Heli Hyttinen

VERKKOSOVELLUKSEN KEHITYS HUOMIOIDEN KÄYTTÖ OFFLINE-TILASSA

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittely

2021

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Tradenomi (AMK)
Tekijä/Tekijät	Heli Hyttinen
Työn nimi	Verkkosovelluksen kehitys huomioiden käyttö offline-tilassa
Toimeksiantaja	Ontec Oy
Vuosi	marraskuu 2021
Sivut	51 sivua
Työn ohjaaja(t)	Arto Väätäinen

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli esitellä työssä toteutettavan sovelluksen kehitykseen käytetyt tekniikat ja kuvata sovelluksen kehitys rakentamalla palvelin- ja selainpuolen sovellukset. Lisäksi selvitettiin, mitä vaaditaan selainsovelluksen toimintaan verkkoyhteydettömässä tilassa. Opinnäytetyön kehittämistavoitteena oli toteuttaa responsiivinen verkkosovellus, joka käyttää REST API -rajapinnan yli tietokantaa ja toimii tarvittaessa ilman verkkoyhteyttä. Käyttöliittymän kehityksessä tuli huomioida QR-koodilukijan käyttö.

Palvelinsovellus kehitettiin ASP.NET Core -alustalle, jonka käyttämä tietokanta toteutettiin SQL Serverille. Relaatiotietokannan kanssa vuorovaikuttamaan valittiin Entity Framework Core -kehys. Selainsovellus rakennettiin Reactilla, jonka käyttöliittymän kehitykseen käytettiin Material-UI-komponentteja. QR-koodilukijaksi valittiin avoimen lähdekoodin sovellus, joka avaa laitteen kameran ja tunnistaa QR-koodin sisällön. Selainsovelluksen toiminta offline-tilassa toteutettiin progressiivisen verkkosovelluksen tekniikoin.

Kehitetyn sovelluksen tarkoituksena on auttaa tarkastusten, kuten teollisuuden laitetarkastusten, teossa. Sovellus esittää tietoja tarkastuksista ja niihin kuuluvista kohteista, lisäksi sillä voidaan suorittaa tarkastusten kirjauksia. Tietyn kohteen kirjausnäkymä voidaan avata lukemalla kohdetta vastaava QR-koodi.

Selainsovelluksen toimintaan verkkoyhteydettömässä tilassa vaaditaan Service Worker -koodi ja selaimen välimuisti. Työssä kehitetty selainsovellus käyttää lisäksi IndexedDB-tietokantarajapintaa ja sen välimuistin käyttöstrategioita ovat Cache-first sekä Network-first. Selainsovellus tallentaa välimuistiin REST API -rajapinnalta aina uusimman tiedon. Kun käyttäjä tekee kirjauksen offline-tilassa, kirjauksen tiedot tallennetaan selaimen paikalliseen tietokantaan. Verkkoyhteyden palautuessa tiedot siirretään selaimen paikallisesta tietokannasta palvelimelle.

Johtopäätöksenä yleisesti alan kannalta voidaan todeta PWA-tekniikoiden kaipaavan lisää kehitystä liittyen tietojen tallennustapaan verkkoyhteydettömässä tilassa. Opinnäytetyön tavoitteet saavutettiin, ja työn tuloksena saatu sovellus vastaa asetettuja vaatimuksia. Kehityskohteita jätettiin jatkokehitykseen ja myös työn rajauksen puolesta lisätoiminnallisuuksien jatkokehittäminen on tarpeen. Toteutukseen käytetyt tekniikat soveltuivat kehitystarpeisiin ja toimivat hyvin yhdessä.

Asiasanat: ohjelmistokehitys, progressiivinen verkkosovellus, REST API, React, ASP.NET

South-Eastern Finland University of Applied Sciences

Degree	Bachelor of Business Administration
Author (authors)	Heli Hyttinen
Thesis title	Development of an offline web application
Commissioned by	Ontec Oy
Time	November 2021
Pages	51 pages
Supervisor	Arto Väätäinen

ABSTRACT

The objective of the bachelor's thesis was to develop a responsive web application that would use a database via REST API and be available without a network connection. Furthermore, the web application should have a possibility to use a QR code scanner. The thesis introduced technologies used and described the full stack development. In addition to the description of the development, it examined what was required for offline use of the developed web application.

The web application was created by using ASP.NET Core with React. ASP.NET Core was used as backend and the database was created on SQL Server. Entity Framework Core was chosen for the interaction with the database. Frontend was created with React and its component library called Material-UI. An open source application opening the camera of the device and identifying the contents of the QR code was chosen as a QR code scanner of the developed application. The offline functionality of the web application was carried out with the technologies of progressive web application. The purpose of the web application was to help inspect different targets, such as industrial equipment. The view of the inspection target would open by scanning the QR code.

The main results of this thesis were the actual web application with offline functionality and description of its development. Service Worker code enabled the web application to run offline using cache API. The caching strategies of the web application were Cache first and Network first. The web application uses the local IndexedDB database for storing data offline and it fetches data from the local database to the server when the network is available.

Based on the results it seemed that the PWA technologies would in general need more development for storing data offline. The thesis and the web application achieved their objectives. The web application would need further development due to the thesis framework and features which were identified during development. The technologies used for development were suitable and operated well together.

Keywords: full stack development, progressive web application, REST API, React, ASP.NET

SISÄLLYS

1 JOHDANTO

Opinnäytetyön tavoitteena on esitellä työssä toteutettavan sovelluksen kehitykseen käytettävät tekniikat ja kuvata sovelluksen kehitys rakentamalla palvelin- ja selainpuolen sovellukset. Lisäksi selvitetään, mitä vaaditaan selainsovelluksen toimintaan verkkoyhteydettömässä tilassa. Opinnäytetyön kehittämistavoitteena on toteuttaa responsiivinen verkkosovellus, joka käyttää REST API -rajapinnan yli tietokantaa ja toimii tarvittaessa ilman verkkoyhteyttä. Sovelluksen käyttöliittymän kehityksessä huomioidaan QR-koodilukijan käyttö. Työ tehdään toimeksiantajayritykselle Ontec Oy. Opinnäytetyön produktiivisen tuotoksen on tarkoitus olla jatkokehitettävä prototyyppi, eli työssä ei oteta huomioon esimerkiksi sovelluksen tietoturvaa.

Opinnäytetyössä luodaan tietokantataulut ja REST API -rajapinta, kehitetään sovellukselle responsiivinen käyttöliittymä ja lisätään selainpuolelle ominaisuudet offline-toimintaa varten. Opinnäytetyön raportin luvussa kaksi käsitellään työssä käytettävät tekniikat ja työvälineet. Kolmas luku kuvaa kehitystyötä ja sovelluksen toimintaa alaotsikoilla palvelinsovellus, selainsovellus ja selainsovelluksen offline-tila. Viimeisessä luvussa on opinnäytetyön päätäntö jatkokehitysehdotuksineen.

Opinnäytetyössä kehitettävän sovelluksen tarkoituksena on auttaa tarkastusten, kuten teollisuuden laitetarkastusten, teossa. Sovellus auttaa loppukäyttäjää tunnistamaan, mitkä tarkastukset ovat seuraavaksi ajankohtaisia ja mitkä ovat mahdollisesti myöhässä. Sovelluksella suoritetaan tarkastusten kirjaus ja esitetään tietoja liittyen tarkastuksiin ja tarkastusten kohteisiin. Tarkastuksiin voi kuulua useita eri kohteita. Jotta tarkastuskokonaisuus voidaan kirjata tarkastetuksi, jokainen tarkastuksen kohde on oltava tarkastettu. Opinnäytetyössä käytetään geneerisiä tietoja sovelluksen toimintaperiaatteita havainnollistamaan tietojen liittymättä toimeksiantajayritykseen. Tarkastuksen kohteina esitetään automerkkejä tai -malleja ja tarkastuksina auton huoltoon yleisesti liittyviä toimenpiteitä kuten auton pesu tai katsastus.

Sovelluksen kehityksessä otetaan huomioon sovelluksen käyttö olosuhteissa, joissa verkkoyhteyttä ei välttämättä ole saatavilla. Sovellus tallentaa ja hakee tiedot tietokannasta REST API -rajapinnan kautta. Kun verkkoyhteyttä ei ole

saatavilla, tiedot haetaan selaimen välimuistista. Käyttäjän lähettämät tiedot tallennetaan paikallisesti verkkoyhteyden palautumiseen saakka.

Käyttöliittymään kehitetään ominaisuus QR-koodin lukemiselle mobiililaitteilla. Kehitystyössä käytetään avoimen lähdekoodin QR-koodilukijaa. QR-koodilukijan on tarkoitus auttaa loppukäyttäjää tarkastuskohteiden tunnistamisessa. Sovellus avaa tarkastuskohteen kirjausnäkymän lukemalla kohdetta vastaavan QR-koodin.

2 KÄYTETTÄVÄT TEKNIIKAT

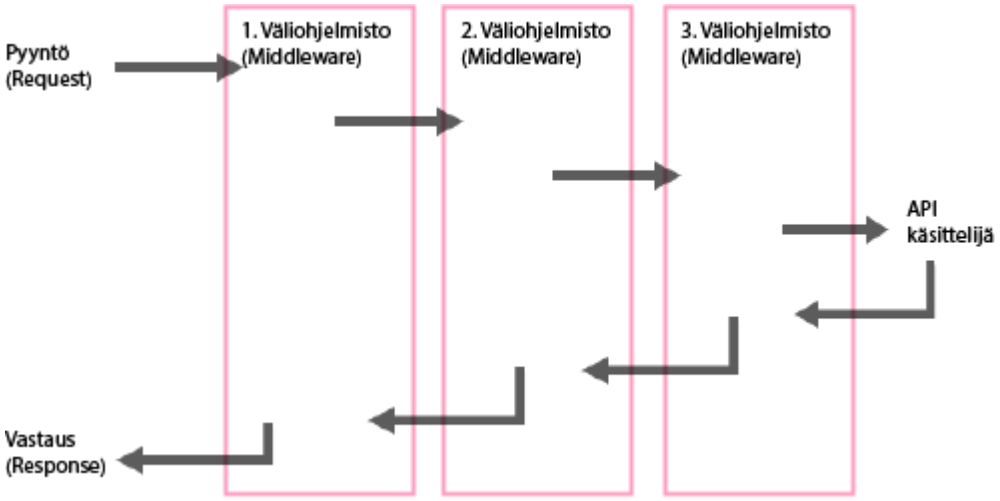
Toimeksiantajan sovellukset ovat pääasiassa toteutettu Windows-ympäristöön suosien Microsoftin sovellustekniikoita, minkä vuoksi tämä työ tehdään suosimalla Microsoftin työvälineitä. Käyttöliittymä kehitetään JavaScriptin React-kirjastolla opinnäytetyön tekijän aiempaan osaamiseen perustuen. Kehitystyö toteutetaan Visual Studiossa ASP.NET Core -kehyksen projektimallilla, jossa valmiina lisäosana on React-sovellus selainpuolen ohjelmointia varten. .NET-kirjastot tukevat C#-kieltä, joten työssä kehitettävä palvelinsovellus tulee käyttämään kyseistä kieltä. Sovelluksen käyttämä relaatiotietokanta luodaan Microsoftin SQL Server -tietokantojen hallintajärjestelmällä. Yhteys tietokantaan luodaan Entity Framework -olio-relaatio-mallinnus-kehyksellä (Object Relational Mapper, ORM).

Sovelluksen toiminta offline-tilassa tapahtuu JavaScript-koodilla selainpuolella. Service Worker -koodin välimuistin käyttöstrategioina käytetään Cache-first-strategiaa ja Network-first-strategiaa. Service Worker tallentaa staattiset ja dynaamiset tiedot sen asennuksen yhteydessä selaimen välimuistiin, josta Service Worker hakee staattiset tiedot riippumatta verkkoyhteydestä. Service Worker tallentaa palvelimelta haettavat dynaamiset tiedot myös aina sivun renderöidessä. Eli Service Workerin välimuistin käyttöstrategiana dynaamisten tietojen kanssa käytetään Network-first-strategiaa. Tällä strategialla varmistetaan, että käyttäjälle näytetään mahdollisimman uusi tieto. Service Worker ei kykene tallentamaan tietoja välimuistiin offline-tilassa, vaan tiedot tallennetaan siinä tapauksessa selaimen paikalliseen IndexedDB-tietokantaan.

2.1 ASP.NET

Microsoft kehitti teknologian ASP (Active Server Pages) dynaamisten verkkosivujen luomiseen versioon ASP 3.0 asti, minkä jälkeen kehitettiin .NET-alusta ohjelmistokehittäjien uusiin oliopohjaisiin kehitystarpeisiin (Evjen ym. 2008, luku 1: Introduction). .NET-alusta on avoimen lähdekoodin alusta ohjelmistokehitykseen. Se koostuu useista työkaluista, kirjastoista ja ohjelmointikielistä, joiden myötä alusta tarjoaa laajat mahdollisuudet eri sovellusten rakentamiseen. ASP.NET on lisäys perusalustaan .NET, joka laajentaa mahdollisuuksia verkkosovellusten rakentamiseen. ASP.NET-sovelluksia voidaan käyttää ja kehittää Windows-ympäristön lisäksi myös Linuxissa, iOS:ssa ja Dockerissa. ASP.NETissä palvelinpuolen ohjelmointikielenä käytetään C#:a, F#:a tai Visual Basicia ja selainpuolelle ASP.NET tarjoaa Razor-mallinnussyntaksin. Razor mahdollistaa dynaamisten verkkosivujen luomisen sekoittaen eri kieliä, kuten JavaScriptia, HTML:ää ja C#:a. ASP.NET integroituu JavaScript-kirjastoihin ja -kehyksiin kuten Reactiin ja Angulariin. Se tarjoaa valmiita projektimalleja lisäosien käyttöön jo projektin luomisvaiheessa. (Microsoft 2021d.)

ASP.NET Core on uudempi versio ASP.NET-kehyksestä. Ratcliffe kuvaa (2018, 39) ASP.NET Coren olevan paljon kevyempi ja modulaarisempi kuin ASP.NET. Lisäksi hän kuvaa näiden tärkeimmän eron koskevan HTTP-pyyntöjen käsittelyä. ASP.NET Core -arkkitehtuurissa on väliohjelmistokomponentteja (middleware), jotka ovat ketjutettu toisiinsa (pipeline). Ketjutuksella ohjataan sitä, miten sovelluksen tulisi käsitellä jokaista HTTP-pyyntöä. Jokaisella väliohjelmistokomponentilla on mahdollisuus tehdä jotain saapuvien pyyntöjen kanssa, kuten käsitellä virheitä. (Ratcliffe 2018, 39.) Väliohjelmistokomponenttien ketjutus on havainnollistettu kuvassa 1.



Kuva 1. ASP.NET Core väliohjelmistojen muodostama ketju (Ratcliffe 2018, 40)

ASP.NET Core -sovellus generoi projektin luomisvaiheessa Startup-luokan, jossa on metodit nimeltä Configure ja ConfigureServices (Ratcliffe 2018, 41). Configure-metodissa rekisteröidään kaikki sovelluksen väliohjelmistokomponentit, kuten UseRouting ja UseEndpoints, jotka vastaavat HTTP-pyyntöjen reitityksestä (Microsoft 2020a). ConfigureServices-metodissa nimensä mukaisesti konfiguroidaan ja rekisteröidään sovelluksen palvelut (Ratcliffe 2018, 42). Microsoft on kehittänyt yksisivuisille sovelluksille SpaService-väliohjelmiston, jonka käyttäminen ASP.NET Core -projektissa on yksi tapa luoda yksisivuinen sovellus React-kirjaston kanssa (Microsoft 2021a).

2.2 Entity Framework

Entity Framework on olio-relaatio-mallinnus-kehys (ORM), joka luo yhteyden sovellusten ja tietokannan väliin. Entity Framework on kehitetty .NET-alustalle helpottamaan sovelluskehitystä, kun sovellus on vuorovaikutuksessa relaatiotietokantojen kanssa. (Microsoft 2020b.) Relaatiotietokantaan tallennetaan yksinkertaisia tietoja, jotka ovat riippumattomia sovelluksesta. Yksinkertaiset relaatiotiedot tallennetaan taulukoihin, kun taas oliopohjainen tietokanta mahdollistaa monimutkaisen tiedon tallennuksen. Oliopohjaiseen tietokantaan tiedot tallennetaan olioina. Oliotiedoilla voi olla yksinkertaisten tietojen lisäksi toimintoja ja ne voivat olla riippuvia sovelluksen toiminnasta. (Parahar 2019.) ORM-kehys vähentää eroja relaatiotietojen ja oliotietojen välillä sekä luo yhteyden tietokantapalvelimeen (Microsoft 2020b).

Entity Framework Core on modernimpi, kevyempi ja lisäosilla laajennettavissa oleva versio Entity Frameworkista. Entity Framework Coressa on ominaisuuksia, joita ei enää tulla kehittämään Entity Frameworkiin. Tämän johdosta Microsoft suosittaa käyttämään uusissa sovelluksissa Entity Framework Corea. (Microsoft 2021b.) Entity Framework Corella voidaan luoda yhteys jo olemassa olevaan tietokantaan database-first-menetelmällä tai tietokanta voidaan luoda code-first-menetelmällä perustuen sovelluksessa käytettyihin luokkiin (EntityFrameworkTutorial.net 2020).

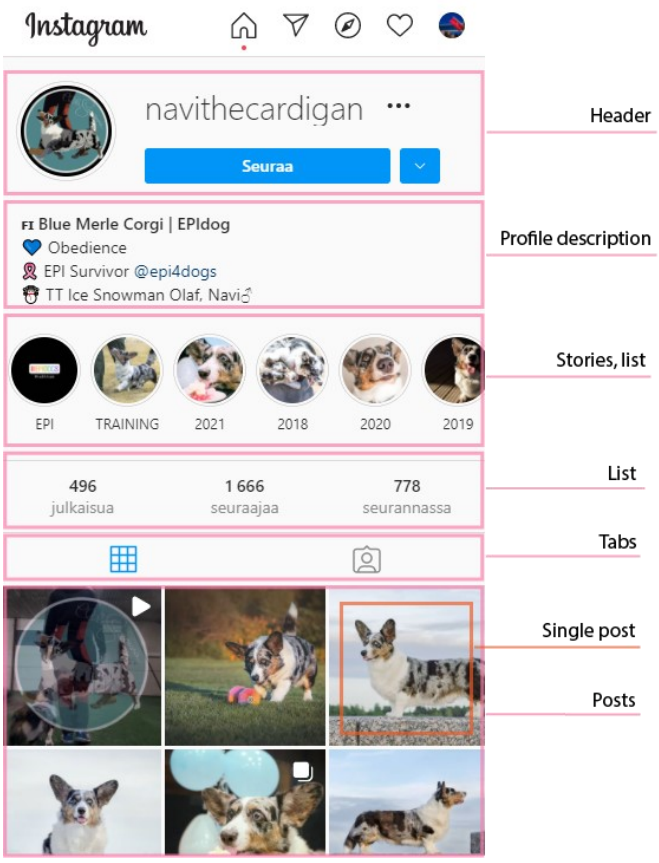
2.3 SQL Server

SQL Server on Microsoftin relaatiotietokantojen hallintajärjestelmä, joka hallitsee ja tallentaa muiden sovellusten pyytämiä tietoja. Sovellukset voivat toimia samassa tai eri tietokoneessa. SQL Server tukee useita sovelluksia ja tarjoaa monia muitakin toimintoja kuin tietokannan hallinnan, kuten BI- ja analytiikkatyökaluja. (Internap Corporation 2019.) Tietokanta koostuu data- ja lokitiedostoista. Lokitiedostossa saatavilla on tietokannan viimeisimmät muutokset. SQL Server kirjoittaa tapahtuman tiedot lokitiedostoon ennen varsinaisen muutoksen tekemistä tietokantaan. Jokaista muutosta ei kirjata tietokantaan suorituskyvyn vuoksi. Vasta kun tietty tarkistuspiste on saavutettu, valmis tapahtuma kirjoitetaan edelleen tietokantaan. Mikäli tietokanta kaatuu odottamattomasti, SQL Server palauttaa tiedot automaattisesti lokitiedoston avulla. (Microsoft 2021c; Masood-Al-Farooq 2014, 40–44.)

Tietokantatauluja käytetään tietojen tallentamiseen ja järjestämiseen relaatiotietokannassa. Taulu muodostuu riveistä ja sarakkeista. Taulun yksi rivi kuvaa yhtä taulun tietuetta ja sarake kuvaa yksittäisen tietueen ominaisuutta. Jokaisella sarakkeella on oma tietotyyppi, joka tunnistaa tiedon tallentamiseen käytettävän muodon. Tietokannan rakenteen suunnittelussa tärkeä osuus on taulujen suunnittelu. Taulujen suunnittelussa tunnistetaan taulujen – eli entiteettien – väliset suhteet ja sarakkeina esiintyvät attribuutit. Esimerkki attribuutista on tietueen ID-numero, kuten asiakas-ID. Attribuuttien avulla viitataan tiettyihin tietoihin entiteettien välillä. Viitetietojen eheyden varmistamiseen käytetään rajoitemäärityksiä, kuten esimerkiksi perusavain (primary key) ja viiteavain (foreign key). (Masood-Al-Farooq 2014, 33–37.)

2.4 React

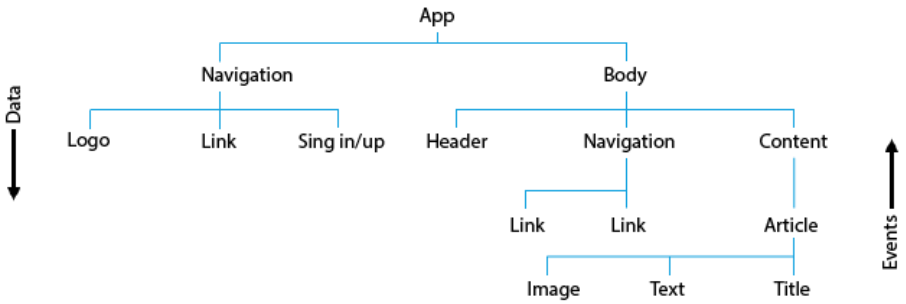
React on avoimen lähdekoodin JavaScript-kirjasto, jolla rakennetaan interaktiivisia käyttöliittymiä komponenttipohjaisesti. React vastaa sovellusarkkitehtuurin näkymäkerroksesta. (Chiarelli 2018, 20–24.) Facebook esitteli Reactin vuonna 2010 ja vuonna 2013 se julkaistiin. Facebookin lisäksi Reactia käyttävät esimerkiksi Instagram, Netflix, WhatsApp, YLE ja Veikkaus. Reactia käytetään erityisesti yksisivuisten käyttöliittymien kehittämiseen, jossa käyttöliittymän elementit ovat eritelty omiksi komponenteikseen. Komponentit lataavat näkymät perustuen käyttäjän toimiin. (Tecci 2020.) Kuvassa 2 on havainnollistettu Instagramin käyttöliittymän rakentuminen React-komponenteista.



Kuva 2. Käyttöliittymän rakentuminen React-komponenteista (Chiarelli 2018, 22)

Kuvaan 2 merkityt laatikot osoittavat React-komponentit käyttöliittymässä. Komponentit tekevät koodista uudelleenkäytettävän ja helposti jatkokehitettävän sekä mahdollistavat sovellusprojektin jakamisen pienempiin osiin (Sufiyan 2021; Tecci 2020). Kuvassa 3 on havainnollistettu puudiagrammin avulla kom-

ponenttien välisiä suhteita. React käyttää yksisuuntaista datavirtaa, jolloin alikomponentteja (child) sijoitetaan vanhempikomponentin (parent) sisään. Näin tieto kulkee yhteen suuntaan ja virheenkorjaus on helpompaa. (Sufiyan 2021.)



Kuva 3. Komponentit puudiagrammissa (Facebook Open Source 2021e)

Kuvassa 3 on nuolin osoitettu datan ja tapahtumien (events) kulkusuunta ali- ja vanhempikomponenttien välillä. Tapahtumien avulla voidaan tietoa välittää myös alikomponentilta vanhempikomponentille (Facebook Open Source 2021e).

React-komponentit reagoivat käyttäjän toimiin interaktiivisesti ja esittävät muuttuvat tiedot heti, kun muutoksia tapahtuu. Interaktiivisuuden mahdolliseksi tekee virtuaalinen DOM (Document Object Model), joka vertaa komponenttien aikaisempia tilatietoja uusiin tietoihin ja päivittää vain muuttuneet tiedot. (Chiarelli 2018, 26–28; Sufiyan 2021.) Kuvassa 4 esitetään esimerkki virtuaalisen DOM:n toiminnasta, joka renderöi React-sovelluksen elementit. Elementit kertovat mitä käyttöliittymässä pitää näyttää ja virtuaalinen DOM renderöi näkymän sovelluksen juuressa sijaitsevaan HTML-tiedostoon, yleensä index.html (Chiarelli 2018, 13–26).

```
// index.html
<div id="root"></div>

// React-app
function Welcome(props){
    return <h1>Welcome {props.name}</h1>
}
const element = <Welcome name="Heli"/>
ReactDOM.render(element, document.getElementById("root"))
```

Kuva 4. Esimerkki React-komponentin virtuaalisesta DOM:sta ja propsien käytöstä

Reactissa käytetään tilattomia ja tilallisia komponentteja. Tilalliset komponentit - toisin sanoen luokkakomponentit - voivat ylläpitää ja hallita tilaa, ne myös

renderöidään erikseen. Tilattomia komponentteja kutsutaan myös funktionaalisiksi komponenteiksi. Tilattomat komponentit voivat saada tietoja muista komponenteista propsien (properties) avulla. (Sufiyan 2021.) Propsit tarkoittavat attribuutteina välitettäviä ominaisuuksia komponentilta toiselle (Facebook Open Source 2021a). Facebook Open Sourcen React-dokumentaatiossa (2021a) kuvataan komponentin olevan kuin JavaScript-funktio. Komponentille voidaan lähettää tietoja propseilla samaan tapaan kuin parametrin funktiolle, tämä on myös havainnollistettu kuvassa 4.

Funktionaalisissa komponenteissa tilamuuttujien kanssa voidaan käyttää Hookeja. Tilamuuttujan voi tallentaa Hookiin. Hookin avulla samaa tilamuuttujaa voidaan käyttää uudelleen ja se voidaan jakaa komponentin ulkopuolelle. Hookit ovat lisätty helmikuussa 2019 Reactin versioon nro 16.8 ja Facebook suosittaa käyttämään funktionaalisia komponentteja luokkakomponenttien sijasta. Luokkakomponenttien kanssa Hookeja ei voi käyttää, mutta Hookeilla voidaan korvata luokkakomponenteissa käytetyt ominaisuudet. Hookeja on kehitetty eri ominaisuuksin, kuten useMemo ja useEffect. (Facebook Open Source 2021b.) Tässä työssä tullaan käyttämään funktionaalisia komponentteja ja Hookeja. Kuvassa 5 on esitetty esimerkit Hookien käytöstä sekä funktionaalisten ja luokkakomponenttien rakenteiden erot.

```
// funktionaalinen komponentti
function Example() {
    const [count, setCount] = useState(0);
    return(
        <div>
            <button onClick={()=>setCount(count+1)}>Click</button>
        </div>
    );
}

// luokkakomponentti
class Example extends React.Component {
    constructor(props){
        super(props);
        this.state = {
            count:0
        };
    }
    render(){
        return(
            <div>
                <button
                onClick={()=>this.setState({count: this.state.count+1})}>
                    Click
                </button>
            </div>
        )
    }
}
```

Kuva 5. Tilamuuttujat eri React-komponenteissa

Luokkakomponentteja ja funktionaalisia komponentteja voidaan käyttää keskenään vanhempi- ja alikomponentteina ilman että komponentit tietävät toistensa tyyppiä (Facebook Open Source 2021d). React-komponentin elinkaaren tapahtumia käsitellään useEffect-Hookilla. Tämä Hook mahdollistaa tapahtumien käsittelyn, kun elementit lisätään DOMiin, kun komponentteja päivitetään ja juuri ennen kuin komponentti poistuu DOM:sta. (Patel 2020.)

Yleensä React-koodi kirjoitetaan XML-tyyppiselle JSX-syntaksille, joka on Facebookin kehittämä JavaScriptin syntaksin laajennus (Facebook Open Source 2021c). React-sovelluksen kehittämiseen usein käytetään Node Package Manager -paketinhallintatyökalua, josta yleisesti käytetään lyhennettä NPM. NPM on kirjoitettu JavaScriptillä ja sen avulla voidaan asentaa lisäosia esimerkiksi React-sovellukseen. Lisäksi se muodostaa (bundles) React-sovelluksen lähdekoodista yhden tiedoston, joka voidaan esimerkiksi Webpackin avulla kääntää selaimen ymmärtämään muotoon. (Kasundra 2021.)

2.5 Progressiivinen verkkosovellus (PWA)

Progressiivinen verkkosovellus (Progressive Web Application), lyhennettynä PWA, toimii verkkosivuna selaimessa ja se voidaan tallentaa laitteeseen kuin natiivisovellus. Responsiivisuuden johdosta progressiivista verkkosovellusta voidaan käyttää eri laitteilla. Verkkosovellusta voidaan pitää progressiivisena verkkosovelluksena sen täyttäessä tietyt vaatimukset ja noudattaessa tiettyjä periaatteita. (MDN Web Docs 2021b.)

MDN Web Docsin (2021b) dokumentaatiossa on kuvattu seuraavat vaatimukset progressiivisen verkkosovelluksen tunnistamiseksi: Sisällön tulee löytyä hakukoneiden kautta, sovelluksen tulee olla asennettavissa laitteeseen, jaettavissa URL-osoitteella, toimia offline-tilassa, toimia vanhoilla sekä uusilla selaimilla, olla responsiivisesti suunniteltu, suojattu ja uudelleenkäytettävä. Vaatimukset eivät ole virallisia, mutta auttavat määrittämään mikä on progressiivinen verkkosovellus.

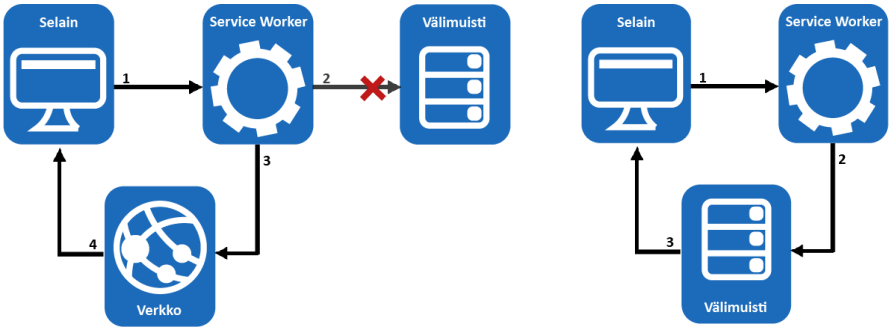
MDN Web Docs (2021a) mukaan progressiivinen verkkosovellus vaatii toimiakseen seuraavat asiat:

- Manifest-JSON-tiedoston, jossa esitetään sovelluksen tiedot kuten kuvakkeen koot ja sovelluksen nimi
- kuvakkeen, joka näkyy laitteessa, kun sovellus on asennettu
- suojatun HTTPS-protokollan käytön
- rekisteröidyn Service Worker -koodin.

Sovelluksen toiminnan verkkoyhteydettömässä tilassa mahdollistaa Service Worker -koodi ja selaimen välimuisti. Service Workerin käyttöön on kehitetty työkaluja ja kirjastoja, joita käytetään parantamaan sovelluksen offline-toimivuutta, kuten välimuistin hallintaa. Googlen Workbox on kokoelma näitä kirjastoja ja se on integroitu valmiiksi muun muassa Reactin sovellusmalliin. (Posnick 2018.)

Service Worker tarkoittaa JavaScript-koodia, joka toimii välityspalvelimena selaimen ja verkon välillä. Se toimii selaimen taustalla erillään sovelluksesta, eikä sillä ole pääsyä DOM-rakenteeseen. Service Worker voi hallita ja muokata pyyntöjä (request) sekä tarjota välimuistista muokattuja vastauksia (response). Tämän vuoksi Service Worker -koodin voi suorittaa vain salatun HTTPS-yhteyden avulla. (MDN Web Docs 2021b.)

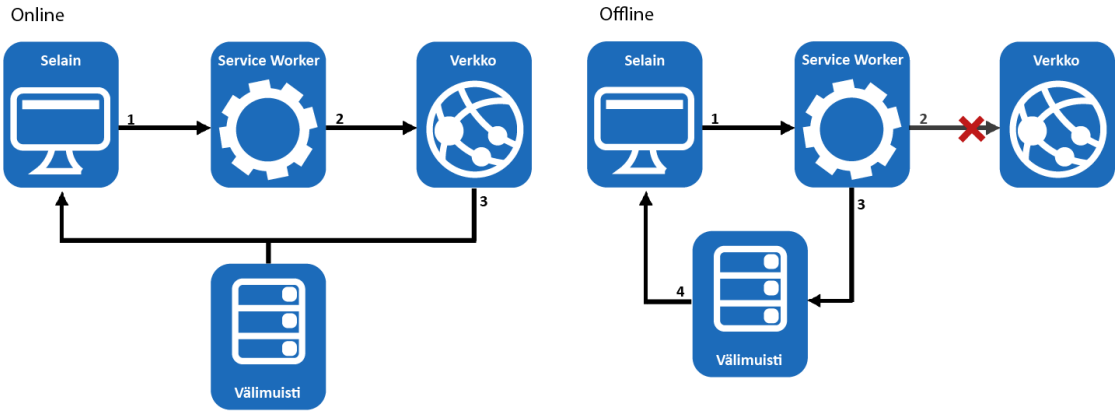
Välimuistin käyttöstrategioita on muutamia erilaisia, joista yleisin on Cache-first-strategia. Cache-first-strategiassa tieto haetaan välimuistista ennen kuin otetaan yhteys verkkoon. Jos tieto löytyy välimuistista, ei yhteyttä oteta verkkoon ollenkaan. Kyseinen strategia mahdollistaa sovelluksen käytön huonolla verkkoyhteydellä tai verkkoyhteydettömässä tilassa hyvin nopeasti ja tehokkaasti, mutta käyttäjälle näytettävä tieto ei välttämättä ole täysin ajantasaista. (Archibald 2014.) Cache-first-strategia on havainnollistettu kuvassa 6.



Kuva 6. Välimuistin käyttöstrategia Cache-first (Archibald 2014)

Kuvan 6 ensimmäisessä osassa vastausta haettavaan tietoon ei löydy vielä välimuistista, jolloin vastaus haetaan verkosta. Kuvan seuraavassa vaiheessa vastaus löytyy välimuistista.

Vastaavasti Network-first-strategiassa Service Worker yrittää hakea uusimman tiedon ensin verkosta. Jos pyyntö verkkoon onnistuu, tiedot tallennetaan välimuistiin. Jos pyyntö verkkoon epäonnistuu, tiedot haetaan välimuistista. Tämä strategia mahdollistaa mahdollisimman tuoreen tiedon esittämisen käyttäjälle, mutta saattaa aiheuttaa hitautta huonon verkkoyhteyden alueella. Käyttäjän on odotettava palvelimen epäonnistunut vastaus, ennen kuin Service Worker hakee tiedot välimuistista. (Archibald 2014.) Network-first-strategia on havainnollistettu kuvassa 7.

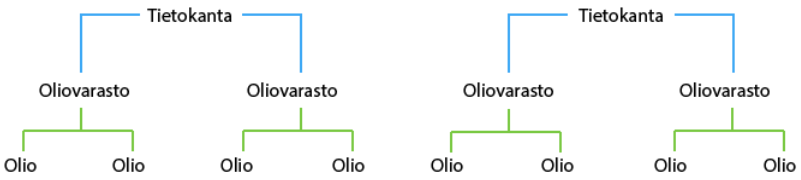


Kuva 7. Välimuistin käyttöstrategia Network-first (Archibald 2014)

Kuvan 7 ensimmäinen vaihe on esimerkki vastauksen hakemisesta sovelluksen ollessa yhteydessä verkkoon. Toinen vaihe kuvaa offline-tilaa, jolloin vastaus palautetaan välimuistista.

Progressiivisen verkkosovelluksen Service Worker -koodi hakee tietoja selaimen välimuistista perustuen selaimen pyyntöihin, mutta ilman verkkoyhteyttä Service Worker ei kykene tallentamaan tietoja välimuistiin. Verkkoyhteydettömässä tilassa se voi käyttää selaimen paikallista IndexedDB-tietokantarajapintaa ja tallentaa esimerkiksi selaimen POST-pyynnöt offline-tilassa sinne. (MDN Web Docs 2021b.) IndexedDB on NoSQL-tietokanta, joka tukee monia tietotyyppejä ja tallentaa avain-arvo-parit (key-value pairs). Se ei ole relaatiotietokanta, joka perustuu taulukoihin ja jonka sarakkeet edustavat eri tietotyyppejä, vaan IndexedDB edellyttää tietotyypeille oliovaraston luomisen. Yhdellä

oliovarastolla voi olla useita olioita ja yhdellä sovelluksella voi olla useita tietokantoja. (MDN Web Docs 2021c.) Rakenne on havainnollistettu kuvassa 8.

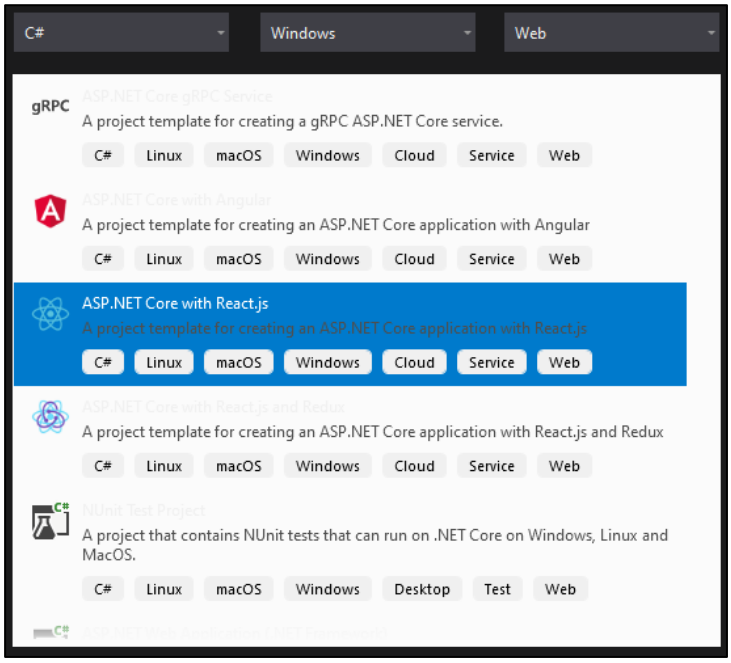


Kuva 8. Esimerkki IndexedDB-tietokantarajapinnan rakenteesta (MDN Web Docs 2021c)

Jokaisella oliovarastolla tulee olla yksilöllinen nimi. Oliovarasto pitää kirjaa olioista avain-arvo-pareilla, jotka lajitellaan avainten mukaan nousevassa järjestyksessä. Olion tietoja voi käsitellä avaimen avulla. Tapahtumapohjaisuuden vuoksi IndexedDB:n oliovarastoon tallennetuiden tietojen kanssa voi olla vuorovaikutuksessa vain transaktioiden kautta. (Hiwarale 2018.) Tietokanta on asynkroninen eli tietokannalta pyydetään tietokantaoperaatioiden tapahtumista. Operaation päätyttyä DOM-tapahtuma ilmoittaa onko operaatio onnistunut vai epäonnistunut. (MDN Web Docs 2021c.) Asynkronisuuden ja yleisimpien selainten tuen vuoksi IndexedDB sopii hyvin sovelluksen offline-käytön tietojen tallennukseen (Hiwarale 2018).

3 SOVELLUKSEN TOTEUTUS

Työssä kehitettävään verkkosovellukseen kuuluu selainpuolen ja palvelinpuolen sovellukset. Palvelinsovellus toimii ASP.NET-alustalla ja se käyttää C#-kieltä sekä Entity Framework -ORM-kehystä. Entity Framework avaa yhteyden relaatiotietokantaan. Tietokanta toimii Microsoftin SQL Serverillä. Selainpuolen sovellus käyttää JavaScriptin React-kirjastoa sekä Reactille kehitettyä Material-UI-käyttöliittymäkomponenttikirjastoa. Material-UI tarjoaa muokattavissa olevia valmiita käyttöliittymäkomponentteja, kuten painike (button) ja Grid-laatikot responsiivisuuden tueksi. Projekti aloitettiin Microsoftin Visual Studio -ohjelmointiympäristön tarjoamalla projektimallilla ’ASP.NET Core with React.js' (kuva 9).



Kuva 9. Projektin aloitus Visual Studio -työkalulla

Projektimallin ASP.NET Core -sovellus tarjoaa selainsovelluksen käyttämän REST API -rajapinnan. Projektimalli sisältää React-sovelluksen omassa kansiossaan, joka on määritetty toimimaan Microsoftin SpaService-väliohjelmiston avulla ASP.NET Core -palvelinsovelluksen rinnalla. Kehitystyössä ASP.NET Core vastaa selainsovelluksen käynnistämisestä luomalla palvelinsovelluksen ja selainsovelluksen välille välityspalvelimen. Sovelluksia voidaan näin kehittää yhtenä kokonaisuutena. Mikäli palvelinsovelluksen koodia muokataan, myös selainsovellus tulee käynnistää uudestaan.

3.1 Palvelinsovellus

ASP.NET Core -sovellus käynnistyy Main-metodissa luokassa nimeltä Program. Main-metodissa käynnistetään saman luokan toinen metodi nimeltä CreateHostBuilder, joka luo verkkopalvelimen ja määrittää Startup-luokan. Projektin luomisvaiheessa Startup-luokassa generoituu automaattisesti React-sovelluksen kehitykseen käytettävä SpaService-väliohjelmiston konfigurointi. Kuvassa 10 esitetään Startup-luokan Configure-metodi.

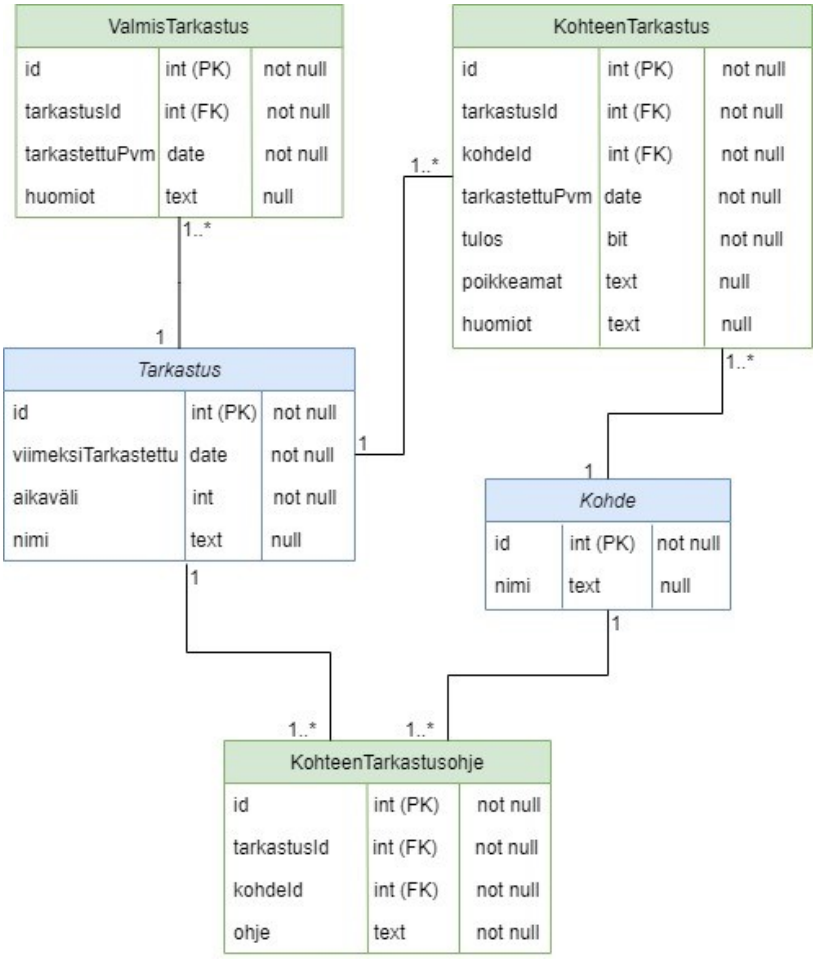
```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())... // err
    else...

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseSpaStaticFiles();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller}/{action=Index}/{id?}");
    });

    app.UseSpa(spa =>
    {
        spa.Options.SourcePath = "reactapp";

        if (env.IsDevelopment())
        {
            spa.UseReactDevelopmentServer(npmScript: "start"); // reactapp: package.json, start
        }
    });
}
```

Kuva 10. Startup-luokan metodi väliohjelmistojen konfiguroimiseksi

Kuvassa 10 esiintyvä Configure-metodi määrittää mitä väliohjelmistoja suoritetaan ja missä järjestyksessä. Kun palvelinsovellus vastaanottaa pyynnön, se kulkee väliohjelmistojen kautta. Selainsovelluksen käyttö palvelinsovelluksen rinnalla määritetään UseSpaStaticFiles-metodilla ennen reititysten ja päätepisteiden (endpoint) määrittämistä. Pyyntö selainsovellukselle kulkee palvelinsovelluksen kautta. Configure-metodin lisäksi samassa luokassa on metodi sovelluksen käyttämien väliohjelmistojen rekisteröimiseksi, jota käsitellään myöhemmin.

3.1.1 Tietokantataulut

Palvelinsovelluksen kehittäminen aloitettiin luomalla sovelluksen käyttämä tietokanta ja tietokantataulut SQL Server Management Studiossa. Tietokantataulut suunniteltiin vastaamaan sovelluksen tavoitteellista käyttötarkoitusta. Tauluihin lisättiin geneerisiä tietoja sovelluksen kehitystyötä varten. Kuvassa 11 on esitetty tietokantataulut, niiden väliset suhteet ja tietotyypit.



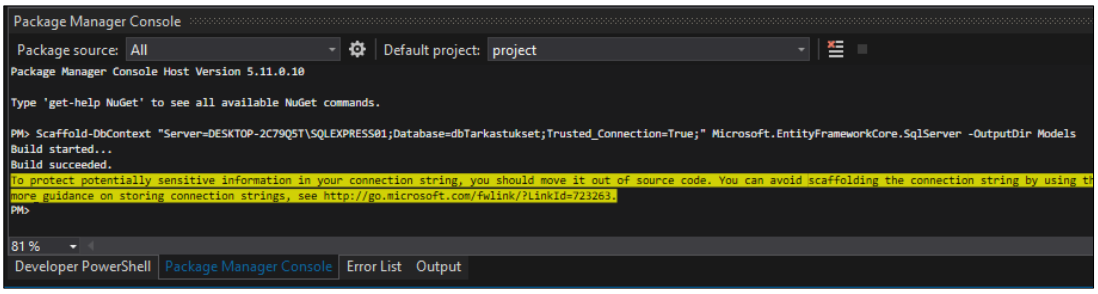
Kuva 11. Tietokantataulut

Jokaisella taululla on perusavaimena id-arvo. Perusavaimet on kuvaan 11 merkitty lyhenteillä PK (primary key). Lapsitaulut on kuvaan merkitty vihreällä värillä ja vanhempitaulut vastaavasti on merkitty sinisellä. Lapsitaulujen ja vanhempitaulujen väliset yhteydet on luotu viiteavaimella, joka on merkitty kuvaan lyhenteellä FK (foreign key). Taulujen väliset suhteet menevät seuraavasti. Yhdellä vanhempitaulun esiintymällä voi olla yhteys useisiin yhden lapsitaulun esiintymiin. Yhdellä lapsitaulun esiintymällä voi olla yhteys vain yhteen sen vanhempitaulun esiintymään. Suhteet ovat one-to-many-suhteita. Tietokannan luomisessa ei tarkemmin otettu huomioon viite-eheyksiä, sillä opinnäytetyössä kehitettävä sovellus ei poista tietoja tietokannasta ja päivitettäviä tietoja ei juurikaan ole. Sovellus tulee pääasiassa aina luomaan uuden tietueen lapsitauluihin.

3.1.2 Entity Framework -metodit

ASP.NET Core -sovellus käsittelee tietokannan tietoja olioina luokkien avulla, kun taas luotu tietokanta on relaatiotietokanta. Relaatio- ja olioteknologioiden

välillä on eroja, joiden väliseen vuorovaikutukseen tässä työssä käytetään Entity Framework Core -ORM-kehystä. Entity Framework Core toimii ASP.NET Coressa väliohjelmistona, joka kommunikoi relaatiotietokannan kanssa. Tästä eteenpäin Entity Framework Coresta käytetään lyhennettä EF. EF asennetaan projektiin NuGet-pakettienhallintaohjelmalla. NuGet-hallintaohjelma on kehitetty Microsoftin kehitysympäristöille. Sen avulla voidaan luoda ja ottaa sovelluksen käyttöön lisäosia joko graafisen käyttöliittymän kautta tai hallintaohjelman komentorivillä. Scaffold-komennon avulla EF avaa yhteyden tietokantaan ja luo tietokantaan perustuen kontekstiluokan sekä malliluokat. Kontekstiluokka on johdettu EF:n DbContext-luokasta ja sitä voidaan käyttää malliluokkien entiteettien kyselyihin ja tallennukseen. Kuvassa 12 on havainnollistettu NuGet-hallintaohjelman komentorivin käyttö ja Scaffold-komennon käyttö aiemmin mainittujen toimien tekemiseksi.



Kuva 12. Yhteys tietokantaan ja luokkien generoiminen Entity Framework Coren avulla

Scaffoldia käytetään EF:n kanssa luomaan koodia automaattisesti tietokantarakenteeseen perustuen, kun toimitaan tietokanta ensin -menetelmällä. Komentoon sisältyy yhteysmerkkijono (connection string), joka määrittää yhteyden muodostamisen tietokantaan. Kuvassa 12 on maininta keltaisella yhteysmerkkijonon siirtämisestä muualle lähdekoodista, sillä yhteysmerkkijono sisältyy automaattisesti generoidun kontekstiluokan koodiin. Yhteysmerkkijono siirretään juurikansion appsettings.json-tiedostoon säilytystä varten.

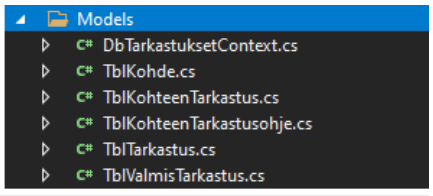
Kuvassa 13 esitetään osa EF:n DbContext-luokasta johdetusta kontekstiluokasta.

```
namespace project.Models
{
    14 references
    public partial class DbTarkastuksetContext : DbContext
    {
        0 references
        public DbTarkastuksetContext()
        {
        }
        0 references
        public DbTarkastuksetContext(DbContextOptions<DbTarkastuksetContext> options)
            : base(options)
        {
        }
        4 references
        public virtual DbSet<TblKohde> TblKohde { get; set; }
        4 references
        public virtual DbSet<TblKohteenTarkastus> TblKohteenTarkastus { get; set; }
        4 references
        public virtual DbSet<TblKohteenTarkastusohje> TblKohteenTarkastusohje { get; set; }
        4 references
        public virtual DbSet<TblTarkastus> TblTarkastus { get; set; }
        4 references
        public virtual DbSet<TblValmisTarkastus> TblValmisTarkastus { get; set; }
        0 references
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.HasAnnotation("Relational:Collation", "Finnish_Swedish_CI_AS");
            modelBuilder.Entity<TblKohde>(entity =>
            {
                entity.ToTable("tblKohde");
                entity.Property(e => e.Id).HasColumnName("id");
                entity.Property(e => e.Nimi)
                    .HasColumnType("text")
                    .HasColumnName("nimi");
            });
```

Kuva 13. Osa Entity Framework Coren generoimasta kontekstiluokasta

Kontekstiluokka kuvaa relaatiotietokantaa. Kuvassa 13 kontekstiluokasta käy ilmi DbSet-luokat kullekin entiteetille. Entiteettien esiintymien kyselyt ja tallennukset suoritetaan DbSet-luokalla. DbSet käyttää LINQ-kyselyitä, jotka sovellus kääntää tietokannan ymmärtämiksi SQL-kyselyiksi. Kuvassa 13 näkyy myös esimerkki entiteetin mallin määrittämisestä, joka suoritetaan sovelluksen käyttäessä ensimmäistä kertaa vastaavan entiteetin esiintymää. Kuvassa 14 on esimerkki malliluokasta ja sen luokkamuuttujista.

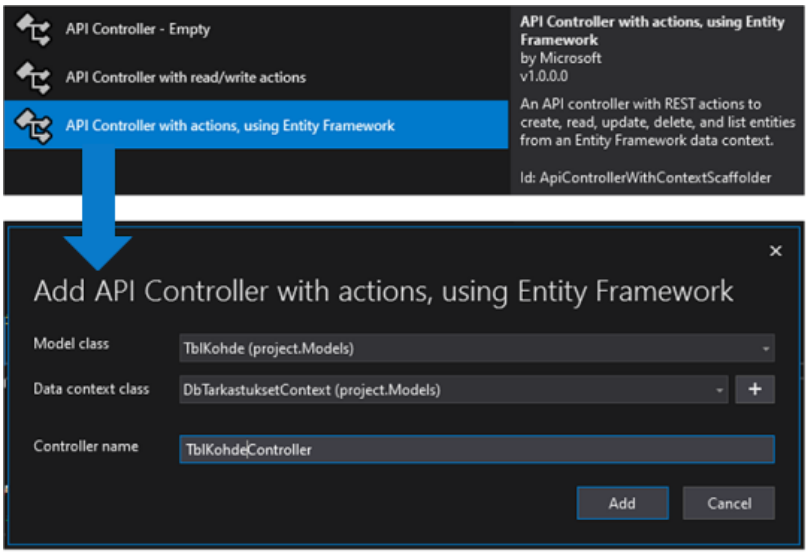
```
namespace project.Models
{
    11 references
    public partial class TblKohteenTarkastusohje
    {
        4 references
        public int Id { get; set; }
        2 references
        public int TarkastusId { get; set; }
        2 references
        public int KohdeId { get; set; }
        1 reference
        public string Ohje { get; set; }
        1 reference
        public virtual TblKohde Kohde { get; set; }
        1 reference
        public virtual TblTarkastus Tarkastus { get; set; }
    }
}
```



Kuva 14. Entity Framework Coren generoimat malliluokat

Kuvassa 14 esitetään EF:n luomat malliluokat ja kontekstiluokka projektin Models-kansiossa sekä esimerkki yhdestä näistä tietokannan pohjalta generoiduista malliluokista. Yksi malliluokka kuvaa yhtä entiteettiä eli relaatiotietokannan taulua.

Jotta kontekstiluokka ja malliluokka toimisivat vuorovaikutuksessa selainpuolen sovelluksen kanssa, luodaan palvelinsovellukseen API-käsittelijät (Controller). API-käsittelijät toimivat väliohjelmistona palvelinsovelluksen ja selaimelta tulevien HTTP-pyyntöjen välissä. API-käsittelijät luodaan EF:n Scaffold-menetelmällä. Kuvassa 15 on esitetty esimerkki API-käsittelijän lisäämisestä projektin Controllers-kansioon.



Kuva 15. API-käsittelijän lisääminen projektiin Scaffold-menetelmällä

Kuvassa 15 API-käsittelijä luodaan vastaamaan TblKohde-malliluokkaa. Malliluokan kontekstiluokka on se kontekstiluokka, jossa on määritetty malliluokan entiteetin ominaisuuksia. Tässä työssä kontekstiluokka on joka malliluokalle sama, sillä käytössä on yksi tietokanta.

API-käsittelijällä on HTTP-pyyntöjen reitit ja se on yhteensopiva CRUD-operaatioille. API-käsittelijät ovat johdettu ControllerBase-kantaluokasta, joka on peräisin ASP.NET Core MVC -suunnittelumallista. Luokan alussa määritetään kontekstiluokka, minkä jälkeen toimintomenetelmät (action method) tiettyjen HTTP-pyyntöjen ja reittien käsittelemiseksi. Kuvassa 16 näkyy jokaista entiteettiä vastaamaan luodun API-käsittelijän tiedostonnimi sekä esimerkki API-käsittelijän koodista.


```
namespace project.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class TblKohdeController : ControllerBase
    {
        private readonly DbTarkastuksetContext _context;
        0 references
        public TblKohdeController(DbTarkastuksetContext context)
        {
            _context = context;
        }
        // GET: api/TblKohde
        [HttpGet]
        0 references
        public async Task<ActionResult<IEnumerable<TblKohde>>> GetTblKohde()
        {
            return await _context.TblKohde.ToListAsync();
        }
        // GET: api/TblKohde/5
        [HttpGet("{id}")]
        0 references
        public async Task<ActionResult<TblKohde>> GetTblKohde(int id)
        {
            var tblKohde = await _context.TblKohde.FindAsync(id);

            if (tblKohde == null)
            {
                return NotFound(); // 404
            }
            return tblKohde;
        }
        // PUT: api/TblKohde/5
        [HttpPut("{id}")]
        0 references
        public async Task<IActionResult> PutTblKohde(int id, TblKohde tblKohde)
        {
            if (id != tblKohde.Id)
            {
                return BadRequest(); // 400
            }
            _context.Entry(tblKohde).State = EntityState.Modified;

            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!TblKohdeExists(id))
                {
                    return NotFound(); // 404
                }
                else
                {
                    throw;
                }
            }
            return NoContent(); // 204
        }
        // POST: api/TblKohde
        [HttpPost]
        0 references
        public async Task<ActionResult<TblKohde>> PostTblKohde(TblKohde tblKohde)
        {
            _context.TblKohde.Add(tblKohde);
            await _context.SaveChangesAsync();

            return CreatedAtAction("GetTblKohde", new { id = tblKohde.Id }, tblKohde);
        }
        1 reference
        private bool TblKohdeExists(int id)
        {
            return _context.TblKohde.Any(e => e.Id == id);
        }
    }
}
```

Kuva 16. Yhden entiteetin API-käsittelijä selainsovelluksen HTTP-pyynnöille

PUT-metodi käyttää kuvassa 16 esiintyvän koodin Private-suojausmääreellä määritettyä Exists-metodia, joka palauttaa boolean-arvon sen perusteella onko vastaavan id:n esiintymää olemassa. DELETE-metodia ei kuvassa 16 näy, sillä se on tässä työssä tarpeeton.

Kuvassa 17 esitetään Startup-luokan ConfigureServices-metodi, jossa rekisteröidään Entity Framework Coren käyttö.

```
public void ConfigureServices(IServiceCollection services)
{
    // Connection string from appsettings.json
    var connection = Configuration.GetConnectionString("DbTarkastukset");
    // Kertoo ASP.NET, että käytetään EF Db Context
    services.AddDbContext<DbTarkastuksetContext>(options => options.UseSqlServer(connection));

    //Kertoo ASP.NET router, että käytetään API Controllers, React hoitaa näkymät
    services.AddControllers();

    // Kun production, React-sovelluksen tiedostot haetaan täältä
    services.AddSpaStaticFiles(configuration =>
    {
        configuration.RootPath = "reactapp/build";
    });
}
```

Kuva 17. Startup-luokan ConfigureServices-metodi

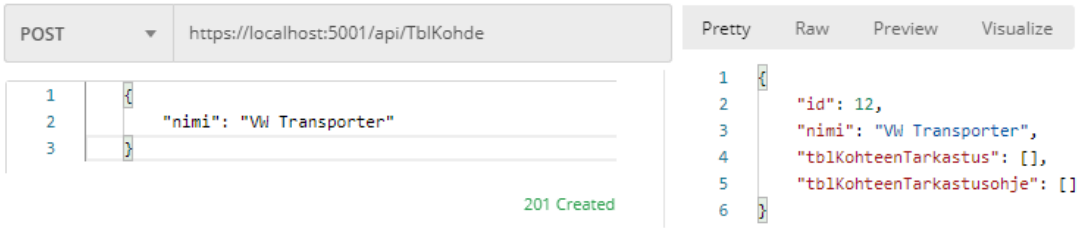
Kuvassa 17 esiintyvässä metodissa palvelinsovellukselle kerrotaan, että käytetään kontekstiluokkaa sekä SQL Serverin tietokantaa. Lisäksi kerrotaan API-käsittelijöiden olevan käytössä ja ettei palvelinsovelluksen tarvitse palauttaa näkymiä, vaan selainpuolen sovellus tulee tekemään sen.

3.1.3 HTTP-pyyntöjen käsittely

HTTP-pyyntöjen testaamiseksi käytettiin Postman-sovellusta. Metodit GET, POST ja PUT toimivat tällä konfiguroinnilla. POST- ja PUT-pyynnöt tullaan selainpuolen sovellukselta tekemään JSON-muotoisena pyynnön body-sisältöön. JSON-datan käsittelemiseksi Startup-luokkaan rekisteröidään ASP.NET Core MVC datan muotoilija (formatter) ja asetetaan asetukset tukemaan JSON-muotoa (Hobab 2019). Postman-sovellus palauttaa kaikille kokeiluille 200-alkuisen statuksen, eli palvelinsovellus toimii halutusti. Kuvissa 18, 19 ja 20 esitetään kokeilut Postman-sovelluksessa.

Kuva 18. GET-pyyntö Postman-sovelluksessa

GET-pyyntö reitillä ’api/TblKohde’ palauttaa TblKohde-tietokantataulun sisällön statuksella ’200 OK’ (kuva 18).



Kuva 19. POST-pyyntö Postman-sovelluksessa

POST-pyyntö samalla reitillä lisää pyynnön body-tietoihin JSON-muodossa annetun uuden tietueen TblKohde-tietokantatauluun statuksella ’201 Created’ (kuva 19).

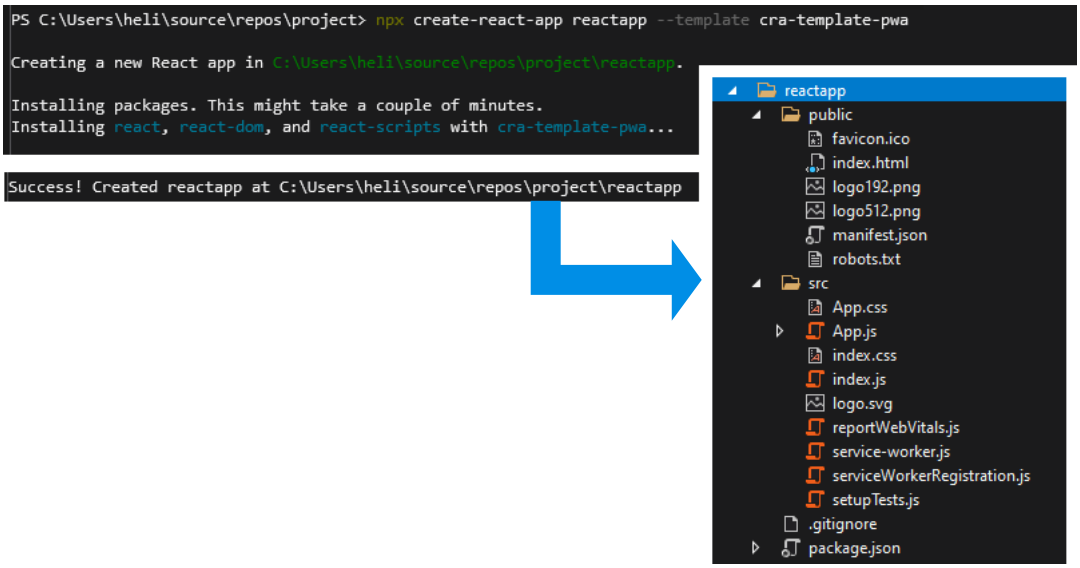
Kuva 20. PUT-pyyntö Postman-sovelluksessa

PUT-pyyntö reitillä ’api/TblKohde/id’, kuvassa id on 1, päivittää id:tä vastaavan tietueen tiedot pyynnön body-sisällöstä (kuva 20). Kuvassa 20 auton malli päivitetään ja esimerkin vuoksi id:tä vastaavan tiedon päivittyminen varmistetaan GET-pyynnöllä. PUT-pyyntö palauttaa ’204 No Content’, jonka on tarkoitus kertoa selainpuolen sovellukselle, että tieto on päivitetty ja sivun voi päivittää. PUT-pyynnön vastaukseksi ei palauteta sisältöä.

3.2 Selainsovellus

ASP.NET Coren luoman React-sovelluksen tilalle luotiin npx-komennolla uusi React-sovellus progressiivisen verkkosovelluksen (PWA) mallilla, jotta sovellus voisi käyttää Workbox-moduuleja PWA-ominaisuuksia toteuttaessa. Workbox-moduulien käyttäminen on konfiguroitu valmiiksi Reactin PWA-mallille.

Kuvassa 21 esitetään React-sovelluksen luominen komentoriviltä ja NPX (Node Package Execute) rakentama React-sovellus nimeltä ’reactapp’, joka rakentui projektikansion juureen.

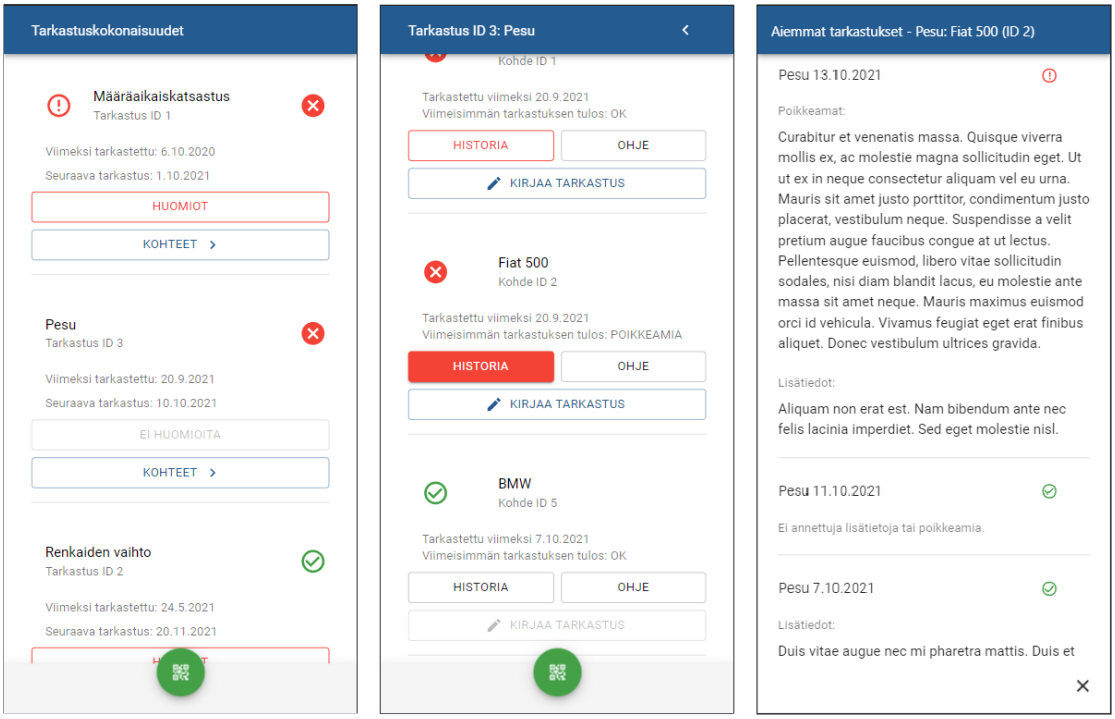


Kuva 21. Selainsovelluksen luominen PWA-mallille

Palvelinsovelluksen koodiin päivitettiin tämän uuden selainsovelluksen nimi ja polku. Varmistettiin, että package.json-tiedostossa on tarvittavat moduulit ja oikeat komennot sovelluksen käynnistämiseen ja rakentamiseen. Tässä vaiheessa projektia molemmat sovellukset toimivat ja käyttöliittymän rakentaminen selainsovellukselle aloitettiin.

3.2.1 Käyttöliittymän ulkoasu

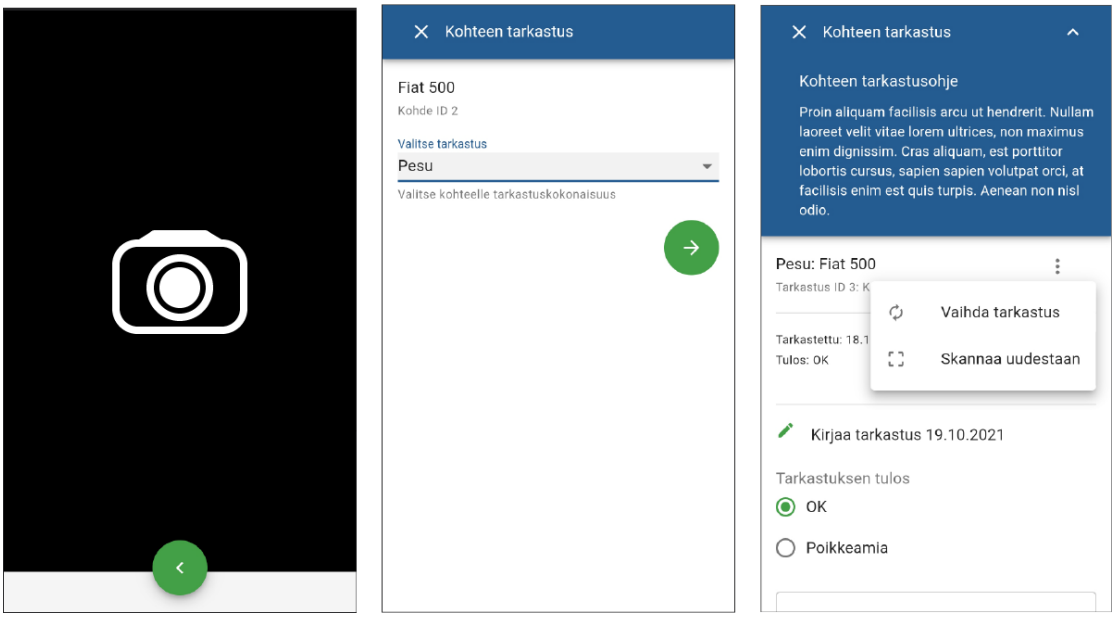
Käyttöliittymän rakentamisessa hyödynnettiin Material-UI-käyttöliittymäkomponentteja. Käyttöliittymässä on kaksi päänäkymää, toinen tarkastuskokonaisuuksille - kuten auton pesu - ja toinen tarkastuskokonaisuuteen kuuluville kohteille. Päänäkymissä tietokantataulujen tiedot esitetään mahdollisimman tiiviisti ja luettavat tiedot, kuten lisähuomiot ja ohjeet, on avattava erikseen painikkeella. Siten päänäkymät pysyvät yksinkertaisina, tärkeät tiedot ovat helposti nähtävillä ja lisätiedot ovat tarvittaessa saatavilla. Päänäkymät sekä lisätietojen esittämiseen käytettävän Dialogi-komponentin ulkoasu on esitetty kuvassa 22. Dialogi-komponentti esittää lisätiedot, jonka painikkeet näkyvät kuvassa nimillä ’Historia’, ’Huomiot’ ja ’Ohje’. Eri lisätietoja sovelluksessa ovat kohteen huomiot, poikkeamat ja ohje sekä tarkastuskokonaisuuden huomiot.



Kuva 22. Tarkastuskokonaisuudet, Kohteet ja Dialogi -näkymien ulkoasut

Kuvassa 22 esitetään kaikki mahdolliset tapaukset tarkastuskokonaisuuksien ja kohteiden tiloista. Tarkastuskokonaisuus-näkymään tiedot järjestetään aikajärjestykseen seuraavan tarkastuspäivämäärän perusteella. Kun tarkastuksen teko on myöhässä, näytetään tarkastuksen nimen vieressä huutomerkki-ikoni. Tarkastuskokonaisuuksien ja kohteiden tarkastusten tilanteesta ilmoitetaan vihreällä tai punaisella ikonilla, riippuen onko tarkastus tehty. Kohteista näytetään painike otsikolla ’Historia’, jonka ulkoasu määrittyy sen mukaan, onko kohteesta kirjattu poikkeamia. Punainen painike ilmaisee viime tarkastuksessa olleen poikkeamia. Kun viime tarkastuksessa ei havaittu poikkeamia, mutta lisätietoja on joskus kirjattu, painikkeen reuna on punainen. Mikäli kohteelle voidaan kirjata tarkastus, painike otsikolla ’Kirjaa tarkastus’ on aktiivinen.

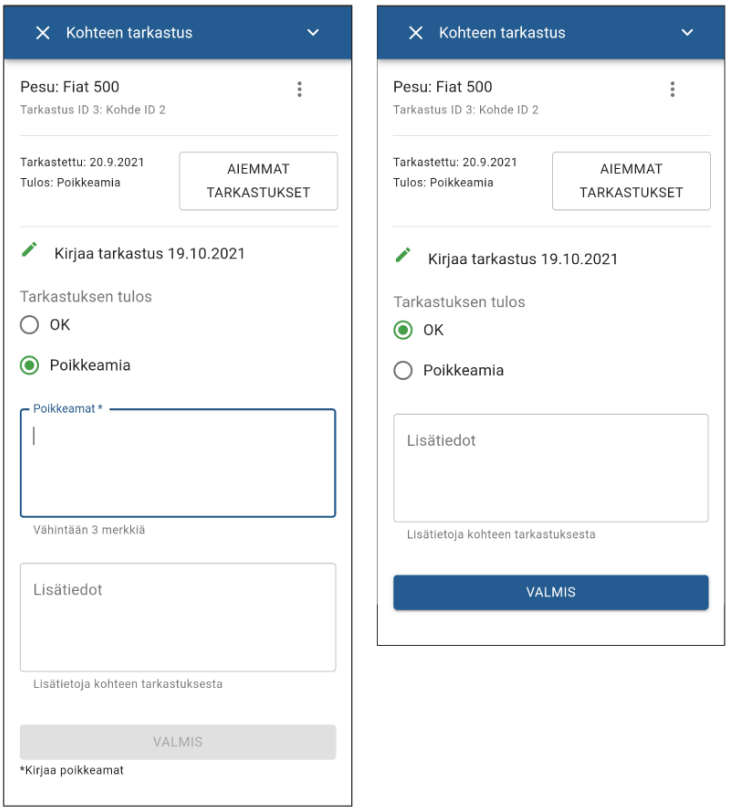
Mobiililaitteilla näytetään painike QR-koodilukijan avaamiseksi. Kuvassa 23 esitetään näkymiä, kun kohteen tarkastus kirjataan QR-koodilukijaa käyttämällä.



Kuva 23. QR-koodilukijan ja tarkastuksen valinnan näkymät sekä kohteen kirjausnäkymän toiminnallisuutta

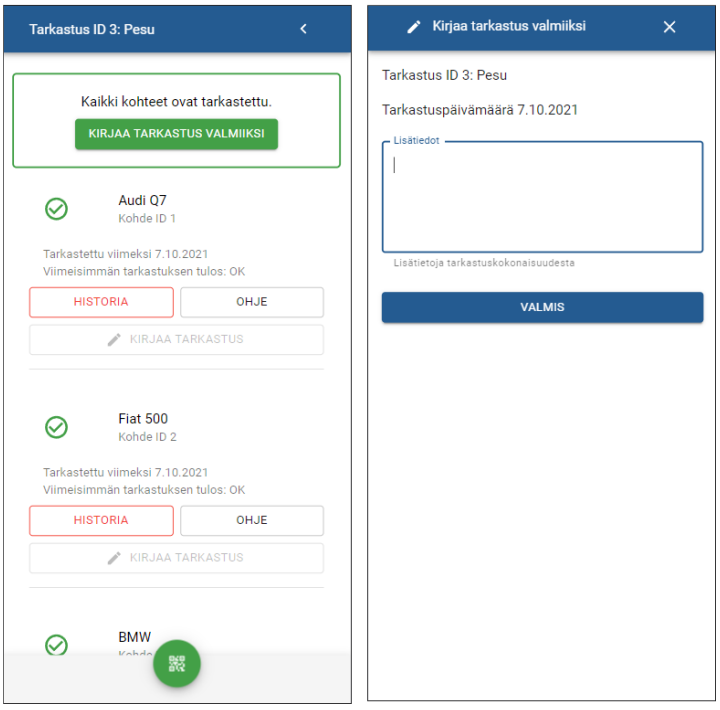
Kuvassa 23 ensimmäisenä on QR-koodilukijan kameranäkymä. QR-koodilukijalta avautuva näkymä tarkastuksen valitsemiselle on esitetty kuvassa keskimmäisenä. Kohteen kirjausnäkymään edetään tarkastuksen valitsemisen jälkeen nuolipainikkeesta. Mikäli kohdetta ei kirjata QR-koodin avulla, kohteen kirjausnäkymä avautuu kuvassa 22 (s. 27) esiintyvästä painikkeesta otsikolla ’Kirjaa tarkastus’. Kohteen kirjausnäkymän toiminnallisuutta esitetään kuvan 23 viimeisessä osassa.

Kohteen kirjausnäkymässä näytetään kohteen ja tarkastuskokonaisuuden nimet, id:t sekä viimeisimmän tarkastuksen päivämäärä ja tulos. Kirjausnäkymästä on mahdollista tarkastaa kohteen ja valitun tarkastuksen ohje Collapse-toiminnolla, palata takaisin tarkastuksen valintaan tai ladata näkymä uudestaan, jolloin näkymäksi renderöidään QR-koodilukijan näkymä. Lisäksi kirjausnäkymästä voidaan avata aiempien tarkastusten historia Dialogi-komponentin painikkeesta. Kuvassa 24 esitetään kohteen kirjausnäkymä ja siinä myös näkyy mainittu Dialogi-komponentin painike.



Kuva 24. Kohteen kirjausnäkymän ulkoasu

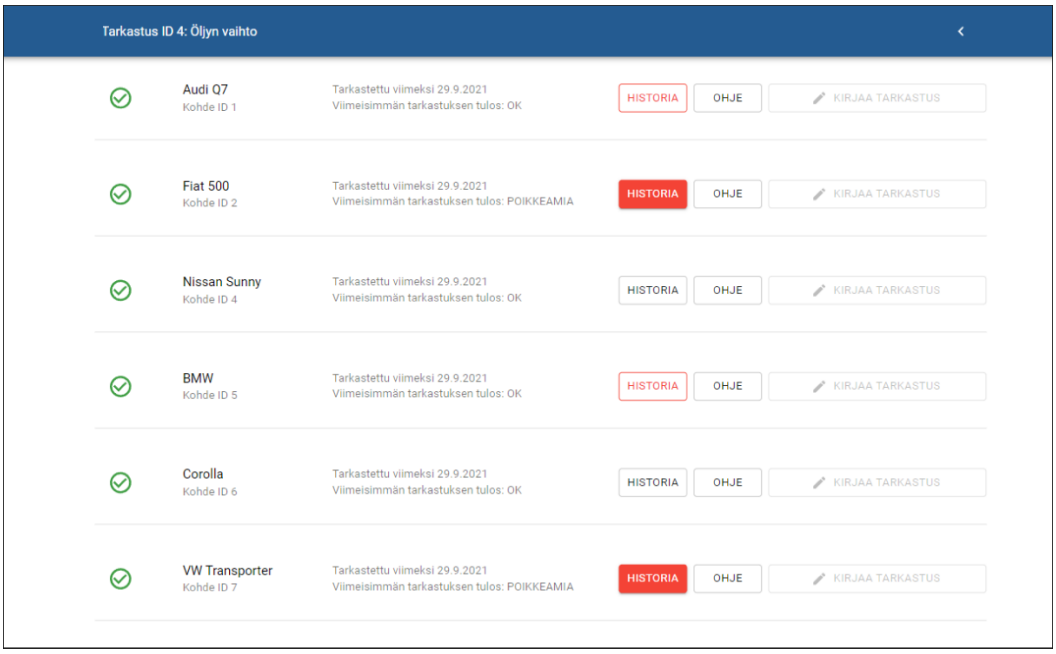
Kohteen kirjauksessa ilmoitetaan tarkastuksen tulos, joko ’OK’ tai ’Poikkeamia’. Tarkastuksesta on mahdollista kirjoittaa lisätietoja ja jos tarkastuksen tulos on ’Poikkeamia’, poikkeama-tekstikenttään on kirjoitettava vähintään 3 merkkiä. Kuvassa 25 esitetään tarkastuskokonaisuuden kirjaus.



Kuva 25. Tarkastuksen kirjaukseen johtava painike ja kirjausnäkymän ulkoasu

Kohteet-näkymässä näytetään painike tarkastuskokonaisuuden valmiiksi kirjaamiselle, kun kaikki tarkastuskokonaisuuden kohteet ovat merkitty tarkastetuksi tarkastuskokonaisuuden viimeisimmän tarkastuspäivämäärän jälkeen. Kuvassa 25 näkyy painike ja painikkeesta avautuva kirjausnäkymä. Tarkastuskokonaisuuden kirjauksessa voidaan kirjoittaa lisätietoja tekstikenttään.

Kuvaan 26 on havainnollistettu käyttöliittymän responsiivisuutta.



Kuva 26. Kohteet-näkymä tietokoneen näytön leveydellä

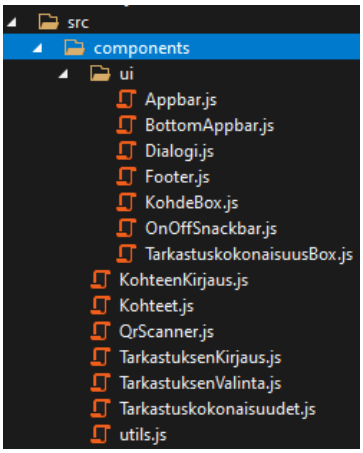
Kuvassa 26 näkyy Kohteet-näkymä tietokoneen näytöltä tarkasteltuna. Kirjausnäkymät avautuvat joka laitteella koko näytön leveydelle. Lisätietoja esittävä dialogi esitetään leveimmillä kuin 1005 pikselin näytöillä pienempinä ikkunoina näkymien päällä.

Käyttöliittymän offline-tilaan liittyvistä ilmoituksista kerrotaan luvussa Selainsovelluksen offline-tila.

3.2.2 React-komponentit

Päänäkymien komponentit hakevat käyttäjälle näytettävät tiedot käyttäen REST API -rajapintaa. Komponenttien näkymät rakentuvat alikomponenteista,

kuten Appbar ja KohdeBox. Kuvassa 27 esitetään kaikki käyttöliittymän komponentit selainsovelluksen hakemistossa. Alikansio ’ui’ sisältää komponentit pelkille visuaalisille komponenteille. Kooditiedosto nimeltä utils.js sisältää komponenttien välillä jaettuja aliohjelmia. Kirjausnäkymät ovat päänäkymien alikomponentteja ja lähettävät tietoja tietokantaan REST API -rajapinnan kautta. TarkastuksenValinta-komponentti on QrScanner-komponentin alikomponentti. TarkastuksenValinta-komponentti hakee tiedot kohteen tarkastuksen kirjausta varten, kun kirjausnäkymä avataan QR-koodilukijalla.



Kuva 27. Käyttöliittymälle rakennetut komponentit hakemistossa

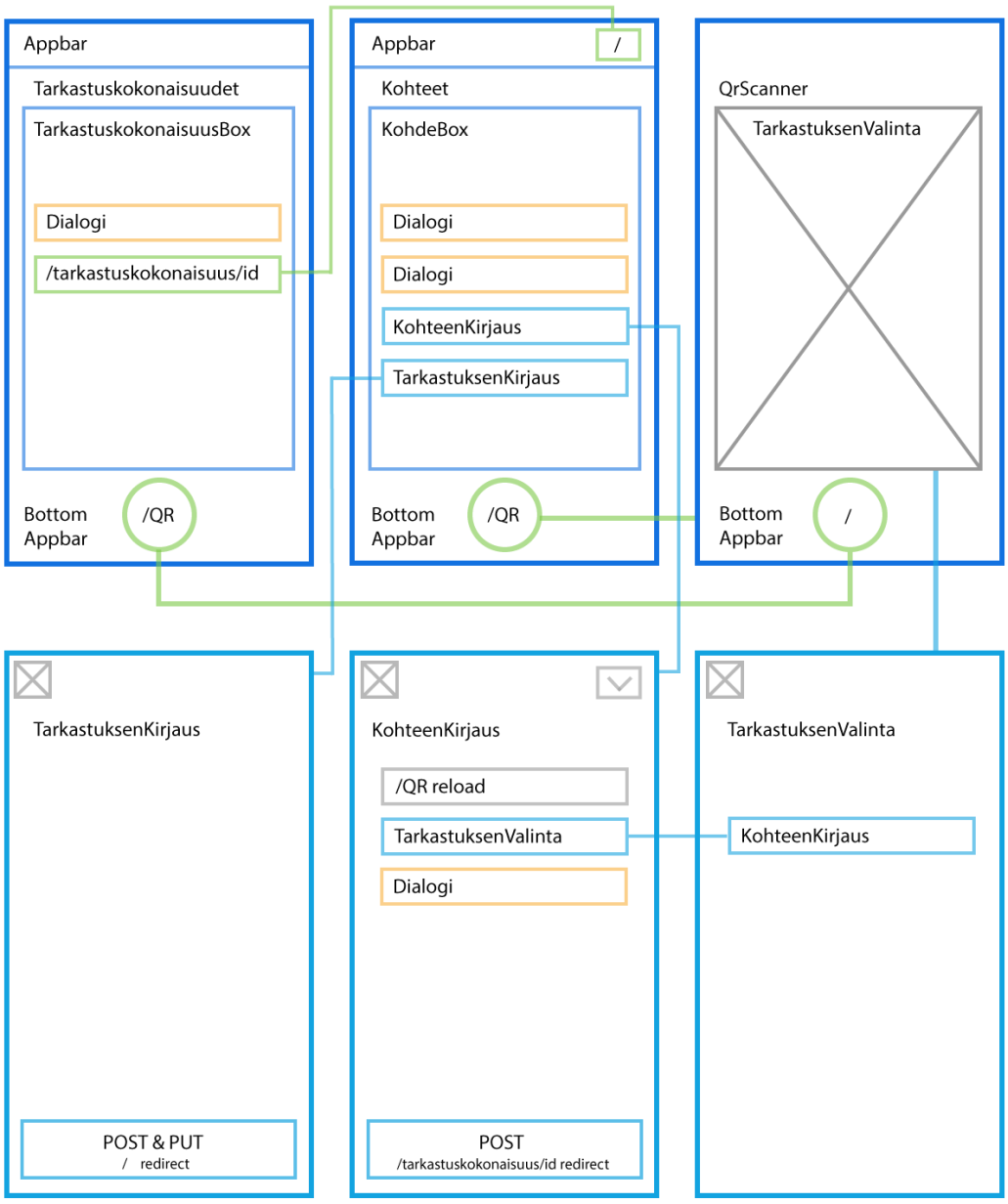
URL-polkujen reitit määritettiin App.js-koodiin (kuva 28). URL-polulta ’/’ ohjataan Tarkastuskokonaisuudet-komponentille, eli polulle ’/tarkastuskokonaisuudet’. Tarkastuksen kohteisiin johtaa URL-polku ’/tarkastuskokonaisuus/tarkastuksen id’ ja QR-koodilukijaan johtaa polku ’/qr’.

```
// BottomAppbar < 1005 px
const useStyles = makeStyles((theme) => ({
    sm: {
        [theme.breakpoints.up('md')]: {
            display:"none",
        }
    },
}));

breakpoints: {
    values: {
        xs: 0,
        sm: 600,
        md: 1005,
        lg: 1200,
        xl: 1536,
    },
},

function App() {
    const classes = useStyles();
    return (
        <ThemeProvider theme={theme}>
            <Router>
                <div className={classes.sm}><BottomAppbar /></div>
                <Route path="/" exact>
                    <Redirect to="/tarkastuskokonaisuudet" />
                </Route>
                <Route path="/tarkastuskokonaisuudet" component={Tarkastuskokonaisuudet} />
                <Route path="/tarkastuskokonaisuus/:tarkastusId" component={Kohteet} />
                <Route path="/qr" component={QrScanner} />
            </Router>
            <OnOffSnackbar />
        </ThemeProvider>
  );
}
export default App;
```

Kuva 28. Sovelluksen URL-reitit App-komponentissa

Kuvasta 28 käy myös ilmi BottomAppbar-komponentin näkyvyys laitteilla, joissa on alle 1005 pikseliä näytön leveyssuunnassa. OnOffSnackbar-komponentti näyttää käyttöliittymässä ilmoituksia liittyen offline-toimintaan.

Kuva 29 havainnollistaa komponenttien suhteita, reittejä ja visuaalisia sijainteja mobiilikäyttöliittymässä.

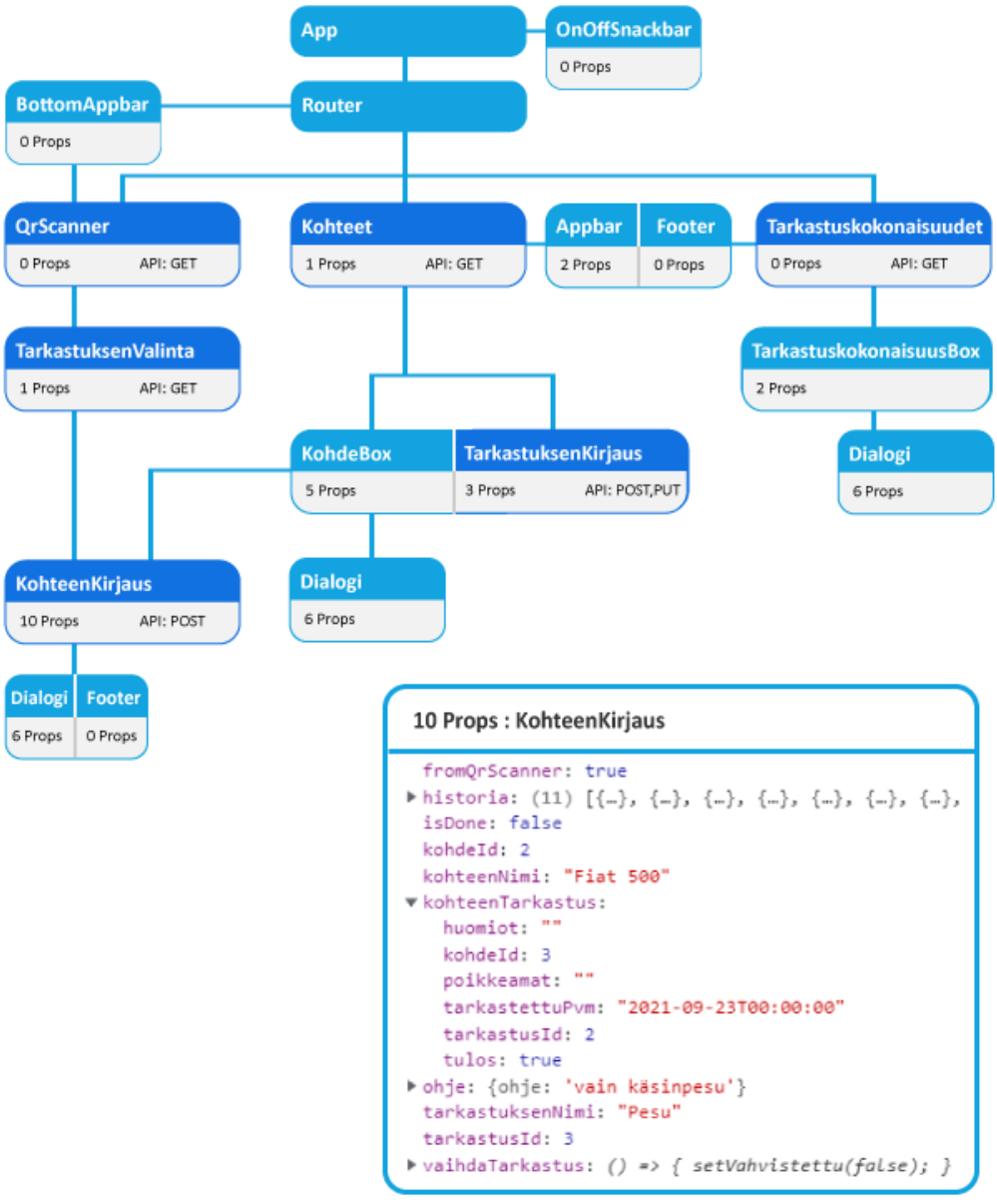


Kuva 29. Komponentit ja reitit komponenttien välillä

Kuvaan 29 on kirjoitettu komponenttien nimiä ja linkkien URL-polkuja. BottomAppbar-komponentilla on linkki QR-koodilukijalle. QrScanner-komponentti tunnistaa, mikäli sovellus on QR-koodilukijan URL-polulla, jolloin painikkeen linkki muutetaan URL-polulle ’/’. TarkastuksenValinta-näkymä avautuu QR-

koodilukijalla, kun QR-koodi on tunnistettu. Kirjausnäkymät avautuvat vanhempikomponentin näkymän päälle ja ovat suljettavissa joko ikonista ’X’ tai kirjaustoiminnolla eli onnistuneella HTTP-pyynnöllä. Mikäli QR-koodilla on edetty kohteen kirjausnäkymään, voi painikkeella palata takaisin tarkastuksen valitsemiseen. Alikomponentti nimeltä Dialogi on tarkoitettu lisätietojen esittämistä varten ja sitä käytetään kaikille eri lisätiedoille.

Tietoja komponentilta toiselle komponentille välitetään propseilla. Kuvassa 30 esitetään komponentit puudiagrammissa sekä esimerkki tiedon siirtymisestä propseina komponentilta toiselle komponentille. Kuvassa esimerkki propseista on TarkastuksenValinta-komponentilta KohteenKirjaus-komponentille.



Kuva 30. Komponenttipuu ja esimerkki propseista

Kuvaan 30 on merkitty komponentteja esittäviin solmuihin komponentille välittyvien props-tietojen lukumäärät – esimerkiksi ’8 Props’ – sekä mahdolliset HTTP-metodit, mikäli komponentti on yhteydessä REST API -rajapintaan. Puun juurisolmuna on ’App’, joka merkitsee DOMiin renderöitävää App-komponenttia. Router-solmu on ohjaa URL-reittejä sen alasolmujen välillä, kuten kuvasta nro 28 (s. 31) ilmenee.

3.2.3 Sovelluksen toiminnot

Sovelluksen päätoimintoja esitellään kuvien avulla. Kuvissa esiintyy komponenttien koodien osia ja toisiinsa liittyviä koodeja eri komponenteissa voidaan esittää samassa kuvassa. Koska komponentteja on useita, kaikkia sovelluksen toimintoja ja koodeja ei esitellä. Päätoimintoja ovat tietojen hakeminen REST API -rajapinnalta, tavat kirjausnäkymien avaamiseksi, QR-koodilukijan käyttö ja kirjauksen tekeminen. Kuvassa 31 esitetään esimerkki tietojen hausta REST API -rajapinnalta. Näkymän päivittyessä useEffect-Hook kutsuu toimintoja tietojen hakemiseksi ja haetut tiedot tallennetaan niille varattuihin React-Hookeihin. Mahdollisesta virheestä ilmoitetaan käyttöliittymässä. Kun kaikki tarvittavat tiedot on haettu, ne näytetään käyttöliittymässä.

```
const [kohteenTarkastus, setKohteenTarkastus] = useState({ list: [], err: null, loading: true });
/* Tarkastuksen kohteiden tarkastushistoria hookiin kohteenTarkastus,
* tiedot per tarkastuksen kohde hookiin Kohde (jättää viimeisimmät tiedot historiasta), viimeisenä haetaan kohteenNimi */
const fetchKohteenTarkastus = async () => {
    try {
        const response = await fetch(`api/TblKohteenTarkastus`)
        const data = await response.json();
        let listArray = [];
        data.map((data) => (
            (data.tarkastusId === tarkastusId)
                ? listArray.push({
                    "kohdeId": data.kohdeId,
                    "tarkastusId": data.tarkastusId,
                    "tarkastettuPvm": data.tarkastettuPvm,
                    "poikkeamat": data.poikkeamat,
                    "tulos": data.tulos,
                    "huomiot": data.huomiot
                })
                : null
        ))
        setKohteenTarkastus({ list: listArray, loading: false }); // kaikki tarkastukset (historia)
         // järjestä tiedot aikajärjestykseen, uusimmat viimeiseksi ja poistaa tuplaoliot arraysta alkaen vanhimmista
        let kohdeArray = removeDuplicates(sortByDate(listArray, "tarkastettuPvm"), "kohdeId");
        // järjestää kohteet tarkastuspäivämäärän mukaan aikajärjestykseen, jos on eri päivinä kirjattu
        setKohde({ list: sortByDate(kohdeArray, "tarkastettuPvm"), loading: false });
        fetchKohteenNimi(kohdeArray); // kohteille nimet TblKohde
    } catch (err) { setKohteenTarkastus({ list: [], err: err, loading: false }); }
}
useEffect(() => { // Suoritetaan ennen kuin lisätään DOMiin
    fetchTarkastus();
    fetchKohteenTarkastus();
    fetchOhje();
    // Jos service worker on rekisteröity, hakee mahdolliset tiedot IndexedDB
    navigator.serviceWorker.getRegistrations().then(registrations => {
        if (registrations.length > 0) { getDataFromIdb() }
    });
}, [])
```

Kuva 31. Komponentin toiminto tietojen hakemiseksi REST API -rajapinnalta

Kuvassa 31 esiintyvä koodi on Kohteet-komponentilla, se hakee ja järjestelee tietoja sekä tallentaa tietoja React-Hookeihin. Tällä tavalla tietyt komponentit tallentavat tietoja käyttöönsä tai välittävät tietoja eteenpäin alikomponenteille. Käyttöliittymään muodostuvan Kohteet-näkymän koodi esitetään kuvassa 32.

Kohteet.js

```
{(!tarkastus.loading)
    ? <Appbar titleLink={true} title={`Tarkastus ID: ${tarkastusId}: ${tarkastus.list[0].nimi}`}/>
    : null
}
<Container className={classes.con}>
    {(idbKohteenTarkastus > 0)
    ? <Box className={classes.greenBox}>
        <Typography>Kohteiden tarkastuksia kirjattu offline-tilassa yhteensä {idbKohteenTarkastus} kpl.</Typography>
    </Box>
    : null}
    {(kohteenTarkastus.err || kohde.err || tarkastus.err || kohteenNimi.err || ohje.err)
    ? <Typography>
        Virhe tietojen haussa.{kohteenTarkastus.err} {kohde.err} {tarkastus.err} {kohteenNimi.err} {ohje.err}
    </Typography>
    : (kohteenTarkastus.loading || kohde.loading || tarkastus.loading || kohteenNimi.loading || ohje.loading)
        ? <Box className={classes.box}><Typography>Tietoja haetaan...</Typography></Box>
        : (kohde.list[0])
            ? <>
                {(isAllDone())
                ? <TarkastuksenKirjaus
                            tarkastusId={tarkastusId} aikavali={tarkastus.list[0].aikavali} nimi={tarkastus.list[0].nimi} />
                : null}
                <KohdeBox tarkastus={tarkastus.list[0]} kohde={kohde.list} kohteenNimi={kohteenNimi.list}
                        kohteenTarkastus={kohteenTarkastus.list} ohje={ohje.list} />
            </>
            : <Box className={classes.box}><Typography>Ei kohteita</Typography></Box>}
    <Footer />
</Container>
```

KohdeBox.js

```
{kohde.map((data) => (
<Box key={data.kohdeId} className={classes.box}>
    <Grid container direction="row" justifyContent="flex-end" alignItems="center" spacing={1}>
        <Grid item xs={3} sm={1} md={1}>
            {(isDone(tarkastus.aikavali, data.tarkastettuPvm))
                ? <ListItem><ListItemIcon><CheckIcon color="secondary" className={classes.icon} /></ListItemIcon></ListItem>
                : <ListItem><ListItemIcon><ExIcon color="error" className={classes.icon} /></ListItemIcon></ListItem>
            }
        </Grid>
        <Grid item xs={9} sm={5} md={2}>
            <ListItem>
                <ListItemText primary={haeKohteenNimi(data.kohdeId)} secondary={`Kohde ID ${data.kohdeId}`} />
            </ListItem>
        </Grid>
        <Grid item xs={12} sm={6} md={4}>
                <Typography variant="body2" color="textSecondary" className={classes.ml2}>
                    Tarkastettu viimeksi {parseDate(data.tarkastettuPvm)}
                </Typography>
                <Typography variant="body2" color="textSecondary" className={classes.ml2}>
                    Viimeisimmän tarkastuksen tulos: {(data.tulos) ? "OK" : "POIKKEAMIA"}
                </Typography>
        </Grid>
        <Grid item xs={6} sm={3} md={1}>
            {(data.poikkeamat)
            ? <Dialogi redButton={true} buttonTitle="Historia"
                    title={`Aiemmat tarkastukset - ${tarkastus.nimi}: ${haeKohteenNimi(data.kohdeId)}`}
                    data={haeHistoriaDialogille(data.kohdeId)} dialogiKey={`historia+${data.kohdeId}`} />
            : (tarkastaHuomiot(data.kohdeId))
                ? <Dialogi redOutlinedButton={true} buttonTitle="Historia"
                        title={`Aiemmat tarkastukset - ${tarkastus.nimi}: ${haeKohteenNimi(data.kohdeId)}`}
                        data={haeHistoriaDialogille(data.kohdeId)} dialogiKey={`historia+${data.kohdeId}`} />
                : <Dialogi buttonTitle="Historia"
                        title={`Aiemmat tarkastukset - ${tarkastus.nimi}: ${haeKohteenNimi(data.kohdeId)}`}
                        data={haeHistoriaDialogille(data.kohdeId)} dialogiKey={`historia+${data.kohdeId}`} />
            }
        </Grid>
        <Grid item xs={6} sm={3} md={1}>
            <Dialogi buttonTitle="Ohje" title={`Tarkastusohje ${tarkastus.nimi}: ${haeKohteenNimi(data.kohdeId)}`}
                    data={haeOhjeDialogille(data.kohdeId)} dialogiKey={`ohje+${data.kohdeId}`} />
        </Grid>
        <Grid item xs={12} sm={6} md={3}>
        {(isDone(tarkastus.aikavali, data.tarkastettuPvm))
        ? <Button className={classes.button} variant="outlined" startIcon={<NoteIcon />} disabled>Kirjaa tarkastus</Button>
        : <KohteenKirjaus tarkastusId={data.tarkastusId} kohdeId={data.kohdeId} tarkastuksenNimi={tarkastus.nimi}
                        kohteenNimi={haeKohteenNimi(data.kohdeId)} kohteenTarkastus={data}
                        ohje={haeOhje(data.kohdeId)} historia={haeHistoriaDialogille(data.kohdeId)}/>
        }
        </Grid>
    </Grid>
    <Divider className={classes.mt2} />
</Box>
))}
```

Kuva 32. Komponenttien käyttöliittymään renderöitäviä koodeja

Kuvassa 32 esiintyvässä koodissa Kohteet-komponentti tarkastaa onko kaikki tarkastuskokonaisuuden kohteet tarkastettu ja KohdeBox-komponentti tarkastaa onko tietty kohde tarkastettu. Nämä toiminnot määrittävät onko kirjausnäkymiä mahdollista avata käyttöliittymästä. Muiden näkymien koodit ovat rakennettu samaan tapaan, joten layout-koodeja ei kokonaisuutena esitellä tämän tarkemmin.

Osa komponenteista käyttää samoja aliohjelmia, jotka jaetaan komponenttien välillä utils.js-tiedostolla. Esimerkiksi kuvissa 31 ja 32 esiintyvät sortByDate ja isDone -toiminnot ovat muidenkin komponenttien käytössä. Kuvassa 33 esitetään esimerkin vuoksi isDone-toiminto.

```
// Päivien määrä negatiivisena ennen seuraavan tarkastuksen kirjausmahdollisuutta, kaikilla tarkastuksilla sama määrä.
export const daysBefore = Number(-7)
/* isDone(days, d):
* Lisää annettuun päivämäärään päivien lukumäärän, vertaa uutta päivämäärää kuluvaan päivään (today).
* Jos näiden päivämäärien välillä on enintään yllä määritetyn (daysBefore) verran vuorokausia, palauttaa false.
*
* Eli laskee onko seuraavaan tarkastuspäivään enintään 7 päivää aikaa, jos on, kohde voidaan kirjata tarkastetuksi.
* * days = tarkastuskokonaisuuden aikaväli
* * d = kohteen viimeksi tarkastettu -pvm
*/
export const isDone = (days, d) => {
    let today = Date.now(); // kuluva päivä
    let date = new Date(d); // kohteen viimeisin tarkastuspäivämäärä

     // lisää tarkastuksen aikavälin (päivien lukumäärä) kohteen tarkastuspäivämäärään
    let nextDate = date.setDate(date.getDate() + days);
    nextDate = new Date(nextDate)

    // verrataan seuraavaa päivämäärää kuluvaan päivään, jos today ja nexDate ovat samoja, arvo on 0
    let daysBetweenDates = (today - nextDate.getTime()) / (1000 * 3600 * 24)

    /* jos kuluvan päivän ja tarkastuskokonaisuuden seuraavan tarkastuspäivämäärän
       välillä on vähintään 7 päivää (-7) palauttaa true */
    if (daysBetweenDates >= daysBefore) { return false } else { return true }
}
```

Kuva 33. Esimerkki useammalle komponentille välitettävästä toiminnosta

Kuvassa 33 näkyy luvun seitsemän määritys daysBefore-vakiomuuttujassa, joka jaetaan päivämääriä laskeville komponenteille. Sovellus laskee jokaisen tarkastuskokonaisuuden seuraavan tarkastuspäivämäärän Tarkastuskokonaisuudet-komponentissa. Sovellus merkitsee tarkastuksen tekemättömäksi, kun seuraavaan tarkastuspäivämäärään on aikaa seitsemän vuorokautta. Kaikki tarkastukset käyttävät tätä samaa määrää seitsemän. Työhön ei kehitetty tarkempia asetuksia määrittämään milloin tarkastus merkitään tekemättömäksi. Sovellus ilmoittaa ikoneilla, jotka esitettiin luvussa Käyttöliittymän ulkoasu, mikäli tarkastus pitäisi suorittaa tai mikäli tarkastuksen teko on myöhässä.

QR-koodilukijana toimii avoimen lähdekoodin toteutus, joka avaa verkkokameran ja tunnistaa QR-koodin sisällön. QR-koodilukija tuodaan QrScanner-komponentille nimellä Scanner. QrScanner-komponentti tunnistaa käytössä olevan laitteen selaimen mitat ja asettaa kameranäkymän vastaamaan näitä mittoja. Kuvassa 34 esitetään QrScanner-komponentin koodi, jolla hallitaan QR-koodilukijan toimintaa. QR-koodin sisältönä käytetään tekstinä kohteen id:tä. Kun QrScanner on tallentanut id-arvon React-Hookiin, avautuu näkymä tarkastuskokonaisuuden valitsemiselle (TarkastuksenValinta-komponentti).

```
// https://www.npmjs.com/package/react-webcam-qr-scanner
import React from "react";
import { useState } from 'react';
import Scanner from "react-webcam-qr-scanner";
import { Snackbar, } from '@material-ui/core';
/* Child components */
import TarkastuksenValinta from './TarkastuksenValinta';

/* QrScanner avataan linkillä (BottomAppbar)
* Kysyy saako kameraa käyttää ja avaa kameranäkymän (video), määritetty valmiissa komponentissa Scanner
* QR-koodin tekstisisältönä olisi numero (kohteen id)
* Kun QR-koodin numero (kohdeId) on tunnistettu,
  hakee tiedot taulusta KohteenTarkastus ja tarkastaa löytyykö numeroa vastaava kohdeId tiedoista
* Jos löytyy, tallentaa Hooksiin ja avaa kirjausnäkymän */

const QrScanner = () => {
    // selaimen mitat
    const [width, setWidth] = useState(window.innerWidth);
    const [height, setHeight] = useState(window.innerHeight - window.innerHeight * 0.07); // -7 % BottomAppbar height

    const [err, setErr] = useState(""); // snackbar msg
    const [snackBarOpen, setSnackBarOpen] = useState(false)
    const handleClose = () => {
        setSnackBarOpen(false);
    };
    const [kohdeId, setKohdeId] = useState(); // QR-koodilla

    const handleDecode = async (result) => { // Kun QR-koodi on tunnistettu / luettu
        console.log("qr result: " + result.data);
        // Tarkastaa löytyykö QR-koodilla oleva numero KohteenTarkastus-taulun kohdeId:stä
        // jos löytyy, asettaa kohdeId Hooksiin numeron
        // jos ei löydy, ilmoittaa snackbarilla, ettei kohteen id:llä löydy tietoja
        try {
            const response = await fetch('api/TblKohteenTarkastus');
            const data = await response.json();
            if (data.some(e => e.kohdeId === Number(result.data))) {
                setKohdeId(Number(result.data))
            } else { setErr(`Kohteen id:tä ${result.data} vastaavaa kohdetta ei löydetty!`); setSnackBarOpen(true)}
        } catch (err) {
            console.log(err);
            setErr("Tietojen haussa tapahtui virhe!"); setSnackBarOpen(true)
        }
    }
    // kun QR-koodi on luettu ja kohdeId on tunnistettu, avataan TarkastuksenValinta-näkymä
    return (
        <>
            <Snackbar
                open={snackBarOpen}
                onClose={handleClose}
                anchorOrigin={{ 'vertical': 'top', 'horizontal': 'center' }}
                autoHideDuration={3000}
                message={err}
            />
            {(kohdeId)
                ? <TarkastuksenValinta kohdeId={kohdeId} />
                : <Scanner
                    onDecode={handleDecode}
                    constraints={{
                        audio: false,
                        video: { facingMode: "environment", width: height, height: width }
                    }}
                    captureSize={{ width: width, height: height }}
                />
            }
        </>
    );
}
export default QrScanner;
```

Kuva 34. QR-koodilukijan hallinnan komponentin koodi

Kuvassa 34 esiintyvä koodi myös havainnollistaa kokonaisen komponentin rakennetta. Komponentille tuodaan (import) sen käyttämät komponentit tai tiedot ja se viedään (export) funktionaalisena komponenttina muiden komponenttien käytettäväksi.

QrScanner-komponentin alikomponenttina on TarkastuksenValinta-komponentti, jolle välitetään propsina tunnistettu kohteen id (kuva 34). Alikomponentti hakee REST API -rajapinnan kautta tiedot QR-koodilukijan antamasta kohteesta ja kaikista sen tarkastuksista. Kun käyttäjä on valinnut kohteelle tarkastuskokonaisuuden, näkymä päivittyy kohdetta ja tarkastuskokonaisuutta vastaavaksi kirjausnäkymäksi (KohteenKirjaus-komponentti). Kaikki tarvittava tieto välitetään propseina kirjausnäkymän komponentille. Kirjausnäkymästä voidaan palata takaisin tarkastuksen valintaan takaisinkutsutoiminnolla (callback). Kun alikomponentille annettu toiminto käynnistyy (trigger), se kutsuu vanhempikomponentin takaisinkutsutoimintoa, joka käsittelee alikomponentilta saamansa tiedon. Käytännössä käyttäjä klikkaa painiketta, kun haluaa vaihtaa tarkastuskokonaisuutta esimerkiksi auton vuosihuollosta saman auton määräaikaiskatsastukseen. Kuvassa 35 näkyy propsina välitetyn funktion toimintaperiaate, QR-koodilta avautuvan komponentin välittämät tiedot kirjauskomponentille sekä kirjauskomponentin tapa tunnistaa avataanko näkymä QR-koodin kautta vai käyttöliittymän painikkeella.

TarkastuksenValinta.js

```
// KohteenKirjaukselta tuleva pyyntö tarkastuksen vaihtamiselle
const handleCallback = () => {
    setVahvistettu(false)
}

return (
    <>
        {(vahvistettu && tarkastusId)
            ? <KohteenKirjaus
                vaihdaTarkastus={handleCallback}
                isDone={params()}
                tarkastusId={tarkastusId} tarkastuksenNimi={haeTarkastuksenNimi()} kohdeId={kohdeId}
                kohteenNimi={kohteenNimi.list[0].nimi} kohteenTarkastus={haeKohteenTarkastus()}
                historia={haeHistoriaDialogille()} ohje={haeOhje()} fromQrScanner={true} />
            : <Dialog fullScreen open={open}>
```

KohteenKirjaus.js

```
useEffect(() => {
    // kirjausikkuna avataan, kun vanhempi-komponentti on TarkastuksenValinta
    if (fromQrScanner) {
        setOpen(true);
    }
}, [])

// Menun linkkiä 'klikattu', 'e' on string 'vaihda' tai 'qr'
const handleMenuClick = (e) => {
    setAnchorEl(null);
    // Tarkastuksen vaihto, palaa parent-komponentin näkymään
    if (e === 'vaihda') {
        // callback parent-komponentille TarkastuksenValinta
        props.vaihdaTarkastus();
    }
```

Kuva 35. Propsit ja takaisinkutsutoiminto komponenttien välillä

KohteenKirjaus-alikomponentille välitettävät tiedot haetaan React-Hookeista, kun käyttäjä on valinnut kohteelle tarkastuksen. Kuvassa 35 React-Hookeista hakeminen näkyy toimintojen niminä. React-Hookeihin on tallennettu REST API -rajapinnalta haetut tiedot.

Kohteen tarkastuksen tulos kirjataan kirjausnäkymässä, mikäli tarkastuksen tekeminen on päivämäärien puolesta mahdollista. QR-koodin lukeminen mahdollistaa tietyn kohteen tietojen tarkastelun, vaikka tarkastusta ei voisi kirjata. Kuvassa 36 esitetään tietojen lähetys POST-metodilla, kun kohde kirjataan tarkastetuksi.

```
//Jos tarkastuksen kirjaus on tallentunut IndexedDB, SW lähettää viestin ja kirjausikkuna suljetaan
const isOffline = () => {
    navigator.serviceWorker.addEventListener('message', event => {
        if (!navigator.onLine) {
            if (!fromQrScanner) { history.go(`/tarkastuskokonaisuus/${tarkastusId}`) } // Renderöi sivun/haku indexedDB
            // Jos avattu QR-koodilta, handleClose ohjaa kuten yllä, joten tässä ei ohjata enää toista kertaa
            handleClose();
        }
    });
}

// POST
const handleSubmit = () => {
    let tulos; (radioValue === "ok") ? tulos = true : tulos = false;
    const newData = {
        "tarkastusId": tarkastusId, "kohdeId": kohdeId, "tarkastettuPvm": today,
        "tulos": tulos, "poikkeamat": poikkeamatValue, "huomiot": lisatiedotValue
    }
    fetch(`api/TblKohteenTarkastus`, {
        method: 'POST',
        body: JSON.stringify(newData),
        headers: {
            'Content-Type': 'application/json'
        }
    }).then((response) => {
        console.log("response: ", (response.status))
        if (!fromQrScanner) { history.go(`/tarkastuskokonaisuus/${tarkastusId}`) } // Renderöi sivun
        // Jos avattu QR-koodilta, handleClose ohjaa kuten yllä, joten tässä ei ohjata enää toista kertaa
        handleClose();
    }).catch(err => // Offline-tilassa error
        console.log(err),
        isOffline() // Jos kirjaus paikallisesti onnistuu (vaatii offline-tilan), kirjausikkuna suljetaan
    );
}
```

Kuva 36. POST-pyynnön lähetys, kun kohde kirjataan tarkastetuksi

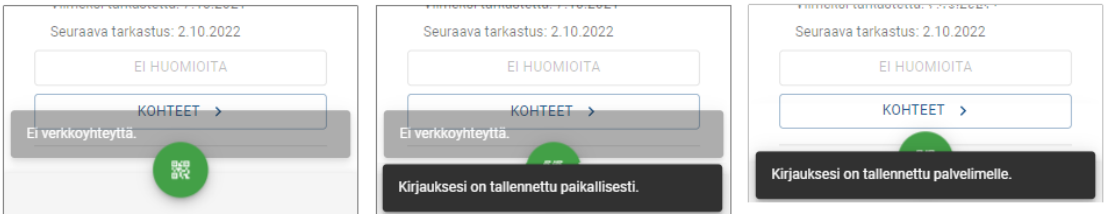
Tarkastuskokonaisuuden kirjaustapahtuma suoritetaan samalla tavalla kuin kuvassa 36 esiintyvä kohteen kirjaustapahtuma. Tarkastuskokonaisuuden kirjaus tallennetaan TblValmisTarkastus-tietokantatauluun, jonka lisäksi tehdään PUT-metodilla päivitys Tarkastus-tietokantatauluun viimeisimmän tarkastuspäivämäärän osalta.

Kun tietojen tallennus on onnistunut, sovellus sulkee kirjausnäkymän ja renderöi näkymän uudestaan, jotta uudet tiedot päivittyvät näkymään. Tämä tapa

toimii QR-koodin kautta tehdyille kirjauksille käyttäjäystävällisesti, sillä kameranäkymästä siirrytään tarkastusta ja kohdetta vastaavaan näkymään. Käyttöliittymästä - ilman QR-koodia - tehdyn kirjauksen näyttäminen näkymän renderöinnillä ei ole käyttäjäystävällisin tapa ja sitä voisi jatkokehittää. Kuvassa 36 näkyy myös virheenkäsittely, kun virhe johtuu verkkoyhteydettömästä tilasta.

3.3 Selainsovelluksen offline-tila

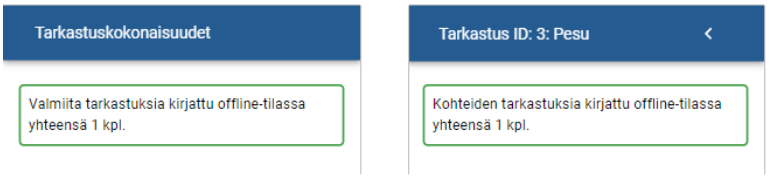
Käyttöliittymässä näytetään ilmoitus verkkoyhteydettömästä tilasta ja kirjauksen onnistuneesta tallennuksesta selaimen paikalliseen tietokantaan. Lisäksi ilmoitus näytetään, kun verkkoyhteys on palautunut ja paikallisesti tallennetut tiedot ovat siirretty onnistuneesti palvelimelle. Käyttöliittymässä näytettävät ilmoitukset esitetään kuvassa 37.



Kuva 37. Offline-tilaan liittyvät ilmoitukset käyttöliittymässä

OnOffSnackbar-komponentti avaa kuvan 37 ilmoitukset päänäkymien päälle. Onnistuneista tallennuksista kertova ilmoitus poistuu näkymästä automaattisesti muutaman sekunnin kuluttua.

Kohteet ja Tarkastuskokonaisuudet -komponentit laskevat paikallisesti tallennetuiden tietojen määrät, kun tiedot kohdistuvat valitun tarkastuksen kohteisiin tai ainoastaan valmiiksi kirjattuihin tarkastuskokonaisuuksiin. Määristä näytetään ilmoitus komponentin näkymässä (kuva 38).



Kuva 38. Paikallisesti tallennetuista tiedoista ilmoittaminen käyttöliittymässä

Tietojen hakeminen kuvan 38 ilmoituksia varten tapahtuu selaimen paikallisesta tietokantarajapinnasta samalla kun komponentti pyytää REST API -rajapinnalta tietoja.

3.3.1 Service Worker

Service Worker -koodi kehitettiin kehitysympäristöön, kun Service Worker saatiin toimivaksi, sitä muokattiin tuotantoon sopivaksi. Tuloksena syntyi kaksi Service Worker -koodia, kehitys- ja tuotantoympäristöille omansa, joiden ero on käyttöliittymäkoodien (app shell) tallentamisessa. Alkuperäisestä suunnitelmasta poiketen Googlen Workbox-kirjastoa ei käytetty perustuen Service Workerin toiminnan parempaan ymmärtämiseen ja oppimiseen opinnäytetyön prosessissa. Workbox voidaan tarvittaessa ottaa myöhemmin käyttöön.

Tuotantoon muodostettavat käyttöliittymäkoodien tiedostonimet sisältävät tunnisteen (hash). Tuotannon Service Worker hakee tiedot manifest-tiedostosta, joka muodostuu selainsovelluksen tuotantoon rakentamisen (build) vaiheessa ja sisältää kaikkien käyttöliittymään sisältyvien tiedostojen nimet. Kehitysvaiheessa tunnisteita ei ole, joten käyttöliittymään liittyvät tiedostonimet tiedetään ennalta ja voidaan nimien perusteella määrittää tallennettavaksi välimuistiin Service Workerin asennuksen yhteydessä.

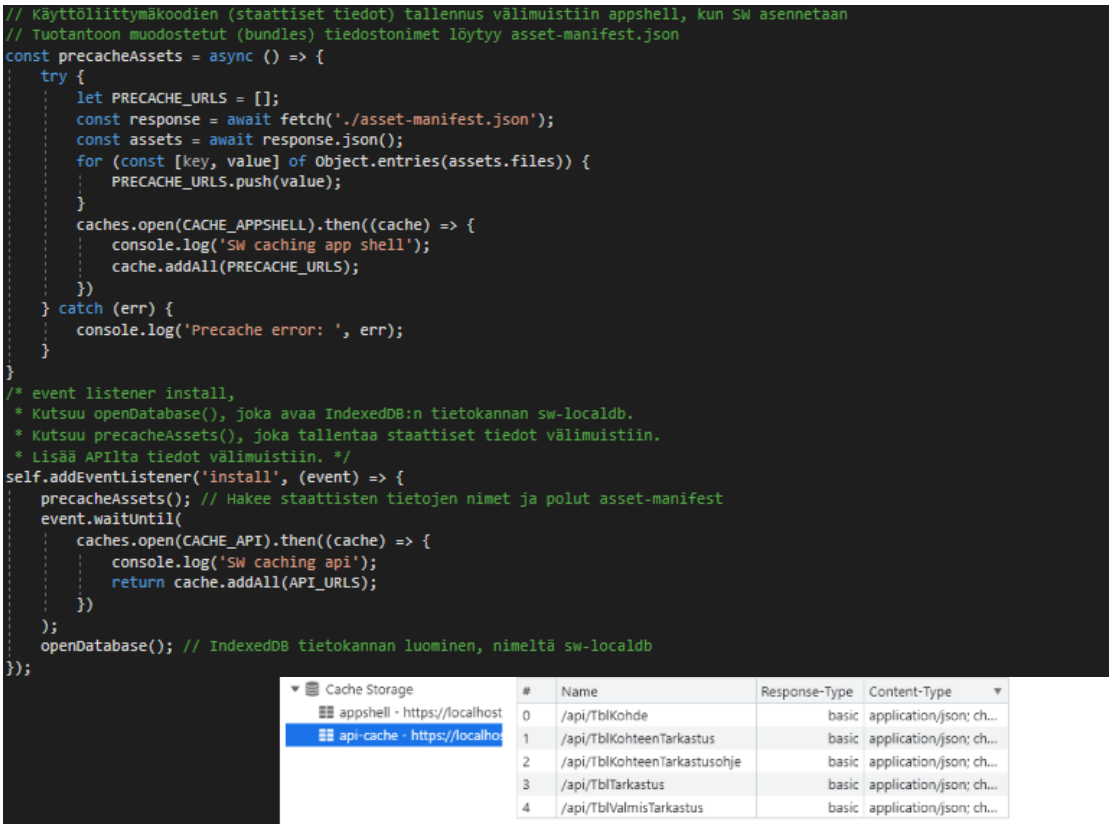
Service Worker rekisteröidään selainsovelluksen juuressa sijaitsevassa swRegistration.js-tiedostossa (kuva 39).

```
export default function swRegistration() {
    // Service worker rekisteröinti, kun Production
    // React-sovelluksen tuotannossa muodostuu asset-manifest.json, joka sisältää UI:n tiedostojen polut/nimet
    // Tuotannon SW hakee tiedostojen nimet polkuineen ja tallentaa välimuistiin
    if (process.env.NODE_ENV === 'production' && 'serviceWorker' in navigator) {  // Jos selain tukee service worker
        window.addEventListener('load', async () => {
            try {
                const reg = await navigator.serviceWorker.register('./sw.js');
                console.log('Service worker registered in production', reg);
            } catch (err) {
                console.log('Service worker registration failed ', err);
            }
        });
    }
    // Service worker rekisteröinti, kun Development, lähinnä opinnäytetyötä varten. UI:n tiedostot tiedetään ennalta
    if (process.env.NODE_ENV === 'development' && 'serviceWorker' in navigator) {  // Jos selain tukee service worker
        window.addEventListener('load', async () => {
            try {
                const reg = await navigator.serviceWorker.register('./dev-sw.js');
                console.log('Service worker registered in development', reg);
            } catch (err) {
                console.log('Service worker registration failed ', err);
            }
        });
    }
}
```

Kuva 39. Service Worker -koodin rekisteröinti

Service Workerin rekisteröivä koodi kuvassa 39 tunnistaa, käynnistetäänkö sovellus kehitys- vai tuotantoympäristössä, ja valitsee sen mukaan rekisteröitävän Service Worker -koodin. Kuvassa 39 esiintyvä koodi tuodaan index.js-tiedostolle, joka suorittaa rekisteröinnin selainsovelluksen käynnistyessä. Rekisteröinnissä kerrotaan selaimelle missä sovelluksen Service Worker -koodi-tiedosto sijaitsee.

Service Workerin rekisteröinnin jälkeen selain aloittaa sen asentamisen. Asennuksen aikana lisätään määritetyt tietosisällöt selaimen välimuistiin (kuva 40).



Kuva 40. Service Worker -koodin asennus

Kuvassa 40 tuotannon Service Worker hakee käyttöliittymätiedostojen polut sekä nimet manifest-tiedostosta ja tiedostot tallennetaan niille avattuun välimuistiin. Kuvassa 40 näkyy tämä selaimeen avattu välimuisti nimellä ’appshell’, jonka sisällöstä muodostuu käyttöliittymä ja sen toiminnallisuus. REST API:lta haettavat tiedot tallennetaan etukäteen välimuistiin nimeltä ’api-cache’. Kuvassa 40 näkyy näiden tietojen nimet, eli url-polut, selaimelle avatussa välimuistissa.

Asennuksen jälkeen Service Worker aktivoidaan ja sen yhteydessä käynnistetään verkkoyhteyden tarkastaminen. Aktivoinnin jälkeen Service Worker kuuntelee selaimen HTTP-tapahtumat (kuva 41).

```
/* Kutsuu checkConnection(),
   joka jatkaa verkkoyhteyden tarkastamista 20 sec välein ja kutsuu sendOfflinePosts() */
self.addEventListener('activate', event => {
    clients.claim();
    checkConnection(); // jos verkossa: sendOfflinePosts()
});
// event listener fetch
self.addEventListener('fetch', (event) => {
    /* Network First */
    /* API GET-pyyntöjen vastaus haetaan verkosta, tallennetaan vastaus välimuistiin api-cache
     * jos verkkoyhteyttä ei ole,
       palautetaan api GET-pyyntöjen vastaus välimuistista api-cache jos sieltä vastaus löytyy */
    if (event.request.method === 'GET' && event.request.url.includes('api/')) { // jos GET ja url api
        event.respondWith(
            fetch(event.request).then((response) => { // haetaan verkosta
                return caches.open(CACHE_API).then((cache) => {
                    if (response.status === 200) {
                        cache.put(event.request, response.clone()); // tallennetaan verkon vastaus välimuistiin
                    }
                    return response; // palautetaan verkon vastaus
                })
            })
            .catch(() => { // jos verkkopyyntö epäonnistuu
                return caches.match(event.request); // palautetaan vastaus välimuistista
            })
        )
    }
    /* Cache First */
    // Palauttaa tiedostot välimuistista appshell tai hakee verkosta tarvittaessa
    else if (event.request.method === 'GET' && !event.request.url.includes('api/')) { // jos GET ja url ei api
        if (event.request.mode === 'navigate') { // url routet (esim '/tarkastuskokonaisuudet')
            event.respondWith(caches.match('/index.html'));
        } else {
            event.respondWith(caches.match(event.request).then((response) => { // haetaan välimuistista vastaus
                if (response) {
                    return response; // jos löytyy, palautetaan
                }
                else {
                    return fetch(event.request).then((response) => { // jos ei löydy, haetaan verkosta
                        return caches.open(CACHE_APPSHELL).then((cache) => {
                            cache.put(event.request.url, response.clone()); // tallennetaan verkon vastaus cacheen
                            return response; // palautetaan verkon vastaus
                        })
                    }).catch(err => console.log(err))
                }
            }))
        }
    }
    // Jos fetch-metodi POST tai PUT ja sovellus ei ole yhteydessä verkkoon, pyyntö sisältöineen funktiolle saveIntoIDB()
    if (!navigator.onLine && event.request.method === 'POST'
        || !navigator.onLine && event.request.method === 'PUT') {
        let url = event.request.url; // /api/TblKohteentarkastus tai / api / TblValmisTarkastus
        let method = event.request.method; // /api/TblValmisTarkastus lähetetään sekä POST että PUT -pyynnöt
        // event.request.text = body, JSON-data käyttäjän lähettämältä lomakkeelta (TarkastuksenKirjaus tai KohteenKirjaus)
        event.request.text().then((body) => {
            saveIntoIDB(url, method, body);
        })
    }
})
```

Kuva 41. Service Workerin aktivointi ja tapahtumankäsittelijä

Kuvassa 41 esiintyvä koodi hakee tietoja välimuistista ja verkosta sekä tallentaa tietoja välimuistiin. REST API -rajapinnalta pyydettävät tiedot haetaan verkkoyhteyden avulla ja verkosta saatu vastaus tallennetaan selaimen API-vastauksille avattuun välimuistiin. Kun verkkoyhteyttä ei ole, tiedot palautetaan välimuistista. Muiden pyyntöjen vastaus haetaan staattisille tiedoille avatusta välimuistista, vaikka sovellus olisi yhteydessä verkkoon. Mikäli tietoja ei löydy välimuistista, ne haetaan verkosta ja tallennetaan välimuistiin myöhempää

käyttöä varten. Käyttäjän verkkoyhteydettömässä tilassa tekemän kirjauksen sisältö selvitetään ja pyynnön tiedot välitetään toiminnolle, joka tallentaa kirjauksen selaimen paikalliseen tietokantaan.

3.3.2 IndexedDB

Service Worker ei kykene ilman verkkoyhteyttä tallentaa tietoja selaimen välimuistiin. Tämän vuoksi POST- ja PUT-pyynnöt tallennetaan selaimen paikalliseen IndexedDB-tietokantaan, kun käyttäjä tekee offline-tilassa kirjauksen. Service Worker -koodiin lisätään toiminnot selaimen paikallisen tietokantarajapinnan käyttämiseksi. Selaimen paikallista tietokantarajapintaa voidaan käyttää riippumatta verkkoyhteydestä.

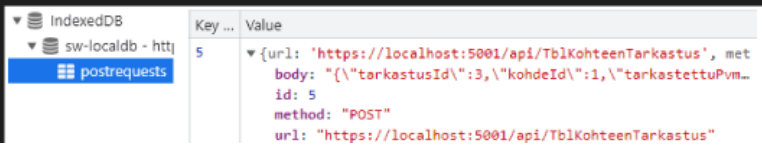
Service Worker pyytää selaimen paikalliselta IndexedDB-tietokantarajapinnalta tietokannan ja oliovaraston avausta sen asennuksen yhteydessä. Mikäli sovelluksen käyttöön ei ole aiemmin avattu tietokantaa ja sen oliovarastoa, ne luodaan Service Workerin asennuksen yhteydessä. Kuvassa 42 esitetään koodi, joka avaa tietokannan.

```
// IndexedDB avaaminen / luominen kun SW asennetaan
const openDatabase = () => {
    let openindexedDB = indexedDB.open(IDB_NAME, IDB_V) // avataan / luodaan tietokanta
    openindexedDB.onerror = (err) => { // jos selain ei tue
        console.log('IndexedDB error: ', err)
    }
    openindexedDB.onupgradeneeded = (event) => { // kun tietokanta luodaan
        event.target.result.createObjectStore(IDB_STORE, { // luodaan oliovarasto
            autoIncrement: true, keyPath: 'id' // juokseva id
        })
        console.log('IndexedDB luotu: ', event.target.result)
    }
}
```

Kuva 42. Tietokannan ja oliovaraston avaus

Kuvassa 42 esiintyvää toimintoa kutsutaan, kun Service Worker asennetaan. Selaimen paikalliseen IndexedDB-tietokantarajapintaan luotua tietokantaa käyttävät toiminnot avaavat tietokannan itse käyttöönsä. Esimerkiksi kuvassa 43 esiintyvä toiminto, joka tallentaa saamansa HTTP-pyynnön kyseiseen tietokantaan, avaa aluksi tietokannan. Kuvassa 43 esitetään myös esimerkki selaimen paikalliseen IndexedDB-tietokantarajapintaan avatun tietokannan ’sw-localdb’ oliovarastoon ’postrequests’ tallennetusta oliosta, kun kohteen kirjaus on tehty offline-tilassa ja kirjauksen tiedot ovat tallennettu paikallisesti.

```
// Kun fetch-metodi POST tai PUT offline-tilassa tallennetaan pyynnöt IndexedDB
const saveIntoIDB = (url, method, body) => {
    let request = {}
    request.url = url;
    request.method = method;
    body = JSON.parse(body);
    request.body = JSON.stringify(body);
    let db = indexedDB.open(IDB_NAME, IDB_V)
    db.onsuccess = (event) => {
        let tx = event.target.result.transaction(IDB_STORE, 'readwrite'); // transaktion määritys
        let store = tx.objectStore(IDB_STORE); // oliovarasto
        let adding = store.add(request); // tallennetaan pyyntöolio oliovarastoon
        adding.onsuccess = () => {
            messageToClient("Kirjauksesi on tallennettu paikallisesti."); // viesti reactappille (snackbar)
        }
        adding.onerror = (err) => {
            console.log(err);
        }
    }
}
```



Kuva 43. Tiedon tallennus selaimen paikalliseen tietokantaan

Kuvassa 43 esiintyvää toimintoa kutsutaan, kun sovellus ei ole yhteydessä verkkoon ja selain lähettää POST- tai PUT-pyynnön kirjauksen tallentamiseksi. Verkkoyhteyden palautuessa kuvassa 44 esiintyvä toiminto siirtää selaimen paikalliseen tietokantaan tallennetut pyynnöt palvelimelle.

```
/* Kun sovellus on yhteydessä verkkoon, checkConnection() kutsuu tätä,
 * haetaan ja lähetetään tiedot IndexedDB:stä palvelimelle */
const sendOfflinePosts = async () => {
    let db = indexedDB.open(IDB_NAME, IDB_V)
    db.onsuccess = (event) => {
        let tx = event.target.result.transaction(IDB_STORE, 'readwrite'); // transaktion määritys
        let store = tx.objectStore(IDB_STORE); // oliovarasto
        let allData = store.getAll(); // kaikki varaston oliot
        allData.onsuccess = () => {
            if (allData.result.length > 0) {
                allData.result.forEach(data => { // kaikki indexeddbn postrequests varastoon tallennetut tiedot
                    if (data.method === "POST") {
                        fetch(data.url.toString(), {
                            method: "POST",
                            body: data.body,
                            headers: {
                                'Content-Type': 'application/json',
                            },
                        }).then((response) => {
                            console.log("response: ", (response.status))
                            if (response.status === 201) { // response status 201 created
                                messageToClient("Kirjauksesi on tallennettu palvelimelle."); // viesti reactappille, snackbar
                                deleteRequest(data.id); // poistetaan tieto indexeddb varastosta
                            }
                        }). catch(err => console.log(err))
                    }
                    if (data.method === "PUT") {
                        fetch(data.url.toString(), {
                            method: "PUT",
                            body: data.body,
                            headers: {
                                'Content-Type': 'application/json',
                            },
                        }).then((response) => {
                            console.log("response: ", (response.status))
                            if (response.status === 204) { // response status 204 no content
                                messageToClient("Kirjauksesi on tallennettu palvelimelle."); // viesti reactappille, snackbar
                                deleteRequest(data.id); // poistetaan tieto indexeddb varastosta
                            }
                        }).catch(err => console.log(err))
                    }
                })
            }
        }
    }
}
```

Kuva 44. Tietojen siirto palvelimelle selaimen paikallisesta tietokannasta

Verkkoyhteyden tarkastaminen käynnistyy Service Workerin aktivoinnin yhteydessä, checkConnection-toiminnossa, joka jatkaa verkkoyhteyden tarkastamista 20 sekunnin välein ja kutsuu kuvassa 44 esiintyvää toimintoa. Onnistuneen palvelimelle lähetyksen jälkeen olio poistetaan selaimen paikallisen tietokannan oliovarastosta. Lisäksi lähetetään viesti selaimelle, joka näytetään ilmoituksena käyttöliittymässä. Kuvassa 45 esitetään toiminnot poiston tekemiseksi ja viestin lähettämiseksi.

```
// olion poisto onnistuneen palvelimelle lähetyksen jälkeen
const deleteRequest = (dataid) => {
    let db = indexedDB.open(IDB_NAME, IDB_V)
    db.onsuccess = (event) => {
        let tx = event.target.result.transaction(IDB_STORE, 'readwrite');
        let store = tx.objectStore(IDB_STORE);
        let del = store.delete(dataid);
        del.onsuccess = () => {
            console.log("Deleted");
        }
        del.onerror = (err) => {
            console.log(err);
        }
    }
}

/* viesti selaimelle,
   määritetty lähettämään string onnistuneista tallennuksista (IndexedDB tai sieltä eteenpäin palvelimelle). */
const messageToClient = (msg) => {
    self.clients.matchAll({ type: 'window' }).then(clients => {
      clients.forEach(client => {
          client.postMessage({
              msg: msg, // komponentti OnOffSnackbar vastaanottaa viestin (App.js)
          })
      });
    })
}
```

Kuva 45. Toiminnot tiedon poistamiseksi ja viestin lähettämiseksi

Viesti lähetetään myös onnistuneesta tiedon tallennuksesta selaimen paikalliseen tietokantaan. Selainsovelluksen OnOffSnackbar-komponentti vastaanottaa Service Workerin lähettämät viestit. Viestin lähettämiseksi Service Worker käyttää oikean asiakkaan (client) löytämiseksi matchAll-menetelmää (kuva 45). Muita asiakkaita voisivat olla muut worker-koodit.

Kun Service Worker -koodiin saatiin lisättyä esitellyt toiminnot vuorovaikuttamaan IndexedDB:n kanssa, selainsovelluksen päätoiminnot toimivat ilman verkkoyhteyttä.

4 PÄÄTÄNTÖ

Opinnäytetyön tavoitteet saavutettiin ja opinnäytetyön tuloksena kehitetty sovellus toimii kuten työn alussa määritettiin. Tekniikat toimivat hyvin yhteen ja soveltuivat kehitystarpeeseen. ASP.NET Coren ja Reactin käyttö yhdessä oli sujuvaa projektimallilla. React oli hyvä valinta käyttöliittymän toteutukseen sen

komponenttipohjaisuuden vuoksi. Käyttöliittymä on helposti jatkokehitettävissä. Joitain kehityskohteita havaittiin prosessin aikana. Sovelluksen testaus jäi myöhempään vaiheeseen, joten myöhemmin voi selvitä muitakin kehityskohteita. Jatkokehityksessä tulisi tarkastaa, onko virheet käsitelty käyttäjäystävällisesti ja käyttöliittymän päänäkymät voisi sivuttaa - käyttäjälle näytettäisi esimerkiksi 10 kohdetta yhdellä sivulla. Kirjauksen tallentamista voisi kehittää käyttäjäystävällisemmäksi tallentamalla tiedot tietokannan lisäksi myös React-Hookiin. Koska tietokantataulut tehtiin suomen kielellä, osa koodeissa käytetyistä funktioista ja muuttujista nimettiin myös suomen kielellä, jotta ne vastaavat tietokannassa käytettyjä nimiä. Tietokannan ja kaikki koodit voisi jatkokehityksessä kääntää englannin kielelle. Tietokannan toimivan rakenteen tärkeys työssä korostui ja jatkokehityksessä voisi tarkastella sen toimivuutta vielä uudestaan.

Opinnäytetyön tavoitteena oli kuvata työssä toteutettavan sovelluksen kehitystä sekä selvittää, mitä vaaditaan selainsovelluksen toimintaan offline-tilassa. Toiminta offline-tilassa progressiivisen verkkosovelluksen tekniikoin toteutui hyvin ja vastaus saatiin raportoitua riittävän selvästi. Selainsovelluksen koodia syntyi paljon, joka loi haasteita raportointiin. Opinnäytetyön tavoite saavutettiin käymällä läpi sovelluksen toteutus alusta asti ja kuvaamalla sen päätoiminnot sekä palvelin- ja selainsovelluksen toimivuutta yhdessä.

Kehitystyössä eniten aikaa vei Service Workerin kehitys ja siihen liittyvien asioiden opiskelu. Workbox-kirjaston dokumentaatio ei riittänyt Service Workerin toimintaperiaatteiden perusteelliseen ymmärtämiseen ja Workboxia käyttämällä olisin voinut luoda toimivan Service Worker -koodin ilman, että olisin juurikaan ymmärtänyt tekniikan toimintaa. Tämä lopulta johti siihen, ettei Workboxia käytetty työssä. Selaimen IndexedDB-tietokantarajapinnan käyttäminen oli ennakko-oletuksiani helpompaa, mutta sen käyttäminen työssä vaati melko paljon koodia.

Opinnäytetyön myötä ymmärrykseni sovellusarkkitehtuurista ja rajapintojen toiminnasta vahvistui, joka oli tärkein oppimistavoitteeni. Käyttöliittymän muotoilussa auttoi mahdollisimman pitkälle tehty suunnitelma elementtien sijainneista. Selainpuolen toimintojen suunnittelu ennen kehityksen aloittamista olisi voinut olla parempaa ja jatkossa varaan suunnittelutyöhön hieman enemmän

aikaa. Selainpuolen koodia voi olla paikoin ulkopuolisen hankala seurata, joten jälkeen päin olen tarkentanut koodien toimintaa paremmalla kommentoinnilla. Tulevissa projekteissani osaan kiinnittää enemmän huomiota koodin kommentointiin. Koodin lukemista voi hankaloittaa kaksikielisyys, mutta vastaavasti suomenkieliset tietokantataulut selkeyttävät opinnäytetyön raporttia.

Työssä tutustuin perusteellisesti progressiivisen verkkosovelluksen toteutukseen ja käyttötarkoituksiin. Uskon PWA-tekniikoiden kehittyvän ja käytön yleistyvän tulevaisuudessa. Mielestäni kehitystä PWA-tekniikoissa vaatisi tietojen tallennustapa verkkoyhteydettömässä tilassa. IndexedDB-tietokantarajapinnan käyttäminen tallentamiseen vaatii tapahtumapohjaisuuden vuoksi paljon koodia tai ulkoisen kirjaston käyttämistä. Lisäksi kahden tallennuspaikan – selaimen välimuisti ja tietokanta – käyttäminen on työläämpää kuin vain yhden.

LÄHTEET

Archibald, J. 2014. The Offline Cookbook. Google Developers. WWW-dokumentti. Päivitetty 28.9.2020. Saatavissa: https://web.dev/offline-cookbook/ [viitattu 4.9.2021].

Chiarelli, A. 2018. Beginning React. E-kirja. Birmingham: Packt Publishing Ltd. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 7.9.2021].

EntityFrameworkTutorial.net. 2020. Development Approaches with Entity Framework. WWW-dokumentti. Saatavissa: https://www.entityframeworktutorial.net/choosing-development-approach-with-entity-framework.aspx [viitattu 14.10.2021].

Evjen, B., Hanselman, S. & Rader, D. 2008. Professional ASP.NET 3.5 In C# and VB. Introduction. E-kirja. Indianapolis: Wiley Publishing Inc. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 4.9.2021].

Facebook Open Source. 2021a. Components and Props. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/components-and-props [viitattu 7.9.2021].

Facebook Open Source. 2021b. Hooks FAQ. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/hooks-faq [viitattu 7.9.2021].

Facebook Open Source. 2021c. Introducing JSX. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/introducing-jsx.html [viitattu 7.9.2021].

Facebook Open Source. 2021d. State and lifecycle. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/state-and-lifecycle [viitattu 7.9.2021].

Facebook Open Source. 2021e. Thinking in React. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/thinking-in-react [viitattu 7.9.2021].

Hiwarale U, 2018. How to use IndexedDB to build Progressive Web Apps. WWW-dokumentti. Päivitetty 4.4.2018. Saatavissa: https://medium.com/jspoint/indexeddb-your-second-step-towards-progressive-web-apps-pwa-dcbcd6cc2076 [viitattu 5.9.2021].

Hobab, H. 2019. Stack Overlflow. ASP.NET Core form POST results in a HTTP 415 Unsupported Media Type response. Keskusteluryhmän artikkeli. Päivitetty 12.10.2019. Saatavissa: https://stackoverflow.com/questions/44538772/asp-net-core-form-post-results-in-a-http-415-unsupported-media-type-response/58826173#58826173 [viitattu 9.9.2021].

Internap Corporation. 2019. The basics of a Microsoft SQL Server architecture. Blogi. Päivitetty 3.8.2019. Saatavissa: https://www.inap.com/blog/microsoft-sql-server-architecture/ [viitattu 7.9.2021].

Kasundra, P. 2021. 5 Reasons to use Nodejs with React for web development. Blogi. Päivitetty 9.9.2021. Saatavissa: https://www.simform.com/blog/use-nodejs-with-react/ [viitattu 10.9.2021].

Masood-Al-Farooq, B. 2014. SQL Server 2014 Development Essentials. E-kirja. Birmingham: Packt Publishing Ltd. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 14.10.2021].

MDN Web Docs. 2021a. How to make PWAs installable. WWW-dokumentti. Päivitetty 24.2.2021. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Installable_PWAs [viitattu 4.9.2021].

MDN Web Docs. 2021b. Introduction to progressive web apps. WWW-dokumentti. Päivitetty 19.8.2021. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction [viitattu 4.9.2021].

MDN Web Docs. 2021c. IndexedDB key characteristics and basic terminology. WWW-dokumentti. Päivitetty 1.9.2021. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Terminology [viitattu 5.9.2021].

Microsoft. 2020a. Routing in ASP.NET Core. WWW-dokumentti. Päivitetty 4.1.2020. Saatavissa: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-3.0 [viitattu 4.9.2021].

Microsoft. 2020b. Entity Framework 6. WWW-dokumentti. Päivitetty 14.10.2020. Saatavissa: https://docs.microsoft.com/fi-fi/ef/ef6/ [viitattu 14.10.2021].

Microsoft. 2021a. SpaApplicationBuilderExtensions.UseSpa Method. WWW-dokumentti. Päivitetty 10.3.2021. Saatavissa: https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.spaapplicationbuilderextensions.usespa [viitattu 14.10.2021].

Microsoft. 2021b. Compare EF Core & EF6. WWW-dokumentti. Päivitetty 15.7.2021. Saatavissa: https://docs.microsoft.com/fi-fi/ef/efcore-and-ef6/ [viitattu 14.10.2021].

Microsoft. 2021c. Database Checkpoints (SQL Server). WWW-dokumentti. Päivitetty 6.8.2021. Saatavissa: https://docs.microsoft.com/en-us/sql/relational-databases/logs/database-checkpoints-sql-server [viitattu 14.10.2021].

Microsoft. 2021d. What is ASP.NET? .NET Series. WWW-dokumentti. Saatavissa: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet [viitattu 4.9.2021].

Parahar, M. 2019. Difference between RDBMS and OODBMS. Blogi. Päivitetty 27.10.2019. Saatavissa: https://www.tutorialspoint.com/difference-between-rdbms-and-oodbms [viitattu 14.10.2021].

Patel, H. 2020. React component lifecycle methods with React hooks. WWW-dokumentti. Päivitetty 21.9.2020. Saatavissa: https://betterprogramming.pub/react-component-lifecycle-methods-with-react-hooks-efcd04987805 [viitattu 7.9.2021].

Posnick, J. 2018. Workbox: your high-level service worker toolkit. Blogi. Päivitetty 5.10.2018. Saatavissa: https://web.dev/workbox/ [viitattu 4.9.2021].

Ratcliffe, S. 2018. ASP.NET Core 2 and Vue.js. E-kirja. Birmingham: Packt Publishing Ltd. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 4.9.2021].

Sufiyan, T. 2021. Introduction to React and its features. WWW-dokumentti. Päivitetty 9.4.2021. Saatavissa: https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs [viitattu 7.9.2021].

Tecci. 2020. React – Frontend kehityksen tetris! Blogi. Päivitetty 9.9.2020. Saatavissa: https://tecci.fi/2020/09/09/react-frontend-kehityksen-tetris/ [viitattu 10.9.2021].