Tampereen yliopisto

Henrik Hartiala

DYNAAMISTEN WEB-SOVELLUSTEN TOTEUTUS JA VERTAILU ASIAKASPÄÄN RENDEROINNILLA, PALVELINPÄÄN RENDEROINNILLA JA STAATTISEN SIVUSTON GENERAATTORIN AVULLA

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Huhtikuu 2020

TIIVISTELMÄ

Henrik Hartiala: Dynaamisten web-sovellusten toteutus ja vertailu asiakaspään renderoinnilla, palvelinpään renderoinnilla ja staattisen sivuston generaattorin avulla
Diplomityö
Tampereen yliopisto
Tietotekniikka, Diplomityö
Huhtikuu 2020

Dynaamisen web-sovelluksen toteutustavalla on iso merkitys sovelluksen suorituskykyyn ensilatauksessa ja sovelluksen sisäisessä navigoinnissa. Googlen tekemän tutkimuksen mukaan yli puolet käyttäjistä hylkää sivun, jos sen lataus kestää yli 3 sekuntia. Google myös antaa paremman sijoituksen hakutuloksissa latausnopeuden mukaan. Jos web-sovellus toteutetaan epäoptimaalisella tavalla käyttötarkoitukseensa nähden, voidaan menettää potentiaalisia käyttäjiä huonon käyttökokemuksen ja huonomman näkyvyyden takia. Web-sovellusten toteutukseen on tällä hetkellä kolme varteenotettavaa vaihtoehtoa: asiakaspään renderointi (CSR), palvelinpään renderointi (SSR) sekä staattisen sivuston generaattori (SSG).

Tässä työssä tutkittiin asiakaspään renderoinnin, palvelinpään renderoinnin sekä staattisen sivuston generaattorin avulla toteutettujen dynaamisten web-sovellusten toimintaa ja miten ne eroavat toisistaan. Työssä myös arvioidaan toteutustapojen heikkouksia ja vahvuuksia. Työ suoritettiin konstruktiivisen tutkimuksen menetelmällä toteuttamalla prototyyppisovellus kullakin edellä mainitulla toteutustavalla. Toteutustekniikkoina oli React, Next.js ja Gatsby. Toteutustapojen suorituskykyä mitattiin käyttäen Google Chrome -selaimen tarjoamaa Lighthouse-työkalua sekä suorituskykytyökalua. Näillä työkaluilla mitattiin sekä ensilatauksen suorituskykyä, että sovelluksen sisäisen navigoinnin suorituskykyä. Mitattavia suorituskyvyn avainarvoja oli neljä. Näiden avainarvojen avulla voitiin arvioida, kuinka käyttäjät kokevat sovelluksen suorituskyvyn. Toteutustapojen mittaustulosten vertailulla voitiin arvioida kunkin toteutustavan soveltuvuus eri käyttötarkoituksiin.

Mittausten tulokset osoittivat, että SSG-ratkaisulla oli paras suorituskyky ensilatauksessa, kun taas CSR-ratkaisun suorituskyky oli huonoin. CSR-ratkaisulla saatiin kuitenkin paras suorituskyky sovelluksen sisäisessä navigoinnissa. SSR-ratkaisu oli suorituskyvyltään paras tilanteessa, missä sovelluksen sisällön tulee olla ajantasalla.

Avainsanat: web-sovellus, Client-side rendering, Server-side rendering, Static site generator, React, Next.js, Gatsby

ABSTRACT

Henrik Hartiala: Dynamic web application implementation and evaluation using Client-side rendering, Server-side rendering and Static site generator
Master's Thesis
Tampere University
Information Technology
April 2020

The implementation method of Dynamic web application has a big impact on page load speed and performance. According to a research done by Google, 53% of users abandon site if the page load takes more than 3 seconds. Page load speed is also a ranking factor for searches. If web application is implemented using a nonoptimal method considering the use, users may be lost as resultant to the bad user experience. There are considerable implementation methods: Client-side rendering, Server-side rendering and Static site generator.

This master's thesis evaluates different implementation methods implementing dynamic web application. These methods are Client-side rendering, Server-side rendering and Static site generator. Main goal is to find out differences between the methods and their strengths and weaknesses. This study was done using constructive research method by implementing prototype application using each previously mentioned implementation method. React, Next.js and Gatsby were the implementation technologies chosen. Performance was measured using Lighthouse tool and performance tool provided by Google Chrome browser. With these tools the initial page load performance and performance during navigation inside the application were measured. There were four key values to measure. With these key values the perceived performance could be evaluated. Using the measurement results the suitability for different uses could be evaluated as well.

The measurement results showed that Static site generator's performance was the best in initial page load, while the Client-side rendering's performance was worst. Client-side rendering had the best performance after initial page load. Server-side rendering was overall the best in a situation where the full content of the page had to be up to date.

Keywords: web application, Client-side rendering, Server-side rendering, Static site generator, React, Next.js, Gatsby

ALKUSANAT

Halusin tehdä diplomityöni jostain web-aiheisesta aiheesta. Sopivan aiheen löytämisessä meni aikaa, mutta lopulta opiskelutoverini Tuomas ehdotti, että voisiko Gatsbyn ja Next.js:n vertailusta saada jonkinlaista aihetta. Kiinnostuin ideasta ja lähdin pallottelemaan aihetta, kunnes lopulliseksi aiheeksi muodostui tämän diplomityön aihe.

Iso kiitos työni tarkastajilleni ja ohjaajilleni Outi Sievi-Korteelle ja Kari Syställe hyvistä kommenteista ja nopeista vastauksista sähköposteihin.

Haluan myös kiittää läheisiäni, perhettäni ja erityisesti isääni motivoinnista ja tuesta työn loppuun saattamiseksi. Kiitos Tuomakselle työni aiheen alkuperäisestä ideasta. Kiitos myös kaikille opiskelutovereilleni vertaistuesta ja motivoivista keskusteluista diplomityön teon aikana.

Tampereella, 29. huhtikuuta 2020

Henrik Hartiala

SISÄLLYSLUETTELO

1 Johdanto 1
2 Web-sivujen toiminta ja teoria 3
2.1 Web-sivujen rakenne 3
2.1.1 HTML 3
2.1.2 CSS 4
2.1.3 JavaScript 5
2.2 Sivun piirto näytölle 5
2.3 Sivusto- ja skriptaustyypit 7
3 Valitut toteutustavat ja niiden teoria 9
3.1 Asiakaspään renderointi 9
3.2 React 11
3.3 Palvelinpään renderointi 11
3.4 Next.js 12
3.5 Staattisen sivuston generaattori 13
3.6 Gatsby 15
3.7 Valittujen toteutustapojen vertailu 16
4 Prototyyppisovellus ja sen toteutus 18
4.1 Konfiguraatiot 18
4.1.1 Prototyyppisovelluksen sivujen reititys 18
4.1.2 GraphQL:n käyttöönotto Next.js:ssä 19
4.1.3 GraphQL:n käyttöönotto React App:ssa 21
4.1.4 GraphQL:n käyttöönotto Gatsbyssä 22
4.1.5 Tuotesivujen generointi Gatsbyssä 23
4.2 Tiedonhaku 24
4.2.1 Tiedonhaku Next.js ja React App toteutuksissa 24
4.2.2 Tiedonhaku Gatsby toteutuksessa 26
4.3 Sisällönkoonti toteutustavoissa 27
5 Suorituskyvyn mittaus 30
5.1 Tuotesivun ensilataus (Gatsby-v1) 33
5.2 Tuotesivun ensilataus (Gatsby-v2) 34
5.3 Tuotesivulle navigointi sovelluksen sisällä (Gatsby-v1) 34
5.4 Tuotesivulle navigointi sovelluksen sisällä (Gatsby-v2) 35
6 Tulokset ja arviointi 37
7 Yhteenveto 40
Lähteet 42
References 44

Liite A Tuotesivun ensilatauksen mittaustulokset 45
Liite B Tuotesivun ensilatauksen mittaustulokset (Gatsby-v2) 46
Liite C Tuotesivun mittaustulokset navigoidessa sovelluksen sisällä 47
Liite D Tuotesivun mittaustulokset navigoidessa sovelluksen sisällä (Gatsby-v2) . . 48
Liite E Gatsby navigointi mouseover-tapahtuman jälkeen (Gatsby-v1) 49
Liite F Gatsby navigointi mouseover-tapahtuman jälkeen (Gatsby-v2) 50
Liite G Lighthouse mittauksen ajonaikaiset asetukset 51

KUVALUETTELO

2.1 DOM-puu 6
2.2 Renderointipuu 7

3.1 CSR sekvenssikaavio 10
3.2 SSR sekvenssikaavio 12
3.3 SSG sekvenssikaavio osa 1 14
3.4 SSG sekvenssikaavio osa 2 15

4.1 React App:n, Gatsbyn ja Next.js:ssän sisällönkoonti 28

TAULUKKOLUETTELO

5.1 Tuotesivun ensilatauksen mittaustulokset (Gatsby-v1) 33
5.2 Tuotesivun ensilatauksen mittaustulokset (Gatsby-v2) 34
5.3 Tuotesivulle navigointi sovelluksen sisällä (Gatsby-v1) -mittaustulokset . . . 35
5.4 Tuotesivulle navigointi mouseover-tapahtuman laukaiseman esilatauksen jälkeen (Gatsby-v1) -mittaustulokset 35
5.5 Tuotesivulle navigointi sovelluksen sisällä (Gatsby-v2) -mittaustulokset . . . 36
5.6 Tuotesivulle navigointi mouseover-tapahtuman laukaiseman esilatauksen jälkeen (Gatsby-v2) -mittaustulokset 36

6.1 Suosituksia toteutustavan valinnalle ensilatauksen suorituskyvyn ollessa tärkeä 38
6.2 Suosituksia toteutustavan valinnalle sovelluksen sisäisen navigoinnin suorituskyvyn ollessa tärkeä 39

OHJELMA- JA ALGORITMILUETTELO

2.1 HTML esimerkki 3
2.2 CSS esimerkki 4
4.1 Reititys Reactissa 19
4.2 GraphQL:n käyttöönotto Next.js:ssä 20
4.3 GraphQL:n välitys komponenteille 21
4.4 GraphQL:n käyttöönotto React App:ssa 21
4.5 Gatsby GraphQL:n käyttöönotto generointivaiheeseen liitännäisellä 22
4.6 Gatsby GraphQL käyttöönotto selaimen puolella 23
4.7 Tuotesivujen generointi Gatsbyssä 24
4.8 Next.js tiedonhaku GraphQL-kyselyllä palvelimella 25
4.9 Next.js tiedonhaku GraphQL-kyselyllä selaimessa 26
4.10 Gatsby.js tiedonhaku GraphQL-kyselyllä 27
5.1 Gatsby tuotesivun muutos 32

LYHENTEET JA MERKINNÄT

API	Application Programming Interface
CSR	Client-Side Rendering
CSS	Cascading Style Sheet
CSV	Comma Separated Values
DOM	Document Object Model
FCP	First Contentful Paint
HMR	Hot Module Replacement
HOC	Higher Order Component
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
LCP	Largest Contentful Paint
SPA	Single Page Application
SSG	Static Site Generator
SSR	Server-Side Rendering
TTI	Time to Interactive
URL	Uniform Resource Locator

1 JOHDANTO

1990-luvun alussa HyperText Markup Language (HTML) oli ainoa kieli web-sivujen luontiin [37]. Tällöin web-sivuston jokaisella sivulla oli oma staattinen HTML-tiedosto palvelimella. Web-teknologioiden kehityttyä web-sivustoihin pystytään liittämään selaimessa tai palvelimella suoritettavaa ohjelmakoodia, jolla sivuston rakennetta ja sisältöä voidaan muokata dynaamisesti. Ohjelmakoodia palvelimella tai selaimessa suorittavaa web-sivua kutsutaan usein myös web-sovellukseksi. Suoritettavan ohjelmakoodin avulla sivun sisältö pystytään rakentamaan kokonaan selaimessa. Tällaista tapaa, missä muokataan yhden HTML-tiedoston sisältöä haluttuun muotoon, kutsutaan *Single Page Applicationiksi* (SPA) [4].

Single Page Application on ollut viime vuosina suosittu tapa toteuttaa web-sivuja. Perinteisellä web-sivulla palvelin vastaa jokaiseen reittipyyntöön omalla HTML-tiedostolla, mutta SPA-ratkaisussa palvelin vastaa yhdellä HTML-tiedostolla, jonka sisältö muokataan reittipyyntöä vastaavaksi.

Asiakaspään renderoinnilla (CSR) toteutetussa SPA:ssa palvelimen palauttama HTML-sivu on aluksi tyhjä, tai se sisältää jonkin latausindikaattorin. Haluttu sisältö tähän HTML-sivuun koostetaan JavaScriptin avulla selaimessa. Palvelinpään renderoinnilla (SSR) toteutetussa SPA:ssa sisältö koostetaan jo palvelimella, ennen HTML:n lähettämistä selaimelle.

Näiden kahden rinnalla on myös kolmas varteenotettava vaihtoehto toteuttaa web-sivuja: staattisten sivujen generaattori (SSG). Sen avulla sivuston kaikista sivuista voidaan mallien avulla generoida HTML-tiedostot, jotka tarjoillaan palvelimelta. Käyttäjän navigoidessa sivulle, palvelin etsii ja lähettää pyydetyn sivun ennalta generoidun HTML:n selaimelle.

Web-sovelluksista halutaan saada mahdollisimman tehokkaita, koska tehokas web-sovellus parantaa käyttökokemusta ja sillä voi myös olla merkittäviä vaikutuksia liiketoimintaan. On tehty monia tutkimuksia, joiden mukaan sivujen latausnopeus vaikuttaa esimerkiksi liikevaihtoon [2]. Vuonna 2016 Googlen tekemän tutkimuksen mukaan 53% käyttäjistä poistuu web-sivulta, jos sen lataaminen kestää yli 3 sekuntia [12]. Googlen hakutuloksien sijoitukseen vaikuttaa myös sivun latausnopeus [17]. Näin Google palvelee käyttäjiä näyttämällä nopeasti lataavat sivut paremmalla sijoituksella.

Tässä työssä vertaillaan ja arvioidaan dynaamisia web-sovelluksia toteutettuina CSR:llä, SSR:llä ja SSG:n avulla. React-pohjaisella teknologialla voidaan rakentaa dynaaminen web-sovellus jokaisella edellä mainitulla toteutustavalla, joista SSR-ratkaisu toteutetaan

käyttäen Next.js sovelluskehystä ja SSG-ratkaisu toteutetaan käyttäen Gatsby.js sovelluskehystä. CSR-ratkaisu toteutetaan käyttäen React-kirjastoa.

Työn tavoitteena on saada selkeä kuva siitä, miten eri toteutustavat eroavat toisistaan, mitkä ovat toteutustapojen vahvuudet ja heikkoudet ja miten toteutustavat soveltuvat eri käyttötarkoituksiin. Työssä toteutetaan prototyyppisovellus em. toteutustavoilla, vertaillaan niiden suorituskykyä Google Chrome -selaimen kehitystyökaluilla ja arvioidaan toteutustapojen soveltuvuutta eri käyttötarkoituksissa. Työ keskittyy web-sovelluksiin, jotka suorittavat myös palvelinpuolen koodia ja täten täysin staattisten sivujen arviointi jää tästä työstä ulkopuolelle.

Työn tutkimusmenetelmänä on konstruktiivinen tutkimus, sillä menetelmä soveltuu hyvin työn ongelman tutkimiseen. Jotta eri toteutustapoja voidaan tarkasti vertailla, tarvitsee samasta prototyypistä kehittää jokaiselle toteutustavalle vastaava versio.

Tämä työ koostuu seitsemästä luvusta. Toisessa luvussa käydään läpi web-sivujen rakennetta, toimintaa sekä teoriaa. Kolmannessa luvussa kerrotaan tarkemmin mitä selainpään renderointi, palvelinpään renderointi sekä staattisten sivustojen generaattori tarkoittaa ja mikä niiden toimintaperiaate on. Lisäksi kolmannessa luvussa esitellään valitut tekniikat prototyyppisovelluksen toteutukselle. Neljäs luku sisältää prototyyppisovelluksen kuvauksen, avainosia konkreettisesta toteutuksesta sekä lisäselvennystä eri toteutustapojen sisällönkoonnista. Viidennessä luvussa on esitelty suorituskyvyn mittaustekniikat, mittaukset sekä mittaustulokset. Kuudennessa luvussa käydään tulokset läpi, sekä arvioidaan niiden perusteella miten eri toteutustavat soveltuvat eri käyttötarkoituksiin. Seitsemäs luku sisältää yhteenvedon työstä.

2 WEB-SIVUJEN TOIMINTA JA TEORIA

Tässä luvussa käydään läpi web-sivujen pääelementit ja niiden toiminta sekä yhteys toisiinsa. Luvussa esitellään myös vaihe vaiheelta, mitä toimenpiteitä selain tekee piirtääkseen web-sivun sisältö käyttäjän näytölle. Lopuksi kerrotaan dynaamisen- ja staattisen sivuston eroja, sekä mitä asiakaspään skriptauksella ja palvelinpään skriptauksella tarkoitetaan.

2.1 Web-sivujen rakenne

Tavanomainen verkkosivu koostuu HTML:stä, CSS:stä (Cascading Style Sheets) ja JavaScriptistä. Näistä HTML ja CSS ovat deklaratiivisia ohjelmointikieliä ja vastaavat sivun esityksen rakenteesta ja tyylistä. 1990-luvun alkupuolella HTML oli ainoa kieli verkkosivujen luontiin [37]. Web-kehittäjien täytyi kirjoittaa sivuston jokainen sivu omana HTML-tiedostona. Jos tällaisella sivustolla oli käytössä esimerkiksi navigointipalkki ja navigointiin tuli joku muutos, niin muutos täytyi tehdä jokaiseen tiedostoon erikseen. Nykyään verkkosivujen luonti on paljon helpompaa ohjelmointikielten, kuten JavaScriptin yleistyttyä verkkosivujen moottorina.

2.1.1 HTML

HTML on jokaisen web-sivun ”ydin” riippumatta käytettyjen teknologioiden määrästä tai sivuston monimutkaisuudesta. HTML on nimensä mukaisesti merkintäkieli missä erityyppiset sisällöt ja komponentit merkataan erilaisilla merkeillä, eli tunnuksilla. Tästä esimerkkinä listaus 2.1.

```
1 <html>
2     <head>
3         <title>Esimerkkinimike</title>
4     </head>
5     <body>
6         <div>
7             <h1>Otsikko</h1>
8             <p>Esimerkkikappaleen teksti.</p>
9         </div>
```

```
10          <div>
11              <h1>Toinen otsikko</h1>
12              <ul>
13                  <li>Ensimmainen</li>
14                  <li>Toinen</li>
15              </ul>
16          </div>
17      </body>
18  </html>
```

Listaus 2.1. *HTML esimerkki*

Erityyppiset sisällöt laitetaan niille kuuluvien tunnusten sisään, kuten esimerkin 2.2 rivi 3 missä sivun nimi on laitettu avaavan tunnuksen *<title>* ja vastaavan sulkevan tunnuksen *</title>* sisään (samanlainen kuin avaava tunnus, mutta avainsanan edessä on kauttaviiva) ja tunnusten välissä oleva data on siis tämän tunnuksen sisältöä. Tunnukset voivat olla sisäkkäin, kuten *head*-tunnuksen (sivun metadata) sisällä löytyvä nimi (selainikkunan yläreunaan tuleva teksti). *Body*-tunnusten sisälle laitetaan itse sivun sisältö. Esimerkissä olevat tunnukset *div* (division) edustaa lohkoa, *h1* (heading) otsikkoa, *p* (paragraph) kappaletta, *ul* (unordered list) järjestelemätöntä listaa ja *li* (list item) listan alkiota. Kaikki edellä mainitut ovat *html*-tunnusten sisällä, minkä avulla selain tunnistaa, että kyseessä on html-dokumentti. [20]

2.1.2 CSS

Siinä missä HTML määrittelee verkkosivun sisällön rakenteen, CSS määrittelee miten HTML:n elementit lopulta esitetään käyttäjälle [19]. CSS:llä voidaan siis määrittää sisällön väriä, kokoa, fontteja ynnä muita. joilla voidaan muuttaa täysin sivun ulkoasua, teemaa ja täten tunnelmaa. CSS:llä voidaan myös määrittää laitekohtaiset tyylit, jotta sivu skaalautuu esimerkiksi puhelimelle.

CSS on siis lista sääntöjä, joita voidaan määrittää HTML -tunnuksille. Säännöt voidaan määrittää joko yksittäisille tunnuksille, ryhmille, kokonaiselle dokumentille tai useille dokumenteille. Nämä ovat eriytetty HTML:stä omaan tiedostoon (tiedostopääte .css), jotta ”vastuualueet” eivät sekoitu; HTML vastaa sivun rakenteesta ja CSS vastaa sivun esitystavasta. Eri selaimilla on myös omat oletustyylit, joita käytetään, jos kehittäjä ei ole määritellyt omaa tyyliä.

```
1  body {
2      background-color: #ddd;
3  }
4
5  div {
6      margin: 16px;
```

```
7       padding: 8px;
8       background-color: #aaa;
9   }
10
11  h1 {
12      color: #414042;
13  }
```

Listaus 2.2. *CSS esimerkki*

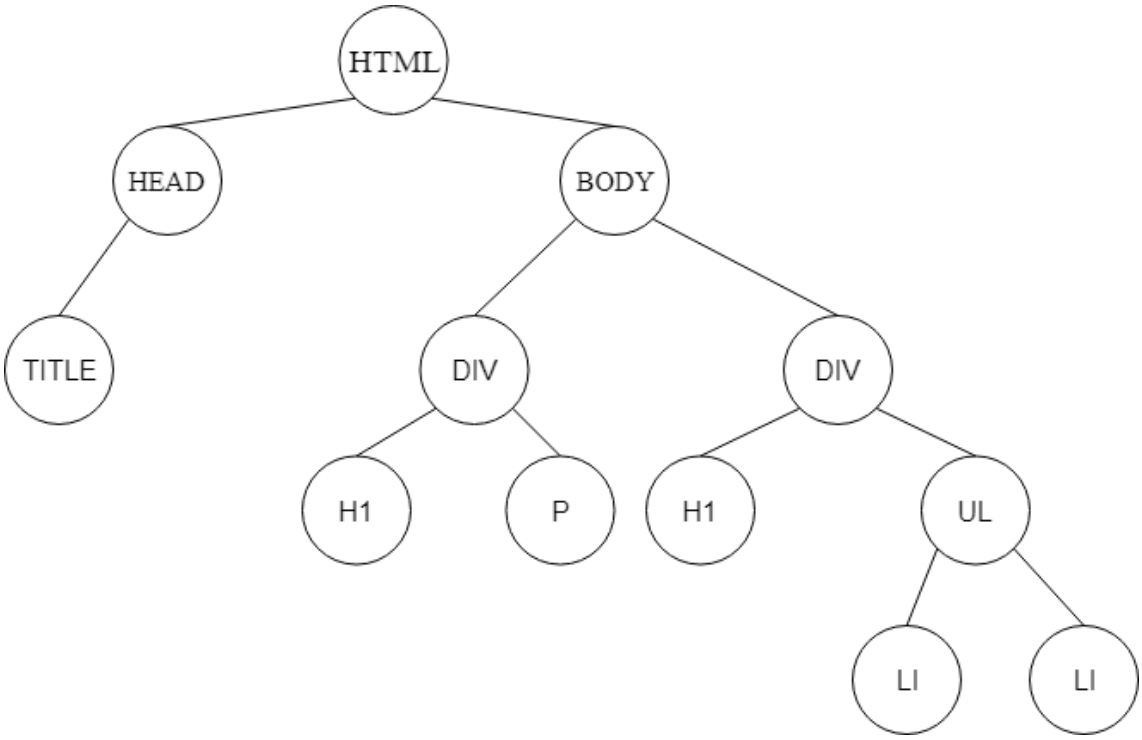
Esimerkissä 2.2 on määritelty tyyliohjeita kolmelle HTML-tunnukselle; *body*, *div* ja *h1*. Body-tunnukselle on määritelty taustan väri *background-color*, joka pätee kaikkialle body-tunnuksen sisällä olevalle sisällölle. Samoin div-elementille annetut background-color, margin ja padding pätevät kaikille div-elementtien sisällöille. Sisemmän elementin tyyli ylikirjoittaa ulomman elementin tyylit jos ne on määritelty [22].

2.1.3 JavaScript

JavaScript on skriptauskieli, jota voidaan suorittaa nykyään lähes jokaisessa selaimessa. JavaScriptin avulla on mahdollista tehdä sivuista toiminnallisesti erittäin monipuolisia ja interaktiivisia. JavaScript on prototyyppipohjainen, moniparadigmainen, yksisäikeinen dynaaminen ohjelmointikieli, joka tukee oliokeskeisiä, imperatiivisia ja deklaratiivisia tyylejä [21]. JavaScriptin avulla voidaan muokata dynaamisesti sivun sisältöä muokkaamalla suoraan esitettävää HTML:ää tai sen tyylejä. JavaScript mahdollistaa myös reagoimisen käyttäjän syötteisiin, kuten hiiren tai näppäimistön painalluksiin ja kursorin liikkeisiin. Vaikka JavaScript on tunnettu parhaiten web-sivujen skriptauskielenä, voidaan sitä suorittaa myös muualla kuin selainympäristössä [18]. Web-sovelluksen asiakaspuoli sekä palvelinpuoli voidaan siis rakentaa käyttäen pelkästään JavaScriptiä.

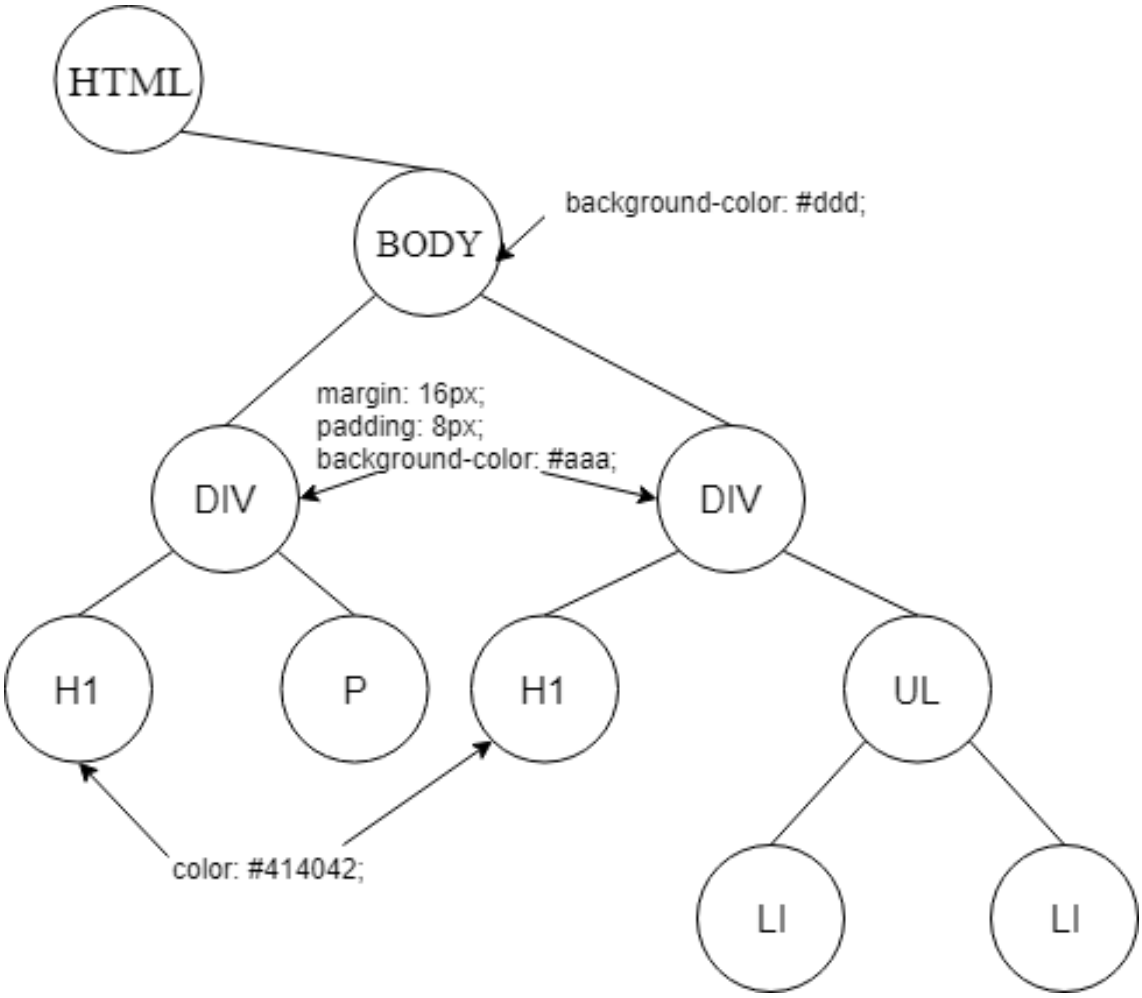
2.2 Sivun piirto näytölle

Käyttäjän navigoidessa web-sivulle, täytyy selaimen tehdä erinäisiä toimenpiteitä voidakseen näyttää haluttu sisältö käyttäjälle. Ensimmäisenä selain tekee HTTP-pyynnön (HyperText Transfer Protocol) kyseiseen osoitteeseen. Osoitteen palvelin vastaa pyyntöön tarvittavilla tiedostoilla, kuten esimerkin 2.1 HTML:llä ja esimerkin 2.2 CSS:llä. Selain parsii HTML:n ja aloittaa piirtoprosessin.



Kuva 2.1. *DOM-puu*

Piirtoprosessin ensimmäinen vaihe on rakentaa HTML:stä DOM-puun (Document Object Model), kuten kuvassa 2.1. [16] DOM-puun rakentamisen jälkeen selain rakentaa CSSOM-puun (CSS Object Model), mikä on kuten DOM-puu, mutta CSS:lle. Kun DOM ja CSSOM on rakennettu, selain yhdistää nämä keskenään ja muodostaa renderointipuun, kuten kuvassa 2.2.



Kuva 2.2. *Renderointipuu*

Renderointipuun rakentaminen alkaa DOM:in juuresta käymällä läpi jokainen solmu. Käydessään läpi DOM:ia, se jättää huomiotta sellaiset solmut, jotka eivät ole varsinaista käyttäjälle piirrettävää sisältöä. Tällaisia ovat esimerkiksi metatietoa sisältävät solmut, skriptaussolmut ja sellaiset solmut, joiden CSS sääntönä on ”display: none”. Jokaiselle näytettävälle solmulle haetaan oikeat CSS säännöt CSSOM:ista. Kun tyylisäännöt ovat kohdallaan, laskee selain kuinka paljon elementit vievät tilaa näytöltä ja missä ne sijaitsevat. Viimeisenä vaiheena on itse piirtovaihe missä sisältö piirretään pikseli pikseliltä näytölle. Jos DOM:ia tai CSSOM:ia muokataan, täytyy kaikki piirtoprosessin vaiheet käydä läpi alusta.

2.3 Sivusto- ja skriptaustyypit

Staattisella sivustolla tarkoitetaan tässä työssä sellaisia sivuja, jotka eivät suorita palvelinpuolen ohjelmakoodia ollenkaan. Kun selain tekee pyynnön tällaiselle sivulle, webpalvelin vain lähettää tarvittavat tiedostot selaimelle. Tällaisia tiedostoja ovat mm. HTML, CSS, JavaScript ja kuvat. Pääosin staattiset sivut koostuvat siis HTML:stä, CSS:stä ja

JavaScriptistä, mutta ei välttämättä kaikista näistä. Yksinkertaisimmillaan staattiset sivut koostuvat pelkästä HTML:stä, mutta yleensä vähintään CSS-tyylit ovat mukana. JavaScriptin avulla voidaan tuoda toiminnallisuutta ja interaktiivisuutta staattisille sivuille. Koska määritelmän mukaisesti staattisella sivustolla ei ole käytössä tietokantoja tai palvelinpuolen ohjelmointia, ovat staattiset sivut kaikille käyttäjille samanlaiset.

Pelkästään HTML:stä ja CSS:stä kootun sivuston hyvät puolet ovat nopeus. Koska selaimen pyyntöön vastataan välittömästi HTML-sivulla ja selain voi piirtää sisällön heti näkyviin. Pienille sivustoille tämä on hyvä ja helppo tapa, mutta sivuston kasvaessa ylläpidosta tulee reilusti työläämpää.

Dynaamisella sivustolla tarkoitetaan tässä työssä sellaista sivustoa, joka suorittaa palvelinpuolen koodia. Näin ollen käyttäjille saadaan haettua dynaamisesti dataa esimerkiksi tietokannasta. Dynaamisilla sivustoilla sisältö voi poiketa riippuen käyttäjästä. Esimerkiksi kun käyttäjä kirjautuu sisään, palvelin hakee käyttäjän tiedot tietokannasta ja esittää ne käyttäjälle. Twitter ja Facebook ovat hyviä esimerkkejä dynaamisista sivustoista. Näillä sivuilla käyttäjät voivat ladata kuvia, kommentoida ja tehdä erinäisiä toimintoja, jotka vaikuttavat sivuston sisältöön.

Asiakaspään skriptauksella tarkoitetaan ohjelmakoodia (yleensä JavaScript), jonka prosessointi suoritetaan selaimessa käyttäjän tietokoneella [33]. Käyttäjän tehdessä pyynnön sivustolle, lähettää palvelin JavaScript-tiedostot selaimelle. Selain parsii JavaScriptin ja alkaa suorittamaan sitä. Asiakaspään skriptauksen käyttötarkoituksena web-ohjelmoinnissa on tuottaa interaktiivisia ja responsiivisia sivuja. Sen avulla voidaan esimerkiksi muuttaa sisältöä dynaamisesti painikkeen painalluksesta ja tehdä pyyntöjä palvelimelle ja käsitellä pyynnöstä saatua vastausta. Tietoa voidaan myös tallentaa selaimen välimuistiin.

Palvelinpään skriptauksella tarkoitetaan sellaista ohjelmakoodia, jonka prosessointi suoritetaan palvelimella. Sen tyypillisiin tehtäviin kuuluu käsitellä selaimelta saatu pyyntö ja vastata siihen tarkoituksen mukaisesti. Pyynnön käsittelyyn voi sisältyä esimerkiksi tietokantayhteyden muodostamisen, tiedonhaku tietokannasta ja tiedon muokkaamisen haluttuun muotoon. [33].

3 VALITUT TOTEUTUSTAVAT JA NIIDEN TEORIA

Asiakaspään renderoinnin, palvelinpään renderoinnin ja staattisen sivuston generaattorin toteutustapojen vertailuksi tässä työssä toteutetaan prototyyppisovellus kullakin toteutustavalla. Prototyyppisovellus toteutetaan käyttäen React-kirjastoa, Next.js-sovelluskehystä sekä Gatsby-sovelluskehystä. Tässä luvussa esitellään valittuja toteutustapoja, sekä kerrotaan niiden teoriaa. Koska kaikkien toteutustapojen asiakaspää toteutetaan Reactilla, voidaan komponenttien koodit jakaa toteutusten välillä lähes sellaisinaan.

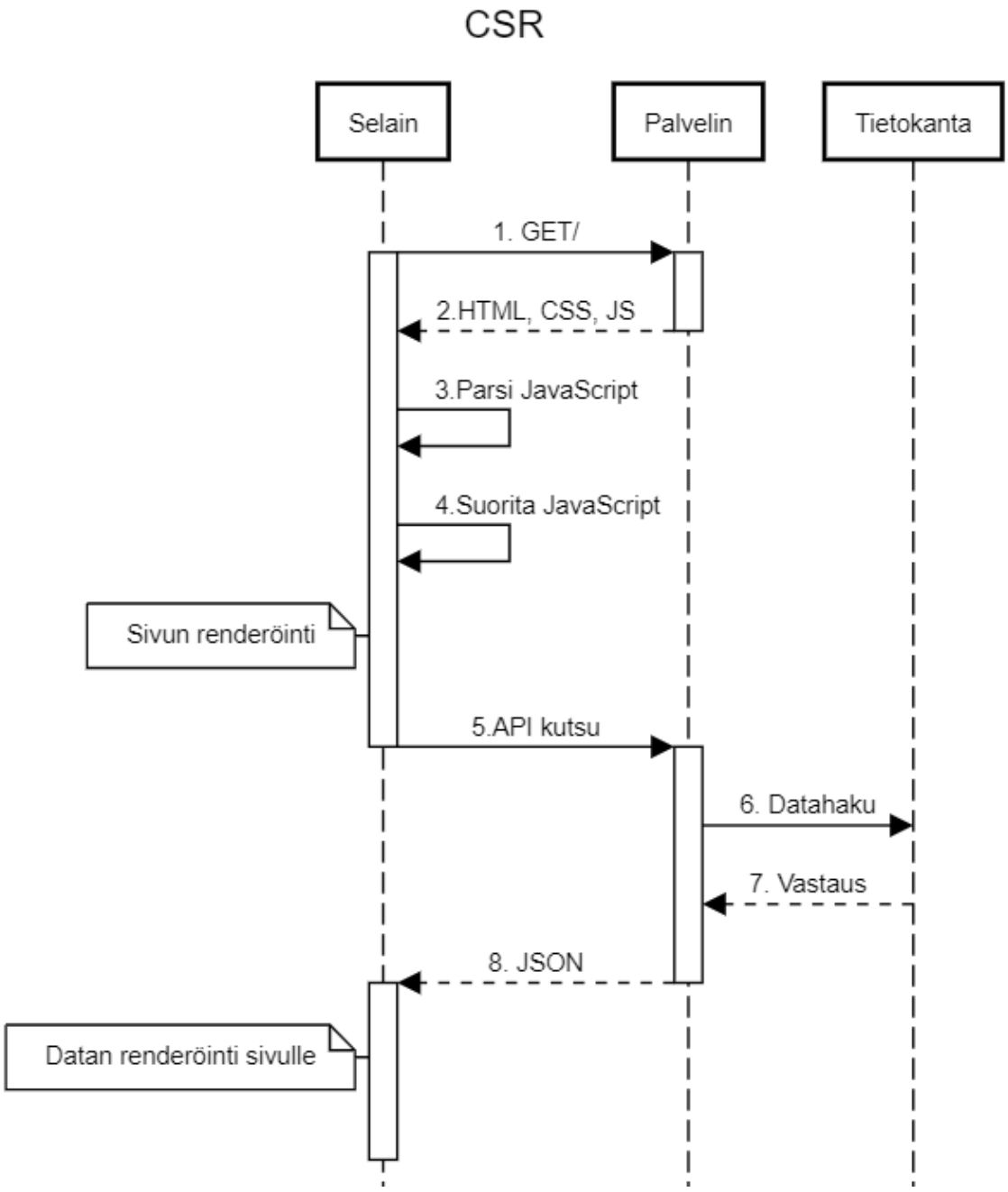
3.1 Asiakaspään renderointi

Asiakaspään renderoinnilla tarkoitetaan sitä, kun web-sivun reititys ja koostaminen tapahtuu kokonaisuudessaan selaimessa. Sivuston logiikka, tiedonhaku, mallit ja reititys ovat kaikki selaimen vastuulla.[15]

CSR-ratkaisussa selain tekee pyynnön palvelimelle ja palvelin palauttaa joko tyhjän tai latausanimaatiollisen HTML-tiedoston. Palvelin lähettää samalla muut tarvittavat tiedostot, kuten JavaScript-tiedostot. Kun tiedostot on saatu lähetettyä, aloittaa selain parsimaan JavaScriptiä. Parsimisen jälkeen koodi voidaan suorittaa ja haluttu sisältö voidaan luoda HTML:ään ja piirtää käyttäjälle näkyviin.

Toisin sanoen CSR on JavaScriptin suoritusta selaimessa, mikä tuottaa HTML:ää sen sijaan, että palvelin palauttaisi valmiin HTML:n selaimeen. Etuna tässä tavassa on se, että käyttäjän tehdessä toimintoja muutokset voidaan näyttää käyttäjälle välittömästi, ilman että selaimen täytyisi suorittaa pyyntö palvelimelle kysyäkseen mitä näyttää käyttäjälle.

Kuvassa 3.1 on sekvenssikaavio asiakaspään renderoinnin toteutumisesta käyttäjän navigoidessa sivulle. Selain tekee pyynnön palvelimelle ja palvelin lähettää selaimelle pyyntöä vastaavat tiedostot. Selain parsii JavaScriptin ja alkaa suorittamaan sitä. Tässä vaiheessa sivusto renderoidaan ensimmäistä kertaa käyttäjän näytölle, mutta se ei välttämättä sisällä kaikkea tarvittavaa tietoa. Kuvassa oleva esimerkkisovellus tekee API-kutsun palvelimelle (joka tässä tapauksessa voi olla mikä tahansa muukin palvelin kuin se, jolla sivustoa ylläpidetään), joka hakee pyydetyn datan tietokannasta ja palauttaa vastauksen selaimelle, yleensä JSON-muodossa (JavaScript Object Notation). Selain rikastaa sivun haetulla datalla ja lopullinen sivu on valmis renderoitavaksi käyttäjän näytölle. API-kutsut suoritetaan asynkronisesti, jolloin sovellus voi toimia estottomasti myös tiedonhaun aikana.



Kuva 3.1. *CSR sekvenssikaavio*

Sekvenssi kuitenkin muuttuu käyttäjän navigoidessa kuvan 3.1 mukaisen tilanteen jälkeen, sillä sivun reititys on ensimmäisen latauksen jälkeen siirtynyt JavaScriptin vastuulle. Käyttäjän navigoidessa toiselle sivulle selaimessa suoritettava JavaScript tekee muutokset heti näytölle, jonka jälkeen suorittaa tarvittavat API-kutsut (Application Programming Interface) mahdolliselle datahaulle ja renderoi sivun käyttäjälle. Sivun reititystä tai muutosta ei siis haeta palvelimen kautta.

3.2 React

React on Facebookin kehittämä deklaratiivinen, komponenttipohjainen JavaScript-kirjasto web-sovellusten käyttöliittymien toteuttamista varten [32]. Reactin komponenttipohjaisuus mahdollistaa monimutkaistenkin käyttöliittymien toteutuksen jaon pienempiin kokonaisuuksiin eli komponentteihin, jotka hallinnoivat omaa tilaansa. React käyttää virtuaalista DOM:ia, minkä avulla näkymän päivittäminen ja juuri oikeiden komponenttien päivittäminen näytölle datan muuttuessa on tehokasta.

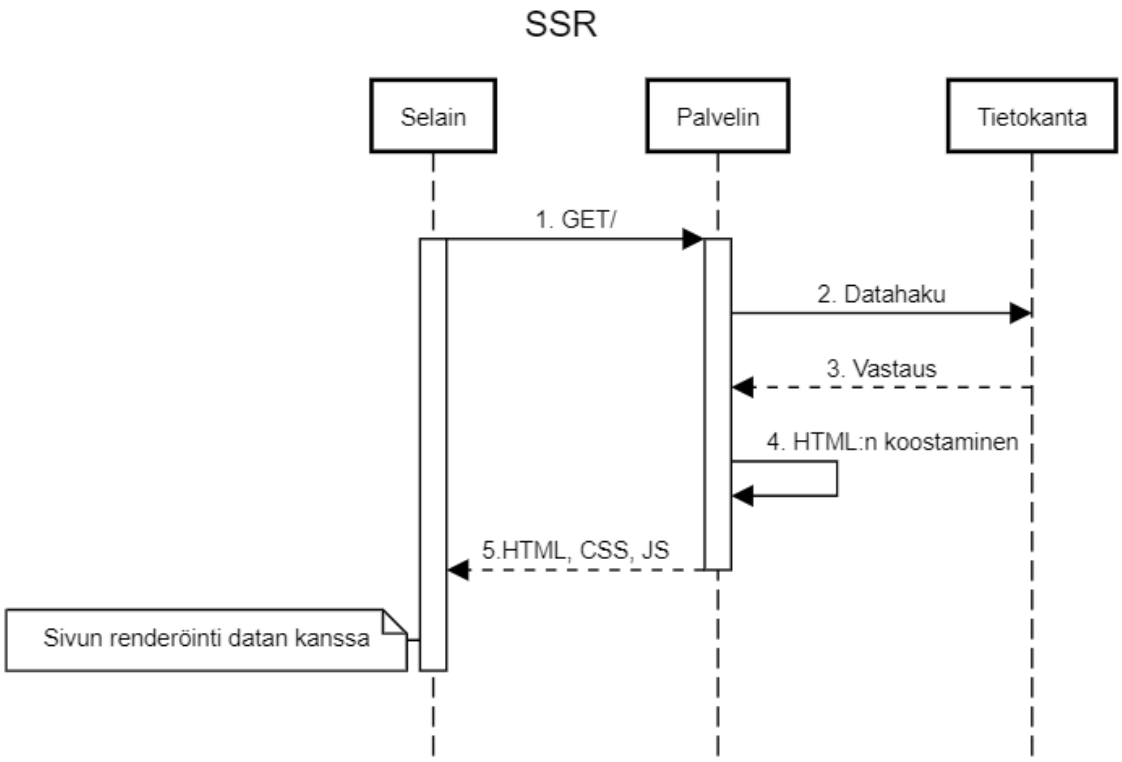
Modernissa interaktiivisessa webissä DOM:ia muokataan ohjelmallisesti, yleensä JavaScriptillä. DOM:in manipulointi on kuitenkin paljon hitaampaa kuin suurin osa JavaScript-operaatioista. DOM:in kaikille muutoksille täytyy tehdä kappaleessa 2.2 esitelty piirtoprosessi. Suurin osa JavaScript-sovelluskehyksistä pahentaa tilannetta sillä, että ne päivittävät DOM:ia paljon enemmän kuin tarvitsisi. Esimerkiksi uuden kohteen lisääminen listan perälle toteutetaan monessa JavaScript-sovelluskehyksessä rakentamalla koko lista uusiksi, vaikka siihen tarvitsisi lisätä vain yksi uusi kohde [3]. Näin ollen piirtoprosessin vaiheet käydään läpi monta kertaa täysin turhaan. Moderneilla websivustoilla saatetaan tehdä todella paljon DOM-manipulaatiota ja tämä epätehokas tapa päivittää DOM:ia muodostuu todelliseksi ongelmaksi. React esitteli tähän ongelmaan ratkaisuksi konseptin nimeltä *Virtual DOM*.

Reactin Virtual DOM:issa jokaista DOM-objektia vastaa virtuaalinen kopio. Nämä kopiot ovat muuten samanlaisia ominaisuuksiltaan kuin oikea DOM-objekti, mutta ne eivät itse muuta mitä näytöllä näkyy, eikä niiden muokkaaminen aiheuta uudelleenpiirtoa näytölle. Tästä syystä virtuaalisen DOM:in muokkaaminen on kevyempää ja nopeampaa. Kun virtuaalinen DOM on saatu päivitettyä, verrataan sitä aikaisempaan virtuaaliseen DOM:iin ja näin saadaan laskettua paras mahdollinen tapa tehdä muutokset oikeaan DOM:iin [23].

3.3 Palvelinpään renderointi

Palvelinpään renderointi tarkoittaa sivun koostamista sisältöineen palvelimen puolella vastauksena käyttäjän navigointiin ja toimintoihin [15]. SSR:n toiminnassa selain tekee pyynnön palvelimelle, joka lähettää vastauksena HTML-tiedoston missä on jo kyseisen sivun sisältöä toisin kuin CSR-ratkaisussa. Tällöin riippuen toteutuksesta joko koko sivu tai osa sivusta piirtyy käyttäjälle nopeasti pyynnön jälkeen. Näin käyttäjän ei tarvitse katsella tyhjää sivua tai latausanimaatiota sillä välin, kun selain lataa, parsii ja suorittaa JavaScriptiä. Käyttäjä voi suorittaa toimintoja, kun JavaScript on parsittu ja suorituksessa. SSR-ratkaisulla voidaan käyttäjälle antaa siis merkityksellistä tekstiä tai muuta sisältöä sillä välin, kun muita latauksia tehdään.

Kuvassa 3.2 on sekvenssikaavio palvelinpään renderoinnin toimintaperiaatteesta. Selaimen tehdessä pyynnön palvelimelle, alkaa palvelin suorittamaan palvelinpuolen ohjelmakoodia, yleensä JavaScriptiä. Palvelin tekee tarvittavat datahaut tietokannasta (tai ulkoi-



Kuva 3.2. *SSR sekvenssikaavio*

silta palvelimilta) ja koostaa valmiin HTML:n lähetettäväksi selaimelle. Selain saa valmiin HTML:n, jonka se voi renderoida käyttäjälle. Käyttäjän navigoidessa tämän jälkeen toiselle sivulle, suoritetaan uudestaan kuvan 3.2 mukainen sekvenssi.

3.4 Next.js

Next.js on Zeit-nimisen yhtiön luoma sovelluskehys sekä palvelinpään renderoiduille React-sovelluksille, kuin myös staattisille sivuille [39]. Next hoitaa suurimman osan konfiguraatioista ja optimoinneista kehittäjän puolesta, minkä takia kehittäjän ei tarvitse niistä huolehtia. Nextissä on intuitiivinen sivujen reititys, missä kohteen URL muodostuu kansiorakenteen perusteella, eikä kehittäjän näin ollen tarvitse huolehtia dynaamisesta reitityksestä. Dynaaminen reititys on kuitenkin mahdollista toteuttaa Nextissä [28].

Next.js tunnistaa datariippumattoman sivun ja luo siitä staattisen tiedoston. Tämän ominaisuuden avulla Next.js voi tuottaa ns. hybdirisovelluksia, mitkä sisältävät sekä palvelinrenderoituja sivuja, että staattisesti generoituja sivuja. Tämä nopeuttaa käyttäjälle lähetettyjä sivuja, sillä ne eivät vaadi laskentaa palvelimen puolella ja voidaan täten lähettää suoraan käyttäjän selaimeen [24].

Next.js tekee automaattista koodin jaottelua (code splitting) sovelluksen sivujen perusteella. Jos jotain komponenttia käytetään yli puolissa sivuista, siirretään se tärkeimpään JavaScript-pakettiin. Muissa tapauksissa se pysyy kyseisen sivun omassa JavaScript-paketissa [29]. Tällä jaottelulla saadaan usein käytettyjen komponenttien JavaScriptit la-

dattua jo ensimmäisen JavaScript-paketin mukana, sen sijaan että ne ladattaisiin jokaisen komponenttia käyttävän sivun kohdalla erikseen. Näin sivujen lataaminen nopeutuu.

Next.js tarjoaa myös sivujen ennalta hakua (prefetch). Next.js käyttää sovelluksen sisäiseen navigointiin *<Link>* -komponenttia. Tämä komponentti lataa automaattisesti sen kohteena olevan sivun JavaScript-tiedoston. Näin navigoidessa kyseiselle sivulle saadaan sivu piirrettyä käyttäjälle nopeasti. [27]

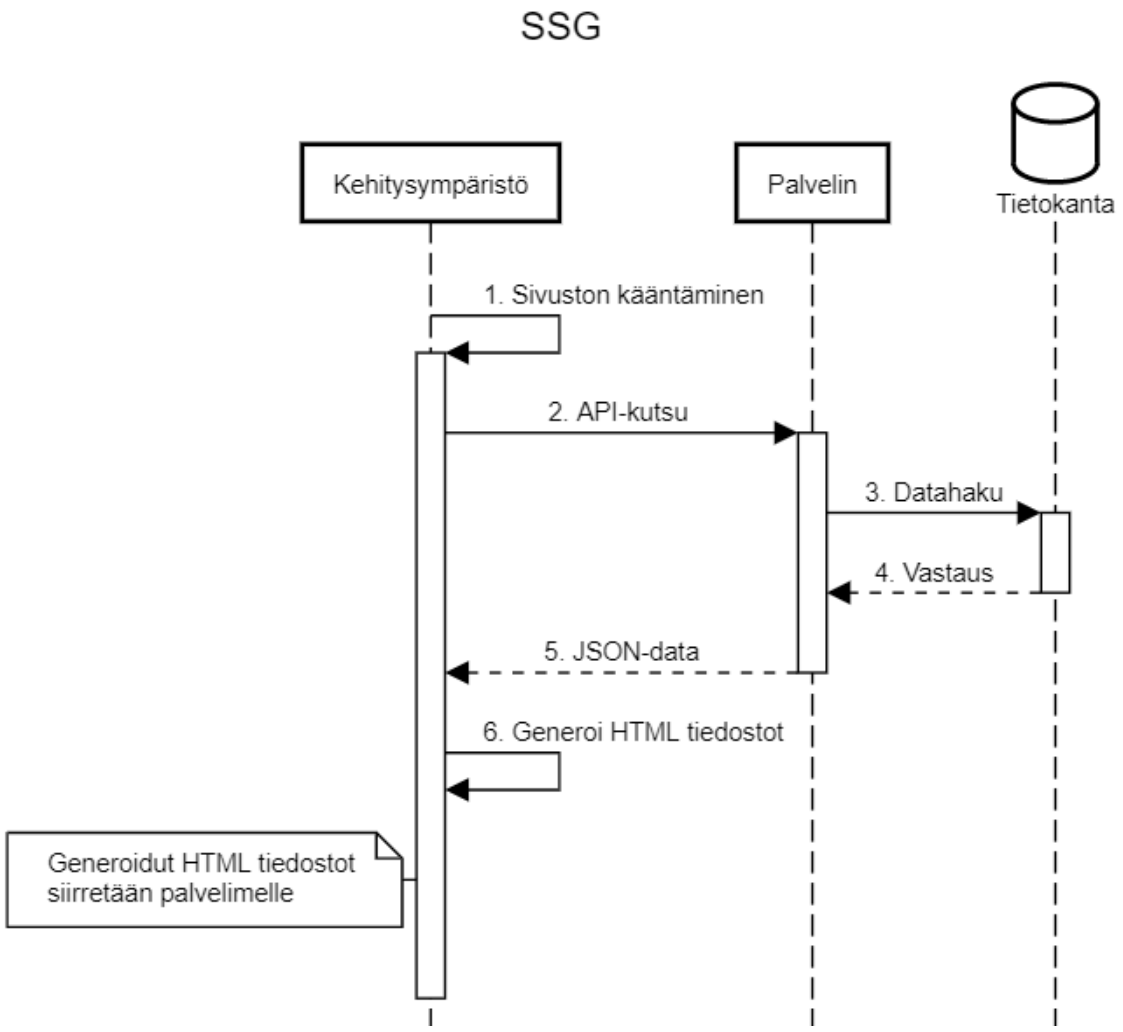
Next.js:sä on webpack-pohjainen kehitysympäristö, mikä tukee HMR:ää (Hot Module Replacement)[28]. HMR:än avulla moduuleita voidaan lisätä, vaihtaa tai poistaa sovelluksen ollessa käynnissä, ilman täyttä sivun päivitystä. Kehittäjän työ voi nopeutua täten huomattavasti. Kehittäjän tehdessä muutoksia sivun koodiin päivittyy muutokset välittömästi sivulle ilman, että sivu menettää tilaansa. Päivitys tapahtuu vain niihin, mitä on muutettu. Myös CSS-tiedostoihin tehdyt muutokset päivittyvät heti [36].

3.5 Staattisen sivuston generaattori

Staattisen sivuston generaattorilla tarkoitetaan tässä työssä sellaista työkalua, jolla kehittäjä voi rakentaa palvelinpohjaisen sivuston lokaalisti käyttäen reititystä, malleja ja tiedonhakua. Kun sivusto on valmis, generoidaan julkaisua varten sivuston jokaisesta sivusta staattiset tiedostot. Näin voidaan siis säilyttää tietoa esimerkiksi tietokannoissa ja rakentaa palvelimella julkaistava sivusto, jonka ei tarvitse tukea palvelinpuolen ohjelmointia. Sivujen hallinnoija voi lisätä tietokantaan tai muuhun sisällönhallintajärjestelmään tietoa ja suorittaa sivustosta uusi generointi.

Tämä lähestymistapa ei kuitenkaan poissulje suorituksenaikaista palvelinpuolen ohjelmointia. Kehittäjä voi rakentaa sivuston, joka generoinnin jälkeen toimii kuten dynaaminen sivusto. Sivusto voi siis myös generoinnin jälkeen luoda tietokantayhteyden ja noutaa dynaamisesti tietoa käyttäjälle näytettäväksi. Tällä yhdistelmällä voidaan saada todella tehokkaita sivustoja aikaiseksi.

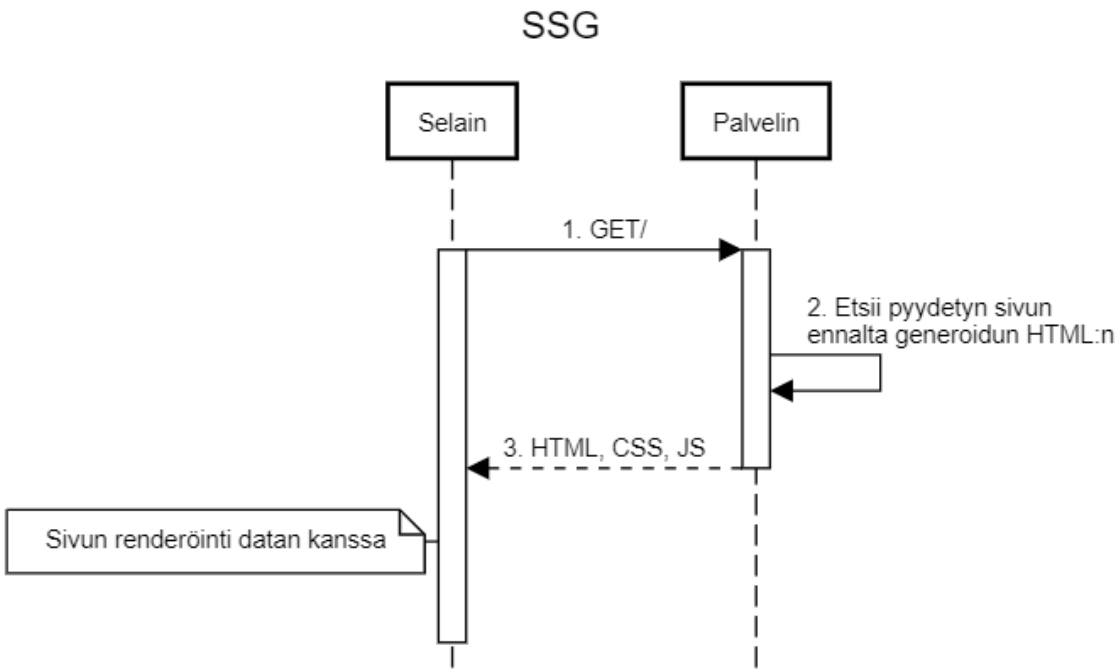
Sivuston kääntövaihe on kuvattu kuvan 3.3 sekvenssikaaviossa. Sivuston generointivaiheessa tehdään tarpeelliset API-kutsut palvelimille, jotka noutavat tiedon sivua varten. Kun tarpeelliset tiedot on hankittu, generoidaan lopulliset HTML-tiedostot, jotka siirretään palvelimelle.



Kuva 3.3. *SSG sekvenssikaavio osa 1*

Kuvan 3.4 sekvenssikaaviossa on kuvattu sivuston toiminta sen jälkeen kun generoidut tiedostot on siirretty palvelimelle. Selain tekee pyynnön palvelimelle, joka etsii ja palauttaa pyyntöä vastaavan datan selaimelle, joka renderoi sivun välittömästi näytölle. Käyttäjän navigoidessa sivulla toistuu kuvan 3.4 mukainen sekvenssi. Tämä sekvenssi vastaa tilannetta, missä sivun sisältö on täysin staattista. Käyttäjän navigoidessa sellaiselle sivulle, jossa on myös dynaamista dataa, vastaa kuvan 3.1 sekvenssiä kohdasta 5. eteenpäin.

Esimerkin mukainen toiminta vastaa staattista dataa, mutta tilanne missä generoidussa sovelluksessa haetaan dynaamisesti dataa, vastaa kuvan 3.1 mukaista toimintaa kohdasta 5. eteenpäin.



Kuva 3.4. *SSG sekvenssikaavio osa 2*

3.6 Gatsby

Gatsby on avoimen lähdekoodin React-pohjainen sovelluskehys. Gatsby on staattisen sivuston generaattori, eli sivuston kääntämisvaiheessa sivustosta generoituu staattisia HTML-tiedostoja, jotka ladataan palvelimelle. Yksi Gatsbyn pääideoista on se, että HTML-sisältö on staattisesti generoitu käyttäen React DOM palvelinpuolen API:a. Tämän jälkeen staattinen HTML-sisältö voidaan ehostaa selainpuolen JavaScriptillä, *React hydration* -metodia käyttäen [11]. Näin sivustosta saadaan interaktiivinen ja dynaaminen sovellus, huolimatta sivuston generoinnin staattisesta luonteesta. Generoiduille HTML-sivuille voidaan siis ladata normaalisti JavaScriptiä suoritettavaksi, ja näin saadaan tehtyä esimerkiksi dynaamisia API-kutsuja [38].

Gatsbyä ajetaan yleensä lokaalisti kehitysympäristössä ja kun haluttu sivusto on valmis, Gatsby generoi staattisen sivuston mikä koostuu HTML:stä, JavaScriptistä, CSS:stä ja kaikista tarvittavista liitetiedostoista. Gatsby on siis työkalu, jolla saadaan tehtyä itse lopputuote. Ajettaessa ”gatsby build” -komentorivikomento käynnistyy Node.js palvelin, joka prosessoi sivuston. GraphQL-skeema luodaan, staattisiin osuuksiin tarvittava data haetaan koodin mukaisilla kyselyillä ja lopuksi rakennetaan jokaisen sivun HTML-tiedostot [11].

Lopputuotteen generointiin Gatsby käyttää Node.js:ssää. Node.js:ssää ajetaan lokaalisti kehitysympäristössä kehittäjän tietokoneella. Kyseessä on sivuston staattisen sivuston generointi, joten Node.js:ssää ei tarvita enää lopputuotteen sijaitsemalla palvelimella.

Tiedonhakuun Gatsby käyttää GraphQL:ää. Kehitysympäristössä ajettavan Node.js:n ja

GraphQL:n yhdistelmän ansiosta saadaan staattisiin osuuksiin käytetty data haettua käytännössä mistä vaan. Tiedonlähteinä voi olla *Markdown*-tiedostoja, tietokantoja, sisällönhallintajärjestelmiä kuten WordPress, tai vaikka yksinkertaisia CSV-tiedostoja (Comma Separated Values). Data haetaan lähteistä GraphQL:n avulla Gatsbyyn, joka käyttää dataa generoidessaan lopputuotteen. Eri datalähteiden käyttöönottoon Gatsby käyttää liitännäisiä. Liitännäiset ovat Node.js -paketteja, jotka toteuttavat Gatsbyn API:t. Näiden liitännäisten avulla saadaan liitettyä ulkoiset datalähteet Gatsbyn GraphQL:ää varten ja muutettua data JSON-objekteiksi tiedostoformaateista kuten *markdown* ja *CSV*.

Myös Gatsby tarjoaa sivujen ennalta hakua. Gatsby käyttää sovelluksen sisäiseen navigointiin *<Link>* -komponenttia. Selain lataa <Link> -komponentin kohteena olevan sivun JavaScript-tiedoston. Tämä lataus tehdään matalalla prioriteetilla, eli vasta kun selain on valmis sen hetkisen sivun toimintojen kanssa. [6] Jos käyttäjä vie kursorin <Link> -komponentin päälle, ladataan kohteen JavaScript-tiedosto resurssien kanssa korkealla prioriteetilla. [7] [5]

Reactin ja CSS:n avulla toteutetaan sivuston ulkoasu ja toiminta. Reactilla toteutetaan siis sivuston mallit, joihin ladataan GraphQL:n avulla haettu data ja lopulta Gatsbyn avulla generoidaan lopputuotteena oleva sivusto.

Gatsby noudattaa PRPL-mallia (Push, Render, Pre-cache, Lazy load). Malli on Googlen kehittämä arkkitehtuurimalli verkkosivujen ja sovellusten rakentamista varten, joiden tulee toimia mahdollisimman hyvin puhelimilla ja muilla laitteilla, joilla on hitaat verkkoyhteydet. Gatsby renderoi sivun staattisen HTML:n käyttäjälle, jonka jälkeen lataa sen sivun JavaScript-paketin. Välittömästi paketin latauksen jälkeen ladataan välimuistiin valmiiksi kyseiseen sivuun liittyvien muiden sivujen resursseja, minkä seurauksena käyttäjän navigoidessa tältä sivulta toiselle sivulle, Gatsby luo uuden sivun selaimessa.

3.7 Valittujen toteutustapojen vertailu

Tarkasteltaessa aiemmin tässä luvussa esiteltyjen toteutustapojen kokonaisuuksia, voidaan todeta, että asiakaspään renderoinnin, palvelinpään renderoinnin ja staattisen sivuston generaattorin avulla toteutettujen sovellusten merkittävin ero on siinä, missä vaiheessa HTML koostetaan. Asiakaspään renderoinnissa HTML koostetaan vasta selaimessa sen jälkeen, kun JavaScript on saatu ladattua ja parsittua. Palvelinpään renderoinnissa HTML koostetaan palvelimella ja lähetetään vastauksena selaimen pyyntöön. Staattisen sivuston generaattori koostaa HTML:n generointivaiheessa ja palvelin lähettää HTML:n vastauksena selaimen pyyntöön.

Asiakaspään renderointia käyttävän React App:n heikkoutena voidaan pitää hidasta ensilatausta, sillä kuten luvussa 3.1 todettiin, sen tarvitsee ladata ja parsia JavaScript ennen kuin se voi piirtää mitään sisältöä käyttäjälle. React App:n vahvuutena pidetäänkin toimintaa ensilatauksen jälkeen. Ensilatauksen jälkeen sovelluksen reititys ja logiikka on selaimen vastuulla, jolloin kaikki muutokset voidaan tehdä nopeasti. Palvelinpään rende-

rointia käyttävä Next.js tarvitsee palvelimen, joka voi suorittaa palvelinpään koodia. Palvelimen avulla Next.js:n ei tarvitse lähettää ja parsia JavaScriptiä saadakseen sisältö käyttäjän näkyviin. Next.js:n vahvuutena voidaan siis tämän perusteella pitää ensilatauksen nopeutta. Staattisen sivuston generaattori Gatsbyn vahvuutena voidaan pitää staattisen sisällön piirron nopeutta ensilatauksessa, sillä kaikki staattinen sisältö mikä on asetettu generointivaiheessa paikoilleen, piirretään heti HTML:n saapuessa selaimeen. Gatsbyn heikkoutena voidaan pitää dynaamisen sisällön piirtoa, sillä kuten React App:ssa täytyy ensin JavaScript ladata ja parsia ennen kuin tarvittava dynaaminen sisältö voidaan hakea tietokannasta ja piirtää käyttäjälle näkyviin.

Toteutustapojen välisiä konkreettisia suorituskykyeroja on kuitenkin vaikea arvioida pelkän teorian pohjalta. Myös kahta eri sovellusta ja kahden eri sovelluksen vertailu ei anna suhteellista kuvaa toteutustapojen suorituskyvystä. Tästä syystä tässä työssä toteutetaan prototyyppisovellus valituilla toteutustavoilla ja näille suoritetaan suorituskykymittauksia. Tällä tavalla saadaan tarkempi vertailu konkreettisista suorituskykyeroista.

4 PROTOTYYPPISOVELLUS JA SEN TOTEUTUS

Toteutettu prototyyppisovellus on idealtaan verkkokauppatyylinen sivusto, jossa käyttäjä voi selata ostettavia tuotteita, lukea niistä lisätietoja ja kirjoittaa kommentteja tuotteista. Tuotteita voi lisätä ostoskoriin, jossa tilattavien tuotteiden määrää voi myös lisätä tai vähentää. Ostoskori on käyttäjäkohtainen, jonka muutokset tallentuvat tietokantaan. Sivusto koostuu etusivusta, tuotteet-sivusta, ostoskorisivusta, tietoa meistä -sivusta sekä jokaisen tuotteen omasta tuotetietosivusta.

Prototyyppisovellusta varten rakennettu GraphQL-palvelin on Herokussa käyttöönotettu Hasura-moottori, joka käyttää PostgreSQL-tietokantaa. Kaikki sivuston käyttämä data sijaitsee tässä tietokannassa. Tietokantarakenne luotiin hasuran graafisen käyttöliittymän avulla, josta hasura generoi GraphQL-skeeman.

Prototyyppisovellus toteutettiin ensimmäisenä Next.js:llä loppuun, jonka jälkeen Gatsbyllä ja viimeiseksi React App:lla. Koska kaikki ovat React-pohjaisia, pystyi suurimman osan komponenteista kopioimaan toteutuksesta toiseen muuttaen vain pieniä osuuksia.

4.1 Konfiguraatiot

Jokainen toteutus vaatii konfiguraatiota, jotta sivujen reititys ja tiedonhaku saadaan toimimaan toteutustavalle ominaisella tavalla. Näiden konfiguraatioiden konkreettisia toteutuksia on esitelty tämän luvun aliluvuissa.

4.1.1 Prototyyppisovelluksen sivujen reititys

Next.js:ssä ja Gatsbyssä sivuston reititys määräytyy oletuksena tiedostonimien ja kansiorakenteen mukaan. Perinteisessä React App:ssa reititys tapahtuu selaimessa, joten se täytyy määritellä JavaScriptin puolella.

React App:ssa reititykseen on käytetty kirjastoa *react-router-dom*, jonka toiminta on kuvattu listauksessa 4.1. Switch-komponentti valitsee ensimmäisen Route-alikomponentin, jonka path-arvo vastaa sen hetkistä URL:ää (Uniform Resource Locator) ja renderoitavaksi jää *component*-arvossa viitattava komponentti [30]. Route-komponentissa oleva *exact*-avainsana asettaa renderoinnille ehdon, että URL:in täytyy vastata täsmälleen

path-arvoa, jotta komponentti renderoidaan. Ilman exact-avainsanaa Switch-komponentti valitsee rivillä 8 olevan polun, URL:in ollessa esimerkiksi */about*.

```
1  ...
2  function App() {
3    return (
4      <ApolloProvider client={client}>
5        <Router>
6          <Layout>
7            <Switch>
8              <Route exact path="/" component={ MainPage } />
9              <Route path="/about" component={ AboutPage } />
10             <Route path="/cart" component={ CartPage } />
11             <Route exact path="/products" component= { ProductsPage } />
12             <Route path="/products/:id" component={ ProductPage } />
13           </Switch>
14         </Layout>
15       </Router>
16     </ApolloProvider>
17   );
18 }
```

Listaus 4.1. *Reititys Reactissa*

Next.js:ssä sivuston reititys määräytyy oletuksena *pages*-kansion mukaan. Sen sisältämistä kansioista ja tiedostoista muodostuu reititettävät polut. Esimerkiksi *pages/about.js* tiedostosta Next.js muodostaa reitityksen URL:in loppuosaksi "/about".

Gatsbyssä *pages*-kansiossa olevista komponenteista generoidaan automaattisesti staattiset tiedostot, joiden reitityspolku muodostuu tiedostonimistä ja kansionimistä samalla tavalla kuin Next.js:ssä. Samaa mallia käyttävät sivut joista generoidaan staattiset tiedostot vaativat kappaleessa 4.1.5 kuvatun konfiguraation, jossa määritellään myös reititys. Prototyyppisovelluksen tuotesivut reitittyvät tämän konfiguraation mukaisesti reittiin "/products/id", jossa id on tuotteen id.

4.1.2 GraphQL:n käyttöönotto Next.js:ssä

Next.js:ssä GraphQL saadaan käyttöön luomalla sitä varten HOC (Higher order component), mikä tarkoittaa käytännössä funktiota, joka ottaa vastaan komponentin ja palauttaa uuden komponentin [31]. Listauksessa 4.2 luodaan *ApolloClient*, jonka konfiguraatiossa annetaan GraphQL-palvelimen osoite ja pääsyavain ja välimuistin konfiguraatiolle annetaan Apollon suosittelema välimuistin implementaatio *InMemoryCache* [1].

```
1  import withApollo from 'next-with-apollo';
2  import ApolloClient, { InMemoryCache } from 'apollo-boost';
3
4  export default withApollo(
5    ({ ctx, headers, initialState }) =>
6      new ApolloClient({
7        uri: 'https://dippa-backend.herokuapp.com/v1/graphql',
8        headers: {
9          'x-hasura-access-key': process.env.HASURA_ACCESS_KEY,
10       },
11       cache: new InMemoryCache().restore(initialState || {})
12     })
13 );
```

Listaus 4.2. *GraphQL:n käyttöönotto Next.js:ssä*

Next.js käyttää *App*-komponenttia alustaessaan sivuja, jonka oletustoteutuksen kehittäjä voi ylikirjoittaa tiedostoon pages/_app.js. Tässä prototyypissä kyseiselle tiedostolle on tehty listauksen 4.3 mukainen toteutus, jossa jokainen sivu ympäröidään kustomoidulla *<LayOut>* -komponentilla asetteluja varten ja tämän lisäksi *<ApolloProvider>* -komponentilla millä saadaan komponenteille käyttöön mm. useQuery ja useMutation funktiot GraphQL kyselyitä varten. Rivillä 20 kutsutaan luotua HOC:ia *withApollo*, jolla saadaan luotua yhteys GraphQL palvelimeen. Tällä konfiguraatiolla GraphQL on käytettävissä kaikilla sovelluksen komponenteilla.

```
import App from 'next/app';
import { ApolloProvider } from '@apollo/react-hooks';
import withApollo from '../lib/withApollo';
import Layout from '../components/Layout';

class MyApp extends App {
  render() {
    const { Component, pageProps, apollo } = this.props;

    return (
      <ApolloProvider client={apollo}>
        <Layout>
          <Component {...pageProps} />
        </Layout>
      </ApolloProvider>
    );
  }
}

export default withApollo(MyApp);
```

Listaus 4.3. *GraphQL:n välitys komponenteille*

4.1.3 GraphQL:n käyttöönotto React App:ssa

React App:ssa GraphQL-yhteys saadaan käyttämällä *apollo-boost* -kirjaston tarjoamaa ApolloClient:iä, jolle annetaan listauksen 4.4 mukaisesti GraphQL-palvelimen URI ja pääsyavain. Luotu ApolloClient annetaan parametrinä listauksessa 4.1 näkyvälle ApolloProviderille, jonka jälkeen GraphQL-palvelin on käytössä sivuston komponenteilla.

```
import ApolloClient from 'apollo-boost';
import { ApolloProvider } from '@apollo/react-hooks';

const client = new ApolloClient({
  uri: 'https://dippa-backend.herokuapp.com/v1/graphql',
  headers: {
  'x-hasura-access-key': process.env.REACT_APP_HASURA_ACCESS_KEY,
  }
});
...
```

Listaus 4.4. *GraphQL:n käyttöönotto React App:ssa*

4.1.4 GraphQL:n käyttöönotto Gatsbyssä

Gatsbyssä tiedonhakua suoritetaan sekä sivuston generointivaiheessa, että suoritusvaiheessa. Tästä syystä GraphQL-yhteys muodostetaan siis kahdessa eri vaiheessa, joita kumpaakin varten täytyy konfiguroida GraphQL-yhteys. Generointivaiheen yhteys GraphQL:ään saadaan käyttämällä GraphQL-liitännäistä *gatsby-source-graphql*. [9]

```
...
plugins: [
    {
      resolve: 'gatsby-source-graphql',
      options: {
        typeName: 'hasura',
        fieldName: 'dippaBackend',
        createLink: () => {
          return createHttpLink({
            uri: 'https://dippa-backend.herokuapp.com/v1/graphql',
            headers: {
              'x-hasura-access-key': process.env.HASURA_ACCESS_KEY,
            },
            fetch,
          });
        }
      }
    },
    ...
  ]
}
```

Listaus 4.5. *Gatsby GraphQL:n käyttöönotto generointivaiheeseen liitännäisellä*

Listauksessa 4.5 on esitettynä osa gatsby-config.js:ssää, jossa määritellään yhteys GraphQL-palvelimeen liitännäisen avulla. Konfiguraatioiksi annetaan *typeName*, fieldName ja *createLink*. Näistä ensimmäinen on mielivaltainen nimi kyselyn skeemalle. Toinen on nimi, jota käytetään GraphQL-kyselyissä, kun otetaan yhteyttä kyseiseen GraphQL-palvelimeen. Kolmannessa muodostetaan URL-osoite palvelimeen pääsyavaimen kanssa.

```
// client.js
import ApolloClient from 'apollo-boost';
import fetch from 'isomorphic-fetch';

export const client = new ApolloClient({
  uri: 'https://dippa-backend.herokuapp.com/v1/graphql',
  headers: {
    'x-hasura-access-key': process.env.HASURA_ACCESS_KEY,
  },
  fetch
});

// wrap-root-element.js
import React from 'react';
import { ApolloProvider } from '@apollo/react-hooks';
import { client } from './client';


export const wrapRootElement = ({ element }) => (
  <ApolloProvider client={client}>{element}</ApolloProvider>
);

// gatsby-browser.js
export { wrapRootElement } from './src/apollo/wrap-root-element';
```

Listaus 4.6. *Gatsby GraphQL käyttöönotto selaimen puolella*

Listauksessa 4.6 on listattuna *client.js*, *wrap-root-element.js* ja *gatsby-browser.js* tiedostojen sisältö. Ensimmäisessä rakennetaan ApolloClient, toisessa asetetaan client ApolloProviderille ja luodaan siitä HOC *wrapRootElement*, jonka exporttaamalla tiedostossa gatsby-browser.js, Gatsby osaa antaa sen käyttöön kaikille komponenteille selaimen puolella [10].

4.1.5 Tuotesivujen generointi Gatsbyssä

Sovelluksessa jokainen tuote on esitelty omalla sivullaan. Tuotesivun malli on jokaiselle tuotteelle sama, mutta tiedot tuotteille haetaan tietokannasta. Jotta jokaisesta tuotteesta saadaan luotua Gatsbyssä oma staattinen sivu, täytyy tuotesivujen generointi konfiguroida tiedostossa *gatsby-node.js*. Listauksessa 4.7 on tarvittava konfiguraatio tuotesivun generointia varten. Rivillä 3 tehdään kysely tietokantaan hakien jokaisen tuotteen id. Riviltä 12 alkaen jokaista haettua id:tä kohti luodaan oma sivu, missä path-arvo vastaa sen sivun URL-osoitetta eli reittiä, *component*-arvoon asetetaan polku sivun malliin, jon-

ka mukaan sivu luodaan ja *context*-arvoon asetetaan haluttuja parametrejä sivun luontia varten. Tässä tapauksessa annetaan tuotteen id, jota käytetään tuotteen tietojen hakua varten mallissa. [8]

```
const path = require(`path`)
exports.createPages = async ({ actions, graphql }) => {
  const { data } = await graphql(`
  query {
    dippaBackend {
      product {
        id
      }
    }
  }
`);
  data.dippaBackend.product.forEach(({ id }) => {
    actions.createPage({
      path: `products/${id}`,
      component: path.resolve(`./src/templates/product.js`),
      context: {
        id: id,
      },
    })
  })
}
```

Listaus 4.7. *Tuotesivujen generointi Gatsbyssä*

Tällä konfiguraatiolla saadaan sivujen käännösvaiheessa luotua html-sivu jokaisesta tietokannassa olevasta tuotteesta.

4.2 Tiedonhaku

Tiedonhaku suoritetaan jokaisessa toteutuksessa hieman toisistaan poiketen. Gatsbyssä suoritetaan tiedonhakua sekä kääntövaiheessa, että suoritusvaiheessa, Next.js:ssä tiedonhaku suoritetaan sekä palvelimella, että selaimella ja React App:ssa pelkästään selaimella.

4.2.1 Tiedonhaku Next.js ja React App toteutuksissa

Next.js toteutuksessa tiedonhaku jaetaan kahteen eri paikkaan: palvelimelle ja selaimelle. Listauksessa 4.8 on kuvattu *ProductPage.js* -komponentin tiedonhaku, mikä suoritetaan

kun käyttäjä navigoi yksittäisen tuotteen sivulle. Jos kyseessä on ensimmäinen sivun lataus, haetaan kaikki sivun sisältämät tiedot palvelinpäässä ja muodostetaan niiden avulla HTML, joka lähetetään selaimeen. Jos sivulle navigoidaan Next.js:n tarjoaman <Link>-komponentin kautta tai dynaamisen reitityksen kautta, suoritetaan tiedonhaku vain selaimessa [26]. Tässä tapauksessa suoritetaan getInitialProps-funktiossa oleva tiedonhaku ennen sivun ensimmäistä piirtoa näytölle ja tämän jälkeen loput mahdolliset sivun sisältämät tiedonhaut. Näin käyttäjälle saadaan oleellinen tieto mahdollisimman nopeasti.

```
...
const ProductPage = (props) => {
  const product = props.product[0];
  return (
    ...
      <Comments product={product} />
    ...
    );
}
...

ProductPage.getInitialProps = ctx => {
  const productID = ctx.query.id;
  const apolloClient = ctx.apolloClient;

  return apolloClient.query({ query: GET_PRODUCT_DETAILS, variables: { ... } })
    .then(res => {
      return { product: res.data.product };
    });
}
export const GET_PRODUCT_DETAILS = gql`
  query($id: Int!, $userID: Int!) {
    ...
`
```

Listaus 4.8. *Next.js tiedonhaku GraphQL-kyselyllä palvelimella*

Listauksessa 4.9 tehty kysely (GET_COMMENTS) suoritetaan vasta selaimessa käyttäen *@apollo/react-hooks* -kirjaston tarjoamaa useQuery-funktiota.

```
1  import { useQuery, useMutation } from '@apollo/react-hooks';
2  ...
3  const Comments = (props) => {
4    const { loading, error, data } = useQuery(GET_COMMENTS, {variables: ...});
5
6    if (loading) return 'Loading...';
7    if (error) return `Error! ${error.message}`;
8
9    return (
10     <div className="comments">
11       <h4>Comments</h4>
12       { data.comment.map(comment => <Comment ... />) }
13     </div>
14   );
15 }
16 ...
17 const GET_COMMENTS = gql`
18   query($productID: Int) {
19     ...
20 `;
```

Listaus 4.9. *Next.js tiedonhaku GraphQL-kyselyllä selaimessa*

React App suorittaa kaikki GraphQL-kyselyt selaimessa, eikä siitä syystä eroa listauksen 4.9 toteutuksesta muulla tavalla, kuin sillä että tuotteen tiedot haetaan myös käyttäen useQuery-funktiota samalla tavalla kuin esimerkiksi kommenttien haku rivillä 4.

4.2.2 Tiedonhaku Gatsby toteutuksessa

Gatsbyssä konkreettinen toteutus kyselyille ei juurikaan eroa kappaleessa 4.2.1 kerrotusta Next.js:n tavasta. Gatsbyssä käännönaikainen GraphQL -kysely määritetään käyttämällä Gatsbyn tarjoamaa *graphql*-funktiota, kuten listauksessa 4.10 rivillä 19. Tämä funktio täytyy vielä asettaa näkyväksi tiedoston ulkopuolelle, asettamalla sen eteen avainsana *export*. Sivujen kääntövaiheessa ajetaan rivillä oleva kysely ja tulos asetetaan komponentin parametreihin data-arvoon. Rivillä 6 tallennetaan tuotteen tiedot muuttujaan. Gatsbyn toteutuksessa suorituksenaikaiset kyselyt tehdään samalla tavalla kuin Next.js:ssä, eli käyttäen @apollo/react-hooks-kirjaston useQuery-funktiota.

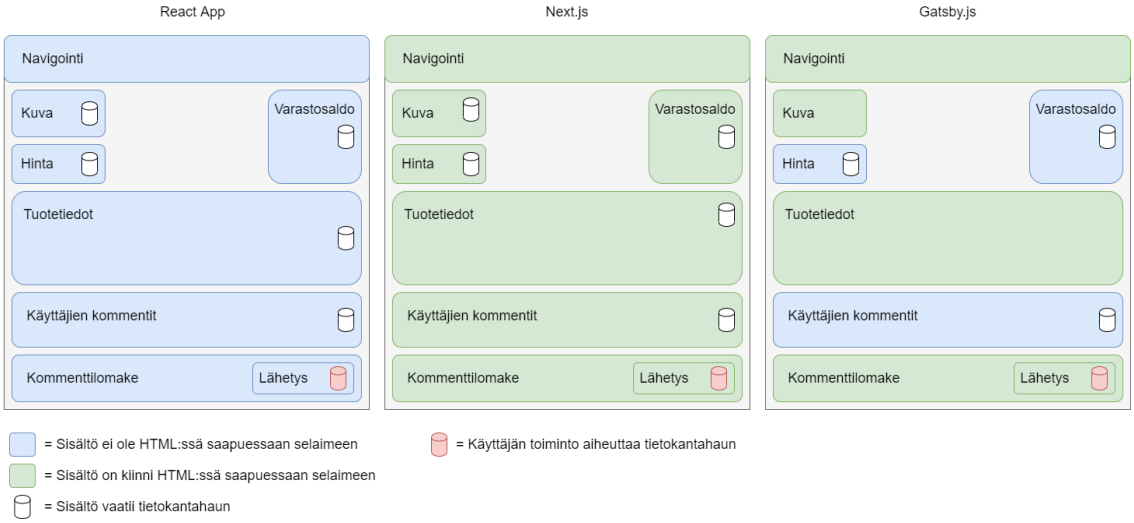
```
import gql from 'graphql-tag';
import { graphql } from 'gatsby';
import { useQuery, useMutation } from '@apollo/react-hooks';
...
const ProductPage = (props) => {
  const product = props.data.dippaBackend.product[0];
    ...
    const Comments = (props) => {
      const { loading, error, data } = useQuery(GET_COMMENTS, {variables: ...});
      ...
    }
  ...
  return (
    ...
      <Comments product={product} />
    ...
    );
}
export const GET_PRODUCT_DETAILS = graphql`
  query($id: Int!) {
    dippaBackend {
      product (where: {id: { _eq: $id }}) {
        ...
      }
    }
  }
`
const GET_COMMENTS = gql`
  query($productID: Int) {
    ...
`;
...
```

Listaus 4.10. *Gatsby.js tiedonhaku GraphQL-kyselyllä*

4.3 Sisällönkoonti toteutustavoissa

Kuva 4.1 havainnollistaa eri toteutustapojen sisällön koontia selaimessa ja palvelimella, kun sivulle navigoidaan ensimmäistä kertaa. Kuvassa on esimerkkinä hahmoteltuna tuotesivun eri komponenttien asettelu. Sininen väri tarkoittaa, että sen komponentin sisältö ei ole palvelimen palauttamassa HTML-tiedostossa, vaan tieto lisätään HTML:ään selaimessa. Vihreä väri tarkoittaa sitä, että komponentin tieto on jo HTML:ssä, kun HTML

saapuu selaimelle. Vihreällä värillä olevat komponentit voidaan siis piirtää käyttäjälle välittömästi, kun HTML-tiedosto saapuu selaimeen. Valkoinen lieriömerkki tarkoittaa, että kyseisen komponentin sisältö on noudettava tietokannasta suorituksen aikana. Punainen lieriömerkki tarkoittaa, että käyttäjän toiminto aiheuttaa tietokantatoiminnon. Tällaisia toimintoja ovat esimerkiksi kommentin lähettäminen ja tuotteen lisääminen ostoskoriin.



Kuva 4.1. *React App:n, Gatsbyn ja Next.js:ssän sisällönkoonti*

Kuvan 4.1 mukaisesti React App koostaa kaiken sisällön selaimessa ja suorittaa tietokantahaun selaimesta, jotta tarvittavat tiedot voidaan piirtää käyttäjälle. Komponentit, jotka vaativat tässä esimerkissä tietokantahakua, ovat kuva, hinta, tuotetiedot, varastosaldo ja käyttäjien kommentit. Näistä tuotteen kuva voi olla sovelluksen kanssa samalla palvelimella, mutta tarvitsee kuitenkin tietokannasta tiedot siihen, mikä kuva valitaan näytettäväksi.

Next.js:n ratkaisussa koko sivun sisältö on jo HTML:ssä, jonka palvelin lähettää selaimelle. Tietokantahaut niitä tarvitseville komponenteille suoritetaan palvelimella ennen HTML:n koontia ja lähetystä selaimelle. Suorituskyvyn ja nopeuden kannalta suurin ero React App:iin on se, että selaimen ei tarvitse ensin ladata JavaScript-tiedostoja, jotka selaimen täytyisi parsia ja suorittaa ennen kuin mitään sisältöä saadaan käyttäjälle näkyviin. Next.js:ssä pyynnön jälkeen suoritetaan tarvittavat tiedonhaut, asetetaan tiedot HTML:ään ja lähetetään se selaimelle.

Gatsbyn ratkaisussa osa sisällöstä on palvelimen lähettämässä HTML:ssä ja osa sisällöstä saadaan tekemällä tietokantahaku selaimesta. Erona muihin ratkaisuihin on kuitenkin se, että osa tiedoista, jotka muissa ratkaisuissa vaativat tietokantahaun, eivät tässä ratkaisussa vaadi tietokantahakua enää suoritusvaiheessa. Nämä tiedot asetetaan paikalleen sivuston generointivaiheessa. Kun käyttäjä navigoi yksittäisen tuotteen sivulle, valitsee palvelin sitä vastaavan HTML-tiedoston, jossa tiedot ovat paikallaan ja lähettää sen selaimeen. Mukana tullut JavaScript-tiedosto parsitaan ja suoritetaan kuten React App -ratkaisussa ja tämän jälkeen voidaan suorittaa tietokantahaut niitä tarvitseville komponenteille (siniset komponentit, joissa lieriö). Jos komponentin tieto päivittyy harvoin,

voidaan se asettaa paikoilleen generointi vaiheessa, jolloin sivu latautuu nopeampaa.

5 SUORITUSKYVYN MITTAUS

Web-sovellusten suorituskyvyn arvioinnissa täytyy määritellä tarkasti mitä avainarvoja tarkastellaan, jotta tuloksesta on hyötyä. Tarkkailemalla oikeita mittareita saadaan hyvä kuva siitä, miten loppukäyttäjä kokee sovelluksen suorituskyvyn. Käyttäjä voi kokea yhden sivun latauksen nopeammaksi kuin toisen, vaikka molempien sivujen lataus olisi valmis yhtä nopeasti. Tähän vaikuttaa se, mitä latauksen aikana tapahtuu. Mitä nopeammin latauksen aloittamisesta piirretään näytölle jotain sisältöä, sitä nopeammaksi käyttäjä kokee latauksen. Sisällön lataus progressiivisesti voi tuoda käyttäjälle näkyviin merkittävää sisältöä jo latauksen aikana, jolloin käyttäjän ei tarvitse odottaa koko latauksen valmistumista. Kokonaissuorituskykyyn vaikuttaa myös se, kuinka nopeasti lataamisen aloittamisesta sivu on valmis ottamaan vastaan käyttäjän syötteitä ja kuinka sulavia ja viiveettömiä toiminnot ovat. [35]

Tässä työssä suorituskyvyn mittaamiseen käytetään Google Chrome -selaimen tarjoamia kehitystyökaluja. Yksi näistä työkaluista on *Lighthouse*-auditointityökalu, jonka avulla voidaan mitata suorituskyvyn kannalta olennaisia avainarvoja [14]. Lighthouse-työkalun suorituskyvyn auditointi sisältää mittarit avainarvojen mittaamiseksi ja näistä mittareista työssä tarkastellaan seuraavia:

- *First Contentful Paint* (FCP): mittaa kauanko kestää sivun latauksen aloituksesta siihen, että jokin osa sivun sisällöstä piirretään näytölle.
- *Largest Contentful Paint* (LCP): mittaa kauanko kestää sivun latauksen aloituksesta siihen, että sivun sisällön isoin elementti piirretään näkyviin.
- *Time to Interactive* (TTI): mittaa kauanko kestää sivun latauksen aloituksesta siihen, että sivu on täysin interaktiivinen.
- *Max Potential First Input Delay*: Mittaa kauanko kestää pisimmässä pääsäikeessä suoritettavassa tehtävässä FCP:n jälkeen.

Mittarien mittaamat arvot auttavat arvioimaan, kuinka käyttäjät kokevat sivun suorituskyvyn. On tärkeää mitata ensimmäinen sisällönpiirto, sillä se on ensimmäinen palaute käyttäjälle siitä, että sivun lataus on käynnistynyt[13]. Suurimman elementin piirron mittaaminen auttaa arvioimaan käyttäjän kokemaa latausnopeutta, sillä suurimmassa elementissä on yleensä sivun pääsisältö [34]. Voidaan siis olettaa, että suurimman elementin piirron jälkeen käyttäjällä on jo jotain käytännöllistä näkyvillä. Täyteen interaktiivisuuteen kuluvan ajan ja suurimman potentiaalisen viiveen mittaaminen on tärkeä, sillä ne vaikuttavat latausajan käyttökokemukseen. Käyttäjä voi turhautua, jos sivu ei vastaa syötteisiin,

vaikka sisältö on näkyvillä.

Lighthouse-työkalun lisäksi Chromen kehitystyökaluista käytetään ajonaikaisen suorituskyvyn analysointiin suorituskykytyökalua. Työkalu sijoittaa kaikki selaimen suorittamat tapahtumat aikajanalle. Näin nähdään esimerkiksi, milloin selain on piirtänyt sivun näytölle ja kauanko sen suorittamiseen meni aikaa. Myös Lighthouse-mittaukset voidaan asettaa tällä aikajanalle tarkasteltavaksi.

Lighthouse-työkalun auditointi pisteyttää mittaustulokset ja antaa konkreettisia parannusehdotuksia auditoinnin kohteena olevalle sivulle. Koska tämän työn mittausten tarkoituksena ei ole parantaa kyseisen sivun suorituskykyä, vaan vertailla toteutustapojen välisiä eroja suorituskyvyssä, voidaan pisteytys ja parannusehdotukset jättää huomiotta.

Mittaukset suoritettiin prototyyppisovelluksen tuotesivulle. Tästä tuotesivusta tehtiin kaksi erilaista Gatsby-versiota mittauksia varten. Ensimmäisessä versiossa tuotetiedot ja tuotteen kuva luokitellaan harvoin päivittyväksi tiedoksi ja hinta, varastosaldot sekä kommentit luokitellaan usein päivittyväksi tiedoksi. Usein päivittyvä tieto haetaan Gatsby toteutuksessa vasta selaimessa, kun harvoin päivittyvä tieto voidaan hakea jo generointivaiheessa. Jatkossa tähän versioon viitataan Gatsby-v1:llä. Toisessa versiossa kaikki haettavat tiedot luokitellaan usein päivittyväksi tiedoksi. Jatkossa tähän versioon viitataan Gatsby-v2:lla. Tekemällä tällainen toinen versio saadaan mitattua Gatsbyn suorituskykyä myös tilanteissa, missä kaikki tiedot haetaan vasta selaimessa.

Konkreettinen muutos on näkyvissä listauksessa 5.1. Nyt tuotetietoja ei haeta käyttäen graphql-funktiota, joka suorittaa kyselyn staattisten tiedostojen generointivaiheessa. Sen sijaan tiedot haetaan selaimessa käyttäen *@apollo/react-hooks*-kirjaston tarjoamaa useQuery-funktiota kuten muissakin selaimessa suoritettavissa kyseilyissä.

```
import { useQuery } from '@apollo/react-hooks';
...
const ProductPage = (props) => {
  ...
  const {loading, error, data} = useQuery(GET_PRODUCT_DETAILS,
  { variables: ... });
  ...
  var product = data.product[0];
  ...
}
export default ProductPage;
...
const GET_PRODUCT_DETAILS = gql`
  query($id: Int!) {
    product (where: {id: { _eq: $id }}) {
      ...
    }
  }
`;
...
`
```

Listaus 5.1. *Gatsby tuotesivun muutos*

Mittauksissa mitattiin jokaisen toteutustavan ensilatauksen suorituskykyä, sekä sovelluksen sisäisen navigoinnin suorituskykyä. Tuotesivun ensilatauksen mittaukset suoritettiin käyttäen Lighthouse-työkalua. Näitä koskevat mittausasetukset ja ympäristötiedot ovat listattuna liitteessä G. Koska Google Chrome -selaimen sisäänrakennetulla Lighthouse-työkalulla ei voi mitata suorituskykyä navigoidessa sovelluksen sisällä, suoritettiin sovelluksen sisäisen navigoinnin mittaukset Google Chrome -selaimen suorituskykytyökalulla.

Lighthouse-työkalu kuristaa auditoinnissa tehtäviin mittauksiin verkkoyhteyttä sekä prosessoritehoa simuloidakseen mobiilikäyttäjää, jolla on hidas verkkoyhteys. Google Chrome -selaimen suorituskykytyökalussa asetetaan myös mittauksia varten kuristus verkkoyhteydelle ja prosessoriteholle. Prosessoritehon kuristus on sama kuin Lighthouse-työkalussa. Verkkoyhteyden kuristuksessa käytetään vaihtoehtoa ”Fast 3G” joka on lähellä Lighthouse-mittauksissa käytettävää kuristusta. Google ei kuitenkaan kerro tarkkaa verkkoyhteyden nopeutta tälle vaihtoehdolle, joten mittaustulosten aika-arvot eivät ole verrattavissa Lighthouse-työkalulla tehtyihin mittausten tuloksiin.

Jotta mittauksessa saadaan minimoitua ulkopuolisista syistä aiheutuva heitto yksittäiseen mittaukseen, suoritettiin mittaus kullekin toteutustavalle kymmenen kertaa ja otettiin näistä mittauksista keskiarvo. Kaikkien yksittäisten mittausten tulokset ovat listattuna liitteissä A, B, C, D, E ja F.

5.1 Tuotesivun ensilataus (Gatsby-v1)

Mittaus suoritettiin prototyyppisovelluksen tuotesivulle, joka on rakenteeltaan kuvassa 4.1 esitetyn sivun kaltainen. Mitattavina oli React App -toteutus, Next.js-toteutus sekä Gatsby-v1-toteutus. Mittaukset suoritetaan tilanteelle, missä käyttäjä navigoi sivustolle ensimmäistä kertaa avaten kyseisen tuotesivun.

Taulukossa 5.1 on listattu React-App toteutuksen, Next.js-toteutuksen sekä Gatsby-v1-toteutuksen mittaustulosten keskiarvot. Taulukkoon on merkattu jokaisen avainarvon paras tulos vihreällä värillä ja huonoin tulos punaisella värillä.

Taulukko 5.1. *Tuotesivun ensilatauksen mittaustulokset (Gatsby-v1)*

	Gatsby-v1	**Next.js**	**React App**
First Contentful Paint	1.4s	1.91s	2.3s
Largest Contentful Paint	1.4s	1.91s	2.94s
Time to Interactive	2.83s	2.43s	2.68s
Max Potential first input delay	134ms	213ms	51ms

Tarkasteltaessa taulukon 5.1 tuloksia, nähdään että Gatsby-v1 ensimmäisessä sisällön piirrossa kestää 26.7% vähemmän aikaa kuin Next.js:n ja 39,0% vähemmän kuin React App:n. Gatsbyn ja Next.js:n ensimmäinen sisällön piirto sisältää jo käyttäjälle merkitsevää tietoa. Merkitsevä tieto on tässä tapauksessa siis tuotetiedot. React App:n ensimmäinen sisällön piirto ei sisällä mitään tietokannasta haettavaa tietoa. React App:n tuotetietojen piirtämiseen kuluu 110% enemmän aikaa kuin Gatsby-v1:llä ja 53,9% enemmän kuin Next.js:llä.

Next.js oli latauksen alkamisen jälkeen nopeiten täysin interaktiivinen. Aikaa kului 14.1% vähemmän kuin Gatsby-v1:llä ja 9,3% vähemmän kuin React App:lla.

Ensimmäisen piirron jälkeen pienimmän potentiaalisen viiveen käyttäjän syötteeseen aiheuttaa React App, joka on 76% pienempi kuin Next.js:n ja 69% pienempi kuin Gatsby-v1.

Toteutusten aika täyteen interaktiivisuuteen ei eroa toisistaan merkittävästi, mutta huomioitavaa on kuitenkin ensipiirron ja täyden interaktiivisuuden välinen aika, jonka aikana käyttäjien syötteeseen voi aiheutua viivettä.

Käyttäjän syötteeseen potentiaalisesti aiheutuva maksimiviive on isoin Next.js-toteutuksessa, jossa se on 213ms. Gatsby-v1 toteutuksessa potentiaalinen maksimiviive on 134ms. Aikaikkuna missä maksimiviive voi aiheutua, on Next.js:llä 0,52s (2,43s-1,91s), Gatsby-v1:llä 1,43s. React App suoriutuu tästä parhaiten, sillä aikaikkuna on vain 0,38s ja potentiaalinen maksimaalinen viive 51ms.

5.2 Tuotesivun ensilataus (Gatsby-v2)

Toista mittausta varten Gatsby-v1 toteutusta muutettiin siten, että myös kuvan tiedostopolku ja tuotetiedot haetaan vasta selaimessa. Näin saadaan Gatsbyn toteutukselle vertailuarvot myös tilanteessa, missä kaikki käyttäjälle merkittävä sisältö luokitellaan usein päivittyväksi tiedoksi.

Taulukossa 5.2 on listattu React-App toteutuksen, Next.js-toteutuksen sekä Gatsby-v2-toteutuksen mittaustulosten keskiarvot. Taulukkoon on merkattu jokaisen avainarvon paras tulos vihreällä värillä ja huonoin tulos punaisella värillä.

Taulukko 5.2. *Tuotesivun ensilatauksen mittaustulokset (Gatsby-v2)*

	Gatsby-v2	**Next.js**	**React App**
First Contentful Paint	1.33s	1.91s	2.3s
Largest Contentful Paint	3.41s	1.91s	2.94s
Time to Interactive	2.65s	2.43s	2.68s
Max Potential first input delay	167ms	213ms	51ms

Taulukon 5.2 tuloksista nähdään, että Gatsby tuottaa myös tässä tapauksessa nopeimman ensimmäisen sisällön piirron. Käyttäjälle merkittävän sisällön piirtoon Gatsby-v2:lla kestää kuitenkin 78,5% enemmän aikaa kuin Next.js:llä ja 16% enemmän kuin React App:lla.

Jos Gatsby-v1 ja Gatsby-v2 toteutuksia verrataan keskenään, voidaan huomata, että Gatsby-v2 merkityksellisen sisällön piirtoon kului 143,6% enemmän aikaa kuin Gatsby-v1-toteutuksessa. Gatsby-toteutuksissa on siis suuri ero siinä, saadaanko käyttäjälle merkityksellistä sisältöä asetettua paikoilleen jo generointi vaiheessa.

5.3 Tuotesivulle navigointi sovelluksen sisällä (Gatsby-v1)

Tässä mittauksessa mitattiin toteutusten suorituskykyä tilanteessa, missä käyttäjä navigoi sovelluksen sisällä tuotesivulle. Mittaus suoritettiin käyttämällä Google Chrome -selaimen suorituskykytyökalua. Konkreettisesti mittaus suoritettiin avaamalla prototyyppisovellus tuotteet-sivulta ja navigoimalla linkin kautta yksittäisen tuotteen sivulle. Suorituskykytyökalulla tallennettiin navigointi linkin painalluksesta sivun latautumisen valmistumiseen. Tämän jälkeen työkalun aikajanalta voitiin laskea avainarvot.

Gatsby:ssä linkin takainen sivu esiladataan jos linkin yli viedään kursori. Tästä syystä Gatsby-toteutukselle tehtiin mittaus kahdella eri tavalla, joista ensimmäisessä linkkiä painettiin heti kursorin mentyä linkin päälle. Toisessa mittauksessa kursori vietiin linkin päälle ja odotettiin esilatauksen valmistuminen ennen linkin painamista. Näin saadaan mitattua

Gatsbyn suorituskyky myös tilanteessa, missä esilataus on suoritettu ennen linkin painamista.

Taulukossa 5.3 on listattu Gatsby-v1, Next.js ja React App -mittaustulokset. Tässä mittauksessa Gatsbyn linkkiä painetaan heti kursorin ollessa sen päällä.

Taulukko 5.3. *Tuotesivulle navigointi sovelluksen sisällä (Gatsby-v1) -mittaustulokset*

	Gatsby-v1	**Next.js**	**React App**
First Contentful Paint	1.05s	0.97s	0.2s
Largest Contentful Paint	1.05s	0.97s	0.75s
Time to Interactive	1.05s	0.97s	0.2s
Max Potential first input delay	39.5ms	25ms	20ms

Taulukosta 5.3 nähdään, että sovelluksen sisäisessä navigoinnissa React App on nopein. React App:lla kestää tuotetietojen piirtämiseen 28,6% vähemmän aikaa kuin Gatsby-v1:llä ja 22,7% vähemmän kuin Next.js:llä.

Taulukossa 5.4 on listattuna mittaustulokset tilanteessa, missä Gatsbyn mouseover-tapahtuman laukaisema esilataus on valmistunut ennen linkin painamista.

Taulukko 5.4. *Tuotesivulle navigointi mouseover-tapahtuman laukaiseman esilatauksen jälkeen (Gatsby-v1) -mittaustulokset*

	Gatsby-v1	**Next.js**	**React App**
First Contentful Paint	0.04s	0.97s	0.2s
Largest Contentful Paint	0.04s	0.97s	0.75s
Time to Interactive	0.04s	0.97s	0.2s
Max Potential first input delay	6.5ms	25ms	20ms

Taulukon 5.4 tuloksista nähdään, että Gatsby-v1 piirtää sivun näytölle lähes täysin viiveettä jos mouseover-tapahtuman laukaisema esilataus on valmistunut.

5.4 Tuotesivulle navigointi sovelluksen sisällä (Gatsby-v2)

Viimeisessä mittauksessa mitattiin Gatsby-v2 toteutuksen navigointi tuotesivulle sovelluksen sisällä. Taulukossa 5.5 on Gatsby-v2 mittauksen tulokset yhdistettynä Next.js ja React App -mittaustuloksiin.

Taulukko 5.5. *Tuotesivulle navigointi sovelluksen sisällä (Gatsby-v2) -mittaustulokset*

	Gatsby-v2	**Next.js**	**React App**
First Contentful Paint	0.8s	0.97s	0.2s
Largest Contentful Paint	1.63s	0.97s	0.75s
Time to Interactive	0.8s	0.97s	0.2s
Max Potential first input delay	5ms	25ms	20ms

Taulukon 5.5 tuloksista nähdään, että Gatsby-v2:lla kestää kauiten piirtää käyttäjälle merkittävää sisältöä. Gatsby-v2:lla kestää 68,0% enemmän aikaa kuin Next.js:llä ja 117,33% enemmän kuin React App:lla.

Taulukko 5.6. *Tuotesivulle navigointi mouseover-tapahtuman laukaiseman esilatauksen jälkeen (Gatsby-v2) -mittaustulokset*

	Gatsby-v2	**Next.js**	**React App**
First Contentful Paint	0.04s	0.97s	0.2s
Largest Contentful Paint	0.83s	0.97s	0.75s
Time to Interactive	0.04s	0.97s	0.2s
Max Potential first input delay	9.5ms	25ms	20ms

Taulukon 5.6 tuloksista nähdään, että mouseover-tapahtuman laukaiseman esilatauksen valmistuttua, Gatsby-v2 piirtää käyttäjälle merkittävää sisältöä nopeampaa kuin Next.js, mutta hitaammin kuin React App.

6 TULOKSET JA ARVIOINTI

Tässä luvussa käydään läpi tutkimuksen tulokset perustuen suorituskykymittauksiin sekä työn aikana opittuihin asioihin. Saatujen tuloksien ja opittujen asioiden pohjalta arvioidaan mikä toteutustavoista soveltuu eri käyttötarkoituksiin parhaiten.

Luvussa 3.7 käytiin läpi toteutustapojen heikkouksia ja vahvuuksia. Jos näitä verrataan mittaustuloksiin, voidaan todeta, että mittaustulokset vastasivat pääosin odotuksia. Mittaustuloksista on nähtävissä CSR:n heikkoudet, sillä CSR-toteutus oli hitain ensipiirrossa ja toiseksi hitain käyttäjälle merkillisen sisällön piirrossa. CSR:n vahvuus ensilatauksen jälkeen on myös todettavissa mittaustuloksista, sen hävitessä sovelluksen sisäisessä navigoinnissa ainoastaan esiladatulle Gatsby-v1 toteutukselle. Next.js:n vahvuus ensilatauksen nopeudessa käy myös ilmi, sen tuottaessa käyttäjälle merkityksellisen sisällön piirto toiseksi nopeiten tilanteessa, missä kyseinen sisältö luokiteltiin harvoin päivittyväksi. Next.js tuotti käyttäjälle merkityksellisen sisällön nopeiten tilanteessa, missä sisältö luokiteltiin usein päivittyväksi.

Hieman yllättävä tulos mittauksissa oli Next.js:n ja Gatsby-v1 merkityksellisen piirron nopeus sovelluksen sisäisessä navigoinnissa. Next.js teki merkityksellisen sisällön piirron 29,3% hitaammin kuin React App ja Gatsby-v1 teki merkityksellisen sisällön piirron 40% hitaammin kuin React App. Erojen suuruus on yllättävän iso siihen nähden, että kaikki toteutuksista tekee muutoksen selaimessa.

Gatsbyn suorituskyky sovelluksen sisäisessä navigoinnissa riippuu paljon sekä käyttäjän toiminnasta että verkkoyhteydestä. Mitä nopeampi verkkoyhteys, sitä nopeammin esilataus on valmis. Mitä kauemmin käyttäjällä kestää painaa linkkiä sen jälkeen, kun kursori on ensimmäistä kertaa viety linkin päälle, sitä suuremmalla todennäköisyydellä esilataus on valmis. Nopean verkkoyhteyden omaavat käyttäjät kokevat navigoinnin esilatauksen ansiosta nopeimmaksi Gatsbyllä, kun taas hitaan verkkoyhteyden omaavat käyttäjät saattavat kokea Gatsbyn navigoinnin hitaammaksi kuin React App:in. On hyvä ottaa huomioon myös mobiilikäyttäjät, joilla ei yleensä ole kursoria käytössä.

Ennen kuin web-sovelluksen optimaalista toteutustapaa voidaan valita, tarvitsee kartoittaa sen käyttötarkoitus. Jos sovellukselle halutaan paljon uusia käyttäjiä, jotka navigoivat sivulle pääosin toisten sivujen kuten Googlen kautta, kannattaa panostaa ensilatauksen suorituskykyyn. Tässä tapauksessa CSR-ratkaisua ei kannata valita, koska sen vahvuuksiin ei kuulu ensilatauksen nopeus. Valittava toteutustapa ensilatauksen perusteella riippuu sekä sisällön päivittyvyyden luonteesta, että arviosta kumpi on tärkeämpää: no-

pea ensipiirto vai merkityksellisen sisällön näyttäminen käyttäjälle nopeasti. Jos merkityksellinen sisältö luokitellaan harvoin päivittyväksi, on SSG-ratkaisu hyvä valinta, sillä se on suorituskyvyltään nopein ensilatauksessa. Jos kuitenkin merkittävä sisältö päivittyy usein, täytyy vielä arvioida, kumpi on tärkeämpää: nopea ensipiirto vai merkityksellisen sisällön näyttäminen nopeasti. Jos nopea ensipiirto on tärkeämpää, voidaan edelleen valita SSG-ratkaisu, koska se on joka tapauksessa nopein tuottamaan käyttäjälle ensipiirto. Jos merkityksellisen sisällön saaminen käyttäjän näkyviin on tärkeämpää, niin SSR on paras valinta sen ollessa nopein merkityksellisen sisällön piirtämisessä myös tilanteessa missä sisältö luokitellaan usein päivittyväksi.

Jos ensilatauksen suorituskyky ei ole tärkeä, mutta sovelluksen sisäisen navigoinnin suorituskyky on tärkeä, voidaan SSR-ratkaisu jättää pois harkinnasta. SSR-ratkaisun vahvuuksiin ei kuulu sovelluksen sisäisen navigoinnin suorituskyky, sen lisäksi että SSR vaatii ohjelmakoodia suorittavan palvelimen. Valittavaksi toteutustavaksi jäljelle jää siis SSG-ratkaisu ja CSR-ratkaisu. Jos tiedetään, että käyttäjillä on hitaat verkkoyhteydet ja mobiililaitteet, niin kannattaa ratkaisuksi valita CSR. CSR-ratkaisu on hyvä valinta, sillä se suoriutui parhaiten sovelluksen sisäisestä navigoinnista, eikä käyttäjät hyödy SSG-ratkaisun esilatauksista mobiililaitteilla ja hitaalla verkkoyhteydellä. Jos kuitenkin tiedetään, että sovellusta käytetään työpöytäkoneilla ja nopeilla verkkoyhteyksillä, voi myös SSG-ratkaisu olla hyvä valinta toteutukselle. Tässä tapauksessa, jos koko sivun ei tarvitse olla ajan tasalla, kannattaa valita SSG-ratkaisu, sen tuottaessa esiladattuja sivuja todella nopeasti. Jos sivun tarvitsee olla ajan tasalla, voidaan silti vielä harkita SSG-ratkaisua. Esiladattuna SSG-ratkaisun tuottama ensipiirto on nopeampi kuin CSR-ratkaisussa, jolloin käyttökokemus voi olla parempi, vaikka merkityksellisen sisällön piirtoon kuluu pidempi aika.

Taulukkoon 6.1 on listattu esimerkkejä toteutustavan suosituksista ensilatauksen suorituskyvyn ollessa tärkeä. Kolumnin otsikossa olevan ehdon toteutuminen on merkattu *x*-merkillä ja rivin suositus on viimeisenä.

Taulukko 6.1. *Suosituksia toteutustavan valinnalle ensilatauksen suorituskyvyn ollessa tärkeä*

Ensilatauksen suorituskyky on tärkeä			
Koko sisällön on oltava ajan tasalla	**Nopea merkityksellisen sisällön piirto on tärkeä**	**Nopea ensipiirto on tärkeä**	**Suositus**
x	**x**		**SSR**
x		**x**	**SSG**
	x		**SSG**
		x	**SSG**

Taulukkoon 6.2 on listattu esimerkkejä suosituksista sovelluksen sisäisen suorituskyvyn ollessa tärkeä. Kolumnin otsikossa olevan ehdon toteutuminen on merkattu *x*-merkillä ja rivin suositus on viimeisenä. Solussa lukeva ”*(ei vaikutusta)*” tarkoittaa sitä, että kyseisen

kolumnin ehdon toteutumisella ei ole vaikutusta sen rivin suositukseen.

Taulukko 6.2. *Suosituksia toteutustavan valinnalle sovelluksen sisäisen navigoinnin suorituskyvyn ollessa tärkeä*

Sovelluksen sisäinen suorituskyky on tärkeä			
Koko sisällön on oltava ajan tasalla	**Käyttäjillä on hidas verkkoyhteys / mobiililaite**	**Nopea ensipiirto on tärkeä**	**Suositus**
(ei vaikutusta)	**x**	(ei vaikutusta)	**CSR**
x			**CSR**
x		**x**	**SSG**
		(ei vaikutusta)	**SSG**

Taulukoiden 6.1 ja 6.2 suositukset ovat esimerkkejä jotka pohjautuvat tämän tutkimuksen tuloksiin. Mitä enemmän käyttäjistä ja käyttötarkoituksesta tiedetään, sitä paremman suosituksen toteutustavalle voi antaa.

7 YHTEENVETO

Dynaamisten web-sovellusten toteutukseen on kolme varteenotettavaa vaihtoehtoa: asiakaspään renderointi, palvelinpään renderointi sekä staattisen sivuston generaattori. Toteutustavat eroavat merkittävimmin siinä, missä vaiheessa sisältö luodaan HTML:ään ja missä vaiheessa tiedonhaku tietokannasta suoritetaan.

Tässä työssä tutkittiin miten asiakaspään renderoinnin, palvelinpään renderoinnin sekä staattisen sivuston generaattorin avulla toteutetut dynaamiset web-sovellukset eroavat toisistaan ja mitkä ovat niiden heikkouksia ja vahvuuksia. Epäoptimaalisen tekniikan valinta web-sovelluksen toteutukseen voi aiheuttaa pitkiä latausaikoja, jolloin käyttökokemus kärsii ja käyttäjä saattaa hylätä sovelluksen. Tällä on esimerkiksi verkkokaupan tapauksessa suora vaikutus myyntiin.

Tutkimus toteutettiin konstruktiivisen tutkimuksen menetelmällä toteuttamalla prototyyppisovellus jokaisella edellä mainitulla toteutustavalla. Toteutustekniikoina prototyypin toteutukselle oli React, Next.js ja Gatsby. Prototyypin tieto haettiin PostgreSQL-tietokannasta GraphQL-palvelimen kautta, joka luotiin Herokussa käyttöönotetun Hasura-moottorin avulla.

Tutkimuksen pohjalta voidaan todeta CSR-ratkaisun olevan varteenotettava vaihtoehto ainoastaan tilanteessa, missä ensilatauksen nopeudella ei ole käyttäjälle väliä. CSR tuottaa kaikista hitaimmin ensipiirron käyttäjälle, sillä se joutuu aina lataamaan ja parsimaan JavaScriptin voidakseen piirtää mitään sisällöllistä käyttäjälle. CSR-ratkaisun vahvuuksia on kuitenkin ensilatauksen jälkeinen toiminta, koska kaikki sovelluksen muutokset tehdään kokonaan selaimessa.

SSG-ratkaisun vahvuudet ovat ensipiirron nopeus sekä staattinen sisältö. Mitä enemmän käyttäjälle merkitsevää sisältöä voidaan asettaa paikoilleen sivuston generointivaiheessa, sitä parempi vaihtoehto toteutukselle SSG-ratkaisu on. SSG-ratkaisun heikkous on dynaaminen sisältö. Jos mitään käyttäjälle merkityksellistä sisältöä ei voida asettaa paikalleen generointivaiheessa, kestää sen piirtämisessä käyttäjälle toteutuksista kaikista kauiten. Tässäkin tapauksessa SSG-ratkaisu tuottaa ensipiirron käyttäjälle kaikista nopeiten, joten punnittavaksi jää kumpi on käyttäjälle tärkeämpää: nopea ensipiirto vai nopea merkityksellinen sisältö.

SSR-ratkaisun vahvuudet ovat dynaamisessa sisällössä. Mitä enemmän usein päivittyvää sisältöä sivusto sisältää, sitä parempi vaihtoehto toteutukselle SSR-ratkaisu on. SSR-ratkaisun heikkouksia on kuitenkin navigointi sovelluksessa ensilatauksen jälkeen.

Optimaalisen toteutustavan valinta riippuu käyttötarkoituksesta sekä sisällön luonteesta. Jos ensilatauksen nopeudella ei ole väliä, mutta sisällön täytyy olla ajan tasalla, valitsisin toteutustavaksi CSR-ratkaisun. Muissa tapauksissa valitsisin toteutustavaksi joko SSR- tai SSG-ratkaisun. Jos ensilatauksen nopeus on tärkeää, mutta ei ennalta tiedä kuinka ajan tasalla sisällön täytyy olla tai pystyykö SSG:n generointeja suorittamaan tarvittaessa, valitsisin toteutustavaksi SSR-ratkaisun. Muissa tapauksissa valitsisin toteutustavaksi SSG-ratkaisun. Mitä enemmän käyttötarkoituksesta ja käyttäjistä tiedetään, sitä paremmin voidaan optimaalinen valinta tehdä.

Työn toteutuksen aikana Next.js julkaisi päivityksen, jossa kerrottiin, että sovelluskehykseen on lisätty SSG-tuki. Päivityksessä kerrotaan, että ”Next.js on ensimmäinen hybridisovelluskehys, jossa voit halutessasi valita jokaiselle sivulle erikseen parhaiten soveltuvan tekniikan” [25]. Next.js:n SSG-tuen tutkiminen ei kuitenkaan ehtinyt työhön, joten se olisi mielenkiintoinen jatkotutkimuskohde. Käyttäjien käyttäytymisen tutkiminen verkkosivustolla olisi myös hyvä jatkotutkimuskohde, sillä esimerkiksi Gatsbyn toteutuksessa huomattiin käyttäjän toimintojen vaikuttavan latausnopeuksiin. Jokainen toteutustavoista on optimoitavissa hyvinkin pitkälle. Koska tähän työhön ei mahtunut toteutustapojen lisäoptimointi, jatkotutkimuksena voisi verrata toteutustapoja optimoituna.

LÄHTEET

[1] Apollo GraphQL Docs. *Apollo Boost migration*. (Viitattu: 26.1.2020) Saatavissa: URL: https://www.apollographql.com/docs/react/migrating/boost-migration/.

[2] E. Bocchi, L. De Cicco ja D. Rossi. Measuring the Quality of Experience of Web Users. *SIGCOMM Comput. Commun. Rev.* 46.4 (joulukuu 2016). (Viitattu: 22.4.2020) Saatavissa: URL: https://doi.org/10.1145/3027947.3027949.

[3] Codecademy. *React: The Virtual DOM | Codecademy*. (Viitattu: 16.2.2019.) Saatavissa: URL: https://www.codecademy.com/articles/react-virtual-dom.

[4] S. Deshmukh, D. Mane ja A. Retawade. Building a Single Page Application Web Front-end for E-Learning site. *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*. 2019, 985–987.

[5] GatsbyJS. *Behind the Scenes: What makes Gatsby Great*. (Viitattu: 20.4.2020) Saatavissa: URL: https://www.gatsbyjs.org/blog/2019-04-02-behind-the-scenes-what-makes-gatsby-great/.

[6] GatsbyJS. *Code Splitting and Prefetching*. (Viitattu: 20.4.2020) Saatavissa: URL: https://www.gatsbyjs.org/docs/how-code-splitting-works/.

[7] GatsbyJS. *Gatsby Link API*. (Viitattu: 20.4.2020) Saatavissa: URL: https://www.gatsbyjs.org/docs/gatsby-link/.

[8] GatsbyJS. *Gatsby Node APIs*. (Viitattu: 7.4.2020) Saatavissa: URL: https://www.gatsbyjs.org/docs/node-apis/.

[9] GatsbyJS. *gatsby-source-graphql*. (Viitattu: 4.4.2020) Saatavissa: URL: https://www.gatsbyjs.org/packages/gatsby-source-graphql/.

[10] GatsbyJS. *The gatsby-browser.js API file*. (Viitattu: 5.4.2020) Saatavissa: URL: https://www.gatsbyjs.org/docs/api-files-gatsby-browser/.

[11] GatsbyJS. *Understanding React Hydration*. (Viitattu: 10.1.2020) Saatavissa: URL: https://www.gatsbyjs.org/docs/react-hydration/.

[12] Google. *Increase the speed of your mobile site with this toolkit*. (Viitattu: 22.4.2020) Saatavissa: URL: https://www.blog.google/products/admanager/increase-speed-of-your-mobile-site-wi/.

[13] Google Developers. *First Contentful Paint | Tools for Web Developers | Google Developers*. (Viitattu: 27.4.2020) Saatavissa: URL: https://developers.google.com/web/tools/lighthouse/audits/first-contentful-paint.

[14] Google Developers. *Lighthouse | Tools for Web Developers | Google Developers*. (Viitattu: 15.11.2019) Saatavissa: URL: https://developers.google.com/web/tools/lighthouse.

[15] Google Developers. *Rendering on the Web | Google Developers*. (Viitattu: 28.2.2019.) Saatavissa: URL: https://developers.google.com/web/updates/2019/02/rendering-on-the-web.

[16] Google Developers. *Render-tree Construction, Layout, and Paint | Web Fundamentals*. (Viitattu: 16.2.2019.) Saatavissa: URL: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction.

[17] Google Developers. *Speed is now a landing page factor for Google Search and Ads | Web*. (Viitattu: 25.4.2020) Saatavissa: URL: https://developers.google.com/web/updates/2018/07/search-ads-speed.

[18] Javascript.info. *An Introduction to JavaScript*. (Viitattu: 27.4.2020) Saatavissa: URL: https://javascript.info/intro.

[19] MDN Web Docs. *CSS: Cascading Style Sheets*. (Viitattu: 14.2.2019.) Saatavissa: URL: https://developer.mozilla.org/en-US/docs/Web/CSS.

[20] MDN Web Docs. *HTML*. (Viitattu: 16.2.2019.) Saatavissa: URL: https://developer.mozilla.org/en-US/docs/Web/HTML.

[21] MDN Web Docs. *JavaScript*. (Viitattu: 14.2.2019.) Saatavissa: URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[22] MDN Web Docs. *Specificity*. (Viitattu: 14.2.2019.) Saatavissa: URL: https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity.

[23] Mosh Hamedani. *React Virtual DOM Explained in Simple English - Programming with Mosh*. (Viitattu: 16.2.2019.) Saatavissa: URL: https://programmingwithmosh.com/react/react-virtual-dom-explained/.

[24] Nextjs.org. *Automatic Static Optimization - Documentation | Next.js*. (Viitattu: 31.12.2019) Saatavissa: URL: https://nextjs.org/docs/advanced-features/automatic-static-optimization.

[25] Nextjs.org. *Blog - Next.js 9.3 | Next.js*. (Viitattu: 25.4.2020) Saatavissa: URL: https://nextjs.org/blog/next-9-3.

[26] Nextjs.org. *Data Fetching: getInitialProps | Next.js*. (Viitattu: 20.4.2020) Saatavissa: URL: https://nextjs.org/docs/api-reference/data-fetching/getInitialProps.

[27] Nextjs.org. *Documentation - Getting Started | Next.js*. (Viitattu: 20.4.2020) Saatavissa: URL: https://nextjs.org/docs/old.

[28] Nextjs.org. *Learn - Getting Started | Next.js*. (Viitattu: 31.12.2019) Saatavissa: URL: https://nextjs.org/learn/basics/getting-started.

[29] Nextjs.org. *Learn - Lazy Loading Modules | Next.js*. (Viitattu: 31.12.2019) Saatavissa: URL: https://nextjs.org/learn/excel/lazy-loading-modules.

[30] React Router Website. *React Router: Declarative Routing for React*. (Viitattu: 22.1.2020) Saatavissa: URL: https://reacttraining.com/react-router/web/api/Switch.

[31] Reactjs.org. *Higher-Order Components – React*. (Viitattu: 26.1.2020) Saatavissa: URL: https://reactjs.org/docs/higher-order-components.html.

[32] Reactjs.org. *React – A JavaScript library for building user interfaces*. (Viitattu: 14.11.2019.) Saatavissa: URL: https://reactjs.org/.

[33] Tech Differences. *Difference Between Server-side Scripting and Client-side Scripting (with Comparison Chart) - Tech Differences*. (Viitattu: 16.2.2019.) Saatavissa: URL: https://techdifferences.com/difference-between-server-side-scripting-and-client-side-scripting.html.

[34] web.dev. *Largest Contentful Paint (LCP)*. (Viitattu: 27.4.2020) Saatavissa: URL: https://web.dev/lcp/.

[35] web.dev. *User-centric performance metrics*. (Viitattu: 18.4.2020) Saatavissa: URL: https://web.dev/user-centric-performance-metrics/#first_paint_and_first_contentful_paint.

[36] Webpack. *Hot Module Replacement | webpack*. (Viitattu: 3.1.2020) Saatavissa: URL: https://webpack.js.org/concepts/hot-module-replacement/.

[37] R. York. *Beginning CSS: Cascading Style Sheets for Web Design*. IN: Wiley Pub, 2005.

[38] Zac Gordon. *What is Gatsby JS and Why Use It?* (Viitattu: 3.1.2020) Saatavissa: URL: https://javascriptforwp.com/what-is-gatsby-js-and-why-use-it/.

[39] Zeit.co. *Next.js - Solutions - ZEIT*. (Viitattu: 31.12.2019) Saatavissa: URL: https://zeit.co/solutions/nextjs.

A TUOTESIVUN ENSILATAUKSEN MITTAUSTULOKSET

First Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4
Next.js	1.9	2.0	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9
React App	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3

Speed Index	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
Next.js	1.9	2.0	1.9	1.9	1.9	1.9	1.9	1.8	1.9	1.9
React App	2.9	2.9	2.8	3.0	2.8	2.8	2.8	2.8	2.9	2.8

Time to Interactive	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	2.6	2.6	2.6	3.3	2.7	2.6	2.6	3.3	2.7	3.3
Next.js	2.5	2.5	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.5
React App	3.1	3.1	2.3	3.1	2.3	2.3	2.3	2.9	3.1	2.3

Largest Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4
Next.js	1.9	2.0	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9
React App	2.9	2.9	2.9	2.9	2.9	2.9	2.9	3.0	3.2	2.9

Max potential FID	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	130	130	130	130	130	140	130	140	140	140
Next.js	250	200	230	200	210	200	200	200	200	240
React App	50	50	50	50	50	50	50	60	50	50

B TUOTESIVUN ENSILATAUKSEN MITTAUSTULOKSET (GATSBY-V2)

First Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v2	1.4	1.3	1.4	1.3	1.3	1.4	1.3	1.3	1.3	1.3
Speed Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Gatsby-v2	2.9	3.0	2.9	3.0	3.0	3.0	2.9	2.8	2.9	2.9
Time to Interactive	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Gatsby-v2	2.6	2.6	2.7	2.6	2.6	2.7	2.6	2.8	2.7	2.6
Largest Contentful Paint	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Gatsby-v2	3.3	3.5	3.4	3.4	3.5	3.5	3.4	3.4	3.3	3.4
Max potential FID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Gatsby-v2	150	150	170	150	150	180	170	220	180	150

C TUOTESIVUN MITTAUSTULOKSET NAVIGOIDESSA SOVELLUKSEN SISÄLLÄ

First Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	0.8	1.4	1.1	1.0	1.0	1.3	0.9	1.0	1.0	1.0
Next.js	0.8	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
React App	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2

Time to Interactive	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	0.8	1.4	1.1	1.0	1.0	1.3	0.9	1.0	1.0	1.0
Next.js	0.8	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
React App	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2

Largest Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	0.8	1.4	1.1	1.0	1.0	1.3	0.9	1.0	1.0	1.0
Next.js	0.8	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
React App	0.8	0.7	0.7	0.7	0.8	0.8	0.8	0.7	0.8	0.7

Max potential FID	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	40	40	45	40	35	40	40	35	40	40
Next.js	25	25	25	25	25	25	25	25	25	25
React App	25	20	20	20	20	20	20	15	20	20

D TUOTESIVUN MITTAUSTULOKSET NAVIGOIDESSA SOVELLUKSEN SISÄLLÄ (GATSBY-V2)

First Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v2	0.9	0.9	0.6	0.9	0.8	0.8	0.8	0.8	0.7	0.8
Time to Interactive	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Gatsby-v2	0.9	0.9	0.6	0.9	0.8	0.8	0.8	0.8	0.7	0.8
Largest Contentful Paint	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Gatsby-v2	1.7	1.7	1.4	1.7	1.6	1.6	1.6	1.7	1.7	1.6
Max potential FID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Gatsby-v2	5	5	5	5	5	5	5	5	5	5

E GATSBY NAVIGOINTI MOUSEOVER-TAPAHTUMAN JÄLKEEN (GATSBY-V1)

First Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	0.05	0.04	0.05	0.04	0.04	0.04	0.04	0.04	0.04	0.05
Time to Interactive	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	0.05	0.04	0.05	0.04	0.04	0.04	0.04	0.04	0.04	0.05
Largest Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	0.05	0.04	0.05	0.04	0.04	0.04	0.04	0.04	0.04	0.05
Max potential FID	1	2	3	4	5	6	7	8	9	10
Gatsby-v1	10	5	10	5	5	5	5	10	5	5

F GATSBY NAVIGOINTI MOUSEOVER-TAPAHTUMAN JÄLKEEN (GATSBY-V2)

First Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v2	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
Time to Interactive	1	2	3	4	5	6	7	8	9	10
Gatsby-v2	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
Largest Contentful Paint	1	2	3	4	5	6	7	8	9	10
Gatsby-v2	0.8	0.9	1.0	0.8	0.9	0.6	0.8	0.9	0.8	0.8
Max potential FID	1	2	3	4	5	6	7	8	9	10
Gatsby-v2	10	10	15	10	10	10	5	10	5	10

G LIGHTHOUSE MITTAUKSEN AJONAIKAISET ASETUKSET

URL	http://localhost:9000/products/16/
Fetch time	Apr 17, 2020, 8:20 PM GMT+3
Device	Emulated Desktop
Network throttling	562.5 ms HTTP RTT, 1,474.6 Kbps down, 675 Kbps up (DevTools)
CPU throttling	4x slowdown (DevTools)
User agent (host)	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.92 Safari/537.36
User agent (network)	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3694.0 Safari/537.36 Chrome-Lighthouse
CPU/Memory Power	1304