SAGE

Technical Manual

Authors

Joanna Talvo (18342523)

Chloe Ward (18302716)

Supervisor

Dr. Annalina Caputo

24th April 2022

1. Introduction

1.1 Overview

Our project is a decision making application. Organising plans with a large number of people can often be difficult. SAGE is a web application that will behave as a decision maker which will ultimately simplify the decision making process for friends, family and colleagues.

The main functionality of SAGE is to provide a group of people with an activity based upon their mutual preferences.The application will offer the group admin with 3 categories to choose from such as activities to do, places to eat and movies to watch. This sets out the user survey for the group members to do. For easy access, there is an in app calendar for users to add events to.

The application has a swipe feature where the user can swipe right for “yes” and left for “no” to indicate whether they like or dislike an activity, there are also two buttons that a user can press to indicate whether they dislike or like an activity. This will ultimately bring users to the most common agreed solution for all users - with the most voted activity chosen.

This application is aimed at a wide range of age groups providing they have internet access, as everyone faces decisions in their everyday lives. SAGE aims to simplify the decision making process for groups of friends by providing them with a range of activities to vote on to make planning with friends easier.

1.2 Motivation

The following reasons have motivated the idea of creating a platform that enables people to plan and organise events with their colleagues or friends with no hassle.

2.1 Choice Overload

Nowadays, people are constantly being fed with too many choices. Some people think that this is a good thing but there are hidden negative effects to this. Research shows that when people are presented with too many options, they tend to feel anxious, disengaged, and confused.

Too many choices can be overwhelming and just enough choices can drive sales. Some businesses have good intentions in giving customers lots of choices but this can backfire and become a barrier to their sales. A research from Episerver said that 46% of customers have failed to complete their online purchase due to overwhelming choices.

People can decide a lot quicker when presented with fewer options. It is a challenge to present them options that they are interested in but it is also important to remember to not bore them out with too many options.

2.2 Finding mutual interest

One of the main problems when dealing with planning with groups is mutual interest. It is undeniable that it is one of the reasons why it takes time to pick a place to eat, or a movie to watch, or an activity to do. Some might be interested in outdoor activities while others prefer to stay indoors. It is indeed challenging to find a mutual interest within a group with different preferences and tastes.

2.3 Personal experience of organising plans with a group of people

One of the reasons why we decided to develop this type of platform is due to our own personal experience. It is a common struggle to plan events with a number of people. Most of us plan our activities through chat applications like messenger, telegram, or discord. The planned activities tend to be forgotten since it is not the general purpose of chat applications to save the event. Having a platform that will cater all these needs for planning events would be beneficial for a lot of people.

1.3 Glossary

1. **API:** Application Programming Interface. It is a gateway that allows two applications to talk to each other. Many different API's exist such as Database API's, Operating System API's. An API allows developers to implement functionality without having to code it themselves.
2. **Firebase:** A platform developed by Google to develop mobile and web applications. It gives developers access to a wide range of tools to develop applications quickly and efficiently to a high standard.
3. **Firebase Authentication:** A firebase library that is used to authenticate our applications users by using emails and passwords. When a user signs up it checks to see if the email address is verified and if it is not then an email verification is sent.
4. **Heroku:** is a cloud platform that enables developers to run, deploy and operate applications on the cloud.
5. **Vercel:** is a cloud platform that enables developers to host websites that deploy instantly, scale automatically, with no configuration.
6. **Cross-Origin Resource Sharing (CORS)**: A CORS policy specifies the settings that can be applied to resources to allow Cross-Origin Resource Sharing. It uses a mechanism with additional HTTP headers to inform a browser to allow a web application running at one origin (domain) to have permission to access selected resources from a server at a different origin.

2. Research

Before development of the project started, a research was rigorously carried out for the development of SAGE to decide on what tools and technologies would be beneficial to use. In addition to this, similar applications that have the similar concept were also investigated to discover what SAGE can offer to get ahead of these.

2.1 Next.js

Research was carried out to find better frameworks we can use for our frontend development. Intensive research was carried out to decide between React.js and Next.js on which frontend framework to use. Next.js is a React Framework which means that it has all the benefits that React.js has along with its own benefits. It was then decided that Next.js was better for frontend development as it offers extra features such as speed, fast rendering, better image optimization, eslint compatibility, and more.

2.2 Swipe Feature

Existing applications with swipe features were also investigated such as Tinder and Bumble. The swipe feature was implemented to show similar likes of the users based on their preferences. There were also a few libraries that offered this feature but it was challenging to implement hence why further research and investigation was carried out to fully implement this feature. During the research, we gained an understanding of how the swiping functionality worked and the idea behind why it's popular and user-friendly.

2.3 Material UI

Along with the investigation of our frontend framework, we also looked at libraries that would help us build our user interface. We looked at different libraries that offer the best in styling components such as Material UI, Ant Design, Bulma, and Bootstrap. We ended up deciding on using Material UI because it is simple, intuitive, and offers numerous components that we can implement in our application. Aside from this, it is also being used by big companies such as Spotify, Amazon, Netflix, Unity, NASA, and Shutterstock which gives it a good reputation when it comes to styling components.

2.4 Puppeteer

Implementation of integration tests is also one of the important things we have to investigate. We researched different integration test libraries out there that are simple and easy to use and implement. Puppeteer, Cypress, and Playwright were the libraries that we considered using. In the end, we decided to use Puppeteer although the three of them are similar when it comes to testing. We are also new to Puppeteer hence why further investigation was carried out.

2.5 YELP API

In order to provide choices to our users, we researched existing platforms that offer content on different businesses for our activities and restaurants category. We wanted a platform that will give us good and fresh data in order to provide good quality choices to our users. YELP, TripAdvisor, and Foursquare were the three platforms we investigated. YELP and TripAdvisor offer various elements such as reviews, location, terms, category, ratings, and more. We decided to use YELP as its an open source platform, it is easier to use, and it is also being used by many developers. TripAdvisor also has some limitations when it comes to using their API.

2.6 TMDB API

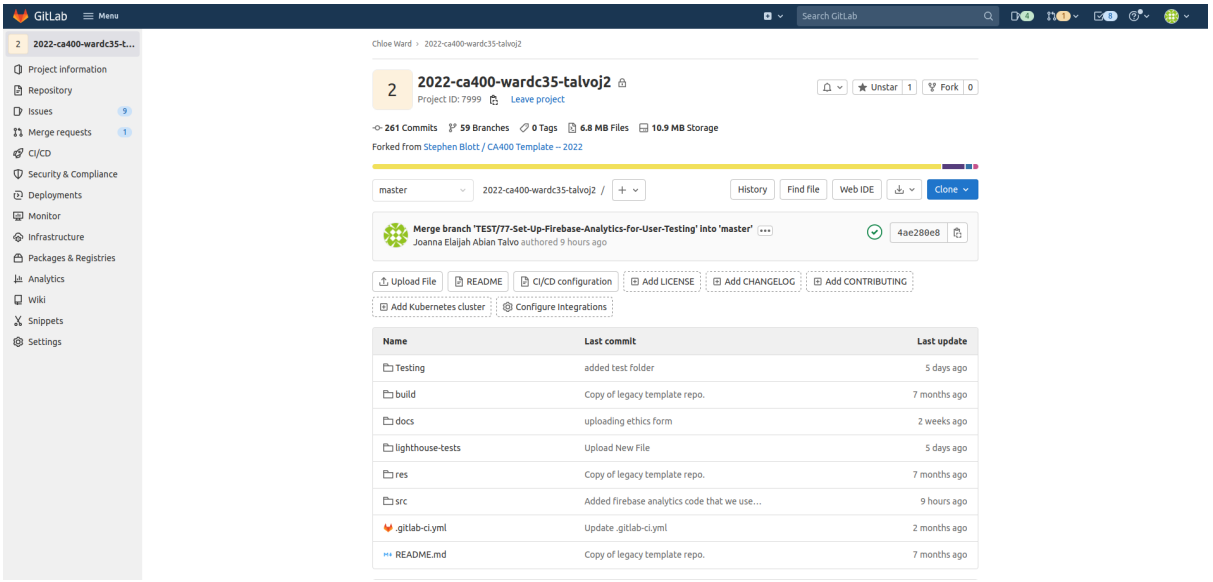
Similar to our activity and restaurant category, we also researched on existing platforms that we can use for our movie category. We looked at TMDB and IMDB, and investigated which provides better content for movies. Unfortunately, IMDB API's free version is limited to 1,000 API calls per day. We opted to use TMDB because it has no rate limits being imposed on the API. Aside from this benefit for us developers, it also offers a wide range of movies to choose from. It is also easy to use and implement.

3. System Architecture

3.1 GitLab

3.1.1 Source Code

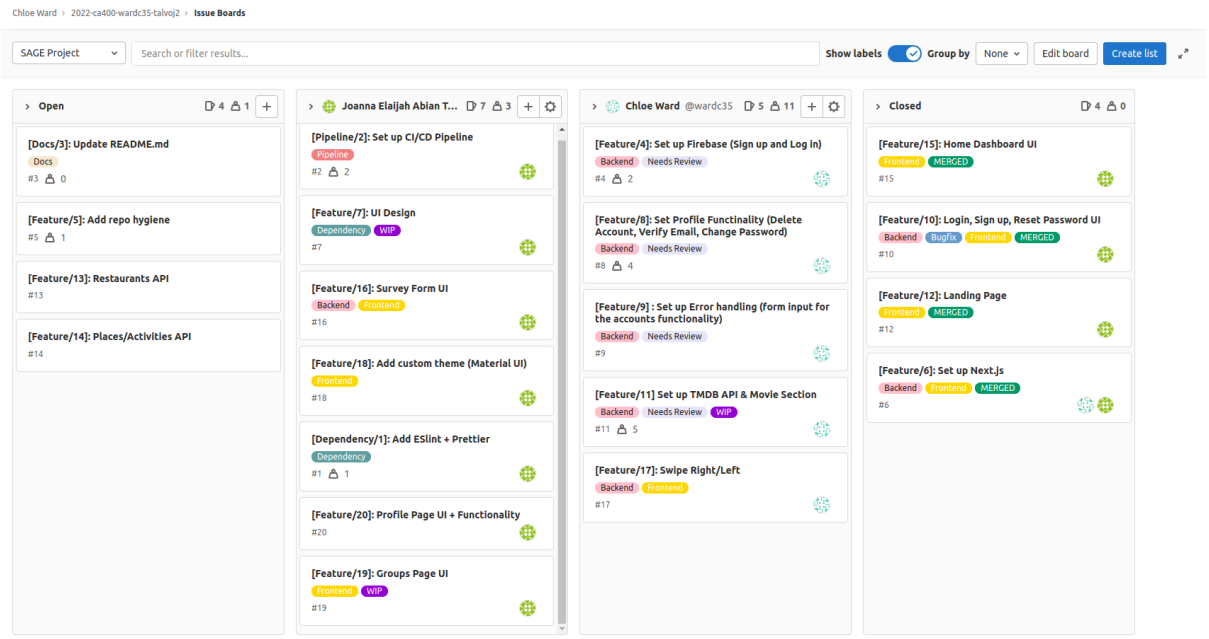
The source code of the project is hosted by Dublin City University's GitLab.



SAGE Gitlab Repository

3.1.2 Collaboration

DCU GitLab is the main collaboration tool, and issue and project tracking management used to develop this project. We have our own naming convention of our issues and labels in order to have a more productive and efficient project management.



Issue Board on Gitlab

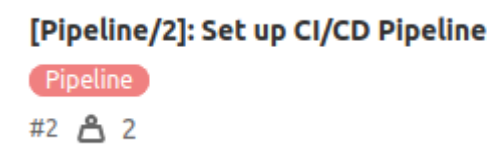
The following are the types of issues we have:

- **Fix** - add this label when the merge request is about fixing a bug.
- **Feature** - add this label when the merge request is about adding a new feature or updating an existing feature.
- **Docs** - add this label when the merge request is about updating / adding information in the documentation.
- **Dependency** - adding dependencies or libraries needed in the project. However, dependencies can be added together with the **Feature** issues. This will only be used when dependencies are added separately.
- **Pipeline** - any issues relating to pipeline.

[TYPE-OF-PR/Issue-number]: Title of PR

[Feature/23]: Adding ‘create group’ button in homepage

Issue number is shown in the issue ID. It looks like **#2**



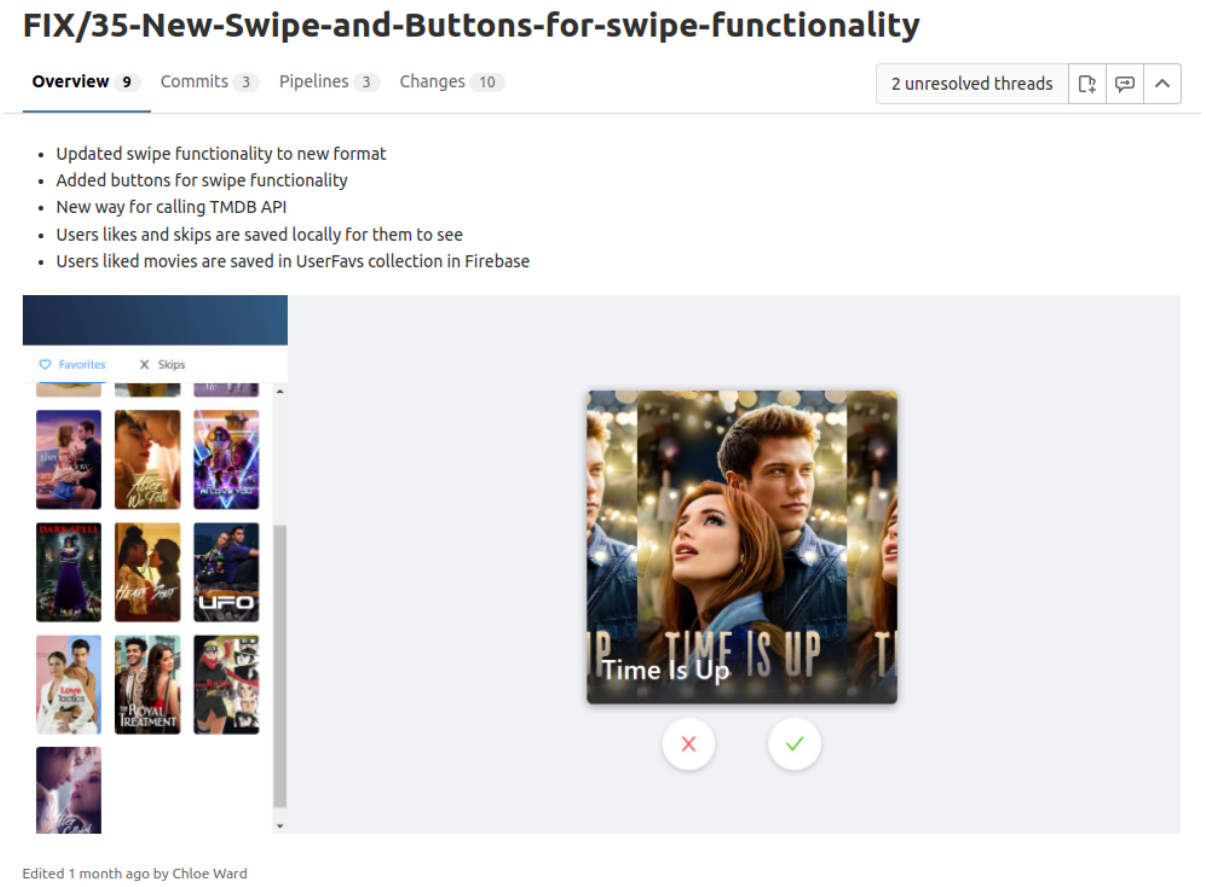
Branch name should match the issue title. For example, the issue title is

[Feature/4]: Set up Firebase (Sign up and Log in)

branch name should be:

FEAT/4-setup-firebase-signup-login

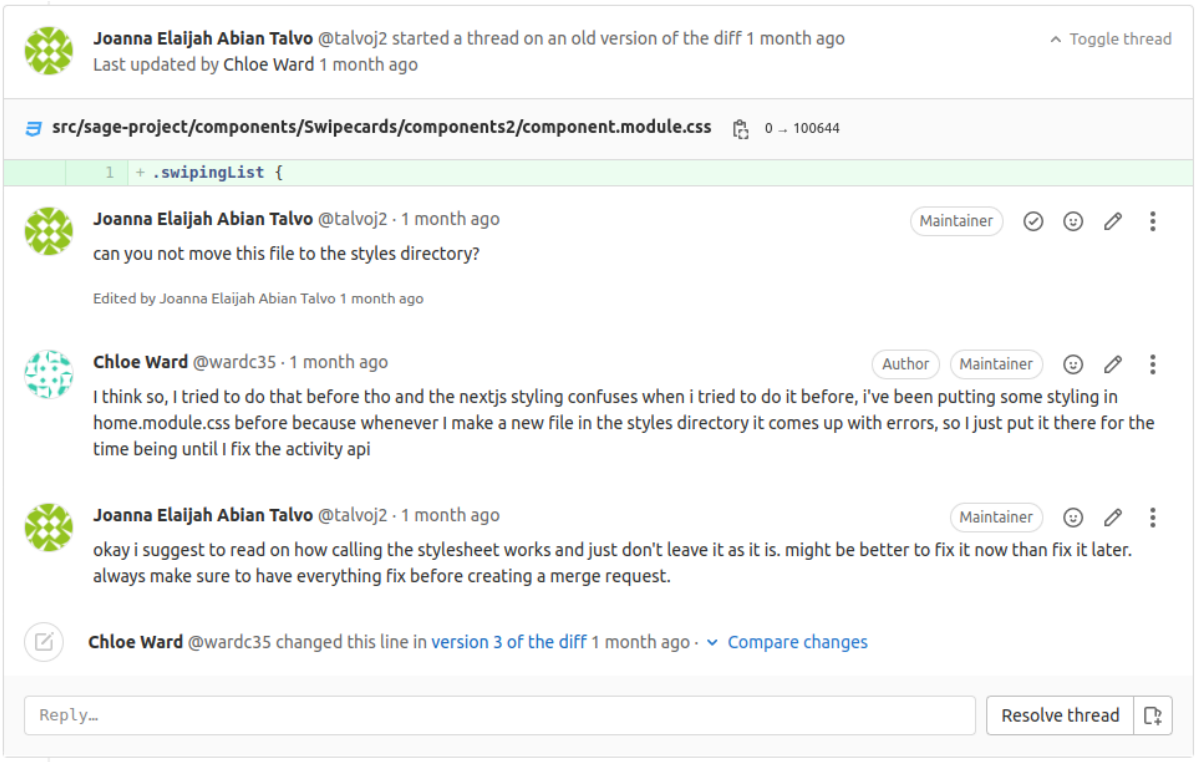
We also made sure to describe the merge request in full detail and include screenshots of the changes as much as possible in order for the code reviewer to understand the changes whilst looking at the code.



Merge Request Overview

3.1.3 Code Reviews

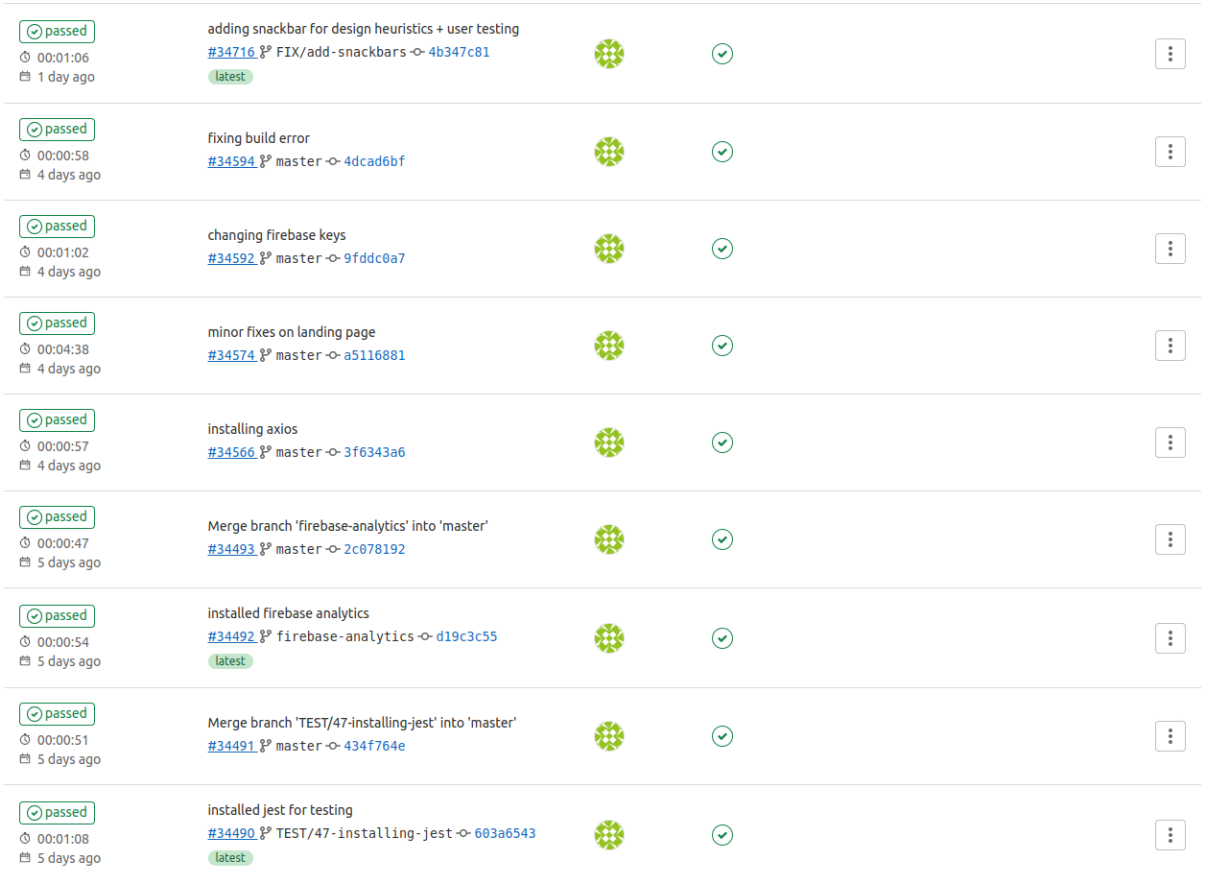
The developers of this project ensured to regularly make code reviews before merging branches into master. It is important to have code reviews in order to improve the quality of the code and spot mistakes. It is ensured that an approval from fellow developers is required before merging into master branch.



Comments on a Merge Request on GitLab.

3.1.4 CI/CD Pipeline

The pipeline is triggered automatically once a merge request has been created. It is set up in a way that the pipeline needs to pass before merging the merge request into the master branch (alongside an approval). This is extremely important and helpful to the developers as it allows us to make sure that all tests and changes on that branch pass. This also boosts the team's confidence level as they see their code pass with no problem.



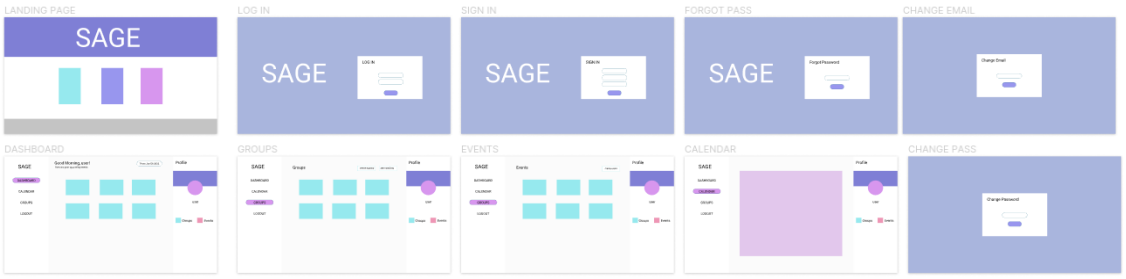
CI/CD Pipeline

3.2 Frontend

The frontend of SAGE is developed using Next.js and Material UI. We decided to use these frameworks for the frontend development of our application as we believe that they offer the best tools for project development.

3.2.1 Figma

The first task of frontend development is to design the user interface. For this project, we made use of Figma which is a web-based prototyping tool for the design of our mockup UI. During the development of the UI design, we made sure to obey Shneiderman's Eight Golden Rules of User Interface. A UI mockup was created in order for developers to have a design guide when developing the frontend. The colour scheme was also initially planned during this phase. Although most of the mockup layout was implemented, some of them were modified in order to produce a more efficient UI. Results from user testing were also taken into account to improve the UI.



Initial User Interface design on Figma

3.2.2 Next.js

The frontend uses Next.js which is a frontend React framework to develop the user interface. Since this is a React framework, it carries the benefits of React. In addition to these benefits, Next.js also offers additional features which help development and project management better. It is easier to code in Next.js as developers just need to create the page and link to the component in the header. It also instantaneously updates the changes we made in our components so reloading the page is not necessary. Apps that are built with Next.js are fast because of the server-side rendering and static generation it offers. Other amazing features that it offers are eslint compatibility, built-in css, easy deployment, and more.

3.2.3 Material UI

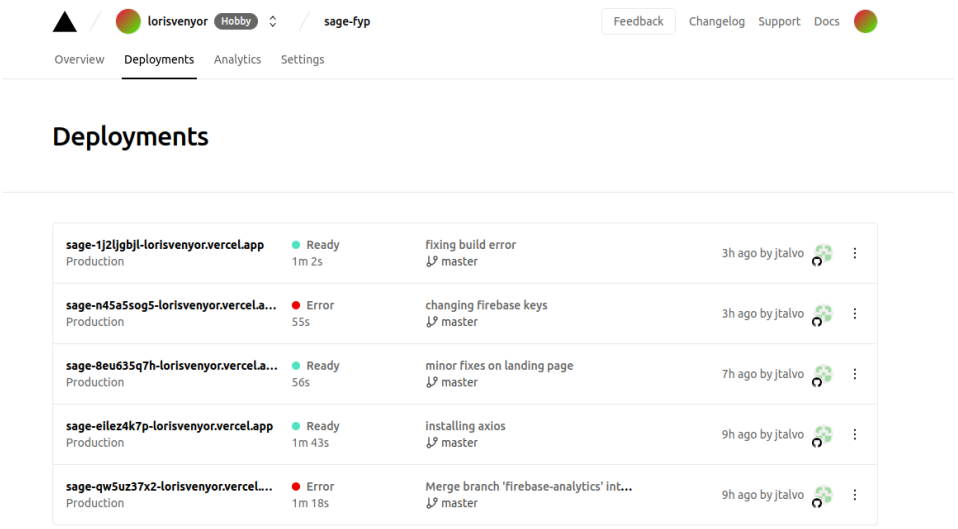
Another React framework, Material UI is a library that allows developers to import and use various different useful components to create user interfaces in React applications. It provides styled components that are easy to implement as well as to customise. It makes developing way easier and efficient as us developers don't need to worry about styling our own components. Components such as drawers, cards, paper, and containers are few of the components that were mostly used in designing the frontend. It also offers customisation of your own theme for your project.

3.2.4 Firebase Authentication

The frontend communicates with Firebase Authentication to determine if a user has been authenticated to use the application. Sign up functionality communicates with Firebase Authentication to verify if the user is eligible to sign up meaning they have no registered account in the application yet. Firebase Authentication makes sure that the information provided by the users is private.

3.2.5 Vercel

The frontend is hosted using Vercel which is a platform for deploying fast React websites. This platform offers the best frontend infrastructure from zero configuration. It provides an easy-to-use experience for the developers and simplifies the process of deploying sites that are both fast and satisfying to users.



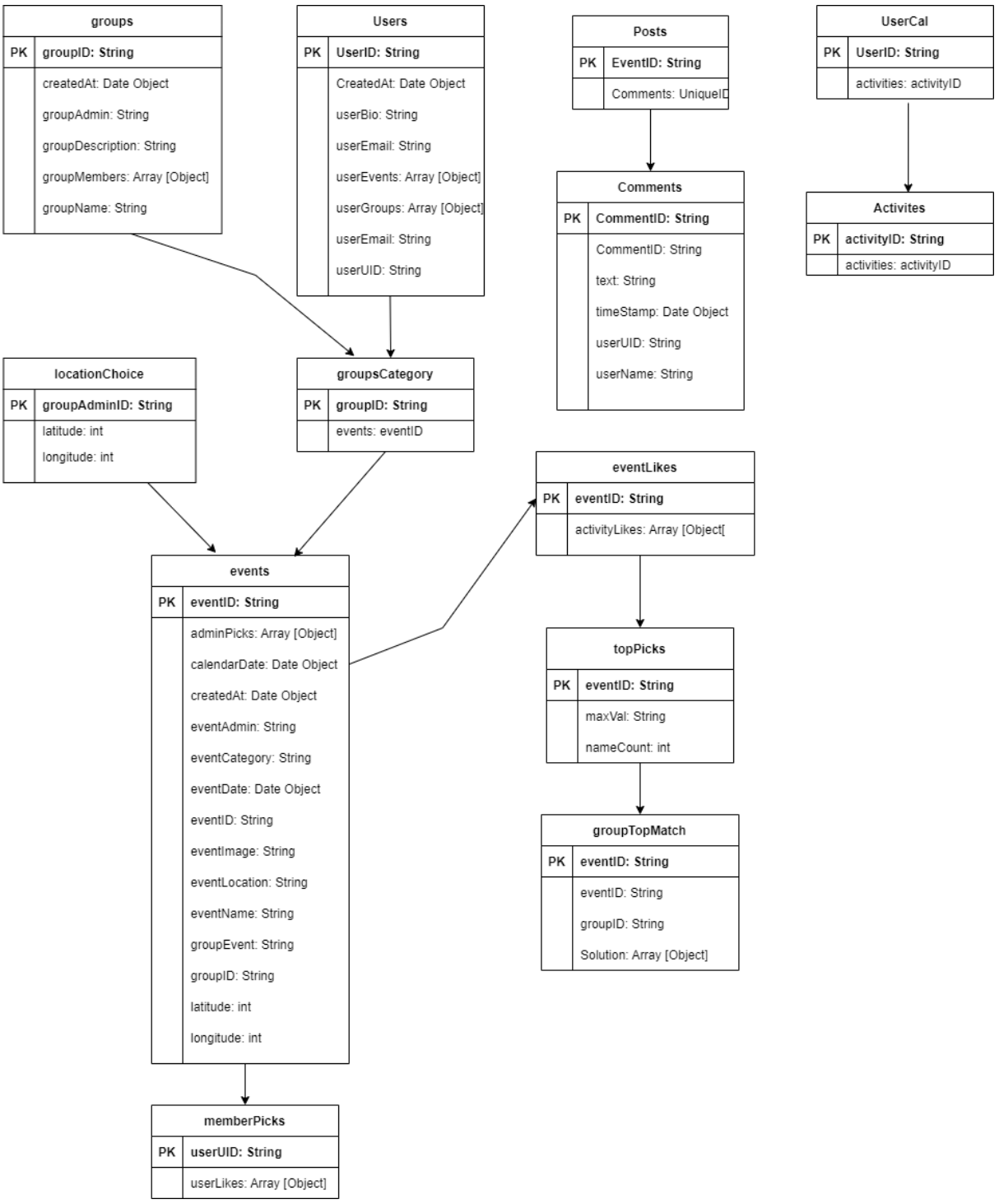
Deployments of application on Vercel

3.3 Backend

The backend functionality is written in Javascript. It is used for the processing and communication between the client and API calls to the endpoint that are set up. Node js interacts with various API's in our application, such as The Movie Database (TMDB), YELP API, Leaflet which is a javascript library for interactive maps etc. Firebase Cloud Firestore is used to store user data so that we can easily retrieve and store data in the form of documents. The backend implementation also includes a server which is hosted on Heroku to bypass the CORS policy for the YELP API.

3.3.1 Database

Firebase Cloud firestore is the database that is used to store the users data. The data is stored in documents, which are organised into collections. We have structured the database in a way which accommodates for the groups changing likes activity, adding to groups etc. There are a total of 12 collections in the database. The diagram below shows the database structure which we have converted from document format to a table format for easy viewing purposes.



Database Diagram

3.3.2 YELP API

Yelp API is used to gather all the information on local businesses, ratings, reviews, location and images in JSON format. We called specific YELP api endpoints to gather businesses, restaurants etc that were near the current users location to populate the application with data for the chosen activity choice of the application.

3.3.3 TMDB API

The Movie Database (TMDB) API was used to populate the application with movies that the users can choose from. The API provides us with a fast and reliable way of getting movie information such as ratings, movie images, overviews etc.

3.3.4 Leaflet Interactive Map

Leaflet is an open-source javascript library for interactive maps. This was used to implement the user location choice functionality with the aid of an interactive map with a location marker that the user can drag and drop which is then written to our database for each individual group.

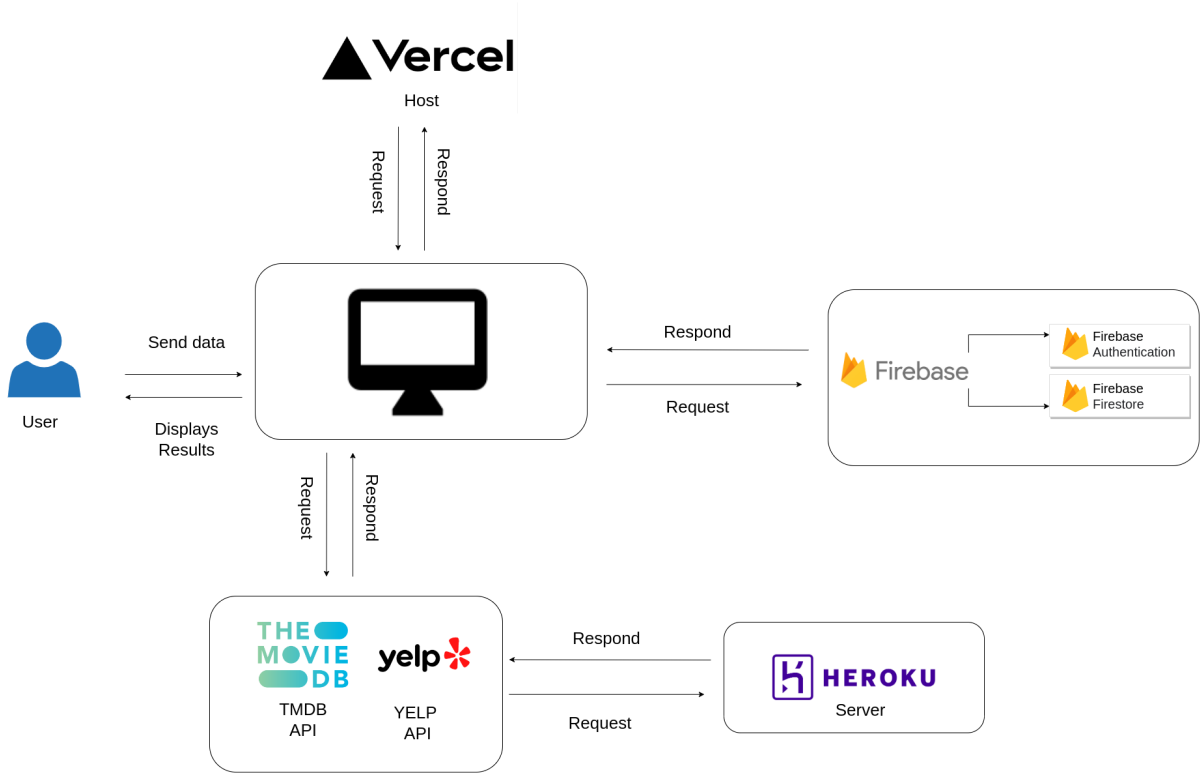
3.3.5 Heroku

We create a server on Heroku which is a cloud platform that enables developers to run, deploy and operate applications on the cloud. The server uses an API that enables cross-origin requests to anywhere to bypass the fact that the YELP API endpoints do not include the Access-Control-Allow-Origin response header.

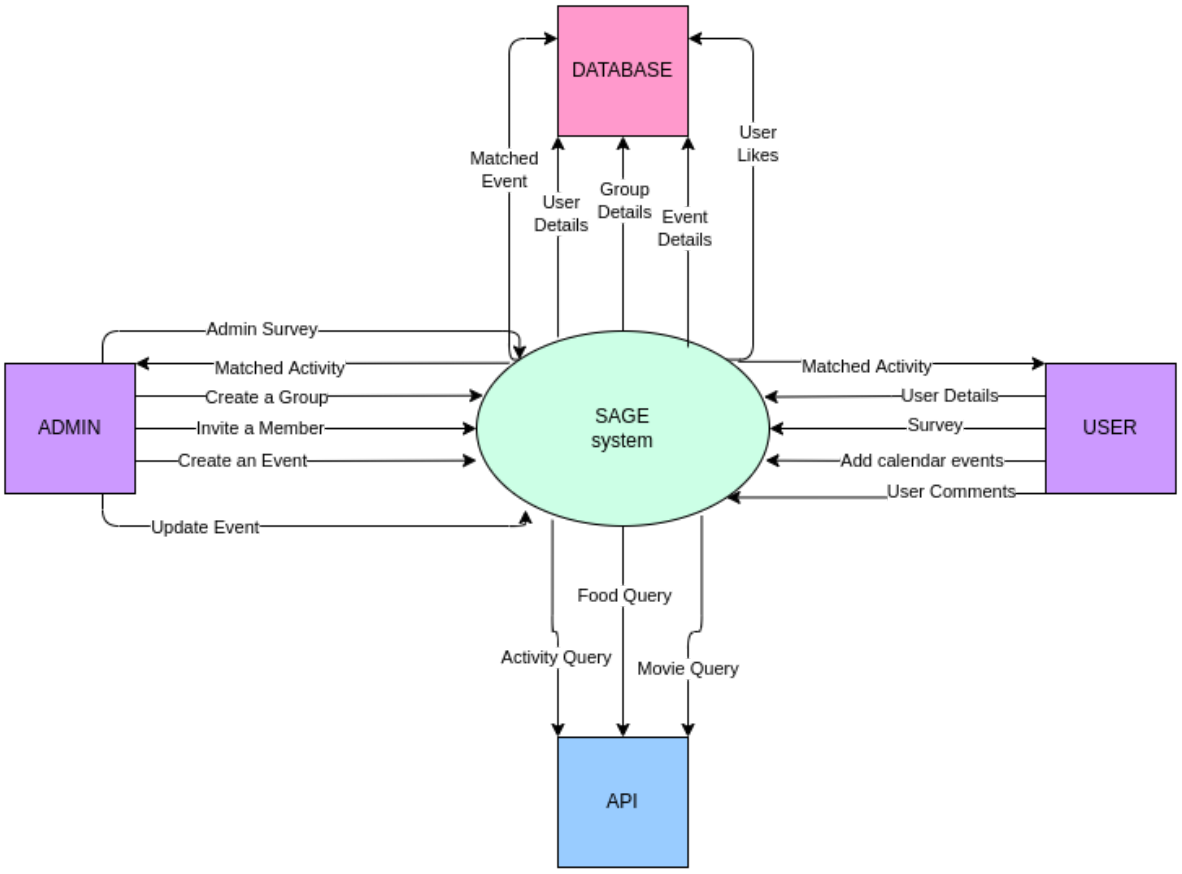
3.3.6 Firebase Authentication

Firebase authentication is a library that is used to authenticate our applications users by using emails and passwords. When a user signs up it checks to see if the email address is verified and if it is not then it is added to the database and the user is able to sign up.

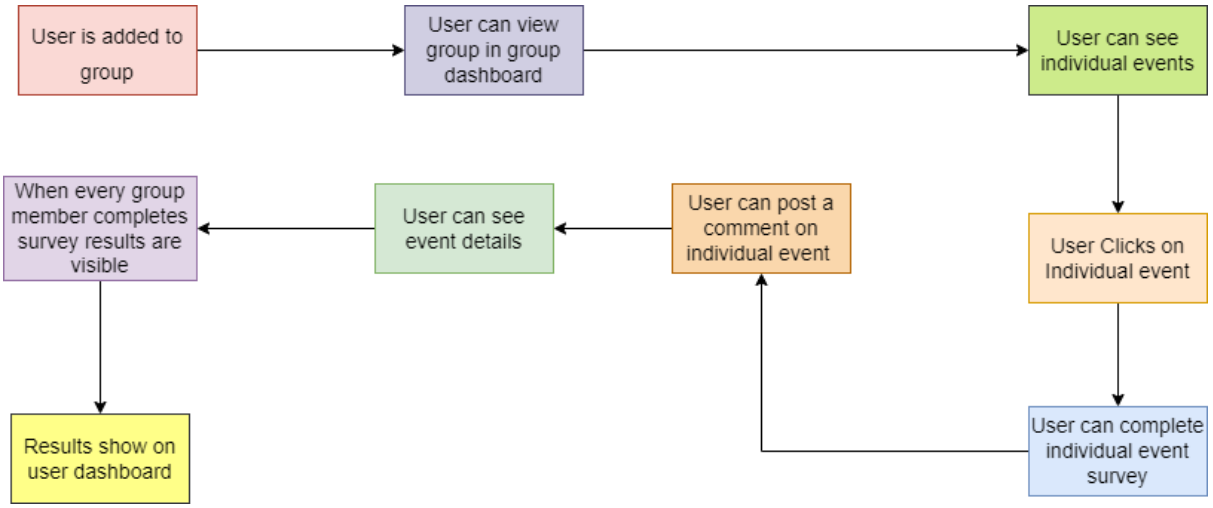
3.4.1 System Architecture



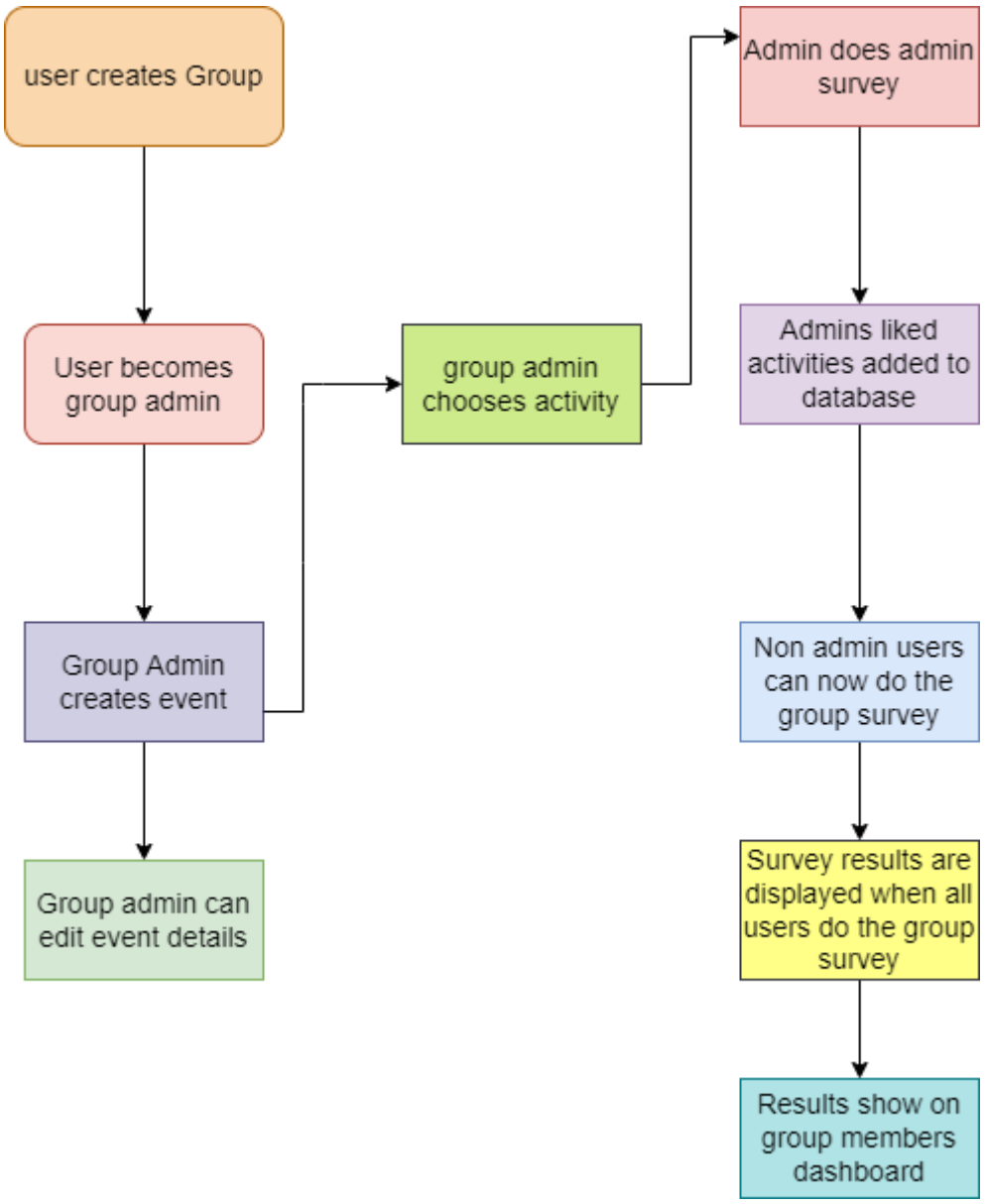
3.4.2 Context Diagram



3.4.3 Data Flow Diagram (Non-Admin)



3.4.4 Data Flow Diagram (Admin)



4. Implementation

4.1 Frontend

4.1.1 Login Page

The login page handles the initial state of user authentication. This page determines if a user trying to login can be authenticated or not. If successful, they already have a record in our database otherwise, they have to create an account. The frontend will communicate with the database in order to verify if the account entered exists in the database, and if not, the frontend will return an error message that the account does not exist in the database. In Figure 1, the login function takes in the email and password and passes it into Firebase function to authenticate the user.

```
async login({ email, password }) {
    return await this.auth.signInWithEmailAndPassword(email, password)
}
```

Figure 1

In figure 2, the login page has a useEffect hook that checks if the user is still logged in to the application. It will redirect the user straight to the home page without having to log in again when accessing the application.

```
useEffect(() => {
    if (firebase.isLoggedIN()) {
        Router.push("/home");
    }
}, []);
```

Figure 2

4.1.2 Sign Up Page

Users with no existing account are redirected to the sign up page. This page determines if the credentials being submitted are valid or not. It ensures that the email address being used is valid. In figure 3, a regEx pattern is implemented to validate the email address.

```
{...register("email", {
    required: "Please enter your email address",
    pattern: {
        value: /^[A-Za-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$/,
        message: 'Invalid email address'}})}
```

Figure 3

In addition to this (figure 4), it also checks if the password is valid. It is set up in a simple way wherein the password should be 6 characters long.

```
{...register("password", {
    required: "Please enter a password",
    minLength: {
        value: 6,
        message: "Incorrect password."
    }})}
```

Figure 4

Upon creating the account, the frontend communicates with the database and saves the credentials that the user has entered which are the email address and the password. In figure 5, the register function takes in name, email, and address. It also sends an email to the user for verification of email address.

```
async register({ name, email, password }) {
    const { user } = await this.auth.createUserWithEmailAndPassword(email, password)
    await createUserDocument(user, name)
    await this.auth.currentUser.sendEmailVerification()
    return this.auth.currentUser.updateProfile({
        displayName: name,
    })

}
```

Figure 5

4.1.3 Sidebar Navigation

The sidebar navigation bar contains 4 buttons. Each button is a link to different pages which are Dashboard, Calendar, Groups, and Logout. The sidebar navigation is set up in a way that the active page will be noticeable with the use of adding colours. The Dashboard button is the initial page when the user is logged in to the application. It contains the upcoming events of the user. The Calendar button redirects the user to the calendar page. The Groups button redirects users into the groups page wherein they can view the groups they are part of. Lastly, the user can interact with the logout button and this will redirect the user back to the login page.

4.1.4 Profile Sidebar

The profile sidebar contains the information about the user; name, bio, email address, number of events, and number of groups. The frontend communicates with the database where it pulls the real time data to be displayed on the profile bar. A couple of useEffect hooks were implemented in order to update the user profile data each time the component is rendered. In figure 6, Firebase has the onSnapshot function that allows the component to pull the real time data.

```
useEffect(() => {
    async function fetchData() {

        await fire.firestore().collection('users').doc(userID)
        .onSnapshot((querySnapshot) => {
            setName(querySnapshot.data().userName)
            setBio(querySnapshot.data().userBio)
        }, error => {
            console.log("Error getting documents: ", error);
        })
    }
    fetchData()
});
```

Figure 6

4.1.5 Dashboard

The dashboard page displays the banner which is customised for each user and time dependent, the current date, and the upcoming events of the user. The events are displayed in order of their creation. The frontend communicates with the database through a useEffect hook in order to pull the real time data each time the component is rendered. In figure 7, the

function fetchAllUpcomingEvents pulls the data from the database which is the upcoming events of the current user.

```
useEffect(() => {
    let isMounted = true;

    async function fetchAllUpcomingEvents() {
        for (const event of allUserEvents){
            await fire.firestore()
            .collection('groupTopMatch')
            .doc(event)
            .onSnapshot(snapshot => {
                if(isMounted){
                    setUpcomingEvents( arr => [...arr, snapshot.data()])
                }
            })
        }
    }

    fetchAllUpcomingEvents()

    return () => {
        isMounted = false
    };
}, [allUserEvents]);
```

Figure 7

4.1.6 Map

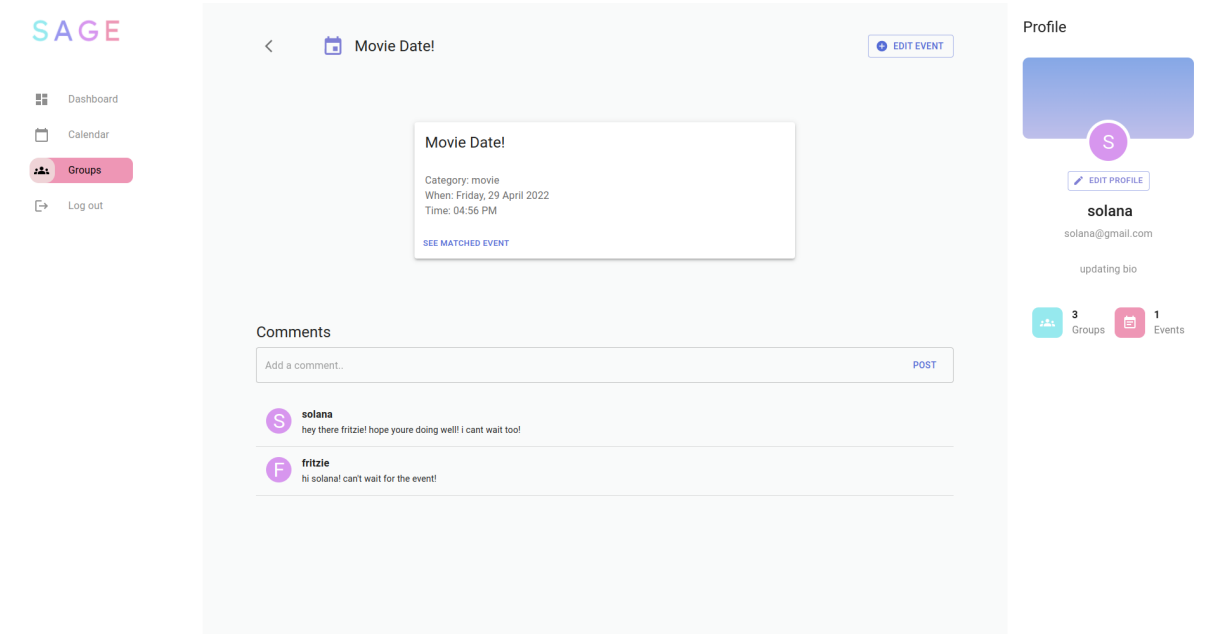
The map page uses the Leaflet library which is a JavaScript library for interactive maps. This allows the user to pick a location they want. This location will then be saved and it will be used to only locate the places that will be offered to the user that are nearby this location. The map is set up in a drag and drop mode to make it user-friendly and interactive. It is also scrollable by mouse to make it more accessible.

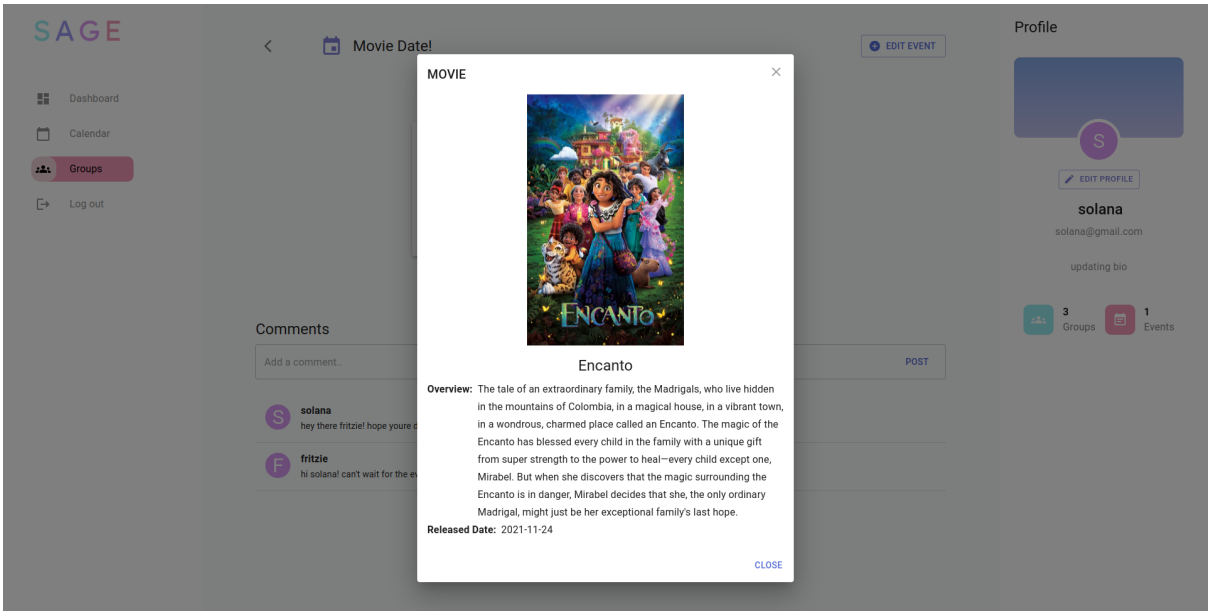
4.1.7 Calendar

The calendar page uses the React Full Calendar library. The user can interact with this by clicking on the date and adding an event. The frontend then communicates with the database and it saves the data into each user's calendar collection. It also pulls the events saved in the calendar each time the calendar component is rendered.

4.1.8 Matched Event

The matched event page displays the matched event of the group along with its details such as the category, date, and time. It also displays the details about the specific category type chosen by the group. A comment section is also included into this page in order for the group members to communicate with each other. The comments are displayed in real time which is done by frontend communicating with the database. The comments are also displayed in order of their timestamp.



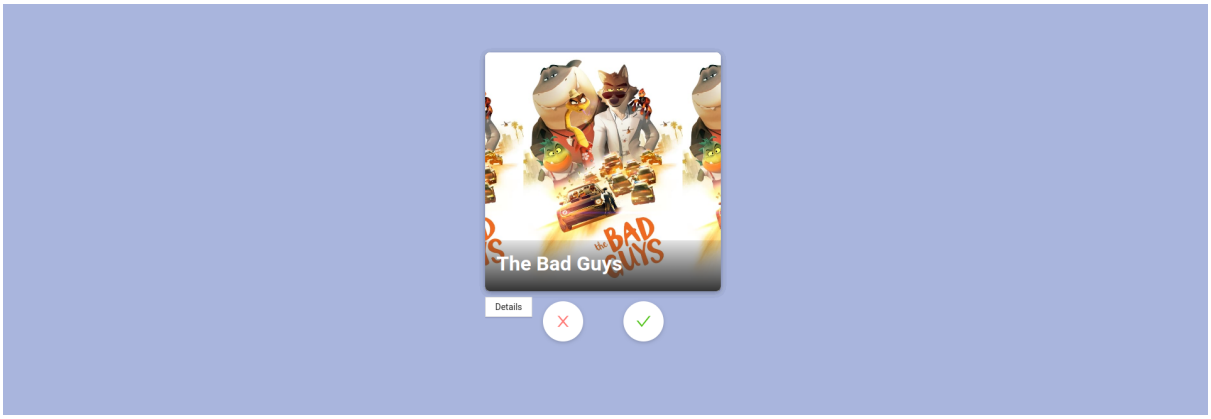


4.1.9 Groups and Events Pages

The groups page displays the groups that the user is a member of while the events page displays the events of a specific group the user is a member of. This is displayed in order of their timestamp in descending order. The frontend communicates with the database by the use of useEffect hooks and Firebase functions. The data is then displayed in real time with the use of the onSnapshot Firebase function.

4.1.10 Swipe Cards

The swipe cards are developed in such a way that it behaves like Tinder or Bumble where the user swipes left if they are not interested and right if they are interested. There is a button functionality where the user can click the button with the green tick to indicate “yes” or the button with the red tick to indicate “no”. There is also a “Details” button which shows the activity details, such as rating, location, overview etc depending on the type of activity.



Below is some of the code that contributed to creating the swipe functionality. Below is a function that handles the swipe, when a card is swiped upwards, the css style opacity is reduced to 0 so that we can see the card behind it, and if the the card was placed down from an upwards position, the “LIKE” or “SKIP” icon that appears on the card depending on which direction the card was going will reduce to 0 so that we can’t see it anymore.

```
const handleMouseUp = ({ currentTarget }) => {
  setInitialXY(null);
  const swipingEl = currentTarget;
  const { x } = getXYFromTransformCss(swipingEl.style.transform);
  const shouldRemoveCard = Math.abs(x) >= 100;
  if (shouldRemoveCard) {
    swipingEl.style.opacity = 0;
    setTimeout(() => {
      x >= 0 ? onSwipeRight() : onSwipeLeft();
    }, 500);
  } else {
    swipingEl.style.transform = initialStyle.transform;
    const [_cardEl, skipMessageEl, likeMessageEl] = swipingEl.children;
    skipMessageEl.style.opacity = 0;
    likeMessageEl.style.opacity = 0;
  }
};
```

4.2 Backend

4.2.1 Communicating with the database

Communicating with the database was a vital aspect for our application. All of the users data is stored in the database so it is vital that when the users input is imported to the database that it is stored in the correct collection.

4.2.2 Collections

When creating a collection in our database, it was vital for us to organise them in such a way that it would allow us to connect documents based on what groups a user was in

and with the event ID's. Everything in the database intertwines and connects with each other,and when we were designing it this was our prime goal.

4.2.3 User Likes

Below is how we designed our *'groupsCategory'* collection in the database. We have the document ID as the individual *'groupID',* a nested collection called *'events'* with another nested document called *'eventID'* which has a *'membersPicks'* collection which contains all of the users likes for an individual event in a group that has multiple events. This implementation was created due to the fact that multiple users could be in multiple groups so by doing this we got the groupID that a user was in and the individual eventID when manipulating the data for the matchmaking algorithm.

```
let currentUserUID = fire.auth().currentUser.uid

const docRef = fire.firestore()
.collection('groupsCategory')
.doc(groupID)
.collection('events')
.doc(eventID)
.collection('memberPicks')
.doc(currentUserUID)

docRef.set({
    userLikes: likedData
})
.catch((err) => {
    alert(err)
    console.log(err)
})
```

4.2.4 User Document

When a user signs up to firebase, we wrote a function which would create an individual firebase user collection with a document of each user's ID. Inside this document it has all the fields such as userName, userEmail, userGroups which is an array that contains the group ID of each group the user was invited to etc.

```
if(!userDocument.exists) {
    console.log('userDocument does not exist, create a document')
    const { email } = user;

    try {
        console.log('adding user credentials to the database')
        userRef.set({
            userName: name,
            userEmail: email,
            userUID: user.uid,
            userBio: '',
            userEvents: [],
            userGroups: [],
            createdAt: new Date(),
        })
        console.log("added to the database")
    } catch(error) {
        console.log(error)
    }
} else {
    console.log('it exists')
}
```

When a user updates their user email with our change email functionality, the user email is updated in the user document with the update() function that firebase offers, which can be seen below.

```
const updateUserEmail = (newemail) => {

    fire.firestore()
    .collection('users')
    .doc(userID)
    .update({
        userEmail: newemail
    })
    .catch((err) => {
        alert(err)
        console.log(err)
    })

    router.push("/home");

}
```

4.2.5 Create Group

When a user is creating a group, it creates a group collection in firebase. A random document id is assigned to the document which acts as the groupsID. All the groups data is stored inside the group document including the groupAdmin which is vital because admin's have extra functionality that non admins do not have.

```
const createGroup = () => {

    let currentUserUID = fire.auth().currentUser.uid

    const docRef = fire.firestore().collection('groups').doc()
    docRef.set({
        groupName: groupName,
        groupDescription: groupDescription,
        groupMembers: groupMembers,
        groupID: docRef.id,
        createdAt: new Date(),
        groupAdmin: currentUserUID
    })
    .catch((err) => {
        alert(err)
        console.log(err)
    })
    // this saves the groups document ID into each user's userGroups array field
    setUserGroups([...userGroups, docRef.id])
}
```

4.2.5 Invite member

A user admin has the ability to invite members into the groups they created by adding the email address of the user they wish to invite. *getUserDocRef()* function uses a *where()* function which checks if the email address that a user once wants to invite is a field in the users collection. This returns the document ID which is the user's ID and it is then added to a *setMembers()* hook

```
// THIS RETURNS THE DOC REFERENCE OF THE USERS TO BE ADDED IN THE GROUP
useEffect(() => {
async function getUserDocRef() {

    await fire.firestore().collection('users').where("userEmail", "==", userEmail)
    .onSnapshot((querySnapshot) => {
        querySnapshot.forEach((doc) => {
            setMembers(doc.id)
        });
    }, error => {
        console.log("Error getting documents: ", error);
    })
}

    getUserDocRef();
}, [userEmail]);
```

We also catered for when a user's email did not exist in the users collection. This is done by querying the Firebase by checking if the userEmail is found in the users collection.

```
const allUsers = () => {

    fire.firestore()
    .collection('users')
    .where("userEmail", "==", userEmail)
    .onSnapshot((querySnapshot) => {
        if(querySnapshot.docs.length == 0){
            // no user
            setNoUser(true)
        } else {
            setNoUser(false)
        }
    }, error => {
        console.log("Error getting documents: ", error);
    })

}
```

When a user is successfully added to a group, the group's document ID is added to the usersGroups array with firebases update() function.

```
const updateUserGroup = () => {
    fire.firestore().collection('users')
    .doc(members)
    .update({
        userGroups: fire.firestore.FieldValue.arrayUnion(groupDocRef)
    })
    .catch((err) => {
        alert(err)
        console.log(err)
    })
}
```

The same theory also applies to when a user is added to a group, the *groupMembers* field is updated with the usersID in the groups collection().

```
const updateGroupMembers = () => {

    fire.firestore().collection('groups')
    .doc(groupDocRef)
    .update({
        groupMembers: fire.firestore.FieldValue.arrayUnion(members)
    })
    .catch((err) => {
        alert(err)
        console.log(err)
    })
}
```

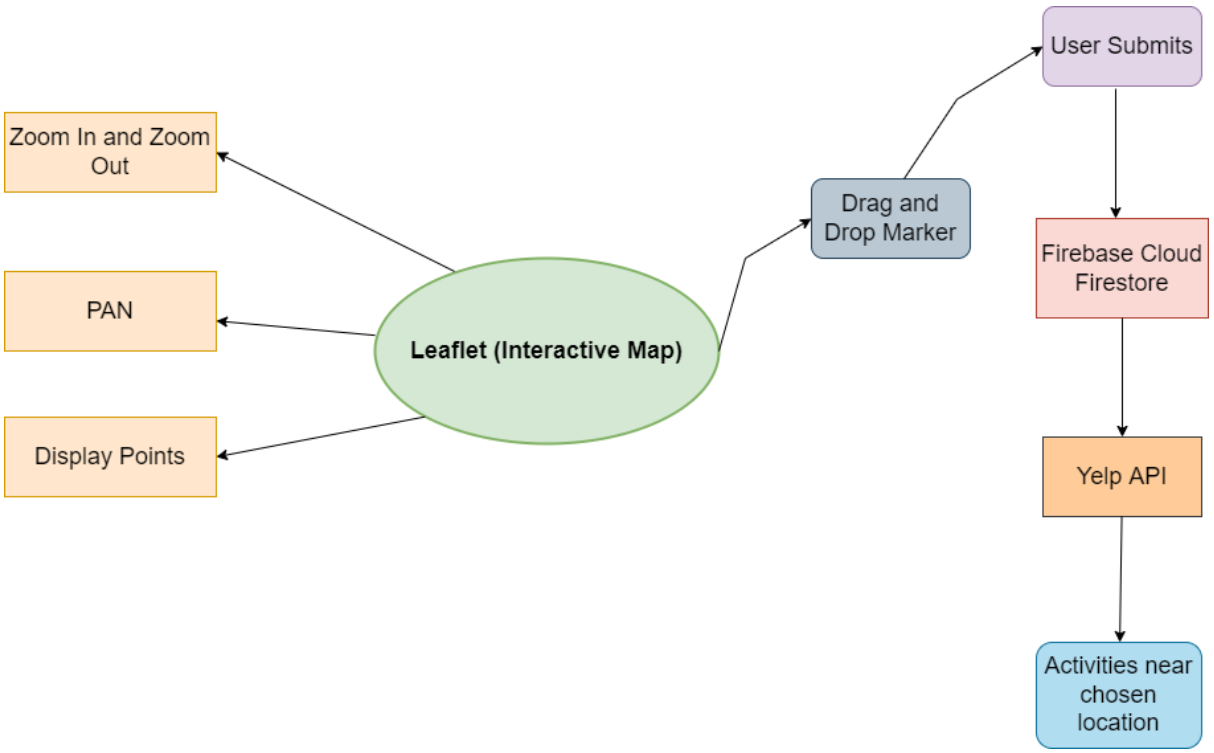
4.2.6 Create Event

When a group Admin presses the *create Event* button, it gets stored in a nested collection where we can access it by groupID. All the data below is stored inside the events collection under a document that is assigned a random ID which acts as the individual event

ID. The event location latitude and longitude is stored in this document, where the admin drags and drops a marker to determine the location of the event.

```
const createEvent = () => {

    const docRef = fire.firestore()
    .collection('groupsCategory')
    .doc(groupID)
    .collection('events')
    .doc()

    docRef.set({
        groupEvent: '',             // this is the result after the matching -- name of the event
        eventImage: '',             // this will be pulled from the matching result
        eventCategory: urlCategory,
        eventDate: '',
        eventTime: '',
        eventLocation: '',
        eventName: docRef.id,    // this can be edited by the Admin only
        eventID: docRef.id,
        adminPicks: [],
        eventAdmin: groupAdmin,
        createdAt: new Date(),
        calendarDate: '',
        groupID: groupID,
        longitude: location.lng,
        latitude: location.lat,
    })
    .catch((err) => {
        alert(err)
        console.log(err)
    })

    updateUserEvents(docRef.id)

    urlCategory == 'activity' ? router.push('/categories/activity/' + groupID + '&' + docRef.id) : router.push('/categories/food/' + groupID + '
```

Below is a data flow diagram of the map functionality with firebase.



Leaflet map with firebase data flow diagram

4.2.7 Edit Profile

When a user wishes to edit their profile, such as their name, or add their own personalised bio, it is updated in the *handlePress()* function. The usersName and userBio fields are updated in the user collection by using the *update()* firebase functionality to overwrite the specified fields only.

```
const handlePress = async () => {

    const db = fire.firestore();
    db.collection("users")
    .doc(currentUserUID)
    .update({
        userName: name,
        userBio: bio
    })
    setOpenSnackBar(true)
    // setOpen(false);
}
```

4.2.8 Map Functionality

The user map functionality works by having a drag Marker, where the latitude and longitude is constantly updated when the marker is moved around the interactive map.

```
<MapLeafletDynamic
    center={location}
    location={location}
    draggable={true}
    title="sample text"
    onDragMarker={(e) => {
        console.log("e",e);
        let loc = {lat: e.lng, lng:e.lat};
        setLocation(loc);
    }}
/>
```

Below, the code checks if the marker doesn't equal to null, then the Leaflet onDragMarker() function, gets the latitude and longitude of the marker's location on the map.

```
const dragHandlers = useMemo(
    () => ({
      dragend() {
          const marker = markerRef.current;
          if(marker != null ) {
              onDragMarker(marker.getLatLng());

          }
      },
    }),
    [onDragMarker]
);
```

We make use of the leaflet OpenStreetMap API, to achieve our interactive map functionality

```
<MapContainer
    center={[53.384810, -6.263190]}
    zoom={15}
    scrollWheelZoom={true}
    style={{ height: "75vh", width: "75vw"}}
>
    <TileLayer
        attribution='&copy; <a href="http://osm.org/copyright">OpenStreetMap</a> contributors'
        url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
        />

    <Marker
        icon={L.divIcon({
        iconSize: [60, 60],
        iconAnchor: [60, 60],
        className: "mymarker",
        html: "🔴",

        })}
        position={[
            location && location.lng ? location.lng : "",
            location && location.lat ? location.lat : "",
        ]}
        draggable={draggable}
        eventHandlers = {dragHandlers}
        ref={markerRef}
    >
        <Popup>{"Chosen Location"}</Popup>
    </Marker>
</MapContainer>

);
```

This data is then added to the database due to the createEvent function, by adding it to the docRef for the individual Event under the fields *Longitude* and *Latitude.*

```
docRef.set({
    groupEvent: '',
    eventImage: '',
    eventCategory: urlCategory,
    eventDate: '',
    eventTime: '',
    eventLocation: '',
    eventName: docRef.id,    //
    eventID: docRef.id,
    adminPicks: [],
    eventAdmin: groupAdmin,
    createdAt: new Date(),
    calendarDate: '',
    groupID: groupID,
    longitude: location.lng,
    latitude: location.lat,
})
```

4.2.9 Comments

When a user adds a comment to the comment section under an individual event, a comments collection is created, with the document name being the individual event ID. Each comment document is then assigned a random ID, and then the data is set with the text, userName, timeStamp and the commentID.

```
const postComment = () => {

    const docRef = fire.firestore()
    .collection("posts")
    .doc(eventID)
    .collection("comments")
    .doc()

    fire.analytics().logEvent('UserPostedAComment', {


        id: fire.auth().currentUser.uid
      });


    docRef.set({
        text: comment,
        userName: name,
        userID: userID,
        timestamp: new Date(),
        commentID: docRef
    })
    .catch((err) => {
        alert(err)
        console.log(err)
    })

    setComment("");
};
```

4.2.10 Login, Register, Log out

Below are the built in firebase functions that we used for when a user logs in, logouts or registers for our application. For login, we use the built in firebase function *signInWithEmailAndPassword* and pass the user's inputted email and password into it. If it passes then they login, and if it fails then an error message will appear.

For the logout functionality, we simply just call *'this.auth.signOut()'* to logout of the application.

For registering, we take the user's inputted email and password and use the firebase built-in function *'createUserWithEmailAndPassword()'* which creates a user document with the user's name, email and password. We then update the usersProfile with the name value that they entered when signing up.

```
async login({ email, password }) {
    return await this.auth.signInWithEmailAndPassword(email, password)
}

async logout() {
    return await this.auth.signOut()
}

async register({ name, email, password }) {
    const { user } = await this.auth.createUserWithEmailAndPassword(email, password)
    await createUserDocument(user, name)
    await this.auth.currentUser.sendEmailVerification()
    return this.auth.currentUser.updateProfile({
        displayName: name,
    })

}
```

4.2.11 YELP API

The YELP API is used in order to fetch various types of businesses depending on the preference of the user. getYELPData() takes in parameters categories, and the location which is the latitude and longitude. These parameters are then passed to the YELP API and it returns a map of the business data. The returned data is then used for displaying the business information in the frontend and also for matching algorithm purposes. The parameters that should be received are name, image, rating, review count, and location.

```
export function getYELPData({ categoriesAdmin, latitudeValue, longitudeValue}) {

    let listCategories = categoriesAdmin.toString()
    // console.log('sjdjfndsjfnsdjnd   ' + latitudeValue, longitudeValue)

    return axios.get(`https://sage-app-decision.herokuapp.com/https://api.yelp.com/v3/businesses/search?categories=${listCategories}&longitude=${latitudeValue}&latitude=${longitudeValue}&
    limit=20`, config)
        .then(response =>
        response.data.businesses.map(({ id, name, image_url, rating, review_count, location}) => ({
            name: `${name}`,
            imgUrl: `${image_url}`,
            rating: `${rating}`,
            reviewCount: `${review_count}`,
            location: `${location.display_address}`,
        })))
}
```

4.2.12 TMDB API

The TMDB API is used in order to fetch movie data. getMovieData() takes in parameters which are the user's preferences. The movieType and genres are all ensured to be valid values to be passed on to the API. This function then calls skimProfileData() which returns a map of the movie data which is the result based on the user's preferences.

```
export async function getMovieData ({ movieType, genres }) {

    console.log('mov typee  ' + movieType)
    console.log('genereeee   ' + genres)

    const response = await fetch(`https://api.themoviedb.org/3/movie/${movieType}?api_key=${api_key}&include_adult=false&with_genres=${genres}&Page1`);
    const { results } = await response.json();

    return skimProfileData(results);
}
```

The parameters we passed for this function are the ones we needed to display in our frontend and for our matching algorithm which are the name, image, overview, genre, and release date.

```
export function skimProfileData(rawData) {
    return rawData.map(({ original_title, poster_path, overview, genres, release_date,popularity }) => ({
      name: `${original_title}`,
      imgUrl:  `https://image.tmdb.org/t/p/w500${poster_path}`,
      overView: `${overview}`,
      genres: `${genres}`,
      releaseDate: `${release_date}`,
    }));
  }
```

5. Testing

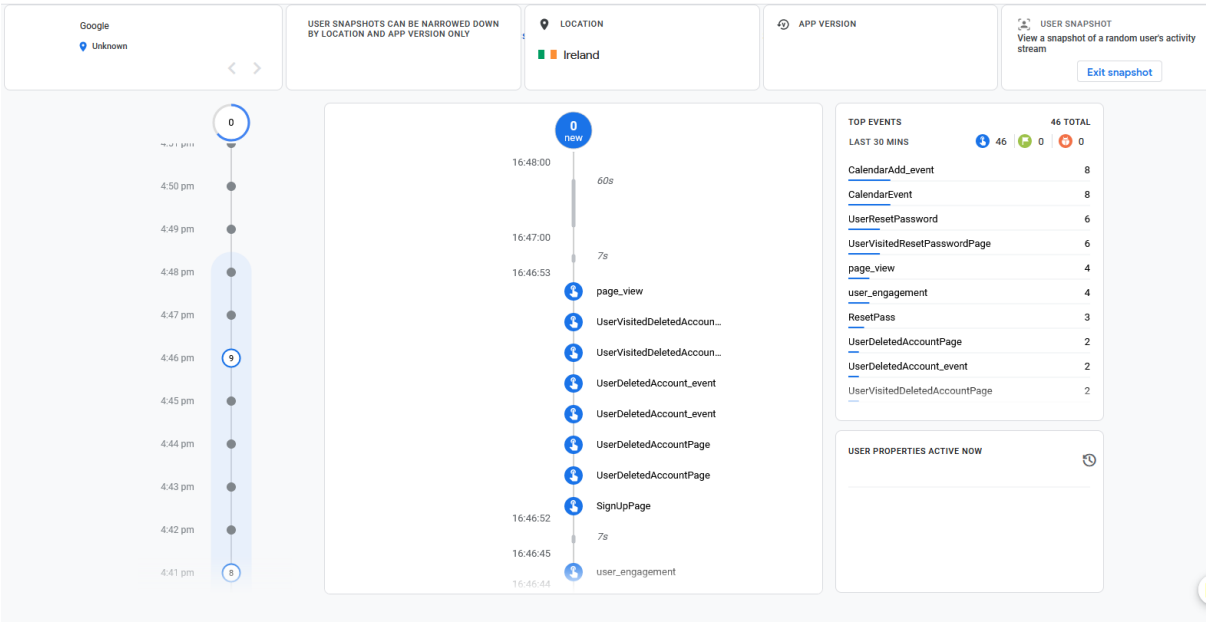
5.1 Firebase Analytics

We implemented firebase analytics into our application, as we felt that it would be highly useful when conducting user testing. The firebase analytics code logs an event

whenever a user visits a page, posts a comment, clicks a button, tries out a certain functionality. Code similar to below is implemented in core functionalities that we wanted to monitor. We set the current screen name of the event , and used *logEvent()* to initialise that an event is occurring for that current user.

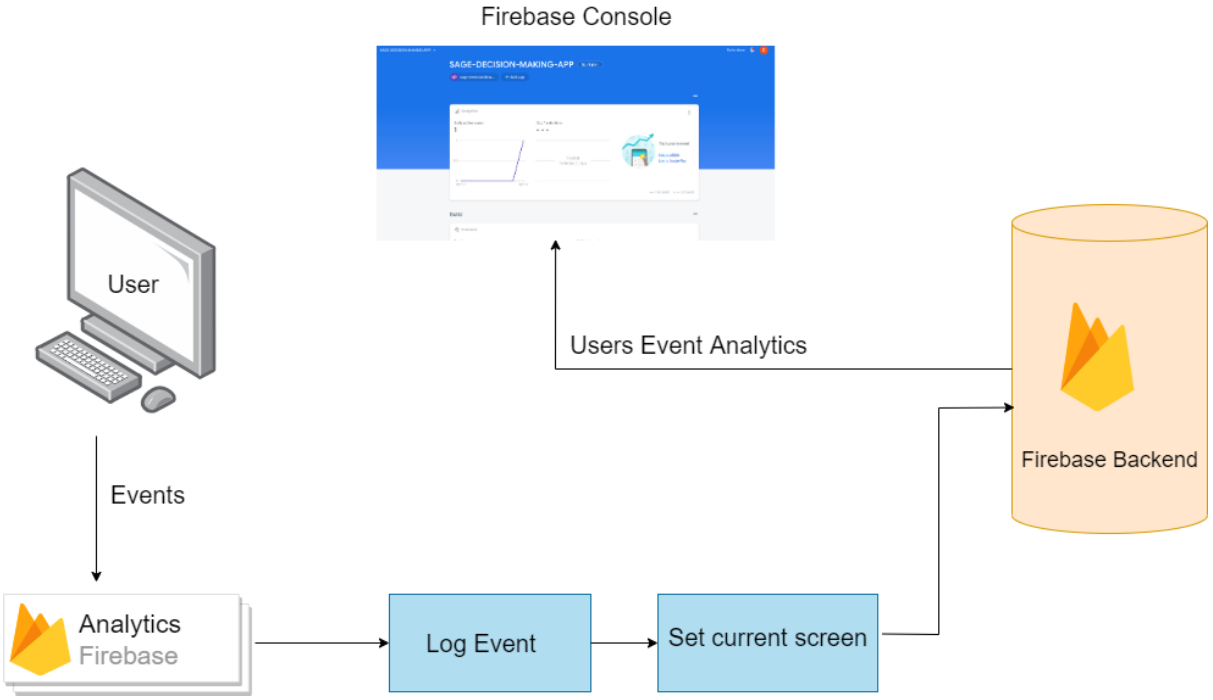
```
if (typeof window !== "undefined") {
    fire.analytics().logEvent('UserVisitedSignUpPage')
  }
fire.analytics().setCurrentScreen('SignUpPage');
fire.analytics().logEvent('UserSignedUp')
```

We can see a random user snapshot, which shows us how a random user is interacting with the application. During user testing this allowed us to be able to watch and see if users were carrying out core functionalities in the application, which showed us which features were being used the most and which features weren't.



Random Firebase Analytics user snapshot

We were also able to see the top events that occurred during the user testing. The events that were rated lowest for user visits allowed us to see the usability of the application, because if a user wasn't navigating to a certain page it led to us improving certain aspects of the pages in terms of buttons etc in case they weren't clear to the users.

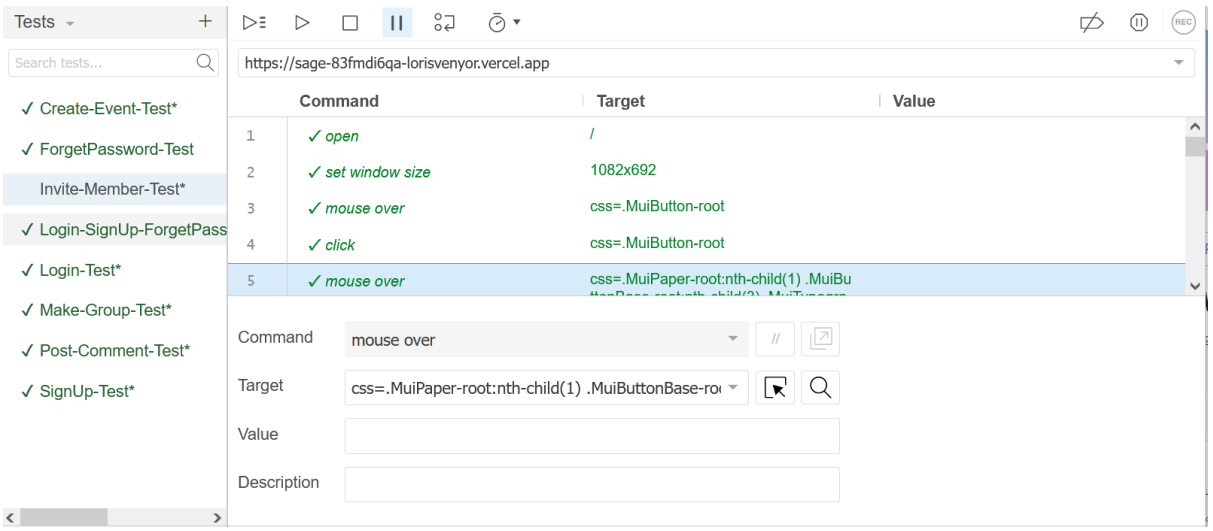


Firebase analytics data flow diagram

5.2 Selenium

We conducted Selenium testing which is an automated testing framework that is used to validate web applications across different browsers and platforms. During selenium testing, we press the record button and it opens a browser with our deployed application. We then navigate throughout the application by pressing buttons and going to pages. Selenium records these actions and then when we finish recording we can press play to see if selenium can recite these actions correctly.

As you can see below our selenium tests passed correctly. We then exported the selenium tests as javascript mocha tests and added them to our tests folder.

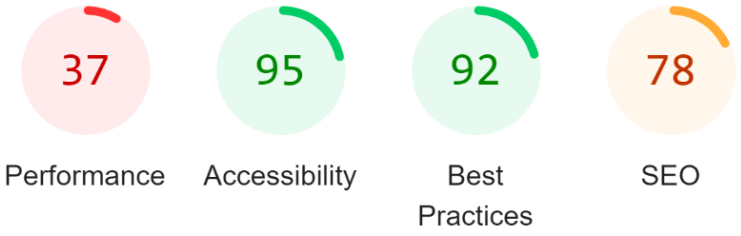


Selenium tests

5.3 Accessibility and Performance Testing

Lighthouse is an open-source, automated tool for improving the quality of web pages. The lighthouse audits that we conducted on our web pages are performance, accessibility, best practices and SEO (Search engine optimization).

We ran a lighthouse performance test for each page of our application, all of our pages had similar results to below, showing that best practices and accessibility were our top rated audits.

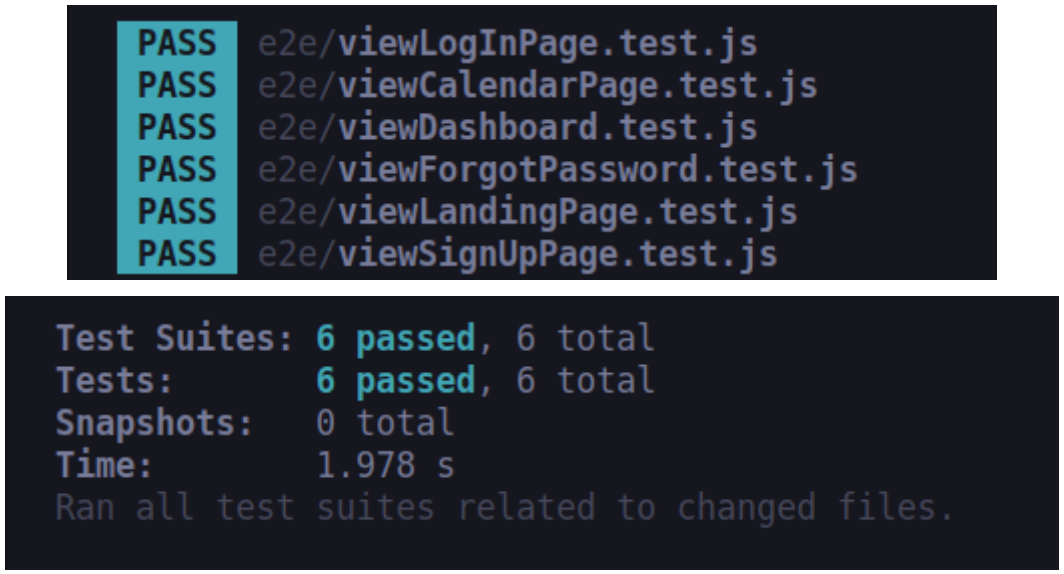


Accessibility and performance test results

5.4 End-to-end Testing

Puppeteer was used for our end-to-end testing. The goal of this test is to simulate what a real user scenario looks like from start to finish. Initially, we were mostly doing manual testing but it was time consuming. We created scripts based on what a user sees when they view various pages in our application such as the login, calendar, dashboard, forgot password, landing, and sign up pages. These tests are not executed as part of our

continuous integration pipeline due to the fact that it can be problematic when the resources are not cleaned up properly so we decided that it is best suited for manual execution.

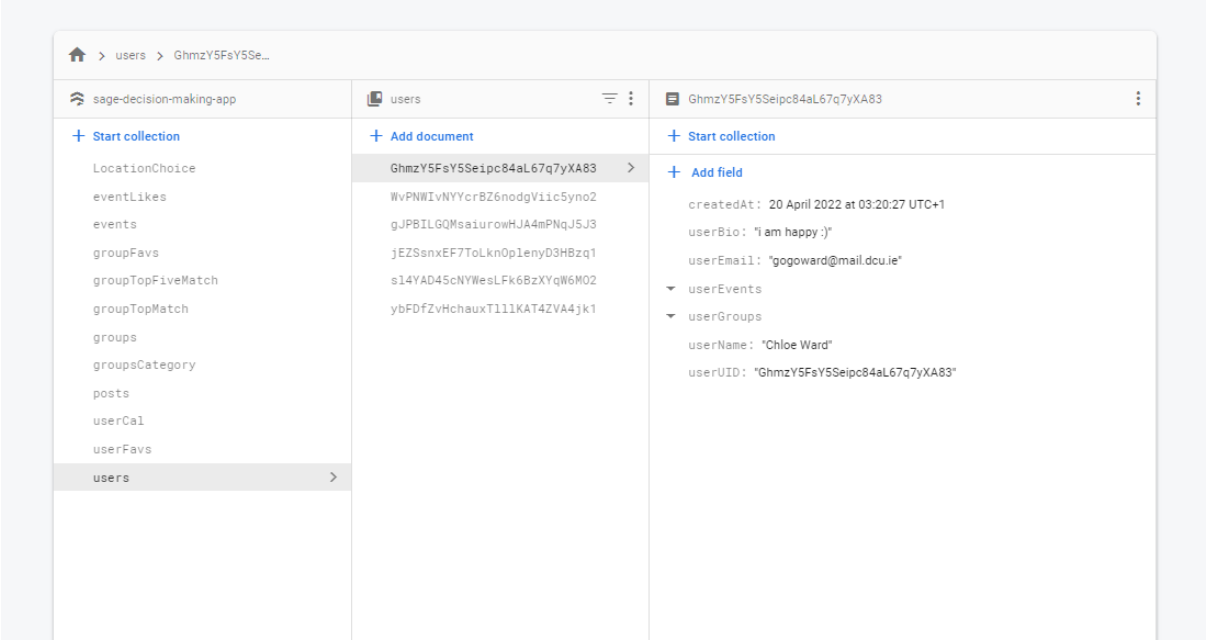


End-to-end tests

5.5 Manual Firebase Testing

For database testing, we decided to follow manual testing which is a test process where you manually test in order to identify bugs. We did this for database testing, as we wanted to make sure that the data was being added to the database in the sequence and format that we wanted. We went through the beginning of the application from signing up all the way to deleting accounts to ensure that the database was behaving the correct way.

An example of our manual testing is shown below, from when we edited our user profile bio.

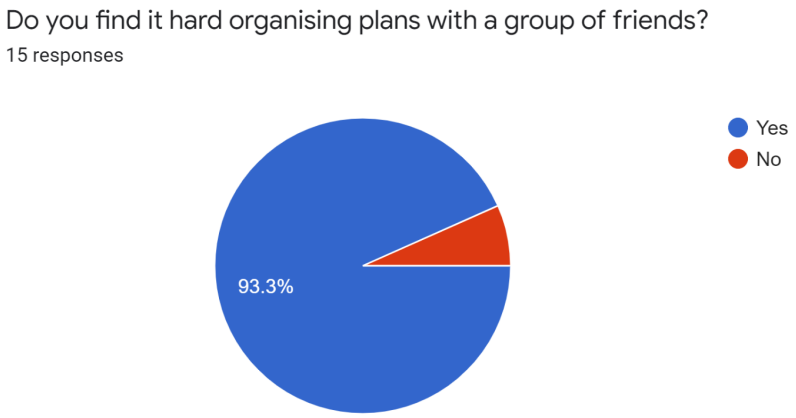


Manual Firebase testing

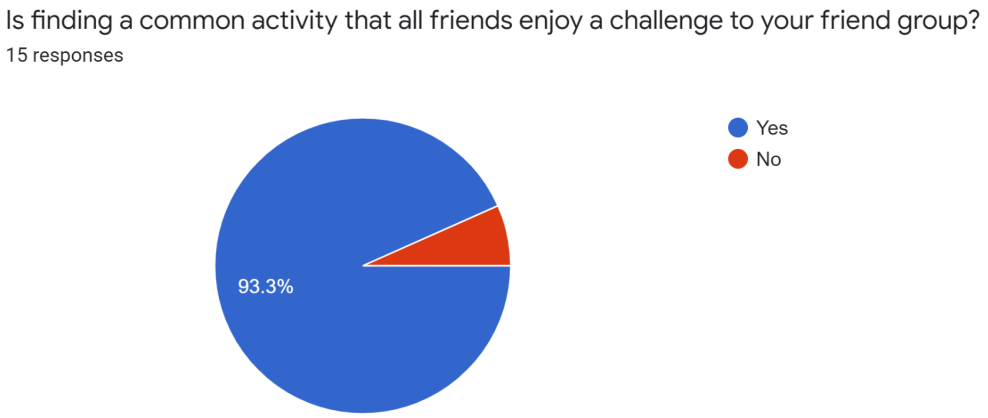
5.6 User Testing

Before beginning the development process, we surveyed 15 people via google forms to see if there was a market for an application like SAGE. This survey ultimately allowed us to see the potential target audience and success for an application like SAGE before we embarked on our SAGE Development survey.

We asked six questions in the predevelopment survey, which included questions such as:



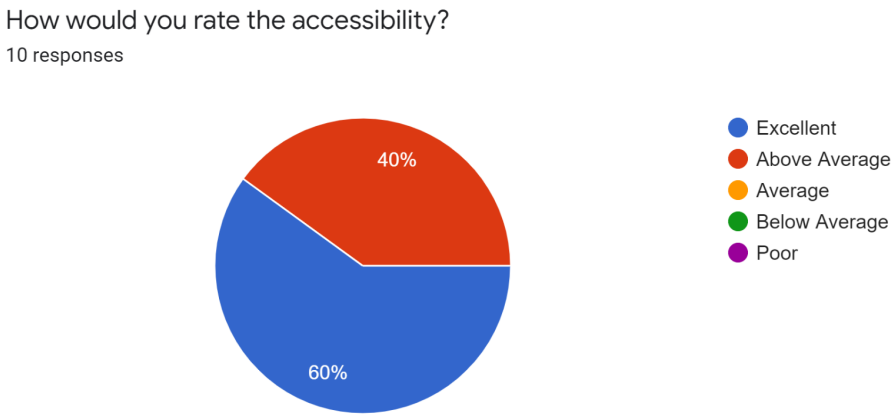
Predevelopment survey question



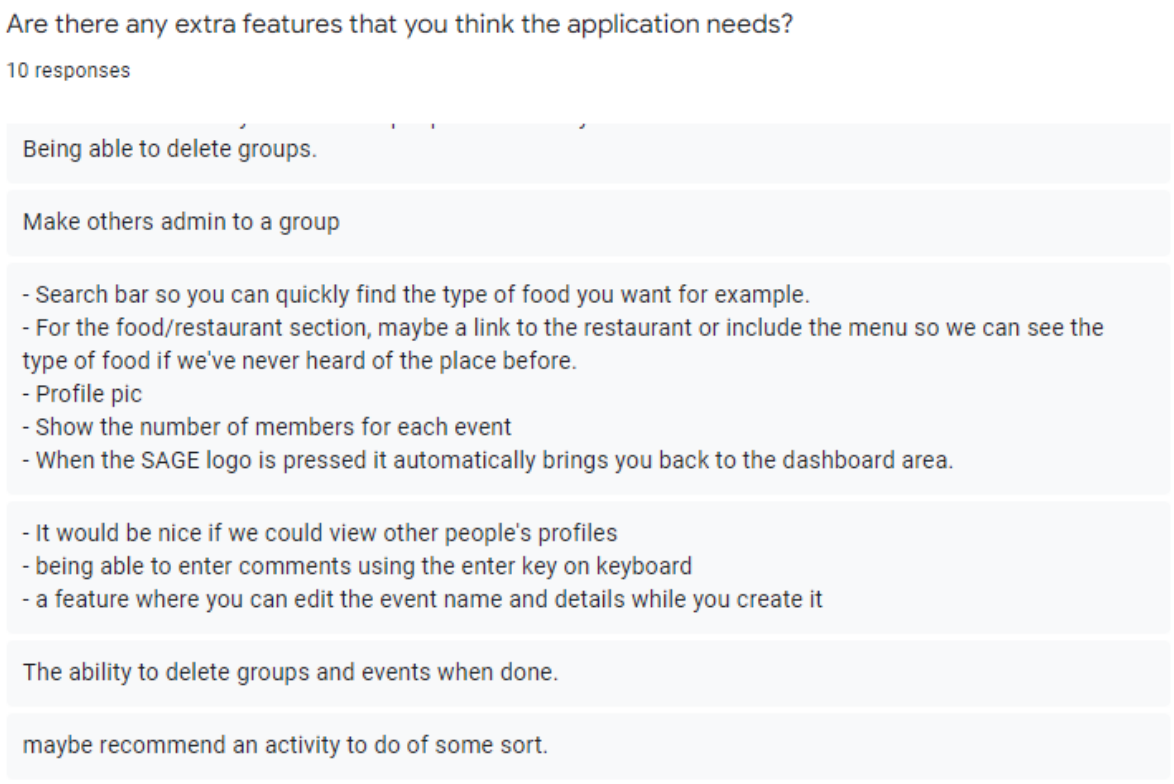
Predevelopment survey question

We asked these questions to see if our application would benefit the target audience that we were catering our application for, and if they encounter this struggle throughout their daily lives.

Once the development of SAGE came to an end, we conducted user testing. We also asked people who took part in our previous pre development survey to take part in the user testing. Before user testing was conducted, we asked the users to sign a plain language statement and do a post user testing survey. Below are some questions that we asked the users: Out of 15 people who did the first survey, 10 people took part in the user testing. Here we got users to navigate throughout the application with a set of steps that we provided them with. We conducted the user testing remotely, so we deployed the application online, where users could then access it from their browsers. This form of testing allowed us to gain an understanding into our users needs and what could be improved in the application to increase the users overall experience.



User testing survey question and response

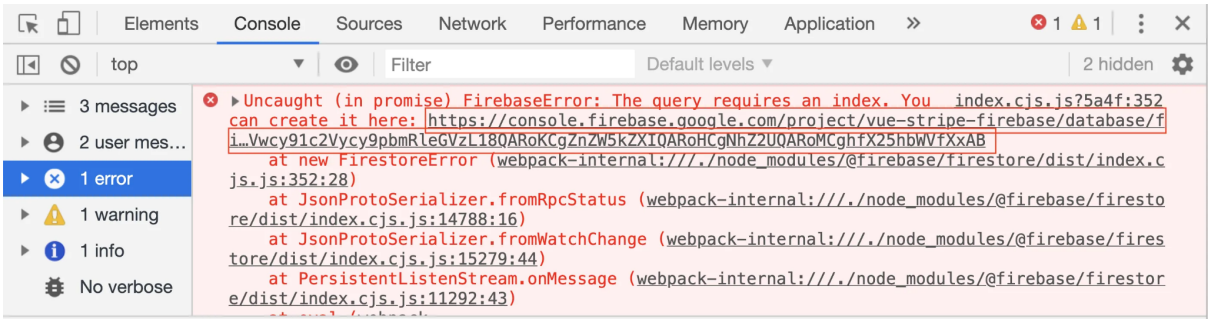


User testing survey question and response

6. Problems and Solutions

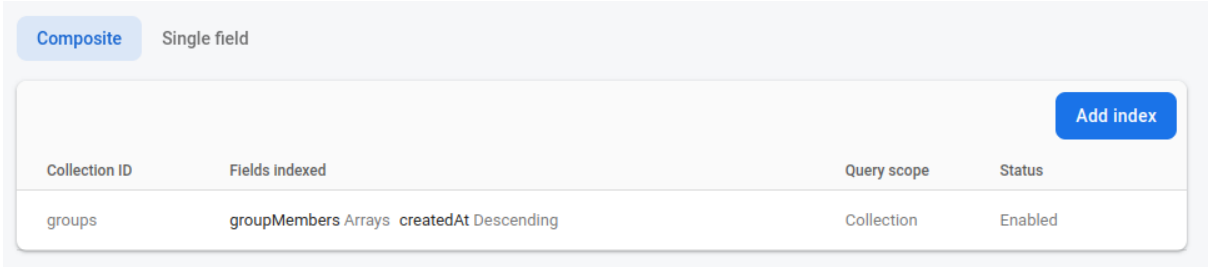
6.1 Indexing Issue

Problem: The groups page displays all the groups that the current user is a member of. Initially, the groups were randomly ordered which can be confusing to the users. Firebase has a function called orderBy() which sorts the data you wish to be sorted. This function was easily implemented by adding one line of code. We thought it would work; however, we encountered an indexing problem. The groups were also not displaying how we thought it would work.



Screenshot of the Indexing issue

Solution: Firebase requires to create an index for this order of documents and this can be done in the Firebase console. We added the collection ID and the fields that needed to be indexed and how we wanted it to be sorted. This process took quite a while before it was enabled and ready to be used.



Firebase indexing issue solution

6.2 Swipe Cards

Problem: We were using a react-tinder-card node package module for the tinder swipe functionality, it was behaving correctly for swiping it with the mouse, but when we tried to implement the button functionality, it would not work well with the API, as getting the swipe cards to move alongside the api with a node package module that didn't have much detail on how the swipe cards functionality worked, we got it half working but it would lag out due to the API and cause errors as it could not fetch the cards in time with the buttons due to the react-tinder-card node package module. We were spending too much time on it so we had to find a new solution as we need button functionality in order to cater for user accessibility.

Solution: We decided to implement our own swipe functionality from scratch, as we wanted to have a deep understanding of how the swipe functionality was working. So we implemented multiple functions, starting from small and building our way up to a fully functional swipe card functionality which allowed us to correctly navigate the cards with buttons.

6.2 CORS Issue

Problem: YELP-API does not support CORS Policy. Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.

Solution: The backend implementation includes a server which is hosted on Heroku to bypass the CORS policy for the YELP API which adds CORS headers to the proxied request and this allows us to call the YELP API without any CORS errors.

6.3 Google Maps

Problem: We originally wanted to use google maps for our map functionality, there was a lot of node package modules, which did exactly what we wanted it to do, but when we entered our API Key, it came up with a big watermark which said “For development purposes only” so after researching we learned that google maps is free for mobile development, and the only way we could use it for web development was to enter our card details in on the website and get free credit but after that we would be charged.

Solution: We decided to look for a free interactive map, so after heavy research we came across Leaflet which is an open source JavaScript library for interactive maps. We then implemented the map with the drag and drop markers with the Leaflet interactive map. It was quite difficult as not many tutorials were online, so it was a real learning experience, it didn’t look as good as google maps but using another interactive map api was a real learning experience as we had to figure out how to implement our own drag and drop markers compared to how google map node package modules would already implement that for us.

7. Future Work

7.1 Runtime

Improve the runtime performance of the application, as the lighthouse tests showed us that the runtime of it was quite low compared to the other audits it completed. During runtime with the user swipe cards when the API data is loading, it’ll take approximately 5 seconds for the data to show up. If we had more time we would fix this to allow the runtime performance to be better.

7.2 Recommender System

Because we were new to using Next.js and learning framework, we spent too much time creating the swipe cards functionality and figuring out how to design our database correctly to enable functionalities such as inviting users. We feel that we could have implemented a recommender system into the application that after each time a user does a survey, it would recommend movies or restaurants based on the groups matched events history. We believe that this functionality would help improve user experience and would be a great feature in our application.

7.3 More Tests and Research

Testing is an important part of development which is one of the elements that validates the performance of an application. We would delve more into testing our

components in the future to further improve the quality of our application. We would also investigate more in relation to our application to understand which features would be best to be implemented.

7.4 Integrate Weather API

We wanted to implement the weather API into our application by using the user's coordinates for latitude and longitude that they choose with the map functionality to get the weather for that certain date or to suggest activities based on the current weather type. We had the weather API results ready alongside with the latitude and longitude implementation but we unfortunately ran out of time to implement that feature.

7.5 In-app Chat Feature

Although we have a comment section which serves as a mode of communication for our users in their group events, it would be a great addition to the application to have an in-app chat feature. Having this feature would further improve user experience and interaction in our application and would save a lot of time planning events.

7.6 Notifications

Providing notifications to the users would be a great feature to be implemented in the future. It can be implemented in a way that the users would be notified when an event is about to come up, or when a group member leaves a comment in the event section, or when they have to do the matching.