Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Joonas Eskonheimo

# Offline-ominaisuuden implementointi PDM-järjestelmään Service Workeria hyödyntäen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

23.04.2020

| Tekijä<br>Otsikko<br><br>Sivumäärä<br>Aika | Joonas Eskonheimo<br>Offline-ominaisuuden implementointi PDM-järjestelmään Service Workeria hyödyntäen<br>49 sivua<br>23.04.2020 |
|---|---|
| Tutkinto | insinööri (AMK) |
| Tutkinto-ohjelma | Tieto- ja viestintätekniikka |
| Ammatillinen pääaine | Ohjelmistotuotanto |
| Ohjaajat | Aluejohtaja Tero Hurskainen<br>Lehtori Simo Silander |




Insinöörityön tilaajana toimii TECHNIA Oy, mutta työn lopputulos toimitettiin TECHNIA Oy:n asiakasyritykselle. Asiakasyritys toimii teollisuuden alalla eikä sen nimeä tulla tässä insinöörityössä mainitsemaan.

Insinöörityössä implementointiin asiakkaan 3DExperience-alustalla olevan tuotetietojen hallinta -järjestelmään TVC Helium -ohjelmiston offline-ominaisuus. Kyseinen ominaisuus on toteutettu hyödyntäen moderneja web-teknologioita, kuten service workeria ja workboxia. Offline-ominaisuus tuo mahdollisuuden asiakkaalle käyttää järjestelmää yhteydettömässä tilassa tietyin rajoituksin.

TECHNIA Oy:n asiakasyrityksellä on ollut pidemmän aikaa suuri tarve saada heidän PDM-järjestelmän tuotetietojen katselmoimiseen käytettävä käyttöliittymä toimimaan yhteydettömässä tilassa. Heillä on tehtaita ympäri maailmaa, ja he käyttävät muun muassa tabletteja tehtaillaan tuotetietojen katselmoimiseen. Ongelmana on ollut, että tehtaalla internetyhteyttä ei ole aina saatavilla tai se on erittäin heikko.

Tavoitteeksi asetettiin yhteydettömässä tilassa työskentely tietyin vaatimuksin sekä toissijaisena tavoitteena latausaikojen lyhentäminen raskaimmissa välilehdissä. Latausaikojen lyhenemisen mittarina toimi järjestelmän raskaimman välilehden latausaika, jota mitattiin ennen ja jälkeen sovitusta.

Lopputuloksena saatiin asiakasyrityksen PDM-järjestelmän tuotetietojen katselmoimiseen käytettävä käyttöliittymä toimimaan yhteydettömässä tilassa vaatimuksien mukaisesti. Latausaika järjestelmän raskaimmassa välilehdessä lyheni neljäsosaan alkuperäiseen verrattuna.

Tämä insinöörityö on TVC Helium -ohjelmiston offline-ominaisuuden ensimmäinen tuotantoon toteutettu sovitus asiakkaalle. Insinöörityön pohjalta kerättiin TVC Heliumin tuotekehitystiimille kehitysehdotuksia, joita käydään läpi insinöörityön lopussa.




| Avainsanat | PDM, Service worker, Workbox, JavaScript |
|---|---|

| | |
|---|---|
| Author<br>Title<br><br>Number of Pages<br>Date | Joonas Eskonheimo<br>Implementing Offline Functionality to PDM System Utilizing Service Worker<br><br>49 pages<br>23 April 2020 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information and Communication Technology |
| Professional Major | Software Engineering |
| Instructors | Tero Hurskainen, Business Area Director<br>Simo Silander, Senior Lecturer |

This thesis was made for TECHNIA Oy and delivered to its customer. The customer operates in the industrial field and is to be remained anonymous.

The study is about implementing TVC Helium's offline functionality for the customer's PDM system running on a 3DExperience platform. The offline functionality enables the usage of the system offline with some limitations. The functionality was developed using modern web technologies such as service worker.

The customer have had the need to get their PDM system UI for viewing product information working offline. They have factories all over the world where they use the UI with e.g. tablets. The problem has been the unavailability of internet connection in parts of the factories or it being very weak.

The main goal of the study was to enable working offline with some limitations. The secondary goal was to decrease page loading time in heavy tabs and pages in the system. The decreasing of page loading time was tested by measuring the loading time of the heaviest tab in the system before and after the implementation.

The outcome was the PDM system UI for viewing product information working offline successfully. The secondary goal was also reached as the loading speed of the heaviest tab was decreased four times compared to before the implementation was made.

The present study was the first implementation of TVC Helium's offline functionality made to production for a customer. The study resulted in improvement suggestions to TVC Helium's product developers also covered here.

| | |
|---|---|
| Keywords | PDM, Service worker, Workbox, JavaScript |

## Sisällys

Lyhenteet

1 Johdanto 1

2 Teoriakehys 2

2.1 Service Worker 2
2.2 JavaScript ja sen rajapintoja 3
2.2.1 Fetch API 4
2.2.2 Promise 5
2.2.3 Cache API 6
2.3 Workbox 7
2.4 Moduuli-suunnittelumalli 12
2.5 PDM 12
2.6 3DExperience 13
2.7 TVC 13
2.8 PWA 14

3 Toimeksianto/projekti 14

3.1 Tarve 14
3.2 Taustaa 14
3.3 Ohjelmistotuotannon vaiheet 14
3.4 Vaatimukset 15

4 Suunnittelu ja toteutus 15

4.1 Offline-ominaisuuden aktivointi ja oman service workerin liittäminen osaksi ohjelmistoa 16
4.1.1 Suunnittelu 16
4.1.2 Toteutus 18
4.2 Resurssien tallentaminen välimuistiin ja välimuistista noutaminen 21
4.2.1 Suunnittelu 21
4.2.2 Ongelmat 24
4.2.3 Toteutus 28

4.3 Yhteyden tilan näyttäminen käyttäjälle 32
4.3.1 Suunnittelu 32
4.3.2 Ongelmat 33
4.3.3 Toteutus 35
4.4 Käyttäjälle ilmoittaminen, kun hän yrittää yhteydettömässä tilassa vierailla sivulla, joka ei ole välimuistissa 39
4.4.1 Suunnittelu 39
4.4.2 Ongelmat 40
4.4.3 Toteutus 40
5 Latausajan mittaus 42
5.1 Mittaus 43
5.2 Mittaustulokset 43
5.3 Päätelmä 44
6 Kehitysehdotukset 45
7 Yhteenveto 46
Lähteet 47

## Lyhenteet

| | |
|---|---|
| API | Application Programming Interface. Ohjelmointirajapinnat määrittelevät, kuinka eri ohjelmat voivat keskustella keskenään. |
| DOM | *Document Object Model*. Kuvaa dokumentin puurakenteena, jonka solmut ovat olioita hierarkiassa. |
| Helium | Progressiivinen web-sovellus, joka kuuluu TVC:n tuoteperheeseen. |
| HTTPS | *Hyper Text Transfer Protocol Secure.* Suojattu WWW:n tiedonsiirtoprotokolla. |
| MVC | *Model View Controller*. Ohjelmistoarkkitehtuuri, jossa on kolme eri kerrosta. |
| PDM | *Product Data Management.* Tuotteen elinkaaren hallinnan yksi osa-alue. |
| PLM | *Product Lifecycle Management*. Tuotteen elinkaaren hallinta. |
| PWA | *Progressive Web App*. Alustariippumaton progressiivinen web-sovellus. |
| RegExp | *Regular Expression*. Säännöllinen lauseke. Käytetään tietojenkäsittelyssä määrittelemään joukkoa merkkijonoja. |
| TVC | *Technia Value Component.* TECHNIA Oy:n ohjelmisto parantamaan 3DExperience alustaa. |

## 1 Johdanto

Insinöörityön tavoitteena oli saada TECHNIA Oy:n asiakasyrityksen PDM-järjestelmä toimimaan tiettyjen vaatimuksien puitteissa ilman internetyhteyttä sekä samalla lyhentämään järjestelmän raskaiden välilehtien latausaikoja. Asiakkaan järjestelmä on tarkoitettu tuotetietojen hallintaan, ja se on tehty Dassault Systemesin 3DExperiencen alustalle, jota on kustomoitu TECHNIA:n omalla tuotteella TVC Heliumilla. Tämän insinöörityön kohteena on mm. heidän tehtaissaan käytössä oleva tuotetietojen ja erilaisten dokumenttien katselmukseen käytettävä erillinen käyttöliittymä, joka on osa tätä järjestelmää.

Asiakkaalla on tarve saada tuotetietojen katselmukseen käytettävä käyttöliittymä toimimaan yhteydettömässä tilassa. Tehtailla työntekijät käyttävät tabletteja ja kannettavia tietokoneita, eikä internetyhteys ole aina saatavilla tai yhteys voi olla erittäin heikko.

TECHNIA:n tuotekehitys on viimeisen kolmen vuoden aikana etsinyt ja kehittänyt keinoja saada haluttu yhteydettömässä tilassa toimiminen osaksi TVC Helium -tuotettaan. Mozillan kehittämä service worker ja siihen valmiita työkaluja tarjoava Googlen kehittämä workbox valikoituivat tuotekehityksessä työkaluiksi ominaisuuden toteuttamiseen.

TVC Heliumiin on tuotekehityksen puolesta toteutettu service workerin ja workboxin asennus ja konfigurointi sekä omat service workerit hoitamaan tuotteeseen liittyvien resurssien tallentamisen välimuistiin. Tehtäväkseni ja insinöörityön alueeksi jää siis suunnitella ja toteuttaa oma service worker, joka on vastuussa dynaamisesta resurssien tallentamisesta välimuistiin ja sieltä tarjoilusta. Tämä tarkoittaa käyttäjän järjestelmässä navigoinnin aikana tapahtuvien palvelinpyyntöjen vastauksien tallentamista selaimen välimuistiin, jotta yhteydettömässä tilassa ne olisivat saatavilla. TVC Heliumiin liittyi myös useita spesifisiä konfigurointeja, mutta ne on rajattu insinöörityön ulkopuolelle.

Insinöörityön teoriakehyksessä esitellään muun muassa service worker, workboxin tarjoamia erilaisia strategioita välimuistin hallintaan sekä JavaScriptin API:ja, joita insinöörityössä tullaan käyttämään ja jotka ovat olennaisia sen ymmärtämiseksi.

Työn edetessä kohdattiin erilaisia ongelmia, joita vaihtelevin osin käsitellään tässä insinöörityössä riippuen niiden sisältymisestä aiheen rajaukseen. Tämä insinöörityö on TECHNIA:n ensimmäinen tuotantokäyttöön tehty toteutus Heliumin offline-ominaisuudesta. Kohdatuista ongelmista syntyi TECHNIA:n tuotekehitykselle kehitysehdotuksia, joita lopuksi käydään läpi.

Asiakasyritystä ei mainita tässä insinöörityössä, eikä suurinta osaa tehdystä koodista tulla paljastamaan. Kuitenkin kuvilla ja havainnollistavilla koodiesimerkeillä pyritään auttamaan asioiden hahmottamisessa. Asiakasyritykseen viitataan työssä nimellä asiakasyritys X.

# 2 Teoriakehys

## 2.1 Service Worker

Service worker on JavaScript-tiedosto, joka toimii käytännössä välityspalvelimena selaimen ja verkon välissä. Se on tarkoitettu toimimaan osana muita komponentteja ja tarjoamaan tehokkaan yhteydettömän käyttökokemuksen sieppaamalla palvelinpyyntöjä ja suorittamalla haluttuja toimenpiteitä verkon tilasta riippuen [1]. Sen toiminta perustuu pitkälti Fetch API:iin, jolla service worker tekee palvelinpyyntöjä kaapattuaan sille määrätyn pyynnön.

Service worker on tapahtumapohjainen web worker, joka ottaa kontrolliinsa web-sivun, johon se on asennettuna. Web workerien tehtävänä on ajaa JavaScriptiä omassa säikeessään, jotta se ei häiritsisi käyttöliittymän toimintaa. Service workerin tehtävänä on siepata verkko- sekä resurssipyyntöjä ja näin ollen vaikuttaa pyyntöjen ohjaamiseen ja resurssien yksityiskohtaisen tallentamisen selaimen välimuistiin. Ne voivat asentua ja toimia vain sivustolla, joka toimii HTTPS-yhteydessä. HTTPS-yhteys on vaadittu, koska service worker sieppaa palvelinpyyntöjä sekä pystyy muokkaamaan palvelimelta tulevia vastauksia. [1; 2; 3.]

Service worker ei pääse suoraan käsiksi dokumenttioliomalliin (DOM), mutta se voi viestittää hallitsemiensa sivujen kanssa postMessage-rajapinnan välityksellä, ja näin kyseiset sivut voivat manipuloida DOM:ia tarpeen mukaan [4].

Service workerin ominaisuudet mahdollistavat web-sovellusten toimia samalla tavalla kuin natiivit sovellukset. Näitä ominaisuuksia ovat mm:

- Channel Messaging API: Mahdollistaa web workerien ja service workerien kommunikoinnin toistensa sekä sovelluksen kanssa, johon ne kuuluvat.

- Notification API: Tapa näyttää ilmoituksia käyttäjälle ja olla interaktiivinen natiivin käyttöjärjestelmän ilmoituksenhallinnan kanssa.

- Push API: Tämä rajapinta antaa applikaatiolle mahdollisuuden käyttää Push-viestejä. Push-viestit toimitetaan service workerille, joka voi käyttää informaatiota hyödyksi päivittämällä tilaansa tai näyttämällä ilmoituksia käyttäjälle. Tämä toimii myös, kun käyttäjän selain ei ole käynnissä.

- Background Sync API: Kyseinen rajapinta mahdollistaa yhteydettömässä tilassa käyttäjän tekemien toimenpiteiden toteutumisen, kun käyttäjällä on taas vakaa yhteys. Rajapinta mahdollistaa myös serverin lähettämään päivitykset applikaatioon, jolloin applikaatio päivittyy seuraavalla kerralla, kun käyttäjällä on pääsy verkkoon. [2.]

## 2.2 JavaScript ja sen rajapintoja

JavaScript on tullut parhaiten tunnetuksi skriptauskielenä web-sivuille, vaikka sitä nykyään käytetään muissakin ympäristöissä. Se on kevyt, tulkattu olio-ohjelmointikieli. Se on myös dynaaminen, asynkronoitu ohjelmointikieli, joka tukee funktionaalisen ohjelmoinnin tyylejä. [5.]

JavaScript pyörii client-puolella selaimessa ja sitä voidaan käyttää ohjelmoimaan web-sivun käyttäytymistä erilaisten tapahtumien johdosta. Sitä pidetään tehokkaana ohjelmointikielenä, joka on myös helppo oppia. [5.]

JavaScriptillä pystytään noutamaan resursseja ympäri verkkoa. Siihen on käytetty muun muassa XMLHttpRequest (XHR) -objekteja keskustelemaan palvelimien välillä [6; 7]. Nykyään Promise-pohjainen Fetch API tarjoaa yksinkertaisemman ja siistimmän tavan suorittaa palvelinpyyntöjä [8].

Seuraavana esitellään JavaScriptin rajapintoja, jotka on oleellista tuntea service workerin kanssa työskennellessä.

### 2.2.1 Fetch API

Fetch API tarjoaa rajapinnan resurssien hakemiseen verkosta. Sen käyttö muistuttaa XHR:n käyttöä, mutta uusi rajapinta tarjoaa tehokkaammat ja joustavammat ominaisuudet.

Fetch API tarjoaa muun muassa geneeriset määritelmät Request- ja Response-objekteille [7]. Request ja Response ovat Fetch API:n rajapintoja ja näitä objekteja voidaan pitää palvelimelle lähtevän pyynnön (Request) ja sieltä tulevan vastauksen (Response) kuvastajina [9; 10]. Request- ja Response-objektit mahdollistavat pyyntöjen ja vastauksien käytön ohjelmassa myös tulevaisuudessa. [7.]

Esimerkkikoodissa 1 nähdään, kuinka Fetch API:n resurssien noutamiseen tarkoitettu fetch-metodi tarvitsee vain yhden pakollisen argumentin, joka on osoite haluttuun resurssiin. Se palauttaa Promise-objektin, joka päättyy (resolve) Response-objektiin kyseiselle pyynnölle, oli se sitten onnistunut tai ei. Kun Response-objekti on saatu, niin then-metodissa sitä voitaisiin käsitellä halutulla tavalla ennen sen palauttamista.

```
fetch('http://example.com/movies')
  .then((response) => {
    return response;
  })
```

Esimerkkikoodi 1. Havainnollistava esimerkki fetch-metodin käytöstä resurssien hakemiseen [7].

### 2.2.2 Promise

Promise-objekti kuvastaa asynkronoidun operaation lopullista valmistumista tai epäonnistumista. Se on palautettu objekti, joka liitetään callback-funktioihin [11]. Callback-funktio on funktio, joka annetaan toiselle funktiolle parametrina ja sitä kutsutaan sen funktion sisällä jonkin prosessin viimeistelyksi [12].

Promise toimii proxyna jollekin arvolle, jota ei vielä välttämättä ole tiedossa promisen luonnin yhteydessä. Se antaa mahdollisuuden liittää käsittelijät eli callback-funktiot asynkronoidun toiminnan lopputuloksen käsittelyyn. Tämä mahdollistaa asynkronoitujen metodien palauttaa arvoja kuten synkronoidut metodit. Asynkronoidut metodit eivät näin ollen palauta heti lopputulemana olevaa haluttua arvoa, vaan ne palauttavat Promisen tarjoaman kyseisen arvon jossain kohtaan tulevaisuudessa. [11.]

Promisella on kolme tilaa, jossa se voi olla. Ne ovat

- pending: alkutila. Promisea ei ole täytetty tai hylätty
- fulfilled: merkki operaation onnistumisesta
- rejected: merkki operaation epäonnistumisesta.

Pending-tilassa oleva Promise voidaan siirtää tilaan fulfilled jollain annetulla arvolla tai hylätä syyllä (error). Kun kumpi tahansa tilasta toteutuu, liitetyt käsittelijät promisen then-metodissa kutsutaan riippuen Promisen tilasta. Kehittäjä voi hylätä ohjelmassaan Promisen reject-metodilla tai merkitä operaation onnistuneeksi resolve-metodilla. [11.]

Yleinen tarve on toteuttaa kaksi tai useampi asynkronoitu operaatio peräkkäin, missä toinen operaatio alkaa vasta, kun toinen operaatio valmistuu saaden tiedon aikaisemman operaation tuloksesta. Tämä onnistuu kehittämällä Promise-ketju. [11.]

Esimerkkikoodissa 2 nähdään, kuinka asynkronoituja operaatioita voidaan ketjuttaa Promisen avulla. Fetch-metodilla tehdään palvelinpyyntö, joka palauttaa Promisen, joka päätyy Response-objektiksi. Ensimmäisessä then-metodissa Response-objektin body-sisältö parsitaan json-metodilla, joka palauttaa uuden Promisen päätyen JavaScript-objektiksi. Toisessa then-metodissa parsittu vastaus tulostetaan konsoliin.

```
fetch('http://example.com/movies.json')
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    console.log(data);
  });
```

Esimerkkikoodi 2. Havainnollistava koodi siitä, kuinka promise-ketju luodaan [7].

### 2.2.3 Cache API

Service worker tallentaa palvelimelta tulevia vastauksia selaimen välimuistiin Cache API:a hyödyntäen Request-Response-parina. Service worker on vastuussa siitä, kuinka Cache-objektit päivittyvät ja poistuvat välimuistista. Vastuu välimuistissa olevien alkioiden poistamisella on siis kehittäjällä. Jokaisella selaimella on omat rajansa siitä, kuinka paljon dataa välimuistiin voidaan tallentaa. [13.]

Cache-olion voi noutaa käyttäen globaalia caches-muuttujaa, joka on CacheStorage-rajapinnan ilmentymä [14]. Cache-olio saadaan kutsumalla caches.open()-metodia antamalla parametrina välimuistin nimi. Tämän jälkeen voidaan Cache.match() -metodilla hakea parametreina annetuilla pyynnöllä ja vapaaehtoisilla asetuksilla välimuistissa olevaa alkiota. Match()-metodi palauttaa promisen, joka päätyy Response-objektiksi eli vastaukseksi. Cache.put()-metodilla voidaan tallentaa avattuun välimuistiin uusi alkio. Put-metodi tarvitsee parametriksi avaimen ja arvon, jotka ovat tyypillisesti pyyntö ja vastaus. Cache.delete() -metodilla voidaan halutulla avaimella ja vaihtoehtoisilla asetuksilla poistaa alkio välimuistista. [13.]

## 2.3 Workbox

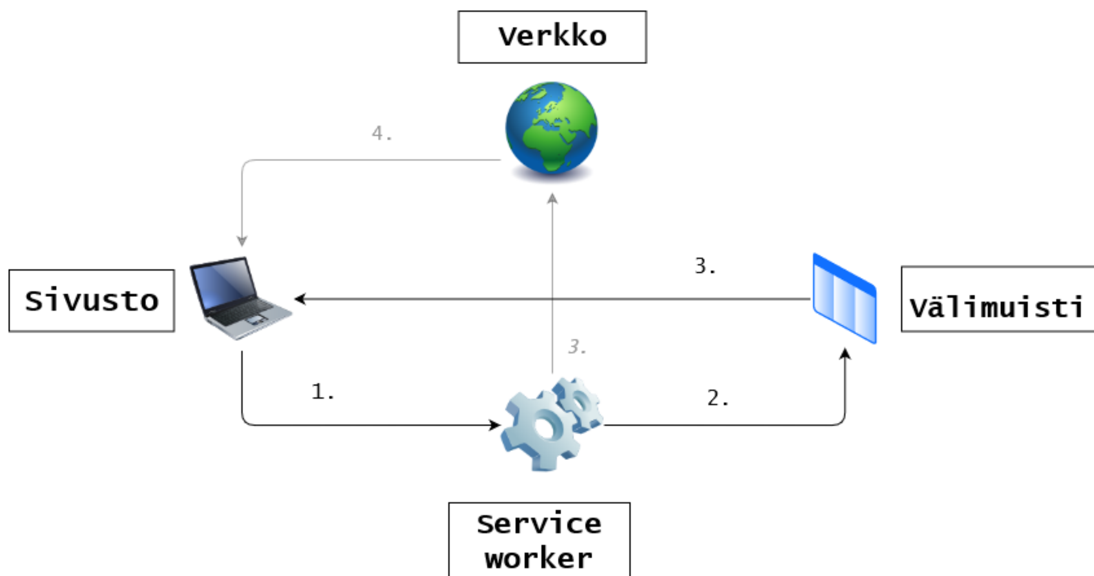
Workbox on kokoelma kirjastoja sekä Node-moduuleja. Se antaa kehittäjälle avaimet parhaimpiin käytäntöihin tuodessaan service workerin web-sivulle. Se on sw-precachen ja sw-toolbox:in jälkeläinen, jota käytetään mm. service workereiden luomiseen, ennalta ja ajonaikaiseen välimuistiin tallentamisen sekä verkkopyyntöjen reitittämiseen. [14; 15.]

Workbox tarjoaa service workerille muun muassa valmiita strategioita verkkopyyntöjen ja välimuistin hallintaan. Välimuistin hallitsemiseen käytettäviä strategioita pystyy konfiguroimaan useilla erilaisilla plugineilla, joilla voi määritellä esimerkiksi välimuistin eliniän, alkioiden maksimimäärän välimuistissa tai callback-funktion onnistuneen ja/tai epäonnistuneen palvelinpyynnön jälkeen. [16; 17.]

Eri pyynnöt palvelimelle vaativat erilaisia tapoja niiden käsittelyyn. Esimerkiksi staattiset tyylitiedostot kannattaa hakea aina välimuistista, kun taas erittäin useasti päivittyvä sisältö suoraan palvelimelta. Seuraavaksi on esitetty eri strategioita, joita workbox tarjoaa kehittäjälle työkaluiksi.

**Cache first**

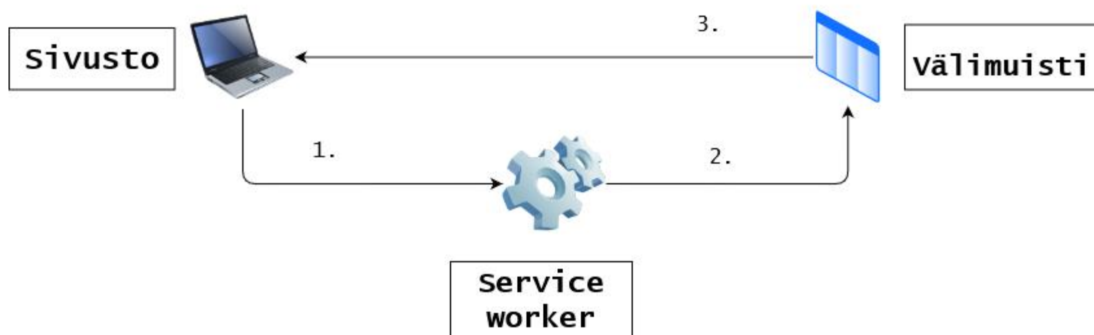
Cache first -strategiassa (kuva 1) service worker pyytää ensin välimuistista vastausta. Jos sitä ei ole välimuistissa, se suorittaa saman pyynnön palvelimelta. Strategia on hyödyllinen pitkään välimuistissa oleville resursseille, kuten tyylitiedostoille [18].




Kuva 1: Cache first-startegia

### Cache only

Cache only -strategiassa (kuva 2) service worker pyytää ainoastaan välimuistista vastausta. Strategiaa on harvoin tarpeellista käyttää, mutta sitä voidaan soveltaa mm. kaikille staattisille tiedostoille, jotka ”versioivat” sivun [19].

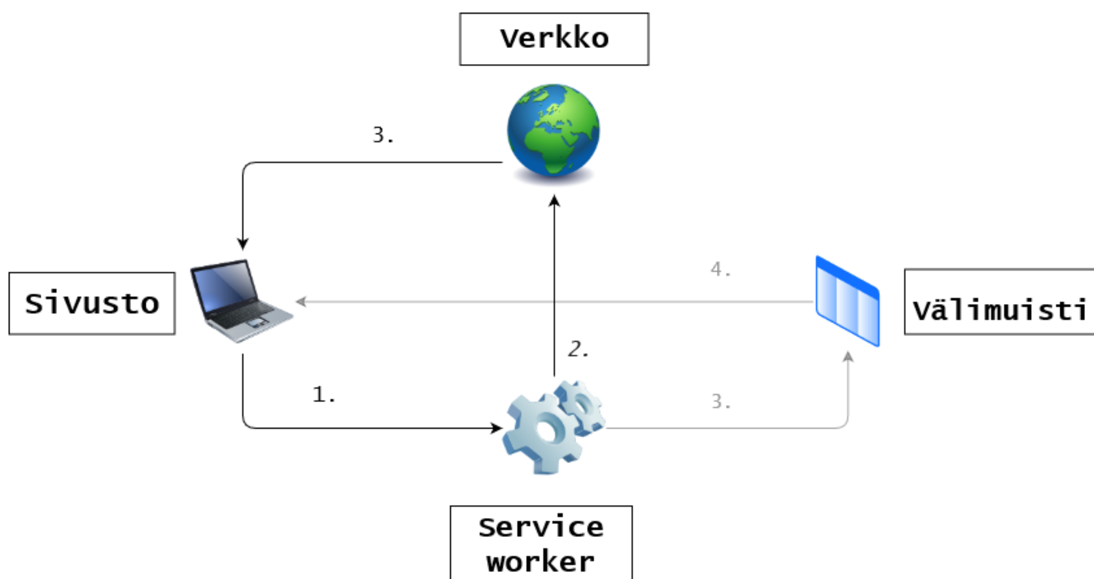



Kuva 2: Cache only -strategia

### Network first

Network first -strategiassa (kuva 3) service worker pyytää ensin vastausta palvelimelta. Jos pyyntö onnistuu, se tallentaa tuoreen vastauksen välimuistiin. Pyynnön epäonnistuttua palvelimelta, service worker suorittaa saman pyynnön välimuistista. Tulee kuitenkin huomioida, että välimuistista tuleva vastaus saattaa sisältää päivittämätöntä tietoa.

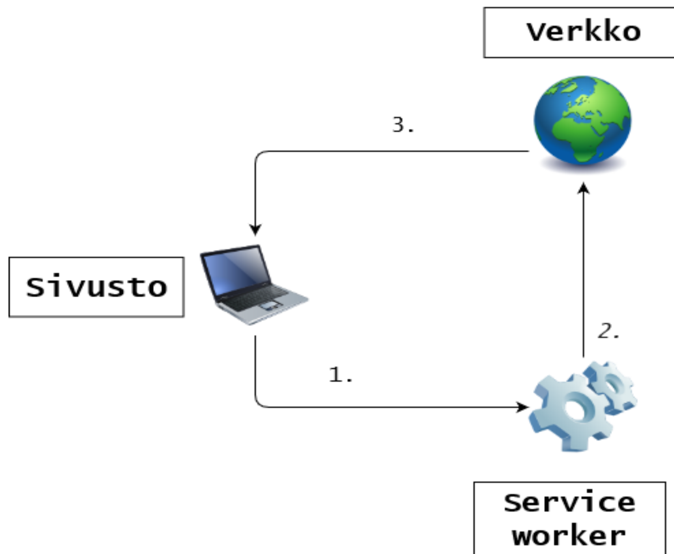
Tämä strategia on kätevä useasti vaihtuvan sisällön hakemiseen, kuten pelien pistetaulut [19].




Kuva 3: Network first -strategia

### Network only

Network only -strategiassa (kuva 4) service worker pyytää vastausta vain palvelimelta, kuten ”normaalisti” tapahtuisi ilman service workeria. Tässä pääsee kuitenkin hyödyntämään workboxin liitännäisiä pyynnön yhteydessä.

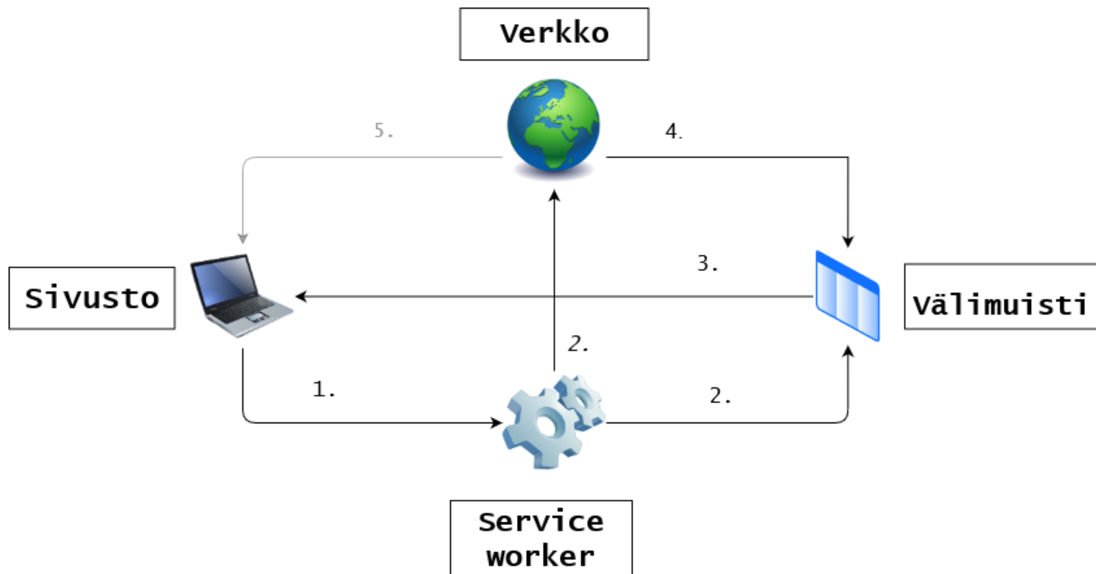



Kuva 4: Network only -strategia

### Stale while revalidate

Stale while revalidate -strategiassa (kuva 5) service worker tekee pyynnön samanaikaisesti välimuistista ja palvelimelta palauttaen kuitenkin ensimmäiseksi välimuistissa olevan vastauksen. Jos välimuistista ei löydy vastausta, odotetaan palvelimelta tulevaa vastausta. Kun palvelimelta tullut vastaus on valmis, se ladataan myös välimuistiin odottamaan seuraavaa pyyntöä. Näin ollen käyttäjä saa aina ”vanhentuneen” vastauksen nopeasti sillä välin, kun uusi vastaus haetaan välimuistiin seuraavaa kertaa varten.

Stale while revalidate on ideaali tapauksissa, jossa haettava sisältö päivittyy useasti, mutta viimeisimmän version saaminen ei ole olennaista [19].




Kuva 5: Stale while revalidate -strategia

Workbox auttaa myös vähentämään koodirivien syntymistä ja lisää koodin luettavuutta. Kun vertaillaan cache first -strategian implementointia workboxilla (esimerkkikoodi 3) ja ilman sitä (esimerkkikoodi 4), huomaamme koodin määrän vähenevän ja luettavuuden paranemisen.

```
workbox.routing.registerRoute(
  new RegExp('/annaKuva/'),
  new workbox.strategies.CacheFirst()
);
```

Esimerkkikoodi 3. Cache first -toteutus workboxilla. ´

```
self.addEventListener('fetch', function(event) {
if(event.request.url.match('/annaKuva/'){
    event.respondWith(
      caches.match(event.request).then(function(response) {
        return response || fetch(event.request);
      })
    );
  }
});
```

Esimerkkikoodi 4. Cache first -toteutus ilman workboxia [19].

## 2.4 Moduuli-suunnittelumalli

Moduuli-suunnittelumalli on yksi yleisemmistä suunnittelumalleista JavaScriptissä. Se on helppo omaksua ja se kapseloi koodin. Moduuli-suunnittelumallin toteutuksia on monia, mutta idea niissä on sama. [20.]

Esimerkkikoodissa 5 nähdään moduuli-suunnittelumallista esimerkkitoteutus. Siinä annetaan muuttujalle anonyymi sulkeuma, joka on periaatteessa funktio. Se käärii koodin ja luo sille suljetun näkyvyysalueen. JavaScriptissä ei ole ”private”-tyyppisiä muuttujia, joten alaviiva nimen edessä on yleinen keino merkata tämän tyyliset muuttujat ja metodit. Paluulauseessa palautetaan objekti, jossa voidaan paljastaa moduulista vain halutut metodit ja muuttujat. [20.]

```
var moduuli = (function() {

  var _privaattiMuuttuja = ”Minuun ei päästä suoraan käsiksi”;
  var julkinenMuuttujas = ”Minuun päästään suoraan käsiksi”;

  function _privaattiMetodi() {
    console.log(_privaattiMuuttuja);
  }

  function julkinenMetodi() {
    _privaattiMetodi();
  }

  return {
    julkinenMetodi: julkinenMetodi,
    julkinenMuuttuja: julkinenMuuttuja
  };
})();

moduuli.julkinenMetodi (); // Minuun ei päästä suoraan käsiksi
console.log(moduuli.julkinenMuuttuja); // Minuun päästään suoraan käsiksi
console.log(moduuli._privaattiMuuttuja); // undefined
moduuli._privaattiMetodi(); // TypeError, _privaattiMetodi on kapsuloitu
```

Esimerkkikoodi 5. Moduuli-suunnittelumallista esimerkkitoteutus [20].

## 2.5 PDM

Product Data Management (PDM) tarkoittaa jonkin sovelluksen tai muun työkalun käyttöä datan seuraamiseen ja hallitsemiseen liittyen johonkin tiettyyn tuotteeseen. Hallittu

data liittyy yleensä tuotteen tekniseen spesifikaatioon, sen valmistamiseen ja tuottamiseen [21]. Esimerkiksi jos tuotteena olisi jugurtti, voitaisiin hallita mm. tietoa reseptistä, reseptin ainesosista, mitä ainesosa sisältää ja missä tehtaassa jugurtti valmistetaan. PDM on osa Product Lifecycle Management:ia (PLM) eli tuotteen elinkaaren hallintaa. [21.]

### 2.6 3DExperience

3DExperience on alusta, jonka on kehittänyt Dassault Systemes. Se on yhdistelmä heidän kehittämiään erilaisia applikaatioita, joilla luodaan ympäristö tuotteen elinkaaren hallintaa varten [22]. Asiakasyritys X:n PDM-järjestelmä on rakennettu 3DExperience-alustalle.

### 2.7 TVC

TVC-tuoteperhe on TECHNIA:n kehittämiä komponentteja ja TVC-lyhenne tulee sanoista TECHNIA Value Components. Näillä komponenteilla parannetaan 3DExperience-alustan käyttökokemusta, järjestelmän suorituskykyä ja omaksumisastetta [22]. Käytännössä 3DExperience-alusta tuo valmiin tietomallin asiakkaan käyttöön ja TVC:n komponenteilla tehdään siihen asiakaskohtaisia konfiguraatioita.

TVC Helium on nykyaikainen progressiivinen web-sovellus (PWA), joka edustaa TVC-tuoteperheen uutta sukupolvea. Helium perustuu pitkälti avoimen lähdekoodin kirjastoihin. Se on siis yhdistelmä jo olemassa olevia ratkaisuja, jotka on sulautettu toimimaan yhdessä toisiaan hyödyntäen.

Heliumiin on rakennettu offline-ominaisuus hyödyntäen service worker- ja workbox-kirjastoja. Sen tarkoitus on mahdollistaa sovelluksen käyttö yhteydettömässä tilassa ja viedä Heliumia askeleen lähemmäksi aitoa progressiivista web-applikaatiota.

### 2.8 PWA

Progressive Web Application (PWA) eli progressiivinen web-sovellus on moderneilla teknologioilla kehitetty web-sovellus, joka on muun muassa responsiivinen, yhteydestä riippumaton, HTTPS:llä toimiva ja sovellusmainen [23]. PWA:t ovat käytettävissä laitteesta riippumatta, mutta ne nojautuvat vain yhteen koodikantaan ja käyttäjällä on aina käytössä uusin versio [24].

## 3 Toimeksianto/projekti

### 3.1 Tarve

Asiakkaan käytössä olevaa, TVC Heliumia hyödyntävää PDM-järjestelmää käytetään kansainvälisesti ympäri maailmaa heidän tehtaillaan niin tietokoneilla kuin tableteilla. Ongelmana on kuitenkin tehtaiden paikoittainen heikko ja katkeileva internetyhteys. Maailmalla matkustavat myyjät tarvitsevat myös ajoittain pääsyä järjestelmään ilman internet yhteyttä.

### 3.2 Taustaa

TVC Heliumin offline-ominaisuutta on tuotekehityksessä aktiivisesti suunniteltu ja rakennettu osaksi tuotetta. Vuoden 2019 alusta kehitys oli tullut siihen pisteeseen, että se oli valmiina toteutettavaksi tuotantokäyttöön. Tämä insinöörityö on ensimmäinen toteutus tuotteemme historiassa, jossa offline-ominaisuus toteutetaan ja viedään tuotantoon asiakkaalle.

### 3.3 Ohjelmistotuotannon vaiheet

Tämä insinöörityö pitää sisällään suunnittelu- ja toteutusvaiheen. Määrittelyvaihe on jätetty pois, koska TVC Heliumin tuotekehitys on jo päättänyt lähestymistavan offline-omi-

naisuuden toteuttamiseen. Myös testaus ja käyttöönotto jätetään insinöörityön ulkopuolelle, koska ne eivät kuuluneet aktiivisesti työtehtäviini projektia koskien. Käyttöönotto tapahtui osana muuta päivitystä ohjelmistoon.

### 3.4 Vaatimukset

Asiakas ei antanut kovin tarkkoja vaatimuksia siitä, miten järjestelmän tulisi toimia yhteydettömässä tilassa. He halusivat saada samalla kertaa myös pienennystä latausaikoihin hyödyntämällä välimuistia. Projektipäällikön ja asiakkaan kanssa käydyn keskustelun pohjalta eristettiin neljä ylätason vaatimusta:

- Käyttäjän pitää yhteydettömässä tilassa saada näkyviin ne sivut, joissa hän on vieraillut viimeisen kahden viikon aikana kyseisellä laitteella.

- Käyttäjän pitää saada raskaimmatkin sivut auki ilman kohtuutonta latausaikaa.

- Käyttäjälle hänen yhteyden tilan näyttäminen käyttöliittymässä.

- Virheenhallinta; Jos käyttäjä yrittää navigoida saavuttamattomalle sivulle tai välilehdelle, käyttäjän pitää saada siitä ilmoitus.

Nämä vaatimukset toimivat runkona suunnittelussa ja toteutuksessa.

## 4 Suunnittelu ja toteutus

Suunnittelu toteutetaan ylätasolla suunnitelmanomaisesti piirtämällä kuvaa lähestymistavasta toteutukseen. Tässä insinöörityössä on tarkoitus antaa suunnittelulla suunta toteutukselle eikä toimia teknisenä dokumentaationa.

Toteutus ja suunnittelu jaettiin alkuvaiheessa neljään osaan vaatimuksien pohjalta:

- Offline-ominaisuuden aktivointi ja oman service workerin liittäminen osaksi ohjelmistoa.

- Verkkopyyntöjen ja omien resurssien välimuistiin tallentaminen ja välimuistista tarjoilu.

- Yhteyden tilan näyttäminen käyttäjälle selaimessa.

- Käyttäjälle ilmoittaminen, kun hän yrittää yhteydettömässä tilassa vierailla sivulla, joka ei ole välimuistissa.

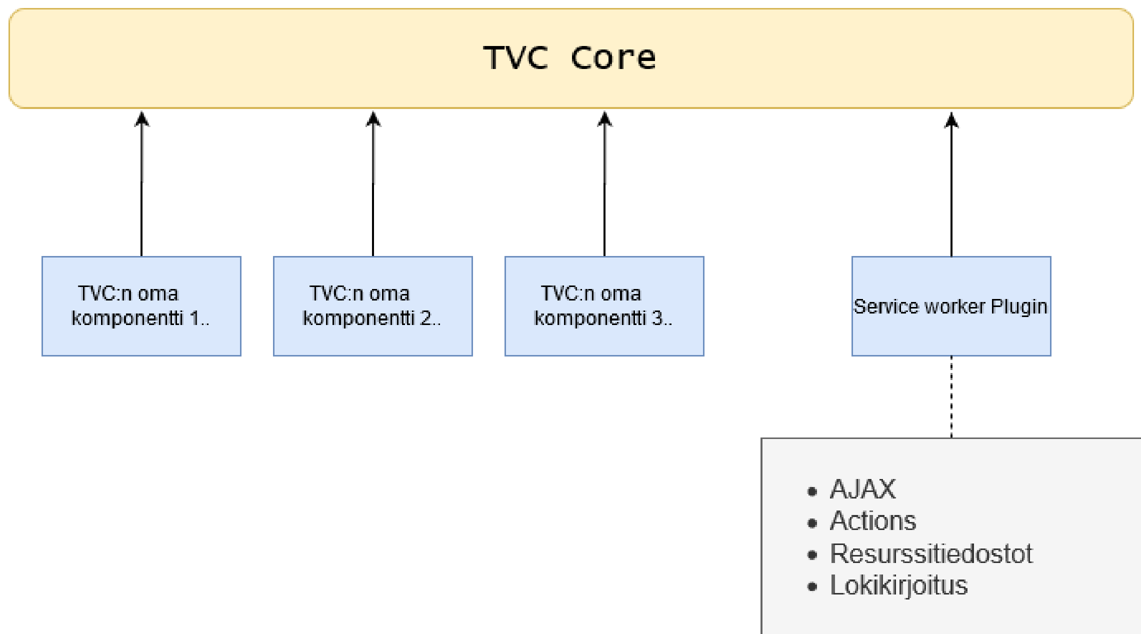
Heliumin tuotekehittäjät ovat luoneet pohjan offline-ominaisuudelle lähdekoodiin, jotta se olisi helposti toteutettavissa eri asiakasjärjestelmiin luomalla vain oma järjestelmäkohtainen service worker. Lähdekoodiin on tehty Heliumin oma service worker ja toteutus service workerien asennukseen. Heliumin service worker hoitaa resurssitiedostojen, kuten CSS-tiedostojen ja JavaScript-tiedostojen välimuistiin tallentamisen. Oma service worker ja muu tarvittava koodi tullaan tekemään Heliumin sovelluskehittäjille varaamaan hakemistoon, jossa ne ovat erossa lähdekoodista.

## 4.1 Offline-ominaisuuden aktivointi ja oman service workerin liittäminen osaksi ohjelmistoa

### 4.1.1 Suunnittelu

Toteutus offline-ominaisuuden aktivointiin ja oman service workerin liittämiseen ohjelmistoon suunniteltiin tuotekehityksen toimesta, joten se ei vaatinut tässä työssä erityistä suunnittelua. Tuotekehitys ohjeisti tarvittavat konfiguraatiot service workerin asennukseen sekä staattisten ja dynaamisten resurssien välimuistiin tallentamisen aktivoimiseen. Tarkoituksena on luoda TVC Plugin liittämään service worker osaksi ohjelmaa ja service worker JavaScript-tiedosto.

TVC Plugin on keino laajentaa TVC:n ohjelmistoa (kuva 6) luomalla oma komponentti muiden komponenttien rinnalle ohjelmistoon. Se on pieni Java-luokka, joka sisältää metadataa omaan pluginiin liittyen. Se ladataan määrittelyn mukaan, kun TVC alustetaan joko sovelluspalvelimen käynnistyessä tai kun TVC käynnistetään ensimmäistä kertaa. Liitännäiselle voidaan määrittää muun muassa oma lokiin kirjoitus, kontrolleri, lokalisaatiotiedostot ja Ajax-palvelut.




**Kuva 6: TVC:n rakennetta ja siihen liitetty oma plugin.**

TVC:n ohjelmisto noudattaa model-view-controller-arkkitehtuuria (MVC). MVC-arkkitehtuurissa kontrolleri (controller) toimii käyttöliittymästä (view) tulevien pyyntöjen käsittelijänä. Kontrolleri käsittelee pyynnön ja muokkaa mallia (model), jossa myös toteutetaan bisneslogiikka ja välitetään malli takaisin näkymäkerrokseen. [25.]

TVC Helium tekee palvelupyynnöt käyttöliittymästä JavaScript-koodilla, jonka ohjelmiston backend-kielenä toimivalla Javalla kirjoitettu kontrolleri käsittelee ja palauttaa tarvittaessa takaisin näkymäkerrokseen JSON-muodossa olevaa tietoa tietokannasta. JSON-muotoista dataa saatetaan myös prosessoida selaimessa lisää, mikä vaikuttaa sen näyttäytymiseen käyttöliittymässä.

Oman service workerin TVC:n ohjelmistoon liittävän TVC Pluginin tärkein merkitys työtä varten on kertoa tieto service workerin JavaScript-tiedoston sijainnista palvelimella. Tarvetta omille asetuksille ei syntynyt suunnitteluvaiheessa, joten pelkästään service workerin sijainnin määrittäminen on tarpeen. Myöskään muita TVC Pluginin mahdollistamia ominaisuuksia ei nähdä tarpeelliseksi käyttää.

Service worker tulee noudattamaan JavaScriptin moduuli-suunnittelumallia kapseloiden muuttujia ja metodeja. Oma service worker rekisteröidään kutsumalla lähdekoodin tarjoamaa rajapintaa, johon välitetään parametrina service workerin moduuli muuttujassa. Tuotekehitys on määritellyt rajapinnanomaisesti neljä metodia rekisteröintiä varten, jotka service workerin tulee toteuttaa. Nämä rekisteröintiin käytettävät funktiot esitetään seuraavaksi.

**init (searchParams)**
Funktio saa parametrina service workereille määritetyt asetukset ja on vastuussa service workerin alustamisesta.

**getValidCacheNames**
Funktion pitää palauttaa Array-muodossa service workerissa määritetyt välimuistien nimet.

**preCache (baseUrl, wokrbox)**
Funktio saa parametreina vakiona pysyvän osan sovelluksen osoitteesta (baseUrl) ja workbox-objektin. Se on vastuussa haluttujen tiedostojen ennalta välimuistiin tallentamisesta workboxin avulla service workerin asennuksen yhteydessä.

**registerRoutes (workbox)**
Funktio saa parametrina workbox-objektin. Se on vastuussa reittien asettamisesta workboxiin ja välimuististrategioiden määrittämisessä.

### 4.1.2 Toteutus

Offline-ominaisuuden aktivoinnissa annetaan Heliumille lupa asentaa service worker ja tallentaa välimuistiin staattisia sekä dynaamisia resursseja konfiguroimalla ohjelmistoa. Olennaista on myös siirtää tiedon käsittely tapahtuvaksi suurimmaksi osaksi selaimessa eikä palvelimella, koska yhteydettömässä tilassa palvelimelle ei ole pääsyä. Nämä konfiguraatiot ovat ainoastaan Heliumiin liittyviä, eikä niiden tarkempi läpikäyminen ole olennaista insinöörityön kannalta.

Ohjelmiston pitää saada tieto service workerin sijainnista palvelimella ja sille määritetyistä asetuksista. Tämän takia luodaan TVC:n tarjoama SettingsProvider-rajapintaa noudattava Java-luokka (esimerkkikoodi 6), jossa määritetään asetukset sekä sijainti service workerille. SettingsProvider-rajapinnan provideSettings-metodissa saadaan parametrina HttpServletRequest-olio, joka tarjoaa tietoa Heliumin lähdekoodissa tapahtuvasta asetuksien hakemisen palvelinpyynnöstä. Metodi palauttaa asetukset Map-tietorakenteena.

```
public class ServiceWorkerSettingsProvider implements SettingsProvider {

    private final String SW_PATH = “custom/sw-moduuli.js”;
    private final String PREFIX = “mymodule”;
    @Override
    public String getPrefix() {
        return PREFIX;
    }


    @Override
    public Map<String, String> provideSettings(HttpServletRequest request) {
        Map<String, String> settings = new TreeMap<>();
         Setting.add(“IsDynamicCacheEnabled”,”true”);
         Setting.add(“IsStaticCacheEnabled”,”true”);

        return settings;
    }


    @Override
    public String getServiceWorkerJsPath() {
        return SW_PATH;
    }
}
```

Esimerkkikoodi 6. Havainnollistava koodi Java-luokasta, jossa määritetään service workerin asetukset.

Service worker saadaan liitettyä ohjelmistoon mukaan TVC Pluginilla. TVC Plugin on pieni Java-luokka, joka tarjoaa metadataa omaan liitettävään komponenttiin liittyen. Pluginin rekisteröidyttyä osaksi TVC:n ohjelmistoa sen init-funktiota kutsutaan ohjelmiston käynnistyksen yhteydessä. Plugin rekisteröidään konfiguroimalla.

Omalle service workerille tarkoitetun pluginin init-funktiossa rekisteröidään aikaisemmin luotu SettingsProvider-rajapintaa noudattava luokka (esimerkkikoodi 7), jolloin Helium tulee saamaan tiedon service workerin JavaScript-tiedoston sijainnista palvelimella.

```
public class ServiceWorkerPlugin extends TVCPluginAdapter {

    private final String PLUGIN_NAME = "Custom Service Worker Plugin";
    @Override
    public void init() {
        SettingsFactory.getInstance().
        registerProvider(new ServiceWorkerSettingsProvider());
    }

    @Override
    public String getName() {
        return PLUGIN_NAME;
    }
}
```

Esimerkkikoodi 7. Havainnollistava koodi luodusta TVC Pluginista.

Alkuvaiheessa on tarpeellista ainoastaan luoda JavaScript-tiedostoon raakaversio (Esimerkkikoodi 8) service workerista, joka tarjoaa vain tyhjät toteutukset tuotekehityksen rajapintamaisesti määrittelemiin funktioihin. Service workerin koodi noudattaa moduulisuunnittelumallia, jossa annetaan pääsy moduulin ulkopuolelta vain init-, precache-, registerRoutes- ja getValidCacheNames-funktioihin.

```
var SWmodule = (function() {

    var init = function(searchParams) {};

    var getValidCacheNames = function() {
        return [];
    };

    var precache = function(baseUrl, workbox{};

    var registerRoutes = function(workbox) {};

    return {
        init: init,
        precache: precache,
        registerRoutes: registerRoutes,
        getValidCacheNames: getValidCacheNames
    };
})();

// SWModule:n rekisteröinti
if (typeof registerSubServiceWorker !== 'undefined') {
    registerSubServiceWorker(SWmodule);
}
```

Esimerkkikoodi 8. Service workerin moduulin raakaversio ilman konkreettisia toteutuksia

Service worker on nyt valmiina asennettavaksi selaimeen, vaikka se ei sisällä mitään toiminnallisuutta.

## 4.2 Resurssien tallentaminen välimuistiin ja välimuistista noutaminen

### 4.2.1 Suunnittelu

Jotta yhteydettömässä tilassa navigointi olisi mahdollista, dynaamiset ja staattiset resurssit täytyy saada tallennettua käyttäjän selaimen välimuistiin. Dynaamisilla resursseilla tarkoitetaan palvelimella muodostuvia vastauksia ja staattisilla resursseilla jo olemassa olevia tiedostoja.

Heliumin oma service worker on vastuussa staattisten resurssien sekä Heliumin joidenkin ydintoimintoihin liittyvien asioiden välimuistiin tallentamisesta ja hallinnasta. Insinöörityössä tehdylle service workerille jää vastuu käyttäjän navigoinnista aiheutuvien palvelinpyyntöjen reitityksestä ja reittikohtaisesta välimuistin hallinnasta.

Reitittämällä luodaan omat reitit eri palvelinpyynnöille, jotta niihin pystytään pyyntökohtaisesti toteuttamaan erilaisia strategioita välimuistinhallinnan sekä muun toimintalogiikan osalta. Reittien rekisteröinti tehdään registerRoutes-funktion parametrina saatuun workbox-moduuliin.

Resurssit tallentuvat välimuistiin avain-arvo-pareina (taulukko 1), jossa avain on pyyntö (Request) ja arvo vastaus (Response). Lähdekoodissa tapahtuvan käyttöliittymän elementtien luonnin yhteydessä saatetaan tehdä myös muita palvelinpyyntöjä, jotka pitää tallentaa välimuistiin.

**Taulukko 1: Välimuistin avain-arvo -pareihin pohjautuva tietorakenne.**

| Avain (Request) | Arvo (Response) |
|---|---|
| **https//:www.AsiakasyritysX.fi/actions/createDashboard/objectId1** | {kojelaudan määritelmä...} |
| **https//:www.AsiakasyritysX.fi/actions/createDashboard/objectId2** | {kojelaudan määritelmä...} |

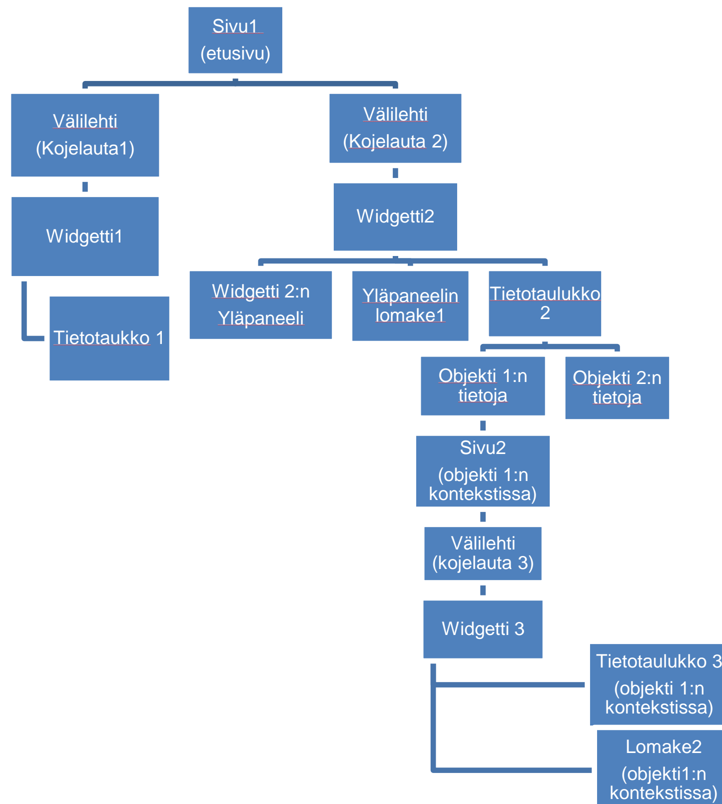
Välimuistit jaetaan eri kategorioihin, jotta eri reitit saadaan hahmoteltua. Kategorioiden suunnittelussa yhdistettiin samankaltaiset pyynnöt tai yhdestä pyynnöstä tapahtuvien pyyntöjen ketju yhteen kategoriaan.

Välimuistin kategoriat ja reitit ovat seuraavat:

- välilehden/sivun vaihtaminen tai avaaminen
- tietotaulukon muodostaminen
- tietotaulukon toiminnot
- lomakkeiden avaus ja alustus
- TVC Wikiin liittyvät toiminnot.

Jakaminen ohjelmassa tapahtuu määrittelemällä reittikohtaisesti vakiomuuttujat, jotka pitävät merkkijonona säännöllisen lauseen (RegExp). RegExp-lauseet kuvastavat palvelinpyynnön URL-osoitetta, joka reitin pitää käsitellä. Vakiomuuttujia tullaan käyttämään reitin workboxin rekisteröinnin yhteydessä.

Asiakkaan järjestelmän sivurakenne on erittäin yksinkertainen (kuva 7), missä navigoidaan eri välilehdillä ja sivuilla. Sivuilla ja välilehdillä näytetään tuotetietoja tai tuotteeseen liittyviä tietoja tietotaulukoilla, lomakkeilla, TVC Wikillä ja dokumenteilla. Dokumentit ovat tiedostopalvelimella eikä niitä saada tallennettua välimuistiin. TVC Wiki on TVC:n oma komponentti, jossa voidaan ylläpitää hieman vapaamuotoisemmin wikipedian omaisesti esimerkiksi tuotteen valmistamiseen liittyvää tietoa.




**Kuva 7: Asiakkaan sivuston tyypillinen rakenne havainnollistettuna.**

Tietotaulukoissa (kuva 8) listataan yleensä 3DExperiencen tietomallin mukaisesti objekteja, jotka kuvastavat tosimaailman asioita esimerkiksi tuotetta tai tuotteen osaa.

EBOM | Specifications | Reference Documents

Indented | Flat | Tiled

FIXED HEADER SINGLELEVELEBOM 19

Filter | Columns... | Print | Promote | Export | Add Part(s) | Remove Part | Quick View2 | Views

| TYPE | NAME | REV | STATE | IMAGE | DESCRIPTION | MATERIAL CATEGORY | WEIGHT | ACTIONS | DROP ZONE | EFFECTIVITY DATE |
|---|---|---|---|---|---|---|---|---|---|---|
| Part | A-0000355-01 | A | Create | | | Metal | 0.0 | | Drop Zone | |
| Part | A-0000386-01 | A | Create | | This is test | Metal | 120.0 | | Drop Zone | |
| Part | A-0000412-01 | A | Create | | | Glass | 0.0 | | Drop Zone | |
| Part | A-0000413-01 | A | Create | | | Metal | 0.0 | | Drop Zone | |
| Part | A-0000414-01 | A | Create | | | Metal | 0.0 | | Drop Zone | |

10 | Showing 1 to 10 of 19 entries | Previous 1 2 Next

**Kuva 8: TVC Heliumin tietotaulukko**

Koska asiakas haluaa lyhentää latausaikoja, tietotaulukoiden muodostavien verkkopyyntöjen reitin käsittelyyn käytetään stale while revalidate -strategiaa. Tällöin käyttäjä saa aina nopeasti vastauksen välimuistista sen ollessa siellä. Myös hidas yhteys ei näin vaikuta latausaikaan. Tietotaulukot sisältävät paikoittain runsaasti dataa ja ovat asiakkaalla suurimmillaan 6 megatavun kokoisia. Niiden sisältö muuttuu kuitenkin harvakseltaan.

Lomakkeiden, TVC Wikin, tietotaulukon toimintojen ja sivujen/väliehtien reittien välimuistien strategiaksi käytetään network first -strategiaa, koska palvelimelta saadut vastaukset ovat suuruudeltaan pieniä eivätkä ne skaalaudu huomattavasti suuremmiksi. Näin käyttäjä näkee verkossa ollessaan tuoreimman version datasta esimerkiksi lomakkeessa, jossa on tietoja tuotantokomponentteihin liittyen. Välimuistit pidetään kuitenkin erillään, jos tulevaisuudessa on tarve muuttaa niiden käyttäytymistä kategoriakohtaisesti.

Ohjelmistossa tapahtuvia palvelinpyyntöjä tutkittaessa todettiin tarpeelliseksi laittaa välimuistiin vain pyynnöt, joiden HTTP-statuskoodi on 200. Kyseinen koodi merkitsee onnistunutta pyyntöä palvelimelta ilman mitään metatietoja [26]. Tätä varten Workbox tarjoaa helppokäyttöisen CacheableResponsePlugin-rajapinnan määrittämään välimuistiin tallentamisen haluttujen statuskoodien mukaan.

Selain pystyy tarjoamaan vain rajoitetun määrän tilaa välimuistin käyttöön. Tästä syystä on tarpeellista huolehtia välimuistissa olevien vanhentuneiden alkioiden poistamisesta.

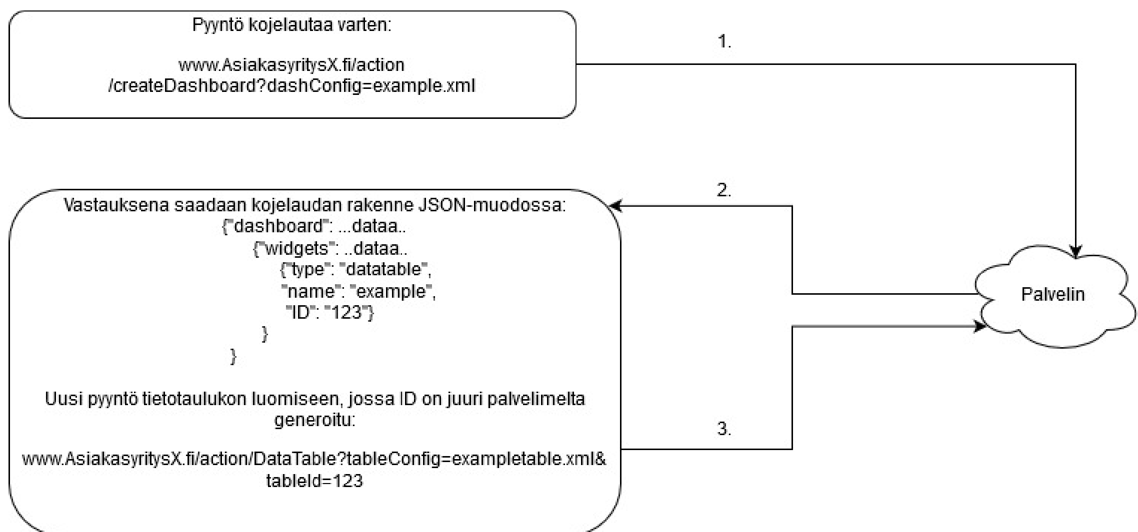
Välimuistin alkioiden eliniän hallitsemiseen käytetään Workboxin tarjoamaa ExpirationPlugin-rajapintaa, johon määritetään välimuistin elinikä sekunneissa. ExpirationPlugin lisätään reitille reitin rekisteröinnin yhteydessä. Kyseinen liitännäinen tarkistaa reittiin kuuluvan välimuistin avaamisen jälkeen alkioiden iän ja poistaa vanhentuneet alkiot CacheStorage-rajapintaa hyödyntäen.

### 4.2.2 Ongelmat

Toteutusta tehtäessä huomattiin ongelma tietotaulukoiden välimuistin käsittelyn stale while revalidate -strategiassa. Saman tietotaulukon palvelinpyynnön URL-osoite muuttuu joka pyynnöllä, jos käyttäjällä on toimiva yhteys palvelimeen. Tämä luo tilanteen,

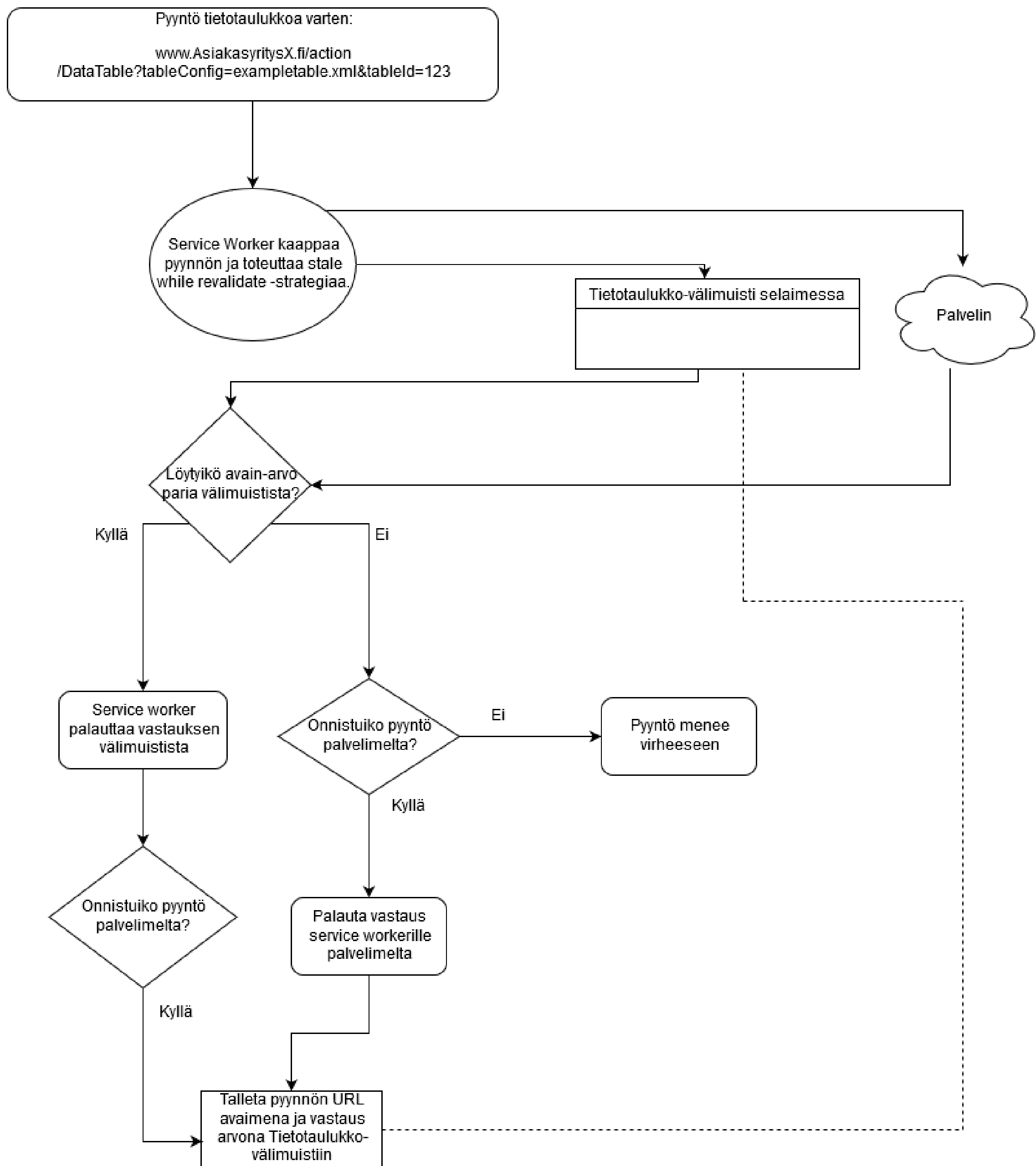
jossa joudutaan aina odottamaan palvelimelta vastausta, koska palvelinpyynnön URL-osoitetta vastaavaa alkiota ei voi löytyä välimuistista.

Tietomallissa kojelauta vastaa yleisesti ottaen välilehteä. Kojelauta koostuu muun muassa tietotaulukoista. Kun käyttäjä klikkaa välilehteä tehdäkseen pyynnön palvelimelle, palvelin palauttaa kojelaudan ja sille juuri palvelimelta luodun ID:n (kuva 9). Seuraavaksi ohjelmiston muodostaessa kojelautaa, se pyytää tietototaulukkoa palvelimelta välittäen URL-parametrinä kojelaudan ID:n. Vastauksena saadaan tietotaulukon muodostamiseen lähdekoodissa tarvittava data JSON-formaatissa, jossa tietotaulukko yksilöidään samalla ID:llä kuin kojelauta.




Kuva 9: Tietotaulukon pyynnön syntyminen yhteydellisessä tilassa.

Tietotaulukon toteuttama stale while revalidate -strategia (kuva 10) ei näin ollen voi toteutua, koska välimuistissa ei ole koskaan avain-arvo-paria tietotaulukon pyynnölle.




**Kuva 10: Tietotaulukoille tarkoitettu alkuperäinen stale while revalidate -strategia.**

Yhteydettömässä tilassa kojelauta tulee välimuistista, jolloin tietotaulukon pyyntöön käytetty ID pysyy aina samana ja on jo viety välimuistiin ensimmäisessä pyynnössä.

Ongelma saadaan ratkaistua luomalla oma logiikka siihen, mitä välimuistissa olevaa avainta pitää käyttää. Tässä onnistutaan tekemällä kustomoitu oma käsittelijä käsittele-

mään reitillä pyynnön ohjaamista ja välimuistin hallitsemista. Oma käsittelijä pääsee käsiksi Cache API:iin, jonka avulla voidaan tarkastella, mitä alkioita selaimen välimuistista löytyy. Sillä pystytään myös noutamaan vastauksia avaimella sekä lisätä ja poistaa alkioita.

Oman käsittelijän tulee palauttaa Promise (esimerkkikoodi 9), joka päätyy vastaukseksi (Response). Parametreinä käsittelijä saa pyynnössä käytetyn URL-osoitteen, FetchEvent-objektin ja parametrejä objektissa. Parametrit lisätään halutessa reitityksestä vastaavassa kustomoidussa käsittelijässä. [27.]

```
const handlerCb = ({url, event, params}) => {
  return fetch(event.request)
  .then((response) => {
    return response.text();
  })
  .then((responseBody) => {
    return new Response(`${responseBody}`);
  });
};
```

Esimerkkikoodi 9. Havainnollistava esimerkki kustomoidusta käsittelijästä [27].

Tietotaulukon palvelinpyynnöstä tarvitaan yksilöiviä tekijöitä, jotta oikea avain löydetään. Tietotaulukon ID täytyy sivuuttaa täysin, jotta stale while revalidate -strategia saataisiin toteutettua.

Tietotaulukon pyyntö koostuu karkeasti:

1. Kiinteästä osasta: ”**https://www.AsiakasyritysX.fi/actions**”
2. Mitä pyydetään: ”https://www.AsiakasyritysX.fi/actions/**dataTable**”
3. Mitä XML:ssä määritettyä tietotaulukon määritystä käytetään: ”https://www.AsiakasyritysX.fi/actions/dataTable?**tableconfig=example.xml**”
4. Jos tietotaulukko tulee tietyn objektin kontekstissa, niin sen yksilöivä ID välitetään mukaan: ”https://www.AsiakasyritysX.fi/actions/dataTable?tableconfig=example.xml&**objectID=1234.5678.9123**”

5. ID, jolla yhdistetään tietotaulu palvelimella ja UI:ssa: ”https://www.AsiakasyritysX.fi/actions/dataTable?tableconfig=example.xml&objectId=1234.5678.9123&**tableid=12345**”

Tietotaulukon määrittelevä XML-tiedosto palvelimella on jokaisella tietotaulukolla uniikki, ja se toimii ns. piirustuksena tietotaulukolle ja sen toiminnallisuudelle. Tämä toimii siis yksilöivänä tekijänä, jolla pystytään tunnistamaan haluttu tietotaulukko. Jos tietotaulukko pitää näyttää kontekstissa, niin objektin ID toimii yksilöivänä tekijänä kontekstin määrittelyssä.

Oman käsittelijän täytyy siis verrata palvelinpyynnön ja välimuistin alkioiden avaimien palvelinpyyntöjen tableConfig- ja objectId-parametria keskenään löytääkseen oikean avaimen. Oikean avaimen löydyttyä sen arvo-parina oleva vastaus noudetaan välimuistista.

Tietotaulukon ID:n vaatimia toimintoja ei käytetä kuin yhdessä tietotaulukossa koko järjestelmässä ja se tullaan mustalistaamaan pois käsittelijässä stale while revalidate -strategiasta.

### 4.2.3 Toteutus

Jotta erilaisia resursseja voidaan tallentaa välimuistiin ja noutaa sieltä, täytyy toteuttaa niistä vastuussa olevien reittien luominen ja rekisteröinti workboxiin. Näin workbox osaa ohjata palvelinpyynnöt oikealle reitille, joka käsittelee ne.

Ohjelmistossa tapahtuvien palvelinpyyntöjen URL-osoitteita on useita ja erilaisia. Suunnitteluvaiheessa mainittujen reittien/välimuistien kategorioiden osalta selvitettiin palvelinpyyntöjen URL-osoitteet kategoriakohtaisesti. Näiden pohjalta luotiin jokaiselle reitille yksi vakiomuuttuja, joka pitää sisällään RegExp-lauseen. Lauseen tarkoituksena on saada kaikki reittiä koskevat URL-osoitteet palauttamaan true-arvo kyseiselle lauseelle. Näin reitti pystyy tunnistamaan, mikä pyyntö kuuluu sille.

Esimerkkikoodissa 10 nähdään, miltä RegExp-lause näyttää ja kuinka itse työssä reitin määrittävät muuttujat on alustettu.

```
//Yksi tai useampi merkki
const ZERO_OR_MORE_PARAMETERS = '(\\?.*)?';

//Sivun ja välilehden reitti RegExp muodossa
const HELIUM_PAGE_URL_PATTERN =
   '/heliumCore/(start|openObject|entryPage|getDashboard|getToolbar)' +
   ZERO_OR_MORE_PARAMETERS +
    '$';

 //Lomakkeen avaus ja alustamisen reitti RegExp muodossa
 const ON_INIT_REQUESTS_URL_PATTERN =
    '/heliumForm/getForm' +
    ZERO_OR_MORE_PARAMETERS +
    '$';

 //Tietotaulukon reitti RegExp muodossa
 const HELIUM_DATA_TABLE_PATTERN =
    '/heliumDataTable\?' +
    ZERO_OR_MORE_PARAMETERS +
    '$';

 //Tietotaulukon toimintojen reitti RegExp muodossa
 const HELIUM_DATA_TABLE_ACTIONS_PATTERN =
    '/heliumDataTable(?!\\?)';


 // Wiki-ominaisuuden reitti RegExp muodossa.
 const TVC_WIKI_SLIM_PATTERN =
    '/heliumWikiViewSlim' +
    ZERO_OR_MORE_PARAMETERS +
    '$';

 //Staattisten resurssien reitti RegExp muodossa
 const STATIC_RESOURCE_PATTERN =
    '/helium\/custom\/.*\.(png|jpg|gif|jpeg)'
```

Esimerkkikoodi 10. Vakiomuuttujien alustus tulevaa workboxin rekisteröimistä varten.

Esimerkki RegExp-lauseen toiminnasta voidaan antaa HELIUM_PAGE_URL_PATTERN-muuttujan avulla. Muuttujan arvona on ”/heliumCore/(start|openObject|entryPage|getDashboard|getToolbar) (\\?.*)?$”. Paloiteltuna lause tarkoittaa, että siihen verrattavasta tekstistä pitää löytyä jostain kohdasta seuraavaa: /heliumCore/start tai /heliumCore/openObject tai /heliumCore/entryPage tai /heliumCore/getDashboard tai /heliumCore/getToolbar, jonka jälkeen (\\?.*)?$ merkitsee sitä, että sen jälkeen saa olla 0 tai enemmän merkkejä.

Kyseinen RegExp-lause palauttaisi arvon ”true”, jos sitä verrattaisiin seuraavaan URL-osoitteeseen: https://www.asiakasyritysX.fi/actions**/heliumCore/openObject**=234

Seuraava lause palauttaisi arvon ”false”, koska vertailu on merkkikokoriippuvainen: https://www.asiakasyritysX.fi/actions**/heliumCore/gettoolbar**

Välimuistien nimeämisessä käytetään välimuistien kategoriakohtaisesti kuvaavia nimiä englanniksi, joiden loppuun tulee käytettävissä olevan Heliumin versionumero. Nimet on tarkoitus luoda ajonaikana init-funktiossa, jotta Heliumin versionumero saadaan parametrinä tulevista asetuksista. Välimuistin nimet tallennetaan CACHE_NAMES-nimiselle muuttujalle, josta myös funktio getValidCacheNames hakee välimuistin nimet ja palauttaa nimet taulukkona.

Reittien rekisteröinti tehdään registerRoutes-funktion sisälle. Tätä funktiota kutsutaan ylätasolla service workerin rekisteröinnin ja asennuksen yhteydessä. Rekisteröinti jaetaan staattisten ja dynaamisten resurssien reittien rekisteröimiseen. Reitit rekisteröidään vain, jos ohjelmistossa on konfiguroimalla aktivoitu staattisten ja dynaamisten resurssien välimuistiin tallentaminen.

Staattisten ja dynaamisten resurssien välimuistiin tallentamisen aktivoinnin tieto saadaan init-funktiossa parametrina tulleesta objektista, joka pitää sisällään tehtyjä konfiguraatioita. Kyseinen objekti otetaan talteen urlParams-nimiseen muuttujaan, jotta siihen päästään käsiksi reittien rekisteröintivaiheessa.

Tiedot staattisten ja dynaamisten resurssien välimuistien aktivoinneista otetaan talteen registerRoutes-funktion alkuosassa. Asetukset ovat arvoltaan true/false ja ne tallennetaan cacheHeliumDynamic- ja cacheStaticHeliumCustom-vakiomuuttujiin parsimalla ne ensiksi JSON-muodosta (esimerkkikoodi 11).

```
const cacheHeliumDynamic = JSON.parse(
                urlParams.get('he-dynamicCacheEnabled')
            );
        const cacheStaticHeliumCustom = JSON.parse(
                urlParams.get('he-customCacheEnabled')
            );
```

Esimerkkikoodi 11. Asetuksien noutaminen urlParams-muuttujasta.

Staattisten ja/tai dynaamisten resurssien välimuistit pitää saada tarvittaessa myös poistettua käyttäjän selaimesta, jos päivityksen yhteydessä halutaan konfiguroida näiden

resurssien välimuistiin tallentaminen pois päältä. Tämän vuoksi toteutetaan myös niiden poisto riippuen asetuksista tulleista konfiguraatioiden arvoista.

Reitit rekisteröidään kahdessa if-else-lauseessa, joissa ehtoina toimivat aikaisemmin alustetut konfiguroinneista kertovat true- tai false-arvon omaavat muuttujat. If-lohkon sisällä tapahtuu reittien rekisteröinti ehdon ollessa tosi ja else-lohkon sisällä välimuistien poisto.

Esimerkkikoodista 12 nähdään, kuinka if-lauseen sisällä rekisteröidään uusi reitti workboxiin käyttäen moduulin funktiota registerRoute. Funktio saa parametrikseen aikaisemmin alustetun vakiomuuttujan kertomaan reitille kuuluvista palvelinpyyntöjen URL-osoitteista. Toisena parametrina registerRoute saa CacheFirst-olion. Olio edustaa cache first -strategiaa, jota service worker tulee noudattamaan kyseisessä reitissä. Tämä funktio saa parametrina objektin, joka pitää strategian asetukset sisällään. Else-lohkon sisällä tapahtuu välimuistin tuhoaminen CacheStorage-rajapinnan avustuksella.

```
var cacheableResponsePlugin = new workbox.cacheableResponse.Plugin({
            statuses: [200]
        });

if(cacheStaticHeliumCustom){
            workbox.routing.registerRoute(
                    new RegExp(STATIC_RESOURCE_PATTERN),
                    new workbox.strategies.CacheFirst({
                        cacheName: CACHE_NAMES.STATIC_RESOURCE_CACHE,
                        plugins: [cacheableResponsePlugin]
                    })
            );
        }
        else {
            caches.delete(CACHE_NAMES.STATIC_RESOURCE_CACHE);
        }
```

Esimerkkikoodi 12. Staattisia resursseja varten reitin rekisteröinti workboxiin.

Dynaamisia resursseja varten reitit rekisteröitiin vastaavasti suunnittelun mukaisesti, mikä käyttää omia niille suunnittelussa määriteltyjä strategioita. Tietotaulukoiden reitille luotiin oma käsittelijä valmiin workboxin stale while revalidate -strategiaa implementoivan tilalle.

Tietotaulukon reitin käsittelijän ideana on verrata pyynnön parametrejä välimuistissa avaimena olevien pyyntöjen parametreihin löytääkseen oikean avaimen, jota käytetään

vastauksen hakemisessa. Käsittelijässä mustalistataan yksi ongelma-osiossa mainittu tietotaulukko pois stale while revalide -strategiasta noudattamaan network first -strategiaa. Käsittelijässä toteutetaan myös alkioiden lisäys välimuistiin ja eliniän hallinta.

## 4.3 Yhteyden tilan näyttäminen käyttäjälle

### 4.3.1 Suunnittelu

Käyttäjän yhteyden tilan näkyminen käyttöliittymässä on oleellista, koska yhteydettömässä tilassa kaikki ominaisuudet eivät ole käytettävissä eikä jokainen sivu tai välilehti ole tavoitettavissa.

Ohjelmistoon luodaan oma JavaScript-tiedosto, jonka tehtävänä on päivittää käyttöliittymään tietoa yhteyden tilasta. Se tulee noudattamaan moduuli-suunnittelumallia kapseloiden itseään.

Käyttäjän reaaliaikaisen yhteyden tilan näyttäminen toteutetaan lisäämällä sivuston yläpalkkiin punainen ikoni, jos käyttäjällä ei ole yhteyttä. Ikoni lisätään Heliumin tarjoamalla omalla konfiguraatiolla. Ikonin ollessa esillä käyttäjälle viestitään, että hänellä ei ole internetyhteyttä.

Ikonin näkyvyyttä hallitaan moduulissa JavasScriptillä muuttaen sen CSS-ominaisuutta ”display”. Moduulissa rekisteröidään uusi tapahtumakuuntelija Heliumin tarjoamaan rajapintaan, joka kutsuu rekisteröintivaiheessa syötettyä callback-funktiota aina, kun käyttäjän yhteyden tila vaihtuu.

Moduulin nimeksi tulee ConnectionStatusManager, ja se tulee tarjoamaan kaksi metodia moduulin ulkopuolelle käytettäväksi:

**initialize:** Metodi alustaa moduulin. Se rekisteröi Heliumin App.ConnectivityService-moduuliin tapahtumakuuntelijan, joka aktivoituu käyttäjän yhteyden tilan vaihduttua. Tapahtumakuuntelija saa callback-funktioksi updateOnlineStatus-metodin. Metodi kutsuu myös updateOnlineStatus-metodia heti aluksi.

**updateOnlineStatus:** Metodi pitää logiikan sisällään, joka kysyy Heliumin App.ConnectivityService.IsOnline-rajapinnalta käyttäjän yhteyden tilan ja laittaa/poistaa punaisen ikonin yläpalkista yhteyden tilasta riippuen.

### 4.3.2 Ongelmat

Toteutuksen testivaiheessa havaittiin huomattavia ongelmia Heliumin tarjoaman ConnectivityService-moduulin IsOnline-rajapinnasta. Rajapinta palautti useasti true-arvon eli merkin yhteydestä, vaikka yhteyttä ei oikeasti ollut. Ongelma oli merkittävä ja korjattava, jotta käyttäjälle saadaan näytettyä hänen oikea yhteyden tilansa.

IsOnline-rajapinnan toimintalogiikka perustuu DOM:n Navigator.onLine-ominaisuuden kuunteluun. Testauksen yhteydessä huomattiin Internet Edgen ei-Chromium-pohjaisen selaimen olevan ainoa, jossa Navigator.onLine palautti oikeaa yhteyden tilaa vastaavia arvoja.

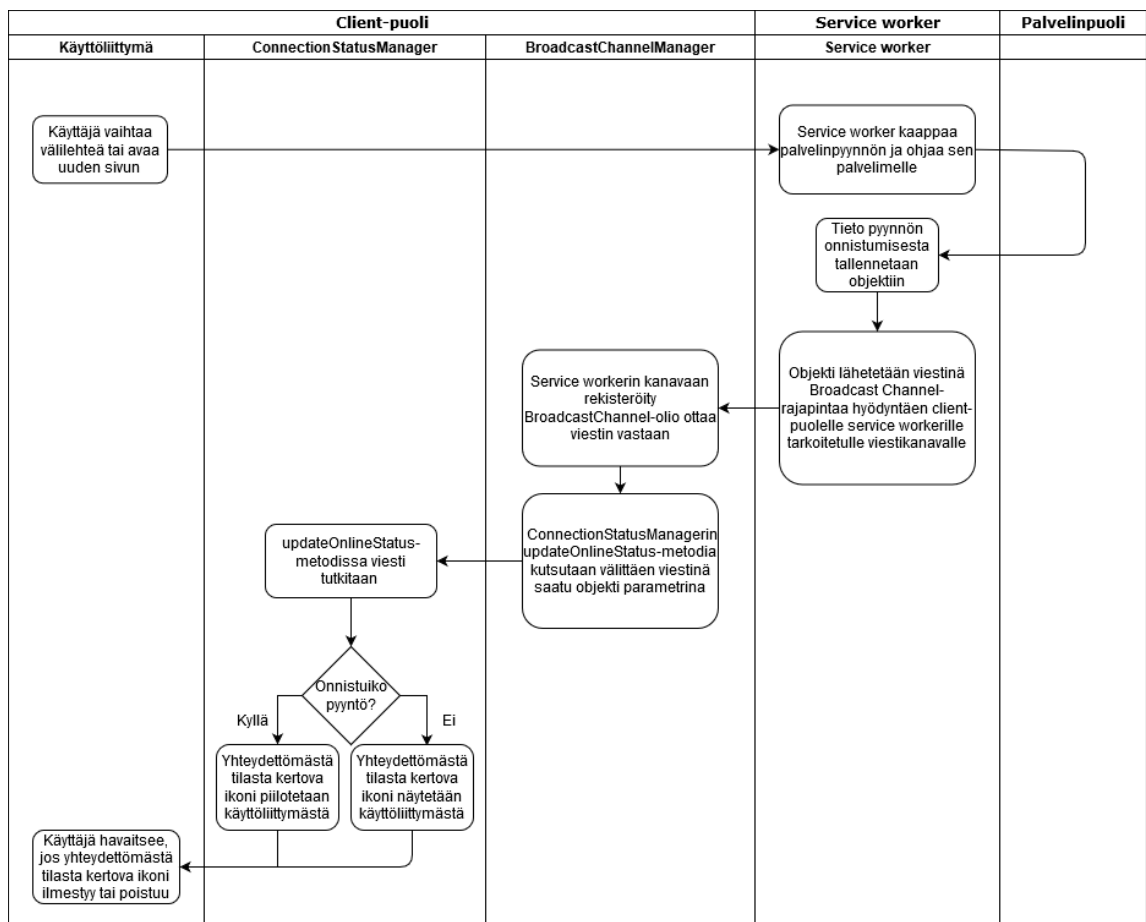
Navigator.OnLine kertoo käyttäjän selaimen yhteyden tilan palauttaen boolean-arvon, mutta eri selaimet toteuttavat ominaisuuden eri tavoin. Esimerkiksi Chrome ja Safari palauttavat false-arvon eli tiedon yhteydettömästä tilasta, jos selain ei saa yhteyttä paikalliseen verkkoon (LAN) tai reitittimeen. Kaikissa muissa tapauksissa selaimet palauttavat true-arvon, joten negatiivisia-positiivisia arvoja voi saada käyttäen kyseistä ominaisuutta yhteyden tilan määrittämiseen. [28.]

Ongelman ratkaisuksi muodostettiin tapa saada tieto service workerissa tapahtuvien palvelinpyyntöjen onnistumisesta ohjelmiston client-puolelle ConnectionStatusManager-moduuliin asti. Service worker toimii worker-kontekstissa, jonka takia sillä ei ole suoraa pääsyä DOM:iin ja client-puolelle [1]. Jos pyyntö palvelimeen epäonnistuu, silloin käyttäjällä ei ole yhteyttä ja toisinpäin.

Tarkoituksena on pitää Navigator.onLine-ominaisuuden kuuntelu pohjalla. Näin saadaan ainakin Internet Edgen käyttäjille näytettyä viiveittä heidän yhteyden tilansa. Muulloin tila päivittyy vasta käyttäjän tehtyä palvelinpyynnön.

Viestin saaminen client-puolelle (kuva 11) toteutetaan Broadcast Channel API:n avulla. Tämä rajapinta mahdollistaa yksinkertaisen tavan kommunikoida eri kontekstien välillä, joilla on sama alkuperä. Se perustuu BroadcastChannel-oliolla viestikanavan luomiseen, johon voidaan rekisteröidä muita BroadcastChannel-olioita eri puolelle ohjelmistoa. BroadcastChannel-oliot välittävät viesteinä mm. objekteja.

Client-puolelle luodaan BroadcastChannelManager-moduuli, joka on vastuussa BroadcastChannel-olioiden viestien käsittelystä. Moduuli hallitsee myös tietoa BroadcastChannelin saavutettavuudesta selaimessa.




**Kuva 11: Tieto palvelinpyynnön onnistumisesta vieminen client-puolelle yhteyden tilan näyttämiseksi.**

Ongelmapuolena Broadcast Channel API:ssa on, että Internet Edgen ei-Chromium-pohjaiset selaimet eivät tue sitä. Edgen ei-Chromium-pohjaiset selaimet tulevat käyttämään aikaisempaa toteutustapaa, joka toimii niissä hyvin.

Service workeriin tehdään muutoksia, jotta tieto pyyntöjen onnistumisista ja epäonnistumisista sekä viestin lähettäminen client-puolelle olisi mahdollista. Tämän vuoksi muokataan tapaa, jolla dynaamisten resurssien käsittelyyn käytettävät reitit toimivat ja rekisteröidään.

### 4.3.3 Toteutus

Tieto palvelinpyynnön onnistumisesta pitää saada client-puolelle, jotta sen perusteella käyttäjälle voidaan näyttää ikoni kertomaan yhteydettömästä tilasta käyttöliittymässä. Kyseinen tieto lähetetään Broadcast Channel API:a hyödyntäen. Service workeriin ja client-puolelle rekisteröidään samalle kanavalle ilmentymät BroadcastChannel-oliosta, jolloin tieto pystytään välittämään kanavaa pitkin objektina.

Esimerkkikoodissa 13 service workerin init-funktiossa try-catch-lohkossa luodaan uusi BroadcastChannel-olio, joka saa ainoana konstruktorin parametrina kanavan nimen. Jatkossa kaikki saman kanavan nimen jakavat BroadcastChannel-oliot kuuluvat samaan kanavaan.

```
const BROADCAST_CHANNEL_NAME = “sw-channel”;
var broadcastChannel;

//Init-funktiota kutsutaan, kun tämä serviceworker liitetään ylätasolle
var init = function(searchParams) {
     //init-funkion muu toiminnallisuus..

     //BroadcastChannel-olion luonti service workerin käyttöön
      try{
          broadcastChannel = new BroadcastChannel(BROADCAST_CHANNEL_NAME)
      } catch(e){
            //Broadcast Channel API ei ole tuettu selaimessa..
        }
    };
```

Esimerkkikoodi 13. BroadcastChannel-olion luonti serviceworkerini init-funktiossa.

Palvelinpyynnön onnistuminen pystytään selvittämään service workerissa käyttämällä workboxin tarjoamaa fetchDidFail-pluginia. Kun pyyntö epäonnistuu palvelimelta yhteyden puuttumisen takia, fetchDidFail-pluginissa määritettyä callback-funktiota kutsutaan. Callback-funktiossa merkataan fetchEvent-objektiin tieto, että palvelinpyyntö päättyi epäonnistumiseen.

FetchDidFail-plugin luodaan esimerkkikoodissa 14 fetchDidFailPlugin-muuttujalla, joka pitää objektin sisällään. Objektilla on ”fetchDidFail” niminen property, jonka arvona on asynkronoitu callback-funktio. Callback-funktio ei palauta mitään arvoa ja sen tehtävä on tässä toteutuksessa vain luoda event-objektiin uusi fetchFailed-property, josta saadaan myöhemmässä vaiheessa tieto pyynnön epäonnistumisesta.

```
const fetchDidFailPlugin = {
           fetchDidFail: async ({originalRequest, request, error, event}) => {
                    event.fetchFailed = true;
                }
         }
var pagesHandler = new workbox.strategies.NetworkFirst({
                     cacheName: CACHE_NAMES.PAGES_CACHE,
                     plugins: [expirationPlugin, cacheableResponsePlugin,
          fetchDidFailPlugin]
                 });
```

Esimerkkikoodi 14. FetchDidFail-pluginin luominen ja lisääminen reitin käsittelijään.

Reitin käsittelijän luonti muuttujaan on tarpeellista helpottamaan koodin lukua. Esimerkkikoodista 14 nähdään, että pagesHandler-muuttuja on ilmentymä NetworkFirst-luokasta, johon on lisättynä edellä luotu fetchDidFailPlugin.

Reitin rekisteröintiä muutettiin hieman, jotta vastausta päästäisiin tutkimaan. Esimerkkikoodissa 15 reitin käsittelijäksi laitetaan funktio, jossa pagesHandlerin handle-funktio käsittelee pyynnön strategian mukaisesti ja palauttaa promisen, joka päätyy vastaukseksi. Vastauksella ei ole mitään arvoa, jos sitä ei saa haettua palvelimelta tai välimuistista.

```
workbox.routing.registerRoute(
                    new RegExp(HELIUM_PAGE_URL_PATTERN),
                    function(ref) {
                        return pagesHandler
                            .handle({
                                event: ref.event
                            })
                            .then(function(response) {
                                return _handleResponse(response, ref.event) ||
                                _getFallbackResponse(ref.event);
                            })
                    }
                );
```

Esimerkkikoodi 15. Sivun ja välilehtien vaihtoon tarkoitetun reitin toteutus.

Tieto pyynnön onnistumisesta lähetetään client-puolelle BroadcastChannelManager-moduuliin, kun vastauksen haku joko palvelimelta tai välimuistista on valmis onnistuneena tai epäonnistuneena. Esimerkkikoodissa 16 vastauksia käsitellään _handleResponse-funktiossa, jos vastaus onnistutaan saamaan joko palvelimelta tai välimuistista. Muulloin kutsutaan _getFallbackResponse-funktiota.

_handleResponse saa responsen ja fetchEvent-objektin parametreinä. Jos response ei ole undefined ja Broadcast Channel -rajapinta on käytössä, metodissa luodaan kopio objEvent-muuttujaan fetchEvent-objektista (Esimerkkikoodi 16). objEvent-muuttujaan luodaan offline-property, joka saa arvonsa fetchEvent-objektin lisätyltä fetchFailed-propertylta. objEvent-muuttujan message-propertyyn laitetaan viesti merkkijonona, jotta client-puolella tiedetään, mikä toimenpide pitää tehdä.

```
var _handleResponse = function(response, event) {
        if (response && broadcastChannel) {
            var objEvent = JSON.parse(JSON.stringify(event));
            objEvent.offline = (event.fetchFailed) ? true : false;
            objEvent.message = UPDATE_ONLINE_STATUS;
            broadcastChannel.postMessage(objEvent)
        }
        return response;
    }
```

Esimerkkikoodi 16. _handleResponse palauttaa client-puolelle tietoa fetch-pyynnön onnistumisesta.

Funktio _getFallbackResponse tulee kutsutuksi vain, jos responsen arvo on undefined. Se välittää tiedon Broadcast Channel-rajapinnalla client-puolelle yhteydettömästä tilasta ja sen toteutusta tarkastellaan myöhemmin.

Broadcast Channel -rajapinnan kautta lähetetty viesti lähtee siihen rekisteröidylle kanavalle. Kanavan ja koko rajapinnan käsittelyyn luodaan ohjelmiston client-puolelle BroadcastChannelManager-moduuli. Sen vastuulla on kuunnella ohjelmistoon luotuja Broadcast Channel -olioiden kanavia, vastaanottaa niiden viestejä ja suorittaa toimenpiteitä niiden perusteella.

BroadcastChannelManager-moduulin tehtävä tässä tapauksessa on välittää viesti eteenpäin ConnectionStatusManager-moduuliin yhteyden tilan päivittämistä varten. Kun

service workerista tulee sille omistetulle kanavalle viesti, niin esimerkkikoodista 17 nähdään serviceworkerBroadcastChannel.onmessage-funktion määrittävän haluttu toimenpide fetchEventiin asetetun viestin perusteella. Tässä tapauksessa viestin ollessa updateOnlineStatus, tapahtuu kutsu _updateOnlineStatus-funktioon, joka on vastuussa connectionStatusManager-moduulin updateOnlineStatus-funktion kutsumisesta välittäen sinne saadun viestin.

```
const SWBroadcastChannelName = 'sw-broadcastchannel';
    var serviceworkerBroadcastChannel;
    try{
        serviceworkerBroadcastChannel = new BroadcastChannel(SWBroad-
castChannelName);
    } catch(e){
    console.log('Broadcast channel is not supported in this browser');
    }

const SHOW_OFFLINE_MESSAGE = "showOfflineMessage";
const UPDATE_ONLINE_STATUS = "updateOnlineStatus";
if(serviceworkerBroadcastChannel){
    serviceworkerBroadcastChannel.onmessage = function(event) {
        switch(event.data.message){
        case SHOW_OFFLINE_MESSAGE:
                _showOfflineMessage();
                break;

        case UPDATE_ONLINE_STATUS:
                _updateOnlineStatus(event);
                break;
        default:
            console.log("Not valid message received");
        };
    };
}
function _updateOnlineStatus(event){
        App.connectionStatusManager.updateOnlineStatus(event);
    }
```

Esimerkkikoodi 17. BroadcastChannelManager-moduulin toteutus yhteyden tilan välittämisestä ConnectionStatusManager-moduuliin.

ConnectionStatusManager-moduulin tarjoama updateOnlineStatus-metodi määrittää yhteyden tilan parametrina saadussa objektissa olevan tiedon mukaan. Kuitenkin, jos sitä kutsutaan ilman parametrejä, se kysyy yhteyden tilaa Heliumin ConnectivityService-moduulin IsOnline-rajapinnalta. Tämä rajapinta nojautui joissain selaimissa epäluotettavaksi todettuun Navigator.onLine-ominaisuuteen. UpdateOnlineStatus-metodi on myös vastuussa punaisen ikonin (kuva 12) esittämisestä ja piilottamisesta yläpalkissa muuttamalla ikonin CSS:n display-ominaisuus none-arvosta visible-arvoon.

Kuva 12: Yhteydettömästä tilasta ilmoittava ikoni yläpalkissa.

ConnectionStatusManager-moduuli kysyy myös init-funktiossa BroadcastChannelManager-moduulilta BroadcastChannel-olion olemassaolosta siihen määritetyllä rajapinnalla. Tällöin se tietää, pitääkö uuden sivulatauksen yhteydessä pyytää tietoa käyttäjän yhteyden tilasta erikseen updateOnlineStatus-funktiolta. Näin ollen ne selaimet, jotka eivät tue Broadcast Channel API:a, saavat sivun päivittyessä tiedon yhteyden tilasta.

## 4.4 Käyttäjälle ilmoittaminen, kun hän yrittää yhteydettömässä tilassa vierailla sivulla, joka ei ole välimuistissa

### 4.4.1 Suunnittelu

Vaatimuksen mukaisesti sivuston sisällön pitää olla käytettävissä yhteydettömässä tilassa, jos käyttäjä on kyseisessä sivuston osassa aiemmin käynyt. Käyttäjän vierailu esimerkiksi jollakin välilehdellä vie välilehden sisällön välimuistiin, jolloin välilehdelle pystytään navigoimaan yhteydettömässä tilassa. Käyttäjä voi kuitenkin helposti pyrkiä navigoimaan yhteydettömässä tilassa sivuston osaan, jota ei löydy välimuistista. Tämän tapahtuessa käyttäjän tulee saada käyttöliittymään ilmoitus yhteydettömästä tilasta, jotta hän ei ihmettele sivuston toimimattomuutta.

Heliumin lähdekoodiin on luotu keino service workerille näyttää käyttäjälle informatiivinen viesti (kuva 13) käyttäjän navigoidessa saavuttamattomissa olevalle sivulle. Viesti toimitetaan lähdekoodissa käyttöliittymään toastr-kirjastolla, ja se ilmestyy oikeaan yläkulmaan hetkellisesti. Viestin lähettäminen käyttöliittymään tapahtuu, jos palvelinpyynnön Response-objektia muokataan ilmoittamaan lähdekoodiin yhteydettömästä tilasta.

Kun käyttäjä vaihtaa välilehteä tai avaa uuden sivun, lähdekoodissa katsotaan, sisältääkö operaation lopuksi saatu Response-objekti eli vastaus offline-propertyn, jonka arvo on true. Tällöin lähdekoodissa lopetetaan pyynnön toteuttaminen sekä muut toimenpiteet, jonka jälkeen avoimeen lähdekoodiin perustuvalla toastr-kirjaston avulla käyttäjälle lähetetään viesti käyttöliittymään.

Service workeriin täytyy toteuttaa Response-objektin luominen, joka aktivoi lähdekoodissa viestin lähettämisen käyttöliittymään. Tämä Response-objekti palautetaan, jos palvelimelta tai välimuistista ei saada vastausta. Viestin sisältö määritetään siihen tarkoitetussa resurssitiedostossa.

### 4.4.2 Ongelmat

Toteutusta tehdessä havaittiin ongelma Heliumin lähdekoodin toteutuksessa, joka aiheutti lähdekoodista tulevan viesti toimimisen satunnaiseksi. Lähdekoodin toteutusta tutkittaessa tuli ilmi, että siinä nojauduttiin myös jo aikaisemmin epäluotettavaksi todetun ConnectivityService-moduulin IsOnline-rajapintaan.

Samassa kohtaa, missä Heliumin lähdekoodi tarkistaa Response-objektin offline-propertyn arvon, se tarkistaa myös IsOnline-rajapinnalta käyttäjän yhteyden tilan. Jos käyttäjällä ei ole yhteyttä, mutta IsOnline-rajapinta palauttaa arvon true yhteydellisen tilan merkiksi, viestin lähettäminen käyttöliittymään ei tapahdu.

Ongelman ratkaisuksi muodostuu ConnectivityService-moduulin IsOnline-rajapinnan toteutuksen uudelleenkirjoitus ConnectionStatusManager-moduulissa. Rajapinta tulee jatkossa palauttamaan yhteyden tilasta kertovan boolean-arvon perustuen aikaisemmin Broadcast Channel API:lla toteutettuun palvelinpyyntöjen onnistumisen kuunteluun. Jos käyttäjän selain ei tue Broadcast Channel API:a, niin silloin toteutuksen uudelleenkirjoitus ei astu voimaan.

### 4.4.3 Toteutus

Välilehden vaihtamiseen ja sivujen avaamiseen tarkoitettu reitti aktivoituu, kun käyttäjä navigoi käyttöliittymässä. Tähän reittiin luodaan toiminnallisuus ilmoittaa yhteydettömästä tilasta käyttäjälle viestillä, jos sivu tai välilehti on saavuttamattomissa.

Viestin lähettäminen tapahtuu lähdekoodissa, kun käyttäjällä ei ole yhteyttä ja reitti palauttaa Response-objektin, jolta löytyy offline-property arvolla true. Kun reitti käsittelee palvelinpyynnön eikä saa palautettavaa vastausta palvelimelta tai välimuistista, niin esi-

merkkikoodissa 10 nähty _getFallbackResponse-funktio luo uuden palautettavan Response-objektin fbr-muuttujaan (esimerkkikoodi 18). Response-objekti fbr sisältää offline-propertyn, jonka arvo on true.

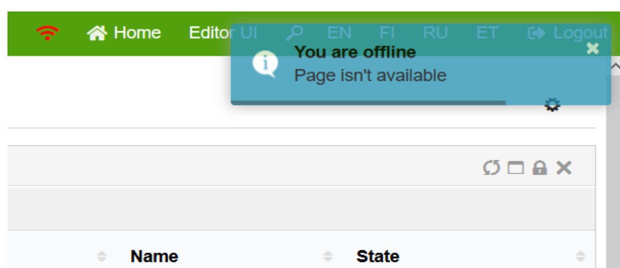
```
var _getFallbackResponse = function(event) {

            var fbr = {
                offline: true
            };
            fbr = new Response(JSON.stringify(fbr), {
                headers: {
                    'Content-Type': 'application/json'
                }
            });
           //Yhteyden tilasta tiedon lähettäminen client-puolelle
            if (broadcastChannel) {
                var objEvent = JSON.parse(JSON.stringify(event));
                objEvent.message = UPDATE_ONLINE_STATUS;
                objEvent.offline = true;
                broadcastChannel.postMessage(objEvent);
            }
            return fbr;
        };
```

Esimerkkikoodi 18. _getFallbackResponsen toteutus

Tiedon saaminen käyttäjän yhteyden tilasta lähdekoodissa perustuu epäluotettavaa ConnectivityService-moduulin IsOnline-rajapintaan. Rajapinnan toteutus uudelleen kirjoitettiin ConnectionStatusManager-moduuliin. Uusi toteutus nojautuu työssä aikaisemmin tehtyyn toiminnallisuuteen, jossa yhteyden tila määritetään palvelinpyyntöjen onnistumisen mukaan. Jos Broadcast Channel API ei ole selaimessa tuettu, niin uusi toteutus IsOnline-rajapinnasta ei myöskään astu voimaan.

Toteutuksen lopputuloksena käyttäjä saa kuvan 13 mukaisen viestin sivuston oikeaan yläkulmaan viiden sekunnin ajaksi, jos hän yrittää yhteydettömässä tilassa navigoida saavuttamattomissa olevalle sivulle tai välilehdelle.




Kuva 13: Heliumin lähdekoodista tuleva viesti ilmoittamaan yhteydettömästä tilasta.

# 5 Latausajan mittaus

Jotta latausaikojen lyheneminen voitiin todentaa, latausajat mitattiin ennen ja jälkeen insinöörityön toteutusta. Latausaikojen lyhenemiseen tähdättiin siirtämällä tietotaulukoiden lataamisen tapahtuvan ensisijaisesti välimuistista. Kokonaisvaltaista mittausta ei suoritettu, vaan ainoastaan raskaimman välilehden lataamiseen käytettyä aikaa. Tämä ei anna tarkkaa tietoa järjestelmän eri osioiden nopeutumisesta, vaan kokonaisvaltaisen kuvan.

Latausajan mittaamisen kohteena käytettiin sivuston raskaimman tietotaulukon sisältävän välilehden lataamiseen ja sen esittämiseen käytettyä kokonaisaikaa. Näin saatiin aito kokonaisaika selville, kauanko käyttäjällä kestää saada välilehti täydellisesti esitettynä selaimessa.

Latausaika koostuu kuudesta osasta: pyynnön lähettämisestä palvelimelle, palvelimella pyynnön käsittelemisestä, palvelimelta tulevan vastauksen saapumisesta, selaimen DOM:n vastauksen käsittelystä, renderoinnista ja lopuksi window.onload-eventin käyttämästä ajasta. [30.]

Palvelimelta tulevan tietotaulukoiden muodostamiseen käytettävän turhan datan määrää karsittiin huomattavasti pois, kun tietotaulukoiden prosessointi siirrettiin konfiguroimalla tapahtumaan selaimessa. Datan määrä saatiin tippumaan huomattavasti ja pelkästään se lyhensi latausaikoja.

Datan karsimisen takia latausajat mitataan toteutuksen jälkeen myös palvelimelta. Näin voidaan vertailla eroa latausajoissa palvelimen ja selaimen välimuistin väliltä. Käyttäjän havaitsema latausaikojen lyheneminen saadaan vertaamalla ennen toteutusta mitattuja latausaikoja ja toteutuksen jälkeen mitattuja latausaikoja.

Välilehden latausajat mitataan kolmesti ja tuloksista lasketaan keskiarvot, jotka toimivat mittareina latausajan muuttumisesta.

## 5.1 Mittaus

Latausaikoihin vaikuttavat mm. verkkoyhteys, palvelimen kuormitus ja testissä käytetyn tietokoneen suorituskyky ja tietokoneen sen hetkinen kuormitus. Mittaus tehdään kolme kertaa samassa verkossa siihen aikaan, kun järjestelmän palvelimella ja mittaukseen käytetyssä tietokoneen verkossa ei ole muita käyttäjiä paikalla.

Mittaus suoritettiin Google Chromen versiolla 80.0.3987.132. Chrome tarjoaa suorituskyvyn ja latausajan mittaukseen tarkoitettua työkalua kehittäjien työkaluissa. Mittaus alkoi välilehden hiirenklikkauksesta, joka aloittaa toimenpiteistä muodostuvan ketjun, joka johtaa testattavan välilehden lataamiseen ja esittämiseen. Loppuaika otettiin koko tapahtumaketjun päättyessä.

## 5.2 Mittaustulokset

Ennen insinöörityön toteutusta mittauksen kohteena olevan välilehden latausajan kolmen mittauskerran keskiarvoksi saatiin 10,34 sekuntia. Toteutuksen jälkeinen välilehden latausajan keskiarvo palvelimelta ladatessa oli 7,93 sekuntia ja osittain välimuistista ladatessa 2,49 sekuntia.

E1 = Mittaus ennen toteutusta: välilehden lataus palvelimelta ja selaimessa esitettynä.

T1 = Mittaus toteutuksen jälkeen: välilehden lataus palvelimelta ja selaimessa esitettynä.

T2 = Mittaus toteutuksen jälkeen: välilehden lataus osittain välimuistista ja selaimessa esitettynä.

| Mittauskerta | E1 | T1 | T2 |
|---|---|---|---|
| 1. | 10,99 s | 7,81 s | 2,64 s |
| 2. | 10,23 s | 7,91 s | 2,48 s |
| 3. | 9,80 s | 8,07 s | 2,36 s |
| Keskiarvo: | 10,34 s | 7,93 s | 2,49 s |

### 5.3 Päätelmä

Käyttäjien kokema latausajan lyheneminen (E1-T2) on 7,85 sekuntia, joka on neljäsosa alkuperäisestä. Tässä ei oteta huomioon ensimmäistä latauskertaa, joka tapahtuu palvelimelta. Välimuistin hyödyntäminen nopeutti välilehden avaamista 5,44 sekuntia (T1-T2).

Mittaustuloksista voidaan todeta, että insinöörityössä tehty sovitus onnistui lyhentämään latausaikoja huomattavasti. Raskaimman välilehden lataaminen onnistuu jatkossa huomattavasti nopeammin riippumatta yhteyden nopeudesta. Muiden tietotaulukon sisältämien välilehtien ja sivujen osalta oletetaan tuloksien olevan saman suuntaisia välimuistin hyödyntämisen takia.

# 6 Kehitysehdotukset

Edellä on käyty läpi TVC Heliumin offline-ominaisuuden implementoinnin eri työvaiheita sekä niissä kohdattuja ongelmia. Tässä luvussa esitetään muutamia kehitysehdotuksia, joiden avulla Heliumin tuotekehitys voi kehittää Heliumia ja sen offline-ominaisuutta.

Nämä kehitysehdotukset muodostuivat offline-ominaisuuden implementointia tehdessä, ja ne palvelevat niin asiakaskohtaisia sovituksia tekeviä sovelluskehittäjiä kuin loppukäyttäjiä. Kehitysehdotuksia ovat seuraavat viisi:

1. Tietotaulukoiden luonnin uudelleensuunnittelu, jotta tietotaulukon pyyntöön käytetty URL-osoite olisi vakio. Tämä mahdollistaisi workboxin StaleWhileRevalidate-luokan käyttämisen ilman kustomointia, mikä tuottaisi vakaamman lopputuloksen. Nykyisellä toteutuksella sovelluskehittäjät joutuvat itse luomaan tietotaulukkoja varten stale while revalidate -strategian toteuttavan käsittelijän, jonka saaminen saumattomasti toimimaan muun ohjelmiston kanssa on hankalaa ja vie aikaa.

2. Käyttäjän yhteyden tilan selvittämiseen käytetyn ConnectivityService-moduulin uudelleensuunnittelu. Nykyinen toteutus moduulissa nojautuu epäluotettavaan Navigator.onLine-ominaisuuden kuunteluun.

3. Tietotaulukoiden konfigurointiin pitää saada mahdollisuus turhan datan pois jättämiseen palvelimelta tulevasta vastauksesta, jos sitä ei tarvita. Vaikka asiakas ei käyttäisi offline-ominaisuutta, niin turhan datan lataaminen yleisesti pidentää latausaikoja ja näin ollen vaikuttaa loppukäyttäjän käyttökokemukseen.

4. Paremman kontrollin antaminen sovelluskehittäjälle, jos palvelinpyynnön aiheuttama prosessointi halutaan keskeyttää hienovaraisesti. Tämä tilanne tulee yleensä eteen, jos palvelinpyyntö ei saa vastausta palvelimelta tai välimuistista. Tällä hetkellä tämä toiminto nojautuu siihen, että service worker palauttaa tietynlaisen objektin, jolloin käyttäjä saa ilmoituksen yhteydettömästä tilasta ja pyynnön prosessointi keskeytyy. Sovelluskehittäjälle pitää luoda toisenlainen tapa saada pyynnön prosessointi pysähtymään, mikä on erillään viestin näyttämisestä käyttäjälle, saadakseen paremman kontrollin ohjelman kulusta.

5. Uuden service workerin asentamisen korjaus vanhan päälle. Tällä hetkellä lähdekoodissa toteutetun service workerin asennus ei välttämättä asenna päivittynyttä service workeria vanhan tilalle selaimeen. Muuttuneen service workerin pitää asentua vanhan tilalle, jotta asiakkaan ohjelmiston päivityksien jälkeen muutokset service workerissa tulevat voimaan automaattisesti.

## 7 Yhteenveto

Insinöörityössä toteutettiin asiakkaan PDM-järjestelmä toimimaan tiettyjen vaatimuksien mukaisesti yhteydettömässä tilassa. Toteutus tehtiin TVC Helium-ohjelmiston offline-ominaisuudella, joka on rakennettu yhdistelemällä service worker- ja workbox-teknologioita. Toissijaisena tavoitteena oli lyhentää raskaiden sivujen ja välilehtien latausaikoja hyödyntämällä service worker -teknologiaa. Service worker mahdollisti staattisten ja dynaamisten resurssien tallentamisen selaimen välimuistiin ja sieltä niiden lataamiseen.

Toteutus oli ensimmäinen tuotantokäyttöön tehty sovitus TVC Heliumin offline-ominaisuudesta. Toteutusta tehdessä havaituista ongelmista saatiin koostettua tärkeitä kehitysehdotuksia TVC Helium -tuotteen tuotekehitykselle. Näiden kehitysehdotuksien pohjalta TVC Heliumin tuotekehitys pystyy viemään tuotteen offline-ominaisuuden askeleen verran eteenpäin tuotantokäyttöön soveltuvaksi ilman sovelluskehittäjän omia ratkaisuja.

Insinöörityön tuloksena saatiin vaatimuksien mukaisesti yhteydettömässä tilassa toimiva PDM-järjestelmän käyttöliittymä, jota asiakas käyttää tuotetietojensa katselmoimiseen. Toissijaisena tavoitteena ollut latausaikojen lyheneminen todennettiin mittauksella. Mittauksessa mitattiin sivuston raskaimman välilehden latausajan eroa ennen ja jälkeen sovituksen, mikä lyheni neljäsosaan. Mittaustuloksesta voidaan päätellä, että latausaikojen lyheneminen onnistui yleisesti ottaen niillä sivuilla ja välilehdillä, joissa voidaan hyödyntää selaimen välimuistia.

## Lähteet


1 Service Worker API. 2019. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API>. 25.10.2019. Luettu 20.12.2019.

2 Introduction to service worker. 2019. Verkkoaineisto. Google LLC. <https://developers.google.com/web/ilt/pwa/introduction-to-service-worker>. 10.06.2019. Luettu 20.12.2019.

3 Using Web Workers. 2020. Verkkoaineisto. Mozilla Foundation. Google LLC <https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers>. 11.01.2020. Luettu 17.02.2020.

4 Gaunt, Matt. 2019. Service Workers: an Introduction. Verkkoaineisto. <https://developers.google.com/web/fundamentals/primers/service-workers>. 09.08.2019. Luettu 20.12.2019.

5 About JavaScript. 2019. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript>. 04.12.2019. Luettu 25.01.2020.

6 XMLHttpRequest. 2019. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>. 12.12.2019. Luettu. 25.01.2020.

7 Fetch API. 2020. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>. 23.01.2020. Luettu 25.01.2020

8 Gaunt, Matt. 2019. Introduction to fetch(). Verkkoaineisto. <https://developers.google.com/web/updates/2015/03/introduction-to-fetch>. 20.03.2019. Luettu 09.02.2020.

9 Request. 2019. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/API/Request>. 14.11.2019. Luettu 25.01.2020.

10 Response. 2020. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/API/Response>. 24.01.2020. Luettu 25.01.2020.

11 Using Promises. 2019. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises>. 19.12.2019. Luettu 25.01.2020.

12 Callback function. 2020. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Glossary/Callback_function>. 03.01.2020. Luettu 09.02.2020.

13 Cache. 2019. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/API/Cache>. 14.08.2019. Luettu 28.12.2019.

14 CacheStorage.open(). 2019. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage/open>. 14.08.2019. Luettu 09.02.2020.

15 Build a PWA using Workbox. Verkkoaineisto. Google LLC. <https://codelabs.developers.google.com/codelabs/workbox-lab/#0>. Luettu 20.12.2019.

16 Workbox. Verkkoaineisto. Google LLC. <https://developers.google.com/web/tools/workbox>. Google LLC. Luettu 20.12.2019.

17 Workbox strategies.Verkkoaineisto. Google LLC. <https://developers.google.com/web/tools/workbox/modules/workbox-strategies>. Luettu 23.12.2019.

18 Using Plugins. Verkkoaineisto. Google LLC. <https://developers.google.com/web/tools/workbox/guides/using-plugins> Luettu 23.12.2019.

19 Workbox-strategies. Verkkoaineisto. Google LLC. <https://developers.google.com/web/tools/workbox/reference-docs/latest/module-workbox-strategies>. Luettu 23.12.2019.

20 Archibald, Jake. 2019. The Offline Cookbook. Verkkoaineisto. Google LLC <https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook>. 12.02.2019. Luettu 23.12.2019.

21 Rylan, Cory. 2015. JavaScript Module Pattern Basics. Verkkoaineisto. <https://coryrylan.com/blog/javascript-module-pattern-basics>. 10.03.2015. Luettu 25.01.2020,

22 Product Data Management. Verkkoaineisto. BusinessDictionary. <http://www.businessdictionary.com/definition/product-data-management-PDM.html>. Luettu 25.01.2020.

23 TECHNIA, Verkkoaineisto. TECHNIA Oy. <https://www.technia.com/>. Luettu 25.01.2020.

24 Progressive Web Apps. 2019. Verkkoaineisto. Drifty. <https://ionicframework.com/docs/intro/what-are-progressive-web-apps>. 12.12.2019. Luettu 06.02.2020.

25 LePage, Pete & Richard, Sam. 2020. What Are Progressive Web Apps. <https://web.dev/what-are-pwas/>. 24.02.2020. Luettu 08.03.2020.

26 MVC Architecture. Verkkoaineisto. TutorialsTeacher. <https://www.tutorialsteacher.com/mvc/mvc-architecture>. Luettu 08.03.2020.

27 RFC 2616 - Status Code Definitions. Verkkoaineisto. World Wide Web Consortium. <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>. Luettu 28.12.2019.

28 Workbox-routing. Verkkoaineisto. Google LLC. <https://developers.google.com/web/tools/workbox/modules/workbox-routing>. Luettu 28.12.2019.

29 Navigator.onLine. 2019. Verkkoaineisto. Mozilla Foundation. <https://developer.mozilla.org/en-US/docs/Web/API/NavigatorOnLine/onLine>. 5.12.2019. Luettu 26.01.2020.

30 Page load timing process. Verkkoaineisto. New Relic Documentation. <https://docs.newrelic.com/docs/browser/new-relic-browser/page-load-timing-resources/page-load-timing-process>. Luettu 08.02.2020