Juho Järvilehto

Progressiiviset verkkosovellukset ja Electron työpöytäsovellusten kehityksessä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

9.6.2025

Tiivistelmä

Tekijä:	Juho Järvilehto
Otsikko:	Progressiiviset verkkosovellukset ja Electron työpöytäsovellusten kehityksessä
Sivumäärä:	30 sivua
Aika:	9.6.2025
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaajat:	Lehtori Peter Hjort

Tämä insinöörityö tutkii vaihtoehtoisia ratkaisuja alustapohjaisille natiivisovelluksille hyödyntäen Electron-sovelluksia ja progressiivisia verkkosovelluksia (PWA). Työssä käsitellään PWA- ja Electron-sovelluksien perusteita ja verkkokehityksen perusteita.

Insinöörityöprojektissa kuvataan jaetun palvelimen, Electron-sovelluksen ja PWA-sovelluksen suunnittelu- ja kehitysvaiheet. Työssä kehitettävät sovellukset toimivat verkottomassa tilassa, tallentavat käyttäjän muistiinpanot laitteen paikalliseen tietokantaan ja synkronoivat paikalliset muutokset jaetulle palvelimelle.

Insinöörityön yhteenvedossa tarkastellaan eroja Electron- ja PWA-sovellusten kehityksessä, resurssivaatimuksissa ja käyttötarkoituksessa. Lopuksi insinöörityössä perustellaan mikä sovelluskehys tulee valita vaatimusten mukaan ja todetaan, että sovelluskehykset ovat kehittyneet tarpeeksi tarjoamaan natiivisovellusta vastaavan kokemuksen suorituskyvyssä ja käytettävyydessä ilman kompromisseja.

Avainsanat:	Electron, progressiivinen verkkosovellus, verkkokehitys, natiivisovellukset

---

Abstract

Author:	Juho Järvilehto
Title:	Development of Desktop Applications Using Progressive Web Apps and Electron
Number of Pages:	30 pages
Date:	9 June 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Mobile Solutions
Supervisors:	Peter Hjort, Senior Lecturer

This study investigates the viability of Electron and Progressive Web Applications (PWA) as alternatives to native applications. It provides an overview of the fundamentals of Electron and PWA technologies and their development, as well as the essential concepts of web development required for developing Electron and PWA applications.

The development process of a shared server for an Electron application and a PWA application was studies in this project. The developed application allows users to create notes, which are stored locally on the device and synchronized with the shared server's database. Both applications are designed to function offline.

The development processes and use cases of each technology were also compared in this study. The thesis concludes by summarizing the key differences between Electron and PWA technologies in development, use case, and hardware requirements. Concluding that both technologies have developed enough to provide a native application experience without compromise in user experience and performance.

Keywords: Progressive Webapps, Electron, web development, native applications

Sisällys

1 Johdanto

Koska verkossa asioidaan enemmän kuin koskaan, on tärkeää, että verkkosovellus tarjoaa nopean ja luotettavan käyttökokemuksen verkkoyhteyden tilasta riippumatta. Verkkosovellukset ovat kehittyneet muistuttamaan alustapohjaisia natiivisovelluksia mahdollistamalla ennen vain natiivisovelluksille saatavilla olevia ominaisuuksia, kuten asennettavuuden ja kyvyn toimia verkottomassa tilassa.

Progressiiviset verkkosovellukset (PWA, Progressive Web Application) ovat olleet saatavilla mobiililaitteilla vuodesta 2015 ja työpöytäympäristössä vuodesta 2018. PWA-sovellukset kehitetään verkkokehityksessä käytettävillä teknologioilla mahdollistaen PWA-sovellusten asennuksen useille alustoille ilman alustapohjaisia kehitystyökaluja. Mahdollisuus kehittää sovellus ilman alustapohjaisia kehitystyökaluja tekee sovelluksesta saavutettavan usealle alustalle yhdellä koodikannalla. PWA-sovellusten jatkuva kehitys ja tuki Googlelta tuo kehittäjien käyttöön ominaisuuksia, jotka saavat PWA-sovellukset toimimaan lähestulkoon kuin natiivisovellukset [1].

Verkkosovellusten kehityksessä käytettävien teknologioiden hyödyntäminen työpöytäsovellusten kehityksessä on mahdollista käyttämällä ohjelmistokehyksiä kuten Electron, joka yhdistää selainmoottorin ja Node-ajoympäristön. Electron-sovelluksessa selainmoottori luo graafisen käyttöliittymän ja Node-ajoympäristö mahdollistaa pääsyn asennetun laitteen rajapintakutsuihin. Pääsy laitteen rajapintakutsuihin antaa Electron-sovellukselle natiivisovelluksen ominaisuudet. [2, 3.]

Insinöörityössä tutkitaan työpöytäsovellusten kehitystä luomalla PWA- ja Electron-sovellukset. Kehitysprosessin aikana tutkitaan sovelluskehyksien ominaisuuksia ja eroja sovelluskehityksessä.

2 Progressiivinen verkkosovellus

Progressiivinen verkkosovellus on verkkosovellus, joka voidaan asentaa käyttäjän laitteelle natiivisovelluksen tapaan. PWA-sovellukset ovat tapa tuoda verkkosovellusten ja verkkokehityksen tarjoamat ominaisuudet kuten alustojen välinen yhteensopivuus, laaja saavutettavuus ja verkkokehityksessä käytettävät työkalut kokonaisuuteen, joka muistuttaa natiivipohjaista sovellusta käyttökokemukseltaan. Natiivit alustapohjaiset sovellukset vaativat, että sovellus kehitetään uudestaan eri koodikantaan, jos halutaan tukea useita alustoja, kuten Windows tai OSX. PWA-sovellusten tarjoama yhteensopivuus ja laaja saavutettavuus näkyvät parhaiten siinä, että sovellus tukee useita alustoja käyttäen yhtä koodikantaa.

PWA-sovellukset ovat asennettavia verkkosovelluksia, mikä tarkoittaa, että PWA-sovellukset kehitetään käyttämällä verkkokehityksen perusperiaatteita ja työkaluja. Seuraavaksi käydään läpi verkkokehitykselle keskeisiä käsitteitä, kuten HTML, CSS, JavaScript ja Document Object Model, jotta saadaan parempi käsitys verkkokehityksestä, PWA-sovellusten toiminnallisuuksista, ominaisuuksista ja kehittämisestä. [3, 4.]

Verkkosivut rakennetaan käyttäen useita eri teknologioita, joista keskeisimpinä ovat HTML-, CSS- ja JavaScript-tiedostot. HTML-tiedostot määrittelevät verkkosivun rakenteen ja sisällön, kuten tekstit, painikkeet, kuvat ja verkkosivulla näkyvät linkit [5]. Esimerkkikoodi 1 sisältää esimerkin HTML-tiedostosta ja sen rakenteesta.

```
<!Doctype html>
<html>
<head>
    <title>Esimerkki</title>
    <link rel=”stylesheet” href=”styles.css”>
</head>
<body>
    <h1>Lorem Ipsum</h1>
    <p>esimerkkisivu</p>
    <bytton id=”painike”>Paina tästä</button>
    <script src=”script.js”></script>
</body>
</html>
```

Esimerkkikoodi 1. HTML-tiedosto määrittelee verkkosivun sisällön ja rakenteen käyttäen HTML-elementtejä.

Verkkosivun ulkoasu muotoillaan käyttäen CSS-tiedostoja. CSS-tiedostoilla voidaan määrittää sivustossa näkyvien elementtien asettelu, tyylitys ja koko [6]. Esimerkkikoodi 2 on esimerkki CSS-tiedostosta, jo verkkosivun rungolle, painikkeille, tekstille ja otsikolle.

```
body {
     font-family: Arial;
     background-color: #ffffff
}
h1 {
     color: aliceblue;
}
button {
     background-color: blue;
     color: white;
     padding: 5px;
}
```

Esimerkkikoodi 2. CSS-tiedostossa määritetään HTML-elementtien tyylit, kuten väri, ilmavuus ja kirjasintyyli.

Document Object Model (DOM) esittää HTML-tiedoston rakenteen ja elementit hierarkkisena puurakenteena. Jokainen puun haara sisältää olioita, joita voi muokata käyttäen JavaScriptiä. JavaScript-tiedostoilla lisätään verkkosivuille toiminnallisuuksia, kuten käyttäjän toimenpiteisiin reagointi, animaatioiden hallinta ja dynaamisten elementtien hallinta. JavaScriptillä ja DOM:illa on mahdollista muuttaa dokumentin rakennetta, tyyliä ja sisältöä lataamatta verkkosivua uudelleen. [7]. Esimerkkikoodi 3 käsittelee painike elementin

noutamista HTML-tiedostosta lisäämällä siihen kuuntelijan, joka reagoi lähettämällä viestin, kun käyttäjä painaa painiketta verkkosivulla.

```
document.getElementById(“painike”).addEventListener(“click”, () => {
     alert(“Painiketta painettu”);
}
```

Esimerkkikoodi 3. HTML-elementti tunnisteella ”painike” noudetaan JavaScriptillä ja lisätään tapahtumakuuntelija, joka reagoi viestillä, kun käyttäjä painaa painiketta.

Verkkosivut ja verkkosovellukset toimivat selaimessa, mikä tarkoittaa, että verkottomassa tilassa verkkosivusta ja verkkosovelluksesta tulee käyttäjälle saavuttamattomat. PWA-sovellukset ovat asennettavissa käyttäjän laitteelle, joten oletetaan, että sovellus ylläpitää käytettävyyden täysin tai osittain tilassa, jossa verkkoyhteys on heikko tai olematon. PWA-sovellukset kykenevät toimimaan verkkotilanteesta huolimatta käyttäen service worker (SW) -tiedostoa. service worker -tiedosto mahdollistaa, että sovellus toimii luotettavasti, tehokkaasti ja nopeasti verkkotilasta huolimatta. Seuraavassa luvussa syvennytään service worker -tiedoston toimintaan, ominaisuuksiin ja tarkoitukseen PWA-sovelluksessa.

2.1 Service Worker

Natiivit alustapohjaiset sovellukset toimivat osittain tai kokonaan verkottomassa tilassa. Sovellusten asennuksen jälkeen toimintaan tarvittavat tiedostot löytyvät laitteen muistista paikallisesti. PWA-sovellukset saavat natiivisovellusten ominaisuuksia käyttämällä service worker -tiedostoa.

Service worker on JavaScript-tiedosto, joka asennetaan laitteelle PWA-sovelluksen asennuksen yhteydessä. Service workerin tarkoitus on toimia välityspalvelimena sovelluksen, verkon ja palvelimen välillä. Service worker toimii itsenäisesti sovelluksen elinkaaresta riippumatta, mikä mahdollistaa sovelluksen tietojen päivittämisen, verkkopyyntöjen hallitsemisen ja useiden eri

toimintojen suorittamisen sovelluksen käytön taustalla, sekä verkottomassa ja suljetussa tilassa. [9].

Välityspalvelimena toimiminen tarkoittaa, että SW-tiedosto kuuntelee sovelluksesta lähteviä ja tulevia verkkokutsuja. SW-tiedosto kykenee reagoimaan ja vastaamaan verkkokutsuihin ennalta määritetyllä tavalla. Verkkokutsuihin vastaaminen vaatii tiedostojen hallintaa, sillä usein verkkopyyntöön vastaaminen SW-tiedostosta vaatii, että sovelluksen tarvittava resurssi palautetaan käyttöliittymälle laitteen tietokannasta tai muistista. [8, 9, 10.] Seuraavaksi käsitellään PWA-sovelluksen strategioita verkkopyyntöjen käsittelyyn ja tiedostojen hallintaan.

2.2 Tiedostojen hallinta

Natiivisovelluksen toimintaan tarvittavat tiedostot ja resurssit löytyvät laitteen muistista asennuksen jälkeen. Kun verkkosivu tai PWA-sovellus avataan, sovellukset tekevät verkkopyyntöjä palvelimelle saadakseen tarvittavat tiedostot sovelluksen toimintaa varten. Verkkoyhteyden ollessa heikko tai olematon sovellus ei saa vastausta palvelimelta, mikä tarkoittaa, että sovellus on käyttökelvoton. [10].

SW-tiedosto mahdollistaa PWA-sovellusten toiminnan verkottomassa tilassa hallitsemalla, kuinka verkkopyyntöihin vastataan ja miten PWA-sovelluksen toimintaan tarvittavat tiedostot tallennetaan laitteelle. PWA-sovellukset tallentavat tarvittavat tiedostot ja resurssit sovelluksen välimuistiin SW-tiedoston avulla. SW-tiedosto kuuntelee sovelluksen tekemiä verkkopyyntöjä, kun sovellus avataan, ja noutaa tarvittavat tiedostot ja resurssit sovelluksen välimuistista. Jos sovelluksen välimuistista ei löydy tarvittavia tiedostoja, noudetaan tiedostot verkosta ja tallennetaan välimuistiin kyseisen verkkopyynnön varalta. [10,11.]

PWA-sovellusten välimuistiin ei tallenneta tiedostoja, jotka eivät ole sovelluksen toiminnalle tai latausnopeuden parantamiseksi tarpeen. PWA-sovellukset

hyödyntävät No-SQL IndexedDB tietokantaa, joka löytyy jokaisesta selaimesta. IndexedDB-tietokantaa käytetään käyttäjän luoman sisällön, rakenteellisen datan ja suurien tiedostojen tallentamiseen käyttäjän laitteelle. [12, 13.]

Välimuistissa olevat tiedostot ja laitteen paikallinen IndexedDB-tietokanta mahdollistavat sovelluksen käyttöliittymän ja sisällön käytön verkottomassa tilassa. IndexedDB:n avulla on mahdollista muokata sovelluksen sisältöä verkottomassa tilassa sekä päivittää tiedot sovelluksen ja palvelimen välillä käyttäen synkronointipyyntöjä. [12, 13.]

PWA-sovellukset käyttävät useita eri strategioita verkkopyyntöjen käsittelyyn ja tiedostojen noutamiseen. Sovelluksessa käytettävät strategiat ovat Network First, Cache First, Cache Only ja Network Only. [10, 11.]

Network first -strategiassa sovelluksen sisällön ajantasaisuus on ensisijaista. Sovellus tekee verkkopyynnön ja odottaa vastausta palvelimelta. Jos verkkopyyntö epäonnistuu, sovellus hakee tallennetuista tiedostoista verkkopyyntöä vastaavat tiedot. On huomioitava, että tiedostojen noutaminen verkosta on hidasta verrattuna tietojen noutamiseen laitteen tietokannasta. Esimerkkikoodi 4 on kuuntelija, joka reagoi sovelluksen verkkopyyntöihin suorittamalla funktion. Suoritettava funktio odottaa vastausta verkkopyynnölle ja palauttaa vastauksen tiedot [10.]

```
self.addEventListener("fetch", async (event) => {
    try{
       const networkResponse = await fetch(event.request);
       if(networkResponse.ok){
         return networkResponse
       }
     } catch (error){
       const cachedResponse = await caches.match(event.request);
       return cachedResponse || Response.error(error);
     }
  });
```

Esimerkkikoodi 4. Kuuntelija reagoi verkkopyyntöihin suorittamalla funktion, joka odottaa vastausta verkosta ja verkkopyynnön epäonnistuessa noutaa tiedot laitteen välimuistista.

Cache First -strategiassa sovellus tarkistaa laitteen muistin ja tietokannan verkkopyynnön toteuttamiseen tarvittavien tiedostojen varalta. Jos laitteen tietokannasta tai muistista ei löydy tarvittavia tiedostoja verkkopyynnön toteuttamiseen, tietojen noutamiseen käytetään verkkoyhteyttä. Kun sovellus saa vastauksen verkosta, sovellus tallentaa verkkopyynnön ja verkosta saadut tiedot laitteelle vastauksena kyseiseen verkkopyyntöön. Cache First -strategiassa ajoittain suoritetaan verkkopyyntö, vaikka vastaukseen tarvittavat tiedostot löytyvät laitteelta. Verkkopyyntö suoritetaan, kun halutaan päivittää tallennetun vastauksen tiedot uusimpaan sisältöön. Cache First -strategialla pyritään parantamaan sovelluksen suorituskykyä ja käyttökokemusta priorisoimalla nopeutta tiedon tuoreuden ja ajantasaisuuden kustannuksella. [10].

Network First - ja Cache First -strategiat hyödyntävät verkkoa sekä laitteen tietokantaa ja muistia pyyntöjen suorittamiseen molemmissa strategioissa. Cache Only - ja Network Only -strategiat pyrkivät suorittamaan verkkopyynnön käyttäen vain laitteen tietokantaa ja muistia tai verkkoyhteyttä. [11].

PWA-sovelluksissa hyödynnetään useita eri strategioita samanaikaisesti. On mahdollista käyttää Cache First -strategiaa sovelluksen käyttöliittymään tarvittavien tiedostojen noutamiseen ja Network First -strategiaa siinä osassa sovellusta, joka vaatii tietoja verkosta, kuten käyttäjien luomaa sisältöä. Seuraavaksi käsitellään Web App Manifest -tiedostoa, joka on PWA-sovelluksen toimintaan tarvittava tiedosto. [10, 11, 12, 13].

2.3 Web Application Manifest

Web App Manifest tiedosto sisältää PWA-sovelluksen asennukseen vaadittuja määritelmiä. Tiedosto sisältää JSON-tyylisen objektin, joka sisältää avaimia, jotka määrittelevät sovelluksen asennuksessa käytettäviä parametreja kuten: nimi, kuvaus, kuvakkeet, teemavärit, näytön orientaatio ja sovelluksen verkko osoite. Web App Manifest tiedosto mahdollistaa PWA-sovellusten asennuksen

laitteelle ja ilmoittaa verkkosovelluksessa vierailijoille, että verkkosovellus on asennettavissa. [14, 15].

3 Electron-viitekehys

PWA-sovellukset eivät ole ainoa tapa hyödyntää verkkokehityksen työkaluja työpöytäsovelluksen kehittämiseen. Electron.js on avoin ohjelmistokehityksen viitekehys, joka, kuten PWA-sovellukset, mahdollistavat työpöytäsovellusten kehittämisen verkkoteknologioilla yhdistämällä selaimen selainmoottorin ja Node.js-ajoympäristön. Selainmoottorin ja Node.js-ajoympäristön yhdistäminen tarkoittaa, että Electron-sovellukset ovat toiminnallisuudeltaan lähempänä natiivisovelluksia kuin PWA-sovelluksia, koska Node.js-ajoympäristö tarjoaa rajoittamattoman pääsyn asennetun laitteen rajapintakutsuihin ja resursseihin [2, 3].

Electron-sovellukset jaetaan kahteen osaan: Renderer- ja Main-prosessiin. Renderer-prosessi hyödyntää selainmoottoria sovelluksen käyttöliittymän luomiseen. Selainmoottorit löytyvät kaikista selaimista, ja niiden tehtävä on hallita ominaisuuksia kuten, verkkopyyntöjä tarvittavien resurssien ja tiedostojen noutamiseen. Selainmoottori on vastuussa myös navigoinnista selaimen verkkosivujen välillä, ja verkkosivun sisällön asettelusta ja ulkonäöstä, käyttämällä verkkopyynnöstä saatuja HTML-, CSS- ja JavaScript-tiedostoja. Electron-sovelluksessa HTML- CSS- ja JavaScript-tiedostot löytyvät laitteen tiedostoista, joten verkkopyyntöjen hallintaa ei vaadita verkottomassa tilassa toimintaan.

Main-prosessi toimii käyttäen Node.js-ajoympäristöä. Node.js-ajoympäristöä käytetään sovelluksen Main-prosessissa elinkaaren hallintaan ja rajapintakutsuihin asennetun laitteen järjestelmään ja resursseihin. Main-prosessi on Electron-sovelluksen versio SW-tiedostosta. Node-ajoympäristö mahdollistaa sovelluksen toiminnan verkottomassa tilassa, koska Node mahdollistaa ulkoisten kirjastojen ja moduulien käytön, kuten MySQL- ja Sqlite-tietokannat.

Main- ja Renderer-prosessit eivät kommunikoi toistensa välillä tietoturvallisuussyistä. Electron-sovelluksessa käytettävä Preload-skripti mahdollistaa prosessien välisen kommunikoinnin tietoturvallisesti käyttäen Electron-viitekehyksen Inter-Process Communication -kutsuja(IPC). IPC-kutsuilla sallitaan tiedon välitys prosessien välillä. Tiedon välittäminen mahdollistaa tiedon lähetyksen tiedostojen tallennukseen, käyttöliittymän päivitykseen ja alustan rajapintakutsujen käsittelyyn. [16, 17]. Esimerkkikoodi 5 käsittelee prosessien välistä kommunikaatiota Main- ja Renderer-prosessin välillä IPC-kutsuilla. Main-tiedostossa määritetään IPC-kutsu, joka vastaa sovelluksen lähettämään sendData-pyyntöön suorittamalla vastaavan sendData-funktion. Preload-tiedostossa luodaan uusi rajapinta, jota voidaan kutsua Renderer-prosessissa tietojen lähettämiseen prosessien välillä.

```
//Main.js
app.whenReady().then(() => {
     ipcMain.handle("sendData", sendData)
})

const sendData = () => {
     console.log("Received IPC call from renderer.js" );
}


//Preload.js
contextBridge.exposeInMainWorld("electronAPI", {
     sendData: () => ipcRenderer.invoke("sendData");
});

//Renderer.js
saveButton.addEventListener("click", async() => {
     await window.electronAPI.sendData()
}
```

Esimerkkikoodi 5. Electron-sovelluksen IPC-kommunikaation toteutus, jossa Main- ja Renderer-prosessit kommunikoivat Preload-tiedoston välityksellä.

4 Sovellusten kehitys

Electron- ja PWA-sovellukset kehitetään tarjoamaan käyttäjälle ominaisuudet lisätä, poistaa, muokata ja tallentaa tehtäviä laitteelle ja jaetulle palvelimelle.

Sovellusten on toimittava verkottomassa tilassa ja synkronoitava muutokset tehtäviin palvelimen ja sovelluksen tietokantojen välillä, kun verkkoyhteys on saatavilla. Jaettu palvelin vastaa sovellusten HTTP-kutsuihin ja ylläpitää tietokantaa tehtävistä. Palvelin kehitetään Express.js-viitekehyksellä Node.js-ajoympäristössä.

4.1 Palvelimen kehitys

Palvelimen kehittäminen aloitetaan luomalla tiedosto, joka sisältää palvelimen toimintaan tarvittavat riippuvuudet Express.js- ja CORS-kirjastoihin. Express.js on Node-ajoympäristössä toimiva verkkokehityksen viitekehys, joka mahdollistaa HTTPS-kutsujen käsittelyn viitekehyksen rajapintakutsuilla ja reitityksen Express-väliohjelmistoilla. [18]. Cross-Origin Resource Sharing (CORS) -väliohjelmisto on HTTPS-kutsuihin pohjautuva mekanismi, joka sallii palvelimelle tehtyjen pyyntöjen käsittelyn, mitkä ovat alkuperäisesti palvelimen ulkopuolelta. CORS sallii palvelimen ja sovellusten väliset rajapintakutsut. [19] Sovellusten HTTPS-kutsut sisältävät JSON-tietoa, joten palvelimelle lisätään Express-väliohjelmisto, joka jäsentää JSON-tiedot HTTPS-kutsuista. Palvelin vastaa HTTPS-kutsuihin portilta 8080. Palvelimelle lisätään erillinen reitti tasks, joka vastaa sovellusten HTTPS-kutsuihin tietokantaan. Esimerkkikoodi 6 sisältää esimerkin palvelinkonfiguraatiosta, joka vastaa kutsuihin portilta 8080, käsittelee JSON-tietoa, hallinnoi CORS-pyyntöjä.

```
const express = require("express");
const cors = require("cors");
const app = express();
const port = 8080;

app.use(cors());
app.use(express.json());

const taskRouter = require("./routes/tasks");
app.use("/task", taskRouter);
app.listen(port, () => {
     console.log(`Listening port ${port}`);
});
```

Esimerkkikoodi 6. Palvelinkonfiguraatio, joka kuuntelee kutsuja portilta 8080, käsittelee JSON-tietoa, hallinnoi CORS-pyyntöjä ja sisältää task-reitin.

Palvelin tarvitsee tietokannan, joka säilyttää käyttäjän tehtäviä ja rajapintakutsut tietokannan käsittelyyn. Palvelimessa käytetään Sqlite3-tietokantaa, joka on kevyt tiedostopohjainen relaatiotietokanta [20]. Tietokanta sisältää listan tehtävistä, ja jokaisessa tehtävässä on tunniste, otsikko, sisältö, luomisaika, päivämäärä määräajalle, aika viimeisestä muokkauksesta ja lippu merkitsemään tiedon tilaa poistoa varten. Esimerkkikoodi 7 suorittaa funktion, joka luo uuden tietokannan, jos kyseistä tietokantaa ei valmiiksi löydy palvelimelta. Funktiossa määritetään myös taulukko, joka sisältää tehtävät ja yksittäiseen tehtävään kuuluvat kentät.

```
db.serialize(() => {
  db.run(
    `CREATE TABLE IF NOT EXISTS tasks (
    id INTEGER PRIMARY KEY,
    title TEXT NOT NULL,
    content TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    due_date TIMESTAMP,
    time_allocated INTEGER,
    last_modified TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    deleted INTEGER DEFAULT 0
)`
  );
});
```

Esimerkkikoodi 7. Tietokantafunktio luo palvelimelle tietokannan, jos sitä ei ole valmiiksi luotu. Tietokantaan luodaan taulukko nimellä tasks ja määritetään tehtävään kuuluvat kentät.

Palvelimella käsitellään HTTP-metodeja REST-rajapinnalla. GET- , POST- , PUT- ja DELETE-metodeilla suoritetaan CRUD-operaatioita, joilla lisätään, poistetaan, muokataan ja päivitetään tietokantaa. Esimerkkikoodissa 8 palvelin käsittelee POST-pyynnön, jonka mukana lähetetään tarvittavat tiedot uuden tehtävän luomiseen. Pyynnöstä vastaanotetaan tehtävän tiedot, jotka puretaan muuttujiksi id, title, content, due_date ja time_allocated. Muuttujia käytetään SQL-lauseessa, joka lisää uuden tehtävän tietokantaan tasks-taulukkoon.

```
router.post("/", (req, res) => {
  const { id, title, content, due_date, time_allocated } = req.body;
  console.log();
  const sql = `INSERT INTO tasks (id, title, content, due_date,
time_allocated) VALUES (?,?, ?, ?, ?)`;
  db.run(sql, [id, title, content, due_date, time_allocated], function
(err) {
    if (err) {
      return res.status(500).json({ error: err.message });
    }
    res.status(201).json({ id: this.lastID });
  });
});
```

Esimerkkikoodi 8. Käsittelee POST-pyynnön Express.js-palvelimella.

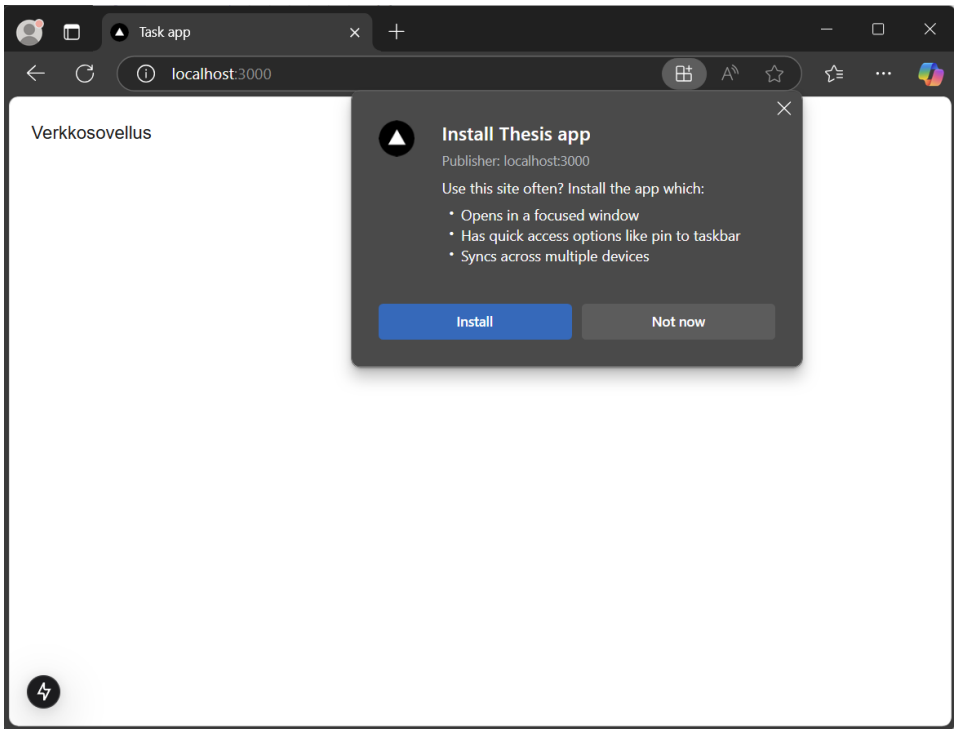
4.2 Progressiivisen verkkosovelluksen kehitys

PWA-sovellus alustetaan luomalla kansio, johon asennetaan React-pohjainen viitekehys Next.js. Next.js-viitekehys asennetaan komennolla ”npx create-next-app@latest”. Sovelluksen IndexedDB-tietokanta ja Cache-storage ovat käytettävissä ilman asennusta. Sovelluksen alustamisen jälkeen lisätään Manifest-tiedosto sovelluksen juurikansiossa sijaitsevaan julkiseen kansioon. Esimerkkikoodi 9 on esimerkki Manifest-tiedostosta, joka sisältää sovelluksen asennukseen vaadittavat metatiedot, kuten nimen, aloitusosoitteen, ikonin, näytön orientaation ja sovelluksen teeman värin.

```
{
  "name": "Thesis",
  "short_name": "Thesis",
  "start_url": "/?home=true",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "192x192",
      "type": "image/png",
      "purpose": "any"
    }
  ],
  "theme_color": "#000000",
  "background_color": "#FFFFFF",
  "display": "fullscreen",
  "orientation": "portrait"
}
```

Esimerkkikoodi 9. Manifest-tiedosto, joka sisältää metatietoja sovelluksen nimestä, aloitusosoitteesta, ikonista, teeman väristä, taustaväristä ja näytön orientaatiosta.

Kun Manifest-tiedosto on lisätty sovelluksen juurikansioon, sovellus on asennettavissa laitteelle selaimesta PWA-sovelluksena. Kuvassa 1 on esitetty selainikkunassa näkyvä ponnahdusikkuna, josta verkkosovellus voidaan asentaa laitteelle PWA-sovelluksena.



Kuva 1. PWA-sovelluksen asennusikkuna selaimessa.

Service worker-tiedosto vaaditaan PWA-sovelluksen toimintojen ylläpitoon verkottomassa tilassa ja tietokantojen hallintaan. Tiedostoon lisätään funktiot, jotka asentavat service workerin, kun verkkosovellus asennetaan laitteelle PWA-sovelluksena. Esimerkkikoodi 10 sisältää kaksi funktiota. Ensimmäinen funktio asentaa service workerin sovelluksen asennuksen yhteydessä. Toinen funktio aktivoi service workerin, kun sovellus avataan.

```
const installEvent = () => {
  self.addEventListener("install", (e) => {
    console.log("service worker installed");
  });
};

const activateEvent = () => {
  self.addEventListener("activate", () => {
    console.log("service worker activated");
  });
};
```

Esimerkkikoodi 10. Esimerkki funktioista, jotka kutsutaan, kun sovellus asennetaan ja aktivoidaan.

Työskentelijä aktivoidaan React-pohjaisissa viitekehyksissä React-komponentissa useEffect-hookilla. Työskentelijän aktivointi mahdollistaa sovelluksen tietokantojen hallinnan, offline-toiminnallisuudet ja tietokantojen synkronoinnin taustalla.

```
useEffect (() => {
     if(”serviceWorker” in navigator){
      navigator.serviceWoker.register(”sw.js”, {scope:”/”})
      .then(reg) => {
      console.log(“scope: “, reg.scope);
      }
     }
});
```

Esimerkkikoodi 11. UseEffect-funktio tarkistaa, onko työskentelijän asennus mahdollista. Jos asennus on mahdollista, rekisteröidään työntekijä tiedostosta ”sw.js” ja annetaan työskentelijälle pääsy kaikkiin tiedostoihin juurihakemistossa.

SW-tiedostoa käytetään sovelluksen tietokannan ja välimuistin hallintaan. Sovelluksessa käytetään IndexedDB-tietokantaa, joka on SQL-pohjainen relaatiotietokanta. Tietokantaa käytetään palvelimelta noudettujen tehtävien tallentamiseen ja tehtävien muokkausten tallentamiseen paikallisesti. Esimerkkikoodissa 12 määritellään funktio, joka avaa IndexedDB-tietokannan. Funktio palauttaa lupauksen, joka ratkaistaan onnistuneesti tietokannan avaamisen yhteydessä tai hylätään virheen sattuessa.

```
function openDatabase() {
  return new Promise((resolve, reject) => {
    const request = indexedDB.open("TaskAppDB", 1);

    request.onupgradeneeded = (event) => {
      const db = event.target.result;
      db.createObjectStore("tasks", { keyPath: "id", autoIncrement:
true });
    };

    request.onsuccess = () => resolve(request.result);
    request.onerror = () => reject(request.error);
  });
}
```

Esimerkkikoodi 12. openDatabase-funktio avaa yhteyden tietokannan tasks-tauluun.

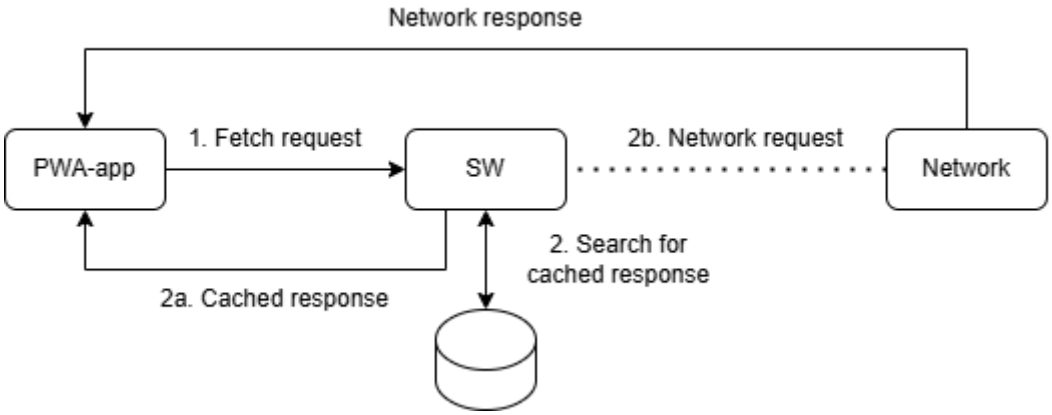
Tietokantaa hallitsevat funktiot tehtävien luontiin, lisäykseen, muokkaukseen ja poistoon lisätään SW-tiedostoon. Esimerkkikoodi 13 on funktio, joka hakee kaikki tiedot tietokannasta ja palauttaa kaikki tallennetut tehtävät taulukkona.

```
async function getAllData() {
  const db = await openDatabase();
  const transaction = db.transaction("tasks", "readonly");
  const objectStore = transaction.objectStore("tasks");

  return new Promise((resolve, reject) => {
    const req = objectStore.getAll();
    req.onsuccess = (event) => {
      resolve(event.target.result);
    };
    req.onerror = (event) => {
      console.log("Error in getAllData: ", event.target.error);
      reject(event.target.error);
    };
  });
}
```

Esimerkkikoodi 13. getAllData-funktio avaa yhteyden tasks-tietokantaan ja luo pyynnön tietojen noutoon tietokannasta. Onnistunut pyyntö palauttaa tietokannasta noudetut tiedot ja ilmoittaa virheestä, jos pyynnön toteutus tietojen noutoon epäonnistuu.

Verkkosovellus noutaa tiedot jaetulta palvelimelta, mikä tarkoittaa, että verkottomassa tilassa tietojen nouto palvelimelta on mahdotonta. PWA-sovelluksessa hyödynnetään Cache First -verkkostrategiaa ja paikallista tietokantaa verkkopyyntöjen toteuttamiseen verkottomassa tilassa. Cache First -

strategiassa sovelluksen verkkopyyntöihin noudetaan vastaus laitteen tietokannasta ja välimuistista. Tilanteessa, jossa vaadittavat tiedot puuttuvat laitteelta, varaudutaan verkkoyhteyteen tietojen noutamiseen palvelimelta. Palvelimelta noudetut tiedot tallennetaan tietokantaan ja välimuistiin saman kyseisen verkkopyynnön uudelleen suorittamisen varalta. Kuva 2 havainnollistaa, kuinka PWA-sovellus käsittelee fetch-pyyntöjä Service Workerin avulla. Prosessi alkaa, kun sovellus lähettää verkopyynnön. Service Worker tarkistaa ensin, löytyyko pyydetty resurssi laitteen tietokannasta ja sovelluksen välimuistista. Jos vastaus löytyy, se palautetaan sovellukselle. Jos vastausta ei löydy, Service Worker luo verkkopyynnön sovellusten jaetulle palvelimelle ja palauttaa vastauksen sovellukselle.



Kuva 2. PWA-sovelluksen Cache First-verkkostrategian toimintaperiaate.

Sovellukseen lisätään kuuntelija, joka mahdollistaa fetch-verkkopyyntöjen käsittelyn SW-tiedostossa. Fetch on moderni JavaScript-rajapinta, joka on käytettävissä verkkopyyntöjen luontiin ja vastaanottoon. Esimerkkikoodi 14 sisältää yksinkertaisen kuuntelijan, joka odottaa vastaanottaa sovelluksen verkkopyyntöjä. Kun kuuntelija havaitsee verkkopyynnön se tulostaa verkkopyynnön tiedot.

```
self.addEventListener(”fetch”, async (event) =>{
     console.log(“Fetch request: “, event);
}
```

Esimerkkikoodi 14. Fetch-kuuntelija suorittaa funktion, kun Fetch-pyyntö havaitaan ja tulostaa pyynnön sisällön.

Kuuntelija vastaanottaa sovelluksen verkkopyyntöjä ja käyttää Cache First-strategiaa noutamalla tiedot paikallisesta tietokannasta ja varautuu verkkoon, jos tiedot puuttuvat paikallisesta tietokannasta ja välimuistista. Esimerkkikoodi 15 on sovelluksen kuuntelijafunktio, joka vastaanottaa sovelluksesta tulevia verkkopyyntöjä tarkistaen kutsun tyypin ja osoitteen. Jos verkkopyyntö sisältää GET-pyynnön, kuuntelija noutaa tiedot laitteen tietokannasta ja välimuistista. Jos laitteen tietokannasta ja välimuistista ei löydy tarvittavia tietoja, kuuntelija noutaa tiedot verkosta. Verkosta noudetut tiedot tallennetaan laitteen tietokantaan ja välimuistiin toistuvan verkkopyynnön varalta.

```
self.addEventListener("fetch", async (event) => {
  const { request } = event;
  // Check if the request is a GET and includes "/tasks"
  if (request.url.includes("/tasks") && request.method === "GET") {
    event.respondWith(
      (async () => {
        try {
          // Try to get data from IndexedDB
          const data = await getAllData();
          if (data && data.length > 0) {
            return new Response(JSON.stringify(data), {
              headers: { "Content-Type": "application/json" },
            });
          }
          const networkResponse = await fetch(request);
          if (networkResponse.ok) {
            const clonedResponse = networkResponse.clone();
            const responseData = await clonedResponse.json();

            await serverToDatabase(Array.isArray(responseData)
            ? responseData : [responseData]);

            return networkResponse;
          }
        } catch (error) {
          return new Response(
            JSON.stringify({ error: "Failed to retrieve data" }),
            { headers: { "Content-Type": "application/json" } }
          );
        }
    }
```

Esimerkkikoodi 15. Kuuntelija, joka käsittelee sovelluksen GET-pyyntöjä hakemalla tiedon laitteen tietokannasta ja välimuistista varautuen tiedon noutamiseen verkosta, jos tieto ei ollut saatavilla laitteelta.

Sovelluksessa tallennetaan palvelimelta noudetut tehtävät ja tehtäviin tehdyt muutokset laitteen tietokantaan. Verkottomassa tilassa palvelin ei vastaanota muutoksia, jotka tehdään paikalliseen tietokantaan. Sovellukseen lisätään uusi tietokanta ja hallintafunktio. Tietokantaan tallennetaan verkottomassa tilassa luodut verkkopyynnöt funktion avulla. Esimerkkikoodissa 16 määritellään savePendingOperation-funktio, joka tallentaa verkototmassa tilassa luodut verkkopyynnöt tietokantaan synkronointia varten.

```
async function savePendingOperation(operation) {
  const db = await openDatabase();
  const transaction = db.transaction("syncTasks", "readwrite");
  const store = transaction.objectStore("syncTasks");
  return new Promise(async (resolve, reject) => {
    await store.add(operation);
    transaction.onsuccess = () => resolve("Success!");
    transaction.onerror = (e) => reject(e.target.error);
  });
}
```

Esimerkkikoodi 16. Funktio kutsutaan verkkopyynnön epäonnistuessa. Funktio avaa yhteyden tietokantaan ja tallentaa verkottomassa tilassa tehdyn verkkopyynnön tietokantaan.

Sovellus välittää synkronointikutsuja, kun sovellus avataan. Service Worker-tiedostoon lisätään uusi kuuntelija sync-pyynnölle. Kuten fetch-verkkopyyntöjä kuunteleva funktio, suoritetaan sync-tapahtumassa funktio, joka tarkistaa syncTasks-tietokannan ja suorittaa tietokantaan tallennetut verkkopyynnöt. Esimerkkikoodi 17 synkronoi sovelluksessa tehdyt muutokset noutamalla tallennetut operaatiot tietokannasta. Tietokannasta noudetut operaatiot suoritetaan verkkopyyntöinä ja poistetaan tietokannasta operaation onnistuessa.

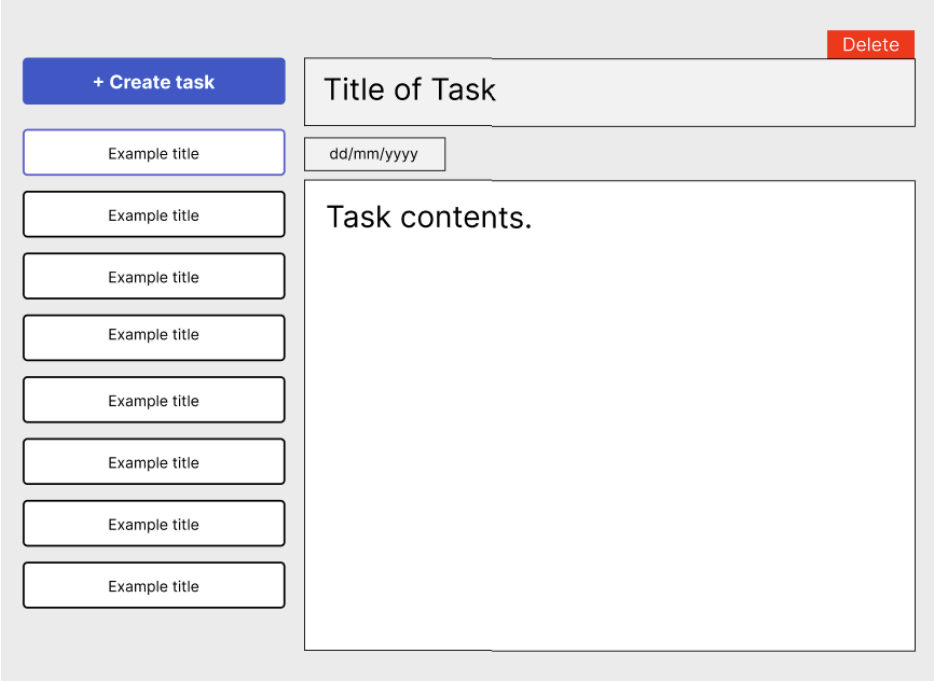
```
const syncPendingOperations = async () => {
  const db = await openDatabase();
  const transaction = db.transaction("syncTasks", "readwrite");
  const store = transaction.objectStore("syncTasks");

  new Promise((resolve, reject) => {
    const req = store.getAll();
    const keys = store.getAllKeys();
    req.onsuccess = async (event) => {
      const operation = req.result;
      await Promise.all(
        operation.map(async (operation, i) => {
          try {
            const response = await fetch(operation.url, {
              method: operation.method,
              headers: { "Content-Type": "application/json" },
              body: JSON.stringify(operation.body),
            });

            if (response.ok) {
              const deleteTransaction = db.transaction(
                "syncTasks",
                "readwrite"
              );
              const deleteStore = deleteTransaction
              .objectStore("syncTasks");
              await deleteStore.delete(keys.result[i]);
            }
          } catch (error) {
            console.log("something went wrong: ", error);
          }
        })
      );
      resolve(event.target.result);
    };
    req.onerror = (event) => {
      reject(event.target.error);
    };
  });
};
```

Esimerkkikoodi 17. Koodi synkronoi sovelluksessa tehdyt muutokset noutamalla tallennetut operaatiot tietokannasta. Tietokannasta noudetut operaatiot suoritetaan verkkopyyntöinä ja poistetaan tietokannasta operaation onnistuessa.

Insinöörityössä on käsitelty sovelluksen kehittämiseen liittyvää kommunikointia jaetun palvelimen kanssa sekä sovelluksen tietokantojen toimintaa. Seuraavaksi käsitellään sovelluksen käyttöliittymän kehitystä.

Sovelluksen käyttöliittymä suunnitellaan Figma-sovelluksella. Käyttöliittymä jaetaan kahteen osaan, vasemmalla puolella käyttöliittymää käyttäjä voi tarkastella tehtäviä listassa, lisätä uusia tehtäviä ja valita aktiivisen tehtävän.

Käyttöliittymän oikealla puolella on lomake, joka sisältää aktiivisen tehtävän tiedot. Lomakkeen avulla voidaan tarkastella ja muokata aktiivisen tehtävän otsikkoa, sisältöä ja päivämäärää. Sovelluksen käyttöliittymä suunnitellaan yksinkertaiseksi ja intuitiiviseksi käyttäjää ajatellen.



Kuva 3. Figma-sovelluksessa suunniteltu käyttöliittymä.

Sovelluksessa käytetään Next.js-viitekehystä, mikä tarkoittaa, että käyttöliittymä kehitetään hyödyntäen React-komponentteja. React-komponentit ovat itsenäisiä ja uudelleenkäytettäviä funktioita, jotka palauttavat HTML-elementtejä. Esimerkkikoodi 18 esittelee kaksi React-komponenttia, joista ensimmäinen palauttaa kiinteän otsikon ja toinen dynaamisen otsikon, joka määräytyy komponentin headerText-muuttujasta.

```
Const reactElement = () => {
     Return(
          <h1> Return this heading </h1>
     )
}

Const reactElementWithVariable = ({headerText}) =>{
     Return(
          <h1> {headerText} </h1>
     )
}
```

Esimerkkikoodi 18. Kaksi React-komponenttia, joista ensimmäinen palauttaa kiinteän otsikon ja toinen dynaamisella otsikolla, joka määritetään komponentille annetusta headerText-muuttujasta

Sovellukseen lisätään esimerkkikoodi 18:n tapaan React-komponentti, joka luo listan tietokannasta noudetuista tehtävistä.

```
const Tasks = ({tasks, currentTask}) => {
  return (
    <div className=" max-h-[550px] overflow-scroll noscroll py-2
    pt-0 mt-3">
      {tasks.map((test, i) => (
        <div
          key={i}
          className="w-full border p-2 my-1 rounded-sm
          flex items-center text-center justify-center"
          onClick={() => currentTask(test)}
        >
          <p>{test.title}</p>
        </div>
      ))}
    </div>
  );
};
```

Esimerkkikoodi 19. React-komponentti, joka listaa tehtävät ja mahdollistaa yksittäisen tehtävän valitsemisen tapahtumankäsittelijällä.

!Sovellus noutaa tehtävät tietokannasta ja tallentaa tehtävät useState-tilamuuttujaan. Tilamuuttuja palauttaa kaksi arvoa, josta ensimmäinen arvo sisältää muuttujan nykyisen tilan ja toinen arvo sisältää funktion, joka päivittää muuttujan arvon. Muuttujan arvoa päivittämällä käyttöliittymän elementit, jotka hyödyntävät muuttujan arvoa, päivittyvät automaattisesti ilman sivun uudelleen lataamista.

```
const [tasks, setTasks] = useState([]);
const [activeTask, setActiveTask] = useState([]);

console.log(activeTask) //returns currently active task
setActiveTask(task) //Sets a new active task
```

Esimerkkikoodi 20. Koodissa määritetään kaksi useState-tilamuuttujaa. Tilamuuttujissa tasks ja activeTasks ovat muuttujan arvo. Muuttujan arvoa voidaan muokata setTasks- ja setActiveTask-funktioilla.

Sovellus tallentaa tehtävät automaattisesti, kun lomakkeen tietoja muokataan. Sovellukseen lisätään funktiot, jotka tallentavat, poistavat ja päivittävät tehtävän tiedot tietokantaan. Esimerkkikoodi 21 sisältää funktion, joka lähettää tehtävään tehdyt muutokset verkkopyyntönä palvelimelle.

```
const updateTask = async (id:number) => {
  const parseDate = new Date(date).getTime();
  const dateNow = Date.now();
  const task = {
    title: title,
    content: content,
    due_date: parseDate,
    last_modified: dateNow,
  };
  await fetch(`http://localhost:8080/tasks/${id}`, {
    method: "PUT",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(task),
  })
    .then((res) => res.json())
    .catch((err) => console.log("Err: ", err));
};
```

Esimerkkikoodi 21. Asynkroninen funktio, joka lähettää päivitetyn tehtävän palvelimelle verkkopyyntönä.

PWA-sovelluksissa, kuten verkkosovelluksissa noudetaan käyttöliittymän resurssit verkosta, kun sovellus avataan. Käyttöliittymän toiminta verkottomassa tilassa vaatii, että HTML, JavaScript, CSS, fontit ja kuvat tallennetaan välimuistiin verkotonta tilaa varten. Välimuisti käyttöliittymän tiedostoille määritetään SW-tiedostossa InstallEvent-funktiossa. Esimerkkikoodi 22 määrittää sovelluksen installEevent-funktiossa thesis-välimuistin, ja lisää sovelluksen käyttöliittymän toimintaan vaadittaat tiedostot välimuistiin.

```
const installEvent = () => {
  self.addEventListener("install", (e) => {
    e.waitUntil(caches.open("thesis").then((cache) =>
      cache.addAll(["/", "/page.tsx", "/manifest.json", "/components",
"/layout.tsx", "/public"])));
  });
};
```

Esimerkkikoodi 22. Työskentelijän asennusfunktio, joka tallentaa määritellyt resurssit välimuistiin caches-rajapinnan avulla.

Työskentelijän asennusfunktiossa oleva caches-rajapinta mahdollistaa sovelluksen käytön ja käyttöliittymän toiminnan verkottomassa tilassa.

4.3 Electron-kehitys

Electron-sovellus kehitetään tarjoamaan samat keskeiset toiminnallisuudet ja ominaisuudet kuten PWA-sovellus. Sovellus tallentaa tehtäviä tietokantaan ja synkronoi tehtäviin tehdyt muutokset palvelimen ja sovelluksen tietokannan välillä. Electron-sovelluksessa hyödynnetään Node-ajoympäristöä Sqlite3-tietokannan isännöintiin ja sovelluksen käyttöliittymä kehitetään ilman viitekehyksiä käyttäen HTML- , CSS- ja JavaScript-tiedostoja.

Electron-sovellus pohjustetaan luomalla kansio, jossa alustetaan Node-ajoympäristö. Electron-kehitykseen asennetaan tarvittavat riippuvuudet komennolla ”npm install electron—save-dev” ja sovelluksen tietokanta sqlite3 ja sen riippuvuudet. Lisätään sovellukseen Main- , db- , index-, renderer- ja preload-tiedostot. Main-tiedosto on vastuussa main-prosessista, sovelluksen elinkaaresta ja kommunikaatiosta sovelluksen tietokantaan sekä laitteen käyttöjärjestelmän rajapintakutsuihin. Index.html tiedosto sisältää sovelluksen käyttöliittymän asettelun ja renderer-tiedosto hallitsee renderer-prosessia, käyttöliittymän tiedonkäsittelyä ja tapahtumia.

Sovelluksessa hyödynnetään Sqlite3-tietokantaa, joka on käytössä myös sovellusten jaetussa palvelimessa. Tämä tarkoittaa, että tietokannan skeema ja tietokantaa hallitsevat funktiot kopioidaan palvelimelta sovellukselle ilman muutoksia. Sovellukseen lisätään kaksi uutta funktiota tietokannan hallintaan.

Ensimmäinen funktio lisää ja päivittää palvelimelta noudetut tehtävät tietokantaan, jos sovelluksen tietokanta on tyhjä tai tehtävät tietokannassa ovat jäljessä palvelimeen verrattuna. Toinen funktio on vastuussa palvelimen ja sovelluksen tietokantojen poistetuksi merkittyjen tehtävien poistosta. Esimerkkikoodi 23 funktion avulla poistetaan kaikki tehtävät, jotka on merkitty poistetuksi tietokannasta.

```
const deleteFlaggedTasksLocal = () => {
     const sqlDelete = `DELETE FROM tasks WHERE deleted = ?`;
     db.run(sqlDelete, [1], function(err) {
          if (err) {
               console.log("Error deleting tasks: ", err.message);
          }
          console.log(`Deleted ${this.changes} rows`
     });
};
```

Esimerkkikoodi 23. Koodi tarkastaa tietokannan poistettavaksi merkityistä tehtävistä ja poistaa merkityt tehtävät paikallisesta tietokannasta.

Main-tiedosto vastaa sovellusikkunan luomisesta ja elinkaaresta, kun sovellus avataan. Tiedostossa määritetään funktiot, jotka hallitsevat kutsuja tietokantaan sovelluksen käyttöliittymästä IPC-kutsujen välityksellä. Esimerkkikoodissa 24:n funktion avulla luodaan uusi Electron-sovellusikkuna. Ikkuna asetetaan 800x600 pikselin korkeuteen ja leveyteen. Funktiossa alustetaan myös preload-tiedosto ja lopuksi asetetaan index.html-tiedosto sovelluksen käyttöliittymäksi.

```
const createWindow = () => {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, "preload.js"),
    },
  });
  win.loadFile("index.html");
};
app.whenReady().then(() => {
  createWindow();
}
```

Esimerkkikoodi 24. Koodi luo uuden 800 ja 600 pikseliä leveän ja korkean ikkunan, joka sisältää index.html-tiedoston sisällön.

IPC-kutsut lisätään sovellukseen ennen createWindow-funktiota. Esimerkkikoodi 25 rekisteröi Main-tiedostossa IPC-kutsut sync ja getAll ennen createWindow-funktion kutsua.

```
app.whenReady().then(() => {
 ipcMain.handle("sync", syncData);
 ipcMain.handle("getAll", getAllData);
 createWindow();
}
```

Esimerkkikoodi 25. Esimerkkikoodissa lisätään IPC-kutsut sync ja getAll, jotka suorittavat funktiot syncData ja getAllData.

Electron-sovellus toimii verkottomassa tilassa ilman tiedostojen tallentamista välimuistiin. Sovellukseen lisätään toiminnallisuus, joka tarkistaa palvelimen saatavuuden verkosta ja synkronoi tiedot paikallisesta tietokannasta palvelimen tietokantaan. Kun sovellus on synkronoitu, paikallisesta tietokannasta poistetaan kaikki poistettavaksi merkityt tehtävät ja lähetetään verkkopyyntö palvelimelle, joka poistaa merkityt tehtävät palvelimen tietokannasta. Esimerkkikodi 26 tarkistaa palvelimen saatavuuden verkosta. Jos palvelin on saatavilla, suoritetaan funktio palvelimen tietokannan ja sovelluksen tietokannan synkronointiin. Jos synkronointi onnistuu, suoritetaan funktio, joka poistaa paikallisesta tietokannasta poistettavaksi merkityt tehtävät ja lähettää pyynnön poistaa vastaavat tehtävät palvelimelta.

```
(async () => {
    const networkConnection = await isReachable("localhost:8080");
    if (networkConnection) {
      const synced = await db.syncDatabase();

      if (synced) {
        console.log("Databases in sync ::: Deleting flagged records");
        const deleted = await db.deleteFlagged();
        console.log(deleted);
        console.log("Local database sync :::");
        await db
          .syncLocal()
          .then((res) => console.log(res))
          .catch((err) => console.log(err));
      }
    }
  })();
```

Esimerkkikoodi 26. Tarkistaa palvelimen saatavuuden verkosta. Jos palvelin on saatavilla, suoritetaan funktio palvelimen tietokannan ja sovelluksen tietokannan synkronointiin. Jos synkronointi onnistuu, suoritetaan funktio, joka poistaa paikallisesta tietokannasta poistettavaksi merkityt tehtävät ja lähettää pyynnön palvelimelle tekemään samoin.

Main-tiedostossa määritetyt IPC-kutsut lisätään Preload-tiedostoon. Preload-tiedostossa luodaan rajapinta, joka on avoin sovelluksen käyttöliittymälle. Rajapinnan avulla kommunikointi Main- ja Renderer-prosessien välillä tapahtuu tietoturvallisesti. Esimerkkikoodi 27 luo uuden IPC-rajapinnan electronAPI, joka on saatavilla sovelluksen Renderer-prosessissa. Rajapinta mahdollistaa Main-prosessissa olevien funktioiden suorittamisen Renderer-prosessista.

```
contextBridge.exposeInMainWorld("electronAPI", {
  getAll: () => ipcRenderer.invoke("getAll"),
  modifyData: async (obj) => ipcRenderer.invoke("modifyData", obj),
  createTask: () => ipcRenderer.invoke("createTask"),
  deleteTask: async (id) => ipcRenderer.invoke("deleteTask", id),
});
```

Esimerkkikoodi 27. Luo uuden IPC-rajapinnan electronAPI, joka mahdollistaa Main-prosessissa olevien funktioiden suorittamisen Renderer-prosessista Preload-tiedoston välityksellä.

Sovelluksen käyttöliittymä kehitetään käyttäen samaa pohjaa ja suunnitelmaa kuin PWA-sovelluksessa. Electron-versiossa ei käytetä verkkokehityksessä käytettäviä viitekehyksiä, vaan käyttöliittymä luodaan HTML-, CSS- ja JavaScript-

tiedostoilla. Lisätään index.html-tiedostoon tarvittavat elementit ja niihin sopivat tunnisteet. Käyttöliittymän lista tehtävistä toteutetaan käyttäen DOM:ia Renderer.js-tiedostossa. Tehtävien hallintaan käytettävä lomake lisätään index.html-tiedostossa ja lomakkeen kenttien arvot ovat saatavilla DOM:n avulla.

Renderer-tiedostosta noudetaan kaikki index-tiedostossa olevat elementit. Renderer-tiedostoon lisätään funktio, joka noutaa tehtävät IPC-kutsun avulla ja luo jokaisesta tehtävästä oman elementin listaan. Käyttäjä valitsee tehtävän painamalla tehtävä-elementtiä. Elementtiä painamalla asetetaan kyseisen tehtävän otsikko, sisältö ja päivämäärä arvoksi lomakkeen kentille. Tiedostoon lisätään funktiot, jotka hallitsevat tehtävien päivittämisen, poistamisen ja uusien tehtävien lisäyksen. Lopuksi sovelluksen sisältö ja asettelu määritetään CSS-tiedostolla Figma-mallin mukaiseksi.

5 Yhteenveto

Insinöörityössä käsitellään työpöytäsovellusten kehitystä progressiivisilla verkkosovelluksilla ja Electronilla vaihtoehtoisena ratkaisuna alustapohjaisiin natiivisovelluksiin. Työssä kehitettävät sovellukset kehitettiin noudattaen parhaita käytäntöjä ja niiden tuli toimia verkottomassa tilassa sekä synkronoida paikalliset tiedot palvelimen kanssa, jossa sijaitsi jaettu tietokanta.

Sovellusten kehitysprosessin aikana voitiin havaita useita eroja teknologioiden välillä. PWA-sovellukset tarjosivat kevyen ja tutun ympäristön, jonka kehitys ja käyttöönotto oli yksinkertaista. PWA-sovellusten hyödyntämä Service Worker mahdollistaa kattavat toiminnallisuudet ja nopean suorituskyvyn hyödyntämällä laitteen välimuistia ja paikallista tallennustilaa verkottomassa tilassa. Electron-sovellukset tarjoavat kokemuksen, joka muistuttaa vielä enemmän natiivisovellusta. Electron-sovellusten tarjoamat työkalut mahdollistavat syvemmän integraation käyttöjärjestelmään verrattuna PWA-sovelluksiin antamalla sovellukselle pääsyn käyttöjärjestelmän rajapintakutsuihin. Electron-sovellukset ovat kooltaan suurempia ja vaativat enemmän resursseja verrattuna

PWA-sovelluksiin johtuen siitä, että sovellus yhdistää selainmoottorin ja node-ajoympäristön.

Sovellusten kehityksen näkökulmasta PWA-sovellusten kehitys mallintaa tavanomaista verkkokehitystä ja vaatii vähemmän alustavaa konfigurointia. Electron sovelluksen kehityksessä on ymmärrettävä verkkokehityksen työtavat sekä Electron-kehityksen työkalut ja tavat. Tärkein asia ymmärtää Electron-kehityksessä on Main- ja Renderer-prosessien välinen kommunikointi.

Electron- sekä PWA-sovellukset kykenevät tarjoamaan kokemuksen, joka on lähellä natiivisovellusta. PWA-sovellusten vahvuudet ovat sovellusten kevyet resurssivaatimukset, kehityksen yksinkertaisuus ja sovellusten jakelu verkon välityksellä. Electron-sovellukset puolestaan soveltuvat paremmin tilanteisiin, jossa sovellukselta vaaditaan laajempia työpöytäominaisuuksia ja integraatiota asennetun järjestelmän kanssa.

Valinta PWA- ja Electron-sovellusten välillä on hyvä perustaa sovelluksen vaatimuksiin. Molemmat osoittavat, että verkkotekniikat ovat kehittyneet tarpeeksi, että niillä voidaan toteuttaa sovelluksia, jotka vastaavat natiivisovelluksia ilman merkittäviä kompromisseja suorituskyvyssä tai käytettävyydessä. PWA- ja Electron-sovellusten jatkuva kehitys takaa sen, että ne tulevat olemaan tulevaisuudessa lähempänä natiivisovelluksia tai korvaavat perinteiset natiivisovellukset monissa käyttötarkoituksissa.

Lähteet

1 PWAs: Past, Present and Future. 2024. Verkkoaineisto. Chromeos.dev. <https://chromeos.dev/en/posts/pwas-past-present-future>. Luettu 9.10.2024.

2 Electron documentation. Verkkoaineisto. Electronjs.org. <https://www.electronjs.org/docs/latest/>. Luettu 9.10.2024.

3 Introduction to Node.js. Verkkoaineisto. Node.js. <https://nodejs.org/en/learn/getting-started/introduction-to-nodejs>. Luettu 10.10.2024.

4 Progressive Web Apps. 2024. Verkkoaineisto. Web.dev. <https://web.dev/learn/pwa/progressive-web-apps>. Luettu 12.10.2024.

5 Getting started. 2021. Verkkoaineisto. Web.dev. <https://web.dev/learn/pwa/getting-started>. Luettu 12.10.2024.

6 HTML. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/HTML>. Luettu 12.10.2024.

7 CSS. 2019. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/CSS>. Luettu 12.10.2024.

8 Introduction to the DOM. 2019. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction>. Luettu 13.10.2024.

9 Service Worker API. 2019. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API>. Luettu 13.10.2024.

10 Serving. 2022. Verkkoaineisto. Web.dev. <https://web.dev/learn/pwa/serving>. Luettu 15.10.2024.

11 Service Workers. Verkkoaineisto. Web.dev. <https://web.dev/learn/pwa/service-workers>. Luettu 15.10.2024.

12 Indexed API. 2019. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API>. Luettu 18.10.2024.

13 IndexeDB. Verkkoaineisto. Web.dev. <https://web.dev/articles/indexeddb>. Luettu 18.10.2024.

14 Caching. 2021. Verkkoaineisto. Web.dev. <https://web.dev/learn/pwa/caching>. Luettu 18.10.2024.

15 Web App Manifest. Verkkoaineisto. MDB Web Docs. <https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Manifest>. Luettu 19.10.2024.

16 Web App Manifest. 2021. Verkkoaineisto. Web.dev. <https://web.dev/learn/pwa/web-app-manifest>. Luettu 20.10.2024.

17 IPC. Verkkoaineisto. Electronjs.org. <https://www.electronjs.org/docs/latest/tutorial/ipc>. Luettu 2.11.2024.

18 Process Model. Verkkoaineisto. Electronjs.org <https://www.electronjs.org/docs/latest/tutorial/process-model>. Luettu 4.11.2024.

19 Express. 2017. Verkkoaineisto. Expressjs.com. <https://expressjs.com/>. Luettu 6.11.2024.

20 CORS. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS>. Luettu 6.11.2024.

21 SQLite. Verkkoaineisto. SQLite. <https://sqlite.org/>. Luettu 10.11.2024.

22 HTTP request methods. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods>. Luettu 11.11.2024