RMIT International University Vietnam

Assignment Cover Page

Subject Code	COSC2769
Subject Name	Full Stack Development
Location	SGS Campus
Title of Assignment	Group Project
Teacher Name	Dr. Tri Dang
Group Name	Group 6
Group Members	s3978081 – Hoang Thai Phuc s3974735 – Tran Manh Cuong s3927502 – Vu Tuan Linh s3977955 – Le Thien Son s3978072 – Nguyen Ha Tuan Nguyen
Assignment Due Date	September 9, 2024
Date of Submission	September 9, 2024
Declaration of Authorship	We declare that in submitting all work for this assessment, we have read, understood and agreed to the content and expectations of Assessment Declaration as specified in https://www.rmit.edu.au/students/my-course/assessment-results/assessment
Consent to Use	We give RMIT University permission to use our work as an exemplar and for showcase/exhibition display

Table of Contents

1. INTRODUCTION

1.1. Overview

Throughout this project, the main goal is to develop an online social media website that allows users to connect with each other, share content, and build/participate in communities. Being a social network website, its main features are allowing users to create accounts and through it create posts they want to share with other users, interact with them by reacting and commenting on each other posts, and administration privilege for admins of groups and the site itself. It also gives them the ability to create groups or join those that match their interests. The site is managed by an admin who can approve group creation and delete communities, contents or suspend users that they deem inappropriate. Utilizing knowledge and website development skills from the Full Stack Development course, we were able to bring this concept to reality. This report will go over how we came up with the design as well as the development process and testing phase of our team.

2. SYSTEM DESIGN

2.1. High-Level Design

2.1.1. Architecture Overview

Our application follows a traditional three-tier architecture consisting of a frontend, backend, and database. Here is an overview of each layer:

- **Frontend Layer**: Built with React TypeScript and TailwindCSS, handles the user interface and interactions, and communicates with the backend via RESTful API calls.
- **Backend Layer**: Built with TypeScript, NodeJS, and ExpressJS, implements the application logic and API endpoints, authentications, authorization handling, and interacts with the database.
- **Database Layer**: Using MongoDB as data storage, stores all the data including users, posts, and other important entities.

2.1.2. Components

Several key components that work together to create an interactive user experience are at the center of our application. Each component represents a core functionality within the system. Having a deep understanding of these components is crucial for grasping the overall structure and capability:

- **User**: Represents registered users of the application, manages user profiles, friends, and notifications.
- **Admin**: Special user type with elevated privileges, manages user accounts, posts, and group creation requests.
- **Group**: Represents user-created groups, manages group members, posts, and visibility settings.
- **Post**: Represents user-generated content, supports text content and image uploads.
- **Comment**: Represents user responses to posts, supports text content only.
- **Reaction**: Represents user reactions to posts or comments, supports different types (Like, Love, Haha, Angry).
- **Request**: Represents various types of requests (Friend Request, Group Member Request, Group Creation Request), manages the lifecycle of these requests.

2.1.3. Relationships and Interactions

The power of a social network lies in the connections and interactions between its various components. This section outlines how users, groups, posts, and other elements of the system relate to and interact with each other. These relationships form the backbone of the application's functionality and user experience:

User - User:

- Users can send friend requests to other users.
- Users can accept or reject friend requests.
- Users can unfriend other users.

User - Group:

- Users can create groups (pending admin approval).
- Users can join public groups or request private groups.
- Users can be admins or members of groups.

User - Post:

- Users can create posts on their profile or in groups.
- Users can edit or delete their own posts.
- Users can view posts based on visibility settings and group membership.

User - Comment:

- Users can add comments to posts.
- Users can edit or delete their own comments.

User - Reaction:

- Users can add reactions to posts or comments.
- Users can change or remove their reactions.

Admin - User:

- Admins can suspend or reactivate user accounts.

Admin - Group:

- Admins can approve or reject group creation requests.

Admin – Post:

- Admins can view, edit, and delete all posts in the system.

Group Admin - Group:

- Group admins can approve or reject group join requests.
- Group admins can remove members from the group.

2.1.4. User Experience Considerations

While not a fully developed product, we have considered focusing on crucial user experience considerations to the application to provide seamless user interaction with the application:

- **Real-time Updates**: Implement real-time updates for all notifications related to the user.
- **Optimistic UI Updates**: Update the UI immediately when a user performs an action, then update the server.
- **Error Handling**: Implement robust error handling to provide error messages to users when API calls fail.
- **Responsive Design**: Ensure the front end is responsive and works well on various device sizes.

2.2. Low-Level Design

2.2.1. Design Choices

MVC (Model-View-Controller) as the design pattern:

- After evaluating the pros and cons of several architectural approaches including MVC, Monolithic, and Microservices, the MVC pattern was selected due to its simplicity, maintainability, and scalability for a medium-sized web application.
- MVC makes it simpler to maintain, test, and expand the code by assigning each component to a specific part of the application: models manage data, controllers handle logic, and the view manages display.
- Moreover, the application's components are well organized, which makes it easier for developers to work on individual components without affecting others.

Use of TypeScript over JavaScript:

- TypeScript provides static typing, allowing for the definition of variable types, function signatures, and object structures, which leads to less runtime errors, improved code quality, and early detection of bugs during the development process.
- Interfaces and types in TypeScript guarantee clearly defined structures of objects, making the codebase more maintainable, while reasoning about quite large codebases becomes easy.
- TypeScript makes use of better editor support. The features include intelligent code completion, inline documentation, and error checking, which creates a much better development experience with high productivity.

2.2.2. Components Details

- **User**: User, like its name, is the person using the application. They can login to the website and the application will get the data for the User and then store it in the cookie for further authentication. They can post, comment, join groups, as well as connect with other users by adding them as friends.
- **Admin**: Admin can interact with every user account, group creation requests, posts and comments. They are able to suspend / reactivate user accounts, approve the creation of groups, and delete comments or posts at will. In short, they manage the application itself. They are represented by a Boolean in each User Session: isAdmin.
- **Group**: Group is a component that represents user-created groups, before it is added to the database it must be approved by the admin first. Groups are managed by the person that created it or Users that are given the group admin privileges, they can delete inappropriate posts, and delete users from the group, note that the website admin can also manage the group's posts and users. The group admin can edit certain characteristics of the group like its name, description, visibility (Public or Private), change the group image, cover image. The user joins the group by creating a join request that the group admin can approve, and they can leave the Group if wanted. In addition, users can set a special privacy setting: private that only allows users in the same group to see the post.
- **Post**: Posts are content generated by users; it contains text and/or images. A user can add, edit, or delete a post and set its visibility to PUBLIC: anyone on the application can see it, or FRIENDS: friends with the user can see the post. Additionally, when posted in a private group only users in that specific group can view the

post. If the post is edited the user can see its past versions also. It depends on components like User, Comment and Reaction to properly display the post.

- **Comment**: Allow Users to leave comments on posts, each comment can be reacted just like a post. The author of the comment can also edit it, and the previous version of the comment will be saved in the editHistory array. The application only allows text comments for now. The component Reaction is a dependency of Comment
- **Reaction**: Allows users to interact with posts and comments by expressing their emotions through predefined reaction types (Like, Love, Haha, Angry). The component supports editing and deleting reactions if the User chooses to do so.
- **Request**: The request component represents the requests below and manages the lifecycle of them, when a request is sent, approved, or denied a notification will also be sent to the user that sent it. There are four main types of requests in the application:
  - **Friend requests**: created when a user wants to add another user, can be approved by the target user or cancel by the one who sent it.
  - **Group join requests**: created when a user wants to join a group, can be approved by the group admin or cancel by the user.
  - **Group creation requests**: created when a user wants to create a group, can be approved by the site admin.

2.2.3. Database Design

The database schema is designed for the application to be implemented on a NoSQL database, MongoDB, where entities are stored in collections. To model the relationships between different entities like User, Post, Group, and many other components, ObjectId references are utilized. The following is the description of how the entities are stored in the MongoDB collections and the visualization of the database design using the Entity Relationship Diagram (ERD).

Database Collections:

- "admins" collection: Store the admin's information.
- "users" collection: Store the user's information, profile image, and notifications.
- "groups" collection: Store the group's information, group image and cover image.
- "posts" collection: Store the post's details, images, reactions, comments, and edit histories.
- "friendrequests" collection: Store the user's friend request details.
- "grouprequests" collection: Store the group member request details.
- "groupcreationrequests" collection: Store the group creation request details.

Entity Relationship Diagram (ERD):

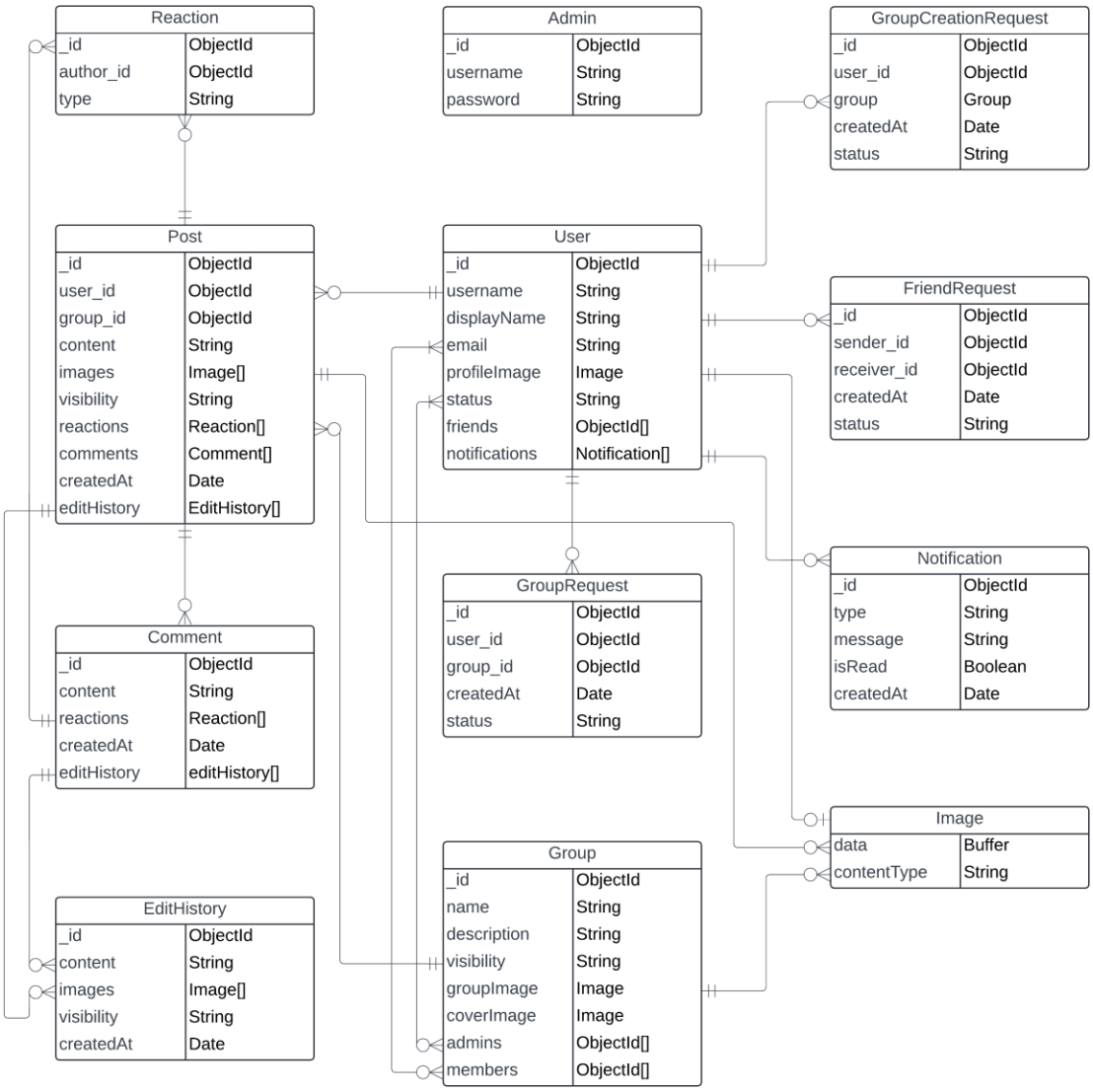

Figure 1: Entity Relationship Diagram (ERD)

2.2.4. API Design

The API of the application is designed in a RESTful design pattern, which makes it very user-friendly for communicating with the system in a stateless way. The design supports basic Create, Read, Update, and Delete operations for core resources like Users, Posts, Groups, and Requests. Here is the summary of the key endpoints:

- **Authentication Endpoints**: Manage user registration, login, and logout. These endpoints further secure the application by authenticating users before accessing other resources. Operations include registering new users, logging in existing users, and managing user sessions.
- **User Endpoints**: Allow users to manage their profiles, view and manage friends, notifications, and requests. Admin users can retrieve and manage all users, including suspending and reactivating accounts. Regular users can retrieve information about other users, get friend recommendations, and manage their notifications.

- **Group Endpoints**: Handle group-related operations, such as creating, updating, and retrieving groups. Group admins can manage group memberships, retrieve group requests, and perform other group-related actions.
- **Post Endpoints**: Manage the creation, retrieval, and modification of posts. Users can create, edit, delete, or interact with posts using comments or reactions. The system ensures posts are properly sorted and filtered for users and admins, and it tracks post editing history for transparency.
- **Friend Request Endpoints**: Allow users to send, accept, and reject friend requests, managing user relationships in the system. Notifications are triggered to inform users of friend request actions.
- **Group Request Endpoints**: Allow users to request membership in specific groups. Group admins can review and respond to these requests, with the system tracking the status as pending, accepted, or rejected.
- **Group Creation Request Endpoints**: Allow users to request the creation of new groups, which are reviewed by admins. Notifications are sent to users about the status of their requests, and group creation includes submitting details like group name, description, visibility, and optional images.

3. IMPLEMENTATION

3.1. Development Process

Before setting up the environment, we had a kickoff meeting to decide on a working process. In this meeting, we agreed on a weekly working process where we would update each other on what has been done and what needs to be done so that every member would know the progress of our application. After that, the project started by setting up a development environment using the MERN (MongoDB, Express, React, Node) stack. The first step was to configure the project using TypeScript. The following steps were taken:

For the front-end, a React project was created using TypeScript and TailwindCSS for styling. This library was chosen due to its ease of use and utility, helping us develop a quick and responsive UI.

We used a RESTful API design with ExpressJS and TypeScript in our back-end. MongoDB was chosen as the database for its flexibility and ease of integration with the Node.js ecosystem. We also did some research and used Multer to integrate image saving and displaying into the application.

This process of ideating and pinpointing the technology needed for our project was completed in just 1 week, after that we went on to develop the basic version of each component as well as setting up controllers, middleware, and routes in our backend. To help debug the back-end while the front-end was still in development, Postman was chosen to test call APIs.

After the back-end was finished, we started working on improving our front end and connecting the two together. In this phase we focused on making the pages dynamic, using routers and links from React. This also highlighted some unforeseen problems in the backend which we fixed right after encountering it by utilizing utility functions and interfaces, ensuring that it would not be a problem in the future. This process continued until the end of the project when our team was satisfied with the design and the application produced no “app-crashing” errors.

3.2. Algorithms and Code Snippets

3.2.1. Authentication Management

Express Session: User authentication is handled through sessions managed by the express-session middleware. When a user logs in, their session data, including their role, is stored on the server, and the session ID is sent to the user's browser in the form of a cookie. This allows the backend to identify the user and maintain their session across multiple requests.

```
declare module "express-session" {
    Cuong Tran, last week | 1 author (Cuong Tran)
    interface SessionData {
        userId: mongoose.Types.ObjectId;
        username: String;
        isAdmin: boolean;
    }
}

app.use(
    session({
        secret: process.env.SESSION_SECRET || "session_secret_key",
        resave: false,
        saveUninitialized: false,
        cookie: {
            maxAge: 1000 * 60 * 60 * 24,
        },
    }),
);
```

Figure 2: User session declaration

CORS: Because our application has a distinct frontend and backend, CORS needed to be enabled to allow the frontend to communicate with the backend easily. The specific setting enables the backend to access session data contained in cookies. By doing this, front-end code needs to send requests with the credentials: 'include' option, which ensures that cookies, including the session ID, are sent along with the request. Hence enables the backend to check the user's session and determine their role through sessions when accessing protected resources.

Authentication Middleware: To restrict role-based access, we implemented two important middleware functions: isAuthenticated and isAdmin. These middleware functions ensure that users are verified and permitted appropriate to their role:

- isAuthenticated checks whether a user is logged in by verifying their session data. If the user is not authenticated, they are redirected to the login page or denied access to the requested resource.
- isAdmin on the other hand is used to verify that the authenticated user has the necessary admin privileges. It checks the user's role stored in the session. If the user is not an admin, they are denied access to admin-level resources.

```
export const isAuthenticated = (req: Request, res: Response, next: NextFunction) => {
    if (req.session && req.session.userId) {
        // User is authenticated, proceed to the next middleware or route handler
        return next();
    } else {
        // User is not authenticated, return an error response
        return res.status(401).json({ message: "Unauthorized: Please log in to access this resource" });
    }
};

export const isAdmin = (req: Request, res: Response, next: NextFunction) => {
    if (req.session && req.session.isAdmin) {
        // User is an admin, proceed to the next middleware or route handler
        return next();
    } else {
        // User is not an admin, return an error response
        return res.status(403).json({ message: "Forbidden: Only Admins can access this resource" });
    }
};
```

Figure 3: Authentication middleware

3.2.2. Client-side Authentication

In addition to implementing server-side role-based access control, we also set up client-side authentication to ensure seamless user data access throughout the application, even after the user closes their browser window. This is achieved through React Context, custom hooks, and local storage.

Auth Provider: The AuthProvider component is responsible for managing and providing user data across the front-end. It leverages React's Context API, enabling other components to access the authenticated user's information without the tedious of passing it down manually through many children and props, allowing the application to have information be globally available. Because of this, our provider context needed to wrap around the whole application as the "first class component".

```
const AuthContext = createContext<AppContext | undefined>(undefined);

const getInitialState = () => {
  const currentUser = localStorage.getItem('user');
  return currentUser ? (JSON.parse(currentUser) as Auth) : {};
};

export const AuthProvider: FC<PropsWithChildren> = ({ children }) => {
  const [auth, setAuth] = useState<Auth>(getInitialState);

  useEffect(() => {
    localStorage.setItem('user', JSON.stringify(auth));
  }, [auth]);

  return (
    <AuthContext.Provider value={{ auth, setAuth }}>
      {children}
    </AuthContext.Provider>
  );
};
```

Figure 4: AuthProvider using ReactContext feature

The user's information is saved in localStorage after they log in. This ensures that the user stays authenticated, and their data remains accessible across sessions. When the application loads, it checks localStorage for existing user data and restores it into the context. This prevents the need for users to re-login after closing the browser or refreshing the page.

Finally, we created a custom useAuth hook to simplify fetching user data, ensuring that developers can quickly access authentication-related information anywhere in the application without interacting directly with the context and reduce many redundant codes while limiting the possible errors in the future.

3.2.3. Data Validation and Security

Data validation and security measures ensure that input from users is processed accurately and securely. The application includes multiple levels of data validation and security to avoid malicious inputs, ensuring system stability.

Input Validation: The system checks for required fields during user registration, post creation, or group creation (e.g., username, email, password). Input is validated for proper format and criteria, such as valid usernames, email addresses, and correct object ID formats, reducing invalid or malformed data.

```
{
  id: 1,
  inputProps: {
    name: 'username',
    type: 'text',
    placeholder: 'Username',
    required: true,
    pattern: '[A-Za-z0-9_]+',
    title: 'Only letters, numbers, underscores, and hyphens are allowed.',
  },
  label: 'Username',
  renderCondition: [formState.LOGIN, formState.SIGNUP],
},
```

Figure 5: Validate user's input from the front-end side – User Registration

```
// Validate the user ID format
if (userId && !mongoose.Types.ObjectId.isValid(userId)) {
    return res.status(400).json({ message: "Invalid user ID" });
}

// Validate the input fields
const { name, description, visibility } = req.body;
if (!name || !description || !visibility) {
    return res.status(400).json({ message: "Name, description, and visibility are required" });
}

// Check visibility enum
if (!["Public", "Private"].includes(visibility)) {
    return res.status(400).json({ message: "Visibility must be either 'Public' or 'Private'" });
}
```

Figure 6: Validate user's input from the back-end side – Group Creation Request

Password Hashing: User passwords are securely hashed by bcrypt, ensuring that even if the database is compromised, attackers cannot retrieve plain-text passwords.

```
// Hash the password
const SALT_ROUNDS = 10;
const hashedPassword = await bcrypt.hash(password, SALT_ROUNDS);
```

Figure 7: Password hashing

3.2.4. File Upload Management

The file upload management of the application was implemented using the Multer middleware, a widely used tool in Node.js for handling multipart/form-data, especially for uploading files like images. As our application has to handle lots of image upload such as user's profile image, group's images, and post's images, this middleware is very essential.

Multer Setup: Multer is configured to accept only images and limits the size of uploaded files. This ensures that only acceptable file types and sizes are processed, preventing large or malicious files from being uploaded.

```
const storage = multer.memoryStorage();
const fileUpload = multer({
    storage: storage,
    limits: {
        // Limit file size to 2MB
        fileSize: 2 * 1024 * 1024,
    },
    fileFilter: (req, file, cb) => {
        if (file.mimetype.startsWith("image/")) {
            cb(null, true);
        } else {
            cb(new Error("Invalid file type, only images are allowed!"));
        }
    },
});
```

Figure 8: Multer storage configuration

Storing Files in the Database: When files are uploaded, multer handles the processing and stores the binary data along with its MIME type in the database. For example, when a user uploads a profile image, the binary data is saved to the profileImage field of the user's document. The same applies to group images or post images.

```
_id: ObjectId('66ceb6b3181e1bbf7a23f260')
username : "cuongtran"
displayName : "Cuong Tran"
email : "cuongtran@gmail.com"
password : "$2b$10$dGL72kjn176eC2oJYf1bOuoEGxWDdosqCd9EOXK0I.erB458Umi4W"
▾ profileImage : Object
    data : Binary.createFromBase64('iVBORw0KGgoAAAANSUhEUgAAAlwAAAJcCAIAAACjfu3nAAAgAElEQVR4nOy9abclx3EglFVd3tbd7/uBrrR2EgsFEgCEAEuLhK…', 0)
    contentType : "image/png"
status : "Active"
▸ friends : Array (empty)
▸ notifications : Array (9)
```

Figure 9: User's profileImage Sample Data

Base64 Encoding: Images are encoded as base64 strings in the user data or post information returned by the API, allowing them to be easily transmitted over HTTP and rendered on the client side. This enables seamless display of images on the front end without additional processing.

```
const userFullDetails = {
  _id: user._id,
  username: user.username,
  displayName: user.displayName,
  email: user.email,
  virtualProfileImage:
    user.profileImage &&
    user.profileImage.contentType &&
    user.profileImage.data
      ? `data:${
          user.profileImage.contentType
        };base64,${user.profileImage.data.toString("base64")}`
      : undefined,
  createAt: user.createdAt,
  updatedAt: user.updatedAt,
};

// Return the user
return res.status(200).json(userFullDetails);
```

Figure 10: Sample return data from the API

3.2.5. User's Notification Management

The application's notification system is important for keeping users informed of what goes on with posts and requests. Here's how it works:

Notifications related to Posts:

- If a user creates a new post, the user's friends receive a notification.
- If a user creates a new post in a group, all group's members receive a notification.
- If a user engages with a post by commenting or reacting to it, the post owner receives a notification.

```
// Notify the post author
if (post.user_id != author_id) {
  // Ensure the user is not notifying themselves
  // Find the comment author to get the displayName
  const commentAuthor =
    await User.findById(author_id).select('displayName');
  if (commentAuthor) {
    const postAuthor = await User.findById(post.user_id);
    if (postAuthor) {
      const notificationMessage = `${commentAuthor.displayName} commented on your post.`;
      postAuthor.notifications.push({
        type: 'Comment',
        message: notificationMessage,
        isRead: false,
        createdAt: new Date(),
      });
      await postAuthor.save();
    }
  }
}
```

Figure 11: Notification when user comment on post

Notifications related to Requests:

- **Friend Requests**: When a user sends a friend request, the receiver is notified. Once the request is accepted or rejected, the user who made the request is informed.
- **Group Requests**: When a user sends a request to join a group, the group's admins are notified. Once the request is accepted or rejected, the user who made the request is informed.
- **Group Creation Requests**: When a user requests to create a new group, the request is sent to the system admins. Once the request is accepted or rejected, the user who made the request is informed.

```
// Send notifications to all group admins
const sender = await User.findById(senderId).select('displayName');

for (const adminId of group.admins) {
  const admin = await User.findById(adminId);

  if (admin) {
    admin.notifications.push({
      type: 'Group',
      message: `${sender!.displayName} has requested to join the group "${group.name}".`,
      isRead: false,
      createdAt: new Date(),
    });
    await admin.save();
  }
}
```

Figure 12: Notification when user request to join group

```
requester.notifications.push({
  type: 'Group',
  message: `Your request to join the group "${group.name}" has been accepted.`,
  isRead: false,
  createdAt: new Date(),
});
await requester.save();
```

Figure 13: Notification when user's group member request is accepted

3.2.6. Application Routing

To handle navigation between pages in my application, we utilized React Router v6, a robust and flexible routing module for React applications. One of the major features we used was the loader function, which fetches data before generating the page and ensures that the necessary data is accessible when the user navigates a specified path.

The loader function allows data fetching to happen before the component is rendered. This ensures that the required data is available as soon as the user navigates to the page, preventing the need for additional loading states after the component has been displayed. Enhancing user experience by avoiding flickering and delays caused by asynchronous data fetching after the page has rendered.

```
path: '/users/:userId',
element: <UserPage />,
loader: async ({ params }) => {
  const endpoint = `http://localhost:8080/users/${params.userId}`;
  const res = await fetch(endpoint, {
    method: 'GET',
    credentials: 'include',
  });
  return await res.json();
},
```

Figure 14: A user specific route leveraging the params and loader function

For example, on a user specific page, the loader fetches user-specific data based on the userId parameter in the route to the endpoint in APIs and access this information in the page by using the useLoaderData() to retrieve the fetched data from the loader. This guarantees that the profile data is preloaded and displayed as soon as the user navigates to the profile page.

```
const UserPage = () => {
  const loaderData = useLoaderData() as User;

  return (
    <Layout stickyRightSideCmp={<UserSideBar userData={loaderData} />}>
      <UserPanel userData={loaderData} />
    </Layout>
  );
};
```

Figure 15: Getting pre-fetch data from loader

3.2.7. Infinite Loading/ Pagination

When retrieving a significant quantity of data, it is not advisable to get everything in the database since large amounts of information require time to transport over the network and the majority of it would be of no use. One way we tackle this is through the implementation of pagination on backend and infinite loading in front-end in order to efficiently handle large datasets by dividing the data into smaller chunks or "pages." This approach enhances performance and user experience, especially when displaying large sets of data, such as lists of users, products, or posts.

On the server side, we implemented pagination by allowing the client to request a specific "page" of data along with “limit” to specified number of items per page through the use of URL parameters. The backend processes this request and returns only the relevant portion of data for that page, rather than sending the entire dataset. For example, when fetching posts our server API is a GET to localhost:8080/posts. By specifying a page and limit like this: localhost:8080/posts?page=1&limit=10 we can tell our backend server to return only 10 first posts in the database. Conveniently, mongoose already provided us with functions to do this by using skip() and limit().

```
// Pagination setup
const pageNumber = parseInt(page as string);
const pageSize = parseInt(limit as string);
const skip = (pageNumber - 1) * pageSize;
```

Figure 16: Pagination in back-end side

```
// Execute the query and process the results
const posts = await postsQuery
  .sort({ createdAt: -1 })
  .skip(skip)
  .limit(pageSize)
  .select(
    "_id user_id group_id content images visibility reactions comments createdAt",
  )
  .populate({
    path: "user_id",
    select: "_id username displayName profileImage contentType",
  })
  .exec();
```

Figure 17: Getting necessary data with pagination

On the client side, we achieved this through the use of the JavaScript built-in IntersectionObserver() to monitor an element as it enters view. By inserting a Fetching at the end of our posts list, we can discern when a user has scrolled to the end of the posts and so inform the JavaScript to request further data from the backend. Furthermore, after each fetch, the page is raised by one to guarantee that the following fetch has the next ten data points that the user needs rather than the old, duplicated data.

```
const [viewRef, inView] = useInView<HTMLDivElement>(hasMore);
```

Figure 18: Custom useInView() hook to observe element

```
// Fetch posts from the endpoint
const fetchPosts = async () => {
  // Set this so fetch won't be called multiple times
  setIsFetching(true);
  try {
    // SearchParams
    const params = new URLSearchParams({
      page: page.toString(),
    });

    const res = await fetch(`${fetchEndpoint}?${params.toString()}`, {
      method: 'GET',
      credentials: 'include',
    });

    console.log(res);

    if (res.ok) {
      // Get data and parse it
      const data: any[] = await res.json();
      const fetchedPosts = data.map((post) => parsePost(post));

      console.log(fetchedPosts);

      // 10 is the limit per page
      if (fetchedPosts.length < 10) {
        setHasMore(false); // If no more posts, set hasMore to false
      }
      setPosts((prevPosts) =>
        prevPosts ? [...prevPosts, ...fetchedPosts] : fetchedPosts,
      );

      // Increase page value for next fetch
      setPage((prev) => prev + 1);
      setIsFetching(false);
    }
  } catch (error) {
    console.error('Failed to fetch posts:', error);
  }
};
```

Figure 19: A fetch function with pagination applied

```
useEffect(() => {
  if (!isFetching && hasMore) {
    fetchPosts();
  }
}, [inView]);
```

Figure 20: Use effect to fetch more data when user scroll to the end

3.2.8. Front-end and Back-end Integration

When data is returned from the backend, it often comes as a JSON string. In order to work with this data seamlessly on the front-end, it needs to be parsed and converted into appropriate types that match the application's data structure. This ensures that operations on the data, such as rendering or processing, are efficient and error-free.

Considering we adopted TypeScript as our programming language, we had to design our own data type on the front-end using the interface keyword to leverage its strict type system. While JSON is a convenient format for

exchanging data between the frontend and backend, it needs to be converted into Typescript objects or other types that can be used directly in the application logic. This also ensures that our program uses the same data type consistently across the app rather than mixing and matching amongst various team members, reducing the possibility of mistakes and defective applications.

```typescript
export interface Group {
  id: string;
  name: string;
  description: string;
  visibility: GroupVisibility;
  groupImage?: string;
  coverImage?: string;
  admins: User[];
  members: User[];
}

export enum GroupVisibility {
  PUBLIC,
  PRIVATE,
}
```

Figure 21: A group interface/datatype

Although JSON is human-readable, the types of certain fields (such as "_id" or "virtualGroupImage") might not be exactly what the front-end expects, requiring conversion to the appropriate types. Therefore, once the JSON data is received in the front-end, we use our customized parse method to convert the JSON string into our object type. During parsing, additional steps are often needed to ensure that certain fields are cast to the correct types.

```typescript
export const parseGroup = (data: any): Group => {
  return {
    id: data._id,
    name: data.name,
    description: data.description,
    visibility:
      GroupVisibility[
        (
          data.visibility as string
        ).toUpperCase() as keyof typeof GroupVisibility
      ],
    groupImage: data.virtualGroupImage,
    coverImage: data.virtualCoverImage,
    admins: data.admins,
    members: (data.members as any[]).map((mem) => parseBasicUser(mem)),
  };
};
```

Figure 22: A parse function to map Json data to group data

4. TESTING

4.1. Testing Strategy

Our testing strategy for this full-stack web application using React, React Router, NodeJS, Express, and MongoDB focused on manual testing and leveraging the built-in capabilities of our tech stack. We employed a multi-layered testing approach, including component testing, API testing, and user acceptance testing, all conducted on standard desktop environments:

- **Component Testing**: For the React frontend, we manually tested each component in isolation. We created a set of test cases for each component, checking their rendering, state changes, and event handling on desktop browsers.
- **API Testing:** We tested our Express API endpoints manually using Postman - a popular software tool for API development and testing.
- **User Acceptance Testing:** We recruited a group of 10 potential users to interact with the application on desktop computers. Their feedback was collected directly.

4.2. Correctness Testing

Ensuring the correctness of our application's functionalities was paramount. We focused on several key areas:

Authentication:

- Users could sign up with valid credentials.
- Users could log in with the correct credentials.
- Invalid login attempts were properly rejected and displayed to the users.

Authorization & Routing:

- Protected React Router routes were only accessible to authenticated users.
- Express middleware properly validated authentication for protected API endpoints.
- MongoDB queries respected user permissions.

Data Integrity:

- All data (posts, comments, requests, etc.) were accurately stored in MongoDB.
- Retrieved data matched the original input and was parsed correctly to display on the front-end.
- Updates on data were correctly reflected in both the database and the UI.

Functionality: conducted manual test to ensure the UI always displays correct data based on the current context. For example, public posts were visible to all users, new friends were added to the friend list after the user accepted a friend request, etc.

4.3. Performance Testing

Load Testing: We simulated concurrent users accessing the application by opening multiple browser tabs and sessions and then observed response times and application behavior. This was tested with different levels of 10, 25, and 50 simultaneous users.

Network Performance Analysis: We used the browser's Network tab in developer tools to analyze and optimize network performance:

- Measured initial page load time, number of requests, and total download size
- Identified large assets (images, CSS, and JS/TS files)
- Analyzed API call performance and optimized slow-performing queries

4.4. User Experience Testing

Usability Testing: We conducted usability testing sessions with our group of 10 test users on desktop computers. Each user was given a set of tasks to complete while we observed their interactions. Any feedback or confusion was all noted down to make UI/UX improvements.

Browser Compatibility: We tested the application on popular desktop browsers to ensure consistent functionality: Google Chrome, Mozilla Firefox, Microsoft Edge, and Safari.

4.5. Results and Evaluation

These were the insights we collected from our testing approach:

- Authentication, Authorization, and Routing mechanisms were functional, with no security vulnerabilities detected.
- Data integrity was maintained across all CRUD operations in MongoDB.
- All functionalities on the UI worked as expected.
- Load Testing saw minimal performance decrease when increasing the number of simultaneous users.
- All metrics for Network Performance Analysis were acceptable.
- Test users were overall satisfied with the UI/UX with small feedback on small components. Adjustments were made accordingly.
- The application showed consistency between different browsers.

5. CONCLUSION

In conclusion, the development of this social network platform successfully achieves its core objective of providing users with a space to connect, share content, and engage with others through posts, comments, and reactions. The platform’s design not only fosters individual connections through friend requests but also enables users to build communities by creating, managing, and joining groups. With role-specific functionalities for group admins and site admins, the system ensures effective moderation and community management.

Moreover, it also provided us with an opportunity to apply our learnings to a practical, relevant project which shows us how an application is built in a professional environment with stakeholders and multiple contributors from different backgrounds, helping us be more prepared for real-life work.