Karelia-ammattikorkeakoulu
Tietojenkäsittely

Käyttäjän autentikoinnin toteutus MERN-sovellukseen

Joni Vepsäläinen

Opinnäytetyö, joulukuu 2023

www.karelia.fi

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Joulukuu 2023
Tietojenkäsittelyn koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600

Tekijä
Joni Vepsäläinen

Nimeke
Käyttäjän autentikoinnin toteutus MERN-sovellukseen

Tiivistelmä

Tämän opinnäytetyön tavoitteena on jatkokehittää selaimessa suoritettavaa MERN-sovellusta lisäämällä sovellukseen käyttäjän autentikointi ja samalla perehtyä toteutuksessa tarvittaviin teknologioihin. Sovellus on oma projekti, joka on kehitetty itsenäisesti keväällä 2023 osana suomalaisen IT-yrityksen rekrytointiprosessia.

Työssä perehdyttiin OAuth 2.0 -protokollan ja JWT-standardin toimintaan ja periaatteisiin sekä siihen kuinka JWT-standardia tukevia kirjastoja hyödynnetään turvallisesti pyyntöjen välittämisessä. Sovelluksen frontend on kehitetty Reactilla ja tilanhallinta on toteutettu Redux/Redux Toolkitilla. Sovelluksen backend on Node.js:llä ja Expressilla toteutettu REST:full API. Käyttäjän tiedot tallennetaan pilvessä toimivaan MongoDB Atlas -tietokantapalveluun.

JWT-standardia hyödyntämällä sovellukseen saatiin luotua toimiva käyttäjätilin luominen, käyttäjätilin aktivoiminen ja kirjautuminen. Toteutuksessa hyödynnettiin JWT:tä sovelluksen kaikilla tasoilla. Sovelluksen käyttöä on rajoitettu sallimalla tietyt reitit ja toiminnot käyttäjän roolin mukaisiksi.

Kieli
suomi

Sivuja 46
Liitteet 0
Liitesivumäärä 0

Asiasanat
React, Node.js, TypeScript, JWT

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
December 2023
Degree Programme in Information Technology

Tikkarinne 9
80200 JOENSUU
FINLAND
+ 358 13 260 600

Author
Joni Vepsäläinen

Title
User Authentication to MERN-Application

Abstract

The purpose of this thesis is to further develop a MERN stack application by implementing user authentication. A further aim is to gain familiarity with the technologies required for this implementation. The application is a personal project that has been developed independently in the spring of 2023 as part of the recruitment process of a Finnish IT company.

The focus was especially on the operation and principles of the OAuth 2.0 protocol and how the JWT token standard is used to securely transmit requests. The user interface of the application is made with React and the state management is handled with Redux/Redux Toolkit. The backend of the application is implemented with Node.js as a RESTful API. The user data is stored in a MongoDB Atlas cloud database.

By utilizing the JWT standard, functional user registration, account activation, and login features were created for the application. The JWT standard was utilized in both the frontend and backend of the application. Access to certain routes and functions in the application was restricted based on the user's role.

Language
Finnish

Pages 46
Appendices 0
Pages of Appendices 0

Keywords
React, Node.js, TypeScript, JWT

Sisältö

1 Johdanto

Käyttäjätietojen käsittely on web-ohjelmoinnissa yksi oleellisimpia taitoja. Sovelluksen suojaaminen ja toimintojen rajaaminen kirjautumistietojen perusteella kuuluu web-kehittäjän perustaitoihin. Tässä työssä tavoitteena on jatkokehittää MERN-teknologioilla kehitettyä sovellusta, liittämällä siihen toimiva esimerkkitoteutus käyttäjän autentikoinnista. Tuloksellisena tavoitteena on kehittää käyttäjän autentikoinnista toimiva esimerkki, jota voidaan hyödyntää mahdollisesti vastaavissa projekteissa. Oppimistavoitteena on syventyä lisää toteutuksessa käytettäviin ja yleisesti autentikaatioon liittyviin teknologioihin ja etenkin OAuth 2.0-protokollan toimintaan sekä perehtyä JWT-standardiin ja sitä tukevaan kirjastoon.

Olen opintojeni aikana suuntautunut erityisesti web-kehityksen ja niin sanotun fullstack-kehityksen suuntaan. Halusin valita aiheen, jossa pääsen työskentelemään sovelluksen kaikilla tasoilla. Tässä yhteydessä sovelluksen tasot tarkoittavat Reactilla tehtyä clienttia, Node.js-serveriä ja pilvessä sijaitsevaa MongoDB Atlas-tietokantapalvelua.

Alun luvuissa esitellään työn aiheeseen liittyvät teknologiat. Seuraavassa luvussa käydään läpi projektin lähtötilanne ja esitellään kehityksen kohteena oleva MERN-sovellus sekä kuvaillaan työn tavoitteet. Lähtötilanteen käsittelyn jälkeen käydään läpi kehitystyön toteutusvaihe. Raportin loppuosa käsittelee tuloksien esittelyn ja niiden pohdinnan sekä jatkokehitysideat projektille.

2 Teknologiat

2.1 MongoDB / MongoDB Atlas

MongoDB on dokumenttitietokanta ja se luokitellaan NoSQL-tyyppiseksi tietokannaksi, joka on vaihtoehto perinteiselle SQL-relaatiotietokannalle.

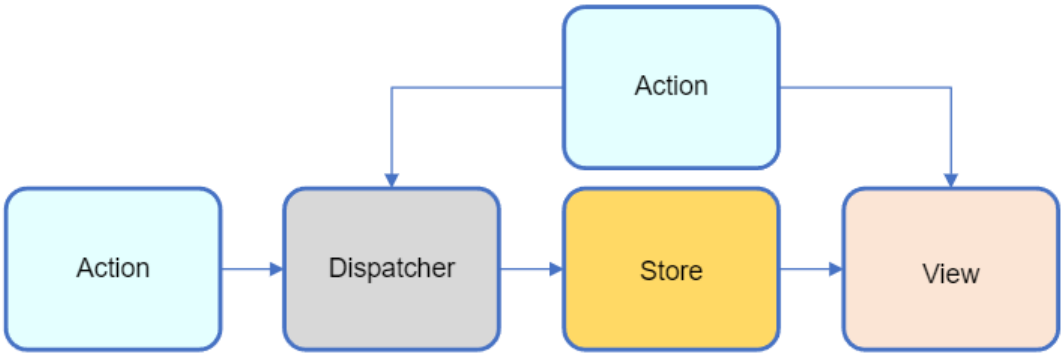
Taulujen ja rivien sijaan MongoDB:n arkkitehtuuri perustuu kokoelmiin ja dokumentteihin. Dokumenttien kentät vastaavat relaatiotietokannan taulujen tietueita. Dokumentit muistuttavat JSON-tyyppistä dataa, mutta ovat oikeasti ns. Binary JSON (BSON) -varianttia. Dokumenttien joukkoa kutsutaan kokoelmaksi ja se vastaa relaatiotietokannan taulua. MongoDB Atlas on pilvitietokantapalvelu samalta palveluntarjoajalta kuin MongoDB. (Gillis 2023.)

Kuten muut NoSQL-tietokannat, MongoDB ei vaadi ennalta määritettyjä skeemoja, joten se tallentaa kaikentyyppisiä tietoja. Tämä antaa käyttäjille joustavuuden luoda dokumenttiin, kuinka monta kenttää tahansa, mikä helpottaa MongoDB-tietokantojen skaalaamista relaatiotietokantoihin verrattuna. MongoDB:n ydintoiminto on sen horisontaalinen skaalautuvuus, mikä tekee siitä hyödyllisen tietokannan suuria datasovelluksia käyttäville yrityksille. (Gillis 2023.)

2.2 React, Redux ja TypeScript

React on Facebookin kehittämä kirjasto, joka on tarkoitettu käyttöliittymien kehittämiseen. Reactilla kehittäessä renderöitävä sisältö määritellään komponenttien avulla. Komponentit voidaan teknisesti mieltää JavaScript-funktioiksi. React-komponentin koodia tarkasteltaessa voi saada virheellisen vaikutelman, että komponentti palauttaisi HTML-koodia. Kyseessä on kuitenkin tapa, miten React-komponentit kirjoitetaan HTML:ää muistuttavalla JSX:llä, joka käännetään vielä JavaScriptiksi Babelin avulla. (Luukkainen 2023.)

Redux on React-sovelluksen tilanhallinnan kirjasto. Toimintaperiaatteeltaan Redux on samanlainen kuin Facebookin React-sovelluksen tilanhallintaan kehittämä Flux-arkkitehtuuri. Ideana on, että sovelluksen tilan hallinta erotetaan kokonaan React-komponenttien ulkopuolisiin storeihin. Storessa olevaa tilaa muutetaan tapahtumien eli actionien avulla, tämä voi olla esimerkiksi napin painallus käyttöliittymässä mikä aiheuttaa näkymän uudelleen renderöinnin (Kuva 1).



Kuva 1. Tilan muutos ja view:n päivitys actionilla. (mukaillen Meta Platforms 2023).

TypeScript on Microsoftin kehittämä ohjelmointikieli ja se on suunniteltu laaja-alaiseen JavaScript-kehitykseen. TypeScript tarjoaa ominaisuuksia, kuten paremman kehityksen aikaisen työkalun, staattisen koodianalyysin, käännösajan tyyppitarkistuksen ja kooditason dokumentaation. TypeScript on ns. JavaScriptin tyypitetty supersetti, ja se lopulta kääntyy tavalliseksi JavaScript-koodiksi. Tämä tarkoittaa käytännössä sitä, että TypeScript sisältää kaikki JavaScriptin ominaisuudet, joten kaikki JavaScript-koodi on kelvollista TypeScriptiä. TypeScriptin syntaksi on samankaltaista kuin JavaScriptin syntaksi mutta ei kuitenkaan täysin samaa. On huomionarvoista painottaa, että TypeScriptin tietotyyppien tarkastaminen toimii vain käännöksen aikana, joten ajonaikaista tietotyyppien tarkistusta ei ole olemassa ja ajonaikaisia virheitä voi olla, vaikka kääntäjä ei niistä ilmoittaisi. (Peuraniemi, Rapo & Torppa 2023.)

2.3 Node.js / Express

Node.js on ajonaikainen ympäristö JavaScript-koodin suorittamiseen verkkoselaimen ulkopuolella. Node.js:n avulla kehittäjät voivat siis käyttää JavaScriptiä sekä clientin että backendin-puolella, mikä tarjoaa yhtenäisen kielen molemmille. Tämä eliminoi kontekstin vaihtamisen tarpeen ja mahdollistaa täten koodin uudelleenkäytön clientin ja backendin välillä. Tämä parantaa tuottavuutta ja lyhentää kehitysaikaa. Node.js:ssä on laaja kattaus moduuleja ja kirjastoja, jotka ovat saatavilla Node Package Managerin (npm)

kautta. Kehittäjät voivat hyödyntää näitä valmiita moduuleja nopeuttaakseen kehitystä ja parantaakseen sovellusten toimivuutta. (Clinton 2023.)

Express on Node.js:n suosituin framework. Se on suunniteltu selainsovelluksien ja API-rajapintojen kehittämiseen. Node.js-sovelluksen backendin kehittäminen alusta alkaen voi olla työlästä ja aikaa vievää. Käyttämällä kehityksessä frameworkkia, kuten Expressiä, kehittäjät voivat näin säästää aikaa ja keskittyä muihin tärkeisiin tehtäviin. Express tarjoaa valmiiksi menetelmiä määrittää, mitä funktiota kutsutaan tietylle HTTP-verbille (GET, POST, PUT jne.) ja URL-mallille (Route). (OpenJS Foundation 2017.)

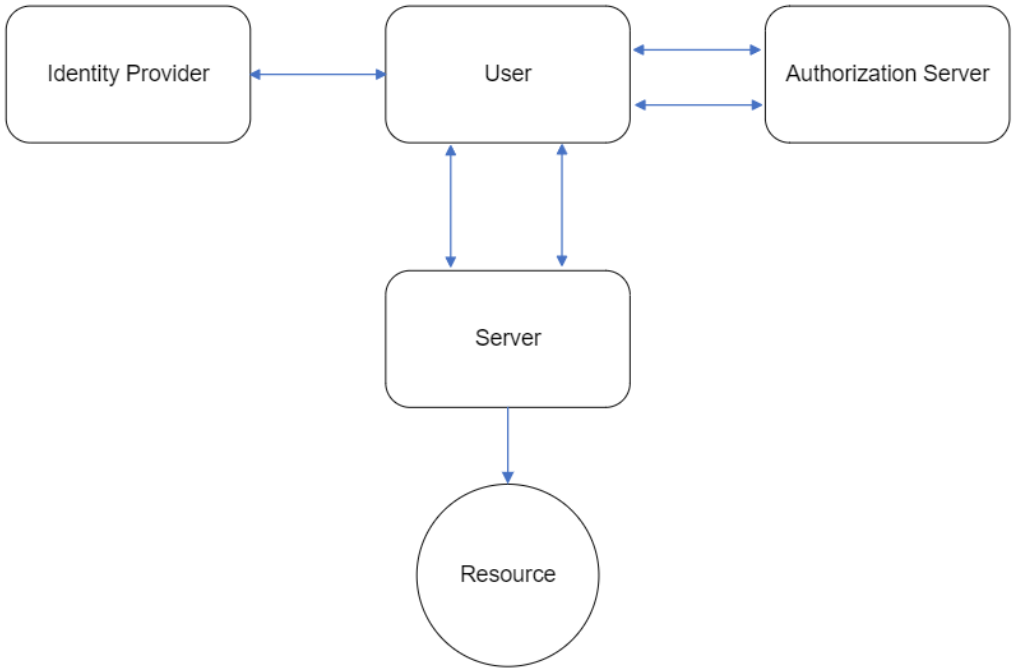
2.4 OAuth 2.0

OAuth 2.0 on autorisointiprotokolla, jota voidaan hyödyntää tietoturvallisen käyttäjä autentikoinnin toteutuksessa. OAuth 2.0 -protokollan avulla voidaan hyödyntää eri palveluihin luotuja käyttäjätunnuksia käyttäjän autentikointiin. Yksiä käyttäjätunnuksia voidaan käyttää OAuth 2.0 -protokollaa hyödyntämällä eri sovelluksiin kirjautumiseen ja myös sovellusten välisten tietojen vaihtamiseen ja resurssien hyödyntämiseen. (Bihis 2015, 1–3.)

2.5 OAuth 2.0 protokollan keskeisiä termejä

Autentikoinnilla (Authentication) tarkoitetaan henkilön tai järjestelmän identiteetin validoimista, toisin sanoen varmistetaan, että henkilö/järjestelmä on todella se mikä väittää olevansa. Arkisessa tilanteessa tämä voisi olla verrattavissa esimerkiksi pankissa asioinnin yhteydessä vaadittuun passilla tai henkilökortilla tunnistautumiseen. Tunnistautumisella (Authorization) taas varmistetaan, että kyseessä olevalla henkilöllä tai järjestelmällä on valtuudet toteuttaa haluttu toimi. Esimerkiksi sovelluksen käyttäjällä voi olla valtuudet tarkastella sovelluksen tietoja, mutta tietojen muokkaamiseen tarvitaan erilliset valtuudet mitkä voivat olla esimerkiksi myönnettynä sovelluksen ylläpitäjälle. (Bihis 2015, 2.)

Federated identity (kuva 2) tarkoittaa mahdollisuutta hyödyntää jonkin palvelun, esimerkiksi Facebookiin luotuja tunnuksia, johonkin toiseen palveluun kirjautumisessa. Erillisiä tunnuksia ei ole siis pakko luoda jokaiseen sovellukseen erikseen.

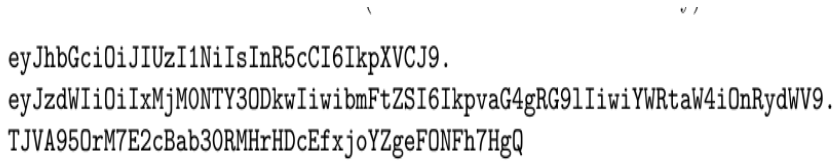


Kuva 2. Common Federated Identity Flow. (mukaillen Peyrott 2018).

Delegated identity eli sovellusten välisten tietojen jakaminen, esimerkiksi Google-tilin yhteystietojen perusteella uusien kontaktien ehdottaminen LinkedIn-sovelluksessa. (Bihis 2015, 3.)

2.6 JSON Web Token (JWT)

JWT eli JSON Web Token on standardi pyyntöjen turvalliseen välittämiseen rajoitetuissa ympäristöissä. Se on tuettu laajasti monissa eri frameworkeissa. Yksinkertaisuus, kompaktius ja käytettävyys ovat sen arkkitehtuurin keskeisiä piirteitä. JSON Web Token näyttää vain sekavalta kirjaimien ja numeroiden sarjalta, mutta on itse asiassa erittäin kompakti esitys sarjasta pyyntöjä sekä allekirjoituksesta sen aitouden varmistamiseksi (kuvat 3 & 4).



Kuva 3. JSON Web Token. (Peyrott 2018, 5).

Kuva 4. JSON Web Token sisältö. (Peyrott 2018, 5–6).

Pyynnöt ovat JWT:n tapauksessa osapuolta tai objektia koskevia määritelmiä tai väitteitä. Jotkut näistä väitteistä ja niiden merkitys on määritelty osana JWT:n omia määritelmiä. Muut ovat käyttäjän määrittelemiä. JWT on standardoinut yleisimpiä toimintoja koskevat vaatimukset, mikä tekee siitä erityisen käytettävän. Esimerkiksi yksi näistä yleisistä toiminnoista on tietyn osapuolen henkilöllisyyden määrittäminen. (Peyrott 2018, 6.)

Toinen JWT:n tärkeä ominaisuus on allekirjoituksen mahdollisuus käyttämällä JSON Web Signatures -teknologiaa (JWS, RFC 75156) ja/tai sen salaaminen JSON Web Encryption -salauksella (JWE, RFC 75167). Yhdessä JWS:n ja JWE:n kanssa JWT:t tarjoavat tehokkaan ja turvallisen ratkaisun moniin erilaisiin ongelmiin (kuva 5).

HMAC signatures require a shared secret. Any string will do:

```
const secret = 'my-secret';

const signed = jwt.sign(payload, secret, {
    algorithm: 'HS256',
    expiresIn: '5s' // if ommited, the token will not expire
});
```

Verifying the token is just as easy:

```
const decoded = jwt.verify(signed, secret, {
    // Never forget to make this explicit to prevent
    // signature stripping attacks
    algorithms: ['HS256'],
});
```

Kuva 5. JWT sign ja verify metodit. (Peyrott 2018, 39).

Vaikka JWT:n päätarkoitus on pyyntöjen välittäminen kahden osapuolen välillä, luultavasti sen tärkein ominaisuus on standardointityö yksinkertaisen, valinnaisesti validoidun ja/tai salatun säilöntätavan muodossa (Peyrott 2018, 6).

3 OAuth 2.0:n käyttötapauksen kuvaus

3.1 Sovellusten välinen resurssien vaihto

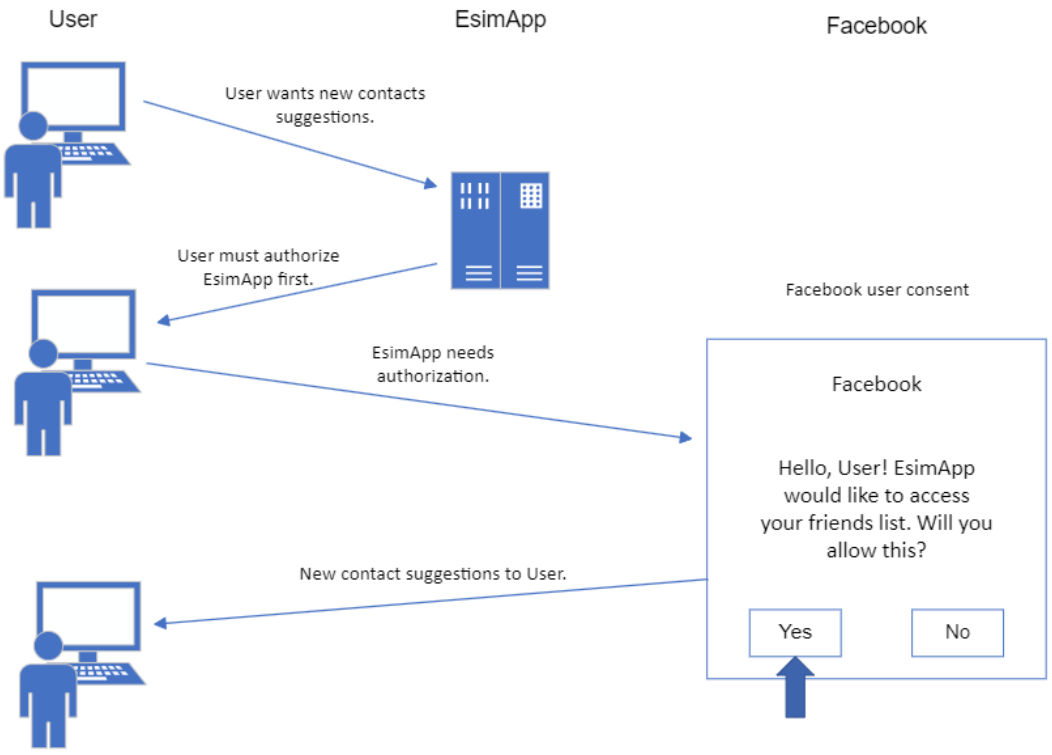
Jos EsimApp-sovellus haluaa päästä käsiksi käyttäjän Facebook-kaverilistaan, ilman OAuth 2.0-protokollan hyödyntämistä, käyttäjä joutuu antamaan Facebook-tunnukset EsimApille. Tunnuksien avulla EsimApp saa noudettua kaverilistan tiedot. Sovellus esiintyy käyttäjänä ja suorittaa Facebookiin kirjautumisen. Käyttäjän kaverilistan tiedot saadaan näin noudettua onnistuneesti. Tämä toteutus tuo mukanaan kuitenkin monia ikäviä ongelmia, sillä käyttäjä antaa oikeat Facebook-tunnukset EsimApille ja samalla käyttäjä on antanut luvan EsimApille päästä käsiksi kaikkeen Facebook tilin sisältöön. Tämä luo mahdollisuuden Facebook tunnuksien verkkoon vuotamiselle. (Bihis 2015, 5–6.)

Jos EsimApp-sovellus hyödyntää OAuth 2.0 -protokollaa, sovellus ensin valtuutetaan noutamaan käyttäjän kaverilista. Minkä jälkeen käyttäjä lähetetään kirjautumaan Facebookin ja samalla käyttäjältä pyydetään Facebookin toimesta

valtuutus EsimApp-sovellukselle päästä käsiksi käyttäjän kaverilistaan. Käyttäjän antaman valtuutuksen jälkeen kaverilistan tiedot saadaan noudettua sovellukseen. (Bihis 2015, 7–8.)

3.2 Korkean tason katsaus OAuth 2.0:n toimintaan

Edellisessä käyttötapauksen kuvauksessa oli esiteltynä yksinkertainen esimerkki siitä, kuinka saadaan noudettua EsimApp-sovellukseen käyttäjän Facebook kaverilista, ilman OAuth 2.0 -protokollaa sekä hyödyntämällä OAuth 2.0 -protokollaa. Tarkastellaan seuraavaksi käyttötapausesimerkkiä, jossa hyödynnetään OAuth 2.0 -protokollaa. Prosessin vaihe, jossa Facebook kysyy käyttäjältä, annetaanko EsimApp-sovellukselle valtuudet tehdä sen haluama toimi, kutsutaan ”käyttäjän suostumukseksi” (user consent) (kuva 6).



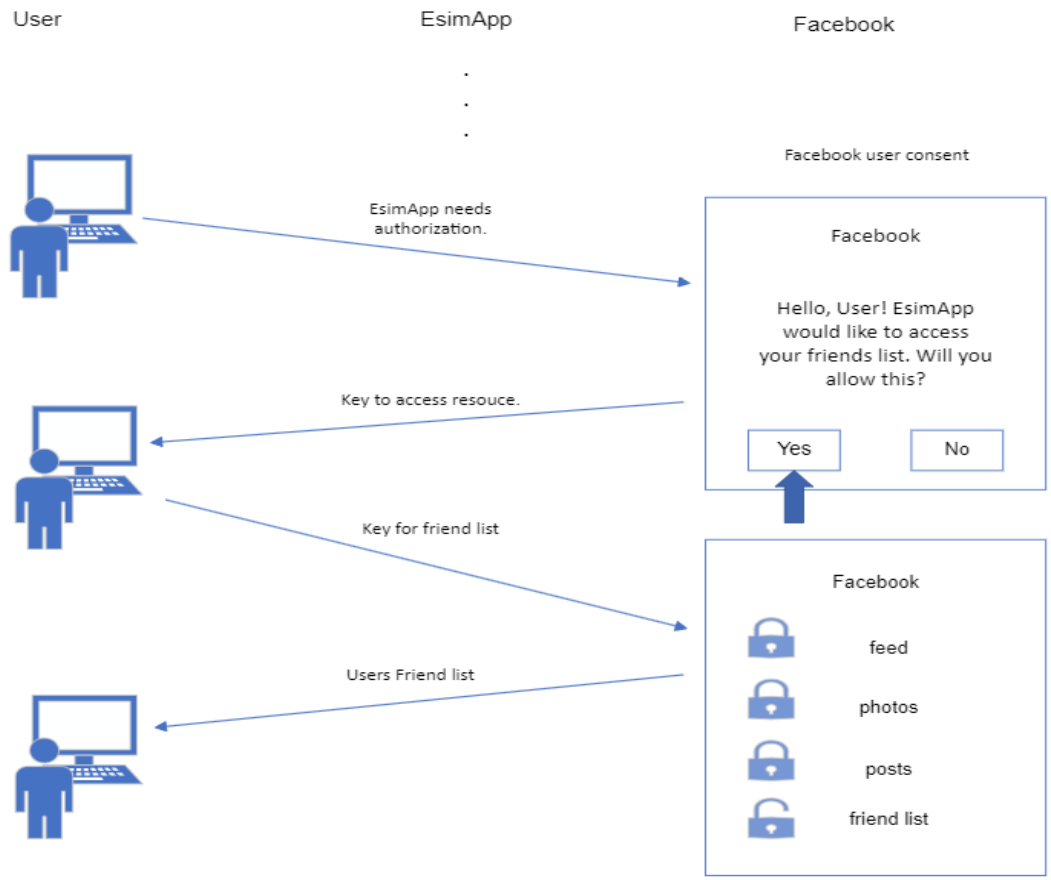
Kuva 6. Käyttäjän valtuutus Facebook kaverilistan käyttöön. (mukaillen Bihis 2015).

Käyttäjän suostumus tarvitaan, kun sovellus haluaa suorittaa toiminnon, joka liittyy käyttäjän omistamaan resurssiin. Esimerkkitapauksessamme EsimApp-sovellus haluaa pääsyn käyttäjän kaverilistaan ja palvelun tarjoajana on Facebook. Tämä toiminto onnistuu, kun Facebook saa suostumuksen käyttäjältä itseltään. Kun käyttäjän suostumus on saatu suoraan käyttäjältä itseltään, EsimApp-sovellus on saanut tarkasteluun oikeuttavat valtuudet käyttäjän Facebook-kaverilistaan. (Bihis 2015, 13–16.)

EsimApp-sovelluksen ja Facebookin välinen tiedonvaihtoprosessi vaikuttaa näin korkealta tasolta tarkasteltuna varsin yksinkertaiselta, mutta pinnan alla tapahtuva prosessi on huomattavasti monimutkaisempi ja on riippuvainen eri tekijöistä. Tiedonvaihtoprosessin kulku riippuu siitä, minkä tyyppinen sovellus on kyseessä ja minkälaiset ovat sen ominaisuudet. OAuth 2.0 -protokolla tukee kahta eri tapaa tiedon vaihtamiseen clientin ja palvelun tarjoajan välillä. OAuth 2.0:n terminologian mukaisesti näitä tapoja kutsutaan nimellä grant types. Authorization code grant voidaan mieltää serverillä tapahtuvaan tiedon prosessointiin ja Implicit grant taas clientin päässä tapahtuvaan tiedon prosessointiin. (Bihis 2015, 17.)

Tiedonvaihtoprosessiin vaikuttaa myös, onko kyseessä niin sanotusti luotettu (trusted) vai ei luotettu (untrusted) sovellus. Sovellus on luotettu, mikäli sen ominaisuuksiin kuuluu tietoturvallinen tapa säilöä ja välittää tietoa. Tämä voi olla esimerkiksi sovellus missä on erillinen backend-serveri hoitamassa tietoturvallista tiedonkäsittelyä. Ei luotettava sovellus määritelmän mukaisesti ei kykene tietoturvalliseen tiedonkäsittelyyn. Tämänkaltainen sovellus on esimerkiksi selaimessa suoritettava HTML/JavaScript sovellus, joka ei hyödynnä erillistä serveriä tietoturvalliseen tiedonkäsittelyyn. (Bihis 2015, 17.)

Mikäli EsimApp on ei luotettu sovellus, tiedonvaihtoprosessin toteutus on kohtuullisen yksinkertainen (kuva 7).

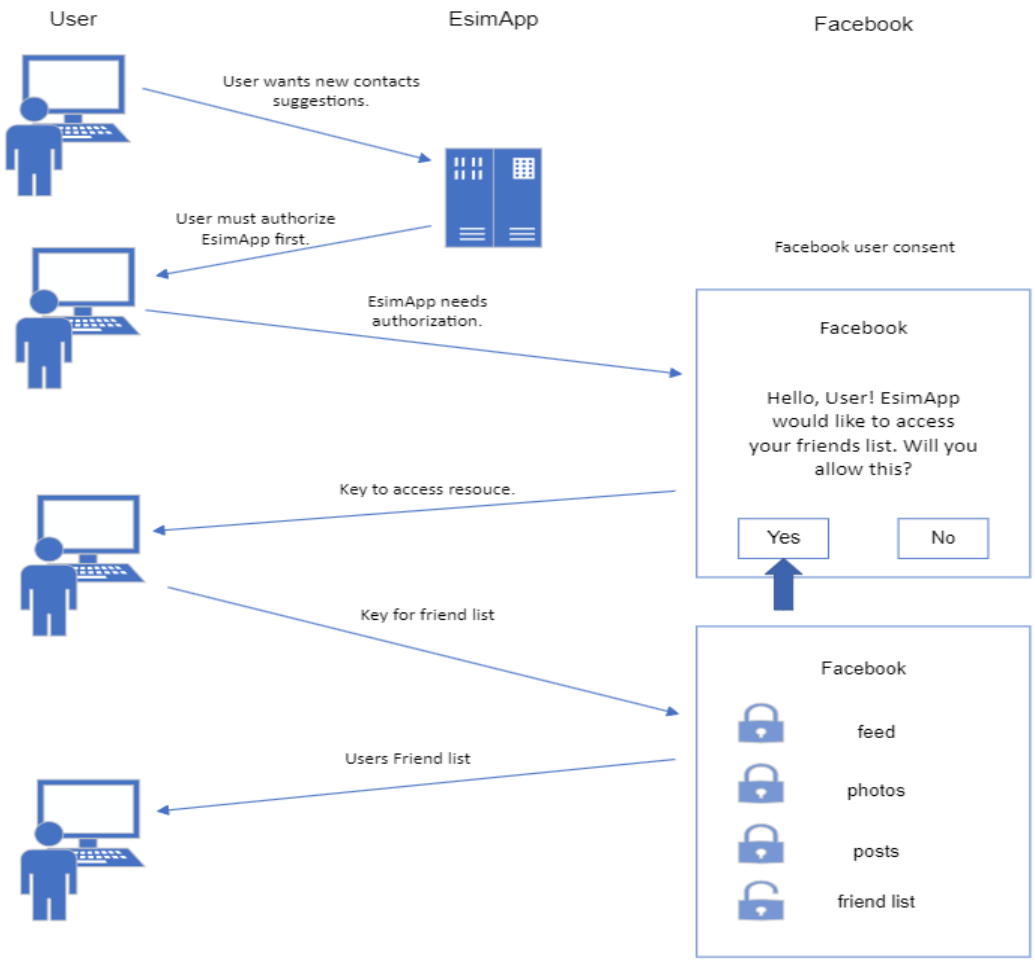


Kuva 7. Ei luotetun clientin prosessi kuvattuna. (mukaillen Bihis 2015).

Kun käyttäjä on antanut valtuutuksen EsimApp-sovellukselle haluttuun resurssiin, seuraavaksi palvelun tarjoaja, tässä tapauksessa Facebook, antaa avaimen mitä kutsutaan OAuth 2.0 protokollan terminologian mukaisesti Access Tokeniksi. Avaimen avulla EsimApp-sovellus voi lähettää pyynnön Facebookille koskien haluttua resurssia. Facebook validoi avaimen ja toteuttaa pyynnön onnistuneen validoinnin jälkeen. Kyseisen avaimen ominaisuuksiin kuulu se, että ne toimivat vain tiettyyn resurssiin, joten jokaista resurssia tai toimea varten on pyydettävä oma avain palvelun tarjoajalta (Implicit grant). Ei luotettavan sovelluksen kanssa toteutetun tiedonvaihtoprosessin heikkous on tietoturva. Koska sovellus ei voi säilöä tietoa turvallisesti joudutaan pyyntöjä ja validointeja suorittamaan useammin. (Bihis 2015, 18–23.)

Jos EsimApp-sovellus on luotettu sovellus ja se sisältää sekä palvelin- että tietokantakerrokset, joiden avulla se pystyy tietoturvallisesti säilömään dataa, on

mahdollista hyödyntää OAuth 2.0 -protokollan Authorization Grant tiedonvaihtoprosessia (kuva 9).



Kuva 8. Luotetun clientin prosessi kuvattuna. (mukaillen Bihis 2015).

Tiedonvaihto tapahtuu serverin puolella tag-tunnisteen avulla mikä pyydetään Facebookilta. Tag-tunnisteen avulla käyttäjä voi pyytää haluttuun resurssiin liittyvää avainta. Saadulla avaimella käyttäjä pyytää resurssia ja Facebook toteuttaa pyynnön avaimen validoinnin jälkeen. Luotettu sovellus voi olla esimerkiksi client/server sovellus missä on HTML/JavaScript client ja esimerkiksi .NET-backend, joka on yhteydessä SQL Server -tietokantaan. Tämänkaltainen tiedonvaihtoprosessi on tietoturvallisempi, mutta se lisää myös samalla toteutuksen kompleksisuutta. (Bihis 2015, 23–28.)

4 Toteutus

4.1 Kuvaus sovelluksesta

Kyseessä on selaimessa suoritettava MERN-sovellus, joka tarjoaa käyttäjille monipuolisia toimintoja liittyen pyörämatkoihin ja asemiin. Sovellus on toteutettu React TypeScriptillä ja käyttää Redux/Redux Toolkittia tilanhallintaan, mikä tekee siitä erittäin responsiivisen ja käyttäjäystävällisen. Frontendissä on useita räätälöitäviä, valmiita komponentteja Material UI:sta, mikä helpottaa modulaarisen sovelluksen kehittämistä.
Backendissä on käytössä Node.js ja Express, jotka tarjoavat tehokkaan ja helppokäyttöisen frameworkin kehittäjille. Sovellus käyttää MongoDB Atlasin tarjoamaa pilvipohjaista tietokantaa datan tallentamiseen ja hakemiseen

4.2 Backend

Toteutus alkoi käyttäjän tietotyyppien lisäämisellä (kuva 9) ja käyttäjädokumentin ominaisuuksien ja metodien määrittelyllä (kuva 10).

```
28
29  export type UserDocument = Document & {
30    name: string;
31    email: string;
32    hashed_password: string;
33    salt: string;
34    role: string;
35    resetPasswordLink: { data: string };
36    // Define methods here
37    authenticate(plainText: string): boolean;
38    encryptPassword(password: string): string;
39    makeSalt(): string;
40  };
41
```

Kuva 9. Käyttäjädokumentin tietotyyppien määrittäminen.

```
import mongoose from "mongoose";
import crypto from "crypto";
import { UserDocument } from "../types";

const userSchema = new mongoose.Schema(
  {
    name: {
      type: String,
      trim: true,
      required: true,
      maxlength: 32,
    },
    email: {
      type: String,
      trim: true,
      required: true,
      unique: true,
      lowercase: true,
    },
    hashed_password: {
      type: String,
      required: true,
    },
    salt: String,
    role: {
      type: String,
      default: "user",
    },
    resetPasswordLink: {
      data: String,
    },
  },
  { timestamps: true }
);
```

Kuva 10. Määritellään käyttäjändokumentin (skeeman) kentät.

Tietotyyppien ja käyttäjädokumentin määrittelyn jälkeen lisättiin käyttäjän autentikointiin liittyvät metodit tilin luomiseen, aktivoimiseen ja sovellukseen kirjautumiseen. Tämän jälkeen määriteltiin middlewaret, joiden avulla validoidaan syötetyt kirjautumistiedot, tarkistetaan kirjautuminen ja käyttäjän rooli. Lopuksi lisättiin reitit sekä reitteihin niille ominaiset middlewaret.

4.2.1 Salasanan käsittely käyttäjä mallissa

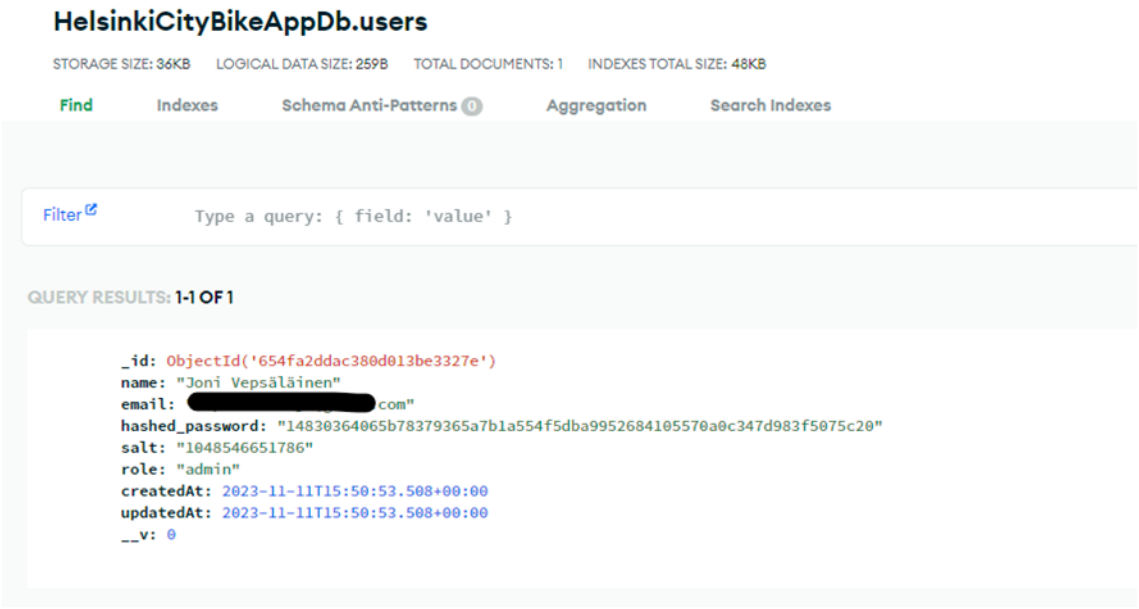
Käyttäjän tietojen määrittely vaati metodien luomista salasanan tietoturvallista käsittelyä varten. Mongoose, MongoDB-objektimallinnustyökalu, määrittelee virtualin ja monia instanssimetodeja, joita käytetään käyttäjädokumentissa

(userSchema). Metodi ”userSchema.virtual(“password”)” luo virtuaalisen ominaisuuden käyttäjädokumentin passwordille. Virtualit ovat dokumentin ominaisuuksia, joita voi hakea ja asettaa, mutta ne eivät tallennu MongoDB:hen. Funktio ”.set(function (this: UserDocument, password: string) {…})” on asetusfunktio, jota kutsutaan, kun asetetaan arvo password-virtuaaliominaisuudelle. Se luo ”suolauksen” käyttämällä makeSalt-metodia, sitten se salaa salasanan käyttämällä encryptPassword-metodia ja tallentaa salatun salasanan käyttäjädokumentin kenttään hashed_password Funktio ”.get(function (this: UserDocument) {…})” on haku-funktio, jota kutsutaan, kun haetaan password-virtuaaliominaisuutta. Se palauttaa yksinkertaisesti salatun salasanan. UserDocument-tyyppiä käytetään tämän funktioissa, joka viittaa mallin instanssiin (dokumenttiin). Metodia ”authenticate” käytetään tarkistamaan selkokielinen salasana käyttäjän salattua salasanaa vastaan. Se palauttaa vertailun tuloksena totuusarvon true tai false. Metodia ”encryptPassword” käytetään salasanan salaamiseen. Se käyttää crypto-moduulin createHmac-metodia luomaan HMAC (Hash-based Message Authentication Code) ja salaamaan salasanan. Sha256-algoritmia käytetään salaamiseen ja käyttäjän suolaa käytetään avaimena. Metodia makeSalt käytetään ”suolan” luomiseen, toisin sanoen luodaan satunnainen merkkijono perustuen nykyiseen aikaleimaan ja satunnaislukuun (kuva 11).

```
// virtual
userSchema
  .virtual("password")
  .set(function (this: UserDocument, password: string) {
    this.salt = this.makeSalt();
    this.hashed_password = this.encryptPassword(password);
  })
  .get(function (this: UserDocument) {
    return this.hashed_password;
  });

// methods
userSchema.methods = {
  authenticate: function (this: UserDocument, plainText: string) {
    return this.encryptPassword(plainText) === this.hashed_password;
  },

  encryptPassword: function (this: UserDocument, password: string) {
    if (!password) {
      throw new Error("Password cannot be empty");
    }
    try {
      const algorithm = "sha256";
      return crypto
        .createHmac(algorithm, this.salt)
        .update(password)
        .digest("hex");
    } catch (err) {
      return "";
    }
  },

  makeSalt: function (this: UserDocument) {
    return Math.round(new Date().valueOf() * Math.random()) + "";
  },
};

export default mongoose.model<UserDocument>("User", userSchema);
```

Kuva 11. Määritellään userScheman metodit ja, että userSchema on määritellyn UserDocument tietotyypin mukainen.

Sen sijaan, että salasanat tallennettaisiin suoraan, tallennetaan salattu versio ja sitä verrataan syötettyyn salasanaan, kun käyttäjä yrittää kirjautua sisään. Tällä tavalla, vaikka joku pääsisi käsiksi tietokantaan, he eivät pysty näkemään käyttäjien todellisia salasanoja (kuva 12).



Kuva 12. Käyttäjä dokumentti MongoDB Atlas-tietokantapalvelussa.

4.2.2 Autentikointi metodit

Metodi signup on asynkroninen funktio, joka ottaa kolme argumenttia: req (pyyntöobjekti), res (vastausobjekti) ja next (takaisinkutsufunktio, joka siirtää ohjauksen eteenpäin. Se ensin purkaa nimen, sähköpostin ja salasanan pyynnön rungosta (req.body). Seuraavaksi tarkistetaan löytyykö tietokannasta käyttäjä, jolla on pyynnöstä saatu sähköposti käyttämällä mongoosen User.findOne({ email }) metodia. Jos käyttäjä löytyy, lähetetään vastaus, jonka tila on 400 ja viesti “Email is taken”. Jos sähköposti ei ole varattu, luodaan JSON Web Token (JWT) mikä sisältää käyttäjän nimen, sähköpostin ja salasanan. Tämä token on allekirjoitettu account_activation_key:llä ja sillä on vanhentumisaika 10 minuuttia. Seuraavaksi määritellään sähköpostin sisältö, joka lähetetään käyttäjälle tilin aktivoimiseksi. Sähköposti sisältää linkin, jonka avulla käyttäjä voi aktivoida tilinsä. Linkki sisältää JWT-tokenin. Sähköposti lähetetään käyttämällä sgMail.send(emailData). Jos sähköposti lähetetään onnistuneesti, se kirjaa “SIGNUP EMAIL SENT” konsoliin ja lähettää vastauksen, jonka tila on 200 ja viestin, joka ohjeistaa käyttäjää noudattamaan sähköpostissa olevia ohjeita tilin aktivoimiseksi. Sähköposti lähetetään

@sendgrid/mail kirjastoa käyttäen, minkä käyttö vaatii ilmaisen tilin luomista SendGrid palveluun (kuva 13).

```
export const signup = async (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  try {
    const { name, email, password } = req.body as {
      name: string;
      email: string;
      password: string;
    };

    User.findOne({ email }).then((user) => {
      if (user) {
        return res.status(400).json({ error: "Email is taken" });
      }

      const token = jwt.sign(
        { name, email, password },
        account_activation_key as string,
        { expiresIn: "10m" }
      );

      //This message will be sent to email address provided by the user
      const emailData = {
        from: sender_email as string,
        to: email,
        subject: `Account activation link`,
        html: `
         <h1>Please use the following link to activate your account</h1>
         <p>${url}/auth/activate/${token}</p>
         <hr />
         <p>This email may contain sensitive information</p>
         <p>${url}</p>
    `,
      };

      sgMail.send(emailData).then((sent) => {
        console.log("SIGNUP EMAIL SENT", sent);
        return res.status(200).json({
          message: `Email has been sent to ${email}. Follow the instructions to activate your account.`,
        });
      });
    });
  } catch (error) {
    if (error instanceof Error) {
      console.log("Invalid Request", 400, error);
      return res.json({
        error: error.message,
      });
    }
    next(error);
  }
};
```

Kuva 13. Rekisteröitymisen (signup) suorittava metodi.

Metodi accountActivation on asynkroninen funktio, joka aktivoi käyttäjän tilin verkkosovelluksessa. Funktio, joka ottaa kaksi argumenttia: req(pyyntöobjekti) ja res (vastausobjekti). Se ensin purkaa tokenin arvon pyynnön rungosta

(req.body). Jos token on olemassa, se yrittää purkaa sen käyttämällä jwt.verify(token, account_activation_key as string) jsonwebtoken kirjaston metodia hyödyntäen. Tämä palauttaa objektin, joka sisältää nimen, sähköpostin ja salasanan. Uusi käyttäjä luodaan User-mallilla, joka luotiin aikaisemmin. Käyttäjän tiedot (nimi, sähköposti ja salasana) saadaan puretusta tokenista. Uuden käyttäjän tiedot tallennetaan tietokantaan käyttämällä mongoosen tarjoamaa save(). metodia. Jos kaikki menee hyvin, lähetetään vastaus, jossa on viesti “Signup success. Please sign in.” Jos prosessissa ilmenee virhe, otetaan se kiinni ja kirjataan konsoliin ja lähetetään vastaus, jonka tila on 401 ja virheviesti “Expired link. Signup again.” Jos tokenia ei ole, lähetetään vastaus, jossa on viesti “Something went wrong. Please try again.” (kuva 14).

```
export const accountActivation = async (req: Request, res: Response) => {
  const { token } = req.body;

  if (token) {
    try {
      const decoded = jwt.verify(token, account_activation_key as string) as {
        name: string;
        email: string;
        password: string;
      };

      const { name, email, password } = decoded;

      const user = new User({ name, email, password });

      await user.save();
      return res.json({
        message: "Signup success. Please sign in.",
      });
    } catch (error: any) {
      console.log("JWT VERIFY AND ACCOUNT ACTIVATION ERROR", error);
      return res.status(401).json({
        error: `Expired link. Signup again. ${error.message}`,
      });
    }
  } else {
    return res.json({
      message: "Something went wrong. Please try again.",
    });
  }
};
```

Kuva 14. Käyttäjätilin aktivoimisen suorittava eli käyttäjän tiedot tietokantaan tallentava metodi.

Metodi signin on asynkroninen funktio, joka kirjaa käyttäjän sisään verkkosovellukseen. Funktio ottaa kaksi argumenttia: req (pyyntöobjekti) ja res (vastausobjekti). Ensiksi puretaan email- ja password-arvot pyynnön rungosta (req.body). Seuraavaksi yritetään löytää käyttäjä tietokannasta sähköpostiosoitteen perusteella käyttämällä mongoosen tarjoamaa findOne() metodia. Jos käyttäjää ei löydy, lähetetään vastaus, jonka tila on 400 ja virheviesti “User with that email does not exist. Please signup.” Jos käyttäjä löytyy, mutta salasana ei täsmää, lähetetään vastaus, jonka tila on 401 ja virheviesti “Email and password do not match.” Jos käyttäjä on todennettu, luodaan JSON Web Token (JWT) käyttäjän _id:n perusteella. Tämä token on allekirjoitettu jwt_secret:llä ja sillä on vanhentumisaika 7 päivää. Seuraavaksi luodaan foundUser-objekti, joka sisältää käyttäjän _id:n, nimen, sähköpostin ja roolin. Lopuksi lähetetään vastaus, joka sisältää tokenin ja foundUser-objektin. Jos prosessissa ilmenee virhe, otetaan virhe kiinni ja kirjataan se konsoliin ja lähetetään vastaus, jonka tila on 500 ja virheviesti “Internal server error.” (kuva 15).

```
export const signin = async (req: Request, res: Response) => {
  try {
    const { email, password } = req.body;

    const user = await User.findOne({ email });

    if (!user) {
      return res.status(400).json({
        error: "User with that email does not exist. Please signup.",
      });
    }

    if (!user.authenticate(password)) {
      return res.status(401).json({
        error: "Email and password do not match.",
      });
    }

    // User is authenticated, handle the success case here
    const token = jwt.sign({ _id: user._id }, jwt_secret as string, {
      expiresIn: "7d",
    });
    const foundUser = {
      _id: user._id,
      name: user.name,
      email: user.email,
      role: user.role,
    };

    return res.json({
      token,
      user: foundUser,
    });
  } catch (error) {
    console.error("Error during signin:", error);
    return res.status(500).json({
      error: "Internal server error.",
    });
  }
};
```

Kuva 15. Kirjautumisen suorittava metodi

4.2.3 Middlewaret ja reitit

Metodi requireSignin on middleware-funktio, joka vaatii käyttäjän kirjautumisen verkkosovellukseen. Funktio käyttää expressjwt-moduulia. Tämä moduuli tarjoaa middleware-toiminnallisuuden Express-sovelluksille JWT-tunnistautumista varten. Funktio ottaa argumenttina objektin, joka sisältää seuraavat avaimet:

- secret: Tämä on salainen avain, jota käytetään JWT:n allekirjoittamiseen ja tarkistamiseen. Tässä tapauksessa se on jwt_secret, joka on määritelty ympäristömuuttujissa.
- algorithms: on taulukko, joka määrittelee, mitä algoritmeja voidaan käyttää JWT:n allekirjoittamiseen ja tarkistamiseen. Tässä tapauksessa se sisältää vain yhden algoritmin, “HS256”, joka on yksi yleisimmin käytetyistä algoritmeista JWT:n kanssa (Peyrott 2018, 61).

Tämä funktio palauttaa middleware-toiminnon, jota voidaan käyttää reitittimessä varmistamaan, että käyttäjä on kirjautunut sisään ennen kuin he pääsevät tiettyihin reitteihin. Jos käyttäjä ei ole kirjautunut sisään, expressjwt lähettää virheen, joka voidaan käsitellä muualla sovelluksessa. Metodi adminMiddleware on asynkroninen funktio, joka tarkistaa, onko käyttäjän rooli admin (ylläpitäjä) sovelluksessa. Funktio ottaa kolme argumenttia: req (pyyntöobjekti), res (vastausobjekti) ja next (takaisinkutsufunktio, joka siirtää ohjauksen seuraavalle tasolle). Ensin puretaan _id-arvo pyynnön auth-objektista, tämä onnistuu express-jwt-kirjaston avulla ja määrittämällä reg tyypiksi sen tarjoama JWTrequest. Seuraavaksi yritetään löytää käyttäjä tietokannasta _id:n perusteella käyttämällä mongoosen findById() metodin avulla. Jos käyttäjää ei löydy, lähetetään vastaus, jonka tila on 400 ja virheviesti “User not found”. Jos käyttäjä löytyy, mutta käyttäjän rooli ei ole “admin”, lähetetään vastaus, jonka tila on 400 ja virheviesti “Admin resource. Access denied.” Jos käyttäjän rooli on admin, kutsutaan next-funktiota, ja siirretään ohjaus seuraavalle tasolle. Virheen

ilmaantuessa, se otetaan kiinni ja kirjataan se konsoliin ja lähetetään sitten vastaus, jonka tila on 500 ja virheviesti “Internal server error.” (kuva 16).

```
// auth middlewares
export const requireSignin = expressjwt({
  secret: jwt_secret as string,
  algorithms: ["HS256"],
});

export const adminMiddleware = async (
  req: JWTRequest,
  res: Response,
  next: NextFunction
) => {
  try {
    const id = req.auth?._id;

    const user = await User.findById(id);

    if (!user) {
      return res.status(400).json({
        error: "User not found",
      });
    }

    if (user.role !== "admin") {
      return res.status(400).json({
        error: "Admin resource. Access denied.",
      });
    }

    next();
  } catch (error) {
    console.error("Error during admin authorization:", error);
    return res.status(500).json({
      error: "Internal server error.",
    });
  }
};
```

Kuva 16. Kirjautumisen tiedon ja admin roolin tarkistavat middlewaret.

Funktio runValidation suorittaa pyynnön validoinnin sovelluksessa. Funktio ottaa kolme argumenttia: req (pyyntöobjekti), res (vastausobjekti) ja next

(takaisinkutsufunktio, joka siirtää ohjauksen eteenpäin). Ensin haetaan virheet pyynnöstä käyttämällä validationResult(req). Jos virheitä on, se lähettää vastauksen, jonka tila on 422 ja virheviestin, joka on ensimmäinen virheviesti virheiden taulukossa. Jos virheitä ei ole, kutsutaan next-funktiota, joka siirtää ohjauksen eteenpäin. Funktion tietotyypit ovat Request, Response ja NextFunction Expressin tyyppimäärittelyistä. Tyyppi validationResult on express-validator-paketista, jota käytetään pyyntöjen validointiin Express-sovelluksissa (kuva 17).

```
api > src > validators > TS index.ts > ...
1   import { Request, Response, NextFunction } from "express";
2   import { validationResult } from "express-validator";
3
4   export const runValidation = (
5     req: Request,
6     res: Response,
7     next: NextFunction
8   ) => {
9     const errors = validationResult(req);
10    if (!errors.isEmpty()) {
11      return res.status(422).json({
12        error: errors.array()[0].msg,
13      });
14    }
15    next();
16  };
```

Kuva 17. Validoinnit suorittava middleware.

Kaksi middleware-funktiota määrittelevät validointisäännöt käyttäjän rekisteröitymiselle ja sisäänkirjautumiselle sovelluksessa.
Funktio userSignupValidator sisältää validointiketjuja (ValidationChain), jotka määrittelevät säännöt käyttäjän nimen, sähköpostin ja salasanan validointiin rekisteröitymisen yhteydessä. Nämä säännöt ovat seuraavat:

- name: Nimi ei saa olla tyhjä. Jos se on, palautetaan viesti “Name is required”.
- email: Sähköpostiosoitteen on oltava validi. Jos se ei ole, palautetaan viesti “Must be a valid email address”.
- password: Salasanan on oltava vähintään 6 merkkiä pitkä. Jos se ei ole, palautetaan viesti “Password must be at least 6 characters long”.

Funktio userSigninValidator sisältää validointiketjuja, jotka määrittelevät säännöt käyttäjän sähköpostin ja salasanan validointiin sisäänkirjautumisen yhteydessä. Nämä säännöt ovat seuraavat:

- email: Sähköpostiosoite ei saa olla tyhjä. Jos se on, palautetaan viesti “Email is required”.
- password: Salasana ei saa olla tyhjä. Jos se on, palautetaan viesti “Password is required”.

Metodi check on express-validator-paketista, jota käytetään pyyntöjen validointiin Express-sovelluksissa (kuva 18).

api > src > validators > TS auth.ts > [e] userSigninValidator

```
1   import { check, ValidationChain } from "express-validator";
2
3   export const userSignupValidator: ValidationChain[] = [
4     check("name").not().isEmpty().withMessage("Name is required"),
5     check("email").isEmail().withMessage("Must be a valid email address"),
6     check("password")
7       .isLength({ min: 6 })
8       .withMessage("Password must be at least 6 characters long"),
9   ];
10
11  export const userSigninValidator: ValidationChain[] = [
12    check("email").not().isEmpty().withMessage("Email is required"),
13    check("password").not().isEmpty().withMessage("Password is required"),
14  ];
15
```

Kuva 18. Rekisteröitymis- ja kirjautumistietojen validointien määrittely.

Backendin toteutuksen viimeisenä vaiheena oli reittien määrittely, sekä middleware-funktioiden liittäminen niihin (kuvat 19 & 20).

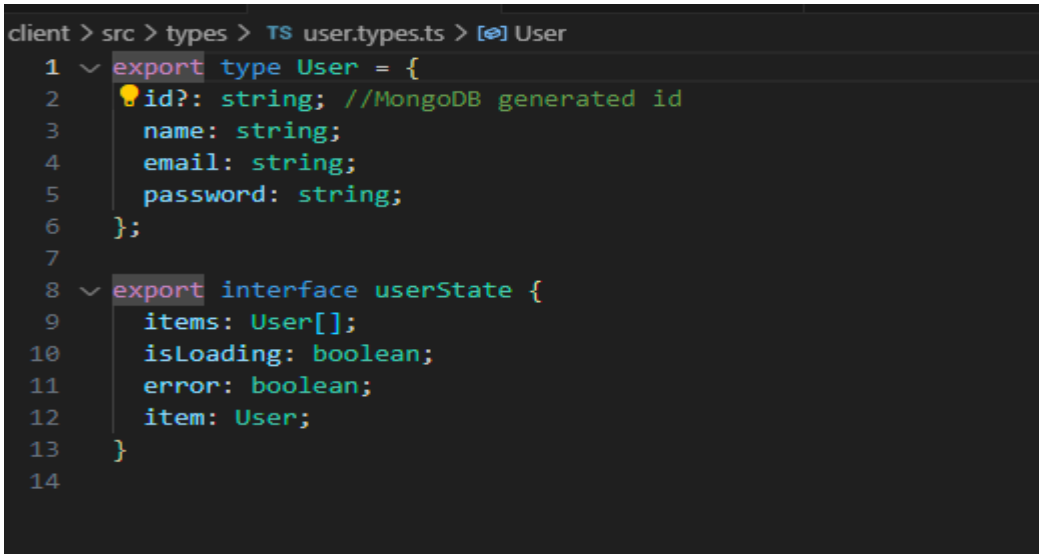
```
import express from "express";
import { runValidation } from "../validators";
import { userSignupValidator, userSigninValidator } from "../validators/auth";
import {
  signup,
  accountActivation,
  signin,
} from "../controllers/auth.controller";

const router = express.Router();

router.post("/signup", userSignupValidator, runValidation, signup);
router.post("/account-activation", accountActivation);
router.post("/signin", userSigninValidator, runValidation, signin);

export default router;
```

Kuva 19. Middlewarejen käyttöönotto rekisteröitymisen, tilin aktivoimisen ja kirjautumisen reitteihin.

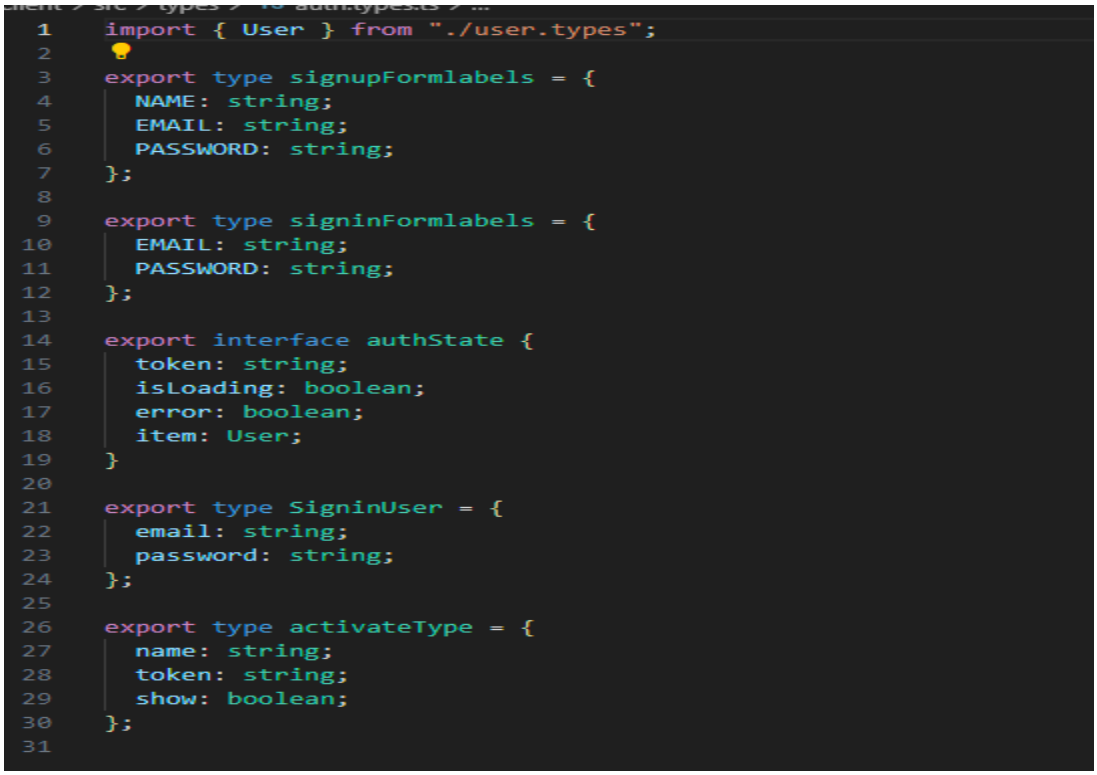
Kuva 20. Middlewarejen liittäminen käyttäjään liittyviin reitteihin.

4.3 Frontend

Frontend toteutus aloitettiin myös tietotyyppien luomisella. Tietotyypit tuli määritellä käyttäjälle, käyttäjän tilalle ja autentikointiin liittyville komponenteille sekä autentikoinnin tilalle (kuvat 21 & 22).



Kuva 21. Käyttäjän ja käyttäjän tilan tietotyyppien määrittely.



```
import { User } from "./user.types";

export type signupFormlabels = {
  NAME: string;
  EMAIL: string;
  PASSWORD: string;
};

export type signinFormlabels = {
  EMAIL: string;
  PASSWORD: string;
};

export interface authState {
  token: string;
  isLoading: boolean;
  error: boolean;
  item: User;
}

export type SigninUser = {
  email: string;
  password: string;
};

export type activateType = {
  name: string;
  token: string;
  show: boolean;
};
```

Kuva 22. Käyttäjän autentikointiin liittyvien tietotyyppien määrittäminen.

Seuraavassa vaiheessa lisättiin kirjautumisen staattiset näkymät. Lopuksi toteutettiin kirjautumisen toiminnallisuudet ja reittien suojaukseen liittyvät komponentit, jotka liitettiin aiemmin luotuihin staattisiin näkymiin.

4.3.1 Kirjautumisen toiminnallisuudet ja tilan määrittely

Kaikki funktiot ovat luotu käyttämällä createAsyncThunk-funktiota, joka on osa Redux Toolkit -kirjastoa, jonka olen valinnut tilanhallinnan työkaluksi projektiin. API_URL on määritelty ympäristömuuttujissa. Funktio signupUser luo uuden käyttäjän sovelluksessa. Tämä funktio luo asynkronisen toiminnon, joka voidaan liittää sovelluksen tilaan. Se ottaa kaksi argumenttia: toiminnon tyyppi (“auth/signup”) ja toiminnon suorittaja, joka on asynkroninen funktio. Toiminnon suorittaja ottaa argumenttina newUser-objektin, joka on User-tyyppiä. Se määrittelee HTTP-pyynnön konfiguraation, joka sisältää pyynnön metodin (POST), URL:n (${API_URL}/auth/signup), datan (newUser) ja headers (tyhjä objekti). Se yrittää lähettää HTTP-pyynnön käyttämällä axios-kirjastoa. Jos

pyyntö onnistuu, se palauttaa objektin, joka sisältää palvelimen vastauksen datan ja tilakoodin. Jos pyynnössä ilmenee virhe, se otetaan kiinni ja heitetään uusi virhe, joka sisältää palvelimen vastauksen ja virheviestin. Funktion signinUser avulla käyttäjä voi kirjautua sovellukseen ja se on toteutettu samalla periaatteella kuin signUpUser funktio. Funktio activateAccount aktivoi käyttäjän tilin sovelluksessa. Toiminnon suorittaja ottaa argumenttina tokenin-arvon, joka on merkkijono ja mikä lähetetään POST-pyynnön rungossa (body). Virheet ovat käsitelty samalla tavoin kuin aikaisemmin kuvatuissa rekisteröitymisen ja kirjautumisen suorittavissa funktioissa (kuva 23). (Abramov 2023.)

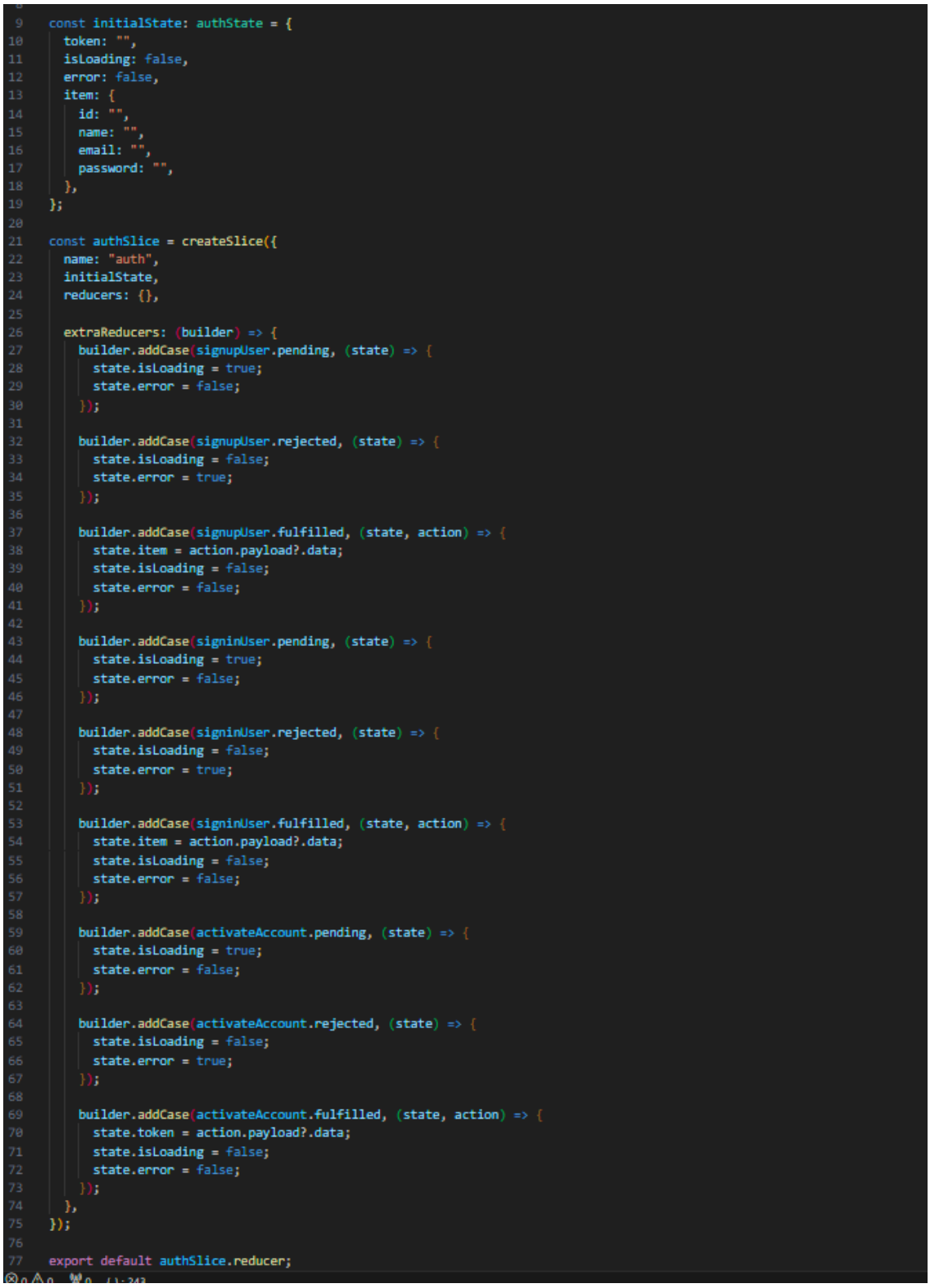
```
export const signupUser = createAsyncThunk(
  "auth/signup",
  async (newUser: User) => {
    const config = {
      method: "POST",
      url: `${API_URL}/auth/signup`,
      data: newUser,
      headers: {},
    };
    try {
      let res = await axios(config);
      return { data: res.data, status: res.status };
    } catch (error: any) {
      throw new Error(error.response.data.error);
    }
  }
);

export const signinUser = createAsyncThunk(
  "auth/signin",
  async (data: SigninUser) => {
    const config = {
      method: "POST",
      url: `${API_URL}/auth/signin`,
      data: data,
      headers: {},
    };
    try {
      let res = await axios(config);
      return { data: res.data, status: res.status };
    } catch (error: any) {
      throw new Error(error.response.data.error);
    }
  }
);

export const activateAccount = createAsyncThunk(
  "auth/activate",
  async (token: string) => {
    try {
      const response = await axios.post(`${API_URL}/auth/account-activation`, {
        token,
      });
      return response.data;
    } catch (error: any) {
      throw new Error(error.response.data.error);
    }
  }
```

Kuva 23. Client puolen rekisteröitymisen, tilin aktivoimisen ja kirjautumisen metodit.

Redux Toolkit -kirjaston createSlice-funktiolla luotu authSlice, hallitsee käyttäjän kirjautumistilan sovelluksessa. Tilanhallinnassa authSlice on viipale/slice, joka sisältää tilan, toimintojen luojat ja reducerit käyttäjän kirjautumistilalle.

- Alkutila (initialState) määrittelee authSlicen alkuarvot. Tässä tapauksessa se sisältää token, isLoading, error ja item-arvot.
- createSlice ottaa argumenttina objektin, joka määrittelee slicen nimen (“auth”), alkutilan ja reducerit. Reducerit ovat funktioita, jotka päivittävät tilaa toimintojen perusteella.
- extraReducers-kenttä määrittelee, miten tila päivittyy asynkronisten toimintojen, kuten signupUser, signinUser ja activateAccount, perusteella. Jokaiselle toiminnolle on kolme tilaa: pending (käynnissä), rejected (hylätty) ja fulfilled (täytetty). Jokaisessa tilassa tilaa päivitetään eri tavalla.

Esimerkiksi, kun signupUser-toiminto on käynnissä (pending), isLoading asetetaan arvoon true ja error asetetaan arvoon false. Kun toiminto on täytetty (fulfilled), item päivitetään palvelimen vastauksen datalla, isLoading asetetaan arvoon false ja error asetetaan arvoon false. Jos toiminto hylätään (rejected), isLoading asetetaan arvoon false ja error asetetaan arvoon true (kuva 24). (Abramov 2023.)



```
const initialState: authState = {
  token: "",
  isLoading: false,
  error: false,
  item: {
    id: "",
    name: "",
    email: "",
    password: "",
  },
};

const authSlice = createSlice({
  name: "auth",
  initialState,
  reducers: {},

  extraReducers: (builder) => {
    builder.addCase(signupUser.pending, (state) => {
      state.isLoading = true;
      state.error = false;
    });

    builder.addCase(signupUser.rejected, (state) => {
      state.isLoading = false;
      state.error = true;
    });

    builder.addCase(signupUser.fulfilled, (state, action) => {
      state.item = action.payload?.data;
      state.isLoading = false;
      state.error = false;
    });

    builder.addCase(signinUser.pending, (state) => {
      state.isLoading = true;
      state.error = false;
    });

    builder.addCase(signinUser.rejected, (state) => {
      state.isLoading = false;
      state.error = true;
    });

    builder.addCase(signinUser.fulfilled, (state, action) => {
      state.item = action.payload?.data;
      state.isLoading = false;
      state.error = false;
    });

    builder.addCase(activateAccount.pending, (state) => {
      state.isLoading = true;
      state.error = false;
    });

    builder.addCase(activateAccount.rejected, (state) => {
      state.isLoading = false;
      state.error = true;
    });

    builder.addCase(activateAccount.fulfilled, (state, action) => {
      state.token = action.payload?.data;
      state.isLoading = false;
      state.error = false;
    });
  },
});

export default authSlice.reducer;
```

Kuva 24. Metodien tilan määrittely.

App-komponentti kääritään react-redux-kirjastosta tuodun Provider-komponentin sisälle. App-komponentti sisältää kaikki muut sovelluksen

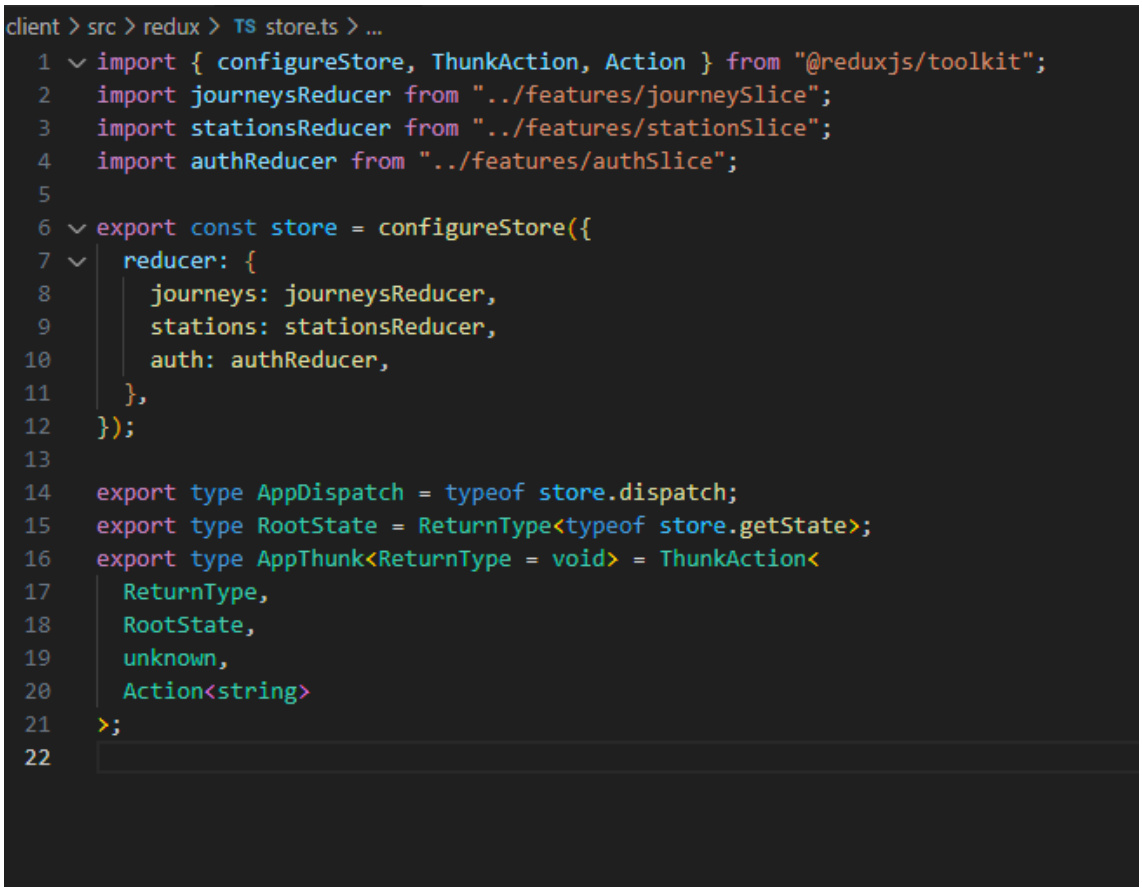
komponentit, joten Provider-komponentti mahdollistaa Redux-storen saatavuuden kaikille sen lapsikomponenteille. (kuva 25)

client > src > TS index.tsx > ...

```tsx
import React from "react";
import App from "../src/components/app/App";
import { createRoot } from "react-dom/client";
import { Provider } from "react-redux";
import { store } from "./redux/store";

import "./index.css";

const container = document.getElementById("root")!;
const root = createRoot(container);

root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);

```

Kuva 25. App-komponentti käärittynä Providerin sisälle.

Provider-komponentti sisältää Redux-storen minne on määritelty sovelluksen tilaa hallitsevat funktiot. Funktio configureStore luo Redux Storen, joka on sovelluksen globaali tila. Sovelluksen storen reducerit ovat journeysReducer, stationsReducer ja authReducer. Koodi määrittelee myös joitakin tyyppejä, jotka auttavat Reduxin kanssa. AppDispatch on tyypitetty store.dispatch-funktioksi, RootState on tyypitetty store.getState-funktion palautusarvoksi ja AppThunk on tyypitetty ThunkAction-funktioksi (kuva 26).



Kuva 26. Autentikoinnin yleisen tilan liittäminen sovellukseen.

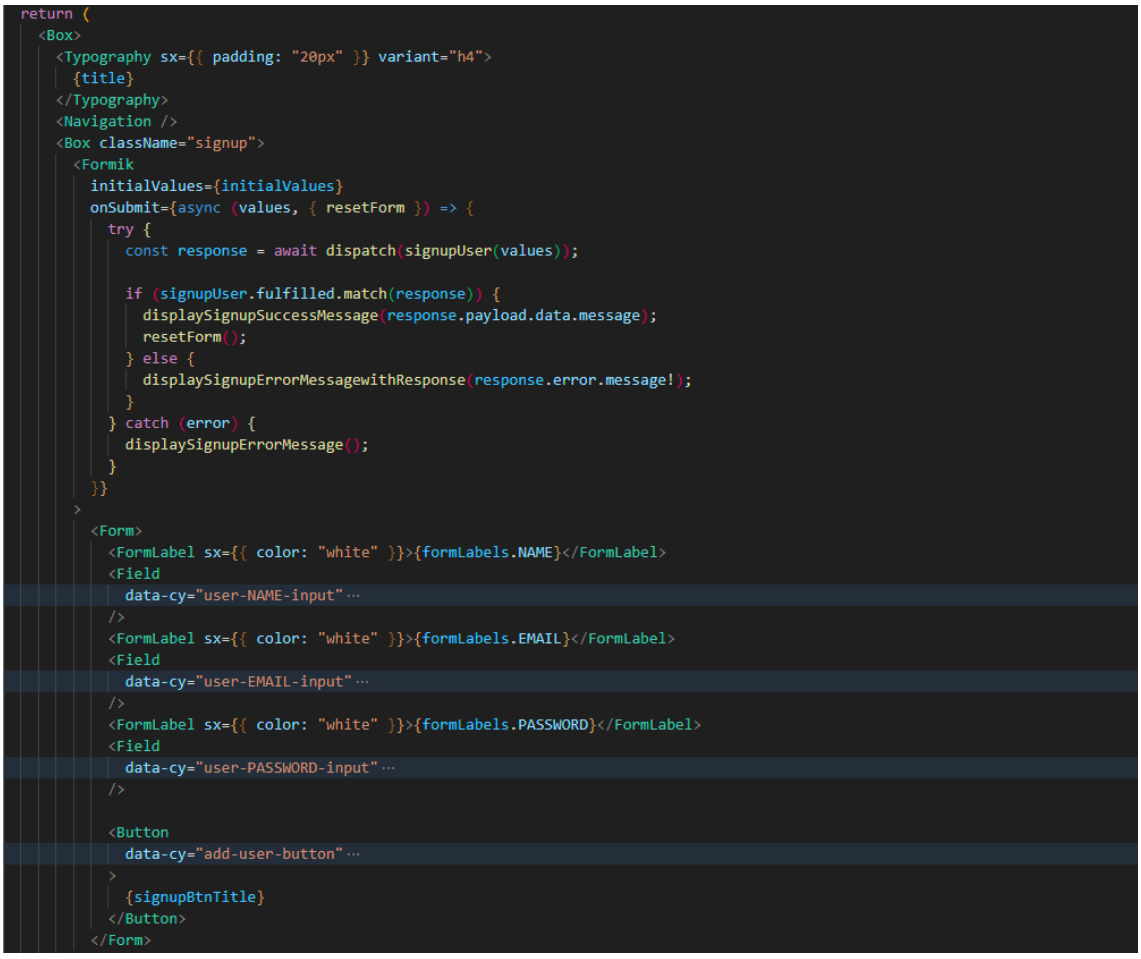
Provider välittää storen sisällön kaikille lapsikomponenteille, joten toimintoja voi kutsua mistä kohtaa sovellusta tahansa eikä niitä tarvitse erikseen välittää niille parametreina. Providerin käytöllä vältetään myös React-kehityksessä tunnettu ”props drilling” (GeeksforGeeks 2021). (Abramov 2015.)

4.3.2 Näkymien komponentit ja reittien suojaus

Signup on React-komponentti, joka renderöi lomakkeen käyttäjän rekisteröitymiseksi. Lomakkeessa on kolme kenttää: nimi, sähköposti ja salasana. Lomake on rakennettu käyttäen Formik-kirjastoa, joka tarjoaa yksinkertaisen tavan hallita lomakkeen tilaa ja validointia (Formium 2020).

Lomake lähetetään käyttämällä onSubmit-funktiota, joka välitetään Formik-komponentille parametrina. Kun lomake lähetetään, onSubmit-funktio lähettää

toiminnon käyttäjän rekisteröimiseksi käyttäen signupUser-funktiota, joka on tuotu auth.services-moduulista. Jos rekisteröinti onnistuu, käyttäjä ohjataan kotisivulle. Jos ilmenee virhe, näytetään virheilmoitus käyttäen displayToast-funktiota. Lomake on tyylitelty Material UI -komponenteilla, mukaan lukien Box, Button, FormLabel ja Typography (Material UI 2023). Lomake sisältää myös Navigation-komponentin, joka tarjoaa linkkejä muihin sovelluksen sivuihin (kuva 27).



```
return (
  <Box>
    <Typography sx={{ padding: "20px" }} variant="h4">
      {title}
    </Typography>
    <Navigation />
    <Box className="signup">
      <Formik
        initialValues={initialValues}
        onSubmit={async (values, { resetForm }) => {
          try {
            const response = await dispatch(signupUser(values));

            if (signupUser.fulfilled.match(response)) {
              displaySignupSuccessMessage(response.payload.data.message);
              resetForm();
            } else {
              displaySignupErrorMessagewithResponse(response.error.message!);
            }
          } catch (error) {
            displaySignupErrorMessage();
          }
        }}
      >
        <Form>
          <FormLabel sx={{ color: "white" }}>{formLabels.NAME}</FormLabel>
          <Field
            data-cy="user-NAME-input" ···
          />
          <FormLabel sx={{ color: "white" }}>{formLabels.EMAIL}</FormLabel>
          <Field
            data-cy="user-EMAIL-input" ···
          />
          <FormLabel sx={{ color: "white" }}>{formLabels.PASSWORD}</FormLabel>
          <Field
            data-cy="user-PASSWORD-input" ···
          />

          <Button
            data-cy="add-user-button" ···
          >
            {signupBtnTitle}
          </Button>
        </Form>
```

Kuva 27. Rekisteröitymisen komponentti.

Signin on React-komponentti, joka renderöi kirjautumislomakkeen. Se tuo useita moduuleja React- ja Material-UI-kirjastoista. Lomake on rakennettu Formik-kirjastolla. Lomakkeessa on kaksi kenttää, yksi sähköpostille ja toinen salasanalle. Lomakkeessa on myös lähetä-painike. Kun käyttäjä lähettää

lomakkeen, onSubmit-funktiota kutsutaan. Funktio lähettää toiminnon Redux-storeen käyttäjän kirjautumiseksi sisään. Jos kirjautuminen onnistuu, käyttäjä ohjataan kotisivulle. Jos kirjautuminen epäonnistuu, virhesanoma näytetään käyttäen react-toastify-kirjastoa (kuva 28) (Khadra 2022).

```
const signinBtnTitle: string = "SIGNIN";
return (
  <Box className="signin">
    <Formik
      initialValues={initialValues}
      onSubmit={async (values, { resetForm }) => {
        try {
          const response = await dispatch(signinUser(values));

          if (signinUser.fulfilled.match(response)) {
            authenticate(response, () => {
              displaySignInSuccessMessage(response.payload.data.user.name);
              resetForm();
              navigate("/home");
            });
          } else {
            displaySignInErrorMessagewithResponse(response.error.message!);
          }
        } catch (error) {
          displaySignInErrorMessage();
        }
      }}
    >
      <Form>
        <FormLabel sx={{ color: "white" }}>{formLabels.EMAIL}</FormLabel>
        <Field
          data-cy="user-EMAIL-input"···
        />
        <FormLabel sx={{ color: "white" }}>{formLabels.PASSWORD}</FormLabel>
        <Field
          data-cy="user-PASSWORD-input"···
        />
        <Button
          data-cy="add-user-button"···
        >
          {signinBtnTitle}
        </Button>
```

Kuva 28. Kirjautumisen komponentti.

4.3.3 LocalStorage ja Cookiet

Funktioden setCookie, setLocalStorage ja authenticate avulla hallitaan selaimen localStoragea ja cookieta eli evästettä. Funktio setCookie asettaa evästeen annetulla avaimella ja arvolla. Funktio setLocalStorage asettaa selaimen localStorageen tiedon annetulla avaimella ja arvolla. Authenticate-funktio ottaa kaksi parametria, response ja next. Se asettaa evästeen avaimella “token” ja arvolla response.payload.data.token. Se asettaa myös localStorageen tiedon avaimella “user” ja arvolla response.payload.data.user. Lopuksi kutsutaan next-funktiota (kuva 29).

```
import cookie from "js-cookie";

export const setCookie = (key: string, value: any) => {
  if (typeof window !== "undefined") {
    cookie.set(key, value, {
      expires: 1,
    });
  }
};

export const setLocalStorage = (key: string, value: any) => {
  if (typeof window !== "undefined") {
    localStorage.setItem(key, JSON.stringify(value));
  }
};

export const authenticate = (response: any, next: any) => {
  setCookie("token", response.payload.data.token);
  setLocalStorage("user", response.payload.data.user);
  next();
};
```

Kuva 29. Kirjautumisessa tarvittavat funktiot evästeen/cookien ja localStoragen asettamiseksi.

Käyttäjän kirjautumistiedot tarkistetaan evästeestä ja localStoragesta getCookie ja isAuth -funktioiden avulla. Funktio getCookie ottaa key-argumentin ja palauttaa evästeen arvon, jolla on kyseinen avain. Se tarkistaa ensin, onko window-objekti määritelty. Jos näin on, se käyttää cookie.get(key)-metodia saadakseen evästeen arvon. Funktio isAuth tarkistaa, onko käyttäjä tunnistettu. Se tekee tämän etsimällä “token” avaimella olevaa evästettä ja localStorage-itemiä nimeltä “user”. Jos molemmat ovat olemassa, käyttäjä on tunnistettu (kuva 30).

```
export const getCookie = (key: string) => {
  if (typeof window !== "undefined") {
    return cookie.get(key);
  }
};

export const isAuth = () => {
  if (typeof window !== "undefined") {
    const cookieChecked = getCookie("token");
    if (cookieChecked) {
      const storedUser = localStorage.getItem("user");
      if (storedUser !== null) {
        return JSON.parse(storedUser);
      } else {
        return false;
      }
    }
  }
};
```

Kuva 30. Evästeen/Cookien ja localStoragen tarkistavat funktiot.

Evästeiden ja localStoragen tyhjentäminen hoidetaan removeCookie, removeLocalStorage ja signout funktioiden avulla. Funktio removeCookie ottaa key-parametrin, joka on poistettavan evästeen nimi. Jos window-objekti on määritelty, funktio kutsuu cookie.remove-metodia poistaakseen evästeen, jossa on määritetty avain ja vanhenemisaika 1 päivä. Funktio removeLocalStorage ottaa key-parametrin, joka on poistettavan paikallisen tallennustiedon nimi. Jos window-objekti on määritelty, funktio kutsuu localStorage.removeItem-metodia poistaakseen datan, jossa on määritetty avain. Funktio signout kutsuu sekä removeCookie- että removeLocalStorage-funktioita avaimilla “token” ja “user”, joiden arvot poistetaan evästeistä ja localStoragesta (kuva 31).

```
export const removeCookie = (key: string) => {
  if (typeof window !== "undefined") {
    cookie.remove(key, { expires: 1 });
  }
};

export const removeLocalStorage = (key: string) => {
  if (typeof window !== "undefined") {
    localStorage.removeItem(key);
  }
};

export const signout = (next: any) => {
  removeCookie("token");
  removeLocalStorage("user");
  next();
};
```

Kuva 31. Uloskirjautuminen. LocalStoragen ja evästeen tyhjentäminen.

React-komponenttia nimeltä PrivateRoute käytetään rajoittamaan pääsyä tiettyihin sovelluksen osiin käyttäjän todennuksen perusteella. PrivateRoute-komponentti ottaa yhden propsin: children, joka on tyyppiä React.ReactElement. PrivateRoute-komponentille välitetty children renderöidään, mikäli sen sisältämä isAuth ehto on tosi. Tämä tarkoittaa, että reitin todellinen sisältö näytetään vain, jos käyttäjä on autentikoitu. Mikäli käyttäjä ei ole autentikoitu renderöidään näkymä, mikä sisältää viestin ja napin mitä painamalla käyttäjä siirtyy kirjautumisnäkymään (kuva 32).

```
import { useNavigate } from "react-router-dom";
import { isAuth } from "../helpers/helpers";
import { Box, Button } from "@mui/material";

import "./PrivateRoute.css";

const PrivateRoute = ({ children }: { children: React.ReactElement }) => {
  const navigate = useNavigate();

  if (!isAuth()) {
    return (
      <Box className="privateRoute">
        <h2>You are not allowed here!</h2>
        <h3>Only authorized users are allowed.</h3>
        <Button onClick={() => navigate("/signin")}>SIGNIN</Button>
      </Box>
    );
  }

  return children;
};

export default PrivateRoute;
```

Kuva 32. PrivateRoute-komponentti reitin suojaamiseen.

AdminRoute-komponentti on toiminnaltaan samankaltainen kuin PrivateRoute-komponentti. Sillä eroavaisuudella, että käyttäjän tulee olla autentikoitu ja käyttäjän rooli tulee olla admin.

```
const AdminRoute = ({ children }: { children: React.ReactElement }) => {
  const navigate = useNavigate();

  if (isAuth() && isAuth().role === "admin") {
    return children;
  }

  return (
    <Box className="adminRoute">
      <h2>You are not allowed here!</h2>
      <h3>Only admin users can access.</h3>
      <Button onClick={() => navigate("/home")}>HOME</Button>
    </Box>
  );
};

export default AdminRoute;
```

Kuva 33. AdminRoute komponentti admin roolin vaaditun reitin suojaamiseen.

Sovelluksen reitit voidaan kääriä PrivateRoute ja AdminRoute-komponenttien sisään sen mukaan, kuinka sovellukseen pääsyä halutaan rajoittaa (kuva 34).

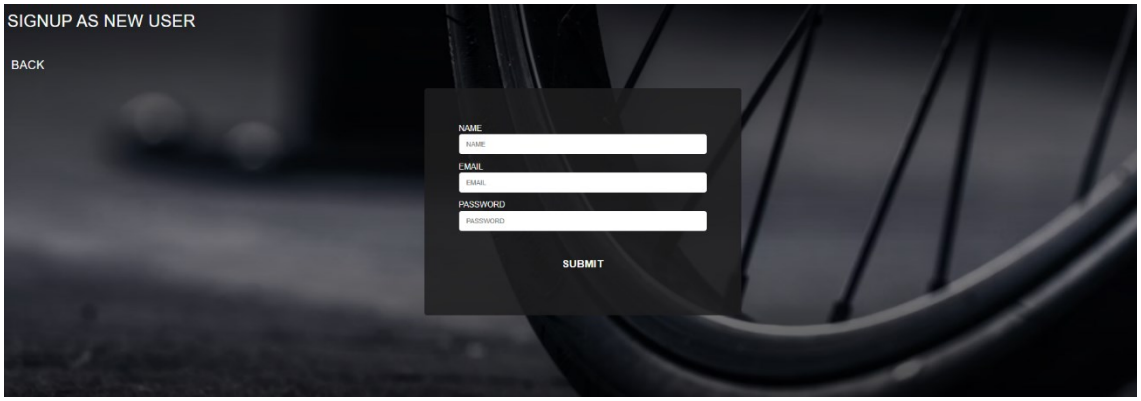
```
return (
  <Box
    sx={{ ···
    }}
  >
    <Box className="img__container">
      <Box className="img__container__gradient">
        <ToastContainer />
        <Typography sx={{ textAlign: "center" }} variant="h1">
          {homeTitle}
        </Typography>

        <BrowserRouter>
          <Navigation />
          <Routes>
            <Route path="/" element={<LandingPage />} />
            <Route
              path="/signup"
              element={<Signup displayToast={displayToastInApp} />}
            />
            <Route path="/auth/activate/:token" element={<Activate />} />
            <Route
              path="/signin"
              element={<Signin displayToast={displayToastInApp} />}
            />
            <Route
              path="/admin"
              element={
                <AdminRoute>
                  <Admin />
                </AdminRoute>
              }
            />
            <Route
              path="/user"
              element={
                <PrivateRoute>
                  <User />
                </PrivateRoute>
              }
            />
```

Kuva 34. Route komponenttien sisällä reitit suojattuina PrivateRoute ja Admin Route komponenteilla.

5 Tulokset

Toteutus mukailee Ryan Dhungelin, MERN Stack Web Development with Ultimate Authentication, tutoriaalin JavaScript toteutuksen periaatetta. Yleisesti katsoen Dhungelin tutoriaalia noudattamalla saa hyvän käsityksen MERN-teknologiolla toteutetusta autentikaatiosta. Dhungelin ratkaisua ei voinut kuitenkaan suoraan hyödyntää työn toteutuksessa joidenkin teknologia poikkeavuuksien ja yhteensopivuus ongelmien vuoksi. Myös joitain muutoksia tehtiin, että saatiin ratkaisu ajantasaiseksi. Esimerkiksi Dhungel käyttää ratkaisussaan salasanan salaamiseen SHA-1-algoritmia, missä on tunnettu olevan haavoittuvuuksia jo vuodesta 2010 lähtien. Tämän työn toteutuksessa käytetään käyttäjän salasanan salaamiseen SHA-256-algoritmia. (Peyrott 2018, 64.)

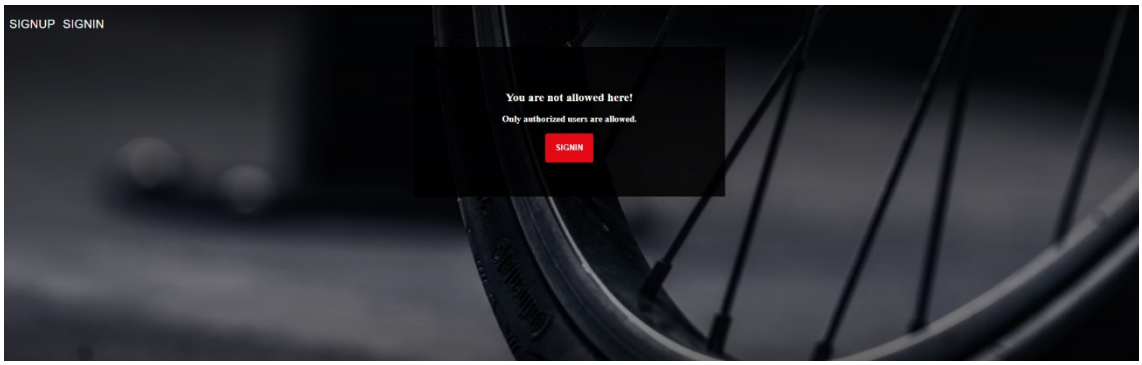
Toteutusvaiheen tuloksena saatiin liitettyä sovellukseen toimiva JWT-standardia tukevia kirjastoja hyödyntävä käyttäjän autentikointi, autorisointi ja tietojen tietokantaan tallentaminen. Käyttäjälle tämä tarkoittaa tilin luomista, kirjautumista ja rajattua pääsyä (kuvat 35 & 36).



Kuva 35. Lomake tilin luomiseen.

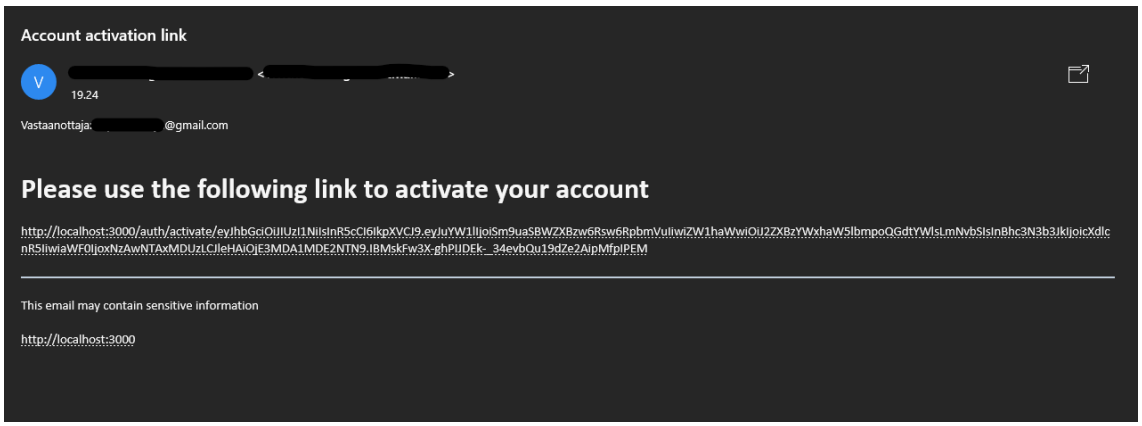
Kuva 36. Onnistunut lomakkeen luominen ja viesti serveriltä.

Sovelluksen käyttäminen on mahdollista ainoastaan tilin luomisen jälkeen kirjautumalla sovellukseen. Mikäli käyttäjä, joka ei ole kirjautunut sovellukseen, päätyy suojattuun endpointtiin, näytetään käyttäjälle informatiivinen varoitusviesti ja ohjataan käyttäjää kirjautumaan sovellukseen (kuva 37).



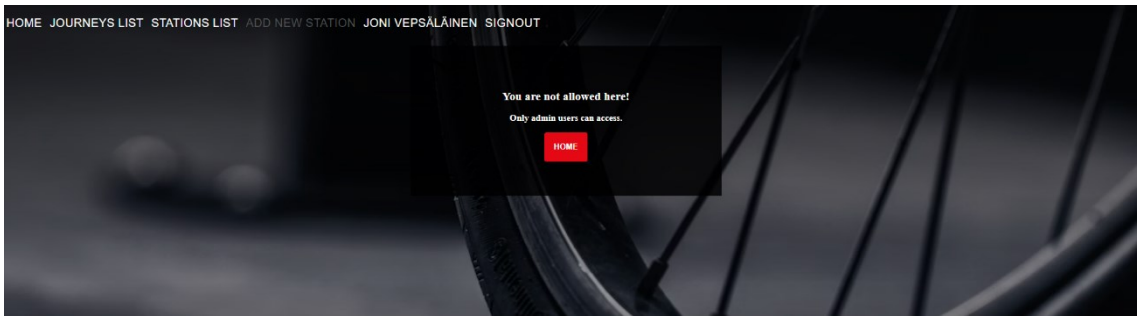
Kuva 37. Suojattu endpoint. Ilmoitus käyttäjälle.

Käyttäjätili tulee myös aktivoida sähköpostiin välitetyn linkin avulla. Tällä tavoin varmistetaan sähköpostin ja käyttäjän aitous. Linkki on endpoint mitä klikkaamalla käyttäjä päätyy tilin aktivointisivulle. Endpoint sisältää myös tokenin, joka pitää sisällään käyttäjän tiedot. Tilin aktivointiprosessissa käyttäjän tiedot otetaan tokenista ja viedään tietokantaan (kuvat 38).



Kuva 38. Sähköposti sisältää endpointin tilin aktivoimiseen.

Kun käyttäjän tiedot ovat tallennettuna tietokantaan, voi tunnuksilla kirjautua sovellukseen. Sovelluksen käyttöä on myös rajattu käyttäjän roolin mukaisesti. Käyttäjälle määritellään alustavasti user-rooli tilin luomisen yhteydessä. Tietokantaan voi lisätä dataa ainoastaan admin-roolin omaava käyttäjä. Lisäystoimintojen sijainnit ovat suojattujen endpointtien takana. Käyttäjän roolia on toistaiseksi mahdollista muuttaa ainoastaan MongoDB Atlas -tietokantapalvelun käyttöliittymän avulla (kuva 42).



Kuva 42. Add new station-toiminto sallittu vain admin käyttäjille.

6 Pohdinta

Toteutusvaihe sujui ilman suurempia haasteita tai yllättäviä ongelmia. Kehitystyössä edettiin suunnitelman mukaisesti ja käytettiin suunnitelman mukaisia kehitystyökaluja. Työ aloitettiin perehtymällä OAuth 2.0 -protokollaan ja JWT-standardiin. Perehtymisen jälkeen toteutusvaiheessa saatiin kehitettyä kaikkia sovelluksen tasoja hyödyntävä, toimiva käyttäjän autentikointi MERN-sovellukseen. Toteutuksessa hyödynnettiin JWT-standardia tukevia kirjastoja, sekä lisättiin reitteihin autentikointia tukevat middlewaret.

Vaikka perehtymisvaiheen tavoitteet saavutettiin täysin, toteutusvaiheessa jouduttiin muuttamaan suunnitelmaa OAuth 2.0 -protokollan osalta. Aikataulusyistä OAuth 2.0 -protokollan hyödyntäminen jätettiin kokonaan pois toteutuksesta. OAuth 2.0 -protokollaa ja sitä tukevia kirjastoja hyödyntämällä, olisi saavutettu käyttäjän olemassa olevien, esimerkiksi Google-tunnuksien hyödyntäminen tilin luomisen ja kirjautumisen yhteydessä. Tavoitteiden mukainen toimiva käyttäjän autentikointi MERN-sovellukseen voidaan kuitenkin määritellä saavutetuksi. Toteutuksen lähteenä sovellettiin JavaScript-toteutusta vuodelta 2019 mitä muokattu toteutukseen sopivaksi (Dhungel 2019).

Perehtymisvaihe opetti OAuth 2.0 -protokollan ja JWT-standardin toimintojen periaatteista. Kun opitut asiat sovellettiin toteutusvaiheessa käytäntöön, sai työn aikana hyvän kokonaiskuvan siitä, kuinka autentikointi voidaan toteuttaa

valituilla teknologioilla. Raportointivaihe olisi ollut helpompi, mikäli toteutusvaiheen aikana olisi rutiininomaisesti kirjattu ylös kehitystyön vaiheet.

7 Kehitysideat

Toteutus on ratkaisuna toimiva ja täyttää suunnitelmassa asetetut vaatimukset, mutta kuitenkin mahdolliseen jatkokehitykseen jäi joitakin asioita mitä on hyvä mainita. Kirjautumisen toimintoa tulisi parantaa. Käyttäjän kirjautumisyrityksiä ei tällä hetkellä rajoiteta mitenkään, mikä mahdollistaa loputtoman määrän yrityksiä salasanan arvaukselle. Salasanalle ei tällä hetkellä ole myöskään muita vaatimuksia kuin, että se tulee olla määritelty ja sisältää vähintään 6-merkkiä. Salasanalle tulisi asettaa enemmän vaatimuksia pituuden ja esimerkiksi erikoismerkkien ja numeroiden suhteen. Salasanojen salauksessa voitaisiin myös mahdollisesti hyödyntää hyvin tuettua bcrypt kirjastoa (LeBlanc, 2016).

OAuth 2.0 -protokollan hyödyntämisen lisääminen toteutukseen JWT:n rinnalle. OAuth 2.0 -protokollassa ei määritellä sen käyttämien tokeneiden muotoa, joten JWT:t sopivat hyvin tähän tarkoitukseen. Allekirjoitettuun JWT:hen voi sisällyttää kaikki tarvittavat tiedot resurssin käyttöoikeuksien erottamiseksi. Ne voivat kantaa vanhentumispäivää ja ne ovat allekirjoitettuja. Useat federated identityn tarjoajat myöntävät pääsytunnuksia JWT-muodossa. (Peyrott, 2018, 19.)

Lähteet

Abramov, D, 2015. React Redux. https://react-redux.js.org/ . 27.11.2023.

Abramov, D. 2023. Redux Toolkit. https://redux-toolkit.js.org/ . 27.11.2023.

Bihis, C. 2015. Mastering Oauth 2.0. Birmingham: Packt Publishing. Ebook Central, https://karelia.finna.fi .

Clinton, D. 2023. What is Node.js? Server-Side JavaScript Development Basics. FreeCodeCamp. https://www.freecodecamp.org/news/node-js-basics/ . 21.9.2023.

Dhungel, R. 2019. MERN Stack Web Development with Ultimate Authentication. Pack Publishing. Training video. https://learning.oreilly.com/videos/mern-stack-web/9781800204799/ . 25.10.2023.

Khadra, F. 2022. ToastContainer. https://fkhadra.github.io/react-toastify/api/toast-container/ . 27.11.2023.

Formium, Inc. 2020. Formik docs. https://formik.org/ . 27.11.2023.

GeeksforGeeks. 2021. What is prop drilling and how to avoid it?. https://www.geeksforgeeks.org/what-is-prop-drilling-and-how-to-avoid-it/ . 25.11.2023.

Gillis, A. 2023. Definition MongoDB. TechTarget. https://www.techtarget.com/searchdatamanagement/definition/MongoDB .21.9.2023.

LeBlanc, M, Messerschmidt, T. 2016. Identity and Data Security for Web Development, 2. Password Encryption, Hashing, and Salting. O'Reilly Media, Inc. https://learning.oreilly.com/library/view/identity-and-data/9781491937006/ . 23.11.2023.

Luukkainen, M. 2023. Full Stack Open, osa 1, Reactin alkeet. Helsingin Yliopisto. https://fullstackopen.com/osa1/reactin_alkeet . 2.8.2023.

Material UI. 2023. Material UI. https://mui.com/core/ . 27.11.2023.

Meta Platforms, Inc. 2023. Flux. https://facebookarchive.github.io/flux/docs/in-depth-overview/ . 20.11.2023.

Meta Platforms, Inc. 2023. React. https://legacy.reactjs.org/ . 2.8.2023.

OpenJS Foundation, Express. 2017. Express, Glossary. https://expressjs.com/en/resources/glossary.html . 21.9.2023.

Peuraniemi, T., Rapo, J. & Torppa, T. Full Stack Open, part 9 Background and introduction. Helsingin Yliopisto.

https://fullstackopen.com/en/part9/background_and_introduction . 19.9.2023.

Peyrott, S. 2016–2018. The JWT Handbook. Auth0 Inc.

https://auth0.com/resources/ebooks/jwt-handbook . 9.10.2023.