SAVONIA

OPINNÄYTETYÖ - AMMATTIKORKEAKOULUTUTKINTO
TEKNIIKAN JA LIIKENTEEN ALA

LAITEHALLINTAJÄRJESTELMÄN KEHITYSTYÖ JA RAPORTOINTI

TEKIJÄ/T: Viivi Leppänen

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Tutkinto-ohjelma Tietotekniikan tutkinto-ohjelma	
Työn tekijä(t) Viivi Leppänen	
Työn nimi Laitehallintajärjestelmän kehitystyö ja raportointi	
Päiväys 11.3.2021	Sivumäärä/Liitteet
Toimeksiantaja/Yhteistyökumppani(t) Säteilyturvakeskus STUK	

Tiivistelmä

Opinnäytetyön tarkoituksena oli toteuttaa Säteilyturvakeskuksen STUK KET-yksikölle laitehallintajärjestelmä, jonka avulla pidetään kirjaa yksikön käyttölaitteista. Laitehallintajärjestelmän avulla on tarkoitus tallentaa yksittäistä laitekohtaista dataa sekä laitteiden ominaisuuksista, että niiden kuntotarkastuksista. Lisäksi laitehallintajärjestelmässä tuli olla erillinen kokonaisuus laitteiden lainaamista varten ja varastotietojen ylläpitämistä varten. Lisäksi opinnäytetyön tarkoituksena oli saada kattavampaa ymmärrystä full-stack ohjelmoinnista ja React-ohjelmointikielestä sekä kehittää aiemmin karttunutta taitoa. Työmenetelmistä päätettiin yhdessä toimeksiantajan kanssa ja työssä käytettiin tarpeiden mukaan jo olemassa olevia ja ylläpidettyjä kirjastoja.

Järjestelmä tuli kehittää MERN-stack menetelmää hyödyntäen. Tähän kuului käyttäjälle näkyvä käyttöliittymäpuoli, keskitason datankeräyspuoli sekä tietokantapuoli datan tallentamista varten. Data varastoitiin pilvessä sijaitsevaan noSQL-tietokantaan.

Opinnäytetyön lopputuloksena valmistui laitehallintarekisteri paikalliseen käyttöön. Järjestelmää hallitaan käyttöliittymän kautta selaimen avulla ja se tallentaa monitasoista dataa pilvessä sijaitsevaan klusteriin.

Avainsanat

MERN, web-ohjelmointi, full-stack, ohjelmistokehitys, JavaScript, NoSQL, MongoDB, React.js

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport	
Degree Programme Degree Programme in Information Technology	
Author(s) Viivi Leppänen	
Title of Thesis Device management system	
Date	Pages/Appendices
Client Organisation /Partners Finnish Radiation and Nuclear Safety Authority STUK	

Abstract

The purpose of this thesis was to create a device management system for Finnish Radiation and Nuclear Safety Authority's KET-unit. The device management system is used for keeping record of the unit's machines and devices. The main goal of the device management system is to store individual device-specific data on both the features of the devices and their condition inspection data. In addition, the device management system is required to have a separate loan-side for lending the devices and to keep the storage data updated. Also, the purpose of the thesis is to provide more comprehensive understanding of full-stack programming with React and to improve previously learned skills in these matters.

Working methods were considered and decided together with the client and already existing and maintained libraries were used when needed. The system was supposed to be developed using the full-stack method. This method includes a user interface visible to the user, a mid-level data collection side, and a database side for storing the applications data. The data of this system is stored in NoSQL database located in the cloud cluster.

The outcome of the thesis was an application for the device management system. The application is controlled via user interface on a web-browser, and it stores multi-level data to the cloud cluster.

Keywords

MERN, web development, full-stack, JavaScript, NoSQL, MongoDB, React.js

SISÄLTÖ

KUVALUETTELO

1 JOHDANTO

Tämän opinnäytetyön tarkoituksena on tuottaa Säteilyturvakeskuksen Kenttä- ja tilannekuvajärjestelmät-yksikölle (KET) laitteiden varastointia helpottavan laitteiden hallintajärjestelmä. Laitehallintajärjestelmää käytetään osaston käyttöobjektien varastohallintaan sekä laitteista kerättävän yksityiskohtaisen datan hallintaan. Järjestelmän tulee olla päätelaitteelta paikallisesti käytettävä ohjelmisto, joka varastoi käyttöobjektien dataa sekä objekteille tehtävien kuntotarkastusten dataa. Ohjelmistossa tulee olla tämän lisäksi myös lainauspuoli varastonhallintaa varten, josta käyttäjät voivat lainata ja palauttaa varastostaan lainaamia tuotteita.

Opinnäytetyö toteutetaan full-stack menetelmällä, johon kuuluu sekä käyttöliittymän, että käyttöliittymän takana toimivan palvelinosuuden kehittäminen. Opinnäytetyön tavoitteena on toteuttaa moniosainen järjestelmä, jossa hyödynnetään NoSQL- tietokantamallia dokumenttien muodossa. Järjestelmään tallennettavat tiedot ovat monitasoisia, joiden tallentamiseen dokumenttimalli on sopivin. Järjestelmän on määrä toimia paikallisesti päätelaitteelta, kun taas sen tietokantaosuus nostetaan pilveen verkkoon.

Opinnäytetyössä käytetyt tekniikat valitaan toimeksiantajan kanssa yhteistyössä ottaen huomioon projektin vaatimukset. Tähän sopivimpana pidetään MERN-stack:ia, joka koostuu neljästä eri full-stack kehitystyökalusta, mutta ohjelmointikielenä ennen kaikkea voidaan pitää käytetyn JavaScript-ohjelmointikieltä. MERN:ssä käytetään tyypillisesti MongoDB:tä, jonka katsotaan olevan monitasoisen datan dokumentointiin sopivin ja joustavin ratkaisu.

Opinnäytetyössä raportoidaan järjestelmän kehitysprosessin vaiheet ja kulku, sekä kehityksessä käytetyt tekniikat. Lisäksi tehdään raportti kehitystyössä tehdyistä havainnoista aiheeseen liittyen. Opinnäytetyössä myös tarkastellaan lopuksi syntynyttä laitehallintajärjestelmää ja jatkokehitysratkaisuja.

1.1 Toimeksiantaja

Säteilyturvakeskus eli STUK, on Suomen säteilyvalvonnasta ja ydinturvallisuudesta vastaava viranomainen. STUK:n toiminnan tavoite on pitää suomalaisten säteilyaltistus niin pienenä sekä turvallisuus niin hyvänä kuin käytännöllisin toimenpitein on mahdollista. STUK:n valvonnan perusta on säteily- ja ydinturvallisuutta koskeva lainsäädäntö, turvallisuusmääräykset ja ohjeet. STUK:n toimintaan kuuluu myös suomalaisen säteilyturvallisuuden kehittäminen yhteistyössä ulkomaisten kumppaneiden kanssa. Tärkeimpiä kotimaisia kumppaneita ovat esimerkiksi eri ministeriöt. Kansainvälisessä yhteistyössä puolestaan STUK pyrkii vaikuttamaan ja kehittämään ohjeistoja niin, että turvallisuus paranee myös maailmanlaajuisesti.

STUK on perustettu vuonna 1958, jolloin se oli vielä pieni lääkintöhallituksen alainen Säteilyfysiikan laitos, jonka tehtävä oli tarkastaa sairaaloissa käytettävät säteilylaitteet. Vuosien kuluessa ja tiedon lisääntyessä sekä säteilyn- ja radioaktiivisten aineiden käytön yleistyessä STUK:n tehtävät moninaistuivat. Nykyisin STUK on täyden palvelun asiantuntijatalo. Sen päätoimipiste sijaistee Roihupellossa Helsingissä, osoitteessa Laippatie 4 ja se työllistää tällä hetkellä 340 henkilöä.

STUK:n toiminta perustuu lakiin Säteilyturvakeskuksesta. Laki määrittää STUK:n tehtävän ja aseman. STUK:n johdossa on Petteri Tiippana, valtioneuvoston nimittämä pääjohtaja.

1.2 Lyhenteet ja määritelmät

Backend	Palvelinpuolen käytännöt
Frontend	Käyttöliittymätoteutukset
Framework	Ohjelmistokehys
JavaScript	Web-sovellusten ohjelmointikieli
MERN-stack	MERN-pino joka koostuu useasta eri ohjelmakokonaisuudesta
React.js	Avoimen lähdekoodin JavaScript-kirjasto
Node.js	Ajoympäristö JavaScriptille
MongoDB	NoSQL-tietokantaratkaisu
NoSQL	Relaatiomallista poikkeava tietokantaratkaisu
SQL	Relaatiokantojen kyselykieli
Express.js	Palvelinpuolen ohjelmistokehys Node.js:lle
CRUD	Luonti, lukeminen, päivitys ja poisto-toiminnoista käytetty lyhenne
CSS	Tyylisivu verkkosivuille
Cluster	Ryhmä palvelimia ja resursseja jotka toimivat yhtenä kokonaisuutena
JSON	JavaScriptistä riippumaton tiedostomuoto
Renderöinti	Koodin visualisoiminen web-sovelluksessa käyttäjälle

1.3 Työn tausta

Säteilyturvakeskuksen KET-yksiköllä on useita yksikön toiminnallisissa tehtävissä käytettäviä laitteita ja laitteisiin liittyviä komponentteja. Laitteita varastoidaan joskus pidempiäkin aikoja, mutta niitä myös kootaan laitekokonaisuuksiksi eri käyttötarkoitusten mukaan. Laitteita käyttää useampi työntekijä, jolloin on tärkeää laitteiden kunnossapidon kannalta kerätä niistä laitekohtaista dataa yhteen järjestelmään. Laitteiden kuntoa myös tarkastetaan määräajoin yksikön työntekijöiden toimesta. Tarkastusdatan tulisi linkittyä kyseisen laitteen yksilölliseen dataan, jotta laitteen tietoja saataisiin monitoroitua ja tiedot laitteen ominaisuuksista ja laitteen fyysisestä kunnosta sekä korjaus- tai päivitystarpeesta olisivat keskitetysti yhden järjestelmän takana. Laitteille suoritetaan tarkastuksia esimerkiksi kuukausittain tai vuosittain, jolloin erilaisista laitteista saatavan tarkastusdatan tulisi olla ajankohtaista silloin, kun laitteen tietoja tarkastellaan järjestelmän avulla. Lisäksi laitteiden ja laitekokonaisuuksien usean käyttäjän johdosta yksiköllä on myös tarvetta laitteiden lainausjärjestelmään, jonka avulla tulisi monitoroida esimerkiksi laitteen sijaintia, onko laite käytössä vai varastossa tai esimerkiksi tarkasteluhetkenä osana laitekokonaisuutta.

1.4 Työn tavoite

Työssä tavoitellaan laitehallintarekisterin kehitystä ja toteutusta. Järjestelmään on tarkoitus toteuttaa useita eri toiminnallisia puolia:

- laitteiden lisäämistä ja tarkastelua varten
- tarkastusten luontia ja laitteille tarkastusten tekemistä varten
- tarkastusten selaamista ja moderointia varten
- laitteiden lainausta ja lainauksien selaamista varten
- sekä niin kutsuttuja järjestelmävalvojan työkaluja, joita tavallisen käyttäjän ei tarvitse päästä käyttämään

Näitä puolia on tarkoitus kehittää limittäin projektin edetessä, sillä tekniikoiden koettiin olevan samankaltaisia ja täten uudelleenkäytettävissä. Työssä tavoitellaan ulkonäöltään yksinkertaista päätelaiteohjelmaa, jota käytetään paikallisesti. Tavoitteissa pyritään ottamaan huomioon rajallinen aika.

1.5 Työn rakenne

Opinnäytetyö koostuu projekti- ja raportointiosuudesta. Työn projektiosuus rakentuu sovelluksen ja sen komponenttien sekä käytettävien kirjastojen ja työkalujen suunnittelusta sekä toteutuksesta. Opinnäytetyön raportointiosuus koostuu työvaiheiden raportoinnista ja kehitykseen liittyvistä havainnoista.

2 TYÖKALUT JA TEKNIIKAT

2.1 TYÖKALUT

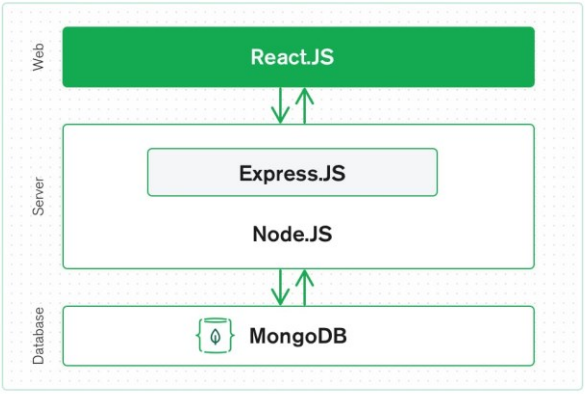
2.1.1 Visual Studio Code

Visual Studio Code on Microsoftin kehittämä ilmainen avoimen lähdekoodin koodieditori. Se on saatavilla yleisimmille käyttöjärjestelmille eli Windowsille, macOSille ja Linuxille. Editorin käytettävyyden keveydestä huolimatta, sen mukana tulevat käytännölliset ominaisuudet ovat tehneet siitä suositun ja laajasti käytetyn sekä tuetun. (Mustafeez) Opinnäytetyötä varten valittiin VS Code sen kevytkäyttöisyyden ja helppokäyttöisyyden takia.

2.1.2 MERN-stack

Kuva 1 - (blog.hyperiondev.com)

MERN-stack on kokoelma teknologioita, joita käytetään full-stack web-sovelluksen luomiseen. Se on JavaScript-kieleen pohjautuva kokoelma, joka on suunniteltu tekemään web-sovellusten kehitysprosessista joustavampaa ja ketterämpää. MERN:n kaltaisia teknologioita käytetään single-page eli SPA sovellusten luomiseen. Single-page applikaatiossa sovellus ja käyttäjä ovat dynaamisessa vuorovaikutuksessa, sovelluksen renderöidessä käyttäjän näkemälle sivulle vain sen vaatiman datan. (Yeshwanthini, 2021) MERN on avoimen lähdekoodin ohjelmistokehys, joka tarjoaa sovelluskehittäjälle mahdollisuuden käyttää koko sovelluskehityskomponenttien skaalaa front-endista back-endiin. Sovelluskehittäjälle se tarjoaa mahdollisuuden hallita sekä sovelluksen takaosaan sijoittuvia algoritmisia sekä loogisia puolia, että sen etuosaan kuuluvia käyttäjäkokemukseen perustuvia ominaisuuksia. (Swonigatechnology, 2021) MERN koostuu neljästä erillisestä osiosta, joita ovat MongoDB, Express.js, React.js ja Node.js. Express ja Node ovat teknologiakokoelman keskiosa, kun taas React on etupää ja MongoDB takaosa. Express on serveripuolen web ohjelmistokehys ja Node puolestaan on suosittu sekä käytännöllinen Javascript-serverialusta. React:ia käytetään käyttäjälle näkyvän puolen kehityksessä ja MongoDB varastoi web-sovelluksen datan. Seuraavissa kappaleissa käsitellään yksityiskohtaisemmin kaikkien neljän ominaisuuksia ja toimintatapoja.



Kuva 2 – MERN-struktuuri (www.mongodb.com)

2.1.2.1 MongoDB

MongoDB on avoimen lähdekoodin tietokantahallintajärjestelmä, jossa perinteisen relaatiomallin rivien ja taulukoiden sijaan useita erilaisia datamuotoja käsitellään ja tallennetaan dokumenttipohjaisen ratkaisun kautta. Toimiakseen joustavasti MongoDB ei vaadi relaatiomallin mukaista hallintajärjestelmää, sillä sen joustava datavarastointikyky saavutetaan NoSQL-tekniikan avulla, jota käsitellään tässä opinnäytetyössä myöhemmin luvussa NoSQL. Näiden tekniikoiden avulla MongoDB:n on kerrottu yksinkertaistavan tietokantojen hallintaa ohjelmistokehittäjien kannalta, mutta myös luovan skaalautuvan ympäristön cross-platform sovelluksille ja palveluille. (IBM Cloud Education, 2020)

MongoDB on kaikista tietokantamoottoreista viidenneksi suosituin toukokuussa 2021. Mainittakoon, että NoSQL malleista se on suosituin, sillä sitä edellä listalla on vain perinteisiä relaatiomallisia tietokantoja. (db-engines.com, 2021) MongoDB:n suosiolle syitä ovat sen helppokäyttöisyys ja edullisuus. MongoDB:tä voi käyttää sekä paikallisesti, että pilvessä sijaitsevan klusterin kautta. Sen käytössä ei vaadita pakollisia lisenssi- tai versionvaihtomaksuja. Relaatiomalleihin verrattuna NoSQL-kannat skaalautuvat paremmin, sillä niihin on mahdollista tallentaa strukturoimatonta dataa. NoSQL-kannat eivät myöskään edellytä rakenteen etukäteismäärittelyä. Relaatiokantojen ja MongoDBn eroavaisuuksia käsitellään tämän opinnäytetyön luvussa 2.2.2.1 NoSQL vs. SQL, jossa käsitellään myös kyseisen tietokantahallintatyökalun valintaa tätä projektia suunniteltaessa.

MongoDB:ssä data tallennetaan niin kutsutussa BSON muodossa, joka tarkoittaa binääristä muotoa JSON-formaatista. Dataa taltioidessa taltioidaan ensisijaisesti avain-arvo (key-value) pareja, joita MongoDB infrastruktuurissa kutsutaan nimellä fields eli kentät. Yksittäiset kentät tallennetaan dokumenttiin (document), joka relaatiokantamallissa vastaa riviä. Dokumentit puolestaan varastoidaan kokoelmiin (collection), jotka puolestaan vastaavat relaatiomallissa tauluja. Relaatiomallissa käytettyjen JOIN-lauseiden sijaan MongoDBssä käytetään sulautettuja dokumentteja (embedded documents). Relaatiomallista poiketen MongoDBssä kaikki data tallennetaan yleensä yhteen kokoelmaan, mutta erotellaan joissain tapauksissa käyttäen sulautettua dokumenttimallia. (guru99.com)

2.1.2.2 Express.js

Express on ilmainen avoimen lähdekoodin web-sovelluskehys Node.JS:lle. Expressiä käytetään web-sovellusten nopeaan ja yksinkertaiseen kehittämiseen ja suunnitteluun. Se tarjoaa suuren kirjon erilaisia ominaisuuksia web- ja mobiilisovelluskehitykseen Noden rinnalla käytettynä. Express on Noden suosituin ohjelmistokehys, josta johtuen sille on aikojen saatossa kehitetty useita erilaisia kirjastoja, jotka auttavat lähes minkä tahansa web-sovelluskehitys ongelman kanssa. Näiden keskitason kirjastojen avulla on mahdollista työskennellä niin evästeiden, sessioiden, käyttäjävalidoinnin, url-parametrien ja monen muun asian parissa, joihin esimerkiksi Node.js ei ilman Express:n apua kykene. Express:n kehittäjätiimi ylläpitää myös tukea monelle ulkopuolisten sovelluskehittäjien keskitason kirjastoille, taatakseen Express:n sujuvan toimivuuden. (Mozilla)

2.1.2.3 Node.js

Node.js eli Node on serveripuolen sovellusalusta, joka on rakennettu Google Chromen V8 JavaScript moottorin päälle. Sen kehittäjä Ryan Dahl julkaisi sen vuonna 2009 ja sen viimeisin versio on v0.10.36. Se kehitettiin helppojen ja nopeiden sekä skaalautuvien web-sovellusten luomista varten. Node käyttää niin kutsuttua non-blocking I/O mallia, joka tekee siitä kevyen ja tehokkaan. (Tutorialspoint, 2021) Kyseiseen malliin palataan opinnäytetyössä luvussa non-blocking I/O**.** Node on avoimen lähdekoodin alustariippumaton JavaScript ajoympäristö serveripuolen ja web-sovellusten kehittämiseen. Seuraavana muutama Noden tärkeimmistä ominaisuuksista, joita voidaan pitää myös tärkeinä tämän opinnäytetyön tekniikan valintaan vaikuttavina ominaisuuksina.

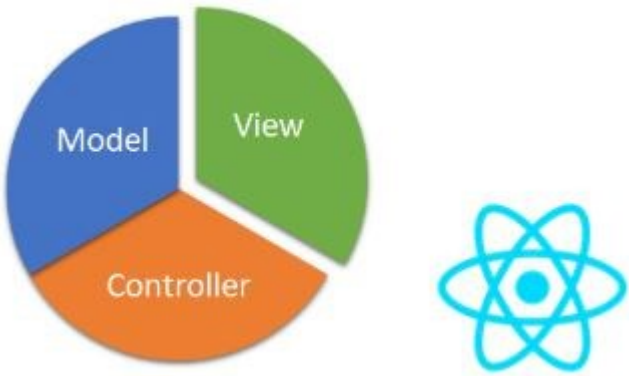
- Yksisäikeisyys
    - Node käyttää yksisäikeisyyttä tapahtumasilmukoissa (event looping). Tapahtumamekanismi auttaa palvelinta vastaamaan rajoittamattomalla tavalla, joka tekee palvelimesta skaalautuvan eli vastakohtaisen perinteisille palvelimille, jotka luovat rajoitettuja säikeitä tapahtumien käsittelyyn. Yksisäikeisyys mahdollistaa sen, että saman ohjelman on mahdollista tarjota palveluita useammalle pyynnölle.
- Non-buffering
    - Node.js-sovellukset eivät koskaan bufferoi dataa. Nodea käyttävät sovellukset tulostavat datan isommissa paloissa, jolloin bufferoinnille ei löydy tarvetta.

2.1.2.4 React.js

React.js, lyhyemmin React, on avoimen lähdekoodin JavaScript-kirjasto, jota käytetään usein käyttöliittymien luomiseen ja kehittämiseen, erityisesti yksisivuisille sovelluksille (SPA). React:ia käyttäen voidaan kehittää sekä verkko- että mobiilisovellusten näkymäkerroksia. React:n käytännöllisyyteen kuuluu uudelleenkäytettävien käyttöliittymäkomponenttien luomisen ja kehityksen mahdollistaminen. Sen on luonut Facebookille työskentelevä ohjelmistosuunnittelija Jordan Walke. React on otettu ensimmäistä kertaa käyttöön Facebookin uutissyötteessä vuonna 2011 ja sitä alettiin käyttämään myös

Instagram.com:issa vuonna 2012. React:n avulla on mahdollista luoda suuria websovelluksia, jotka pystyvät muuttamaan dataa lataamatta kuitenkaan sitä sisältävää sivua uudelleen. React:n päämäärä on siis olla mahdollisimman nopea sekä skaalautuva mutta kuitenkin yksinkertainen. Sen on määrä toimia ainoastaan sovellusten käyttöliittymissä. Käytännössä tämä vastaa MVC-mallin näkymää. (Pandit, 2021) MVC eli Model-View-Controller tarkoittaa arkkitehtistä kaavaa, joka erittelee sovelluksen kolmeen erilliseen loogiseen komponenttiin, joita ovat malli (model), näkymä (view) ja käsittelijä (controller). Jokainen näistä komponenteista on suunniteltu käsittelemään sovelluksen tiettyä kehitysaspektia. Se on yleisimmin käytetty sovelluskehitysmalli skaalautuvien ja laajennettavien sovellusten kehittämiseen. (Tutorialspoint, 2021) React:ssa tätä voidaan hyödyntää usein eri tavoin, kuten esimerkiksi käyttämällä sitä yhdessä muiden JavaScript-kirjastojen ja viitekehysten kanssa.

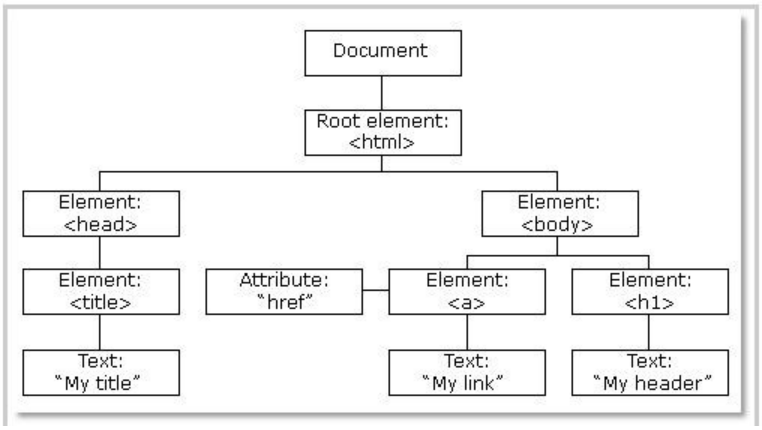
React:ia suositaan sen yksinkertaisuuden sekä suorituskyvyn takia. React:n komponenttilähtöisyys, hyvin määritelty elinkaari ja JavaScriptin yksinkertainen käyttö tekevät React:sta nopeasti ja helposti opittavan, mutta kuitenkin ammattimaisen tavan rakentaa sekä web- että mobiilisovelluksia. Tähän opinnäytetyöhön React-ohjelmointikielen valintaan vaikuttivat sen käytännönläheisyys ja skaalautuvuus opinnäytetyön projektin suunnitteluvaiheessa. React katsottiin ketterimmäksi kehityskieleksi, sekä yksinkertaisesti tehokkaimmaksi opinnäytetyön projektin kannalta. Lisäksi opinnäytetyön tekijällä ei ollut aikaisempaa kokemusta React:sta, joten myös sen opiskelu opinnäytetyön merkeissä katsottiin edulliseksi valtiksi.



Kuva 3 - MVC arkkitehtuuri

2.1.2.4.1 Virtuaalinen DOM

DOM:lla tarkoitetaan W3C:n standardoimaa dokumenttiobjektimallia (Document Object Model). Tämä tarkoittaa alustaa, joka on ohjelmointikielelle neutraali käyttöliittymä sillä se mahdollistaa ohjelmien sekä koodien dynaamisen pääsyn ja muokkausmahdollisuuden dokumentin sisältöön, tyyliin sekä struktuuriin. Yksi React:n pääominaisuuksista on sen käyttämä virtuaalinen DOM eli VDOM. Tämä tarkoittaa sitä, että React varastoi muistiinsa kevyen representaation niin sanotusta todellisesta DOM:sta. Koska muutoksia ei tehdä näkyvillä, todellisen DOM:n käsittely ja siihen muutosten tekeminen on huomattavasti hitaampaa kuin VDOM:n. Koodiin kirjoitetun objektin tilan muuttuessa, VDOM muuttaa kyseistä objektia todellisessa DOM:ssa, kaikkien objektien päivittämisen sijaan.



Kuva 4 - Websivun DOM

2.1.2.4.2 Yhdensuuntainen Data-Flow

Useissa perinteisissä JavaScript-viitekehyksissä käytetään kaksisuuntaista tiedonsiirtomallia. Usein se tarkoittaa kahden asian yhdistämistä työskentelyn helpottamiseksi, esimerkkinä vaikka joissain viitekehyksissä käytetty tekstiruudun sitominen muuttujaan. Käyttäjän muuttaessa tekstiruudun arvoa, myös muuttujan arvo päivittyy. Kuitenkin ongelmana tällaisessa tilanteessa nähdään ulkopuolisten seikkojen vaikutus muuttujan arvoon, jolloin sekä muuttujan, että tekstiruudun arvon täytyisi päivittyä. Kaksisuuntaisen tiedonsiirron ollessa yksinkertaista ja sopiessa mustavalkoisempiin käyttötarkoituksiin, toisaalla sovellusten kehittyessä yhä monimutkaisemmiksi, vaaditaan auto-binding:n (tietojen automaattisen päivittämisen) kaltaisia toimintatapoja. Tässä muuttujien React:lle kahteen kertaan toisiinsa sitomisen sijaan kaikki kulkee aina samaa tiedonsiirtoreittiä. Yksisuuntainen tiedonsiirto pitää sovelluksessa tapahtuvan työn modulaarisena ja nopeana. Tämä tarkoittaa myös kehitysvaiheessa tehtävää strukturointityötä, jossa alikomponentit sijoitetaan pääkomponentteihin siten, että sovelluskehittäjän on helppoa virheen ilmetessä ymmärtää missä se on ilmennyt. Tämä johtaa verkkosovelluksen kokonaisvaltaisempaan hallintaan ja kehitystyön johdonmukaisuuteen. (Mazaika, 2021)

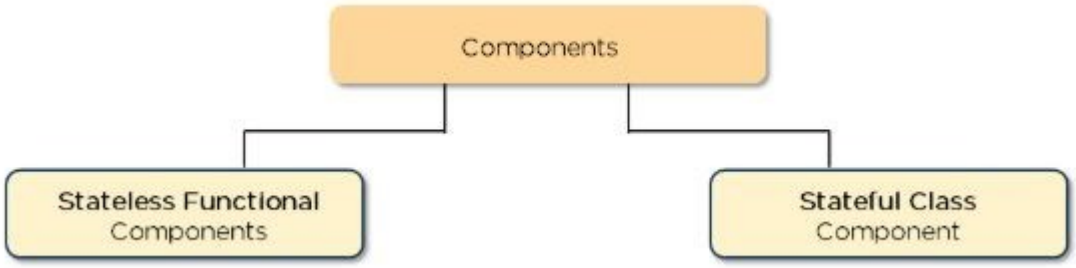
2.1.2.4.3 Komponentit

Komponentit ovat minkä tahansa React-sovelluksen rakennuspalikoita ja yksittäinen sovellus pitää sisällään useita erilaisia komponentteja. Komponentti on erittäin oleellinen osa käyttöliittymää ja React jakaakin käyttöliittymän useiksi itsenäisiksi, uudelleenkäytettäviksi osiksi, joita on mahdollista käsitellä erikseen. React:ssa käytetään kahta erilaista komponenttimallia, joita ovat funktionaaliset komponentit (functional komponents) sekä luokkakomponentit (class components).

Funktionaaliset komponentit eivät ennen pitäneet sisällään omaa state:a eli tilaa, (josta seuraavassa luvussa) vaan ne sisälsivät vain niiden oman renderöintimenetelmän, joten niitä kutsuttiin statelesskomponenteiksi. Kuitenkin nykyisin React Hook:ien avulla myös funktionaalinen komponentti voi käyttää tilaa. Hook:it esiteltiin Reactin versiossa 16.8 ja ne ovat funktioita, jotka linkittyvät React:in tilaan funktionaalisten komponenttien sisältä.

Myös luokkakomponentit säilyttävät ja hallitsevat tilaa. Luokkakomponenteilla on myös erillinen renderöintimenetelmä, jotta ne voivat palauttaa näytölle JSX:ää. JSX on JavaScriptin syntaksilaajennus,

jota suositellaan käytettäväksi React:n kanssa. Luokkakomponentteja saatetaan myös kutsua nimellä stateful, johtuen niiden ominaisuudesta pitää sisällään tilan. Funktionaalisten komponenttien suosion ollessa kasvussa Hook:ien joustavuudesta johtuen, luokkakomponenttien käyttö uusissa projekteissa on vähentynyt. (Yamazaki, 2020)



Kuva 5 – React-komponentit

2.1.2.4.4 Tila

State on sisäänrakennettu React-objekti, jonka tarkoitus on sisältää tietoja komponentista. Komponentin tila voi vaihdella useinkin. Komponentin tilan muuttuessa, koko komponentti renderöidään uudelleen. Tilan muutos voi olla esimerkiksi järjestelmän luoma tapahtuma tai vastaus käyttöliittymän päässä tapahtuvaan käyttäjän toimintaan. Nämä muutokset puolestaan määräävät komponentin käyttäytymisen sekä sen renderöinnin käyttäjälle.

2.1.2.4.5 Properties

Properties eli lyhyemmin props, on tilan lailla React:n sisäänrakennettu objekti, joka tallentaa tagin tai tunnisteen määritteiden arvon, ja toimii HTML-määritteiden kaltaisesti. Propsit tarjoavat tavan siirtää dataa komponentista toiseen funktion argumenttien kaltaiseen tapaan. (Sufiyan, 2021)

2.2 TEKNIIKAT

2.2.1 JavaScript

JavaScript, joka usein lyhennetään JS, on olioperustainen, dynaaminen sekä kevytrakenteinen ohjelmointikieli, jossa käytetään first-class-funktioita. First-class-funktiolla tarkoitetaan funktioita, joita käytetään kuten mitä tahansa ohjelmointikielen muuttujaa. Tämä mahdollistaa esimerkiksi funktioiden viemisen argumentteina toisille funktioille, sekä niiden palauttamisen. Lisäksi first-class funktioita voidaan määritellä arvoiksi ohjelmassa käytetyille muuttujille. JavaScript on parhaiten tunnettu skriptikielenä, jota käytetään web-sivujen ohjelmoinnissa, mutta useiden käytännöllisten ominaisuuksiensa ansiosta sitä käytetään myös monissa selaimen ulkopuolisissa ympäristöissä. Sen ollessa myös prototyyppiperustainen ohjelmointikieli, on JavaScriptillä myös mahdollista luoda objekteja ennen niiden luokan määrittelyä.

JavaScriptiä ajetaan pääasiallisesti asiakassovellus-puolella ja sen perussyntaksissa on tarkoituksenmukaisesti kaavailtua yhdenkaltaisuutta sekä Javan, että C++ kielen syntaksien kanssa, jotta kielen oppiminen olisi sujuvampaa.

Client-puolella JavaScript hyödyntää core-kieltä toimittaen objekteja selaimen ja sen asiakirjaobjektimallin (DOM) ohjaamiseen. Tästä esimerkkinä: käyttöliittymän laajennusten avulla sovelluksen on mahdollista sijoittaa erilaisia elementtejä HTML-lomakkeelle, sekä vastata käyttäjän tekemiin toimintoihin, esimerkiksi hiiren klikkaukseen, lomakkeeseen tietojen syöttöön tai sivun navigointiin.

Serveripuolella JavaScript puolestaan hyödyntää core-kieltä toimittamalla JavaScript-serverille sen toimintaan tarvittavia objekteja. Serveripuolen laajennukset esimerkiksi mahdollistavat sovelluksen kommunikoinnin tietokannan kanssa, tarjoavat tiedon jatkumisen kutsujen välillä, sekä tiedostojen käsittelyn palvelimella. Tässä opinnäytetyössä käytetty Node.js on yksi JavaScriptin serveripuolen ratkaisuista.

JavaScript on Ecma Internationalin standardisoima. Ecma on Euroopan tieto- ja viestintäjärjestelmien standardointiliitto, joka loi standardoidun kansainvälisen JavaScript-pohjaisen ohjelmointikielen. Tämä JavaScriptin standardisoitu versio on nimeltään ECMAScript, ja se toimii kaikissa standardia tukevissa sovelluksissa. Täten yritykset voivat käyttää avointa vakiokieltä JavaScriptin toteutusta kehitettäessä. (Mozilla)

JavaScript on joustava ja käytännöllinen ohjelmointikieli, jota useimmat käytetyistä selaimista tukevat. Yhdessä HTML ja CSS kielien kanssa, se JavaScript on yksi websovelluskehityksen ydinkomponenteista. HTML:n ollessa vastuussa websivun rakenteesta, ja CSS:n puolestaan sen tyylistä ja ulkoasusta, jää javaScript:n tehtäväksi sekä toiminnallisuus, että interaktiivisuus selaimen kanssa. Tästä johtuen JavaScript on yleisesti käytetty websovelluksessa sekä siitä on johdettu useita erilaisia kirjastoja ja viitekehyksiä. Viitekehyksen ja kirjaston pääasiallinen ero on se, että kirjastot koostuvat funktioista, joita sovellukset voivat kutsua jonkin erillisen tehtävän suoritukseen, kun taas viitekehykset määrittelevät, kuinka sovelluskehittäjä kehittää sovelluksen. Nykypäivänä kirjastojen suosio on laskussa, kun taas viitekehykset nostavat suosiotaan. (Elliot, 2017) Viitekehyksistä mainittakoon sekä React, että Node.js, joita tässäkin opinnäytetyössä hyödynnetään ja käsitellään.

2.2.2 NoSQL

NoSQL eli ”Not only SQL” tai “non-SQL”, tarkoittaa relaatiomallista poikkeavaa datan varastointitekniikkaa, joka mahdollistaa datan varastoimisen lisäksi sen kyselyiden suorittamisen perinteisten SQL-rakenteiden ulkopuolella. NoSQL-tekniikan mahdollistama vapaus relaatiomallien raameista ei kuitenkaan poista sen kykyä tallentaa dataa myös relaatiomalleista. Relaatiomalleille tyypillisen taulukkomallin sijaan NoSQL tallentaa datan yhteen datastruktuuriin, esimerkiksi JSON-dokumenttiin. Relaatiomallista poikkeavan datavarastointijärjestelmän ansiosta NoSQL-mallit eivät vaadi kaaviota eli skeemaa, joten kyseinen tapa mahdollistaa vikkelän skaalautuvuuden sekä tyypillisesti epästrukturoidut datakokoelmat. NoSQL:ää käytetään myös hajautetun tietokannan tyyppinä. Tämä tarkoittaa datan olevan kopioitu ja varastoitu useammalle eri serverille, jotka voivat olla joko etäkäyttöisiä tai paikallisesti suoritettuja. Hajautettu tietokantatyyppi takaa tietojen saatavuuden sekä luotettavuuden. Jos jostain syystä osaan tiedostoista ei saada yhteyttä, hajautetussa mallissa tietokannan muut osat kuitenkin toimivat ja datan käsittelyä voidaan jatkaa. Tämä tulee eduksi erityisesti suurissa yrityksissä, joilla on tarvetta käsitellä suuria määriä dataa suurien nopeuksien puitteissa unohtamatta tarvetta skaalata dataa nopealla volyymillä verkkosovellusten ajoa varten. Pilvipalveluiden, Big Datan

eli suurten järjestelemättömien datamäärien sekä mobiili- ja websovellusten yleistyessä NoSQL-tietokannat tarjoavat edellä mainittuja ominaisuuksia, joka tekee kyseisestä mallista suositun valinnan sovelluskehittäjien keskuudessa sen suorituskyvyn ja helppokäyttöisyydenkin takia. (IBM, 2019)

2.2.2.1 NoSQL vs. SQL

NoSQL- malliset tietokannat ovat kasvattaneet suosiotaan viime vuosina johtuen muun muassa relaatiomallisten (RDBMS, Relational Database Management System) kantojen huonosta horisontaalisesta skaalautuvuudesta Big Datan kanssa, sekä RDBMS:ään liittyvien tilanpuuteongelmien takia. NoSQL-mallit puolestaan tarjoavat tätä skaalautuvuutta monissa eri CAP-lauseen (Consistency, Availabitity, Partition Tolerance) kompromisseissa. Lauseen mukaan hajautettu tietovarasto voi antaa kerrallaan enintään kaksi seuraavista takuista: Johdonmukaisuus eli consistency (jokainen lukukerta palauttaa joko viimeisimmän muutoksen tai virheen), saatavuus eli availability (jokainen pyyntö palauttaa vastauksen, riippumatta node:jen yksilöllisestä tilasta), sekä eri osioiden toleranssi eli partition tolerance. Toleranssilla tarkoitetaan sitä, että klusteri ei anna virhettä huolimatta siitä, pudottaako verkko viestejä solmujen välillä.

NoSQL ja SQL eroavaisuudet voidaan määrittää kolmeen pääkategoriaan. Ensimmäinen eroavaisuus on datamallissa. RDBMS tietokannat ovat normalisoitua strukturoitua (taulukoitu) dataa varten, joka noudattaa tiukasti relaatiomallia. NoSQL-tietokannat puolestaan soveltuvat ei-relaatiomalliseen dataan, joista esimerkkejä ovat avain- arvo parit (key-value), asiakirjapuut (document tree), sekä kaaviot. Toinen eroavaisuus puolestaan on tapahtumakutsuissa. Kaikki RDBMS-kannat tukevat ACID-tapahtumia. ACID-johdonmukaisuus tarkoittaa tietojen eheyttä eli tiedot ovat johdonmukaisia niiden suhteiden ja rajoitusten kanssa jokaisen tietokantatapahtuman jälkeen, kun taas useimmat NoSQL-tietokannat tarjoavat puolestaan BASE-tapahtumia:

- useaan paikkaan taltioitu
- aina saatavilla oleva data
- kopioiden hetkellinen epäjohdonmukaisuus, kuitenkin muuttuen johdonmukaiseksi ilman täyttä varmuutta johdonmukaistumisen ajankohdasta

Kolmas pääeroavaisuus tulee CAP-lauseen kompromisseissa:

- RDBMS tietokannat priorisoivat vahvan johdonmukaisuuden kaiken muun edelle
- NoSQL-tietokannat tyypillisesti priorisoivat datan saatavuuden ja osioiden toleranssin eli horisontaalisen skaalautuvuuden
- NoSQL-tietokannat tarjoavat vain osittaista johdonmukaisuutta

Lisäksi eroista mainittakoon suorituskyky. RDBMS-tietokannat ovat suunniteltu nopeisiin tiedonsiirtoihin, päivittäen useita rivejä taulukoissa samanaikaisesti monimutkaisilla eheysrajoituksilla. SQL-kyselyt ovat ilmavia ja deklaratiivisia. RDBMS optimoi kyselyt relaatioalgebran avulla ja löytää parhaan toteutussuunnitelman. NoSQL-kannat puolestaan ovat suunniteltu käsittelemään tehokkaasti paljon RDBMS:ää suurempia datamääriä. NoSQLssä dataan ei liity suhteellisia rajoituksia eikä datan tarvitse olla taulukkomuotoista. NoSQL tarjoaa siis suorituskykyä korkeammalla tasolla luopumalla RDBMS:lle

tyypillisestä vahvasta johdonmukaisuudesta. Tässä mallissa tietoihin pääsy tapahtuu usein REST-rajapintojen kautta. REST:llä tarkoitetaan http-malliin perustuvaa arkkitehtuurimallia. Tosin NoSQL-kyselykielet (esimerkiksi GraphQL) eivät ole kehityksessä samalla tasolla SQL-kehityskielien kanssa, joten sovelluskehittäjän työksi jää itse huolehtia, kuinka ratkaista kyselyiden tehokkuus käyttötarkoituksesta riippuen. (Satish Chandra Gupta, 2021)

2.2.3 React-Bootstrap

Bootstrap on suosituin CSS ohjelmistokehys, jota käytetään responsiivisten ja mobiilille suunnattujen websivujen suunnitteluun ja toteutukseen. Projektissa käytetty React-Bootstrap korvaa Bootstrap JavaScript koodin. Kaikki sen komponentit on käännetty React-komponenteiksi, jotta niitä voi käyttää ilman tarpeettomia välikirjastoja. React-Bootstrap on yksi Reactin vanhimmista kirjastoista ja se on kasvanut ja kehittynyt yhdessä Reactin kasvun ja kehityksen mukana. React-Bootstrap perustuu jo olemassaolevaan Bootstrap stylesheetiin, joka mahdollistaa CSS:stä tuttujen Bootstrap teemojen käytön myös React-sovelluksissa. (React-Bootstrap, 2021)React-Bootstrapin nykyinen versio on 1.6.0, jota on käytetty myös opinnäytetyössä toteutetussa projektissa.

2.2.4 Non-Blocking I/O

2.2.4.1 Blocking

Blocking tarkoittaa käytännössä sitä, että erillisen JavaScript-osion suorittamisen Node.js prosessissa on odotettava, kunnes joku muu kuin JavaScript-toiminto on suoritettu. Tällainen tilanne syntyy sen johdosta, että tapahtumasilmukka ei pysty jatkamaan JavaScriptin suorittamista silloin, kun blocking-tapahtuma esiintyy.

Node.JS:sä JavaScript:ia joka osoittaa huonoa suorituskykyä johtuen suorittimen kulutuksesta sen sijaan, että se odottaisi muuta kuin JavaScript-toimintoa, ei yleensä kutsuta blocking-tapahtumaksi. Yleisimmin käytettyjä blocking-tapahtumia ovat Node.JS vakiokirjaston synkroniset menetelmät, jotka käyttävät libuv-kirjastoa. Libuv on monialustainen kirjasto, joka on alun perin luotu Node.JS:lle. Se on suunniteltu tapahtumalähtöisen asynkronisen input/output mallin ympärille. Myös Node.JS:n alkuperäisillä moduuleilla voi olla blocking-menetelmiä.

Kuitenkin kaikki Node.js vakiokirjaston I/O-menetelmät tarjoavat asynkronisia versioita, joissä ei esiinnyt blocking-tapahtumia, ja jotka hyväksyvät callback-funktiot. Lisäksi joillain metodeilla on myös blocking-vastineita, joiden nimet päättyvät ”Sync”.

2.2.4.2 Non-Blocking

Blocking-tapahtumat suoritetaan synkronisesti, kun taas non-blocking-tapahtumat suoritetaan asynkronisesti. Seuraavana kuvataan synkronisen tiedoston luku File System moduulin avulla:

```
1  const fs = require('fs');
2  const data = fs.readFileSync('/file.md');
3  //pysähtyy tähän, kunnes tiedosto on luettu
```

Kuva 6 - Esimerkki blocking-tapahtumasta

Alapuolella esimerkki vastaavanlaisesta asynkronisesta tapahtumasta:

```
1  const fs = require('fs');
2  fs.readFile('/file.md', (err, data) => {
3    if (err) throw err;
4  });
```

Kuva 7 – Esimerkki NonBlocking-tapahtumasta

Kuvassa 7 oleva esimerkki vaikuttaa toista esimerkkiä yksinkertaisemmalta, mutta siinä on epäedullinen blocking-tapahtuma. Toinen rivi estää seuraavan mahdollisen JavaScript-osan suorituksen siihen saakka, kunnes koko tiedosto on luettu. Lisäksi esimerkkien synkronisessa versiossa virheen tapahtuessa ohjelma kaatuu, sillä siitä puuttuu virheenkäsittely. Asynkronisessa versiossa puolestaan koodin kirjoittajalle jää päätettäväksi, näytetäänkö tulostuksessa mahdollinen esiintynyt virhe.

Seuraavana laajennettu esimerkki:

```
1  const fs = require('fs');
2  const data = fs.readFileSync('/file.md');
3  //pysähtyy tähän, kunnes tiedosto on luettu
4  console.log(data);
5  moreWork(); //suoritetaan console.log-funktion suorittamisen jälkeen
```

Kuva 8 - Laajennettu esimerkki blocking-tapahtumasta

Alapuolella samankaltainen, muttei vastaavanlainen asynkroninen esimerkki:

```
1  const fs = require('fs');
2  fs.readFile('/file.md', (err, data) => {
3    if (err) throw err;
4    console.log(data);
5  });
6  moreWork(); //suoritetaan ennen console.login suorittamista
```

Kuva 9 - Laajennettu NonBlocking-esimerkki

Kuvassa 9 olevassa esimerkissä console.log-funktiota kutsutaan ennen moreWork-funktiota. Toisessa esimerkissä fs.readFile-funktio on non-blocking-funktio, jolloin JavaScriptin suorittaminen jatkuu ja moreWork-funktiota voidaan kutsua ensin. Mahdollisuus kutsua moreWork-funktiota odottamatta koko tiedoston lukemista ensin on hyvää mallisuunnittelua, joka mahdollistaa paremman suorituskyvyn.

JavaScriptin suorittaminen Node.JS:ssä on yksisäikeistä (single threaded), joten samanaikaisuus viittaa tapahtumaketjun kapasiteettiin suorittaa JavaScriptin callback-funktioita muun työn suorittamisen jälkeen. Callback:lla tarkoitetaan funktiota, joka annetaan parametrina ulommalle funktiolle jonkin tehtävän suorittamiseksi myöhemmin. Kaiken koodin, jonka oletetaan suoriutuvan samanaikaisuutta mukaillen, on sallittava tapahtumaketjun jatkavan myös ei-JavaScript-operaatioiden (esimerkiksi I/O) suorittamista.

3 TYÖN SUUNNITTELU

Työssä tarkoituksena oli toteuttaa laiterekisteri käyttöobjektien (esimerkiksi iPhone) varaston hallintaa ja laitteiden kunnossapitotietojen varastoimista varten. Ohjelmiston suunnittelussa päätettiin yhdessä toimeksiantajan kanssa sen olevan päätelaitteelta paikallisesti käytettävä ohjelmisto. Päätelaitteena on tarkoitus käyttää tablettia ja tietokonetta. Ohjelmiston tarkoituksena on varastoida käyttöobjektien dataa sekä objekteille tehtävien kuntotarkastusten dataa. Lisäksi ohjelmiston on tarkoituksena toimia kirjaston lainausperiaatetta mukaillen käyttöobjektien lainaus- sovelluksen tavoin.

Tavoitteena oli kehittää laitteidenhallintajärjestelmä, jossa suunnitteluvaiheessa päätettiin hyödyntää NoSQL-tietokantamallia dokumenttitietokannan muodossa. Järjestelmään tallennettavien tietojen ollessa monitasoisia, katsottiin dokumenttimallin palvelevan parhaiten käyttötarkoitusta. Päätelaiteella olevan käyttöliittymän takana toimiva tietokantaosuus on tarkoitus sijoittaa verkkoon, jottei se esimerkiksi laitteen rikkoutuessa katoa. Suunnitteluvaiheessa tavoiteltiin selkeän ja loogisen käyttöliittymän luontia sekä sitä tukemaan mahdollisimman joustavasti toimiva tietokantaratkaisu. Suunnittelussa tietokantaratkaisun joustavuuden katsottiin olevan erityisen tärkeää hallintajärjestelmän kannalta. Näistä syistä rakentamiseen valittiin MERN-stack.

Kaikkia projektin osia ei suunniteltu heti projektin alussa vaan muutoksiin ja tarpeisiin reagoitiin ketterästi projektin edetessä.

Hallintajärjestelmään oli määrä toteuttaa:

- lainauspuoli laitteiden lainaamista varten
- tarkastuspuoli laitteiden tarkastusten tekemistä varten
- laitepuoli laitevaraston hallitsemista varten

Lisäksi oli määrä toteuttaa järjestelmänvalvoja-puoli ns. järjestelmänvalvojatyökaluille. Käyttöliittymä suunniteltiin toteutettavaksi React.JS:llä, sovelluksen keskiosan rajapintojen toteutuksessa suunniteltiin puolestaan käytettävän Express:a sekä Node:a. Data päätettiin varastoida MongoDB klusteriin, johon luodaan tietokanta sekä taulukot erilaisille tallennettaville datakokonaisuuksille. MongoDB klusteri sijaitsee pilvipalvelussa, eikä tietokantaa ole tarkoitus ajaa paikallisesti lainkaan.

3.1 TIETOKANNAN SUUNNITTELU

Laitepuolella tietokantaan on tarkoitus tallentaa:

- yksittäisen laitteen tiedot
- laitekorttipohjat
- laitekokonaisuudet

Laitteita on useita samanlaisia, joten tarkoituksena on olla yksi laitekortti per laitetyyppi, jolloin laitetta tallennettaessa siitä haetaan valmis laitekorttipohja, jonka perusteella tallennetaan kyseessä olevan laitteen yksilökohtaiset tiedot.

Laitekorttia luotaessa sille voi tallentaa sekä yksittäisiä ominaisuuksia että ominaisuusosioita. Laitekorteissa tulee olla sekä valmiiksi kannasta haettavia ominaisuuksia, jotka ovat kaikilla laitekorttipohjasta luoduilla laitteilla saman oletusarvon omaavia ominaisuuksia, että ominaisuuksia, joille käyttäjä luodessa määrittää laitekohtaisen arvon. Yksittäisiä ominaisuuksia voi olla esimerkiksi laitteen valmistajan määrittelemä tuotenumero, kun taas osiosta esimerkki voi olla vaikkapa valmistaja, jolloin osioon tallennetaan yksittäisinä ominaisuuksina esimerkiksi:

- valmistajan nimi
- yhteyshenkilö
- sähköposti

Laitteilla voi olla siis oletusominaisuuksia ja osioita, joita ei tarvitse erikseen päivittää jokaista laitetta luotaessa, sillä ne ovat kaikille laitteille samat arvot omaavia. Tällaisia vakio-ominaisuuksia sekä osioita voivat olla esimerkiksi laitteen virtalähteen tyyppi (akku/paristo) sekä osiosta vakioesimerkki voi olla esimerkiksi valmistajan tiedot. Laitekorteissa tulee lisäksi olla määritettynä:

- versionumero niiden päivittämistä ja ylläpitämistä varten
- aikaleima niiden luontipäivästä
- aikaleima päivityksestä
- korttipohjan yksilöivä _id, tietokannan toimivuutta varten

Yksittäiset ominaisuudet on tarkoitus tallentaa taulukkoon JSON-objektien kaltaisina kokonaisuuksina, jolloin laitekortilla sekä laitteella on yksi taulukko sen ominaisuuksille sekä yksi sen osioille. Yksittäisistä ominaisuuksista tallennetaan nimi, kuvaus sekä arvo.

Kuva 10 - Ominaisuudet-taulukon esimerkkiobjekti

Ylempänä kuvassa 10 esimerkki ominaisuudet-jonon objektista. Ominaisuuksien tavoin myös osiot tallennetaan omaan moniulotteiseen taulukkoonsa. Osiosta on tarkoituksena tallentaa:

- id,
- nimi
- kuvaus
- arvo- taulukko

Arvo- taulukko koostuu osiolle määritettävistä ominaisuuksista, joiden rakenteen tulee olla samanlainen, kuin aiemmin mainittujen, ylemmällä tasolla käytettävien yksittäisten ominaisuuksien. Osioita on tarkoitus kyetä käyttämään universaalisti, joten eri laitteen laitekorttia luotaessa voisi käyttää esimerkiksi Valmistaja-osiota, jolloin kaikkia siihen luotuja ominaisuuksia voi joko käyttää uudelleen, tai poistaa, jos niitä ei haluta laitekorttipohjalla käytettävän.

Laitteita luotaessa on siis tarkoitus hakea kannasta valmiiksi luotu laitekortti, johon on valmiiksi määritelty laitteesta tallennettavat tiedot. Täten jokaisen laitteen kantaan luominen on valmiiksi määritelty prosessi. Laitteille ei ole kantaan määritelty muita pakollisia tallennettavia tietoja kuin:

- yksilöivä _id
- versionumero
- luontiaikaleima
- päivitysaikaleima

Lisäksi on tallennettava tieto yksittäisen laitteen luonnissa käytetystä laitekorttipohjasta, jotta laitekorttipohjaa päivitettäessä myös kaikille yksittäisille laitteille tulee samat ominaisuudet tai osiot, joita laitekorttipohjaan lisätään. Lisäksi laitteille tulee tarvittaessa tallentaa:

- tieto siitä, onko laite kunnossa vai epäkunnossa
- sekä tieto siitä, onko laite lainassa
- kenellä se on lainassa

Loput laitteille tallennettavat tiedot määräytyvät käytetyn laitekorttipohjan luonnissa määriteltyjen ominaisuuksien ja osioiden perusteella. Rakenne osioissa ja ominaisuuksissa ei muutu, ainoastaan se, onko tieto esitäytetty pohjaa luotaessa, vai täytetäänkö siihen arvo vasta laitetta luotaessa.

Lisäksi laitepuolella on ominaisuus laitekokonaisuuksien tekemiselle. Laitekokonaisuuksista tulee tallentaa tiedot laitekokonaisuuden

- nimestä
- kuvauksesta
- _id
- luonti-, muokkaus- ja versioleimat

Lisäksi siinä tulee olla jonomuotoinen tieto sen sisältämistä laitteista, jossa laitteet eritellään label:in sekä value:n mukaan, jotka tässä tapauksessa tarkoittavat label:in olevan laitteen selkokielinen nimi yhdistettynä sen laitenumeroon, sekä value:n viittaavan laitteen kannassa käytettyyn yksilöivään _id tunnisteeseen. Tämän lisäksi laitekokonaisuudessa on oltava erikseen jono laitekokonaisuuden omille ominaisuuksille, joiden rakenne toimii kuten ylempänä kuvattujen yksittäisten ominaisuuksien taulukko.

Tarkastuspuolella kantaan tallennetaan tietoja tarkastuspohjista sekä laitteille tehdyistä tarkastuksista. Tarkastuslomakkeen rakenne on käytännössä samankaltainen, kuin laitekorttipohjienkin. Tarkastuslomakkeelle määritetään ensimmäisen tason tietoja, joita ovat:

- yksilöivä _id
- lomakkeen nimi
- lomakkeen kuvaus
- luontileima
- päivitysleima
- versionumero

- tarkastukset-taulukko

Tarkastukset- taulukkoon tallennetaan laitteelle tehtäviä tarkastuksia, jotka toimivat laitteille lisättyjen ominaisuuksien kaltaisesti. Näihin voidaan määritellä avain-arvo pareja, ja yhdessä objektissa on oltava ainakin arvot:

- nimi
- kuvaus

Edellä mainittujen lisäksi tarkastus- objektilla voi olla arvona boolean eli totuusarvo, sekä tyyppi, joka on merkkijono-muotoista tietoa siitä, millaisena tarkastusta tehtäessä kyseiseen kohtaan arvo tulee syöttää.

Tarkastuksista puolestaan on tarkoitus tallentaa edellä mainittuja osa-alueita enemmän tietoa, jotta käyttöliittymässä kyetään käsittelemään niitä halutulla tavalla. Ensimmäiselle tasolle tallennetaan:

- _id
- nimi
- kuvaus
- laitteen numero, jolle tarkastus tehdään
- totuusarvo siitä, oliko tarkastus kunnossa vai ei
- luonti- ja päivitysleimat
- versionumero
- tarkastukset-taulukko

Tarkastukset-taulukkoon tulee tarkastusobjekteja, joissa jokainen tarkastus pitää sisällään ainakin nimi-merkkijonon, kuvaus-merkkijonon, sekä kunnossa-totuusarvon. Lisäksi tarkastusobjektilla voi olla:

- tarkastuspohjan luonnissa lisätty arvo- totuusarvo
- tyyppi-merkkijono
- sekä arvo-input-merkkijono

Lainaus-puolella kantaan on tallennettava tietoa lainauksista. Lainaus-dokumentin ensimmäisellä tasolla on:

- _id
- lainaaja- objekti
- sekä laitteet- taulukko
- luontileima
- päivitysleima
- versionumero

Lainaaja-objekti pitää sisällään:

- lainaajan nimen
- yksilöivän numeron

Laitteet-taulukkoon puolestaan tallennetaan objekteina lainatut laitteet, joista tallennetaan nimi sekä kannassa yksilöivään tunnisteeseen viittaava laitteelle sen luonnissa määritelty _id.

Edellä mainittujen lisäksi kannassa on omat kokoelmat käyttäjille, sekä laitekorttien luonnissa käytetyille oletusosioille. Oletusosioista tallennetaan:

- yksilöivä _id
- luontileima
- versionumero
- päivitysleima
- nimi
- kuvaus
- arvot-taulukko

Arvot-taulukko pitää sisällään objekteja, jotka toimivat ominaisuus- taulukon tavoin, jolloin jokaisen osion jokaisella ominaisuudella on arvot nimi, kuvaus sekä arvo. Kaikki edellä mainitut ovat merkkijonoja.

3.2 KESKITASON SUUNNITTELU

Työssä on tavoitteena tehdä monisivuinen web-sovellus, joten keskitason valinnassa sekä rakenteessa on otettava huomioon sen komponenttien kykeneväisyys tämän ominaisuuden ratkaisemiseen. Lisäksi sovelluksen toteuttamiseksi on suunniteltava rajapinta käyttöliittymän ja palvelimen väliin. Sovelluksessa on tarkoitettu käytettäväksi Model View Controller-mallia, jota keskiosan tulisi myös tukea. Kannasta tullaan hakemaan ja lisäämään sekä muokkaamaan dataa useiden erilaisten pyyntöjen avulla ja tämä on otettava huomioon keskiosan suunnittelussa ja valinnassa.

3.3 KÄYTTÖLIITTYMÄN SUUNNITTELU

Käyttöliittymää on tarkoitus käyttää vain yhdeltä päätelaitteelta paikallisesti, joten sen suunnittelussa ei tarvitse ottaa huomioon seikkoja tämän ulkopuolelta. Käyttöliittymä rakennetaan React:illa ja sillä voidaan hallinnoida kaikkia hallintajärjestelmän osa-alueita. Käyttöliittymässä on tarkoitus olla:

- laite-
- tarkastus-
- lainaus-
- järjestelmänvalvoja-puolet

Järjestelmänvalvojapuolelle on tarkoitus päästä ainoastaan henkilöiden, joille järjestelmänvalvoja-oikeudet on annettu, täten ns. normaalin tason käyttäjät eivät pääse tälle puolelle lainkaan, eivätkä käsiksi järjestelmänvalvojatyökaluihin.

Käyttöliittymän suunnittelussa otettiin huomioon ohjelman käyttäjien tietämys projektin rakenteesta. Projekti suunniteltiin rajatulle käyttöryhmälle, jotka tuntevat rekisteriin rekisteröitävät laitteet ja tarkastustoimenpiteet ennalta. Käyttöliittymän suunnittelussa otettiin huomioon yksinkertaisuus, jotta erilaisten toimintojen tekeminen olisi joustavaa ja helppokäyttöistä. Käyttöliittymän suunnittelussa pyrittiin mahdollisimman johdonmukaiseen ratkaisuun.

3.3.1 Laitepuoli

Laitepuolella on tarkoitus pystyä luomaan laitekorttipohjia. Laitekorttipohjiin kerätään erilaisia tietoja laitemalleista, joten siihen on tarkoitus kyetä sekä lisäämään ja poistamaan osioita sekä ominaisuuksia, jonka jälkeen laitekorttipohja on kyettävä tallentamaan. Laitepuolen käyttöliittymään kuuluu myös laitekorttien täyttäminen. Tällöin on tarkoituksena hakea valmiiksi täytetty laitekorttipohja, ja täyttää siihen yksittäisen laitteen tiedot. Laitepuolella tulee olla myös erikseen valikko laitteiden selaamista varten, josta voi tarkastella kaikkia järjestelmään syötettyjä laitteita sekä niiden yksilökohtaista laitekorttia. Laitteiden selauksesta tulee päästä laitteen valinnan jälkeen selaamaan kyseiselle laitteelle tehtyjä tarkastuksia, jossa tarkastusten tulee listautua, ja kyseisen tarkastuslomakkeen aueta sitä napauttamalla. Laitteen tietoja tarkastellessa tulee päästä myös muokkaamaan kyseisen laitteen laitekorttia, jolloin avautuvalle laitekortille tulee päästä lisäämään ja poistamaan osioita sekä ominaisuuksia. Laitekortti tulisi myös kyetä poistamaan, tosin mahdollisesti tämän ominaisuuden tulisi olla järjestelmänvalvojatyökalujen takana. Laitepuolella tulisi olla myös listaus viallisista laitteista, josta pääsee suoraan tarkastelemaan kaikkia laitteita, joiden edellisen tarkastuslomakkeen täytössä on tultu sellaiseen lopputulokseen, että laite on merkitty vialliseksi.

Lisäksi laitepuolella tulee olla valinta laitekokonaisuuksien luomiselle. Tämä tarkoittaa esimerkiksi sitä, että erilaisia laitteita kerätään vaikkapa salkkuun, jolloin niistä luodaan laitekokonaisuus. Laitekokonaisuuden luonnissa sille tulisi olla mahdollista antaa nimi sekä kuvaus, lisätä kokonaisuudelle erilaisia ominaisuuksia sekä liittää laitteita kyseiseen salkkuun. Salkku tulee voida myös tallentaa. Tämän lisäksi laitepuolella tulee olla valinta laitekokonaisuuksien selaamiselle, jossa näkyy kaikki luodut laitekokonaisuudet. Valittaessa laitekokonaisuus, tulee kyseisen kokonaisuuden laitteiden listautua, sekä kokonaisuudesta pitäisi kyetä poistamaan laitteita. Tämänkin poisto-ominaisuuden lopullista sijaintia harkitaan laitepuolen ja järjestelmänvalvoja-puolen välillä.

3.3.2 Tarkastuspuoli

Tarkastuspuolella puolestaan tulee olla erikseen valinnat yksittäisen laitteen tarkastuksen tekemistä varten, uuden tarkastuslomakepohjan luomista varten sekä selausvalikko laitteelle tehdyille tarkastuksille. Laitteen tarkastusta varten on kyettävä valitsemaan ensin laite, jolle tarkastus tehdään ja sen jälkeen valittava laitteelle tehtävä tarkastus listasta kyseisen laitemallin tarkastuskortteja. Tämän jälkeen tulisi siirtyä tarkastuksen tekonäkymään, jossa tarkastuskorttipohjaa luotaessa ennalta määritetyt tarkastuskohteet tulevat näkyviin tarkastuksen kuvauksen sekä kunnossa/ei kunnossa valinnan ja lisätietokentän kanssa. Puolestaan tarkastuslomakepohjaa luotaessa siitä kerätään ylös erilaisia tarkastuskohteita, joille tulee lisätä nimi, kuvaus, sekä voidaan valita, syötetäänkö tarkastus-

kohteelle arvo laitteen tarkastusta tehtäessä. Arvolle tulisi olla mahdollista asettaa myös tyyppi. Tarkastuspuolen valikon viimeisessä kohdassa eli laitteiden tarkastusten selaamisessa, avautuu laitelistaussivu, josta valitsemalla laite, päästään tarkastelemaan sille tehtyjä tarkastuksia.

3.3.3 Lainauspuoli

Lainauspuoli on suunniteltu toteutettavaksi siten, että se sisältää valikon toiminnallisuudet lainaamiselle, lainausten listaukselle sekä palautuksille. Laitteita lainatessa täytyy ensin tunnistautua, jonka jälkeen on mahdollista valita alasvetovalikosta haluamansa laitteet. Tämän jälkeen tehdään jonkinlainen tarkistus käyttäjän lainaamista laitteista, esimerkiksi ponnahdusikkunan muodossa. Lisäksi lainauspuolella tulisi olla listaus käyttäjän lainaamista laitteista sekä mahdollisuus palauttaa jokainen laite erikseen sekä poistaa lainakuitti. Tämän lisäksi tulisi olla mahdollisuus kaikkien lainakuittien tarkastelulle, josta on mahdollista nähdä, kenellä on mikäkin laite lainassa.

Järjestelmänvalvoja-työkalujen puolella tulisi olla valikot osioiden muokkaukselle, sekä jo valmiiksi täytettyjen laitekorttipohjien muokkaukselle. Osioiden muokkauksen valittua, tulisi päästä listaukseen valmiiksi esitäytetyistä osioista, josta yksittäisen osion valitsemalla tulisi päästä muokkaamaan sen ominaisuuksia. Osioiden muokkauspuolella tulisi olla myös mahdollisuus lisätä uusi esitäytetty osio. Järjestelmänvalvoja-työkalujen laitekorttien muokkauspuolella sen sijaan tulisi olla mahdollisuus valita kaikkien laitekorttipohjien listasta muokattava laitekorttipohja. Tämän valittua tulisi aueta näkymä, jossa on mahdollista lisätä, muokata sekä poistaa laitekorttipohjalle ominaisuuksia sekä osioita. Järjestelmänvalvoja-työkaluihin tulisi sisällyttää myös mahdollisuus poistaa laitekorttipohja.

4 TYÖN TOTEUTUS

4.1 TOTEUTUKSEN OSIA

Sovelluksessa käytetään useaan kertaan samaa logiikkaa sen takana toimivassa ohjelmistossa. Tässä kappaleessa käsitellään esimerkkejä sovelluksessa käytetyistä sille tyypillisistä logiikoista ja toimintamalleista.

Mitä tahansa tietojenkeräämislomaketta luotaessa ensin renderöidään näkymä, joka pitää sisällään kentät nimelle ja kuvaukselle sekä ominaisuudet-jonon ensimmäisen muuttujan tietokentät. Käyttäjän kirjoittaessa merkin mihin tahansa kenttään, kutsutaan funktiota, joka tarkkailee syöttökenttiä. Alla kuvassa 11 havainnointi funktiosta kommentteineen.

```
handleChange = (e) => {
    e.preventDefault();

    //tutkii, onko eventissä kyseessä ominaisuudet-arrayn objekti
    if (["osioNimiInput", "osioKuvausInput", "osioArvoInput"].includes(e.target.dataset.key)) {
        //tekee kopion arraysta
        let values = [...this.state.inputValues]
        //käyttää targetin dataset.idtä ja dataset.keytä määrittelemään dynaamisesti paikan arrayssa,
        //johon asettaa targetin valuen, eli käyttäjän syöttämän arvon
        values[e.target.dataset.id][e.target.dataset.key] = e.target.value;
        //tallentaa stateen
        this.setState({ values }, () => { })
    }
    else {
        //lisää stateen targetin nimen mukaisen objektin, jolle antaa arvoksi targetin valuen
        this.setState({ [e.target.name]: e.target.value });
    }
}
```

Kuva 11 - Tyypillinen handleChange-funktio

Uutta kenttää luotaessa tilaan luodaan uusi objekti:

```
lisaaUusiKentta = (e) => {

    this.setState((prevState) => ({
        inputValues: [...prevState.inputValues, { nimiInput: "", kuvausInput: "", arvoInput: "" }],
    }));
}
```

Kuva 12 - Uuden kentän luontifunktio

```
poistaOminaisuus = e => {
    let inputValues = [...this.state.inputValues];
    let index = e.target.dataset.index
    inputValues.splice(index, 1);

    this.setState({ inputValues }, () => console.log(this.state.inputValues))
}
```

Kuva 13 - Ominaisuuden poistofunktio

Modalista eli dialogista tuotu data tallennetaan luotavan tietojenkeruulomakkeen tilaan sille varattuun muuttujaan.

```
getModalData = (osio) => {

    if (typeof (osio) !== 'undefined' || osio != null) {

        const newState = Object.assign({}, this.state);
        newState.lomake.osiot.push(osio);
        this.setState(newState);
    }

}
```

Kuva 14 - Dialogin datan tuonti

```
deletePartition = (id) => {

    let partitionId = id;
    let lomake = this.state.lomake;
    let index = lomake.osiot.findIndex(partition => partition.osionId === partitionId);
    lomake.osiot.splice(index, 1);
    this.setState({ lomake }, () => console.log(this.state.lomake))
}
```

Kuva 15 - Osion poisto tilasta

Alla kuvassa 16 sovellukselle tyypillisen renderöintifunktion logiikkaosuus havainnoivine kommentteineen.

```
//mapataan staten inputValues- array
inputValues.map((itemValue, index) => {
    //dynaamisuuden vuoksi mapataan erikseen objektien key-value parit
    let objectKeys = Object.keys(inputValues[index]);
    let inputItems = objectKeys.map((value, i) => {
        return (
            //inputitem-komponentti saa propseina ominaisuuden arvon, lomakkeen tyypin, objektin kokonaisuudessaan sekä indexin.
            <InputItem value={itemValue[value]} formType="laitekortti" property={objectKeys[i]} index={index}></InputItem>
        )
    })
    return (
        //poistonapille annetaan parametreinä ominaisuuden indeksi, sekä ominaisuuden poistofunktio
        <div>
            {inputItems}
            <Button key={"delButton" + index} variant="outline-danger" data-index={index} style={{ float: 'right' }} onClick={(e)
        </div>
    )
})
```

Kuva 16 – Renderöinti

Käyttäjän syötettyä kaikki tiedot ja painettua Valmis-nappia, avataan results-dialogi, jota käytetään useaan eri otteeseen sovelluksessa. Results-dialogiin renderöidään data näytettäväksi käyttäjälle tarkastusta varten. Käyttäjän ollessa tyytyväinen dataa, tutkitaan mistä lähteestä data results-dialogille saatiin, jonka perusteella se lähetetään eteenpäin tietokantaan.

```
handleSubmit = (e) => {

    e.preventDefault();

    let data = this.state.dataFromCreate;
    let url = "";

    if (this.props.callingComponent === "addNewDeviceCardTemplate") {
        url = 'http://localhost:5000/devicecardtemplates/add'
    }
    else if (this.props.callingComponent === "addNewDevice") {
        url = 'http://localhost:5000/devices/add'
    }
    else if (this.props.callingComponent === "addNewCheckUpCard") {
        url = 'http://localhost:5000/checkuptemplates/add'
    }
    else if (this.props.callingComponent === "updateDevice") {
        url = 'http://localhost:5000/devices/update/' + this.state.dataFromCreate._id

    } else if (this.props.callingComponent === "updateDeviceCardTemplate") {
        url = 'http://localhost:5000/devicecardtemplates/update/' + this.state.dataFromCr
    }
    //luodaan databody kantaan meneville arvoilla
    if (typeof (data) !== 'undefined' || data != null) {
        let databody = {
            lomakkeenNimi: this.state.dataFromCreate.lomakkeenNimi,
            lomakkeenKuvaus: this.state.dataFromCreate.lomakkeenKuvaus,
            sarjanumero: this.state.dataFromCreate.sarjanumero,
            ominaisuudet: this.state.dataFromCreate.ominaisuudet,
            tarkastukset: this.state.dataFromCreate.tarkastukset,
            osiot: this.state.dataFromCreate.osiot,
            laitekortti: this.state.dataFromCreate.laitekortti
        }
        //kutsutaan axiosta ja viedään data kantaan
        axios.post(url, databody)
            .then(res => console.log(res.data));

        this.handleClose()
    }
}
```

Kuva 17 - Results-dialogin kantaan vientifunktio

Tämän jälkeen tietolomake viedään oikean reitityksen kautta kantaan sille varattuun kokoelmaan.

```
//lisää uusi
router.route('/add').post((req, res) => {
  const lomakkeenNimi = req.body.lomakkeenNimi;
  const lomakkeenKuvaus = req.body.lomakkeenKuvaus;
  const sarjanumero = req.body.sarjanumero;
  const ominaisuudet = req.body.ominaisuudet;
  const osiot = req.body.osiot;
  const laitekortti = req.body.laitekortti;
  const lainassa = '';

  const newDevice = new Device({
    lomakkeenNimi,
    lomakkeenKuvaus,
    sarjanumero,
    ominaisuudet,
    osiot,
    laitekortti,
    lainassa
  });

  newDevice.save()
    .then(() => res.json('Device added!'))
    .catch(err => res.status(400).json('Error: ' + err));
});
```

Kuva 18 - Laitteen reititys kantaan

Kaikille kantaan vietäville tiedoille on reitityksessä sekä sovelluksen logiikassa CRUD-toiminnallisuudet. Näiden lisäksi on toteutettu useita muitakin palveluita, esimerkiksi laitteille on erikseen lainassa-muuttujan päivitykselle oma funktio. Tämän lisäksi sovellukseen on tehty erilaisia funktiota esimerkiksi kaikkien laitteiden hakemiselle, jotka on luotu tietyllä laitekorttipohjalla sekä funktio kaikkien lainaamattomien laitteiden hakemiselle.

```
//päivitä lainassa-property
router.route('/updateloan/:id').post((req, res) => {
  Device.findOneAndUpdate({
    "_id": req.params.id
  },
    {
      $set:
        { 'lainassa': '' }
    })
    .then(receipt => res.json(receipt)
    .catch(err => res.status(400).json('Error: ' + err))
      );
});
```

Kuva 19 - Lainassa propertyn päivitys

4.2 KÄYTTÖLIITTYMÄ

Projektin aikana luotiin sovellukselle yksinkertainen käyttöliittymä, jonka avulla järjestelmän ominaisuuksia pystytään testaamaan ja järjestelmää hallinnoimaan. Käyttöliittymää luotaessa otettiin huomioon sen paikallisuus sekä käyttäjäkunta, joka koostuu pienestä osasta osaston työntekijöitä. Käyttöliittymä toteutettiin yhden sivun kokonaisuutena, toisin sanottuna Single Page Applicationina (SPA). Sovellus käynnistyy etusivulle, joka sisältää lyhyen tekstin sovelluksesta. Käyttöliittymässä on neljä eri pääosiota, joita ovat laitepuoli, tarkastuspuoli, lainauspuoli sekä järjestelmänvalvojan-työkalut. Nämä neljä pääosiota on sovelluksessa sijoitettu navigointipalkkiin sovelluksen yläreunaan. Näillä kaikilla neljällä pääosiolla on omat alavalikkonsa, joista navigoidaan hallintajärjestelmän eri toiminnallisuuksiin. Sovelluksen ulkoasun muotoilussa käytettiin React-Bootstrapin kirjastoja ja valmiita moduuleita. React Bootstrap korvaa React-sovelluksissa Bootsrap JavaScriptin. React Bootstrapissa kaikki komponentit on rakennettu tyhjästä ilman ylimääräisiä kirjastoja ja se on yksi React:n vanhimpia kirjastoja.

Kuva 20 – Esimerkki sovelluksen yhdestä päänäkymistä

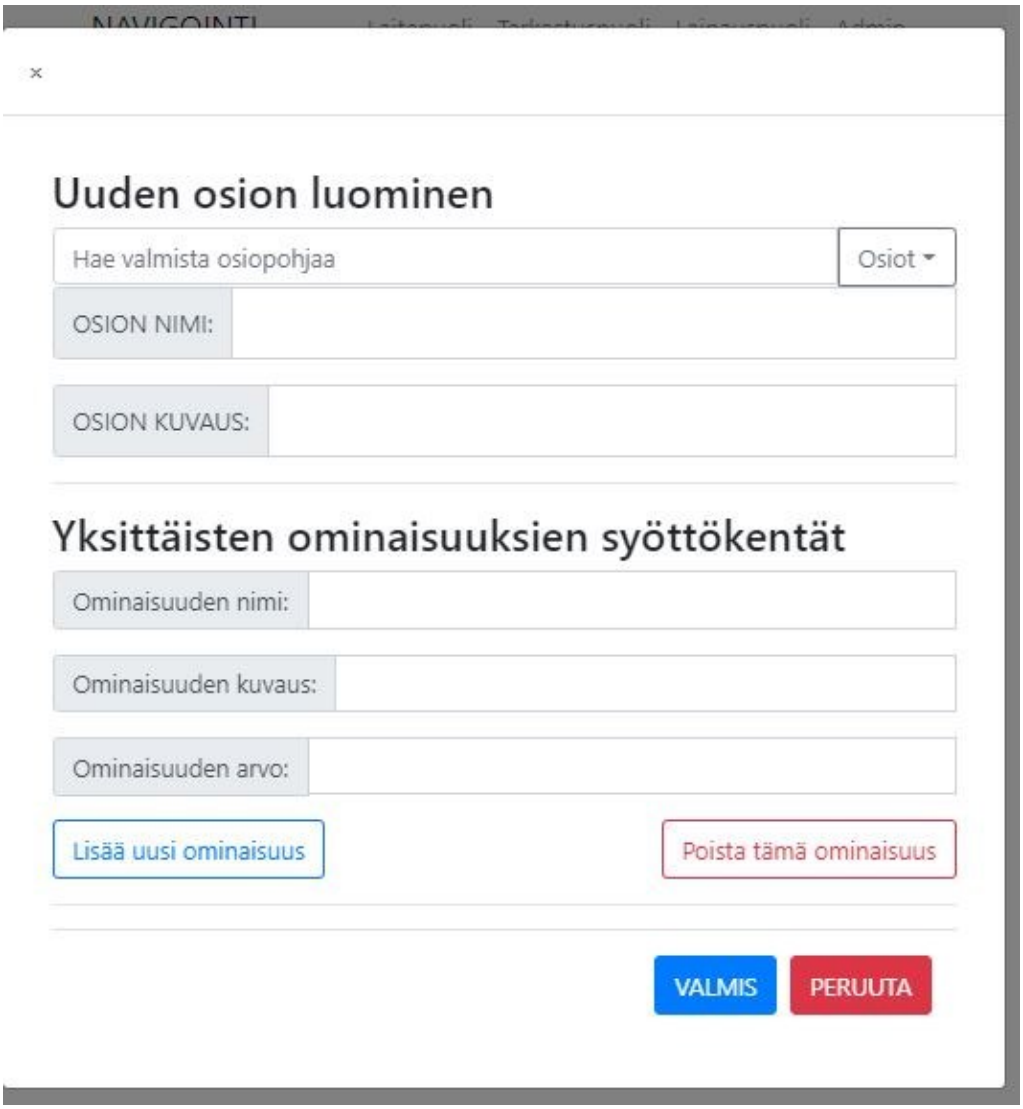
Ylempänä kuvattuna laitepuolen etusivu. Linkkejä painamalla sovellus siirtyy eteenpäin toiminnallisuuksiin. Yläpalkki on sovelluksen kaikilla sivuilla aina samassa paikassa, jotta sen kautta olisi helppo

navigoida. Uuden laitekorttipohja luominen-linkkiä painamalla sovellus navigoituu uuden laitekorttipohjan luontisivulle.

Kuva 21 - Uuden laitekorttipohjan luominen

Yllä kuvattu näkymä on sovelluksessa tyypillinen tietojen keräysnäkymä. Tämän kaltaisia näkymiä on sovelluksessa useamman kerran, laitelomakkeen täytössä, laitekokonaisuuden luonnissa, tarkastuslomakkeen luonnissa, tarkastuksen teossa, osioiden luonnissa sekä muokkauksessa sekä laitekorttipohjien muokkauksessa. Näkymässä kerätään käyttäjältä tietoja luotavasta lomakkeesta, jossa käyttäjä voi itse määritellä lomakkeelle tulevien ominaisuuksien ja osioiden määrän. Osio-nappia paina-

malla käyttäjälle aukeaa Osio-modal eli dialogi, johon käyttäjä voi lisätä osion tiedot, tai valita valmiina olevista osioista.



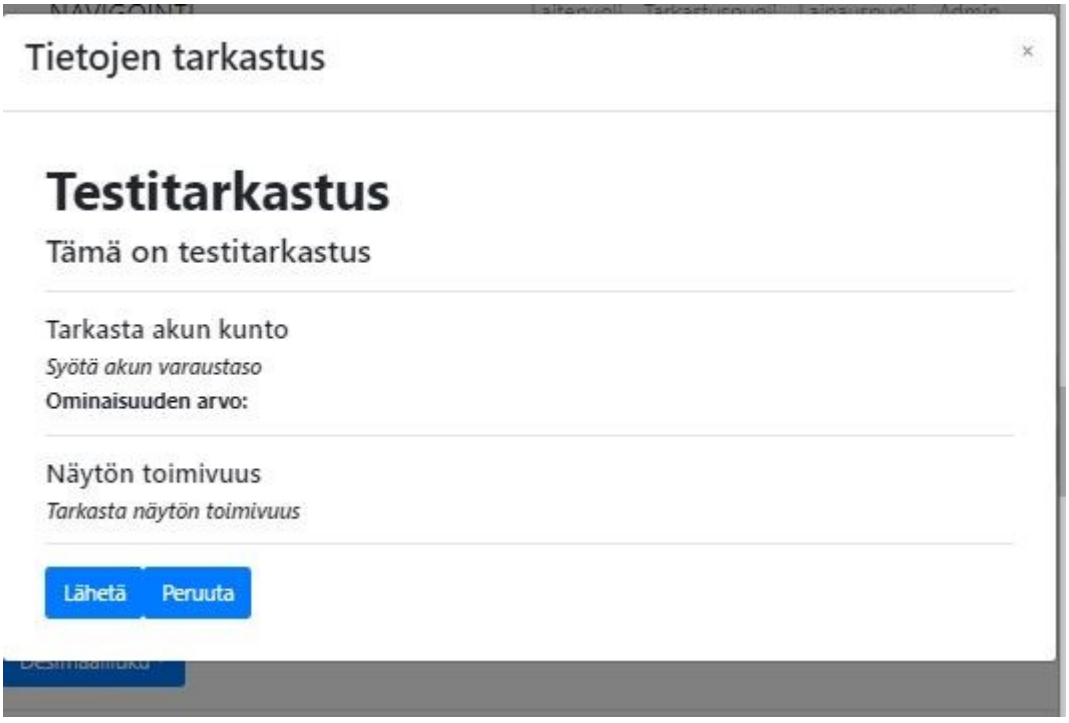
Kuva 22 – Osio-dialogi avattuna

Yllä kuvassa 22 osio-dialogi avattuna. Dialogin alussa on alasvetovalikko valmiin dialogi-pohjan haulle. Valikkoon haetaan kannasta valmiiksi luodut pohjat ja niiden ominaisuudet. Dialogista on ohjelmassa viisi eri versiota, joilla on eri toiminnallisuudet. Alla esimerkkejä dialogin toiminnallisuuksista.

```
handleChange = (e) => {
    e.preventDefault();

    //tutkii, onko eventissä kyseessä ominaisuudet-arrayn objekti
    if (["osioNimiInput", "osioKuvausInput", "osioArvoInput"].includes(e.target.dataset.key)) {
        //tekee kopion arraysta
        let values = [...this.state.inputValues]
        //käyttää targetin dataset.idtä ja dataset.keytä määrittelemään dynaamisesti paikan arrayssa,
        //johon asettaa targetin valuen, eli käyttäjän syöttämän arvon
        values[e.target.dataset.id][e.target.dataset.key] = e.target.value;
        //tallentaa stateen
        this.setState({ values }, () => { })
    }
    else {
        //lisää stateen targetin nimen mukaisen objektin, jolle antaa arvoksi targetin valuen
        this.setState({ [e.target.name]: e.target.value });
    }
}
```

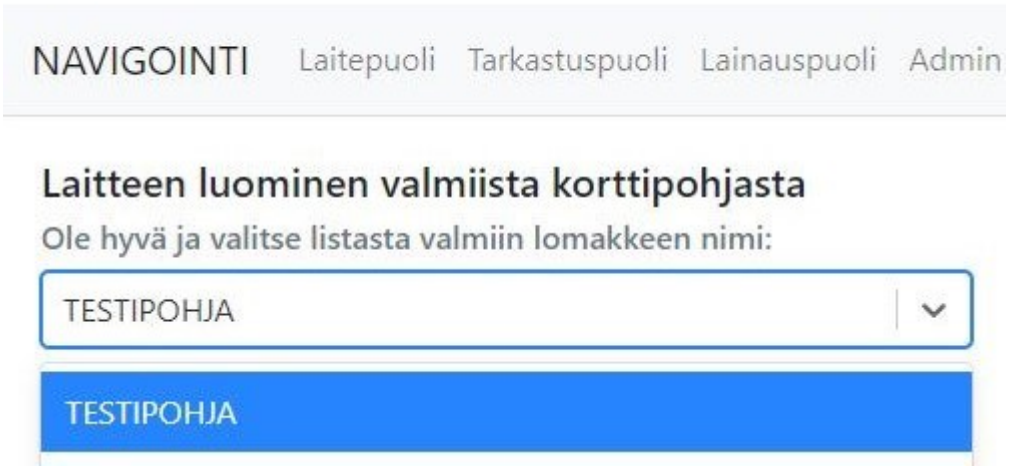
Kuva 23 – Osio-dialogin handleChange-metodi

Dialogeja käytetään projektissa myös kantaan vietävien tietojen yhteenvetoa tehtäessä, tietojen oikeellisuuden tarkistamiseksi. Alla esimerkki tietojen koonnissa käytettävästä dialogista, jossa on tehty uusi tarkastuslomakepohja.



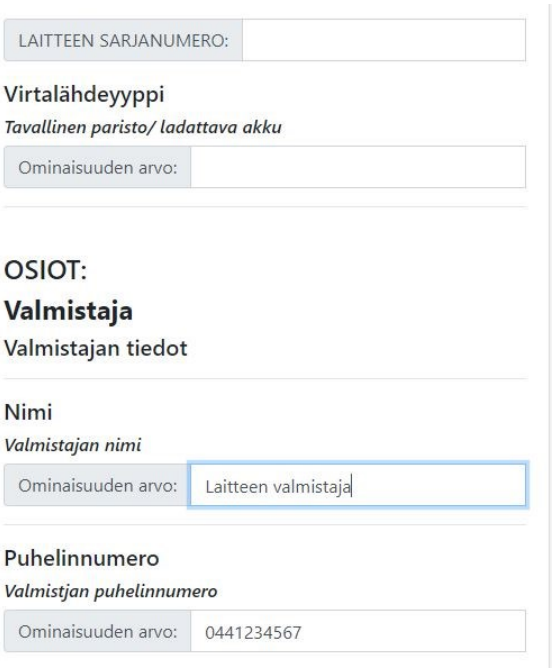
Kuva 24 - Tietojen tarkastuksessa käytetty dialogi

Uutta laitetta luotaessa valmiiksi tehdystä laitekorttipohjasta, käyttöliittymä avaa ensin valikon, josta valitaan listasta pohjana käytettävä laitekorttipohja.

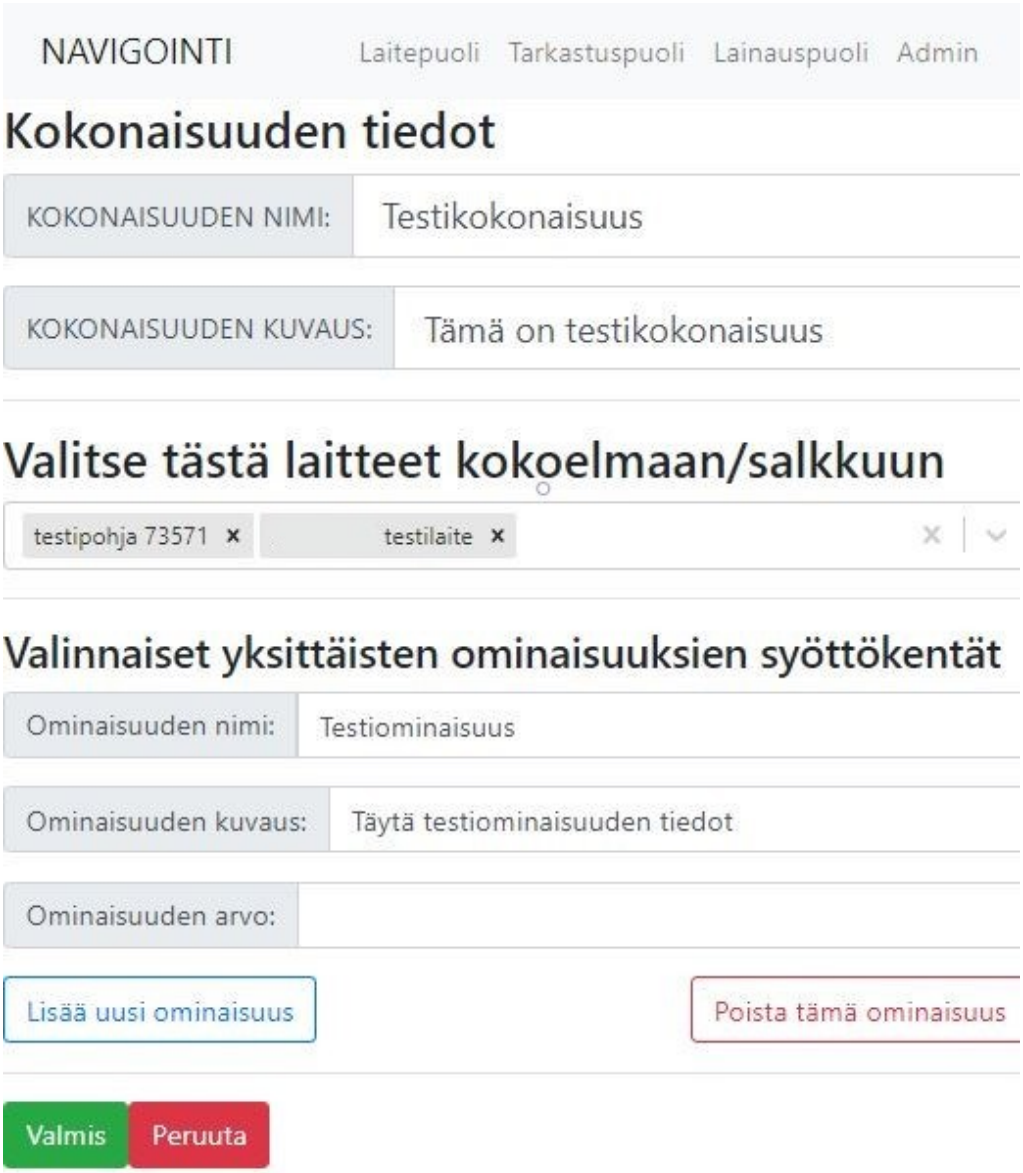


Kuva 25 - Laitepohjan valintanäkymä

Tämän jälkeen käyttöliittymä ohjaa laitteen tietojen täyttösivulle, alapuolella esimerkki laitteen luonnista, jossa on määritelty ominaisuudeksi akku, sekä laitekorttipohjan luonnissa on tiedetty kaikilla kyseisen tyyppisillä laitteilla olevan sama valmistaja, jolloin valmistajan tiedot on esitetäytetty osioon. Täten tietokannasta tulee valmiiksi tiedot osiossa kysyttäville ominaisuuksille, eikä niitä tarvitse täyttää erikseen jokaisen laitteen luonnin kohdalla. Laitteen luonnin jälkeen painettaessa ok-painiketta, avautuvat laitteen tiedot jo aiemmin kuvailtuun dialogiin, niiden tarkistamista varten ennen kantaan vientiä.



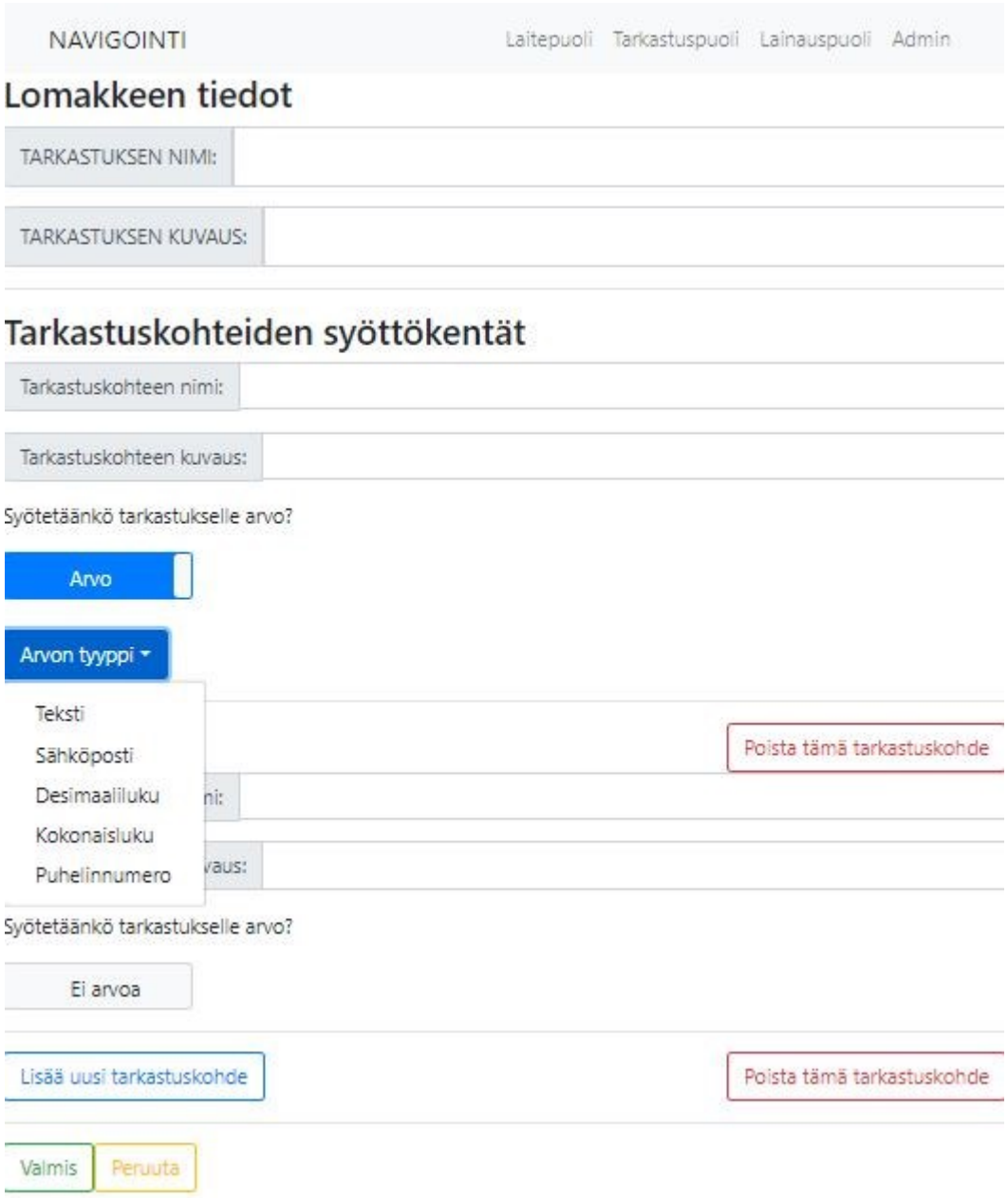
Kuva 26 - Esimerkki laitteen tietojen täyttämisestä



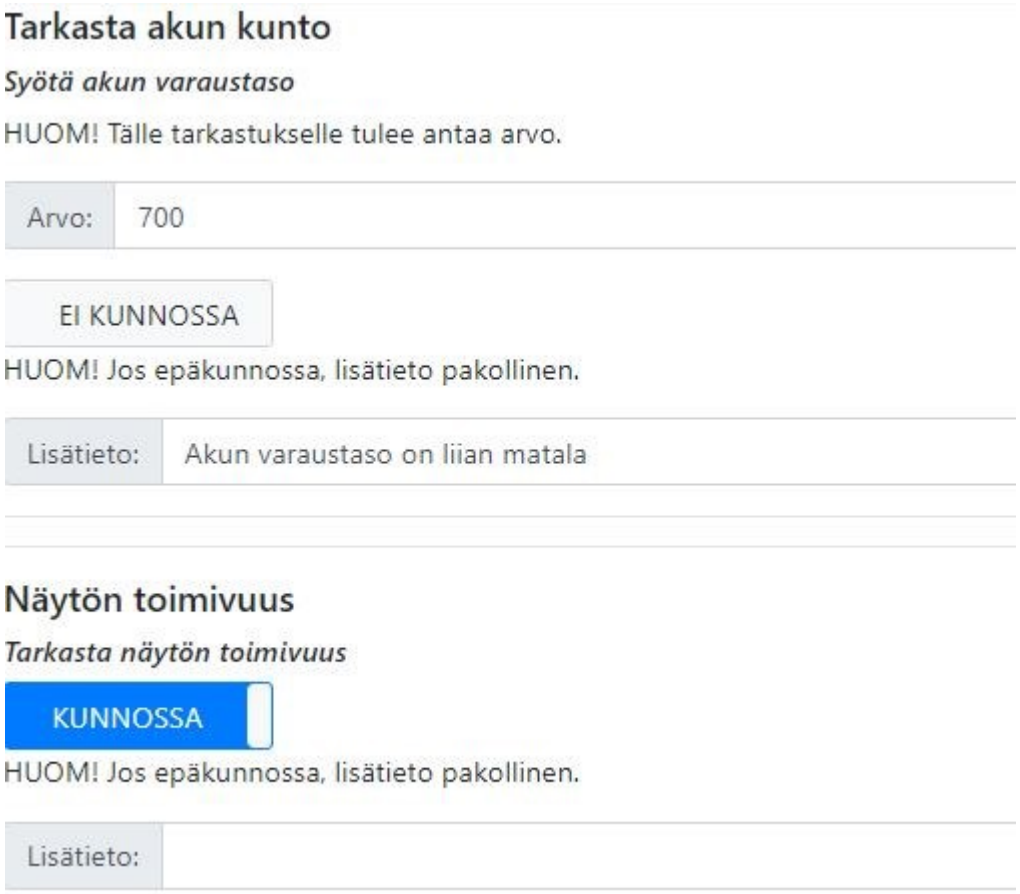
Kuva 27 - Kokonaisuuden luontinäkymä

Yllä kuvattuna kokonaisuuden luomisnäkymä. Tämä on projektille tyypillinen näkymä, jossa kerätään tietoja kantaan tallennettavasta objektista. Alussa on kentät nimen ja kuvauksen luomiselle, tämän jälkeen alasvetovalikko, josta valitaan kokonaisuuteen vietävät laitteet, ja sen jälkeen kentät ominaisuuksien luomiselle.

Alla kuvattuna tarkastuslomakkeen luominen. Tarkastuslomakkeelle voi lisätä haluamansa määrän tarkastuskohteita, joille tulee valita liu'uttamalla joko arvo tai ei arvoa. Käyttäjän vetäessä slider:n, tulee näkyviin arvon tyyppi-alasvetovalikko, josta voidaan valita tarkastukselle syötettävän arvon tyyppi.

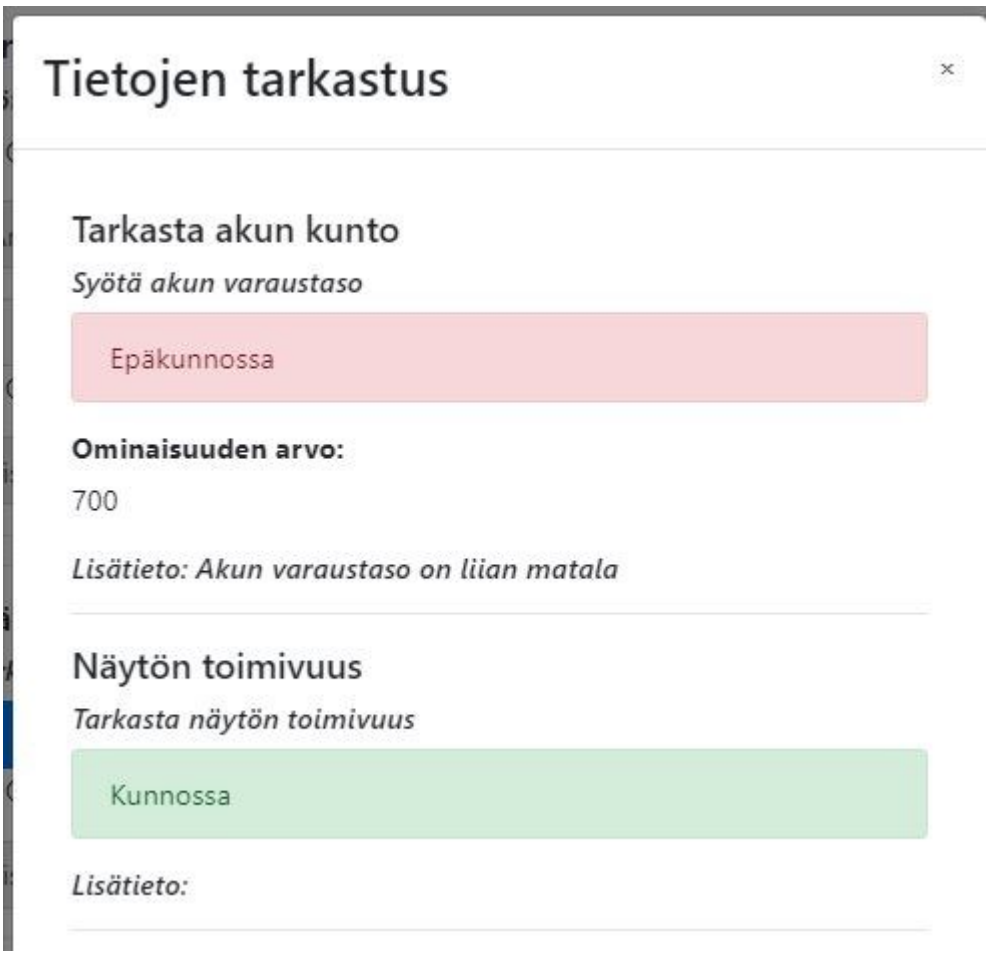


Kuva 28 - Tarkastuslomakkeen luontinäkymä

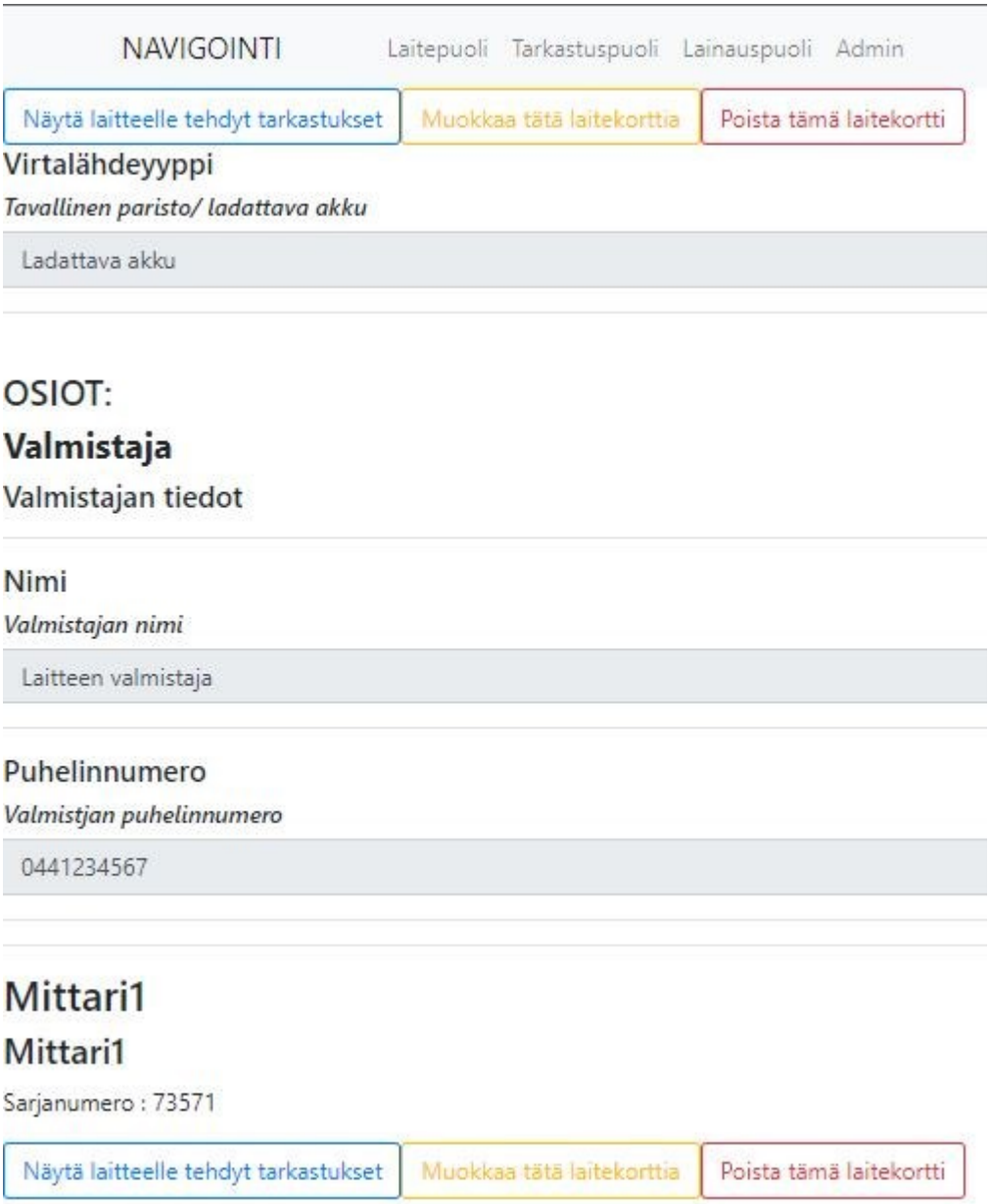


Kuva 29 - Näkymä tarkastuksen tekemisestä

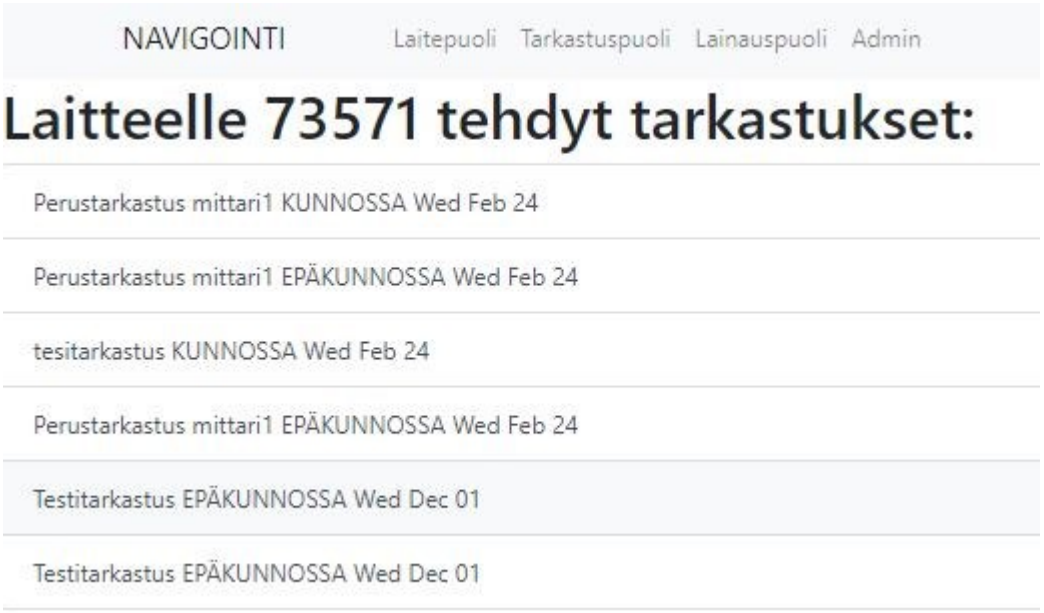
Yllä kuvattu projektissa syntynyt näkymä tarkastuksen tekemisestä. Tarkastuskohteet listautuvat allekkain, jonka jälkeen React:ssa haetaan tieto siitä, tuleeko tarkastukselle syöttää arvo vai ei. Jos arvo tulee syöttää, tulostetaan tarkastuskohteelle myös arvon syöttökenttä. Jos arvolle määriteltiin tyyppi tarkastuspohjan luonnissa, käyttöliittymän ei tule päästää käyttäjää eteenpäin ennen kuin syöttökenttään on annettu kyseistä tyyppiä oleva arvo. Slider:sta tulee määritellä, onko tarkastuskohde kunnossa vai ei. Jos tarkastuskohde ei ole kunnossa, tulee sille syöttää lisätieto. Lisätieto on myös valinnainen täytettäessä tarkastuskohdetta, joka on kunnossa. Alapuolella vielä esimerkki tietojen koontidialogista, joka aukeaa painettaessa Valmis-painiketta, ja johon on listattu tarkastuksessa kerätyt tiedot.



Kuva 30 - Tarkastustietojen koontidialgoi



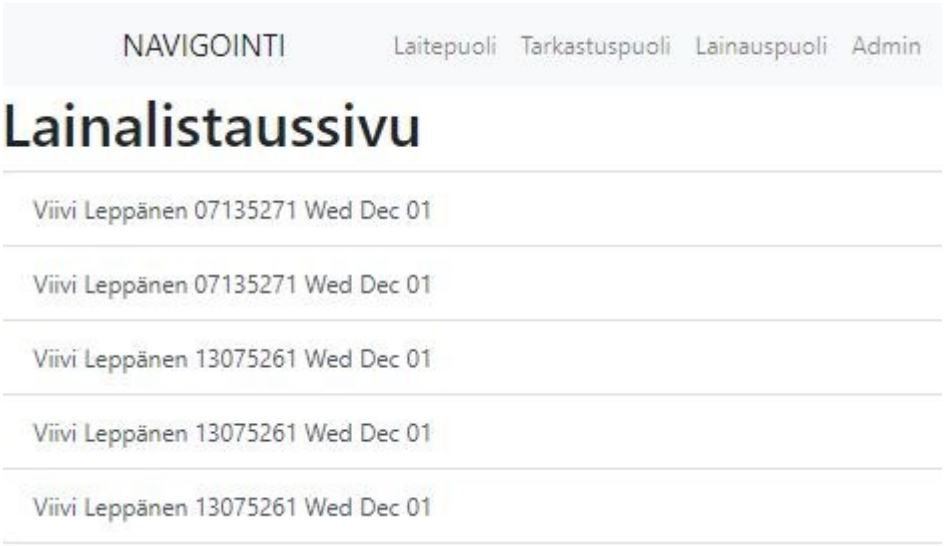
Kuva 31 – Laitesivu



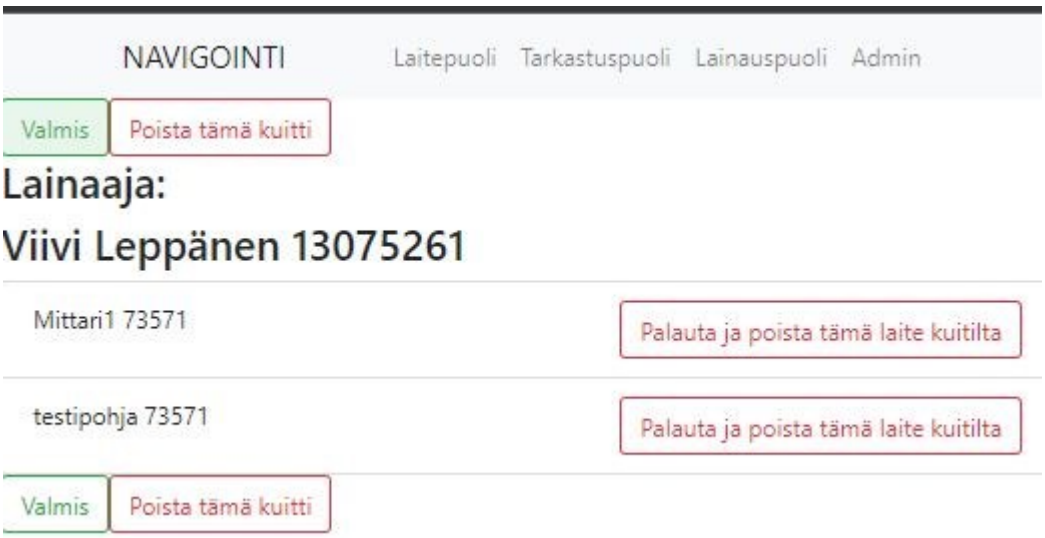
Kuva 32 - Testilaitteelle tehtyjen tarkastusten listaus

Yllä kuvattuna laitteen tietojen tarkastelunäkymä sekä sille tehtyjen tarkastusten listausnäkymä. Listausnäkymässä laitteen tarkastusta painamalla käyttöliittymä ohjaa käyttäjän tarkastuskohteiden listaukseen, josta käy ilmi mitkä kyseisen tarkastuslomakkeen tarkastuksista olivat kunnossa ja mitkä ovat jääneet puutteellisiksi.

Lainauspuolella on kaksi eri toiminnallisuutta lainausten tekemiselle ja lainausten selaamiselle. Käyttäjätietojen jälkeen alasvetovalikosta valitaan lainattavat laitteet, jonka jälkeen tiedot varmistetaan tarkastusdialogin avulla. Lainauslistauksessa puolestaan voidaan avata lainaus listauksesta, josta aukeaa lista lainatuista laitteista. Tästä näkymästä voi poistaa lainatun laitteen oman lainauksensa kohdalta.



Kuva 33 – Lainalistaussivu



Kuva 34 - Yksittäinen lainaus

Järjestelmänvalvoja-puolella käyttöliittymässä on kaksi toimintoa, osioiden muokkaus sekä laitekorttipohjien muokkaus. Osion muokkausta painettaessa aukeaa valikko, johon on listattu valmiiksi tehdyt osiot, joita klikkaamalla aukeaa aiemmin esitelty osiodialogi, jossa on esitäytetyt tiedot, joita näkymässä pääsee käsittelemään tai poistamaan sekä lisäämään. Osioiden muokkaus- sivulla on myös oma nappi uuden osion luomiselle, josta painamalla aukeaa tyhjä dialogi osion luomiselle. Tässä näkymässä tallennettu osio näkyy laitekorttipohjien luontinäkymässä vaihtoehtona muutosten tallentamisen jälkeen.

4.3 TIETOKANNAN TOTEUTUS

Tietokantana toteutettiin MongoDB Atlas Cloud Clusterissa toimiva tietokantaratkaisu, jota ei ajettu lainkaan paikallisesti. Tietokannan hallintaan sekä sen tarkasteluun käytettiin MongoDb:n omaa selaimella toimivaa käyttöliittymää, sekä testauksia suoritettiin Postman-ohjelman avulla. Tietokanta luotiin MongoDb:n luonti-ikkunassa, jonka jälkeen kantayhteys luotiin jo aiemmin kuvatulla tavalla, MongoDB:n luoman yhdistyslauseen avulla, Mongoose:a apuna käyttäen. Tietokannan ja React-sovelluksen keskinäinen kommunikaatio toteutettiin reitityksen ja väliohjelmiston kautta. Reitityksessä käytettiin Expressin Router-luokkaa, jonka avulla tehdään modulaarisia reitityksiä. Router-instanssi on myös itsessään kokonainen väliohjelmisto ja reititysjärjestelmä.

```
const deviceRouter = require('./routes/devices');
app.use('/devices', deviceRouter);
```

Kuva 35 - Palvelimelle kerrottu reitti

```
//hae kaikki
router.route('/').get((req, res) => {
    DeviceTemplate.find()
    .then(templates => res.json(templates))
    .catch(err => res.status(400).json('Error: ' + err));
});
```

Kuva 36 - Router-instanssilla laitekorttipohjien haku kannasta

```
//lisää
router.route('/add').post((req, res) => {
  const lomakkeenNimi = req.body.lomakkeenNimi;
  const lomakkeenKuvaus = req.body.lomakkeenKuvaus;
  const ominaisuudet = req.body.ominaisuudet;
  const osiot = req.body.osiot;
  const newDeviceTemplate = new DeviceTemplate({
    lomakkeenNimi,
    lomakkeenKuvaus,
    ominaisuudet,
    osiot,
  });

  newDeviceTemplate.save()
  .then(() => res.json('Device template added!'))
  .catch(err => res.status(400).json('Error: ' + err));
});
```

Kuva 37 - Router-instanssilla laitekorttipohjien lisäys kantaan

```
//päivitä
router.route('/update/:id').post((req, res) => {
  DeviceTemplate.findById(req.params.id)
    .then(template => {
      template.lomakkeenNimi = req.body.lomakkeenNimi;
      template.lomakkeenKuvaus = req.body.lomakkeenKuvaus;
      template.ominaisuudet = req.body.ominaisuudet;
      template.osiot = req.body.osiot;

      template.save()
        .then(() => res.json('Template updated!'))
        .catch(err => res.status(400).json('Error: ' + err));
    })
    .catch(err => res.status(400).json('Error: ' + err));
});
```

Kuva 38 - Router-instanssilla laitekorttipohjan päivitys kantaan

Opinnäytetyön aikana valmistuneeseen tietokantaan luotiin kokoelmat:

- käyttäjät
- lainakuitit
- laitekorttikokonaisuudet
- laitekorttipohjat
- laitteet
- osiopohjat
- tarkastukset
- tarkastuspohjat

MongoDBssä ei kokoelmiin määritellä valmiita pakollisia dokumentteja (MySQL:ssä rivit), vaan ne voidaan määritellä skeemana React-applikaation kautta. Tämä vaihe on myös mahdollista jättää välistä, jota myös tässä projektissa vaaditun joustavuuden takia pohdittiin, kuten aiemmin mainittiin. Lopullisessa ratkaisussa päädyttiin kuitenkin skeemojen tekoon, jotta ne antaisivat raamit kantaan tallennettavalle datalle.

Alla kuvina kokoelmien skeemat, joista käy ilmi käytetyt tietotyypit, niiden pakollisuus sekä mihin kokoelmaan kuvassa määritellyn kaltainen dokumentti sijoittuu tietokannassa.

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const DeviceSchema = new Schema({
  lomakkeenNimi: { type: String, required: true },
  lomakkeenKuvaus: { type: String, required: true },
  sarjanumero: { type: String, required: true },
  ominaisuudet: { type: Array, required: true },
  osiot: { type: Array, required: false },
  laitekortti: { type: String, required: true },
  lainassa: { type: String, required: false }
}, {
  timestamps: true,
  _id: true
});

//Tämä malli on yhteydessä 'laitteet'- tauluun kannassa
const Device = mongoose.model('Device', DeviceSchema, 'laitteet');

module.exports = Device;
```

Kuva 39 - Laite-skeema

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const DeviceCardTemplateSchema = new Schema({
  lomakkeenNimi: { type: String, required: true },
  lomakkeenKuvaus: { type: String, required: true },
  ominaisuudet: { type: Array, required: true },
  osiot: { type: Array, required: true },
}, {
  timestamps: true,
  _id: true
});

//Tämä malli on yhteydessä 'laitekorttipohjat'- tauluun kannassa
const DeviceCardTemplate = mongoose.model('DeviceCardTemplate', DeviceCardTemplateSchema, 'laitekorttipohjat');

module.exports = DeviceCardTemplate;
```

Kuva 40 - Laitekorttipohjan skeema

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const devCollectionSchema = new Schema({
  kokonaisuudenNimi: { type: String, required: true },
  kokonaisuudenKuvaus: { type: String, required: true },
  laitteet: { type: Array, required: true },
  ominaisuudet: { type: Array, required: true },
}, {
  timestamps: true,
  _id: true
});

const devCollection = mongoose.model('devCollection', devCollectionSchema, 'laitekorttikokonaisuudet');

module.exports = devCollection;
```

Kuva 41 - Laitekokonaisuuden skeema

```
const { ObjectID } = require('bson');
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const ReceiptSchema = new Schema({
  lainaaja: { type: Array, required: true },
  laitteet: { type: Array, required: true },
  _id: { type: ObjectID, required: true}
}, {
  timestamps: true,
  _id: false
});

//Tämä malli on yhteydessä 'lainakuitit'-tauluun kannassa
const Receipt = mongoose.model('Receipt', ReceiptSchema, 'lainakuitit');

module.exports = Receipt;
```

Kuva 42 - Lainauskuitin skeema

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const CUTemplateSchema = new Schema({
  lomakkeenNimi: { type: String, required: true },
  lomakkeenKuvaus: { type: String, required: true },
  tarkastukset: { type: Array, required: true },
}, {
  timestamps: true,
  _id: true
});

//Tämä malli on yhteydessä 'tarkastuspohjat' tauluun tietokannassa
const CUTempalte = mongoose.model('CUTempalte', CUTemplateSchema, 'tarkastuspohjat');

module.exports = CUTempalte;
```

Kuva 43 - Tarkastuspohjan skeema

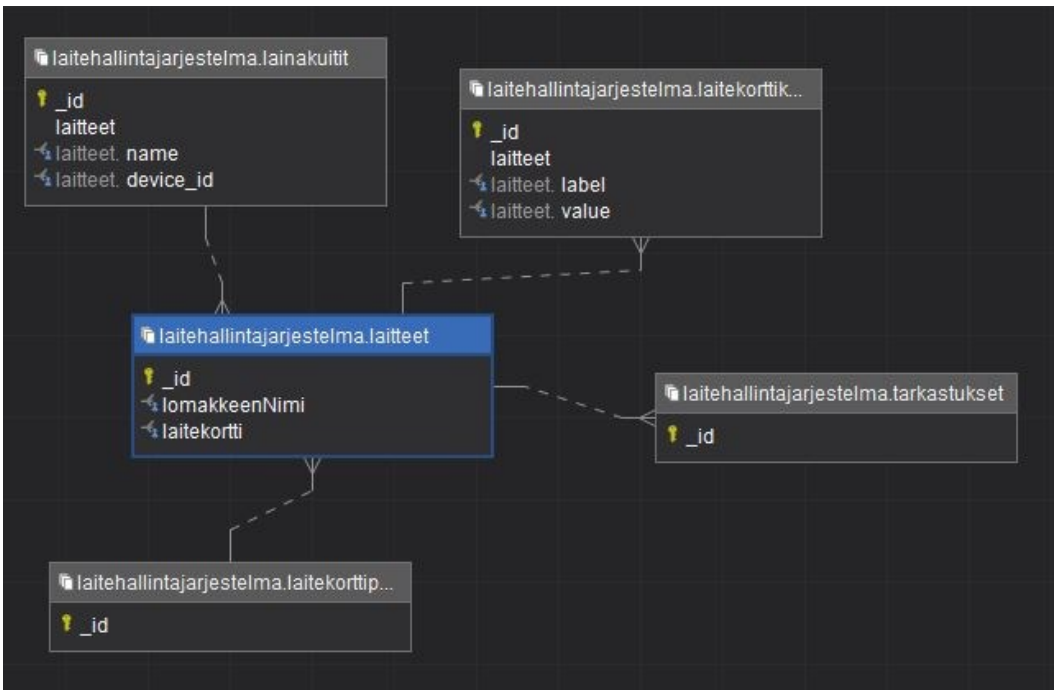
```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const CheckUpSchema = new Schema({
  lomakkeenNimi: { type: String, required: true },
  lomakkeenKuvaus: { type: String, required: true },
  tarkastukset: { type: Array, required: true },
  kunnossa: { type: Boolean, required: true },
  laite: { type: String, required: true},
}, {
  timestamps: true,
  _id: true
});

//Tämä malli on yhteydessä 'tarkastukset' tauluun tietokannassa
const CheckUp = mongoose.model('CheckUp', CheckUpSchema, 'tarkastukset');

module.exports = CheckUp;
```

Kuva 44 - Tarkastuksen skeema

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const PartitionSchema = new Schema({
  osioNimi: { type: String, required: true },
  osioKuvaus: { type: String, required: true },
  arvot: { type: Array, required: true },
}, {
  timestamps: true,
  _id: true
});

//Tämä malli on yhteydessä 'osiopohjat'- tauluun kannassa
const Partition = mongoose.model('Partition', PartitionSchema, 'osiopohjat');

module.exports = Partition;
```

Kuva 45 - Osion skeema

MongoDB:ssä relaatiot ja datan yhdistely tehdään joko upotetun lähestymistavan, tai viittauksien avulla. Tässä projektissa käytettiin niistä molempia tarpeen mukaan. Alla olevassa kuvassa on piirrettynä kaaviot projektissa käytetyistä relaatioista.



Kuva 46 - Relaatiot kuvattuna ER-mallissa

Yllä olevasta kuvasta käy ilmi kannassa käytetyt ns. relaatiot. Lainakuiteissa on käytetty upotettua relaatiota, jossa lainakuitilla olevaan laitteet-jonoon tallennetaan omaksi objektiseen laitteen nimi, sekä _id. Laitteella voi olla kerrallaan vain yksi lainakuitti mutta lainakuitilla voi olla useita laitteita, täten lainauksen päivittyessä laitteen tiedot poistetaan lainakuitilta ja siirretään uudelle lainakuitille. Laitekorttikokonaisuudella puolestaan voi olla useita laitteita, mutta yksittäinen laite voi kuulua vain yhteen laitekokonaisuuteen. Myös laitekokonaisuuksissa käytetään upotettua relaatiota, jossa laitekokonaisuuden laitteet-jonoon merkitään laitteen nimi sekä _id. Laitteen ja laitekorttipohjan välinen relaatio taas on viitattu, sillä laitteelle merkataan laitekortti-identifikaattoriksi pelkästään laitekorttipohjan _id omaan tietokenttäänsä. Laitteen ja laitekorttipohjien välinen relaatio on yhden suhde moneen siten, että laitteella voi olla vain yksi laitekorttipohja, mutta laitekorttipohjasta on voitu luoda useita laitteita. Tarkastuksen ja laitteen välisessä relaatiossa käytetään myöskin viitattua relaatiota, siten että laitteen _id tallennetaan tarkastukseen sille varattuun omaan tietokenttään. Tämäkin relaatio on yhden suhde moneen, sillä laitteella voi olla monta tarkastusta, mutta tarkastuksella voi olla vain yksi laite.

5 POHDINTA

MERN:n sanotaan olevan helposti opittavissa oleva ohjelmistokokonaisuus JavaScript:iin perustumisensa ansiosta. Jos ensin osaa perusteet JavaScript:istä, React:n oppiminen on kevyttä, sillä se on verraten yksinkertainen kieli ja oppimateriaalia löytyy paljon verkosta. MongoDB:n oppiminen taas joustavuutensa ansiosta on huomattavasti suoraviivaisempaa, kuin perinteisten relaatiomallisten tietokantojen. Node ja Express:kin ovat yksinkertaisia käytettävyydeltään, ja niihin on helppo perehtyä internetissä tarjottavan materiaalin avulla. Node:en ja Express:iin tutustuminen kannattanee käytännössä hoitaa samanaikaisesti, jolloin yhteistyön kehityksestä tulee saumattomampaa sekä yksinkertaisempaa. MERN-stack:illa ohjelmistokehitys on mielenkiintoista, mutta sanoisin sen oppimisen kuitenkin olevan aikaa vievää, vaikka perusasiat JavaScript:stä olisivatkin kunnossa. Hakukoneista materiaalin esiin kaivaminen voi koitua työlääksi prosessiksi, sillä tietoa on tarjolla jopa liikaakin.

Opinnäytetyönä valmistui laiterekisterikokonaisuus, jonka avulla on mahdollista pitää kirjaa käyttöobjekteista. Sen avulla voidaan tallentaa laitekohtaista dataa yksittäisistä laitteista sekä niiden ominaisuuksista, suorittaa kuntotarkastuksia ja pitää kirjaa näiden tilasta. Opinnäytetyössä kehitettiin sekä käyttöliittymän näkyvä puoli, keskiosan datankäsittely sekä palvelinpuolen tietokantaratkaisu NoSQL-mallin muodossa. Laitepuolen toteutus jäi osittain puutteelliseksi, mutta käytännössä pääelementit saatiin koottua projektiin. Data tallennetaan pilvessä sijaitsevaan klusteriin, kun taas käyttöliittymää hallitaan paikallisen sovelluksen kautta.

Projektin edetessä on saatu huomattavasti kattavampi käsitys full-stack ohjelmoinnista sekä sen haasteista ja ominaisuuksista. Ennen kaikkea ymmärrys MERN-stack:ista on kasvanut suuresti. Omatoiminen tiedonhaku ja dokumentaation seuraaminen on tärkeää osata ohjelmoijana ja opinnäytetyö opetti sitä suuresti. Kuitenkaan minkään projektin kehitys ei ole yksinkertaista, joten tämänkin aikana törmättiin aikataulullisiin ongelmiin, jotka olisi mitä luultavammin pystytty estämään riittävällä suunnittelulla ja aikatauluttamisella.

Opinnäytetyössä eteen tulleiden aikaongelmien takia kehitettävää jäi jossain määrin. Käyttöliittymän ulkoasu jäi yksinkertaiseksi, joten esimerkiksi siinä on parantamisen varaa. Myös laitepuolen toiminnallisuuksista esimerkiksi käyttäjäautentikointi jäi kokonaan toteutumatta, sekä RDIF-kooditus laitteiden lukemista varten esimerkiksi lainauspuolelle sekä laitteen tietojen etsimistä varten. Lisäksi laitekorttipohjien muokkaus siten, että niihin tehdyt muutokset tulevat voimaan kaikkiin laitteisiin, joiden perusteella pohja on luotu, jäi muutaman funktion toiminnallisuudesta kiinni. Vaikka sovellukseen jäi kehityskohteita, lopputulos oli mielestäni onnistunut sen puitteissa mitä ajallisesti oli mahdollista.

LÄHTEET

db-engines.com. 2021. db-engines.com. [Online] 31. 5 2021. [Viitattu: 31. 5 2021.] https://db-engines.com/en/ranking.

Elliot, Eric. 2017. JavaScript Scene. *medium.com.* [Online] Medium, 29. 12 2017. [Viitattu: 18. 8 2021.] https://medium.com/javascript-scene/top-javascript-libraries-tech-to-learn-in-2018-c38028e028e6.

guru99.com. https://www.guru99.com/. [Online] [Viitattu: 31. 5 2021.] https://www.guru99.com/what-is-mongodb.html.

IBM Cloud Education. 2020. ibm.com. *https://www.ibm.com.* [Online] IBM, 21. 12 2020. [Viitattu: 31. 05 2021.] https://www.ibm.com/cloud/learn/mongodb.

IBM. 2019. ibm.com. *IBM.* [Online] IBM, 6. 8 2019. [Viitattu: 18. 8 2021.] https://www.ibm.com/cloud/learn/nosql-databases.

Mazaika, Ken. 2021. http://blog.thefirehoseproject.com/. [Online] Firehose Project, 2021. [Viitattu: 02. 09 2021.] http://blog.thefirehoseproject.com/posts/reactjs-101/.

Mozilla. https://developer.mozilla.org. *MDN Web Docs.* [Online] [Viitattu: 31. 5 2021.] https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction.

Mozilla. MDN Web Docs. [Online] [Viitattu: 18. 08 2021.] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction.

Mustafeez, Anusheh Zohair. educative.io. [Online] [Viitattu: 29. 05 2021.] https://www.educative.io/edpresso/what-is-visual-studio-code.

Educative. educative.io. [Online] [Viitattu: 29. 05 2021.] https://www.educative.io/edpresso/what-is-visual-studio-code.

Pandit, Nitin. 2021. C#Corner. *www.c-sharpcorner.com.* [Online] 10. 02 2021. [Viitattu: 02. 09 2021.] https://www.c-sharpcorner.com/article/what-and-why-reactjs/.

React-Bootstrap. 2021. https://react-bootstrap.github.io/. [Online] 2021. [Viitattu: 31. 5 2021.] https://react-bootstrap.github.io/.

Satish Chandra Gupta. 2021. towards data sciense. *Medium.* [Online] Medium, 7. 6 2021. [Viitattu: 20. 8 20.] https://towardsdatascience.com/datastore-choices-sql-vs-nosql-database-ebec24d56106.

Schaefer, Lauren. MongoDB. *MongoDB.* [Online] [Viitattu: 20. 08 2021.] https://www.mongodb.com/nosql-explained.

Sufiyan, Taha. 2021. simplelearn.com. [Online] Simplelearn Solutions, 09. 04 2021. [Viitattu: 02. 09 2021.] https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs.

Swonigatechnology. 2021. Swonigatechnology. *https://swonigatechnology.com.* [Online] 18. 4 2021. [Viitattu: 29. 5 2021.] https://swonigatechnology.com/mern-stack/.

TechTarget. 2021. SearchITOperations. *TechTarget.* [Online] 2021. [Viitattu: 09. 09 2021.] https://searchitoperations.techtarget.com/definition/event-driven-application.

Tutorialspoint. 2021. https://www.tutorialspoint.com. [Online] 2021. [Viitattu: 31. 5 2021.] https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm.

Tutorialspoint. 2021. https://www.tutorialspoint.com/. *https://www.tutorialspoint.com/.* [Online] 2021. [Viitattu: 02. 09 2021.] https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm.

Tutorialspoint. 2021. tutorialspoint.com. [Online] 2021. [Viitattu: 09. 09 2021.] https://www.tutorialspoint.com/nodejs/nodejs_event_loop.htm.

Yamazaki, Shiori. 2020. Twilio. *Twilio Blog.* [Online] Twilio, 18. 08 2020. [Viitattu: 14. 12 2021.] https://www.twilio.com/blog/react-choose-functional-components.

Yeshwanthini. 2021. Medium.com. *www.medium.com.* [Online] 21. 4 2021. [Viitattu: 29. 5 2021.] https://medium.com/techiepedia/what-exactly-a-mern-stack-is-60c304bffbe4.