System documentation
Dumpster Finder

Tore Bergebakken Jon Åby Bergquist Helene Yuee Jonson

Spring 2021

Task 17
version 1.2

Audit history

Date	Version	Description	Author
12.01.2021	0.1	Creation of document	Helene Y. Jonson
02.05.2021	0.2	WIP draft	Tore Bergebakken
10.05.2021	1.0	First complete draft	Tore Bergebakken, Jon Åby Bergquist
18.05.2021	1.1	Final version	Tore Bergebakken, Jon Åby Bergquist
20.05.2021	1.2	Final version, now with class diagrams	Tore Bergebakken

Contents

1 Introduction

This document was written by three students at the Norwegian University of Science and Technology as part of our bachelor thesis in computer engineering. Our assignment was to create an application where dumpster divers can share information about different dumpsters with other dumpster divers. The purpose of this document is to explain how the system was structured and other aspects of its design. It will explain the project's architecture, how its files are structured, detail the database schema, explain the different endpoints in the API, go through the different means of security enhancements, provide instructions for installing and running the project, explain where the source code documentation can be found and how to generate it, and go through the continuous integration pipeline.

2 Architecture

As Figure 1 shows, our system consists of a mobile app and a server split into several containerized services: an API server, a file server and a database, connected to the outside world through a reverse proxy.

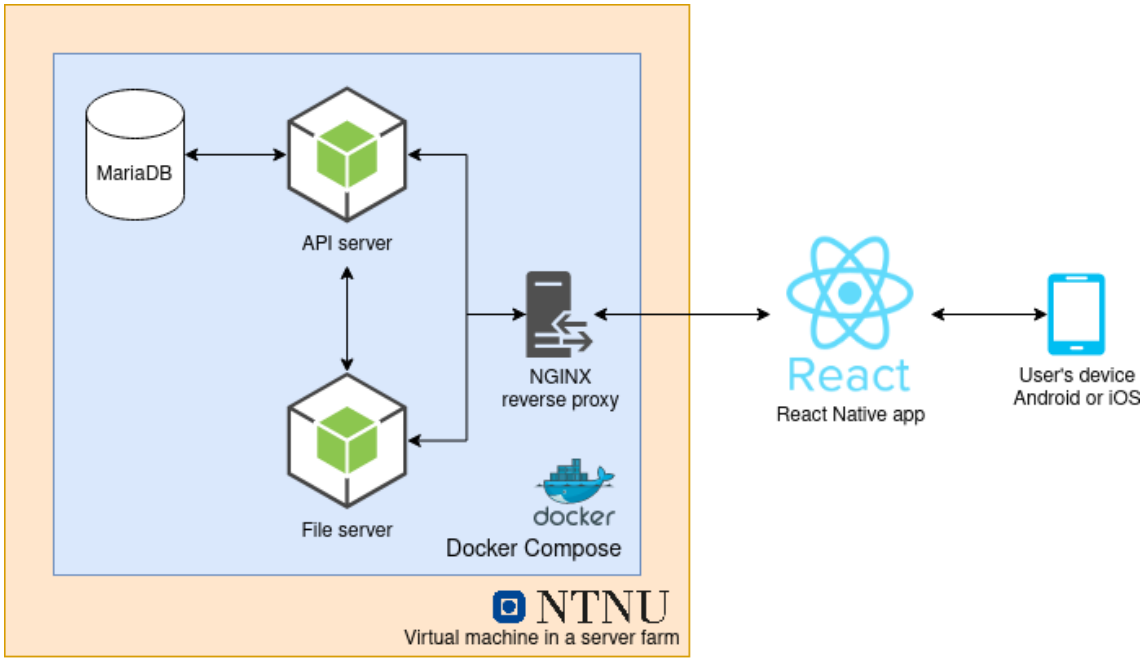

Figure 1: Architecture diagram

3 Project structure

The root folder contains a README and some useful things.

3.1 backend

Here, all components of our backend are configured.

- `docker-compose.yml`: Defines the set of containers that the backend consists of, and any links between them
- `renew_certs.sh`: Script for renewing SSL certificates for the sake of HTTPS support
- `dumpster.service`: systemd unit file
- `Makefile`: Contains scripts that set up the database and more
- `tables.sh`: Related to the Makefile

3.1.1 api

- `@types`: Some extra TypeScript definitions
- `config`: Files that extract environment variables and set up some object (like our Sequelize instance and our logger), and a file that exports a function for test setup
- `daos`: Data Access Objects that serve as wrappers around Sequelize cards, with some transactional logic and error reporting
  - `__tests__`: Unit tests for these DAOs
- `middleware`: Express middleware functions that handle things like tokens, errors and rate limiting
- `models`: Sequelize models that correspond to tables in the database
- `node_modules`: The folder where all our third-party dependencies end up
- `routes`: API routes, exported as Express routers that are connected to their respective paths in `server.ts`
- `types`: TypeScript interfaces for objects that the API server sends and receives
- `utils`: Utilities for hashing and generating user IDs and handling tokens
- `validators`: Joi validators that check that data the API receives fits with a certain schema
- `index.ts`: An entry point that imports and starts the Express server
- `server.ts`: Connects all routes to their paths, as described in section 6
- `package.json`: The usual file where our dependencies and commands are listed

- `.env`: Contains *secret* environment variables necessary for letting the API know things like which port it should listen on
- `Dockerfile`: Defines how the API server's container should be set up

3.1.2 `db`

Database scripts.

- `init.sql`: Defines the structure of the database and a necessary function
- `constants.sql`: Adds categories and types of stores and dumpsters
- `data.sql`: Inserts test data that is not meant to be used in production
- `Dockerfile`: Defines how the database container should be set up
- `Makefile`: For quick setup and refresh of the database while developing
- `setup.sh`: Related to the Makefile

3.1.3 `nginx`

Configuration files for the NGINX proxy, which is responsible for routing requests to the API or photo server depending on the path, and enabling HTTPS.

- `web`: Web content
- `nginx.conf`: The configuration file
- `Dockerfile`: Defines the NGINX container, and replaces `example.com` in the config file with the actual domain name

3.1.4 `pics`

Picture server, the place where pictures uploaded through the app are sent, and where clients will receive images from. Roughly the same as `api`, but without any Sequelize models. No need to elaborate.

- `@types`
- `config`
- `controllers`: Inappropriately named; contains a file with some functions used when a photo is uploaded
- `middleware`
- `node_modules`
- `routes`

- `types`
- `validators`
- `index.ts`
- `server.ts`
- `package.json`
- `.env`
- `Dockerfile`

3.2 `frontend`

The files that make up the React Native client for our backend.

- `@types`: Extra TypeScript definitions
- `assets`
  - `fonts`: Extra non-system fonts
  - `images`: Map marker icons and placeholder images
- `components`: React components (or widgets) that are *not* entire screens themselves
  - `basicComponents`: Simple components
  - `cards`: Components with a card at the top level – for displaying information about a single entity
  - `compoundComponents`: More complex components
  - `dumpsterInfo`: Components that display information about a dumpster
  - `map`: Components related to maps
  - `modals`: Components that are used as modals (pop-ups)
  - `selects`: Various kinds of dropdown selection components
  - `settings`: Components used on the settings screen
  - `textComponents`: Components that display some static text
- `constants`: Various constant values
- `coverage`: Produced by Jest on test runs
- `docs`: Documentation generated by typedoc
- `hooks`: Custom hooks – functions called in React components that set up some functionality
- `models`: Interfaces and classes that describe our data models

- `navigation`: Files that set up the navigation structure used by React Navigation – mostly from the template we used
- `redux`: Handles global application state
  - `slices`: Redux slices that take care of specific parts of the application state
  - `store.ts`: Combines the state of each slice into a whole
  - `tokenInterceptors.ts`: Axios interceptors that handle tokens in requests – placed here because they use Redux state
- `node_modules`: The folder where all our third-party dependencies end up
- `screens`: Entire screens in the application
  - `addDumpster`: Screens that are part of the process of adding a dumpster
  - `dumpsterInfo`: Screens that display and/or edit info about a dumpster
  - `main`: Top-level screens — the four you get to through the icons in the tab bar
  - `photo`: Screens that display, take or upload photos
- `services`: Each service handles calls to a specific part of the API
- `translations`: JSON files with translations for all text displayed in the application
- `utils`: Miscellaneous utilities used in some parts of the application – date formatting, distance calculation, dumpster filtering and error reporting
- `app.config.js`: Expo configuration
- `App.tsx`: Contains the top-level component of the app. Deals with (re)setting state when the app loads, and several other important tasks.
- `i18n.tsx`: Sets up i18next, which handles translations
- `package.json`: The usual file where our dependencies and commands are listed
- `types.tsx`: Types used by React Navigation
- `.env`: Environment variables like the URL to the API and photo server

4 Class diagram

Our code base contains very few classes. We have service classes, *some* classes for data models and classes for models in our ORM. In this section, we provide diagrams to visualize some of them.

The data models in the frontend are shown in Figure 2. Very few of them have any methods at all, and their reason for being classes in the first place was to translate strings with dates to actual date objects. Each of them have a constructor that takes data from the API as raw JavaScript objects converted directly from JSON, which does not support the Date type.

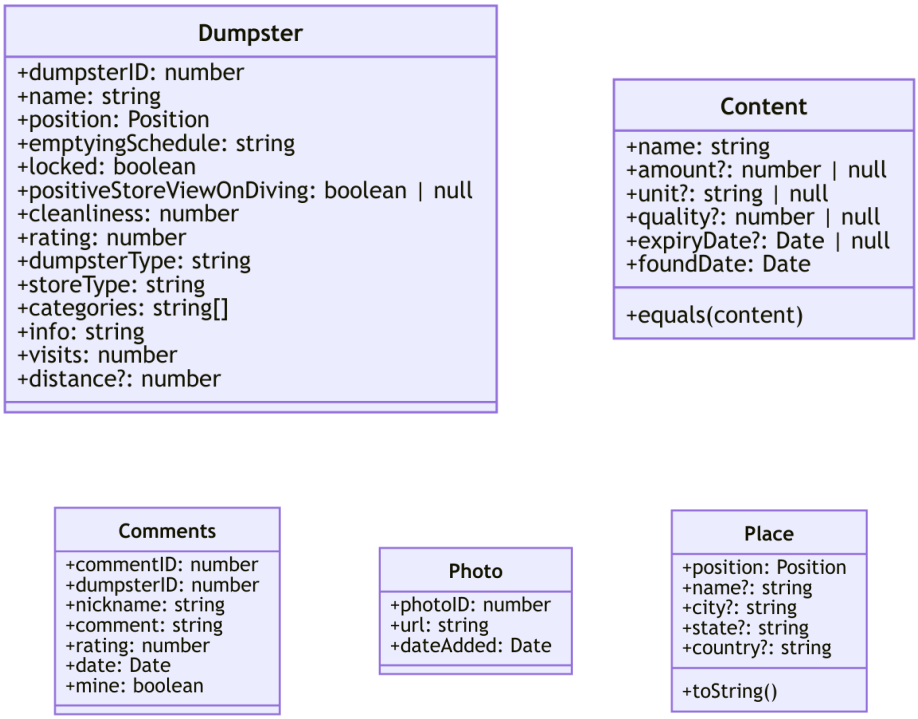

Figure 2: Frontend data classes

The frontend's service classes, which send requests to the API, are shown in Figure 3. Since the only thing they have in common is that their constructor takes an instance of an object that performs HTTP requests, it did not seem necessary to have them inherit an abstract class.

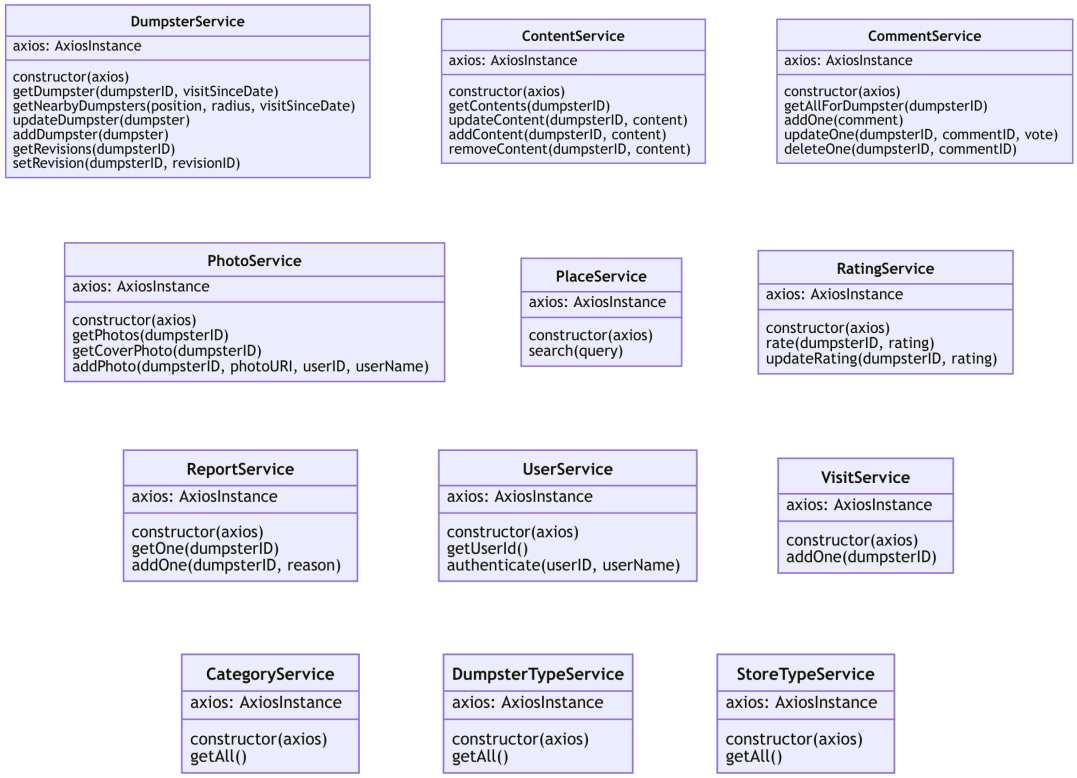

Figure 3: Service classes

Sequelize, our ORM of choice, requires classes that act as definitions of tables in the database. Since most of the Sequelize models just reflect the database model (Figure 5), we only show a few of the model classes in Figure 4. Each model inherits Sequelize's Model class and implements an interface with attributes. The latter is necessary for correct type hints with TypeScript.

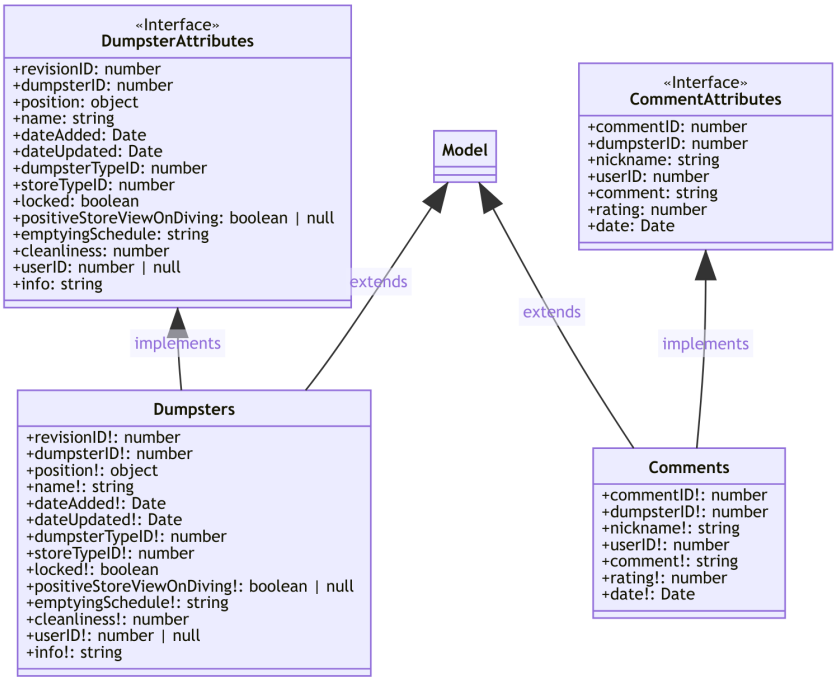

Figure 4: Sequelize models

5 Database model

Figure 5 shows our database model. We visualize complex junction tables for many-to-many relations with a line going down to the relation it describes, but make exceptions in cases where it makes more sense to treat the table as its own, separate entity — and in those cases, the entities have custom primary keys.

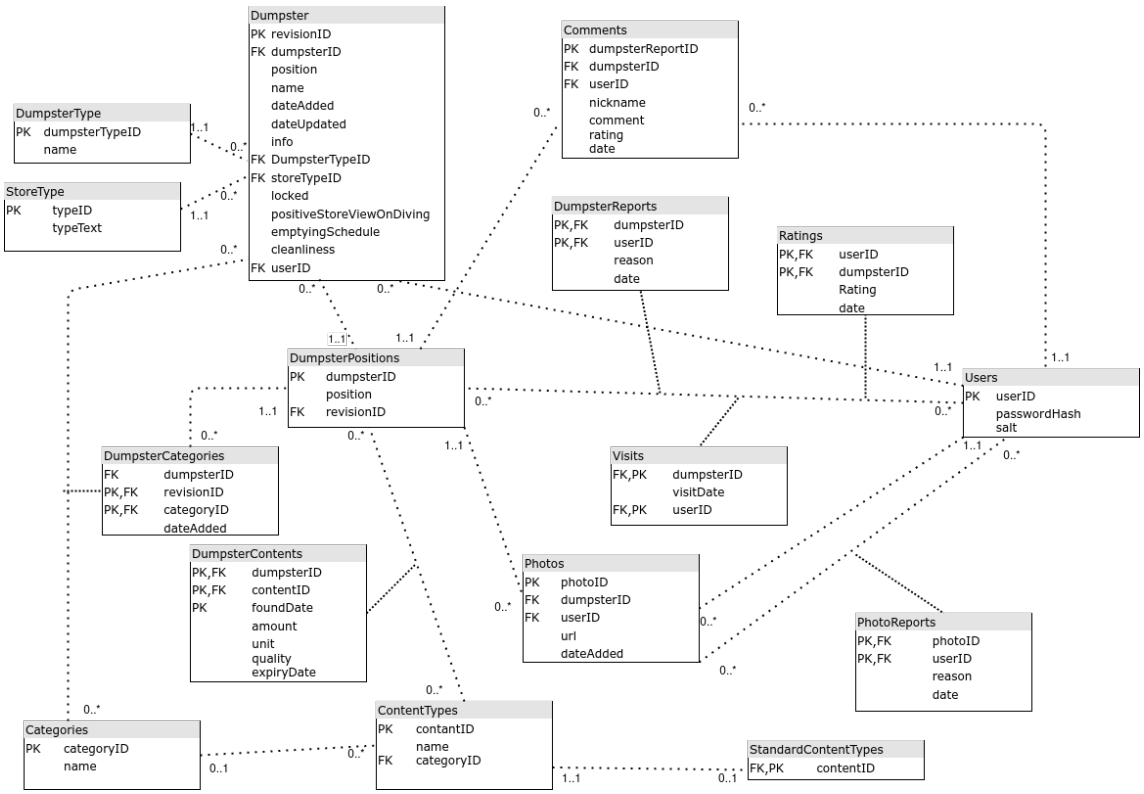

Figure 5: ER diagram showing our product's database schema

6 Server services

This section contains a brief summary of the resources provided by our REST servers. We recommend reading the OpenAPI documentation found at `/api/spec` and `/pic/spec` on (development) instances of our backend.

6.1 API server

The API server handles storing, modification and retrieval of regular information, excluding actual image data. Its endpoint layout is shown in figures 6 to 8. All paths are prefixed with `/api`.

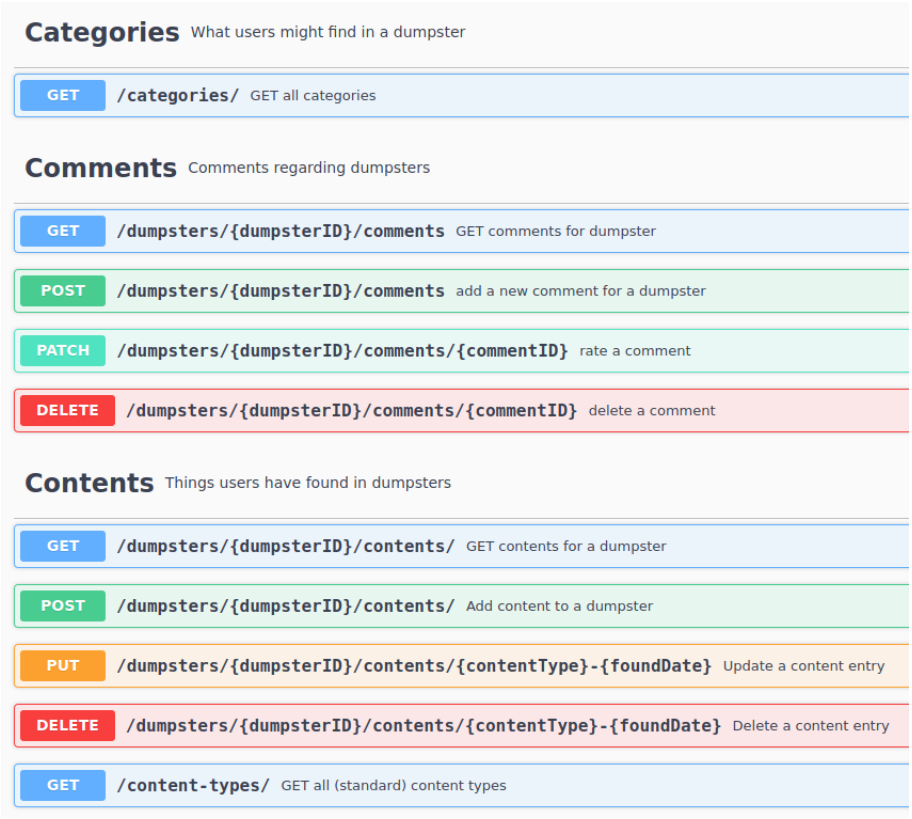

Figure 6: API endpoints, part 1

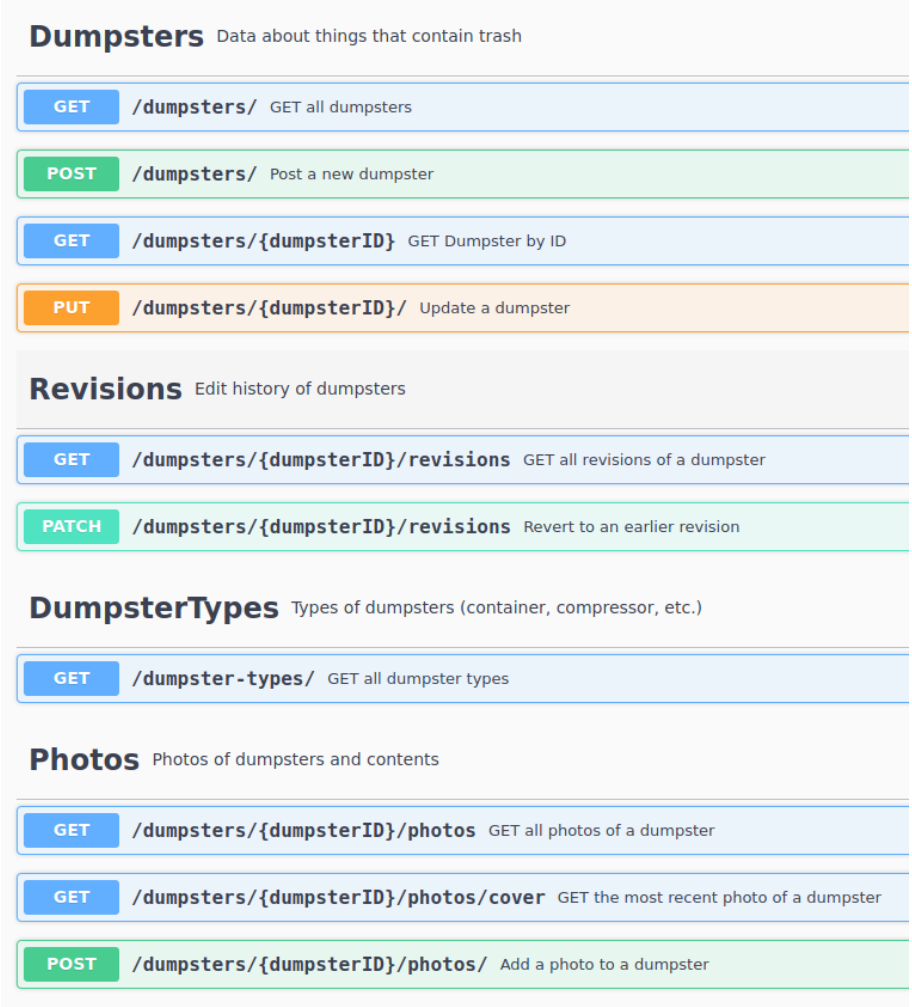

Figure 7: API endpoints, part 2

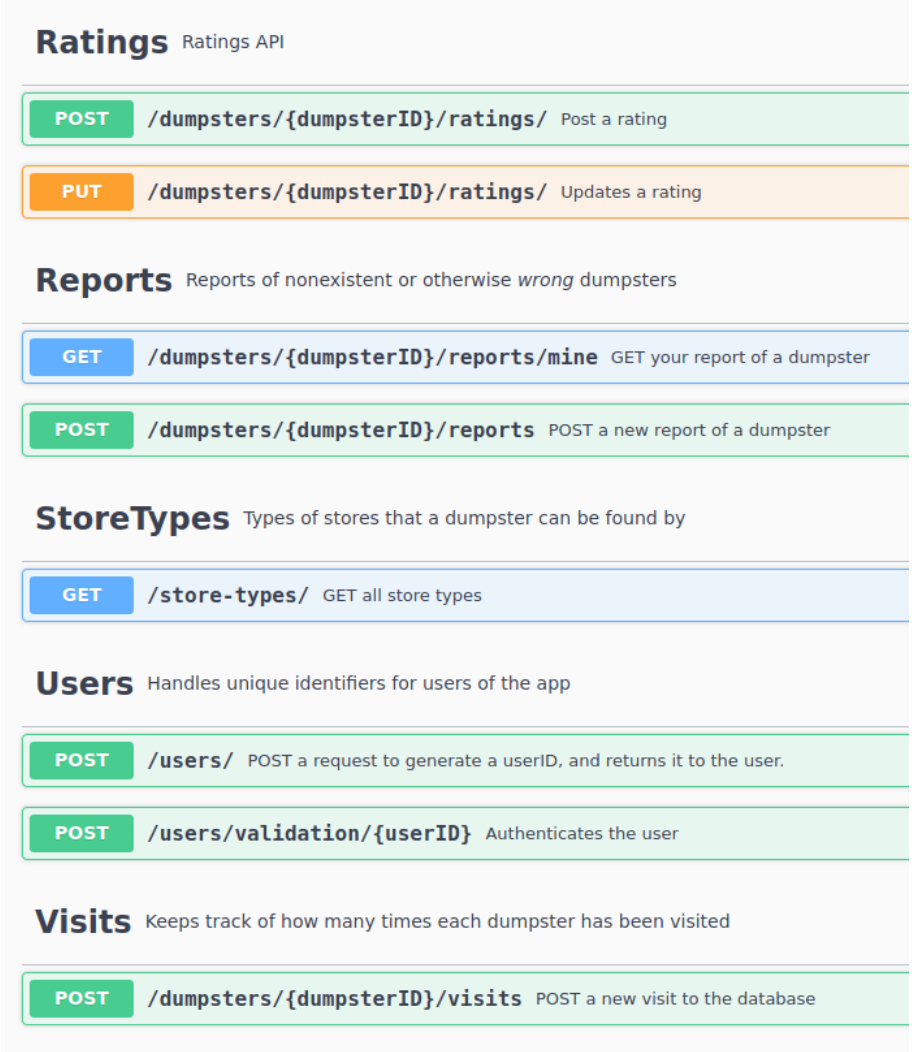

Figure 8: API endpoints, part 3

`/categories/` All known categories, GET only

`/content-types/` Standard types of contents, GET only

`/dumpster-types/` Types of dumpsters, GET only

`/store-types/` Store types, GET only

`/dumpsters/` Information about specific dumpsters

`/dumpsters/xx/comments/` Thoughts about a dumpster

`/dumpsters/xx/contents/` Things found in a dumpster

`/dumpsters/xx/photos/` Photos of a dumpster or its contents

`/dumpsters/xx/photos/cover/` Cover photo to display if there's only room for one image

`/dumpsters/xx/reports/` Reports of a dumpster – it might not exist

`/dumpsters/xx/revisions/` Revision history for a dumpster

`/dumpsters/xx/visits/` Accepts POST requests to register visits to a dumpster

`/users/` POST to get your own user ID

`/users/validation/xx/` POST to authenticate and receive a token

6.2 Photo server

The API is shown in Figure 9. It stores and distributes actual image files. Relatively uncomplicated, with only *two* endpoints.

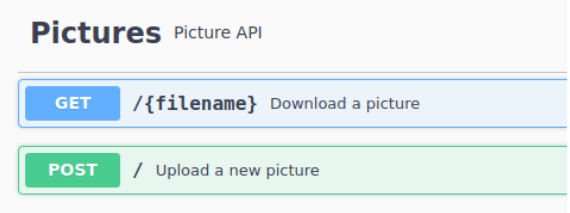

Figure 9: Photo API

`/pic/` POST to add a photo

`/pic/xx.[jpg|png]` GET to download a photo

7 Security

7.1 Encrypted data transfer

The app communicates with our server over HTTPS, which ensures that no data except the host name or IP is transmitted in plaintext form — it ensures some degree of confidentiality. Our reverse proxy uses Certbot to get signed SSL certificates from Let's Encrypt [1]. We used Mozilla's suggested set of ciphers for moderately broad compatibility [2] and got a good rating in a SSL-Labs scan, see Figure 10.

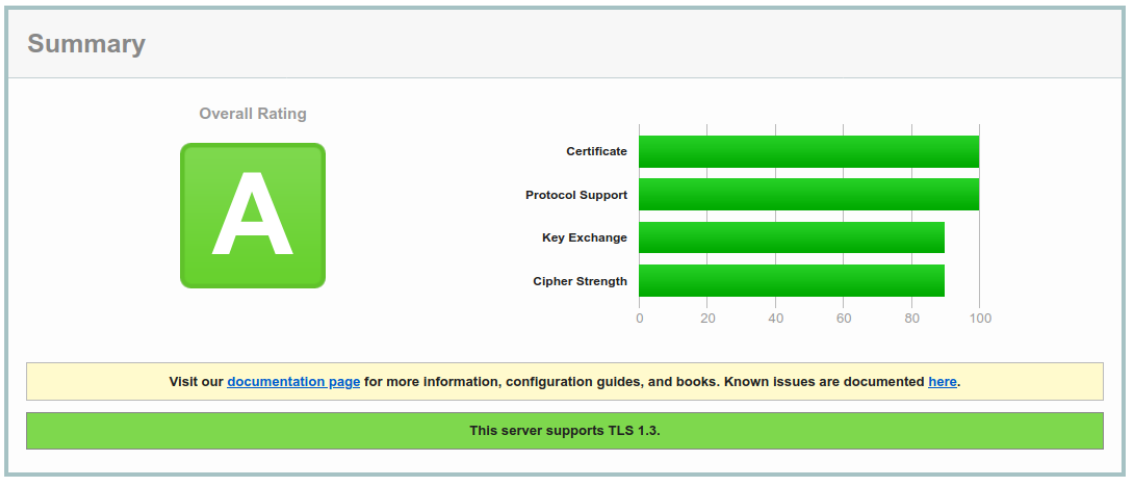

Figure 10: SSL-Labs' rating of our server

7.2 Handling of user IDs

Because we didn't want to store any user information, we needed a way to identify users that wasn't email/username and password. The problem with using an email or username is that it can be used to identify the user, because people reuse usernames, or use usernames that might contain information that could potentially identify the user. We also wanted a system where you could potentially change device and still keep your old account. Something like a UUID would be difficult for the user to remember and tedious to write down. A simple numeric auto-increment ID on its own would be useful for identifying a user, but offers very little in terms of security, since it's fairly easy to guess. We ended up using a system similar to the seed phrases that cryptocurrency wallets use, which generates (in our case) 6 words from a list of 7776 words. Crypto wallets normally use 12 words, but the need for security isn't as important in our app, since all you really can do with your userID is delete images and comments that were made by your user. There is also the possibility of being able to track the user's past locations, since some information like comments, pictures and visits could potentially get accessed — however, even if the user was compromised it would be quite difficult to know who the user belongs to, since everything is generated pseudo-randomly. So we felt that 6 words struck a nice balance between being relatively easy to remember/write down, and still providing security.

We originally made it so that the first 4 words would be hashed naively to find the correct entry in the database and the whole 6 words was hashed with a random salt. This solution was more than enough in terms how many users could exist ($7776^4 = 3.65615844006 \cdot 10^{13}$), but left the security lacking since the naive hashes are vulnerable to rainbow tables, and if the hash was solved, you would only need $7776^2 = 60466176$ attempts to guarantee access to the user. So we removed this 4-word feature, and replaced it with a numeric ID with auto-increment to identify the row in the table as well as the 6 words as input to validate the user. There is currently no way to transfer the userID from an old phone to a new one, but it is one of the future features that are planned.

7.3 Tokens

A simple JWT (JSON Web Token) system was implemented to save time on authenticating users for every endpoint call that required validation. It is just a simple system that checks the database if the 6 word + userID combination is valid and creates a token based on the userID + the time it expires using a secret. The server then receives this token for all post requests, update requests or other requests where userID is relevant. The token is validated using middleware. There is a 30 minute active period as well as a one hour grace period where a new token is generated automatically and sent back to the user. Naturally it checks if the secret is correct when decoding the token.

The token does not really add much in terms of security, it just saves time that would have been used when authenticating the 6 words + userID. Tokens do not compromise the security much either, there is a risk of it being caught and used, and theoretically there is a risk of it being reused with the grace period constantly, however HTTPS does scramble the token so this is unlikely. There is no real risk of the secret being guessed, as it is a 128 character long random string, so people generating their own token is not really feasible.

7.4 SQL injection

Our API server uses Sequelize to access the database, a library which handles escaping of raw values in SQL queries well [3]. We ran some quick tests with sqlmap, a tool for testing how vulnerable a server is to SQL injections. No vulnerabilities were discovered.

7.5 Input validation

We made sure to validate all input the API receives through its endpoints, and validate the user's input in the app before it is sent to the API. Instead of writing our own functions for it, and potentially running into bugs, we used Joi, which simplified the process a lot.

7.6 Rate limiting

Another important concern in security is *availability*. To prevent our server from being overloaded by requests, we added rate limiting functionality, so that users (or bots) can only send

a limited number of requests from a given IP within a given timeframe.

7.7 Path traversal

This *could* have been an issue in the file server — however, each request to the GET endpoint has to have a filename that matches `[a-zA-Z0-9]+[.](jpg|png)`. There is simply *no way* for an attacker to traverse the server's folder structure when this restriction is in place. The only feasible way would be if that part of the Express library did not work properly.

7.8 Sensitive data exposure

The only piece of *potentially* sensitive information in this application, would be the anonymous identifier that could *theoretically* be used to identify a person, though only together with the data connected to it and a significant amount of external location data. This identifier connects users and data like ratings, comments and dumpster visits. However, the API *should not* reveal other people's identifier *in any case*.

7.9 Server security

We configured our firewall to prevent access to any ports other than those for SSH and the NGINX server itself. Password login was disabled, requiring your SSH keys to match those on the server in order to access it.

8 Installation and running

8.1 Running the backend as a developer

8.1.1 Prerequisites

For all parts of the system except the database, you need a (recent) version of npm installed. It is required for installing packages for Node.js and running the scripts that start the server and the like.

For running the database, you need to install MariaDB, e.g. by running `apt install mariadb-server` on a Debian-based Linux distro.

8.1.2 Database

The following instructions assume you are standing in the `backend/db` folder.

To build the image and start it:

```
make maria
```

Alternatively, to start a local MariaDB server on WSL:

```
make wsl
```

To run the setup script against the database, which creates the necessary tables and procedures:

```
make tables
```

To fill the tables with test data:

```
make data
```

NB: You might need to wait for a few seconds before executing this command. Additionally, *make sure* to have a `.env` file in the backend folder.

To clean up the containers:

```
make clean
```

Note for Windows users, or others who run into trouble: If these instructions do not work, try cloning the repo with LF line endings instead of Windows' default CRLF – or just look at the `Makefile` and setup script (`setup.sh`) and adapt the commands to your needs.

8.1.3 API server

The following instructions assume you are standing in the `backend/api` folder.

Create a `.env` file containing something like the following:

```
API_HOST=127.0.0.1
API_PORT=3000
DB_HOST=127.0.0.1
# and so on
```

`...` as specified in the `.env.template` file.

Note that you *must* use `127.0.0.1` and not `localhost` as the database host if you run it in a Docker container.

Depending on where you run the app, you may need to change the `PHOTO_URL` variable to match the IP the app makes contact with.

Run `npm install` to install dependencies, then run `npm start` to start the API server. *Make sure that your database is running*, otherwise the API server will crash after 10 unsuccessful connection attempts.

8.1.4 Photo server

The following instructions assume you are standing in the `backend/pics` folder.

Create a `.env` file containing something like the following:

```
PIC_PORT=3000
PIC_HOST=localhost
API_URL=http://localhost:3000/api/
# and so on
```

`...` as specified in the `.env.template` file.

Run `npm install` to install dependencies, then run `npm start` to start the picture server.

8.2 Backend deployment

Acquire a server with a recent Linux distro. Install rsync, Docker and Docker Compose, e.g.:

```
apt install rsync docker.io docker-compose
```

You should *set up SSH keys and disable password-based authentication, see section 8.2.1.*

Create a user (e.g. `dumpster`) without administrative privileges, but in the `docker` group:

```
useradd --create-home dumpster
groupadd docker
usermod -aG docker dumpster
```

Let `$SERVER_IP` be your server's IP in the following snippets.

Transfer the contents of your repository to an appropriate folder in the `dumpster` user's home directory: (this assumes you stand in the root directory of the repo)

```
rsync --archive
      --exclude='.git'
      --exclude='node_modules'
      --exclude='.env'
      backend/ "dumpster@$SERVER_IP:dumpster"
```

Copy your API's .env template file and make a dynamic link to it:

```
scp backend/api/.env.template "dumpster@$SERVER_IP:dumpster/api"
ssh dumpster@$SERVER_IP ln -s dumpster/api/.env dumpster/.env
```

Then tweak it to fit the following pattern:

```
HTTPS=true
PROJECT_PATH=/home/dumpster/dumpster
# API server:
API_PORT=3000
API_HOST="<your server's domain or IP>"
NODE_ENV=production
TOKEN_SECRET="<some random, long string>"
# Photo server:
PHOTO_URL="https://<your server's domain or IP>/pic/"
# Database:
DB_NAME=dumpster
DB_USER=root
DB_PASSWORD="<your database password>"
DB_HOST=db
DB_PORT=3306
DB_DIALECT=mariadb
# Certbot:
EMAIL="<email of whoever wants to sign this>"
DOMAIN_NAME="<your server's domain or IP>"
```

Copy over the photo server's .env template

```
scp backend/pics/.env.template "dumpster@$SERVER_IP:dumpster/pics"
```

and tweak it a little as well – it should look like this:

```
PIC_PORT=3000
PIC_HOST=pics
PIC_URL="https://<your server's domain or IP>/pic/"
API_URL=http://api:3000/api/
PIC_FOLDER=/var/uploads/
PIC_MAX_SIZE=10000000
```

Ideally, you'd set up HTTPS for some extra security here, see section 8.2.2 for instructions.

Copy over the `systemd` unit, reload the daemon and start the service:

```
scp backend/dumpster.service \
    "dumpster@$SERVER_IP:.config/systemd/user/"
ssh dumpster@$SERVER_IP systemctl daemon-reload
ssh dumpster@$SERVER_IP systemctl --user start dumpster
```

Wait a few seconds, then create the database tables:

```
ssh dumpster@$SERVER_IP "cd dumpster && make tables"
```

After this, the `.gitlab-ci.yml` file should make GitLab CI perform updates automatically after changes to `develop`. The server should be up and running, accessible from port 443.

8.2.1 SSH hardening

Using SSH keys and disabling password authentication are important security measures you may want to take. Specifically, generate an SSH key for your computer, add the public key to `.ssh/authorized_keys`, make sure you can log in without a password, and finally disable the `PasswordAuthentication` option in the SSH config (and perhaps disable `PermitRootLogin` as well).

You can also install `fail2ban` and run it with a basic configuration like this:
(in `/etc/fail2ban/jail.local`)

```
[DEFAULT]
; a rather strict penalty
bantime = 1d

[sshd]
enabled = true
; since we use ufw, make fail2ban use it too
banaction = ufw
; (this could be a bad idea)
ignoreip = <your server's IP>
```

8.2.2 SSL certificates

The HTTPS setup detailed in this section was influenced by a Digital Ocean tutorial.

In order to secure the connection between the app and your instance of the server, you should acquire an SSL certificate and enable HTTPS. Our setup uses `certbot` to get certificates signed by Let's Encrypt, for no cost at all. You *do* need a server that is available to the outside world, otherwise the servers of Let's Encrypt won't be able to access your server.

Create a Diffie-Hellman key in the `backend` folder:

```
mkdir dhparam
openssl dhparam -out dhparam/dhparam-2048.pem 2048
```

Comment out the second server block in the NGINX config, and uncomment the parts of the first server block that are indicated by other comments. Since you do not yet have a certificate, everything must happen through HTTP. Let's Encrypt needs to be able to query for your challenge file.

```
http {
    # (...)

    server {
        # (...)

        # (uncomment when first acquiring SSL cert)
        root /var/www/html;
        index index.html index.htm index.nginx-debian.html;

        location ~ /.well-known/acme-challenge {
            allow all;
            root /var/www/html;
        }

        # (uncomment when first acquiring SSL cert)
        location / {
            allow all;
        }

        # (...)
    }

    # (comment out when first acquiring SSL cert)
    # server {
    # (...)
    # }
}
```

Start the service and check the logs with `docker-compose logs certbot`. If no issues crop up, proceed.

Now that you *do* have a certificate, revert your changes to `nginx.conf` and restart the service. It should be possible to access your server through a normal web browser. Confirm that your connection is encrypted.

To renew your certificate automatically, add an entry in your crontab (with `crontab -e`):

```
0 6 * * * PROJECT_PATH=/home/user/dumpster-finder /home/user/dumpster-finder/renew_certs.sh >>
```

(the `PROJECT_PATH` is required to let the script navigate into the correct folder)

8.3 Running the app

To run the app with connection to an instance of our backend, you need to specify the address to the server. **Make sure the server is running.**

Create a `.env` file with variables like those set in `.env.template` – it might look like this:

```
API_URL=http://xxx.yy.zz:3000/api/
PIC_URL=http://xxx.yy.zz:3001/pic/
```

Or like this, if you have a domain name and are running a proper instance with HTTPS:

```
API_URL=https://your.domain/api/
PIC_URL=https://your.domain/pic/
```

Then start the app in development mode:

```
npm start
```

Now you should be able to connect an emulator or a device to the Expo server.

8.4 Publishing the app

Create an Expo account and run `expo build:android` or `expo build:ios` to build and publish the app. On subsequent deployments, you should just need to do `expo publish` to update the JS bundle, since the native code should stay more or less the same.

9 Documentation of source code

9.1 Source code documentation

We used Typedoc to generate documentation for our TypeScript files, based on types and TSDoc comments. During development, you would for the most part use your IDE's way of viewing documentation. Documentation can be generated for each part of the project by entering its corresponding folder (`backend/api`, `backend/pics`, and `frontend`) and running `npm run docs` (assuming you've run `npm install` first). This command should generate a folder called `docs`, in which you'll find the generated documentation. Open `docs/index.html` in a web browser to view it.

The most recent documentation is also available at GitLab Pages:

- API server docs
- Photo server docs
- Frontend docs

For the actual repository with all our code, see

- NTNU's GitLab (with CI)
- GitHub (without CI, publicly available)

9.2 API documentation

We used Swagger to document our API, which entailed writing comments with OpenAPI specifications for each endpoint. The API documentation is generated each time you run the API or photo server, and is available at `http://<your domain or IP + port>/api/spec` and `http://<your domain or IP + port>/pic/spec`, respectively.

10 Continuous integration and testing

10.1 CI pipeline

We use GitLab CI and run our CI jobs inside the official Node.js and MariaDB docker containers.

Figure 11 shows a successful run of our pipeline. The amount of jobs differs depending on the branch a commit was pushed to. On pushes to master or develop, documentation is generated and the current version of the system is deployed to our test server and published through Expo. On other branches, those stages are dropped. In all cases, the dependency, test and audit stages run. Dependencies are only cached for the API server, since the app has dependencies that are too large to fit in the cache, and the photo server is not tested. The test stage runs unit tests and GitLab's check for exposed secrets (passwords, API keys, etc.). The audit stage uses Auditjs to scan our project's dependencies for known vulnerabilities.

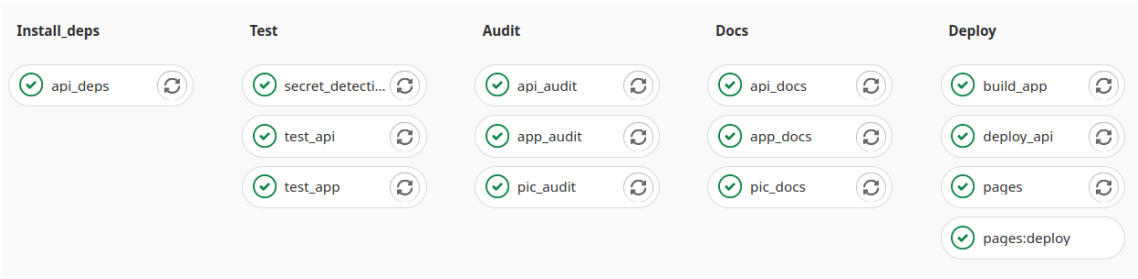

Figure 11: Screenshot of a successful pipeline run

10.2 Tests

To check for possible regressions or malfunctioning additions, we run some tests on each push. We run unit tests of the DAOs in the API server, and some functionality in the app. However, we do not run any tests of the app's UI components. Unfortunately, we did not have time to debug our problems with testing React components in the frontend. This is an issue we've faced before, but it seemed to be even more challenging with React Native and our choice of component library.

In total, we had around 95% coverage of the files we *did* test.

To run the unit tests yourself, stand in either `frontend/` or `backend/api/` and run `npm test`, after having installed dependencies and set up your environment as specified in 8.3 and 8.1.

The easiest way to test the API's functionality is to open the Swagger documentation in your web browser and go through each endpoint. We decided to avoid writing tests for the API itself, since most of the functionality stems from the DAOs and Swagger makes manual testing easy.

References

[1] Electronic Frontier Foundation. *Certbot*. URL: https://certbot.eff.org/. (retrieved 10.05.2021).

[2] Mozilla. *Mozilla SSL Configuration Generator*. URL: https://ssl-config.mozilla.org/#server=nginx&version=1.17.7&config=intermediate&openssl=1.1.1d&guideline=5.6. (retrieved 10.05.2021).

[3] Sequelize community. *Sequelize's function for escaping values in SQL queries*. URL: https://github.com/sequelize/sequelize/blob/main/lib/sql-string.js. (retrieved 08.05.2021).