Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma

Kandidaatintyö

Topias Mikonsaari

YHDEN SIVUN VERKKOSOVELLUKSEN KOONTIVAIHEEN MENETELMÄT SIVULATAUKSEN NOPEUTTAMISEKSI

Työn tarkastaja: Tutkijatohtori Ari Happonen

Työn ohjaaja: Tutkijatohtori Ari Happonen

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma

Topias Mikonsaari

Yhden sivun verkkosovelluksen koontivaiheen menetelmät sivulatauksen nopeuttamiseksi

Kandidaatintyö

2018

40 sivua, 13 kuvaa, 1 taulukko, 1 liite

Työn tarkastaja: Tutkijatohtori Ari Happonen

Hakusanat: yhden sivun verkkosovellus, sivulataus, latausaika, koonti
Keywords: single-page application, page load, load time, build

Hidas sivulataus on yksi tunnetuista yhden sivun verkkosovelluksen haasteista. Tässä työssä tutkitaan yhden sivun verkkosovelluksen koontivaiheen tekniikoita sivulatauksen nopeuttamiseksi. Esitellyt menetelmät pyrkivät vaikuttamaan kolmeen työssä tunnistettuun sivulatausta haittaavaan tekijään: ladattavien resurssien koko ja lukumäärä sekä sovelluksen suorituskyky. Työ on vain yleiskatsaus tunnetuista menetelmistä eikä yksityiskohtaisia kuvauksia tai käyttöohjeita anneta. Työn johtopäätös on, että yhden sivun verkkosovelluksen koonnista voi olla apua sivulatauksen nopeuttamiseksi. Monet menetelmät ovat kuitenkin vaihtokauppoja vaikuttavien tekijöiden välillä, joten sovelluskohtainen testaaminen on tärkeää menetelmien todellisen hyödyn määrittämiseksi.

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science

Topias Mikonsaari

Build-time techniques for speeding up single-page application page load

Bachelor's Thesis

40 pages, 13 figures, 1 table, 1 appendix

Examiners: D.Sc. (Tech.) Ari Happonen

Keywords: single-page application, page load, load time, build

Slow page load time is a known challenge in single-page applications. In this paper, build-time techniques for speeding up single-page application page load are explored. The presented techniques aim to improve three identified factors hindering page load time: the size and the number of downloaded resources, and application performance. This study is only an overview of known methods and neither detailed descriptions nor instructions for use are given. The conclusion of this study is that using build tools can help improve single-page application page load time. However, many of the methods are trade-offs between the hindering factors, so application specific benchmarking is essential to determine the true benefits of the methods.

ALKUSANAT

Haluan kiittää ohjaajaani Ari Happosta hyvistä neuvoista ja kärsivällisyydestä työn aiheen muotoutuessa nykyiseen tilaansa. Kiitän myös kaikkia läheisiäni, jotka tukivat minua työn kirjoittamisen aikana.

Lappeenrannassa 30.9.2018

Topias Mikonsaari

SISÄLLYSLUETTELO

1 JOHDANTO 4

1.1 Työn tausta 4

1.2 Tavoitteet ja rajaukset 5

1.3 Työn rakenne 5

2 YHDEN SIVUN VERKKOSOVELLUKSET JA KOONTI 7

2.1 Yhden sivun verkkosovelluksen hyödyt 7

2.2 Yhden sivun verkkosovelluksen tunnetut haasteet 8

2.3 Yhden sivun verkkosovelluksen koonti 8

3 VERKKOSOVELLUKSEN SIVULATAUS JA KOONNIN VAIKUTUSMAHDOLLISUUDET 10

3.1 Verkkosovelluksen sivulataus 10

3.2 Koontivaiheen vaikutusmahdollisuudet sivulatauksen nopeuteen 12

3.2.1 Ladattavien resurssien koko 12

3.2.2 Ladattavien resurssien lukumäärä 13

3.2.3 Sovelluksen suorituskyky 14

3.3 Optimoinnin aloittaminen 15

4 KOONTIVAIHEEN TEKNIIKAT SIVULATAUKSEN NOPEUTTAMISEKSI 17

4.1 Resurssien paketointi, jakaminen ja viivytetty lataaminen 17

4.1.1 Reittikohtainen jako 19

4.1.2 Toiminnallinen jako 20

4.2 Koodin tiivistäminen 20

4.2.1 Tarpeettomien merkkien poisto 21

4.2.2 Nimien lyhentäminen 21

4.2.3 Tosiarvojen muunto 21

4.2.4 Muuttujaesittelyiden yhdistäminen 22

4.2.5 Tiivistyksen tuomat säästöt 22

4.3 Kuolleen koodin eliminointi 23

4.3.1 Ympäristökohtaiset käskyt 23

4.3.2 Puunravistelu 24

4.4 Esilaskenta 25
4.4.1 Vakioiden levitys 25
4.4.2 Esirenderointi 25
4.5 Templaattien kääntäminen 27
4.6 Riville sijoittaminen 27
4.7 Kuvien optimointi 28
4.7.1 Vektorikuvat 28
4.7.2 Rasterikuvat 29
4.8 Menetelmien saatavuus ja käyttö 29
5 POHDINTA, TULEVAISUUS JA YHTEENVETO 30

LÄHTEET

LIITTEET
Liite 1. Työkalut

SYMBOLI- JA LYHENNELUETTELO

AJAX	Asynchronous Javascript and XML
CSS	Cascading Style Sheets
CSSOM	Cascading Style Sheets Object Model
DOM	Document Object Model
ES	ECMAScript
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IPv4	Internet Protocol version 4
JSON	JavaScript Object Notation
Less	Leaner style sheets
Sass	Syntactically awesome style sheets
SPA	Single-page Application
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
WWW	World Wide Web
XML	Extensible Markup Language

1 JOHDANTO

Tässä kandidaatintyössä käsitellään yhden sivun verkkosovelluksen (eng. single-page application, SPA) koontivaiheen tekniikoita sivulataukseen kuluvan ajan pienentämiseksi. Tässä luvussa työ esitellään selvittämällä sen tausta, tavoitteet ja rajaukset sekä rakenne.

1.1 Työn tausta

WWW (World Wide Web) on kehittynyt sen lähes 30-vuotisen elinkaaren aikana verrattain yksinkertaisesta dokumenttiverkosta mittavaksi sovellusten ja sisällön jakeluympäristöksi (Taivalsaari & Mikkonen 2011, 170). Verkkosivujen käyttökokemusta on pyritty parantamaan tänä aikana monilla eri tavoilla, joista yksi on laatia paikallista sovellusta muistuttava yhden sivun verkkosovellus. Yhden sivun verkkosovellus poikkeaa tavanomaisesta monisivuisesta verkkosivustosta siten, että se generoi sivun sisällön verkkoselaimessa eikä lataudu uudelleen käytön aikana (Shahzad 2017, 410-411). Malli voi parantaa sovelluksen vuorovaikutuksellisuutta ja nopeuttaa sovelluksessa navigointia, sillä vain sivulla vaihtuneet osat täytyy päivittää. Monet suuretkin sivut ovat nykyään yhden sivun verkkosovelluksia, kuten Microsoftin sähköpostisovellus Outlook Web App ja Googlen videoiden jakopalvelu YouTube. Mallia voidaan käyttää sekä mobiili- että työpöytälaitteille suunnatuissa verkkosovelluksissa, joissa vaatimuksena on paikallisen sovelluksen tasoinen vuorovaikutuksellisuus.

Yksi merkittävä yhden sivun verkkosovelluksen haaste on ensimmäiseen sivulataukseen kuluva aika, mikä voi mallin toimintatavasta johtuen kasvaa helposti suureksi. Sivulataukseen kuluva aika on tärkeä käyttökokemukseen vaikuttava tekijä (Nah 2004, 153; Galletta ym. 2004, 17-20), jolla on myös muita negatiivisia vaikutuksia, kuten mahdollisesti heikompi sijoittuminen (ainakin Googlen) hakukoneissa (Singhal & Cutts 2010, Wang & Phan 2018.) Kelvollisen nopeuden saavuttaminen muuttuu myös aina haastavammaksi mitä laajempia markkinoita tavoittelee, sillä esimerkiksi verkkoteknologioiden kehitys ja käyttöönotto ei näy tasaisesti ympäri maailman: Akamain raportin (2017, 12-13) mukaan Suomen keskimääräinen työpöytälaitteiden IPv4 (Internet Protocol version 4) yhteysnopeus oli 20,5 Mbps vuoden 2017 ensimmäisellä neljänneksellä, kun globaali keskiarvo oli 7.2 Mbps.

Sovelluksen latausajan optimointi koontivaiheessa on houkuttelevaa, sillä se on sovelluksen julkaisua edeltävä vaihe eikä lisää ajonaikaisia kustannuksia. Koontityökalujen avulla tekniikat saattavat olla automatisoitavissa eivätkä siten aiheuta suurempaa rasitetta kehittäjäkokemukselle. JavaScript ekosysteemissä erilaisia työkaluja on saatavilla runsaasti ja niitä tulee jatkuvasti lisää, joten on hyödyllistä tietää mitä tekniikoita on olemassa ja mihin ne perustuvat, jotta työkalujen vertailu on mahdollista.

1.2 Tavoitteet ja rajaukset

Tämän kandidaatintyön tavoitteena on selvittää, mitä tekniikoita JavaScriptillä toteutetun yhden sivun verkkosovelluksen koontivaiheessa voi hyödyntää sovelluksen sivulataukseen kuluvan ajan pienentämiseksi. Työn pyrkimys on tuoda esiin verkkosovelluksen sivulatauksen haasteita ja niiden koontivaiheen ratkaisuja. Haasteiden monimuotoisuus tuodaan esiin tutustumalla laajaan valikoimaan erilaisia tekniikoita. Laajuuden mahdollistamiseksi työ tarjoaa ainoastaan yleisluontoisen käsityksen vallitsevista menetelmistä, eikä tekniikoiden yksityiskohdista anneta kattavia selityksiä tai työkaluista käyttöohjeita. Rajauksen taustalla on ajatus, että vaikka työkalujen kehitys on nopeaa ja elinkaari toisinaan lyhyt, ovat tekniikoiden ydinideat ja motiivit pitkäikäisempiä.

Työtä on tarkoitus pystyä käyttämään lukijan oman sovelluskehityksen tukena: työn avulla voi arvioida menetelmien tarpeellisuutta omassa sovelluksessa sekä vertailla työkaluja niiden tarjoamien ominaisuuksien perusteella. Työssä ei kuitenkaan tulla suorittamaan kattavia testejä tekniikoiden todellisesta hyödystä, sillä ne ovat usein sovelluskohtaisia ja yleistyksiä tekniikoiden vaikutuksista on hankala tehdä. Menetelmien todellisen hyödyn mittaaminen tulee siis toteuttaa itse.

1.3 Työn rakenne

Työn tutkimusosuus rakentuu kahdesta osasta. Ensimmäisessä osassa (luku 2) tutustutaan yhden sivun verkkosovelluksen toimintamalliin ja koontiin. Toisessa osassa (luku 3 ja 4) tarkastellaan sivulatauksen vaiheita, koonnin vaikutusmahdollisuuksia ja olemassa olevia menetelmiä sivulatauksen nopeuttamiseksi. Tarkempi lukukohtainen jako on esitetty taulukossa 1.

Luku 2. Yhden sivun verkkosovellukset ja koonti	Esittelee yhden sivun verkkosovellukset ja koontivaiheen roolin.
Luku 3. Verkkosovelluksen sivulataus ja koontivaiheen vaikutusmahdollisuudet	Esittelee miten sivulatauksen voi määritellä ja tutkitaan miten koontivaiheella voi sivulataukseen vaikuttaa.
Luku 4. Koontivaiheen tekniikat sivulatauksen nopeuttamiseksi	Esittelee tekniikoita yhden sivun verkkosovelluksen sivulatauksen nopeuttamiseksi ja kuinka niitä voi toteuttaa sovelluksen koontivaiheessa.
Luku 5. Pohdinta, tulevaisuus ja yhteenveto	Tarkastellaan luvussa 3 ja 4 esitettyjä tuloksia ja havaintoja. Summataan saavutetut tulokset ja niistä tehdyt johtopäätökset. Pohditaan tutkimuksen onnistumista ja jatkokehityksen mahdollisuuksia.

Taulukko 1. Työn rakenne

2 YHDEN SIVUN VERKKOSOVELLUKSET JA KOONTI

WWW-sovelluksien vuorovaikutus käyttäjän kanssa on pitkään perustunut monisivuiseen malliin, jossa jokainen palvelinpyyntö johtaa koko käyttöliittymän päivittämiseen. AJAX-tekniikoiden (Asynchronous Javascript And XML) myötä vastausta klassisen monisivuisen verkkosovelluksen heikkoon vuorovaikutuksellisuuteen on yritetty löytää yhden sivun verkkosovelluksista. Tässä mallissa sovelluksen käyttöliittymä rakentuu yksittäisistä komponenteista, joita voidaan päivittää tai vaihtaa itsenäisesti manipuloimalla DOMia (Document Object Model) JavaScriptillä ja hakemalla tarvittava data asynkronisesti palvelimelta. Yhden sivun verkkosovelluksessa navigointi perustuu näkymien muuttamiseen sovelluksen tilan perusteella (Meshab & Deursen 2007, 181-190). Monisivuisessa verkkosovelluksessa toiminnot, kuten linkin seuraaminen ja lomakkeen lähetys, voivat aiheuttaa selainkontekstin navigoitumisen uuteen resurssiin ja uuden dokumenttiobjektin rakentamiseen (WHATWG 2018, 824-825, 832), toisin kuin yhden sivun verkkosovelluksessa, jossa dokumentti ladataan ja alustetaan vain kerran. Yhden sivun verkkosovelluksia voi siis ajatella palvelimelta ladattavina painavina asiakaspäätteinä (eng. fat client) (Mikowski & Powell 2013, 4).

2.1 Yhden sivun verkkosovelluksen hyödyt

Yhden sivun verkkosovelluksen toimintamalli on samankaltainen paikallisen sovelluksen kanssa ja siksi sen hyödyt ovat myös samanlaiset. Etuja monisivuiseen malliin verrattuna on esitetty Mikowskin ja Powellin (2013, 20-21) kokoamassa listassa:

- Renderoituu nopeasti kuten paikallinen sovellus, kun vain käyttöliittymän päivittyneet osat täytyy piirtää uudelleen ja vältetään monisivuisen verkkosivun päivityksen yhteydessä havaittava tauko.
- Vastausaika toimintoihin on nopea, sillä suurin osa toimintalogiikasta on sisällytetty sovellukseen ja siten päätöksenteko voidaan hoitaa paikallisesti.
- Palvelimen kanssa kommunikoidessa voidaan näyttää indikaattori sovelluksen lataustilasta.

2.2 Yhden sivun verkkosovelluksen tunnetut haasteet

Yhden sivun verkkosovelluksiin liittyy myös haittapuolia suorituskyvyn näkökulmasta. Koska yhden sivun verkkosovellukset vaativat ylimääräisen JavaScriptin lähettämisen sivun päivittämiseen, generointiin ja muun tarvittavan toiminnallisuuden saamiseksi, on sivun ensimmäisen latausaika yleensä pidempi. Jos lataukseen sisällytetään toiminnallisuuksia, joita ei välittömästi – tai ollenkaan – tarvita, kasvattaa se tarpeettomasti ensimmäisen sivulatauksen kokoa ja sivun muodostamiseen kuluvaa aikaa. (Souders 2008.) Mallin käyttö on siis eräänlainen vaihtokauppa: se mahdollistaa pidempiaikaisessa käytössä nopeasti reagoivan sovelluksen ensimmäisen sivulatauksen kustannuksella.

Seuraavat asiat ovat myös yhden sivun verkkosovelluksien käyttöönotossa huomioitavia asioita, mutta ne eivät kuulu tämän työn aihealueeseen, eikä niitä tässä työssä seuraavia mainintoja enempää käsitellä:

- Dynaamisesti vaihtuva sisältö voi heikentää sivuston saavutettavuutta (eng. accessibility), minkä vuoksi sovellusta rakentaessa tulee noudattaa tiettyjä sääntöjä apuvälineiden toimivuuden varmistamiseksi (W3C 2017b).
- Sivun renderointi vaatii JavaScriptin suorittamisen, joten verkkosivuja lataavat ohjelmat (esim. jotkut hakurobotit), jotka odottavat kokonaista dokumenttia pyynnön vastauksena, eivät sivun sisältöä näe sen muodostuessa vasta latauksen jälkeen asiakkaan puolella.

2.3 Yhden sivun verkkosovelluksen koonti

Ohjelman koonnilla tarkoitetaan prosessia, jossa lähdekoodi muunnetaan toimivaksi ohjelmaksi. Prosessiin voi kuulua muun muassa testausta, erilaisten metriikoiden keräämistä ja ohjelman käyttöönotto. (Kawalerowicz 2013, 1.) Tässä työssä koontiprosessista keskitytään lähinnä lähdekoodin kääntämiseen ja ohjelman tarvitsemien resurssien toimittamiseen käyttöönottovalmiiksi.

JavaScriptillä tehtyjen yhden sivun verkkosovelluksien koonti ei ole välttämätöntä, mutta siitä voi olla monella tavalla hyötyä. JavaScript saa nopeaan tahtiin uusia ominaisuuksia, mutta verkkoselaintuki ei aina pysy ajan tasalla eikä uusia ominaisuuksia välttämättä siirretä

vanhempiin versioihin. Koonnin aikana voidaan tunnistaa huonosti tuetut toiminnallisuudet ja korvata ne vastaavilla, mutta paremmin tuetuilla versioilla (eng. polyfill). Lisäksi koonti mahdollistaa eri tyylilehtikielien, kuten Sass (Syntactically awesome style sheets) tai Less (Leaner style sheets), käännön verkkoselaimen tukemaksi CSS:ksi (Cascading Style Sheets). Koontivaiheeseen sisällytetään usein myös erilaisien virheiden tunnistaminen ja varoitusten esittäminen. Koonnin hyödyllisyys tulee esiin tietysti myös tässä työssä sivulatauksen nopeuttamisen merkeissä.

3 VERKKOSOVELLUKSEN SIVULATAUS JA KOONNIN VAIKUTUSMAHDOLLISUUDET

Tässä luvussa avataan sivulatauksen määritelmää ja tunnistetaan koontivaiheen vaikutusmahdollisuudet sivulatauksen nopeuttamiseksi.

3.1 Verkkosovelluksen sivulataus

Sivulataukseen kuuluu monta tärkeää vaihetta. Verkkoselaimissa dokumentin valmiustilaa voi tutkia dokumentin *readyState*-muuttujan avulla. HTML-spesifikaation (Hypertext Markup Language) mukaan muuttuja voi olla kolmessa eri tilassa (WHATWG 2018, 113):

1. latautumassa (loading), kun dokumenttia vielä ladataan
2. interaktiivinen (interactive), kun dokumentti on ladattu ja jäsennetty, mutta odottaa vielä aliresursseja
3. valmis (complete), kun aliresurssien lataus on päättynyt.

Dokumentin tilaa voidaan seurata myös kahden tapahtuman avulla: *DOMContentLoaded* aktivoidaan dokumenttioliossa, kun dokumentti on siirtynyt interaktiiviseksi, ja *load* aktivoidaan ikkunaoliossa, kun dokumentin resurssit ovat latautuneet (WHATWG 2018, 1214-1215).

Edellä mainitut tilat eivät välttämättä vastaa käyttäjän kokemusta sivun latauksesta. Käyttäjälähtöisempiä latauksen vaiheita voi mitata esimerkiksi Googlen kehittämällä Lighthouse-työkalulla. Työkalussa käyttäjän kokemia latauksen vaiheita tunnistetaan kolme (Lighthouse 2018):

1. Ensimmäinen merkityksellinen maalaus: tarkoittaa hetkeä, jolloin suurin näytöllä näkyvä sijoittelumuutos on tapahtunut ja fontit ovat latautuneet, eli kun käyttäjä tuntee ensisijaisen sisällön olevan näkyvissä.
2. Ensimmäisen kerran vuorovaikutettavissa: tarkoittaa hetkeä, jolloin suurin osa elementeistä on vuorovaikutettavissa ja sivu keskimäärin vastaa käyttäjän syötteeseen kohtuullisessa ajassa.

3. Konsistentisti vuorovaikutettavissa: tarkoittaa hetkeä, jolloin sivu on kokonaan vuorovaikutettavissa, eli kun kontrolli palautuu pääsäikeeseen (eng. main thread) vähintään kerran 50 ms välein ja verkko on toimeton.

Kuvassa 1 nähdään sivun https://m.youtube.com latauksen vaiheet. Ensimmäiseen merkitykselliseen maalaukseen menee 2,77 sekuntia ja vaiheet 2 sekä 3 toteutuvat samanaikaisesti 6,73 sekunnin kohdalla. Piirtymisen jälkeen kuluu siis vielä 3,96 sekuntia ennen kuin käyttäjä pystyy kunnolla käsittelemään sivun elementtejä. Kuvassa 2 on samasta latauksesta piirretty aikajana. Aikajanassa sininen pystyviiva merkitsee *DOMContentLoaded*- ja punainen viiva *load*-tapahtumaa. *DOMContentLoaded* tapahtui ennen ensimmäistä merkityksellistä maalausta noin 2,7 sekunnin kohdalla ja *load* tapahtui noin 3,75 sekunnin kohdalla.

Kuva 1. Sivulatauksen vaiheet https://m.youtube.com verkkosivustossa. Generoitu Lighthouse-työkalulla mobiililaitteen emulointi päällä ja suorituskykyrajoituksilla.

Kuva 2. Aikajana https://m.youtube.com sivulatauksesta. Generoitu Chrome kehittäjätyökalulla.

Tässä työssä sivulatausta ajatellaan edellä esitettynä monivaiheisena ilmiönä, eli verkkosovelluksen sisällön näyttämisen lisäksi otetaan huomioon myös vuorovaikutuksellisuus.

3.2 Koontivaiheen vaikutusmahdollisuudet sivulatauksen nopeuteen

Seuraavissa alaluvuissa esitellään ne kolme työssä tunnistettua sivulataukseen vaikuttavaa tekijää, joihin luvussa 4 esitellyillä menetelmillä voi pyrkiä vaikuttamaan.

3.2.1 Ladattavien resurssien koko

Verkkosivujen koko kasvaa vuosi vuodelta. Tämän vuoden alusta mitattuna verkkosivujen mediaanikoko on ollut pöytäkoneversiona 1502,5 kt ja mobiiliversiona 1568,2 kt. Ulkoista JavaScriptiä ladattiin vastaavasti 399,4 kt:n ja 350,5 kt:n edestä. Kuvia taas 649.6 kt:n ja 794.0 kt:n edestä. (HTTPArchive 2018.) Resurssien kokoon on tärkeä kiinnittää huomiota, sillä niillä on suora vaikutus sivulatauksen nopeuteen. Resurssia ei voi siirtää nopeampaa kuin mikä alla olevan yhteyden kaistanleveys on, mikä kertoo kuinka monta bittiä sekunnissa yhteys voi kuljettaa sekunnissa (Spero 1994). Mitä suurempi resurssi on, sitä enemmän bittejä on kuljetettavana ja sitä kauemmin sen siirto palvelimelta asiakkaalle kestää (Souders 2007, 29).

Eri resurssityypeillä on erilainen vaikutus sivulataukseen, vaikka ne olisivatkin saman kokoisia. Kun dokumentin jäsentäjä kohtaa skriptin, jäsennys keskeytetään skriptin suorituksen ajaksi. Jos skripti on ulkoinen resurssi, se täytyy ensin myös ladata. Tätä käyttäytymistä voidaan muuttaa skriptin *async* ja *defer* attribuuteilla. Molemmat attribuutit kertovat, että HTML-jäsentäjä voi jatkaa dokumentin jäsentämistä skriptin latauksen aikana, mutta niiden käyttäytyminen eroaa latauksen valmistuttua. *async*-skripti suoritetaan heti latauksen jälkeen, mutta *defer*-skripti suoritetaan vasta, kun skripti on ladattu ja dokumentti on kokonaan jäsennetty. (WHATWG 2018, 595-596.) JavaScriptin kustannukset eivät kuitenkaan lopu siihen, sillä latauksen ja jäsennyksen lisäksi se tulee myös kääntää. Suorituskyvyltään heikommille laitteille, kuten mobiililaitteille, käännöksen ja jäsennyksen aiheuttama viive voi olla merkittävä. (Osmani 2018a.) Chromen kehittäjätyökaluilla mitattuna (kuva 3) https://angular.io:n sivulatauksesta noin 700 ms kuluu JavaScriptin kääntämiseen ja noin 270 ms parsimiseen. Yhdessä ne kattoivat siis yli 60% JavaScriptin latauksen jälkeisestä sivulataukseen vaikuttavasta aikaviiveestä.

Self Time		Total Time		Activity
1551.5 ms	62.2 %	1551.5 ms	62.2 %	▼ JavaScript
427.9 ms	17.1 %	706.1 ms	28.3 %	▶ Compile
246.1 ms	9.9 %	269.5 ms	10.8 %	▶ Parse

Kuva 3. Jäsennyksen ja käännön kustannukset mitattuna Chromen kehittäjätyökalulla. Rajoitukset mobiilikokemuksen imitoimiseksi: nopea 3G-verkko ja nelinkertainen CPU hidastus.

JavaScriptin lisäksi myös CSS:llä voi olla merkittävä vaikutus sivulatauksen edistymiseen, sillä se on HTML:n kanssa oletuksellisesti piirtämisen estävä resurssi. Koska sivun piirto on verkkoselaimelle kallis operaatio (tästä lisää luvussa 3.2.3), eikä sivu välttämättä ole käytettävä ilman tyylilehtiä, odottaa verkkoselain tyylilehden latautumista ennen sivun ensimmäistä piirtoa. Tyylilehtien kokoon tulee siis kiinnittää huomiota. (Grigorik 2018e.)

3.2.2 Ladattavien resurssien lukumäärä

Paras ja optimaalisin resurssi on sellainen, jota ei lähetetä (Grigorik 2018a). HTTP-pyyntöjen (Hypertext Transfer Protocol) vähentäminen on ensimmäinen Soudersin (2007) esittämässä priorisoidussa listassa verkkosivujen nopeuttamiseksi ja suurin latausaikaan vaikuttava tekijä Butkiewicz ym. (2014, 954) toteuttamassa tutkimuksessa. Resurssien lukumäärän aiheuttama negatiivinen vaikutus johtuu pitkälti siitä, kuinka HTTP käyttää alla olevaa TCP-yhteyttä (Transmission Control Protocol). HTTP/1.x ei kunnolla tue montaa samanaikaista pyyntöä yhteyttä kohden (1.0 ei ollenkaan), joten sovelluksen viiveen pienentämiseksi usean pyynnön tekemiseen on käytetty montaa yhteyttä. Yhteyden muodostaminen ja pyyntöjen otsikkokenttien usein toistuva sekä runsassanainen luonne lisää resurssin pyynnön yleiskustannuksia. (Belshe ym. 2015, 4.) Samanaikaisten HTTP-pyyntöjen lukumäärää yleensä myös rajoitetaan verkkoselaimissa: esimerkiksi Mozilla Firefoxin versiossa 61.0.2 sallitaan oletusarvoisesti kuusi yhteyttä palvelinta kohden.

Vuonna 2015 esitelty HTTP:n uusi versio, HTTP/2, pyrkii tehostamaan verkkoresurssien käyttöä mahdollistamalla otsikon pakkauksen ja sallimalla monen samanaikaisen pyynnön samalla yhteydellä (Belshe ym. 2015, 4-6). Voi siis olla, että ladattavien resurssien lukumäärällä ei jatkossa ole niin paljon merkitystä kuin ennen, mutta vielä tällä hetkellä verkkoselaimien tuki HTTP/2:lle on noin 84% (Deveria 2018), joten jonkin aikaa resurssien lukumäärän optimointi on edelleen tärkeää.

Tässä työssä ei ajatella resurssien tarpeellisuutta eli pohdita niiden tuomaa arvoa käyttäjille tai sivuston ylläpitäjälle, joten työn menetelmiin ei kuulu turhien resurssien karsiminen. Työn menetelmät yksinkertaisesti vain pyrkivät vähentämään ladattavien resurssien lukumäärää, vaikuttamaan resurssin latausjärjestykseen tai -ajankohtaan sivulatauksen nopeuttamiseksi.

3.2.3 Sovelluksen suorituskyky

Yhden sivun verkkosovellus renderoidaan verkkoselaimessa asiakkaan laitteella, joten sovelluksen suorituskyky on yksi hidastava tekijä sivua ladattaessa. Suorituskykyyn liittyy kuitenkin tärkeitä yksityiskohtia, joita avataan seuraavaksi.

JavaScriptin suorittamiseen on saatavilla pääasiassa yksi säie, jota kutsutaan pääsäikeeksi (eng. main thread). Pääsäikeen kutsupino suoritetaan aina kokonaisuudessaan alusta loppuun. Samanaikaisen (ei rinnakkaisen) suorittamisen mahdollistaa niin kutsuttu tapahtumasilmukka (eng. event loop), joka siirtää tapahtumienkäsittelijöitä (eng. event handler) ja lupauksienkäsittelijöitä (eng. promise handler) verkkoselaimen työjonoista (eng. job queue) yksitellen kutsupinoon. (Ecma International 2015, 76.) Koska suorituskontekstilla on vain yksi kutsupino ja uusia tehtäviä suoritetaan vasta kun pino on tyhjä, voi pitkään yhtäjaksoisesti suorituksessa oleva skripti estää uusien tapahtumien käsittelyn. Sovelluksen käyttäjälle kutsupinon estyminen näkyy sovelluksen hitaana responsiivisuutena, mikä heikentää sovelluksen vuorovaikutuksellisuutta. (Osmani 2018a.) Tapahtuman- ja lupauksienkäsittelijät tulisi siis aina pitää mahdollisimman nopeina. Raskaita ja pitkäkestoisia suorituksia kannattaa – jos mahdollista – siirtää Web Workereihin, jotka toimivat omissa suorituskonteksteissaan, eikä niillä siten ole suoraa vaikutusta sovelluksen responsiivisuuteen (WHATWG 2018, 970).

Kuten luvussa 3.2.1 mainittiin, JavaScriptin suoritus vaikuttaa myös HTML:n jäsennykseen. Kun HTML:n jäsentäjä kohtaa dokumentissa JavaScriptia, DOMin rakennus keskeytetään ja kontrolli annetaan JavaScript-moottorille. Skriptin suorituksen jälkeen jäsennysprosessi jatkuu, mutta DOMin rakentuminen on viivästynyt skriptin suoritukseen kuluvan ajan verran. (Grigorik 2018c.)

Kolmas mahdollisesti käyttäjälle näkyvä sivulatauksen hidastuminen johtuu sovelluksen muistinhallinnasta. JavaScriptissä muistia ei hallita itse, vaan siitä huolehtii automaattinen roskienkeruu (eng. garbage collection). Roskienkeruulla on seuraamuksia suorituskyvylle, sillä se estää pääsäikeen suorituksen roskienkeruuprosessin ajaksi. (Osmani 2018a.)

Viimeisenä sovelluksen suorituskykyyn vaikuttavana asiana mainittakoon eräs verkkoselaimen suorittama kallis operaatio, eli renderointipuun luonti ja piirto. Sivun piirtämiseksi verkkoselain rakentaa renderointipuun yhdistämällä DOMin ja CSSOM:än (Cascading Style Sheet Object Model), jonka jälkeen puun solmut sijoitetaan ja piirretään ruudulle. Prosessi toistetaan aina, kun DOM tai CSSOM muuttuu (Grigorik 2018d.) Kehykset, kuten ReactJS ja VueJS, pyrkivät minimoimaan piirron kustannuksia päivittämällä aluksi vain virtuaalista DOMia ja aika-ajoin viemällä muutokset suuremmassa erässä oikeaan DOMiin, jolloin sivua ei tarvitse piirtää niin usein. Tähän ei niinkään voi koontivaiheessa vaikuttaa, mutta siihen kannattaa kiinnittää huomiota sovellusta rakentaessa.

3.3 Optimoinnin aloittaminen

Luvussa 3.1 esiteltiin sivulataus monivaiheisena ilmiönä ja luvussa 3.2 esiteltiin kolme kehittämiskohdetta sivulatauksen nopeuttamiseksi. Koska työssä ei rajata millaiselle sovellukselle optimointia tehdään, on vaikea sanoa minkä suuruinen vaikutus kunkin asian optimoinnilla voi sivulataukseen olla. Ongelmaa vaikeuttaa myös se, että sivulatausta hidastavat tekijät voivat vaikuttaa eri asiakkaisiin eri tavoin. Hiukan kärjistäen voidaan ajatella, että tehokkaalla laitteella varustettu asiakas ei sovelluksen heikkoa suorituskykyä välttämättä havaitse, mutta jos hänen yhteyden nopeus tai vakaus ovat heikkoja, voi resurssien koolla ja lukumäärällä olla runsaasti merkitystä. Toisaalta jos asiakkaan yhteys on hyvä, mutta laitteen suorituskyky heikko, voi sovelluksen suorituskyvyn optimointi olla kriittisempää. Sovelluksen suorituskyvystä voidaan vielä sanoa, että monet kehykset pyrkivät ratkaisemaan juuri suorituskykyyn liittyviä ongelmia, joten sovelluksessa käytetyn kehyksen parhaisiin käytäntöihin ja ohjeisiin kannattaa tutustua.

Minkä tahansa verkkosovelluksen optimointi kannattaa aloittaa mittaamalla mikä sivulatauksessa vie aikaa. Mittaus tulisi mieluiten suorittaa sellaisessa ympäristössä, missä sovelluksen asiakaskunta sitä käyttää. Mittaamisen tueksi voi käyttää työkaluja, kuten

luvussa 3.1 käytettyä Googlen kehittämää Lighthousea, mutta hyvää tietoa voi kerätä jo verkkoselaimen omalla kehittäjätyökalulla. Tarkempien mittauksien tekemiseen on kehitteillä uusi *PerformanceNavigationTiming*-rajapinta (W3C 2017a).

4 KOONTIVAIHEEN TEKNIIKAT SIVULATAUKSEN NOPEUTTAMISEKSI

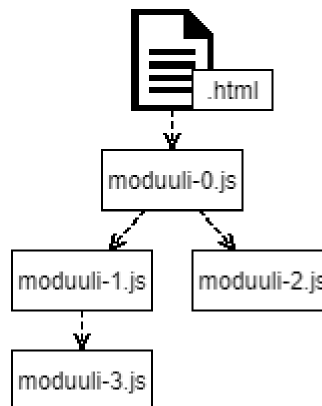
Tässä luvussa esitetään erilaisia menetelmiä yhden sivun verkkosovelluksen sivulatauksen nopeuttamiseksi. Kaikkia esitettyjä menetelmiä ei käytetä ainoastaan yhden sivun verkkosovelluksissa, vaan niitä voidaan hyödyntää myös muissa verkkosovelluksissa. Olemassa olevia menetelmiä on lukuisia ja niitä tulee jatkuvasti lisää, eikä kaikista voida tässä työssä kirjoittaa. Tähän lukuun on pyritty poimimaan sellaisia menetelmiä, jotka todennäköisesti tulevat vastaan yhden sivun verkkosovelluksen kehityksessä tai etsiessä ratkaisuja latausnopeuden parantamiseksi. Tyypillisesti käytettyjä menetelmiä etsittiin eri kehysten (ReactJS, VueJS, Angular) dokumentaatioista, kyseisissä dokumentaatioissa tai kehyksien keskustelupalstoilla mainituista työkaluista tai niiden kilpailijoista, sekä kehyksille tehdyistä hallintatyökaluista. Lista työssä hyödynnetyistä työkaluista löytyy liitteestä 1.

4.1 Resurssien paketointi, jakaminen ja viivytetty lataaminen

Moderni JavaScript-kehitys rakentuu moduulien ympärille ja erilaisia moduulimäärittelmiä on esitetty monia. ECMAScript standardin ECMA-262 kuudennen version (Ecma International 2015, 266) myötä moduulit ovat vihdoin tulossa viralliseksi osaksi JavaScriptiä. Monissa koontityökaluissa on tuki erilaisille moduulijärjestelmille, mutta tässä työssä keskitytään vain viralliseen järjestelmään, eli ES-moduulijärjestelmään (ECMAScript).

Modulaarisuus on tärkeä apu koodikannan hallinnassa. Kuitenkin koodin jakaminen moneen pieneen osaan kasvattaa HTTP-pyyntöjen lukumäärää (Souders 2007, 15.) Toinen tärkeä huomio on se, että moduulit määrittelevät omat moduuliriippuvuutensa. Jotta moduulin riippuvuudet voidaan tunnistaa, täytyy se ladata ja jäsentää. Vasta riippuvuuksien tunnistamisen jälkeen ne voidaan tyydyttää lataamalla, mutta koska myös näillä moduuleilla voi olla riippuvuuksia, täytyy sama ketju toistaa jokaisen moduulin kohdalla. Verkkoselaimessa ES-moduulien lataustavan määrittelee WHATWG:n (2018, 871-874) HTML spesifikaatio.

Riippuvuuspuun (kuva 4) lataamista on havainnollistettu kuvassa 5. Esimerkissä HTML-dokumentin sisääntulomoduuli on moduuli-0, joka on riippuvainen moduuleista 1 ja 2. Moduulien 1 ja 2 lataus ei kuitenkaan ala, ennen kuin moduuli 0 on ladattu ja riippuvuudet ovat tunnistettu. Tässä yksinkertaisessa esimerkissä HTML dokumentin ja jokaisen moduulin lataus kestää noin 2 sekuntia, mikä tarkoittaa, että sivulataukseen kuluu kokonaisuudessaan 8 sekuntia.



Kuva 4. ES-moduulien riippuvuuspuu.

Name	Status	Type	Initiator	Size	Time	Waterfall
module-doc.html	200	document	Other	452 B	2.02 s	
module-0.js	200	script	module-doc.html	389 B	2.03 s	
module-1.js	200	script	module-0.js:1	417 B	2.02 s	
module-2.js	200	script	module-0.js:2	354 B	2.04 s	
module-3.js	200	script	module-1.js:1	399 B	2.01 s	

Kuva 5. ES moduulien latausketju. Generoitu Chrome kehittäjätyökaluilla.

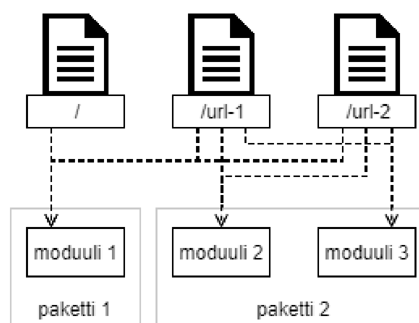
Riippuvuuspuun ketjumaisen lataustavan voi ohittaa paketoinnilla säilyttäen silti moduulien hyvät ominaisuudet. Paketointi yhdistää riippuvuudet yhteen tiedostoon, jolloin kaikki riippuvuudet voidaan ladata kerralla (Souders 2007, 15-16). Näin yksinkertaisessa paketoinnissa on kuitenkin omat puutteensa. Suuremmissa verkkosovelluksissa löytyy todennäköisesti toiminnallisuuksia, joita ei jokaisessa näkymässä tai heti sivulatauksen yhteydessä tarvita. Tällaisissa tapauksissa voi olla hyödyllistä jakaa ohjelmapaketti osiin, jotta sivulatauksessa ei lähetetä asiakkaalle ylimääräisiä resursseja hidastaen näkymän renderointia. Korkealla tasolla menetelmä on kaksiosainen:

1. koodi jaetaan osiin halutun strategian mukaisesti
2. ohjelman ajon aikana osat ladataan, kun niitä tarvitaan.

Erilaisia jako- ja latausstrategioita voi olla monia, mutta tässä työssä esitellään kaksi (luvut 4.1.1 ja 4.1.2).

4.1.1 Reittikohtainen jako

Yhden sivun verkkosovelluksen kulloisenkin näkymän määrittelee sen tila (Shahzad 2017, 412), johon voi sovelluksesta riippuen kuulua myös HTML5 määrittelemän Location-rajapinnan (W3C 2014) kautta saatava dokumentin sen hetkinen URL (Unified Resource Locator). URL:ää hyödyntävissä sovelluksissa eräs käytetty koodinjakomenetelmä on vetää jakoraja sovelluksessa määriteltyihin tunnettuihin reitteihin. Sovelluksesta muodostetaan monia skriptipaketteja siten, että jokaisella reitillä on oma pakettinsa. Reiteille yhteiset toiminnallisuudet erotetaan omaan pakettiin toiminnallisuuden kahdentamisen välttämiseksi. (Webpack 2018b.) Asiakkaan tehdessä pyyntö palvelimelle johonkin URL:ään, liitetään vastauksena lähetettyyn dokumenttiin pyynnön URL:ää vastaavan reitin paketti sekä yhteisen toiminnallisuuden sisältävä paketti. Kuvassa 6 esitetyssä esimerkissä moduuli 1 voisi olla sovelluksen käyttämä yhden sivun verkkosovelluksen kehys, josta kaikki reitit ovat riippuvaisia. Moduulit 2 ja 3 sisältävät muuta toiminnallisuutta, josta sekä /url-1 ja /url-2 ovat riippuvaisia. Tässä yksinkertaisessa esimerkissä tehokas jakotapa voisi olla tehdä 2 pakettia, joista ensimmäiseen laitetaan vain moduuli 1 ja toiseen moduulit 2 ja 3. Ladatessa reittiä /, vain paketti 1 lähetetään. Reittien /url-1 ja /url-2 latauksessa lähetetään molemmat paketit.



Kuva 6. Esimerkki reittiperusteisesta koodijaosta

Ensimmäisen latauksen jälkeen seuraavien näkymien pakettien lataus tulee hoitaa sovelluksessa. Näkymän paketit voidaan ladata, kun ohjelman tila on muuttunut näyttämään kyseistä näkymää, tai ennakoidusti. Eräs ennakointitapa voisi olla ladata tarvittavat paketit silloin, kun käyttäjä on siirtänyt osoittimen näkymään johtavan linkin kohdalle. Toinen vielä

aggressiivisempi tapa on ladata paketit kaikkiin näkymiin, joihin sen hetkisestä näkymästä on linkki. HTTP/2 mahdollistaa myös resurssien puskemisen palvelimelta heti ensimmäisen pyynnön yhteydessä, mutta silloin asiakkaat, joilla resurssi löytyy jo välimuistista, joutuu lataamaan resurssin turhaan.

4.1.2 Toiminnallinen jako

Hankalammin toteutettava jako voi olla lähettää sivulatauksen yhteydessä ainoastaan näkymän renderointiin tarvittava toiminnallisuus, ja vasta sivun renderoinnin jälkeen lähettää toiminnallisuus, joka tarvitaan näkymässä vuorovaikuttamiseen. Souders (2009, 21) antaa tällaisesta jaosta esimerkkinä pudotusvalikon: sen toiminnallisuuden aikaansaavaa skriptiä ei sivun renderointiin tarvita, joten sen lataamista voidaan viivästyttää. Tekniikan käyttö ei kuitenkaan ole niin suoraviivaista, sillä renderoinnin jälkeen pudotusvalikko on nähtävissä, mutta ei käsiteltävissä, ennen kuin viivästetty lataus on suoriutunut loppuun ja tarvittavat tapahtumankäsittelijät ovat siihen yhdistetty. Ratkaisuna voi olla muuttaa pudotusvalikon ulkomuotoa tai käyttää tynkäfunktioita, kunnes toiminnallisuus on kytketty. Tarvittavien skriptien tunnistamiseksi voi käyttää opetusta, eli ladata sivu ja tallentaa mitä toiminnallisuuksia tarvitaan pelkästään renderoimiseen. (Souders 2009, 24.)

4.2 Koodin tiivistäminen

Koodin tiivistämisen tavoitteena on pienentää resurssien kokoa. Suositussa Babel kääntäjässä koodin tiivistämisen menetelmiä kutsutaan yleisesti minifioinniksi, mutta Soudersin (2007, 69-71) määritelmän mukaan useammat näistä menetelmistä olisivat luokiteltavissa obfuskoinniksi, jonka hän esittelee olevan minifioinnille vaihtoehtoinen, hiukan pidemmälle viety optimointikeino. Tässä työssä koodin tiivistämismenetelmien katsotaan olevan minifiointia, ja koodin obfuskaatio vain seurausta joidenkin tiivistämismenetelmien käytölle. Koontivaiheessa tehtävä tiivistäminen ei ole vaihtoehtoinen optimointitapa verkkopalvelimien suorittamalle pakkaukselle (gzip, deflate, br, jne.), vaan kumpaakin voidaan tehdä samanaikaisesti ja osa tiivistämismenetelmistä voi jopa edistää pakkaamista. Tiivistämisen hyvä puoli on se, että tiivistettyä resurssia ei tarvitse purkaa ennen käyttöä, toisin kuin pakatulle resurssille tulee tehdä. Yksittäisiä menetelmiä on niin monta, että tässä työssä tyydytään nostamaan esiin vain muutama tiivistämisen perusidean havainnollistamiseksi.

4.2.1 Tarpeettomien merkkien poisto

Tarpeettomien merkkien poisto on tekniikka, jolla pienennetään lähdekoodin kokoa poistamalla siitä kaikki osat, joita sen suorittamiseen ei tarvita. Tämä menetelmä vastaa Soudersin (2007, 69-71) määritelmää minifioinnista. Menetelmässä tyypillisiä poistettavia kohteita ovat kommentit ja merkit kuten esimerkiksi välilyönnit, sarkaimet ja rivinvaihdot. Tarpeettomien merkkien poisto on erityisen hyödyllistä verkkosovelluksissa, sillä ne rakennetaan tulkatuilla ohjelmointikielillä ja lähetetään sellaisenaan palvelimelta asiakkaalle. (Souders 2007, 69.) Verkkosovelluksissa soveliaita menetelmän kohderesursseja ovat mm. HTML, JavaScript, CSS, JSON (JavaScript Object Notation) sekä SVG (Scalable Vector Graphics), sillä kaikkia näitä resursseja käsitellään yleensä helpommin luettavassa muodossa jäsennettynä sarkaimilla ja rivinvaihdoilla.

4.2.2 Nimien lyhentäminen

Nimien lyhentämisessä koodissa määritellyt muuttujien, parametrien ja funktioiden nimet muutetaan lyhyemmiksi (kuva 7). UglifyJS ja Babel työkaluissa tätä menetelmää kutsutaan koodin murjomiseksi (eng. mangle).

```
// Ennen
function someVeryLongFunctionName(longParameterName) {
  var longVariableName;
  // Do something
}

// Jälkeen
function a(_0) {
  var _1;
  // Do something
}
```

Kuva 7. Esimerkki nimien lyhentämisestä

4.2.3 Tosiarvojen muunto

Tosiarvojen muunto (kuva 8) on yksinkertainen muunnos, jossa tosiarvot vaihdetaan lyhyempään esitysmuotoon. JavaScriptissä arvo 0 on epätosi ja nollan negaatio !0 siis epätoden negaatio, eli tosi. Menetelmän avulla voi tavoitella pieniä säästöjä tiedonsiirrossa.

```
// Ennen
true;
false;

// Jälkeen
!0;
0;
```

Kuva 8. Tosiarvojen muunto

4.2.4 Muuttujaesittelyiden yhdistäminen

Muuttujaesittelyiden yhdistäminen (kuva 9) on toinen pieni muunnos, millä voi säästää tärkeitä tavuja. Muunnossa useampaan lauseeseen eritellyt muuttujaesittelyt yhdistetään yhteen lauseeseen, jolloin tarvitaan vain yksi esittelyavainsana monen sijasta.

```
// Ennen
var v1 = "string 1";
var v2 = "string 2";

// Jälkeen
var v1 = "string 1",
    v2 = "string 2";
```

Kuva 9. Muuttujamäärittelyiden yhdistäminen

4.2.5 Tiivistyksen tuomat säästöt

Koodin tiivistäminen voi vähentää tiedonsiirtokustannuksia merkittävästi. Vaikka iso osa muunnoksista on pieniä, niin suuremmissa ohjelmissa kokonaissäästöt voivat olla merkittävät. Yksinkertaisesta JavaScript esimerkistä (kuva 10) nähdään, miten alkuperäisessä muodossa oleva JSDocilla dokumentoitu funktio vaatii noin 6,4 kertaisen määrän tilaa prosessoituun koodiin verrattuna. Tiivistys tuottaa koodin, joka on toiminnaltaan vastaava, mutta tarvitsee huomattavan paljon vähemmän tilaa. Tässä esimerkissä suurimman osan tilasta vie funktion dokumentaatio, eli 126 tavua, mutta myös ilman kommentteja prosessoimaton koodi vaatii miltei kolminkertaisen määrän tilaa prosessoituun koodiin verrattuna.

```
// Ennen
/**
 * Returns the factorial of a nonnegative integer
 * @param {number} value A nonnegative integer
 * @returns {number}
 */
function factorial(value) {
  if (value < 2) {
    return 1;
  }
  return factorial(value - 1) * value;
}

// Jälkeen
function r(n){return n<2?1:r(n-1)*n}
```

Kuva 10. Esimerkki koodin tiivistämisestä

Tiivistämisellä on joitakin huomioon otettavia seuraamuksia. Tiivistämismenetelmät muuttavat koodin usein vaikeammin luettavaan muotoon, mikä Soudersin (2007, 70) mukaan voi olla obfuskoinnin yksi tavoite takaisinmallinnuksen (eng. reverse-engineering) hankaloittamiseksi. Toisaalta menetelmät, jotka muuttavat koodin rakennetta muutenkin kuin poistamalla tarpeettomat merkit, voivat tuottaa koodiin hankalasti löydettäviä virheitä. (Souders 2007, 70.)

4.3 Kuolleen koodin eliminointi

Kuolleen koodin eliminointi (eng. dead code elimination) on vanha koodin optimointitekniikka, joka edeltää WWW:n keksimisen ja jota hyödynnetään monissa ohjelmointikielien kääntäjissä. Se saa nimensä prosessista, jossa käyttämättömät – tai kuolleet – käskyt poistetaan ohjelmasta. Käskyjen katsotaan olevan kuolleita, jos niitä ei sijaintinsa takia voida suorittaa tai niiden tuottamia tuloksia ei ohjelmassa käytetä. (Allen & Cocke 1971, 20.) Tällaisen optimoinnin tärkeä tavoite on pienentää ohjelman suoritusaikaa (Allen & Cocke 1971, 1) ja sulautetuissa sovelluksissa ohjelman kokoa (Aho ym. 2006, 16), jotka ovat molemmat tärkeitä tavoitteita myös verkkosovelluksille.

4.3.1 Ympäristökohtaiset käskyt

Kuolleen koodin optimointia käytetään runsaasti ympäristökohtaisessa koonnissa: ohjelman kehitysversiossa ohjelma voi sisältää ylimääräisiä ajonaikaisia tarkistuksia ja tulostuksia,

mutta tuotantoversiossa ne ovat ohjelmasta poistettu. Esimerkissä (kuva 11) muuttuja __DEV__ on vakio, joka kertoo, onko ohjelma kehitysversio vai ei. Kehitysversiossa muuttuja saa tosiarvon, jolloin ehtolause on tosi ja lohko suoritetaan. Tuotantoversiossa ehtolause on aina epätosi, jolloin lohkoa ei koskaan suoriteta; se on siis kuollutta koodia ja turvallista poistaa.

```
// Ennen
function circleArea(radius) {
  if (__DEV__) {
    if (radius < 0) throw "Radius cannot be negative";
  }
  return Math.PI * radius * radius;
}

// Jälkeen (__DEV__ = epätosi)
function circleArea(radius) {
  return Math.PI * radius * radius;
}
```

Kuva 11. Esimerkki kuolleen koodin optimoinnista

4.3.2 Puunravistelu

Puunravistelu (eng. tree shaking) on yksinkertainen, JavaScript-yhteisössä suosittu menetelmä, jonka päämäärä on päästä eroon käyttämättömistä käskyistä. Sen toiminta perustuu luvussa 4.1 esiteltyyn moduulijärjestelmään, missä moduulien tuonti- ja vientilausekkeet ovat eksplisiittisesti ilmaistu ja helposti staattisesti analysoitavissa. (Rollup 2018; Webpack 2018a.) Puunravistelu toimii rakentamalla malli ohjelman riippuvuuksista seuraamalla tuontilausekkeita sisääntulomoduulista lähtien. Mallia voi ajatella puuna, jonka juuri on sisääntulomoduuli. Kaikki ohjelman riippuvuudet ovat edustettuna puussa solmuina. Koodi, jota ei puusta löydy, on tarpeetonta ja jätetään pois ohjelmasta sen koontivaiheessa.

Vaikka puunravistelu ei poista kuollutta koodia esimerkiksi funktioiden sisältä, voi se silti tuoda suuret säästöt lopullisen ohjelman kokoon. JavaScript moduuliriippuvuudet linkitetään kokonaisuudessaan osaksi ohjelmaa ja ilman puunravistelua lyhyenkin koodin lisäys esimerkiksi jostain kirjastosta tuo mukanaan koko kirjaston.

4.4 Esilaskenta

Laskemalla ohjelman tuloksia jo koontivaiheessa voidaan pienentää ohjelman kokoa ja muistijalanjälkeä sekä välttää ylimääräinen laskenta asiakkaan laitteessa, mikä nopeuttaa ohjelman suoritusta ja latausaikaa.

4.4.1 Vakioiden levitys

Vakioiden levitys (eng. mm. subsumption, constant folding, constant propagation) on hyvin tunnettu tekniikka, joka koskee nimensä mukaisesti vakioiden käsittelyä. Tekniikassa vakioiksi asetetut muuttujat korvataan vakioarvolla itsellään, ja lauseet, joiden operandit ovat vakioita, suoritetaan. (Allen & Cocke 1971, 18-19.) Menetelmä toimii hyvin kuolleen koodin eliminoinnin kanssa, sillä sisällytyksen jälkeen jää usein käyttämätöntä koodia. Yksinkertaisessa esimerkissä (kuva 12) on joukko vakioiden määrittelyjä, sekä lopputuloksen tulostus. Vakioiden levityksen jälkeen muuttujalle x sekä tulostusfunktiolle on laskettu arvot valmiiksi. Yhdistettynä kuolleen koodin eliminoinnin kanssa jäljelle jää ainoastaan sivuvaikutuksellinen tulostusfunktion kutsu.

```
// Ennen
const a = 100;
const b = 5;
const x = a / (5 + b);
console.log(x);

// Vakioiden levityksen jälkeen
const a = 100;
const b = 5;
const x = 10;
console.log(10);

// Vakioiden levityksen + kuolleen koodin eliminoinnin jälkeen
console.log(10);
```

Kuva 12. Vakioiden levitys

4.4.2 Esirenderointi

Kuten luvussa 3 esitettiin, kuuluu sivulataukseen myös resurssien latauksen ja parsimisen lisäksi sivuston renderointi, joka yhden sivun verkkosovelluksien tapauksessa tapahtuu suorittamalla JavaScriptiä. Jotkin luvussa 4 esitetyt menetelmät voivat nopeuttaa

renderointia, mutta kuten luvussa 3.2 todettiin, voi se etenkin suoritusteholtaan heikommilla laitteilla viedä suuren lohkon sivulataukseen kuluvasta ajasta. Esirenderoinnilla asiakkaan laitteella tehtävä ensimmäisen sivun renderointi voidaan jättää välistä kokonaan.

Esirenderointi on ajatuksena melko yksinkertainen: sen sijaan, että ensimmäisessä sivulatauksessa asiakkaalle lähetettäisiin tyhjä dokumentin keho ja koko sivu renderoitaisiin verkkoselaimessa, renderoidaan sovelluksen näkymät valmiiksi palvelimella ja pyyntöön vastataan jo renderoidulla dokumentilla. Näin sivu on näkyvissä käyttäjälle aikaisemmin. Sivulatauksen jälkeen sovellus kiinnittyy jo renderoituun dokumenttiin ja jatkaa toimintaa normaalin yhden sivun verkkosovelluksen mukaisesti. (Osmani 2018a.) Tuki palvelimella renderointiin ja verkkoselaimessa DOMiin kiinnittymiseen löytyy joistakin suosituista kehyksistä, kuten ReactJS ja VueJS. Kehyksistä riippumaton mahdollinen ratkaisu voi olla käyttää päätöntä (eng. headless) verkkoselainta, kuten Googlen Headless Chromium, sivun lataukseen ja DOMin poimimiseen.

Monissa tapauksissa palvelimella renderointi on mahdollista reaaliajassa – aivan kuten normaali verkkosovellus toimii, eli palvelin renderoi HTML-dokumentin vastauksena palvelinkutsuun. Työn aiheen mukaisesti keskitytään kuitenkin koontivaiheen menetelmiin, joten dokumentti tulee renderoida koontivaiheessa ja se aiheuttaa tiettyjä rajoitteita. Monet esirenderointityökalut tuottavat sivuston tunnettujen polkujen perusteella, eli jokaiselle tunnetulle polulle tuotetaan oma HTML-dokumentti. Menetelmä ei välttämättä ole mahdollinen, jos sivuston sisältö riippuu esimerkiksi käyttäjäkontekstista, sillä generoitavien ja talletettavien sivujen lukumäärä voi kasvaa liian suureksi. Esirenderointityökalut, kuten GatsbyJS ja VuePress, kutsuvatkin itseään usein staattisen sivuston generaattoreiksi. Hyvä keskitie voi olla esirenderoida vain laskeutumissivut (eng. landing page), joiden nopea latautuminen on tärkeää, kuten esimerkiksi verkkosivuston etusivu. Toinen vaihtoehto voisi olla omaksua progressiivisten verkkosovelluksien käyttämä sovelluskuorimalli (eng. app shell model), jonka tavoite on erottaa minimaalinen, helposti välimuistiin varastoitava osuus dynaamisesta sisällöstä (Osmani 2018b), ja pitää sovelluksen kuoren renderointi esirenderoinnin tavoitteena.

Esirenderointi voi auttaa myös luvussa 2 esitetyissä yhden sivun verkkosovelluksien heikkouksissa, koska pyyntöön voidaan vastata täytetyllä dokumentilla, eikä JavaScript ole

välttämättön sisällön tuottamiseen. Menetelmällä on myös haittapuolia, sillä esirenderoitu sivulataus on yleensä suurempi eikä näkyviä elementtejä voi käyttää, ennen kuin niiden käsittelyyn tarvittavat skriptit ovat myös latautuneet, mikä voi hidastaa sovelluksen siirtymistä vuorovaikutettavaan tilaan (Osmani 2018a).

4.5 Templaattien kääntäminen

Kehykset käyttävät usein templaatteja (ReactJS, Vue.js, AngularJS) kehittäjäkokemuksen parantamiseksi. Näitä itse määriteltyjä templaattisyntakseja ei tueta verkkoselaimissa, joten ne tulee kääntää (kuva 13) verkkoselaimen ymmärtämään muotoon. Kääntäjän voi usein ladata ja ajaa verkkoselaimessa, mutta se kasvattaa ajonaikaisia kustannuksia lisäämällä tarvittavien resurssien ja suoritettavien skriptien määrää. Templaattien kääntäminen kannattaa siis tehdä jo koontivaiheessa.

```
// Alkuperäinen, ReactJS JSX
<div>
  {show && (
    <h1 className="greeting">
      Hello, { name }!
    </h1>
  )}
</div>

// Käännetty puhtaaksi JavaScriptiksi
React.createElement(
  "div",
  null,
  show &&
  React.createElement("h1",{className:"greeting"},"Hello, ",name,"!")
);
```

Kuva 13. Templaattisyntaksin käännös

4.6 Riville sijoittaminen

Riville sijoittaminen (eng. inlining) on tekniikka, jossa resurssi sijoitetaan suoraan lähdekoodiin, eikä sitä ladata ulkoisena resurssina. Koska resurssi sijoitetaan osaksi toista resurssia, ei sitä tarvitse ladata erikseen, mikä vähentää tarvittavien HTTP-pyyntöjen määrää. (Souders 2007, 13; Souders 2009, 69.) Riville sijoittamista voi tehdä monille eri resurssityypeille:

- JavaScript voidaan sijoittaa HTML-dokumenttiin script-tagien avulla
- CSS voidaan sijoittaa JavaScriptiin merkkijonona tai HTML-dokumenttiin style-tagien avulla
- Kuvia ja muuta dataa Data URL:ää hyödyntämällä.

Kaikkien resurssien riville sijoitus ei ole kannattavaa. Sovelluksessa voi esimerkiksi olla monta komponenttia, jotka käyttävät samaa SVG-ikonia käyttöliittymässään. Jokaisen komponentin koko on riville sijoituksen seurauksena ikonin koon verran suurempi. Sen sijaan, että ikoni ladattaisiin kerran ja talletettaisiin uudelleenkäyttöä varten välimuistiin, ladataan ikoni jokaisen komponentin mukana uudelleen. (Souders 2007, 14.) JavaScriptiä riville sijoittaessa täytyy olla vielä varovaisempi, sillä suorituksen ajaksi dokumentin parsiminen keskeytetään. Siksi sisällytetty JavaScript tulisi sijoittaa dokumentin loppuun ja tehdä siitä asynkronisesti suoriutuva, jos mahdollista. (Souders 2009, 70.)

4.7 Kuvien optimointi

Vuoden 2018 alusta kesäkuun loppuun kuvien mediaanikoko verkkosivuilla on ollut työpöytäversioissa 649,6 kt (43,2 %) ja mobiiliversiossa 794,0 kt (50,6 %) (HTTPArchive 2018). Ne ovat siis merkittävä osuus verkkosivujen kokonaispainosta. Kuvien koon vaikutus suorituskykyyn ei kuitenkaan ole yhtä vakava kuin JavaScriptin, koska kuvat eivät yleensä estä pääsäiettä dekoodauksen ja rasteroinnin aikana (Osmani 2018a). Silti niillä voi olla suuri vaikutus sovelluksen suorituskykyyn kilpaillessa verkkoresursseista muiden resurssien kanssa (Grigorik 2018b).

4.7.1 Vektorikuvat

Vektorikuvat perustuvat XML-merkintäkieleen (Extensible Markup Language) ja sisältävät usein ylimääräistä tietoa, jota ei tarvita kuvan renderointiin verkkoselaimessa. Tästä syystä yksi optimointitapa on käyttää luvussa 4.2 esitettyä tiivistämistä. (Grigorik 2018b, Osmani 2018c.) Toinen vektorikuville tehtävä muunnos on vähentää määrittelyiden numeroiden tarkkuutta. Liiallinen tarkkuuden vähennys voi kuitenkin huomattavasti heikentää kuvan laatua, joten varovaisuutta tulee noudattaa. (Osmani 2018c.)

4.7.2 Rasterikuvat

Rasterikuvien optimointi on haastavampaa vektorikuviin verrattuna. Käytettävät menetelmät perustuvat käytännössä hyvän kuvaformaatin valintaan, kuvan laadun pienentämiseen, kuvan skaalaukseen sekä kuvan metadatan poistoon. (Grigorik 2018b.) Metadatan poistoa lukuun ottamatta kaikki menetelmät voivat huonontaa kuvan laatua näkyvästi, joten niiden automatisointi voi olla riskialtista.

4.8 Menetelmien saatavuus ja käyttö

Edellä esiteltyjä menetelmiä etsittiin jo olemassa olevista suosituista avoimen lähdekoodin työkaluista, joten jonkin asteinen tuki niiden käytölle löytyy jo. Menetelmiä käytetään tyypillisesti tehtävänajajiin (eng. task-runner), kuten Gulp ja Grunt, tai paketointityökaluihin (eng. bundler), kuten Webpack ja Parcel, yhdistettävillä liitännäisillä, jotka mahdollistavat koonnin eri vaiheiden suorituksen yhtenäisenä ketjuna. Työkalujen käyttöönotto vaatii usein vain pienen konfiguroinnin, jonka jälkeen menetelmän toteuttava työkalu suoritetaan osana koontiketjua.

Tässä työssä esitellyistä menetelmistä ja niiden haittavaikutuksista voidaan huomata, että monet niistä ovat vaihtokauppoja sivulataukseen vaikuttavien tekijöiden välillä. Tällaisissa tapauksissa menetelmien käyttöä on tärkeä arvioida sovelluskohtaisesti, koska menetelmän käytöllä voi olla käänteinen vaikutus sivulatauksen nopeuteen. Kuitenkaan osalla menetelmistä, kuten tarpeettomien merkkien poistolla, kuolleen koodin eliminoinnilla ja templaattien kääntämisellä ei oikein toimiessaan ole haittavaikutuksia, muuten kuin koontiketjun koontivaiheiden määrän lisääntyminen, joten erityisesti sellaisten menetelmien käyttö on usein hyödyllistä ja riskitöntä.

5 POHDINTA, TULEVAISUUS JA YHTEENVETO

Sivulatauksen hitaus on yksi tunnetuista yhden sivun verkkosovelluksen haasteista. Tässä työssä tutkittiin yhden sivun verkkosovelluksen koontivaiheen tekniikoita sivulatauksen nopeuttamiseksi. Sivulatauksen vaiheita ja aikaisempaa kirjallisuutta tutkimalla työssä löydettiin kolme sivulatausta hidastavaa tekijää, joihin koontivaiheessa voi pyrkiä vaikuttamaan: ladattavien resurssien koko ja lukumäärä sekä verkkosovelluksen suorituskyky. Menetelmiä pyrittiin havainnollistamaan koodikatkelmilla, joiden tavoite oli selkeyttää tekniikoiden toimintatapoja sanallisen ulosannin ollessa riittämätöntä. Menetelmiä esiteltiin ainoastaan korkealla tasolla ja todellisen hyödyn arviointi jätettiin lukijalle. Menetelmiä tuotiin esiin monia, joista suurin osa liittyi JavaScript-resurssien optimointiin, mutta myös muita tärkeitä verkkosovelluksissa käytettyjä resurssityyppejä käsiteltiin. Menetelmistä voitiin havaita, että sovelluksen koonnilla voidaan nopeuttaa sivulatausta, mutta osalla menetelmistä voi olla myös päinvastainen vaikutus, joten menetelmien sovelluskohtainen testaaminen on tärkeää.

Työtä voisi kehittää tutkimalla menetelmien potentiaalista hyötyä käyttämällä niitä erilaisissa sovelluksissa ja ympäristöissä, ja mittaamalla menetelmien vaikutusta eri sivulatauksen vaiheisiin. Näin menetelmien vaikutuksista ja niiden implementaation prioriteetista voisi antaa edes jonkinlaisen arvion, mitä tässä työssä ei sovellusvalikoiman ja mittausten puuttuessa pystytty tarjoamaan. Työssä kuitenkin tunnistettiin menetelmien mahdollisten hyötyjen lisäksi myös niiden haittavaikutuksia, mikä auttaa ymmärtämään tekniikoiden vaihtokaupat ja käytön seuraamukset.

Tässä tutkimuksessa pohdinta keskittyi menetelmien käyttöön loppukäyttäjän kokemuksen parantamiseksi. Resurssien koko ja lukumäärä voi kuitenkin vaikuttaa myös palveluntarjoajan kustannuksiin isännöidessä palvelua paikassa, jossa hinnoitteluun kuuluu siirrettävien objektien lukumäärä ja/tai datamäärä. Jatkotutkimusta voisi tehdä siis menetelmien käytön vaikutuksesta palvelun isännöintikustannuksiin ja sen minimoimiseksi tehtävien päätösten mahdollisista vaikutuksista palvelun sivulataukseen ja käyttökokemukseen.

Työn tutkimussuunnitelmassa yksi työlle asetetuista tavoitteista oli esitellä suosittuja työkaluja ja vertailla niiden ominaisuuksia menetelmien toteuttamiseksi. Tätä osuutta ei työssä kuitenkaan tehty, sillä työkaluja on monia ja niiden suosion määrittäminen sekä ominaisuuksien vertailu osoittautui työn aihealueen laajuudessa vaikeaksi. Työkalujen nopean kehittymisen takia vertailu ei myöskään pitkään pitäisi paikkaansa.

Yleisesti verkkosovelluksien tulevaisuus näyttää hyvin valoisalta. Koontityökalujen nopean kehittymisen lisäksi uudet WWW-tekniikat, kuten HTTP/2, Service Worker ja WebAssembly, avaavat uusia mahdollisuuksia verkkosovelluksien kehittäjille. Yhden sivun verkkosovelluksien natiivin käyttökokemuksen tavoittelu muuttuu näiden tekniikoiden myötä aina vain helpommaksi. Verkkoselaimen ollessa saatavilla lähes kaikissa laitteissa, tekee se alustariippumatonta sovellusta laatiessa verkkosovelluksesta varteenotettavan vaihtoehdon.

LÄHTEET

Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D. 2006. Compilers: Principles, Techniques, & Tools. Second Edition. Boston: Pearson Education.

Akamai. 2017. Akamai's Q1 2017 State of the Internet connectivity report. Vol 10. No. 1. Viitattu 9.10.2018. Saatavilla https://www.akamai.com/fr/fr/multimedia/documents/state-of-the-internet/q1-2017-state-of-the-internet-connectivity-report.pdf

Allen, F. E., Cocke, J. 1971. A catalogue of optimizing transformations. Yorktown, NY.

Belshe, M., Peon, R., Thomson, M. 2015. Hypertext Transfer Protocol Version 2 (HTTP/2). May 2015. Status: Proposed Standard. DOI: 10.17487/RFC7540.

Butkiewicz, M., Madhyastha, H. V., Sekar, V. 2014. Characterizing Web Page Complexity and Its Impact. IEEE/ACM Transactions of networking, Vol 22. No. 3, June 2014.

Deveria, A. 2018. Can I Use. HTTP/2 protocol feature test. Viitattu 29.7.2018. Saatavilla: https://caniuse.com/#feat=http2.

Ecma International. 2015. ECMAScript® 2015 Language Specification. Standard ECMA-262. 6th Edition / June 2015.

Galletta, D. F., Henry, R., McCoy, S., Polak, P. Web Site Delays: How Tolerant are Users? Journal of the Association for Information Systems. Vol. 5 No. 1, pp. 1-28/Jan 2004.

Grigorik, I. 2018a. Eliminating Unnecessary Downloads. Viitattu 29.7.2018. Saatavilla: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/eliminate-downloads.

Grigorik, I. 2018b. Image Optimization. Viitattu 3.8.2018. Saatavilla: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization.

Grigorik, I. 2018c. Adding Interactivity with JavaScript. Viitattu 4.8.2018. Saatavilla: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/adding-interactivity-with-javascript.

Grigorik, I. 2018d. Render-tree Construction, Layout, and Paint. Viitattu 22.8.2018. Saatavilla: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction.

Grigorik, I. 2018e. Render blocking CSS. Viitattu 22.8.2018. Saatavilla: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-blocking-css.

HTTPArchive. 2018. Report: Page Weight. Aikajakso 1.1.2018 – 1.7.2018. Viitattu 14.7.2018. Saatavilla: https://httparchive.org/reports/page-weight?start=2018_01_01&end=2018_07_01.

Kawalerowicz, M. 2012. Classification of automatic software build methods. Prace doktorantów / Články doktorandů (pp. 37–39). Opole, Poland: Opole University of Technology.

Lighthouse. 2018. Työkalu verkkosivujen auditointiin. Viitattu 29.7.2018. Saatavilla: https://developers.google.com/web/tools/lighthouse/.

Meshab, A., Deursen, A. 2007. Migrating Multi-page Web Applications to Single-page AJAX Interfaces. 11th European Conference on Software Maintenance and Reengineering, Amsterdam, 2007.

Mikowski, M., Powell, J. 2013. Single Page Web Applications: JavaScript end-to-end. First edition. Shelter Island, NY: Manning Publications.

Nah, F. 2004. A study on tolerable waiting time: how long are Web users willing to wait? Behaviour & Information Technology, Vol. 23, Issue 3. 2004.

Osmani, A. 2018a. JavaScript Start-up Optimization. Viitattu 27.7.2018. Saatavilla https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/javascript-startup-optimization/.

Osmani, A. 2018b. The App Shell Model. Viitattu 2.8.2018. Saatavilla https://developers.google.com/web/fundamentals/architecture/app-shell.

Rollup. 2018. Tree shaking. Koontityökalu. Viitattu: 4.8.2018. Saatavilla: https://rollupjs.org/guide/en#tree-shaking.

Shahzad, F. 2017. Modern and Responsive Mobile-enabled Web Applications. Science Direct. The 12th International Conference on Future Networks and Communications. Procedia Computer Science, Vol. 110.

Singhal, A., Cutts, M. 2010. Using site speed in web search ranking. Kirjoitus Googlen Webmaster Central Blogissa. Viitattu 10.6.2018. Saatavilla https://webmasters.googleblog.com/2010/04/using-site-speed-in-web-search-ranking.html

Souders, S. 2007. High Performance Web Sites. First Edition. Sebastopol: O’Reilly Media.

Souders, S. 2008. High-Performance Web Sites. Communications of the ACM. Surviving the data deluge.

Souders, S. 2009. Even Faster Web Sites. First Edition. Sebastopol: O’Reilly Media.

Spero, S. E. 1994. Analysis of HTTP Performance Problems. Viitattu 23.7.2018. Saatavilla: https://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html.

Taivalsaari, A., Mikkonen, T. 2011. The Web as an Application Platform: The Saga Continues. In: 37th EUROMICRO Conference on Software Engineering and Advanced Applications, 30 Aug. - 2 Sept. 2011, Oulu, Finland. pp. 170-174.

W3C. 2014. HTML5 – A vocabulary and associated APIs for HTML and XHTML. Viitattu: 16.7.2018. Saatavilla: https://www.w3.org/TR/2018/SPSD-html5-20180327/browsers.html#the-location-interface.

W3C. 2017a. Navigation Timing Level 2. Viitattu 4.8.2018. Saatavilla: https://www.w3.org/TR/2017/WD-navigation-timing-2-20171205/.

W3C. 2017b. Accessible Rich Internet Applications. Viitattu 2.7.2018. Saatavilla: https://www.w3.org/TR/wai-aria-1.1/#intro_ria_accessibility.

Wang, Z., Phan, D. 2018. Using page speed in mobile search ranking. Kirjoitus Googlen Webmaster Central Blogissa. Viitattu 30.6.2018. Saatavilla: https://webmasters.googleblog.com/2018/01/using-page-speed-in-mobile-search.html.

Webpack 2018a. Tree shaking. Koontityökalu. Viitattu: 4.8.2018. Saatavilla: https://webpack.js.org/guides/tree-shaking/.

Webpack 2018b. Code Splitting. Koontityökalu. Viitattu: 4.8.2018. Saatavilla: https://webpack.js.org/guides/code-splitting/.

WHATWG. 2018. HTML Living Standard. Päivitetty 24.8.2018.

LIITE 1. Työkalut

- Työkalut eri kehyksillä tehtyjen sovelluksien hallintaan
    - Create React App
    - Vue CLI
- Paketoijat, kääntäjät ja optimoijat
    - Webpack
    - Browserify
    - Rollup
    - Parcel
    - Babel
    - UglifyJS
    - Prepack
- Tehtävienajajat
    - Gulp
    - Grunt
- Staattisen sivun generaattorit
    - Gatsby
    - VuePress
- Päätön selain
    - Headless Chromium
- Sivulatauksen mittaustyökalut
    - Selainten omat kehittäjätyökalut
    - Lighthouse