TAMPEREEN TEKNILLINEN YLIOPISTO

JANNE RÄSÄNEN

YHDEN SIVUN SOVELLUS PROJEKTIEN KEHITYKSEN SEURANTATYÖKALUNA

Diplomityö

Tarkastaja: professori Kari Systä
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston kokouksessa
5. toukokuuta 2014

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
RÄSÄNEN, JANNE: Yhden sivun sovellus projektien kehityksen seurantatyökaluna
Diplomityö, 47 sivua, 0 liitesivua
Toukokuu 2014
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Kari Systä
Avainsanat: Single-page Application, SPA, Web Application, JavaScript, Backbone, Marionette, AngularJS, Meteor, Trello, CFD

Työssä vertailtiin kolmea nykyaikaista Single-page Applicationia (SPA): AngularJS -kehystä, Backbone + Marionette -yhdistelmää (BM) ja Meteor-kehystä. BM:n arkkitehtuuria ja käyttöä käytiin lävitse käytännön esimerkkiprojektin kautta.

Työssä toteutettiin työkalu, jonka avulla projektia vetävä henkilö saa vaivattomasti kuvan siitä, kuinka paljon ja miten projektin budjettia on kulutettu, mikä ennuste on budjetin kulumiselle, ja miten ominaisuuksien toteuttaminen etenee ajan funktiona.

Työ toteutettiin SPA:na joka integroitiin olemassaolevaan Business Intelligence -työkaluun käyttäen BM:a. BM todettiin hyväksi ratkaisuksi kyseessä olevalle pienehkölle projektille. Projektin testattavuus ja tuottavuus olivat hyvällä tasolla.

Yleisellä tasolla BM todettiin hyväksi ratkaisuksi projektille, jonka toteuttajat haluavat vapauden toteuttaa asioita omalla tavallaan, jossa tarvitaan kypsä luotettava kirjasto jonka osasia voi vapaasti vaihdella. Toisaalta BM on myös hyvä ratkaisu, mikäli toteuttajat haluavat oppia SPA:iden teon perusteet.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
RÄSÄNEN, JANNE: Single-Page Application as a project development tracking tool
Master of Science Thesis, 47 pages, 0 Appendix pages
May 2014
Major: Software engineering
Examiner: Professor Kari Systä
Keywords: Single-page Application, SPA, Web Application, JavaScript, Backbone, Marionette, AngularJS, Meteor, Trello, CFD

The thesis compares three modern Single-Page Application (SPA) technologies: AngularJS, Backbone + Marionette -combination (BM) and Meteor. The architecture and usage of BM is investigated more thoroughly using example project.

As part of the thesis a project tracking tool was implemented. Using the tool the project lead can with a single glance get an idea how much budget is left for the project, about when the budget will run out, how many features are still not finished and about how long finishing those features might take.

The tracking tool was implemented as an SPA which was integrated into existing Business Intelligence -tool using BM. BM was found to be a good solution for this smallish project. The testability and productivity of the project was good.

In general BM was found to be a good solution for a project where the developers want the freedom to do things their own way or where a mature and reliable library with possibility to easily change parts of is needed. On the other hand, BM is also a good solution if the developers want to learn the basics of SPA development.

ALKUSANAT

Tämä on Tampereen Teknillisen Yliopiston tietotekniikan laitokselle tehty diplomityö. Työn yhteydessä toteuttiin Tamperelaiselle ohjelmistoyritys Vincitille sisäinen työkalu projektien seurantaan.

Haluan kiittää työn ohjaajaa professori Kari Systää rakentavista kysymyksistä, jotka kerta kerran jälkeen ohjasivat minut oikealle polulle. Tämä työ ei olisi valmistunut myöskään ilman Vincitiltä tullutta apua. Haluan kiittää Pekkaa Virtasta ohjauksesta, diplomityön sisältöön liittyvistä neuvoista ja yleensäkin lähestymistavasta kirjoitustyöhön sekä aktiivisesta osallistumisesta ProLen budjettipuolen kehitykseen. Kiitokset Juha Riipille aktiivisesta osallistumisesta ProLen CFD-puolen kehitykseen. Kiitokset myös Juha Siposelle hyvästä katselmoinnista sekä neuvoista BITtiin ja sen kehitystyökaluihin liittyen. Vielä erittäin suuret kiitokset Vincitille sekä joustamisesta opintojen sitä vaatiessa että kahden viikon palkallisesta kirjoitusvapaasta, joka auttoi merkittävästi tämän työn loppuun saattamisessa.

Tampereella 13.5.2014

Janne Räsänen

SISÄLLYS

TERMIT JA NIIDEN MÄÄRITELMÄT

Ajax	Asynchronous JavaScript And XML. Joukko teknologioita joilla on parannettu Web-sovellusten käyttöliittymää ja käyttökokemusta.
Application Server	Sovelluspalvelin. Ohjelmisto joka pyörittää palvelimella palvelinohjelmistoa.
Backbone	Minimalistinen SPA-kirjasto.
BDD	Behavior Driven Development. Ohjelmistojen kehitystapa joka pohjautuu TDD:iin, mutta määrittelee että testattavan yksikön pitäisi olla ohjelmiston käytös (Behavior).
Business Intelligence Tool	Työkalu joka esittää olemassaolevaa tietoa muodossa, joka hyödyttää yhtiön liiketoimintaa.
BIT	Vincitin talon sisäisesti omaan käyttöön rakentama BI-työkalu.
CRUD	Create, Read, Update, Delete. Pysyväismuistin neljä perusoperaatiota.
CSS	Cascading Style Sheets. Kieli joka kuvailee dokumentin muotoilua ja ulkonäköä. Käytetään erityisesti laajalti Web-sivustoilla.
D3.js	Datalähtöinen JavaScript -kirjasto, joka tarjoaa työkaluja datan muunnoksiin. Käytetään laajalti web-sovelluksissa ja -sivustoissa luomaan SVG-pohjaisia visualisointeja datasta.
DOM	Document Object Model. Puuesitys HTML/XML -dokumentin elementeistä. Selaimet käyttävät DOM:ia kuvatessaan sivua sisäisesti, ja tätä puuta voi lukea ja muokata sivuston JavaScriptilla.
Gerrit	Git -pohjainen katselmointityökalu jota käytetään selaimen kautta.
Git	Hajautettu versionhallintaohjelmisto. Suosittu ja laajalti käytössä.
Istanbul	JavaScript -pohjainen testauskattavuustyökalu
JBehave	Java -pohjainen BDD-kehys. Sisältää tuen selkokielisten tarinoiden (Story) rakentamiseen.
Jasmine	JavaScript -pohjainen BDD-kehys.
JavaScript	Dynaaminen tulkattava kieli. Laajalti käytössä nykyaikaisissa web-sivustoissa ja -sovelluksissa.
JSON	JavaScript Object Notation, standardi jossa data

	esitetään avain-arvo -pareina. Alunperin lähtöisin JavaScript:ista mutta ei standardina mitenkään sidottu siihen.
Kanban	Lean-tyyppinen tuotannonohjausjärjestelmä jossa pyritään poistamaan prosessista kaikki turhat asiat. Käyttää kortteja työkuormasta signalointiin.
Kehys	Framework. Kokoelma uudelleenkäytettävää koodia. Yleensä tarkoitettu jonkin tietyn ongelman ratkaisemiseen. Yleensä kehysten toteuttajat olettavat niitä käytettävän melko tiukasti tietyllä tavalla.
Lean	Lean manufacturing, tapa parantaa prosessia siten, että eliminoidaan kaikki turha, eli kaikki mikä ei tuota asiakkaalle arvoa.
LocalStorage	HTML 5 mukana tullut pysyväissäilö, johon voi avain-arvo-pareina tallettaa tietoa, joka säilyy yli sessioiden ja selaimen uudelleenkäynnistymisen.
Marionette	Backbonen laajennos joka sisältää paljon rakenteita joita SPA:n rakentamisessa usein tarvitaan. Vähentää Backbonen käytössä tarvittavan boilerplate-koodin määrää.
Vahvasti ohjaileva kirjasto	Opinionated Software. Toteuttaja on olettanut että kirjastoa käytetään tiukasti jollain tietyllä tavalla. Tällöin tuottavuuden pitäisi olla suurempi.
MongoDB	NoSQL-pohjainen dokumenttitietokanta.
MVC	Model-View-Controller, suunnittelumalli graafisten käyttöliittymien toteutukseen. Kun ohjelmisto jaetaan kolmeen loogisesti erilaiseen osaan jotka kommunikoivat tiettyjen rajoitusten mukaisesti, koodista tulee modulaarisempaa, testattavampaa ja ylläpidettävämpää.
Node.js	Palvelinohjelmisto JavaScript -ohjelmistojen ajamista varten. Node.js -sovellukset käyttävät ei-estävää IO:ta ja asynkronisia tapahtumia. Lisäksi yksittäistä sovellusta varten ei luoda omaa säiettä, mikä mahdollistaa monien pienten sovellusten ajamisen rinnakkain.
NVD3	D3.js- ja JavaScript -pohjainen kaaviokirjasto.
OAuth	Autentikaatiomenetelmä jossa käyttötunnisteen (token) avulla käyttäjä voi antaa toiselle käyttäjälle tai palvelulle pääsyn jonkin toisen palvelun dataan kuitenkaan jakamatta varsinaisia tunnuksiaan palveluun. Lisätietoja http://tools.ietf.org/html/rfc5849
Project Lead	Vincitillä projektin vedosta vastuussa oleva henkilö.

ProLe	Project Leadin työkalu BIT:issä, jolla voi seurata projektin budjetin ja toteutettujen toimintojen etenemistä.
Responsiivinen suunnittelu	Mukautuva suunnittelu. Tapa suunnitella web-sovelluksia ja -sivustoja siten, että hyvinkin erilaisilla päätelaitteilla (desktop, tabletti, älypuhelin) saisi hyvän käyttökokemuksen. Sovelluksen pitäisi osata mm. mukautua eri kokoisilla näytöille.
REST	Tilaton ohjelmistoarkkitehtuurityyli, jonka tavoitteena on säilyttää erilaisten hajautettujen hypermediajärjestelmien yhteentoimivuus asettamalla rajoitteita niiden rajapinnoille.
Rich Internet Application (RIA)	Sivusto jonka käyttöliittymää on täydennetty (”rikastettu”) joko Ajax-teknologialla tai sitten jonkinlaisella selaimeen asennettavalla ulkoisella ohjelmistolla (esim. Flash tai Silverlight).
Single-page Application (SPA)	Web-sovellus joka pyörii JavaScript -pohjaisesti käyttäjän selaimessa. Ensimmäisen sivulatauksen jälkeen ei tehdä lisää sivulatauksia, vaan resursseja ladataan asynkronisesti taustalla. Pyrkii tuomaan Web-sovelluksen käyttökokemuksen yhä lähemmäs desktop-sovellusta.
Smoke test	Testi jossa tarkastetaan lähinnä, että sovelluksen toiminto onnistuu kaatumatta. Erittäin pintapuolinen ja nopea testi.
SVG	Scalable Vector Graphics. XML-pohjainen formaatti vektorigrafiikan esittämiseen. Laajalti käytössä myös web-sivustoissa ja -sovelluksissa.
Takaisinkutsufunktio	Callback. Funktio joka annetaan parametrina toiselle funktiolle. Toinen funktio sitten kutsuu takaisinkutsufunktiota jonkin asian tapahtuessa.
TDD	Test Driven Development. Tapa kehittää ohjelmistoja jossa ensin kirjoitetaan automatisoitu testi, sitten vasta koodi.
Trello	Ilmainen web-pohjainen projektinhallintatyökalu, jossa projektia (tai sen isoa osuutta) vastaa taulu. Kussakin taulussa on työvaiheita, ja vaiheiden alla on kortteja.Taulua voidaan käyttää Kanban-tauluna.
URL	Uniform Resource Locator
Vincit	Tamperelainen ohjelmistoalan yritys, joka tarjoaa räätälöityjä ohjelmistokehityspalveluja.

1 JOHDANTO

Perinteistä web-sivustoa selatessaan käyttäjä klikkaa linkkiä, odottaa että seuraava sivu resursseineen latautuu ja jatkaa selaamistaan. Ajan myötä käyttäjät ovat suhtautuneet yhä kriittisemmin sivujen latausaikoihin. Vuonna 2009 tehdyn tutkimuksen mukaan kolme neljäsosaa käyttäjistä odottaa sivun latautuvan alle kahdessa sekunnissa. 40% käyttäjistä lopettaa sivuston käytön mikäli odotusajat ovat jatkuvasti yli kolme sekuntia (Forrester 2009). Aikarajat ovat todennäköisesti nykypäivänä vielä tiukempia.

Vuodesta 1995 JavaScript on tehostanut ja siloittanut kokemusta. Nykyaikaisten web-sivustojen käyttäjäkokemus on yhä enenevissä määrin lähestynyt desktop-sovellusta, missä sivusto vastaa välittömästi käyttäjän syötteeseen. Mallia jossa sivusto alkuun lataa itse sovelluksen ja alkudatan, ja sen jälkeen kommunikoi asynkronisesti palvelimen kanssa taustalla ilman uusia sivulatauksia ja jossa iso osa koodista suoritetaan käyttäjän selaimessa, kutsutaan yhden sivun sovellukseksi (Single Page Application, SPA). SPA-teknologioilla saadaan minimoitua ladattavan datan määrää sekä käyttäjän kokemaa odottamisaikaa. Iso osa webin tulevaisuudesta tulee mitä todennäköisimmin rakentumaan SPA:n tai SPA:sta kehitettävän teknologian päälle.

Sulavan ja nopean käyttöliittymän mahdollistamisen lisäksi SPA:t on luontaista rakentaa responsiivisiksi. Lisäksi REST-rajapinnan päälle rakennettu SPA on luonnollisen modulaarinen ratkaisu, mikä auttaa ylläpidossa. On olemassa useita SPA -frameworkkeja ja -kirjastoja, jotka nopeuttavat kehitystyötä, auttavat tietoturvan suhteen ja helpottavat ylläpitovaihetta. Osaa näistä on käytetty useita vuosia vaativissa olosuhteissa. Uudemmat frameworkit taas yleensä tarjoavat innovatiivisempia ratkaisuja ja nopeampaa kehitystahtia.

Yhä isomman osan logiikasta siirtyessä selaimiin palvelimien tarvitsee tehdä yhä vähemmän työtä, ja niiden rooli muuttuu enemmän tietosäilöiksi ja staattisten resurssien tarjoajiksi. Tällöin niitä voidaan paremmin optimoida juuri tähän tarkoitukseen. Kaikki tämä tarkoittaa, että voidaan vähemmillä resursseilla palvella yhä useampaa käyttäjää ja on helpompi rakentaa paremmin skaalautuvia ohjelmistoja.

Työn alussa vertaillaan kolmea SPA- kehystä ja -kirjastoa, joista valitaan yksi. Tämän jälkeen esitellään valitun työkalun ominaisuuksia ja käyttöä tarkemmin esimerkkien kautta. Valitun työkalun ohjelmistotuotannollisia ominaisuuksia tarkastellaan erityisesti aiempien julkaisuiden nostamien haasteiden valossa. Työn lopussa analysoidaan sitä kuinka hyvin toisaalta SPA ja erityisesti valittu työkalu saavuttavat työlle asetettuja tavoitteita. Lisäksi pohditaan kuinka hyvin valittu työkalu

selviytyi tuottavuuden, laadun ja testattavuuden suhteen ja kuinka paljon se tarjosi uutta hyödyllistä opittavaa.

Työssä ratkaistiin se reaalimaailman ongelma, että Tamperelaisella ohjelmistoalan yrityksellä Vincitillä ei ollut riittävän hyviä ja helppoja työkaluja projektin budjetin ja ominaisuuksien valmistumisen seurantaan. Erityisesti pyrittiin siihen, että projektin vastuuhenkilö saisi vaivattomasti yhdellä silmäyksellä idean sekä projektin budjetin kulumisen että ominaisuuksien valmistumisen tilanteesta. Ohjelma toteutettiin valitulla SPA-työkalulla ja integroitiin osaksi olemassaolevaa Business Intelligence -työkalua. Työkalu hyödyttää kaikkein eniten Vincitillä Project Leadeja, joten sen nimeksi keksittiin ProLe. SPA oli ProLelle luonnollinen toteutustekniikka sen tarjoaman modulaarisuuden, käyttökokemuksen sulavuuden ja responsiivisuuden takia.

Luvussa 2 käydään lävitse SPA:iin liittyvää teoriaa sekä Vincitiin liittyvää taustaa josta syntyy osa työkalun vaatimuksista. Luvussa 3 esitellään kolme SPA -frameworkkia/kirjastoa joita harkittiin, ja perustellaan valinta. Luvussa 4 käydään lävitse Backbone/Marionette -yhdistelmän arkkitehtuuri. Luvussa 5 käsitellään Backbone/Marionette -yhdistelmää ohjelmistotuotannon haasteiden näkökulmasta. Luvussa 6 esitellään diplomityön teknisenä osana tehty ProLe -työkalu. Luvussa 7 arvioidaan kuinka hyvin työn tavoitteet saavutettiin. Luvussa 8 tehdään johtopäätöksiä arvioinnin tuloksista.

2 TEOREETTINEN TAUSTA

2.1 Web-arkkitehtuureista

2.1.1 Perinteinen web-arkkitehtuuri

Perinteinen web-arkkitehtuuri pohjautuu malliin, jossa käyttäjän navigoidessa uudelle sivulle käyttäjän selain lähettää pyynnön (HTTP Request) palvelimelle. Palvelin suorittaa palvelimen päässä koodia ja vastaa pyyntöön (HTTP Response) (Fielding et al. 1999). Vastauksen yhteydessä on HTML-dokumentti. Tämän jälkeen selain tekee mahdollisesti lisää pyyntöjä sivuun liittyviin resursseihin, kuten esimerkiksi kuviin, ja piirtää HTML -dokumentin.

Ajax (Asynchronous JavaScript and XML) yhdistettiin sivupohjaiseen arkkitehtuuriin pyrkien nopeampaan ja sujuvampaan käyttöliittymään. Kun sivupohjaisessa arkkitehtuurissa ladataan koko sivu ja sen resurssit kerralla, Ajax-teknologiaa käytettäessä voidaankin ladata vain tiettyjä resursseja käyttäen XMLHttpRequest -pyyntöjä. Tällöin esimerkiksi kuvagalleriassa käyttäjän navigoidessa seuraavaan kuvaan voidaankin ladata pelkästään seuraava kuva, ei kaikkia muita elementtejä ja resursseja jotka pysyvät joka tapauksessa samoina. Ajaxilla täydennettyä sivupohjaista arkkitehtuuria alettiin kutsumaan nimellä Rich Internet Application (RIA) (Farrel, Nezlek 2007). Ajax on laajalti käytössä nykypäivän Web-sivustoilla.

Representational State Transfer (REST) -arkkitehtuuri kuvaa ”hyvin käyttäytyvän” Web-palvelun, ja pyrkii täten standardisoinnin kautta helpottamaan toisaalta palvelun kehitystä mutta myös palveluiden yhdistämistä toisiinsa. REST-arkkitehtuurissa on resursseja joilla on tunniste (resource identifier). Tekemällä operaatioita (action) resurssista voi saada esityksen (representation) (Fielding, Taylor 2002). Kun REST:iä sovelletaan HTTP-protokollaan, tunnisteina toimivat URI:t ja operaationa toimivat HTTP:n toiminnot (mm. GET, PUT, UPDATE, DELETE). Tällöin esimerkiksi kun tehdään GET-operaatio kuvagallerian osoitteeseen `kuvat/42` saadaan vastaukseksi yksittäinen tietty kuva, ja vastaavasti PUT-operaatio osoitteeseen `kuvat` siirtää uuden kuvan palvelimelle. REST on nykyään laajalti käytössä Web-palveluissa.

2.1.2 Yhden sivun sovellus

Yhden sivun sovellus (SPA) on sovellus joka toimii selaimen kautta yhdellä nettisivun latauksella. SPA toimii latauksen jälkeen selaimessa suoritettavalla koodilla (yleensä JavaScript). Taustalla sovellus mahdollisesti hakee resursseja asynkronisesti palvelimelta tai tekee muutoksia palvelimelle (Mikowski, Powell 2014). SPA on käsitteenä RIA:n osajoukko. SPA on Ajax -teknologiaa vietynä yhä aiempaa pidemmälle.

SPA:n tavoitteena on tarjota käyttäjälle sujuva ja nopea käyttöliittymä, joka muistuttaa enemmän desktop-sovellusta kuin perinteistä webbisivustoa. Itse termi SPA luotiin vuonna 2005, mutta jossain muodossa SPA -toteutuksia on ollut olemassa ainakin vuodesta 2002 (Morris). Nykyisin SPA:t ovat yleisiä ja suosittuja. Hyviä esimerkkejä ovat monet Googlen palvelut, esimerkiksi Gmail (mail.google.com) tai Google Calendar (calendar.google.com).

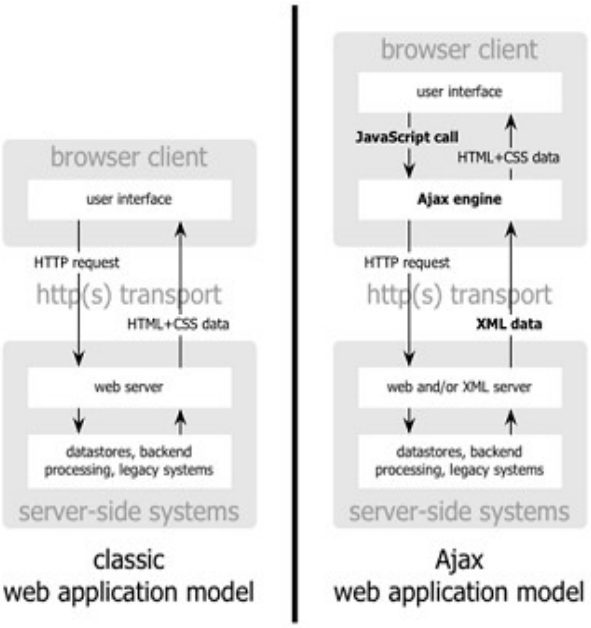
Ajan myötä on syntynyt monia JavaScript- kirjastoja ja -kehyksiä jotka nopeuttavat ja helpottavat SPA:n toteutustyötä. Kirjoitushetkellä kehitys on edelleen erittäin kiivasta ja uusia syntyy erittäin aktiivisesti, mutta toisaalta myös useita vuosia käytännössä hyviksi testattuja vaihtoehtoja on jo olemassa.

2.1.3 SPA vs perinteinen Web-arkkitehtuuri

Perinteisessä mallissa joudutaan aina sivunvaihdon yhteydessä tekemään palvelimelle pyyntö, odottamaan tuloksia ja mahdollisesti vielä tämän jälkeen lataamaan resursseja, mikä aiheuttaa lisää odottamista. Tämä aiheuttaa käyttömallin, jossa käyttäjä klikkaa linkkiä ja sitten odottaa sivun latautumista. Koska SPA pyörii käyttäjän selaimessa, ainakin käyttöliittymä kykenee vastaamaan välittömästi käyttäjän syötteeseen.

Riippuen sovelluksesta, SPA:ssa voidaan hakea resursseja etukäteen. Esim. kuvagalleriassa voitaisiin pitää aina tietty määrä tulevia kuvia muistissa, jolloin käyttäjän selatessa eteenpäin voidaan tulokset näyttää välittömästi. Mikäli kyseessä on sovellus missä tällainen etukäteen lataaminen ei ole järkevää toteuttaa (esim. ladattavaa dataa olisi liian paljon) ja resurssin lataaminen voidaan aloittaa vasta käyttäjän syötteen jälkeen, voidaan käyttöliittymässä kuitenkin välittömästi näyttää esim. latausikoni, ja näin saavuttaa välittömän vastauksen elämys.

Kuvassa 2.1 verrataan perinteisen web-sovelluksen ja SPA:n tekemiä pyyntöjä.



***Kuva 2.1**. HTTP-pyyntöjen eroavaisuudet (Farrel, Nezlek 2007)*

Koska SPA pyörii täysin selaimen päässä, on myös varsin luonnollista luoda sovelluksesta responsiivinen, jolloin se kykenee mukautumaan erilaisiin päätelaitteisiin ja erityisesti erikokoisiin näyttöihin. Logiikan suorituksen siirtyessä palvelimelta selaimelle, palvelimen rooli muuttuu enemmänkin tietosäilöksi, nykyään usein REST-arkkitehtuuriin nojautuen. Tällöin palvelin voi siirtyä tilattomaksi koneeksi, kun taas perinteisessä webbisivustossa palvelin yleensä säilyttää tilaa.

Kuten muutkin AJAX -pohjaiset web-sivustot SPA:t vaativat että käyttäjän selain tukee ja sallii JavaScriptin käytön. Mikäli tällaista tukea ei löydy ja sivusto halutaan silti pitää käytettävänä kyseisellä selaimella, joudutaan SPA:n rinnalle rakentamaan perinteinen webbisivusto. Tällaisen rakentaminen ja erityisesti ylläpitäminen voi käydä kalliiksi. Käytännössä kuitenkin nykyaikaiset selaimet tukevat JavaScriptia ja yleensä käyttäjät pitävät JavaScript -tuen mahdollistettuna selaimissaan.

Vaikka JavaScript -tuki löytyisikin selaimesta, JavaScript -tuen laadullinen taso saattaa vaihdella selainten välillä. Mikäli rajoittaudutaan ainoastaan suosittujen selainten uusimpiin versioihin, voi melko turvallisesti olettaa JavaScriptin toimivan hyvinkin samankaltaisesti kussakin selaimessa. Mikäli tarvitsee tukea vanhempia selaimia, erityisesti Internet Explorer (IE) 9 tai vanhempaa, voi vaatia paljonkin työtä, että sivusto näyttää ja käyttäytyy samalla tavalla eri selaimissa. Erilaiset JavaScript -kirjastot, esim. suosittu jQuery, auttavat samaa koodia toimimaan samalla tavalla erilaisissa selaimissa. Google Chrome Frame on plugin jonka voi asentaa IE:n versioihin 6 – 9, ja sen avulla voi käyttää sivustoja jotka ilman sitä eivät IE:llä toimisi.

SPA:n perinteisiä haasteita ovat toimiminen hakukoneiden, selaimen historian eli eteen/taakse -nappien (back/forward) kanssa ja mahdollisuus tehdä kirjanmerkkejä ja jakaa linkkejä SPA:n tiettyyn tilaan. Haasteena on siis se, että perinteisessä

webbisivustossa yksittäinen URL identifioi sivuston osan, kun taas SPA on määritelmänsä mukaan yhdellä sivulla pyörivä sovellus. Esimerkiksi kuvagallerian tapauksessa käyttäjä saattaisi haluta tallettaa linkin yksittäiseen gallerian kuvaan. Perinteinen ratkaisu kaikkiin näihin haasteisiin on säilyttää SPA yhdellä sivulla pyörivänä sovelluksena, mutta tuoda tila mukaan URL:iin hash fragmentilla, eli varsinaista sivun URL:ia seuraavalla osiolla. Esimerkiksi kuvagallerian tapauksessa joka on osoitteessa http://kuvagalleria.fi/galleria, käyttäjän siirtyessä katsomaan yksittäistä kuvaa, muutetaan JavaScriptilla osoitteeksi http://kuvagalleria.fi/galleria#yksittainen/42 (ilman että sivua ladataan uudestaan). Tätä ratkaisua kutsutaan syvälinkkaamiseksi (deep linking).

2.2 Muuta oleellista teoriaa

Seuraavissa luvuissa käydään lävitse Vincitiin taustoja, joista kumpuaa ProLeen liittyviä vaatimuksia. Lisäksi kerrotaan Trellosta, projektityökalusta josta tuotavasta tieto visualisoidaan aluekaaviona ominaisuuksien toteutuksen kehittymisestä.

2.2.1 Vincit

Vincit on kirjoitushetkellä 80 henkeä työllistävä Tamperelainen yritys, joka tekee ohjelmistoprojekteja. Vincit pyrkii tyytyväisiin asiakkaisiin ja tyytyväisiin työntekijöihin. Vincit pyrkii ketteryyteen, syleilemään muutosta. Vincit pyrkii työtapojen ja koko työkulttuurinsa suhteen tekemään asiat niin yksinkertaisesti kuin mahdollista, mutta ei yhtään sen yksinkertaisemmin. Näiden seikkojen pohjalta Vincit päätyi kehittämään oman BI -työkalun.

Vincitin projektit tehdään aina tilaustyönä, ts. Vincitillä ei ole mitään omia ohjelmistotuotteita. Projektien koot vaihtelevat yhden hengen kuukauden projektista usean hengen useita vuosia kestäviin projekteihin. Projektit ovat joko kiinteähintaisia tai sitten tehdyistä tunneista veloitetaan sovitulla hinnalla työntekijäkohtaisesti. Projekteja vedetään ketterästi ja nopeasti asiakkaalle arvoa tuottaen, esim. nopeasti protoilemalla. Projektityökalujen suhteen pyritään yksinkertaisuuteen ja tehokkuuteen ja käyttämään juuri parhaiten kutakin projektia palvelevia työkaluja.

Vincit panostaa työntekijöihinsä ja näiden tyytyväisyyteen. Vincit valittiin vuonna 2014 Great Place to Work -kisassa Suomen parhaimmaksi työpaikaksi. Tätä kautta Vincitille on tärkeää tarjota työntekijöidensä käyttöön mahdollisimman hyvät työkalut. Projektin seuranta on yksi tärkeä osa onnistuvaa ohjelmistoprojektia.

2.2.2 Business Intelligence Tool

Ponomarjovs (2013, s. 11 – 16) käy lävitse useita määritelmiä käsitteelle Business Intelligence (BI). Tiivistettynä BI-työkaluilla tarkoitetaan työkaluja jotka esittävät olemassaolevaa tietoa muodossa, joka hyödyttää yhtiön liiketoimintaa.

2.2.3 BIT

BIT on Vincitillä sisäisesti kehitetty BI-työkalu Vincitin sisäiseen käyttöön, joka otettiin käyttöön vuoden 2013 tammikuussa. BIT oli ProLen kehityksen alkaessa (marraskuu 2013) työkalu jonne syötettiin työtunteja, joiden pohjalta asiakkaita laskutettiin. Lisäksi sitä käytetään myös myyntivaiheessa olevien projektien seurantaan sekä työntekijöiden resurssoinnin seurantaan ja tätä kautta projekteihin saatavilla ihmisten selvittämiseen. BIT:tiä kehitetään aktiivisesti.

BIT on web-pohjainen työkalu, pääasiallisesti tarkoitettu desktop-käyttöön mutta toimii kohtuullisesti myös mobiililaitteilla. BIT pyörii ennenkaikkea Spring Frameworkin päällä, jonka käyttöliittymää on tuettu Ajax -ratkaisuilla ja osittain toteutettu puhtaana SPA:na Backbone + Marionetten kanssa.

BIT:issä kukin projekti sisältää 0..n työtehtävää. Mikäli asiakas haluaa tarkkaan eritellyn laskun, työtehtäviä luodaan esim. kullekin toteutettavalle ominaisuudelle. Kullakin työtehtävällä on oma tuntihintansa. Vincitin työntekijät merkitsevät tunteja projektien työtehtäville (käyttäen siihen erikseen kehitettyä desktop-sovellusta). Lisäksi työntekijöitä voi allokoida projekteihin eri määrillä eri viikoille, millä tarkoitetaan sitä että kiinnitetään jokin määrä työntekijän viikottaisista työtunneista projektille. Työn tekemisen yhteydessä BIT:tiin lisättiin myös tuki ostotilauksille (purchase order), jolloin kuhunkin projektiin voi liittyä 0..n ostotilausta, joilla kullakin on oma arvonsa ja alkamispäivämääränsä.

ProLen kehityksen alkaessa BIT:istä sai haettua projektin toteutuneen kokonaisbudjetin (joka laskettiin merkityistä työtunneista) ja työntekijäkohtaisesti kokonaismäärän projektiin tehdyille työtunneille. Vincitin Project Leadit hakivat nämä tiedot käsin, ja ainakin osa heistä vei näitä tietoja säännöllisesti taulukkolaskentaohjelmaan, minkä avulla he kykenivät seuraamaan budjetin kulumista pidemmällä aikavälillä.

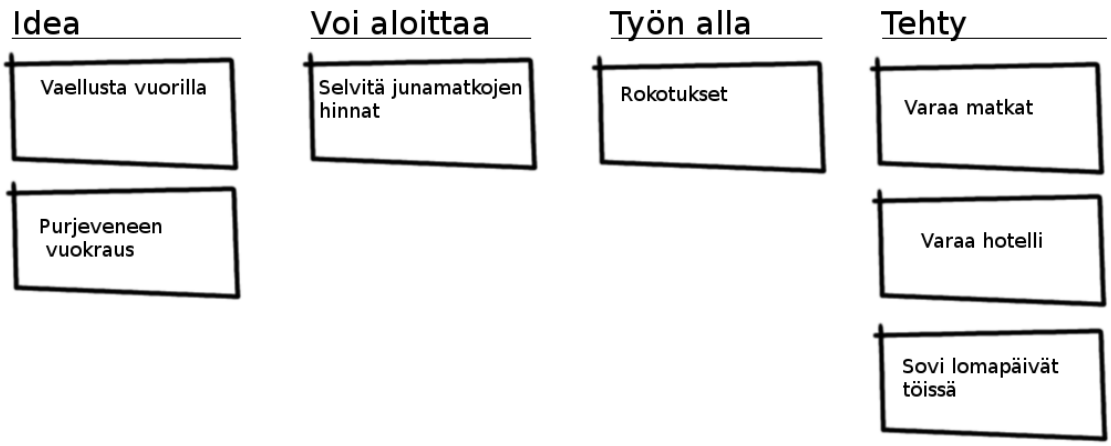
ProLen kehityksen alkaessa BIT:issä ei ollut integraatioita tietojen hakemiseen mistään projektityökalusta. BIT:in kehittäjillä on pitkän ajan tavoite jossa nykyinen BIT jäisi yhä enemmän pelkäksi REST -rajapinnaksi, tietosäilöksi ja toisaalta kokoelmaksi toisistaan arkkitehtuurillisesti erillisiä pienehköjä sovelluksia.

2.2.4 Trello

Trello on web-pohjainen projektityökalu, joka on vapaasti ja ilmaiseksi käytettävissä osoitteessa https://trello.com/. Trellon käyttöliittymä on yksinkertainen ja intuitiivinen.

Trellossa kutakin projektia kohden luodaan yksi taulu. Yksi kortti vastaa jotain projektissa toteutettavaa kokonaisuutta. Vaihe vastaa jotain tekemisen vaihetta. Kortteja voi siirrellä vaiheiden välillä, ja usein kortit siirtyvät enimmäkseen vasemmalla olevista vaiheista oikealla oleviin vaiheisiin samalla kun kokonaisuus valmistuu. Kuvassa 2.2

kuvataan loman valmistelun projektia. Vaiheita ovat Idea, Voi aloittaa, Työn alla ja Tehty. Kukin suorakulmio on yksi kortti.

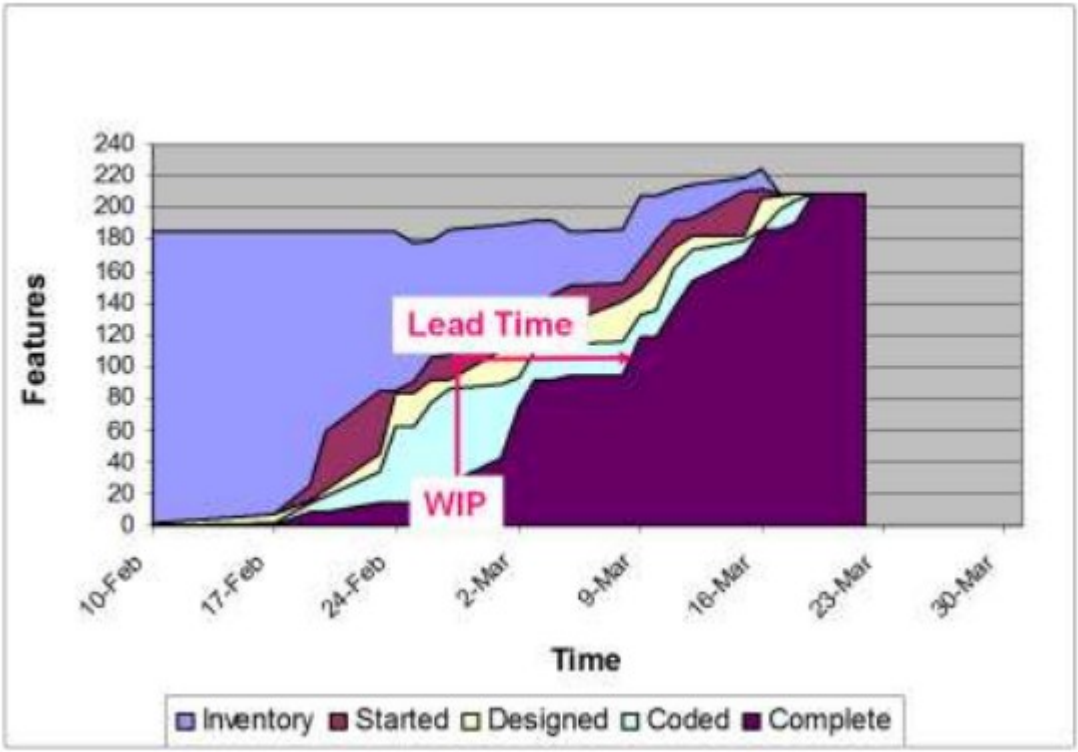


Kuva 2.2. *Yksinkertainen esimerkki Trello-taulusta*

Trello -taulu visualisoi työnteon etenemistä. Trelloa voi käyttää monenlaisten projektien etenemisen seurannan työkaluna, ei ainoastaan ketterien ohjelmistoprojektien. Trellossa useat käyttäjät voivat muokata tauluja, vaiheita ja kortteja ja muutokset näkyvät reaaliajassa muille käyttäjille. Trello on ilmainen työkalu (www.trello.com).

Kortteihin ei arvioida työmääriä eikä merkitä toteutuneita työmääriä. Mikäli haluaa että korttien perusteella voisi sanoa jotain jäljelläolevasta työmäärästä olisi hyvä, että korttien työmäärä keskimäärin vastaisi toisiaan, eli käytännössä eritellä erittäin isojen työmäärien kortit pienemmiksi tehtäviksi ja yhdistää erittäin pieniä tehtäviä suuremmiksi kokonaisuuksiksi. Vincitin projekteissa näin yleensä tapahtuu.

Cumulative Flow Diagram (CFD) kuvaa aluekaaviossa ajan funktiona korttien määriä eri vaiheissa. Tästä kaaviosta näkee miten kortit ovat edenneet ajan myötä eri vaiheiden lävitse. CFD kuvaa koko projektin työmäärän muutosta, ei yksittäisten asioiden etenemistä. Olettaen että projektin taulussa on vasemmalla puolen jonkinlainen ”työn alla” -vaihe ja oikealla puolen jonkinlainen ”tehty” -vaihe, CFD:stä näkee kullekin ajanhetkelle pystysuunnassa kesken olevien töiden määrän (Work In Progress, WIP) ja vaakasuunnassa keskimääräisen keston työn aloittamisesta sen loppuun saattamiseen (Lead Time). Kuvassa 2.3 on esimerkki CFD:stä johon on piirretty punaisilla viivoilla WIP ja Lead Time yhdelle tietylle päivälle projektin aikana (Anderson 2010).



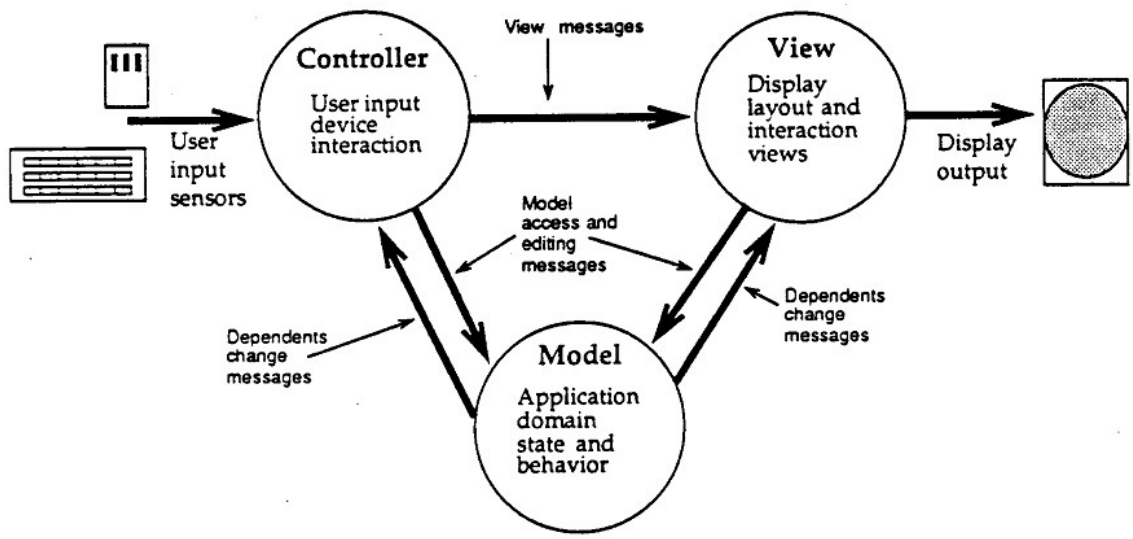
Kuva 2.3. *CFD -esimerkki (Anderson 2010)*

CFD:stä voi havaita työprosessin mahdolliset pullonkaulat, eli siis vaiheet jotka kestävät pitkään ja estävät myöhempien vaiheiden etenemisen. Lisäksi CFD:n perusteella voi tehdä projektioita siitä, koska nykyinen työmäärä on valmis. Andersonin mukaan, olettaen että käytetään Kanban -kehitysmenetelmää, vähentämällä samanaikaisesti keskeneräisen työn kokonaismäärää saadaan paitsi työvaiheita nopeammin valmiiksi, myös laadullisesti merkittävästi parempia tuloksia.

Trello tarjoaa APIn jonka kautta projektin korttien tiedot saadaan käyttöön. Autentikaatio hoidetaan OAuth 1.0 -menetelmällä. Menemättä teknisiin yksityiskohtiin, osaksi Web- palvelua voidaan liittää sivu jolla Trelloon autentikoitunutta käyttäjää pyydetään antamaan suostumuksensa siihen, että sivusto saa hakea Trellosta hänen tauluihinsa liittyviä tietoja. Mikäli käyttäjä antaa suostumuksensa, Trellon API antaa palvelulle avaimen (token) jota käyttäen palvelu voi hakea API:sta tietoja. Lisätietoja OAuthista on esimerkiksi RFC 5849 -dokumentissa (Hammer-Lahav 2010).

2.2.5 MVC ja sen johdannaiset

Model-View-Controller on arkkitehtuuri käyttöliittymien toteuttamiseen. Malli (Model) sisältää kaiken datan ja bisneslogiikan. Näkymässä (View) näytetään mallin data jossain muodossa. Käsittelijä (Controller) vastaanottaa käyttöliittymästä tulevat käskyt ja tekee niiden perusteella muutoksia malliin ja näkymään. Interaktiot MVC-mallin osien välillä on esitetty kuvassa 2.4 (Krasner, Pope 1988).



Kuva 2.4. *Interaktiot MVC-mallin osien välillä (Krasner, Pope 1988)*

MVC erottaa kätevästi kolme loogisesti erilaista käsittelyn tasoa. Kun tämä looginen jako on tehty, jokainen osa hoitaa vain sille kuuluvia vastuita ja osat interaktoivat MVC-mallin mukaisesti, toteutus helpottuu merkittävästi. Malli voidaan toteuttaa irrallaan muista osasista ja sitä voidaan uudelleenkäyttää. Näkymän ja käsittelijän välinen riippuvuus on suhteellisen pieni. Loogiset kokonaisuudet löytyvät sieltä mistä olettaisikin, eikä jonkin toiminnallisuuden toteuttavaa koodia joudu etsimään. MVC myöskin radikaalisti helpottaa ohjelmiston testaamista, koska koodi jakautuu loogisiin osasiin joilla on selkeä tarkoitus ja lisäksi muiden komponenttien korvaaminen tyngillä (stub) tai älykkäillä tyngillä (mock) helpottuu. Kaiken tämän lisäksi ylläpito helpottuu, koska uuden henkilön tullessa projektiin mukaan häntä odottaa tuttu suunnittelumalli. MVC tai jokin siitä periytetty arkkitehtuuri on laajalti käytössä suurimmassa osassa ohjelmistoja joissa on graafinen käyttöliittymä.

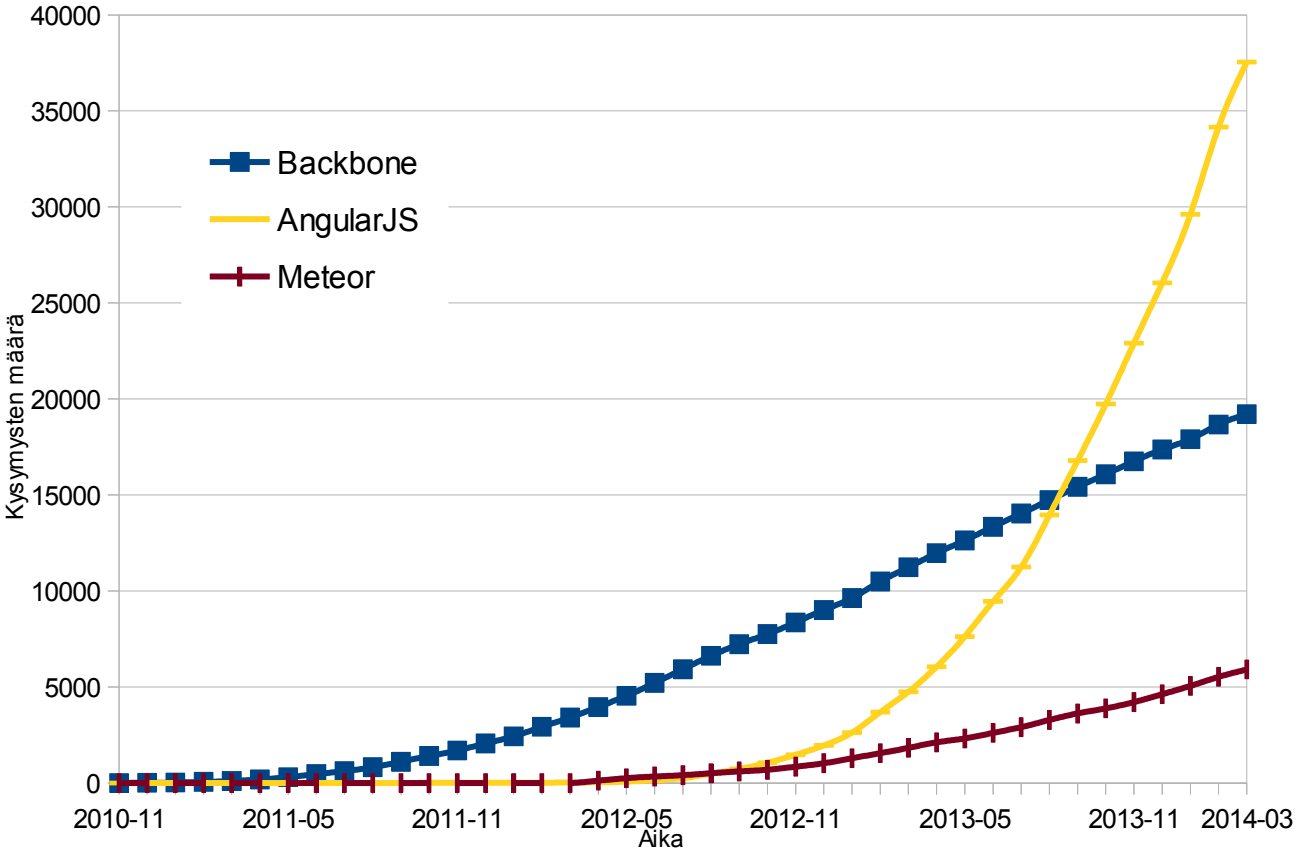
MVC:sta on kehitetty useita muita malleja, esim. suosiota saanut MVVM eli Model-View-ViewModel (Fowler 2004). Eri mallit ovat kilpailleet keskenään, ja tästä kilpailusta on myöskin noussut MV-Whatever-näkemys, minkä mukaan työkalu tarjoaa tuen malleille ja näkymille, ja työkalun käyttäjän harteille jätetään joko kokonaan tai osittain kolmannen osasen toteuttaminen. Tällöin käyttäjälle jää vapaus käyttää MVC-mallia tai mitä tahansa siitä periytettyä mallia (Minar 2012).

3 HARKITUT SPA KEHYKSET JA KIRJASTOT

Toteutuksen työkaluksi harkittiin kolmea nykyaikaista SPA-työkalua: kehyksiä AngularJS ja Meteor sekä Backbone+Marionette -kirjastojen yhdistelmää.

Kaikki harkitut työkalut sisältävät jonkinlaisen MVC-mallista periytetyn mallin jossa automaattisesti pidetään View ja Model synkronisoituina, jonkinlaisen REST-rajapinnan kanssa keskustelemisen, URL:ien liittämisen näkymiin (Routes) ja tuen syvälinkkaamiselle. Kaikki työkalut sisältävät jonkinlaisen tuen HTML templaateille, joita näkymät voivat hyödyntää. Kaikki työkalut sisältävät jonkinlaisen mekaniikan jolla DOM-puun elementtejä päivitetään Modelin muuttuessa.

Kaikki harkitut työkalut ovat avoimen lähdekoodin projekteja, suhteellisen suosittuja ja pyrkivät eri tavoilla nopeuttamaan kehityksen nopeutta ja minimoimaan tarvittavan lähdekoodin määrää. Kuvassa 3.1 näkyy stackoverflow -sivuston aiheeseen liittyvien kysymysten määrä eri aikoina, mikä kertoo työkalujen suosion muutoksesta.



Kuva 3.1: Työkalujen kysymysten määrä stackoverflow.com:ssa (stackexchange 2014)

Työkaluihin liittyy käsite siitä, kuinka vahvasti ohjaileva (opinionated) se on. Mikäli työkalu on vahvasti ohjaileva, työkalun tekijät ovat tarkoittaneet sitä käytettävän tiukasti jollain tietyllä tavalla. Mikäli työkalu on kevyesti ohjaileva, työkalua ei ole tiukasti tarkoitettu käytettävän jollain tietyllä tavalla. Yleensä vahvan ohjailevilla työkaluilla voi työkalun tekijöiden alkuperäisen tarkoituksen mukaisesti toimiessaan välttää työtä (Bedell 2006), mutta tarkoituksen ulkopuolella olevat asiat ovat merkittävästi vaikeampia tai joskus jopa mahdottomia toteuttaa. Usein kehykset ovat ohjailevampia kuin kirjastot.

Työkalun ohjailevuus on kiistakapula joka on lähivuosina usein tullut esille JS -kehittäjien keskuudessa. Mitään yleistä totuutta ei kuitenkaan ole olemassa, vaan työkalut on valittava sen mukaan, mikä sopii kyseiseen projektiin ja sen kehittäjille parhaiten.

3.1 AngularJS

AngularJS:n kehittäjien mukaan HTML on mainio kieli staattisten dokumenttien kuvaamiseen, mutta heikko kuvaamaan dynaamisia sovelluksia. Yleinen ratkaisu on korjata asiaa vain dynaamiselta puolelta JavaScriptin kanssa. AngularJS pyrkii tämän lisäksi hoitamaan asian vieläkin paremmin laajentamalla HTML:ää omilla direktiiveillään.

AngularJS:n periaatteet ovat tiivistettynä:

- Käytä deklaratiivista koodia käyttöliittymän ja ohjelman komponenttien yhdistämiseen ja imperatiivista koodia bisneslogiikkaan
- Pidä DOM:n manipulointi erillään sovelluslogiikasta. Tämä parantaa testattavuutta.
- Testaa ohjelmasi automatisoidusti ja kattavasti. Ohjelman rakenne vaikuttaa merkittävästi ohjelman testattavuuteen.
- Pidä client erillään palvelimesta. Tällöin kehitys voi edistyä rinnakkain ja ohjelmasta tulee modulaarisempi.
- AngularJS ohjaa suhteellisen tiukasti sovelluksen kehityksen kulkua.

AngularJS on voimakkaasti ohjaileva kehys, eli sitä on tarkoitettu käytettäväksi vain tietyllä tavalla. Paljon AngularJS:n toteutuksesta hoidetaan direktiivien laajentamalla HTML:llä. Komponenttien väliset yhteydet ilmaistaan deklaratiivisesti, minkä jälkeen AngularJS:n Dependency Injection pitää huolen yhteyksien luomisesta.

AngularJS on Googlen kehittämä kehys, jonka ensimmäinen versio julkaistiin jo vuonna 2009 ja 1.0 kesäkuussa 2012, mutta joka nousi suureen suosioon vasta vuonna 2013. Kehystä kehitetään aktiivisesti (angularjs.org 2014). AngularJS:llä voi halutessaan toteuttaa vain yksittäisen osan sivustostaan ilman että se vaikuttaa mitenkään muuhun sivustoon.

AngularJS:ää käsitteleviä kirjoja on alkanut ilmestymään. AngularJS:n käytössä on melko paljon uusia Angular-spesifisiä asioita omaksuttavana, jolloin aloittelijalla kestää tovi omaksua miten AngularJS:ää käytetään, ja miten sitä käytetään tekijöiden tarkoittamalla tavalla.

3.2 Meteor

Meteorin tavoite on kotisivujensa mukaan radikaalisti nopeuttaa sovellusten kehittämistä: prototyyppi päivässä parissa, oikea sovellus muutamassa viikossa. Meteor on ns. full stack framework eli se pyrkii sisältämään kaiken mitä web-sovelluksen kehittämiseen tarvitsee.

Meteorissa käytetään JavaScriptiä sekä selaimen että palvelimen päissä. Näin voidaan myös uudelleenkäyttää osaa koodista, erityisesti bisnes-logiikasta, kummassakin päässä. Meteor ajaa MongoDB- dokumenttitietokantaa sekä palvelimella että selaimessa, ja huolehtii automaattisesti näiden synkronisoinnista. Tämän ansiosta voi selaimen päässä kirjoittaa koodia kuin olisi palvelimella, ja mikäli jokin muutos ei menekään palvelimelta lävitse, tästä lähetetään ilmoitus.

Automaattinen synkronisointi myös vähentää latenssia, sillä kun tehdään muutos selaimen päässä, muutos näytetään välittömästi näkymässä. Mikäli palvelimelta tuleekin korjaus muutokseen, päivitetään silloin näkymää.

Palvelinpuolella koodi ajetaan Meteor-serverillä Node.js:n päällä. Tässä mielessä Meteor on myös täysi sovelluspalvelin (application server). Meteorin mukana tulee meteor-työkalu, jolla voi suorittaa monia kehitykseen liittyviä tehtäviä. Esimerkiksi sillä voi yhdellä komennolla luoda toteutuksesta paketin, minkä voi purkaa ja suorittaa missä tahansa missä on Node.js. Meteorissa on myös mielenkiintoinen hot code push -ominaisuus, jolla voi päivittää ajossa olevia sovelluksia lennossa.

Meteorista ei ole vielä julkaistu 1.0 -versiota, ja tietyt ominaisuudet ovat vielä kehitysvaiheessa. Esimerkiksi tällä hetkellä Meteor tukee ainoastaan MongoDB -kannan käyttöä. Tulevaisuudessa on tarkoitus laajentaa tuki muihinkin tietokantoihin (meteor.com 2014).

3.3 Backbone + Marionette

Backbone on yksinkertainen ja pieni kirjasto, joka tarjoaa mallit (Model), mallien kokoelmat (Collection), näkymät (View), keinot sitoa sovelluksen näkymät URL:eihin (Router) ja keinot hakea tai persistoida dataa palvelimelta tai palvelimelle (Sync) (Ashkenas 2014).

Backbone pyrkii olemaan minimalistinen kokoelma valmiita rakenteita joita SPA:n rakentaja voisi tarvita. Backbone on erittäin kevyesti ohjaileva (Ashkenas 2014). Backbone on tarkoituksellisen yksinkertainen, mutta toisaalta tarjoaa plugin-

arkkitehtuurin, jonka kautta sitä on helppoa laajentaa. Backboneen onkin kirjoitettu paljon laajennoksia.

Mallit ovat avain- arvo -parisäiliöitä. Kokoelmat sisältävät 0..n mallia ja hyödyllisen funktiokokoelman joka helpottaa niiden käyttöä. Mallit sidotaan näkymiin, ja mallin muuttuminen laukaisee tapahtuman (Event), minkä kautta näkymät saavat tiedon muuttuneesta tiedosta ja voivat päivittää itsensä. Backbone tukee REST- API:a, minkä kautta voidaan tehdä datan hakemista tai sen tallettamista (Ashkenas 2014).

Suosittu Underscore.js -kirjasto on myöskin Backbonen toteuttajan kirjoittama. Underscore.js sisältyy Backboneen, ja malleihin ja kokoelmiin on sisäänrakennettu suurin osa Underscore.js:n funktioista. Backbone tekee olettamuksen siitä, että sitä käytetään jonkinlaisen Model-View-Whatever mallin mukaan, mutta ei rajaa tai pyri ohjaamaan käyttäjäänsä johonkin tiettyyn ratkaisuun.

Eräs Backbonen laajennoksista on Marionette, joka sisältää usein toistuvien suunnittelumallien toteutuksia. Marionetten käyttäminen vähentää ns. boilerplate -koodin kirjoittamista kun toteutetaan monimutkaisia Backbone -sovelluksia.

Kirjastona Backbone on suosittu ja pitkään käytössä ollut. Backbonen ensimmäinen julkaisu oli lokakuussa 2010, ja 1.0 julkaistiin maaliskuussa 2013. Sekä Backbonesta että Marionettesta on kirjoitettu useita kirjoja, internetissä on paljon tutoriaaleja ja StackOverFlowissa on paljon kysymyksiä vastauksineen. Backbone + Marionette on myös yksinkertainen eikä vaadi paljoakaan uusien käsitteiden omaksumista ohjelmoijalta jolle web-ohjelmointi ja MVC ovat tuttuja.

3.4 Valittu framework ja perustelut

Toteutus päätettiin tehdä Backbone + Marionettella (BM). BM:ää on laajalti ja onnistuneesti käytetty Vincitissä käytetty useissa eri projekteissa. Tämä auttaa myös jonkun muun kuin alkuperäisen toteuttajan tekemässä ylläpidossa. BM on myöskin ”kypsä” kirjasto, kun taas AngularJS ja erityisesti Meteor ovat suhteellisen tuoreita tapauksia.

BM on helppo omaksua ja sen kanssa pääsee siksi nopeasti liikkeelle. BIT:issä on jo osia toteutettu BM:lla, mikä myöskin auttaa alkuun pääsemisessä.

Meteor on naimisissa MongoDB:n kanssa. Uuden kannan tuominen järjestelmään ja sen keskusteleminen olemassaolevien kantojen kanssa nähtiin työläänä. Meteor on myös erittäin tuore teknologia mistä ei ole julkaistu edes stabiilia versiota. Tämä tarkoittaa että kehityksessä saattaa helposti tulla vastaan kaikenlaisia yllättäviä ongelmia. Lisäksi ohjelmointirajapinnat saattavat muuttua paljonkin Meteorin päivittyessä, mikä käytännössä tarkoittaa lisätyötä.

Koska kyseessä on sisäinen työkalu jonka käyttäjämäärät ovat siksi varsin pieniä, kirjastojen kokojen erot tai työkalun tarjoama tuki skaalautuvuudelle eivät olleet kovinkaan relevantteja vertailukohtia. Koonkin suhteen BM oli tosin hyvä valinta,

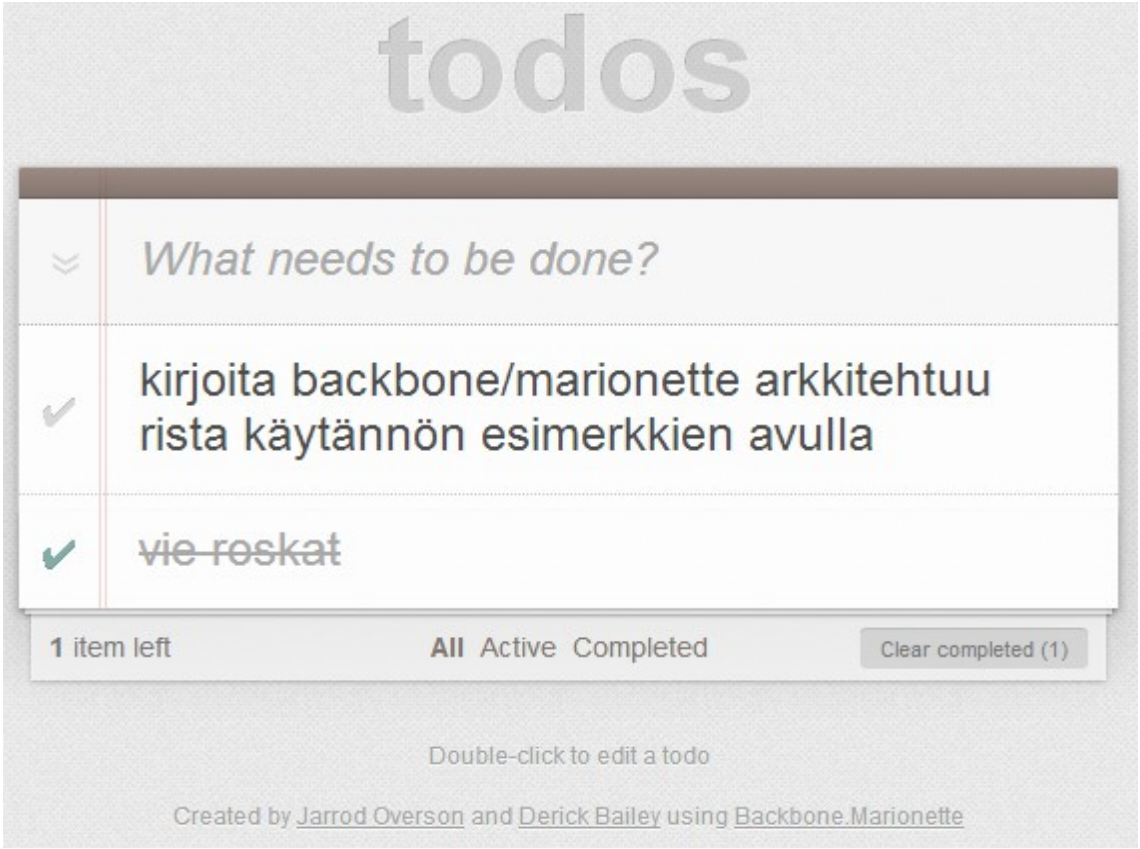
koska BM on jo osa BIT:tiä, eikä sen käyttäminen näin ollen suurenna lopullista JS-tiedostoa yhtään tavua.

Useimmiten kuultu Backbonen kritiikki kohdistuu siihen, että Backbonella joutuu kirjoittamaan ns. boilerplate-koodia eli ylimääräistä toistuvaa koodia. Marionette hyvinkin pitkälle korjaa tämän ongelman, jolloin tämä kritiikki on vähemmän relevanttia.

4 BACKBONE/MARIONETTE ARKKITEHTUURI

Monet ohjelmoijat oppivat esimerkkejä lukiessaan enemmän kuin pelkkää teoriaa lukiessaan (ja vielä enemmän itse koodia kirjoittaessaan kuin muiden esimerkkikoodia lukiessaan). Tämän havainnon pohjalta on syntynyt TodoMVC -projekti, jonka ideana on rakentaa yksinkertainen CRUD-operaatioita tekevä sovellus käyttäen erilaisia kehyksiä ja laittaa sitten hyvin kommentoitu koodi vapaasti saataville Githubiin (Overson, Bailey 2014). Tässä luvussa käydään lävitse esimerkkikoodin kanssa Backbone.Marionette TodoMVC -toteutus (Backbone.Marionette TodoMVC 2014). Mikäli muuta ei sanota, kaikki esimerkkikoodi on Oversonin ja Baileyn kirjoittamaa. Joidenkin rivien järjestystä on vaihdettu selkeyden vuoksi.

Esimerkkisovellus on web-sovellus, jossa on lista johon käyttäjä voi syöttää työtehtäviä, merkitä asioita tehdyiksi ja poistaa asioita listalta. Kuvassa 4.1 on sovelluksen käyttöliittymä.



***Kuva 4.1**: TodoMVC -sovelluksen käyttöliittymä (Overson, Bailey 2014)*

Kohdissa 4.1 - 4.8 käydään lävitse oleellisimmat ja välttämättömät osaset joista BM TodoMVC -sovellus rakennetaan ja miten ne suhtautuvat toisiinsa. Kohdan 4.9 alla käydään lävitse mekaniikkoja jotka joko helpottavat asioita tai tarjoavat lisätoiminnallisuutta.

4.1 Underscore.js ja periyttäminen

Underscore.js on mukana Backbonessa ja monet Backbonen luokat sisältävät suoran tuen Underscore.js:n funktioille. Monia BM:n luokkia käytetään periyttämällä Underscore.js:n extend-mekaniikalla:

```
var OmaMalli = Backbone.Model.extend();
```

Tällöin `OmaMalli` saa kaikki ominaisuudet mitä Backbone.Model -objektilla on. Backbonessa toistuva suunnittelumalli on mahdollisuus antaa `extend` -funktiolle useita optionaalisia parametreja, jotka vaikuttavat luokan toimintaan. Nämä parametrit voivat olla mitä tahansa JS-objekteja, mm. funktioita. Mikäli annetun parametrin nimi ei vastaa mitään BM:n odottamaa nimeä, lisätään parametrin niminen ominaisuus määriteltävään luokkaan.

4.2 Marionetten Application-luokka ja moduulit

BM-sovellus rakennetaan Marionetten Application-luokan ympärille.

```
var TodoMVC = new Backbone.Marionette.Application();
```

Tähän objektiin kiinnitetään myöhemmin kaikki sovellukseen liittyvät muut objektit. Tämän kautta myös hoidetaan sovelluksen käynnistäminen. Lisäksi `Application` laukaisee kolme tapahtumaa (event): ennen alustuksen alkamista, alustuksen päättymisen jälkeen ja ohjelman alkaessa.

Marionette tarjoaa mahdollisuuden jaotella koodi loogisiin kokonaisuuksiin käyttäen moduuleja. Moduuli luodaan `Application`- objektin kautta antamalla moduulin nimi:

```
TodoMVC.module('Todos', function (Todos, App, Backbone) {
…
});
```

Moduuli määritellään siis funktion sisällä, jota Marionette kutsuu. Tämän funktion ensimmäinen parametri on itse moduuli-objekti, toinen sovelluksen Application (eli TodoMVC -objekti) ja kolmas Backbone-kirjasto. Itse funktion sisälle kirjoitetaan moduulin toteutus. Toteutus on jätetty pois listauksesta, mutta ensimmäinen moduulin sisälle kuuluva koodilistaus on kohdassa 4.3. Koska moduulin määrittely on funktion sisällä, voi siellä määritellä paikallisia muuttujia jotka eivät näy moduulin ulkopuolelle. Kiinnittämällä objekteja `Todo` -parametriin saa ne näkymään moduulin ulkopuolelle. Kutsu `TodoMVC.module('Todos')` palauttaa aina saman moduulin, ja tämän kautta voi halutessaan jakaa saman moduulin toteutuksen useisiin eri tiedostoihin. Esimerkki-

sovelluksessa luodaan moduulit malleille, näkymille, käsittelijöille ja Layoutille (mistä myöhemmin lisää).

4.3 Mallit

Backbonen mallit (Model) ovat MVC-mallin malleja. Kuhunkin malliin liittyy joukko kenttiä, jotka toteutetaan avain-arvo -pareina. Backbone sisältää myös kokoelmat (Collection) jotka ovat mallien kokoelmia. Allaolevassa koodissa sekä `Todo` että `TodoList` ovat luokkamäärittelyitä, joiden perusteella voi instantioida objekteja.

`Todos` -moduulin sisällä määritellään malli yksittäiselle työtehtävälle ja usean työtehtävän kokoelma. Yksittäinen työtehtävä määritetään periyttämällä Backbonen `Model` -luokasta `Todos`-moduulin sisälle:

```
Todos.Todo = Backbone.Model.extend({
    defaults: {
           title: '',
           completed: false,
           created: 0
    },

    initialize: function () {
           if (this.isNew()) {
                  this.set('created', Date.now());
           }
    },

    toggle: function () {
           return this.set('completed', !this.isCompleted());
    },

    isCompleted: function () {
           return this.get('completed');
    }
});
```

Mikäli luodaan uusi työtehtävä, mutta jokaiselle `default` -kohdassa listatulle ominaisuudelle ei ole annettu arvoa, kyseinen ominaisuus saa arvokseen mitä `defaults` -kohdassa on määritetty.

`Initialize` -funktiota kutsutaan aina kun malli luodaan. Avain-arvo-parin arvo saadaan `get` -funktiolla ja avaimen arvo asetetaan `set` -funktiolla. Kummassakin tapauksessa avain annetaan merkkijonona. `toggle` ja `isCompleted` ovat bisneslogiikkaa eli funktioita joita voidaan määrittelyn jälkeen kutsua mille tahansa `Todo` -objektille.

Backbonen `Collection`-luokka sisältää valmiiksi määriteltyjä ominaisuuksia jotka helpottavat kokoelmien käyttöä. Edelleen `Todos`-moduulin sisällä työtehtävien kokoelma määritetään periyttämällä:

```
Todos.TodoList = Backbone.Collection.extend({
    model: Todos.Todo,

    comparator: function (todo) {
          return todo.get('created');
    },


    _isCompleted: function (todo) {
          return todo.isCompleted();
    },


    getCompleted: function () {
          return this.filter(this._isCompleted);
    },

    getActive: function () {
          return this.reject(this._isCompleted);
    }
});
```

`model` -parametrin kautta määritetään yksittäinen malli mitä tämä kyseinen kokoelma sisältää. Kokoelmalle määritellään `comparator` -parametrin kautta vertailufunktio, joka ottaa sisään yhden mallin ja antaa tuloksena objektin, jolle voidaan kutsua `valueOf` -funktiota, joka antaa arvon jota voidaan vertailla (suurempi kuin- ja pienempi kuin -operaattoreilla) muihin vertailufunktion palauttamien arvojen `valueOf` -funktioiden tuotoksiin. Mikäli kokoelmalle annetaan vertailufunktio, Backbone pitää tämän jälkeen kokoelman automaattisesti järjestyksessä järjestettynä vertailufunktion tuloksen perusteella.

`_isCompleted` -funktiota käytetään parametrina `getCompleted`- ja `getActive` -funktioissa. `Collection.filter` palauttaa taulukkona kaikki kyseisen kokoelman mallit joille parametrina annettu funktio palauttaa jotain mitä JavaScript pitää totena. Vastaavasti `Collection.reject` palauttaa kaikki mallit joille parametrina annettu funktio palauttaa jotain mitä JavaScript ei pidä totena. `reject` on siis `filter`:in vastakohta.

Sekä `filter` että `reject` ovat Underscore.js:ssä määriteltyjä funktioita. Ne ja monet muut Underscore.js:n funktioista ovat käytettävissä suoraan Backbonen Model- ja Collection -luokkien kanssa. Esim. kokoelmille löytyy `forEach`, `reduce`, `map`, `find` jne. Nämä monipuoliset funktiot helpottavat ja yksinkertaistavat mallien ja kokoelmien käyttöä.

4.4 Näkymät

Backbone sisältää melko kevyen tuen näkymille (View), mutta Marionette laajentaa tätä merkittävästi sisältäen tuen yleisille tapauksille. Kaikille näkymille yhteinen koodi on `View`-luokassa, mutta kyseistä luokkaa ei ole tarkoitus käyttää suoraan. Sen sijaan `ItemView` tarjoaa toteutusta yksittäisen asian esittämiseen, `CollectionView` kokoelman esittämiseen ja `CompositeView` sopii esim. tilanteeseen missä on esitettävä puu-muodossa sekä vanhempi että sen lapset. `ItemView` on tarkoitettu yksittäisen mallin

esittämiseen (mutta sillä voi esittää myös kokoelman) kun taas `CollectionView` ja `CompositeView` näyttävät kokoelman käyttäen tietyntyyppisiä `ItemView`:ejä kokoelman yksittäisten alkioiden näyttämiseen.

Kaikille näkymille yhteistä on tapahtumien kautta kommunikointi, esim. mallin muuttuessa päivitetään näkymää viestimällä asiasta tapahtumista. BM lähettää nämä viestit itse automaattisesti ja joissain tapauksissa myös kuuntelee ja reagoi niihin itse automaattisesti. Näkymillä on myös omat takaisinkutsufunktiot (callback) sidottuina sovelluksen elinkaaren eri hetkiin. Tapahtumia käsitellään tarkemmin kohdassa 4.8.

TodoMVC:n näkymät on määritetty oman moduulinsa sisälle:

```
  TodoMVC.module('TodoList.Views', function (Views, App, Backbone,
Marionette, $) {
  …
  });
```

Näkymät määritetään periyttämällä Marionetten näkymistä.

4.4.1 Yksittäisen työtehtävän näkymä

Yksittäiselle työtehtävälle määritetään näkymä periyttämällä `ItemView` -luokasta:

```
  Views.ItemView = Marionette.ItemView.extend({
      template: '#template-todoItemView',
```

`template` -parametrilla identifioidaan näkymän käyttämä html -templaatti. Tässä identifiointi on tehty käyttäen JQuery -valitsinta (selector). Vaihtoehtoisesti `template`-parametrina olisi voinut antaa funktion joka palauttaa suoraan halutun templaatin. Underscore.js sisältää myös oman templaatti-järjestelmän jota voi halutessaan suoraan käyttää näkymien kanssa, tai sitten voi vapaasti valita miltei minkä tahansa templatointikirjaston. Yksittäisen työtehtävän näkymä käyttää seuraavaa Underscore.js:n templaattia, joka on määritelty omassa html-tiedostossaan:

```
  <script type="text/html" id="template-todoItemView">
      <div class="view">
            <input class="toggle" type="checkbox"
            <% if (completed) { %>checked<% } %> >
            <label><%- title %></label>
            <button class="destroy"></button>
      </div>
      <input class="edit" value="<%- title %>">
</script>
```

Prosenttimerkkien sisällä olevat kohdat korvataan käyttäen mallin arvoja annetuilla avaimilla. Loppu on puhdasta HTML:ää. `Id`:n avulla BM osaa löytää oikean templaatin oikealle näkymälle.

Lisäksi näkymille voi antaa luomisen yhteydessä `ui`-parametrin, jolle annetaan hasheina jQuery-selectoreita, jotka sitten sidotaan ui-muuttujaan. TodoMVC:ssä määritetään

```
ui: {
        edit: '.edit'
},
```

jonka jälkeen näkymän koodissa `ui.edit` viittaa suoraan DOM:in elementtiin jonka `.edit` valitsee (eli HTML-templaatissa esiintyvän input- elementin). Tämän avulla jQuery -selektorit voidaan koota yhteen ja samaan paikkaan, sen sijaan että niitä olisi siellä täällä muun koodin seassa.

Näkymä hoitaa syötteen vastaanottamisen käyttöliittymältä sitomalla funktioita joita kutsutaan DOM tapahtumien tapahtuessa. TodoMVC:ssä annetaan näkymän luomisen yhteydessä events-parametri:

```
events: {
        'click .destroy': 'destroy',
        'click .toggle': 'toggle'
        'dblclick label': 'onEditClick',
        'keydown .edit': 'onEditKeypress',
        'focusout .edit': 'onEditFocusout',
},
```

Vasemmalle on tapahtuma, oikealla on funktion nimi jota tapahtuman tapahtuessa kutsutaan. Funktio saa parametrinaan näkymän ja näkymän mallin tai kokoelman, mikäli sellainen oli asetettu. Tämä helpottaa toteutusta ja myös mahdollistaa sen, että yksi käsittelijäfunktio voi käsitellä näppärästi useita eri näkymiä.

TodoMVC:n tapahtumien käsittelijäfunktiot on määritelty seuraavasti:

```
destroy: function () {
        this.model.destroy();
},
```

Kun siis käyttäjä klikkaa destroy -nappia (määritelty aiemmassa templaatissa), tuhotaan tämän näkymän malli (eli siis yksittäinen työtehtävä, luokkaa `Todo`). Tämä aiheuttaa tapahtuman (event) jonka kokoelma ottaa kiinni, ja poistaa tämän seurauksena mallin kokoelmasta. Tämän seurauksena myös näkymä poistetaan, jolloin käyttäjän ruudulta katoaa työtehtävä jonka tuhoamisnappia hän painoi.

Käyttäjän klikatessa toggle-nappia työtehtävä merkitään tehdyksi tai ei-tehdyksi, riippuen työtehtävän nykyisestä tilasta:

```
toggle: function () {
        this.model.toggle().save();
},
```

Kun käyttäjä tuplaklikkaa työtehtävän tekstiä, siirrytään työtehtävän tekstin editointimoodiin:

```
onEditClick: function () {
        this.$el.addClass('editing');
        this.ui.edit.focus();
        this.ui.edit.val(this.ui.edit.val());
},
```

Koodissa `this.$el` on Backbonen automaattisesti luoma muuttuja, joka viittaa jQuery -objektiin joka viittaa näkymän elementtiin. Koodissa annetaan elementille luokka, joka johtaa CSS-määrittelyjen kautta ruudulla olevan tekstin ulkonäön

muuttumiseen. `this.ui.edit` viittaa aiemmin määriteltyyn ui -komponenttiin, ja sitä kautta DOM -elementtiin.

Kun käyttäjä enter-näppäintä siirrytään editointilaan. Kun käyttäjä painaa ESC-näppäintä, siirrytään pois editointitilasta:

```
onEditKeypress: function (e) {
	var ENTER_KEY = 13, ESC_KEY = 27;
	if (e.which === ENTER_KEY) {
		this.onEditFocusout();
		return;
	}

	if (e.which === ESC_KEY) {
		this.ui.edit.val(this.model.get('title'));
		this.$el.removeClass('editing');
	}
},
```

Editointitilan menettäessä fokuksen (esmes käyttäjän klikatessa jotain eri osaa sivustosta) lopetetaan editointitila poistamalla CSS-luokka ja asetetaan malliin tekstikentän nykyinen arvo:

```
onEditFocusout: function () {
	var todoText = this.ui.edit.val().trim();
	if (todoText) {
		this.model.set('title', todoText).save();
		this.$el.removeClass('editing');
	} else {
		this.destroy();
	}
},
```

Kun malli muuttuu, tästä lähtee automaattisesti BM:n hoitamana viesti näkymälle. Alla olevassa koodin seurauksena mallin muuttuessa kutsutaan uudelleenpiirtofunktiota:

```
modelEvents: {
	'change': 'render'
},
```

Tällöin mallin muuttuessa näkymä piirretään uudestaan, jolloin DOM:n elementti päivittyy. `modelEvents` -parametrilla voi sitoa funktioita näkymän oman mallin tapahtumiin.

Näkymän `onRender` -funktiota kutsutaan piirron yhteydessä. Seuraava koodi uudelleenasettaa mallista riippuen oikean CSS -luokan DOM-elementille:

```
onRender: function () {
	this.$el.removeClass('active completed');
	if (this.model.get('completed')) {
		this.$el.addClass('completed');
	} else {
		this.$el.addClass('active');
	}
},
```

CSS-luokan asettaminen aiheuttaa muutoksen ulkonäössä.

4.4.2 Työtehtävien kokoelman näkymä

Ylläolevassa koodissa on määritelty näkymä yksittäisen työtehtävän näyttämiselle, eli siis kuvan 4.1 yksittäiselle riville. Näistä yksittäisistä riveistä koostetaan lista käyttämällä Marionetten `CompositeView:iä` (edelleen `TodoList.Views` -moduulin sisällä):

```
Views.ListView = Backbone.Marionette.CompositeView.extend({
    template: '#template-todoListCompositeView',
```

Kuten aiemmallekin näkymälle, myös tälle näkymälle määritellään viittaus templaattiin käyttäen `template` -parametria. Templaatti on määritelty HTML -tiedostoon seuraavasti:

```
<script type="text/html" id="template-todoListCompositeView">
    <input id="toggle-all" type="checkbox">
    <label for="toggle-all">Mark all as complete</label>
    <ul id="todo-list"></ul>
</script>
```

Näkymälle annetaan parametrina `itemViewContainer-` parametri, joka on myös jQuery-valitsin. Sen avulla näkymä tietää minkä elementin alle yksittäisistä alkioista syntyvä sisältö sijoitetaan:

```
    itemViewContainer: '#todo-list',
```

`ItemView` -parametrilla määritetään että jokaista kokoelman mallia kohden luodaan yksi `Views.ItemView` -näkymä (joka määritettiin yllä):

```
    itemView: Views.ItemView,
```

Kun siis `Views.ListView` piirretään, luodaan ensin `input-`, `label-` ja `ul`-elementit, minkä jälkeen jokaista työtehtävää kohden piirretään työtehtävän piirron tulokset.

Käyttöliittymä-elementit määritetään kuten edelläkin käyttäen ui-parametria:

```
    ui: {
           toggle: '#toggle-all'
    },
```

Valitsin `toggle-all` viittaa templaatissa määritettyyn valintaruutuun (checkbox). Käyttäjän vaihtaessa valintaruudun tilaa halutaaan kutsua funktiota, ja se hoidetaan taas `events` -parametrilla:

```
	events: {
		'click #toggle-all': 'onToggleAllClick'
	},

	onToggleAllClick: function (e) {
		var isChecked = e.currentTarget.checked;
		this.collection.each(function (todo) {
			todo.save({ 'completed': isChecked });
		});
	}
```

Käsittelyfunktio saa parametrinaan elementin, jolta haetaan tieto siitä onko valintaruutu rastitettu vai ei. Tämän perusteella muutetaan jokaisen kokoelman alkion tila joko valmiiksi tai ei-valmiiksi.

Aivan kuten `ItemView`:ille voi antaa parametrina malliin liittyvän `modelEvents` -parametrin, `CompositeView`:ille voi antaa paramaterina kokoelmaan liittyvän `collectionEvents` -parametrin:

```
	collectionEvents: {
		'all': 'update'
	},
```

Tässä tapauksessa kaikkiin kokoelmaan liittyviin viesteihin reagoidaan `update`-funktiolla. Myös piirron yhteydessä kutsutaan samaa funktiota:

```
	onRender: function () {
		this.update();
	},
```

Joka on vihdoin määritelty:

```
	update: function () {
		function reduceCompleted(left, right) {
			return left && right.get('completed');
		}
		var allCompleted = this.collection.reduce(reduceCompleted,
true);
		this.ui.toggle.prop('checked', allCompleted);
		this.$el.parent().toggle(!!this.collection.length);
	},
```

Viimeisellä rivillä asetetaan tämän näkymän elementin isä-elementti joko näkyväksi tai piilotetuksi riippuen siitä onko kokoelmassa yhtään elementtejä tai ei.

4.4.3 Ylätunniste ja alatunniste

Vastaavasti jo esille tuotuja mekaniikkoja käyttäen luodaan näkymät käyttöliittymän ylätunnisteella ja alatunnisteelle laajentaen Marionetten `ItemView` -luokkaa.

Alatunnisteeseen liittyy muutama uusi temppu, joita käsitellään tämän luvun loppupuolella omien otsikoidensa alla.

4.5 Region

Alueita (Region) määritellään jQuery -valitsimilla. Myöhemmin alueisiin voidaan sitoa näkymään näkymiä.

```
TodoMVC.addRegions({
    header: '#header',
    main: '#main',
    footer: '#footer'
});
```

Koodissa `TodoMVC` on itse sovellus. Koodin suorittamisen jälkeen `TodoMVC.header`, `TodoMVC.main` ja `TodoMVC.footer` ovat olemassa alueina.

4.6 Käsittelijä

TodoMVC -sovellukselle päätettiin rakentaa käsittelijä (Controller) seuraten perinteistä MVC-mallia. `TodoList` -moduulin sisälle määritettiin

```
TodoList.Controller = function () {
    this.todoList = new App.Todos.TodoList();
};
```

Koodissa luodaan uusi kokoelma työtehtäviä ja sidotaan viittaus siihen käsittelijään. Käsittelijälle lisätään vielä muutama uusi funktio käyttäen Underscore.js:n `extend`-funktiota:

```
_.extend(TodoList.Controller.prototype, {
    start: function () {
           this.showHeader(this.todoList);
           this.showFooter(this.todoList);
           this.showTodoList(this.todoList);
           this.todoList.fetch();
    },
```

`start` -funktiota kutsutaan myöhemmin, kun käsittelijä käynnistetään. `todoList.fetch()` hakee kokoelman pysyväismuistista kuten palvelimelta tai HTML5:n LocalStoragesta. Pysyväismuistiin tallettamista käsitellään tarkemmin kohdassa 4.9.1. Alla on määritelty yksi kutsutuista `show`-funktioista:

```
    showTodoList: function (todoList) {
          App.main.show(new TodoList.Views.ListView({
                collection: todoList
          }));
    },
```

Funktiolle annetaan parametrina käsittelijän aiemmin luoma kokoelma. Aiemmin määriteltyyn alueeseen main sidotaan uusi näkymä, joka on käyty lävitse kohdassa 4.4.2 ja välitetään kokoelma sille. Muut `show` -funktiot ovat hyvin samanlaisia, ainoastaan alue ja siihen sidottava näkymä muuttuvat.

4.7 Alustimet

Alustimilla (Initializer) määritellään funktioita jotka suoritetaan kun sovellus käynnistetään. Käsittelijän luominen hoidetaan alustimella:

```
TodoList.addInitializer(function () {
    var controller = new TodoList.Controller();
    controller.router = new TodoList.Router({
          controller: controller
    });
    controller.start();
});
```

`controller.start` on aiemmassa kohdassa määritellyn käsittelijän `start`-funktio. Alustimet ovat yleinen tapa BM-sovelluksissa hoitaa sovellus käyttövalmiiksi. Reititintä (router) joka luodaan keskellä koodia käsitellään tarkemmin kohdassa 4.9.2.

4.8 Tapahtumat

Backbone sisältää mekaniikan jolla mistä tahansa JS-koodista voidaan lähettää tapahtumia (Event) ja rekisteröityä kuuntelemaan tapahtumia. Backbone lähettää myös itse tapahtumia, esim. kun jokin malli muuttuu, poistuu tai onnistuneesti synkronisoidaan palvelimen kanssa. Tapahtumia käytetään ohjelmiston eri osien väliseen kommunikointiin.

Kokoelmanäkymä (määritetty kohdassa 4.4.2) asetetaan kuuntemaan tapahtumaa jolla käsketään näkymää näyttämään kaikki työtehtävät tai ainoastaan sellaiset joiden tila on keskeneräinen tai valmis:

```
    App.vent.on('todoList:filter', function (filter) {
          filter = filter || 'all';
          $('#todoapp').attr('class', 'filter-' + filter);
    });
```

Mikäli tapahtuman yhteydessä ei ole annettu parametria, käytetään parametrina merkkijonoa `'all'`. Käyttäen jQuery:n valitsinta asetetaan CSS -luokka, joka muuttaa elementin ulkonäköä, tässä tapauksessa piilottaen sen kokonaan näkyvistä.

Viesti lähetetään käsittelijän funktiosta filterItems:

```
    filterItems: function (filter) {
    App.vent.trigger('todoList:filter', (filter && filter.trim()) || '');
}
```

Kyseistä funktiota kutsutaan reittien (Routes) kautta mistä lisää kohdassa 4.9.2. Tärkeää on ymmärtää että tapahtumia voi lähettää ja vastaanottaa mistä tahansa osasta sovellusta.

4.9 Lisätoiminnallisuudet

Edellä on käyty lävitse oleellisimmat osaset joista TodoMVC -sovellus koostuu. Tämän kohdan alla käydään lävitse käteviä lisätoiminnallisuuksia.

4.9.1 Sync ja REST API

Backbone sisältää tuen mallien ja kokoelmien CRUD -operaatioille palvelimen kanssa. Oletustoteutus (de)serialisoi mallin JSON-muotoon ja kommunikoi palvelimen kanssa REST API:n ylitse. Käyttäjä voi halutessaan uudelleentoteuttaa haluamansa osan synkronisointikoodista.

Mallille voidaan asettaa URL, jonka jälkeen voidaan kutsua mallille `fetch`, `save`, `delete` -funktioita, ja sync -toteutus hoitaa asian taustalla. Funktioille annetaan takaisinkutsufunktiot (callback), joiden kautta ohjelma saa tiedon siitä onnistuiko operaatio vai menikö mahdollisesti jotain pieleen.

Tämä sama tuki tarjoaa erittäin helpon tavan integroida sovellus olemassaoleviin REST -rajapintoihin jotka tarjoilevat dataa JSON-muodossa. Esimerkiksi:

```
var m = new Backbone.Model();
m.url = 'http://host.domain/path/to/resource/42';
m.fetch({ success: onSuccessFunction, error: onErrorFunction });
```

Fetch -kutsun onnistumisen jälkeen saatua dataa voi käsitellä käyttäen kaikkia Backbonen ja Underscoren tarjoamia käteviä funktioita.

HTML 5 toi mukanaan LocalStorage-ominaisuuden jolla sovellus voi tallettaa dataa käyttäjän selaimen kautta lokaalisti käyttäjän kovalevylle. `Todos.TodoList` -kokoelmaan olisi voinut määritellä

```
    localStorage: new Backbone.LocalStorage('todos-backbone-marionette'),
```

jolloin kokoelma olisi talletettu käyttäen LocalStoragea. Parametrina annettu tunnus on avain jota käytetään erottamaan tämä sovellus mahdollisista muista samalla sivustolla pyörivistä sovelluksista jotka käyttävät LocalStoragea.

4.9.2 Reititys

Reitityksellä (Routing) sidotaan URL:it käsitteleviin funktioihin. TodoMVC -sovelluksessamme reititin määritellään käsittelijän yhteydessä:

```
TodoList.Router = Marionette.AppRouter.extend({
    appRoutes: {
          '*filter': 'filterItems'
    }
});
```

`appRoutes`- parametrin kautta kerrotaan, että käyttäjän navigoidessa mihin tahansa filter -päätteiseen URL:iin kutsutaan `filterItems` -funktiota (määritelty kohdassa 4.8). Varsinainen instanssi tästä määrittelystä luodaan käsittelijän luomisen yhteydessä. TodoMVC:n HTML-tiedostossa on määritetty alatunnisteen yhteydessä yksinkertaisesti linkkejä:

```
<ul id="filters">
<li> <a href="#">All</a> </li>
<li> <a href="#active">Active</a> </li>
<li> <a href="#completed">Completed</a> </li> </ul>
```

Tällöin nämä linkit toimivat nappeina jotka muuttavat ohjelman tilaa mutta samalla tallettavat ohjelman tilan osaksi koko selaimen osoitetta (tässä tapauksessa käyttäen hash fragmentteja). Tällöin käyttäjä voi tallettaa kirjanmerkin joka vie suoraan johonkin sovelluksen sisäiseen tilaan tai jakaa tämän kirjanmerkin kaverinsa kanssa.

On suositeltavaa luoda kullekin loogiselle toimintokokonaisuudelle oma routerinsa. Hyvä tapa hoitaa tämä on käyttää alustajaa.

4.9.3 Historia

TodoMVC:n tapauksessa halutaan sovelluksen initialisoinnin jälkeen aloittaa Backbonen syvälinkkauksen mahdollistava historia-ominaisuus:

```
TodoMVC.on('initialize:after', function () {
    Backbone.history.start();
});
```

Tämä mahdollistaa mm. sen, että kun käyttäjä klikkailee reittilinkkejä sovelluksessa, eteen (forward)- ja taakse (back) -napit toimivat selaimessa oikein. Backbone toteuttaa historian ja linkityksen oletuksena HTML 5:ssa tulleella History API:lla (W3C 2014). Tällöin URL:it ovat muodoltaan samanlaisia kuin perinteisessä web-sivustossa. Mikäli

käyttäjän selain ei vielä tue History API:a, Backbone vaihtaa automaattisesti hash fragmentteihin (esim. http://esim.erkki/sovellus#kiwis/42).

4.9.4 Templaattien apufunktiot

Joskus templaattien sisällöksi kaivataan monimutkaisemman logiikan tuloksia. Tällöin logiikkaa ei tarvitse koodata templaattiin, vaan templaatista voi kutsua näkymän funktiota. TodoMVC -sovelluksessa tätä käytetään aktiivisten työtehtävien näyttämisen yhteydessä näyttämään oikealla tavalla joko monikko tai yksikkö sanasta. Osa alatunnisteen templaattikoodia:

```
<span id="todo-count">
    <strong><%= activeCount %></strong> <%= activeCountLabel() %>
</span>
```

`activeCountLabel` -funktio on määritelty alatunnisteen näkymän määrittelyssä seuraavasti:

```
templateHelpers: {
    activeCountLabel: function () {
           return (this.activeCount === 1 ? 'item' : 'items') + ' left';
    }
},
```

Eli siis `templateHelpers` -parametrilla määritetään kaikki apufunktiot joita tämän näkymän templaattikoodista voi kutsua. Apufunktiosta pääsee käsiksi kaikkeen näkymässä olevaan dataan.

4.9.5 Mallin serialisointi templaattia varten

Kun templaatteja täytetään näkymän tiedoilla, BM:n oletustoteutus serialisoi näkymän mallin tai kokoelman JSON-muotoon ja välittää tämän templaatille. Tämän takia templaateissa käytetään notaatiota

```
<p> <% name %> <p>
```

hakemaan mallista arvo avaimella name. Serialisoinnin voi kuitenkin uudelleentoteuttaa, ja näin on tehty TodoMVC:n alatunnisteen näkymässä:

```
serializeData: function () {
    var active = this.collection.getActive().length;
    var total = this.collection.length;
    return {
           activeCount: active,
           totalCount: total,
           completedCount: total - active
    };
},
```

Kun näkymän määrittelyssä annetaan serializeData -parametri extend-funktiolle, annettu parametrifunktio uudelleentoteuttaa näkymän serialisoinnin. Funktion pitää palauttaa validi JSON-objekti. Näitä käytetään templaatissa samalla notaatiolla kuin aiemmissakin templaateissa:

```
<script type="text/html" id="template-footer">
    <span id="todo-count">
           <strong><%= activeCount %></strong> <%= activeCountLabel() %>
    </span>
    <ul id="filters">
           <li> <a href="#">All</a> </li>
           <li> <a href="#active">Active</a> </li>
           <li> <a href="#completed">Completed</a> </li>
    </ul>
    <button id="clear-completed"
           <% if (!completedCount) { %>class="hidden"<% } %>>
           Clear completed (<%= completedCount %>)
    </button>
</script>
```

Serialisoinnin uudelleentoteutuksella voi siis näkymän toteutuksessa valita, mitä tietoja annetaan templaatin käyttöön.

5 MITEN BACKBONE/MARIONETTE VASTAA OHJELMISTOTUOTANNON HAASTEISIIN?

Mikkonen & Taivalsaari (2007) sekä Taivalsaari et al. (2008) käsittelevät JavaScriptilla tehostettujen sivustojen sekä selaimessa suoritettavien web-sovellusten haasteita erilaisista ohjelmistotuotannon näkökulmista. Tässä luvussa tarkastellaan miten BM vastaa näihin haasteisiin. BM:n tapauksessa on muistettava, että BM kevyesti ohjailevana kirjastona ei pakota käyttämään mitään alla käsitellyistä toiminnallisuuksista. Alla olevissa kappaleissa BM:ea on siis käsitelty kuin sitä käyttävä kehittäjä olisi järkevä ja vastuullinen ammattilainen joka aidosti välittää ohjelmistonsa laadusta.

5.1 Modulaarisuus

Modulaarisuus mittaa kuinka iso osa ohjelmasta on toteutettu *moduuleilla*. *Moduuli* on itsenäinen ohjelman komponentti, jolla on hyvin määritelty rajapinta. Ohjelma on modulaarinen, mikäli sen moduuleja voi vaihtaa tai uudelleentoteuttaa ilman että se aiheuttaa isoja muutoksia muihin moduuleihin (Parnas 1971).

Mikkosen & Taivalsaaren mukaan JavaScriptilla tehostetut sivustot eivät ole modulaarisia, koska niissä 1) sekoitetaan deklaratiivista (esim. CSS ja HTML) ja proseduraalista (JavaScript) koodia vapaasti ilman kunnollista järjestystä tai rakennetta, 2) sekoitetaan sovelluslogiikkaa, kaikkea käyttöliittymään liittyvää koodia sekä käyttöliittymän ulkoasuun liittyvää koodia ja 3) ollaan riippuvaisia työkalujen käytännöistä, tavoista ja oletuksista (Mikkonen & Taivalsaari 2007).

Haasteeseen 1 BM erityisesti ja SPA-teknologiat yleensä vastaavat ensinnäkin jakamalla kunkin tyyppisen koodin omiin tiedostoihinsa. Esimerkiksi CSS-koodit menevät loogisiin kokonaisuuksiin jaettuna omiin tiedostoihinsa, kuten myös HTML-koodi. Sekä BM:ssa että monissa muissa tapauksissa templaattikoodin voi pilkkoa omiin pieniin tiedostoihinsa, jolloin esimerkiksi kaikki tietyn sovelluksen tietyn loogisen kokonaisuuden kaikki templaatti-koodit ovat omassa hakemistossaan. JavaScript -koodin voi vastaavasti jakaa omiin tiedostoihinsa. Esim. BM:n tapauksessa yksi tapa suorittaa jako olisi luoda sovelluksen alihakemiston alle malleille, kokoelmille, näkymille, käsittelijälle sekä reitittimille kullekin oma tiedostonsa. Näitä

voi jakaa vielä tarkemminkin loogisiin kokonaisuuksiin sovelluksen ollessa suurempi, sillä BM:n moduuli-järjestelmän (käsitelty kohdassa 4.2) avulla samaan moduuliin pääsee käsiksi eri tiedostoista. Kun koodi on näin jaettu moniin tiedostoihin, on suositeltavaa käyttää sovelluksen julkaisemista varten jonkinlaista työkalua yhdistämään pienet tiedostot yhdeksi isoksi tiedostoksi, jotta sovelluksen lataaminen tapahtuisi tehokkaasti.

Haasteisiin 1 ja 2 BM ja muut SPA-teknologiat vastaavat MVC:lla tai jollain sen johdannaisella. Ohjelmakoodin jakaminen loogisiin kokonaisuuksiin tuo monia etuja, kuten on käsitelty tarkemmin kohdassa 2.2.5. Lisäksi BM vastaa haasteeseen 2 näkymien ui-muuttujalla, jonka avulla jQuery -valitsimet muunnetaan muuttujiksi kootusti yhdessä paikassa sen sijaan että valitsimia olisi siellä täällä näkymän koodin seassa.

Haasteen 3 suhteen BM aiheuttaa kevyesti ohjailevana kirjastona suhteellisen vähän ongelmia. BM-sovelluksen uudelleenkirjoittaminen käyttämään jotain muuta SPA-kirjastoa tai -kehystä on verrattavissa työläytensä puolesta kevyesti ohjailevaa kirjastoa käyttävän desktop-sovelluksen uudelleenkirjoittamiseen jollain toisella kirjastolla.

5.2 Konsistenttius, yksinkertaisuus ja eleganssi

Konsistenttius on ohjelmistotuotannon periaate minkä mukaan ohjelmistossa pitäisi olla minimimäärä käsitteitä ja yksinkertaiset säännöt niiden yhdistelemiseen sekä asioiden samankaltaisuuden pitäisiä näkyä niistä ulospäin (MacLennan 1999). Mikkosen & Taivalsaaren mukaan web-sovellusten kehitystyökalut ovat heikosti konsistentteja, koska 1) on useita tapoja tehdä sama asia ja 2) asiat tapahtuvat sivuvaikutusten kautta eikä eksplisiittisesti (Mikkonen & Taivalsaari 2007).

BM vastaa haasteeseen 1 melko heikosti ja tekee näin suorastaan tarkoituksellisesti. BM:n antaa käyttäjänsä valita tavan jolla kirjastoa käyttää. Haasteen 2 BM ottaa vastaan hyvin: osien välinen kommunikointi hoidetaan joko suorilla kutsuilla, takaisinkutsufunktioilla tai tapahtumilla. Kaksi viimeistä tapaa eivät ole yhtä eksplisiittisiä kuin ensimmäinen ja laajalti viljeltyinä saattavat johtaa ns. takaisinkutsu-helvettiin, jossa ohjelman kulkua on haastava seurata. Ratkaisuna tähän on esitetty reaktiivista ohjelmointia (reactive programming) ja lupauksia (promise) (Kambona et al. 2013).

Laadukkaat ohjelmistot pyrkivät olemaan yksinkertaisia ja toteutuksen ratkaisuiltaan elegantteja. Mikkonen & Taivalsaari toteavat, että web-järjestelmät eivät sitä ole (Mikkonen & Taivalsaari 2007). Marionette sisältää paljon valmiita rakenteita usein SPA:n kehityksessä toistuville suunnittelumalleille, mikä vähentää kirjoitettavan ja ylläpidettävän koodin määrää. Vahvasti ohjailevien työkalujen kanssa tosin koodia pitäisi syntyä vieläkin vähemmän.

Mikkonen & Taivalsaari toteavat myös, että web-sovellusten koodi on rakenteeltaan huonoa ja täten vaikealukuista (Mikkonen & Taivalsaari 2007). Tähän haasteeseen on vastattu kohdassa 5.1 ja samat vastaukset pätevät yksinkertaisuuden ja eleganttiuden suhteen. Mikkonen & Taivalsaari esittävät, että useiden eri teknologioiden (mm. HTML, JavaScript, CSS, XML) sekoittaminen keskenään ei ole eleganttia (Mikkonen & Taivalsaari 2007). Tämän suhteen tilanne ei ole muuttunut mihinkään, vaan pikemminkin web-kehittäjät ovat tottuneet tilanteeseen.

5.3 Uudelleenkäytettävyys ja siirrettävyys

Ohjelman komponetti on *uudelleenkäytettävä* mikäli sitä voi suhteellisen vähällä työllä käyttää jonkin toisen ohjelman komponenttina. Mikkonen & Taivalsaari toteavat että web-sovellusten komponenteista on mahdollista rakentaa uudelleenkäytettäviä mikäli toteuttajat onnistuvat pitämään asiasta huolta (Mikkonen & Taivalsaari 2007). Näin on tilanne myös BM:n kanssa. BM:n MVC-malli auttaa merkittävästi tämän asian kanssa.

Siirrettävyys tarkoittaa tarvittavan muutostyön määrää mikäli ohjelmaa haluttaisiin suorittaa toisessa ympäristössä kuin mihin se alunperin tarkoitettiin. Mikkonen & Taivalsaari ottavat esille selainten välisten eroavaisuuksien aiheuttamat ongelmat, jotka usein pohjautuvat DOM-puun toteutuksen eroihin. BM käyttää jQuerya, joka ratkoo useimmat tämän tyyppiset ongelmat. ProLe -sovelluksen toteutuksessa ei joutunut kirjoittamaan riviäkään selainkohtaista koodia. Toisaalta tämä johtuu myös siitä, että kohteena on vain suosituimpien selaimien uusimmat versiot.

5.4 Käytettävyys

Käytettävyys on laadullinen ominaisuus joka arvioi kuinka helppo käyttöliittymä on käyttää (Nielsen 2012). Mikkonen & Taivalsaari toteavat että selaimen I/O -malli on perustavanlaatuisesti epäsoveltuva desktop-tyyppisten sovellusten rakentamiseen, koska sovellukset joutuvat kommunikoimaan selaimen kanssa pääasiallisesti DOM-puun kautta (Mikkonen & Taivalsaari 2007). BM:n tapauksessa DOM-puu on yhä olemassa, mutta se on abstrahoitu kirjastojen taakse eikä sitä muokata suoraan puumuodossa. Sen sijaan käsitellään MVC-mallin osia ja jQueryn avulla haettuja yksittäisiä elementtejä. Templaattien avulla jQuerylla elementteihin navigoiminen on helppoa, vaivatonta ja helposti ylläpidettävää.

Mikkonen & Taivalsaari totevat, että perinteinen web-sivustojen sivupohjainen malli tuo mieleen 1970-luvun antiikkiset terminaalit (Mikkonen & Taivalsaari 2007). Juuri tätä ongelmaa SPA:t korjaavat. Koska kyseessä on JS-koodi jota ajetaan selaimessa, käyttöliittymä reagoi välittömästi ja käyttäjälle tulee näin vähemmän odottamisen tunnetta.

Mikkonen & Taivalsaari toteavat, että monet selaimen perusominaisuuksista toimivat huonosti selaimessa pyörivien sovellusten kanssa. Esimerkkeinä he nostavat sivun uudelleenlatauksen, latauksen pysäyttämisen ja eteen- ja taakse -napit (Mikkonen & Taivalsaari 2007). BM:ssa eteen- ja taakse -napit toimivat syvälinkkauksen ansiosta järkevällä tavalla. Uudelleenlataus ja latauksen pysäyttäminen sen sijaan ovat nappuloita joille BM ei tarjoa järkevää käyttötarkoitusta.

Eräs käytettävyyden haasteista on ohjelmiston offline -käyttö yhteyden katketessa palvelimelle. Taivalsaari et al. toteavat tämän aikansa teknologioilla miltei mahdottomaksi haasteeksi (Taivalsaari et al. 2008). Backbonelle on useita laajennoksia jotka ratkaisevat ongelman eri tavoilla (Anderson 2014, Kinvey 2014). Yksi tapa on datan tallettaminen HTML 5 LocalStorageen, ja datan synkronisointi kun yhteys palvelimelle palaa. Tällaisen sovelluksen rakentaminen on kuitenkin hieman tavallista haastavampaa. Kehittäjän pitää ottaa huomioon esim. se, että yhteyden palatessa palvelimelle ja synkronisoinnin syystä tai toisesta epäonnistuessa sovelluksen käyttäjä saattaa hyvin olla navigoinut täysin eri osaan sovellusta tai että datan tallettamisen yhteydessä useampi operaatio saattaa epäonnistua.

5.5 Testattavuus

Testattavuus kuvaa sitä kuinka helppoa ohjelmalle on kirjoittaa kattavia automaattisia testejä. Mikäli koodi on huonosti testattavissa, koodi on todennäköisesti myös laadullisesti huonoa ja vaikeaa ylläpitää.

Taivalsaari et al. toteavat, että web-sovelluksen JavaScript -koodi pitää testata inkrementaalisesti ja että kattavien testien kirjoittaminen on tärkeää JavaScriptin äärimmäisen dynaamisen luonteen takia (Taivalsaari et al. 2008). Myös BM:ssa nämä toteamukset pitävät paikkansa. Onneksi hyvin rakennettu BM on helposti testattavissa. Joko puhdasta tai osittaista TDD:tä voi suositella BM:n kanssa kehittämiseen. Ohjelmassa pitää olla automatisoituja testejä joita ajetaan automaattisesti ja joiden kattavuutta seurataan kätevän automatisoidun työkalun kanssa.

5.6 Tuottavuus

Tuottavuus tarkoittaa sitä kuinka nopeasti työkalulla pystyy toteuttamaan ominaisuuksia ja tekemään muutoksia ohjelmaan tietyllä etukäteen kiinnitetyllä laadulla. Siirrettävyyden yhteydessä käsitellyt selainten välisten erojen aiheuttamat ongelmat aiheuttavat ongelmia myös tuottavuuden kannalta, mikäli tarvitsee tukea selaimien vanhoja versioita. BM:n etu on tässä suhteessa se miten laajalti sitä on käytetty, jolloin mahdollisiin ongelmiin on helppo löytää apua.

Taivalsaari et al. kiinnittävät huomiota siihen, että JavaScript -koodissa ei ole staattista tyyppitarkastusta eikä tukea staattiselle verifioinnille (Taivalsaari et al. 2007).

Tässä suhteessa nykyaikaisen IDE:n ja staattisen koodianalysoijan käyttäminen (esim. JSHint) auttaa merkittävästi, koska ne saavat bugeja kiinni samaan aikaan kun kirjoitat koodia.

Taivalsaari et al. nostavat esille ongelmat joita voi tulla sovelluksen päivittämisen yhteydessä. Esimerkiksi osa käyttäjistä saattaa käyttää vanhaa versiota sovelluksesta vaikka palvelinpää on jo päivitetty (Taivalsaari et al. 2007). Tähän ongelmaan on kaksi ratkaisua. Ensimmäinen on toteuttaa Meteorin hot code push -tyyppisen ominaisuuden, jolla päivitetään ajossa olevia sovelluksia lennossa. Toinen vaihtoehto on versioida sekä palvelinpää että sovellus ja ajaa useita sovelluksia rinnakkain. Versioinnissa on omat haasteensa, mutta toisaalta sen toteuttamisen jälkeen voi tehdä mielenkiintoisia asioita, kuten esimerkiksi uuden version massatestausta suhteellisen pienellä määrällä oikeista käyttäjistä. Tällöin saadaan oikea ja realistinen massa testausta varten, mutta kuitenkin mahdollisista ongelmista joutuu kärsimään vain suhteellisen pieni määrä käyttäjistä.

Lopuksi on todettava, että vahvasti ohjailevien kehysten pitäisi alkuajan opettelun jälkeen tarjota parempi tuottavuus kuin BM.

6 PROLE- SOVELLUKSEN TOTEUTTAMINEN

6.1 Työkalut ja kirjastot

Backbonen ja Marionetten lisäksi työssä käytetään luonnollisesti muitakin työkaluja. Grunt -työkalua käytetään automatisoimaan JavaScriptin paketointia. Grunttiin on myös sidottu kiinni JSHint-työkalu joka automaattisesti paketoinnin yhteydessä tarkistaa koodin laadun ja raportoi mahdollisista ongelmista, estäen paketoinnin virheiden löytyessä. JS-koodin yksikkötestit on kirjoitettu Jasminea (BDD-testauskehys) apuna käyttäen. Istanbul-työkalulla tarkastetaan koodin testikattavuutta. Jasminella testien ajaminen ja Istanbulin raportoinnin ajaminen ovat myös sidottu Gruntin paketoinnin automaattiseksi vaiheeksi.

BIT:issä oli jo aiemmin käytetty JBehavea (BDD-testauskehys) integraatio- ja järjestelmätestien ajamiseen. JBehave ajaa BIT:issä selkokielisiä tarinoita oikeassa selaimessa (tällä hetkellä pelkästään Firefoxissa, mutta voisi ajaa muillakin selaimilla). JBehave otettiin käyttöön myös ProLen smoketestaamiseen.

Kaavioiden piirtämiseen käytetään NVD3 -kirjastoa, joka hoitaa kaavioiden piirtämistä JavaScriptilla. NVD3 pohjautuu suosittuun D3.js -kirjastoon.

6.2 Projektin eteneminen

Toteutus lähti liikkeelle nopealla protoilulla. Hyvin nopeasti käyttäen pelkkää JavaScriptia ja D3.js -kirjastoa syntyi ensimmäinen proto joka haki BIT:istä dataa ja visualisoi budjetin kulumista. Ensimmäinen proto otettiin myös samantien käyttöön BIT:issä.

Tämän jälkeen budjettikaavio uudelleenkirjoitettiin käyttäen Backbone + Marionettea. Tälle versiolle kirjoitettiin Jasminea käyttäen yksikkötestit. Tässä versiossa siirryttiin käyttämään D3.js:n päälle rakennettua NVD3:a.

Tähän asti BIT oli ainoastaan tarjonnut yksittäisen summan projektin budjettina. Tässä vaiheessa todettiin että backendiin tarvitsee toteuttaa tuki jonain päivinä tulleille ostotilauksille (purchase order). BIT:tiin lisättiin yksinkertainen CRUD ja JBehave -testit.

Projekti eteni ketterästi käyttäen Trelloa etenemisen seuraamiseen. Muu Vincitin BIT-tiimi katselmoi kaiken tuotetun koodin Gerrit-järjestelmän kautta.

6.3 Testaus

Backbonen mallit ja kokoelmat funktioineen testataan Jasmine -yksikkötesteillä. Jasmine on myös sidottu Grunt-työkalun automaattiseen kehitys- ja julkaisusykliin siten, että testit ajetaan näissä aina automaattisesti.

JBehavella testataan että sivulle ilmestyy jonkinlainen kaavio kun dataa on saatavilla ja oikea virheilmoitus silloin kun riittävää dataa projektin työtunneista ei vielä löydy, Trelloon ei saada yhteyttä tai Trellosta ei ole dataa saatavilla. Loppua SPA-käyttöliittymää ei testata mitenkään, joten käyttöliittymän testien suhteen voidaan ajatella lähinnä olevan kyse ns. smoke testeistä.

Sovellusta testattiin kehityksen aikana käsin pääsääntöisesti Chromen uusimmalla versiolla. Sovellus on kevyesti testattu toimivaksi myös Firefoxin ja IE:n uusimmilla versioilla ja yhden mobiililaitteen mobiili-selaimella (Samsung Galaxy Note, oletuselain, uusin versio). Kattavaan testaukseen eri selaimilla ei siis ole edes pyritty.

6.4 Integraatiot

ProLe hakee BIT:istä kaikki budjettiin liittyvät tiedot ja Trellosta kaikki ominaisuuksien etenemiseen liittyvät tiedot.

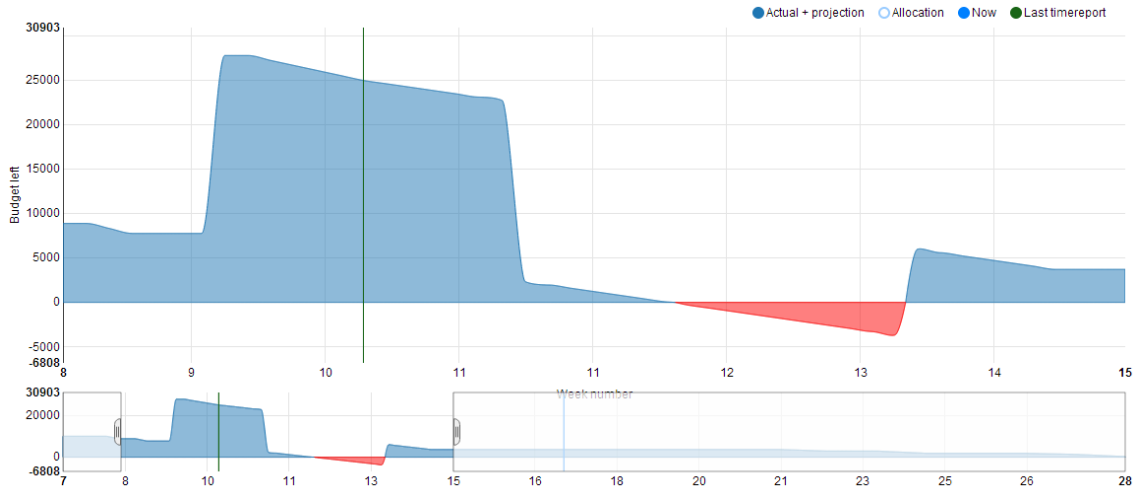
BIT:istä haetaan kaikki tarvittavat tiedot projektista, projektin tehtävistä, tehtäviin merkityistä työtunneista ja työntekijöiden allokaatioista projekteille. Data haetaan yksinkertaisen REST -rajapinnan ylitse JSON:ina.

Trellosta data haetaan käyttäen Trellon tarjoamaa client.js -kirjastoa, mikä hoitaa käyttäjälleen mm. OAuth -autentikoinnin ja muutenkin helpottaa asioita melkoisesti. Trello -integraatio hoidetaan niin, että kuka tahansa projektiin kirjoitusoikeudet omaava BIT:in käyttäjä voi ProLe -sovelluksen kautta assosioida Trello -taulun projektiin. Assosiointi hoidetaan käyttäen client.js -kirjastoa, jolloin käytännössä käyttäjälle avautuu uusi ikkuna, jossa hän voi autentikoitua Trelloon ja antaa suostumuksensa siihen, että BIT saa toistaiseksi voimassaolevan lukuoikeuden kaikkiin käyttäjän tauluihin. Taulun tunniste ja erityisesti käyttäjän Trello-avain (token) talletetaan BIT:in palvelimelle. Tämän jälkeen muut BIT:in käyttäjät joilla on lukuoikeus projektiin pystyvät näkemään CFD:n jonka tiedot haetaan käyttäen toisen käyttäjän Trello-avainta.

6.5 ProLen ominaisuudet

ProLe näyttää kaksi kaaviota. Ensimmäinen näistä visualisoi jäljellä olevan budjetin ajan funktiona. Siinä on kaksi pehmennettyä viivakuvaajaa, joista ensimmäinen on toteutuneiden kulujen ja kulujen tulevaisuusennusteen yhdistelmä, ja toinen pelkkä ennuste koko projektin ajalta. Kuvassa 5.1 näytetään budjettikaavio. Siinä näytetään selkeyden vuoksi ainoastaan toteutuneiden kulujen ja ennusteen yhdistelmä, eli siis pelkän ennusteen kuvaaja on piilotettu kaavion käyttöliittymän kautta. Yläosa on

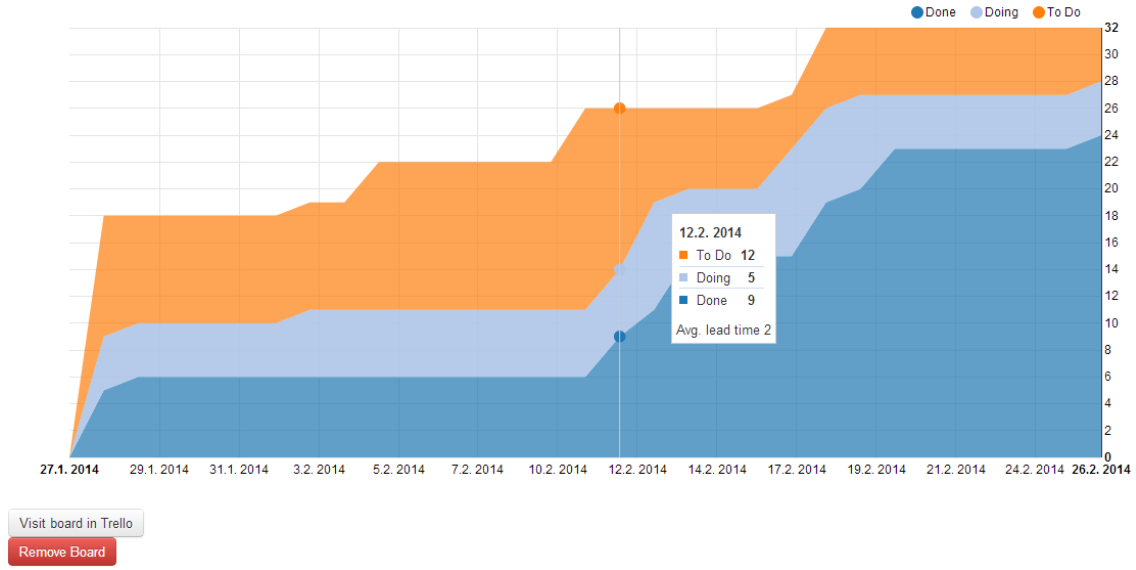
varsinainen kaavio ja alaosassa on interaktiivinen aikavälin valitsin. X-akselilla juoksevat viikkonumerot.



***Kuva 5.1**: ProLen budjettikaavio*

Negatiiviselle menevä budjetti piirretään punaisella. Vihreä pystyviiva merkitsee viimeisimmän tuntimerkinnän päivää, jonka jälkeinen tilanne projisoidaan henkilöiden allokaatioiden ja tuntimerkinnöistä lasketun keskimääräisen tuntihinnan perusteella.

Toinen kaavio on Trellon tietojen perusteella luotava CFD. Se on aluekaavio, jossa näytetään ajan funktiona korttien määrä kussakin vaiheessa. Kuvassa 5.2 näytetään CFD.



***Kuva 5.2**: ProLen CFD*

Mikäli hiirtä kuljettaa CFD:n päällä piirretään kaavion päälle pystyviiva joka näyttää päivän jota tarkastellaan, ja sen vieressä kelluvan laatikon. Laatikossa listataan kussakin vaiheessa olevien korttien määrän kyseiselle päivälle. Lisäksi kyseiselle päivälle lasketaan ja näytetään keskimääräinen kesto tehtävän aloituksesta sen valmiiksi saattamiseen (Lead time).

Molemmat kaaviot ovat interaktiivisia siinä mielessä, että oikean yläkulman selitteen tekstejä tai palloja painamalla voi kytkeä kaavion kuvaajia näkyville tai pois näkyvistä. Jos esimerkiksi haluaisi nähdä CFD:ssa ainoastaan Doing -vaiheessa olevien korttien vaihtelun ajan funktiona, tämä onnistuisi kytkemällä muut vaiheet pois näkyvistä.

Yhdessä näistä kaavioista projektin leadi näkee yhdellä silmäyksellä miten budjetti on kulunut tähän asti ja koska se nykyisen ennusteen mukaan loppuu (tai on jo ehtinyt loppua) sekä myös sen, miten työtehtävien toteutuminen on edennyt.

6.6 Datan käsittely

Kuten kohdassa 2.3.3 todetaan, BIT:issä kuhunkin projektiin liittyy 0..n työtehtävää ja 0..n ostotilausta. Työntekijät merkitsevät tehtyjä tunteja työtehtäville. Jokaisella työtehtävällä on oma tuntihintansa. Lisäksi työntekijöiden tunteja allokoidaan projektien viikoille.

ProLe hakee BIT:in REST-rajapinnasta tiedot tehdyistä työtunneista, allokaatioista, työtehtävistä ja ostotilauksista. Budjetti kasvaa kun projektiin tulee ostotilauksia. Haetusta datasta lasketaan kunkin päivän toteutuneet kulut. Lisäksi tehdyistä työtunneista lasketaan kunkin henkilön keskimääräinen tuntihinta, ja kun tämä yhdistetään allokaatioihin, saadaan ennuste tulevaisuuden kuluista.

Budjettikaavion ensimmäinen kuvaaja alkaa päivien toteutuneilla kuluilla, minkä jälkeen kuvaajaa jatketaan allokaatioihin perustuvalla tulevaisuuden kuluennusteella. Toinen kuvaaja on pelkkä allokaatioihin perustuva ennuste. Tämän kuvaajan tarkoituksena on antaa projektin leadille tuntumaa ennusteen ja toteutuman suhteesta, ja tätä kautta arviota ennusteen luotettavuudesta.

Kuten kohdassa 2.2.4 todetaan, Trellossa työtehtävää kuvaava kortti kuuluu aina yhteen työvaiheeseen, ja kortteja siirrellään työvaiheiden välillä. Trellon API:sta on mahdollista hakea tiedot siitä, miten kortit ovat liikkuneet työvaiheiden välillä. Näiden tietojen perusteella muodostetaan kullekin kortille ajallinen jatkumo, minkä perusteella lopulta lasketaan kullekin päivälle kuinka monta korttia oli kussakin vaiheessa sinä päivänä.

6.7 Tekninen toteutus

Toteutus sidottiin olemassaolevaan Marionette.Applicationiin omaksuen projektissa käytetyt tavat. Sekä budjettikaaviosta että CFD:sta tehtiin oma sovelluksensa.

Toteutuksessa käytettiin MVC-mallia. Kaavioille luotiin omat näkymät. BIT:istä haettavalle datalla luotiin useita omia malleja ja kokoelmia ja osittain pystyttiin käyttämään olemassaolevia. Kaavioiden käyttöliittymä ja interaktiivisuus hoidettiin täysin NVD3:n kautta. NVD3:a laajennettiin hieman, mm. jotta yhteen viivakuvaajaan saatiin vaihtuvia värejä ja jotta aluekaavion päällystyökalussa saatiin näytettyä päivästä riippuvaista dataa.

Dataa käsitellään Backbonen malleina ja kokoelmina, kunnes aivan lopussa se annetaan nv3d:lle sen vaatimana mallien taulukkona. Tällöin NVD3:lle annetaan hakufunktiot (accessor function) joiden avulla se osaa hakea datan Backbonen mallista.

Sovelluksen jatkokehitysajatuksissa on toteuttaa aikavalitsin myös CFD:iin ja tämän jälkeen syvälinkata valittu ajanjakso. Tämä tarkoittaisi sitä, että käyttäjän rajatessa näytettyä aikaa kummalle tahansa kaaviolle tai molemmille niistä, sovelluksen URL päivittyisi välittömästi. Tällöin käyttäjä voisi esim. jakaa toiselle käyttäjälle linkin joka veisi suoraan sovelluksen tilaan, jossa kaavioissa on rajattu näytetty aikaväli.

7 TULOKSET JA NIIDEN TARKASTELU

Kuten edellä on todettu, BIT:tiä kehittävät Vincitiläiset haluavat että uudet kehitettävät BIT:in ominaisuudet ovat arkkitehtuurillisesti irrallisia BIT:istä ja mahdollisesti myös muista BIT:in osista. SPA modulaarisuutensa kautta täytti hyvin tämän toiveen. Ylläpidon kannalta modulaarisuudesta on varmasti myöskin hyötyä.

Kokonaisuutena BM oli hyvä ratkaisu. Sillä oli helppo ja nopea päästä liikkeelle. Kehitys oli suhteellisen helppoa ja vaivatonta, eikä se pysähtynyt merkittävän pitkiksi ajoiksi BM:n kanssa taistelemiseen. Tuottavuus tuntui siis olevan hyvällä tasolla. Tähän varmasti vaikutti se, että sovellukset olivat loppujen lopuksi suhteellisen yksinkertaisia erityisesti käyttöliittymän suhteen. Mikäli sovelluksissa olisi ollut monimuotoisempia ja enemmän käyttöliittymiä, AngularJS olisi todennäköisesti ollut parempi ratkaisu. BM sopi myös nopean protoilun lähestymistapaan, eli BM taipui varsin hyvin nopean protoilun aiheuttaman jatkuvan refaktorointitarpeen alla.

Ylläpidon kannalta BM oli hyvä ratkaisu, koska BM on tuttu monille Vincitin työntekijöille ja myös nykyisille BIT:in kanssa työskenteleville. Lisäksi työn tekemisen yhteydessä syntyneitä moduuleita pystyy mahdollisesti käyttämään tulevaisuudessa muussa BIT:in kehityksessä.

BM:lla tuntui olevan luontevaa kirjoittaa koodia joka on rakenteeltaan helposti testattavaa. Testauksen ulkopuolelle jäi ainoastaan NVD3:n generoimien kaavioiden sisältö. Näiden testaaminen olisikin haastavaa, koska kyseessä on SVG:n path-elementti, jossa olevien pisteiden sisältö muuttuu mm. selaimesta ja selaimen koosta riippuen. Sovelluksen osat ovat kuitenkin testattavissa kuitenkin siihen asti, että sovellus tuottaa ulos kokoelman kaavioiden kuvaajia, eli siis joukon kuvaajien pisteitä.

JBehave -testaaminen hoiti integraatiotestaamisen erittäin hyvin. JBehave oli yksi uusista työkaluista joihin työn aikana pääsi tutustumaan ja kokemus oli varsin positiivinen.

Kattavien testien kirjoittamiseen kului merkittävästi aikaa. JBehavessa itse tarinan (Story) kirjoittaminen oli nopeahkoa, mutta uusien askelmien (Step) eli tarinan vaiheita tukevan koodin kirjoittamiseen paloi aikaa. Yksikkötestien ajaminen oli nopeaa, mutta JBehave -tarinan ajaminen kesti minuutista kahteen. Kokonaisuutena kuitenkin automatisoituun testaukseen käytetty aika on todennäköisesti jo maksanut itsensä takaisin ja lisäksi lisännyt sovellusten laatua merkittävästi.

Yksikkö- että integraatiotestien oleminen osa CI-palvelimen automaattista sykliä on erittäin mainio turvaverkko ja automatisoitu tapa havaita bugit mahdollisimman aikaisessa vaiheessa.

Jasmine ja Istanbul olivat teknisesti suhteellisen helppoja tuoda mukaan projektiin. Jasminen tuominen BIT:tiin helpottaa muiden testaamisen aloittamista ja madaltaa kynnystä aloittaa. JSHint oli mainio työkalu, joka paitsi korotti koodin laatua myös havaitsi bugeja erittäin varhaisessa vaiheessa.

Katselmointi paransi koodin laatua. Ensinnäkin katselmoinnin ansiosta toteuttaessa pyrki selkeämpään ja ylläpidettävämpään koodiin. Toiseksi toisen silmäparin kommenttien ansiosta löytyi bugeja ja koodi parani laadullisesti.

BM oli henkilökohtaisen oppimisen kannalta huono valinta. BM on yksinkertainen kirjasto, jossa ei ole käytetty mitään kovinkaan mullistavia ideoita. Mikäli on jo aiempaa kokemusta SPA-kehityksestä, BM ei tarjoa merkittävästi uutta opittavaa.

CFD tuntuu olevan melko hyödyllisen oloinen metriikka kun arvioidaan projektia ja erityisesti sen valmiusvaihetta. Pienillä (noin kk kestävillä) projekteilla on enemmän haasteita, koska keskimääräiset heitot korttien työmäärissä aiheuttavat enemmän heittelyä. Pienissä projekteissa korostuu siis tarve pilkkoa liian isot kortit pienemmiksi kokonaisuuksiksi ja toisaalta koota liian pienistä korteista isompia kokonaisuuksia. CFD on hyödytön, mikäli projektissa ei käytetä Trelloa Kanban -tauluna. Vaikuttaa kuitenkin siltä, että suhteellisen monissa Vincitin projekteissa käytetään Trelloa sekä nyt että lähitulevaisuudessa.

Sovelluksessa tehdään suhteellisen paljon datan käsittelyä JavaScript-koodissa. Olisiko laadullisesti parempaa tehdä tämä suoraan palvelimella, missä on vahva tyypitys ja kenties luonnollisempaa kirjoittaa yksikkötestejä, ja sitten vain tarjoilla valmiimpaa dataa sovellukselle? Asiaan ei ole yksiselitteistä vastausta. Palvelimella datan käsitteleminen olisi toisaalta myös hieman ristiriidassa BIT:in kehittäjien toiveiden kanssa. Kenties olisi pitänyt olla jokin erillinen palvelin jossa olisi erityisesti datan muokkaukseen ja käsittelyyn keskittyvä teknologia, joka sitten tarjoaisi dataa eteenpäin. Toisaalta tämä olisi tarkoittanut yhden ison liikkuvan osasen lisäämistä systeemiin.

Integraatiot BIT:in REST-rajapintaan ja Trelloon olivat vaivattomia ja helppoja tehdä BM:n tarjoaman hyvän tuen ansiosta. Lisäksi Trello tarjoaa hyvän JavaScript-clientin, jonka kanssa pääsee nopeasti alkuun. Toisaalta Trellon API on betassa ja dokumentaatio erittäin minimaalista. Lisäksi Trellon API:sta saatu data oli silloin tällöin hieman epäilyttävää: esimerkiksi kahta eri kautta sai eri tietoja kortteihin liittyvistä muutoksista, eikä tälle ole löytynyt tähän mennessä mitään selitystä.

NVD3:lla oli helppo luoda nopeasti kaavioita ja sitä oli helppo muokata sen tekijöiden etukäteen ajattelemilla tavoilla. Kaaviot olivat näyttäviä ja niiden interaktiiviset osat intuitiivisia ja oikeasti paransivat kokemusta. Ulkoista dokumentaatiota ei ole, ainoastaan esimerkkejä NVD3:n sivuilla. Kun nousi tarve muuttaa kaavioita tavoilla joita tekijät eivät olleet ottaneet huomioon, joutui käyttämään melko paljon aikaa NVD3:n lähdekoodin lukemiseen ja muokkaamiseen ja myöskin D3.js:n opiskeluun. D3.js:stä on tosin olemassa erittäin kattava ja ajantasalla oleva dokumentaatio, mikä helpotti asiaa melkoisesti.

8 JOHTOPÄÄTÖKSET

Työssä lähdettiin tutustumaan kolmeen nykyaikaiseen SPA-työkaluun ja selvittämään kuinka yksi niistä selviytyy melko pienen sovelluksen toteuttamisessa.

Harkituista työkaluista Meteor on tuore kehys, joka hoitaa kaiken mitä web-sovelluksen kehittämiseen tarvitsee. JavaScriptia käytetään sekä selaimen että palvelimen päässä, ja käytössä on Node.js ja MongoDB.

AngularJS on saavuttanut vakaan julkaisun ja paljon suosiota. Meteorin tavoin se on vahvasti ohjaileva kehys joka pyrkii nopeaan kehitykseen ja mahdollisimman vaivattomaan ylläpitoon.

Backbone + Marionette (BM) on minimalististen kirjastojen yhdistelmä joka pyrkii tarjoamaan perusosat SPA:iden rakentamiseen rajoittamatta paljoakaan käyttäjäänsä. BM on suosiota saavuttanut ja kypsä kirjasto.

Kaikki vertaillut työkalut vaikuttavat vähintäänkin kohtuullisilta tavoilta toteuttaa SPA:eja.

BM sopii projektiin paremmin, mikäli

- projektissa on vähän tai kohtuullinen määrä erilaisia käyttöliittymiä
- toteuttajat haluavat vapautta tehdä asiat omalla tavallaan
- toteuttajat haluavat kypsän ja luotettavan kirjaston, jota on luontevaa käyttää myös vain osan sivuston toteuttamisessa
- tekijät haluavat kirjaston johon voi vapaasti vaihdella osasia ja johon on kypsiä laajennoksia paljon tarjolla
- toteuttajat ovat enimmäkseen aloittelijoita jotka eivät ole ennen toteuttanut SPA:ta ja haluavat oppia perusteet

BM sopii projektiin huonommin, mikäli

- toteuttajat haluavat että työkalu ohjaa voimakkaasti miten työtä tehdään
- projektissa on paljon erilaisia käyttöliittymiä. Tällöinkään BM ei ole huono vaihtoehto, mutta AngularJS olisi parempi, koska sillä näiden kehittäminen on merkittävästi nopeampaa ja ylläpidonkin pitäisi olla vaivattomampaa
- toteuttajat ovat enimmäkseen kokeneita web-kehittäjä ja haluavat oppia uutta

On erittäin mielenkiintoista nähdä mihin suuntaan Meteor kehittyy. Vakaa julkaisu ei varmastikaan ole kovinkaan kaukana.

BM oli asetettuun ongelmaan hyvä ratkaisu. Sillä pääsi nopeasti vauhtiin ja tuottavuus pysyi projektin ajan hyvällä tasolla. Koodi oli miltei täysin testattavissa melko vaivattomasti. Ylläpitovaiheesta ei ole tässä vaiheessa erityisiä pelkoja.

Kuitenkin, mikäli tekisin nykyisillä tiedoilla valintaa uudestaan, valitsisin AngularJS:n. Erityisesti isompaan projektiin jossa on enemmän käyttöliittymiä suosittelisin AngularJS:ää. BM:ää voi kuitenkin suositella, erityisesti pienempiin projekteihin.

LÄHTEET

Anderson D., 2010. Kanban: Successful Evolutionary Change for Your Technology Business. Sequim, Washington, USA, Blue Hole Press.

Anderson E., Backbone.dualStorage: A dual (localStorage and REST) sync adapter for Backbone.js, 2014 [WWW]. [Viitattu 6.5.2014]. Saatavissa https://github.com/nilbus/Backbone.dualStorage.

AngularJS – Superheroic JavaScript MVW Framework, 2014 [WWW]. [Viitattu 12.4.2014]. Saatavissa http://angularjs.org/.

Ashkenas J., Backbone.js, 2014 [WWW]. [Viitattu 31.3.2014]. Saatavissa http://backbonejs.org/.

Bedell K. 2006. Opinions on opinionated software. Linux Journal, volume 2006, issue 147, pp 1.

Farrel J., Nezlek G., 2007. Rich Internet Applications, The Next Stage of Application Development. Proceedings of the ITI 2007 29th Int. Conf. On Information Technology Interfaces, June 25 – 28, 2007, Cavtat, Croatia, pp. 413 – 418.

Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T, 1999. RFC 2616, Hypertext Transfer Protocol – HTTP/1.1. [WWW]. [Viitattu 26.4.2014]. Saatavissa http://www.w3.org/Protocols/rfc2616/rfc2616.html.

Fielding R., Taylor R., 2002, Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology, Vol. 2, No. 2, New York, USA. pp. 115 – 150.

Forrester Consulting, eCommerce Web Site Performance Today, 2009 [WWW]. [Viitattu 12.4.2014]. Saatavissa http://www.damcogroup.com/white-papers/ecommerce_website_perf_wp.pdf.

Fowler M., Presentation Model, 2004 [WWW]. [Viitattu 28.4. 2014]. Saatavissa http://martinfowler.com/eaaDev/PresentationModel.html.

Hammer-Lahav E., The Oauth 1.0 Protocol, 2010 [WWW]. [Viitattu 28.4.2014]. Saatavissa http://tools.ietf.org/html/rfc5849.

Kambona K., Boix E., De Meuter W., 2013. An evaluation of reactive programming and promises for structuring collaborative web applications. Proceedings of the 7th Workshop on Dynamic Languages and Applications, Article No. 3, July 1, 2013, Montpellier, France.

Kinvey, Caching and Offline, 2014 [WWW]. [Viitattu 6.5.2014]. Saatavissa http://devcenter.kinvey.com/backbone/guides/caching-offline.

Krasner Glenn E., Pope Stephen T., 1988. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, vol. 1 Issue 3, Aug. / Sept. 1988. SIGIS Publications, Denville, NJ, USA.

MacLennan B. 1999. Principles of programming languages (3rd ed.): design, evaluation, and implementation. New York, USA, Oxford University Press.

Meteor, A better way to build apps, 2014 [WWW]. [Viitattu 12.4.2014]. Saatavissa https://www.meteor.com/.

Mikowski M., Powell J. 2014. Single Page Web Applications. New York, USA, Manning Publications. 407 p.

Minar I., MVC vs MVVM vs MVP, 2012 [WWW]. [Viitattu 28.4.2014]. Saatavissa https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV

Morris S., The story of www.slashdotslash.com, 2014 [WWW] [Viitattu 28.12.2013]. Saatavissa http://stunix.outanet.com/?p=slash.

Nielsen J., Usability 101: Introduction to Usability, 2012 [WWW]. [Viitattu 6.5.2014]. Saatavissa http://www.nngroup.com/articles/usability-101-introduction-to-usability/.

Parnas D, 1971. Information distribution aspects of design methodology. Pittsburgh, Pensylvania, Carnegie Mellon University.

Ponomarjovs A., 2013, Business Values of Business Intelligence. Master of Science Thesis. Tampere. Tampereen teknillinen yliopisto. 92 p.

Stackexchange, Data explorer [WWW] [Viitattu 31.3. 2014]. Saatavissa: http://data.stackexchange.com/stackoverflow/query/165851?Tag1=backbone.js&Tag2=angularjs&Tag3=meteor&Months=123

Trello – Organize anything, together. 2014 [WWW]. [Viitattu 28.4.2014]. Saatavissa www.trello.com.

TodoMVC - Helping you select an MV* framework. 2014 [WWW]. [Viitattu 28.4.2014]. Saatavissa https://github.com/tastejs/todomvc.

Overson J. ja Bailey D., Backbone.Marionette TodoMVC, 2014 https://github.com/tastejs/todomvc/tree/gh-pages/labs/architecture-examples/backbone_marionette

W3C, Session history and navigation, 2014 [WWW]. [Viitattu 6.5.2014]. Saatavissa http://www.w3.org/html/wg/drafts/html/master/browsers.html#history.