Samppa Valkama

JavaScript nykyaikaisessa web-kehityksessä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Mediatekniikan koulutusohjelma

Insinöörityö

7.4.2014

Metropolia

Tekijä Otsikko Sivumäärä Aika	Samppa Valkama JavaScript nykyaikaisessa web-kehityksessä 51 sivua 7.4.2014
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaajat	Teknologiajohtaja Teemu Huttunen Lehtori Ilkka Kylmäniemi

Insinöörityön tarkoituksena oli toimia perehdytyksenä ohjelmointiyrityksen käyttämiin teknologioihin sekä oppia web-kehityksestä JavaScript-ohjelmointikielellä.

Insinöörityössä perehdyttiin nykyaikaisen JavaScriptin tarjoamiin mahdollisuuksiin kuten käytetyimpiin kirjastoihin ja verrattiin niiden käyttöä tavalliseen JavaScriptiin. Tutkittiin miten REST-arkkitehtuuri toimii käytännössä ja miten sitä voidaan testata cURL-komentojen avulla.

JavaScript on yksi maailman suosituimmista ohjelmointikielistä, ja sen suosio kasvaa jatkuvasti HTML5:n suosion myötä. Se toimii lähestulkoon kaikissa älylaitteissa, joten se tavoittaa enemmän käyttäjiä kuin mikään muu yksittäinen ohjelmointikieli. JavaScriptin avulla voidaan toteuttaa perinteisten Internet-sovellusten lisäksi myös palvelinympäristö, joten kehittäjien on helpompi työskennellä molempien tasojen parissa samanaikaisesti.

Insinöörityössä toteutettiin nykyaikainen REST-arkkitehtuurimallilla toimiva verkkosovellus käyttämällä ainoastaan JavaScript-ohjelmointikieltä. Sovellus rakennettiin MVC-ohjelmistoarkkitehtuurityyliä käyttäen. MVC-arkkitehtuurin tarkoituksena on eriyttää eri osa-alueiden komponentit siten, että ne ovat mahdollisimman vähän riippuvaisia toisistaan. Verkkosovelluksen lisäksi toteutettiin yksinkertainen Node.js-palvelin, joka tallentaa ja hakee tietoa JSON-tietomuodossa MongoDB-tietokannasta.

Sovelluksen toteutusvaiheessa perehdyttiin siihen, kuinka kirjastoriippuvuuksia voidaan hallita paketinhallintatyökaluilla. Lopuksi sovellusta optimointiin minimoimalla HTTP-hakujen määrä sekä pakkaamalla ohjelmakoodit.

Avainsanat	JavaScript, REST, MVC-arkkitehtuuri, Backbone, Node.js, Web-kehitys

Author Title Number of Pages Date	Samppa Valkama JavaScript in modern web development 51 pages 7 April 2014
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Teemu Huttunen, Chief of Technology Ilkka Kylmäniemi, Lecturer

The object of this thesis was to be an introduction to technologies used in the information technology company and to learn about modern web development with JavaScript.

JavaScript is one of the most widely used programming languages. Its popularity has been rising since the HTML5 arrived. JavaScript functions on almost every smartphone and thus it is more wide spread than any other programming language. It is possible to implement servers with JavaScript in addition to traditional web applications. It eases the gap between front-end and back-end developers since the same language is used on both ends.

A modern web application was built for this project. It was implemented with pure JavaScript and it follows the principles of the Model-View-Controller software pattern. The main principle of MVC is that software components are separated from each other so that even a bigger codebase is easier to maintain and keep clean. A simple server implementation with Node.js was built to provide solid base for the web application. All application data were saved to MongoDB database in JSON format by the Node.js server.

Another software architectural style used was the representational state transfer (REST) which was introduced in HTTP protocol version 1.1. REST implementation was thoroughly tested with cURL.

Working with package managers and JavaScript helper libraries was looked into during the project. At the end the project was optimized with Grunt by minifying and concatenating all the source code.

Keywords	JavaScript, REST, MVC-pattern, Backbone, Node.js, Web development,

Sisällys

Lyhenteet ja käsitteet

AMD	Asynchronous Module Definition, mahdollistaa tiedostojen ja niiden riippuvuuksien asynkronisen lataamisen tarvittaessa.
C/C++	UNIX-käyttöjärjestelmälle kehitetty imperatiivinen ohjelmointikieli. C++ pohjautuu C-kieleen, mutta siihen on lisätty uusia ominaisuuksia.
CSS	Cascading Style Sheets, WWW-dokumenttien tyylimäärittelykieli.
cURL	Komentorivityökalu, jonka avulla voidaan lähettää ja vastaanottaa HTTP-komentoja.
DOM	Document Object Model, dokumenttioliomalli, W3C:n määrittelemä ohjelmointirajapinta. Yhdessä JavaScriptin kanssa sillä voidaan toteuttaa interaktiiviisia sivustoja.
Flash	Adobe Systemsin rakentama kehitysympäristö, jolla voidaan tuottaa multimediaesityksiä internettiin.
HTML	HyperText Markup Language, hypertekstidokumenttien merkintäkieli. Käytännössä kaikki www-sivut on tehty HTML-kielellä. Se on tarkoitettu käytteväksi muiden web-tekniikoiden ja ohjelmien kanssa.
HTML5	HTML:n uusin versio, ja se tarkoittaa toisiinsa kytkeytyviä asioita kuten HTML -merkintäkieltä, CSS-tyylimääritteitä sekä JavaScriptillä toteutettuja toiminnallisuuksia.
Java	Laaja ohjelmistoalusta, johon kuuluu muun muassa oliopohjainen ohjelmointikieli luokkakirjastoineen.
Java-sovelma	Asiaskaspuolen tietokoneen selaimessa suoritettava Java-ohjelma.

JavaScript	Yleiskäyttöinen, kevyt ja löyhästi tyypitetty ohjelmointikieli, jota käytetään pääasiassa internetohjelmoinnissa dynaamisuuden lisäämiseksi web-sivuille.
JSON	JavaScript Object Notation, JavaScript-pohjainen tiedonsiirtomuoto. Nimestään huolimatta sitä voidaan käyttää myös muilla ohjelmointikielillä, ei pelkästään JavaScriptillä.
MVC	Model-View-Controller eli malli-näkymä-käsittelijä, ohjelmistoarkkitehtuuri, jonka avulla sovellus erotellaan kolmeen osaan.
MySQL	Relaatiotietokantaohjelmisto, joka on hyvin suosittu web-palveluiden tietokantana.
NPM	Node Packaged Modules, Node.js:n työkalu pakettien hallintaan. Sen avulla voidaan helposti asentaa ja hallita tarvittavia liitännäisiä sovellukseen. Käytetään komentoriviltä esimerkiksi kommenolla "npm install <package>".
Perl	Practical Extraction and Report Language, tulkattava skriptimäinen ohjelmointikieli.
PHP	Hypertext Preprocessor, skriptimäinen ohjelmointikieli, jota käytetään laajasti verkkopalvelimilla dynaamisten verkkosivujen yhteydessä.
REST	Representational State Transfer, HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen.
SQL	Structured Query Language, kyselykieli, jonka avulla relaatiotietokantaan voi tehdä hakuja, muutoksia ja lisäyksiä.
TCL	Tool Command Language, tulkattava ohjelmointikieli, jota käytetään muun muassa ohjelmien komentosarjakielenä ja testauksessa.

URI	Uniform Resource Identifier, merkkijono, jolla kerrotaan tietyn tiedon paikka. URIa hyödynnetään RESTful-web-sovellusten kutsuissa.
W3C	World Wide Web Consortium, kansainvälinen yhteenliittymä, joka ylläpitää ja kehittää web-standardeja.

1 Johdanto

Insinöörityön tarkoituksena on tutkia ja selvittää, kuinka JavaScriptiä käytetään osana nykyaikaisia verkkosovelluksia ja -palveluita. Työssä perehdytään MVC-ohjelmistoarkkitehtuurin eli malli-näkymä-käsittelijäarkkitehtuurin käyttämiseen ja sen erilaisiin ominaisuuksiin ja siihen, miten ne voivat helpottaa kehittäjien työtä. Kun MVC-arkkitehtuuri hoitaa käyttäjälle näkyviä komponentteja, tutkitaan myös, miten JavaScriptillä toteutettu Node.js-ratkaisu toimii käytännössä palvelimella.

JavaScriptin roolia ei voi vähätellä nykyaikaisessa web-kehityksessä. Nykyään lähes jokaisella rakennetulla kotisivulla on käytetty enemmän tai vähemmän JavaScriptiä. Sen avulla voidaan rikastuttaa selaimessa tapahtuvaa interaktiivisuutta ja parantaa sivuston käyttökokemusta huomattavasti. Flashin poistumisen ja uusien HTML5-elementtien, kuten canvas, video ja audio, myötä JavaScript tarjoaa myös keinot multimediakokemuksen rakentamiseen. Käyttökokemusta voidaan parantaa myös kaventamalla selainten välisiä eroja käyttämällä JavaScript-kirjastoja.

JavaScriptin rooli ei kuitenkaan rajoitu yksin kotisivujen tekemiseen käytetyksi ohjelmointikieleksi, vaan se toimii melkein missä tahansa laitteessa älypuhelimesta lukulaitteisiin. JavaScript on nykyään varteenotettava vaihtoehto myös palvelimella käytettäväksi ohjelmointikieleksi.

JavaScriptin suosiota lisää myös sen keveys ja se, että kieli on varsin helppo oppia verrattuna C- tai Java-ohjelmointikieliin. JavaScriptillä on takanaan suuri yhteisö kannattajia ja kehittäjiä, joten internetpalstoilta löytyy aina apua ongelmien ratkaisuun. Myös kielen opetteluun on saatavilla erinomaiset resurssit, joten ei ole ihme, että sen suosio kasvaa jatkuvasti. JavaScriptiä ei tarvitse kääntää erikseen käyttöä varten vaan, se käännetään käyttäjän selaimessa, minkä ansiosta palvelinkoneelta ei tarvita niin paljoa suorituskykyä.

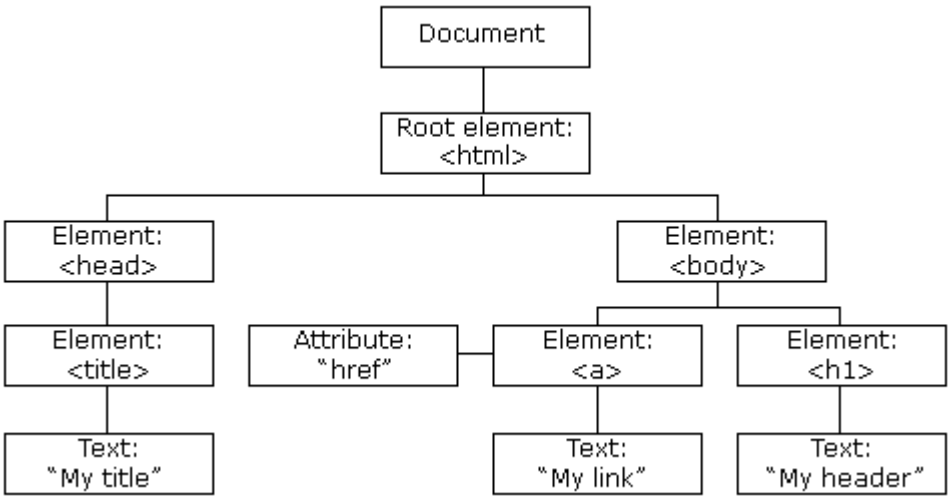
Olen insinöörityön tekoa aloittaessani lähes täysin kokematon MVC-arkkitehtuurien käytössä ja vasta-alkaja JavaScript-kehityksessä. Insinöörityön tarkoituksena on luoda työn tekijälle vahva perustietopohja MVC-arkkitehtuureista ja JavaScript-ohjelmointikielestä yleisesti. Insinöörityö toteutetaan Pieni piiri Oy:lle, ja sen tarkoituksena on toimia johdatuksena yrityksen käyttämiin teknologioihin.

2 Käytetyt teknologiat

2.1 Web-kehitys aikaisemmin

Ennen JavaScriptin aikakautta verkkosivuilla käytettiin erilaisia teknologioita, kuten PHP:ta ja Java-sovelmia, jotka olivat silloin suosittuja ohjelmointikieliä. Tietysti näitä tekniikoita käytetään edelleen joillakin sivustoilla, ja ne sopivat sinne erinomaisesti. Kehittäjät kokivat JavaScriptin alussa hyvin hankalaksi ja sekavaksi skriptauskieleksi, eikä se ollut lainkaan suosittu alkuaikoina.

Verkkosivustot itsessään olivat hyvin pitkälti staattisia, ja JavaScriptiä käytettiin ainoastaan antamaan hieman lisämaustetta sivustolle. Kuvassa 1 nähdään esimerkki HTML-sivun DOM-puusta eli sivuston puurakenteesta. Sitä muokattiin vähäisessä määrin, mikä on nykyään yksi JavaScriptin keskeisimmistä tehtävistä. Graafisesta lisukkeesta, kuten vaihtuvista kuvista ja animaatioista, vastasi useimmiten Flash. Graafisuutta toteutettiin aikaisemmin myös jonkin verran JavaScriptin avulla, sillä CSS-tyylimääritteet eli internetsivustojen tyylikieli eivät olleet nykyisellä tasolla. JavaScriptin avulla saatettiin esimerkiksi vaihtaa linkkinäppäimen taustaväriä tai -kuvaa hiiren osoittimen ollessa sen kohdalla mouseOver-funktion avulla.



Kuva 1. Esimerkki HTML-sivun DOM-puurakenteesta [1].

Datan liikuttelu palvelimelta käyttäjän nähtäväksi ja toisinpäin toteutettiin useasti PHP:n ja SQL:n eli relaatiotietokantojen kanssa toimivan kyselykielen avulla. PHP on edelleen ylivoimaisesti eniten käytetty ohjelmointikieli palvelinpuolella. Yli 81 % sivustoista, joiden palvelimella käytetty ohjelmointikieli tiedetään, käyttää PHP:ta palvelimellaan [2].

PHP:lla on takanaan suuri kehittäjien yhteisö, ja vastaan tuleviin ongelmiin on helppo saada apua sieltä.

MySQL on edelleen yksi suosituimmista relaatiotietokantajärjestelmistä. Se muodostuu tietokannoista, jotka sisältävät tauluja, jotka puolestaan koostuvat kentistä ja riveistä. Perinteisesti tällaiseen tietokantaan tallennettiin esimerkiksi lista verkkokaupan tuotteista ja asiakkaista. Yksi tietokannan tauluista voisi sisältää tiedot asiakkaista, jolloin taulun kenttinä voisivat olla asiakkaan asiakasnumero, nimi, yhteystiedot ja salasana. Yksi rivi edellä mainitussa taulussa sisältäisi tiedot yhdestä uniikista käyttäjästä.

2.2 JavaScript-ohjelmointikieli

Vuonna 1995 verkkoselainmarkkinoita hallitsi Netscape Navigator. Netscape päätti rikastuttaa käyttäjiensä kokemusta lisäämällä interaktiota. Lopputuloksena syntyi kevyt oliopohjainen ohjelmointikieli. Ensimmäinen versio kielestä syntyi vain kymmenessä päivässä Brendan Eichin toimesta, ja se sai nimekseen Mocha, joka kuitenkin pian muutettiin LiveScriptiksi. Vuoden 1995 joulukuussa Netscape ja Sun valmistelivat lisensointisopimuksen, jonka myötä kieli sai lopullisen nimensä, JavaScript. Samalla Netscape julkaisi ensimmäisen selainmensa, joka sisälsi tuen JavaScriptille. [3, s. 37.]

JavaScriptin alkuperäinen tarkoitus oli lisätä tuki Java-sovelmille selaimessa ja lisäksi tehdä niiden kehittämisestä niin helppoa, että myös muut kuin kokeneet Java-kehittäjät pystyisivät niitä tekemään. Vaikka JavaScript oli helppokäyttöinen, eivät kaikki kehittäjät kuitenkaan ottaneet sitä lämmöllä vastaan, sillä se vaikutti monilta osin keskeneräiseltä. Vaikeasta alusta huolimatta JavaScript on nykyään käytännössä ainut verkkoselaimissa käytetty ohjelmointikieli.

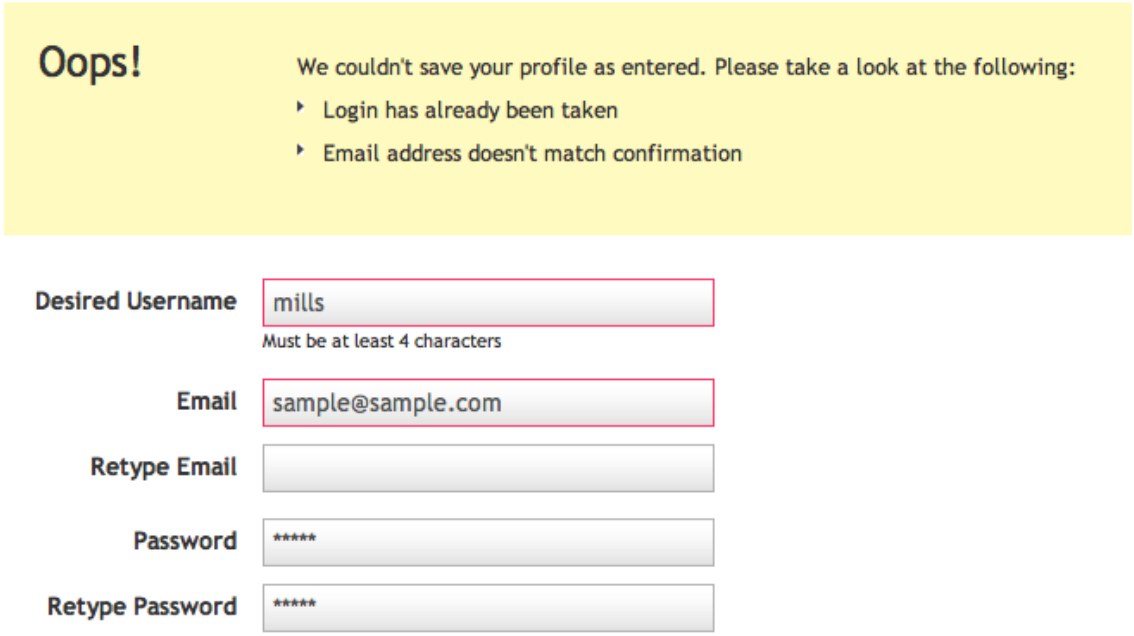
JavaScript kehittyi erittäin tehokkaaksi web-ohjelmointikieleksi, sillä siihen koostettiin parhaat ominaisuudet Perlistä, kuten hakurakenne, löysä tyypitys ja säännölliset lausekkeet. Toisaalta siihen kehitettiin myös C:n, C++:n ja Javan parhaita ominaisuuksia, kuten syntaksi, oliot ja luokat sekä matemaattiset käsittelijät. TCL:stä JavaScriptiin tuotiin selaintuki, jota on pidetty yllä aina nykypäivään saakka. [4.]

JavaScriptin alkuaikoina interaktiot olivat hyvin mitättömiä. Yleensä ne olivat jotakin palautetta käyttäjälle esimerkiksi alert-funktion muodossa. Tällaiset ponnahdusikkunat kuten alert koettiin ärsyttäviksi, mutta muita keinoja asioiden viestimiseen loppukäyttä-

jälle ei ollut. Jos lomakkeesta jäi jokin kohta käyttämättä, voitiin näyttää viesti, jossa kehotettiin täyttämään kaikki kohdat. Toinen paljon käytetty oli confirm-funktio, jonka avulla voitiin kysyä käyttäjältä, saadaanko jokin toiminto suorittaa. Molempia käytetään tänäkin päivänä, mutta asian toteutustapa on muuttunut ajan kuluessa.

Web-sivustoja pitää usein muuttua reaaliaikaisesti käyttäjän toimintojen mukaan ilman palvelinyhteyttä, mikä onnistuu JavaScriptin avulla DOM:a (Document Object Model) muokkaamalla. DOM on puurakenteinen dokumentaatiomalli, joka yhdessä JavaScriptin kanssa mahdollistaa paljon rikkaampien interaktioiden toteuttamisen internetsivuilla. Se on yksi keskeisimmistä JavaScriptin käyttötavoista nykyaikaisessa web-kehityksessä.

JavaScriptin avulla DOM:ssa esiteltyjä elementtejä voidaan piilottaa tai näyttää tai niiden ulkonäköä ja sisältöä voidaan muuttaa. Myös aivan uusien elementtien luominen onnistuu "lennosta". Yhdessä JavaScriptin tapahtumien käsittelyn kanssa voidaan ilmoittaa käyttäjälle paljon tarkemmin, mikäli jokin kohta jäi lomakkeesta täyttämättä, ja se voidaan myös esittää kauniimmalla ja vähemmän hyökkäävällä tavalla kuin alert-funktiolla, kuten kuvassa 2 voidaan nähdä. Tapahtumien käsittelyllä tarkoitetaan sitä, että sivusto reagoi käyttäjän toimintoihin ilman uusia latauksia vain DOM:a muokkaamalla. Mahdollisuudet ovat käytännössä rajattomat.



Kuva 2. Käyttäjälle näytetään havainnollistavaa tietoa lomakkeen puuttuvista kohdista [5].

JavaScriptin avulla voidaan myös vaikuttaa web-sovelluksen graafiseen ilmeeseen. Elementtejä voidaan animoida ja liikuttaa selaimessa tai niitä voidaan värjätä eri väreillä huomion saamiseksi. Käyttöliittymää saadaan usein myös responsiivisemmaksi JavaScriptin avulla. Responsiivisuudella eli mukautuvuudella tarkoitetaan, että sivusto skaalautuu erikokoisille näyttöpäätteille kauniisti ja siten, että käytettävyys ei kärsi. Responsiivisuus on tärkeää web-kehityksessä, sillä internetiä selataan yhä enemmän matkapuhelimien ja tablet-laitteiden avulla. [5.]

2.3 Nykyaikainen JavaScript

JavaScriptin suosio on kasvanut 2010-luvulla paljon, ja se tarjoaa vaihtoehtoisen toteutustavan kaikelle, mitä luvussa 2 on aikaisemmin esitetty. Selaimessa JavaScript onkin jo lähes universaalisti käytettyä teknologiaa, mutta myös palvelintoteutus onnistuu JavaScriptillä. Lisäksi palvelimen ja selaimen välillä liikutellaan dataa JSON-tiedostomuodossa, joka sekin on JavaScriptiä.

Yhä useammat sovellukset, jotka aikaisemmin oli toteutettu työpöydällä toimiviksi, ovat nykyään olemassa myös verkkosivun muodossa. HTML5:n ja erityisesti JavaScriptin kehitys on mahdollistanut verkkosivustojen käyttämisen tehokkaasti, ja lähes kaikki samat toiminnallisuudet voidaan toimittaa myös verkkoympäristöön.

Nykyään on mahdollista toteuttaa paljon pelkästään JavaScriptin avulla. Tietoa voidaan tallentaa huomaamatta käyttäjän tietokoneella hänen täyttäessään pitkään lomaketta, ja jos jokin menee vikaan, jo täytetyt tiedot eivät hukkuneet lomakkeesta minnekään, vaan täyttyvät valmiiksi lomakkeeseen käyttäjän saapuessa sivulle. Lisäksi lomakkeen täyttämistä voidaan myös nopeuttaa ehdottamalla käyttäjälle mahdollisia valintoja. Samalla tavoin voidaan automaattisesti ehdottaa täydennystä hakukentässä suosituimpien hakusanojen avulla. Toisaalta myös uutta sisältöä voidaan ladata koko ajan ilman, että käyttäjä päivittää sivua. Esimerkiksi uutiset päivittyvät tietyin väliajoin vain yhteen sivuston elementtiin ilman koko sivuston päivittymistä.

JavaScriptiin on lukuisia kirjastoja ja liitännäisiä, jotka helpottavat kehittäjien työtä. Joidenkin kirjastojen tarkoituksena on vain helpottaa JavaScriptin integraatiota jonkin toisen ohjelmointikielen kanssa, kuten esimerkiksi PHP:n tai Rubyn.

Ylivoimaisesti eniten käytetty JavaScript-kirjasto on jQuery, joka helpottaa HTML-tiedoston DOM-rakenteen manipulointia ja tapahtumien käsittelyä ja yksinkertaistaa JavaScriptin syntaksia monin paikoin. [6.] Juuri helppokäyttöisyyden vuoksi jQuery on saavuttanut eniten käytetyn kirjaston aseman. Koodiesimerkki 1 havainnollistaa, kuinka jQuery helpottaa kehittäjien työtä. Molemmat funktiot toteuttavat saman asian, mutta jQuery hakee tekstikentän www-dokumentista huomattavasti lyhyemmällä koodilla ja käyttää valmista text-funktiota sen muuttamisessa. [7.]

```
1  function changeTextJquery(){
2          $('p#hello').text('jQuery greets you!')
3  }
4
5  function changeTextJavaScript(){
6          document.getElementById('hello').innerHTML = 'JavaScript greets you!';
7  }
```

Koodiesimerkki 1. Tekstin muutos toteutettuna jQueryn ja JavaScriptin avulla

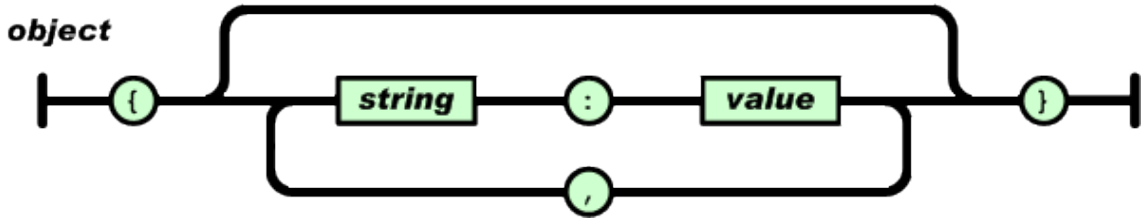
Eräs jQueryn kilpailija on MooTools, joka pyrkii myös helpottamaan JavaScript-kehittäjän DOM-rakenteen hallintaa ja muita toiminnallisuuksia sisäänrakennettujen valmiiden funktioiden avulla. MooTools on oliokeskeinen kirjasto, minkä takia se ei ole aivan niin helppo oppia kuin jQuery. MooTools mainostaakin itseään soveliaaksi keskinkertaisesta etevälle JavaScript-kehittäjälle.

Joskus mietittiin, voidaanko käyttää erilaisia kirjastoja, sillä käyttäjän täytyy ladata ne palvelimelta sivun latauksen yhteydessä. Onneksi kuitenkin nykyään internet-yhteydet ovat yleensä todella nopeita ja käytetyt kirjastot puolestaan mahdollisimman pieneksi pakattuja. Lisäksi käytetyimmät kirjastot usein löytyvät käyttäjän selaimen välimuistista jo valmiiksi, joten latausta ei tarvita. [8.]

2.4 JSON-tiedonsiirtomuoto

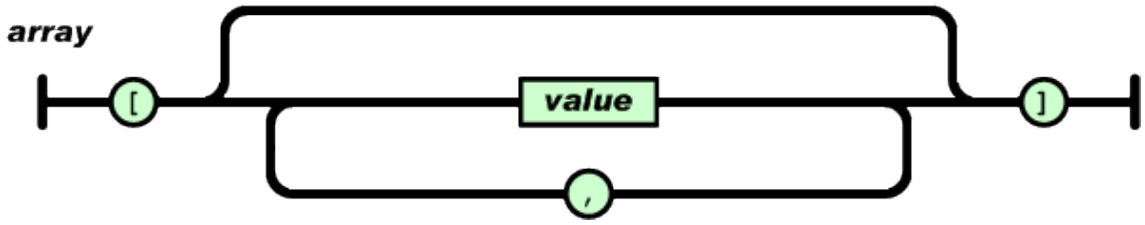
JSON (JavaScript Object Notation) on JavaScript-pohjainen yksinkertainen tiedonsiirtomuoto, vaikka se onkin täysin riippumaton käytetystä ohjelmointikielestä. JSONin perimmäisenä tarkoituksena on olla käyttäjälle helppo lukea ja kirjoittaa sekä tietokoneille helppo jäsennellä ja toteuttaa.

JSONin syntaksi on itseselitteinen ja yksinkertainen. JSON noudattaa kahta erilaista rakennetta: kokoelma olioita tai kokoelma jonoja. Oliolla tarkoitetaan nimi-arvoparia, joka esitetään aaltosulkeiden välissä. Kuvassa 3 on graafinen malli JSON-oliosta.



Kuva 3. JSON-olio [9].

Jonot puolestaan ovat järjestettyjä kokoelmia, jotka voivat sisältää arvoja tai olioita. Jonot esitetään hakasulkeiden välissä, ja listan alkiot on erotettu toisistaan pilkulla. Kuvassa 4 on graafinen malli JSON-jonosta.



Kuva 4. JSON-jono [9].

JSONissa arvo voi olla merkkijono, joka esitetään lainausmerkeissä tai se voi olla numero, totuusarvo, kuten "true" tai "false", tai null, joka tarkoittaa määrittelemätöntä. JSON voi sisältää sisäkkäisiä olioita tai jonoja. Koodiesimerkkissä 2 on esimerkki JSON-oliosta, joka erityyppisiä arvoja. Helppolukuisuuden lisäksi JSON-tietomuotoa on helppoa käyttää JavaScript-koodissa. JavaScriptin avulla JSON-olio voidaan tallentaa suoraan JavaScript-olioksi, sillä tietomuoto on täysin sama. Ylimääräistä datan parsimista ei tarvitse suorittaa, ja JavaScript-oliosta saadaan nopeasti haettua haluttu tietosisältö. [9.]

```
{
    name: "John Smith",
    age: 38,
    married: false,
    contact: [
        {
            phone: 1234567,
            email: "johnsmith@example.com"
        }
    ]
}
```

Koodiesimerkki 2. Esimerkki JSON-tiedonsiirtomuodosta.

2.5 RESTful-verkkopalvelut

REST (Representional State Transfer) on HTTP-protokollan 1.1-versiossa esitelty hybridiarkkitehtuurityyli, joka soveltuu erityisesti verkkosovellusten toteutukseen. REST on tilaton asiakas-palvelinmalli, eli se ei ylläpidä tietoa asiakkaidensa tilasta. Lisäksi jokainen palvelimelle tehty kutsu sisältää kaiken tarpeellisen tiedon vastauksen tuottamiseen. HTTP-protokollan ansiosta REST on täysin riippumaton käytetystä käyttöjärjestelmä tai ohjelmointikielestä. Käyttäjällä voi olla käytössään Windows-käyttöjärjestelmä ja palvelimella Unix-pohjainen käyttöjärjestelmä.

REST-mallissa selain tekee palvelimelle kutsun, jonka palvelin käsittelee ja palauttaa asianmukaisen vastauksen. REST-malliin perustuvia verkkopalveluita, jotka on toteutettu HTTP-protokollaan, kutsutaan yleisesti RESTful-verkkopalveluiksi. Ne hyödyntävät erilaisia HTTP-metodeja kutsuissa ja vastauksissa. REST:n hyödyntämiä metodeja ovat GET, POST, PUT ja DELETE.

Tiedettyä dataa, joka on tunnistettu URI:n (Uniform Resource Identifier) avulla, voidaan hallita käyttämällä edellä mainittuja HTTP-metodeja. Perinteisesti verkkopalvelussa tällaisia URI:a käytetään esimerkiksi käyttäjätietojen hakemisessa. URI, jonka avulla haettaisiin kaikki palvelun käyttäjätiedot, voisi olla http://example.com/users/. Yksittäisen käyttäjän tiedot puolestaan saataisiin URI:sta http://example.com/users/666, jossa viimeinen osa kenoviivan jälkeen on uniikki käyttäjätunniste. [10.]

HTTP GET

GET-metodilla voidaan hakea mitä tahansa tietoa, joka on tunnistettu selaimen pyytämän URI:n avulla. Palvelin palauttaa selaimelle URI:a vastaavan sisällön. [10, s. 53.] Taulukossa 1 on esitelty, kuinka GET-metodia käytetään.

Taulukko 1. Esimerkki GET-metodin käytöstä.

Resurssi-URI	**HTTP GET**
Kokoelman URI, http://example.com/resources	Haetaan listaus kaikista kokoelman alkioista.
Yksittäisen tietosisällön URI, http://example.com/resources/item1	Haetaan yksittäisen tietosisällön tiedot kyseisestä kokoelmasta.

HTTP POST

POST-metodia käytetään uusien resurssien luomiseksi palvelimelle. Esimerkiksi käyttäjäkokoelmaan voidaan lisätä uusia käyttäjiä POST-metodia käyttämällä. [10, s. 54.] Taulukossa 2 on esitelty, kuinka POST-metodia käytetään.

Taulukko 2. Esimerkki POST-metodin käytöstä.

Resurssi-URI	**HTTP POST**
Kokoelman URI, http://example.com/resources	Luo uuden yksittäisen jäsenen kokoelmaan.
Yksittäisen tietosisällön URI, http://example.com/resources/item1	Ei käytetä, sillä uuden yksittäisen jäsenen luominen toteutetaan ylemmän tason URI:n avulla.

HTTP PUT

PUT-metodin avulla suoritetaan tietosisällön muuttamista ja päivittämistä. POST- ja PUT-metodit ovat käytännössä täysin samanlaisia, mutta PUT-metodi tarkastaa, löytyykö muutettavalta datalta ID-arvoa. [10, s. 54.] Taulukossa 3 on esitelty, kuinka PUT-metodia käytetään.

Taulukko 3. Esimerkki PUT-metodin käytöstä.

Resurssi-URI	**HTTP PUT**
Kokoelman URI, http://example.com/resources	Korvaa tai päivittää kokoelman uudella kokoelmalla.
Yksittäisen tietosisällön URI, http://example.com/resources/item1	Korvaa tai päivittää yksittäisen tietosisällön tietoa/tietoja kyseisessä kokoelmassa, tai luo uuden, mikäli tietoa ei ole olemassa ennestään.

HTTP DELETE

Käyttämällä DELETE-metodia voidaan poistaa tietosisältöä palvelimelta. Sen avulla voidaan poistaa kokonainen kokoelma käyttäjiä tai vain yksi ainut käyttäjä riippuen URI:sta, johon DELETE-metodi on kohdennettu. [10, s. 55.] Taulukossa 4 on esitelty, kuinka DELETE-metodia käytetään.

Taulukko 4. Esimerkki DELETE-metodin käytöstä.

Resurssi-URI	**HTTP DELETE**
Kokoelman URI, http://example.com/resources	Poistaa koko kokoelman.
Yksittäisen tietosisällön URI, http://example.com/resources/item1	Poistaa yksittäisen tietosisällön kyseisestä kokoelmasta. Esimerkiksi yhden henkilön tiedot monien henkilöiden kokoelmasta.

MVC-arkkitehtuuri

MVC (Model-View-Controller) eli malli-näkymä-käsittelijäarkkitehtuuri auttaa kehittäjiä organisoimaan koodia. Arkkitehtuurin perusajatus on, että tuotetun koodin toiminnallisuudet jaetaan kolmeen eri osioon: malliin, näkymään ja käsittelijään. MVC-arkkitehtuuri on paljon käytetty toimintamalli ohjelmistokehityksessä, ja sitä voidaan soveltaa käyttöön myös JavaScriptin kanssa.

Kun sovellukseen kertyy useampi tuhat riviä koodia, on hankalaa löytää oikeat kutsujat ja funktiot, jos kaikki toiminnallisuudet ovat koodissa satunnaisessa järjestyksessä ja jopa yhdessä tiedostossa. MVC-arkkitehtuurin avulla koodiin saadaan selkeä rakenne,

ja sen vuoksi ohjelmakoodi on ymmärrettävämpää ja helpommin ylläpidettävissä pidemmällä aikavälillä. Jos samaa sovellusta on kehittämässä useita kehittäjiä, yhteiset tavat ja rakenne koodissa helpottavat kehittämistä.

Aivan yksinkertaisimman kotisivun tai sovelluksen tekemiseen MVC-arkkitehtuureja ei välttämättä kannata lähteä sekoittamaan, sillä ne vaativat aina oman työnsä pystyttämisessä, erityisesti jos arkkitehtuuri on ennalta tuntematonta teknologiaa. MVC-arkkitehtuurit ovat kuitenkin erittäin suosittuja yhden sivun verkkosivujen toteutukseen, kuten Facebookin tai GMailin.

Ajan kuluessa JavaScript-MVC-arkkitehtuurit ovat hioutuneet tarpeiden mukaan, ja monet arkkitehtuurit eivät täysin noudata MVC-arkkitehtuurimallia. Tällainen arkkitehtuuri on esimerkiksi Backbone.js, jota tässä insinöörityössä käytetään apuna esimerkeissä. Backbone-arkkitehtuurissa osa käsittelijän tehtävistä toteutetaan näkymissä ja osa Backbone.router-reitittimessä. Lisäksi muita komponentteja on tuotu puuttuvan käsittelijän tilalle. Backbonesta puhutaan useasti käsittelijän puuttuessa MC*-arkkitehtuurina. [12.]

Monessa arkkitehtuurissa käytetään kokoelmia (collections), joiden avulla on helppo käsitellä useampaa samanlaista mallia yhdessä näkymässä. Esimerkki joukko "kappale"-malleja voisi yhdessä muodostaa kokoelman "albumi". [11.]

Malli

Malli kuvaa ohjelman tietorakennetta tai yhden mallin tietorakennetta. Yhdessä ohjelmassa voi olla useita erilaisia malleja, esimerkiksi käyttäjämalli, joka sisältää uniikin tunnuksen, nimen ja sähköpostiosoitteen, kuten koodiesimerkkissä 3. Mallissa siis määritellään yhdelle oliolle datasisältö ja datan tyyppi. Data voi olla mitä vain tekstisisällöstä totuusarvomuuttujiin.

```
1  var Person = Backbone.Model.extend({
2      defaults: {
3          "id": null,
4          "firstName": "John",
5          "lastName": "Smith",
6          "email": "john.smith@example.com"
7      }
8  });
9
10 var Samppa = new Person({
11     firstName: 'Samppa',
12     lastName: 'Valkama',
13     email: 'samppa.valkama@metropolia.fi'
14 });
```

Koodiesimerkki 3. Backbone-arkkitehtuurilla toteutettu Person-malli.

Koodiesimerkissä 3 luodaan ensin malli "Person", jolle annetaan oletusarvoja. Oletusarvot eivät ole välttämättömiä, mutta ne voivat säästää kehittäjältä paljon työtä ja testaamista, sillä ohjelma ei välttämättä suoritu oikein, jos joitakin käytettyjä arvoja puuttuu.

Rivillä 10 luodaan olio "Samppa" aiemmin luodun mallin perusteella. Oliota luotaessa sille annetaan oletusarvojen tilalle uudet arvot. Id-arvoa ei yleensä anneta ollenkaan, kun luodaan uutta oliota, sillä Backbone luo aina uniikin id:n itsenäisesti, kun käytetään POST-metodia. Vaihtoehtoisesti jos id-arvo löytyy, Backbone osaa automaattisesti tulkita sen päivitykseksi ja käyttääkin PUT-metodia POST-metodin sijaan. [13.]

Näkymä

Näkymä on nimensä mukaisesti loppukäyttäjän näkymä eli ohjelman käyttöliittymän määritelmä. Näkymässä määritellään mallin tietojen, kuten kuvien tai tekstisisällön, esitystapa. JavaScript-MVC-arkkitehtuureja käytettäessä näkymä muodostuu HTML-tiedostosta, jossa tietoja esitellään lausekkeiden avulla. Koodiesimerkissä 4 luodaan näkymä.

```
1  var MyView = Backbone.View.extend({
2      template: $('#my-view-template').html(),
3
4      initialize: function() {
5          this.render()
6      },
7
8      render: function() {
9          var template = _.template(this.template);
10
11         var data = this.model.toJSON();
12         var html = template(data);
13
14         this.$el.html(html);
15     }
16 });
```

Koodiesimerkki 4. Näkymä Backbone-arkkitehtuurilla.

Koodiesimerkkissä 4 ensimmäisellä rivillä luodaan uusi näkymä, "MyView". Seuraavaksi näkymälle kerrotaan template eli mallinne, jota sen tulee käyttää renderöinnissä. Mallinne on pääosin HTML-merkintäkieltä, mutta useimmiten se sisältää myös muuttujia, joita voidaan muuttaa dynaamisesti. Tämän työn esimerkeissä käytetään Underscore.js-nimistä mallinnekirjastoa, joka ei ole ainoa tarjolla oleva liitännäinen tähän tarkoitukseen. Underscore-kirjaston käytöstä Backbonen kanssa on tullut kehittäjien keskuudessa tapa, vaikka Backbone ei ota kantaa siihen, mitä mallinnekirjastoa sen kanssa käytetään. Ember.js-MVC-arkkitehtuurin kanssa kehittäjillä on tapana käyttää Underscoren sijaan Handlebars-mallinnekirjastoa.

Rivillä 4 luodaan "initialize"-funktio, jota kutsutaan aina ensimmäisenä, kun näkymä luodaan. Tässä tapauksessa initialize kutsuu vain "render"-funktiota, joka kirjoittaa datan sivustolle mallinteen määrittelemällä tavalla. Rivillä yhdeksän mallinne sidotaan näkymään Underscore.js:n avulla. Underscore.js on JavaScript-kirjasto, joka helpottaa datan siirtämistä näkymiin. Koodiesimerkissä 5 luodaan mallinne Underscore.js:n avulla. Rivillä 11 data-arvoon sidotaan koodiesimerkkissä 1 luotu malli ja muutetaan se JSON-muotoon. Tämän jälkeen luodaan HTML-koodi mallinteen ja mallista saadun datan avulla. Lopuksi rivillä 14 HTML-koodi viedään DOM:iin käyttäjän nähtäväksi. [14.]

```
<script type="text/template" id="my-view-template">

        <p>First name: <%= firstName %> </p>
        <p>Last name: <%= lastName %> </p>
        <p>Email: <%= email %> </p>

</script>
```

Koodiesimerkki 5. Underscore.js-mallinne Person-mallille.

Mallinne kirjoitetaan script-tagien sisälle, ja sen tyypiksi määritellään "text/template", jolloin JavaScript tietää jättää sen suorittamatta ennen oikeaa hetkeä. Lisäksi mallinteelle annetaan id-arvo, joka on sama kuin aiemmin "myView"-näkymässä template-muuttujalle jQueryn avulla annettu osoitin. Underscore-mallinteita käytettäessä koodia kirjoitetaan "<%"- ja "%>"-tagien väliin. Tässä tapauksessa avaavan tagin yhteydessä esitetään myös yhtä kuin -merkki, joka tarkoittaa sitä, että muuttuja tulostetaan. [15.]

Käsittelijä

Käsittelijässä määritellään, kuinka käyttäjän tekoihin, kuten klikkauksiin, reagoidaan. Yleensä käyttäjän toimintoihin reagoidaan muuttamalla näkymää tai muuttamalla mallin tilaa. Tällainen tilamuutos voisi olla esimerkiksi tekstikentällä, joka on oletusarvoltaan passiivinen, mutta kun siihen klikataan, se tulee aktiiviseksi ja tekstiä voidaan muuttaa.

Kuten aiemmin mainittiin, MVC-arkkitehtuuri ei välttämättä noudata täysin perinteistä mallia ja käsittelijä on voitu jättää pois. Backbone-arkkitehtuurissa useimmat käsittelijän tehtävistä on hoidettu näkymissä sekä reitittimessä, joka hallitsee sovelluksen tilaa URL-osoitteen avulla. Esimerkiksi mallin muuttuessa siitä saadaan tieto näkymässä ja tällöin mallia näyttäneet kohdat ohjelmassa renderöidään uudestaan. Myös klikkaukset ja muut tapahtumat selaimessa hallitaan ja käsitellään näkymissä.

Backbonessa itsessään ei varsinaista käsittelijän ominaisuutta ole, mutta haluttaessa sellainen voidaan helposti tuoda käyttöön. Marionette.js on Backboneen lisättävissä oleva kirjasto, joka tuo Backbonesta puuttuvia ominaisuuksia kehittäjän käyttöön. [16.]

2.6 Palvelinpuolen JavaScript

JavaScriptin kehityksen myötä yhä enemmän JavaScript-pohjaisia ratkaisuja ilmestyy käytettäviksi myös palvelinpuolelle. Niistä kaikista selvästi pisimmälle viety on Node.js-kirjasto. Nod e.js:n perusajatuksena on viedä jo selaimesta tuttu asynkroninen kieli käyttöön myös palvelinympäristöön. Etu on itsestäänselvä: kahden tai useamman ohjelmointikielen sijaan kehittäjä voi kirjoittaa pelkkää JavaScriptiä.

Ymmärtääkseen Nodea täytyy ymmärtää, miten Node.js suorittaa koodin. Kuvitellaan tilanne, jossa halutaan ladata tietokannasta paljon resursseja, joiden lataaminen voi kestää useita sekunteja. Tavallisesti JavaScript jäisi odottamaan, että halutut resurssit on ladattu, ja etenisi vasta sitten koodissa. Välttääkseen tätä Node.js mahdollistaa asynkroniset funktioiden kutsumiset eikä jumiudu pelkästään yhden suorittamiseen. Sen sijaan, että Node.js odottaisi tietojen latautumista tietokannasta, se jatkaa koodin suorittamista. Kun kaikki on ladattu tietokannasta, Node.js suorittaa tietokantahaun sisällä määritetyt tehtävät, ja kun kaikki on suoritettu, se palaa silmukkaan odottamaan uusia tehtäviä. Sillä välin, kun pyyntöön odotetaan vastausta, Node.js ei varaa dedikoitua prosessia käyttöönsä vaan aktivoituu vasta, kun tietokanta palauttaa vastauksen hakupyyntöön.

Nodejs-palvelin voi siis käsitellä useita yhtäaikaisia hakupyyntöjä yhdessä prosessissa, kun perinteinen PHP-toteutus aloittaisi uuden oman prosessin jokaisen hakupyynnön kohdalla. Samalla JavaScriptin ja Node.js:n heikkous on, että se saa käyttöönsä vain yhden prosessorin ytimistä, kun PHP-toteutus voi tarvittaessa käyttää kaikkia.

Yksinkertaisen HTTP-pavelimen toteutus Node.js:llä onnistuu vain muutaman koodirivin avulla. Koodiesimerkissä 6 ensimmäinen koodirivi hakee käyttöön Node.js:n mukana tulevan http-moduulin ja asettaa sen http-muuttujaan. Seuraavaksi kutsutaan http-moduulin tarjoamaa createServer-funktiota. Se palauttaa olion, jolla on metodi nimeltä "listen", joka puolestaan tarkoittaa palvelimen kuuntelemaa porttinumeroa. Esimerkissä tämä porttinumero on asetettu 1337:ksi. Kun palvelin käynnistetään paikallisesti, palvelimen tarjoama sisältö löytyy osoitteesta localhost:1337.

```
1  var http = require('http');
2
3  http.createServer(function (req, res) {
4    res.writeHead(200, {'Content-Type': 'text/plain'});
5    res.end('Hello World\n');
6  }).listen(1337, '127.0.0.1');
7
8  console.log('Server running at http://127.0.0.1:1337/');
```

Koodiesimerkki 6. Node.js:lla toteutettu Hello World -web-palvelin.

CreateServer-funktion sisällä kutsutaan anonyymiä funktiota, joka asettaa HTTP-vastaukseen sisällön. Vastauksen writeHead-funktio palauttaa selaimelle HTTP-statuksen 200, joka tarkoittaa, että kaikki meni kuten pitikin. Lisäksi se kertoo selaimelle, että tuleva sisältö on tekstimuotoista eikä esimerkiksi JSON-dataa. Seuraavaksi vastauksen end-fuktioon kirjoitetaan varsinainen tarjoiltava tekstisisältö. [17.]

3 Esimerkkisovellus

3.1 Sovelluksen suunnittelu

Insinöörityön osana luotiin skaalautuva, helposti ylläpidettävissä oleva yhden sivun sovellus aiemmin esitellyillä teknologioilla. Sovellus on yksinkertainen, mutta kaikkia osa-alueen komponentteja käytetään hyödyksi. Vankkojen perustuksien rinnalle rakennetaan RESTful-Node.js-palvelin ja lopuksi käyttäjälle näkyvät komponentit Backbonella ja sen alakirjastolla Marionetellä. Palvelimen ja käyttöliittymäkomponenttien välillä siirretään tietoa JSON-muodossa, joka tallennetaan MongoDB-tietokantaan. Kaikki komponentit rakennetaan siis JavaScriptin avulla.

Sovellus on työnimeltään "Creature Card Deck", ja sen avulla voi luoda erilaisista otuksista kortteja. Yhdellä otuksella on erilaisia ominaisuuksia, jotka määritellään otusta luotaessa. Ominaisuuksia ovat nimi, elementti, kuva, hyökkäystaito ja puolustustaito. Näiden lisäksi yksi otus saa aina uniikin tunnisteen id:n, jonka avulla tietoja voidaan hakea pavelimelta. Jotta sovelluksen toteuttaminen olisi helpompaa, aluksi kannattaa miettiä hieman, mitä HTTP-pyyntöjä tarvitaan sovelluksen toteuttamiseen. Taulukossa 5 on avattu sovelluksessa tarvittavia HTTP-pyyntöjä. Palvelimen tulee palauttaa tarvittaessa yhden uniikin otuksen tietoja ja pystyä muuttamaan niitä. Samalla sen pitää pystyä palauttamaan tiedot kaikista otuksista samanaikaisesti.

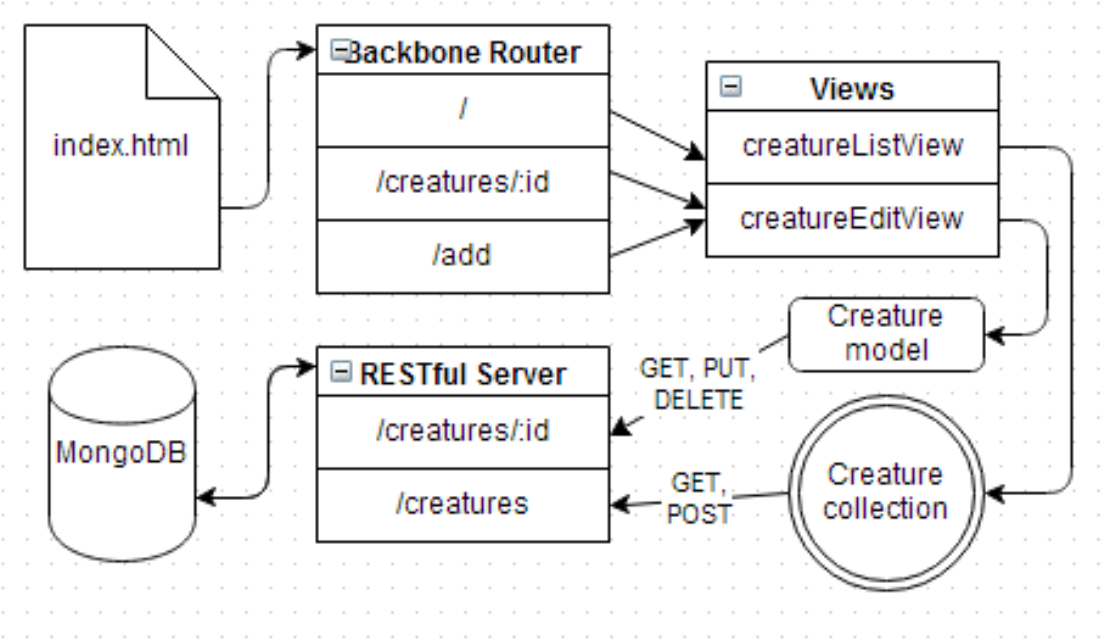
Taulukko 5. Esimerkkisovelluksessa käytetyt HTTP-pyynnöt selitettynä.

Resurssi-URI	**HTTP-metodi**	**Tapahtuma**
www.example.com/creatures	HTTP GET	Palauttaa kaikkien otuksien tiedot.
www.example.com/creatures/:id	HTTP GET	Palauttaa yhden id:llä tunnistetun otuksen tiedot.
www.example.com/creatures	HTTP POST	Luo uuden otuksen.
www.example.com/creatures/:id	HTTP PUT	Päivittää yhden id:llä tunnistetun otuksen tietoja.
www.example.com/creatures/:id	HTTP DELETE	Poistaa yhden id:llä tunnistetun otuksen.

Nyt tiedetään, mitä dataa halutaan REST-palvelimen tarjoavat mitäkin resurssi-URI:a vastaan. Seuraavaksi voidaan miettiä, mitä ohjelmistokomponentteja tarvitaan toimivan käyttöliittymän saavuttamiseksi. Tarvitaan näkymät kaikkien otuksien listaamiseen, yhden otuksen muokkaamiseen sekä uuden otuksen luomiseen. Koska yhden otuksen muokkaamiseen ja uuden luomiseen tarvittavat näkymät ovat hyvin samanlaiset, voidaan käyttää vain yhtä näkymää molempien kanssa.

Sovellus käynnistyy päänäkymään, jossa näkyvät kaikki otukset. Näkymästä pääsee eteenpäin kahdella tapaa. Näkymästä voidaan valita jokin valmis olio, jolloin Backbone Router vie käyttäjän osoitteeseen www.example.com/creatures/id, jossa id on otuksen uniikki tunniste, tai käyttäjä voi painaa näppäintä, jolla luodaan uusi olio, jolloin hän päätyy osoitteeseen www.example.com/add. Molemmat toiminnot vievät käyttäjän samaan näkymään, jossa ovat tekstisyötteet, joihin tulee kirjoittaa otuksen tiedot. Ero on siinä, että kun käyttäjä painaa jo olemassa olevaa otusta, näkymän tekstikenttiin haetaan otuksen tiedot tietokannasta valmiiksi. Uutta otusta luotaessa kaikki tekstikentät ovat tyhjiä tai niissä ovat oletusarvo täytettynä.

Samasta näkymästä voi siis luoda uuden otuksen tai sen tietoja voi päivittää tai poistaa otuksen kokonaan. Tieto tapahtumasta menee HTTP-metodin avulla Node.js-palvelimelle, joka tekee muutokset MongoDB-tietokantaan. Käyttäjän palatessa etusivulle sivusto lähettää HTTP-pyynnön ja saa vastauksena päivitetyt otukset. Kuvassa 5 kuvataan sovelluksen toteusta. Siitä on nähtävissä HTTP-pyyntöjen ja datan kulku tietokannasta käyttöliittymälle.



Kuva 5. Esimerkkisovelluksen ohjelmistokomponentit ja datan kulku.

Sovelluksen tulee olla helposti ylläpidettävissä, joten tarvitaan selkeä kansiorakenne, josta kaikki tarvittavat tiedostot löytyvät tarvittaessa nopeasti. Lisäksi Backbonen ansiosta JavaScript-tiedostot noudattavat tiettyä rakennetta, joka helpottaa oikean kohdan löytämistä koodista. Ylläpidättävyyden lisäksi sovelluksen täytyy olla myös minimoitu eli mahdollisimman kevyt ja nopea. Sovelluksen optimointi kannattaa aina. Pätevän "build"-skriptin tekeminee helpottaa valmiin koodin siirtämistä tuotantopalvelimelle. Ideaalisesti tuotanpalvelin tekisi vain yhden tai muutamia HTTP-hakuja. Lisäksi haettavien tiedostojen tulisi olla kompressoituja, eli pitkät muuttujien nimet on muutettu lyhyisiin, välilyönnit poistettu ja koodista on tehty ihmiselle vaikealukuista, mutta tietokoneelle tehokasta luettavaa. JavaScriptin lisäksi myös tyylitiedostot ja kuvat voidaan optimoida yhdellä skriptin suorittamisella.

3.2 Sovelluksen perustukset

Sovelluksen perustuksien rakentamiseen tarvittiin Node.js ja Node.js:n paketinahallintatyökalu Npm, jonka avulla asennettiin aluksi Require.js- ja Bower-kirjastot. Bower auttaa sovelluksen tarvitsemien kirjastojen asentamisessa ja niiden hallitsemissa. Boweriin voidaan määritellä haluttu versio kaikista kirjastoista, jolloin vältytään ikäviltä yllätyksiltä, jos kirjaston päivittyessä uudempaan versioon jonkin käytetyn komponentin tuki on poistettu tai kaikki kirjastot eivät vain toimi enää yhdessä.

Asennettiin myös Require.js, joka on AMD (Asynchronous Module Definition), joka on JavaScript-rajapinta modulien ja niiden riippuvuuksien määrittelylle, jotta ne voidaan ladata asynkronisesti vain, kun niitä tarvitaan. Asynkronisuus on ei-reaaliaikaista kommunikointia, josta oikean elämän esimerkki on sähköposti. Kun lähetetään sähköpostia jollekulle, pystytään jatkamaan muita toimia ilman välitöntä vastausta ja palaamaan vastaukseen myöhemmin, kun se saadaan. Samalla tavoin JavaScript-koodi ei suoritu rivi kerrallaan ylhäältä alas vaan jatkaa suorittamista, vaikka johonkin tehtävään vastauksen saaminen kestäisi. [18.]

Kun Require.js ja Bower oli asennettu, voitiin aloittaa kansiorakenteen työstäminen. Selkeä kansiorakenne auttaa oikean tiedoston löytymisessä nopeasti. Ensin luotiin Project-kansio, jonka alle app-kansio, joka on sovelluksen asiakaspuolen toteutuksen pääkansio ja sisältää kaikki sen osat. Sen alle luotiin kansiot "css" tyylitiedostoja varten sekä "js" JavaScript-tiedostoille. Lisäksi tehtiin HTML-tiedosto "index.html", jonka tehtävä on näyttää sisältö Backbone-arkkitehtuurin avulla.

Seuraavaksi tehtiin js-kansion sisälle uusi kansio "lib", jonne Bowerin halutaan asentavan tarvittavat kirjastot. Bower kuitenkin asentaa kirjastot muualle kuin lib-kansioon oletusarvoisesti, joten Project-kansion juureen luotiin tiedosto nimeltä ".bowerrc", joka on JSON-tiedosto, jonka avulla määritellään haluttu asennuspolku kirjastoille [19]. Tiedostoon kirjoitettiin, kuten koodiesimerkkissä 7 on esitetty.

```
1  {
2          "directory": "app/js/lib"
3  }
```

Koodiesimerkki 7. Bowerin konfiguraatiotiedosto määrittelee, minne kirjastot asennetaan.

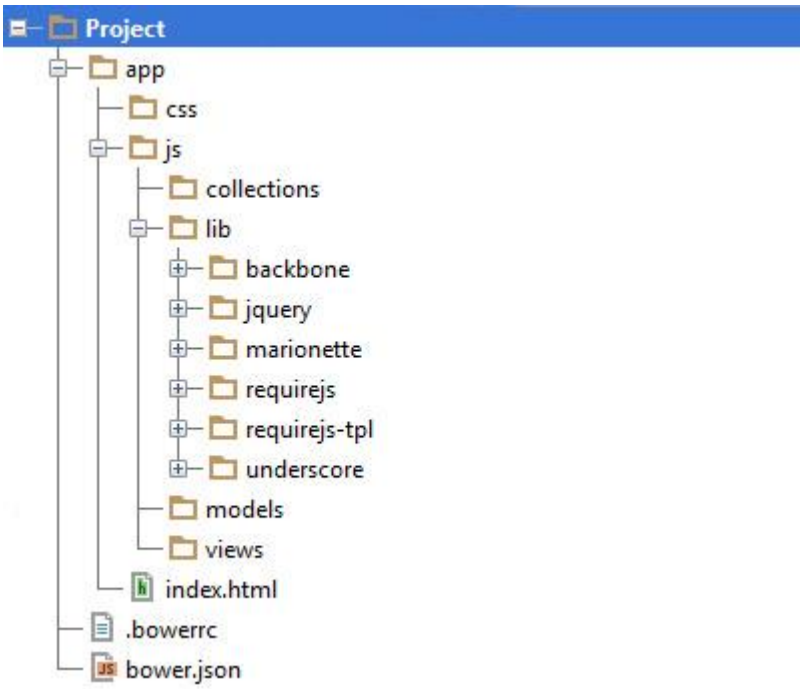
JQuery asennettiin haluttuun lib-kansioon komentorivin avulla navigoimalla sillä projektin juureen ja suorittamalla komento "bower install jquery". Nyt voitaisiin helposti asentaa samalla tavalla kaikki tarvittavat kirjastot ja riippuvuudet, mutta sen sijaan käytettiin toista Bowerin ominaisuutta hallita kirjastoja ja luotiin project-kansioon jälleen uusi JSON-tiedosto "bower.json", joka on koodiesimerkissä 8.

```
{
    "name": "Creature Cards",
    "version": "0.0.1",
    "dependencies": {
        "jquery": null,
        "underscore": null,
        "backbone": null,
        "marionette": null,
        "requirejs": null,
        "requirejs-tpl": null
    }
}
```

Koodiesimerkki 8. Bower-json, jossa määritellään sovelluksen kirjastoriippuvuudet.

Bower.json-tiedostoon kirjoitetaan ensimmäisenä sovelluksen nimi. Seuraavaksi annetaan sovelluksen versionumero ja sitten sovelluksen kirjastoriippuvuudet. Kirjaston nimen jälkeen tulisi antaa haluttu versionumero kirjastosta, mutta arvoksi voi antaa myös null, jolloin Bower käyttää viimeistä vakaata versiota kirjastosta. Kun Bower oli konfiguroitu dokumentaation mukaisesti, voitiin ajaa komentorivillä käsky "bower install". Käskyn saadessaan Bower käy läpi bower.json-tiedoston ja asentaa riippuvuudet sen käskemällä tavalla. Etuna Bowerin käytössä on, että tarvittavia kirjastoja ei tarvitse liikutella projektin mukana vaan ne ovat helposti asennettavissa yhdellä komennolla. Useimmiten isossa projektissa tulee myös pitää kirjastojen versiointi tiukasti hallussa, sillä jokin sovellus saattaa toimia eri tavoilla eri kirjastojen kanssa, ja jos sovellus toimii eri lailla jokaisella sen kehittäjällä, on hankala kehittää yhtenäistä tuotetta.

Js-kansion alle luotiin vielä kansiot "models", "collections" ja "views". Näihin kansioihin rakennetaan myöhemmin sovelluksen toiminnallisuus alusta loppuun. Kuvassa 6 on viimeistelty MV*-arkkitehtuurimallin mukainen kansiorakenne.



Kuva 6. Projektin viimeistelty kansiorakenne, kun Bower on täysin konfiguroitu.

3.3 Sovelluksen palvelin

Tässä esimerkissä pystytettiin Node.js-palvelin project-kansion juureen. Se olisi ollut mahdollista eriyttää omaan kansioon, ja se olisi ollut myös suotavaa, jos palvelintoteutus olisi monimutkaisempi, mutta tämä toteutus on äärimmäisen yksinkertainen.

Palvelimen luominen aloitettiin tekemällä project-kansion juureen tiedostot server.js ja package.json. Package.json on aivan samanlainen kuin aiemmin luotu bower.json, mutta kun Bower hallitsee asiakaspuolen kirjastoriippuvuuksia, package.json hallitsee Node-palvelimella tarvittavia kirjastoja. Koodiesimerkissä 9 lisättiin package.json-tiedostoon riippuvuudeksi Express-niminen kirjasto, joka on pieni Node.js-arkkitehtuuri web-sovelluksille. Lisäksi tarvittiin rajapinta MongoDB-tietokannalle [20], ja sovelluksessa käytettiin Mongoose-kirjastoa tähän tarkoitukseen.

```
{
    "name": "creature-cards",
    "version": "0.0.0",
    "dependencies": {
        "express": "3.x.x",
        "mongoose": ">= 3.5.x"
    }
}
```

Koodiesimerkki 9. Package.json-tiedosto, jossa kirjastoriippuvuuksina Express ja Mongoose.

Kun package.json on paikoillaan project-kansion juuressa, komentorivillä voitiin suorittaa "npm install" samalla tavoin kuin Bowerin kanssa. Npm, joka on Node.js:n oma pakettienhallintatyökalu, lukee package.json:sta tarvitut riippuvuudet ja lataa ne project-kansion sisälle omaan node_modules-hakemistoon. Nyt tarvittavat kirjastoriippuvuudet ovat paikallaan ja itse palvelimen tekeminen voidaan aloittaa.

Koodiesimerkkissä 10 luotiin server.js-tiedosto, jossa haetaan käyttöön Express- ja Path-kirjastojen lisäksi oma creatures-moduuli. Koodiesimerkissä 11 tehtiin creatures-moduuli, jossa otetaan käyttöön toistaiseksi käyttämättötön kirjasto, Mongoose. Sen avulla avataan yhteys MongoDB-tietokantaan ja luodaan otusskeema eli tietomalli, jota noudatetaan, kun otuksien tietoja haetaan tai tallennetaan tietokantaan. Skeema-tyyppejä on erilaisia, kuten numero, totuusarvo, päivämäärä tai jono, mutta sovelluksessa käytetaan vain merkkijonoa eli String-tyyppiä otusskeemassa. [21.]

```
var express = require('express'),
    creatures = require('./routes/creatures'),
    path = require('path');

var app = express();

app.configure(function () {
    app.use(express.bodyParser());
    app.use(function(req, res, next) {
        res.header("Access-Control-Allow-Origin", "*");
        res.header("Access-Control-Allow-Headers", "Cache-Control,
        Pragma, Origin, Authorization, Content-Type, X-Requested-With");
        res.header("Access-Control-Allow-Methods", "POST, GET,
        PUT, DELETE, OPTIONS");
        next();
    });
    app.use(express.static(path.join(__dirname, 'app')));
});

app.get('/creatures', creatures.findAll);
app.get('/creatures/:id', creatures.findById);
app.post('/creatures', creatures.addCreature);
app.put('/creatures/:id', creatures.updateCreature);
app.delete('/creatures/:id', creatures.deleteCreature);

app.listen(9999);
console.log('Listening on port 9999...');
```

Koodiesimerkki 10. Node.js-palvelin server.js-tiedostossa

Seuraavaksi luodaan server.js-tiedostoon sovelluksen käyttämät REST-metodit. Aloitetaan funktiot aina "exports.funktionNimi", jolloin funktiot ovat Exports-kirjaston avulla käytettävissä myös server.js-tiedostossa, jossa ne yhdistetään URL-osoitteen avulla kutsumaan oikeata funktiota.

Jokaiselle funktiolle annetaan myös kaksi vakiota: "req" eli request, joka tarkoittaa HTTP-pyyntöä, ja "res" eli response, joka tarkoittaa HTTP-vastausta. Niiden avulla saadaan tietoa sisääntulevasta pyynnöstä sekä kirjoitettua vastaus pyyntöön. Lisäksi jokaisessa funktiossa tehdään virheenhallintaa, jolloin virheen tapahtuessa Node.js kirjoittaa lokiin tapahtumasta selvityksen.

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/creaturedb');

var db = mongoose.connection;
db.on('error', console.error.bind(console, 'mongoose connection error:'));
db.once('open', function callback() {
    console.log('mongoose connection open');
});

var CreatureSchema = mongoose.Schema({
    name: String,
    element: String,
    img: String,
    attack: String,
    defense: String
});

var Creature = mongoose.model('Creature', CreatureSchema);
```

Koodiesimerkki 11. MongoDB-yhteyden avaaminen ja Otuksen skeema.

Koodiesimerkkissä 12 on kaikkien otuksien sekä yhden otuksen haku HTTP GET-metodilla. Ensimmäinen funktio hakee tietokannasta kaikki Creatures-skeemaa käyttävät oliot ja palauttaa vastauksena pyyntöön. Toinen etsii otuksen, jolta löytyy URL-osoitteessa määritelty uniikki id-arvo, ja palauttaa löytyneen otuksen JSON-objektina.

```
exports.findAll =  function(req, res) {
    return Creature.find(function(err, creatures) {
        if (!err) {
            return res.send(creatures);
        }else{
            return console.log(err);
        }
    });
};

exports.findById = function(req, res) {
    return Creature.findById(req.params.id, function(err, creature) {
        if (!err) {
            return res.send(creature);
        }else{
            return console.log(err);
        }
    });
};
```

Koodiesimerkki 12. HTTP GET -metodien käsittely Node.js:n avulla.

Koodiesimerkissä 13 tehtiin seuraava "addCreature"-funktio, joka luo uuden objektin Creature-skeeman avulla, johon se tallentaa otuksen tiedot HTTP-pyynnön mukana tulleen datan avulla. Palautuksessa kutsutaan save-funktiota, joka luo uuden otuksen

tietokantaan ja palauttaa käyttäjälle uuden otuksen objektina. Huomattavaa on, että missään vaiheessa koodissa ei tarvitse antaa uudelle otukselle id-arvoa. MongoDB tietää luoda otukselle uuden id:n, kun pyynnön mukana ei tullut sellaista.

Otuksen päivitys toimii hyvin samalla tavoin kuin uuden luominen. Tietääkseen, mitä otusta tulee päivittää, Mongoose etsii "findById"-funktion avulla tietokannasta otuksen. Otuksen löytäminen onnistuu, sillä uniikki id-arvo tulee pyynnön mukana kohdassa "req.params.id". Kun Mongoose on löytänyt oikean otuksen, se kirjoittaa sen tietojen päälle HTTP-pyynnön mukana tulleet tiedot. Tiedot päivittyvät tietokantaan, kun save-funktiota kutsutaan.

```
exports.addCreature = function(req, res) {
    var creature = new Creature({
        name: req.body.name,
        element: req.body.element,
        img: req.body.img,
        attack: req.body.attack,
        defense: req.body.defense
    });
    return creature.save(function (err) {
        if(!err) {
            console.log('creature created');
            return res.send(creature);
        }else{
            return console.log(err);
        }
    });
};

exports.updateCreature = function(req, res) {
    return Creature.findById(req.params.id, function(err, creature) {
        creature.name = req.body.name;
        creature.element = req.body.element;
        creature.img = req.body.img;
        creature.attack = req.body.attack;
        creature.defense = req.body.defense;

        return creature.save(function(err) {
            if(!err) {
                console.log("creature updated");
                return res.send(creature);
            }else{
                console.log(err);
            }
        });
    });
};
```

Koodiesimerkki 13. Otuksen lisäämisen ja päivittämisen käsittely creatures.js-tiedostossa.

Koodiesimerkissä 14 tehtiin vielä puuttuva funktio otuksen poistamiselle. Mongoose etsii jälleen tietyn otuksen "findById"-funktion avulla ja kutsuu sille remove-funktiota, joka poistaa sen tietokannasta. Jos kaikki onnistuu eikä virheilmoitusta saada, ohjelma palauttaa tyhjän creature-objektin selaimelle, jolloin se tietää jatkaa muita tehtäviä.

```
1  exports.deleteCreature = function(req, res) {
2      return Creature.findById(req.params.id, function(err, creature) {
3          return creature.remove(function(err) {
4              if(!err){
5                  console.log("creature removed");
6                  return res.send(creature);
7              }else{
8                  console.log(err);
9              }
10         });
11     });
12 };
```

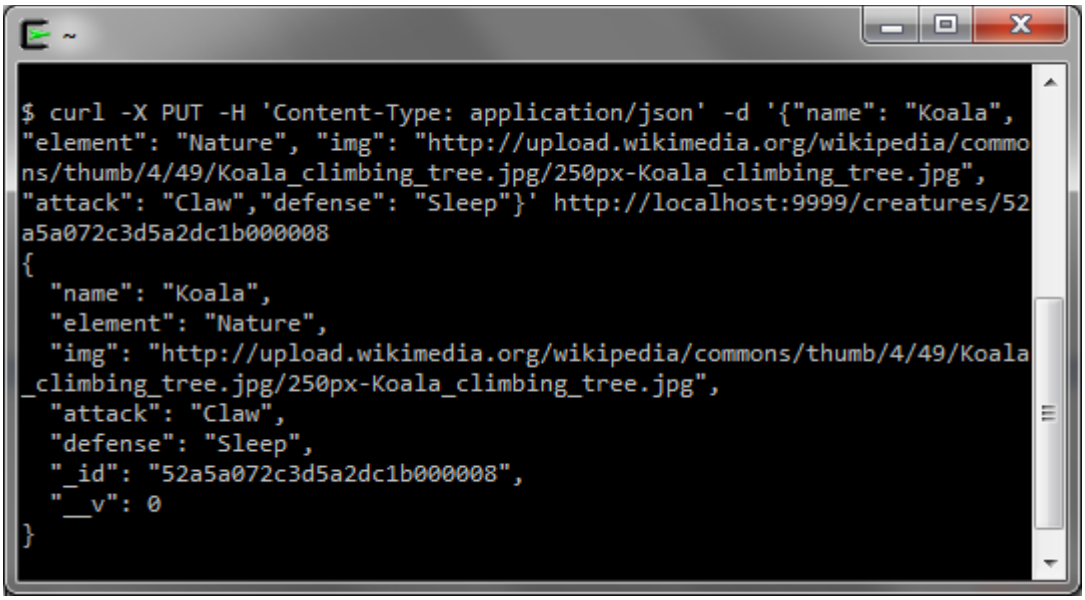
Koodiesimerkki 14. Olion poistaminen.

Kun oli saatu HTTP-komentojen käsittelyt valmiiksi, tehtiin REST-palvelimen nopea testaus curlilla ja selaimella osoitteessa http://localhost:9999/creature. Node-palvelin käynnistettiin kirjoittamalla komentoriville projektikansion juuressa "node server.js", jolloin palvelin käynnistyi ja ilmoitti kuuntelevansa porttia 9999. MongoDB:n tuli olla myös käynnissä, jotta voitiin kokeilla palvelimen ja tietokantayhteyden toimivuutta. Mikäli tietokantayhteys onnistui, komentorivillä näkyi teksti "mongoose connection open", jota kutsutaan koodiesimerkissä 11 rivillä 7.

Curlin avulla tietokantaan lisätään otus nimeltä Koala. Curl-skriptissä määritellään ensin käytettävä HTTP-metodi, joka lisättäessä uutta dataa on POST. Seuraavaksi HTTP Headeriin lisätään "-H"-merkinnän avulla datamuoto, jotta sovellus osaa käsitellä tietoa JSON-datana sekä itse data JSON-muodossa. Viimeisenä kirjoitetaan osoite, johon pyyntö tehdään. Kuvassa 7 on curl-komento kokonaisuudessaan.

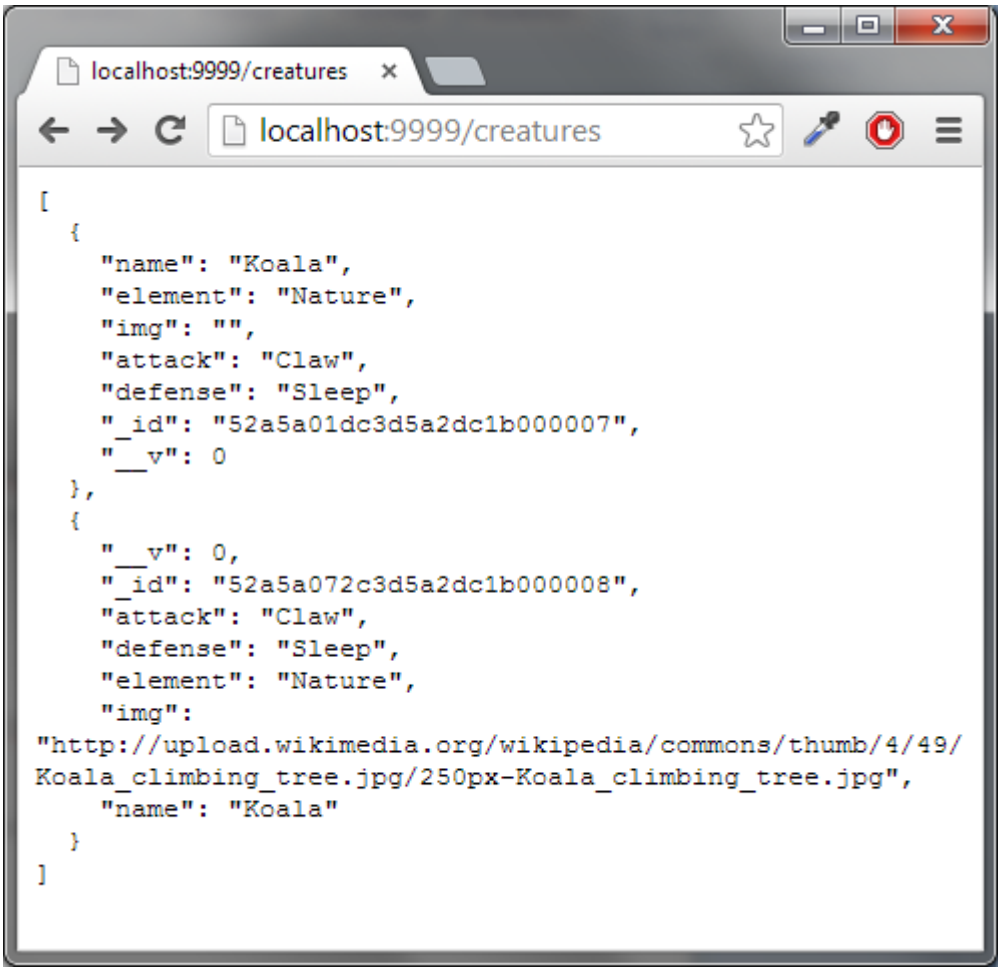
Kuva 7. Curlin avulla lisätään otus tietokantaan.

Komento voidaan suorittaa useita kertoja, ja jokaisella kerralla MongoDB antaa uniikin "_id":n otukselle. Kuvassa 7 lisättiin otus, jolla on kaikki muut tiedot paitsi kuva eli img-arvo. Kuvassa 8 päivitetään otukselle kuva HTTP PUT -metodin avulla. Tällä kertaa tehdään curlin avulla pyyntö eri osoitteeseen, jossa on mukana tietyn otuksen "_id"-arvo.



Kuva 8. Päivitetään otukselle kuva-arvo curlin avulla.

Luodut otukset saadaan helposti näkyviin selaimella menemällä osoitteeseen "localhost:9999/creatures", kuten kuvasta 9 nähdään. Aiemmin koodiesimerkeissä 10 ja 12 määriteltiin, että osoite tekee tietokantaan haun, joka listaa kaikki olemassa olevat otukset. Yksittäisen otuksen tiedot saataisiin lisäämällä osoitteen perään otuksen id-arvo.



Kuva 9. Kaikki otukset listattuna selaimessa JSON-muodossa.

Kuten kuvasta 9 nähdään, tietokannassa on kaksi Koala-nimistä oliota. Kaksoiskappaleet voidaan poistaa yksinkertaisella curl-komennolla: "curl -i -X DELETE http://localhost:9999/creatures/id", jossa id tulee korvata otuksen "_id"-arvolla. [22.]

3.4 Sovelluksen käyttöliittymä

Seuraavaksi luotiin asiakaspuolen käyttöliittymä Backbonen ja Marioneten avulla, joka käyttää tietojen hakemiseen juuri toteutettua REST-rajapintaa.

Sovellus toimii vain yhden html-tiedoston avulla, jonka sisältöä muutetaan dynaamisesti JavaScriptin avulla. Koodiesimerkkissä 15 akaisemmin toteuttettua html-tiedostoa työstettiin eteenpäin.

```
<!DOCTYPE html>
<html>
<head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
        <title>Creature Cards</title>
        <meta name="viewport" content="width=device-width">
        <link rel="stylesheet" type="text/css" media="all" href="css/main.css"/>
</head>
<body>
        <div id="header">
                <h1>Creature Cards</h1>
                <a href="#/add" id="add-button" class="btn btn-default">Add Creature</a>
        </div>
        <div id="content"></div>

        <script data-main="js/main" src="js/lib/requirejs/require.js"></script>
</body>
</html>
```

Koodiesimerkki 15. Index.html-tiedosto.

Tiedoston alussa head-tagin sisällä käytetään HTML5-metatageja. Niiden avulla asetetaan käytetty merkistö UTF-8:ksi, käsketään selaimen käyttää uusinta mahdollista versiota Internet Explorer -selaimesta ja mahdollistetaan Chrome-selaimella Chrome-kehyksen käyttö, joka on JavaScriptin suorittamista nopeuttava selainliitännäinen [23]. Viimeisessä meta-tagissa käsketään selaimen käyttää näkymän leveytenä laitteen näytön leveyttä. Jos sivustoa käytetään esimerkiksi puhelimella tai tabletilla, se skaalautuu laitteen näytön koon mukaisesti ja sivustosta saadaan käyttökelpoinen mobiililaitteille helposti tekemättä muita responsiivisia tyylimääritteitä. Meta-tagien lisäksi head-tagin alta löytyy CSS-tiedosto sivuston tyylimääritteitä varten.

Body-tagissa on kaksi eri div-elementtiä, joista header on täysin staattista sisältöä tarjoava. Siellä näytetään sivuston otsikko ja ankkurilinkki, joka vie käyttäjän lomakkeeseen, jolla voidaan lisätä uusia otuksia. Content-elementti puolestaan on index.html-tiedostossa tyhjä, mutta todellisuudessa JavaScript tuottaa sinne kaiken dynaamisen sisällön.

Tiedoston on script-tagi, joka käskee Require.js-kirjastoa lataamaan JavaScript-tiedoston polusta "js/main", johon se lisää .js-päätteen main-tiedostolle. Tämä toimii ainoana yhdyspisteenä index.html:n ja JavaScript-sovelluksen välillä. Reruire.js:n ansiosta main.js-tiedosto ladataan asynkronisesti. [18.]

Jokainen JavaScript-tiedosto sisältää require- tai define-funktion, kun tehdään sovellusta Require.js-kirjaston kanssa. Require-funktiota käytetään vain main.js-tiedostossa, jossa ladataan tarvittavat kirjastot käyttöön heti sovelluksen käynnistyessä. Define-funktiota käytetään kaikissa muissa tiedostoissa. Sen avulla määritellään kunkin tiedoston omat riippuvuudet, ja ne haetaan käyttöön funktion avulla. Jokaisen tiedoston lopussa myös palautetaan koko luotu objekti, esimerkiksi näkymä tai malli. Tämän tiedon muut tiedostot saavat käyttöönsä, kun ne määrittelevät tietyn tiedoston define-funktion sisällä. [18.]

Koodiesimerkissä 16 luotiin js-kansioon uusi main.js-tiedosto, jossa tehdään kolme asiaa. Siellä on Require.js-konfiguraatio, jossa sille asetetaan tarvittavien kirjastojen sijainnit lib-hakemistosta ja määritellään eri kirjastojen riippuvuudet, jotta Require.js osaa ladata ne oikeassa järjestyksessä virheiden välttämiseksi. Seuraavaksi haetaan määritellyt kirjastot käyttöön require-funktion avulla, joiden lisäksi haetaan sovellukselle oleelliset tiedostot app.js ja router.js samassa funktiossa.

```
require.config({
    paths: {
        "jquery": "lib/jquery/jquery",
        "underscore": "lib/underscore/underscore",
        "backbone": "lib/backbone/backbone",
        "marionette": "lib/marionette/lib/backbone.marionette",
        "tpl": "lib/requirejs-tpl/tpl"
    },
    shim: {
        underscore: {
            exports: '_'
        },
        backbone: {
            deps: ['jquery', 'underscore'],
            exports: 'Backbone'
        },
        marionette: {
            deps: ['jquery', 'underscore', 'backbone'],
            exports: 'Marionette'
        }
    }
});

require(["jquery", "underscore", "backbone", 'app', 'router'],
    function (jQuery, _, Backbone, App, AppRouter) {

        App.addInitializer(function(options) {
            App.router = new AppRouter(options);
            Backbone.history.start();
        });

        App.start();
        return App;

    }
);
```

Koodiesimerkki 16. Main.js-tiedosto, jossa tapahtuu Require.js:n konfiguraatio ja sovelluksen käynnistys.

App.start-funktio kutsuu app.js-tiedostoa, jossa sovellus luodaan ja se käynnistyy. App.js-tiedosto on esitelty koodiesimerkissä 17. Sovelluksen käynnistyessä se kutsuu main.js-tiedostosta addInitializers-funktiota, joka puolestaan kutsuu reititintä ja aloittaa Backbonen historian seuraamisen, jotta selaimen takaisin-nappi toimisi. App.js-tiedostossa määritellään myös sovelluksen alueet eli regionit. Niitä voi olla useita, mutta tässä sovelluksessa on vain yksi, nimeltään "mainRegion", joka on yhdistetty index.html-tiedostossa tyhjäksi jätettyyn content-elementtiin. Aina kun mainRegion-alueen sisältöä päivitetään, sovellus tietää päivittää content-nimisen HTML-elementin sisältöä.

```
define(["marionette"],function(Marionette){

    var App = new Marionette.Application();

    App.addRegions({
        mainRegion: "#content"
    });

    return App;
});
```

Koodiesimerkki 17. App.js-tiedosto, jossa sovellus luodaan.

Sovelluksen reititin sijaitsee router.js-tiedostossa. Nyt toteutetaan sovelluksen päänäkymä, joka listaa kaikki sovelluksen otukset yhteen näkymään. Reititin toimii siten, että se lukee selaimesta URL-osoitteen ja suorittaa sitä vastaavan funktion reitittimestä. Koodiesimerkissä 18 rivillä 7 ohjelma suorittaa home-funktion, kun osoite on tyhjä eli osoittaa sovelluksen juureen.

```
define(['backbone','views/creatureListView','app','collections/creatureCollection'],
    function(Backbone,CreatureListView,App,CreatureCollection) {

    var AppRouter = Backbone.Router.extend({

        routes: {
            "" : "home"
        },

        home: function() {
            this.creatureCollection = new CreatureCollection();
            var creatureListView = new CreatureListView({
                collection: this.creatureCollection
            });

            this.creatureCollection.fetch({
                success: function() {
                    App.mainRegion.show(creatureListView);
                }
            });
        }
    });
    return AppRouter;
});
```

Koodiesimerkki 18. Router.js eli sovelluksen reititin.

Home-funktiossa luodaan uusi kokoelma otuksia ja uusi näkymä "creatureListView", jonka parametrina annetaan juuri luotu otuksien kokoelma. Kaikki otukset haetaan fetch-funktion avulla, joka suorittaa HTTP-pyynnön tietokantaan. Jos HTTP-pyyntö onnistuu otukset sisältävä näkymä sijoitetaan alueeseen "mainRegion", joka vastaa DOM:ssa content-elementtiä. Jotta kaikki toimisi, tarvitaan vielä JavaScript-tiedostot

näkymälle, mallille ja kokoelmalle. Tehdään "js/views"-polkuun tiedostot "creatureView.js" ja "creatureListView.js". Seuraavaksi tehdään "js/collection"-hakemistoon "creatureCollection.js"-tiedosto ja "js/models"-hakemistoon "creatureModel.js"-tiedosto.

Koodiesimerkissä 19 otus-mallille annetaan arvot "urlRoot" ja "idAttribute", jotka molemmat liittyvät tiedon hakemiseen. UrlRootissa määritellään osoite, johon HTTP-pyyntö tehdään, kun halutaan hakea malliin liittyvää tietoa. Id-arvo pitää määrittää erikseen, jotta MongoDB-tietokanta osaa etsiä tietoa oikealla arvolla. Backbonessa vakioarvo olisi "id", mutta MongoDB-tietokanta käyttää arvoa "_id", joten se asetaan käsin vastaamaan MongoDB-toteutusta.

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

    var CreatureModel = Backbone.Model.extend({
        urlRoot: '/creatures',
        idAttribute: '_id'
    });

    return CreatureModel;
});
```

Koodiesimerkki 19. Otuksen malli tiedostossa creatureModel.js.

Koodiesimerkissä 20 luodussa kokoelmassa on paljon samaa kuin mallissa, mutta tällä kertaa tulee myös määrittää, mitä mallia kokoelman tulee käyttää. Määritetään kokoelman "model"-avaimelle arvoksi juuri luodun CreatureModelin URL-osoite, joka toimittaa kokoelmassa samaa asiaa kuin mallissa. [24.]

```
define(["backbone","models/creatureModel"],
    function(Backbone,CreatureModel) {

    var CreatureCollection = Backbone.Collection.extend({
        model: CreatureModel,
        url: '/creatures'
    });

    return CreatureCollection;
});
```

Koodiesimerkki 20. Otuksien kokoelma tiedostossa creatureCollection.js.

Sekä kokoelma että malli vaativat itselleen omat näkymät. Näkymiä ei suoraan liitetä tässä esimerkissä toisiinsa, vaan reititin asettaa niille sisällön. Backbonesta ei löydy

erilaisia näkymiä, joten käytetään Marioneten näkymiä. Yhden otuksen näkymälle käytetään koodiesimerkkissä 21 esiteltyä Marionette.ItemView'ta ja kokoelman näkymälle koodiesimerkki 22 käytetään Marionette.CollectionView'ta. Käyttämällä Marioneten tarjoamia näkymiä selvitään huomattavasti vähemmällä koodilla ja vältetään turhaa toistoa [25]. Lisäksi Marionette-kirjasto auttaa tapahtumien käsittelyssä ja välttämään niin sanottuja zombinäkymiä, jotka johtuvat vapauttamattomista tapahtumista ja näkymistä, jotka jäävät taustalle varaamaan muistia turhaan käyttöön [26].

Vakioasetuksilla näkymä luo aina DOM-puuhun div-elementin, jota voidaan haluttaessa muuttaa "tagName"-parametrin avulla. Voidaan esimerkiksi antaa kokoelman näkymälle tagiksi ul ja mallille li, jolloin saataisiin aikaiseksi perinteinen lista kaikista malleista.

```
1  define(["marionette","app","tpl!../../templates/creature-view.tpl"],
2      function(Marionette,App,CreatureViewTemplate) {
3
4      var CreatureView = Marionette.ItemView.extend({
5          template: CreatureViewTemplate,
6          className: 'creature-item'
7      });
8
9      return CreatureView;
10 });
```

Koodiesimerkki 21. Otuksen näkymä tiedostossa creatureView.js.

Näkymät tarvitsevat itselleen templaten eli Underscore-mallinteen, joka on yleensä HTML-merkintäkieltä. Sen avulla voidaan määritellä tarkalleen, missä kohdassa mikäkin mallin arvo näytetään tai jätetään näyttämättä. Tehtiin uusi kansio "templates" app-hakemistoon ja sen sisälle yksinkertainen template-tiedosto "creature-view.tpl", jotta nähdään, tulostaako sovellus halutut arvot ruudulle. Mallinne on esitelty koodiesimerkissä 22.

```
1  name: <%=name%> <br>
2  element: <%=element%>
```

Koodiesimerkki 22. Yksinkertainen Underscore-mallinne creature-view.tpl.

Koodiesimerkissä 23 kokoelman näkymälle annetaan itemView'lle arvoksi aikaisin tehty CreatureView, jolloin se renderöi näkymään niin monta kertaa CreatureView-näkymän

kuin otus-malleja löytyy. Myös kokoelman näkymälle voitaisiin asettaa samaan tapaan template kuin yhden otuksen näkymälle, jos lisämuotoilua haluttaisiin toteuttaa.

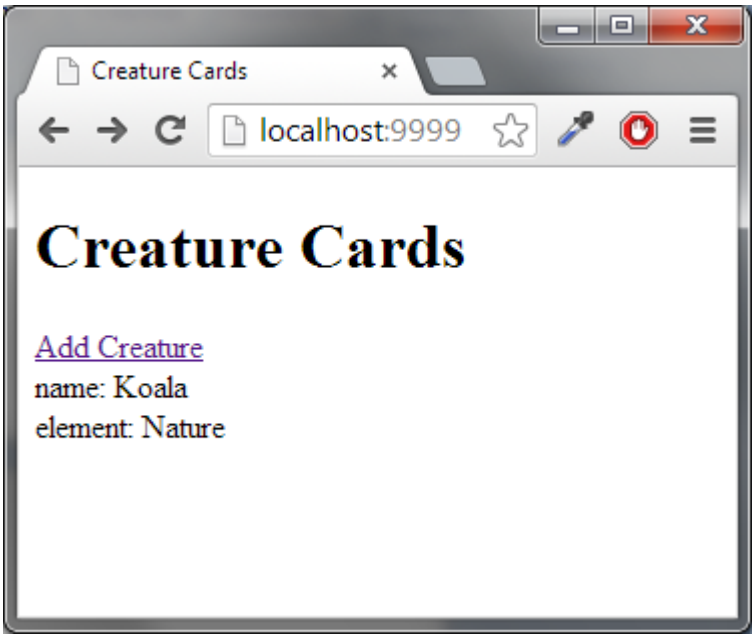
```
define(["marionette","views/creatureView"],
    function(Marionette,CreatureView) {

    var CreatureListView = Marionette.CollectionView.extend({
        itemView: CreatureView,
        className: 'creature-list'
    });

    return CreatureListView;
});
```

Koodiesimerkki 23. Kaikkien otuksien listausnäkymä tiedostossa creatureListView.js.

Seuraavaksi kokeiltiin sovelluksen toimivuutta, ja tietokannassa on valmiiksi REST-rajapinnan testauksen jäljiltä otuksia. Kun selaimella mennään osoitteeseen localhost:9999 ja Node-palvelin on käynnissä, näkymän pitäisi näyttää samalta kuin kuvassa 10.



Kuva 10. Ensimmäinen toteutus Creature cards -sovelluksesta selaimessa.

Sovellus ei varsinaisesti tee vielä mitään muuta kuin listaa tietokannasta löytyvät otukset päänäkymään. Sovelluksesta pitäisi olla mahdollista lisätä otuksia, muokata tai poistaa niitä. Kuvassa 10 näkyy "Add Creature" -linkki, joka toistaiseksi ei vie käyttäjää minnekään. Myös otusta klikkaamalla käyttäjän pitäisi päästä muokkaamaan sen tieto-

ja. Kuten sovellusta kuvaavasta kuvasta 5 voi nähdä, nämä molemmat toiminnot vievät käyttäjän samaan näkymään sillä erotuksella, että toiseen tuodaan otuksen tiedot ja toinen jätetään tyhjäksi. Ei siis tarvita kahta näkymää, vaan tarvitaan vain hieman enemmän koodia kyseistä näkymää toteuttavaan Underscore-mallinteeseen, jotta erilaiset tilanteet saadaan hallittua halutulla tavalla.

Koodiesimerkissä 24 jatketaan aiemmin tehtyä Backbone.routeria router.js-tiedostossa, jonne luodaan kaksi uutta reititystä osoitteisiin "add" ja "creautures/:id". Molemmat laukaisevat saman editCreature-funktion.

```
1     var AppRouter = Backbone.Router.extend({
2
3        routes: {
4            "" : "home",
5            "creatures/:id" : "editCreature",
6            "add" : "editCreature"
7          },
8
9         home: function() {
10            ...
11        },
24
25        editCreature: function(id) {
26            if ( this.creatureCollection ) {
27                this.creature = this.creatureCollection.get(id);
28            }
29
30            this.creatureEditView = new CreatureEditView({model: this.creature});
31            App.mainRegion.show(this.creatureEditView);
32        }
33    });
34    return AppRouter;
35 });
```

Koodiesimerkki 24. Päivitetty reititin otuksen muokkausmahdollisuudella.

EditCreature-funktio saa parametrinä id:n, kun polku "creatures/:id" laukaisee sen. Sen avulla haetaan juuri se otus tietokannasta, jota halutaan muokata. Lopuksi sovellukselle kerrotaan, että sen tulee näyttää mainRegion-alueessa creatureEditView-näkymä. Nyt päänäkymän "Add Creature" -linkki toimii halutulla tavalla, mutta tarvitaan myös creatureView-näkymään muutos, joka ohjaa käyttäjän samaan editView-näkymään kun otusta klikataan päänäkymän listauksesta. Muutos toteutetaan koodiesimerkissä 25.

```
    var CreatureView = Marionette.ItemView.extend({
        template: CreatureViewTemplate,
        className: 'creature-item',

        events: {
            'click': 'onCreatureClicked'
        },
        onCreatureClicked: function () {
            App.router.navigate('creatures/' + this.model.id, true);
        }
    });
    return CreatureView;
});
```

Koodiesimerkki 25. CreatureView-näkymään päivitetty klikkauksen käsittely.

Toistaiseksi ei ole olemassa creatureEditView-näkymää, joka toteutetaan seuraavaksi koodiesimerkissä 26. Näkymässä tulee olla lomake, joka on joko tyhjä tai täytetty muokattavan otuksen tiedoilla. Sen lisäksi näkymässä on erilaisia nappeja riippuen siitä, ollaanko tekemässä uutta vai päivittämässä vanhaa otusta.

```
define([...],
    function(...) {

    var CreatureEditView = Marionette.ItemView.extend({
        template: CreatureEditViewTemplate,

        events: {
            'click #delete-creature-button': 'onDeleteCreatureClicked',
            'click #back-button' : 'onBackButtonClicked',
            'click #save-creature-button': 'onSaveCreatureClicked'
        },

        onDeleteCreatureClicked: function(){
            this.model.destroy({
                success:function(){
                    App.router.navigate('/', true);
                }
            });
            return false;
        },

        onBackButtonClicked: function(){
            App.router.navigate('/',true);
            return false;
        },
        ...
```

Koodiesimerkki 26. Ensimmäinen osa editCreatureView-näkymää, jossa käsitellään otuksen poistaminen.

Koodiesimerkin 26 riveillä 7-11 määritellään eri nappien toiminnot tuttuun tapaan. Underscore-mallinteessa määritetään elementti, jolla on id-arvo "delete-creature-button". Se näytetään käyttäjälle vain muokattaessa jo olemassa olevaa otusta. Elementin painaminen kutsuu funktiota onDeleteCreatureClicked, joka kutsuu otus-mallille destroy-metodia, joka poistaa mallin ja vie sovelluksen poistamisen jälkeen takaisin päänäkymään. OnBackButtonClicked-funktio eli takaisin-napin toiminnallisuus tekee aivan saman asian eli vie käyttäjän sovelluksen päänäkymään. Molempien funktioiden lopussa palautetaan epätosi eli "return false", joka estää sivuston oman toiminnallisuuden toteutumisen, joka voisi olla esimerkiksi lomakkeen palautus.

Otuksen tallennusnappi näytetään aina, mutta Underscore-mallinteessa muutetaan napissa lukevaa tekstiä sen mukaan, muokataanko vai ollaanko luomassa uutta otusta. Mutta koska sovelluksessa käytetään samaa nappia, pitää koodissa myös ottaa huomioon, ollaanko tallentamassa vai luomassa uutta mallia. Tämä toteutetaan koodiesimerkissä 27.

```
    onSaveCreatureClicked: function() {
            if(this.model === undefined) {
                var creature = new CreatureModel({
                    name: $('input#creatureInputName').val(),
                    element: $('select#creatureInputElement').val(),
                    img: $('input#creatureInputImageURL').val(),
                    attack: $('input#creatureInputAttack').val(),
                    defense: $('input#creatureInputDefense').val()
                });
                creature.save(null, {
                    success: function(){
                        App.router.navigate('', true);
                    }
                });
            }else{
                this.model.set({
                    name: $('input#creatureInputName').val(),
                    element: $('select#creatureInputElement').val(),
                    img: $('input#creatureInputImageURL').val(),
                    attack: $('input#creatureInputAttack').val(),
                    defense: $('input#creatureInputDefense').val()
                });
                this.model.save(null, {
                    success: function(){
                        App.router.navigate('', true);
                    }
                });
            }
            return false;
        }
    });
    return CreatureEditView;
});
```

Koodiesimerkki 27. Toinen osa editCreatureView-näkymää, jossa on otuksen tallennuksen käsittely.

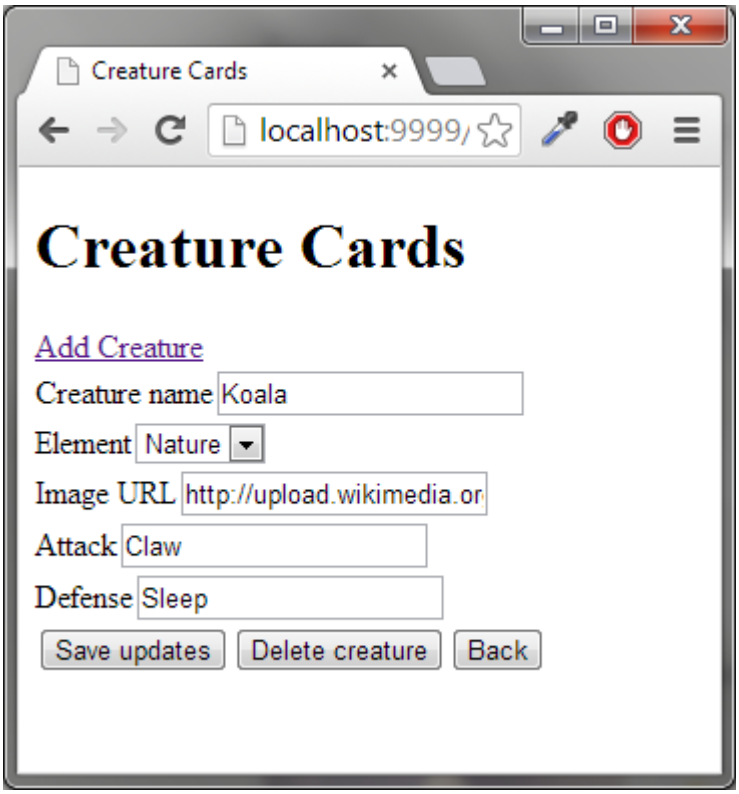
Ylimmällä tasolla tarkistetaan, onko malli määrittelemätön, jolloin tiedetään, että ollaan luomassa uutta otusta. Silloin luodaan uusi CreatureModel tiedoilla, jotka haetaan lomakkeesta jQuery-kirjastossa määritellyn val-funktion avulla. Jos malli on määritelty, tiedetään, että tulee päivittää jo olemassa olevaa otusta. Tässä tapauksessa otetaan samalla tavalla lomakkeesta tiedot ja korvataan vanhat arvot uusilla. Suurin ero tapauksissa on se, että uutta otusta tehtäessä kutsutaan "new CreatureModel" -funktiota, kun taas päivitettäessä kutsutaan löytyneelle mallille set-funktiota. Lopuksi kutsutaan save-funktiota, joka tallentaa tiedot MongoDB-tietokantaan. Jos tietojen tallennuksen onnistuu, käyttäjä viedään takaisin päänäkymään, jolloin uusi tai päivitetty otus haetaan tietokannasta käyttäjän nähtäväksi.

Luodaan vielä CreatureEditView-näkymälle Underscore-mallinnetiedosto nimeltään "creature-edit-view.tpl". Tpl-tiedostot ovat käytännössä HTML-kieltä, mutta koodiesimerkkissä 28 käytetään JavaScriptiä apuna. Koodissa on monessa kohtaa käytetty if - else - ehtolausetta: "typeof(_id) !== "undefined", jonka avulla tiedetään, muokataanko valmista otusta ja tuodaanko sen tiedot näkyviin vai jätetäänkö kohdat tyhjiksi. Esimerkiksi rivillä 44 nappulan teksti on "Save updates", jos "_id" ei ole määrittelemätön, mutta jos arvo löytyy, niin tekstiksi asetetaan "Create new creature". Muissa kohdissa, joissa ehtolause alkaa samalla tavalla, tarkastetaan myös, löytyykö kyseistä arvoa otukselta ja näytetään joko arvo tai tyhjä kenttä.

```
1  <form role="form">
2          <div class="form-group">
3                  <label for="creatureInputName">Creature name</label>
4                  <input type="text" class="form-control" id="creatureInputName"
5                  placeholder="Enter a name for creature"
6                  value="<%= typeof(_id) !== "undefined" && typeof(name) !== "undefined" ? name : "" %>">
7          </div>
8          <div class="form-group">
9                  <label for="creatureInputElement">Element</label>
10                 <select class="form-control" id="creatureInputElement">
11                         <option <%= typeof(_id) !== "undefined" && element === "Nature" ? 'selected' : '' %> >
12                           Nature
13                         </option>
14                         <option <%= typeof(_id) !== "undefined" && element === "Water" ? 'selected' : '' %> >
15                           Water
16                         </option>
17                         <option <%= typeof(_id) !== "undefined" && element === "Fire" ? 'selected' : '' %> >
18                           Fire
19                         </option>
20                         <option <%= typeof(_id) !== "undefined" && element === "Air" ? 'selected' : '' %> >
21                           Air
22                         </option>
23                 </select>
24         </div>
25         <div class="form-group">
26                 <label for="creatureInputImageURL">Image URL</label>
27                 <input type="text" class="form-control" id="creatureInputImageURL"
28                 placeholder="Enter an image URL for creature"
29                 value="<%= typeof(_id) !== "undefined" && typeof(img) != "undefined" ? img : "" %>">
30         </div>
31         <div class="form-group">
32                 <label for="creatureInputAttack">Attack</label>
33                 <input type="text" class="form-control" id="creatureInputAttack"
34                 placeholder="Enter a name for creatures attack"
35                 value="<%= typeof(_id) !== "undefined" && typeof(attack) != "undefined" ? attack : "" %>">
36         </div>
37         <div class="form-group">
38                 <label for="creatureInputDefense">Defense</label>
39                 <input type="text" class="form-control" id="creatureInputDefense"
40                 placeholder="Enter a name for creatures defense"
41                 value="<%= typeof(_id) !== "undefined" && typeof(defense) != "undefined" ? defense : "" %>">
42         </div>
43         <button id="save-creature-button" class="btn btn-success">
44                 <%= typeof(_id) !== "undefined" ? "Save updates" : "Create new creature" %>
45         </button>
46         <% if(typeof(_id) !== "undefined"){ %>
47                 <button id="delete-creature-button" class="btn btn-danger">Delete creature</button>
48         <% } %>
49         <button id="back-button" class="btn btn-default">Back</button>
50 </form>
```

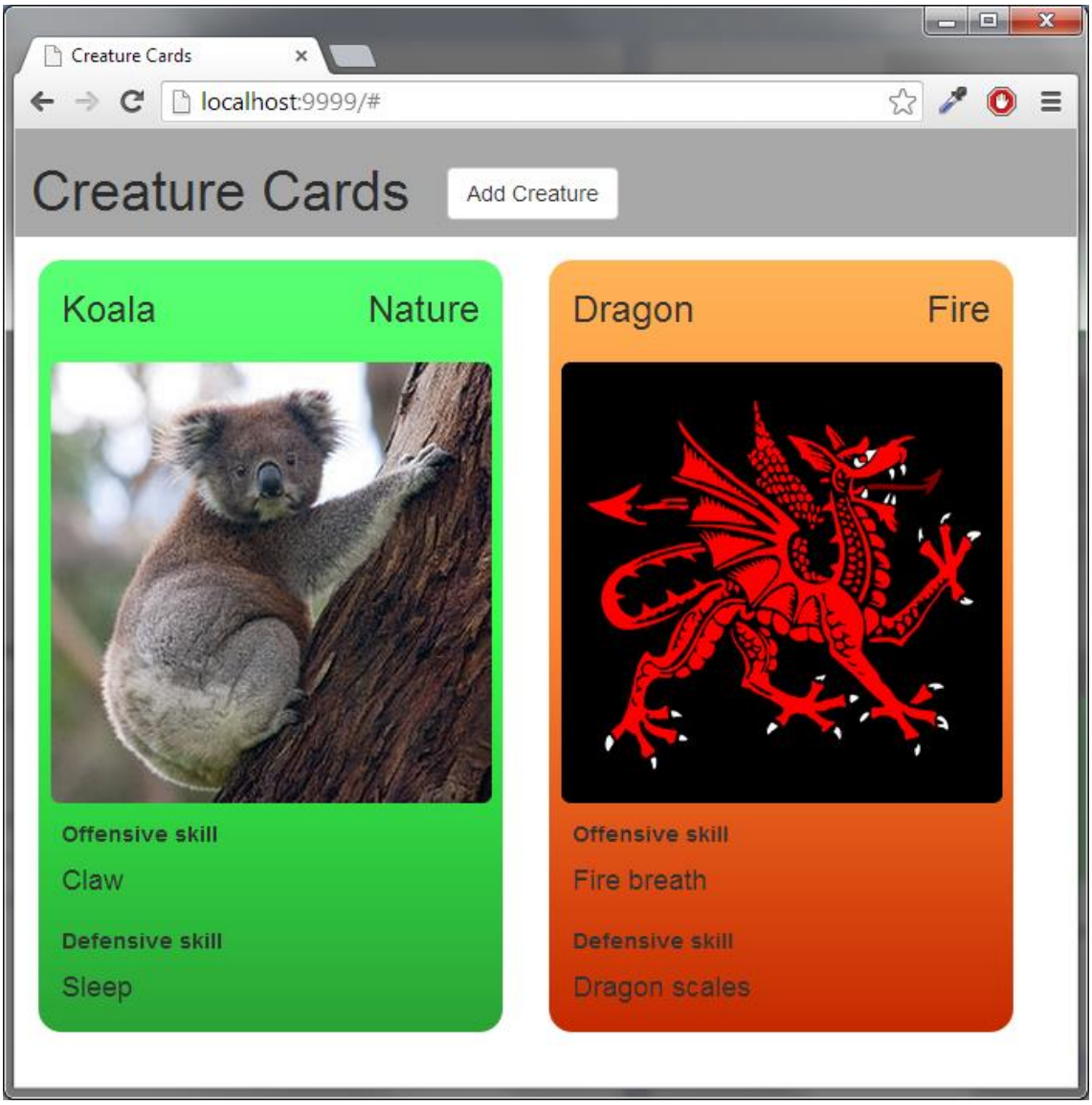
Koodiesimerkki 28. CreatureEditView-näkymän Underscore-mallinne.

Otuksen lisääminen, poistaminen ja muokkaaminen on nyt mahdollista selaimessa. Kuvissa 10 ja 11 on nähtävissä, miltä sovellus näyttää ilman tyylimäärittelyitä. Lisäksi toistaiseksi päänäkymässä näytetään vain osa otuksen tiedoista. Sovelluksen nimen mukaisesti oli tarkoitus tehdä otuksista kortteja.



Kuva 11. Otuksen muokkaustilanne CreatureEditView-näkymässä.

Tyylimääritteisiin käytettiin hyödyksi otuksille valittavaa elementtiä, jonka avulla kortille annetaan sopiva taustaväri, esimerkiksi luontoelementin otukselle vihreä tausta. Lisäksi nappuloista ja lomakkeen tekstikentistä saadaan pienellä työllä kauniin näköisiä käyttämällä valmista bootstrap-tyylikirjastoa. [27.] Kuvassa 12 nähdään, miltä sovellus näyttі lopulta tyylimääritteiden kanssa.



Kuva 12. Sovelluksen päänäkymän viimeistelty versio tyylimäärittelyineen.

3.5 Sovelluksen optimointi

Kun sovellus on päällisin puolin valmis käyttöön, on hyvä ottaa huomioon sen raskaus ja mahdollisuudet optimointiin. Web-optimoinnilla tarkoitetaan sivun latausajan saamista mahdollisimman lyhyeksi, mikä tarkoittaa käytännössä pakattua koodia, pakattuja kuvia ja mahdollisimman pientä määrää palvelimelle tehtyjä hakupyyntöjä.

Require.js auttaa yhdistämään kaikki JavaScript-tiedostot yhdeksi tiedostoksi. Otetaan käyttöön uusi työkalu nimeltä Grunt. Se mahdollistaa automaattisten käyttöönottoskriptien suorittamisen nopeasti. Sen avulla voidaan toteuttaa kaikki tarvittava optimointi kuvien pienentämisestä koodin pakkaamiseen. Tälle sovellukselle Gruntin tärkein teh-

tävä on käydä läpi Require.js-konfiguraatio ja yhdistää tiedostot yhdeksi main.js-tiedostoksi [28]. Jotta Grunt ei ylikirjoittaisi tehtyä koodia, luodaan app-hakemiston juureen build-kansio, johon Grunt työstää optimoidun version sovelluksesta.

Asennetaan Grunt tuttuun tapaan NPM:n avulla komennolla "npm install -g grunt-cli". Komento asentaa Gruntin käyttöjärjestelmän määrittelemään sijaintiin, jolloin Gruntia voi käyttää missä tahansa kansiossa ja projektissa yhden asennuksen jälkeen. Grunt tarvitsee toimiakseen konfigurointitiedoston gruntfile.js, joka luodaan projektikansioon. Toimiakseen Grunt tarvitsee NPM:ää, joten konfiguraatioon määritellään ensimmäisenä polku package.json-tiedostoon.

Jokaisen Grunt-ajon yhteydessä halutaan, että äsken luotu build-kansio tyhjennetään, jotta voidaan aloittaa uuden luominen tyhjästä. Asennetaan Gruntiin clean-liitännäinen komennolla "npm install grunt-contrib-clean --save-dev". Asennuskommennon loppuosa kirjoittaa package.json-tiedostoon liitännäiseksi clean-kirjaston, joten sitä ei tarvitse sinne erikseen kirjoittaa. Asennetaan vielä Require.js-liitännäinen ja kopiointiliitännäinen kirjoittamalla "npm install grunt-contrib-requirejs --save-dev" ja "npm install grunt-contrib-clean --save-dev".

Gruntfile.js:ssä on funktio grunt.initConfig, jonka sisällä määritellään kunkin liitännäisen tehtävä. Clean-liitännäinen postaa build-kansion. Require.js-liitännäisen käsketään etsiä sovelluksen aloituspiste eli main.js-tiedosto, josta se lähtee selvittämään sovelluksen käyttämiä liitännäisiä eteenpäin Require.js:n konfiguraatiotiedoston avulla. Lopulta se yhdistää kaikki JavaScript-tiedostot yhdeksi tiedostoksi build-kansion alle js-hakemistoon. Lopuksi kopioidaan index.html- ja require.js-tiedostot build-kansioon.

Kuten kooesimerkissä 29 nähdään, gruntfile.js:n lopussa haetaan NPM:n avulla Grunt-liitännäiset käyttöön ja registerTask-funktiossa kerrotaan, mitkä Grunt-toimenpiteet tulee ajaa milloinkin ja missä järjestyksessä. Kun komentoriville kirjoitetaan: "grunt", se ajaa clean-, requirejs- ja copy-toimenpiteet.

```
module.exports = function(grunt) {
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        clean: ['app/build'],
        requirejs: {
            compile: {
                options: {
                    baseUrl: 'app/js',
                    name: 'main',
                    mainConfigFile: 'app/js/main.js',
                    out: 'app/build/js/main.js'
                }
            }
        },
        copy: {
            main: {
                files: [
                    {   expand:true,
                        cwd: 'app/',
                        src: ['index.html','js/lib/requirejs/require.js'],
                        dest: 'app/build/'
                    }
                ]
            }
    });

    grunt.loadNpmTasks('grunt-contrib-clean');
    grunt.loadNpmTasks('grunt-contrib-requirejs');
    grunt.loadNpmTasks('grunt-contrib-copy');

    grunt.registerTask('default', ['clean','requirejs','copy']);
};
```

Koodiesimerkki 29. Gruntin konfiguraatiotiedosto gruntfile.js.

Viimeisenä tulee muistaa muuttaa server.js tarjoilemaan tiedostot build-kansiosta app-kansion sijaan. Require.js:n avulla saatiin vähennettyä JavaScript-tiedostoja hakevien HTTP-pyyntöjen määrä kuudestatoista hausta vain kahteen. Lisäksi Require.js pienentää koodin automaattisesti muuttamalla muuttujien ja funktioiden nimet yksikirjaimisiksi ja poistamalla turhat välit. [28.]

4 Yhteenveto

JavaScriptin ensimmäinen versio ilmestyi vuonna 1995, ja siitä lähtien sitä on käytetty internetselaimissa erilaisten tehtävien suorittamiseen. JavaScript on kehittynyt paljon, ja se kehittyy edelleen lisää, minkä ansiosta siitä on tullut yksi maailman suosituimmista ohjelmointikielistä. Yksi suosion pääsyistä on varmasti JavaScriptin helppo ymmärrettävyys verrattuna monimutkaisempiin ohjelmointikieliin. Vaikka JavaScriptillä voi toteuttaa myös haastavia asioita, sen kanssa pääsee alkuun nopeasti. Toinen tärkeä syy suosioon on JavaScriptin moniulotteisuus. Sen avulla voidaan kehittää sovelluksia selaimeen, mobiililaitteisiin natiivisovelluksen tavoin PhoneGap-ohjelmiston avulla ja palvelintoteutuksia Node.js:n ja CommonJS:n avulla. Tessel.io mahdollistaa JavaScriptillä käytön jopa fyysisillä laitteilla [30].

Insinöörityön tavoitteena oli oppia mahdollisimman paljon insinöörityön tilaajan Pieni piiri Oy:n käyttämistä teknologioista. Esimerkkiprojektina toteutettiin JavaScript-sovellus, jonka käyttöliittymäkomponentit olivat Backbone.js:n ja Marionette.js:n muodostama MVC-arkkitehtuuri, joiden apuna käytettiin Underscore.js-mallinnekirjastoa. Marionette.js täydentää Backbone.js:stä puuttuvia ominaisuuksia, kuten puuttuvia käsittelijöitä (controllers), ja Marioneten avulla vältetään saman koodin kirjoittamista uudelleen, sillä se sisältää useita ennalta määriteltyjä komponentteja, joita on helppo käyttää ja joiden avulla esimerkiksi näkymiä tai kokoelmia on koodillisesti siistimpä toteuttaa. Underscore.js:n avulla perinteisistä HTML-tiedostoista saadaan helposti dynaamisia ja käyttäjälle pystytään näyttämään kohdennettua tietoa vähemmällä työllä.

Palvelin toteutettiin suositulle Node.js-alustalle käyttäen REST-arkkitehtuurimallia. Vaikka sovellukseen toteutettu Node.js-palvelin oli yksinkertainen, se oli myös tehokas ratkaisu pienen sovelluksen tarpeisiin. Käytännössä palvelin jakoi tarvitut tiedostot, kun käyttäjä niitä selaimen avulla pyysi, ja suoritti tietojen tallentamisen ja hakemisen MongoDB-tietokannasta. Sinne tallennettiin vain yhden dataskeeman mukaista tietoa eli otuksia. Kaikki tieto, jota tietokantaan tallennettiin, oli tiedonsiirtomuodoltaan JSONia eli JavaScriptin syntaksia noudattavaa tietoa. Rajapinta MongoDB:n ja Node.js-palvelimen välille toteutettiin Mongoose-nimisen kirjaston avulla.

Lopuksi sovellus optimoitiin käyttämällä Require.js- ja Grunt.js-kirjastoja. Require.js mahdollisti sovelluksen eri komponentien lataamisen asynkronisesti vain, kun niitä tarvitaan. Grunt.js:n avulla pystyttiin toteuttamaan skripti, joka tekee koodista ja muusta sisällöstä tehokkaasti verkossa jaettavaa. Samalla kuitekin voidaan pitää toinen versio koodista helppolukuisena ja helposti ylläpidettävänä.

Insinöörityö oli erittäin opettavainen, ja monet uudet uudet teknologiat, joita JavaScript sisältää, tulivat tutuksi. Muun muassa perehdyttiin paketinhallintatyökaluihin, jotka helpottivat liitännäisten asentamista. Käytettyjä paketinahallintatyökaluja olivat Node.js:n NPM ja käyttöliittymän yhteydessä käytetty Bower. Mielestäni tavoitteet saavutettiin paremmin kuin hyvin.

Muutama vuosi takaperin perheillä oli jaettu PC olohuoneessa, mutta nykyään jokaisella käyttäjällä on useampia tietokoneita. Kutsumme niitä sitten älypuhelimiksi, tableteiksi, läppäreiksi, lukulaitteiksi tai ihan vain tietokoneiksi, niissä kaikissa toimii JavaScript. Tulevaisuudessa tällaisia laitteita on yhä useampia, joten JavaScript ei ole ainakaan ihan heti poistumassa kuvioista.

Lähteet

1 JavaScript HTML DOM. Verkkodokumentti. W3Schools. <http://www.w3schools.com/js/js_htmldom.asp>. Luettu 11.11.2013.

2 Usage of server-side programming languages for websites. 2014. Verkkodokumentti. W3Techs. <http://w3techs.com/technologies/overview/programming_language/all>. 30.1.2014. Luettu 30.1.2014.

3 Rauschmayer, Axel. 2012. The Past, Present, and Future of JavaScript. E-kirja: O'Reilly Media.

4 Champeon, Steve. 2001. JavaScript: How Did We Get Here? Verkkodokumentti. <http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html>. 6.4.2001. Luettu 15.11.2013.

5 Heilmann, Chris. 2009. What can you do with JavaScript? Verkkodokumentti. <http://dev.opera.com/articles/view/javascript-uses/>. 3.2.2009. Luettu 15.11.2013.

6 Usage of JavaScript libraries for websites. 2014. Verkkodokumentti. W3Techs. <http://w3techs.com/technologies/overview/javascript_library/all>. 30.1.2014. Luettu 30.1.2014.

7 jQuery. 2013. Verkkodokumentti. jQuery. Luettu 15.11.2013.

8 JavaScript Libraries. Verkkodokumentti. W3Schools. <http://www.w3schools.com/js/js_libraries.asp>. Luettu 14.11.2013.

9 Introducing JSON. 2013. Verkkodokumentti. JSON.org. <http://www.json.org/>. Luettu 15.11.2013.

10 Hypertext Transfer Protocol -- HTTP/1.1. 1999. Verkkodokumentti. IETF. <http://www.ietf.org/rfc/rfc2616.txt>. Luettu 16.11.2013.

11 Osmani, Addy. 2013. What is MVC. Verkkodokumentti. <http://addyosmani.github.io/backbone-fundamentals/#what-is-mvc>. Luettu 16.11.2013.

12 How does Backbone relate to "traditional" MVC? 2013. Verkkodokumentti. Backbone.js. <http://backbonejs.org/#FAQ-mvc>. Luettu 20.11.2013.

13 Backbone.Model. 2013. Verkkodokumentti. Backbone.js. <http://backbonejs.org/#Model>. Luettu 20.11.2013.

14 Backbone.View. 2013. Verkkodokumentti. Backbone.js. <http://backbonejs.org/#View>. Luettu 20.11.2013.

15 Underscore.js. 2013. Verkkodokumentti. Underscore.js. <http://underscorejs.org/>. Luettu 20.11.2013.

16 Osmani, Addy. 2013. Controllers. Verkkodokumentti. <http://addyosmani.github.io/backbone-fundamentals/#controllers>. Luettu 16.11.2013.

17 Node's goal is to provide an easy way to build scalable network programs. 2013. Verkkodokumentti. Node.js. <http://nodejs.org/about/>. Luettu 21.11.2013.

18 Require.js. 2013. Verkkodokumentti. Require.js. <http://requirejs.org/>. Luettu 22.11.2013.

19 Bower spec. 2013. Verkkodokumentti. Bower. <https://docs.google.com/document/d/1APq7oA9tNao1UYWyOm8dKqlRP2blVkROYLZ2fLIjtWc/edit#heading=h.4pzytc1f9j8k>. Luettu 22.11.2013.

20 The MongoDB 2.4 Manual. 2013. Verkkodokumentti. MongoDB, Inc. <http://docs.mongodb.org/manual/>. Luettu 26.11.2013.

21 Schemas. 2013. Verkkodokumentti. Mongoose. <http://mongoosejs.com/docs/guide.html>. Luettu 26.11.2013.

22 Curl Man Page. 2013. Verkkodokumentti. Curl. <http://curl.haxx.se/docs/manpage.html>. Luettu 27.11.2013.

23 Google Chrome Frame. 2013. Verkkodokumentti. Google. <https://developers.google.com/chrome/chrome-frame/>. Luettu 27.11.2013.

24 Backbone.Collection. 2013. Verkkodokumentti. Backbone.js. <http://backbonejs.org/#Collection>. Luettu 20.11.2013.

25 Use cases for the different views. 2013. Verkkodokumentti. Marionette.js. <https://github.com/marionettejs/backbone.marionette/wiki/Use-cases-for-the-different-views>. Luettu 27.11.2013.

26 Zombies! RUN! (Managing Page Transitions In Backbone Apps). 2013. Verkkodokumentti. Bailey, Derick. <http://lostechies.com/derickbailey/2011/09/15/zombies-run-managing-page-transitions-in-backbone-apps/>. Luettu 27.11.2013.

27 Overview. 2013. Verkkodokumentti. Bootstrap. <http://getbootstrap.com/css/>. Luettu 28.11.2013.

28 Why AMD? 2013. Verkkodokumentti. Require.js. <http://requirejs.org/docs/whyamd.html>. Luettu 22.11.2013.

29 Grunt. 2013. Verkkodokumentti. Grunt.js. <http://gruntjs.com/>. Luettu 28.11.2013.

30 Tessel.io. 2013. Verkkodokumentti. Technical Machine. <http://tessel.io/>. Luettu 29.11.201