Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Juha Nuutinen

Yhden sivun applikaation optimointi palvelinpuolen renderöinnillä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

27.04.2020

Tekijä Otsikko Sivumäärä Aika	Juha Nuutinen Yhden sivun applikaation optimointi palvelinpuolen renderöinnillä 43 sivua 27.04.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander

Insinöörityössä tutkittiin palvelimen puolella tapahtuvaa React-applikaation renderöintiä ja selvitettiin, voidaanko sen avulla parannella yhden sivun applikaation suorituskykyä ja hakukoneoptimointia.

Työssä suunniteltiin ja toteutettiin kaksi SPA:ta, joista toinen toteutettiin täysin selaimessa renderöitynä ja toisessa käytettiin lisäksi palvelinpuolen renderöintiä sekä staattisia sivuja. Palvelinpuolen renderöintiin käytettiin Next-sovelluskehystä. Sovellusten suorityskykyä auditoitiin Googlen Lighthouse-työkalulla.

Palvelinpuolen renderöinti ja Reactin vieminen staattiseksi sivuksi nopeuttivat SPA-sivun ensimmäistä latautumista. Tällöin tyhjän sivun sijasta selain saikin palvelimelta sisällöllisen sivun, mikä nopeutti sivun näyttämistä käyttäjälle. Työssä huomattiin, että sivun kompleksisuuden kasvaessa palvelinrenderöinnistä saatavat suorituskykyhyödyt kasvoivat.

Hakukoneoptimoinnin todettiin olevan todennäköisesti helpompaa palvelinrenderöintiä käyttäen, joskin sitä ei työssä päästy todentamaan käytännössä hankaluutensa takia. Huomattiin, että sivun esikatselun näyttäminen ulkopuolisissa palveluissa on toimivampaa palvelinrenderöinnin kanssa.

Next todettiin toimivaksi työkaluksi palvelinrenderöityjen tai staattisten React-sivujen toteuttamiseen. Se on kuitenkin jokseenkin rajoittava työkalu, ja ongelmia voi mahdollisesti esiintyä joidenkin kirjastojen käytössä. Vaihtoehtoisesti palvelinrenderöintiin voi käyttää Reactin sisäänrakennettua tukea sille, mutta tällöin palvelimen toteuttaminen vaatii enemmän työtä.

Avainsanat	ohjelmointi, web, JavaScript, HTML, React, palvelin, SPA

Author Title Number of Pages Date	Juha Nuutinen Optimizing Single Page Application Using Server-Side Rendering 43 pages 27 April 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer

This bachelor's thesis studied server-side rendered React and examined if it can be used to improve the performance and search engine optimization of a single page application.

Two single page applications were designed and developed. The first was implemented using only client-side rendering, and in the second also server-side rendering and static exporting of pages were used. The framework used for server-side rendering and static exports was Next. The performance of the developed applications was audited using Google's Lighthouse tool.

Server-side rendering and static export of React pages sped up the initial load of the SPA. Instead of the client receiving an empty page from the server, they received a page that already had content which resulted in the user seeing the page sooner. It was noted that when the complexity of the app grew, the performance gains from using server-side rendering was also increased.

Search engine optimization was concluded to probably be easier to handle using server-side rendering, however, it could not be verified in the present study due to it being difficult to do so. Showing a preview of the page in external services was noticed to work better with server-side rendered pages.

Next was concluded to be a good tool for developing server-side rendered React applications. However, it can be somewhat restricting and opinionated, and there could be issues with some libraries when it comes to server-side rendering. As an alternative for server-side rendering using Next, one could use React's built-in support for SSR. It will require more work for developing the server but it could be easier to take into use in existing projects instead of porting the application to Next.

Keywords	programming, web, JavaScript, HTML, React, server, SPA

Sisällys

Lyhenteet

1 Johdanto 1

2 Renderöinti palvelimella ja selaimessa 2

2.1 Internetsivujen perustoimintaperiaate 2
2.2 Palvelinpuolen renderöinti 3
2.3 Selainpuolen renderöinti ja SPA 8
2.4 Isomorfinen web-applikaatio 12

3 Teknologioiden kartoitus 16

3.1 React 16
3.2 Next.js 18
3.3 Zeit Now 19

4 Demosivujen toteutus 20

4.1 Suunnittelu 21
4.2 Single page application -toteutus 22
4.3 Isomorfisen web-applikaation toteutus 28

5 Demosivujen vertailu ja auditoinnin tulokset 32

5.1 Raskaan sivuston simulointi 32
5.2 Suorituskyky 33
5.3 Hakukoneoptimointi 38

6 Yhteenveto 40

Lähteet 42

Lyhenteet

API	Application programming interface, ohjelmointirajapinta. Määritelmä, joka kuvailee ohjelmistojen kommunikointitoiminnallisuutta, web-kehityksessä tarkoittaa yleensä määritelmää datan pyytämiseen palvelimelta.
CRA	Create React App. Työkalu, joka alustaa valmiin pohjan uusille React-ohjelmille, mikä tekee kehityksen aloituksesta helppoa ja nopeaa.
CSR	Client-side rendering. Internetsivun toimintatapa, jossa sivua muokataan dynaamisesti selaimen puolella.
CSS	Cascading Style Sheets. Määrittelee HTML-sivun visuaalisen ulkonäön.
DOM	Document Object Model. Rajapinta, joka kuvaa HTML- tai XML-dokumentin puuna ja tarjoaa operaatiot puun olioiden manipulointiin.
HTML	HyperText Markup Language. Kuvauskieli, jolla web-sivujen rakenne määritellään.
JS	JavaScript. Ohjelmointikieli, jota käytetään useimmiten dynaamisten selainpuolen skriptien määrittämiseen internetsivuilla, mutta usein myös palvelimien ohjelmoimiseen.
SEO	Search engine optimization, hakukoneoptimointi. Prosessi, jossa sivuston kävijämäärää pyritään kasvattamaan tai hakutuloksia kohdentamaan paremmin parantamalla sivuston näkyvyyttä hakukoneissa.
SPA	Single-page application. Web-applikaatio, joka koostuu vain yhdestä sivusta, jonka osia päivitetään dynaamisesti käytön aikana.
SSR	Server-side rendering. Internetsivun toimintatapa, jossa sivu rakennetaan palvelimella ja palautetaan selaimelle näytettäväksi.

UWA	Universal web application. Web-applikaatio, joka on sekoitus palvelinrenderöintiä ja SPA:ta. Kutsutaan myös isomorfiseksi web-applikaatioksi.

1 Johdanto

Työn tavoitteena on tutkia palvelimen puolella tapahtuvaa React-applikaation renderöintiä ja selvittää, voiko tällä tavoin parannella tavallisen yhden sivun applikaation suorituskykyä sekä hakukoneoptimointia. Aihe valittiin sen takia, koska yhden sivun applikaatiot ovat suosittuja, mutta ne ovat saaneet osaksi kritiikkiäkin. Osa kritiikistä liittyy niiden mahdollisiin suorituskykyyn ja hakukoneoptimointiin liittyviin ongelmiin. Tutkimuksesta voisi olla hyötyä vastauksen saamisessa siihen, kannattaako esimerkiksi uutta web-projektia aloittaessa ottaa käyttöön työkalut, joilla palvelinrenderöintiä voi yhdistää yhden sivun applikaatioon.

Reactilla toteutetut sivut saattavat niiden kompleksisuuden kasvaessa kärsiä hitaasta sivun ensimmäisestä latauksesta, ja ne voivat myös olla hakukoneille haasteellisia. Tässä työssä toteutetaan kaksi ulkoisesti samanlaista web-applikaatiota, joista toinen on perinteinen yhden sivun applikaatio ja toinen käyttää hyväksi myös palvelinrenderöintiä. Tällöin sivujen ominaisuuksia päästään mittaamaan ja arvioimaan vertailukelpoisesti. Tuloksena työstä halutaan tietoa, saako palvelinrenderöinnistä sen verran mittavia hyötyjä, että sen toteuttamiseen kannattaisi pyrkiä Reactia käyttävässä projektissa.

2 Renderöinti palvelimella ja selaimessa

Perinteisesti nettisivut ovat toimineet siten, että ne pyytävät palvelimelta sivua ja saavat sen vastauksena. Sitten sivu näytetään käyttäjälle. Nykyisin on myös yleistynyt toinen tapa toteuttaa nettisivuja, ja se on selainpuolen renderöinti. Tällöin selain pyytää sivua palvelimelta, mutta palvelin palauttaakin tyhjän sivun. Sivuun on linkitetty koodia, jonka selain suorittaa. Koodin suorittaminen selaimessa rakentaa sivun, ja kun se on valmis, selain voi näyttää sen käyttäjälle. Tässä luvussa esitellään tarkemmin nämä kaksi toimintatapaa sekä niiden käyttötapoja.

2.1 Internetsivujen perustoimintaperiaate

Tässä luvussa esitellään lyhyesti internetsivujen toimintaperiaatetta sekä palvelin- ja selainpuolen renderöintiä. Web-sivun ydinteknologiat muodostavat HTML, CSS ja JavaScript. HTML eli HyperText Markup Language on kuvauskieli, jolla web-sivujen rakenne määritellään [1]. Se kertoo esimerkiksi, mikä osa sivun tekstistä on leipätekstiä, mikä otsikkoa ja mikä listaa. HTML määrittelee sivun rakennetta ottamatta kantaa sivun ulkonäköön. Sivun rakenteeseen liittyy käsite DOM (Document Object Model), joka tarkoittaa rajapintaa HTML-sivun rakenteeseen pääsyyn ja sen manipulointiin. CSS eli Cascading Style Sheets määrittelee sivun visuaalisen ulkonäön [2]. Jos HTML kertoo, että tietty elementti on nappi, niin CSS sen sijaan määrittelee kyseisen napin ulkonäön. Sivun interaktiivisuus ja dynaaminen toiminta määritellään JavaScript-ohjelmointikieltä käyttäen. JavaScript on perinteisesti ollut vain selaimessa suoritettava ohjelmointikieli, mutta nykyisin sitä voidaan käyttää muuallakin. JavaScriptin käyttäminen palvelimien ohjelmoinnissa on saanut suurta suosiota Node.js JavaScript runtime -ympäristön ansiosta. [3].

Toimintatapaa, jossa palvelin rakentaa valmiita sivuja selaimessa näytettäväksi, kutsutaan palvelinpuolen renderöinniksi (server-side rendering, eli SSR). Renderöinti käsitteenä voi toimia synonyyminä piirtämiselle, mutta tässä tapauksessa ne eivät tarkoita samaa asiaa. Selain hoitaa aina sivun visuaalisen piirtämisen. Kun puhutaan palvelinpuolen renderöinnistä, tarkoitetaan toimintaa, jossa palvelin generoi sivun rakennetta, ennen sen palauttamista selaimelle. Vastaavasti selaimessa tapahtuvaa sivun rakentamista kutsutaan selainpuolen renderöinniksi (client-side rendering, eli CSR). [4, s. 80.]

Selainpuolen renderöinti mahdollistaa niin kutsuttujen yhden sivun applikaatioiden (SPA, Single Page Application) laatimisen. SPA-sivut ovat suhteellisen uusi käsite internetin elinkaaressa, ja ne ovat saaneet suurta suosiota viime vuosien aikana.

Muutosta toimintatapoihin ja uusiin teknologioihin web-kehityksessä ajaa muun muassa se, että perinteisen dokumenttipohjaisen internetin sijaan internetiä käytetään yhä enemmän sovelluksien ja palveluiden tarjoamiseen. Sovelluksien täytyy tukea rikkaita käyttäjäinteraktioita, jotta ne olisivat hyödyllisiä. Perinteinen käsite sivuista koostuvasta internetistä ei aina sovellu hyvin web-applikaatioiden toteuttamiseen, mutta selainpuolen renderöinti ja SPA-sivut mahdollistavat monimutkaisempien ohjelmistojen toteuttamisen internetsivuina. Enää ei tarvitsekaan tehdä ohjelmasta työpöytäversioita eri käyttöjärjestelmille, vaan voidaan luoda web-applikaatio, joka toimii kaikkialla. Web-applikaatio työpöytäohjelman sijasta on helppo ratkaisu sekä kehittäjille että käyttäjille. Kehittäjien ei tarvitse ylläpitää montaa eri versiota työpöytäohjelmistosta, eikä käyttäjien tarvitse ladata uusia ohjelmia koneillensa.

Sekä palvelin- että selainpuolen renderöinnillä on omat vahvuutensa, joten valinta niiden kahden välillä ei ole aina yksiselitteinen vaan riippuvainen esimerkiksi tuotettavan sivuston ominaisuuksista. Palvelin- ja selainpuolen renderöinti eroavat toisistaan oleellisella tavalla myös esimerkiksi suorituskyvyn näkökulmasta. Palvelinpuolella suoritettava renderöinti käyttää palvelimen resursseja, ja selainpuolella tehtävä renderöinti käyttää loppukäyttäjän laitteen resursseja.

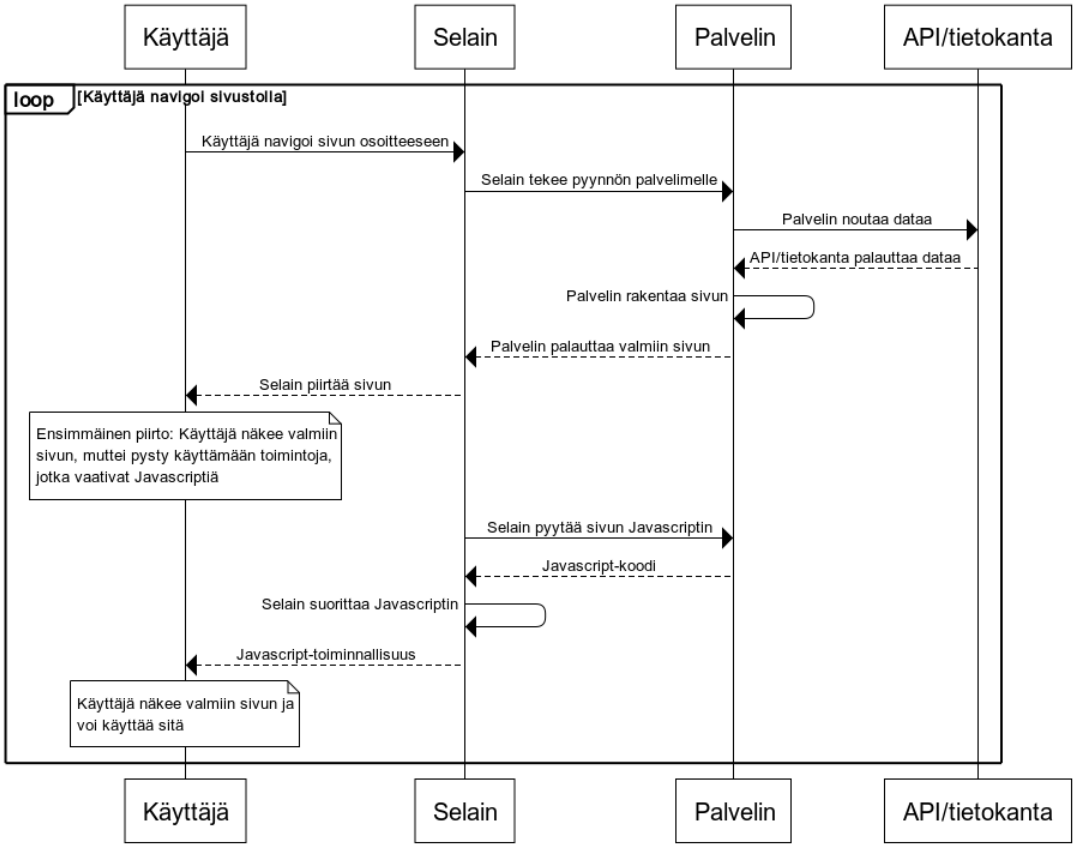
2.2 Palvelinpuolen renderöinti

Perinteisen toimintatavan mukaan web-sivut toimivat siten, että selain pyytää sivua palvelimelta, joka vastaa pyyntöön HTML-sivulla. Saatuaan sivun selain näyttää sen käyttäjälle. Tämä toimintatapa sopii erityisen hyvin sisällöltään staattisiin sivuihin. HTML-sivua voidaan ajatella dokumenttina, jonka käyttäjä pyytää palvelimelta, ja verkon selailua voi ajatella eri dokumenttien selailuna. Kun käyttäjä haluaa nähdä jonkin toisen sivun, hän navigoi uuteen osoitteeseen tai klikkaa hiirellä jotakin linkkiä, jonka jälkeen palvelin palauttaa selaimelle kokonaan uuden HTML-sivun. Internetin selaaminen sivu kerrallaan toimii hyvin varsinkin staattisen sisällön selailuun. Usein sivut eivät kuitenkaan ole vain

staattisten dokumenttien näyttämistä varten, vaan niihin sisältyy enemmän tai vähemmän ohjelmiston kaltaista toiminnallisuutta, jonka kautta käyttäjä voi olla vuorovaikutuksessa sivun kanssa. Tätä varten on olemassa ohjelmointikieli JavaScript, joka mahdollistaa interaktiivisten sivujen luomisen. JavaScript-skriptit sisällytetään tai linkitetään HTML-dokumentteihin, jolloin selaimen JavaScript-moottori voi suorittaa koodin selaimessa palvelimelta saadun sivun kautta. JavaScriptiä käytetään usein esimerkiksi sivuston sisällön dynaamiseen manipulointiin DOM:in kautta ja datan hakemiseen asynkronisesti ilman, että koko sivu täytyy ladata uudestaan tai että sivu jäätyy datan hakemisen ajaksi [3].

Palvelinpuolella sivujen tyyppi voidaan jaotella sen mukaan, ovatko sivut staattisia vai dynaamisia. Staattiset sivut tarkoittavat sitä, että palvelin tarjoaa palvelimella tallennettuja valmiita HTML-dokumentteja, ilman että se prosessoi niitä ennen tarjoamista. Palvelinpuolen dynaamiset sivut taas ovat sivuja, joita palvelin prosessoi, ennen kuin ne ovat valmiita tarjottavaksi. Prosessointiin liittyy yleensä datan hakemista ja HTML-koodin generoimista. Palvelinpuolen dynaamiset web-sivut toteutetaan palvelinpuolen skriptauksella. [5, s. 20.] Palvelinpuolen dynaamisten sivujen tarjoilua kutsutaan palvelinpuolen renderöinniksi, koska web-sivujen sisältöä generoidaan palvelimella. Pelkkien staattisten sivujen tarjoaminen palvelimelta ei siis vielä ole palvelinpuolen renderöintiä.

Kuva 1 havainnollistaa palvelinpuolen dynaamisien sivujen renderöinnin prosessia sekvenssikaavion avulla. Tässä tapauksessa se tarkoittaa sitä, että palvelin hakee jostakin tietolähteestä dataa, jota se käyttää sivun dynaamiseen rakentamiseen palvelimella. Tietolähde voi olla esimerkiksi API (application programming interface) eli rajapinta, josta voidaan kysyä dataa web-pyynnöillä.



Kuva 1. Sekvenssidiagrammi dynaamisesta palvelinpuolen renderöinnistä, kun sivuston tarvitsema data haetaan synkronisesti palvelimen puolella.

Kuvasta 1 nähdään, että joka kerta kun käyttäjä avaa uuden sivun sivustolla, selain pyytää sivua palvelimelta, joka palauttaa sen valmiiksi rakennettuna. Palvelin noutaa tarvittavan datan suoraan palvelimen puolella esimerkiksi tietokannasta tai jostakin rajapinnasta ja täyttää HTML-sivun datalla jo palvelimen puolella. Datan noutamisessa palvelimen puolella on se hyöty, että silloin päästään käyttämään hyödyksi palvelimien suuria laskentatehoja ja nopeita yhteyksiä tietokantoihin tai muihin palvelimiin, jolloin sivun rakentaminen voi käydä nopeammin kuin käyttäjän laitteella.

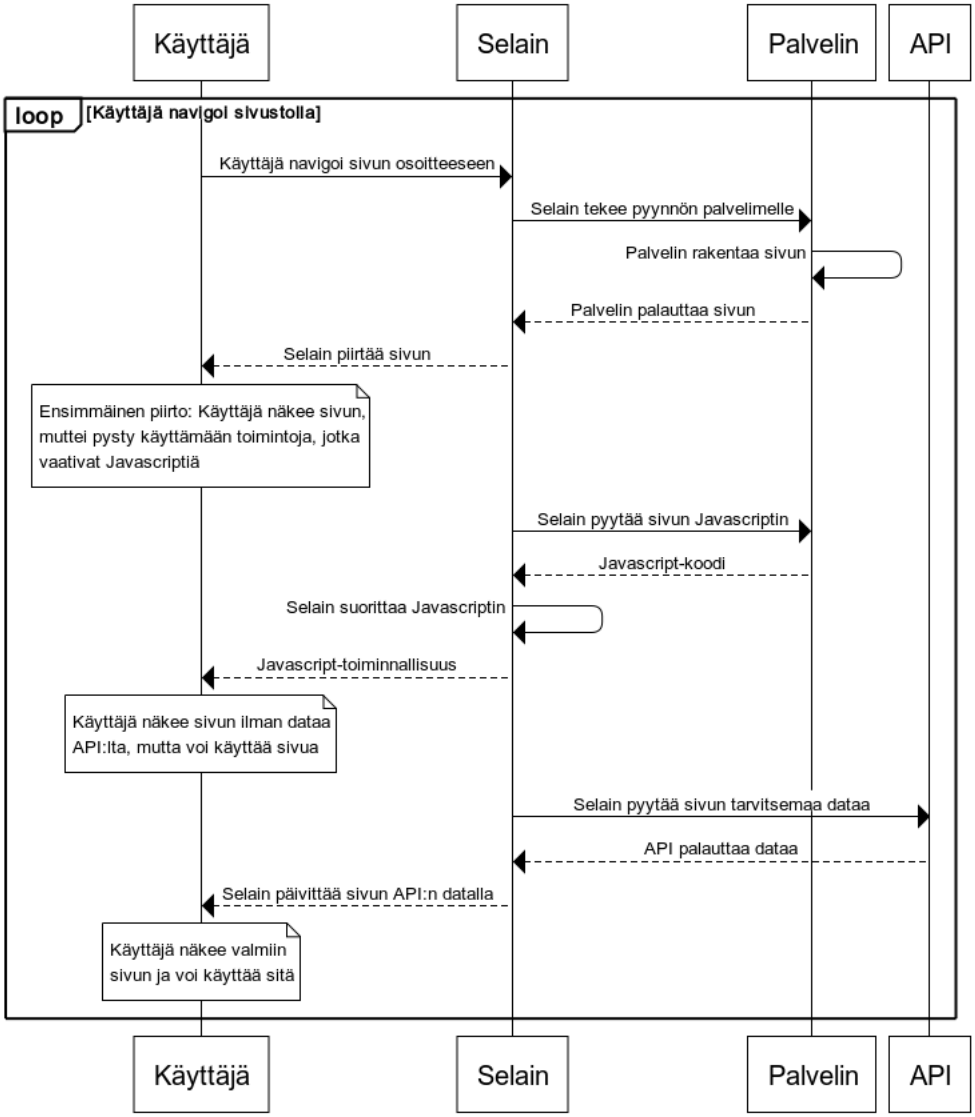
Kuvasta 1 nähdään myös, että selain piirtää palvelimelta haetun sivun kokonaan uudestaan aina, kun käyttäjä navigoi uudelle sivulle sivustolla. Sivun lataaminen ja piirtäminen kokonaan uudestaan saattaa johtaa töksähtelevään käyttäjäkokemukseen. Lisäksi se voi johtaa siihen, että sekä selain että palvelin tekevät turhaa työtä. Palvelinpuolella renderöity sivu piirretään alusta saakka uudestaan, vaikka vain tietty osa sivusta tarvitsisi päivittämistä.

Esimerkkinä tästä tilanteesta kuvitellaan sivusto, jossa on kaikilla sivuston sivuilla näkyvissä oleva navigaatiomenu. Kun käyttäjä navigoi sivujen välillä, navigaatiomenu pysyy muuttumattomana, mutta sivun sisältö vaihtuu. Tällöin koko sivun sisältöä tuskin rakennettaisiin JavaScriptillä dynaamisesti, vaan uudet sivut haettaisiin palvelimelta. Tällöin myös navigaatiomenu haetaan muun sivun mukana palvelimelta ja piirretään uudestaan, vaikka se pysyy muuttumattomana eri sivujen välillä. Uuden sivun lataaminen näkyy myös käyttäjälle, ja varsinkin hitaalla internetyhteydellä uusi sivu saattaa latautua viiveellä.

Kuvan 1 tapauksessa sivun tarvitsema ulkopuolinen data noudetaan palvelimella synkronisesti. Synkronisuus tässä tarkoittaa sitä, että palvelin odottaa datan hakemisen ajan ennen koodin suorituksen jatkamista. On tapauskohtaista, onko tämä menettelytapa järkevä ratkaisu vai ei. Esimerkkinä voisi olla ulkopuoliselta API:lta dataa näyttävä sivu, eli dataa ei haettaisi esimerkiksi palvelimella olevasta paikallisesta tietokannasta tai välimuistista. Tällöin tiedon hakemiseen saattaa liittyä huomattavaa viivettä, jonka suuruus on riippuvainen esimerkiksi API:in kohdistuvasta liikenteen määrästä sekä API-palvelimen omasta internetyhteyden nopeudesta. Pahimmassa tapauksessa tämä API-kutsu voi kuitenkin kestää jopa sekunteja. Kyseisen sivuston palvelin siis joutuisi odottaa näiden API-kutsujen valmistumista, ennen kuin se pääsee generoimaan ja palauttamaan pyydettyä sivua. Loppukäyttäjä näkee koko tämän ajan vain selaimen lataamassa sivua, kunnes valmis sivu lopulta latautuu kerralla.

API-kutsujen tekeminen palvelimen puolella edellä kuvatussa tapauksessa johtaa siis siihen, että loppukäyttäjät näkevät huomattavia viiveitä siinä, että sivu edes alkaa piirtymään heidän selaimessaan. Tähän liittyy käsite TTFB, eli time to first byte. TTFB kuvaa sen välistä ajanjaksoa, kun selain pyytää jotakin sivua ja kun selain saa vastauksena ensimmäisen tavun [6]. Vaihtoehto tähän API-kutsujen tekemiseen synkronisesti palvelimen puolella on käyttää asynkronisia kutsuja, jotka suoritetaan vasta selaimen puolella. Tällöin saadaan lyhyempi TTFB. API-kutsujen tekeminen asynkronisesti selaimessa ei oleellisesti vaikuta datan ilmestymisen nopeuteen, mutta se mahdollistaa muun sivun näyttämisen käyttäjälle jo ennen kuin hitaat API-kutsut ovat valmiita. Tällöin käyttäjän ei tarvitse katsella tyhjää selaimen sivua latauksen ajan, vaan hän näkee muun sivuston jo API-datan haun aikana. Tämä luo vaikutelman, että sivu toimii nopeammin. Asynkronisia API-kutsuja käytettäessä käyttäjä voi myös käyttää muuta sivua sillä aikaa, kun kutsuja

tehdään. Asynkroniset selaimessa tehdyt kutsut ovat ehkäpä yleisin tapa hakea dataa API:lta web-sivuissa. Kuvassa 2 esitetään kyseinen prosessi sekvenssikaavion avulla.



Kuva 2. Sekvenssidiagrammi palvelinpuolen renderöinnistä, kun sivuston tarvitsema data haetaan asynkronisesti selaimen puolella.

Kuten kuvasta 2 nähdään, tällä kertaa API-data haetaan tapahtumaketjussa myöhemmin kuin kuvan 1 esimerkissä. Käyttäjä näkee sivun nopeammin kuin aiemmassa esimerkissä, mutta API-datan perusteella rakennetut sivun elementit tulevat vasta viiveellä näkyviin.

2.3 Selainpuolen renderöinti ja SPA

SSR:n vastakohtana voidaan pitää täysin selaimen puolella tapahtuvaa renderöintiä. Siinä palvelin palauttaa selaimelle ainoastaan tyhjän HTML-sivun pohjan, johon linkitetty JavaScript hoitaa itse sivun rakentamisen ja manipuloimisen suoraan selaimessa. Selainpuolen renderöinti tarkoittaa yleisesti sitä, että sivua muokataan dynaamisesti selaimessa. Selainpuolen renderöintiä on siis mukana myös pääasiassa palvelimen puolella renderöidyissä sivuissa, jos ne käyttävät JavaScriptiä sivun muokkaamiseen selaimessa. Kun tässä työssä vertaillaan puhdasta palvelin- ja selainpuolen renderöintiä, niillä tarkoitetaan seuraavia määritelmiä:

- *Palvelinpuolen renderöinnillä* tuotettu sivu on rakennettu dynaamisesti palvelimella, mutta se saattaa sisältää myös selaimessa suoritettavaa, JavaScriptillä toteutettua dynaamista toiminnallisuutta.

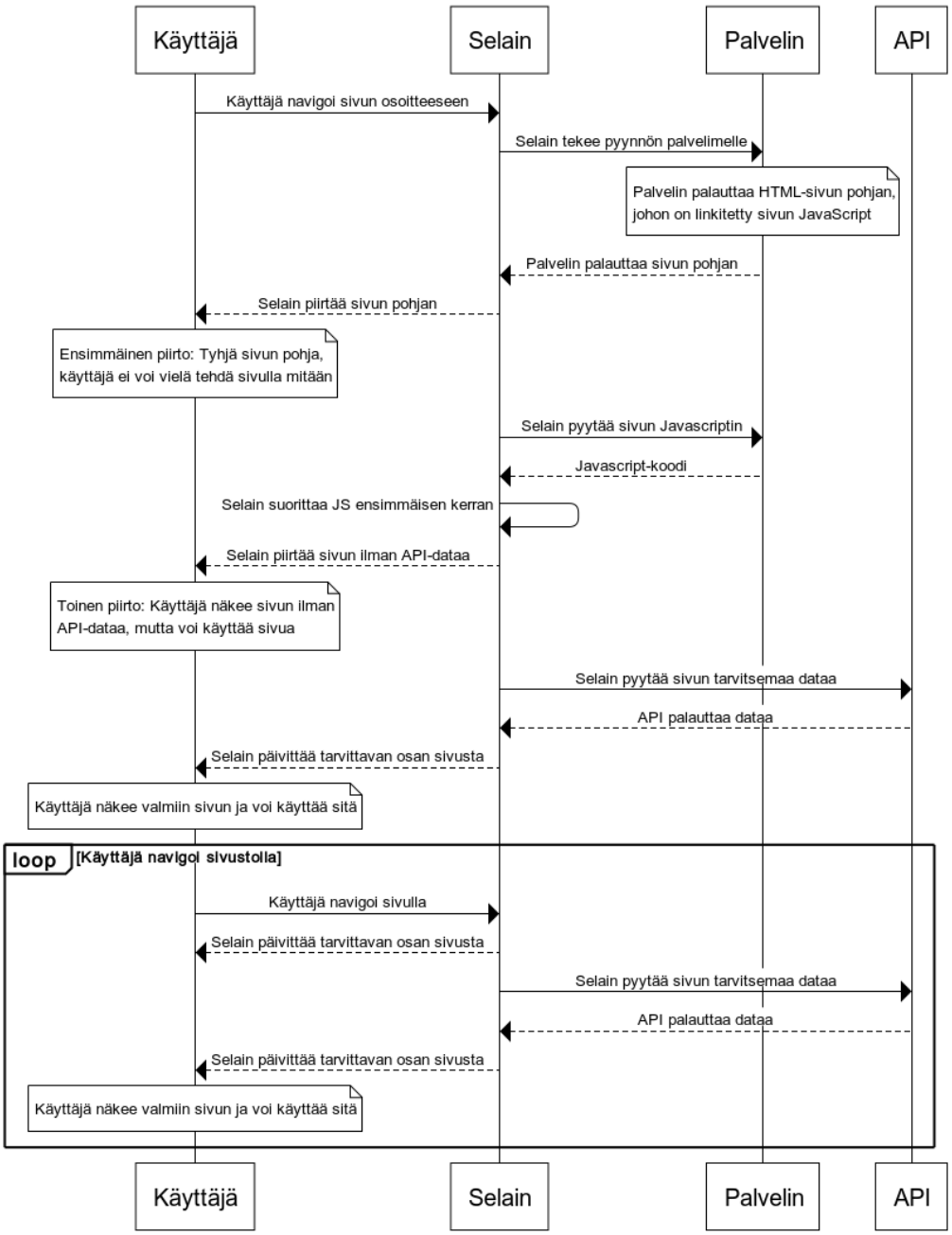
- *Selainpuolen renderöinnillä* tuotettu sivu rakennetaan kokonaan selaimessa, ja palvelin tarjoaa ainoastaan sivun pohjan ja JavaScript-koodin. Se on siis yhden sivun applikaatio.

Selainpuolen renderöinti SPA-kontekstissa on prosessi, jossa sivu rakennetaan käyttäjän selaimessa sen sijaan, että sivu tulisi valmiiksi rakennettuna palvelimelta. Selainpuolen renderöinti on uudempi tapa toimittaa sivuja loppukäyttäjille kuin palvelinpuolen renderöinti. CSR mahdollistaa sen, että sivua voidaan muokata tai uusia sivuja luoda suoraan selaimessa ilman, että selaimen pitää pyytää uusia sivuja palvelimelta. Tämä sivujen muokkaaminen ja luominen selaimessa tarjoaa usein nopeamman ja luontevamman käyttäjäkokemuksen. Se johtuu siitä, että sivun muokkaamista varten ei tarvitse pyytää kokonaan uutta sivua palvelimelta, vaan sivua muokataan paikallisesti käyttäjän laitteella päivittämällä sivun osia tarpeen mukaan.

Selainpuolen renderöinti tapahtuu siten, että ensiksi selain pyytää sivuston ”pohjan” palvelimelta, joka on usein (ainakin suurimmaksi osaksi) tyhjä HTML-sivu. Palvelimelta saatu sivun pohja sisältää linkin siihen JavaScript-koodiin, joka hoitaa sivun rakentamisen selaimessa. Kun sivun pohjaan linkitetty JavaScript on haettu palvelimelta, se suoritetaan selaimessa, jolloin sivua aletaan piirtämään selaimessa, ja alun perin tyhjänä ollut sivu alkaa saamaan sisältöä. Ensimmäisen JavaScript-koodin haun jälkeen CSR-

sivua päivitetään tarpeen mukaan suoraan selaimessa, eikä uusia sivuja tarvitse enää hakea palvelimelta. SPA-sivut eliminoivat tarpeen ajatella sivustoa kokoelmana erillisiä sivuja, joiden välillä käyttäjä navigoi; sen sijaan SPA-sivu voi muistuttaa enemmän mitä tahansa sovellusta, jonka sisältö muokkautuu dynaamisesti käytön mukaan. Selainpuolen renderöinti on yleensä toteutettu jotakin JavaScript-sovelluskehystä käyttäen. Näistä tunnetuimpia ovat React, Vue.js sekä Angular, joista React on käytössä tässä työssä.

Kuvassa 3 on esitetty samanlainen tilanne kuin kuvassa 1 ja kuvassa 2, mutta tällä kertaa selailtava sivusto on toteutettu kokonaan selainpuolen renderöintiä käyttäen. CSR-tyylillä toteutetun sivun toimintalogiikka sijaitsee selaimessa, ja se ladataan palvelimelta sivun pohjaan linkitetyn JavaScriptin kautta silloin, kun sivu avataan ensimmäisen kerran. Tuon koodin täytyy sisältää kaikki tarvittava ohjelmisto sivun rakentamiseen ja toimimiseen, eli esimerkiksi käytetyn JavaScript-sovelluskehyksen, kirjastojen sekä itse sivun toimintalogiikan koodit. Koodien lisäksi ensimmäisellä sivun latauskerralla selain hakee myös mahdolliset CSS-tyylit sekä fontit. Sen jälkeen, kun koodit on haettu selaimeen, se aloittaa koodin suorittamisen. Varsinkin isossa applikaatiossa koodin hakeminen verkon yli ja suorittaminen alusta asti sivun ensimmäisellä latauskerralla voi johtaa viiveeseen sivun latautumisessa. Ensimmäisen ja mahdollisesti hitaan sivun latautumisen jälkeen CSR-tyylillä toteutettu sivu toimii yleensä kuitenkin nopeasti. Silloin sivua päivitetään dynaamisesti selaimessa sen sijaan, että uusia sivuja haettaisiin aina verkon yli, tai että sivuston koodeja täytyisi enää hakea tai suorittaa aivan alusta saakka.



Kuva 3. Sekvenssidiagrammi selainpuolen renderöinnistä, jossa dataa noudetaan API:lta.

Sivun päivittäminen dynaamisesti selaimessa tarkoittaa sitä, että selain suorittaa JavaScriptiä, joka johtaa siihen, että vain tarvittavat elementit sivulla muutetaan. Kuvitellaan esimerkin vuoksi Reactilla toteutettu sivusto, jossa on kaikilla sivuilla näkyvissä oleva navigaatiopalkki. Sivut eroavat sisällöiltään, mutta navigaatiopalkki pysyy samana kaikilla eri sivuilla. Tällöin Reactilla toteutetulla sivulla selain piirtää sivua vaihtaessa sivun sisällön uudestaan, mutta navigaatiopalkkia sen ei tarvitse piirtää uudestaan. Kun

sivua vaihdetaan, React alkaa suorittamaan koodia, jonka johdosta React rakentaa uuden sivun DOM-rakenteen. Kun se on valmis, selain vertaa Reactin rakentamaa DOM-rakennetta tällä hetkellä sivulla näkyvissä olevaan rakenteeseen. Niiltä osin miltä rakenne eroaa, selain piirtää Reactilta saadun rakenteen. Identtiset elementit jätetään olemaan sellaisenaan eikä selaimen tarvitse piirtää niitä uudestaan. Koska selaimessa näkyvissä oleva navigaatiopalkki on samanlainen kuin minkä React rakensi DOM:iin uudellakin sivulla, selaimen ei tarvitse piirtää tätä elementtiä uudestaan. Navigaatiopalkki jätetään sellaisenaan sivulle.

Selaimessa tapahtuva renderöinnin tehokkuus on riippuvainen siitä, millaisella päätelaitteella se tapahtuu, kun taas palvelimella tapahtuva renderöinti tarjoaa yhtä tehokkaan renderöinnin kaikille päätelaitteille. Matalatehoisella päätelaitteella, esimerkiksi matkapuhelimella, monimutkainen CSR-tyylillä toteutettu sivu saattaa toimia hitaasti. Palvelimen puolella tapahtuva sivujen rakentaminen taas tarjoaa tasaisen suorituskyvyn kaikille päätelaitteille, koska sivujen rakentaminen tapahtuu palvelimella, jossa on usein käytössä paljon laskentatehoa sekä nopeat internetyhteydet. Selainpuolella tapahtuvalla renderöinnillä voi myös olla taloudellisia hyötyjä juurikin sen takia, että sivun rakentaminen tapahtuu käyttäjien laitteilla. Palvelimen kustannukset voivat olla osaksi riippuvaisia sillä suoritettavaan laskentatyöhön. Sivun rakentamisen siirtäminen palvelimelta käyttäjien omille laitteille voi tarjota tavan vähentää sivuston ylläpidon kustannuksia, koska tällöin käytännössä laskentatyötä siirretään pois palvelimelta käyttäjien omille laitteille.

Koska CSR-tyylillä toteutetun sivun pohja on yleensä tyhjä, CSR-sivua on vaikea lukea ja analysoida koneellisesti. Esimerkkejä sivun koneellisesta lukemisesta on esimerkiksi hakukoneiden indeksointi, sekä sivun esikatselun näyttäminen jollakin toisella sivulla, kuten sosiaalisessa mediassa. Hakukoneiden indeksointi on prosessi, jossa hakukone prosessoi sivun sisältöä ja tallentaa sen tietokantaan, jotta sivusto voidaan näyttää hakutuloksissa [7]. Jotkut hakukoneet kykenevät suorittamaan myös JavaScriptiä omilla Javascript-moottoreillaan. Hakukoneet indeksoivat sivuja lukemalla palvelimen palauttamaa HTML-sivua, mutta CSR-tyylisessä sivussa tyhjästä sivun pohjasta ne eivät välttämättä saa paljoa tietoa sivun lopullisesta sisällöstä. Tämä johtuu siitä, että sivun rakentava JavaScript ei välttämättä tule suoritetuksi hakukoneiden indeksoijalla tai silloin, kun jokin muu palvelu näyttää esikatselua sivusta. Työn laatimisen hetkellä tunnetuimmista hakukoneista ainakin Google pystyy indeksoimaan selaimen puolella renderöityjä sivuja,

mutta kaikki hakukoneet eivät siihen pysty. Googlen hakukoneryömijä Googlebot indeksoi ensiksi sivut ainoastaan palvelimen palauttaman HTML-tiedoston pohjalta, jonka jälkeen se lisää sivut renderöintijonoon. Googlebot avaa renderöintijonossa olevat sivut hieman myöhemmin siten, että se myös suorittaa sivun JavaScriptin, jonka jälkeen se indeksoi sivun uudestaan. [8.]

Prosessi, jossa parannellaan sivuston näkyvyyttä ja/tai relevanttiutta hakutuloksissa näkymisessä, on nimeltään hakukoneoptimointi eli SEO (search engine optimization). SEO-ongelmat ovat yksi tunnetuista haasteista SPA-sivuissa. Hakukoneoptimoinnin tapoja ovat esimerkiksi sivuston koneellisen analysoinnin helpottaminen, sisällön määritteleminen ihmisten hakutarpeita vastaavaksi, relevanttien avainsanojen käyttäminen sekä sivuston metatietojen määritteleminen [9]. SPA-sivuilla on haasteita juurikin sivuston koneellisen lukemisen kanssa, koska ne vaativat JavaScriptin suorittamista, jotta sivulle saadaan sisältöä.

2.4 Isomorfinen web-applikaatio

Web-applikaatioiden ja SPA-sivujen yleistyminen on luonnollisesti johtanut siihen, että puhtaaseen selainpuolen renderöintiin liittyviin ongelmiin on kehitetty ratkaisuja. Näitä selainpuolen renderöintiin liittyviä ongelmia ovat tyhjän tai merkityksettömän ensimmäisen sivun näyttäminen, tuon ensimmäisen sivun mahdollisesti hidas latausnopeus sekä huono hakukoneoptimointi. Ratkaisuna edellä mainittuihin ongelmiin on palvelinpuolen renderöinnin yhdistäminen selainpuolen renderöintiin samassa web-applikaatiossa, eli niin kutsuttu isomorfinen web-applikaatio. Isomorfisia web-applikaatioita kutsutaan myös nimellä universaali web-applikaatio. Edellä mainittuihin käsitteisiin liittyy myös isomorfinen JavaScript, jossa web-applikaation koodia suoritetaan sekä palvelimella että selaimessa. [10.]

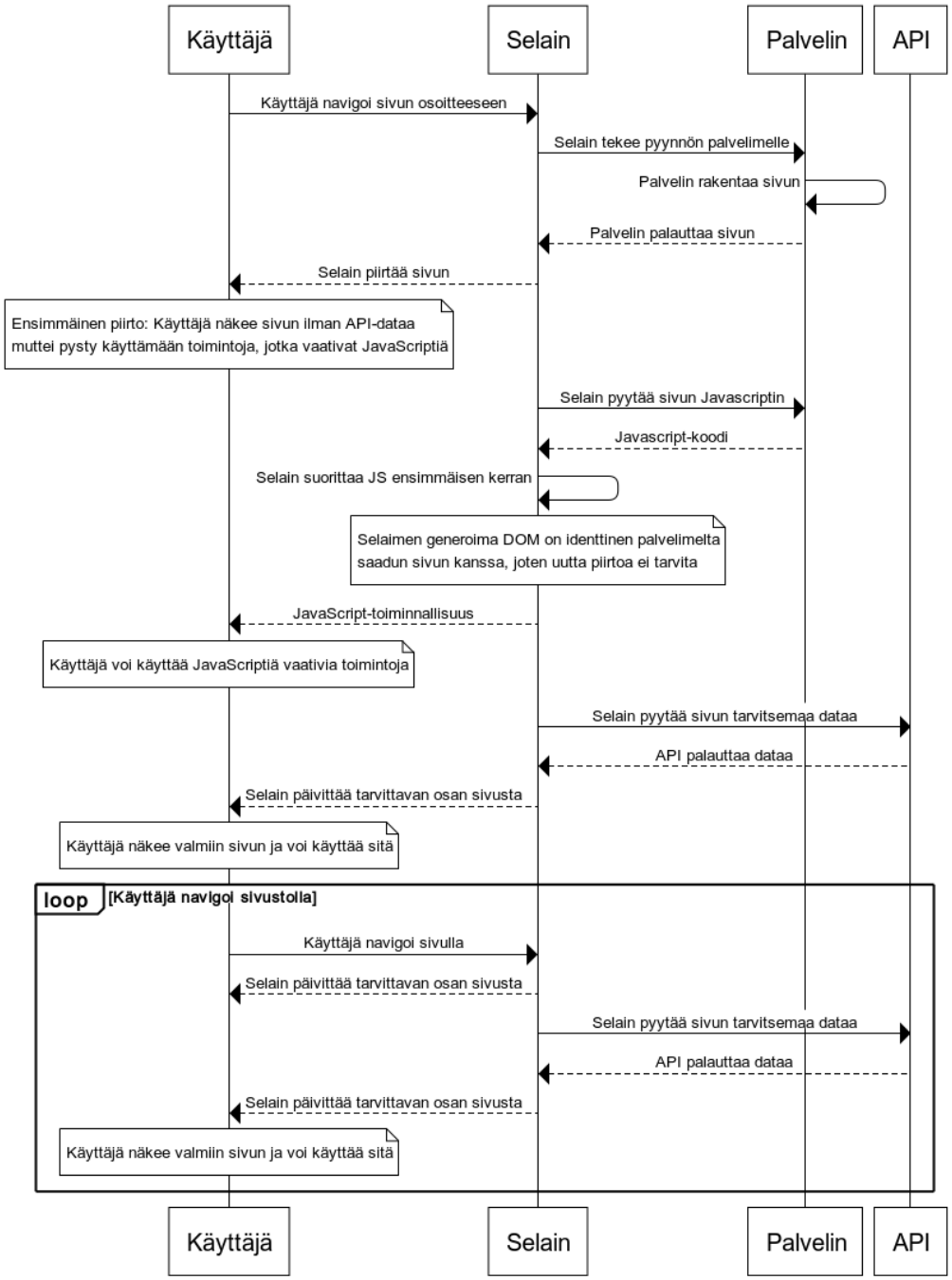
JavaScriptiä voidaan käyttää palvelimien ja API:en ohjelmoimiseen Node.js runtime -ympäristön avulla, jolloin voidaan käyttää samaa ohjelmointikieltä sekä front-endin että back-endin ohjelmoimiseen. JavaScriptin käyttäminen sekä front-endissä että back-endissä ei kuitenkaan vielä tarkoita, että käytettäisiin isomorfista JavaScriptiä, sillä isomorfisessa JavaScript-koodissa selain ja palvelin jakavat samaa koodia. Esimerkkinä ei-isomorfisesta JavaScriptin käytöstä on React-sovelluskehystä käyttävä web-applikaatio,

joka käyttää datan hakemiseen Node.js runtime -ympäristöllä ja Express-sovelluskehyksellä toteutettua rajapintaa. Usein näissä tilanteissa käytetään nimitystä universaali JavaScript, kun JavaScriptiä käytetään front- ja back-endissä, mutta koodi ei ole jaettua niiden kesken [10]. Monesti näitä käsitteitä näkee kuitenkin käytettävän synonyymeinä, ja tässäkin työssä käytetään käsitettä universaali web-applikaatio (UWA eli universal web application). Vastaavasti isomorfisesta JavaScriptistä on kyse esimerkiksi silloin, kun web-applikaatio on toteutettu React-sovelluskehyksellä siten, että myös palvelin suorittaa React-koodia. Tällöin sivun renderöinti tapahtuu osittain palvelimen puolella, ja osittain selaimessa. Isomorfinen JavaScript tarkoittaa SPA-sivujen yhteydessä yleensä sitä, että sivun ensimmäisen latauksen yhteydessä palvelin hoitaa sivun rakentamisen selaimen sijaan. Siitä eteenpäin selain päivittää itse sivua dynaamisesti. Tällä toimintatavalla vältetään kokonaan selainpuolella renderöityjen sivujen hidasta sivun ensimmäistä latauskertaa, mutta päästään silti hyödyntämään selaimessa suoritettavaa dynaamista web-applikaation toimintaa.

Jotta isomorfinen web-applikaatio voi toimia, tulee palvelimen pystyä suorittaa sillä sovelluskehyksellä laadittua koodia, jolla web-applikaatio on toteutettu. Tässä työssä tutkitaan SPA-sivuja Reactin näkökulmasta, ja UWA-sivun toteuttamiseen sekä prosessien tutkimiseen käytetään Next.js-sovelluskehystä. Next on monipuolinen React-ohjelmistokehys, joka mahdollistaa muun muassa palvelinrenderöinnin ja staattisten sivujen generoinnin. Next myös helpottaa kehitystä automatisoimalla osan sivuston kehittämisestä, esimerkiksi tarjoamalla valmiin ratkaisun sivujen väliseen reititykseen. Osin juurikin näiden valmiiden ratkaisuiden takia Next on niin kutsuttu ”opinionated”-sovelluskehys. Toisin sanoen Nextin käyttäminen edellyttää sitä, että tietyt asiat ohjelmassa täytyy toteuttaa juuri sillä tavalla, kuin Nextin kehittäjät ovat tarkoittaneet. [11.]

Kuva 4 esittää sekvenssikaaviota samasta sivun lataus- ja käyttötilanteesta kuin aiemmissakin luvuissa, mutta tällä kertaa sivu on toteutettu UWA:na, ja API-data haetaan asynkronisesti selaimessa. Kuvasta voimme huomata, että prosessi muistuttaa ensimmäisen sivun lataukseen asti palvelinpuolen renderöinnin prosessia, mutta siitä eteenpäin se on samanlainen kuin SPA:ssa. Aluksi käyttäjä avaa jonkin sivuston sivun selaimessa, jolloin selain lähettää pyynnön palvelimelle. Sen sijaan että palvelin palauttaisi aina vain saman sivun pohjan kuten SPA-sivuilla, palvelin aloittaakin React-käyttöliittymäkoodin suorittamisen. Palvelin suorittaa React-koodia samankaltaisesti kuin selainkin

tekisi, ja rakentaa oikean sivun riippuen reitistä, johon pyyntö tehtiin. Kun sivu on rakennettu, palvelin palauttaa sen selaimelle. Tämä on suurin ero normaaleihin SPA-sivuihin; siinä missä SPA saa palvelimelta tyhjän sivun, UWA saakin jo valmiin sivun, jonka se voi näyttää heti käyttäjälle. Tämän jälkeen UWA aloittaa React-koodin suorittamisen selaimessakin. Kun koodi on suoritettu myös selaimessa, sivu on täysin valmis käytettäväksi.



Kuva 4. Sekvenssidiagrammi isomorfisen web-applikaation toiminnasta, kun sivuston tarvitsema data haetaan asynkronisesti selaimen puolella.

Taulukossa 1 vertaillaan yhteenvetona palvelin- ja selainpuolen renderöinnin sekä universaalien web-applikaatioiden ominaisuuksia. Huomioitavaa tässä on se, että kaikki

taulukon ominaisuudet ovat aina ensisijaisesti riippuvaisia sivuston toteutuksesta, ja yhteenvedot ovat vain yleistyksiä. Taulukosta voimme huomata, että UWA näyttäisi toteuttavan kaikki hyvät puolet sekä palvelinrenderöinnistä että SPA:sta.

Taulukko 1. Vertailuja SSR-, CSR- ja UWA-toimintatapojen ominaisuuksista.

Ominaisuus	**SSR**	**CSR (SPA)**	**UWA**
Sivuston tyyppi, johon parhaiten soveltuu	Staattiset sivustot	Web-applikaatiot	Web-applikaatiot + staattiset sivut
Sivun rakentaminen	Palvelimella	Selaimessa	Palvelimella ja selaimessa
Sivun ensimmäinen lataus	Normaali	Riippuu laitteesta ja sivusta	Normaali
Nopeus ensimmäisen latauksen jälkeen	Normaali	Nopea	Nopea
Nopeus matalatehoisella laitteella	Normaali	Riippuu laitteesta ja sivusta	Riippuu laitteesta ja sivusta
SEO	Täysi tuki	Osittainen tuki	Täysi tuki
Sivun esikatselun näyttäminen muissa palveluissa	Täysi tuki	Ei yleensä	Täysi tuki

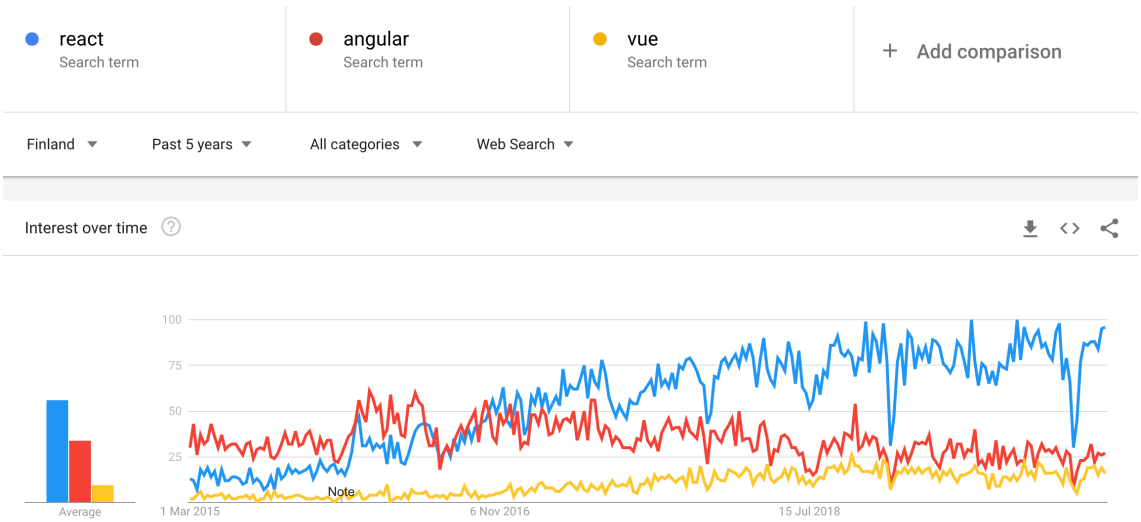
3 Teknologioiden kartoitus

Työn tarkoituksena on arvioida SPA-sivujen mahdollisia heikkouksia ja niiden vakavuutta, ja sitten tutkia, löytyisikö niihin ratkaisuja isomorfisen JavaScriptin avulla. Tarkoitus on myös tutkia, tuoko isomorfinen JavaScript ja UWA:t mukanaan joitain muita ongelmia, jotka tekisivät niistä vähemmän ilmiselvän valinnan SPA:n rinnalla. Jotta tutkimus voidaan toteuttaa, tässä työssä suunnitellaan ja toteutetaan kaksi esimerkkiapplikaatiota – toinen SPA- ja toinen UWA-sivuna. Perinteisempää palvelinpuolen renderöinnillä toteutettua monisivuista sivustoa tässä työssä ei toteuteta, koska työn fokus on nimenomaan selainpuolen web-applikaatioissa. Tässä luvussa käsitellään työhön valittuja teknologioita.

3.1 React

Suosituimpia JavaScript-käyttöliittymäsovelluskehyksiä ovat React, Angular ja Vue [12]. React vaikuttaisi olevan näistä kolmesta suosituin ainakin Suomessa. Reactin suosio käy ilmi sekä vertailemalla sellaisten avointen työpaikkojen määrää Suomessa, joihin

liittyy joko React, Angular tai Vue, että vertailemalla Google-hakujen määrää eri kehysten välillä. Kuva 5 näyttää vertailun Google-hakutulosten määrän perusteella Suomessa näiden sovelluskehysten välillä ajalta 1.3.2015 – 1.3.2020. Tässä työssä selainpuolen renderöinnin tutkimiseen ja demo-ohjelman toteuttamiseen valikoitui React sen suosion takia sekä sen takia, että olen käyttänyt sitä aiemminkin.



Kuva 5. Eri JavaScript-käyttöliittymäkehysten Google-hakutulosten kehitys Suomessa [13].

React on Facebookin kehittämä avoimen lähdekoodin sovelluskehys. React on niin kutsuttu ”unopinionated”-sovelluskehys, eli se ei aseta paljoa vaatimuksia sille, kuinka sillä tehty ohjelma pitäisi rakentaa tai mitä muita teknologioita sen kanssa pitäisi käyttää. Tämä tekee Reactista hyvin monikäyttöisen ja joustavan. Reactissa käyttöliittymää ajatellaan kokoelmana komponentteja, jotka esittävät jotakin tiettyä kokonaisuutta käyttöliittymässä ja voivat myös määritellä sen toimintaa. Esimerkkinä komponentista olisi esimerkiksi painike, jonka ulkoasu määritellään kerran yhdessä paikassa, minkä jälkeen sitä voi käyttää monessa eri paikassa käyttöliittymää. Komponentteja voi luoda kahdella eri tavalla: luokkina ja funktioina. React tukee ja suosittelee sen kanssa käytettäväksi JSX-syntaksia, joka mahdollistaa markup-tyylisten elementtien käytön suoraan JavaScriptin seassa. Nämä JSX-elementit muunnetaan käännöksen yhteydessä normaaleiksi JavaScript-funktioiksi, jotka palauttavat komponentteja JavaScript-olioina [14]. Esimerkkikoodissa 1 esitellään funktionaalinen React-komponentti AboutView, joka on toteutettu JSX-lausekkeilla. Huomataan, kuinka siinä käytetään markup-tyylisiä JSX-tageja JavaScriptin seassa.

```
***cra/src/components/About.js
const AboutView = () => {
  simulateProcessing();
  return (
    <div>
      <p>
        This is a demo single page application made using create-react-app.
      </p>
      <p>
        This app is made heavier than needed by purpose, to simulate a
        larger and more complex app.
      </p>
      <Image src={banner} alt="movie night" width="40%"/>
      <FillerData />
    </div>
  );
};
```

Esimerkkikoodi 1. Ote demoapplikaation React-koodista, jossa esitellään AboutView-komponentti.

JSX muistuttaa syntaksiltaan hyvin paljon HTML-syntaksia, mutta JSX ja HTML ei kuitenkaan ole sama asia. Osalla JSX-elementeistä on sama tagi kuin vastaavalla HTML-elementillä. JSX-elementit esittävät ohjeita Reactille, että mitä DOM-elementtejä mihinkin kohtaan on tarkoitus renderöidä. Kuten edellä mainittiin, nämä muutetaan käännöksen yhteydessä tavallisiksi JavaScript-funktioiksi, jotka luovat tarvittavat HTML-elementit. Lisäksi JSX:n ja HTML:n erottaa toisistaan se, että JSX-elementit lisätään JavaScriptin sekaan sellaisenaan eikä merkkijonoina. Sen sijaan HTML käsitellään yleensä koodissa merkkijonoina.

Uusien React-ohjelmien luomiseen käytetään usein Create React App -nimistä työkalua. Create React App eli CRA luo ja konfiguroi valmiin pohjan uudelle React-ohjelmalle, tehden kehityksen aloittamisesta nopeaa ja helppoa. [15]. Tässä työssä tullaan luomaan SPA-demoapplikaatio CRA-työkalua käyttäen.

3.2 Next.js

Universaalin web-applikaation toteutusta varten tähän työhön valittiin Next.js-sovelluskehys, joka rakentuu Reactin päälle [11]. Se on melko suosittu työkalu, jonka päällimmäisin ominaisuus on Reactin renderöinti palvelimen puolella. Next tarjoaa palvelinpuolen renderöinnin lisäksi paljon muutakin toiminnallisuutta. Näistä esimerkkejä ovat staattisten sivujen generointi, automaattinen staattinen optimointi, serverless functions, sivujen (dynaaminen) reititys ja progressive web application (PWA) -tuki. Automaattinen

staattinen optimointi tunnistaa koodista sellaiset sivut, joista voidaan jo ohjelman koontivaiheessa luoda valmiit HTML-sivut. Tällöin sivuja ei tarvitse renderöidä palvelimen puolella, vaan ne voidaan tarjoilla suoraan. Serverless on käsite, jossa palveluntarjoaja hallitsee palvelinta ja sen resursseja, kun taas asiakas määrittelee itse palvelimen operaatiot. Sivujen dynaaminen reititys tarkoittaa sitä, että Next osaa reitittää sivut, joiden reittiä ei tiedetä etukäteen. Esimerkkinä tästä on esimerkiksi demoapplikaatioiden reitti /shows/123, jossa 123 on jonkin TV-ohjelman tunniste. Progressive web application tarkoittaa sivustoa, joka toimii kuten natiivisovellus mobiililaitteilla. PWA:t tukevat offline-käyttöä ja ne voi lisätä mobiililaitteiden ohjelmavalikkoon, aivan kuten natiivisovelluksenkin. Näiden monipuolisten ominaisuuksien kääntöpuolena voisi nähdä sen, että Next jossain määrin pakottaa tekemään asiat juuri sillä tavalla kuin sen kehittäjät ovat visioineet. Eli Next ei ole niin yleiskäyttöinen ja taipuva kaikkeen kuin pelkkä React.

Muita vaihtoehtoja Nextille ovat esimerkiksi Gatsby ja Reactin sisäänrakennettu tuki palvelinpuolen renderöinnille. Gatsby erikoistuu staattisten sivujen generointiin, joka on myös yksi Nextin toiminnollisuuksista. Lisäksi React tukee jo valmiiksi jossain määrin palvelinrenderöintiä, jolloin komponenttihierarkia renderöidään tekstiksi palvelimella. Reactin sisäänrakennettu tuki palvelinrenderöinnille on siinä mielessä kevyt ratkaisu, että se tarjoaa minimaaliset työkalut palvelinrenderöinnin toteuttamiseen. Jos sitä haluaa käyttää, täytyy itse ohjelmoida joitakin sellaisia toiminnollisuuksia, jotka tulisivat valmiiden ratkaisujen mukana automaattisesti. Esimerkiksi oma Node-palvelin täytyy pystyttää. Hyvä puoli Reactin sisäänrakennetussa SSR-tuessa on kuitenkin se, että se on joustava juurikin sen minimaalisuuden takia. Tässä työssä päädyttiin Nextiin sen ilmeisen helppouden ja suosion takia.

3.3 Zeit Now

CRA- ja Next-projektit voidaan käynnistää niin kutsutussa kehitystilassa, joka tekee projektin kehityksestä helpompaa. Kehitystilassa muutokset lähdekoodissa päivittyvät paikallisesti käynnistetyssä applikaatiossa ilman, että applikaatiota täytyy pysäyttää ja käynnistää uudestaan. Kehitystilassa voi käyttää myös muita kehitystä helpottavia työkaluja. Tätä toimintoa tulee kuitenkin käyttää vain kehityksen aikana, koska se tekee ohjelman suorituksesta raskaampaa. Vastaavasti oikeissa tilanteissa applikaatio täytyy ensiksi kääntää (build), ja sitten käynnistää käännetty ohjelma. Ohjelman kääntäminen tuottaa

ohjelmasta minimoidun ja optimoidun paketin, joka toimii tehokkaammin kuin kehitysversio. Minimointi (minify) tarkoittaa toimenpidettä, jossa JavaScript-koodi on muunnettu mahdollisimman kompaktiin pakettiin sen hinnalla, että se ei enää ole ihmiselle helposti luettavaa. Koodin luettavuudella ei kuitenkaan ole enää väliä siinä tilanteessa, kun ohjelma on käynnissä oikeassa ympäristössä. Jotta demoapplikaatioiden suorituskykyä voidaan arvioida realistisesti, täytyy ohjelmat kääntää ja viedä oikeaan ympäristöön. Oikea ympäristö tarkoittaa siis oikeaa palvelinta eikä esimerkiksi kehittäjän omaa tietokonetta. Tällöin voidaan arvioida sekä applikaation suorituskykyä että web-kutsujen vaikutusta toimintaan.

Sekä Create React App että Next käyttävät oletuksena NPM-työkalua riippuvuuksien ja kääntämisskriptien hallintaan, joten prosessi niillä toteutettujen ohjelmien kääntämiseen ja viemiseen ympäristöihin on samankaltainen. Koska prosessi on helppo kummallakin työkalulla, DevOps ja ympäristöihin liittyvät asiat jäävät tässä työssä vähälle käsittelylle. Oikeaan ympäristöön vientiä varten tähän työhön haluttiin jokin työkalu, jolla se onnistuisi mahdollisimman vähällä vaivalla ja kätevästi. Nextin takana oleva yritys Zeit on luonut työkalun nimeltä Now, joka vaikutti juuri sopivalta työkalulta tähän käyttöön. Now-työkalulla projektin ympäristöön vienti on vain yhden komennon takana, ja Now hoitaa automaattisesti kaiken muun domainien konfiguroinnista lähtien. Now tukee luonnollisesti Next-ohjelmien vientiä, mutta se tukee suoraan myös Create React App -työkalulla toteutettuja ohjelmia. [16.] Siispä tähän työhön valittiin Now, jotta demoapplikaatiot saadaan toimintaan oikeissa ympäristöissä nopeasti. Samankaltaisia työkaluja on myös muita. Esimerkiksi Heroku on toinen suosittu vaihtoehto.

4 Demosivujen toteutus

Työssä haluttiin tutkia SPA- ja UWA-kehitystä mahdollisimman realistisissa puitteissa, mikä täytyi ottaa huomioon demosivuja suunnitellessa. Demosivujen täytyisi olla laajempia kuin yksinkertaiset ”Hello, World!” -esimerkit, jotta ne kuvastaisivat oikean elämän applikaatioita. Tässä luvussa käydään läpi demosivujen suunnittelua ja käytännön toteutusta.

4.1 Suunnittelu

Oikeassa käytössä olevissa web-sivuissa on yleensä tiettyjä ominaisuuksia. Näitä ovat esimerkiksi

- API-kutsut
- kuvat
- useat sivut
- lukuisat DOM-elementit
- käyttäjien autentikointi
- CSS-tyylittelyt.

Edellä mainitut asiat joko lisäävät web-applikaation JavaScript-koodin kokoa, luovat viiveitä, tai tekevät sivun piirtämisestä raskaampaa DOM-elementtien ja CSS-tyylien rakenteen ja määrän takia. Jos demosivu olisi vain hyvin pelkistetty ja kevyt sivu ilman API-kutsuja, isoa rakenteisuutta ja tyylejä, se ei kuvaisi oikean elämän tilannetta. Tällöin todennäköisesti ei huomattaisi juurikaan eroa suorituskyvyssä, oli web-sivu sitten toteutettu SPA:na tai UWA:na, koska se varmaankin toimisi hyvin nopeasti joka tapauksessa. Edellä mainituista oikeisiin sivuihin liittyvistä piirteistä käyttäjien autentikointi on ominaisuus, joka ei vaikuta olevan niin tärkeä niiden kysymysten kannalta, johon tässä työssä etsitään vastausta. Käyttäjien autentikointi lisäisi kyllä sivun kokoa, mutta hyöty siitä olisi pientä verrattuna vaivannäköön, joka autentikoinnin implementoimiseen kuluisi. Käyttäjien autentikoinita lukuun ottamatta muut edellä mainitut piirteet on hyvä ottaa mukaan demosivuun, jotta se vastaisi oikeaa applikaatiota.

Demosivuksi päätettiin tuottaa sivu, jolla pystyy selailemaan ja hakemaan tietoa TV-ohjelmista. TV-ohjelmien tiedot noudetaan TVmaze:n API:sta asynkronisilla kutsuilla selaimessa. TVmaze on sivusto TV-ohjelmatietojen selailemiseen, ja se tarjoaa myös ilmaisen rajapinnan niille. Demosivu sisältää etusivun, jolla näytetään satunnaisia TV-oh-

jelmia lyhyen kuvauksen ja kuvan kanssa. Sivu sisältää navigaatiopalkin sivun ylälaidassa, jossa on hakukenttä. Hakukentän avulla voi hakea TV-ohjelmia, jolloin vain hakukriteerit täyttävät TV-ohjelmat näytetään. Kun jotakin TV-ohjelmaa klikkaa hakutuloksissa tai etusivulla, avataan tietosivu kyseiselle TV-ohjelmalle. Tällä sivulla näytetään tietoja ainoastaan tälle ohjelmalle. Lisäksi sivustolla on tietoja-sivu, jossa kerrotaan tietoja tästä sivustosta. Tälle sivulle pääsee klikkaamalla linkkiä navigaatiopalkissa. Näiden toiminnollisuuksien kanssa sivusto on vielä melko kevyt, joten projektiin asennetaan lisäksi ylimääräisiä kirjastoja, jotka otetaan käyttöön applikaatiossa niitä kuitenkaan käyttämättä. Tällöin sivuston JavaScript-koodin kokoa saadaan kasvatettua helposti. Sivuston tietoja-sivulle on myös generoitu paljon DOM-elementtejä simuloimaan monimutkaisempaa sivua. Edellä mainittujen sivua raskaammaksi tekevien keinojen lisäksi tietoja-sivulle toteutetaan vielä ylimääräistä kuormittavaa laskentatyötä, joka suoritetaan silloin, kun sivulle navigoidaan tai sivu päivitetään. Tällöin saavutetaan tilanne, jossa tietoja-sivu simuloi isoa määrää JavaScript-koodia, lukuisia DOM-elementtejä, sekä raskasta laskentatyötä. Tätä raskasta laskentatyötä ei lisätä applikaation muille sivuille, jotta voidaan arvioida sivuston tehokkuutta myös ilman pakotettua hidastusta. Demosivun tyylittelyt toteutetaan React Bootstrap -kirjastolla, joka tarjoaa suositun Bootstrap-kirjaston tyylittelyjä helppokäyttöisinä React-komponentteina. Tällöin saadaan oikeaa tilannetta vastaavan kokoiset ja raskaat tyylit mukaan projektiin, ja samalla käyttöliittymän kehittäminen saadaan nopeaksi, kun tyylittelyjen yksityiskohtia ei tarvitse itse kehittää.

4.2 Single page application -toteutus

Yhden sivun applikaatio toteutettiin siis Create React App -työkalua käyttäen, mikä on suosittu tapa luoda uusia React-applikaatioita. Reactia pyrittiin käyttämään hyvien toimintatapojen mukaisesti, eli esimerkiksi suosimalla yleiskäyttöisiä ja modulaarisia komponentteja, sekä erottelemalla eri ohjelmaosien vastuut selkeästi. Esimerkkikoodissa 2 on esitelty CRA:lla toteutetun SPA-applikaation tiedosto- ja hakemistorakenne.

```
cra
├── package-lock.json
├── package.json
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
```

```
│   ├── manifest.json
│   └── robots.txt
└── src
    ├── App.js
    ├── assets
    │   ├── banner.png
    │   └── movie.png
    ├── components
    │   ├── About.js
    │   ├── Banner.js
    │   ├── ErrorView.js
    │   ├── FIllerData.js
    │   ├── Header.js
    │   ├── MainLayout.js
    │   ├── Search.js
    │   ├── ShowDetails.js
    │   ├── ShowList.js
    │   ├── ShowListItem.js
    │   └── SimpleAlert.js
    ├── hooks
    │   └── useQuery.js
    ├── index.js
    ├── reducers
    │   ├── showDetailsReducer.js
    │   └── showsReducer.js
    ├── serviceWorker.js
    ├── services
    │   └── shows.js
    ├── store.js
    └── utils.js

7 directories, 30 files
```

Esimerkkikoodi 2. SPA-demoapplikaation hakemisto- ja tiedostorakenne.

Ohjelman aloituspisteenä toimii src/index.js, jossa ohjelman pääkomponentti renderöidään HTML-sivun pohjan root-elementtiin. Sivun pohja sijaitsee polussa public/index.html. Ohjelman pääkomponentti on esitelty tiedostossa src/App.js. App-komponentti sisältää ohjelman pääreitityslogiikan, tilanhallinnan alustuksen sekä kaikille sivuille yhteiset käyttöliittymäelementit. Yksittäiset komponentit sijaitsevat src/components-hakemistossa, ja operaatiot tilan muokkaamiseen sijaitsevat src/reducers-hakemistossa. Applikaation tarvitsemat API-kutsut on kapseloitu src/services/shows.js-tiedostoon, joka tarjoaa asynkroniset funktiot datan hakemiseen ilman, että muun applikaation tarvitsee tietää mitään siitä, mistä tai miten data haetaan.

Ohjelman tilanhallintaan käytettiin Redux- ja React Redux -kirjastoja. Redux on suosittu JavaScript-kirjasto applikaatioiden tilanhallintaan, varsinkin monimutkaisissa applikaati-

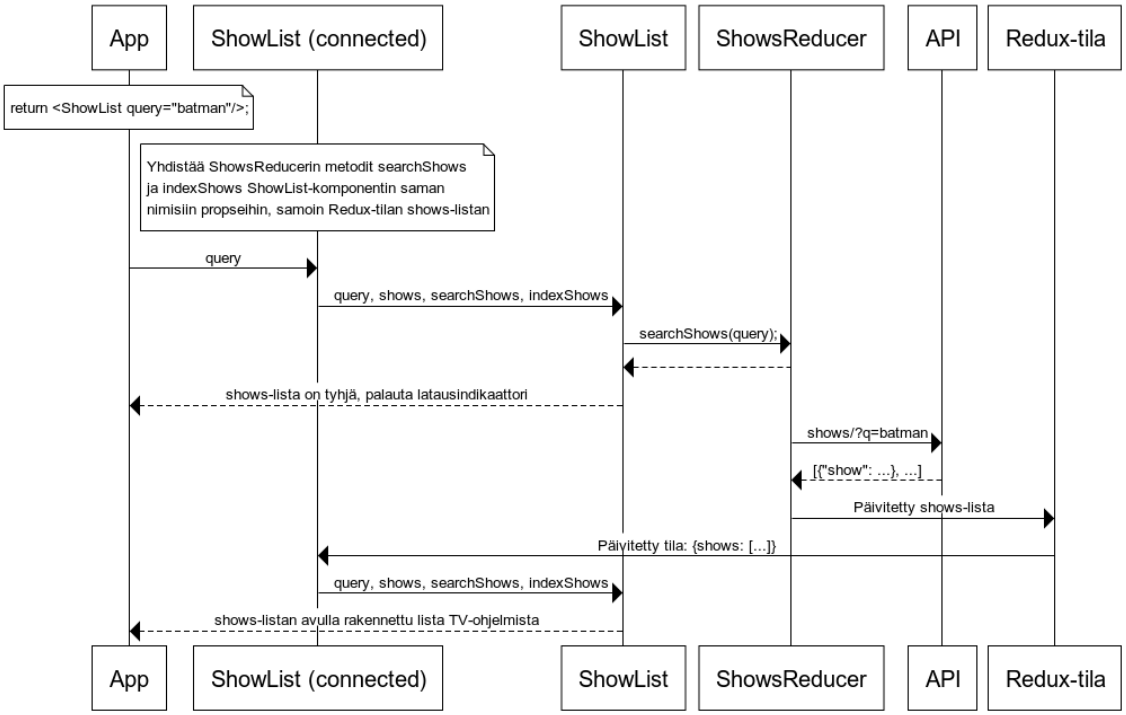
oissa [17]. Vaikka tässä työssä kehitetty applikaatio ei yksinkertaisuutensa takia sitä niinkään tarvitsisi, kirjasto haluttiin ottaa silti käyttöön sen takia, että se on suosittu. Tällöin voidaan tutkia, ilmeneekö Reduxin käytössä haasteita, kun siirrymme kehittämään palvelinpuolella renderöityä Reactia isomorfista applikaatiota työstettäessä. Lisäksi Redux tarjoaa hyvän tavan erottaa tilanhallinta kokonaan React-komponenteista, jolloin ne ovat muun muassa helposti testattavia. Tilanhallinnan erottaminen komponenteista on myös siinä mielessä järkevää, että tällöin komponenttien ei tarvitse välittää mitään siitä, mistä niiden tarvitsema data tulee ja minkälaisilla operaatioilla sitä voidaan muokata. Toki tämä menettelytapa on yleinen hyvä käytäntö React-maailmassa myös ilman Reduxia, ja sitä kutsutaan nimellä ”Lifting state up” [18]. Tässä menettelytavassa Reactin sisäänrakennettu tilanhallinta pyritään nostamaan komponenttihierarkiassa mahdollisimman ylös yleiselle tasolle, jottei yksittäisten komponenttien tarvitse hallita sitä ja jotta tätä tilaa voidaan jakaa useille eri komponenteille. Tilanhallinnan erottamista komponenteista Reduxilla havainnollistaa hyvin esimerkkikoodi 3, jossa on ote ShowList-komponentin esittelystä.

```
***cra/src/components/ShowList.js
import React, {useEffect} from 'react';
import {connect} from 'react-redux';
import {searchShows, indexShows} from '../reducers/showsReducer';

const showsAsList = (shows) => { /* Palauttaa JSX listan TV-ohjelmista */ };

const ShowList = ({query, shows, searchShows, indexShows}) => {
  /* Ei näytetty: virheilmoitusten ja latausindikaattorin hallinta */
  const isError = shows ? shows.isError : false;
  useEffect(() => {
    if (query) searchShows(query);
    else indexShows();
  }, [query, searchShows, indexShows]);

  return (
    <div>
      {!isError && shows && <div>{showsAsList(shows)}</div>}
    </div>
  );
};

const mapStateToProps = (state) => ({shows: state.shows});
const mapDispatchToProps = {searchShows, indexShows};
export default connect(mapStateToProps, mapDispatchToProps)(ShowList);
```

Esimerkkikoodi 3. Riisuttu ote ShowList.js-tiedostosta, jossa esitellään React Redux -kirjaston käyttöä ShowList-komponentin kanssa.

React-komponentit tukevat datan syöttämistä niille props-mekanismilla, joka muistuttaa parametrien antamista funktioille. Esimerkkikoodissa 3 voidaan nähdä, että ShowList-

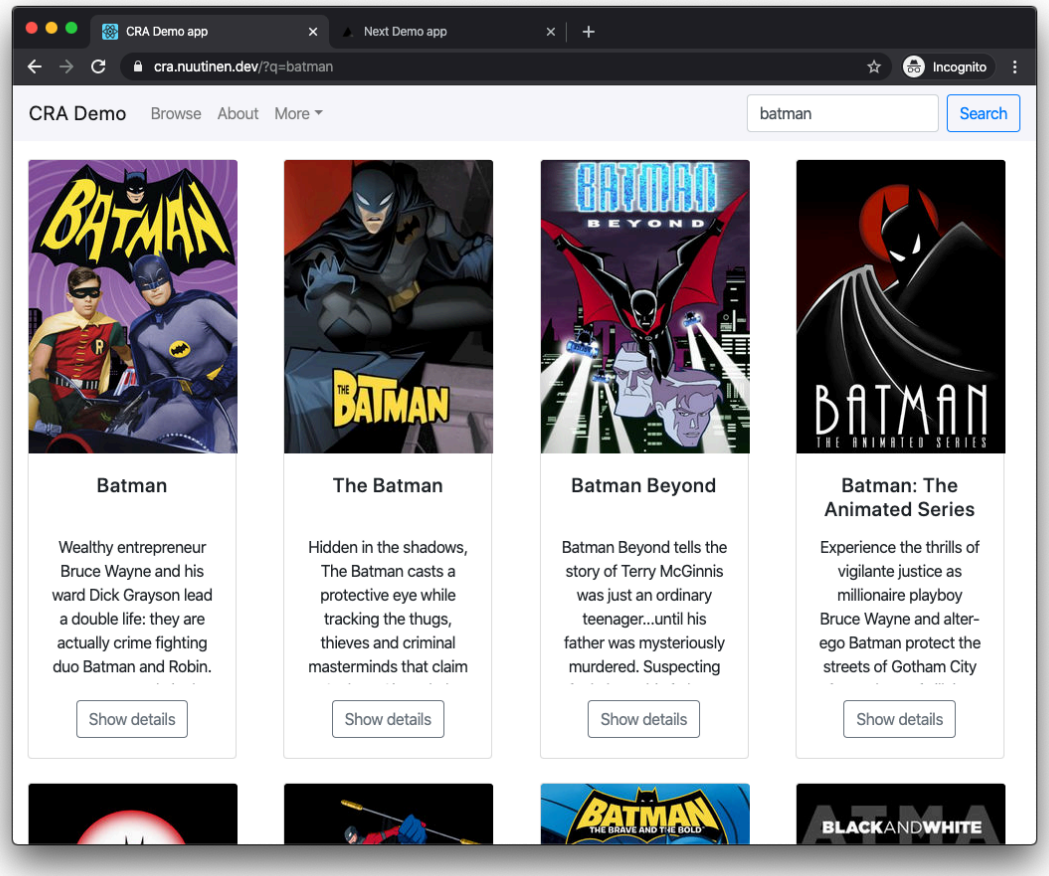
komponentti odottaa, että sille annetaan query-, shows-, searchShows-, sekä indexShows-nimiset propsit. query esittää mahdollista TV-ohjelmien hakutekstiä, shows esittää listaa näytettävistä TV-ohjelmista, searchShows on metodi TV-ohjelmien hakuun, ja indexShows on metodi, jolla voidaan hakea satunnaisia TV-ohjelmia, jos hakua ei ole tehty. ShowList komponentti ei siis välitä, mistä nämä propsit tulevat tai miten esimerkiksi searchShows-prop on toteutettu. Tästä näemmekin, että komponentti ei pidä sisällään mitään tilanhallinnan yksityiskohtia eikä myöskään mitään Redux-spesifistä koodia, mikä on hyvä asia testattavuuden kannalta. Sen sijaan Redux-spesifistä koodia voimme nähdä otteen viimeisillä kolmella rivillä, joissa määritellään, mitä propseja Reduxin halutaan syöttävän ShowList-komponentille aina, kun joku kutsuva koodi käyttää tätä komponenttia. Viimeisellä rivillä kutsutaan React Redux -kirjaston connect-metodia, jolla syötetään automaattisesti komponentille propseina sen tarvitsema tila sekä tilaoperaatiot. Kutsuva koodi voi käyttää ShowList-komponenttia siten, että sen tarvitsee enää antaa query-prop, koska muut propsit annetaan komponentille connect-metodin avulla. Projektissa käytettiin myös redux-thunk -kirjastoa, joka mahdollistaa esimerkiksi asynkronisten kutsujen tekemisen Reduxin tilaa muokkaavissa operaatioissa. Esimerkkikoodissa 3 searchShows ja indexShows ovat funktioita, jotka käyttävät tätä toiminnallisuutta hyväkseen. Ne suorittavat asynkronisia web-kutsuja, jonka jälkeen ne päivittävät Reduxin tilaa tarpeen mukaan. Edellä mainittua tilannetta havainnollistaa myös kuvan 6 sekvenssikaavio.



Kuva 6. Sekvenssikaavio Redux-kirjaston käyttötavasta ShowList-komponentin kanssa.

Kuvasta 6 nähdään, että ShowList-komponentti ei suoraan tunne tilaa hallitsevaa ShowsReduceria. Sen sijaan ShowList-komponentti vain käyttää niitä metodeja, jotka sille on annettu propseina. Tässä tapauksessa propseina annettavat tilanhallinnan metodit ja listan TV-ohjelmista annetaan React Redux -kirjaston connect-metodin avulla. ShowList.js-tiedosto esittelee muulle ohjelmalle ainoastaan connect-metodilla yhdistetyn version ShowList-komponentista. Tällöin kutsuva koodi kutsuu aina connect-metodilla yhdistettyä versiota ShowList-komponentista, jolloin kutsuvan koodin tarvitsee antaa propseina ainoastaan query eli hakutermi. Kuvasta 6 nähdään myös, kuinka TV-ohjelmien tiedot haetaan asynkronisesti. Tilan päivittyminen uusilla TV-ohjelmien tiedoilla laukaisee Reactissa uudelleenrenderöinnin, jolloin ShowList-komponentti renderöidään uudelleen uusilla tilatiedoilla.

Kaiken kaikkiaan projektin toteutus Create React App -työkalua käyttäen sujui ongelmitta, koska se on tuttu ja joustava työkalu. Tuloksena saatiin suunnitellun mukainen TV-ohjelmien hakusivusto, josta on ruutukaappaus kuvassa 7.



Kuva 7. Kuvakaappaus Create React App -työkalulla luodusta SPA-sivusta.

Sivuston käyttöliittymä oli helppo ja nopea luoda React Bootstrap -kirjastoa käyttäen, joten sivuston ulkoasun työstämiseen ei kulunut paljoa aikaa. SPA-sivu vietiin Zeit Now -työkalulla oikeaan ympäristöön, jotta sitä päästiin käyttämään miltä laitteelta tahansa. Tuotettu sivu on nopea, ja liikkuminen eri sivujen välillä on jouhevaa yhden sivun applikaation ansiosta. TV-ohjelmien tiedot haetaan asynkronisesti selaimessa, ja niiden haun ajan sivustolla näkyy latausanimaatio. Sivustoon ei vielä lisätty ylimääräisiä riippuvuuksia tai tarkoituksenmukaisia hidasteita simuloimaan isompaa applikaatiota, vaan seuraavaksi siirryttiin kehittämään vastaavaa sivua universaalina web-applikaationa. Tässä vaiheessa SPA-sivu näyttäisi toimivan ripeästi. Suorituskykyongelmia ei tunnu olevan edes sivun ensimmäisellä latauskerralla, joten jää nähtäväksi pystyykö UWA päihittämään SPA-sivun suorituskyvyssä. Sivun suorituskykyä ja SEO:ta käsitellään tarkemmin myöhemmin, kun kumpaakin tuotettua sivustoa auditoidaan Googlen Lighthouse-työkalulla.

4.3 Isomorfisen web-applikaation toteutus

Isomorfisesta web-applikaatiosta pyrittiin tekemään ulkoisesti ja toiminnaltaan mahdollisimman samankaltainen aiemmin toteutetun SPA-sivun kanssa. Siten päästään ensinnäkin arvioimaan sivujen suorituskykyä vertailukelpoisesti, mutta samalla päästään arvioimaan, kuinka hyvin olemassa olevan SPA-sivun voi muuttaa Nextillä toteutetuksi. Siispä lähtökohdaksi otettiin CRA:lla tuotettu SPA-sivu, ja sitä lähdettiin muokkaamaan Nextille sopivaksi. Nextiä käyttäessä tulee jo alkuun ilmi yksi projektin rakenteeseen liittyvä seikka, joka pitää huomioida: Nextissä eri sivuille täytyy tehdä omat tiedostonsa pages-hakemiston alle. CRA:ssa mitään tällaista rajoitusta ei ollut, ja itseasiassa CRA:lla tehdyssä sivussa ei pakosti edes ole mitään tarkkaa rajausta eri sivujen välillä, koska se on yhden sivun applikaatio. Pages-hakemiston alla täytyy olla index.js-niminen tiedosto, joka esittää Next-applikaation pääsivua. Nextillä toteutetun demoapplikaation lopullinen hakemisto- ja tiedostorakenne on kuvattu esimerkkikoodissa 4.

```
next
├── package-lock.json
├── package.json
├── public
│   ├── favicon.ico
│   └── static
│       ├── banner.png
│       └── movie.png
└── src
    ├── components
    │   ├── Banner.js
    │   ├── ErrorView.js
    │   ├── FIllerData.js
    │   ├── Header.js
    │   ├── MainLayout.js
    │   ├── Search.js
    │   ├── ShowListItem.js
    │   └── SimpleAlert.js
    ├── pages
    │   ├── _app.js
    │   ├── about.js
    │   ├── index.js
    │   └── shows
    │       └── [id].js
    ├── reducers
    │   ├── showDetailsReducer.js
    │   └── showsReducer.js
    ├── services
    │   └── shows.js
    ├── static
    │   ├── banner.png
    │   └── movie.png
```

```
    ├── store.js
    └── utils.js

9 directories, 24 files
```

Esimerkkikoodi 4. Next-demoapplikaation hakemisto- ja tiedostorakenne.

Esimerkkikoodista 4 voidaan huomata, että paljon alkuperäisestä SPA-projektin rakenteesta on vielä tallella. Monet komponentit, store.js- ja utils.js-tiedostot sekä services- ja reducers-hakemistot pystyttiin kopioimaan Next-projektiin ilman mitään muutoksia. Joissakin komponenteissa tarvittiin vain pieniä muutoksia. React Router -kirjaston linkkikomponenttien vaihtaminen Nextin vastaaviin oli tällainen tilanne. Suurimmat muutokset täytyi tehdä sivujen järjestelyyn ja reittien hallintaan. Joissakin tapauksissa myös Reduxin käyttäminen vaati hieman pohdiskelua.

Sivujen jaottelu omiin tiedostoihinsa pages-hakemiston alle ei ole ainoastaan projektin jäsentelyyn liittyvä käytäntö Nextissä. Next reitittää pages-hakemiston sisältämät tiedostot automaattisesti sivuston osoitteiksi. Esimerkiksi src/pages/about.js -tiedostossa esitelty React-komponentti tulee olemaan automaattisesti saatavilla osoitteessa /about. Next tukee myös dynaamista reititystä, josta esimerkkinä on yksittäisten TV-ohjelmien lisätietoja-sivu. Tämän sivun tiedosto löytyy polusta src/pages/shows/[id].js, ja Next reitittää sen automaattisesti osoitteeseen /shows/123, jossa 123 on jonkin TV-ohjelman tunniste. Nextillä eri sivujen reittejä ei siis tarvitse määritellä koodissa, vaan reitit generoidaan automaattisesti. CRA-applikaatiossa sivun reitit piti itse määritellä React Router -kirjaston avulla. Nextin tapa on kätevä, mutta se on ehkäpä vähemmän joustava kuin täysin itse määritellyt reitit React Routerilla.

Datan hakemisen komponenttien käyttöön voi hoitaa usealla eri tavalla Nextissä. Datan voi hakea asynkronisesti selaimen puolella aivan kuten tavallisesti Reactilla, mutta dataa voi hakea myös palvelimen puolella. Datan hakeminen palvelimen puolella on järkevää silloin, jos data on tärkeää SEO:n kannalta. Niin kuin kuvassa 1 näytettiin, datan hakeminen sivuun palvelimen puolella kuitenkin johtaa siihen, että selain joutuu odottamaan sivua palvelimelta pidempään. Joissakin tilanteissa sivun sisältö on kuitenkin niin tärkeää hakukoneoptimoinnin kannalta, että tämä vaihtokauppa hitaamman latauksen kanssa on järkevää tehdä. Toinen tapa hakea dataa Nextin sivujen käyttöön on sama kuin normaalistikin Reactilla, eli sen tekeminen selaimen puolella.

Aluksi projektissa kokeiltiin datan hakemista palvelimella, ja tulokset olivat odotetut. TV-ohjelmia näyttävän sivun TTFB kärsi huomattavasti. Kun palvelimen palauttamaa HTML-sivua tarkasteli, huomattiin, että se sisälsi jo tiedot kaikista TV-ohjelmista, ja ainoastaan kuvat haettiin selaimen puolella. Tämä ominaisuus voisi olla erittäin tärkeä esimerkiksi nettikaupoissa, joissa on tärkeää, että hakukoneiden hakutuloksissa voidaan näyttää tiettyjä tuotteita. Jos nettikaupan valikoima haettaisiin vasta selaimen puolella, eivät hakukoneet pystyisi indeksoimaan sitä yhtä tehokkaasti. Tässä demoprojektissa siirryttiin kuitenkin lopulta käyttämään täysin samaa tapaa noutaa TV-ohjelmien tiedot kuin SPA-sivulla, eli asynkronisilla kutsuilla selaimen puolella. Se vähensi myös koodiin tarvittavien muutosten määrää, kun siirrettiin projektia CRA:sta Nextiin. Vaikka TV-ohjelmien data haetaan vasta selaimen puolella, muu osa sivusta tulee kuitenkin valmiina palvelimelta Nextin ansiosta. Esimerkiksi navigaatiopalkki on jo valmiina HTML-sivussa, kun selain vasta on aloittamassa Reactin suorittamista.

Nextissä on ominaisuus nimeltä SSG, joka luo automaattisesti staattiset versiot niistä sivuista, jotka eivät hae dataa palvelimen puolella esimerkiksi SEO-hyötyjen takia [19]. Tämä tarkoittaa sitä, että kun Nextiltä pyydetään SSG-optimoitua sivua, Next ei alakaan itse suorittamaan Reactia rakentaakseen tämän sivun, vaan Next palauttaa jo koontivaiheessa luodun sivun. Tämä on siis käytännössä sama prosessi, mitä staattisia sivuja tarjoavat palvelimet tekevät. Ne palauttavat valmiiksi luotuja sivuja ilman, että niitä prosessoidaan palvelimella. Kun Next-projekti kootaan, SSG tunnistaa automaattisesti sellaiset sivut, jotka voidaan optimoida. Näistä sivuista luodaan sitten HTML-sivut jo koonnin aikana. Kun palvelimelle tulee pyyntö johonkin tällaiseen sivuun, Next voi vain palauttaa jo valmiiksi rakennetun sivun tarjoten erittäin nopean vastausajan. Tästä seuraa todella nopeat vastausajat optimoiduille sivuille, ja myös esimerkiksi kustannussäästöjä, kun palvelimen resursseja ei tarvitse käyttää näiden sivujen rakentamiseen jokaisella pyynnöllä. SSG:tä ei voi käyttää sellaisille sivuille, jotka hakevat dataa palvelimen puolella. Aluksi Next-demosivu luotiin siten, että TV-ohjelmien data haettiin palvelimella, joten TV-ohjelmia näyttävä sivu ei ollut SSG-optimoitu. Kun sivu siirrettiin käyttämään selainpuolen asynkronisia kutsuja API-datan hakemiseen, Next optimoi sen automaattisesti. Tämä lisäsi entisestään Next-demoapplikaation suorituskykyä. Automaattinen optimointi tuntuu todella tehokkaalta ominaisuudelta, koska se on täysin automaattinen, eikä vaadi mitään lisätyötä ohjelmoijalta, mutta antaa kuitenkin suuria hyötyjä.

Reduxin käyttö demoprojektissa vaati jonkin verran muutoksia siinä vaiheessa, kun API-data haettiin vielä palvelimen puolella. Siinä tapauksessa tarvitsemme Redux-tilataltion sekä palvelimen että selaimen puolella. Palvelimen puolella Redux-tila luodaan jokaiselle pyynnölle, mutta selaimen puolella Redux-tila on pysyvä. Tällöin piti ottaa käyttöön kirjasto next-redux-wrapper, joka mahdollistaa Reduxin käyttämisen palvelinpuolella renderöityjen sivujen kanssa. Nämä muutokset kuitenkin rampauttivat erään Nextin tärkeistä ominaisuuksista, sillä automaattista staattista optimointia ei enää pystynyt käyttämään missään sivuston sivuista. Tätä käyttötapausta varten täytyisi hyväksyä se, ettei SSG:tä voida käyttää. Toinen vaihtoehto olisi keksiä jokin ratkaisu siihen, miten Redux-tilan käyttöä voidaan välttää niissä metodeissa, joilla haetaan dataa komponenteille palvelimella. Kuitenkin siinä vaiheessa, kun demoapplikaatio muutettiin hakemaan data selaimen puolella, next-redux-wrapper-kirjastoa ei enää tarvinnut käyttää. Tällöin Reduxia käytettiin täysin samalla tavalla kuin SPA-sivussa ja voitiin silti käyttää SSG:n tarjoamia hyötyjä.

Aiemmassa CRA-demoapplikaatiossa Bootstrap-tyylit otettiin käyttöön App-komponentissa, jolloin ne olivat saatavilla muulle applikaatiolle. Nextin kanssa toimittiin saman tapaisesti, mutta koska App-komponenttia ei enää ollut olemassa Nextin puolella, luotiin uusi tiedosto src/pages/_app.js. Se on erityinen tiedosto Nextissä, jonka määrittämällä pystyy globaalisti kontrolloimaan kaikkien sivujen alustusta. Yksi _app.js -tiedoston käyttötapauksista on juurikin globaalien CSS-tyylien käyttöönotto. CSS-tyylit otettiin käyttöön viittaamalla niihin tässä tiedostossa. Lisäksi tässä tiedostossa alustettiin Redux-kirjaston Provider-komponentti, joka tarjoaa Redux-tilataltion applikaatiolle.

Next-applikaation tuottaminen onnistui melko hyvin. Siihen meni kuitenkin enemmän aikaa kuin Create React App -työkalulla luodun sivun tekemiseen. Tämä johtui suurimmaksi osaksi varmaankin siitä, että Next oli minulle ennestään tuntematon työkalu, kun taas CRA oli jo tuttu. Ulkoisesti Next-applikaatiosta tuli saman näköinen kuin aiemmin tuotetusta kuvan 7 CRA-demosivusta.

Yhteenvetona tämän projektin siirtäminen CRA:sta Nextillä toteutetuksi onnistui suhteellisen helposti. Suurimmat muutokset liittyivät sivujen hallintaan, reitittämiseen ja Reduxin käyttöön. CRA:ssa niin kutsuttuja sivuja ei sinänsä edes ole, mutta Nextin maailmassa sivut ovat keskeinen käsite ja sivuston reititys perustuu niihin. Reduxin käyttäminen onnistuu yksinkertaisissa tilanteissa helposti ja melko samalla tavalla kuin CRA:n kanssa,

mutta palvelinpuolen renderöinti yhdistettynä Reduxiin vaatii työtä. Tällöin joudutaan tilanteeseen, jossa Redux-tilataltio täytyy luoda jokaisella kutsulla palvelimelle, mutta myös selaimen puolella. Tätä käyttötapausta varten on olemassa kirjasto next-redux-wrapper, mutta sen käyttäminen poistaa käytöstä Nextin tehokkaan SSG-ominaisuuden. Työn tekemisen aikana tutkittiin Bootstrapin lisäksi myös Styled Components -kirjastoa. Se on suosittu tyylittelykirjasto Reactille. Tämä kirjasto tukee myös palvelinpuolen renderöintiä, joten sen käyttö onnistuu pienillä muutoksilla myös esimerkiksi Nextin kanssa. Täytyy pitää mielessä, että tuotetut applikaatiot olivat yksinkertaisia eikä niissä käytetty suurta määrää ulkoisia kirjastoja. Yksi epävarmuustekijä Nextin käyttöönotossa onkin se, että kaikki tarvittavat kirjastot eivät pakolti tue palvelinpuolen renderöintiä.

5 Demosivujen vertailu ja auditoinnin tulokset

Demosivujen toteuttamisen jälkeen oli aika arvioida niiden ominaisuuksia. Työn tarkoituksena oli arvioida, voivatko isomorfiset web-applikaatiot ratkaista SPA-sivuihin liittyviä ongelmia. Käytännössä tämä tarkoittaa sen tutkimista, tarjoaako palvelinpuolen renderöinti sivun ensimmäisellä latauksella hyötyjä suorituskykyyn ja hakukoneoptimointiin. Sivujen suorituskyvyn ja SEO:n auditointiin valittiin Googlen Lighthouse -työkalu. Lighthouse tulee Googlen Chrome-selaimen mukana, ja sillä voidaan auditoida sivustojen ominaisuuksia. Lighthouse pystyy auditoimaan sivustojen suorituskykyä, saavutettavuutta, hyviä toimintatapoja, hakukoneoptimointia sekä progressive web app -valmiutta. Lighthouse antaa pisteet kustakin kategoriasta välillä 0–100, jossa 100 on täydet pisteet.

5.1 Raskaan sivuston simulointi

Tuotetut demosivut ovat yksinkertaisia, joten ne eivät itsessään kuvaa hyvin oikean elämän tilannetta isosta web-applikaatiosta. Suorituskyky kummallakin on erittäin hyvä, koska palvelimelta haettavan JavaScript-koodin koko on pieni, sivuston vaatima prosessointi ei ole suurta, ja sivusto ei sisällä suurta määrää DOM-elementtejä. Näitä kolmea ominaisuutta varten sivustoihin toteutettiin tarkoituksenmukaisia hidasteita.

Palvelimelta haettavan JavaScript-koodin kokoa kasvatettiin siten, että projekteihin asennettiin useita riippuvaisuuksia, jotka haetaan koodissa mutta joita ei oikeasti käytetä. Nämä kirjastot olivat Ant Design, Material UI ja Moment. Sen lisäksi että ne asennettiin projektiin, nämä kirjastot importattiin koodissa, jotta niiden koodit oikeasti haetaan aina muunkin koodin mukana. Tämä lisäsi huomattavasti sen koodin kokoa, joka haetaan palvelimelta sekä CRA:lla että Nextilä, jotta sivustoa voidaan käyttää. Se siis lisäsi sivuston latausaikaa tekemällä HTTP-pyynnöistä hitaampia.

DOM-elementtien määrää kasvatettiin tarkoituksenmukaisesti tietoja-sivulla generoimalla sinne paljon elementtejä. Lisäksi tietoja-sivu valittiin myös kohteeksi raskaan prosessointityön simuloimiseen. Aina kun tietoja-sivun avaa, se laukaisee koodissa simuloidun raskaan laskentatyön, joka kuormittaa prosessoria. Tämän laskentatyön koko on sama kummassakin demoapplikaatiossa. Laskentatyö on käytännössä vain erittäin suuri toisteisrakenne. Näihin tarkoituksiin valittiin tietoja-sivu sen takia, jottei TV-ohjelmien hakusivun käyttöä haitattaisi. Tietoja-sivu simuloi siis isoa applikaatiota koodin koon, elementtien määrän sekä prosessointityön kannalta.

5.2 Suorituskyky

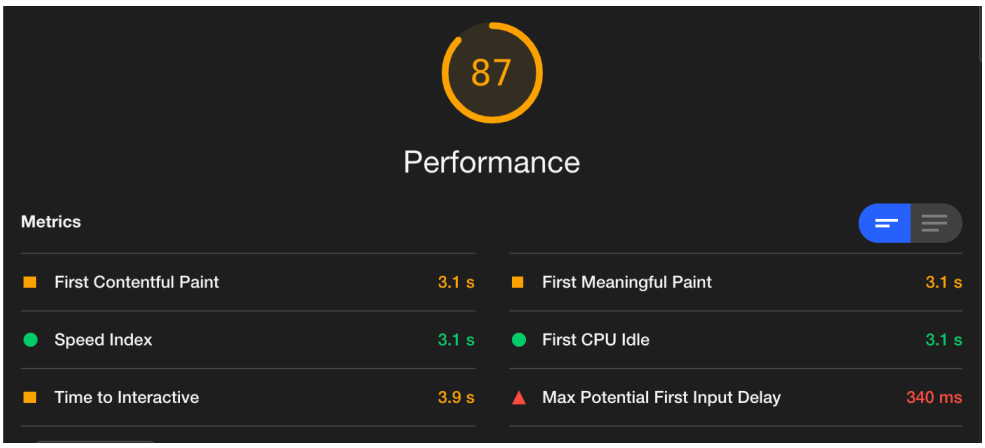
SPA-sivut ovat yleisesti ottaen nopeita käyttää lukuun ottamatta sivun ensimmäistä latausta. Sivun ensimmäinenkin lataus voi nykyisillä tehokkailla laitteilla ja internetyhteyksillä olla todella nopea, eikä mitään viiveitä välttämättä edes huomaa. Tilanne voi muuttua, jos esimerkiksi internetyhteys on hidas tai sivu avataan vaikkapa vanhalla puhelimella. Tällöin voidaan mahdollisesti saada parempia suorituskykytuloksia vaihtoehtoisilla tavoilla, kuten isomorfisilla web-applikaatioilla.

Tässä työssä tuotettujen sivujen suorituskyvyt selainrenderöinnissä vaikuttaisivat olevan yhtä hyvät ainakin silmämääräisen arvioinnin perusteella. Tämä johtuu siitä, että kummankin sivun käyttöliittymät ovat toteutettu Reactilla ja ne käyttävät paljon samaa koodia. React hoitaa sivun päivittämisen suoraan selaimessa sen jälkeen, kun sivu on ensimmäisen kerran saatu palvelimelta. Selainrenderöinti tarkoittaa siis sivun toimintaa ensimmäisen latautumisen jälkeen, jolloin sivuja ei pyydetä palvelimelta vaan sivustoa päivitetään paikallisesti selaimessa, kun käyttäjä navigoi sivustolla. Kun käyttäjä klikkaa auki jonkin TV-ohjelman tietosivun, selain päivittää tietosivun näkyviin paikallisesti ja tekee

kutsun TV-ohjelmien API:in. Sekä CRA:lla että Nextillä toteutetut demosivut toimivat näin. Mutta kun alettiin tutkia sivujen ensimmäisen latautumisen nopeutta, demosivujen suorituskyvyissä huomattiin eroja.

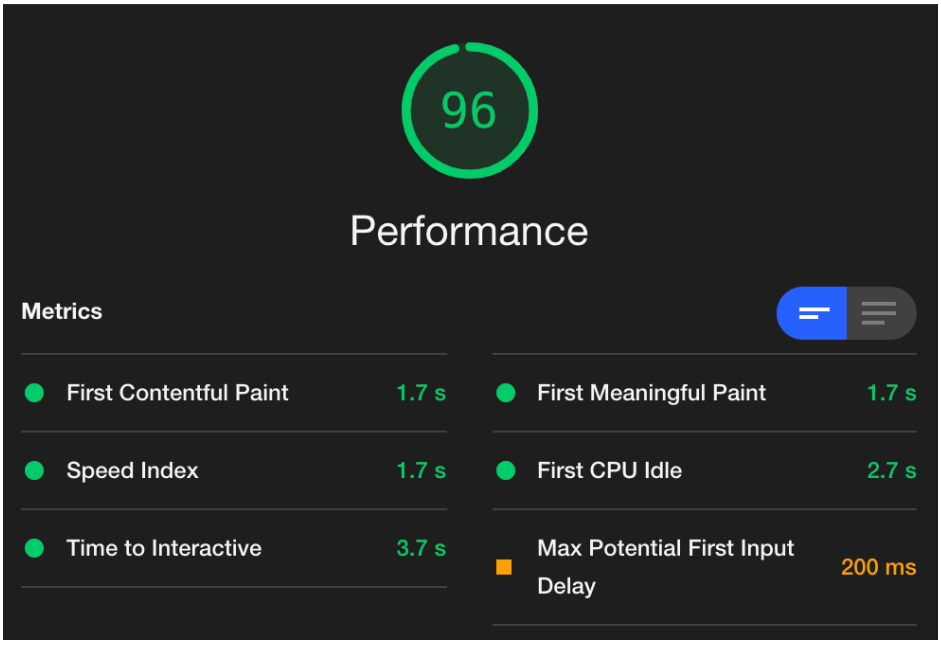
Lighthousen avulla ei pysty mittaamaan React-applikaation selainrenderöinnin suorituskykyä ensimmäisen sivun latautumisen jälkeen. Tämä johtuu siitä, että suorituskyvyn mittaaminen sivun päivittämisen tai ensimmäisen latauksen jälkeen vaatisi vuorovaikutusta käyttäjän ja sivuston välillä. Käyttäjän täytyisi esimerkiksi klikkailla linkkejä, ja samaan aikaan sivuston suorituskykyä täytyisi jotenkin mitata. Sen sijaan Lighthousea voi käyttää sivun ensimmäisen latautumisen tai päivittämisen aikaisen suorituskyvyn mittaamiseen. Yksinkertaistettuna tämä tarkoittaa sivun latautumiseen kuluvan ajan mittaamista, kun sivun osoitteeseen navigoidaan. Sen lisäksi Lighthouse mittaa esimerkiksi kuinka paljon aikaa kului siihen, että ensimmäinen sisällöllinen piirto tapahtui selaimessa.

Aivan ensimmäiseksi Lighthouse-auditointia ajettiin kumpaakin sivua vasten siten, ettei niissä ollut käytössä aiemmin mainittuja hidasteita. Tällöin suorituskykytulokset olivat todella lähellä toisiaan kummassakin, lähellä täydellistä 100 pisteen tulosta Lighthousessa. Mutta kun raskasta ohjelmaa simuloivat hidasteet otettiin käyttöön, erot demosivujen suorituskyvyissä korostuivat. Tästä eteenpäin kaikissa suorituskykytuloksissa edellä mainitut hidasteet ovat käytössä. Kuva 8 ja kuva 9 esittävät auditoinnin suorituskykytuloksia TV-ohjelmien hakusivulla SPA- ja Next-demosivuilla.



Kuva 8. Lighthouse-suorituskykyauditointi hakutulosten sivusta (SPA).

Voimme huomata, että SPA-demosivulla kului 3,1 sekuntia siihen, että ensimmäinen sisältöä näyttävä piirto tehtiin selaimessa. Tätä edeltävä aika kului siis JS-koodien lataamiseen ja Reactin suorittamiseen. Siitä eteenpäin kului vielä 0,8 sekuntia, että sivusto oli täysin interaktiivinen.



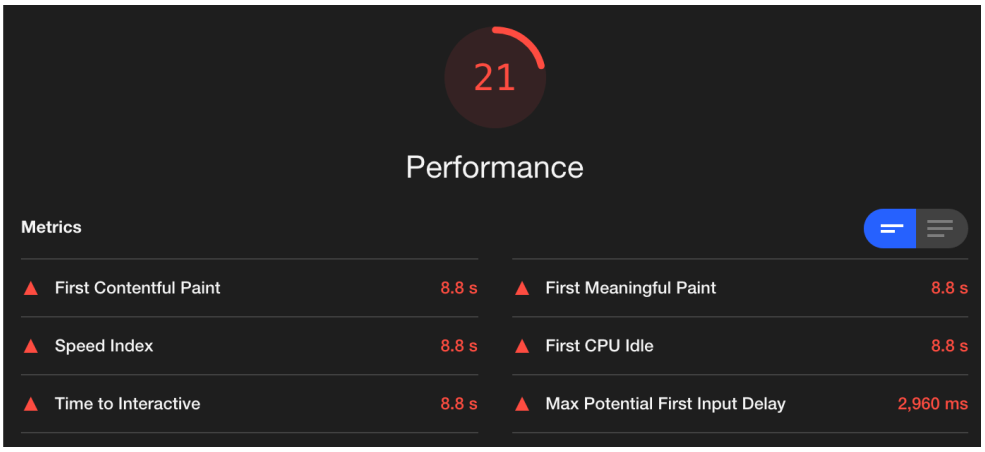
Kuva 9. Lighthouse-suorituskykyauditointi hakutulosten sivusta (Next ja SSG).

Kuten kuvasta 9 voimme huomata, Next sai paremman tuloksen suorituskykytestissä samalla sivulla. Nextillä sivusta tuli interaktiivinen melkein yhtä nopeasti kuin SPA-sivusta (3,7 sekuntia Nextillä, 3,9 sekuntia SPA:lla), mutta ensimmäinen sisällöllinen piirto selaimessa tapahtui huomattavasti nopeammin. Ensimmäinen sisällöllinen piirto tapahtui Nextillä 1,7 sekunnissa, kun taas SPA:lla se vei 3,1 sekuntia.

Tulokset ovat linjassa odotuksiin nähden. Hakusivu on Nextillä staattisesti optimoitu, eli Nextin ei tarvitse suorittaa Reactia palvelimella ennen kuin se voi palauttaa sivun. Tästä syystä Next voi tarjota sivun todella nopeasti, kun taas SPA:lla täytyy ensin hakea sivun pohja, sitten aloittaa Reactin suorittaminen, ja sitten vasta sivu voidaan näyttää. SSG-optimoitu Next-sivu voidaan taas näyttää käyttäjälle heti, kun selain saa sen palvelimelta, ja Reactia voidaan alkaa suorittamaan taustalla samanaikaisesti, kun sivua näytetään jo käyttäjälle. Jos taas Nextillä tuotettu sivu ei olisi staattisesti optimoitu, saataisiin päätyä vastaavanlaiseen tilanteeseen joka tapauksessa. Tällöin palvelin suorittaisi Reactia ja rakentaisi sivun, ja sitten palauttaisi sen selaimelle. Kun selain saa sivun, se piirtää sen

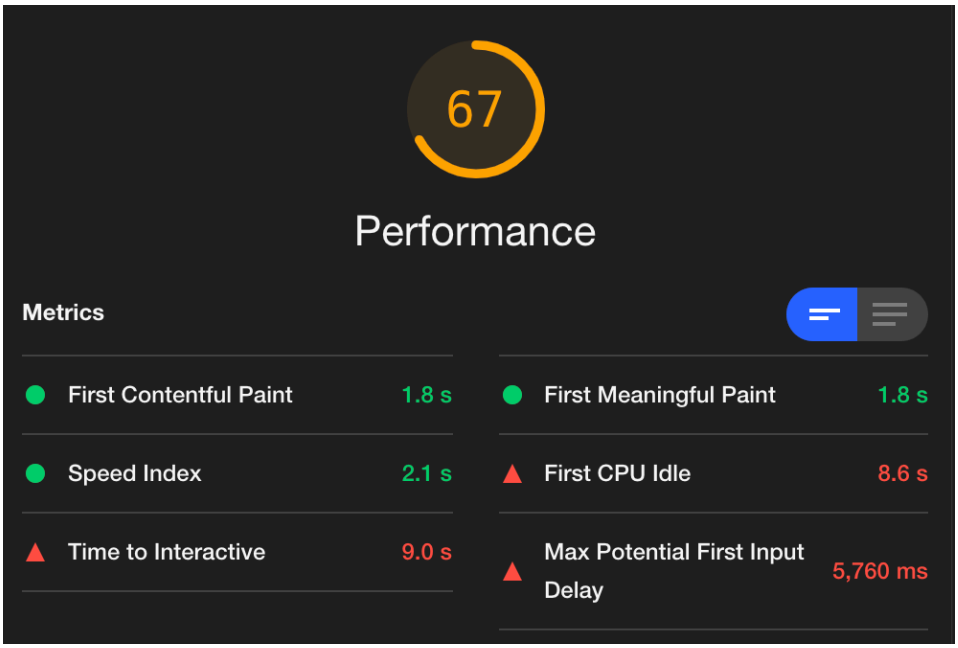
käyttäjälle ja alkaa suorittamaan Reactia samalla. Tässä tapauksessa Next-palvelimen ei tarvitsisi hakea JS-koodia jostakin kaukaiselta lähteeltä (niin kuin selaimen täytyy pyytää se palvelimelta), vaan koodit ovat palvelimella paikallisesti, joten Next-palvelin voi rakentaa pyydetyn sivun nopeasti. Tätä teoriaa testattiin käytännössä, ja sama auditointi suoritettiin myös siten, että Nextin SSG oli tahallaan poistettu käytöstä. Ilman SSG:tä Next sai samanlaisessa auditoinnissa tulokseksi 96 pistettä, kuten myös SSG:n kanssa. Sivun lataamisen viiveet erosivat näissä tapauksissa ainoastaan sekunnin sadasosilla.

Suorituskyvyn auditointi tehtiin myös tietoja-sivulla, johon oli lisätty tahallista raskasta prosessointia sekä paljon DOM-elementtejä. Kuvassa 10 on SPA-sivun tulokset.



Kuva 10. Lighthouse-suorituskykyauditointi tietoja-sivusta (SPA).

Raskas prosessointi aiheutti huomattavan, usean sekunnin viiveen sivun latautumiseen. Koko tämän ajan SPA-sivulla käyttäjälle näkyi vain tyhjä sivun pohja. Kuva 11 esittää auditoinnin tuloksia samasta tietoja-sivusta Nextillä toteutetussa demoapplikaatiossa.



Kuva 11. Lighthouse-suorituskykyauditointi tietoja-sivusta (Next ja SSG).

Tietoja-sivujen auditoinneista voimme nähdä vielä kärjistyneemmän esimerkin siitä, minkälaiset erot raskaiden sivujen latautumisessa voi olla riippuen siitä, saadaanko palvelimelta tyhjä sivun pohja vai valmiiksi rakennettu sivu. Kummassakin esimerkissä kesti lähes yhtä kauan siinä, että sivu oli kokonaan ladattu ja prosessointi lopetettu. Nextin tilanteessa kesti kuitenkin vain 1,8 sekuntia, kun ensimmäinen sisällöllinen piirto tehtiin selaimessa, kun vastaava aika CRA-sivulla oli 8,8 sekuntia. Tietoja-sivu oli staattisesti optimoitu Nextillä, minkä takia ensimmäinen piirto pystyttiin tekemään niin nopeasti. Jos sivu vaatisi palvelinpuolen renderöintiä, olisivat erot näiden esimerkkien välillä paljon pienemmät.

Suorituskykyauditoinnin tulokset osoittavat, että staattisten React-sivujen latausnopeus on huomattavasti nopeampaa kuin tavallisten selaimessa rakennettujen React-sivujen. Myöskin ei-staattisten palvelimella rakennettujen React-sivujen lataaminen voi olla nopeampaa, koska palvelimen ei tarvitse hakea tarvittavia koodeja verkon yli ja se voi hyödyntää palvelimen omaa tehokasta suorituskykyä. Näissä kummassakin tapauksessa selain voi alkaa piirtää palvelimelta saatua HTML-sivua jo samanaikaisesti, kun se aloittaa React-koodin suorittamisen.

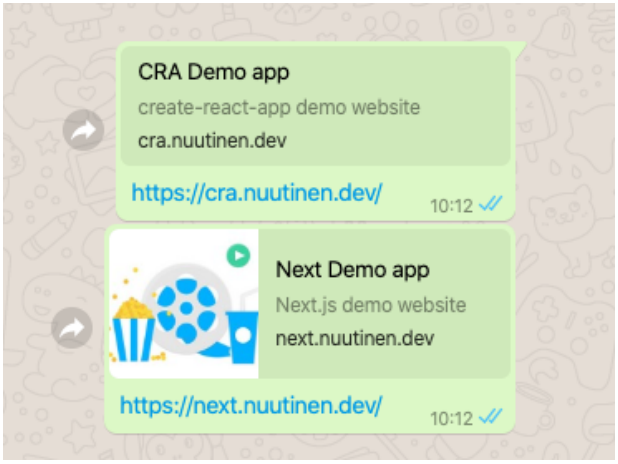
5.3 Hakukoneoptimointi

HTML-sivujen head-elementtiin voi lisätä metatietoja sivusta. Nämä metatiedot kuvailevat sivun sisältöä, tarkoitusta ja ominaisuuksia. Ne ovat tärkeitä SEO:n kannalta, koska niiden avulla sivustosta voidaan saada tietoa, vaikkei muuta sivua edes tutkittaisi. Näihin metatietoihin sisältyvät esimerkiksi sivuston kuvaus, tekijä ja avainsanat. Sivun metatietoja pystyy helposti muokkaamaan sekä Nextillä että tavallisella Reactilla. Kummatkin sivut saivat täydet 100 pistettä SEO-auditoinnissa. Työn demosivuissa ei ole huomattavia eroja SEO:n kannalta, koska kummassakin demosivussa itse sivun tietosisältö eli TV-ohjelmat haetaan vasta selaimessa. Nextillä TV-ohjelmat pystyisi kuitenkin hakemaan jo palvelimenkin puolella, mutta koska kummatkin sivut saivat jo täydet 100 pistettä ilman palvelinrenderöintiä, SEO:ta ei sen enempää voi auditoida Lighthousen avulla. Tietoa oikeassa käytössä olevan sivuston hakukoneystävällisyydestä voisi varmaankin saada esimerkiksi tutkimalla sivustoon kohdistuvan liikenteen määrää ennen ja jälkeen palvelinrenderöinnin käyttöönottoa.

Nextissä on tuki palvelinrenderöinnille, eli jos haluaisimme, että TV-ohjelmien tiedot olisivat jo palvelimelta saadussa sivussa, sen tekeminen olisi mahdollista. Tällöin TV-ohjelmien tiedot olisivat varmasti osana hakukoneiden indeksointia. TV-ohjelmien haku palvelimen puolella tulisi kuitenkin suorituskyvyn kustannuksella, eli siinä tilanteessa täytyisi miettiä kumpi on tärkeämpää: suorituskyky ensimmäisellä latauksella vai hakukoneoptimointi. On kuitenkin olemassa tapoja, joilla palvelinrenderöinnin suorituskykyä voidaan parantaa. Suurin hidaste palvelinrenderöinnissä tässä tapauksessa on TV-ohjelmien tietojen hakeminen ulkoisesta tietolähteestä. Voisimme kuitenkin ehkäistä tätä tallentamalla itse ulkoisesta lähteestä haettuja TV-ohjelmia palvelimelle välimuistiin tai paikalliseen tietokantaan. Tällöin TV-ohjelmien tiedot käytäisiin silloin tällöin päivittämässä ulkoisesta lähteestä, mutta useimmiten tiedot haettaisiin palvelimen lähellä olevasta paikallisesta kannasta tai jopa välimuistissa olevasta tilapäistaltiosta.

Toinen hakukoneoptimointiin ja palvelinrenderöintiin liittyvä seikka on sivun esikatselun näyttäminen. Tässä tapauksessa tutkitaan sivun tietojen näyttämistä WhatsApp-viestintäohjelmassa silloin, kun sivun linkki lähetetään viestissä. Muita samankaltaisia tilanteita ovat esimerkiksi sivun esikatselun näyttäminen sosiaalisessa mediassa tai muissa viestintäpalveluissa. Normaalin React-sivun tiedoista WhatsApp pystyy näyttämään metatie-

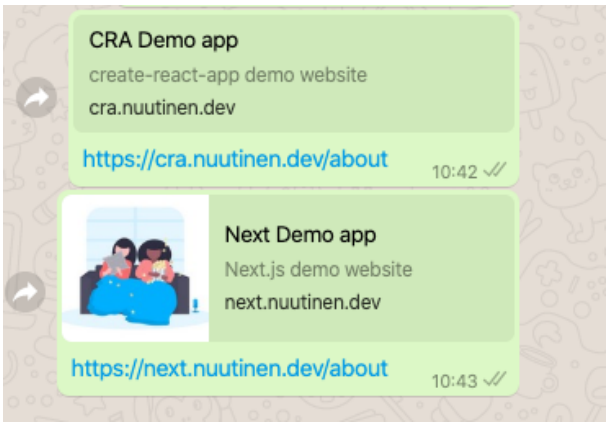
doissa esitellyn otsikon sekä sivun kuvauksen. Nextin tilanteessa WhatsApp pystyy näiden tietojen lisäksi näyttämään myös sivun alkuosassa olevan kuvan. Tätä havainnollistaa kuva 12.



Kuva 12. Ruutukaappaus demosivujen linkeistä WhatsApp-viestintäsovelluksessa.

WhatsApp näyttää automaattisesti sivuston alkuosassa olevan kuvan, kun sivu linkitetään viestissä.

Kummankin demosivun SEO on pelkkien metatietojen perusteella hyvä, ainakin Lighthouse-auditoinnin perusteella. Lighthousen SEO-auditointi ei kuitenkaan kerro kaikkea. Esimerkiksi linkit sivustolla ovat tärkeä osa SEO:ta, ja monessa tapauksessa näitä linkkejä lisätään sivuun vasta renderöinnin yhteydessä. Tällöin on tärkeää, että palvelimelta tulevassa sivussa on jo nämä sisällöt valmiiksi. Toinen ero sivujen välillä on se, että CRA-sivu sisältää vain yhden loogisen sivun hakukoneiden silmissä, kun taas Nextillä toteutettu sivu sisältää useamman sivun. Esimerkiksi tietoja-sivu voitaisiin näyttää hakutuloksissa erikseen Nextin tapauksessa, koska pyyntö sille sivulle palauttaa oikeasti tietoja-sivun, kun taas perinteisellä Reactilla palautettaisiin siltikin vain sivun pohja. Tätä eroa loogisten sivujen olemassaolossa normaalin Reactin ja Nextin välillä näytetään kuvassa 13.



Kuva 13. Ruutukaappaus demosivujen tietoja-sivun linkeistä WhatsApp-viestintäsovelluksessa.

Kuvasta 13 voimme huomata, että myös tietoja-sivusta näytetään Next-toteutuksessa juuri sen sivun esikatselua silloin, kun sivu tulee valmiina palvelimelta. Normaalin Reactin tapauksessa kutsu tietoja-sivulle palauttaa täysin saman sivun pohjan kuin kutsu mille tahansa muullekin sivulle (vrt. kuva 12).

6 Yhteenveto

Tässä työssä oli tavoitteena tutkia, voiko palvelimen puolella suoritettavan Reactin avulla parannella kokonaan selaimen puolella renderöidyn SPA-sivun suorituskykyä ja hakukoneoptimointia. Työssä vertailtiin selain- ja pelvelinpuolen renderöinnin ominaisuuksia, toteutettiin samanlaiset sivut perinteisenä SPA:na sekä palvelinrenderöintiä käyttävänä SPA:na, ja lopuksi vertailtiin toteutettujen sivujen teknisiä ominaisuuksia. Sivustojen suorituskykyä ja hakukoneoptimointia mitattiin Googlen Lighthouse-työkalua käyttäen.

Lighthouse-auditoinnin tuloksista nähtiin, että perinteisen, täysin selaimessa renderöidyn React-sivun ensimmäisen sivun latauksen suorituskykyä pystytään parantelemaan tietyin keinoin. Näitä keinoja ovat joko Reactin suorittaminen ja renderöinti palvelimella tai staattisten sivujen palauttaminen palvelimelta. Myös hakukoneoptimointia voidaan parannella, kun otetaan mukaan jompikumpi samoista taktiikoista.

React-sovelluksen ehkä suurin suorituskykyongelma on se, että sivun ensimmäisen latauksen aikana hoidetaan monta eri asiaa sillä aikaa, kun käyttäjä näkee vain tyhjän sivun pohjan. JS-koodit ja tyylit noudetaan palvelimelta, koodeja aletaan suorittamaan ja

koodin suorittamisen jälkeen näytetään sivu käyttäjälle. Jos taas palvelimelta saadaan valmis sivu, selaimella on ensimmäisellä latauksella vähemmän asioita hoidettavana ennen sivun piirtämistä. Tällöin palvelimelta saadun sivun tyylit haetaan, sivu näytetään käyttäjälle, ja samaan aikaan haetaan JS-koodit sekä aletaan suorittamaan niitä. Sivun näyttäminen käyttäjälle alkaa siis nopeammin, jos palvelimelta saadaan sisällöllinen sivu tyhjän pohjan sijaan. On olemassa useampi tapa, jolla voidaan hoitaa se, että palvelimelta saadaankin sisällöllinen sivu React-applikaatiossa. Sivu voi olla jo valmiiksi rakennettuna palvelimella (staattinen sivu), ja palvelin vain pyydettäessä palauttaa tämän sivun. Voidaan myös käyttää palvelinpuolen renderöintiä, eli että palvelin suorittaa React-koodia rakentaen sivun ja sitten palauttaa sen selaimelle. Tässä työssä käytettiin Next-sovelluskehystä, joka pystyy automaattisesti hoitamaan sekä staattisten sivujen palvelun että palvelinpuolen renderöinnin. Toisiakin vaihtoehtoja on, ja palvelinpuolen renderöinnin pystyy tekemään myös manuaalisesti itse ilman valmiita sovelluskehyksiä, koska React tukee DOM:in renderöintiä tekstijonoksi. Palvelin siis suorittaisi Reactia ja sitten muuntaisi DOM:in tekstijonoksi, jonka se sitten palauttaisi selaimelle. Tämä on siinä mielessä joustava ratkaisu, että siinä ei tarvitse ottaa ulkoisia sovelluskehyksiä käyttöön. Se vaatii tietenkin oman työnsä, koska palvelin täytyy ohjelmoida itse. Siispä tässä työssä käytettiin Nextin valmista ratkaisua.

Tuloksista näkyivät selvät suorituskykyparannukset, kun käytettiin Nextiä perinteisen selainpuolella renderöidyn Reactin sijaan. Palvelinrenderöinti tarjoaa hyvät puitteet myös hakukoneoptimointiin ja sosiaalisen median esikatseluihin. Next itsessään osoittautui helppokäyttöiseksi ja tehokkaaksi ratkaisuksi isomorfisen web-applikaation tuottamiseen, mutta siihen liittyy kuitenkin tiettyjä asioita, jotka on syytä pitää mielessä. Valmiiksi tuotetun React-applikaation siirtäminen Nextiin voi osoittautua haasteelliseksi. Tämä riippuu käytetyistä kirjastoista ja niiden SSR-tuesta sekä siitä, kuinka monimutkainen olemassa oleva sivu on. Suosittu Redux-kirjasto tukee palvelinrenderöintiä, mutta Nextin kanssa sen käyttö saattaa tietyissä tilanteissa vaatia työtä, jotta sen saa toimimaan kunnolla. Next vaatii tietyn rakenteen projektilta, joten siirtymistä perinteisestä Reactista Nextiin kannattaa harkita tarkoin. Jos taas ollaan kehittämässä täysin uutta applikaatiota, Next voisi olla hyvä valinta sen toteuttamiseen helppoutensa ja tehokkuutensa ansiosta.

Lähteet

1 Definitions of Web-related terms: HTML. 2019. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Glossary/HTML>. Luettu 18.01.2020.

2 Definitions of Web-related terms: CSS. 2019. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Glossary/CSS>. Luettu 18.01.2020.

3 Definitions of Web-related terms: JavaScript. 2019. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Glossary/JavaScript>. Luettu 18.01.2020.

4 Kaulinzski, Philip; Moore, John. 2016. Mastering JavaScript Single Page Application Development. Birmingham: Packt Publishing Ltd.

5 Mohammad, Majid; Golrokh, Mirzaei. 2015. Mastering AngularJS for .NET Developers. Birmingham: Packt Publishing Ltd.

6 Definitions of Web-related terms: Time to first byte. 2019. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Glossary/time_to_first_byte>. Luettu 22.03.2020.

7 Mueller, Britney. How Search Engines Work: Crawling, Indexing, and Ranking. 2020. Verkkoaineisto. <https://moz.com/beginners-guide-to-seo/how-search-engines-operate>. Luettu 29.03.2020.

8 Understand the JavaScript SEO basics. 2019. Verkkoaineisto. Google. <https://developers.google.com/search/docs/guides/javascript-seo-basics>. Luettu 20.01.2020.

9 Mueller, Britney. The Beginner's Guide to SEO. 2020. Verkkoaineisto. <https://moz.com/beginners-guide-to-seo>. Luettu 29.03.2020.

10 Rauschmayer, Axel. Is “Isomorphic JavaScript” a good term? 2015. Verkkoaineisto. <https://2ality.com/2015/08/isomorphic-javascript.html>. Luettu 29.03.2020.

11 Next.js by ZEIT - The React Framework. 2020. Verkkoaineisto. ZEIT. <https://nextjs.org>. Luettu 29.03.2020.

12 Martin, Shifa. The Top JavaScript Frameworks For Front-End Development in 2020. Verkkoaineisto. freeCodeCamp. <https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks-2019>. Luettu 08.04.2020.

13 Google Trends. 2020. Verkkoaineisto. <https://trends.google.com>. Google. Luettu 01.03.2020.

14 Introducing JSX. 2020. Verkkoaineisto. Facebook. <https://reactjs.org/docs/introducing-jsx.html>. Luettu 29.03.2020.

15 Create a New React App. 2020. Verkkoaineisto. Facebook. <https://reactjs.org/docs/create-a-new-react-app.html>. Luettu 08.04.2020.

16 Introduction to ZEIT Now. 2020. Verkkoaineisto. ZEIT. <https://zeit.co/docs>. Luettu 08.04.2020.

17 Abramov, Dan; Dorr, Timothy. React-redux – npm. Verkkoaineisto. <https://www.npmjs.com/package/react-redux>. Luettu 15.03.2020.

18 Lifting State Up. 2020. Verkkoaineisto. Facebook. <https://reactjs.org/docs/lifting-state-up.html>. Luettu 15.03.2020.

19 Pages. 2020. Verkkoaineisto. ZEIT. <https://nextjs.org/docs/tag/v9.2.2/basic-features/pages>. Luettu 08.04.2020.