# Full stack web-sovellus yhtyeelle:

## verkkosivut, verkkokauppa ja maksunvälitysintegraatio

Waltteri Lehtinen

OPINNÄYTETYÖ
Kesäkuu 2025

Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

# TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

LEHTINEN WALTTERI:
Full stack web-sovellus yhtyeelle:
verkkosivut, verkkokauppa ja maksunvälitysintegraatio

Opinnäytetyö 59 sivua, joista liitteitä 5 sivua
Kesäkuu 2025

---

Opinnäytetyön tavoitteena oli toteuttaa full stack -web-sovellus pirkanmaalaisen metalliyhtyeen, Nøkian Monarkin käyttöön. Sovelluksen tarkoituksena oli koota yhtyeen keskeiset tiedot ja materiaalit yhteen paikkaan helposti saavutettavalla tavalla sekä mahdollistaa fanituotteiden myynti yhtyeen verkkokaupan kautta. Työ syntyi yhtyeen tarpeesta parantaa näkyvyyttään, ammattimaisuuttaan ja tavoittaa laajempi yleisö.

Työssä tuotettiin räätälöity web-sovellus, joka sisältää yhtyeen kotisivut, uutisosion, hallintapaneelin sisällön muokkaamiseen sekä verkkokauppatoiminnallisuuden. Maksunvälityksen toteuttamiseen integroitiin kolmannen osapuolen palvelu, Stripe, joka mahdollistaa turvallisen ja sujuvan maksukokemuksen asiakkaille. Sovellus rakennettiin modernilla MERN-teknologiakokonaisuudella: frontend toteutettiin Reactilla, backend Node.js:llä ja Expressillä, ja tietokanta sijoitettiin MongoDB Atlas -pilvipalveluun. Sovellus julkaistiin Fly.io-palvelussa.

Sovellus vastaa yhtyeen tarpeisiin keskittämällä tiedot ja tuotteet yhtenäiselle, hallitulle alustalle, parantaen sekä yhtyeen sisäisiä toimintoja että fanien käyttäjäkokemusta. Jatkokehityksen mahdollisuuksia ovat esimerkiksi uusien tuotteiden lisäämisen mahdollistaminen suoraan hallintapaneelin kautta, käyttöliittymään toteutettavat vaihtoehtoiset dark mode ja light mode -näkymät sekä uutisosioon lisättävä tuki monipuolisemmalle sisällölle, kuten kuville ja linkeille.

---

Asiasanat: full stack -kehitys, web-sovellus, verkkokauppa

**ABSTRACT**

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

LEHTINEN, WALTTERI:
A Full Stack Web Application for a Band:
Website, E-commerce, and Payment Gateway Integration

Bachelor's thesis 59 pages, appendices 5 pages
June 2025

---

The objective of this thesis was to design and implement a full stack web application for the use of the metal band Nøkian Monark from the Pirkanmaa Region. The goal was to create a centralized platform to present the band’s key information and materials in an easily accessible format, and to enable merchandise sales through an integrated e-commerce solution. The project originated from the band’s need to improve their visibility, professional image, and audience reach.

A customized web application was developed, featuring a public website, a news section, an administrative panel for content management, and an e-commerce functionality. The payment process was handled by integrating the third-party payment service Stripe. The application was built using the modern MERN technology stack: the frontend was implemented with React, the backend with Node.js and Express, and the database was hosted on MongoDB Atlas. The completed application was deployed on the Fly.io cloud platform.

The application meets the band's needs by providing a controlled and unified platform for their content and merchandise while also improving the fan experience. Future development possibilities include allowing the administrators to add new products directly via the management panel, implementing user-selectable dark mode and light mode themes, and expanding the news section to support richer content such as images and links.

---

Key words: full stack development, web application, e-commerce

**SISÄLLYS**

1 JOHDANTO ... 8
2 TAUSTATIETOA ... 10
2.1 Nøkian Monark -yhtye ... 10
2.2 Nøkian Monark web-sovellus ... 10
2.2.1 Sovelluksen hyödyt ... 11
2.2.2 Sovelluksen kustannukset ... 11
2.3 Full stack -web-sovelluskehitys yleisesti ... 12
2.3.1 Frontendin rooli ... 13
2.3.2 Backendin rooli ... 14
2.3.3 Tietokanta ... 14
2.3.4 Full stack -kehityksen haasteita ... 15
3 SOVELLUKSEN OMINAISUUDET JA RAKENNE ... 17
3.1 Sovelluksen tärkeimmät ominaisuudet ... 17
3.1.1 Fanituotemyynti verkkokaupassa ... 17
3.1.2 Ostoskorisivu ja sen toiminnallisuus ... 19
3.1.3 Yhtyeen uutisten ylläpito ... 21
3.1.4 Verkkokaupan inventaarion ylläpito ... 21
3.2 Sovelluksen rakenne ... 22
3.2.1 MERN-teknologiapino ... 23
3.2.2 Frontendin rakenne ... 24
3.2.3 Backendin rakenne ... 26
3.2.4 Tietokannan ja datan rakenne ... 29
4 SOVELLUKSEN TOTEUTUS ... 31
4.1 Tietoturvan ja tietosuojan toteutus sovelluksessa ... 31
4.2 Sovelluksen tärkeimmät riippuvuudet ... 33
4.3 Middleware ... 35
4.3.1 Middleware Expressissä ... 36
4.3.2 Middlewarejen käyttö sovelluksessa ... 37
4.4 Toteutusvaiheen keskeiset haasteet ja ratkaisut ... 39
4.4.1 Frontend – haasteet ja ratkaisut ... 40
4.4.2 Backend – haasteet ja ratkaisut ... 42
4.4.3 Tietokanta – haasteet ja ratkaisut ... 43
5 STRIPE-INTEGRAATIO ... 46
5.1 Stripe yleisesti ... 46
5.2 Hinnoittelu ja kustannusarvio ... 47
5.3 Maksuprosessin kuvaus ... 48

5.4 Stripe-integraation toteutus ... 49
6 JOHTOPÄÄTÖKSET JA POHDINTA ... 52
LÄHTEET ... 54
LIITTEET ... 55
Liite 1. Bändiuutisen eli postauksen Mongoose -skeema ... 55
Liite 2. Tuotteen Mongoose -skeema ... 56
Liite 3. Käyttäjän Mongoose -skeema ... 59

## LYHENTEET JA TERMIT

| | |
|---|---|
| API | Application Programming Interface |
| backend | sovelluksen palvelinpuoli |
| CRUD | Create, Read, Update, Delete – tietokantaoperaatioita |
| CSS | Cascading Style Sheets – tyylimäärittelykieli |
| debuggaaminen | ohjelmiston virheiden etsintää ja korjaamista |
| express | viitekehys backendin kehittämiseen |
| frontend | sovelluksen käyttöliittymä – näkyy loppukäyttäjälle |
| hash | kryptografinen tiiviste mm. salasanojen turvaamiseen |
| hook | Reactissa käytettävä funktio, joka lisää komponentteihin ominaisuuksia kuten tila tai elinkaarikäsittely |
| hostauspalvelu | palvelu, jonka kautta sovellus on saatavilla verkossa |
| HTML | HyperText Markup Language |
| HTTP | HyperText Transfer Protocol, verkkoviestinnässä käytettävä protokolla |
| HTTPS | HTTP:n salattu versio – suojaa tiedonsiirron |
| JavaScript | selainten tukema ohjelmointikieli |
| JSON | JavaScript Object Notation – tiedon esitysmuoto |
| JWT | JSON Web Token – turvallinen tapa käyttäjän todentamiseen ja valtuutukseen |
| localStorage | selainpohjainen tallennusratkaisu tiedon pysyvään säilyttämiseen paikallisesti |
| markdown | merkintäkieli – käytetään dokumenttien muotoiluun |
| MERN | MongoDB, Express, React ja Node.js – teknologiapino |
| middleware | backendin ohjelmakomponentti – käsittelee pyyntöjä ennen niiden päätymistä varsinaiselle reittikäsittelijälle |
| NPM | Node Package Manager – projektin riippuvuuksien hallintaohjelma |
| NoSQL | tietokanta, joka ei noudata relaatiomallia |
| Node.js | ajoympäristö – mahdollistaa JavaScriptin suorittamisen palvelinpuolella |
| React | Metan kehittämä JavaScript-kirjasto frontendien kehittämiseen |

| | |
|---|---|
| repositorio | versionhallinnassa käytetty säilytyspaikka lähdekoodille ja sen versiohistorialle, esim. GitHubissa |
| REST | Representational State Transfer, arkkitehtuurityyli – käytetään verkkopalveluiden suunnittelussa |
| RESTful | REST periaatetta noudattava ohjelmointirajapinta |
| SDK | Software Development Kit – ohjelmistokehityspaketti, joka sisältää tarvittavat työkalut ja kirjastot tietyn teknologian hyödyntämiseen |
| sessionStorage | selainpohjainen tallennusratkaisu, joka säilyttää tietoa paikallisesti mutta vain kyseisen istunnon ajan |
| SPA | Single Page Application – verkkosovellus, jossa ladataan vain yksi HTML-sivu, jonka sisältöä päivitetään dynaamisesti |
| SQL | Structured Query Language – relaatiotietokantojen kyselykieli |
| tietokanta | järjestelmä, johon sovelluksen data tallennetaan |
| URI | Uniform Resource Identifier – yksilöi resurssin verkossa |

# 1 JOHDANTO

Näkyvyys ja tavoitettavuus ovat keskeisiä tekijöitä, joiden avulla artistit ja yhtyeet voivat saavuttaa yleisönsä ja laajentaa fanikuntaansa. Verkkosivut tarjoavat keinon koota yhteen oleellinen tieto, materiaalit ja linkit yhtyeen toimintaan, jolloin ne ovat helposti saatavilla yhdestä paikasta.

Tämä opinnäytetyö saa lähtönsä omasta musiikkiharrastuksestani. Soitan rumpuja pirkanmaalaisessa metalliyhtyeessä nimeltään **Nøkian Monark**. Yhtyeellä on käytössä sosiaalisen median kanavia ja musiikkiamme on saatavilla yleisimmissä suoratoistopalveluissa, mutta halusimme laajentaa digitaalista läsnäoloamme rakentamalla omat verkkosivut.

Verkkokauppojen merkitys on kasvanut yhtyeille, sillä suoratoistopalveluiden tuottama ansainta on usein vähäistä ja epävarmaa. Omien verkkosivujen ja -kaupan kautta on mahdollista lisätä taloudellista omavaraisuutta sekä parantaa yhteyttä fanikuntaan. Tarkoituksena on hyödyntää verkkosivuja fanituotteiden myyntiin sekä uutisten ja muiden sisältöjen jakamiseen.

Sivut olisi ollut mahdollista toteuttaa valmiilla palveluilla, kuten WordPressillä, mutta halusin syventyä moderniin web-sovelluskehitykseen ja hyödyntää tilaisuutta opinnäytetyönä. Sivujen toteuttaminen alusta asti tarjoaa mahdollisuuden oppia käytännössä laajasti frontend- ja backend-teknologioista.

Sovellukseen sisältyy verkkokauppatoiminnallisuus, jonka avulla fanit voivat ostaa yhtyeen tuotteita. Tällä hetkellä tarjolla on t-paita, mutta valikoimaa on tarkoitus laajentaa tulevaisuudessa, esimerkiksi fyysisiin julkaisuihin. Sovellukseen kuuluu myös ylläpidon käyttöliittymä, jonka kautta hallitaan uutisia ja tuotetietoja.

Henkilökohtaisena tavoitteenani on kehittää osaamistani modernissa web-kehityksessä ja verkkokaupan toteutuksessa sekä tuottaa toimiva ja viimeistelty sovellus, jota voi hyödyntää myös omassa portfoliossa ja työnhaussa. Yhtyeen nä-

kökulmasta tavoitteena on saada käyttöön selkeä, visuaalisesti miellyttävä ja helposti saavutettava verkkosovellus, joka tukee ammattimaisempaa esiintymistä ja parantaa näkyvyyttä.

Toteutuksessa käytetään modernia MERN-teknologiapinoa: käyttöliittymä rakennetaan Reactilla, palvelinpuoli Node.js:n ja Expressin avulla ja tietokanta MongoDB Atlas -pilvipalvelussa. Maksuliikenne hoidetaan kolmannen osapuolen Stripe-integraation kautta, jotta käyttäjille voidaan tarjota turvallinen ja sujuva maksukokemus.

# 2 TAUSTATIETOA

## 2.1 Nøkian Monark -yhtye

Nøkian Monark on Pirkanmaalla, Nokialla vuonna 2020 perustettu metalliyhtye. Yhtyeen musiikkityyliä voidaan kuvailla termeillä black 'n' roll – rokkaavaa ja groovaavaa, black metal -vaikutteista metallimusiikkia. Yhtyeen viidestä jäsenestä kolmella on paljon kokemusta ja historiaa suomalaiselta rock- ja metallimusiikin kentältä. Yhtyeelle musiikkia säveltää myös jäsenistä kolme ja sovituksiin osallistuu koko kokoonpano.

Joulukuussa 2021 Nøkian Monark julkaisi debyyttisinglensä Shadowman musiikkivideon kera. Helmikuussa 2023 yhtye julkaisi EP:n nimeltään Of Things That Define Us, jolta julkaistiin musiikkivideo kappaleesta The Writer's Block saman vuoden maaliskuussa. Tällä hetkellä yhtye työstää uutta musiikkia tulevaa albumia varten.

## 2.2 Nøkian Monark web-sovellus

Web-sovelluksen erottaa perinteisistä verkkosivuista se, että se ei ainoastaan esitä sisältöä, vaan tarjoaa myös interaktiivisia toiminnallisuuksia, kuten käyttäjän syötteiden käsittelyä ja tilanhallintaa. Nämä ominaisuudet vaativat taustalla ajettavaa logiikkakoodia ja usein myös tietokantayhteyksiä. Web-sovellus ei siis ole vain staattinen tietojen esityspaikka, vaan se mahdollistaa dynaamisen ja käyttäjälähtöisen vuorovaikutuksen.

Tässä opinnäytetyössä toteutettavan **NM-sovelluksen** keskeisiä interaktiivisia toiminnallisuuksia ovat verkkokauppatoiminnot, kuten tuotteiden lisääminen ostoskoriin, ostoskorin tyhjentäminen ja kassatoiminnot. Muita toiminnallisuuksia ovat muun muassa ylläpitäjän kirjautumislomake sekä hallintapaneelin CRUD-operaatiot, joilla voidaan hallita sivuston uutisia ja verkkokaupan inventaariota. Hallintapaneelissa ylläpitäjä voi esimerkiksi muokata tuotteiden lukumääriä sekä lisätä ja poistaa tuotteita tietokannasta.

### 2.2.1 Sovelluksen hyödyt

Johdannossa mainittujen hyötyjen, kuten yhtyeen sisäisen toiminnan helpottamisen, fanien osallistamisen ja laajemman yleisön saavuttamisen lisäksi hyvin toteutetut verkkosivut auttavat yhtyettä erottumaan edukseen ammattimaisemmalla vaikutelmalla. Suurella osalla pienempiä yhtyeitä ei ole omia verkkosivuja, saati fanituotteita myyvää verkkokauppaa.

Omien verkkosivujen avulla yhtye voi hallita omaa brändiään ja visuaalista ilmettään vapaammin kuin pelkkien sosiaalisen median alustojen kautta. Lisäksi keskitetty alusta mahdollistaa kaiken yhtyeeseen liittyvän sisällön kuten uutisten, keikkatietojen ja fanituotteiden esittämisen selkeästi ja yhtenäisesti. Tämä parantaa käyttäjäkokemusta ja tekee tiedon löytämisestä helpompaa ja nopeampaa.

Verkkokaupan lisääminen verkkosivuille tarjoaa yhtyeelle myös taloudellisia etuja. Verkkokaupan avulla yhtye voi tavoittaa potentiaalisia asiakkaita myös kansainvälisesti verrattuna aiempaan tilanteeseen; vain keikkojen yhteydessä tapahtuvaan fanituotemyyntiin. Samalla yhtye voi hallita itse tuotteiden lähettämistä, asiakaspalvelua ja tiedottamista, mikä mahdollistaa suoremman yhteyden fanikuntaan.

### 2.2.2 Sovelluksen kustannukset

Pienelle toimijalle, kuten tässä opinnäytetyössä käsiteltävälle yhtyeelle, verkkosivujen ylläpitokustannukset voivat nopeasti kasvaa kannattamattoman korkeiksi. Tämän vuoksi on pyritty valitsemaan edullisia mutta skaalautuvia ja helppokäyttöisiä palveluita.

Sovelluksen frontend on pakattu osaksi backendiä tuotantoversiossa ja koko paketti pyörii Fly.io:n pilvialustalla. Fly.io laskuttaa käytön perusteella ja se tarjoaa viiden dollarin edestä ilmaisia resursseja, joten jos käyttö pysyy tämän rajan alapuolella, palvelu on maksuton. Tulevien laskujen suuruus riippuu sivuston käytöstä, mikä selviää vasta sovelluksen julkaisun jälkeen.

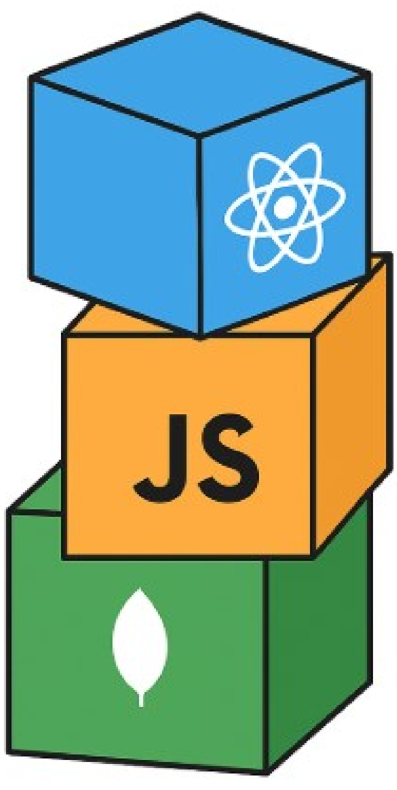
MongoDB Atlas tarjoaa pilvessä ajettavia tietokantoja useilla eri tilaustasoilla. Saatavilla on myös täysin ilmainen tilaus, joka sisältää 0,5 gigatavua tallennustilaa. Tässä projektissa valittiin tilaus, joka tarjoaa 5 gigatavua tallennustilaa. Myös Atlasissa laskutus perustuu käyttöön mutta tässä valitussa tilauksessa maksimi kuukausimaksu on 30 dollaria. Jos laskut alkavat tuntua liian suurilta, voidaan kokeilla siirtymistä tietokannan ilmaisversioon ja arvioida, riittääkö sen kapasiteetti sovelluksen tarpeisiin.

Stripen hinnoittelu on selkeä. Stripe veloittaa pienen provision jokaisesta sen kautta käsitellystä maksutapahtumasta. Stripellä ei ole kiinteitä kuluja, mikä tekee siitä hyvän vaihtoehdon pienille projekteille.

### 2.3 Full stack -web-sovelluskehitys yleisesti

**Full stack** -kehitys viittaa tyypillisesti web-sovelluskehitykseen, vaikka termiä voidaan käyttää myös esimerkiksi mobiilisovelluksista puhuttaessa. Full stack -sovellukset sisältävät vähintään kaksi pääkerrosta: käyttäjän selaimessa näkyvän osan eli frontendin sekä taustalla toimivan palvelinpuolen eli backendin. Backend huolehtii toiminnallisuuksista, kuten tietojen käsittelystä ja käyttäjäautentikoinnista. Se kommunikoi usein myös tietokannan kanssa. Tämä kerroksellinen rakenne muistuttaa pinoa (kuva 1), mistä termi ”stack” juontuu. Koko sovellus rakentuu useista eri teknologioista koostuvasta pinosta.

> Aiemmin on ollut yleisempää, että sovelluskehittäjät ovat erikoistuneet tiettyyn sovelluksen osaan, esim. backendiin. Tekniikat backendissä ja frontendissä ovat saattaneet olla hyvin erilaisia. Full stack -trendin myötä on tullut tavanomaiseksi, että sovelluskehittäjä hallitsee riittävästi kaikkia sovelluksen tasoja ja tietokantaa. Usein full stack -kehittäjän on myös omattava riittävä määrä konfiguraatio- ja ylläpito-osaamista, jotta kehittäjä pystyy operoimaan sovellustaan esim. pilvipalveluissa (Mitä on Full stack -websovelluskehitys? 2025).




KUVA 1. Full stack web-sovelluksen pinomainen rakenne (Kuva: OpenAI).

### 2.3.1 Frontendin rooli

Kuten jo mainittiin, frontend on sovelluksen näkyvä osa eli se, jonka kanssa käyttäjät ovat suoraan vuorovaikutuksessa. Frontendissä käytettävät teknologiat ovat hyvin pitkälti vakiintuneita. Ne ovat HTML, CSS ja JavaScript. Karkeasti luokiteltuna HTML:ää käytetään verkkosivujen rakenteen luomiseen, CSS:ää tyylien ja visuaalisen ilmeen määrittelyyn ja JavaScriptiä dynaamisten ja interaktiivisten osien luomiseen. Lisäksi on olemassa paljon JavaScript kirjastoja ja viitekehyksiä, kuten opinnäytetyössä käytettävä ReactJS sekä muita, muun muassa VueJS, Angular ja Svelte.

Frontend -kehitys ei ole vain ulkoasun hiomista. Siinä keskitytään myös hyvään ja intuitiiviseen käyttäjäkokemukseen, sivustolla navigoinnin helppouteen ja kaikkien käyttäjien, myös apuvälineitä käyttävien, saavutettavuuteen.

### 2.3.2 Backendin rooli

Backend toimii siis frontendin taustalla ja vastaa sovellusten taustalla tapahtuvasta logiikasta, tiedon käsittelystä ja dataliikenteen hallinnasta. Se toimii niin sanottuna välittäjänä frontendin ja tietokannan välillä, vastaanottaen pyyntöjä käyttäjän selaimelta, käsitellen ne ja palauttaen tarvittavat tiedot takaisin selaimelle. Backendin keskeisiä tehtäviä ovat esimerkiksi käyttäjäautentikaatio, liiketoimintalogiikan toteuttaminen, tietokantakyselyiden hallinta sekä ulkoisten palveluiden, kuten maksujärjestelmien ja kolmannen osapuolen ohjelmointirajapintojen eli APIen integrointi.

Koska backend toimii sovelluksen moottorina, sen suorituskyky ja skaalautuvuus ovat tärkeitä tekijöitä erityisesti suurempien järjestelmien kohdalla. Yleisiä backend teknologioita ovat opinnäytetyössä käytettävän NodeJS:n ja ExpressJS:n lisäksi muun muassa Django, Spring Boot ja Ruby on Rails, jotka tarjoavat työkaluja tehokkaaseen palvelinpuolen kehitykseen.

### 2.3.3 Tietokanta

Tietokanta on kokoelma järjestelmällisesti ja sähköisesti tallennettua tietoa eli dataa. Sen tehtävänä on mahdollistaa tiedon tehokas tallennus, haku ja hallinta. Pienet tietokannat voidaan tallentaa yksinkertaisina tiedostoina, kuten CSV- tai JSON-muodossa mutta useimmat web-sovellukset hyödyntävät tietokantapalvelimia, jotka tarjoavat strukturoidun tavan käsitellä suuria tietomääriä.

Tietokannat voidaan jakaa nykyään karkeasti kahteen päätyyppiin: relaatiotietokannat (SQL) ja NoSQL -tietokannat. Relaatiotietokannat, kuten PostgreSQL ja MySQL, tallentavat tiedon taulukoihin ja käyttävät SQL-kyselykieltä tietojenkäsittelyyn. NoSQL -tietokannat, kuten opinnäytetyössä käytetty MongoDB tarjoavat joustavamman tavan tallentaa tietoa esimerkiksi dokumenttipohjaisessa muodossa.

Web-sovelluksissa tietokannan rooli on keskeinen, sillä se säilöö käyttäjätiedot, tuotetiedot, tilaukset ja muut sovelluksen tarvitseman tiedot. Tietokanta toimii yhdessä backendin kanssa, joka huolehtii tiedon hakemisesta, käsittelystä ja päivittämisestä käyttäjän tai sovelluksen toiminnan perusteella.

### 2.3.4 Full stack -kehityksen haasteita

Full stack -kehitys on monin tavoin haastavaa ja kärsivällisyyttä vaativaa. Kolmesta sovelluskerroksesta koostuva kokonaisuus – kuten tämän opinnäytetyön aihe – voi olla etenkin vähemmän kokeneelle kehittäjälle laaja ja monimutkainen hallittava, jonka kanssa kannattaa edetä suunnitelmallisesti suurimpien sudenkuoppien välttämiseksi. Yleisiä haasteita full stack -kehityksessä ovat muun muassa virheiden selvittämisen eli debuggaamisen monikerroksisuus sekä jatkuvasti kehittyvien ja vaihtuvien ohjelmointikirjastojen, -työkalujen ja -kielten mukana pysyminen.

Kuten full stack -kehittäminen itsessään, myös tällaisen sovelluksen debuggaaminen edellyttää vahvaa kokonaisuuden hallintaa. Debuggaus on erityisen vaativaa, koska ongelmia joudutaan jäljittämään usean sovelluskerroksen läpi – joskus alueilla, joista kehittäjällä on vain pintapuolinen ymmärrys. Vaikka virhe näyttäytyisi frontendissä, sen todellinen syy voi löytyä backendistä tai tietokannasta. Full stack -debuggauksen ydin on ongelman seuraaminen kerros kerrokselta ja sen rajaaminen mahdollisimman nopeasti. Tämä ei ole helppoa, erityisesti monimutkaisissa järjestelmissä, joissa eri osat voivat vaikuttaa toisiinsa vaikeasti ennakoitavilla tavoilla.

Ohjelmistokehityksen teknologiat kehittyvät ja vaihtuvat nopeassa tahdissa – erityisesti JavaScriptin ympärillä – mikä tekee kehittäjän työstä entistä haastavampaa. Mitä useammasta sovelluksen osa-alueesta kehittäjän on oltava perillä, sitä vaikeampaa on pysyä mukana muutoksissa ja trendeissä. Tätä ilmiötä kuvaamaan on JavaScript-maailmassa vakiintunut termi *JavaScript fatigue*, eli JavaScript-väsymys.

Nämä haasteet näkyivät myös tämän opinnäytetyön toteutuksessa. Esimerkiksi virheiden paikantaminen frontendin ja backendin rajapinnan välillä vaati erityistä huomiota, runsasta *console.log* -printtailua sekä huolellista viestintää eri sovelluskerrosten välillä. Tästä huolimatta haasteet tarjosivat arvokasta oppia ja syvensivät ymmärrystä full stack -kehityksen vaatimuksista käytännössä.

# 3 SOVELLUKSEN OMINAISUUDET JA RAKENNE

## 3.1 Sovelluksen tärkeimmät ominaisuudet

Sovellus sisältää ominaisuuksia, jotka tukevat sekä yhtyeen ulkoista viestintää että fanituotteiden verkkokauppamyyntiä. Käyttäjä voi selata tuotteita, lisätä niitä ostoskoriin ja siirtyä tilaamaan ne ulkoisen maksupalvelun kautta. Ylläpitäjällä on mahdollisuus hallita uutisia ja tuotetietoja suojatun käyttöliittymän kautta.

Tässä luvussa esitellään sovelluksen keskeiset toiminnallisuudet ja niiden rooli opinnäytetyön kokonaisuudessa. Esittelyssä keskitytään erityisesti verkkokauppaan, uutisten hallintaan ja inventaarion ylläpitoon.

### 3.1.1 Fanituotemyynti verkkokaupassa

Verkkokaupassa näkyvät tuotetiedot sisältävät tuotteen nimen, kuvauksen, kategorian, materiaalin (soveltuvissa tapauksissa), hinnan, kuvan sekä joko kokovalikon tai tuotemäärän riippuen tuotetyypistä. Käyttäjälle näytetään kaikki tietokantaan lisätyt tuotteet, myös ne, joita ei ole asetettu saataville tai joista ei ole varastosaldoa jäljellä. Jos tuote ei ole saatavilla tai se on loppuunmyyty, tuotekortissa (kuva 2) näytetään teksti ”Out of stock”, eikä ostoskoriin lisäämisen painiketta tai koon valintanappeja näytetä. Jos tuotteesta on kokoja jäljellä, mutta käyttäjä ei ole valinnut kokoa, painikkeet näkyvät passivoituina. Kun käyttäjä valitsee saatavilla olevan koon, tuotteeseen liittyvät painikkeet aktivoituvat (kuva 3).

**NM crown logo t-shirt**

*A black t-shirt with bronze Nøkian Monark crown logo, made of 100 % cotton*

**Price:** 19.95 €

**Material:** cotton

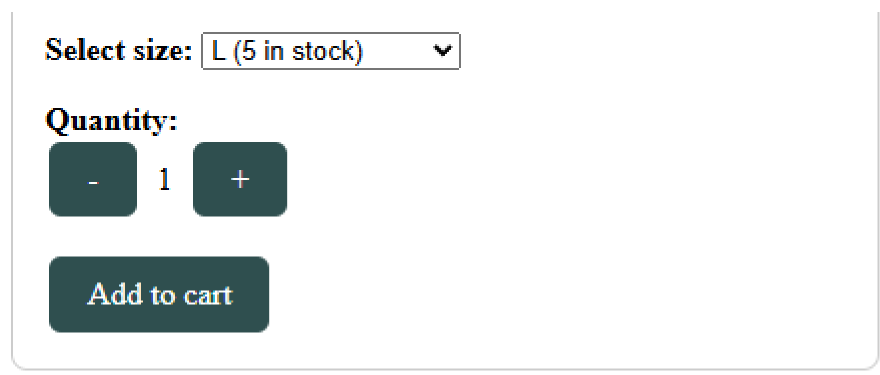
**Category:** apparel

**Select size:** --- Choose size ---

**Quantity:**
- 1 +

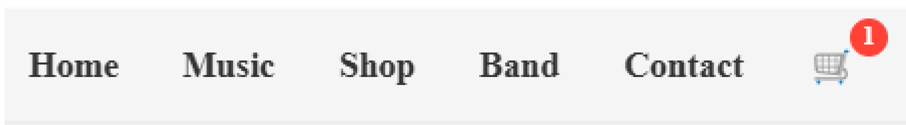
Add to cart

KUVA 2. Yksittäinen tuotekortti verkkokaupassa (Kuva: Waltteri Lehtinen).




KUVA 3. Tuotteen painikkeet aktivoituvat koon valinnan jälkeen (Kuva: Waltteri Lehtinen).

Tuotteen lisääminen ostoskoriin aktivoi sovelluksen navigointipalkissa ostoskorikuvakkeen, jonka vieressä näkyy punainen tilanilmaisin. Tilanilmaisin kertoo ostoskorissa olevien tuotteiden lukumäärän (kuva 4) ja päivittyy dynaamisesti ostoskorin sisällön mukaan. Kun ostoskori tyhjennetään, ostoskorikuvake poistuu navigointipalkista. Sovellus estää nollan tai negatiivisen määrän lisäämisen ostoskoriin sekä varastosaldon ylittävien määrien valinnan.

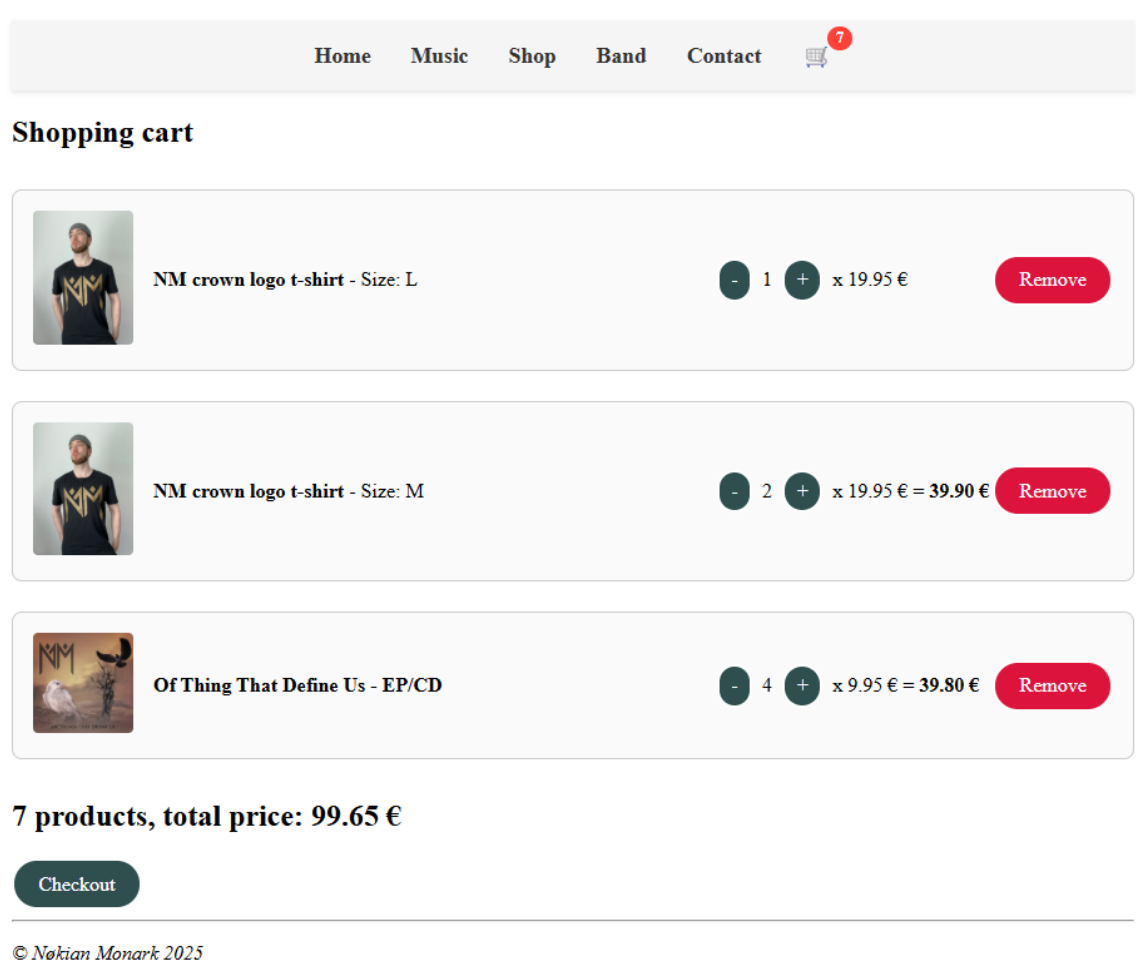



KUVA 4. Ostoskorin tilanilmaisin navigointipalkissa (Kuva: Waltteri Lehtinen).

### 3.1.2 Ostoskorisivu ja sen toiminnallisuus

Ostoskorisivulla (kuva 5), jolle siirrytään klikkaamalla ostoskorikuvaketta, näytetään kaikki ostoskoriin lisätyt tuotteet koon mukaan eriteltyinä. Jokainen tuotekokoyhdistelmä esitetään omana rivinään. Käyttäjä voi muuttaa tuotteiden kappalemääriä saatavuuden rajoissa. Jos tuotteesta on lisätty useampi kappale, lasketaan niiden yhteishinta automaattisesti.

Sivun alaosassa näytetään ostoskorissa olevien tuotteiden kokonaismäärä ja yhteissumma. Tuotteen voi poistaa ostoskorista painamalla ”Remove”-painiketta, joka on tyylitelty punaisella.




KUVA 5. Ostoskorisivun sisältö ja käyttöliittymän asettelu (Kuva: Waltteri Lehtinen).

Ostoskorisivun tiedot päivittyvät dynaamisesti, eli kaikki muutokset kappalemäärissä tai tuotteiden poistamisessa näkyvät välittömästi sivulla ilman erillistä päivittämistä. Ostoskorin tila säilyy myös istuntojen välillä: kun sovellus käynnistyy, se tarkistaa automaattisesti, onko selaimen paikallisessa tallennustilassa (localStorage) ostoskoria vastaavaa tietoa. Jos tallennettu sisältö löytyy, se asetetaan sovelluksen tilaan ja näytetään käyttäjälle sellaisenaan.

Mikäli ostoskori on tyhjä, sivulla näytetään viesti ”Your shopping cart is empty”, joka ilmaisee tilanteen selkeästi käyttäjälle. Käyttöliittymä pohjautuu korttipohjaiseen listaukseen, jossa jokainen tuote esitetään erillisenä laatikkona. Tuotekuvan, nimen ja valitun koon lisäksi kortissa näkyy tuotekohtainen määrä, yksikköhinta sekä mahdollinen yhteishinta. Sivun alaosassa näytetään tuotteiden kokonaismäärä ja loppusumma korostetusti sekä maksusivulle siirtävä ”Checkout”-painike.

### 3.1.3 Yhtyeen uutisten ylläpito

Sovelluksen etusivulla näytetään uutislista, joka sisältää yhtyeen toimintaan liittyviä ajankohtaisia tiedotteita. Uutiset esitetään aikajärjestyksessä uusimmasta vanhimpaan. Jokainen uutinen sisältää otsikon ja varsinaisen tekstisisällön. Näkyvä lista päivittyy automaattisesti aina, kun uusi uutinen lisätään tai olemassa oleva muokataan tai poistetaan.

Uutisten hallinta tapahtuu ylläpitäjälle tarkoitetun hallintapaneelin kautta. Ylläpitäjä voi lisätä uuden uutisen antamalla sille otsikon ja kirjoittamalla siihen liittyvän tekstin. Uutista voi myös muokata tai sen voi poistaa tarvittaessa. Tiedot tallennetaan tietokantaan, ja muutokset tulevat voimaan välittömästi. Hallintapaneeli on suojattu, eikä se ole julkisesti saavutettavissa ilman autentikointia.

Uutiset näytetään tällä hetkellä yksinkertaisessa tekstilistausmuodossa etusivulla, eikä erillistä uutisnäkymää ole käytössä. Kaikki uutisen sisältö näytetään heti ilman erillistä esikatselua tai korttipohjaista esitystapaa. Käyttäjälle ei näytetä uutisen ajankohtaa tai viimeisintä muokkausta, vaikka tietokanta tallentaa nämä automaattisesti. Tekstieditori ei tue muotoiluja, mutta mahdollisuus lisätä esimerkiksi ”Markdown”-tuki on huomioitu jatkokehitystarpeena.

### 3.1.4 Verkkokaupan inventaarion ylläpito

Tuotteisiin liittyviä tietoja voidaan muokata suojatun hallintapaneelin kautta. Ylläpitäjä voi avata tuotekohtaisen muokkauslomakkeen (kuva 6), jossa on mahdollisuus päivittää tuotteen nimi, kuvaus, kategoria, kuvan tiedostopolku, materiaali, hintatiedot sekä saatavuustieto. Lomakkeen avulla voidaan myös hallita varastosaldoja: joko kokonaismäärää tai tuotekohtaisia kokosaldoja tuotetyypin mukaan.

Inventaarion hallinta on keskeinen osa verkkokauppatoiminnallisuutta, sillä se määrittää, mitkä tuotteet ovat ostettavissa. Sovellus estää käyttäjää lisäämästä ostoskoriin tuotteita, joita ei ole varastossa, ja ”saatavilla”-merkintä ohjaa tuotteen

näkyvyyttä kauppasivulla. Tuotetiedot päivittyvät hallintapaneelin kautta välittömästi käyttöliittymään. Tilausprosessin automatisointia ja varastosaldojen automaattista päivittymistä suunnitellaan osaksi tulevaa jatkokehitystä.



KUVA 6. Tuotteen muokkauslomake hallintapaneelissa (Kuva: Waltteri Lehtinen).

### 3.2 Sovelluksen rakenne

Sovelluksen rakenne koostuu kolmesta pääkomponentista: käyttöliittymästä (frontend), palvelinpuolesta (backend) ja tietokannasta. Nämä osat muodostavat yhdessä toimivan kokonaisuuden, jonka avulla käyttäjä voi selata fanituotteita,

lukea uutisia ja tehdä ostoksia, ja ylläpitäjä voi hallita sisältöjä ja inventaariotietoja. Jokainen osa-alue hoitaa oman tehtävänsä ja vastaa tietyistä toiminnoista sekä tiedon käsittelystä.

Tässä luvussa esitellään, miten kukin sovelluksen osa on rakennettu ja miten ne liittyvät toisiinsa. Frontend vastaa käyttäjän näkymistä ja vuorovaikutuksesta, backend toimii välikerroksena ja käsittelee liiketoimintalogiikan, kun taas tietokanta tallentaa pysyvän tiedon tuotteista ja uutisista. Rakenne pohjautuu moderniin full stack -kehitysmalliin, jossa jokainen osa toteutetaan erillään mutta saumattomasti yhteen toimivaksi.

### 3.2.1 MERN-teknologiapino

Sovelluksen toteutuksessa käytetään MERN-teknologiapinoa, joka koostuu MongoDB-tietokannasta, Express-palvelinpuolesta, React-käyttöliittymäkirjastosta ja Node.js-ajoympäristöstä. Valinta perustuu aikaisempaan kokemukseeni kyseisestä pinosta. Olen käyttänyt sitä aiemmin opintojeni aikana. Koska tämän opinnäytetyön sovellus on laajin projektini tähän mennessä, halusin hyödyntää teknologioita, jotka ovat minulle ennestään tuttuja ja joihin voin luottaa kehitystyön aikana.

MongoDB valittiin tietokantaratkaisuksi sen dokumenttipohjaisen rakenteen vuoksi. JSON-muotoinen tieto on luontevaa käsitellä JavaScript-pohjaisessa kehityksessä, ja olen kokenut MongoDB:n rakenteet ja työkalut käytännössä sujuvammiksi kuin relaatiopohjaiset ratkaisut, kuten MySQL tai PostgreSQL. Myös Express ja Node tarjoavat tehokkaan ja modulaarisen tavan rakentaa REST-rajapintoja ja palvelinlogiikkaa, ja ne integroituvat hyvin muihin pinon osiin. React puolestaan mahdollistaa joustavan, responsiivisen ja uudelleenkäytettävän käyttöliittymän rakentamisen komponenttipohjaisesti.

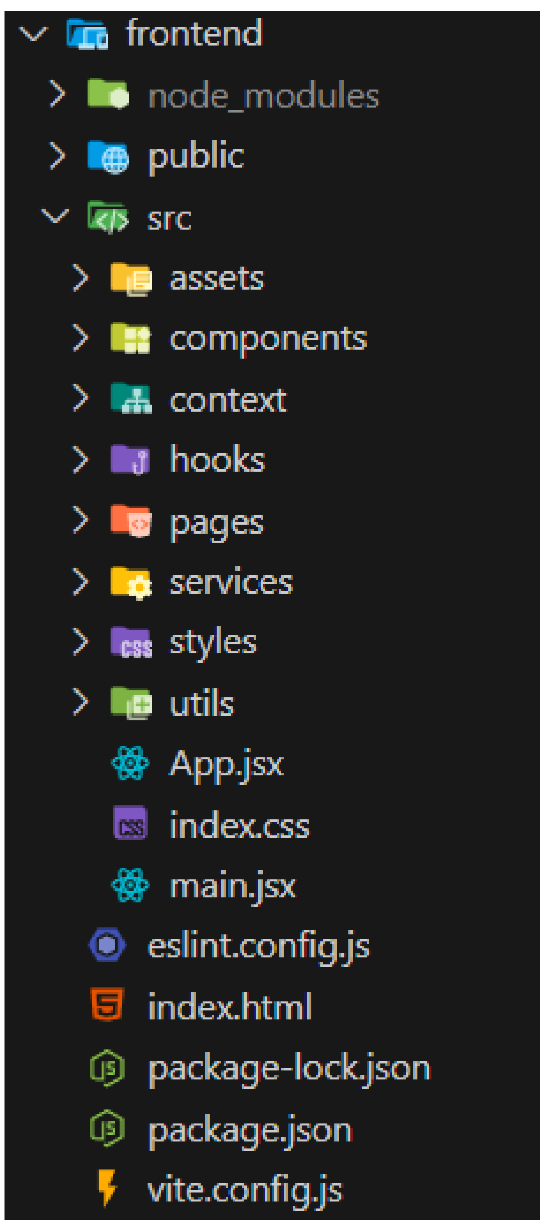
Aluksi harkitsin vaihtoehtona WordPressiä, joka olisi mahdollistanut nopeamman sivuston pystyttämisen valmiilla työkaluilla. Hylkäsin kuitenkin tämän vaihtoehdon nopeasti, koska halusin toteuttaa sovelluksen itse alusta alkaen ja syventää

osaamistani modernissa full stack -kehityksessä. PHP-ohjelmointikielen vähäinen tuntemus oli myös yksi syy luopua WordPress-pohjaisesta ratkaisusta.

MERN-pino sopii projektin tarpeisiin erinomaisesti, koska se mahdollistaa koko sovelluksen toteuttamisen yhdellä ohjelmointikielellä, JavaScriptillä. Tämä yksinkertaistaa kehitysprosessia, helpottaa ongelmanratkaisua ja nopeuttaa oppimista. Lisäksi kyseiset teknologiat ovat nykyaikaisia ja laajasti käytettyjä, mikä tekee sovelluksesta myös jatkokehityksen ja mahdollisen skaalauksen kannalta hyvin perustellun.

### 3.2.2 Frontendin rakenne

Sovelluksen frontend on rakennettu Reactilla, joka on komponenttipohjainen JavaScript-kirjasto käyttäjien kanssa vuorovaikuttavien käyttöliittymien toteuttamiseen. Sovellus jakautuu komponentteihin, joista osa on uudelleenkäytettäviä ja osa toteutettu yksittäisiä näkymiä tai toimintoja varten. Uudelleenkäytettävät komponentit parantavat koodin ylläpidettävyyttä ja selkeyttä, kun taas yksittäiset komponentit palvelevat sovelluksen ainutkertaisia tarpeita. Tämä rakenne mahdollistaa selkeän koodin organisoinnin, tehokkaan tilanhallinnan sekä helpottaa ylläpitoa ja laajennettavuutta. Frontendin kansiorakenne (kuva 7) tukee komponenttipohjaista ajattelua, ja tiedostot on jaettu loogisesti kansioihin kuten components, pages, context, services jne.




KUVA 7. Frontendin kansiorakenne komponenttipohjaisessa toteutuksessa (Kuva: Waltteri Lehtinen).

Käyttäjälle näkyvä frontend koostuu useista pääsivuista: etusivu, kauppa, ostoskori, yhteydenotto, bändin esittely, musiikki sekä ylläpitäjän hallintapaneeli. Etusivulla näytetään bändin uutisia. Shop-sivulla käyttäjä voi selailla tuotteita, valita kokoja ja määriä sekä lisätä tuotteita ostoskoriin. Ostoskorisivu tarjoaa mahdollisuuden tarkastella ostoksia, muokata määriä tai poistaa tuotteita ostoskorista. Hallintapaneelin kautta ylläpitäjä voi muokata uutisia ja verkkokaupan tuotteita. Navigaatio eri sivujen välillä tapahtuu ilman sivun uudelleenlatausta, mikä nopeuttaa siirtymiä ja säilyttää sovelluksen tilan käyttäjän näkymässä. Tämä tekee käyttöliittymästä sujuvamman ja responsiivisemman, mikä parantaa kokonaisvaltaista käyttökokemusta.

Reititys toteutetaan react-router-dom -kirjastolla, joka hallinnoi selaimen osoitepalkin URL-osoitteiden ja sovelluksen näkymien vastaavuutta. Reitityksen logiikka on määritelty App.jsx-tiedostossa, jossa käytetään <Routes>- ja <Route>-komponentteja. Sovellus tukee myös dynaamisia reittejä, kuten ylläpitäjän kirjautumissivu, jonka URL sisältää yksilöllisen UUID-tunnisteen. Reittien suojaus on toteutettu ProtectedRoute -komponentilla, joka estää pääsyn suojattuihin näkymiin ilman sovelluksen ylläpitäjän kirjautumista.

Komponenttien välinen tiedonsiirto hoidetaan pääasiassa props -parametrien ja Reactin sisäänrakennettujen hookien, kuten useState, useEffect ja useContext, avulla. Esimerkiksi ostoskorin globaali tila hallitaan CartContext -kontekstin ja useReducer -hookin avulla. Näiden avulla voidaan jakaa tila useiden komponenttien kesken ilman tarvetta nostaa sitä manuaalisesti komponenttihierarkiassa ylemmäs. Lisäksi käytössä on räätälöityjä hookeja, kuten useCart, jotka kapseloivat toistuvaa logiikkaa helpommin käytettävään muotoon.

### 3.2.3 Backendin rakenne

Sovelluksen backend eli palvelinpuoli on rakennettu Node.js:n ja Expressin avulla. Node.js toimii JavaScript-pohjaisena ajoympäristönä, joka mahdollistaa palvelinlogiikan toteuttamisen samalla ohjelmointikielellä frontendin kanssa. Express tarjoaa joustavan tavan rakentaa REST-rajapintoja ja hallita HTTP-pyyntöjä modulaarisesti. Palvelinlogiikka on jaettu selkeisiin kontrolleritiedostoihin, jotka vastaavat eri resurssien, kuten tuotteiden, postausten (eli bändiuutisten) ja ylläpitäjän toiminnallisuuksista. Mongoose-kirjastoa käytetään MongoDB-tietokannan kanssa kommunikointiin.

Sovelluksessa noudatetaan REST-arkkitehtuurin periaatteita. HTTP-metodien käyttö perustuu standardeihin, joissa esimerkiksi GET-pyynnöt ovat safe ja idempotentteja, eli ne eivät muuta palvelimen tietokannan tilaa ja niiden toistaminen ei muuta palvelimen lähettämää vastausta. PUT- ja DELETE-pyynnöt ovat idempotentteja mutta eivät safe, koska ne voivat muuttaa palvelimen tietokannan sisältöä. POST on ainoa metodi, joka ei ole safe eikä idempotentti, sillä sen toistaminen voi johtaa useiden uusien resurssien luomiseen.

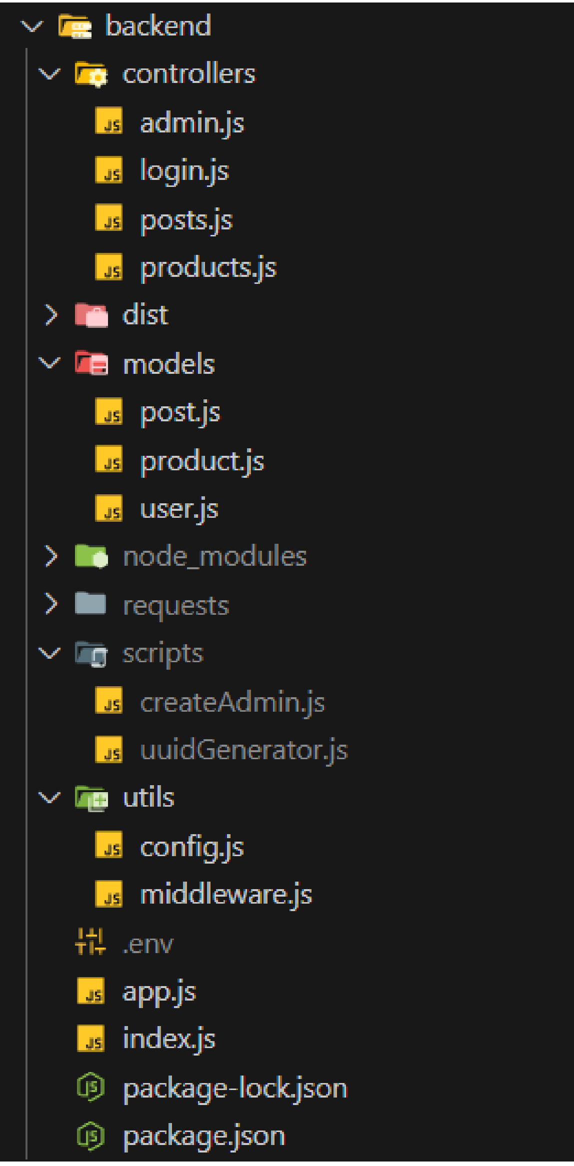
HTTP-standardin mukaisesti safe-ominaisuus tarkoittaa sitä, että pyynnön suorittamisella ei saa olla sivuvaikutuksia — esimerkiksi palvelimen tietokannan tilan muokkaamista — ja se palauttaa ainoastaan dataa. Tämä ei kuitenkaan ole teknisesti taattu, vaan toteuttajan vastuulla on huolehtia, että GET- ja HEAD-pyynnöt todella pysyvät luonteeltaan safe-ominaisuuden mukaisina. Idempotenttius puolestaan tarkoittaa, että saman pyynnön toistaminen useita kertoja tuottaa aina saman lopputuloksen. Esimerkiksi PUT-pyyntö, joka päivittää resurssin tiettyyn tilaan, saavuttaa saman lopputuloksen tietokannan tilassa riippumatta siitä, kuinka monta kertaa pyyntö toistetaan. (Huomioita HTTP-pyyntötyyppien käytöstä 2025.)

Nämä ominaisuudet vaikuttavat käytännössä siihen, miten REST-reitit on toteutettu. Sovelluksessa GET-pyynnöt on suunniteltu palauttamaan pelkkää dataa muuttamatta mitään, ja PUT- ja DELETE-pyynnöt on rakennettu idempotenteiksi. POST-pyynnöt lisäävät uusia resursseja, joten niiden mahdolliset sivuvaikutukset ja kertautuva vaikutus on tiedostettu. Tämä lähestymistapa tukee RESTful-periaatteiden mukaista ennustettavaa ja turvallista sovellusrakennetta.

REST-rajapinnat on jaettu loogisesti eri resurssien mukaan: /api/products käsittelee tuotteita, /api/posts bändiuutisia ja /api/login kirjautumista. Tuotteiden ja bändiuutisten GET-pyynnöt ovat kaikille avoimia, kun taas POST-, PUT- ja DELETE-pyynnöt on suojattu käyttäjäautentikoinnilla. Näiden kautta ylläpitäjä voi lisätä uusia tuotteita ja uutisia, muokata niitä tai poistaa tarpeen mukaan. Lisäksi käytössä on /api/admin/verify -reitti, joka tarkistaa frontendiltä tulevan UUID-tunnisteen. Reitti vertaa pyynnön mukana toimitettua UUID:ta backendin .env-tiedostossa määriteltyyn arvoon ja palauttaa tiedon siitä, onko tunniste hyväksytty. Tätä käytetään piilotetun ylläpitäjän kirjautumissivun näkyvyyden hallintaan.

Sovelluksessa hyödynnetään Expressin tarjoamia ja itse toteutettuja middleware-toimintoja, joita käsitellään tarkemmin luvussa 4.3. Ne sijaitsevat keskitetysti middleware.js -tiedostossa ja liittyvät esimerkiksi autentikointiin, virheenkäsittelyyn ja pyyntöjen lokitietojen kirjaamiseen.

Sovelluksen palvelinpuoli on jäsennelty loogiseen kansiorakenteeseen, joka tukee selkeyttä ja ylläpidettävyyttä (kuva 8). Kansio controllers sisältää eri resurssien toiminnallisuudet eriytettyinä tiedostoihin, kuten products.js, posts.js ja login.js. Kansioon models on sijoitettu Mongoose-mallit tuotteille, uutisille ja käyttäjälle. Kansiossa utils sijaitsevat muun muassa konfiguraatiotiedosto ja kaikki itse toteutetut middlewaret yhdessä tiedostossa. Lisäksi scripts-kansio sisältää irrallisia kehitystyön apuskriptejä, kuten UUID-tunnisteen luova skripti sekä ylläpitäjän käyttäjäntilin tietokantaan luova skripti. Sovelluksen käynnistyksestä vastaavat index.js ja app.js, joista viimeksi mainittu määrittelee yhteydet tietokantaan, middlewaret ja reitit. Tämä kokonaisrakenne tukee moduulipohjaista kehittämistä ja helpottaa yksittäisten osien testaamista sekä jatkokehitystä.




KUVA 8. Backendin kansiorakenne ja tiedostojen looginen jako toiminnallisuuksittain (Kuva: Waltteri Lehtinen).

### 3.2.4 Tietokannan ja datan rakenne

Sovelluksen tietokanta on toteutettu MongoDB:n avulla, joka on dokumenttipohjainen NoSQL-tietokanta. Tietokanta sijaitsee MongoDB Atlas -pilvipalvelussa, ja siihen muodostetaan yhteys palvelinpuolelta Mongoose-kirjaston avulla. Mongoose mahdollistaa selkeän skeemapohjaisen rakenteen, joka helpottaa datan validointia, käsittelyä ja muokkaamista. Sovelluksessa käytetään kolmea pääasiallista skeemaa: tuotteet, uutiset ja käyttäjä.

Postaukset eli uutiset tallennetaan Post-skeemaan (liite 1), joka sisältää kaksi pakollista kenttää: title ja content. Molemmilla kentillä on määritelty vähimmäispituus (3 merkkiä), ja skeemassa käytetään timestamps: true -asetusta, jolloin luonti- ja muokkausajankohdat tallentuvat automaattisesti. Tietoturvan ja esitysmuodon parantamiseksi skeemaan on määritelty toJSON-muunnos, jossa muun muassa MongoDB:n sisäinen _id muunnetaan id:ksi, ja versiotieto __v poistetaan.

Tuotteet määritellään Product-skeemassa (liite 2), jossa on useita validoituja kenttiä, kuten name, description, category, imageUrl, material, price ja available. Lisäksi tuotteessa voi olla joko yksittäiskappaleiden lukumäärä (quantity) tai koottain jaotellut määrät (sizes), mutta ei molempia yhtä aikaa. Tätä varten skeemaan on toteutettu oma pre("validate”) -tarkistus, joka estää virheellisen yhdistelmän luomisen. category-kenttä hyväksyy vain ennalta määritetyt arvot (”apparel”, ”media”, ”accessory”), ja hinnalle on asetettu vähimmäisarvo. Useat kentät, kuten name, description, imageUrl ja category, siivotaan automaattisesti trim-asetuksen avulla. Niiden alusta ja lopusta siis poistetaan mahdolliset white spacet eli turhat välilyönnit. Koot (sizes) tallennetaan aliskeemassa sizeSchema, jossa määritellään varastosaldot kokokohtaisesti (S, M, L, XL). Tuotteille, joilla ei ole kokoja, käytetään quantity-kenttää. Tällaisia tuotteita ovat esimerkiksi levyt tai tarrat. Asiakaskäyttöliittymässä näytetään tuotteen materiaali ja saatavilla olevat koot, mutta available-kenttä jää näkyville vain ylläpitäjän hallintapaneelissa.

User-skeema (liite 3) on yksinkertainen ja turvallisuuteen keskittyvä. Siinä määritellään pakollinen ja yksilöllinen username, jolla on vähimmäispituus, sekä passwordHash, joka tallennetaan suojatussa muodossa. Käytössä on myös toJSON-

muunnos, joka poistaa näkyvistä paitsi sisäiset kentät myös passwordHash:n, sillä sitä ei haluta lähettää eteenpäin frontendille tietoturvasyistä. Sovelluksessa on käytössä vain yksi käyttäjä, joka toimii ylläpitäjänä, eikä tietokantaan voida lisätä useita käyttäjiä.

Kaikki skeemat on suunniteltu yhteensopiviksi frontendin kanssa, ja niissä käytetään yhtenäistä toJSON-muotoilua, joka helpottaa datan käsittelyä ja esittämistä asiakaspuolella. Tietorakenne on jaoteltu resursseittain, ja validointien avulla varmistetaan, että tallennettava tieto on rakenteeltaan ja sisällöltään oikeanlaista.

# 4 SOVELLUKSEN TOTEUTUS

## 4.1 Tietoturvan ja tietosuojan toteutus sovelluksessa

Sovelluksen tietoturva perustuu useisiin käytännön tasoihin, jotka liittyvät sekä käyttäjien tunnistamiseen että tiedon turvalliseen säilyttämiseen ja siirtoon. Tietosuojaa tukee se, että järjestelmässä on vain yksi käyttäjä, jonka tunnistautumista hallitaan suojatusti palvelinpuolella. Sovelluksessa ei kerätä henkilötietoja tai käyttäjälokeja, joten tietosuojan näkökulmasta tietomäärä on hyvin rajattu ja hallittavissa.

Salasanan käsittelyssä käytetään bcrypt -kirjastoa (kuva 9), jonka avulla salasana hajautetaan toisin sanottuna hashataan ennen tallennusta. Salasanan hash luodaan erillisellä createAdmin.js-nimisellä apuskriptillä, eikä raakatekstinä olevaa salasanaa tallenneta koskaan tietokantaan. Hajautuksessa käytetään kymmentä suolauksen kierrosta (saltRounds = 10), mikä vastaa yleistä turvallisuussuositusta. Tässä sovelluksessa olisi ollut teknisesti mahdollista käyttää suurempaakin lukua, koska salasana luodaan vain kerran eikä rekisteröitymistoimintoa ole. Ei siis olisi haitannut, että salasanan hajauttaminen olisi ollut laskennallisesti raskaampikin toimenpide. Toisaalta kymmenen kierrosta tarjoaa hyvän tasapainon tietoturvan ja suoritustehon välillä, ja on laajasti käytetty arvo tuotantokäytössä.

```
const hashedPassword = await bcrypt.hash(config.PASSWORD, 10);

const adminUser = new User({
  username:
  passwordHash: hashedPassword,
});

await adminUser.save();
console.log("Admin account created successfully!");
```

KUVA 9. Ylläpitäjän käyttäjän luova ja tietokantaan tallettava koodinpätkä (Kuva: Waltteri Lehtinen).

Käyttäjän tunnistautuminen toteutetaan JSON Web Token (JWT) -menetelmällä. Kirjautumisen yhteydessä login.js -kontrolleri luo tokenin, joka sisältää käyttäjän tunnistetiedot (username, id) ja on voimassa kaksi tuntia (kuva 10). Token luodaan jsonwebtoken-kirjastolla ja allekirjoitetaan SECRET-arvolla, joka on tallennettu ympäristömuuttujana. Token välitetään clientille kirjautumisvastauksessa ja toimitetaan myöhemmissä pyynnöissä osana HTTP Authorization -headeria. Tokenia ei tallenneta tietokantaan.

```
const userForToken = {
  username: user.username,
  id: user._id,
};

// valid tokens can be generated only with the secret that
is used in the digital signing. Token is set to expire in
120 * 60 seconds (two hours)
const token = jwt.sign(userForToken, config.SECRET, {
  expiresIn: 120 * 60,
});
```

KUVA 10. Tokenin digitaalinen allekirjoitus (Kuva: Waltteri Lehtinen).

Ympäristömuuttujien käyttö estää salassa pidettävien tietojen kovakoodaamisen sovellukseen. Ympäristömuuttujat sisältävässä .env -tiedostossa säilytetään muun muassa tietokannan verkko-osoite eli URI (MONGODB_URI), paikallisen kehitysympäristön palvelimen portti, JWT-salainen avain (SECRET) sekä ylläpitäjän kirjautumissivun näkyvyyteen liittyvä ADMIN_UUID. Fly.io-alustalla nämä arvot toimitetaan sovellukselle fly secrets -toiminnallisuuden kautta, jolloin ne eivät päädy näkyviin lähdekoodiin tai julkisiin repositorioihin.

Sovelluksen kehitysvaiheessa käytetään HTTP-protokollaa yksinkertaisemman testaamisen mahdollistamiseksi. Tuotantoympäristössä Fly.io salaa kaiken liikenteen HTTPS:n kautta oletuksena, joten verkkoliikenne on suojattua loppukäyttäjän näkökulmasta. Sovelluksen frontend ja backend kommunikoivat siten salatun yhteyden yli, vaikka kehitysympäristössä käytettäisiinkin HTTP:tä.

Sovellus ei kerää henkilötietoja. Ainoa käyttäjätieto, joka tallennetaan, on username ja siihen liittyvä passwordHash. Käyttäjänimi näkyy kirjautumisen jälkeen ylläpitäjän hallintapaneelissa, mutta ei missään muussa näkymässä. Tämä täyttää tietosuojan periaatteet, kuten tarpeellisuusperiaatteen (data minimization), koska ylimääräistä henkilötietoa ei kerätä, eikä yksityisyyttä vaaranneta. Lisäksi ylläpitäjän kirjautumissivun näkyvyys on rajattu frontend-reitillä, joka vaatii UUID-tunnisteen. Tunniste tarkastetaan backendissä, ja vain oikean UUID:n kautta saapuva käyttäjä pääsee kirjautumislomakkeeseen. Tämä suojaustapa vähentää hyökkäyspinta-alaa piilottamalla kirjautumissivun satunnaisilta vierailijoilta.

Tietokantayhteys MongoDB Atlasiin on kehitysvaiheessa rajoitettu vain tiettyyn IP-osoitteeseen. Tämä suojaa tietokantaa ulkopuolisilta yhteyksiltä. Julkaisuvaiheessa IP-rajoitusta joudutaan muokkaamaan, jotta Fly.io-alustalla toimiva sovellus voi kommunikoida tietokannan kanssa. Tietokantayhteys on määritelty Mongoose-kirjastolla, ja sen URI sisältää tietokantaklusterin kirjautumistiedot, jotka on suojattu ympäristömuuttujissa.

HTTP-headerien turvakäytännöt, kuten Content Security Policy tai Helmet.js, eivät ole vielä käytössä. Tämä on mahdollinen jatkokehityskohde, jos sovelluksen käyttö laajenee tai halutaan suojautua tehokkaammin mm. XSS-hyökkäyksiä vastaan.

## 4.2 Sovelluksen tärkeimmät riippuvuudet

Sovelluksen toteutuksessa on hyödynnetty useita avoimen lähdekoodin NPM-kirjastoja, jotka tukevat sovelluksen toimintalogiikkaa, kehitystyön nopeutta sekä tietoturvaa. Riippuvuudet on valittu siten, että ne ovat hyvin dokumentoituja, yhteensopivia MERN-pinoon ja tukevat tarkoituksenmukaista työnjakoa frontendin ja backendin välillä.

**Frontendin riippuvuudet:**

- **axios**,
  käytetään HTTP-pyyntöjen lähettämiseen palvelimelle. Kirjasto tarjoaa yk-

sinkertaisen tavan käsitellä asynkronisia pyyntöjä ja vastauksia. Sovelluksessa axiosia käytetään tuotteiden ja uutisten CRUD-operaatioihin, kirjautumistoiminnon yhteydessä sekä UUID:n varmentamisessa. Axiosia ei kutsuta suoraan React -komponenteista, vaan jokaiselle resurssille (esim. postaukset, tuotteet) on oma palvelutiedosto services -kansiossa, jossa määritellään tiedonhakuun ja muokkaukseen liittyvät metodit. Tämä rakenne selkeyttää koodia, mahdollistaa tokenin liittämisen pyyntöihin keskitetysti ja edistää hyvää eriyttämisen periaatetta frontend logiikassa.

- **react-toastify,**

  vastaa käyttäjälle näytettävistä tilanneviesteistä. Sovelluksessa sitä käytetään ilmoittamaan muun muassa onnistuneesta ja epäonnistuneesta sisään- ja uloskirjautumisesta, tuotteiden ja uutisten lisäämisestä, muokkaamisesta tai poistamisesta sekä tuotteiden ostoskoriin lisäämisestä. Viestityypit vaihtelevat tilanteen mukaan: success, error, warning ja info. Toastit parantavat käyttäjäkokemusta tarjoamalla välitöntä palautetta tapahtumista.

**Backendin riippuvuudet:**

- **express**,

  sovelluksen HTTP-palvelin toteutetaan Expressin avulla. Se mahdollistaa REST-rajapintojen luomisen ja pyynnönkäsittelyn yksinkertaisesti ja joustavasti. Reitit, middlewaret ja virheenkäsittely on rakennettu Expressin tarjoaman rakenteen mukaisesti.

- **mongoose**,

  MongoDB tietokantaa hallitaan mongoose -kirjastolla, joka mahdollistaa skeemojen luonnin, tiedon validoinnin ja yhteyden hallinnan. Skeemat määrittelevät selkeästi sovelluksen resurssien rakenteen ja niihin on sisällytetty muun muassa pituusrajoitteita, pakollisuusmäärittelyjä sekä oma validointilogiikka (tuotteiden sizes ja quantity kenttien keskinäinen poissulkevuus).

- **jsonwebtoken**,

  tokenit toteutetaan tämän kirjaston avulla. Se mahdollistaa käyttäjän tunnistamisen turvallisesti ilman istunnon ylläpitoa palvelimella. Tokenit sisältävät käyttäjän tunnisteet ja vanhentuvat kahden tunnin kuluttua luomisesta.

- **bcrypt**,
  vastaa salasanojen hajauttamisesta eli hashaamisesta ennen niiden tallentamista tietokantaan sekä kirjautumisyrityksen yhteydessä syötetyn salasanan vertaamisesta oikeaan salasanaan.
- **cors**,
  kehitysvaiheessa kirjastoa käytetään mahdollistamaan frontendin ja backendin välinen viestintä eri porteista. Tuotannossa, kun frontendin build-versio tarjotaan suoraan Expressin kautta samasta domainista, cors ei ole enää tarpeellinen ja kannattaakin poistaa käytöstä, jotta hyökkäyspinta-alaa saadaan pienennettyä, tietoturvaa näin parantaen.
- **dotenv**,
  on ympäristömuuttujien hallintaan käytettävä kirjasto. Tietoturvan kannalta kriittiset arvot, kuten tietokannan URI, JWT-salaisuus sekä ylläpitäjän UUID, säilytetään .env-tiedostossa ja niitä ei kovakoodata lähdekoodiin. Dotenv on käytössä vain backendissä, sillä frontendin ajonaikaiset muuttujat ovat nähtävissä selaimesta.

### 4.3 Middleware

Middleware sai alkunsa ratkaisuna, jolla voitiin yhdistää uusia sovelluksia vanhoihin järjestelmiin ilman, että alkuperäistä koodia tarvitsi kirjoittaa uudelleen. 1980-luvulla se alkoi yleistyä ohjelmistokehityksessä, ja nykyään middleware nähdään keskeisenä osana hajautettuja järjestelmiä, joissa useat ohjelmistokomponentit keskustelevat keskenään. (What is Middleware? n.d.)

Middleware voi olla itsenäinen ohjelmisto tai ohjelmistokomponentti. Middleware toimii välitasolla sovelluksen eri osien välillä. Middleware voidaan ajatella eräänlaisena yhdistävänä siltana monimuotoisten teknologioiden, työkalujen ja tietokantojen välillä. Middleware voi käsitellä esimerkiksi autentikointia, kirjaamista (logging), välimuistia, virheenkäsittelyä ja paljon muuta – riippuen järjestelmän tarpeista. Käsite middleware on yleistynyt erityisesti verkkopalveluissa, joissa eri järjestelmäkerrokset (frontend, backend, tietokanta) kommunikoivat keskenään

HTTP-pyyntöjen välityksellä. Tällöin middleware toimii välikerroksena, joka käsittelee pyynnöt ennen kuin ne pääsevät sovelluksen ytimeen. (What is Middleware? 2025.)

### 4.3.1 Middleware Expressissä

Express on verkkokehys, joka helpottaa reititystä sekä middlewaren käyttöä. Express-sovellus on käytännössä sarja middleware funktiokutsuja. Express middlewaret ovat funktioita, joilla on pääsy pyyntö- (request) ja vastaus (response) -olioihin sekä seuraavaan middleware funktioon sovelluksen pyyntö-vastaussyklissä. Suoritusjärjestyksessä seuraava middleware merkitään yleisesti muuttujalla nimeltä ”next”. (Using middleware. n.d.)

Express middlewaret voivat suorittavat tärkeitä tehtäviä (Middleware in Express. 2025), kuten:

- Ajaa mitä tahansa ohjelmakoodia, kun pyyntö vastaanotetaan.
- Muokata pyyntö- ja vastausolioita.
- Päättää pyyntö-vastaussyklin lähettämällä lopullisen vastauksen clientille.
- Luovuttaa hallinta pinossa seuraavana olevalle middlewarelle kutsumalla next -funktiota.

Expressissä middlewaret jaetaan tyypillisesti neljään ryhmään (Middleware in Express. 2025), jotka ovat:

- **Sovellustason middleware** – otetaan käyttöön suoraan sovelluksen ylimmällä tasolla käyttäen komentoa app.use().
- **Reititason middleware** – liitetään suoraan tiettyyn reittiin tai Routeriin.
- **Virheenkäsittelymiddleware** – tunnistetaan siitä, että ne vastaanottavat neljä parametria (error, request, response, next).
- **Rakenteeseen kuuluvat kolmannen osapuolen middlewaret** – esimerkiksi cors, helmet tai body-parser.

Expressissä middlewaret muodostavat sarjan käsittelijöitä, jotka toimivat ikään kuin ketjuna. Pyyntö kulkee ensimmäisestä viimeiseen käsittelijään asti niiden

käyttöönottojärjestyksessä ohjelmakoodissa, kunnes se kohtaa lopullisen reittivastauksen tai virheen.

### 4.3.2 Middlewarejen käyttö sovelluksessa

NM-sovelluksessa hyödynnetään useita middlewareja niin käyttöoikeuksien hallinnassa, pyynnön esikäsittelyssä kuin virheidenhallinnassa. Itsetoteutetut middlewaret (taulukko 1) on toteutettu utils/middleware.js -tiedostossa ja otettu käyttöön pääosin keskitetysti sovelluksen backendin ylimmällä tasolla app.js -tiedostossa.

TAULUKKO 1. Sovelluksen itsetoteutetut middlewaret käyttötarkoituksineen.

| **Middleware** | **Käyttötarkoitus** |
|---|---|
| requestLogger | Tulostaa pyynnön tiedot konsoliin. Voidaan käyttää jatkokehityksessä lokitietojen keräämiseen. |
| tokenExtractor | Hakee Authorization -headerista JWT-tokenin ja liittää sen pyynnön kenttään ”token” Bearer-token muodossa. |
| authenticateUser | Tarkastaa, että pyynnön mukana vastaanotettu token on olemassa sekä edelleen voimassa. |
| unknownEndpoint | Palauttaa vastauksen statuskoodilla 404 (not found), jos pyynnön reittiä ei löydy. |
| myErrorHandler | Käsittelee mm. virheelliset ID:t, validointivirheet ja vanhentuneet tokenit. |

app.js -tiedostossa middlewaret otetaan käyttöön loogisessa järjestyksessä. Ensin otetaan käyttöön cors ja express.json, jonka jälkeen kaikki pyynnöt kulkevat requestLogger -middlewaren läpi. Tämän jälkeen pyyntö ohjataan oikeaan Routeriin, esimerkiksi postsRouter tai productsRouter. Reittitasolla käyttöönotettavia middlewareja sovelluksessa ovat tokenExtractor ja authenticateUser (kuva 11). tokenExtractor otetaan käyttöön kaikkilla /api/posts ja /api/products -reitteihin

osoitetuilla pyynnöillä. authenticateUser liitetään vain suojattuihin postauksia ja tuotteita koskeviin reitteihin, kuten POST-, PUT- ja DELETE-metodeihin. Näin vain tunnistautunut käyttäjä voi tehdä muutoksia tietokannan dataan.

```
const authenticateUser = (request, response, next) => {
  const decodedToken = jwt.verify(request.token, config.SECRET);

  if (!decodedToken.id)
    return response.status(401).json({ error: "invalid token" });

  request.user = decodedToken;
  next(); // pass request to the protected route handler
};
```

KUVA 11. Esimerkki itsetoteutetusta middleware funktiosta - authenticateUser (Kuva: Waltteri Lehtinen).

Kaikki pyyntö-vastaussyklit päättyvät viimeistään unknownEndpoint- tai myError-Handler -middlewareihin. Ensimmäinen palauttaa statuskoodin 404, jos mitään soveltuvaa reittiä ei löydy ja jälkimmäinen käsittelee kaikki sovelluksessa mahdollisesti syntyvät virheet (ValidationError, CastError, JsonWebTokenError, MongoServerError, TokenExpiredError).

Tällä hetkellä sovelluksessa ei ole käytössä turvallisuutta parantavia middlewareja. Niiden toteuttaminen on kuitenkin korkealla jatkokehityksen prioriteettilistalla. Tulevassa kehityksessä suunnitellaan käyttöön otettavaksi rate limiting -ratkaisu, jolla voidaan estää pyyntöjä nopeuden ja/tai määrän perusteella sekä mahdollisesti Helmet -kirjasto, joka auttaa suojaamaan sovellusta HTTP-headertason hyökkayksiltä (esim. XSS).

Rate limiting on tärkeä osa verkkopalveluiden suojausta, koska sen avulla voidaan rajoittaa, kuinka monta pyyntöä yksittäinen asiakas (esimerkiksi tietty IP-osoite) saa tehdä tietyssä aikayksikössä. Sen päätarkoitus on estää automatisoidut hyökkäykset, kuten brute force -salasananmurtamisyritykset, palvelunestohyökkäykset (DoS) tai muuten epätavallisen runsas liikenne, joka voi kuormittaa palvelinta ja estää normaalia käyttöä.

Sovelluksen nykyisessä vaiheessa rate limiting voitaisiin ottaa käyttöön erityisesti kirjautumisreitillä (/api/login), jotta yksittäinen käyttäjä ei voi yrittää arvata salasanaa lähettämällä tuhansia pyyntöjä peräkkäin. Rajaus voitaisiin toteuttaa esimerkiksi niin, että tietty IP-osoite saa tehdä enintään 5 kirjautumisyritystä viidessä minuutissa. Tämän jälkeen pyyntöihin vastattaisiin virheilmoituksella ja mahdollisesti pidennetyllä viiveellä. Näin sovellus suojautuisi tehokkaammin väärinkäytöksiltä säilyttäen kuitenkin normaalikäyttäjän käyttökokemuksen sujuvana.

## 4.4 Toteutusvaiheen keskeiset haasteet ja ratkaisut

Sovelluksen toteutusvaiheessa kohdattiin useita käytännön haasteita, jotka liittyivät frontendin, backendin ja tietokannan kehittämiseen. Kehitystyö tehtiin vaiheittain, ja kussakin vaiheessa pyrittiin löytämään ratkaisuja, jotka olivat teknisesti toimivia, turvallisia ja ylläpidettäviä. Haasteet syntyivät osin valittujen teknologioiden erityispiirteistä ja osin toteutuksen luonteesta, jossa sovelluksen eri osien – kuten kirjautumistoimintojen, verkkokaupan, maksuprosessin ja tietorakenteiden – oli toimittava saumattomasti yhteen.

Toteutus edellytti muun muassa turvallista ja piilotettua kirjautumisratkaisua, monimutkaisten tuotelomakkeiden hallintaa, ostoskorin tilanhallintaa ja säilytystä sekä käyttäjäystävällisen käyttöliittymän rakentamista saavutettavuusperiaatteet huomioiden. Kehitystyön aikana siirryttiin maksullisesta MongoDB Atlas klusterista ilmaisklusteriin. Lisäksi Reactin kehitystilaan liittyvät ominaisuudet, kuten StrictMode, toivat mukanaan opettelua vaativia tilanteita.

Tässä luvussa esitellään näitä keskeisiä haasteita ja niihin löydettyjä ratkaisuja. Alaluvuissa tarkastellaan erikseen frontendin, backendin ja tietokannan näkökulmia.

### 4.4.1 Frontend – haasteet ja ratkaisut

Frontendin toteutuksessa suurimmat haasteet liittyivät tilanhallintaan, saavutettavuuteen sekä komponenttien rakenteeseen. Sovelluksen käyttäjäpuoli rakentuu Reactin komponenttipohjaisella arkkitehtuurilla ja sen käyttöliittymä sisältää muun muassa tuotesivun, ostoskorin, hallintapaneelin ja uutisnäkymän. Vaikka komponenttien jakaminen pienempiin osiin on periaatteessa tuttua, monimutkaisempien toimintojen, kuten lomake- ja ostoskorilogiikan, hallinta vaati erityistä huomiota.

Ostoskorin kehityksessä ilmeni teknisiä haasteita. Käyttäjän lisäämät tuotteet säilytettiin tilassa, joka jaettiin sovelluksen laajuisesti useContext ja useReducer -yhdistelmällä. Tilanhallinnan lisäksi ostoskorin tiedot tallennettiin selaimen localStorage -muistiin (kuva 12), jotta ostokset säilyivät myös selainistunnon päätyttyä. Tilanteen teki haastavaksi se, että tuotteet saattoivat sisältää kokovalintoja ja niille määritettyjä enimmäismääriä (maxQuantity), jotka vaihtelivat tuotekohtaisesti. Näiden yhdistäminen ostoskorin validointiin ja näkymien tilakontrolliin vaati huolellista toteutusta. Ratkaisuksi rakennettiin logiikka, joka tarkisti lisäyksen yhteydessä tuotteen saatavuuden ja määrärajoitukset ja esti virheelliset lisäykset. Lisäksi frontend näyttää käyttäjälle ilmoituksia react-toastify-kirjaston avulla esimerkiksi ostoskorin onnistuneesta päivittämisestä tai virheistä.

```
> localStorage.getItem("nm-cart")
< '[{"id":"67fa9e95eca5cf638bfea95f","name":"NM crown logo t-shirt","price":19.95,"imageUrl":"/images/NM-crown-tee-2316x3088.jpg","available":true,"maxQuantity":5,"quantity":2,"selectedSize":"L"},{"id":"67fb6beca26c385233dc77bc","name":"Of Thing That Define Us - EP/CD","price":9.95,"imageUrl":"/images/OTTDU-album-cover-1200x1200.jpg","available":true,"maxQuantity":100,"quantity":1},{"id":"67fa9e95eca5cf638bfea95f","name":"NM crown logo t-shirt","price":19.95,"imageUrl":"/images/NM-crown-tee-2316x3088.jpg","available":true,"maxQuantity":7,"quantity":1,"selectedSize":"M"}]'
```

KUVA 12. Ostoskorin tiedot selaimen localStorage -muistissa (Kuva: Waltteri Lehtinen)

Toinen merkittävä kehityshaaste liittyi saavutettavuuteen ja HTML:n semanttiseen rakenteeseen (kuva 13). Projektin alkuvaiheessa käytetty HTML ei noudattanut semanttisia periaatteita, mikä vaikeuttaa ruudunlukijoiden toimintaa ja heikentää saavutettavuutta. Kehityksen edetessä otettiin käyttöön semanttiset HTML elementit, kuten <article>, <section>, <header>, <footer>, <figure> ja

<figcaption> sekä varmistettiin otsikkotasojen ja aria-attribuuttien oikeaoppinen käyttö. Tämä paransi paitsi saavutettavuutta myös koodin luettavuutta ja ylläpidettävyyttä.

```
18    return (
19      <article className="page">
20        <header>
21          <h1>Shop</h1>
22        </header>
23
24        <section className="shop-grid">
25          {availableProducts.map((product) => (
26            <Product
27              key={product.id}
28              product={product}
29              onAddToCart={handleAddToCart}
30            />
31          ))}
32        </section>
33
34        <footer>
35          <BackToTopButton />
36        </footer>
37      </article>
38    );
```

KUVA 13. Esimerkki semanttisten HTML elementtien oikeaoppisesta käytöstä (Kuva: Waltteri Lehtinen).

Kehitysympäristön osalta haasteita aiheutti Reactin StrictMode-tila, joka kehitysvaiheessa suorittaa tietyt elinkaarikoukut kahteen kertaan. Tämä näkyi erityisesti tilanteissa, joissa komponentin useEffect-koukku aktivoitui odottamattomasti. Tilanne ratkesi, kun StrictModen aiheuttama toiminta ymmärrettiin tarkemmin.

Näiden haasteiden ratkaiseminen syvensi ymmärrystä Reactin toiminnasta ja frontend -kehityksen hyvistä käytännöistä. Tuloksena syntyi joustava ja saavutettava käyttöliittymä, joka tukee myös jatkokehitystä ja skaalautuvuutta.

### 4.4.2 Backend – haasteet ja ratkaisut

Sovelluksen backend rakennettiin Node.js:n ja Expressin avulla RESTful -periaatteita noudattaen. Kehitystyön aikana backendin toteutuksessa kohdattiin useita haasteita, jotka liittyivät erityisesti turvallisuuteen, reittien suojaamiseen ja ympäristöjen hallintaan.

Yksi merkittävimmistä haasteista liittyi turvallisen ylläpitäjän kirjautumisjärjestelmän toteuttamiseen. Sovellukseen ei haluttu kaikille näkyvää kirjautumisnäkymää tai yleisesti arvattavaa URL-osoitetta. Ratkaisuksi kehitettiin järjestelmä, jossa kirjautumissivulle pääsee ainoastaan, jos käyttäjä osaa navigoida oikeaan URL-osoitteeseen suoraan selaimen osoitepalkin kautta ja sisällyttää sen tiettyyn kohtaan oikean UUID-osoiteparametrin. Tämä UUID tarkistetaan palvelimella erillisen /api/admin/verify -reitin kautta (kuva 14). Vain, jos UUID täsmää palvelimen ympäristömuuttujissa määriteltyyn arvoon, palautetaan tieto, jonka perusteella frontend renderöi kirjautumislomakkeen. Näin sovellus toteuttaa piilotetun sisäänkirjautumistoiminnon ilman tarvetta monimutkaiselle roolinhallinnalle.

```
const adminRouter = require("express").Router();
const { ADMIN_UUID } = require("../utils/config");

adminRouter.post("/verify", (request, response) => {
  const uuid = request.body.uuid?.trim();

  if (uuid === ADMIN_UUID) return response.json({ valid: true });
  else
    return response.status(403).json({ valid: false, error: "invalid UUID" });
});
```

KUVA 14. UUID:n varmentava reitinkäsittelijä (Kuva: Waltteri Lehtinen).

Myös autentikoinnin toteutus JWT-pohjaisella ratkaisulla sisälsi haasteita. Sovelluksen kehityksen alkuvaiheissa tokeneille ei ollut asetettu vanhenemisaikaa, joten kerran luodut tokenit olivat ns. ikuisesti voimassa. Tämä on tietenkin tietoturvariski – jos token päätyisi vääriin käsiin, sitä voitaisiin käyttää niin kauan, kunnes hyökkääjän toiminta havaitaan, jos ylipäätään havaitaan. Ongelma ratkaistiin asettamalla tokeneille kahden tunnin voimassaoloaika sekä vaihtamalla tokenin

luomisen eli digitaalisen allekirjoituksen yhteydessä käytettävä secret -arvo. Toimenpide mitätöi kaikki aikaisemmalla secretillä luodut tokenit ja varmistaa uusien tokeneiden järkevän voimassaoloajan.

Reittien suojaamiseen käytetään itse toteutettuja middlewareja, kuten luvussa 4.3.2 todettiin. tokenExtractor middleware poimii pyynnön headerista JWT-tokenin ja tallentaa sen pyyntöobjektiin, minkä jälkeen authenticateUser middleware varmistaa sen aitouden ja liittää tunnistetun käyttäjän tiedot pyyntöön. Suojatut reitit, kuten postauksen tai tuotteen luonti, muokkaus ja poisto, ovat käytettävissä ainoastaan kirjautuneelle ylläpitäjälle.

Eräs keskeinen haaste liittyi ympäristömuuttujien ja käyttöympäristöjen hallintaan. Kehitysvaiheessa sovellusta ajettiin paikallisesti (localhost) ja tietokantaan yhdistettiin suoraan. Tuotantoon siirryttäessä (Fly.io-palveluun) oli huomioitava mm. CORS-rajoitukset, palvelinosoitteiden muutokset ja tietoturvakäytännöt. Ratkaisuna sovellukseen rakennettiin erillinen config -moduuli ja .env-tiedosto, joiden avulla tärkeät arvot – kuten tietokannan URI, JWT-secret ja adminin UUID – eroteltiin ja käsiteltiin turvallisesti. Fly.io:n julkaisuprosessissa .env-tiedoston arvot määritellään palvelun omiin salaisuusasetuksiin (fly secrets).

### 4.4.3 Tietokanta – haasteet ja ratkaisut

Sovelluksen tietokantaratkaisuna käytettiin MongoDB:n ilmaisversiota pilvipohjaisen Atlas-palvelun kautta. Tietokannan skeemat toteutettiin Mongoose -kirjastolla, joka tarjosi valmiit työkalut datan validointiin, automatisoituihin aikaleimoihin ja JSON-muunnoksiin. Kehitystyön aikana tietokantarakenteisiin ja niiden hallintaan liittyi useita teknisiä ja käytännöllisiä haasteita.

Tuotemallin skeemassa suurin yksittäinen logiikkaa vaatinut ratkaisu liittyi tilanteeseen, jossa osa tuotteista määriteltiin kokovalikoiman mukaan ja osa kiinteällä kappalemäärällä. Tämän rakenteen seurauksena tuotteessa käytetään joko sizes -kenttää (jossa on alaobjekti eri kokojen saldoille) tai quantity -kenttää (kokovapaissa tuotteissa). Skeemassa otettiin käyttöön erillinen sizeSchema, joka määrittelee koot (S, M, L, XL) ja niiden oletussaldot. Ala-skeemalle määriteltiin "_id":

false, jotta se ei saa turhaa tunnistetietoa MongoDB:n dokumentissa. Lisäksi määriteltiin mukautettu validointifunktio (pre("validate")), joka estää sizes ja quantity kenttien samanaikaisen käytön tuotteen datassa. Tämä paransi tietokannan eheyden valvontaa ja yksinkertaisti frontendin tilan ja näkymien hallintaa.

Skeemassa huomioitiin myös muita laadunvarmistukseen liittyviä tekijöitä. Useissa kentissä, kuten name, description, category ja imageUrl, käytettiin trim: true -asetusta, joka poistaa automaattisesti syötteen alusta ja lopusta mahdolliset ylimääräiset välilyönnit. Tämä estää virheellisen datan tallentamisen erityisesti tekstipohjaisissa lomakkeissa. category -kenttä rajoitettiin enum -arvoilla vain sallittuihin vaihtoehtoihin ja price-kentälle määriteltiin minimirajaksi nolla. Nämä ratkaisut paransivat datan luotettavuutta.

Postausten skeemassa käytettiin kehityksen alkuvaiheessa omaa manuaalista created -kenttää kuvaamaan uutisen luontiajankohtaa. Kun Mongoose -kirjaston tarjoamat automaattiset aikaleimaominaisuudet tulivat tutummiksi, tästä omasta aikaleimakentästä luovuttiin ja siirryttiin käyttämään createdAt- ja updatedAt-kenttiä, jotka muodostuvat automaattisesti jokaiselle dokumentille. Molemmille kentille title ja content määriteltiin pakollisuus (required: true) ja vähimmäispituusvaatimus (minlength: 3), mikä estää tyhjien tai merkityksettömien uutisten tallentamisen.

Käyttäjän skeemassa tietoturvaan liittyvät seikat olivat erityisen tärkeitä. username-kenttä määriteltiin yksikäsitteiseksi (unique: true), pakolliseksi ja vähintään kolmen merkin pituiseksi. passwordHash-kenttä oli alkuperin määritelty ilman validointia, mutta myöhemmin myös siihen lisättiin vaatimus (required: true), jotta voidaan varmistaa salasanahajautuksen tallentuminen jokaiselle käyttäjälle. Skeemassa käytetään lisäksi set("toJSON") -muunnosta, jolla muunnetaan MongoDB:n _id-kenttä intuitiivisempaan id -muotoon ja poistetaan vastausobjektista __v ja erityisesti passwordHash. Tämä estää salasanahajautuksen paljastumisen frontendille ja on tärkeä osa tietoturvan toteutusta. Vastaava toJSON-muunnos otettiin käyttöön myös tuotteiden ja postausten skeemoissa yhtenäisen toteutuksen varmistamiseksi.

Yksi käytännön haaste liittyi tietokannan hintatasoon. Alun perin kehitystyössä käytettiin MongoDB:n maksullista palvelutasoa, mutta tämä osoittautui nopeasti liian kalliiksi pienimuotoiseen kehityskäyttöön. Tilanne ratkaistiin siirtymällä takaisin ilmaisversioon ja luomalla uusi klusteri, johon otettiin käyttöön puhdas tietokanta. Koska tässä vaiheessa tuotetietoja ei vielä ollut talletettu tietokantaan, migraatio rajoittui muutamaan dummy-postaukseen, eikä se aiheuttanut teknisiä ongelmia.

Datan testaus ja sisältö rajattiin tietoisesti pieneksi. Postauksista valtaosa oli dummy-dataa, jota käytettiin ainoastaan kehityksessä. Tuotteiden osalta yksi T-paita oli oikea, myytävä tuote, mutta sen rinnalle luotiin kaksi testausta varten kehitettyä tuotetta, joista toinen käytti sizes-kenttää ja toinen quantity-kenttää. Näin pystyttiin varmistamaan, että skeeman logiikka ja käyttöliittymä toimivat molemmissa tapauksissa odotetusti.

Sovelluksessa ei käytetty dokumenttien välisiä viittauksia (ref), koska siihen ei ollut tarvetta. Postaukset ja tuotteet toimivat itsenäisinä kokonaisuuksina, eikä käyttäjiä ole kuin yksi – ylläpitäjä. Tämä yksinkertaisti skeemarakenteita.

Tietoturvan näkökulmasta tietokannan yhteydet rajattiin aluksi yksinomaan kehityskoneen IP-osoitteeseen. Tuotantoon siirtyessä (Fly.io-palvelun kautta) tämä rajoitus kuitenkin purettiin, jotta sovellus pystyi muodostamaan yhteyden tietokantaan. Tulevaisuudessa yhteyksien IP-rajaus olisi hyvä toteuttaa myös tuotantoympäristössä tietoturvan vuoksi, esimerkiksi Fly.io:n tarjoamien kiinteiden ulostulopalvelinosoitteiden avulla.

# 5 STRIPE-INTEGRAATIO

## 5.1 Stripe yleisesti

Stripe on kansainvälisesti toimiva finanssiteknologiayritys, jolla on juuret Irlannissa ja Yhdysvalloissa. Stripe tarjoaa erityisesti verkkomaksujen käsittelyyn suunnattuja palveluita. Yrityksen perustivat vuonna 2010 veljekset Patrick ja John Collison. Nykyään Stripe on yksi suurimmista sekä tunnetuimmista alan toimijoista maailmanlaajuisesti. Suomessa Stripe ei ole niin tunnettu kuin esimerkiksi sen kilpailija PayPal, mutta kansainvälisesti Stripe on hyvin suosittu. Se toimii jo 46 maassa ja tukee 135 valuuttaa. (About Stripe n.d.; How to accept… 2023; Murphy 2025.)

Stripe on niin sanottu kolmannen osapuolen maksupalvelu, mikä tarkoittaa, että se mahdollistaa maksujen vastaanottamisen ilman tarvetta myyjän sopimuksille pankkien tai luottokorttiyhtiöiden kanssa. Stripe toimii välikätenä loppuasiakkaan ja myyjän välillä, huolehtien korttitietojen käsittelystä, turvallisuudesta sekä maksujen välityksestä. (What is Stripe and is it trustworthy? n.d.)

Stripe on saavuttanut laajan suosion erityisesti ohjelmistokehittäjien keskuudessa. Yksi sen keskeisistä vahvuuksista on selkeä ja hyvin dokumentoitu API, jonka avulla maksutoiminnallisuuksia voidaan integroida sovelluksiin joustavasti ja tehokkaasti. Stripe tarjoaa käyttöliittymiä ja SDK-kirjastoja, jotka tukevat nopeaa kehitystä ja mahdollistavat esimerkiksi toistuvien maksujen, lahjakorttien tai kansainvälisten maksujen toteuttamisen ilman raskasta taustajärjestelmäarkkitehtuuria. (Stripe n.d.)

Palvelun turvallisuutta lisäävät muun muassa sisäänrakennetut petostentorjuntatyökalut, kuten Stripe Radar, sekä kattavat testaus- ja kehitysympäristöt. Stripe toimii PCI DSS -sertifioituna palveluntarjoajana, mikä vähentää yksittäisen kehittäjän vastuuta maksutietojen käsittelystä ja salauksesta. (Stripe n.d.; Murphy 2025.)

Stripen suosio perustuu näin ollen paitsi sen laajaan maksutapavalikoimaan ja kansainväliseen kattavuuteen, myös sen helppokäyttöisyyteen, skaalautuvuuteen ja hyvään dokumentaatioon, joiden ansiosta se soveltuu hyvin myös pienille verkkopalveluille ja kehittäjälähtöisille projekteille.

## 5.2 Hinnoittelu ja kustannusarvio

Stripe perustuu transaktiokohtaiseen hinnoittelumalliin, jossa peruskäyttöön ei sisälly kiinteitä kuukausimaksuja. Maksut muodostuvat onnistuneista maksutapahtumista veloitettavasta provisiosta ja kiinteästä osasta. Euroopan talousalueella tehdyissä maksutapahtumissa Stripe veloittaa 1,5 % + 0,25 € per maksutapahtuma, kun kyseessä on eurooppalaiselta kortilta tehty maksu. Muiden alueiden tai valuuttojen maksutapahtumista Stripe veloittaa korkeampia provisioita kuin Euroopan sisäisistä maksuista. Perusmaksu, 0,25 €, on kuitenkin sama kaikissa korttimaksuissa. (Pricing built for… n.d.)

Tämä hinnoittelumalli soveltuu hyvin pienimuotoiseen verkkokauppaan, jossa maksutapahtumien määrä on rajallinen ja vaihteleva. Koska kiinteitä kuluja ei ole, kustannukset mukautuvat myynnin määrään ilman riskiä ylikapasiteetin maksamisesta hiljaisina ajanjaksoina.

Sovellus palvelee pientä yhtyettä, jolla on aluksi vain yksi fanituote myynnissä. Jos esimerkiksi kuukaudessa myytäisin yksittäisinä ostotapahtumina kahdeksan bändipaitaa, joiden kappalehinta on 20 €, Stripe veloittaisi yhteensä 4,40 € seuraavan kaavan mukaan:

$$\begin{aligned}&(8\ ostotapahtumaa) \times (0{,}25\ € + 1{,}5\ \% \times 20\ €)\\&= 8 \times (0{,}25\ € + 0{,}3\ €)\\&= 4{,}40\ €\end{aligned}$$

Tämä antaa käsityksen siitä, että pienimuotoisessa toiminnassa kustannukset pysyvät matalina ja helposti ennakoitavina.

Kehitysvaiheessa sovellusta on testattu Stripen tarjoamassa testitilassa, jossa maksutapahtumat suoritetaan testikorttinumeroilla. Tämän vuoksi todellista maksuliikennettä tai kustannuksia ei ole vielä syntynyt. Varsinaiset kulut realisoituvat vasta, kun sovellus otetaan käyttöön ja maksut aktivoidaan tuotantoympäristössä. Jokaisesta Stripen tuottamasta kulusta syntyy samalla kuitenkin enemmän tuloja, joten Stripe sopii yhtyeen käyttötapaukseen hyvin.

### 5.3 Maksuprosessin kuvaus

Sovelluksen maksuprosessi toteutettiin hyödyntäen Stripe Checkout -palvelua, joka tarjoaa valmiiksi rakennetun ja turvallisen maksusivun loppuasiakkaalle. Prosessi käynnistyy asiakkaan siirtyessä ostoskoriin ja painaessa "Checkout"-painiketta. Tällöin ostoskorin sisältö, mukaan lukien tuotteet, määrät ja mahdolliset valitut koot, lähetetään yhdellä kertaa POST-pyyntönä palvelimen /api/checkout/create-checkout-session reitille.

Frontend ei suorita ostoskoridatan validointia ennen lähettämistä, vaan turvallisuussyistä tuotetiedot tarkistetaan ja haetaan uudelleen palvelimella. Tämä estää esimerkiksi tilanteet, joissa asiakas voisi manipuloida tuotehintaa tai tilata loppuunmyytyjä tuotteita. Turvallisuusperiaatteena maksutapahtumassa on, että maksun välityksessä luotetaan vain palvelimen hallinnassa olevaan tietoon.

Palvelin vastaanottaa ostoskorin ja hakee siihen sisältyvät tuotteet uudelleen tietokannasta yksilöllisten tunnisteiden (id) perusteella. Tietokannan tuotteiden ajantasaisia hintatietoja ja saatavuustietoja hyödyntäen palvelin muodostaa Stripe Checkout -istuntoon tarvittavan line_items -rakenteen. Jokainen tuote, mukaan lukien valittu koko (jos sellainen on), muunnetaan Stripelle sopivaan muotoon, jossa yksikköhinta ilmoitetaan sentteinä. Jos tuotteen saatavuutta ei voida vahvistaa, istunto keskeytetään virheilmoituksella.

Stripe Checkout -istunnon luonnin yhteydessä määritellään maksutavaksi korttimaksu sekä palautusreitit onnistuneen ja peruutetun maksun varalta. Jos maksu

onnistuu, asiakas ohjataan sovelluksen näkymään, jossa näkyy onnistuneen tilauksen vahvistus. Mikäli maksuprosessi keskeytyy, asiakas palautetaan takaisin ostoskoriin.

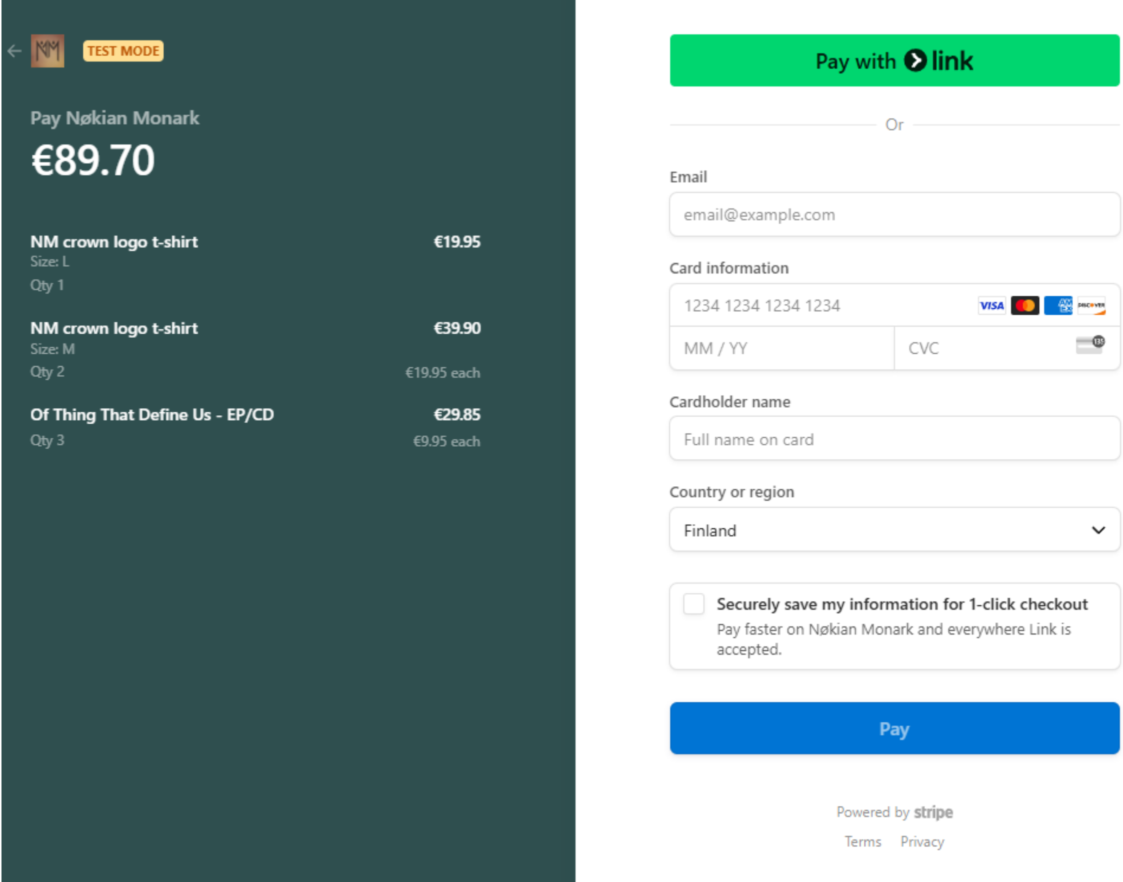
Stripe toimii PCI DSS -sertifioituna maksunvälittäjänä, ja checkout-palvelu siirtää maksutapahtuman käsittelyn täysin Stripen vastuulle. Tämä vapauttaa sovelluksen kehittäjän monimutkaisista turvallisuus- ja salaustoimista, joita maksukorttitietojen käsittely muutoin edellyttäisi.

Tämä toteutustapa mahdollistaa turvallisen maksuprosessin ilman asiakastunnuksia, hyödyntäen Stripe Checkoutin vakioituja ominaisuuksia. Samalla se rajaa mahdolliset tietoturvariskit, koska yksittäisiin hintoihin tai saatavuustietoihin ei luoteta frontendistä saapuvassa datassa.

## 5.4 Stripe-integraation toteutus

Checkout-prosessi käynnistyy frontendin CartSummary-komponentista, jossa asiakkaan ostoskorin tiedot siirretään services/checkout.js -tiedostossa määritetyn funktion kautta palvelimelle POST-pyynnöllä. Pyyntöön sisältyy koko ostoskori, jossa jokaisella tuotteella on seuraavat kentät: id, name, price, quantity, mahdollinen selectedSize sekä imageUrl.

Kun backend on luonut maksutapahtuman Stripe-palvelussa, se palauttaa maksulinkin frontendille, joka ohjaa asiakkaan Stripen ylläpitämälle maksusivulle (kuva 15). Palvelimen puolella on oma checkoutRouter, joka vastaanottaa ostoskorin ja tarkistaa sen sisällön. Kaikki ostoskorissa olevat tuotteet haetaan uudelleen Product-mallin kautta tietokannasta niiden id-arvojen perusteella. Jos yksikään tuote ei ole enää saatavilla tai id on virheellinen, maksuprosessi keskeytyy.




KUVA 15. Stripe Checkout -maksusivu (Kuva: Waltteri Lehtinen).

Stripe odottaa saavansa tuotteiden hinnat senteissä, joten unit_amount lasketaan kertomalla tuotteen hinta sadalla ja pyöristämällä lähimpään kokonaislukuun. Jokaiselle tuotteelle luodaan Stripe-yhteensopiva line_items -rakenne (kuva 16), jossa voidaan määritellä tuotteen nimi, määrä sekä valinnainen kuvaus, esimerkiksi valittu koko.

```
// generate Stripe line_items
const line_items = cart.map((item) => {
  const matched = productDocuments.find(
    (p) => p._id.toString() === item.id && p.available
  );

  if (!matched) throw new Error(`Product ID invalid: ${item.id}`);

  const name = matched.name;
  const unit_amount = Math.round(matched.price * 100);

  console.log("item.quantity =", item.quantity);
  const product_data = { name };
  if (item.selectedSize) {
    product_data.description = `Size: ${item.selectedSize}`;
  }

  return {
    price_data: {
      currency: "eur",
      product_data,
      unit_amount,
    },
    quantity: item.quantity,
  };
});
```

KUVA 16. Stripe yhteensopiva tuotteiden line_items -rakenne (Kuva: Waltteri Lehtinen).

Maksuistunto luodaan stripe.checkout.sessions.create -funktion avulla, ja sen yhteydessä määritetään hyväksytyt maksutavat (sovelluksen tapauksessa korttimaksut) sekä success_url ja cancel_url, joihin asiakas ohjataan maksun onnistuessa tai peruuntuessa. Sovelluksessa käytetään erillisiä kehitysympäristö- ja tuotanto-osoitteita Stripen maksutapahtuman luomiseen. Eli testaaminen ja lopullinen tuotantokäyttö on selkeästi eroteltu. Maksutapahtumaa on voitu testata Stripen tarjoamilla testimaksukorttitiedoilla.

## 6 JOHTOPÄÄTÖKSET JA POHDINTA

Opinnäytetyön tavoitteena oli suunnitella ja toteuttaa yhtyeelle tarkoitettu moderni web-sovellus, joka kokoaa yhteen verkkosivut, verkkokaupan sekä maksunvälitysintegraation. Työn tekniset osat toteutettiin MERN-teknologiapinolla ja maksuliikenne Stripe-integraation avulla. Kehitystyön aikana asetettiin sekä henkilökohtaisia oppimistavoitteita että yhtyeen käytännön tarpeisiin liittyviä tavoitteita.

Työn myötä saavutettiin merkittävää edistystä opiskelijan osaamisen kehittämisessä. Projektin aikana syvennyttiin muun muassa Reactin komponenttipohjaiseen kehitykseen, Express-sovelluslogiikkaan, MongoDB:n datarakenteisiin sekä mongoose-kirjaston skeemasuunnitteluun ja validointimekanismeihin. Lisäksi opittiin käyttämään semanttista HTML:ää ja middlewareja tehokkaammin. Samalla projekti paljasti myös selkeitä kehityskohteita erityisesti sovelluksen käyttöliittymän osalta. Visuaalinen ilme jäi vielä luonnosmaiseksi ja mobiilikäytettävyyden parantaminen vaatii jatkotoimia ennen kuin sovelluksen viitsii esitellä portfoliossa.

Yhtyeen näkökulmasta sovellus vastasi asetettuihin tavoitteisiin teknisesti: käyttökelpoinen ja turvallinen verkkopalvelu onnistuttiin tuottamaan, vaikka sen visuaaliset ratkaisut kaipaavat vielä viimeistelyä. Sivuston ulkoasuun, värimaailmaan, fonttivalintoihin ja brändäykseen liittyvät päätökset on tarkoitus tehdä yhteistyössä yhtyeestä löytyvien graafikkojen kanssa.

Projektin edetessä ilmeni haasteita erityisesti tuotetietojen monimuotoisuuden kanssa. Skeemasuunnittelussa jouduttiin ottamaan huomioon esimerkiksi se, että osalla tuotteista oli kokoihin perustuva varasto, kun taas toiset tuotteet käyttivät suoraa kappalemäärää. Tämä edellytti robustia validointia ja tarkkaa lomakelogiikkaa. Lisäksi projektin laajuus osoittautui alun perin suunniteltua suuremmaksi, mikä johti joidenkin ominaisuuksien siirtämiseen jatkokehitykseen.

Sovelluksen tietoturvaratkaisut onnistuivat hyvin. Käyttöön otettiin muun muassa JWT-pohjainen kirjautuminen, tokenien voimassaolon hallinta, salasanan hajau-

tus, sekä piilotettu, UUID-pohjainen kirjautumissivu. Maksuprosessissa frontendin lähettämään tietoon ei luoteta sokeasti, vaan tilausdata validoidaan palvelimella ajantasaisen tietokantatiedon perusteella ennen maksusivulle ohjaamista.

Tulevaisuudessa sovellusta on tarkoitus kehittää edelleen. Parannuksia kaivataan muun muassa visuaalisen ilmeen hiomiseen, automaattiseen varastopäivitykseen tilausten yhteydessä sekä toimituskulujen erittelyyn. Lisäksi ylläpitotyötä helpottaisi, jos uusia tuotteita voitaisiin lisätä suoraan ylläpidon näkymästä käsin. Tietoturvaa voidaan parantaa edelleen lisäämällä kirjautumissivulle esimerkiksi brute force -hyökkäyksiä estävä rate limiting -mekanismi. Myös kuvien ulkoinen tallennusratkaisu, kuten Firebase, voisi helpottaa ylläpitoa ja parantaa palvelun skaalautuvuutta.

Opinnäytetyö tarjosi mahdollisuuden yhdistää opiskelijan henkilökohtainen kiinnostusalue ja konkreettinen kehitysprojekti. Se tarjosi arvokasta kokemusta ohjelmistokehityksen kokonaisvaltaisesta prosessista ja loi pohjan tuleville kehitysprojekteille. Kokonaisuutena työ täytti sille asetetut tavoitteet kohtuullisen hyvin ja osoitti, että tekninen sovellusratkaisu voidaan toteuttaa yksittäisen kehittäjän toimesta alusta loppuun asti, kunhan aikataulut ja laajuus suhteutetaan realistisesti käytettävissä oleviin resursseihin.

## LÄHTEET

About Stripe. n.d. Stripe. Verkkosivu. Viitattu 1.6.2025. https://stripe.com/en-fi/newsroom/information

Almog, S. 2024. The Art of Full Stack Debugging. DEV Community. Verkkosivu. Viitattu 30.4.2025. https://dev.to/codenameone/the-art-of-full-stack-debugging-3oba

How to accept international payments: What businesses need to know. 2023. Stripe. Verkkosivu. Viitattu 1.6.2025. https://stripe.com/en-fi/resources/more/how-to-accept-international-payments

Huomioita HTTP-pyyntötyyppien käytöstä. 2025. Helsingin yliopisto. Verkkosivu. Viitattu 11.5.2025. https://fullstackopen.com/osa3/node_js_ja_express#huomioita-http-pyyntotyyppien-kaytosta

Middleware in Express. 2025. GeeksforGeeks. Verkkosivu. Viitattu 19.5.2025. https://www.geeksforgeeks.org/middleware-in-express-js/

Mitä on Full stack -websovelluskehitys? 2025. Helsingin yliopisto. Verkkosivu. Viitattu 30.3.2025. https://www.helsinki.fi/fi/hakeminen-ja-opetus/avoin-yliopisto/monitieteiset-teemakokonaisuudet/full-stack

Murphy, R. 2025. What Is Stripe, and How Does It Work to Accept Payments? Nerdwallet. Verkkosivu. Viitattu 1.6.2025. https://www.nerdwallet.com/article/small-business/what-is-stripe

Pricing built for businesses of all sizes. n.d. Stripe. Verkkosivu. Viitattu 1.6.2025. https://stripe.com/en-fi/pricing

Stripe. n.d. Documentation. Verkkosivu. Viitattu 1.6.2025. https://docs.stripe.com/

Using middleware. n.d. Express. Verkkosivu. Viitattu 19.5.2025. https://expressjs.com/en/guide/using-middleware.html

What is Middleware? n.d. Amazon Web Services. Verkkosivu. Viitattu 19.5.2025. https://aws.amazon.com/what-is/middleware/

What is Stripe and is it trustworthy? n.d. Marknad. Verkkosivu. Viitattu 1.6.2025. https://marknad.fi/en/2020/06/what-is-stripe/

**LIITTEET**

Liite 1. Bändiuutisen eli postauksen Mongoose -skeema

backend > models > post.js > ...

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

const postSchema = new mongoose.Schema(
  {
    title: {
      type: String,
      required: [true, "Post title is required"],
      minlength: 3,
    },
    content: {
      type: String,
      required: [true, "Post content is required"],
      minlength: 3,
    },
  },
  { timestamps: true }
);

postSchema.set("toJSON", {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString();
    delete returnedObject._id;
    delete returnedObject.__v;
  },
});

module.exports = mongoose.model("Post", postSchema);
```

Liite 2. Tuotteen Mongoose -skeema

1(3)

backend > models > product.js > ...

```
1   const mongoose = require("mongoose");
2
3   const sizeSchema = new mongoose.Schema(
4     {
5       S: { type: Number, default: 0 },
6       M: { type: Number, default: 0 },
7       L: { type: Number, default: 0 },
8       XL: { type: Number, default: 0 },
9     },
10    { _id: false }
11  );
12
13  const productSchema = new mongoose.Schema(
14    {
15      name: {
16        type: String,
17        required: [true, "Product name is required"],
18        trim: true,
19      },
20      description: {
21        type: String,
22        required: [true, "Product description is required"],
23        trim: true,
24      },
25      category: {
26        type: String,
27        required: [true, "Product category is required"],
28        trim: true,
29        enum: ["apparel", "media", "accessory"],
30      },
```

(jatkuu)

```
31          imageUrl: {
32            type: String,
33            required: [true, "URL/path for product image is required"],
34            trim: true,
35          },
36          material: {
37            type: String,
38            trim: true,
39          },
40          sizes: sizeSchema,
41          quantity: Number,
42          price: {
43            type: Number,
44            required: [true, "Product price is required"],
45            min: 0,
46          },
47          available: {
48            type: Boolean,
49            default: true,
50          },
51        },
52        { timestamps: true }
53      );
```

(jatkuu)

3(3)

```
54
55  // custom validator to ensure sizes and quantity are mutually exclusive
56  // must use regular functions instead of arrow functions for
57  // Mongoose middleware hooks (pre, post, etc.) when you need access to this.
58  productSchema.pre("validate", function (next) {
59    if (this.sizes && this.quantity != null) {
60      return next(
61        new Error(
62          "Product can't have both 'sizes' and 'quantity'. Use one or the other."
63        )
64      );
65    }
66    next();
67  });
68
69  productSchema.set("toJSON", {
70    transform: (document, returnedObject) => {
71      returnedObject.id = returnedObject._id.toString();
72      delete returnedObject._id;
73      delete returnedObject.__v;
74    },
75  });
76
77  module.exports = mongoose.model("Product", productSchema);
78
```

Liite 3. Käyttäjän Mongoose -skeema

backend > models > user.js > ...

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

const userSchema = mongoose.Schema({
  username: {
    type: String,
    required: [true, "Username is required"],
    unique: true,
    minlength: 3,
  },
  passwordHash: {
    type: String,
    required: [true, "Password hash is required"],
  },
});

userSchema.set("toJSON", {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString();
    delete returnedObject._id;
    delete returnedObject.__v;
    delete returnedObject.passwordHash;
  },
});

module.exports = mongoose.model("User", userSchema);
```