LUT School of Engineering Science

CT30A3203 Web Applications

Antti Knutas

Web Applications Course Project
MICROBLOG

15.12.2019

Juho Kontiainen 0503209

Table of contents

1 INTRODUCTION

This is a documentation for a web applications course project assignment. The application is a MERN stack application (MERN standing for MongoDB, Express.js, React and Node.js) and it's called *MicroBlog*. On this chapter a description for the assignment will be represented along with the chosen features and technologies. Program code available at https://github.com/JuhoKon/microblog.

1.1 Assignment description and chosen features

The task was to create a microblogging service, like Twitter or Mastodon, using full stack Node.js JavaScript web technologies. Any Node.js web framework was allowed for back-end and any front-end framework was allowed for front-end.

Mandatory requirements were set for the assignment worth of 30 points as well as optional requirements and features that students could choose all the way to a maximum of 60 points. Minimum implementation is described below along with a column representing if the functionality is implemented in the application.

Table 1 Mandatory functional requirements.

Functional requirements	Implemented
The system must support posting new posts	x
The system must support multiple users	x
The system must support viewing other users' posts	x
The system must use HTML and at least basic CSS	x

Table 2 Mandatory non-functional requirements.

Non-functional requirements	Implemented
The system must be implemented using Node.js backend	x
The system must use npm package manager. If you use an alternative, such as yarn, document its usage.	x
The system must be runnable by a npm or equivalent start command.	x

The system must store data. The data storage can be in-memory at this stage.	x
The code must be stored in a Git or Mercurial version control system	x

All the mandatory requirements set for the assignment are matched in the application. Moving on to the optional requirements and grading. As said earlier, implementing more features and using different technologies will increase the point total. All the chosen optional requirements and features are represented below along with a column representing acquirable points for each requirement. I will only list features which are implemented in the application. The chosen technologies are written in bold.

Table 3 Implemented optional requirements with the points.

Feature	**Points**
Using React.js front-end Using redux and an advanced React architecture	15 pts +5 pts
Running your application in a Docker container (or having the application Docker compatible)	5 pts
Using a database, such as **Mongo**, Redis, or any SQL-compatible Use an ORM and models in backend, such as the **Mongoose** (MongoDB) or Sequalize (SQL)	5 pts +3 pts
Supporting pictures storage and display	5 pts
Using a cache (backend)	3 pts
Using XHR (also known as AJAX/AJAJ) for data transfer and frontend synchronization	4 pts
User registration, authentication, and password storage	5 pts
Provide data from your application through an API and document it with a suitable tool, e.g. with **apiDoc** or API Blueprint.	3 pts
Total points from optional requirements	53 pts
Overall points	30+53 = 83pts

1.2 Estimate of point total

With the 30 points acquired from mandatory requirements, a total of **83 points** are accumulated. But with the 60 points being maximum for the assignment, **the assignment is worth of 60 points**. My reasoning behind creating more features and using more technologies is that I simply wanted to learn more and more importantly create a project that could help me present my skills as a software developer for future.

2 DESCRIPTION OF THE SOFTWARE

On this chapter a description for the software will be provided along with a figure of the program architecture including program components and explaining where they are located and how they communicate.

2.1 Software description

The application consists of Express.js back-end and React.js front-end as seen on the figure 1 (Figure 1). The software uses a MongoDB database to store data. Also, a simple server-side cache is built using Redis. I will go through the libraries chosen and design choices separately for back-end and front-end.

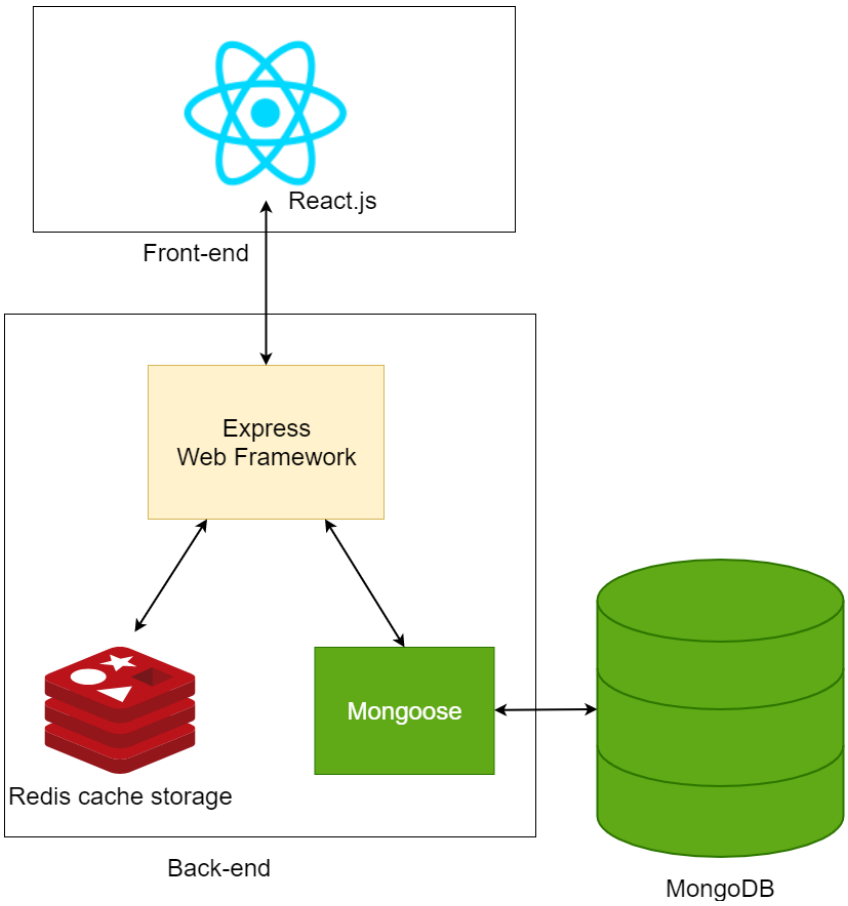

Figure 1 Program architecture.

The main feature of the program is creating posts (with or without uploading images) and viewing them. There are set requirements for the posts (text max 256 characters and title 50

character, images max 10MB and only jpeg/png formats). In order to create posts, you have to register/login to the service. You can view other people's posts and visit other people's profile pages which now only shows posts by that user, and to visit other profiles, you must be logged in. You can also edit/delete your own posts on your own profile page.

2.2 Back-end

The back-end (server) is built using Express.js. The server is responsible for database interactions and providing data to the front-end through a RESTful API. Choices for libraries will be represented below.

2.2.1 Back-end libraries

jsonwebtoken

JSON Web Tokens are used for user authentication in the application. This package will help in setting up the protected routes and protected content in general. Tokens are set to expire in one hour.

bcrypt.js

This package is used in hashing and checking the passwords.

nodemon

Simple tool for helping to develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.

Mongoose

Communication with the database is implemented through Mongoose which will serve as an ORM (Object Relational Mapping). ORM is used to convert information from the database to a JavaScript application without SQL statements. Database models will be created using Mongoose as well.

dotenv

For loading environment variables from a .env file into process.env.

Multer

Multer is a middleware for handling multipart/form-data, which is primarily used for uploading files. On our application Multer is used for uploading pictures to the server. The path to the file is sent to the database along with the information about the created post.

Redis

Redis client for node.js. Used for communicating with the Redis cache.

fs

fs-module is used to access and interact with the file system. Used for deleting the uploaded files from the server.

express-validator

Used for validating blog and user data in the application.

2.2.2 Back-end program components

The program components used for creating the server is described below (Figure 2). Routers

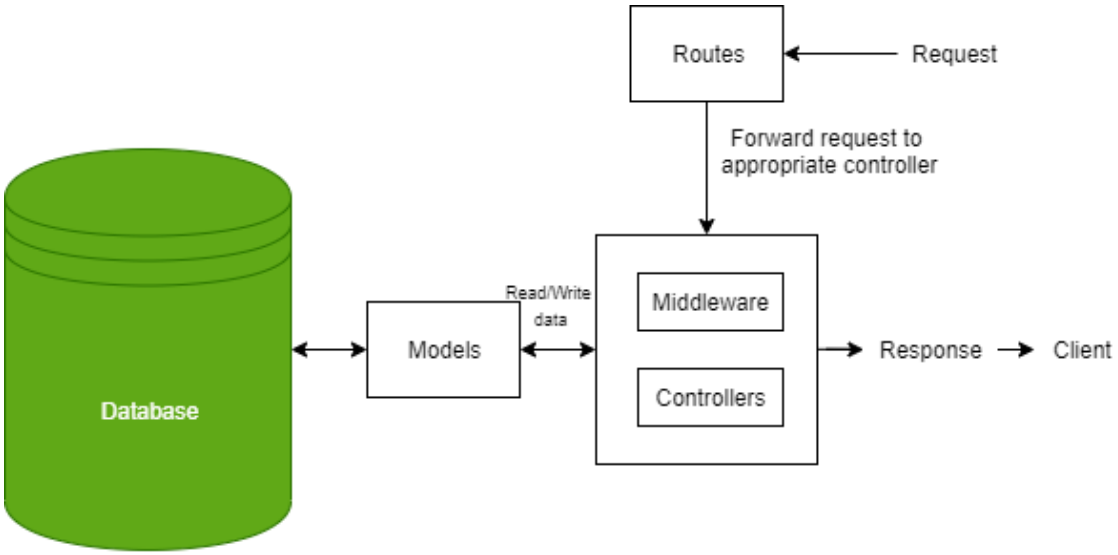

Figure 2 Backend architecture.

forward the requests to the appropriate middleware and appropriate controller functions. Middleware used in this application consists of auth-, multer- and two validator-middleware (as seen on Figure 3). Controller functions get the requested data from the models and returns the client a JSON response containing either requested data or message if the request was successfully completed or not. More information about the available API calls is found in the API documentation (error messages, request examples and more).

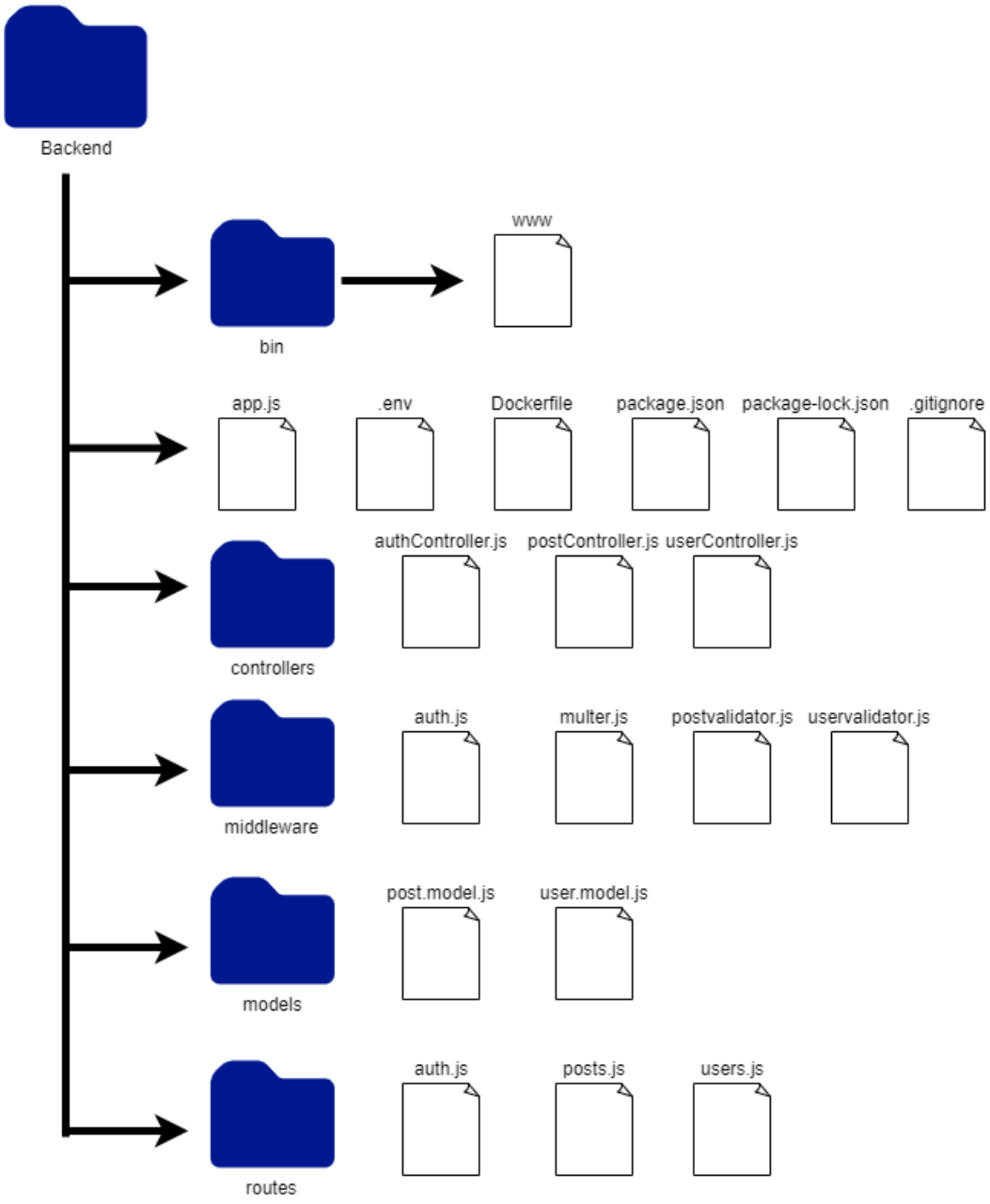

Figure 3 Backend file structure

2.3 Front-end

The front-end (client) is built using React.js. The solution uses redux and an advanced React architecture with actions and reducers. The data flow in the application is seen below (Figure 4). “Connected” React components listen on the parts of the store. The components create actions through the imported actions. We are using *Thunk* middleware for Redux which makes it possible to communicate asynchronously with an external API and makes it easy to dispatch actions that follow the lifecycle of a request. The actions come to the reducer and then the reducer changes the state and return the new state to the store. The connected React components now receive the new state from store as props and re-render.

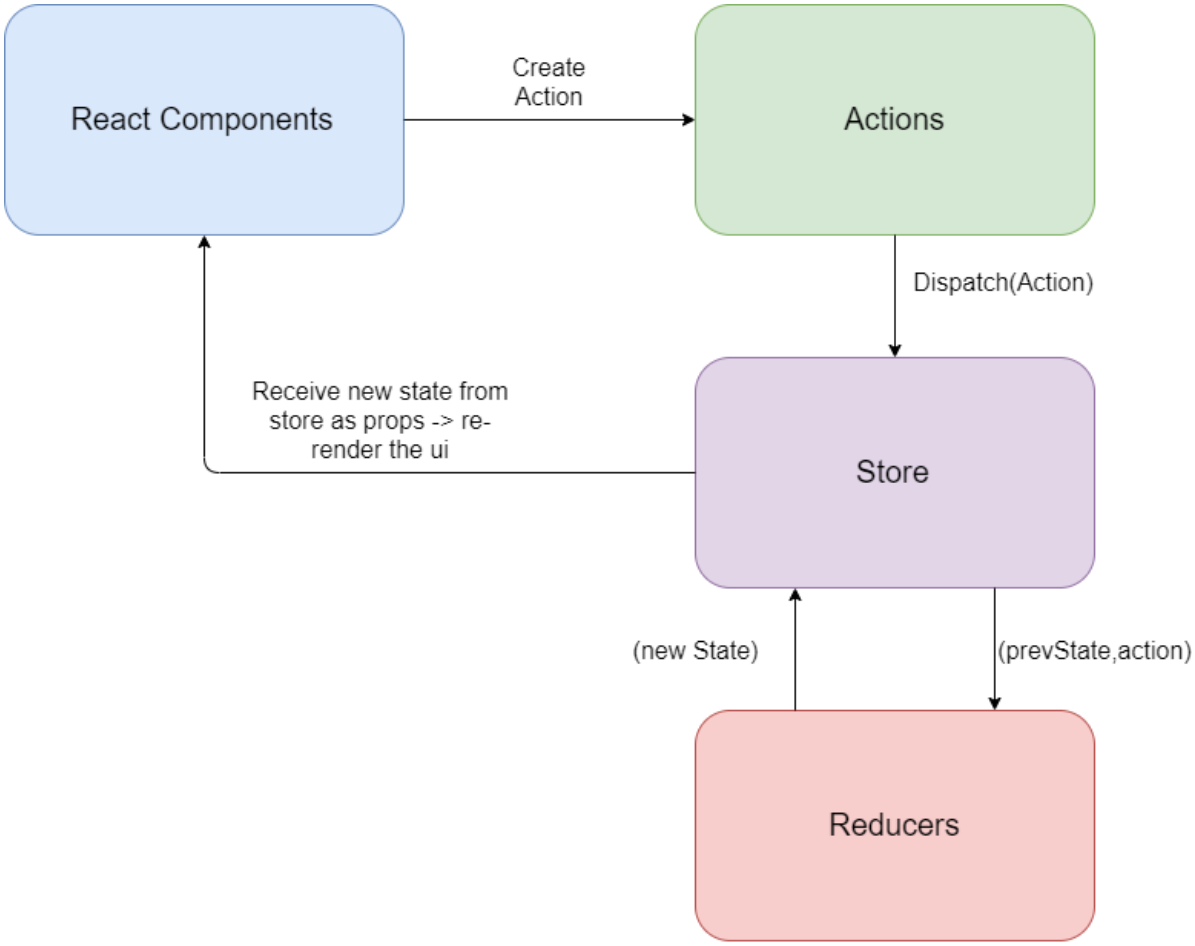

Figure 4 Redux data flow.

2.3.1 Front-end program components

The architecture of front-end is seen on the figure below (Figure 5) and the file structure on the figure 6 (Figure 6). The routes on the architecture figure stand for the routes found in App.js. Inside the components there is an *Auth* folder which contains components and modals used in login, registration and logout activities. As the components create actions (for example fetch user data from the server) and dispatch async actions, with the help of *thunk*¸ we wait for the async response and then dispatch the new action with the data retrieved to reducer.

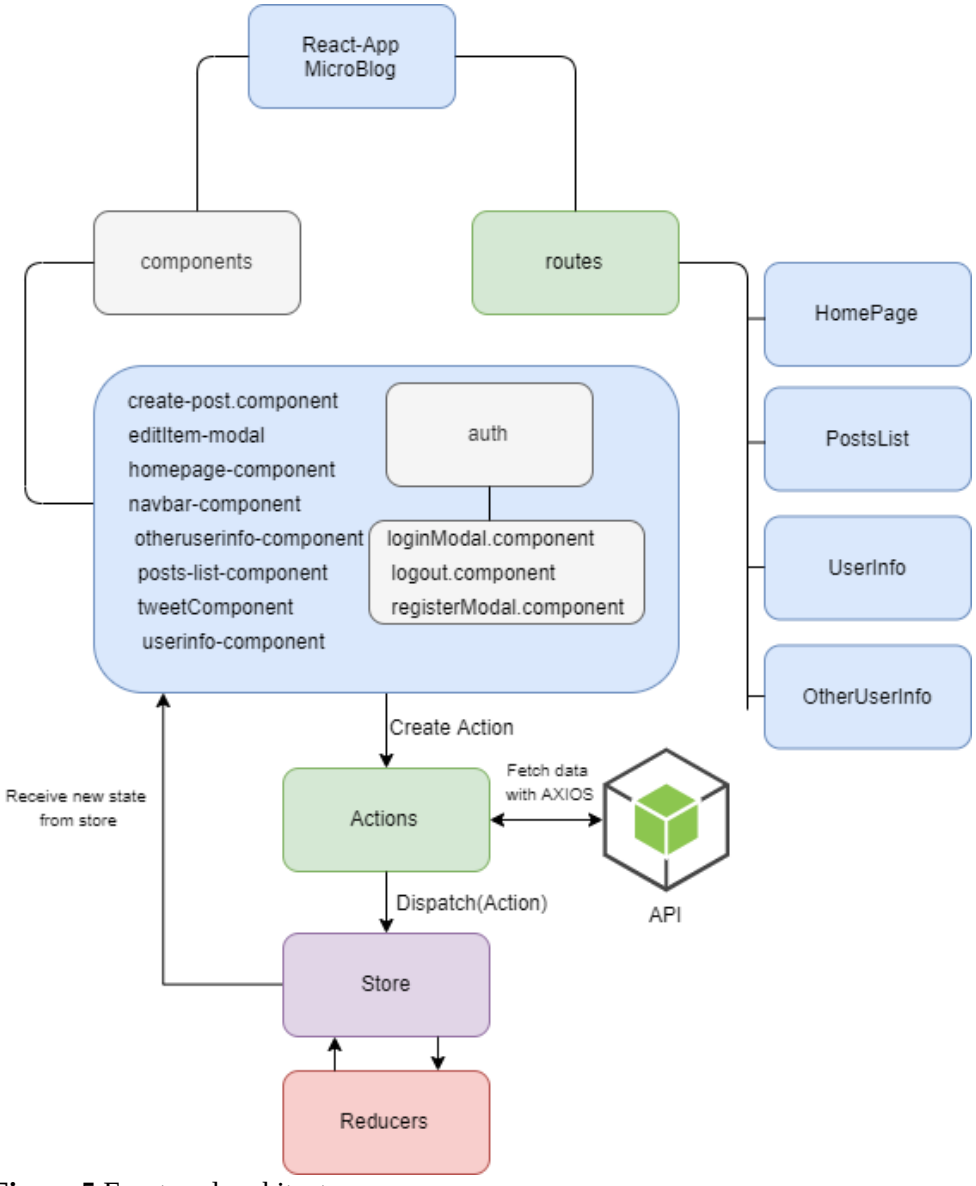

Figure 5 Front-end architecture

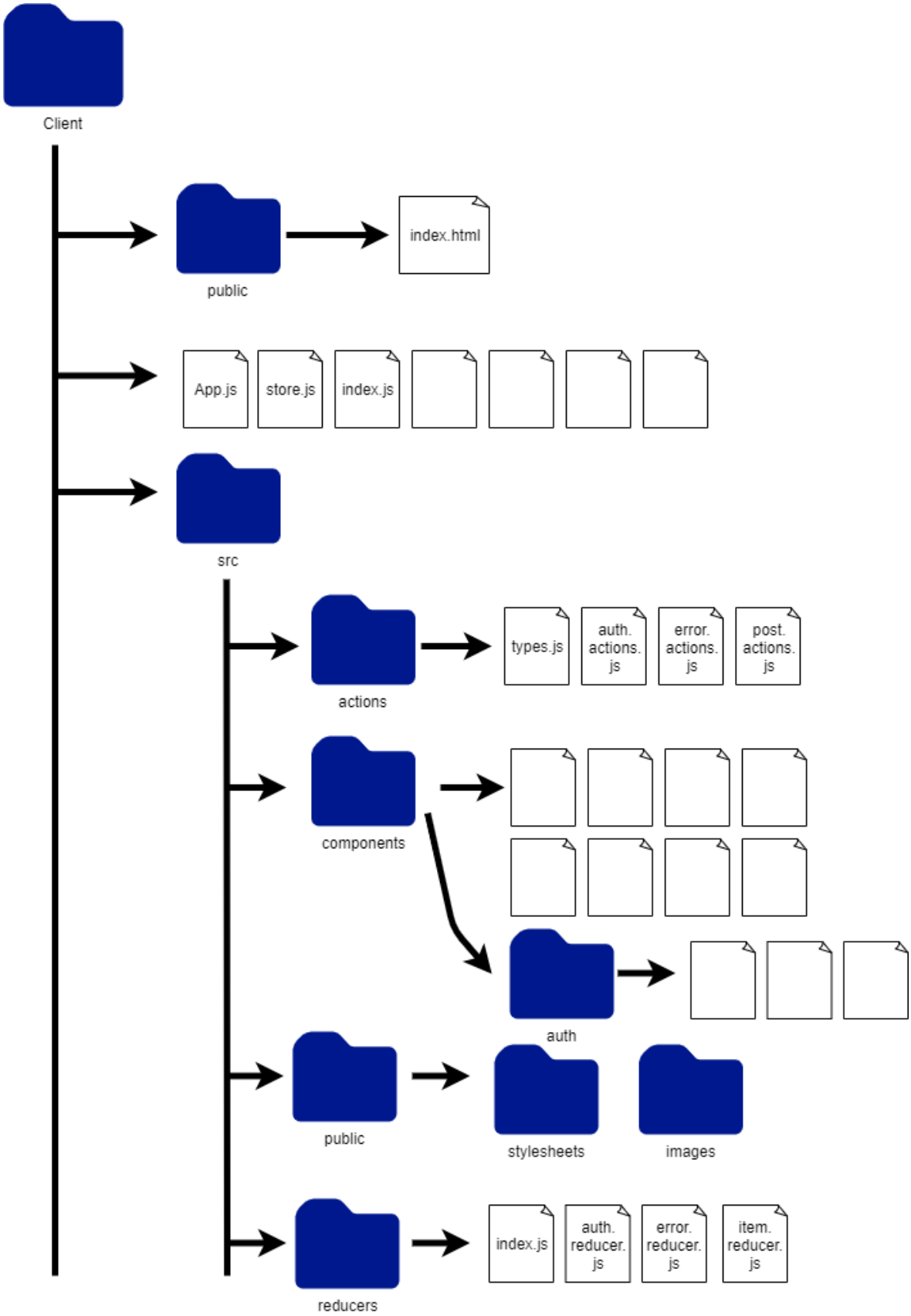

Figure 6 File structure for front-end (with the most important files)

2.3.2 Word about the components and logic how they work together

The components (seen on Figure 5 and on Figure 6) are building blocks for the app. The routes defined earlier call these components and they return a React element that describes how a section of the UI should appear. For example the *posts-list-component* gathers all the posts and then renders them by calling another component, *tweetComponent.*

Input validation is mostly done by the server, but some validation is also done in the front-end such as making sure that the fields aren't empty. Error messages sent by the API in JSON-format are rendered on the app informing user about the error. Possible error messages that are rendered on the UI:

1. Registration
    a. Username is already in use
    b. Username has less than 3 characters
    c. Username has whitespaces in it
    d. Email is already in use
    e. Email isn't in correct email format
    f. Password is less than 8 characters long
2. Login
    a. All fields aren't entered
    b. User doesn't exist
    c. Invalid credentials
3. Create post/Edit post
    a. Title has more than 50 characters
    b. Text has more than 256 characters
    c. Invalid file type
    d. File is too large

As seen on the figure above (Figure 6) we have multiple reducers and actions. The types.js in the actions folder stands for a module which just contains all the type properties. The index.js in the reducers folder stands for a *combineReducers* helper function which turns an object whose values are different reducing functions into a single reducing function which

we can pass to the *createStore* function in the store.js file. This way each reducer manages their independent parts of the state. For example *auth.reducer* handles the auth state and *error.reducer* handles the error state. These states can be furthermore acquired by the components with the connect – function and mapStateToProps function inside the components. Dynamic content on the site is implemented this way and authentication as well. As the user logins or creates a new account, a newly made JSON Web Token is returned from the API to the client. As the authentication is checked by *loadUser*-action it is being read from the Redux store's auth state and then returned to the API as a header to check if the token is valid or not. Now it is possible to read auth state in the components and simply check if the user is authenticated or not. This is also how some content is made available only to logged in users (in front-end).

2.3.3 Used libraries in the front-end

react

Javascript library created by Facebook and is used for building UI components.

redux

Redux is a predictable state container for Javascript apps

redux-thunk

Middleware that allows to call action creators that return a function instead of an action object. Makes it possible to communicate asynchronously with an external API – making it easy to dispatch actions that follow the lifecycle of the request. For example if the request is successful, we can dispatch a new action with the received data or vice versa if the request fails.

moment

Library for working with time formats. Used for converting date formats into more readable state for rendering.

react-fast-compare

Used for comparing props to each other in the *tweetComponent*-component.

prop-types

Used for type checking. Makes it easier to catch bugs. Currently only props from the store's state are here.

reactstrap

React Bootstrap 4 components. Used for creating a responsive UI with UI components built by professionals.

bootstrap

Needed for reactstrap.

axios

Promise based HTTP client for the browser and node.js. Used for communicating with the back-end's REST API. All the *axios* requests are found in the actions.

2.3.4 Design choices

I chose reacstrap for styling and for the React Bootstrap 4 components it offers. Images seen on the homepage are from undraw.co which has a constantly updated collection of svg images that are free to use and without attribution. Custom css was made for coloring and positioning navbars, buttons and the other elements.

3 ENVIRONMENT SETUP

On this section a tutorial on how to setup the environment and how to get the application up and running.

3.1 Non-docker setup

Note: If not using docker you **must** have mongoDB and redis running on your computer.

1. Git clone the repository to a folder you'd like:
   - *git clone https://github.com/JuhoKon/microblog.git*
2. Next navigate to the backend folder in the terminal and install dependencies using npm install – command. After that do the same on the client folder.
   - *cd to backend and client*
   - *npm install on both*
3. Run mongoDB server on port 27017 (default port)
4. Run redis on port 6379 (default port)
5. To get backend running type npm run devstart on the backend folder
   - *cd to backend*
   - *npm run devstart*
   - Ensure that mongoDB and redis are running. Otherwise server will not run correctly.
6. To get frontend running type npm start on the client folder
   - *cd to client*
   - *npm start*

Now you should have the front-end application running on localhost:3000 and back-end on localhost:8080.

3.2 Docker setup

1. Git clone the repository to a folder you'd like:
   - *git clone https://github.com/JuhoKon/microblog.git*
2. Navigate to backend folder and edit .env file

- *cd backend*
- *ls -a*
    - You should see the .env file
- Pick your favorite text editor and edit the .env file
- Change address on REDIS_HOST = “127.0.0.1” to “redis”
- Change address on MONGO_URL = “mongodb://127.0.0.1:27017/dbtest” to “mongodb://mongo:27017/dbtest”

3. Navigate to client folder and edit package.json file
    - Change the proxy address from http://localhost:8080 to http://server:8080
    - Found after dependencies as “proxy”
4. Build the service
    - On the microblog folder execute “docker-compose build”

5. Start the service
    - On the microblog folder execute “docker-compose up”

docker commands:

docker-compose build

docker-compose up

Or with sudo, if doesn’t work.