Pyry Mäkäräinen

Web-sovelluskehitys SPA-arkkitehtuurilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Mediatekniikan koulutusohjelma

Insinöörityö

26.5.2017

Metropolia

Tekijä Otsikko Sivumäärä Aika	Pyry Mäkäräinen Web-sovelluskehitys SPA-arkkitehtuurilla 42 sivua 26.5.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaaja	Lehtori Olli Alm

Insinöörityön tarkoituksena oli perehtyä web-sovelluskehitykseen SPA-arkkitehtuurilla (Single-page application) ja tavoitteena oli toteuttaa asiakasyritykselle web-palvelu, joka kokoaa kattavasti yhteen median ja viestinnän kannalta merkittäviä tapahtumia Suomesta ja ulkomailta. Työ aloitettiin tutustumalla perusteknologioihin web-sovellusten taustalla, minkä jälkeen syvennyttiin tarkemmin SPA-arkkitehtuurin pääpiirteisiin. Havainnollistamisen apuna hyödynnettiin vertailua perinteisiin jatkuvasti palvelinyhteydessä oleviin toteutuksiin. Työssä tutustuttiin myös web-sovelluksissa käytettyihin ohjelmistoarkkitehtuureihin ja niitä selainpäässä ilmentäviin JavaScript-ohjelmistokehyksiin.

Insinöörityössä kehitetty palvelu pyrkii käyttäjälähtöisesti vastaamaan viestinnän tekijöiden haasteisiin tarjoamalla selkeän, helppokäyttöisen ja luotettavan ympäristön tapahtumatiedon monipuoliseen tarkasteluun. Toteutettu tapahtumasovellus jakautuu useaan itsenäiseen mikropalveluun, jotka toimivat Amazonin pilvipalvelinympäristössä. Insinöörityössä käydään läpi mikropalveluista SPA-arkkitehtuuria hyödyntävää käyttöliittymäkerrosta.

Insinöörityöraportti toimii osin varsinaisen toteutuksen dokumentaationa, ja siksi käyttöliittymäkerroksen kehitystä kuvattiin käymällä tarkoin läpi käytetyt teknologiat ja joukko kehitystä helpottavia työkaluja. Sovelluksen selainpäässä toimiva logiikka toteutettiin hyödyntämällä JavaScript-kirjastoista Reactia ja Reduxia, joiden toimintaa kuvailtiin kattavasti esimerkkien kautta. Projektin onnistumisen kannalta sovelluksen modulaarisella rakenteella oli oleellinen rooli. Käyttöliittymäkerroksen SPA-arkkitehtuuri mahdollisti sovelluskehityksen aikana nopean reagoinnin asiakaspalautteeseen ja se helpottaa jatkossa uusien ominaisuuksien luomista sekä yleistä ylläpidettävyyttä.

Luotu tapahtumasovellus vastasi ensimmäistä tuotantokelpoista versiota, jonka avulla testattiin palvelun tarvetta markkinoilla. Tuote julkaistiin tammikuussa 2017, ja kohderyhmän keskuudessa se otettiin hyvin vastaan. Sovellus paransi selkeästi luotettavan reaaliaikaisen tapahtumatiedon läpinäkyvyyttä ja löydettävyyttä. Loppukäyttäjiltä tuli myös runsaasti positiivista palautetta palvelun ulkonäöstä, nopeudesta ja käytettävyydestä. Onnistuneesta toteutuksesta kertoo myös ehdokkuus vuoden 2017 parhaaksi yritystoimintaa edistäväksi sovellukseksi Grand One -kilpailussa.

Avainsanat	Web-sovellus, SPA, HTML5, JavaScript

Author	Pyry Mäkäräinen
Title	Web application development with SPA architecture
Number of Pages	42 pages
Date	26 May 2017
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructor	Olli Alm, Senior Lecturer

The purpose of this thesis was to study web application development with SPA (Single-page application) architecture and to develop a web-service which aggregates Finnish and foreign events from media and communications point of view. The subject was approached by researching basic web technologies behind web applications followed by more detailed analysis of SPA architectures basic principles. The thesis also examines web-application architectures and few JavaScript frameworks based on them.

The end product aims to meet the challenges of communication factors providing an easy to use platform for viewing event information. The application is divided into several independent microservices that runs on Amazon Web Services. This thesis goes through a front-end application layer which is based on SPA architecture.

Technologies and group of development tools are documented in this thesis to clarify the basic structure and principles of the front-end application. The business logic of the application presentation layer is implemented with React and Redux JavaScript libraries. The functionality of these libraries has been described with detailed examples that have been used in the application.

The application was released in January 2017 and it raised interest in the target group immediately. The product has improved the transparency and the availability of real-time event information. There is also a lot of positive feedback from end-users about performance and usability. The application is also nominated for the best Business to Business application in Grand One 2017 awards.

Keywords	Web application, SPA, HTML5, JavaScript

Sisällys

Lyhenteet

Ajax	*Asynchronous JavaScript And XML.* Joukko tekniikoita asynkronisen ohjelmoinnin mahdollistamiseksi.
CSS	*Cascading Style Sheets.* Merkintäkieli tyyliohjeiden määrittelyyn.
DOM	*Document Object Model.* Standardi dokumentin esittämiseen olioina.
HTML5	*Hypertext Markup Language 5.* HTML-merkintäkielen uusin versio.
HTTP	*Hypertext Transfer Protocol.* Tiedonsiirtoprotokolla.
JSON	*JavaScript Object Notation.* Tiedostomuoto tiedonvälitykseen.
MV*	*Model-View-Whatever.* MVC:stä johdettu ohjelmistoarkkitehtuuri.
MVC	*Model–view–controller.* Käyttöliittymän sovellustiedoista erottava ohjelmistoarkkitehtuuri.
MVP	*Minimum viable product.* Ensimmäinen tuotantokelpoinen versio, joka sisältää vain välttämättömimmät ominaisuudet.
npm	*node package manager.* Paketinhallintajärjestelmä Node.js-ympäristössä.
REST	*Representational state transfer.* Arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen.
RIA	*Rich Internet application.* Työ- ja mobiilisovelluksia muistuttavat web-sovellukset.
Sass	*Syntactically Awesome Style Sheets.* CSS-esikäsittelijä tyylimäärittelyiden monipuolisempaan kirjoittamiseen.
SPA	*Single-page application.* Yhdellä sivulatauksella toimiva monipuolinen web-sovellus.

URL *Uniform Resource Locator.* Internetissä olevan sivuston tai tiedoston ja sen käyttöön tarvittavan yhteyskäytännön yksilöivä tunnus.

W3C *World Wide Web Consortium.* WWW:n standardeja ylläpitävä ja kehittävä yhteisö.

1 Johdanto

Insinöörityön tarkoituksena on perehtyä web-sovelluskehitykseen SPA-arkkitehtuurilla (Single-page application) ja tavoitteena on toteuttaa asiakasyritykselle web-palvelu, joka kokoaa kattavasti yhteen median ja viestinnän kannalta merkittäviä tapahtumia Suomesta ja ulkomailta. Insinöörityö pyrkii myös tarjoamaan laajan katsauksen JavaScript-vetoiseen web-sovelluskehitykseen tarkastelemalla käsitteitä ja teknologioita selainpään toteutuksissa.

Insinöörityön aihetta käsitellään tutustumalla perusteknologioihin web-sovellusten taustalla, minkä jälkeen syvennytään tarkemmin SPA-arkkitehtuurin pääpiirteisiin. Havainnollistamisen apuna hyödynnetään vertailua perinteisiin jatkuvasti palvelinyhteydessä oleviin toteutuksiin. Tarkoituksena on myös perehtyä web-sovelluksissa käytettyihin ohjelmistoarkkitehtuureihin ja niitä selainpäässä ilmentäviin JavaScript-ohjelmistokehyksiin. SPA-sovellusten pohjana usein toimivista JavaScript-ohjelmistokehyksistä otetaan yksityiskohtaisempaan tarkasteluun AngularJS, Ember.js, React ja Vue.js. Työssä analysoidaan näiden teknologioiden hyviä ja huonoja puolia yleisellä tasolla.

Osana insinöörityötä on tarkoitus kehittää asiakasyritykselle palvelu, joka pyrkii käyttäjälähtöisesti vastaamaan viestinnän tekijöiden haasteisiin tarjoamalla selkeän, helppokäyttöisen ja luotettavan ympäristön tapahtumatiedon monipuoliseen tarkasteluun. Tavoitteena on luoda sovelluksesta ensimmäinen tuotantokelpoinen versio, minkä jälkeen tuotetta mahdollisesti lähdetään jatkokehittämään. Sovelluskehitystä pyritään ohjaamaan ketterän kehityksen keinoin ja asiakaslähtöisesti heti alkuvaiheista lähtien. Teknisesti palvelun tulee olla modulaarinen, helposti hallittava kokonaisuus, jonka jatkokehittäminen on tehty mahdollisimman vaivattomaksi. Insinöörityössä käsiteltävä toteutusosio sisältää käyttöliittymäkerroksen sovelluskehityksen keskittyen JavaScriptillä toteutetun toiminnallisuuden kuvaamiseen.

2 SPA-arkkitehtuuri

2.1 Web sovellusten alustana

Julkaisualustana alun perin alkanut web toimii kasvavassa määrin monipuolisten sovellusten toimintaympäristönä. Suosio perustuu pitkälti yksinkertaiseen arkkitehtuuriin, yleisesti käyttöönotettuun HTTP-protokollaan ja HTML:ään sekä laajalle levinneisiin toimintamalleihin. Nykyaikaiset selaimet muodostavat moniin muihin alustoihin verrattuna vahvasti standardisoidun ympäristön. Tässä ympäristössä sovellusten käyttö on mahdollista sijainnista riippumatta ilman erillisiä asennuksia tai manuaalisia päivityksiä. Selain toimiikin usein ensisijaisena julkaisuympäristönä uusille nykyaikaisille sovelluksille. [1, s. 1–2; 2.]

Web-sovellusten historia voidaan jakaa karkeasti kolmeen vaiheeseen. Ensimmäisessä vaiheessa sivustot olivat staattisia dokumentteja, jotka sisälsivät lähinnä tekstiä ja kuvia. Sivustolla liikuttiin hyperlinkkien kautta, jolloin uusi sivu ladattiin kokonaisuudessaan palvelimelta. Seuraavassa vaiheessa mukaan tulivat animaatiot, käyttäjien liikkeisiin reagointi ja erilaiset lisäosat. Ne loivat muun muassa mainostajille ja sisällöntuottajille uuden ympäristön, jossa kohdeyleisön saavutettavuus nousi uudelle tasolle. Teknologioiden kehittyessä JavaScript-ohjelmointikieli syntyi, ja Adobe Flashin, Quicktimen ja Shockwaven kaltaisten lisäosien avulla sisällöstä saatiin yhä monimuotoisempaa. Viimeisimmässä vaiheessa kehityssuunta on kääntynyt kohti sovellusmaista olemusta. Dynaamisesti luotu sisältö, asynkroniset palvelinpyynnöt ja sivuston sisäinen navigointi ovat nykyisin web-sovellusten perusominaisuuksia. Kolmannen vaiheen vahvasti sovellusmaisia toteutuksia kutsutaan usein nimellä Rich Internet Applications (RIA), joihin SPA-sovellukset myös kuuluvat. Vaikka kehitys on tuonut mukanaan uusia ominaisuuksia, ovat kaikki edellä esitellyt vaiheet yhä selvästi esillä nykyajan web-toteutuksissa. [1, s. 2–4.]

Ymmärtääkseen paremmin web-sovelluskehitystä ja sen luomia mahdollisuuksia on hyvä tutustua perusteknologioihin toteutusten taustalla. Seuraavissa alaluvuissa käydään läpi HTML5-merkintäkieltä ja siihen yleisesti liitettäviä rajapintoja sekä tyylimäärittelyiden luomista ja tutustutaan toimintalogiikan mahdollistavaan JavaScript-ohjelmointikieleen.

2.1.1 HTML5-merkintäkieli

HTML (HyperText Markup Language) on merkintäkieli, jota käytetään sisällön esittämiseen web-toteutuksissa. HTML5 on merkintäkielen uusin versio, jossa on panostettu erityisesti uusiin elementtityyppeihin ja ominaisuuksiin, jotka mahdollistavat monipuoliset sovellukset. Käsitteenä HTML5 mielletään yleisesti merkintäkielen ohella kokoelmaksi erilaisia ohjelmointirajapintoja, joiden avulla toteutuksien rakentaminen onnistuu aina yksinkertaisista sivuista laajoihin sovelluksiin. [2.]

HTML5:n uudet elementit ovat ymmärrettävästi merkintäkielen näkyvin uudistus. Ne kuvaavat verkossa esitettävän sivuston rakennetta ja niillä kaikilla on oma selkeä rooli. HTML5:n standardissa footer, header ja section ovat esimerkkejä uusista elementeistä, jotka selkeyttävät oleellisesti sivustojen rakennetta [3]. Uusien elementtien myötä esteettömyys nousee myös paremmalle tasolle. Palveluiden digitalisoituessa on tärkeää huomioida erityisryhmiin kuuluvat ihmiset, joita arvellaan olevan noin 10–20 % internetin käyttäjistä. Sisällön määrittely elementtien avulla mahdollistaa myös heille palveluiden käytön. [4; 20, s. 165.] Selkeyden ohella uudet mediaelementit ovat keskeinen osa standardin uudistuksia. Video- ja audioelementit poistavat tarpeen lisäosille, jotka usein vaativat erillisen asennuksen ja sisältävät joukon turvallisuusuhkia [5, s. 65–69]. Ohjelmallisesti luodun graafisen sisällön luominen onnistuu standardin tuoman canvas-elementin avulla, joka sopii erityisen hyvin animaatioiden ja interaktiivisen sisällön luomiseen [5, s. 25]. Sovellusmaista käyttökokemusta täydentää monista työpöytäsovelluksista tuttu elementtien raahaus- ja pudotusominaisuus (HTML Drag and Drop) [6].

HTML5 tuo mukanaan myös joukon erilaisia ohjelmointirajapintoja, jotka nostavat websovellusten mahdollisuudet uudelle tasolle. Lähtökohdat sovelluskehitykseen ja HTML:n rajapintoihin tarjoaa DOM (Document Object Model), joka on HTML-, XML- ja SVG-dokumenteille kehitetty rajapinta. DOM luo sivustosta mallin, jossa jokainen elementti on esitettynä. DOM:n tarjoaman rajapinnan kautta yksittäisten elementtien rakennetta, tyyliä ja sisältöä päästään muokkaamaan. Vaikka yleisimmin muokkaus tehdään JavaScript-ohjelmointikielen avulla, ei DOM ota kantaa käytettävään käsittelytapaan. [7.]

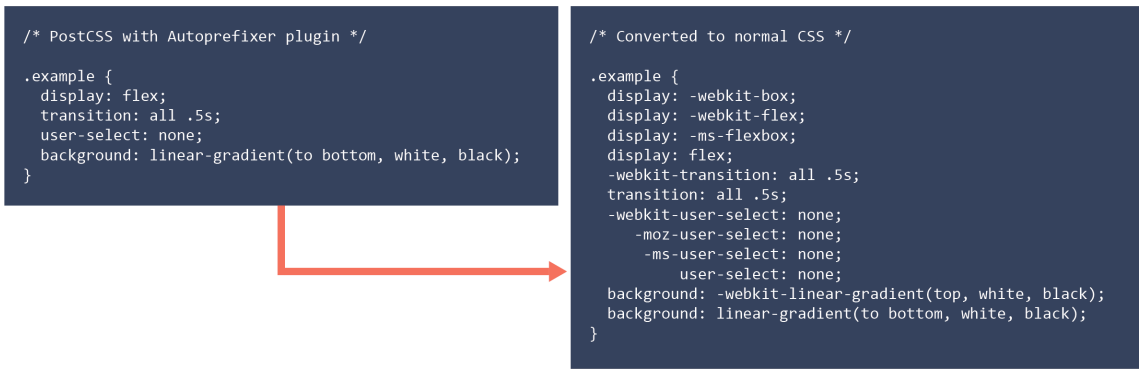
Standardisoitujen rajapintojen määrä on nyt jo suuri ja uusia syntyy hiljalleen jatkuvasti. Sovellusmaista käyttökokemusta parantavat monet mobiili- ja työpöytäsovelluksista jo löytyvät ominaisuudet, kuten pääsy kameraan, sijaintiin tai laitteen tiedostojärjestelmiin.

Myös selainympäristöä ja JavaScript-ohjelmointikieltä edistäviin uudistuksiin on panostettu. Niihin kuuluvat muun muassa web-sovelluksien taustaprosessit mahdollistava Web Workers, reaaliaikaisen tiedon välittämistä helpottava WebSocket ja tiedon paikallisen säilyttämisen mahdollistava Web Storage. [8.]

2.1.2 CSS-merkintäkieli

CSS (Cascading Style Sheets) on W3C:n ylläpitämä tyylien määrittämiseen tarkoitettu standardisoitu merkintäkieli. Sen avulla tyylimäärittelyt saadaan eriytettyä muusta sisällöstä omaksi kokonaisuudeksi. CSS on jaettu tasoihin, joiden perustella uudet ominaisuudet julkaistaan. Suurin osa uusista ominaisuuksista on vielä tason 3 (CSS Level 3) ominaisuuksia, joissa on painotettu erityisesti animaatioita ja ulkoasun responsiivisuutta. Uusimmassa neljännessä tasossa mukaan tuodaan joukko erilaisia valitsimia, joiden avulla kehittämistä pyritään helpottamaan ja monipuolistamaan. Jatkuvasti kehittyvä kielen ominaisuudet ovat yhä jatkuvassa kehityksessä ja selaintuki tästä syystä vielä hyvin vaihtelevaa. [9.]

CSS:n tuottamista helpottamaan on kehitetty esiprosessoituja CSS-merkintäkieliä, joiden tarkoituksena on tehdä tyylimäärittelyiden tekemisestä dynaamisempaa, organisoidumpaa ja tuottavampaa. Suosituimpia CSS:n esikäsittelijöitä ovat Sass, Less ja Stylus. Kielet kirjoitetaan omalla syntaksillaan, joka julkaisuvaiheessa käännetään CSS:ksi. Tämän vuoksi CSS:n lainalaisuudet eivät päde esikäsittelijöihin, jolloin niissä voidaan käyttää esimerkiksi muuttujia, funktioita ja tyylimäärittelyjä kokoavia luokkia (CSS mixins). [10.] Tyylitiedostojen käsittelyn uusimpia suuntauksia ilmentää PostCSS, joka mahdollistaa tyylien muokkaamisen JavaScriptillä. Lisäosapohjaisen järjestelmän avulla tyylejä voidaan validoida, niihin voidaan automaattisesti lisätä vanhempien selaimien vaatimia ominaisuuksia tai esimerkiksi kääntää neljännen tason CSS-ominaisuudet nykyselaimien ymmärtämään muotoon. Kuvassa 1 on PostCSS:n Autoprefixer-lisäosan avulla täydennetty CSS-syntaksia, jotta ominaisuudet saadaan toimintaan mahdollisimman monessa selainympäristössä. [11.]



Kuva 1. PostCSS mahdollistaa lisäosien kautta muun muassa selainkohtaisten tyylimäärittelyiden automaattisen lisäämisen.

2.1.3 JavaScript-ohjelmointikieli

JavaScript on erittäin laajasti käytetty ohjelmointikieli, jota käytetään pääosin moderneissa selaimissa. Se on dynaamisesti tyypitetty korkean tason ohjelmointikieli, joka sopii olio- ja funktionaaliseen ohjelmointiin. Yleisesti JavaScriptiä käytetään web-sivustoissa ulkonäön ja toiminnallisuuksien muokkaamiseen käyttäjän liikkeiden perusteella. [12.] Kielen kehitti Netscapella työskennellyt Brendan Eich vuonna 1995. JavaScript on Java-ohjelmointikielestä täysin eroava tekniikka, ja yhtenevä nimi olikin ainoastaan osa markkinointikampanjaa, jonka Netscape toteutti liittouduttuaan Java-kielen kehittäneen Sun Microsystemsin kanssa vuonna 1995. Tekijänoikeudellisista syistä standardisoidun kielen viralliseksi nimeksi tuli ECMAScript European Computer Manufacturers Association päätöksellä vuonna 1997. Vakiintunut nimitys ohjelmointikielelle on kuitenkin JavaScript ja siksi sitä myös pääosin kutsutaan tässä insinöörityössä. [13, s. 1–2.]

JavaScriptin nykyisen laajan suosion pohja syntyi 2000-luvun vaihteessa, jolloin Microsoft lisäsi XMLHttpRequest-ominaisuuden Internet Explorer -selaimeen. Tämän ominaisuuden avulla Outlook-sähköpostiohjelma pystyi hakemaan palvelimelta dataa ilman sivulatausta. Nykyisin Ajaxina (Asynchronous JavaScript and XML) tunnettu ominaisuus sisällytettiin nopeasti myös muiden selainvalmistajien tuotteisiin. Innovaatio mahdollisti monipuolisten työpöytäsovellusten kaltaisten web-sovellusten synnyn. Hiljalleen JavaScriptin käyttö sivuilla kasvoi, mikä pakotti selainvalmistajat optimoimaan selainten JavaScriptin-moottoreita. JavaScript-moottorien tehtävä on kääntää (compile) ja tulkata (interpret) JavaScriptilla kirjoitettu toteutus, jolloin mahdollistetaan koodin suoritus (execute) alustan ymmärtämällä kielellä. Tämä mahdollisti yhä monipuolisempien sivustojen rakentamisen, mikä edesauttoi myös HTML- ja CSS-kielien kehitystä. [14.]

Vaikka perinteisesti JavaScript ajetaan selaimessa, ei ajatus saman ohjelmointikielen käyttämisestä palvelinpuolen toteutuksissa ole uusi. Netscape käytti JavaScriptiä jo omalla Enterprise HTTP-palvelimellaan vuonna 1996, mutta laajempaan käyttöön se ei vielä tuolloin levinnyt. Selainmoottorien kehityksen myötä idea nousi yli kymmenen vuotta myöhemmin taas ajankohtaiseksi. Googlen avoimen lähdekoodin V8 JavaScript-moottoriin perustuva Node.js julkaistiin vuonna 2009, ja sen myötä myös palvelinpuolen toteutuksissa pystytään hyödyntämään JavaScriptiä. [15.]

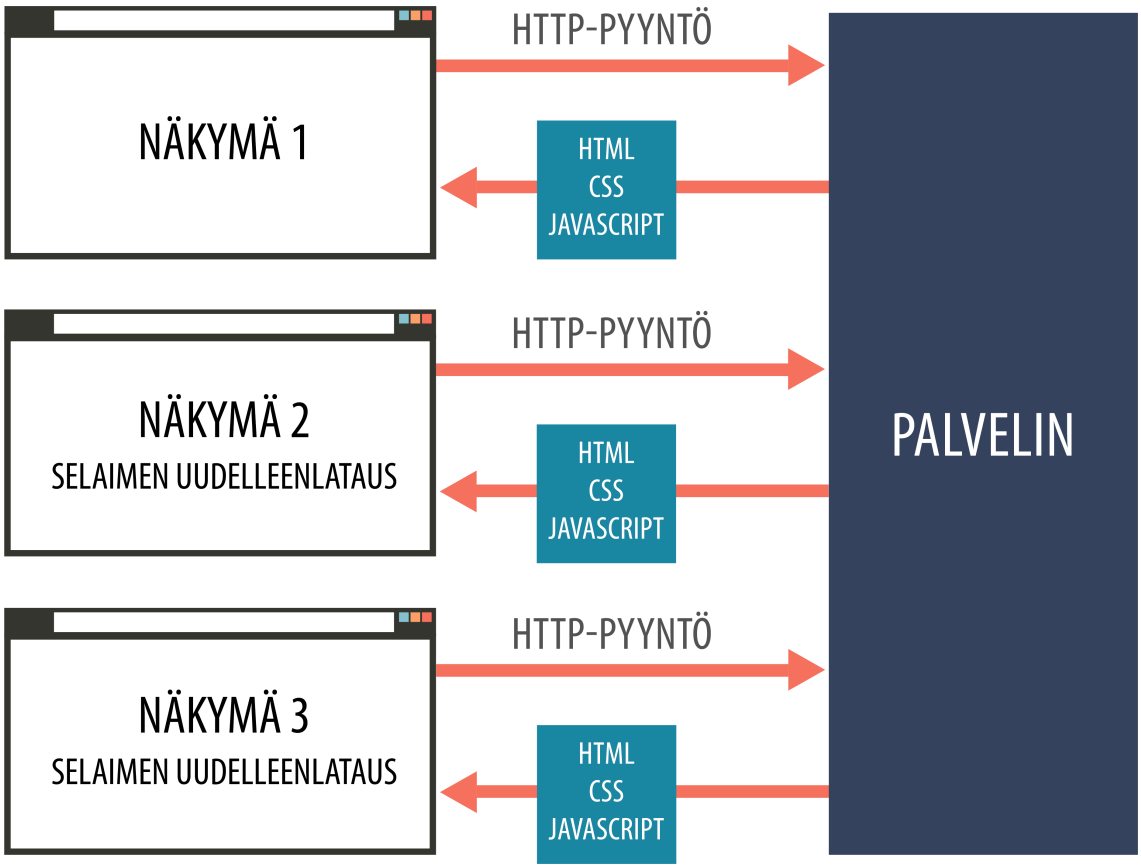
JavaScriptin käytön kasvaessa myös kieli itsessään on viime vuosina kehittynyt paljon. Standardisoidun JavaScriptin, ECMAScriptin, kuudes versio on laajin päivitys standardiin sitten ensimmäisen version julkaisun. Virallisesti kesäkuussa 2015 julkaistu versio tunnetaan yleisesti nimillä ECMAScript 2015, ECMAScript 6 ja ES6. Kuudes versio toi mukanaan uusitun syntaksin monipuolisten web-sovellusten toteuttamiseen. Merkittävimpinä ominaisuuksina voidaan pitää monissa muissakin ohjelmointikielissä yleisesti käytössä olevia luokkia (Classes) ja moduuleita (Modules), asynkronisten kutsuille annettavia lupauksia (Promises), vakioarvojen antamista funktioiden parametreille (Default parameters) ja olioliteraalien paranneltua syntaksia (Enhanced Object Literals). [16.]

2.2 SPA-arkkitehtuurin pääpiirteet ja periaatteet

Single-page application (SPA) on sovellus, joka toimii selaimessa yhdellä sivulatauksella ja tuo käyttäjälle nopean ja sulavan käyttökokemuksen. Käyttäjäystävällisyyden parantuminen perustuu toimintalogiikan siirtämiseen palvelimelta selaimeen, jolloin ensimmäisen sivulatauksen yhteydessä ladatuilla HTML-, CSS- ja JavaScript-tiedostoilla voidaan luoda kaikki sovelluksen näkymät ja toiminnot ilman sivuston uudelleenlatausta. Tämä tekee näkymänvaihdoksista nopeita ja sisällöstä usein vahvasti dynaamista. Logiikkaa täydennetään tarvittaessa asynkronisilla kutsuilla, joiden avulla SPA-sovellus on yhteydessä palvelimella sijaitseviin tietoihin rajapinnan kautta. SPA-arkkitehtuuri perustuukin nämä ominaisuudet mahdollistaviin tekniikoihin ja toimintatapoihin, jotka käyttäjäystävällisyyden ohella usein selkeyttävät toteutuksen rakennetta ja parantavat sovellusten skaalautuvuutta. [17, s. 6–7; 18, s. 11–13.]

Perinteiset verkkosivut ja -sovellukset ovat lähtökohtaisesti jatkuvassa yhteydessä palvelimeen. Useimmat käyttäjien toiminnat aiheuttavat HTTP-pyynnön, jonka aikana data kulkee eri tasojen läpi muodostaen palvelimella selaimeen palautettavan HTML-sivun.

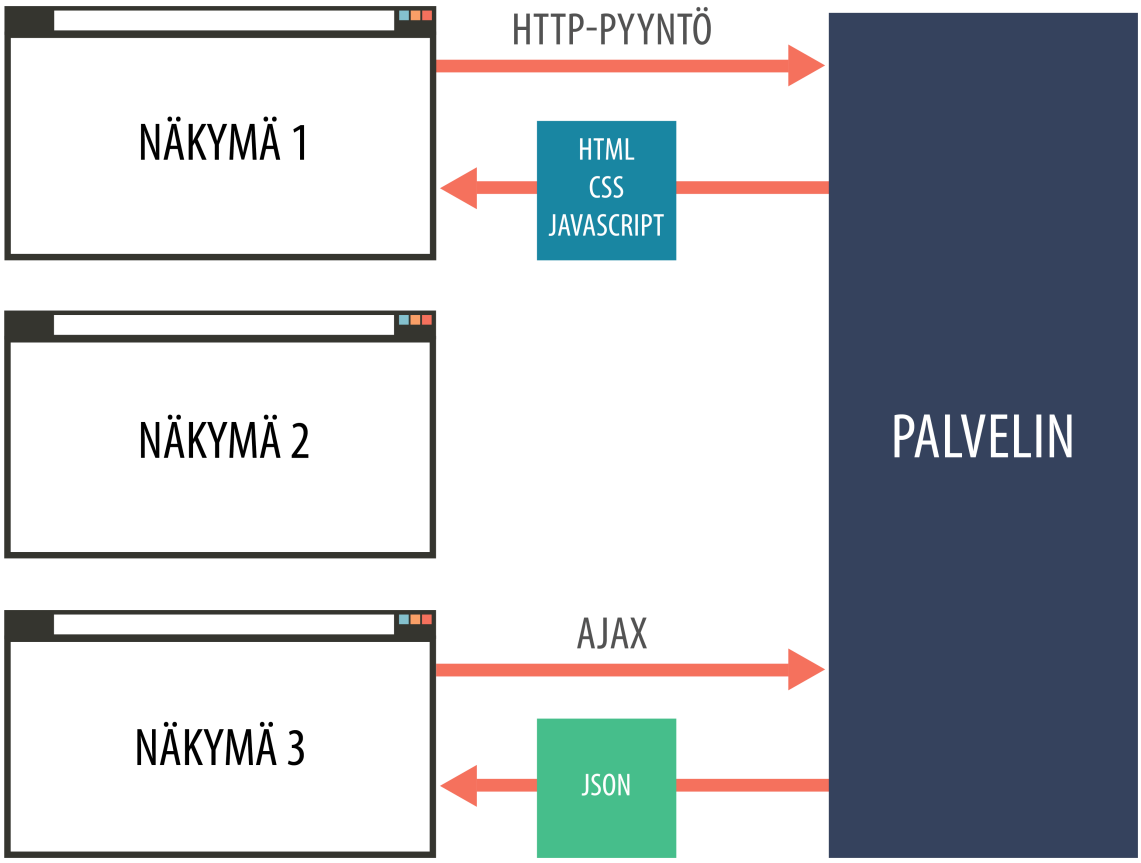
Valmis HTML-sivu lähetetään selaimeen, jolloin sivulatauksen yhteydessä vanha sivu korvataan täysin uudella. [17, s. 5–6.] Tässä perinteisessä mallissa selainta käytetään vain näkymien muodostamiseen valmiin HTML:n pohjalta. Myös JavaScriptin rooli kyseisen mallin toteutuksissa on usein varsin pieni: se kattaa ainoastaan pienet käyttöliittymämuutokset ja animaatiot. Koska toimintalogiikan sijaitsee palvelimella, sovelluksen tilaan perustuvat tapahtumat vaativat aina kierroksen palvelinpuolella. Käyttäjäystävällisyyteen tällä on merkittävä vaikutus, koska sovelluksen käyttö keskeytyy aina näkymää vaihdettaessa. Tällainen toiminta on myös täysin riippuvainen koko käytön ajan toimivasta internetyhteydestä, kuten on nähtävissä perinteisen verkkosivun toimintaa esittävässä kuvassa 2 [18, s. 6–8.]



Kuva 2. Perinteisissä sivustoissa näkymien päivittäminen vaatii aina erillisen palvelinpyynnön ja selaimen uudelleenlatauksen [18, s. 6–8].

SPA-sovellusten ero perinteisiin, useilla HTTP-pyynnöillä toimiviin toteutuksiin on merkittävä. Toimintalogiikan siirto selaimeen vapauttaa sovellukset toimintaa hidastuttavista HTTP-pyynnöistä ja mahdollistaa työpöytä- ja mobiilisovellusten kaltaisten sovelluksien toteuttamisen. Tässä apuna toimii JavaScript, joka kehittynyt laaja-alaiseksi toimintalo-

giikan apuvälineeksi. Kuvassa 3 on havainnollistettu SPA-sovellusten toimintaperiaatetta, jossa ensimmäisen sivulatauksen jälkeen kaikki sovelluksen tarvittavat rakennuspalikat ovat käytettävissä, jolloin vain dataa välitetään Ajaxilla palvelimen ja selaimen välillä. Pelkästään dataa sisältävät palvelinpyynnöt tekevät kutsuista nopeita ja kevyitä. Näkymien luonnin siirtyminen selainpäähän paitsi vauhdittaa sivuston toimintaa myös vapauttaa palvelinresursseja. Selaimen paikallista laskentatehoa pystytään hyödyntämään, ja näin käyttäjälle pystytään tarjoamaan entistä miellyttävämpi käyttökokemus. Selain- ja palvelinpään erottaminen toisistaan tekee myös niiden toisistaan riippumattomasta kehittämisestä yksinkertaisempaa. Selainpään sovelluskehityksessä apuna käytetään usein JavaScript-ohjelmistokehyksiä, jotka mahdollistavat eri ohjelmistoarkkitehtuurimallien avulla selkeämmän ja modulaarisemman toteutustavan sovelluskehityksen tueksi. [17, s. 6–13.]



Kuva 3. SPA-sovelluksissa kaikki sovelluksen näkymät voidaan muodostaa ensimmäisen sivulatauksen yhteydessä ladatuilla tiedostoilla, minkä jälkeen vain dataa siirretään selaimen ja palvelimen välillä.

Yhdellä sivulla toimivat sovellukset tuovat sovelluskehitykseen omat haasteensa. Sivustojen muodostaminen selainpäässä, näkymien dynaamisuus ja JavaScript-painotteisuus hankaloittavat hakukoneiden toimintaa. Sivustot saattavat indeksoitua väärin tai jäädä tyystin hakukoneiden ulottumattomiin. Ongelma on mahdollista kiertää esimerkiksi muodostamalla toteutuksesta erillinen palvelinversio, jolloin hakukoneet pääsevät sivustoon ja sen sisältöön tavalliseen tapaan käsiksi. [19.] Google on myös reagoinut SPA-sovellusten kasvavaan määrään lisäämällä sivuja käsittelevään algoritmiinsa tuen JavaScriptin suorittamiselle [20]. Tämä on sovelluskehityksen kannalta merkittävä kehityssuunta, joka mahdollistaa aikaisempaa paremman hakukoneoptimoinnin SPA-sovelluksille. Ongelmia sovelluksiin aiheuttaa lisäksi sivuston sisäinen navigointi, tilanhallinta ja toteutusten monimutkaisuus. Näihin löytyy nykyisin kuitenkin valmiita ratkaisumalleja, joihin perehdytään tarkemmin tässä insinöörityöraportissa.

2.2.1 Ajax-teknologia

Tärkein yksittäinen teknologia SPA-sovellusten taustalla on Ajax (Asynchronous JavaScript and XML), jonka myötä sisältöä pystytään päivittämään ilman selaimen uudelleenlatausta. Ajax on kokoelma tekniikoita, joihin kuuluvat HTML, CSS, DOM, XMLHttpRequest-objekti ja JavaScript. XMLHttpRequest-objektin avulla tehdään palvelinpyynnöt, HTML:n ja CSS:n kautta data esitetään, DOM mahdollistaa dynaamisen esittämisen sekä manipuloinnin ja JavaScript hoitaa toimintalogiikan eri toimintojen välillä. Nykyselaimet tukevat hyvin kaikkia näitä elementtejä. [21.] Alun perin XML-muotoinen data on pitkälti korvautunut JSON:lla (JavaScript Object Notation). JSON on nimestään huolimatta ohjelmointikieliriippumaton tietomuoto, joka muodostuu helposti luettavista avain-arvopareista. JSON:n ja XML:n ohella data voidaan siirtää myös pelkkänä tekstinä tai esimerkiksi HTML-muodossa. [22.]

Toiminnot, jotka aikaisemmin olivat mahdollisia vain erillisten HTTP-pyyntöjen kautta, voidaan suorittaa kutsumalla JavaScriptillä Ajax-moottoria. Ajax käsittelee pyynnöt asynkronisesti, mikä tuo merkittävän eron perinteisiin verkkosovelluksiin nähden. Perinteisissä palvelinpyynnöissä käyttäjän toiminta on estynyt selaimen hakiessa uutta sisältöä palvelimelta. Ajax mahdollistaa sivuston käyttämisen pyynnön aikana, mikä kasvattaa verkkototeutusten käyttäjäystävällisyyttä runsaasti. Esimerkiksi sähköpostiohjelmissa käyttäjää voidaan informoida viestin onnistuneesta lähetyksestä, karttasovelluksessa tietoa hakea käyttäjän liikkeiden perusteella tai hakutuloksia etsiä ilman erillistä painallusta. Ajaxin avulla voidaan myös käsitellä useita pyyntöjä yhtäaikaisesti, mikä

mahdollistaa monipuolisemmat interaktiot sivuilla ja sovelluksissa. Palvelimen ja selaimen välinen liikenne myös usein supistuu, kun dataa liikkuu kerralla vähemmän [21; 23].

Jo yli vuosikymmenen käytössä olleen XMLHttpRequest-objektin tapa sisällyttää kaikki tarvittavat tiedot yhden objektin alle on nykyään jo osin vanhentunut. Modulaarisuuden puute heikentää toiminnan selkeyttä erityisesti laajoissa palvelinpyynnöissä. Myös tapahtumapohjainen malli on väistymässä lupauksiin perustuvan mallin tieltä asynkronisessa ohjelmoinnissa. Tätä suuntausta helpottamaan on kehitetty Fetch-rajapinta, joka tarjoaa yhtäläiset toiminnallisuudet XMLHttpRequest-objektin kanssa tehokkaammassa ja skaalautuvammassa muodossa. Rajapinta on myös täysin lupauspohjainen, mikä helpottaa HTML5 Service Workereiden mahdollistamien säikeiden käyttöä. Fetch-rajapinnan standardointityö on kehityksen alla, ja siihen on odotettavissa vielä parannuksia. [23.]

2.2.2 REST-arkkitehtuurimalli

REST (Representational State Transfer) on ohjelmointirajapintojen toteuttamiseen tarkoitettu arkkitehtuurimalli. REST-arkkitehtuurin on tarkoitus parantaa rajapintojen suorituskykyä, skaalautuvuutta ja luotettavuutta [24].

REST:n perusperiaatteisiin kuuluvat resurssit, joita voivat olla dokumentti, kuva, informaatio tai muut vastaavan kaltaiset tiedot. Arkkitehtuurin ideana on luoda palvelu, josta halutut resurssit ovat helposti saatavilla esimerkiksi teksti-, XML- tai JSON-muodossa yksilöidyn URL-osoitteen kautta. Tällaista palvelua kutsutaan nimellä RESTful. RESTful-mallinen palvelu tarjoaa rajapinnan, jota käytetään URL- ja HTTP-metodien avulla. HTTP-pyynnössä määritellään haluttu toiminto ja URL-osoitteessa toimintoon vaikuttavat yksilölliset tekijät. HTTP-protokollaan pohjautuva palvelu tarjoaa metodit tiedon lisäämiseen, lukemiseen, päivittämiseen ja poistamiseen kuvan 4 mukaisesti. [17, s. 172–174.]

RESURSSI	POST LUO / CREATE	GET LUE / READ	PUT PÄIVITÄ / UPDATE	DELETE POISTA / DELETE
/TUOTE	LUO UUSI	LISTAA TUOTTEET	PÄIVITÄ TUOTTEET	POISTA TUOTTEET
/TUOTE/1234	VIRHE	NÄYTÄ TUOTE1234	JOS LÖYTYY: PÄIVITÄ TUOTE 1234 JOS EI LÖYDY: VIRHE	POISTA TUOTE 1234

Kuva 4. RESTful-mallisissa palveluissa tietoa käsitellään HTTP-protokollan metodien avulla.

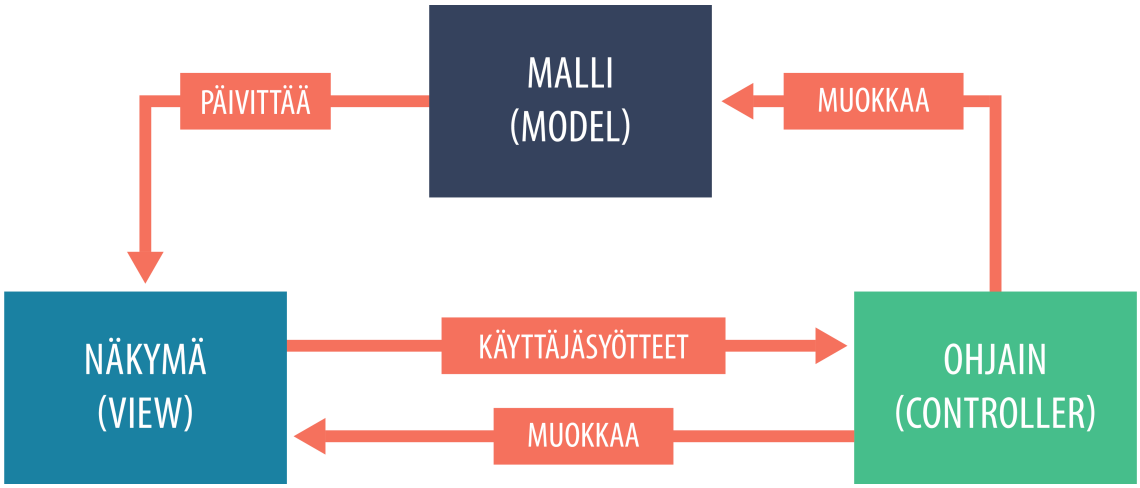
RESTful-mallinen palvelu tarjoaa SPA-sovelluksille yksinkertaisen pääsyn resursseihin ilman tarvetta huolehtia palvelinpuolen toteutuksesta. Toiminnallisuuksien jako käyttöliittymä- ja palvelinpäähän mahdollistaa yksinkertaisemmat komponentit ja näin tekee monipuolisten toteutusten kehittämisestä helpompaa. Selkeä jako mahdollistaa toisistaan riippumattoman kehityksen ja palvelinpuolen toteutuksen hyödyntämisen myös muissa alustoissa, esimerkiksi mobiilisovelluksissa. Arkkitehtuurimallissa jokaisessa pyynnössä on tultava ilmi kaikki kutsuun vaikuttavat tekijät, koska tilaa ei tallenneta palvelimelle koskaan. Tämä yksinkertaistaa palvelun rakennetta, parantaa luotettavuutta ja lisää skaalautuvuutta. Haittana ilmenee verkon kuormittumista, jota pyritään kompensoimaan arkkitehtuurimallissa muun muassa tallentamalla dataa selaimen välimuistiin. [24.]

2.2.3 Modulaarisuus

Sovelluskehitys sisältää usein laajoja kokonaisuuksia, joiden hallitseminen voi muodostua hankalaksi, jos rakennetta ei ole mietitty tarkkaan tai se muuttuu kehityksen aikana. Olennaista rakenteen selkeyden kannalta on toteutuksen pilkkominen itsenäisesti toimiviin osioihin, joissa jokaisella on selkeä oma rooli. Kommunikaatio kahden eri osion välillä tapahtuu selkeästi määritellyn rajapinnan kautta. Huolellisesti toteutettu jako mahdollistaa toisista riippumattoman osioiden itsenäisen kehittämisen, mikä helpottaa huomattavasti ylläpidettävyyttä ja testattavuutta. Eri ohjelmistokehysten käyttö ajaa kehitystä usein modulaariseen suuntaan käytetyn mallin ehdoilla. [17, s.45–48.]

Malli-näkymä-ohjain (MVC) on web-sovelluksissa paljon käytetty ohjelmistoarkkitehtuuri, jolla eri tasot erotetaan toisistaan. Kuvan 5 mukaisesti MVC:ssä malli sisältää järjestelmään tallennetun datan, näkymä määrittää käyttöliittymän ja ohjain käsittelee komennot

ja muokkaa mallia ja näkymää niiden mukaan. Osiot ovat siis keskenään yhteydessä, mutta erillisiä kokonaisuuksia. Jaottelu mahdollistaa eri osa-alueiden muokkaamisen ilman suoraa vaikutusta toiseen osioon. Tällöin voidaan esimerkiksi saman mallin pohjalta luoda erilliset käyttöliittymät mobiili- ja selainsovelluksiin. [25, s. 114.]

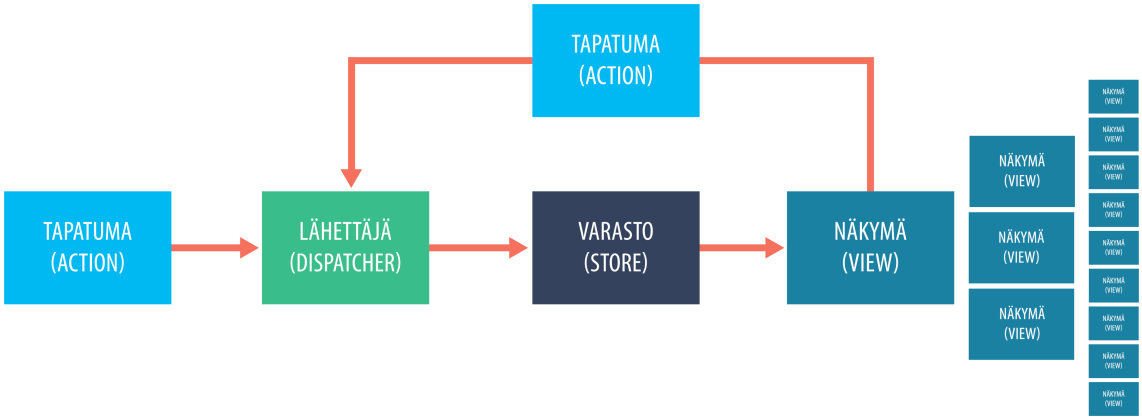


Kuva 5. MVC-ohjelmistoarkkitehtuuri erottaa käyttöliittymän sovellustiedoista.

Toimintalogiikan sijainti eri toteutuksissa ei usein täysin vastaa MVC:n mukaista ajattelutapaa. Käsittelijän muuttuneen roolin myötä MVC:n rinnalle on kehittynyt uusia arkkitehtuurimalleja sovelluksen rakennetta jakamaan. Tunnetuimmat näistä ovat MVP (Model-View-Presenter) ja MVVM (Model-View-ViewModel). MVP:ssä tarkoitus on eriyttää malli vahvemmin kahdesta muusta komponentista korvaamalla käsittelijä esittäjällä (presenter). Tässä arkkitehtuurimallissa näkymä on passiivinen ja sen toimintalogiikka on siirretty esittäjälle. Esittäjä muokkaa sekä näkymää että mallia ja huolehtii toimintojen ohjaamisesta. Microsoftin esittelemässä MVVM-ohjelmistoarkkitehtuurissa vastaavasti näkymä sisältää osan toimintalogiikasta. Näkymä pitää yhteyden datasidonnalla näkymä-malliin, joka käsittelee ja ohjaa toimintoja mallin ja näkymän välillä. MVVM:ssä näkymällä ei ole kuitenkaan MVC:stä poiketen suoraa yhteyttä malliin. SPA-sovelluskehityksessä yleisesti käytetyt ohjelmistokehykset hyödyntävät usein edellä esiteltyjä malleja soveltaen, minkä vuoksi monesti puhutaan yleisesti MV*-arkkitehtuurimallista. Tyypillisesti näissä malli ja näkymä ovat selkeämmin esillä ja tähti edustaa kulloinkin vallitsevaa toimintalogiikkaa, joka voi vaihdella runsaastikin. [17, s. 27–29; 25, s. 114.]

MV*-mallissa kaksisuuntainen kommunikointi eri osioiden välillä ja sen vaikutus koko toteutuspohjaan voi laajoissa toteutuksissa aiheuttaa epäjohdonmukaisuuksia, joita on

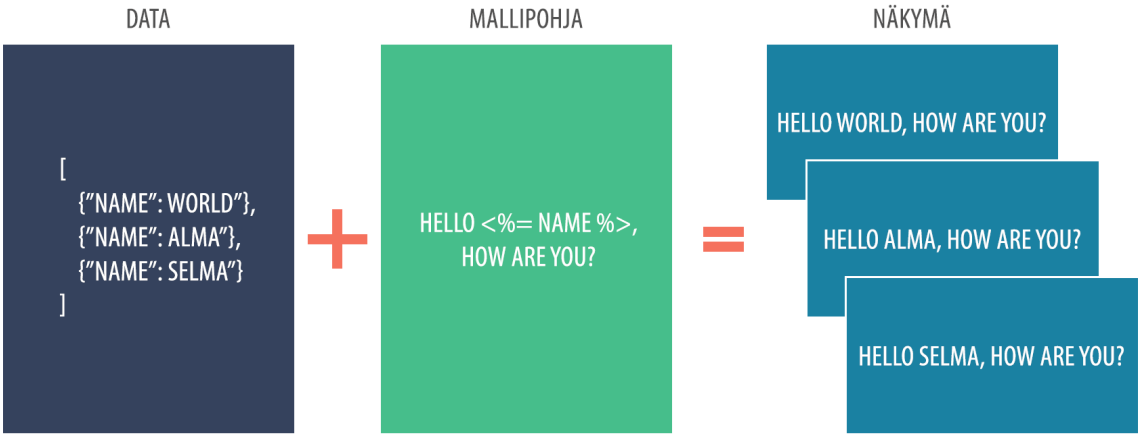
hankala jäljittää ja korjata. Tätä ongelmaa ratkaisemaan Facebook kehitti yksisuuntaiseen tiedonsidontaan pohjautuvan Flux-arkkitehtuurin. Flux jakautuu kolmeen päätekijään, jotka ovat lähettäjä (dispatcher), varasto (store) ja näkymä (view). MVC-arkkitehtuurista poiketen käsittelijät sijaitsevat näkymien sisällä. Näkymät muodostavat puumallisen kokonaisuuden, joka saa datansa yleisestä varastosta. Varastoon tapahtuneet muutokset välitetään puumalliselle näkymän ylätasolle ja sen kautta eteenpäin mahdollisille alinäkymille. Näkymät sisältävät myös joukon erilaisia apumetodeita, joilla saadaan yhteys tilaa muuttavaan lähettäjään ja sitä kautta varastoon. Kuvassa 6 on esitetty Fluxin päätekijät lähettäjä, varasto ja näkymä sekä datavirtauksen käynnistävät apumetodit (actions). Flux-arkkitehtuuri sopii hyvin komponenttipohjaisiin toteutuksiin, joissa data virtaa sovelluksen läpi yhteen suuntaan. [26.]



Kuva 6. Flux-ohjelmistoarkkitehtuuri on suoraviivainen tapa dataan perustuvien web-sovellusten toteuttamiseksi.

2.2.4 Datasidonta

Datasidonta (Data Binding) yhdistää käyttöliittymän elementit toimintalogiikkaan. Näin monimutkaisissakin sovelluksissa riippuvuussuhteiden ylläpito yksinkertaistuu ja kirjoitettavan koodin määrä usein vähenee. Sidottava data voi sisältää tekstin lisäksi esimerkiksi tyylejä, attribuutteja tai erilaisia toimintoja. Datasidonta on yleisesti käytössä eri ohjelmistokehyksissä, mutta sen ominaisuudet vaihtelevat kehyksestä riippuen. [17, s. 36–37.] Kuvassa 7 listamuotoinen objekteja sisältävä data on sidottu mallipohjaan, jonka avulla saadaan luotua joukko erilaisia näkymiä.



Kuva 7. Yhdistämällä data uudelleenkäytettäviin mallipohjiin saadaan luotua erilaisia näkymiä.

Datasidonta on mahdollista toteuttaa kaksi- tai yksisuuntaisesti tai kertaluontoisesti. Kaksisuuntaisessa sidonnassa muutokset näkymässä heijastuvat malliin ja vastaavasti mallin näkymään. Tämä pitää molemmat osapuolet päivitettyinä ilman tarvetta asettaa erillisiä tapahtumakutsujia. Kertaluontoisessa sidonnassa side pysyy yhden muutoksen ajan mallista näkymään, minkä jälkeen yhteys katkeaa. Yksisuuntaisessa sidonnassa data kulkee vain samaan suuntaan, jolloin muutokset näkymässä eivät automaattisesti päivitä sovelluksen tilaa. Tällöin tilaan käsiksi pääseville funktioille on tarvetta. Suorituskyvyltään yksisuuntainen yhteys on yleensä kaksisuuntaista tehokkaampi, mikä ilmenee käyttäjälle näkymien nopeampana päivittymisenä. [17, s. 37–39.]

2.2.5 Reititys

Perinteisissä sivustoissa liikutaan yleisesti URL-osoitteiden kautta, jolloin uusi sisältö haetaan palvelimelta. Koska SPA-sovellukset toimivat yhdellä sivulla, näkymänvaihdokset tapahtuvat selaimessa eivätkä ne vaadi URL-osoitteen päivittymistä tai uutta sivulatausta. Käyttäjä usein kuitenkin olettaa pystyvänsä liikkumaan tavallisen sivuston tavoin selaimen navigointinäppäimillä ja siirtymään haluttuun kohtaan URL-osoitteen kautta. Nämä ominaisuudet mahdollistamaan on tehty selainpään reitityksiä, joiden avulla navigaatio SPA-toteutusten sisällä tapahtuu. [17, s. 86–87.]

SPA-sovelluksissa reititys hallitsee sovelluksen tilamuutoksia, joiden perusteella se pitää navigointinäppäimet ja URL-osoitteen toiminnassa ja ajan tasalla. Reititys muuttaa URL-osoitteen halutuiksi toiminnoiksi ja näkymiksi, jolloin sovelluksen tila, tai osa siitä, voi-

daan tallentaa URL-osoitteella kirjanmerkkeihin tai lähettää linkkinä eteenpäin. Reitityksen perustehtäviin kuuluu myös perusreittien määritteleminen, jos osoitepalkkiin syötettyyn linkkiin ei löydy reitittimestä vastinetta. Näiden lisäksi eri reititykseen kuuluu usein joukko lisäominaisuuksia, jotka ovat yleensä vahvasti kytköksissä käytettyyn ohjelmistokehykseen tai reitityksen mahdollistaviin lisäkirjastoihin. [17, s. 87–94.]

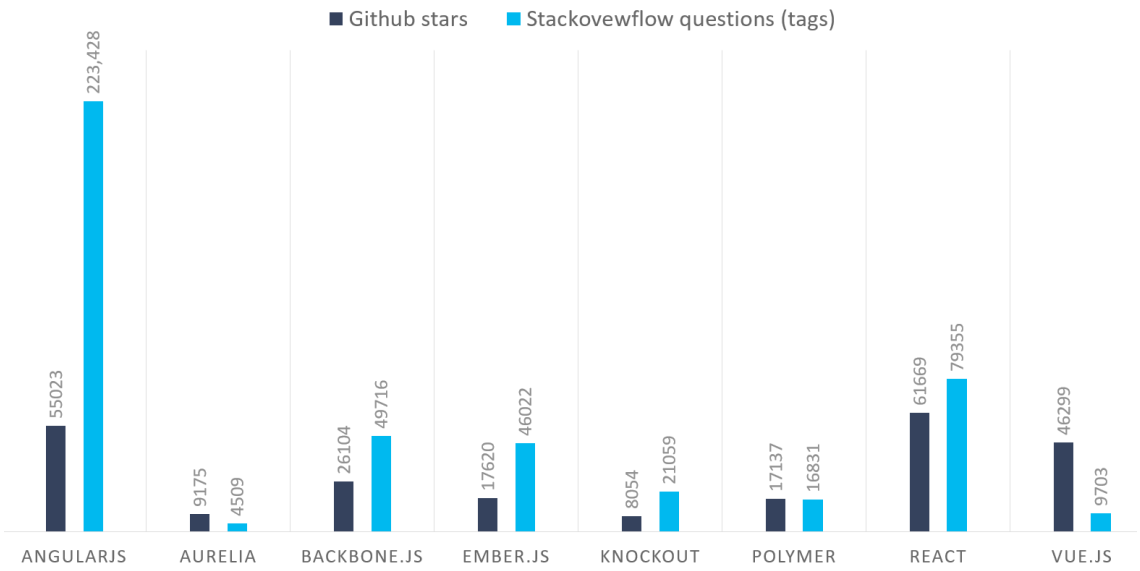
Selainpään reititys hyödyntää yleisesti joko URL-osoitteen erillistä tunnistinta tai HTML5 History -rajapintaa. Molemmat metodit mahdollistavat ilman palvelinta toimivan navigaation hieman eri tavoin. Erillistä tunnistetta hyödyntävässä metodissa URL-osoitteessa käytetään ristikkomerkkiä (#) etuliitteenä, jolla viitataan osioon dokumentissa. Selaimet käsittelevät ristikkomerkin jälkeisen osan erillisenä muusta osoitteesta, ilman uutta palvelinpyyntöä. Muutos tallentuu kuitenkin selaushistoriaan, jolloin navigaationäppäimet pysyvät toiminnassa. HTML5 History -rajapintaa hyödyntävä menetelmä on uudempi tapa hoitaa sivuston reititys. Selaintuen hiljalleen parantuessa se on yleistynyt reitityksen toteutusratkaisuna. Etuna tässä ratkaisussa on siistimpi URL-osoite, jossa ei ole enää tarvetta käyttää ristikkomerkkiä. Selaushistorian tallennuksessa käytetään sen sijaan HTML5 History -rajapinnan tarjoamia pushState- ja replaceState-metodeita, jotka eivät päivitä selainta URL-osoitteen muuttuessa [22, s. 94–96]. Hyvin toteutettu reititys eri näkymien välillä HTML5 History -rajapinnan avulla parantaa myös sovelluksen hakukoneystävällisyyttä. [27.]

2.3 JavaScript-ohjelmistokehykset

SPA-sovelluksia toteutettaessa käytetään yleisesti apuna JavaScript-ohjelmistokehyksiä, joiden avulla pystytään hallitsemaan kontrolloidusti suurempia kokonaisuuksia. SPA-sovelluksista muodostuu helposti monimutkaisia, jolloin rakenteen selkeys on oleellisessa osassa toteutusten kehityksessä ja ylläpidossa. Ohjelmistokehysten tarkoituksena on tarjota ratkaisut SPA-sovelluskehityksen yleisimpiin haasteisiin ja mahdollistaa kehittäjän keskittyminen kokonaisvaltaisemmin varsinaiseen toteutukseen. Yleisesti tämä tarkoittaa valmiiksi mietityn roolituksen modulaarisuuteen esimerkiksi MV*-mallin mukaisesti. Valmiit ratkaisumallit nopeuttavat kehitystä ja vähentävät usein kehittäjän kirjoittaman koodin määrää. Myös skaalautuvuus paranee, koska modulaarisuuteen pohjautuvan jaottelu on yleisesti ohjelmistokehysten lähtökohtana. [17, s. 22–23.]

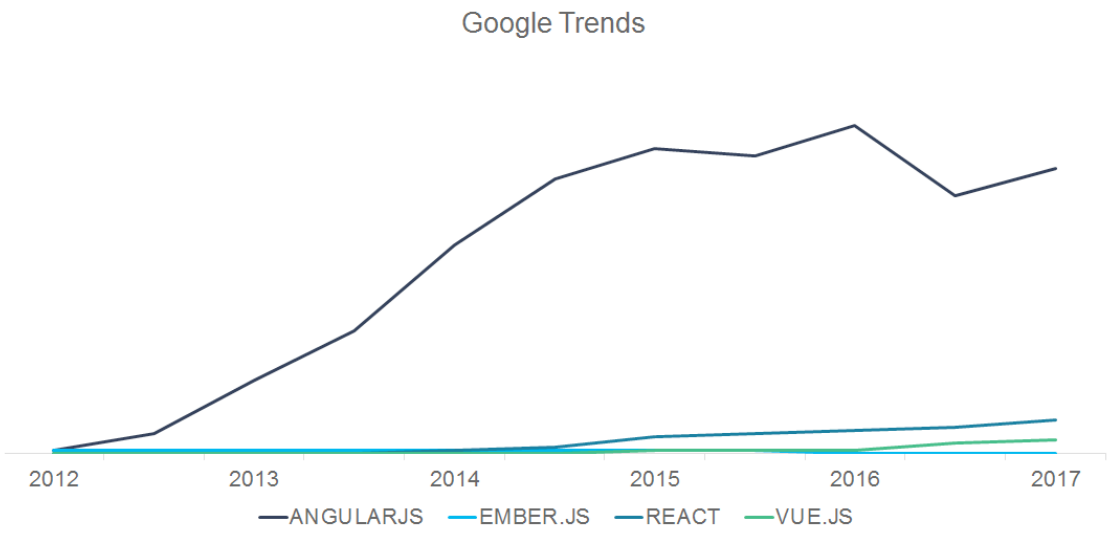
JavaScript-ohjelmistokehys pyrkii ratkaisemaan toteutuksen rakenteen ja tarjoaa usein selkeät suuntaviivat monipuolisten SPA-sovellusten toteuttamiseen. Tätä varten ohjelmistokehykseen on sisäänrakennettu kaikki SPA-sovelluksen tarvitsemat ominaisuudet tietokantakutsuista reititykseen. Valmiit ratkaisumallit kuitenkin heikentävät muokattavuutta, minkä vuoksi pienempiin osakokonaisuuksiin keskittyvien JavaScript-kirjastojen käyttö on kasvanut. Termiä JavaScript-ohjelmistokehys käytetäänkin yleisesti myös näistä toteutusten pohjana toimivista JavaScript-kirjastoista. Näissä toteutuksissa rinnalle tarvitaan kuitenkin muita JavaScript-kirjastoja eheän SPA-sovelluksen toteuttamiseksi. [28.]

JavaScriptin käytön kasvu ja kehityksen nopeus näkyy kenties kaikkein selkeimmin erilaisten ohjelmistokehysten ja kirjastojen suurena määränä. Toteutukset usein pyrkivät ratkaisemaan saman ongelman eri lähtökohdista painottaen eri osa-alueita, jonka vuoksi yleispätevää ratkaisua ei toistaiseksi ole olemassa. Jossain määrin osviittaa antaa kuitenkin suosio, jota voidaan arvioida monin erilaisin keinoin. Kuvassa 8 on esitelty joukko yleisimpiä avoimen lähdekoodin alaisia JavaScript-ohjelmistokehyksiä ja -kirjastoja. Kuvasta nähdään verkkopalvelu Githubissa esillä olevien projektien suosiota suhteessa toisiinsa ja Stackoverflow-palvelun kysymysmäärät kunkin projektin osalta maaliskuussa 2017. Kuvan perusteella suosituimmat projektit Githubissa ovat React, AngularJS ja Vue.js. Stackoverflow-palvelussa esitettyjä kysymyksiä on AngularJS:n osalta ylivoimaisesti eniten. [29; 30.]

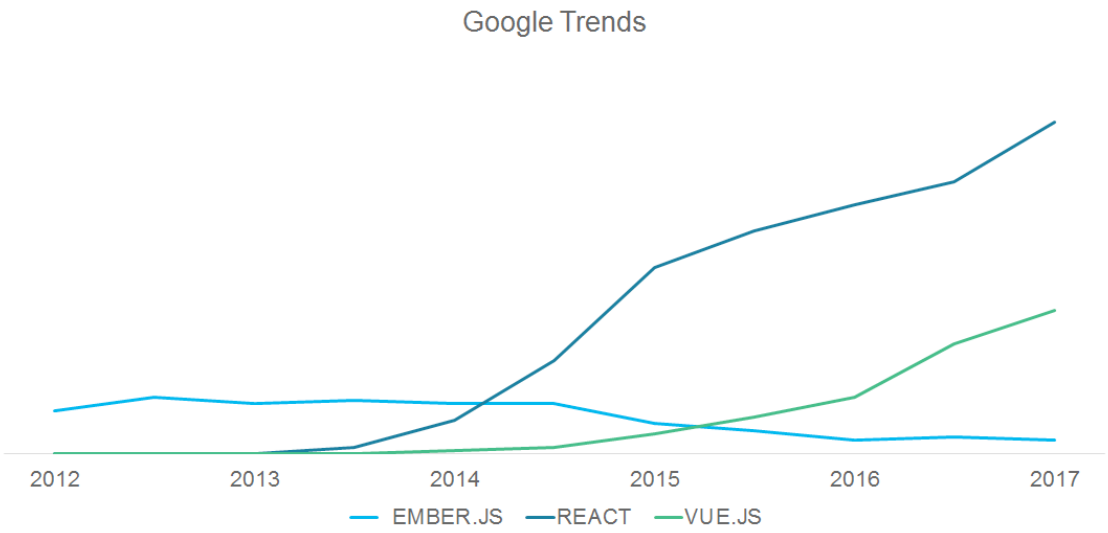


Kuva 8. SPA-sovelluskehityksessä yleisesti käytettyjen ohjelmistokehysten ja kirjastojen suosiovertailu maaliskuussa 2017 [29; 30].

Yksityiskohtaisempaan tarkasteluun valikoituivat AngularJS, Ember.js, React ja Vue.js. Valinta pohjautui Githubin tähtimerkintöihin, Stackoverflow'n kysymysmääriin ja Google Trendsin tarjoamiin hakutuloksiin. Kuvassa 9 on nähtävissä näiden neljän suosio Googlen hakutuloksissa viimeisten viiden vuoden ajalta. Kuvassa 10 on saman aikavälin tiedot ilman AngularJS-ohjelmistokehystä, jolloin muiden suosioerot tulevat selkeämmin esille. AngularJS:n vahva asema muihin verrattuna on selkeästi havaittavissa näistä kuvista. React ja Vue.js ovat lyhyessä ajassa kasvattaneet suosiotaan runsaasti, minkä myötä ne ovat tällä hetkellä vahvimman ehdokkaat murtamaan AngularJS:n vuosia jatkuneen suosion. Selkeästi laskusuhdanteessa on nelikosta ainoastaan Ember.js, joka hiljalleen on menettänyt suosiotaan. [29; 30; 31.] Edellä mainitut muutokset voidaan havaita kaikista kolmesta kuvasta 8, 9 ja 10.



Kuva 9. AngularJS:n Ember.js:n, Reactin ja Vue.js:n suosio Googlen hakutuloksissa aikavälillä 2012–2017. Y-akseli esittää hakumäärää suhteutettuna kaikkiin Googlessa tehtyihin hakuihin [31].



Kuva 10. Ember.js:n, Reactin ja Vue.js:n suosio Googlen hakutuloksissa aikavälillä 2012–2017. Y-akseli esittää hakumäärää suhteutettuna kaikkiin Googlessa tehtyihin hakuihin [31].

2.3.1 AngularJS-ohjelmistokehys

AngularJS on Googlen ylläpitämä ohjelmistokehys, joka julkaistiin vuonna 2010. Se on yksi suosituimmista toteutuspohjista SPA-sovellusten kehittämiseen ja sopii niin pieniin kuin suuriinkin sovelluksiin. AngularJS pyrkii olemaan kokonaisvaltainen selainpään ratkaisu monipuolisten sovellusten toteuttamiseen. Se tarjoaa valmiit mallit reitityksestä, datasidontaan ja validoinnista Ajax-kutsujen tekemiseen. Ohjelmistokehys jakautuu tällä hetkellä 1.x- ja 2.x-versioihin. [32.]

Kirjoitushetkellä versiossa 1.5 oleva AngularJS tuo MVC-tyyppisen arkkitehtuurin selaintoteutuksiin helpottamaan kehityksen organisointia. Arkkitehtuurin mukaisesti malli, näkymä ja kontrolleri ovat toisistaan erillisiä. Mallina toimii objektimuotoinen data, näkymänä selaimen DOM ja kontrollerin ominaisuudet ovat kokoelma JavaScript-funktioita. Toimintalogiikka perustuu HTML:n syntaksin laajentamiseen AngularJS:n omilla merkitsimillä, joita kutsutaan direktiiveiksi. Direktiivit helpottavat ulkoasun ja logiikan erottamista, jolloin MVC-mallin mukaisesti näkymää pystytään helposti vaihtamaan HTML:n ja CSS:n avulla muuttamatta logiikkaa JavaScriptissä. Tämä helpottaa myös sovelluksen testattavuutta ja osien uudelleenkäytettävyyttä. DOM:n automaattinen muokkaaminen tapahtuu kaksisuuntaisella datasidonnalla, jolloin sovelluksen mallin muutokset siirtyvät näkymään ja vastaavasti näkymän malliin automaattisesti. Muutoksen tapahtuessa An-

gularJS käy läpi koko sovelluksen mallin ja päivittää osapuolet riippuvuussuhteiden mukaisesti. AngularJS:n perusperiaatteisiin kuuluu vahva erottelu selain- ja palvelinpään toteutuksiin ja ohjelmistokehyksen läsnäoloon kehityksen jokaisessa vaiheessa. Sovelluskehys pyrkii myös alentamaan käyttöönoton kynnystä keskittymällä aloittamisen helppouteen. [32.]

Ohjelmistokehyksen täysin uudelleenkirjoitettu versio Angular 2 muuttaa merkittävästi sovelluskehitystä aikaisempiin versioiden nähden. Syyskuussa 2016 julkaistu uusi versio tuo kehitykseen mukaan Microsoftin kehittämän TypeScript-ohjelmointikielen, joka mahdollistaa JavaScriptin staattisen tyypityksen ja luokkapohjaisen olio-ohjelmoinnin. TypeScriptin käyttö ohjelmistokehyksen kanssa ei ole pakollista, mutta kehittäjätiimi suosittaa sitä vahvasti. Uuden version komponenttipohjaisuus, yksinkertaistetut direktiivit ja yksisuuntaiseen datasidontaan siirtyminen pyrkivät modernisoimaan ohjelmistokehystä vastaamaan nykyaikaisen sovelluskehityksen haasteisiin. [33.]

AngularJS:n parhaisiin puoliin kuuluu ehdottomasti kattavan ohjelmistokehyksen myötä monipuoliset ratkaisut SPA-sovelluksen rakentamiseen. Suosion myötä yhteisö ja lisäosien määrä ohjelmistokehyksen ympärillä on laaja, minkä ansioista moniin ongelmiin löytyy valmiit ratkaisut. AngularJS:n 1.x-versiot kärsivät kuitenkin vanhentuneesta toteutuksestaan, joka heijastuu erityisesti suorituskyky- ja testausongelmina. Uudelleenkirjoitetun version myötä osa näistä ongelmista on selätetty, mutta nähtäväksi jää, pystyykö uusiutunut AngularJS todellisuudessa vastaamaan Reactin ja Vue.js:n kaltaisten modulaaristen toteutustapojen kasvavaan suosioon. [34.]

2.3.2 Ember.js-ohjelmistokehys

Ember.js on JavaScript-ohjelmistokehys, jota käytetään yleensä suurien SPA-sovellusten toteuttamiseen. Ohjelmistokehys pohjautuu vahvasti MVC-arkkitehtuuriin ja tarjoaa kaikki SPA-sovelluksessa tarvittavat ominaisuudet. Niiden lisäksi kehityksen tueksi tarjotaan joukko lisäominaisuuksia Sass-välikääntäjistä JavaScriptin pienentäjiin (minifier). Ember.js on pääasiallisesti ohjelmistokehys web-sovelluksille, mutta sillä voi myös tehdä työpöytä- ja mobiilisovelluksia. AngularJS:stä ja Reactista poiketen Ember.js:n taustalla ei ole yksittäistä yritystä vaan lukuisia sponsoreita. Ember.js onkin vahvasti yhteisövetoinen, ja se pyrkii aktiivisesti omaksumaan parhaita puolia muista ohjelmistokehyksistä ja JavaScript-kirjastoista. Kirjoitushetkellä Ember.js:n uusin vakaa versio on 2.4. [35.]

Ember.js:n ydinkäsitteisiin kuuluvat HTML-pohjat, mallit, komponentit, reitit ja reittienkäsittelijät. HTML-pohjat vastaavat pitkälti perinteisiä HTML-sivuja, jotka hyödyntävät Handlebars-kirjastoa mallinnuksessa. Kirjaston avulla pohjiin saadaan liitettyä esimerkiksi muuttujia. Mallit ovat tilallisia objekteja, joihin Ember.js tarjoaa erilaisia kehittämistä helpottavia funktioita. Tyypillisesti mallien data säilötään palvelimelle, mutta myös HTML5:n Web Storage -rajapintaa pystytään hyödyntämään paikallisessa tallennuksessa. Reitityksen tehtävä on muuntaa URL-osoitteet sarjaksi malleja, joiden perusteella muodostetaan pohjat ja ladataan niihin kuuluvat mallit. [35.]

Ember.js:n laajuus ja pyrkimys ratkaista kaikki web-sovelluskehityksen ongelmat ovat ohjelmistokehyksen suurin vahvuus ja heikkous. Moniin ongelmiin on luotu parhaat käytännöt, mikä vähentää oleellisesti kehittäjän kirjoittaman koodin määrää. Käytännössä tämä näkyy muun muassa joukkona oletusasetuksia, joiden perusteella reitit määritellään tai yksikkötestit suoritetaan. Vahvat suuntaviivat ohjelmistokehyksen toimesta vapauttavat kehittäjän keskittymään paremmin varsinaiseen tarkoitukseen, mutta heikentävät muokattavuutta. [35; 36.]

2.3.3 React-ohjelmistokirjasto

React on Facebookin ylläpitämä JavaScript-kirjasto käyttöliittymien rakentamiseen. Kirjasto kehitettiin Facebookin toimesta ratkaisemaan suorituskykyongelmat paljon reaaliaikaisesti dataa sisältävän DOM:n muokkaamisessa. Toukokuussa 2013 Facebook julkaisi Reactin avoimena lähdekoodina, ja kirjoitushetkellä uusin vakaa versio on 15.4. [37; 38.]

React pohjautuu Flux-arkkitehtuuriin, jossa sen tehtävänä on hoitaa näkymien muodostaminen. Pelkkä esityskerros on harvoin riittävä, minkä vuoksi React tarvitsee käytännössä aina rinnalleen muita kirjastoja hoitamaan SPA-sovelluksissa olennaisessa osassa olevia asioita, kuten reititystä. Ero AngularJS:n tai Ember.js:n kaltaisiin JavaScript-ohjelmistokehysiin on merkittävä, koska valmiit ratkaisumallit puuttuvat ja kehittäjälle siirtyy suurempi vastuu toimintojen rakentamisesta. Modulaarisuus tuo kuitenkin kehitykseen monipuolisuutta, joka helpottaa uusien ominaisuuksien ja menetelmien käyttöönottoa. [37.]

Olennaisessa osassa Reactissa on virtuaalinen DOM, joka on sivun varsinaista DOM:a simuloiva puumallinen objekti. Näkymien muodostaminen ja päivittäminen perustuvat

Reactissa virtuaalisen DOM:n muokkaamiseen. Muokattua versiota verrataan sivun rakenteeseen, johon lopulta päivitetään ainoastaan muuttuneet osat. DOM-rakenteita vertaileva algoritmi pyrkii päivittämään näkymän tekemällä siihen mahdollisimman vähän muutoksia. Käyttäjälle tämä näkyy näkyminen nopeana päivittymisenä. Reactissa komponentit kirjoitetaan XML-tyyppisellä JSX-syntaksilla, joka käännetään JavaScriptiksi ennen liittämistä virtuaaliseen DOM:iin. JSX selkeyttää komponenttien kirjoittamista, mutta ei ole pakollinen osa Reactia. JavaScriptin vahva hyödyntäminen on yleistä Reactin kanssa, kuten nähdään kuvan 11 esimerkistä, jossa esitystapa, tyyli ja logiikka on kirjoitettu samalla kielellä. [37.]

Kuva 11. Reactissa komponenttien kirjottamiseen voidaan käyttää XML-tyyppistä JSX-syntaksia. Varsin yleistä on myös esitystavan, tyylin ja logiikan kirjoittaminen samalla kielellä.

Komponenttipohjaisuus ja virtuaalinen DOM tuovat mukanaan myös uuden ulottuvuuden sovelluskehitykseen. Facebook julkaisi maaliskuussa 2015 React Nativen, kirjaston natiivisovelluksien luomiseen Applen IOS- ja Googlen Android-laitteille. React Nativen avulla web-teknologioin kehitetty koodi käännetään mobiilialustojen omille kielille. Mobiilialustojen aitojen natiivielementtien myötä suorituskyky ja käyttökokemus eivät kärsi ja kehitykseen saadaan selaintoteutuksien ratkaisumalleja ja työkaluja. React Native ei kuitenkaan pyri olemaan ratkaisu, jossa sama koodi toimii automaattisesti käännettynä kaikkialla. Lähtökohtana on ennemmin samojen React-periaatteiden hyödyntäminen alustariippumattomasti eri toteutuksissa, oli sitten kyseessä web-, IOS- tai Android-sovellus. [39.]

2.3.4 Vue.js-ohjelmistokehys

Vue.js on vuonna 2014 julkaistu JavaScript-ohjelmistokehys, jonka kehitti Googlella AngularJS:n parissa työskennellyt Evan You. Monia ominaisuuksia AngularJS:stä lainaava Vue.js keskittyy vahvasti näkymienhallintaan ja modulaariseen rakenteeseen, jossa kokonaisuuksia pystytään helposti hallitsemaan ja muuttamaan. Vue.js:n käyttöönoton helppous on pyritty vahvasti huomioimaan kehittäjätiimin toimesta. Tämä näkyy muun muassa modulaarisessa rakenteessa, jonka myötä näkymienmuodostaja voidaan irrallisena ottaa käyttöön ja vasta myöhemmin lisätä reitityksen mahdollistavat osakokonaisuudet. Kirjoitushetkellä Vue.js:n uusin vakaa versio on 2.2. [40.]

Vue.js ei suoraan ilmennä MVVM-ohjelmistoarkkitehtuuria, mutta hyödyntää monia sen periaatteita. Toimintojen keskipisteenä on ViewModel, joka kaksisuuntaisen datasidonnan avulla pitää näkymässä nähtävillä olevan datan ja sovelluksen mallin sidoksissa toisiinsa. Käytännössä tämä tehdään käyttämällä Vue.js:n HTML-mallipohjia. Pinnan alla mallipohjat yhdistetään DOM-rakenteeseen virtuaalisen DOM:n avulla, samaan tapaan kuin Reactissa. Sovelluksen malli puolestaan on JavaScript-objekti, johon tehdyt muutokset aiheuttavat näkymien päivityksen. Vue.js tarjoaa optimoidun näkymien päivitystarpeen tarkistuksen oletuksena, mikä tekee tilanhallinnasta yksinkertaista ja intuitiivista. [40.]

Muihin JavaScript-ohjelmistokehyksiin verrattuna Vue.js pyrkii olemaan helposti käyttöönotettava modulaarinen ratkaisu, joka skaalautuu niin suuriin kuin pieniin toteutuksiin. Se on aikaisemmin esiteltyihin ohjelmistokehyksiin verrattuna suorituskykyisin, mutta kärsii vielä ainakin toistaiseksi pienehköstä ekosysteemistä ja sen myötä erilaisten liitännäisten puutteesta. Vue.js:n kehityksen taustalla ei ole suuryritystä, minkä vuoksi kehitys on toistaiseksi vahvasti riippuvaista pienen kehittäjäjoukon työpanoksesta. Tämä asettaa ohjelmistokehyksen tulevaisuuden osin ennalta arvaamattomaan tilaan. Suosio on kuitenkin tällä hetkellä kovassa kasvussa, minkä myötä käyttö ja kehitykseen osallistuvien henkilöiden määrä kasvaa nopeasti. [40.]

3 SPA-sovelluksen toteutus

3.1 Lähtökohdat

Toimitusten ja organisaatioiden viestintäosastojen pöydille ja sähköposteihin tulvii päivittäin useita satoja kutsuja ja tiedotteita. Nämä yhdistettynä eri sosiaalisen median palveluissa ilmoitettuihin tapahtumiin ja muihin hajanaisiin julkaisualustoihin kasvattavat hajanaisuutta entisestään. Merkityksellisten tapahtumien löytämisestä muodostuu tällöin helposti sattumanvarainen prosessi, joka vaatii runsaasti resursseja ja ripauksen onnea. Tätä ongelmaa ratkaisemaan syntyi insinöörityönä uudenlainen tapahtumapalvelu.

Tässä insinöörityössä kehitettiin asiakkaalle web-sovellus, joka pyrki vastaamaan käyttäjälähtöisesti viestinnän tekijöiden haasteisiin. Palvelun tavoitteena oli koota kattavasti yhteen median ja viestinnän kannalta merkittäviä tapahtumia Suomesta ja ulkomailta, minkä jälkeen niitä pystyisi selaamaan ja suodattamaan helposti ja luotettavasti. Näin laajasta tarjonnasta oleelliset tapahtumat on helpompi löytää ja resurssit saadaan kohdistettua entistä paremmin.

Projekti lähti käyntiin käyttäjähaastatteluilla, joissa selvitettiin organisaatioviestijöiden ja median osapuolien tarvetta tämänkaltaiselle viestinnän ennakointityökalulle. Käyttäjien tarpeiden ja arvonmuodostuksen selvittämisen jälkeen pystyttiin hahmottelemaan palvelun ensimmäisen MVP-version (Minimum viable product) sisältö. Asiakkaiden toiveisiin perustuen palvelusta lähdettiin rakentamaan minimalistista, helppokäyttöistä ja reaaliaikaista huomioiden käytettävyys mobiilissa kontekstissa. Tästä hetkestä käynnistyi toteutusprosessi, johon Solita Oy:n puolelta oli koottu kolmihenkinen tiimi, joka koostui palvelumuotoilijasta, palvelin- ja integraatioasiantuntijasta ja käyttöliittymäkehittäjästä. Tavoitteena projektin neljä kuukautta kestävälle ensimmäiselle vaiheelle oli testata idean toimivuus markkinoilla ja luoda tapahtumasovelluksesta ensimmäinen tuotantokelpoinen versio. Tässä insinöörityössä kehitettyä palvelua ja sen oleellisimpia asioita kuvataan käyttöliittymäpuolen kehittäjän näkökulmasta.

Palvelun kehitys seurasi jatkuvan kehittämisen mallia, jossa edettiin sykleittäin keräten jatkuvasti palautetta asiakkaalta. Kehityksen oikean suunnan pitämiseksi jo projektin alkuvaiheessa koottiin myös ryhmä loppukäyttäjiä, jotka osallistuivat kehitykseen antamalla arvokasta palautetta aina uuden version julkistamisen myötä. Versiomuutokset py-

rittiin pitämään mahdollisimman pieninä, eli uusien ominaisuuksien valmistuttua ne tuotiin välittömästi loppuasiakkaiden saataville jatkuvan julkaisemisen (continuous delivery) periaatteiden mukaisesti. Muiltakin osin toimitusmallissa mukailtiin ketterää kehitysmallia, jotta projektin riskit pysyisivät pieninä ja asiakaslähtöisyys keskiössä. Osana ketteriä toimintamalleja käytettiin muun muassa työnseurantaan Kanban-taulua, jonka tarkoituksena on lisätä projektin läpinäkyvyyttä projektitiimin ja asiakkaan välillä. Menetelmässä työvaiheet pilkotaan pieniksi kokonaisuuksiksi, joiden edistymistä seurataan sijoittamalla ne kaikkien nähtävillä olevalle taululle sarakkeisiin aloittamatta (todo), työn alla (doing), tehty (done). Työvaiheiden edistymistä seurattiin viikoittaisissa määrittely- ja suunnittelupalavereissa.

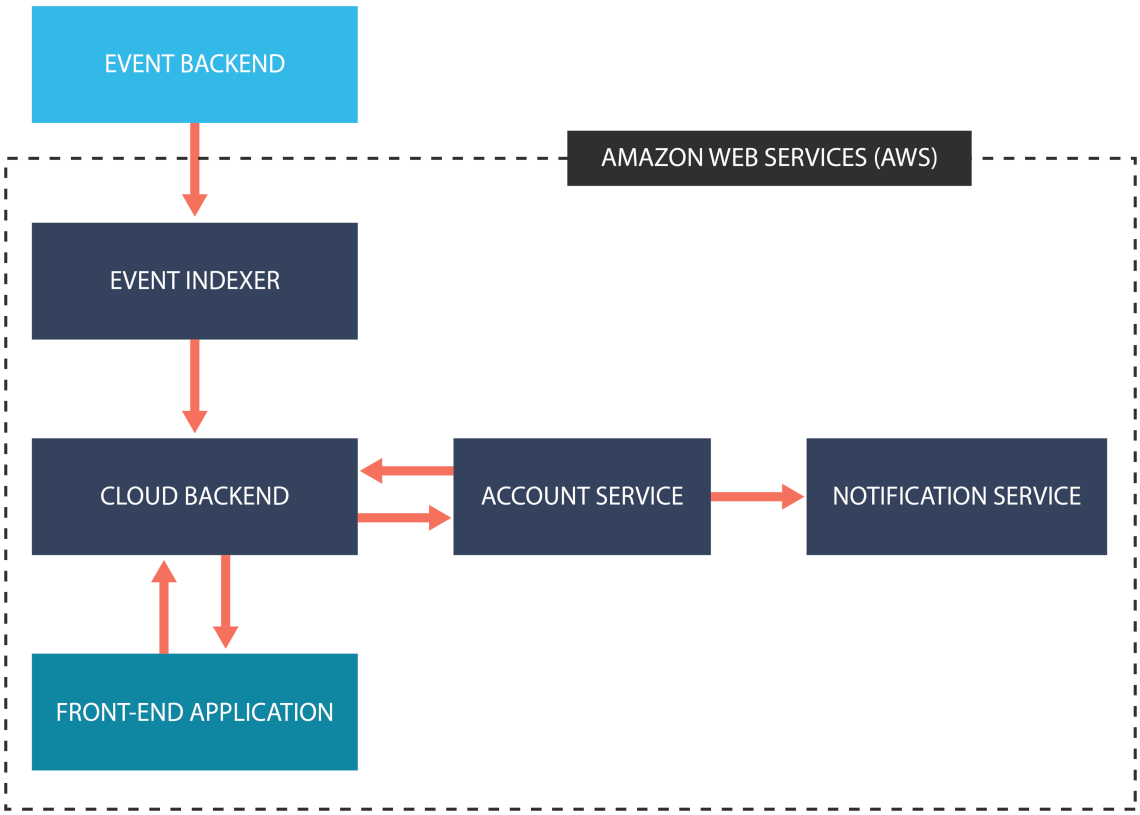
3.2 Teknologiat ja kehitystyökalut

Tapahtumasovellus jakautui kuuteen erilliseen mikropalveluun, jotka kommunikoivat keskenään tuottaen halutun toiminnallisuuden. Tapahtumasovelluksen mikropalvelut olivat nimeltään

- Event Backend (tapahtumatiedon koostaminen)
- Event Indexer (tapahtumatiedon jalostaminen)
- Notification Service (ilmoituskeskus)
- Account Service (käyttäjäpalvelut)
- Cloud Backend (taustapalveluiden yhdistäminen ja datan tarjoaminen käyttöliittymäkerrokselle)
- Front-end Application (käyttöliittymäkerros).

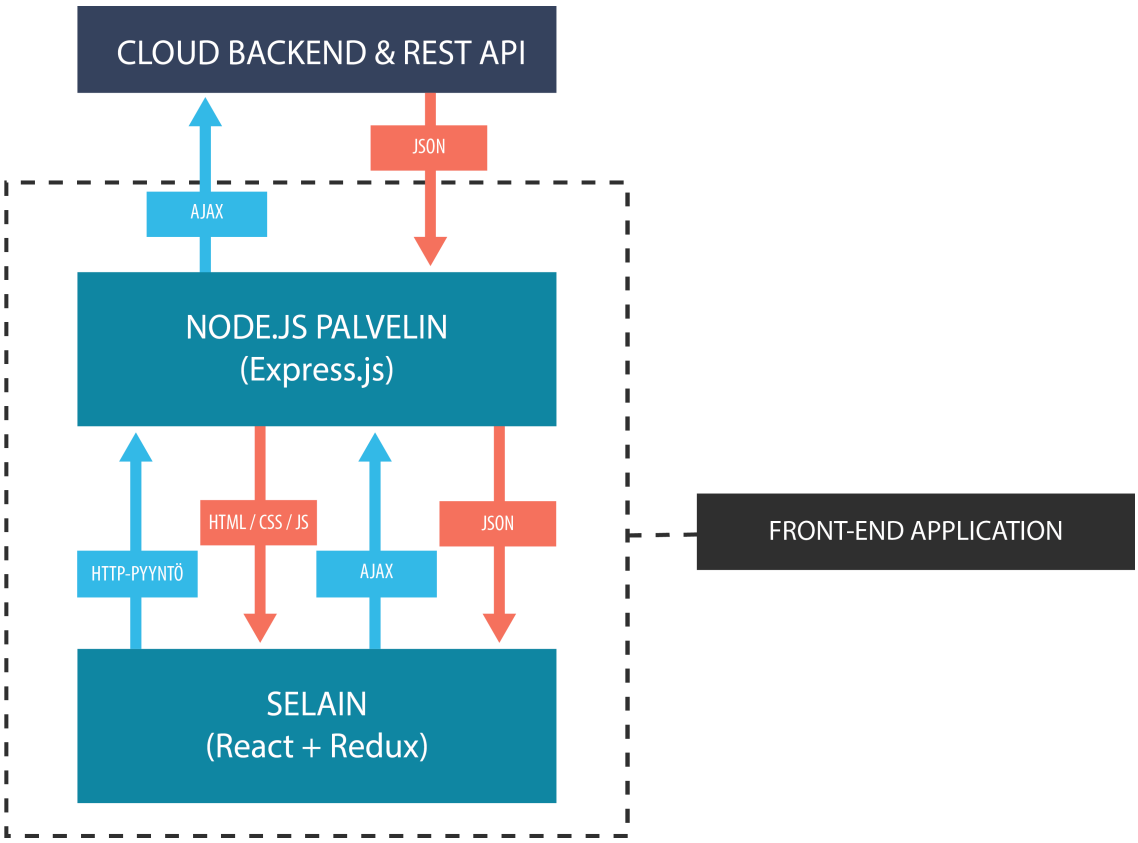
Asiakkaan omassa konesalissa ollutta Event Backendiä lukuun ottamatta kaikki mikropalvelut toimivat Amazonin pilvipalvelinympäristössä (Amazon EC2 Container Service). Kuvassa 12 on esitetty yhteydet eri mikropalveluiden välillä. Tapahtumatiedon alkulähde on asiakkaan Event Backendissä, josta data lähetetään tasaisin väliajoin Event Indexeriin XML-muodossa. Tässä vaiheessa data jalostetaan paremmin hyödynnettäväksi

muun muassa yhdistelemällä tietoja ja karsimalla ylimääräisiä osioita. Käsittelyn jälkeen data tallennetaan tietokantaan, jotta Cloud Backend pääsee siihen jatkossa käsiksi. Käyttäjähallinta (Account Service) ja sähköpostiviestien lähetyksestä vastaava Notification Service on myös eriytetty omiksi mikropalveluikseen, joiden toimintaa hallinnoidaan Cloud Backendin kautta. Tässä insinöörityössä keskitytään käyttöliittymäkerroksen Front-end Application -mikropalveluun, joka kommunikoi muihin mikropalveluihin Cloud Backendin RESTful-mallisen ohjelmointirajapinnan avulla.



Kuva 12. Tapahtumasovellus koostuu useasta eri mikropalvelusta, jotka kommunikoivat keskenään. Suurin osa mikropalveluista sijaitsee Amazonin pilvipalvelinympäristössä.

Käyttöliittymäkerroksen mikropalvelu on jaettu Node.js:llä toimivaan taustajärjestelmään ja SPA-arkkitehtuurilla hyödyntävään käyttöliittymäkerrokseen. Taustajärjestelmä käyttää Node.js:lle tehtyä Express.js-ohjelmistokehystä, joka yksinkertaistaa yleisimpiä toimintoja. Niihin kuuluvat muun muassa palvelinpyyntöihin reagoinnit. Kuvassa 13 näkyy käyttöliittymäkerroksen mikropalvelun rakenne. Koska tapahtumatietoa palauttava rajapinta sijaitsee mikropalvelun ulkopuolella, taustajärjestelmän rooliksi jää staattisten tiedostojen jakaminen ja palvelinpyyntöjen ja niiden antamien vastausten välittäminen. Suurin osa mikropalvelun logiikasta sijaitsee SPA-arkkitehtuurin mukaisesti selainpäässä.



Kuva 13. Tapahtumasovelluksen käyttöliittymäkerros on oma SPA-arkkitehtuurilla toimiva mikropalvelu.

Käyttöliittymäkerroksen logiikka toteutettiin Flux-ohjelmistoarkkitehtuurilla ja JavaScript-ohjelmistokirjasto Reactia hyödyntäen. Komponenttipohjaisuudelle ja modulaarisuudelle oli selkeä tarve, koska sovelluksen tuli olla helposti muokattavissa kehityksen aikana tulevien käyttäjäpalautteiden perusteella. Reactin valintaa puolsivat myös pitkän aikavälin suunnitelmat, joissa sovelluksen tulisi skaalautua pienestä MVP-tuotteesta hiljalleen yhä suuremmaksi kokonaisuudeksi. Skaalautuva kirjastopohjainen ajattelu helpottaa nopeasti muuttuvan JavaScript-kehityksen ylläpidettävyyttä, koska vanhentuneet tai muuten sopimattomat osat voidaan päivittää helposti uusilla. Näin ollen toimintamallit eivät ole riippuvaisia AngularJS:n tai Ember.js:n kaltaisen kokonaisen JavaScript-ohjelmistokehyksen kehityssuunnasta ja sen mukanaan tuomista muutoksista.

Käyttöliittymäkerroksen modulaarista rakennetta edesauttamaan valittiin kehitystyöhön Node.js ja sen pakettienhallinta npm (node package manager). Tämä yhdistelmä mahdollistaa kehittäjille koodin helpon julkaisemisen, jakamisen ja hyödyntämisen erilaisissa toteutuksissa. Paketinhallintajärjestelmä helpottaa moduuleista koostuvien web-sovel-

lusten rakentamista tarjoamalla kolmannen osapuolen julkaisemia kehitysratkaisuja ongelmakohtiin. Yleensä nämä ongelmakohdat ovat pieniä ja tarkoin määriteltyjä, minkä vuoksi tyypillinen web-sovellus sisältää runsaasti erilaisia paketteja. Npm:n package.json-nimisen tiedoston avulla lokaalisti asennettavien pakettien hallinta onnistuu helposti komentorivin kautta ja sovelluksen ulkopuoliset riippuvuussuhteet pysyvät helposti ajan tasalla tiimin jäsenten kesken. Myöskään kolmannen osapuolen paketteja ei tarvitse kuljettaa jakelupaketin mukana, koska ne saadaan package.json-tiedoston määritysten mukaisesti paketinhallintajärjestelmästä kehitysympäristön asennuksen yhteydessä. [41.] Projektin package.json-tiedostoon on myös määritelty joukko npm-komentoja, joiden avulla esimerkiksi kehitysympäristön pystytys ja tuotantoversion julkaisuun liittyvät toimenpiteet on automatisoitu. Näiden npm-komentojen avulla on korvattu tarve erilliselle Gulpin tai Gruntin kaltaiselle tehtävänsuorittajalle.

Toimintojen jakaminen pieniin palasiin tarkoittaa yleensä lukuisia tiedostoja ja kansiorakenteita. Perinteisesti nämä tiedostot on liitetty HTML-sivuun script-elementeillä, joiden avulla ladataan tiedostot palvelimelta selaimeen. Jokainen tiedosto vaatii erillisen HTTP-pyynnön palvelimelle, mikä hidastaa sivun latausaikaa oleellisesti. Ongelmaa ratkaisemaan on kehitetty työkaluja, joilla moduulit niputetaan isommiksi kokonaisuuksiksi, yleensä yhdeksi tiedostoksi, jolloin HTTP-pyynnöt ja latausajat vähenevät merkittävästi. Samalla kirjoitetusta koodista on hyvä karsia ylimääräiset välilyöntien ja rivinvaihdosten kaltaiset suorituksen kannalta turhat sisällöt, jolloin tiedostokokoa saadaan pienennettyä muuttamatta koodin toiminnallisuuksia. [42.] JavaScriptin lisäksi samat toimenpiteet on hyvä tehdä myös tyylitiedostoille. Näitä tehtäviä hoitamaan sovelluspohjaan lisättiin Webpack, joka erityisesti laajemmissa projektikokonaisuuksissa on monipuolisuudessaan ja muokattavuudessaan kilpailijoitaan tällä hetkellä edellä [43].

Webpack sisältää runsaasti erilaisia monipuolisia säätömahdollisuuksia, joilla ladattuja moduuleita pystytään automatisoidusti käsittelemään. Yksi merkittävimmistä ominaisuuksista ovat lataajat (loaders), joiden avulla pystytään käsittelemään erilaisia tiedostoja aina JSON:n parsimisesta JavaScript-kääntäjiin. Lataajien avulla Webpack pystyy käsittelemään JavaScript-koodin ohella muun muassa tyylitiedostoja, kirjaisintyyppejä ja kuvia, mikä tekee sovelluskehityksestä entistä jouhevampaa ja toteutuksesta optimoidumpaa. Yleisimmille CSS-esikäsittelijöille on omat lataajansa, joiden avulla Sass:n, Less:n ja Styluksen kaltaiset kielet kääntyvät selainten ymmärtämäksi CSS:ksi. Lataajien

avulla voidaan myös ECMAScript 2015:llä tai uudemmalla kirjotettu koodi kääntää ECMAScript 5:tä käyttävien JavaScript-moottorien ymmärtämään muotoon. Samalla tavalla myös Reactin hyödyntämä JSX-syntaksi muutetaan JavaScriptiksi. [44.]

SPA-sovellusten JavaScript-keskeisyys altistaa helposti pienille virheille, joiden etsiminen vie turhaa aikaa varsinaiselta sovelluskehitykseltä. JavaScript-koodin kirjoittamista helpottamaan on hyvä käyttää validointityökalua, joka ilmoittaa käyttäjälle syntaksi- tai nimeämisvirheiden kaltaisista häiriöistä jo kirjoitusvaiheessa. Saman työkalun avulla voidaan myös määrittää tiimin yhteiset tyyliohjeet koodin kirjoittamiseen. Eri työkalujen joukosta sovelluspohjaan valikoitui ESLint, joka sopii erityisen hyvin yhteen uusimpien ECMAScriptien ja JSX:n kanssa. ESLint on myös hyvin muokattavissa käyttäjien tarpeiden mukaisesti. [45.]

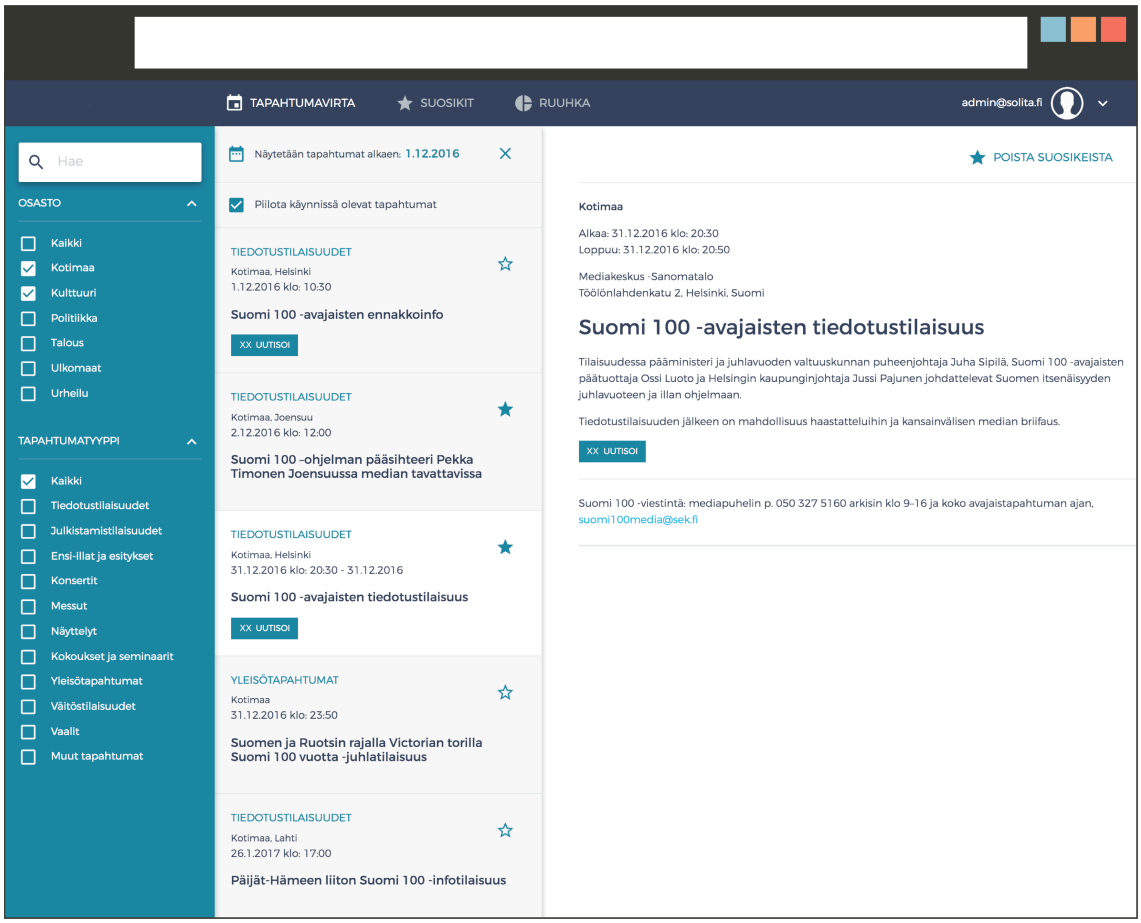
Usean kehittäjän voimin hallinnoitavan projektin ylläpidossa ja seurannassa käytettiin apuna Git-versionhallintajärjestelmää. Versionhallinnan avulla muutokset tallennetaan rekisteriin, josta ne ovat jäljitettävissä ja tarpeen vaatiessa muutettavissa. Näin esimerkiksi edelliseen versioon voidaan myöhemmässä vaiheessa palata. [46.] Versionhallintajärjestelmää hyödynnettiin projektissa myös erottamaan tuotanto- ja kehitysversio toisistaan.

3.3 Näkymien muodostaminen

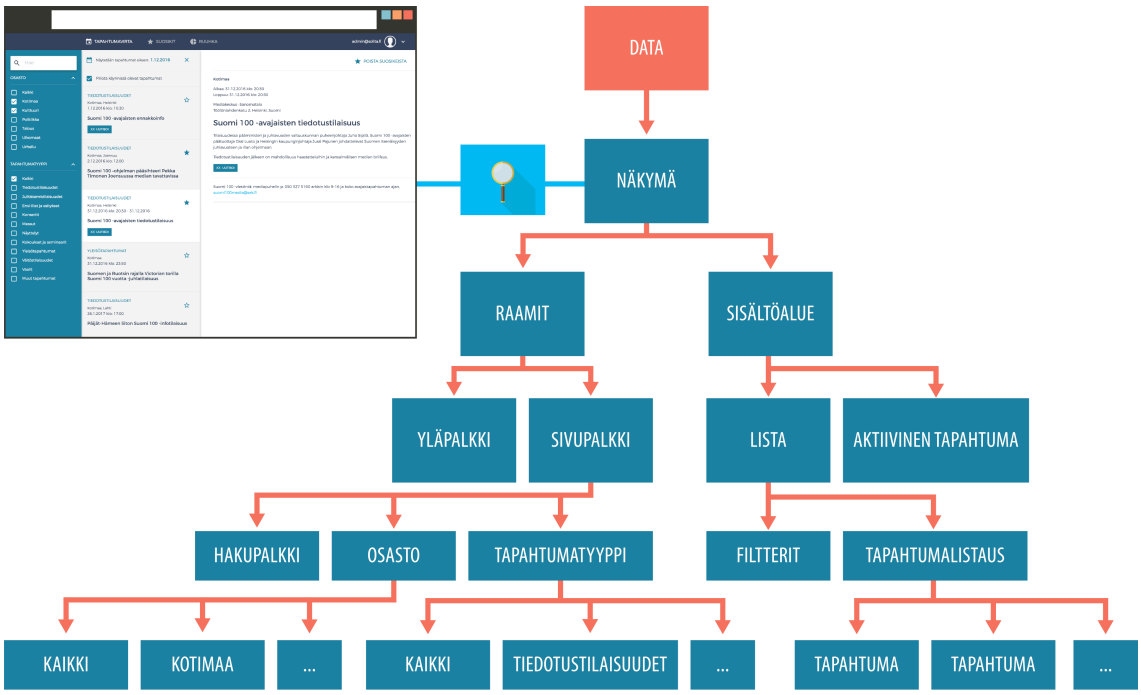
Näkymät muodostetaan Reactissa virtuaalisen DOM:n elementtien kautta. Elementit ovat Reactissa objekteja, joiden avulla määritetään käyttäjälle näkyvä sisältö. Niihin kuuluvat muutamaa poikkeusta lukuun ottamatta HTML:n peruselementit ja luodut komponentit. Komponentit ottavat sisäänsä ominaisuuksia palauttaen osakokonaisuuksia puumallisesta rakenteesta. Ominaisuuksien kasvaessa yksittäinen komponentti on hyvä jakaa uudestaan pienempiin osioihin, jotka yhdessä muodostavat suuremman kokonaisuuden. Ideaalitilanteessa jokainen komponentti suorittaa vain yhtä määrättyä asiaa. Pienemmät kokonaisuudet mahdollistavat dynaamiset uudelleenkäytettävät näkymät. [47.]

Kuvassa 14 on esitetty sovelluksen päänäkymä. Monipuolinen kokonaisuus sisältää joukon Reactilla luotuja komponentteja, jotka jakautuvat puumallisen rakenteen mukaan it-

senäisiin osioihin. Kuvassa 15 on esitetty kuvan 14 päänäkymä puumallisena rakenteena karkealla tasolla. Koko näkymä jakautuu ylä- ja sivupalkista koostuviin raameihin ja sisältöalueeseen. Jokainen näistä osioista sisältää joukon alikomponentteja, joista esimerkkeinä sisältöalueen lista ja aktiivinen tapahtuma. Puumallisessa rakenteessa voidaan samaan tapaan pureutua yhä syvemmälle, kunnes saavutetaan yksittäiset HTML:n peruselementit. Yksisuuntainen datasidonta näkyy kuvissa datan suuntana, joka virtaa aina ylemmältä tasolta alemmas. Datan perusteella voidaan dynaamisesti esimerkiksi muodostaa tapahtumalistauksen yksittäiset tapahtumakomponentit.

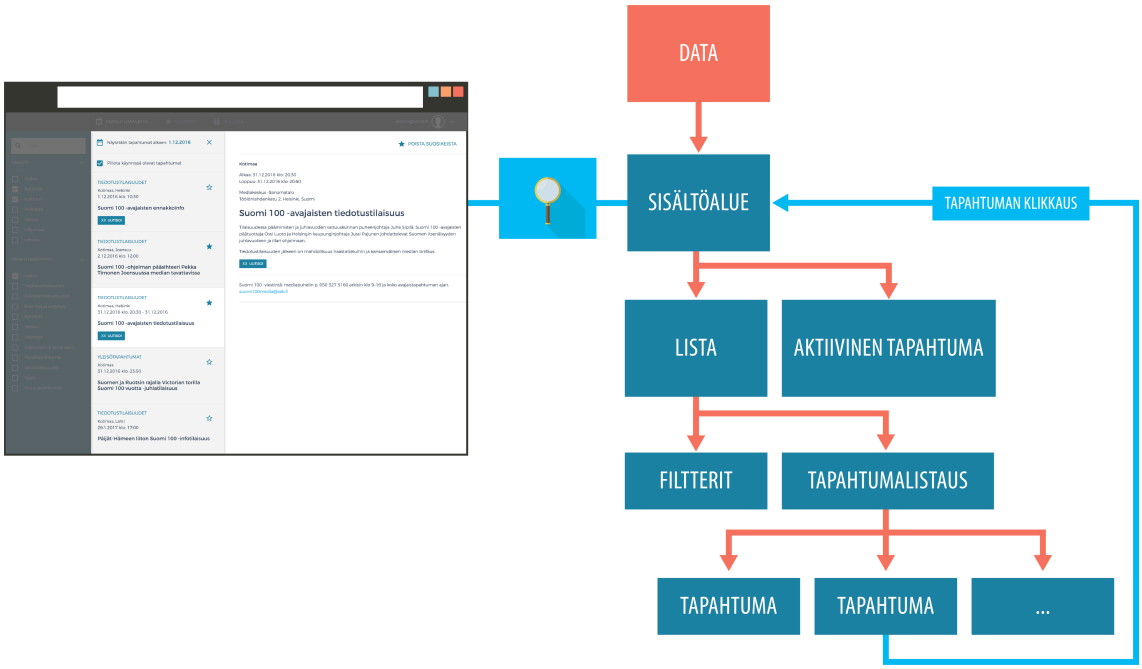


Kuva 14. Tapahtumasovelluksen päänäkymä sisältää ylä- ja sivupalkista koostuvat raamit, tapahtumalistauksen ja aktiivisen tapahtuman.



Kuva 15. Päänäkymän elementit jakautuvat Reactissa puumalliseksi kokonaisuudeksi, joka voidaan jakaa hyvin pieniin osiin.

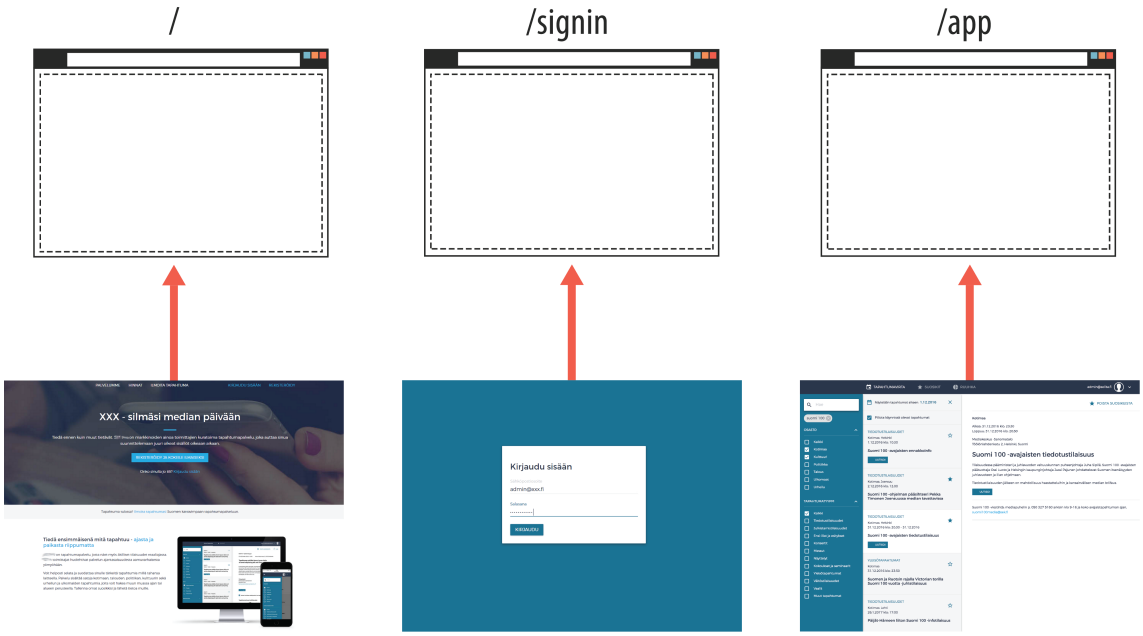
Yksisuuntaisen datasidonnan mukaisesti puurakenteessa alempana oleva komponentti ei voi suoraan muuttaa ylemmän tason tilaa. Elementin luonnin yhteydessä komponentille on kuitenkin mahdollista antaa ominaisuutena palautusfunktiota, jolla saadaan linkitettyä tilaa muuttavat funktiot tarvittaessa alemmille tasoille. [37.] Kuvassa 16 sinisellä nuolella kuvataan linkitettyä funktiota, jolla ylemmän tason tilamuutoksiin päästään käsiksi. Esimerkissä tapahtumalistauksen yksittäisen tapahtuman klikkaus muuttaa palautusfunktiona sisältöalueen tilaa. Muutoksen tapahduttua uusi tila valuu puurakenteessa alemmas ja muuttaa tässä tapauksessa näkymän aktiivisen tapahtuman.



Kuva 16. Tapahtumasovelluksen yksisuuntaisessa datasidonnassa ylemmän tason toimintoihin on mahdollista päästä käsiksi palautusfunktioilla. Kuvan esimerkissä tapahtumaa klikattaessa palautusfunktio muuttaa sisältöalueen aktiivista tapahtumaa.

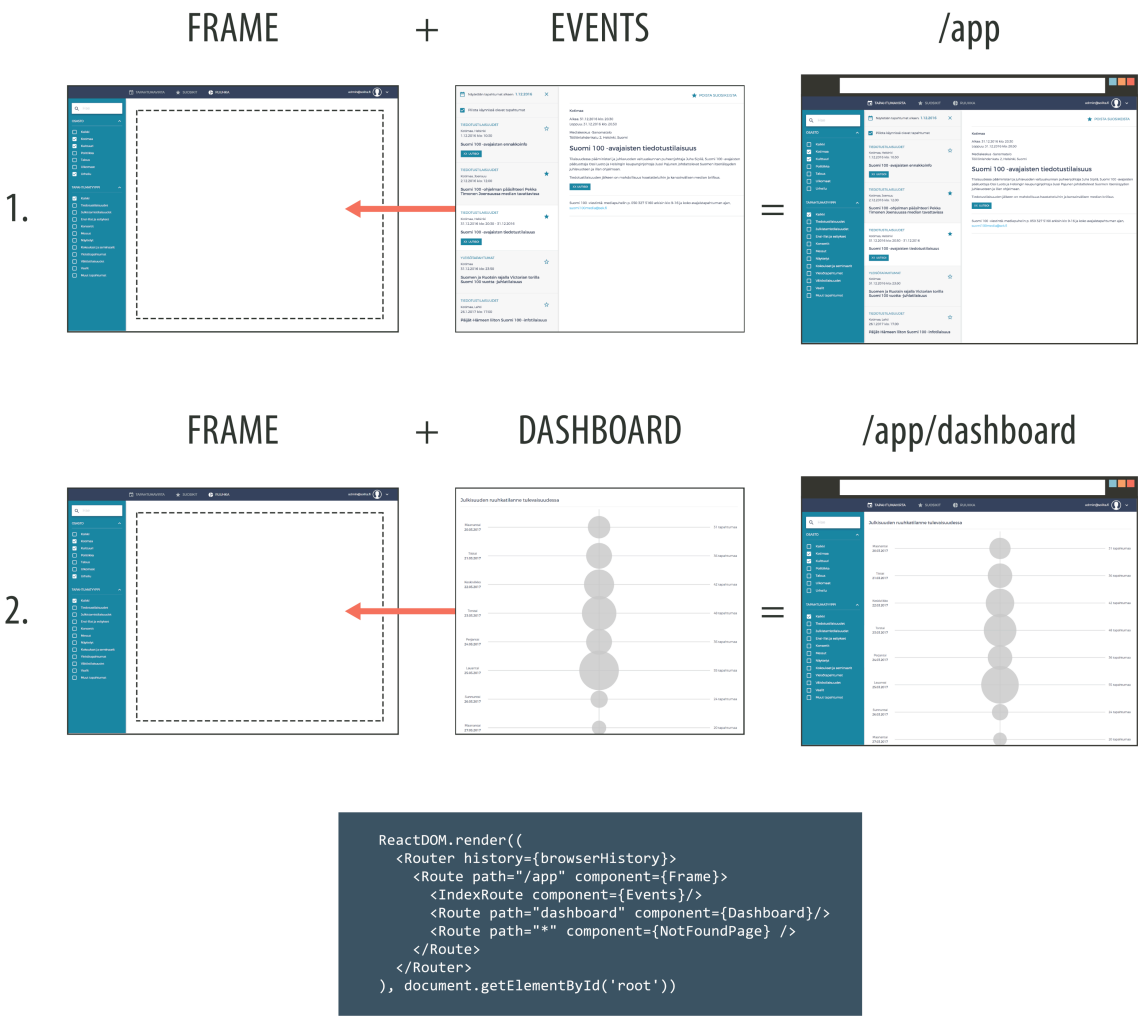
3.4 Reititys

Sovelluksen selainpään reititys mahdollistettiin JSX-syntaksia hyödyntävän React-Router-lisäkirjaston avulla. Jokainen sovelluksen reitti viittaa haluttuun komponenttiin, jonka avulla muodostetaan URL-osoitteen mukainen näkymä. Kuvassa 17 on esitetty kolme päätason reittiä, jotka ovat etusivu, kirjautuminen ja tapahtumasivu. Kaikissa kolmessa esimerkissä koko sivu muodostuu yhdestä ylemmän tason komponentista, joka sijoitetaan katkoviivalla merkittyyn alueeseen. Vaikka sivut ovat erillisiä, reititys toimii SPA-sovellusten perusperiaatteiden mukaisesti, eli URL-osoitteen tai näkymän vaihtuminen ei luo uutta pyyntöä palvelimelle.



Kuva 17. Selainpään reititys mahdollistaa eri näkymien näyttämisen URL-osoitteen perusteella ilman erillistä kutsua palvelimelle.

Tapahtumapalvelussa on edellä mainittujen lisäksi sivuja, joissa pyritään kattavasti hyödyntämään uudelleenkäytettäviä komponentteja muodostamalla näkymistä sisäkkäisiä kokonaisuuksia. Näin esimerkiksi käyttöliittymän raamit voidaan rakentaa erilliseen komponenttiin, jonka sisältämät osakokonaisuudet vaihtuvat käyttäjien tekemien toimintojen mukaisesti. Kuvassa 18 kohdassa 1 on esitettynä jo aiemmin nähty tapahtumasivu, joka sisältää raamit kokoavan ylä- ja sivupalkin sekä sisältöalueen. Tämä näkymä muodostaa URL-osoitteen oletuspolun, joka ilmaistaan IndexRoute-elementillä. Koska sovelluksen raamit muodostavat erillisen komponentin, niitä voidaan hyödyntää myös tapahtumaruuhkaa esittävällä sivulla. Kuvan 18 kohdassa 2 sisältöalueelle tuodaan esimerkin mukaisesti uusi komponentti, joka hyödyntää samoja tapahtumasivun raameja. Näkymien sisäkkäisyyttä voidaan tarvittaessa jatkaa puurakennemallin mukaisesti syvemmälle komponenttirakenteessa. Mikäli URL-osoitteelle ei löydy sovelluksesta vastinparia, ohjataan käyttäjä virhesivulle kuvan 18 koodiesimerkin mukaisesti asteriskilla (*).



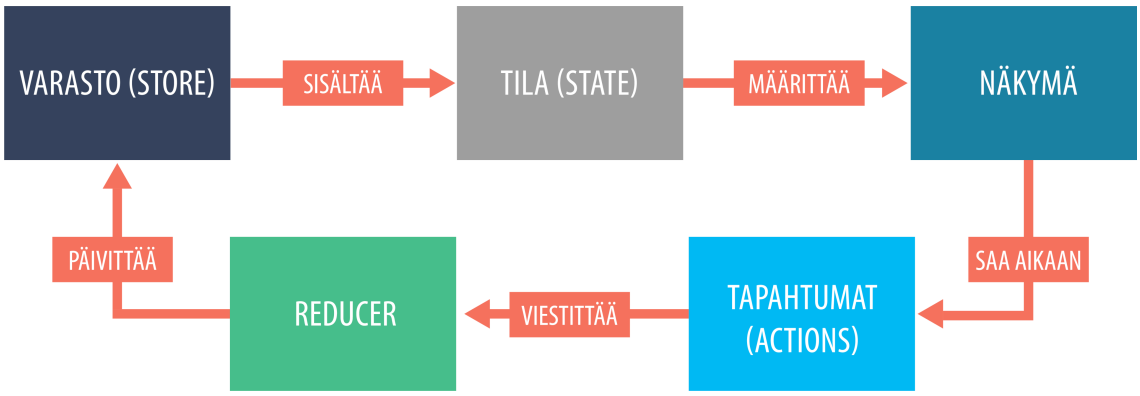

```
ReactDOM.render((
  <Router history={browserHistory}>
    <Route path="/app" component={Frame}>
      <IndexRoute component={Events}/>
      <Route path="dashboard" component={Dashboard}/>
      <Route path="*" component={NotFoundPage} />
    </Route>
  </Router>
), document.getElementById('root'))
```

Kuva 18. Reitityksen avulla tapahtumasovellukseen luotiin näkymiä, jotka hyödyntävät uudelleenkäytettäviä komponentteja.

3.5 Sovelluksen tila

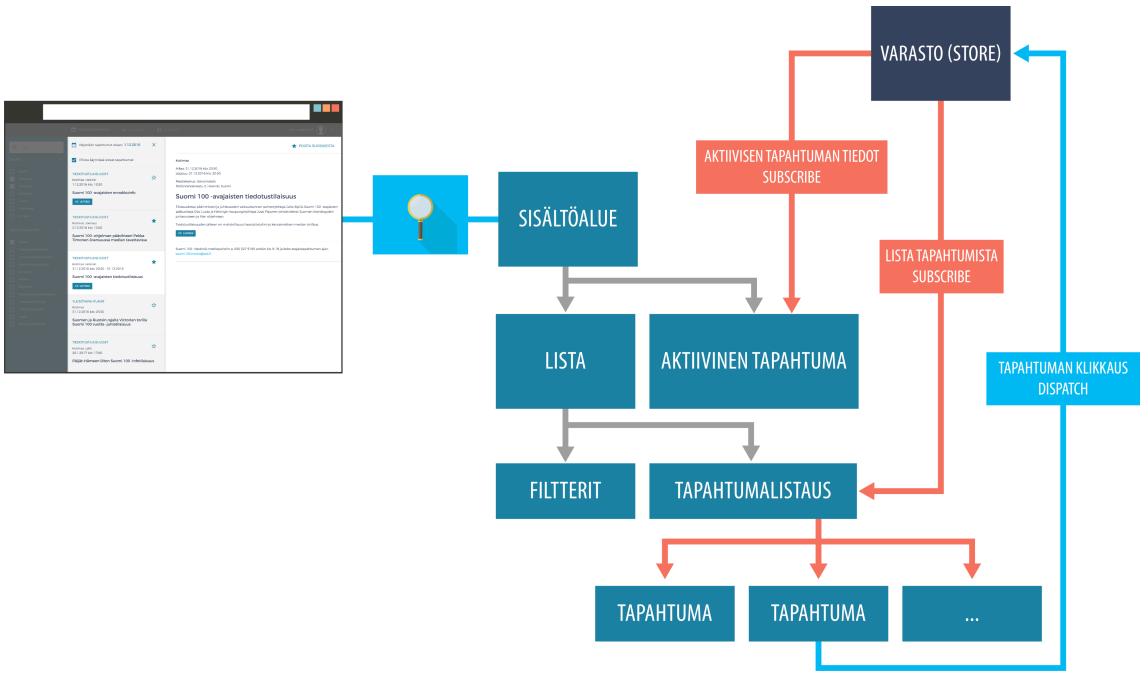
Reactissa komponentit itsessään voivat sisältää tarvittavat tiedot sovelluksen tilasta. Monimutkaisissa toteutuksissa komponentteihin hajautettu tila kuitenkin pidemmän päälle heikentää sovelluksen selkeyttä ja ylläpidettävyyttä. Tämän vuoksi tilanhallintaa selkeyttämään toteutuspohjaan otettiin Flux-arkkitehtuuria versioiva Redux. Reduxissa tilaa ei säilötä komponentteihin vaan objektiin, jota kutsutaan storeksi (varasto). Objektin rakenteeseen ei ole tarkempia määrityksiä, joten se voidaan luoda sovelluksen tarpeiden mukaisesti. [48.]

Reduxin tila on muuttumaton (immutable), minkä vuoksi koko objektista luodaan aina uusi versio muutoksen yhteydessä. Tämä helpottaa tilanmuutosten seurantaa, ennustettavuutta ja testausta. Muutos tilaan saadaan aikaan puhtailla funktioilla, joita Reduxissa kutsutaan reducereiksi. Puhtaalla funktiolla tarkoitetaan toimintoa, jota kutsuttaessa samojen argumenttien tulee palauttaa sama arvo. Tämän vuoksi tietokantakutsuja ei reducereissa tule tehdä. Reducerit ottavat sisäänsä parametreina aikaisemman tilan ja halutun muutoksen. Niiden avulla funktio palauttaa sovellukselle uuden tilan, jonka perusteella komponentit tarpeen vaatiessa päivitetään. Edellä kuvattu toimintamalli on esitetty kuvassa 19. On hyvä huomata, että Reduxin käyttö ei estä tilan säilömistä yksittäisiin komponentteihin. Tapauskohtaisesti yksittäisiä näkymään vaikuttavia asioita voi olla järkevä pitää komponentin sisällä. Suositeltavaa kuitenkin on tilan säilöminen Reduxiin, mikäli sitä tarvitaan useammassa kuin yhdessä komponentissa. [48.]



Kuva 19. Flux-ohjelmistoarkkitehtuuriin pohjautuva Redux hyödyntää vahvasti yksisuuntaista datasidontaa.

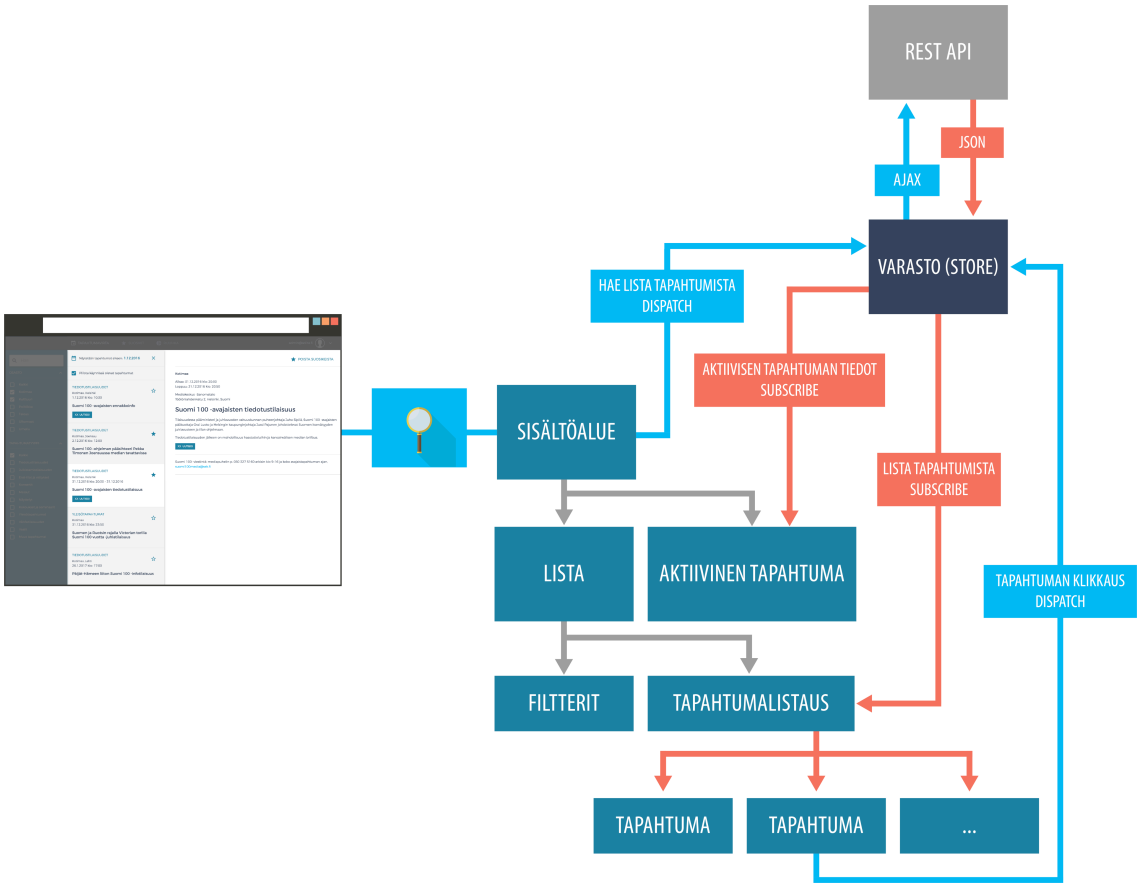
Kuvassa 20 on jo aiemmin esimerkeissä ollut sisältöalue, joka koostuu tapahtumia sisältävästä listasta ja aktiivisesta tapahtumasta. Eri tasoilla sijaitsevat komponentit pystyvät varastoon (storeen) yhdistymällä hakemaan yleisestä tilasta dataa ja kutsumaan tilamuutoksen aikaansaavia funktioita. Esimerkissä oranssit nuolet kuvaavat tilasta komponenttiin haettavia tietoja, jotka tässä tapauksessa ovat aktiivinen tapahtuma, tapahtumalistaus ja aktiivista tapahtuman vaihtoa kutsuva funktio. Kun funktiota kutsutaan, tieto välittyy varastoa muuttavalle reducerille, joka ottaa aikaisemman tilan ja halutun muutoksen palauttaen sovellukselle uuden tilan. Kuvan esimerkissä funktiota kutsutaan tapahtumaa klikattaessa ja se vaihtaa sisältöalueen aktiivisen tapahtuman.



Kuva 20. Reduxissa tila on säilötty varastoon (store), josta yksittäiset komponentit voivat sitä tarkastella (subscribe). Tilamuutokset saadaan aikaan komponenttien lähettämillä kutsuilla (dispatch).

Reduxin hallinnoidessa sovelluksen tilaan on loogista sisällyttää myös tilaan oleellisesti vaikuttavat Ajax-pyynnöt sen huolehdittavaksi. Koska oletuksena Redux ei tue asynkronista rakennetta, sovelluspohjaan lisätään Ajax-kutsujen mahdollistamiseksi redux-thunk. Lisäkirjaston avulla tapahtumakutsujat voivat palauttaa objektien ohella myös funktioita. Palautettavien funktioiden ei tarvitse olla puhtaita, jolloin myös asynkroniset rajapintakutsut ovat mahdollisia. Epäpuhtaiden asynkronisten kutsujen palautusfunktioina voidaan käyttää Reduxin tavallisia puhtaita funktioita tilan muuttamiseksi. [48.]

Tapahtumasovelluksen sisältö saadaan RESTful-mallisen rajapinnan kautta. Rajapintaa hyödynnetään tapahtumakutsujien avulla ja palvelimen vastatessa kutsutaan tilaa muuttavaa funktiota. Näin rajapinnan palauttama data saadaan varastoitua sovelluksen storeen komponenttien hyödynnettäväksi. Varsinaiset Ajax-kutsut suoritetaan lupauspohjaista Fetch-rajapintaa hyödyntäen. Kuvassa 21 on esitetty asynkronisten tapahtumien rakenne sovelluspohjassa. Esimerkissä sisältöalue pyytää Reduxin tapahtumakutsujaa hakemaan rajapinnasta listan tapahtumista. Palvelimen palauttama JSON-muotoinen data varastoidaan Reduxin storeen, jonka pohjalta luodaan uusi tila ja tapahtumalistauskomponentin näkymä päivitetään.



Kuva 21. Redux-thunkin avulla mahdollistettiin tapahtumasovelluksen asynkroniset palvelinkutsut sovelluksen tilanhallinnassa.

3.6 Tuotantoonvienti

Tuotantoonviennissä hyödynnettiin avoimeen lähdekoodiin perustuvaa Docker-järjestelmää. Se mahdollistaa sovelluksien kehittämisen, jakamisen ja suorittamisen monipuolisesti eri alustoilla, muun muassa kehittäjän työasemalla, eri pilvipalveluissa ja virtuaalikoneilla. Docker paketoi sovelluksen levykuvaksi (Docker image), joka sisältää kaikki sovelluksen tarvitsemat ominaisuudet lähdekoodista ajonaikaisiin kirjastoihin ja järjestelmätyökaluihin. Varsinainen sovellus suoritetaan Docker-kontissa (Docker container), joka on eristetty ympäristö ja sisältää oman tiedostojärjestelmän ja ympäristömuuttujat. [49.] Tapahtumasovelluksen mikropalveluarkkitehtuurissa jokainen erillinen palvelu paketoitiin omaan Docker-konttiin, ja ne sijoitettiin Amazonin pilvipalvelinympäristöön.

Kehittämisen tueksi Amazoniin luotiin erikseen testaus- ja tuotantoympäristö. Käytössä olleen jatkuvan toimittamisen mukaisesti uudet ominaisuudet testattiin ensin kehitysympäristössä, minkä jälkeen ne julkaistiin tuotantoversiossa. Jatkossa osa nykyisin manuaalisesti tehtävästä testauksesta ja tuotantoonviennistä on tarkoitus automatisoida Jenkins-palvelua hyödyntäen. Jenkinksin avulla voidaan esimerkiksi automatisoida versionhallintaan ilmestyneiden muutosten myötä uuden Docker-levykuvan luonti, ohjelmiston automaattitestien suorittaminen ja Docker-kontin siirto Amazonin pilvipalvelinympäristöön. [50.]

4 Yhteenveto

Web on vuosien saatossa kasvanut yksinkertaisten dokumenttien esittäjästä monipuolisten sovellusten alustaksi. SPA-arkkitehtuurin avulla rakennetut web-sovellukset yhdistävät monipuolisen sisällön sulavaan käyttökokemukseen. Arkkitehtuurissa toimintalogiikasta huolehtii selaimessa JavaScript, joka mahdollistaa yhä suurempien ja mutkikkaampien kokonaisuuksien toteuttamisen. JavaScriptin käytön kasvaessa uusia kehitystä helpottavia ohjelmistokehyksiä, kirjastoja ja työkaluja on syntynyt mahdollistamaan yhä monipuolisemmat sovellukset. Insinöörityössä perehdyttiin näihin SPA-arkkitehtuurin teknologioihin ja toimintatapoihin ja pyrittiin tuomaan esiin JavaScript-painotteisen sovelluskehityksen nykytilaa asiakkaalle toteutetun tapahtumapalvelun kautta.

Kehitetty tapahtumapalvelu toteutettiin mikropalvelukokonaisuutena, joka sisältää runsaasti uudelleenkäytettäviä komponentteja ja helposti jatkokehitettäviä ratkaisumalleja. Insinöörityössä keskityttiin käyttöliittymäkerroksen mikropalveluun ja sen toteutusprosessiin. SPA-toteutuksena se ilmentää monipuolista selaimessa toimivaa JavaScript-toteutusta, joka rakentuu kirjastopohjaisen modulaarisen kokonaisuuden varaan. Toteutuksessa on kattavasti otettu huomioon joukko uusimpia kehitystä helpottavia työkaluja ja menetelmiä.

Projektin onnistumisen kannalta sovelluksen modulaarisella rakenteella oli oleellinen rooli. Helposti muokattavien pienempien kokonaisuuksien myötä sovelluksesta saatiin muutamassa viikossa käyttöön ensimmäinen versio, jota lähdettiin jatkokehittämään yhdessä loppuasiakkaiden kanssa. Heiltä saadun palautteen perusteella toteutukseen tehtiin jatkuvasti muutoksia ja sovelluksen toiminnallisuudet muuttuivat jatkuvasti. Modulaa-

risuus mahdollisti nopean reagoinnin palautteeseen, mutta aiheutti myös ongelmia. Sovelluksen muuttuva rakenne ja kasvavat vaatimukset pakottivat jatkuvaan koodin uudelleenkirjoittamiseen (code refactoring). Toisistaan erillisten mikropalveluiden kehitys aiheutti myös satunnaisesti yhteensopivuusongelmia, jotka pystyttiin kuitenkin korjaamaan jatkuvan tuotantoonviennin myötä nopeasti. Muutosten yhteydessä tapahtuvien virheiden välttämiseksi ja kasvavan kokonaisuuden hallittavuuden helpottamiseksi jatkokehityksessä on tarkoitus ottaa käyttöön kattavammat testausmenetelmät. Tavoitteena on, että tuotantoonvienti onnistuu vain, jos jokainen osio läpäisee sille kirjoitetut automaattitestit. Testauksen avulla määritetään sovelluksen osioille ehdot, jotka sen tulee täyttää. Näin varmistetaan, että muutokset eivät aiheuta virhetilanteita jo aiemmin luodussa lähdekoodissa.

Tapahtumasovellus julkaistiin tammikuussa 2017, ja kohderyhmän suuren kiinnostuksen myötä tavoite idean toimivuuden testauksesta voidaan todeta saavutetun. Luotu palvelu on selkeästi parantanut luotettavan ja reaaliaikaisen tapahtumatiedon läpinäkyvyyttä ja löydettävyyttä. Palvelun kautta viestinnän tekijät pääsevät helposti ja suoraan käsiksi korkeatasoiseen uutistoimittajien laatimaan ja päivittämään tapahtuma- ja suunnittelutietoon. Loppukäyttäjiltä on tullut myös runsaasti positiivista palautetta palvelun ulkonäöstä, nopeudesta ja käytettävyydestä. Onnistuneesta toteutuksesta kertoo myös ehdokkuus vuoden 2017 parhaaksi yritystoimintaa edistäväksi sovellukseksi Grand One -kilpailussa.

Lähteet

1 Ingalls, Dan & Mikkonen, Tommi & Palacz, Krzysztof & Taivalsaari, Antero. 2008. Web Browser as an Application Platform: The Lively Kernel Experience. Verkkodokumentti. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.2713&rep=rep1&type=pdf>. Luettu 15.3.2016.

2 HTML5 Differences from HTML4. 2014. Verkkodokumentti. W3C. <https://www.w3.org/TR/html5-diff/>. 9.12.2014. Luettu 10.3.2016.

3 HTML5 A vocabulary and associated APIs for HTML and XHTML. 2014. Verkkodokumentti. W3C. <https://www.w3.org/TR/2014/REC-html5-20141028/dom.html>. 28.10.2014. Luettu 10.3.2016.

4 Peterson, Clarissa. 2012. Accessibility in HTML5. Verkkodokumentti. <http://www.clarissapeterson.com/2012/11/html5-accessibility/>. 19.11.2012. Luettu 1.4.2016.

5 Albers, Brian & Lubbers, Peter & Salim, Frank. 2010. Pro HTML5 Programming. New York: Apress.

6 HTML5. 2016. Verkkodokumentti. Mozilla Developer Network. <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>. 1.2.2016. Luettu 11.3.2016.

7 Document Object Model (DOM). 2016. Verkkodokumentti. Mozilla Developer Network. <https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model%20/>. 26.2.2016. Luettu 11.3.2016.

8 Web APIs. 2016. Verkkodokumentti. Mozilla Developer Network. <https://developer.mozilla.org/en-US/docs/Web/API>. 10.2.2016. Luettu 14.3.2016.

9 CSS SPECIFICATIONS. 2016. Verkkodokumentti. W3C. <https://www.w3.org/Style/CSS/specs>. 12.2.2016. Luettu 10.3.2016.

10 Bradley, Steven. 2012. Sass And LESS: An Introduction To CSS Preprocessors. Verkkodokumentti. <http://vanseodesign.com/css/css-preprocessors/>. 9.4.2012. Luettu 10.3.2016.

11 PostCSS. 2016. Verkkodokumentti. PostCSS. <http://postcss.org/docs>. Luettu 17.4.2016.

12 About JavaScript. 2015. Verkkodokumentti. Mozilla Developer Network. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript >. 30.10.2015. Luettu 20.2.2016.

13 Flanagan, David. 2011. JavaScript The Definitive Guide. Sebastopol: O'Reilly.

14 Designing a Language in 10 Days. 2012. Verkkodokumentti. IEEE Computer Society. <https://www.computer.org/csdl/mags/co/2012/02/mco2012020007.html>. 1.2.2012. Luettu 20.2.2016.

15 Wayner, Peter. 2011. JavaScript conquers the server. Verkkodokumentti. <http://www.infoworld.com/article/2621690/application-development/javascript-conquers-the-server.html>. 25.5.2011. Luettu 21.2.2016

16 ECMAScript 2015 Language Specification. 2015. Verkkodokumentti. ECMA International. <http://www.ecma-international.org/ecma-262/6.0/>. Luettu 21.2.2016.

17 Emmi, Scott. 2016. SPA Design and Architecture. New York: Manning.

18 Fink, Gil & Flatow, Ido. 2014. Pro Single Page Application Development. New York: Apress.

19 Kamenzky, Nicolai. 2015. "Precomposing" a SPA may become the Holy Grail to SEO. Verkkodokumentti. <http://www.analog-ni.co/precomposing-a-spa-may-become-the-holy-grail-to-seo>. 9.1.2015. Luettu 12.4.2016.

20 Understanding web pages better. 2014. Verkkodokumentti. Google. <https://webmasters.googleblog.com/2014/05/understanding-web-pages-better.html>. 23.5.2014. Luettu 12.4.2016.

21 Garrett, Jesse. 2005. Ajax: A New Approach to Web Applications. Verkkodokumentti. <https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php>. 18.2.2005. Luettu 20.3.2016.

22 Speed Up Your AJAX-based Apps with JSON. 2006. Verkkodokumentti. DevX. <http://www.devx.com/webdev/Article/32651>. 5.10.2006. Luettu 20.3.2016

23 Using Fetch. 2016. Verkkodokumentti. Mozilla Developer Network. <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch>. 18.4.2016. Luettu 19.4.2016.

24 Fielding, Roy. 2000. Architectural Styles and the Design of Network-based Software Architectures. Verkkodokumentti. <http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>. Luettu 22.3.2016.

25 Elliot, Eric. 2014. Programming JavaScript Applications. Sebastopol: O'Reilly.

26 Flux Overview. 2015. Verkkodokumentti. Facebook. <https://facebook.github.io/flux/docs/overview.html#content>. Luettu 19.3.2016.

27 Deprecating our AJAX crawling scheme. 2015. Verkkodokumentti. Google. <https://webmasters.googleblog.com/2015/10/deprecating-our-ajax-crawling-scheme.html>. 14.10.2015. Luettu 12.4.2016.

28 Brett-Bowen, Rhys. 2013. A Library or a Framework? Verkkodokumentti. <http://modernjavascript.blogspot.fi/2013/04/a-library-or-framework.html>. 17.4.2013. Luettu 28.3.2016.

29 Front-end JavaScript frameworks. 2017. Verkkodokumentti. Github. <https://github.com/showcases/front-end-javascript-frameworks/>. Luettu 12.4.2017.

30 Stackoverflow Tags. 2017. Verkkodokumentti. Stackoverflow. <http://stackoverflow.com/tags/>. Luettu 12.4.2017.

31 Google Trends. 2017. Verkkodokumentti. Google Trends. <https://trends.google.fi/trends/explore?q=angularjs,ember.js,reactjs,vue.js/>. Luettu 12.4.2017.

32 Guide to AngularJS Documentation. 2016. Verkkodokumentti. AngularJS. <https://docs.angularjs.org/guide/>. Luettu 17.3.2016

33 Documentation overview. 2016. Verkkodokumentti. AngularJS. <https://angular.io/docs/ts/latest/>. Luettu 17.3.2016.

34 Elliot, Eric. 2016. Angular 2 vs React: The Ultimate Dance Off. Verkkodokumentti. https://medium.com/javascript-scene/angular-2-vs-react-the-ultimate-dance-off-60e7dfbc379c#.7nfjxlbav/>. 2.10.2016. Luettu 2.3.2017.

35 Guides and Tutorials. 2016. Verkkodokumentti. Ember. <https://guides.emberjs.com/v2.4.0/>. Luettu 17.3.2016.

36 Ember Sponsors and Friends. 2016. Verkkodokumentti. Ember. <http://emberjs.com/sponsors/>. Luettu 17.3.2016.

37 Guides. 2016. Verkkodokumentti. React. <https://facebook.github.io/react/docs/>. Luettu 15.3.2016.

38 Dawson, Chris. 2014. JavaScript's History and How it Led To ReactJS. Verkkodokumentti. <http://thenewstack.io/javascripts-history-and-how-it-led-to-reactjs/>. Luettu 18.2.2016.

39 A Framework for Building Native Apps Using React. 2016. Verkkodokumentti. React Native. <https://facebook.github.io/react-native/>. Luettu 18.2.2016.

40 Guide. 2017. Verkkodokumentti. Vue.js. <https://vuejs.org/v2/guide/>. Luettu 16.3.2017.

41 Docs. 2016. Verkkodokumentti. Npm. <https://docs.npmjs.com/>. Luettu 30.3.2016.

42 Kasireddy, Preethi. 2016. JavaScript Modules Part 2: Module Bundling. Verkkodokumentti. <https://medium.freecodecamp.com/javascript-modules-part-2-module-bundling-5020383cf306#.d3oxsjo6j>. 2.2.2016. Luettu 30.3.2016.

43 Goel, Naman. 2015. Browserify VS Webpack - JS Drama. Verkkodokumentti. <http://blog.namangoel.com/browserify-vs-webpack-js-drama>. 16.1.2015. Luettu 30.3.2016.

44 Zen, Cassio. 2015. Appendix A: Webpack for React. Verkkodokumentti. <http://www.pro-react.com/materials/appendixA/>. Luettu 25.2.2016.

45 User Guide. 2016. Verkkodokumentti. ESLint. <http://eslint.org/docs/user-guide/>. Luettu 20.4.2016.

46 Chacon, Scott & Straub Ben. 2014. Pro Git. Mountain View: Apress.

47 Abramov, Dan. 2015. React Components, Elements, and Instances. Verkkodokumentti. <https://medium.com/@dan_abramov/react-components-elements-and-instances-90800811f8ca#.azumn6mn0>. 14.12.2015. Luettu 16.4.2016.

48 Redux. 2016. Verkkodokumentti. Redux. <http://redux.js.org/>. Luettu 23.4.2016.

49 Docker Documentation. 2017. Verkkodokumentti. Docker. <https://docs.docker.com/>. Luettu 21.3.2017.

50 Jenkins Documentation. 2017. Verkkodokumentti. Jenkins. <https://jenkins.io/doc/>. Luettu 21.3.2017.