kun DOM on kokonaan latautunut.

Koodin suoritus DOM:n latautumisen jälkeen onnistuu myös ilman ready-funktiota tai tapahtumien kuuntelua sijoittamalla suoritettava JavaScript-koodi aivan HTML-sivun loppuun. Tämä lisää suorituskykyä, koska useimmat selaimet eivät lataa muuta sivua sillä aikaa, kun JavaScriptiä käännetään. [17, s. 12.] Tähän aiheeseen palataan tarkemmin luvussa kahdeksan.

5 Backbone

Edellisessä kappaleessa käsiteltiin DOM:a sekä sen läpikäyntiä helpottavaa jQuery-kirjastoa. jQuery on erinomainen apuväline DOM-elementtien käsittelyyn, mutta sen avulla emme voi jakaa koodia järkeviin osiin MVC-mallin mukaisesti. Tarvitsemme siis MVC-kirjaston. Tämän hetken suurin ongelma MVC-kirjastojen valinnassa on se, että kirjastoja on valtavasti ja juuri itselleen sopivan kirjaston valinta voi osoittautua hyvinkin vaikeaksi. Hyvä paikka kirjastojen vertailuun on Addy Osmanin ylläpitämä verkkosivusto todomvc.com, missä on toteutettu pienehkö ohjelma kaikilla tämän hetken suosituimmilla MVC-kirjastoilla.

Backbone on yksi tämän hetken suosituimmista JavaScript MVC-kirjastoista [19]. Backbonen lisäksi suosittuja MVC-kirjastoja ovat tällä hetkellä mm. Ember ja AngularJS. Backbone erottuu monista muista kirjastoista keveydellään. Pakattu versio vie tilaa ainoastaan neljä kilotavua ja sisältää vain tärkeimmät asiat eli kuvaukset malleista, tapahtumista (engl. event), näkymistä, kokoelmista, käsittelijöistä ja riippuvuuksista.

Backbonen ainoa kirjastoriippuvuus on Underscore.js, joka lisää Backboneen noin 60 yleiskäyttöistä funktiota. Backbone onkin helposti sovitettavissa monenlaisiin projekteihin, koska se ei sido käyttäjää tiettyyn koodaustyyliin, vaan antaa vain apuvälineitä projektin rakenteen ylläpitämiseen. [20, s. 165.]

5.1 Käyttöönotto

Backbonen ja Underscoren käyttöönotto tapahtuu lisäämällä verkkosivun head-osioon kirjastoihin viittaava polku. Tämän jälkeen kirjastot ovat käytössä kyseisellä sivulla. On syytä huomata, että Underscore-kirjasto on ladattava ensin, koska Backbone on riippuvainen siitä. [21.]

JavaScriptin kirjastot eivät ole osa JavaScriptin moottoria eivätkä ne siis löydy suoraan selaimesta. Kirjastot ladataan muiden tiedostojen tapaan sivustoa ylläpitävältä palvelimelta.

```
<html>
  <head>
    <title>Kirjastojen käyttöönotto</title>
    <script src="../libs/underscore-min.js"></script>
    <script src="../libs/backbone-min.js"></script>
  </head>
  <body>
    <p>Kirjastot ovat nyt käytössä.</p>
  </body>
</html>
```

5.2 Näkymä

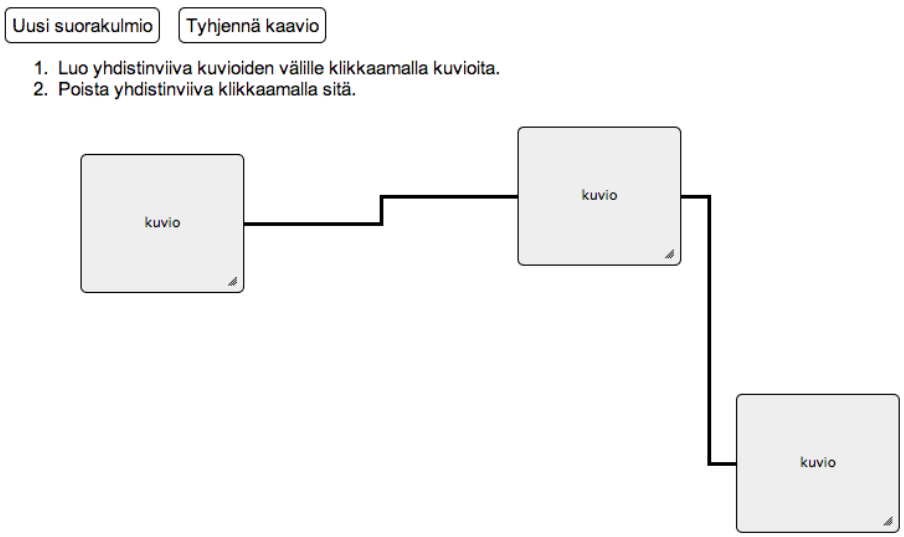
Näkymien (engl. view) tarkoitus Backbonessa on piirtää mallien sisältämä tieto näkyviin sekä kuunnella tapahtumia ja reagoida niihin [22, s. 2]. Jokainen näkymä sisältää vähintään kaksi funktiota, jotka ovat nimeltään initialize ja render.

Initialize-funktio toimii näkymän alustajana (engl. constructor) eli sitä kutsutaan aina, kun näkymä luodaan. Oletuksena render on tyhjä funktio, mutta se on syytä ylikirjoittaa piirtämään tarvittavat sivupohjat (engl. template). Sivupohjista kerrotaan tarkemmin luvussa 5.2.5. Render-funktiota kutsutaan aina, kun näkymä halutaan piirtää uudestaan näkyviin. [20, s. 169.] Yksinkertaisimmillaan näkymä näyttää seuraavalta:

```
EsimerkkiNäkymä = Backbone.View.extend({
    initialize: function() {
        console.log('näkymän alustus');
    },
    render: function() {
        console.log('näkymän piirto');
    }
});
```

Hieman konkreettisempi esimerkki näkymästä saadaan tarkastelemalla liitteenä olevaa esimerkkiohjelmaa. Esimerkkiohjelma on pienehkö sovellus, jonka avulla voi piirtää suorakulmioita ruudulle, liikutella niitä ja yhdistää niitä toisiinsa viivoilla. Esimerkkiohjelmaa voisi siis käyttää hyvin yksinkertaisten kaavioiden piirtämiseen, ja se on laajennettavissa helposti monimutkaisemmaksi sovellukseksi. Esimerkkiohjelmassa on kaksi näkymää. Päänäkymä on nimeltään main.js ja suorakulmionäkymä taas rectangle.js.

Kuvassa 2 on kuvankaappaus esimerkkiohjelmasta. Kuvankaappauksessa näkyy tilanne, jossa on luotu kolme ilmentymää suorakulmionäkymästä. Yläreunassa olevat painikkeet ja selitystekstit kuuluvat päänäkymään. Näkymä itsessään on vain MVC-mallin looginen osa eikä siitä näy käyttäjälle muuta kuin sen piirtämät sivupohjat, jotka ovat HTML-koodia.



Kuva 2. Kuvankaappaus esimerkkiohjelmasta

Yhdellä sivulla voi siis olla lukuisia näkymiä. Käytännössä sivu kannattaa jakaa järkeviin kokonaisuuksiin ja yhtä näkymää voi käyttää sivulla useaan kertaan, kuten yllä olevassa kuvassa suorakulmionäkymää. Näkymään voi myös liittää malleja. Malleja voi olla yksi, useita tai ei yhtään. Esimerkkiohjelmassa kuhunkin suorakulmionäkymään liittyy kuviomalli. Päänäkymään taas ei liity lainkaan mallia.

Ohjainta Backbonessa ei ole lainkaan, mutta sivustoon liittyy yleensä myös yksi reititin (luku 5.5). Esimerkkiohjelma on sen verran yksinkertainen, että siinä ei ole reititintä lainkaan.

5.2.1 el-muuttuja

Jokaisella näkymällä on tieto omasta DOM-elementistä, johon näkymä piirtää sivupohjansa. Tätä kyseistä DOM-elementtiä kutsutaan el-muuttujaksi. El-muuttuja on oletuksena tyhjä div-elementti, mutta se voidaan myös asettaa osoittamaan jo olemassa olevaan DOM-elementtiin. Asettaminen onnistuu vasta, kun haluttu DOM-elementti on ladattu sivulle. [20, s. 170.]

Kuten edellisessä luvussa mainittiin, näkymiä voi olla yksi tai useampia. Jokaiselle DOM-elementille ei ole syytä tehdä omaa näkymäänsä.

```
EsimerkkiNäkymä = Backbone.View.extend({
    el: $('.esimerkki-luokka')
});
```

Esimerkkikoodissa el-muuttuja asetetaan viittaamaan olemassa olevaan elementtiin, jonka luokka on nimeltään ”esimerkki-luokka”. Luokan nimi on käärityt sulkuihin ja dollarimerkkiin, koska elementti etsitään jQueryn avulla. Tämän jälkeen näkymän el-muuttujaan voidaan sijoittaa tietoa render-funktiossa seuraavasti:

```
EsimerkkiNäkymä = Backbone.View.extend({
    el: $('.esimerkki-luokka'),

    render: function() {
        $(this.el).html('Tämä menee el muuttujaan');
    }

});
```

html-funktio tyhjentää el-muuttujan ja sijoittaa sinne lainausmerkkien sisällön.

5.2.2 this-sana

Edellisessä koodiesimerkissä esiintyi this-sana, jonka sivuutimme täysin. JavaScriptissä this on varattu sana, jota voidaan käyttää kaikissa funktioissa. This viittaa aina siihen näkymään, josta kyseistä funktiota on kutsuttu.

This-sanan avulla päästään käsiksi näkymän muuttujiin funktion sisältä. Edellisessä esimerkissä this-sana viittaa EsimerkkiNäkymä-nimiseen näkymään ja this.el viittaa näkymän el-nimiseen muuttujaan. Katso myös luku 7.1. ”Sisäfunktion this-sana”.

5.2.3 Tapahtumiin reagoiminen

Backbonen avulla tapahtumienkäsittelijät voidaan listata näkymässä events-olioon. Olioon määritellään avaimeksi tapahtuman tyyppi sekä valitsin ja arvoksi suoritettavan funktion nimi.

```
EsimerkkiNäkymä = Backbone.View.extend({
    events: {
        'click .luokanNimi': 'suoritettavaFunktio'
    },

    suoritettavaFunktio: function() {
        console.log('Elementtiä klikattiin.');
    }
});
```

Valitsin voi olla esimerkiksi luokka-, id-määre tai mikä tahansa CSS-valitsin. Jos valitsinta ei määritetä, kuuntelija kuuntelee koko el-muuttujaa.

5.2.4 Näkymän päivittäminen datan muuttuessa

Joskus on syytä sitoa näkymä kuuntelemaan mallin muutoksia ja piirtämään itsensä tarvittaessa uudestaan. Tällöin mallille asetetaan näkymässä kuuntelija on-funktiolla.

```
EsimerkkiNäkymä = Backbone.View.extend({
    initialize: function() {
        this.esimerkkiMalli.on(
            'change',         // Tapahtuman tyyppi
```

```
                this.render,      // Vastakutsu
                this              // Konteksti
            );
        },
        render: function() {
            console.log('Piirretään näkymä uudelleen.');
        }
});
```

On syytä huomata, että on-funktion toinen parametri this.render on funktio, eikä sille kirjoiteta sulkeita perään. Jos sulkeet kirjoitettaisiin, funktio suoritettaisiin heti eikä vasta tapahtuman lauettua. [20, s. 171.] Parametrina annettavia funktioita kutsutaan vastakutsuiksi (engl. callback).

Kolmas parametri on konteksti, jonka avulla vastakutsu linkittyy oikeaan funktioon. Underscore-kirjasto tarjoaa myös bindAll-funktion, joka sitoo kaikki sille annetut funktiot samaan kontekstiin. Tällöin erillistä kontekstia ei tarvitse välittää. [23.]

```
EsimerkkiNäkymä = Backbone.View.extend({
    initialize: function() {
        // Annetaan bindAll-funktiolle konteksti
        // ja siihen sidottavien funktioiden nimet.
        _.bindAll(this, 'render');

        // Nyt kontekstia ei määritetä on-funktiossa
        this.esimerkkiMalli.on('change', this.render);
    },
    render: function() {
        console.log('Piirretään näkymä uudelleen.');
    }

});
```

5.2.5 Sivupohjat

Sivupohja (engl. template) on tapa erottaa HTML-koodi JavaScriptistä. Sivupohja voidaan erottaa omaksi koodilohkokseen tai vielä selkeämmin omaan tiedostoonsa. Näin JavaScript-koodi säilyy selkeämpänä ja helpompana ylläpitää.

Mikäli sivupohjat kirjoitetaan omiin tiedostoihin, ne voidaan ladata näkymän alussa esimerkiksi RequireJS-kirjaston avulla. RequireJS:n toimintaan perehdytään tarkemmin luvussa 6.2.

Backbonen alikirjasto Underscore tarjoaa sivupohjien tekemiseen template-funktion. Seuraavassa koodiesimerkissä luodaan HTML-sisältöä kyseisen funktion avulla.

```
var sivupohja = _.template(
    "<p>Tämä on sivupohja</p>"
);

sivupohja(); // Piirtää p-tagin teksteineen
```

Usein on tarpeen sisällyttää sivupohjaan muuttujia tai yksinkertaista logiikkaa. Tämä onnistuu <%= %>, <%- %> ja <% %> -merkintöjen avulla. Ensimmäinen merkintä on muuttujia varten ja sallii kaikki merkit. Toinen merkintä muuntaa tietyt erikoismerkit HTML-nimiksi ja estää näin koodin suorittamisen. Kolmas merkintä taas on ehtolauseita ja silmukoita varten.

```
var sivupohja = _.template("
    <p>Nimeni on <%- nimi %></p>
    <p>Ikäni on <%= ika %></p>
    <p>
        Olen siis <% if(ika > 65) { %>
            vanha
        <% } else { %>
            nuori
        <% } %>
    </p>
");

sivupohja({ nimi: '&Jari', ika: 24 });

// Tuotettu HTML-koodi (Huomaa &-merkin muuttuminen):
// <p>Nimeni on &amp;Jari</p>
// <p>Ikäni on 24</p>
// <p>Olen siis nuori</p>
```

5.3 Malli

Mallien (engl. model) voidaan ajatella olevan abstrakteja versioita tietokannan raa'asta tiedosta. Backbonen mallit sisältävät tietoa sekä logiikkaa eli funktioita, joilla tietoa käsitellään. Backbonen malli saadaan luotua kutsumalla extend-funktiota Backbone.Modelille. [20, s. 166.]

Näkymän tapaan myös mallissa alustajana toimii initialize-funktio, joka suoritetaan aina, kun tehdään uusi malli. Initialize-funktion määrittäminen ei ole pakollista, mutta suotavaa. [22, s. 5.]

Mallille voidaan asettaa ominaisuuksia antamalla alustajalle parametrina JavaScriptin objekti, johon ominaisuuksien avaimet ja arvot on listattu tai käyttämällä set-funktiota. [22, s. 6.] Yksinkertaisimmillaan mallin luominen ja ominaisuuksien asettaminen tapahtuu seuraavasti:

```
EsimerkkiMalli = Backbone.Model.extend({
    initialize: function () {
        console.log('mallin alustaja');
    }
});

var malli = new EsimerkkiMalli({ 'pituus': 5 });
var malli2 = new EsimerkkiMalli();
malli2.set({ pituus : 5 });
```

Mallien ominaisuuksien lukeminen tapahtuu vastaavasti get-funktiolla, jolle annetaan parametrina halutun arvon nimi. Myös oletusarvojen asettaminen on mahdollista. Tällöin mallille määritellään defaults-ominaisuus, joka on avain-arvopareja sisältävä objekti. Malli käyttää oletusarvoisia ominaisuuksiaan aina, mikäli niitä ei ylikirjoiteta.

Mallille voidaan myös kirjoittaa validate-funktio, joka on oletusarvoisesti tyhjä. Validate-funktioon voidaan itse määrittää säännöt, joita mallin ominaisuuksien tulee noudattaa. Edelliseen esimerkkiin viitaten voisimme määritellä vaikkapa, että pituuden täytyy olla vähintään kymmenen. Silloin validate-funktio näyttäisi seuraavalta:

```
EsimerkkiMalli = Backbone.Model.extend({
    validate: function (ominaisuudet) {
        if (ominaisuudet.pituus < 10) {
            return 'pituus täytyy olla vähintään 10';
        }
    }
});
```

Jos validate-funktio menee läpi, sen ei kuulu palauttaa mitään. Mikäli jokin ominaisuus on virheellinen, validate-funktio voi palauttaa joko virheestä kertovan merkkijonon tai erillisen virheen (engl. error). Jos validate-funktio palauttaa virheen, set- ja save-funktiot eivät jatka toimintojaan ja error-tapahtuma laukeaa. Tätä tapahtumaa voidaan myös kuunnella on-funktiolla, josta kerrotaan seuraavassa luvussa enemmän. [20, s. 167.]

Esimerkkiohjelmassamme on kaksi mallia: yhteysmalli connection.js sekä kuviomalli shape.js. Alla on kuviomallin koodi. Kuviomalli sisältää kuvion sijaintitiedot, koon, tyy-

pin, id:t sekä kuvion tekstin. Mallilla on myös tieto siitä, minne mallin tiedot tallennetaan. Tässä tapauksessa tallennuspaikka on selaimen paikallinen varasto (engl. local-storage), mutta se voisi myös olla palvelimen url-osoite. Esimerkkiohjelman mallit eivät sisällä funktioita.

```
define(['backbone'], function(Backbone){

    var Model = Backbone.Model.extend({
        // Annetaan mallin parametreille oletusarvot.
        defaults: {
            left    : 50,
            top     : 50,
            width   : 100,
            height  : 80,
            type    : undefined,
            id      : null,
            shapeId : '',
            text    : ''
        },
        localStorage: new
            Backbone.LocalStorage("shape-collection")
    });

    return Model;
});
```

5.3.1 Kuuntelijat

Kaikille mallin ominaisuuksille voidaan Backbone-kirjaston avulla määritellä kuuntelija (engl. listener), joka tarkkailee tiettyyn mallin ominaisuuteen kohdistuvia muutoksia. Toisin sanoen aina, kun kuunnellun ominaisuuden arvo muuttuu, kuuntelija havahtuu. Voimme itse päättää, miten reagoimme tapahtumaan kuuntelijan sisällä. [22, s. 8.]

Kuuntelijan asettaminen tapahtuu on-funktiolla, jolle voidaan kertoa tarkasti, minkä ominaisuuden muutoksia kuunnellaan. Kuuntelija voidaan sitoa myös kuuntelemaan kaikkia malliin kohdistuvia muutoksia kirjoittamalla on-funktion parametriksi vain sana ”change” ilman ominaisuuden nimeä.

```
EsimerkkiMalli = Backbone.Model.extend({
    defaults: {
        pituus: 5
    },

    initialize: function () {
        this.on('change:pituus', function (malli) {
            var uusi = malli.get('pituus');
```

```
            console.log('uusi pituus: ' + uusi);
        });
    }
});

// Laukaistaan kuuntelija muuttamalla pituutta
EsimerkkiMalli.set(pituus: 6);
```

Ominaisuuksien lisäksi voimme myös kuunnella esimerkiksi error-tapahtumaa, joka laukeaa silloin, kun mallin validate-funktio palauttaa virheen.

```
EsimerkkiMalli.on('error', function(malli, virhe) {
    console.log('virhe mallin validoinnissa');
});
```

5.3.2 Kommunikointi palvelimen kanssa

Malli kuvastaa tietoa palvelimella, joten mallin funktioiden toiminnan täytyy myös vastata palvelimen käskyjä. Malli käyttää REST-tyyppisiä käskyjä palvelimen kanssa kommunikointiin. REST (engl. Representational State Transfer) on arkkitehtuuri verkossa toimivien koneiden yhdistämiseen. REST:n avulla tietoa siirretään koneiden välillä käyttäen hyväksi HTTP-pyyntöjä. [24.] HTTP-pyynnöt ovat tapa siirtää tietoa asiakkaan eli selaimen ja palvelimen välillä. Selain lähettää pyynnön palvelimelle, joka käsittelee pyynnön ja palauttaa vastauksen. Selaimen ja palvelimen välille muodostetaan yhteys vain tämän tapahtuman ajaksi.

Jos halutaan käsitellä mallia tietokannassa, sillä täytyy olla osoite. Osoite määritellään urlRoot-parametrin avulla. Id-parametrin avulla itse malli voidaan paikantaa tietokannasta. Tietokanta voi olla relaatiotietokanta, kuten MySQL tai NoSQL-tietokanta kuten MongoDB.

5.3.3 Mallin tallennus ja haku palvelimelta

Uuden mallin luonti tietokantaan onnistuu, kun jätetään id määrittämättä. Kun kutsutaan mallin save-funktiota id:n ollessa null, Backbone lähettää palvelimelle POST-pyynnön. Pyyntö lähetetään mallissa määritettyyn urlRoot-osoitteeseen ja tietokantaan luodaan uusi malli. Save-funktion ensimmäiseksi parametriksi voidaan antaa JSON-muotoinen olio, jossa määritellään halutut ominaisuudet. Muuten malli käyttää oletusarvojaan.

Tietokannasta voidaan myös hakea vanhoja malleja. Tämä onnistuu antamalla luontivaiheessa haluttu id parametriksi uudelle mallille ja kutsumalla fetch-funktiota. Tällöin Backbone lähettää palvelimelle GET-pyynnön, joka palauttaa id:tä vastaavan mallin. [22, s. 9.]

Backbone käyttää eri HTTP-pyyntömenetelmiä mallien tallennukseen ja hakuun, koska menetelmien toiminnassa on olennaisia eroja. GET-menetelmää ei käytetä tietojen lähettämiseen, koska lähetettävät tiedot kulkevat suojaamattomana URL:n mukana, eikä GET-menetelmä salli URL:n olla kuin korkeintaan 256 merkkiä pitkä. Tästä syystä kaikki lähetettävä tieto ei välttämättä mahdu kulkemaan GET-pyynnössä. POST-menetelmässä vastaavaa merkkirajoitusta ei ole eivätkä tiedot kulje URL:n mukana. Tietojen hakemiseen palvelimelta GET-menetelmä sopii paremmin siksi, että se on nopeampi kuin POST. [25.]

```
var EsimerkkiMalli = Backbone.Model.extend({
    urlRoot: '/esimerkkiOsoite',
    defaults: {
        pituus: 5
    }
});

// Luodaan uusi malli ja tallennetaan se palvelimelle
var uusiMalli = new EsimerkkiMalli();
uusiMalli.save({ pituus: 10 }, {
    success: function () {
        console.log(uusiMalli.toJSON());
        // Konsoli tulostus näyttää seuraavalta:
        // { id: 0, pituus: 10 }
    }
});

// Haetaan vanha malli palvelimelta
var vanhaMalli = new EsimerkkiMalli({ id: 0 });
vanhaMalli.fetch();
```

ToJSON-funktio palauttaa kaikki mallin ominaisuudet yhdessä JSON-muotoisessa oliossa.

5.3.4 Mallin päivittäminen ja poistaminen palvelimelta

Äskeisessä esimerkissä luotiin uusi malli ilman id:tä ja haettiin vanha malli id:n kanssa. Vanhan mallin päivittäminen onnistuu määrittämällä tallennettavalle mallille id. Tällöin Backbone lähettää POST-pyynnön sijaan PUT-pyynnön ja päivittää vanhaa mallia.

Mallin poistaminen onnistuu kutsumalla Backbonen destroy-funktiota, joka lähettää palvelimme DELETE-pyynnön.

5.4 Kokoelma

Kokoelma (engl. collection) on joukko malleja. Yksi malli voi kuulua useaan eri kokoelmaan, mutta kokoelma sisältää aina vain yhdentyyppisiä malleja. Mallit voidaan antaa kokoelmalle parametreina luonnin yhteydessä tai lisätä jälkikäteen add-funktiolla. Mallien poistaminen onnistuu vastaavasti remove-funktion avulla. [22, s. 13.] Backbone sisältää monta kymmentä valmiiksi tehtyä funktiota kokoelmien käsittelyyn.

5.5 Reititin

Reititin (engl. router) ei ole perinteisen MVC-mallin osa, mutta Backbonesta sellainen löytyy. Reitittimen avulla voimme määrittää, mitä tapahtuu, kun menemme tiettyyn URL-osoitteeseen sivulla. Reititin käsittelee URL-osoitetta #-merkistä eteenpäin.

```
var Router = Backbone.Router.extend({
    routes: {
        'etusivu'         : 'etusivu',
        'haku/:parametri' : 'hae'
    },

    etusivu: function() {
        // Piirrä etusivu näkyviin
    },

    hae: function(parametri) {
        // Piirrä haku sivu ja hae parametrilla
    }
});
```

Edellä on esimerkki yksinkertaisesta reitittimestä. Routes-objektiin määritellään osoitteet, joihin reititin reagoi sekä funktiot, jotka suoritetaan kyseisiin osoitteisiin mentäessä.

Oletetaan, että sivuston URL-osoite on seuraava: http://www.sivunosoite.com. Jos käyttäjä menee osoitteeseen http://www.sivunosoite.com/#etusivu, niin reititin suorittaa etusivu-funktion. Vastaavasti, jos käyttäjä siirtyy osoitteeseen

http://www.sivunosoite.com/#haku/hakusana, niin hae-funktio tulee kutsutuksi parametrilla ”hakusana”. [22, s. 15.]

6 Modulaarinen JavaScript-ohjelmointi

Modulaarinen ohjelmointi tarkoittaa ohjelmakoodin jakamista erillisiin uudelleenkäytettäviin moduuleihin. Moduulien ansiosta ohjelmasta tulee helpommin hallittava ja testattava. Modulaarisessa ohjelmoinnissa moduulien täytyy pystyä myös kommunikoimaan keskenään, jotta ohjelma pystyy ylipäänsä toimimaan. [26.]

JavaScript-kehityksessä on ollut tapana kirjoittaa koko ohjelma vain muutamaan tiedostoon jolloin tiedostojen koot ovat olleet huomattavan isoja. Massiivisten tiedostojen ylläpitäminen on erittäin hankalaa, mutta niiden käyttöön on ollut myös syynsä. Vähäinen tiedostojen määrä tarkoittaa vähemmän HTTP-pyyntöjä, mikä on aina eduksi suorituskyvylle. [27.]

Myös tiedostojen välisten riippuvuuksien toteutus on perinteisesti ollut monimutkainen asia JavaScriptissä. Viittaukset riippuvuuksiin on toteutettu globaaleiden muuttujien avulla. Kehittäjien on myös täytynyt tietää eri koodin osien tarkka suoritusjärjestys, jotta niihin on voitu viitata.

6.1 AMD

AMD (engl. Asynchronous Module Definition) on rajapinta, jonka avulla JavaScript-moduuli ja sen riippuvuudet voidaan ladata yhtäaikaisesti eli asynkronisesti. AMD parantaa suorituskykyä yhtäaikaisen latauksen ansiosta, mutta helpottaa myös kehittäjien työtä, koska se mahdollistaa koodin kapseloinnin useisiin tiedostoihin. Kehittäjän ei myöskään tarvitse murehtia, ovatko kaikki moduulit latautuneet oikeassa järjestyksessä, koska AMD huolehtii myös latausjärjestyksen oikeellisuudesta. [28.]

6.2 RequireJS-kirjasto

RequireJS on yksi AMD-rajapintaa käyttävistä JavaScript-kirjastoista. RequireJS mahdollistaa JavaScript-moduulien keskinäisen kommunikaation sekä esimerkiksi sivupohjien lataamisen moduulin alussa. RequireJS toimii asiakas- ja palvelinpään ohjelmissa.

RequireJS on myös oiva kumppani Backbonelle, koska Backbone ei kerro lainkaan, kuinka koodi tulisi jakaa tiedostoihin. Tässä kohtaa Backbone-kirjasto eroaa selvästi monista MVC-kehyksistä, jotka määrittävät koko kehitysympäristön tarkasti. [22, s. 18.]

6.2.1 Tiedostorakenne

Kun ohjelmistoprojekti kasvaa yhtään suuremmaksi, on syytä käyttää hetki tiedostorakenteen järjestämiseen. Hyvästä tiedostorakenteesta löytää helposti hakemansa ja näin työteho parantuu. [22, s. 19.] Tässä on liitteenä olevan esimerkkiohjelman tiedostorakenne.

```
|- images            // Ohjelman kuvakansio
|   |- jqueryui      // Kansio jqueryUI:n kuville
|- css               // SASS:n generoimat CSS:t
|   |- style.css
|   |- jquery-ui.css
|   |- jqsimple.css
|- scss              // Itse tehdyt SASS-tiedostot
|   |- style.scss
|   |- _main.scss
|   |- _shape.scss
|- templates         // Sivupohjat
|   |- main.html
|   |- rectangle.html
|- js                // JavaSctipt tiedostot
|   |- libs
|   |   |- jquery.js
|   |   |- jquery-ui.js
|   |   |- jqsimple.js
|   |   |- underscore.js
|   |   |- backbone.js
|   |   |- backbone-localstorage.js
|   |   |- require.js
|   |- models       // Mallit
|   |   |- connection.js
|   |   |- shape.js
|   |- views        // Näkymät
|   |   |- connection.js
|   |   |- rectangle.js
|   |- collections  // Kokoelmat
```

```
|   |   |- connection.js
|   |   |- shape.js
|   |- bootstrap.js // Esilatausohjelma
|   |- app.js       // Ohjelman päämoduuli
|   |- text.js      // RequireJS text!-lisäosa
|- index-html       // Ohjelman pääsivu
```

6.2.2 Sivuston rakentaminen RequireJS:n avulla

RequireJS:n ansiosta ohjelman pääsivulle ei tarvita kuin yksi JavaScript-kutsu. Muutoin kaikki käytettävät JavaScript-tiedostot jouduttaisiin listaamaan pääsivulle ja vieläpä oikeassa järjestyksessä. Seuraava esimerkki havainnollistaa, miltä ohjelman pääsivu eli index.html näyttää RequireJS:n avulla. [22, s. 20.]

```
<html lang=”fi”>
    <head>
        <title>Pääsivu</title>
        <script
            data-main=”javascript/esilatausohjelma.js”
            src=”js/kirjastot/requirejs.js”>
        </script>
    </head>
    <body>
        <!-- Sivun sisältö -->
    </body>
</html>
```

6.2.3 Esilatausohjelma

RequireJS:lle määritetään aluksi esilatausohjelma (engl. bootstrap), joka kertoo mitä kirjastoriippuvuuksia ohjelmalla on ja mikä on ohjelman päämoduuli. Esilatausohjelman nimi määritellään data-main attribuutin avulla pääsivulla (katso edellisen luvun esimerkki). Esimerkin lopussa kutsutaan ohjelman päämoduulia, josta ohjelmamme suoritus jatkuu heti esilatausohjelman jälkeen. [22, s. 21.] Päämoduulin koodiesimerkki löytyy liitteen sivuilta kaksi ja kolme.

```
require.config({
    paths: {
        ’jquery’: ’js/kirjastot/jquery’,
        ’underscore’: ’js/kirjastot/underscore’,
        ’backbone’: ’js/kirjastot/backbone’
    }
});

define([
    ’jquery’,
```

```
    'underscore',
    'backbone',
    'moduuli'
], function($, _, Backbone, Moduuli){
        Moduuli.initialize(); // Ohjelman päämoduuli
    }
);
```

On myös syytä huomata, että ”.js” päätettä ei lisätä RequireJS:n yhteydessä polkuihin, vaan RequireJS täydentää ne automaattisesti. [22, s. 21.]

6.2.4 Esilatausohjelma ja AMD

RequireJS osaa normaalisti käsitellä kirjastoja vain, jos ne tukevat AMD:tä (luku 6.1). JQuery on AMD-kirjasto, mutta Backbone ja Underscore eivät ole. Tästä syystä edellisen kappaleen esimerkki palauttaa virheen.

Jotta esimerkki toimisi, tarvitaan esimerkkikoodiin pieni lisäys. RequireJS 2.0 esitteli uuden tavan käsitellä kirjastoja, jotka eivät tue AMD:tä. Kirjastot esitellään shim-ominaisuuden avulla. Shim-ominaisuus määrittää, mitkä ovat kirjaston riippuvuudet toisista kirjastoista ja mikä on kirjaston globaali nimi. Seuraavassa esimerkissä on tarvittava korjaus edelliseen esimerkkiin. [29.]

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

define([
    ... // Myös tämä osio on sama kuin edellä
);
```

6.2.5 Sivupohjien lataus RequireJS:n avulla

RequireJS:n text-lisäosa mahdollistaa myös muiden, kuin JavaScript tiedostojen lataamisen moduulin alussa. Tämä on erittäin kätevää, kun haluamme kirjoittaa sivupoh-

jan omaan tiedostoonsa ja liittää sen Backbone näkymään. [22, s. 24.] Laajempi esimerkki text-lisäosan käytöstä löytyy suorakulmionäkymän koodista liitteen sivulta seitsemän.

```
define([
    'jquery',
    'underscore',
    'backbone',
    'text!sivupohjat/sivupohja.html'
], function($, _, Backbone, Sivupohja) {
    var nakyma = Backbone.View.extend({
        el: $('body'),
        initialize: function() {
            this.render();
        },
        render: function() {
            this.$el.html(Sivupohja);
        }
    });
});
```

7 JavaScriptin erityispiirteet

Luvussa kaksi käsittelimme joitain JavaScriptin erityispiirteitä. Nuo asiat olivat helpohkoja asioita, jotka ovat tuttuja muista ohjelmointikielistä. Tässä luvussa käydään läpi hieman edistyneempiä JavaScriptin erityispiirteitä.

7.1 Sisäfunktion this-sana

Jos funktion sisään määritetään toinen funktio, ei sisäfunktion this-sana ole enää sama kuin ulkofunktion this-sana. Näin ollen ulkofunktiolle joudutaan aina määrittämään muuttuja, johon this-sana kopioidaan talteen ja tätä muuttujaa voidaan käyttää sisäfunktioissa viittaamaan ulkofunktion this-sanaan. Koodiesimerkki selventää asiaa. [3, s. 29.]

```
initialize: function () {
    this.collection = New NumberCollection();
},

shout: function (text) {
    alert(text);
},
```

```
render: function () {
    var self = this; // Kopioidaan this talteen

    // Kutsutaan kokoelmalle funktiota, jolle
    // määritellään sisäfunktio.
    this.collection.fetch({
        error: function() {
            // Sisäfunktiossa this sanan kutsuminen
            // suoraan ei toimi.
            this.shout('Tämä aiheuttaa virheen');
            self.shout('Tämä toimii');
        }
    });
}
```

7.2 Arguments-olio

JavaScriptissä voidaan määrittää funktio, jolle ei nimetä parametreja tai nimetään vain osa parametreista. Nimeämättömiin parametreihin päästään käsiksi arguments-olion kautta.

```
var yhdistaSanat = function () {
    var teksti;
    for (i=0; i < arguments.length; i++) {
        teksti += arguments[i];
    }
    return teksti;
};

// Kutsutaan funktiota
yhdistaSanat('oma', 'koti', 'talo');
// Palauttaa tekstin 'omakotitalo'
```

Esimerkissä funktion määrittelyssä ei ole yhtään parametria, mutta sitä kuitenkin kutsutaan kolmella parametrilla. Funktion sisällä olevassa silmukassa käytetään hyödyksi arguments-oliota, johon parametrit ovat tallentuneet. Silmukassa käydään läpi kaikki arguments-olion sisältämät parametrit ja lisätään ne muuttujaan, joka lopuksi palautetaan funktiosta.

7.3 Funktio parametrina

JavaScriptissä on mahdollista antaa funktiolle parametriksi toinen funktio. Menetelmä saattaa aluksi vaikuttaa hankalalta, mutta se selkeyttää monesti koodia etenkin tapahtumien ja kuuntelijoiden kanssa.

```
function laskeYhteen(luku1, luku2) {
    console.log(luku1 + luku2);
}

function jokuFunktio(funktio, arvo1, arvo2) {
    funktio(arvo1, arvo2);
}

jokuFunktio(laskeYhteen, 2, 5);
```

Edellisessä esimerkissä esitellään ensiksi kaksi funktiota. Ensimmäinen laskee kaksi lukua yhteen ja tulostaa ne. Toinen ottaa parametrikseen funktion sekä kaksi arvoa. Lopuksi kutsutaan toista funktiota ja annetaan sille parametriksi ensimmäinen funktio sekä kaksi mielivaltaista lukua. Tämä koodi tulostaisi selaimen konsoliin lukujen summan. [30.]

8 Suorituskykyyn vaikuttavia tekijöitä

Web-sivun latausnopeuteen ja suorituskykyyn voi vaikuttaa monella tavalla. Jo muutamalla perusparannuksella sivusta voi saada huomattavasti nopeamman ja samalla käytettävämmän.

Seuraavassa aliluvuissa perehdytään tiedostojen sijoitteluun eli siihen, minne esimerkiksi tyylitiedostot kannattaa sijoittaa web-sivulla. Tämän jälkeen puhutaan HTTP-pyyntöjen lukumäärän merkityksestä sivun nopeuteen sekä aliverkoista. Viimeisinä suorituskyky asioina käydään läpi muuttumattoman datan palvelimia sekä roskienkeruun tehostamista.

8.1 Tiedostojen sijoittelu

Ensimmäinen asia on sijoittaa CSS-tyylitiedostot aina sivun alkuun eli head-elementin sisään. Tällä tavoin varmistetaan, että selain pystyy aloittamaan sivun piirtämisen heti, eikä sen tarvitse odottaa tyylien latautumista.

JavaScript-tiedostot sen sijaan kannattaa sijoittaa mahdollisimman loppuun. Tällä varmistetaan se, ettei niiden suoritus estä muuta sivua piirtymästä näkyviin. [31, luku 1.1.]

8.2 Vähemmän HTTP-pyyntöjä ja enemmän pakattuja tiedostoja

Suorituskyvyn kannalta on myös merkittävää pitää HTTP-pyynnöt mahdollisimman vähissä. Jokainen kuva, tyylitiedosto, JavaScript-tiedosto tai mikä tahansa palvelimelta ladattava resurssi vaatii aina uuden HTTP-pyynnön.

HTTP-pyyntöjen määrää voidaan vähentää runsaasti esimerkiksi yhdistämällä kaikki sivulla käytettävät ikonit yhdeksi kuvatiedostoksi ja valitsemalla oikea ikoni CSS:n avulla. On myös suositeltavaa tehdä kaikista tyyli- ja JavaScript-tiedostoista yksi pakattu tiedosto. [31, luku 1.2.]

Suosittuja pakkaajia ovat mm. avoimen lähdekoodin YUI Compressor ja Douglas Crockfordin JSMin. Pakkaajat poistavat muun muassa kaikki turhat välilyönnit ja rivinvaihdot koodista sekä nimeävät muuttujat lyhyemmin. Lopullinen tiedosto on yleensä yksirivinen ja on hyvin vaikealukuinen ihmiselle. Koodi toimii kuitenkin samalla tavalla kuin alkuperäinenkin, vie vähemmän tilaa ja aiheuttaa ainoastaan yhden HTTP-pyynnön.

8.3 Document fragment -olio

DOM:iin kirjoittaminen ja sieltä lukeminen on hidasta. Varsinkin jos DOM:iin joudutaan kirjoittamaan toistuvasti, ohjelma saattaa hidastua huomattavasti. Jos sivulle pitää esimerkiksi lisätä lukuisia elementtejä, sitä ei kannata tehdä yksi elementti kerrallaan. Tässä on aluksi esimerkki huonosta toteutuksesta.

```
for(var i = 0; i < 100; i++) {
    $(body).append(<p>Tekstiä</p>); // Ei näin!
}
```

Järkevämpi tapa on kirjoittaa elementit aluksi ns. varastoon ja lisätä koko varasto kerralla DOM:iin. Näin DOM:iin kirjoitetaan vain kerran. Varastona seuraavassa esimerkissä toimii Document fragment -olio. [32.]

```
var frag = document.createDocumentFragment();

for(var i = 0; i < 100; i++) {
    frag.appendChild(<p>Tekstiä</p>);
}
```

```
$(body).append(frag);
```

8.4 Rinnakkaisuus

Selain sallii vain rajallisen määrän yhtäaikaisia eli rinnakkaisia HTTP-pyyntöjä yhtä verkko-osoitetta kohti. Tämä voi rajoittaa sivuston nopeutta. Ratkaisu on jakaa HTTP-pyynnöt useaan osoitteeseen eli ottaa käyttöön aliverkko-osoitteita. [31, luku 1.3.]

Haittapuolena aliverkko-osoitteiden käytössä on DNS-tarkistus, joka vie yleensä noin 20-120 millisekuntia [31, luku 2]. DNS on nimipalvelujärjestelmä, joka huolehtii verkko-tunnusten kääntämisestä IP-osoitteiksi [33, s. 20]. Tästä syystä aliverkko-osoitteita kannattaa käyttää vain, jos niistä saatava nopeusetu on suurempi kuin DNS-tarkistukseen kulunut aika.

8.5 CDN

CDN on lyhenne sanoista Content Delivery Network. CDN on tarkoitettu muuttumattoman sisällön varastoksi ja sen etu tavallisiin palvelimiin on lyhyemmät latausajat. Lyhyemmät latausajat perustuvat fyysisesti lähelle sijoitettuihin palvelimiin, jotka sijaitsevat parhaassa tapauksessa samassa maassa, kuin itse käyttäjä. [20, s. 138.] Yahoo!:n mittausten mukaan käyttäjän odotusaika vähenee CDN:n avulla ainakin 20 prosenttia [34].

CDN:stä on hyötyä kaikille verkkosivuille, joilla on käyttäjiä eri puolilla maailmaa. Suurimman hyödyn saavat ne verkkosivut, joilla on paljon videoita tai muita ladattavia tiedostoja. Suurien tiedostojen latausajat ovat yleensä huomattavasti pienemmät, kun tiedostopalvelin sijaitsee lähellä käyttäjää. [35.]

8.6 Roskienkeruu

Yksi tapa optimoida JavaScript-ohjelmaa nopeammaksi on auttaa roskienkeruuta toimimaan paremmin. Roskienkeruun tehtävä on poistaa kaikki oliot ja muuttujat, joita ohjelma ei enää tarvitse. Ne oliot ja muuttujat, joihin vielä viitataan, välttyvät roskienkeruulta. JavaScriptissä roskienkeruuta ei voi itse käskeä päälle, vaan se toimii aina automaattisesti parhaaksi katsomanaan aikana.

Viittauksia tarvitsee harvoin alkaa itse poistamaan. Delete-funktion kutsumisesta oliolle on enemmän haittaa kuin hyötyä, koska olioiden rakenteiden muutokset ovat hitaita suorittaa. Myös olion asettamista null-arvoiseksi kannattaa välttää. Mikäli olioon viitataan jostain muualta, sen asettaminen yhdessä paikassa null-arvoiseksi ei tee siitä poistettavaa. [36.]

Hyvin sijoitellut muuttujat auttavat roskienkeruuta toimimaan paremmin. Kannattaa siis suosia paikallisia muuttujia, eli sijoittaa muuttujat funktion sisään, eikä sen ulkopuolelle. Paikalliset muuttujat poistetaan heti, kun niitä ympäröivään funktioon ei enää viitata. Globaalit muuttujat pysyvät niin kauan elossa, kuin sivu on auki käyttäjän selaimessa. Roskienkeruu poimii globaalit muuttujat vasta, kun sivu päivitetään, suljetaan tai siirrytään toiselle sivulle. [36.]

```
function esimerkkiFunktio() {
    var muuttuja = new olio();
    muuttuja.teeJotain();
    return muuttuja;
}
```

Edellisessä esimerkkikoodissa olio on valmis roskienkeruun poistettavaksi heti, kun funktio on suoritettu, jos mikään ohjelmanosa ei ota talteen return-lauseen palauttavaa paluuarvoa.

```
function esimerkkiFunktio() {
    var muuttuja = new olio();
    muuttuja.teeJotain();
    return muuttuja;
}

// Kutsutaan esimerkkifunktiota jostain
var toinenMuuttuja = esimerkkiFunktio();
```

Tässä tapauksessa olio säilyy, koska siihen on viittaus. Olio vapautuu roskienkeruulle vasta, kun ”toinenMuuttuja” saa uuden arvon. [36.]

8.7 Kuuntelijoiden pysäyttäminen

Tapahtuman kuuntelijat on syytä pysäyttää aina, kun niitä ei enää tarvita jotta ne eivät turhaan varaa selaimen muistia. Etenkin silloin, jos kuunneltava DOM-elementti on jo

poistettu DOM-puusta. jQuery- ja Backbone -kirjastoissa on off-funktio, jolla tapahtuman kuuntelija voidaan pysäyttää. [36.]

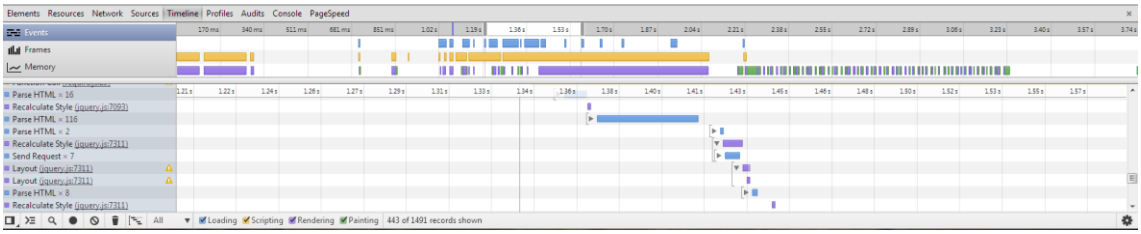
```
// Kuunnellaan klikkaus tapahtumaa
$('.elementti').on('click', function(){
    console.log('Klik!');
});

// Pysäytetään kuuntelija
$('.elementti').off('click');
```

8.8 Työkaluja suorituskyvyn mittaamiseen

Suorituskykyyn vaikuttavia tekijöitä on hyvä tunnistaa jo kehitysvaiheessa, mutta välillä joudutaan etsimään ohjelmaa hidastavia kohtia jälkikäteen. Monet selaimet tarjoavat erilaisia suorituskyvyn analysointityökaluja vähintäänkin lisäosina. Chrome-selaimen mukana tulee työkalupaketti nimeltään Developer tools, joka tarjoaa kaksi hyvää työkalua suorituskyvyn mittaamiseen.

Timeline-työkalu (kuva 3) on näistä ensimmäinen, ja sen avulla voidaan tarkastella, mihin kaikkeen aikaa kuluu sovelluksen suorittaessa tehtäviään. Timeline-työkalulla voidaan siis nauhoittaa käyttäjän tekemiä toimia ohjelmassa ja analysoida niihin kulunutta aikaa sekä muistia.



Kuva 3. Chrome-selaimen timeline-työkalu

Toinen suorituskyky työkalu on nimeltään Audits. Audits-työkalu kertoo kehittäjälle, mitä ohjelmalle tulisi tehdä, jotta se latautuisi nopeammin. Audits-työkalu tunnistaa huomattavan määrän suoristuskykyyn negatiivisesti vaikuttavia tekijöitä ja osaa myös kertoa korjausehdotuksia.

9 Palvelinpään JavaScript

JavaScriptiä käytettiin pitkään ainoastaan asiakaspään ohjelmissa, mutta viime vuosina myös palvelinpään ratkaisut ovat yleistyneet. Tässä luvussa käydään pintapuolisesti läpi palvelimen pystyttämistä JavaScriptin avulla.

9.1 Node.js

Node.js on kehitysympäristö, jossa on mahdollista suorittaa JavaScript-koodia selaimen ulkopuolella. Node.js on rakennettu Googlen V8 JavaScript moottorin päälle. V8 on sama moottori, jota käytetään myös Googlen omassa Chrome-selaimessa. [37, s. 3.]

Node.js on paitsi kehitysympäristö, myös palvelin. Node.js:n toimintaperiaate on kuitenkin varsin erilainen kuin esimerkiksi PHP:n tai Javan. PHP:ssä jokaista HTTP-pyyntöä kohden luodaan aina uusi prosessi eli säie. Node.js kuitenkin suoritetaan yhdessä säikeessä. Tämän takia Node.js ei voi odottaa palvelimen vastausta yhteen pyyntöön, koska tällöin kaikki muut pyynnöt joutuisivat odottamaan ja palvelusta tulisi auttamattoman hidas. Node.js toimii asynkronisesti ja käsittelee useampaa pyyntöä yhtäaikaisesti. Kun jokin pyyntö on valmis se lähettää vastakutsun, joita Node.js kuuntelee tapahtumasilmukassa jatkuvasti. Kutsut eivät siis estä toistensa toimintaa lainkaan. [30.]

9.1.1 Ensimmäinen palvelin

Seuraava koodiesimerkki havainnollistaa palvelimen luonnin helppoutta Node.js:n avulla. Esimerkin ensimmäisellä rivillä luodaan muuttuja, johon tallennetaan Node.js:n HTTP-kirjasto. Tämän jälkeen itse palvelin luodaan kirjaston createServer-funktion avulla. Lopuksi palvelin asetetaan kuuntelemaan porttia 8888. Näin yksinkertainen palvelin kuuntelee ainoastaan porttia, eikä tee mitään muuta. [30.]

```
var http = require('http');
var palvelin = http.createServer();
palvelin.listen(8888);
```

9.1.2 Palvelin, joka vastaa pyyntöihin

Muokataan edellistä esimerkkiä hieman, jotta palvelin vastaa pyyntöihin. Näin pystymme varmistamaan, että palvelin toimii oikein.

```
var http = require('http');
var palvelin = http.createServer(
    function(pyynto, vastaus) {
        vastaus.writeHead(
            200,
            {'Content-type': 'text/plain'}
        );
        vastaus.write('Palvelin toimii!');
        vastaus.end();
    }
).listen(8888);
```

CreateServer-funktiolle annetaan nyt parametriksi anonyymi funktio. Funktio saa parametrikseen pyynnön ja vastauksen. [37, s. 10.]

10 Yhteenveto

Tämän insinöörityön ensisijainen tavoite oli selvittää, miten hyvin MVC-arkkitehtuuri sopii web-kehitykseen. Työssä oli myös tarkoitus perehtyä uusimpiin JavaScript-tekniikoihin, modulaariseen ohjelmointiin sekä verkkosovellusten suorituskyvyn parantamiseen.

Aloin hahmotella tätä insinöörityötä syksyllä 2012 ja silloin hyväksytin myös aiheeni. Olin tuolloin töissä Bublaa Oy -nimisessä yrityksessä, joka oli kasvava verkkopalveluihin keskittyvä IT-yritys. Kyseisessä yrityksessä käytettiin paljon uusia web-tekniikoita, kuten esimerkiksi moderneja JavaScript-kirjastoja.

Suunnittelimme esimieheni Aki Lehtisen kanssa, että olisi hyvä, jos tekisin heille insinöörityön yrityksessä käytettävistä sovelluskehitystekniikoista. Insinöörityön tulisi palvella myös uusia työntekijöitä ns. aloitusoppaana, jonka takia tämä työ sisältääkin huomattavan määrän selkeitä esimerkkejä.

Kirjoitusprosessin aikana tuli selväksi, että JavaScript sopii erinomaisesti web-kehitykseen jo pelkän selaintukensa ansiosta. Perinteinen MVC-arkkitehtuuri sen sijaan harvoin soveltuu sellaisenaan web-maailmaan, jonka tarpeet ovat usein hieman erilai-

set, kuin esimerkiksi työpöytäohjelmissa. Siksi monet JavaScriptin MVC-kirjastot ovatkin soveltaneet MVC-arkkitehtuuria tai pitävät sitä vain suuntaa antavan ohjeena.

Yksi suurimmista ongelmista MVC-arkkitehtuurin käytössä web-sovelluksissa on MVC-kirjastojen suuri määrä. On hyvin vaikea päättää, millä tekniikoilla uutta projektia kannattaisi tehdä ja jatkuuko valittujen kirjastojen kehitys varmasti myös tulevaisuudessa.

Ylipäänsä on ongelmallista, että yhteen JavaScript ohjelmaan saatetaan tarvita lukuisia kirjastoja hoitamaan eri tehtäviä: yksi DOM:n manipulointiin, toinen MVC-arkkitehtuuriin ja kolmas vaikkapa moduulien hallintaan. Ja näiden kirjastojen yhteistoiminta ei aina ole itsestäänselvyys. Uskon kuitenkin, että tulevaisuudessa tietyt kirjastot valikoituvat muita suosituimmiksi ja niiden ympärille saadaan rakennettua parempia kehitysympäristöjä, jotta ohjelmoijat voivat keskittyä enemmän itse koodin kirjoittamiseen kuin oikeiden kirjastojen valintaan ja kehitysympäristöjen rakentamiseen.

Työssä esiteltiin erityyppisiä JavaScript-kirjastoja monipuolisesti. Toki olisi ollut mahdollista tehdä enemmän vertailua eri vaihtoehtojen kesken, mutta ajattelin, että on parempi esitellä muutama kirjasto kattavasti kuin monta pintapuolisesti.

Insinöörityön suorituskykyosio voisi olla laajempikin. Itse asiassa web-sovellusten suorituskyvyn parantamisesta saisi aiheen kokonaiseen insinöörityöhön. Suorituskyky on kuitenkin niin tärkeä asia, etten halunnut sivuuttaa sitä kokonaan. Keräsin tärkeimmät suorituskykyyn vaikuttavat tekijät tähän työhön ja niillä pääsee jo varmasti hyvin alkuun.

Yksi iso ongelma itse insinöörityön kirjoitusprosessissa oli se, että jouduin vaihtamaan työpaikkaa kesken kaiken. Työpaikan vaihdon jälkeen myös työn ohjaus entisen työnantajani puolelta katkesi käytännössä kokonaan. Uudessa työpaikassa käyttämäni tekniikat olivat jokseenkin samat kuin Bublaa Oy:ssä, mutta eroavaisuuksiakin löytyi. Työpaikanvaihdos hidasti työn valmistumista, mutta uusi työpaikka vaikutti insinöörityöhöni kuitenkin enemmän positiivisella tavalla. Sain paljon uusia ideoita ja eri näkökulmia asioihin, joista olin jo kirjoittanut.

Lähteet

1 Young, Alex. 2010. History of JavaScript: Part 1. Verkkodokumentti. <http://dailyjs.com/2010/05/24/history-of-javascript-1/>. 24.5.2010. Luettu 25.12.2012.

2 Object copy – Shallow copy. Verkkodokumentti. <http://en.wikipedia.org/wiki/Object_copy#Shallow_copy> 4.11.2012. Luettu 16.1.2013

3 Crockford, Douglas. 2008. JavaScript: The Good parts. Sebastopol, California: O'Reilly Media Inc.

4 Sarkka, Simo. 1998. JavaScriptin Objektit. Verkkodokumentti. <http://users.tkk.fi/u/ssarkka/javascript/obj.html> 1.7.1998. Luettu 11.5.2013

5 Mozilla Development Network. 2012. Differential inheritance in JavaScript. Verkkodokumentti. <https://developer.mozilla.org/en/docs/Differential_inheritance_in_JavaScript> 1.10.2011. Luettu 16.1.2013

6 Osmani, Addy. 2012. Journey Through The JavaScript MVC Jungle. Verkkodokumentti. <http://coding.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/> 27.7.2012. Luettu 2.1.2013

7 Burbeck, Steve. 1992. Applications Programming in Smalltalk-80(TM) - How to use Model-View-Controller (MVC). Verkkodokumentti. <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html> Luettu 2.1.2013

8 Anderson, Steve. 2012. Rich JavaScript Applications – the Seven Frameworks. Verkkodokumentti. <http://blog.stevensanderson.com/2012/08/01/rich-javascript-applications-the-seven-frameworks-throne-of-js-2012/>. 1.8.2012. Luettu 26.12.2012

9 Osmani Addy. 2012. Developing Backbone.js Applications. Sebastopol, California: O'Reilly Media Inc.

10 Le Hégaret, Philippe. 2005. Document Object Model (DOM). Verkkodokumentti. <http://www.w3.org/DOM/> 19.1.2005. Luettu 2.1.2013

11 Koch, Peter-Paul. 2007. W3C DOM - Introduction. Verkkodokumentti. <http://www.quirksmode.org/dom/intro.html> Luettu 22.2.2013

12 Kantor, Ilya. 2011. Searching elements in DOM. Verkkodokumentti. <http://javascript.info/tutorial/searching-elements-dom> Luettu 23.3.2013

13 Garrett, Jesse James. 2005. Ajax: A New Approach to Web Applications. Verkkodokumentti. <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications> 18.2.2005. Luettu 5.8.2013

14 Narayan, Sheo. 2011. What is jQuery and How to Start using jQuery?. Verkkodokumentti. <http://www.codeproject.com/Articles/157446/What-is-jQuery-and-How-to-Start-using-jQuery> 22.2.2011. Luettu 5.8.2013

15 Murphey, Rebecca. 2012. jQuery Fundamentals - Selecting Elements. E-kirja. <http://jqfundamentals.com/legacy#chapter-3>. Luettu 9.1.2013

16 Murphey, Rebecca. 2012. jQuery Fundamentals – Working with Selections. E-kirja. <http://jqfundamentals.com/legacy#chapter-3>. Luettu 9.1.2013

17 jQuery Community Experts. 2010. jQuery Cookbook. Sebastopol, California: O'Reilly Media Inc.

18 Mozilla Development Network. 2012. window.onload. Verkkodokumentti. <https://developer.mozilla.org/en-US/docs/DOM/window.onload> 10.11.2012. Luettu 10.1.2013

19 Franklin, Jack. 2012. Essential JavaScript – the top five MVC frameworks. Verkkodokumentti. <http://www.netmagazine.com/features/essential-javascript-top-five-mvc-frameworks> 23.11.2012. Luettu 2.1.2013

20 MacCaw, Alex. 2011. JavaScript Web Applications. Sebastopol, California: O'Reilly Media Inc.

21 Davis, Thomas. 2011. Backbone.js Tutorial – by noob for noobs. Verkkodokumentti. <http://thomasdavis.github.com/2011/02/01/backbone-introduction.html> 1.2.2011. Luettu 24.3.2013

22 Davis, Thomas. 2012. Backbone Tutorials. E-kirja. <https://leanpub.com/backbonetutorials>. Version julkaisupäivä 23.10.2012.

23 Ashkenas, Jeremy. 2012. Backbone.Events. Verkkodokumentti. <http://backbonejs.org/#Events> Luettu 28.1.2013

24 Elkstein, M. 2008. Learn REST: A Tutorial. Verkkodokumentti. <http://rest.elkstein.org/2008/02/what-is-rest.html> 1.2.2008. Luettu 11.5.2013

25 Jaakkola, Matti. 2001. HTTP-pyyntömenetelmät. Verkkodokumentti. <http://edu.phkk.fi/Opiskelu/secure/Internet-ohjelmointi/www-ohjelmointi_CGI/http_pyynnot.htm> 8.5.2001. Luettu 24.3.2013

26 Silander, Simo. 1999. Modulaarinen ohjelmointi. Verkkodokumentti. <http://cs.stadia.fi/~silas/ohjelmointi/c_opas-Modulaar.html> Luettu 27.3.2013

27 Hardy, Aaron. 2012. Dependency management with RequireJS. Verkkodokumentti. <http://www.adobe.com/devnet/html5/articles/javascript-architecture-requirejs-dependency-management.html> 29.3.2012. Luettu 30.1.2013

28 Osmani, Addy. 2011. Writing Modular JavaScript With AMD, CommonJS & ES Harmony. Verkkodokumentti. <http://addyosmani.com/writing-modular-js/> Luettu 27.3.2013

29 Barlow, Mike. 2013. Re-Learning Backbone.js – Require.js (AMD and Shim). Verkkodokumentti. <http://bardevblog.wordpress.com/2013/01/05/re-learning-backbone-js-require-js-and-amd/> 5.1.2013. Luettu 2.4.2013

30 Kiessling, Manuel. 2011. The Node Beginner Book. Verkkodokumentti. <http://www.nodebeginner.org/>. Luettu 1.4.2013

31 Roberts, Harry. 2012. Front-end performance for web designers and front-end developers. Verkkodokumentti. <http://csswizardry.com/2013/01/front-end-performance-for-web-designers-and-front-end-developers/> 20.1.2013. Luettu 31.1.2013

32 Walsh, David. 2012. JavaScript DocumentFragment. Verkkodokumentti. <http://davidwalsh.name/documentfragment> 12.11.2012. Luettu 5.8.2013

33 J. Botha, C. Bothma ja Pieter Geldenhuys. 2008. Managing E-Commerce in Business: Second edition. Cape Town, South Africa: Juta & Company Ltd

34 Yahoo! Performance Team. Best Practices for Speeding Up Your Web Site. Verkkodokumentti. <http://developer.yahoo.com/performance/rules.html#cdn> Luettu 1.2.2013

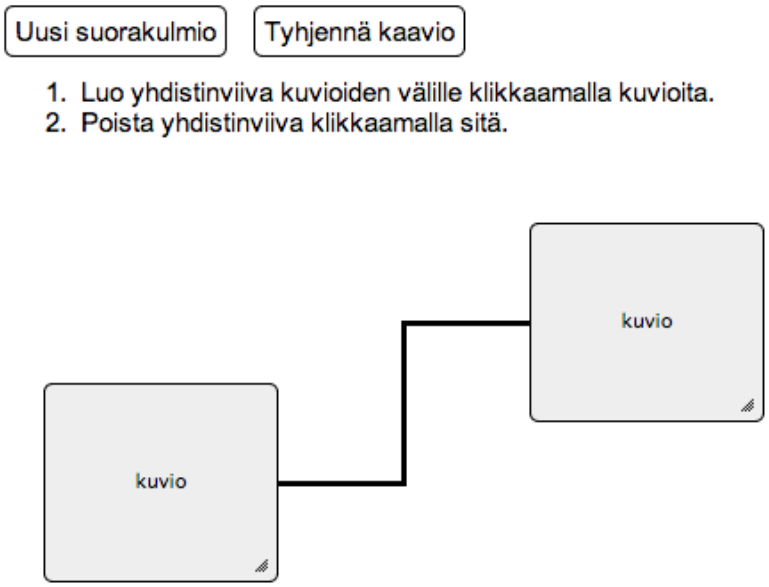
35 Sevcik, Peter ja Wetzel, Rebecca. 2013. Who needs a CDN?. Verkkodokumentti. <http://www.networkworld.com/community/blog/who-needs-cdn> 25.2.2013. Luettu 25.3.2013

36 Osmani, Addy. 2012. Writing Fast, Memory-Efficient JavaScript. Verkkodokumentti. <http://coding.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/> 5.11.2012. Luettu 1.2.2013

37 Hughes-Croucher, Tom ja Wilson, Mike. 2012. Node: Up and running. Sebastopol, California: O'Reilly Media Inc.

Esimerkkiohjelma

Esimerkkiohjelmassa yhdistetään työssä läpikäytyjä tekniikoita. Ohjelma käyttää modulaarista ohjelmointia ja RequireJS:ää. MVC-kirjastona toimii Backbone. Tämän lisäksi käytössä ovat jQuery-, jQueryUI-, Underscore- ja BackboneLocalStorage-kirjasto.



Ohjelma itsessään on yksinkertainen piirto-ohjelma, jossa on mahdollista piirtää suorakulmioita ja yhdistää niitä viivoilla. Suorakulmiot ja yhdistinviivat tallennetaan selaimen paikalliseen varastoon, joten ne säilyvät ruudulla vaikka selainikkuna päivitetään.

Esimerkkiohjelman koodit

On syytä huomata, että esimerkkiohjelmaa täytyy ajaa palvelinympäristössä. Chrome-selaimella on myös mahdollista suorittaa ohjelma lokaalisti käynnistämällä Chrome parametrilla --allow-file-access-from-files.

index.html

```
<!doctype html>
<html>
    <!--
        Tämä on se html-tiedosto, josta kaikki alkaa.
        Koska käytämme RequireJS:ää, niin tämä
        tiedosto on tosi yksinkertainen, eikä täällä
        ole kuin yksi scripti-tagi - requireJS itse.
    -->
```

```
    <head>
        <!-- Kerrotaan, mitä merkistökoodausta sivu käyttää. -->
        <meta charset='utf-8'>
        <title>Inssityö demo-ohjelma</title>
        <!--
            Määritellään tyylitiedostojen sijainnit.
            style.css on SASSilla generoimamme
            tyylitiedosto ja kaksi jälkimmäistä
            ovat kirjastojen omia tyylejä.
        -->
        <link rel="stylesheet" type="text/css" href="css/style.css">
        <link rel="stylesheet" type="text/css"
              href="css/jquery-ui.css"/>
        <link rel="stylesheet" type="text/css"
              href="css/jqsimple.css"/>

        <!--
            Data-main määrittää esilatausohjelmamme
            nimen, jota kutsutaan seuraavaksi.
        -->
        <script data-main="js/bootstrap" src="js/libs/require.js">
        </script>
    </head>

    <body>
        <!--
            Body on tyhjä, koska generoimme kaiken sisällän
            JavaScriptillä.
        -->
    </body>
</html>
```

js/bootstrap.js

```
// RequireJS esilataustiedosto, jossa määrittelemme
// kirjastoriippuvuuksien sijainnit sekä pääohjelman,
// josta ohjelman suoritus jatkuu.
requirejs.config({
    // Kirjastojen polut ILMAN .js päätettä.
    paths: {
        'jquery'       : 'libs/jquery',
        'jqueryUI'     : 'libs/jquery-ui',
        'underscore'   : 'libs/underscore',
        'backbone'     : 'libs/backbone',
        'localstorage' : 'libs/backbone-localstorage',
        'jqsimple'     : 'libs/jqsimple'
    },
    // RequireJS tukee normaalisti vain sellaisia kirjastoja,
    // jotka ovat AMD-yhteensopivia. Kirjastoille, jotka eivät
    // AMD:tä on olemassa shim-funktio. Esimerkiksi Backbone
    // ja Underscore eivät ole AMD-yhteensopivia, joten niille
    // tulee määritellä shimissä erikseen riippuvuudet ja nimi,
    // jolla kirjasto tunnetaan. Näin RequireJS osaa käyttää
    // kirjastoja oikein.

    // Toinen vaihtoehto shim-funktion käytölle olisi etsiä
    // kirjastoista erikoisversiot, joihin AMD-yhteensopivuus
    // on koodattu.
```

```
    shim: {
        'underscore': {
            exports: '_'
        },
        'backbone': {
            deps: ['jquery', 'underscore'],
            exports: 'Backbone'
        }
    }
});

define([
    'jquery',
    'jqueryUI',
    'underscore',
    'backbone',
    'localstorage',
    'jqsimple',
    'app'
], function(
    // Näillä nimillä voimme viitata kirjastoihin mistä vaan
    // tiedostosta ilman, että meidän tarvitsee edes erikseen
    // esitellä tai vaatia kirjastoja tiedoston alussa.
    $,
    jqueryUi,
    _,
    Backbone,
    localstorage,
    jqsimple,
    App
    ) {
        // Esilataustiedoston lopuksi kutsumme pääohjelmaa,
        // joka on tässä tapauksessa app.js-tiedosto.
        App.initialize();
    }
);
```

js/app.js

```
// Sovelluksen pääohjelma, jossa luomme päänäkymän.
// Tätä tiedostoa kutsutaan esilatausohjelmasta.
define([
    'views/main'
], function(
    MainView
    ){

    var initialize = function() {
        var mainView = new MainView();
    };

    return {
        initialize: initialize
    };
});
```

js/views/main.js

```
// Define on RequireJS:n funktio, joka kertoo, mitä tiedostoja
// tämä näkymä tarvitsee. Seuraava funktio saa parametrinaan
// nimet, joilla näihin riippuvuustiedostoihin viitataan.
define([
    // Kirjastoriippuvuuksia ei yleensä tarvitse esitellä
    // enää näkymissä, jos ne on jo esitelty esilatausohjelmassa.
    // Ohjelma kuitenkin herjaa ajoittain, että Backbonea ei löydy.
    // Tämä johtuu ilmeisimmin siitä, että Backbone ei tue AMD:tä,
    // eikä siksi ole vielä ehtinyt latautua näkymän sitä vaatiessa.
    // Siispä vaadimme backbonen näkymän alussa, jotta se varmasti
    // ehtii mukaan.
    'backbone',
    // Tässä käytetään requireJS text-pluginia,
    // joka mahdollistaa muiden, kuin JS-tiedostojen lataamisen
    // RequireJS:n avulla.
    'text!../../templates/main.html', // Näkymän sivupohja
    'views/rectangle',                // Suorakulmio näkymä
    'collections/shape',              // Kokoelma kuvioille
    'collections/connection'          // Kokoelma yhteyksille
], function(
    Backbone,
    mainTemplate,
    RectangleView,
    ShapeCollection,
    ConnectionCollection
){
    var View = Backbone.View.extend({
        // el määrittää sen DOM-elementin, johon tämä näkymä
        // piirretään.
        el: $('body'),

        // events-olioon voimme listata näkymään liittyviä tapahtumia.
        // Vasen puoli kertoo tapahtuman tyypin sekä DOM-elementin,
        // johon tapahtuma liittyy. Oikea puoli taas kertoo, mitä
        // funktiota kutsutaan tapahtuman lauetessa.
        events: {
            'click .new-rect'        : 'newRectangle',
            'click .delete-diagram'  : 'deleteDiagram',
            'click .shape'           : 'connect',
            'click .jqSimpleConnect' : 'removeConnection'
        },

        initialize: function() {
            // Underscoren bindAll-funktio varmistaa, että
            // konteksti eli this pysyy samana kaikilla tämän näkymän
            // funktioilla. Konteksti saattaisi muuten vaihtua,
            // jos jotain funktiota kutsuttaisiin jostain
            // muusta näkymästä.
            _.bindAll(
                this,
                'initialize',
                'render',
                'connect',
                'newRectangle',
                'deleteDiagram',
                'removeConnection'
            );

            // Tallennetaan näkymän konteksti this muuttujaan nimeltä
            // self. Tämä tehdään siksi, että myöhemmin pystymme
```

```
// viittaamaan this sanaan sellaisissa paikoissa,
// joissa konteksti on vaihtunut. Käytännössä siis
// sisäfunktiot käyttävät self muuttujaa viitatessaan
// näkymän this-sanaan.
var self = this;

// Apumuuttuja mallin id:tä varten.
this.counter = 0;

// Renderiä on kutsuttava ennen kuin kokoelma täytetään.
// Muuten $('#main')-elementti ei ole vielä DOMissa,
// eikä siihen silloin voi viitata. Render-funktio siis
// piirtää näkymään liittyvän sivupohjan (html-tiedoston)
// määrittämäämme paikkaan, eli tässä tapauksessa sivun
// bodyyn.
this.render();

// Alustetaan kuviokokoelma.
this.shapeCollection = new ShapeCollection();

// Määritetään kuuntelija add-tapahtumalle.
// Aina, kun kuvio kokoelmaan lisätään uusi kuvio,
// suoritetaan toisena parametrina annettu funktio.
this.shapeCollection.on('add', function(model) {
    // Luodaan uusi suorakulmio näkymä.
    // Nyt olemme sisäfunktiossa, joten joudumme
    // käyttämään self-sanaa viitataksemme laskuriin.
    var rectangleView = new RectangleView({
        // Annetaan id:ksi uniikki merkkijono.
        id : 'shape' + self.counter,
        // Annetaan näkymälle parametrina myös siihen
        // liittyvä malli.
        model : model
    });

    // Lisätään juuri luodun suorakulmionäkymän html-
    // sisältö tämän näkymän elementtiin, jonka id on
    // "main".
    self.$('#main').append(rectangleView.el);

    // Kasvatetaan laskurimuuttujaa, jotta kuvioiden id
    // pysyy uniikkina.
    self.counter++;
});

// Fetch-funktio hakee kokoelmaan liittyvät mallin
// palvelimelta. Tässä tapauksessa siis localStoragesta,
// jonne olemme ne tallentaneet. Fetch-laukaisee add-
// tapahtuman aina, kun se hakee uuden mallin, joten
// edellä määritetty kuuntelija laukeaa aina uuden mallin
// tullessa.
this.shapeCollection.fetch();


// Alustetaan yhteyskokoelma.
this.connectionCollection = new ConnectionCollection();

// Yhteyksien luomiseen käytämme valmista jq-simple-
// connect -kirjastoa (https://code.google.com/p/jq-
// simple-connect/). Kirjaston tehtävä on laskea, miten
```

```
        // yhteydet piirretään. Tästä syystä emme tarvitse omaa
        // näkymää yhteyksille vaan riittää, että tallennamme
        // yhteysmallin palvelimelle.

        // Kuunnellaan taas mallien lisäilyä.
        this.connectionCollection.on('add', function(model) {

            // Kuunnellaa remove-tapahtumaa, jonka kokoelma
            // laukaisee. Malli poistaa itsensä tapahtuman
            // lauetessa.
            model.on('remove', function() {
                this.destroy();
            });

            // Asetetaan mallille id:t.
            model.set({
                sourceId: self.sourceId || model.get('sourceId'),
                targetId: self.targetId || model.get('targetId')
            });

            // Tallennetaan malli palvelimelle eli tässä
            // tapauksessa localStorageen.
            model.save();

            // Esitellään muuttuja, johon yhteyden id
            // tallennetaan.
            var connectionId;

            // Piirretään yhdistinviiva käyttäen jq-simple-connect
            // -kirjaston connect-funktiota. Funktion parametrit:
            // lähde, kohde, viivan tyyli.
            if(self.sourceId === undefined) {
                connectionId = jqSimpleConnect.connect(
                    '#' + model.get('sourceId'),
                    '#' + model.get('targetId'),
                    { radius: 3, color: '#000' }
                );
            }
            else {
                connectionId = jqSimpleConnect.connect(
                    '#' + self.sourceId,
                    '#' + self.targetId,
                    { radius: 3, color: '#000' }
                );
            }
            // Alustetaan sourceId ja targetId muuttujat, jotta
            // niitä voidaan käyttää taas uudestaan toisen mallin
            // kanssa.
            self.sourceId = undefined;
            self.targetId = undefined;

            model.save({ connectionId : connectionId });
        });

        // Haetaan mallit palvelimelta.
        this.connectionCollection.fetch();
    },

    render: function() {
        // Kirjoitetaan sivupohja tiedoston sisältö määrittämäälle
```

```
    // el muuttujaan eli tässä tapauksessa sivun bodyyn.
    // Html-funktio tyhjentää DOM-elememtin, kun taas append
    // lisää olemassa olevien elementtien perään uutta
    // tavaraa.
    this.$el.html(mainTemplate);

    return this;
},

connect: function(e) {
    // Uuden yhteyden voi luoda klikkaamalle ensiksi
    // haluamaansa lähde kuviota ja sitten kohde kuviota.

    // e-parametrista saamme kaivettua tiedon siitä,
    // mitä kuviota klikattiin. Tässä on toteutettu logiikka
    // sille, että yhteys luodaan vasta, kun kahta kuviota
    // on klikattu.
    if(this.sourceId === undefined) {
        this.sourceId = e.currentTarget.id;
    }
    else {
        this.targetId = e.currentTarget.id;
        this.connectionCollection.create();
    }
},

newRectangle: function() {
    // Luodaan uusi kuviomalli kuviokokoelmaan.
    // Create-funktio laukaisee add-tapahtuman,
    // jolle olemme määrittäneet kuuntelijan.
    // Create-funktio tallentaa myös juuri luodun
    // mallin palvelimelle.
    this.shapeCollection.create();
},

deleteDiagram: function() {
    // Tyhjennä local storage ja päivitä sivu
    // tämänhetkiseen sijaintiin.
    localStorage.clear();
    location.reload();
},

removeConnection: function(e) {
    // Haetaan sen yhteyden id, jota klikattiin.
    var connectionId =
            e.currentTarget.className.split(' ')[1];

    var self = this;

    // Käydään yhteyskokoelma läpi Underscoren each-
    // funktiolla, joka toimii niin kuin for each -silmukka.
    _.each(this.connectionCollection.models,
        function(connection) {

        // Vertailuissa tulee aina käyttää joko === tai !==
        // merkintää. Kaksi yhtäsuuruutta ei ota muuttujan
        // tyyppiä huomioon, jolloin on mahdollisuus
        // virheeseen.

        // Etsitään se yhteys, jota klikattiin vertaamalla
```

```
                // id:itä ja poistetaan se kokoelmasta. Samalla
                // laukeaa remove-tapahtuma, jota mallimme kuuntelee.
                if(connectionId === connection.get('connectionId')) {
                    self.connectionCollection.remove(connection);
                }
            });

            // Pyydetään jqSimpleä poistamaan yhteys DOMista.
            jqSimpleConnect.removeConnection(connectionId);
        }
    });

    return View;
});
```

templates/main.html

```
<!-- Sivupohja on vain yksinkertainen html-tiedosto. -->
<div id="main">
    <span class="button new-rect">Uusi suorakulmio</span>
    <span class="button delete-diagram">Tyhjennä kaavio</span>
    <ol>
        <li>
            Luo yhdistinviiva kuvioiden välille klikkaamalla kuvioita.
        </li>
        <li>Poista yhdistinviiva klikkaamalla sitä.</li>
    </ol>
</div>
```

js/views/rectangle.js

```
// En ole kommentoinut samoja asioita tässä näkymässä,
// jotka olen jo selittänyt main-näkymässä.
define([
    'backbone',
    'text!../../templates/rectangle.html'
], function(
    Backbone,
    RectangleTemplate
){
    var View = Backbone.View.extend({
        // Emme valitse el-muuttujaksi mitään valmista
        // DOM-elementtiä, kuten main-näkymässä. Tällä
        // kertaa Backbone luo meille el-muuttujaksi
        // uuden div-elementin, jolle määritämme luokkanimen.
        className: 'shape rectangle',

        events: {
            // Aina, kun tämän kuvion input-kentän arvo muuttuu,
            // kutsutaan inputChange-funktiota.
            'change input': 'inputChanged'
        },

        initialize: function(options) {
```

```
    _.bindAll(
        this,
        'initialize',
        'inputChanged',
        'makeDraggable',
        'makeResizable',
        'render'
    );

    // Annetaan määrittelemämme sivupohja-tiedosto
    // underscoren template-muuttujalle, koska
    // haluamme käyttää sivupohjassa muuttujaa.
    // Tässä tapauksessa muuttuja on input-kentän
    // merkkijono, jonka käyttäjä voi vaihtaa.
    this.template = _.template(RectangleTemplate);

    // Annetaan el-muuttujalle eli tämän näkymän
    // div-elementille id, jonka saimme parametrina
    // main-näkymästä.
    this.$el.attr('id', this.options.id);

    // Asetetaan sama id myös mallille ja tallennetaan malli.
    this.model.set({
        shapeId: this.options.id
    });
    this.model.save();

    // Kutsutaan render-funktiota.
    this.render();

    // Kutsutaan funktioita, jotka tekevät kuviosta
    // skaalattavan ja liikuteltavan.
    this.makeDraggable();
    this.makeResizable();
},

inputChanged: function() {
    // Haetaan muuttunut merkkijono input-elementistä
    // ja tallennetaan se malliin.
    this.model.set({
        text: this.$('input').val()
    });
    this.model.save();
},

makeDraggable: function(shape) {
    var self = this;

    // Tehdään tämän näkymän el-muuttujasta liikuteltava
    // jQueryUI:n draggable-funktion avulla.
    this.$el.draggable({
        // Aina kun kuviota liikutellaan, piirretään
        // mahdolliset yhteydet uudestaan. Tätä voisi
        // optimoida, että piirrettäisiin vain tähän kuvioon
        // liittyvät yhteydet uusiksi.
        drag: function() {
            jqSimpleConnect.repaintAll();
        },
        stop: function() {
            // Kun liikuttelu loppuu, tallennetaan kuvion
```

```
            // muuttuneet koordinaatit malliin.
            self.model.set({
                top  : self.$el.css('top'),
                left : self.$el.css('left')
            });
            self.model.save({}, {
                // Voimme halutessamme määrittää success-
                // funktion, joka laukeaa silloin, kun mallin
                // tallentaminen on onnistunut. Error-funktio
                // on tämän vastakohta.
                success: function() {
                    // Console.log kirjoittaa selaimen
                    // konsoliin halutun merkkijonon tai
                    // vaikkapa JavaScript-olion. Tätä voi
                    // käyttää vaikkapa alkeelliseen
                    // debuggaukseen, mutta on suositeltavam-
                    // paa laittaa koodiin breakpointteja.
                    console.log(
                       'Uudet koordinaatit tallennettu.'
                    );
                    console.log(self.model);
                }
            });
        }
    });
},

makeResizable: function(shape) {
    // Toimii täsmälleen samoin, kuin edeltävä funktio. Tässä
    // tapauksessa jQueryUI tekee kuviosta skaalattavan.
    var self = this;
    this.$el.resizable({
        resize: function() {
            jqSimpleConnect.repaintAll();
        },
        stop: function() {
            self.model.set({
                width  : self.$el.css('width'),
                height : self.$el.css('height')
            });
            self.model.save({}, {
                success: function() {
                }
            });
        }
    });
},

render: function() {
    // Nyt emme kirjotakkaan suoraan sivupohjatiedostoamme
    // el-muuttujaan, vaan kirjoitamme sinne määrittämme
    // template-muuttujan. Template muuttujalle kerromme,
    // että text-muuttjan arvo sivupohjassa on joko mallista
    // löytyvä arvo tai mikäli sellaista ei löydy, niin sitten
    // merkkijono "kuvio".
    this.$el.html(this.template({
        text: this.model.get('text') || 'kuvio'
    }));

    // Määritellään div-elementtimme koko ja sijainti mallista
```

```
            // haettavilla tiedoilla.
            this.$el.css({
                'height' : this.model.get('height'),
                'width'  : this.model.get('width'),
                'left'   : this.model.get('left'),
                'top'    : this.model.get('top')
            });

            return this;
        }
    });
    return View;
});
```

templates/rectangle.html

```
<form>
    <!--
        Value sisältää muuttujan, jonka arvo annetaan näkymässä.
    -->
    <input type="text" value="<%- text %>" />
</form>
```

js/models/shape.js

```
define(['backbone'], function(Backbone){

    var Model = Backbone.Model.extend({
        // Annetaan mallin parametreille oletusarvot.
        defaults: {
            left    : 50,
            top     : 50,
            width   : 100,
            height  : 80,
            type    : undefined,
            id      : null,
            shapeId : '',
            text    : ''
        },

        // Localstorage nimen on oltava sama, kuin kokoelmassa,
        // jotta mallit tallentuvat oikeaan paikkaan.
        localStorage: new Backbone.LocalStorage("shape-collection")
    });

    return Model;
});
```

js/models/connection.js

```
define(['backbone'], function(Backbone){

    var Model = Backbone.Model.extend({
        defaults: {
```

```
            // Yhteysmallin attribuutit ovat id, lähtökuvion id sekä
            // kohdekuvion id.
            id           : null,
            // Tämä annetaan jqSimple kirjastoa varten, jotta se
            // osaa poistaa yhteyksiä.
            connectionId : null,
            sourceId     : '',
            targetId     : ''
        },

        // Localstorage nimen on oltava sama, kuin kokoelmassa,
        // jotta mallit tallentuvat oikeaan paikkaan.
        localStorage:
            new Backbone.LocalStorage("connection-collection")
    });

    return Model;
});
```

js/collections/shape.js

```
define([
    'underscore',
    'backbone',
    'models/shape'
], function(
    _,
    Backbone,
    ShapeModel
){
    var Collection = Backbone.Collection.extend({
        // Kokoelma sisältää tiedon mallistaan sekä nimen, jolla
        // kokoelma tallennetaan local storageen.
        model        : ShapeModel,
        localStorage : new Backbone.LocalStorage("shape-collection")
    });

    return Collection;
});
```

js/collections/connection.js

```
define([
    'underscore',
    'backbone',
    'models/connection'
], function(
    _,
    Backbone,
    ConnectionModel
){
    var Collection = Backbone.Collection.extend({
        // Kokoelma sisältää tiedon mallistaan sekä nimen, jolla
        // kokoelma tallennetaan local storageen.
        model        : ConnectionModel,
        localStorage :
```

```
            new Backbone.LocalStorage("connection-collection")
    });

    return Collection;
});
```

scss/style.scss

```
// Tämä tiedosto kokoaa kaikki muut (tässä tapuksessa main ja shape)
// SASS-tiedostot yhteen. Tästä tiedostosta SASS luo sitten
// lopullisen CSS-tiedoston.

// Alitiedostojen nimet alkavat alaviivalla, jotta SASS ymmärtää
// ettei niistä tule luoda omia CSS-tiedostojaan. Alaviivaa ei
// kuitenkaan mainita importin polussa, vaan SASS osaa täydentää
// sen itse.
@import "main";
@import "shape";
```

scss/_main.scss

```
#main {
    .button {
        border        : 1px solid #000;
        border-radius : 5px;
        padding       : 5px;
    }

    .new-rect {
        position : absolute;
        top      : 5px;
        left     : 5px;
    }

    .delete-diagram {
        position : absolute;
        top      : 5px;
        left     : 145px;
    }

    .button:hover {
        color  : #f00;
        cursor : pointer;
    }

    .ui-draggable {
        cursor : move;
    }

    ol {
        padding-top: 30px
    }
}
```

scss/_shape.scss

```
.shape {
    border-radius    : 5px;
    background-color : #eee;
    border           : 1px solid #000;
    padding          : 15px;
    position         : absolute !important;
    display          : inline-block;

    input {
        position   : absolute;
        top        : 40%;
        left       : 10px;
        right      : 10px;
        border     : 0;
        background : transparent;
        text-align : center;
    }
}
```