RMIT

RMIT International University Vietnam

Assignment Cover Page

Subject Code:	COSC2769
Subject Name:	Full Stack Development
Location & Campus (SGS or HN) where you study:	SGS Campus
Title of Assignment:	**Group Project**
Teachers Name:	Dr. Tri Dang Tran
Group Name:	Group Project 13
Group Members (names and id numbers):	Duong Phu Dong – s3836528 Nguyen Hai Lam – s3979802 Ly Minh Phuc – s3976250 Nguyen Bao Thinh (Johnny) – s4029791 Du Tuan Vu – S3924489
Assignment due date:	9 Sep by 5:00
Date of Submission:	
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.vn/students/my-studies/assessment-and-exams/assessment
Consent to Use:	We give RMIT University permission to use our work as an exemplar and for showcase/exhibition display.

RMIT

Table of Contents

Group Project 1

Abstract 4

1. Project Overview: 4

2. Project Design: 4

1. Application Architecture Overview: 4

2. Administrator Application Architecture: 6

2.2.1. Admin role and Responsibility: 6

2.2.2. Admin Interface Design: 6

2.2.3. Admin Interaction Flow: 8

Login and Accessing the Dashboard 8

Dashboard Page 8

2.2.4. Managing Groups 8

Viewing Group Information 8

Viewing Group Details 9

Deleting a Group 9

4. Handling Group Creation Requests 9

2.2.5. Managing Users 11

Viewing User Information 9

Suspending or Resuming a User Account 9

Viewing User Activity Details 10

Logging Out 10

3. User Application Architecture: 10

2.3.1. User Roles and Responsibilities: 10

2.3.2. User Interface Design: 10

2.3.3. User Interaction Flow: 11

3. Project Implementation: 12

1. Administrator Page: 12

2. User Interface Implementation – Homepage, ProfilePage, GroupPage: 18

4. Project Testing 26

5. Project Process 26

6. Conclusion 29

RMIT

Table of Figures

Figure 1 Culture Canvas Architecture 6
Figure 2 Admin Interaction Flow 8
Figure 3 User Interaction Flow 11
Figure 4 Package Imported in Admin Page 12
Figure 5 State Initialization in Admin Page 14
Figure 6 React function selectPage to update the active page status 14
Figure 7 React authCheck component for admin authorization and UI rendering 12
Figure 8 Package imported in Home Page 18
Figure 9 Component Definition and Hooks in Home Page 18
Figure 10 Event Handler in Home Page 19
Figure 11 Render Method in Home Page 18
Figure 12 Package Imported in Profile Page 19
Figure 13 Component and State Definition in Profile Page 19
Figure 14 Data Fetching with useEffect 20
Figure 15 Conditional Rendering and JSX Return in Profile Page 20
Figure 16 Package in GroupPage 21
Figure 17 Component Definition in GroupPage 22
Figure 18 Fetching Data in GroupPage 20
Figure 19 Render Method in GroupPage 21
Figure 20 PeoplePage Implementation 24

RMIT

Abstract

This project presents the development of an online social network platform, designed to offer features similar to Facebook, with a focus on user interaction and group dynamics. Built using the MERN stack (MongoDB, Express, React, Node.js) with MySQL or MongoDB for data management, the system implements a RESTful API architecture to enable smooth communication between the frontend and backend. Key functionalities include user and group management, post creation and interaction, comment handling, and reaction tracking. The platform supports both public and private group visibility, as well as a robust notification system for user activities such as friend requests, group approvals, and post interactions. The application is designed to maintain core functionalities during network outages by caching user actions locally and syncing them with the server once the connection is restored. Security features include user authentication, role-based access control, and input sanitization to protect against common vulnerabilities. Future enhancements, such as real-time chat and advanced privacy controls, are being considered to expand the platform's capabilities further. This report details the technical implementation, user experience strategies, and system architecture of the social network platform.

Keywords: Social Network, MERN Stack, RESTful API, MongoDB, MySQL, User Authentication, Offline Functionality, Data Security.

1. Project Overview:

The Culture Canvas platform is designed to facilitate dynamic social interactions similar to Facebook but emphasizes user-driven content creation and group collaboration. The backend, powered by Express and Node.js, interfaces with MongoDB for data storage, ensuring scalability and performance. The frontend is developed with React, providing a responsive user experience and real-time data interaction. This dual-database approach allows for flexible data management and complex query execution, which is critical for the platform's extensive social features.

2. Project Design:

The system is designed using the MERN stack, with MongoDB as the primary databases. The architecture follows a RESTful API design to facilitate efficient communication between the frontend and backend. Key components of the system include user and group management, post creation and interaction, comment handling, reactions, and a notification system. The design ensures scalability, security, and a smooth user experience, even during network disruptions.

1. Application Architecture Overview:

This Application Architecture for Culture Canvas is based on three-tier architecture

a. Presentation Layer – Frontend:

By using ReactJS, this layer is responsible for rendering the user interface (UI) and handling user interactions. It communicates with the backend via API calls to fetch data or execute operations. Components using in this layer are:

- **React Components:** GroupPage, ProfilePage, HomePage, and other components can handle various parts of the UI.
- **State Management:** Redux can be used to manage application-wide states like user information, posts, comments, and notifications.
- **Offline Functionality:** Service workers and local storage are employed to provide offline support, caching posts, comments, and reactions to synchronize with the backend once the connection is restored.

b. Application Logic Layer – Backend:

Using ExpressJS, this layer handles the business logic of the application, processing user requests, managing sessions, and interacting with the database. It exposes a RESTful API that the frontend consumes. Components using in this layer are:

- **Express Routes:** Defines endpoints for operations such as user registration, login, posting, commenting, group management.
- **Authentication and Authorization:** User authentication is handle through sessions (using express-session). Role-based access control (RBAC) is implemented to differentiate between users, group admin, and site admin.
- **Middleware:** Handle tasks like input validation, error handling, session management, and security.

c. Data Layer – Database:

Culture Canvas using MongoDB for managing data, ensuring user data, posts, comments, groups, and other information are stored securely and can be retrieved efficiently. Components using in this layer are:

- **MongoDB:** Document-based database for handling unstructured or semi-structured data, such as comments or reactions that may require more flexibility in schema design.
- **ORM/ODM:** Mongoose is used to interact with the database, abstracting away MongoDB commands into an object-oriented paradigm.

d. Culture Canvas Architecture Diagram Overview:

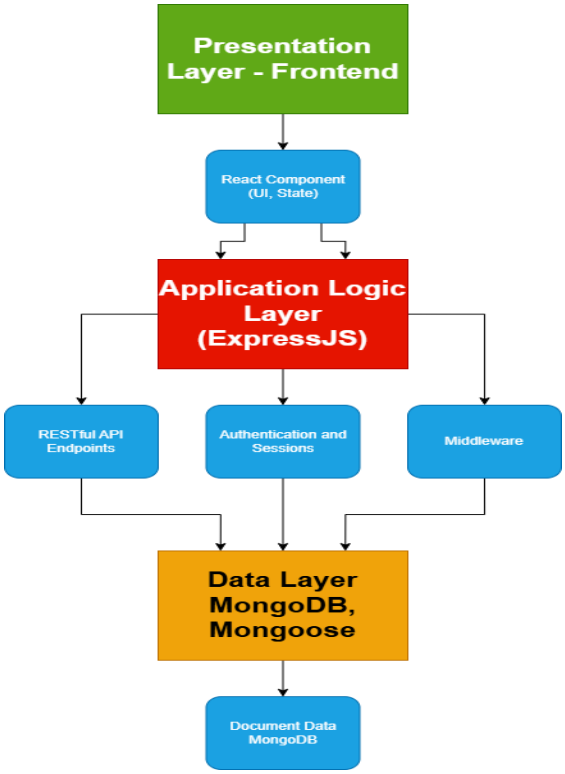

Figure 1 Culture Canvas Architecture

2. Administrator Application Architecture:

2.2.1. Admin role and Responsibility:

- **Approve Group Creation Requests**: Review and either approve or deny requests for creating new user groups. Ensure the group complies with community guidelines and adds value to the platform.
- **Suspend/Resume User Accounts**: Temporarily suspend user accounts if they violate community rules, ensuring the suspension period is justified and documented. The admin is also able to resume accounts after corrective actions are taken.
- **Delete Accounts**: Admins can permanently delete accounts that consistently violate terms of service or pose security risks.
- **Monitor User Behavior**: Regularly review user activity for inappropriate or malicious behavior, including spamming, harassment, or any other form of misconduct

2.2.2. Admin Interface Design:

Dashboard Page

Top Information Boxes:

- Display 5 key metrics:
  - **Total Users**: Shows the total number of users registered on the platform.
  - **Total Posts**: Displays the total number of posts created by users.

- **Posts Today**: Shows the count of posts made on the current day.
- **Total Groups**: Displays the total number of groups created.
- **Group Creation Requests**: Displays the number of pending requests for group creation.

  Bar Chart: Below the information boxes, there will be a bar chart to visually represent the metrics, such as total posts over the last week or the number of groups created over time.

Groups Page

Group List:

- Admin can view a list of all groups, with each group displaying: Group Name and Group Admin Name.

View Group Details:

- The admin can click on a group to view all posts and comments within that group.
- Posts and comments will be listed chronologically, with the option to delete them individually if necessary.

Delete Group:

- There is an option to delete the entire group if it violates any policies.

Group Creation Request Page:

- On this secondary page, the admin can view pending group creation requests and choose to either accept or deny them.

Users Page

- This page will display all users, along with key information such as:
  - **Username**.
  - **Email**.
  - **Registration Date**.
  - **Status** (active, suspended).
  - **Suspend/Resume User Accounts**:
- Only available if the user is the site admin. Admin will have the privilege over regular user, which is to suspend or resume any user's account if they violate or comply with community rules.

User Details:

- Admins can click on a user's name to view more detailed information, including their activity history, posts, and any reported behavior.

2.2.3. Admin Interaction Flow:

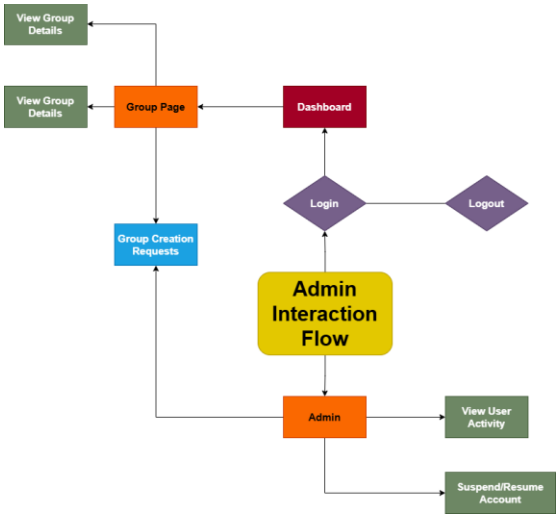

Figure 2 Admin Interaction Flow

Login and Accessing the Dashboard

- **Action**: Admin logs into the platform using their credentials.
- **Result**: The system authenticates the admin and redirects them to the **Dashboard** page.

Dashboard Page

- **View Overview Information**:
  - **Action**: Admin views the 5 boxes showing metrics (Total Users, Total Posts, Posts Today, Total Groups, Group Creation Requests).
  - **Result**: Admin gains a high-level overview of the platform's current status.
- **Analyze Data via Bar Chart**:
  - **Action**: Admin scrolls down to the bar chart for a visual breakdown of metrics like post activity over time.
  - **Result**: Admin can analyze trends or spot unusual spikes or drops in activity.

2.2.4. Managing Groups

Viewing Group Information

- **Action**: Admin navigates to the **Groups Page**.

- **Result**: A list of all existing groups appears, each showing the **Group Name** and **Group Admin Name**.

Viewing Group Details

- **Action**: Admin clicks on a specific group to see its posts and comments.
- **Result**: A new page loads with all posts and comments displayed in chronological order.
- **Option**: Admin can choose to delete specific posts or comments by selecting the appropriate delete button.

Deleting a Group

- **Action**: On the group's detail page, admin clicks the **Delete Group** button.
- **Result**: A confirmation prompt appears. Admin confirms the action, and the group is deleted from the platform.

4. Handling Group Creation Requests

- **Action**: Admin navigates to the **Group Creation Requests** page from the Groups Page.
- **Result**: A list of all pending group creation requests is displayed.
- **Option 1**: Admin clicks the **Accept** button next to a request to approve it.
  - **Result**: The group is created, and the request is removed from the list.
- **Option 2**: Admin clicks the **Deny** button to reject a group creation request.
  - **Result**: The request is denied and removed from the list.

2.2.5. Managing Users

Viewing User Information

- **Action**: Admin navigates to the **Users Page**.
- **Result**: A list of all users is displayed, showing key information such as **Username**, **Email**, **Registration Date**, and **Account Status**.

Suspending or Resuming a User Account

- **Action**: Admin clicks the **Suspend** button next to a user who violates community rules.

- **Result**: The system suspends the user's account in short term, and their status changes to "**Suspended"**. Suspended accounts cannot comment or post anything. They can, however, still view other people's posts and group posts.
- **Option**: If an account is already suspended, the admin can click the **Resume** button to reactivate the account.

Viewing User Activity Details

- **Action**: Admin clicks on a user's name to view their full profile and activity history.
- **Result**: A detailed page shows the user's posts, comments, and any reports or violations they have received.

Logging Out

- **Action**: Admin clicks the **Logout** button from the navigation menu.
- **Result**: Admin is securely logged out of the platform and redirected to the login page.

3. User Application Architecture:

2.3.1. User Roles and Responsibilities:

- Users: Register, login, send/accept friend requests, create posts, comment, and react to content.
- Group Members: Participate in group activities by creating posts, commenting, and reacting within groups.
- Group Admins: Manage their specific groups by approving membership requests, removing members, and moderating content.

2.3.2. User Interface Design:

- Home Page (Post Feed):
  - Displays posts from friends and group posts that the user is a member of the group.
  - Includes excerpts for long posts with a "Read More" option/
  - React, comment, and share functionality directly on posts.
- Profile Page:
  - Show user information (name, profile picture, posts).
  - Option to edit personal information and view friend lists.
- Group Page:
  - Displays group information, including description, members, and posts.

- Ability to create and manage group posts and comments.

2.3.3. User Interaction Flow:

- Post Creation:
  - Users can create posts on their home feed or within groups. Posts can include text and images.
  - Posts have visibility settings (Public or Friends-only).
  - Posts are editable as well (only by their author), and the edit history is available to view.
- Commenting and Reacting:
  - Users can add comments to posts and react with one of the available reaction types.
  - Users can edit or delete their comments, with a history of edits available.
- Friend List Management:
  - Users can send friend requests, accept/decline requests, and unfriend other users.
- Group Membership:
  - Users can create groups and send requests to join existing groups. The admin is the user to verify the group creation request.
  - There exists group admins and group users within a group. Groups admins monitor the activities of a group

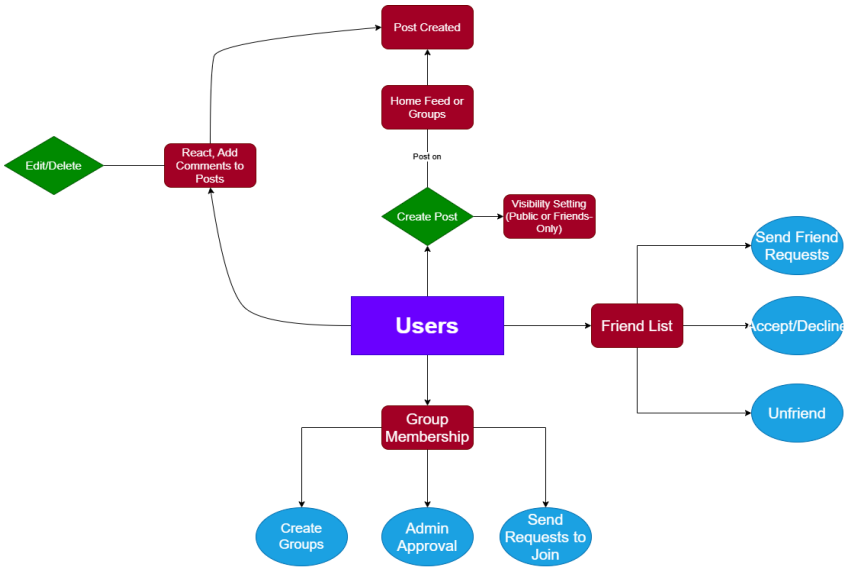

Figure 3 User Interaction Flow

4. Project Implementation:

1. Administrator Page:

4.1.1. Import Package

```
import React, { useState } from "react";
import { useNavigate, Link } from "react-router-dom";
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap-icons/font/bootstrap-icons.css';
import Sidebar from "./sidebar";
import Dashboard from "./dashboard";
import Groups from "./groups";
import Users from "./users";
import Posts from "./posts";
import { useAuth } from "../../context/authContext";
```

Figure 4 Package Imported in Admin Page

React and useState: Imported from the react package, they are essential for building React components and managing state within them.
useNavigate and Link: From react-router-dom, these are used for navigation within the application; useNavigate allows programmatic redirections, and Link provides declarative navigation links.
Bootstrap CSS and Icons: Imported to style the application with Bootstrap's UI toolkit and to use its icon set for enhancing UI elements.
Component Imports (Sidebar, Dashboard, Groups, Users, Posts): These are local components representing different sections of the admin interface, enabling separate management and display functionalities for various aspects like user data, group settings, and posts.
useAuth: This hook, imported from a custom context, facilitates access to authentication states and logic, crucial for conditionally rendering content based on the user's authentication status and permissions.

```
const authCheck = () => {
  if (user && user.admin === true) {
    return (
      <div className="container-fluid bg-secondary min-vh-100">
        <div className="row">
          {toggle && (
            <div className="col-2 bg-white vh-100">
              <Sidebar selectPage={selectPage} />
            </div>
          )}
          <div className="col">
            {activePage === "Dashboard" && <AdminDashboard Toggle={Toggle} />}
            {activePage === "Groups" && <Groups Toggle={Toggle} />}
            {activePage === "Users" && <Users Toggle={Toggle} />}
            {activePage === "Posts" && <Posts Toggle={Toggle} />}
            <button onClick={Toggle}>Toggle Sidebar</button>
          </div>
        </div>
      </div>
    );
  } else {
    return (
      <div>
        <h1>You are not authorized to access this page.</h1>
        <Link to="/login">Please login via this page.</Link>
      </div>
    );
  }
};
```

Figure 5 React authCheck component for admin authorization and UI rendering

selectPage Function: Called when a page is selected from the Sidebar, updating the activePage state to reflect the current user choice, which in turn controls which admin section is displayed in the main content area.

Conditional Rendering in authCheck:If the user is an admin (user && user.admin === true), the main admin layout is rendered with a dynamic sidebar and a content area that updates based on activePage.The layout includes a grid structure facilitated by Bootstrap classes where the sidebar occupies 2 columns when visible, and the content area occupies the remaining space.Each administrative component like AdminDashboard, Groups, Users, Posts can potentially have access to the Toggle function if needed for additional UI control.

Access Control: If the user is not an admin, or not logged in, the component renders a block with a warning message and a link to the login page, enforcing role-based access control to ensure that only authorized users can access the admin functionalities.

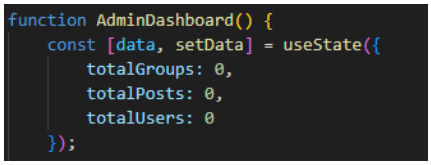

Figure 9: component definitions and hooks in admin dashboard

The AdminDashboard is defined as a functional React component that utilizes hooks to manage its state and side effects. The useState hook initializes the component's state with a structure to store data about total groups, posts, and users. The useEffect hook is employed to fetch this data from the server when the component mounts, ensuring that the dashboard displays up-to-date information without manual refreshing.

60 Event handler in admin dashboard

The component includes an asynchronous function fetchData within useEffect to handle data fetching. This function constructs an array of URLs for the groups, posts, and user endpoints,

sends requests to these URLs, and processes the responses. Errors in fetching or response processing are caught and logged, highlighting robust error handling practices within the component.

```
    return (
        <div>
            <h1>Admin Dashboard</h1>        You, 15 h
            <p>Total Groups: {data.totalGroups}</p>
            <p>Total Posts: {data.totalPosts}</p>
            <p>Total Users: {data.totalUsers}</p>
        </div>
    );
}

export default AdminDashboard;
```

Figure 711 Render Method in Admin Dashboard

The return statement in AdminDashboard defines its JSX structure, which organizes the UI elements into a simple yet informative display. It shows the total numbers of groups, posts, and users, fetched from the server, directly reflecting the real-time status of these entities on the platform. This structure not only provides critical information at a glance but also serves as a real-time monitoring tool for the admin to gauge activity on the platform.

```
return (
    <div>
        <h1>Group List</h1>
        {groups.map((group, index) => (
            <div key={index}>
                <span>{group.name} - Admin: {group.admin}</span>
                <button onClick={() => deleteGroup(group._id)}>Delete</button>
            </div>
        ))}
    </div>
);
```

Figure 85Render Method in Groups Component

Group List Display: Each group is listed with its name and the admin's identifier. This helps users understand who is managing the group.

Delete Button: Each group entry has a delete button which, when clicked, triggers the deleteGroup function with the group's unique identifier (_id). This allows for direct management of groups from the UI.

```
function Users() {
    const [users, setUsers] = useState([]);

    // Fetch users from the backend
    useEffect(() => {
        fetch('http://localhost:8000/user', {
            credentials: 'include',
        })
            .then(response => response.json())       You, 15 hours ago • add ad
            .then(data => {
                const usersArray = Array.isArray(data) ? data : [data];
                setUsers(usersArray);
            })
            .catch(error => console.error('Failed to fetch users:', error));
    }, []);
```

Figure 97 Data fetching and state management

Data Fetching: Within useEffect, a fetch call is made to 'http://localhost:8000/user' with credentials included, essential for handling sessions or secure data transfer. Successful fetch operations update the state with user data, converting it into an array format if necessary to standardize the handling of the data within the component.
Error Handling: Catches and logs errors that occur during the fetch process, providing feedback necessary for debugging and user notifications.

```
// Function to handle user deletion
const deleteUser = (userId) => {
    fetch(`http://localhost:8000/user/${userId}`, {
        method: 'DELETE',
        credentials: 'include',
    })
        .then(response => {
            if (response.ok) {
                setUsers(users.filter(user => user._id !== userId));
            } else {
                alert('Failed to delete the user.');
            }
        })
        .catch(error => console.error('Failed to delete user:', error));
};
```

Figure 108 Event Handler in users component

deleteUser Function: This function is vital for user management, allowing administrators to remove users directly from the system. It sends a DELETE request to the backend and, on successful deletion, filters the removed user out of the state, updating the UI reflectively. Error handling within this function ensures that failures are managed gracefully, alerting the user to any issues encountered during the deletion process.

```
return (
    <div>
        <h1>User List</h1>
        {users.map((user, index) => (
            <div key={user._id}>
                <span>{user.username}</span>
                <button onClick={() => deleteUser(user._id)}>Delete</button>
            </div>
        ))}
    </div>
);
```

Figure 119 Render method in users component

The return block of the Users component constructs the UI:

User List: Dynamically lists all users fetched from the backend, displaying each user's username.
Delete Button: Each user listing includes a delete button that, when clicked, triggers the deleteUser function with the specific user's ID. This functionality is crucial for maintaining an up-to-date and accurate user base.

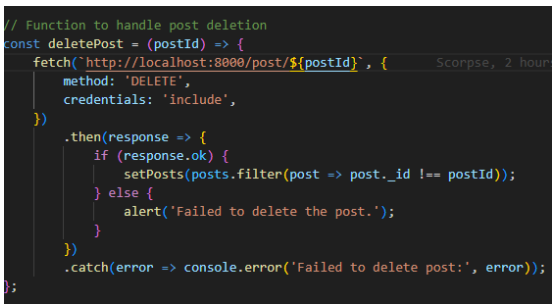

Figure 122 Delete post functionality

The deletePost function sends a DELETE request to the backend to remove a specific post by its ID. If successful, the state is updated by filtering out the deleted post, ensuring the UI remains accurate without needing to reload.
Error handling ensures that any issues encountered during deletion are communicated to the user, maintaining a robust and user-friendly interface.

Figure 133 Render Method

The component renders a list of posts. Each post includes its title and a preview of its content. If the content is longer than 100 characters, it's truncated for brevity.

Each post also includes a delete button, which when clicked, invokes the deletePost function with the post's unique ID.

```
import React, { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import '../admin/admin.css';
import { useAuth } from "../../context/authContext";
```

Figure 214 Package Imported in Sidebar Page

React and Hooks: The component imports React and the useState hook, which is used to manage state within the component.
useNavigate: From react-router-dom, useNavigate is used to programmatically navigate between routes.
Styling and Context: Imports custom CSS for styling and the authentication context to manage authentication states.

```
function Sidebar({ selectPage }) {
  const navigate = useNavigate();
  const [activeItem, setActiveItem] = useState('Dashboard');

  const handleLogout = async () => {
    await logout();
    navigate('/login');
  };

  const handleSelectPage = (page) => {
    if (page === 'Logout') {
      handleLogout(); // Call handleLogout when "Logout" is selected
    } else {
      selectPage(page);
      setActiveItem(page); // Set active item state
      navigate(`/${page.toLowerCase()}`); // Navigate to the selected page
    }
  };
```

Figure 15 Component Definitions

Function Sidebar: Defines the Sidebar component, which receives selectPage as a prop from its parent. This prop is a function intended to change the current active page in the parent component.

handleLogout: Asynchronously triggers the logout process. It calls a logout function (presumably provided via the useAuth context but not shown in the import in your snippet), then navigates to the login route after successfully logging out the user.
handleSelectPage: Handles page selection from the sidebar. If the 'Logout' option is selected, it triggers handleLogout. Otherwise, it sets the active page state and navigates to the corresponding route, making use of dynamic routing based on the page name.

2. User Interface Implementation – Homepage, ProfilePage, GroupPage, PeoplePage:

4.2.1. Import Package

```
import React from "react";
import { useNavigate } from "react-router-dom"; // Correct import for useNavigate
import { useDispatch } from "react-redux";
import CreatePost from "../Posts/CreatePost";
import PostFeed from "../Posts/PostFeed";
import FriendList from "../Friends/FriendList";
import FriendRequest from "../Friends/FriendRequest";
import GroupList from "../Groups/JoinedGroupList";
import JoinedGroupList from "../Groups/JoinedGroupList";
import UnjoinedGroupList from "../Groups/UnjoinedGroupList";

import { setLogout } from "../../slices/authSlice";
import { useAuth } from "../../context/authContext";
```

Figure 167Package imported in Home Page

In the import section, React is imported to enable the creation of React components. The useNavigate hook from react-router-dom is used for programmatically navigating to different pages with the application. To manage state changes and interactions, useDispatch from react-redux allows dispatching actions to the Redux store. Various UI components such as CreatePost, PostFeed, FriendList, FriendRequest, and JoinedGroupList are imported for rendering different parts of homepage. Additionally, setLogout is an action from the Redux slice used to handle user logout, and useAuth is a custom hook that provides access to authentication-related functions.

4.2.2. Component Definition and Hooks

```
const HomePage = () => {
        const navigate = useNavigate();

        const dispatch = useDispatch();
        const { setUser } = useAuth();
```

Figure 178Component Definition and Hooks in Home Page

The HomePage component is a functional React component that ultilizes the navigate function for routing to various pages within the application. It uses dispatch to send actions to the Redux store, enabling state management. Additionally, setUser is employed to update the user state in the authentication context, ensuring that user-related information is kept current.

4.2.3. Event Handlers

```
const handleRedirect = (e) => {
        e.preventDefault();
        navigate("/people");
};

const handleRedirectToFriendRequests = (e) => {
        e.preventDefault();
        navigate("/friendRequest");
};

const handleRedirectToMoreGroups = (e) => {
        e.preventDefault();
        navigate("/moregroups");
};

const handleRedirectToGroupsAdmin = (e) => {
        e.preventDefault();
        navigate("/groupadmin");
};

const handleLogout = async (e) => {
        try {
                e.preventDefault();

                const response = await fetch("http://localhost:8000/logout", {
                        method: "GET",
                        credentials: "include",
                });

                if (!response.ok) {
                        throw new Error("Logout failed");
                }

                dispatch(setLogout());
                setUser(null);

                navigate("/home");
        } catch (e) {
                console.log("Unable to logout");
        }
};
```

Figure 189Event Handler in Home Page

The handleRedirect function navigates to the /people route when triggered. While handleRedirectToFriendRequests takes the user to the /friendRequest route to view friend requests. Similarly, handleRedirectToMoreGroups directs users to the /moregroups route to explore additional groups, and handleRedirectToGroupsAdmin leads to the /groupadmin route for group management. The handleLogout function manages user logout by sending a request to the server, updating the Redux state, clearing the user context, and then navigating back to the home page.

3.2.7. Component and State Definition in Profile Page

```
const ProfilePage = () => {
  const { id } = useParams(); // Getting user ID from route parameters.
  const [profile, setProfile] = useState(null); // Local state for storing profile data.
  const [posts, setPosts] = useState([]); // Local state for storing posts.
  const token = useSelector((state) => state.auth.token); // Accessing auth token from Redux store.
```

Figure 192Component and State Definition in Profile Page

This define the ProfilePage functional component. It uses the useParams hook to extract the id from the URL, which identifies the user. Local state is managed using useState for storing the profile data and posts. The Redux useSelector hook retrieves the authentication token from the store, which will be used to authorize API requests.

3.2.8. Conditional Rendering and JSX Return:

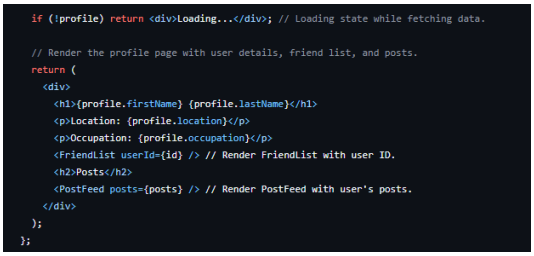

Figure 204Conditional Rendering and JSX Return in Profile Page

This part of the component handles rendering. If the profile data has not been loaded yet (profile is null), is displays a loading message. Once the data is available, it renders the user's profile details including their name, location, and occupation. The FriendList component is rendered to show the user's friends, and the PostFeed component displays the user's posts. This ensures that the page dynamically updates with the correct information once the data fetching is complete.

3.2.11. Fetching Data in GroupPage

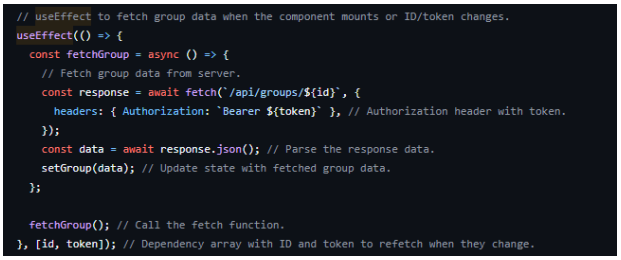

Figure 217Fetching Data in GroupPage

The useEffect hook is employed to handle side effects, specifically for fetching group data when the component mounts or when the id or token changes. An asynchronous function, fetchGroup, is defined to make a fetch request to the API endpoint for the group data. The request includes an authorization header with the token to ensure that the API call is secure. Upon receiving the response, it is parsed as JSON and the component state is updated with this data using setGroup. The dependencies array [id, token] ensures that the effect is triggered whenever either the group ID or the token changes, keeping the component up-to-date with the latest data.

3.2.12. Render Method in GroupPage

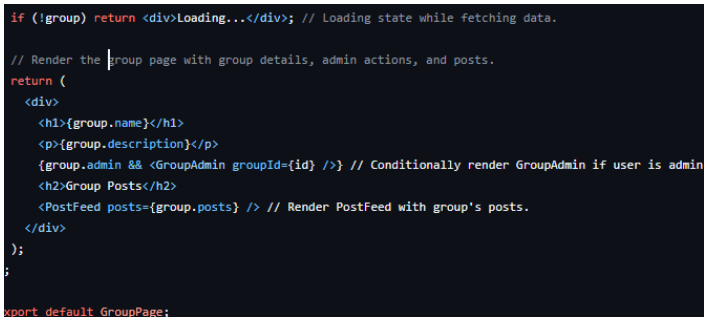

Figure 228Render Method in GroupPage

While the group data is being fetched, the component renders a loading message if the group state is still null. This ensures that users are informed that data is being loaded and prevents the display of incomplete or undefined information. This loading state is crucial aspects of managing user experience, providing feedback while the data is being retrieved. Once the data is successfully fetched and the group state is populated, the component renders the group detail. It displays the group name and description, which are extracted from the group state. If the group has an admin property, indicating that the current user has administrative privileges, the GroupAdmin component is rendered to provide admin-specific functionalities. Below the group details, the PostFeed component is rendered with the group posts, allowing users to view and interact with content associated with the group. This structure ensures a comprehensive and dynamic presentation of the group's information and functionality.

5. Backend Implementation:

Controller Implementation

Get Groups that have NOT been joined

Figure 230Get Groups that have NOT been joined

The getGroup function retrieves groups that the user has not yet joined. It first identifies the user by userId, retrieves the user joined groups, and then filters out groups that the user has already joined by using the $nin MongoDB operator to exclude them from the query result.

Get Groups that have been joined

```
const getUserGroups = async (req, res) => {
        const userId = req.user._id;

        try {
                const user = await userModel.findById(userId).populate("groups").exec();
                if (user && Array.isArray(user.groups)) {
                        const groupIdWithUser = user.groups.map((group) => group._id);
                        console.log("- groupIdWithUser:", groupIdWithUser);

                        const groups = await groupModel.find({
                                _id: { $in: groupIdWithUser },
                        });

                        res.status(200).json(groups);
                } else {
                        console.log([]);
                }
        } catch (err) {
                // handle the error
                console.error(err);
                res.status(500).send("Error fetching groups");
        }
};
```

Figure 242Get Groups that have been joined

The getUserGroups function retrieves all groups the user has joined. It finds the user by userId, retrieves their groups, and then queries the groupModel to get the details of these groups using the $in operator.

Get Groups with Admin Privilege

```
const getAdminGroups = async (req, res) => {
        const userId = req.user._id;

        try {
                const user = await userModel.findById(userId).populate("groups").exec();
                if (user && Array.isArray(user.groups)) {
                        const groupIdWithUserAsAdmin = user.groups
                                .filter((group) => group.admins.includes(userId))
                                .map((group) => group._id);
                        console.log("- groupIdWithUserAsAdmin:", groupIdWithUserAsAdmin);

                        const groups = await groupModel.find({
                                _id: { $in: groupIdWithUserAsAdmin },
                        });

                        res.status(200).json(groups);
                } else {
                        console.log([]);
                }
        } catch (err) {
                // handle the error
                console.error(err);
                res.status(500).send("Error fetching groups");
        }
};
```

Figure 253Get Groups with Admin Privilege

The getAdminGroups function retrieves groups where the user has admin privileges. It filters the user groups to include only those where the user is listed as an admin and returns the details of those groups.

Get Group Requests (for Admins)

```
const getGroupRequests = async (req, res) => {
        const userId = req.user._id;
        const groupId = req.params.id;

        // Check if the ID is valid
        if (!mongoose.isValidObjectId(groupId)) {
                return res.status(404).json({ error: "Incorrect ID" });
        }

        try {
                // Find the group
                const group = await groupModel.findById(groupId);
                if (!group) {
                        return res.status(404).json({ error: "Group not found" });
                }

                // Check if the user is an admin
                if (!group.admins.includes(userId)) {
                        return res.status(403).json({ error: "Only admins can view requests"
                } else {
                        // If the user is an admin, populate requests and return them
                        await group.populate("requests");
                        res.status(200).json(group.requests);
                }
        } catch (error) {
                res.status(500).json("Cannot get group join request: ", error)
        }
};
```

Figure 264Get Group Requests (for Admin)

The getGroupRequests function allows groups admins to view join requests. It validates the group ID and checks if the user is an admin of the group. If so, it populates and returns the list of join requests.

Create a New Group

```
const createGroup = async (req, res) => {
        const { name } = req.body;
        const userId = req.user._id;

        try {
                // Check if the name is provided
                if (!name) {
                        return res.status(400).json({ error: "Name is required" });
                }

                // Create the group
                const group = await groupModel.create({ name });

                // Add user to the group member and admin
                await groupModel.findByIdAndUpdate(group._id, {
                        $push: { members: userId, admins: userId },
                });

                // Add group to user
                await userModel.findByIdAndUpdate(userId, {
                        $push: { groups: group._id },
                });

                res.status(200).json({ message: "Group created successfully", group });
        } catch (error) {
                res.status(500).json("Cannot get create group: ", error)
        }
};
```

Figure 276Create Group

The createGroup function allows a user to create a new group. It requires a group name and automatically adds the user as both a member and an admin of the new group. The group is also added to the user list of joined groups.

Request to Join a Group

```
const requestJoinGroup = async (req, res) => {
        const userId = req.user._id;
        const groupId = req.params.id;

        // Check if the ID is valid
        if (!mongoose.isValidObjectId(groupId)) {
                return res.status(404).json({ error: "Incorrect ID" });
        }

        try {
                // Find the group
                const group = await groupModel.findById(groupId);
                if (!group) {
                        return res.status(404).json({ error: "Group not found" });
                }

                // Check if the user is already a member
                if (group.members.includes(userId)) {
                        return res
                                .status(400)
                                .json({ error: "You are already a member of this group" });
                }

                // Check if a request has already been sent
                if (group.requests.includes(userId)) {
                        return res
                                .status(400)
                                .json({ error: "You have already sent a join request to this group" });
                }

                // Send join request
                await groupModel.findByIdAndUpdate(groupId, {
                        $push: { requests: userId },
                });

                res.status(200).json({ message: "Join request sent successfully", group });
        } catch (err) {
                // handle the error
                console.error(err);
                res.status(500).send("Error sending join request");
        }
};
```

Figure 287Request Join Group

The requestJoinGroup function allows a user to request to join a group. It check if the user is already a member or has already sent a request. If not, the user request is added to the group list of join request.

Approve Join Request

```
const approveJoinGroup = async (req, res) => {
        const userId = req.user._id;
        const groupId = req.params.id;
        const requestId = req.params.requestId;

        // Check if the ID is valid
        if (
                !mongoose.isValidObjectId(groupId) ||
                !mongoose.isValidObjectId(requestId)
        ) {
                return res.status(404).json({ error: "Incorrect ID" });
        }

        try {
                // Find the group
                const group = await groupModel.findById(groupId);
                if (!group) {
                        return res.status(404).json({ error: "Group not found" });
                }

                // Find the request
                const request = group.requests.find(
                        (request) => request.toString() === requestId
                );
                if (!request) {
                        return res.status(404).json({ error: "Request not found" });
                }

                // Check if approver is an admin
                if (!group.admins.includes(userId)) {
                        return res.status(403).json({ error: "Only admins can approve requests" });
                } else {
                        // Approve the request
                        group.members.push(requestId);
                        group.requests.splice(group.requests.indexOf(requestId), 1);
                        await group.save();

                        // Add group to user
                        await userModel.findByIdAndUpdate(requestId, {
                                $push: { groups: groupId },
                        });

                        res.status(200).json({ message: "Request approved successfully", group });
                }
        } catch (error) {
                res.status(500).json("Cannot approve join request: ", error)
        }
};
```

Figure 298Approve Join Request

The approveJoinGroup function enables group admins to approve join requests. It checks that the request exists and that the approver is an admin, then adds the requesting user to the group members, removes the request, and updates the the user group list.

Delete User From Group

Figure 309Delete Member

The deleteMemberFromGroup function allows an admin to remove a user from a group. It validates the IDs, ensures the user is an admin, and then removes the target user from both the group members and the user joined groups.

Post Controller Implementation

Create new Post

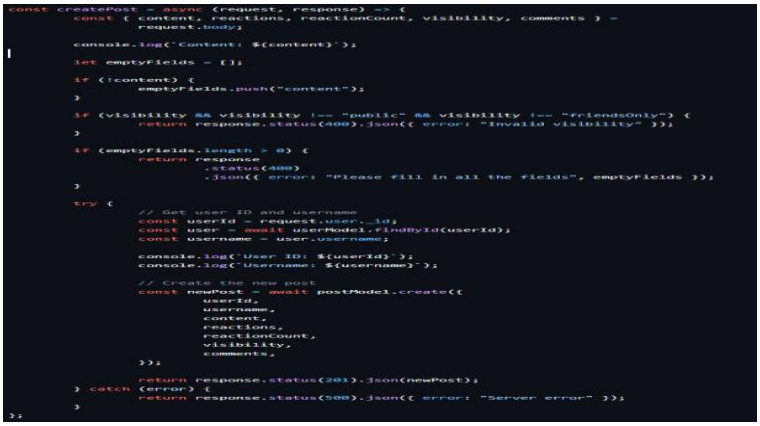

Figure 310Create new Post function

This function creates a new post using data from the request body, checks for valid input, and retrieves the current user ID and username to associate with the post. The created post is then saved to the database

Get All Post

```
// Get all posts
const getPosts = async (request, response) => {
        try {
                const posts = await postModel.find({}).select("-oldVersions").exec();

                for (let i = posts.length - 1; i >= 0; i--) {
                        const j = Math.floor(Math.random() * (i + 1));
                        [posts[i], posts[j]] = [posts[j], posts[i]];
                }

                response.status(200).json({ status: "success", data: posts });
        } catch (error) {
                response
                        .status(500)
                        .json({ status: "error", message: "Failed to retrieve posts" });
        }
};
```

Figure 321Get all Post function

This function retrieves all posts from the database, shuffles the order randomly, and then returns them in the response.

6. Project Testing

Comprehensive testing was conducted to ensure the system reliability, security, and performance. The testing process began with unit testing, where individual components such as user registration, post creation, and comment handling were thoroughly tested to verify their correctness. This was followed by integration testing, which focused on the interaction between the frontend and backend, ensuring smooth communication and consistent data handling, particularly for actions like post creation and notification handling.

To simulate real-world scenarios, end-to-end testing was performed, covering the entire workflow from user registration to group interactions and notifications reception. Special attention was given to offline mode, where testing ensured that user actions were properly cached and synchronized with the server once connectivity was restored. Finally, security testing was conducted to identify and mitigate vulnerabilities, including cross-site scripting and session hijacking, safeguarding user data and ensuring the system overall integrity.

7. Project Process

Project Planning

Week	**Date Range**	**Task Description**	**Team Member**	**Responsibilities**
1	Aug 5 - Aug 11	Setup and Initial Planning	Lam, Thinh	Set up the project repository, define coding standards,

				prepare development environments.
			Vu, Phuc, Dong	Draft initial project documentation, including scope and requirements.
2	Aug 12 - Aug 18	Coding Core Functionalities	Thinh	Develop User Authentication module (login, registration, session management for users and site admins).
			Dong	Set up Backend Framework and Database Schema (MongoDB setup, Express routes).
			Lam	Design Friend Request and Management System.
			Vu	Begin work on Group Management System (creation, membership rules, admin controls).
			Phuc	Implement basic Post and Comment functionalities.
3	Aug 19 - Aug 25	Continue Development and Start Integrations	Lam	Implement encryption and secure access for user data.
			Thinh	Develop Blog Posting features (CRUD operations for blogs, admin moderation tools
			Dong	Enhance the Friend System with notifications and interactive features, admin oversight on disputes.
			Vu	Expand Group Management System with admin functionalities and

				settings (including admin group controls).
			Phuc	Enhance Post System with reaction options, further comment capabilities, and admin moderation tools.
4	Aug 26 - Sep 1	Testing and Debugging	Vu	Test all functionalities (unit testing, integration testing).
			Lam, Thinh	Focus on fixing backend issues, ensuring database integrity, and testing admin functionalities.
			Dong, Phuc	Handle frontend bugs, improve user interface interactions, including admin dashboards.
5	Sep 2 - Sep 8	Finalization and Documentation	Everyone	Conduct final testing and polish all features, including comprehensive admin features.
			Lam, Thinh	Prepare the final project report detailing development processes, architecture, usage, and admin functionalities.
			everyone	Create a video demonstration of the platform, highlighting key features, user and admin workflows.
Final Day	Sep	Project Submission	Vu	Review the final report and video demo. Submit the final project package to the course supervisor. Conduct a retrospective meeting to discuss overall project.

8. Conclusion

In conclusion, the development of Culture Canvas by Group Project 13 has been a transformative experience in our full-stack development course. This project allowed us to apply the MERN stack—MongoDB, Express, React, and Node.js—enhancing our understanding of web development in a practical setting.

Through building Culture Canvas, we tackled key features of a modern social networking platform such as user and group management, interactive posts, and an extensive notification system. These tasks challenged us to effectively manage both MySQL and MongoDB databases, boosting our data handling skills.

Working collaboratively, we sharpened our teamwork, communication, and problem-solving abilities, tackling challenges such as secure authentication and maintaining functionality during network disruptions. This project was not only about enhancing user interaction but also served as a practical application of our theoretical knowledge.

We are thankful for this hands-on experience, which has prepared us for more complex projects and helped us grow as full-stack developers, ready to adapt to the evolving tech landscape.