



POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

IT Department

Department of Software Engineering

Software and database engineering

Antoni Malinowski

Album number s20824

How to efficiently connect people online? Implementing a social media platform that fosters software maintainability and collaboration between developers.

Diploma

Written under the direction

mgr inż. Maciejewska Katarzyna

Warszawa 2024



POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Wydział Informatyki

Katedra Inżynierii Oprogramowania
Inżynieria Oprogramowania i Baz Danych

Antoni Malinowski
Numer albumu s20824

Jak skutecznie łączyć ludzi w Internecie? Implementacja platformy społecznościowej, która ułatwia utrzymywanie oprogramowania i współpracę pomiędzy developerami.

Praca inżynierska napisana pod
kierunkiem:
mgr inż. Maciejewska Katarzyna

Warszawa 2024

Abstract

Since the early 2000s, many social media platforms have been launched. The purpose of these applications is to provide users with a space to share opinions and meet new people with similar interests. Platforms like Instagram, Facebook, or X (previously Twitter) all use very similar approaches to sharing content. The idea focuses on creating connection opportunities for users who upload multimedia content (text, images, videos), comment on each others' posts and follow like minded accounts.

In recent years, new platforms serving similar purposes have arised. Mastodon or Odysee serve the same purpose of connecting its users, but they have fundamentally different backbone architecture of their service. They are fully or partially open-sourced and their focus is greatly put on users' privacy, decentralization and the P2P nature of the platform.

The purpose of this paper is to present a modern approach to running and using social media platforms. This approach comes directly from the evolution that the internet has been going through for the past years and comes down to openness, transparency, accessibility and privacy.

This paper goes through some of the problems that social media platforms face today, how developers approach to solve them and what are the different challenges that may arise when using such platforms.

Moreover, the author proposes an implementation of such a social platform, describing its core functionality, challenges and plans for further development.

Keywords: software, connecting people, React, Docker, social media, collaboration, maintainability, open-source

Abstrakt

Od początku lat 2000. wiele platform mediów społecznościowych zostało uruchomionych. Celem tych aplikacji jest zapewnienie użytkownikom przestrzeni do dzielenia się opiniami i poznawania nowych osób o podobnych zainteresowaniach. Platformy takie jak Instagram, Facebook czy X (dawniej Twitter) stosują podobne podejście do dzielenia się treściami. Pomysł ten skupia się na daniu możliwości nawiązywania połączeń przez użytkowników, którzy dzielą treściami multimedialnymi (tekst, obraz, wideo), komentują posty innych oraz obserwują podobne konta.

W ostatnich latach pojawiły się nowe platformy służące podobnym celom. Mastodon czy Odysee mają ten sam cel łączenia użytkowników, lecz różnią się fundamentalnie na poziomie architektonicznym. Posiadają w pełni lub w części otwarty kod źródłowy i kładą duży nacisk na prywatność użytkowników, decentralizację oraz charakter P2P (bezpośrednich połączeń pomiędzy użytkownikami).

Celem niniejszej pracy jest przedstawienie nowoczesnego podejścia do prowadzenia platform społecznościowych oraz korzystania z nich. Podejście to wynika bezpośrednio z ewolucji, przez którą przechodził internet w ostatnich latach i sprowadza się do otwartości, przejrzystości, dostępności i prywatności takiego rozwiązania.

Praca ta przeanalizuje niektóre z problemów, z jakimi borykają się dzisiaj platformy społecznościowe, jak deweloperzy podchodzą do ich rozwiązywania oraz jakie różne wyzwania mogą pojawić się podczas korzystania z takich platform.

Ponadto autor proponuje implementację takiej platformy społecznościowej, opisując jej podstawową funkcjonalność, wyzwania i plany dalszego rozwoju.

Słowa kluczowe: oprogramowanie, łączenie ludzi, React, Docker, media społecznościowe, współpraca, utrzymywanie, otwarte źródło

Introduction.....	6
The structure of the diploma.....	7
Dictionary of technical terms.....	8
1. Overview of the problem.....	10
2. The case for modern interaction.....	13
3. Technologies used for the implementation of the project.....	13
3.1. JavaScript.....	13
3.2. React.....	14
3.3. Node.js.....	14
3.4. Express.....	14
3.5. Multer.....	14
3.6. Postgres.....	15
3.7. Docker.....	15
4. Platform architecture.....	15
4.1. Backstory of JavaScript and Node runtime.....	15
4.2. ShaReCon architecture.....	16
4.2.1. Platform architecture.....	16
4.2.1.1. Functionalities Use Cases diagram.....	19
4.2.1.2. Sequence diagrams.....	22
5. Platform implementation.....	25
5.1. Open-source code.....	25
5.2. React frontend.....	25
5.2.1. Account creation.....	26
5.2.2. Signing in.....	33
5.2.3. Posting content.....	36
5.2.4. Deleting content.....	40
5.3. Database choice.....	40
6. The future of the platform.....	44
6.1. A detailed look at the rising technologies.....	44
6.1.1. Web vs mobile.....	44
6.1.2. Decentralized vs centralized.....	45
6.1.3. P2P sharding vs compression.....	46
6.1.4. UX vs UI.....	47
6.2. Observation vs participation.....	47
7. Summary.....	48
Bibliography.....	49

Introduction

Today, social media giants like Facebook, Instagram or Twitter play a leading role in the way we communicate with each other online. They amass billions of users every month. Their business and architecture models are ones to look up to as an aspiring developer as they often serve as a blueprint for how to create successful social media.

On the other hand, there is a social platform called StackExchange (with its most famous child - StackOverflow as presented on Figure 1) which gathers millions of passionate people and experts, keeping the main focus of sharing useful knowledge and strictly keeping the threads relevant to the main discussion.

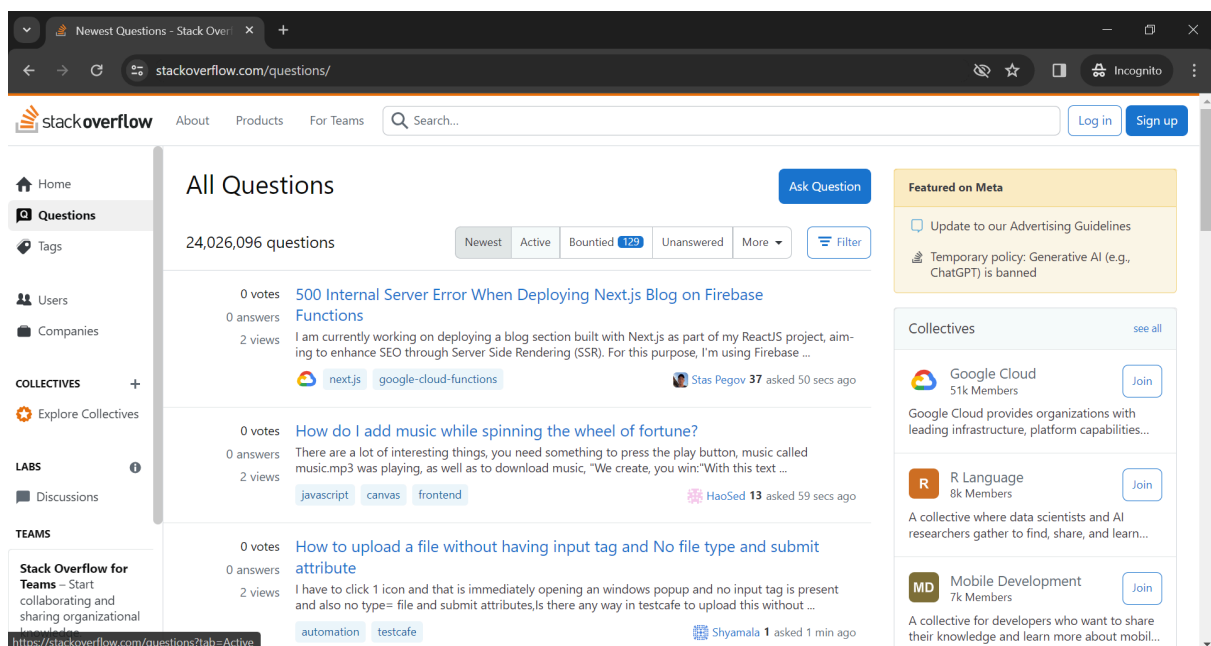


Figure 1. Stack Overflow [1]

In recent years however, with the users' growing interest in privacy, ownership and inclusiveness platforms like Mastodon with its open-source and decentralized nodes, Odyssey implemented on LBRY blockchain as presented on Figure 2 and other similar sites have been created. The people of the internet have been visibly concerned about their online activity and seem to care what's the underlying philosophy and architecture of applications they use.

For this reason, it is important to present one's own approach to building online communities. For when more ideas are present, the greater options we, as users, have in choosing the right suite. More solutions also create opportunities to step back and think about the essence of what makes social activity pleasant. Coming back to the core problem and tackling it allows the creation of better products and services.

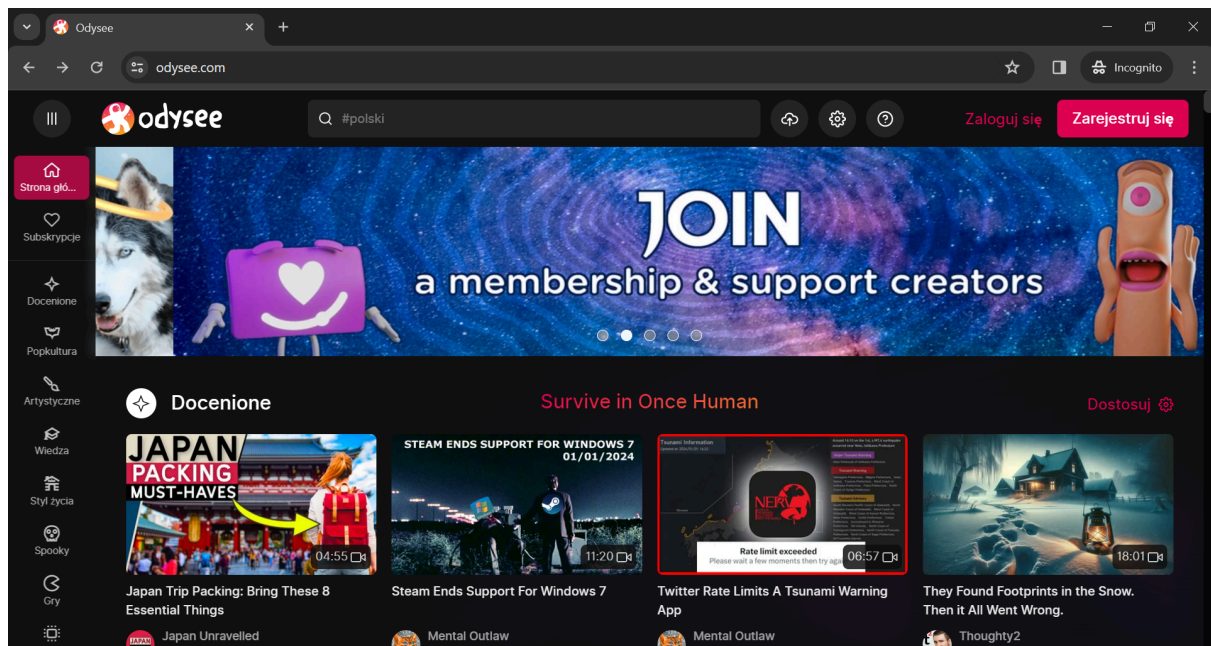


Figure 2. Odysee [2]

The structure of the diploma

The diploma thesis has been divided into four parts.

In the first part, the author identifies the problems associated with traditional social media platforms. This section also delves into the modern approach and examines how the software's accessibility plays a pivotal role in shaping the final product.

The second part builds a compelling case for modern interaction, expanding the scope from conventional social media activities to collaborative product development and active user contributions to the platform.

The third part is focused on a detailed breakdown of the current state of the proposed open-source social media platform. This chapter is primarily dedicated to the technical analysis of the author's solution, providing a clear rationale for the use of featured technologies.

In the last, fourth part, the author reflects on potential future developments for the platform and discusses the incorporation of technologies that could be integrated into the code. The author lays out the motivation behind choosing specific technologies over others, establishing a foundation for introducing a list of promising technologies that could enhance the current tech stack in the foreseeable future.

Dictionary of technical terms

Below is the dictionary describing technical terms used throughout the paper. Dictionary serves the purpose of briefly describing terms that some of the readers may not be familiar with but are willing to follow along with the paper.

Acronym	Full name	Definition
ShaReCon	Share, Reference, Connect	social media platform for sharing content between users
VS Code	Visual Studio Code	source-code editor made by Microsoft with the Electron Framework, for Windows, Linux and macOS.
JS	Javascript	high-level programming language that is one of the core technologies of the World Wide Web
AWS	Amazon Web Services	cloud computing platform by Amazon offering (alongside other products) storage buckets, compute instances and a messaging queue
-	LBRY	blockchain-based digital content distribution platform and protocol
S3, bucket	Simple Storage Service	service offered by AWS that provides object storage on the cloud through a web service interface
React	React.js or ReactJS	free and open-source front-end JavaScript library for building user interfaces based on UI components
Node	Node.js	open-source back-end JavaScript runtime environment, runs on the V8 JavaScript Engine, and executes JavaScript code outside a web browser
Express	Express.js	back end web application framework for building RESTful APIs with Node.js. It is designed for building web applications and APIs
-	Multer	“Multer is a Node.js middleware for handling multipart/form-data, which is primarily used for uploading files”[3]
API	Application Programming Interface	a way for two or more computer programs to communicate with each other
UI	User Interface	space where interactions between humans and

		computer programs occur - simply what the user sees as the final product of development
URL	Uniform Resource Locator	reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it (colloquially termed as a web address)
MVP	Minimum Viable Product	version of a product with just enough features to be usable by early customers who can then provide feedback for future product development
P2P	Peer-to-peer	distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the network
GUN	GUN.js, gunDB, GUN ecosystem	GUN is an ecosystem of tools that let you build community run and encrypted applications
CDN	Content Delivery Network	geographically distributed network of proxy servers and their data centers
WebRTC	Web Real-Time Communication	free and open-source project providing web browsers and mobile applications with real-time communication via APIs
RAD	Radisk Storage Engine	“in-memory, as well as on-disk radix tree that saves the GUN database graph for fast and performant look-ups. Radix trees have a constant lookup-time”[6]
IPFS	InterPlanetary File System	protocol, hypermedia and file sharing peer-to-peer network for storing and sharing data in a distributed file system
-	Neo4j	graph database management system; an ACID-compliant transactional database with native graph storage and processing
psql	PostgreSQL	open source object-relational database system
sql	Structured Query Language	structured query language used mainly for querying databases, schema and records creation and database administration and management
-	Docker	Development tool used to create isolated environments for applications without the need for installing libraries and tools locally.

1. Overview of the problem

The proprietary nature of platforms like Facebook, Instagram, and Twitter (prior to the overtake that happened in 2022) often places a heavy emphasis on shaping the public dialogue. This approach is not flawed by itself only when it takes into consideration users' wellbeing and free speech culture cultivated at every stage of the product development. These platforms, driven by the principles of surveillance capitalism, have been known to collect vast amounts of user data, often without explicit consent, and leverage this data to create highly personalized, intrusive advertisements and manipulate the perception of ongoing political and socio-economic events. The consequences of such data practices have raised concerns regarding user privacy, the ethical boundaries of data usage and free speech limitations.

Closed-source proprietary platforms are by definition not designed for user's privacy and fair treatment designed and controlled by an organization, where the source code is not available to the public for viewing, modification, or redistribution.. This opacity extends to algorithms that determine what content users see, how moderation decisions are made, and the criteria for banning or suspending user accounts. This lack of transparency leaves users in the dark about the rules and policies governing their online interactions, raising questions about accountability and the potential for bias in content moderation.

On the contrary to proprietary software, open-source social media or software in general embrace transparency by the very definition. Their source code is accessible to anyone, allowing users to inspect the inner workings of the platform and verify that data handling is in line with their expectations. This transparency fosters trust and accountability within the community.

Great example of such transparency is the community notes functionality of X as presented on Figure 3. Community notes serve the role of a fact checking system that aims to verify whether the information posted on X are truthful or misleading in any way. (The global rollout of Community Notes has occurred in December 11, 2022 when the platform was still known as Twitter)

X
Uwaga


Sam Parker


@SamParkerSenate · 21 sty


BREAKING: 70 American soldiers have just died fighting another war for Israel.



Why are Americans being sent to die for a country that is perfectly capable of conducting its own wars? ...

[Pokaż więcej](#)


Obecnie oznaczona jako pomocna
21 sty · [Zobacz szczegóły](#)
...


Pokaż w serwisie X · wyświetlenia: ponad 2 100


Bezpośrednio odnosi się do argumentów podanych we wpisie · Zawiera przydatny kontekst

The post is incorrect
The NY Times (link below) stated unjured

70 American soldiers are not dead
The Americans were injured

<https://twitter.com/AJABreaking/status/1748948506092392723?t=vDE-ydfCseOZHVjQaj8U2w&s=19>

<https://www.nytimes.com/live/2024/01/20/world/israel-hamas-news#a->

Figure 3. Community Note written under a post [4]

As visible on Figure 4, Community Notes code is open-source and available for transparent examination.

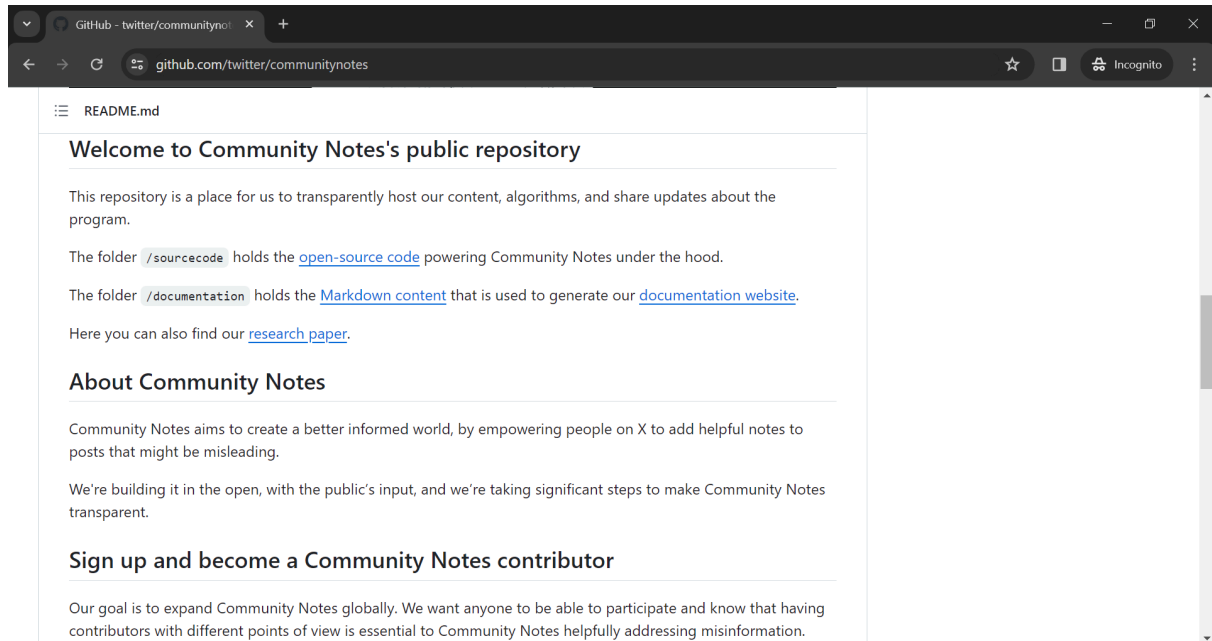


Figure 4. GitHub repository for Community Notes [5]

All software is vulnerable. Both flaw and power of proprietary software comes from the same source - its closed control. The closed-source model offers a tightly managed development lifecycle. This allows for rigorous testing and security measures to ensure the product is safe and stable before it's released for the general public. However, it also means that the responsibility for identifying and addressing vulnerabilities rests solely with the company that owns and controls the software.

In contrast, open-source software leverages the collective expertise and vigilance of a global community of developers, enhancing the capacity to rapidly identify and resolve security issues. This collaborative approach fosters a culture of continuous improvement and security in open-source projects.

2. The case for modern interaction

In the digital age, the concept of "interaction" has evolved far beyond typical user engagement on social media platforms. While traditional social media focuses on the interactions between individuals, the modern understanding of interaction extends its meaning to encompass a broader and more profound scope. This evolved perspective acknowledges that interaction is not only about connecting with friends, sharing photos, or posting status updates but also includes the collective effort required for fostering, maintaining, and developing software-social platforms.

Modern software aims to foster a sense of community and collective growth among users. Modern software encourages active participation in project creation, which goes far beyond liking and sharing. Even platforms like Stack Overflow and Youtube foster the sense of participation by rewarding users' activity with badges and trophies. This simple way of engaging people extends to individuals contributing their expertise, time, and ideas towards the collective development of a platform that reflects their values and priorities. By fostering the sense of participation and inclusiveness users are encouraged to speak their minds freely without the fear of deplatformation. By creating a space where users can actively participate in the development process, modern interaction not only redefines the relationship between users and technology but also empowers individuals to take an active role in shaping their digital environment. This empowerment, in turn, serves as a catalyst for a more democratic, user-focused, and transparent approach to social media and software development.

In the subsequent sections of this thesis, we will delve deeper into the practical implementation and impact of this modern understanding of interaction, exploring how it can enhance the social media landscape and empower users to take control of their digital footprint.

3. Technologies used for the implementation of the project

3.1. JavaScript

JavaScript is a high-level programming language primarily used in web development. It solved the problem developers had if they wanted dynamic content for their HTML and CSS driven website. Initially designed for client-side web applications, JavaScript has evolved to include server-side applications through environments like Node.js. JavaScript is single-threaded by design. However, JavaScript is also non-blocking, asynchronous, and concurrent. This might seem contradictory, but JavaScript achieves this by using an event loop and a call stack. JavaScript's compatibility with major browsers, along with its support for event-driven, functional, and imperative programming styles, makes it a fundamental tool in modern web development. Its ecosystem is rich in various frameworks and libraries that can be used for efficient UI design, computer graphics and even game development.

3.2. React

React is an open-source, front-end library developed by Facebook, widely recognized for its role in simplifying the creation of interactive user interfaces. React uses component-based architecture which enables developers to build reusable UI components, leading to more efficient code management and easier debugging. React uses virtual DOM feature, which optimizes rendering and improves application performance by minimizing direct manipulation of the DOM. React's unidirectional data flow ensures a more predictable state management, making it easier to track changes and debug. As a JavaScript library, React implements single threaded design with the possibility of using asynchronous features.

3.3. Node.js

Node.js, created by Ryan Dahl in 2009, is a free, cross-platform JavaScript environment used for executing web applications outside of a browser. It uses Google's V8 engine and the libUV platform abstraction layer, Node.js provides developers with a robust suite of tools for the non-blocking, event-driven I/O model. This makes it both efficient and easy to handle heavy-in-data applications through its asynchronous, event-driven architecture. Node.js excels as a server-side proxy, capable of managing numerous simultaneous connections efficiently without blocking. It's particularly effective in scenarios like proxying various services with different response times or aggregating data from multiple sources. Node.js is built using Google's V8 JavaScript engine, the libUV layer, and its core library is written in JavaScript.

3.4. Express

Express.js, often referred to simply as Express, is a minimalistic and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It is known for simplifying the server-side scripting process through its numerous middleware modules, enabling easy management of HTTP requests and responses. Express streamlines the routing of web requests, providing a more efficient way to handle different HTTP verbs and URLs, and thus facilitating the development of both web applications and RESTful APIs.

3.5. Multer

Multer is a middleware for Node.js that facilitates the handling of file uploads in web applications. It is specifically designed for use with Express.js and is widely used for managing multipart/form-data, which is primarily used for uploading files. Multer provides an easy and flexible way to upload files to the server by adding a body object and a file or files object to the request object (req). This makes it intuitive to access the uploaded files and their respective metadata.

3.6. Postgres

PostgreSQL, often referred to as Postgres, is an advanced, open-source relational database management system (RDBMS) known for its reliability, robustness, and performance. It offers a comprehensive range of features to securely store and scale the most complicated data workloads. Postgres supports advanced data types and performance optimization features, making it suitable for handling large volumes of data with high concurrency.

3.7. Docker

Docker is an innovative platform that changes the way software is developed and deployed. It utilizes containerization technology to package applications and their dependencies into a single, lightweight unit called a container. This approach ensures consistency across multiple development, testing, and production environments, and simplifies configuration. Docker containers are isolated from each other and the host system, making them secure and efficient. They can run on any machine that has Docker installed, regardless of the underlying operating system, leading to significant improvements in portability and scalability.

4. Platform architecture

4.1. Backstory of JavaScript and Node runtime

“JavaScript was created in 1995 by Brendan Eich with the goal of adding an easy to learn scripting language to the Netscape browser. It is most well known for building front-end web applications because it is the only language (other than WebAssembly) that is natively supported in browsers.”[6][7]

In September of 2008 Google Chrome introduced the V8 engine. V8 engine completely changed the way JavaScript was executed by introducing just-in-time (JIT) compilation. Unlike traditional interpreters that execute code line-by-line, or ahead-of-time compilers that compile the code before execution, V8 compiles JavaScript directly to native machine code at runtime. This approach significantly improves performance by reducing the execution time. This made it a viable option for high performance applications both in the browser and server-side.

In May of 2009, Ryan Dahl introduced Node.js - a server-side runtime for JavaScript built on top of V8 that included an event loop. This was a unique concept for the time and allowed you to write event-driven non-blocking code. Because of these characteristics Node.js became known as a great solution for building real-time web applications that scale and it also made it possible for developers to build their entire web application stack with a single programming language.

Today, techstacks like PERN stack (Postgres, Express, React, Node) or MEAN stack (MongoDB, Express, Angular, Node) allow you to build an entire application using a single programming language - JavaScript and a database of your choice.

4.2. ShaReCon architecture

4.2.1. Platform architecture

For ease of accessibility the platform was decided to be a web application. The frontend has been implemented in React, a JavaScript library. The backend has been made with Node.js which is a JavaScript runtime environment and TypeScript, a superset of JavaScript. Database of choice is Postgres.

As mentioned before, the focus was put into accessibility and ease of development of the platform. Therefore, all of the crucial parts - frontend, backend and database has been containerized using Docker. Docker allows for packaging and running applications regardless of the host environment.

By creating isolated environments for the application developers can run them on any device without the need to set the environment up manually. As Docker comes with a wide variety of so-called images - packages that can be downloaded into an isolated docker system, it is very easy to set up such a reusable system with a single script as shown on Figure 25 and 27. Using Docker, the author prepared scripts with such dedicated environments that can be reused by developers and ultimately ease both the development process and developer collaboration. During the initialization of the platform it is sufficient to run a single startup.sh script as seen on Figure 22 which will execute the three necessary docker-compose files (Figure 23, 24, 26) that are the backbone of the platform's docker engine.

As presented on Figure 5, the platform architecture consists of frontend, backend and database all containerized using Docker technology.

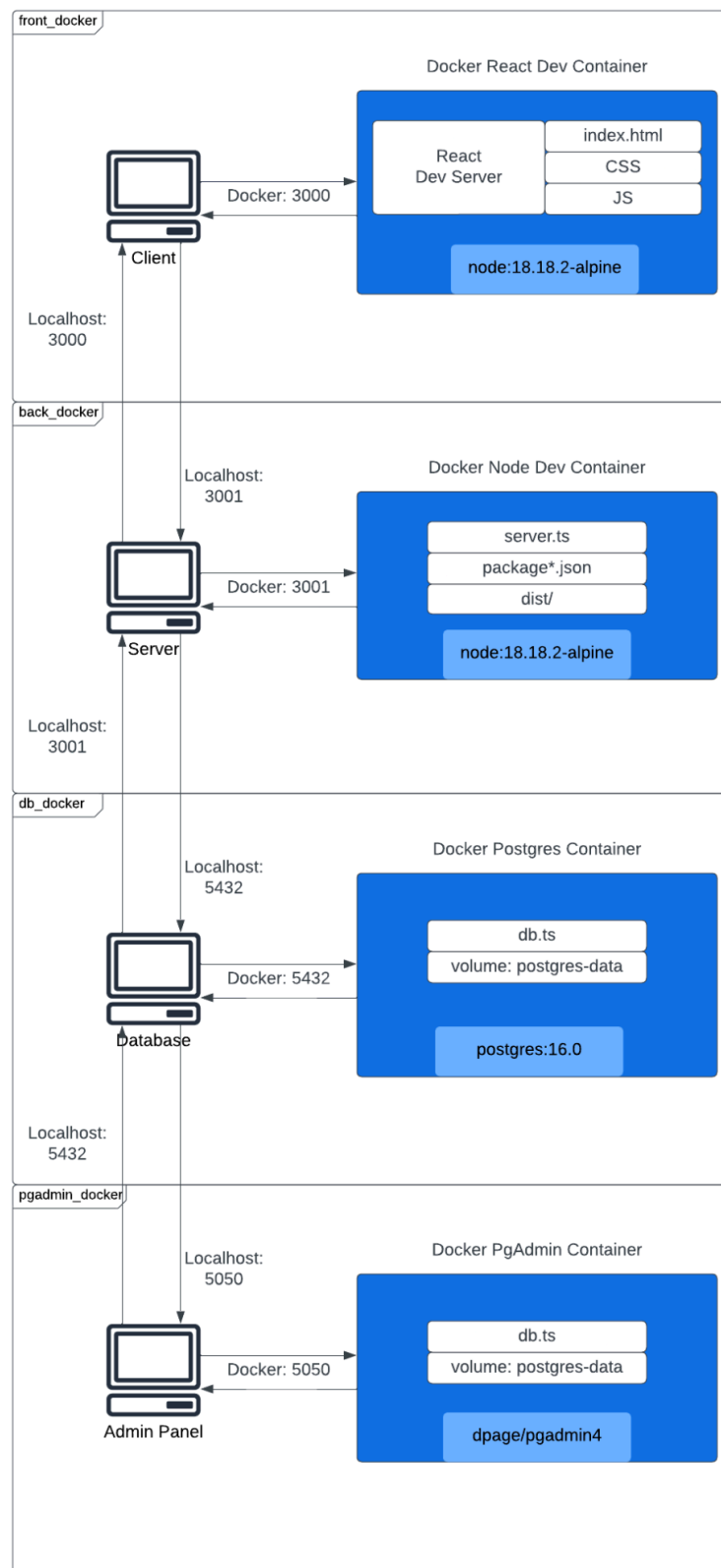


Figure 5. Platform architecture (visualized with LucidChar - source: author's work)

As presented on Figure 6a, the directory structure of the project consists of frontend and backend code. The database configuration is included inside of the backend directory.

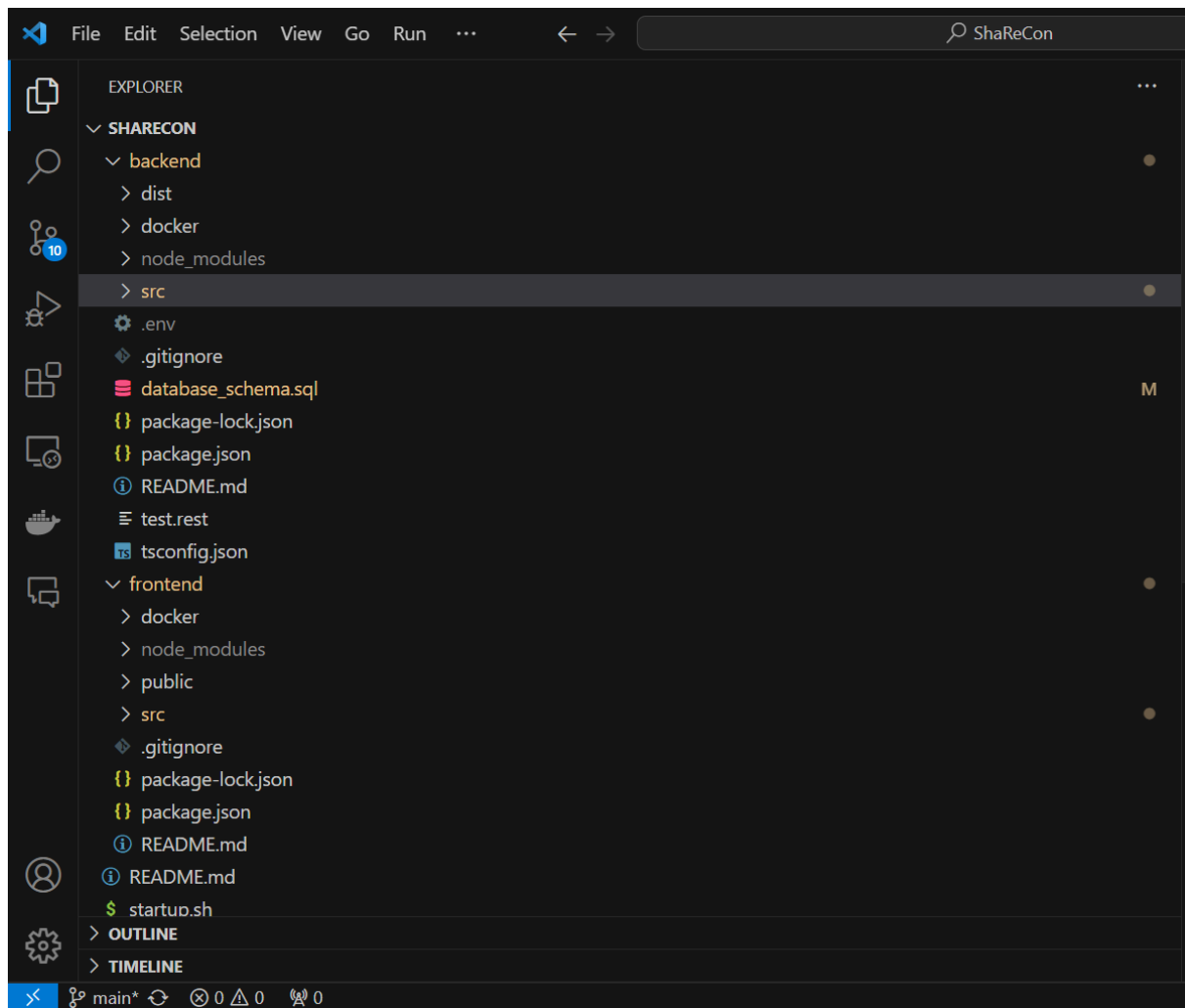


Figure 6a. Directory structure (presented with VS Code - source: author's work)

As presented on Figure 6b, the platform is clearly divisible between frontend and backend. At the same time it is easy to setup both of the environments using the startup/setup script presented on both Figure 6a and Figure 22.

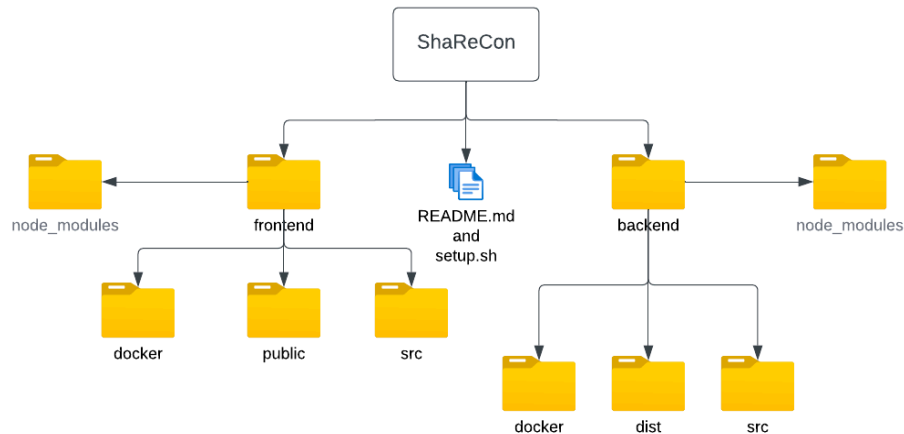


Figure 6b. Directory structure (visualized with LucidChart - source: author's work)

4.2.1.1. Functionalities Use Cases diagram

The following diagram describes the use cases for feed activities for different user roles. There are two roles - Viewer and User. They are differentiated by the presence of active cookie token in the browser. Viewer has not been authenticated and therefore is treated as an anonymous user. User, on the other hand, has been authenticated and has a valid token corresponding to the token in the browser.

As presented on Figure 7a and 7b, there is a clear difference between what React and other technologies provide and what has to be implemented to make the platform work both efficiently and effectively.

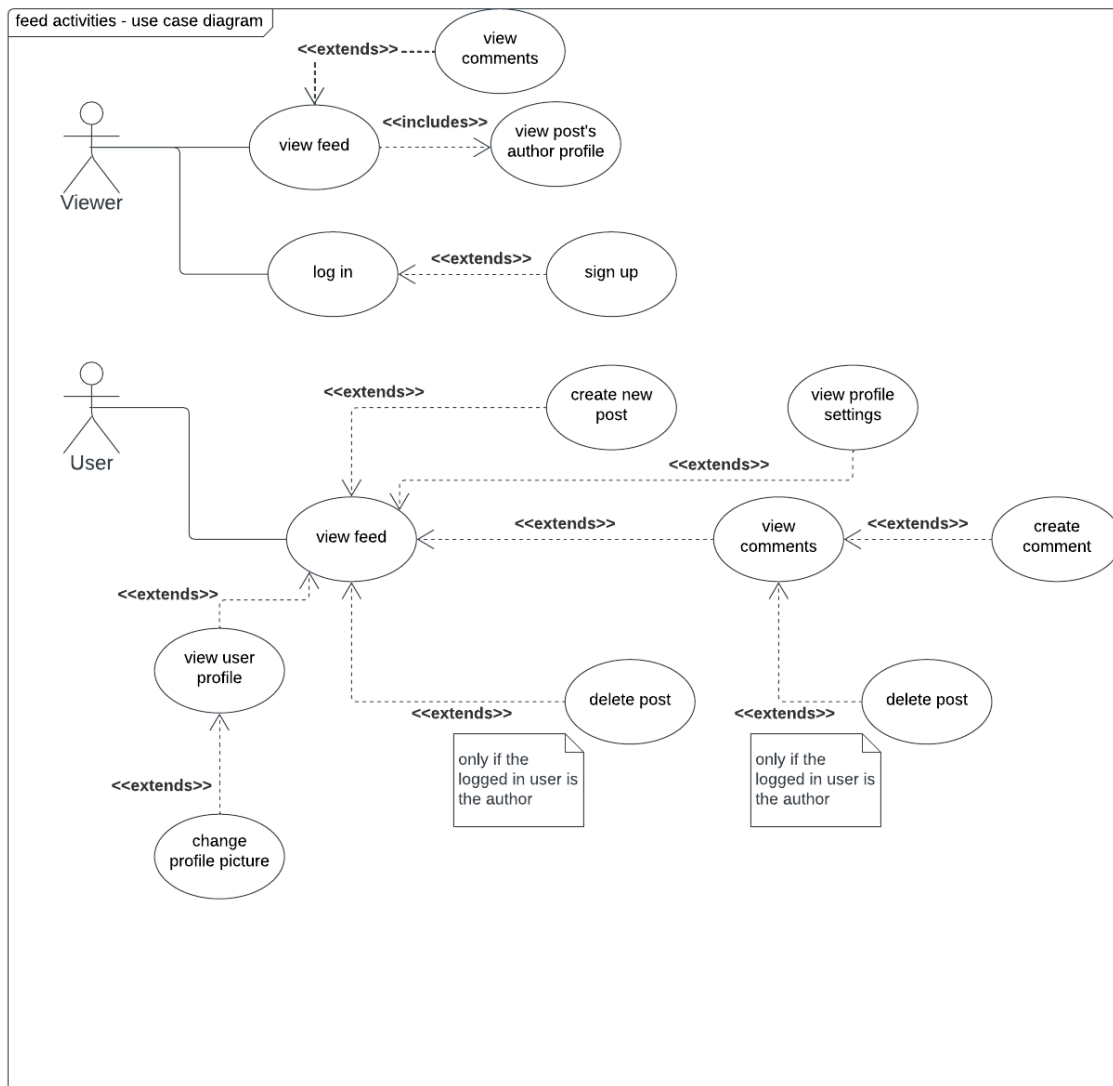


Figure 7a. Functionalities use case diagram (visualized with LucidChart - source: author's work)

Application that has no underlying implementation of crucial functionality has no practical use for users. Presented below Figure 7b, marks in red what has been implemented by the author to make the platform work

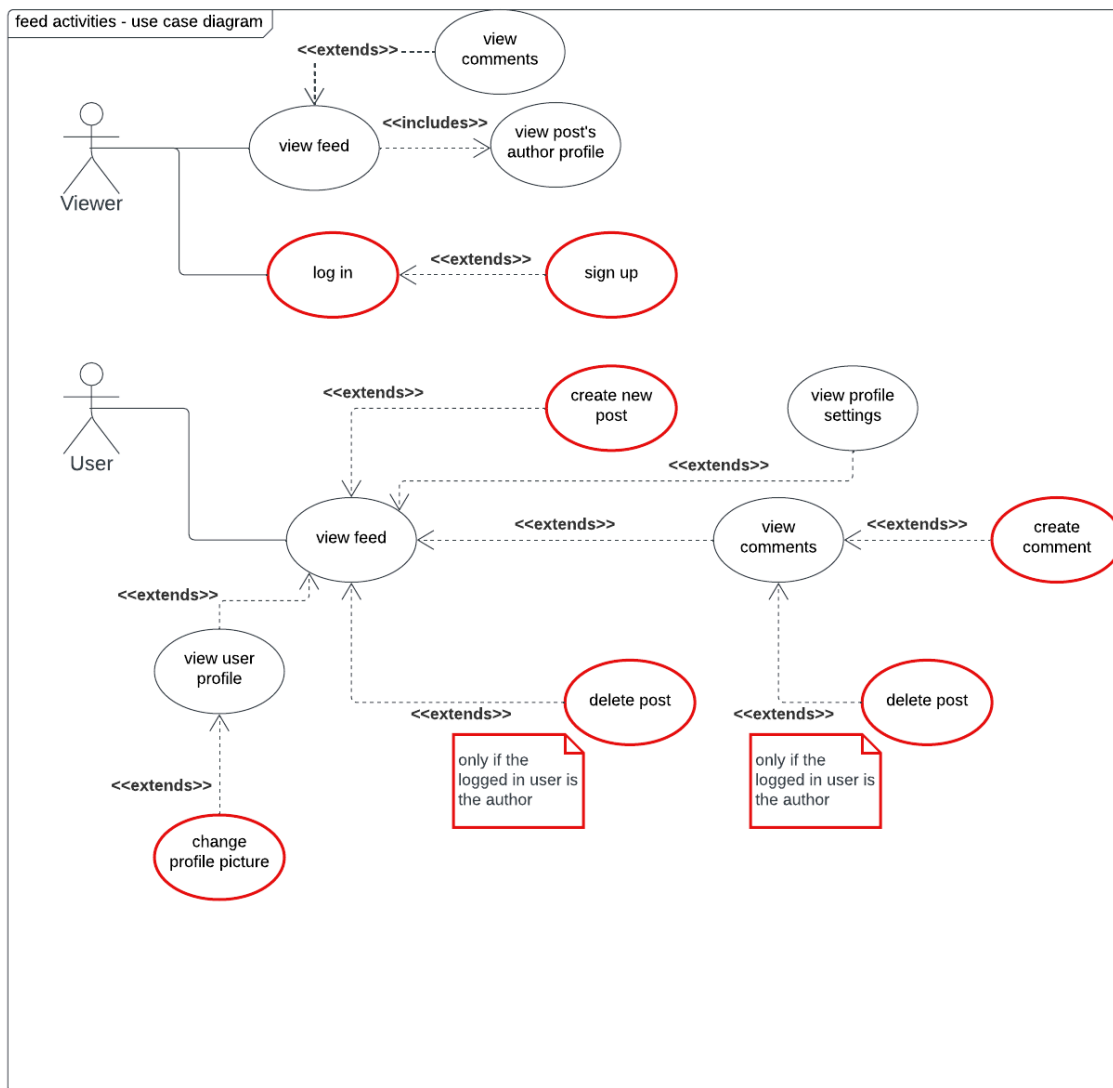


Figure 7b. Functionalities use case diagram implemented by author (visualized with LucidChart - source: author's work)

Functionalities of displaying content on the User Interface can be easily handled with React. However, all interaction that is performed on the website that involves modifying both its content and the database records must be implemented as it is later described.

4.2.1.2. Sequence diagrams

Figure 8, presents the sequence of actions that need to be performed in order to post new content to the website.

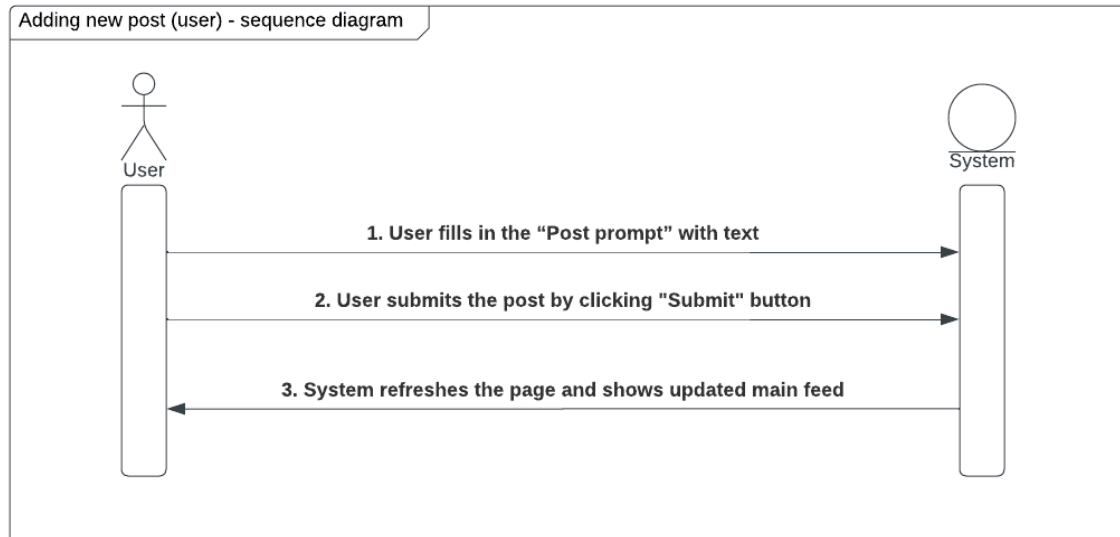


Figure 8. Sequence diagram for adding new post (visualized with LucidChart - source: author's work)

Below is presented the basic flow of action as seen on diagram from Figure 8.

Use case	Adding new post
Description	User wants to add a new post
Actors	User
Goals	Add new post, display updated feed
Pre-conditions	User is logged in, System is up and running
Post-conditions	User adds new post and displays the updated feed
Basic flow	<ol style="list-style-type: none">1. Actor fills in the "Post prompt" with text and submits the post by clicking "Submit" button2. System refreshes the page and shows updated main feed

Figure 9, presents the sequence of actions that need to be performed in order for the Viewer to be registered on the platform. The next page includes the basic flow of the action.

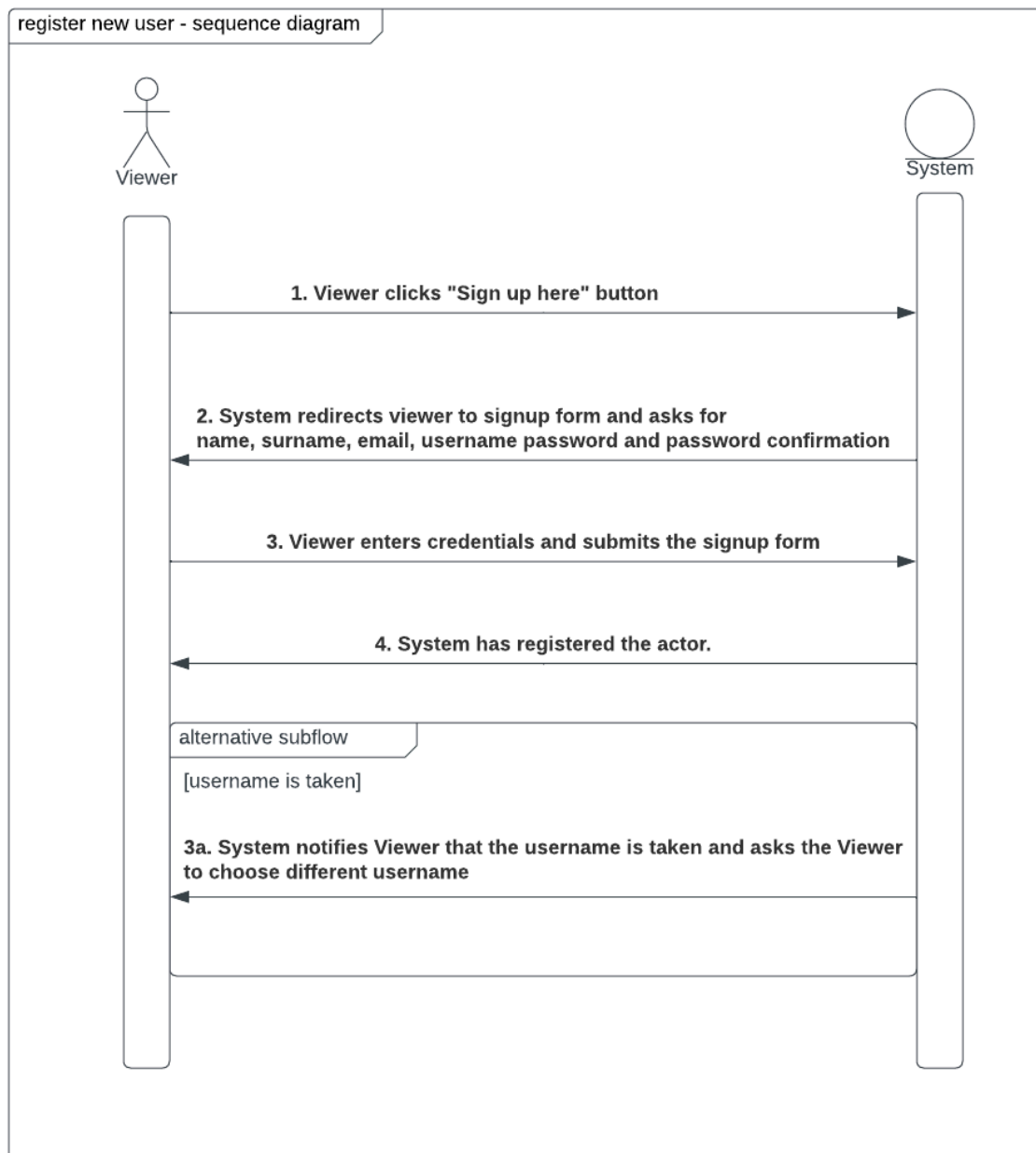


Figure 9. Sequence diagram for registering new user (visualized with LucidChart - source: author's work)

Below is presented the basic flow of action as seen on diagram from Figure 9.

Use case	Registering new user
Description	Viewer wants to register
Actors	Viewer
Goals	Create account
Pre-conditions	System is up and running
Post-conditions	Viewer creates account and becomes User
Basic flow	<ol style="list-style-type: none">1. Actor clicks "Sign up here" button2. System redirects viewer to signup form and asks for name, surname, email, username password and password confirmation3. Viewer enters credentials and submits the signup form4. System has registered the actor.
Alternative subflow	<ol style="list-style-type: none">3a. System notifies viewer that username is taken and asks the actor to choose different username

5. Platform implementation

5.1. Open-source code

The platform is fully open-sourced and available on github. However, the project is not Libre meaning Free to use. This is important because despite the code being open for display it cannot be reused until there is appropriate license included.

Figure 10 presents the code repository of the platform hosted on Github.com.

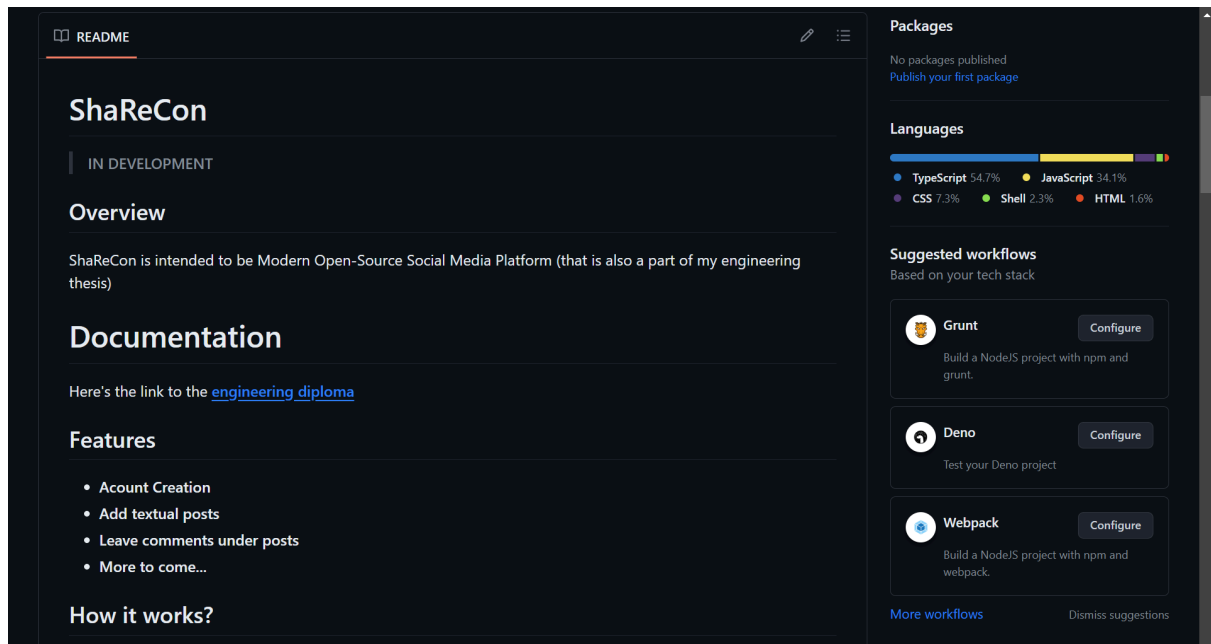


Figure 10. Project repository on github (source: author's work published on github [8])

5.2. React frontend

Application UI was done using the latest React version. According to the 2022 *Stack Overflow Developer Survey*, React is the most wanted web technology to learn with 68.19% (15 873 responses)[9] satisfaction rate. In the *State of JavaScript 2021* survey React placed first in the usage category with 80% usage rate (out of 16 085 respondents)[10].

Blindly following trends is not advisable for building any computer program, however surveys like these provide insight into the current state of the IT industry (at least a part of the industry).

React was developed at Facebook in 2013 and has amassed a massive community around itself since then. This is often a good reason to choose a technology for a stack because there is a pool of resources to refer to maintained by a large number of actively working developers. The library's main purpose is to help with building components as reusable parts of the UI. Component in React is just a JavaScript function whose return value is a special syntax called JSX - composition of HTML and JS.

5.2.1. Account creation

React itself is a great library for utilizing reusable components, but it only helps with the UI design and efficiency as shown on the functionalities use case diagram as presented on Figure 7a. To implement the functionality of account creation it is mandatory to provide necessary functionality for handling user authentication as shown on the diagram 7b.

Anyone who wants to actively contribute to the platform can create an account. To do so, a person can navigate to the signup page, by first opening the login form. From there, by clicking the *Sign up here* link, we are navigated to the Signup form.

Figure 11a displays the graphical user interface of the login form.

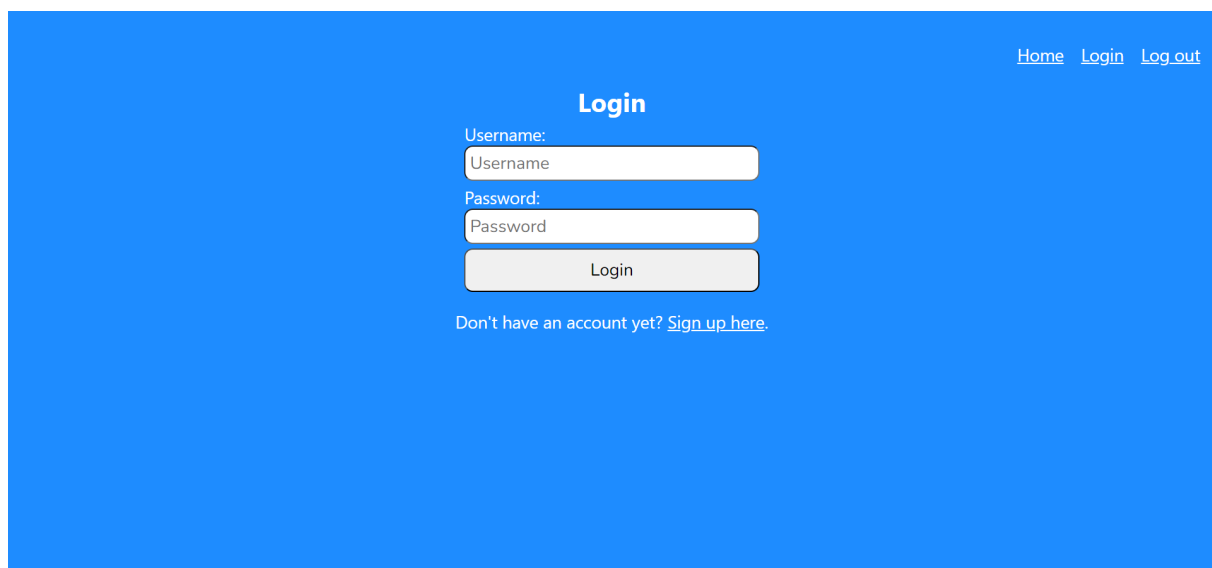
The image shows a web application with a solid blue background. In the top right corner, there are three links: "Home", "Login", and "Log out". In the center, the word "Login" is displayed in white. Below it, there are two input fields: "Username:" and "Password:". Each input field has a light gray border and a light gray placeholder text. Below the "Password:" field is a "Login" button with a light gray border and a light gray background. At the bottom, there is a link that says "Don't have an account yet? [Sign up here.](#)".

Figure 11a. Log in form (project implementation, localhost:3000/login - source: author's work)

Figure 11b shows the graphical user interface of the signup form.

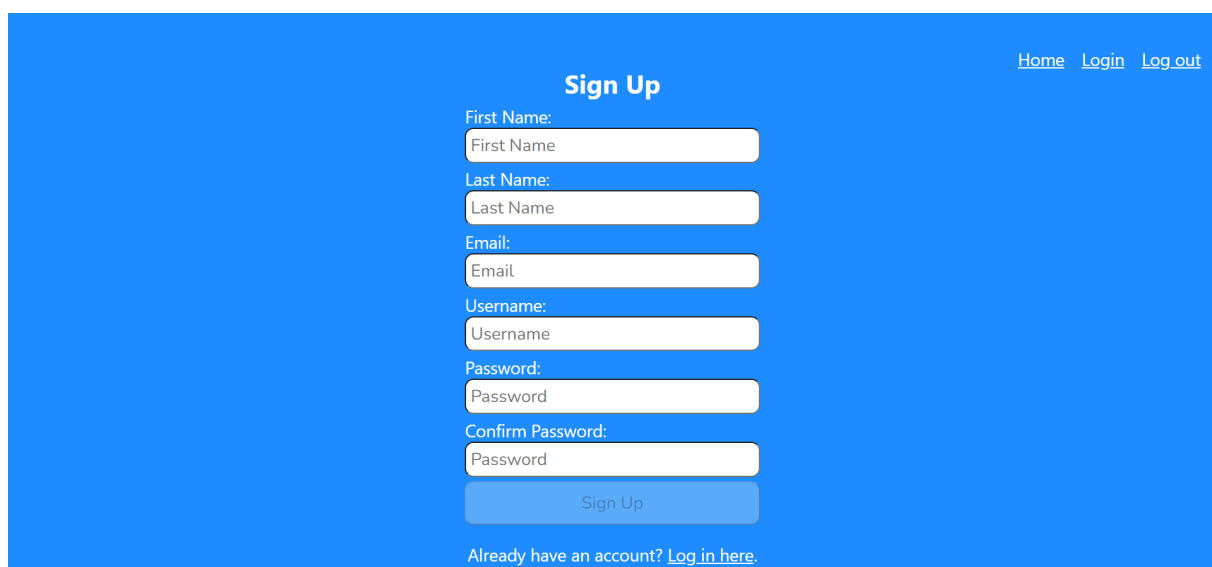
The image shows a web application with a solid blue background. In the top right corner, there are three links: "Home", "Login", and "Log out". In the center, the words "Sign Up" are displayed in white. Below it, there are six input fields: "First Name:", "Last Name:", "Email:", "Username:", "Password:", and "Confirm Password:". Each input field has a light gray border and a light gray placeholder text. Below the "Confirm Password:" field is a "Sign Up" button with a light blue border and a light blue background. At the bottom, there is a link that says "Already have an account? [Log in here.](#)".

Figure 11b. Sign up form (project implementation, localhost:3000/signup - source: author's work)

To create an account we are required to fill in the form with basic personal information like first name, last name, email and password. Additionally, users are asked to confirm their password input with the *Confirm Password* field, which increases the correctness of the provided password.

As shown on Figures 11b, 12 and 13 sign up form requires providing:

- valid email address (regex validation)
- valid username (regex validation & available username)
- valid password (regex validation)
- password confirmation (matching password)

```
// validate username
Complexity is 5 Everything is cool!
useEffect(() => {
  const result = USER_REGEX.test(username);

  Complexity is 4 Everything is cool!
  async function fetchData() {
    try {
      const response = await fetch(`http://localhost:3001/api/users/${username}`);
      console.log(`firstname: ${firstName}, lastname: ${lastName}, email: ${email}, username: ${username}, password: ${password}`);

      if (response.ok) {
        setAvailableUsername(false);
      } else {
        setAvailableUsername(true);
      }
    } catch (error) {
      setError('Username Already Taken.');
```

Figure 12. Checking availability of the username (project implementation, source code - source: author's work)

```
const EMAIL_REGEX = /^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}(\?:\.[A-Za-z]{2,})?$/;
const USER_REGEX = /^[A-z][A-z0-9_-]{3,23}$/;
const PWD_REGEX = /^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%])-.{8,24}$/;
```

Figure 13. Regex patterns for sign up form (project implementation, source code - source: author's work)

As shown on Figure 14a, if the username to be registered is already taken - present in the database, the system clearly notifies about it. Person should then enter a different username. Figure 14b, shows the source code behind this functionality.

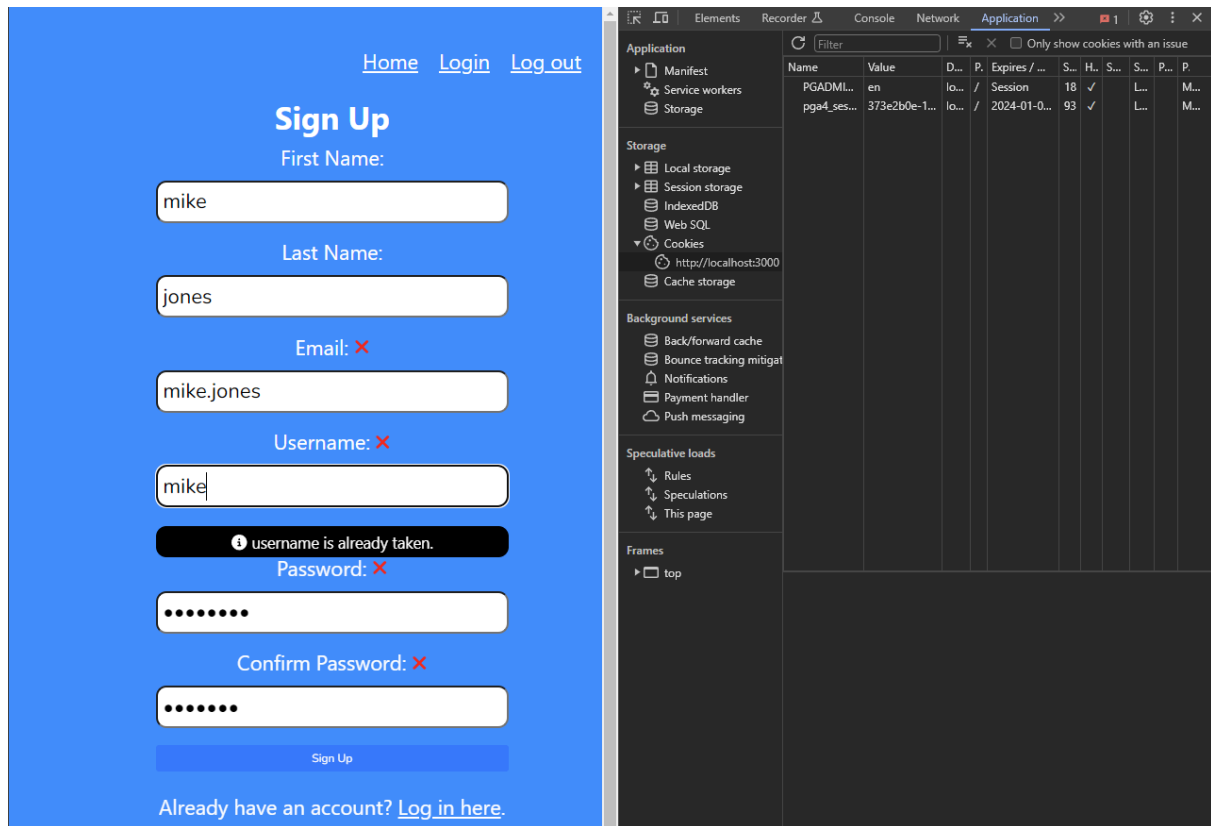


Figure 14a. Invalid sign up form (project implementation with developer tools opened on application/cookies, localhost:3000/login - source: author's work)

```

○○○

const accountController = require('./accountController'); // Adjust the path as needed
const bcrypt = require('bcryptjs'); // For password hashing and comparison
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();
const { createTokens } = require('../utils/JWT');
const Account = require('../models/account');

app.use(express.json());
app.use(cookieParser());

const AuthController = {
  login: async (req: any, res: any) => {
    const { username, password } = req.body;

    try {
      // Check if the username exists
      const account = await accountController.loginByUsername(req.body.account.username);

      if (!account) {
        return res.status(401).json({ message: 'Invalid username or password' });
      }

      // Compare the provided password with the stored hashed password
      const passwordsMatch = await bcrypt.compare(req.body.account.password, account.password);

      if (passwordsMatch) {
        const accessToken = createTokens(req.body.account.username);
        Account.updateAccessToken(req.body.account.username, accessToken);

        // Authentication successful
        res.cookie('jwt', accessToken, { sameSite: 'None', httpOnly: false, maxAge: 24 * 60 * 60 * 1000 });
        return res.status(200).json({ auth: true, token: accessToken, result: account });
      } else {
        // Authentication failed
        return res.status(500).json({ auth: false, message: "no user exists" });
      }
    } catch (error) {
      console.error(error);
      return res.status(500).json({ auth: false, message: "no server response" });
    }
  },

  signup: async (req: any, res: any) => {
    const reqNew = {
      firstName: req.body.account.firstname,
      lastName: req.body.account.lastname,
      email: req.body.account.email,
      username: req.body.account.username,
      password: req.body.account.password,
    };
    console.log(reqNew);

    const accessToken = createTokens(reqNew.username);

    try {
      // Create a new account, checking if the username is available
      const newAccount = await accountController.signup(reqNew.username, reqNew.firstName, reqNew.lastName,
        reqNew.email, reqNew.password, accessToken);

      if (newAccount && !newAccount.error) {
        res.cookie('jwt', accessToken, { sameSite: 'None', httpOnly: false, maxAge: 24 * 60 * 60 * 1000 });
        return res.status(201).json({ auth: true, token: accessToken, result: newAccount });
      } else if (newAccount && newAccount.error === 'Username is already taken') {
        res.status(400).json({ message: 'Username is already taken' });
      } else if (!reqNew.firstName || !reqNew.lastName || !reqNew.email || !reqNew.username || !reqNew.password) {
        res.status(400).json({ message: 'Field(s) are missing' });
      } else {
        res.status(500).json({ error: 'Internal server error' });
      }
    } catch (err) {
      console.error(err);
      res.status(500).json({ error: 'Internal server error' });
    }
  },
};

module.exports = AuthController;

```

Figure 14b. Handling authentication on the backend (image generated with <https://carbon.now.sh/>; project implementation of the authentication mechanism, localhost:3001/login - source: author's work)

The authentication is handled on a different API route than other functionalities as shown on Figures 15a, 15b, 15c, 15d and 15e. This is necessary because it is a good software engineering practice to differentiate between the authentication functionality (signing in, signing up, routes access authorization) and usability function (posting content, deleting content, changing profile settings etc)

```
const express = require("express");
const AuthController = require('../controllers/authController');
const router = express.Router();

router.post("/login", AuthController.login);
router.post("/signup", AuthController.signup);

module.exports = router;
```

Figure 15a. Specifying API routes for authentication endpoints (project implementation of the server side - source: author's work, screenshot of the source code)

```
const express = require("express");
const AccountController = require('../../controllers/accountController');
const multer = require('multer');
const upload = require('../../utils/multerConfig');
const router = express.Router();

router.get('/:username', AccountController.getByUsername);
router.get('/:isValid/:token', AccountController.getByToken);
router.get('/', AccountController.getAllAccounts);
router.post('/', AccountController.createNew);
router.put('/:username', AccountController.updateByUsername);
router.delete('/:username', AccountController.deleteByUsernameAndPassword);

router.post('/uploadPfp/:username', upload.single("profile_picture"), AccountController.uploadImage);

module.exports = router;
```

Figure 15b. Specifying API routes for account functionalities endpoints (project implementation of the server side - source: author's work, screenshot of the source code)

```

const express = require("express");
const CommentController = require('../controllers/commentController');
const router = express.Router();

router.get('/', CommentController.getAllComments);
router.get('/:id', CommentController.getById);
router.get('/post/:id', CommentController.getByPostId);
router.get('/acc/:username', CommentController.getByUsername);
router.post('/', CommentController.createNewWithToken);
router.delete('/:id', CommentController.deleteById);

module.exports = router;

```

Figure 15c. Specifying API routes for comment functionalities endpoints (project implementation of the server side - source: author's work, screenshot of the source code)

```

const express = require("express");
const PostController = require('../controllers/postController');
const router = express.Router();

router.get('/:id', PostController.getById);
router.get('/acc/:username', PostController.getByUsername);
router.get('/', PostController.getAllPosts);
// router.post('/', PostController.createNew);
router.post('/', PostController.createNewWithToken);
router.put('/:id', PostController.updateById);
router.delete('/:id', PostController.deleteById);

module.exports = router;

```

Figure 15d. Specifying API routes for post functionalities endpoints (project implementation of the server side - source: author's work, screenshot of the source code)

```

○ ○ ○

const express = require('express')
const cors = require('cors'); // Import the cors middleware
const bodyParser = require('body-parser');
const bcrypt = require('bcryptjs')
const session = require('express-session')
const cookieParser = require('cookie-parser');
var pool = require('./db')
const { validateToken } = require('./utils/JWT')
const path = require('path');
const app = express();
const port = 3001;

app.use(cors({ origin: 'http://localhost:3000' }));
app.use(express.json());
app.use(bodyParser.json());
app.use(cookieParser());
app.use(session({
  secret: 'secret',
  resave: false,
  saveUninitialized: false,
  cookie: {
    secure: false,
    maxAge: 24 * 60 * 60 * 1000
  }
})))

app.use('/assets', express.static(path.join(__dirname, '../assets/')));

// CONTROLLER ROUTES
const accountApiRouter = require('./routes/api/accountApiRoutes');
app.use('/api/users', accountApiRouter);
const postApiRouter = require('./routes/api/postApiRoutes');
app.use('/api/posts', postApiRouter);
const commentApiRouter = require('./routes/api/commentApiRoutes');
app.use('/api/comments', commentApiRouter);

const authRoutes = require('./routes/authRoutes'); // Import the
Authenticated endpoint for user login
app.use('/', authRoutes);

// Defines a protected route
app.get('/isUserAuth', validateToken, (req: any, res: any) => {
  res.send("You are authenticated");
});

// Endpoint for checking active session
app.get('/check-session', validateToken, (req: any, res: any) => {
  res.json('session active');
});

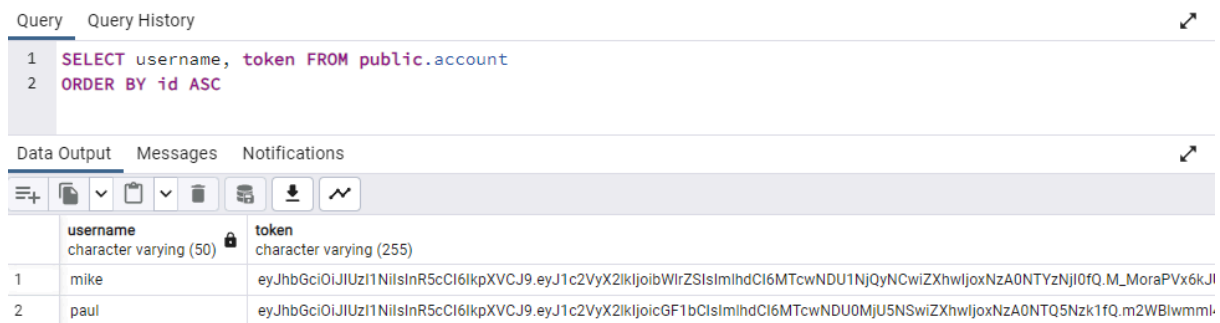
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});

```

Figure 15e. Server configuration on the backend (image generated with <https://carbon.now.sh/>; project implementation of the server side - source: author's work)

5.2.2. Signing in

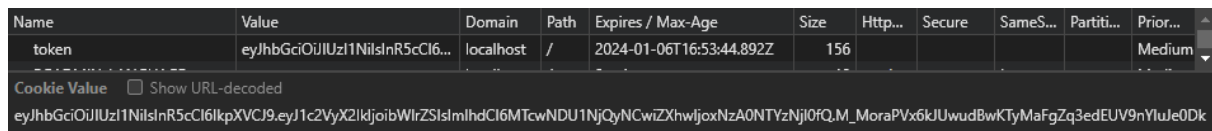
If the user already has an account he can sign in directly from the Sign in form by providing his credentials. Once the user successfully logs in (or creates an account), a token is created based on his credentials and stored both in the cookies section of the browser and in the database. After successful authentication, the user gets redirected to the home page. As shown on Figure 16a, the username record includes the username and updated token - the same as in the browser window -> developer tools -> application -> cookies (as shown on Figure 16b).



The screenshot shows the pgAdmin interface. The 'Query' tab is active, displaying a SQL query: `1 SELECT username, token FROM public.account` and `2 ORDER BY id ASC`. Below the query, the 'Data Output' tab shows the results of the query. The results are presented in a table with two columns: 'username' and 'token'. The 'username' column is labeled 'character varying (50)' and the 'token' column is labeled 'character varying (255)'. There are two rows of data: one for 'mike' and one for 'paul'. The 'token' values are long, alphanumeric strings.

	username character varying (50)	token character varying (255)
1	mike	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoibWlrZSIsImhhdCI6MTcwNDU1NjQyNCwiZXhwIjozA0NTYzNjI0fQ.M_MoraPVx6kJl
2	paul	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoicGF1bCI6MTcwNDU0MjU5NSwiZXhwIjozA0NTQ5Nzk1fQ.m2WBlwmmk

Figure 16a. Updated token in the database (project implementation pgadmin panel, localhost:5050 - source: author's work)



The screenshot shows the 'Cookies' tab in a browser's developer tools. It displays a single cookie with the name 'token'. The 'Value' column shows a long, alphanumeric string. The 'Domain' is 'localhost' and the 'Path' is '/'. The 'Expires / Max-Age' is '2024-01-06T16:53:44.892Z'. The 'Size' is '156'. The 'Http...' and 'Secure' columns are empty. The 'SameS...' column is 'SameSite'. The 'Partiti...' column is empty. The 'Prior...' column is 'Medium'. Below the table, the 'Cookie Value' is shown as 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoibWlrZSIsImhhdCI6MTcwNDU1NjQyNCwiZXhwIjozA0NTYzNjI0fQ.M_MoraPVx6kJlUwudBwKTyMaFgZq3edEUv9nYluJe0Dk'.

Name	Value	Domain	Path	Expires / Max-Age	Size	Http...	Secure	SameS...	Partiti...	Prior...
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoibWlrZSIsImhhdCI6MTcwNDU1NjQyNCwiZXhwIjozA0NTYzNjI0fQ.M_MoraPVx6kJl	localhost	/	2024-01-06T16:53:44.892Z	156			SameSite		Medium

Figure 16b. Cookie token present in the browser after successful sign in (project implementation with developer tools opened on application/cookies, localhost:3000/login - source: author's work)

Handling of the Account functionality was implemented entirely from scratch (as shown on the Figure 16c, 16d) as there is no sufficient tool that automates this development process. Only the approaches that utilize containerization - like Docker, allow for easier deployment and connection between backend and database.

```

const Account = require('../models/account');
const { createTokens } = require('../utils/JWT');
const upload = require('../utils/multerConfig');

module.exports = {
  // Get all accounts
  getAllAccounts: async (req: any, res: any) => {
    try {
      const accounts = await Account.getAllAccounts();
      res.status(200).json({ accounts });
    } catch (err) {
      console.error(err);
      res.status(500).json({ error: 'Internal server error' });
    }
  },

  // Get Account by username
  getByUsername: async (req: any, res: any) => {
    const { username } = req.params; // Use req.params to get the username from the route params

    try {
      const data = await Account.getByUsername(username);

      if (data) {
        res.status(200).json({ account: data });
      } else {
        res.status(404).json({ message: 'Account not found' });
      }
    } catch (err) {
      console.error(err);
      res.status(500).json({ error: 'Internal server error' });
    }
  },

  // Get Account by token
  getByToken: async (req: any, res: any) => {
    const { token } = req.params; // Use req.params to get the username from the route params

    try {
      const data = await Account.getByToken(token);

      if (data) {
        console.log('accountController: ' + token)
        console.log('accountController: ' + JSON.stringify(data))

        if (data.token === token) {
          res.status(200).json({ account: data });
        } else {
          res.status(404).json({ message: 'Account not found' });
        }
      }
    } catch (err) {
      console.error(err);
      res.status(500).json({ error: 'Internal server error' });
    }
  },

  // Log the user in
  loginByUsername: async (username: string) => {
    try {
      const data = await Account.getByUsername(username);

      if (data) {
        return data
      } else {
        return null;
      }
    } catch (err) {
      console.error(err);
    }
  },

  // Sign the user up
  signup: async (username: string, firstName: string, lastName: string, email: string, password: string, token: string) => {
    try {
      const data = await Account.getByUsername(username);

      if (data) {
        return console.error('Username already exists');
      }

      const newData = await Account.createNew(username, firstName, lastName, email, password, token);

      if (newData) {
        return newData;
      } else {
        return null;
      }
    } catch (err) {
      console.error(err);
    }
  },
};

```

Figure 16c. 1/2 part of the accountController functionality (image generated with <https://carbon.now.sh/>; project implementation - source: author's work)

```

    ○ ○ ○

createNew: async (req: any, res: any) => {
  // const { username, firstName, lastName, email, password } = req.body
  const reqNew = {
    firstName: req.body.account.firstName,
    lastName: req.body.account.lastName,
    email: req.body.account.email,
    username: req.body.account.username,
    password: req.body.account.password,
  }
  console.log(req.body);
  console.log(reqNew);

  const accessToken = createTokens(reqNew.username);
  console.log(accessToken);

  try {
    // Create a new account, checking if the username is available
    const newAccount = await Account.createNew(reqNew.username, reqNew.firstName, reqNew.lastName, reqNew.email,
    reqNew.password, accessToken);

    if (newAccount && !newAccount.error) {
      res.status(201).json({ message: 'Account created successfully', account: newAccount });
    } else if (newAccount && newAccount.error === 'Username is already taken') {
      res.status(400).json({ message: 'Username is already taken' });
    } else if (!reqNew.username || !reqNew.firstName || !reqNew.lastName || !reqNew.email || !reqNew.password) {
      res.status(400).json({ message: 'Field(s) are missing' });
    } else {
      res.status(500).json({ error: 'Internal server error' });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
},

uploadImage: async (req: any, res: any) => {
  const { username } = req.params;
  // const user = Account.getByUsername(username);
  console.log(req.file);
  // Check if file upload was successful
  if (!req.file) {
    return res.status(400).send({ message: 'No file uploaded.' });
  }

  // Construct the path to the saved image
  const imagePath = `../../assets/${req.file.originalname}`;
  console.log(imagePath);
  // Here you can call a function to update the user's profile with the new image path
  try {
    const updatedUser = await Account.updatePfpByUsername(username, imagePath);
    res.status(200).json({ message: 'Image uploaded successfully.', updatedUser });
  } catch (error) {
    res.status(500).json({ message: 'Error updating profile image.', error });
  }
},

// Update an account by username
updateByUsername: async (req: any, res: any) => {
  const { username } = req.params; // Get the original username from the route params
  const reqUpdates = {
    newUsername: req.body.account.username || null,
    firstName: req.body.account.firstName || null,
    lastName: req.body.account.lastName || null,
    email: req.body.account.email || null,
    password: req.body.account.password || null,
  }

  // Check if the newUsername is already taken
  const isUsernameTaken = await Account.getByUsername(reqUpdates.newUsername);
  const doesUsernameExist = await Account.getByUsername(username);

  if (!doesUsernameExist) {
    return res.status(400).json({ message: 'No such username' });
  } else if (isUsernameTaken) {
    return res.status(400).json({ message: 'Username is already taken' });
  }

  // Proceed with the update if the newUsername is available
  const updatedAccount = await Account.updateByUsername(username, reqUpdates.newUsername, reqUpdates.firstName,
  reqUpdates.lastName, reqUpdates.email, reqUpdates.password);

  if (updatedAccount) {
    res.status(200).json({ message: 'Account updated successfully', account: updatedAccount });
  } else {
    res.status(404).json({ message: 'Account not found' });
  }
},

// Delete an account by username and password
deleteByUsernameAndPassword: async (req: any, res: any) => {
  const { username } = req.params; // Get the username from the route params
  const password = req.body.account.password; // Get the password from the request body

  try {
    // Call the model method to delete the account by username and password
    console.log(password);
    const result = await Account.deleteByUsernameAndPassword(username, password);

    if (result) {
      // Account was deleted successfully or password was correct
      res.status(200).json(result);
    } else {
      // Account not found or password incorrect
      res.status(404).json({ message: 'Account not found or password incorrect' });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
},
};

```

Figure 16d. 2/2 part of the accountController functionality (image generated with <https://carbon.now.sh/>; project implementation - source: author's work)

5.2.3. Posting content

To post to the main feed and make comments under posts, users must be logged in first (Figures 17, 18). Every successful sign in and sign up, regenerates the jwt token. Token is sent as the response from the backend but also saved into the database within the UPDATE or INSERT sql statement. (Figures 17, 18, 19)

Posts and comments are handled in a very similar way.

```
updateAccessToken: async (username: string, token: string) => {
  try {
    // Fetch the old data for the user
    const oldData = await Account.getByUsername(username);

    if (!oldData) {
      return null; // Handle the case where the user does not exist
    }

    const query = {
      text: `UPDATE account
            SET token = $2
            WHERE username = $1
            RETURNING *`,
      values: [username, token],
    };

    const data = await pool.query(query);

    if (data.rows.length > 0) {
      return data.rows[0];
    } else {
      return null;
    }
  } catch (err) {
    console.error('Error in updateByUsername:', err);
    throw err;
  }
},
```

Figure 17. Source code for updating access token on the backend (project implementation, source code of account.ts - source: author's work)

```

const accountController = require('./accountController'); // Adjust the path as needed
const bcrypt = require('bcryptjs'); // For password hashing and comparison
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();
const { createTokens } = require('../utils/JWT')
const Account = require('../models/account')

app.use(express.json());
app.use(cookieParser());

const AuthController = {
  login: async (req: any, res: any) => {
    const { username, password } = req.body;

    try {
      // Check if the username exists
      const account = await accountController.loginByUsername(req.body.account.username);

      if (!account) {
        return res.status(401).json({ message: 'Invalid username or password' });
      }

      // Compare the provided password with the stored hashed password
      const passwordsMatch = await bcrypt.compare(req.body.account.password, account.password);

      if (passwordsMatch) {
        const accessToken = createTokens(req.body.account.username);
        Account.updateAccessToken(req.body.account.username, accessToken);

        // Authentication successful
        res.cookie('jwt', accessToken, { sameSite: 'None', httpOnly: false, maxAge: 24 * 60 * 60 * 1000 });
        return res.status(200).json({ auth: true, token: accessToken, result: account });
      } else {
        // Authentication failed
        return res.status(500).json({ auth: false, message: "no user exists" });
      }
    } catch (error) {
      console.error(error);
      return res.status(500).json({ auth: false, message: "no server response" });
    }
  },

  signup: async (req: any, res: any) => {
    const reqNew = {
      firstName: req.body.account.firstname,
      lastName: req.body.account.lastname,
      email: req.body.account.email,
      username: req.body.account.username,
      password: req.body.account.password,
    };
    console.log(req.body.account);

    const accessToken = createTokens(reqNew.username);

    try {
      // Create a new account, checking if the username is available
      const newAccount = await accountController.signup(reqNew.username, reqNew.firstName, reqNew.lastName,
        reqNew.email, reqNew.password, accessToken);

      if (newAccount && !newAccount.error) {
        res.cookie('jwt', accessToken, { sameSite: 'None', httpOnly: false, maxAge: 24 * 60 * 60 * 1000 });
        return res.status(201).json({ auth: true, token: accessToken, result: newAccount });
      } else if (newAccount && newAccount.error === 'Username is already taken') {
        res.status(400).json({ message: 'Username is already taken' });
      } else if (!reqNew.firstName || !reqNew.lastName || !reqNew.email || !reqNew.username || !reqNew.password) {
        res.status(400).json({ message: 'Field(s) are missing' });
      } else {
        res.status(500).json({ error: 'Internal server error' });
      }
    } catch (err) {
      console.error(err);
      res.status(500).json({ error: 'Internal server error' });
    }
  },
};

module.exports = AuthController;

```

Figure 18. Source code for handling authentication on the backend (project implementation, source code of authController.ts - source: author's work)

```

const handleLogin = async (e) => {
  e.preventDefault();

  try {
    setUsername('');
    setPassword('');
    // Send a POST request to your '/login' endpoint with the username and password
    fetch('http://localhost:3001/login', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      withCredentials: false,
      body: JSON.stringify({
        "account": {
          "username": username,
          "password": password
        }
      })
    })
    .then((response) => {
      if (response.ok) {
        // Assuming the response data is in JSON format, you can parse it
        return response.json();
      } else {
        // Handle login error and set error message
        setError('Login or password incorrect');
      }
    })
    .then((data) => {
      // 'data' will contain the response data, which includes "auth" and "token"
      // console.log(data);
      if (data) {
        // Now you can access the "auth" and "token" properties
        const { auth, token, result } = data;
        // Handle the data as needed
        if (auth) {
          // Authentication was successful, and you can use the 'token'
          // Set the 'token' in your React state or cookies
          setCookie("token", token, { path: "/", maxAge: 60 * 60 });
          // console.log(cookies);

          history.push(`/`);
        }
      } else {
        setError('No Account Found');
        // Authentication failed
        // Handle the error or show a message to the user
        console.log("Login failed");
      }
    })
  } catch (err) {
    if (!err?.response) {
      setError('No Server Response');
    } else if (err.response?.status === 400) {
      setError('Missing Username or Password');
    } else if (err.response?.status === 401) {
      setError('Unauthorized');
    } else {
      setError('Login Failed');
    }
    errRef.current.focus();
  }
};

```

Figure 19. Source code for submitting log in form on the frontend (project implementation, source code of LoginForm.jsx - source: author's work)

As mentioned previously, after successful authentication user token is created and visible on the Figure 20 and 21.

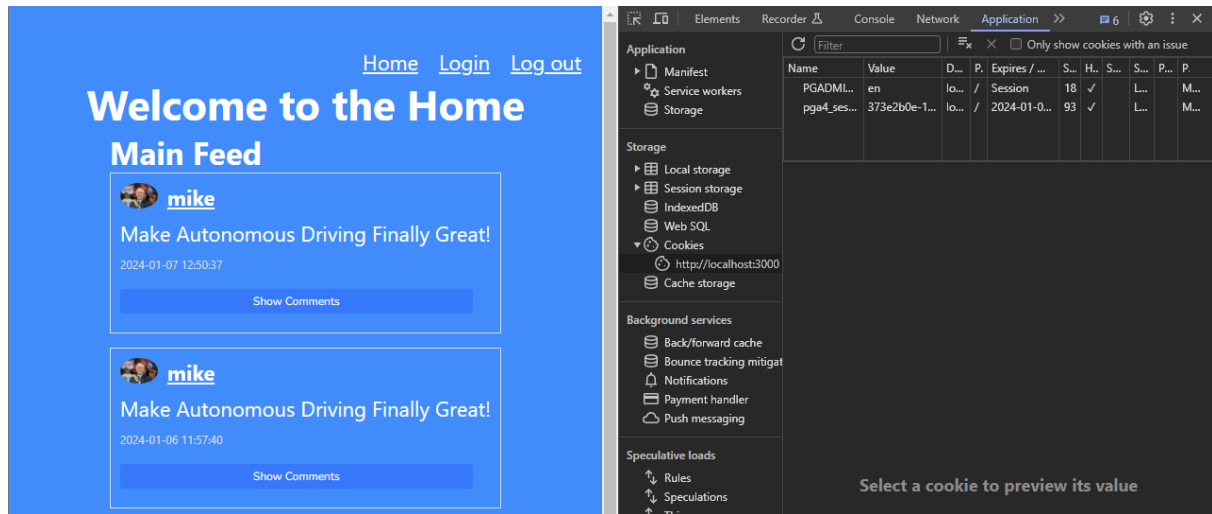


Figure 20. Main feed UI for the Viewer - there is no active cookie token (project implementation with developer tools opened on application/cookies, localhost:3000/ - source: author's work)

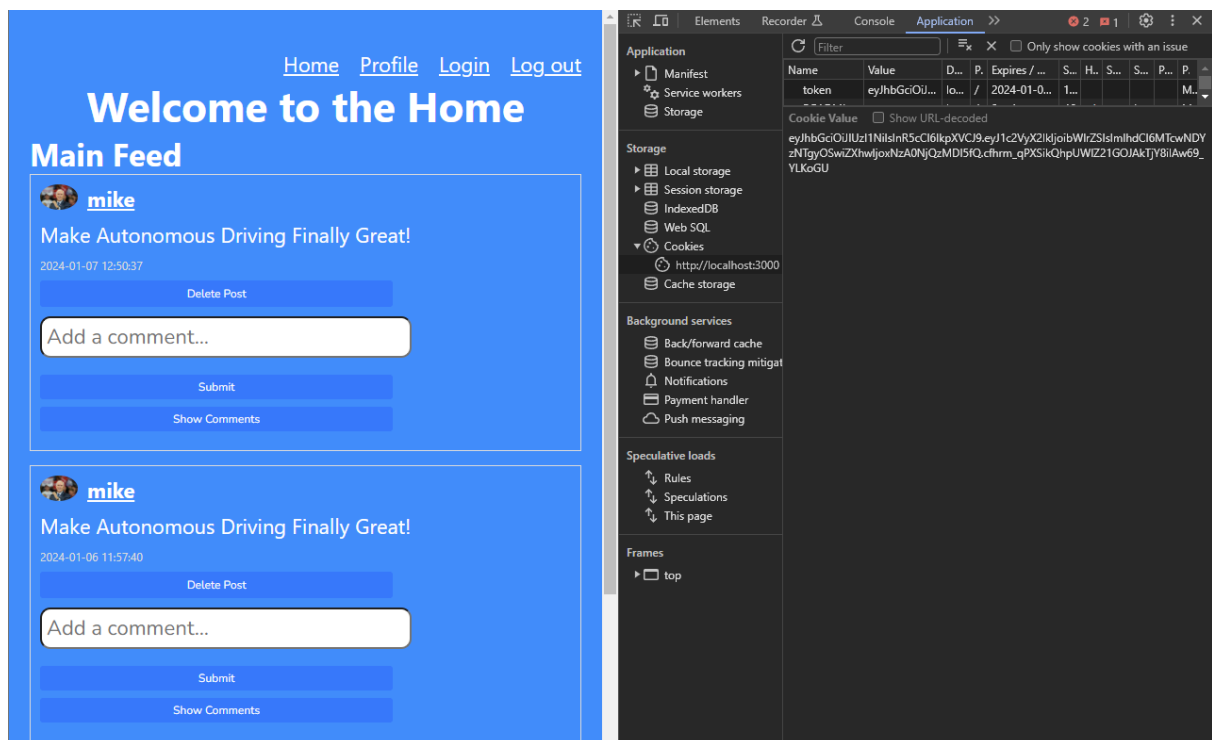


Figure 21. Main feed UI for the User - there is an active cookie token (project implementation with developer tools opened on application/cookies, localhost:3000/ - source: author's work)

5.2.4. Deleting content

Logged in users can delete both individual posts and individual comments. To explain the algorithm behind this let us look at the example of deleting a post.

Delete request is sent to the `/api/posts/:id` including two body parameters - token (browser cookie token) and `post_id`. Verification of the browser's cookie token is handled inside the `deleteById` function in `postController.ts`. Based on the provided request parameters, account data and post data is fetched from the database. By comparing `post.username` with `account.username` it can be determined whether the holder of that token is also the author of the post. Only then a deletion query is executed on the database, resulting in deleting given content.

In contrast to deleting a single comment, deleting a single post results in deleting all the comments associated with it.

5.3. Database choice

In the process of choosing a database, many solutions available on the market were tried. Platform started as a mobile application for Android (at the beginning it was called RemindsMe but due to too strong association with the reminder app it was later renamed). For this reason, the popular database choice was Google's Firebase. However, as the author parted from the idea for a mobile application, Firebase as a database was also abandoned. Despite that, in the future it is not impossible that the platform will serve as a mobile application where Firebase will serve as the provider for the user authentication service.

Fed up with the abandonment of Google, the author dedicated himself to finding the best possible solution which at that time was for me a P2P, decentralized database. After thorough research GUN was chosen - a decentralized graph database. The choice was not only considered as a dedication to the technology but most importantly as a dedication to the vision of its creator Mark Nadal (more on that in the last chapter dedicated to an *overview of the possible enhancements to the platform*). Sadly, However, the struggle of incorporating his solution into the platform and because of the time constraints given to making a presentable product the technology was once again not implemented into the project.

The willingness of using a graph database was still valid (as it is done by Twitter) so the next choice was to try Neo4j. This was however a very short journey as the author realized, it was not so much the graph database as a technology but rather as the idea of decentralized peer-to-peer network. The author once again failed to deliver on that idea and returned to the technology known the best among those mentioned so far.

For storing file description in a convenient way, the author relied on the local Postgres database. In such a relational database, we can easily bind a content (post or comment) to the corresponding account.

The advantage of postgres in the described project relies heavily on the usage of Docker as a containerization tool. This way the entire infrastructure can be easily mounted with a single script and the database schema can be recreated (with a helper script) regardless of the host environment.

Development process with Docker can be somewhat automated by writing scripts that could set up the development environment with a minimum amount of reconfiguration. For the reason of efficiency, reusability and maintainability it was wise to create such a shell script as shown on Figure 22.

The script present on the Figure 22 executes subscripts to set up docker environments for the database, backend and frontend as shown on Figures 23, 24, 25, 26, 27.

```
○ ○ ○

#!/bin/bash

#####
# Stage 1 logic
#####

echo "Executing Stage 1: Setting up environment..."

#####
# BACKEND
# compose database container
echo "Starting Stage 1.1-database: Composing database containers..."
docker-compose -f backend/docker/docker-compose-db.yaml up --build -d

# Check the exit code of the last command
if [ $? -eq 0 ]; then
    echo "Stage 1.1-database: Docker-compose command succeeded."
else
    echo "Stage 1.1-database: Docker-compose command failed."
    echo "Stopping the script..."
    exit 1
fi

# compose dev backend container
echo "Starting Stage 1.1-backend: Composing backend dev container..."
docker-compose -f backend/docker/docker-compose-dev.yaml up --build -d

# Check the exit code of the last command
if [ $? -eq 0 ]; then
    echo "Stage 1.1-backend: Docker-compose command succeeded."
else
    echo "Stage 1.1-backend: Docker-compose command failed."
    echo "Stopping the script..."
    exit 1
fi

# BACKEND
#####

#####
# FRONTEND
# compose dev container
echo "Starting Stage 1.1-frontend: Composing frontend dev container..."
docker-compose -f frontend/docker/docker-compose-dev.yaml up --build -d

# Check the exit code of the last command
if [ $? -eq 0 ]; then
    echo "Stage 1.1-frontend: Docker-compose command succeeded."
else
    echo "Stage 1.1-frontend: Docker-compose command failed."
    echo "Stopping the script..."
    exit 1
fi

# FRONTEND
#####

echo "Finished all sub-stages from Stage 1. Waiting 10s for environments to startup..."

#####
# Stage 1 logic completed
# wait for a few seconds to ensure database is up and running
sleep 10
#####

#####
# Stage 2 logic
#####

echo "Executing Stage 2: Setting up database schema..."

#####
# DATABASE SCHEMA
# execute the database schema creation script
echo "Starting Stage 2.1-postgres-schema: Setting up postgres schema..."
docker exec -i postgres16 psql -U postgres -d postgres < backend/database_schema.sql

# Check the exit code of the last command
if [ $? -eq 0 ]; then
    echo "Stage 2.1-postgres: Docker exec command succeeded."
else
    echo "Stage 2.1-postgres: Docker exec command failed."
    echo "Stopping the script..."
    exit 1
fi

echo "Finished all sub-stages from Stage 2"

#####
echo "All stages executed successfully"
```

Figure 22. Shell script that manages setting up docker development environment (project implementation, source: author's work)

```

version: "3.8"
services:
  postgres:
    image: postgres:16.0
    container_name: postgres16
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: postgres
    ports:
      - "5432:5432"
    volumes:
      - postgres-data:/var/lib/postgresql/data

  pgadmin:
    image: dpage/pgadmin4
    container_name: pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@admin.com
      PGADMIN_DEFAULT_PASSWORD: admin
      PGADMIN_LISTEN_PORT: 5050
    ports:
      - "5050:5050"
    depends_on:
      - postgres

volumes:
  postgres-data:

```

Figure 23. Docker compose script for the database and database admin panel (project implementation, source: author's work)

```

version: '3.8'
services:
  sharc_back_dev:
    container_name: sharc_back_dev
    build:
      context: ../
      dockerfile: docker/Dockerfile.dev
      target: development
    image: sharc_back_dev_image
    volumes:
      - ../usr/src/app/backend
      - /usr/src/app/node_modules
      - ../../assets:usr/src/app/assets
    ports:
      - 3001:3001
    command: npm run dev

```

Figure 24. Docker compose script for the backend (project implementation, source: author's work)

```

FROM node:18.18.2-alpine as development

WORKDIR /usr/src/app/backend

COPY package*.json .

RUN npm install

COPY . .

RUN npm run build

```

Figure 25. Dockerfile backend (project implementation, source: author's work)

```

version: '3.8'
services:
  sharc_react_dev:
    container_name: sharc_react_dev
    build:
      context: ../
      dockerfile: docker/Dockerfile.dev
    image: sharc_react_dev_image
    ports:
      - "3000:3000"
    volumes:
      - ../src:/usr/src/app/frontend/src:ro
      - ../../assets:/usr/src/app/assets
    env_file:
      - .env

```

Figure 26. Docker compose script for the frontend(project implementation, source: author's work)

```

FROM node:18.18.2-alpine

WORKDIR /usr/src/app/frontend

COPY package*.json .

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]

```

Figure 27. Docker compose script for the frontend (project implementation, source: author's work)

6. The future of the platform

6.1. A detailed look at the rising technologies

6.1.1. Web vs mobile

ShaReCon started as a mobile application, however lack of prior experience in this area of development led to inefficiency of work within that field. Moreover, neither Flutter nor any other cross-platform tool increased the efficiency and led to successful development. Hence, the choice was made that the optimal solution will be a web application made within the Javascript ecosystem - the one path that the author richly experienced and feels comfortable working in.

The author was, once again, driven by the broad accessibility of the majority of people having internet on their mobile and home devices. Moreover, it is certain that by using React to develop the platform the platform will be able to expand its reach by extending to a mobile version with React Native later on. That being said, by no means is the author ruling out the possible necessity of learning native mobile language for iOS and/or Android after completing ShaReCon MVP.

6.1.2. Decentralized vs centralized

Developers should be extremely infatuated with the idea of decentralization. The vision for ShaReCon is to be a platform providing architecture for sharing media freely, openly articulating one's and connecting people. The author does not wish to collect personal data nor distribute such data to third parties. It is believed that for user convenience, responsibility of authentication should be distributed across the user and the platform. Every user should be responsible for his account, however to prevent unfortunate accidents of losing one's password, the login key could be divided into three parts:

- one to be stored by the user, used for every login and able to be regenerated in the event of being lost
- one to be stored in the platform database, able to be reminded if the user forgot it and be semi-composition of the complete login key
- one to be stored only by the user, to be kept in the safe place with little-to-no possibility of being lost and accessible only offline

These three parts would compose safe, convenient and user-exclusive authentication. One of the technologies that caught my attention was GUN.js. As described by its creator and the lead maintainer Mark Nadal "*GUN is an Open Source Firebase for JAMstack apps*".[7] By definition JavaScript, API and HTML (JAM) stack generates static web code in advance. Before the content is ever requested the code is distributed to a Content Delivery Network (CDN). Users fetch content from the physically closest server in the shortest time possible. Dynamic content is available only through API calls from the application.

What is special about GUN is that the data and infrastructure are not centralized by one entity, instead they are decentralized across the entire user base. At the high level, GUN stores a small subset of data on each user based on the actual data that they consume in the application. When a user makes a query for some data it will search across the network for other users having that data and synchronize it using technologies like WebRTC. Although GUN relies heavily on cryptography it is not a blockchain technology.

“By default, data is stored in the browser’s local storage which is limited to 5MB. In the event of a user clearing their browser cache, all of the local data will be lost if it is not persisted somewhere else on the network. To prevent this, a relay server using a radisk storage mechanism is deployed”[6]. This allows to store a lot more data on the server disk and makes the network more robust as queries may fall back to a relay server if the data is not available from another peer. The main advantages of using GUN is that it:

- provides a real time p2p state synchronization
- is a graph database meaning it provides key-value storage
- is local-first, offline and decentralized with end-to-end encryption
- creates low-latency illusion

6.1.3. P2P sharding vs compression

For a fast and reliable data sharing system one of two approaches should be taken. One is already being used throughout the internet - compression. When talking about file sharing only lossless compression can be taken seriously, because unlike lossy image compression, files like desktop applications or games cannot be missing any information - otherwise they may not be usable. *“It is sensible to be saving bandwidth by transferring a smaller (compressed) file across the network at the cost of CPU power (of decompressing such file) which is generally much cheaper than network bandwidth”*[8]. Moreover, considering that oftentimes the same file will be transferred multiple times it is reasonable to store a smaller version of it on a server and then make users decompress it on their devices.

P2P sharding however, takes advantage of the whole network by storing small chunks of the file on each node. By doing so, no compression is necessary (it is optional but not required) as the file is scattered all over the network making it easily accessible by its owner at the same time stored securely as each piece is safely encrypted and the file cannot be recomposed without being decrypted first.

The system that is planned to be implemented into ShaReCon is IPFS. It was developed in 2015 by Juan Benet, written in Go programming language and has been kept as an open-source project ever since. *“The main difference between IPFS and the current web file system is that the former operates in a content based addressing. This means that to find a resource, instead of telling our browser where to find it, we specify what we are looking for. Not only does it make the content-search reliable as it does not depend on its source website existence, it also supports versioning of the files just like Git does. This prevents files from being mutated and keeps the history of the changes so that you can trace back to any version you wish to access.”*[9]

6.1.4. UX vs UI

Saying has it that *“people consume with their eyes”*. It is rarely the case that a poor performing website will keep newcomers just with a beautiful look. Functionality and performance is what makes a user return to the platform. Looking at the 4chan example, no advanced UI is needed to make a website successful. It is recommended to take care of both - the functionality and the design. Even though most users either do not understand or care about the implementation of the application it is in great practice to take care of well-structured code. It is often the case that a badly implemented solution affects the usability of the project and so the design is affected as well. Out of many libraries attempting to make CSS easier to deal with, Tailwind seems to be the choice among React developers. It seems to be the cleanest and the least demanding way to style components in JavaScript. With its easy to learn acronyms-based syntax and in-line style it gives clear predictions of what will be displayed.

6.2. Observation vs participation

ShaReCon will not require users to create an account in order to be an observant in a community. Anyone can go to the ShaReCon website and freely browse through the content. The minimum viable observation requires paying attention to what's happening. To participate however, an observer (viewer) is required to create an account. There are two main reasons leading to such decision:

1. ease of implementation - managing content created by observers requires different approaches: How to responsibly log anonymous users in a database? How to differentiate anonymous authors?
2. user participation - it is believed that for greater user engagement, observers should register themselves. It creates a sense of involvement in the community. Users are allowed to create anonymous accounts - unlinked to their true identity. It ultimately creates a sense of freedom to speak one's mind freely and disconnect from real life expectations.

7. Summary

As described in this paper, open-source culture is seen as the major path in development of software. Both the accessibility and collaboration lay in the core of its practice. Social media platforms have faced a rapid change throughout the years and will continue to face challenges as more users are becoming mindful and responsible for their profiles.

The same change affects developers as they need to adapt to the ever growing need of accessibility and reusability. It is no longer sufficient to make a monolithic app that runs well. It should be in great interest to provide a multi-environment experience. The development process shall not be restricted to a single workstation. This approach extends the life of an app and provides the blueprint for diverse development.

The presented platform is seen as both the practical implementation of the core software engineering and open-source practices and lays a solid foundation for future enhancements to the social media connectivity. It is in great interest of the author to further explore the essence of interaction. As the world keeps on accelerating, it is a duty of responsible developers to provide meaningful and efficient connectivity opportunities. Connectivity seems to be the underlying backbone of today's modern societies and with that in mind, the author will further explore what it means to connect entities together and what it means to interact.

Bibliography

- [1]<https://stackoverflow.com/questions/>
- [2]<https://odysee.com/>
- [3] <https://www.npmjs.com/package/multer>
- [4]<https://twitter.com/SamParkerSenate/status/1748951632405201103>
- [5]<https://github.com/twitter/communitynotes>
- [6]<https://thenewstack.io/brendan-eich-on-creating-javascript-in-10-days-and-what-hed-do-differently-today/>
- [7]The Weird History of JavaScript - <https://youtu.be/Sh6lK57Cuk4>
- [8]<https://github.com/imasharc/ShaReCon>
- [9]<https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-webframe-love-dread>
- [10][https://2021.stateofjs.com/en-US/libraries/front-end-frameworks/#front end frameworks experience ranking](https://2021.stateofjs.com/en-US/libraries/front-end-frameworks/#front_end_frameworks_experience_ranking)
- [11]<https://gun.eco/docs/Radisk>
- [12]<https://github.com/amark>
- [13]How File Compression Works - <https://youtu.be/HNPqWaSEMJI>
- [14]IPFS in 2 minutes - <https://youtu.be/k1EQC7tdh70>

Figures listing

- Figure 1. Stack Overflow (source: <https://stackoverflow.com/questions/>)
- Figure 2. Odysee (source: <https://odysee.com/>)
- Figure 3. Community Note written under a post
(source: <https://twitter.com/SamParkerSenate/status/1748951632405201103>)
- Figure 4. GitHub repository for Community Notes (source: <https://github.com/twitter/communitynotes>)
- Figure 5. Platform architecture (visualized with LucidChar - source: author's work)
- Figure 6a. Directory structure (presented with VS Code - source: author's work)
- Figure 6b. Directory structure (visualized with LucidChart - source: author's work)
- Figure 7a. Functionalities use case diagram (visualized with LucidChart - source: author's work)
- Figure 7b. Functionalities use case diagram implemented by author (visualized with LucidChart - source: author's work)
- Figure 8. Sequence diagram for registering new user (visualized with LucidChart - source: author's work)
- Figure 9. Sequence diagram for registering new user (visualized with LucidChart - source: author's work)
- Figure 10. Project repository on github (source: author's work published on <https://github.com/imasharc/ShReCon>)
- Figure 11a. Log in form (project implementation, localhost:3000/login - source: author's work)
- Figure 11b. Sign up form (project implementation, localhost:3000/signup - source: author's work)
- Figure 12. Checking availability of the username (project implementation, source code - source: author's work)
- Figure 13. Regex patterns for sign up form (project implementation, source code - source: author's work)
- Figure 14a. Invalid sign up form (project implementation with developer tools opened on application/cookies, localhost:3000/login - source: author's work)
- Figure 14b. Handling authentication on the backend (image generated with <https://carbon.now.sh/>; project implementation of the authentication mechanism, localhost:3001/login - source: author's work)
- Figure 15a. Specifying API routes for authentication endpoints (project implementation of the server side - source: author's work, screenshot of the source code)
- Figure 15b. Specifying API routes for account functionalities endpoints (project implementation of the server side - source: author's work, screenshot of the source code)
- Figure 15c. Specifying API routes for comment functionalities endpoints (project implementation of the server side - source: author's work, screenshot of the source code)
- Figure 15d. Specifying API routes for post functionalities endpoints (project implementation of the server side - source: author's work, screenshot of the source code)
- Figure 15e. Server configuration on the backend (image generated with <https://carbon.now.sh/>; project implementation of the server side - source: author's work)
- Figure 16a. Updated token in the database (project implementation pgadmin panel, localhost:5050 - source: author's work)
- Figure 16b. Cookie token present in the browser after successful sign in (project implementation with developer tools opened on application/cookies, localhost:3000/login - source: author's work)
- Figure 16c. 1/2 part of the accountController functionality (image generated with <https://carbon.now.sh/>; project implementation - source: author's work)
- Figure 16d. 2/2 part of the accountController functionality (image generated with <https://carbon.now.sh/>; project implementation - source: author's work)
- Figure 17. Source code for updating access token on the backend (project implementation, source code of account.ts - source: author's work)
- Figure 18. Source code for handling authentication on the backend (project implementation, source code of authController.ts - source: author's work)
- Figure 19. Source code for submitting log in form on the frontend (project implementation, source code of LoginForm.jsx - source: author's work)
- Figure 20. Main feed UI for the Viewer - there is no active cookie token (project implementation with developer tools opened on application/cookies, localhost:3000/ - source: author's work)
- Figure 21. Main feed UI for the User - there is an active cookie token (project implementation with developer tools opened on application/cookies, localhost:3000/ - source: author's work)

Figure 22. Shell script that manages setting up docker development environment (project implementation, source: author's work)

Figure 23. Docker compose script for the database and database admin panel (project implementation, source: author's work)

Figure 24. Docker compose script for the backend (project implementation, source: author's work)

Figure 25. Dockerfile backend (project implementation, source: author's work)

Figure 26. Docker compose script for the frontend(project implementation, source: author's work)

Figure 27. Docker compose script for the frontend(project implementation, source: author's work)