TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JARI KUISMA

DOM-OPTIMOINTI MONIALUSTAISESSA WEB-SUUNNITTELUSSA

Diplomityö

Tarkastaja: prof. Petri Ihantola
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 26. huhtikuuta 2017

TIIVISTELMÄ

JARI KUISMA
Tampereen teknillinen yliopisto
Diplomityö, 69 sivua, 2 liitesivua
Toukokuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Petri Ihantola

Avainsanat: front-end, JavaScript, DOM, HTML, React, Angular 2, mobiili

Web-sivut ovat kehittyneet staattisista sivukokoelmista kohti dynaamisia JavaScriptillä toimivia web-sovelluksia. DOM-mallin luominen sivusta on olennainen operaatio rakennettaessa HTML- ja CSS-dokumenteista kuvausta web-sivun sisällöstä. Tämä tapahtuu usean operaation seurauksena ja jokainen vaihe luotaessa DOM:ia saattaa aiheuttaa pullonkaulan, joka heikentää sivun toimintaa. Operaation hidastuessa selaimen nopeus vastata käyttäjän pyyntöihin laskee ja käyttäjäkokemus heikkenee. Ongelma korostuu suorituskyvyltään heikommissa laitteissa kuten älypuhelimissa ja tableteissa. Tilastot osoittavat, että internetiä käytetään jo yleisemmin mobiililaitteella kuin perinteisellä työpöytäselaimella. Tämän takia suorituksen optimointiin myös mobiililaitteissa tulee kiinnittää erityistä huomiota. Suorituskykyä ja web-sovellusten toimivuutta yritetään optimoida useilla erilaisilla keinoilla. Tässä työssä on tutkittu millaisia vaikutuksia erilaisilla DOM:in optimointitavoilla on. Erilaiset JavaScript-sovelluskehykset ja -kirjastot ratkaisevat suorituskykyongelmia eri tavoilla ja tämän työn puitteissa vertaillaan tavallisen DOM toteutuksen suorituskykyä suhteessa virtual DOM ja shadow DOM toteutuksiin.

Tutkimuksessa toteutettiin kolme web-sovellusta, jotka suorittavat DOM:ia kuormittavia operaatioita luomalla joukon testielementtejä. Elementtien lisäys-, muokkaus- sekä poisto-operaatioiden kestoa vertaillaan erikokoisilla HTML-elementtien testijoukoilla. Vertailut tehtiin kolmella erilaisella laiteella: kannettavalla tietokoneella, tablet-laitteella ja älypuhelimella. Testauksessa käytettiin Chrome-selaimen Developer Tools:in Time Line -työkalua, jolla mitattiin käyttäjän toiminnosta kuluvaa aikaa käyttöliittymän ruudunpäivitysnopeuden normalisoitumiseen operaation päätyttyä.

Tuloksista nähdään, että suorituskyvyllä on suuri vaikutus operaatioiden tehokkuuteen erilaisilla laitteilla. Harkitusti tehdyllä toteutusratkaisun valinnalla voi olla suurikin vaikutus suoritusaikaan ja näin ollen myös käyttäjäkokemukseen. Työhön valituista toteutusratkaisuista React ja sen toteuttama virtual DOM suoriutuivat tehokkaimmin kaikilla alustoilla lähes kaikissa toiminnoissa. Varsinaista toteutustapaa valittaessa on kuitenkin tärkeä huomioida ratkaistavan ongelman ympäristö ja pohtia mitkä mahdollisista tehokkuuden pullonkauloista ovat merkittävimpiä. Jokaisella ratkaisulla on omat etunsa, mutta JavaScript-sovelluskehysten ja -kirjastojen yleisenä etuna voidaan pitää parempaa testattavuutta ja yksinkertaisempaa uudelleenkäytettävän koodin luomista.

ABSTRACT

JARI KUISMA
Tampere University of Technology
Master of Science Thesis, 69 pages, 2 Appendix pages
May 2017
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Assistant-Professor Petri Ihantola

Keywords: front-end, JavaScript, DOM, HTML, React, Angular 2, mobile

Web pages have evolved from static page collections into dynamic JavaScript powered web applications. Creating a DOM model of the web page is an important operation when creating a web page presentation of HTML and CSS documents. This process consists of multiple operations and every stage introduces a possible performance issue. When this operation slows down, the browser's ability to correspond to the user's requests lowers and this directly affects the level of user experience. The problem is more crucial with lower end devices such as smartphones and tablets. Statistics show that using internet with mobile devices has grown to be a bigger trend than traditional use of desktop browser which is why special interest should be paid to mobile performance. There have been attempts to improve the performance and usability of web applications with multiple ways and this thesis concentrates on improving performance of DOM. Various JavaScript frameworks and libraries have presented different solutions for the performance issues and this thesis focuses on three different solutions which are traditional DOM, shadow DOM and virtual DOM.

Three web applications that run performance heavy tasks on DOM were created for the purpose of the thesis. The applications generate test elements and the creating, editing and removing operation times are compared. The comparisons are done with laptop, tablet and smart phone. For the testing the Chrome browser's Time Line tool of the Developer Tools was used. The measured variable was the time from the user operation into the normalization of the frame refresh rate after the operation.

From the results, it can be seen that the available performance capacity bears great impact on the efficiency of the operations run on different devices. Well-chosen implementation can greatly affect the run time and this way also the user experience. From the implementations chosen, React and the virtual DOM solution it uses turned out to be the best performing in most of the operations. When choosing the implementation method, it is important to find out the domain and which are the actual points that can cause performance issues. Each of the implemented solutions had its own strengths but in general JavaScript frameworks and libraries can be seen to promote better testability and simpler creation of reusable code.

ALKUSANAT

Festina lente, muuten työ olisi valmistunutkin jo aiemmin. Haluan kiittää täydestä sydämestä kaikkia ystäviäni vertaistuesta ja muunkin ajateltavan tarjoamisesta, Bitwise-työnantajaani työlle varatusta ajasta, kollegaani Jani Tarmilaa oikolukemisesta, ohjaajaani Petri Ihantolaa erityisesti motivoinnista web-ohjelmoinnin kursseilla ja puolisoani sekä koiraamme jaksamisesta pitkän kevään aikana.

Tampereella, 24.5.2017

Jari Kuisma

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. MODERNI WEB-SUUNNITTELU 5
2.1 Web-kehitys 5
2.1.1 Hypertekstikokoelmista dynaamisiksi sovelluksiksi 5
2.1.2 Selainlaitteiden muutos 6
2.1.3 Responsiivinen web-suunnittelu 7
2.1.4 Responsiiviset tekniikat 9
2.2 Web-sovelluskehitys 11
2.2.1 Yhden sivun sovellukset (SPA) 11
2.2.2 MVC-malli 12
2.3 JavaScript 14
2.3.1 Erityispiirteet 14
2.3.2 Nimet ja versiot 15
2.3.3 Front-end-kirjastot 16
2.3.4 Komponentit 18
3. DOKUMENTTIEN JÄSENTÄMINEN 20
3.1 Dokumenttioliomalli (DOM) 20
3.2 Millainen DOM on? 21
3.3 Virtuaalinen DOM (V-DOM) 23
3.3.1 DOM:in päivittäminen 23
3.3.2 Renderöinti 24
3.3.3 Tapahtumien hallinta 27
3.4 Shadow DOM 27
3.4.1 Varjokapselointi 28
3.4.2 Koostaminen 29
3.4.3 Komponentin tyyli 30
3.4.4 Tapahtumien hallinta 32
4. SOVELLUSTEN KUVAUS 33
4.1 Toiminnallisuus 33
4.2 Toteutetut sovellukset 37
4.2.1 Angular 2 -sovellus 37
4.2.2 React-sovellus 43
4.2.3 jQuery ja HTML -sovellus 49
4.3 Mittausten toteutus 52
5. TULOKSET JA ARVIOINTI 56
6. JOHTOPÄÄTÖKSET 61

LIITE 1: YHTEISET FUNKTIOT

KUVALUETTELO

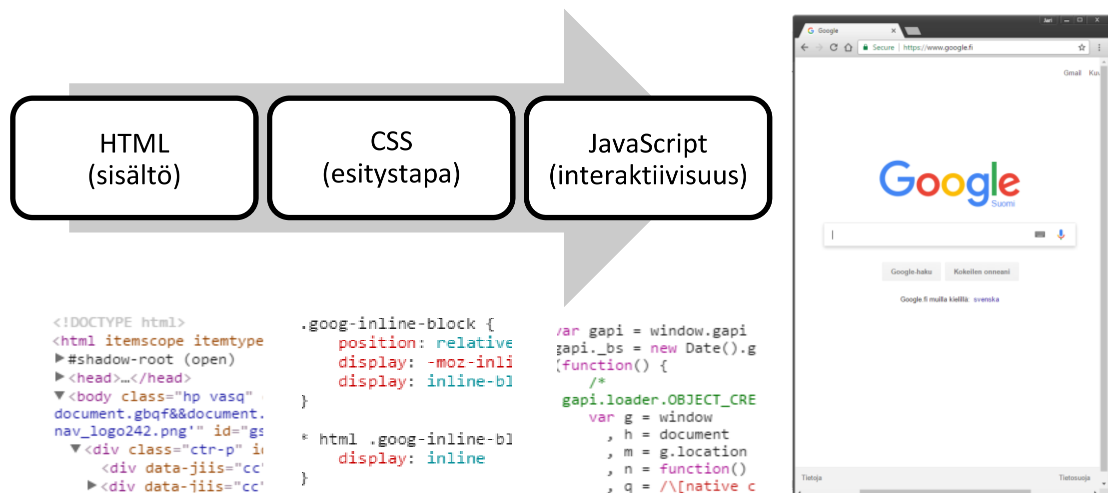
Kuva 1. Web-ohjelmoinnin ydintekniikat rakentavat yhdessä web-sivun. 1
Kuva 2. Pikselin ruudulle määrittelevä putki. Perustuen: (Lewis, 2017). 3
Kuva 3. HTTP-pyynnön lähettäminen. Perustuen: (King, 2008). 6
Kuva 4. Lautapelikahvila.fi-verkkosivu toimii hyvänä esimerkkinä responsiivisesta verkkosivusta. Kuvassa sama näkymä sekä työpöytä- että mobiiliselaimella. 8
Kuva 5. Erilaisia resoluutioita 20 suosituimmassa laitteessa (visualisointi (Jehl, 2014), tiedot peräisin lähteestä OpenSignal, 2013). 8
Kuva 6. Mediakyselyn ratkaiseminen (W3C, 2016a). 10
Kuva 7. Tavallisen web-sivun ja yhden sivun sovelluksen kommunikointi palvelimen kanssa (Wasson, 2013). 12
Kuva 8. MVC-mallin "oppikirjaesimerkki" (Syromiatnikov & Weyns, 2014). 13
Kuva 9. MVC-malli web-sovelluksissa (Syromiatnikov & Weyns, 2014). 14
Kuva 10. TypeScript laajentaa ES5- ja ES6-kielten määrittämää JavaScript standardia tyyppimäärittelyillä. Perustuen: (Fullstack.io, 2017). 16
Kuva 11. Front-end ohjelmistokehitys. Perustuen: (Wodehouse, 2015). 16
Kuva 12. Erilaisten front-end-kirjastojen kiinnostavuus ja käyttöaste (Greif, 2016).m 17
Kuva 13. Wikipedian etusivun mahdollinen karkea jako komponentteihin. 19
Kuva 14. Selain rakentaa HTML- ja CSS-dokumenttien pohjalta DOM-puun, jonka käsittelylle DOM toimii rajapintana. Perustuen: (Mobidev, 2014). 21
Kuva 15. Dokumenttioliomallin puumainen rakenne (Stepp, et al., 2012). 22
Kuva 16. Reactin DOM-puun tasoihin perustuva vertailu. Perustuen: (Chedeau, 2013). 24
Kuva 17. Likaisten solmujen merkitseminen ja niiden renderöinti. Perustuen: (Chedeau, 2013). 25
Kuva 18: Alipuurenderöinti likaisten solmujen perusteella. Perustuen: (Chedeau, 2013). 26
Kuva 19. Dokumentti voi koostua useista toisistaan eristetyistä varjopuista. Perustuen: (Dodson, 2013). 28
Kuva 20. Yksi kolmesta toteutetusta sovelluksesta työpöydän selainikkunassa. 34
Kuva 21. Chromen DevTool:sin käyttö testauksessa. 54
Kuva 22. Mittaustulokset puhelimella. 57
Kuva 23. Mittaustulokset tabletilla.. 59
Kuva 24. Mittaustulokset kannettavalla tietokoneella. 60

OHJELMAKOODILUETTELO

Ohjelmakoodi 1. Varjopuusta luodaan DOM-rakenne käyttäjän toteutuksen perusteella. 29
Ohjelmakoodi 2. Host-selektorin käyttö esimerkkitapauksissa. 31
Ohjelmakoodi 3. Tyyli alueelle, johon yksittäisen elementit renderöidään. 36
Ohjelmakoodi 4. Esimerkki mediakyselyistä, joita käytetään responsiivisuuden määrittämiseen erilaisilla näytönleveyksillä. 36
Ohjelmakoodi 5. Yksittäisen renderöitävän row-item-luokkaisen elementin tyyli. 37
Ohjelmakoodi 6. RowItemComponent:in määrittely Angular-toteutuksessa. 39
Ohjelmakoodi 7. Angular-toteutuksen RowItem-komponentin HTML-malli. 40
Ohjelmakoodi 8. AppComponent:in perusrakenne Angular 2 -sovelluksessa. 41
Ohjelmakoodi 9. AppComponent:in toteuttama rajapinta RowItem:ien muokkaamiseen Angular 2 -sovelluksessa. 42
Ohjelmakoodi 10. AppComponent:in HTML-määrittely Angular 2 -sovelluksessa. 43
Ohjelmakoodi 11. Toteutus RowItem-komponentista Reactilla. 45
Ohjelmakoodi 12. Toteutus RowItem:in Update-funktiosta Reactilla. 46
Ohjelmakoodi 13. Reactin renderItems-funktio, joka luo yksittäiset RowItem-komponentit. 47
Ohjelmakoodi 14. Reactin App-komponentin reftsToArray-funktio, joka palauttaa yksittäisiin RowItem-komponentteihin viittaukset taulukossa. 47
Ohjelmakoodi 15. React-toteutus App-komponentista. 48
Ohjelmakoodi 16. React-toteutus App-komponentin rajapinnasta RowItem:ien muokkaamiseen ja näkymän päivittämiseen. 49
Ohjelmakoodi 17. Perinteisessä toteutuksessa elementin lisääminen näkymään. 51
Ohjelmakoodi 18. Perinteisen toteutuksen yksittäistä elementtiä päivittävä funktio. 51
Ohjelmakoodi 19. Isäntäelementin rajapinta perinteisessä toteutuksessa. 52

1. JOHDANTO

Web-sivu rakentuu usean teknologian yhteistyönä. Jokainen selaimen näyttämä sivu koostuu ainakin kuvauksesta sivun sisällöstä. Tämä kuvaus löytyy HTML-tiedostosta *(HyperText Markup Language)* ja se ladataan palvelimelta, kun selaimella siirrytään tiettyyn resurssiin eli verkko-osoitteeseen. Kuvassa 1 näkyy sivu `http://www.google.fi`, jonka HTML-dokumentti koostuu rakenteen määrittelystä esimerkiksi lomakkeen hakusanan syöttökentälle ja sen alla oleville painikkeille. HTML-sivu koostuu elementeistä, joista jokaisella on omat ominaisuutensa ja ne esitetään selaimessa yksikäsitteisellä tavalla. Tämän jälkeen kuvataan CSS:n *(Cascaded Style Sheets)* avulla miltä web-sivun tulee näyttää. Kuvan 1 esimerkissä CSS määrittelee muun muassa sivun fontin ja alapalkin värin. Viimeisenä sivuston ominaisuuksiin vaikuttaa JavaScript. JavaScript on ohjelmointikieli, jota käyttämällä selainikkunassa voidaan reagoida käyttäjän toimintoihin muuttamalla sivun rakennetta, ulkoasua ja toiminnallisuutta. Ilman JavaScriptiä web-sivut vain esittäisivät ennalta määriteltyä sisältöä. Esimerkiksi kuvassa 1 mikrofoni-painikkeen klikkaaminen antaa mahdollisuuden syöttää hakutermejä käyttäen puhetta. HTML, CSS ja JavaScript muodostavat yhdessä pohjan julkaisemiselle web-ympäristössä.



Kuva 1. *Web-ohjelmoinnin ydintekniikat rakentavat yhdessä web-sivun.*

Moderni web-suunnittelu on muuttunut staattisten sivukokoelmien ylläpitämisestä kohti dynaamisten web-sivujen suunnittelua. Webin käyttäjät odottavat sivustojen olevan interaktiivisia ja toimivan nopeasti (Kinlan, 2014). Uudet tavat toteuttaa web-sivuja sekä mobiili- että työpöytäympäristöihin ovat kasvattaneet suosiotaan. Samoja web-sivuja käytetään erilaisilla alustoilla ja erikokoisilla näytöillä. Erityisesti mobiililatteisiin liittyy suorituskykyrajoituksia esimerkiksi suorittimien, akkujen ja datayhteyksien puutteiden takia (Meier, 2012). Työpöytäympäristöissä on myös paljon eroja, esimerkiksi niin käytetyn selaimen kuin järjestelmän suorituskyvyn puolesta. Mobiililaitteiden määrä on kasvanut ja internetin käyttäminen mobiiliselaimella on yleistynyt viimeisten viiden vuoden aikana. Tilastokeskuksen (2015) tutkimuksen mukaan Suomessa jo 69 prosentilla 16–89-vuotiaista on omassa käytössään älypuhelin, joista lähes kaikki ovat kosketusnäytöllisiä. Jo pelkästään vuodesta 2014 vuoteen 2015 älypuhelinten käyttäjien osuus kasvoi kuudella prosenttiyksiköllä. Älypuhelimen käyttämisen arkipäiväistyminen ja laitteiden käytettävyyden sekä internet-palveluiden kehittyminen näkyvät kasvaneena webin käyttönä laitteilla. Vuonna 2012 älypuhelinten omistajista vain runsas 60 prosenttia käytti älypuhelimen internet-yhteyttä viikoittain, kun taas vuonna 2015 näin toimi jo 90 prosenttia älypuhelinten käyttäjistä. Web-sivujen mobiilikäyttö onkin tärkeää ottaa huomioon sivuja suunniteltaessa.

Dokumenttioliomalli *(Document Object Model)* eli DOM on rajapinta HTML-dokumenttien käsittelyyn. Dokumentti voi tarkoittaa useita eri asioita web-sivusta laajaan tietovarastoon, mutta kummassakin tapauksessa DOM:ia käytetään tiedon loogisen rakenteen esittämiseen. Sen avulla määritetään myös miten tietoa voidaan käyttää ja hallita. DOM:in avulla voidaan rakentaa dokumentteja, navigoida niiden rakenteessa sekä lisätä, muuttaa ja poistaa sisältöä. DOM:in rakentaminen ja muokkaaminen ovat tärkeitä osa-alueita interaktiivisen ja nopean web-sivun luomisessa. DOM tarjoaa tehokkaan ja nopean ohjelmointirajapinnan. Tästä huolimatta selaimen tekemä sivun päivittäminen, johon kuuluu CSS:n uudelleenkäsittely, elementtien sijoittelun ratkaiseminen ja web-sivun uudelleenpiirtäminen, on raskas operaatio selaimelle ja vaatii monimutkaisilla sivuilla paljon muistia päätelaitteelta.

Muokatessa DOM:ia joudutaan päivittämään näkyvä sivu kokonaan tai osittain. Tarkasteltaessa yhden pikselin esittämistä web-sivulla siihen vaikuttavat seuraavat tekijät: JavaScript, tyyli, ladonta, piirto ja kooste (katso kuva 2). Yleensä JavaScriptiä käytetään sivuston sisällön muuttamiseen. Tällainen muutos voi olla esimerkiksi listan uudelleenjärjestäminen tai elementin poistaminen näkyvistä. Tyylin uudelleenlaskeminen tarkoittaa CSS-dokumentin tyylisääntöjen joukosta sisältöön tiettynä hetkenä vaikuttavien sääntöjen selvittämistä. Tämän jälkeen jokaiselle yksittäiselle elementille määritetään lopul-

liset siihen vaikuttavat tyylisäännöt. Kun selain tietää mitkä tyylisäännöt vaikuttavat elementteihin, voidaan laskea kuinka paljon tilaa jokainen elementti käyttää ja mihin elementti sijoitetaan ruudulla. Elementit vaikuttavat toisiinsa ladottaessa niitä sivulle. Ylemmän tason elementti periyttää oletuksena maksimileveyden lapsielementeilleen. Toisaalta lapsielementin leveys voi vaikuttaa myös ylemmän tason elementtiin, joten elementtien kokojen ja sijaintien määrittely on selaimelle monimutkainen operaatio. Yhdenkin elementin muuttaminen voi vaikuttaa kaikkiin DOM-mallissa sitä edeltäviin, seuraaviin ja samassa tasossa oleviin elementteihin. Kun ladonta valmistuu, voidaan piirtää varsinaiset näkyvät pikselit. Tässä vaiheessa piirretään teksti, värit, kuvat, reunat ja varjot eli kaikki sivun näkymän koostavat visuaaliset yksityiskohdat. Tämä tehdään yleensä useissa kerroksissa, koska elementit voivat olla päällekkäin. Lopulta koostevaiheessa eri kerrokset piirretään oikeassa järjestyksessä näkymään, jotta elementit näkyvät ja jäävät piiloon yksikäsitteisellä tavalla. Jokainen vaihe mahdollistaa tehokkuuden pullonkaulan ja saattaa hidastaa sivua. Tätä prosessia, jossa dokumenttien pohjalta luodaan näkymä ruudulle, kutsutaan web-sivun renderöinniksi. (Lewis, 2017; Grigorik, 2017) Renderöintiin palataan vielä myöhemmin luvun 3 ensimmäisessä aliluvussa.

Kuva 2. *Pikselin ruudulle määrittelevä putki. Perustuen: (Lewis, 2017).*

Tarkasteltaessa hyvin yksinkertaista, mutta lähes kaikille laitteille yhteistä tehokkuuden mittaria, ruudun päivitysnopeutta, tehokkaan toimivuuden takaamisen haasteellisuus konkretisoituu. Suurin osa nykypäivän laitteista päivittää ruutua 60 kertaa sekunnissa *(fps, frames per second).* Jos sivulla näkyvä sisältö muuttuu, esimerkiksi sisällön päivittymisen takia, selain pyrkii vastaamaan laitteen ruudunpäivitysnopeuteen ja yrittää tarjota uuden kuvan 16 ms aikana (1 s / 60 = 0.016 s). Todellisuudessa selaimella on kuitenkin paljon muitakin operaatioita suoritettavana samaan aikaan. Jotta päivitysnopeus ei alene, uuden kuvan luominen pitää käytännössä suorittaa 10 ms aikana. Päivitysnopeuden laskiessa ja vaihdellessa sivu vaikuttaa hitaalta tai kokonaan toimintoihin reagoimattomalta. Tämä vaikuttaa suoraan kielteisesti käyttökokemukseen (Lewis, 2017; MDN, 2015; Dirty Performance Secrets of HTML5, 2015). Ruudunpäivitysnopeuteen vaikuttavat laitteeseen ja verkon toimivuuteen liittyvät seikat, mutta myös useat sellaiset tekijät, joihin web-kehittäjällä on vaikutusmahdollisuus.

Web-sovelluksissa raskain toiminto suorituskyvyn kannalta on DOM:in päivittäminen. Nopeuden takaamiseksi muutosaluetta voidaan pienentää eli voidaan päivittää vain osaa näkymästä. Tähän pyrkivät useat modernit JavaScript-sovelluskehykset ja -kirjastot, jotka ovat ottaneet käyttöön erilaisia tapoja hallinnoida DOM:in päivittämistä ja optimointia. Niiden avulla pyritään helpottamaan web-kehitystä abstrahoimalla pois tarve päivittää DOM:ia käsin. Virtual DOM luo virtuaalisen mallin DOM:ista ja muuttaa ainoastaan sivun sitä osaa, johon muutos kohdistuu. Shadow DOM sen sijaan koostuu useista pienemmistä DOM:eista, joista jokaisella on vain pieni vaikutusalue. Tällaisten DOM-optimointien merkitys korostuu sivuston monimutkaisuuden ja koon kasvaessa. Molemmat näistä tavoista optimoida DOM:ia ovat kiinnostavia ja tarjoavat erilaisen tavan kehittää web-sovelluksia.

Tässä tutkimuksessa selvitetään miten nämä kolme DOM:ia (tavallinen DOM, virtual DOM ja shadow DOM) eroavat toisistaan selainpään toteutuksessa ja mitä vaikutusta käytetyllä laitteella on toteutuksen toimintaan. Tämä tehdään tutkimalla ensin kirjallisuuden avulla eri DOM:ien eroja ja lopulta toteuttamalla testisovellukset, joilla testattiin toteutuksen vaikutusta sovelluksen nopeuteen. Sovellusten käyttöä testataan kolmella erilaisella laitteella, jotta nähdään laitteen vaikutus toteutuksen toimivuuteen. Erityisesti virtual DOM:in suorituskykyä ja sen suhdetta tavalliseen DOM:iin on tutkittu ja testattu paljon, mutta yhteyttä shadow DOM:in nopeuteen taas kovin vähän. Näiden toimintaa taas erillisillä laitteilla on tutkittu vielä vähemmän, joten tämä työ pyrkii tuomaan nämä kaikki seikat saman kontekstin alle.

Työn rakenne alkaa toisen luvun yleiskatsauksesta web-kehitykseen viimeisten vuosikymmenten aikana jatkuen nykytrendien kartoittamiseen. Luvussa luodaan myös pohja front-end-kehityksen ja erityisesti JavaScript-kirjastojen olemassa olon ja mahdollisuuksien ymmärtämiselle. Kolmannessa luvussa käydään läpi DOM:in, virtual DOM:in ja shadow DOM:in ominaisuudet ja niiden tarjoamat mahdollisuudet web-kehitykselle. Neljännessä luvussa kerrotaan tutkimusmenetelmästä ja varsinaisista toteutetuista sovelluksista. Viidennessä luvussa käsitellään työssä saavutettuja tuloksia. Lopulta kuudennessa luvussa luodaan katsaus saavutettuihin tuloksiin, pohditaan työn merkitystä ja käydään läpi jatkokehitysideoita.

2. MODERNI WEB-SUUNNITTELU

Tässä luvussa tarjotaan tarvittavat taustatiedot web-kehityksen historian ja haasteiden ymmärtämiseksi sekä luodaan pohja web-sovelluksien kehittämisen ymmärtämiselle. Nopeasti muuttuvana kenttänä web-suunnitteluun liittyy hyvin paljon käsitteitä ja sisältöä, mikä olisi hyvä tietää työn ymmärtämisen kannalta. Rajaus on tehty niin, että se tukee mahdollisimman hyvin JavaScript-sovelluskehyksillä tehtävää web-kehitystä.

Aliluvussa 2.1 käsitellään web-kehityksen historiaa, selainlaitteiden muutosta ja modernia web-kehitystä. Selainlaitteiden muutoksesta siirrytään luontevasti tekniikoihin, jotka auttavat vastaamaan tähän muutokseen eli responsiiviseen suunnitteluun. Aliluvussa 2.2 käsitellään web-sovelluksia ja esitellään perinteisimpiä malleja web-sovelluksen toteuttamiseen. Lopulta aliluku 2.3 keskittyy esittelemään ensin JavaScriptiä yleisellä tasolla ja paneutuu sitten erilaisten JavaScriptin front-end-kirjastojen esittelyyn ja erityisesti komponenttipohjaisenkehitysmallin käyttämisen hyötyihin.

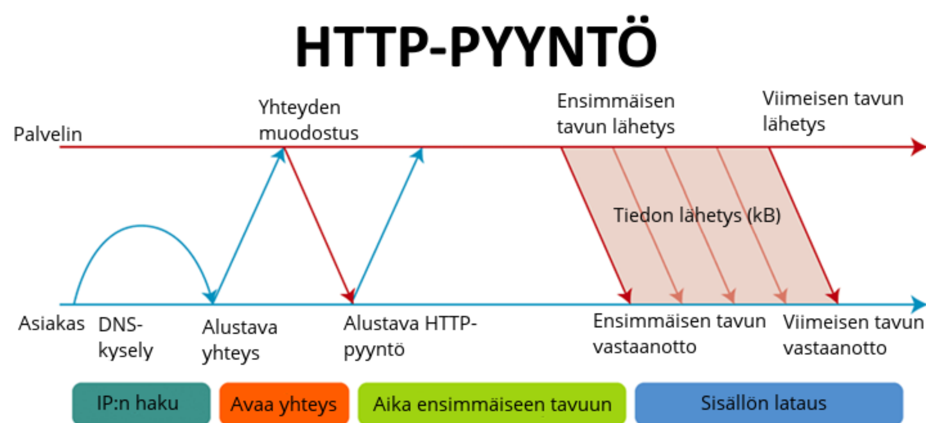
2.1 Web-kehitys

Web-sivut ovat lyhyen historiansa aikana olleet jatkuvan muutoksen kohteena ja alan suositellut käytännöt ovat muuttuneet useaan kertaan. Tähän on vaikuttanut niin tekniikan kehittyminen kuin erilaiset tarpeet web-sivuille ja selainlaitteiden muutos.

2.1.1 Hypertekstikokoelmista dynaamisiksi sovelluksiksi

Web-sivut koostuivat WWW:n (*World Wide Web*) alkuaikoina hypertekstidokumenteista, joiden välillä pystyi navigoimaan. Jo vuoden 1990 lokakuussa luotiin pohja modernille webille kolmen ydinteknologian avulla. HTML *(HyperText Markup Language)* vastaa web-sivun sisällön esittämisestä. URI *(Uniform Resource Identifier)* kertoo yksikäsitteisesti resurssin osoitteen internetissä. HTTP *(HyperText Transfer Protocol)* taas mahdollistaa linkitettyjen resurssien noutamisen verkosta (World Wide Web Foundation, 2017). Nopeasti pelkästä tiedon esittämisestä siirryttiin kohti dynaamisia web-sivuja, joissa käyttäjäkokemuksen ja vuorovaikutuksen merkitys kasvoi huomattavasti. Dynaamisten, käyttäjän toimintoihin reagoivien web-sivujen perustana toimii JavaScript-ohjelmointikieli, joka ensin mahdollisti asiakaspään *(client)* monimuotoisemmat ratkaisut ja voi nykyään toimia myös palvelinpuolen ohjelmointikielenä.

Web-kehityksen keskiössä toimii edelleen kommunikointi palvelimen kanssa. HTTP-protokolla on luonteeltaan pyyntö–vastaus-tyyppinen protokolla. Oikein muotoiltuun asiakkaan lähettämään pyyntöön palvelin vastaa pyydetyllä formaatilla. (Fielding, et al., 1999) Tällaista pyynnön lähettämistä esittää kuva 3. IP-osoitteen löytymisen jälkeen asiakas lähettää varsinaisen HTTP-pyynnön, johon palvelin vastaa lähettämällä asiakkaalle pyydetyn resurssin. Kuvasta nähdään myös, miten varsinainen latausaika muodostuu todellisuudessa IP-osoitteen hankkimisesta, yhteyden avaamisesta, vastauksen odottamisesta ja varsinaisen latauksen kestosta.



Kuva 3. *HTTP-pyynnön lähettäminen. Perustuen: (King, 2008).*

Aiemmin palvelin vastasi täysin tiedon käsittelystä ja sen muokkaamisesta esitettävään muotoon, mutta JavaScriptin mahdollistaman asiakaspään prosessoinnin avulla asiakkaan vastuu tiedon käsittelyssä ja muokkaamissa on kasvanut. 2000-luvun aikana myös web-sivujen sosiaalinen merkitys muuttunut. Erilaisten sosiaalisten medioiden käyttö on lisääntynyt ja käyttäjien itsensä luomaa sisältöä on saatavilla paljon enemmän kuin aiemmin. Tämä näkyy esimerkiksi erilaisten YouTuben video-blogien ja perinteisiä medioita yhdistelevien blogien suosiona. Tällaisen tietomäärän esittäminen ja sujuvan käyttäjäkokemuksen tarjoaminen suurille käyttäjämäärille ovat keskeisimpiä haasteita, joihin modernit JavaScript-kirjastot pyrkivät vastaamaan.

2.1.2 Selainlaitteiden muutos

Fielding (2014) kirjoittaa web-sivuihin kohdistuneista muutoksista 2000-luvun aikana. Vielä 90-luvulla web-sivujen ulkoasu muodostui taulukoista ja vaikka kaskadiset tyyliohjeet (*Cascading Style Sheets*) eli CSS julkaistiin 1996, CSS:n merkitys oli vähäinen ennen vuotta 2003. Tällöin julkaistiin sivusto CSS Zen Garden, joka esitteli CSS tehokkuutta ja sitä, miten sen avulla oli mahdollista täysin muuttaa sivun tyyliä koskematta lainkaan HTML-koodiin. Ennen tätä HTML vastasi niin sivuston sisällön kuin tyylin määrittelystä.

Tyylin ja sisällön jakaminen toisistaan riippumattomiin kokonaisuuksiin lisäsi koodin modulaarisuutta ja mahdollistaa koodin tehokkaamman uudelleenkäytön. Tämä voidaan nähdä SoC-ohjelmointiperiaatteen[1] mukaisen kehityksenä. CSS:n suosion myötä kehitysyhteisö käytännössä standardoi kohdenäytön kooksi 1024 pikseliä leveän ja 800 pikseliä korkean näytön. Tätä suuremmissa näytöissä sivuille lisättiin yleensä valkoiset reunat ja pienemmillä näytöillä vaadittiin sivun liikuttelua vierityspalkin avulla. Näin sivustot pystyttiin suunnittelemaan ja toteuttamaan mahdollisimman suurelle käyttäjämäärälle. (Fielding, 2014)

Mobiiliselaimet kehittyivät nopeasti ja vuonna 2007 julkaistu iPhone tarjosi ensimmäisenä modernin mobiiliselaimen. Yksinkertaistetut ja vaikeasti käytettävät puhelinten selaimet olivat pian menneisyyttä ja työpöytätason selain oli taskussa mukana kuljetettavassa muodossa. Yritysten ensimmäinen reaktio oli luoda erillinen mobiilioptimoitu versio web-sivusta, koska kohdistetun käyttäjäkokemuksen oletettiin lisäävän myyntiä. Useimmiten tällaiset sivut olivat kuitenkin yksinkertaistettuja versioita täysistä web-sivuista. Tämän takia niistä saattoi puuttua sisältö, jota sivulta etsittiin ja tämän löytäminen vaati kuitenkin alkuperäisen sivun käyttöä. (Fielding, 2014)

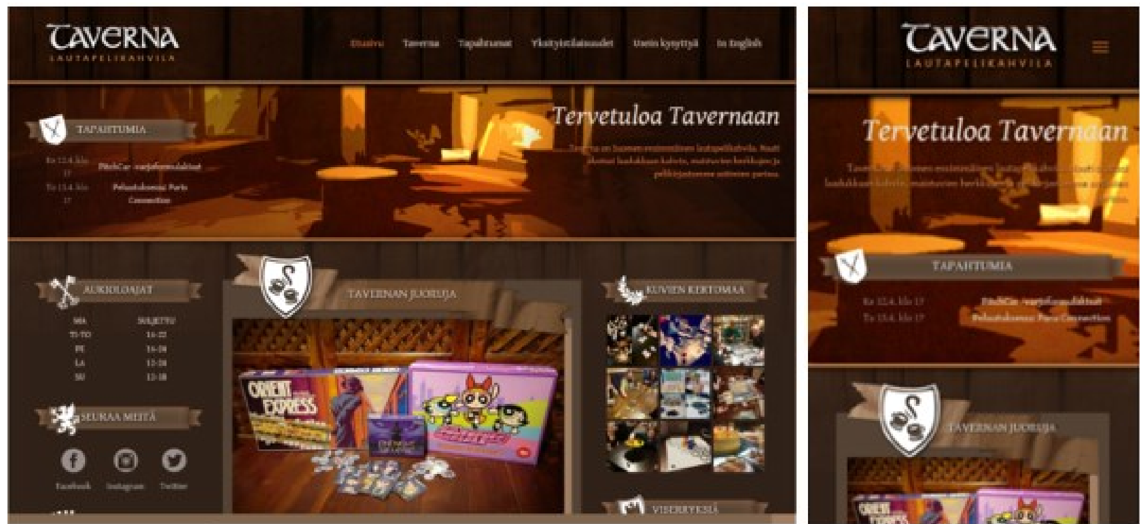
2.1.3 Responsiivinen web-suunnittelu

Fielding (2014) kuvaa responsiivista web-suunnittelua selaimen tapana muuttaa toimintaansa ympäristön perusteella. Responsiivinen web-suunnittelu mahdollistaa sen että selain, laite ja näyttökoko vaikuttavat käyttäjäkokemukseen mahdollisimman vähän. Responsiivisiksi sunnitellut web-sivut mukauttavat ulkoasuaan käyttämällä joustavaa ruudukointia (fluid grid), joustavaa sisältöä (esimerkiksi kuvat, videot ja teksti) ja CSS3[2]:n mediakyselyjä (media query).

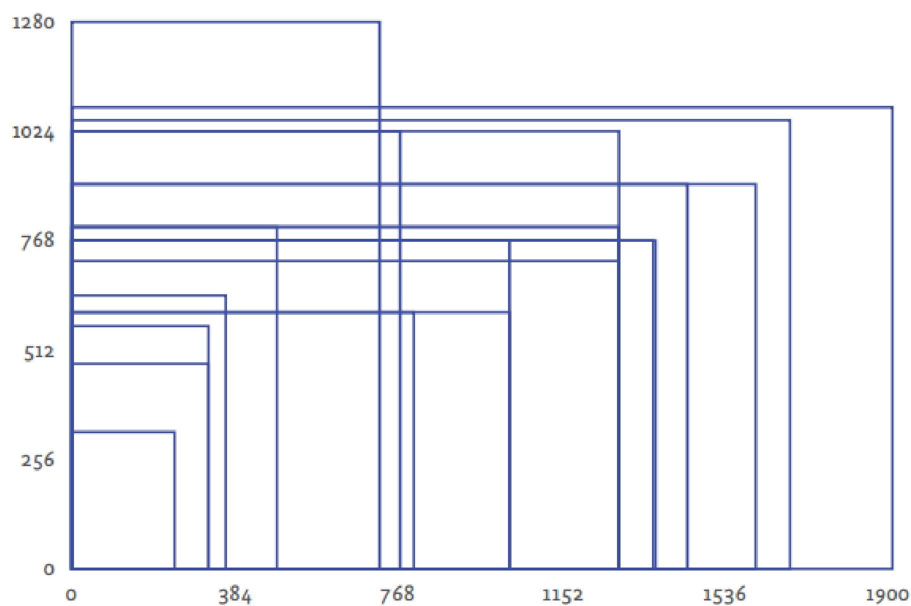
Responsiivisessa suunnittelussa vältetään absoluuttisia yksiköitä, kuten pikseleitä tai millimetrejä, ja suositaan suhteellisia yksiköitä, kuten prosentteja. Käytännössä tämä tarkoittaa sitä, että sivun mitat muuttuvat näkymän mittojen muuttuessa. Kuvassa 4 hahmottuu responsiivisen web-sivun toiminta käytännössä. Siinä esitetään samat elementit kuin varsinaisella sivulla, mutta niiden sijoittelu ja leveydet muuttuvat suhteessä käytetyn näytön kokoon. Kuvasta 5 nähdään vuonna 2013 tehdyn tutkimuksen perusteella suosituimmat näyttökoot. Muutosta nykypäivään tietysti on paljon, mutta olennaista on kuvasuhteiden ja näyttökokojen suuri vaihteluväli.

[1] Separation of Concerns eli SoC on yleinen ohjelmistokehityksen periaate, jossa erilaisen sovelluksen ominaisuudet jaetaan mahdollisimman pieniksi kokonaisuuksiksi. Näiden kokonaisuuksien vastuualueet ja riippuvuudet eriytetään toisistaan. (Microsoft, 2017)

[2] CSS3 on viimeisin standardoitu versio CSS:stä (Fielding 2014).



Kuva 4. *Lautapelikahvila.fi-verkkosivu toimii hyvänä esimerkkinä responsiivisesta verkkosivusta. Kuvassa sama näkymä sekä työpöytä- että mobiiliselaimella.*



Kuva 5. *Erilaisia resoluutioita 20 suosituimmassa laitteessa (visualisointi (Jehl, 2014)**)**, tiedot peräisin lähteestä OpenSignal, 2013).*

Ensimmäisiä responsiivisesta web-suunnitteluista kirjoittaneita oli Ethan Marcotte vuonna 2010. Marcotte vertaa rakennusarkkitehtuuria ja web-suunnittelua toisiinsa. Arkkitehtuuriset päätökset muovaavat fyysistä tilaa pitkäksi ajanjaksoksi ja asettavat rajoitteet tilan käytölle. Web-suunnitelua sen sijaan määrittää lyhytikäisyys ja pääte-

laitteiden moninaisuus. Maailmanlaajuisesti onkin jo pitkään näyttänyt siltä, että internetin käyttö mobiilialustoilla on ohittamassa suosiossa perinteisen internetin työpöytäkäytön ja tämän rajan ylittämisestä uutisoi ensimmäisenä StatsCounter vuoden 2016 marraskuussa.

Harva enää kiistää mobiililaitteille suunnittelemisen tärkeyttä, mutta responsiivinen suunnittelutapa jakaa myös mielipiteitä. Jakob Nielsen (2012) kirjoittaakin, että vaikka on edullista käyttää uudelleen sisältöä ja tyyliä erilaisissa laitteissa, niin responsiivinen suunnittelu kuitenkin samalla myös heikentää sivua. Hänen mielestään ylivertainen käyttäjäkokemus vaatii tiivistä integrointia alustan kanssa. Fielding (2014) kuitenkin toteaa että, yksinkertaisuus on responsiivisuuden avainetu. Mobiilitarjontaan ei tarvitse keskittyä, sillä samalle sivulle toimii yhteinen verkko-osoite ja sivusto jakaa yhteisen koodikannan. Yhteinen koodikanta yksinkertaistaa muutosten tekemistä ja vähentää testaukseen kuluvia resursseja. Myös sisällönhallinta yksinkertaistuu responsiivisen suunnittelun myötä, sillä lähtökohtaisesti sama sisältö esitetään erilaisilla laitteilla. Tämän lisäksi Googlen hakukone nostaa tuloksissaan responsiivisia sivuja korkeammalle sijalle, mikä myös motivoi responsiivisen sivun luomista erillisen mobiiliversion sijaan (Protalinski, 2016).

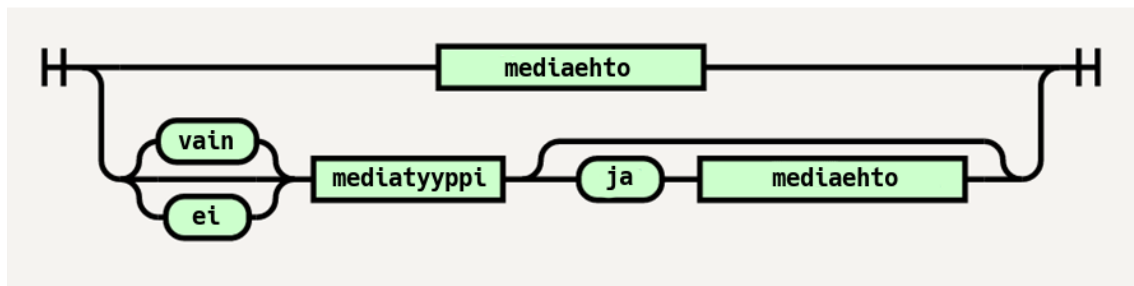
2.1.4 Responsiiviset tekniikat

Mediakysely (*media query*) esiteltiin ensimmäistä kertaa CSS3:n julkaisun yhteydessä. Vuonna 2010 julkaistiin ensimmäistä kertaa mediakyselyjä tukevat mobiiliselaimet. Mediakyselyt toivat mukanaan mahdollisuuden kohdistaa tyylejä ainoastaan tietyille näyttökoille. Samoihin aikoihin kun näyttöjen koko puhelimissa muuttui suuresti, kasvatettiin myös näyttöjen resoluutiota. Apple otti ensimmäisenä käyttöön termin retina-näyttö (*retina display*) kuvaamaan käyttämiään suuren pikselitiheyden näyttöjä ja tämän jälkeen korkearesoluutioiset puhelinten näytöt ovat yleistyneet. Pikselitiheydeltään suuria näyttöjä valmistetaan nykyään myös työpöytäympäristöihin. Tämän takia onkin tärkeää pystyä varmistamaan oman web-sivun näyttäminen oikealta tällaisillakin laitteilla, mikä onnistuu parhaiten hyödyntämällä responsiivisia tekniikoita.

Responsiiviset tekniikat antavat web-kehittäjälle mahdollisuuden hallita sivulla käytettäviä CSS-tyylejä esimerkiksi laitteen näytön korkeuden, leveyden, kuvasuhteen, resoluution ja orientaation perusteella.

```
@media screen and (max-width: 500px) {
	body {
		color: red;
	}
}
```

Yllä oleva yksinkertainen mediakysely määrittää sen, että mediatyypille `screen`, joka tarkoittaa selaimen ikkunaa, otetaan käyttöön mediaehto. Tässä tapauksessa määritetään, että jos ikkunan koko on alle 500 pikseliä leveä, niin `body`-elementin tekstin väri on punainen. Muista mediatyypeistä olennaisin on `print`, joka muuttaa tyyliä tulostettavalle versiolle sivusta. Maksimileveyden sijaan taas muita yleisiä määritteitä mediaehdolle ovat `min-width`, `orientation` ja `resolution`. (Myers, 2015)



Kuva 6. *Mediakyselyn ratkaiseminen (W3C, 2016a).*

W3C määrittelee mediakyselyn tapana testata tiettyjä ehtoja käyttäjäagentin tai dokumenttia esittävän laitteen ominaisuuksista. Kuvassa 6 esitellään miten mediakyselyn ehdon voimassa olo ratkaistaan. Mediakyselyt ovat lähes aina riippumattomia dokumentin sisällöstä, sen tyyleistä tai muista sisäisistä tekijöistä. Ne riippuvat ainoastaan ulkoisesta informaatiosta, paitsi jos jokin muu ominaisuus riippuu mediakyselyiden tuloksista, kuten `@viewport`-sääntö. (W3C, 2016a) `@viewport`-säännöllä viitataan siihen, että viewport-merkintää käyttämällä näkymän ominaisuudet voidaan CSS:n avulla määrittää halutuiksi.

```
@viewport {
    width: 320px auto;
}
```

Tällainen määrittely määrittää näkymän leveydeksi vähintään 320 pikseliä, mutta asettaa sen automaattisesti maksimileveydeksi, jos ikkunan leveys on tätä suurempi (W3C, 2016b).

Määrittelyssä korostetaan sitä, että käyttäjäagenttien tulee uudelleenarvioida mediakyselyitä, jos ne huomaavat muutoksia ympäristössä, kuten laitteen kääntäminen vaaka-asennosta pystyasentoon. Muutoksen huomaamisen jälkeen toimintaa vastaava mediakysely otetaan käyttöön (W3C, 2016a). Mediakyselyitä voidaan käyttää lisäämällä ne osaksi CSS-määrittelyä. Mediakyselyiden käyttöönotto vaatii myös `meta`-tagin käyttöä. Jos `meta`-tagia ei käytetä, mobiiliversio sivusta latautuu täsmälleen samanlaisena kuin työpöytäselaimellakin. Tällöin kerralla näytettävää sisältöä joutuu zoomauksen ja sivun vierittämisen avulla selailemaan (Myers, 2015).

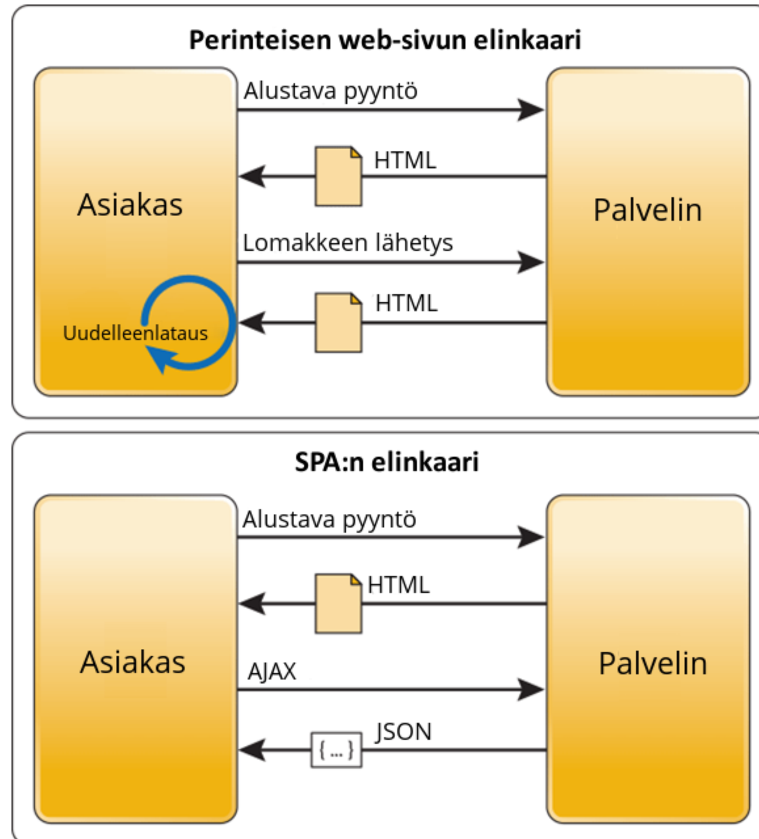
2.2 Web-sovelluskehitys

Web-sovellukset ovat sovelluksia, joita voidaan käyttää suoraan web-selaimessa ilman tarvetta asentaa mitään tietokoneelle. Web-sovellus voi olla esimerkiksi kompleksinen videopalvelun käyttöliittymä, kuten Netflix- tai Youtube-sovellus, tai vain yksinkertainen tehtävälista, kuten Google Keep. Raja dynaamisten web-sivujen ja varsinaisten web-sovellusten välillä on häilyvä, mutta hyvänä määrittelynä voidaan pitää samanlaista toiminnallisuutta mitä tietokoneelle tai mobiililaitteelle asennettavalta sovellukselta voidaan odottaa. Myös HTML5:n ominaisuuksien käyttäminen voi olla yksi kriteeri. HTML 5:n avulla voidaan esimerkiksi tallentaa tietoa paikallisesti selaimeen ja jatkaa toimintaa offline-tilassa (MDN, 2016a).

2.2.1 Yhden sivun sovellukset (SPA)

Yhden sivun sovelluksista (*SPA, single-page application*) puhutaan silloin, kun kaikki sovelluksen toiminnan kannalta tarvittava ladataan kerralla ja perinteisen linkkeihin perustuvan navigoinnin sijasta päivitetään vain yhtä muuttuvaa sivua. Tällaisen lähestymistavan hyödyt ovat selkeitä: sovellukset ovat joustavampia ja pystyvät vastaamaan nopeammin pyyntöihin, ilman tarvetta ladata ja uudelleenrenderöidä sivua jokaisen toiminnon jälkeen.

Kuvassa 7 esitetään sovelluksen tilamuutos asiakaspäässä sekä tavallisen web-sivun, että yhden sivun sovelluksen yhteydessä. Tavallinen web-sivu pyytää sisältöä ja saa HTML-tiedoston, jonka selain sitten renderöi selainikkunaan. Kun käyttäjä lähettää sivulla täyttämänsä lomakkeen, tehdään POST-pyynnön palvelimelle. Tähän pyyntöön palvelin vastaa lähettämällä päivittyneen HTML-tiedoston. Tämän jälkeen muuttunut HTML-tiedosto voidaan ladata näkymään päivittämällä sivu. Yhden sivun sovellus poikkeaa tästä niin, että se lähettää käyttäjän toimintojen perusteella palvelimelle AJAX-pyynnön. AJAX (*Asynchronous JavaScript and XML*) mahdollistaa pyyntöjen lähettämisen ja vastaanottaminen asynkronisesti. Tällaisen pyynnön lähettämisen jälkeen palvelin vastaa JSON-muotoisella datalla, josta selviää vastaus asiakkaan toiminnolle. JSON (*JavaScript Object Notation)* on tapa muotoilla tietoa yhdenmukaisesti, niin että samalla säilytetään sen luettavuus. (Wasson, 2013)



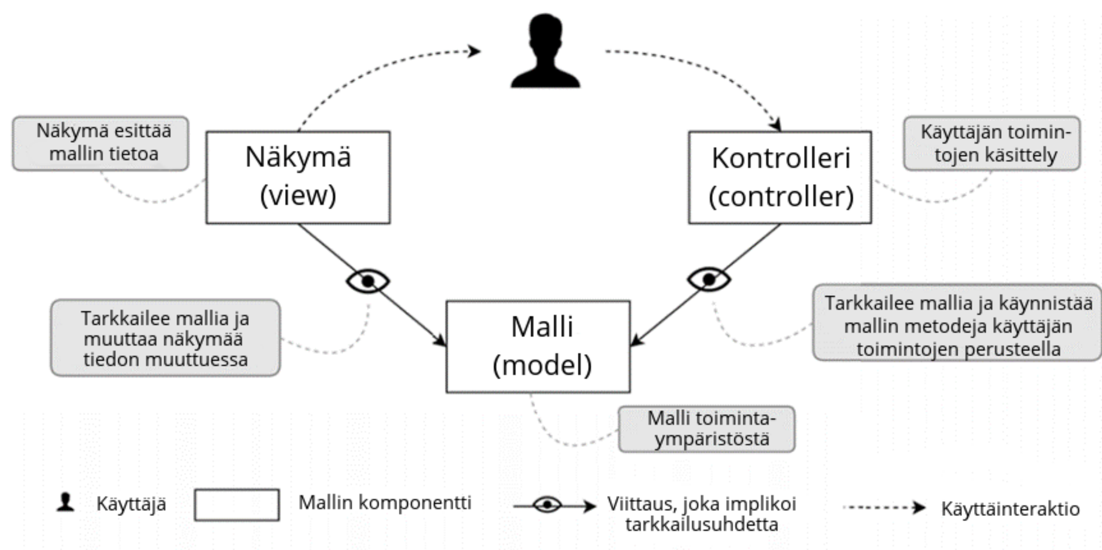
Kuva 7. *Tavallisen web-sivun ja yhden sivun sovelluksen kommunikointi palvelimen kanssa (Wasson, 2013).*

Puhtaassa yhden sivun sovelluksessa kaikki vuorovaikutus tapahtuu asiakaspäässä JavaScriptin ja CSS:n avulla. Sivun latauksen jälkeen palvelin toimii ainoastaan palvelurajapintana. Asiakassovelluksen tarvitsee vain tietää millaisia HTTP-pyyntöjä lähetetään ja minne. Palvelinpään toteutuksella siis ei ole vaikutusta asiakkaan toimintaan. Tällaisessa toteutuksessa kummatkin osapuolet ovat itsenäisiä ja voitaisiin tarvittaessa korvata toisella toteutuksella.

2.2.2 MVC-malli

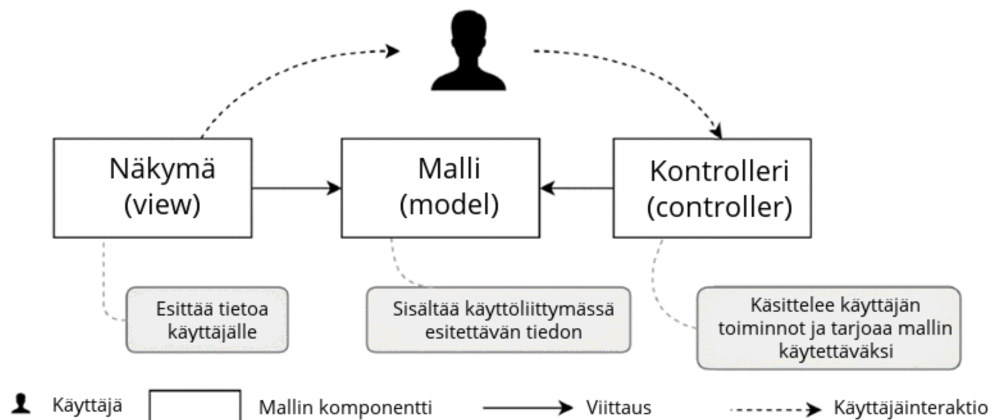
Erilaiset MV*-mallit (MVC, MVVM, MVP jne.) ovat ohjelmistoarkkitehtuurin malleja, joilla rakennetaan käyttöliittymiä tietojärjestelmille. Perinteisesti MVC-mallilla erotetaan kolme toisistaan toiminnallisuudeltaan poikkeavaa osaa, jotta voidaan eriyttää tiedon sisäinen esitystapa käyttäjälle esitetystä tiedosta. Nämä osat erotetaan toisistaan, jotta parantaa ohjelmakoodin uudelleenkäytettävyyttä ja komponenttien itsenäisyyttä. Monet JavaScript-ohjelmistokehykset ja kirjastot tarjoavat erilaisia toteutuksia ja versioita MV*-malleihin pohjautuen. Tässä esitellään niistä MVC-mallin mukainen rakenne, jotta ymmärretään esimerkiksi Angularin tapa hallita tiedon ja käyttöliittymän välistä yhteyttä.

MVC (*Model-View-Controller*) on eniten vaikuttanut muihin vastaaviin suunnittelumalleihin. Tämä lähestymistapa esiteltiin ensimmäistä kertaa 1980-luvulla. Lähtökohtaisesti MVC:tä käytetään monipuolisten työpöytäsovellusten suunnitteluun ja rakentamiseen. Mallia on jatkuvasti kehitetty ja sen pohjalta on luotu useita erilaisia malleja uusille teknologia-alustoille ja erilaisiin tarpeisiin. MVC-mallia käytetään edelleen käyttöliittymä- ja sovelluslogiikan yhdistämiseksi erilaisissa kohteissa, kuten mobiili- ja web-sovelluksissa. (Syromiatnikov & Weyns, 2014)



Kuva 8. *MVC-mallin "oppikirjaesimerkki" (Syromiatnikov & Weyns, 2014).*

Kuva 8 selkeyttää MVC-mallin ajatusta. Malli (*model*) on vastuussa tiedosta ja sen käsittelystä. MVC-mallissa malli ei ole yhteydessä muihin MVC-mallin komponentteihin eli tiedon esittäminen ja muokkaaminen eivät ole riippuvaisia tiedon tallennus- tai hallintatavasta. Näkymä (*view*) on vastuussa näkymän esittämisestä käyttäjälle ja lopulta kontrolleri (*controller*) vastaa käyttäjän toimintojen hallinnasta. Näkymä ja kontrolleri toimivat siis yhdessä ja vaikuttavat käyttäjän käyttöliittymässä. Jos käyttäjä muuttaa arvoa käyttöliittymässä, kontrolleri muuttaa arvoa mallin sisällä ja kun arvo on muuttunut se lähettää kaikille sitä seuraaville komponenteille tiedon tämän arvon muuttumisesta. Tällä tavoin näkymä saa lopulta tiedon arvon muuttumisesta ja voi muuttaa käyttöliittymän esitystä.



Kuva 9. *MVC-malli web-sovelluksissa (Syromiatnikov & Weyns, 2014).*

MVC malli sopii hyvin web-ympäristöön ja kuvassa 9 esitellään MVC-mallin käyttöä web-sovelluksessa. Web-ympäristössä näkymän ja kontrollerin vastuualueet erotetaan toisistaan luonnollisesti. Tieto esitetään HTML-sivujen avulla asiakaspäässä, kun taas muutokset tehdään tietoon palvelinpäässä. Oletetaan, että käyttäjä haluaa käydä web-sivulla etsimässä tietoa kirjoista ja menee web-selaimellaan tällaiseen palveluun. Palvelin vastaanottaa pyynnön ja käynnistää tarvittavan kontrollerin, joka liittyy kyseiseen osoitteeseen. Kontrolleri käynnistää toimintalogiikan kirjalistan saamiseksi ja luo mallin. Tämän jälkeen malli siirtyy näkymälle renderöitäväksi. Näkymä luo lopulta esityksen web-selaimelle. (Syromiatnikov & Weyns, 2014)

2.3 JavaScript

JavaScript esiteltiin ensimmäistä kertaa vuonna 1995 ja sen alkuperäinen tarkoitus oli tarkastaa käyttäjän syötteitä, joka aiemmin hoidettiin palvelimen puolella erilaisten palvelinkielten avulla. Ennen tätä jopa pelkän tyhjän kentän huomaaminen lomakkeesta vaati pyynnön lähettämisen palvelimelle ja vastauksen odottamisen. Netscape Navigator -selain pyrki ensimmäisenä helpottamaan syötevalidointia esittelemällä JavaScriptin. Tämän jälkeen JavaScript on laajentunut monen selaimen tärkeimmäksi ominaisuudeksi. Nykyään JavaScript voi vaikuttaa lähes kaikkeen, mitä selainikkunassa näkyy. (Zakas, 2012)

2.3.1 Erityispiirteet

JavaScript on korkean tason dynaaminen, tyypittämätön ja tulkattava ohjelmointikieli, joka toteuttaa sekä olio-ohjelmoinnin että funktionaalisen ohjelmoinnin paradigmaa. JavaScriptin perussyntaksi on peräisin Javasta, ensimmäisen luokan funktiot Schemestä ja

prototyyppipohjainen periyttäminen Selfistä. JavaScript on siis paljon muutakin kuin skriptauskieli, jollaisena sitä usein pidetään. Nykyisessä muodossaan se ei juurikaan toteuta skriptausparadigmaa, vaan on robusti ja yleiskäyttöinen ohjelmointikieli. (Flanagan, 2011)

2.3.2 Nimet ja versiot

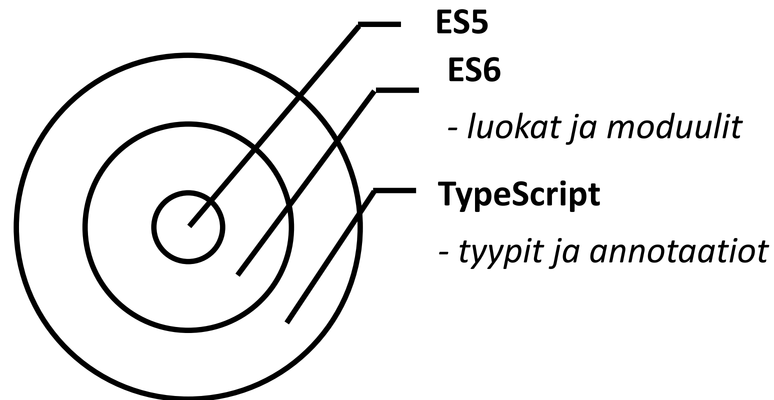
NetScapen julkaistessa JavaScriptin, se lisensöi termin JavaScript käytettäväksi omasta versiostaan kielestä. Nykyisin NetScape on muuttunut Mozillaksi. ECMA (European Computer Manufacturer’s Association) standardoi kielen ja rekisteröityyn tuotemerkkiin liittyvien ongelmien vuoksi standardoitu versio kielestä tunnetaan nimellä ECMAScript. Samoista rekisteröintisyistä johtuen Microsoftin versio kielestä on nimeltään JScript. Viime vuosikymmenestä alkaen kaikki selaimet ovat implementoineet version vähintään standardin ES3 eli kolmannen version ECMAScriptistä. Viimeisin laajemmassa käytössä oleva versio on ES5 ja viimeisimpänä julkaistu versio on jo seitsemäs ja se julkaistiin heinäkuussa 2016 (ECMA, 2016).

Osa uusimmista JavaScript-kirjastoista ja -sovelluskehyksistä käyttää TypeScriptiä. TypeScript ei ole uusi kieli vaan ES6-standardin päälle rakennettu laajennus. TypeSciptin suhdetta JavaScript-standardiversioihin esitellään kuvassa 10. TypeScript laajentaa ES5-standardin mukaista JavaScriptiä lisäämällä siihen esimerkiksi tyypit, luokat ja tuonnit eli importit. Varsinainen hyöty on kuitenkin tyypityksen tuominen JavaScriptiin. Se helpottaa koodin lukemista ja auttaa bugien paikantamista käännösaikana. TypeScript on Microsoftin ja Googlen yhteisen työn tulos. Tätä voi pitää osoituksena kielen merkittävyydestä, koska suuret ohjelmistotalot tukevat sen yleistymistä. Käytännössä TypeScriptin tuki selaimissa varmistetaan kääntämällä se ES5-koodiksi, jota lähes kaikki selaimet tukevat. (Fullstack.io, 2017)

Työssä näkyy myös JSX-muotoista koodia. Se on ES6:n laajennus, joka tuo lisänä JavaScriptiin mahdollisuuden kirjoittaa HTML-muotoisia JSXElementtejä eli esimerkiksi alla olevan mukainen `divElement`:in märittely suoraan JavaScriptissä on mahdollista.

```
var divElement = <div>Esimerkki</div>;
```

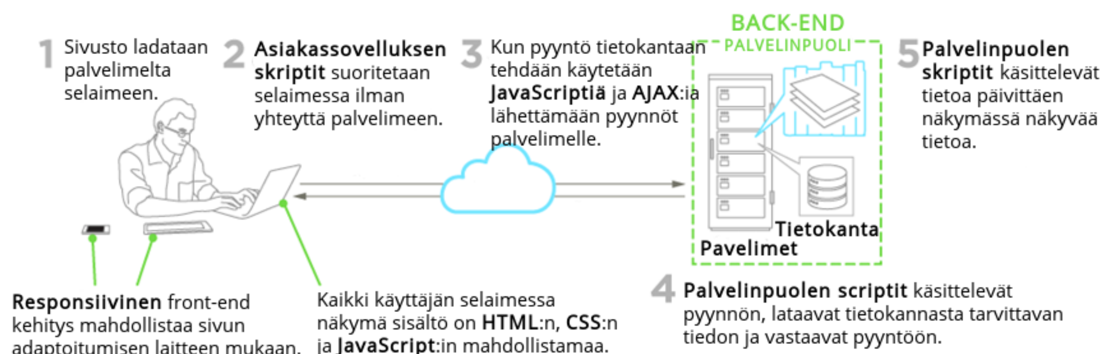
Tämä helpottaa esimerkiksi elementtien renderöintiä sekä käsittelyä ja mahdollistaa syntaktisesti tutun ilmaisun käyttämisen elementtien luomiseen (Facebook, 2014).



Kuva 10. *TypeScript laajentaa ES5- ja ES6-kielten määrittämää JavaScript standardia tyyppimäärittelyillä. Perustuen: (Fullstack.io, 2017).*

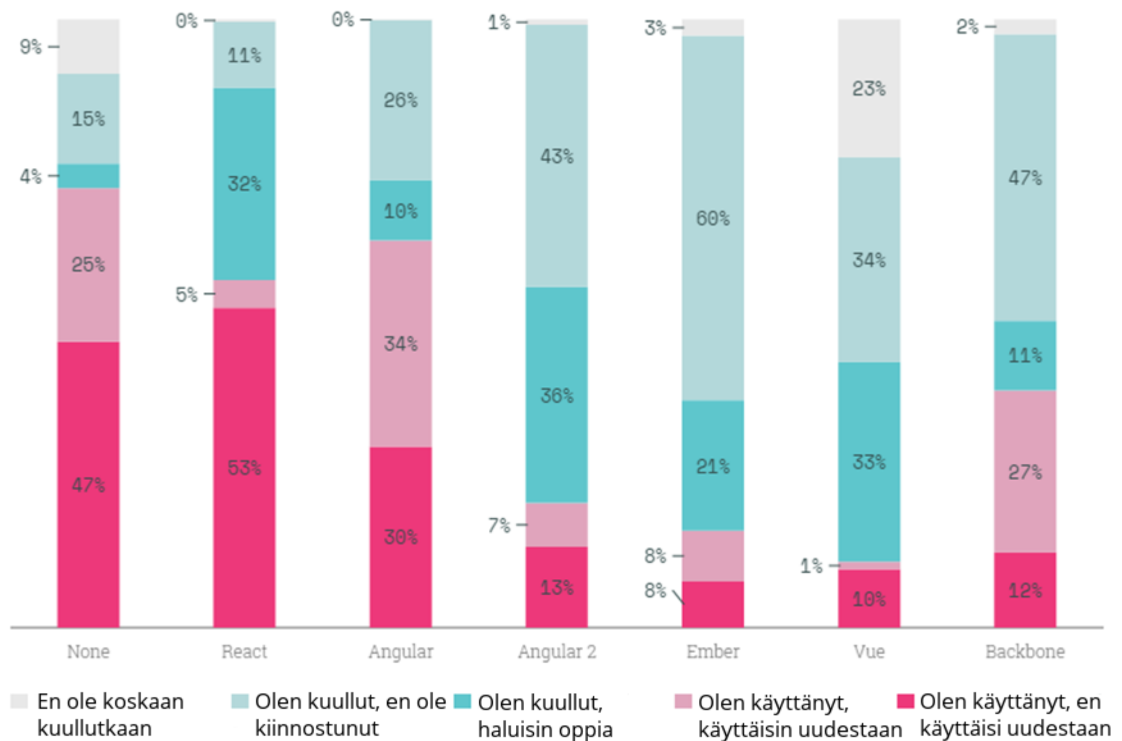
2.3.3 Front-end-kirjastot

Front-end on käytännössä termi selainsovellukselle. Tässä työssä front-end-termillä tarkoitetaan selaimen päässä olevaa toiminnallisuutta eli kuvan 11 kohtia 1 ja 2. Front-end-kehityksessä hyödynnetään useita erilaisia teknologioita, kuten aiemmin mainitut CSS, HTML, XML, JavaScript, JSON, URI, HTTP ja niin edelleen. Kuvassa 11 kuvataan front-end kehityksen osa-alueita. Kuvan 11 kohdat 1 ja 2 kuvaavat sivuston lataamista selaimeen ja asiakas-puolen scriptejä, joita ajetaan ilman palvelimelle eli back-endiin tehtäviä pyyntöjä. Kohdat 3–5 kuvaavat taas pyyntöihin vastaamista, tietokannasta tiedon hakemista hyödyntäen palvelintoteutusta.



Kuva 11. *Front-end ohjelmistokehitys. Perustuen: (Wodehouse, 2015).*

Asiakaspäässä JavaScriptiä käytetään usein jQueryn avulla. jQuery on JavaScript-kirjasto, jonka tarkoitus on helpottaa JavaScriptin käyttöä. JavaScript ei vaadi modulaarisen rakenteen käyttämistä tai juuri muutenkaan aseta rajoitteita koodin laadulle. Tämän seurauksena ohjelman koodista tulee helposti niin sanottua spagettikoodia ja näin ollen vaikeaa ylläpitää ja kehittää.



Kuva 12. *Erilaisten front-end-kirjastojen kiinnostavuus ja käyttöaste (Greif, 2016).m*

Erilaisia front-end-kirjastoja löytyy useita. Kuvassa 12 on listattu niistä muutamia, jotka ovat suosittuja tällä hetkellä. Kuvaan poimittuja kirjastoja ovat React, Angular, Angular 2, Ember, Vue ja Backbone. Vertailun vuoksi mukaan on otettu myös kirjaston käyttämättä jättäminen. Kuvassa pylväät kuvaavat suhtautumista ja käyttöastetta kyseiseen kirjastoon. Kuva on vuodelta 2016, joten Angular 2:en käyttäjien määrä on vähäinen suhteessa Reactiin ja Angular 1:seen. Nyt kun Angular 2:sesta on julkaistu release-versio, voidaan olettaa ainakin käyttöasteen lisääntyneen erilaisissa projekteissa. Kypsemmän teknologian käyttäminen on luonnollisesti pienempi teknologiaan liittyvä riski ohjelmistoprojektissa. On tärkeää huomata, että vaikka tässä työssä käytetään termiä front-end-kirjasto sekä Angularin eri versioista, että Reactista, niiden välillä on kuitenkin iso ero toiminnallisuuden ja vastuualueiden suhteen. Angularista olisi parempi käyttää termiä ohjelmistokehys (*framework*), kun taas React on vain front-end-kirjasto.

Tällaisista kirjastoista on hyötyä monella eri osa-alueella. Ne mahdollistavat uudelleenkäytettävien komponenttien kirjoittamisen yksinkertaisesti ja tiedon päivittämisen näkymään helpommin (*data-binding*). Ne myös tarjoavat reitityksen SPA-sovelluksissa. Yleisenä hyötynä voidaan pitää sekä modulaarisuuden että turvallisuuden parantumista käytettäessä avointa, vertaisarvioitua kirjastoa. (Shukla, 2014)

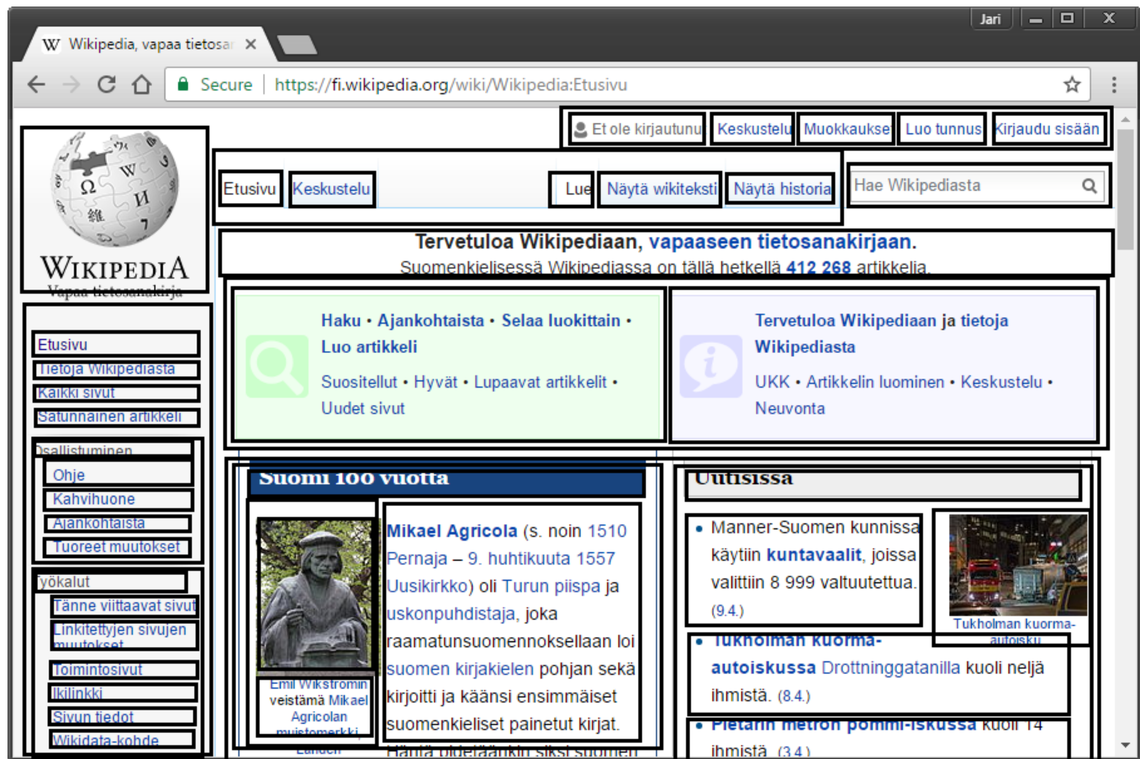
2.3.4 Komponentit

Komponentti on itsenäinen moduuli, joka koostuu sisäisestä tilasta ja toimintalogiikasta. Komponentilla on selkeä rajapinta, jonka avulla sitä voidaan hyödyntää. (Bailey, 2015) Tämä voi olla helposti sekoitettavissa olio-ohjelmoinnin olioon. Oliopohjainen ja komponenttipohjainen ohjelmistokehitys kuitenkin eroavat siinä, että oliopohjaisessa ohjelmistokehityksessä luodaan moduuleista suuri kokonaisuus ja komponenttipohjaisessa ohjelmistokehityksessä taas yksittäiset moduulit toimivat itsenäisesti. (Lowy, 2005) Komponenttien avulla voidaan luoda uudelleenkäytettäviä itsenäisiä moduuleja web-sovelluksiin.

Tässä työssä keskitytään vain käyttöliittymäkomponentteihin. Käytännössä minkä tahansa sivun sisällön voisi jakaa komponentteihin, mutta se ei välttämättä tuo yhtä suurta hyötyä kaikissa tapauksissa. Kuvassa 13 havainnollistetaan, miten esimerkiksi Wikipedia-sivuston sisältö voitaisiin jakaa itsenäisiin komponentteihin. Esimerkiksi vasemman sivun linkeistä koostuva palkki voisi olla `MasterNavigation`-komponentti, joka koostuu yksittäisistä `NavigationItem`-komponenteista. Tärkein hyöty komponenttien käytöstä on uudelleenkäytettävyys ja sisäisen toteutuksen riippumattomuus.

Front-end-kirjastot tarjoavat erilaisia tapoja toteuttaa komponentteja, mutta pohjimmiltaan ne eivät kuitenkaan suuresti eroa toisistaan. Komponenttipohjaiseen web-kehitykseen pyritään myös web-komponenttien (*Web Components*) avulla. Web-komponenteilla tarkoitetaan joukkoa web-selaimen rajapintoja, jotka mahdollistavat kustomoitujen, uudelleenkäytettävien ja kapseloitujen HTML-tagien luomisen web-sivuille ja web-sovelluksiin. Web-komponentit perustuvat olemassa oleviin standardeihin ja niiden tuki on lisääntymässä jatkuvasti. Nämä ovat erityisen mielenkiintoisia siksi, että yksi näistä tekniikoista on juuri shadow DOM, jota myöhemmin työssä käsitellään tarkemmin. Muita web-komponenttien tekniikoita ovat kustomoidut elementit (*Custom Elements*), HTML-tuonnit (*HTML Imports*) ja HTML-pohjat (*HTML Template*). Kustomoidut elementit mahdollistavat uusien DOM-elementtien luomisen ja suunnittelun, HTML-tuonnit taas mahdollistavat HTML:n uudelleenkäytön toisen HTML-dokumentin sisällä ja HTML-pohja mahdollistaa HTML-fragmenttien täyttämisen ja kopioimisen suorituksen aikana. Web-

komponentit ovat hyvin mielenkiintoinen kokonaisuus myöhemmin yleiseen käyttöön tulevia tekniikoita. Toistaiseksi niiden käyttö kaikissa eri selaimissa vaatii kuitenkin jonkun komponenttikirjaston tai Polyfillien[3] käyttöä ja vain harva selain tukee vielä natiiveja web-komponentteja. (WebComponents.org, 2017)



Kuva 13. Wikipedian etusivun mahdollinen karkea jako komponentteihin.

[3] Polyfill tarkoittaa JavaScript-kirjastoa, joka luo selaintuen puuttuessa automaattisesti rajapinnan halutun ominaisuuden käytölle.

3. DOKUMENTTIEN JÄSENTÄMINEN

Tässä luvussa käsitellään dokumenttien jäsentämistä. Luku 3 rakentuu niin, että ensin aliluvuissa 3.1 ja 3.2 käsitellään perinteistä DOM:ia ja käydään läpi sen ominaisuuksia ja syitä sen käytölle. Tämän jälkeen esitellään erilaisia tapoja optimoida DOM-esitystä. Aliluku 3.3 esittelee virtual DOM:ia ja aliluku 3.4 shadow DOM:ia. Perinteistä DOM:ia käsiteltäessä keskitytään DOM:in käyttökohteisiin, merkitykseen ja rakenteeseen. Virtual DOM:in yhteydessä pääpaino taas on DOM-rakenteen renderöinnissä ja shadow DOM:in yhteydessä keskitytään kapseloidun toteutuksen hyötyihin.

3.1 Dokumenttioliomalli (DOM)

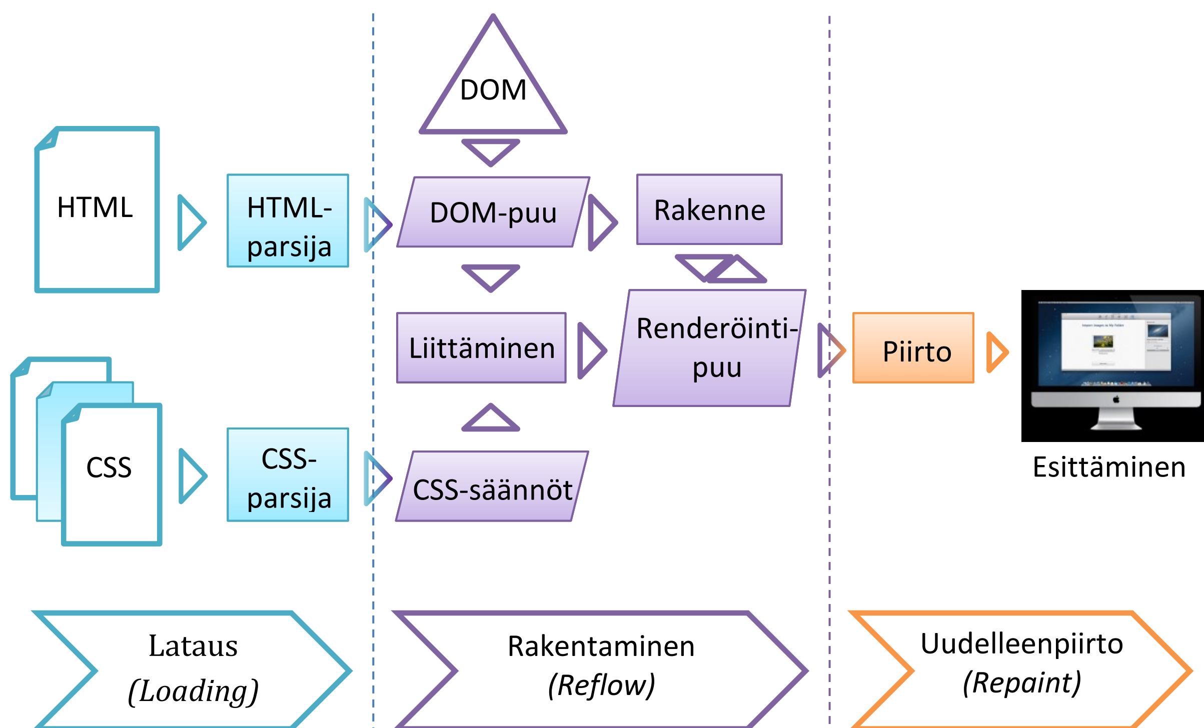
DOM eli dokumenttioliomalli (*document object model*) on ohjelmointirajapinta HTML- ja XML-dokumenteille. Se määrittelee dokumenttien loogisen rakenteen sekä tavan käsitellä ja manipuloida dokumenttia. Dokumenttioliomallin yhteydessä dokumentin käsite on laajempi kuin ”asiakirja” tai ”tiedosto”. XML-muotoa (*eXtensible Markup Language*) käytetään monimutkaisen tiedon tallentamiseen erilaisissa järjestelmissä. Dokumenttioliomallia voidaan kuitenkin käyttää XML-muotoisen tiedon hallintaan. (Robie, 1998; MDN, 2016b; W3C, 2005)

Dokumenttioliomallin tarkoitus oli toimia määritelmänä JavaScriptin käytölle web-selaimissa. Dynaaminen HTML oli suora edeltäjä dokumenttioliomallille ja se suunniteltiin vahvasti selainten ehdoilla. Kuitenkin kun dokumenttioliomallia suunniteltiin, mukana oli paljon toimijoita myös muista ympäristöistä, mukaan lukien HTML- ja XML-editorien ja dokumenttien tallentamisen parista. Useat näistä toimijoista olivat työskennelleet SGML:n[4] parissa ennen kuin XML oli kehitetty, minkä seurauksena dokumenttioliomalliin on vaikuttanut vahvasti SGML:n puurakenne. (Robie, 1998)

Kuvassa 14 esitellään, miten HTML- ja CSS-dokumenteista luodaan näytöllä näkyvä esitys web-sivusta. Dokumentit kulkevat ensin parsijoiden läpi ja tuottavat DOM-puun ja erilaiset CSS-säännöt. Nämä liitetään yhteen ja niistä luodaan renderöintipuu. Renderöintipuun perusteella päivitetään kuvaus sivun asettelusta eli rakenteesta sekä määritetään piirto-operaatiot. Lopulta varsinaisten piirto-operaatioiden perusteella luodaan esitys sivusta selainikkunaan. Selaimella on siis muistissaan kuvaus DOM-puusta, jota voidaan

[4] SGML (Standard Generalized Markup Language): HTML-merkkauksen edeltäjä ennen HTML5:n julkaisua (W3C, 2015)

muokata DOM-rajapinnan avulla. Eri operaatiot on ryhmitelty latausoperaatioon (*loading*), rakentamisoperaatioon (*reflow*) ja uudelleenpiirto-operaatioon (*repaint*).



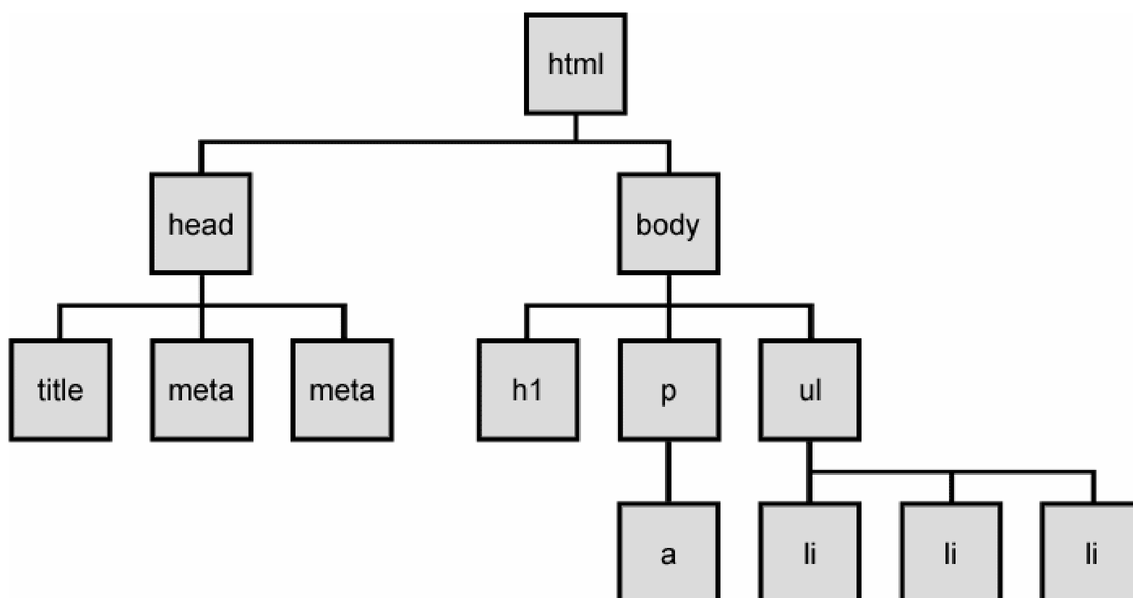
Kuva 14. *Selain rakentaa HTML- ja CSS-dokumenttien pohjalta DOM-puun, jonka käsitellylle DOM toimii rajapintana. Perustuen: (Mobidev, 2014).*

Dokumenttioliomallin avulla ohjelmoijat voivat luoda ja rakentaa dokumentteja, navigoida niiden rakenteessa ja lisätä, muokata ja poistaa niiden elementtejä ja sisältöä. Mitä tahansa HTML- tai XML-dokumentista löytyvää voidaan käsitellä edellä mainituin tavoin dokumenttioliomallin avulla, muutamaa poikkeusta lukuun ottamatta. Dokumenttioliomallin määritelmä on luotu niin, että se tarjoaa standardisoidun ohjelmointirajapinnan, jota voidaan käyttää erilaisissa ympäristöissä ja sovelluksissa. (Robie, 1998)

3.2 Millainen DOM on?

Dokumenttioliomalli on ohjelmointirajapinta ja sen malli itsessään muistuttaa rakennetta, jota se mallintaa. Dokumenttioliomallin dokumentit muodostavat loogisen, puumaisen rakenteen. Termi metsä kuitenkin kuvaisi rakennetta yleensä paremmin, koska malli koostuu yleensä useammista puista. Kuvassa 15 visualisoidaan dokumenttioliomallin puumaista rakennetta. Dokumenttioliomalli määritelmä ei kuitenkaan vaadi puumaista toteutusta. Olennaista on kuitenkin, että dokumenttioliomalli rakenne on rakenteellinen

isomorfia (*structural isomorphism*). Tämä tarkoittaa sitä, että jos dokumenttiobjektimallin toteutus luodaan esityksenä samalle dokumentille useita kertoja, on tuloksena aina sama rakenteellinen malli täysin samoilla olioilla ja niiden suhteilla. (Robie, 1998)



Kuva 15. *Dokumenttioliomallin puumainen rakenne (Stepp, et al., 2012).*

Nimitys dokumenttioliomalli on valittu, koska se on oliomalli sanan suorassa merkityksessä. Dokumentteja mallinnetaan olioilla ja malli ei ainoastaan koostu dokumentin rakenteesta, vaan myös siihen kuuluvan dokumentin ja olioiden käyttäytymisestä, suhteista ja yhteyksistä. SGML-dokumenttien rakennetta on perinteisesti mallinnettu abstraktilla tietorakenteella, eikä objektimallilla. Abstraktissa tietorakenteessa malli keskittyy dataan. Olio-orientoituneissa ohjelmointikielissä data suljetaan olioihin, jotka suojaavat dataa ulkoilta muutoksilta. Näiden olioiden tarjoamat metodit taas määrittävät, miten tietoa voidaan muuttaa, ja ovatkin osa oliomallia. (Robie, 1998)

DOM määrittelee rajapinnat, joita voidaan käyttää XML- ja HTML-dokumenttien hallintaan. Nämä rajapinnat ovat abstraktioita, ainoastaan määritelmiä sovelluksen sisäisen dokumenttimallin käyttämiselle ja käsittelylle. Käytännössä rajapinnat eivät oleta mitään tiettyä toteutusta. Jokainen DOM-sovellus voi säilyttää dokumentteja millä tahansa sopivalla tavalla, kunhan se tukee DOM:in määritelmän mukaisia rajapintoja. On myös hyvä huomata, että DOM:in määritelmä määrittää rajapinnat, eikä sitä mitä objekteja luodaan. DOM ei voi siis tietää mitä rakentajia kutsutaan DOM:in käyttäjän luodessa DOM-rakenteen. Luotaessa rakennetta DOM:in implementaatiot luovat oman sisäisen toteutuksen näille rakenteille. (Robie, 1998)

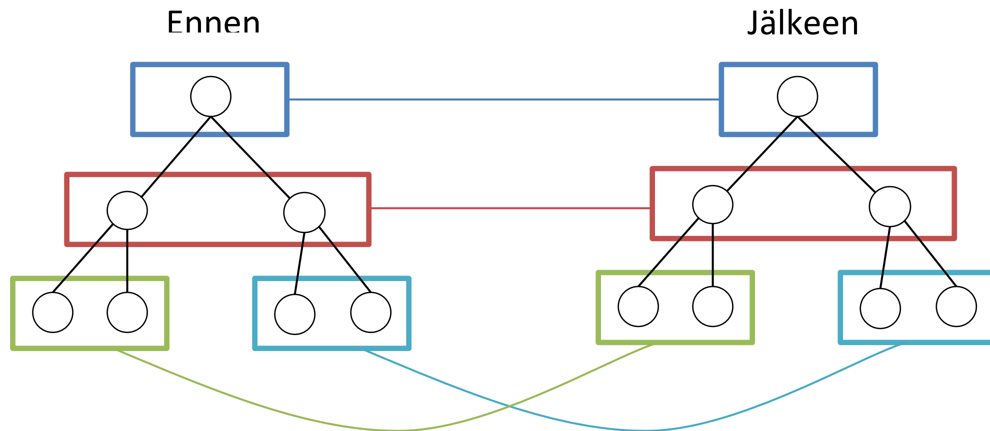
3.3 Virtuaalinen DOM (V-DOM)

Web-sovelluksissa raskain toiminto suorituskyvyn kannalta on DOM:in muuttaminen. Reactissa tämä ongelma yritetään ratkaista luomalla virtualisoitu DOM-esitys, josta käytetään termiä virtual DOM (*VDOM*, *V-DOM*). React pitää selaimen muistissa kuvausta virtual DOM:ista. Muutosalgoritmin avulla voidaan laskea muutos oikeaan DOM:iin nähden ja päivittää lopulta vain sitä osaa DOM:ista, mikä todellisuudessa on muuttunut. Varsinainen näkymään kohdistuva muutos on siis pienempi, minkä tulisi suoraan nopeuttaa sovelluksen toimintaa. Tästä on erityisesti hyötyä sovelluksen monimutkaisuuden kasvaessa. (Akshat & Nalwaya, 2016)

Useat frameworkit kuten Maquette ja Incremental DOM toteutettavat virtual DOM:in, mutta React on yksi helpoimpia tapoja saada virtuaalisen DOM:in hyödyt käyttöön. Tämän takia työssä tarkastellaan Reactin toteuttamaa versiota virtual DOM:ista. Reactin avulla kuvataan miltä käyttöliittymän halutaan näyttävän ja renderöinnin tuloksena on JavaScript-objekteista koostuva kuvaus todellisen DOM:in sijaan. (Chedeau, 2013; Fedosejev, 2015) React-sovellukset kirjoitetaan JavaScriptillä, jolla päivitetään React-komponentteja ja React taas päivittää DOM:ia. Tällöin sovelluskehittäjän ei tarvitse itse huolehtia vuorovaikutuksesta DOM:in kanssa. Virtuaalinen DOM on puumuotoinen listaus elementeistä, näiden attribuuteista ja sisällöstä samaan tapaan kuin perinteinenkin DOM. (Minnick, 2016)

3.3.1 DOM:in päivittäminen

DOM:in päivittäminen suoraan on haastavaa ja edellisestä DOM:in tilasta selvillä pysyminen on hyvin hankalaa. React ratkaisee ongelman pitämällä kahta kopiota virtuaalisesta DOM:ista. Muutoksen tapahtuessa muutosalgoritmin avulla tarkistetaan, mitä olivat todelliset muutokset ja palautetaan tieto tarvittavista DOM:in muutosoperaatioista. Näiden DOM-operaatioiden perusteella tehdään varsinaiset muutokset selaimen DOM:iin, mikä johtaa näkymän päivittymiseen. (Akshat & Nalwaya, 2016)



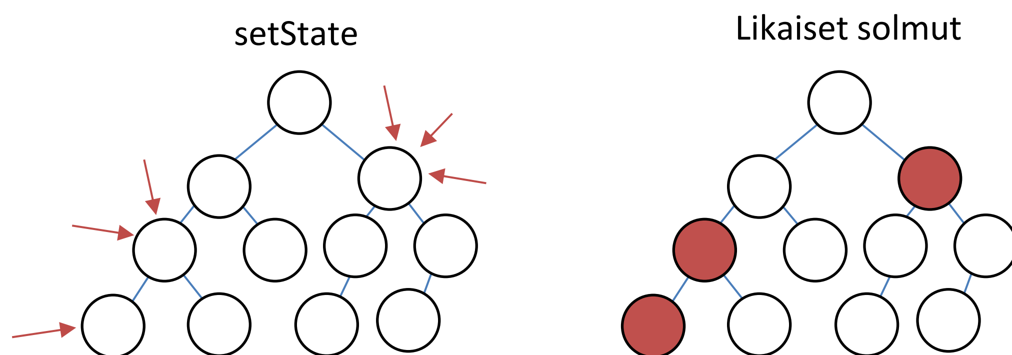
Kuva 16. *Reactin DOM-puun tasoihin perustuva vertailu. Perustuen: (Chedeau, 2013).*

Mahdollisimman pienen muutosten lukumäärän löytäminen kahden satunnaisen puun väliltä on $O(n^3)$-ongelma (Bille, 2007). Tällainen operaatio on raskas käyttöliittymän tarpeisiin nähden. React kuitenkin optimoi operaation $O(n)$-nopeuteen käyttämällä tehokkaita heuristisia menetelmiä hyvän approksimaation löytämiseen. Käytännössä React tekee tämän testaamalla puiden tasoja yksi kerrallaan, kuten kuvassa 16 havainnollistetaan. Operaation kompleksisuus vähenee tällöin huomattavasti ja erojen löytäminen helpottuu. Tällainen vertailu toimii, koska web-sovelluksissa komponentteja harvoin siirretään toiselle tasolle puussa, vaan ne liikkuvat yleensä samassa tasossa muiden lapsisolmujen keskuudessa. (Chedeau, 2013)

3.3.2 Renderöinti

Komponentti vastaa sisältämästään tiedosta ja sille voidaan luoda rakentavassa tila-objekti. Komponentin tilaa voidaan muokata tämän jälkeen käyttäen `setState`-metodia, joka päivittää tai asettaa tila-objektin arvoja. Kutsuttaessa `setState`-metodia yksittäiselle komponentille, React merkkaa komponentin likaiseksi (*dirty*) eli muuttuneeksi.

Kuvassa 17 esitellään miten `setState`-metodin avulla muuttuneet solmut merkataan likaisiksi. Jokaisen tapahtuman jälkeen React tarkistaa kaikki likaiseksi merkityt komponentit ja renderöi ne uudelleen tarvittaessa. Tällainen toimintamalli johtaa siihen, että tapahtuman jälkeen on selkeä hetki, jolloin DOM:ia päivitetään. Tämä on avainominaisuus suorituskykyisen sovelluksen rakentamiselle ja kuitenkin hyvin vaikeasti saavutettava tavoite käyttämällä puhdasta JavaScript-toteutusta. (Chedeau, 2013)



Kuva 17. *Likaisten solmujen merkitseminen ja niiden renderöinti. Perustuen: (Chedeau, 2013).*

Listamuotoisen sisällön renderöinti

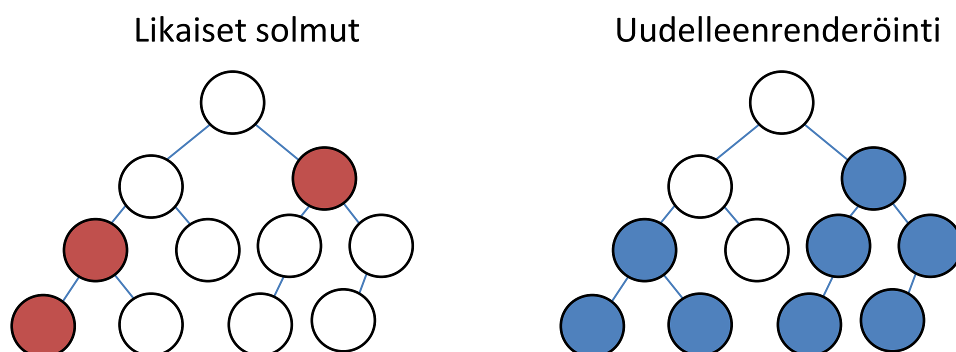
Oletetaan tilanne, jossa komponentti renderöi viisi komponenttia (esimerkiksi tehtävälistan tehtävät). Kun näkymää päivitetään, lisätään uusi komponentti olemassa olevien väliin. Tämän tiedon avulla on vaikea tietää miten nämä kaksi komponenttilistaa rinnastuvat toisiinsa. Oletuksena React assosioi alkuperäisen listan ensimmäisen komponentin päivitetyn listan ensimmäisen komponentin kanssa ja niin edelleen. Reactille on myös mahdollista määrittää avain-arvo komponentille assosioinnin helpottamiseksi. (Chedeau, 2013)

Komponenttien renderöinti

React-sovellus yleensä koostuu useista käyttäjän määrittämistä komponenteista, jotka muodostavat useista elementeistä koostuvan puurakenteen. Tätä lisätietoa rakenteesta hyödynnetään muutosalgoritmissa, sillä React yrittää verrata vain keskenään saman tyyppisiä komponentteja. Jos DOM:issa esimerkiksi `<header>`-elementti korvataan `<div>`-elementillä, React yksinkertaisesti poistaa otsikon ja luo uuden <div>-elementin. Aikaa ei tuhlata kahden todennäköisesti hyvin erilaisen komponentin vertailuun. (Chedeau, 2013)

Alipuurenderöinti

Kutsuttaessa `setState`-metodia, komponentti uudelleenrakentaa lastensa virtual DOM:in. Jos `setState`-metodia kutsutaan juurisolmulle, koko React-sovellus uudelleenrenderöidään. Renderöitävien solmujen valintaa havainnollistetaan kuvassa 18.



Kuva 18: *Alipuurenderöinti likaisten solmujen perusteella. Perustuen: (Chedeau, 2013)*

Tällöin kaikkien komponenttien, jopa niiden jotka eivät muuttuneet, `render`-metodia kutsutaan. Tämä kuulostaa raskaalta ja epätehokkaalta operaatiolta, mutta käytännössä se toimii tehokkaasti, sillä todelliseen DOM:iin ei kosketa. Ensimmäisenä on tärkeää huomata, että kyse on näkymän luomisesta käyttöliittymään. Käyttäjän näyttötila on rajoittunut, joten kaikkea ei tarvitse näyttää kerralla. Toinen tärkeä huomio on, että kirjoitettaessa React-koodia, harvoin on tarvetta kutsuta `setState`-metodia juurisolmulle. Yleensä sitä kutsutaan komponentille, joka vastaanottaa muutostapahtuman tai sen isäntäkomponentille. Käytännössä muutokset tapahtuvat vain alueella, jolla käyttäjä vuorovaikuttaa. (Chedeau, 2013)

Selektiivinen alipuurenderöinti

Kehittäjälle tarjotaan myös mahdollisuus estää haluttujen alipuiden renderöinti tarvittaessa. Jos komponentti toteuttaa metodin

```
boolean shouldComponentUpdate(object nextProps, object nextState)
```

voidaan seuraavaan ja edelliseen tilaan perustuen päätellä onko komponentti muuttunut ja välittää tieto Reactille. Oikein hyödynnettynä, tämän avulla on mahdollista saavuttaa suuri hyöty tehokkuudessa. Jotta metodia voidaan käyttää, pitää pystyä vertailemaan kahta eri JavaScript-objektia. Tähän liittyy useita ongelmia, kuten se halutaanko vertailun olevan pinnallisella vai syvällä tasolla. Jos kyse on syvätason vertailusta, pitää käyttää muuttumattomia (*immutable*) tietorakenteita tai tehdä syvätason kopioita objekteista. Tätä metodia voidaan joutua kutsumaan jokaisella käyttöliittymäsilmukan iteraatiolla, joten tämä tulee huomioida metodin toteutuksessa. Lähtökohtaisesti sen pitäisi siis pystyä suoriutumaan vertailuoperaatiosta nopeammin, kuin mitä heuristisella muutosalgoritmilla kestää.

3.3.3 Tapahtumien hallinta

Tapahtumankuuntelijoiden (*event listener*) liittäminen DOM:iin on erittäin hidasta ja muistia kuluttavaa. Tämän sijaan React käyttää suosittua tekniikkaa, jotka kutsutaan tapahtumadelegaatioksi (*event delegation*). React vie tapahtumadelegaatiota vielä pidemmälle ja implementoi W3C-määritelmän kanssa yhteensopivan tapahtumamallin. Tämä tarkoittaa sitä, että Internet Explorer 8 -selaimen tapahtumien käsittelyongelmat on korjattu ja tapahtumien nimet ovat yhtenäiset eri selaimilla. Käytännössä tapahtumamalli on toteutettu liittämällä dokumentin juureen yksi tapahtumankuuntelija. Kun tapahtuma laukaistaan, selaimelta saadaan tietoon tapahtuman kohde. Kuitenkaan kuljetettaessa tapahtumaa DOM-hierarkian läpi, Reactissa ei käydä läpi virtuaalista DOM-puuta, vaan käytetään hyväksi React-komponenttien uniikkeja nimiä. Suorituskyvyn kannalta onkin tehokkaampaa siirtää kaikki tapahtumankuuntelijat yhteen rakenteeseen sen sijaan, että tapahtumankuuntelijat liitettäisiin osaksi virtuaalisen DOM:in solmuja. Selain luo uuden tapahtumaobjektin jokaiselle tapahtumalle ja tapahtumankuuntelijalla. Tämän hyvä puoli on, että viitteet näihin objekteihin voidaan säästää ja niitä voidaan myös tarvittaessa muuttaa. Tämä kuitenkin voi johtaa suureen muistin kulutukseen ja tämä ongelma on Reactissa korjattu käyttämällä objektiallasta (*object pool*). Aina kun tapahtumaa tarvitaan, se käytetään uudelleen altaasta. (Chedeau, 2013)

3.4 Shadow DOM

Shadow DOM on suunniteltu komponenttipohjaisten sovellusten rakennusvälineeksi. Se tarjoaa ratkaisuja useisiin ongelmiin web-kehityksessä. Shadow DOM:in avulla saavutetaan eristetty DOM (*isolated DOM*), jossa komponentin DOM on eriytetty varsinaisesta dokumentin DOM:ista. Sen avulla rakennetaan DOM-puita toisten puiden sisään luoden lopulta yksittäinen, hierarkkinen rakenne. Esimerkiksi valittaessa luokka-attribuutin avulla elementtejä voidaan etsiä elementtiä lokaalista puusta globaalin haun sijaan (Motto, 2016).

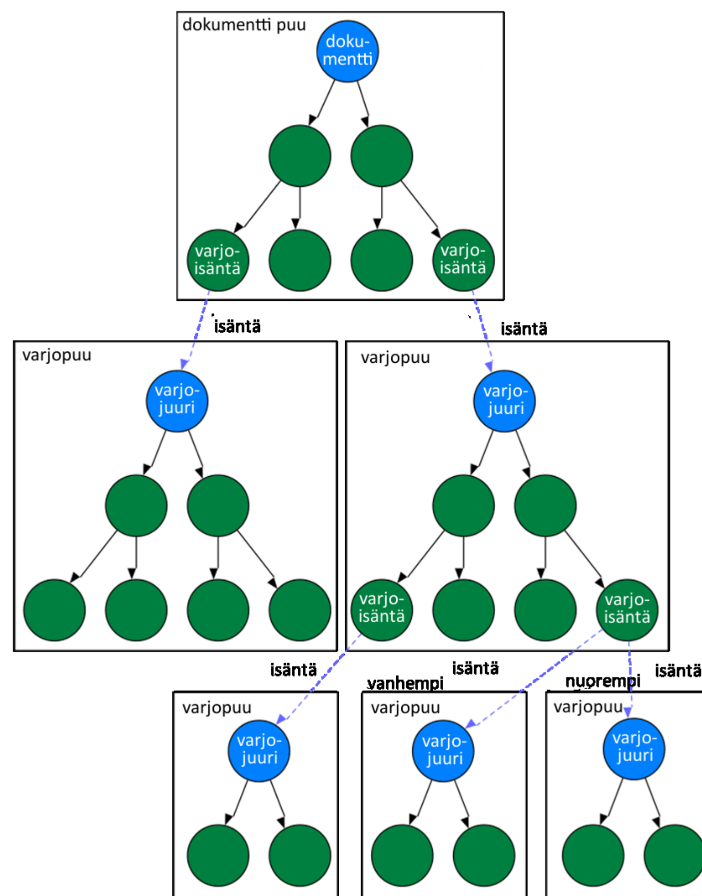
Shadow DOM on yksi web-komponenttien tekniikoista. Shadow DOM:in hyödyntäminen ei vaadi web-komponenttien käyttöä, mutta hyödyt saattavat tällöin olla suuremmat. Komponentin tyylin vaikutusalue on rajattu, jolloin shadow DOM:in sisällä määritellyt tyylit eivät vuoda komponentit ulkopuolelle ja ulkopuoleiset tyylit eivät vuoda sen sisään. Tällöin voidaan käyttää yksinkertaisempia CSS-sääntöjä ja geneerisempiä nimiä sekä välttyä nimeämiskonflikteilta. (Bidelman, 2016)

Toisin kuin virtual DOM:ia, shadow DOM:ia hyödyntäviä sovelluskehyksiä ja kirjastoja on vielä vähän. Näistä suosituin on kuitenkin Angular 2 ja se valittiin tähän työhön shadow DOM:in tutkittavaksi implementaatioksi.

3.4.1 Varjokapselointi

Shadow DOM eroaa kahdella tavalla perinteisestä DOM:ista: sen luominen sekä käyttökohde ovat erilaiset ja se käyttäytyy muusta sivusta poiketen. Normaalisti luodut DOM-solmut lisätään toisten elementtien lapsiksi. Shadow DOM:in kanssa luodaan vaikutusalueeltaan rajattu DOM-puu, joka on yhdistetty elementtiin, mutta on erillään elementin oikeista lapsista. (Bidelman, 2016)

Tällaista dokumenttifragmenttia kutsutaan varjopuuksi (*shadow tree*) ja se voidaan sitoa mihin tahansa DOM:in elementtiin. Tätä varjopuihin liittämistä kuvataan kuvassa 19. Varjopuun juuressa on solmu, jota kutsutaan varjojuureksi (*shadow root)*. Elementillä voi olla rajoittamaton määrä varjopuita, jotka järjestetään luomisen yhteydessä. Viimeisimpänä luotu varjopuu on elementin aktiivinen varjopuu. Elementtiä, johon varjopuu on liitetty, kutsutaan varjoisännäksi (*shadow host*) ja luonnollisesti tämä elementti on varjopuiden isäntäelementti (*host element*). (Atkins & Etemad, 2014)



Kuva 19. *Dokumentti voi koostua useista toisistaan eristetyistä varjopuista. Perustuen: (Dodson, 2013).*

3.4.2 Koostaminen

Koostamisen (*composition*) mahdollistaminen on yksi tärkeimpiä shadow DOM:in etuja. Koostamisen avulla web-kehityksessä erilaiset vakioelementit (kuten `<div>`, `<header>`, `<input>`) voivat muodostaa komponentteja. Natiivielementit ovat monikäyttöisiä juuri koostamisen takia. Kaikki natiivielementit (kuten `<select>`, `<details>` ja `<video>`) hyväksyvät tiettyjä elementtejä lapsikseen ja muuttavat toimintaansa niiden mukaan. Esimerkiksi lisäämällä `select`-elementtiin `option`- tai `optgroup`-elementtejä, saadaan luotua pudotusvalikko- ja monivalintawidgettejä.

```
<!-- varjopuu -->
#shadow-root
  <div id="tabs">
    <slot id="tabsSlot" name="title"></slot>
  </div>
  <div id="panels">
    <slot id="panelsSlot"></slot>
  </div>

<!-- käyttäjän toteutus -->
<fancy-tabs>
  <button slot="title">Title</button>
  <button slot="title" selected>Title 2</button>
  <button slot="title">Title 3</button>
  <section>content panel 1</section>
  <section>content panel 2</section>
  <section>content panel 3</section>
</fancy-tabs>

<!-- renderöity DOM-rakenne -->
<fancy-tabs>
  #shadow-root
    <div id="tabs">
      <slot id="tabsSlot" name="title">
        <button slot="title">Title</button>
        <button slot="title" selected>Title 2</button>
        <button slot="title">Title 3</button>
      </slot>
    </div>
    <div id="panels">
      <slot id="panelsSlot">
        <section>content panel 1</section>
        <section>content panel 2</section>
        <section>content panel 3</section>
      </slot>
    </div>
</fancy-tabs>
```

Ohjelmakoodi 1. *Varjopuusta luodaan DOM-rakenne käyttäjän toteutuksen perusteella.*

Käyttämällä `<slot>`-tagia komponentin kehittäjä voi kutsua käyttäjiä lisäämään omaa sisältöään komponenttiin. Hyödyntämällä `slot`-elementtejä shadow DOM koostaa lopullisen esityksen komponentista. Ne ovat niin sanotusti paikanmerkkejä komponentin sisällä

ulkoisille elementeille. Elementit voivat siis sen avulla ylittää shadow DOM:in rajan. Varsinaisesti DOM:in sisältö ei muutu, vaan `slot`-elementit vain renderöidään lopputuloksessa eri sijaintiin. Komponentti voi määritellä nolla tai useampia `slot`-elementtejä shadow DOM:issa. Slotit voidaan renderöidä tyhjinä tai oletussisällöllä. Nimettyjen slotien avulla määritetään tietty kohta kyseiselle elementille toteutuksessa. Ohjelmakoodissa 1 näkyy miten komponentille `fancy-tabs` luotu shadow DOM renderöidään lopulliseksi DOM-rakenteeksi hyödyntäen koostamista `slot`-elementtien avulla. Käyttämällä nimiattribuuttia on saatu kohdistettua luodut elementit tiettyyn kohtaan toteutusta. (Bidelman, 2016)

3.4.3 Komponentin tyyli

Komponenttia, joka käyttää shadow DOM:ia, voidaan tyylitellä usealla eri tavalla. Tyyli voi tulla pääsivun tyylistä, omista tyyleistä tai käyttäjän toteutuksen perusteella.

Komponentin sisältämä tyyli

Shadow DOM:in sisällä määritellyt CSS-määrittelyt vaikuttavat vain paikallisesti kyseisen komponentin tyyliin. Tämä mahdollistaa geneeristen luokkien ja tunnisteiden käytön, sillä konflikteista muun sivun tyylimäärittelyjen kanssa ei tarvitse välittää. Yksinkertaisia CSS-määrittelyjä on myös helpompi ylläpitää ja ne renderöidään keskimäärin nopeammin (Souders, 2009). Tyylittely voidaan määritellä joko suoraan `<style>`-elementille varjojuuressa tai ulkoisella tyylitiedostolla käyttämällä `<link>`-elementtiä. Komponentin lasten lisäksi myös itse komponentin tyyliä voidaan muokata käyttämällä `:host`-selektoria. Host-selektorin käyttöä on esitelty ohjelmakoodissa 2. Sen huono puoli on se, että jos käyttäjä on muokannut elementtiä, on pääsivun CSS-määrittelyillä korkeampi tarkkuusaste, kuin `:host`-selektoria käyttävillä säännöillä. Tämän seurauksena pääsivun tyyli korvaa komponentin omat tyylit. Funktionaalinen muoto `:host(<selector>)` taas sallii `host`-elementin muokkaamisen, jos suluissa määritelty ehto täyttyy. Hyödyntämällä tätä ominaisuutta voidaan kapseloida käyttäjäinteraktioon tai tilaan perustuvaa toiminnallisuutta tai tyylitellä lapsisolmuja isäntäsolmun tilaan perustuen. (Bidelman, 2016)

Host-elementin tyylit voivat myös olla kontekstisidonnaisia ja tämä tehdään käyttämällä `:host-context(<selector>)` valintaa. Sen ehto toteutuu, jos host-komponentti tai jokin sen isäntäelementeistä toteuttaa annetun ehdon. Tämän avulla voi esimerkiksi toteuttaa erilaisia teemoja, joita käyttäjä voi sitten hyödyntää. Myös käyttäjän lisäämiä `<slot>`-elementtejä voidaan tyylitellä erityisellä CSS-valinnalla. Valinta `::slotted(<compound-selector>)` toimii päätason elementteihin, jotka on renderöity `<slot>`-elementin tilalle. (Bidelman, 2016)

```
<style>
  :host {
    opacity: 0.4;
    will-change: opacity;
    transition: opacity 300ms ease-in-out;
  }
  :host(:hover) {
    opacity: 1;
  }
  :host([disabled]) { /* jos hostilla on disabled attribuutti */
    background: grey;
    pointer-events: none;
    opacity: 0.4;
  }
  :host(.blue) {
    color: blue; /* jos hostilla on luokka blue */
  }
  :host(.pink) > #tabs {
      color: pink; /* jos hostin luokka on blue, sen #tabs lapsielementin
                      väri on pinkki */
  }
</style>
```

Ohjelmakoodi 2. *Host-selektorin käyttö esimerkkitapauksissa.*

Komponentin ulkopuoleinen tyylittely

Komponenttia voidaan tyylitellä suoraan käyttämällä komponentin nimeä. Suoraan komponenttiin kohdistuva tyyli ohittaa shadow DOM:issa määritellyt tyylit. Käyttäjän on kuitenkin helpompi käyttää jo valmiiksi määriteltyjä tyylikoukkuja (*style hooks*), jos komponentin tekijä on sellaisia määritellyt. Nämä toimivat tyyleille samalla tavalla kuin `<slot>`-elementit. Shadow DOM:issa voidaan määritellä esimerkin mukainen tyylikoukku kuten:

```
host([background]) {
  background: var(--fancy-tabs-bg, #9E9E9E);
  border-radius: 10px;
  padding: 10px;
}
```

Tällaiselle tyylikoukulle käyttäjä voi omassa tyylimärittelyssään tehdä toteutuksen taustan tyylille määrittelemällä sen pääsivulla, kuten alla olevassa koodikatkelmassa. Elementin `<fancy-tabs>` väri määräytyy nyt käyttäjän määrittämänä mustaksi, kun oletuksena se olisi väri `#9E9E9E`.

```
<!-- main page -->
<style>
  fancy-tabs {
    margin-bottom: 32px;
    --fancy-tabs-bg: black;
  }
</style>
<fancy-tabs background>...</fancy-tabs>
```

Shadow DOM:in mahdollistamien tyylimäärittelyjen ja erityisesti tyylien komponenttikohtaisten kapselointien avulla on mahdollista luoda monimutkaisia uudelleenkäytettäviä elementtejä, jotka ovat helposti muokattavissa ja käyttäjän tyyliteltävissä halutun laisiksi. Koodin uudelleenkäytettävyys kasvaa, koska toisella sivulla ei tarvitse olla huolissaan tyylien vuotamisesta varjopuun puolelta toiselle ja komponentit näkyvät sellaisena kuin niiden kehittäjä on ne suunnitellut.

3.4.4 Tapahtumien hallinta

Tapahtuman kulkiessa ylöspäin shadow DOM:in sisällä, sen kohdetta muutetaan jatkuvasti toteuttamaan shadow DOM -kapselointi. Tällöin puhutaan tapahtuman kuplimisesta (*bubbling*) puurakenteen sisällä, joka kuvaa hyvin tapahtuman pyrkimistä korkeammalle tasolle, kunnes se käsitellään. Tapahtumaa muutetaan siis niin, että vaikuttaa kuin se tulisi suoraan komponentista, sen sijaan että se tulisi shadow DOM:in sisään piilotetusta elementistä. Kaikkia tapahtumia ei edes kuljeteta shadow DOM:in ulkopuolelle. Varjopuun sisällä luodut mukautetut tapahtumat eivät ylitä DOM-rajapintaa, paitsi jos tapahtuma on luotu käyttäen `composed: true` arvoa. (Bidelman, 2016)

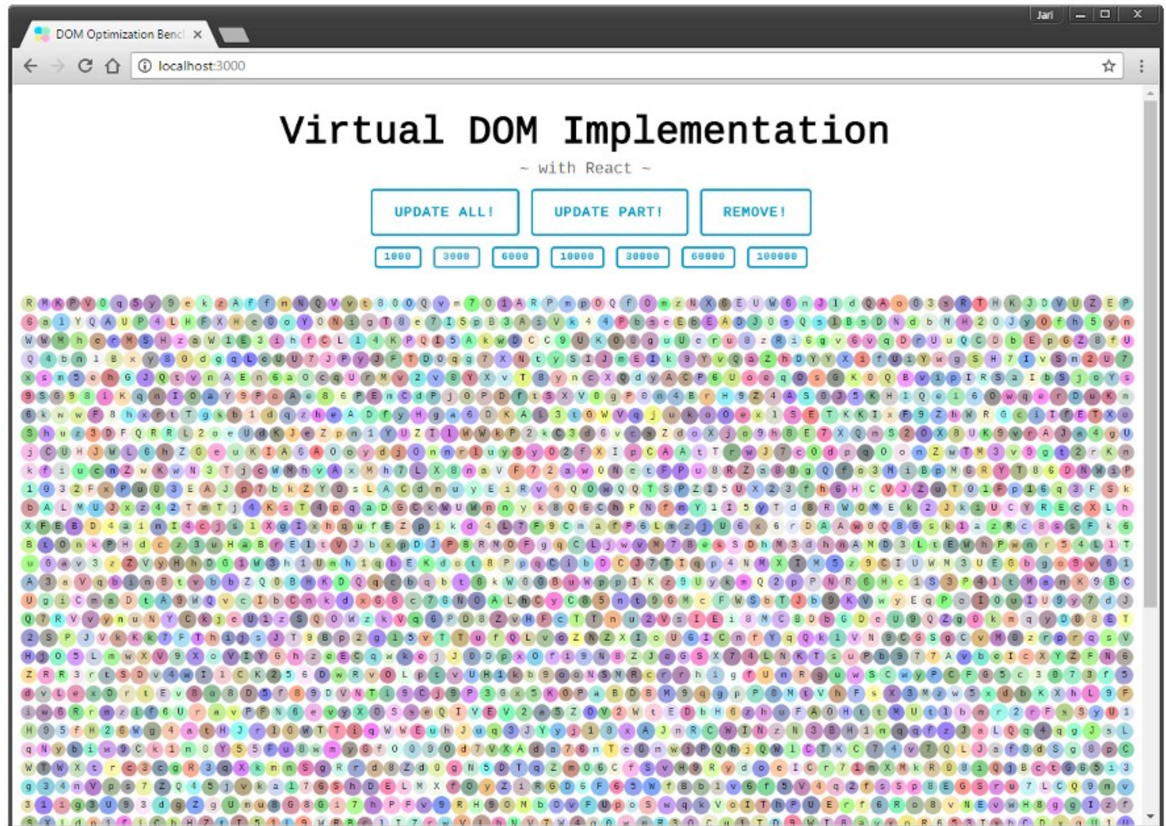
4. SOVELLUSTEN KUVAUS

Tähän työhön liittyen on toteutettu kolme yksinkertaista web-sovellusta, jotka toteuttavat saman toiminnallisuuden. Toiminnallisuus suunniteltiin yksinkertaiseksi, mutta myös selainta DOM:in muokkausoperaatioilla kuormittavaksi. Tärkeä suunnittelukriteeri oli myös saman toiminnallisuuden ja ulkoasun varmistaminen erilaisilla alustoilla. Tämän takia sovellukset toimivat responsiivisen web-suunnittelun periaatteiden mukaan ja tarjoavat ulkoasun puolesta saman käyttökokemuksen. Suoritustehokkuudessa taas on eroja, jotka ovat erityisen kiinnostavia tämän työn puitteissa. Ideoita sovelluksen toiminnallisuuteen haettiin aiemmin julkaistuista JavaScript-kirjastojen ja -sovelluskehysten testauksista. Object Partners (2015) on testannut React.js ja natiivi DOM:in suorituskykyä yksinkertaisella taulukoita luovalla sovelluksella. Auth0:n blogissa taas Peyrott on testannut erilaisten virtual DOM -ratkaisujen suorituskykyä tietokannasta haettujen tietojen esittämisessä (Peyrott, 2016). Vielä kolmantena työhön vaikuttaneena vertailuna on Krausen blogi-artikkelien sarja, joissa vertaillaan erilaisia JavaScript-kirjastoja ja -sovelluskehyksiä useilla taulukon päivitysoperaatiolla (Krause, 2016). Näiden töiden vaikutuksesta päädyttiin valittuun testausmenetelmään ja valittiin sovellusten toteuttama toiminnallisuus. Edellä mainituissa töissä käyttöliittymän muutokset ovat käytännössä tekstipohjaisen tiedon ja taulukoiden esittämistä ja testaus on perustunut pitkälti muistikulutuksen tai JavaScriptin suoritusaikaan. Tämän työn puitteissa kuinkin tahdottiin testata suorituskykyä monimutkaisempien *reflow*- ja *repaint*-operaatioiden yhteydessä huomioiden koko operaation suoritusajan.

Tässä luvussa käsitellään toteutettujen sovellusten samanlaisuuksia ja eroavaisuuksia. Aliluvussa 4.1 esitellään toiminnallisuus yleisellä tasolla. Erilaiset toteutukset taas käsitellään aliluvussa 4.2. Lopulta viimeisessä aliluvussa 4.3 käsitellään mittausten toteutusta. Toteutettujen sovellusten koodit ovat saatavissa lähteessä (GitHub, 2017).

4.1 Toiminnallisuus

Kaikki kolme sovellusta toteuttavat saman toiminnallisuuden. Ne luovat näkymän, jossa on kolme toimintopainiketta ja seitsemän painiketta sisällön luomiseen.



Kuva 20. *Yksi kolmesta toteutetusta sovelluksesta työpöydän selainikkunassa.*

Sovellus toteuttaa käytännössä neljä toimintoa:

- Lisää n kappaletta elementtejä
- Päivitä kaikki elementit
- Päivitä osa elementeistä
- Poista kaikki elementit

Sovelluksen luomat elementit ovat DIV-elementtejä, joista jokaiselle on määritelty satunnainen taustaväri, satunnainen yhden merkin pituinen sisältö ja satunnainen koko. Päivitettäessä kaikkia elementtejä muutetaan kaikkien kolmen muuttujan arvoa, joka johtaa näkymän muuttumiseen. Näihin kolmeen muuttujaan päädyttiin, koska niiden avulla DOM-puusta saadaan riittävän monimutkainen ja CSS:n tyylien ja latomisen ratkaiseminen on tehokkuuden mittaamisen kannalta riittävän vaikea operaatio. Jokainen luotu elementti on oma yksikkönsä ja vastaa omasta sisällöstään ja tyylistään. Kaikkien kolmen toteutuksen toiminnallisuus on kirjoitettu JavaScriptiä käyttäen, joka mahdollistaa kaikissa tapauksissa lähes saman koodin hyödyntämisen. Kuvassa 20 esitellään miltä lopulta toteutettu sovellus näyttää. Ylhäällä otsikon alla ovat toimintopainikkeet ja näiden alla ovat varsinaiset sovelluksen luomat elementit.

Satunnainen sisältö luodaan JavaScriptin Math-objektin `random()`-metodin avulla, joka luo satunnaisia kokonaislukuja. Tässä esitellään apufunktiot, joita jokaisessa toteutuksessa käytetään lähes sellaisenaan sisällön luomiseen eri toteutuksissa. Ainoa kaikista sovelluksista löytyvä apufunktio, joka ei osallistu sisällön määrittämiseen, on alla oleva `scrollToBottom`-funktio.

```
scrollToBottom() {
    window.scrollTo(0, document.body.scrollHeight);
}
```

Selain voi optimoida suoritusta luomalla esityksen vain tiettynä hetkenä näkyvälle osalle sivusta ja renderöidä muun osan sivua tarvittaessa myöhemmin. Tämän `scrollToBottom`-funktion avulla pyritään varmistamaan, että koko näkymä on varmasti päivittynyt ja sitä kutsutaan aina näkymän päivittymisen jälkeen.

Erilaiset apufunktiot, joilla luodaan elementtien satunnainen sisältö, esitellään ovat koodin tasolla työn liitteessä 1. Satunnainen tekstisisältö luodaan `randomContent()`-funktiolla, joka palauttaa halutun mittaisen merkkijonon satunnaisesti valittuja merkkejä. Tässä tapauksessa palautettavan merkkijonon pituus on vain yksi merkki. Elementin sivun pituuden palauttava funktio, `getRandomSize()`, on myös hyvin yksinkertainen. Se palauttaa merkkijonon, joka määrittää satunnaisen leveyden väliltä 14–20 pikseliä. Tätä arvoa voidaan suoraan käyttää CSS-määrittelyissä komponentille. Suhteellisen leveyden arvon sijaan käytetään absoluuttista arvoa, koska responsiivisuudesta huolehditaan latomalla elementit juostavasti. Viimeisenä apufunktiona sovelluksissa on käytössä `generateColor()`, joka luo satunnaisen kirkkaan värin. Tämä funktio sekoittaa valkoisen värikanaviin satunnaisesti luodun värin arvot ja muuttaa sen HEX-arvoiseksi väriksi. Palautettua merkkijonoa voidaan suoraan käyttää CSS-määrittelynä väristä. Yhteisten apufunktioiden lisäksi kaikissa eri toteutuksissa käytetään käytännössä samaa CSS-määrittelyä, mutta sitä on pilkottu komponentteihin toteutustapaan soveltuvalla tavalla. CSS-määrittely koostuu pääpiirteissään kolmesta osasta: tyylin määrittelystä pääsivulle ja sen painikkeille, tyylin määrittelystä elementtien alueelle ja yksittäisten elementtien tyylistä.

Ohjelmakoodissa 3 esitellään `DIV`-elementin tyyli, jonka sisälle yksittäiset elementit luodaan. Tärkeä huomio on, että elementtien järjestämiseen käytetään CSS3:sen `display: flex` tyyppistä joustavaa säiliötä (*flexbox*). Lisäämällä siihen flex-flow attribuutiksi määrittely `row wrap`, sisälle renderöitävät yksittäiset elementit tasataan riveihin automaattisesti. Tällöin ei ole tarvetta luoda säiliöitä esimerkiksi jokaiselle riville erikseen, vaan sisältö tasataan dynaamisesti sisällön pituudesta tai näytön leveydestä huolimatta.

```
div.container {
    display: flex;
    width: 100%;
    flex-flow: row wrap;
    -webkit-flex-flow: row wrap;
    justify-content: space-between;
    align-items: center;
    margin-top: 215px;
}
```

Ohjelmakoodi 3. *Tyyli alueelle, johon yksittäisen elementit renderöidään.*

```
@media screen and (max-width: 560px) {
    h1 { font-size: 30px !important }
    .btn:not(.add-btn) { font-size: 12px; padding: 8px 15px !important}
    div.container {
      margin-top: 175px;
    }
}
```

Ohjelmakoodi 4. *Esimerkki mediakyselyistä, joita käytetään responsiivisuuden määrittämiseen erilaisilla näytönleveyksillä.*

Tämän lisäksi dynaamisen sisällön esittämiseen vaikuttaa ohjelmakoodin 4 mukainen mediakysely. Vastaavat mediakyselyt toteutettiin myös maksimileveyksille 350 pikseliä, 330 pikseliä ja 293 pikseliä. Nämä muutospisteet löydettiin tähän sovellukseen kokeilemalla sivun toimintaa erilaisilla näyttöleveyksillä. Yksittäisen renderöitävän elementin tyyli on ohjelmakoodin 5 mukainen. Sen avulla jokainen elementti on ympyrän muotoinen, kuten aiemmin esitellyssä kuvassa 20 näkyy. Joustavan säiliön käyttäminen vaatii `display: flex` -ominaisuuden määrittämistä myös yksittäisille elementeille.

```
.row-item {
    font-family: 'Cousine', monospace;
    font-size: 10px;
    text-align: center;
    border-radius: 50%;
    display: flex;
    vertical-align: middle;
    align-items: center;
    justify-content: center;
}
```

Ohjelmakoodi 5. *Yksittäisen renderöitävän row-item-luokkaisen elementin tyyli.*

4.2 Toteutetut sovellukset

Tässä luvussa esitellään kolmen eri toteutuksen tapoja toteuttaa sama toiminnallisuus. Ratkaisuissa on pyritty toteutustekniikalle ominaiseen tapaan ilmaista sama asia ja luoda mahdollisimman luonnollinen, mutta myös tehokas ratkaisu. Esitysjärjestys on valittu työn todellisen toteutusjärjestyksen mukaan. Jokainen sovellus pyrittiin toteuttamaan toisista riippumatta, mutta alussa tehdyt toteutusratkaisut ovat varmasti vaikuttaneet seuraaviin sovelluksiin, joten myös työn esittely toteutusjärjestyksessä tuntuu perustellulta. Sovelluksista ensimmäisenä kirjoitettiin Angular 2 -toteutus, seuraavana React-toteutus ja viimeisenä perinteisempi HTML- ja jQuery-pohjainen toteutus.

4.2.1 Angular 2 -sovellus

Angular 2 -sovellus on luotu käyttäen Angular CLI -työkalua, koska sen avulla kehitysympäristö oli helppo alustaa ja sitä suositellaan Angular 2 -sovellusten rakentamiseen (Google, 2017). Tämä aiheuttaa tietynlaisen hakemistorakenteen ja tiettyjä riippuvuuksia sovellukselle. Jatkossa tässä työssä Angular-termillä viitataan aina Angular 2 -versioon. Angular-sovellusten rakentamiseen käytetään JavaScriptin TypeScript-laajennusta. Tämä sovellus koostuu kahdesta komponentista: `AppComponent`:ista ja `RowItemComponent`:ista. Seuraavaksi esitellään sovelluksen kansiorakenne, joka kertoo myös koodin ja toiminnallisten rakenteiden jaosta.

```
./angular2
    src/
        app/
            row-item/
                row-item.component.css
                row-item.component.html
                row-item.spec.ts
                row-item.componenet.ts
            app.component.css
            app.component.html
            app.component.spec.ts
            app.module.ts
        favicon.png
        index.html
```

Komponentit koostuvat oletuksena neljästä tiedostosta, jotka muodostavat komponentin tyylin, rakenteen sekä toiminnallisuuden. Neljäntenä `spec.ts`-päätteiseen tiedostoon voidaan kirjoittaa komponentin yksikkötestit. Tällainen jako lisää suoraan modulaarisuutta, koska tietyn ominaisuuden määrittävä koodi on selkeästi eriytetty omaan tiedostoonsa. Samalla kuitenkin tällainen toteutus lisää hakemistorakenteen kompleksisuutta suuresti.

`RowItemComponent`:in vastuualueena on luoda yksittäinen elementti ja säilöä sen ulkoasua koskeva tieto. Se myös tarjoaa rajapinnan sisällön päivittämiseen ja implementoi aiemmin määritellyt apufunktiot. Ohjelmakoodissa 6 esitellään luotua `RowItemComponent`:ia. Ensimmäisellä rivillä määritetään importit eli luokat, joihin viitataan ulkoisissa lähteissä. `Component`-luokka tuo Angular-tyyppisen komponentin toiminnallisuuden luodulle komponentille ja mahdollistaa `@component`-tyyppisen lisämäärittelyn (*decoration*) käytön ohjelmakoodissa. `ViewEncapsulation` mahdollistaa komponentille natiivin shadow DOM:in käyttämisen emuloidun tai tavallisen DOM:in sijaan. Tämä on olennaista sallia, jotta päästään testaamaan juuri shadow DOM:in tehokkuutta.

Komponentille määritellään kolme yksityistä merkkijonomuotoista muuttujaa: *content*, *color* ja *size*. Luokan rakentaja kutsuu komponenttia luotaessa funktiota `update`, joka aiemmin määriteltyjä apufunktioita käyttäen luo sisällön komponentille ja muuttaa näitä kolmea muuttujaa. Funktio `updateSome` taas toimii niin, että se pyytää satunnaista kokonaislukua suljetulta väliltä 1–5 ja arvon ollessa 5, kutsuu myös `update`-funktiota. Käytännössä riittävän suurella joukolla, joka viides elementti päivittyy.

```
import { Component, ViewEncapsulation } from '@angular/core';

@Component({
  selector: 'app-row-item', templateUrl: './row-item.component.html',
  styleUrls: ['./row-item.component.css'],
  encapsulation: ViewEncapsulation.Native
})
export class RowItemComponent {

  private content: string
  private color: string
  private size: string

  constructor() {
    this.update();
  }

  update() {
    this.content = this.randomContent();
    this.color = this.generateColor();
    this.size = this.getRandomSize();
  }

  updateSome() {
    var rand = Math.floor((Math.random() * 5) + 1);
    if (rand === 5) {
      this.update();
    }
  } // lyhennetty...
}
```

Ohjelmakoodi 6. *RowItemComponent:in määrittely Angular-toteutuksessa.*

Varsinainen `RowItemComponent`:in HTML-toteutus on hyvin yksinkertainen ja koostuu ainoastaan yhdestä DIV-elementistä, jolle on määritelty tyyli Angularin tyylisidosten (*style binding*) avulla. Varsinainen sisältö taas tuodaan näkymään interpolaation eli tupla-

aaltosulkusyntaksin avulla. Sen avulla merkkijonomuotoinen `content`-muuttuja näytetään elementin sisältöänä. Angular sitoo muuttujan arvon käyttöliittymäelementtiin ja päivittää sitä automaattisesti sisällön muuttuessa. Toteutus näkyy ohjelmakoodissa 7.

```
<div class="row-item"
    [style.background]="color"
    [style.height]="size"
    [style.width]="size">{{ content }}</div>
```

***Ohjelmakoodi** 7. Angular-toteutuksen RowItem-komponentin HTML-malli.*

`AppComponent`:in vastuualueena on luoda näkymä, johon sovellukset luomat yksittäiset `RowItemComponent`:it voidaan sijoitta. Se vastaa elementtien luomisesta ja käynnistää niiden päivittämisen sekä poistamisen. Tällainen isännältä lapsi-elementille kulkeva tapahtuman aktivointi on vastoin komponenttien yleistä käyttötapaa tuoda arvoja ja tapahtumia hierarkiassa ylöspäin. Kuitenkin näin pienessä ja triviaalissa sovelluksessa tästä yleisestä suunnittelutavasta poiketaan, koska se sopii paremmin halutun toiminnallisuuden toteuttamiseen.

Ohjelmakoodissa 8 esitellään `AppComponent`:in perusrakenne. Tuonneista mielenkiintoisia ovat `ViewChildren` ja `QueryList`, joiden avulla on mahdollista viitata komponentin lapsielementteihin ja näin käynnistää päivitysoperaatio `RowItem`-komponenteissa. Tässä luodaan paikallinen muuttuja `rowItems`, joka on käytännössä lista viittauksia yksittäisiin `RowItem`-elementteihin. `AfterViewChecked` liittyy Angularin näkymän elinkaareen. Käytännössä tämä mahdollistaa funktion `ngAfterViewChecked` implementoinnin. Tämä funktio suoritetaan näkymän päivittämisen jälkeen, kun näkymä on päivitetty. Tässä tapauksessa se kutsuu funktiota `scrollToBottom` eli siirtää näkymän luodun sisällön loppuun.

```
import { Component, ViewEncapsulation, ViewChildren, QueryList,
AfterViewChecked } from '@angular/core';
import { RowItemComponent } from './row-item/row-item.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  encapsulation: ViewEncapsulation.Native
})
export class AppComponent implements AfterViewChecked {

  @ViewChildren('rowItem') rowItems:QueryList<RowItemComponent>;
  items = [];

  ngAfterViewChecked () {
    this.scrollToBottom();
  }
  // lyhennetty...
}
```

Ohjelmakoodi 8. *AppComponent:in perusrakenne Angular 2 -sovelluksessa.*

`AppComponent` toteuttaa ohjelmakoodin 9 mukaisen rajapinnan yksittäisten elementtien muokkaamiselle aiemman toimintojen määrittelyn mukaisesti. Se on toiminnallisuudeltaan ja syntaksiltaan yksinkertainen. Näistä `add`-funktio muuttaa näkymässä olevien elementtien määrää parametrinsa mukaisesti. Käytännössä se luo uuden tyhjän taulukon, joka on pituudeltaan saamansa parametrin mukainen. Myöhemmin selitetään miksi tämä muuttaa näkymän `RowElement`:tien lukumäärää. Funktiot `updateAll` ja `updateSome` käsittelevät listaa `RowItem`-viittauksista ja kutsuvat jokaiselle komponentille vastaavaa funktiota, jonka `RowItem`-komponentti toteuttaa. Lopulta `add`-funktiota vastaavaa mekanismia hyödyntäen `remove`-funktio poistaa elementit asettamalla `items`-taulukon tyhjäksi taulukoksi.

```
add(count: number) {
  this.items = new Array(count);
}
updateAll() {
   this.rowItems.forEach( (item) => item.update() );
}
updateSome() {
   this.rowItems.forEach( (item) => item.updateSome() );
}
remove() {
  this.items = [];
}
```

Ohjelmakoodi 9. *AppComponent:in toteuttama rajapinta RowItem:ien muokkaamiseen Angular 2 -sovelluksessa.*

Ohjelmakoodissa 10 määritellään AppComponent:in HTML-muoto. Käytännössä alustetaan painikkeet ja alue, johon `RowItem`:it sijoitetaan. Kolmanneksi viimeisellä rivillä toteutetaan logiikka, jonka perusteella jokaista items-taulukossa olevaa solua kohden luodaan uusi `app-row-item`-elementti. Tämä tehdään käyttämällä Angularin `*ngFor`-silmukkaa. Sen sisältö luodaan näkymään niin monta kertaa kuin `items`-taulukossa on elementtejä. Luoduille elementille määritetään viite nimellä `#rowItem`, joka mahdollistaa ohjelmakoodin 9 mukaisen viittauksen `RowItem`-elementtehin.

```
<div id="root">
  <div class="header">
    <h1>Shadow DOM Imp&shy;le&shy;ment&shy;ation</h1>
    <span class="subheader">~ with Angular 2 ~</span>
    <div class="btn-row">
      <button class="btn btn-primary outline" id="update-all"
           (click)="updateAll()">Update All!</button>
      <button class="btn btn-primary outline" id="update-part"
           (click)="updateSome()">Update Part!</button>
      <button class="btn btn-primary outline" id="remove"
           (click)="remove()">Remove!</button>
    </div>
    <div class="btn-row">
      <button class="btn btn-primary outline add-btn"
           (click)="add(1000)">1000</button>
      <!-- lyhennetty muiden lisäyspainikkeiden määrittely -->
    </div>
  </div>

  <div class="container" id="container-for-row-items">
    <app-row-item #rowItem *ngFor="let item of items"></app-row-item>
  </div>
</div>
```

Ohjelmakoodi 10. *AppComponent:in HTML-määrittely Angular 2 -sovelluksessa.*

4.2.2 React-sovellus

React-sovelluksen luomiseen on käytetty React-ympäristön vastinetta Angular CLI -työkalulle eli *Create React App* -työkalua. Se on Facebookin virallisesti tukema työkalu React-sovellusten kehitysympäristön yksinkertaiseen pystyttämiseen (Abramov, 2016). Se luo käytännössä alustavan rakenteen projektille.

Työkalun avulla luotu rakenne on olennaisilta osin esitetty allaolevassa hakemistokuvauksessa. `RowItem`-komponentille on luotu oma hakemisto *components*, johon olisi voitu sijoittaa myös `App`-komponentti. Myös Reactin tapauksessa tyyli ja toiminnallisuus on erotettu toisistaan, mutta verrattaessa Angular-toteutukseen huomataan, että erillisiä HTML-tiedostoja ei ole. Tämä johtuu Reactin JSX-implementaatiosta, joka tekee niistä tarpeettomia. Mielestäni tämä luo turhaa kompleksisuutta JavaScript-tiedostoihin, mutta toimii hyvin riittävän lyhyissä HTML-tiedostoissa.

```
react-impl/
    public/
        favicon.png
        index.html
    src/
        components/
            rowItem.css
            rowItem.js
        app.cs
        app.js
        app.test.js
        index.css
        index.js
```

Reactin toteutus `RowItem`-komponentista näkyy ohjelmakoodissa 11. Siinä hyödynnetään tuonteja ja laajennetaan `RowItem`-luokkaa Reactin `Component`-luokalla. Reactissa komponentin tila asetetaan tila-objektiin (`state`). Sinne on tässä tapauksessa tallennettu komponentin sisältö. Komponentin koko ja väri taas määritellään `style`-objektin avulla. Koon määrittävä muuttuja (*size*) ei ole `state`-objektin sisällä, koska komponentti voidaan renderöidä ennen kuin varsinainen tila on asetettu. Muuttuja on erillään myös sen takia, että `style`-objektille voidaan tällöin helposti käyttää sekä leveyden että korkeuden määrittämiseen samaa arvoa. Reactille hyvin omalaatuinen kohta ohjelmakoodissa on `render`-metodi. Sitä kutsutaan aina kun komponentin esitys näkymässä päivitetään. Sen sisällä käytetään JSX-tyyppistä esitystä sisällöstä, joka halutaan renderöidä. Varsinainen sisältö on jälleen DIV-elementti, jolle on määritelty luokaksi *row-item*. Tyyliksi sille on asetettu aiemmin määritelty `style`-objekti ja sisältö taas on `state`-objektista noudettu sisältö, joka saadaan näkymään käytetyllä aaltosulkusyntaksilla.

```
import React, { Component } from 'react';
import './RowItem.css';

class RowItem extends Component {
    size = this.getRandomSize();
    constructor() {
        super();
        this.state = {
            content: this.randomContent()
        };
        this.style.backgroundColor = this.generateColor();
        this.size = this.getRandomSize();
    }
    render() {
        return (
             <div className="row-item" style={ this.style }>
                 { this.state.content }
             </div>
        );
    }

    style = {
        backgroundColor: this.generateColor(),
        height: this.size,
        width: this.size
    };
    // lyhennetty...
}
export default RowItem;
```

Ohjelmakoodi 11. *Toteutus RowItem-komponentista Reactilla.*

Reactilla luotu `RowItem`-komponentti toteuttaa saman rajapinnan kuin Angularilla toteutettu versio. `Update`-funktio esitellään ohjelmakoodissa 12. Ohjelmakoodin alussa muutetaan komponentin sisältöä kutsumalla `setState`-metodia. Tyylin muuttaminen vaatii tyyli-objektin kopioimista tai muuten kääntäjä antaa varoituksen tyylin muuttamisesta

suoraan. Varsinaista `render`-metodia ei tarvitse kutsua, koska komponentin tilan muuttaminen `setState`-funktiolla käynnistää näkymän päivittämisen aliluvun 3.3.1 mukaisesti. RowItemin `updateSome`-funktio vastaa käytännössä Angular-version vastaavaa funktiota, joten sitä ei käsitellä tässä tarkemmin.

```
update( ) {
    this.setState({ content: this.randomContent( ) });
    var clone = Object.assign({}, this.style);
    this.style = clone;
    this.style.backgroundColor = this.generateColor( );
    this.size = this.getRandomSize( );
}
```

Ohjelmakoodi 12. *Toteutus RowItem:in Update-funktiosta Reactilla.*

Myös React-sovelluksessa haetaan viitteet yksittäisiin päivitettäviin *RowItem*-komponentteihin kuten Angular-sovelluksessa. Viitteet haetaan hyödyntämällä Reactin `ref`-attribuuttia. Reactin kehitysohjeissa kehotetaan käyttämään niin sanottua ”*callback-mallia*” viittausten lisäämiseen ja välttämään niiden liiallista käyttöä. Sovelluksessa kuitenkin haluttiin, että tieto muutospyynnöstä kulkee isännältä lapsikomponenteille kuten Angular-sovelluksessa. Tämän takia päädyttiin käyttämään merkkijonomuotoisia `ref`-viittauksia, jotta koko lapsikomponenttien joukkoon saadaan mahdollisimman helposti viitattua. Ohjelmakoodi 13 sisältää funktion, jonka avulla saadaan halutun kokoinen joukko `RowItem`-komponentteja. Siinä käytetään JSXElementtiä hyödyksi ja jokaiselle `RowItem`:ille luodaan oma `ref`-viite ja indeksiin perustuva `key`-attribuutti. React käyttää tätä `key`-attribuuttia yksittäisen elementin muutosten havainnointiin (Facebook, 2017). Ohjelmakoodissa 14 esitellään `refsToArray`-funktio, joka perustuu lähteeseen ("FakeRainBrigand", 2014). Sen avulla saadaan taulukko, joka sisältää viittaukset kaikkiin RowItem-komponentteihin.

```
renderItems() {
    let items = [];
    var _refi = 0;
    var makeRef = function () { return 'RowItem-' + (_refi++); };
    for (var i = 0; i < this.state.count; ++i) {
      var val = "item-" + i;
      items.push(<RowItem ref={ makeRef() } key={ val }></RowItem>);
    }
    return items;
}
```

Ohjelmakoodi 13. *Reactin renderItems-funktio, joka luo yksittäiset RowItem-komponentit.*

```
refsToArray(ctx, prefix) {
  let results = [];
  for (var i = 0; ; i++) {
    var ref = ctx.refs[prefix + '-' + String(i)];
    if (ref) results.push(ref);
    else return results;
  }
}
```

Ohjelmakoodi 14. *Reactin App-komponentin reftsToArray-funktio, joka palauttaa yksittäisiin RowItem-komponentteihin viittaukset taulukossa.*

Lopulta varsinainen App-komponentti rakennetaan Reactilla ohjelmakoodin 15 mukaisesti. Komponentin rakentajassa määritellään `state`-objekti, joka pitää sisällään luotavien elementtien lukumäärän. Kun `render`-metodia kutsutaan, se luo näkymän ja kutsuu ohjelmakoodin 13 mukaista `renderItems`-funktiota.

```
import React, { Component } from 'react';
import './App.css';
import RowItem from './components/RowItem.js';

class App extends Component {
// lyhennetty rakentaja, joka alustaa state-objektille arvon count: 0
render( ) {
  return (
    <div id="wrap">
      <!-- lyhennetty otsikot ja painikkeet -->

      <div className="container" id="container-for-row-items">
        { this.renderItems( ) }
      </div>
    </div>
    );
  }  // lyhennetty...
}
export default App;
```

Ohjelmakoodi 15. *React-toteutus App-komponentista.*

Varsinainen toiminnallisuus App-komponentille luodaan hyvin samaan tapaan kuin Angular-toteutuksessa. Tätä esitellään ohjelmakoodissa 16. Lisääminen tapahtuu muuttamalla komponentin tila-objektin `count`-muuttujan arvoa ja samoin arvoa muuttamalla toimii myös elementtien poistaminen. Päivittävissä funktioissa pyydetään viitteet yksittäisiin `RowItem`-elementteihin, minkä jälkeen kutsutaan vastaavaa funktiota ulkoasun päivitykseen. Metodia `componentDidUpdate` kutsutaan, kun komponentti on päivittynyt eli kun elementtejä on lisätty tai poistettu. Metodi toimii samoin kuin `ngAfterViewChecked`-metodi Angularissa ja siirtää näkymän sivun sisällön loppuun. Kuitenkin `RowItem`:in väriä, kokoa ja sisältöä muutettaessa `Update`-funktioilla, pitää `scrollToBottom`-funktiota kutsua käsin, sillä lapsikomponenttien päivittyessä varsinaisesti isäntäkomponentin sisältö ei Reactin mielestä muutu.

```
add(count) {
    this.setState({count: count})
}
updateAll() {
  this.refsToArray(this, "RowItem").forEach( (i) => i.update() );
  this.scrollToBottom();
}
updateSome() {
  this.refsToArray(this, "RowItem").forEach( (i) => i.updateSome() );
  this.scrollToBottom();
}
remove() {
  this.setState({count: 0});
}
componentDidUpdate() {
  this.scrollToBottom();
}
```

Ohjelmakoodi 16. *React-toteutus App-komponentin rajapinnasta RowItem:ien muokkaamiseen ja näkymän päivittämiseen.*

4.2.3 jQuery ja HTML -sovellus

Perinteistä DOM:ia hyödyntävä toteutus käyttää jQuery-kirjastoa, joka helpottaa JavaScriptin kirjoittamista ja on kirjoittajalle tutumpi toteutustapa kuin puhdas JavaScript-toteutus. Tässä toteutuksessa DOM:in käsittely ja suorituksen optimointi on sovelluskehittäjän vastuulla, mutta samalla myös mahdollisuudet sisällön päivittämiseen ja sen toiminnan säätämiseen ovat rajoittamattomat. Perinteistä DOM:ia käyttävä sovelluksen hakemistorakenne on alla olevan muotoinen.

Sovelluksen varsinainen rakenne on yksinkertaisempi kuin muissa toteutuksissa ja käytännössä kaikki tietyntyyppinen toiminnallisuus on koottu siitä vastaavan tiedoston alle. HTML:ää lukuun ottamatta tyylitiedostoa ja JavaScript-tiedostoa olisi voitu jakaa pienempiin osiin Angularia ja Reactia hyödyntävien toteutusten tyyliin, mutta niin ei ole tehty koska mikään toteutukseen liittyvä syy ei vaadi sitä.

```
regular/
    css/
        styles.css
    js/
        dom-benchmark.js
        jquery-3.2.0.min.js
    favicon.png
    index.html
```

Yllä on esitelty toteutetun sovelluksen rakenne. HTML-toteutusta ei käydä tässä kovin syvällisesti läpi, koska se sisältää hyvin vähän mitään edellisistä toteutuksista poikkeavaa, paitsi että siinä määritellään kerralla koko sivun rakenne yhdessä tiedostossa. Kuitenkin yksityiskohtana alla olevan koodikatkelman mukainen `template`-tagin käyttö on hyvä esitellä.

```
<template id="row-item-template">
  <div class="row-item"></div>
</template>
```

Tätä `template`-tagia käytetään myöhemmin pohjana luotaville `RowItem`-komponenttien instansseille, vaikka tässä toteutuksessa komponenttitermin käyttö on harhaanjohtavaa. Tavallisessa toteutuksessa JavaScriptissä ei ole luokkia, joilla olisi omia metodeja tai tila. Tällaisella template-tagillä kerrotaan selaimelle, että kyseistä elementtiä ei renderöidä sivulle sivua ladatessa, vaan sen varsinainen sisältö täytetään JavaScriptillä myöhemmin (MDN, 2017).

Yksittäisen elementin lisääminen onnistuu tässä toteutuksessa `template`-tagia hyödyntäen ohjelmakoodin 17 mukaisesti. Lisäys-painikkeen painaminen kutsuu tätä `add`-funktiota, joka saa halutun testielementtien lukumäärän parametrinaan. Käyttämällä kahta eri selektoria pyydetään viite sekä komponentin malliin (`template`) että varsinaiseen sijaintiin HTML-koodissa (`slot`). Kahta eri tapaa käytetään siksi, että mallia tulee käsitellä HTML-elementtinä, kun taas sijainnin kohteen käy myös jQuery-objekti. Silmukkarakenteessa luodaan elementin mallista varsinainen uusi elementti ja päivitetään sille alustava sisältö `update`-funktiota käyttäen. Lopulta luotu elementti sijoitetaan `slot`-elementin sisään viimeiseksi lapsielementiksi.

```
function add(count) {
    var template = document.querySelector('#row-item-template');
    var slot = $('#container-for-row-items');

    for (var i = 0; i < count; ++i) {
      rowItem = template.content.querySelector('.row-item');
      update($(rowItem));
      var clone = document.importNode(template.content, true);
      $(clone).appendTo(slot);
    }
    scrollToBottom();
}
```

Ohjelmakoodi 17. *Perinteisessä toteutuksessa elementin lisääminen näkymään.*

Ohjelmakoodissa 18 esitellään, miten yksittäistä elementtiä päivitetään tässä toteutuksessa. Funktio `update` saa parametrinaan jQuery-muotoisen objektinviitteen, jolle määritetään jQueryn `html`- ja `css`-metodeja käyttäen uudet arvot. Niillä pystytään suoraan muokkaamaan parametrina saadun elementin sisältöä ja tyyliä. Tällöin muutos päivittyy heti näkymään ja edustaa suoraa vuorovaikutusta DOM:in kanssa.

Varsinainen isäntäelementin rajapinta taas pelkistyy ohjelmakoodin 19 mukaiseksi toteutukseksi. Elementtien luokan perusteella haetaan jQueryn avulla elementeistä koostuva joukko, jonka jokaiselle alkiolle kutsutaan vastaavaa metodia ja päivitetään näkymä sivun alaosaan.

```
function update(item) {
    item.html(randomContent());
    item.css('backgroundColor', generateColor());
    var size = getRandomSize();
    item.css('height', size);
    item.css('width', size);
  }
```

Ohjelmakoodi 18. *Perinteisen toteutuksen yksittäistä elementtiä päivittävä funktio.*

```
function removeAll() {
    $('#row-for-row-items').empty();
}
function updateAll() {
  $('.row-item').each(function () {
    update($(this));
  });
  scrollToBottom();
}
function updatePart() {
  $('.row-item').each(function () {
    updateSome($(this));
});
  scrollToBottom();
}
```

Ohjelmakoodi 19. *Isäntäelementin rajapinta perinteisessä toteutuksessa.*

Suora vuorovaikutus DOM:in kanssa saattaa vaikuttaa yksinkertaisimmalta toteutukselta ja varmasti onkin sitä näin yksinkertaisessa tapauksessa. Vastuu tehokkuudesta kuitenkin jää täysin kehittäjälle ja monimutkaisemman toiminnallisuuden toteuttaminen tällaisella tavalla voi osoittautua hyvin hankalaksi ylläpitää. Tällaisen mallin avulla toteutusta on myös vaikea jakaa helposti yksikkötestattaviksi osiksi, joka tekee koodin laadun seuraamisesta hankalaa.

4.3 Mittausten toteutus

Mittaukset toteutettiin käyttämällä Chrome-selaimen *Developer Tools* -työkalun *Timeline*-toiminnallisuutta. Tämän työkalun etuna ovat monipuoliset tulokset ja mahdollisuus suorittaa mittauksia mobiililaitteilla *Remote Devices* -toiminnon avulla. Mittaukset tehtiin ainoastaan yhdellä selaimella, mikä vaikuttaa saatuihin mittaustuloksiin. Käyttämällä myös muita selaimia ja hyödyntämällä niiden erilaisia toteutuksia JavaScript-moottorista olisi saatu parempi käsitys erilaisen toteutustapojen absoluuttisesta tehokkuudesta. Pelkkään Chrome-selaimeen kuitenkin päädyttiin, koska siitä on olemassa mobiiliversio samalla selainmoottorilla, sen tarjoamaa Timeline-työkalua voidaan suoraan käyttää myös mobiililaitteiden testauksessa ja se implementoi natiivin tuen shadow DOM:in käytölle

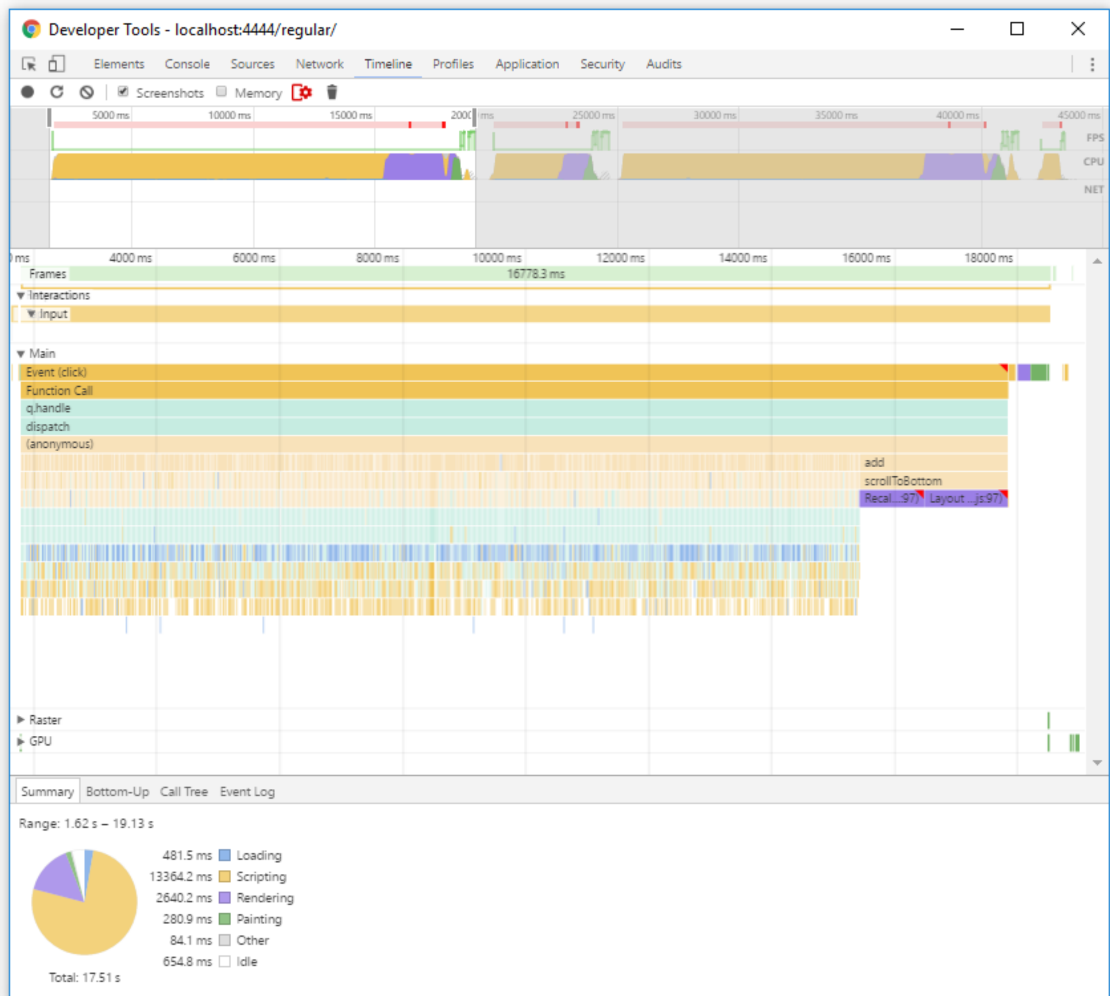
niin mobiili- kuin työpöytäselaimella (Google, 2016). Chrome-selain on myös sekä Suomessa että globaalisti käytetyin selain 20–30% erolla kilpailijoihin nähden (Netmarketshare, 2017; W3Counter, 2017; StatsCounter, 2017). Eri selaimilla testauksen sijaan toteutettuja sovelluksia testattiin kolmella erilaisella laitteella, jotka on esitelty taulukossa 1. Työhön valitut laitteet edustavat puhelinta, tablettia ja kannettavaa tietokonetta.

Taulukko 1. *Testilaitteet.*

Tyyppi	Laite	Tekniset ominaisuudet
Puhelin	Sony Xperia Z3	Näyttö: 1080 x 1920, 8,9 tuumaa Suoritin: Snapdragon 801 2.5 GHz Muisti: 3 GB RAM Käyttöjärjestelmä: Android 6.0
Tabletti	Google Nexus 9	Näyttö: 2048 x 1536, 5,2 tuumaa Suoritin: NVIDIA Tegra K1 2.3 GHz Muisti: 2 GB RAM Käyttöjärjestelmä: Android 7.1.1
Kannettava tietokone	Lenovo Thinkpad T460P	Näyttö: 2560 x 1440, 14 tuumaa Suoritin: Intel Core i7-6820HQ 2.7 GHz Muisti: 16 GB RAM Käyttöjärjestelmä: Windows 10 Pro

Kuvassa 21 näkyy näytönkaappaus yhdestä testisekvenssistä. Testaus toteutettiin niin, että Timeline-työkalulla tallennettiin seuraava testisekvenssi:

- Klikkaa ”lisää *n”*-painiketta → odota näkymän muutos
- Klikkaa Muokkaa osa -painiketta → odota näkymän muutos
- Klikkaa Muokkaa kaikki -painiketta → odota näkymän muutos
- Klikkaa Poista-painiketta → odota näkymän muutos



Kuva 21. *Chromen DevTool:sin käyttö testauksessa.*

Kuvan mukaisesta testisekvenssistä tarkasteltiin näkymän päivitykseen kulunutta aikaa. Työhön tarkasteltavaksi muuttujaksi valittiin näkymän muuttumiseen kuluva aika käyttäjän vuorovaikutuksesta ruudunpäivitys nopeuden normalisoitumiseen. Tämä kuvaa mahdollisimman hyvin toteutustavan valinnan vaikutusta käyttäjäkokemukseen. Mittaukset suoritettiin Chrome-selaimen *Incognito*-tilassa, jotta esimerkiksi selaimen laajennukset ovat poissa käytöstä ja eivät hidasta suoritusta.

Testaus jouduttiin tekemään käsityönä, jotta voitiin mitata aika käyttäjätoiminnasta JavaScriptin suoritukseen, selaimen *paint*-operaatioon ja lopulta ruudunpäivitysnopeuden normalisoitumiseen. Muilla JavaScriptin profilointityökaluilla, kuten JavaScriptissä käytettävät `performance.now()` ja `console.time()`, olisi ollut mahdollista saada selville

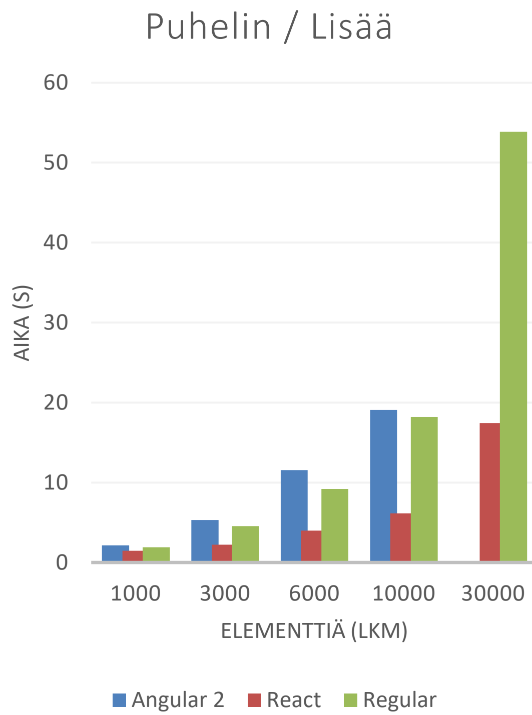
vain varsinaiseen prosessointiin kuluva aika. Testien lukumäärää jouduttiin karsimaan, koska tulokset jouduttiin hankkimaan manuaalisesti. Lopulta jokaisella laitteella testattiin kaikkia kolmea toteutusta ja niiden neljää eri operaatiota seitsemällä elementtijoukon koolla viisi kertaa. Kaikilla alustoilla kuitenkaan seitsemän elementtijoukon testaaminen ei ollut ajallisesti ja tuloksellisesti järkevää, esimerkiksi Angularin testaus 100 000 alkiolla kesti yli neljä minuuttia ja DevToolsin bufferi, jossa testin tiedot säilytetään ennen visualisointia, loppui kesken. Tästä johtuen tuloksissa testit rajattiin elementtijoukoille 1000, 3000, 6000, 10000 ja 30000. Näin myös saatujen kuvaajien tulokset pysyivät luku- ja vertailukelpoisina.

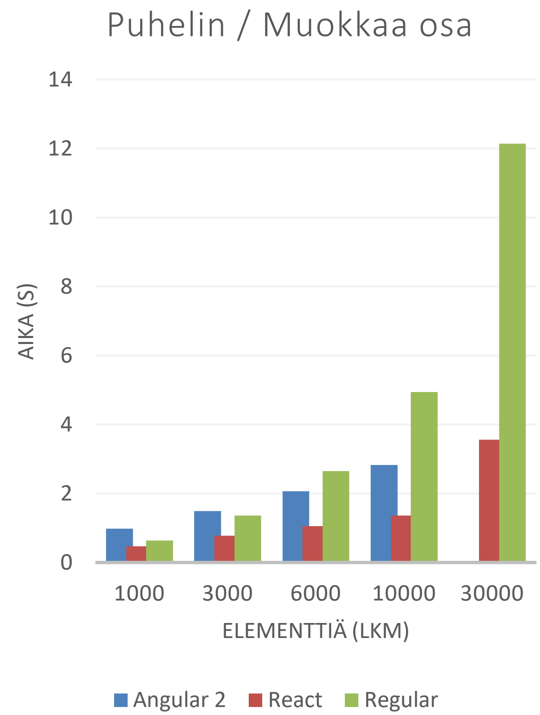
5. TULOKSET JA ARVIOINTI

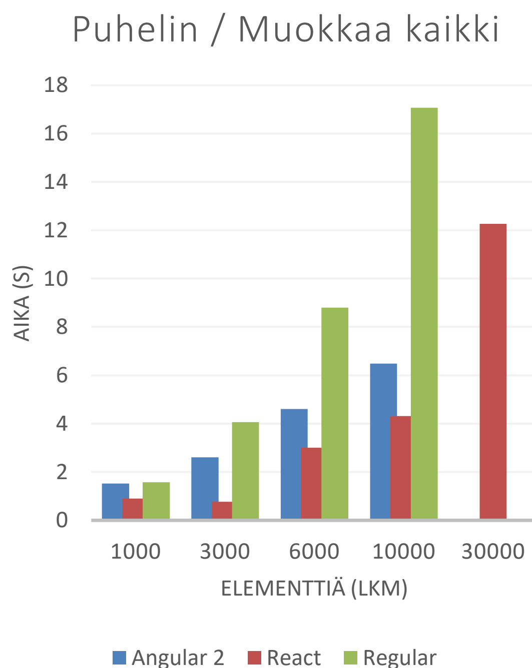
Suoritetuista testeistä kerätyt tiedot koottiin Excel-taulukkoon. Mediaanin sijaan työssä käytetään mittaustulosten keskiarvoa varsinaisena tulosarvona. Erilaisia mittaustuloksista johdettuja arvoja on siis luvun 4 aliluvun 4.3 perusteella 180. Tulokset on visualisoitu 12 kuvaajaan, jotka on edelleen ryhmitelty laitteittain kolmeen eri kuvaan.

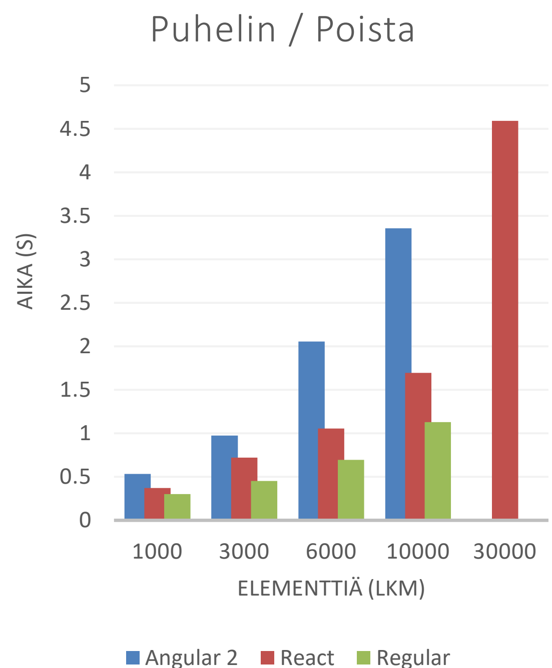
Puhelimella saadut testitulokset esitellään kuvassa 22. Työssä käytetyn puhelimen suorituskykyominaisuudet olivat kaikkein heikoimmat testilaitteiden joukosta ja se näkyy selkeästi mittaustuloksissa. Tätä keskeisempi huomio on kuitenkin Angular-toteutuksen tulosten puuttuminen 30 000 elementin testijoukolla eri operaatioista. Tämä johtuu siitä, että testausta ei pystytty kyseisellä toteutuksella suorittamaan valittua työkalua käyttäen. Chromen DevTool:sin bufferi täyttyi ennen kuin varsinainen testitapahtuma oli valmistunut. Lisäyksessä kesti myös ajallisesti niin pitkään, että se olisi vääristänyt mittaustuloksia liikaa ja tehnyt pienempien testijoukkojen tuloksista lukukelvottomia.

Puhelimella React-toteutus osoittautui tehokkaimmaksi lisäys-, osittainen muokkaus ja kaikkien muokkausoperaatioissa. Kuitenkin poisto-operaatiossa perinteisen toteutuksen tehokkuus oli parempi kuin muilla laitteilla. Lisäysoperaatiossa Angular- ja React-toteutuksissa ei juurikaan ole nopeuseroa, mutta pienellä marginaalilla tavallinen toteutus suoriutui operaatioista nopeammin. Tavallisen ja React-toteutuksen ero korostuu suurimmilla testielementtien määrillä ja testijoukon koon kasvaessa ero näyttää vain kasvavan. Muokkausoperaatioissa taas Angular suoriutui paremmin kuin perinteinen toteutus. Poisto-operaatiosta on jätetty kuvaajasta pois tavallisen toteutuksen pitkä kesto suurimmalla testijoukolla, että kuvaaja säilytetään lukukelpoisena.







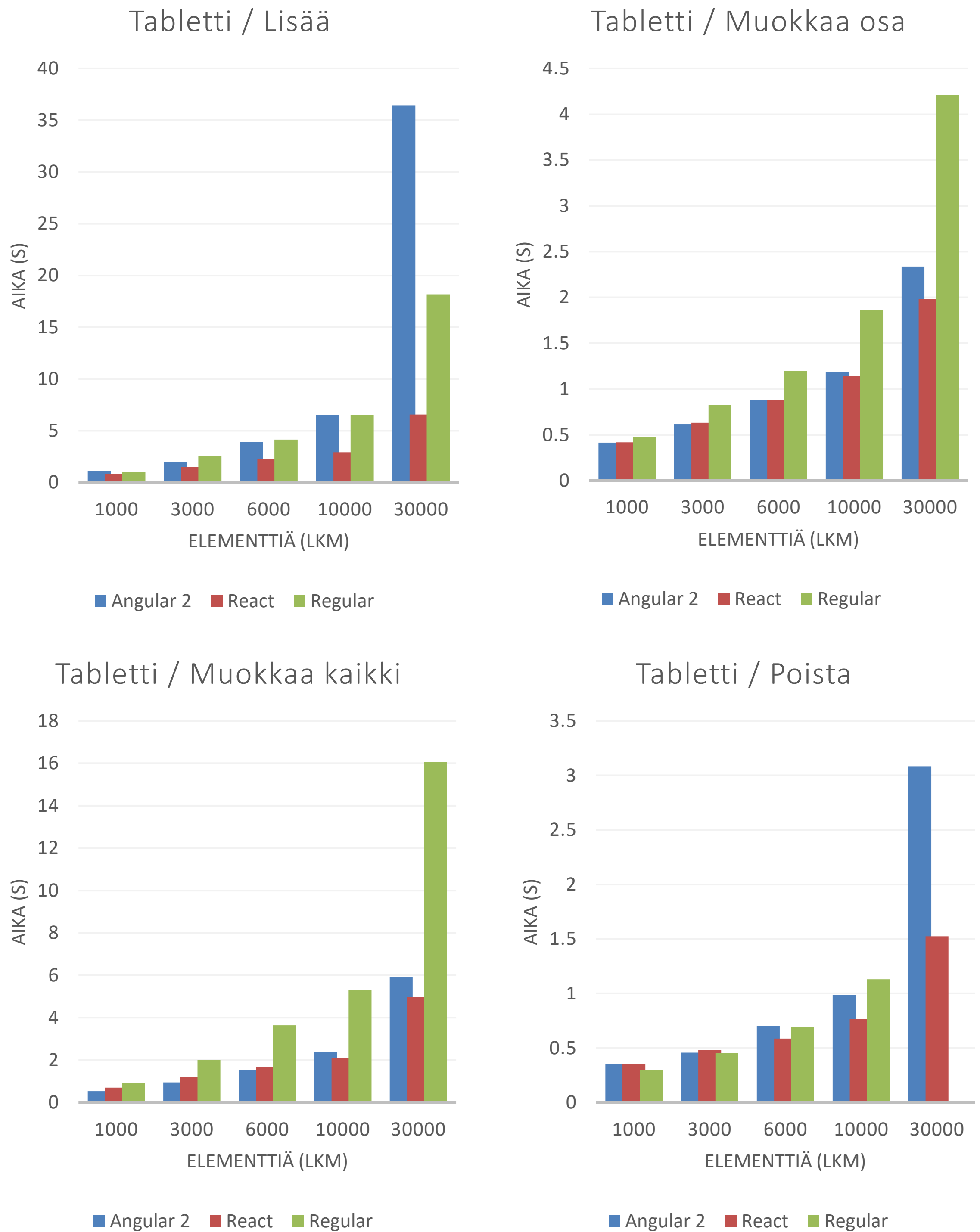


Kuva 22. *Mittaustulokset puhelimella.*

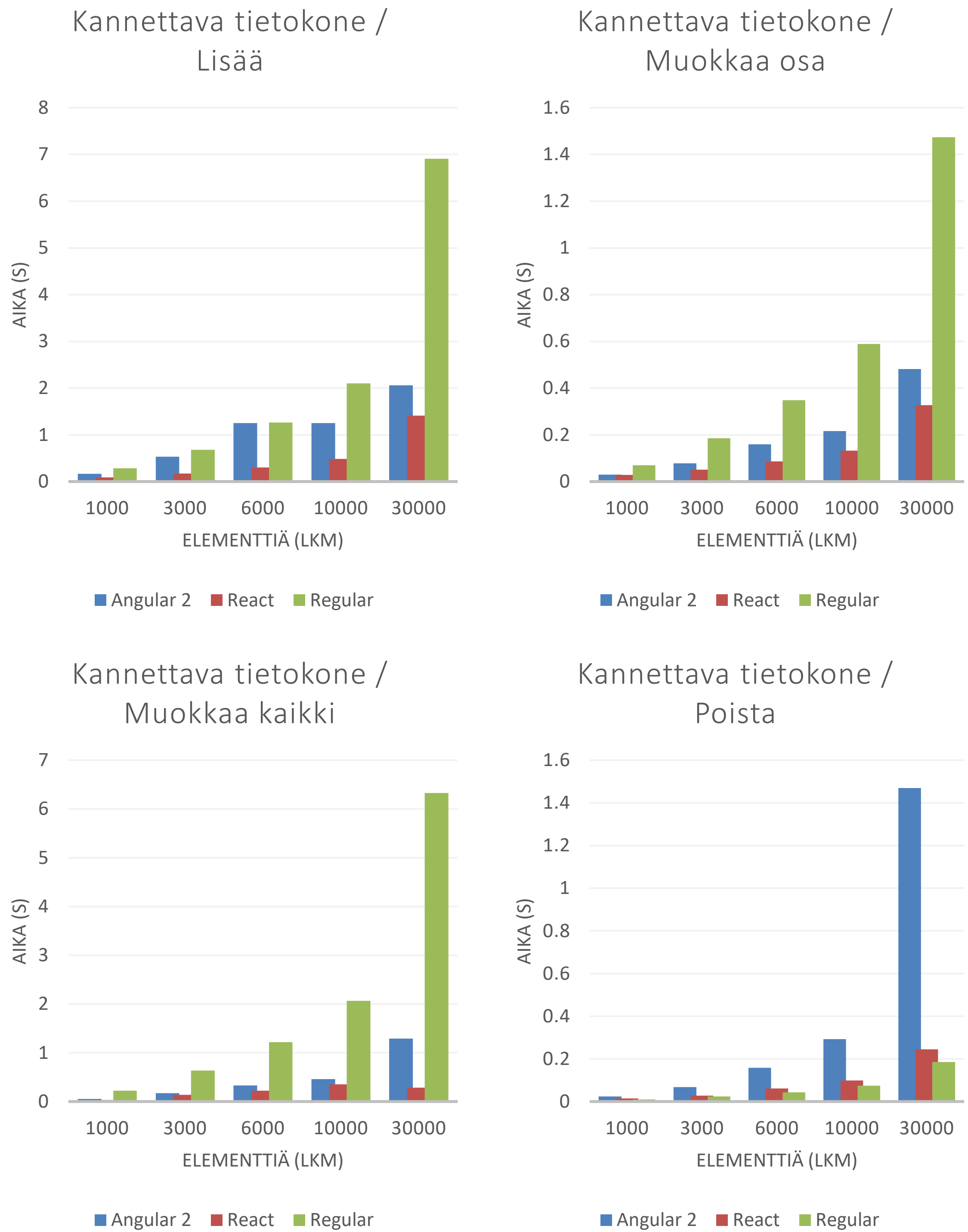
Tabletilla kaikkien testitapausten suorittaminen oli jo mahdollista, mikä johtuu ensisijaisesti paremmasta suorituskyvystä. Tämä näkyy kuvassa 23 myös lyhempinä suoritusaikoina. Suorituskyvyn lisäys näyttää tasanneen lisäysoperaatiossa entisestään tavallisen ja Angular-toteutuksen eroja, mutta React säilyttää asemansa nopeiten suoriutuvana toteutuksena, kuten myös osittaisessa muokkausoperaatiossa. Kuitenkin kaikkien elementtien muokkauksessa Angular-toteutus on hieman nopeampi 10 000 testielementtiin asti. Myös tabletilla sovelluskehystoteutukset suoriutuvat huomattavasti tasaisemmin kuin tavallinen toteutus, jonka heikkous korostuu kaikkien elementtien muokkausoperaatiolla. Se on noin 2,5 kertaa hitaampi kuin Angular- tai React-toteutus. Parempi suorituskyky suhteessa puhelimeen näyttää vaikuttavan myös poisto-operaation nopeuteen. Kaikkein tehokkain toteutus on Angular ja jo 30 000 tuloksen kohdalla tavallisen toteutuksen hitaus korostui niin paljon, että se jätettiin kuvaajasta pois. On yllättävää kuinka paljon tabletin tehokkuus aiheuttaa eroja Angular- ja React-toteutusten välille muokkausoperaatiossa verrattuna puhelimeen. Myös poisto-operaation suhteellinen hitaus tehokkaammalla laitteella käyttäen tavallista toteutusta on yllättävä.

Kannettavalla tietokoneella tehtyjä mittauksia käsitellään kuvassa 24. Suhteessa älypuhelimeen ja tablettiin, kannettava tietokone on huomattavasti muista testattuja mobiililaitteita tehokkaampi, mikä näkyy yleisesti huomattavasti edelleen lyhempinä suoritusaikoina. Tavallisen toteutuksen nopeus lisäys-operaatioissa pysyy Angularin kanssa lähes tasoissa, mutta 30 000 elementin kohdalla tavallisen toteutuksen hitaus korostuu lähes kolme kertaiseksi. Vastaava trendi 30 000 elementin testitapauksissa näkyy myös muokkaus-operaatioissa tavallisen toteutustavan kohdalla. React- ja Angular-toteutuksen hyöty näkyy parhaiten muokkausoperaatioissa ja näistä React suoriutuu kaikissa paitsi poisto-operaatiosta nopeiten. Poisto-operaatiossa hitain on Angular-toteutus ja ero 30 000 testielementin tapauksessa suhteessa muihin toteutuksiin on suuri. Tällöin Angular-toteutus on seitsemän kertaa hitaampi kuin poisto-operaatio on keskimäärin muilla toteutuksilla.

Saadut tulokset ovat linjassa oletuksen kanssa siitä, että DOM-optimoinneista on erityisesti hyötyä heikomman suorituskyvyn laitteilla ja valitsemalla sopivin toteutustapa voidaan saavuttaa parhaimmillaan suuriakin tehokkuushyötyjä. Keskimäärin kaikki toteutukset vaikuttavat vastaavan lineaarisesti testijoukon kasvuun, mutta näistä ainoastaan React pystyy todella vastaamaan testijoukon kasvattamiseen. Myös tavallisen toteutuksen suoritusajan kasvu voidaan arvioida suhteellisen hyvin lineaarisella mallilla. Puhelimen lisäysoperaatiossa Angular-toteutus riittävän isolla joukolla muuttuu hyvin raskaaksi, joka näkyy myös tabletin lisäoperaation huomattavan pitkänä suorituksena.



Kuva 23. *Mittaustulokset tabletilla..*



Kuva 24. *Mittaustulokset kannettavalla tietokoneella.*

6. JOHTOPÄÄTÖKSET

Käyttämällä DOM:ia optimoivaa toteutusta voidaan parhaimmillaan saada suuriakin tehokkuushyötyjä alustasta riippumatta. Hyödyt korostuvat alemman suorituskyvyn laitteissa, joissa pitkä suoritusaika ja raskaat operaatiot kuormittavat jo valmiiksi vähäisiä resursseja. JavaScript-sovelluskehyksiin ja -kirjastoihin perustuvat toteutukset helpottavat kehittäjän työtä abstrahoimalla pois tarpeen suoralla vuorovaikutukselle DOM:in kanssa ja ohjaavat tuottamaan laadukkaampaa ja uudelleenkäytettävämpää koodia. Testisovellusten toteuttamisessa Angular ja React olivat hyvin samanlaiset haastavuudeltaan ja ne tarjoavat useita komponenttipohjaiseen kehitykseen soveltuvia ominaisuuksia. Perinteisessä toteutuksessa sovelluksen jako funktionaalisiin rakenteisiin oli paljon ontuvampi ja esimerkiksi kaiken JavaScript-koodin keskittäminen yhteen tiedostoon oli turhankin helppoa. Käyttöliittymään lisättävät elementit ja niiden kanssa vuorovaikuttaminen toteutettiin tässä tapauksessa käsin, kun taas kirjastopohjaiset toteutustavat yksinkertaistivat elementtien luomista ja käsittelyä. Taulukkoon 2 on kerätty työn aikana kerättyjä huomioita ja tiivistetty tuloksia.

Taulukko 2. *Työn tärkeimmät tulokset ja huomiot.*

	Virtual DOM	**Shadow DOM**	**DOM**
Hyödyt	Nopeus, DOM-vuorovaikutuksen abstrahointi pois kehittäjän vastuulta, suoritusajan ennustettavuus	Toteutuksen kapselointi, selkeämpi CSS, ei komponenttien välistä tyylin vuotamista, hyödyt webkomponenteille	Tarkempi vuorovaikutus DOM:in kanssa, suora näkymän muuttaminen
Ongelmat	Heuristiikkojen aiheuttamat mahdolliset ongelmat suorituskyvyn suhteen	Suorituskykyongelmat suorituskyvyltään heikommilla laitteilla, puutteellinen selaintuki (+mahdolliset Polyfillien käytön aiheuttamat ongelmat)	Spagettikoodi, ei komponenttipohjaista kehitystä, vastuu toteutuksen suorituskyvystä kehittäjällä

Työssä tutkittiin miten kolme erilaista DOM-ratkaisua toimivat niin mobiili- kuin työpöytäjärjestelmissä ja vertailtiin niiden hyötyjä. Shadow DOM ja virtual DOM olisi voitu toteuttaa monella eri tavalla, mutta suosion ja helppouden vuoksi ne toteutettiin käyttäen

React-kirjastoa ja Angular 2 -sovelluskehystä. Esitettyihin tutkimuskysymyksiin onnistuttiin vastaamaan työn aikana. Tuloksista huomataan selkeitä eroja näiden kolmen toteutuksen välillä. Vaikka tulosten perusteella ei voida täysin yleistää miten erilaiset shadow DOM ja virtual DOM toteutukset toimivat ja suoriutuvat, voidaan saavutettuja tuloksia pitää hyvänä suuntaviittana valittaessa mahdollista toteutusratkaisua. Molemmat toteutusratkaisut ovat myös saavuttaneet laajaa suosiota, joten niiden suhteen DOM-optimoinnin tarkastelu oli hyvin mielenkiintoista.

Mittausten ongelmana on niiden toteutus käsityönä. Jos testaus olisi saatu automatisoitua, yksittäistä testitapausta olisi voitu helpommin suorittaa useita kertoja peräkkäin ja näin saada tilastollisesti luotettavampia tuloksia. Työn loppuvaiheessa huomattiin, että yleisimpien selainten toteuttama `getBoundingClientRect`-metodin pohjalta luotu apufunktio viimeiselle piirrettävälle elementille voisi toimia keinona määrittää milloin selain on piirtänyt sisällön ruudulle ("Dan", 2017). Tätä toteutustekniikkaa voitaisiin tutkia osana testien automatisointia, mutta mahdollisesti senkään avulla ei voida määrittää tarkkaa kohtaa, milloin ruudunpäivitysnopeus palautuu normaaliksi. On myös tärkeää huomata, että koska mittaukset on toteutettu vain yhdellä JavaScript-moottorilla eli Chromen V8:lla, niitä ei voida pitää mittarina eri toteutustapojen suorituskyvylle yleisesti. Tuloksista kuitenkin nähdään toteutustapojen suhteellinen suorituskyky suositulla selaimella ja alustan suorituskyvyn vaikutus toteutuksen nopeuteen.

Tulokset ovat linjassa aiemmin esiteltyihin suorituskykytarkasteluihin. Peyrott (2016) tai Krause (2016) eivät tarkastelleet tavallista DOM-toteutusta suhteessa sovelluskirjastojen tai kirjastojen käyttöön, mutta molempien testien tulokset päätyvät Reactin pieneen nopeushyötyyn suhteessa Angular 2:seen. Object Partnersien (2015) tutkimuksesta taas huomataan, että tavallinen toteutus suoriutuu nopeammin kuin React. Todennäköisesti tämä ero johtuu erilaisesta testaustavasta suhteessa tämän työn testeihin. Object Partnersin tutkimuksessa mitataan ajallisesti vain JavaScriptin suoritusta eikä huomioida *repaint-* ja *reflow*-operaatioiden kestoa.

Reactin etuna oli nopeus testitapauksissa, mutta juuri tällaisissa DOM:in muokkausoperaatioissa virtual DOM oletetustikin on vahvimmillaan. React-toteutus myös suoriutui testitilanteista hyvin ennakoitavasti testijoukon kasvaessa. Tästä ennakoitavuudesta on hyötyä luotaessa arviota sovelluksen suorituskyvystä sovellusta kuormitettaessa. Shadow DOM:in etu taas on toteutuksen kapselointi, jossa jokaisen elementin tyylit ja rakenne on eristetty varjorajapinnan avulla. Tästä on hyötyä erityisesti luotaessa uudelleenkäytettäviä komponentteja. Erityisesti web-komponenttien yhteydessä shadow DOM:in käyttö on perusteltua, sillä sen tarjoamat ominaisuudet antavat kehittäjälle mahdollisuuden luoda kontekstista riippumattomia komponentteja. Varjorajapinnasta on kiistämättä hyötyä, mutta

tällaisen abstrahoinnin luominen jokaiselle luotavalle testielementille johtaa React-toteutusta hitaampaan suoritukseen. Natiivi shadow DOM -toteutus Chrome-selaimessa saattaa myös olla vielä joltain osin huonosti optimoitu, mikä voi vaikuttaa toteutuksen toimintaan.

Virtuaalista DOM:ia on ehdotettu ratkaisuksi myös laajojen XML-dokumenttien mallintamiseen, kuten Psailan tutkimuksessa aiheesta havaitaan (2008). Tavallisen DOM:in suorituskyky jää liian tehottomaksi suurilla datamäärillä ja toteutuksen virtualisointi tuo huomattavia suorituskykyhyötyjä. Vuotilainen et al. pohtii monialustaohjelmoinnin ongelmia web-sovelluksen tilan säilyttämisessä (2016). Tähänkin ongelmaan tarjotaan virtual DOM:in pohjalta luotua ratkaisumallia. Myös shadow DOM:ille löytyy mielenkiintoisia sovelluskohteita. De Ryck et al. (2015) kartoittavat shadow DOM:in hyötyä luotaessa turvallisia web-komponentteja, jotka voisivat piilottaa arkaluontoista tietoa väärinkäytön ulottumiin. Tällaisenaan shadow DOM ei tarjoa täydellistä suojaa arkaluontoisen tiedon suojeluun, mutta voisi toimia osana luotaessa turvallisempaa tiedonsiirtoa verkossa.

Työtä voisi jatkaa tutkimalla eri toteutusten suoriutumista eri selaimilla ja myös eri hintaluokan laitteissa. Mittaustuloksista voisi tarkastella tarkemmin Timeline-työkalun tarjoamaa tietoa ja tutkia miten kokonaisaika koostuu eri operaatioista. Samalla olisi hyvä löytää tehokkaampi testaustapa, joka mahdollistaisi automaattisen testauksen, sillä nykyisellä toteutustavalla testaus oli hyvin puuduttavaa ja aikaa vievää.

LÄHTEET

"Dan”, 2017. *How to tell if a DOM element is visible in the current viewport?* ", verkkosivu. Saatavissa (viitattu 8.5.2017): http://stackoverflow.com/a/7557433.

"FakeRainBrigand", 2014. *Stackoverflow: How does one select all child components (not DOM elements) used in a Reactjs component template?*, verkkosivu. Saatavissa (viitattu 15.4.2017): http://stackoverflow.com/a/27561036.

Abramov, D. *,* 2016. *Create Apps with No Configuration*, verkkosivu. Saatavissa (viitattu 14.4.2017): https://facebook.github.io/react/blog/2016/07/22/create-apps-with-no-configuration.html.

Akshat, P. & Nalwaya, A., 2016. *Chapter 1 - Learning the Basics: A Whistle-Stop Tour of Reac.* Apress.

Atkins, T. J. & Etemad, E. J., 2014. *CSS Scoping Module Level 1: W3C First Public Working Draft*, verkkosivu. Saatavissa (viitattu: 28.11.2016): https://www.w3.org/TR/css-scoping-1/.

Bailey, D., 2015. *Building A Component-Based Web UI With Modern JavaScript Frameworks*, verkkosivu. Saatavissa (viitattu 11.4.2017): https://derickbailey.com/2015/08/26/building-a-component-based-web-ui-with-modern-javascript-frameworks/.

Bidelman, E., 2016. *Shadow DOM v1: Self-Contained Web Components*, verkkosivu. Saatavissa (viitattu 6.1.2017): https://developers.google.com/web/fundamentals/getting-started/primers/shadowdom.

Bille, P., 2007. *A Survey on Tree Edit Distance and Related,* Copenhagen NV: The IT University of Copenhagen.

Chedeau, C., 2013. *React's diff algorithm*, verkkosivu. Saatavissa (viitattu 30.1.2017): https://calendar.perfplanet.com/2013/diff/.

De Ryck, P. et al., 2015. Protected Web Components: Hiding Sensitive Information in the Shadows. *IT Professional,* 17(1).

Gal, A., 2015. *Dirty Performance Secrets of HTML5*, O'Reilly, video. Saatavissa (viitattu 30.4.2017): https://youtu.be/t8x40JXUeWA.

Dodson, R., 2013. *Shadow DOM: The Basics*, verkkosivu. Saatavissa (viitattu 14.4.2017): http://robdodson.me/shadow-dom-the-basics/.

ECMA, 2016. *ECMAScript® 2016 Language Specification*, verkkosivu. Saatavissa (viitattu 20.3.2017): http://www.ecma-international.org/ecma-262/7.0/index.html.

Facebook, 2014. *Draft: JSX Specification*, verkkosivu. Saatavissa (viitattu 14.4.2017): https://facebook.github.io/jsx/.

Facebook, 2017. *Lists and Keys*, verkkosivu. Saatavissa (viitattu 15.4.2017): https://facebook.github.io/react/docs/lists-and-keys.html.

Fedosejev, A., 2015. *React.js Essentials.* Birmingham: Packt Publishing Ltd..

Fielding, J., 2014. *Beginning Responsive Web Design with HTML5 and CSS3.* Books24x7, Apress.

Fielding, R., Irvine, U. & J., G., 1999. *Hypertext Transfer Protocol -- HTTP/1.1*, verkkosivu. Saatavissa (viitattu 20.3.2017): https://tools.ietf.org/html/rfc2616#section-1.4.

Flanagan, D., 2011. *JavaScript: The Definitive Guide.* 6th ed. Sebastobol: O'Reilly Media, Inc.

Fullstack.io, 2017. *ng-book 2: TypeScript.* , verkkosivu. Saatavissa (viitattu 14.4.2017): https://www.ng-book.com/2/p/TypeScript/.

GitHub, 2017. *dom-optimization-benchmark*, verkkosivu. Saatavissa (viitattu 21.5.2017): https://github.com/acoustic-/dom-optimization-benchmark.

Google, 2016. *Shadow DOM v1*, verkkosivu. Saatavissa (viitattu 17.1.2017): https://www.chromestatus.com/feature/4667415417847808.

Google, 2017. *CLI QUICKSTART*, verkkosivu. Saatavissa (viitattu 12.4.2017): https://angular.io/docs/ts/latest/cli-quickstart.html.

Greif, S., 2016. *Front-end Frameworks*, verkkosivu. Saatavissa (viitattu 20.3.2017): http://stateofjs.com/2016/frontend/.

Grigorik, I., 2017. *Render-tree Construction, Layout, and Paint*, verkkosivu. Saatavissa (viitattu 4.4.2017): https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction

Jehl, S., 2014. *Responsible Responsive Design.* New York: A Book Apart.

King, A. B., 2008. *Website Optimization: Speed, Search Engine & Conversion Rate Secrets.* 1st ed. Sabestopol, O'Reilly Media.

Kinlan, P., 2014. *What do people want from a news experience?*, verkkosivu. Saatavissa (viitattu 29.3.2017): https://paul.kinlan.me/what-news-readers-want/.

Krause, S., 2016. *JS web frameworks benchmark – Round 4*, verkkosivu. Saatavissa (viitattu 12.4.2017): http://www.stefankrause.net/wp/?p=316.

Lewis, P., 2017. *Rendering Performance*, verkkosivu. Saatavissa (viitattu 19.3.2017): https://developers.google.com/web/fundamentals/performance/rendering/.

Lowy, J., 2005. *Component-Oriented Versus Object-Oriented Programming*, verkkosivu. Saatavissa (viitattu 11.4.2017): https://www.safaribooksonline.com/library/view/programming-net-components/0596102070/ch01s02.html.

Marcotte, E., 2010. *Responsive Web Design*, verkkosivu. Saatavissa (viitattu 16.2.2017): http://alistapart.com/article/responsive-web-design.

MDN, 2015. *Frame rate*, verkkosivu. Saatavissa (viitattu 1.4.2017): https://developer.mozilla.org/en-US/docs/Tools/Performance/Frame_rate.

MDN, 2016a. *HTML5*, verkkosivu. Saatavissa (viitattu 21.3.2017): https://developer.mozilla.org/en/docs/Web/Guide/HTML/HTML5.

MDN, 2016b. *Document Object Model (DOM)*, verkkosivu. Saatavissa (viitattu 25.3.2017): https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

MDN, 2017. *<template*, verkkosivu. Saatavissa (viitattu 15.4.2017): https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template.

Meier, R., 2012. *Professional Android 4 Application Development.* Indianapolis (Indiana): John Wiley & Sons.

Microsoft, 2017. *Key Principles of Software Architecture*, verkkosivu. Saatavissa (viitattu 1.4.2017): https://msdn.microsoft.com/en-us/library/ee658124.aspx.

Minnick, C., 2016. *The Real Benefits of the Virtual DOM in React.js*, verkkosivu. Saatavissa (viitattu 28.11.2016): https://www.accelebrate.com/blog/the-real-benefits-of-the-virtual-dom-in-react-js/.

Mobidev, 2014. *How To Optimize The Performance Of PhoneGap Apps*, verkkosivu. Saatavissa (viitattu 28.4.2017): https://mobidev.biz/blog/how_to_optimize_the_performance_of_phonegap_apps.

Motto, T., 2016. *Emulated or Native Shadow DOM in Angular 2 with ViewEncapsulation*, verkkosivu. Saatavissa (viitattu 28.11.2016): https://toddmotto.com/emulated-native-shadow-dom-angular-2-view-encapsulation.

Myers, C., 2015. *Responsive Web Design Patterns.* Packt Publishing Ltd.

Netmarketshare, 2017. *Desktop Browser Market Share*, verkkosivu. Saatavissa (viitattu 29.4.2017): https://www.netmarketshare.com/browser-market-share.aspx.

Nielsen, J., 2012. *Repurposing vs. Optimized Design*, verkkosivu. Saatavissa (viitattu 16.2.2017): https://www.nngroup.com/articles/repurposing-vs-optimized-design/.

Object Partners, 2015. *Comparing React.js performance vs. native DOM*, verkkosivu. Saatavissa (viitattu 12.4.2017): https://objectpartners.com/2015/11/19/comparing-react-js-performance-vs-native-dom/.

OpenSignal, 2013. *Android Fragmentation Visualized*, verkkosivu. Saatavissa (viitattu 17.2.2017): https://opensignal.com/reports/fragmentation-2013/.

Peyrott, S., 2016. *More Benchmarks: Virtual DOM vs Angular 1 & 2 vs Others*, verkkosivu. Saatavissa (viitattu 12.4.2017): https://auth0.com/blog/updated-and-improved-more-benchmarks-virtual-dom-vs-angular-12-vs-mithril-js-vs-the-rest/.

Protalinski, E., 2016. *Venture Beat: Google will start ranking 'mobile-friendly' sites even higher in May*, verkkosivu. Saatavissa (viitattu 16.2.2017): http://venturebeat.com/2016/03/16/google-will-start-ranking-mobile-friendly-sites-even-higher-in-may/.

Psaila, G., 2008. *Virtual DOM: an Efficient Virtual Memory Representation for Large XML Document,* Turrin.

Robie, J., 1998. *What is the Document Object Model?*, verkkosivu. Saatavissa (viitattu 28.11.2016): https://www.w3.org/TR/WD-DOM/introduction.html.

Shukla, M. K., 2014. *Benefits and drawbacks of using client side frameworks*, verkkosivu. Saatavissa (viitattu 21.3.2017): http://stackoverflow.com/a/25152609.

Souders, S., 2009. *Performance Impact of CSS Selectors*, verkkosivu. Saatavissa (viitattu 16.1.2017): https://www.stevesouders.com/blog/2009/03/10/performance-impact-of-css-selectors/.

StatsCounter, 2016. *Mobile and tablet internet usage exceeds desktop for first time worldwide*, verkkosivu. Saatavissa (viitattu 16.2.2017): http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide.

StatsCounter, 2017. *Browser Market Share in Finland*, verkkosivu. Saatavissa (viitattu 29.4.2017): http://gs.statcounter.com/browser-market-share/all/finland.

Stepp, M., Miller, J. & Kirst, V., 2012. *Web Programming Step by Step.* 2nd ed, Step by Step Publishing.

Syromiatnikov, A. & Weyns, D., 2014. A Journey Through the Land of Model-View-* Design Patterns. *2014 IEEE/IFIP Conference on Software Architecture*, pp. 1-5.

Tilastokeskus, 2015. *Väestön tieto- ja viestintätekniikan käyttö 2015,* Edita Publishing Oy.

W3C, 2005. *Document Object Model (DOM*, verkkosivu. Saatavissa (viitattu 25.3.2017): https://www.w3.org/DOM/.

W3C, 2015. *HTML5*, verkkosivu. Saatavissa (viitattu 25.3.2017): https://www.w3.org/TR/html5/syntax.html#parsing.

W3C, 2016a. *Media Queries Level 4*, verkkosivu. Saatavissa (viitattu 17.2.2017): https://www.w3.org/TR/mediaqueries-4/.

W3C, 2016b. *CSS Device Adaptation Module Level 1*, verkkosivu. Saatavissa (viitattu 17.2.2017): https://www.w3.org/TR/css-device-adapt/#at-ruledef-viewport.

W3Counter, 2017. *Browser & Platform Market Share: March 2017*, verkkosivu. Saatavissa (viitattu 29.4.2017): https://www.w3counter.com/globalstats.php.

Wasson, M., 2013. *ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET*, verkkosivu. Saatavissa (viitattu 21.3.2017): https://msdn.microsoft.com/en-us/magazine/dn463786.aspx.

WebComponents.org, 2017. *Introduction*, verkkosivu. Saatavissa (viitattu 11.4.2017): https://www.webcomponents.org/introduction.

Wodehouse, C., 2015. *The Role of a Front-End Web Developer: Creating User Experience & Interactivity*, verkkosivu. Saatavissa (viitattu 20.3.2017): https://www.upwork.com/hiring/development/front-end-developer/.

World Wide Web Foundation, 2017. *History of the Web*, verkkosivu. Saatavissa (viitattu 22.3.2017): http://webfoundation.org/about/vision/history-of-the-web/.

Vuotilainen, J.-P., Mikkonen, T. & Systä, K., 2016. *Synchronizing Application State Using Virtual DOM Trees,* Cham, Springer.

Zakas, N. C., 2012. *Professional JavaScript for Web Developers.* 3rd ed. s.l.:Wrox Press.

LIITE 1: YHTEISET FUNKTIOT

Yhteiset funktiot

```
function randomContent() {
  // Luodaan yksinkertainen satunnaisgeneraattori: sen palauttaa
  // stringLength muuttujan mukaisen lukumäärän satunnaisia merkkejä
  // määritellystä merkkijonosta. Merkkijono koostuu kaikista englannin
  // isoista ja pienistä kirjaimista sekä numeroista.
  var chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            + "abcdefghijklmnopqrstuvwxyz0123456789";
  var stringLength = 1;
  var content = "";
  for (var i = 0; i < stringLength; ++i) {
    // Lisää content-merkkijonoon yksi merkki satunnaisesta sijainnista
    content += chars.charAt(Math.floor(Math.random() * chars.length));
  }
  return content;
}
```

```
function getRandomSize() {
  // Palauttaa merkkijonomuotoisena elementin sivun pituuden.
  // Palautettava leveys on väliltä 14–20. Loppuun lisätään
  // ”px”, jotta tulosta voidaan käyttää suoraan CSS:n kanssa.
  return Math.floor((Math.random() * 6) + 1) + 13 + "px";
}
function generateColor() {
  return generateRandomColor(256, 256, 256);
}

function generateRandomColor(inRed, inGreen, inBlue) {
  // Perustuu allaoleviin lähteisiin:
  // http://stackoverflow.com/a/43235, http://stackoverflow.com/a/5624139

  var red = Math.floor(Math.random() * 256);
  var green = Math.floor(Math.random() * 256);
  var blue = Math.floor(Math.random() * 256);

  // Sekoita syötetty ja satunnainen väri
  if (inRed && inGreen && inBlue) {
    red = Math.floor((red + inRed) / 2);
    green = Math.floor((green + inGreen) / 2);
    blue = Math.floor((blue + inBlue) / 2);
  }
  // Palauta HEX-muotoinen merkkijonoesitys väristä
  return "#" + ((1 << 24) + (red << 16) + (green << 8) +
blue).toString(16).slice(1);
}
```