DM8001
Web Design Assignment

Table of Contents

Table of Figures

Planning & Research

The main aim of this website is for people of the construction industry to connect with their peers and communicate freely on this platform. They can share your ideas, their thoughts or whatever they want on this social media platform. Included in this website is the comment and like functions as well. They can comment and like other user's post and this would generate interaction and communication.

This website is inspired by Twitter, hence, many of the design elements and functions are similar to the Twitter platform.

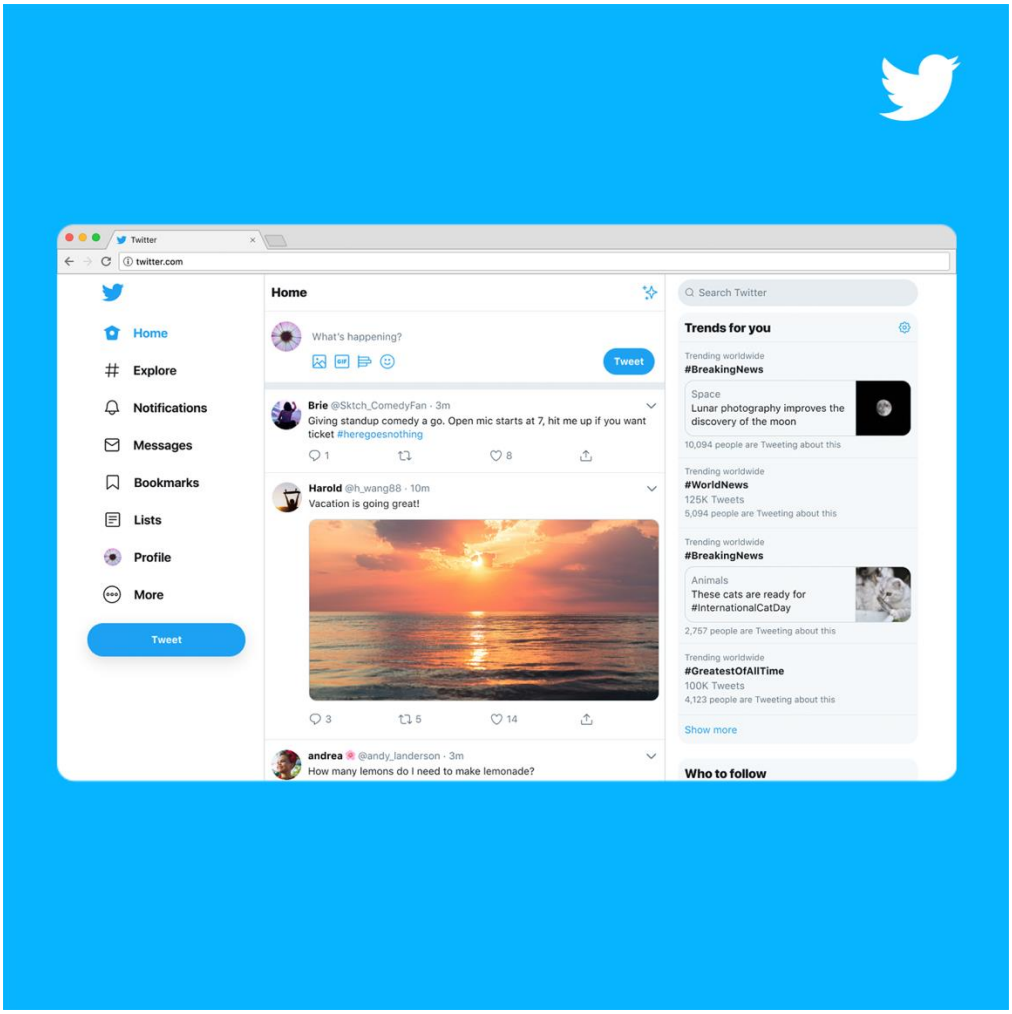


Figure 1:Twitter UI

Target Audience

The main target audience for this platform is for the people that are in the construction industry although others are also free to join. The construction industry has been heavily battered by COVID-19 and one of the main reasons is due to them not innovating in enough. This platform hopes to connect all of them together and generate new ideas which could bring the construction industry forward.

Design

I will be using Material Design for the design of this platform. Material is a design system created by Google to help teams build high-quality digital experiences for Android, iOS, Flutter, and the web.

With the help of Figma's material design kit, I can quickly spin up my designs and prototype in a material design theme.

Wireframes & Prototypes

I will be using Figma for the prototyping of this platform.

Here is the link to my prototyping which is in the starting phase.

Design Link: https://www.figma.com/file/WDP6EFLQXvaIP32LbKFZ72/DM8001-Web-Design?node-id=0%3A1

Prototype Link: https://www.figma.com/proto/WDP6EFLQXvaIP32LbKFZ72/DM8001-Web-Design?node-id=205%3A1581&scaling=min-zoom&page-id=0%3A1

Self Evaluation

Lessons learned

I have gained an in-depth knowledge of using Cascading Style Sheets (CSS) as well as using JavaScript to improve the user interface of the website.

Throughout the web design lessons as well as the project, I have learned valuable web programming skills. Before this course started, I was familiar with simple web programming and was stronger with handling the backend of a website. This was because my course, Information engineering, and Media (IEM), was more of an engineering school rather than a design school. Hence, I was familiar with handling databases and creating application programming interfaces (APIs).

However, my front-end was weak as I was not trained in design. I knew about CSS but did not know how to create a beautiful design out it. The lessons from this course and online articles taught me the art of design in web development. With this, my front-end designing skills have improved tremendously.

The lessons provided by professor Sven were pleasant as it was more hands on rather than just watching coding lectures. Learning while coding is one of the most effective ways to learn.

Improvements

There are many areas that I can improve on in terms of web programming. Firstly, would be my CSS skills, I still have not fully grasped the full potential of CSS as I still mainly use CSS libraries that provide components that you can customize. There is also much more CSS stuff that is yet to be learned, such as animations which can be used in conjunction with JavaScript.

Secondly, would be my eye for design. I hope that I can improve on my design skills which are subpar compared to the other art students in class. Even though there are Design Systems and libraries that I can utilize to create good designs; I still want to reach a point where I can create my design and style.

Victories

Learning proper design and CSS would be considered a victory for me. Design is hard to learn and it is harder to teach, hence, I would consider it a victory.

Creating a full-stack website from scratch would also be considered a victory for me. Both the front-end and back-end of the website was tough to make. However, I managed to complete it in the end and learned a lot through the process. I would consider this a victory.

Failures

There are no failures from me. But if I had to point out one point where I wished that I could have done better, it would have been on the design of the website as well as adding more functionality to it. The design of the website could have been better especially the part on the media queries and size adjustments. I also wished that I could have added more animations to the website as well.

Future goals

In the future, I hope that my website design skills as well as my programming skills will improve. Taking online courses and watching YouTube videos have really helped me in this process. Once, I am finished with web programming, I may transition to mobile development. It seems like mobile apps are now becoming increasingly popular, and I hope that the school would one day create a module or a lesson on that.

Design - Visual Aesthetics

Main Theme

Prototype 1: https://www.figma.com/file/WDP6EFLQXvaIP32LbKFZ72/DM8001-Web-Design?node-id=0%3A1

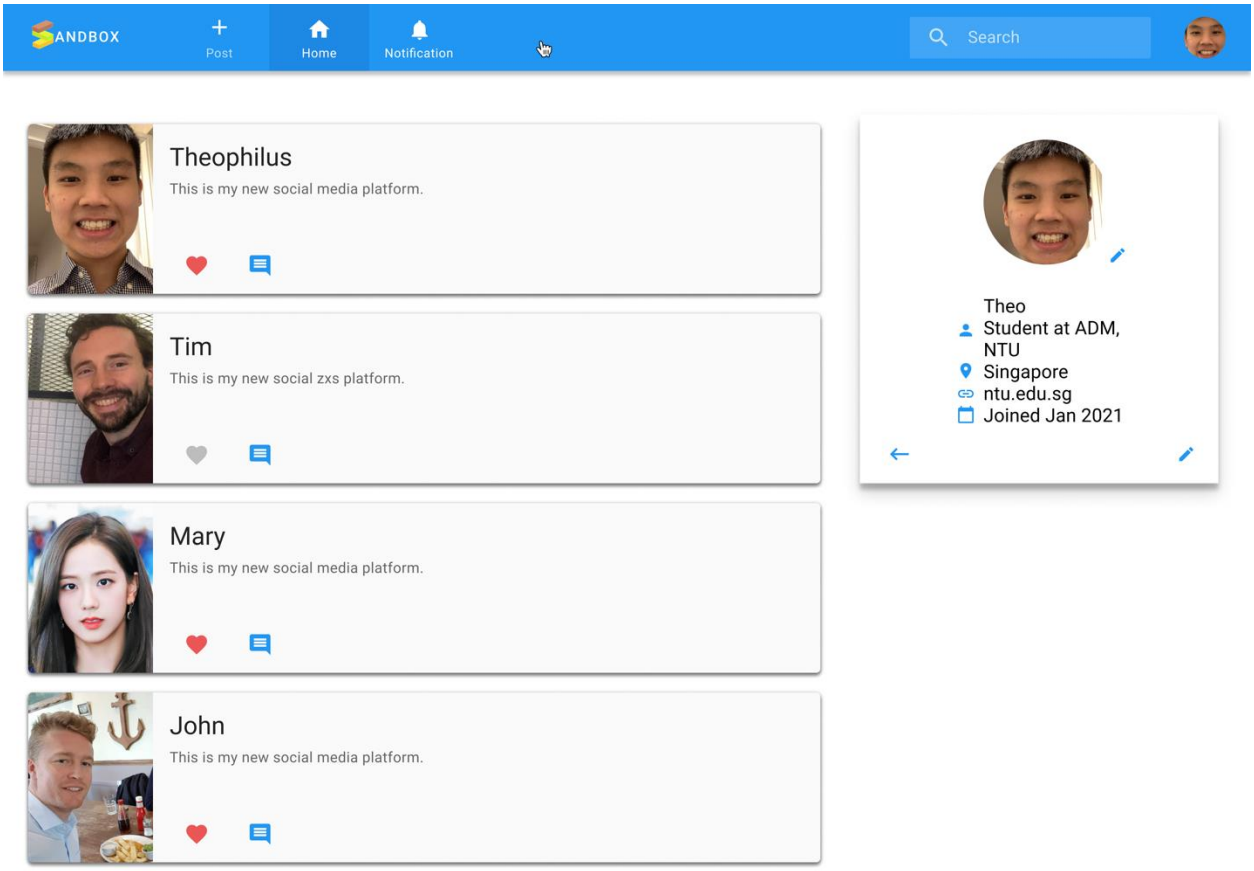


Figure 2: Main theme

Construction Green Theme

Prototype 2: https://www.figma.com/file/WDP6EFLQXvaIP32LbKFZ72/DM8001-Web-Design?node-id=215%3A0

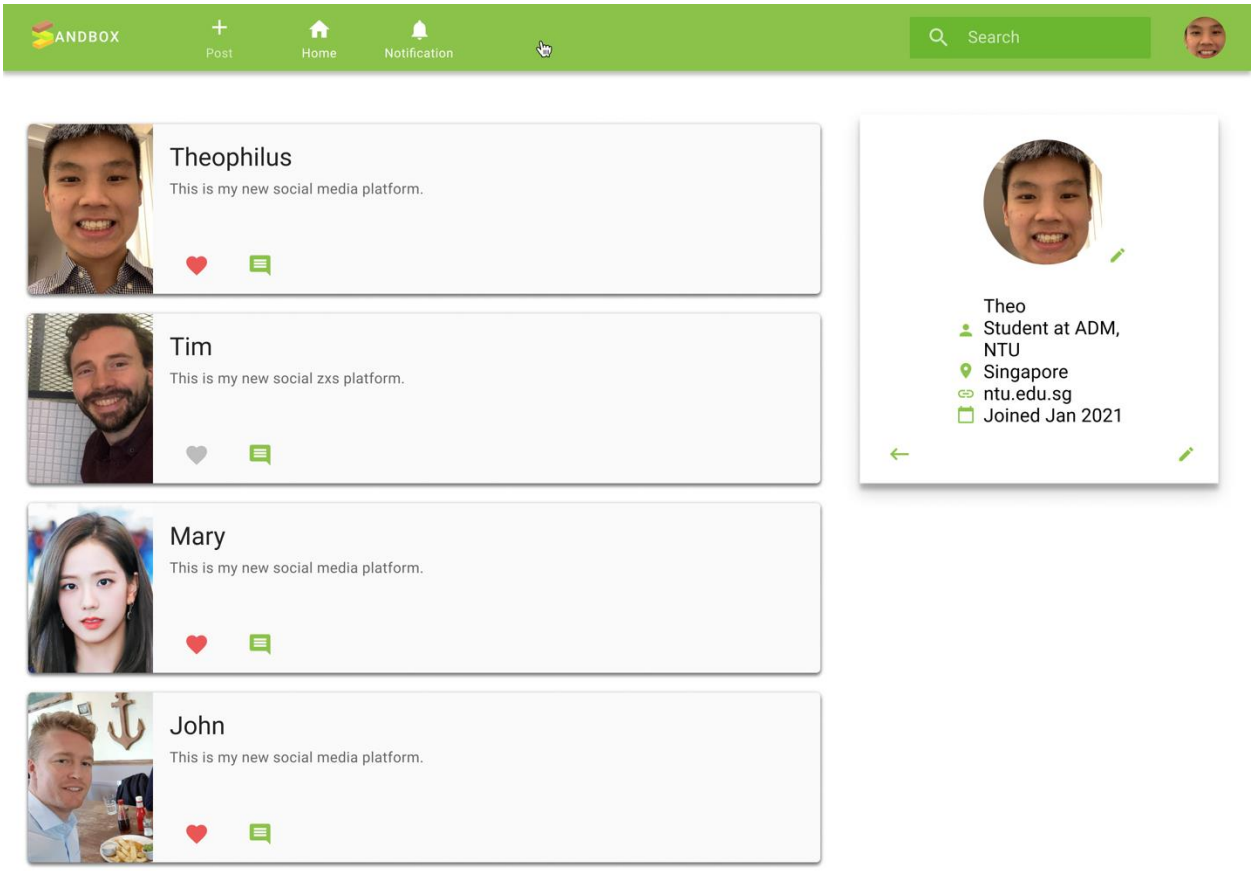


Figure 3: Construction Green Theme

Dark Mode

Prototype 3: https://www.figma.com/file/WDP6EFLQXvaIP32LbKFZ72/DM8001-Web-Design?node-id=220%3A3687

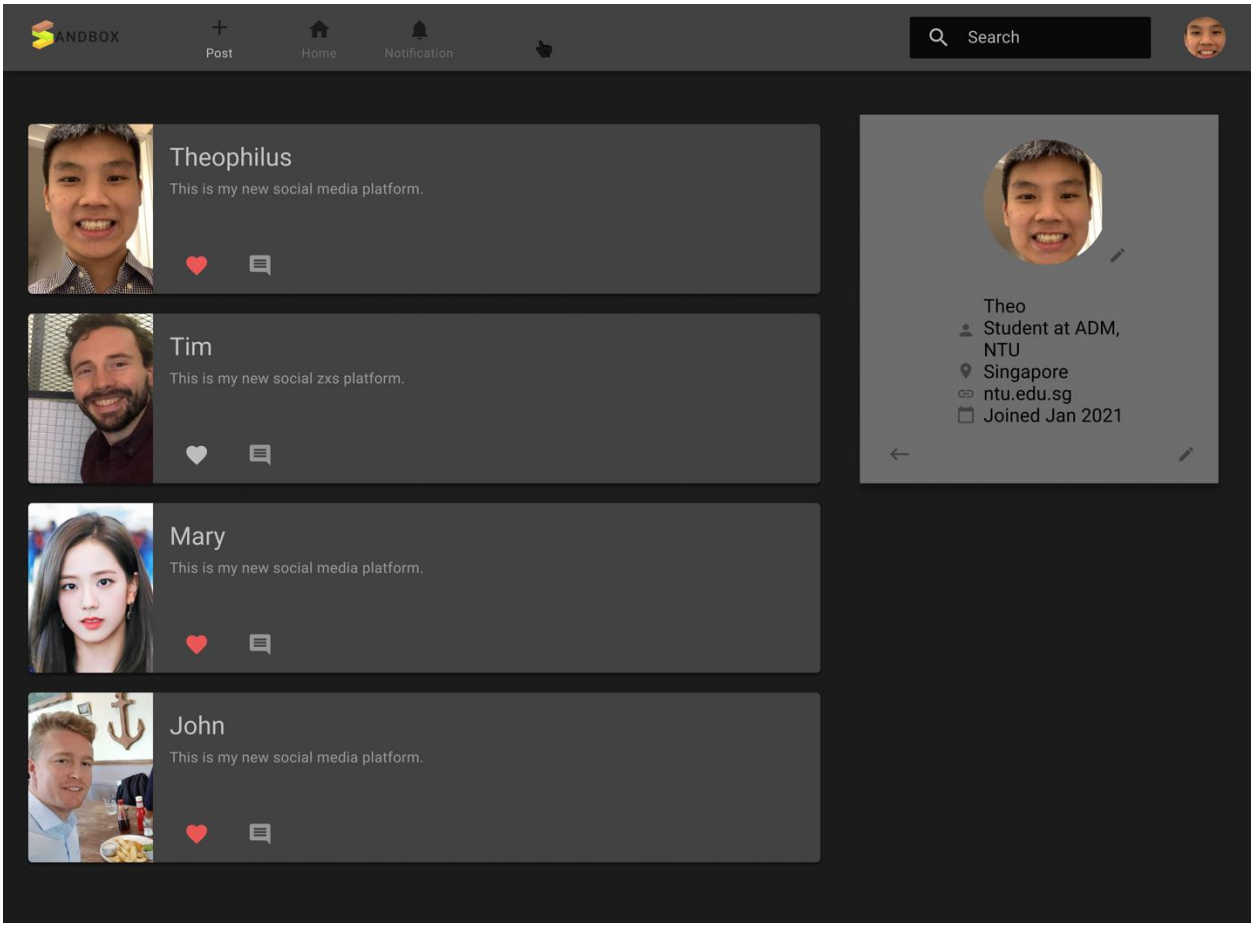


Figure 4: Dark Mode

Development

Chapter 1:

Software Architecture Design

This web application will be used by many users; hence, it must be scalable and able to handle a large amount of data. According to present-day website development architecture, the three main components of a website are the front-end, back-end, and database. The figure below illustrates the software architecture of this social media platform.

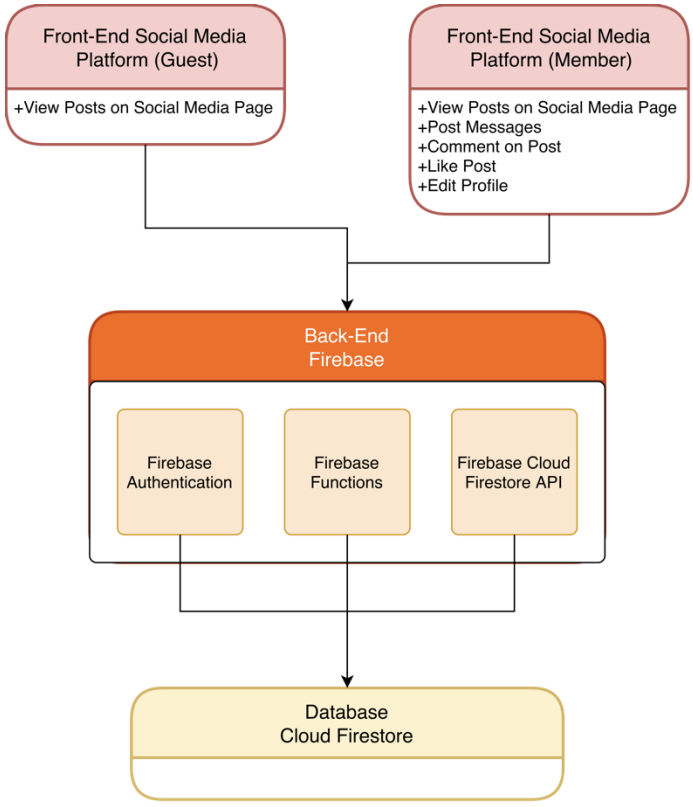


Figure 5:Sandbox Architectural Design

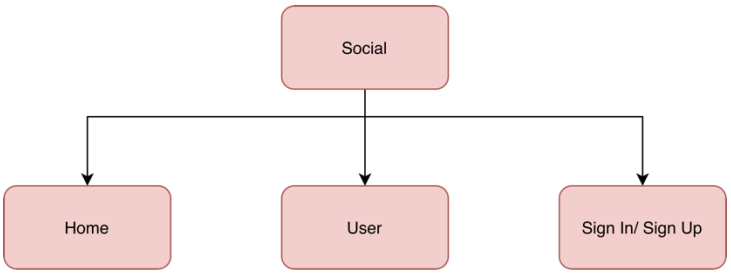


Figure 6:Sandbox Sitemap Design

Front-end of Sandbox Social Platform

The front-end of the social platform, is created using React.js. Since React.js is a JavaScript library, we will be using Facebook's Create React App (CRA) as the framework to build this platform up. The design of the main platform is done using Material UI, which is a React UI framework. These are the key pages for the social platform:

- Home
  - Post
  - Comment
  - Profile
  - Notifications
- User
- Sign In/ Sign Up

1.1.1 Home Page

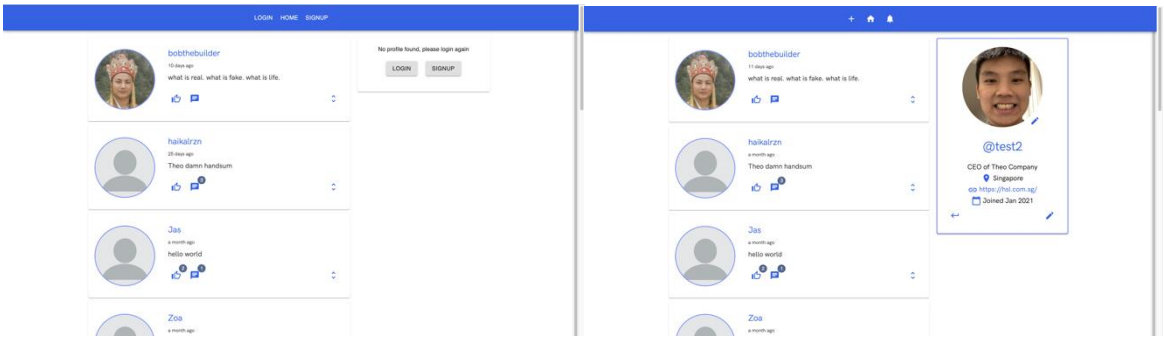


Figure 7: Home page not logged in and logged in

The figure above shows the home page. It lists all the post made by registered users on the platform. Registered users will have their profile status placed on the right column of the page. Registered users will be able to like and comment and other user's post.

1.1.1.1 Post Component

Figure 8: A post made by test2 with 1 likes and 4 comments

The figure above shows a post. It contains the profile picture, name, and message of the sender. It will also show the time difference between the current time and the time of the creation for the post. It will contain a like button and a comment button. The number beside the like button would show the number of likes received and the same goes for the comments.

There will be a delete button at the top right of your post. This gives you an option to delete your post in case of an error or mistake.

Figure 9: + icon in the navigation bar

To post a message, click the + icon on the navigation bar at the top of the screen.

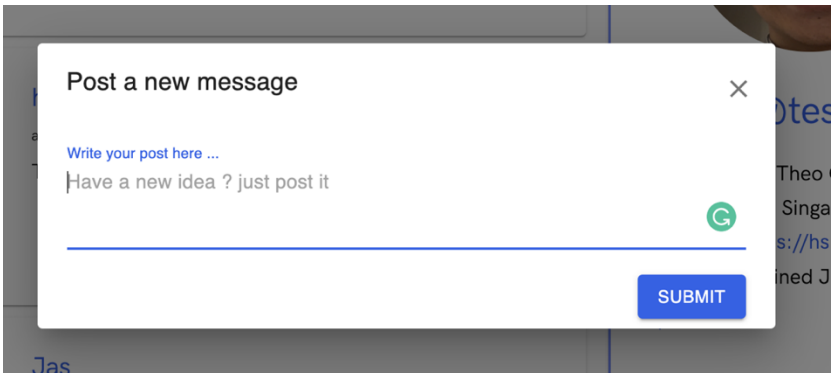


Figure 10: Post modal

A modal will appear when you click the + icon. This allows you to post your message onto the platform.

1.1.1.2 Comment Component

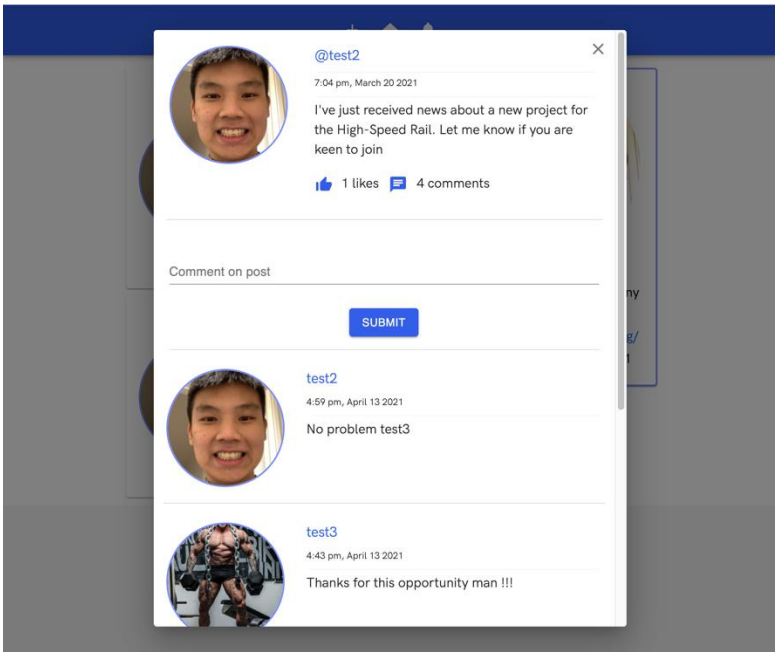


Figure 11: Full thread for post made by test2

Comments will be shown when you click on the expand button on the bottom right button. It will show all the comments related to this post. A user comment will contain the user's profile picture, name, message, and date of creation.

To post a comment, write your comment at the comment line notated by the grey font “Comment on the post” and click the submit button.

1.1.1.3 Profile Component

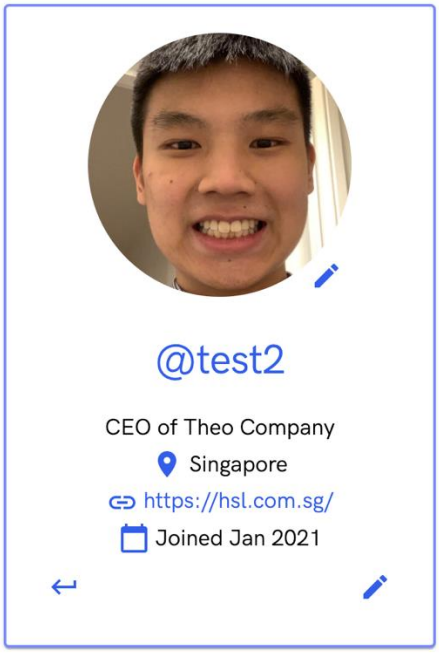


Figure 12: Profile details of test2

The figure above shows the bio of test 2. It contains the name, bio, location, website, and date of creation for the account. At the top, it is an image of the user and it is editable as well by clicking the pen icon on the right side of the profile picture.

The details of the user can also be changed by clicking the pen icon at the bottom right side of the card component. To log out of your account, you can click the go back icon at the bottom left side of the card component.

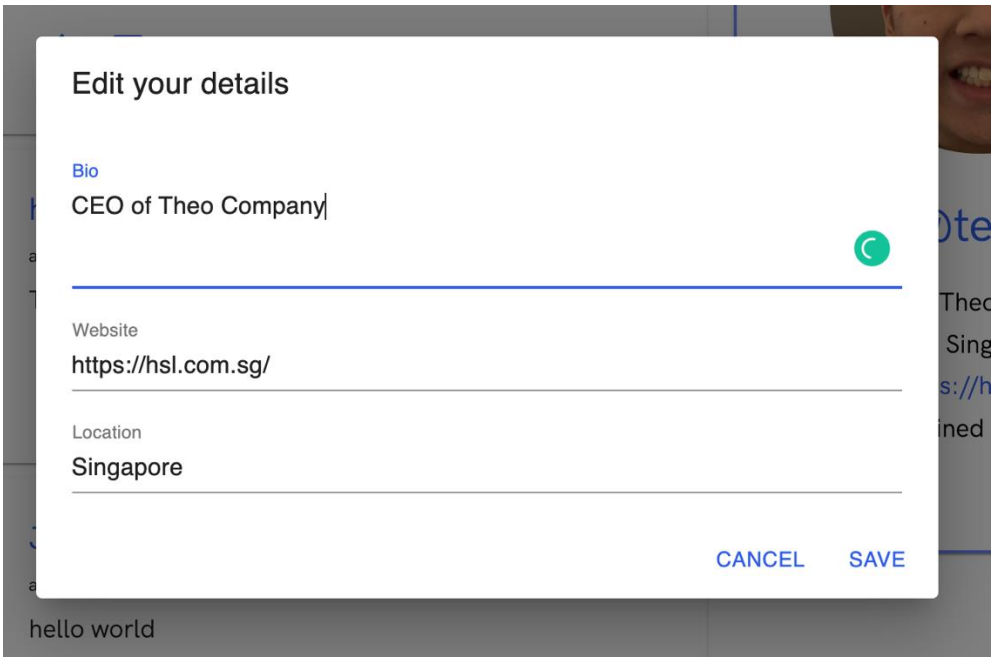


Figure 13: Editing form for details of test2

The figure above shows the profile details editing modal. It will appear when the profile details edit icon is clicked. Here, you can edit your bio, website, and location.

1.1.1.4 Notification Component

Figure 14: Notification bell icon

The figure above shows the navigation bar which contains the bell icon. The bell icon represents the notification for the user. A red badge will appear if there are any notifications. Upon clicking the icon, a popover will appear that lists the activities surrounding your account.

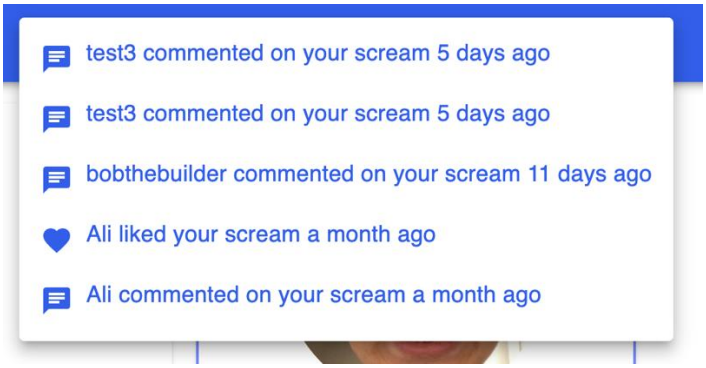


Figure 15: Notification menu

The figure above shows the list of notifications for the current user. It will show the activity such as liking or commenting the user who liked or commented, and the time difference of the activity.

1.1.2 User Page

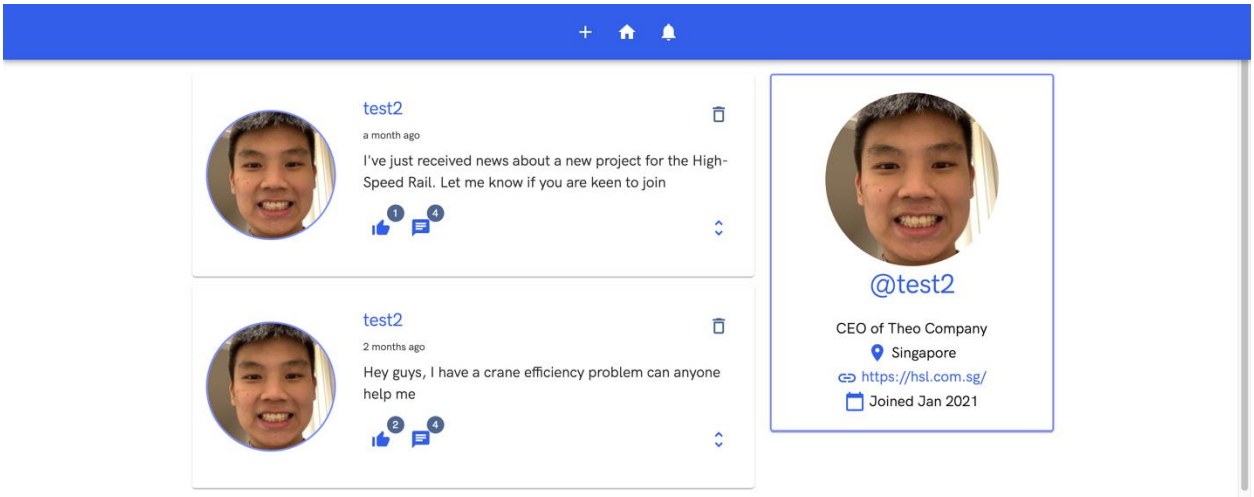


Figure 16: User page for test2

The figure above shows the user page for the test2. It lists all the posts that the user test2 has made. The user can review the post that he has made and go through the message thread for some of them. Having a centralized user's page would be helpful to the user in situations where there are many posts, and the message threads are long.

1.1.3 Sign In/Sign Up Page

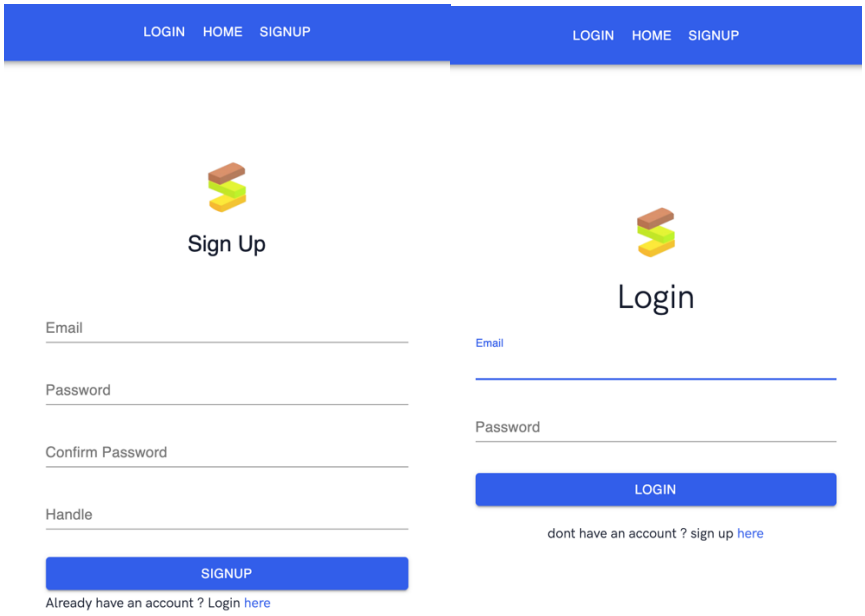


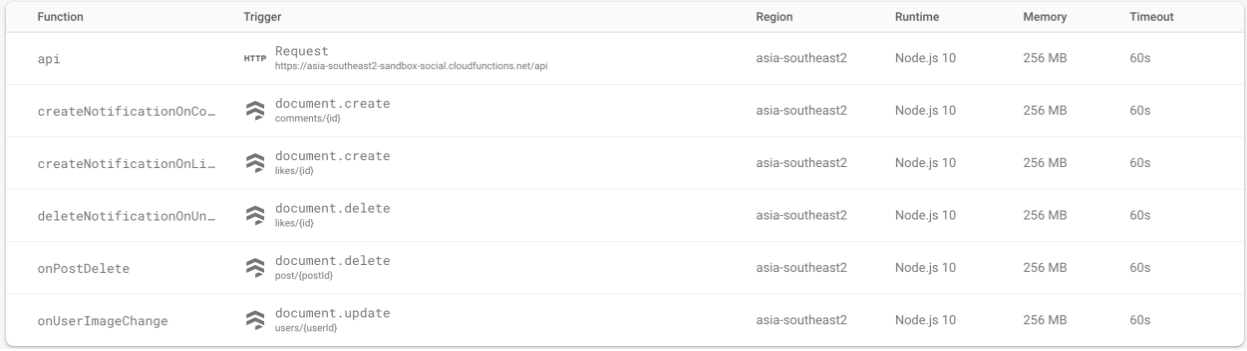
Figure 17: Sign Up and Log In page for Social platform

The figure above shows the Sign-Up and login page for the Social platform. New users can register on the platform by entering their email address, password, and name/handle. Registered users can go to the login in page and enter their email and password, this will bring them to the home page and they will be able to see the message feed and their profile at the same time.

Backend of Sandbox Social Platform

1.2.1 Cloud Functions

The social application of Sandbox is run Firebase Cloud functions. This allows me to run my backend code on Firebase servers and automatically respond to any events that are triggered by Firebase features or HTTPS requests. The code is written in JavaScript and is run on a Node.js environment.



Function	Trigger	Region	Runtime	Memory	Timeout
api	HTTP Request https://asia-southeast2-sandbox-social.cloudfunctions.net/api	asia-southeast2	Node.js 10	256 MB	60s
createNotificationOnCo…	document.create comments/{id}	asia-southeast2	Node.js 10	256 MB	60s
createNotificationOnLi…	document.create likes/{id}	asia-southeast2	Node.js 10	256 MB	60s
deleteNotificationOnUn…	document.delete likes/{id}	asia-southeast2	Node.js 10	256 MB	60s
onPostDelete	document.delete post/{postId}	asia-southeast2	Node.js 10	256 MB	60s
onUserImageChange	document.update users/{userId}	asia-southeast2	Node.js 10	256 MB	60s

Figure 18: Overview of Firebase cloud functions for social application

From the image above, you can see that six cloud functions have been created to serve the social application of Sandbox. The main API is named “API” and it is triggered by HTTPS requests that are connected to https://asia-southeast2-sandbox-social.cloudfunctions.net/api. The other five APIs are createNotificationOnComment, createNotificationOnLike, deleteNotificationOnUnLike, onPostDelete and onUserImageChange. These five APIs are mainly to connect to the Cloud Firestore database.

From here you can see the region that these APIs are connected to, all APIs are located at Asia-southeast2 which is located at Jakarta, Indonesia, APAC.

The runtime used is Node.js 10, memory allocated is 256 MB and the timeout is set to 10 seconds. All these settings have been placed because it is the most cost-efficient.

Database of Sandbox Social Platform

1.3.1 User Schema

When a user creates an account, only the createdAt, email, handle, imageUrl, and userId are created. The imageUrl created is a temporary placeholder that the user can change when they edit their profile picture. The bio, location, and website of the user can be added later when the user decides to edit their profile.

```
{
  "users": {
```

```
    "bio": { "Type": "string" },
    "createdAt": { "Type": "string" },
    "email": { "Type": "string" },
    "handle": { "Type": "string" },
    "imageUrl": { "Type": "string" },
    "userId": { "Type": "string" },
    "location": { "Type": "string" },
    "website": { "Type": "string" }
  }
}
```

1.3.2 Post Schema

When a user creates a post, it will record the message body, the time of creation, the user's name/handle, and image. The starting like count and comment count will be 0. It will be updated when another user likes, unlike, and comment on the post.

```
{
  "post": {
    "body": { "Type": "string" },
    "commentCount": { "Type": "number" },
    "createdAt": { "Type": "string" },
    "likeCount": { "Type": "number" },
    "userHandle": { "Type": "string" },
    "userImage": { "Type": "string" }
  }
}
```

1.3.3 Comment Schema

When a user creates a comment, it records the comment body, the time of creation, the post identification (ID) of the post to which the comment is replying, the user's name/handle, and image.


```
{
  "comments": {
    "body": { "Type": "string" },
    "createdAt": { "Type": "string" },
    "postId": { "Type": "string" },
    "userHandle": { "Type": "string" },
    "userImage": { "Type": "string" }
  }
}
```


1.3.4 Notification Schema

A notification is created when another user likes, unlike, or comments on your post. It records the time of creation, the post ID of the activity, the user's name, the sender's name, whether it is a comment-type activity or a like-type activity, and whether the notification has been read or not.


```
{
  "notification": {
    "createdAt": { "Type": "string" },
    "postId": { "Type": "string" },
    "read": { "Type": "boolean" },
```

```
    "recipient": { "Type": "string" },
    "sender": { "Type": "string" },
    "type": { "Type": "string" }
  }
}
```


1.3.5 Like Schema

Liking a post records the name of the user who likes the post and the post ID itself.


```
{
  "likes": {
    "postId": { "Type": "string" },
    "UserHandle": { "Type": "string" }
  }
}
```

Chapter 2:

Implementation

These are the dependencies that were used for the Front-end of sandbox Social Platform:

```
"dependencies": {
  "@material-ui/core": "^4.11.2",
  "@material-ui/icons": "^4.11.2",
  "@testing-library/jest-dom": "^5.11.8",
  "@testing-library/react": "^11.2.2",
  "@testing-library/user-event": "^12.6.0",
  "axios": "^0.21.1",
  "dayjs": "^1.10.3",
  "jwt-decode": "^3.1.2",
  "react": "^17.0.1",
  "react-dom": "^17.0.1",
  "react-redux": "^7.2.2",
  "react-router": "^5.2.0",
  "react-router-dom": "^5.2.0",
  "react-scripts": "4.0.1",
  "redux": "^4.0.5",
  "redux-thunk": "^2.3.0",
  "web-vitals": "^0.2.4"
}
```

Figure 19: Dependencies for front-end

These are the dependencies that were used for the Back-end of sandbox Social Platform:

```
"dependencies": {
  "busboy": "^0.3.1",
  "cors": "^2.8.5",
  "express": "^4.17.1",
  "firebase": "^8.2.3",
  "firebase-admin": "^9.4.2",
  "firebase-functions": "^3.13.1",
  "uuidv4": "^6.2.6"
},
```

Figure 20: Dependencies for backend

Important Libraries used:

Library	Description
axios	Promise based HTTP client for the browser and node.js
firebase	Firebase SDK
react	JavaScript library for building UI components
React-router-dom	Provide Dynamic Routing
dayjs	JavaScript library that manipulates dates
jwt-decode	Library that helps with decoding JWT tokens
redux	State management
Reduce-thunk	Middleware for redux that returns a function instead of action
Busboy	Streaming parser for HTML forms data for node.js
cors	Node.js cross origin resource sharing middleware
express	Node.js web application framework
Firebase-admin	Firebase admin SDK
Firebase-functions	Firebase functions SDK
uuidv4	universally unique identifier library

Figure 21: Important libraries used for Sandbox Social Platform

2.1 Front end

2.1.1 One message feed/post

```
const styles = {
  root: {},
  card: {
    marginBottom: 10,
    marginLeft: 5,
    marginRight: 10,
    minWidth: 345,
    '@media (min-width: 415px)': {
      marginBottom: 10,
      marginLeft: 5,
      marginRight: 10,
      position: 'relative',
      display: 'flex',
    },
  },
  image: {
    minWidth: 200,
  },
  content: {
    paddingRight: 25,
    paddingTop: 25,
    paddingBottom: 25,
    objectFit: 'cover',
  },
  avatar: {
    borderColor: '#768bff',
    border: '2px solid #768bff',
    '@media (min-width: 415px)': {
      width: 150,
      height: 150,
      borderColor: '#768bff',
      border: '2px solid #768bff',
    },
  },
  header: {
    marginRight: 0,
    paddingRight: 0,
  },
  cardActions: {
    disableSpacing: true,
    paddingLeft: 0,
  },
  ScreamDialog: {
    transform: 'rotate(0deg)',
    marginLeft: 'auto',
  },
  username: {
    color: 'primary',
  },
  date: {
    color: 'secondary',
  },
  actions: {
    '@media (min-width: 415px)': {
      display: 'block',
      alignItems: 'center',
      justifyContent: 'space-between',
    },
  },
  delete: {
    position: 'absolute',
    left: '90%',
    top: '10%',
  },
}

```

Figure 22: CSS for message post

Here is the CSS for one message post. Here, we style the avatar, the header contents like name and date, the message, and the actions, such as like, comment, and expand. We even have our media queries in here to adapt to the size of the screen.

2.1.2 Comment thread

```
const styles = (theme) => ({
  ...theme.spreadThis,
  profileImage: {
    borderColor: '#768bff',
    border: '2px solid #768bff',
    '@media (min-width: 415px)': {
      width: 150,
      height: 150,
      borderColor: '#768bff',
      border: '2px solid #768bff',
    },
  },
  dialogContent: {
    padding: 20,
  },
  closeButton: {
    position: 'absolute',
    left: '90%',
  },
  expandButton: {
    marginLeft: 'auto',
    '@media (min-width: 415px)': {
      position: 'absolute',
      left: '90%',
    },
  },
  spinnerDiv: {
    textAlign: 'center',
    marginTop: 50,
    marginBottom: 50,
  },
  username: {
    color: 'primary',
  },
  date: {
    color: 'secondary',
  },
})

```

Figure 23: CSS for comment thread

Here is the CSS style of the comment thread. Here, we style the expand button that you see in the previous message post. When you click the expand button on the message post, it will bring you to the comment thread. I even included a spinner to show that the comments for this post are loading.

2.1.3 Profile

```
const styles = (theme) => ({
  ...theme.spreadThis,
  paperBackground: {
    color: theme.palette.tertiary.contrastText,
    padding: 20,
    borderColor: theme.palette.primary.light,
    border: '2px solid #768bff',
    minWidth: 221,
  },
})
```

Figure 24: CSS for profile

This is the CSS of the profile section. Here, we design the background, padding, and border.

2.1.4 Post form modal

```
const styles = (theme) => ({
  ...theme.spreadThis,
  submitButton: {
    position: 'relative',
    float: 'right',
    marginTop: 10,
  },
  progressSpinner: {
    position: 'absolute',
  },
  closeButton: {
    position: 'absolute',
    left: '91%',
    top: '6%',
  },
})
```

Figure 25: CSS for Post form

This is the CSS of the modal in which you post a message. There are the submit button and the close button. I even included a spinner to show that the post is loading.

2.1.5 NavBar

```
export class Navbar extends Component {
  render() {
    const { authenticated } = this.props
    return (
      <div>
        <Appbar>
          <Toolbar className="nav-container">
            {authenticated ? (
              <Fragment>
                <PostScream />
                <Link to="/">
                  <MyButton tip="Home">
                    <HomeIcon></HomeIcon>
                  </MyButton>
                </Link>
                <Notifications />
              </Fragment>
            ) : (
              <Fragment>
                <Button color="inherit" component={Link} to="/login">
                  Login
                </Button>
                <Button color="inherit" component={Link} to="/">
                  Home
                </Button>
                <Button color="inherit" component={Link} to="/signup">
                  Signup
                </Button>
              </Fragment>
            )}
          </Toolbar>
        </Appbar>
      </div>
    )
  }
}
```

Figure 26: Code for Navigation bar

Here is the code for the navigation bar. It checks if you are an authenticated user before showing the post message icon and notifications icon. If you are not authenticated, you will see the login, home, and signup icons instead.

2.1.6 Like button

```
const styles = {
  favourite: {
    paddingLeft: 0,
    paddingRight: 10,
  },
}

const likeButton = !authenticated ? (
  <Link to="/login">
    <MyButton tip="Like" tipClassName={classes.favourite}>
      <ThumbUpAltOutlinedIcon color="primary" />
    </MyButton>
  </Link>
) : this.likedScream() ? (
  <MyButton
    tip="Undo like"
    onClick={this.unlikeScream}
    tipClassName={classes.favourite}
  >
    <ThumbUpAltIcon color="primary" />
  </MyButton>
) : (
  <MyButton
    tip="Like"
    onClick={this.likeScream}
    tipClassName={classes.favourite}
  >
    <ThumbUpAltOutlinedIcon color="primary" />
  </MyButton>
)
return likeButton
```

Figure 27: Code for Like button

Here is the code and CSS for the like button. The like button will show a “hollow” icon if it has not been liked and a “full” icon if it has been liked.

2.1.7 Notification

```
let notificationsIcon;
    if (notifications && notifications.length > 0) {
      notifications.filter((not) => not.read === false).length > 0
        ? (notificationsIcon = (
            <Badge
              badgeContent={
                notifications.filter((not) => not.read === false).length
              }
              color="secondary"
            >
              <NotificationsIcon />
            </Badge>
          ))
        : (notificationsIcon = <NotificationsIcon />);
    } else {
      notificationsIcon = <NotificationsIcon />;
    }
    let notificationsMarkup =
      notifications && notifications.length > 0 ? (
        notifications.map((not) => {
          const verb = not.type === "like" ? "liked" : "commented on";
          const time = dayjs(not.createdAt).fromNow();
          const iconColor = not.read ? "primary" : "secondary";
          const icon =
            not.type === "like" ? (
              <FavoriteIcon color={iconColor} style={{ marginRight: 10 }} />
            ) : (
              <ChatIcon color={iconColor} style={{ marginRight: 10 }} />
            );

          return (
            <MenuItem key={not.createdAt} onClick={this.handleClose}>
              {icon}
              <Typography
                component={Link}
                color="default"
                to={/users/${not.recipient}/scream/${not.screamId}}
              >
                <h5>
                  {not.sender} {verb} your scream {time}
                </h5>
              </Typography>
            </MenuItem>
          );
        })
      ) : (
        <MenuItem onClick={this.handleClose}>
          You have no notifications yet
        </MenuItem>
      );
    return (
      <Fragment>
        <Tooltip placement="top" title="Notifications">
          <IconButton
            aria-owns={anchorEl ? "simple-menu" : undefined}
            aria-haspopup="true"
            onClick={this.handleOpen}
          >
            {notificationsIcon}
          </IconButton>
        </Tooltip>
        <Menu
          anchorEl={anchorEl}
          open={Boolean(anchorEl)}
          onClose={this.handleClose}
          onEntered={this.onMenuOpened}
        >
          {notificationsMarkup}
        </Menu>
      </Fragment>
    );
```

Figure 28: Code for Notification

Here is the code for notifications. The icon will appear once the user receives an activity from the backend. There will be a badge that tells you the number of notifications you have.

2.1.8 Comments

```
const styles = (theme) => ({
  ...theme.spreadThis,
  commentImage: {
    borderColor: '#768bff',
    border: '2px solid #768bff',
    '@media (min-width: 415px)': {
      width: 150,
      height: 150,
      borderColor: '#768bff',
      border: '2px solid #768bff',
    },
  },
  commentData: {
    marginLeft: 10,
    '@media (min-width: 415px)': {
      marginLeft: 40,
    },
  },
  username: {
    color: 'primary',
  },
  date: {
    color: 'secondary',
  },
})
```

Figure 29: CSS for comment

Here is the CSS style for the comment section. Here, we design the avatar, message, date, and title of the comment. We even included media queries here so that it adapts to the screen size of the browser.

2.1.9 Home page layout

```
const styles = {
  root: { minWidth: 345 },
  item1: {
    order: 2,
    '@media (min-width: 600px)': {
      order: 1,
    },
  },
  item2: {
    order: 1,
    '@media (min-width: 600px)': {
      order: 2,
    },
  },
}

class home extends Component {
  componentDidMount() {
    this.props.getScreams()
  }
  render() {
    const { screams, loading } = this.props.data

    const { classes } = this.props
    let recentScreamsMarkup = !loading ? (
      screams.map((scream) => <Scream key={scream.postId} scream={scream} />)
    ) : (
      <ScreamSkeleton />
    )
    return (
      <Grid container spacing={1} className={classes.root}>
        <Grid item sm={8} xs={12} className={classes.item1}>
          {recentScreamsMarkup}
        </Grid>

        <Grid item sm={4} xs={12} className={classes.item2}>
          <Profile />
        </Grid>
      </Grid>
    )
  }
}
```

Figure 30: Code for home page

This is the CSS and code for the home page layout. It displays the message feed as well as the profile of the authenticated user. The media queries included here switch the profile to the top of the page when the screen size becomes smaller.

2.1.10 Login page

```
const styles = (theme) => ({
  ...theme.spreadThis,
  paper: {
    display: 'flex',
    flexDirection: 'column',
    alignItems: 'center',
    marginBottom: theme.spacing(14),
  },
  avatar: {
    margin: theme.spacing(1),
    width: theme.spacing(7),
    height: theme.spacing(7),
    alignSelf: 'center',
    justifyContent: 'center',
    display: 'flex',
  },
  submit: {
    margin: theme.spacing(3, 0, 2),
  },
})
```

Figure 31: CSS for login page

This is the CSS for the login page. The CSS for the login page is similar to the sign-up page as well. Here, we position the avatar, which is the logo, to be at the top and center of the page. This is followed by the actual form for logging in.

2.1.11 Theme

This long piece of code is the theme for the whole website. It includes the CSS for all the components as well as the font being used. The font that I choose is ‘HKGrotesk Pro’.

```
const HKGrotesk = {
  fontFamily: 'HKGroteskPro',
  fontStyle: 'normal',
  fontDisplay: 'swap',
  fontWeight: 400,
  src: `
    url(${HKGroteskPro}) format('woff2')
  `,
  unicodeRange:
    'U+0000-00FF, U+0131, U+0152-0153, U+02BB-02BC, U+02C6, U+02DA, U+02DC, U+2000-206F, U+2074, U+20AC, U+2122, U+2191, U+2193, U+2212, U+2215, U+FEFF',
}
const theme = createMuiTheme({
  palette: {
    primary: {
      light: '#768bff',
      main: '#335eea',
      dark: '#0035b7',
      contrastText: '#ffffff',
    },
    secondary: {
      light: '#7f94c0',
      main: '#506690',
      dark: '#223c62',
      contrastText: '#ffffff',
    },
    tertiary: {
      light: '#eeffff',
      main: '#bbdefb',
      dark: '#8aacc8',
      contrastText: '#000000',
    },
  },
  spreadThis: {
    typography: {
      fontFamily: HKGrotesk,
      useNextVariants: true,
    },
    form: {
      textAlign: 'center',
    },
    image: {
      margin: '20px auto 20px auto',
    },
    pageTitle: {
      margin: '10px auto 10px auto',
    },
    textField: {
      margin: '10px auto 10px auto',
    },
    button: {
      marginTop: 20,
      position: 'relative',
    },
    customError: {
      color: 'red',
      fontSize: '0.8rem',
      marginTop: 10,
    },
    progress: {
      position: 'absolute',
    },
    invisibleSeparator: {
      border: 'none',
      margin: 4,
    },
    visibleSeparator: {
      width: '100%',
      borderBottom: '1px solid rgba(0,0,0,0.1)',
      marginBottom: 20,
    },
    paper: {
      padding: 20,
    },
    profile: {
      '& .image-wrapper': {
        textAlign: 'center',
        position: 'relative',
        '& button': {
          position: 'absolute',
          top: '80%',
          left: '70%',
        },
      },
      '& .profile-image': {
        width: 200,
        height: 200,
        objectFit: 'cover',
        maxWidth: '100%',
        borderRadius: '50%',
      },
      '& .profile-details': {
        textAlign: 'center',
        '& span, svg': {
          verticalAlign: 'middle',
        },
        '& a': {
          color: '#335eea',
        },
      },
      '& hr': {
        border: 'none',
        margin: '0 0 10px 0',
      },
      '& svg.button': {
        '&:hover': {
          cursor: 'pointer',
        },
      },
    },
    buttons: {
      textAlign: 'center',
      '& a': {
        margin: '20px 10px',
      },
    },
  },
})
```

Figure 32: Code for theme

2.2 Back end

2.2.1 Authentication

The authentication for the social platform will be using the Firebase Authentication. However, we will be using additional measures such as JSON Web Tokens (JWT).

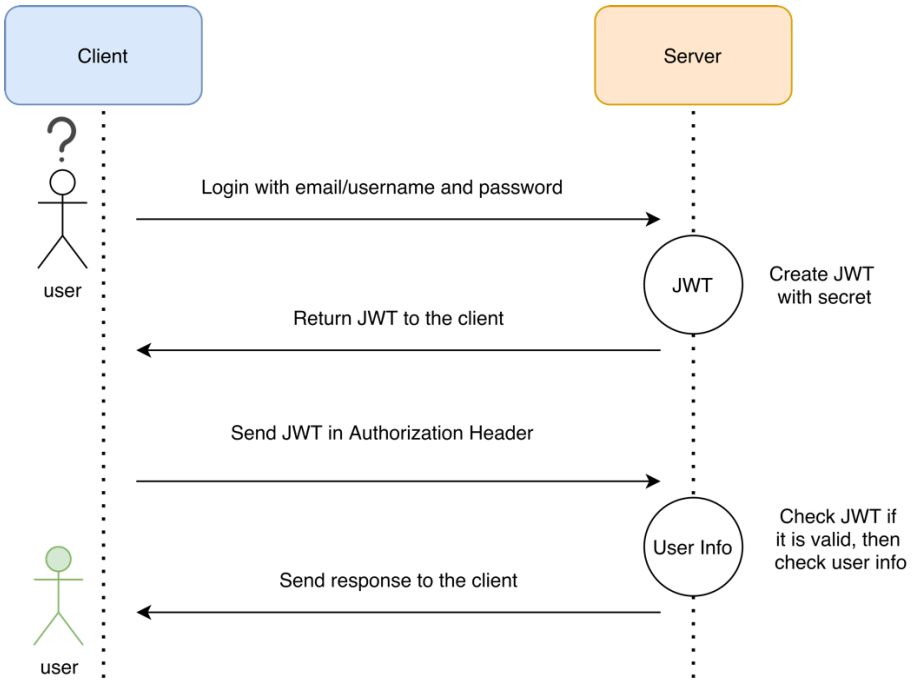


Figure 33:Explanation for how JWT works

The figure above is an explanation of how JWT works. When the user logins in with their email and password, the server would create a unique token for them. When they request anything from the server, they will now have to validate with the token. The server will then check if it is an authenticated user or if the token itself is still valid. If the token expires, the server will not respond.

2.2.1.1 Validation

1. Regex validation for email.

```
const isEmail = (email) => {
  const regEx = /^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/
  if (email.match(regEx)) return true
  else return false
}
```

Figure 34: Regex code

2. Validation for sign up data. Creation of errors to notify users.

```
exports.validateSignupData = (data) => {
  let errors = {}

  if (isEmpty(data.email)) {
    errors.email = 'Must not be empty'
  } else if (!isEmail(data.email)) {
    errors.email = 'Must be a valid email address'
  }

  if (isEmpty(data.password)) errors.password = 'Must not be empty'
  if (data.password !== data.confirmPassword)
    errors.confirmPassword = 'Passwords must match'
  if (isEmpty(data.handle)) errors.handle = 'Must not be empty'

  return {
    errors,
    valid: Object.keys(errors).length === 0 ? true : false,
  }
}
```

Figure 35: Error creation code

3. Validation for login up data. Creation of errors to notify users.

```
exports.validateLoginData = (data) => {
  let errors = {}

  if (isEmpty(data.email)) errors.email = 'Must not be empty'
  if (isEmpty(data.password)) errors.password = 'Must not be empty'

  return {
    errors,
    valid: Object.keys(errors).length === 0 ? true : false,
  }
}
```

Figure 36: Validation code

2.2.1.2 Sign Up

1. We first import the Firebase admin SDK so that we can create the admin functions. We then initialize the app and export both the Firebase admin and the Firestore database.

```
1 const admin = require('firebase-admin')
2 admin.initializeApp()
3 const db = admin.firestore()
4 module.exports = { admin, db }
```

Figure 37: Import Firebase admin and initialize app

2. We then import the admin and database from the previous step. The configuration and Firebase SDK module are also needed here. We then initialize the firebase app.

```
1 const { admin, db } = require("../util/admin");
2 const config = require("../util/config");
3 const { uuid } = require("uuidv4");
4 const firebase = require("firebase");
5 firebase.initializeApp(config);
```

Figure 38: Import Firebase and relevant modules to initialize app

3. We create a sign-up function that requires a request and sends a response to the user. It requires the email, password, confirm password, and their handle (name). It then sends the user's response to the validation step to validate the inputs of the user.

   If the user's data is valid, we then check if their handle(name) is being used. If it is used, we respond by telling them that the name has already been taken. If it is not in use, we proceed with the creation of the user.

   We then get the user's Unique identifier and the user's identifier token from the data of the creation for the user. We call the `getIdToken()` method to get the token from the user data.

After the creation of the user, we will send the user's credentials to the database for storage. A temporary user image will be created for the user until they change their profile picture.

When everything has been created successfully, we send a JSON status of 201 which represents that everything is good. The user's token will then be included in the JSON message for authentication of the user.

```
exports.signup = (req, res) => {
  const newUser = {
    email: req.body.email,
    password: req.body.password,
    confirmPassword: req.body.confirmPassword,
    handle: req.body.handle,
  }

  const { valid, errors } = validateSignupData(newUser)

  if (!valid) return res.status(400).json(errors)

  const noImg = 'no-Img.png'

  let token, userId
  db.doc(`/users/${newUser.handle}`)
    .get()
    .then((doc) => {
      if (doc.exists) {
        return res.status(400).json({ handle: 'this handle is already taken' })
      } else {
        return firebase
          .auth()
          .createUserWithEmailAndPassword(newUser.email, newUser.password)
      }
    })
    .then((data) => {
      userId = data.user.uid
      return data.user.getIdToken()
    })
    .then((idToken) => {
      token = idToken
      const userCredentials = {
        handle: newUser.handle,
        email: newUser.email,
        createdAt: new Date().toISOString(),
        imageUrl: `https://firebasestorage.googleapis.com/v0/b/${config.storageBucket}/o/${noImg}?alt=media`,
        userId,
      }
      return db.doc(`/users/${newUser.handle}`).set(userCredentials)
    })
    .then(() => {
      return res.status(201).json({ token })
    })
    .catch((err) => {
      console.error(err)
      if (err.code === 'auth/email-already-in-use') {
        return res.status(400).json({ email: 'Email is already is use' })
      } else {
        return res
          .status(500)
          .json({ general: 'Something went wrong, please try again' })
      }
    })
}
```

Figure 39: User Sign up code

4. Users can now sign up using the API endpoint https://asia-southeast2-sandbox-social.cloudfunctions.net/api + the sign-up route. This is because we have exported it to the Firebase Cloud Functions API and posted the sign-up route.

```
app.post('/signup', signup)

exports.api = functions.region('asia-southeast2').https.onRequest(app)
```

Figure 40: Sign up API

2.2.1.3 Sign In

1. The initial steps for signing in are almost the same that as the sign-up section.

   For the sign-in section, we require the user's login credentials and if everything is validated, we send them a response.

   We first receive the user's email and password which will be validated. After the validation, if the data is alright, we will proceed with the signing in of the user and the validated user will receive their JWT as a response. If not, they will be receiving
   errors for incorrect regex or credentials.

```
exports.login = (req, res) => {
  const user = {
    email: req.body.email,
    password: req.body.password,
  }

  const { valid, errors } = validateLoginData(user)

  if (!valid) return res.status(400).json(errors)

  firebase
    .auth()
    .signInWithEmailAndPassword(user.email, user.password)
    .then((data) => {
      return data.user.getIdToken()
    })
    .then((token) => {
      return res.json({ token })
    })
    .catch((err) => {
      console.error(err)
      return res
        .status(403)
        .json({ general: 'Wrong credentials, please try again' })
    })
}
```

Figure 41: User Sign in code

2. The API route for the login is similar to the one for the sign up section.

```
app.post('/login', login)
```

Figure 42: Sign in API

2.2.2 Profile

2.2.2.1 Get User's Data

1. We have created a function which allows anyone to see their profile. They only need to type in the API endpoints for the user's collection with the user's handle(name).

   If the user is found, we will return the data to the next section. If not, we will return a JSON error message which tells them the user is not found.

```
exports.getUserDetails = (req, res) => {
  let userData = {};
  db.doc(`/users/${req.params.handle}`)
    .get()
    .then((doc) => {
      if (doc.exists) {
        userData.user = doc.data();
        return db
          .collection("posts")
          .where("userHandle", "==", req.params.handle)
          .orderBy("createdAt", "desc")
          .get();
      } else {
        return res.status(404).json({ errror: "User not found" });
      }
    })
```

Figure 43: Get user profile

2. The data from part one will then be passed on and be put into objects for the data to be mapped. This data will then be passed on as a JSON message to the front-end.

```
.then((data) => {
      userData.screams = [];
      data.forEach((doc) => {
        userData.screams.push({
          body: doc.data().body,
          createdAt: doc.data().createdAt,
          userHandle: doc.data().userHandle,
          userImage: doc.data().userImage,
          likeCount: doc.data().likeCount,
          commentCount: doc.data().commentCount,
          postId: doc.id,
        });
      });
      return res.json(userData);
    })
    .catch((err) => {
      console.error(err);
      return res.status(500).json({ error: err.code });
    });
};
```

Figure 44: Convert to JSON

2.2.2.2 Add User's Data

1. For adding and editing the user's details, we have created a function that receives input from the user. It then sends the data for reduction with the function for reducing user details.

```
exports.reduceUserDetails = (data) => {
  let userDetails = {}

  if (!isEmpty(data.bio.trim())) userDetails.bio = data.bio
  if (!isEmpty(data.website.trim())) {
    // https://website.com
    if (data.website.trim().substring(0, 4) !== 'http') {
      userDetails.website = `http://${data.website.trim()}`
    } else userDetails.website = data.website
  }
  if (!isEmpty(data.location.trim())) userDetails.location = data.location

  return userDetails
}
```

Figure 45: Trim data

2. After the reduction, we then submit the data to the database which returns a JSON message about the success of the update.

```
exports.addUserDetails = (req, res) => {
  let userDetails = reduceUserDetails(req.body)

  db.doc(`/users/${req.user.handle}`)
    .update(userDetails)
    .then(() => {
      return res.json({ message: 'Details added successfully' })
    })
    .catch((err) => {
      console.error(err)
      return res.status(500).json({ error: err.code })
    })
}
```

Figure 46: Update user details

2.2.3 Message Feed

1. To receive all the messages from the message feed, we have to import the database module.

```
1 const { db } = require('../util/admin')
```

Figure 47: Import database module

2. We then call the post collection from the database, which will be pushed into a JSON object and returned to the user.

```
1 exports.getAllPost = (req, res) => {
2   db.collection('posts')
3     .orderBy('createdAt', 'desc')
4     .get()
5     .then((data) => {
6       let screams = []
7       data.forEach((doc) => {
8         screams.push({
9           postId: doc.id,
10
11          ...doc.data(),
12        })
13      })
14
15      return res.json(screams)
16    })
17    .catch((err) => {
18      console.error(err)
19      res.status(500).json({ error: err.code })
20    })
21 }
```

Figure 48: Pull data from Post collection and convert into JSON

2.2.4 Post

1. To post a message, the user will have to submit the JSON data to the server. The JSON data will have to contain a message-body if not an error will appear. If all the required fields are found in the JSON body, a post will be created with 0 likes and comments.

```
1 exports.postOnePost = (req, res) => {
2   if (req.body.body.trim() === "") {
3     return res.status(400).json({ body: "Body must not be empty" });
4   }
5   const newScream = {
6     body: req.body.body,
7     userHandle: req.user.handle,
8     userImage: req.user.imageUrl,
9     createdAt: new Date().toISOString(),
10    likeCount: 0,
11    commentCount: 0,
12  };
```

Figure 49: Validate post data

2. The post will then be sent to the database to be added to the post collection. You will then be able to view the post on the message feed.

```
db.collection("posts")
  .add(newScream)
  .then((doc) => {
    const resScream = newScream;
    resScream.postId = doc.id;

    res.json(resScream);
  })
  .catch((err) => {
    res.status(500).json({ error: "something went wrong" });
    console.error(err);
  });
};
```

Figure 50: Send post to database

2.2.5 Comment

1. To comment on a post, we follow the same method as the post.The user will have to submit the JSON data to the server. The JSON data will have to contain a message-body if not an error will appear. The data for the comment will have to contain the post ID of the post it is commenting on, this is the additional requirement that is different from a post.

```
exports.commentOnPost = (req, res) => {
  if (req.body.body.trim() === "") {
    return res.status(400).json({ comment: "must not be empty" });
  }
  const newComment = {
    body: req.body.body,
    createdAt: new Date().toISOString(),
    postId: req.params.postId,
    userHandle: req.user.handle,
    userImage: req.user.imageUrl,
  };
```

Figure 51: Validate Comment data

2. If all the required fields are found in the JSON body, it will search for the post in the posts collection. If the post is found, the comment count will be updated to increase by 1. The comment will then be added to the comments collection in the database.

```
db.doc(`/posts/${req.params.postId}`)
    .get()
    .then((doc) => {
      if (!doc.exists) {
        return res.status(404).json({ error: "post not found" });
      }
      return doc.ref.update({ commentCount: doc.data().commentCount + 1 });
    })
    .then(() => {
      return db.collection("comments").add(newComment);
    })
    .then(() => {
      res.json(newComment);
    })
    .catch((err) => {
      console.error(err);
      res.status(500).json({ error: "Something went wrong" });
    });
};
```

Figure 52: Add comment to database

2.2.6 Notification

2.2.6.1 Notifications for Like

The notification for likes is created by using the Firestore database module. We then call a snapshot from the database collection for likes. This would allow us to receive real-time updates from the likes collection.

We then create a new document in the notifications document and set the type to like. The read property will also be set to false since it is a new notification.

```
exports.createNotificationOnLike = functions.firestore
  .document('likes/{id}')
  .onCreate((snapshot) => {
    return db
      .doc(`/posts/${snapshot.data().postId}`)
      .get()
      .then((doc) => {
        if (
          doc.exists &&
          doc.data().userHandle !== snapshot.data().userHandle
        ) {
          return db.doc(`/notifications/${snapshot.id}`).set({
            createdAt: new Date().toISOString(),
            recipient: doc.data().userHandle,
            sender: snapshot.data().userHandle,
            type: 'like',
            read: false,
            postId: doc.id,
          })
        }
      })
      .catch((err) => console.error(err))
  })
```

Figure 53: Create Notification in database on like

2.2.6.2 Notifications for Comment

The notification for comments is very similar to the one for likes. This time we will be “listening” to the comments collection instead. The notification type will also be set to comment.

```
exports.createNotificationOnComment = functions.firestore
  .document('comments/{id}')
  .onCreate((snapshot) => {
    return db
      .doc(`/posts/${snapshot.data().postId}`)
      .get()
      .then((doc) => {
        if (
          doc.exists &&
          doc.data().userHandle !== snapshot.data().userHandle
        ) {
          return db.doc(`/notifications/${snapshot.id}`).set({
            createdAt: new Date().toISOString(),
            recipient: doc.data().userHandle,
            sender: snapshot.data().userHandle,
            type: 'comment',
            read: false,
            postId: doc.id,
          })
        }
      })
      .catch((err) => {
        console.error(err)
        return
      })
  })
```

Figure 54: Create notificaiton for database on comment

2.2.7 Like

2.2.7.1 Liking a Post

For liking a post, the user will click the like button of the post. This sends data to the API which receives the information of the post as well as the user who liked it.

If the post can be found in the post collection, a new “like” document will be created, and this increases the like count for the post itself.

```
exports.likePost = (req, res) => {
  const likeDocument = db
    .collection('likes')
    .where('userHandle', '==', req.user.handle)
    .where('postId', '==', req.params.postId)
    .limit(1)

  const screamDocument = db.doc(`/posts/${req.params.postId}`)

  let screamData

  screamDocument
    .get()
    .then((doc) => {
      if (doc.exists) {
        screamData = doc.data()
        screamData.postId = doc.id
        return likeDocument.get()
      } else {
        return res.status(404).json({ error: 'post not found' })
      }
    })
    .then((data) => {
      if (data.empty) {
        return db
          .collection('likes')
          .add({
            postId: req.params.postId,
            userHandle: req.user.handle,
          })
          .then(() => {
            screamData.likeCount++
            return screamDocument.update({ likeCount: screamData.likeCount })
          })
          .then(() => {
            return res.json(screamData)
          })
      } else {
        return res.status(400).json({ error: 'post already liked' })
      }
    })
    .catch((err) => {
      console.error(err)
      res.status(500).json({ error: err.code })
    })
}
```

Figure 55: Create Like in database

2.2.7.2 Un-liking a Post

Un-liking a post is similar to that of liking a post. This time we just decrease the like count by 1 for the specific post.

```
exports.unlikePost = (req, res) => {
  const likeDocument = db
    .collection('likes')
    .where('userHandle', '==', req.user.handle)
    .where('postId', '==', req.params.postId)
    .limit(1)

  const screamDocument = db.doc(`/posts/${req.params.postId}`)

  let screamData

  screamDocument
    .get()
    .then((doc) => {
      if (doc.exists) {
        screamData = doc.data()
        screamData.postId = doc.id
        return likeDocument.get()
      } else {
        return res.status(404).json({ error: 'post not found' })
      }
    })
    .then((data) => {
      if (data.empty) {
        return res.status(400).json({ error: 'post not liked' })
      } else {
        return db
          .doc(`/likes/${data.docs[0].id}`)
          .delete()
          .then(() => {
            screamData.likeCount--
            return screamDocument.update({ likeCount: screamData.likeCount })
          })
          .then(() => {
            res.json(screamData)
          })
      }
    })
    .catch((err) => {
      console.error(err)
      res.status(500).json({ error: err.code })
    })
}
```

Figure 56: Delete Like in database

Reference

1. “the collaborative interface design tool.,” Figma. [Online]. Available: https://www.figma.com/. [Accessed: 18-Apr-2021].

2. “UI: A popular React UI framework,” Material. [Online]. Available: https://materialui.com/. [Accessed: 18-Apr-2021].

3. J. T. Mark Otto, “Bootstrap,” Bootstrap · The most popular HTML, CSS, and JS library in the world. [Online]. Available: https://getbootstrap.com/. [Accessed: 18-Apr-2021].

4. “React – A JavaScript library for building user interfaces,” – A JavaScript library for building user interfaces. [Online]. Available: https://reactjs.org/. [Accessed: 18-Apr-2021].

5. “Firebase Brand Guidelines,” Google. [Online]. Available: https://firebase.google.com/brand-guidelines. [Accessed: 18-Apr-2021].