UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Computer Science Report 2

'Kasie

Group 8:

Phillip Schommarz (22621105)

Nicol Visser (16986431)

Lizé Steyn (21772037)

JC Mouton (22549889)

May 19, 2022

Contents

List of Figures

Nomenclature

Acronyms and abbreviations

CMS	Content Management System
MVC	Model–View–Controller (design pattern)
URL	Uniform Resource Locator
GUI	Graphical User Interface
NF	Normal Form
BCNF	Boyce Codd Normal Form
REST	Representational State Transfer
SQL	Structured Query Language
LINQ	Language-Integrated Queries
IDE	Integrated Development Environment
HTTP	Hypertext Transfer Protocol
JWT	Jason Web Token
SMTP	Simple Mail Transfer Protocol

Chapter 1

Introduction

Social media has become a staple of our everyday lives. Be it Twitter, Facebook, Instagram or Snapchat; these websites are graced by our presence on the daily. However, one fundamental flaw irks them all: the disconnect from the world around us. The user is overwhelmed by sheer amounts of content from all over the world, events and posts from places they've only dreamed of ever seeing, and while at a first glance this seems like a desired feature, it leads to us losing touch with the beauty of life that surrounds us.

This is where **'Kasie** comes into play. Similar to the aformentioned social networking sites, 'Kasie also delivers content from all around the world. With one small twist. With an extensive map tool feature, the user takes the driving seat when it comes to what they want to see. By allowing you to filter content in a radius surrounding you, 'Kasie aims to reestablish the connection between social media and real life once again. If you're in a rush, 'Kasie's "sort by nearest" feature achieves a similar outcome. Posts nearest to your location will rise to the top of your feed, allowing you to quickly and easily connect with the friends near you.

To further give back the connection to reality, the posts a user sees are only those which he wants to see. By joining groups of like-minded people and adding friends, a user's feed is kept clutter free and provides only content which the user approved. No more irrelevant information overload from things you definitely did not have to see. Finally, with a direct messaging feature the user can directly chat to their friends.

Chapter 2

Overall Description

This website serves as social networking a platform for users to connect and interact with each other, with a focus on location based posts. Such a system, that allows users to easily share various types of geo-tagged posts, create groups, add friends and direct message other users, requires the development of many different design components.

Firstly the needs and access levels of the users are considered to determine possible use cases in the system. In order to implement these use cases the provided data must be efficiently modelled and stored in a database. The backend is developed to provide the ability to edit the data and access the desired information. A user interface or frontend is developed to grant users the ability to interact with the data.

This chapter describes each of these components and the different technologies and design approaches associated with them.

2.1. Use Case Diagram

The needs of users are assessed to form a list of different use cases. These use cases, as illustrated in Figure 2.1, describe how users interact with the system. In this system there is one main user type or actor, although the user capabilities will differ based on their relationships with other users and their status within groups.

Users are first registered through a sign up process and thereafter require a login each time they wish to interact with the system. Users will be able to search for and view their own profiles, group profiles and profiles of other users. These views will differ in order to maintain a level of user privacy and to provide editing capabilities on the users own profile.

Users may create text, photo or video posts with either automatic or specified geo-tags. Users will be able to view these posts in both a feed view and a map view. Each view allows users to comment on posts and comes with its own set of filtering and sorting capabilities, such as filtering posts that were made within a certain radius.

Users are able to create groups and befriend other users. If users own groups they may assign other group admins that are able to edit the group, although only the group owner will be able to delete the group. Friends are also able to communicate using direct messaging.

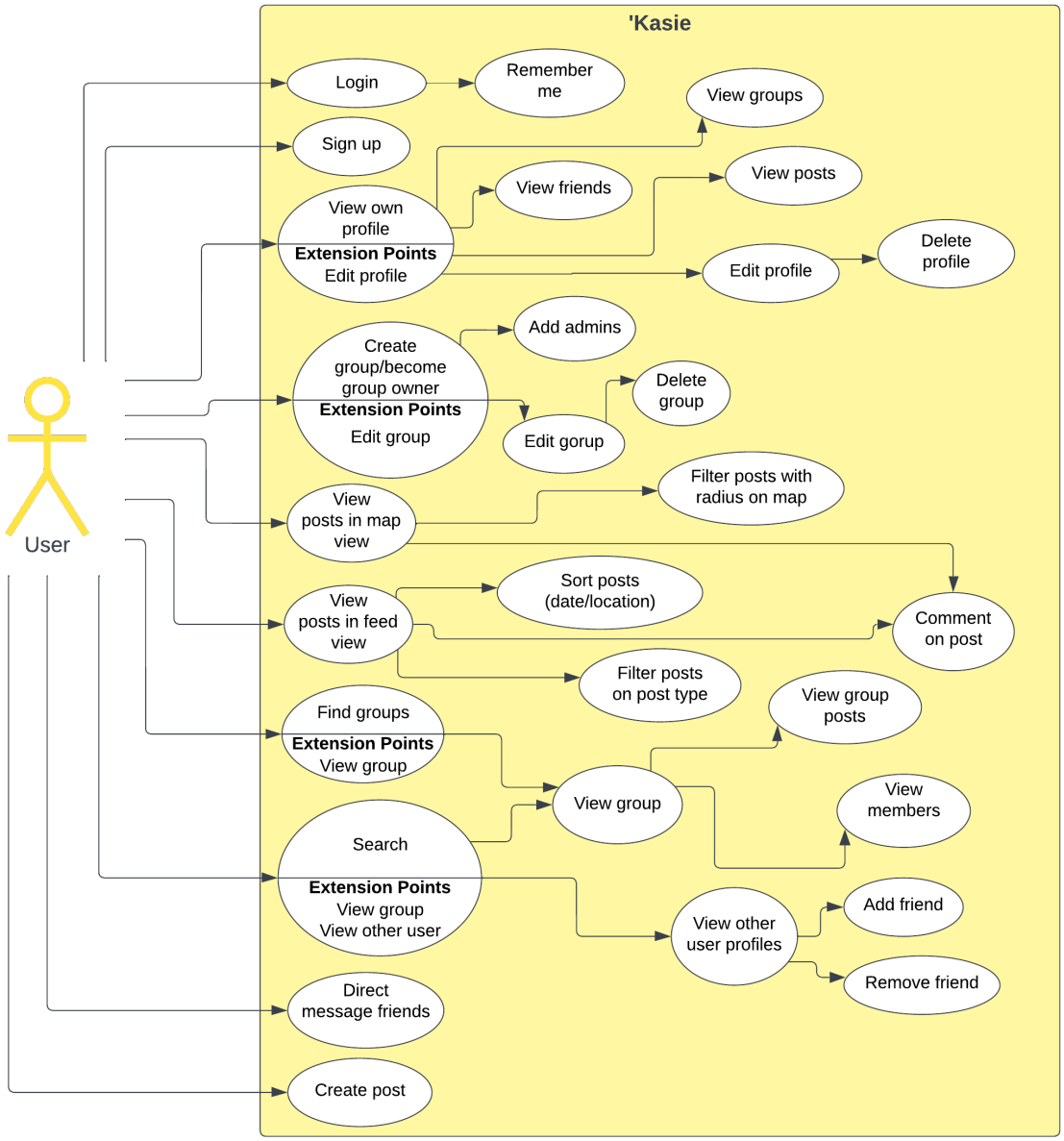

Figure 2.1: Use case diagram

Each use case will either collect, manipulate or return certain data to the user. This detail is not captured in the high level representation of a use case diagram, but the next section will provide more insight into the types of data associated with each use case.

2.2. Data Modelling

In order to ensure data integrity and to prevent data duplication, the data model has to adhere to 3NF (Third Normal Form) and BCNF (Boyce Codd Normal Form). Seeing as BCNF is an extension of 3NF, meeting BCNF ensures 3NF has been met as well. The primary requirement of BCNF is that for every functional dependency $X \rightarrow Y$, X needs to be the super key of the table [1]. When taking a closer look at the Entity Relationship Diagram in figure

2.2, we notice that by using this model the BCNF requirement (and thus 3NF requirement) is met.

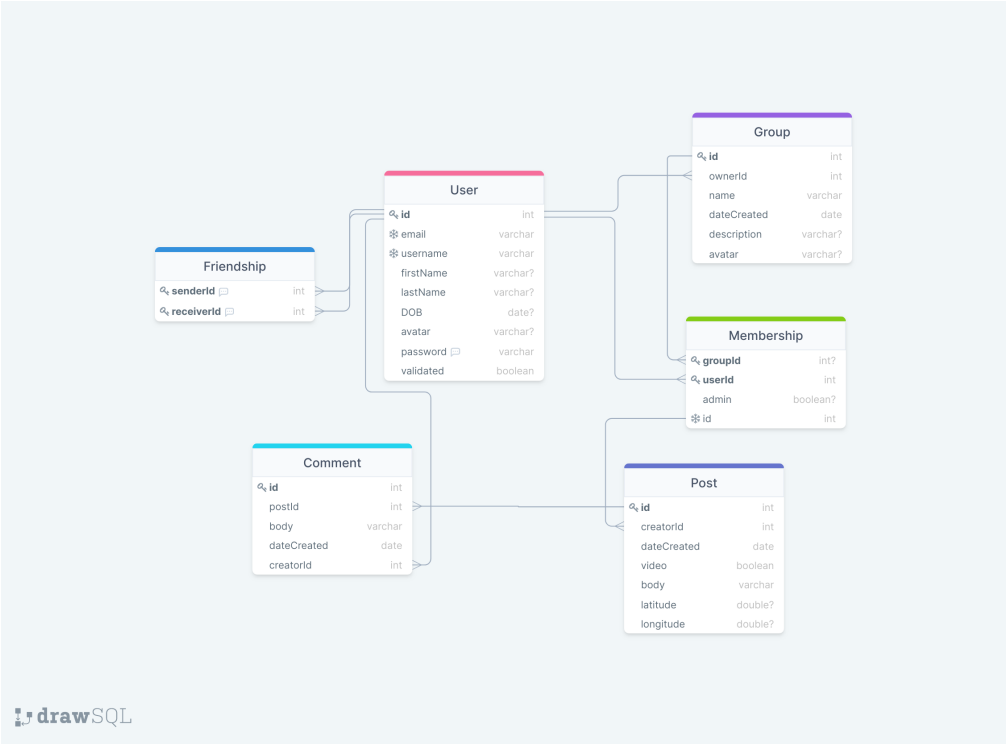

Figure 2.2: Entity Relationship Diagram

2.3. Operating Environment

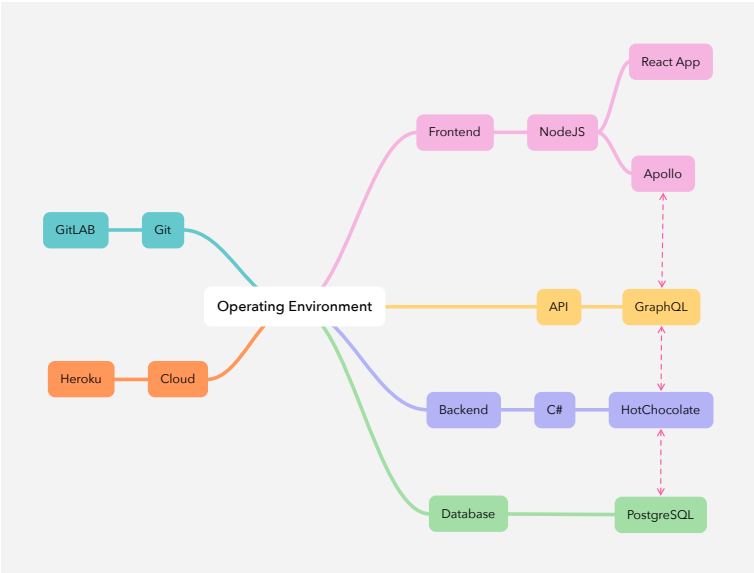

Figure 2.3: Operating Environment of our Web App

There are a wide variety of technologies available to build a web application. Figure 2.3 shows the operating environment of our webs application with some of the core technologies that we decided to incorporate.

From the start of the project we hosted our workspace in the cloud. This was such that all developers can work on the same database and such that the frontend developers have access to the APIs without having to run the backend server locally. We decided to go with Heroku to host our backend server. Heroku has a generous free tier for hosting small scale servers. By adding a React build pack, Heroku automatically builds the frontend files and routes the user to them when the site is accessed. 'Kasie can be accessed by going to cs334proj2group8.herokuapp.com. Please note that since we are using the free tier of Heroku, the server will go into a sleep mode and the first time loading the site might take a while.
JavaScript is commonly used to build the frontend of websites instead of plain HTML and CSS. When using JavaScript the user is able to interact with the website without having to load a completely new HTML page on every interaction. React is a JavaScript library for developing powerful and responsive web apps. React's state management system allows developers to efficiently split up the web application in reusable components. If developers follow React design principles, it will ensure that only relevant parts of the web app get reloaded on each interaction from the user.

We decided on using React instead of other frameworks like Angular due to its popularity, widespread availability of information and due to our team's familiarity with React after the first project. In Section 3.1 we will further discuss our web app's design from a React perspective. In order for the frontend to communicate with the backend, we have implemented a GraphQL API. The React App on the frontend makes use of the Apollo library to make API calls to the GraphQL server.

C# is arguably one of the best high level general-purpose programming languages though often gets hate for being Microsoft's star child. C# is frequently improved upon and has conquered its compatibly issues with the release of the .NET core framework that will be at the center of our backend. C# was chosen as the developing language for the backend as it has similar syntax to C, has robust and reliable web development frameworks and would be a step up from previously used Python.

PostgreSQL is free, reliable, and open-source database management system. Due to having previous experience with PostgreSQL along with the aforementioned qualities it was picked over MSSQL (and other options). To be able to view and manage our database (hosted on Heroku) in a non-terminal environment, pgaAdmin was used to provide a GUI.

To allow the frontend to view and interact with the API Postman was used during development, acting as a graphQL playground and documentation. The full API workspace can be viewed on https://www.postman.com/cs334project1group8/workspace/cs334pro2.

... Java feels like C# from the late 2000's.

Reddit intellectual [2]

Chapter 3

Solution

3.1. Frontend Solution

On a high level, we tried to follow two main design methodologies while developing the frontend. Most importantly we wanted modular, reusable and maintainable code. Furthermore we wanted our code to reflect a Model-View-Controller (MVC) design pattern where needed. We found that splitting React code up into reusable components and by also using React context providers and consumers to be very helpful to keep our code maintainable. See section 3.1.2 for more information on how we used context providers.

3.1.1. Libraries

React

At the heart of the frontend solution lies the React Library. Due to it's maturity and industry-wide use, this framework was chosen. Its component-based design allowed us modularity and re-usability in the development stage. Our React app has a structure as shown in Figure 3.1.

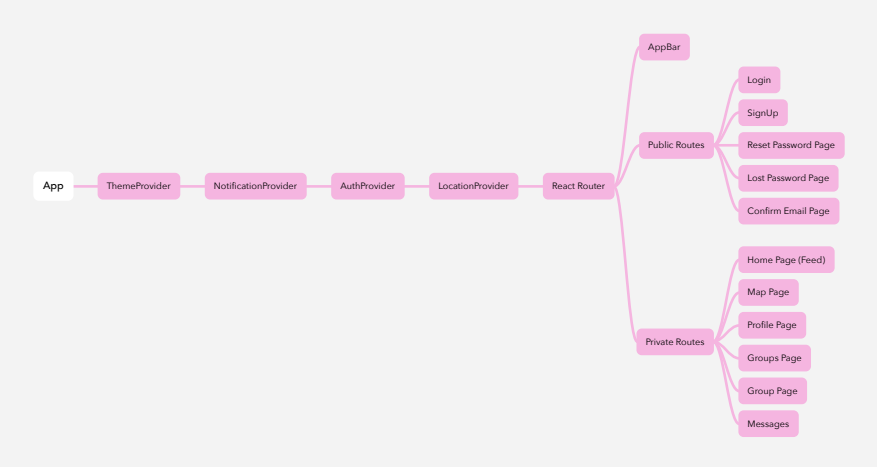

Figure 3.1: Simplified React App Structure

React Router

The React Router library provides a way to perform nested routing. By using React Router properly, we could keep some components such as the AppBar and the NotificationProvider active, while the user appears to be changing pages. This makes for a far more enjoyable and

smooth user experience.

If we look again at Figure 3.1, on the lowest level of the App structure, we have components that correspond to the different pages the user is visiting. The React Router library allows us to show the correct component based on the route (URL) the user is visiting. React Router was also leveraged to distinguish public routes from private routes. If the user is not logged in (i.e. the PrivateRoute component registers that the *authUser* retrieved from the AuthProvider is null), then the user immediately gets redirected to the sign in page.

Material UI

The React Material UI (MUI) library provides numerous, high quality user-interface components. We used the MUI library as an extensive crutch in our development as it saved countless hours of work which could have been lost to basic CSS coding. In addition, due to MUI's flexibility and easy customisability, we were able to create a beautiful interface for our client side. Furthermore, one additional component based library was used called Swiper. This swiper component was used on the map page of the website for a more enjoyable user experience.

Apollo

As discussed in section 3.2, GraphQL was used as the query language for our API. To thus make the querying process from the frontend much easier, the Apollo library was used, which was specifically designed around GraphQL. Apollo enabled the frontend to quickly and easily write GraphQL queries and to then handle the successive responses in a clean and simplistic manner.

React Leaflet

The React Leaflet library was used to provide all map features across the website. This includes the map page, which shows the user's subscribed posts and allows them to be filtered by radius, as well as the location selector when creating a new post. Leaflet was chosen due to its wide variety of features, such as changing the tile layer provider (allowing us to use a beautiful dark themed map); placing pins and interacting with them; as well as generating circles (used to visually indicate the post search radius).

i18Next (Bonus Marks)

In order to provide multiple language translations for the website, the i18Next translation library was used. i18Next provides advanced features such as interpolation and provides the option to use JSON data to provide the separate translations. This latter feature allowed us to write short python scripts to make the translation process into various languages much faster. Translations were generated using DeepL.

Gun (Bonus Marks)

Seeing as direct messaging was a bonus mark feature, and that the backend of our project was quite mature and complicated, when work on direct messaging was started a somewhat disconnected solution was required. This solution came in the form of the Gun library. By providing a decentralized graph database, as well as real-time connection capabilities, this library brought exactly what was needed to the table.

An extremely light-weight gun server (hosted on Heroku) establishes peer-to-peer connections between users, and then, once a listener has been established, pushes all non-localised graph data to these peers. In order to provide an extra layer of data stability, the server was set up to store the graph database as well. This ensures that at least one peer (the server itself) will always be able to serve the database to the peers which establish connections to it. To help us establish this functionality, the Gun documentation, as well as a video by the YouTube channel Fireship [3] was referenced.

Unfortunately, due to time constraints and this being a feature only needed for the bonus mark section, the various direct messaging nodes are not encrypted or authentication restricted; meaning that theoretically all users could read direct messages of other users.

3.1.2. React Context Providers, Consumers and Hooks

The Model-View-Controller design pattern is a popular choice in developing User Interfaces. It separates concerns between the underlying application and the user interface (UI) components. In our web application, the ”model” part refers to the underlying application from the backend which can be accessed via the API. The ”view” part refers to the code related to the UI component. This should ideally be purely for display purposes. The ”controller” part refers to the code that connects the view to the model. To separate concerns in React, we implemented custom context providers with corresponding context consumers via hooks. The below code shows some examples of how we used the context provider and hooks in our code.

```
// Home.js (feed view)
...
return (
        ...
    <PostProvider page="feed">  // Controller Component
        <PostSorter />          // UI Component
        <AddPostCard />         // UI Component
        <PostList />            // UI Component
    </PostProvider>
    ...
)
```

```
// PostList.js
...
const posts = usePosts()
...
return (
    ...
    posts.map((post) =>
        <PostCard postId={post.id}>
    )
    ...
)
```

```
// AddPostCard.js
...
const addPost = useAddPost()
const refreshPosts = useRefreshPosts()
...
```

```
// PostSorter.js
...
const filterPostsBy = useFilterPosts();
const sortPostsBy = useSortPosts();
...
```

The PostProvider provides context for the post data and functions that act on this data. Descendants of PostProvider can use a hook to get access to the data or functions. PostList uses the usePosts() hook to be able to render a PostCard for each post item in the data. AddPostCard uses the useAddPost and useRefreshPosts hooks to add a new post and then refresh the data (to include the newly added post). PostSorter calls functions to sort or filter the posts in different ways. This way of coding separates the concerns between the UI and the controller. All code that queries and mutates the posts can be found in the PostProvider component. This keeps the UI components nice and clean.

It is difficult to always separate the concerns between the view and controller. Sometimes for a simple UI component, which is also not used somewhere else in the app, the overhead of splitting it up into separate UI and controller parts outweighs the benefit of doing so. Especially with modern style hooks such as the useQuery hook from Apollo, it is sometimes very convenient to perform an API call directly in the UI component. This useQuery hook returns the variables, loading, error and data. You can then show a different component (such as a spinner) while the query is loading or if it results in an error.

We found 5 areas that benefited a lot from creating a dedicated Provider-and-consumer system. These are summarised in Table 3.1.

Table 3.1: A list of React Context Providers in our Application.

Provider Name	Function	Why a Provider was Used
AuthProvider	Provides authentication features (login, signup, logout). Delivers some basic info about user to components.	Used by many components. Complexity of code.
CommentProvider	Queries comments from model. Handles creation, deletion of comments in model.	Clean code and modularity.
LocationProvider	Provides current user location to components.	Used many times.
NotificationProvider	Provides a pop up with message for the user to read.	Used many times.
PostProvider	Provides different list of postIDs based on its props. Handles change in sorting and filtering method. Handles creating and deleting posts.	Used by many components in different configurations. Complexity

3.1.3. Hosting Services

We once again wanted to keep the server as data-light as possible, thus videos and images should not be stored directly in the database. We realise that a Content Management System (CMS) could be a very elegant solution to this requirement. Web applications that make use of a CMS can very easily be scaled for large amounts of traffic, but these services are often quite costly. We decided to go with a much simpler approach.

Images - ImgBB

There are several image hosting websites such as ImgBB and Imgur that allow anyone with an account to upload images via the service's API. We decided to go with ImgBB's API which is very simple to use with minimal authentication requirements.

ImgBB's API has a POST request at the endpoint https://api.imgbb.com/1/upload. You have to pass it two parameters, the API key associated with the account that will be storing the image and the image (encoded in base64 format) that you want to update. If the request is a success the response includes a URL to where the image is stored. We have a resuable *AvatarPicker* component, that allows the user to upload an image from their filesystem. This image is encoded to base64 and uploaded via the above API. On sign up, group creation or profile edit, we then only send the URL to the database.

Videos - Cloudinary

Cloudinary is a video hosting service with a generous free tier. They have embeddable widgets and players which deal with much of the low level aspects of uploading and playing videos.

To enable the use of Cloudinary's upload widget we included the recommended script in our index.html file. We can access and load this widget via the window object in ReactJS. Videos

are uploaded to our Cloudinary account. Each video is assign an 'public ID' which is stored in our backend database.

There are many ways of integrating a Cloudinary video player into a React application [4]. One of the simplest ways was to use an *iframe* which allows features like autoplay, fullscreen and picture-in-picture on many browsers.

3.2. Backend Solution

Instead of designing a RESTful API we chose to follow a GraphQL architecture. A GraphQL server has many advantages that we desired. These advantages include: fetching only requested data, fetching many resources in a single request, strictly-typed interfaces and ease of documentation. Although the initial setup is more complex than a REST API, the payoff is well worth it as it allows for queries to be written effortlessly with built-in searching, sorting and filtering. GraphQL generates Structured Query Language (SQL) queries based on the GraphQL request, limiting the need for writing complicated SQL OR Language-Integrated Queries (LINQ) on the backend. Only a single endpoint is needed thus the frontend does not need to worry about multiple routes that could change.

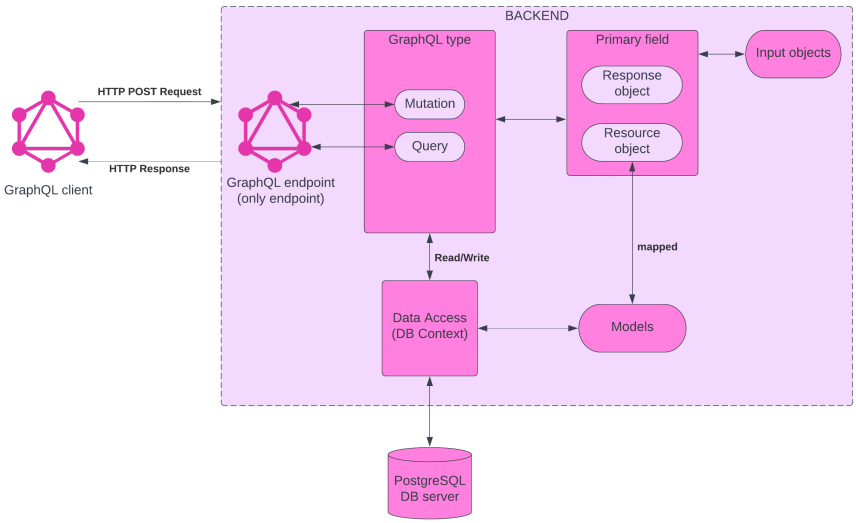

Figure 3.2: Backend architecture

GraphQL

A popular choice for developing GraphQL servers on C#'s .NET framework is Hot Chocolate. This library allowed us to fluently implement GraphQL into our backend as well as provide its own GraphQL integrated development environment (IDE) Banana Cake Pop that could be used instead of Postman. Both of the previously mentioned IDEs auto fetched the schemas and enforced strictly-typed interfaces (self-sanitizing), showing warnings and giving

errors if a request contains invalid fields or missing required fields. They also allowed for auto-completion of field names, allowing us to view the sub-fields within an object. Additionally Hot Chocolate provided a route that graphically represents the API's schemas via GraphQL Voyager allowing for self-documentation. These two features can both be viewed on our app through: /graphql, /graphql-voyager.

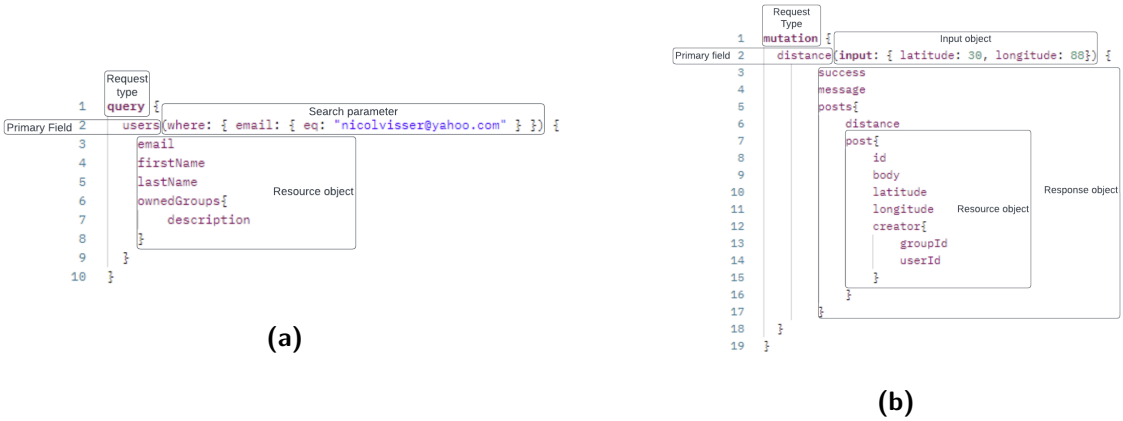

Figure 3.3: (a) Query example (b) Mutation example

For a GraphQL client to communicate with the single GraphQL endpoint, the POST HTTP method needs to be used for all requests. A request must contain a type. For our application only the two special types were used, query and mutation type. Query type can be seen as a "getter", while mutation type is used to modify server-side data. After the GraphQL type is selected, a primary field is specified along with desired sub-fields. If the type is a mutation, the primary field could represent a "function" which is able to take input objects. Query types can also be filtered, searched or sorted. If a request is handled without error the client should expect resource objects (if asked for) and possibly a response object that clarifies whether the request was a success in terms of writing and reading to and from the database. Resource objects are mapped with Object Relational Mapping (ORM) to the database models. Figures 3.3 and 3.2 may help clarify.

Object Relational Mapping (ORM)

Since C# is an object oriented language, it only made sense to treat the data in the database as objects as well. A thorough description was given by Mia Liang: "An object-relational mapper provides an object-oriented layer between relational databases and object-oriented programming languages without having to write SQL queries. It standardizes interfaces reducing boilerplate and speeding development time" [5]. The .NET framework has a great ORM library for PostgreSQL based on Microsoft's Entity Framework. Npgsql EF Provider allowed for a flawless implementation of an ORM in our backend.

Authentication and Authorization

To prevent unauthorized users from accessing the API it has to be restricted. Furthermore the API has to disallow a user from editing the data of another user. These requirements were achieved with token-based authentication specifically using JSON Web Tokens (JWT). This was chosen over session-based authentication as session IDs will need to be stored on the server or in a database, a problem not present with the token-based system.

Most of the API's queries and mutations requires general authorization. To authorize a user they must first be authenticated. This was done through an unrestricted login mutation that generates a JWT (using a symmetric encryption key saved on server side) on successful authentication. Claims can be attached to the JWT to allow users access to certain data. The JWT is then stored on the browser and can be added to the authorization header on future requests. The server then only needs to validate the JWT signature using the symmetric encryption key again. Depending on the claims contained in the JWT, a user may edit their own data but not that of others, as they do not have a claim on other users' IDs. Extracting the user's ID from the JWT claim also avoided more authorization checks in the case that the client was to send an ID that did not match that of the JWT claim. Additionally, to ensure that a token is not used by another user, tokens were set to have an expiration time of 30 minutes. However, if the user is still on the website after 30 minutes their token will be refreshed. Figure 3.4 shows the authentication and authorization process. Authorization was achieved with HotChocolate.AspNetCore.Authorization.

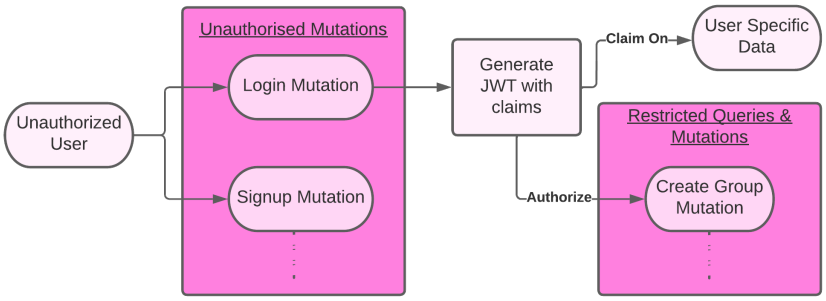

Figure 3.4: Authentication and authorization diagram

Password Hashing

Ensuring the safety of our user's data requires their passwords to be protected in some way. Traditional encryption is not safe enough these days and a more secure method is required: hashing. Hashing of a password refers to essentially transforming a string (password) into a random string of characters. The problem is that every string has its own unique hash code, therefore rainbow tables can be used to crack a password using previously known string-hash code pairs. To reduce this, one can add salt to the password string, changing the string and

hence the hash. The salt must also be stored which could be a problem as ideally every password should have its own salt. Hashing algorithms (such as the SHA family) have become very efficient and GPUs can crack them fairly easily, therefore, for password hashing, one wants a ”slower” algorithm. Although salting is recommended for simplicity it was left out. Now considering that we want a fairly slow algorithm we chose the BCrypt algorithm.

Spatial data and Distances

Due to Hot Chocolate not officially supporting spatial data (point type) we were unable to make use of PostgreSQL's spatial database extender PostGIS and its benefits when it comes to sorting by distances and filtering by radii. Hence a singular column for storing point location data could not be used thus two separate columns (latitude and longitude) would represent a geographic coordinate. Posts would have coordinates saved and from this, the spatial basis of the spatio-temporal social network would be derived.

To calculate the distance between posts (i.e. 2 coordinate points) a mathematical formula would be needed. The Haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes [6]. To avoid reinventing the wheel a library was used to implement the Haversine formula. The Geolocation package was utilized. There were many other options but this one proved to be sufficient and accurate.

Emails

Validating a users email is important to ensure that future communication can take place between the service provider and the consumer. Allowing users to reset their passwords is a vital attribute of websites in today's world as many passwords must be remembered.

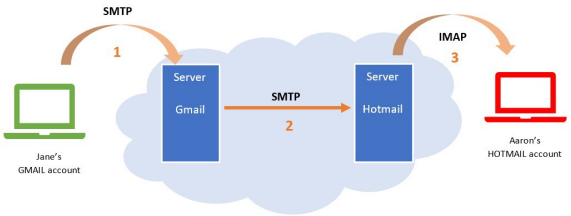

Figure 3.5: Email protocols diagram

Enabling our application to send emails was therefore important. A simply way to achieve this is by using Google's Simple Mail Transfer Protocol (SMTP) server and a Gmail account. In which our API would act as an SMTP client that is capable of sending emails to the SMTP server. The email is then processed by the SMTP server and passed to a user dependent Internet Message Access Protocol server and into their mailbox. To visualize this process refer to Figure 3.5 above.The FluentEmail library was used to simplify the sending of emails via SMTP.

Bibliography

[1] JavaPoint , "Boyce codd normal form (bcnf)." [Online]. Available: https://www.javatpoint.com/dbms-boyce-codd-normal-form

[2] bromega, "To everyone who claims to hate c# but have never used it." 2022. [Online]. Available: https://www.reddit.com/r/ProgrammerHumor/comments/uo56o7/to_everyone_who_claims_to_hate_c_but_have_never/

[3] Fireship, "I built a decentralized chat dapp // gun web3 tutorial," 2021. [Online]. Available: https://www.youtube.com/watch?v=J5x3OMXjgMc

[4] Rebecca Peltz, "Five ways for integrating the cloudinary video player into react applications," 2021. [Online]. Available: https://dev.to/rebeccapeltz/five-ways-for-integrating-the-cloudinary-video-player-into-react-applications-2kcb

[5] Mia Liang , "Understanding object-relational mapping: Pros, cons, and types." [Online]. Available: https://www.altexsoft.com/blog/object-relational-mapping/

[6] Wikipedia contributors, "Haversine formula — Wikipedia, the free encyclopedia," 2022, [Online; accessed 19-May-2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Haversine_formula&oldid=1083812895

Appendix A

Appendix

A.1. Work done by each group member

Name	Student Number	Work done
Nicol Visser	16986431	Client-side solution, Report
Phillip Schommarz	22621105	Client-side solution, Report
Lizé Steyn	21772037	Client-side solution, Report
JC Mouton	22549889	Back-end API, Database Solution, Report

Table A.1: Work done by each group member.