Antti Salo

Model-View-Controller-arkkitehtuuri JavaScriptissä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Mediatekniikan koulutusohjelma

Insinöörityö

22.5.2015

Metropolia

Tekijä Otsikko Sivumäärä Aika	Antti Salo Model-View-Controller-arkkitehtuuri JavaScriptissä 18 sivua + 1 liite 22.5.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaajat	Päällikkö Risto Kyyrö Lehtori Aarne Klemetti

Insinöörityössä tehtiin Model-View-Controller-arkkitehtuuria noudattava sovellus, joka suoritetaan selaimessa. Toteutus tehtiin käyttäen JavaScript-ohjelmointikieltä ja sovelluskehystä nimeltä AngularJS. Työssä ei käyty läpi muita samaan suunnittelumalliin nojaavia sovelluskehyksiä.

Työssä perehdyttiin MVC-arkkitehtuurin historiaan ja valotettiin sen teoreettisia toimintaperiaatteita. Tutustumalla suunnittelumallin periaatteisiin palvelinkehityksen maailmassa luotiin pohja haasteista, joita MVC-mallin tuominen selaimeen aiheuttaa. MVC-mallin toteutumiseen paneuduttiin AngularJS-nimisen sovelluskehyksen näkökulmasta. Sen rakenteeseen ja toimintaperiaatteisiin tutustuttiin ja ne selitettiin korkealla tasolla. Lisäksi selitettiin sovelluskehyksen rakenteen yhteneväisyydet perinteiseen MVC-malliin. Insinöörityössä toteutettiin käytännön sovellus käyttäen AngularJS-sovelluskehystä. Sovellus toteutettiin televisioyhtiölle.

AngularJS:n todettiin kuvantavan MVC-mallia osittain löyhästi todellisen Mallin puutteen vuoksi. Sovelluskehyksen rakenteesta löydettiin osia, joiden voidaan tulkita kuvantavan Mallia niin, kuin se on mahdollista selaimen kontekstissa.

Selainpuolen kehityksessä AngularJS:n todettiin olevan erinomainen työkalu, jolla voidaan nopeuttaa sovellusten kehitystä. MVC-arkkitehtuurin mallinnuksen katsottiin hyödyttävän enemmän projektin rakenteen suunnittelua ja hallintaa kuin itse sovelluksen kehitystä selaimessa. Lisäksi todettiin kehittäjällä olevan lukuisia tapoja lähestyä sovelluskehitystä AngularJS:n kanssa.

Insinöörityöprojektissa toteutetun sovelluksen todettiin helpottavan käyttäjien työtä, ja se koettiin miellyttäväksi käyttää. Projektin suunnittelussa ei kuitenkaan onnistuttu ottamaan huomioon loppukäyttäjien toiveita, mikä vaikeutti projektin saattamista toimintavalmiuteen. Sovellusta käytetään edelleen.

Avainsanat	MVC, model-view-controller, JavaScript, AngularJS, SPA

Author	Antti Salo
Title	Model-View-Controller-arkkitehtuuri JavaScriptissä
Number of Pages	18 pages + 1 appendice
Date	22 May 2015
Degree	Bachelor of Engineering
Degree Programme	Media engineering
Specialisation option	Digital media
Instructors	Risto Kyyrö, Manager Aarne Klemetti, Senior lecturer

This thesis describes the design of an application following the Model-View-Controller design pattern. In practice, the design was implemented using JavaScript programming language and a frontend framework called AngularJS. Other frameworks expressing the MVC design pattern are not evaluated in this thesis.

The history of the MVC-architecture and its theoretical principles are discussed in this thesis. The challenges of bringing the MVC pattern to the browser are described in the context of server side development. AngularJS was used as the point of view in implementing the MVC pattern in the browser. The basic principles of the structure of AngularJS and functional principles were familiarized and are explained briefly. The resemblance of the implementation of the MVC pattern in AngularJS is also explained in comparison to the traditional MVC pattern. In practice, an application was designed and implemented for a Finnish television company using AngularJS.

Due to the lack of a real Model AngularJS was discovered to loosely implement the MVC pattern. Parts of the frontend framework were found to represent the Model in MVC pattern in the way it is possible in the context of the browser.

AngularJS was found to be an excellent tool to speed up browser side development. The implementation of the MVC architecture was viewed to be beneficial more to the structuring of the project than the actual development in the browser. Additionally it was stated that with AngularJS the developer has multiple ways to view application development.

The application that was built in the course of the final year project was seen to ease the end user's work and the application was viewed as pleasant to use. However the wishes of the end user failed to be recognized. This made it difficult to finish the process. However, the application is still in use.

Keywords	MVC, model-view-controller, JavaScript, AngularJS, SPA

Sisällys

Lyhenteet

dynaamisesti päivitettävä	Staattisen vastakohta. Sivun tietoja voidaan päivittää käyttäjän syötteiden mukaan sivulatauksen jälkeen.
Singleton-pattern	Ohjelmistotekniikan suunnittelumalli, jonka mukaan luokasta voi kerrallaan olla käytössä vain yksi instanssi
HTTP	Hyper Text Transfer Protocol. Protokolla, jonka avulla selain keskustelee palvelimen kanssa.
natiivisovellus	Käyttöjärjestelmän omalla tavalla toteutettu sovellus, joita kehitetään yleensä rajatuilla sovelluskehyksillä
RIA	Rich Internet Application. Sovellus, joka toiminnaltaan muistuttaa natiivisovellusta.
SPA	Single Page Application. Yhden http-sivun puitteissa toimiva sovellus.
AJAX	Asynchronic JavaScript and XHR. JavaScriptin avulla toteutettu asynkroninen HTTP-pyyntö
asynkroninen	Synkronisen vastakohta. Pyyntö, joka suoritetaan muun sovelluksen jatkaessa toimintaansa.
DOM	Document Object Model. Verkkosivun rakenteen mallinnus.
bootstrap	Alustustiedosto tai malli, jonka perusteella sovellus saa oletusasetuksensa.
HTML	Hyper Text Markup Language. Yleisin DOM-rakenteen kuvauskieli.
JSON	JavaScript Object Notation. Tietorakenteen kuvauskieli.

1 Johdanto

Insinöörityön tavoitteena on tarkastella Model-View-Controller-suunnittelumallin käyttämistä selainpuolen sovelluksen suunnittelussa. Tarkoituksena on perehtyä MVC-mallin historiaan ja toimintaperiaatteisiin teoriatasolla. Lisäksi työssä paneudutaan AngularJS-sovelluskehyksen toimintaan ja siihen, miten MVC-malli ilmentyy selainpuolen kehyksessä. Sovelluskehystä käytetään lopuksi MTV Oy:n käyttöön tulevan sovelluksen toteutukseen.

Ohjelmistojen suunnittelussa on otettava huomioon mahdolliset päivitykset, lisäykset ja muutokset. Muutokset tulee pystyä toteuttamaan siten, että vain muutoksen alainen ohjelmiston osa päivitetään työn yhteydessä. Muu ohjelmiston runko pysyy ennallaan eikä täten altistu muutoksille. Valitettavasti perusteellinen suunnittelu on varsinkin www-sivujen tapauksessa enemmän poikkeus kuin sääntö. Projektit ovat usein laajuudeltaan vaatimattomia, eikä ylläpidettävyyden katsota olevan avainkysymys. Jos sivusto ei sisällä dynaamisesti päivitettäviä tai päivittyviä osia, katsotaan suunnitteluun panostamisen olevan usein lähinnä ajan hukkaa.

Kun käyttäjällä on aktiivisempi vuorovaikutus sivuston tai ohjelmiston kanssa, huolellisen suunnittelun ja suunnittelumallien käyttötarve korostuu. Suunnittelumalleilla tarkoitetaan ohjelmistotekniikassa kuvausta, joka määrittää ohjelmistossa toistensa kanssa vuorovaikutuksessa tai yhteistoiminnassa olevien luokkien ja osien avainkohdat. Tarkoituksena on esittää ratkaisu olemassa olevaan suunnitteluongelmaan tietyssä kontekstissa. Puhuttaessa suunnittelumallista on kyse pitkän ajan kuluessa testatuista ja hyväksi todetuista suunnitteluratkaisuista.

Insinöörityöraportissa käsittelen ohuesti yhtä suunnittelumallia, Model-View-Controller eli MVC, jonka alkuperä on 1970-luvun lopulla valmistuneessa Smalltalk-projektissa. Norjalaisen ohjelmistokehittäjän Trygve Renskaugin sanotaankin olevan MVC-arkkitehtuurin isä tämän Palo Altossa Yhdysvalloissa läpiviedyn projektin ansiosta. Suurten tietomäärien käsittely graafisen käyttöliittymän kautta oli osoittautunut ongelmalliseksi, ja ratkaisuksi tähän Reenskaug kehitti mallinsa. Tarkoituksena oli erottaa ohjelmiston taustalogiikka käyttöliittymän logiikasta yhdistäen ”käytävällä”, jonka nimeksi tuli mittavan pohdinnan jälkeen ”Controller” [1, s. 1].

Puhuttaessa suunnittelumalleista ja etenkin MVC:sta tuntuu olennaiselta selittää ongelma, haasteiden eriyttäminen, englanninkieliseltä nimeltään ”Separation of concerns”. MVC-arkkitehtuurin modulaarinen lähestymistapa ohjelmiston suunnitteluun on yksi tapa toteuttaa haasteiden eriyttämistä. Modulaarisuus toteutetaan ryhmittelemällä loogisesti toisiinsa liittyviä elementtejä, kuten ohjeita, proseduureja, muuttujien alustuksia ja olioiden määreitä [2, s. 85].

Kukin ohjelmiston kerros toteuttaa sille määrätyt asiat ja tarjoaa ne palveluina seuraavaksi ylemmälle kerrokselle. Samalla kerros ”piilottaa” itseään alemman kerroksen toiminnan ylemmältä kerrokselta. [3, s. 43.]

Pääosa insinöörityöstäni keskittyy kuvaamaan Model-View-Controller-mallia JavaScript-kielen kontekstissa ja tarkemmin MVC-mallin ilmentämistä AngularJS-nimisen sovelluskehyksen tapauksessa. MVC-mallin käyttö selainpuolella on haasteellista, sillä todellista kerrosten erottelua ei voida tehdä. Tämä johtuu JavaScriptin luonteesta selaimessa suoritettavana kielenä, johon jokaisella selaimen käyttäjällä on näkyvyys. Katson tarpeelliseksi, että tämän insinöörityön puitteissa valotetaan näitä haasteita mahdollisimman tarkasti.

Insinöörityössä toteutetaan MTV-yhtymälle yhtiön sisäiseen käyttöön tarkoitettu työkalu, jonka avulla toimittajat ja sisällöntuottajat voivat luoda vuorovaikutteista sisältöä verkkosivuille. Toteutus vastaa tarpeeseen päivittää olemassa oleva ja hankalaksi koettu toteutus moderniin kehykseen, mikä helpottaa ja nopeuttaa sisällöntuottajien työtä.

2 Suunnittelumalli – Model-View-Controller

On perusteltua olettaa, että jokainen ohjelmisto muuttaa ulkoasuaan elinkaarensa aikana ja sillä voi olla monta graafista käyttöliittymää samanaikaisesti [5, s. 1]. Professori Trygve Reenskaug kehitti MVC-mallin erottamaan graafisen käyttöliittymän ja sovelluksen taustalla suoritettavan koodin toisistaan eri kerroksiin. Kerrosten nimiksi annettiin Model (Malli), View (Näyttö) ja Controller (Ohjain). On vakiintunut käytäntö erottaa MVC-malli näihin kolmeen kerrokseen. Malli sisältää systeemisen tiedon ja metodit sen käsittelyyn. Näyttö esittää käsitellyn tai haetun data käyttäjälle. Ohjain ottaa vastaan käyttäjän komennot ja päivittää Mallin ja Näytön asianmukaisesti [4, s. 121].

Omiin eristettyihin kerroksiin eroteltu malli, kuten MVC, on oiva esimerkki haasteiden eriyttämisestä. Ohjelmiston rakenne erotellaan kerroksiin, jotka keskustelevat keskenään, mutta eivät ole suorassa yhteydessä toisiinsa. Näin muutokset yhdessä kerroksessa eivät suoraan vaikuta toisen kerroksen toimintaan, vaan on mahdollista muokata tai päivittää olemassa olevaa toteutusta koko ohjelmiston toiminnan häiriintymättä. Ohjelmistokehityksen kannalta on edullista pitää esimerkiksi esitystapakerros erillään taustalla toimivasta liiketoimintalogiikasta jo siksi, että eri alueisiin erikoistuneet kehittäjät voivat työskennellä kerroksessa, joka vastaa heidän osaamistaan parhaiten. Näin kaikkien kehittäjien ei tarvitse olla ammattilaisia kaikissa osa-alueissa, eikä kaikkien tarvitse tuntea ohjelmiston kaikkia toimintoja yksityiskohtaisesti. Myös itsenäisten moduulien uudelleenkäyttö helpottuu, kun ohjelmiston osat erotellaan toisistaan tähän tapaan.

Steve Burbeck toteaa julkaisussaan: ”Soveltaakseen MVC-mallia tehokkaasti ohjelmoijan täytyy ymmärtää työnjako mallin eri osien välillä. Täytyy myös ymmärtää kuinka osat kommunikoivat keskenään. Yhden hiiren, näppäimistön ja näytön jako useiden sovellusten kesken vaatii tarkkaan suunniteltua yhteistyötä ja kommunikointia.” [1, s. 2.] Ohjelmiston eri osien keskinäisen kommunikaation tuntemus on tärkeää, mutta yksityiskohtaisella tasolla kaikkien ei tarvitse olla moniosaajia.

Edellä mainittu haasteiden eriyttäminen puoltaa vahvasti suunnittelumallin, tässä tapauksessa MVC:n, käyttöä ohjelmiston suunnittelussa ja kehityksessä. Ohjelmiston selkeys ja rakenteen päivittämisen helppous ovat eriyttämisen kautta tavoiteltavia ominaisuuksia. Yrityksille edullinen seikka MVC-mallin käyttöön kannustamisessa on, että se kannustaa tehokkaaseen työskentelyyn. Seuraavaksi esittelen yleisellä tasolla suunnittelumalleille tyypillisen rakenteen, joka toteutuu myös Model-View-Controllerin kohdalla.

2.1 Ratkaisumallin nimi

Suunnittelumallin nimi kuvaa mallin sisältämän periaatteen lyhyesti ja yksiselitteisesti. Mallin nimen perusteella voidaan nopeasti hahmottaa käytetty rakenne, sen tuottama hyöty ja siihen liittyvät rajoitukset. Mallin nimi auttaa ohjelmiston suunnittelijan ajattelemaan ohjelmistorakennetta yleisellä (abstraktilla) tasolla, ja sen käyttö voi helpottaa suunnittelijoiden välistä keskustelua. [7, s. 3.] Esimerkiksi Singleton-pattern (engl. Single

= yksikkö) on malli, jonka mukaan tietty objekti voi esiintyä kerrallaan vain yhdessä paikassa eikä sitä voi samanaikaisesti monistaa. MVC-mallissa Model-View-Controller-sanat ilmaisevat suunnittelumallin käyttötavan ja ohjelmistokehitystyylin, jota malli edustaa.

2.2 Suunnitteluongelma

Suunnittelumallia ei tule käyttää, ellei siihen ole perustelua. Yleisesti perustelu on ratkaisua vaativa ongelma, johon suunnittelumalli vastaa. Ongelma kuvaa ohjelmistotekniikassa tilannetta, jonka ratkaisussa suunnittelumallista on ollut todennettavaa hyötyä. Ongelman tulee ilmetä toistuvasti monenlaisissa ohjelmistoissa, ja sen tulee olla yleisluontoinen (esim. ohjelmointikielestä riippumaton). Muut kuin arkkitehtuuri- tai yksityiskohtaiseen suunnitteluun liittyvät ongelmat rajataan pois. [8, s. 105.] Ongelma kertoo kehittäjille, vastaako suunnittelumalli käsillä olevaan suunnitteluongelmaan.

2.3 Ongelmayhteys

Usein suunnitteluongelma ja ongelmayhteys esitetään yhdessä. Ongelmayhteys ilmaisee kokonaisuuden, jossa ongelma ilmenee. Konteksti kertoo myös tilanteen, jossa suunnittelumallia voidaan käyttää ongelman ratkaisemiseksi. MVC-mallin tapauksessa ongelma voi esimerkiksi olla ohjelmiston jatkokehitys ja ylläpito. Ongelmayhteys määrittyy esimerkiksi, kun edellä mainitut vastuut jäävät sovelluksen kehittäneelle yritykselle ja ongelmiin haetaan sujuvaa ratkaisua. Ongelman ratkaisuun MVC-mallin käyttö on perusteltua. Ongelman ja ongelmayhteyden ei siis tarvitse liittyä mikrotason kysymyksiin, vaikka useimmiten näin onkin.

2.4 Ongelman ratkaisu

MVC-malli hajauttaa ohjelmiston kehityksen kolmeen toisistaan eristettyyn tasoon. Niistä jokaisella on oma tehtävä, ja ne voivat seurata toisissaan tapahtuvia muutoksia ja ilmoittaa ilmenevistä tason tilan päivitystarpeista.

Malli

Ohjelmiston käyttäjälle näkyvän kerroksen taustalla suoritettavien loogisten toimien kerrosta kutsutaan Malliksi. Se sisältää ohjelmiston niin sanotun liiketoimintalogiikan. Tyypillisesti Malli sisältää työkaluja tietokanta-abstraktioon, sähköpostin lähetykseen, käyttäjäsyötteiden validointiin js käyttöoikeuksien varmennukseen [9, s. 201]. Malli koostuu liiketoimintalogiikan suorittamiseen suunnitelluista ohjelmistoluokista. Sillä ei ole suoraa yhteyttä käyttäjään, vaan komennot ja syötteet kulkevat erillisen välittäjäkerroksen läpi Malliin. Usein välittäjäkerroksena MVC-mallissa toimii Ohjain.

Näyttö

Ohjelmiston esitystapakerrosta kutsutaan Näytöksi. Tapa, miten Näyttö käyttäjälle esiintyy, voi vaihdella toteutuksen mukaan. Yleisesti Näyttö on osa graafisen käyttöliittymän kokonaisuutta tai vaikkapa tekstikomentoja vastaanottava osakokonaisuus. Vaikka Näyttö mielletään usein juuri graafisia komponentteja sisältäväksi koosteeksi, sen ei ole pakko näyttäytyä niin [5, s. 2].

MVC-mallin mukaan Näytön tiedot Ohjaimesta ja Mallista ovat rajalliset. Sen ei tarvitsekaan olla tiukasti yhteydessä Malliin, mutta Näytön tulee tietää Mallin olemassaolosta. Tietoisuus Mallista ei ole välttämättä eksplisiittinen, vaan tämä tapahtuu yleensä Ohjaimen välityksellä.

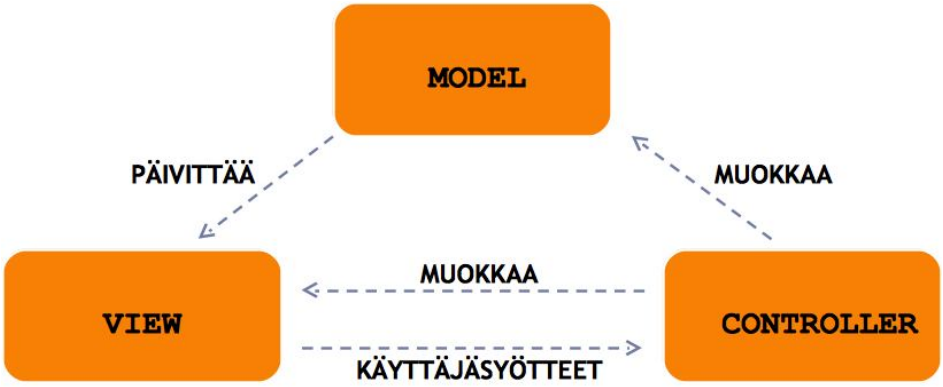
Ohjain

Trygve Reenskaugin mukaan [1, s. 1] Ohjain on linkki käyttäjän ja systeemin välillä. Ohjaimella on kaksi pääasiallista tehtävää. Se vastaa Näytön päivittämisestä Mallissa tapahtuneiden muutosten jälkeen ja välittää Näytöltä vastaanotetut syötteet Mallille, joka tämän sitten päivittää itsensä. Käyttäjäsyötteet käsitellään aina Ohjaimen kautta.

Järjestelmän kannalta ajateltuna Ohjain tarjoaa käyttäjälle herätteet eli tarvittavat Näytöt paikkoihin, joissa niiden tulee kussakin ohjelmiston tilassa olla saatavilla. Se myös mahdollistaa käyttäjän vasteiden vastaanottamisen tarjoamalla dataa, komentoja ja valikoita käyttäjän saataville. [1, s. 1.] Yksinkertaistettuna Ohjaimen ja Näytön yhteyttä voi luonnehtia siten, että Ohjain vastaa sisääntulevista signaaleista, kun Näyttö vastaa ulostulosta [5, s. 2].

Toteutus

Kuvassa 1 nähdään tyypillinen tapahtumien kulku MVC-mallissa. Malli selitetään yleisesti seuraavalla tavalla. Ohjaimen muuttaessa tietoa tai ominaisuuksia mallissa, malli ilmoittaa muutoksesta näytöille, jotka päivittävät itsensä [1, s. 11]. Käytännössä Mallin ollessa tiedoton sen muutoksia ilmentävästä Näytöstä tapahtuu päivitys aina Ohjaimen kautta. Käyttäjän tekemät muutokset tai pyynnöt välitetään ensin Ohjaimelle. Ohjain suorittaa käyttäjäsyötteille tarvittavat toimet saattaakseen ne Mallin ymmärtämään muotoon. Tämän jälkeen syöte ohjataan Mallille, joka suorittaa tilanteen vaatiman operaation. Operaatioihin voi kuulua esimerkiksi laskenta, tietokantahaku tai muu datan muokkaus. Malli palauttaa tuloksen Ohjaimelle, joka vastaavasti kertoo Näytölle päivitystarpeesta.



Kuva 1 MVC-malli.

MVC-arkkitehtuuri on laajalti käytetty arkkitehtoninen lähestymistapa interaktiivisiin sovelluksiin [1, s. 13]. Ohjain kääntää vuorovaikutuksen näytön kanssa toiminnaksi, jonka Malli suorittaa. Yksittäisen käyttäjän toiminta voi olla näppäimen painaminen tai valikkotoiminnon käyttäminen. Web-sovelluksessa nämä esiintyvät HTTP-pyyntöinä. Ohjaimen toimintoihin kuuluvat liiketoimintaprosessien käynnistäminen tai mallin tilan muuttaminen [1, s. 15].

3 MVC selaimessa

Mallin, Näytön ja Ohjaimen erottaminen selaimessa ei välttämättä eroa palvelimella suoritettavien MVC-ratkaisujen rakenteesta. Tulkintoja suunnittelumallin käytöstä on monia, sillä MVC-mallin käyttö JavaScriptillä kirjoitetuissa sovelluskehyksissä on viime vuosina yleistynyt. Mallin ja Näytön erottaminen on yleisesti selkeämpää kuin Näytön ja Ohjaimen, sillä niiden roolien erottaminen toisistaan on yksiselitteisempää. Malli vastaa datan

käsittelystä ja liiketoimintalogiikasta, kun taas Näyttö keskittyy datan esittämiseen tietyn muotoisena, eikä ota kantaa siihen, mitä sille syötetään.

Ohjaimen toteutukset erottavat sovelluskehysten suunnittelumallit toisistaan. Läheskään kaikki selaimessa suoritettavat sovelluskehykset eivät toteuta MVC-mallia täydellisesti vaan keskittyvät erottamaan Mallin ja Näytön. Näissä tapauksissa Ohjain voidaan toteuttaa esimerkiksi reitittimen kaltaisena kokonaisuutena, johon on lisätty kyky sitoa Näytön elementteihin toiminnallisuutta. Puhuttaessa JavaScript-pohjaisista sovelluskehyksistä sanotaan niiden yhdistyvän MV*-perheen alle, jossa jokaisella voi olla hieman omanlainen tapa lähestyä MVC-arkkitehtuuria [10].

3.1 Sovelluskehysesimerkki - AngularJS

AngularJS on selaimessa toimiva JavaScriptilla toteutettu sovelluskehys, joka ilmentää MVC-arkkitehtuuria melko puhtaasti. Sovelluskehityksen käyttö www-sivun tai palvelun toteutuksessa tähtää yleensä rikastettuun, natiivisovelluksen kaltaiseen käyttökokemukseen. Usein käytetyt termit tai paradigmat, jotka kuvaavat tätä www-sivujen tyyliä, ovat SPA ja RIA. Nämä kaksi termiä eroavat toisistaan laajuudeltaan ja ajattelutavaltaan merkittävästi, mutta molemmissa on paljon yhtäläisyyksiä.

SPA (Single-page-application) tarkoittaa www-sivua, joka toimii yhden palvelimen tarjoaman sivun kontekstissa. Sivu on yleensä sovelluksen kaltainen. Vaikka sivu ei sisälläkään palvelimelta tarjoiltuna useampia sivuja, SPA voi sisältää useita sivunäkymiä, joiden välillä liikutaan sivun navigaation kautta. Eri näkymät pyydetään SPA:n taustalla olevan selaimessa suoritettavan sovelluskehyksen sisäisen reitityslogiikan kautta [11]. Tästä syystä sivupyynnöt palvelimelle ovat rajoittuneet yhteen. RIA (Rich Internet application) näyttäytyy toiminnaltaan samankaltaisena kuin SPA. RIA:n tapauksessa on yleistä, että käyttäjä joutuu asentamaan kolmannen osapuolen sovelluksen, joka hyödyntää kohdekäyttöjärjestelmän resursseja [12].

Molemmille yhtäläinen toimintamalli on pyytää palvelimelta käyttöön ulkopuolisia resursseja, vaikka sovellus esitetäänkin saman www-sivun kontekstissa. Yleisin tämän toteuttamiseen käytetty tekniikka on AJAX.

Malli

AngularJS:n tapauksessa Malli on tavallinen JavaScript-olio. Ohjelmoija ei ole velvoitettu käyttämään kehykselle spesifisiä tapoja luoda tai jäsentää oliota, vaan on mahdollista käyttää mitä tahansa jo olemassa olevaa oliota Mallin kuvaukseen. Jotta Mallin tieto saadaan sovelluksen käyttöön, se täytyy paljastaa tietoa ilmentävälle tai käyttävälle scopelle [13, s. 14]. Esittelen scopen hieman tarkemmin tuonnempana.

Malli otetaan scopessa käyttöön AngularJS:lle tyypillisesti periaatteella, jota kutsutaan nimellä dependency injection. Olkoon suomenkielinen nimi tässä riippuvuuden injektointi. Se on tehokas tapa syöttää tarvittava Malli-olio Ohjaimelle tai muulle tietoa hyödyntävälle sovelluksen osalle. Riippuvuusinjektion ansiosta Mallin ominaisuudet tuodaan sovelluksen käyttöön vain paikoissa, jossa ne ovat tarpeen.

Ohjain

Ohjaimet esiintyvät tavallisina funktioina [13, s. 14]. Kuvassa 2 nähdään, miten Ohjain laajentaa moduulia nimeltään themeTestControllers. Ohjaimen alustuksessa huomioidaan, mihin Malleihin tai ulkopuolisiin moduuleihin sillä on riippuvuuksia. Kirjoitusasu ei ole vahvasti määritelty.

```
var themeTestControllers = angular.module('themeTestCtrl', []);

themeTestControllers.controller('formsCtrl', [
'$scope',
'$routeParams',
'QuestionService',
'ImgListService',
'$fileUploader',
'$location',
function($scope, $routeParams,
    QuestionService, ImgListService,
    $fileUploader, $location){
```

Kuva 2 AngularJS - Ohjaimen rakenne.

AngularJS:llä ei ole tarpeen periyttää Ohjaimia kehyksen luokista. Kytköksiä kehyksen omiin rajapintoihinkaan ei ole pakollista synnyttää [10]. Ohjaimen ensisijainen tehtävä on alustaa sen käyttämän Mallin tai Mallien data. Tässä voidaan käyttää ulkopuolista rajapintaa ja taustasovellusta tai alustaa Malli-olio käyttäen staattisia, sovelluksen alustustilalle määriteltyjä arvoja. Toinen Ohjaimen tärkeimmistä tehtävistä on laajentaa itsensä muodostamaa tai perittyä scopea Näytön vuorovaikutuksellisiin toimiin tarvittavilla funktioilla.

Näyttö

AngularJS on erikoistunut luomaan Näytöstä selainpuolen kehityksen kannalta helppokäyttöisen ja helpon kehittää. Kuten kuva 3 osoittaa, Näyttö nojaa tavalliseen HTML-rakenteeseen, eikä erityistä mallinekehystä tai mallineiden luomiseen tarkoitettua kieltä tarvita.

```
<div class="form-container">
  <div class="test-type-select margin-left20">
    <select ng-model="testFormSelection"
      ng-options="opt.label for opt in testFormSelectionOptions">
    </select>
    <span> mukaan arvioitava testi.</span>
  </div>
  <div ng-if="testFormSelection.value == 1">
    <form ng-submit="submit()" name="myFormChoiseCount">
      <div class="test-name">
        <span class="heading">Testin nimi:</span><input type="text" name="testi" ng-model="testFormChoiceCount.name" />
      </div>
      <ul class="unstyled" ng-repeat="input in testFormChoiceCount.questions" >
        <li>
          <p class="question">Kysymys [{{$index + 1}}]:</p>
          <div class="input">
            <input name="kysymys{{$index}}" type="text"
              ng-model="testFormChoiceCount.questions[$index].question" />
            <span class="remove">[<a href ng-click="removeQuestion($index)">X</a>]</span>
          </div>
```

Kuva 3. Näyttö koostuu HTML:stä.

Vahva osa Angularin toimintaa on kyky rikastaa alkuperäistä HTML-elementtien sanastoa omilla ohjeistuksillaan [13, s. 22]. Ohjeistukset tuovat sovelluskehyksen sisäänrakennetun toimintaperiaatteen HTML:n jatkoksi ja mahdollistavat samalla DOM-rakenteen päivityksen ilman käyttäjän suoraa komentoa.

3.2 AngularJS-sovelluskehyksen rakenne ja toiminta

Seuraavassa käyn hieman tarkemmin läpi AngularJS-sovelluskehyksen rakenteen pääpiirteissään. Koko sovelluskehyksen toiminnan tarkempi tarkastelu on tämän työn laajuuden ulottumattomissa, joten yritän luoda käsityksen tärkeimmistä ominaisuuksista. Seuraavassa käsitellään ominaisuuksia, jotka laajentavat angular-nimiavaruutta. Nimiavaruudella tarkoitetaan tässä sovelluskehyksen itsensä käyttöön ottamaa nimettyä aluetta muistista. Tämän nimiavaruuden alaiset ominaisuudet, oliot ja funktiot periytyvät siitä ja ovet täten tunnistettavissa AngularJS:n kontekstissa.

Module

Kuten edellä on mainittu, AngularJS ei rajoita ohjelmoijaa käyttämään ennalta määriteltyä tapaa kirjoittaa koodia. On kuitenkin olemassa suositeltuja tapoja jäsentää ohjelman eri osat jo pelkän koodin selkeyden ja luettavuuden takia. Module on yksi angular-nimiavaruuden ominaisuuksista, jonka avulla sovelluksen koodi on helpompi siistiä ja hallita [13, s. 26]. Koodin selkeys tai liittäminen nimiavaruuteen eivät suinkaan ole Modulen ainoita hyötyjä.

Moduulien avulla voidaan määritellä sovelluksen niin sanottu bootstrap-moduuli ja esimerkiksi koostaa Ohjaimet, Mallit, Direktiivit ynnä muut omiin kokonaisuuksiinsa, jotka voidaan tarpeen vaatiessa myöhemmin injektoida sovelluksen käyttöön. Moduulit siis hyödyntävät myös aiemmin mainittua riippuvuusinjektiota. Sen ansiosta voidaan kuvailla riippuvuuksia eri ohjelmiston osien välille.

Scope

Scope on JavaScript-olio, jolla viitataan sovelluksen Malliin tai Malleihin. Se on viitekehys, jota vastaan suoritetaan ohjelmoijan kirjoittamia määritelmiä. Scope jäljittelee DOM-rakennetta ja muodostaa näin hierarkian, jonka lähtötasoa nimitetään termillä rootScope [13]. RootScope on ensimmäinen olio, josta muut scopet periytyvät. Sovelluskehys alustaa sen sovelluksen käynnistyksen yhteydessä, mutta ohjelmoija voi laajentaa sitä halutessaan.

Scope sisältää aina tietyt sovelluskehyksen omat metodit ja rajapinnat määritteiden seurantaan ja muokkaamiseen. Se tavallaan liimaa sovelluksen Ohjaimen ja Näytön yhteen, sillä kaikki käyttäjälähtöiset suoritteet arvioidaan kulloistakin scopea vastaan.

Directive

Direktiivi voidaan käsittää eräänlaisena tietosanana. Se pitää sisällään määrittelemättömän määrän omaa toiminnallista logiikkaa, jota tarvitaan, kun direktiiviin syötettyä dataa käsitellään. Tieto direktiiviin voi tulla perittynä scopesta, jonka sisällä direktiivi on, tai eksplisiittisesti sille annettuna. Se, miten direktiivi näyttäytyy kehittäjälle tai sivun lähde-

koodissa, on kuin mikä tahansa HTML-elementti. Direktiivisanastoa laajentaakseen AngularJS osaa ohjata selainta siinä, kuinka mukautettua sanastoa tulee käsitellä ja kuinka se tulee ymmärtää.

Services

Kuten edellä on mainittu, AngularJS:n tapauksessa Malli on tavallinen JavaScript-olio. Haasteiden eriyttämisen tapaan Malli tulee pitää erillään muusta ohjelman logiikasta. Liiketoimintalogiikan tiettyyn tehtävään erikoistuneet kokonaisuudet erotellaan Angularin tapauksessa Service-olioiksi. On toki mahdollista olla kirjoittamatta oliopohjaista koodia ja tehdä Mallista vain ”getter”-tyyppinen eli kyselyyn vastaava. Kuitenkin useimmiten halutaan mahdollistaa kahdensuuntainen ”getter/setter”-tyyppinen käyttäytyminen, ja paras keino saavuttaa tämä toiminnallisuus on Angularin kaksisuuntainen tietosidos. Tämän sovelluskehyksen sisäisen toiminnan aikaansaaminen vaatii Service-olion käyttöä siten, että käyttäjän syötteet tai komennot päivittävät jotain Service-olion ominaisuutta, joka on sidottu senhetkiseen aktiiviseen scopeen.

Two way data binding

Two way data binding tarkoittaa kaksisuuntaisen tietosidoksen ominaisuutta. Se on yksi AngularJS:n kehutuimmista ominaisuuksista. Se mahdollistaa tiedon sidonnan Mallin ja Näytön välille siten, että ohjelmoijan ei välttämättä tarvitse kirjoittaa erillisiä getter- ja setter-metodeja. Sovelluskehyksessä on sisäänrakennettu toiminnallisuus kaksisuuntaisia tietosidoksia varten [13, s. 12].

AngularJS mahdollistaa kaksisuuntaisten tietosidosten käytön myös muiden kuin Mallioliioiden kontekstissa, mutta tämän työn tarkoitus ei ole käydä yksityiskohtaisesti läpi AngularJS:n toimintaa, vaan valottaa, miten MVC-malli toteutuu tämän sovelluskehyksen tapauksessa.

Dependency Injection

Dependency injectionilla eli riippuvuusinjektiolla tarkoitetaan sitä, kun jokin tietty ohjelmiston sisäinen palvelu eli riippuvuus syötetään isäntäolion käyttökontekstiin. Tässä tapauksessa isäntäohjelmiston osa, MVC-mallissa tavallisesti Ohjain, ei tee pyyntöä ulko-

puoliseen palveluun, vaan riippuvuusinjektiomallissa sovelluskehys hakee käyttökontekstin riippuvuudet itsenäisesti. Riippuvuusinjektio on kokonaan oma suunnittelumallinsa, joten tässä tapauksessa paneudun siihen lyhyesti ja vain AngulasJS:n tapauksessa.

AngularJS hyödyntää riippuvuusinjektiota jokaisen scopen rakentamisessa. Scope-olio itsessään syötetään kuhunkin käyttökontekstiin eli senhetkiseen sovelluksen tilaan. Scopen rinnalla voidaan syöttää yksi tai useampi Service Malli-olion muodossa tai omana tietyn yhdensuuntaisen toiminnallisuuden mahdollistavana kokonaisuutena. Scope voidaan syöttää myös ilman Malli-olioita.

Angularin riippuvuusinjektiomoottori on vastuussa kolmesta tehtävästä [13, s. 27]:

1 ymmärtää tarpeen kutsua olion kuvaama ulkopuolinen palvelu yhteistoimintaan

2 etsiä kyseinen ulkopuolinen palvelu

3 kytkeä riippuvuudet yhteistoimintaan toimivaksi kokonaisuudeksi.

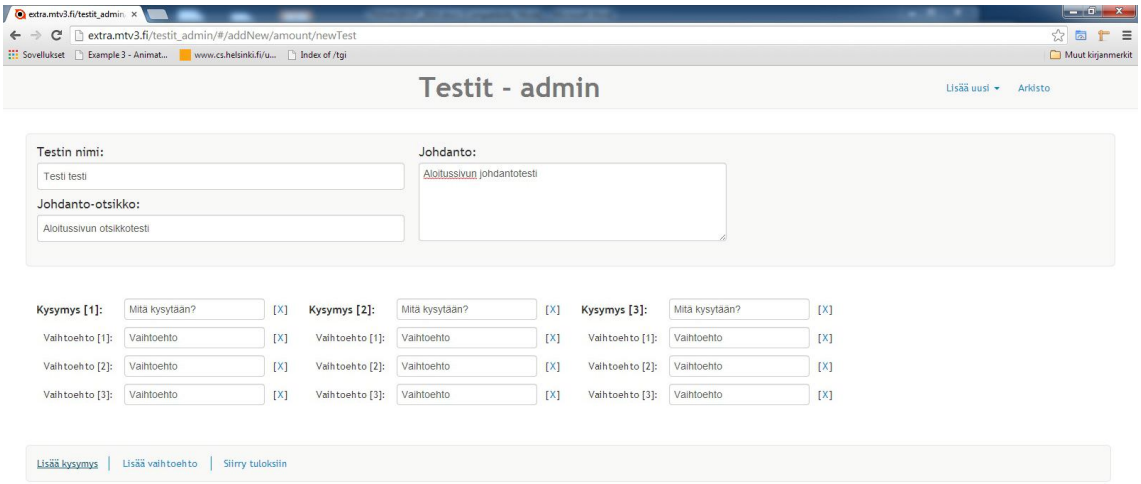
Riippuvuusinjektio mahdollistaa myös unit-testien kirjoittamisen, sillä testitilanteet pystytään helposti eristämään omiin ”sandboxeihin” eli ympäröivästä sovelluksesta eristettyihin tiloihin.

4 Testityökalu – selainpohjainen MVC-sovellus

Tein insinöörityönä MTV Oy:lle työkalun, jota toimittajat ja sisällöntuottajat käyttävät verkkosivujen käyttäjiä aktivoivien testien ja visailujen toteutukseen. Testit ovat monivalintatehtävistä koostuvia kokonaisuuksia, joiden rakenne tallennetaan MySQL-kantaan ja JSON-tietorakenteeksi. Verkkosivulla näytettävän osuuden koostamisesta vastaa JavaScript-toteutus, joka lukee testityökalun tallentamaa tietorakennetta ja jäsentää sen haluttuun HTML-rakenteeseen. Testityökalu itsessään on nähtävissä ainoastaan MTV:n sisäverkossa. Sovellus on rakennettu AngularJS-sovelluskehyksen päälle. Liite 1 pitää sisällään toimeksiannon käyttäjätarinat.

4.1 Sovelluksen käyttö

Käyttäjä hakee testityökalun käyttöliittymän sille palvelimella varatusta URI:sta. Kyseessä on yksi HTML-sivu, joka lataa sovelluksen kehyksen ja alustaa sen URI:n mukaisella tietorakenteella tai tyhjäksi alustettuna. Käyttäjällä on tämän jälkeen mahdollisuus liikkua kolmen eri näkymän välillä ja suorittaa uuden testin luomiseen tai olemassa olevan päivitykseen liittyviä tehtäviä. Kuvassa 4 nähdään olemassa oleva testi ladattuna sovelluksen käyttöön.



Kuva 4 Sovelluksen ensimmäinen vaihe.

Aloitussivun alasvetovalikosta käyttäjä voi valita, halutaanko luoda vastausten määrän vai yhteispistemäärän perustella arvioitava testi. Alasvetovalikon valinta aiheuttaa siirtymän kuvassa 4 esitetylle lomakesivulle, jossa määritellään testin nimi, aloitussivun johdanto, aloitussivun otsikko ja varsinaiset testin kysymykset. Testissä on alustavasti kaksi kysymyskenttää. Vasemman alareunan painikkeista voidaan lisätä kysymyksiä ja vaihtoehtoja sekä siirtyä testin luomisen toiseen vaiheeseen määrittelemään tulokset. Kysymyksiä ja vastauksia voidaan poistaa rivien päässä olevia painikkeita painamalla.

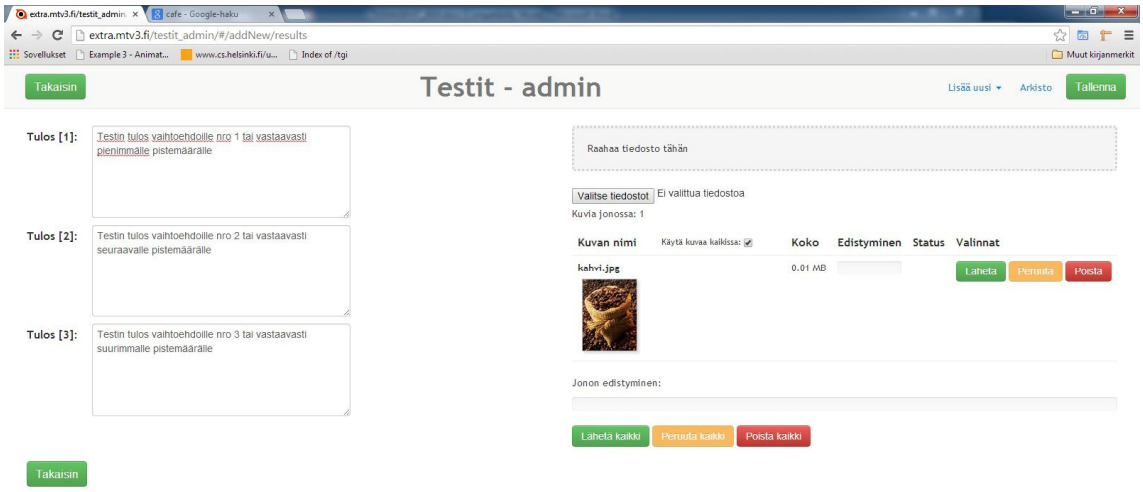
Vaihtoehtojen lisäyspainike lisää uuden vaihtoehdon jokaiseen testin kysymykseen. Vastaavasti vaihtoehdon poistaminen poistaa kyseisen vaihtoehdon jokaisesta kysymyksestä. Tämä oli toimeksiannon mukainen toimintaperiaate, joten en nähnyt tarvetta rakentaa monipuolisempaa logiikkaa kysymyskohtaisten vaihtoehtojen käsittelyyn.

Tuloskenttiä on yhtä monta kuin kysymyksillä vaihtoehtoja. Tuloskentät luodaan automaattisesti vastausvaihtoehtojen lukumäärän perusteella. Tuloskenttiä ei voi erikseen

lisätä tai poistaa, vaan tämä tehdään päivittämällä vastausvaihtoehtojen määrää lomakenäkymässä. Tämä logiikka oli perua vastausvaihtoehtojen määrän säätelytavasta. Logiikan muuttaminen tässä yhteydessä olisi riidellyt toimeksiannon kanssa ja tuottanut turhaa lisätyötä sovelluksen suunnittelussa.

Tuloksen yhteyteen voi myös lisätä linkin ulkopuoliseen palveluun joko omalle rivilleen tai tekstin sekaan. Linkki lisätään syöttämällä seuraavanlainen merkintä tulokseen: link=http://www.example.com/examplepath/,title=Linkin otsikko!=link.

Tulosvaihtoehtojen kanssa voidaan esittää myös kuvia. Kuvia sallitaan yksi vaihtoehtoa kohden, mutta kuvan lisääminen ei ole pakollista. Testeissä käytettävät kuvat on ensin ladattava palvelimelle, josta sovellus voi ne hakea käyttöönsä. Tämä tapahtuu lisäämällä kuvat Valitse tiedostot -painikkeesta avautuvan tiedostoikkunan kautta tai raahaamalla valitut kuvat latausnäkymään sovelluksessa. Kuvat lähetetään painamalla kuvassa 5 näkyviä Lähetä- tai Lähetä kaikki -painiketta.



Kuva 5 Lisäosa kuvien lataukseen.

Kuvassa 5 nähdään sovellukseen upotettu vapaan lähdekoodin lisäosa, jota käytetään kuvien lataukseen palvelimelle. Se on rakennettu AngularJS:n moduuliksi ja sen arkkitehtuurin pohja on suositussa jQuery-kielen lisäosassa. Se nojaa HTML5-standardia tukevien selainten tiedostojen latauksessa käytettävään rajapintaan ja sisältää myös tuen vanhemmille selaimille, jotka eivät edellä mainittua rajapintaa sisällä.

Ladatut kuvat näkyvät listassa, joka nähdään painamalla Tiedostojen valintaan -painiketta. Jokaisessa kuvakentässä on alasvetovalikko, joka sisältää testin tulokset. Tuloksen valinta rekisteröi kuvakentän kuvan valitun tuloksen yhteyteen. Valittu kuva ilmestyy näkyviin tuloksen vieressä näkyvään kuvakenttään. Ennen kuvan valintaa tuloksen viereinen kenttä sisältää tekstin ”Ei kuvaa valittuna”. Tarkoituksena oli kehittää kuvien valintaan raahaustoiminto, jolloin erillistä valikkoa ei olisi tarvittu. Käyttäjä olisi voinut hiirtä käyttäen raahata haluamansa kuvan mihin tahansa tulokseen, jolloin sovelluksen Malli olisi päivittynyt sisältämään kuvan yhdistettynä haluttuun tulokseen. Tämän ominaisuuden toteutukseen ei projektin puitteissa jäänyt aikaa.

Esikatselu-painikkeesta päästään näkymään, jossa testin voi käydä läpi ja tarkastaa mahdolliset puutteet. Jos korjattavaa on, voidaan muutokset tehdä tarvittaviin paikkoihin ja joko esikatsella testi uudelleen tai tallentaa testi Tallenna-painikkeesta.

Testin tallennuksen jälkeen näytetään Päivitä-painike. Se näkyy myös, kun avataan olemassa oleva testi Arkisto-näkymästä. Käyttäjä voi tarvittaessa tehdä korjauksia aiemmin tallennettuun testiin ja tallentaa muutokset painamalla Päivitä-nappia. Tällöin testin id säilyy samana eikä uutta testiä luoda tietokantaan.

Arkisto-listaus näyttää aiemmin tallennetut testit. Testin nimeä tai ID:tä painamalla ladataan testin tiedot tietokannasta JSON-tietorakenteeksi käännettyjä. JSON luetaan sovelluksen käyttöön, minkä jälkeen se on muokattavissa.

4.2 Sovelluksen tekninen toteutus

Työssä kehitetyllä työkalulla on riippuvuudet MySQL-palvelimeen ja tietokantaan sekä PHP-ohjelmointikielellä toteutettuun taustajärjestelmään, joka huolehtii tietojen tallennuksesta, hausta ja muokkaamisesta. Selaimessa suoritettava sovelluskehys ei itsenäisesti kykene näihin toimiin, sillä selaimella ei ole suoraa pääsyä palvelimen tiedostoihin. Testityökalu keskustelee PHP-pohjaisen rajapinnan kanssa, joka puolestaan jakaa komentoja eteenpäin taustajärjestelmän käsiteltäviksi.

Sovellus rakentuu kolmesta eri Ohjaimesta, viidestä Näytöstä ja kolmesta Mallista. Työkalun näkymät on näin ollen jaettu omiksi kokonaisuuksikseen. Ainut sovelluskehyksen ulkopuolinen lisäosa on esikatseluikkunan toiminnallisuudesta vastaava skripti, joka on

kirjoitettu JavaScript-pohjaisella jQuery-ohjelmointikielellä, mutta tuotu osaksi sovelluskehyksen toimintaa DOM-rakennetta laajentavan direktiivin muodossa.

Mallit sisältävät puumaisen datarakenteen ja metodit rakenteen käsittelyyn. Ohjaimelta lähetetään käyttäjän toimia toteuttava käsky päivittää, lisätä tai poistaa yksittäisiä tai useampia ominaisuuksia puurakenteesta. Näytön päivityksen yhteydessä Ohjain komentaa Mallia lähettämään haluttu osa datasta ja tallentaa sen scopen muuttujaan. Tieto on siten Näytön käytettävissä ja kaksisuuntaisen tietosidoksen alainen. Näyttö käyttää dataa joko yksittäisen Mallin ominaisuuden tai luupissa käsiteltävän rakenteen esitykseen. Näyttö on kirjoitettu käyttäen HTML5-standardin kirjoitusasua ja muutamia AngularJS:n sisäisiä direktiivejä.

Kuten edellä mainittiin, sovellus käyttää MySQL-kantaan tallennettua tietoa Mallien datarakenteen tallennukseen ja hakuun. Uuden testin luomisen yhteydessä kantaan tallennetaan lomakkeiden tiedot ja referenssit kysymyksiin ja vastauksiin liittyvistä kuvista. Sovelluksen tieto välitetään rajapinnan kautta taustalle JSON-muotoisena, josta se käännetään PHP:n luettavaan muotoon taulukoksi. Olemassa olevan testin päivittämisen yhteydessä tausta toimii sisään tulevan datan kanssa samalla tavalla, mutta kannan päivitykseen käytettävät MySQL-funktiot eroavat uuden testin luomisesta.

Uuden testin luomiselle, vanhan päivitykselle ja testin haulle yhteistä on lopullisen taulukkorakenteen käännös takaisin JSON-muotoon. Tämä tietorakenne tallennetaan palvelimen levylle hakemistoon, josta se haetaan kuluttajille näytettävän sivun käyttöön.

Sovelluskehyksen valinta projektin selainpuolen toteutustekniikaksi valikoitui oman kiinnostuksen kautta. Lähtötilanteessa en ollut lainkaan tietoinen, millaista on kehittää sovellus AngularJS:llä. Kokemukseni sovelluskehyksistä palvelinpuolelta olivat olleet hyviä, ja sen vuoksi kiinnostuin opiskelemaan vastaavaa tekniikkaa selaimessa.

4.3 Johtopäätökset

Insinöörityöprojektini oli suunnitella ja toteuttaa sovellus MTV Oy:n sisällöntuottajien käyttöön. Toteutustekniikaksi valitsin AngularJS:n, sillä oma kiinnostukseni aiheeseen oli herännyt aiempien palvelinpuolen projektien kautta. Suunnitteluvaiheessa havaitsin,

että aiempi tietämys AngularJS:stä olisi ollut hyödyllistä. Projektin edetessä monet tekemäni ratkaisut osoittautuivat joko mahdottomiksi kehittää eteenpäin, vaikeiksi muokata tai onnekkaan sattuman kautta toimiviksi. Tässä sovelluskehyksessä on myös paljon intuitiivisia ominaisuuksia, jotka tekevät sovelluksen rakenteesta helposti loogisiin kokonaisuuksiin jaettavan. Esimerkiksi sisäinen $watch-toiminto auttaa pitämään liiketoimintalogiikan Mallissa tekemällä itsestään päivittyvän sidoksen Näytön ja Mallin välille Ohjaimen kautta.

Angularin tapauksessa ei ole merkityksellistä, mistä sen saama data haetaan. Ainoa rajoitus on, että data tulee sisään JSON-muotoisena sovelluskehyksen käsiteltäväksi. Kuitenkin nykyisin suosittu tapa on rakentaa palvelinpuolen toiminnallisuus NodeJs-nimisellä JavaScript-tekniikalla, jolloin palvelintoiminnot saadaan sujuvasti yhdistettyä edustan ja Angularin edustan kehitystä askeleen lähemmäs natiivisovelluksen käyttötuntumaa. NodeJs:n käyttöä tukee myös tehtäväsuorittajien valinta projektin ylläpitoon ja esimerkiksi CSS-tyylien kääntämiseen. Tämän projektin tapauksessa PHP:n valinta palvelinpuolen toteutukseen oli tapa nopeuttaa projektin valmistumista eikä paras mahdollinen valinta.

5 Yhteenveto

Insinöörityössä kehitettiin MTV Oy:n käyttöön työkalu, joka korvaa aiemmin käytössä olleen ja jo vanhentuneen sovelluksen. Sovelluksen avulla sisällöntuottajat voivat rakentaa yksinkertaisia monivalintatestejä käytettäväksi www.mtv.fi-sivustolla. Sovelluksen käyttäjät kokivat sovelluksen onnistuneeksi ja sen käytön aiempaa toteutusta helpommaksi ja monipuolisemmaksi.

Projektissa siirryttiin kokonaan uuteen tapaan toteuttaa sovellus. Entinen Flash-toteutus oli käyttäjälle hidas ja hankalasti opittava. Se päätettiin korvata AngularJS-sovelluskehyksen päälle toteutetulla ratkaisulla. Itselläni ei ollut aiempaa kokemusta tästä teknologiasta, joten jouduin opiskelemaan sovelluskehyksen periaatteet alusta. Työtä hankaloitti hieman sekin, ettei kukaan työtovereistani tuntenut tekniikkaa.

AngularJS ei ole yksinkertainen sovelluskehys, eikä sen periaatteiden opiskelu voi viedä paljon aikaa varsinkin, jos MVC-malli ei ole entuudestaan tuttu. Monet kokeneetkin kehittäjät pitävät Angularin lähestymistapaa vääränä tai huonona siksi, että se on rajoittava

ja pakottaa opettelemaan uuden lähestymistavan ongelmiin, joiden ratkaisuun on ennen voitu soveltaa monta eri lähestymistapaa. AngularJS odottaa kehittäjän toteuttavan osia sovelluksen toiminnallisuudesta tietyllä tavoin.

Vaikka oppimiskäyrä AngularJS:n tapauksessa onkin melko jyrkkä, on helppo huomata, miten tehokasta sovelluskehyksen avulla on työskennellä. Oikeiden työtapojen omaksuminen vie aikaa, mutta osaamisen kartuttua sovellusten kehittäminen nopeutuu huomattavasti verrattuna selaimessa suoritettavan logiikan kirjoitukseen jQueryn saati puhtaan JavaScriptin avulla. Sama joustavuus ja tehokkuus on yhtenevä tekijä monissa muissakin edustasovelluskehyksissä, mutta Angularin tapauksessa kehityksestä tekee erityisen ketterää Näytön toteutus puhtaan HTML:n avulla. Erillistä mallinekieltä ei tarvita.

MVC-mallin tuominen selaimessa suoritettavaan sovelluskehykseen mahdollistaa ohjelmiston logiikan jäsentämisen loogisiin kokonaisuuksiin, mutta ei pakota tiettyyn konventioon. Miten selkeää, modulaarista ja eriytettyä koodi on, riippuu täysin kehittäjästä ja hänen tietotaidostaan jäsennellä projekti tietyn metodologian mukaiseksi hallittavaksi kokonaisuudeksi. AngularJS antaa vapaat kädet suunnitella projektin hakemistorakenne halutunlaiseksi.

AngularJS:n mallinnus MVC-arkkitehtuurista ei ole täysin yksiselitteinen. Näyttö ja Ohjain on toteutettu selkeästi, ja varsinkin Näytön toteutus HTML:nä vauhdittaa kehitystä. Ohjaimen tehtävä on selkeä. Se ottaa vastaan komentoja Näytöltä, jonka tila on sidottu Ohjaimen alustamaan scopeen. Ohjain reitittää pyynnöt riippuvuusinjektion avulla scopeen tuoduille Service-oliolle tai käyttää niiden tarjoamia palveluita. Missään vaiheessa ei ole tarpeen suorittaa ohjelmiston loogisia tehtäviä suoraan Ohjaimessa, vaan ne voidaan aina välittää Mallin käsiteltäväksi.

Mallin toteutus on sen sijaan hieman hämmentävä. Vaikka voidaan selkeästi sanoa, että Malli on tavallinen JavaScript-olio, ei sovelluskehyksen rakenteen nimeämiskäytännöissä ole viittausta Malliin. Service-oliot käyttäytyvät AngularJS:n kohdalla Mallin tavoin, ja onkin katsottu oikeaoppiseksi käyttää Service-olioita sovelluksen liiketoimintalogiikan kätkemiseen. Dependency injection tekee mielestäni logiikasta ymmärrettävän, sillä Mallin tehtävähän on toimia silloin, kun sitä pyydetään.

Käyttäjän pyyntöjen reititys on joissain selainpohjaisissa MVC-kehyksissä toteutettu siten, että reitittimen roolia hoitavat toiminnallisuudet on upotettu osaksi Ohjainta, tai siten,

ettei Ohjaimella ole missään vaiheessa MVC-mallin mukaista tehtävää. AngularJS:n tapauksessa reititys määritellään omaan konfiguraatiomoduuliin, joten tällä ei sotketa Malli, Näyttö, Ohjain -kaavaa.

Insinöörityöprojektiin sisältyi myös lukuisia ongelmia, jotka aiheuttivat hankaluuksia jatkokehitysvaiheessa. Toimeksiantoni osoittautui täysin puutteelliseksi, ja loppukäyttäjien palautteen perusteella kävi ilmi, ettei heitä ollut otettu projektin aloituksessa lainkaan huomioon. Tästä alkoi muutostöiden sarja, jonka aikana huomasin, kuinka tärkeää olisi ollut tuntea sovelluskehys paremmin. Tekemäni ratkaisut eivät antaneet mahdollisuutta itsenäisten moduulien refaktorointiin. Huomasin myös, että monia sovelluksen osia olisi voitu pilkkoa pienemmiksi osakokonaisuuksiksi parempaa erottelua varten.

Insinöörityöprojektin suunnittelulle ja toteutukselle määritelty aika ei ollut riittävä kokonaisvaltaisesti hyvän toteutuksen aikaansaantiin. Annetuilla tiedoilla onnistuttiin kuitenkin kehittämään sovellus, joka tuo helpotusta käyttäjien toimiin. Loppukäyttäjien mukaan ottaminen projektin suunnitteluvaiheessa olisi auttanut määrittämään ne avaintoiminnot, joita sovellukselta toivotaan. Täten sovelluksen suunnittelussa olisi voitu ottaa huomioon seikat, joihin jouduttiin myöhemmin puuttumaan. Toteutus on käytössä ja palvelee tarkoitustaan. Sovelluksen rakenteen tarkastelu paljastaisi kuitenkin lukuisia kehitys- tai parannustarpeita, joihin ei insinöörityöprojektin aikana ehtinyt puuttumaan.

Lähteet

1. Burbeck, Steve. 1992. Application programming in Smalltalk. Verkkodokumentti. <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>. Luettu 20.2.2013.

2. Raatikainen, Kimmo. 2007. Johdatus tietojenkasittelytieteeseen: Tarinoita tietojenkasittelytieteen osa-alueilta. Helsinki: Helsingin yliopisto.

3. Hansen, Stuart. 2005. Refactoring model-view-controller. Kenosha: Department of Computer Science, University of Wisconsin.

4. Deacon, John. 2009. Model-View-Controller (MVC) Architecture. John Deacon Computer Systems Development, Consulting & Training.

5. Laplante, Phillip A. 2007. What every engineer should know about software engineering. Boca Raton: CRC Press.

6. Gamma, Erich, Helm, Richard, Johnson, Ralph & Vlissides, John. 2001. Olio-ohjelmoinnin Suunnittelumallit. Suomentanut Anita Toivanen. Alkuperäisteos Design Patterns: Elements of Reusable Object-Oriented Software. Helsinki: IT Press.

7. Koskimies, Kai & Mikkonen, Tommi. 2005. Ohjelmistoarkkitehtuurit. Jyväskylä: Talemtum.

8. McArthur, Kevin. 2008. Pro PHP – Patterns, Frameworks and More. USA: Apress.

9. Reenskaug, Trygve. 1979. Models - views - controllers. Verkkodokumentti. <http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf> 10.12.1979. Luettu 5.3.2015.

10. Osmani, Addy. 2012. Understanding MVC and MVP. Verkkodokumentti. <http://addyosmani.com/blog/understanding-mvc-and-mvp-for-javascript-and-backbone-developers/> 16.1.2012. Luettu 2.3.2015.

11. Upadhyay, Vishal. 2015. Memory Management in Single Page Application. Verkkodokumentti. <https://upvishal.wordpress.com/2015/01/11/memory-management-in-single-page-application/> 11.1.2015. Luettu 2.4.2015.

12. Voces Merayo, Ramon. 2011. Rick Internet Applications and Web Accessibility. Verkkodokumentti. <http://www.upf.edu/hipertextnet/en/numero-9/ria-and-web-accessibility.html> 1.9.2011. Luettu 3.5.201

13. Kozlowski, Pawel . Bacon Darwin, Peter. 2013. Mastering web application development with AngularJS. Birmingham: Packt Publishing.

14. What are scopes? Verkkodokumentti. AngularJS. <https://docs.AngularJS.org/guide/scope> Luettu 10.3.2015.

Käyttäjätarinat

ID	Käyttäjätarinat	Prioriteetti
	Käyttäjänä voin	
K1	valita testin tyypin	1
K2	nimetä testin (uniikilla nimellä)	1
K3	kirjoittaa testin kuvauksen sille varattuun kenttään	1
K4	päättää, montako kysymystä testissä on	1
K5	määritellä kysymykset ja niille vastausvaihtoehdot	1
K6	lisätä ja poistaa kysymyksiä yksitellen	1
K7	lisätä ja poistaa vastauksia yksitellen	1
K8	valita kuvia tietokoneeltani ja lisätä ne sovelluksen käytettäväksi	1
K9	lisätä kysymyksen yhteyteen kuvan	2
K10	siirtyä sovelluksen näkymien välillä menettämättä tietojani	1
K11	määritellä testin tulokset	1
K12	lisätä tulostekstiin linkin ulkopuoliseen palveluun	2
K13	valita tuloksen yhteydessä näytettävän kuvan	1
K14	valita, onko tuloksen yhteydessä tarpeellista näyttää kuva	1
K15	selata sovelluksen käyttöön arkistoituja kuvia	1
K16	tallentaa testin	1
K17	nähdä listan tallennetuista testeistä	1
K18	avata olemassa olevan testin sovellukseen	1
K19	tehdä muutoksia olemassa olevaan testiin ja tallentaa muutokset	1
K20	esikatsella luomani testi	1
K21	poistaa aiemmin luotu testi	1