Johanna Virtanen

# Varastonhallintasovelluksen suunnittelu ja toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

1.9.2021

# Tiivistelmä


Tekijä: Johanna Virtanen

Otsikko: Varastonhallintasovelluksen suunnittelu ja toteutus
Sivumäärä: 63 sivua
Aika: 1.9.2021

Tutkinto: Insinööri (AMK)
Tutkinto-ohjelma: Tieto- ja viestintätekniikka
Ammatillinen pääaine: Ohjelmistotuotanto
Ohjaajat: Lehtori Simo Silander

---


Insinöörityön aiheena oli suunnitella ja toteuttaa varastonhallintasovellus, jota voisi hyödyntää esimerkiksi erilaisten elektroniikka- ja mekaniikkaprojektien ajan- ja resurssienhallinnassa.

Työssä määriteltiin ensin ohjelmiston toiminnallisuusvaatimukset, joiden pohjalta lähdettiin suunnittelemaan ohjelmistoarkkitehtuuria, käyttöliittymää ja tietokannan rakennetta. Toiminnallisuusvaatimukset listattiin pakollisiksi ja valinnaisiksi käyttäjätarinoiksi. Sovelluksen tärkeimmiksi vaatimuksiksi määriteltiin komponenttien tietojen lisääminen, muokkaaminen ja poistaminen tietokannasta käyttöliittymän avulla sekä tietokannasta löytyvien komponenttien tarkastelu ja järjestely erilaisten vaihtoehtojen mukaan.

Sovelluksessa käytettävien teknologioiden valitsemisessa kiinnitettiin huomiota nykyaikaisuuteen ja pitkäikäisyyteen sekä siihen, että sovelluksen jatkokehitys olisi tulevaisuudessa mahdollisimman mutkatonta. Sovelluksen toteutuksessa hyödynnettiin JavaScriptin Node.js-ajoympäristöä ja React.js-kirjastoa, Node.js:n Express.js-sovelluskehystä sekä HTML- ja CSS-kieliä. Tietokantana käytettiin MySQL-pohjaista MariaDB Server -relaatiotietokantaa.

Insinöörityön lopputuloksena oli toimiva varastonhallintasovellus. Alussa määriteltyjen pakollisten käyttäjätarinoiden toiminnallisuudet saatiin toteutettua, ja valinnaisten käyttäjätarinoiden toiminnallisuudet jätettiin odottamaan jatkokehitystä.



Avainsanat: Web-kehitys, JavaScript, Node.js, React.js, MariaDB Server, varastonhallinta

## Abstract


| Author: | Johanna Virtanen |
| --- | --- |
| Title: | Designing and Developing Stock Management Application |
| Number of Pages: | 63 pages |
| Date: | 1 September 2021 |
| | |
| Degree: | Bachelor of Engineering |
| Degree Programme: | Information and Communication Technology |
| Professional Major: | Software Engineering |
| Supervisors: | Simo Silander, Senior Lecturer |


---


The goal of the study was to design and develop a stock management application for saving time and resources in miscellaneous electronical and mechanical projects.

The study was started by defining the functional requirements of the software. The planning of the software architecture, the user interface and the database structure were commenced based on the established requirements. The functional requirements were described as mandatory and optional user stories. The most important functional requirements of the application were defined to be the capability to insert, update and remove components from the database using the user interface and to inspect and sort components existing in the database.

The software technologies were selected in accordance to their modernity and validity, not only in present but also in the future. The software was developed using Node.js (a JavaScript Runtime), React.js (a JavaScript library), Express (a Node.js web application framework), HTML and CSS. The MySQL based relational database MariaDB Server was selected to be used as the database solution.

As a result, a functioning application was designed and developed. The functionalities of the defined mandatory user stories were carried out as planned, and the functionalities of the optional user stories were put in hold for further development.



Keywords: Web Development, JavaScript, Node.js, React.js, MariaDB Server, Stock Management

# Sisällys

Lyhenteet

1 Johdanto 1

2 Lähtökohta ja tavoitteet 2

2.1 Asiakastarpeet 4
2.2 Suunnitelman laatiminen 4

3 Ohjelmiston arkkitehtuuri ja käytetyt teknologiat 6

3.1 Ohjelmiston rakenne ja toimintalogiikka 6
3.2 React.js 9
3.3 Node.js 11

4 Ohjelmiston toiminnallisuus 16

4.1 Front-end 16
4.1.1 Reititys 17
4.1.2 Kategoriat 20
4.1.3 Tuotteet 24
4.1.4 Yksittäinen tuote 33
4.1.5 Tietojen lisääminen, muokkaaminen ja poistaminen 37
4.2 Back-end 43
4.3 Tietokantaratkaisu 46

5 Käyttöliittymä 49

5.1 Tietojen tarkastelu 50
5.2 Tietojen lisääminen, muokkaaminen ja poistaminen 54
5.3 Muut toiminnot 58

6 Jatkokehityskohteet 59

7 Yhteenveto 61

Lähteet 62

## Lyhenteet

API: *Application Programming Interface.* Ohjelmointirajapinta, jonka avulla ohjelmat voivat vaihtaa tietoa keskenään.

CLI: *Command Line Interface.* Komentoliittymä, jossa voi kirjoittaa komentoja.

CSS: *Cascade Style Sheet.* Tyyliohjeiden määrittelyyn tarkoitettu ohjelmointikieli.

C#: *C Sharp.* Ohjelmointikieli, jota voidaan käyttää esimerkiksi työpöytä- ja web-sovellusten kehityksessä.

DOM: *Document Object Model*. Dokumenttioliomalli, joka kuvaa dokumentin rakenteen puuna.

HTML: *Hyper Text Markup Language.* Web-selaimessa esitettäväksi tarkoitettujen dokumenttien merkintäkieli.

HTTP: *Hypertext Transfer Protocol.* Protokolla, jota käytetään WWW-palvelinten ja selainten tiedonsiirrossa.

JSON: *JavaScript Object Notation.* Tiedostomuoto, joka sisältää dataa avain-arvo-pareina.

MVC: *Model-View-Controller*. Ohjelmistoarkkitehtuuri, joka erittelee ohjelmiston logiikan kolmeen osa-alueeseen.

npm: *Node Package Manager.* Paketinhallintajärjestelmä JavaScript-ohjelmointikielelle.

PHP: *PHP: Hypertext Processor.* Ohjelmointikieli, jota käytetään usein web-kehityksessä.

REST: *Representational State Transfer*. Ohjelmointirajapintojen toteuttamiseen tarkoitettu HTTP-protokollaan perustuva arkkitehtuurimalli.

SPA: *Single Page Application*. Web-sivu tai sovellus, joka päivittää ruudulla esitettävän tiedon aina aiemmin esitetyn tiedon päälle sen sijaan, että lataisi kokonaan uuden sivun tiedon esittämistä varten.

SQL: *Structured Query Language.* Kieli, jota käytetään tiedon tallettamiseen, hakemiseen ja poistamiseen relaatiotietokannoista.

URL: *Uniform Resource Locator*. Datan hakemiseen tarvittavan tiedon sisältämä merkkijono, useimmiten verkko-osoite.

VDOM: *Virtual DOM*. Reactin hyödyntämä kevyempi JavaScript-esitysmuoto DOMista.

# 1 Johdanto

Kaikenlaisissa projekteissa suunnittelu ja järjestelmällisyys helpottavat työntekoa ja säästävät aikaa. Kun puhutaan elektroniikka- tai mekaniikkaprojekteista, liittyy työhön usein kymmeniä, ellei satoja erilaisia komponentteja, joiden ominaisuuksien ja eroavaisuuksien havainnointi ja ulkoa muistaminen voi olla haastavaa. Lisäksi komponenttien ja projektien määrän kasvaessa on vaikeaa muistaa, millaisia tarvikkeita mihinkin projektiin tarvitaan ja kuinka paljon.

Tämän insinöörityön tavoitteena on suunnitella ja toteuttaa varastonhallinta-sovellus, jota voi hyödyntää erilaisten projektien varastonhallintatarpeisiin. Sovelluksen tärkeimpänä ominaisuutena on komponenttien tietojen lisääminen, muokkaaminen ja poistaminen tietokannasta käyttöliittymän avulla. Lisäksi käyttöliittymässä tulee voida tarkastella tietokannan sisältämiä komponentteja ja esimerkiksi järjestellä niitä erilaisten vaihtoehtojen perusteella. Ohjelmisto tulee suunnitella siten, että vähimmäisvaatimusten päälle on tulevaisuudessa helppoa tehdä jatkokehitystyötä. Insinöörityön teknologiat valitaan sen perusteella, että ne olisivat modulaarisia ja mahdollisimman moderneja, mutta valideja myös tulevaisuudessa.

Insinöörityössä toteutettava ohjelmisto on toteutettu full stackina, eli se pitää sisällään palvelinpuolen (back-end), käyttöliittymäpuolen (front-end) sekä tietokantaratkaisun. Ohjelmiston palvelinpuoli on kirjoitettu kokonaisuudessaan JavaScriptillä. Käyttöliittymän toteutuskielinä ovat JavaScript, HTML sekä CSS. Lisäksi ohjelmistossa hyödynnetään esimerkiksi JavaScriptin React.js-kirjastoa ja JavaScript-koodin suorittamisen palvelimella mahdollistavaa Node.js-ajoympäristöä. Tässä raportissa käydään läpi insinöörityön lähtökohta ja tavoitteet sekä ohjelmiston toteutuksessa käytetyt teknologiat ja arkkitehtuuri. Lisäksi paneudutaan ohjelmiston toiminallisuuteen koodiesimerkkien avulla ja esitellään ohjelmiston käyttöliittymä sekä tietokantaratkaisu. Lopuksi pohditaan ohjelmiston jatkokehitystarpeita ja -mahdollisuuksia.

# 2 Lähtökohta ja tavoitteet

Insinöörityö sai alkunsa elektroniikka- ja mekaniikkaharrastelijan (jatkossa ”työn tilaaja”) varastonhallintavaikeuksista. Intohimoisessa harrastuksessa erilaista pienrautaa ja työkaluja kertyy vuosien varrella vaihtelevien projektien myötä niin paljon, että lopulta työn tilaajan on vaikea muistaa, mitä tuotteita hän jo omistaa ja kuinka monta kappaletta.

Ennen kuin ohjelmistoa alettiin rakentaa, määriteltiin työn tärkeimmät tavoitteet miettimällä erilaisia käyttötilanteita eli käyttäjätarinoita (engl. user stories) [1]. Käyttäjätarinoiden avulla on helppoa hahmottaa, millaisia toiminnallisuuksia ohjelmistolta vaaditaan.

Käyttäjätarinat jaettiin pakollisiin käyttäjätarinoihin ja valinnaisiin käyttäjätarinoihin. Pakolliset käyttäjätarinat ovat niitä, joita ilman sovelluksen käytettävyys kärsii huomattavasti ja valinnaiset käyttäjätarinat niitä, jotka luovat sovellukselle lisäarvoa ja parantavat käyttömukavuutta, mutta joiden toteuttaminen voidaan jättää odottamaan jatkokehitystä. Lopulliset käyttäjätarinat on esitetty taulukossa 1.

Taulukko 1. Ohjelmiston käyttäjätarinat kuvaavat vaadittuja toiminnallisuuksia.

| **Käyttäjätarina** | **Prioriteetti** |
|---|---|
| Käyttäjänä haluan kirjautua sovellukseen sisään, jotta voin käyttää sovelluksen toimintoja. | Pakollinen |
| Käyttäjänä haluan kirjautua sovelluksesta ulos. | Pakollinen |
| Käyttäjänä haluan nähdä kaikki tietokantaan lisätyt kategoriat kuvakkeina. | Pakollinen |
| Käyttäjänä haluan nähdä kaikki tietokantaan lisätyt kategoriat listana. | Pakollinen |
| Käyttäjänä haluan nähdä kaikki tietokantaan lisätyt hyllyt. | Pakollinen |
| Käyttäjänä haluan nähdä kaikki tietokantaan lisätyt tuotteet. | Pakollinen |

| Käyttäjänä haluan hakea tietokannasta tiettyä kategoriaa nimellä, jotta pääsen helposti tarkastelemaan ja muokkaamaan kyseisen kategorian tietoja. | Pakollinen |
|---|---|
| Käyttäjänä haluan hakea tietokannasta tiettyä tuotetta nimellä, jotta pääsen helposti tarkastelemaan ja muokkaamaan kyseisen tuotteen tietoja. | Pakollinen |
| Käyttäjänä haluan lisätä tietokantaan kategorioita. | Pakollinen |
| Käyttäjänä haluan lisätä tietokantaan hyllyjä. | Pakollinen |
| Käyttäjänä haluan lisätä tietokantaan tuotteita. | Pakollinen |
| Käyttäjänä haluan muokata tietokannassa olevia kategorioita. | Pakollinen |
| Käyttäjänä haluan muokata tietokannassa olevia hyllyjä. | Pakollinen |
| Käyttäjänä haluan muokata tietokannassa olevia tuotteita. | Pakollinen |
| Käyttäjänä haluan poistaa kategorioita tietokannasta. | Pakollinen |
| Käyttäjänä haluan poistaa hyllyjä tietokannasta. | Pakollinen |
| Käyttäjänä haluan poistaa tuotteita tietokannasta. | Pakollinen |
| Käyttäjänä haluan palauttaa unohtuneen salasanani. | Valinnainen |
| Käyttäjänä haluan saada yhteenvedon loppumaisillaan olevista tuotteista sähköpostiini. | Valinnainen |
| Käyttäjänä haluan, että voin lisätä tuotteita keräilylistalle. | Valinnainen |
| Käyttäjänä haluan, että voin yhtä painiketta painamalla vähentää tietokannasta kaikkien keräilylistalle lisäämieni tuotteiden kappalemääriä. | Valinnainen |
| Uutena käyttäjänä haluan luoda itselleni tunnukset sovellukseen kirjautumista varten. | Valinnainen |
| Admin-tunnusten haltijana haluan, että vain minä voin poistaa kokonaisia kategorioita tietokannasta. | Valinnainen |
| Admin-tunnusten haltijana haluan, että vain minä voin poistaa hyllyjä tietokannasta. | Valinnainen |
| Admin-tunnusten haltijana haluan, että vain minä voin poistaa kokonaisia tuotteita tietokannasta. | Valinnainen |

Käyttäjätarinat jakaantuivat kahden eri roolin mukaisesti käyttäjän toiminnallisuuksiin ja ylläpitäjän eli admin-tunnusten haltijan toiminnallisuuksiin. Lisäksi yhdessä käyttäjätarinassa roolina on ”uusi käyttäjä”, joka voidaan laskea kuuluvaksi ”käyttäjä”-rooliin. Työn tavoitteena oli siis luoda toimiva varastonhallintaohjelmisto toteuttamalla vähintään pakollisten käyttäjätarinoiden toiminnallisuudet.

## 2.1 Asiakastarpeet

Työn tilaaja koki, että uuden projektin aloittaminen tai vanhan, tauolla olleen projektin jatkaminen olisi mielekkäämpää, mikäli tarvittavien osasten etsimiseen ja selvittelyyn ei tarvitsisi käyttää niin paljon aikaa ja vaivaa. Esimerkiksi uusien elektroniikkaosien turha ostelu rasittaa sekä lompakkoa että ympäristöä. Työn tilaaja säästäisi myös merkittävästi aikaa voimalla helposti tarkistaa, tarvitaanko uutta projektia varten esimerkiksi 12 millimetriä pitkää ja halkaisijaltaan 5 millimetriä paksua ristipääruuvia 20 kappaletta lisää vai löytyykö tuotetta jo varastosta tarvittava määrä. Lisäksi työn tilaajan tulisi saada nopeasti tieto tuotteen sijainnista hyllykköjärjestelmässä.

Taulukossa 1 esitetyt käyttäjätarinat kuvaavat asiakastarpeita. Käyttäjätarinat laadittiin yhdessä työn tilaajan kanssa. Käyttäjätarinat hieman muovaantuivat ja niiden lukumäärä lisääntyi työn edetessä, kun työn tilaajalle hahmottui tarkemmin, minkälaisia toiminnallisuuksia hän sovellukselta toivoo.

## 2.2 Suunnitelman laatiminen

Ennen tarkkaa suunnitelmaa tarvittavasta tietokannasta ja käyttöliittymäkomponenteista piti selvittää, millaisia erilaisia ominaisuuksia ja tärkeitä mittoja eri tuotteilla ja komponenteilla mahdollisesti on. Melko varhaisessa vaiheessa tuli todetuksi, että on kovin vaikeaa tehdä kattavaa listaa erilaisista ominaisuuksista, joita tuotteisiin halutaan liittää. Esimerkiksi elektroniikkaosien ominaisuudet eroavat sekä toisistaan että mekaniikkakomponenteista oikeastaan täydellisesti, sillä ruuvin tärkeimpiin ominaisuuksiin kuuluvat pituus, paksuus ja kannan

tyyppi, kun taas vaikkapa valovastuksen suhteen tärkeimpiä tietoja ovat vastusarvo, tehonkesto ja maksimijännite. Tämän vuoksi päädyttiin siihen, että jokaisella tuotteella on oma vapaamuotoinen lisätietokenttä, johon voi sisällyttää tietoja ominaisuuksiin liittyen. Lisäksi päätettiin, että jokaisen tuotteen tulee olla liitettynä johonkin hyllyyn sekä johonkin kategoriaan. Tuotteen erilaisten ominaisuuksien lisäksi käytiin yhdessä työn tilaajan kanssa läpi myös se, millaisia tietoja tietokantaan tulisi pystyä tallentamaan yksittäisestä hyllystä tai kategoriasta.

Käyttäjätarinoiden perusteella lähdettiin suunnittelemaan käyttöliittymää. Pohdittiin, millaisia näkymiä käyttöliittymän tulisi sisältää, jotta käyttäjätarinoiden toiminnallisuudet saataisiin toteutettua. Todettiin, että käyttöliittymästä tulee rakentaa sellainen, että kaikki pakollisten käyttäjätarinoiden toiminnallisuudet ovat toteutettavissa mahdollisimman vaivattomasti. Lisäksi navigoinnin käyttöliittymässä tulisi olla sujuvaa ja nopeaa.

# 3 Ohjelmiston arkkitehtuuri ja käytetyt teknologiat

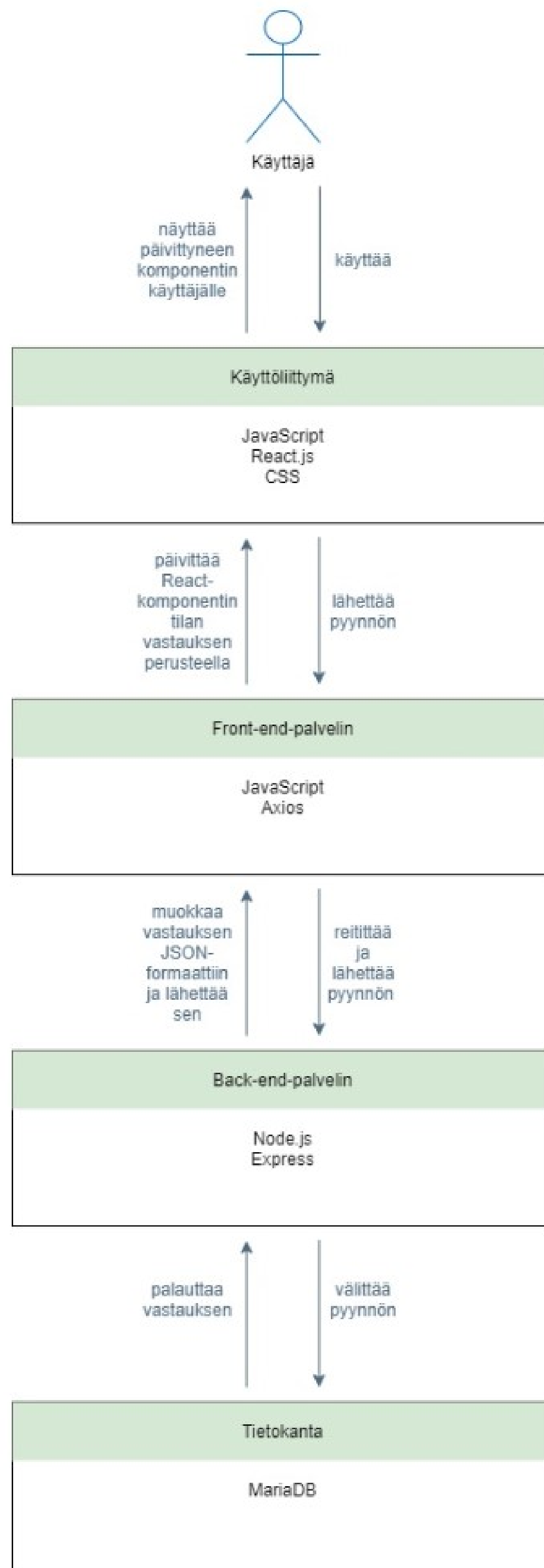
Web-sovellusarkkitehtuuri kuvaa sitä, miten ohjelmiston käyttöliittymäpuoli ja palvelinpuoli kommunikoivat keskenään ja millaista väliohjelmistoa (engl. middleware) niiden välillä käytetään [2].

Käyttöliittymäpuoli suoritetaan selaimessa ja on usein kirjoitettu HTML:n, CSS:n ja JavaScriptin avulla. Palvelinpuolen toteutuksessa voidaan käyttää ohjelmointikielenä esimerkiksi PHP:tä, Pythonia, Rubya, C#:a tai JavaScriptiä.

Väliohjelmisto määrittää sen, miten tieto kulkee sovelluksen käyttöliittymäpuolen ja palvelinpuolen välillä. Väliohjelmisto voi esimerkiksi määritellä API-kutsut sekä sen, miten autentikointi tapahtuu [3]. Tässä insinöörityössä väliohjelmistoksi voidaan laskea esimerkiksi Node.js:n Express.js-sovelluskehys. Web-sovellusten tiedonsiirrossa käytetään HTTP-protokollaa, joka mahdollistaa viestien välittämisen käyttöliittymien ja palvelinten välillä [4].

## 3.1 Ohjelmiston rakenne ja toimintalogiikka

Tässä insinöörityössä toteutettu sovellus sisältää käyttöliittymän, front-end-palvelimen, back-end-palvelimen sekä tietokannan. Sovelluksen arkkitehtuuri on esitetty kuvassa 1.




Kuva 1. Ohjelmiston arkkitehtuuri.

Kuvassa 1 esitetään, miten sovelluksen käyttäjä lähettää käyttöliittymän avulla HTTP-pyyntöjä, jotka kulkevat front-end- ja back-end-palvelinten kautta tietokannalle. Tietokanta palauttaa vastauksen, ja käyttäjä näkee päivittyneen tiedon ruudullaan.

Käyttöliittymä on tässä ohjelmistossa yhden sivun sovelluksena (SPA) toteutettu HTML-dokumentti. DOM eli *Document Object Model* kuvaa HTML-dokumentin Node-olioina ja täten mahdollistaa HTML:n muokkaamisen esimerkiksi JavaScriptin avulla [5]. Koska DOMin päivittäminen on hidasta ja vie paljon resursseja, React hyödyntää DOMista kevyempää, JavaScript-kielistä esitysmuotoa eli VDOMia. React vertaa VDOMia ja DOMia keskenään, ja kun käyttöliittymästä lähtee kutsu tietokantaan ja tietokanta palauttaa vastauksen, React päivittää ensin VDOMin. Tämän jälkeen React varmistaa, että VDOM ja DOM vastaavat toisiaan päivittämällä DOMin tarpeen mukaan. [6.]

Käyttöliittymästä lähtevä HTTP-pyyntö (engl. HTTP request) kulkee ensin front-end-palvelimelle. Pyyntö voi joko hakea tietokannasta tietoa, tallentaa tai päivittää sinne uutta tietoa tai poistaa tietoa.  Promise-pohjainen Axios-kirjasto helpottaa HTTP-pyyntöjen ja -vastausten käsittelyä ja toimii myös vanhemmilla selaimilla [7].

Back-end-palvelin vastaanottaa front-end-palvelimen lähettämän kutsun ja käyttää Node.js:n Express-sovelluskehystä pyynnön välittämiseen tietokannalle. Kysely tietokannalle tehdään SQL-kyselykielen avulla. MariaDB:n Node.js Connector -API mahdollistaa SQL-muotoisen kyselyn lähettämisen.

Tietokantakysely palauttaa back-end-palvelimen kautta front-end-palvelimelle promise-olion, joka sisältää tiedon asynkronisen toiminnon tilasta. Promise-olio mahdollistaa HTTP-pyynnön loppuun suoriutumisen seuraamisen front-end-palvelimella (esimerkkikoodi 1).

```
axios.post(
     '/new-category',
     {
          name: newCategoryName,
          url: newImageUrl
     }
)
.then(res => {
     console.log(res.status);
})
```

Esimerkkikoodi 1. Then-metodin sisältämä koodi suoritetaan, kun *post*-pyyntö on joko onnistunut tai epäonnistunut.

Esimerkkikoodissa 1 tulostetaan back-end-palvelimen palauttaman promise-olion status konsoliin. Promise-olion statusta voidaan hyödyntää esimerkiksi toiminnon onnistumisen ilmoittamiseen käyttäjälle.

Promise-olio voi palauttaa paljon muutakin kuin tiedon HTTP-pyynnön onnistumisesta. Mikäli tietokannasta olisi haettu tieto HTTP:n *get*-metodilla, olisi promise-olio voinut palauttaa esimerkiksi tietokannasta löytyneet kategoriat. Front-end-palvelimella nämä olisi voitu tallettaa Reactin useState-koukun avulla tilaolioon, jolloin VDOM ja sitä myöten DOM olisivat tulleet päivitetyiksi käyttäjän ruudulle.

Kun ohjelman front-end ja back-end on kirjoitettu JavaScriptillä, kehittäjän työ saattaa helpottua, kun kehityskieltä ei tarvitse vaihtaa ja syntakseja opetella ja muistaa kahden eri ohjelmointikielen suhteen. Toisaalta tämä myös hämärtää selkeää rajanvetoa käyttöliittymäpuolen ja palvelinpuolen koodien välille. Ohjelman tiedostopuu onkin rakennettu siksi siten, että käyttöliittymäpuolen koodi ja palvelinpuolen koodi sijaitsevat eri kansioissa ja molemmilla on omat Git-versionhallintansa.

## 3.2 React.js

Ohjelman front-end eli käyttöliittymäpuoli (MVC-mallissa view eli näkymä) on toteutettu React.js-kirjastolla. Tässä sovelluksessa React.js-projekti käyttää Node.js-ajoympäristöä. React.js eli React on Facebookin kehittämä ja ylläpitämä JavaScript-kirjasto [8]. Ensimmäinen versio Reactista julkaistiin vuonna

2013 [9]. Vuonna 2019 Reactin käyttäjien joukkoon lukeutui monia tunnettuja yrityksiä, kuten Facebook, Netflix, Uber, Instagram, Twitter, Snapchat, Tinder ja LinkedIn [10].

React-sovelluksen arkkitehtuuri sisältää olennaisena osana komponentteja, jotka muodostavat oman hierarkiansa. React-komponentit ovat uudelleenkäytettäviä koodin osia, jotka voivat olla luokka- tai funktiotyyppisiä. Reactin komponentit palvelevat samaa tarkoitusta kuin JavaScriptin funktiot, mutta erotuksena JavaScriptin funktioihin React-komponentit toimivat erillään muusta koodista ja palauttavat HTML:ää [11]. Yksittäinen React-komponentti yleensä esittää ruudulla jonkin yksittäisen käyttöliittymän osasen. Poikkeuksena tästä on *App*-komponentti, joka pitää sisällään kaikki muut React-komponentit.

Luokkakomponentit eroavat funktionaalisista komponenteista pääasiassa syntaksin osalta. Aiemmissa React-versioissa tilaolion (engl. state object) ja elinkaarimetodien (engl. lifecycle methods) käyttö oli mahdollista vain luokkakomponenttien yhteydessä, mutta versiossa 16.8 julkaistiin koukut (engl. hooks), minkä ansiosta tilan ja elinkaarimetodien käyttö on mahdollista myös funktionaalisten komponenttien kanssa.

Reactin koukkujen tilaolioissa voidaan säilyttää muuttujien arvoja, ja elinkaarimetodien avulla komponentti voidaan uudelleenrenderöidä tilan sisältämien arvojen muuttuessa, jolloin käyttöliittymän näytöllä esittämät asiat päivittyvät. Tila voi sisältää kaikentyyppisiä muuttujia (esimerkiksi kokonaislukuja, totuusarvoja tai merkkijonoja).

Elinkaarimetodit voivat keskittyä kolmeen eri tilanteeseen:

- siihen, kun komponentti renderöidään
- siihen, kun komponenttia päivitetään
- siihen, kun komponenttia ollaan poistamassa näkyviltä.

Reactin versio 16.8 muutti elinkaarimetodeja siten, että esimerkiksi *UseEffect*-koukku korvaa aiempien React-versioiden *componentDidMount*-, *component-DidUpdate*- ja *componentWillUnmount*-metodit. Tämä yksinkertaistaa ja nopeuttaa kehittäjän työtä ja vähentää tarvittavan koodin määrää.

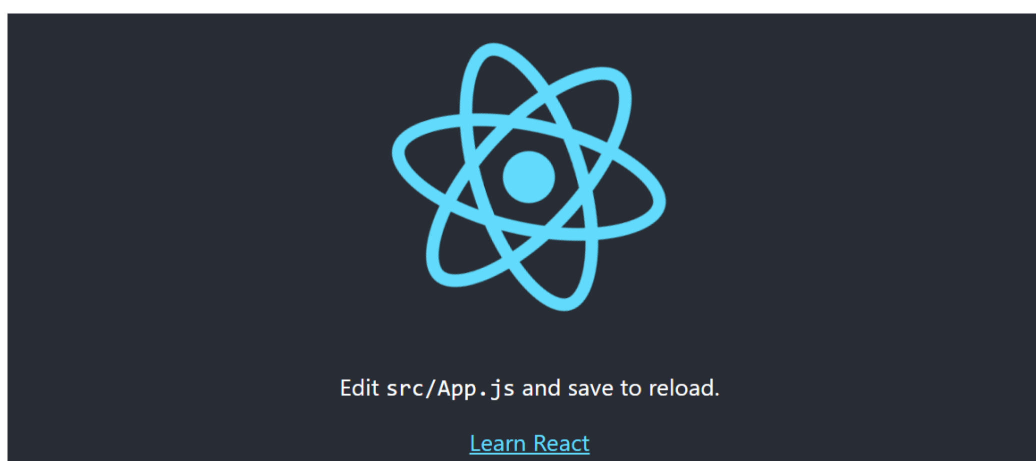
## 3.3 Node.js

Sekä sovelluksen back-end-palvelimen että front-end-palvelimen ajoympäristönä toimii vuonna 2009 kehitetty Node.js. Node.js mahdollistaa JavaScript-koodin asynkronisen suorittamisen palvelimella Google Chromen V8-JavaScript-moottorin avulla. Node.js:n asynkronisuuden ansiosta ohjelma ei jää odottamaan vastauksia APIlta vaan jatkaa ohjelman muiden osien suorittamista, mikä nopeuttaa sovelluksen toimintaa. [12.]

Ohjelmassa on käytetty myös Expressiä, Node.js:n suosituinta web-sovelluskehystä. Express.js helpottaa ohjelman jaottelua MVC-mallin mukaisiin osiin ja tarjoaa kehittäjälle oikotien API-kutsujen ja -vastausten kirjoittamiseen sekä reititykseen (engl. routing) eli siihen, mikä URL-osoite on linkitetty mihinkin tietokantakutsuun.

Node.js:n Node Package Manager (npm) sisältää komentoliittymän (CLI), jonka avulla sovellukseen on helppo asentaa erilaisia paketteja ja moduuleja [13]. Npm on JavaScript-kielinen paketinhallintajärjestelmä, jonka tarkoituksena on helpottaa pakattujen ohjelmamoduulien jakamista muiden ohjelmistokehittäjien kanssa. Useimmat ohjelmistokehityksessä eteen tulevat ongelmat ovat sellaisia, joihin joku muu on jo aiemmin törmännyt ja kehittänyt ratkaisun. On niin sanotusti turhaa keksiä pyörää uudelleen, jos toimiva pyörä on jo olemassa. Npm myös ylläpitää tietoa asennettujen pakettien ja moduulien versioinnista, ja tarjoaa kehittäjälle nopean ja helpon tavan päivittää vanhentuneet versiot uusimpiin.

Esimerkiksi ohjelmistokehittäjän alkaessa kehittää uutta React-pohjaista käyttöliittymää tarvitsee kehittäjän kirjoittaa vain komento ”npx create-react-app <projektin-nimi>”. Komento luo projektille automaattisesti neljä tiedostoa (*.gitignore*, *package-lock.json*, *package.json* ja *README.md)* sekä kolme kansiota (*node_modules*, *public* ja *src*). Nämä automaattisesti luodut tiedostot ja kansiot sisältävät kaiken tarvittavan siihen, että ohjelmiston front-end-palvelimen voi jo nyt käynnistää komennolla ”npm start”. Kehittäjän avatessa ”http://localhost:3000”-URL-osoitteen selaimessaan päivittyy ruudulle React-sovelluksen alkunäkymä (kuva 2).




Kuva 2. Luodun React-sovelluksen alkunäkymä.

Kuten kuvasta 2 voidaan todeta, React-sovelluksen alkunäkymä ohjeistaa kehittäjää editoimaan tiedostoa *App.js* näkymän muokkaamiseksi. Kuten aiemmin mainittiin, *App*-komponentti pitää sisällään kaikki muut React-komponentit. Kehittäjä voi joko rakentaa omia React-komponentteja tai ladata niitä julkisesta npm-arkistosta (engl. npm public registry). Npm käyttää oletusarvoisesti npm-arkistoa pakettien ja moduulien lataamiseen [14]. *BrowserRouter*- ja *Router*-moduulit tarjoavat ratkaisun URL-osoitteiden reitittämiseen eri komponenteille, jolloin kehittäjän ei tarvitse keskittyä luomaan ohjelmalogiikkaa tältä osin itse.

Uusia moduuleja on helppoa ladata julkisesta npm-arkistosta ”npm install” -komennon avulla. Kun kehittäjä haluaa ladata projektiinsa tietyn moduulin, annetaan sen nimi ”npm install” -komennolle parametrina, esimerkiksi “npm install

axios”. Uudet moduulit tallentuvat projektin *package.json*-tiedostoon riippuvuuksiksi (engl. dependencies). [15.] Uuden henkilön tullessa mukaan ohjelmistokehitykseen *package.json*-tiedoston tarkoituksena on helpottaa projektin käyttöönottoa – uusi käyttäjä saa kaikki jo projektille määritetyt riippuvuudet käyttöönsä ajamalla komennon ”npm install” [16].

Kuten jo aiemmin mainittiin, ”npm start”-komennolla käyttäjä voi käynnistää sovelluksen front-end-palvelimen. Myös back-end-palvelimen alustus ja käynnistys onnistuvat npm:n avulla. Kehittäjä voi alustaa uuden Node.js-projektin ”npm init” -komennolla. Komento luo *packages.json*-tiedoston, ja kehittäjä voi luoda nyt *server.js*-tiedoston, jonka sisällä määritellään sovellus käyttämään Express.js-sovelluskehystä. Yksinkertaisin Express.js-sovellus on kuvattu esimerkkikoodissa 2.

```
const express = require(‘express’);
const app = express();
const port = 3003;
app.get(‘/’, (req, res) => {
     res.sendStatus(404);
});
app.listen(port, () => {
     console.log(‘Kuunnellaan porttia ${port}’);
});
```

Esimerkkikoodi 2. Express-sovellus yksinkertaisimmillaan.

Esimerkkikoodissa 2 luodaan Express-sovellus, joka kuuntelee porttia 3003. Käyttäjän navigoidessa URL-osoitteeseen ”http://localhost:3003/” palauttaa sovellus statuksen 404. Esimerkkikoodin 2 neljännen rivin *app* on Expressin ilmentymä. *Get* on HTTP-pyynnön metodi, ja sen jälkeen esitetään puolilainausmerkeissä URL-osoitteen polku. Mikäli front-end-palvelimelta saapuva HTTP-pyyntö vastaa Express-endpointia, suoritetaan määritelty funktio eli esimerkkikoodin 2 tapauksessa palautetaan front-end-palvelimelle status 404. HTTP-pyynnön metodeja on yhdeksää erilaista tyyppiä:

- get (tiedon hakemiseen)
- post (tiedon tallettamiseen)
- put (tiedon korvaamiseen)

- delete (tiedon poistamiseen)
- patch (tiedon muokkaamiseen)
- head (otsikkotietojen hakemiseen)
- options (viestintävaihtoehtojen määrittelyyn)
- trace (testaukseen ja virheiden määrittelyyn).

Näistä yhdeksästä pyynnöstä kaikki paitsi *trace* ovat hyvin eri selainten kanssa yhteensopivia [17]. Mikäli tietokantaan talletetaan *post*-metodilla uutta dataa, tarvitsee kyselylle määrittää myös parametreina talletettavat tiedot (esimerkkikoodi 3).

```
app.post('/new-category', (req, res) => {
     pool.getConnection()
     .then(conn => {
          return conn.query(
               "INSERT INTO categories (name, imageURL) VALUES (?,?);",
               [req.body.name, req.body.imageURL]
          )
     });
});
```

Esimerkkikoodi 3. Back-end-palvelin välittää front-end-palvelimelta tulleen *post*-muotoisen HTTP-pyynnön tietokannalle.

Esimerkkikoodissa 3 *categories*-tauluun lisätään uusi rivi, joka sisältää arvot sarakkeille *name* ja *imageURL*. Nämä tiedot saadaan front-end-palvelimen lähettämästä HTTP-pyynnöstä, ja ne ovat alun perin käyttäjän syötteitä käyttöliittymässä. Myös datan poistamiseen, muokkaamiseen tai taulun tietyn rivin hakemiseen tietokannasta tarvitaan parametreja front-end-palvelimelta, kun taas esimerkiksi tietyn taulun, vaikkapa kategorian, kaikkien rivien hakeminen tapahtuu kyselylauseella ”SELECT * FROM categories”. Tällöin kyselyn suorittamiseksi ei tarvita parametreja front-end-palvelimelta.

”Npm init” -komennon luoma *packages.*json-tiedosto sisältää JSON-muodossa erilaista Node.js-projektiin liittyvää dataa. Mikäli *packages.json*-tiedoston “scripts”-objektissa ei määritellä “start”-avainta, ajaa npm automaattisesti ”node server.js”-komennon. Front-endin puolella React-sovelluksen alustaminen ge-

neroi automaattisesti *packages.json*-tiedoston *scripts*-objektille tarvittavat avaimet, joten ”npm start” -komento toimii sekä palvelinpuolen että käyttöliittymäpuolen käynnistämisessä.

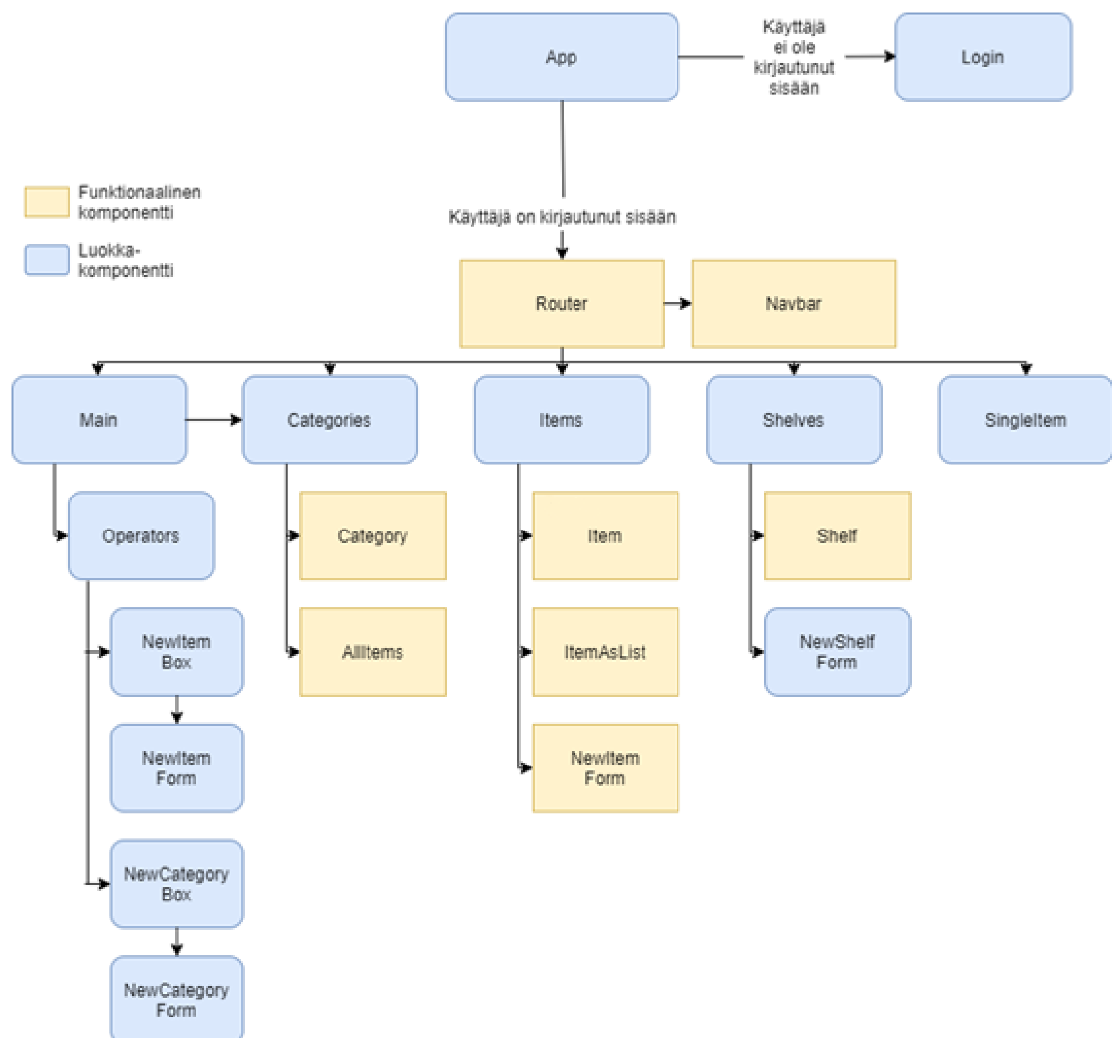
Front-end-palvelimella toimivan React-sovelluksen ja back-end-palvelimella toimivan Express-sovelluksen avulla on mahdollista luoda toimiva yhteys käyttöliittymän ja tietokannan välille. Sekä back-end- että front-end-palvelinten ollessa käynnissä sovelluksesta saadaan lopulta toimiva kokonaisuus käyttöliittymineen ja tietokantoineen. Koska Node.js on avoimen lähdekoodin ohjelma, tulee käyttäjän aina varmistua lataamiensa moduulien lähteiden luotettavuudesta [18].

# 4 Ohjelmiston toiminnallisuus

Edellisessä luvussa käytiin läpi yleisellä tasolla tässä insinöörityössä käytettyjä teknologioita. Tässä luvussa paneudutaan tarkemmin ohjelmiston front-endin ja back-endin toiminnallisuuksiin ja siihen, miten teknologioita on hyödynnetty näiden toiminnallisuuksien toteutuksessa.

## 4.1 Front-end

Tässä insinöörityössä kehitetyssä sovelluksessa käytetty komponenttihierarkia eli komponenttien rakenne on esitetty kuvassa 3.




Kuva 3. Sovelluksen React-komponenttien hierarkia.

Kuvan 3 esittämässä komponenttihierarkiassa siniset suorakulmiot ovat luokkakomponentteja ja keltaiset suorakulmiot funktionaalisia komponentteja.

Komponenttihierarkia kuvaa osittain sitä, miten käyttäjä pääsee etenemään sovelluksessa. Sen lisäksi, että komponentit sisältävät ja näin ollen renderöivät toisia komponentteja, on joissakin komponenteissa määritelty suora polku esimerkiksi painikkeen avulla toiseen näkymään. Tällaiset polkujen kautta renderöitävät komponentit on määritelty *Router*-komponentissa, ja niiden renderöiminen myös päivittää näkymän.

### 4.1.1 Reititys

*App*-luokkakomponentti tarkistaa, onko käyttäjä kirjautunut sisään ja sen perusteella renderöi joko sisäänkirjautumissivun tai kutsuu *Router*-komponenttia (esimerkkikoodi 4).

```
import React from 'react';
import axios from 'axios';
import { observer } from 'mobx-react';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';
import { faSpinner } from '@fortawesome/free-solid-svg-icons';
import Router from './components/Router';
import Login from './components/Login/Login';
import UserStore from './components/Login/UserStore';

class App extends React.Component {
     constructor(props) {
          super(props);
          this.state = {
               username: UserStore.username ? UserStore.username : '',
               isLoggedIn: UserStore.isLoggedIn
                    ? UserStore.isLoggedIn : false,
          }
     }
     async componentDidMount() {
          try {
               UserStore.loading = true;
               await axios.post('http://localhost:3003/isloggedin', {
                    method: 'post',
                    headers: {
                         'Accept': 'application/json',
                         'Content-Type': 'application/json',
                    }
               }).then(res => {
                    UserStore.loading = false;
                    if (res.data.status === 200) {
                         UserStore.isLoggedIn = true;
                    } else {
                         UserStore.isLoggedIn = false;
                    }
               });
          } catch(e) {
               UserStore.loading = false;
               UserStore.isLoggedIn = false;
          }
     }
     render() {
          if (UserStore.loading) {
               return (<FontAwesomeIcon icon={faSpinner} />);
          } else if (UserStore.isLoggedIn) {
               return (
                    <div>
                         <Router />
                    </div>
               );
          } else {
               return (
                    <div>
                         <Login />
                    </div>
               );
          }
     }
};
```

Esimerkkikoodi 4. *App*-komponentti tarkastaa, onko käyttäjä kirjautunut sisään.

*Router*-komponentin tarkoituksena on toimia reitittimenä URL-osoitteiden ja näkymien välillä. *Router*-komponentissa määritellään, minkä sovelluksen näkymän mikäkin URL-osoite renderöi käyttäjän ollessa kirjautuneena sisään (esimerkkikoodi 5).

```
import React from ‘react’;
import { BrowserRouter, Route, Switch } from ‘react-router-dom’;
import Categories from ‘./Categories/Categories’;
import Footer from ‘./Footer/Footer’;
import Items from ‘./Items/Items’;
import Main from ‘./Main/Main’;
import Navigation from ‘./Navigation/Navigation’;
import Shelves from ‘./Shelves/Shelves’;
import SingleItem from ‘./Items/SingleItem’;

const Router = () => {
     return (
          <BrowserRouter>
               <Navigation />
               <Switch>
                    <Route exact path=”/”>
                         <Main/>
                    </Route>
                    <Route exact path=”/kategoriat”>
                         <Categories/>
                    </Route>
                    <Route exact path=”/tuotteet”> <Items/>
                    </Route>
                    <Route exact path=”/kategoria/:category_id”>
                         <Items/>
                    </Route>
                    <Route exact path=”/hylly/:shelf_id”>
                         <Items/>
                    </Route>
                    <Route exact path=”/hyllyt”>
                         <Shelves/>
                    </Route>
                    <Route exact path=”/tuote/:item_id”>
                         <SingleItem/>
                    </Route>
               </Switch>
               <Footer />
          </BrowserRouter>
     );
};

export default Router;
```

Esimerkkikoodi 5. Itsemääritelty *Router*-komponentti sisältää Reactin *Route*-komponentteja.

Router-komponentti on itse kirjoitettu, kun taas *BrowserRouter*- ja *Route*-komponentit on ladattu julkisesta npm-arkistosta. *BrowserRouter*-komponentti käyttää *HTML5 history APIa*, ja *Route*-komponentin perimmäinen tehtävä on renderöidä ruudulle tiettyä sisältöä, mikäli sen attribuutti *path* eli polku vastaa URL-osoitteen polkua, johon käyttäjä on itsensä navigoinut [19].

Esimerkiksi kategoriat-sivulle johtava URL-osoitteen polku on ”/kategoriat”, jolloin renderöidään *Categories*-komponentti, kun taas tietyn kategorian sisältämiä tuotteita esittelevä *Items*-komponentti renderöityy URL-osoitteen polulla ”/kategoria/:category_id”, esimerkiksi ”/kategoria/4531”.

*Items*-komponenttia käytetään kategorian sisältämien tuotteiden renderöinnin lisäksi myös kaikkien tietokannan tuotteiden renderöinnissä sekä tiettyyn hyllyyn kuuluvien tuotteiden renderöinnissä. Kaksoispiste esimerkiksi *category_id*-nimisen URL-parametrin edessä merkitsee, että parametri on dynaaminen, eikä URL-osoitteen tarvitse kirjaimellisesti sisältää sanaa ”category_id”. Tämä mahdollistaa kategorian tietokantaID:n sijoittamisen URL-osoitteeseen ja näin myös oikeiden tuotteiden hakemisen tietokannasta *Items*-komponentin renderöimässä näkymässä.

*Router*-komponentti sisältää myös itse laaditut *Navigation*- ja *Footer*-komponentit, joista *Navigation* sisältää sovelluksen ylävalikon ja hyödyntää myös julkisen npm-arkiston komponentteja. *Footer* puolestaan on sivuston alalaitaan sijoitettu osio, jota käytetään usein yhteystietojen tai tekijänoikeustietojen esittämiseen sivustolla. Esimerkkikoodin 5 viimeisellä rivillä komponentti saatetaan muiden komponenttien saataville avainsanalla *export*. Näin muualta ohjelmistosta saadaan kyseinen komponentti käyttöön avainsanalla *import*.

### 4.1.2 Kategoriat

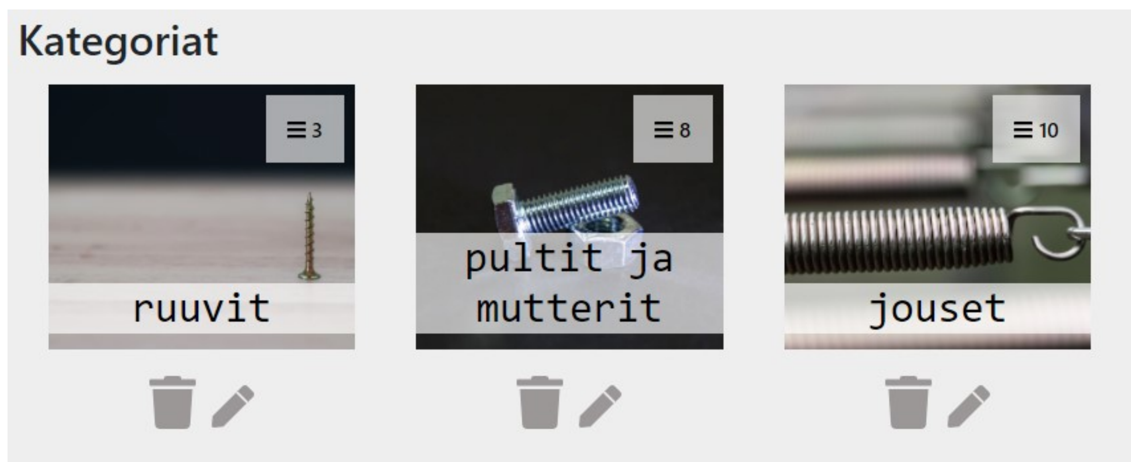
Käyttäjän navigoidessa itsensä URL-osoitteeseen, jonka polku on ’/kategoriat’, renderöidään siis *Router*-komponentin reitityksen mukaisesti *Categories*-komponentti (esimerkkikoodi 6).

```
import React, { Component } from 'react';
import axios from 'axios';
import { Container } from 'react-bootstrap';
import Category from '../Categories/Category';
import AllItemsButton from '../Categories/AllItemsButton';

class Categories extends Component {
     state = {
          categories: [{ id: 0, name: '', image: '',
          },],
          items: [{ name: '', id: 0, image: '', groupID: 0,
               category_name: '', shelf: '', info: '',
               dateModified: '', quantity: 0
          },],
     }
     async componentDidMount() {
          await axios.get('http://localhost:3003/kategoriat')
          .then(res => {
               this.setState({categories: res.data})})
          await axios.get('http://localhost:3003/tuotteet')
          .then(res => {
               this.setState({items: res.data});});
     }
     render() {
          let categoriesInRows = [];
          this.state.categories.forEach(c => {
               let itemsInThisCategory = [];
               this.state.items.forEach(i => {
                    if (i.groupID === c.id) {
                         itemsInThisCategory.push(i);
                    }
               })
               categoriesInRows.push(
                    <Category
                         key={c.id}
                         id={c.id}
                         name={c.name}
                         image={c.imageURL}
                         items={itemsInThisCategory}
                    />
               );
          })
          return(
               <Container>
                    <h2>Kategoriat</h2>
                    <div>
                         {categoriesInRows}
                         <AllItemsButton />
                    </div>
               </Container>
          );
     }
}

export default Categories;
```

Esimerkkikoodi 6. *Categories*-komponentti renderöi jokaisen tietokannasta löytyvän kategorian ja painikkeen, josta pääsee näkemään kaikki tietokannan tuotteet. Tietokannasta löytyvien kategorioiden renderöiminen tapahtuu joukolla *Category*-komponentteja (kuva 4).




Kuva 4. Yksi *Category*-komponentti renderöi ruudulle yksittäisen kategoriakortin.

Kuvassa 4 esitetään ruudulla kolme *Category*-nimistä komponenttia. Komponenttien logiikka on keskenään samanlainen, mutta komponenttien argumentit eli ”propsit” (engl. properties) ovat keskenään erilaiset. Esimerkkikoodissa 7 kuvataan *Category*-komponentin rakennetta.

```
const Category = ({ name, image, categoryId, items }) => {
     const [ open, setOpen ] = React.useState(false);
     const url = '/kategoria/' + categoryId;

     return (
          <Col>
               <a href={url}>
                    <Card.Body>
                         <Card.Title>
                              <FontAwesomeIcon icon={faBars}/>
                              {items.length}
                         </Card.Title>
                         <Card.Text>
                              {name}
                         </Card.Text>
                    </Card.Body>
               </a>
               <ButtonContainer
                    open={open}
                    setOpen={setOpen}
                    name={name}
                    image={image}
                    categoryId={categoryId}
               />
          </Col>
     );
}
export default Category;
```

Esimerkkikoodi 7. *Category*-komponentti renderöi yksittäisen kategorian tietoja sekä *ButtonContainer*-komponentin.

*Category* on funktionaalinen komponentti, ja sen argumentteja ovat *name, image, categoryId* ja *items* eli nimi, kuva, kategorian ID ja kategorian sisältämät tuotteet. Lisäksi komponentin sisällä määritellään useState-koukku-tyyppinen tilaolio, joka kuvaa modaalin, eli näkymän päälle avautuvan toisen näkymän tilaa. Komponentin sisällä määritellään myös *url*-muuttuja, joka käyttää categoryId-argumenttia määrittääkseen sen URL-osoitteen, johon käyttäjä siirtyy kategoriakorttia painaessaan. *Image*-argumenttia käytetään Reactin *Card*-komponentin rungon taustakuvana. *Card*-komponentin otsikko esittää *items*-argumenttia hyödyntäen kategoriaan kuuluvien tuotteiden lukumäärän, ja *Card*-komponentin tekstiosio sisältää kategorian nimen.

### 4.1.3 Tuotteet

Mikäli käyttäjä painaa vaikkapa *Pultit ja mutterit* -kategoriaa, navigoituu hän polkuun '/kategoria/2'. Kyseiselle URL-osoitteelle löytyy vastaavuus *Router*-komponentista, joka määrittelee renderöitäväksi *Items*-komponentin (esimerkkikoodi 8).

```
class Items extends Component {
	state = {
		items: [
			{name:'',id:0,image:'',groupID:0,category_name:'',
			shelfID:'',info:'',dateModified:'',quantity:0
			},
		],
		newItem: {
			name:'',id:0,image:'',groupID:0,category_name: '',
			shelfID:'',info:'',dateModified:'',quantity:0
		},
		listViewSelected: false,
		open: false,
		urlParams: {},
		title: '',
	}
	constructor(props) {
		super(props);
		this.state.urlParams = props.match.params;
		this.setOpen = this.setOpen.bind(this);
		this.setClose = this.setClose.bind(this);
		this.sortByName = this.sortByName.bind(this);
		this.sortByQuantityAsc = this.sortByQuantityAsc.bind(this);
		this.sortByQuantityDesc = this.sortByQuantityDesc.bind(this);
		this.sortByDate = this.sortByDate.bind(this);
		this.selectListView = this.selectListView.bind(this);
		this.selectIconView = this.selectIconView.bind(this);
	}

	// Funktioiden toiminnallisuus on esitetty esimerkkikoodeissa 9 ja
	// 10.

	render() {
		// Render-metodin sisältö on esitetty esimerkkikoodeissa 11
		// ja 12.
	}
}
export Default Items;
```

Esimerkkikoodi 8. *Items*-komponentti sisältää paljon toiminnallisuuksia.

*Items*-komponentissa määritellään ensin tilaoliossa alustetut tiedot tuotteille (*items*), uudelle tuotteelle (*newItem*), valitulle näkymälle (*listViewSelected*), tiedolle uuden tuotteen lisäämisen mahdollistavan modaalin tilasta (*open*), URL-

parametreille (*urlParams*) sekä renderöitävälle otsikolle (*title*). Konstruktorin avulla mahdollistetaan tilaolion tietojen muokkaaminen komponentin funktioiden avulla. Nämä funktiot on esitetty tarkemmin esimerkkikoodissa 9.

```
setOpen() {this.setState({open:true});}
setClose() {this.setState({open: false});}
sortByName() {
     let sortedItems = this.state.items;
     sortedItems.sort(function(a, b) {
          if (a.name.toLowerCase()<b.name.toLowerCase()){return -1}
          else if (a.name.toLowerCase()>b.name.toLowerCase()){return 1}
          return 0;
     })
     this.setState({items: sortedItems})
sortByDate() {
     let sortedItems = this.state.items;
     sortedItems.sort(function(a, b) {
          let dateA = new Date(a.dateModified);
          let dateB = new Date(b.dateModified);
          if (dateA < dateB) { return -1 }
          else if (dateA > dateB) { return 1 }
          return 0;
     })
     this.setState({items: sortedItems})
}
sortByQuantityAsc() {
     let sortedItems = this.state.items;
     sortedItems.sort(function(a, b) {
          if (a.quantity < b.quantity) { return -1 }
          else if (a.quantity > b.quantity) { return 1 }
          return 0;
     })
     this.setState({items: sortedItems})
}
sortByQuantityDesc() {
     let sortedItems = this.state.items;
     sortedItems.sort(function(a, b) {
          if (a.quantity > b.quantity) { return -1 }
          else if (a.quantity < b.quantity) { return 1 }
          return 0;
     })
     this.setState({items: sortedItems})
}
selectListView() {this.setState({listViewSelected: true})}
selectIconView() {this.setState({listViewSelected: false})}
```

Esimerkkikoodi 9. *Items*-komponentin funktiot muokkaavat tilaoliota.

*Items*-komponentin itse määriteltyjä funktioita on kahdeksan. *setOpen*- ja *set-Close*-funktiot liittyvät modaalin avaamiseen ja sulkemiseen. *sortByName*-, *sort-ByDate*-, *sortByQuantityAsc*- ja *sortByQuantityDesc*-funktiot lajittelevat tuotteet

nimen, päivämäärän tai kappalemäärän perusteella. *selectListView*- ja *selectIconView*-funktiot tallettavat valitun näkymän (lista tai kuvakkeet) tilaolioon. Lisäksi *Items*-komponentti sisältää *componentDidMount*-metodin, joka sisältää axios-kutsuja tiedon hakemiseksi tietokannasta (esimerkkikoodi 10).

```
async componentDidMount() {
     if (this.state.urlParams.category_id) {
          await axios.get('http://localhost:3003/kategoria/'+
               this.state.urlParams.category_id
          ).then(res => {this.setState({items: res.data});});
          await axios.get(
               'http://localhost:3003/kategoriatiedot/'+
               this.state.urlParams.category_id
          ).then(res => {
               this.setState({title: 'Kategoria '+.data[0].name});});
     } else if (this.state.urlParams.shelf_id) {
          await axios.get( 'http://localhost:3003/hylly/'+
               this.state.urlParams.shelf_id
          ).then(res => {this.setState({items: res.data});});
          await axios.get( 'http://localhost:3003/hyllytiedot/'+
               this.state.urlParams.shelf_id
          ).then(res => {
               this.setState({title: 'Hylly '+res.data[0].name});});
     } else {
          await axios.get('http://localhost:3003/tuotteet')
          .then(res => {this.setState({items: res.data});});
          this.setState({title: 'Kaikki tuotteet'});
     }
}
```

Esimerkkikoodi 10. *componentDidMount*-metodin avulla haetaan tietoa tietokannasta.

*componentDidMount*-metodi suoritetaan aina komponentin renderöityessä. Tilaolion sisältämän *urlParams*-avaimen avulla määritellään, mitä tietoa tietokannasta haetaan. Mikäli *urlParams* sisältää kategorian ID:n, haetaan tietokannasta kyseisen kategorian sisältämät tuotteet ja kategorian tarkemmat tiedot. Mikäli *urlParams* sisältää hyllyn ID:n, haetaan vastaavasti kyseisen hyllyn sisältämät tuotteet ja hyllyn tarkemmat tiedot. Muussa tapauksessa haetaan kaikki tietokannasta löytyvät tuotteet. *render*-metodin avulla määritellään, mitä komponentti palauttaa (esimerkkikoodi 11).

```
render() {
     let itemsToDisplay = [];
     let itemsAsList = [];
     let listViewSelected = this.state.listViewSelected;
     itemsAsList.push(
          <tr>
               <th>Nimi</th><th>Kuvaus</th><th>Kpl</th>
               <th>Hylly</th><th>Kategoria</th>th>Id</th>th></th>
          </tr>
     );
     let items = this.state.items;
          if(items[0].id!==0) {
               items.forEach(i => {
                    itemsToDisplay.push(
                         <Item
                              key={i.id} itemId={i.id}
                              name={i.name} image={i.imageURL}
                              info={i.info} quantity={i.quantity}
                              date={i.dateModified}
                              categoryId={i.groupID}
                              shelfId={i.shelfID}
                         />
                    );
                    itemsAsList.push(
                         <ItemAsList
                              key={i.id} itemId={i.id}
                              name={i.name} image={i.imageURL}
                              info={i.info} quantity={i.quantity}
                              date={i.dateModified}
                              categoryId={i.groupID}
                              shelfId={i.shelfID}
                         />
                    );
               })
          }
     return (
          // Esitetty esimerkkikoodissa 12.
     );
}
```

Esimerkkikoodi 11. *Items*-komponentin *render*-metodi sisältää muuttujien määrittelyitä ja *return()*-lauseen.

*Render*-metodin sisällä määritellään ensin muuttujat kuvakkeina näytettäville tuotteille sekä listana näytettäville tuotteille. Lisäksi määritellään muuttuja sille, onko lista- vai kuvakenäkymä valittuna. Mikäli tuotteiden haku tietokannasta on onnistunut, lisätään jokainen tietokannasta löytynyt tuote sekä kuvakenäkymään *Item*-komponenttina että listanäkymään *ItemAsList*-komponenttina. Tämä mahdollistaa sen, että valitun näkymän vaihtaminen käyttöliittymässä renderöi tuotteet näytölle nopeasti, kun tiedot ovat jo valmiina eikä niitä tarvitse prosessoida näkymän vaihtamisen yhteydessä. *return*-lauseella palautetaan renderöitävä HTML-sisältö (esimerkkikoodi 12).

```
<Container>
     <h5>Lajittele ja suodata</h5>
     <Dropdown>
          <Dropdown.Toggle>Lajittele</Dropdown.Toggle>
          <Dropdown.Menu>
               <Dropdown.Item>Nimen mukaan
               </Dropdown.Item>
               <Dropdown.Item>Varastosaldon mukaan (pienin ensin)
               </Dropdown.Item>
               <Dropdown.Item>Varastosaldon mukaan (suurin ensin)
               </Dropdown.Item>
               <Dropdown.Item>Muokkauspäivän mukaan
               </Dropdown.Item>
          </Dropdown.Menu>
     </Dropdown>
     <Dropdown>
          <Dropdown.Toggle>Näytä</Dropdown.Toggle>
          <Dropdown.Menu>
               <Dropdown.Item
                    onClick={this.selectListView}>
                    Lista
                    {listViewSelected
                    ?
                    <FontAwesomeIcon icon={faCircle}>
                    : ''}
               </Dropdown.Item>
               <Dropdown.Item
                    onClick={this.selectIconView}>
                    Kuvakkeet
                    {!listViewSelected
                    ?
                    <FontAwesomeIcon icon={faCircle}>
                    : ''}
               </Dropdown.Item>
          </Dropdown.Menu>
     </Dropdown>
     <div>
          <h1>{this.state.title}</h1>
          <NewItemForm
               open={this.state.open}
               setOpen={this.setOpen}
               setClose={this.setCLose}
          />
     </div>
     <div>
     {listViewSelected
          ?<table>{itemsAsList}</table>:{itemsToDisplay}
     }
     </div>
</Container>
```

Esimerkkikoodi 12. *Items*-komponentin *return*-lause palauttaa renderöitävää HTML-koodia.

*Items*-komponentin *return*-lauseen HTML-koodi sisältää ensin *Dropdown*-komponentit tuotteiden lajittelulle nimen, varastosaldon tai muokkauspäivän mukaan ja tuotteiden esittämiselle lista- tai kuvakenäkymän mukaisesti. Lisäksi HTML-

koodi sisältää *NewItemForm*-komponentin uuden tuotteen lisäämiseksi ja valitun näkymän mukaan joko joukon *Item*-komponentteja tai *ItemAsList*-komponentteja.

Yksittäinen *Item*-komponentti näytetään ruudulla tuotekorttina (kuva 5).

Kuva 5. Yksittäinen *Item*-komponentti renderöidään ruudulla tuotekorttina.

*Item*-komponentin taustalla oleva koodi on esitelty esimerkkikoodissa 13.

```
const Item = ({itemId, name, image, info, quantity, date, categoryId,
              shelfId}) => {
     const [open, setOpen] = React.useState(false);
     const [categoryState, setCategory] = React.useState({});
     const [shelfState, setShelf] = React.useState({});

     useEffect(() => {
          axios.get(‘http://localhost:3003/kategoriatiedot/’
               +categoryId)
          .then(res => {
               setCategory(res.data);
          });
          axios.get(‘http://localhost:3003/hyllytiedot/’+shelfId)
          .then(res => {
               setShelf(res.data);
          });
     }, [])
     const handleRemoveClick = (event) => {
          event.preventDefault();
          let data = { id: itemId};
          if (window.confirm(‘Poistetaanko tuote”’+name+’”?’) {
               axios.delete(‘http://localhost:3003/poista-tuote)
               .then(res => {
                    window.location = ‘/kategoria/’+categoryId;
               });
          } else { return false; }
     }
     let url = ‘/tuote/’+itemId;
     let shelfUrl = '/hylly/' + shelfId;
     let categoryUrl = '/kategoria/' + categoryId;
     let dateModified = new Date(date);
     let shelf = shelfState[0] ? shelfState[0] : {};
     let category = categoryState[0] ? categoryState[0] : {};

     return (
          // Esitetty esimerkkikoodissa 14.
     );
}
export default Item;
```

Esimerkkikoodi 13. *Item*-komponentti renderöi yksittäisen tuotteen tiedot tuotekorttina.

Esimerkkikoodissa 13 esitetty *Item*-komponentti on funktionaalinen komponentti, jolle on annettu argumentteina tuotteen ID, nimi, kuvan URL-osoite, lisätieto, kappalemäärä ja muokkauspäivämäärä sekä tieto siitä, mihin kategoriaan ja mihin hyllyyn tuote kuuluu. *useEffect*-metodilla haetaan tietokannasta tiedot kategoriasta ja hyllystä komponentin renderöityessä. Tietokannan palauttamat tiedot talletetaan *categoryState*- ja *shelfState*-tilaolioihin. *handleRemoveClick*-funktion avulla poistetaan tuote tietokannasta. Mikäli käyttäjä vahvistaa haluavansa poistaa tuotteen, lähetetään *delete*-muotoinen HTTP-kutsu ja kutsun valmistuttua navigoidaan käyttäjä takaisin tuotteet-näkymään, jossa renderöidään

poistettuun tuotteeseen linkitetyn kategorian jäljellä olevat tuotteet. Käytännössä käyttäjä oli jo ennen tuotteen poistamista kyseisessä näkymässä, joten tuotteen poistaminen tietokannasta lataa sivun uudelleen ja hakee tietokannasta ajantasaiset tiedot renderöitäväksi.

Lisäksi esimerkkikoodissa 13 määritellään ennen *return*-lausetta kuusi muuttujaa. Ensin määritellään URL-osoitteet tuotteelle, hyllylle ja kategorialle. Tämän jälkeen määritellään muokkauspäivämäärästä luotu uusi päivämäärämuuttuja. Viimeiseksi tallennetaan *shelf*- ja *category*-muuttujiin tilaolioista noudetut hyllyn ja kategorian tiedot, mikäli kyseisten tietojen tietokantakysely on suoritettu onnistuneesti.

*Item*-komponentin palauttama HTML-koodi on esitetty esimerkkikoodissa 14.

```
<div>
     <Col>
          <Card.Body>
               <LinkContainer to={
                    pathname: url,
                    props: {
                         item_id: itemId,
                         category_id: categoryId,
                         name: name,
                         image: image,
                         info: info,
                         quantity: quantity,
                         dateModified: dateModified
               }}}><img src={image}/></LinkContainer>
               <div>
                    <Card.Title>{name}</Card.Title>
                    <Card.Text>{infoText}</Card.Text>
               <div>
          </Card.Body>
     </Col>
     <div>
          <button onClick={() => {window.location.href=shelfUrl}}
          >hylly <b>{shelf.name}</b>
          </button>
          <button onClick={() => {window.location.href=categoryUrl}}
          >kategoria <b>{category.name}</b>
          </button>
     </div>
     <div><b>ID: </b>{itemId}</div>
     <div>
          <form onSubmit={handleRemoveClick}>
               <button title="Poista tuote type="submit">
                    <FontAwesomeIcon icon={faTrash} />
               </button>
               <button title="Muokkaa" type="button"
                    onClick={() => setOpen(true)}>
                    <FontAwesomeIcon icon={faPen} />
               </button>
          </form>
          <NewItemForm
               newItemName={name}
               newItemInfo={info}
               newItemId={itemId}
               newItemQuantity={quantity}
               newItemShelf={shelf.name}
               newItemShelfInfo={shelf.info}
               newItemShelfId={shelfId}
               newItemFilePath={image}
               newItemCategory={categoryId}
               newItemCategoryName={category.name}
               open={open}
               setOpen={setOpen}
          />
     </div>
</div>
```

Esimerkkikoodi 14. *Item*-komponentin *return*-lauseen sisältämä HTML-koodi.

Esimerkkikoodissa 14 esitetään, miten jokainen tuotekortti rakentuu erilaisista elementeistä. Esimerkiksi *LinkContainer*-komponentin avulla määritellään, mitä tapahtuu käyttäjän klikatessa tuotekortin kuvaa. Kuvan alapuolelle renderöidään tuotteen nimi ja lisätieto. Tämän jälkeen renderöidään kaksi painiketta – toisen avulla käyttäjä voi poistaa tuotteen ja toisen painikkeen painaminen avaa tuotteen muokkausnäkymän, eli *NewItemForm*-komponentin muuttamalla *open*-olion tilaa.

### 4.1.4 Yksittäinen tuote

Käyttäjän yhä jatkaessa komponenttihierarkiassa eteenpäin painamalla jotakin *Items*-komponentin sisältämää yksittäistä *Item*-komponenttia renderöidään ruudulle yksittäisen tuotteen tiedot. *LinkContainer*-komponentin *pathname*-parametrin avulla määritellään, mihin polkuun käyttäjä navigoidaan, mikäli tuotekorttia painetaan. Esimerkiksi tuotteen *Pultti 15 mm* kohdalla *pathname*-parametri saisi arvon *'/tuote/3'*. Esimerkkikoodin 5 *Route*-komponenteista yksi sisältää tämän saman polun parametrina, ja *component*-parametrin mukaan kyseinen polku renderöi komponentin *SingleItem* (esimerkkikoodi 15).

```
class SingleItem extends Component {
     state = {
          item: {
               name:'',id:0,imageURL:'',groupID:0,shelfID:'',
               info:'',quantity:0,dateModified:'',
          },
          category: {name:'',id:'',},
          shelf: {name:'',id:0,info:''},
          open: false,
     }
     constructor(props) {
          super(props);
          this.setOpen = this.setOpen.bind(this);
     }
     async componentDidMount() {
          await axios.get('http://localhost:3003/tuote/'
          + this.props.match.params.item_id)
          .then(res => {
               this.setState({item: res.data[0]});
          });
          await axios.get('http://localhost:3003/kategoria/'
          + this.state.item.groupID)
          .then(res => {
               this.setState({category: res.data[0]})
          });
          await axios.get('http://localhost:3003/hylly/
          + this.state.item.shelfID)
          .then(res => {
               this.setState({shelf: res.data[0]})
          });
     }
     setOpen() {
          this.setState({open: false});
     }

     render() {
          // Esitetty esimerkkikoodissa 16.
     }
};
export default SingleItem;
```

Esimerkkikoodi 15. *SingleItem*-komponentissa määritellään ennen renderöintiä tila, konstruktori, *componentDidMount()*-metodi ja *setOpen*-funktio.

*SingleItem*-komponentin tilaolio (*state*) sisältää alustetut tiedot tuotteesta, kategorioista, hyllyistä sekä tiedon modaalin tilasta (*open*). Konstruktoria käytetään, jotta *setOpen()*-funktion avulla päästään muokkaamaan tilaolion *open*-avaimen arvoa. *componentDidMount()*-metodin sisältämä koodi suoritetaan aina, kun komponentti renderöidään. *SingleItem*-komponentin tapauksessa *componentDidMount()*-metodi hakee tietokannasta tuotteen, kategorian ja hyllyn ja tallettaa ne tilaolioon. *render*-metodi määrittelee sen, mitä komponentti palauttaa (esimerkkikoodi 16).

```
let item = this.state.item.name != “”
     ? this.state.item : undefined;
let category = this.state.category.name != “”
     ? this.state.category : undefined;
let shelf = this.state.shelf.name != “”
     ? this.state.shelf : undefined;
     if (item && category && shelf) {
          return (
               // Esitetty esimerkkikoodissa 17.
          )
     } else {
           return <div></div>
     }
```

Esimerkkikoodi 16. *SingleItem*-komponentin *render*-metodi sisältää muuttujien määrittelyitä ja *return()*-lauseen.

*SingleItem*-komponentin *render*-metodissa määritellään ensin muuttujat ”tuote” (*item*), ”kategoria” (*category*) ja ”hylly” (*shelf*). Jokaisen muuttujan kohdalla tarkistetaan, onko kyseistä muuttujaa edustavan tilaolion nimi tyhjä merkkijono. Mikäli merkkijono on tyhjä, ei tuotteen tietoja ole saatu haettua tietokannasta. Mikäli tuotteen tietojen haku tietokannasta on onnistunut, määritellään muuttujat vastaamaan tilaolion sisältämiä tietoja.

Jos tuotteen, kategorian tai hyllyn tietoja ei ole saatu haettua tietokannasta, palautetaan return-lauseella tyhjä elementti. Mikäli tietokantahaku on onnistunut, palautetaan HTML-koodina renderöitävä sisältö (esimerkkikoodi 17).

```
<div>
     <h1>{item.name}</h1>
     <img src={item.imageURL} />
     <TableContainer>
          <Table>
               <TableBody>
                    <TableRow>
                         <TableCell>Varastosaldo</TableCell>
                         <TableCell>{item.quantity} kpl</TableCell>
                    </TableRow>
                    <TableRow>
                         <TableCell>Lisätiedot</TableCell>
                         <TableCell>{item.info}</TableCell>
                    </TableRow>
                    <TableRow>
                         <TableCell>Kategoria</TableCell>
                         <TableCell>
                              <button>{category.name}</button>
                         </TableCell>
                    </TableRow>
                    <TableRow>
                         <TableCell>Hylly</TableCell><TableCell>
                              <button>{shelf.name}</button>
                              <span>{shelf.info}</span>
                         </TableCell>
                    </TableRow>
                    <TableRow>
                         <TableCell>Muokattu viimeksi</TableCell>
                         <TableCell>{item.dateModified}</TableCell>
                    </TableRow>
               </TableBody>
          </Table>
     </TableContainer>
     <Button onClick={() => {
          this.setState({open: true})}}
     >Muokkaa tuotteen tietoja
     </Button>
     <NewItemForm
          newItemName={item.name}
          newItemInfo={item.info}
          newItemId={item.id}
          newItemQuantity={item.quantity}
          newItemShelf={shelf.name}
          newItemShelfInfo={shelf.info}
          newItemShelfId={item.shelfId}
          newItemFilePath={item.image}
          newItemCategory={item.categoryId}
          newItemCategoryName={category.name}
          open={this.state.open}
          setOpen={this.setOpen}
     />
</div>
```

Esimerkkikoodi 17. *SingleItem*-komponentti palauttaa tietokantahaun onnistuttua HTML-koodia.

*SingleItem*-komponentin palauttama HTML-koodi sisältää *div*-elementin, joka puolestaan sulkee sisäänsä muita elementtejä ja React-komponentteja. Näistä

React-komponenteista *TableContainer*-komponentti ja sen sisältämät *Table-*, *TableBody-, TableRow-* ja *TableCell*-komponentit renderöivät ruudulle tuotteen tietoja, kuten lisätiedot ja varastosaldon sekä kategorian, johon tuote kuuluu. Lisäksi renderöidään painike, jota painamalla *NewItemForm*-komponentille annettava argumentti *open* saa arvon *true*.

*SingleItem*-komponentti renderöi näytölle vain yhden tuotteen tiedot, ja näin ollen tuotteen tietojen esittämiselle jää laajemmin tilaa. Esimerkiksi kuva näytetään *SingleItem*-komponentissa suurempana kuin *Item*-komponentissa.

### 4.1.5 Tietojen lisääminen, muokkaaminen ja poistaminen

Sen lisäksi, että sovelluksen avulla voidaan tarkastella tietokannasta löytyviä tietoja, voi käyttäjä myös lisätä, muokata ja poistaa tietoja tietokannasta. Aiemmin esitellyn esimerkkikoodin 7 *Category*-komponentti pitää sisällään myös itse määritellyn komponentin *ButtonContainer,* joka sisältää painikkeet kategorian poistamista ja muokkaamista varten (esimerkkikoodi 18).

```
const ButtonContainer = ({ open, setOpen, name, image, categoryId })
=> {
     // handleRemoveClick-funktio on esitelty esimerkkikoodissa 6.
     return (
          <div>
               <form onSubmit={handleRemoveClick}>
                    <button
                         title="Poista kategoria"
                         type="submit"

                    ><FontAwesomeIcon icon={faTrash}/></button>
                    <button
                         title="Muokkaa kategorian tietoja"
                         type="button"
                         onClick={() => setOpen(true)}
                    ><FontAwesomeIcon icon={faPen}/><button/>
               </form>
               <NewCategoryForm¨
                    categoryId={categoryId}
                    name={name}
                    image={image}
                    open={open}
                    setOpen={setOpen}
               />
          </div>
     );
};
export default ButtonContainer;
```

Esimerkkikoodi 18. *ButtonContainer*-komponentti mahdollistaa kategorian muokkaamisen ja poistamisen.

*ButtonContainer*-komponentissa kutsutaan edelleen komponenttia *NewCategoryForm*, jolle annetaan argumentteina kategorian ID, nimi, kuvan URL-osoite sekä Reactin tilaolio modaalin avaamiseksi ja sulkemiseksi. Esimerkiksi käyttäjän painaessa muokkausikonia avautuu näkymän ylle uusi näkymä eli modaali, joka sisältää lomakkeen kategorian tietojen muokkausta varten. Modaali tarvitsee tiedon siitä, tuleeko sen olla näkyvillä. Tämä onnistuu *open*-tilaolion avulla. Lähtökohtaisesti *open*-olion arvo on *false*, mutta käyttäjän painaessa muokkausikonia saa olio arvon *true.* Uusi arvo asetetaan *setOpen*-actionilla, jolloin modaali avautuu näkymän ylle. Modaali sulkeutuu, kun käyttäjä painaa *Peruuta*-näppäintä. Tällöin *open*-oliolle annetaan *setOpen*-actionin avulla jälleen arvo *false*. *NewCategoryForm*-komponentin sisältämä React-komponentti *Popup* hyödyntää *open*-parametria määritelläkseen, renderöidäänkö komponentti ruudulle vai ei.

Modaalin sisältö on määritelty *NewCategoryForm*-komponentissa, jonka palauttama HTML-koodi on esitetty hieman tiivistetysti esimerkkikoodissa 19.

```
return (
     <Popup open={props.open} modal>
          <span>
               <h1>Muokkaa kategoriaa}</h1>
               <form onSubmit={handleUpdateCategory}>
                    <label htmlFor=”categoryName”>
                         Kategorian nimi
                    </label>
                    <input
                         required
                         name=”categoryName”
                         type=”text”
                         value={categoryName}
                         onChange={handleNameChange}
                    />
                    <label htmlFor=”categoryImage”>
                         Kuvan lisääminen
                    </label>
                    <input
                         type=”file”
                         onChange={handleImageChange}
                    />
                    <input
                         type=”submit”
                         value=”Tallenna”
                    />
                    <button
                         type=”button”
                         onClick={() => {
                              props.setOpen(false);
                         }}
                    >
                         Peruuta
                    </button>
               </form>
          </span>
     </Popup>
);
```

Esimerkkikoodi 19. *NewCategoryForm*-komponentin palauttama koodi sisältää esimerkiksi *input*-kenttiä käyttäjän syötteitä varten.

Käyttäjän painaessa käyttöliittymässä esimerkiksi *Poista kategoria* -painiketta lähettää front-end-palvelin back-end-palvelimelle delete-muotoisen HTTP-pyynnön Axios-kirjaston avulla. *Poista kategoria* -painikkeelle on lisätty *handleRemoveClick*-niminen *onSubmit*-tapahtumakuuntelija, jonka avulla määritellään, mitä ohjelmassa tapahtuu, kun käyttäjä painaa painiketta (esimerkkikoodi 20).

```
const handleRemoveClick = (event) => {
     event.preventDefault();
     let data = {
          id: categoryId
     }
     if (window.confirm(Poistetaanko kategoria "'+categoryName+'"?')) {
          axios.delete(
               'http://localhost:3003/poista-kategoria',
               {data: data}
          )
          .then(res => {
               if (res.status === 200) {
                    window.location = '/';
               }
          }
     } else {
          return false;
     }
}
```

Esimerkkikoodi 20. HandleRemoveClick-metodin koodi suoritetaan käyttäjän painaessa *Poista kategoria* -painiketta.

Esimerkkikoodin 20 *event.preventDefault()*-metodi estää sivun uudelleenlataamisen painikkeen painamisen yhteydessä. Käyttäjältä varmistetaan, että kategoria todella halutaan poistaa, minkä jälkeen Axios-kirjaston avulla lähetetään delete-muotoinen HTTP-pyyntö, joka saa parametreina URL-osoitteen kategorian poistamiselle sekä tiedon kategorian ID:stä. Mikäli pyyntö palauttaa statuksen 200 (pyyntö onnistunut), ohjataan käyttäjä takaisin etusivunäkymään. Jos käyttäjä ei vahvista kategorian poistamista, HTTP-pyyntöä ei lähetetä.

Hyllyn ja tuotteen poistaminen toimii samalla tavalla kuin kategorian poistaminenkin. Tuotteen poistamisella tarkoitetaan tuotteen ja sen kaikkien tietojen poistamista tietokannasta, ei tuotteen kappalemäärän vähentämistä. Tuotteen kappalemäärän voi vähentää minimissään nollaan, eikä tämä poista tuotetta tietokannasta, vaan sen kappalemääräksi tallentuu nolla. Tuotteen kappalemäärää voidaan muuttaa muokkaamalla tuotteen tietoja. Tuotteen tietojen muokkaaminen tapahtuu lomakkeella, jossa käyttäjälle renderöidään valmiiksi arvot tuotteen nimestä, lisätiedoista, kappalemäärästä, kategoriasta ja hyllypaikasta (kuva 6).

Kuva 6. Tuotteen tietojen muokkaaminen tapahtuu lomakkeen avulla.

Tietojen muokkaamisen jälkeen käyttäjä painaa *Tallenna*-painiketta, mikä aktivoi *handleUpdateSubmit*-tapahtumakuuntelijan (esimerkkikoodi 21).

```
handleUpdateSubmit = event => {
     event.preventDefault();
     let data = {
          name: this.state.newItemName,
          info: this.state.newItemInfo,
          id: this.state.newItemId,
          quantity: this.state.newItemQuantity,
          groupID: this.state.newItemCategory,
          shelfID: this.state.newItemShelf,
          imageURL: this.state.newItemFilePath,
          dateModified: this.state.newItemDateModified
     }
     axios.post('http://localhost:3003/muokkaa-tuotetta', data)
     .then(res => {
          window.location = '/kategoria/' + this.state.newItemCategory;
     })
}
```

Esimerkkikoodi 21. *handleUpdateSubmit*-tapahtumakuuntelija lähettää *post*-tyyppisen HTTP-kutsun.

*handleUpdateSubmit*-tapahtumakuuntelija määrittelee oliomuodossa tuotteen tiedot ja lähettää ne *post*-tyyppisenä HTTP-pyyntönä back-end-palvelimelle. Kun pyyntö on valmistunut, navigoidaan käyttäjä sivulle, joka renderöi tuotteet äskettäin muokatun tuotteen kategoriassa.

Tiedon lisääminen tietokantaan vaatii käyttäjältä yleensä jonkinnäköisiä syötteitä. Esimerkiksi uutta kategoriaa lisätessään käyttäjä syöttää kategorian nimen, valitsee halutessaan kategorialle kuvan ja painaa *Tallenna*-painiketta (kuva 7).

Kuva 7. *Tallenna*-painikkeen painaminen lähettää pyynnön uuden kategorian lisäämisestä tietokantaan.

Käyttäjän painettua *Tallenna*-painiketta käyttöliittymä välittää front-end-palvelimelle uuden kategorian nimen ja mahdollisen kuvan. Front-end-palvelin lähettää *post*-muotoisen HTTP-pyynnön back-end-palvelimelle uuden tiedon lisäämistä varten (esimerkkikoodi 22) hyödyntäen JavaScriptin Axios-kirjastoa.

```
axios.post(
     '/uusi-kategoria',
     {
          name: newCategoryName,
          url: newImageUrl
     }
)
```

Esimerkkikoodi 22. *post*-muotoinen front-end-palvelimen lähettämä HTTP-pyyntö sisältää parametreina URL-osoitteen (/uusi-kategoria) ja back-end-palvelimelle lähetettävät, käyttäjän käyttöliittymässä lisäämät tiedot oliomuodossa.

Muita käyttäjän syötteitä vaativia tietokantakutsuja ovat esimerkiksi tuotteen, kategorian tai hyllyn lisääminen tietokantaan tai näiden tietojen muokkaaminen tietokannasta. Tuotteen, kategorian tai hyllyn poistaminen ei vaadi käyttäjältä varsinaisia syötteitä, vaan ainoastaan vahvistuksen siitä, että poistoa ollaan todella suorittamassa eikä kyseessä ollut esimerkiksi vahinkopainallus. Poistettavan kategorian, hyllyn tai tuotteen tiedoista tarvitaan ainoastaan ID, ja se lähetetään front-end-palvelimelta back-end-palvelimelle ja edelleen tietokantaan ilman käyttäjän syötteitä.

## 4.2 Back-end

Back-end-palvelimen REST API -rajapinta on luotu back-end-palvelimella toimivan Express-sovelluskehyksen avulla. Back-end-palvelimen REST API -rajapinta reitittää front-end-palvelimelta tulleet pyynnöt oikeisiin endpointeihin. Esimerkiksi esimerkkikoodissa 20 kuvattu pyyntö tietyn kategorian poistamiseksi on lähtenyt front-end-palvelimelta *delete*-tyyppisenä, ja sille määriteltiin tietty URL-osoite. Kun reititysmetodi (engl. router method) eli *delete* ja polku vastaavat front-end-palvelimelta saapunutta pyyntöä, lähetetään tietokantaan kysely tietyn kategorian poistamiseksi (esimerkkikoodi 23).

```
app.delete('/poista-kategoria', (req, res) => {
     const categoryId = req.body.id;
     const queryString = "DELETE FROM categories WHERE id = ?";

     pool.getConnection()
     .then(conn => {
          conn.query(queryString, [categoryId])
          .then(() => {
               res.sendStatus(200);
               conn.end();
          })
          .catch(err => {
               console.log('Query error: ', err);
               conn.end();
          })
     })
     .catch(err => {
          console.log('Connection error: ', err);
          conn.end();
     });
})
```

Esimerkkikoodi 23. Oikean endpointin löydyttyä suoritetaan määritelty tietokantakysely.

Esimerkkikoodin 23 tietokantakyselyssä poistetaan kategoriataulusta se kategoria, jonka id on annettu pyynnölle parametrina. Kun poisto on suoritettu onnistuneesti, palautetaan front-end-palvelimelle status 200. Mikäli jotain menee pieleen, tulostetaan virheviesti konsoliin.

Vastaavasti tiedon hakeminen tietokannasta tapahtuu *get*-metodin avulla (esimerkkikoodi 24).

```
app.get('/kategoriat', (req, res) => {
     pool.getConnection()
     .then(conn => {
          conn.query('SELECT * FROM categories')
          .then((rows) => {
               res.json(rows);
               conn.end();
          })
     })
})
```

Esimerkkikoodi 24. Kategorioiden hakeminen tietokannasta tapahtuu *get*-metodilla.

Esimerkkikoodissa 24 kysely ei käytä front-end-palvelimelta tulevia parametreja, vaan hakee tietokannasta kaikki tauluun *categories* kuuluvat rivit ja palauttaa ne

front-end-palvelimelle. Esimerkkikoodissa 25 on esitetty *post*-muotoinen pyyntö, jonka avulla talletetaan uusi kategoria tietokantaan.

```
app.post('/uusi-kategoria', (req, res) => {
     pool.getConnection()
     .then(conn => {
          return conn.query(
               `INSERT INTO categories (name, imageURL) VALUES (?,?);`,
               [req.body.name, req.body.imageURL]
          )
     })
})
```

Esimerkkikoodi 25. *Post*-metodin avulla talletetaan tietokantaan uusi kategoria.

Esimerkkikoodissa 25 tietokantakyselylle annetaan parametreina front-end-palvelimelta (esimerkkikoodi 22) saadut tiedot kategorian nimestä ja kuvan URL-osoitteesta. Kuten jo aiemmin mainittiin, käyttäjän syötteitä vaativia tietokantakutsuja ovat myös tuotteen, kategorian tai hyllyn lisääminen tietokantaan tai näiden tietojen muokkaaminen tietokannasta. Myös tiedon muokkaaminen tapahtuu *post*-muotoisella pyynnöllä, mutta pyynnön tietokantakysely eroaa tiedon tallettamisen *post*-kyselystä (esimerkkikoodi 26).

```
app.post('/muokkaa-tuotetta', (req, res) => {
     const name = req.body.name;
     const id = req.body.id;
     const quantity = req.body.quantity;
     const info = req.body.info;
     const imageURL = req.body.imageUrl;
     const shelfID = req.body.shelfID;
     const groupID = req.body.groupID;
     const dateModified = req.body.dateModified;
     const queryString = "UPDATE ITEMS set name = ?, quantity = ?, info
= ?, imageURL = ?, shelfID = ?, groupID = ? WHERE id = ?";

     pool.getConnection()
     .then(conn => {
          return conn.query(queryString, [
               name, quantity, info, imageURL, shelfID, groupID, id
          ])
     })
})
```

Esimerkkikoodi 26. Myös tiedon muokkaaminen tapahtuu *post*-metodin avulla.

Esimerkkikoodissa 26 määritellään ensin muuttujiksi front-end-palvelimelta tulevan pyynnön parametrit, jotta niihin olisi helpompi viitata jatkossa. Tietokannan avainsana *UPDATE* viittaa tiedon päivittämiseen. Tietokantakysely päivittää sen tuotteen tiedot, jonka ID vastaa HTTP-pyynnön *id*-parametria. Myös kategorian ja hyllyn tietojen päivittäminen toimii samalla logiikalla; vain päivitettävät tiedot eroavat toisistaan.

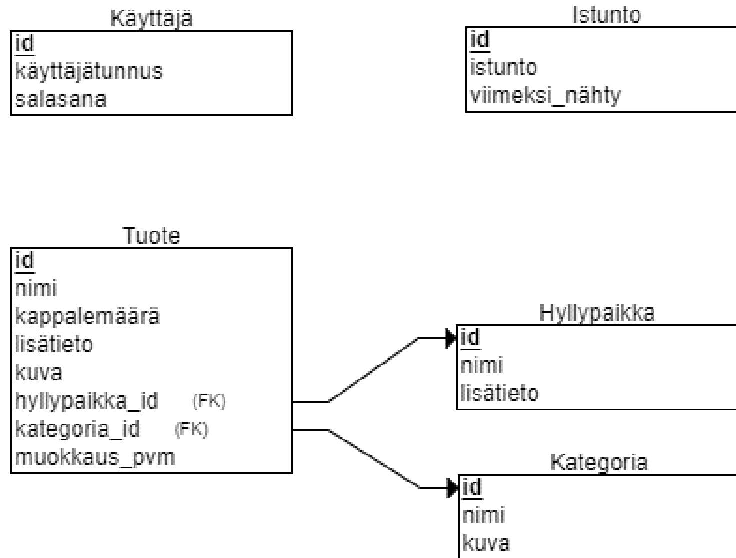
## 4.3 Tietokantaratkaisu

Tässä insinöörityössä tietokantana on käytetty MariaDB Server -tietokantaa. MariaDB on SQL-pohjainen relaatiotietokanta, jonka kehittäjät ovat kehittäneet myös ruotsalaisen MySQL-relaatiotietokantaohjelmiston. Projektissa käytetyn MariaDB:n versio on 10.5.4, sillä kyseinen versio oli projektin aloitushetkellä viimeisin vakaa MariaDB:n ohjelmistojulkaisu [20].

Vaikka MariaDB ja MySQL ovat monilta osin hyvin samankaltaisia ja esimerkiksi niiden datatiedostot ovat yhteensopivia, on näiden tietokantaohjelmistojen välillä kuitenkin myös eroja. MariaDB tukee useampia tietokantamoottoreita, joista esimerkiksi Memory-tietokantamoottori mahdollistaa tiedon syöttämisen tietokantaan 24 % nopeammin kuin MySQL:n standarditietokantamoottori [21].

Tässä projektissa käytettiin MariaDB:n oletustietokantamoottoria, joka on sama kuin MySQL:llä (InnoDB), sillä projektin vaatimuksia määriteltäessä kyseinen tietokantamoottori vaikutti täysin riittävältä projektin tarpeisiin nähden. Tietokantamoottorin vaihtaminen on tarvittaessa helppoa ja onnistuu yhden komennon avulla [22]. Tietokantamoottorin vaihtamisessa tulee kuitenkin olla tarkkana, sillä esimerkiksi perusavainten määrittelyn puuttuminen voi johtaa virheelliseen indeksointiin [23].

Tietokantaa käytetään hylly-, kategoria- ja tuotetietojen säilytykseen sekä käyttäjätietojen hallintaan. Projektin vaatimusten mukaisesti tietokantaan tarvittiin ainoastaan viisi taulua, joten tietokannan rakenne on hyvin yksinkertainen (kuva 8).




Kuva 8. Tietokannan taulut ja taulujen sarakkeet.

Jokaisella taululla on sarakkeita, joista ID eli yksilöllinen tunniste toimii jokaisessa taulussa perusavaimena ja on aina pakollinen. Todellisuudessa tietokannan taulujen ja kenttien nimet ovat englanniksi, mutta selkeyden vuoksi ne esitetään tässä raportissa suomennettuina.

*Käyttäjätaulua* käytetään sisäänkirjautumisessa. Käyttäjällä on ID:n lisäksi käyttäjätunnus- ja salasanakentät sisäänkirjautumista varten. *Tuotetaulu* sisältää ID:n lisäksi tiedot tuotteen nimestä ja kappalemäärästä, tuotteen lisätiedon, kuvan sekä viiteavaimina hyllypaikan ja kategorian ID:n. Lisäksi tuotteen tietoihin tallentuu automaattisesti viimeisin muokkauspäivämäärä, jota käyttäjän ei tarvitse määrittää, vaan joka generoituu aina muokkauksen ajankohdan perusteella. Viiteavainten avulla tuote voidaan yhdistää hyllypaikkaan ja kategoriaan. Tuotteiden tiedoista ainoastaan ID, nimi, kappalemäärä, kategoria ja hylly ovat pakollisia tietoja – loput kentät ovat vapaaehtoisia. *Hyllypaikkataululla* on ID:n lisäksi nimi- ja lisätietokentät, joista ID ja nimi ovat pakollisia ja lisätieto vapaa-

ehtoinen. *Kategoriataulun* sarakkeista ID ja nimi ovat pakolliset ja kuva vapaa-ehtoinen. Kukin tuote voi kuulua kerrallaan vain maksimissaan yhteen hyllyyn ja vain maksimissaan yhteen kategoriaan.

Lisäksi sisään kirjautuneen käyttäjän istuntotietoja varten on *istuntotaulu*, joka pitää sisällään 32-merkkisen ID-kentän, evästeitä ja vanhenemistietoja sisältävän istuntokentän sekä päivämääräkentän, johon talletetaan tieto siitä, milloin kyseinen istunto on viimeksi ollut aktiivinen.

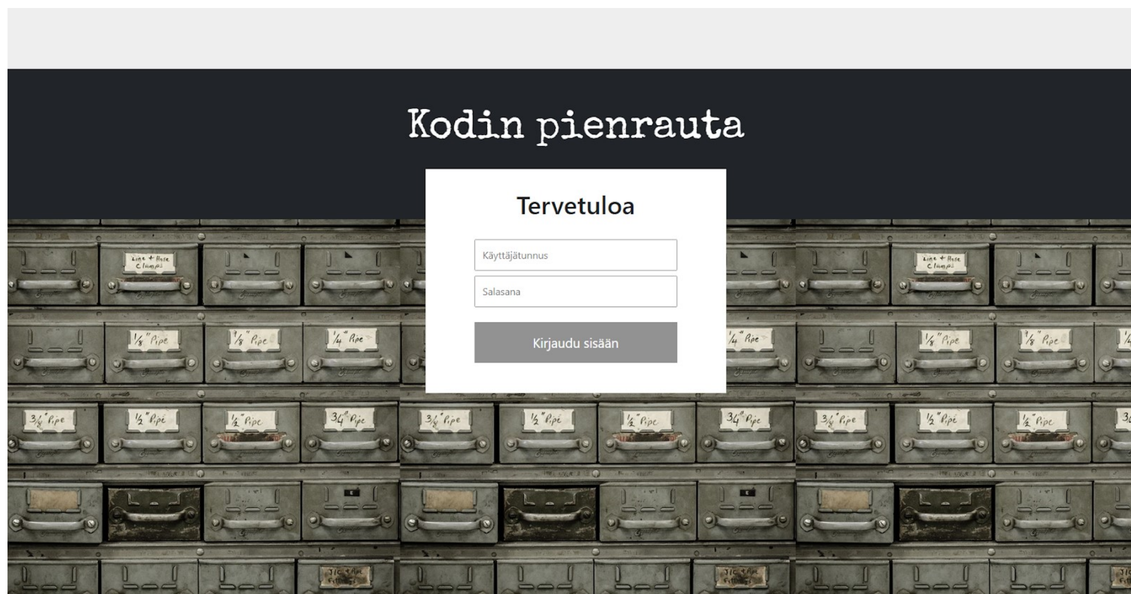
# 5 Käyttöliittymä

Kuten jo edellä esitettiin, sovelluksen käyttöliittymä toteutettiin React.js-kirjastolla. Käyttöliittymä koostuu komponenteista, joiden tarkoitus on palvella taulukossa 1 esiteltyjä käyttäjätarinoita. Jokainen komponentti (poislukien *App*- ja *Router*-komponentit) renderöi jonkin näkymän tai näkymän osan.

Käyttöliittymän näkymät voidaan jakaa kolmeen kategoriaan:

- tietojen tarkastelu
- tietojen lisääminen, muokkaaminen ja poistaminen
- muut toiminnot.

Näiden näkymien lisäksi sovellus sisältää sisäänkirjautumisnäkymän. Ilman käyttäjän sisäänkirjautumista muut näkyvät eivät ole käyttäjän saatavilla. *Login*-komponentin renderöimä sisäänkirjautumisnäkymä on hyvin yksinkertainen (kuva 9).



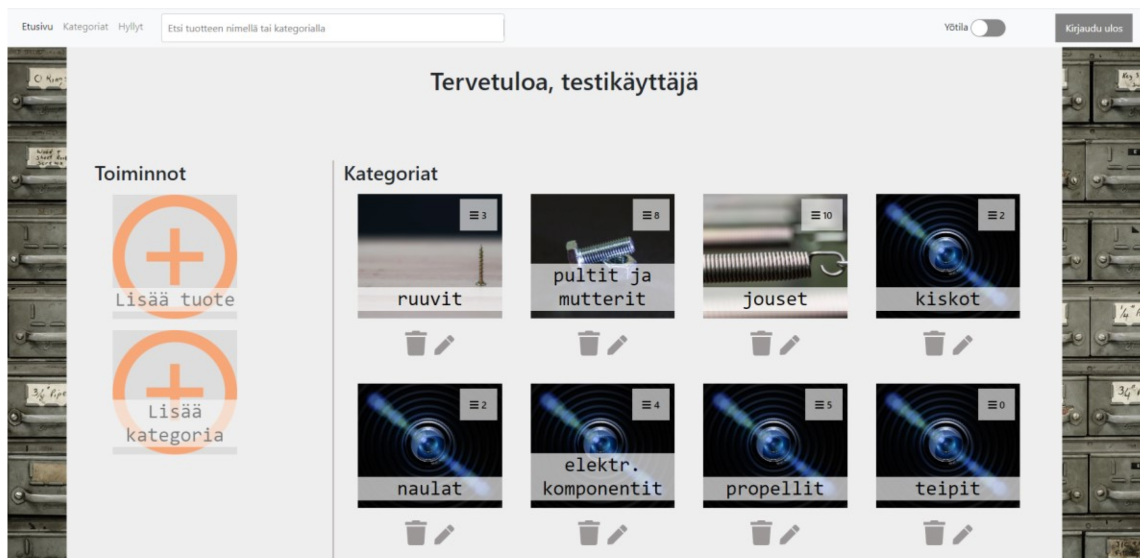

Kuva 9. Sovelluksen sisäänkirjautumisnäkymä pyytää käyttäjää syöttämään käyttäjätunnuksen ja salasanan.

Sovelluksen sisäänkirjautumisnäkymä mahdollistaa käyttäjän kirjautumisen sisään  sovelluksen toimintojen käyttämiseksi. Kirjautuakseen sisään käyttäjän tulee syöttää käyttäjätunnuksensa ja salasanansa sivulla oleviin kenttiin ja painaa *Kirjaudu sisään* -painiketta. Sisäänkirjautumisen toteutuksessa on hyödynnetty MobX-kirjastoa, joka helpottaa käyttäjän sisäänkirjautumistilan tarkkailussa (engl. observing).

Kaikki käyttöliittymässä käytetyt kuvat ovat *https://pixabay.com/*-sivustolta. *Pixabay* on sivusto, jonka sisältöä saa käyttää vapaasti *Pixabay License* -lisenssin rajoissa [24].

## 5.1 Tietojen tarkastelu

Mikäli käyttäjän painettua *Kirjaudu sisään* -painiketta löytyy tietokannan käyttäjätaulusta vastaavat tiedot käyttäjätunnukselle ja salasanalle, ohjataan käyttäjä *Main*-komponentin renderöimään sovelluksen etusivunäkymään (kuva 10).

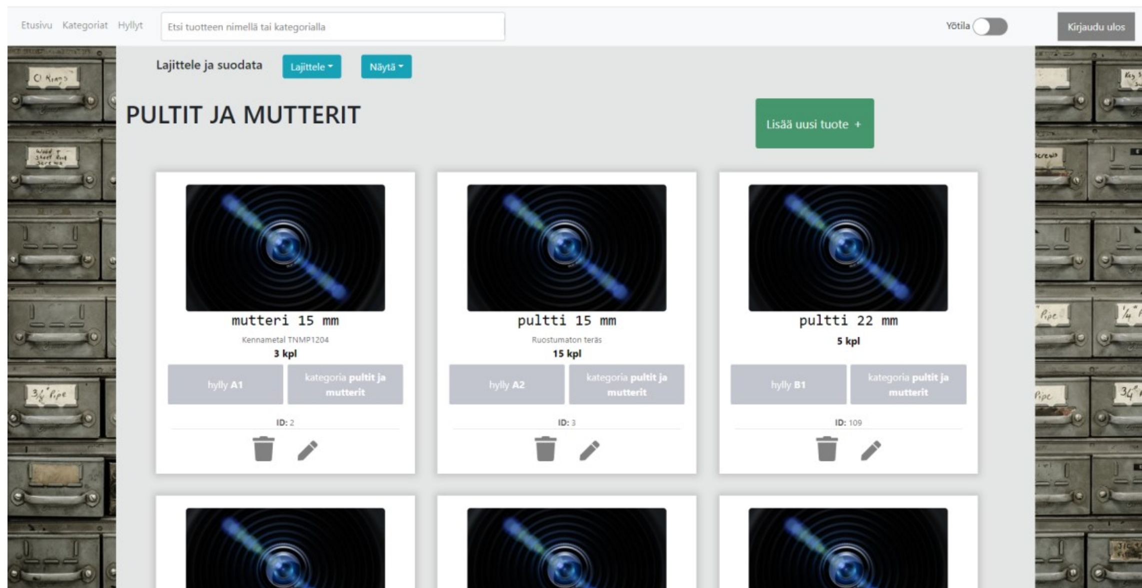



Kuva 10. Sovelluksen etusivunäkymä.

Sovelluksen etusivunäkymästä käyttäjä voi navigoida itsensä joko *Navbar*-komponentin renderöimän ylävalikon (engl. navigation bar) kautta selailemaan kate-

gorioita tai hyllyjä, hakea hakukentästä tuotteen nimellä tai kategorialla tai kirjautua ulos. Ylävalikko näkyy kaikilla sovelluksen sivuilla käyttäjän ollessa kirjautuneena sisään. Lisäksi etusivunäkymässä on painikkeet uuden tuotteen (*NewItemBox*-komponentti) ja kategorian (*NewCategoryBox*-komponentti) lisäämiselle sekä luettelo tietokantaan lisätyistä kategorioista (*Categories*-komponentti). Jokaisen kategorian kohdalla näytetään kategoriakortissa (*Category*-komponentti) tieto siitä, kuinka monta erilaista tuotetta kategoria sisältää. Jokaisen kategorian kohdalla on myös painikkeet kategorian poistamiseksi tai muokkaamiseksi. Etusivunäkymän tarkoituksena on luoda käyttäjälle nopea yleiskuva tietokannan sisällöstä ja tarjota oikotiet niihin toimintoihin, joita käyttäjä todennäköisesti tulee eniten tarvitsemaan.

Käyttäjän painaessa jotakin yksittäistä kategoriaa avautuu tuotteet-näkymä (*Items*-komponentti), jolloin käyttäjä pääsee näkemään tarkemmin kyseisen kategorian sisältämät tuotteet (kuva 11).

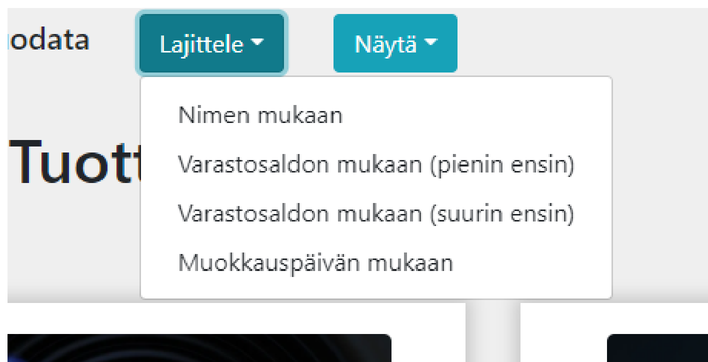



Kuva 11. Tuotteet-näkymä.

Tuotteet-näkymä päivittää käyttöliittymään ruudulle tuotteita (*Item*-komponentteja) sen mukaan, minkälaisen URL-osoitteen polun kautta se renderöidään. Kuvan 11 tilanteessa näkymä renderöi kategorian *pultit ja mutterit* tuotteet.

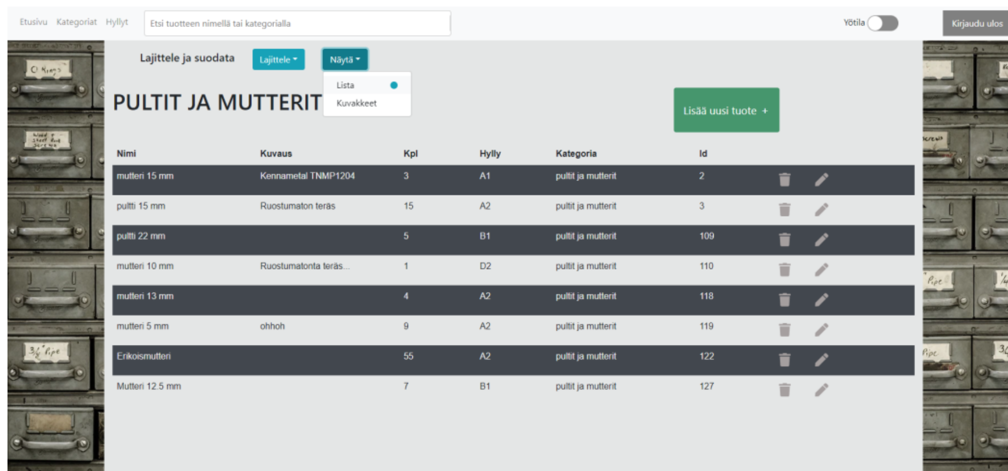
Tuotteet-näkymä renderöi joko kaikki tuotteet tai tuotteet tietystä kategoriasta tai hyllystä.

Jokaisesta tuotteesta näytetään nimi, mahdollinen lisätieto, kappalemäärä sekä hylly- ja kategoriatiedot sekä tietokantatunniste (id). Lisäksi jokaiselle tuotteelle on painikkeet poistamista ja muokkaamista varten.  Tuotteet-näkymän voi lajitella nimen, varastosaldon tai muokkauspäivän mukaisesti (kuva 12).



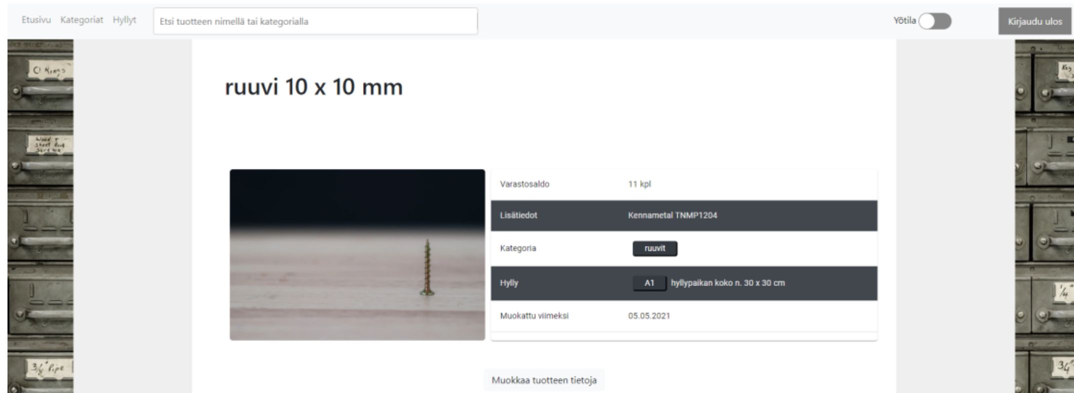

Kuva 12. Tuotteet-näkymän lajitteluvaihtoehdot.

Tuotteet-näkymässä käyttäjä voi myös päättää, haluaako nähdä tuotelistauksen kuvakkeina vai listana. Listanäkymässä tuotteiden tiedot ovat kompaktimmassa muodossa, sillä tuotteiden kuvat on jätetty pois näkymästä (kuva 13).




Kuva 13. Tuotteet-näkymässä tuotteita voi tarkastella listana tai kuvakkeina.

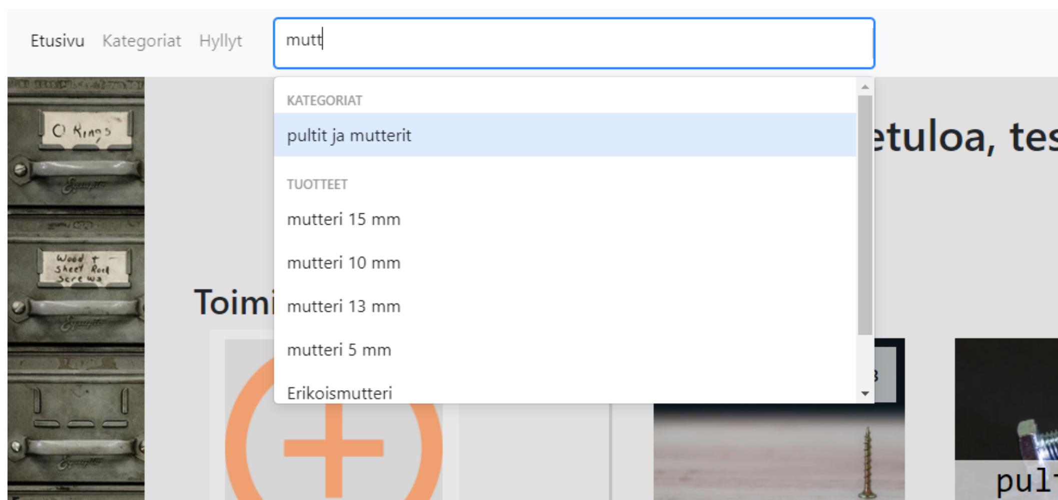
Käyttäjän painaessa jotakin riviä listanäkymästä tai tuotekorttia kuvakenäkymästä avautuu tuotenäkymä (kuva 14).




Kuva 14. Tuote-näkymä sisältää tuotteen tietoja.

Tuote-näkymä (*SingleItem*-komponentti) näyttää käyttäjälle tuotteesta hieman tarkempia tietoja tuotteet-näkymään verrattuna. Tuote-näkymässä tuotteen kuva näkyy isompana ja hyllystä kerrotaan myös mahdollinen lisäkuvaus.

Jotta sovelluksen käyttö olisi mahdollisimman sujuvaa, käyttäjä voi ylävalikon hakukentän avulla hakea tietokannasta löytyviä tuotteita tai kategorioita. Haun tulokset näytetään käyttäjälle samalla, kun hakutermin kirjoitus tapahtuu, ja tuloksissa kategoriat ja tuotteet on ryhmitelty erikseen (kuva 15).

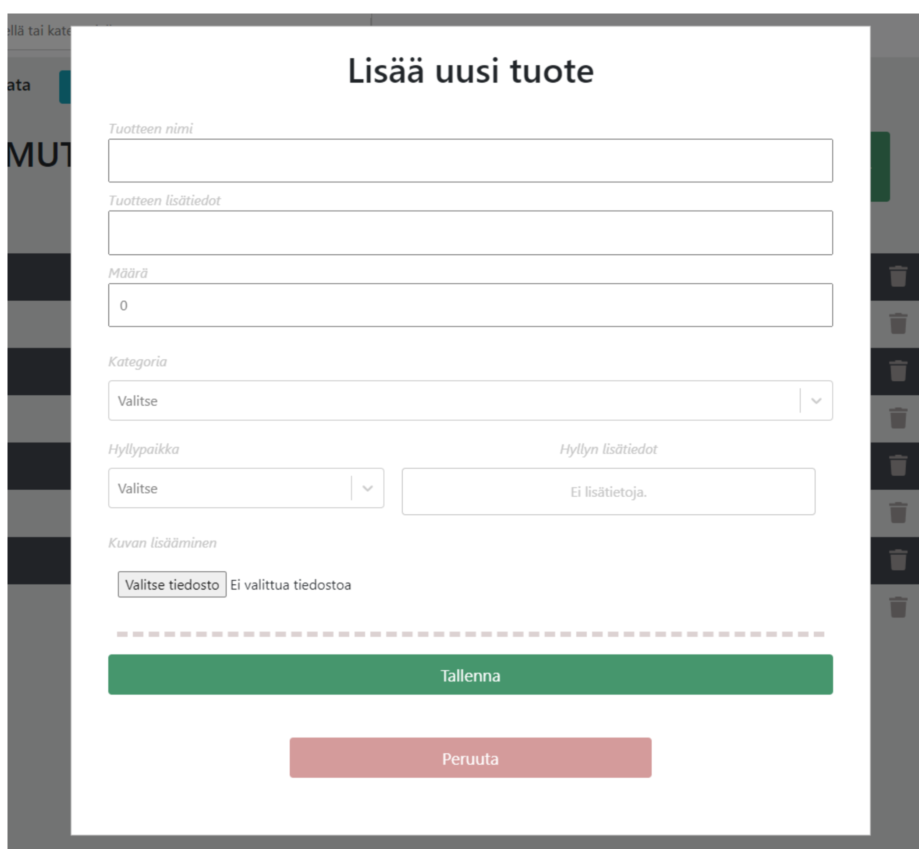



Kuva 15. Hakukentän tulokset ryhmitellään kategorioihin ja tuotteisiin.

Mikäli käyttäjä painaa hakutuloksista jotakin kategoriaa, renderöidään kuvassa 11 esitetty tuotteet-näkymä, joka sisältää kyseiseen kategoriaan kuuluvat tuotteet. Käyttäjän painaessa hakutuloksista jotakin tuotetta avautuu kuvassa 14 esitetty tuotenäkymä.

## 5.2 Tietojen lisääminen, muokkaaminen ja poistaminen

Tietojen lisääminen, muokkaaminen ja poistaminen tapahtuu painikkeiden kautta. *Lisää uusi tuote* -painike löytyy etusivunäkymästä ja tuotteet-näkymästä. *Lisää uusi tuote* -painikkeen painaminen renderöi *NewItemForm*-komponentin sisältämän lomakkeen sivuston ylle avautuvaan modaaliin (kuva 16).




Kuva 16. Uuden tuotteen lisääminen tapahtuu lomakkeen avulla.

Lomakkeen kentät täytettyään käyttäjä saa tallennettua tuotteen tietokantaan painamalla *Tallenna*-painiketta. *Peruuta*-painikkeen painaminen sulkee modaalin muutoksia tallentamatta.

Uuden tuotteen lisäämisessä ja jo olemassa olevan tuotteen muokkaamisessa käytetään samaa *NewItemForm*-komponenttia, mutta komponentille annettavat argumentit eroavat keskenään. Tuotteen muokkauksessa komponentille annetaan argumentteina tuotteen nimi, lisätiedot, kappalemäärä, kategoria, hyllypaikka ja kuvan osoite, ja komponentti lisää nämä tiedot lomakkeelle valmiiksi, jotta käyttäjä voi niitä helposti tarvittaessa muokata. Myös lomakkeen ohjetekstit eroavat toisistaan sen mukaan, mitä käyttäjä on tekemässä. Tuotteen lisäämistilanteessa näkyvä otsikko *Lisää uusi tuote* vaihtuu tuotteen muokkaamistilanteessa tekstiksi *Muokkaa tuotetta* (kuva 17).

Kuva 17. Tuotteen tietojen muokkaaminen tapahtuu lomakkeen avulla.

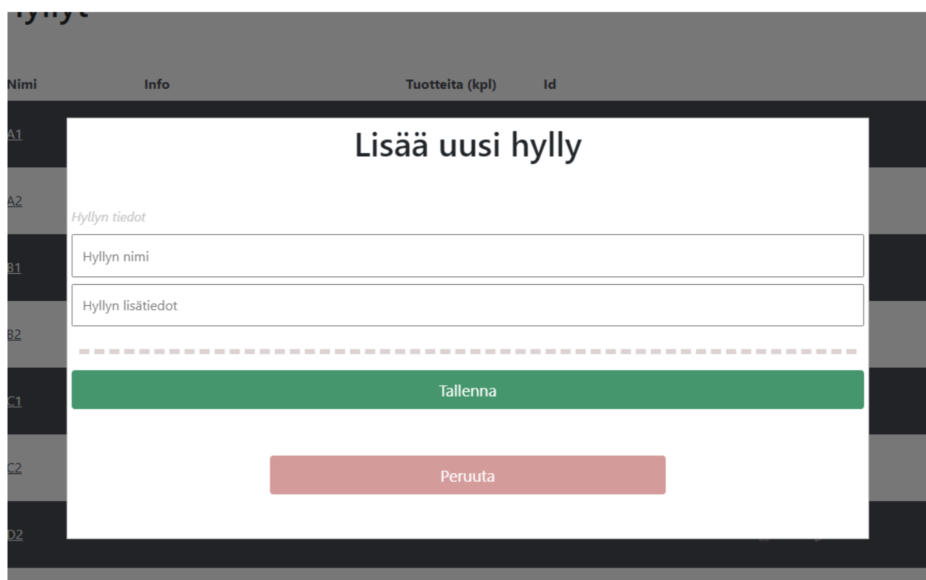
Myös uuden kategorian lisäämisessä (kuva 18) käytetään samaa *NewCategoryForm*-komponenttia kuin jo olemassa olevan kategorian tietojen muokkaamisessa, ja argumentteja hyödynnetään samalla logiikalla kuin *NewItemForm*-komponentissakin.

Kuva 18. Uuden kategorian lisääminen tapahtuu lomakkeen avulla.

Uuden hyllyn lisääminen tapahtuu hyllyt-näkymästä (kuva 19).

Kuva 19. Hyllyt-näkymä sisältää painikkeen uuden hyllyn lisäämiselle.

*Lisää uusi hylly* + -painiketta painamalla avautuu modaalissa lomake (kuva 20).

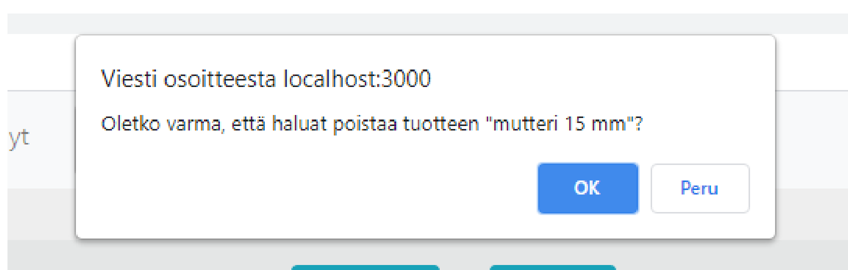



Kuva 20. Hyllyn lisääminen tapahtuu modaalin avulla.

Hyllyn lisäämiseen tarkoitetun lomakkeen renderöinnistä vastaa *NewShelfForm*-komponentti, jonka toimintalogiikka on samanlainen *NewItemForm*- ja *New-CategoryForm*-komponenttien kanssa.

Kuvassa 17 esitetyn tuotteen muokkaamiseen tarkoitetun lomakkeen avulla tuotteen varastosaldoa eli kappalemäärää voi vähentää tai lisätä. Tuotteen varastosaldon vähentäminen nollaan ei kuitenkaan poista koko tuotetta tietokannasta vaan muokkaa vain tuotteen tietoja.

Sen sijaan tuotteen poistaminen poistaa koko tuotteen tietokannasta. Tuotteen poistaminen tapahtuu tuotteet-näkymän roskakori-ikonia painamalla. Tällöin käyttäjältä varmistetaan, että tämä todella haluaa poistaa tuotteen (kuva 21).

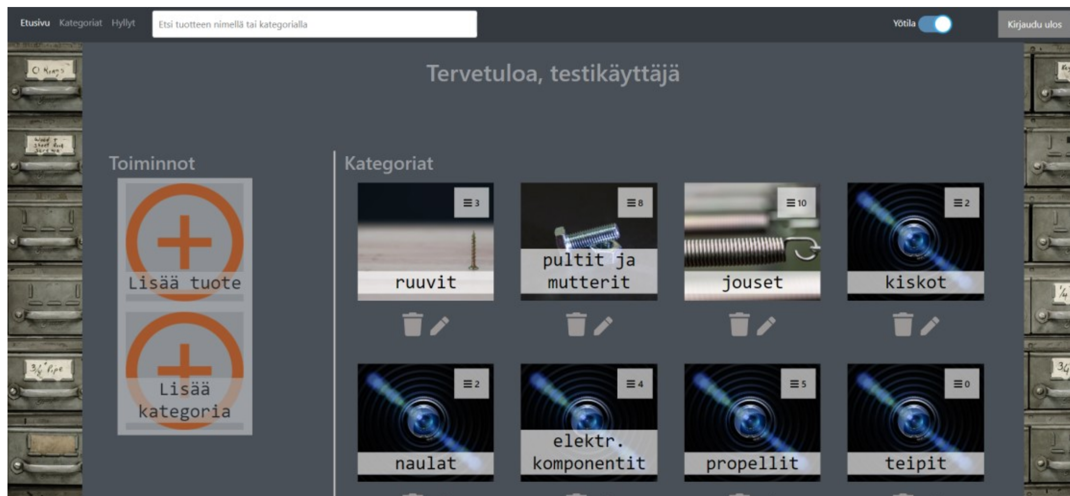



Kuva 21. Tuotteen poistaminen tietokannasta varmistetaan käyttäjältä.

Kategorian ja hyllyn poistaminen tapahtuu myös roskakori-ikonien kautta, ja poistaminen varmistetaan käyttäjältä samalla tavalla kuin tuotteen poistamisen yhteydessä.

## 5.3 Muut toiminnot

Työn tilaajan toivomuksesta sovellukseen lisättiin myös mahdollisuus yötilan käyttöön (kuva 22).




Kuva 22. Sovelluksen yötila on valittuna.

*Yötila*-komponentin avulla sovelluksen yleisilme muuttuu tummemmaksi ja hämärässä tilassa silmille ystävällisemmäksi kuin alkuperäinen vaalea ulkoasu. Yötila vaikuttaa sekä tekstin että taustan väriin.

# 6 Jatkokehityskohteet

Tässä insinöörityössä toteutettu sovellus palvelee alussa määriteltyjä vaatimuksia hyvin. Sovellusta tullaan kuitenkin tulevaisuudessa jatkokehittämään, sillä insinöörityön alussa määritellyt valinnaiset käyttäjätarinat jäivät toteuttamatta.

Sovelluksen käyttö vaatii käyttäjän sisäänkirjautumisen, ja olisi toivottavaa, että salasanan unohtuessa käyttäjä voisi esimerkiksi sähköpostin avulla vaihtaa salasanansa uuteen. Tätä varten tulisi tietokannan käyttäjätauluun lisätä sähköpostisarake, johon käyttäjän sähköpostiosoite voitaisiin tallettaa. Käyttäjän sähköpostia olisi mahdollista hyödyntää myös esimerkiksi tilanteessa, jossa järjestelmästä lähtisi viestinä yhteenveto käyttäjälle loppumaisillaan olevista tuotteista.

Lisäksi uuden käyttäjän rekisteröinti ei onnistu tällä hetkellä muutoin kuin manuaalisesti tietokantaan käyttäjiä lisäämällä. Tämäkin olisi helppoa toteuttaa etusivulle lisättävän *rekisteröidy*-painikkeen avulla.

Yksi työn tilaajan oleellisista toiveista – keräilylista – jäi puuttumaan sovelluksesta. Työn tilaaja toivoi, että tuotteita voisi lisätä keräilylistalle samalla tavalla kuin esimerkiksi useimmissa verkkokaupoissa ostoksia lisätään ostoskoriin. Keräilylistaa voisi hyödyntää esimerkiksi isompien projektien aloituksessa niin, että kaikki projektissa tarvittavat tuotteet voisi lisätä keräilylistalle, ja lopuksi käyttäjän painaessa esimerkiksi *keräile valitut tuotteet* -painiketta listalla olevat tuotteet poistettaisiin tietokannasta. Työn tilaaja toivoi myös, että tuotteen ja hyllyn voisi merkitä kuuluvaksi tiettyyn projektiin. Tätä varten tulisi tietokantaan lisätä projektitaulu, jonka perusavain toimisi tarvittaessa tuotetaulun ja hyllytaulun viiteavaimena.

Lisäksi tuotteille voisi olla viisasta lisätä esimerkiksi QR- tai viivakoodi. Tällöin tuotteiden nimiä ei tarvitsisi muistaa vaan tuotehaun voisi suorittaa lukemalla koodin.

Mikäli sovellusta halutaan jatkokehittää niin, että sen käyttö myös yritystason varastonhallinnassa olisi mahdollista, olisi tietokantaan hyvä lisätä käyttäjille erilaisia rooleja. Esimerkiksi työntekijä voisi vain kasvattaa ja vähentää tuotteiden lukumääriä, mutta tuotteiden täydellisen poistamisen tietokannasta voisi suorittaa vain henkilö, jolla on admin-tunnukset sovellukseen kirjautumista varten.

# 7 Yhteenveto

Tämän insinöörityön tavoitteena oli suunnitella ja toteuttaa varastonhallintaohjelmisto, jonka avulla käyttäjä voi ylläpitää ja tarkastella tietoa omistamistaan elektroniikka- ja mekaniikkakomponenteista ja täten säästää aikaa ja resursseja. Ohjelmiston kehitykseen haluttiin valita sellaisia teknologioita, joiden modulaarisuus ja ajanmukaisuus mahdollistaisivat ohjelmiston pitkäikäisyyden ja mutkattoman jatkokehityksen tulevaisuudessa.

Projekti aloitettiin ohjelmiston toiminnallisuuksien perusteellisella määrittelyllä. Tämän tuloksena saatujen käyttäjätarinoiden pohjalta voitiin alkaa suunnitella tarkemmin tietokantaratkaisua, käyttöliittymää ja ohjelmiston yleistä arkkitehtuuria.

Projektin back-end-teknologioina hyödynnettiin JavaScriptin Node.js-ajoympäristöä ja Express.js-sovelluskehystä. Front-end-teknologiaksi valittiin JavaScriptin React.js-kirjasto. Tietokantana päädyttiin käyttämään avoimen lähdekoodin MariaDB Server -tietokantaa.

Projektin alkuvaiheessa määritellyt käyttäjätarinat ohjasivat ohjelmistokehitystä laajalti. Osa käyttäjätarinoista tarkentui ja niitä syntyi myös muutamia lisää työn tilaajan päästessä käyttämään sovellusta kehitystyön edetessä. Jokaisesta käyttäjätarinasta määriteltiin, onko sen kuvaileman toiminnallisuuden toteuttaminen pakollista sovelluksen käytön kannalta vai voiko toiminnallisuus jäädä odottamaan jatkokehitystä.

Jokainen pakolliseksi määritelty käyttäjätarina saatiin toteutettua, ja insinöörityössä toteutettu sovellus toimii vaaditulla tavalla. Valinnaisiksi määritellyt käyttäjätarinat jätettiin odottamaan jatkokehitystä.

## Lähteet


1 User Story. Verkkoaineisto. <https://www.productplan.com/glossary/user-story/>. Luettu 27.5.2021.

2 Banga, Swapnil. 2020. What is Web Application Architecture? Components, Models and Types. Verkkoaineisto. <https://hackr.io/blog/web-application-architecture-definition-models-types-and-more>. 16.6.2020. Luettu 15.4.2021.

3 Gillis, Alexander S. 2020. Middleware. Verkkoaineisto. <https://searchapparchitecture.techtarget.com/definition/middleware>. Maaliskuu 2020. Luettu 15.4.2021.

4 Hypertext Transfer Protocol (HTTP). Verkkoaineisto. <https://www.extrahop.com/resources/protocols/http/>. Luettu 15.4.2021.

5 Pavlutin, Dmitri. 2021. What's the Difference between DOM Node and Element?. Verkkoaineisto. <https://dmitripavlutin.com/dom-node-element/>. 5.1.2021. Luettu 19.5.2021.

6 Reconciliation. Verkkoaineisto. <https://reactjs.org/docs/reconciliation.html>. Luettu 19.5.2021.

7 Ferrari, Cesare. 2019. Working with Axios in React. Verkkoaineisto. <https://dev.to/cesareferrari/working-with-axios-in-react-540c>. 27.11.2019. Luettu 19.5.2021.

8 Team. Verkkoaineisto. <https://reactjs.org/community/team.html>. Luettu 9.4.2021.

9 Facebook / React Releases. 2013. Verkkoaineisto. <https://github.com/facebook/react/releases?after=v0.8.0>. 19.12.2013. Luettu 15.4.2021.

10 React. Verkkoaineisto. <https://stackshare.io/react>. Luettu 13.4.2021.

11 React Components. Verkkoaineisto. <https://www.w3schools.com/react/react_components.asp>. Luettu 20.5.2021.

12 Node.js – Introduction. Verkkoaineisto. <https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm>. Luettu 13.4.2021.

13 What is npm?. Verkkoaineisto. <https://www.w3schools.com/whatis/whatis_npm.asp>. Luettu 13.4.2021.

14 Registry. Verkkoaineisto. <https://docs.npmjs.com/cli/v7/using-npm/registry>. Luettu 20.5.2021.

15 Cyren, Tierney. 2017. An Absolute Beginner’s Guide to Using npm. Verkkoaineisto. <https://dzone.com/articles/an-absolute-beginners-guide-to-using-npm-1>. 28.2.2017. Luettu 13.4.2021.

16 What is the file `package.json`?. 2011. Verkkoaineisto. <https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/>. 26.8.2011. Luettu 13.4.2021.

17 HTTP request methods. 2021. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>. 31.3.2021. Luettu 1.6.2021.

18 When and Why to Use Node.js for Backend Development. 2020. <https://clever-solution.com/when-and-why-to-use-node-js-for-backend-development/>. 10.4.2020. Luettu 13.4.2021.

19 React Training / React Router. Verkkoaineisto. <https://reactrouter.com/web/api>. Luettu 20.5.2021.

20 MariaDB 10.5.4 Release Notes. 2020. Verkkoaineisto. <https://mariadb.com/kb/en/mariadb-1054-release-notes/>. 24.6.2020. Luettu 16.4.2021.

21 MariaDB vs MySQL: Key Performance Differences. Verkkoaineisto. <https://www.guru99.com/mariadb-vs-mysql.html>. Luettu 16.4.2021.

22 Angeloma. 2020. Change the storage engine on MariaDB / MySQL. Verkkoaineisto. <https://www.osradar.com/change-the-storage-engine-mariadb-mysql/>. 15.8.2020. Luettu 16.4.2021.

23 Converting Tables from MyISAM to InnoDB. Verkkoaineisto. <https://mariadb.com/kb/en/converting-tables-from-myisam-to-innodb/>. Luettu 16.4.2021.

24 Simplified Pixabay License. Verkkoaineisto. <https://pixabay.com/service/license/>. Luettu 1.6.2021.