WEB-SOVELLUKSEN TEKO REACTILLA JA DJANGO REST FRAMEWORKILLA

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Tieto- ja viestintätekniikan koulutusohjelma

Kevät, 2022

Henri Mattila

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikan koulutusohjelma Tiivistelmä
Tekijä Henri Mattila Vuosi 2022
Työn nimi Web-sovelluksen teko Reactilla ja Django REST Frameworkilla
Ohjaaja Petri Kuittinen

Opinnäytetyön tavoitteena oli rakentaa sosiaalisen median sovellus, käyttäen moderneja web-sovelluskehityksen tekniikoita. Sosiaalisen median sovelluksessa käyttäjä pystyy kirjautumaan sisään, seuraamaan muita käyttäjiä ja lähettämään viestejä omalle seinälleen, jossa näkyvät myös seurattujen käyttäjien viestit.

Teoriaosuudessa käydään läpi projektissa käytettyjä tekniikoita ja toimintatapoja. Sovellus oli tarkoitus tehdä moderneilla tekniikoilla, joten käyttöliittymä tehtiin React-kirjastolla, jolla onnistuu SPA tyylisen web-sovelluksen teko. Sovelluksen palvelinpuolen ohjelma rakennettiin Django REST Frameworkilla.

Toteutus osuudessa käydään läpi sovelluksen rakennuksen eri vaiheita. Ensimmäisenä palvelinpuolen ohjelman tekeminen ja sen jälkeen käyttöliittymän toteutus. Lopuksi käydään läpi, kuinka valmis sovellus toimii ja näyttää selaimessa.

Projekti opetti kuinka web-sovelluksia rakennetaan ja julkaistaan verkkoon. Käytettyjen tekniikoiden osaaminen kehittyi ja hyvät ohjelmointitavat tulivat tutuiksi. Lopullinen sovellus saatiin julkaistua muiden käyttäjien käytettäväksi.

Avainsanat Web-sovellus, Javascript, React, Python, Django
Sivut 25 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Information Technology
Author Henri Mattila
Subject Web application creation with React and Django REST Framework
Supervisors Petri Kuittinen

Abstract
Year 2022

The aim of the thesis was to build a social media application, using modern web application development techniques. In a social media application, a user can log in, follow other users, and send messages to their own wall, which also displays messages from the users they follow.

The theoretical part reviews the techniques and procedures used in the project. The application was to be made with modern technologies, so the interface was made with the React library, which makes it possible to create a SPA web application. The server-side program of the application was built with the Django REST Framework.

The implementation part goes through the different phases of the application building. Making the backend first and then implementing the interface. Finally, we go through how the finished application works and displays in the browser.

The project taught how to build and publish web applications online. Knowledge of the techniques used developed and good programming practices became familiar. The final application was released for use by other users.

Keywords Web application, Javascript, React, Python, Django
Pages 25 pages

Sisällys

1 Johdanto

Opinnäytetyön tavoitteena oli rakentaa sosiaalisen median web-sovellus moderneilla web-sovelluskehityksen tekniikoilla. Projektissa olisi eriytetty käyttöliittymä palvelinpuolen sovelluksesta. Selaimessa toimivan sovelluksen tulisi olla responsiivinen, jotta käyttöliittymä olisi selkeä ja toimiva sekä pienillä että suurilla näyttökooilla. Työn aikatauluksi asetettiin kolme kuukautta.

Sovelluksessa olisi käyttäjän autentikointi, jotta siinä voisi kommunikoida ja toimia muiden käyttäjien kanssa. Jokaisella käyttäjällä olisi oma yksilöity seinä, jossa näkyisi kaikki omat, sekä seurattavien käyttäjien viestit. Ne näkyisivät uusimmasta vanhimpaan ja sivulla näkyisi vain ensimmäiset 20 viestiä kerrallaan. Loput olisivat sivutettuina muille sivuille. Kaikkia viestejä pitäisi pystyä kommentoimaan ja tykkäämään, lukuunottamatta omia viestejä.

Sisäänkirjautunut käyttäjä voisi hakukentästä etsiä seurattavia tai blokattavia kavereita. Mikäli kaveria alkaa seuraamaan, näkyvät hänen viestinsä seuraajan sivulla. Blokkauksen jälkeen puolestaan blokattu käyttäjä ei pystyisi seuraamaan blokkaajaa ennen kuin blokkaus on peruutettu. Käyttäjän pitäisi pystyä muokkaamaan omia tietojaan, kuten käyttäjänimeä. Käyttäjällä olisi myös mahdollisuus lisätä profiilikuva, joka näkyisi viestien ja kommenttien yhteydessä.

Sovellus ei sisältäisi kuva-albumia, vaan ainoastaan yhden profiilikuvan käyttäjä voisi ladata sovellukseen. Myöskään muiden tiedostotyyppien lisääminen ei onnistu, esimerkiksi viestin liitteeksi.

2 Käytetyt tekniikat

Sovellus haluttiin tehdä käyttäen moderneja web-sovelluskehityksen tekniikoita. Käyttäjälle näkyvä selaimen käyttöliittymä on tehty hyödyntäen Reactia, jolla saa tehtyä Single Page App web-sovelluksia. Muita käyttöliittymässä käytettyjä tekniikoita oli Javascript, Redux Toolkit, Material UI, HTML5 ja CSS. Palvelinpuolen logiikka tehtiin Pythonilla Django REST -sovelluskehystä hyödyntäen. Tietokantana käytettiin PostgreSQL:ää, joka sopii hyvin

projekteihin, jotka halutaan julkaista Heroku-alustalla. Sovelluksen julkaisu tehtiin Heroku -pilvipalvelulla, koska se on ilmainen tiettyyn pisteeseen asti. Sovelluksessa on myös mahdollista tallentaa profiilikuva, joten siihen tarkoitukseen oli hyvä valita erillinen palvelu tiedostojen tallentamiseen. Googlen Firebase Cloud Storage valikoitui alustaksi kuvien tallentamiseen.

2.1 Palvelinpuolen käytetyt tekniikat

Sovelluksen palvelinpuolen ympäristössä käytettiin ohjelmointikielenä Pythonia. Projektin teossa hyödynnettiin Django REST Framework nimistä ohjelmistokehystä. Ilman ohjelmistokehystä tai kirjastoa olisi projektin tekeminen hyvin työlästä. REST-rajapinnassa tallennetaan tieto tietokantaan, joten siihen tarkoitukseen valikoitui PostgreSQL-järjestelmä. Sovelluksessa tallennetaan myös kuvatiedostoja, mutta ne on järkevämpää säilöä muualla kuin tietokannassa. Mikäli tietokantaan tallentaisi yli yhden megatavun tiedostoja, hidastaisi se tietokannan toimintaa merkittävästi. Tiedostoja kannattaakin mieluummin tallentaa esimerkiksi palvelimen tiedostojärjestelmään. (Gray, 2006) Tiedostojen tallentamiseen otettiin käyttöön Firebase Cloud Storage -pilvipalvelu. Lopuksi koko projekti julkaistaan Heroku pilvipalvelussa.

2.1.1 Django REST Framework

Django REST Framework on tehokas ja joustava sovelluskehys erilaisten web-sovellusten rajapintojen tekemiseen. Django on avoimeen lähdekoodiin perustuva korkeantason Python ohjelmointikielellä rakennettu sovelluskehys. (Django, 2022) Ohjelmistokehyksellä tehty projekti noudattaa REST arkkitehtuurityyliä. Sen avulla voidaan lähettää asiakasohjelmalle tietoa esimerkiksi JSON tai XML muodossa. (Hat, 2020) REST -rajapinnan hyviä puolia on esimerkiksi se, että se ei ota kantaa siihen, millä tekniikalla asiakasohjelma on rakennettu. Sovellukset jotka käyttävät HTTP tai HTTPS protokollaa voivat hakea tietoa REST -rajapinnasta ja lisäksi niiden pitää pystyä käsittelemään sitä tietotyyppiä, mitä rajapinta palauttaa.

2.1.2 PostgreSQL

Yksi tärkeä osa sovellusta on tiedon tallentaminen muistiin palvelimelle. Se tapahtuu tekemällä esimerkiksi SQL kyselyjä relaatiotietokantaan. SQL on IBM:n kehittämä kyselykieli, joka on kehitetty 1970-luvulla. (cs.helsinki, 2022) Relaatiotietokanta on tietokanta, joka tallentaa ja tarjoaa tietoa, jotka ovat yhteydessä toisiinsa. Ne perustuvat relaatiomalliin, joka on intuitiivinen ja suoraviivainen tapa esittää tietoja taulukoissa. Taulukon jokaisella rivillä on yksilöllinen tunnus, jota kutsutaan avaimeksi. (Oracle, 2022) Relaatiotietokannan hallintajärjestelmäksi valikoitui PostgreSQL. Muita hyviä ilmaisia avoimen lähdekoodin järjestelmiä olisivat olleet mm. Firebird, MySQL ja MariaDB. Lisäksi on olemassa erilaisia suljetun lähdekoodin hallintajärjestelmiä, kuten Oracle ja Microsoft SQL Server. Yksi syy miksi PostgreSQL valikoitui tähän projektiin oli se, että sovellus julkaistiin Heroku ympäristöön, joka tukee PostgreSQL järjestelmää.

2.1.3 Heroku

Sovelluksen ollessa valmis käyttöön, pitää se julkaista muiden käyttäjien saataville verkkoon. Web-sovelluksen julkaisemiseen on olemassa monia hyviä pilvipalvelin alustoja. Suosittuja palveluita ovat mm. Microsoftin Azure, Amazonin AWS ja Heroku. Näistä vaihtoehdoista valikoitui julkaisualustaksi Heroku. Se on ilmainen tiettyyn pisteeseen asti ja se tukee hyvin Pythonilla tehtyjä sovelluksia. Heroku on perustettu vuonna 2007 ja se tukee monia ohjelmointikieliä, kuten Node.js, PHP, Ruby, Python ja Java. Nykyisin sen omistaa Salesforce niminen yritys, joka osti Herokun vuonna 2010. (Mentormate, 2022)

2.1.4 Firebase Cloud Storage

Firebase Cloud Storage on Googlen kehittämä pilvipalvelu tiedostojen tallentamiseen. Se on tarkoitettu käytettäväksi sovelluskehittäjille, jotka haluavat, että heidän sovelluksessaan on käyttäjillä mahdollisuus tallentaa tai ladata tiedostoja. Tiedostotyypit voivat olla esimerkiksi kuvia tai videoita. Firebasessa on myös muita palveluita, joita voisi hyödyntää tiedostojen tallentamisen lisäksi. Niitä ovat esimerkiksi käyttäjien autentikointi, reaaliaikainen tietokanta

ja pilvifunktiot. (Google, 2022) Muita vastaavia palveluita tiedostojen tallentamiseen olisi löytynyt mm. Amazonin AWS:tä ja Microsoftin Azuresta.

2.2 Käyttöliittymässä käytetyt tekniikat

Sovelluksen selaimessa käyttäjälle näkyvä käyttöliittymä on tehty hyödyntäen erilaisia Javascript-kirjastoja, HTML:ää ja CSS:ää. Käyttöliittymästä haluttiin mahdollisimman responsiivinen, joten siitä tehtiin SPA sovellus. SPA sovelluksissa haetaan käyttäjälle vain kerran web-dokumentti palvelimelta, jonka jälkeen kaikki sovelluksen myöhemmin tarvitsema data haetaan taustalla ilman, että kaikkia web-dokumentteja tarvitsee hakea uudestaan. SPA:n huonoina puolina pidetään sovelluksen tilanhallintaa, joka joissain tilanteissa voi olla työlästä. Hakukoneoptimointia ei myöskään pysty niin hyvin toteuttamaan, kuin tavallisissa nettisivuissa. (Mozilla, 2022 -d)

2.2.1 HTML, CSS ja Javascript

HTML on web-dokumentin perustus, joka määrittelee verkkosisällön merkityksen ja rakenteen. HTML merkkauskieli perustuu elementteihin, joilla sivu saadaan näyttämään tietyltä tai toimimaan tietyllä tavalla. (Mozilla, 2022 -c) CSS on tyylikieli, joka kuvaa, kuinka HTML elementit tulee renderöidä näytölle (Mozilla, 2022 -b). Sillä voidaan esimerkiksi muuttaa elementin väriä, asettelua ja kokoa näytöllä. Javascript on ohjelmointikieli, jolla ohjelmoidaan, kuinka verkkosivut käyttäytyvät tapahtuman sattuessa (Mozilla, 2022 -a). Esimerkiksi hiiren klikkauksella web-sivuun voidaan triggeröidä jokin tapahtuma, jonka Javascript suorittaa.

2.2.2 React.js

React on tällä hetkellä yksi suosituimmista Javascript-kirjastoista. Yksinkertaistettuna sillä luodaan komponentteja, joita liittämällä yhteen muodostetaan käyttöliittymä. (Chiarelli, 2018) Sillä voi rakentaa SPA-tekniikan käyttöliittymiä ja nykyään sillä voidaan rakentaa myös mobiilisovelluksia. Reactin on kehittänyt Facebookissa työskennellyt insinööri Jordan Walke. Ensimmäisenä kirjasto otettiin käyttöön vuonna 2011 Facebookin uutisvirtaosiossa. React

käyttää JSX syntaksia, joka muistuttaa HTML-merkkauskieltä. Sitä kirjoitetaan suoraan Javascript-tiedostoon ja JSX lopulta renderöityy oikeaksi HTML:ksi. Syntaksin etuna on se, että sillä voidaan asettaa Javascriptia tavallaan suoraan HTML-dokumenttiin ja sitä kautta voidaan esimerkiksi toistaa listasta arvoja ja HTML-elementtejä dokumenttiin. (Pandit, 2021) Kirjaston yksi oleellinen osa on tilanhallinta. Sillä saadaan muutettua käyttöliittymän näkymässä arvoja asynkronisesti. (Ström, 2019)

2.2.3 Redux

Olennainen osa varsinkin laajoja SPA-web-sovelluksia on tilanhallinta. Mitä laajempi ja isompi sovellus on, sitä vaikeammaksi menee tilanhallitseminen. Tähän avuksi on kehitetty Redux-kirjasto. Se on tavallaan kuin käyttöliittymän tietokanta, jossa tieto on saatavilla ja muokattavissa kaikissa sivuston komponenteissa. (ElHousieny, 2021) Esimerkiksi, jos käyttäjä kirjoittaa sivulle tekstin, niin se on sen jälkeen saatavilla kaikilla sivuston näkymissä. Sovelluksen taustalla teksti mahdollisesti tallentuu myös palvelimen tietokantaan ilman, että käyttäjä sitä huomaa.

2.2.4 Material-UI

Material-UI on React-komponenttikirjasto, jolla saa lisättyä sovellukseen valmiita komponentteja. Esimerkkejä valmiista komponenteista ovat esimerkiksi navigointipalkki, modaali, nappi ja lataus spinneri. Kirjasto perustuu Googlen Material Designiin, joten komponentit muistuttavat paljon esimerkiksi Androidin komponentteja. (Freecodecamp, 2018)

2.3 Versionhallinta

Versionhallinnalla tarkoitetaan sovelluksen koodin muutosten seuraamista ja niiden hallintaa. Varsinkin jos projektia on työstämässä useampi henkilö, tulee versionhallinta entistä tärkeämmäksi työkaluksi. Mikäli koodiin tekee yhtaikaa useampi henkilö muutoksia, kannattaa jokaisen koodarin ottaa käyttöön oma haara sovelluksen versiosta. Silloin muutoksia on helpompi seurata ja verrata muiden muutoksiin. Lopuksi kun haaran

muutokset ovat valmiit, liitetään se takaisin alkuperäiseen haaraan, jolloin muutokset siirtyvät sinne. Versionhallinta on myös hyödyllinen, jos projektia tekee vain yksi henkilö. Silloin ei välttämättä tarvitse erikseen luoda omia haaroja päähaarasta, vaan silloin voi esimerkiksi seurata sovelluksen muutosten historiaa. Mikäli huomaa, että sovelluksen aiempi versio olikin parempi, niin siihen voi palata takaisin versiohallinnan avulla. (Atlassian, 2022) Versionhallintatyökaluja on olemassa useampia, kuten Git ja SVN. Projektissa käytettiin Git:ä.

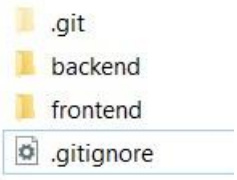
2.4 Koodieditori

Koodieditorina toimi Microsoftin kehittämä Visual Studio Code. Se on ilmainen avoimen lähdekoodin työkalu ja sen voi asentaa Windowsille, MacOS:lle ja Linuxille. Editori on kevytkäyttöinen ja se tukee ison joukon eri ohjelmointikieliä ja kirjastoja. (Mustafeez, 2022) Lisäksi siihen on asennettavissa paljon erilaisia lisäosia, kuten esimerkiksi koodin formatointiin auttavia työkaluja.

3 Toteutus

Toteutusosuudessa käydään läpi projektin rakentaminen. React- ja Django-sovellukset eriytetään omiin kansioihin, mutta ne pidetään saman kansion alla. Projekti hakemiston juuressa luodaan komentokehotteella git-varasto komennolla **”git init”**. Tällöin saadaan työn muutoshistoria talteen. Projektin juureen lisätään myös gitignore niminen tiedosto, johon lisätään kaikki kansiot ja tiedostot, joita ei haluta mukaan git-varastoon (Kuva 1).

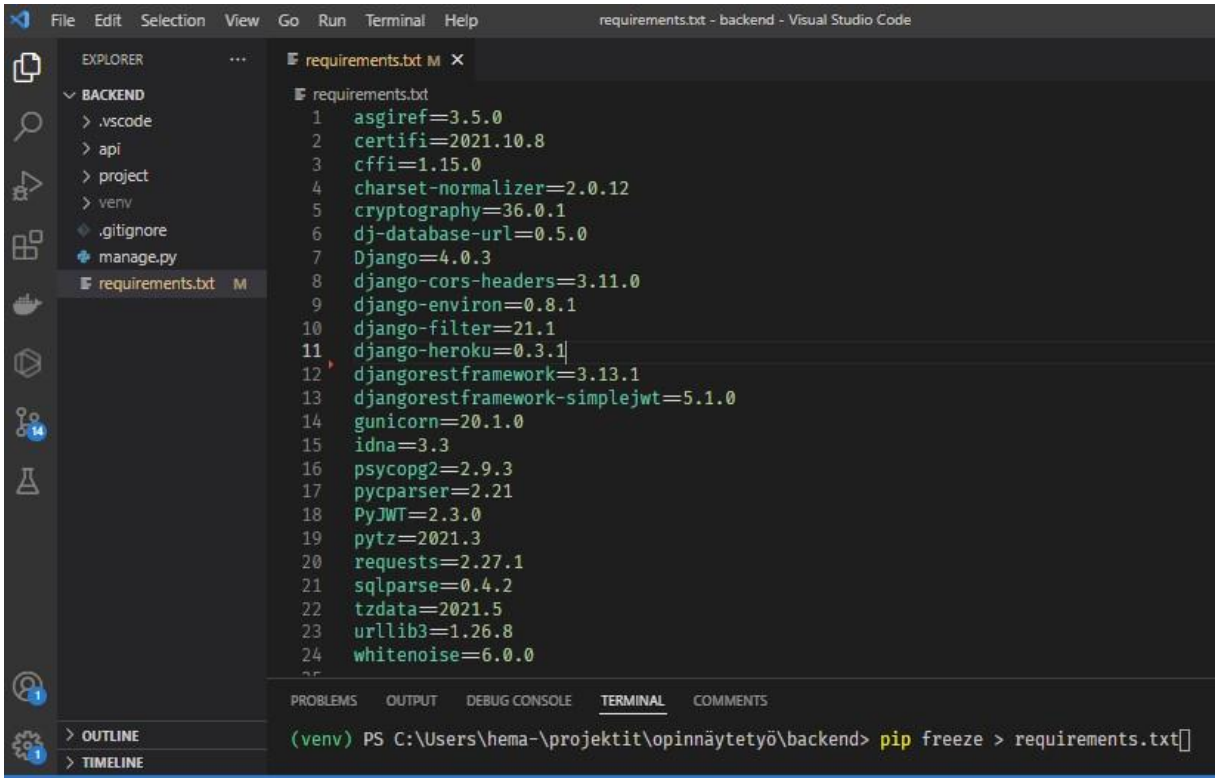
Kuva 1. Projektin juurihakemisto.



3.1 Palvelinpuolen sovelluksen rakentaminen

Palvelinpuolen sovellus aloitettiin ensimmäisenä, koska silloin saatiin rajapinta React-sovellukselle, josta hakea tietoa. Django REST sovelluskehys on tehty Python ohjelmointikielellä, joten ensimmäisenä piti asentaa uusin versio Pythonista tietokoneelle. Asennuksen mukana tulee oletuksena PIP, joka on Python-pakettien ja -moduulien asennustyökalu. Ennen sovellukseen tarvittavien pakettien asennusta, on hyvä aktivoida virtuaaliympäristö. Sen avulla saadaan eristettyä Python-sovellus muista projekteista, jolloin asennetut paketit eivät vaikuta mahdollisesti muihin koneella oleviin projekteihin. (Hillar, 2018) Seuraavana asennetaan komennolla **”pip install”** sovellukseen tarvittavat paketit, joita oli mm. django, djangorestframework ja qunicorn. Asennettujen pakettien versiot on hyvä siirtää lopuksi tiedostoon requirements.txt komennolla **”pip freeze > requirements.txt”**. Silloin saadaan siirrettyä tiedoston välityksellä tieto projektin tarvitsemista riippuvuuksista myös palvelimelle, johon sovellus lopulta viedään. (Kuva 2). Lopuksi asennetaan PostgreSQL tietokantaohjelma ja luodaan sinne käyttäjä sekä uusi tietokanta projektia varten.

Kuva 2. REST API projektin alustava rakenne ja asennetut riippuvuudet.



3.2 Palvelinpuolen sovelluksen rakentamisen työvaiheet

Uusi projekti luotiin terminaalin kautta projektin juuressa komennolla **”django-admin startproject”** ja siihen sovellus komennolla **”django-admin startapp”**. Seuraavana työvaiheena oli käydä muuttamassa settings.py tiedostosta asetukset kuntoon, jolloin saatiin esimerkiksi tietokantayhteys paikalliseen tietokantaan. Djangossa tulee mukana admin-käyttöliittymä, jonka kautta voi hallita sovelluksen tietokantaan tallennettuja tietoja. Sovelluskehyksessä tulee myös valmiina käyttäjienhallinta, joten sitä ei tarvinnut erikseen rakentaa. Komennolla **”python manage.py migrate”** saadaan ajettua tietokantaan taulut Python-malleista. Migraation jälkeen luotiin sovellukseen ensimmäinen käyttäjä, jolla on kaikki admin-oikeudet. Ensimmäinen käyttäjä saadaan luotua suoraan komentoriviltä komennolla **”python manage.py createsuperuser”**.

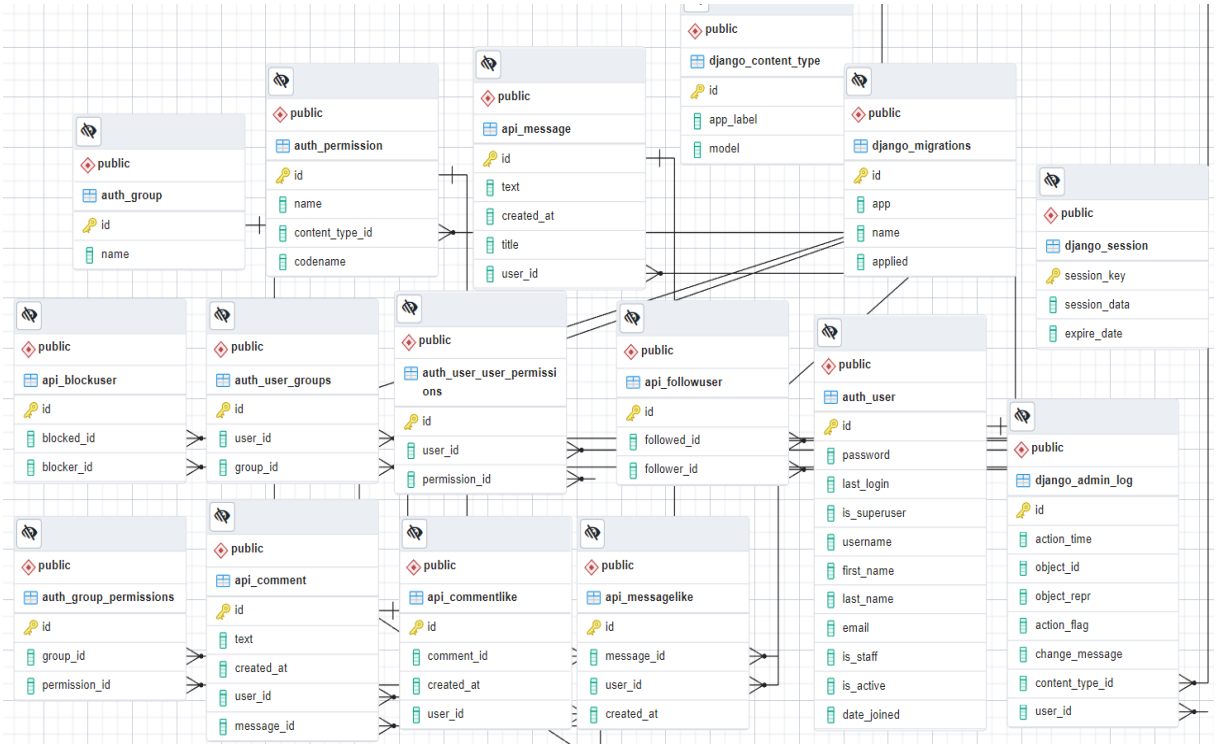
Models.py tiedostoon tehdään mallit, joiden mukaan luodaan tietokantaan taulut ja niiden väliset suhteet. Taulut rakentuvat Python-luokista, jotka sisältävät attribuutteja, joista muodostuvat tietokantataulun kentät. Jokaiselle attribuutille määritellään Python luokassa tietotyyppi, koska SQL-tietokannat voivat tallentaa vain tietoa, jonka tyyppi tiedetään. ER-kaaviosta näkee tietokannan taulut ja niiden väliset suhteet (Kuva 3).

Ohjelmakoodi 1. Osa sovelluksen malleista.

```
class Message(models.Model):
    title = models.CharField(max_length=100, null=True)
    text = models.CharField(max_length=1000)
    user = models.ForeignKey(User, related_name="messages",
on_delete=models.CASCADE, null=True, to_field="id", )
    created_at = models.DateTimeField(auto_now_add=True)
    def __str__(self):
        return str(self.id)

class Comment(models.Model):
    text = models.CharField(max_length=1000)
    message = models.ForeignKey(Message, related_name="comments",
on_delete=models.CASCADE, null=True)
    user = models.ForeignKey(User, related_name="comments",
on_delete=models.CASCADE, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    class Meta:
        ordering = ['-created_at']
```

```
    def __str__(self):
        return str(self.id)

class MessageLike(models.Model):
    message = models.ForeignKey(Message, related_name="message_likes",
on_delete=models.CASCADE, null=True, to_field="id",)
    user = models.ForeignKey(User, related_name="user_id",
on_delete=models.CASCADE, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    class Meta:
        unique_together = ['message', 'user']

class CommentLike(models.Model):
    comment = models.ForeignKey(Comment, related_name="comment_likes",
on_delete=models.CASCADE, null=True)
    user = models.ForeignKey(User, related_name="users_id",
on_delete=models.CASCADE, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    class Meta:
        unique_together = ['comment', 'user']


class FollowUser(models.Model):
    follower = models.ForeignKey(User, related_name="follower",
on_delete=models.CASCADE, null=False)
    followed = models.ForeignKey(User, related_name="followed",
on_delete=models.CASCADE, null=False)

    class Meta:
        unique_together = ['follower', 'followed']
```

Kuva 3. ER-kaavio tietokannasta.



Siinä vaiheessa kun halutaan saada tietoa luettavaan muotoon esimerkiksi selaimeen, otetaan käyttöön serialisaattorit. Niiden avulla saadaan muutettua Python-objektit JSON-muotoon, jota on helppo lukea selaimen puolella (Ohjelmakoodi 2). Django REST Frameworkissa on serializer-luokka, jolla saa serialisoitua Python-mallin suoraan JSON-muotoon sekä deserialisoitua JSON-tiedon takaisin Python-objektiksi. Luokka sisältää myös valmiin validaattorin, joka tarkistaa ovatko tiedot oikein.

Ohjelmakoodi 2. Osa serialisaattoreista.

```
class CommentLikeSerializer(serializers.HyperlinkedModelSerializer):
    comment =
serializers.PrimaryKeyRelatedField(queryset=Comment.objects.all(), many=False)
    user = serializers.PrimaryKeyRelatedField(queryset=User.objects.all(),
many=False)
    class Meta:
        model = CommentLike
        fields = ['comment', 'user']
    def validate(self, data):
        if data["user"] != self.context['request'].user:
            raise serializers.ValidationError("You cannot like for other users
on their behalf")
        if Comment.objects.get(id=data["comment"].id).user ==
self.context['request'].user:
```

```
            raise serializers.ValidationError("You cannot like your own
comment")
        return data

class CommentSerializer(serializers.ModelSerializer):
    message =
serializers.PrimaryKeyRelatedField(queryset=Message.objects.all(), many=False)
    comment_likes_count = serializers.SerializerMethodField(read_only=True)

    def get_comment_likes_count(self, comment):
        return comment.comment_likes.count()

    def create(self, validated_data):
        return Comment.objects.create(**validated_data)

    class Meta:
        model = Comment
        fields = ['id', 'text', 'message', 'user', 'created_at',
'comment_likes_count']

class MessageSerializer(serializers.ModelSerializer):
    comments = CommentSerializer(many = True, read_only = True)
    message_likes_count = serializers.SerializerMethodField(read_only=True)

    def get_message_likes_count(self, message):
        return message.message_likes.count()

    class Meta:
        model = Message
        fields = ['id', 'title', 'text', 'user', 'created_at', 'comments',
'message_likes_count']
```

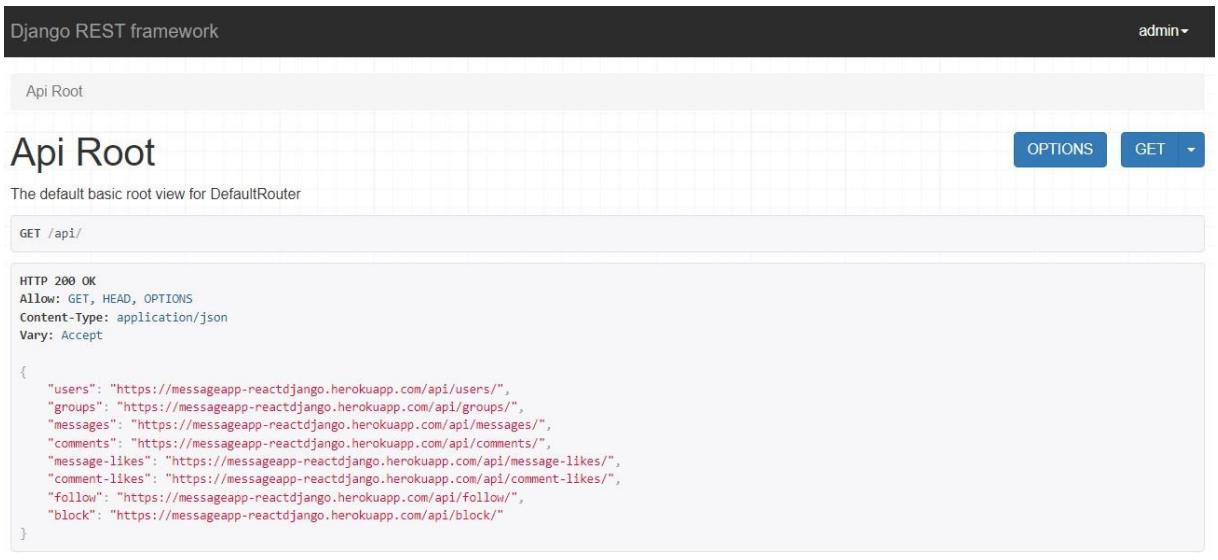
Django REST-rajapinnan näkymät tehtiin pääosin hyödyntäen ModelViewSet-luokkaa. Se luo automaattisesti näkymään oletus CRUD-toiminnallisuudet sekä URL-reitit (Ohjelmakoodi 3). Rajapinnan käyttö vaatii autentikoidun käyttäjän. Käyttäjän autentikoinnissa käytetään apuna JSON Web Tokenia.

Ohjelmakoodi 3. Osa näkymistä.

```
class UserViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.
    """
    queryset = User.objects.all().order_by('-date_joined')
    serializer_class = UserSerializer
```

```
    filter_backends = [filters.SearchFilter, DjangoFilterBackend]
    search_fields = ['username']
    filterset_fields = ['id']
    filterset_class = UserFilterSet
    permission_classes = []


    def get_permissions(self):
        if self.request.method == 'POST':
            return []
        else:
            return [permissions.IsAuthenticated()]


class GroupViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Group.objects.all()
    serializer_class = GroupSerializer
    permission_classes = [permissions.IsAuthenticated]

class MessageViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Message.objects.all().order_by('-created_at')
    serializer_class = MessageSerializer
    permission_classes = [permissions.IsAuthenticated]
    filterset_class = MessageFilterSet
```

Kuva 4. Näkymä REST rajapinnan juuresta adminille.



Näkymien jälkeen tehtiin URL-reititykset, jotta resursseihin pääsee käsiksi selaimella. Reittien luonnissa käytettiin hyväksi DefaultRouter-luokkaa, joka luo automaattisesti perus CRUD-toiminnallisuuteen tarvittavat URL-reitit. DefaultRouterilla luodut reitit otetaan käyttöön lisäämällä se urlpatterns nimiseen listaan. Siihen lisättiin myös itse luodut näkymät, kuten esimerkiksi sisäänkirjautumisen näkymä nimeltään login (Ohjelmakoodi 4).

Ohjelmakoodi 4. URL määritykset.

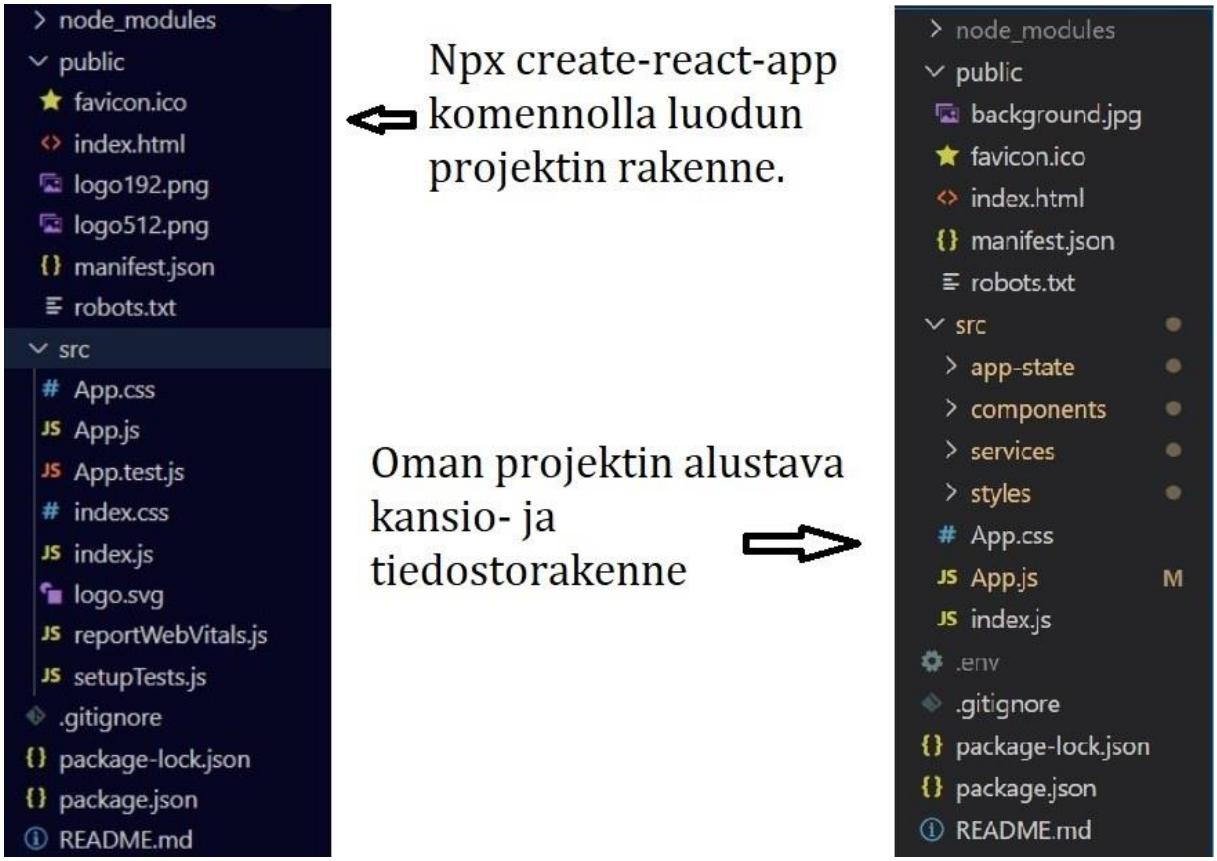
```
router = routers.DefaultRouter()
router.register(r'users', views.UserViewSet)
router.register(r'groups', views.GroupViewSet)
router.register(r'messages', views.MessageViewSet)
router.register(r'comments', views.CommentViewSet)
router.register(r'message-likes', views.MessageLikeViewSet)
router.register(r'comment-likes', views.CommentLikeViewSet)
router.register(r'follow', views.FollowUserViewSet)
router.register(r'block', views.BlockUserViewSet)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include(router.urls)),
    path('api-auth/', include('rest_framework.urls',
namespace='rest_framework')),
    path('api/token/', TokenObtainPairView.as_view(),
name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(),
name='token_refresh'),
    path('api/login/', views.MyTokenObtainPairView.as_view(), name='login'),
    path('api/followed-users/', views.FollowedUsersView.as_view()),
    path('api/blocked-users/', views.BlockedUsersView.as_view()),
    path('api/user-messages/', views.MessageForUserViewSet.as_view(),
name='user-messages'),
    re_path('.*', TemplateView.as_view(template_name='index.html'))
]
```

3.3 Käyttöliittymäsovelluksen rakentaminen

Käyttöliittymä tehtiin React.js Javascript-kirjastolla ja se eriytettiin projektissa omaksi sovellukseksi. Tietokoneelle asennettiin ensimmäisenä Node.js Javascript-ajoympäristö. Sen mukana tulee myös NPM ja NPX työkalut. NPM työkalulla voi esimerkiksi ladata omaan sovellukseen Javascript-paketteja ja NPX alkuisella terminaalikomennolla voi luoda alustavan

pohjan omalle React-projektille. Node.js:n asennuksen jälkeen ajettiin komento **”npx create-react-app”**, jolla saatiin alustava pohja sovellukselle. NPX komennon mukana tulee työkalut, joilla sovellusta voi kehittää tehokkaasti omassa lokaalissa ympäristössä. Lisäksi mukana tulee työkalu, jolla React-projektin voi pakata pienempään kokoon tuotantoversiota varten. Komennon jälkeen alustavaan React-sovellukseen tulee kuitenkin ylimääräisiä tiedostoja, joita ei projektissa tarvita. Ylimääräiset tiedostot poistettiin ja rakennettiin alustava kansiorakenne sovellukselle (Kuva 5).

Kuva 5. Muutos React-sovelluksen rakenteessa.



3.4 Käyttöliittymäsovelluksen rakentamisen työvaiheet

React-sovelluksen rakenne tehdään funktio-komponenteilla. App-niminen komponentti kokoaa sovelluksen muut komponentit yhteen (Ohjelmakoodi 5). Reactia käytettäessä pidetään hyvänä käytäntönä jakaa komponentit mahdollisimman pieniksi osiksi. Silloin niitä on helpompi käyttää uudelleen ja ne pysyvät helppolukuisina. Esimerkiksi sovelluksen etusivu, joka näkyy käyttäjän kirjautuessa sisään, sisältää monta komponenttia. Jokainen

sivulle renderöityvä viesti on oma komponenttinsa, joka puolestaan rakentuu useammasta pienemmästä komponentista. Etusivulla näkyvä viestin lähetyskenttä ja navigointipalkki ovat myös omia komponenttejaan.

Ohjelmakoodi 5. App komponentti.

```
function App() {
  return (
    <div>
      <NavBar />
      <AppContainer>
        <Routes>
          <Route path="" element={<Guide />} />
          <Route path="messages" element={<Homepage />} />
          <Route path="signup" element={<SignUp />} />
          <Route path="login" element={<Login />} />
          <Route path="followed-and-blocked" element={<FollowedAndBlocked />}
/>
          <Route path="images" element={<Images />} />
          <Route path="user" element={<EditUser />} />
        </Routes>
      </AppContainer>
      <BottomNavBar />
      <ToastContainer
        position="top-right"
        autoClose={3000}
        pauseOnHover
        theme="colored"
      />
    </div>
  );
}

export default App;
```

Services-kansioon on koottu useampaan tiedostoon REST-rajapinnasta tietoa hakeva logiikka. Esimerkiksi kaikki käyttäjään liittyvät kutsut löytyvät omasta tiedostostaan. Tiedonhakuun rajapinnasta käytettiin apuna Axios.js-kirjastoa. Sillä sai hieman yksinkertaistettua HTTP-pyyntö funktioiden rakentamista. Toinen vaihtoehtoinen tapa olisi ollut käyttää esimerkiksi Javascriptista oletuksena löytyvää Fetch API:a. Services-kansiosta löytyy myös Firebase-määritykset, jotta saadaan sovelluksesta yhteys Firebasen kuvien säilytyspalvelimelle.

Sovelluksen tilanhallinta toteutettiin Redux Toolkit-kirjastolla. Sen avulla saatiin pidettyä sovelluksen tila samana kaikissa komponenteissa. Tarvittava tieto saadaan haettua Reduxin tietovarastosta useSelector-funktiolla (Ohjelmakoodi 6). Mikäli tietoon halutaan tehdä muutoksia, tapahtuu se puolestaan dispatch-funktiolla. Dispatchilla tapahtuvat muutokset ja tiedon hakeminen tilavarastoon määritellään createSlice-funktioilla (Ohjelmakoodi 7).

Ohjelmakoodi 6. UseSelectorilla voidaan hakea Redux varastosta tietoa.

```
const { images } = useSelector((state) => state.images);
```

Ohjelmakoodi 7. CreateSlicessa määritellään haetulle tiedolla tila.

```
export const getAllImages = createAsyncThunk(
  "images/getAll",
  async (thunkAPI) => {
    try {
      let imgList = [];
      const imageListRef = ref(storage, `images/`);
      const url = await listAll(imageListRef);
      await Promise.all(
        url.items.map(async (item) => {
          let url = await getDownloadURL(item);
          imgList.push({ url: url, name: item.name });
        })
      );
      return imgList;
    } catch (err) {
      console.log(err);
      return thunkAPI.rejectWithValue(err);
    }
  }
);

export const imageSlice = createSlice({
  name: "images",
  initialState: {
    images: [],
    status: null,
  },
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(getAllImages.pending, (state, action) => {
      state.status = "loading";
    });
    builder.addCase(getAllImages.fulfilled, (state, action) => {
      state.images = action.payload;
```

```
      state.status = "fulfilled";
    });
    builder.addCase(getAllImages.rejected, (state, action) => {
      state.status = "rejected";
    });
  },
});

export const { loadImages } = imageSlice.actions;

export default imageSlice.reducer;
```

Reactissa on monia eri tapoja käyttää tyylejä komponenteissa. Tyylit voi tehdä esimerkiksi Javascriptilla, tyylitellyillä komponenteilla (Styled Components) tai perinteisellä CSS:llä. Projektissa käytettiin apuna tyylien tekoon Material UI-kirjastoa sekä Javascriptia. Material UI-kirjastolla luotiin pysäytyspisteet eri näyttökooille, jotta sovelluksesta saadaan responsiivinen. Isoimmista komponenteista tehtiin tyyliteltyjä komponentteja (Ohjelmakoodi 8). Vähemmän tyylimäärityksiä vaativiin JSX-elementteihin käytettiin Javascriptia (Ohjelmakoodi 9). Kaikki tyylit pidettiin samassa tiedostossa, joka helpotti niiden ottamista käyttöön eri paikoissa.

Ohjelmakoodi 8. Material UI kirjastolla luotuja tyylejä.

```
import { styled, createTheme } from "@mui/system";
import { Card } from "@mui/material";

export const theme = createTheme({
  breakpoints: {
    values: {
      xxs: 0, // small phone
      xs: 350, // phone
      sm: 600, // tablets
      md: 900, // small laptop
      lg: 1200, // desktop
      xl: 1536, // large screens
    },
  },
});

export const FormContainer = styled(Card)({
  display: "flex",
  flexDirection: "column",
  width: "80%",
```

```
  gap: "15px",
  padding: "2%",
  [theme.breakpoints.up("md")]: {
    width: "30%",
  },
});

export const SearchUsersContainer = styled("div")({
  width: "20%",
  backgroundColor: "white",
  [theme.breakpoints.down("lg")]: {
    width: "40%",
  },
  [theme.breakpoints.down("sm")]: {
    width: "60%",
  },
});
```

Ohjelmakoodi 9. Javascriptilla luotuja tyylejä.

```
export const myStyles = {
  textFieldStyle: {
    backgroundColor: "white",
    [theme.breakpoints.up("sm")]: {},
  },
  paperStyles: {
    backgroundColor: "#757de8",
    position: "fixed",
    bottom: 0,
    left: 0,
    right: 0,
    display: "flex",
    justifyContent: "center",
    zIndex: "1",
  },
  boxStyles: {
    backgroundColor: "#757de8",
    width: 500,
    display: "flex",
    justifyContent: "center",
  },
```

3.5 Julkaiseminen tuotantoon

Lopuksi, kun sovellus oli valmis, julkaistiin se tuotantoon muiden käyttäjien käytettäväksi. Julkaisualustana käytettiin tiettyyn pisteeseen asti ilmaista Heroku-palvelua. Herokuun julkaisu tehdään käyttäen Heroku CLI-työkalua. Se on komentokehotetyökalu, jolla saadaan esimerkiksi siirrettyä sovellus lokaalista ympäristöstä Herokun palvelimelle. Työkalulla voisi myös määrittää muita julkaisuun tarvittavia asetuksia, mutta ne onnistuvat myös helposti Herokun omilta web-sivuilta. Heroku tarvitsee myös projektin juureen Procfile nimisen tiedoston. Sillä määritellään komennot, jotka ajetaan sovelluksen käynnistyksen yhteydessä. Projektiin laitettiin ensimmäisenä tietokanta-migraatio-komento ja sen jälkee komento, jolla HTTP-serveri lähtee päälle (Kuva 6).

Kuva 6. Procfile komennot.

```
Procfile
1   release: python manage.py migrate
2   web: gunicorn project.wsgi --log-file -
```

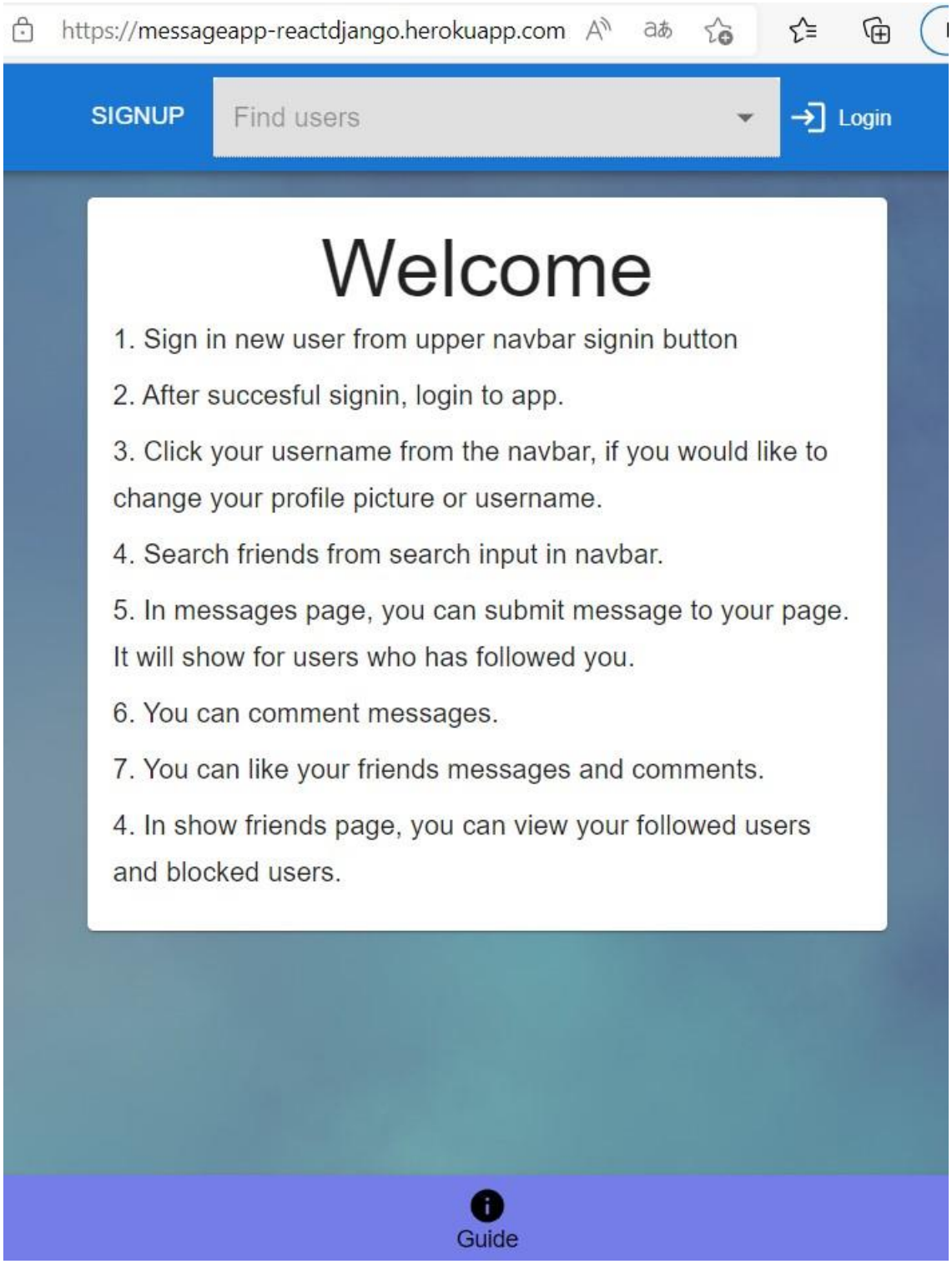
Ennen julkaisua piti vielä tehdä muutama muutos Django-projektiin ja siirtää pakattu tuotantoversio frontendista backend-projektin juureen. React-projektin juuressa annettiin komentoriviltä komento **”npm run build”**, jolla saadaan luotua pakattu tuotantoversio käyttöliittymästä. Komento luo build-kansion, josta löytyy koko sovellus. Kansio siirrettiin Django-projektin juureen, koska Python-sovellus toimi samalla HTTP-palvelimena. Djangon asetuksista piti käydä muuttamassa tieto siitä, mistä löytyy staattiset tiedostot. Polku laitettiin osoittamaan build-kansioon, jos React-projekti löytyi. Lisäksi urls.py-tiedostosta piti määrittää, että sovellus lähettää index.html-tiedoston, kun sen juureen mennään selaimella. Lopuksi kun lokaalisti oli saatu kaikki määritykset valmiiksi tuotantoversiota varten, siirrettiin se Herokun CLI-työkalulla pilvipalvelimelle.

4 Valmiin projektin esittely

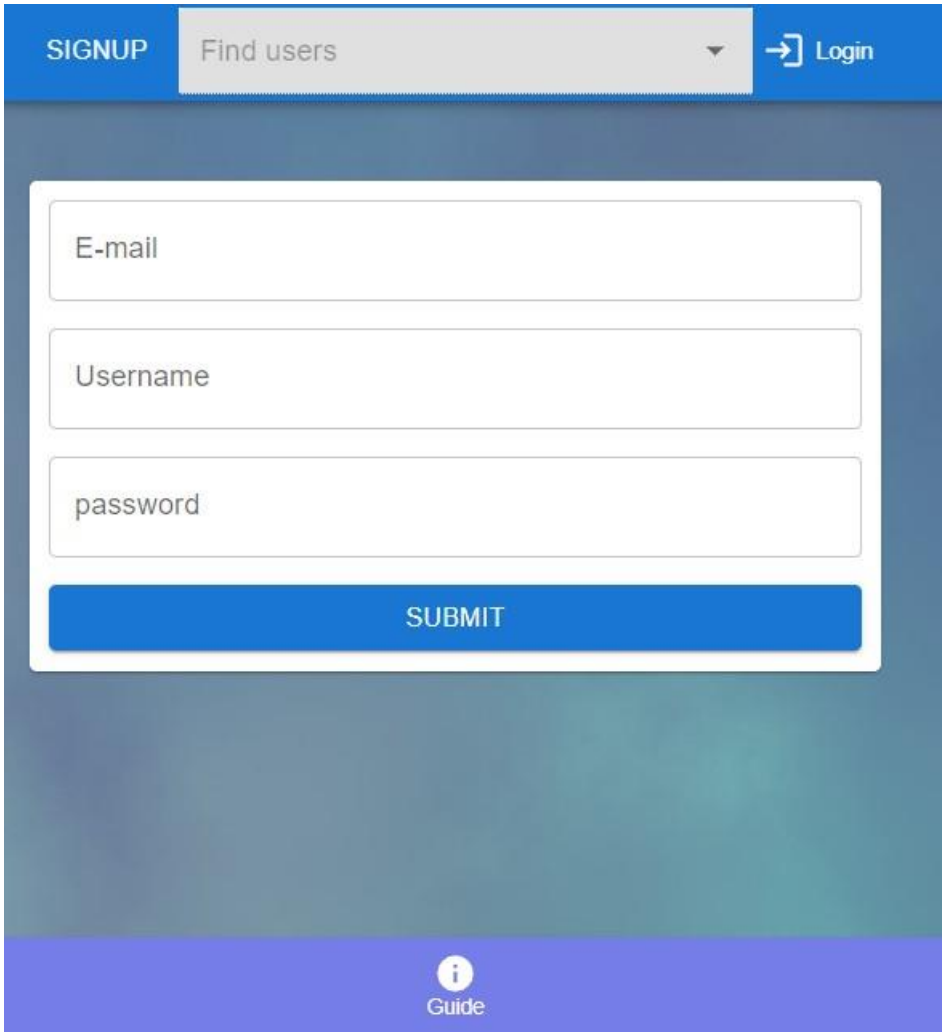
Sovellukseen mentäessä ensimmäisen kerran, aukeaa näkymä, jossa on käyttöohjeet (Kuva 7). Sovelluksessa ei voi tehdä mitään ilman, että siihen luo oman käyttäjän. Edes hakukentästä ei pysty etsimään käyttäjiä, ennen kuin on rekisteröitynyt sivulle. Ohjeissa

opastetaan luomaan ensimmäisenä oma käyttäjä signup-sivulta, joka löytyy ylemmästä navigointipalkista (Kuva 8).

Kuva 7. Sovelluksen etusivu, jossa on käyttöohjeet.



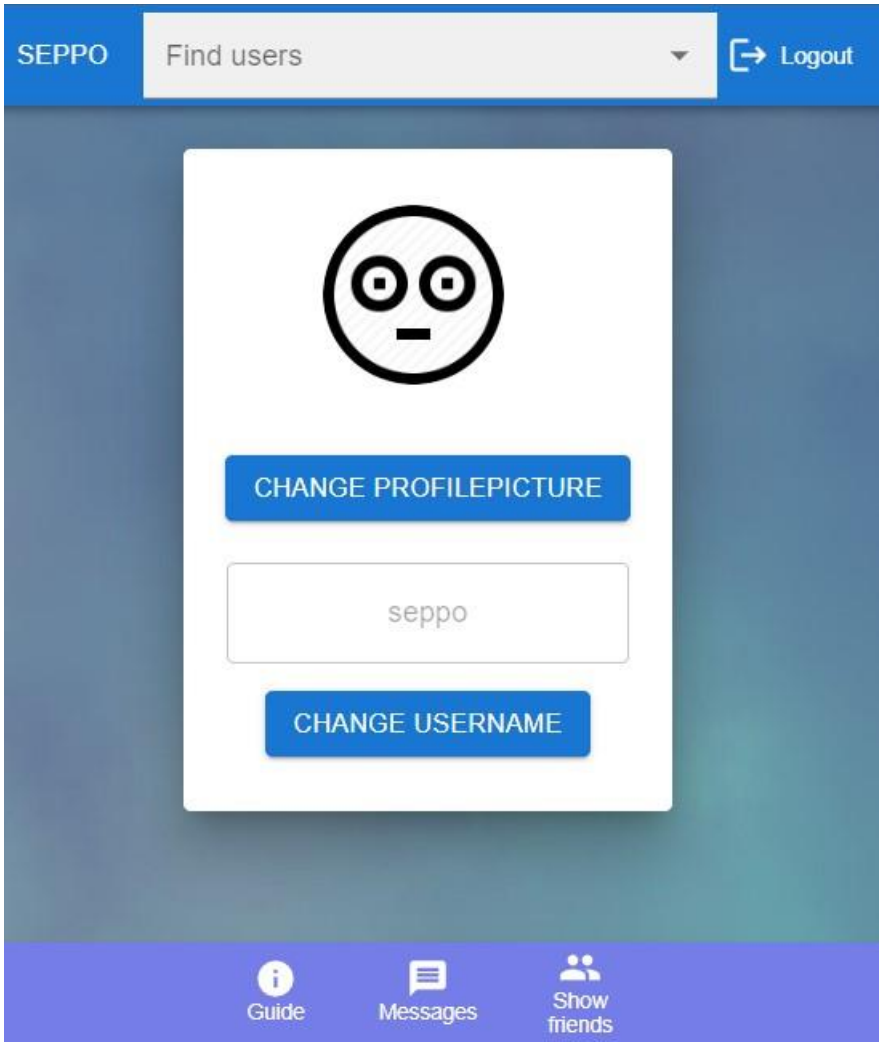
Kuva 8. Käyttäjän luonti näkymä.



Uuden käyttäjän luonnin jälkeen ohjaa sivu vielä sisäänkirjautumisen näkymään. Sisäänkirjautumisen jälkeen aukeaa mahdollisuus etsiä seurattavia kavereita, kirjoittaa viestejä ja muokata omaa profiilikuvaansa. Sisäänkirjautuneesta käyttäjästä jää merkintä selaimen muistiin, jolloin käyttäjän ei tarvitse seuraavalla kerralla kirjautua uudestaan. Selaimeen tallentuu Django-sovelluksen generoima yksilöllinen JSON Web Token, jota käytetään käyttäjän tunnistamisessa, kun selain on yhteydessä sovelluksen palvelimeen.

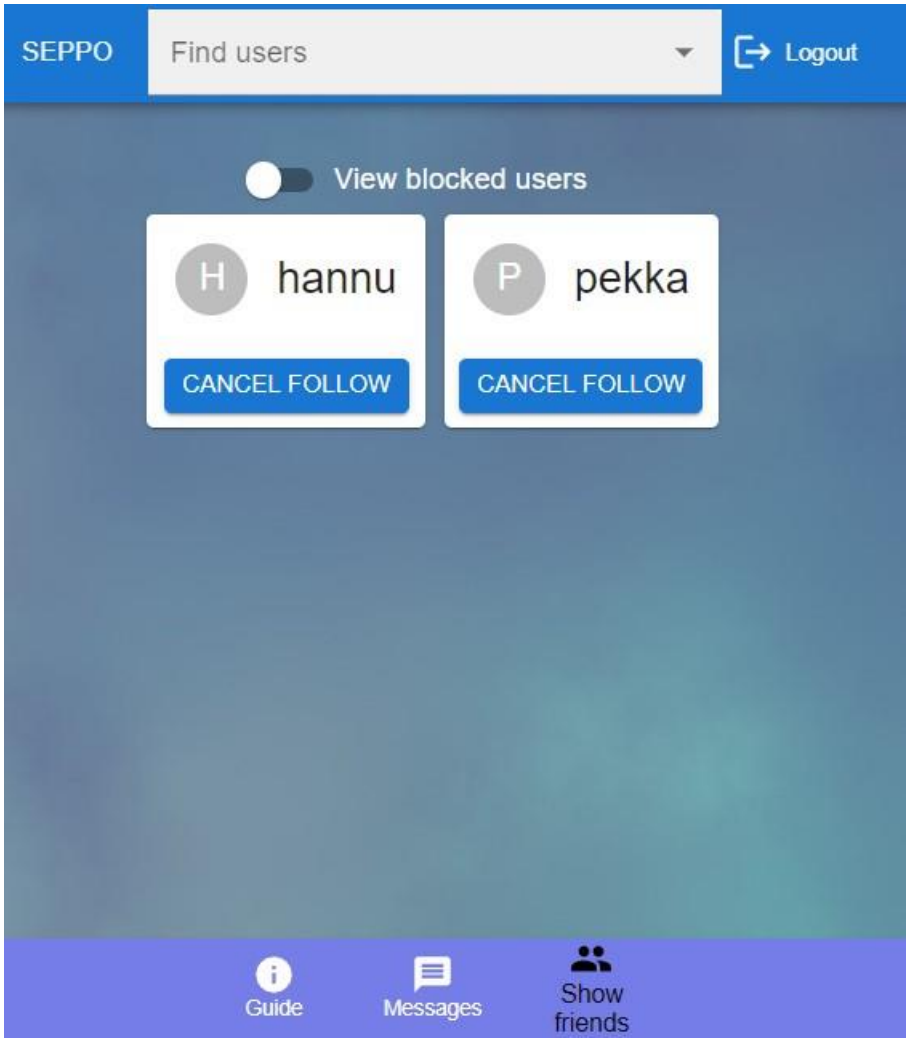
Sisäänkirjautuneen käyttäjänimi tulee näkyville navigointipalkin vasempaan yläreunaan. Siitä klikkaamalla aukeaa sivu, jossa voi muokata omaa käyttäjänimeään ja lisätä oman profiilikuvan (Kuva 9). Mikäli profiilikuvaa ei lisätä, näkyy profiilikuvan paikalla käyttäjänimen ensimmäinen kirjain.

Kuva 9. Käyttäjätietojen muokkaus näkymä.



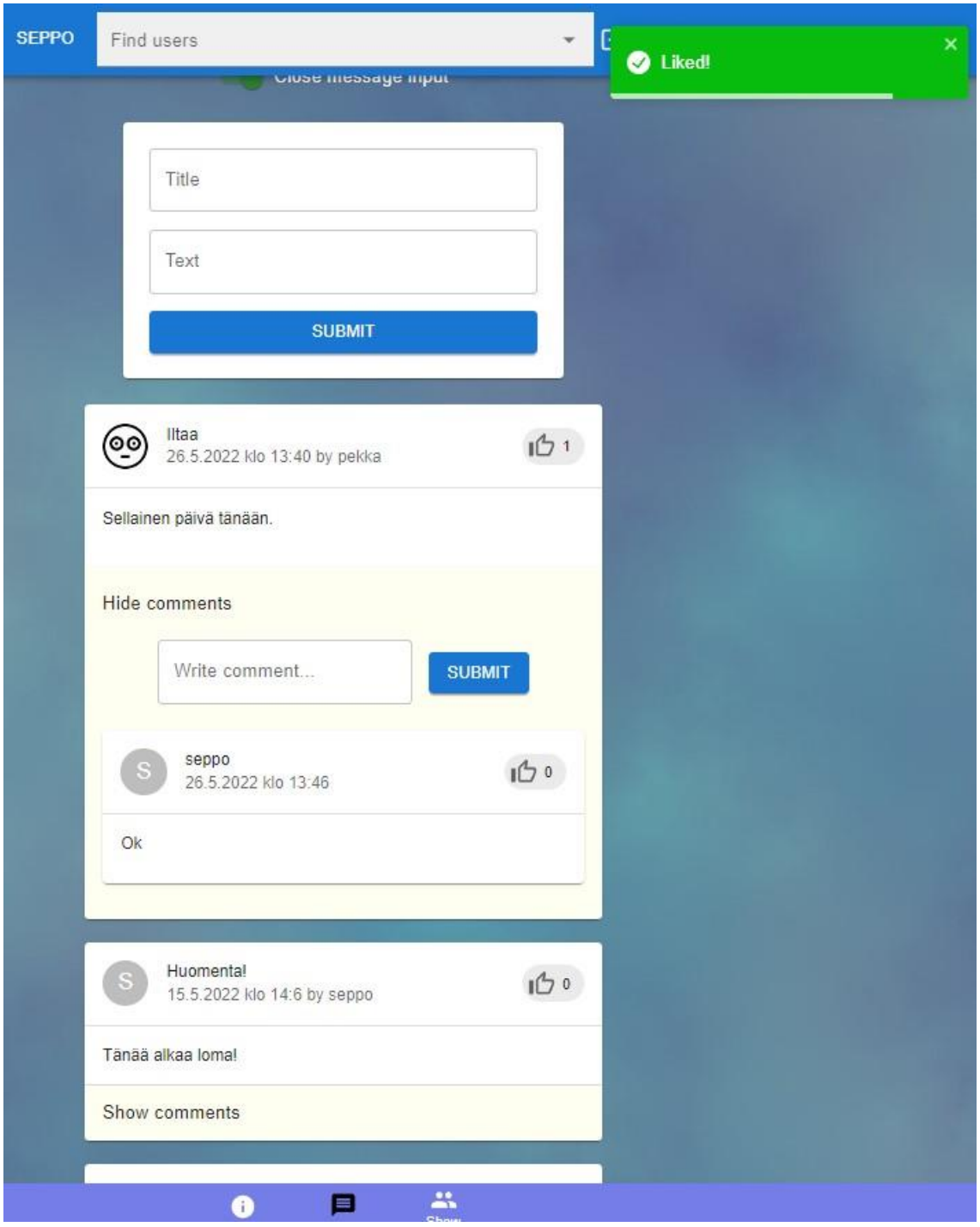
Yläreunan navigointipalkista löytyy hakukenttä, josta voi etsiä seurattavia käyttäjiä. Kenttään täytyy syöttää vähintään yksi kirjain, jolloin se listaa käyttäjänimiä, joista kyseinen kirjain löytyy. Listalta löytyvää nimeä klikatessa aukeaa modaalinäkymä, josta voi joko blokata tai seurata käyttäjää. Mikäli käyttäjä päättää seurata toista henkilöä, näkyy sen jälkeen hänen lähettämänsä viestit seuraajan messages-sivulla. Blokkauksen jälkeen puolestaan blokattu henkilö ei voi seurata blokkaajaa ollenkaan, ennen kuin blokkaus on poistettu. Alareunan navigointipalkista löytyy show friends-linkki, joka vie seurattujen ja blokattujen käyttäjien näkymään (Kuva 10). Siellä voi halutessaan poistaa seuraamiset ja blokkaukset.

Kuva 10. Seurattujen- ja blokattujen käyttäjien näkymä.



Messages sivu on sovelluksen pääsivu, jossa kaikki oleellinen tapahtuu (Kuva 11). Siellä näkyy listattuna allekkain uusimmasta vanhimpaan kaikki käyttäjän ja seurattujen henkilöiden viestit. Sivulla näkyy kerrallaan kymmenen viestiä ja muut loput sivuttuvat eri sivuille. Yksi viesti sisältää tekstin ja otsikon, sekä kommenttikentän. Se on oletuksena piilossa, mutta sen saa nappia painamalla näkyville. Kaikkia viestejä ja kommentteja voi tykätä, paitsi omia. Oikeasta yläreunasta ilmestyy muutaman sekunnin näkyvä ilmoitus aina, kun käyttäjä tekee jonkun toiminnon. Esimerkiksi tykkäysnappia painettaessa ilmestyy vihreä ”Liked!”-tekstillä varustettu ilmoitus (Kuva 11). Mikäli tykkäystä painaa uudestaan, poistuu silloin tykkäys ja siitä tulee ilmoituksena keltainen viesti. Epäonnistuneesta kirjautumisesta tulee punainen ilmoitus.

Kuva 11. Messages sivu.



5 Yhteenveto

Opinnäytetyön tavoitteeseen päästiin, sillä lopputuloksena syntyi toimiva sosiaalisen median web-sovellus. Sovelluksessa käyttäjä pystyy autentikoitumaan, seuraamaan ja blokkaamaan käyttäjiä, kirjoittamaan viestejä ja kommentteja sekä lisäämään itsellensä profiilikuvan.

Aikatauluna oli, että projekti olisi valmis kolmen kuukauden sisällä. Aikataulussa pysyttiin ja työ valmistuikin reilussa kahdessa kuukaudessa.

Opinnäytetyön tekeminen opetti, kuinka rakennetaan web-sovellus. Rakentamisen prosessiin kuului monta osaa, kuten suunnittelu, palvelinpuolen- ja selainsovellusten rakentaminen ja julkaiseminen tuotantoon. Sosiaalisen median sovelluksessa tietokantataulujen välisten suhteiden saaminen toimiviksi oli haasteellisinta, mutta sekin saatiin tehtyä. Käytetyt teknologiat olivat ennestään tuttuja, mutta niistä tuli uutta oppia työtä tehdessä.

Web-sovellusta on myös mahdollista jatkokehittää. Sivustosta tulisi monipuolisempi, jos siihen lisäisi kuvien tallentamisen omaan galleriaan tai viestien liitteeksi. Tiedostojen tallentaminen lisäisi kuitenkin tarvetta ottaa käyttöön isommat resurssit pilvipalveluista. Toinen hyvä lisä sovellukselle olisi toisten viestien tai kommenttien lainaus. Silloin olisi selkeämpää vastata toisen käyttäjän viestiin. Projektiin olisi mahdollista lisätä myös reaaliaikainen uusien viestien renderöityminen sivulle Socket.IO-protokollalla.

Lähteet

Atlassian. (2022). *What is version control?* https://www.atlassian.com/git/tutorials/what-is-version-control

Chiarelli, A. (2018). *Beginning React : Simplify Your Frontend Development Workflow and Enhance the User Experience of Your Applications with React.* Packt Publishing, Limited. https://ebookcentral-proquest-com.ezproxy.hamk.fi/lib/hamk-ebooks/reader.action?docID=5477665&query=

cs.helsinki. (2022). *History of SQL*. https://www.cs.helsinki.fi/u/laine/tuelip/sql_material/sql_history.html

Django. (2022). *Django REST Framework Documentation*. https://www.django-rest-framework.org/

ElHousieny, R. (30.1.2021). *What Is Redux?* https://medium.com/swlh/what-is-redux-b16b42b33820

Freecodecamp. (28.4.2018). *Meet Material-UI — your new favorite user interface library*. https://www.freecodecamp.org/news/meet-your-material-ui-your-new-favorite-user-interface-library-6349a1c88a8c/

Google. (9.5.2022). *Cloud Storage for Firebase*. https://firebase.google.com/docs/storage

Gray, J. (1.4.2006). *To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem*. https://www.microsoft.com/en-us/research/publication/to-blob-or-not-to-blob-large-object-storage-in-a-database-or-a-filesystem/

Hat, R. (8.5.2020). *What is a REST API?* https://www.redhat.com/en/topics/api/what-is-a-rest-api

Hillar, G. C. (2018). *Django RESTful Web Services : The Easiest Way to Build Python RESTful APIs and Web Services with Django.* Packt Publishing, Limited. https://ebookcentral-proquest-com.ezproxy.hamk.fi/lib/hamk-ebooks/detail.action?docID=5254606

Mentormate. (2022). *What is Heroku and What is it Used For?* https://mentormate.com/blog/what-is-heroku-used-for-cloud-development/

Mozilla. (2022 -a). *About JavaScript*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript

Mozilla. (2022 -b). *CSS: Cascading Style Sheets*. https://developer.mozilla.org/en-US/docs/Web/CSS

Mozilla. (2022 -c). *HTML basics*. https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics

Mozilla. (2022 -d). *SPA (Single-page application)*. https://developer.mozilla.org/en-US/docs/Glossary/SPA

Mustafeez, A. Z. (31.5.2022). *What is Visual Studio Code?* https://www.educative.io/edpresso/what-is-visual-studio-code

Oracle. (2022). *What is a Relational Database (RDBMS)?* https://www.oracle.com/database/what-is-a-relational-database/

Pandit, N. (10.2.2021). *What And Why React.js*. https://www.c-sharpcorner.com/article/what-and-why-reactjs/

Ström, C. (9.7.2019). *React pähkinänkuoressa*. https://joinex.fi/react-pahkinankuoressa/