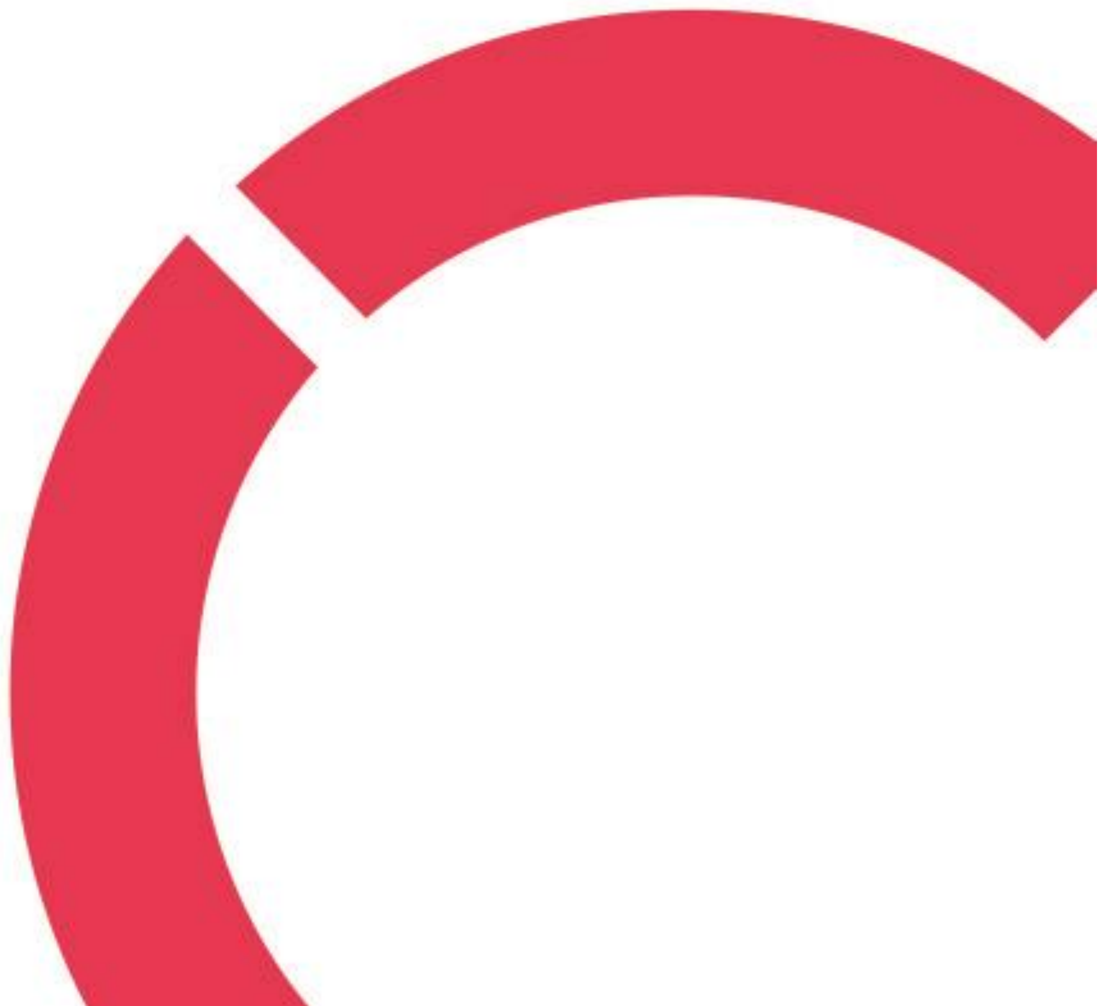
Antti-Pekka Sipola

FULL-STACK SOSIAALINEN MEDIA -SOVELLUKSEN KEHITTÄMINEN

Opinnäytetyö
CENTRIA-AMMATTIKORKEAKOULU
Tieto- ja viestintätekniikan koulutus
Maaliskuu 2022



centria ammattikorkeakoulu

TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Centria-ammattikorkeakoulu	**Aika** Maaliskuu 2022	**Tekijä/tekijät** Antti-Pekka Sipola
Koulutus Tieto- ja viestintätekniikka		☒ AMK ☐ YAMK
Työn nimi FULL-STACK SOSIAALINEN MEDIA -SOVELLUKSEN KEHITTÄMINEN		
Työn ohjaaja Heikki Ahonen		**Sivumäärä** 46 + 1

Tämän työn aiheena oli kehittää full-stack sosiaalinen media -sovellus. Työssä hyödynnettiin teknologioita, kuten React.js, Redux, Node.js, Express.js ja relaatiotietokannat. Näiden teknologioiden lisäksi sovelluksessa hyödynnettiin monia muitakin kirjastoja.

Opinnäytetyö kuvailee sovelluksen kehityksen prosessia, ja se käy myös läpi ja tutkii erilaisia keskeisiä projektissa hyödynnettyjä teknologioita.

Työn tuloksena saatiin kehitettyä toimiva full-stack sosiaalinen media -sovellus, joka sopeutuu näytön leveyden muutoksiin. Sovelluksen ominaisuuksia ovat muun muassa rekisteröiminen, kirjautuminen, postauksien luominen sekä niiden äänestäminen. Käyttäjät voivat myös kommentoida ja äänestää postauksia ja muita kommentteja.

Asiasanat
React.js, Node.js, Express.js, relaatiotietokannat, full-stack-kehitys, webkehitys, sosiaalinen media

Centria University of Applied Sciences

ABSTRACT

Centria University of Applied Sciences	**Date** March 2022	**Author** Antti-Pekka Sipola
Degree programme Information and communication technology		
Name of thesis DEVELOPMENT OF A FULL-STACK SOCIAL MEDIA APPLICATION		
Centria supervisor Heikki Ahonen		**Pages** 46 + 1

The goal of this thesis was to develop a full-stack social media application. In the project the technologies used were React.js, Redux, Node.js, Express.js, relational databases and many other libraries.

The thesis describes the process of developing the application and it also dives deeper into some important technologies for the application.

The end result of the project was a functional full-stack social media application that is responsive to screen width changes. Some main features of the application are registration, logging in and creating or browsing posts. Users can also comment or vote posts or other comments.

Key words

React.js, Node.js, Express.js, relational databases, full-stack development, web development, social media

KÄSITTEIDEN MÄÄRITTELY

Backend

Sovelluksen palvelinpuolen ohjelmistosta käytetty nimitys.

CSS

Verkkoympäristössä käytettävä tyyliohje verkkosivuille.

DOM

Kuvaa muun muassa HTML-dokumentin rakennetta puuna, jonka eri olioita voi käsitellä eri tavoilla hyödyntäen esimerkiksi JavaScriptiä.

Frontend

Sovelluksen asiakaspuolen ohjelmistosta käytetty nimitys.

Full-stack

Sovelluksen asiakaspuolen ja palvelinpuolen ohjelmistojen yhteisnimitys.

HTML

Standardi merkintäkieli verkkosivuille.

HTTP

Selaimissa ja WWW-palvelimissa tiedonsiirtoon tarkoitettu protokolla.

JavaScript

Suosituin pääasiassa verkkoympäristössä käytettävä ohjelmointikieli.

LocalStorage

Mahdollistaa avain/arvo-olioiden tallennuksen selaimeen ilman vanhentumisaikaa.

REST API -rajapinta

REST-arkkitehtuurilla suunniteltu ohjelmointirajapinta, jonka kautta ohjelmat voivat tehdä pyyntöjä tai lähettää tietoja toisilleen.

Web API

Selaimessa olevia ohjelmointirajapintoja, jotka laajentavat selaimen ominaisuuksia. Ne auttavat monimutkaisten operaatioiden suorittamisessa. Web API voi esimerkiksi suorittaa taustalla tehtäviä, joiden suoritusta muuten jouduttaisiin odottamaan koodissa.

WebSocket

Viestintä-protokolla, joka sallii kommunikoinnin kahteen suuntaan yhden TCP-yhteyden päällä.

TIIVISTELMÄ
ABSTRACT
KÄSITTEIDEN MÄÄRITTELY
SISÄLLYS

1 JOHDANTO 1

2 TAUSTATIETO 4
2.1 JavaScript 4
2.2 React.js 5
2.2.1 Flux-arkkitehtuuri 6
2.2.2 Virtual DOM 7
2.2.3 Komponentit 8
2.2.4 Koukut 9
2.2.5 Redux 9
2.3 Node.js 10
2.3.1 NPM 11
2.3.2 Express 12

3 FULL-STACK SOSIAALINEN MEDIA -SOVELLUKSEN TOTEUTUS 13
3.1 Frontend-puolen projektin aloitus 13
3.1.1 Frontend-puolen alustus 13
3.1.2 TailwindCSS 15
3.1.3 React-redux 21
3.2 App.js 23
3.2.1 React-router-dom 24
3.2.2 Redux-varasto 25
3.3 Käyttäjän kirjautuminen 25
3.4 Postauksien selaaminen 27
3.4.1 Vain tarvittavien postauksien hakeminen 28
3.4.2 Muokattu koukku ja WebSocket 29
3.5 Postauksien äänestäminen 32
3.6 Backend-puolen rakenne, tietokanta ja HTTP REST API -rajapinnat 32
3.6.1 Relaatiotietokannan toteutus 33
3.6.2 HTTP REST API -rajapinnat 36
3.7 AWS S3 38
3.8 Yksittäisen postauksen sivu 39
3.9 Uuden postauksen luomisen sivu 41

4 POHDINTA 42

LÄHTEET 43

LIITTEET
Liite 1. Projektin lähdekoodi

KUVAT
KUVA 1. JavaScript tapahtumasilmukka 5
KUVA 2. Flux-arkkitehtuuri 7
KUVA 3. React-komponentin elinkaari 9

KUVA 4. Redux toimintamalli ... 10
KUVA 5. React-sovelluksen alustamiseen sekä sovelluksen käynnistämiseen tarkoitetut komennot ... 14
KUVA 6. Frontend-puolen hakemistojen rakenne ... 15
KUVA 7. Postcss.config.js tiedosto ... 16
KUVA 8. Tailwind.config.js tiedosto ... 16
KUVA 9. CSS-koodin käsittely PostCSS:llä ... 17
KUVA 10. Vaaditut tailwindCSS-koodirivit pää CSS-tiedostossa ... 17
KUVA 11. Esimerkki tailwindCSS-kirjaston CSS-luokkien käytöstä ... 18
KUVA 12. Sovelluksen etusivun ulkoasu kapealla näytöllä ... 19
KUVA 13. Sovelluksen etusivun ulkoasu leveällä näytöllä ... 20
KUVA 14. Redux-siivujen, toimintojen sekä vähentäjien määrittely ... 22
KUVA 15. Redux-varaston luominen ... 23
KUVA 16. Sovelluksen reitityksen määrittely ... 24
KUVA 17. Provider-komponentilla Redux-varaston tarjoaminen lapsikomponenteille ... 25
KUVA 18. Sovelluksen kirjautumislomake ... 27
KUVA 19. Metodi, jolla seurataan viimeisen postauksen sijaintia näytöllä ... 28
KUVA 20. Postausten elementtien määrittely map-metodilla ... 29
KUVA 21. Muokatun koukun tilojen määrittely ... 30
KUVA 22. Muokatun koukun WebSocket-viesti ... 30
KUVA 23.WebSocket-protokollan toimintaperiaate ... 31
KUVA 24. Backend-puolen WebSocket-viestien käsittelijä, joka hakee lisäpostauksia ... 31
KUVA 25. Backend-puolen hakemistojen rakenne ... 33
KUVA 26. Heroku komentoja PostgreSQL-tietokannan luomiseen ... 34
KUVA 27. Sovelluksen tietokannan kaavio ... 35
KUVA 28. Sequelize-tietokantataulumallien yhteyksien määrittely index.js-tiedostossa ... 36
KUVA 29. Express-reitittimien käyttöönotto ... 37
KUVA 30. Postauksen kuvan lataaminen AWS S3-varastoon ... 38
KUVA 31. Yksittäisen postauksen sivu leveällä näytöllä ... 39
KUVA 32. Yksittäisen postauksen sivu kapealla näytöllä ... 40
KUVA 33. Uuden postauksen luomisen sivu ... 41

TAULUKOT

TAULUKKO 1. Backend-puolen tarjoamat HTTP REST API -rajapinnat ... 37

1 JOHDANTO

Sosiaalinen media on nykyään osa monien ihmisten arkea. Sosiaalinen media -sovellukset yleensä tarvitsevat sekä palvelin- että asiakaspuolet eli full-stack-sovelluksen. Sosiaalinen media tarkoittaa erilaisia internetin välityksellä toimivia ympäristöjä, joissa käyttäjät voivat kommunikoida tai jakaa sisältöä toisilleen. Jaettu sisältö voi olla esimerkiksi tekstiä, videoita tai kuvia. Sosiaalisessa mediassa käyttäjät voivat olla ryhmittyneitä yhteisöihin. Eri sosiaalisen median tyyppejä ovat esimerkiksi foorumit, sosiaaliset verkostot, blogit sekä videoiden tai kuvien jakamisen sivustot. Sosiaalista mediaa käytetään usein tietokoneilla tai mobiililaitteilla. Syitä, joiden vuoksi ihmiset käyttävät sosiaalista mediaa, voivat olla esimerkiksi verkostoituminen tai yhteyksissä pysyminen sekä vuorovaikutus muiden ihmisten kanssa.

Tämä opinnäytetyöaihe valittiin halusta kehittyä ja oppia uutta ohjelmoijana. Opinnäytetyön tavoitteena on oppia uutta ohjelmoinnista ja kehittää full-stack-sovellus, jolla on monia sosiaalisen median ominaisuuksia kuten rekisteröinti, kirjautuminen, uusien postauksien luominen sekä niiden selaaminen, äänestäminen ja kommentointi.

Full-stack-sovellus tarkoittaa sovellusta, jolla on asiakas- ja palvelinosiot, toisin sanoen frontend- sekä backend-puoli. Frontend-puolella kehitetään sovelluksen graafista käyttöliittymää eli sovelluksen osiota, jonka kautta käyttäjä tekee toimintoja sovelluksessa. Backend-puolella puolestaan kehitetään sovelluksen taustajärjestelmiä, tietokantoja sekä toimintalogiikkaa. Frontend- ja backend-puoli toimivat yhdessä niin, että frontend-puoli lähettää tarvittaessa pyyntöjä backend-puolelle ja sitten backend-puolen tehtävänä voi olla esimerkiksi datan prosessointi tai tiedon haku tietokannasta, jonka jälkeen se lähettää vastauksen frontend-puolelle.

Sovelluksessa tulee ratkaista ongelmia, kuten miten käyttäjien, postauksien, kommenttien sekä yhteisöjen tiedot varastoidaan ja miten ne liittyvät toisiinsa. Sovelluksessa tulee myös olla ominaisuuksia, jotka hakevat edellä mainittuja tietoja ja sitten näyttävät ne käyttäjälle eri tavoilla. Muita sovelluksen kehityksessä selvitettäviä ongelmia ovat esimerkiksi käyttäjän kirjautuminen, uusien postauksien ja kommenttien luominen sekä niiden äänestäminen ja kommentointi. Näitä ongelmia sovelluksessa ratkotaan hyödyntämällä eri teknologioita kuten relaatiotietokantoja tai erilaisia kirjastoja kuten esimerkiksi TailwindCSS-kirjastoa, jolla saadaan muotoiltua verkkosivun ulkoasua.

Sovelluksen frontend-puolella käytetään monia eri teknologioita. Sovelluksen frontend rakennetaan React.js-kirjastoa hyödyntäen, joka on kirjasto käyttöliittymien rakentamista varten. Sovelluksen toimimiseen tarvittava logiikka frontend-puolella kirjoitetaan JavaScript-ohjelmointikielellä. JavaScriptia hyödyntäen voidaan verkkosivun ulkoasua muuttaa dynaamisesti käyttäjän toiminnan seurauksena. Frontend- sekä backend-puolella käytetään tässä sovelluksessa JavaScriptin viimeisintä ES6-versiota. Käyttöliittymän rakennuksessa käytetään myös kirjastoja kuten TailwindCSS sekä MUI. TailwindCSS on kirjasto, joka tuo valmiiksi määriteltyjä CSS-luokkia sovellukseen, joita voi antaa suoraan HTML-elementtien ”className”-kentän arvoksi, joka sitten muuttaa kyseisen elementin ulkoasua verkkosivulla. MUI on kirjasto, josta voi hakea valmiiksi luotuja ja tyyliteltyjä käyttöliittymäkomponentteja, kuten painikkeita tai tekstikenttiä, joita voidaan hyödyntää sovelluksen käyttöliittymää kehitettäessä.

Sosiaalista mediaa käytetään yleensä sekä mobiililaitteilla että isompinäyttöisillä laitteilla, joten sovelluksen käyttöliittymästä kehitetään näytön leveyteen sopeutuva hyödyntämällä TailwindCSS-kirjastoa ja sen valmiiksi määrittelemiä keskeytyskohtia, jotka ovat näytön leveyden kohtia, joissa voi muokata elementtien CSS-luokkia, niin että sovelluksen ulkoasu pysyy ehjänä kaikilla näytön leveyksillä.

Frontend-puolella käytetään myös axios- ja socket.io-kirjastoa backend-puolen kanssa kommunikointiin. Axios-kirjastoa sovelluksessa hyödynnetään niin, että sen avulla tehdään HTTP-pyyntöjä backend-puolelle. Socket.io-kirjastolla puolestaan lähetetään ja kuunnellaan WebSocket-viestejä, joiden toinen osapuoli on backend-puoli. WebSocket-viestintä toimii niin, että kun yhteys on luotu frontend-puolen ja backend-puolen välille, sen avulla osapuolet voivat lähettää ja vastaanottaa viestejä reaaliaikaisesti keskenään ilman, että yhteyttä suljetaan ennen kuin toinen osapuoli päättää sulkea sen. HTTP-pyynnöissä yhteys suljetaan, kun frontend-puoli on saanut vastauksen backend-puolelta pyyntöönsä.

Sovelluksen backend-puolella käytetään relaatiotietokantaa ja Amazon Web Services S3 -pilvipalveluvarastoa eli AWS S3 -pilvipalveluvarastoa datan varastointiin. Relaatiotietokannan avulla sovelluksessa saadaan varastoitua dataa sekä toteutettua tarvittaessa monen-suhde-moneen -yhteyksiä, joita sovelluksessa tarvitaan yhteisöön liittyneiden käyttäjien sekä käyttäjän äänestämien postauksien seuraamiseen. AWS S3 -varastoa sovelluksessa käytetään postauksille annettujen kuvien varastoimiseen.

Tietokantamallit backend-puolella toteutetaan Sequelize-kirjastoa hyödyntäen. Sequelize-kirjasto on ORM-työkalu, jonka avulla voidaan kommunikoida relaatiotietokannan kanssa ilman SQL-kyselykielen kirjoittamista. SQL-kieli on relaatiotietokantojen kanssa kommunikointiin tarkoitettu kyselykieli.

Backend-puoli toimii Node.js ajonaikaisessa ympäristössä. Node mahdollistaa JavaScriptin kirjoittamisen backend-puolella. Backend-puolen logiikka kirjoitetaan sovelluksessa JavaScriptillä. Backend-puolen sovelluskehyksenä sovelluksessa käytettiin Express.js-kirjastoa. Sen avulla toteutettiin HTTP REST API -rajapinnat. Sovelluksessa hyödynnettiin myös muun muassa sen middleware-ominaisuutta, jonka avulla voidaan käsitellä valitut pyynnöt halutulla tavalla. Backend-puolella WebSocket-viestien vastaanottamiseen ja lähettämiseen käytettiin Socket.io-kirjastoa.

Tämän sovelluksen frontend- ja backend-puoli toimivat molemmat Node.js ajonaikaisessa ympäristössä. Sovelluksessa hyödynnetään Node Package Manager -pakettienhallintatyökalua, jonka avulla voidaan esimerkiksi ladata ja asentaa kirjastoja. NPM-pakettienhallintatyökalua käytetään frontend- ja backend-puolilla ja se asennetaan automaattisesti Noden kanssa.

Opinnäytetyössä käytettiin lähteinä enimmäkseen dokumentaatiosivustoja ja asiantuntijoiden artikkeleita.

2 TAUSTATIETO

2.1 JavaScript

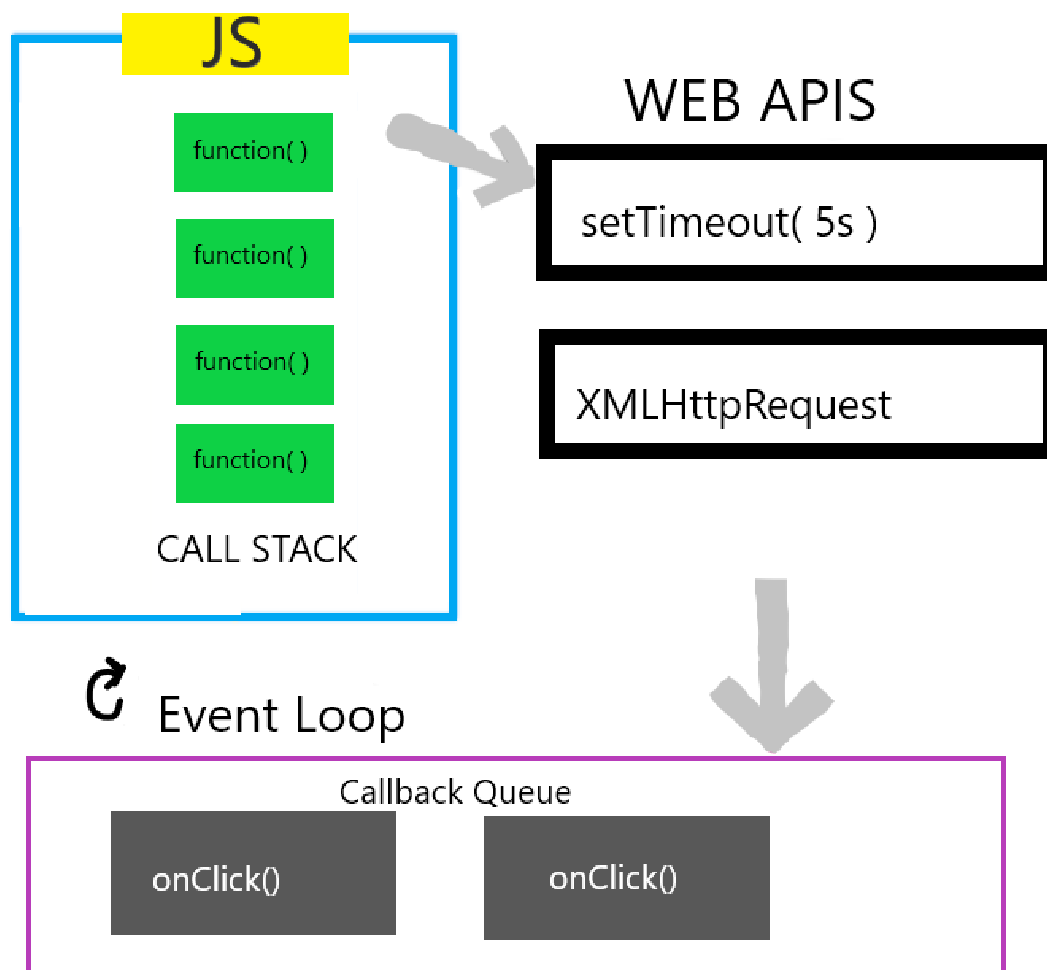
JavaScript on vuonna 1995 julkaistu Netscapen kehittämä ohjelmointikieli, jota käytetään pääasiassa verkkoympäristössä ohjelmointiin graafisissa webselaimissa. Se on kompakti ohjelmointikieli, jonka avulla HTML-dokumenteista voidaan tehdä dynaamisesti käyttäjän toiminnan perusteella mukautuvia ilman, että verkkosivua täytyisi päivittää joka kerta kun näytöllä halutaan näkyvän muutos. JavaScriptiä ei tarvitse koota ennen suoritusta vaan koodi täytyy syöttää selaimessa olevalle tulkille, joka saa sen toimimaan selaimessa. (Javatpoint 2022.)

JavaScript on dynaamisesti tyypitetty ohjelmointikieli. JavaScriptin muuttujille voi antaa tyypiksi esimerkiksi var tai let, jotka sitten voivat varastoida mitä tahansa tyyppiä edustavia arvoja kuten numeroita tai merkkijonoja. Tämä tekee JavaScriptistä dynaamisesti tyypitetyn. (Hiwarale 2018.)

JavaScriptistä tuli ECMA-standardontijärjestön standardoima vuonna 1997 ja sen virallinen nimi on ECMAScript. ECMAScriptistä on tullut monia versioita sen julkaisemisen jälkeen. ECMAScriptin eri versioiden lyhenteitä ovat ES1, ES2, ES3, ES5 ja ES6. ES6-versio lisäsi ECMAScriptiin eri ominaisuuksia kuten let-tyyppi, const-tyyppi, nuolimetodit, luokat, lupaukset, asynkroniset metodit, lisää listametodeja sekä monia muita ominaisuuksia. ES6 toimii kaikilla suosituimmilla selaimilla, mutta ei Internet Explorer -selaimella. (W3schools.)

JavaScriptin tapahtumasilmukkaa on kuvailtu kuvassa 1 ja se toimii selaimessa niin, että kun koodissa suoritetaan funktio, se lisätään kutsupinoon odottamaan suoritusta. Kutsupinosta suoritetaan metodeja yksi kerrallaan, koska JavaScript on yksisäikeinen ohjelmointikieli. Koska metodeja suoritetaan yksi kerrallaan, koodin suoritus voi olla hidasta, jos metodin suoritus kestää pitkän ajan. Tämän välttämiseksi JavaScriptissä metodeista voi tehdä asynkronisia. Asynkronisesti suoritettava metodi tarkoittaa sitä, että metodin suoritusta ei jäädä odottamaan JavaScriptin koodissa ja metodi palauttaa arvon vasta sitten kun se on valmis. Kun metodi suoritetaan asynkronisesti, se siirretään web API:lle esimerkiksi odottamaan suoritusta myöhemmin koodissa tai suorittamaan paljon aikaa vievää pyyntöä palvelimelle. Kun web API:lla tehty pyyntö tai odotus on valmis, sen suorituksen tuloksena

suoritettava koodi siirretään takaisinkutsuttavien metodien listalle, josta ne siirretään JavaScriptin kutsupinolle suoritettavaksi sitten kun JavaScriptillä ei ole enää suoritettavia metodeja jonossa sen omassa kutsupinossaan. (Hiwarale 2018.)



KUVA 1. JavaScript tapahtumasilmukka. (Roberts 2014)

2.2 React.js

React on Facebookin julkaisema ja kehittämä avoimen lähdekoodin JavaScript-kirjasto, jota hyödynnetään käyttöliittymien rakentamisessa. Se julkaistiin vuonna 2013, jonka jälkeen siitä on julkaistu monia uusia versioita. Reactilla on paljon erilaisia ominaisuuksia, jotka tekevät käyttöliittymien rakentamisesta nopeaa sekä yksinkertaista. Reactissa käyttöliittymiä rakennetaan komponenttien avulla. React-yhteisö on luonut Reactille paljon erilaisia kirjastoja, joita voidaan

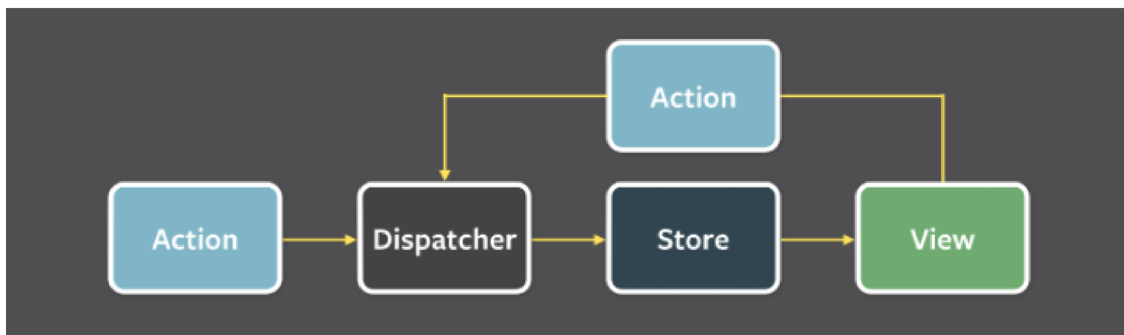
hyödyntää sovellusten rakentamisessa. Verkkosivustoja, joiden rakentamisessa on hyödynnetty Reactia ovat muun muassa Facebook, Instagram, Netflix ja Reddit. (Tutorialspoint.)

2.2.1 Flux-arkkitehtuuri

Reactissa sovelluksen tilan hallinta ja komponenttien uudelleen renderöinti noudattaa flux-arkkitehtuuria (KUVA 2). Flux-arkkitehtuuri on tapa, jolla sovelluksen tilaa varastoidaan ja jolla sitä muutetaan. Siinä komponenttien tilaa voidaan muuttaa toimintojen avulla, esimerkiksi käyttäjän toiminnan seurauksena. Toiminta lähettää tilan muutoksen varastoon, joka sitten saa aikaan komponentin uudelleen renderöitymisen näytölle. Varastot varastoivat sovelluksen tilaa globaalisti ilman, että komponentilla vaadittaisiin olevan omaa tilaansa. (Luukkainen 2022.)

Flux-arkkitehtuurissa sen yksisuuntaisen tiedonvirtauksen seurauksena dataa on helpompi hallita, sovelluksen virheitä on helpompi korjata ja niitä on myös helpompi välttää. Yksisuuntaista tiedonvirtausta hyödynnetään pääasiassa toiminnallisessa reaktiivisessa ohjelmoinnissa. Yksisuuntainen tiedonvirtaus tarkoittaa sitä, että datalla on ainoastaan yksi suunta, johon se voi kulkea sovelluksen sisällä. Reactissa tämä tarkoittaa sitä, että lapsikomponentti ei voi suoraan muuttaa vanhempikomponentilta saamaansa muuttujaa. Reactissa tilan muutos komponentissa vaikuttaa ainoastaan sen lapsikomponentteihin eikä vanhempikomponenttiin tai sisarkomponentteihin. (Immukul 2019.)

Lähettäjä on datavirtauksen hallintaan tarkoitettu osa arkkitehtuuria. Sen tarkoitus on lähettää toimintoja varastolle. Sen päätehtävänä on varastoida varastojen rekisteröintejä ja niiden takaisinkutsumetodeja. Se toimii niin, että kun lähettäjälle annetaan uusi toiminta, kaikki varastot ottavat vastaan kyseisen toiminnan takaisinkutsumetodien kautta. Takaisinkutsumetodissa toiminta käsitellään niin, että sen tyyppi-parametriä verrataan takaisinkutsumetodissa määriteltyihin toimintojen tyyppeihin ja sitten käsittelyn tuloksen perusteella varastossa tehdään muutoksia tilaan. Sitten kun tilaan on tehty muutos, varasto ilmoittaa muutoksesta ja sitten kyseisen muutoksen perusteella saadaan päivitettyä muutos näytölle. (Tay 2019.)



KUVA 2. Flux-arkkitehtuuri. (Luukkainen 2022)

2.2.2 Virtual DOM

DOM eli Document Object Model kuvailee HTML-dokumentin rakennetta ja sen olioita puuna (Mozilla 2022). Virtual DOM on Reactin kopio oikeasta DOM:ista. Sen avulla DOM päivitetään niin, että React tekee ensin kaikki muutokset Virtual DOM:iin ja sitten se vertaa sen uutta versiota vanhempaan versioon, joka edelsi muutoksia ja sitten se etsii ainoastaan muuttuneet oliot päivitetystä virtual DOM:ista ja sitten päivittää kyseisten etsittyjen muuttuneiden olioiden perusteella oikean DOM:in. (Codeacademy)

ReactDOM-kirjaston avulla voidaan suorittaa virtual DOM:n päivittäminen tai luominen (Reactjs 2021b). Se luo puun React-elementtejä, jonka muutoksien perusteella päivitetään oikea DOM (Reactjs 2021b). React hakee uuden puun luomiseen vaaditut vähiten suoritusvoimaa vaativat operaatiot hyödyntämällä diffing-algoritmiä. Diffing-algoritmi toimii nopeammin kuin jotkut muut algoritmit, joita voitaisiin myös käyttää tähän tarkoitukseen. Diffing-algoritmi olettaa, että kaksi eri tyyppiä olevat elementit luovat erilaiset puut ja että koodissa voidaan antaa elementille key-ominaisuus, joka sanoo, että se on vakaa renderöitäessä. Kun diffing-algoritmia käytetään, se ensin vertaa verrattavien puiden juuri-elementtejä. Jos juuri-elementeillä on eri tyypit, sitten React luo kokonaan uuden puun vanhan tilalle. Kun luodaan uutta puuta, vanhat DOM-elementit poistetaan kokonaan ja niiden paikalle asetetaan uusia DOM-elementtejä. Jos elementillä jokin sen ominaisuuksista muuttuu, sitä ei korvata kokonaan uudella elementillä vaan siihen päivitetään ainoastaan vaaditut muutokset. (Reactjs 2021b.)

Virtual DOM:issa olevat objektit ovat kopioita oikeassa DOM:issa olevista objekteista. Virtual DOM ei voi tehdä muutoksia suoraan näytölle vaan, jos halutaan tehdä muutoksia näytölle, täytyy muutokset tehdä oikeaan DOM:iin. DOM:in päivittäminen virtual DOM:in avulla on nopeaa, koska virtual

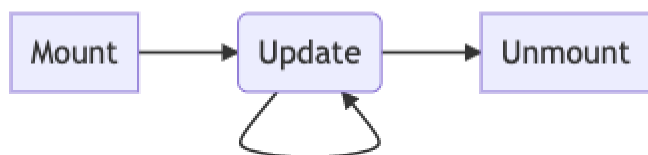
DOM:in päivittäminen ei saa aikaan muutoksia näytölle ja se tekee myös vain vaaditut päivitykset DOM:iin. DOM:in päivittäminen on hidasta, koska kun sitä päivitetään, samalla muutos näkyy myös näytöllä. (Codeacademy.)

2.2.3 Komponentit

Reactissa käyttöliittymä voidaan jakaa komponentteihin. Komponentit ovat pääasiassa JavaScript-metodeja, joille voi syöttää arvoja. Arvoja joita komponentille syötetään ei saa muuttaa suoraan komponentin koodissa. Komponentteja käytetään muissa komponenteissa niin, että ne sijoitetaan JSX-koodin sisälle. Komponentit palauttavat React-elementtejä, jotka kuvailevat sitä mitä sovelluksen pitäisi näyttää käyttäjälle. Samaa komponenttia voidaan käyttää uudelleen monessa eri paikassa koodia. Komponentteja voidaan määritellä JavaScript-metodeilla sekä ES6-luokilla. (Reactjs 2022.)

Reactin komponenteissa voidaan käyttää JSX:a, joka on Reactin JavaScript-laajennus, jonka avulla JavaScriptissä voidaan kirjoittaa HTML:n kaltaista kieltä. Tämä tekee React-komponenttien ohjelmoinnista helpompaa ja suoraviivaisempaa. Se mahdollistaa muun muassa komponenteille tyylien antamisen suoraan HTML-elementin "style"-kentän arvoksi ilman, että luodaan erillinen CSS-tiedosto tyylin määrittelemiseksi. Se mahdollistaa myös esimerkiksi elementtien suoraviivaisen ja dynaamisen lisäämisen HTML:ään malliliteraalien kautta. Malliliteraaleja käytetään niin, että HTML:n sekaan voi lisätä aaltosulkeiden sisällä esimerkiksi merkkijono-muuttujan arvon. (Elliott 2020.)

Reactissa komponenteilla on elinkaari (KUVA 3). Sen tarkoitus on estää komponentin tilan muutos, silloin kun React on piirtämässä kyseistä komponenttia. Tilan muutoksen estääkseen piirtämisen aikana, menee komponentti sellaiseen tilaan, jossa sen tilaa ei voi muuttaa ja sitten komponentti vasta piirretään. Piirtämisen jälkeen tilan muutos on taas mahdollista. Komponenttien elinkaari koostuu DOM:iin kiinnityksestä, komponentin päivityksestä sekä irrottamisesta DOM:ista. Päivitysvaiheessa komponentti menee kolmen eri vaiheen läpi, joita ovat renderöinti, ennakkositoutuminen ja sitoutuminen. Kun komponentin tila muuttuu, päivitysvaihe käynnistyy uudelleen eli se ensin renderöidään, sitten ennakkosidotaan ja sitten sidotaan DOM:iin. (Elliott 2020.)



KUVA 3. React-komponentin elinkaari. (Elliott 2020)

2.2.4 Koukut

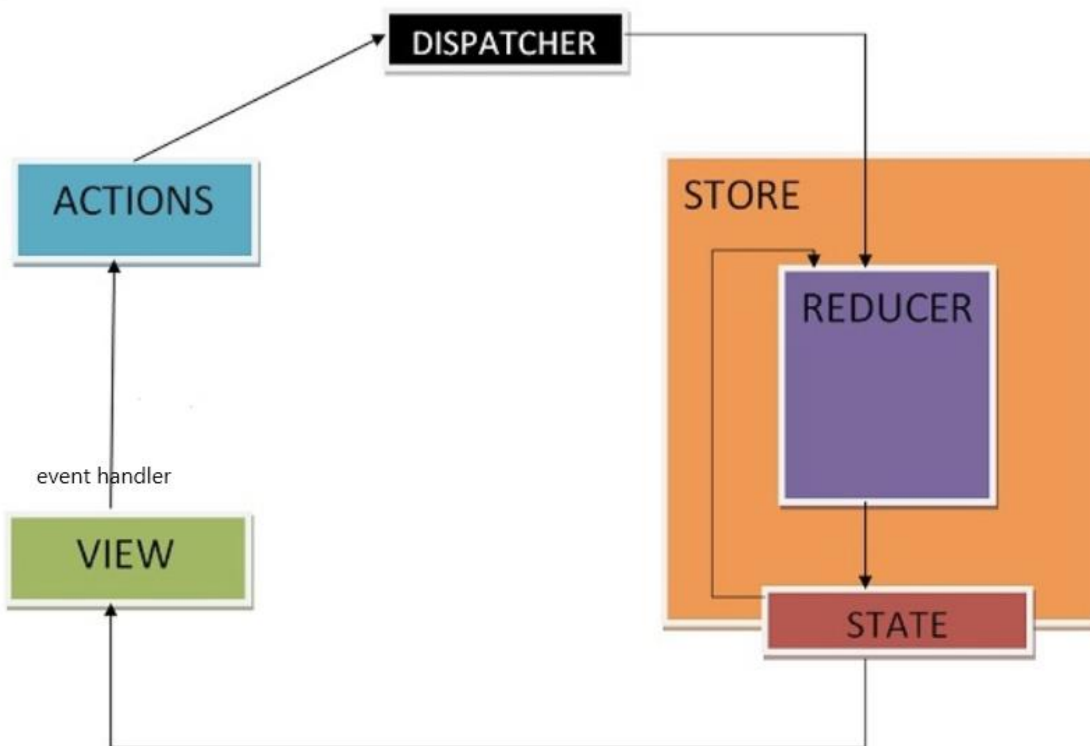
Koukut julkaistiin Reactin versiossa 16.8. Reactissa koukku on metodi, joka pystyy käyttämään Reactin ominaisuuksia kuten useState-koukkua, joka antaa metodille oman tilansa (Reactjs 2020). UseState-koukku palauttaa listan, joka sisältää kaksi arvoa (Reactjs 2020). Toinen muuttuja sisältää tilan arvon ja toinen muuttuja on metodi, jonka avulla tilan arvoa muutetaan (Reactjs 2020). UseState-koukulle ensimmäinen annettu parametri määrittelee ensimmäisen muuttujan alustavan arvon (Reactjs 2020). Reactin mukana tulee monia erilaisia koukkuja. Toinen yleisesti käytetty koukku on useEffect-koukku, jonka avulla voidaan määritellä metodi, joka suoritetaan aina renderöinnin jälkeen (Reactjs 2019). UseEffect-koukulle voidaan antaa toinen parametri, joka kertoo sen, että halutaanko useEffect-koukussa määriteltyä koodia suorittaa silloin kun määritellyn muuttujan arvo muuttuu (Reactjs 2019).

2.2.5 Redux

Redux on kirjasto, jonka tarkoitus on huolehtia sovelluksen tilan hallinnasta ja päivityksestä. Se toimii globaalina varastona sovelluksen tiloille. Siinä tilaa voidaan päivittää toiminnoilla, jotka lähetetään toiminnan ottamien parametrien kanssa Redux-varastoon käsiteltäväksi. Tilan hallintaa tarvitaan monimutkaisia vaatimuksia omaavien sovelluksien toteuttamiseksi. Tilan avulla voidaan esimerkiksi tallentaa palvelimen vastaus tai muuta dataa muistiin. Redux on hyödyllinen sovelluksissa, joissa tilaa käytetään tai päivitetään monessa paikassa. Redux mahdollistaa sen, että Redux-tilan voi hakea tai sitä voi päivittää mistä tahansa JavaScript-tiedostosta. (Redux 2022.)

Reduxin varastossa olevaa tilaa ei saa koskaan muokata suoraan vaan sitä muokataan lähettämällä toimintoja varastoon, jotka sitten viedään Reduxin vähentäjälle, joka laskee uuden tilan edellisen tilan ja lähetetyn toiminnan perusteella (KUVA 4). Sitten kun vähentäjä on tehnyt muutoksen tilaan, se

huomauttaa siitä tilaajia, jonka jälkeen DOM päivittyy tilan muutoksen perusteella ja näytölle päivittyvät komponentit, jotka käyttävät kyseistä tilaa. Tilaaja voidaan määritellä Redux-kirjaston mukana tulevan useSelector-koukun avulla. Tilaaja hakee aina viimeisimmän arvon halutulle datalle kun komponentti renderöidään tai kun uusi toiminta lähetetään varastoon. Uuden toiminnan lähettäminen toimii dispatch-metodin avulla. Dispatch-metodin referenssi saadaan määriteltyä useDispatch-koukun avulla. Dispatch-metodi ottaa parametrina toiminnan, joka halutaan suorittaa. (Redux 2022.)



KUVA 4. Redux toimintamalli. (Tutorialspoint)

2.3 Node.js

Node.js on vuonna 2009 julkaistu avoimen lähdekoodin asynkroninen tapahtumalähtöinen ajonaikainen ympäristö. Se toimii V8 JavaScript-moottorilla, joka on yksi suosituimmista JavaScript moottoreista. Siitä kehitetään jatkuvasti uusia ja nopeampia versioita. Node.js mahdollistaa sen, että palvelinpuolta voidaan ohjelmoida JavaScriptillä. Tämä on hyödyllistä esimerkiksi full-stack-kehityksessä, koska ohjelmoijan ei ole pakko opetella täysin uutta ohjelmointikieltä palvelinpuolen

ohjelmointia varten, jos ohjelmoija osaa jo asiakaspuolen ohjelmointia JavaScriptillä. (Flaviocopes ja muut osallistujat 2022.)

Noden käyttämä Googlen V8-moottori, mahdollistaa JavaScript ajonaikaisen ympäristön sekä Noden tapahtumasilmukan. Noden tapahtumasilmukka toimii samalla tavalla kuin JavaScriptin tapahtumasilmukka selaimessa. Noden tapahtumasilmukka käyttää C-ohjelmointikielellä kirjoitettua libuv-kirjastoa. (Hiwarale 2018.)

Nodella on ei-estävä tapahtumaohjattu asynkroninen I/O-arkkitehtuuri. Tämä nimitys tulee siitä, että kun V8-moottori, tapahtumajono sekä tapahtumasilmukka toimivat kaikki yksittäisen säikeen päällä, työntekijäsäikeet pitävät huolen asynkronisuutta vaativista I/O-operaatioista. (Hiwarale 2018.)

Nodessa on estäviä sekä ei-estäviä operaatioita. Estäviä operaatiota suoritetaan synkronisesti ja ei-estäviä asynkronisesti. Estävät operaatiot ovat Nodessa operaatioita, joissa suoritetaan muuta ohjelmointikieltä kuin JavaScriptiä ja niiden suoritusta täytyy odottaa, koska Node ei voi jatkaa muita suorituksia kun estävää operaatiota suoritetaan. Yleisimmät estävät operaatiot, joita Noden kirjastossa on ovat sellaisissa kirjastoissa, jotka käyttävät libuv-kirjastoa. Nodessa kaikista estävistä järjestelmän kovalevyä tai verkkoa käyttävistä I/O-metodeista on olemassa asynkroniset ei-estävät versiot. Joillakin metodeilla Nodessa on myös estävä versio. Estävien ja ei-estävien metodien käyttäminen yhdessä voi luoda ongelmia. Nämä ongelmat voidaan ratkoa sillä tavalla, että määritellään takaisinkutsumetodi, jossa voidaan suorittaa konfliktin aiheuttamia metodeja sitten kun yksi konfliktin osapuolena oleva metodi on suoritettu. Tämä saa aikaan sen, että metodit suoritetaan oikeassa järjestyksessä. (Nodejs 2019.)

Node.js pystyy käsittelemään kymmeniä tuhansia yhteyksiä kerrallaan sen vuoksi, että ainoastaan välittömästi suoritettavat operaatiot lisätään pääsäikeen tapahtumajonoon ja muut aikaa vievät operaatiot lisätään toiselle säikeelle. Tämä tekee siitä hyvän valinnan esimerkiksi reaaliaikaisia arkkitehtuureja varten kuten chat-sovellukset, joissa on paljon sellaista liikennettä, joka vaatii vähän prosessointia. (Capan 2013.)

2.3.1 NPM

NPM eli Node Package Manager on standardi pakettienhallintatyökalu Node.js:lle. Sen avulla voidaan automatisoida uusien kirjastojen asennus, päivitys ja konfigurointi. Sen kirjastotarjonnassa on useita satoja tuhansia kirjastoja. NPM tarjoaa monia komentoja kuten “run”-komennon, jolla voidaan käyttää erilaisia komentosarjoja, joita on määritelty Node-projektin package.json tiedostossa. Komentosarjojen käyttö on hyödyllistä, koska niiden avulla voidaan määritellä hankalasti muistettavia komentoja. (Flaviocopes ja muut osallistujat 2021.)

2.3.2 Express

Express on vuonna 2010 julkaistu suosittu avoimen lähdekoodin Node.js-ohjelmistokehys, joka tarjoaa monia erilaisia ominaisuuksia. Expressin avulla voidaan käsitellä erilaisia HTTP-pyyntöjä erilaisiin URL-osoitteisiin ja se tekee palvelinpuolen reitityksestä helppoa ja nopeaa. Expressiä käytetään monen muun suositun Node.js-ohjelmistokehyksen perustana. (MDN contributors 2022.)

Muihin Expressin ominaisuuksiin kuuluu muun muassa eri osoitteisiin tulevien pyyntöjen käsittely, mahdollisuus käsitellä pyyntöjä väliohjelmistoilla sekä staattisten tiedostojen tarjoaminen. Väliohjelmistot ovat pyyntöjen lisäprosessointitapa. Ne voidaan asettaa käsittelemään jokainen pyyntö tai yksittäisiä pyyntöjä. Expressissä pyyntöjen käsittelyssä tulleita virheitä voidaan käsitellä vapaaehtoisesti virheidenkäsittelyyn tarkoitetulla väliohjelmistolla. (MDN contributors 2022.)

3 FULL-STACK SOSIAALINEN MEDIA -SOVELLUKSEN TOTEUTUS

Opinnäytetyön tavoitteena on rakentaa sosiaalinen media full-stack-sovellus. Sovellukseen siis kehitetään frontend- ja backend-puoli. Frontend-puolella sovellukseen rakennetaan käyttöliittymä hyödyntäen muun muassa HTML, CSS, React, JavaScript sekä useita eri kirjastoja. Backend-puolella sovellukseen suunnitellaan relaatiotietokantayhteydet, REST API -osoitteet ja niihin tulevien pyyntöjen käsittelijät. Backend-puolella hyödynnetään JavaScript, Node.js, Express.js ja monia erilaisia kirjastoja. HTTP REST API -osoitteet kehitetään muun muassa käyttäjiin, postauksiin, kommentteihin sekä yhteisöihin liittyviin pyyntöihin. Sovelluksessa käytetään myös WebSocket-protokollaa esimerkiksi postauksien hakemiseen ja niiden äänestämiseen HTTP REST API -rajapinnan sijaan. Projektin lähdekoodi on avoimesti saatavilla Github-verkkosivustolta (LIITE 1).

3.1 Frontend-puolen projektin aloitus

Tässä osioissa käydään läpi frontend-puolen projektin aloittamista. Osiossa kerrotaan muun muassa komennoista, joilla projekti alustetaan sekä myös frontend-puolen projektin rakenteesta. Osiossa esitellään myös frontend-puolen käyttöliittymän suunnittelua ja kehitystä. Osiossa myös luodaan frontend-puolen globaali varasto sovelluksen tiloille. Globaali varasto toteutetaan hyödyntämällä react-redux-kirjastoa.

3.1.1 Frontend-puolen alustus

Projektin frontend-puoli alustetaan “create-react-app"-komennolla (KUVA 5). Se konfiguroi automaattisesti kehitysympäristön niin, että sovelluskehittäjän ei tarvitse itse tehdä muuta kuin aloittaa kirjoittamaan koodia. Sen mukana tulee valmiit komennot muun muassa sovelluksen käynnistämiseen, tuotantoversion rakentamiseen sekä testaukseen. Se käyttää Babelia sekä Webpackia konfigurointiin (Reactjs 2021a). Babel on työkalu, jolla voidaan kääntää esimerkiksi uudempia ECMAScript-versioita tai JSX-syntaksia selaimille yhteensopivaan JavaScript-versioon (Babeljs). Webpack on työkalu jonka avulla voidaan niputtaa JavaScript-tiedostoja yhteen tiedostoon, jota voidaan käyttää selaimessa (Freecodecamp 2018).

```
npx create-react-app my-app
cd my-app
npm start
```

KUVA 5. React-sovelluksen alustamiseen sekä sovelluksen käynnistämiseen tarkoitetut komennot. (Reactjs 2021)

Kuvassa 6 on kuvailtu projektin asiakaspuolen hakemistojen rakennetta. Siinä on eroteltu eri kategorioihin kuuluvat tiedostot toisistaan, niin että tiettyyn kategoriaan liittyvä tiedosto löytyy kyseisen tiedoston kategorian hakemistosta.

```
frontend/
├─ node_modules/
├─ public/
│  ├─ favicon.ico
│  ├─ index.html
├─ src/
│  ├─ store.js
│  ├─ components/
│  │  ├─ components...
│  ├─ hooks/
│  │  ├─ hooks...
│  ├─ pages/
│  │  ├─ pages...
│  ├─ slices/
│  │  ├─ slices...
│  ├─ styles/
│  │  ├─ styles...
│  ├─ utils/
│  │  ├─ utils...
│  ├─ websockets/
│  │  ├─ websockets...
│  ├─ App.js
│  ├─ index.js
├─ package.json
├─ package-lock.json
├─ postcss.config.js
├─ tailwind.config.js
├─ README.md
```

KUVA 6. Frontend-puolen hakemistojen rakenne.

3.1.2 TailwindCSS

TailwindCSS on kirjasto, jonka avulla voidaan käyttää valmiiksi määriteltyjä CSS-luokkia. Se konfiguroidaan tähän sovellukseen PostCSS-lisäosana. Toinen PostCSS-lisäosa sovelluksessa on autoprefixer-lisäosa, joka käsittelee CSS-koodin automaattisesti niin, että siihen lisätään eri selaimille yhteensopivia CSS-muuttujia (KUVA 10).

TailwindCSS:n konfigurointia varten luodaan postcss.config.js tiedosto ja siihen lisätään kaikki PostCSS:n käyttämät lisäosat (KUVA 7). TailwindCSS:ää varten luodaan myös tailwind.config.js-tiedosto, johon voidaan lisätä erilaisia asetuksia tailwindCSS:lle kuten lisäosia, teemoja tai tiedostoja joista etsitään käyttämättömiä tailwindCSS-luokkia, jotka poistetaan projektista tilan säästämiseksi (KUVA 8).

```
JS postcss.config.js ×
JS postcss.config.js > ...
1  module.exports = {
2    plugins: {
3      tailwindcss: {},
4      autoprefixer: {},
5    },
6  }
7
```

KUVA 7. Postcss.config.js tiedosto, jossa määritellään PostCSS:n lisäosat.

```
JS tailwind.config.js M ×
JS tailwind.config.js > [@] <unknown> > variants
1  module.exports = {
2    mode: "jit",
3    purge: [
4      "./src/**/*.{js,ts,jsx,tsx}",
5      "./src/components/**/*.{js,ts,jsx,tsx}",
6    ],
7    theme: {
8      extend: {
9        colors: {},
10     },
11   },
12   variants: {
13     extend: {},
14   },
15   plugins: [require("@tailwindcss/line-clamp")],
16 };
17
```

KUVA 8. Tailwind.config.js tiedosto, jossa määritellään erilaisia asetuksia kuten lisäosia, teemoja tai tiedostoja joista etsitään käyttämättömiä tailwindCSS-luokkia poistamista varten.

PostCSS käsittelee kirjoitetun CSS:n riippuen sille annetuista lisäosista. Autoprefixer-lisäosa lisää CSS:ään automaattisesti vaaditut etuliitteet käytetyille CSS-muuttujille niin, että ne toimivat kaikissa selaimissa (KUVA 9).

```
.example {
    display: grid;
}

// ennen käsittelyä
```

```
.example {
    display: -ms-grid;
    display: grid;
}

// käsittelyn jälkeen
```

KUVA 9. CSS-koodi PostCSS:n ja sen Autoprefixer-lisäosan käsittelyä ennen ja sen jälkeen.

TailwindCSS:n toimivaksi saatavaksi lisätään sovelluksen pää CSS–tiedostoon “@tailwind”-koodirivit (KUVA 10). Kyseinen CSS-tiedosto tulee sisällyttää sovelluksen pääkomponentin tiedostossa, joka on tässä sovelluksessa App.js.

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

KUVA 10. Vaaditut tailwindCSS-koodirivit pää CSS-tiedostossa. (TailwindCSS)

TailwindCSS mahdollistaa valmiiksi määriteltyjen CSS-luokkien käytön. CSS-luokkia voidaan antaa HTML-elementtien “className”-ominaisuuden arvoksi. Valmiiksi määriteltyjen CSS-luokkien käyttö tekee käyttöliittymän kehittämisestä nopeaa, koska samoja CSS-luokkia voidaan käyttää kaikkien HTML-elementtien muotoiluun, joten ohjelmoijan ei tarvitse luoda erillistä CSS-luokkaa jokaiselle HTML-elementille. Sen avulla sovelluksista voi tehdä myös muun muassa selaimen ikkunan leveyteen reagoivia niin, että sovelluksen käyttöliittymä toimii sekä mobiililaitteilla että isompinäyttöisillä tietokoneilla. Kuvassa 11 on esimerkki siitä miten div-elementin leveys muuttuu koko ympäröivän elementin leveydestä tuhannen pikselin leveyteen kun ruudun leveys kasvaa tietyn leveäksi.

```
<div className="w-full md:w-[1000px]" ></div>
```

KUVA 11. Esimerkki TailwindCSS-luokkien käytöstä, kun halutaan HTML-elementin olevan koko ympäröivän elementin levyinen mobiililaitteilla ja 1000 pikselin levyinen keskikokoisilla näytöillä.

Tässä projektissa hyödynnettiin TailwindCSS:n keskeytyskohtia, eli TailwindCSS:n määrittelemiä kohtia joissa voi muuttaa CSS-luokkien muuttujien arvoja kun näytön koko muuttuu. Kuvassa 12 on esimerkki siitä miten sovellus on sopeutunut siihen, että näytön suuruus on vain 326 pikseliä. Kuvassa 13 on näkyy sovelluksen ulkoasu kun näytön leveys on 1145 pikseliä. Kuvista huomaa, että kun näytön leveys kapenee, HTML-elementtien leveys pienenee tai niiden järjestys muuttuu ja jotkin HTML-elementit piilotetaan, jotta postaukset näkyisivät selkeästi käyttäjälle.

Dimensions: Responsive ▼ 326 × 1066 75% ▼ No throttlin

SOCIAL MEDIA

Search communities

Log In Sign Up

Loremipsum

posted by Anpes99 23 days ago

Lorem ipsum dolor, sit amet consectetur adipisicing elit.

27

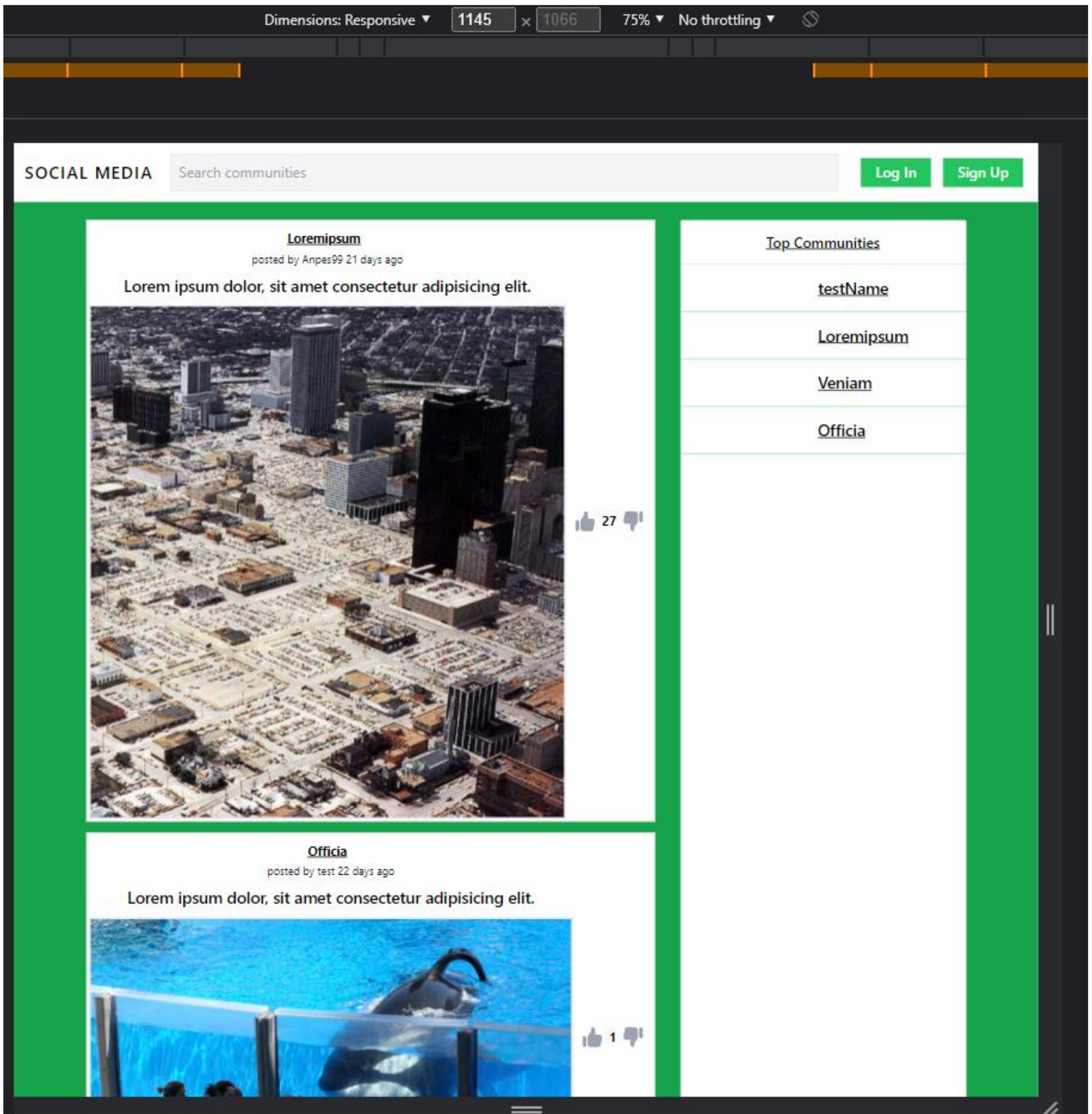
Officia

posted by test 24 days ago

Lorem ipsum dolor, sit amet consectetur adipisicing elit.

1

KUVA 12. Sovelluksen etusivun ulkoasu, kun näytön leveys on 326 pikseliä. HTML-elementit ovat sopeutuneet näytön leveyden muutoksiin, koska niiden CSS-luokkien arvot ovat muuttuneet näytön leveyden muutoksen seurauksena.



KUVA 13. Sovelluksen etusivun ulkoasu, kun näytön leveys on 1145 pikseliä.

3.1.3 React-redux

Frontend-puolella sovellus tarvitsi keinon hallita globaalia tilaa muun muassa käyttäjän tietoja, kirjautumis-/rekisteröitymislomaketta sekä nykyisen yhteisön seuraamista varten. Globaalin tilan avulla esimerkiksi käyttäjän tiedot voidaan hakea localStoragesta ainoastaan yhdessä komponentissa, joka lähettää käyttäjän tiedot globaaliin varastoon, josta sitten kaikki komponentit voivat käyttää kyseisiä tietoja ilman, että jokaisen komponentin pitäisi yksitellen hakea käyttäjän tiedot localStoragesta. Tämä vähentää tarvittavan koodin määrää. Myös aina kun globaaliin tilaan tehdään muutos, kaikki komponentit, jotka käyttävät kyseistä tilaa päivittyvät näytölle tilan muutoksen perusteella. Globaalin tilan mahdollistamiseksi sovelluksessa hyödynnettiin suosittua react-redux-kirjastoa. Tässä projektissa hyödynnettiin myös redux-toolkit-kirjastoa, joka tekee Reduxista selkeämpää ja vähentää kirjoitettavan koodin määrää.

Jotta Redux saadaan toimimaan tulee sovellukselle luoda ensin vähentäjä niin, että sen avulla voidaan sitten luoda Redux-varasto. Redux-siivu on yksi osa varastoa ja niitä voi lisätä varastoon niin monta kuin on tarve (Redux-toolkit 2021). Siivulle voidaan määritellä monta eri tilaa sekä monia toimintoja. Toiminta on metodi, jolla Redux-tilaa voidaan muuttaa (Redux-toolkit 2021). Redux-tilaa ei saa muuttaa suoraan, mutta kun käytetään Redux-toolkitin createSlice-metodia, se on näennäisesti mahdollista, koska se käyttää Immer-kirjastoa (Redux-toolkit 2021.).

Immer-kirjasto toimii niin, että se luo nykyisestä tilasta väliaikaisen piirustuksen, jota voi muokata suoraan ja sitten kun muokkaus on valmis, Immer muuttaa Redux-tilaa oikeaoppisesti piirustuksen perusteella. (Redux-toolkit 2022.)

Eri globaaleja tiloja, joita sovellukseen tarvitaan, ovat kirjautuneen käyttäjän tila, kirjautumis- ja rekistöintiponnahdusikkunoiden näkyvyystilat sekä käyttäjän sen hetkisen sivun yhteisön nimi. Näiden tilojen muuttamiseen luodaan jokaiselle tilalle oma toiminta (KUVA 14). Toimintojen avulla voidaan päivittää globaaleja tiloja.

```
JS appSlice.js ●
src > slices > JS appSlice.js > ...

1   import { createSlice } from "@reduxjs/toolkit";
2
3   const initialState = {
4     user: null,
5     loginVisible: false,
6     signUpVisible: false,
7     currentCommunity: null,
8   };
9
10  export const appSlice = createSlice({
11    name: "app",
12    initialState,
13    reducers: {
14      setUser: (state, action) => {
15        state.user = action.payload;
16      },
17      setLoginVisible: (state, action) => {
18        state.loginVisible = action.payload;
19      },
20      setSignUpVisible: (state, action) => {
21        state.signUpVisible = action.payload;
22      },
23      setCurrentCommunity: (state, action) => {
24        state.currentCommunity = action.payload;
25      },
26    },
27  });
28  export const {
29    setUser,
30    setLoginVisible,
31    setSignUpVisible,
32    setCurrentCommunity,
33  } = appSlice.actions;
34
35  export default appSlice.reducer;
36
```

KUVA 14. Redux-siivujen, toimintojen sekä vähentäjien määrittely appSlice.js-tiedostossa.

Store.js tiedostossa luodaan Redux-varasto ja app-siivu reduxjs/toolkit-kirjastosta tuodulla configureStore-metodilla, jolle syötetään olio, jonka reducer-kentän arvona on olio, joka määrittelee eri siivut varastossa (KUVA 15). App-siivu määritellään reducer-kentän arvona olevan olion kenttänä, jonka avain on ”app” ja sen arvo on kuvassa 14 luotu vähentäjä. Alustettua varastoa käytetään sovelluksen App-komponentissa, jossa sitä tarjotaan kaikille App-komponentin lapsikomponenteille.

```
JS store.js ×
src > app > JS store.js > ...
1  import { configureStore } from "@reduxjs/toolkit";
2  import appReducer from "../slices/appSlice";
3
4  export const store = configureStore({
5    reducer: {
6      app: appReducer,
7    },
8  });
9
```

KUVA 15. Redux-varaston luominen store.js tiedostossa configureStore-metodin avulla. Varastoon määritellään app-siivu appSlice.js-tiedostosta tuodun vähentäjän avulla.

3.2 App.js

Tässä osiossa käydään läpi App.js tiedostossa määriteltyä App-komponenttia, jossa tapahtuu useita tärkeitä asioita sovelluksen toiminnan kannalta. Tässä sovelluksessa kaikki muut komponentit ovat tämän komponentin lapsikomponentteja. App-komponentti renderöidään index.js tiedostossa ReactDOM.render-metodilla. Tiedostossa muun muassa reititetään sovelluksen eri osoitteet, haetaan

kirjautunut käyttäjä ja tarjotaan Redux-varastoa lapsikomponenteille. Nämä toiminnot mahdollistetaan tuomalla tiedostoon erilaisia komponentteja asennetuista kirjastoista sekä muista tiedostoista.

3.2.1 React-router-dom

React-sovelluksissa eri osoitteiden reitittämiseen tarvitaan erillinen kirjasto. Tässä projektissa käytetään reititykseen react-router-dom-kirjastoa, joka on suosituin kirjasto reitityksen toteuttamiseen React-sovelluksissa. React-router-domissa lapsikomponentit kääritään Router-komponentin sisälle, joka sitten osaa renderöidä oikean HTML:n riippuen siitä missä osoitteessa käyttäjä on. Erilliset osoitteet määritellään Route-komponenteilla, joille syötetään element-kentän arvoksi haluttu HTML-koodi tai komponentti, joka määrittelee sen mitä kyseinen osoite näyttää käyttäjälle (KUVA 16). Route-komponentin path-kentän arvoksi asetetaan merkkijono, joka määrittelee missä osoitteessa kyseinen Route-komponentti toimii.

```
<Router>
  <Provider store={store}>
    <Routes>
      <Route
        path="/community/:communityName/submit"
        element={<SubmitNewPostPage />}
      />
      <Route
        path="/community/:communityName/post/:postId/:postTitle"
        element={<PostPage />}
      />
      <Route
        path="/community/:communityName"
        element={<CommunityPage />}
      />
      <Route path="/" element={<HomePage />} />
    </Routes>{" "}
  </Provider>
</Router>
```

KUVA 16. Sovelluksen reitityksen määrittely App.js tiedostossa react-router-dom-kirjastosta tuotujen komponenttien avulla. Sovelluksen eri osoitteet ja niiden komponentit määritellään Route-komponenteissa. Route-komponentit asetetaan Router-komponentin sisäpuolelle.

3.2.2 Redux-varasto

Redux-varasto tuodaan App.js-tiedostoon store.js-tiedostosta ja sitä tarjotaan lapsikomponenttien käyttöön react-redux-kirjastosta tuodulla Provider-komponentilla (KUVA 17). Provider-komponentti tarjoaa varastoa kaikille sen alapuolella oleville komponenteille. Kun Redux-varastoa tarjoillaan kaikille lapsikomponenteille, ne pääsevät käyttämään varastossa olevia globaaleja tiloja.

```
<Provider store={store}>
```

KUVA 17. Provider-komponentti, jolle syötetään store-kentän arvoksi Redux-varasto, jota Provider-komponentti tarjoaa sen alla oleville komponenteille.

App.js-tiedostossa käytetään tässä sovelluksessa useEffect-koukkua, joka hakee localStoragesta kirjautuneen käyttäjän tiedot ja sitten löydetty tiedosto lähetetään vähentäjille käsiteltäväksi setUser-toiminnan ja Redux-varaston dispatch-metodin avulla. UseEffect-koukussa määritelty metodi suoritetaan ainoastaan kerran ensimmäisen renderöinnin jälkeen siitä syystä, että sille on annettu toinen parametri, joka on tyhjä lista eli sillä ei ole muuttujia jotka voisivat muuttua joten kyseinen metodi suoritetaan vain kerran.

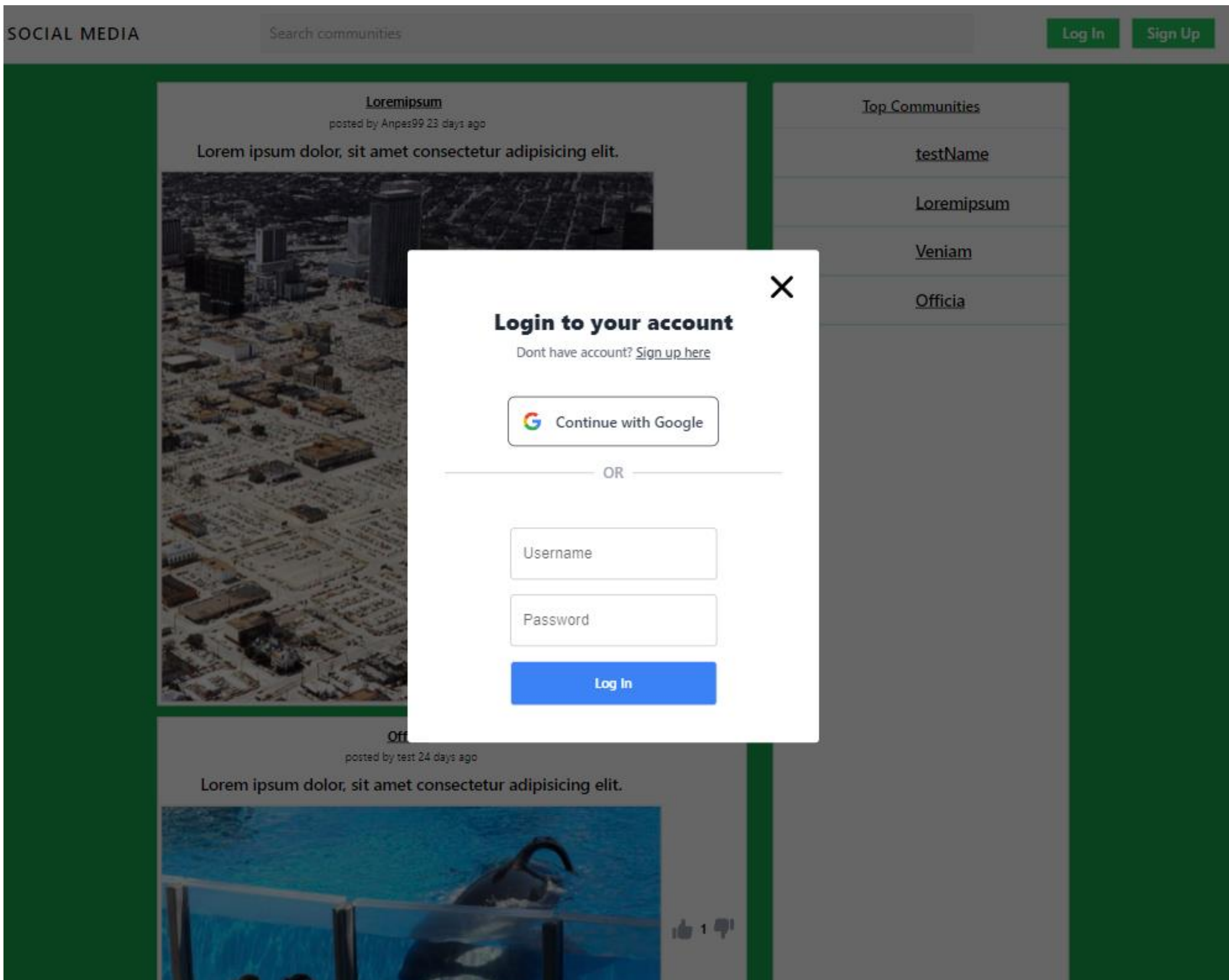
3.3 Käyttäjän kirjautuminen

Käyttäjätilinsä luotuaan sovelluksen rekisteröintilomakkeella voi käyttäjä kirjautua sisään sovellukseen ja alkaa käyttämään kirjautuneille käyttäjille tarkoitettuja ominaisuuksia kuten postauksien äänestäminen, kommentointi ja omien postauksien luominen.

Kirjautumislomakkeelle annetaan käyttäjän käyttäjätunnus ja salasana, jotka lähetetään backend-puolelle käsiteltäväksi. Backend-puolen kirjautumisesta huolehtiva REST API -rajapinta ottaa vastaan lähetetyt tiedot ja sitten se hakee tietokannasta käyttäjän, jolla on kyseinen käyttäjätunnus. Jos käyttäjä löytyy ja käyttäjätunnus sekä salasana ovat oikein niin backend luo JWT-tokenin kirjautumista varten. JWT-tokenia (JSON Web Token) käytetään token-perusteisessa kirjautumisessa ja se on suosittu menetelmä pitää käyttäjän kirjautuminen muistissa sekä antaa aikaraja, jonka jälkeen käyttäjän pitää kirjautua sisään uudelleen.

Token-perusteinen kirjautuminen toimii niin, että frontend vastaanottaa tokenin onnistuneen kirjautumisen jälkeen ja sitten se varastoidaan localStorageen, josta se haetaan aina kun sivu päivitetään. Tokenin voimassaolo tarkistetaan aina kun se liitetään sellaisten pyyntöjen mukaan, jotka vaativat kirjautumisen. Tällaisia pyyntöjä ovat esimerkiksi henkilökohtaisten tietojen hakeminen tai uuden postauksen luominen. Jos token ei ole enää voimassa, backend lähettää vastauksena virheen ja frontend poistaa vanhentuneen tokenin localStoragesta ja käyttäjän pitää kirjautua sisään uudelleen halutessaan suorittaa kirjautumisen vaativia toimintoja.

Kirjautumis- ja rekisteröintilomake näyttävät tässä sovelluksessa samalta ja ne näytetään ruudulla samalla menetelmällä. Molemmille lomakkeille luotiin omat komponenttinsa ja sitten ne lisättiin Header-komponenttiin, jotta niitä on mahdollista käyttää jokaiselta sivulta, koska Header-komponentti on näkyvissä jokaisella tämän sovelluksen sivulla. Lomakkeiden näyttäminen toimii Redux-tilojen avulla niin, että kun käyttäjä painaa nappia kirjautuakseen sisään niin napille annettu tapahtumankäsittelijä lähettää Reduxin vähentäjille toiminnan, joka asettaa kirjautumislomakkeen tilan näkyväksi ja saa siten tilan muutoksen seurauksena lomakkeen renderöitymään ruudulle (KUVA 18).



KUVA 18. Sovelluksen kirjautumislomake, jota voidaan käyttää jokaisella sovelluksen sivulla.

3.4 Postauksien selaaminen

Postauksien selaamista varten rakennettiin PostFeed-komponentti, joka näyttää listan postauksia. Yksittäisille postauksille luotiin myös PostItem-komponentti. Sovellukseen rakennettiin myös muokattu koukku postauksien palvelinpuolelta hakemista varten. Kyseinen koukku palauttaa metodin, jolla voi hakea lisää postauksia. Se palauttaa myös erilaisia muuttujia, jotka kertovat koukun eri tiloista. Koukun tilat mahdollistetaan useState-koukulla, joka palauttaa tilan arvon sisältävän muuttujan sekä metodin, jolla voi muokata tilaa. Muokatulla koukulla postauksien hakeminen toimii niin, että kun haetaan lisää postauksia niin se ensin asettaa loading-tilan todeksi ja sitten se käyttää WebSocket-protokollaa postauksien hakemiseksi backend-puolelta ja sitten kun postaukset on haettu,

se asettaa loading-tilan epätodeksi ja sitten yhdistää uudet haetut postaukset muokatun koukun postaukset-tilaan. Postaukset-tilan kautta koukkua käyttävä tiedosto saa uudet postaukset käyttöönsä ja kun postaukset-tila päivittyy, myös uudet postaukset renderöidään näytölle.

3.4.1 Vain tarvittavien postauksien hakeminen

Postauksien hakemisessa liian suuria hitaita yksittäisiä HTTP-pyyntöjä voidaan vähentää tekemällä monia pienempiä WebSocket-pyyntöjä aina kun viimeinen postaus tulee näkyviin käyttäjän ruudulle. Sovelluksessa määritellään metodi, joka käyttää IntersectionObserver API:a, joka seuraa viimeisen postauksen sijaintia (KUVA 19). Kun seurattu postaus tulee näytölle, IntersectionObserver API:lle syötetty metodi hakee lisää postauksia. Kun postauksia on haettu lisää, postausten renderöinti suoritetaan uudelleen ja postausten listan uutta viimeistä postausta aletaan seurata. Postauksen seuraaminen tapahtuu syöttämällä postauksen elementin ref-kentän arvoksi metodi, joka seuraa kyseistä elementtiä (KUVA 20).

```
const observer = useRef();
const lastPostRef = useCallback((node) => {
  if (loading) return;
  if (observer.current) observer.current.disconnect();
  observer.current = new IntersectionObserver((entries) => {
    if (entries[0].isIntersecting) {
      fetchMorePosts(orderType);
    }
  });
  if (node) observer.current.observe(node);
});
```

KUVA 19. Viimeisen postauksen ref-kentän metodin määrittely. Metodissa IntersectionObserver API hakee lisää postauksia, jos sen tarkastelema elementti on näytöllä. Metodi määritellään useCallback-koukun avulla, joka tallentaa metodin muistiin niin, että sitä ei luoda aina uudelleen kun sen vanhempikomponentti renderöityy uudelleen.

```
<div>
        {posts.map((post, i) => {
          if (i + 1 === posts.length) {
            return (
              <div key={post.id} ref={lastPostRef}>
                <PostFeedItem post={post} />
              </div>
            );
          } else return <PostFeedItem post={post} key={post.id} />;
        })}
</div>
```

KUVA 20. Postausten elementit määritellään JSX-syntaksia sekä posts-listan map-metodia hyödyntäen. Map-metodi sijoittaa postaukset kuvassa näkyvien div-elementtien väliin järjestyksessä. Viimeiselle postaukselle annetaan ref-kentän arvoksi metodi, jolla seurataan milloin se tulee näkyviin näytölle.

3.4.2 Muokattu koukku ja WebSocket

Postauksien hakeminen eristettiin omaan koukkuunsa. Tämä tekee koodista siistimpää ja tekee sen jatkokehityksestä helpompaa. Postauksien hakemiseen tarkoitettua koukkua on kuvailtu kuvissa 21 ja 22. Kuvassa 21 näkyy koukun parametrien, nimen sekä sen eri tilojen määrittely useState-koukkujen avulla. Kuvassa 22 näkyy WebSocket-viesti, jolla koukku hakee backend-puolelta lisää postauksia. WebSocket-viestille syötetään erilaisia arvoja. Viimeinen arvo, joka sille syötetään on takaisinkutsumetodi, joka suoritetaan frontend-puolella ainoastaan silloin kun sitä kutsutaan backend-puolelta kyseisen WebSocket-viestin käsittelijässä. Backend-puoli voi tämän takaisinkutsu-metodin avulla lähettää dataa vastauksena WebSocket-viestiin. Tätä hyödynnettiin sovelluksessa niin, että backend-puoli lähettää tietokannasta haetut postaukset sekä niistä tarvittua tietoa takaisinkutsu-metodin avulla (KUVA 24).

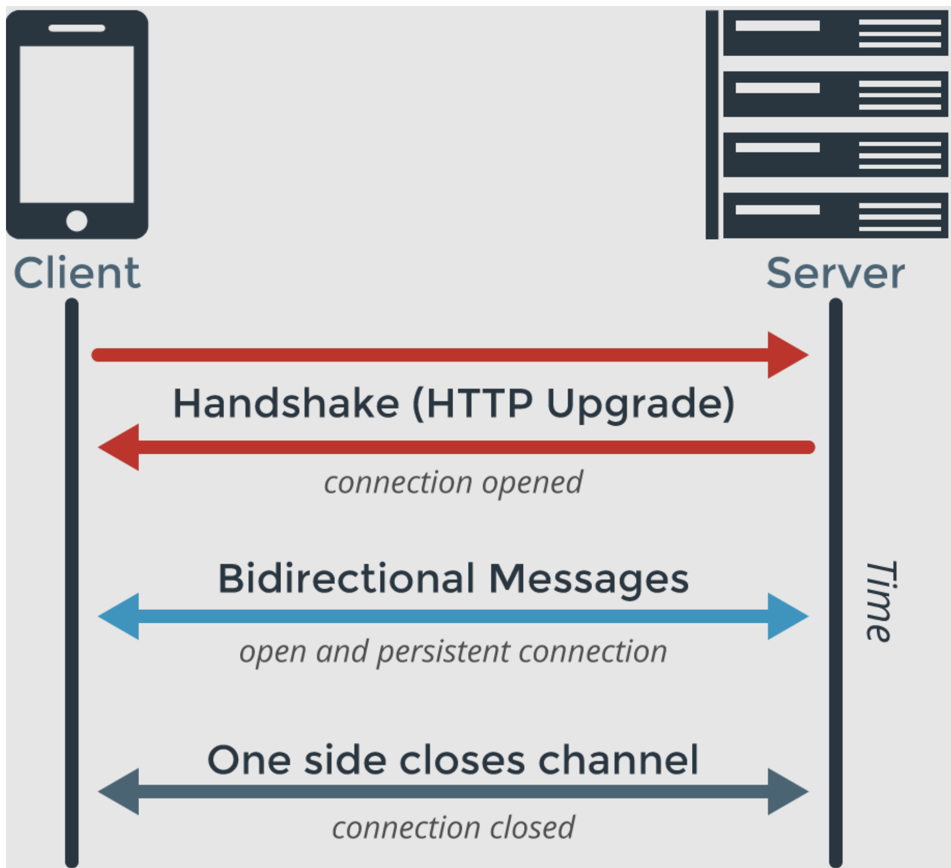
```
const useGetPosts = (queryParamSort, communityId, orderType) => {
  const [posts, setPosts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [totalPostCount, setTotalPostCount] = useState(true);
  const [error, setError] = useState(false);
```

KUVA 21. Muokattu koukku postauksien hakemista varten. Koukun eri tiloja on määritelty useState-koukkujen avulla.

```
setLoading(true);
socket.emit(
  "fetch_more_posts",
  order,
  sortBy,
  posts.length,
  5, //limit
  communityId,
  (data) => {
    console.log("socket data ", data);
    setPosts([...posts, ...data.posts]);
    setTotalPostCount(data.totalCount);
    setLoading(false);
  }
);
```

KUVA 22. Muokatun koukun WebSocket-viesti, jolla se hakee lisää postauksia backend-puolelta. WebSocket-viestille annetaan erilaisia parametreja, jotka kuvailevat miten lisäpostauksia pitää hakea.

Postauksien hakemiseen tarvittujen pyyntöjen korkean määrän takia sovelluksessa käytetään WebSocket-protokollaa. WebSocket-protokollan toimintaperiaatetta on kuvailtu kuvassa 23. WebSocket-protokolla toimii HTTP-protokollaan verrattuna niin, että se ei sulje luotua yhteyttä ennen kuin toinen yhteyden osapuolista päättää sulkea sen (ArpitAsati 2022). Pyydettäessä lisää postauksia palvelinpuolelta ei ole tarvetta avata uutta yhteyttä.



KUVA 23. WebSocket-protokollan toimintaperiaate. (Gamage 2017)

```
io.on("connection", (socket) => {
  socket.on(
    "fetch_more_posts",
    async (order, sortBy, offset, limit, communityId, cb) => {
      const totalCountOptions = {};
      if (communityId)
        totalCountOptions.where = { communityId: Number(communityId) };
      const totalCount = await Post.count(totalCountOptions);

      const options = {
        order: [["createdAt", "DESC"]],
        offset,
        limit,
        include: [
          { model: User, attributes: ["username"] },
          { model: Community, attributes: ["name"] },
        ],
      };
      if (communityId) options.where = { communityId: Number(communityId) };
      if (order && sortBy) options.order = [[sortBy, order]];
      const posts = await Post.findAll(options);
      cb({ posts, totalCount });
    }
  );
```

KUVA 24. Backend-puolen lisäpostauksien hakemiseen tarkoitettujen WebSocket-viestien käsittelijä, joka ottaa vastaan parametreja joiden perusteella se hakee postauksia tietokannasta ja sitten lähettää dataa takaisin viestin lähettäjälle takaisinkutsumetodin avulla.

3.5 Postauksien äänestäminen

Postauksien äänestämisessä hyödynnettiin myös WebSocket-protokollaa. Äänestäminen toteutettiin sovellukseen niin, että kun käyttäjä äänestää postausta, siitä ilmoitetaan jokaiselle käyttäjälle ja kyseisen äänen saaneen postauksen äänet päivitetään kaikilla sovelluksen käyttäjillä. Sovelluksen äänestämisjärjestelmä toimii niin, että käyttäjä voi äänestää postausta ja sitten myös peruuttaa äänestyksen. Kun käyttäjä lähettää äänestyksen ja saa backendiltä vastauksen siihen, globaalissa Redux-varastossa olevan käyttäjän tiedot päivitetään niin, että postauksen id ja annettu ääni lisätään käyttäjän äänestämien postauksien listalle. Äänen antamisen jälkeen käyttäjä voi vain perua äänen tai antaa vastakkaisen äänen postaukselle.

Backend-puolella äänestys toimii niin, että äänet pidetään tallessa monen-suhde-moneen-yhteyksien avulla relaatiotietokannassa, josta ne sitten joko poistetaan tai niitä päivitetään, jos käyttäjä muuttaa ääntään.

3.6 Backend-puolen rakenne, tietokanta ja HTTP REST API -rajapinnat

Palvelinpuolen rakennetta on kuvailtu kuvassa 25. Siinä on selkeästi organisoitu toisiinsa liittyvät asiat omiin hakemistoihinsa ja tiedostoihinsa. Controllers-hakemisto sisältää HTTP REST API -osoitteiden käsittelijät. Models-hakemistossa määritellään tietokantamallit ja niiden yhteydet. Utils-hakemistossa sijaitsee muun muassa väliohjelmistot, konfiguraatioita, tietokannan alustaminen sekä sekalaisia apumetodeja. Websocket-hakemistossa on määritelty WebSocket-viestien käsitteleminen ja niiden lähettäminen.

```
backend/
├─ node_modules/
├─ store.js
├─ controllers/
│   ├─ comments.js
│   ├─ login.js
│   ├─ posts.js
│   ├─ communities.js
│   ├─ users.js
│   ├─ s3.js
├─ models/
│   ├─ Comment.js
│   ├─ Community.js
│   ├─ index.js
│   ├─ Post.js
│   ├─ User.js
│   ├─ UserRatedPosts.js
│   ├─ UserCommunities.js
├─ utils/
│   ├─ config.js
│   ├─ db.js
│   ├─ middleware.js
│   ├─ s3.js
│   ├─ utils.js
├─ websockets/
│   ├─ likes.js
│   ├─ posts.js
├─ .env
├─ app.js
├─ index.js
├─ package.json
├─ package-lock.json
├─ README.md
```

KUVA 25. Backend-puolen hakemistojen rakenne.

3.6.1 Relaatiotietokannan toteutus

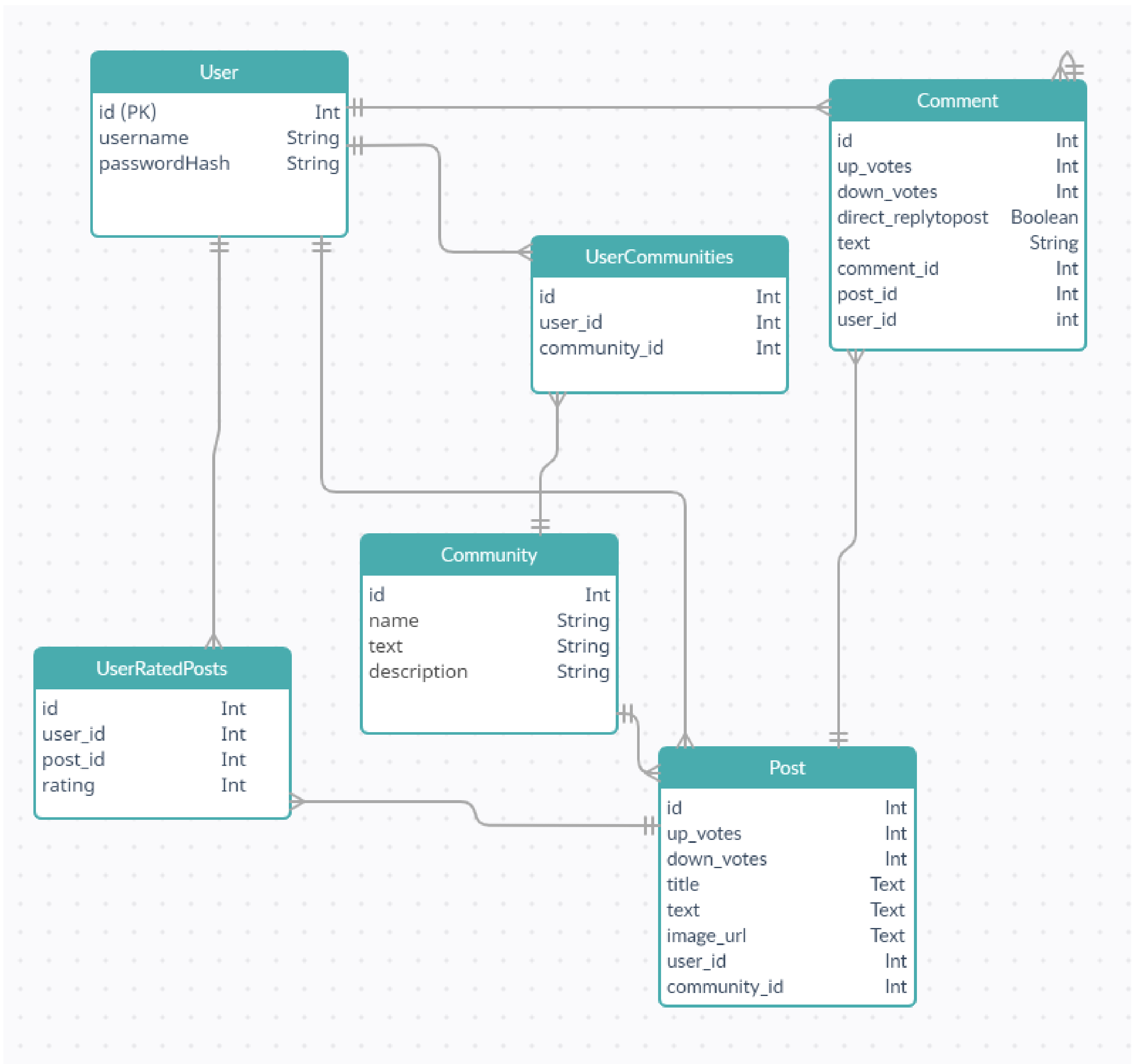
Sovelluksen tietokantana käytetään relaatiotietokantaa, koska sovellus tarvitsee monen-suhde-moneen-yhteyksiä, että se voi yhdistää käyttäjät heidän äänestämiin postauksiin sekä yhteisöihin, joihin he ovat liittyneet mukaan. Tietokantana projektissa hyödynnettiin Heroku-pilvipalvelualustan tarjoamaa ilmaista PostgreSQL-tietokantaa. Herokun kautta relaatiotietokannan saa alustettua kun ensin luodaan

heroku-projekti ja sitten siihen lisätään PostgreSQL-tietokanta lisäosana (KUVA 26). Tietokannan osoite lisätään backend-puolen ympäristömuuttujiin, jotta sitä voidaan käyttää koodissa niin, että sovelluksen lähdekoodia katselevat henkilöt eivät näe tämän sovelluksen tietokantaosoitetta.

```
heroku create
heroku addons:create heroku-postgresql:hobby-dev
heroku config
```

KUVA 26. Komennot heroku-projektin luomiseen, PostgreSQL-lisäosan lisäämiseen sekä PostgreSQL-tietokannan tietojen hakemiseen. (Luukkainen 2022)

Sovelluksen relaatiotietokannalle suunniteltiin relaatiotietokantakaavio, johon lisättiin taulut muun muassa käyttäjille, yhteisöille, postauksille, kommenteille sekä liitostaulut monen-suhde-moneen-yhteyksiä varten (KUVA 27).



KUVA 27. Sovelluksen tietokannan kaavio.

Tietokantataulut sekä tietokannasta hakeminen toteutettiin Sequelize-kirjastolla. Sequelize on ORM-työkalu, erilaisille relaatiotietokannanhallintajärjestelmille. ORM eli Object-Relational Mapping tarkoittaa tapaa, jolla muutetaan dataa relaatiotietokannoista koodissa määriteltyihin olio-malleihin (Visual-paradigm 2018). Olio-mallit kuvailevat tietokannassa olevaa dataa ja niiden avulla voidaan käyttää tietokantaa ilman, että ohjelmoijan pitäisi kirjoittaa relaatiotietokantojen käyttöön tarkoitettua SQL-kyselykieltä (Visual-paradigm 2018). Sequelize-kirjastoa käytettäessä määriteltiin jokaiselle tietokantataululle malli ja sitten kaikki mallit tuotiin yhteiseen tiedostoon nimeltä index.js niiden omista tiedostoista. Index.js tiedostossa määriteltiin yhden-suhde-moneen sekä monen-suhde-moneen-

yhteydet (KUVA 28). Index.js on tässä sovelluksessa se tiedosto josta tarvittaessa haetaan taulujen mallit eri toimintoja varten.

```
JS index.js
models > JS index.js > ...
1  const Comment = require("./Comment");
2  const User = require("./User");
3  const Post = require("./Post");
4  const Community = require("./Community");
5  const UserCommunities = require("./UserCommunities");
6  const UserRatedPosts = require("./UserRatedPosts");
7
8  User.hasMany(Post);
9  User.hasMany(Comment);
10 Comment.hasMany(Comment);
11 Post.hasMany(Comment);
12 Community.hasMany(Post);
13 Community.belongsToMany(User, {
14   through: UserCommunities,
15 });
16 User.belongsToMany(Community, {
17   through: UserCommunities,
18 });
19 User.belongsToMany(Post, { through: UserRatedPosts, as: "ratedPosts" });
20 Post.belongsToMany(User, { through: UserRatedPosts, as: "usersRated" });
21 Post.belongsTo(Community);
22 Comment.belongsTo(Post);
23 Comment.belongsTo(User);
24 Post.belongsTo(User);
25
26 module.exports = {
27   User,
28   Comment,
29   Post,
30   Community,
31   UserCommunities,
32   UserRatedPosts,
33 };
34
```

KUVA 28. Sequlize-tietokantataulumallit tuodaan omista tiedostoistaan index.js-tiedostoon, jossa määritellään niiden monen-suhde-moneen- ja yhden-suhde-moneen yhteydet.

3.6.2 HTTP REST API -rajapinnat

Sovelluksen backend-puolella luotiin HTTP REST API -rajapinnat. Sovelluksen ominaisuudet, jotka vaativat HTTP REST API -rajapintoja ovat muun muassa kirjautuminen sisään sekä uuden kommentin luominen (TAULUKKO 1). Frontend-puoli voi tehdä pyyntöjä REST API -rajapintoihin joiden kautta se voi esimerkiksi hakea tietoa. HTTP REST API -rajapinnat toteutetaan tässä sovelluksessa Expressin-reitittimiä hyödyntäen. Expressin reitittimen avulla voidaan määritellä toisiinsa liittyviä, mutta erilaisia osoitteita samassa tiedostossa. Erilliset reitittimet määritellään tässä sovelluksessa omissa tiedostoissaan controllers-hakemiston sisällä. Reititin tuodaan sen määrittelyn jälkeen päätiedostoon, jossa ne otetaan palvelimen käyttöön (KUVA 29).

```
app.use("/api/login", loginRouter);
app.use("/api/users", usersRouter);
app.use("/api/posts", postsRouter);
app.use("/api/comments", commentRouter);
app.use("/api/communities", communitiesRouter);
app.use("/api/utils", utilsRouter);
app.get("*", (req, res) => {
    res.sendFile(path.join(__dirname, "build", "index.html"));
});
```

KUVA 29. Express-reitittimien käyttöönotto ja niiden käsittelijöiden yhteisen osoitteen määrittely päätiedostossa.

TAULUKKO 1. Backend-puolen tarjoamat REST API-rajapinta osoitteet sekä niiden tarkoitukset.

Toiminta	**Metodi**	**URL**
Luo uusi kommentti	POST	/api/comments
Vahvista käyttäjän syöttämät käyttäjänimi ja salasana ja palauta JWT-tokeni, jos ne ovat oikein.	POST	/api/login
Hae kaikki postaukset	GET	/api/posts
Hae yksittäinen postaus	GET	/api/posts/(postID)
Luo uusi postaus	POST	/api/posts
Liitä käyttäjä yhteisöön	POST	/api/communities/(community ID)/user
Poista käyttäjä yhteisöstä	DELETE	/api/communities/(community ID)/user

Hae kaikki yhteisöt	GET	/api/communities
Luo uusi käyttäjä	POST	/api/users
Hae uusi AWS S3 lataus URL	GET	/api/utils/s3Url

3.7 AWS S3

Kun käyttäjä tekee uuden postauksen, tarvitaan jokin tapa varastoida postaukselle mahdollisesti ladattu kuva. Tämä toteutetaan tässä sovelluksessa niin, että kuvat varastoidaan Amazon Web Services S3 -pilvipalveluvarastoon. Kuvien latauksen AWS S3 -palveluun voisi suorittaa sekä frontend-puolella että backend-puolella. Tässä sovelluksessa kuvat lähetetään frontend-puolelta S3-varastoon (KUVA 30). Tämän mahdollistamiseksi luotiin backend-puolella REST API -rajapinta, josta frontend-puoli pyytää osoitetta, jonka kautta se voi ladata S3-varastoon kuvan, jonka käyttäjä antoi postaukselle. Pyydettyyn osoitteeseen voi asettaa aikarajan kuinka kauan sinne voi ladata kuvia niin, että ainoastaan postauksen tekijä voi ladata kuvan, koska osoite vanhentuu nopeasti ja sitä on hankala arvata.

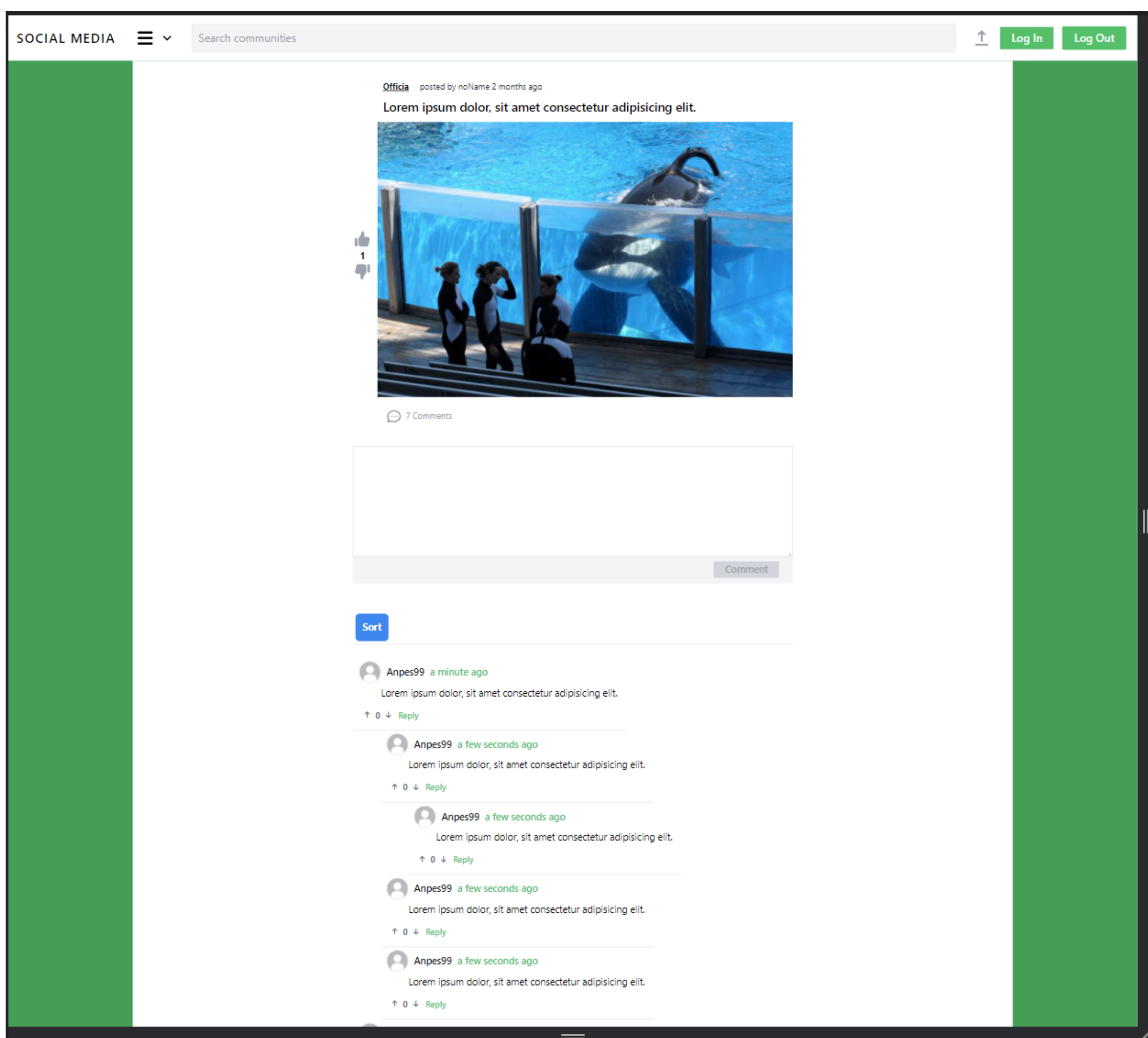
Kuvien yhdistäminen uuteen postaukseen toimii niin, että kuvan lataamiseen käytetystä osoitteesta erotetaan kaikki parametrit pois ja sitten se lisätään postauksen tietokantaolion imageUrl-kentän arvoksi. Kuvan pääsee hakemaan parametrien erottamisen tuloksena olevasta osoitteesta.

```
if (img) {
  const res = await axios.get("/api/utils/s3Url", {
    // get aws s3 put url // expires in 1min
    headers: {
      Authorization: `bearer ${user.token}`,
    },
  });
  await axios.put(res.data.url, img); //  put request to save img in aws s3 storage
  imageUrl = res.data.url.split("?")[0];
  newPost.imageUrl = imageUrl;
}           // post req to save post data to db
const response = await axios.post("/api/posts", newPost, {
  headers: {
    Authorization: `bearer ${user.token}`,
  },
});
```

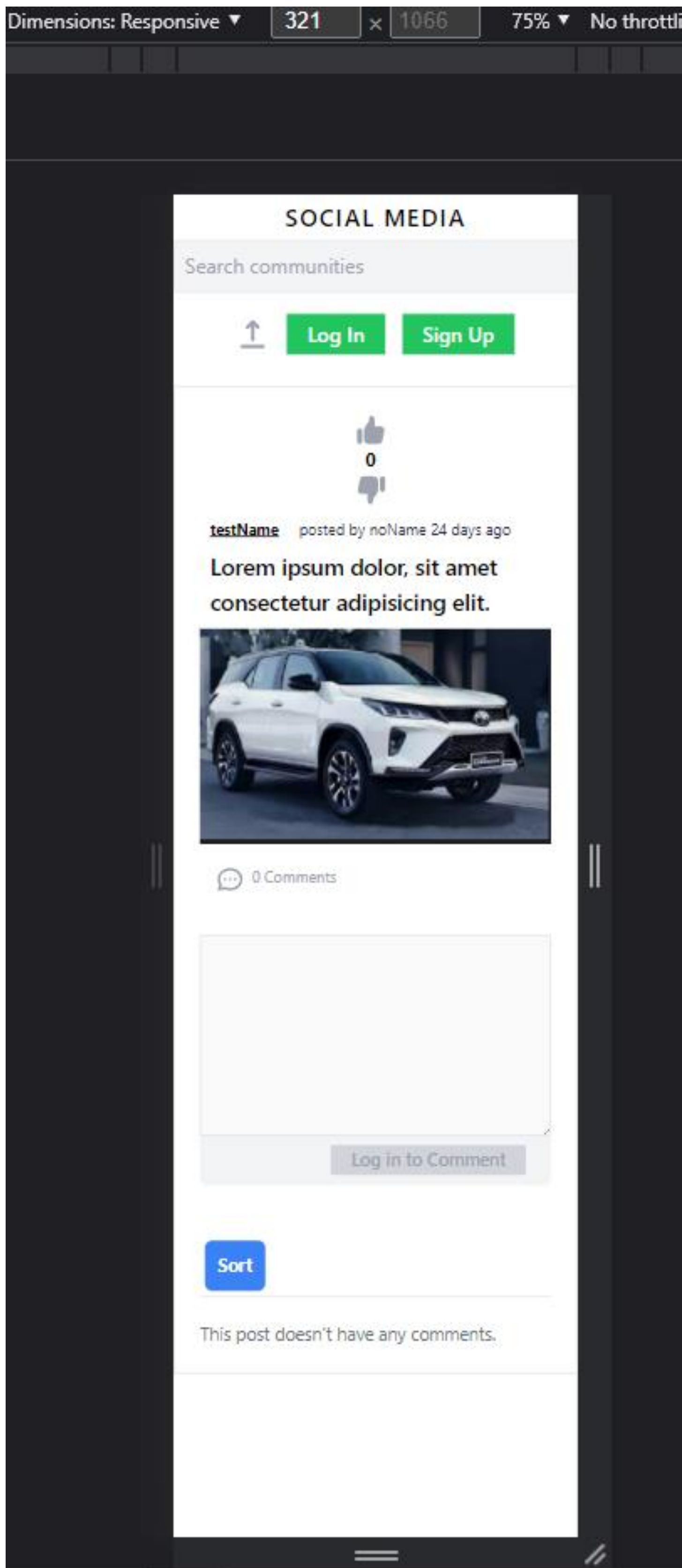
KUVA 30. Frontend-puolella kuvan lataaminen S3-varastoon sekä postauksen muiden tietojen lähettäminen Backend-puolelle tietokantaan varastoitavaksi.

3.8 Yksittäisen postauksen sivu

Sovelluksessa yksittäisen postauksen sivulle pääsee joko käyttämällä oikeaa osoitetta tai klikkaamalla kyseistä postausta kotisivulla tai yhteisösivulla. Postauksen sivulla näkyy postauksen kuva, otsikko sekä kommentit (KUVA 31). Postausta ja muita kommentteja voi kommentoida tai äänestää, jos käyttäjä on kirjautunut sisään. Kommentteja voi myös järjestää niiden iän tai niiden saamien äänien mukaan. Käyttäjä pääsee myös luomaan oman postauksen napauttamalla oikeassa yläkulmassa olevaa latausnappia, joka siirtää käyttäjän uuden postauksen luomisen sivulle. Hyödyntämällä TailwindCSS:ää tehtiin myös sovelluksen yksittäisen postauksen sivusta näytön leveyteen sopeutuva, jotta sen käyttäminen olisi helppoa myös mobiililaitteilla (KUVA 32).



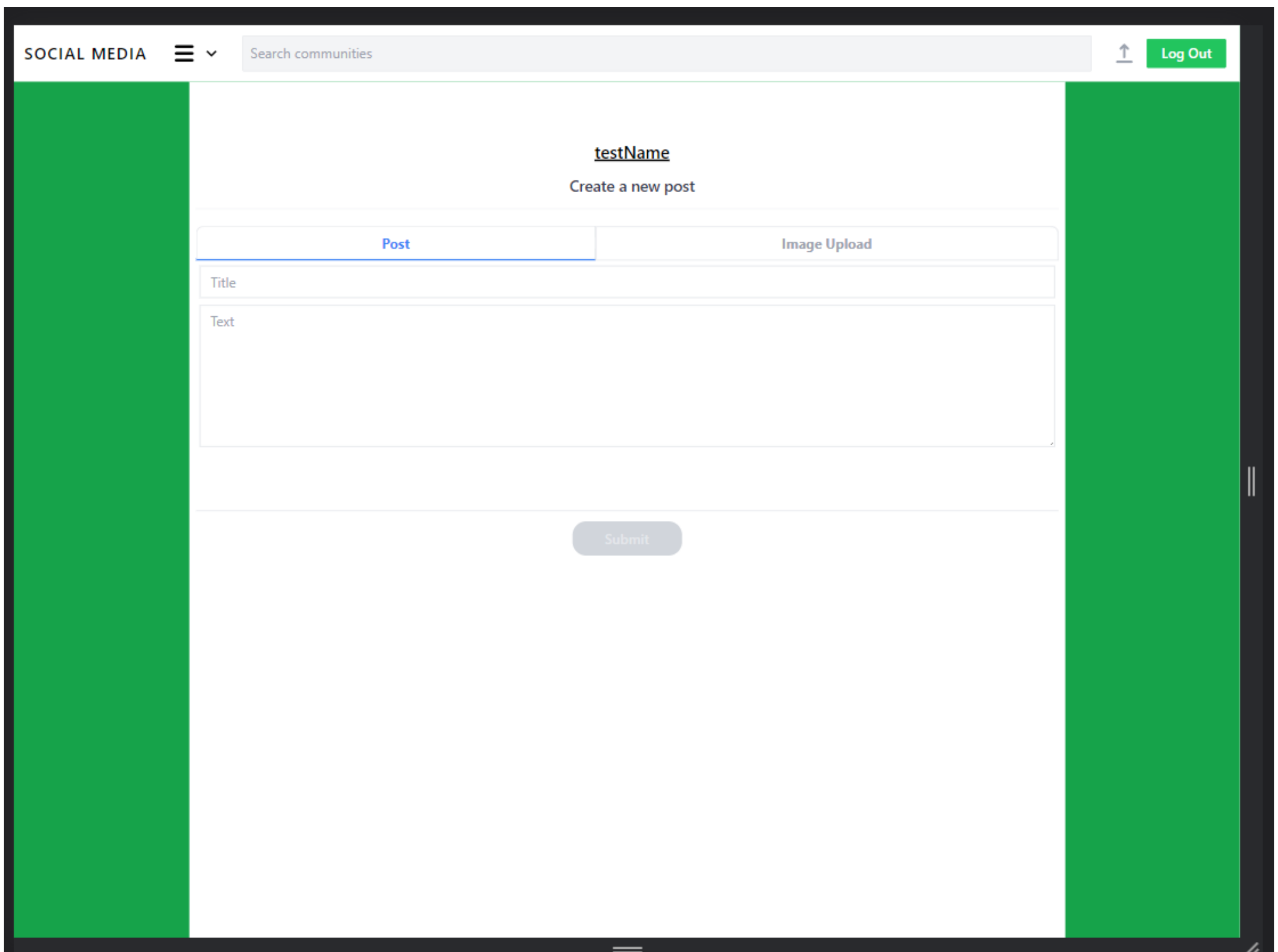
KUVA 31. Yksittäisen postauksen sivu, jossa näkyvät muun muassa postauksen kommentit, kuva ja otsikko.



KUVA 32. Yksittäisen postauksen sivu kapealla näytöllä. HTML-elementit ovat sopeutuneet näytön leveyden muutoksiin, koska niiden CSS-luokkien arvot ovat muuttuneet näytön leveyden muutoksen seurauksena.

3.9 Uuden postauksen luomisen sivu

Uuden postauksen luomiseen tarkoitettu sivu sisältää lomakkeen, jolla kirjautunut käyttäjä voi luoda uuden postauksen, joka on liitetty tiettyyn yhteisöön (KUVA 33). Yhteisö johon postaus liitetään valitaan niin, että osoitteesta erotetaan sen yhteisön nimi jonka alla lomaketta katsellaan. Yhteisön nimi näkyy myös lomakkeen yläpuolella. Lomakkeessa on kolme kenttää. Kentät on tarkoitettu otsikolle, päätekstille sekä kuvan lataamiselle. Lomakkeen validointi tapahtuu niin, että postaus voidaan julkaista ainoastaan, jos otsikon sekä päätekstin kentissä on tarpeeksi tekstiä.



KUVA 33. Uuden postauksen luomisen sivu.

4 POHDINTA

Tämän opinnäytetyön tavoitteena oli oppia uutta ohjelmoinnista ja kehittää verkkosivusto, jolla on monia sosiaalisen median ominaisuuksia kuten rekisteröinti, kirjautuminen, uusien postauksien luominen sekä niiden selaaminen, äänestäminen ja kommentointi.

Full-stack sosiaalinen media -sovelluksen kehittämisessä kehitettiin sovellukselle frontend-puoli sekä backend-puoli. Frontend-puolella suunniteltiin ja toteutettiin käyttöliittymä, joka sopeutuu näytön kokoon. Frontend-puolella käytettiin myös monia eri kirjastoja erilaisten ominaisuuksien toteuttamiseksi.

Backend-puolella suunniteltiin ja toteutettiin muun muassa tietokanta, HTTP REST API -rajapinnat sekä WebSocket-rajapinnat.

Lopputuloksena olevalla full-stack-sovelluksella on erilaisia ominaisuuksia kuten rekisteröinti, kirjautuminen, liittyminen yhteisöihin, postauksien äänestäminen sekä niiden selaaminen ja luominen. Sovelluksessa käyttäjältä vaaditaan kirjautuminen tiettyjä ominaisuuksia varten. Sovellukseen kehitettiin myös yhteisöille omat sivut, joissa voi selata pelkästään kyseisen yhteisön postauksia. Myös yksittäisille postauksille kehitettiin oma sivu, joka näyttää postauksen tiedot kokonaisuudessaan. Siinä käyttäjä pääsee myös kommentoimaan tai äänestämään postausta tai toisia kommentteja. Sovelluksen backend-puolelle kehitettiin myös mahdollisuus järjestää postauksia äänien tai päivämäärän mukaan WebSocket-rajapintaan.

Jatkokehitysideoita sovellukselle voisi olla esimerkiksi postauksien järjestämisominaisuuden lisääminen etusivulle sekä yhteisöjen sivuille.

LÄHTEET

ArpitAsati. 2022. What is web socket and how it is different from the HTTP?. Luettavissa: https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/ Luettu: 28.2.2022

Babeljs. What is Babel?. Luettavissa: https://babeljs.io/docs/en/ Luettu: 22.3.2022

Capan, T. 2013. Why The Hell Would I Use Node.js? A Case-by-Case Tutorial. Luettavissa: https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js Luettu: 27.2.2022

Codeacademy. React: The Virtual DOM. Luettavissa: https://www.codecademy.com/article/react-virtual-dom Luettu: 10.3.2022

Elliott, E. 2020. The missing introduction to React. Luettavissa: https://medium.com/javascript-scene/the-missing-introduction-to-react-62837cb2fd76 Luettu: 26.2.2022

Flaviocopes ja muut osallistujat. 2022. Introduction to Node.js. Luettavissa: https://nodejs.dev/learn/introduction-to-nodejs Luettu: 27.2.2022

Flaviocopes, MylesBorins, LaRuaNa, jgb-solutions, amiller-gh, ahmadawais. 2021. An introduction to the npm package manager. Luettavissa: https://nodejs.dev/learn/an-introduction-to-the-npm-package-manager Luettu: 26.2.2022

Freecodecamp. 2018. Let's learn how module bundlers work and then write one ourselves. Luettavissa:https://www.freecodecamp.org/news/lets-learn-how-module-bundlers-work-and-then-write-one-ourselves-b2e3fe6c88ae/ Luettu: 22.3.2022

Gamage, T. 2017. HTTP and Websockets: Understanding the capabilities of today's web communication technologies Luettavissa: https://medium.com/platform-engineer/web-api-design-35df8167460 Luettu: 7.5.2022

Hiwarale, U. 2018. How does JavaScript and JavaScript engine work in the browser and node?. Luettavissa: https://medium.com/jspoint/how-javascript-works-in-browser-and-node-ab7d0d09ac2f Luettu: 26.2.2022

Immukul. 2019. Unidirectional data flow. Luettavissa: https://www.geeksforgeeks.org/unidirectional-data-flow Luettu:26.2.2022

Javatpoint. 2022. JavaScript tutorial. Luettavissa: https://www.javatpoint.com/javascript-tutorial Luettu: 26.2.2022

Luukkainen, M. 2022. Flux arkkitehtuuri ja Redux. Luettavissa: https://fullstackopen.com/osa6/flux_arkkitehtuuri_ja_redux Luettu: 1.3.2022

MDN contributors. 2022. Express/Node introduction. Luettavissa: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction Luettu: 26.2.2022.

Mozilla. 2022. Introduction to the DOM. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction Luettu: 22.3.2022

Nodejs. 2019. Overview of Blocking vs Non-Blocking. Luettavissa: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/ Luettu: 13.3.2022

Redux. 2022. Redux Fundamentals, Part 1: Redux Overview. Luettavissa: https://redux.js.org/tutorials/fundamentals/part-1-overview Luettu: 27.2.2022

Reactjs. 2022. Components and Props. Luettavissa: https://reactjs.org/docs/components-and-props.html Luettu: 10.3.2022

Reactjs. 2021a. Create a new React app. Luettavissa: https://reactjs.org/docs/create-a-new-react-app.html Luettu: 1.3.2022

Reactjs. 2021b. ReactDOM. Luettavissa: https://reactjs.org/docs/react-dom.html Luettu: 12.3.2022

Reactjs.2021c. Reconciliation. Luettavissa: https://reactjs.org/docs/reconciliation.html Luettu: 13.3.2022

Reactjs. 2020. Hooks at a Glance. Luettavissa: https://reactjs.org/docs/hooks-overview.html Luettu: 28.2.2022

Reactjs. 2019. Using the Effect Hook. Luettavissa: https://reactjs.org/docs/hooks-effect.html Luettu:10.3.2022

Redux-toolkit. 2021. Createslice. Luettavissa: https://redux-toolkit.js.org/api/createslice Luettu: 28.2.2022

Redux-toolkit. 2022. Writing Reducers with Immer. Luettavissa: https://redux-toolkit.js.org/usage/immer-reducers Luettu: 1.3.2022

Roberts, P. 2014. What the heck is the event loop anyway?. Katsottavissa: https://www.youtube.com/watch?v=8aGhZQkoFbQ Katsottu: 26.2.2022

Tay, Y. 2019. In-Depth Overview. Luettavissa: https://facebook.github.io/flux/docs/in-depth-overview/ Luettu: 10.3.2022

TailwindCSS. Get started with TailwindCSS. Luettavissa: https://tailwindcss.com/docs/installation/using-postcss Luettu: 22.3.2022

Tutorialspoint. Reactjs introduction. Luettavissa: https://www.tutorialspoint.com/reactjs/reactjs_introduction.htm Luettu: 21.3.2022

Visual-paradigm. 2018. What is Object Relational Mapping (ORM)?. Luettavissa: https://circle.visual-paradigm.com/docs/database-design-engineering/programmers-guide/what-is-object-relational-mapping-orm/ Luettu: 22.3.2022

W3schools. JavaScript ES6. Luettavissa: https://www.w3schools.com/js/js_es6.asp Luettu: 11.3.2022

LIITE 1

Projektin lähdekoodi: https://github.com/Anpes99/Social_Media_App