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 15, 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

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

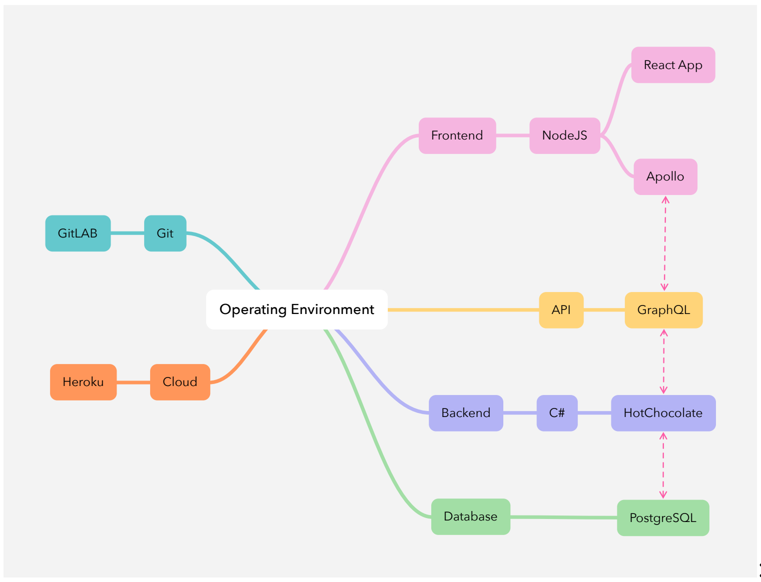
Overall Description

2.1. Use Case Diagram

2.2. Data Modelling

2.3. Operating Environment

There are a wide variety of technologies available to build a web application. Figure 2.1 shows the operating environment of our webs application with some of the core technologies that we decided to incorporate.



;

Figure 2.1: Operating Environment of our Web App

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.

Insert short description of database here

Insert short description of backend server here

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.

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.

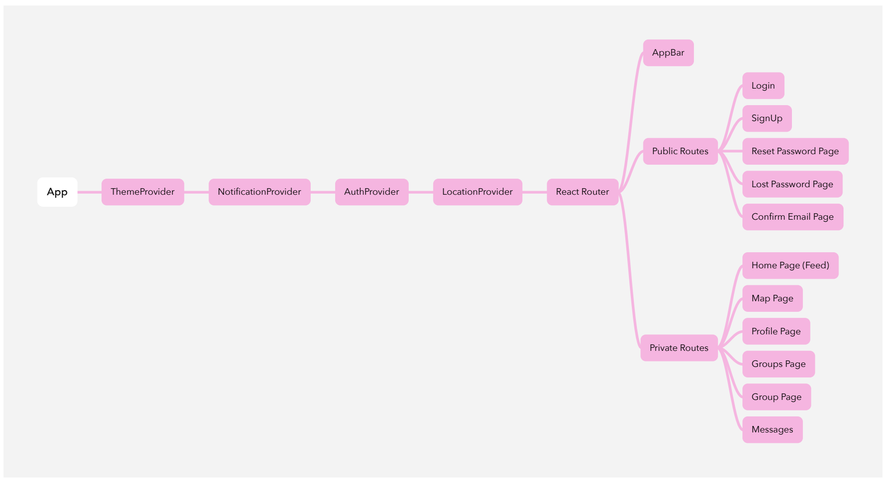


Figure 3.1: Simplified React App Structure

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. It 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 [1] 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 holds context for the post data and functions that act on this data. Descendants of PostProvider can use a hook to have 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.

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 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 [2]. 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. GraphQL

3.3. Backend Solution

Bibliography

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

[2] 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

Appendix A

Appendix

A.1. Work done by each group member

Name	Student Number	Work done
Nicol Visser	16986431	Client-side solution, Back-end API 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.