-

Subject Name: **Backend Engineering**

Subject Code: CS186

Cluster: iGamma

Department: **DCSE**

Group: **G19**



Project Name: VibeSync

**Submitted By:**

* Vanshika 2110991959
* Shivaksh Sharma 2110992079
* Anant Sharma 2110991978
* Sakhib Ali 2110992002
* Hritik Mishra 2110991960

# Submitted To:

Mr. Ashwani Dubey

# VIBESYNC

## Introduction

Welcome to the documentation of VibeSync Event Gallery, a React-based web application designed for a social media MERN (MongoDB, Express, React, Node.js) app called Vibesync. This Event Gallery is crafted to showcase and manage a collection of images, providing users with an immersive and visually appealing experience.

## Technologies Used:

* Nodejs
* MongoDB
* CSS
* JavaScript
* React

## Key Components

## Front-end

### FlexBetween.jsx

### Core Component: "FlexBetween" is a core component crafted using Material-UI's `Box` and `styled` utility.

### Layout Flexibility: Creates a flexible container with a space-between layout along the main axis.

### Content Spacing: Positions content with equal space between elements, enhancing visual appeal.

### Alignment: Centers content along the cross-axis, ensuring a balanced and visually pleasing layout.

### Simple Integration: Easily integrated into React components for streamlined development of flexible and well-spaced layouts.

### Friend.jsx

* Represents an individual friend entry with name, subtitle, and profile image.
* Allows navigation to the friend's profile and supports adding or removing friends.
* Utilizes Redux for managing user data, including the friend list.
* Implements dynamic styling for visual feedback and Material-UI components for a consistent UI.
* Performs PATCH requests to the server for adding or removing friends, updating the Redux store accordingly.

### UserImage.jsx

* The UserImage component is designed to display a user's profile image with optional customization of size.
* Fetches the user's profile image from the server using the provided image prop.
* Renders a box containing an image, ensuring a consistent size and shape for user profile pictures.

### WidgetWrapper.jsx

### The WidgetWrapper component is a styled container designed to wrap and stylize widgets or components.

### Wrap your desired content or components with WidgetWrapper to apply the defined styling.

### HomePage

### The HomePage component represents the main content page of the application.

### Utilizes Material-UI's Box for structural elements and useMediaQuery for responsiveness.

### Displays a Navbar component at the top.

### Retrieves user data, including user ID and profile picture path, using useSelector from the Redux store.

### Adjusts the layout based on screen size using conditional rendering and the useMediaQuery hook.

### On non-mobile screens (min-width: 1000px), displays additional widgets for advertisements and the friend list.

### LoginPage

### The Form component manages user authentication with dynamic switching between login and registration forms.

### Utilizes Formik and Yup for form handling and validation.

### Dynamically switches between login and registration schemas based on the current form type.

### Adapts layout and styling based on screen width using Material-UI's use MediaQuery for enhanced responsiveness.

### Sends registration or login requests to the server based on the current form type.

### Dispatches user data to the Redux store upon successful login, enabling navigation to the home page.

### ProfilePage

### ProfilePage renders user-specific information and widgets for a given userId.

### Utilizes the fetch API to retrieve user data from the server using the provided userId and the Redux store's token.

### Employs Material-UI's Box and useMediaQuery for a responsive layout, adjusting based on screen size.

### Includes widgets such as UserWidget, FriendListWidget, MyPostWidget, and PostsWidget to present user-related information and posts.

### State(index.jsx)

### authSlice defines an initial state for the Redux store with properties for the theme mode, current user, token, and an array of posts.

### setMode: Toggles between "light" and "dark" modes in the application.

### setLogin: Sets the current user and token upon successful login.

### setLogout: Resets user and token to null upon logout.

### setFriends: Updates the user's friend list based on the provided payload.

### setPosts: Sets the array of posts in the Redux store.

### setPost: Updates a specific post within the array based on the provided payload.

### These reducers can be dispatched to update the state throughout the application. Example: dispatch(setLogin({ user: userData, token: authToken }))

### Import and use the provided actions (setMode, setLogin, etc.) in your React components or middleware to interact with the Redux store.

### Index.jsx

### Root Rendering: Utilizes ReactDOM.createRoot to render the React application into the root element with the ID "root" in the HTML document.

### Redux Store Configuration: Configures the Redux store using configureStore from @reduxjs/toolkit.

### Integrates redux-persist to persist and rehydrate the state, enhancing user experience across sessions.

### Redux Persist Configuration: Defines a persistConfig object with key "root," using redux-persist to store the state in localStorage.

### Creates a persistedReducer by wrapping the original authReducer with the persistReducer function.

### Middleware Customization: Configures middleware to ignore specific actions during serialization using getDefaultMiddleware.

### Root Component: Wraps the App component with Provider from react-redux to provide the Redux store to the entire application.

### Utilizes PersistGate to delay rendering the App until the state is rehydrated from storage.

### CSS Styling

### The CSS code stylizes a React application by centering its content, applying a spinning animation to the logo, and styling the header with a dark background. Font size dynamically adjusts, and links have a distinctive color. The design prioritizes visual appeal, responsiveness, and accommodates user preferences for reduced motion.

## Back-end

### auth.js

### User Registration: The `register` function handles user registration by securely hashing the password using bcrypt. It creates a new user instance with provided details and saves it to the database, including random profile statistics.

### Password Hashing: Utilizes bcrypt to generate a salt and hash the user's password, ensuring the security of stored credentials.

### Login Authentication: The `login` function authenticates users by checking the existence of the provided email and validating the password using bcrypt. Upon successful authentication, it generates a JSON Web Token (JWT) containing the user's ID.

### JWT Token Generation: JWT.sign is used to create a token containing the user's ID, providing a secure means of authentication and authorization.

### Error Handling: Implements error handling to respond appropriately to registration or login failures, ensuring consistent and informative responses.

### posts.js

* Create Post: The `createPost` function creates a new post with user details, description, and an optional picture. It retrieves the user details by the provided userId, saves the post, and returns all posts in the response.
* Get Feed Posts: The `getFeedPosts` function retrieves all posts from the database and returns them in the response, forming the feed.
* Get User Posts: The `getUserPosts` function fetches posts specific to a user, identified by their userId, and returns them in the response.
* Update Post Likes: The `likePost` function toggles likes on a specific post identified by its id. It checks if the user has already liked the post, then adds or removes the like accordingly, updating the post in the database.
* Error Handling: Implements error handling to provide appropriate status codes and messages in case of successful or failed operations.

### Users.js

* Get User Profile: The `getUser` function retrieves user details based on the provided user id and responds with a JSON object containing user information.
* Get User Friends: The `getUserFriends` function fetches a user's friends by their id and formats the data before responding with an array of friend objects.
* Add/Remove Friend: The `addRemoveFriend` function toggles the friendship status between two users, identified by their respective ids. If the friend is already in the user's friend list, it removes them, and vice versa. The updated friend lists are then saved.
* Error Handling: Implements error handling to provide appropriate status codes and messages in case of successful or failed operations.
* Friend List Formatting: The controller formats the friend list data by extracting specific fields (e.g., \_id, firstName, lastName, occupation) for a more concise and relevant response.

### Middleware

### Get User Profile: The `getUser` function retrieves user details based on the provided user id and responds with a JSON object containing user information.

### Get User Friends: The `getUserFriends` function fetches a user's friends by their id and formats the data before responding with an array of friend objects.

### Add/Remove Friend: The `addRemoveFriend` function toggles the friendship status between two users, identified by their respective ids. If the friend is already in the user's friend list, it removes them, and vice versa. The updated friend lists are then saved.

### Error Handling: Implements error handling to provide appropriate status codes and messages in case of successful or failed operations.

### Friend List Formatting: The controller formats the friend list data by extracting specific fields (e.g., \_id, firstName, lastName, occupation) for a more concise and relevant response.

### Data

### Users Data Array: ‘users’ array contains sample user data with attributes like `\_id`, `firstName`, `lastName`, `email`, `password`, `picturePath`, `friends`, `location`, `occupation`, `viewedProfile`, `impressions`, `createdAt`, and `updatedAt`. It simulates user profiles with various details.

### Posts Data Array: `posts` array includes sample post data with attributes such as `\_id`, `userId`, `firstName`, `lastName`, `location`, `description`, `picturePath`, `userPicturePath`, `likes` (using a Map), and `comments`. It emulates user posts with associated information.

### ObjectId Instances: `userIds` array consists of instances of `mongoose.Types.ObjectId()` used as unique identifiers for users. Each element represents an individual user's `\_id`.

### Password Hashing: User passwords in the `users` array are hashed using bcrypt. Hashed passwords enhance security by storing user credentials securely in the database.

### Relationships and Associations: The data structure establishes relationships between users and posts using `userId`. Posts include references to the user who created them, creating an association between users and their posts. The `friends` array in the user data simulates friendship associations.

### Models / User.js

### User Schema Structure: The `UserSchema` defines the structure of the user document. It includes fields for `firstName`, `lastName`, `email`, `password`, `picturePath`, `friends`, `location`, `occupation`, `viewedProfile`, and `impressions`. These fields represent various attributes of a user.

### Validation Constraints: String fields like `firstName`, `lastName`, `email`, and `password` have defined constraints. They are marked as required and have specified minimum and maximum lengths to ensure data integrity and completeness.

### Default Values: The schema includes default values for `picturePath` (an empty string), `friends` (an empty array), `location`, `occupation`, `viewedProfile`, and `impressions`. Default values provide fallbacks for fields that might not have values initially.

### Timestamps: The schema is configured with `{timestamps: true}` to automatically generate `createdAt` and `updatedAt` timestamps for each document. This helps in tracking when a user document was created or last updated.

### 5. Model Creation: The `User` model is created using `mongoose.model()` by passing the name "User" and the defined `UserSchema`. This model can be used to interact with the MongoDB collection associated with users, providing CRUD operations and other functionalities.

### Post.js

### Post Schema Structure: The `postSchema` defines the structure of a post document. It includes fields such as `userId`, `firstName`, `lastName`, `location`, `description`, `picturePath`, `userPicturePath`, `likes`, and `comments`. These fields capture essential information about a user's post.

### Field Types: `userId`, `firstName`, and `lastName` are of type String, while `location`, `description`, `picturePath`, and `userPicturePath` are also strings. `likes` is defined as a Map with keys of type string and values of type boolean, representing user likes. `comments` is an array capturing post comments.

### Required Fields: Fields such as `userId`, `firstName`, `lastName`, and `description` are marked as required, ensuring that essential information is present when creating a post. This helps maintain data consistency.

### Default Values: The schema includes default values for `likes` (an empty Map) and `comments` (an empty array). This provides initial values for these fields, ensuring they exist even if not explicitly provided during post creation.

### Timestamps: `{timestamps: true}` is set for the schema, enabling automatic generation of `createdAt` and `updatedAt` timestamps for each post document. This aids in tracking when a post was created or last updated.

### Routes (auth.js)

### Express Router Setup: The express library is used to create an instance of a router, and it's assigned to the variable router. This router will be used to handle login-related routes.

### Route Definition: The router has one route defined using router.post("/login", login). This route is set to handle HTTP POST requests to the "/login" endpoint.

### Controller Function: The login function from the "../controllers/auth.js" file is specified as the callback function to handle requests to the "/login" endpoint. The login function likely contains the logic for user authentication and generating a token upon successful login.

### Export: The router is exported as the default export of the module, making it available for use in other parts of the application.

### Routes (posts.js)

### Express Router Setup: The express library is used to create an instance of a router, assigned to the variable router.

### Middleware Import: The verifyToken middleware is imported from "../middleware/auth.js". This middleware is likely responsible for verifying the authenticity of a user token.

### Routes Defined: The router has three routes defined:

### GET /: This route, when accessed with a valid token (using the verifyToken middleware), triggers the getFeedPosts function from "../controllers/posts.js". It is expected to retrieve posts for the feed.

### GET /:userId/posts: This route, when accessed with a valid token, triggers the getUserPosts function from "../controllers/posts.js". It is expected to retrieve posts for a specific user based on the provided userId.

### PATCH /:id/like: This route, when accessed with a valid token, triggers the likePost function from "../controllers/posts.js". It is expected to handle the liking/unliking of a post based on the provided id.

### Middleware Usage: The verifyToken middleware is applied to routes that require token verification, ensuring that only authenticated users can access these routes.

### Export: The router is exported as the default export of the module, making it available for use in other parts of the application.

### Routes (users.js)

### Express Router Setup: The express library is used to create an instance of a router, assigned to the variable router.

### Middleware Import: The verifyToken middleware is imported from "../middleware/auth.js". This middleware is likely responsible for verifying the authenticity of a user token.

### Routes Defined: The router has three routes defined:

### GET /:id: This route, when accessed with a valid token (using the verifyToken middleware), triggers the getUser function from "../controllers/users.js". It is expected to retrieve information about a specific user based on the provided id.

### GET /:id/friends: This route, when accessed with a valid token, triggers the getUserFriends function from "../controllers/users.js". It is expected to retrieve the friends of a specific user based on the provided id.

### PATCH /:id/:friendID: This route, when accessed with a valid token, triggers the addRemoveFriend function from "../controllers/users.js". It is expected to add or remove a friend for a specific user based on the provided id and friendID.

### Middleware Usage: The verifyToken middleware is applied to routes that require token verification, ensuring that only authenticated users can access these routes.

### Export: The router is exported as the default export of the module, making it available for use in other parts of the application.

### Index.jsx

### Imports: Import necessary libraries such as express, body-parser, mongoose, cors, dotenv, multer, helmet, morgan, path, and fileURLToPath.

### Express App Setup: Create an instance of the Express app (app).

### Configurations: Use various middlewares and configurations:

### express.json() to parse JSON data.

### helmet() and helmet.crossOriginResourcePolicy for security headers.

### morgan("common") for HTTP request logging.

### bodyParser for handling JSON and form data.

### cors for enabling Cross-Origin Resource Sharing.

### Serve static files from the "public/assets" directory for the "/assets" route.

### Configure environment variables using dotenv.

### File Storage Configuration (Multer): Set up Multer for handling file uploads. Define storage settings for user profile pictures and post pictures.

### Routes with Files: Define routes that handle file uploads:

### POST "/auth/register": Register a user with an uploaded profile picture.

### POST "/posts": Create a post with an uploaded picture.

### Route Definitions: Use modular route handlers from separate files:

### "/auth": Routes related to authentication (imported from "authRoutes").

### "/users": Routes related to user data (imported from "userRoutes").

### "/posts": Routes related to posts (imported from "postRoutes").

### Mongoose Setup: Connect to the MongoDB database using mongoose.connect. Start the server on the specified port (PORT) once the database connection is successful. There are commented-out sections (/\* ADD DATA ONE TIME \*/) for inserting sample data into the MongoDB collections (users and posts).

### Server Start: Listen on the specified port and log a message to the console when the server is running.

### .env

### 1. MongoDB Connection URL: The `MONGO\_URL` environment variable specifies the connection URL for MongoDB. It contains the necessary information to connect to your MongoDB database, including the username, password, cluster URL, and other options. Ensure that the URL is correctly formatted and points to your MongoDB database.

### 2. JWT Secret Key: The `JWT\_SECRET` environment variable holds a secret key used for JSON Web Token (JWT) signing and verification. JWTs are often employed for authentication and authorization purposes. Ensure that the secret key is strong and kept confidential to enhance the security of your application.

### 3. Server Port: The `PORT` environment variable determines the port on which your Express.js server will listen for incoming HTTP requests. In this case, it is set to `3001`. Make sure this port is available and not blocked by other processes. You can access your application by navigating to `http://localhost:3001` in your web browser.

### 4. MongoDB Atlas Cluster: The MongoDB connection URL appears to be configured for MongoDB Atlas, a cloud-based MongoDB service. Ensure that you have set up a MongoDB Atlas account, created a cluster, and replaced the placeholder values (username, password, cluster URL) in the connection URL with your actual credentials.

### 5. Security Measures: The code includes security measures such as using the `helmet` middleware for securing HTTP headers and configuring CORS (`cors`) to control cross-origin resource sharing. These measures enhance the overall security posture of your Express.js application. Ensure that these security features align with your application's requirements and policies.

## Overview:

## The Social Media App is a platform for users to connect, share posts, and interact with friends.

## Developed using MERN stack (MongoDB, Express, React, Node.js) and Redux for state management.

## Features:

## User Authentication:

## Register: Users can create an account with personal details.

## Login: Secure login using hashed passwords and JWT authentication.

## Token-based Auth: JWT tokens used to authenticate and authorize user actions.

## User Profiles:

## View User Profile: Users can view their profile and details.

## Friend List: Display friends of the user.

## View Others: Explore profiles of other users.

## Posts:

## Create Post: Users can create posts with text and optional images.

## Feed: View a feed of posts from friends.

## Like & Comment: Interact with posts through likes and comments.

## Friendship:

## Add/Remove Friends: Users can send/receive friend requests.

## Friend List: Display friends, and mutual friendships are updated in real-time.

## Frontend:

## React Components:

## Modular components for User Profile, Posts, and Friend Lists.

## Redux: State management for user data and posts.

## Backend:

## Express Server:

## Routes: Separated routes for authentication, users, and posts.

## Middleware: JWT verification to secure routes.

## MongoDB Database:

## User & Post Schemas: Defined structures for users and posts.

## Mongoose: Object Data Modeling (ODM) for MongoDB.

## File Storage:

## Multer: Handle file uploads for user profile pictures and post images.

## Security: Helmet & Morgan:

## Enhance security and log HTTP requests.

## Environment Variables: MongoDB URL, JWT Secret, and Port configured as environment variables.

## Deployment: Deployed on a server with MongoDB Atlas for database hosting.

## Development Setup: Clone the repository, install dependencies, and configure environment variables.

## Future Enhancements:

## Real-time Notifications: Implement WebSocket for real-time notifications.

## Additional Features: Explore options for additional features and improvements.