TAMPEREEN TEKNILLINEN YLIOPISTO

MIKKO HAAPANEN

SELAINKÄYTTÖLIITTYMÄ PALAUTTEENKERÄYSJÄRJESTELMÄLLE

Diplomityö

Tarkastaja: professori Seppo Kuikka
Tarkastaja ja aihe hyväksytty
Teknisten tieteiden tiedekuntaneuvoston
kokouksessa 13.8.2014

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Automaatiotekniikan koulutusohjelma
HAAPANEN, MIKKO: Selainkäyttöliittymä palautteenkeräysjärjestelmälle
Diplomityö, 62 sivua, 6 liitesivua
Elokuu 2014
Pääaine: Automaation ohjelmistotekniikka
Tarkastaja: professori Seppo Kuikka
Avainsanat: AJAX, JavaScript, MVC, single-page-sovellus

Palaute on tärkeää, sillä sen avulla palautteen saaja voi kehittää esimerkiksi opetustaan vastaamaan paremmin yleisön tarpeita ja osaamistasoa. Palautteen kerääminen aiheuttaa kuitenkin ylimääräistä vaivaa sekä palautteen kerääjälle että palautteen antajalle. Työn aiheena on web-pohjaisen palautteenkeräysjärjestelmän toteutus modernien, toteutuksen painopistettä selaimessa suoritettavaksi siirtävien teknologioiden avulla. Järjestelmän on tarkoitus olla sekä palautteen kerääjälle että antajalle vaivaton ja helppokäyttöinen.

Tehdyn toteutuksen ja sen arvioinnin perusteella voidaan todeta, että käyttämällä niin kutsuttua single-page-toteutustapaa web-sovelluksessa voidaan saavuttaa perinteistä web-sovellusta parempi käytettävyys parantuvien sovelluksen reaktioaikojen ansiosta. Lisäksi huomioitavia seikkoja single-page-sovelluksen toteutusta punnittaessa on toteutustyön eroavuus verrattuna perinteiseen web-sovelluksen kehitystyöhön sekä mahdolliset vaikutukset sovelluksen skaalautuvuuteen usealle käyttäjälle.

Järjestelmän koekäyttöjen perusteella voidaan todeta, että palautteen antajat ovat halukkaita käyttämään esimerkiksi älypuhelimen selaimessa käytettävää palautteenkeräysjärjestelmää palautteiden antamiseen. Järjestelmässä annettavat palautteen ovat joko ylläpitäjän ennalta kirjoittamia tai käyttäjän itse kirjoittamia lyhyitä tekstipalautteita. Vaikka palautteen antaminen on järjestelmässä tehty helpoksi ja nopeaksi oli annetuissa palautteissa silti informaatioarvoa palautteen saajalle. Tilaisuuden palautteista automaattisesti muodostettavasta graafista oli koetilaisuuksissa nähtävillä merkityksellisiä trendejä.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Automation Technology
HAAPANEN, MIKKO: A Web User Interface for a Feedback Collection System
Master of Science Thesis, 62 pages, 6 Appendix pages
August 2014
Major: Automation Software Engineering
Examiner: Professor Seppo Kuikka
Keywords: AJAX, JavaScript, MVC, single-page application

Feedback is important. Based on the given feedback, for instance teachers can adjust their teaching to better match the audience and its pre-existing knowledge. Feedback collection is, however, additional work for both the collector and the audience. The topic of this work is implementing a web-based feedback collection system utilising modern web-technologies which shifts the weight of the application towards the client browser. Using the system is intended to be effortless for both the feedback collector and the giver.

Based on the implementation it can be argued that by utilising single-page application technologies, a web-application can be made more responsive compared to traditional thin-client web-technologies. In addition, when choosing the implementation technologies for a web-application, it should be noted that with single-page technologies the development work differs from traditional web-development. The scalability of the application can also be affected.

The demo-uses of the application gave hints that the audience is willing to give feedback using, for example, the browser of a smartphone. The feedbacks given with the system are short texts either pre-written by the organiser or written by the giver. Even though the feedback giving was made as effortless as possible the information value of the feedbacks was not cut out. The graphs automatically composed from the feedbacks showed some meaningful trends.

ALKUSANAT

Haluan kiittää kaikkia diplomityön teossa auttaneita ja ohjanneita, erityisesti toteutetun järjestelmän koekäyttäjiä.

Tampereella 9.8.2014

Mikko Haapanen

SISÄLLYS

1. Johdanto 1
1.1 Jatkuvan palautteen keräys web-järjestelmällä 2
1.2 Single-page-web-sovellus 5
1.3 Työn tavoitteet ja tutkimuskysymykset 7
1.4 Työn rakenne 7
2. Käyttöliittymien suunnittelumalleista 9
2.1 Web-käyttöliittymien suunnittelumalleista 15
2.2 Single-page-toteutustavasta 16
3. Järjestelmän tekninen tausta 21
3.1 JavaScript ja AJAX 21
3.2 REST-web-sovelluspalvelu 22
3.3 Single-page-palvelin 23
3.4 HTML-sivupohjamoottori 24
3.5 Transaktiot single-page-sovelluksessa 25
3.6 Selainliitännäiset 26
4. Toteutusteknologiavaihtoehtoja 27
4.1 Shell-malli 29
4.2 Knockout 30
4.3 Backbone.js 31
4.4 Angular.js 34
4.5 Single-page-toteutustekniikoiden vertailu 37
4.6 Django 39
5. Palautteenkeräysjärjestelmän toteutus 41
5.1 Toteutettava järjestelmä 41
5.2 Valitut toteutustekniikat ja ratkaisut 43
5.3 Single-page-mallin hyödyntäminen toteutuksessa 46
6. Toteutuksen arviointi 48
6.1 Kokemuksia järjestelmän koekäytöstä 48
6.2 Teknisen toteutuksen arviointi 51
6.3 Johtopäätökset single-page-toteutusmallista 53
7. Yhteenveto 57
Lähteet 59
A. Nettipalaute-järjestelmän käyttöohjeet 63
B. UML-sekvenssikaavio ylläpitonäkymän avaamisesta 68

TERMIT JA NIIDEN MÄÄRITELMÄT

.NET	Microsoftin ensisijaisesti Windows-käyttöjärjestelmälle suunnattu sovelluskehys.
AJAX	(Asynchronous JavaScript and XML) Kokoelma web-teknologioita, joiden avulla on mahdollista toteuttaa web-sivuja, jotka kommunikoivat asynkronisesti ja käyttäjälle huomaamatta palvelimen kanssa.
API	Ohjelmointirajapinta (engl. Application Programming Interface) Ohjelmakohtaisesti määritettävä ja toteutettava rajapinta, jonka avulla ohjelmat voivat keskustella keskenään.
CRUD	Lyhenne tulee termeistä luo (engl. create), lue (engl. read), muokkaa (engl. update) ja poista (engl. delete), jotka kuvaavat neljää eri vuorovaikutusta dataan, esimerkiksi tietokannassa.
CSS	(Cascading Style Sheets) Erityisesti HTML-dokumenttien yhteydessä käytettävä tyyliohje. Erillisen tyyliohjetiedoston avulla voidaan erottaa HTML:stä pelkästään käyttöliittymän ulkoasuun vaikuttavat määritykset.
DOM	(Domain Object Model) Ohjelmointirajapinta, jonka avulla selainohjelma paljastaa HTML-tiedostosta jäsentämänsä web-sivun rakenteen esimerkiksi JavaScript-ohjelman tarkasteltavaksi ja muokattavaksi.
Django	Python-ohjelmointikielen sovelluskehys palvelinpainotteisten web-sivujen toteutukseen.
Eväste	(engl. Cookie) HTTP-viestien mukana kuljetettava ja selaimen tallentama vapaaehtoinen tilatieto.
HTML	(Hypertext Markup Language) Kuvauskieli, jota selainohjelma jäsentää ja jolla kuvataan web-sivun näkymän rakenne.
HTTP	(Hypertext Transfer Protocol) Tilaton tiedonsiirtoprotokolla asiakas-palvelin-kommunikointiin.
Historia API	HTML5:n tarjoama rajapinta, jolla voi käsitellä selaimen sivuhistoriatietoa sekä selaimen URL:a laukaisematta uutta sivulatausta.

JSON	(JavaScript Object Notation) Tiedonsiirtomuoto, jossa siirrettävä data esitetään tekstinä samassa muodossa kuin JavaScriptin oliot tyypillisesti esitetään. Vaihtoehto XML:lle.
Kontrolleri	(engl. Controller) MVC-suunnittelumallin kontrollerikerros, joka vastaa esimerkiksi käyttöliittymään tehtyihin käyttäjän syötteisiin ja käsittelee mallia.
Malli	(engl. Model) MV*-suunnittelumallin mallikerros, joka sisältää sovelluksen toimintalogiikan.
MV*	Termillä kuvataan yleisesti MVC-suunnittelumallia, sekä sen eri muunnoksia, kuten MVT ja MVVM.
MVC	(Model View Controller) Käyttöliittymäohjelmien toteutuksessa käytettävä korkean tason suunnittelumalli.
MVT	(Model View Template) Django-ohjelmistokehyksen käyttämä muunnos MVC-mallista.
MVVM	(Model View ViewModel) MVC-mallin muunnos, joka on saanut alkunsa Microsoftin WPF-teknologian myötä, mutta jota nykyään käytetään myös muiden teknologioiden kanssa.
MVW	(Model View Whatever) Angular.js-ohjelmistokehyksen yhteydessä joskus käytettävä termi, jolla viitataan siihen, että kehys ei toteuta mitään tiettyä MV*-mallia, vaan antaa toteuttajalle vapauden käyttää haluamaansa tapaa.
Näkymä	(engl. View) MV*-suunnittelumallin näkymäkerros, jossa on (usein deklaratiivisella kielellä) määritetty sovelluksen käyttöliittymä.
ORM	(Object-relational mapping) Olio-relaatiokuvaus. Tapa tai toteutus, jolla kuvataan olio-ohjelmointikielen luokat relaatiotietokannan rakenteiksi.
Palaute	Palautteen antajan palautteenantojärjestelmään antama palaute, joka on joko tyypitetty (liittyy palautetyyppiin) tai tyypitön (vapaa palaute).
Palautetyyppi	Järjestelmään luotava valmis palautetyyppi, jonka tyyppisiä palautteita palautteen antajat voivat antaa. Esimerkiksi teksti "Hyvää työtä". Luotu palautetyyppi voi liittyä nollaan, yhteen tai useaan sessiotyyppiin.

RPC (Remote Procedure Call) Etämetodikutsu.

Sessio Sessiotyypin ilmentymä. Esimerkiksi *Ohjelmointikurssi 1:n luentokerta 24. huhtikuuta 2014 klo 14.00 - 16.00.*

Sessiotyyppi Palautteenantojärjestelmään luotava tyyppi, joka sitoo yhteen samaa tilaisuutta koskevat palautteet ja palautetyypit. Esimerkiksi *Ohjelmointikurssi 1.*

Sidonta (engl. Binding) Ohjelmointikielen, sovelluskehyksen tai -kirjaston tarjoama yksinkertainen menetelmä, jolla näkymän ja mallin (tai View Modelin) arvoja voidaan sitoa toisiinsa siten, että kun toista muutetaan, vastaava arvo muuttuu automaattisesti myös toisessa. Sisäisessä toteutuksessa voidaan hyödyntää tarkkailija-suunnittelumallia.

Single-page-sovellus Web-sovellus, jossa sovellus on toteutettu niin, että selain lataa palvelimelta aluksi ja vain aluksi selaimessa suoritettavan kooditiedoston, sivun tyylitiedostot ja sivun HTML-pohjan.

Suunnittelumalli (engl. Desing pattern) Yleinen, abstrakti, ratkaisumalli tietynlaiseen ohjelmistotekniikan ongelmaan. Suunnittelumalleja voidaan hyödyntää suunniteltaessa ohjelman arkkitehtuuria. Suunnittelumalli voi koskea sovellusta korkealta tasolta tai olla tarkemmin toteutusyksityiskohtaan liittyvä.

Tilaisuus Tilaisuus on tässä työssä mikä tahansa tapahtuma, esitys, koulutus tai vastaava, josta halutaan kerätä palautetta ja joka mallinnetaan järjestelmään sessiona ja sessiotyyppinä.

Toimintalogiikka (engl. Business logic) Sovelluksen osa, joka mallintaa toimialueen toimintoja. Toimintalogiikkaan ei kuulu esimerkiksi tiedon esittäminen.

UI (User Interface) Käyttöliittymä

URL (Uniform Resource Locator) Web-osoite, esimerkiksi http://www.pal.fi/.

View Model (lyh. VM) MVVM-suunnittelumallin kerros. Yksittäinen VM-olio erikoistaa mallista yhtä näkymä-oliota koskevat osuudet ja pitää yllä pelkästään käyttöliittymään liittyvää tilatietoa.

Web-sovelluspalvelu (engl. Web service) WWW:ssä tarjottu palvelu, jonka tarkoitettu käyttäjä on toinen sovellus. Esimerkiksi hakukonepalvelu voi

tarjota ihmiskäyttäjille suunnatun web-sivukäyttöliittymän lisäksi saman hakupalvelun web-sovelluspalveluna, jolloin hakuja voidaan suorittaa ohjelmasta.

WPF (Windows Presentation Foundation) Microsoftin .NET-sovelluskehyksen osajärjestelmä, jota käytetään graafisten käyttöliittymien esittämiseen.

XML (Extensible Markup Language) Tagien käyttöön pohjautuva merkintäkieli, jonka avulla data ja sen rakenne voidaan kuvata tekstinä. XML:n pyrkimys on olla sekä ihmisen että koneen luettavissa oleva esitysmuoto.

XAML (Extensible Application Markup Language) XML-pohjainen Microsoftin kehittämä, olioiden alustukseen käytettävä kuvauskieli, jolla tyypillisesti kuvataan WPF-sovellusten käyttöliittymiä.

1. JOHDANTO

Palaute on hyödyllistä. Positiivinen palaute vahvistaa palautteen saajan itsetuntoa ja käsitystä siitä, mikä meni hyvin. Rakentava palaute voi olla vielä hyödyllisempää, kunhan se osataan antaa ja ottaa vastaan oikealla tavalla.

Palaute on usein ensisijaisesti hyödyllistä palautteen saajalle, koska se auttaa kehittymään ja tunnistamaan vahvuudet. Palautteen antaminen voi olla myös antajalle hyödyllistä, sillä palautteen saaja voi sen avulla mukautua niin, että palautteen antaja saa palautteen käsittelyn jälkeen suuremman hyödyn. Usein kuitenkin ongelmana voi olla, että vaikka palautetta haluttaisiin saada, sitä ei haluta antaa.

Syitä siihen, miksi palautetta ei haluta antaa, vaikka sitä pyydetään tai kehotetaan antamaan on monia. Luultavasti esimerkiksi seuraavat asiat voivat vaikuttaa päätökseen olla antamatta palautetta:

- Palautteen antaminen jännittää ihmiskontaktin takia. Kehun tai kritiikin antaminen kasvotusten voidaan kokea sosiaalisesti hankalaksi tilanteeksi.
- Palautteen anto on ylimääräistä vaivannäköä. Erityisesti tämä pätee, jos palaute kerätään niin, että siitä ei selvästi voi olla antajalle itselleen hyötyä.
- Palautteen anto ei ole anonyymia. Esimerkiksi luennolla viittaaminen ja sen tunnustaminen, että ei ymmärrä mistä luennoitsija puhuu, on suuren kynnyksen takana, koska koko luentosali kuulee mielipiteen.
- Palautteelle ei ole olemassa selvää kanavaa siinä tilanteessa, kun palautteen antajalle asia olisi aiheellinen.

Yhtä lailla voi olla erilaisia syitä, miksi palautetta ei haluta saada. Kriittisen palautteen saanti etenkin kasvotusten voi olla jännittävää ja siksi sen antoon ei kannusteta. Myös palautteen kerääminen ja käsittely voi olla työlästä saatavaan hyötyyn nähden. Esimerkiksi, jos palautetta kerätään paperilomakkeilla, yhden lomakkeen käsittely kestää palautteen kerääjältä 15 sekuntia ja luennolla on 300 osallistujaa, menee palautteiden käsittelyyn aikaa tunti ja 15 minuuttia, eikä tähän ole laskettu mukaan lomakkeiden tekoa ja jakamista.

1.1 Jatkuvan palautteen keräys web-järjestelmällä

Tähän työhön liittyen toteutettava järjestelmä on internetselaimessa käytettävä palautteenkeräysjärjestelmä. Järjestelmällä palautetta kerätään tilaisuuden aikana, jolloin palautteen antoaika tarjoaa palautteeseen lisäarvoa. Todennäköisimpänä palautteenantolaitteena voidaan pitää älypuhelinta, joka on selaimella ja internetyhteydellä varustettu laite, joka useimmilla on aina saatavilla.

Toteutettavan järjestelmän perusidea on mahdollistaa anonyymina annettavan palautteen vaivaton keräys. Taustaideana on ollut, että koska palautteen antaminen on usein ylimääräistä työtä, joka ei suoraan hyödytä antajaa, tulisi palautteen antamisen olla mahdollisimman vaivatonta, jotta sitä ylipäänsä annetaan. Palautteen saaja toivoo palautteelta suurta informaatiosisältöä, joka kuitenkin tarkoittaa palautteen antamisen vaivan lisäämistä, esimerkiksi siten, että palautteen antajan täytettävän lomakkeen kohdat lisääntyvät, tai niissä pyydetään laajoja vastauksia. Toteutettavalla järjestelmällä pyritään liittämään annettavaan palautteeseen automaattisesti informaatiota, jolla on arvoa palautteen saajalle, mutta joka ei silti lisää palautteen antajan vaivaa. Tällaisena lisäinformaationa toimii palautteen antoaika.

Järjestelmän on tarkoitus olla enemmän kvantitatiivinen kuin kvalitatiivinen keräysjärjestelmä. Yksittäisen annetun palautteen arvo on pieni irrallaan muista annetuista palautteista, mutta kun palautetta saadaan tilaisuuden aikana enemmän ja usealta osallistujalta, voidaan annettuja palautteita tarkastella ajan suhteen ja saada käsitys tilaisuuden onnistumisesta hetki hetkeltä.

Vaikka toteutettava järjestelmä ja sen aihealue ovat suhteellisen yksinkertaisia, voi käsitteissä silti tulla sekaannuksia. Tämän välttämiseksi määritetään käytettäväksi seuraavanlainen termistö palautteen antoon ja toteutettavaan järjestelmään liittyen:

- *Sessiotyyppi* on järjestelmään luotava tyyppi, joka sitoo yhteen samaa asiaa koskevat palautteet ja palautetyypit. Esimerkiksi *Ohjelmointikurssi 1.*
- *Sessio* on sessiotyypin ilmentymä. Esimerkiksi *Ohjelmointikurssi 1:n luentokerta 24. huhtikuuta 2014 klo 14.00 - 16.00.*
- *Palautetyyppi* on esimerkiksi teksti "Hyvää työtä". Luotu palautetyyppi voi liittyä nollaan, yhteen tai useaan sessiotyyppiin.
- *Palaute* (liittyen toteutettavaan järjestelmään) on palautteen antajan antama palaute, joka on tyypiltään esimerkiksi "Hyvää työtä". Keskenään samantyyppisiä annettuja palautteita voi olla mielivaltainen määrä. Esimerkiksi yhden tilaisuuden aikana moni palautteen antaja voi antaa palautteen "Hyvää työtä". Palaute voi liittyä nollaan palautetyyppiin, jos se on ns. vapaa palaute, eli käyttäjän itse kirjoittama teksti.

- *Tilaisuus* on esimerkiksi tapahtuma, esitys tai koulutus, josta halutaan kerätä palautetta ja joka mallinnetaan järjestelmään sessiona ja sessiotyyppinä.

Toteutettavan järjestelmän käyttökohdetta ei ole ennalta määrätty, mutta yksi esimerkki on luento. Järjestelmän on kuitenkin tarkoitus olla sopiva kaikenlaisiin tilaisuuksiin, jotka täyttävät tietyt perusehdot:

- Palautetta antavia osallistujia on yli 10 henkeä. Koska palautteen keräys järjestelmällä on enemmän kvantitatiivista kuin kvalitatiivista, eivät pienellä palautteen antajajoukolla saatavat tulokset ole yhtä hyviä kuin suuremmalla joukolla.

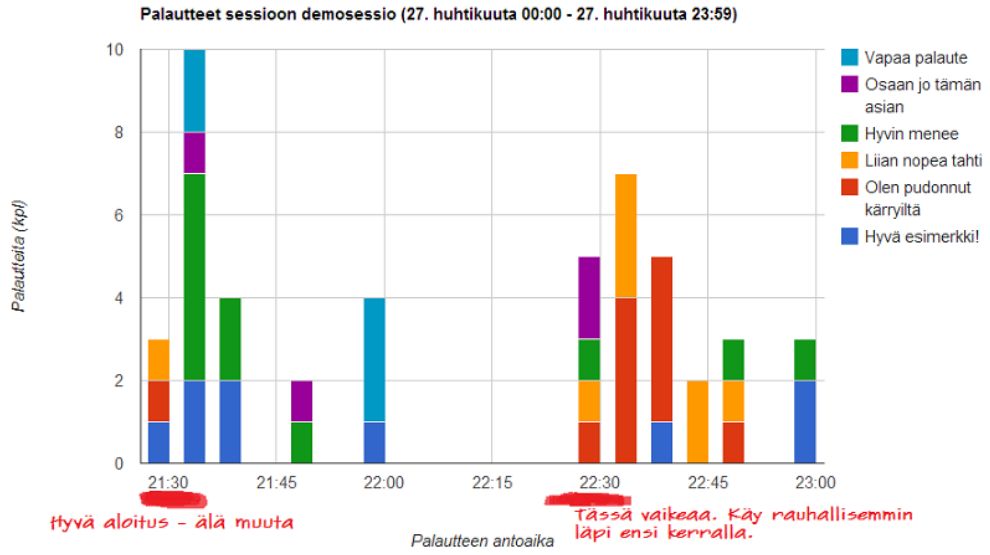
- Tilaisuudella, johon palautteet liittyvät, on selkeä ja tiedetty eteneminen ajassa. Koska järjestelmän ideana on saada hyötyä palautteen antoajasta, ei järjestelmä sovellu esimerkiksi palautteen keräämiseksi kirjasta, jonka palautteen antaja lukee itsenäisesti. Tällöin antoaika olisi hyödytöntä tietoa, koska palautteita tarkastellessa ei voitaisi tietää mitä kohtaa kirjasta kukin antaja on mihinkin aikaan lukenut. Sen sijaan esimerkiksi luennot, esitelmät, videot ja näytelmät voisivat olla sellaisia tilaisuuksia, joissa voidaan tietää, mikä kohta tilaisuudesta on ollut menossa mihinkin aikaan ja jokainen palautteen antaja kokee saman tilaisuuden osan samalla ajanhetkellä.

- Tilaisuuden aikana osallistujien on mahdollista antaa palautetta esimerkiksi älypuhelimella ilman, että se häiritsee tilaisuuden kulkua.

Esimerkiksi yliopistojen peruskurssien massaluennoilla voidaan kerätä kurssin lopulla palautetta paperisilla lomakkeilla, joissa voidaan antaa palautetta koskien koko kurssin toteutuskertaa. Tällaisella palautteella on hankala saada selville tarkemmin, mitkä luentojen kohdat ovat olleet vaikeita tai mitkä liian helppoja. Toisaalta jälkeenpäin annettavalla palautteella palautteen antaja osaa antaa paremmin palautetta koko käsiteltävästä kokonaisuudesta. Jälkeenpäin annettava palaute ja tilaisuuden aikana annettava jatkuva palaute voivat siis täyttää erilaisia tarpeita.

Perinteinen tapa saada palautetta tilaisuuden aikana on kannustaa yleisiöä kysymyksiin ja keskeytyksiin. Esimerkiksi luennoija voi rohkaista luennolla olijoita esittämään kysymyksiä kesken luennon, jos asia on epäselvä tai se käsitellään liian nopeasti. Näin annettava palaute ei ole anonyymiä, mikä lisää kynnystä palautteen antoon ja vääristää annettua palautetta.

Kuvassa 1.1 on esitetty, miltä saatu palaute voi esimerkiksi palautteen antajalle näyttää järjestelmän graafi-esityksenä. Kuvaan on lisäksi lisätty kahteen kohtaan



Kuva 1.1: Esimerkki millaista kerätty palaute voi olla, miten se esitetään ja miten sitä tulkitaan.

esimerkkijohtopäätökset, jotka palautegraafista voitaisiin nopeasti tehdä. Kuvan esimerkin tapauksessa tilaisuuden alussa on saatu positiivista palautetta, josta voitaisiin päätellä, että hetkeä ennen positiivisen palautteen piikkiä on tilaisuudessa ollut erityisen hyvä kohta. Vastaavasti myöhemmin aikajanalla on nähtävillä piikki palautteita, joista voidaan päätellä, että hetki ennen palautepiikkiä tilaisuudessa käsitelty asia ei ole ehkä esitetty tarpeeksi selkeästi, jotta se oltaisiin ymmärretty.

Palautteen antaminen halutaan toteuttaa web-järjestelmänä (web, eli World Wide Web tai lyh. WWW), koska nykyään suurella osalla ihmisistä on jatkuvasti mukanaan internetyhteydellä ja selaimella varustettu laite, kuten älypuhelin [14]. Websovelluksen etuja natiiviin, varta vasten tietylle käyttöjärjestelmälle suunnattuun sovellukseen verrattuna ovat se, että sama sovellus on käytettävissä millä tahansa laitteella, sovellusta ei tarvitse erikseen asentaa, käytettävän sovelluksen versio on aina ajantasainen ja keskitetty palvelin (joka palautteen keräyksessä on tarpeen) on web-teknologioilla toteutuksen kannalta automaattisesti mukana oleva asia ja vahvasti teknologioiden tukema.

Seuraavassa on esitetty tiivistettynä listana hyviä puolia, joita järjestelmällä halutaan saavuttaa sekä palautteen kerääjän että antajan näkökulmasta.

Hyödyt palautteen kerääjälle:

- Koska palautteen antaminen on tehty mahdollisimman helpoksi, sitä saa enemmän.
- Koska palautteen anto on täysin anonyymia, saatu palaute noudattelee yleisön

todellisia ajatuksia. Esimerkiksi kärryiltä tipahtaminen voi olla noloa myöntää viittausäänestyksessä muiden nähden.

- Annettu palaute tallennetaan ja muutetaan automaattisesti ja reaaliaikaisesti helppolukuiseksi graafiksi. Palautteita ei tarvitse käsitellä manuaalisesti.
- Kerättävään palautteeseen saadaan lisäarvoa antoajasta, ja palautteesta on mahdollista nähdä, mikä kohta tilaisuudessa on yleisön mielestä erityisen hyvä tai missä olisi parantamisen varaa. Näin tarkkaa tietoa on vaikea saada, jos palautetta kerätään kerralla esimerkiksi koko menneestä kurssista.

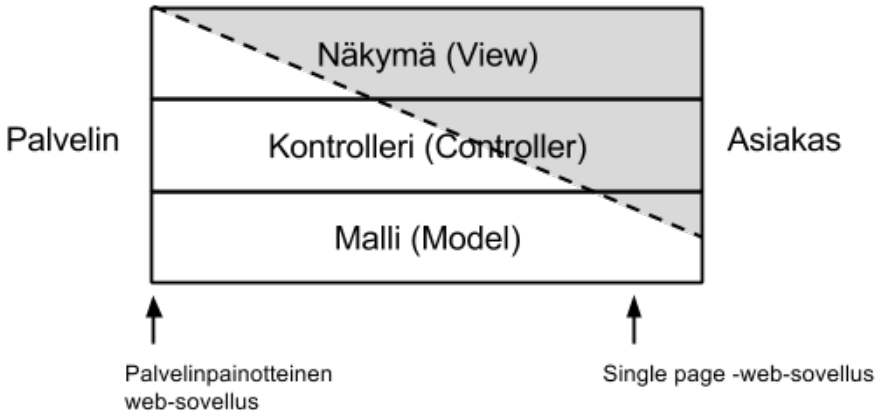
Hyödyt palautteen antajalle:

- Palautteen anto on täysin anonyymia. Kukaan ei näe millaista palautetta antaja antaa. Esimerkiksi kesken luennon kysyminen ei ole kovinkaan anonyymia ja siksi kynnys aiheellisiinkin keskeytyksiin voi olla suuri.
- Palautteen antaminen on helppoa, mutta palautteessa on silti antoaikaan sitomisen ansiosta merkittävä informaatioarvo saajalle.
- Palautteeseen voidaan reagoida nopeasti. Palautteen kerääjä näkee annetut palautteen halutessaan välittömästi ja voi reagoida niihin nopeasti, eikä esimerkiksi vasta kurssin seuraavalla toteutuskerralla.

1.2 Single-page-web-sovellus

Single-page-web-sovelluksella tarkoitetaan web-sovellusta, jossa sovellus on toteutettu niin, että selain lataa palvelimelta aluksi ja vain aluksi selaimessa suoritettavan kooditiedoston, sivun tyylitiedostot ja muut sivun muuttumattomat tiedot. Tämän jälkeen sovellus toimii itsenäisesti asiakaspäässä selaimessa ja siellä suoritettava koodi noutaa tarvittaessa palvelimelta sovelluksen ajon aikana tietoa, joka ei ole valmiiksi esitettäväksi jäsennettyä muotoa, toisin kuin tyypillisen web-palvelimen palauttamat HTML-tiedostot. Single-page-sovellukset rakentuvat AJAX-teknologioiden päälle. AJAX-teknologiat ovat kokoelma web-teknologioita, joiden avulla on mahdollista toteuttaa web-sivuja, jotka kommunikoivat asynkronisesti ja käyttäjälle huomaamatta palvelimen kanssa. Tarkemmin AJAX-teknologiat on kuvattu luvussa 3.1. Single-page-termille ei ole vakiintunutta suomennosta, joten tässä työssä käytetään alkuperäistä, englanninkielistä termiä. Synonyymejä single-page-sovellukselle ovat ainakin SPA (single-page application) ja SPI (single-page interface).

Yleinen tapa toteuttaa web-sovelluksen korkean tason arkkitehtuuri on hyödyntää MVC-suunnittelumallia tai sen muunnosta, jossa käyttöliittymä (näkymä) ja toimintalogiikka (malli) pyritään erottamaan toisistaan. MVC-suunnittelumalli muunnoksineen käsitellään tarkemmin luvussa 2. MVC-malliin ja muihin sen kaltaisiin malleihin viitataan termillä MV*.



Kuva 1.2: MVC-mallin osien toteutuksen jakautuminen palvelimen ja asiakkaan välillä erityyppisissä web-sovelluksissa.

Kuvassa 1.2 on esitetty MVC-mallin osien jakautuminen palvelimen ja asiakkaan välillä perinteisessä palvelinpainotteisessa, ohuen asiakkaan, web-sovelluksessa sekä single-page-web-sovelluksessa. Perinteisessa, staattisen HTML-sivun esittäväs-sä web-sovelluksessa kaikki MVC-mallin osat on toteutettu palvelimella ja asiakas vain esittää valmiiksi muodostetun HTML-näkymän. Single-page-sovelluksessa puolestaan näkymä, niin kutsuttu sovelluslogiikkakerros eli kontrolleri ja osa mallista kuuluvat asiakaspään toteutukseen. Single-page-websovelluksen ei tarvitse olla välttämättä juuri tässä kohtaa kuvaajassa, vaan arkkitehtuurista ja sen tulkinnasta riippuen esimerkiksi malli voi olla kumman tahansa, palvelimen tai asiakkaan, harteilla täysin tai jonkinlainen välimuoto niistä. Single-page-mallissa siis paljon tyypillisesti palvelinpuolen toteutukseen kuuluvia osia on viety asiakaspäässä suoritettavaksi.

AJAX-teknologioita hyödyntävä sovellus voi tarkoittaa mitä tahansa täyden single-page-sovelluksen ja palvelinpainotteisen sovelluksen, jossa on pieni AJAX:a hyödyntävä yksityiskohta, välillä. AJAX:a hyödyntävässä, mutta ei single-page-sovelluksessa haitaksi voi muodostua toteutuksen kannalta se, että MV*-mallin kerrokset pääsevät jakautumaan palvelimella ja asiakaspäässä suoritettavan koodin kesken epäjohdonmukaisella tavalla, mikä tekee toteutuksesta sekavamman ja vaikeammin ymmärrettävän. Lisähaasteena palvelin- ja asiakasteknologiat ovat usein keskenään erilaiset. AJAX-teknologioiden hyödyksi mainitaan yleisesti se, että selaimen ei tarvitse suorittaa kokonaista sivulatausta, kun näkymässä halutaan päivittää vain pieni osa tietoa.

Tiivistettynä single-page-sovelluksen oletetut hyödyt tämän työn ja toteutettavan järjestelmän kannalta tärkeysjärjestyksessä ovat

- parempi käytettävyys verrattuna palvelinpainotteiseen toteutukseen sovelluksen nopeamman reagoivuuden ansiosta,

- monimutkaista ja monipuolista toiminnallisuutta sisältävän asiakassovelluksen toteutus on helppo tehdä, koska teknologiat on tarkoitettu tähän,
- toteutuksen kannalta perinteistä selkeämpi rakenne, sillä MV*-mallin mukaisten kerrosten toteutus on vain joko asiakas- tai palvelinpäässä ja
- palvelinpainotteista sovellusta parempi skaalautuvuus, sillä palvelinkuorma per asiakas on pienempi.

1.3 Työn tavoitteet ja tutkimuskysymykset

Tavoitteena on toteuttaa web-pohjainen palautteenkeräysjärjestelmä, jota on mahdollista koekäyttää esimerkiksi opetuskäytössä. Erilaisia käyttökohteita ja järjestelmän kehitysmahdollisuuksia pyrittiin ideoimaan opinnäytetyön teon aikana.

Järjestelmän toteutuksen yhteydessä syvennytään erityisesti single-page-sovellusarkkitehtuuriin, sen toteutustapoihin, hyötyihin ja haittoihin erityisesti silmälläpitäen tämän työn toteutustarpeita.

Tavoitteena on myös koekäyttää ja saada palautetta työssä toteutettavasta järjestelmästä sekä verrata sitä mahdollisiin muihin samankaltaisiin, olemassaoleviin järjestelmiin. Työssä pyritään keräämään palautteenkeräysjärjestelmästä kokemuksia sekä selvittämään sen potentiaalia ja sitä, tuoko se hyötyä palautteen keräykseen.

Ensisijaisesti työssä selvitetään single-page-toteutustyylin soveltuvuutta web-pohjaisen palautejärjestelmän ylläpitosivun (palautteen kerääjän käyttöliittymä) toteutukseen. Ennakko-oletuksena on saavuttaa sivulla 6 mainitut edut, erityisesti loppukäyttäjälle näkyvä käyttöliittymän sulavuus. Tämän oletuksen paikkansapitävyyttä halutaan testata.

Tutkimuskysymys on, voidaanko single-page-toteutustavalla saavuttaa perinteiseen web-sivun toteutustapaan verrattuna käytettävyysetuja tai muita etuja web-käyttöliittymän toteutuksessä. Toissijainen tutkimuskysymys on, onko jatkuvan palautteen kerääminen web-järjestelmällä hyödyllistä palautteen kerääjille.

Työn puitteissa ei tehdä erityistä käyttäjätutkimusta järjestelmäidean toiminnasta, vaan tätä osaa tuodaan kirjallisessa työssä esille lähinnä koekäyttöön perustuvalla yleisessä kuvalla idean mielekkyydestä (ts. onko toteutettavassa palautejärjestelmäideassa potentiaalia ja koetaanko se hyödylliseksi). Tässä kirjallisessa tuotoksessa ei käsitellä palautteen keräämiseen liittyviä pedagogisia asioita tai muuta palautteeseen liittyvää teoriaa.

1.4 Työn rakenne

Tässä johdantoluvussa esitetään toteutettavan palautteenkeräysjärjestelmän perusidea, perustellaan miksi toteutuksessa halutaan käyttää web-teknologioita ja erityisesti single-page-tekniikoita sekä mitä hyötyä niillä toivotaan saavutettavan. Lisäksi

esitellään tämän diplomityön tavoitteet ja tutkimuskysymykset.

Luvussa 2 esitellään käyttöliittymien toteutusta korkean tason suunnittelumallien avulla. Erikseen esitellään juuri web-ympäristöön liittyvät suunnittelumallit sekä vielä tästä erityistapauksena single-page-sovellusten suunnittelumalleja ja single-page-toteutuksen yleisiä ominaisuuksia. Web-suunnittelumalleja verrataan yleisiin suunnittelumalleihin, jotta saadaan käsitys web-ympäristön erityispiirteistä käyttöliittymän toteutusarkkitehtuurin kannalta.

Lukuun 3 on listattu minkälaisilla teknologioilla ja tavoilla voidaan toteuttaa luvussa 2 esitellyn kaltaisia korkean tason kokonaisarkkitehtuurirakenteita. Luvussa 3 ei esitellä kattavasti kaikkia eri mahdollisuuksia, vaan nimenomaan ne, joiden tiedetään olevan merkityksellisiä toteutettavan sovelluksen kannalta.

Luvussa 4 on esitelty neljä erilaista kirjasto- tai sovelluskehysratkaisua single-page-sovelluksen toteutukseen sekä perusteltu Angular.js-sovelluskehyksen valinta toteutukseen. Lisäksi esitellään palvelinpuolen sovelluskehys Django, jota käytetään toteutuksessa.

Lukujen 2, 3, ja 4 suhde on siis sellainen, että luku 2 esittelee korkean tason suunnittelumallit. Luku 3 paneutuu tarkemmin teknologioihin ja malleihin, joilla tiettyjä 2. luvussa esiteltyjä osa-alueita voidaan toteuttaa ja jotka todennäköisesti liittyvät tarkoista teknologioista riippumatta kaikkiin single-page-sovelluksiin. Luku 4 on vielä enemmän yksityiskohtiin paneutuva. Tässä työssä erityishuomiota on haluttu kiinnittää single-page-toteutusmalliin, joten tarkkaan tarkasteluun ja vertailuun on valittu tähän liittyviä konkreettisia teknologiavaihtoehtoja, jotka ovat käytännössä usein toisensa poissulkevia. Luvun 4 teknologiat siis pohjautuvat luvussa 3 (ja 2) esiteltyihin asioihin.

Luvussa 5 käydään läpi palautteenkeräysjärjestelmän toteutus. Erityisesti paneudutaan toteutukseen single-page-sovelluksen näkökulmasta.

Luvussa 6 käydään läpi palautteenkeräysjärjestelmän koekäytössä saatuja käyttökokemuksia sekä järjestelmän idean että valitun toteutustavan kannalta. Lisäksi kommentoidaan luvussa 5 esiteltyä toteutusta ja perustellaan, miksi valittua toteutustyyliä voidaan pitää onnistuneena. Luvussa 6 vastataan tutkimuskysymykseen.

Luku 7 vetää koko tämän kirjallisen työn sisällön yhteen.

2. KÄYTTÖLIITTYMIEN SUUNNITTELUMALLEISTA

Tässä luvussa esitellään yleisiä käyttöliittymäsuunnittelumalleja ja web-käyttöliittymäsuunnittelumalleja. Lisäksi käsitellään erityisesti single-page-sovelluksiin käytettävissä olevia suunnittelumalleja ja verrataan niitä perinteisiin web-käyttöliittymäsuunnittelumalleihin sekä yleisiin käyttöliittymäsuunnittelumalleihin.

Laajasti tunnettu ja sovellettu käyttöliittymien suunnittelumalli on MVC ja sen muunnokset. MVC-malli koostuu

- mallista (engl. Model),
- näkymästä (engl. View) ja
- kontrollerista (engl. Controller).

MVC-malli kehitettiin alunperin Smalltalk-ohjelmointikielen ja sillä tehtävän käyttöliittymäohjelmoinnin tueksi 1970-luvulla [12, s. 3]. Myöhemmin MVC:n idea on otettu käyttöön yleisenä suunnittelumallina ja sitä on sovellettu ja jatkokehitetty monissa yhteyksissä. Mainittakoon, että Gamma et al. [9] kuvaa MVC:n varsinaista suunnittelumallia laajemmaksi kokonaisuudeksi, joka sisältää esimerkiksi tarkkailija-suunnittelumallin (engl. observer pattern). Tässä työssä MVC:stä kuitenkin puhutaan suunnittelumallina, josta huolimatta on silti hyvä pitää mielessä, että MVC:n kuvaama kokonaisuus on selvästi keskivertoa suunnittelumallia laajempi ja jättää paljon toteutuksen suunnittelua sovelluksen tekijälle.

Seuraavissa kappaleissa kuvataan MVC-mallin ja sen tunnettujen muunnosten idea. Malli- ja näkymäosa ovat lähes identtiset eri muunnoksissa. Sen sijaan kontrolleri, tai muu kolmas osa, ja sen rooli vaihtelevat.

MVC:n malli (Model) tarkoittaa ohjelmaan mallinnettua osaa käsiteltävästä toimialueesta. Esimerkiksi relaatiotietokantaan mallinnettu rakenne (relaatiotietokannan schema), tietokannan tietosisältöä suoraan muokkaava sovelluksen osa sekä sovelluksen toimintalogiikka kuuluvat MVC:n malliin. Malli on usein MVC-suunnittelumallin laajin osa, ja sen sisäiseen toteutukseen on olemassa useita eri tapoja. MVC-malli kuitenkin painottuu nimenomaan käyttöliittymäsuunnitteluun, ja tältä näkökannalta malli on sopiva abstraktio sen sisältämälle, ehkä hyvin monimutkaisellekin kokonaisuudelle. Malli ei ota millään tavalla kantaa käyttöliittymään, eli

siihen kuinka sen sisältämä data esitetään, eikä malli myöskään saa olla tietoinen näkymistä, jotka sen tietoa esittävät. Mallin vastuulla on kaikki sovelluksen toimintalogiikaksi (engl. business logic) luokiteltavat operaatiot, eli toimialueeseen liittyvät rakenteet ja algoritmit. Malli itsessään voi tarjota vain ohjelmallisen rajapinnan näihin toimintoihin. [8, s. 47] [9, s. 14] [10]

Useimmiten sovelluksessa on yksi malli, kun taas muita MVC-mallin osia voi olla, ja monasti onkin, useita (toisin sanoen esimerkiksi näkymäkerros sisältää useita näkymäkomponentteja). Voi olla myös tapauksia, joissa esimerkiksi samalla käyttöliittymällä halutaan esittää tietoa useasta ennalta olemassa olevasta sovelluksesta. Tällöin voidaan ajatella, että MVC-mallin toteuttavaan sovellukseen kuuluu useampi kuin yksi malli. Tämän tyyppisissä sovelluksissa lisähaasteena voi olla mallien tarjoamat toisistaan poikkeavat rajapinnat.

Näkymä MVC:ssä tarkoittaa käyttöliittymää [8, s. 47], joka voidaan ajatella käyttäjän näkymänä ja rajapintana ohjelman mallikerrokseen. Useimmiten näkymä on graafinen, mutta myös esimerkiksi äänen avulla ohjattava ja tilansa esittävä järjestelmä, kuten varaston keräilijän käyttämä sovellus on mahdollinen. Näkymä voidaan monasti kuvata kuvauskielellä, kuten web-sivujen kuvaukseen käytettävä HTML (Hypertext Markup Language) tai Microsoftin kehittämä ohjelmointikielen olioiden alustukseen käytettävä XAML (Extensible Application Markup Language). Näkymä voi koostua uudelleen käytettävistä käyttöliittymäkomponenteista. Näkymä ei saa olla vastuussa mistään ohjelman toimintalogiikkaan kuuluvasta toiminnosta tai algoritmista [8, s. 47].

MVC:n yksi tärkeimmistä tavoitteista on käyttöliittymän ja mallin irrottaminen toisistaan [8, s. 47]. Tällä pyritään esimerkiksi siihen, että saman mallin päälle voidaan toteuttaa useita käyttöliittymiä ja että sovelluksen toimintalogiikka voidaan testata automaattisesti käyttöliittymästä erillään. Esimerkiksi sovellus voitaisiin toteuttaa aluksi natiiviksi käyttöliittymäsovellukseksi, ja myöhemmin samaa mallia hyödyntäen voitaisiin toteuttaa mobiilisovellus, web-käyttöliittymä sekä puheohjauksella toimiva käyttöliittymä ilman, että mallia tarvitsee muokata ja ilman, että mitään toimintalogiikan osaa tarvitsee toteuttaa mihinkään käyttöliittymään kopioimalla sitä aiemmin toteutetusta käyttöliittymästä. Sivuseuraus (joka voi olla varsinaista tavoitetta tärkeämpi) näkymän ja mallin erottamispyrkimyksestä on, että muunkinlaiset muutokset helpottuvat ja sovelluksen rakenteesta tulee selkeän jaon ansiosta paremmin ymmärrettävä, mikä on hyödyksi, vaikka saman mallin päälle ei koskaan toteutettaisikaan useita käyttöliittymiä. Sovelluksen rakenteen ymmärrettävyyttä lisää jo pelkästään sekin seikka, että MVC-malli on hyvin laajasti tunnettu, eli esimerkiksi uuden kehittäjän siirtyessä projektiin hän todennäköisesti tietää heti korkealla tasolla sovelluskoodin järjestelyn logiikan.

Automaattiset testit saadaan MVC-mallia hyödyntävässä sovelluksessa katta-

maan koko toimintalogiikka niin, että koko näkymäkerros voidaan testeissä sivuuttaa, sillä käyttöliittymä toimii vain rajapintana sovelluksen toimintoihin. Ohjelmallisen rajapinnan testaaminen automaattisesti on helpompi toteuttaa ja helpompi ylläpitää kuin graafisen käyttöliittymän automaattinen testaaminen.

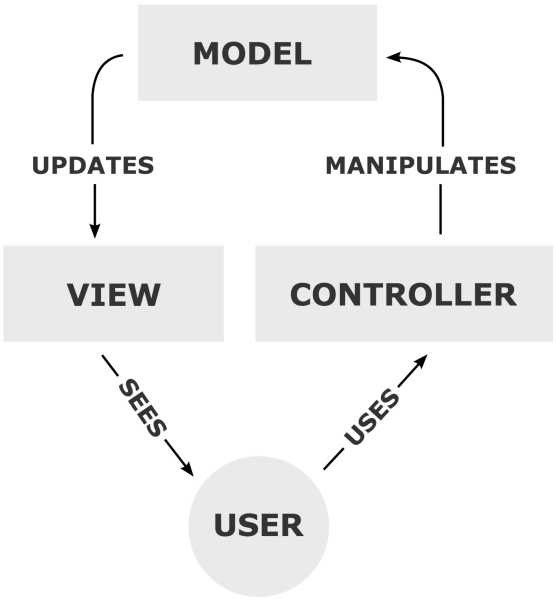
Se, mikä osa sovelluksesta lasketaan toimintalogiikaksi ja mikä käyttöliittymään kuuluvaksi ei aina ole selvää. Epäselvä rajatapaus voisi olla, jos ohjelmassa on lämpötilaa esittävä numerokenttä, jossa halutaan esittää punaisella ja lihavoidulla fontilla tietyn raja-arvon ylittävät lämpötilat. Tässä tapauksessa voitaisiin ajatella, että ohjelman mallin vastuulle kuuluu päätellä, onko lämpötila vaarallisella alueella, mutta näkymään toteutetaan vaarallisen lämpötilan esitystapa. Näkymän tulisi siis saada mallilta vain tieto siitä, onko lämpötila vaarallisella vai normaalilla alueella. Jos saman mallin päälle toteutettaisiin uusi käyttöliittymä, niin vaarallisen lämpötilan raja olisi sama käyttöliittymästä riippumatta, mutta tarkka vaarallisen lämpötilan esitystapa olisi käyttöliittymäkohtainen.

Gamma et al. [9, s. 15] (myös Fowler [5]) korostaa (Smalltalkin) MVC-mallissa mallin ja näkymäkomponentin tai useamman samaan mallia käyttävän näkymäkomponentin muodostamaa tarkkailija-suunnittelumallia. On oleellista, että malli ei tiedä käyttäjistään, koska tällöin malli ja näkymäkomponentit ovat erotettu toisistaan sillä tavalla, että mallin kehityksessä ei ikinä tarvitse huomioida näkymiä. Gamma et al. mukaan MVC-mallissa näkymä voi itsenäisesti päättää päivittävänsä tietonsa mallilta tai tilata mallilta tarkkailija-suunnittelumallin mukaisesti tiedon tilamuutoksista.

Tässä työssä MVC-mallia ja sen eri versioita yleisesti tarkoitettaessa käytetään termiä MV*. Eri MV*-versioissa kolmannen, mallin ja näkymän lisänä olevan, osan tarkka rooli vaihtelee. Seuraavaksi kuvataan ensin MVC:n kontrolleri siten, kuin se yleisimmin on lähteissä selitetty. Tämän jälkeen tuodaan esiin eroavia tulkintoja ja MV*-muunnoksia.

MVC:n kontrollerin rooli on ottaa vastaan käyttäjän ohjelmalle antamat syötteet, kuten nappien painallukset. Kontrolleri päättää miten syötteeseen tulee reagoida ja esimerkiksi valitsee mikä näkymäkomponentti käyttäjälle tulee milloinkin näyttää. Kontrolleri voi noutaa tietoa mallilta kuten näkymäkin. Tämän lisäksi, toisin kuin näkymä, kontrolleri voi muuttaa mallin tilaa mallin paljastaman rajapinnan kautta. Näkymä voi saada esittämänsä tiedot joko kontrollerilta tai suoraan mallilta. Kontrolleri on tietoinen sekä mallista, että siihen itseensä liittyvistä näkymäkomponenteista. Se, miten näkymä reagoi käyttäjän syötteeseen voi muuttua ilman, että näkymään itseensä tarvitsee tehdä muokkauksia, sillä tämä osuus on kontrollerin vastuulla [9]. Kontrolleri voi syötteen perusteella valita, mihin tilaan se ohjaa näkymän, ja tätä reagointitapaa voi kontrollerissa vaihtaa.

Kontrolleri- ja näkymäkomponentit[1] tai oliot toimivat parina ja näitä pareja on tyypillisessä sovelluksessa useita. Jokainen tietyn näkymän elementti on yksi kontrolleri-näkymä-pari ja näistä elementeistä koostuva kokonainen näkymä on myös yksi näkymä-kontrolleri-pari. [5]



Kuva 2.1: Tyypillinen tapa kuvata MVC-malli graafisesti. [38]

Kuvassa 2.1 on esitetty tyypillinen kuvaus MVC-mallista. Kuvassa käyttäjä näkee järjestelmän tilan näkymän kautta, kun taas käyttäjän syötteet ohjautuvat kontrollerin käsiteltäväksi. Vain kontrolleri voi muokata mallin tilaa. Näkymä saa esittämänsä tiedon suoraan mallilta, yleisesti tarkkailija-suunnittelumallilla, jossa malli on tarkkailtava, jonka tilamuutoksia näkymä (tai näkymät) tarkkailee.

MVC:n muunnos MVVM on Microsoftin Gossmanin vuonna 2005 lanseeraama suunnittelumalli ja termi [10]. Termi koostuu osista malli (engl. Model) ja näkymä (engl. View) kuten MVC, mutta kontrollerin tilalla MVVM:ssä on View Model, jolla tarkoitetaan mallista tiettyä näkymää varten erikoistettua alimallia. (View Modelille ei ole vakiintunutta suomen kielistä vastinetta. View Modelista voidaan käyttää lyhennettä VM.) MVVM oli tarkoitettu käyttöliittymäsuunnittelumalliksi erityisesti Microsoftin uuden työpöytäsovelluksien kehitykseen tarkoitetun WPF:n (Windows Presentation Foundation) kanssa, mutta myöhemmin samaa termiä on käytetty myös muissa yhteyksissä. Tämän työn aiheen kannalta merkittävin esimerkki on Knockout-sovelluskehys (luku 4.2).

[1]Näkymällä tai kontrollerilla voidaan kontekstista riippuen tarkoittaa joko koko MVC-mallin kerrosta tai yksittäistä näkymä- tai kontrollerikerroksen oliota, esimerkiksi yhtä dialogia ja siitä vastaavaa kontrolleria.

Gossman määrittää MVVM:n merkittäväksi muutokseksi aiempiin MVC-variantteihin verrattuna sen, että WPF ja MVVM yhdessä mahdollistivat sen, että käyttöliittymän ulkoasuosuuden pystyi tuottamaan graafiseen suunnitteluun erikoistunut henkilö ilman, että hänen tarvitsi tietää ohjelman muusta koodista. Vastaavasti ohjelman toimintalogiikan toteuttajien ei välttämättä tarvitse puuttua XAML-ulkoasutiedostoihin. Lisäksi ohjelman toimintalogiikan koodi ja käyttöliittymä voidaan kehittää omilla, asiaan erikoistetuilla työkaluilla. Tämän erotuksen mahdollisti WPF:n tarjoama sidontamekanismi (engl. binding). WPF:ssä näkymän kenttiä voidaan sitoa suoraan malliin tarjotulla yksinkertaisella syntaksilla. Tämä on toimiva tapa, jos kyseessä on esimerkiksi yksinkertainen lippumuuttuja tai numeroarvo. Sidonta voi olla joko kaksisuuntainen, jolloin tiedon päivitys kumpaan tahansa sidonnan osapuoleen johtaa sen päivittymiseen myös toisessa tai yksisuuntainen, jolloin päivitykset kulkevat automaattisesti vain toiseen suuntaan. Monimutkaisia näkymän ja mallin liitoksia varten View Modeliksi erikoistetaan mallista alimalli, joka tarjoaa ominaisuuksinaan (engl. property) arvot, joihin näytön esittämiä arvoja voi sitoa. View Model sisältää myös tarvittavan muunnoslogiikan näkymän ja mallin välillä, jos tämä logiikka on luonteeltaan sellaista, että se ei selvästi kuuluu mallin vastuulle, muttei näkymäänkään. Esimerkiksi, jos näkymän kannalta jokin asia on yksinkertainen päällä-pois-valinta, mutta malliin tämä kuvautuu monimutkaisemmin ja useaan kenttään, niin View Model on oikea paikka toteuttaa muunnos tai kuvaus.

Lisäksi View Model pitää yllä näkymän tilatietoa. Esimerkiksi, onko näkymä editointi-moodissa, tai mikä valinta tietyssä listakomponentissa on päällä (jos valinta ei suoraan vaikuta malliin). MVVM:ään kuuluu Gossmanin mukaan periaate, että näkymän tilatiedot reititetään saman näkymän eri komponenteille aina VM:n tai varsinaisen mallin kautta. Esimerkiksi käyttöliittymässä voi olla lista sekä muokkausruutu, jonka muokattava sisältö määräytyy listasta valitun elementin perusteella. Jos muokkausruutu saisi tiedon suoraan listakomponentilta, olisi näkymän komponentit sidottu yhteen, mikä aiheuttaisi paljon muokkaustyötä, jos haluttaisiin lisätä uusia komponentteja, jotka myöskin riippuvat listan valinnasta, tai haluttaisiin vaihtaa listakomponentti toiseen. Parempi, MVVM-mallin suosittama tapa on reitittää eri käyttöliittymäkomponenteille toistensa tilatiedot VM:n kautta, jolloin tilaliitokset kulkevat yhden pisteen kautta sen sijaan, että eri käyttöliittymäkomponenttien välille syntyisi monimutkaisia liitosverkostoja. Tämä toteutuu käytännössä sidontaa käyttämällä automaattisesti, kun samaan VM:n ominaisuuteen sidotaan useita näkymäkerroksen kenttiä.

Kritiikkinä MVVM-mallista mainittakoon Gossmanin itsensä mainitsemat seikat: suunnittelumallin toteutuksen vaatima ylimääräinen koodi (lähinnä VM-kerros) voi olla hyötyynsä nähden turhaa, monimutkaistavaa painolastia pienehköissä sovelluk-

sissa ja sitominen voi olla suorituskykynäkökulmasta raskasta. [11]

Mitä MVC-mallin eri osat tarkoittavat, on tulkintaeroja, mikä johtunee osittain eroavaisuuksista eri toteutustekniikoiden, kuten esimerkiksi WPF ja web-sivu, välillä. Web-sivujen palvelintoteutuksen tekoon tarkoitetun Django-sovelluskehyksen tekijät tulkitsevat MV*-mallin osat niin, että näkymä tarkoittaa sitä, mikä tieto näytetään, eikä miten tieto näytetään [30], ja näin Djangon näkymä toteuttaa sellaisen osan, jonka voisi mieltää myös MVC-mallin kontrolleriksi. Djangon käyttämä malli on nimetty MVT-malliksi, joka muodustuu mallista (model), näkymästä (view) ja templatesta (jolla tarkoitetaan sivupohjaa). Tässä mallissa template vastaa tyypillisen MVC-mallin näkymää. Djangon MVT-malli on selostettu yksityiskohtaisemmin luvussa 4.6[2].

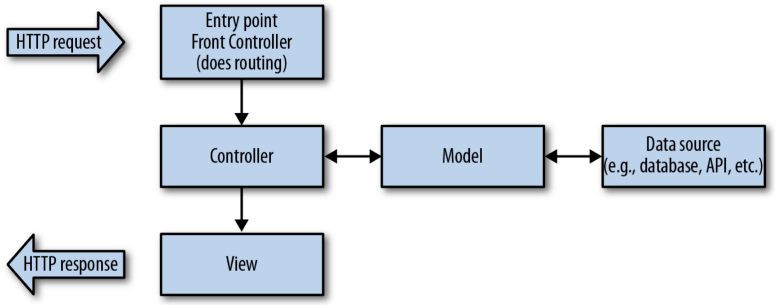
Angular.js-web-sovelluskehyksen puolestaan mainitaan toteuttavan MVW-mallin. Nimen M ja V ovat samat malli ja näkymä kuin yleisesti MV*-malleissa. W:llä tarkoitetaan mitä tahansa (engl. whatever), jolla sovelluskehyksen tekijät ovat halunneet tuoda esille näkemyksensä, että suunnittelumallin yksityiskohtiin keskittymisen asemesta sovellus tulisi toteuttaa kulloinkin parhaimmalta vaikuttavalla tavalla, kuitenkin niin, että se on selkeästi jaettu tarkoituksenmukaisiin osiin [17]. Angular.js-sovelluskehyksen tarkempi kuvaus on luvussa 4.4.

Laajasti käytetyssä kolmen kerroksen mallissa (engl. three-layer pattern), jonka kerrokset ovat näkymä, malli ja DAL (data access layer, yleensä tietokannan kanssa kommunikoinnista vastaava sovelluksen kerros), on erona MVC-malliin se, että sen käyttöliittymäkerroksessa on mukana enemmän ohjelman toiminnallisuutta, kuin MV*-mallin näkymässä [8, s. 51]. Esimerkkinä kolmen kerroksen mallin käytöstä mainittakoon Microsoftin Windows Forms, jossa käyttöliittymäluokalla on aina vastaava ns. code behind -kooditiedosto, johon on esimerkiksi toteutetava metodi, joka suoritetaan kun käyttöliittymän tiettyä nappia painetaan. Code behind -luokassa ajettavan koodin automaattinen testaus tulee suorittaa käyttöliittymän läpi. Code behind -metodeja voidaan kutsua myös automaattitestijärjestelmästä, mutta kutsuttavan metodin vaikutusten todentaminen ei ole aina helppoa, sillä käyttöliittymän tekstikenttiä ynnä muita elementtejä päivitetään code behind -metodien koodissa, eikä näiden elementtien arvoihin voida päästä käsiksi erillisestä testikoodista esimerkiksi paluuarvojen kautta. Microsoftin Windows Formsia uudempi, myöskin työpöytäsovellusten tekoon tarkoitettu, WPF tarjoaa mahdollisuuden jättää Windows Formsia vastaava code behind pois. Tällöin vastaava koodi voidaan View Modelissa testata automaattisesti huomattavasti helpommin, sillä View Modelin rajapintana tarjoamia ominaisuuksia, joihin käyttöliittymä sidotaan, on helppo käsitellä myös ohjelmallisesti. Edellä kuvattu työpöytäsovelluksiin liittyvä esimerkki kuvaa myös

[2]MVT-malli on selvästi Djangoon liittyvä, eikä yleisemmin käytetty, jonka takia mallin selostus on loogisempaa samassa luvussa, jossa Django-sovelluskehys käsitellään.

yleisemmin MV*-mallin etua testauksen kannalta.

2.1 Web-käyttöliittymien suunnittelumalleista



Kuva 2.2: Tyypillinen palvelinpään web-käyttöliittymän MV*-malli. [19, s. 13]

Toisin kuin ympäristössä, jossa MVC-malli alunperin oli käytössä, on MVC:n soveltamisessa web-ympäristössä omana erityispiirteenä webin asiakas-palvelin-jaottelu. MVC:n näkymän esittäminen on selkeästi aina asiakkaan harteilla, mutta kontrollerin ja mallin sijainti asiakas-palvelin-akselilla on päätettävä toteutuskohtaisesti. [15] Tehtyä päätöstä jaottelusta voi olla hankala muuttaa myöhemmässä vaiheessa, koska tyypillisesti asiakas- ja palvelintoteutukset toteutetaan eri ohjelmointikielillä.

Tyypillinen ratkaisu web-ympäristöön sovelletussa MVC:ssä on ollut toteuttaa kontrolleri ja malli palvelimella. Asiakas-palvelin-mallista ja kommunikoinnista HTTP:n (Hypertext Transfer Protocol) avulla seuraa tällöin se, että näkymä ei ole yhteydessä malliin sen jälkeen, kun näkymä on luotu ja lähetetty HTTP-vastauksena asiakkaalle. Alkuperäisen MVC:n kaltaista sidontaa mallin ja näkymän välille ei siis voida toteuttaa, vaan näkymä jää esittämään mallin tilaa siltä hetkeltä, kun näkymä muodostettiin. [19, s. 10]

Kuvassa 2.2 esitetään, kuinka MV*-malli tyypillisesti ymmärretään web-käyttöliittymän yhteydessä. Lähde [19] käytti kuvaa Ruby on Rails -ohjelmistokehyksen MV*-mallin kuvaamiseen, mutta sama idea pätee hyvin myös esimerkiksi Django-ohjelmointikehykseen. Kuvasta ilmenee, kuinka asiakas (ja siten näkymä) on yhteydessä kontrolleriin HTTP:n välityksellä tekemällä pyyntöjä. Valmiiksi muodostettu näkymä palautetaan HTTP-vastauksena. Ehdottomasti suurin osa toteutuksesta on siis palvelimella.

Vaikka lähde [19] käyttää MVC-termiä kuvan 2.2 kaltaisen mallin toteuttamiseen, se mainitsee, että kyseinen MVC:n kaltainen malli on oikemmin tyypiltään Model2. Toisaalta esimerkiksi [22, s. 26] ei käytä MVC:tä ja Model2:ta toisensa poissulkevina sanoessaan, että Struts-web-sovelluskehys toteuttaa MVC:n ja pohjautuu Model2:een. Tässä työssä käytetään MVC-termiä myös web-ympäristön toteutuksista.

Web-sivujen toteutusmallia kuvataan usein karkealla tasolla sen perusteella, kuinka paljon toiminnallisuutta on toteutettu palvelimella ja kuinka paljon asiakaspään selaimella ajettavana ohjelmana (kuva 1.2 sivulla 6). Ääripäinä ovat web-sivuina toteutetut dokumentit, joissa selaimen ainoa tehtävä on jäsentää palvelimen palauttama HTML-tiedosto ja esittää sen sisältö. Toisessa päässä on single-page-sovellus, jossa suuri osa toteutuksesta on siirretty asiakaspäähän, ja palvelin on vain tietovarasto ilman sovelluksen toimintalogiikkaa. Ensin kuvatusta ääripäästä käytetään nimitystä kevyt (tai ohut) asiakas (engl. thin client), jonka vastakohtana on paksu asiakas (engl. thick client). Samasta asiasta käytetään myös termejä asiakas- tai palvelinpainotteinen toteutus. Jaottelu tällä akselilla on liukuva ja web-sivuissa voidaan tunnistaa monia välimalleja.

2.2 Single-page-toteutustavasta

Tässä luvussa käsitellään single-page-sovellusten toteutusmalleja sekä käydään yleisesti läpi single-page-sovelluksille tyypillisiä ominaisuuksia, jotka aiheutuvat sivujen arkkitehtuurista, joka on väistämättä erilainen verrattuna perinteisiin web-sivuihin. Tässä luvussa ei käsitellä tarkasti niitä toteutusteknologioita, joilla tässä luvussa kuvatut yleiset single-page-ominaisuudet saavutetaan.

Single-page-sovelluksissa käytetään tavallisesti samankaltaisia MV*-malleja kuin muunkin tyyppisissä käyttöliittymäsovelluksissa. Huomioitavaa on kuitenkin, että vaikka suunnittelumalli olisi sama ja sen loogiset osat samat, on niiden toteutuksessa ja yksityiskohdissa silti väistämättä eroja johtuen teknologioiden erityispiirteistä (esimerkiksi tilaton HTTP).

Takada huomauttaa, että single-page-sovellusta kehitettäessä on varottava, ettei arkkitehtuuriin muodosteta totutun MVC-mallin kannustamana kontrolleri-nimistä kerrosta vain siksi, että MVC-malliin tällainen kuuluu [21]. Takada ei kritisoi itse MVC-mallia, vaan sitä, että hänen kokemuksensa mukaan web-arkkitehtuureissa on joskus taipumus nimetä kaikenlainen sekavakin näkymää ja mallia yhteenliittävä koodi MVC-mallin mukaiseksi kontrolleriksi. Onkin tärkeää, että luotavan sovelluksen arkkitehtuurin suunnittelussa havaitaan ongelma, johon pyritään löytämään sopiva suunnittelumalli, sen sijaan, että valitaan suunnittelumalli jonka raameihin sovellus yritetään pakottaa. Single-page-sovelluksen rakenne ja siinä käytettävät tekniikat ovat siinä määrin erilaisia verrattuna esimerkiksi sellaiseen ohjelmointiympäristöön, johon alkuperäinen Smalltalkin MVC-malli kehitettiin, että ei voida olettaa, että sama rakenne sellaisenaan on toimiva. On löydettävä tarkemmin ne syyt, miksi MVC-malli on todettu toimiviksi niissä ympäristöissä, joissa sitä on jo pitkään käytetty. Tällä tavalla nämä parhaat suunnittelumallit voidaan ottaa soveltuvin osin käyttöön single-page-sovelluksen arkkitehtuuriin. Samaan tapaan perinteisemmissä web-sovelluskehyksissä, kuten Django tai Ruby on Rails, MV* on toteutukseltaan

erilainen kuin työpöytäsovelluksissa käytetyt, mutta vastaavasti nimetyt toteutusmallit, vaikka samankaltaiset pääosat onkin erotettavissa.

DOM (Domain Object Model) on selaimen tarjoama API (Ohjelmointirajapinta, engl. Application Programming Interface), jonka kautta voidaan olla vuorovaikutuksessa HTML-dokumentin olioiden kanssa. Single-page-web-sivu ei ole staattinen HTML-dokumentti, vaan kokonainen selaimessa ajettava sovellus, joka pystyy tarvittaessa muokkaamaan DOM:a myös kokonaan ilman kommunikointia palvelimen kanssa. Selain saa normaalisti web-sivun HTTP:n välityksellä web-palvelimelta. Single-page-web-sivun tapauksessa yksittäisellä HTTP-pyynnöllä pyydetään palvelimelta kokonainen selaimessa ajettava sovellus, joka koostuu minimissään HTML-tiedostosta ja siihen upotetusta tai tiedostona liitetystä JavaScript-ohjelmakoodista. Mikä tahansa JavaScript-koodi ei kuitenkaan tee sivusta single-page-sovellusta, vaan web-sovelluksen käyttö ei saa vaatia kokonaisia uusia sivulatauksia palvelimelta käytön aikana. Käytännössä tämä edellyttää MV*-mallin osien toteuttamista selaimessa ajettavaan sovellukseen lukuunottamatta malli-kerrosta tai osaa siitä. Koska palvelimen ei ole tarkoitus palauttaa valmiita näkymiä, jää ainoaksi vaihtoehdoksi käyttäjän syötteisiin reagointi ja näkymien muodostus asiakassovelluksessa.

Single-page-arkkitehtuurilla toteutetun web-sivun oletetut suurimmat suoraan käyttäjälle näkyvät hyödyt ovat toteutustekniikan mahdollistama sivun käytön sulavuus ja nopeus verrattuna vastaavaan, mutta perinteisellä web-arkkitehtuurilla toteutettuun sivuun. Nopeusedut saavutetaan, koska jokainen interaktio sivulla ei vaadi synkronista sivulatausta palvelimelta. Niinkin pienet, kuin satojen millisekuntienkin luokassa olevat nopeutukset sivujen latausajoissa vaikuttavat positiivisesti web-sivun käyttäjien käytökseen. [47].

Yleisesti käyttöliittymien vasteajoissa voidaan käyttää karkeita raja-arvoja 100ms, 1s ja 10s [18].

- 100ms rajan alle oleva viive luo käyttäjälle vaikutelman, että järjestelmä toimii välittömästi, eikä esimerkiksi odotusindikaattoria tarvita.

- 100ms ja 1s välillä olevat viiveet eivät häiritse käyttäjän keskittymistä järjestelmän käyttöön, mutta käyttäjä huomaa selvästi, ettei järjestelmä ole välitön. Odotusindikaattori ei ole alle 1s viiveillä välttämätön, sillä toiminnon ja seurauksen suhde käyttöliittymässä on vielä intuitiivinen viiveen puolesta.

- Yli 10s viiveet järjestelmässä saavat käyttäjän keskittymisen herpaantumaan; käyttäjä kerkiää siirtämään ajatuksensa pois sovelluksen käytöstä viiveen aikana. Tämän suuruisilla viiveillä tulisi esittää käyttäjälle odotusindikaattori, joka kertoo jollain tavalla kauanko odotusta on vielä jäljellä.

Fowler erottelee termit vasteaika (engl. response time) ja reagoivuus (engl. responsiveness) [7, s. 16]. Vasteaika on se aika, joka järjestelmällä kestää, että se saa

prosessoitua pyynnön, esimerkiksi käyttöliittymän napin painalluksen. Reagoivuus puolestaan tarkoittaa sitä, kuinka nopeasti järjestelmä osoittaa, että se on vastaanottanut pyynnön. Pahimmillaan reagoivuus on sama kuin vasteaika, mutta jos vasteaika ei ole käyttäjän näkökulmasta välitön, olisi reagoivuuden syytä olla vasteaikaa nopeampi. Järjestelmä voi esimerkiksi osoittaa vastaanottaneensa pyynnön esittämällä tiimalasi-kuvakkeen kursorina tai näyttämällä edistymispalkin. Ohut asiakas -tyylisissä web-sivuissa reagoivuus on sama kuin palvelimen vasteaika (koska sivulatauksen aikana selain ei voi tehdä muuta kuin odottaa), kun taas paksu asiakas -tyyppisissä web-sivuissa reagoivuutta voidaan parantaa. Fowlerin mukaan huono reagoivuus voi turhauttaa käyttäjiä, vaikka järjestelmän vasteaika olisikin verrattain hyvä. Kun selain kommunikoi palvelimen kanssa, tuo viestin välitys internetin yli aina viivettä. Fowler suositteleekin järjestelmän etäkutsujen määrän minimointia.

Single-page-toteutuksella voidaan saavuttaa sekä vasteaika-, että reagoivuusetuja, sillä suurin viipeen aiheuttaja web-sivujen käytössä on datan siirto verkon yli. Single-page-sivut voivat minimoida siirrettävän datan määrän, koska asiakaspään sovellus voi pyytää palvelimelta vain tarvitsemansa tiedon. Reagoivuutta voidaan parantaa single-page-sivuilla, koska datan siirto verkon yli tapahtuu taustalla ja käyttöliittymä voi siirron ajan pysyä reagoivana ja esimerkiksi esittää käyttäjälle odotusanimaation avulla, että taustalla tapahtuu tiedon lautausta.

Taustalla asynkronisesti tapahtuva tiedonsiirto mahdollistaa myös sen, ettei single-page-sovelluksen asiakasosan tarvitse hakea dataa palvelimelta samaa tahtia, kuin vastaava perinteinen web-sovellus tekisi, vaan asiakassovellus voi tallentaa itselleen tietoa välimuistiin ja yrittää arvata etukäteen, mitä tietoa seuraavaksi tarvitaan. Single-page-sovelluksen verkkoliikenteen määrää tai käyttöliittymän nopeutta voidaan siis entisestään parantaa, sillä välimuistia ja ennakointia hyödynnettäessä HTTP:n yli siirrettävä tieto ei ole vain eri muotoista kuin perinteisessä web-sovelluksessa, vaan HTML:n ylimääräisen kuorman (jos haettaisiin uudestaan koko sivu) lisäksi siirrettävän tiedon määrä ja siirron ajankohta voi olla optimoitu. Esimerkiksi mobiilisovelluksissa mobiililaitteiden tapa käyttää radiota aiheuttaa sen, että on laitteen akun keston kannalta edullista siirtää kerralla suuri määrä tietoa (vaikka osa tiedosta olisi turhaa), kuin vähän dataa useasti. (Mobiililaitteen radion käytön energiankulutuksen ja sovellusten tietoliikenteen käytön problematiikkaa käsittelee esimerkiksi Athivarapu et al. [1].) Välimuistin ja ennakoivan datan latauksen käyttö ei silti ole tarpeen, jotta single-page-sovelluksella voidaan saavuttaa käytettävyysetuja, eikä tämän kaltaisia optimointeja luultavasti usein tehdä.

Esimerkki välimuistin hyödyntämisestä web-sovelluksesta on Googlen Gmail-sähköpostipalvelun käyttöliittymä, jossa sähköpostiviestejä voi selata sivu kerrallaan. Uusien sivujen sisältö ladataan sivua vaihdettaessa, mutta aiemmin selatut sivut

tallennetaan välimuistiin, eikä niiden sisältöä noudeta palvelimelta uudelleen, jos käyttäjä selaa uudelleen jo aiemmin käydylle sivulle. Tämä voidaan todeta tarkkailemalla verkkoliikennettä käytettäessä Gmail-palvelua. Toinen yleinen optimointiesimerkki on useallakin sivustolla käytettävä tapa ladata vieritettävään sivuun sisältöä lisää sitä mukaa, kun käyttäjä vierittää sivua alaspäin, niin että sisältöä on aina tietty määrä puskurissa (esim. Facebook). Parhaimmillaan tällainen vieritys toimii niin, että käyttäjän näkökulmasta tietoa riittää loputtomasti ilman lataus- tai sivunvaihtokatkoja, mutta sellaista dataa, jota käyttäjä ei katso, ei ladata koskaan kuin vain marginaalinen määrä, eli verkkoliikennettä on vain sen verran kuin dataa oikeasti tarvitaan.

Single-page-toteutuksen haittoja ovat ainakin seuraavat.

- Jotkin totutut selaintoiminnot, kuten takaisin-toiminto ja kirjanmerkit täytyy huomioida erikseen single-page-toteutuksessa tai hyväksyä niiden toimimattomuus tai toimiminen käyttäjän kannalta odottamattomalla tavalla.

- Single-page-sovellus vaatii toimiakseen modernin selaimen sekä se nojaa evästeisiin (engl. cookie, HTTP-viestien mukana kuljetettava vapaaehtoinen tilatieto) ja JavaScriptiin, jotka pohjimmiltaan eivät ole sellaisia, joiden voi olettaa olevan aina käytettävissä.

- Myös web-sivujen koneellinen jäsennys, jota esimerkiksi hakukoneiden robotit suorittavat, ei välttämättä toimi single-page-sovelluksessa, eivätkä hakukoneet osaa indeksoida sovelluksen koko sisältöä, koska monet robotit eivät aja JavaScriptiä lainkaan. Viimeaikoina tähän tosin on tullut parannuksia, eikä asia ole yksiselitteisesti näin. [37]

Kirjanmerkkien ja takaisin-toiminnon haastavuuden taustalla single-page-sovelluksissa on sama syy, joka on kierrettävissä. Single-page-sovelluksissa nimensä mukaisesti on vain yksi sivu, kun taas edellä mainitut toiminnot nojaavat URL:n (Uniform Resource Locator, HTTP:n yhteydessä arkinen nimitys on web-osoite) muutoksiin. Yleinen ratkaisuvaihtoehto tähän ongelmaan nojautuu HTML:n ankkureihin. URL:n perässä (#-merkillä erotettuna) olevan ankkurin avulla voidaan kuvata websivun tilaa vastaavalla tavalla kuin normaalisti URL:ssa itsessään. Kirjanmerkit ja takaisin-toiminto huomioivat ankkurit vastaavasti kuin muunkin URL:n. Alunperin ankkurielementit on tarkoitettu mahdollistamaan linkitys yhden HTML-dokumentin tiettyyn kohtaan, esimerkiksi väliotsikkoon, mutta single-page-sovelluksissa toimintoa voidaan hyödyntää kuvaamaan laajemmin näytettävän sivun tilaa.

Modernin selaimen vaatimusta voidaan pitää marginaalisena haittana, sillä nykyisin käytännössä esimerkiksi jokaisen uudehkon älypuhelimen selain tukee erittäin hyvin JavaScriptiä ja evästeitä. Voi silti olla olla joitain tilanteita, joissa asia tulee huomioida, kuten sivut, joiden saatavuus tulee taata laajalle käyttäjäjoukolle.

Voitaneen kuitenkin olettaa, että web-sivujen vaihtoehtoisia jäsentämiskeinoja pyritään parantamaan toimimaan single-page-tyyppisten web-sivujen kanssa, sillä pitkän ajan trendi on, että web-sivut on toteutettu entistä enemmän JavaScriptillä ja single-page-sovelluksina tai niiden omaisesti. Näin voidaan olettaa, sillä web-sivut sisältävät yhä monimutkaisempia ja kehittyneempiä käyttöliittymiä ja toisaalta selainliitännäisteknologioiden käyttö on vähenemässä.

3. JÄRJESTELMÄN TEKNINEN TAUSTA

Tässä luvussa esitellään ne web-teknologiat, jotka yleisesti ovat hyvin tyypillisiä single-page-web-sivuille, mutta jotka eivät silti ole millään tavalla sidottuja pelkästään single-page-ajatukseen, vaan niitä voi hyödyntää, ja laajasti myös onkin hyödynnetty yksittäin ja muussa kuin single-page-tarkoituksessa. Toisin sanoen tämä luku kuvaa ne perustekniikat, joiden varaan single-page-arkkitehtuuri on ollut mahdollista rakentaa. Lisäksi käsitellään omina alilukuinaan muita huomion arvoisia seikkoja.

3.1 JavaScript ja AJAX

JavaScriptin käyttö yhdistetään ehdottomasti eniten selaimiin ja niissä esitettäviin web-sivuihin, jossa käytössä JavaScript onkin käytännössä ainoa mahdollisuus toteuttaa ohjelmia (jos ei huomioida selaimessa erilaisten lisäosien kanssa ajettavia sovellusteknologioita, kuten Flash). Tämän takia JavaScript on hyvin käytetty kieli ja nykyään sen käyttömahdollisuus on laajentunut pelkistä selainsovelluksista myös esimerkiksi web-palvelimen toteutukseen [39]. Selaimessa JavaScript-koodista on esimerkiksi pääsy selaimen tarjoamaan DOM-rajapintaan ja mahdollisuus tehdä HTTP-pyyntöjä web-palvelimelle [3]. Yllä mainitut ominaisuudet ovat mahdollistaneet niin kutsutun AJAX:n (Asynchronous JavaScript and XML), joka tarkoittaa tapaa tehdä palvelimelle pyyntö JavaScript-koodista, ja sijoittaa pyyntöön saatu vastaus ohjelmallisesti selaimen esittämään näkymään DOM-rajapinnan kautta. Näin voidaan dynaamisesti muokata ja päivittää selaimen näyttämää web-sivua ilman kokonaisia sivulatauksia. (Näitä aiheita on käsitelty tämän työn tekijän kandidaatintyössä [13].)

AJAX-tyyppisiä ominaisuuksia halutaan yleensä toteuttaa niiden tuomien käytettävyysetujen takia. Tällaiset lisäykset muuten ohuena asiakkaana toteutettuun websivuun voivat kuitenkin helposti rikkoa MV*-mallia ja siirtää esimerkiksi toimintalogiikkaa tai kontrollerille kuuluvaa toteutusta osin näkymän harteille, jos näkymään siirretty AJAX-toiminnallisuus ei käsittele pelkästään MV*:n näkymälle kuuluvia asioita. Vaikka monissa tapauksissa suuri osa tai kaikki single-page-toteutuksen loppukäyttäjälle näkyvistä hyödyistä olisi saavutettavissa lisäämällä ohueen asiakkaaseen joitain AJAX-toiminnallisuuksia, on täysin single-page-sovelluksena tehty toteutus silti ohjelmiston arkkitehtuurin kannalta parempi vaihtoehto. Single-page-

sovelluksessa MV*-malli voidaan säilyttää ehyenä AJAX:n hyödyntämisestä huolimatta, mikä auttaa sovelluksen toteutuksen rakenteen ymmärrettävyydessä ja näin ollen helpottaa ylläpitoa.

3.2 REST-web-sovelluspalvelu

Ihmiskäyttäjien selaimella käyttämien web-sivujen lisäksi webistä löytyy paljon ohjelmien käytettäväksi tarkoitettuja web-sovelluspalveluita (engl. Web service). Esimerkiksi verkkokauppa Amazonin tarjontaa voi selata normaalin web-sivun asemesta myös ohjelmallisen rajapinnan kautta. Tämän kaltaisia ohjelmalliseen käyttöön tarkoitettuja web-sovelluspalveluita yhdistää se, että niiden tiedonvälityksen protokollana toimii lähes poikkeuksetta HTTP ja hyvin usein data välitetään HTTP-viesteissä XML-muodossa (Extensible Markup Language) [20, s. 4]. Toinen suosittu datan muoto on JSON (JavaScript Object Notation), jonka etuna on helppo jäsennettävyys JavaScript-olioiksi. Sekä XML että JSON ovat tapoja esittää kirjoitusmerkeillä rakenteellista tietoa ohjelmointikieliriippumattomasti. JSON on usein syntaksinsa ansiosta hieman tiiviimpi tapa esittää tietoa kuin XML.

Tyypillinen tapa käyttää web-soveluspalveluita on hakea haluttu tieto tai tehdä jokin toiminto oman web-sovelluksen palvelinpään toteutuksessa, joka vastaa oman web-sovelluksen asiakaspään HTTP-pyyntöön. Tällöin web-sovelluspalvelu on kolmannen osapuolen tarjoama. Vastaavia tekniikoita voidaan kuitenkin käyttää myös tarjoamaan dataa tai toimintoja asiakaspään sovellukselle, ja web-sovelluspalvelun tarjoaja voi kolmannen osapuolen sijaan olla oma palvelin. Tällä tavalla single-page-sovelluksen palvelintoteutus ei enää tarjoilekaan valmiiksi muodostettuja, ihmiskäyttäjälle tarkoitettuja HTML-näkymiä selaimelle, vaan selaimessa ajettava single-page-sovellus on yhteydessä omaan alkuperäpalvelimeensa jollakin web-sovelluspalvelu-protokollalla ja hakee sieltä tarvitsemiaan tietoja, jotka single-page-sovellus itse muokkaa ihmiskäyttäjälle esitettävään muotoon.

REST (Representational State Transfer) on tapa toteuttaa web-sovelluspalveluja. REST-rajapinnalle ominaista on paljastaa rajapinnan kautta käsiteltävissä olevat resurssit URL:n avulla. URL kuvaa resurssin (substantiivi) kun taas resurssiin kohdistettava toiminto (verbi) kerrotaan URL:iin kohdistettavassa HTTP-pyynnön tyypissä. Myös HTTP-tilakoodeja hyödynnetään REST-rajapinnassa. [20] Esimerkiksi HTTP 404 -tilakoodi vastauksessa kertoo, että pyynnön URL:lla kohdennettua resurssia ei ole olemassa.

REST-rajapinta soveltuu parhaiten CRUD-tyyppisiin käyttökohteisiin CRUD-operaatioiden ja HTTP-verbien helpon kuvauksen ansiosta. Lyhenne CRUD tulee termeistä luo (engl. create), lue (engl. read), muokkaa (engl. update) ja poista (engl. delete), jotka kuvaavat neljää eri vuorovaikutusta dataan. Sovelluksia tai rajapintoja, joiden kautta voidaan tarkastella, luoda, muokata tai poistaa erilaisia tietueita

voidaan kutsua CRUD-tyyppisiksi. Esimerkiksi palauteolioiden luontia ja tarkastelua sisältävä sivu on CRUD-tyyppinen, kun taas tietokonepeli usein ei ole.

REST-rajapinnassa CRUD-operaatiot on kuvattu HTTP-verbeiksi seuraavasti:

CRUD	HTTP
create	POST (tai PUT)
read	GET
update	PUT (tai PATCH)
delete	DELETE

W3C jakaa web-sovelluspalvelut karkealla tavalla kahteen kategoriaan: REST, jossa HTTP-verbit määräävät suoritettavissa olevat operaatiot ja sellaiset, joissa suoritettavat operaatiot voidaan määrittää vapaasti ja palvelukohtaisesti [2].

Myös Richardson ja Ruby painottavat web-sovelluspalveluiden jaottelussa sitä, että REST hyödyntää HTTP:n omia metodeja. He erottavat REST:n lisäksi toiseksi pääryhmäksi web-sovelluspalvelut, jotka ovat periaatteeltaan RPC-protokollia (remote procedure call, etämetodikutsu), joissa operaation suorittamiseksi tarvittava tieto kuljetetaan esimerkiksi osana URL:a tai viestinä HTTP POST -pyynnön kuormana [20]. Tämä on hyvin samankaltainen jaottelu kuin W3C:n.

3.3 Single-page-palvelin

Single-page-sovellukset ovat siinä mielessä hyvin lähellä natiiveja työpöytäsovelluksia, että suoritettava sovellus sijaitsee kokonaisuudessaan paikallisessa (asiakkaan) ympäristössä ja sovelluksen käyttäjän oma tietokone ajaa sovellusta itsenäisesti. Miksi sitten sovellus halutaan toteuttaa single-page-web-sivuna eikä natiivina työpöytäsovelluksena esimerkiksi .NET WPF:llä? Mikowski ja Powell [16] tarjoavat tähän syitä. Tärkeimpinä ovat sovelluksen helppo päivitettävyys (version päivitys palvelimelle usean erillisen työaseman sijaan) ja sovelluksen kohdealueen asettama tarve hajautetuille käyttäjille ja keskitetylle datalle. Web-palvelin-teknologiat tarjoavat erittäin hyvät ja tutut lähtökohdat viimeksi mainitun seikan toteuttamiseen.

Single-page-palvelimen tarvitsee tukea vain yhtä, uusinta versiota selainsovelluksesta. Jos esimerkiksi natiivina työpöytäsovelluksena toteutettu sovellus kommunikoi palvelimen kanssa, muodostuu ongelmaksi, että samaan palvelimeen voi olla yhteydessä monia eri versioita samasta sovelluksesta, sillä mikään ei pakota käyttäjää päivittämään työpöytäsovellusta. Single-page-sovelluksissa vastaava ongelma on huomattavasti pienempi, sillä käyttäjän sovellus päivittyy uusimpaan versioon aina single-page-sovelluksen latauksen yhteydessä. Marginaalisena ongelmana on silti se, että käyttäjällä voi olla single-page-sovellus käytössä niin, että aiemmin käyttöön ladatun sovelluksen käyttö jatkuu tehdyn palvelinpäivityksen yli. Sovelluskohtaisesti tulee arvioida, onko tässä tilanteessa mahdollisuus aiheutua sellaisia ongelmia tai

virheitä, että tilanteeseen tulisi varautua esimerkiksi kuljettamalla versiotunnistetta kaikissa HTTP-kutsuissa.

Single-page-sovelluksissa palvelimen rooli supistuu ja sen tehtäväksi jää lähinnä dataan liittyvien yksityiskohtaisten pyyntöjen käsittely. Palvelimen voisi siis nähdä koko web-sovelluksen teknologiapinossa olevan abstraktiokerros tietokannan päällä; palvelin tarjoaa selaimen ajamalle sovellukselle soveltuvan rajapinnan tietokannan sisältöön. Tämän lisäksi palvelimen tulee vastata käyttäjien autorisoinnista ja autentikaatiosta, sekä tallennettavien tietojen tarkastuksesta, eli validaatiosta [16, s. 230], sillä ei voida luottaa, että palvelimelle tulevat HTTP-kutsut tulisivat aina toteuttajan suunnittelemasta asiakassovelluksesta.

Vaihtoehtona yllä kuvatulle äärimmäisen ohuelle palvelimelle on toteutusmalli, jossa toimintalogiikka eli MV*-mallin mallikerros on toteutettu palvelimelle, ja palvelin tarjoaa valmiit toimintalogiikan abstrahoivat metodit esimerkiksi REST-rajapinnan läpi. Tämä toteutus voi olla hyvä valinta, jos samaa palvelinta hyödyntää usea asiakas (esimerkiksi työpöytäsovellus ja web-sovellus), koska tällöin samaa logiikkaa ei tarvitse toteuttaa useaan asiakkaaseen. Lisäksi toimintalogiikan toteutus palvelimelle voi vähentää asiakkaan ja palvelimen välisiä kutsuja. Esimerkiksi usein käytetyssä tilisiirtoesimerkkitapauksessa palvelin voi tarjota valmiin rajapinnan, jolla voidaan suorittaa rahan siirto tililtä toiselle. (Siirrä summa X tililtä A tilille B.) Äärimmäisen ohut palvelin voisi samassa tilanteessa tarjota lähempänä tietokannan toimintaa olevat rajapinnat, joita käyttämällä yksi tilisiirto vaatisi useamman pyyntö-vastaus-parin. (Vähennä rahaa summa X tilitä A. Lisää rahaa summa X tilille B.) Lisäksi ongelmaksi muodostuisi transaktioiden käsittely (ks. luku 3.5).

3.4 HTML-sivupohjamoottori

Usein HTML-näkymän muodostamiseen ohjelmallisesti käytetään HTML-sivupohjamoottoria (engl. Web template engine). Sivupohja tarkoittaa sitä, että HTML-kielen sekaan voidaan kirjoittaa sivupohjakielen omia merkintöjä, joilla voi esimerkiksi merkitä muuttujan nimeä. Kun varsinainen HTML-sivu halutaan muodostaa, yhdistetään sovelluksen tarjoamat muuttujan arvot sekä sivupohja, johon nämä arvot voidaan sijoittaa. Sivupohjakielet sisältävät usein myös monimutkaisia rakenteita, kuten mahdollisuuden käydä listamuuttujan arvot läpi silmukkarakenteella. Tällöin voidaan esimerkiksi muodostaa listaesityksiä, joiden pituutta ei tarvitse ennalta määrätä. Lisäksi tyyppillinen sivupohjakieli mahdollistaa jonkin tyyppisen perinnän, jolloin voidaan määrittää yhdellä sivupohjalla usealle sivulle yhteiset asiat, kuten päävalikko. [7, s. 303]

Erilaisia sivupohjakielivaihtoehtoja on tarjolla lukuisia. Perinteisen ohut/paksuasiakas jaottelun mukaisesti HTML-sivun muodostus sivupohjan pohjalta voidaan suorittaa palvelinpäässä (kuten esimerkiksi Django-ohjelmistokehyksessä), mutta

nykyään single-page-ohjelmistokehyksien myötä on tullut myös valmiita ratkaisuja, jotka toimivat asiakaspäässä ja joita voidaan käyttää JavaScript-kielellä kirjoitetusta ohjelmasta.

3.5 Transaktiot single-page-sovelluksessa

Useimmiten transaktioita ajatellaan, kun ohjelmassa ollaan suoraan tekemisissä tietokannan kanssa. Web-sovelluksen tapauksessa tietokantatransaktioista huolehditaan ainoastaan palvelintoteutuksessa, eivätkä transaktiot, kuten ei myöskään koko tietokanta ole näkyvissä asiakaspäässä.

Single-page-sovelluksen toteutus voi kuitenkin olla sellainen, että tyypillisesti palvelimen suorittamat atomiset kokonaisuudet ovatkin sellaisia, että asiakaspää suorittaa atomisen kokonaisuuden osat, joiden tulisi kuulua samaan transaktioon.

Esimerkkinä voidaan käyttää monesta moneen -suhdetta käsitteiden välillä ja tapauksen mahdollisia toteutuksia REST-rajapintaan. Esimerkiksi palautteenkeräysjärjestelmässä voisi olla käytössä käsitteet tilaisuus ja osallistuja. Yhteen tilaisuuten voi liittyä monta osallistujaa ja yksi osallistuja voi osallistua moneen tilaisuuteen. Relaatiotietokantaan tällainen suhde mallinnettaisiin liittotaululla, joka kuvaa yhden osallistujan ja yhden tilaisuuden välistä liitosta, osallistumista.

Kun esimerkin tilanteesta luodaan REST-rajapinta, yksi mahdollisuus on luoda kolme resurssia: tilaisuus, osallistuja ja osallistuminen. Asiakaspään sovelluksessa voisi olla toiminto, jolla voidaan luoda tilaisuuksia, ja luonnin yhteydessä myös liittää niihin osallistujat.

Tässä kuitenkin nousee ongelmaksi se, että kun käyttöliittymässä käyttäjä suorittaa yhden tallennuksen luomalleen tilaisuudelle, pitää asiakassovelluksen kutsua kahta erillistä REST-rajapinnan resurssia tallentaakseen sekä tilaisuuden että erikseen sen osallistumistiedot. Lisäksi osallistumisresurssia voidaan joutua kutsumaan useita kertoja, jos osallistujia on useita. Transaktion tulisi olla usean REST-kutsun mittainen, sillä ei ole toivottua, että tietokantaan tallentuu onnistuneesti esimerkiksi luotu tilaisuus ja muutama osallistumistieto, jos taas joidenkin osallistumistietojen tallennus epäonnistuu.

Helppo ratkaisu on suunnitella REST-resurssit siten, että transaktiot eivät jakaudu useaan resurssiin. Esimerkin tapauksessa poistetaan osallistumisresurssi ja sisällytetään vastaava tieto tilaisuusresurssiin. Tällöin yksittäisen osallistumistiedon lisäys tarkoittaa tilaisuuden päivittämistä ja esimerkin tilaisuuden luominen osallistujineen onnistuu yhdellä HTTP-kutsulla. Voidaan myös erikseen ottaa käyttöön REST-resursseja, jotka paketoivat useamman muun resurssin tietoja yhteen transaktion kokoiseksi paketiksi. Esimerkkinä voisi olla "tilaisuus osallistujineen-resurssi, jos osallistujatietoa ei jostain syystä haluta sisällyttää varsinaiseen tilaisuusresurssiin.

Yllä kuvattu transaktio-ongelman välttäminen REST-resurssien suunnittelulla ei

riko REST-periaatteita. Harkitusti REST-rajapintaan voidaan myös lisätä RPC-tyyppisiä resursseja, jotka ovat yhden transaktion kokoisia, tai toteuttaa koko rajapinta RPC:nä REST:n asemesta. Tällöin substantiivi-resurssien sijaan käytössä voisi olla resurssi, johon sisältyy myös verbi. Klassisessa tilisiirtoesimerkin tapauksessa resurssi voisi olla "suorita tilisiirto", mutta tällainen yhdistettynä muuten REST-periaatteiden mukaisesti totetettuun rajapintaan voi olla hämmentävää; mitä tarkoittavat esimerkiksi GET tai DELETE -kutsu tähän resurssiin, joka jo itsessään sisältää verbin.

Jos rajapinta halutaan toteuttaa REST-periaatteiden mukaisesti, eikä resursseista saa muodostettua transaktion kokoisia, voidaan ottaa käyttöön ottaa myös erillinen transaktio-resurssi [20, s. 231].

3.6 Selainliitännäiset

Omanlaisiaan single-page-sovelluksia ovat myös esimerkiksi selaimessa käytettävät Flash- ja Java-sovellukset. Erona tässä työssä pääasiassa käsiteltäviin single-page-sovelluksiin näissä on se, että tällaiset single-page-sovellukset vaativat selaimeen erikseen asennettavan selainliitännäisen.

Adobe Flash -teknologialla toteutetut sovellukset vaativat ajoalustakseen joko selainliittännäisen (Flash Player) tai erillisen työpöytäsovelluksen. Yksi Flashin tarkoitettuja käyttökohteita on pelien tekeminen [24]. Flashin eduksi mainitaan sovellusten yhdenmukaisuus eri ajoalustoilla (toisin sanoen sama sovellus näyttää täsmälleen samalta esimerkiksi eri selainohjelmissa ajettuna) [24]. Flashin käyttö verkkosivuilla on ollut viimevuodet tasaisessa laskussa. Kesäkuussa 2014 noin 15% verkkosivuista käyttää Flash-teknologiaa. [45]

Java-sovelmia (Java applet) ajetaan selaimessa Java-selainliitännäisen avulla, joka luo yhteyden selaimen ja Java-ajoalustan (Java Runtime Environment, JRE) välille [35]. Vaikka Java on yksi suosituimpia ohjelmointikieliä [41][43] Javan käyttö selainsovellusten asiakaspään teknologiana on verrattain pientä; noin 0,1% verkkosivuista käyttää Javaa asiakaspäässä (tilanne kesäkuussa 2014) [46].

Yhteenvetona selainliitännäisten avulla toimivista single-page-sovelluksista voidaan todeta, että ne on ensisijaisesti tarkoitettu muunlaiseen kuin CRUD-tyyppisten sovellusten tekoon, esimerkiksi selaimessa toimivia pelejä varten. Lisäksi niiden käytön suosio on laskussa.

4. TOTEUTUSTEKNOLOGIAVAIHTOEHTOJA

Tässä luvussa käydään läpi single-page-sovelluksen toteutusteknologiavaihtoehtoja, jotka nojautuvat aiemmassa luvussa esiteltyihin perustekniikoihin. Lisäksi luvun lopussa esitellään palvelinsovelluskehys Django, jota ei kuitenkaan yritetä verrata luvun single-page-teknologioihin, koska Djangoa on tarkoitus käyttää valitun single-page-teknologian kanssa. Single-page-sovellus on tyypiltään sellainen, että erilaisia tähän kategoriaan kuuluvia yksittäisiä sovelluksia on toteutettu hyvin laaja joukko. Samanlaisia sovelluksia kehitettäessä törmätään usein samanlaiseen problematiikkaan. Web-sivujen tapauksessa hyvin suuri osa sovelluksista joutuu esimerkiksi toteuttamaan käyttäjän autentikoinnin, käyttäjän tilan tunnistuksen ja tallennuksen (usein evästeitä hyödyntämällä) sekä palvelintoteteutuksessa saman domainin eri osoitteisiin tulleiden HTTP-pyyntöjen reitityksen sovelluskoodin tiettyihin osiin. Yleisiin ohjelmistokehityksen ongelmiin, kuten edellä mainittuihin, on tarjolla ratkaisuja muun muassa sovelluskehysten, kirjastojen ja suunnittelumallien muodossa.

Sovelluskehykset ovat suosittuja web-sovelluksissa ja niitä on tarjolla lukuisia. Sovelluskehykselle ominaista on se, että se on valmista koodia, joka otetaan käyttöön ja jonka asettamiin reunaehtoihin oma sovellus toteutetaan. Sovelluskehys ottaa kantaa sovelluksen arkkitehtuuriin ja rajaa, mitä sovelluskehyksen puitteissa sovelluksessa voidaan tehdä. Hyvänä puolena sovelluskehys abstrahoi näkymättömiin monia sovellusalueen yleisiä ongelmia. Esimerkiksi web-sovelluskehys voi tarjota autentikointiin ja tilan tunnistukseen hyvin yksinkertaisen rajapinnan.

Ohjelmistokirjastolla tarkoitetaan valmista joukkoa luokkia tai funktioita, joita omaa sovellusta toteuttaessa voidaan hyödyntää toteuttamaan jokin yleinen asia. Kirjaston ja sovelluskehyksen ero on siinä, että sovelluskehys kutsuu käyttäjän toteuttamaa koodia, kun taas kirjastojen toteutuksia tulee kutsua itse [9, s. 40]. Sovelluskehys ottaa kirjastoa vahvemmin kantaa siihen, mitä ja miten sovellus voi tehdä.

Suunnittelumallit ovat sovelluskehyksiä ja kirjastoja abstraktimpia, sillä ne eivät tarjoa lainkaan valmista toteutusta. Suunnittelumalli antaa toteuttajalle sovelluskehystä vapaammat kädet toimia, eikä rajaa sovelluksen toimintaa sovelluskehyksen lailla. [9, s. 41] Toisaalta ohjelman toteutus suunnittelumallien pohjalta jättää paljon sovelluskehystä enemmän toteuttajan harteille. Yksittäinen suunnittelumalli koskee myös tyypillisesti pienempää yksityiskohtaa kuin sovelluskehys. Sovelluskehyksen toteutuksessa voidaan hyödyntää useita suunnittelumalleja.

Single-page-web-sovellusten toteuttamiseen on tarjolla laaja joukko avoimen lähdekoodin MVC-mallia tai sen variaatiota tukevia JavaScript-sovelluskehyksiä. Tässä luvussa esitellään ja verrataan toisiinsa joitain suosittuja sovelluskehysvaihtoehtoja. Yhtenä vaihtoehtona käsitellään myös Mikowskin ja Powelin [16] esittelemä arkkitehtuuri, joka ei nojaudu mihinkään valmiiseen MV*-sovelluskehykseen. Mikowskin ja Powelin esittelemä toteutustapa ei ole kirjasto tai sovelluskehys vaan parhaiten se voitaisiin määritellä (laajaksi) suunnittelumalliksi.

Toiveet, joita vasten eri vaihtoehtoja verrataan ovat seuraavat:

- Vaihtoehto soveltuu hyvin CRUD-tyyppisen sovelluksen luontiin, koska toteutettava sovellus on selkeästi CRUD-tyyppinen.
- Yhdellä kirjastolla tai sovelluskehyksellä saadaan katettua mahdollisimman paljon tarpeita, jotta erilaisten opeteltavien teknologioiden määrä pysyy minimissä ja toisekseen, jotta teknologioiden välisiltä yhteensopivuusongelmilta vältyttäisiin. Tunnistettuja tarpeita sovellukselle ovat ainakin HTML-näkymän luominen ja päivittäminen dynaamisesti halutulla tietosisällöllä sekä palvelinkommunikointi REST-tyyppisesti.
- Teknologian tulee olla tarpeeksi kypsä soveltuakseen tarvittaessa myös tuotantokäyttöön. Halutaan lisäksi, että käyttöä varten on saatavilla tukea.

Lueteltujen lisäksi sovelluksen tulee pystyä piirtämään (dynaaminen) graafi palvelimelta saatavan datan perusteella. Myös ulkoasussa halutaan hyödyntää valmista ulkoasukirjastoa, jotta ulkoasusta saadaan siisti ja käytettävä vähällä vaivalla. Single-page-kirjastoista ei voi olettaa löytyvän valmiina näitä ominaisuuksia, sillä ne ovat selvästi omia kokonaisuuksiaan. Valitun single-page-tekniikan tulee kuitenkin mahdollistaa tällaistan ominaisuuksien vaivaton käyttö.

Eräs laajasti käytetty ja suosittu ulkoasukirjasto on Bootstrap [28], joka koostuu sekä CSS- (Cascading Style Sheets, web-sivuille tarkoitettu ohjelmallisesti tulkittavaksi tarkoitettu tyyliohje) että JavaScript-tiedostoista. Graafin piirtämiseen kandidaatiksi valittiin Google Charts [32], joka on hyvin dokumentoitu ja jolla voi piirtää dynaamisia ja halutunlaisia graafeja. Nämä ovat teknologioita, joiden kanssa valittavan single-page-toteutusteknologian ensisijaisesti toivottaisiin toimivan hyvin, mutta valinnat eivät ole ehdottomia.

REST-kommunikoinnin vaatimus johtuu siitä, että järjestelmän ensimmäinen prototyyppiversio toteutettiin Django-sovelluskehystä käyttäen (ennen single-page-lähestymistavan päättämistä). Mahdollisuuksien mukaan haluttaisiin hyödyntää jo Djangolla toteutettua palautteenantonäkymää sekä tietomallia. Djangon on mahdollista kommunikoida asiakkaan kanssa REST-tyyppisesti, joka muutenkin on standardinomainen kommunikointitapa single-page-sovelluksissa asiakkaan ja palvelimen

välillä. On kuitenkin punnittava, olisiko mahdollisesti helpointa luopua olemassaolevista palvelintoteutuksista, jos niiden hyödyntäminen vaikuttaisi hankalalta.

4.1 Shell-malli

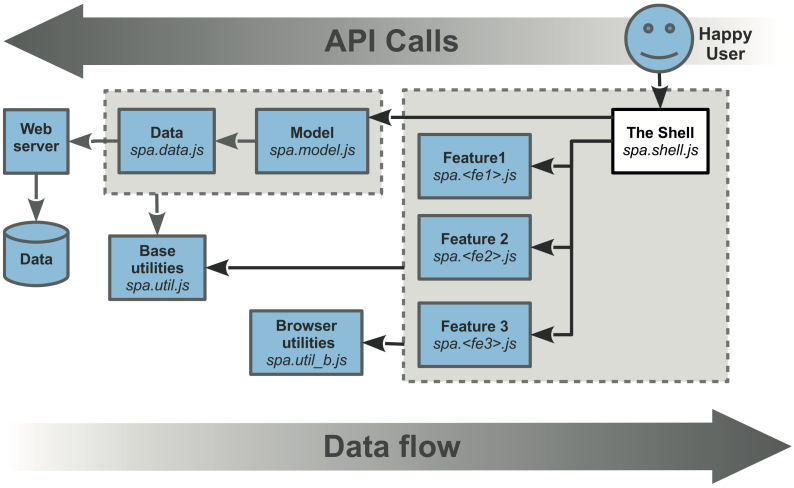
Mikowski ja Powell [16] esittelevät single-page-arkkitehtuurimallin, joka ei käytä hyväkseen valmiita ohjelmointikehyksiä. Oleellisin osa kirjan arkkitehtuuria on esitelty suunnittelumalli single-page-asiakassovelluksen toteutukseen JavaScriptillä. Esitellyssä kokonaisarkkitehtuurissa on lisäksi kuvattu palvelintoteutus sekä käytetty esimerkeissä tiettyä tietokantaa. Edellisten valinta on kirjassa tehty niin, että koko sovelluksessa ainoa käytettävä ohjelmointikieli on JavaScript. Tämän on mahdollistanut palvelimen toteutus Node.js:llä sekä MongoDB-tietokannan käyttö. Mikowski ja Powell perustelevat näitä valintoja sillä, että tämän ansiosta voidaan hyödyntää samaa koodia sekä asiakas että palvelintoteutuksessa, esimerkiksi kirjoitettuja apukirjastoja tietojen validointiin. Yhtenäinen ohjelmointikieli asiakas- ja palvelintoteutuksessa ei ole lainkaan itsestäänselvää web-sovelluksissa, sillä suosittuja palvelinkieliä ovat esimerkiksi PHP, Python ja Java. Asiakaspuolen toiminnallisuutta, jota ei HTML-CSS-yhdistelmällä pystytä tekemään, tehdään lähes poikkeuksetta JavaScriptillä.

Kirjan esittelemä selaintoteutus on tyypiltään koostesovellus (engl. composite application). Natiiveissa työpöytäsovelluksissa vastaavanlainen koostelähestymistapa on Microsoftin Prism-suunnittelumalli [40]. Koostesovelluksella tarkoitetaan sovellusta, joka koostuu useista löyhästi yhdistetyistä (engl. loosely coupled) käyttöliittymäkomponenteista. Tämä mahdollistaa käyttöliittymäkomponenttien helpon itsenäisen kehittämisen tai vaihtamisen toisiin. Koostesovelluksen erottaa muista käyttöliittymäkomponentteja hyödyntävistä menetelmistä se olennainen piirre, että komponentit kootaan pohjaansa ajoaikaisesti, eikä pohja tiedä, mitä komponentteja se sisältää [40].

Pohja, johon käyttöliittymän komponentteja kootaan on nimetty Mikowskin ja Powellin mallissa shelliksi (samaa nimeä käytetään myös Prism-suunnittelumallissa). Mikowski ja Powell eivät itse ole nimenneet esittelemäänsä arkkitehtuuria, mutta tässä työssä siitä käytetään selvyyden vuoksi nimitystä Shell-malli.

Shell-mallissa asiakaspään keskeisenä moduulina on shell, joka paitsi sisältää HTML-pohjan, johon muut käyttöliittymäkomponentit voidaan upottaa, myös vastaa MVC-mallin kontrolleria. Kuvassa 4.1 on esitelty arkkitehtuurin moduulijakoa. Shell ottaa vastaan käyttäjältä tulevat syötteet ja orkestroi muiden moduulien toimintaa tämän pohjalta.

Mikowski ja Powell mainitsevat, että heidän esittelemänsä arkkitehtuuri on fraktaali-MVC. Fraktaali-MVC on yleinen käsite, jolla tarkoitetaan, että sovelluksen arkkitehtuurissa voidaan tunnistaa MVC-malli usealta tasolta. Shell-mallin koko-



Kuva 4.1: Shell-mallin moduulijako. [16]

naisarkkitehtuuri toteuttaa MVC:n siten, että shell on kontrolleri. Toisaalta MVC voidaan tunnistaa myös pelkästä palvelintoteutuksesta tai yhdestä käyttöliittymäkomponentista.

Shell-mallin koostamat yksittäiset komponentit ovat toteutukseltaan samankaltaisia kuin webissä käytettävät kolmannen osapuolen liitännäiset, joissa omaan websivun HTML:ään lisätään elementti, johon liitännäinen voi itsenäisesti ladata ja sijoittaa itsensä kun selain lataa sivun. Esimerkiksi Facebookin tarjoamat sosiaaliset liitännäiset (engl. social plugins) ovat tällaisia [42]. Yksi esimerkki on Facebookin tykkää- ja jaa-painikkeet, joita voi lisätä helposti mille tahansa web-sivulle.

Hyviä puolia Shell-mallissa on hyvä laajennettavuus ja mahdollisuus jakaa kehitystyötä selvän moduulijaon ansiosta. Lisäksi Shell-malli on suunniteltu huomioimaan selaimen historiatoiminnot. Huonona puolena voidaan pitää sitä, että Shell-mallilla tehtyyn toteutukseen ei ole saatavilla verkosta samalla tavalla tukea ja ohjeita, kuin suosittuihin sovelluskehyksiin, vaan tukea saa lähinnä mallin esittelevästä kirjasta.

4.2 Knockout

Knockout on JavaScriptillä toteutettu MVVM-suunnittelumallin käyttöön ohjaava kirjasto [36]. Knockout määritellään omassa web-dokumentaatiossaan ohjelmointikirjastoksi, mutta siinä on myös paljon ohjelmistokehykselle ominaisia piirteitä, sillä Knockout suorittaa toimintoja taustalla ilman, että sitä eksplisiittisesti kutsutaan omasta sovelluskoodista.

Knockout tuo mahdollisuuden lisätä HTML-koodin sekaan omia, Knockout-spesifejä attribuutteja, joiden avulla HTML-kielellä määritetyn ulkoasun kenttiin voidaan luoda sidontoja JavaScriptillä määritettyihin olioihin, jotka edustavat MVVM-

mallin View Modelia. Knockout painottaa ominaisuutensa nimenomaan näkymään, View Modeliin ja näiden välisiin sidoksiin. Malliin, palvelintoimintaan tai asiakas-palvelin-yhteyteen Knockout ei tarjoa toiminnallisuutta, eikä kirjasto ota kantaa, miten nämä asiat tulisi toteuttaa.

Alla on annettu esimerkki Knockout-kirjaston käytöstä. HTML:ään voidaan sitoa View Modelissa oleva arvo seuraavasti.

```
<p>Palaute: <strong data-bind="text: palaute"></strong></p>
```

Yllä Knockoutin avainsanoja ovat attribuutti *data-bind* sekä sen arvona olevasta merkkijonosta osuus *text:*. Merkkijonon osuus *palaute* taas vastaa View Modelissa olevaa muuttujan nimeä, johon sidonta halutaan tehdä. Taustalla oleva View Model on kuvattu alla.

```
function PalauteViewModel() {
    this.teksti = ko.observable("Hyvää työtä!");
    this.aika = ko.observable("9.2.2014 9.30");

    this.palaute = ko.computed(function() {
        return this.aika() + " " + this.teksti();
    }, this);
}

// Aktivoi Knockout sidonnat
ko.applyBindings(new PalauteViewModel());
```

Realistisemmassa tilanteessa arvot eivät olisi kirjoitettu suoraan muuttujiin, vaan ne noudettaisiin jollain tavalla palvelimelta. Jos JavaScript-koodissa muutetaan myöhemmin esimerkiksi muuttujan *teksti* arvoa, päivittyy muutos tällöin automaattisesti myös HTML:n kohtiin, jotka on sidottu muuttujaan *palaute*, koska *palaute* on muuttujasta *teksti* johdettu arvo. Esimerkissä käytettävä muuttuja *ko* on Knockout-kirjaston määrittelemä luokka, jonka tarjoamilla palveluilla voidaan esimerkiksi luoda Knockoutin tarkkailtavia muuttujia ja määritellä olio View Modeliksi. Yllä olevien esimerkkien tapauksessa selain esittäisi prosessoinnin jälkeen tekstin "Palaute: Hyvää työtä! 9.2.2014 9.30".

4.3 Backbone.js

Backbone.js on JavaScript-kirjasto, joka on tarkoitettu käytettäväksi asiakaspään selainympäristössä. Backbone.js tarjoaa joukon työkaluja käytettäväksi, muun muassa mallien toteutukseen selaimessa ja mallien tietosisällön synkronointiin palvelimen kanssa. [19]

Backbone.js ei tarjoa sovelluskehyksille tyypillistä runkoa, jonka koukkuihin oma toteutus tulisi tehdä, ja se on siksi *kutsuttava sovelluskehys* (tai voitaisiin puhua myös kirjastosta). Tekijöidensä mukaan Backbone.js on pyritty pitämään sellaisena, että se rajoittaisi mahdollisimman vähän sitä, millä tavalla sen tarjoamia työkaluja voi käyttää [27].

Backbone.js:n oleellisin osa on sen tarjoamat moduulit

- View,
- Model,
- Event ja
- Collection,

jotka tarjoavat kirjastotyyppisiä toteutuksia tyypillisesti single-page-asiakaspään kehityksessä vastaan tuleviin tarpeisiin. Seuraavissa kappaleissa on tiivistetysti selostettu nämä tärkeimmät moduulit käyttäen lähteenä kirjaston omaa dokumentaatiota [27]. Selvyyden vuoksi Backbone.js:n termejä ei ole suomennettu, sillä osa suomennoksista olisi samoja kuin MV*-mallin osista käytetyt suomenkieliset termit.

Backbone.js tukee MV*-tyylistä arkkitehtuuria, jossa erillistä kontrolleria ei selvästi ole, vaan tyypillisiä MVC:n kontrollerin tehtäviä voidaan toteuttaa Backbone.js:n View-luokalla. Backbone.js:n View-luokka ei tarjoa korkeamman tason tai selainyhtenäistettyä toteutusta johonkin JavaScriptin toiminnon päälle, vaan se toimii moduulina, joka tarjoaa käytännön toteuttaa näkymään liittyvät asiat. View-luokat edustavat Backbone.js:ssä loogisia kokonaisuuksia, jotka halutaan kerralla renderöidä, kun jokin kyseisen näkymän tieto on muuttunut. Backbone.js:n View-tyyppi ei ole HTML:ää eikä se muutenkaan suoraan toteuta käyttäjälle näkyvää osuutta. Sen sijaan View tietää, mihin sivupohjan HTML-elementtiin sen tulee piirtää sisältönsä. Tärkeä osuus View-tyypissä on sen *render*-metodi, johon toteutetaan HTML:n sijoittaminen sivupohjaan. *Render*-metodi voidaan esimerkisi liittää tapahtumien avulla kutsuttavaksi, kun näkymän esittämä tieto muuttuu mallissa.

Model-tyyppi tarjoaa perustominnallisuuden, jonka avulla käsiteltävä tieto ja siihen liittyvä logiikka voidaan paketoida. Käsite on siis Backbone.js:ssä samanlainen kuin yleisesti MV*:ssä. Model tarjoaa valmiiksi metodit, joiden avulla asiakaspään Backbone-mallien tila voidaan synkronoida palvelimen kanssa. Oletuksena Backbone.js tukee palvelinkommunikoinnissa REST-tyyppisiä rajapintoja, joissa tieto kuljetetaan JSON-muodossa. Palvelinsynkronointiin liittyvät metodit voi myös ylikirjoittaa käyttämään muunlaista kommunikointoprotokollaa. Backbone.js:n Model-instanssien tilamuutokset eivät automaattisesti ja jatkuvasti päivity palvelimen kanssa, vaan vain erikseen synkronointia kutsuttaessa.

Backbone.js sisältää Events-moduulin, jonka tarjoamilla keinoilla voidaan lähettää tai kuunnella tapahtumia. Oletuksena Backbone.js:n Module-, Collection- tai Router-tyypistä luodut oliot laukaisevat tietyistä tilan muutoksistaan tapahtumia, joihin voidaan sitoa toiminnallisuutta. Kirjaston valmiiden tapahtumien lisäksi on myös mahdollista luoda omia tapahtumia ja laukaista ne halutussa kohdassa.

Backbone.js-Modelien instansseja voidaan koostaa Collectioneiksi. Collectioneiden avulla näkymät voivat tarkkailla kokonaista Model-instanssikokoelmaa usean yksittäisen Modelin sijaan. Esimerkiksi näkymä voi tarkkailla Collectionin sisällön muutoksesta laukeavaa tapahtumaa. Collectionit myös mahdollistavat sellaisten funktioiden toteutuksen, jotka vaikuttavat useaan Modeliin. Collection ei ole pelkästään tapahtumia lisäävä kerros minkään tietyn JavaScriptin natiivin tietorakennetyypin päälle. Collection on tietorakennetyypiltään järjestetty *set*, eli se voi sisältää saman olion vain kerran. Collection tarjoaa laajan joukon rajapintametodeja tietorakenteen käsittelyyn, esimerkiksi *push*, *pop*, *at* ja *sortBy*.

Router-moduulin avulla voidaan hallita selaimen URL:n ja single-page-sovelluksen tilan välistä yhteyttä. Sen avulla voidaan reitittää tietynlaiset URL:t tiettyihin funktiokutsuihin. Backbone.js:n Router on korkeamman tason toteutus HTML5:n historia API:n päälle. Router voi tarvittaessa käyttää myös ankkurimetodia osoitteiden esittämiseen, jos selain ei tue historia API:a.

Seuraavassa on esitetty esimerkki Backbone.js:n käytöstä. Esimerkissä luodaan oma Model- sekä Collection-tyyppi, ja näytetään kuinka tapahtumia voidaan käyttää. Esimerkkikoodin toiminta on selostettu kommentein koodin seassa. Koodissa *Backbone*-muuttuja on kirjaston tarjoama olio, jonka avulla Backbone.js:n toiminnallisuuksia otetaan käyttöön.

```
// Uusi Backbone Model-tyyppi
var Feedback = Backbone.Model.extend({
  toString: function() {
    return this.get("text");
  }
});

// Uusi Backbone Collection-tyyppi
var Feedbacks = Backbone.Collection.extend({
  model: Feedback
});

// Instanssi Collection-tyypistä
var allFeedbacks = new Feedbacks;
```

```
// Sidotaan tapahtumakäsittelijä
// Collection-instanssin add-tapahtumaan.
allFeedbacks.on("add", function(feedback) {
  alert("Uusi palaute: " + feedback.toString());
});

// Feedback-instanssi, jolle asetetaan attribuutti.
var fb1 = new Feedback;
fb1.set({"text" : "Hyvää työtä!"});

// Lisätään Collection-instanssiin Feedback-instanssi.
allFeedbacks.add(fb1);
// Prompt: "Uusi palaute: Hyvää työtä!"

// Sama tapahtuma voidaan laukaista myös manuaalisesti.
allFeedbacks.trigger("add", fb1);
// Prompt: "Uusi palaute: Hyvää työtä!"
```

4.4 Angular.js

Angular.js on Googlen ylläpitämä avoimen lähdekoodin JavaScript-sovelluskehys CRUD-tyyppisten single-page-sovellusten asiakaspään kehitykseen. Angular.js-sovelluskehystä käytettäessä HTML:ään lisätään erilaisia sovelluskehyksen omia merkintöjä, jotka ohjaavat kehyksen vuorovaikutusta HTML:n kanssa. Angular.js:n perusfilosofiaksi mainitaan se, että ulkoasu on helpointa määritellä deklaratiivisella kielellä, kun taas toimintalogiikka on helppoa määrittää imperatiivisella kielellä. Tässä tapauksessa deklaratiivinen kieli on HTML (ja CSS) ja imperatiivinen kieli JavaScript. [25] Tämä on samantyylinen osa-alueiden jako kuin esimerkiksi Microsoftin WPF:ssä, jossa vastaavina kielinä toimivat XAML ja C# (tai muu .NET-kieli). Sama HTML ja JavaScript -jako on luontaisesti muissakin Single-page-sovelluskehyksissä, vaikkei sitä eksplisiittisesti usein mainitakaan.

Angular.js:n suunnittelumalliksi mainittu MVW (katso luku 2, sivu 14) viittaa, että Angular.js:ää ei ole haluttu tarkasti määrittää toteuttamaan jotain tiettyä MV*-mallia. Se osa, joka Angular.js-sovelluksessa ei ole mallia tai näkymää, vaan jollain tavalla edesauttaa näiden yhteen sovittamista, on moduulit (engl. module) ja niissä olevat komponentit. Yksi mahdollinen komponenttityyppi Angular.js:ssä on nimeltään controller. Triviaalissa sovelluksessa controller-komponentit voivat sisältää kaiken koodin, joka ei kuulu malliin tai näkymään, mutta Angular.js:n suositeltu käyttötapa on jakaa tätä toteutusta myös muiden kuin controller-komponenttien

vastuulle. Angular.js:n controller ei siis täysin vastaa MVC-mallin kontrolleria, vaan Angular.js sisältää tapoja, joilla kontrolleri-tyyppisen kerroksen toiminnallisuutta voidaan jäsennellä (ja yksi mahdollinen jäsennyksen työkalu on controller), eikä tätä kokonaisuutta ole haluttu nimetä juuri MVC-mallin kontolleriksi.

Angular.js-sovelluskehys sisältää oman HTML-sivupohjamoottorinsa, joka perustuu kehykseen valmiiksi kuuluvien tai itse kirjoitettavien direktiivien (engl. directives) käyttöön. Direktiivejä voivat olla esimerkiksi HTML-attribuutit tai elementit. Yksi esimerkki sovelluskehyksen tarjoamasta direktiivistä on *ngBind*, jonka avulla näkymässä voidaan määrittää datan sidonta.

Datan sidonta tehdään Angular.js:ssä kehyksen erityiseen *$scope*-olioon, jota voidaan pitää kehyksen implisiittisenä View Modelina (samassa merkityksessä, kuin MVVM-mallin View Model, josta on kerrottu luvussa 2, sivulla 12). Käytännössä implisiittisyys tarkoittaa esimerkiksi sitä, että näkymässä voidaan sitoa arvoja toisiin saman näkymän arvoihin (käyttämällä niille samaa muuttujan nimeä) ilman, että tätä varten tarvitsee kirjoittaa muuta kuin deklaratiivista näkymäkoodia. Kehys luo näkymäkoodin perusteella kyseisen näkymän käyttämään *$scope*-olioon oman muuttujan jokaiselle näkymässä käytetylle muuttujalle. Jokainen saman niminen näkymän muuttuja sidotaan automaattisesti kaksisuuntaisesti tähän samaan *$scope*-olioon. Oikeissa sovelluksissa edellä mainitun kaltainen tarve on harvinainen, mutta Angular.js:n controller-komponentissa voidaan myös käsitellä samaa *$scope*-oliota, jota näkymäkin käsittelee. *$scope*-olio on täysin kehyksen sisäistä toimintaa, ja se on yksilöllinen jokaiselle sovelluksessa ajonaikaisesti luotavalle controller-näkymä-parille. *$scope*-olion johdosta näkymän ei tarvitse tietää mihin controller-komponenttiin se liittyy eikä controller-komponentti tiedä näkymästä, joka esittää sen käsittelemiä tietoja. Implisiittinen View Model on yksi hyvä syy, miksi Angular.js:n toteuttamaa suunnittelumallia ei voida selvästi sanoa MVC:ksi, eikä myöskään MVVM:ksi.

Angular.js-sovellus koostuu moduuleista. Moduuli on Angular.js:n oma termi, jolla tarkoitetaan sovelluksen osan paketoivaa kokonaisuutta. Angular.js:n suositeltu moduulijako on oma moduuli sovelluksen eri toiminnoille ja yksi moduuli, joka koostaa varsinaisen sovelluskokonaisuuden muista moduuleista. Yhdessä moduulissa voi olla mielivaltainen kokoelma Angular.js-komponentteja, joita ovat esimerkiksi seuraavat: controller, directive, filter ja service (ohjelmistokehyksen omia termejä ei yritetä tässä suomentaa selvyyden vuoksi). Suositeltava tapa on tehdä sovellukseen esimerkiksi yksi moduuli, joka sisältää kaikki sovellukseen tehtävät controller-komponentit, toinen moduuli joka sisältää kaikki directive-komponentit ja niin edelleen. Moduuli voi olla sovelluskehittäjän itse tekemä, kehyksen valmiina tarjoama tai kolmannen osapuolen tekemä ja jakama. [48]

Angular.js-kehyksessä hyödynnetään dependency injection -suunnittelumallia. De-

pendency injection on tapa sitoa komponentteja löyhästi toisiinsa, niin että ne ovat esimerkiksi automaattisia testejä varten helposti vaihdettavissa mock-olioihin [6]. Esimerkiksi jokin komponentti voisi käyttää HTTP-kommunikoinnin toteutuksen tarjoavaa toista komponenttia. Jos HTTP-komponentti otetaan käyttöön dependency injection -mallin mukaisesti, se voidaan yksikkötesteissä helposti korvata saman rajapinnan toteuttavalla testikomponentilla, joka ei oikeasti suorita HTTP-kutsuja. Angular.js:ssä dependency injection -mallia käytetään hyväksi, kun komponentissa halutaan hyödyntää toista komponenttia. Esimerkiksi, kun controller-komponenttissa halutaan käyttää service-komponenttia.

Angular.js:n *ngRoute*-moduulin *route*-servicen (route-niminen service-tyyppinen komponentti) avulla yhdistetään URL:ja näkymä-controller-pareihin. *Route*-palvelu käsittelee single-page-sovelluksen sisäisiä tiloja URL:n ankkurielementin avulla. (Valmista tukea HTML5:n historia API:lle ei kirjoitushetkellä ole.) Jos sovelluksessa halutaan siirtyä tilasta toiseen, niin tämä toteutetaan Angular.js:ssä hyperlinkeillä, jotka muuttavat ankkurielementtiä. Tällöin kaikki tilan muutokset menevät *route*-palvelun kautta riippumatta siitä, tullaanko tilaan suoraan osoiteriviä muuttamalla, sivun sisäisestä linkistä, tai selaimen kirjanmerkin kautta. Angular.js-sovelluksen URL:t noudattavat tyypillisesti seuraavan esimerkin mukaista muotoa: www.nettipalaute.fi#/yllapito/palaute/3, eli ankkurielementti muodostetaan hierarkisesti /-merkkejä käyttäen samaan tapaan kuin normaalikin URL, jossa ei ole ankkuriosuutta.

Angular.js:n servicet ovat yksi tapa jakaa toiminnallisuutta omiin komponentteihinsa moduulin sisällä. Angular.js tarjoaa valmiina servicejä, kuten *$http* ja *$location*, jotka abstrahoivat AJAX-kyselyiden tekemistä HTTP:n yli ja vuorovaikutuksen selaimen esittämän URL:n kanssa. Omia servicejä voi tehdä esimerkiksi MV*-jaon malli-osalle. Servicet ovat singleton-suunnittelumallin toteuttavia luokkia, joista Angular.js-kehys takaa, että käytössä on vain yksi instanssi (joskin tarvittaessa myös useammat yksilölliset instanssit ovat mahdollisia, mutta normaalisti tämä ei ole toivottua service-komponenttien luonteen takia). Servicet vastaavat käsitteenä jossain määrin normaaleja olio-ohjelmoinnin luokkia. Luokasta poiketen Angular.js tarjoaa serviceille yksinkertaisen tavan toteuttaa dependecy injectionin, joka vähentää servicejen välisten riippuvuuksien eksplisiittisen merkinnän tarvetta. Muita, Angular.js:n servicen kaltaisia komponenttityyppejä ovat muun muassa provider ja factory, mutta näiden välisiä eroja ei käsitellä tässä työssä, sillä toiminta ja käyttö sillä kuvauksen tasolla, jota nyt käytetään, ovat samat.

Testausominaisuudet ovat vahvasti integroituna Angular.js:ään. Kehys tarjoaa valmiit työkalut sekä yksikkö, että tätä laajempaan testaukseen. Testattavia yksiköitä Angular.js-sovelluksessa ovat esimerkiksi service- ja controller-komponentit. Yksikkötestattavat yksiköt tulisi kehitettäessä eristää DOM-manipulaatioista ja an-

taa kehyksen implisiittisen View Modelin (*$scope*-olio) tehdä arvojen yhdistäminen näkymään. Angular.js:n dependency injectionin avulla on myös mahdollista tehdä yksikkötestikoodissa yksinkertaisia mock-olioita tai palveluita, jos testattavalla yksiköllä on riippuvuuksia, jotka muuten monimutkaistaisivat testausta. Esimerkiksi palvelu voi noutaa normaalisti dataa palvelimelta *$http*-palvelun avulla, mutta yksikkötestausta varten testattavalle palvelulle voidaan antaa oikean HTTP-palvelun asemesta yksinkertainen mock-palvelu, joka palauttaa aina saman kovakoodatun datan.

Yksikkötestit Angular.js:ssä ovat syntaksiltaan Jasmine-testauskehyksen [34] mukaisia. Angular.js-tiimi on toteuttanut Karma-nimisen testityökalun, joka ajaa esimerkiksi yksikkötestejä aina kun testattavaa koodia sisältäviä tiedostoja muokataan, ja antaa näin välitöntä palautetta tehdyistä muutoksista. Karma ei varsinaisesti ole osa Angular.js-kehystä, mutta näiden yhteiskäyttö on tehty sujuvaksi.

Laajat integraatiotestit (end-to-end-testit) ovat web-sivuille normaalisti paljon yksikkötestejä pitempiä ajoajaltaan, koska ne sisältävät selaimessa tapahtuvaa DOM-käsittelyä. Integraatiotestejä ei tämän takia voida ajaa automaattisesti Karman avulla jokaisen tiedostomuokkauksen jälkeen kuten yksikkötestejä. Integraatiotestit Angular.js:ssä kirjoitetaan myös Jasmine-tyyppisellä syntaksilla ja ne voidaan ajaa manuaalisesti Karma-työkalun avulla.

4.5 Single-page-toteutustekniikoiden vertailu

Edellä esiteltyjen toteutustekniikoiden vertailu ei ole helppoa, sillä ne eivät pyri ratkaisemaan samaa ongelmaa, eivätkä ne ole välttämättä täysin toisensa poissulkevia. Karkealla tasolla voidaan eritellä, että shell-malli tarjoaa laajan yleisohjeen sovelluksen toteuttamisesta. Shell-arkkitehtuuri on suunnittelumalli ja kokoelma pienempiä suunnittelumalleja, joita voi myös hyödyntää, vaikka toteutuksessa käyttäisi valmista kirjastoa tai kehystä.

Knockout on esitellyistä vaihtoehdoista suppein keskittyen MVVM-mallin mukaisen toteutuksen tukemiseen nimenomaan View Modelin osalta, ja tarjoten mahdollisuuden helppoon muuttujien sitomiseen View Modelista näkymään.

Backbone.js on Knockoutia hieman selvemmin kirjasto, joka tarjoaa myös sidontamahdollisuuden. Sidonnan lisäksi Backbone.js:ssä on moduuleita, joita voi käyttää reititykseen ja omiin tapahtumiin sovelluksen tarpeiden mukaisesti. Backbone.js tarjoaa vaihtoehdoista selvimmin vain joukon yksittäisiä työkaluja, joita voi käyttää hyödyksi monenlaisissa sovellusrakenteissa.

Angular.js on laajin vertailluista ohjelmistokehyksistä ja -kirjastoista. Hyvänä puolena tämä tarjoaa paljon valmista runkoa, jonka ympärille oman sovelluksen voi kehittää. Väistämättä huonona puolena tästä seuraa se, että Angular.js soveltuu hyvin vain tietyn tyyppisiin sovelluksiin, mikä tässä nimenomaisessa tapauksessa

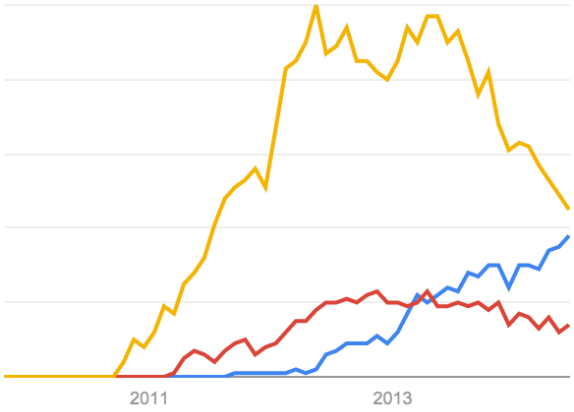
Ominaisuus	Shell-malli	Knockout	Backbone.js	Angular.js
Soveltuu CRUD-sovellukseen	**	**	**	**
Sama kirjasto tai kehys kattaa paljon tarpeita	*	*	*	**
Kypsä teknologia, dokumentaatio ja tukea saatavilla		**	**	**
Bootstrap ja Google Charts käyttö teknologian kanssa	*	*	*	**

Taulukko 4.1: Toteutusteknologiavaihtoehtojen vertailu ominaisuustoiveita vasten. Tyhjä solu on huonoin vastaavuus ja kaksi tähteä paras.

ei ole huono puoli, sillä Angular.js on tarkoitettu nimenomaan CRUD-tyyppisille sovelluksille.

Vertailusta on tehty yhteenveto taulukkoon 4.1. Taulukossa on vertailtu teknologioiden sopivuutta ominaisuustoiveisiin. Sopivuus on merkitty tähdillä siten, että nolla tähteä on huonoin sopivuus ja kaksi tähteä hyvä sopivuus. Kaikki teknologiat soveltuvat CRUD-sovelluksen toteutukseen hyvin. Angular.js kattaa eniten tarpeita tarjoten niihin valmista toteutusta tai runkoa. Myös Shell-malli ottaa kantaa laajaan joukkoon tarpeita, mutta suunnittelumallina se ei tarjoa varsinaista valmista toteutusta. Shell-mallin tuki rajoittuu kirjaan, jossa se esitellään. Muista teknologioista löytyy internetistä hyvä dokumentaatio ja paljon keskustelua sekä esimerkkejä. Bootstrap ja Google Charts -teknologioiden käyttöön teknologian kanssa ei ole näkyvissä esteitä minkään vertailun teknologian kanssa, mutta Angular.js:ään on tarjolla valmiita lisäosia molempien käyttöön. ("AngularJs Google Chart Tools directive"lisäosa Google Chart -taulukoiden käyttöön Angular.js directives-merkinnän avulla [26]. Bootstrapin ja Angular.js:n yhteiskäyttöä helpottamaan on Angular.js tiimin itse toteuttama lisäosa "UI Bootstrap"[44].) Vertailussa Angular.js menestyy parhaiten.

Google Trends on Googlen tarjoama palvelu, jolla hakutermeillä suoritettujen hakukonehakujen suhteellista määrää voidaan tarkastella ajan suhteen [33]. Palvelun avulla voidaan vertailla, kuinka suosittuja tarkasteltavat single-page-teknologiat ovat. Shell-mallia ei voida ottaa mukaan vertailuun, koska sille ei ole selkeää hakutermiä, joka ei sekottuisi hakuihin, jotka on tehty täysin muussa tarkoituksessa. 28.6.2014 suoritetun vertailun tulokset ovat nähtävissä kuvassa 4.2. Voidaan päätellä, että teknologiat ovat keskenään suurin piirtein yhtä suosittuja, mutta Angular.js on ainoa, jonka suosio on kasvussa, kun taas Knockoutin ja Backbone.js:n suosittuus on kääntynyt laskuun. Täytyy kuitenkin muistaa, että tehty vertailu on vain suuntaa antava, sillä Google-hakujen määrä ei kerro koko totuutta teknologioiden



Kuva 4.2: Google Trends vertailu "Angular.js"(sininen), "Backbone.js"(keltainen) ja "Knockout.js"(punainen) hakutermien suhteellisista Google-hakumääristä.

käytöstä. Lisäksi vertailtavat termit eivät todennäköisesti sisällä kaikkia teknologiaan liittyen tehtyjä Google-hakuja. Kaikkiin hakutermeihin sisällytettiin .js-pääte[1], vaikka tämän päätteen käyttö ei välttämättä ole yhtä tavallista kaikkien teknologioiden kanssa, koska muuten suurin osa termin sisältävistä hauista olisi jotain muuta kuin tässä tarkoitettuun teknologiaan liittyvää (angular, backbone ja knockout ovat englannin kielisiä normaaleja sanoja). Myös tämä vertailu tukee Angular.js:n valintaa.

4.6 Django

Django on Python-kielinen web-sovelluskehys [29]. Django käyttää MVT-suunnittelumallia, ja se on tarkoitettu pääasiassa palvelinpainotteisten verkkosivujen tekoon.

Djangon MVT-malli koostuu seuraavista osista:

- Malli (Model), joka Djangossa koostuu sovelluksen toteuttajan tekemistä Python-luokista, jotka periytetään Djangon tarjoamasta kantaluokasta *Model*.
- Näkymä (View), joka poikkeaa selvästi normaalin MV*-mallin näkymästä. Djangon yksittäinen näkymä on metodi, joka on kuvattu tiettyyn web-sovelluksen URL:aan ja joka suoritetaan URL:a kutsuttaessa. Näkymämetodi käsittelee tyypillisesti mallikerroksen luokkia, ja noutaa niiden avulla tilanteessa tarvittavat tiedot, ja tarjoaa ne eteenpäin HTML-sivupohjamoottorille HTML-sivun muodostukseen (josta Djangon näkymäkerros siis ei enää vastaa).

[1].js-pääte on JavaScript-tiedostojen yleinen tiedostopääte (lyhenne JavaScript-termistä). On muodustunut tavaksi käyttää tätä päätettä JavaScript-kirjastojen ja -kehysten nimissä, vaikkakin esimerkiksi Knockoutin internetsivulla .js-päätettä nimessä ei käytetä.

- Template, joka vastaa melko läheisesti tyypillisen MV*-mallin näkymää. Djangon template-kerros koostuu HTML-sivupohjista, joihin sijoitetaan näkymäkerroksen sivupohjalle tarjoamat tiedot. Sivupohjasta muodostettu HTML-näkymä palautetaan kutsuneelle asiakkaalle.

Django tarjoaa ORM:n (Object-relational mapping, olio-relaatiokuvaus) mallikerroksen luokille, eli luokkia määritellessä Djangossa käytännössä määritellään samalla tietokantamallin rakenne. Malliluokkiin voi lisätä erilaisia määritteitä, joilla voi halutessa ohjata tietokantamallin generointia, jos automaattitoteutus esimerkiksi monesta moneen -suhteille ei ole tyydyttävä tai jos tietokannan taulujen ja kenttien nimissä halutaan noudattaa muuta periaatetta kuin Djangon automaattisesti luomaa. Malliluokkien jäsenfunktioiksi on myös tarkoitus toteuttaa toimintalogiikka, jos sovellus sisältää muuta logiikkaa, kuin yksinkertaista olioiden luomista, muokkaamista ja poistoa. Yksinkertaisessa CRUD-sovelluksessa Djangon malliluokkiin ei välttämättä ole tarvetta toteuttaa lainkaan jäsenfunktioita, sillä malliluokkien yhteinen kantaluokka tarjoaa geneeriset toteutukset olioiden luontiin ja muokkaukseen. Lisäksi tiedon noutoa varten Django tarjoaa SQL-kyselyt abstrahoivan kerroksen, joilla tietokantaan tallennettuja malliluokkien olioiden tiloja voidaan hakea sovelluksen käytettäväksi. Etuna tietokantatoteutuksen abstrahoinnissa on vapaus valita useasta eri tietokannasta ilman, että tämä vaikuttaa lainkaan sovelluskoodiin. Esimerkiksi voi olla kätevää pitää kehitysympäristössä tietokantana yksinkertaista SQLite-tietokantaa ja tuotantoympäristössä esimerkiksi PostgreSQL-tietokantaa.

REST-rajapintojen toteutusta helpottamaan Djangoon on tehty lisäosa Django REST framework [31], joka tarjoaa valmiit toteutukset yleisiin REST-rajapintoihin palvelinpäässä tarvittaviin toimintoihin, esimerkiksi Djangon mallikerrosten luokkien sarjallistaminen ja purkaminen.

5. PALAUTTEENKERÄYSJÄRJESTELMÄN TOTEUTUS

Tässä luvussa selostetaan toteutetun järjestelmän toteutusratkaisut tekniseltä kannalta, sekä käyttöliittymä siltä osin, kuin sen toiminta on oleellista tietää teknisten ratkaisujen kontekstina. Tehtyä, tässä luvussa kuvattua toteutusta ja valittuja ratkaisuja kommentoidaan luvussa 6.2. Toteutettu järjestelmä on saatavissa osoitteessa http://nettipalaute.fi.

5.1 Toteutettava järjestelmä

Tässä luvussa on kuvattu toteutettavan järjestelmän toiminnot ja käyttöliittymä. Järjestelmän toiminta-ajatus yleisemmällä tasolla on kuvattu luvussa 1.1. Toteutus tekniseltä kannalta on puolestaan kuvattu luvussa 5.2.

Käyttäjien kannalta toteutettava järjestelmä koostuu kahdesta osasta: palautteen antajan näkymästä sekä palautteen kerääjän hallintasivusta. Lisäksi web-sivulla on yhteinen aloitussivu, yleistä tietoa järjestelmästä ja käyttöohjeet eri käyttäjille.

Järjestelmän toiminta ja käyttö etenee niin, että

- palautteen kerääjä luo itselleen käyttäjätunnuksen järjestelmään ja kirjautuu sisään,
- palautteen kerääjä luo järjestelmään session ja liittää siihen haluamansa palautetyypit,
- palautteen kerääjä antaa luomansa session tunnuksen palautteen antajille tilaisuuden alussa,
- palautteen antajat kirjautuvat sessioon tilaisuuden alussa ja antavat palautetta tilaisuuden kuluessa.
- Palautteen kerääjä voi tilaisuuden aikana tai sen jälkeen tarkastella saatuja palautteita, jotka esitetään graafissa ajan suhteen.

Tarkemmin järjestelmän käyttö on kuvattu liitteenä olevassa käyttöohjeessa (A), jossa on myös kuvat käyttöliittymän oleellisista osista, joista osasta tässä luvussa jäljempänä puhutaan.

Tämän työn kannalta oleellinen osa koko toteutettavaa järjestelmää on palautteen kerääjille suunnattu ylläpitonäkymä, jossa voidaan luoda uusia sessioita, muokata vanhoja sekä tarkastella saatuja palautteita. Vain tämän osuuden toteutuksessa hyödynnetään single-page-tekniikoita.

Ylläpitonäkymän pääsivulla esitetään aina graafina palautteet siihen sessioon, johon on viimeksi annettu palautetta. Lisäksi näkyvistä valikoista voi valita tarkasteltavaksi jonkin toisen sessiotyypin ja siihen liittyvän session. Lisäksi päänäkymässä on nappien takana toiminnot sessiotyyppien luomiseen ja muokkaamiseen. Toteutetun järjestelmän nykyisessä versiossa toteutettiin sessiotyyppiin liittyvien sessioiden luonti implisiittisesti siten, että sessio on aina vuorokauden mittainen ja sessio luodaan sessiotyypille jokaista vuorokautta kohti, jona on annettu yksi tai useampi palaute liittyen sessiotyyppiin.

Pääsivun lisäksi ylläpitosivulla merkittävänä asiana on dialogi-tyyppisenä toteutettu näkymä, jossa voi sekä luoda että muokata sessiotyyppejä. Dialogi sisältää sessiotunnuksen annon sekä palautetyyppien luomisen ja liittämisen sessiotyyppiin. Palautetyypit voi luoda joko sessiotyyppikohtaisiksi tai yleiskäyttöisiksi. Yleiskäyttöisenä luodut palautetyypit ovat valmiina valittavissa kaikkiin sessiotyyppeihin, kun ne ensin on yhden sessiotyypin yhteydessä luotu järjestelmään. Järjestelmän nykyisen version tekstipalautteilla yleiskäyttöisten palautetyyppien luonti voi vaikuttaa turhalta ominaisuudelta. Tarkoitus on ollut varautua mahdollisuuteen, että palautetyypit ovat monimutkaisempia, kuin vain lyhyt teksti. Lisäksi nykyisellään yleiset palautetyypit ja niiden käyttö antaa mahdollisuuden luoda tulevaisuudessa järjestelmässä sessiotyyppirajat ylittäviä tilastoja palautteiden annosta.

Palautteen antajalle sessiokäsite ei näy mitenkään, vaan palautteen antaja liittyy aina sessiotyyppiin sessiotyypin sessiotunnuksella. Järjestelmä liittää annetut palautteet automaattisesti oikeaan sessioon. Näin palautteen antajan tarvitsee vain tietää, että hän on esimerkiksi *Ohjelmointikurssi 1*:n luennolla, mutta varsinaisella luentokerralla ei ole merkitystä.

Toteutettu sessiotyyppi- ja sessiojako olettaa, että esimerkiksi saman kurssin kaikilla luentokerroilla halutaan kerätä palautetta samoilla palautetyypeillä. Jos näin ei ole, niin eri luentokerroille tulee luoda omat sessiotyypit. Etuna valitussa toteutuksessa on, että jos palautetta halutaan kerätä esimerkiksi usealla saman kurssin luennolla samoilla palautetyypeillä, riittää, että sessiotyyppi luodaan kerran. Tämän jälkeen sessiot luodaan automaattisesti aina, kun palautetta annetaan, eli ensimmäisen kerran jälkeen ylläpitäjällä ei ole mitään lisätyötä ja palautteen antajat voivat antaa palautetta joka kerta saman sessiotunnuksen kautta.

5.2 Valitut toteutustekniikat ja ratkaisut

Tässä luvussa esitellään toteutuksessa käytettäväksi valitut tekniikat sekä arkkitehtuuriset ratkaisut. Osa käytettäväksi valituista avainteknologioista on jo esitelty luvussa 4. Tässä luvussa kuvataan korkealla tasolla kokonaisjärjestelmän toiminta. Single-page-toteutus on kuvattu tarkemmin luvussa 5.3.

Työn aiheen asettama vaatimus käytettäville tekniikoille oli single-page-toteutustekniikan hyödyntäminen, ja tätä varten asiakaspään teknologiana päädyttiin käyttämään Angular.js-ohjelmistokehystä. Toteutettavan järjestelmän ylläpitosivu on tyypillinen CRUD-sivu, jolloin Angular.js tarjoaa laaja-alaisesti sopivat raamit, eikä ohjelmistokehystä ole tarvetta kiertää tai käyttää vastoin tarkoitusta. Muut vertaillut tekniikat ja tavat olivat Angular.js:ää löyhempiä, joten Angular.js:n katsottiin tarjoavan suurimman hyödyn, kun toteutettava järjestelmä oli sille sopiva. Lisäksi Angular.js:n dokumentaatio oli hyvä ja ohjelmistokehyksen suosio oli valintahetkellä kasvava, joten voidaan olettaa, että kehyksen ylläpito ja kehitys ei ole loppumassa lähitulevaisuudessa.

Palvelintoteutus tarjoaa kaikki ylläpitonäkymän tarvitsemat pysyvät resurssit REST-rajapinnan kautta. Palvelimen toteutustekniikoiksi valikoitui Django-sovelluskehys sekä siihen saatava Django REST framework -niminen laajennos, joka auttaa REST-rajapintojen toteutuksessa. Suurin syy Django-sovelluskehyksen valintaan oli se, että se oli ennalta tuttu.

Asiakas-palvelin-kommunikoinnin arkkitehtuuriksi valittiin REST-rajapinta, koska REST-rajapinta sopii CRUD-sovellukselle (ks. luku 3.2), ja se on tyypillinen valinta single-page-sivun palvelinkommunikointiin. Lisäksi Angular.js tarjoaa valmiit puitteet REST-tyyppiseen kommunikointiin, joten sekä asiakas- että palvelintoteutuksessa oli valitulle kommunikaatiotavalle tukea.

Autentikaation ja valtuutuksen suhteen päädyttiin toteutusratkaisuun, jossa single-page-asiakaspää ei vastaa autentikaatiosta millään tavalla, vaan käyttäjä autentikoidaan ennen varsinaiselle single-page-sivulle siirtymistä. Single-page-osuus saa valmiina valtuutuksen, jonka se asettaa mukaan jokaiseen tekemäänsä palvelinkyselyyn. Valtuutus kulkee evästeenä. Näin single-page-toteutus voidaan toteuttaa niin, että sen toteutuksessa ei oteta lainkaan kantaa käyttäjän tunnistukseen tai siihen, että palvelimelta tietoja kysellessä tietokannassa on oikeasti monen käyttäjän tietoja, vaikka REST-kyselyt kuvautuvat paikoin hyvin suoraan tietokantakyselyihin. Palvelin rajaa asiakkaalle palautettavat tiedot asiakkaan mukaan. Esimerkiksi, kun asiakas pyytää palauttamaan kaikki palautteet, palvelin tietää rajata kyselyn oikealla käyttäjällä. Myös REST-rajapintaan voitaisiin toteuttaa rajausmahdollisuus käyttäjän mukaan, mutta koska ei ole nähtävissä tulevaa tarvetta, jossa käyttäjien tarvisi nähdä toistensa tietoja (voisi olla esimerkiksi pääkäyttäjänäkymä, jossa näh-

dään useiden käyttäjien sessioita), ei ole tarpeen tuoda käyttäjiä REST-rajapintaan mukaan lainkaan. Sen sijaan jokainen asiakaspään sovellus voi käyttää rajapintaa ikään kuin se olisi palvelimen ainoa käyttäjä. Palvelimen tulee joka tapauksessa aina tarkastaa käyttäjän valtuutus, mutta tehdyllä toteutuksella asiakkaan ei tarvitse ottaa asiaan mitään kantaa.

Koska toteutusteknologiapinossa on mukana kaksi erillistä sovelluskehystä (Angular.js ja Django), jotka molemmat toteuttavat MV*-mallin, on hyvä selventää toteutetun järjestelmän toimintaa MV*-suunnittelumallin kannalta. Tässä tapauksessa sovelluskehysten tarjoama tuki MV*-mallille ei suinkaan ole yhteensopiva, vaan Django tukee palvelinpainotteista, perinteisempää mallia, kun taas Angular.js asiakaspainotteista single-page-toteutusta. Tämän takia esimerkiksi se, mikä Django-kehyksen näkökulmasta on näkymä, ei ole lainkaan sama asia kuin Angular.js-kehyksen näkymä.

Tilanteessa voidaan nähdä fraktaali-MVC, jossa uloimmasta MVC-mallista vastaa palvelinsovelluskehys Django. Järjestelmän pääsivu, sisäänkirjautumistoiminnot sekä palautteen antajan näkymä toimivat selkeästi sellaisina palvelinpainotteisina sivuina, joihin Django erityisesti on tarkoitettu. Esimerkiksi palautteen antajan näkymä jakautuu Djangon MVT-malliin seuraavasti:

- Model: Tietokantaan kuvautuvat (Python) luokat, sekä niitä vastaavat relaatiotietokannan taulut. Esimerkin tapauksessa tietokannassa on tiedot sessiosta sekä siihen liittyvistä palautetyypeistä, jotka halutaan esittää käyttäjälle.

- View: Tiettyyn URL:aan tehtyä kyselyä vastaava Python-funktio, jossa noudetaan tietokannasta tarvittava tieto ja välitetään se templaten muodostukseen.

- Template: HTML-sivupohja, johon syötetään muuttujatiedot ja josta muodostetaan varsinainen asiakasselaimelle HTTP:n yli lähetettävä käyttäjälle näkyvä HTML-sivu.

Jaottelusta näkyy selvästi palvelinpainotteinen toteutus, sillä palvelin palauttaa valmiita HTML-sivuja, jotka asiakasselain vain esittää sellaisenaan. Palautepainikkeet on toteutettu normaalin HTML-lomakkeen tapaan: napin painallus aiheuttaa palvelimelle HTTP POST -kutsun ja kutsussa kulkee lomakkeen tietona painetun palautteen tyyppi. Tehtyyn POST-pyyntöön palvelin palauttaa uuden kokonaisen HTML-sivun, jossa palautteen antamista voi jatkaa.

Sen sijaan, kun palvelimelta pyydetään järjestäjänäkymä, on tilanne Djangon MVT-mallin kannalta hieman erilainen, joskin kehyksessä hyödynnetään täysin samoja toimintoja.

- Model: Tietokantaan tallennettavia tietoja ei tarvita.

- View: Tarvittava toteutus on hyvin ohut, sillä näkymän luonnissa ei tarvita tiedon noutoa tai käsittelyä.

- Template: HTML-sivupohja on Djangon näkökulmasta hyvin staattinen, eli siihen ei ole tarvetta sijoittaa paljoa muuttujatietoa. HTML-sivupohjasta muodostettava sivu on kuitenkin Angular.js:n käyttämä pääsivu, johon Angular.js luo koko järjestäjänäkymän ja sen toiminnot - mutta vasta asiakaspäässä. Luotava HTML-sivu sisältää linkit tarvittaviin JavaScript-tiedostoihin, jotka palvelin tarjoaa staattisina tiedostoina (kuten JavaScript-, CSS- ja kuvatiedostot normaalistikin).

Djangon MVT-mallin Template-kerros siis tavallaan sisältää koko Angular.js:n MVW-kokonaisuuden. Kun Angular.js-koodin suoritus alkaa selaimen vastaanotettua yllämainittu HTML-sivupohjasta muodostettu Angular.js:n pääsivu, on sovelluspinon toinen ja sisempi MV*-malli käynnistynyt. Angular.js:n MVW-mallin osat ovat seuraavanlaiset, kun esimerkiksi halutaan esittää tietyn session palautegraafi.

- Model: JavaScriptin REST-kommunikoinnista vastaava toteutus, palvelin sekä sen alla toimiva tietokanta kokonaisuudessaan sekä REST-kutsujen palauttamien tietojen luokat. Esimerkiksi palvelimelle tehdään REST-kutsu palauteresurssiin. Kyselyä rajataan lisäparametreilla, joissa kerrotaan aikaväli, jonka sisään jäävät palautteet halutaan tietää. Palvelin lisäksi suodattaa palautettavaa tietoa oikean käyttäjän mukaan. Djangossa käydään läpi kaikki MVT:n osat, mutta useat näistä ovat tässä tilanteessa triviaaleja.

- View: Angular.js:n oman HTML-sivupohjan pohjalta muodostettava näkymä.

- Controller (tai vastaava): Koodi, jolla mallilta saatava data muutetaan sopivaan muotoon ja välitetään näkymälle.

Palvelimella on siis kolme erilaista tilannetta. Ensimmäisenä täysiverisen MVT-mallin mukaan toiminta. Toisena tapaus, jossa asiakas pyytää ensimmäistä single-page-sivua, jolloin toiminta on MVT-mallin mukaista, mutta palvelimen kannalta yksinkertaista. Kolmantena on tilanne, jossa palvelin toimii asiakas-MV*:n mallin osana vastaten REST-pyyntöihin. Tällöinkin MVT-osat voidaan halutessa erottaa palvelimen toiminnasta, mutta tämä ei ole mielekästä, sillä käytännössä palvelin on vain ohut REST:n toteuttava kerros tietokannan päällä.

Koska palvelintoteutuksessa on hyödynnetty kirjastototeutusta, joka sarjallistaa ja purkaa suoraan Djangossa toteutettuja malliluokkia, on vaarana, että asiakaspään toteutus toteutetaan riippuvaiseksi palvelintoteutuksen malli-luokkien yksityiskohdista. Jos Djangon malliluokat lähetettäisiin sellaisenaan sarjallistettuna asiakaspäähän, jossa taas JSON-muotoinen viesti puretaan JavaScript-olioksi, tarkoittaa tämä

sitä, että esimerkiksi muutos Djangon malliluokan kentän nimeen johtaa siihen, että JavaScript-koodissa joudutaan tekemään vastaavia muutoksia. Jos oletettavissa olisi paljon tämän kaltaisia muutoksia, olisi asiakastoteutus hyvä suojata palvelimen luokkamuutoksilta tekemällä jonkinlainen keskitetty malliluokkien muunnoskerros asiakastoteutukseen, niin, että asiakaspään koodissa käsiteltäisiin aina omia luokkia, jotka on muunnoskerroksessa kuvattu REST-rajapinnan palauttamiin tietoihin. Näin raskasta suojautumista muutoksia vastaan ei nähty tarpeelliseksi. Osittain muutoksilta suojaa se, että Django REST Framework tarjoaa mahdollisuuden vaikuttaa sarjallistukseen siten, että esimerkiksi sarjallistuksen tuloksen kenttien nimiä voi muuttaa ja kokonaan uusia kenttiä voidaan muodostaa sarjallistuksen tulokseen esimerkiksi johtamalla metodilla yksi arvo useasta malliluokan kentästä.

Ylläpitonäkymän sisäinen navigointi tai tila ei toteutuksessa heijastu selaimen URL:aan. Ylläpitonäkymässä ei suoriteta sen kaltaisia näkymän tilamuutoksia tai navigointia, että tämä ominaisuus olisi tarpeellinen järjestelmän nykyisessä, koekäyttöön tarkoitetussa versiossa. Ainoa nähtävissä oleva hyöty olisi mahdollisuus luodan selaimen kirjanmerkkejä suoraan eri sessiotyyppien ylläpitosivuille.

5.3 Single-page-mallin hyödyntäminen toteutuksessa

Tässä luvussa on kuvattu järjestelmän single-page-toteutusmallia hyödyntävän ylläpitonäkymän toteutus.

Liitteessä B on kuvattu sekvenssikaaviona ylläpitonäkymän toiminta näkymän avauksen yhteydessä. Sekvenssikaavion oliot ovat Angular.js:n komponentteja lukuun ottamatta olioita "WWW-selain"ja "Palvelin (Django)", jotka kuvaavat nimiensä mukaisia abstrakteja kokonaisuuksia. Sekvenssikaavio kuvaa tilannetta, jossa käyttäjä navigoi jollain tavalla ylläpitonäkymän URL:aan. Tällöin sivulle tulee näkyviin lista kaikista sessiotyypeistä, valituksi sessiotyypiksi tulee automaattisesti se, johon liittyy tuorein annettu palaute ja tähän valitun sessiotyypin uusimpaan sessioon liittyen ladataan kaikki palautteet. Lisäksi asiakas jää kiertokyselemään (engl. polling) tasaisin välein valitun session palautteita, jotta graafi pysyy ajantasalla silloin, kun palautteita annetaan samalla kun ylläpitonäkymää pidetään avoinna. Tiedon lataus palvelimelta tapahtuu AJAX-tyyppisesti asynkronisesti taustalla, eli käyttöliittymä pysyy näkyvissä ja käytettävänä esimerkiksi sillä aikaa, kun taustalla suoritetaan graafin datan noutoa ja päivitystä.

Käytetyt teknologiat ylläpitonäkymässä ovat Angular.js, Bootstrap ja Google Charts. Angular.js toteutus pyrittiin tekemään mahdollisimman paljon yleisten suositusten ja esimerkkien mukaiseksi. Toteutus jaettiin moduuleihin siten, että esimerkiksi kaikki luodut controller-komponentit ovat omassa moduulissaan.

Toteutuksessa service-komponentteja kokoavassa moduulissa on omat moduulit jokaista MV*-mallin mallikerroksen luokkaa varten (esimerkiksi *Sessiotyyppi-*

Service). Näitä malliluokka-servicejä käytetään datan noutamiseen palvelimelta, siten että tiettyä mallikerroksen luokkaa vastaavan servicen kautta voi noutaa, luoda ja päivittää kyseisen tyyppisiä malliolioita. Toteutetut palvelinkommunikointia suorittavat Angular.js-servicet hyödyntävät sovelluskehyksen valmista *ngResource*-moduulia ja sen tarjoamia yleisiä REST-kommunikointitoteutuksia, joiden avulla omaa toteutusta kutakin REST-resurssia kohden vaadittiin vain muutamia koodirivejä. Epästandardeissa kommunikointitavoissa toteutus täytyisi itse laatia tätä matalammalla tasolla. Palvelinkommunikoinnin lisäksi tietomuunnos mallikerroksen tyypeistä GoogleCharts API:n ymmärtämään muotoon eristettiin omaksi Angular.js-serviceksi, sillä muunnos ei ole triviaali toteuttaa. Tietoa muuntavan komponentin rajapinnat pyrittiin suunnittelemaan niin, että päänäkymän controller-komponenttia ei tarvisi muuttaa, jos käyttöön haluttaisiin ottaa erilaisella API:lla varustettu graafikomponentti.

Ylläpitonäkymä koostettiin kahdesta erillisestä MV*-mallin näkymästä. Ensimmäinen näkymä on pääsivu ja sen sessiotyypin ja session valinnat sekä graafi ja siihen liittyvä tieto. Toinen näkymä on sessiotyypin luontia ja muokkausta varten tehty komponentti, jota käytetään Bootstrap-kirjaston tarjoamassa web-sivulla esitettävän modaalisen dialogin rungossa. Kumpaakin näkymää varten on toteutettu oma Angular.js-controller, joka vastaa näkymän sisältävän tiedon noutamisesta näytettäväksi ja tehtyihin valintoihin ja muokkauksiin reagoinnista. Esimerkiksi dialogin controllerissa on toteutettuna metodi, joka on sidottu (Angular.js:n syntaksin avulla) suoritettavaksi, kun dialogin tallenna-nappia painetaan.

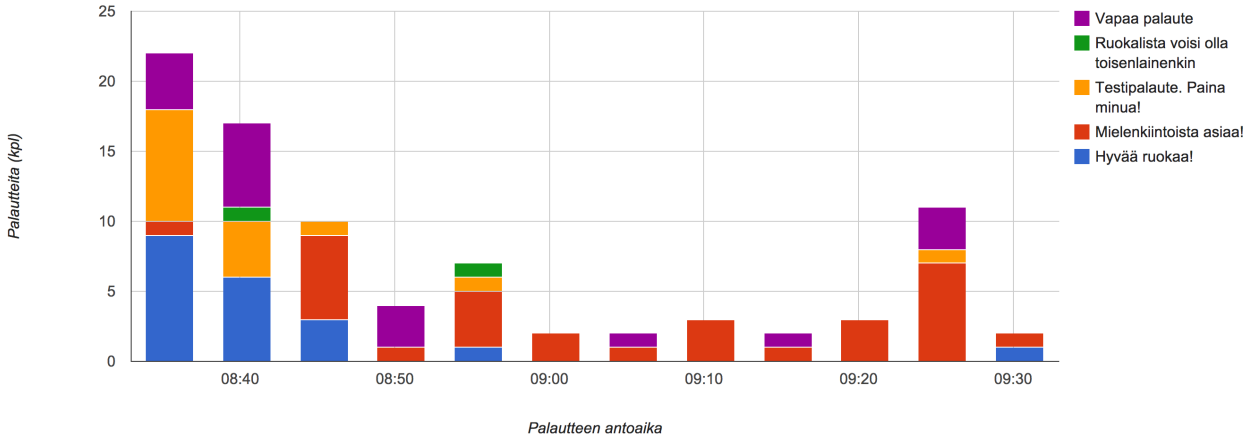
Koko sivustolle yhteinen yläpalkin valikko ei ole single-page-toteutuksen vastuun alla ylläpitonäkymässä, vaan palvelintoteutus tuottaa sen järjestelmän jokaiselle sivulle samalla tavalla. Ylläpitonäkymässä palvelimen palauttama runko, jossa on Angular.js:lle määritetty HTML-elementti single-page-sovelluksen näkymän luontia varten, sisältää myös ylävalikon, jonka olemassaolosta asiakaskoodilla ei näin ollen ole tietoa. Tämä ratkaisu johtuu siitä, että sama yläpalkin valikko on näkyvillä sivuston kaikilla sivuilla, ja nykyisellä toteutustavalla samaa palkkia ei toteutettu kahteen kertaa eri paikkoihin, vaikka huonona puolena ratkaisu saattaa hämärtää sovelluksen toiminnan hahmottamista, koska palvelinpainotteista- ja single-page-toteutusta on yhdistetty samaan selaimessa näkyvään näkymään.

6. TOTEUTUKSEN ARVIOINTI

Tässä luvussa esitellään toteutetun järjestelmän koekäytöistä saatuja tuloksia. Koska käyttökokemuksista ei tehty erityistä tutkimusta, eikä se ole työn päätutkimuskysymys, ei käyttökokemusten keräämisestä ja siihen käytetyistä metodeista ole erikseen lukua. Sen sijaan tässä luvussa käydään käyttökokemukset ja niiden keräys tuloksia samalla arvioiden läpi. Tekninen toteutus puolestaan on kuvattu luvussa 5 ja seuraavissa luvuissa on esitettynä kommentit ja arviot tästä aiemmin kuvatusta toteutuksesta.

6.1 Kokemuksia järjestelmän koekäytöstä

Tässä luvussa on kerrottu suoritetuista järjestelmän koekäytöistä käyttötilanteen luonne, kuvattu miten järjestelmä toimi tilanteessa ja koettiinko sen tarjoamat tulokset hyödyllisiksi. Koekäyttäjiksi saatiin sovittua sekä yrityksen sisäinen infotilaisuus, että yliopiston luento. Haasteita koekäyttäjien etsinnälle aiheutti se, että tämän työn puolesta koekäytöille sopiva aika oli kesälomakaudella. Jonkinlaista viitettä kiinnostuksesta järjestelmän koekäyttöön voi antaa se, että 16 kesäopetusta järjestävästä opettajasta 2, eli noin 13% oli kiinnostunut koekäytöstä. Kiinnostuneiden osuutta kontaktoiduista henkilöistä voitanee pitää kohtuullisena. Potentiaalisia koekäyttäjiä lähestyttiin sähköpostitse.



Kuva 6.1: Saatujen palautteiden graafiesitys erään yrityksen sisäisestä aamupala-infotilaisuudesta.

Järjestelmää koekäytettiin erään yrityksen kuukausittaisella, koko henkilöstön laajuisessa, noin tunnin mittaisessa infotilaisuudessa, jonka yhteydessä tarjoillaan myös aamupalaa. Tilaisuudessa oli järjestelmän koekäytön yhteydessä noin 30 henkilöä. Session palautteiden graafiesitys on nähtävissä kuvassa 6.1.

Yksittäisiä palautteita järjestelmään annettiin tilaisuuden aikana noin 80, eli hieman yli kaksinkertainen määrä osallistujamäärään nähden. Vapaiden palautteiden osuus annetuista palautteista oli noin 12% loppujen ollessa valmiiksi järjestelmään määritettyjä palautteita. Sessioon oltiin lisätty yksi valmispalautetyyppi, "Testipalaute. Paina minua!", jonka oli tarkoitus olla palautearvoltaan neutraali testipalaute, jolla yleisö voi testata järjestelmän toimintaa. Annettujen testipalautteiden osuus kaikista palautteista oli noin 18%. Saaduista palautteista erottuu muutamia kohtia, joissa esitetty asia on ollut keskimääräistä mielenkiintoisempaa. Erityisesti mielenkiintoinen kohta erottui esityksen loppupuolelta, jossa kerrottua asiaa oltiin kehuttu myös vapaissa palautteissa. Huomion arvoista on, että valmiissa palautetyypeissä oli negatiiviset palautetyypit "Liian pitkä tilaisuus"ja "Tämä asia ei kiinnosta". Kumpaakaan näistä palautteista ei annettu lainkaan. Sen sijaan tilaisuuden ruokalista sai joitain kriittisiä palautteita sekä valmiina palautetyyppinä, että vapaissa palautteissa. Voitaisiin siis päätellä, että yleisö antoi negatiivistakin palautetta, jos se antajan mielestä oli aiheellista, ja näin ollen tilaisuuden sisältöä ja kestoa voitaisiin pitää onnistuneena.

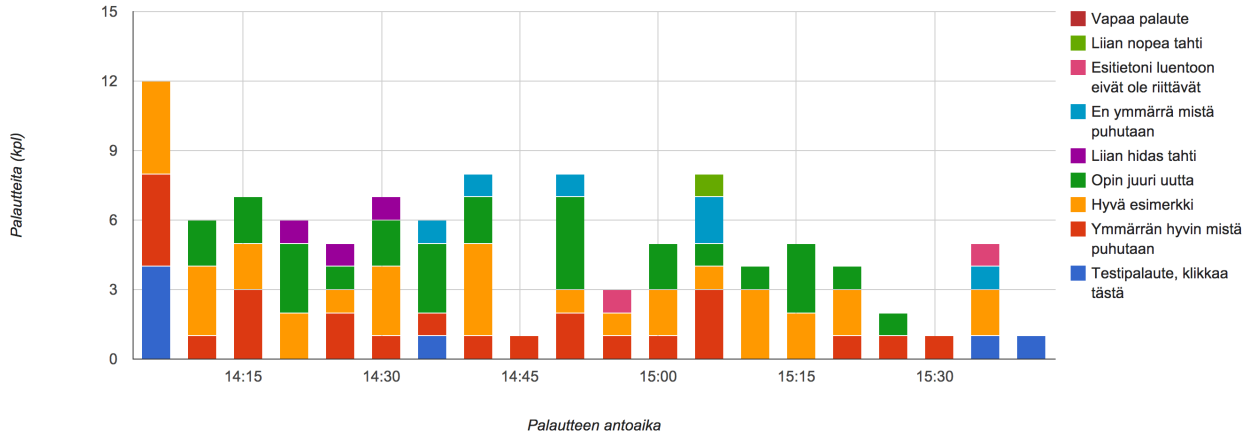
Tiukasti antoaikaan liittyvien palautetyyppien lisäksi sessiossa oli myös palautetyyppejä liittyen tarjolla olleeseen aamupalaruokatarjoiluun, joka oli saatavilla ennen varsinaisen infotilaisuuden alkua. Suuri osa tämän tyyppisistä, ei selvästi antoaikaansa liittyvistä, palautteista annettiin tilaisuuden alkupuolella. Myös testipalautteita annettiin selvästi eniten tilaisuuden alussa. Muut palautteet jakautuivat melko tasaisesti pitkin koko tilaisuutta, kuitenkin niin, että muutama piikkikohta erottuu.

Palautteen saajat pitivät aamupala-infotilaisuuden yhteydessä tehtyä koekäyttöä onnistuneena ja muita mahdollisia yrityksen tilaisuuksia, joissa järjestelmästä voitaisiin hyötyä alettiin pohtia, joskaan konkreettisia suunnitelmia järjestelmän käytöstä ei ole vielä tätä työtä kirjoittaessa tehty.

Tilaisuuden palautteita tarkastellessa tuli ilmi tarve tehdä ylläpitonäkymässä omia aikarajauksia automaattisesti luotavien vuorokauden mittausten sessioiden sijaan. Yksittäinen, esimerkiksi testimielessä annettu palaute tilaisuuden todellisen tapahtuma-ajan ulkopuolella vaikeuttaa graafin lukemista, sillä kaikki saman vuorokauden aikana annetut palautteet skaalataan automaattisesti näkymään graafissa.

Lisäksi kokeiluissa tuli ilmi tarve pystyä jälkikäteen helpottamaan sen muistamista, mitä tilaisuudessa on tapahtunut tiettyyn kellon aikaan. Yksi nykyisellä järjestelmällä toimiva tapa olisi kirjoittaa järjestelmään vapaa palaute -toiminnolla omia

muistutuksia sopivissa avainkohdissa.



Kuva 6.2: Saatujen palautteiden graafiesitys yliopiston luennolta.

Järjestelmän toinen koekäyttö suoritettiin yliopiston automaatiotekniikan luennolla. Osallistujia luennolla oli 12 henkilöä. Session palautteiden graafiesitys on nähtävissä kuvassa 6.1.

Palautteita järjestelmään annettiin tilaisuuden aikana noin 100. Keskimäärin jokainen osallistuja siis antoi noin 8 yksittäistä palautetta tilaisuuden aikana. Testipalautteita ("Testipalaute, klikkaa tästä") annetuista palautteista oli noin 5%. Vapaita palautteita ei annettu yhtään kappaletta.

Annetut palautteet jakautuivat melko tasaisesti koko tilaisuuden ajalle tarkasteltiinpa palautteita sitten kokonaisuutena tai palautetyyppikohtaisesti. Selviä piikkejä palautteissa ei erotu. Kuitenkin voidaan huomata, että vain tilaisuuden ensimmäisen puolen tunnin aikana oltiin annettu palautetta "Liian hidas tahti"(yhteensä 3kpl) ja että ainoastaan luennon ensimmäisen puolen tunnin jälkeen oltiin annettu negatiiviseksi laskettavia palautteita (2kpl "Esitietoni luentoon eivät ole riittävät", 6kpl "En ymmärrä mistä puhutaan"ja 1kpl "Liian nopea tahti"). Voidaan päätellä, että luennon alkuosa koettiin hieman helpommaksi kuin luennon loppuosa. Valtaosa annetuista palautteista oli positiivisia. Syinä siihen, että palautteista ei selvästi erotu erityisen hyvin tai huonosti menneitä kohtia luennosta voi olla se, että palautteenantajien määrä oli pieni kvantitatiiviselle palautteenkeräysmenetelmälle ja se, että opetuksessa ei oikeasti ollut selvästi muuta luentoa parempia tai huonompia kohtia.

Palautteen saaja koki, että luennolla tehdystä palautteenkeräyksestä toteutetulla järjestelmällä voi olla hyötyä seuraaviin luentoihin ja myös jo kuluvan luennon aikana. Luentokokeilun aikana järjestelmän ylläpitonäkymän käytössä tuli vastaan ongelma, joka johtui käytetystä selaimesta, joka oli asetettu estämään evästeiden käyttö. Järjestelmä osasi huomioida estetyt evästeet vain osittain. Muutoin järjestelmän käytössä ei havaittu ongelmakohtia.

Tässä työssä asetettuun tutkimuskysymykseen järjestelmän hyödystä palautteen kerääjälle voidaan todeta tehtyjen koekäyttöjen perusteella, että toteutetun järjestelmän mahdollistama tapa kerätä jatkuvaa palautetta on toimiva, sillä palautteen kerääjät kokivat järjestelmän hyödylliseksi ja palautteen antajat antoivat järjestelmällä palautetta kiitettävät määrät. Annetuista palautteista ei esimerkiksi ollut koetilaisuuksissa havaittavissa, että palautteenantoinnokkuus olisi ollut suurin tilaisuuden alussa tai lopussa, vaan palautteita annettiin koko tilaisuuksien ajan. On kuitenkin huomioitava, että koekäyttöjä tehtiin vain hyvin pieni määrä (kaksi kappaletta). Tämä seikka laskee esitettyjen johtopäätösten arvoa. Lisäksi on todettava, että tätä työtä tehdessä järjestelmää on käytetty todellisissa tilaisuuksissa vain erikseen pyydettäessä, eikä palautteen kerääjien omasta aloitteesta. Voi siis olla, että järjestelmää ei koeta niin hyödylliseksi, että sitä haluttaisiin käyttää, ja että koetilaisuuksien palautteenkerääjien antamat kommentit järjestelmän hyödyllisyydestä johtuvat osin inhimillisestä halusta miellyttää kysyjää. Lopullisesti järjestelmän idea voitaisiin todeta toimivaksi ja hyödylliseksi vasta, kun järjestelmällä olisi joukko käyttäjiä, jotka käyttäisivät sitä omasta aloitteestaan.

Työtä tehdessä ei löydetty muita vastaavanlaisia palautteenkeräysjärjestelmiä, joihin toteutettua järjestelmää olisi voitu verrata. Muita palautteenkeräystapoja on esimerkiksi tilaisuuden aikana pidettävät viittausäänestykset tai keskeyttämiseen kannustaminen, joilla saadaan kerättyä palautetta tilaisuuden aikana ja sen tiettyyn kohtaan liittyen. Tietotekniikkaa hyödyntäviä tapoja on esimerkiksi web-palautelomakkeet, joita voidaan pyytää täyttämään tilaisuuden tai kurssin päätyttyä.

6.2 Teknisen toteutuksen arviointi

Toteutetussa järjestelmässä tietynlaisena epäkohtana voidaan nähdä se, että tutkittavien teknologioiden nimestä (joka luonnollisesti juontuu teknologioiden tarkoitetusta käyttötavasta) huolimatta single-page-teknologioilla toteutettiin vain yksi osa web-järjestelmästä, eikä koko järjestelmää yhtenä single-page-sovelluksena. Single-page-tekniikoilla toteutettavaksi valittu ylläpitonäkymä oli silti siinä määrin itsenäinen kokonaisuus, että tekniikoiden käytöstä saatiin tarkoitettua käyttöä vastaavia kokemuksia. Osaltaan single-page-tekniikoiden käyttäminen vain osassa toteutettavaa järjestelmää johtui siitä, että järjestelmän toteutusta oltiin jo alettu hahmotella ennen kuin single-page-teknologioiden käyttö valikoitui tämän työn tutkimuskysymykseksi.

Toinen syy, miksi koko järjestelmää ei toteutettu yhtenä single-page-sovelluksena oli se, että toteutuksen suunnittelua aloitettaessa päädyttiin varhaisessa vaiheessa päätökseen, että palautteen antajan toimintojen tulee toimia perinteisenä websivuna, jotta sivu olisi mahdollisimman laajan käyttäjäjoukon käytettävissä. Jäl-

kikäteen tätä päätöstä voidaan pitää huonosti perusteltuna; käytännössä jokaisella potentiaalisella käyttäjällä on käytettävissään selain, joka tukee JavaScriptia, jos käyttäjä ei erikseen ole itse sen suoritusta estänyt. Esimerkiksi kaikki käytetyimmistä älypuhelinkäyttöjärjestelmistä sisältävät selaimen, joka tukee kaikkia single-page-websovelluksen vaatimia ominaisuuksia (tärkeimpänä JavaScript). Palautteenantonäkymää voidaan myös pitää luonteeltaan sellaisena, että ei ole ehdottoman tärkeää pystyä takaamaan jokaiselle sen käytettävyyttä. Toisaalta esimerkiksi näkövammaisille tarkoitetut selaimet ovat kehittyneet siten, että niilläkin on mahdollista käyttää JavaScriptiä käyttävää single-page-sovellusta, kunhan tietyt asiat otetaan toteutuksessa huomioon [23]. Käytettävyyden ja saatavuuden kannalta olisikin luultavasti lähes tai täysin merkityksetöntä, onko toteutus perinteinen web-sivu vai single-page-websovellus. Ainoat merkittävät erot jäävät siis kehitystyön ja ylläpidon työmäärään. Tältä kannalta single-page-toteutuksen hyötyjä olisi ollut koko järjestelmän laajuinen yhteinen asiakaspään toteutustapa, ohjelmistokehys ja ohjelmointikieli. Saman asian toisena puolena olisi myös luonnollisesti palvelintoteutuksen roolin vakioiminen samanlaiseksi, kuin se nyt on single-page-ylläpitonäkymälle. Tehdyn single-page-ylläpitonäkymän pohjalta voitaisiin myös arvioida, että työmäärä ei olisi oleellisesti erilainen, joskin ennalta tutut teknologiat nopeuttivat tässä tapauksessa palautteen antaja -toiminnallisuuden toteutusta palvelinpainotteisesti. Nykyinen, eri toteutustapoja sekoittava toteutus voi kuitenkin olla hidasteena järjestelmän teknistä rakennetta selvitettäessä, jos järjestelmää joudutaan tulevaisuudessa ylläpitämään.

Nykyisessä järjestelmän demoamiseen tarkoitetussa versiossa ei kehitystyössä kohdattu ongelmia tai hidasteita, joiden olisi koettu johtuvat siitä, että palautteen antajan ja ylläpitäjän näkymät toteutettiin eri teknologioilla. Vaikka palautteen antajan näkymä on toiminnallisuudeltaan hyvin suppea, törmättiin silti tilanteisiin, joissa single-page-toteutustekniikat olisivat olleet hyödyksi. Yksi tällainen on palautteen antajan näkymässä näytettävä info-laatikko, jolla käyttäjälle kerrotaan, että palaute on vastaanotettu onnistuneesti tai että palautteen käsittelyssä on tapahtunut virhe. Info-laatikossa olisi ollut käytettävyyden kannalta selkeää näyttää esimerkiksi numerolla kuinka monta palautetta on vastaanotettu sen jälkeen, kun käyttäjä viimeksi on kuitannut info-laatikon nähdyksi. Nykyisellä palvelinpainotteisella toteutuksella tämä vaatisi ylimääräisen tilatiedon kuljettamista aina palautteen annon yhteydessä asiakkaan ja palvelimen välillä. Single-page-sovelluksessa laskuriominaisuuden lisäys olisi vaikuttanut pelkästään asiakassovellukseen ja olisi ollut helppo toteuttaa; laskuria varten olisi tehty uusi integer-muuttuja JavaScript-toteutukseen, todennäköisimmin Angular.js controller-komponenttiin. Vaikka tämä laskuriominaisuus on melko merkityksetön, osoittaa sen tarkastelu, että nykyinen ratkaisu verrattuna vastaavaan single-page-toteutukseen voi olla huomattavasti vaikeampi laa-

jentaa, jos tulevaisuudessa palautteen antajalle haluttaisiin lisätä mitään nykyistä monimutkaisempia toimintoja. Jos tarvetta palautteen antajan näkymän laajentamiselle tulisi, niin luultavasti samalla ainakin tullaan harkitsemaan koko näkymän muuttamista single-page-teknologioilla toteutetuksi.

Single-page-toteutusta suunniteltaessa yhtenä suurimpana epävarmuutena oli Django-sovelluskehyksen soveltuvuus käytettäväksi single-page-web-sovelluksessa sekä asiakas- ja palvelinsovellusten keskinäisen kommunikoinnin toteutuksen työmäärä. Oli tiedossa, että molempiin päihin oli saatavilla valmiita REST-kommunikoinnin mahdollistavia yleisiä toteutuksia, mutta näiden yhteensopivuus esimerkiksi sarjallistamismuotojen ja kyselyn GET-parametrien asettamisen ja tulkitsemisen osalta, muissa kuin kaikista triviaaleimmissa tapauksissa, oli epävarmaa. Vaikka jonkinlaisia hidasteita osattiin odottaa, sujui toteutus yllättävänkin sujuvasti. Molemmissa päissä käytetyt REST-kommunikoinnin abstrahoivat toteutukset täyttivät kaikki tarpeet ja ennen kaikkea toteuttivat ne keskenään yhteensopivalla tavalla. Ainoa pieni ongelma tuli vastaan siinä, kuinka kirjastot halusivat, tai eivät halunneet, REST-resurssien URL:n perässä olevan /-merkkiä, mutta tämänkin ratkaisu oli helpohkosti konfiguroitavissa Djangoon, joskin vaaditun konfiguraation dokumentointi olisi voinut olla nykyistä selkeämpi ja paremmin saatavilla.

Yleisesti kaikkia toteutuksessa käytettyjä teknologioita voidaan pitää kypsinä ja tuotantoon soveltuvina. Ainoa poikkeus tähän on käytetty kolmannen osapuolen toteuttama Angular.js-laajennos GoogleChartin käyttöön. Laajennoksen dokumentaatio ei ollut samalla tasolla muiden käytettyjen teknologioiden kanssa. Tämä on siedettävissä oleva puute, kun huomioi laajennoksen erittäin rajatun vaikutusalueen sovelluksessa; laajennos on kohtalaisella työmäärällä korvattavissa, jos ongelmia ilmaantuu.

6.3 Johtopäätökset single-page-toteutusmallista

Ennakko-oletuksena työssä oli, että single-page-toteutusmallin avulla web-sovelluksen käyttöliittymästä saataisiin käytettävyyden kannalta perinteiseen verrattuna parempi nopeuden ja välittömyyden ansiosta. Lisäksi oletettiin, että selvästi enemmän käyttöliittymällä varustettua sovellusta kuin staattista dokumenttia muistuttava ylläpitonäkymä saataisiin toteutettua pienemmällä vaivalla, koska käytettävät teknologiat olisivat kotonaan toteutetun järjestelmän kehityksessä. Myös ylläpitonäkymän skaalautuvuuden oletettiin olevan palvelinpainotteista web-sovellusta parempi, koska prosessointikuormaa on siirretty palvelimelta asiakkaalle.

Käyttöliittymän vasteaikoja ja reagoivuutta arvioidessa ei ole yksiselittäistä mihin toteutettua järjestelmää tulisi verrata. Täysin JavaScriptitön toteutus on hieman naiivi vertailukohta, sillä monasti palvelinpainotteisetkin web-sivut voivat sisältää JavaScriptiä asiakaspäässä esimerkiksi pieniä käytettävyysparannuksia tuomassa.

Palvelinpainotteisella MV*-mallilla toteutettu ja asiakaspäästä AJAX-teknologioilla höystetty toteutus voi puolestaan olla lähes mitä tahansa naiivin palvelinpainotteisen toteutuksen ja täysiverisen single-page-toteutuksen välillä. Koska single-page-toteutusmalli on käytännössä pelkästään tietynlainen tapa käyttää AJAX-teknologioita, ja kaikki käytettävyyteen vaikuttavan single-page-toteutuksen edut aiheutuvat nimenomaan AJAX-teknologioista, voidaan todeta, että käytettävyyden kannalta millä tahansa AJAX-teknologioita hyödyntävällä toteutuksella voidaan saavuttaa samanlaiset käyttöliittymän vasteajat ja reagoivuuden kuin single-page-toteutuksella.

Single-page-toteutustavan etu muunlaiseen AJAX-teknologioiden käyttöön verrattuna on kuitenkin se, että MV*-mallin noudattaminen on single-page-toteutuksessa helpompaa ja luonnollisempaa. Palvelinpainotteisessa toteutuksessa AJAX-teknologioiden hyödyntäminen yksittäisissä ominaisuuksissa voi helposti aiheuttaa esimerkiksi toimintalogiikan hajoamisen sekä palvelin että asiakastoteutukseen epäselvällä tavalla, jos toteutusta ei suunnitella etukäteen erityisen hyvin.

Single-page-toteutusmallin käyttäjälle näkyviä nopeusetuja tutkittiin mittaamalla viive sessiotyypin muokkausdialogin avauksesta. Mittaukset tehtiin tutkimalla selaimen tapahtumia Google Chrome (versio 35.0) -selaimeen sisäänrakennetulla kehittäjätyökalulla (Developer Tools). Chromen kehittäjätyökalun avulla voidaan tarkastella sivun latauksen ja muodostuksen vaiheita ajan suhteen. Tällä työkalulla mitattiin usean kerran peräkkäin aika hiiren painallus -tapahtumasta siihen, kun viimeinen muokkausdialogissa tarvittava tieto oli ladattu palvelimelta. Mittaukset olivat keskenään samaa suuruusluokkaa, noin 500ms. Mittauksissa käytetyssä tietokoneessa oli Inter Core i5 prosessori ja OS X 10.9 -käyttöjärjestelmä. Verkkoviive mittaustietokoneelta (asiakas) palvelimelle oli noin 33ms.

Dialogin avauksen 500ms viive on sen suuruinen, että jos järjestelmä reagoisi käyttäjän syötteeseen vasta sen kuluttua, kokisi käyttäjä, että järjestelmä ei ole välitön (viiverajat mainittu luvussa 2.2, sivu 17). Näin olisi, jos järjestelmä olisi toteutettu palvelinpainotteisesti ja ilman AJAX-teknologioita. Nykyisessä järjestelmässä kuitenkin dialogin runko avautuu lyhyellä animaatiolla, joka alkaa heti käyttäjän syötteen jälkeen. Samalla käynnistetään myös tarvittavan tiedon lataus taustalla, ja sitä mukaa kun palvelimelta saadaan kysellyt tiedot, ne populoidaan dialogiin. Käytännössä mitatun suuruisilla latausviiveillä dialogi sisältää kaiken tiedon heti kun sen avausanimaatio on päättynyt. Järjestelmän reagoivuus on siis huomattavasti parempi kuin tiedon lautauksiin tarvittava 500ms ja järjestelmä vaikuttaa tämän takia sulavalta ja välittömältä.

Ylläpitonäkymässä esitetään saadut palautteen graafina ja graafia pidetään jatkuvasti ajantasalla kiertokyselemällä palvelimelta uusia palautteita. Järjestelmässä tämä on toteutettu niin, että uusien palautteiden lataus tapahtuu täysin taustalla ja käyttäjälle tämä näkyy ainoastaan kun graafiin piirretään näkyviin uutta tietoa.

Järjestelmä pysyy käytettävänä jatkuvasti kiertokyselyn ohella. Palvelinpainotteisessa toteutuksessa tämän kaltainen taustalla tapahtuva kiertokysely ei olisi mahdollista, vaan koko sivu täytyisi ladata ajastetusti uudelleen, jolloin latausviiveestä tulisi käyttäjälle näkyvä, eikä sen aikana järjestelmä olisi käytettävissä.

Verrattuna palvelinpainotteiseen toteutukseen käyttäjälle näkyvät edut voidaan siis todeta saavutetuiksi single-page-mallin ansiosta. Viiveitä verratessa on syytä muistaa, että latausviiveet eivät olisi identtiset nykyisen toteutuksen ja palvelinpainotteisen toteutuksen kesken, sillä siirrettävä tieto vastaavissa tilanteissa olisi erilainen. Nykyinen järjestelmä siirtää palvelimelta ainoastaan tarvittavan tiedon JSON-sarjallistettuina mallikerroksen olioina, kun taas palvelinpainotteinen toteutus siirtäisi valmiin HTML-näkymän, joka sisältäisi muun muassa esitysmuotoon muunnettuna samat tiedot. Tässä single-page-toteutuksen voidaan sanoa olevan parempi, koska siirrettävän tiedon määrä on pienempi. Toinen ero totetustapojen välillä olisi se, että järjestelmä noutaa nykyisessä toteutuksessa näkymän tarvitsemia tietoja usealla kyselyllä (esimerkiksi kun näkymään tarvitaan tietoa usealta REST-resussilta). Palvelinpainotteisessa toteutuksessa siirrettäisiin vain yksi valmis HTML-tiedosto riippumatta miten monen malliluokan tietoja se sisältää. Tämä ero puolestaan on palvelinpainotteisen toteutuksen hyväksi, koska tietyn tietomäärän siirtäminen on nopeampaa yhdellä HTTP-kysellyllä kuin usealla. Nämä erot eivät kuitenkaan muuta aiemmin todettua johtopäätöstä.

Järjestelmän kehitystyö käyttäen Angular.js-kehystä muistutti luonteeltaan enemmän esimerkiksi WPF-työpöytäsovelluksen kehitystä kuin perinteisillä web-teknologioilla tehtävää web-sivun toteutusta. Se, onko tämä hyvä vai huono asia on lähinnä makuasia. Vastaavanlaisen käyttöliittymän toteutus toimintoineen ilman single-page-kehystä (mutta AJAX-teknologioita hyödyntäen) ja palvelinpainotteisemmin olisi kuitenkin ollut luultavasti työläämpää, koska asiakkaan ja palvelimen työnjako ei olisi ollut yhtä selvä.

Palautteen antajan perinteisen web-näkymän toteutuksessa tuli vastaan tilanne, jossa asiakaspään toiminnon toteutus olisi ollut single-page-sovellukseen hyvin yksinkertainen toteuttaa, mutta palvelinpainotteiseen toteutukseen oltaisiin vaadittu huomattavasti enemmän työtä vastaavan ominaisuuden aikaansaamiseksi. Toteutuksessa saadun tuntuman perusteella tämä ei ole vain yksittäinen anekdootti, vaan toimintorikkaan asiakassovelluksen toteutus todennäköisesti on helpompaa single-page-teknologioilla kuin palvelinpainotteisena sovelluksena, koska asiakassovelluksessa on suora pääsy vuorovaikuttamaan näkymän kanssa ilman, että jokainen näkymän muutos vaatii myös kierrosta palvelimella. Oleellisena seikkana ja single-page-teknologioiden etuna on se, että asiakaspäässä voidaan sovelluksessa säilyttää jatkuvasti tilatieto, eikä sitä tarvitse kuljettaa edestakaisin palvelimen ja asiakkaan välillä evästeissä jokaisessa HTTP-pyynnössä.

Skaalautuvuusetujen toteutumista ei tutkittu. Oletettavasti ylläpitonäkymällä on niin vähän käyttäjiä, ettei varsinaista tarvetta monien samanaikaisten käyttäjien käytön optimoinnille ole. Tämän toteutuksen perusteella olisi myös hankala vetää yleispäteviä johtopäätöksiä asiasta, sillä järjestelmät ja niiden palvelinkommunikoinnin tarve ovat hyvin erilaisia. Single-page-sovelluksen etuna on, että siirrettävä tietomäärä voi olla pienempi, koska samoja tietoja ei tarvitse kuljettaa toisteisesti. Toisaalta voi kuitenkin olla, että yksittäisiä kyselyitä tarvitsee tehdä useita, kuten toteutetussa järjestelmässä. On järjestelmästä riippuvaa, kumpi seikoista on merkittävämpi. Jos aikaa vievää toimintalogiikan suoritusta saadaan single-page-toteutuksella siirrettyä palvelimelta asiakkaiden harteille, on tämä ehdottomasti suuri skaalautuvuusetu, koska kuormaa saadaan automaattisesti hajautettua niin monelle erilliselle suorittajalle, kuin asiakkaita kerralla on.

7. YHTEENVETO

Tämän työn tavoitteena oli selvittää single-page-toteutustavan soveltuvuutta palautteenkeräysjärjestelmän web-ylläpitosivun toteutuksessa. Single-page-sovellus tarkoittaa web-sovellusta, jossa toiminnallisuutta on siirretty perinteiseen web-sivuun verrattuna suoritettavaksi asiakaspäässä selaimessa. Käyttöliittymäsovellusten toteutuksessa hyödynnetään usein MVC-suunnittelumallia tai sen muunnosta. MVC-mallin ja sen muunnosten tärkeimmät osat ovat malli, joka sisältää esimerkiksi tietokannan ja toimintalogiikan sekä näkymä, joka vastaa malli-kerroksen tietojen esittämisestä. Malli ja näkymä on tarkoitus pitää selkeästi erillisinä kokonaisuuksina ja löyhästi toisiinsa sidottuina. Perinteisessä web-sivussa koko malli ja osa näkymää on toteutettu palvelimella, kun taas single-page-sovelluksessa näkymä ja osa mallia kuuluvat asiakaspään toteutukseen. Usein asiakaspäässä suoritettava single-page-sovellus toteutetaan JavaScript-kielellä, ja sille ominaista on sovelluksen suorittama asynkroninen tiedonsiirto asiakkaan ja palvelimen välillä. Single-page-sovelluksen nimi juontuu siitä, että sovelluksen suorituksen aluksi selain lataa palvelimelta yhden HTML-sivun sekä siihen liittyvät JavaScript- ynnä muut tarvittavat tiedostot. Tämän jälkeen uusia kokonaisia HTML-sivuja ei ladata session aikana palvelimelta, vaan JavaScript-sovellus vastaa käyttäjän syötteisiin reagoinnista ja niiden mukaan HTML-näkymän päivittämisestä.

Ennakko-odotuksena single-page-toteutustavan valinnan hyötynä oli sivun nopeammasta reagointiajasta johtuva käytettävyyden parantuminen. Lisäksi oletettiin, että toteutustyö olisi helpompaa ja selkeämpää sekä että sovelluksen skaalautuvuus usealle samanaikaiselle käyttäjälle olisi parempi verrattuna perinteisiin web-toteutustapohin.

Työn toissijaisena tavoitteena oli selvittää toteutettavan palautteenkeräysjärjestelmän toimintaa. Palautteenkeräysjärjestelmässä ideana on kerätä tilaisuuden aikana palautteita, jotka palautteen antajat antavat valitsemalla esimerkiksi älypuhelimen selaimessa auki olevasta näkymästä valmiita palautteita tai kirjoittamalla itse lyhyitä palautteita. Annetut palautteet sidotaan niiden antoaikaan ja esitetään palautteen kerääjälle graafina ajan suhteen.

Mahdollisiksi toteutusteknologioiksi valittiin neljä vaihtoehtoa, joiden soveltuvuutta toteutettavaan sovellukseen tutkittiin. Selvityksen perusteella kaikilla vaihtoehdoilla olisi mahdollista toteuttaa tuotantokäyttöön soveltuva single-page-sovel-

lus. Käytettäväksi teknologiaksi valittiin Angular.js-JavaScript-sovelluskehys, koska se oli vertailluista teknologioista laajin ja eniten toteutusta ohjaileva. Tämän uskottiin nopeuttavan toteutustyötä.

Sovelluksen toteutuksessa ei törmätty odottamattomiin haasteisiin ja tuloksena saatiin toiveiden mukainen ylläpitokäyttöliittymä. Käyttöliittymän reaktioaikamittauksen sekä käyttökokemusten perusteella single-page-toteutustapa toi parannusta ylläpitonäkymän käytettävyyteen, jos toteutusta verrattaisiin vastaavaan web-käyttöliittymään, joka ei hyödyntäisi AJAX-teknologioita.

Toteutustyön helppoutta ja mielekkyyttä ei voitu tutkia millään absoluuttisella mittarilla, mutta toteuttajan mielipide oli se, että toteutustyö Angular.js-sovelluskehystä käyttäen muistutti enemmän esimerkiksi graafisten työpöytäsovellusten kehitystyötä kuin perinteistä web-ohjelmointia. Sovelluksen rakenteen eheyteen kannustavan MV*-mallin käyttö single-page-toteutustan kanssa oli luontevaa. Skaalautuvuusetuja ei tutkittu, mutta single-page-toteutukselle ominaisten tiedonsiirtoerojen pohjalta pääteltiin, että toteutettavan sovelluksen tyypistä ja tiedonsiirtotarpeista riippuen single-page-toteutus voi tarjota skaalautuvuusetua vähentyneen yhden asiakkaan vaatiman tiedonsiirtotarpeen ja prosessointiajan ansiosta.

Palautteenkeräysjärjestelmää koekäytettiin sekä yrityksen infotilaisuudessa (30 henkinen yleisö) sekä yliopiston luennolla (12 henkinen yleisö). Tilaisuuksien kestot olivat muutaman tunnin luokkaa. Yksittäisiä palautteita tilaisuuksissa annettiin kahdesta kahdeksaan kertainen määrä osallistujien määrään nähden ja palautteita annettiin keskimäärin tasaisesti tilaisuuksien ajan, eikä vain esimerkiksi tilaisuuden alussa tai lopussa. Yrityksen infotilaisuuden palautteista oli nähtävillä trendejä esimerkiksi siitä, mikä kohta tilaisuudesta oli yleisön mielestä erityisen mielenkiintoinen. Luennolla yhtä selviä trendejä palautteista ei ollut nähtävillä, mutta luennon alkuosa koettiin palautteiden perusteella hieman helpommaksi kuin luennon loppuosa. Järjestelmän ongelmakohtia oli esimerkiksi vaikeus muistaa jälkikäteen mitä asiaa tilaisuudessa on käyty läpi minäkin ajanhetkenä. Yleisesti järjestelmäideaa pidettiin hyödyllisenä, mutta kaksi erillisestä pyynnöstä tehtyä koekäyttöä on liian pieni aineisto kertomaan luotettavasti, onko järjestelmästä niin suuri hyöty, että palautteen kerääjät haluaisivat käyttää sitä.

LÄHTEET

[1] Athivarapu, P. K. & al. RadioJockey: Mining Program Execution to Optimize Cellular Radio Usage [WWW]. [viitattu 3.2.2014]. Saatavissa: http://research.microsoft.com/pubs/168487/RadioJockey-Mobicom12.pdf.

[2] Booth, D. & al. Web Services Architecture [WWW]. 2004. [viitattu 3.2.2014]. Saatavissa: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[3] Crockford, D. JavaScript: the good parts. Beijing 2008, O'Reilly. 153 s.

[4] Fielding, R. & al. Hypertext Transfer Protocol – HTTP/1.1 [WWW]. 1999. [viitattu 3.2.2014]. Saatavissa: http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[5] Fowler, M. GUI Architectures [WWW]. 2006. [viitattu 3.2.2014]. Saatavissa: http://martinfowler.com/eaaDev/uiArchs.html.

[6] Fowler, M. Inversion of Control Containers and the Dependency Injection pattern [WWW]. [viitattu 23.7.2014]. Saatavissa: http://www.martinfowler.com/articles/injection.html.

[7] Fowler, M. Patterns of enterprise application architecture. Boston 2003, Addison-Wesley. 516 s.

[8] Freeman, A. Pro ASP.NET MVC 4. 4. painos. Berkeley, Calif. 2012, Apress. 729 s.

[9] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. Design patterns: elements of reusable object-oriented software. Reading, Mass. 1995, Addison-Wesley. 431 s.

[10] Gossman, J. Introduction to Model/View/ViewModel pattern for building WPF apps [WWW]. 2005. [viitattu 3.2.2014]. Saatavissa: http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx.

[11] Gossman, J. Advantages and disadvantages of M-V-VM [WWW]. [viitattu 3.2.2014]. Saatavissa: http://blogs.msdn.com/b/johngossman/archive/2006/03/04/543695.aspx.

[12] Green, B. & Seshadri, S. AngularJS. Sebastopol, CA. 2013, O'Reilly. 183 s.

[13] Haapanen, M. Ajax-tekniikoiden hyödyntäminen Vaisala Rosa -sääaseman selainkäyttöliittymässä. Kandidaatintyö. Tampere 2010. Tampereen teknillinen yliopisto. 17 s.

[14] Hämmäinen, H. & al. Mobile Handset Population in Finland 2005-2013 [WWW]. 2014. [viitattu 1.8.2014]. Saatavissa: https://research.comnet.aalto.fi/public/Mobile_Handset_Population_2005-2013.pdf

[15] Leff, A. & Rayfield J. T. Web-Application Development Using the Model/-View/Controller Design Pattern. Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International. Seattle, WA. pp. 118 - 127.

[16] Mikowski, M. S. & Powell, J. C. Single page web applications: JavaScript end-to-end. Shelter Island 2014, Manning. 396 s.

[17] Minar, I. MVC vs MVVM vs MVP [WWW]. 2012. [viitattu 14.7.2014]. Saatavissa: https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV.

[18] Nielsen, J. Usability engineering. Boston 1993, Academic Press. 362 s.

[19] Osmani, A. Developing Backbone.js applications. Farnham 2013, O'Reilly. 354 s.

[20] Richardson, L. & Ruby, S. RESTful web services. Farnham 2007, O'Reilly. 419 s.

[21] Takada, M. Single page apps in depth [WWW]. [viitattu 3.2.2014]. Saatavissa: http://singlepageappbook.com/single-page.html.

[22] Turner, J., & Bedell, K. Struts kick start. Indianapolis, Ind. 2003, Sams. 504 s.

[23] Accessible JavaScript [WWW]. 2013. [viitattu 14.7.2014]. Saatavissa: http://webaim.org/techniques/JavaScript/.

[24] Adobe Flash runtimes [WWW]. [viitattu 29.6.2014]. Saatavissa: http://www.adobe.com/products/flashruntimes/benefits.html.

[25] Angular.js developer guide [WWW]. [viitattu 3.2.2014]. Saatavissa: http://docs.angularjs.org/guide/introduction/.

[26] AngularJs Google Chart Tools directive [WWW]. [viitattu 28.6.2014]. Saatavissa: http://bouil.github.io/angular-google-chart/#/fat.

[27] Backbone.js [WWW]. [viitattu 3.2.2014]. Saatavissa: http://backbonejs.org/.

[28] Bootstrap [WWW]. [viitattu 28.6.2014]. Saatavissa: http://getbootstrap.com/.

[29] Django [WWW]. [viitattu 29.6.2014]. Saatavissa: https://www.djangoproject.com/.

[30] Django documentation FAQ: General [WWW]. [viitattu 3.2.2014]. Saatavissa: https://docs.djangoproject.com/en/dev/faq/general/.

[31] Django REST framework [WWW]. [viitattu 29.6.2014]. Saatavissa: http://www.django-rest-framework.org/.

[32] Google Charts [WWW]. [viitattu 28.6.2014]. Saatavissa: https://developers.google.com/chart/.

[33] Google Trends [WWW]. [viitattu 28.6.2014]. Saatavissa: http://www.google.com/trends/.

[34] Jasmine [WWW]. [viitattu 23.7.2014]. Saatavissa: http://jasmine.github.io/.

[35] Java Plug-in Technology [WWW]. [viitattu 29.6.2014]. http://www.oracle.com/technetwork/java/index-jsp-141438.html.

[36] Knockout [WWW]. [viitattu 9.2.2014]. Saatavissa: http://knockoutjs.com/documentation/introduction.html.

[37] Making AJAX Applications Crawlable [WWW]. [viitattu 28.7.2014]. Saatavissa: https://developers.google.com/webmasters/ajax-crawling/.

[38] Model-view-controller (viitattu kuva: MVC-Process.svg) [WWW]. [viitattu 3.2.2014]. Saatavissa: http://en.wikipedia.org/wiki/File:MVC-Process.svg.

[39] Node.js [WWW]. [viitattu 28.7.2014]. Saatavissa: http://nodejs.org/.

[40] Prism 4.1 - Developer's Guide to Microsoft Prism [WWW]. [viitattu 9.2.2014]. Saatavissa: http://msdn.microsoft.com/en-us/library/gg406140.aspx.

[41] Programming Language Popularity [WWW]. [viitattu 29.6.2014]. Saatavissa: http://langpop.com/.

[42] Social Plugins [WWW]. [viitattu 9.2.2014]. Saatavissa: https://developers.facebook.com/docs/plugins.

[43] TIOBE Index [WWW]. [viitattu 29.6.2014]. Saatavissa: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[44] UI Bootstrap [WWW]. [viitattu 28.6.2014]. Saatavissa: http://angular-ui.github.io/bootstrap/.

[45] Usage of Flash for websites [WWW]. [viitattu 29.6.2014]. Saatavissa: http://w3techs.com/technologies/details/cp-flash/all/all.

[46] Usage of Java for websites [WWW]. [viitattu 29.6.2014]. Saatavissa: http://w3techs.com/technologies/details/cp-javaruntime/all/all.

[47] Velocity and the Bottom Line [WWW]. [viitattu 29.6.2014]. Saatavissa: http://programming.oreilly.com/2009/07/velocity-making-your-site-fast.html.

[48] What is a Module? [WWW]. [viitattu 10.7.2014]. Saatavissa: https://docs.angularjs.org/guide/module.

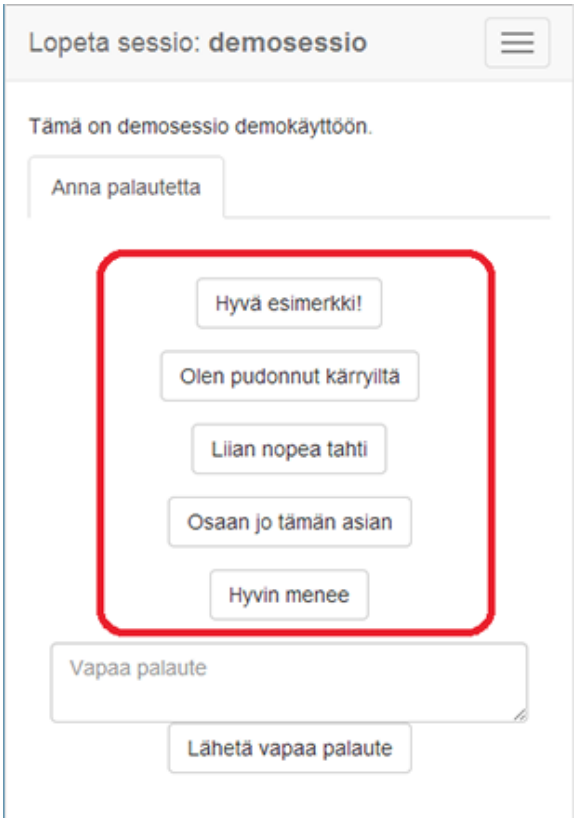
A. NETTIPALAUTE-JÄRJESTELMÄN KÄYTTÖOHJEET

Käyttöohjeet saatavilla myös: http://www.nettipalaute.fi/info/.

Käyttöohjeet

Palautteen antajan käyttöohjeet

1. Saat tilaisuuden järjestäjältä **sessiotunnuksen**. Syötä se etusivun sessiotunnus-kenttään ja paina Liity sessioon .
2. Palautteenantonäkymässä on joukko session ylläpitäjän ennalta luomia valmiita palautteita (ao. kuvassa merkitty punaisella). Näitä et voi itse lisätä tai muuttaa. Katso tilaisuuden alussa valmiit palautevaihtoehdot läpi, pidä palaute sivu avoinna esimerkiksi puhelimessasi, ja kun tilaisuuden aikana jokin palautteista tuntuu aiheelliselta, paina nappia.



3. Voit antaa myös vapaata palautetta. Sekin sidotaan antoaikaan.

- **Älä ylimieti yksittäistä palautetta - anna se, jos yhtään siltä tuntuu!** Oleellista palauttaan saajalle on monen ihmisen antamista palautteista muodostuva kokonaiskuva, eikä yksittäinen klikkaus.
- Vaikka palautteet sidotaan aina antoaikaan, muista, että ihminen tulkitsee palautteet. Oletus luultavasti onkin, että annetut palautteet koskevat aikaa hieman ennen palautteen antoa. Lisäksi vapaasta palautteesta varmasti ymmärretään, koskeeko se antohetkeä vai tilaisuutta yleisemmin.
- Samaa palautetta saa, ja kuuluukin, antaa useamman kerran samassa tilaisuudessa, jos se kuvaa tuntemuksiasi.

Palautteen kerääjän käyttöohjeet

Lyhyesti

1. **Ennen tilaisuutta.** Luo uusi sessiotyyppi ja valitse siihen sopivat palautetyypit.

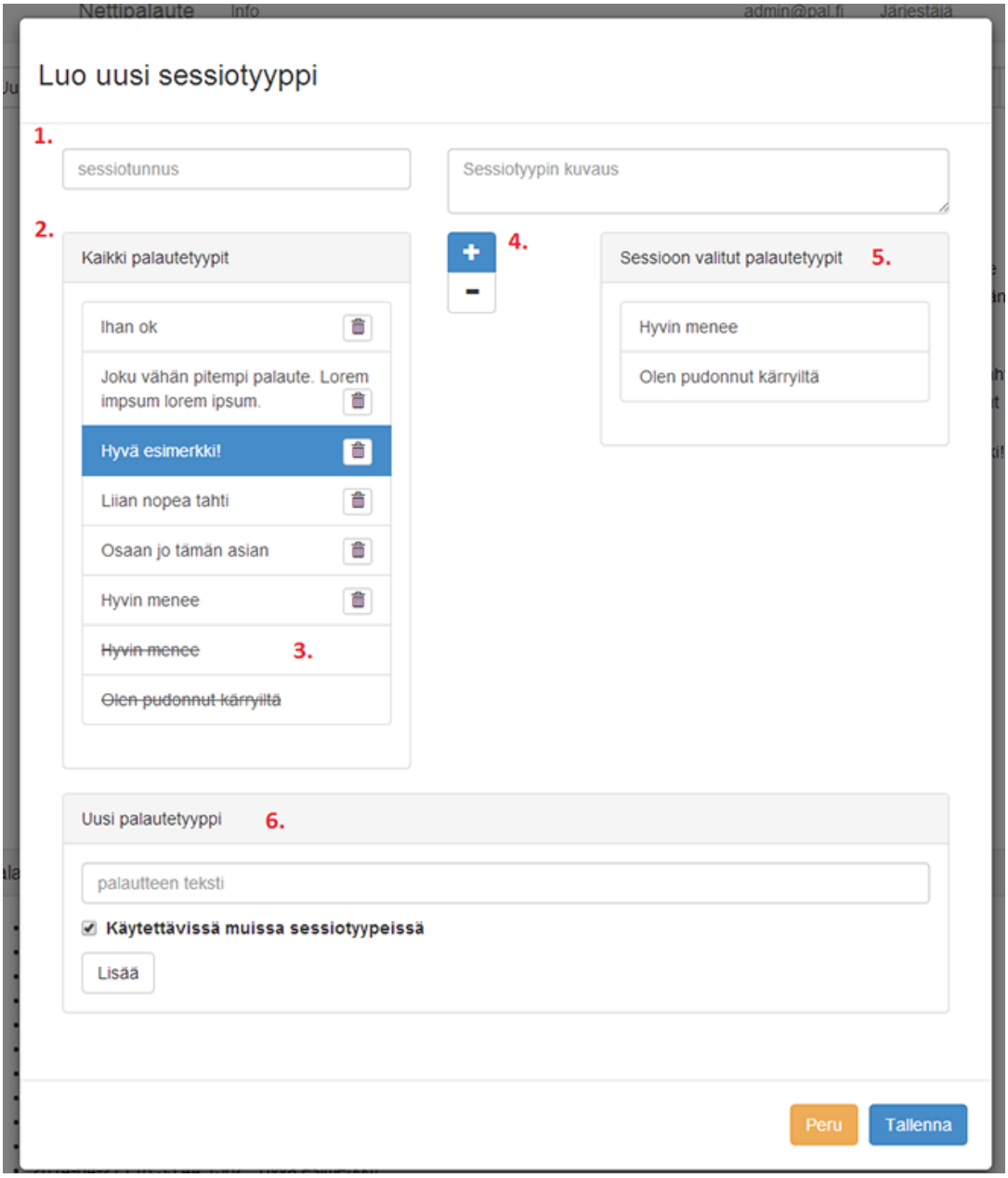
2. **Tilaisuuden alkaessa.** Anna sessiotunnus osallistujille.
3. Voit halutessasi seurata reaaliajassa palautteiden kertymistä tilaisuuden aikana tai selata saatuja palautteita kaikista menneistä tilaisuuksista.
4. Jos pidät saman tyyppistä tapahtumaa useamman kerran, muista, että sessiotyyppi tarvitsee luoda vain kerran ja sama sessiotunnus toimii jokaisella kerralla. Sessiotyyppi jaetaan automaattisesti yksittäisiin kertoihin, eli sessioihin.

Termeistä

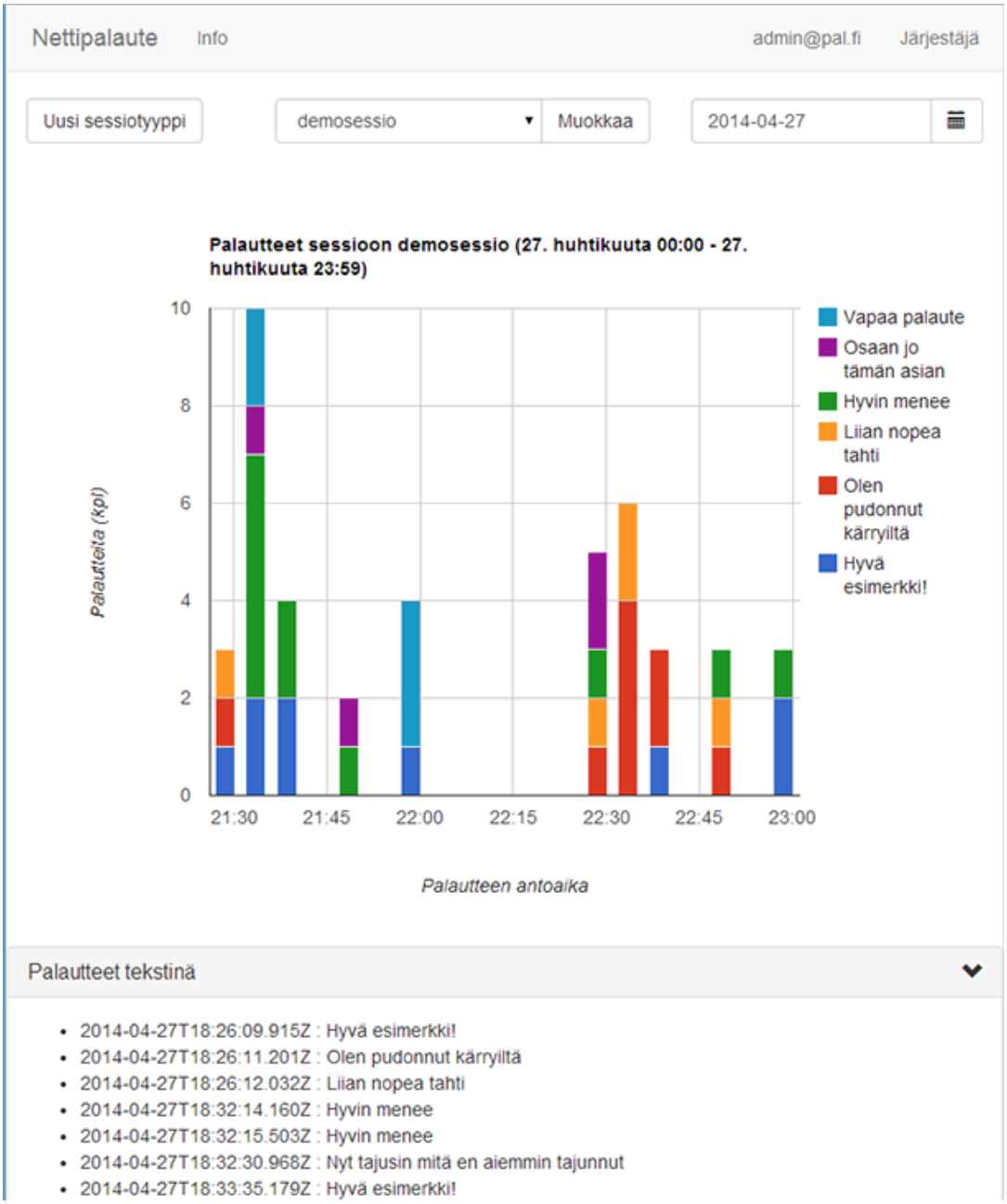
- **Sessiotyyppi** on esimerkiksi *Ohjelmointikurssi 1*. Nämä tulee luoda Järjestäjä-näkymässä.
- **Sessio** on sessiotyypin ilmentymä. Esimerkiksi *Ohjelmointikurssi 1:n luentokerta 24. huhtikuuta 2014 klo 14:00 - 16:00*. Sessiotyyppiin liittyvät sessiot luodaan aina automaattisesti, eikä niistä tarvitse huolehtia. Järjestelmän nykyisessä versiossa sessiotyypit pätkitään aina vuorokauden mittaisiin sessioihin.
- **Palautetyyppi** on esimerkiksi teksti "*Hyvää työtä*". Luotu palautetyyppi voi liittyä nollaan, yhteen tai useaan sessiotyyppiin.
- **Palaute, tai "annettu palaute"**, on palautteen antajan antama palaute, joka on *tyypiltään* esimerkiksi "*Hyvää työtä*". Keskenään samantyyppisiä annettuja palautteita voi olla mielivaltainen määrä. Esimerkiksi yhden tilaisuuden aikana moni palautteen antaja voi antaa palautteen "*Hyvää työtä*"

Pitkästi

1. Pääset järjestäjä-näkymään yläpalkin Järjestäjä-napista. Kirjaudu sisään tai rekisteröidy.
2. Luo uusi sessiotyyppi (ks. alla olevan kuvan numerointi)
    - 1. sessiotunnus on uniikki, lyhyt tunniste sessiotyypille, jota palautteen antajat käyttävät liittyäkseen sessioon. Hyvä sessiotunnus on esimerkiksi kurssin tunnus.
    - 2. Kaikki palautetyypit -listassa näkyy kaikki aiemmin (minkä tahansa sessiotyypin yhteydessä) luodut palatetyypit.
    - 3. Listassa viimeisenä näytetään palautetyypit, jotka on jo valitty käsiteltävään sessiotyyppiin.
    - 4. + -näppäimellä vasemmasta listasta lisätään valittu palautetyyppi oikeaan listaan. − -näppäimellä valittu palautetyyppi poistetaan oikeasta listasta.
    - 5. Listassa näytetään tähän sessiotyyppiin valitut palautetyypit. Palautteen antajat voivat antaa tässä sessiotyypissä näitä palautteita.
    - 6. Uuden palautetyypin luonti. Palautteen voi luoda niin, että se voidaan valita käyttöön myös muihin sessiotyyppeihin (näkyy muissa sessiotyypeissä listassa (kuvan numero 2.).



3. Tarkastele saatuja palautteita (ks. alla oleva kuva)
    - Valittuun sessiotyyppiin liittyvät sessiot näet kalenterivalinnalla.
    - Graafi näyttää annetut palautteet antoajan suhteen.
    - Kaikki graafissa näytettävät palautteet näytetään myös tekstimuodossa sivun alaosassa. Täältä näkee esimerkiksi pitemmät vapaat palautteet kätevämmin.


Nettipalaute
Info
admin@pal.fi
Järjestäjä
Uusi sessiotyyppi
demosessio
Muokkaa
2014-04-27
Palautteet sessioon demosessio (27. huhtikuuta 00:00 - 27. huhtikuuta 23:59)
Palautteita (kpl)
0
2
4
6
8
10
21:30
21:45
22:00
22:15
22:30
22:45
23:00
Palautteen antoaika
Vapaa palaute
Osaan jo tämän asian
Hyvin menee
Liian nopea tahti
Olen pudonnut kärryiltä
Hyvä esimerkki!
Palautteet tekstinä
2014-04-27T18:26:09.915Z : Hyvä esimerkki!
2014-04-27T18:26:11.201Z : Olen pudonnut kärryiltä
2014-04-27T18:26:12.032Z : Liian nopea tahti
2014-04-27T18:32:14.160Z : Hyvin menee
2014-04-27T18:32:15.503Z : Hyvin menee
2014-04-27T18:32:30.968Z : Nyt tajusin mitä en aiemmin tajunnut
2014-04-27T18:33:35.179Z : Hyvä esimerkki!

B. UML-SEKVENSSIKAAVIO YLLÄPITONÄKYMÄN AVAAMISESTA