Opinnäytetyö (AMK)

Tieto- ja viestintätekniikka

2020

Tatu Laakso

WEB-SOVELLUSKEHITYS MERN-PINOLLA

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2020 | 39 sivua

Tatu Laakso

WEB-SOVELLUSKEHITYS MERN-PINOLLA

Verkkosovelluksia voidaan kehittää useilla eri tavoilla. Yksi yleisimpiä tapoja on käyttää ohjelmistopinoa. Ohjelmistopinolla tarkoitetaan kokoelmaa teknologioista, jotka yhdessä tukevat sovelluksen suorittamista.

Opinnäytetyön tarkoituksena oli tutkia MERN-ohjelmistopinoa, jonka muodostavat MongoDB, Express.js, Reactjs ja Node.js. Teknologioiden keskeiset käsitteet avattiin ja ohjelmistopinon avulla kehitettiin prototyyppisovellus. Prototyyppisovelluksen kehitys taltioitiin vaiheittain.

Sovelluksen kehitys aloitettiin suunnittelusta, jossa määritettiin sovelluksen rakenne ja toimintaperiaatteet. Palvelinpuolella luotiin ohjelmistorajapinta, joka kykeni tietojen vastaanottamiseen, lähettämiseen ja käsittelyyn. Käyttöliittymä luotiin yhden sivun mallin mukaisesti. Käyttöliittymässä keskityttiin erityisesti logiikan toimivuuteen ja uudelleenkäytettävyyteen.

Opinnäytetyön tuloksena syntyi toimiva prototyyppisovellus, joka täytti asetetut kriteerit. Sovelluksen kehityksessä tuotiin esille MERN-pinon ja JavaScriptin uusimpia ominaisuuksia.

ASIASANAT:

MERN, MongoDB, Express.js, Reactjs, Node.js, JavaScript

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and communications technology

2020 | 39 pages

Tatu Laakso

WEB APPLICATION DEVELOPMENT WITH MERN-STACK

Web applications can be developed in several different ways. One of the most common ways is to use a software stack. Software stack refers to a collection of technologies that together support the execution of an application.

The purpose of the thesis was to study the MERN software stack, which consists of MongoDB, Express.js, Reactjs and Node.js. The key concepts of the technologies were opened, and a prototype application was developed using the software stack. The development of the prototype application was recorded in stages.

The development of the application started with the design phase, which defined the structure and operating principles of the application. On the server side, a software interface was created that was capable of receiving, sending, and processing data. The user interface was created according to a one-page template. The user interface focused in particular on the functionality and reusability of client-side logic.

The result of the thesis was a working prototype application that met the set criteria. During the development of the application, the latest features of the MERN software stack and JavaScript were brought to the fore.

KEYWORDS:

MERN, MongoDB, Express.js, React Node.js

SISÄLTÖ

KUVAT

KÄYTETYT LYHENTEET

Backend	Koodi, joka ajetaan sivuston palvelimella.
Frontend	Koodi, joka ajetaan verkkoselaimessa käyttäjän silmien edessä ja jonka kanssa käyttäjä voi olla tekemisissä.
HTTP	Hypertext Transfer Protocol. Protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon
CSS	Cascading Style Sheets. Erityisesti WWW-dokumenteille kehitetty tyyliohjeiden laji.
JSON	JavaScript Object Notation. Tiedonvälityksessä käytetty avoimen standardin tiedostomuoto.
Väliohjelmisto	Ohjelmistokomponentti, joka toimii osien tai sovelluksien välisenä rajapintana tai palveluna.
CRUD	Create, Read, Update, Delete. Pysyvän varastoinnin neljä perusfunktiota.
Hookki	Funktio, jonka avulla pystytään kytkeytymään Reactin tilaan ja elinkaaren ominaisuuksiin.
Token	JSON-koodattu esitys vaatimuksista, jotka voidaan siirtää kahden osapuolen välillä.

1 JOHDANTO

HackerRankin vuoden 2020 selvityksessä ilmenee, että 38 % rekrytointivastaavista näkevät full stack -kehittäjien olevan tärkein täytettävä rooli teknologiayrityksissä vuonna 2020. Vaikka ominaisuudet, jotka määrittelevät full stack -kehittäjän, ovat väittelyn kohteena, useimmat ovat yhtä mieltä siitä, että heillä tulisi olla perustiedot kaikista ohjelmistopinon kerroksista ja heidän tulisi pystyä luomaan ”MVP” eli pienin julkaisukelpoinen tuote itsenäisesti. [1]

JavaScript-ohjelmointikielen ympärille on vuosien aikana muodostunut suuri ekosysteemi, joka mahdollistaa verkkopalvelujen kehittämisen yhtä kieltä käyttäen erilaisten ohjelmistopinojen avulla. Näitä ohjelmistopinoja ovat esimerkiksi MEAN-, MERN- ja MEVN-pinot. Opinnäytetyössä tarkastellaan MERN-pinoa, jossa JavaScriptiä käytetään palvelimella, tietokannassa ja käyttöliittymässä. Lyhenne MERN tulee nimistä MongoDB, Express.js, Reactjs ja Node.js. Pinossa MongoDB vastaa tietokannasta, Reactjs käyttöliittymästä, Node.js ja Express.js vastaavat palvelinpuolesta ja rajapinnasta. Opinnäytetyön tavoitteena on tarkastella MERN-pinon toimintaperiaatteita ja rakentaa pinolla prototyyppisovellus. Sovelluksen tavoitteena on ymmärtää kehittämisen eri vaiheet suunnittelusta testaukseen ja taltioida sekä avata luotuja ratkaisuja ja niiden toimintatapoja. Kehitystä helpottavia ja toiminnallisuutta lisääviä kirjastoja tutkitaan ja lisätään sovellukseen tarvittavissa vaiheissa. Sovelluksessa pyritään noudattamaan hyvää koodauskäytäntöä ja tuomaan esille JavaScriptin uudempia ominaisuuksia, kuten esimerkiksi async/await -syntaksia.

2 TEKNOLOGIAT

2.1 JavaScript

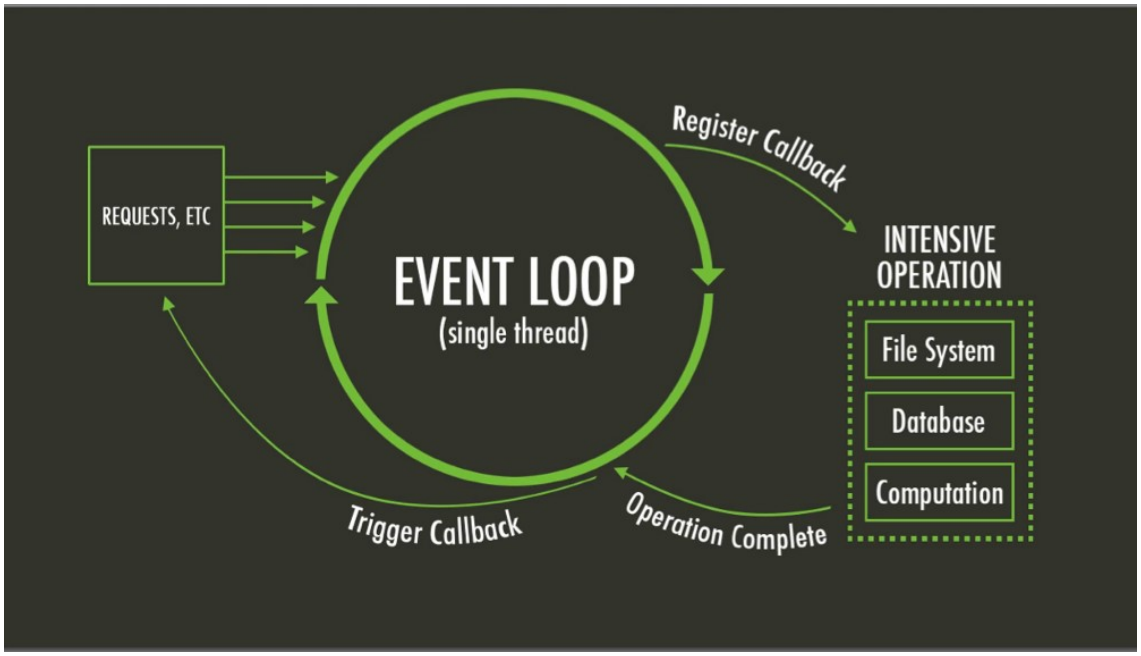
Javascript on Brendan Eichin vuonna 1995 kehittämä dynaaminen komentosarjakieli. Javascript on standardoitu ECMA-262 standardin mukaisesti 1997 lähtien ja kieltä kehitetään edelleen. Standardoidusta JavaScriptistä käytetään nimitystä ECMAScript. ECMAScriptin viimeisin, yhdestoista editio, joka virallisesti tunnetaan nimellä ECMAScript 2020, julkaistiin kesäkuussa 2020. [2]

JavaScriptistä on tullut webin de-facto standardi kieli sen ensimmäisen liikkeen edun, helpon integroinnin dokumenttioliomalliin ja dynaamisen luonteen ansiosta. JavaScriptiä käytetään nykyisin myös webin ulkopuolella muun muassa pelien, mobiili- ja työpöytäsovellusten kehittämiseen. Stack Overflown vuosittaisessa kehittäjäkyselyssä 2020 JavaScript oli kahdeksatta vuotta peräkkäin ammattilaisten eniten käyttämä ohjelmointikieli. [3]

2.2 Node.js

Node.js on JavaScript-ajoympäristö, joka perustuu Google Chromen V8 -moottoriin ja sen on kehittänyt Ryan Dahl vuonna 2009. Node.js mahdollistaa JavaScript-koodin suorittamisen palvelimella, jonka ansiosta ohjelmistokokonaisuuksia pystytään kirjoittamaan yksinomaan JavaScriptiä käyttäen. Asynkronisena tapahtumapohjaiseen arkkitehtuuriin pohjautuvana ajoympäristönä Node.js on suunniteltu skaalautuvien verkkosovellusten rakentamiseen. [4]

Node.js-tapahtumasilmukka on yksisäikeinen. Toisin kuin useat muut kielet, jotka vaativat uuden säikeen tai prosessin jokaiselle pyynnölle Node.js ottaa kaikki pyynnöt ja delegoi työn järjestelmän sisällä libuv -kirjaston avulla. Kun työ on tehty taustalla, Node.js laukaisee rekisteröidyt callback-funktiot automaattisesti ja jatkaa operaatioita. Tapahtumasilmukan toimintaperiaate on havainnollistettuna kuvassa 1. On toivottavaa, että suorittimelle intensiivisiä toimintoja vältetään, koska ne saattavat ylikuormittaa ainoan säikeen. [5]



Kuva 1. Node.js-tapahtumasilmukka. [6]

Node Package Manager

Node Package Manager eli npm toimii Node.js:n oletettuna paketinhallintahallintajärjestelmänä. Npm rekisteristä löytyy useita moduuleja, jotka helpottavat sovellusten kehittämistä. Npm:n toiminnot kuten moduulien asennus tapahtuvat komentoriviltä.

2.3 Express.js

TJ Holowaychukin vuonna 2010 esittelemä Express.js on minimaalinen ja joustava Node.js-verkkosovelluskehys, joka tarjoaa vankan ominaisuusjoukon verkko-, mobiilisovellusten ja ohjelmointirajapintojen luontiin. [7]

Expressin avulla voidaan hallinnoida palvelinpuolen reititystä, istuntoja, http-pyyntöjä ja virhekäsittelyä. Yksinkertainen ja nopea kustomointi, konfigurointi ja liitettävyys tietokantoihin ovat Express.js:n vahvuuksia. Vaikka kaikki koodi voitaisiin kirjoittaa myös JavaScriptiä käyttäen, Express.js:n avulla selvitään vähemmällä määrällä koodia ja siten nopeutetaan kehitysaikaa.

2.4 Reactjs

React on JavaScript-kirjasto, jota ylläpitää Facebook. React on suunniteltu käyttöliittymien kehittämiseen, ja on komponenttipohjainen. Komponentit kuvaavat tiettyä osaa sovelluksen käyttöliittymässä. Komponenttien sisällä saattaa olla datana joku propseja tai state eli tila. Komponentit, joiden avulla tilaa säädetään, kutsutaan statefull tai tilallisiksi komponenteiksi, kun taas tilattomista käytetään yleisimmin nimitystä stateless. Reactia voidaan kirjoittaa JSX-ohjelmointikielellä. JSX muistuttaa vahvasti HTML:ää, mutta se tulee kaikilla JavaScriptin mahdollisuuksilla. [8]

Reactin virtuaalinen asiakirjaobjektimalli tekee siitä erityisen tehokkaan työkalun yhden sivun sovellusten kehittämiseen. React ei päivitä muutoksia suoraan asiakirjaobjektimalliinsa, vaan sen sijasta tilamuutosten tapahtuessa muutoksia verrataan virtuaalisessa asiakirjaobjektimallissa. Kun uusi komponentti luodaan tai komponentissa tapahtuu tilamuutos, luodaan myös uusi virtuaalinen asiakirjaobjektimalli. React sen jälkeen vertaa uutta virtuaalista asiakirjaobjektimallia edelliseen ja päivittää näkymässä vain komponentit, jotka ovat muuttuneet koko näkymän päivittämisen sijasta. Tällä operointitavalla saavutetaan erittäin hyvä suorituskyky etenkin yhden sivun sovelluksissa. [9]

React Hooks

React koukut eli hookit esiteltiin Reactin versiossa 16.8 ja ne antoivat kehittäjille mahdollisuuden käyttää tiloja ja muita Reactin ominaisuuksia luomatta luokkia. Hookit kehitettiin tekemään tilojen ja logiikan uudelleenkäytöstä helpompaa sekä parantamaan suurien komponenttien ymmärrettävyyttä.[10]

Reactin sisäänrakennetut hookit on lajiteltu Basic ja Additional ryhmiin.

Basic Hooks:

-useState

-useEffect

-useContext.

Additional Hooks:

-useReducer

-useCallback

-useMemo

-useRef

-useImperativeHandle

-useLayoutEffect

-useDebugValue.

Basic -ryhmässä ovat yleisemmin käytössä olevat yksinkertaisemman hookit, kun taas Additional ryhmän hookit ovat joko muunnoksia Basic ryhmän hookeista tai niitä tarvitaan vain harvinaisemmissa rajatapauksissa. [11]

Esimerkissä (kuva 2) luodaan useState hookin avulla tilamuuttuja count ja funktio setCount sen päivittämiseen. Tilamuuttujan arvoa voidaan kasvattaa painamalla button elementtiä, joka laukaisee setCount funktion. Arvo näkyy p elementissä. Komponentin päivittyessä useEffect hookin avulla voidaan luoda sivuvaikutuksia, jotka suoritetaan komponentin jokaisen uudelleen renderöinnin jälkeen tai vain haluttujen arvojen muuttuessa, jotka asetetaan vapaaehtoisena argumenttina asetettavaan taulukkoon. Taulukon voi myös jättää tyhjäksi, jolloin efekti suoritetaan vain kun komponentti luodaan.

Esimerkissä efektin avulla päivitetään dokumentin otsikko, ja se suoritetaan vain count tilamuuttujan arvon vaihtuessa.

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  },[count]);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Kuva 2. Esimerkki hooks.

Vaikka hookit teknisesti ottaen ovat vain JavaScriptfunktioita, on niille asetettu kaksi sääntöä. Hookkeja kutsutaan vain React-funktioista, sekä hookkeja kutsutaan vain ylimmältä tasolta. Hookkeja ei siten saada kutsuta esimerkiksi silmukoiden, tai ehdollisten if-lauseiden sisällä. [12] Kehittäjät voivat myös itse luoda hookkeja.

React Router

React Router pitää käyttöliittymän synkronoituna URL-osoitteen kanssa. Siinä on yksinkertainen rajapinta, jolla on tehokkaat ominaisuudet, kuten laiska koodin lataus, dynaaminen reitin sovitus ja sijaintimuutosten hallinta. [13] React Routerin uusimpiin

ominaisuuksiin kuuluvat hookit. Esimerkiksi useParamns hookki, joka mahdollistaa reittitietoihin pääsyn mistä tahansa reitin alapuolelta.

2.5 MongoDB

MongoDB on dokumenttipohjainen NoSQL-tietokanta. Toisin kuin SQL-tietokannoissa, joissa data tallennetaan tauluihin ja riveihin, MongoDB:ssä tallennus tapahtuu JSON-objekteja muistuttaviin dokumentteihin ja kokoelmiin. Dokumentit käyttävät BSON varianttia, joka mahdollistaa suuremman määrä datatyyppejä perinteiseen JSON objektiin verrattuna. MongoDB ei vaadi esimääritettyjä skeemoja ja tallentaa kaikentyyppistä dataa. [14] Tämä antaa kehittäjille mahdollisuuden luoda haluamansa määrän kenttiä dokumenttiin, tehden siten MongoDB-tietokannan skaalauksesta helpompaa kuin relaatiotietokannoissa

Mongoose

Mongoose on objektidatan mallintamiskirjasto, joka tarjoaa mallinnusympäristön datalle ja varmistaa rakenteen tarpeen mukaan, säilyttäen kuitenkin joustavuuden. Datan mallintaminen tapahtuu skeemoilla. Skeema määrittelee muun muassa asiakirjan rakenteen oletusarvot ja validoijat. Mallia voidaan ajatella kääreenä skeemalle. Malli tarjoaa käyttöliittymän tietokantaan luomista, päivittämistä, hakemista ja poistamista varten. [15]

2.6 Muut

2.6.1 Bcryptjs

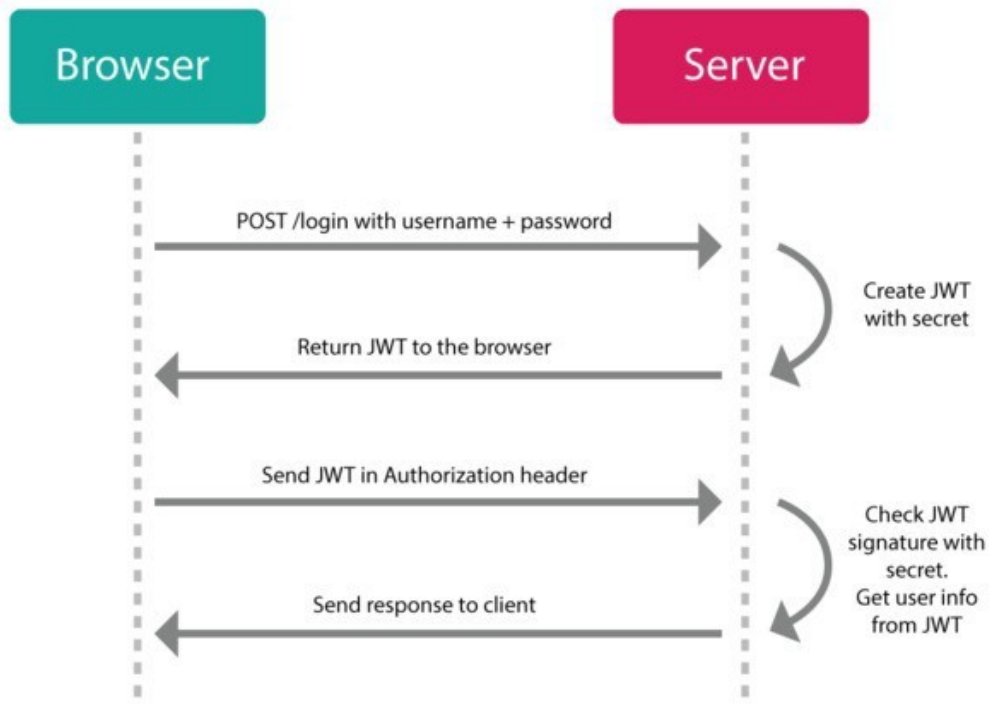
Bcryptjs on kirjasto Npm-rekisterissä, jolla voidaan hajauttaa ja verrata salasanoja Node.js-ympäristössä. Bcryptin yleisin käyttötarkoitus onkin salasanojen suolattu hajauttaminen (salted hashing). Luomalla satunnaisia tavuja (suolaa) ja yhdistämällä sen salasanaan ennen hajautusta, voidaan luoda täysin uniikki hajautettu salasana. Vaikka

useammalla käyttäjällä olisi sama salasana, on hajautettu tietokantaan tallennettu salasana silti erilainen jokaisessa tapauksessa. Tämän avulla pystytään eliminoimaan niin sanotut ”rainbow table” hyökkäykset, missä suolaamattomia hajautettuja salasanoja muutetaan takaisin ihmiselle luettavaan muotoon tunnettujen hajautusalgoritmien avulla.[16]

2.6.2 JSON Web Token

JSON Web Token (JWT) on avoin standardi (RFC 7519) tietojen turvalliseen lähettämiseen osapuolten välillä JSON-objektina. JWT: n käytön tarkoituksena ei ole tietojen piilottaminen, vaan tietojen aitouden varmistaminen. JSON Web Token koostuu kolmesta eri osasta: otsikko(header), sisältö(payload) ja allekirjoitus(signature). Otsikko koostuu token tyypistä ja algoritmista. Esimerkkinä tyyppinä voisi olla JWT ja algoritmina HS256. Sisältö osio koostuu väitteistä(claims). Kehittäjä voi luoda omia kustomoituja väitteitä. Allekirjoitus lasketaan koodaamalla otsikko ja sisältö käyttäen Base64url-koodausta ja yhdistämällä ne, erottaen ne kuitenkin toisistaan pisteellä. Mikäli otsikko tai sisältö muuttuu, allekirjoitus on laskettava uudelleen. [17]

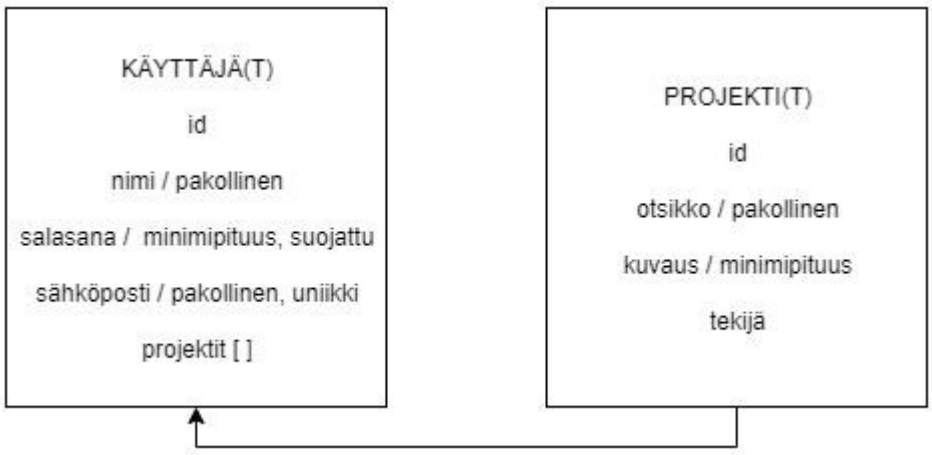
Token luodaan yleisimmin sisäänkirjautumisen ja rekisteröinnin yhteydessä. Tokenia voidaan liikuttaa osana pyyntöjä Authorization-otsikkokentässä, kun autentikointia vaaditaan. Tämä toimintaperiaate on havainnollistettuna kuvassa 3.



Kuva 3. JSON Web Token toiminta havainnollistettuna. [18]

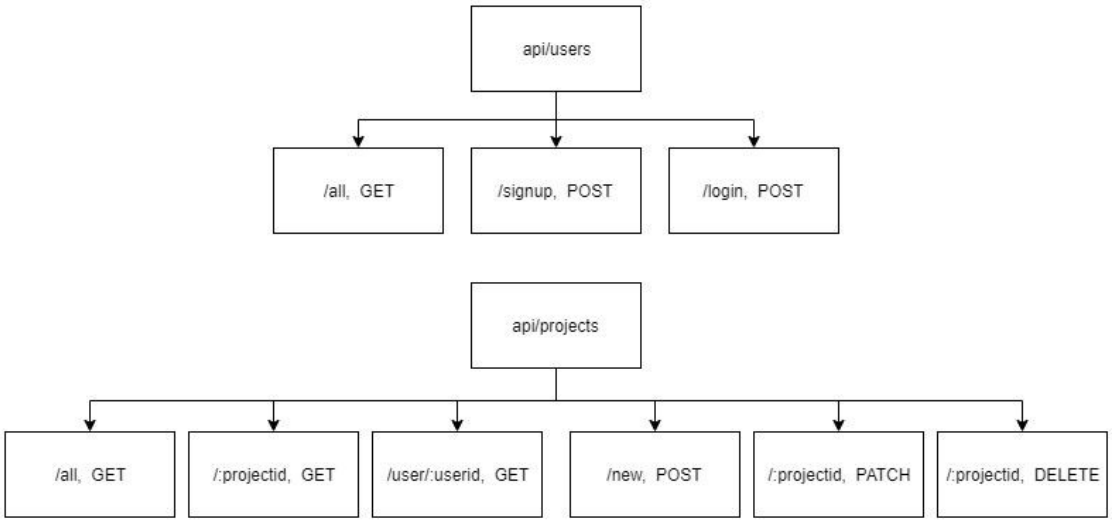
3 SUUNNITTELU

Prototyyppisovelluksen perusideana on luoda web-sovellus, jossa käyttäjät voivat jakaa ideoita projekteista toisilleen. Sovelluksen data malleina toimivat käyttäjät sekä projektit, jotka ovat yhteydessä toisiinsa (kuva 4). Käyttäjä voi luoda rajoittamattoman määrän projekteja, mutta jokainen projekti voi kuulua vain yhdelle käyttäjälle. Käyttäjille datana toimivat MongoDB-id, nimi, salasana, sähköpostiosoite ja taulukko luotujen projektien id-arvoille. Nimi asetetaan pakolliseksi. Salasanalle asetetaan minimipituus ja tallennettaessa tietokantaan se hajautetaan bcryptjs:n avulla. Sähköpostiosoite asetetaan pakolliseksi ja sen on oltava uniikki. Projekti malli sisältää MongoDB-id:n, otsikon, kuvauksen ja tekijäarvon, jolla viitataan projektin luoneen käyttäjän id-arvoon. Otsikko asetetaan pakolliseksi ja kuvaukselle annetaan minimipituus.



Kuva 4. Mallit käyttäjille ja projekteille.

Palvelinpuolella luodaan rajapinta, joka kykenee vastaanottamaan, lähettämään ja käsittelemään tietoa tietokannan ja käyttöliittymän välillä. Tietokantana käytetään pilvipalveluna toimivaa MongoDB Atlas -palvelua. Päätepisteisiin luodaan reititys ja tietojen käsittelyyn tarvittava logiikka. Päätepisteitä luodaan yhteensä 9 kappaletta ja ne ovat havainnollistettuna kuvassa 5. Projektien luomiseen, päivittämiseen ja poistamiseen käytettävät reitit sallitaan vain rekisteröityneille käyttäjille. Tämä toiminta saavutetaan JSON Web Tokeneja hyödyntäen. Projekteja poistettaessa ja päivittäessä varmistetaan myös, että käyttäjän id vastaa projektimallissa määritettyä tekijäarvoa.



Kuva 5. Rajapintasuunnitelma havainnollistettuna.

Käyttöliittymä koostuu selaimen yläreunassa sijaitsevasta navigaatioelementistä ja sen alle sijoitettavasta sisällöstä. Selaimen näkymä vastaa sivun URL-osoitetta. React komponentit liitetään eri reitteihin React Routerin avulla. Navigaatio mukautuu määritettävien tilojen perustella. Sisäänkirjautuneille käyttäjille lisätään navigaatioelementtiin painikkeet uloskirjautumiseen ja projektien luontiin. Samanaikaisesti sisäänkirjautuminen sekä rekisteröinti poistetaan näkymästä. Käyttäjän katsoessa omia projektejaan, lisätään projektinäkymään painikkeet projektin päivittämiseen ja poistamiseen. Rekisteröityessä sekä sisäänkirjautuessa, lähetetään palvelimelta token, joka varastoidaan selaimen paikallisessa tallennustilassa. Kun käyttäjä kirjautuu ulos tai vaihtoehtoisesti token-voimassaoloaika täyttyy, poistetaan käyttäjän data paikallisesta tallennustilasta.

Sovellus kehitetään Windows 10 -käyttöjärjestelmällä. Tekstieditoriksi valikoitui Microsoftin Visual Studio Code.

4 BACKEND

4.1 Perusrakenne

Palvelinpuolen ensimmäisenä tehtävänä alustetaan ympäristö komennolla ”npm init”. Komennon jälkeen näkyviin ilmestyy package.json tiedosto, josta voidaan hallita npm-lisäosia, luoda komentosarjoja ja tarkastella projektin metadataa. Seuraavaksi asennetaan Express.js -kehys, body-parser -väliohjelmisto ja Mongoose -mallintamiskirjasto. Varsinaiset reitit voidaan nyt alustaa aikaisemmin tehdyn suunnitelman kaltaisiksi. Jotta tiedostot ovat kehityksen ajan helposti löydettävissä, jaetaan reitit kahteen eri osioon, projekti- ja käyttäjäreitteihin. Kummallekin luodaan omat tiedostonsa, joissa määritetään reittien osoite ja suoritettava funktio. Kaikki funktiot määritetään async/await -syntaksin mukaan, koska kyselyitä tai muita tietokantaoperaatioita esiintyy niissä jokaisessa. Kyselyt ja muut operaatiot ovat joko thenable-funktioita, joissa määritetään ”then” metodi, tai ne palauttavat lupauksen(promise). Kun kyselyiden eteen liitetään await-avainsana, pysäytetään koodin suorittaminen kyseiselle riville, kunnes lupaus täyttyy. Sillä välin palvelimelle annetaan mahdollisuus suorittaa muuta koodia ja siten parannetaan suorituskykyä. [19]

4.2 Tietokanta

MongoDB-tietokannan voi luoda itse lokaalisti, tai ottaa käyttöön pilvipalveluna. Tässä projektissa käyttöön valittiin pilvitietokantapalvelu MongoDB Atlas. Palvelun käyttö vaatii rekisteröitymisen sivuille mongodb.com. Kun rekisteröinti on suoritettu, clusteri on luotavissa. Projektiin valittiin ilmaisversio, joka mahdollistaa määrällisesti 512mb tallennuksen. Palveluntarjoajaksi valittiin AWS ja sijainniksi alun perin Frankfurt. Sijainti vaihtui myöhemmin Pohjois-Virginiaan yhteysongelmien vuoksi. Clusteri voidaan myös nimetä tässä vaiheessa. Clusterin luomisen jälkeen luodaan yhteys tietokantaan. Network Access-valikosta mahdollistetaan yhteyden luominen nykyiseen IP-osoitteeseen ”whitelistaamalla” osoite. Jotta tietokantaa voidaan käyttää, on luotava käyttäjätunnus halutuilla

oikeuksilla. Käyttäjän autentikointi tavaksi valittiin salasana. Luodulle käyttäjälle annetaan oikeudet lukea ja luoda dataa. Clusterin connection-valikosta löydetään tavat, joilla tietokanta voidaan yhdistää sovellukseen. Tässä projektissa käyttöön valittiin suora yhteys. Suoran yhteyden muodostaminen tapahtuu connection stringin avulla, joka lisätään suoraan sovellukseen. Ennen stringin luovuttamista on valittava ajurit ja niiden version, jotka tässä projektissa ovat nodejs ja versio 3.6 tai myöhäisempi. Connection string voidaan nyt syöttää kuvassa 6 esiintyvällä tavalla mongoosen connect-metodin argumentiksi.

```
mongoose
.connect(
  'mongodb+srv://<username>:<password>@tjlthesis-jsg8s.mongodb.net/<dbname>?retryWrites=true&w=majority'
)
.then(() => {
  app.listen(5000);
})
.catch((error) => {
  console.log(error);
});
```

Kuva 6. MongoDB Atlas connection string connect-metodin argumenttina.

Muutettavia tietoja ovat aiemmin luodun käyttäjän käyttäjänimi, salasana ja omavalintainen tietokannan nimi. Tietojen muuttamisen jälkeen yhteys tietokantaan on luotu.

Tietokantaan tallennettavalle datalle luodaan mallit. Malleille luodaan erillinen alikansio. Ensimmäisenä skeema määritetään Schema-luokan avulla. Tämän jälkeen olio muotoon määritetään avaimet, joina toimivat kenttien nimet, sekä datatyyppi jokaiselle kentälle. Käyttäjäskeemaan (kuva 7) määritettiin 4 kenttää, joista kaikki ovat pakollisia. Sähköpostin on oltava ainoa laatuaan tietokannassa ja sille lisätään sen vuoksi myös unique-arvo. Kolmen kentän datatyyppinä toimii merkkijono, kun taas projektit ovat taulukossa, jossa referoidaan toisen mallin määrittämiin objekteihin. Salasanoille on myös määritetty minimipituus, joka on 5 merkkiä. Käytössä on myös mongoose-unique-validator -laajennus. Laajennuksella pystytään lisäämään ennen tietokantaan tallennusta tapahtuva validointi varmistamaan, että mikäli tietokannasta löytyy samalla arvolla varustettu kenttä, saadaan takaisin käyttäjäystävällisempi virheviesti kuin oletus E11000 virhe. Täten virheiden paikantaminen helpottuu. Viimeiseksi luodaan malli tehdystä skeemasta model funktion avulla. Funktiossa määritetään mallin nimi ja sekä skeema, jonka mukaan malli luodaan. Mallin nimi määrittää tietokannassa olevan kokoelman oletusnimen monikkomuodossa.

```
const mongoose = require('mongoose');
const monUniqValidator = require('mongoose-unique-validator');

const Schema = mongoose.Schema;

const userSchema = new Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true, minlength: 5 },
  projects: [{ type: mongoose.Types.ObjectId, required: true, ref: 'Project' }],
});

userSchema.plugin(monUniqValidator);

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

Kuva 7. Käyttäjän tiedoille luotu Mongoose-malli.

Projekteille luotavassa skeemassa määritetään 3 kenttää: otsikko, kuvaus ja tekijä. Otsikko ja kuvaus ovat merkkijonoja ja tekijällä referoidaan käyttäjä objektin id:hen. Referoimalla luodaan yhteys mallien välille.

4.3 Virheenkäsittely ja validointi

Expressissä on sisäänrakennettu virhekäsittelijä, joka huolehtii sovelluksessa mahdollisesti esiintyvistä virheistä. Tämä oletusarvoinen virheenkäsittelyväliohjelmistotoiminto lisätään väliohjelmistotoiminnasta vastaavan pinon loppuun. [20] Virheenkäsittelyväliohjelmisto on tunnistettavissa sen ensimmäisestä argumentista err/error, kokonaisuudessaan argumentteja on 4. Virhekäsittelijässä ensimmäisenä tarkistetaan, onko vastaus jo pystytty lähettämään, vaikka virhe on tapahtunut. Mikäli näin on käynyt, palautetaan virhe, jotta toista vastausta ei lähetetä. Tapaus, jossa vastausta ei ole lähetetty, lähetetään virhekoodi ja viesti. Molemmat voidaan asettaa erikseen kehittäjän toimesta. Jos kehittäjä ei ole erikseen määrittänyt virhekoodia ja viestiä turvaudutaan oletusarvoihin.

Express-reitit pystyvät ottamaan vastaan useita callback-funktioita, jotka käyttäytyvät kuten väliohjelmat. Tämän mekanismin avulla voidaan asettaa ennakkoehtoja reiteille ja siirtää siten ohjaus seuraaville reiteille, jos ei ole mitään syytä jatkaa nykyistä reittiä. [21] Tätä ominaisuutta hyväksikäyttäen otetaan käyttöön express-validator -kirjasto. Express-validatorin avulla voidaan liittää reitteihin ennakkoehtoja. Kuvassa 8 käyttäen

check-metodia tarkastetaan /signup reitin bodysta, että nimi, sähköposti ja salasana kentät ovat kriteerien mukaiset.

```
router.post(
  '/signup',
  [
    eValidator.check('name').not().isEmpty(),
    eValidator.check('email').normalizeEmail().isEmail(),
    eValidator.check('password').isLength({ min: 5 }),
  ],
  usersLogic.signup
);
```

Kuva 8. Express-validator tarkastus.

Ennakkoehtoja asetetaan myös projektin luomiseen ja päivittämiseen käytettävissä reiteissä.

4.4 Logiikan määrittely

Reittien toiminnallisuuden kehittäminen aloitetaan, koska alustavat vaiheet ovat suoritettu. Projektit tulevat olemaan osana käyttäjän projektitaulukkoa, joten on loogista aloittaa käyttäjän luomisesta. Käyttäjän luomisesta vastaava funktio toimii seuraavasti.

Pyynnölle annetut ennakkoehdot tarkastetaan, mikäli data ei ole kriteerien mukaista suorittamien pysäytetään. Pyynnöstä saatavat arvot, nimi, sähköposti ja salasana puretaan niiden käytön helpottamiseksi. Muuttuja emailTaken luodaan try/catch blokin ulkopuolella, jotta se on myös myöhemmin käytettävissä. Tietokantaan suoritetaan kysely findOne-metodilla, jossa etsitään pyynnöstä saatua sähköpostiosoitetta jo olemassa olevilta käyttäjiltä. Kyselyn eteen liitetään myös avainsana await, koska kyselyn tuloksen saaminen saattaa kestää ja tulosta käytetään myös seuraavassa vaiheessa. Näin varmistetaan, että koodia ei suoriteta eteenpäin ennen vastauksen saamista. Kysely voi epäonnistua, ja siten se kiedotaan try/catch blokin sisälle. Epäonnistuessaan suorittaminen voidaan pysäyttää ja päästään palauttamaan virhe, jonka avulla vika on helpompi

paikantaa. Kun kyselyn tulos on saatu, varmistetaan if-lauseella, ettei toista sähköposti-osoitetta löytynyt. Toteutus on havainnollistettuna kuvassa 9.

```
const signup = async (req, res, next) => {
  const errors = eValidator.validationResult(req);
  if (!errors.isEmpty()) {
    const error = new Error('Invalid inputs');
    error.code = 422;
    return next(error);
  }
  const { name, email, password } = req.body;

  let emailTaken;
  try {
    emailTaken = await User.findOne({ email: email });
  } catch (err) {
    const error = new Error('Fetching email failed');
    error.code = 500;
    return next(error);
  }

  if (emailTaken) {
    const error = new Error('Email not available');
    error.code = 422;
    return next(error);
  }
```

Kuva 9. Datan kriteerien ja sähköpostiosoitteen tarkastus rekisteröityessä.

Salasanan hajautus suoritetaan bcryptjs -kirjaston avulla. Muuttuja luodaan blokin ulkopuolella, ja sille annetaan nimi passwordHashed. Salasanan hajautus saattaa epäonnistua, joten se kiedotaan try/catch-blokkiin. Bcyptin hash-funktio palauttaa lupauksen ja siksi on suotavaa käyttää avainsanaa await sen yhteydessä. Funktio ottaa argumenteikseen hajautettavan salasanan ja suolauskierrosten määrän. Funktion tulos asetetaan aikaisemmin luotuun passwordHashed muuttujaan. Saatu data voidaan nyt liittää uuteen käyttäjämalliin, jossa nimi ja sähköposti vastaavat pyynnöstä saatuja arvoja, salasanaksi

asetetaan hajautettu salasana ja projekti taulukko alustetaan tyhjänä. Malli tallennetaan mongoosen save-metodin avulla tietokantaan. Tietokantaoperaationa vaihe kiedotaan try/catch-blokkiin ja operaation tulosta odotetaan. Operaatio voi epäonnistua esimerkiksi koodista riippumattomien yhteysongelmien vuoksi. Toteutus on havainnollistettuna kuvassa 10.

```
let passwordHashed;
try {
  passwordHashed = await bcrypt.hash(password, 10);
} catch (err) {
  const error = new Error('Password hashing failed');
  error.code = 500;
  return next(error);
}

const createdUser = new User({
  name,
  email,
  password: passwordHashed,
  projects: [],
});

try {
  await createdUser.save();
} catch (err) {
  const error = new Error('Saving user failed');
  error.code = 500;
  return next(error);
}
```

Kuva 10. Salasanan hajautus ja käyttäjän tietojen tallentaminen tietokantaan rekisteröityessä.

Sign-metodin avulla luodaan token, joka sisältää digitaalisesti allekirjoitetussa muodossa luodun käyttäjän id:n. Token on digitaalisesti allekirjoitettu käyttämällä salaisuutena olevaa ”secretprivatekey” merkkijonoa. Allekirjoitus varmistaa sen, että ainoastaan

salaisuuden tuntevilla on mahdollisuus generoida validi token. Kolmantena argumenttina metodiin liitetään vaihtoehtona vanhenemisaika, joka asetetaan kestoltaan kahteen tuntiin. Jos kaikki vaiheet pystytään suorittamaan onnistuneesti, palautetaan statuskoodina 201 ja objekti, jonka sisältönä on käyttäjän id ja luotu token. Toteutus on havainnollistettuna kuvassa 11.

```
let token;

try {
  token = jwtoken.sign({ userId: createdUser.id }, 'secretprivatekey', {
    expiresIn: '2h',
  });
} catch (err) {
  const error = new Error('Token creation failed');
  error.code = 500;
  return next(error);
}

res.status(201).json({ userId: createdUser.id, token: token });
};
```

Kuva 11. Tokenin luominen rekisteröityessä.

Projektin luominen tietokantaan on hyvin samankaltainen operaatio kuin käyttäjän luominen. Suurin ero tapahtuu, kun data tallennetaan tietokantaan, koska samanaikaisesti muokataan jo olemassa olevaa dokumenttia. Tapahtumassa on varmistettava luotavan projektin tallentuminen tietokantaan sekä sen lisääminen käyttäjän projektitaulukkoon. Tätä varten käyttöön otetaan Mongoosen transactions-toiminto, jonka toimita on havainnollistettu kuvassa 12. Toiminnon avulla voidaan suorittaa useita operaatioita session aikana ja varmistaa, että vain kaikkien operaatioiden onnistuessa muutokset tallennetaan.

```
try {
  const sessio = await mongoose.startSession();
  sessio.startTransaction();
  await createdProject.save({ session: sessio });

  user.projects.push(createdProject);
  await user.save({ session: sessio });
  await sessio.commitTransaction();
} catch (err) {
  const error = new Error('Saving failed');
  error.code = 500;
  return next(error);
}
```

Kuva 12. Mongoose transactions-sessio projektia luotaessa.

Käyttäjiä ja projekteja haettaessa GET-pyynnöillä suoritetaan tietokantaan kysely find-metodin avulla. Mallien perusteella suoritettava kysely palauttaa kummastakin kyselystä kaikki dokumentit, jotka täyttävät mallien kriteerit. Find-metodille asetetaan käyttäjiä haettaessa toisena argumenttina kielto palauttaa salasana. Kun projekteja halutaan päivittää tai poistaa, on projekti pystyttävä tunnistamaan. Reitin osoitteeseen lisätäänkin dynaaminen segmentti, joka tapahtuu kaksoispisteellä ja tunnisteella. Tunnistetietona käytetään projektien id:tä. Id saadaan näin req.params.projectid objektista ja sitä voidaan käyttää tietokantaoperaatioissa.

4.5 Todennus ja CORS

Cross-origin resource sharing (CORS) on mekanismi, jonka avulla verkkosivun rajoitettuja resursseja voidaan pyytää toisesta verkkotunnuksesta sen verkkotunnuksen ulkopuolella, josta ensimmäinen resurssi on annettu. Jotta vältytään CORS-virheiltä palvelinpuolen ja selaimen kommunikoinnin aikana, liitetään www.enable-cors.org sivuilta saatava ohjelmistokoodi (kuva 13) sovellukseen. Kehityksen ajaksi kaikki

verkkotunnukset pystyvät tekemään pyyntöjä ja HTTP-metodit GET, POST, PATCH ja DELETE sallitaan. Sallittujen otsikkotietojen listalle lisätään myös Authorization-otsikko, joka seuraa suositeltua nimeämistapaa, kun tokenia liikutetaan otsikkotiedoissa.[22]

```
app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader(
    'Access-Control-Allow-Headers',
    'Origin, X-Requested-With, Content-Type, Accept, Authorization'
  );
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PATCH, DELETE');
  next();
});
```

Kuva 13. CORS-asetukset.

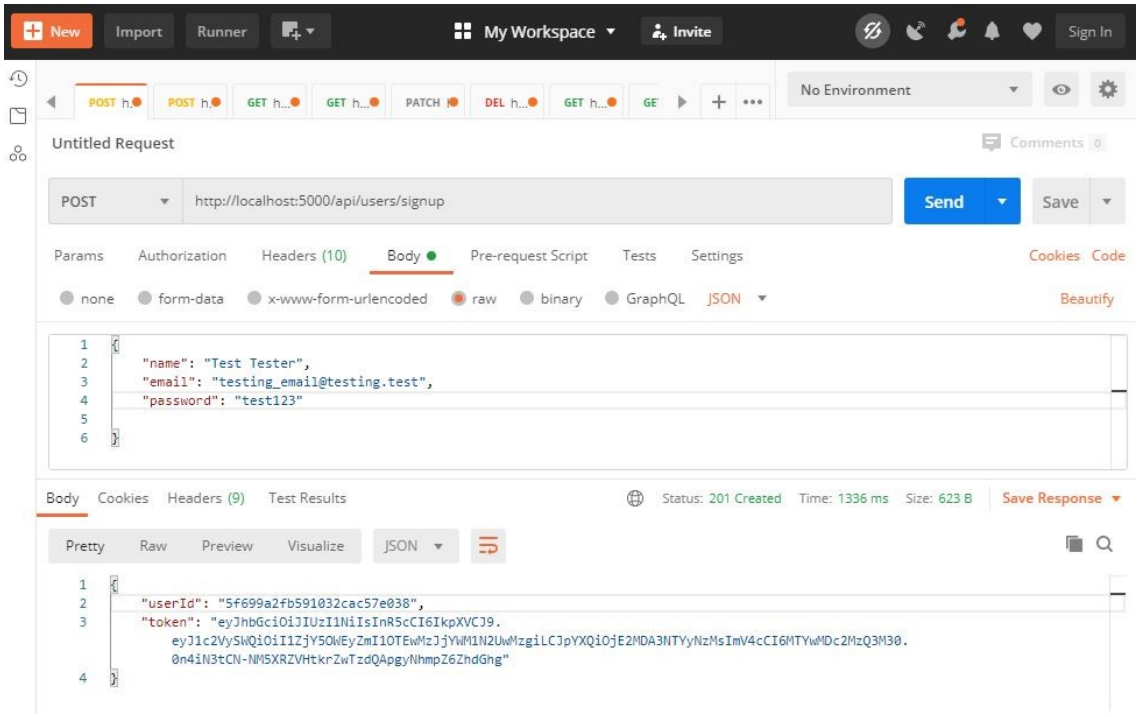
Authorization-otsikossa eteen suositellaan sanaa Bearer indikoimaan, että käyttäjä ”kantaa” tokenia, mutta koska sovelluksessa ei liikuteta muuta dataa Authorization-otsikossa, ei Bearer sanaa liitetä.

Projektien luomiseen, päivittämiseen ja poistamiseen käytettävät reitit ovat käytettävissä vain käyttäjillä, joiden pyyntöön on liitetty voimassa oleva token. Reitit sijoitetaan pinossa alimmaisiksi ja näitä ennen suoritettavaksi asetetaan väliohjelmisto, joka tarkastaa tokenin olemassaolon ja aitouden. Väliohjelmisto luodaan itse. Väliohjelmistossa varmistetaan, että token on oikeellinen metodilla verify. Tokenin ollessa pätemätön, pysäytetään suorittaminen, josta siirrytään catch-blokkiin ja kirjataan virhe. Selaimien käyttäytymiseen liittyen, sijoitetaan väliohjelmistoon myös tarkistus tulevan pyynnön metodille. Selaimet lähettävät automaattisesti OPTIONS-pyynnöt kaikille ei GET-tyyppisille pyynnöille varmistaakseen, että ne ovat sallittuja. Jotta CORS-virheiltä vältytään, OPTIONS-pyynnöt sallitaan.

4.6 Testaaminen

Reittien toimivuutta testataan kehityksen aikana Postman ohjelmiston avulla, jolla voidaan suorittaa kutsuja rajapintaan ilman erillistä käyttöliittymää. Kuvassa 14 rekisteröitymiseen käytettävään reittiin, /signup, lähetetään pyyntö metodilla POST, johon on liitetty objekti, mikä sisältää nimen, sähköpostin ja salasanan. Pyynnön otsikoissa sen tyypiksi

asetetaan ”application/json”. Vastauksena oletetaan statusta 201 ja objektia, joka sisältää luodun käyttäjän id-arvon sekä käyttäjälle luodun tokenin.



Kuva 14. Rekisteröinnin testaus Postmanilla.

Pyynnön lähettämisen jälkeen tietokanta tarkastetaan ja varmistetaan, että käyttäjän data vastaa oletettuja arvoja. Kuvassa 15 on aikaisemman pyynnön seurauksena tietokantaan tallentunut data.

```
_id: ObjectId("5f699a2fb591032cac57e038")
projects: Array
name: "Test Tester"
email: "testing_email@testing.test"
password: "$2a$10$nPL.ls1qnuFUBiuNoHT/YeVs5liiNHQ20vazvuo1z7LZUtwRHGki."
__v: 0
```

Kuva 15. Tietokantaan tallennettu rekisteröityneen käyttäjän data.

Kun rajapinnan kaikki reitit toimivat halutulla tavalla, voidaan siirtyä käyttöliittymän kehittämiseen.

5 FRONTEND

Projektin käyttöliittymä luodaan Reactjs -kirjaston avulla. Create-React-App -pohjan avulla saadaan nopeasti luotua moderni perusrakenne React-sovellukselle ilman ylimääräistä konfigurointi. Paketin asennus tapahtuu komentoriviltä. Ennen varsinaista kehitysvaihetta voidaan nyt luodusta public-kansion index.html tiedostosta muuttaa sivun ”title” attribuutti haluttuun arvoon.

5.1 Deklaratiivinen reititys ja navigaatio

Käyttöliittymän kehittämisessä tullaan käyttämään react-router -kirjastoa, joka mahdollistaa deklaratiivisen reitityksen. Sovellus tulee pyörimään selaimessa, joten kirjasto asennetaan suorittamalla komento ”npm install –save react-router-dom”, jotta erityisesti selaimiin tarkoitettu versio asennetaan. App.js -tiedostoon tuodaan import-komenolla kirjastosta komponentit BrowserRouter, Route, Redirect ja Switch. Reitit jaetaan aikaisemman suunnitelman mukaisesti kahteen osaan kirjautumattomille ja kirjautuneille käyttäjille. Jako on havainnollistettuna kuvassa 16. Kirjautumattomille käyttäjille mahdollisia reittejä ovat projektien ja käyttäjien tarkastelu, sisäänkirjautuminen ja rekisteröinti. Esimerkiksi navigaatiosta linkkiä painamalla käyttäjä ohjataan näkymään, joka vastaa komponenttia joka reittiin on liitetty. Virheelliset osoitteet ohjataan sisäänkirjautumissivulle. Kirjautuneille käyttäjille tuodaan esiin myös projektien luonti- ja päivittämisreitit. Samanaikaisesti sisäänkirjautumisreitti poistetaan valikoimasta. Reittivaihtoehdot asetetaan if-lauseen sisällä, jossa tarkastetaan token-tilan oikeellisuus.

```
if (token) {
  validRoutes = (
    <Switch>
      <Route path='/users/all' exact>
        <Users />
      </Route>
      <Route path='/projects/all' exact>
        <Projects />
      </Route>
      <Route path='/:userId/projects' exact>
        <UserProjects />
      </Route>
      <Route path='/projects/new' exact>
        <NewProject />
      </Route>
      <Route path='/projects/:projectId'>
        <UpdateProject />
      </Route>
      <Redirect to='/projects/all' />
    </Switch>
  );
} else {
  validRoutes = (
    <Switch>
      <Route path='/users/all' exact>
        <Users />
      </Route>
      <Route path='/projects/all' exact>
        <Projects />
      </Route>
      <Route path='/:userId/projects' exact>
        <UserProjects />
      </Route>
      <Route path='/login'>
        <LogIn />
      </Route>
      <Redirect to='/login' />
    </Switch>
  );
}
```

Kuva 16. Reittijako.

Reitit asetetaan BrowserRouter-komponentin sisään yhdessä NavPrimary-komponentin kanssa, joka vastaa navigaatiosta. NavPrimary-komponentti koostuu header-elementistä, jonka sisällä sijaitsevat logoteksti, joka ohjaa käyttäjän kaikkien projektien näkymään ja navigaatiolinkit. Navigaatiolinkit toimivat erillisessä NavLinks-komponentissa, jossa määritetään näytettävät react-router linkkeinä toimivat painikkeet isLoggedIn muuttujan perusteella. Painikkeita ulkoisesti muistuttavat linkit ohjaavat käyttäjän haluttuun näkymään. Muuttujan isLoggedin ollessa tosi, lisätään uloskirjautumispainike, joka laukaisee logout-funktion. NavPrimary- ja Navlinks-komponetteihin lisätään CSS-määritteet käytettävyyden parantamiseksi.

5.2 Käyttäjäkomponentit

Käyttäjille ensimmäisenä luotava komponentti ”Users” (kuva 17) on vastuussa käyttäjätietojen saamisesta palvelimelta ja niiden jakamisesta muille komponenteille. Komponentissa alustetaan tila loadedUsers mikä vastaa palvelimelta saadun pyynnön dataa. Tila loadedUsers luodaan käyttämällä hookkia useState, johon tila ja sen muuttava funktio määritetään. Datan hakeminen suoritetaan useEffect-hookin avulla. Hookille määritetään kutsuttava funktio ja lista riippuvuuksia, joiden täyttyessä funktio suoritetaan uudestaan. Funktio halutaan suorittaa vain kerran, kun komponentti kiinnittyy ja täten riippuvuuslista jätetään tyhjäksi. Hookin palautusarvoksi ei suositella lupaus tyyppistä vastausta, ja koska async-funktiot palauttavat aina lupauksen, luodaan heti suoritettava funktio eli ”IIFE”. Funktiossa lähetetään pyyntö palvelimelle fetch-rajapinnan avulla. Fetch tarjoaa yleisen määritelmän pyyntö- ja vastausobjekteista, sekä muista verkkopyyntöihin liittyvistä asioista. [23] Saatu data asetetaan loadedUsers-tilan arvoksi. Koska pyyntö voi teoriassa epäonnistua se kiedotaan try/catch-blokkiin.

```
const Users = () => {
  const [loadedUsers, setLoadedUsers] = useState();

  useEffect(() => {
    const userRequestIife = async () => {
      try {
        const response = await fetch('http://localhost:5000/api/users/all');

        const responseData = await response.json();

        if (!response.ok) {
          throw new Error(responseData.message);
        }

        setLoadedUsers(responseData.users);
      } catch (err) {
        console.log(err);
      }
    };
    userRequestIife();
  }, []);

  return (
    <React.Fragment>
      {loadedUsers && <UserList items={loadedUsers} />};
    </React.Fragment>
  );
```

Kuva 17. Käyttäjätietojen hakeminen tietokannasta ja tuloksen määrittäminen tilaksi.

Pyynnöstä saatu data liitetään käyttäjälista komponentille proppina. Käyttäjälista komponentti ”UserList” tarkistaa onko proppina saatu lista pituudeltaan 0 ja renderöi tekstin, jossa ilmoitetaan, ettei käyttäjiä löytynyt, mikäli väite on tosi. Muissa tapauksissa proppeina saatu objektilista liitetään map-funktion avulla jonossa seuraavaan komponenttiin ”UserSingle”. Jokaiselle Usersingle-komponetille annetaan arvot key, id, name ja projects saaduista propseista. Usersingle-komponentissa renderöidään li-elementti, joka toimii linkkinä käyttäjän projekteihin id-arvon avulla. Elementissä esitetään myös käyttäjän nimi ja projektien lukumäärä.

5.3 Lomakkeet

Käyttäjän sisäänkirjautuminen ja rekisteröinti vaativat uusia komponentteja, joita voidaan myös myöhemmin käyttää projektien luomisessa ja päivittämisessä. Kaikki edellä mainitut käyttävät lomake eli form-elementtiä tietojen määrittämisen. Jotta sovellus ei nojaa täysin palvelinpuolen validointiin, luodaan Input-komponentti, joka tarkastaa käyttäjän syöttämän tiedot ja jonka tiloja voidaan käyttää sovelluksen lomakkeiden pätevyyden määrittämiseen kokonaisuudessaan. Input-komponentin lopullinen palautetta jakoelementti pitää sisällään yksirivisen syöttökenttään tai laajemman tekstialueen, otsikon ja mahdollisen paragrafin virhetekstille. Elementin tyyppi, virheteksti, otsikko ja mahdollinen validointi vaihtoehto määritetään propsien avulla komponenttia luodessa. Komponentissa otetaan käyttöön useReducer-hookki, johon määritetään argumentteina suoritettava reducer-funktio ja alkutila. Hookin palautusarvoina ovat tila ”inputState” ja dispatch-metodi. Tila on määritetty objektimuotoon, ja siihen määritetään kolme arvoa: value kuvaa elementtiin syötettyä arvoa, isValid validointitilan oikeellisuutta perustuen syötettyyn arvoon ja isTouched, joka kuvaa elementin koskemattomuutta. Reducer-funktiossa käytetään switch/case -rakennetta, jossa tapauksia on kolme, vaihto ”CHANGED”, kosketus ”TOUCHED” ja default. Kun elementtiin syötetyssä arvossa tapahtuu muutos, kutsutaan käsittelijäfunktio, joka laukaisee dispatch-metodin ja antaa sille tyypin ”CHANGED”. Metodi antaa myös elementin tämänhetkisen arvon ja mahdollisesti liitetyn validointikriteerin, esimerkiksi minimipituuden. Vaihto tapauksessa reducer-funktiossa aikaisempi tila kopioidaan kokonaisuudessaan, koska yksittäisiä arvoja muutettaessa objektimuotoisen tilan muut kentät nollautuisivat. Tämän jälkeen tilan arvo päivitetään vastaamaan dispatch-metodista saatua, kentässä muuttunutta arvoa. Viimeisenä arvo testataan erillisessä validointifunktiossa, mikäli arvolle on määritetty kriteerejä. Validointitilan oikeellisuus määritetään vastaamaan funktiosta saatua tulosta.

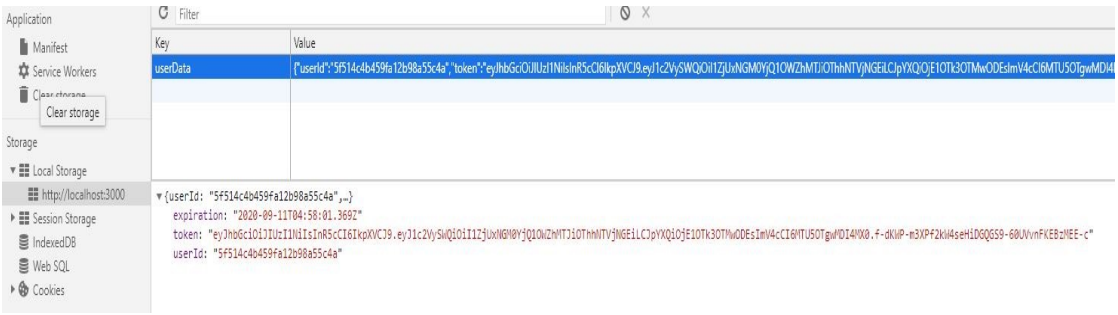
Jotta komponentti ei aloita virhetilasta, elementin koskemattomuutta määrittävä tila asetetaan alkutilassa valheelliseksi. Kun elementti menettää tarkkuustilan, suoritetaan käsittelijäfunktio ja laukaistaan dispatch-metodi, jonka tyypiksi on määritetty ”TOUCHED”. Tapauksessa aikaisempi tila kopioidaan ja ainut muutos tapahtuu koskemattomuutta määrittävään arvoon ”isTouched”, joka muutetaan oikeelliseksi. Näin käyttäjälle annetaan mahdollisuus antaa kriteerit täyttävä arvo ennen virhetilan aktivoitumista. Validointitilan ollessa virheellinen ja koskemattomuustilan ollessa oikeellinen liitetään elementtiin CSS-määritteitä. CSS-määriteet tekevät input-elementin reunoista ja rungosta punaiset

ja elementin alapuolelle lisätään paragrafi, jonka sisältönä näkyy ohjaava virheteksti. Input-komponentti vastaa vain yhtä kenttää, mutta koko lomakkeen kaikkien kenttien arvo ja validoinnin oikeellisuus kokonaisuudessaan pitää myös pystyä määrittämään. Ratkaisuksi tähän ongelmaan muodostuu kustomoitu hookki. Hookkien nimeämisessä suositellaan vahvasti aloittamaan sanalla use, täten kustomoidun hookin nimeksi annetaan ”useForm”. Hookille annetaan argumentteina alkuperäiset arvot input-elementeistä sekä lomakkeen alkuperäinen validointitila. Käyttämällä useReducer-hookkia kustomoidun hookin sisällä voidaan molemmat saadut argumentit sijoittaa useReducer-hookin määrittämän lomakkeen kokonaistilan alkuperäistilaksi. Kaksi palautettavaa arvoa ovat täten uusi päivitetty tai tarkastettu tila kokonaisuudessaan sekä dispatch-metodi. Dispatch-metodin laukaisevia käsittelijäfunktioita luodaan kaksi. Kumpikin funktioista kiedotaan riippuvuuksettomiin useCallback-hookkeihin. UseCallback-hookkeja käytetään, koska dispatch-funktioista toinen inputHandler-funktio halutaan suorittaa myös onInput tapahtuman seurauksena. Näin input-komponentissa määritetyn useEffect-hookin sisällä kutsuttava funktio ja hookin riippuvuudet eivät aiheuttaisi ääretöntä silmukkaa. Input-elementin kentän arvon vaihtuessa lähetetään reducer-funktioon kentän vaihtunut arvo, komponentin validoinnin oikeellisuuden tila, elementin id-arvo sekä toiminnon tyyppi. Funktiossa luodaan kokonaisarvoa kuvastava boolean-muuttuja formIsValid, jolle annetaan arvona tosi. Objektit käydään läpi forIn-loopin avulla tarkastaen, täsmääkö input-komponentin id saadun toiminnon elementin id-arvoon. Tämän ollessa tosi tarkastetaan saadun muutoksen pätevyys. Palautusarvona saadaan vanha kopioitu tila, mahdolliset muutokset kenttien arvojen tiloissa sekä kaavakkeen pätevyys kokonaisuudessaan. Toinen dispatch-metodi, tyypiltään ”UPDATED_DATA”, vastaa datan asettamisesta tilanteissa, jossa kaavake on alustettava tyhjänä, kun tietokannasta haettava data ei ole komponentin renderöinti vaiheessa saatavilla. Näin esimerkiksi projektia päivittäessä kentät voidaan täyttää jo olemassa olevilla tietokantaan tallennetuilla arvoilla.

5.4 Sisään- ja uloskirjautuminen, konteksti

Sovelluksen näkymään ja toiminnallisuuteen vahvasti vaikuttavavia arvoja ovat tieto käyttäjän sisään kirjautumisesta, sisään kirjautuneen käyttäjän id sekä token. Tietojen liikuttaminen komponentista toiseen propseissa ”poraamalla” hierarkia puun läpi olisi äärimmäisen työlästä. Reactin kontekstirajapinta toimii erinomaisena ratkaisuna

varmistamaan, että kaikilla sovelluksen komponenteilla on tarvittaessa pääsy näihin tietoihin. Kontekstille luodaan erillinen tiedosto, jossa se alustetaan. Tiedostoon tuodaan createContext React -kirjastosta, jonka avulla uusi kontekstiobjekti luodaan. Kontekstiobjektiin määritetään arvoina käyttäjän id, token, sekä login- ja logout-funktiot. App.js tiedostoon tuotu kontekstiobjekti liitetään provider-komponentin kanssa kaikkien palautettavien reittien ympärille. Provider-komponenttiin liitetään arvona jaettava objekti ja näin varmistetaan, että sovelluksen jokaisella komponentilla on pääsy kontekstiobjektiin. App-komponenttiin lisätään tiloina käyttäjän id ”userId” sekä tokeni ”token” jotka viittaavat konteksti objektissa olevia arvoja. Login- ja logout-funktiot luodaan molemmat useCallback-hookkien ympärille. Hookeille ei anneta riippuvuuksia. Login-funktio vastaanottaa kolme argumenttia, jotka ovat käyttäjän id, token ja aika. Argumenttina saatu token ja id asetetaan kuvaamaan tilojaan. Mikäli aika-arvoa ei annettu argumenttina, aika-arvo asetetaan nykyseiksi ajaksi plus 720 000 joka vastaa 2 tuntia nykyhetkestä millisekunneissa. Argumenttina saatu arvo ei koe muutoksia. Aika-arvo kuvaa tokenin vanhenemisaikaa ja sille luodaan myös oma tila useState-hookkia käyttäen. Arvot tallennetaan objektina palvelimen paikalliseen tallennustilaan, joka on havainnollistettuna kuvassa 18.



Kuva 18. Käyttäjän data selaimen paikallisessa tallennustilassa.

Sivun uudelleenpäivitys tapauksissa laukaistaan useEffect-hookin avulla funktio, joka tarkistaa paikallisen tallennustilan ja ohjaa tallennetun objektin käyttäjädatan login-funktiolle. Logout-funktiossa asetetaan kaikille tiloille arvo null ja paikallisesta tallennustilasta poistetaan käyttäjän dataa kuvaava objekti. Automaattinen uloskirjautuminen tapahtuu, tokenin vanhentuessa, vertaamalla reaaliaikaa tokenin vanhenemisaika-arvoon ja asettamalla näiden erotus setTimeout-metodiin, joka ajan täyttyessä laukaisee logout-funktion. Metodi asetetaan useEffect-hookin sisällä ja sen riippuvuuksiksi asetetaan tokentila, aika-arvotila sekä logout-funktio. Mikäli logout-funktio havaitaan, suoritetaan clearTimeout-metodi ja ajastin näin ollen poistetaan käytöstä.

Sisäänkirjautumis- ja rekisteröintinäkymät toimivat perusteeltaan kaavakkeina. Näkymä koostuu maksimissaan kolmesta Input-komponentista ja kahdesta painikkeesta. Painikkeet vastaavat tietojen lähetyksestä ja tilan loginMode vaihtamisesta. Input-komponenttien määrä määräytyy tilan loginMode perusteella, jossa kuvataan, suoritetaanko sisäänkirjautumista vai rekisteröintiä. Suoritettaessa rekisteröintiä kenttiä on käytössä kolme, ne kuvastavat nimeä, sähköpostia sekä salasanaa. Kenttien arvoja sekä pätevyyttä hallitaan useForm-hookin avulla. Kun kaikki kentät on täytetty hyväksyttävästi, muuttuu kaavakkeen lähettämisestä vastaava painike käytettäväksi. Painike laukaisee käsittelijäfunktion, jossa fetch-rajapinnan avulla lähetetään POST-pyyntö palvelimelle rekisteröintiin tarkoitettuun reittiin. Operaatio on havainnollistettu kuvassa 19. Pyyntöön liitetään JSON-muodossa kenttien arvot. Vastauksena odotetaan objektia, joka sisältää käyttäjän id:n sekä tokenin. Kun vastaus on saatu, liitetään id ja token kontekstin avulla saatavaan login-funktioon argumentteina. Sisäänkirjautumisen logiikka toimii identtisesti, mutta nimelle ei tarvita kenttää.

```
try {
  const response = await fetch('http://localhost:5000/api/users/signup', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      name: formState.inputs.name.value,
      email: formState.inputs.email.value,
      password: formState.inputs.password.value,
    }),
  });

  const responseData = await response.json();
  if (!response.ok) {
    throw new Error(responseData.message);
  }

  auth.login(responseData.userId, responseData.token);
} catch (err) {
  console.log(err);
  setError(err.message || 'An error has occured.');
}
```

Kuva 19. Rekisteröintitietojen määritys ja lähetys.

5.5 Projektikomponentit

Kaikkien projektien näkymästä vastaava komponentti toimii lista/yksikkö perusteella samankaltaisesti kuin käyttäjien näkymä. Fetch-rajapinnan avulla palvelimelle tehtävä GET-pyyntö, joka palauttaa listan projekteista. Lista asetetaan ladattujen projektien tilamääritteeksi ja ohjataan proppina projektilistakomponentille. Projektilistakomponentissa saatu lista ja sen sisältämät arvot jaetaan map-metodin avulla yksittäisistä projekteista vastaaville komponenteille. Yksittäisessä projektikomponentissa otsikolle sekä sisällölle luodaan näkymä listaelementin sisällä. Elementissä on myös kaksi valinnaista painiketta. Projektin luojan ja sisäänkirjautuneen käyttäjän id arvon ollessa samat painikkeet renderöidään. Sisäänkirjautuneen käyttäjän id saadaan kontekstista. Ensimmäinen painike ohjaa käyttäjän projektin muokkausnäkymään ja toinen painike vastaa projektin poistamisesta. Poistamisoperaatiossa lähetetään fetch-rajapinnan kanssa ”DELETE” pyyntö palvelimelle reittiin, jota projektin id vastaa. Pyyntöön liitetään myös konteksti objektista saatava token, joka palvelinpuolella vaaditaan. Operaation onnistuneen suorittamisen jälkeen lähetetään kaikkien projektien näkymästä vastaavaan komponenttiin propsien avulla tieto tapahtumasta funktioon, joka päivittää ladattujen projektien listan ja kyseiseen informaatioon asetetun tilan. Projektien päivittämisnäkymässä käytetään kustomoitua useForm-hookkia ja kahta Input-komponenttia tietojen esittämiseen ja muokkaamiseen. Kun näkymä ladataan, suoritetaan saadulle projektin id:tä vastaavaan reittiin GET-pyyntö tietokantaan. Input-komponenttien näytettävät arvot asetetaan kuvaamaan saatuja tietoja. Kentissä olevia tietoja voi nyt muokata. Muokattujen tietojen lähetys tapahtuu UPDATE -painikkeella, joka antaa käskyn suorittaa päivitys funktion. Funktiossa suoritetaan PATCH-pyyntö fetch-rajapinnan avulla. Pyyntöön liitetään Input-komponenttien lähetyshetken arvot sekä kontekstista saatava token. Kun operaatio on suoritettu, ohjataan käyttäjän react-router paketista saatavan useHistory-hookin avulla käyttäjän omien projektien näkymään.

6 POHDINTA

Opinnäytetyössä tavoiteltiin MERN-pinon kaikkien osa-alueiden tutkimista ja prototyyppisovelluksen rakentamista pinon avulla. Pinon teknologioiden peruskäsitteet avataan ja prototyypin kehityksessä käytettiin myös edistyneempiä ominaisuuksia. Prototyypin kehityksessä haluttiin erityisesti tuoda esiin yleisesti esiintyviä vaiheita kehityksessä ja ominaisuuksia, joita web-sovelluksissa tänä päivänä esiintyy. Tietokannassa malleille luotiin toimiva One-To-Many-yhteys, jossa dataa toisesta mallista referoidaan. Palvelinpuolella rajapinnan avulla pystytään suorittamaan CRUD-operaatioita. Käyttäjätiedoista salasanat hajautetaan ennen niiden tallentamista tietokantaan. Kun dataa halutaan tallentaa tai muuttaa, varmistetaan palvelimella, että dataa vastaa määrityksiä. Rajapinnassa osa reiteistä asetettiin käytettäviksi vain, mikäli pyyntö pystyttiin todentamaan. Käyttöliittymässä React-hookit toimivat keskeisenä osana projektia ja logiikkaa pyritään kierrättämään mahdollisimman paljon. Tämä näkyy erityisesti komponenttien rakenteessa ja kustomoidun hookin kehittämisessä. Tiedonkulku tietokannasta käyttöliittymään asti pyrittiin pitämään yksinkertaisena ja koodi mahdollisimman hyvin ymmärrettävänä, kuitenkin käyttämällä JavaScriptin uusimpia ominaisuuksia.

Teknologioiden ja mahdollisten toteutustapojen tutkimiseen käytettiin huomattavasti aikaa ja lopputulokseen kirjoittaja on tyytyväinen. Ymmärrykseni sovelluksen kehittämisen vaiheista ja prosesseista kehittynyt. Mahdollisia jatkokehitysideoita ovat käyttöliittymän mobiilioptimointi ja tiedostojen tallentaminen tietokantaan, jotta käyttäjille voitaisi määritellä profiilikuvat. Tämä olisi mahdollista esimerkiksi multer-kirjaston avulla.

Opinnäytetyön aihe valikoitui kirjoittajan omien mielenkiinnon kohteiden sekä full stack JavaScriptin kasvavan suosion vuoksi. Node.js ja React ovat vakiinnuttaneet paikkansa web-sovellusten ympäristössä. Viimevuosia vahvassa nousussa ollut TypeScript on nostanut JavaScriptiin tukeutuvien teknologioiden suosiota entisestään. Full stack -kehittäjien kysyntä on kasvanut alalla huomattavasti viimeisien vuosien aikana. Pinojen kehittyessä ja muuttuessa jatkuvasti, samaa odotetaan myös kehittäjiltä. Yhden ohjelmistopinon hallitseminen helpottaakin huomattavasti jatkossa muiden pinojen oppimista. Opinnäytetyöstä kertynyt tieto mahdollistaa osaamiseni kehittämisen tulevaisuudessa kokonaisvaltaisempaan suuntaan.

LÄHTEET

[1] HackerRank, 2020 HackerRank Developer Skills Report [www-sivu]. Saatavilla: https://research.hackerrank.com/developer-skills/2020 (Luettu: 25.9.2020)

[2] Ecma International, ECMAScript® 2020 Language Specification [www-sivu]. Saatavilla: http://www.ecma-international.org/ecma-262/11.0/index.html (Luettu: 21.9.2020)

[3] Stack Overflow, 2020 Developer Survey [www-sivu]. Saatavilla: https://insights.stackoverflow.com/survey/2020 (Luettu: 21.9.2020)

[4] Node.js, About Node.js [www-sivu]. Saatavilla: https://nodejs.org/en/about/ (Luettu: 11.7.2020)

[5] Ofoegbu, V., The only NodeJs introduction you'll ever need [www-sivu]. Saatavilla: https://codeburst.io/the-only-nodejs-introduction-youll-ever-need-d969a47ef219 (Luettu: 11.7.2020)

[6] Santos, L., Node.js Under The Hood #3 - Deep Dive Into the Event Loop [www-sivu]. Saatavilla: https://dev.to/khaosdoctor/node-js-under-the-hood-3-deep-dive-into-the-event-loop-135d (Luettu: 11.7.2020)

[7] Express, Fast, unopinionated, minimalist web framework for Node.js [www-sivu]. Saatavilla: https://expressjs.com/ (Luettu: 30.5.2020)

[8] Facebook Inc., React A JavaScript library for building user interfaces [www-sivu]. Saatavilla: https://reactjs.org/ (Luettu: 30.5.2020)

[9] Hamedani, M., React Virtual DOM Explained in Simple English [www-sivu]. Saatavilla: https://programmingwithmosh.com/react/react-virtual-dom-explained/ (Luettu: 31.5.2020)

[10] Facebook Inc., Hooks API Reference [www-sivu]. Saatavilla: https://reactjs.org/docs/hooks-intro.html (Luettu: 23.9.2020)

[11] Facebook Inc., Introducing Hooks [www-sivu]. Saatavilla: https://reactjs.org/docs/hooks-reference.html (Luettu: 23.9.2020)

[12] Facebook Inc., Rules of Hooks [www-sivu]. Saatavilla: https://reactjs.org/docs/hooks-rules.html (Luettu: 23.9.2020)

[13] Brontesewell, React Router [www-sivu]. Saatavilla: https://dev.to/brontesewell/react-router-530n (Luettu: 10.8.2020)

[14] Rouse, M., Definition MongoDB [www-sivu]. Saatavilla: https://searchdatamanagement.techtarget.com/definition/MongoDB (Luettu: 16.7.2020)

[15] Karnik, N., Introduction to Mongoose for MongoDB [www-sivu]. Saatavilla: https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/ (Luettu: 16.7.2020)

[16] Rohan, P., How bcryptjs works [www-sivu]. Saatavilla: https://medium.com/javascript-in-plain-english/how-bcryptjs-works-90ef4cb85bf4 (Luettu: 1.8.2020)

[17] Kumar, S., How JSON Web Token(JWT) authentication works? [www-sivu]. Saatavilla: https://medium.com/@sureshdsk/how-json-web-token-jwt-authentication-works-585c4f076033 (Luettu: 13.6.2020)

[18] McLarty, R., What is JSON Web Token [www-sivu]. Saatavilla: https://medium.com/myplanet-musings/what-is-a-json-web-token-2193f383e963 (Luettu: 13.6.2020)

[19] MDN web docs, Making asynchronous programming easier with async and await [www-sivu]. Saatavilla: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await (Luettu: 28.6.2020)

[20] Express, Error Handling [www-sivu]. Saatavilla: https://expressjs.com/en/guide/error-handling.html (Luettu: 25.4.2020)

[21] Express, Routing [www-sivu]. Saatavilla: https://expressjs.com/en/guide/routing.html (Luettu: 26.4.2020)

[22] JWT, Introduction to JSON Web Tokens [www-sivu]. Saatavilla: https://jwt.io/introduction/ (Luettu: 20.9.2020)

[23] MDN web docs, Fetch API [www-sivu]. Saatavilla: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (Luettu: 26.9.2020)