DDR

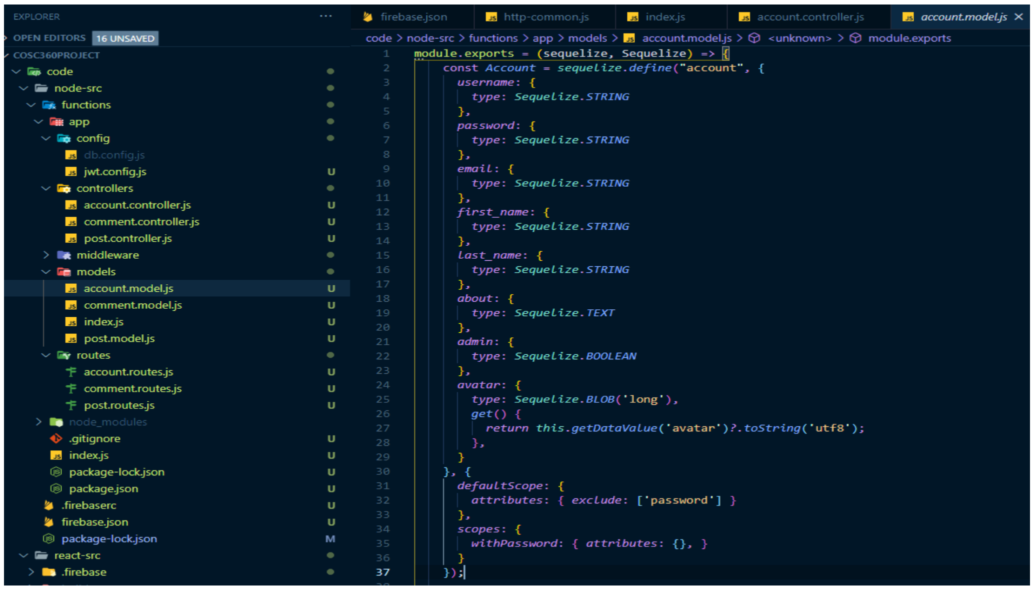
Server-side Document

COSC 360 Project

By: Damyn Filipuzzi
Daulton Baird
Ross Morrison

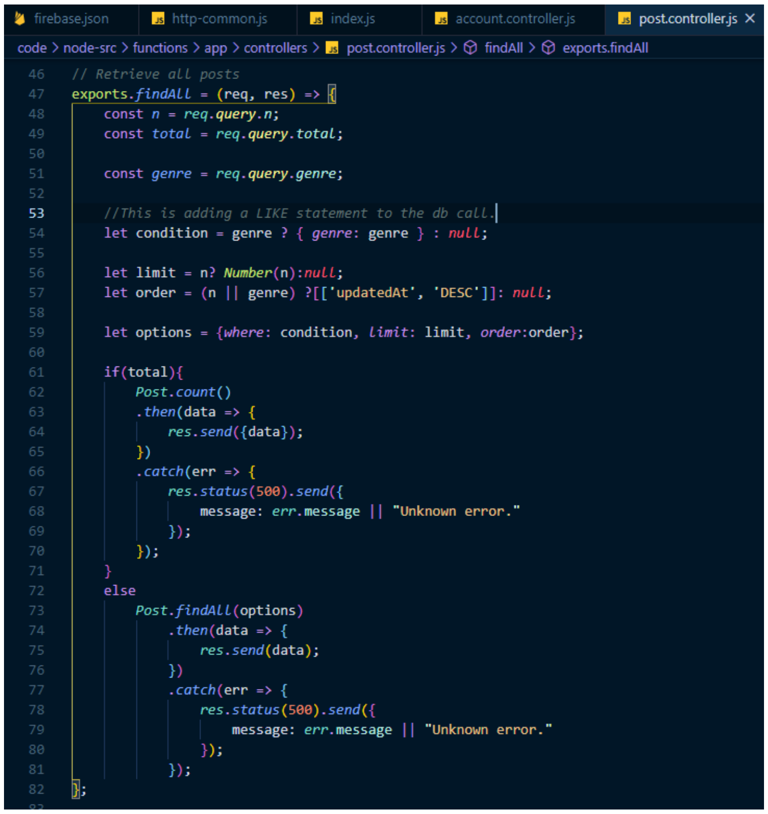
Getting Started

For the database backend we chose to use Sequelize (part of Node.Js) and express to serve HTTP endpoints for retrieving JSON data. Sequelize works by defining the database tables in code instead of writing SQL statements. Sequelize also handles data sanitation in queries.



```javascript
module.exports = (sequelize, Sequelize) => {
  const Account = sequelize.define("account", {
    username: {
      type: Sequelize.STRING
    },
    password: {
      type: Sequelize.STRING
    },
    email: {
      type: Sequelize.STRING
    },
    first_name: {
      type: Sequelize.STRING
    },
    last_name: {
      type: Sequelize.STRING
    },
    about: {
      type: Sequelize.TEXT
    },
    admin: {
      type: Sequelize.BOOLEAN
    },
    avatar: {
      type: Sequelize.BLOB('long'),
      get() {
        return this.getDataValue('avatar')?.toString('utf8');
      },
    }
  }, {
    defaultScope: {
      attributes: { exclude: ['password'] }
    },
    scopes: {
      withPassword: { attributes: {}, }
    }
  });
```

Figure 1: Example model file, containing the types and names of columns in the database.

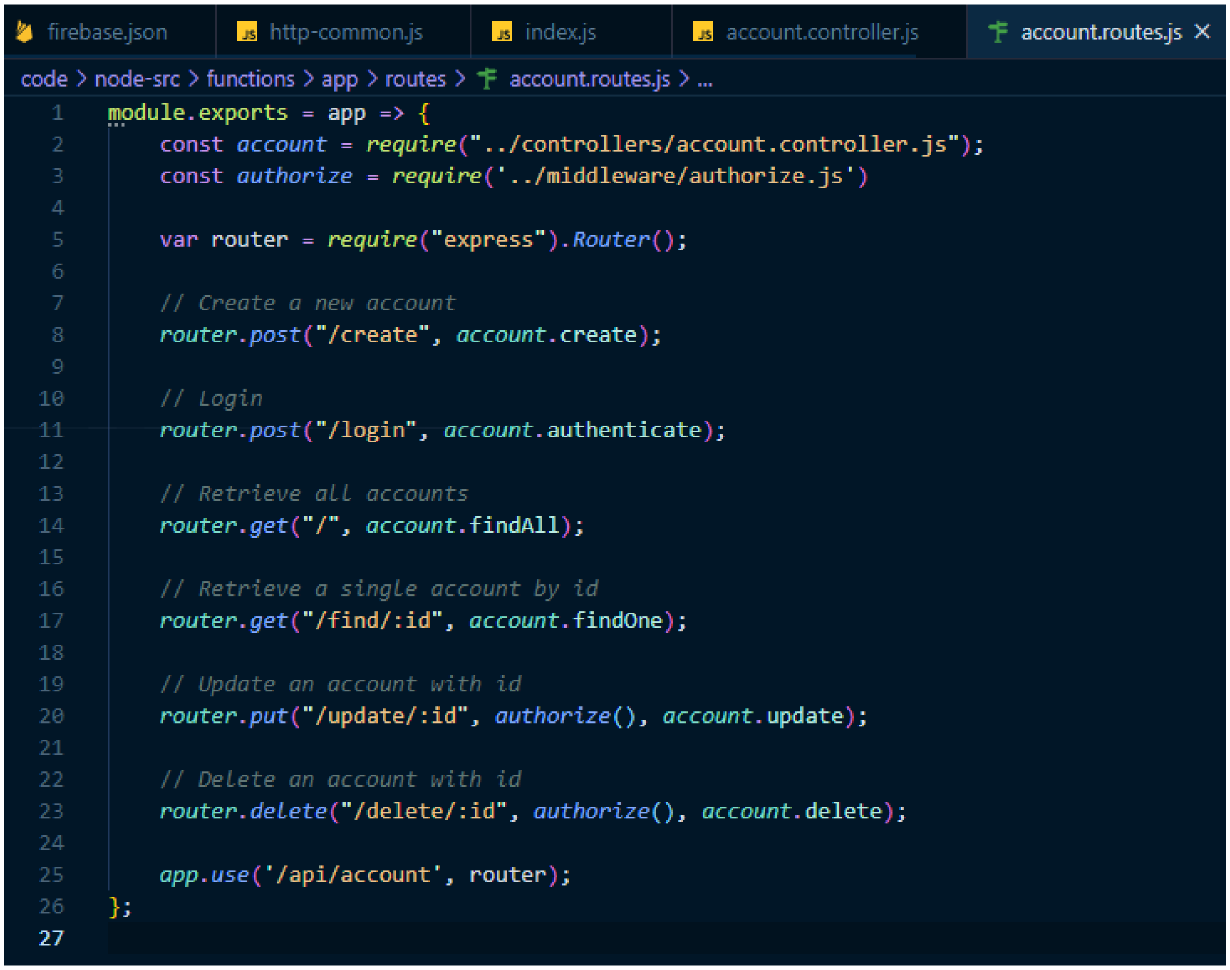

firebase.json
http-common.js
index.js
account.controller.js
post.controller.js
code > node-src > functions > app > controllers > post.controller.js > findAll > exports.findAll

```javascript
// Retrieve all posts
exports.findAll = (req, res) => {
    const n = req.query.n;
    const total = req.query.total;

    const genre = req.query.genre;

    //This is adding a LIKE statement to the db call.
    let condition = genre ? { genre: genre } : null;

    let limit = n? Number(n):null;
    let order = (n || genre) ?[['updatedAt', 'DESC']]: null;

    let options = {where: condition, limit: limit, order:order};

    if(total){
        Post.count()
        .then(data => {
            res.send({data});
        })
        .catch(err => {
            res.status(500).send({
                message: err.message || "Unknown error."
            });
        });
    }
    else
        Post.findAll(options)
            .then(data => {
                res.send(data);
            })
            .catch(err => {
                res.status(500).send({
                    message: err.message || "Unknown error."
                });
            });
};
```

Figure 2: Example controller.

Controllers

The database controllers handle all of the “query” statements that Sequelize does. These take in an HTTP request as parameters and return the data in JSON format. Error handling and also some data validation are handled in these. When logging in or creating an account, bCrypt is used to hash and salt the password before inserting into the database.



```
module.exports = app => {
    const account = require("../controllers/account.controller.js");
    const authorize = require('../middleware/authorize.js')

    var router = require("express").Router();

    // Create a new account
    router.post("/create", account.create);

    // Login
    router.post("/login", account.authenticate);

    // Retrieve all accounts
    router.get("/", account.findAll);

    // Retrieve a single account by id
    router.get("/find/:id", account.findOne);

    // Update an account with id
    router.put("/update/:id", authorize(), account.update);

    // Delete an account with id
    router.delete("/delete/:id", authorize(), account.delete);

    app.use('/api/account', router);
};
```

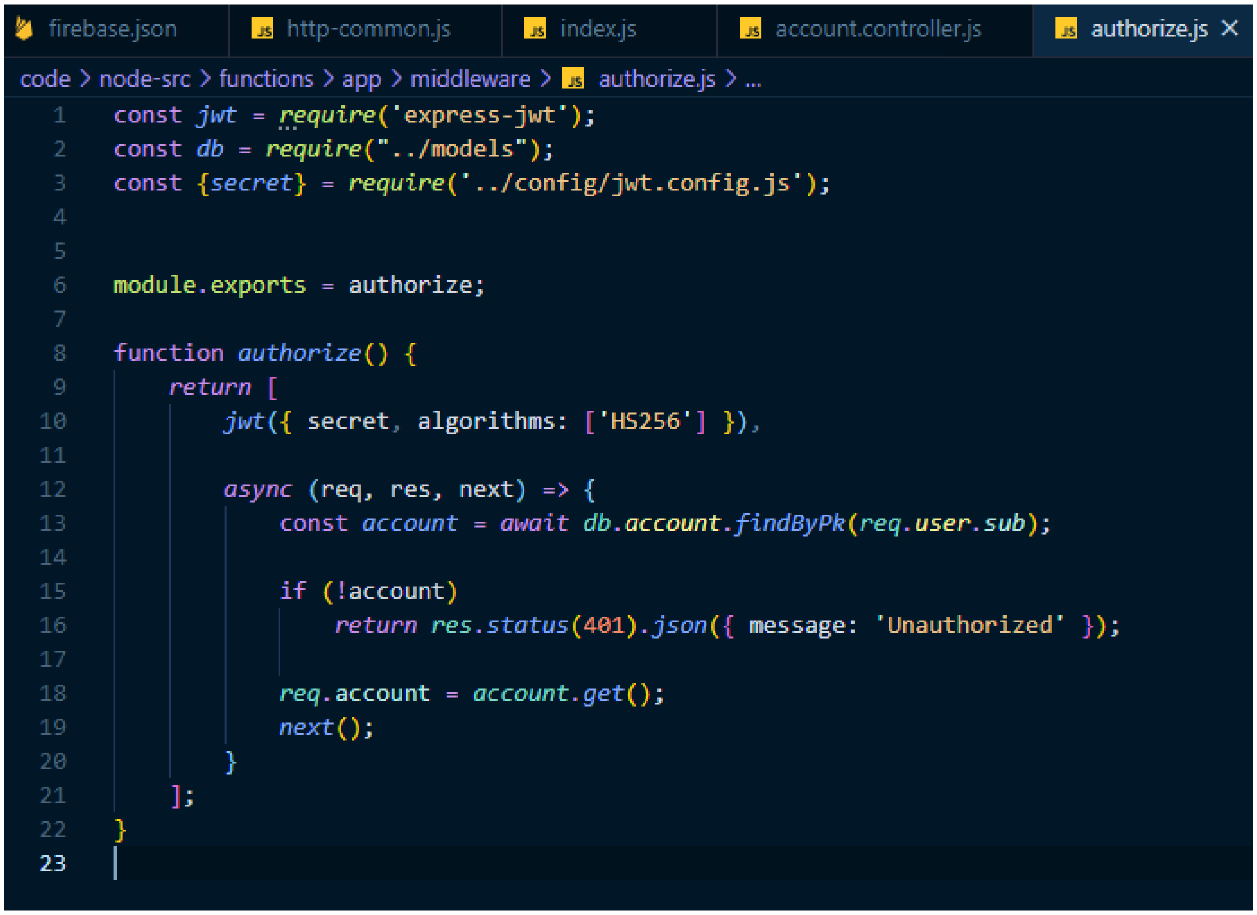
Figure 3: Example route

Routing

Since we are using Express to handle requests, each controller has an associated route file to go along with it. The route file turns HTTP api requests (eg. /api/posts) into controller functions. Authentication is also handled here with the “authorize()” method.

The type of route can determine which actions to handle like GET vs POST but also supports PUT and DELETE for updating or deleting records.

The :id parameter takes all input after the given path and converts it into a variable accessible in the controller. This simplifies the HTTP requests.



```
const jwt = require('express-jwt');
const db = require("../models");
const {secret} = require('../config/jwt.config.js');


module.exports = authorize;

function authorize() {
    return [
        jwt({ secret, algorithms: ['HS256'] }),

        async (req, res, next) => {
            const account = await db.account.findByPk(req.user.sub);

            if (!account)
                return res.status(401).json({ message: 'Unauthorized' });

            req.account = account.get();
            next();
        }
    ];
}

```

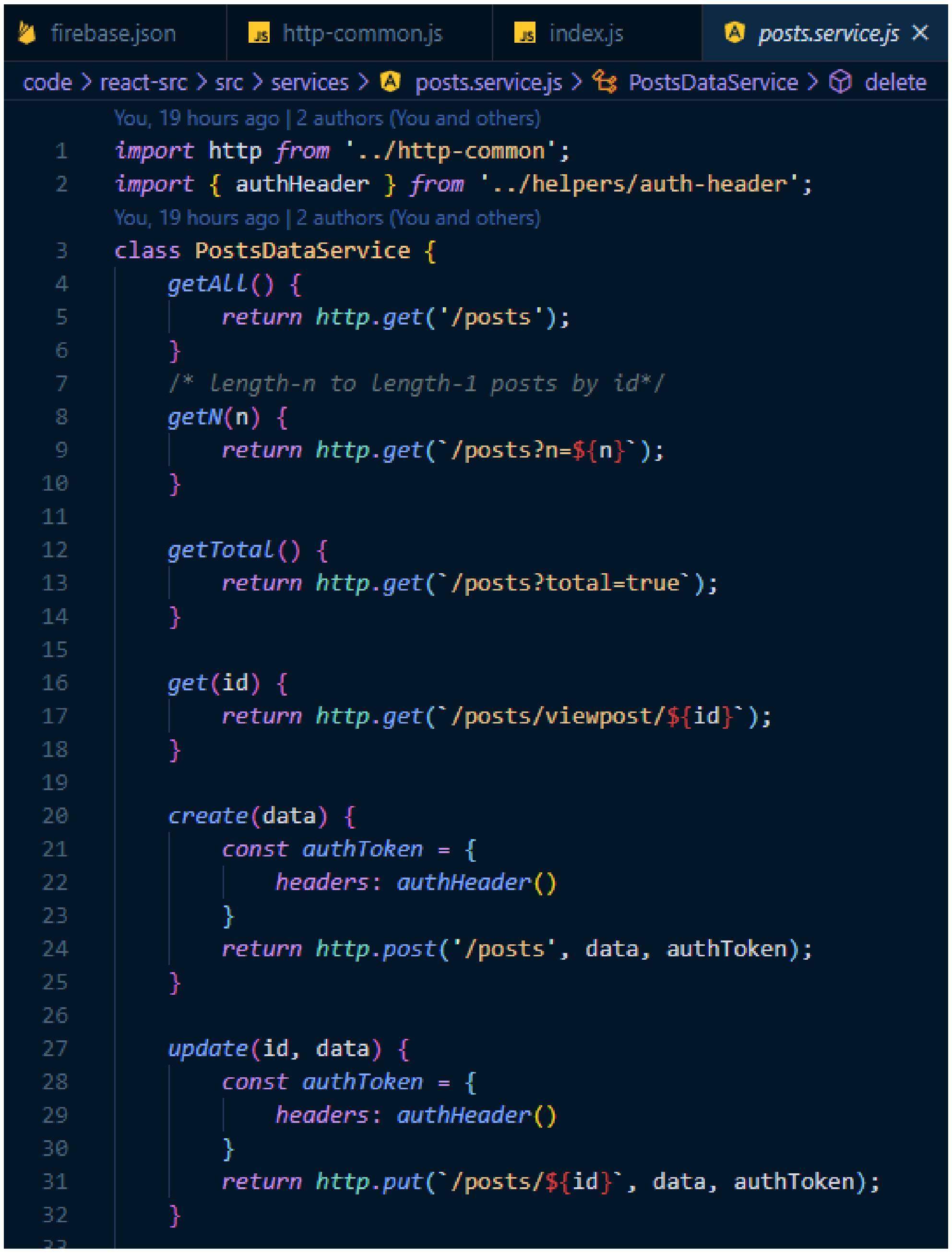
Figure 4: JWT Tokens

JWT Tokens

For keeping track of logged in users, we went with JWT tokens. These are generated server side and then sent to each logged in user for authenticating when calling the HTTP endpoints. The tokens are added to certain HTTP requests to prevent non logged in users from even viewing certain pages.

File Storage

Avatars, post uploads, and comment uploads are stored in Base64 strings in the database. This is not a very effective way of saving the data but is easy to set up and use and does not require any external file servers or CDN hosting. Because of the easy expandability of Sequelize, the media can be up to 4GB in size before it starts having errors. However, that much data for each request would overload the server and most likely cause it to crash.


firebase.json
http-common.js
index.js
posts.service.js
code > react-src > src > services > posts.service.js > PostsDataService > delete
You, 19 hours ago | 2 authors (You and others)
You, 19 hours ago | 2 authors (You and others)

```
import http from '../http-common';
import { authHeader } from '../helpers/auth-header';
class PostsDataService {
    getAll() {
        return http.get('/posts');
    }
    /* length-n to length-1 posts by id*/
    getN(n) {
        return http.get(`/posts?n=${n}`);
    }

    getTotal() {
        return http.get(`/posts?total=true`);
    }

    get(id) {
        return http.get(`/posts/viewpost/${id}`);
    }

    create(data) {
        const authToken = {
            headers: authHeader()
        }
        return http.post('/posts', data, authToken);
    }

    update(id, data) {
        const authToken = {
            headers: authHeader()
        }
        return http.put(`/posts/${id}`, data, authToken);
    }
```

Figure 5: Client-side HTTP request handlers.

Client side HTTP requests

To fetch the data from the server, Data Services are used to build HTTP requests. The HTTP requests must have the required parameters or authentication added at this step before they are sent using Axios to the API url. These data services then can be implemented in the client side code for access to the database from almost anywhere.