HAAGA-HELIA
ammattikorkeakoulu

# Full-Stack JavaScript: MEAN.JS-projektin luominen

Harri Huhtala

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2014

HAAGA-HELIA
ammattikorkeakoulu

**Tiivistelmä**

19.11.2014

Tietojenkäsittelyn koulutusohjelma

| **Tekijä tai tekijät**<br>Harri Huhtala | **Ryhmätunnus tai aloitusvuosi**<br>2011 |
|---|---|
| **Raportin nimi**<br>Full-Stack JavaScript: MEAN.JS-projektin luominen | **Sivu- ja liitesivumäärä**<br>41 |
| **Opettajat tai ohjaajat**<br>Mirja Jaakkola | |


Node.js alustan julkaisun myötä web-sovelluskehitys elää murrosvaihetta ja perinteisen kehitystyön rinnalle on noussut uusia menetelmiä. Kehitystyössä JavaScript-kielen käyttö on lisääntynyt ja uusia JavaScript-pohjaisia komponentteja julkaistaan kiihtyvällä tahdilla. Node.js alustan ympärille on syntynyt suuntaus nimeltään Full-Stack JavaScript.

Opinnäytetyö toteutettiin syksyllä 2014 ja sen tavoitteena on tutkia mitä tarkoittaa termi Full-Stack JavaScript ja mitä yleisiä tekniikoita se sisältää. Tekniikoiden pohjalta rakennetaan prototyyppisovellus, jonka rakenne ja komponentit kuvataan.

Prototyypin rakennukseen käytetään suosittua Full-Stack JavaScript sovelluskehystä MEAN-pakkaa ja pakan tärkeimmät toiminnot kuvataan. Opinnäytetyön ulkopuolelle rajataan sovelluksen vienti tuotantoon ja testausohjelmien kirjoittaminen.

Tuloksissa kuvataan, kuinka Full-Stack JavaScript-sovelluksessa tiedon tallentaminen tapahtuu tietokantaan ja esitetään, kuinka prototyypin komponentit kommunikoivat keskenään. Sovelluksen kansiorakenne käydään läpi tiedon muokkauksen yhteydessä.

Johtopäätöksissä todetaan Full-Stack JavaScript kehittämisen olevan MEAN-pakalla ketterää. Komponentit kommunikoivat hyvin keskenään ja sovellusarkkitehtuurin toteuttaminen yhdellä ohjelmointikielellä sekä helpottaa että keventää kehitystyötä. Tämän mahdollistaa Node.js alusta ja sen ympärille kehittynyt ekosysteemi.



**Asiasanat**
JavaScript, Full-Stack JavaScript, MEAN, Single-Page Application

HAAGA-HELIA
University of Applied Sciences

**Abstract**

19.11.2014

Degree programme in Business Information Technology

| **Authors**<br>Harri Huhtala | **Group or year of entry**<br>2011 |
|---|---|
| **The title of thesis**<br>Full-Stack JavaScript: Creating MEAN.JS project | **Number of report pages and attachment pages**<br>41 |
| **Advisor(s)**<br>Mirja Jaakkola | |


After the release of Node.js platform, web application development has been in the middle of a change and new techniques are rising next to the traditional ones. The use of the JavaScript language in software development has increased and new JavaScript based components are being released at an accelerating rate. A new development trend has emerged, which is built around Node.js platform. The trend is called Full-Stack JavaScript.

The thesis had been carried out by fall 2014, and the aim was to study what the term Full-Stack JavaScript mean and what kind of general techniques it contained. Based on the techniques, a prototype application was developed and the structure and components of the prototype were described.

In the development process, a MEAN Full-Stack JavaScript framework was used and most fundamental parts of the stack were explained. Deployment to production and a writing unit test were excluded from the scope of the study.

As a result, the study has illustrated how the Full-Stack JavaScript application saved data to a database and described how application components interacted with other components. The folder structure of the application was explained when modifying data occurred.

The thesis concludes that developing Full-Stack JavaScript application with the MEAN-stack is agile. Communication of the components is excellent and creating software architecture with one language facilitates and streamlines the developing process. It is possible due to the Node.js platform and the ecosystem built around it.



JavaScript, Full-Stack JavaScript, MEAN, Single-Page Application

# Sisällys

# 1 Johdanto

Viime vuosina perinteisen web-sovellusarkkitehtuurin rinnalle on noussut käsite full stack. Termin rajat ovat olleet liukuvat, mutta alkuperäisessä merkityksessä Facebook yhtiön työntekijä Carlos Buenon kuvaili sitä:

”Full-stack-ohjelmoija on yleismies, joka pystyy luomaa monipuolisia sovelluksia yksin. Kukaan ei voi tietää kaikesta kaikkea, mutta ohjelmoijan pitäisi pystyä visualisoimaan mitä ylhäällä ja alhaalla pakassa (stack) tapahtuu, kun sovellus suorittaa toiminnon.” (Bueno 2010.)

Mike Loukides tarkensi stack-termiä kuvailemalla pakan historiaa. LAMP (Linux, Apache, MySql, Perl/PHP) oli ensimmäinen paketti, jossa komponentit muodostivat yhtenäisen kokonaisuuden: käyttöjärjestelmä, palvelin, tietokanta ja väliohjelmisto (middleware). Tuolloin työtaakka suoritettiin palvelimella ja JavaScript-kieltä käytettiin käyttöliittymän toiminnallisuuksien luontiin asiakkaan selaimessa. (Loukides 2014.)

Kielenä JavaScript on tehnyt pitkän matkan. Alkuperäisestä käyttöliittymän muokkaamiseen tarkoitetusta kielestä on kehittynyt 2000-luvulla täysiverinen ohjelmointikieli, jota voidaan käyttää sekä asiakkaan- että palvelimen puolella. Tämän myötä kielen suosio kehitystyössä on lisääntynyt ja on syntynyt käsite Full-Stack JavaScript. Termillä tarkoitetaan sovellusta, joka on toteutettu JavaScript-kielellä läpi sovellusarkkitehtuurin.

## 1.1 Opinnäytetyön tavoitteet ja rajaus

Opinnäytetyön aihe valittiin sen ajankohtaisuuden vuoksi. Node.js alustan julkaisun myötä web-kehitys on murroksessa. Perinteinen Linux, Apache, MySql ja PHP (LAMP) pakka on saanut kilpailijan Node.js alustan ympärille kehittyneestä ekosysteemistä. Erityisesti Full-Stack JavaScript suuntaus on saavuttanut suosiota kehittäjien keskuudessa.

Opinnäytetyön tarkoituksena on esitellä yleiset tekniikat, joilla Full-stack JavaScript web-sovellus voidaan rakentaa. Näiden tekniikoiden pohjalta rakennetaan yksisivuinen

Single-Page Application (SPA) web-sovellusprototyyppi ja tärkeimmät kehitysvaiheet kuvataan. Prototyypin avulla tutkitaan, miten eri web-tekniikoiden sitominen yhteen toteutetaan JavaScript-kielellä.

Teoriaosuudessa esiteltyjen tekniikoiden pohjalta pyritään löytämään vastaus kysymyksiin mitä tarkoitetaan käsitteellä Full-Stack JavaScript ja mistä yleisistä tekniikoita se rakentuu? Tutkivassa osuudessa selvitetään vastausta kysymyksiin, miten rakennetaan Full-Stack JavaScript-sovellus, mitä komponentteja sovellukseen asennetaan ja mitkä ovat niiden yleisimmät tehtävät?

Opinnäytetyön ulkopuolelle rajataan sovelluksen tuotantoon vienti ja testausohjelmien kirjoittaminen. Myöhempi jatkojalostaminen on epätodennäköistä. Tutkimuksessa kuvataan prototyypin ja MEAN.JS-pakan kansionrakennetta. Kuvaus sisältää yksityiskohtaista tietoa, jotta tietovirta sovellusarkkitehtuurissa saadaan kuvattua.

### 1.2 Käsitteet

**AJAX (Asyncronous JavaScript and XML)** = kokoelma eri tekniikoita, joilla HTML-dokumentin osittainen päivitys voidaan suorittaa ilman sivun uudelleenlatausta

**AngularJS =** Googlen kehittämä MV* sovelluskehys, siirtää perinteistä palvelinpuolen logiikkaa asiakkaan puolelle

**API (Application programming interface) =** ohjelmointirajapinta

**Callback-funktio =** parametrissa välitettävä funktiokutsu, pyrkii varmistamaan oikean ajojärjestyksen funktiossa

**CDN (Content Delivery Network) =** laaja hajautettu palvelinverkosto, joka tarjoaa asiakkaalle nopeasti dataa

**DOM (Document Object Model)=** ohjelmointirajapinta HTML-dokumentin muokkausta varten

**Express** = kevyt palvelinpuolen sovelluskehys, joka tarjoaa menetelmiä mm reititykseen

**I/O (input/ouput) =** kirjoitus- ja lukuoperaatio

**JSON (JavaScript Object Notation) =** alustariippumaton tiedonsiirtoformaatti

**LAMP =** Linux, Apache, MySql, Perl (myöhemmin PHP)

**Middleware =** väliohjelmisto, joka tarjoaa palveluja ohjelmistolle

**MEAN (MongoDB, Express, Angular, Node.js) =** ohjelmistopakka, joka rakentaa JavaScript-pohjaisen web-sovelluksen rungon

**Mongoose =** sovelluskehys helpottamaan tiedon tallentamista MongoDB kantaan

**Node** = yksittäinen solmu DOM puussa, joka sisältää informaatiota HTML elementistä

**Node.js =** alusta, jolla voidaan ajaa JavaScript-kieltä palvelimen puolella

**Non-blocking** = ei-estävä luku- ja kirjoitusoperaatio

**npm (Node Package Manager)** = Node.js:n mukana tuleva paketinhallintajärjestelmä

**MV* ( Model View *) =** yleistermi ohjelmiston arkkitehtuurimalleille, joilla nykyaikainen web-sovellus toteutetaan. MVC (malli, näkymä, käsittelijä) kuuluisin

**RDBMS (relational database management system)** = perinteinen tietokantamalli, joka perustuu relaatiomalliin

**Runtime =** sovelluksen ajamisen aikainen tapahtuma

**REST (Representational State Transfer )** = arkkitehtuurimalli HTTP -protokollaan perustuvien ohjelmointirajapintojen luomiseen.

**RWD (Responsive Web Design)** = suunnittelumalli websivuille, jonka avulla sivun sisältö skaalautuu käyttäjän laitteen näkymään

**SPA (Single Page Application) =** yksisivuinen websovellus, joka toimii kuorena koko sivuston sisällölle

**Yeoman** = Komentorivipohjainen ohjelma, jolla esirakennetaan websovelluksen runko ja haetaan sovelluksien riippuvuudet

# 2 JavaScript

Modernin verkon HTML-sivustojen arkkitehtuuri voidaan jakaa kolmeen osaan: HTML kuvaa rakenteen, CSS kuvaa sivuston tyylit ja JavaScript vastaa toiminnallisuudesta. JavaScript on modernin verkon ohjelmointikieli, suurin osa uusista sivustoista sisältää sitä. (Flanagan 2011, 1.)

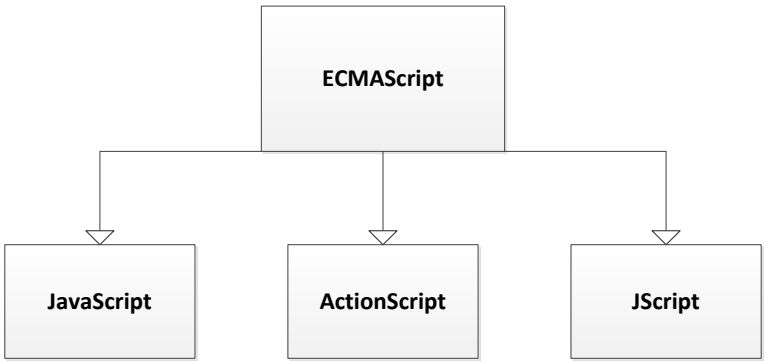
JavaScript kielen kehitti amerikkalainen ohjelmoija Brendan Eich, joka työskenteli Netscapella (nykyään Mozilla). Kieli kehitettiin vain kymmenessä päivässä ja julkaistiin 1995. Brendan Eich kirjoittaa Effective JavaScript kirjan esipuheessaan kehitysvaatimuksista seuraavasti:

"Seuraavat vaatimukset täytyivät toteutua: helppo aloittelijalle, näyttää Javalta ja kielellä voidaan kontrolloida melkein kaikkea Netscapen selaimessa." (Hernan 2012, foreword.)

JavaScriptin vahvuus oli sen rakenne, jos osasi entuudestaan jonkun ohjelmointikielen, JavaScript tuntui alusta lähtien tutulta. Kieli sai vaikutteita useista eri skriptauskielistä. Aluksi sitä suunniteltiin tukemaan Javaa, mutta kieli kehittyi ja lopulta ohitti Javan suosion. (Hernan 2012, 1.)

Vuonna 1997 JavaScript lähetettiin standardoitavaksi ECMA:lle ( The European Computer Manufacturer´s Accociation), mutta nimen kanssa ilmenneiden tuotemerkkiongelmien vuoksi se muutettiin ECMAScript-nimiseksi ja nykyinen JavaScript on yksi murre siitä. (Web Education Community 2012.)

JavaScript rakentuu kolmesta osasta: kielen ytimen muodostaa ECMAScript standardi. Tämän lisäksi kieleen kuuluu DOM (Document Object Model) ja BOM (Browser Object Model). ECMAScript määrittelee kielen pohjarakenteen olematta sidoksissa ympäristöön ja. ECMAScriptiä voidaan toteuttaa muuallakin kuin selaimessa. DOM ja BOM puolestaan laajentavat ECMAScriptin ominaisuuksia ja määrittelevät sen sidonnaisuuksia. (Zakas 2010, 3.)



Kuvio 1 ECMAScript ja esimerkkejä sen perillisistä

Rakenteeltaan JavaScript on korkeantason, dynaamisesti tyypitetty, tulkattava kieli. Se on prototyyppipohjainen skriptikieli, jolla on olio-ohjelmointiin liittyviä ominaisuuksia. Syntaksi perustuu löyhästi C-kieleen. Nykyään Oraclen hallinnoiman Java-kielen kanssa sillä ei ole juuri tekemistä, nimen samankaltaisuus perustui aikoinaan markkinointisyihin. (Mozilla Developer Network. 2014a.)

Skriptauskielillä laajennetaan HTML-sivua lisäämällä sinne koodia, jonka selain kääntää ja ajaa lennosta (runtime). Selainten tuki JavaScriptille on erinomainen, jopa 99.1-99.9% käyttää sitä. (Steyer 2013, 32-34.)

Kielen luonne on muuttunut alkuvuosista. Sivuston dynaamisuuden toteuttamisen lisäksi sitä käytetään selainpuolen (front-end) ohjelmoinnista palvelinpuolen ohjelmiin. JavaScriptillä tehdään myös mobiili- ja työpöytäsovelluksia. (Hernan 2012, 2.)

## 2.1 DOM

DOM (Document Object Model) on ohjelmointirajapinta (API), joka kuvaa HTML-sivun hierarkkisena, puumaisena rakenteena. JavaScriptin toiminta perustuu DOM-rajapinnan tarjoamiin menetelmiin, joilla päästään käsiksi sivun elementteihin. Rajapinnan avulla sivun elementtejä voidaan lisätä, poistaa ja muokata.

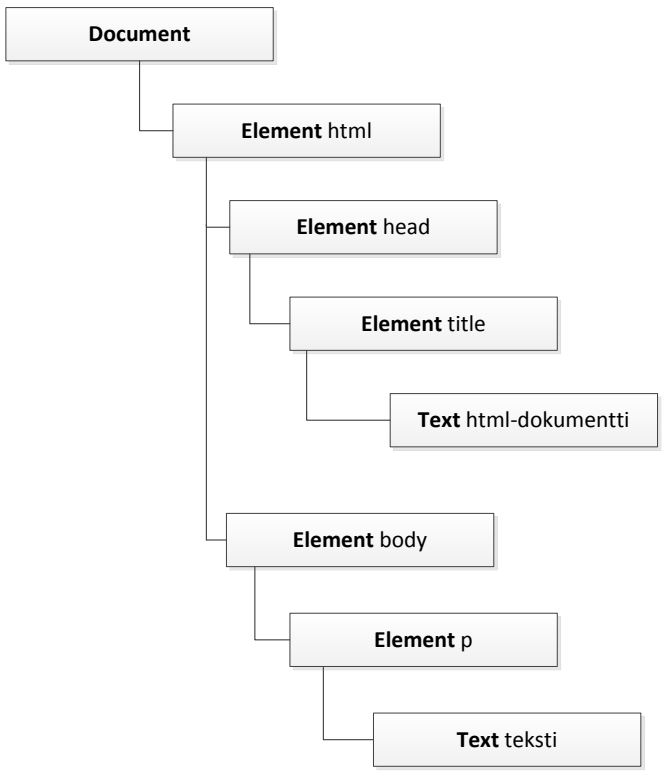
Jokainen HTML-sivu rakentuu DOM-puusta, mutta eri selaimet saattavat kuvata sivun rakennetta eri tavoin. Myös DOM ohjelmointirajapinnan menetelmät voivat vaihdella eri selaimissa ja jotkut voivat jopa puuttua kokonaan. (Steyer 2013, 83.)

HTML-elementit kuvataan DOM-puussa solmuina (node), jotka sisältävät informaatiota tai kuvauksen elementistä. Solmut muodostavat sidosryhmän toisiinsa, jonka avulla rakennetaan hierarkkinen kuvaus dokumentista. Document-solmu muodostaa juuren, josta HTML-sivun lapsielementit (childelement) periytyvät. HTML-sivuilla document-solmuna toimii <html> elementti jonka sisällä kaikki lapsielementit sijaitsevat. (Zakas 2010, 261-262.)

```
<html>
  <head>
    <title>html-dokumentti</title>
  </head>
  <body>
    <p>Teksti</p>
  </body>
</html>
```

Kuvio 2 Esimerkki html-sivu, josta muodostetaan DOM-puu

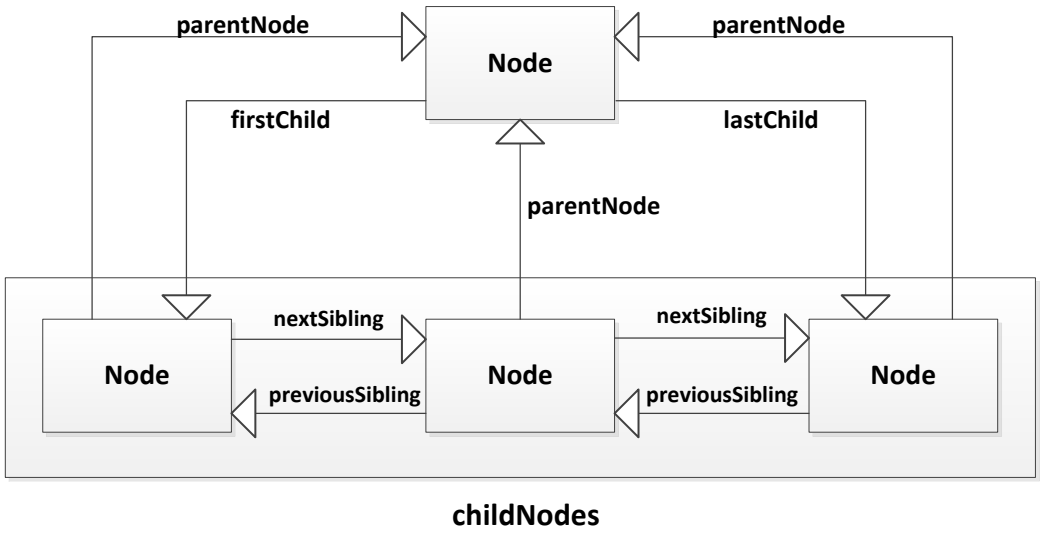
Solmutyypejä (nodeType) on määritelty yhteensä 12 kpl (W3schools.com). Yllä olevassa html-dokumentissa on käytetty kahta tyyppiä, ELEMENT_NODE ja TEXT_NODE. Dokumentista muodostuu seuraavanlainen DOM-puu.




Kuvio 3 DOM-puu kahdella eri solmutyypillä

Solmut muodostavat toisiinsa suhteen, joka kuvataan perhesuhteiden tavoin. HTML-dokumentissa <html> elementti on vanhempi (parent), josta sisarukset (siblings) ja lapset (child) periytyvät. Kuviossa 3<html> elementti on vanhempi, <body> ja <head> elementit sisaruksia ja esimerkiksi <title> elementti on <head> elementin lapsi.

Jokaisella solmulla on NodeList taulukko, joka sisältää tietoa solmun suhteista dokumentin muihin solmuihin. Solmuilla on parentNode ominaisuus, joka osoittaa vanhempaansa ja childNodes lista, joka sisältää solmun sisarukset. Dokumentin solmuissa voidaan navigoida previousSibling, nextSibling, firstChild, lastChild ja ownerDocument ominaisuuksia hyväksi käyttäen. (Zakas 2010, 264-266.)




Kuvio 4 Solmujen suhteet ja ominaisuudet, joilla päästään käsiksi solmuun (Zakas 2010, 266)

## 2.2 jQuery

jQuery on John Resigin luoma JavaScript-kieleen perustuva kirjasto. Kirjaston sisältämät kaikki funktiot voidaan ohjelmoida perinteisellä JavaScript-kielellä, mutta kirjasto yksinkertaistaa funktioiden käyttöä. Sen käyttöön ei tarvita liitännäisiä (plug-in) tai laajennuksia (extension), vaan kaikki modernit selaimet tukevat sitä. (Steyer 2013, 34.)

Yksi kirjaston tärkeimmistä ominaisuuksista on funktioiden ajo, niitä ei toteuteta ennen kuin kaikki DOM-elementit sijaitsevat paikoillaan. Perinteisellä JavaScript-kielellä toteutetun funktion ongelmaksi voi muodostua ajoitus. Funktio saatetaan ajaa ennen kuin dokumentin  DOM on rakennettu, ja näin dokumentista uupuu osia. Jos funktio sisältää DOM:sta puuttuvia elementtejä, se ei toimi.

Yleisimmät tavat jQuery-kirjaston käyttöönottoon ovat lataaminen ja tallentaminen lokaalisti koneeseen tai viittaaminen Content Delivery Network:n (CDN). Lokaalia tallennusta käytetään erityisesti sivuston kehitysvaiheessa. HTML-sivu tarvitsee viittauksen paikalliseen jQuery-kirjastoon, tämä kirjoitetaan HTML-sivun <head> elementin sisälle.

Viittaaminen CDN:n tapahtuu samalla tavalla <head> elementin sisällä, mutta kirjaston osoite on ulkoisella palvelimella. Palvelun tarjoajia on useita mm. Google, Microsoft ja jQuery. CDN:n hyöty on sen nopeus. Koska jQuery on erittäin suosittu kirjasto, on todennäköistä, että selain on tallentanut sen välimuistiin edellisillä käyttökerroilla.

```
<!DOCTYPE html>
<html>

  <head>

    <script src="//code.jquery.com/jquery-1.11.0.min.js"></script>
    <script src="polku/lokaalisti/tallennettuun/tiedostoon/jquery-1.11.0.min.js"></script>

    <title>jQuery viittaus</title>

  </head>
```

Kuvio 5 Viittaaminen jQuery-tiedostoon, ylempi CDN, alempi lokaali polku

Kun jQuery-kirjastoon viitatataan CDN:n avulla, protokollamäärittely kannattaa jättää pois. Tällä tavalla protokolla määräytyy automaattisesti (http tai https) ja vältytään sivun latausongelmilta.

## 2.3 AJAX

Vuonna 2005 Jesse Garrett julkaisi artikkelin Ajax: A New Approach to Web Applications. Se kuvasi uuden menetelmän HTML-sivujen lataukseen palvelimelta. Perinteises-

sä mallissa käyttäjä suoritti toiminnon, esimerkiksi lomakkeen täytön. Kun lomake oli valmis, käyttäjä painoi sivulla lähetä nappia ja selain lähetti pyynnön palvelimelle. Palvelin palautti uuden sivun, jonka selain latasi uudelleen koneelle. Käyttäjälle kokemus oli hidas ja kankea, jos esimerkiksi lomakkeentäytössä oli tapahtunut virheitä.

Uudessa menetelmässä muutoksen sisältänyt data ladattiin palvelimelta ja sivusto päivitettiin osittain ilman sivuston uudelleenlatausta, kuten Google oli toteuttanut Google Suggest ja Google Maps palveluissaan. Näin käyttäjäkokemus parani huomattavasti ja käytettävyys läheni työpöytäsovellusta. Tämän menetelmän Garrett nimesi AJAX-termiksi. Käsite muodostuu viidestä eri tekniikasta: (Garrett 2005.)

1. HTML ja CSS dokumentin esittämiseen
2. DOM dynaamisen sisällön muokkaamiseen ja esittämiseen
3. XML datan vaihtoon ja muokkaukseen (nykyään suositaan JSON -formaattia)
4. Asynkroninen datan nouto XMLHttpRequest olion avulla
5. JavaScript-kieli tekniikoiden yhteensitomiseen

Nimenä AJAX (Asyncronous JavaScript XML) on hieman harhaanjohtava, sillä sitä voidaan käyttää myös synkronisesti ja tiedonsiirtoon käytettävä formaatti on yleensä JSON. XML:n ja JSON:n lisäksi AJAX:lla voidaan lähettää ja vastaanottaa myös HTML- ja tekstitiedostoja (Mozilla Developer Network. 2014b).

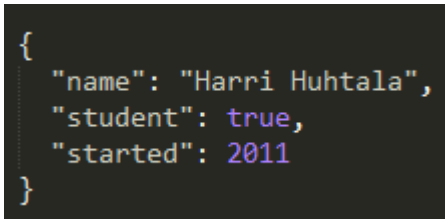
AJAX sisältää paljon hyötyjä: koko sivuston uudelleenlatauksen sijaan, haluttu alue dokumentissa muutetaan. Näin säilytetään yhtenäisempi käyttökokemus ja säästytään turhalta odottelulta. Kun palvelimelta pyydetään ainoastaan haluttua dataa, se näkyy pienempänä palvelinliikenteenä. Datan toimituksessa ei tarvitse enää lisätä HTML-elementtejä, jotta se voidaan näyttää käyttäjälle ja samaan aikaan kun dataa noudetaan palvelimelta, käyttäjä voi jatkaa toimintaansa sivustolla keskeytymättä. (Heilmann 2010.)

## 2.4 JSON

Douglas Crockford loi vuonna 2001 käsitteen JSON (JavaScript Object Notation). AJAX:n lailla tekniikkana JSON oli olemassa, mutta Crockford yhdisti nämä yhden termin alle. (Rauschmayer 2014, 334.)

JSON on kieli, joka on suunniteltu tiedonsiirtoon. Sitä käytetään vaihtoehtona XML-kielelle, jonka rakenne on raskas verrattuna JSON-syntaksiin. Keveyden lisäksi kielen vahvuuksiin kuuluu helppo luettavuus ja konventiot, jotka ovat tuttuja monista C-pohjaisista kielistä.

Kielellä on kaksi eri rakennetyyppiä: nimi/arvo (name/value) ja listat (arrays). Nimi/arvo on olio (object), joka sisältää nimi/arvo pareja .



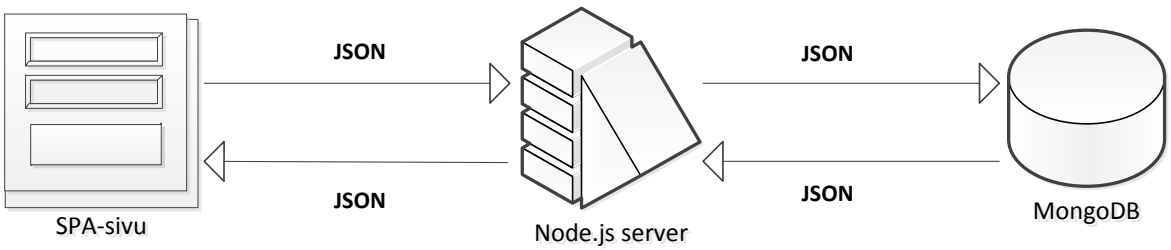

Kuvio 6 Olio, jonka sisällä nimi/arvo pareja JSON-formaatissa

Listat ovat kokoelma ominaisuuksia. Listat voivat sisältää String, number, olio, lista, boolean (true/false) ja null arvoja. (json.org.)


Kuvio 7 Olio, jonka sisällä lista (array) olioita JSON formaatissa

JSON-kielen suosio perustuu sen nopeuteen ja helppouteen käsitellä tekstiä JavaScriptin keinoin. JSON-objekti sisältää kaksi tärkeää metodia: JSON.parse() ja JSON.stringfy(). JSON.parse -metodia käytetään, kun JSON-teksti halutaan JavaScript olioksi ja JSON.stringfy() -metodi suorittaa päinvastaisen asian, kääntää JavaScript olion JSON-formaattiin. (Mozilla Developer Network. 2014c)

Kun asiakaskone (client) luo AJAX-kutsun, käännetään välitettävä data JSON-formaattiin. Palvelimelle saapuessa JSON-objekti käsitellään palvelimen kielellä, esimerkiksi JavaScriptillä. Käsittelyn ollessa valmis, palautetaan data JSON-formaatissa takaisin asiakaskoneelle, jossa se kootaan HTML-dokumenttiin. Jos palvelimena toimii Node.js ja tietokantana käytetään MongoDB:tä, tietotyyppi pysyy muuttumattomana (kuvio 8).




Kuvio 8 JSON-formaatissa olevan tiedon reitti palvelimelle, tietokantaan ja takaisin (Mikowski & Powell 2013, 267)

# 3 Palvelinpuolen JavaScript

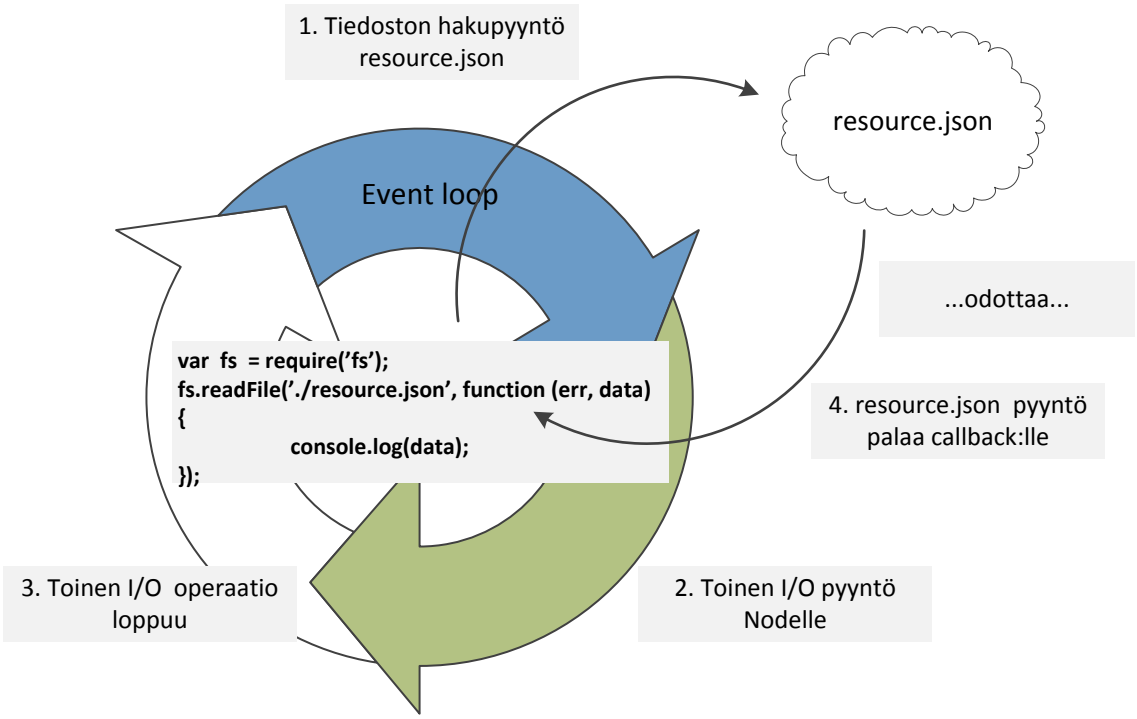
Vaikka JavaScript kehitettiin toimimaan palvelimen puolella, suorituskyky pakotti kielen siirtymään asiakkaan puolelle. Node.js julkaisu muutti kielen käyttömahdollisuuksia. Alustan avulla JavaScript-kieltä voitiin käyttää sekä asiakkaan että palvelimen puolella. NoSQL-kantojen kehitys, suosituimpana niistä MongoDB, muuttivat tietokantamallin rakennus- ja tallennustapaa.

## 3.1 Node.js

Node.js on alusta, jolla voidaan toteuttaa palvelinpuolen sovelluksia JavaScript-kielellä. Alusta perustuu Chrome-selaimen JavaScript V8-moottoriin. Ryan Dahl esitteli Node.js:n vuonna 2009 Berliinissä JSCONF konferenssissa. Se oli suunnattu skaalautuville web-sovelluksille, jotka tarvitsivat korkeaa suoritustasoa. Dahl valitsi JavaScript-kielen, sillä kielessä ei ollut valmista I/O (kirjoitus/luku) ohjelmointirajapintaa (API). Näin Dahl muokkasi asynkronisen, tapahtumavetoisen (event-driven) ja ei-estävän (non-blocking) mallin. Malli oli kevyt ja tehokas, joka sopi hyvin reaaliaikaisille web-sovelluksille. (Cantelon, Harter, Holowaychuk & Rajlich 2014, 3.)

Perinteinen palvelin (Apache) luo säikeen (thread) tai uuden prosessin, kun web-sivu pyytää siltä palvelua. Palvelin on yleensä nopea ja säe siivotaan pois resurssien noutamisen jälkeen. Kun palvelinta käyttää useampi käyttäjä, prosessit ja säikeet kuormittavat muistia, sillä tapahtumat käsitellään yhtä aikaa. Tämä tapa skaalautuu huonosti reaaliaikaisiin sovelluksiin.

Node.js käsittelee selaimen pyynnöt yhdessä säikeessä. Jos pyyntöjä saapuu useampi, palvelin ei luo uusia säikeitä. Pyynnöt käsitellään asynkronisesti nk. tapahtumakierteessä (event loop). Kun pyyntö saapuu palvelimelle, siihen kytketään nk. callback-funktio. Palvelin ei jää odottamaan funktion toteutumista, vaan siirtyy seuraavaan pyyntöön suorittamaan saman asian. Kun callback funktio on suoritettu, esimerkiksi haettava resurssi tietokannasta, pyyntö palautetaan selaimelle. Tällä tavalla Node-palvelin on erittäin tehokas muistinhallinnan kannalta. (Powers 2012, 14.)




Kuvio 9 Esimerkki ei-estävästä I/O tapahtumasta (Cantelon ym. 2014, 11)

Kuvio 9 havainnollistaa asynkronisen toiminnan. Selain on luonut Ajax-pyynnön resources.json-tiedostosta ja Node.js luo ensimmäisen I/O operaation (1.). Kun tiedosto palaa, laukaistaan callback-funktio, joka tulostaa tiedoston sisällön konsoliin (4.). Operaatio ei estä muiden toimintojen jatkumista. Tapahtumakierteessä (event loop) syntyy toinen I/O operaatio (2.) ja se suoritetaan loppuun, ennen kuin ensimmäinen operaatio päättyy (3.). Viimeisenä saapuu resource.json-tiedosto, joka viedään callback-funktiolle (4.). (Cantelon ym. 2014, 6.)

Node.js:n mukana tulee suosittu paketinhallintajärjestelmä npm (The Node Package Manager). Paketinhallintajärjestelmän avulla voidaan julkaista ja asentaa kolmannen osapuolen kehittämiä moduuleja. Moduuleista on tullut web-sovelluksien rakentamisessa arkipäivää ja npm -yhteisö tarjoaa monia laadukkaita paketteja. Pakettien asentaminen suoritaan helposti komentoriviltä.

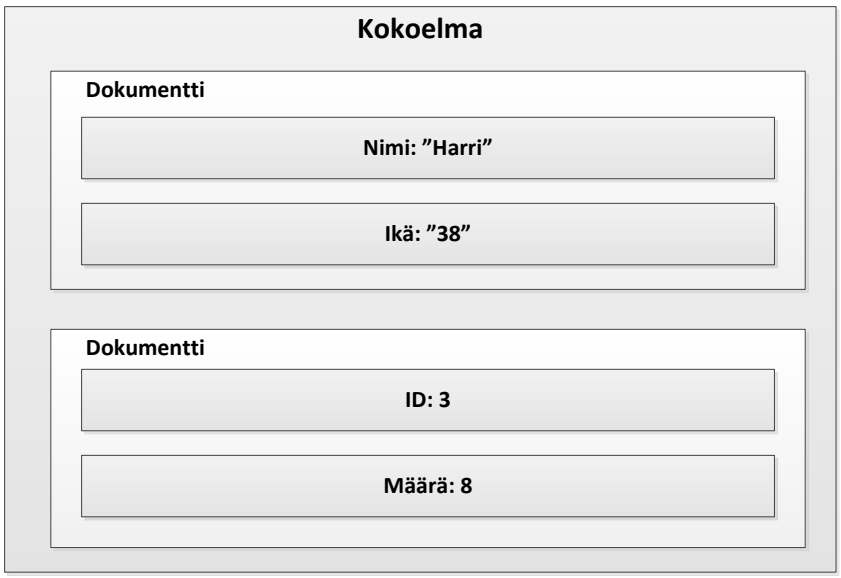
Node.js asema Full-stack JavaScript-kehityksessä on erittäin tärkeä. Ilman palvelinpuolen JavaScript-pohjaista API:a ei olisi mahdollista rakentaa koko arkkitehtuurin kattavaa

JavaScript web-sovelluksia. Node.js toimii alustana, joka sitoo yhteen sekä asiakas- että palvelinpuolen ja tarjoaa menetelmiä, joilla voidaan kommunikoida tietokannan kanssa. Npm-paketinhallintajärjestelmän kautta saadaan asennettua lukemattomia sovelluskehyksiä (mm. MongoDB, Express, AngularJS), joita hyödynnetään sovelluskehityksessä.

## 3.2 MongoDB

MongoDB on yksi suosituimmista NoSQL tietokannoista. Sen toiminta eroaa perinteisistä relaatiotietokannoista tiedon tallennusmallissa. MongoDB tallettaa tiedot kantaan kokoelmina (collections), joiden ei tarvitse noudattaa samaa mallia (schema). Tämä mahdollistaa erilaisten dokumenttien tallentamisen samaan kokoelmaan, mitä perinteisessä RDBMS kannassa ei voida toteuttaa.

MongoDB:n eduiksi voidaan luetella kolme ominaisuutta: dokumentti-tyylinen varastointi, dynaamiset skeemat (mallit) ja skaalautuvuus. MongoDB on suunniteltu skaalautumaan toimintakykynsä säilyttäen. Dokumentti-tyylisellä tallentamisella tarkoitetaan JSON-kielellä tallennettua dokumenttia. Jos tieto on lähetetty palvelimelle JSON-formaatissa, sitä ei tarvitse erikseen muuttaa tietokantaan sopivaksi. Tällä saavutetaan mm suorituskykyhyötyjä.




Kuvio 10 Kokoelma voi sisältää erilaisia dokumentteja (Cantelon ym. 2014, 117)

Dynaamisella skaalautuvuudella tarkoitetaan tietokannan kokoelmassa olevien dokumenttien eroja. Näiden ei tarvitse olla samanlaisia, vaan kokoelma voi sisältää eri attribuuteilla olevia dokumentteja (kuvio 10). Toisaalta skeeman puute voidaan nähdä negatiivisena ominaisuutena, sillä ne eivät ole tiukasti määriteltyjä. Tämä voi johtaa epäjohdonmukaisuuksiin, koska dokumentti voi sisältää paljon rakenteellisesti eroavaa tietoa. (Mikowski & Powell 2013, 268-269.)

# 4 Sovellusarkkitehtuuri

Ohjelmistokehityksessä on vaiheita, jotka toistuvat eri projekteissa. Ratkaisujen pohjalta voidaan luoda yleisiä malleja, joita sovelletaan toistuviin kehitysvaiheisiin. Näitä ratkaisuja kutsutaan suunnittelumalleiksi. Suunnittelumallien suurimmat hyötynäkökulmat voidaan jakaa kolmeen osaan: mallit ovat hyväksi todettuja ratkaisuja, malleja voidaan käyttää uudelleen ja mallien ratkaisut ovat selkeitä. (Osmani 2012, 3.)

## 4.1 MVC/MV*-suunnittelumallit

Suunnittelumalleilla määritellään ohjelman rakenne, kuinka eri asioita toteuttavat ohjelmalliset ominaisuudet organisoidaan sovelluksessa ja miten komponentit kommunikoivat keskenään. Tärkeimpinä malleina voidaan pitää MVC (Model-View-Controller)-mallia, ja sen pohjalta johdettuja MVP (Model-View-Presenter) ja MVVM (Model-View-ViewModel) -malleja. (Osmani 2012, 111.)

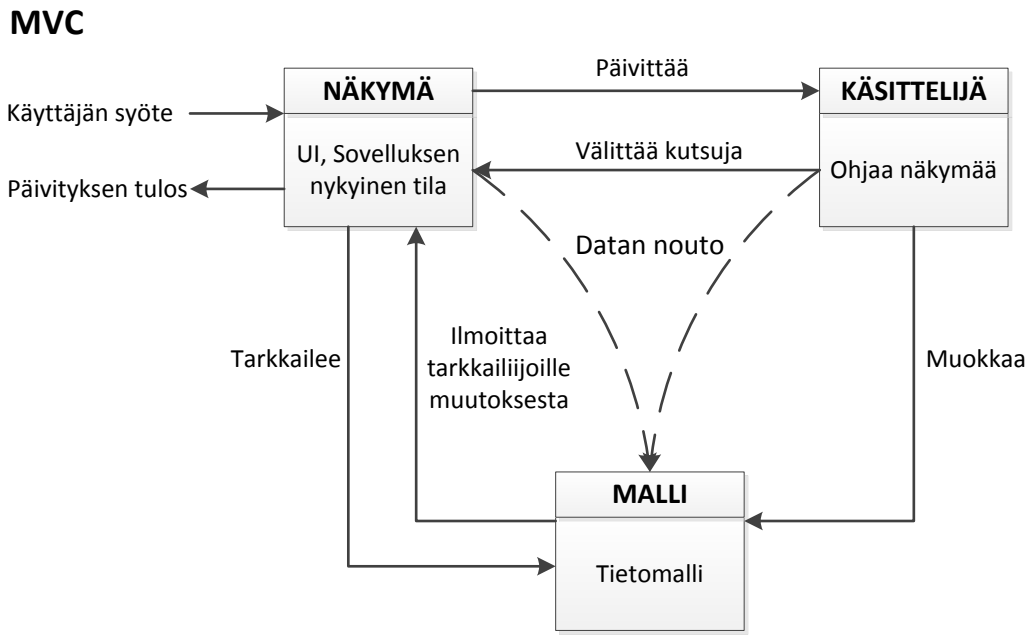
Termillä MV* viitataan MVC-mallin sukulaismalleihin. Modernit web-sovellukset eivät välttämättä noudata klassista MVC-mallia tiukasti, vaan käsittelijän rooli toteutetaan eri tavalla sovelluskehyksestä riippuen. JavaScript-pohjaiset sovelluskehykset, kuten AngularJS ja Ember, nojaavat vahvasti MV*- malleihin. (Osmani 2012b.)

Trygve Reenskaug laati MVC-mallin 1970-luvun lopussa Smalltalk-80 kielelle. Sen pääperiaate on ohjelmiston rakenteen eriyttäminen kolmeen eri komponenttiin: malliin (model), näkymään (view) ja käsittelijään (controller). Mallin tehtävänä on käsitellä sovelluksen tietoa. Se rakentaa tietomallin sovelluksen käyttämästä datasta. Mallin yhteys näkymään on eriytetty. Kun mallissa tapahtuu muutos, se ilmoittaa näkymälle muutoksesta ja näkymä reagoi tilan muutoksen päivittämällä näkymää. (Osmani 2012, 113.)

Näkymä tuottaa sovelluksen mallin pohjalta visuaalisen esityksen, käytännössä se tarkoittaa loppukäyttäjälle näkyvää HTML-sivua. Käyttäjät voivat lukea näkymää ja manipuloida sitä. Yleensä näkymää kuvaillaan ”tyhmäksi”, sillä sen tieto sovelluksen muista komponenteista on rajallinen. Näkymällä on tarkkailijoita, jotka saavat tiedon mallin muutoksen jälkeen. Vanhemmissa web-sovelluksissa näkymä vastasi tilanhallinnasta,

mutta JavaScript-pohjaiset yksisivuiset sovellukset määrittelevät tilanhallinnan eri tavalla johtuen sivun osittaisesta päivityksestä. (Osmani 2012, 115.)

Käsittelijän rooliksi jää toimia siltana mallin ja näkymän välillä. Kun käyttäjä haluaa muokata näkymää, näkymä välittää tiedon käsittelijälle ja se suorittaa mallin päivityksen. Malli ilmoittaa näkymän tarkkailijoita muutoksesta ja näkymä suorittaa päivityksen. (Osmani 2012, 117-118.)



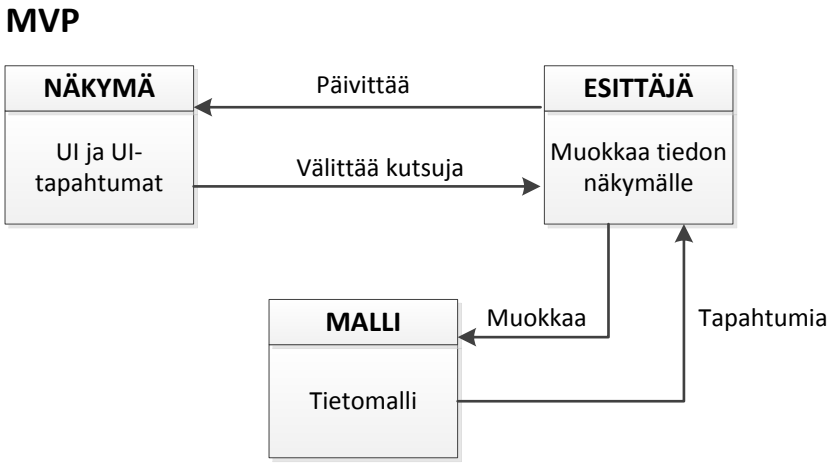

Kuvio 11 JavaScript-arkkitehtuurin mukainen MVC-malli (Osmani 2012, 113)

MVC-mallin mukainen komponenttien eriyttäminen selkeyttää ohjelmointia ja ohjelmiston ylläpidettävyys helpottuu. Jos ohjelmistossa syntyy tarve päivittää sovelluksen visuaalista puolta, löytyvät sovelluksen visuaaliset komponentit näkymän alta. Erotuksen avulla säästytään sovelluksen logiikan läpikäymiseltä ja muokkaamiselta. Myös logiikan turha toisto vähenee, kun samaa asiaa tekevää komponenttia voidaan käyttää useassa paikassa. Sovelluksen rakenteen ollessa modulaarinen, testien kirjoittaminen helpottuu. (Osmani 2012, 120.)

MVP-mallia kehiteltiin 1990-luvulla IBM toimesta Taligent-käyttöjärjestelmän yhteydessä. Taligent oli olio-ohjelmointipohjainen käyttöjärjestelmä, joka suunniteltiin kilpailijaksi NeXTSTEP käyttöjärjestelmälle. MVP poistaa käsittelijän roolin, käsittelijä kor-

vataan esittäjällä (presenter). Näkymä muuttuu passiiviseksi, joka tarkoittaa näkymän logiikan siirtämistä esittäjälle. Esittäjä huolehtii näkymän tapahtumien ohjaamisesta ja muokkaa sekä näkymää että mallia. (Fowler 2006.)

MVP-mallia käytetään yleensä monimutkaisissa yritystason sovelluksissa. Mallin avulla saadaan näkymää yksinkertaistettua logiikan siirtyessä esittäjän puolella. Kehitysvaiheessa tämä mahdollistaa ohjelmointilogiikan toteutuksen ilman näkymän olemassaoloa. (Osmani 2012, 122-123.)



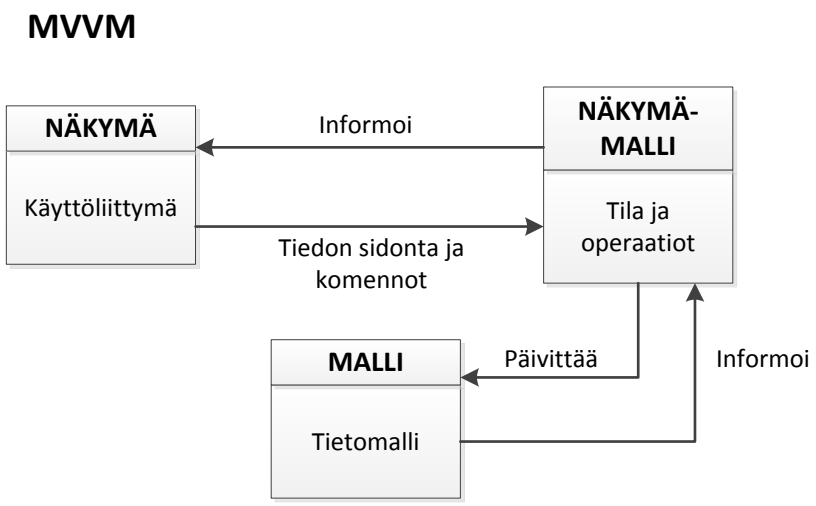

Kuvio 12 MVP-malli, Model-View-Presenter (Osmani 2012, 122)

MVVM-malli on Microsoftin vuonna 2005 nimeämä ratkaisu erottaa UI-kehitys ohjelmiston sovelluslogiikasta. Martin Fowler kirjoitti vuonna 2004 MVPM (Model-View-PresentationModel) -mallista, jonka pohjalta Microsoft rakensi MVVP-mallin, suurelta osin käyttäen samaa toteutustapaa. (Osmani 2012, 126.)

Mallin rooli on MVC mukaisesti toimia jäsenneltynä mallina ohjelmiston käyttämästä tiedosta. Malli ei ole vastuussa tietojen muokkaamisesta. MVVM-mallissa näkymä tuottaa käyttäjälle näkyvän HTML-sivun, mutta passiivisuuden sijaan, näkymää pidetään aktiivisena. Tämä tarkoittaa osittaista sovelluslogiikan sisältymistä näkymään.

Logiikan avulla tarkkaillaan tapahtumia ja suoritetaan tiedon sitomista (databinding), jonka avulla näkymä-malli (ViewModel ) manipuloi mallin tietoja. Näkymä-malli toimii sekä käsittelijänä että välimiehenä näkymän ja mallin välillä, sisältäen suurimman osan

molempien muokkauslogiikasta. MVVM ja MVC-mallin ero on näkymän ”tietoisuudesta” sovelluksen malliin. MVVM näkymällä ei ole suoraa yhteyttä malliin, jonka puolestaan MVC mahdollistaa. (Osmani 2012, 127-130.)




Kuvio 13 MVVM -malli (Microsoft Developer Network, 2012)

Kaikki kolme suunnittelumallia toteuttavat saman asian, sovelluksen komponenttien eriyttämisen. Jokainen suunnittelumalli lähestyy ongelmaa hiukan eri näkökulmasta, mutta toteutustavasta riippumatta arkkitehtuurin valinta ei tulisi näkyä loppukäyttäjälle.

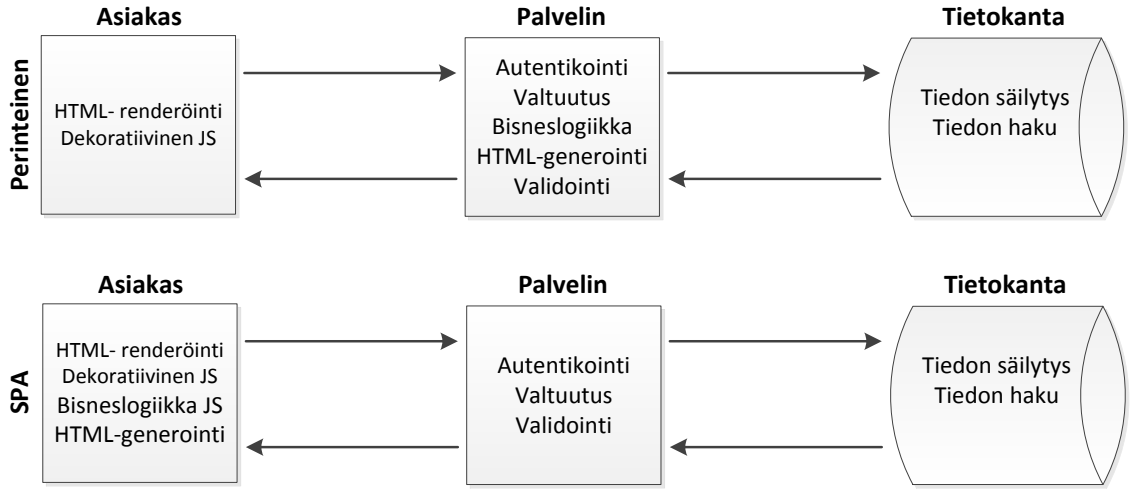
## 4.2 Single-Page Application

Perinteisessä web-sovelluksessa käyttäjän suorittamat pyynnöt (request) aiheuttavat koko sivun uudelleenlatauksen. Selain kokoaa sivun, kun HTTP-pyyntö saapuu asiakkaalle palvelimelta. AJAX:n käytön lisääntymisen myötä sivustojen osittainen päivittäminen muutti käyttäjäkokemusta. Web-sovelluksen rakenne pysyi kuitenkin samana. Yleensä sivusto muodostui useista HTML-dokumenteista ja toiselle sivulle siirryttäessä, aiheutui uudelleenlataus. Single-Page Application käsite muutti sovelluksen perinteistä rakennetta.

Single-Page Application (SPA) termiä käytetään web-sovelluksesta, jossa yksi HTML-sivu toimii kuorena sovelluksen kaikille sivuille. Kun sisältö muuttuu, selain ei lataa sivua kokonaan uudelleen, vaan päivittää sitä AJAX:n avulla. Sovelluksen on tarkoitus

toistaa työpöytäympäristössä käytettävän ohjelman ominaisuuksia käyttöympäristön ollessa selain. (Mikowski & Powell 2013, 4.)

SPA:ssa ei toteuteta täyttä Postback-menetelmää, jolla tarkoitetaan esimerkiksi lomakkeen täytössä tapahtuvaa tilanhallintaa ja sivun uudelleenlatausta. Kaikki täydet sivuston uudelleenlataukset uupuvat. SPA siirtää paljon perinteistä palvelinpuolen logiikkaa asiakkaan puolelle käyttämällä modernien sovelluskehyksien tarjoamaa front-end reititystä ja malleja (template). (Fink & Flatow 2014, 11)




Kuvio 14 Perinteinen sovellusarkkitehtuuri ja SPA arkkitehtuuri (Mikowski & Powell 2013, 8)

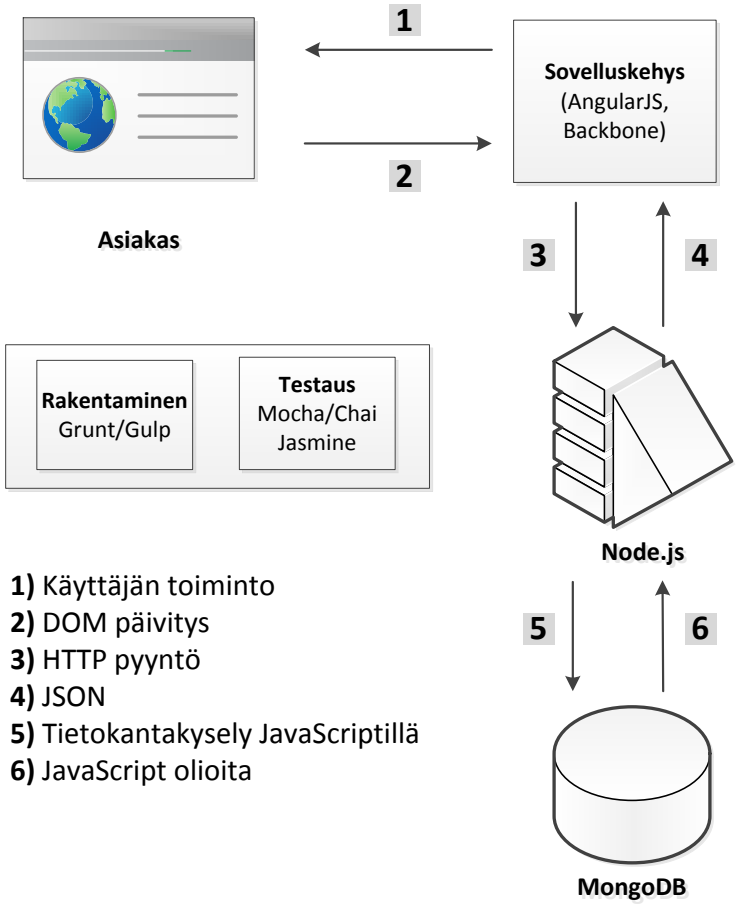
Kuvio 14 havainnollistaa, kuinka Single-Page Applicationin arkkitehtuurin rakenne eroaa perinteisestä mallista. Palvelimen bisneslogiikka ja HTML-sivujen generointi on siirretty selaimen toteutettavaksi. Näin vähennetään pyyntöjen lähettämistä palvelimelle ja kuorma pienenee. Turvallisuuteen liittyvät toiminnot kuten autentikointi, valtuutus ja validointi suoritetaan kuitenkin palvelimella. Tällä lisätään sovelluksen tietoturvaa.

### 4.3 Full-Stack JavaScript

Käsite Full-Stack JavaScript tarkoittaa web-sovellusta, jonka arkkitehtuuri on toteutettu yhdellä ohjelmointikielellä, JavaScript-kielellä. Ero perinteisen web-sovelluksen arkkitehtuuriin on komponenttien kielten määrässä. Perinteinen web-sovellus sisältää useampia ohjelmointikieliä ja kehittäjälle tämä tarkoittaa vähintään kahden eri kielen hallit-

semista. Kun sovellus kommunikoi eri komponenttien kanssa, täytyy kielille kirjoittaa muunnokset. Myös palvelin kuormittuu käännösprosessissa.

Elementit JavaScript-pohjaisten web-sovellusten synnylle luotiin 2000-luvun loppupuolella. Vuonna 2008 Google julkaisi V8 JavaScript-moottorin ja vuotta myöhemmin 2009 julkaistiin V8-moottoriin perustuva Node.js alusta. Tämä mahdollisti sovelluksen arkkitehtuurin sitomisen yhteen yhdellä kielellä ja samaa JavaScript-koodia voitiin käyttää sekä asiakas- että palvelinympäristössä. (Loukides, 2014).




Kuvio 15 Full-Stack JavaScript sovelluksen rakenne ja toiminnot (Storz 2013)

Kuvio 15 esittää Full-Stack-sovelluksen rakenteen. Kun asiakaspäässä käyttäjä luo toiminnon, sovelluskehys suorittaa DOM-manipuloinnin (1-2). Sovelluskehys irrottaa riippuvuussuhteen näkymän (View) ja mallin (Model) välillä. Mallin logiikka sijaitsee Node.js palvelimella. Kommunikointi palvelimen välillä (3-4) tapahtuu HTTP-protokollan avulla, käyttäen JSON-formaattia. Palvelin muodostaa JSON:sta JavaSc-

ript- olioita ja suorittaa niillä halutut toimenpiteet. Tietokantana toimii MongoDB, jonka tietokantiedustelut ja tallentamiset tapahtuvat JavaScript oliolla (5-6).

Testaukseen voidaan käyttää useita eri JavaScript-pohjaisia testikirjastoja, kuten Mocha/Chai tai Jasmine. Näillä voidaan rakentaa yksikkötestejä ja suorittaa regressiotestit jokaisen kehitysvaiheen jälkeen. Sovelluksen rakennus tapahtuu Grunt/Gulp automaatiotyökaluilla, jotka toimivat JavaScriptillä. Määrittelyistä riippuen, automaatiotyökalut suorittavat rakennusvaiheessa eri toimenpiteitä. Automaatiotyökalujen päätehtävä on noutaa riippuvuudet, rakentaa ja käynnistää sovellus.

# 5 Tutkimus

Tutkimuksessa rakennetaan prototyyppisovellus ja sen tärkeimmät kehitysvaiheet kuvataan. Valittujen komponenttien, kirjastojen ja sovelluskehyksien tehtävät kuvataan lyhyesti. Prototontyypin rakentamisella haetaan vastauksia tutkimuskysymyksiin.

## 5.1 Tutkimuskysymykset

Teoriaosuus pyrki vastaamaan kysymyksiin mitä tarkoitetaan käsitteellä Full-Stack JavaScript ja mistä yleisistä tekniikoista se rakentuu? Tutkivassa osuudessa rakennetaan sovellusprototyyppi ja tämän avulla haetaan vastauksia kysymyksiin:

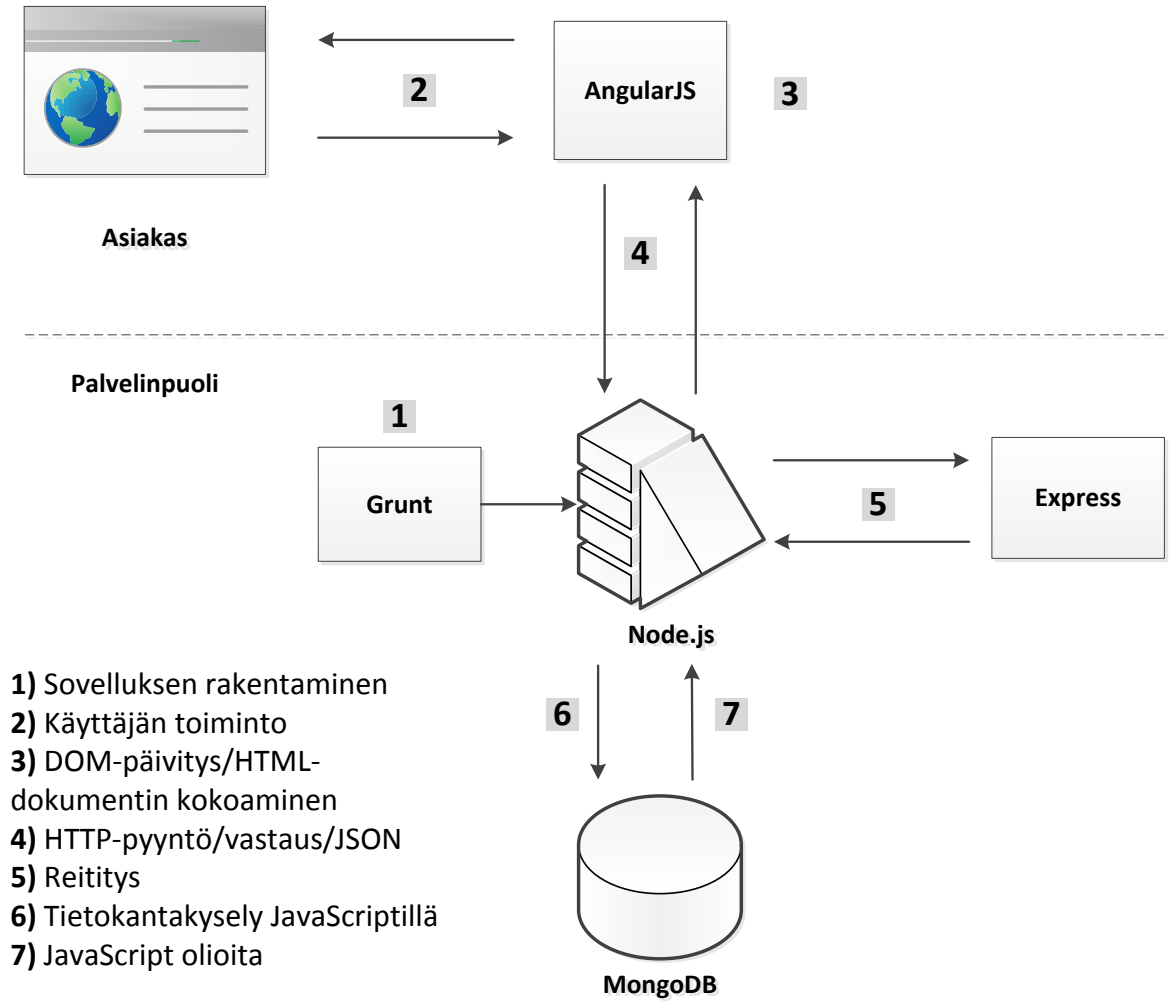
- Miten rakennetaan Full-Stack JavaScript-sovellus?
- Mitä komponentteja sovellukseen asennetaan ja mitkä ovat niiden yleisimmät tehtävät?

Ensimmäiseen kysymykseen vastataan kuvaamalla prototyypin rakentaminen tärkeimmät vaiheet. Toiseen tutkimuskysymykseen vastataan kertomalla prototyypin osien organisoituminen sovelluksessa ja kuinka ne kommunikoivat keskenään.

## 5.2 Prototyypin määrittely

Prototyyppi noudattaa kuvion 16 rakennetta. Asiakaspuolella MVC-malli toteutetaan AngularJS sovelluskehyksellä ja perinteistä palvelinpuolen logiikkaa siirretään sovelluskehyksen avulla asiakkaan puolelle. AngularJS avulla suoritetaan DOM -päivitykset/manipuloinnit ja HTML-sivujen kokoaminen asiakkaan puolella.

HTTP-pyyntö ohjataan Node.js palvelimelle, joka käsittelee ne. Palvelimen puolella ohjelma on jaoteltu myös MVC-arkkitehtuurin mukaisiin kansioihin, joka ovat erotettu asiakaspuolen logiikasta. Node.js:n päälle asennetaan Express, nk. middleware komponentti, joka toimii mm reitittimenä asiakkaan ja palvelimen välillä.




Kuvio 16 Prototyyppisovelluksen arkkitehtuuri ja komponentit

Tiedon säilyttämistä ja organisoimista varten valitaan MongoDB NoSQL tietokanta. Näin sovelluksen arkkitehtuurin läpi voidaan käyttää JSON-tietotyyppiä ja tietotyyppimuunnoksia ei tarvitse kirjoittaa toiselle kielelle.

Prototyypin ulkoinen rakenne tulee noudattamaan Singe-Page Applicationin mallia, sisältö näytetään yhdellä sivulla, eikä sitä jaeta hyperlinkeillä eri osiin. Sivuston ulkoasusta tulee responsiivinen ja tyylien esittämiseen käytetään Bootstrap-sovelluskehystä. Sovelluksen komponentit esirakennetaan Yeomanin työkaluilla käyttäen MEAN.JS generaattoria.

## 5.3 Yeoman ja MEAN

Yeoman on esirakennusohjelma, jolla rakennetaan kehys websovellukselle. Vastaava työkalu Java-ohjelmistokehityksessä on Maven. Yeoman sisältää kolme työkalua: Yo, Grunt ja Bower. Grunt on automaatiotyökalu, jolla automatisoidaan tehtäviä, kuten sovelluksen rakentaminen, lähdekoodin validointi ja sovelluksen testaaminen. Gruntilla hallinnoidaan myös sovelluksen riippuvuuksia.

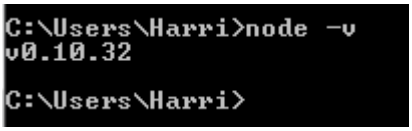
Bower on paketinhallintajärjestelmä, joka toimii asiakkaan puolella. Bowerin paketinhallintajärjestelmä ei ole rajoittunut pelkästään JavaScript-pohjaisiin kirjastoihin, sillä voidaan noutaa ja asentaa lähes kaikkia paketteja, kuten esimerkiksi Git:n repositoriosta.

Yo on komentorivipohjainen ohjelma, joka rakentaa web-sovelluksen perusrakenteen Gruntin ja Bowerin asetustiedostojen pohjalta. Yo tarjoaa useita eri malleja web-sovellusten toteuttamiselle generaattoreidensa avulla ja kirjoitushetkellä niitä oli yli 1100 kpl.

Yeomanin sisältää suositun Full-Stack JavaScript generaattorin nimeltään MEAN.JS. Generaattori lataa MEAN-paketin, joka on kokoelma ohjelmia JavaScript-pohjaisen web-sovelluksen rakentamiseen. MEAN-kokoelmaa voidaan kutsua vastineeksi LAMP:lle, joka on Linux-pohjainen ohjelmistokokoelma. MEAN on lyhenne sanoista MongoDB, Express, AngularJS ja Node.js. MEAN-pakan komponentit voidaan ladata erikseen ilman Yeomania, mutta Yeoman helpottaa riippuvuuksien hakemista ja sovelluksen rakentamista.

## 5.4 Prototyypin komponenttien asennus

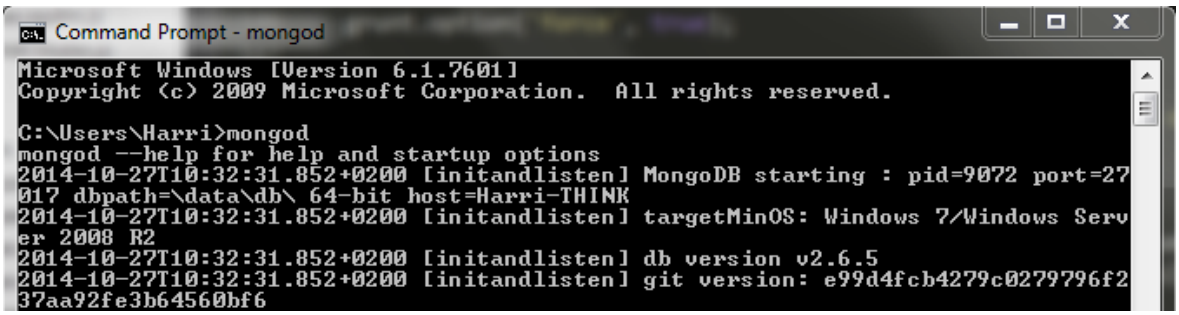
Toimiakseen MEAN -pakka tarvitsee Node.js -alustan ja MongoDB tietokannan esiasennettuna. Node.js asennus tapahtuu hakemalla tiedosto Node.js-sivustolta ja valitsemalla oikea käyttöjärjestelmäversio. Prototyyppi rakennetaan Windows 7 64-bit käyttöjärjestelmällä ja asennus tapahtuu Windows Installer (.msi) 64-bit tiedostolla käynnistämällä asennusopas. Node.js asennuksen onnistumisen voi tarkistaa kirjoittamalla komentokehotteeseen node -v, mikä tulostaa tiedon Node.js versiosta.




Kuvio 17 Node.js version tarkistus asennuksen jälkeen

MongoDB tietokannan asennus tapahtuu lataamalla asennustiedosto MongoDB kotisivuilta. Asennuspaketit ovat käyttöjärjestelmäkohtaisia. Prototyypin asennustiedosto on Windows Installer (.msi) 64-bit. Ennen tiedoston asentamista Windows 7 alustaan on asennettava Microsoftin tarjoama pikakorjaus, jolla korjataan muistiongelma.

Tietokanta tarvitsee paikan, jossa säilyttää kannan tietoja. Kansio luodaan C: polun alle, ja nimetään data\db . MongoDB:n käynnistystiedosto mongod.exe kannattaa lisätä käyttöjärjestelmän ympäristömuuttujaan, jolloin tätä voidaan ajaa komentokehotteessa globaalisti, projektin kansioiden ulkopuolelta. Lisääminen tapahtuu kirjoittamalla ympäristömuuttujaan MongoDB:n bin kansion sijainti: \;C:\Program Files\MongoDB 2.6 Standard\bin\

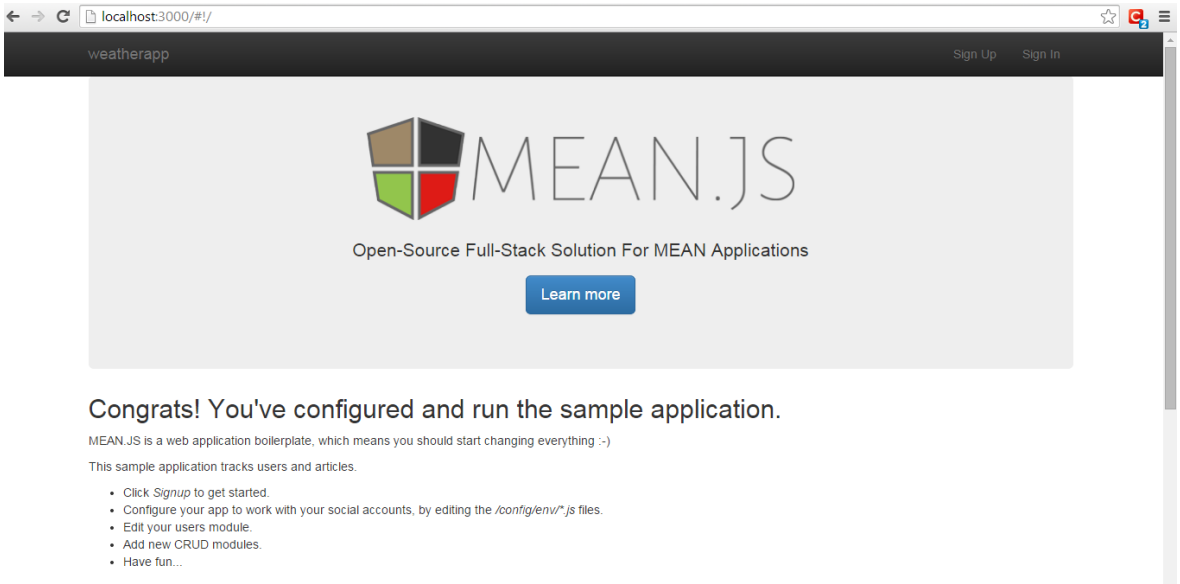



Kuvio 18 Kun mongod.exe on lisätty ympäristömuuttujaan, voidaan tietokanta käynnistää globaalisti

Yeoman asennetaan Node.js paketinhallintajärjestelmän npm kautta. Komentokehotteeseen kirjoitetaan: npm install –g yo. Paketinhallintajärjestelmä noutaa ja asentaa Yeomanin komponentit . Komponenttien uudelleenkäytettävyyden vuoksi ne kannattaa asentaa globaalisti (-g valitsin), näin operaatio täytyy suorittaa vain kerran. Kun asennus on valmis, voi onnistumisen tarkistaa kirjoittamalla komentokehotteeseen: yo –v. Komento tulostaa Yo versionnumeron.

MEAN.JS generaattori on versio MEAN-pakasta, jonka on kirjoittanut Yeomanin kehitystiimi. Generaattori asennetaan npm kautta kirjoittamalla komentokehotteeseen: npm install –g generator-meanjs. Prototyypille luodaan kansio ja kansion sisällä kirjoitetaan komento: yo meanjs. Komento käynnistää Yeomanin asennusohjelman Mean.js generaattorilla. Generaattori kysyy sovelluksesta perusasetustiedostoja ja vastausten jälkeen npm ja Bower noutaa kaikki riippuvuudet.

Riippuvuuksien noudon jälkeen prototyypin kansioon on luotu sovelluksen runko, joka sisältää varsinaisen MEAN-pakan ja paljon kehitystyötä helpottavia komponentteja. Sovelluksen toimivuutta voi testata kirjoittamalla komentokehotteeseen projektin kansiossa komennon: grunt. Grunt suorittaa määritellyt tehtävät ja käynnistää palvelimen. Tämän voi tarkistaa kirjoittamalla selaimeen http://localhost:3000 ja selaimessa pitäisi näkyä MEAN.JS generoima oletussivu. Jotta sovellus voi käynnistyä, täytyy MongoDB:n olla päällä.

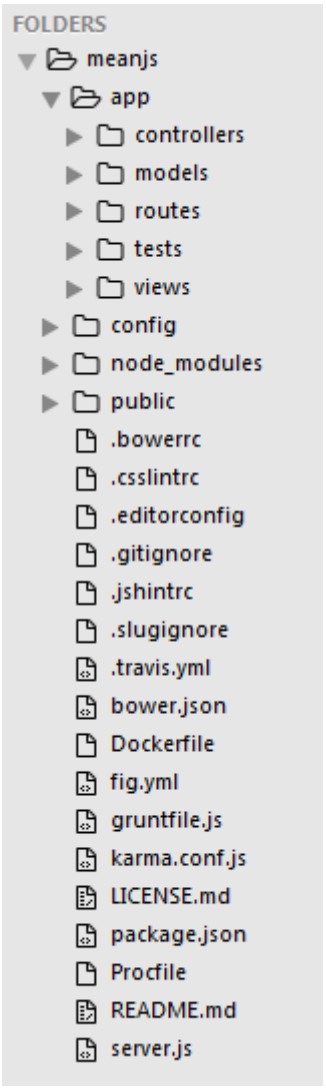



Kuvio 19 Mean.js generoima aloitussivu kun palvelin on käynnistetty.

## 5.5 MEAN.JS-pakan kansiorakenne

MEAN.JS-pakka luo projektille automaattisesti kansiorakenteen. Kansioiden organisointi toteutetaan MVC-mallin mukaisesti. Ne voidaan jakaa neljään peruskansioon: app, config, node_modules ja public. Projektin juureen on koottu tiedostoja, joiden

avulla voidaan määritellä esimerkiksi sovelluksen kaikki pakettiriippuvuudet (package.json) ja Gruntin suorittamat tehtävät (gruntfile.js).



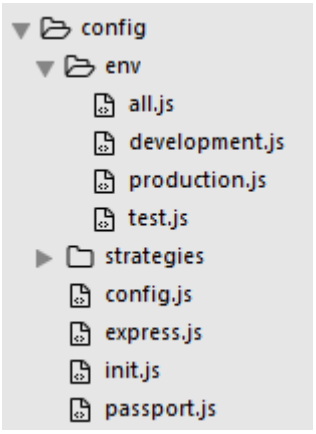

Kuvio 20 MEAN.JS avulla luotu kansiorakenne, ylimpänä projektin nimikansio meanjs.

Ensimmäinen kansio projektin sisällä on nimeltään app (kuvio 20). Se on jaettu viiteen alikansioon: controllers, models, routers, tests ja views. Näiden kansioiden sisällöstä muodostuu palvelinpuolen lähdekoodi. Palvelinpuoli toteutetaan MVC-mallia noudattaen.

Kansio controllers sisältää reitittäjän (Express) viittaamat käsittelijät (controllers), models-kansio sisältää tietokannan mallien kuvaukset ja validoinnit, routes-kansiossa on palvelimen reititykseen tarvittavat tiedot, tests-kansiossa on sovelluksen testit ja view-

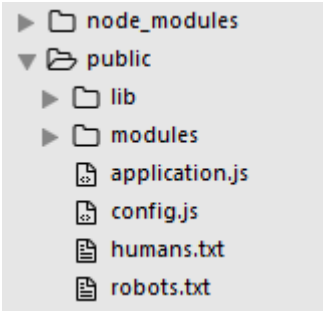
kansio sisältää sovelluksen aloitussivun rungon, index.server.view.html. Asiakaspuolen view-kansio sisältää suurimman osan sovelluksen näkymistä.

Seuraava pääkansio on nimeltään config (kuvioi 21). Se sisältää sovelluksen asetustiedostoja, joista tärkein on config.js. Tiedoston pohjalta ladataan oikea NODE_ENV, millä määritellään onko ympäristö kehitystä, tuotantoa vai testausta varten. Ympäristön määritykset kuvaillaan config/env-kansiossa. Jokainen ympäristö sisältää omat määritykset sovellukselle ja näin se antaa kehittäjälle lisää liikkumatilaa. Asetustiedostojen alustaminen tapahtuu init.js-tiedostossa.




Kuvio 21 config-kansio avattuna

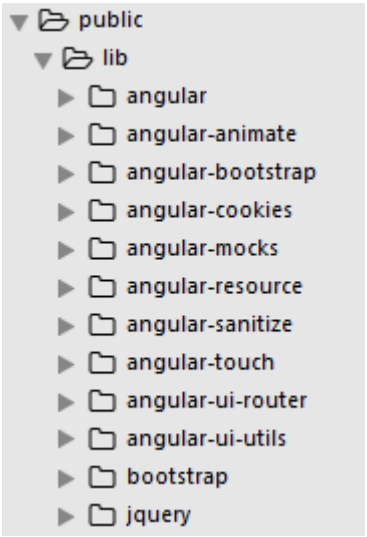
Kaksi seuraavaa pääkansiota (kuvio 22) ovat nimeltään node_modules ja public. Node_modules-kansio sisällä ovat kaikki npm-moduulit. Moduulit ladataan package.json -tiedoston perusteella, kuten esimerkiksi Express, Grunt ja Mocha.




Kuvio 22 node_modules ja asiakaspuolen lähdekoodin sisältävä public-kansio.

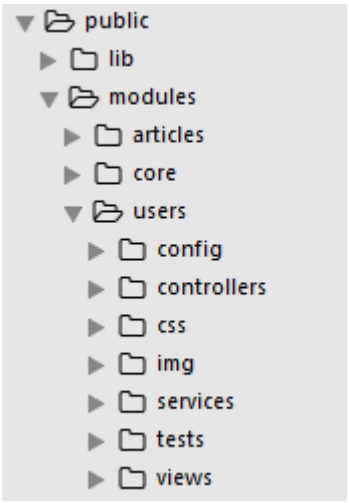
Public–kansio sisältää asiakaspuolen lähdekoodin ja AngularJS komponentit jäsenneltynä MVC-mallin mukaisesti. Public/lib-kansiossa (kuvio 23) sijaitsevat kaikki Bower paketinhallintajärjestelmän lataamat asiakaspuolen paketit.

Tärkein paketeista on AngularJS, joka sisältää varsinaisen AngularJS-sovelluskehyksen. Lisäksi lib-kansioon ladataan paketteja, jotka helpottavat kehitystä Angularin kanssa, kuten angular-cookies moduuli, jolla luodaan evästeitä AngularJS:n avulla. Angularista riippumattomia paketteja ovat bootstrap ja jquery. Public/lib–kansion on Bowerin osoittama ja halutessaan kansiolle voidaan määritellä eri polku .bowerrc–tiedostossa.




Kuvio 23 Bower paketinhallintajärjestelmän lataamat asiakaspuolen paketit sijaitsevat public/lib-kansiossa.

Public/modules/-kansion sisältö (kuvio 24) koostuu AngularJS:n käyttämistä moduuleista. Moduuleilla tarkoitetaan jäsennettyä lähdekoodia, joka tuottaa yhden sovelluksen komponentin, esimerkiksi sovelluksen käyttäjän (users) kuviossa 24. Angularin kehitys on vahvasti sidoksissa moduuleihin, näin ohjelmiston komponentit saadaan eroteltua omiin kokonaisuuksiin ja lähdekoodi selkeytyy.




Kuvio 24 Angularin käyttämät moduulit public/modules-hakemistossa

Users-kansion sisältö jaetaan seitsemään alikansioon (kuvio 24). Ensimmäinen kansio config sisältää moduulikohtaisia asetustiedostoja. Moduulin mallien (templates) reitityk-set määritellään täällä. Seuraava controllers-kansio määrittelee moduulin käsittelijät (controllers). Controllers-kansion sisällä ovat kaikki käsittelijään liittyvät funktiot, kuten esimerkiksi signup-funktio, jota käytetään käyttäjän luonnissa (kuvio 25 ).

```
10    $scope.signup = function() {
11        //lähettää tiedot /auth/signup reitittimelle, palvelimen puolella app-kansiossa
12        //$scope.credentials sisältää tiedot (nimi, salasana)
13      $http.post('/auth/signup', $scope.credentials).success(function(response) {
14        // If successful we assign the response to the global user model
15        $scope.authentication.user = response;
16
17        // And redirect to the index page
18        $location.path('/');
19      }).error(function(response) {
20        $scope.error = response.message;
21      });
22    };
```

Kuvio 25 AuthenticationController users-moduulissa, jota kutsutaan uuden käyttäjän luonnin yhteydessä

Css-kansiossa sijaitsee moduulin tyylitiedot ja img-kansiossa moduulin liittyvät kuvat. Services-kansio sisältää Angularin services-paketit. Service on toiminnallisuus, joka ei istu MVC-mallin määritteisiin, sen käyttäminen koskee useampaa komponenttia. Test-kansioon laaditaan moduulin Jasmine-testit ja viimeinen kansio views sisältää kaikki moduuliin liittyvät HTML-tiedostot.

Projektin varsinaisen kansiorakenteiden alle jää 17 tiedostoa (kuvio 20). Tärkeimmät tiedostot ovat nimeltään server.js ja gruntfile.js . Server.js (kuvio 26) suorittaa sovelluksen alustuksen. Sen tehtävänä on käynnistää tietokantayhteys (rivit 15-20), alustaa Express reitityssovelluskehys (rivi 23) ja käynnistää sovellus määriteltyyn porttiin (rivi 29). Tiedoston alussa saadaan viittaus palvelinpuolen asetustiedostoihin (rivit 5-6), jotka sijaitsevat ./config-kansion sisällä. Asetustiedostoissa määritellään muun muassa portti.

```
1  'use strict';
2  /**
3   * Module dependencies.
4   */
5  var init = require('./config/init')(),
6    config = require('./config/config'),
7    mongoose = require('mongoose');
8
9  /**
10  * Main application entry file.
11  * Please note that the order of loading is important.
12  */
13
14 // Bootstrap db connection
15 var db = mongoose.connect(config.db, function(err) {
16   if (err) {
17     console.error('\x1b[31m', 'Could not connect to MongoDB!');
18     console.log(err);
19   }
20 });
21
22 // Init the express application
23 var app = require('./config/express')(db);
24
25 // Bootstrap passport config
26 require('./config/passport')();
27
28 // Start the app by listening on <port>
29 app.listen(config.port);
30
31 // Expose app
32 exports = module.exports = app;
33
34 // Logging initialization
35 console.log('MEAN.JS application started on port ' + config.port);
```

Kuvio 26 server.js-tiedoston määrittelyt, jotka huolehtivat sovelluksen käynnistyksestä

Grunt.js–tiedostossa (kuvio 27) määritellään projektin automaatiota vaativat tehtävät. Tiedosto sisältää paljon kehitystyötä helpottavia tehtäviä (task). Gruntin avulla voidaan esimerkiksi tarkkailla tiedostoja ja niissä tapahtuvia muutoksia. Muutokset rekisteröi-

dään ja palvelin käynnistetään uudestaan. Jos selainikkuna on auki, se päivittyy uuden lähdekoodin mukaan.

Muita tärkeitä tehtäviä, joita voidaan gruntfile.js avulla suorittaa, on lähdekoodin validointi eri liitännäisten kautta. Gruntin avulla yksikkötestien ajaminen voidaan automatisoida. Kun sovellus on valmis tuotantoon, liitännäisten avulla voidaan suorittaa lähdekoodin pakkaus ja rakentaa erillinen kansio, joka sisältää tuotantoon tarkoitetun sovelluksen.

Kuviossa 27 riveillä 5 -12  luodaan olio, joka sisältää kaikki sovelluksen tiedostot ja kansiot joita Grunt tarkkailee. Rivillä 15 alustetaan Gruntin astutusolio, jonka sisällä sijaitsevat määritellyt tehtävät.

```
1  'use strict';
2
3  module.exports = function(grunt) {
4    // Unified Watch Object
5    var watchFiles = {
6      serverViews: ['app/views/**/*.*'],
7      serverJS: ['gruntfile.js', 'server.js', 'config/**/*.js', 'app/**/*.js'],
8      clientViews: ['public/modules/**/views/**/*.html'],
9      clientJS: ['public/js/*.js', 'public/modules/**/*.js'],
10     clientCSS: ['public/modules/**/*.css'],
11     mochaTests: ['app/tests/**/*.js']
12   };
13
14   // Project Configuration
15   grunt.initConfig({
16     pkg: grunt.file.readJSON('package.json'),
17     watch: {
18       serverViews: {
19         files: watchFiles.serverViews,
20         options: {
21           livereload: true
22         }
23       },
```

Kuvio 27 Osa gruntfile.js-tiedostosta

Rivillä 16 saadaan viittaus package.json riippuvuustiedostoon, joka sisältää npm-moduulit ja riveillä 17-23 määritellään watch-komento. Komennolla tarkkaillaan aikaisemmin luodun watchFiles-olion muutoksia. Kun muutos on havaittu rivillä 20-21, livereload lataa tiedoston ja selain päivittää sivun. Tämä ominaisuus on hyödyllinen sovelluksen kehitysvaiheessa, jolloin lähdekoodin muokkaaminen näkyy lähes välittömästi tai virheen sattuessa virheviesti tulostuu komentokehotteeseen.

## 5.6 Sääpäiväkirja

Prototyyppisovelluksesta rakennetaan yksinkertainen MEAN.JS-pakan päälle web-sovellus, jonne voidaan tallettaa ja muokata tietoa päivän säästä. Käyttäjän tulee olla luonut itselleen käyttäjätunnuksen ja kirjautunut sovellukseen sisälle. Tunnuksen luonti ja autentikointi tulevat MEAN.JS-pakan mukana. Kun käyttäjä on rekisteröitynyt, havaintojen kirjoittaminen tulee mahdolliseksi havaintosivulla. Prototyypin tarkoituksena on näyttää, kuinka JavaScriptillä toteutettu arkkitehtuuri toimii eri vaiheissa. Käyttökokemus jää tällöin taustalle.

Sovellus käynnistetään kirjoittamalla komentokehotteeseen mongod, joka käynnistää MongoDB-tietokannan. Kun MongoDB on ylhäällä, käynnistetään Grunt, kirjoittamalla grunt komentokehotteeseen. Grunt suorittaa sovelluksen käynnistyksen sen gruntfile.js-tiedoston tehtävien mukaan.

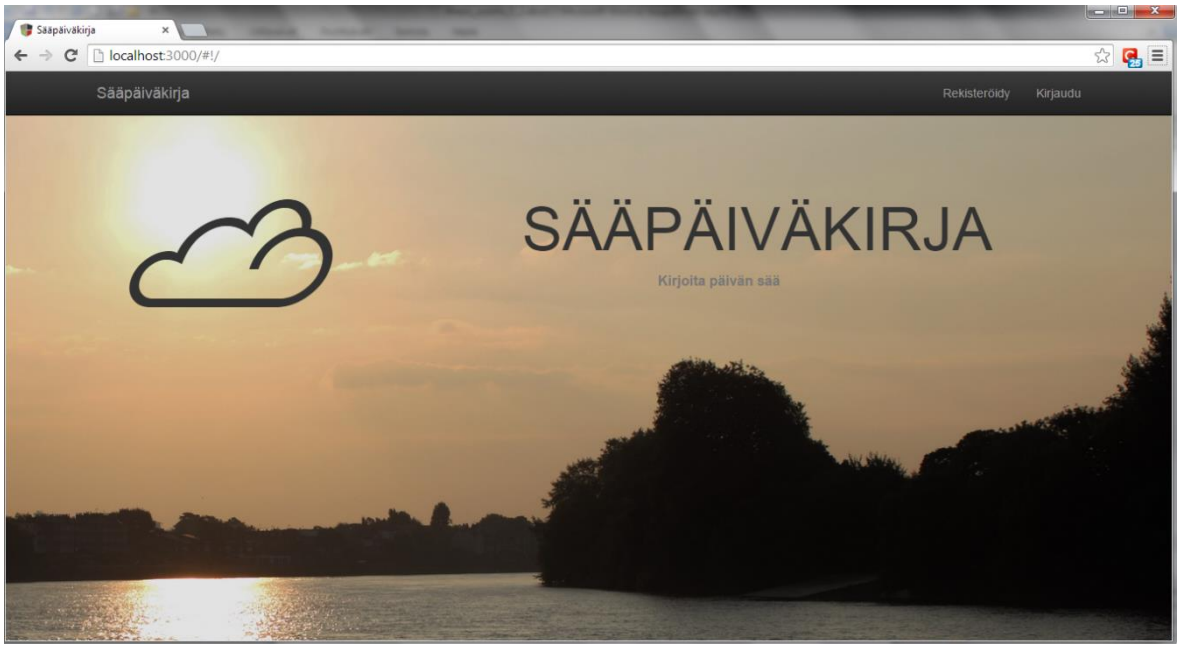
Kuviossa 28 on osa sovelluksen lähdekoodia, joka kuvaa reititystä sovelluksen käynnistyttyä . Palvelimelle saapuu get-pyyntö, joka reititetään app/routes-kansion sisällä olevaan core/server/routes .js-tiedostoon (ylempi kuvio). Tiedosto toimii sovelluksen aloitusreitittäjänä. Reititys tapahtuu app.route('/') funktiossa. Kaikki get-pyynnöt, jotka sisältävät /-viivan, saavat vastauksena index-sivun. Index–funktio sijaitsee controllers/core.server.controller.js-tiedostossa (alempi kuvio).

```
module.exports = function(app) {
  // Root routing
  var core = require('../../app/controllers/core');
  app.route('/').get(core.index);
};
```

```
exports.index = function(req, res) {
  res.render('index', {
    user: req.user || null
  });
};
```

Kuvio 28 core/server/routes .js (ylempi kuvio) ja controllers/core.server.controller.js-tiedostot

Funktio hyväksyy HTTP-pyynnön (request) ja HTTP-vastauksen (response). Funktio res.render kokoaa mallin (index) ja lisää muuttujan user sille varatulle paikalle mallissa. Sivu lähetetään asiakkaalle ja AngularJS kokoaa näkymän valmiiksi.

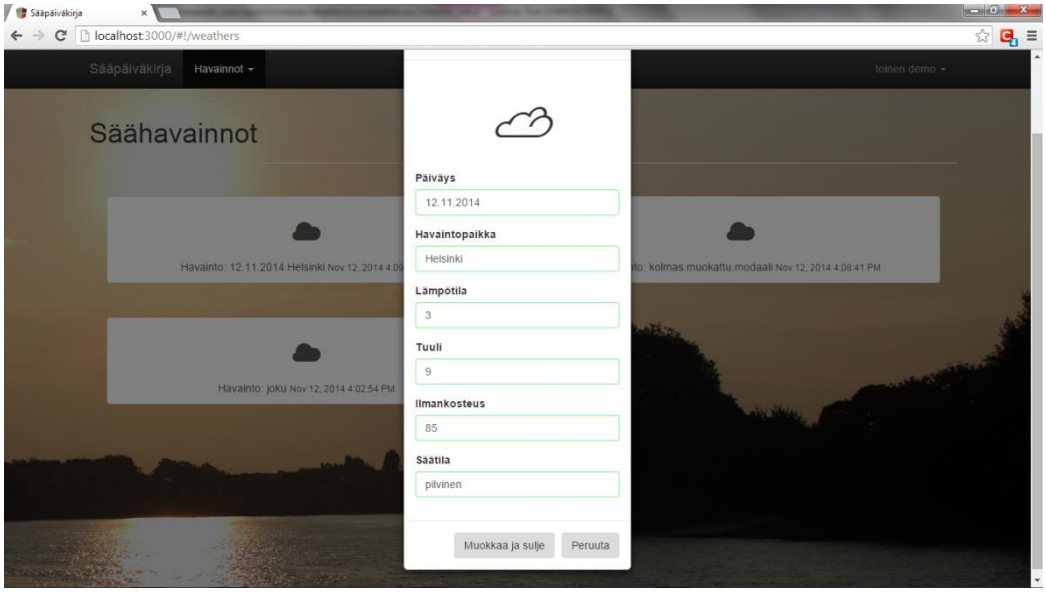



Kuvio 29 Prototyypin kotisivu

Pääsivu on toteutettu MEAN.JS generoiman rungon mukaan, navigointipalkki on luotu ylhäälle. Palkissa näkyy sovelluksen nimi, rekisteröidy- ja kirjaudu-linkki. Kun sovellukseen on kirjauduttu sisään, voidaan tallentaa/muokata havaintoja. Sovelluksen tiedonkulku arkkitehtuurissa kuvataan muokkaus-toiminnon avulla.

# 6 Tulokset

Prototyypin rakentamisella pyrittiin kuvaamaan, kuinka sovelluksen komponentit ovat sidoksissa toisiinsa ja kuinka ne kommunikoivat keskenään. Sovellukseen rakennettiin sivu, jossa käyttäjä tallentaa/muokkaa tietoja päivän säästä, tiedot vietiin tietokantaan ja näkymä päivitettiin. Muokkaa-toiminnosta laadittiin tapahtumaketju, joka kuvaa sovelluksen vaiheita toiminnon siirtyessä ketjussa eteenpäin.

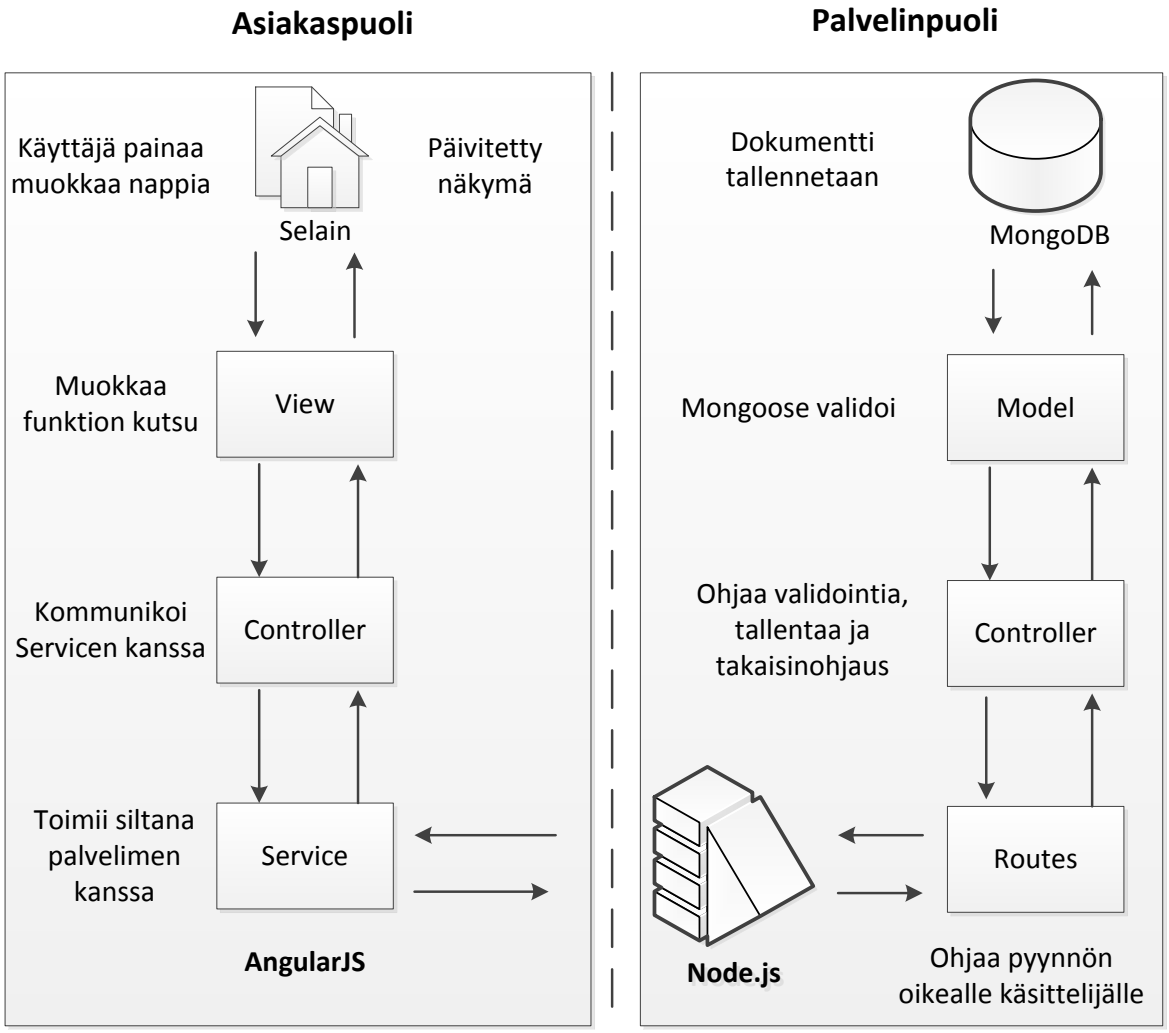



Kuvio 30 Havainnon muokkaus tapahtuu modaali-ikkunassa

Kirjautumisen jälkeen käyttäjä pystyy siirtymään havainto-sivulle. Täällä käyttäjä näkee tallennetut havainnot ja niitä painamalla aukeaa modaali-ikkuna, jossa voidaan muokata havaintotietoa. Kuviossa 31 havainnollistetaan, kuinka prototyypin komponentit kommunikoivat keskenään ja sovelluksen kansiorakenne käydään läpi tiedon muokkauksen yhteydessä.

Kun käytättäjä on muokannut havaintoa, käyttäjä painaa selaimessa muokkaa ja sulje nappia. Napissa sijaitsee AngularJS ng-click direktiivi (view-kansio), jonka sisällä oleva funktio kutsuu update-funktiota (controller-kansio). Funktio saa parametrina havainnon tiedot.

Asiakaspää yhdistetään palvelimeen service-kansiossa olevalla factory-metodilla. Metodi toimii REST-siltana käyttäjän ja palvelimen kanssa välittäen tiedon Node.js palvelimella olevalle routers-tiedostolle. Tiedosto reitittää kutsun URL:n mukaan oikealle controller-kansiolle.




Kuvio 31 Informaation kulku käyttäjän suorittaessa säätilan tallennuksen/päivityksen

Controller-kansio validoi tiedon Mongoosen avulla ja tieto tallennetaan MongoDB dokumenttiin. Controller palauttaa päivitetyn havainnon takaisin asiakaspäähän ja AngularJS päivittää näkymän selaimessa. Koko sovelluksen arkkitehtuuri käydään läpi muokkaa napin painamisesta tiedon tallentamisen kautta näkymän päivitykseen JavaScript-kielellä.

# 7 Johtopäätökset ja päätelmät

Tutkimuksen tarkoituksena oli perehtyä, miten rakennetaan Full-Stack JavaScript prototyyppi, mitä komponentteja sovellukseen asennetaan ja mitkä ovat niiden yleisimmät tehtävät. Prototyypin rakennukseen käytettiin Yeoman-esirakennusohjelmaa ja sen sisältämää MEAN.JS-generaattoria.

Yeoman havaittiin erittäin hyväksi esirakennusohjelmaksi. Ohjelma oli helppo asentaa ja ottaa käyttöön. Yeomanin sisältämien generaattoreiden määrästä löytyi useita vaihtoehtoja ja prototyypin rakentamiseen valittu MEAN.JS osoittautui laadukkaaksi rungoksi.

MEAN.JS tarjosi laajan esimääritellyn rungon ja generoi valmiiksi monia ominaisuuksia, joita yleisesti sovelluksien rakentamisessa käytetään, esimerkiksi rekisteröityminen. Näiden valmiiksi luotujen ominaisuuksien pohjalta oli helppo muokata niitä prototyyppiin sopivaksi. MVC-mallin mukaan luotu kansiorakenne pakotti lähdekoodin jakautumaan loogisesti sekä asiakkaan- että palvelimenpuolella. Asetustiedostojen konfiguroiminen JavaScript-kielellä lisäsi kehityksen helppoutta.

MEAN-pakka muodosti saumattomasti kommunikoivan kokonaisuuden. Pakan jokaisella komponentilla oli paikkansa ja sillä oli ketterää kehittää Full-Stack JavaScript sovellusta. AngularJS rakensi selkeän MVC-kokonaisuuden asiakkaan puolelle. Sivujen kokoaminen injektoimalla mallit (templates) kotisivun sisään, tuotti SPA sovelluksen, joka näennäisesti toimi useammalla sivulla.

Express-sovellusskehyksen avulla palvelimen puolen lähdekoodi jaettiin MVC-mallin mukaisesti ja sovelluskehys hoiti reitityksen. Tiedon tallentaminen MongoDB kantaan oli luontevaa, kun välissä oli Moongoose-sovelluskehyksen tarjoamat tarkistukset ja funktiot . Erityisesti tallennettavan tiedon muuttumattomuus toiseen kieleen oli ominaisuus, joka selkeytti kehitystä. Näin sovelluskehityksestä jäi yksi vaihe pois, joka omalta osaltaan kevensi palvelua.

Tärkein pakan komponenteista oli Node.js ja npm paketinhallintajärjestelmä. Node.js alustaa ja MongoDB tietokantaa lukuun ottamatta kaikki tarvittavat kehitystyökalut ladattiin npm kautta. Paketinhallintajärjestelmä tarjosi runsaan valikoiman kehitystyökaluja. Lisäksi näiden asennus oli helppoa. JavaScript-pohjaisten komponenttien sitominen yhteen tapahtui Node.js alustan avulla. Ilman tätä olisi ollut mahdotonta kehittää Full-Stack JavaScript web-sovellusta.

Web-kehityksen murroksen myötä JavaScript on saavuttanut täysiverisen ohjelmointikielen statuksen. Matka käyttöliittymän toiminnallisuuksien luonnista koko sovellusarkkitehtuurin kattavaan kieleen on ollut pitkä. Moni ”vakavasti otettavien” ohjelmointikielten kannattaja on yllättynyt kuinka moneen asiaan JavaScript-kieli on viime vuosina taipunut. Suurin kunnia tästä kuuluu Node.js alustalle ja sen synnyttämälle ekosysteemille.

## 7.1 Oma oppiminen

Päädyin opinnäytetyön aiheeseen kahdesta eri syystä. Ensimmäinen syy oli aiheen ajankohtaisuus web-sovelluskehityksessä tapahtuneen murroksen myötä. Toinen syy oli JavaScript-pohjaisten tekniikoiden ja sovelluskehyksien opettelu sekä taitojen syventäminen. HAAGA-HELIA:n tietojenkäsittelyn koulutusohjelmaan ei tällä hetkellä sisälly modernien web-sovelluskehyksien opetus, pääpaino on Java-kielessä ja Java-pohjaisissa back-end sovelluskehyksissä.

Aloitin opinnäytetyön kirjoittamisen keväällä 2014, mutta se keskeytyi viideksi kuukaudeksi suorittamani työharjoittelujakson ajaksi. Suoritin työharjoittelun Veikkauksella, missä roolini oli Front-End harjoittelija. Tämä kokemus oli erittäin arvokasta opinnäytetyötä ajatellen. Kun harjoittelu loppui syyskuussa, jatkoin projektia ja käytännössä kirjoitin opinnäytetyön alun uudelleen. Projekti lähti hyvin liikkeelle ja asettamani aikataulutavoitteet toimivat hyvin. Projektiin kului aikaa yhteensä kaksi kuukautta.

Opinnäytetyö on opettanut paljon eri asioita, analyyttista ajattelua, tiedon jäsentämistä teksti- ja kuva muodossa, web-sovelluskehityksen historiaa, ajankohtaisten lähteiden seulomista ja tietysti syventänyt laajasti kehitystyötä JavaScript-työkaluilla. Suunnittelu-

malleihin perehtyminen avarsi syitä, miksi erityylisiä suunnittelumalleja käytetään ja mitkä ovat niiden tehtävät. Node.js alustaan tutustuminen opetti miksi JavaScript-kieli ja sen komponentit skaalautuvat hyvin reaaliaikaisissa sovelluksissa ja miten on mahdollista rakentaa Full-Stack JavaScript sovelluskokonaisuus.

Tutkimuksessa rakennettuun prototyyppiin käytettiin useita työkaluja ja sovelluskehyksiä, jotka olivat aikaisemmin tuntemattomia, esimerkiksi AngularJS ja MEAN.JS-generaattorin projektikehys. Näihin tutustuminen selitti kehyksien tehtävät web-sovelluksessa. Prototyypissä toteutettu säätilan muokkaus ja sen pohjalta luotu kuva tarkensi, kuinka sovelluksen komponentit kommunikoivat keskenään ja kuinka tieto liikkuu sen sisällä. Kun nämä kaikki palaset liitettiin yhteen, rakentui Full-Stack JavaScript web-sovellus.

# Lähteet


Bueno, C. 2010. The Full Stack, Part I. Luettavissa: https://www.facebook.com/note.php?note_id=461505383919. Luettu: 8.10. 2014.

Cantelon, M. Harter, M. Holowaychuk, T.J. & Rajlich, N. 2014. Node.js in action. Manning Publications Co. NY.

Fink, G. & Flatow, I. 2014. Pro Single Page Application development: Using Backbone.js and ASP.NET. Apress Media LCC. New York.

Flanagan, D. 2011. JavaScript: The Definete Guide. Sixth Edition. O´Reilly Media. Sebastopol.

Fowler, M. 2006. GUI architectures. Luettavissa: http://martinfowler.com/eaaDev/uiArchs.html. Luettu: 31.10.2014.

Garret, J. 2005. Ajax: A New Approach to Web Applications. Luettavissa: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/. Luettu: 30.9.2014.

Heilmann, C. 2010. Developing Sites With AJAX: Design Challenges and Common Issues. Luettavissa: http://www.smashingmagazine.com/2010/02/10/some-things-you-should-know-about-ajax/. Luettu: 1.10.2014.

Hernan, D. 2012. Effective JavaScript. Addison-Wesley. Boston.

json.org. Introducing JSON. Luettavissa: http://json.org/. Luettu: 2.10.2014.

Loukides, M. 2014. Full-stack developers. Luettavissa: http://radar.oreilly.com/2014/04/full-stack-developers.html. Luettu: 8.10.2014.

Microsoft Developer Network, 2012. The MVVM Pattern. Luettavissa: http://msdn.microsoft.com/en-us/library/hh848246.aspx. Luettu: 2.11.2014.

Mikowski, M. S. & Powell, J.C.2013. Single Page Web Applications JavaScript end-to-end. Manning Publications Co. Greenwich.

Mozilla Developer Network. 2014a. JavaScript. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript. Luettu: 14.1.2014.

Mozilla Developer Network. 2014b. Ajax, getting started. Luettavissa: https://developer.mozilla.org/en-US/docs/AJAX/Getting_Started. Luettu: 30.9.2014.

Mozilla Developer Network. 2014c. Using native JSON. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_native_JSON. Luettu: 6.10.2014.

Mulder, P. 2014. Next-generation Web apps with full stack JavaScript. Luettavissa: http://radar.oreilly.com/2014/07/next-generation-web-apps-with-full-stack-javascript.html. Luettu: 15.10.2014.

Osmani, A. 2012. Journey Through The JavaScript MVC Jungle. Luettavissa: http://www.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/. Luettu: 29.10.2014.

Osmani, A. 2013. Learning Design Patterns. O´Reilly Media. Sebastopol.

Powers, S. 2012. Learning node. O´Reilly Media. Sebastopol.

Rauschmayer, A. 2014. Speaking JavaScript. O´Reilly Media. Sebastopol.

Stayer, R. 2013. Learning jQuery: A Hands-on Guide to Building Rich Interactive Web Front Ends. Addison-Wesley Professionals. Boston.

Storz, E. 2013. An introduction to Full-Stack JavaScript. Luettavissa: http://www.smashingmagazine.com/2013/11/21/introduction-to-full-stack-javascript/. Luettu: 16.10.2014.

Zakas, N. C. 2009. Professional JavaScript for Developers. Second Edition. Wiley Publishing. Indianapolis.

Web Education Community Group Wiki. 2012. A Short History of JavaScript. Luettavissa: http://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript. Luettu: 3.2.2014.

W3schools.com. HTML DOM nodeType Property. Luettavissa: http://www.w3schools.com/jsref/prop_node_nodetype.asp. Luettu: 26.9.2014.