VIETNAM NATIONAL UNIVERSITY – HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

COLLABORATIVE SYSTEM

Course: Web Application Development
IT093IU

BY
Phan Trần Anh Quân – ITITIU23019
Nguyễn Minh Quân - ITITIU23035

Instructor:
Assoc. Prof. Nguyen Van Sinh
Msc. Nguyen Trung Nghia

Ho Chi Minh City, Vietnam, 2025

Table of Contents

I. INTRODUCTION

1. Background

Our project is called Collaborative System, it is a platform that combines the features Google Doc and Discord into one integrated system. Users can create projects and work together in real time by typing and editing documents, drawing on a shared whiteboard, and communicating through text messaging and voice calls within the same workspace. By bringing document collaboration and team communication into a unified environment, the system aims to improve productivity, reduce context switching, and support effective teamwork for academic and professional use.

2. Programming Language

Because this is a Web Application Development course, so our project is based on these programming languages:

a. HTML is used to define the interface of the webpages.

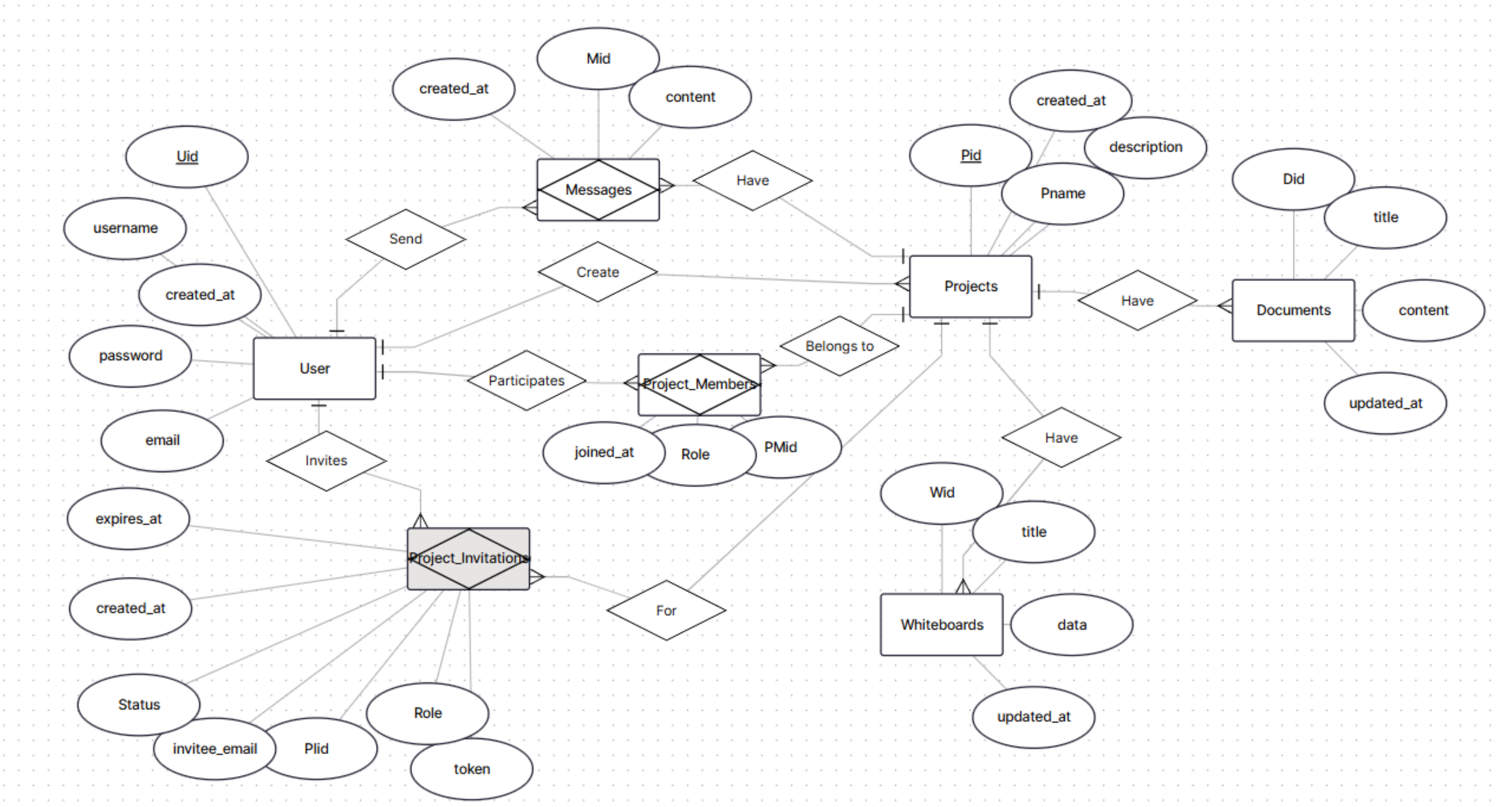
b. CSS is used to design the looks of the webpages

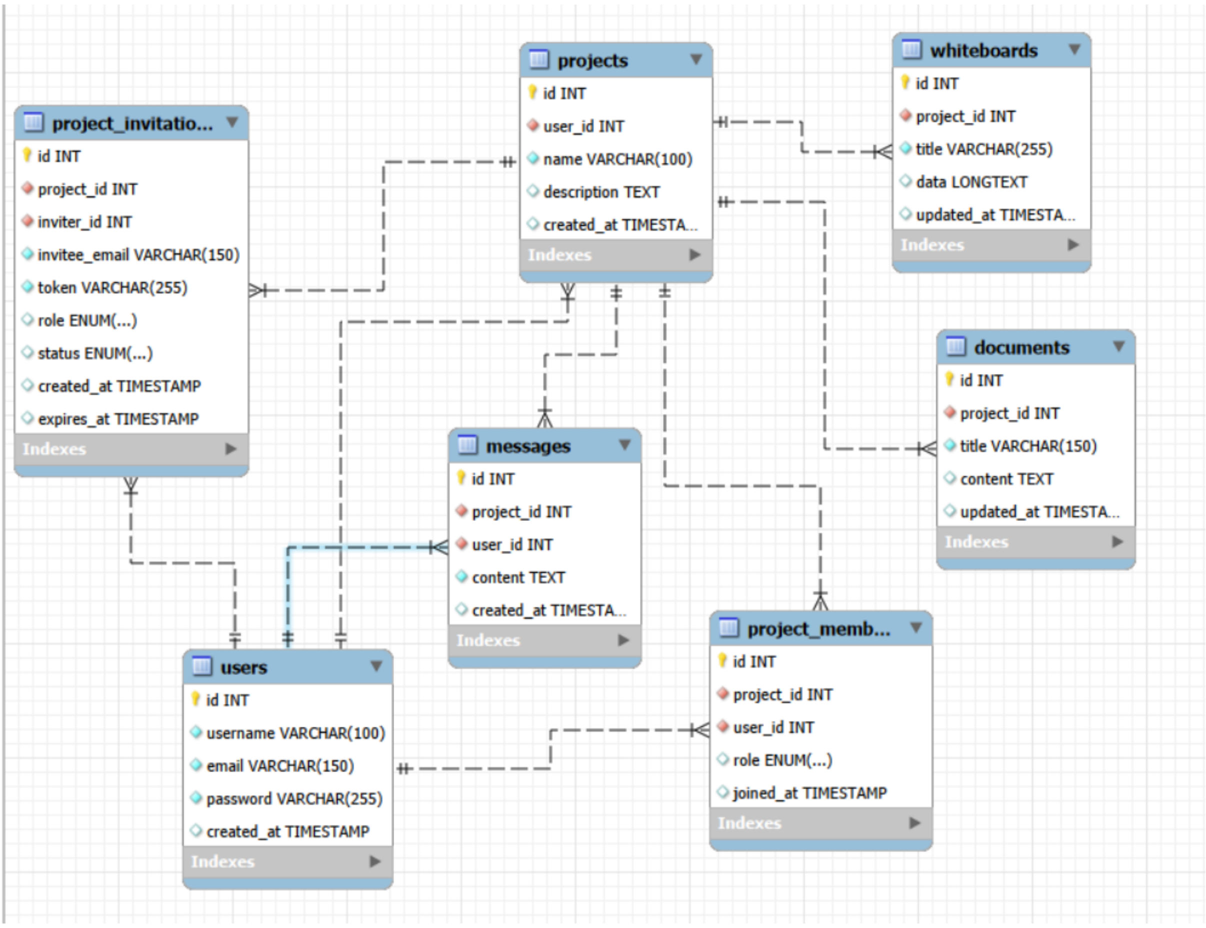
c. JavaScript is used to implement client-side logic, handle user interactions, and enable real-time collaboration features.

d. Node.js is used to build the backend server, manage APIs, and handle real-time communication between users.

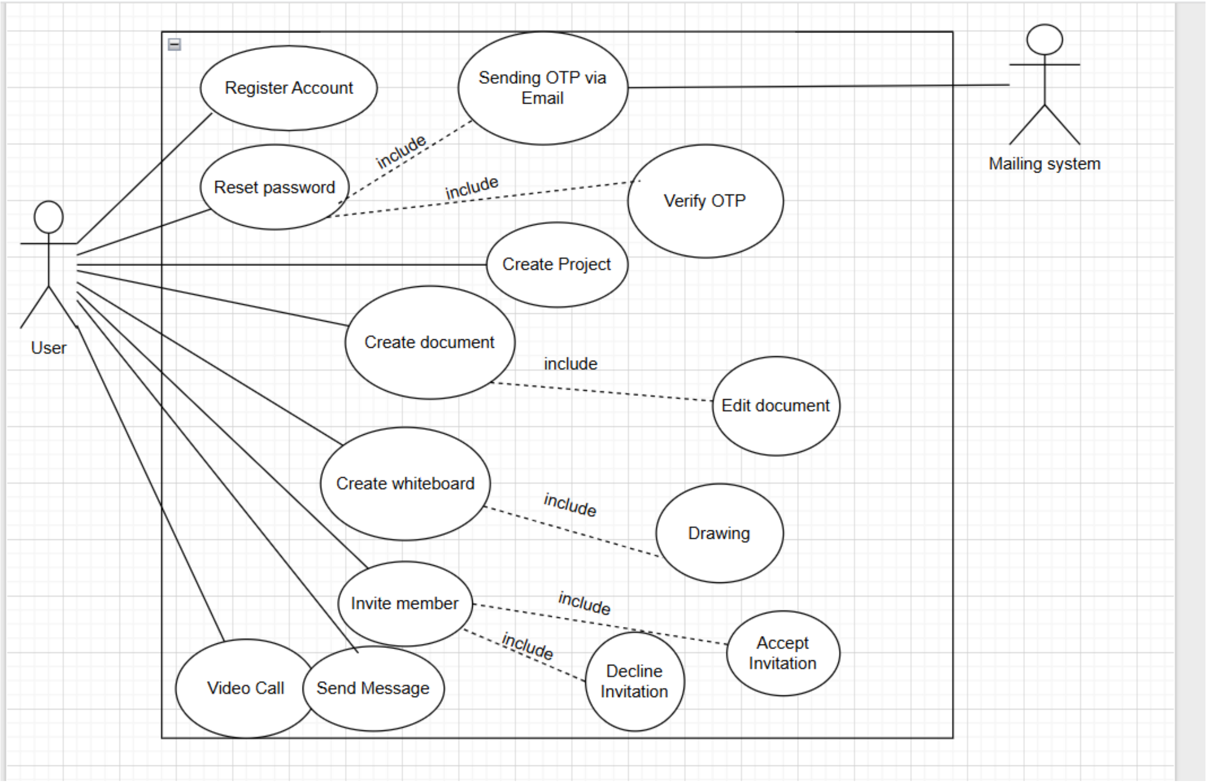
e. MySQL is used as the database system to store user data, project information, and collaboration content.

3. Entity-relationships Diagram (ERD)

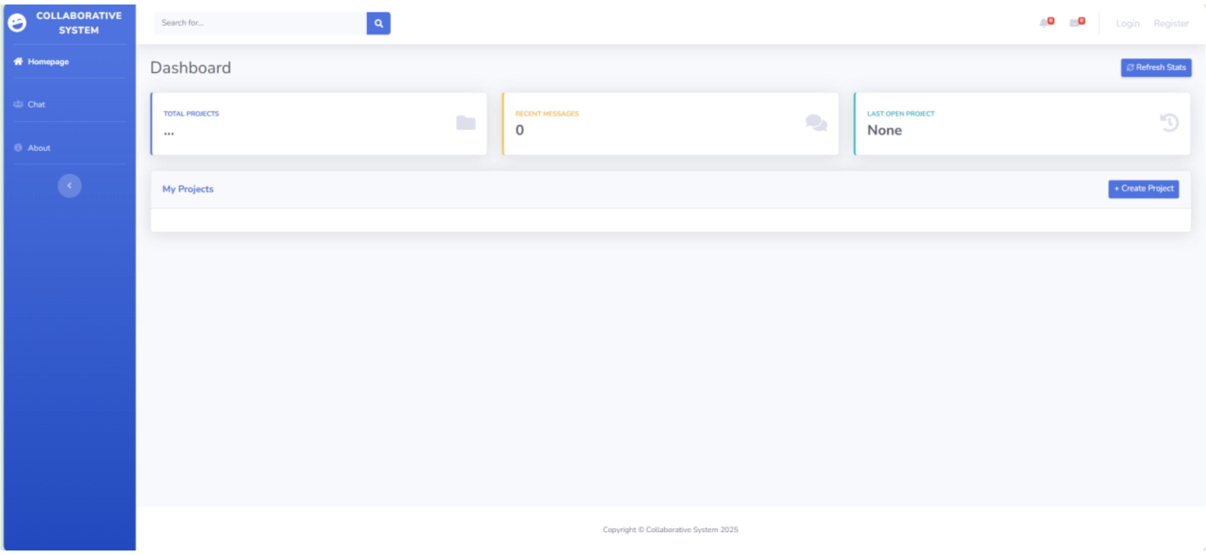




4. Use case diagram



II. Implementation



Dashboard

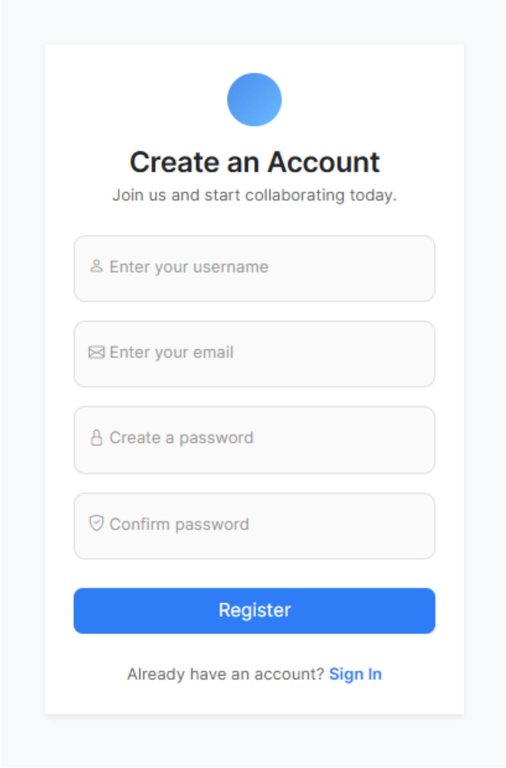
TOTAL PROJECTS
...

RECENT MESSAGES
0

LAST OPEN PROJECT
None

My Projects

Main interface (Before login)



Create an Account

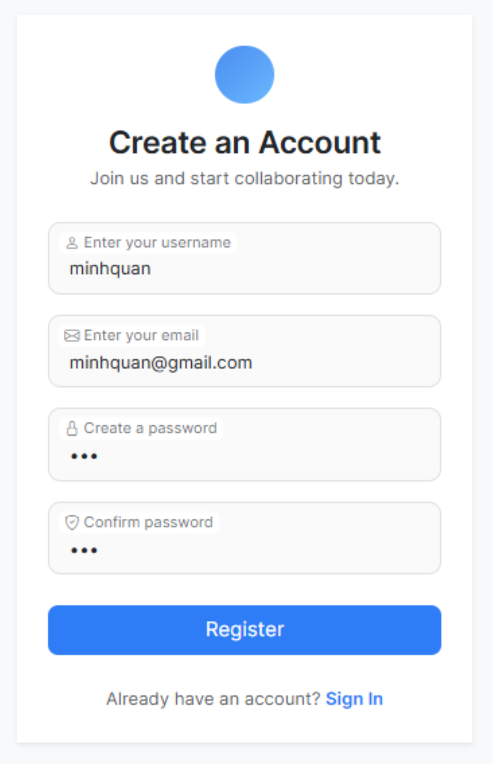
Join us and start collaborating today.

Enter your username

Enter your email

Create a password

Confirm password



Create an Account

Join us and start collaborating today.

Enter your username
minhquan

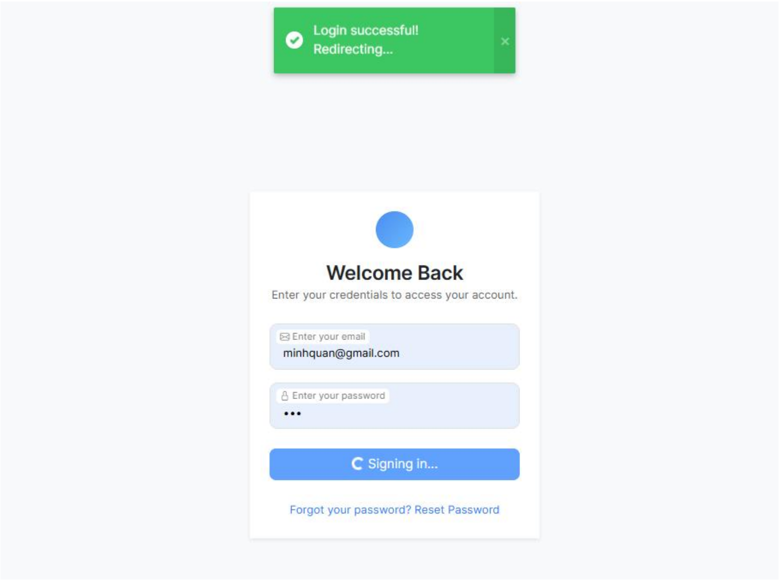
Enter your email
minhquan@gmail.com

Create a password
•••

Confirm password
•••

Register interface

Fill in the new user information


Login successful! Redirecting...

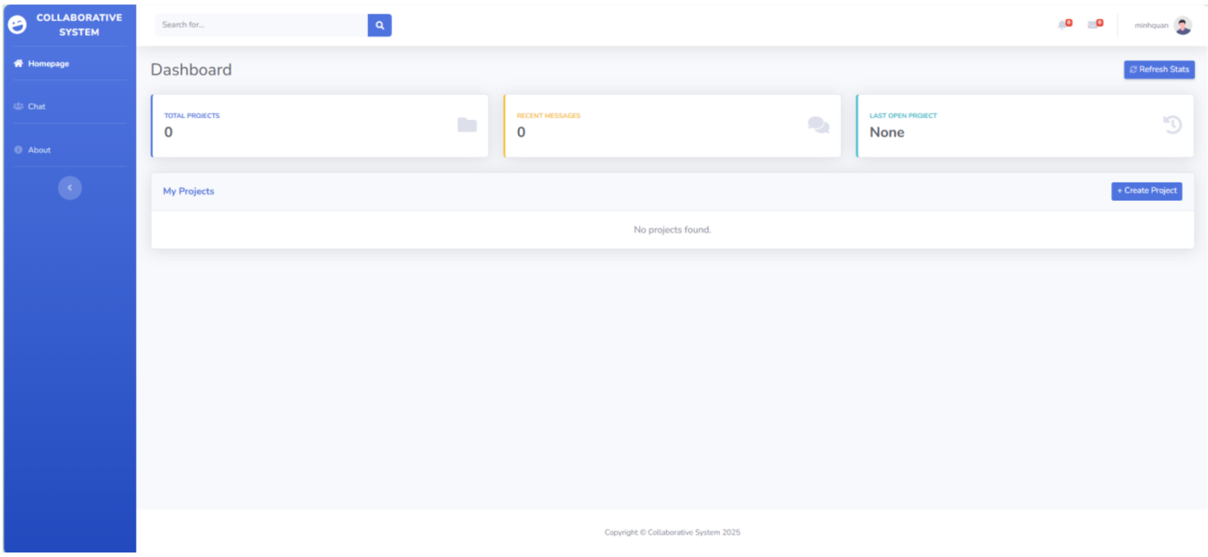
Welcome Back

Enter your credentials to access your account.

Enter your email: minhquan@gmail.com

Enter your password: •••

After registering successfully, it will move the user to the login interface.

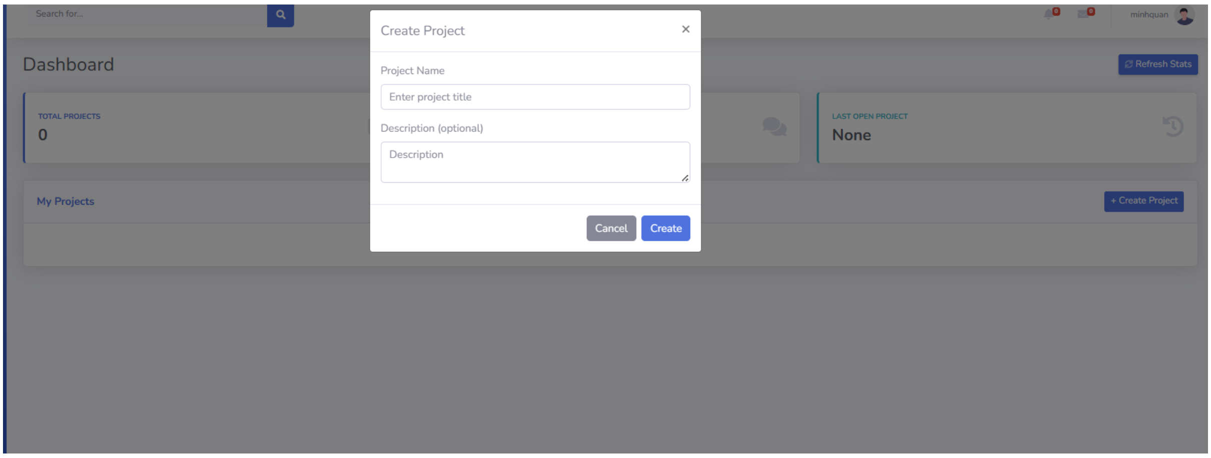


TOTAL PROJECTS	RECENT MESSAGES	LAST OPEN PROJECT
0	0	None

No projects found.

Copyright © Collaborative System 2025

The main interface after login successfully

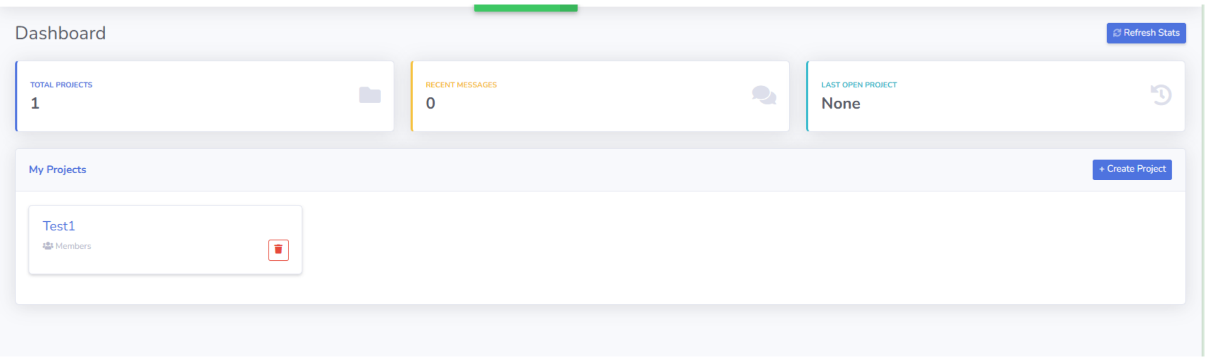


Create Project

Project Name: Enter project title

Description (optional): Description

Create a new project



Dashboard

TOTAL PROJECTS 1

RECENT MESSAGES 0

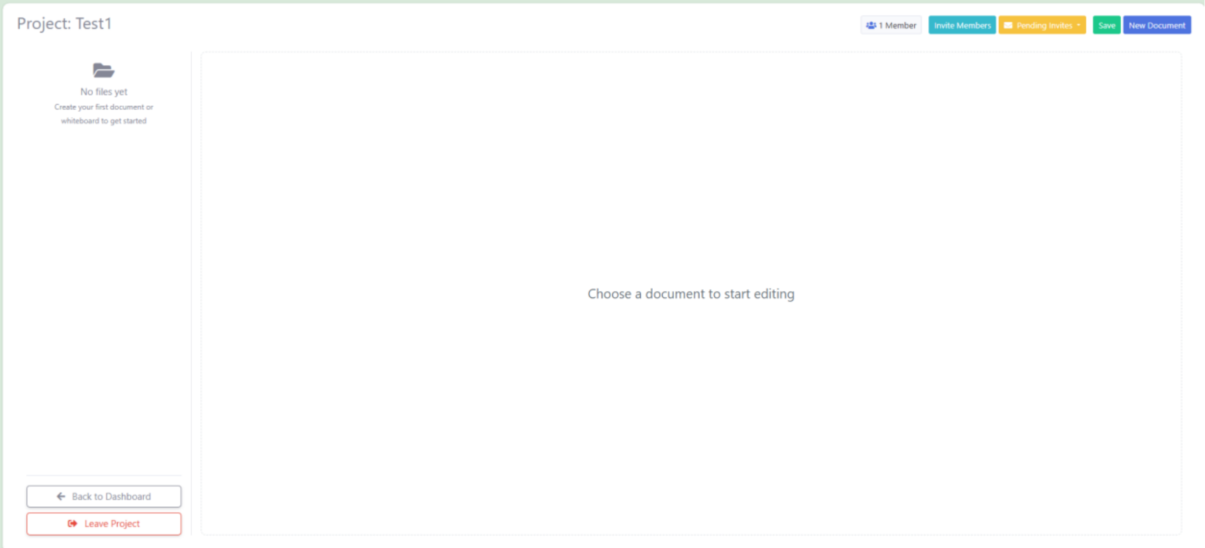
LAST OPEN PROJECT None

My Projects

Test1

Members

Creating Test1 successfully



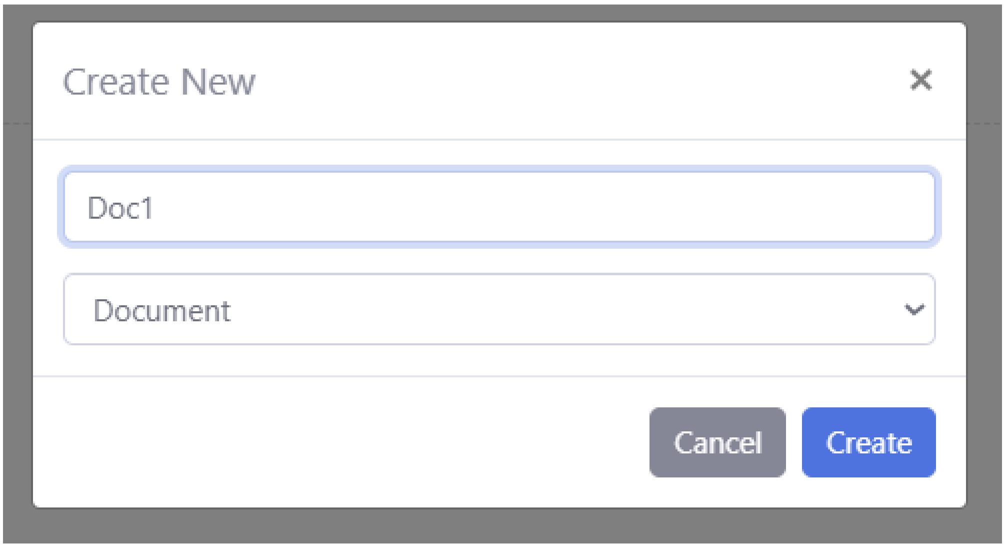
Project: Test1

No files yet

Create your first document or whiteboard to get started

Choose a document to start editing

After clicking on the new project

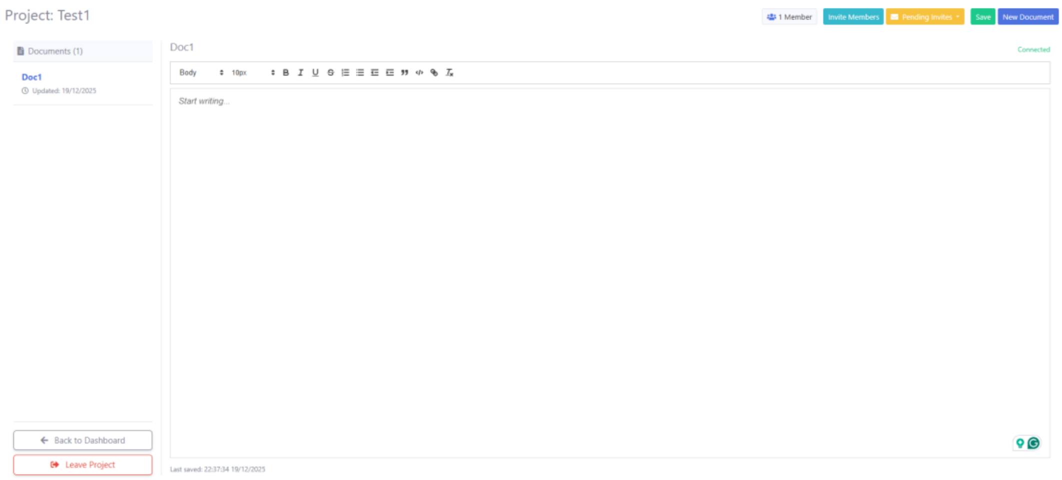


Create New

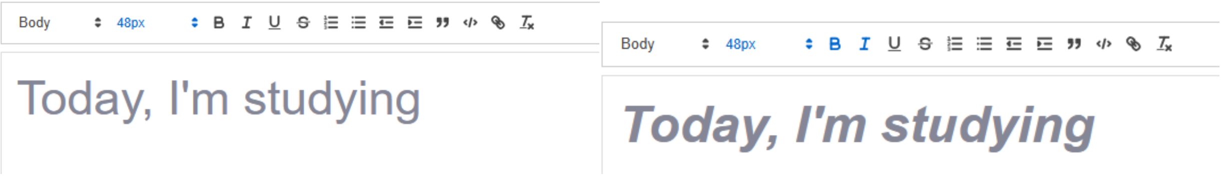
Doc1

Document

Create a new document file



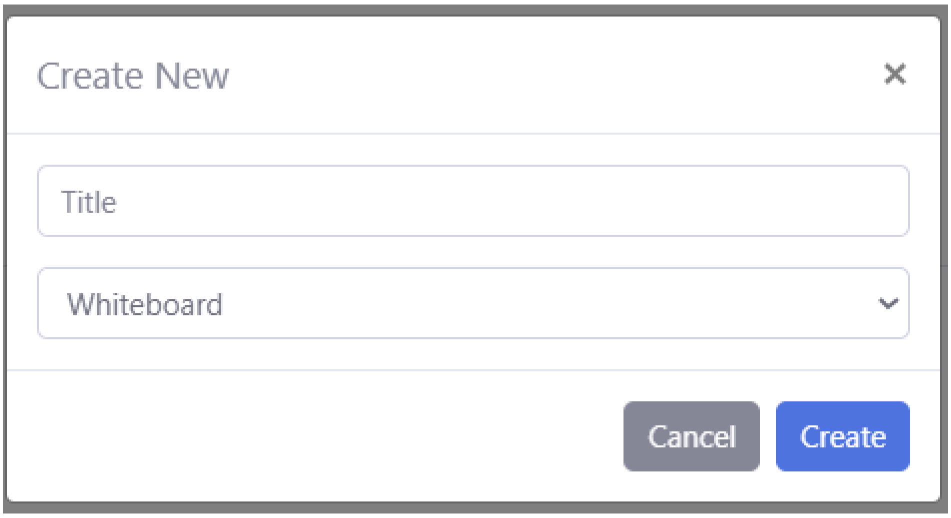
Creating successfully



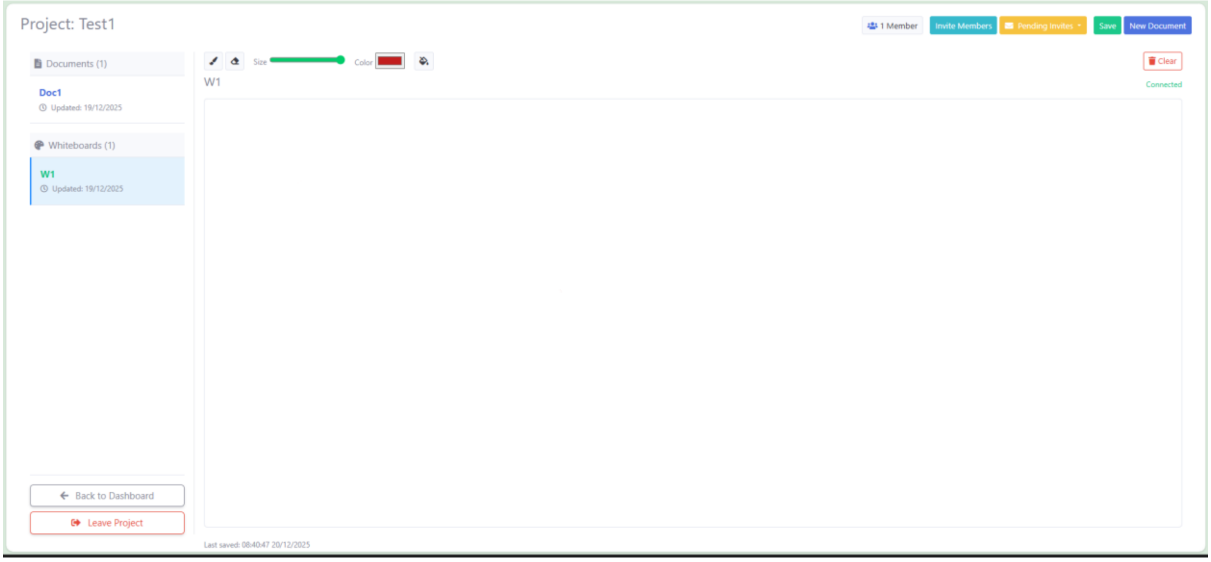
Today, I'm studying

Today, I'm studying

Changing the font, style of the document

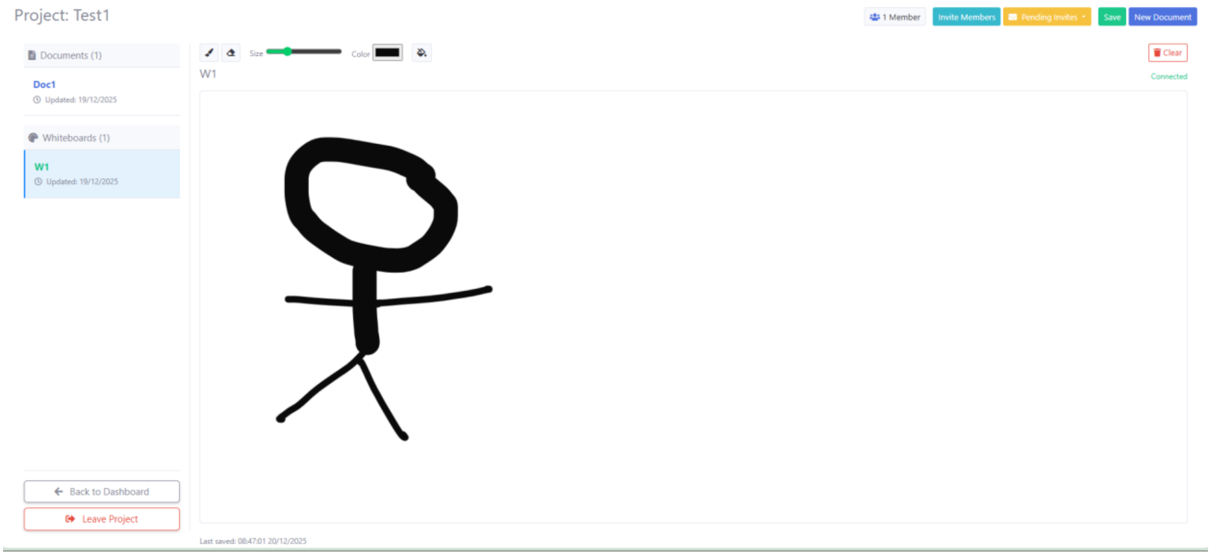


Create a new whiteboard



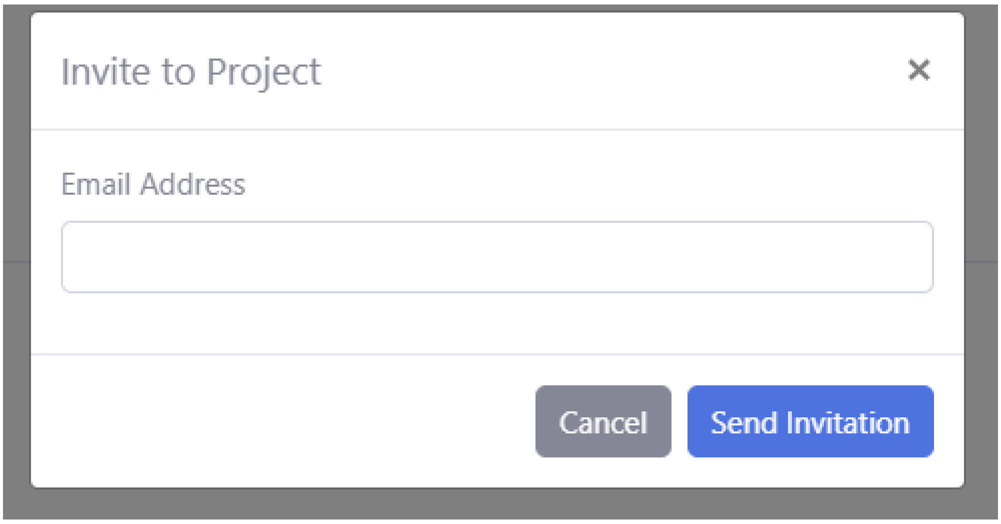
Last saved: 08:40:47 20/12/2025

Whiteboard interface

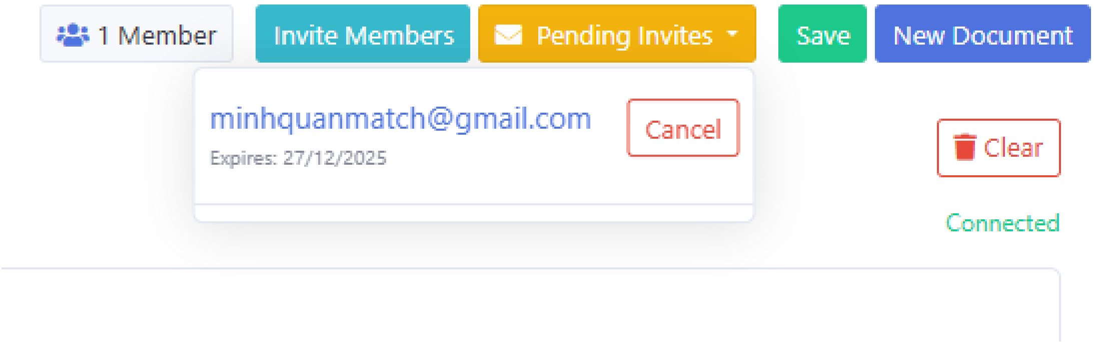


Last saved: 08:47:01 20/12/2025

Drawing



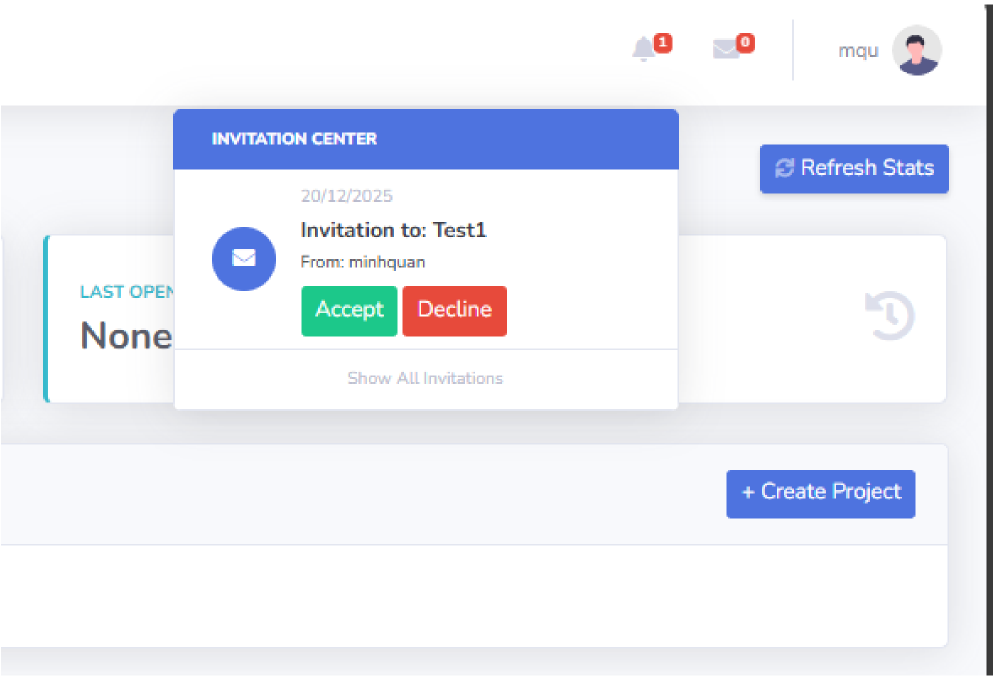
Invite another member to the project via email



minhquanmatch@gmail.com
Expires: 27/12/2025

Connected

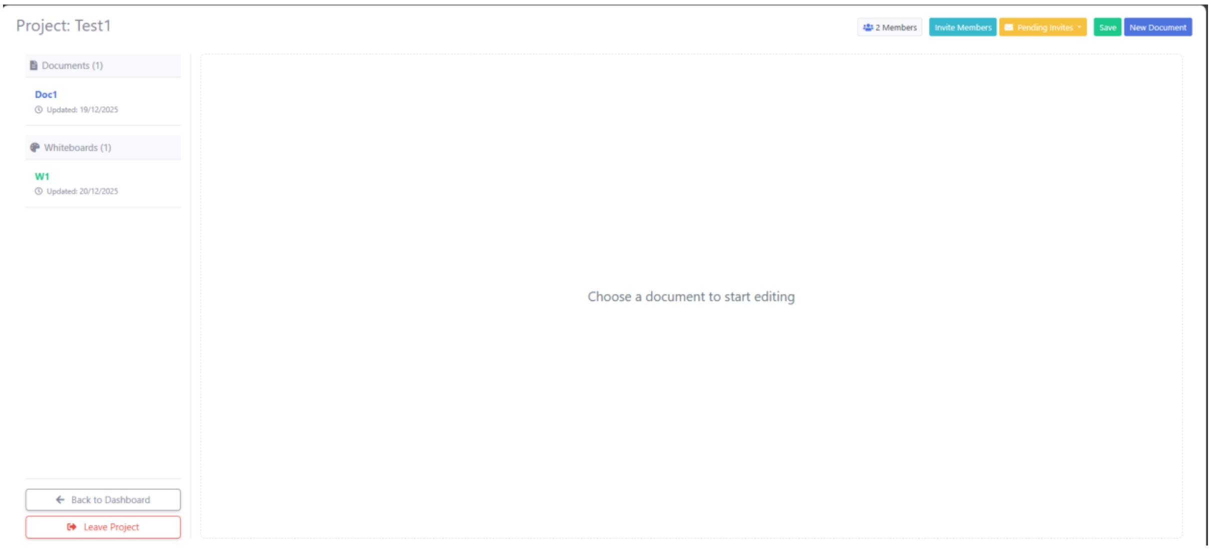
Invite another member to the project



20/12/2025
Invitation to: Test1
From: minhquan

LAST OPEN
None

The invitee’s notification



Project: Test1

Documents (1)

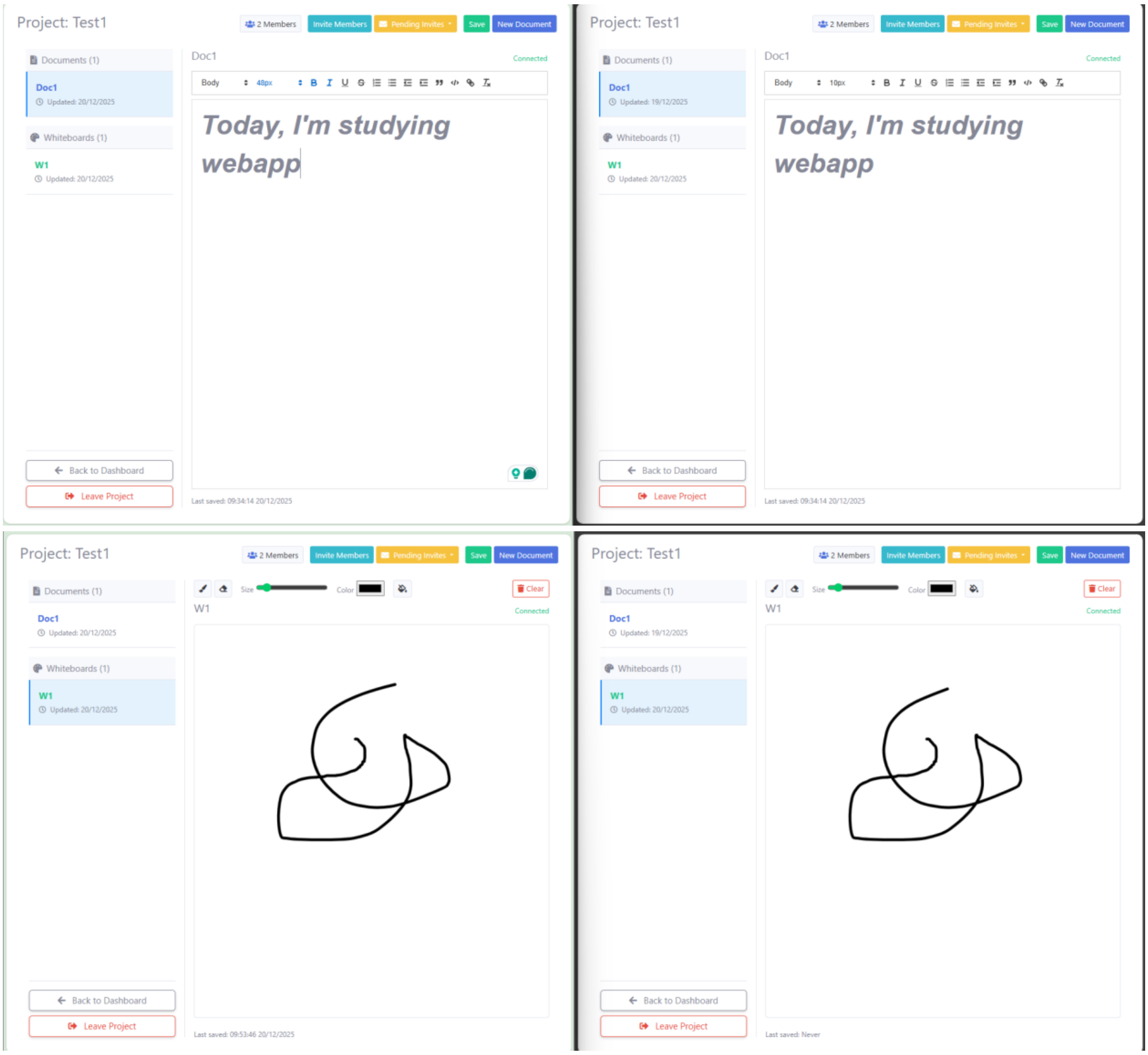
Doc1
Updated: 19/12/2025

Whiteboards (1)

W1
Updated: 20/12/2025

Choose a document to start editing

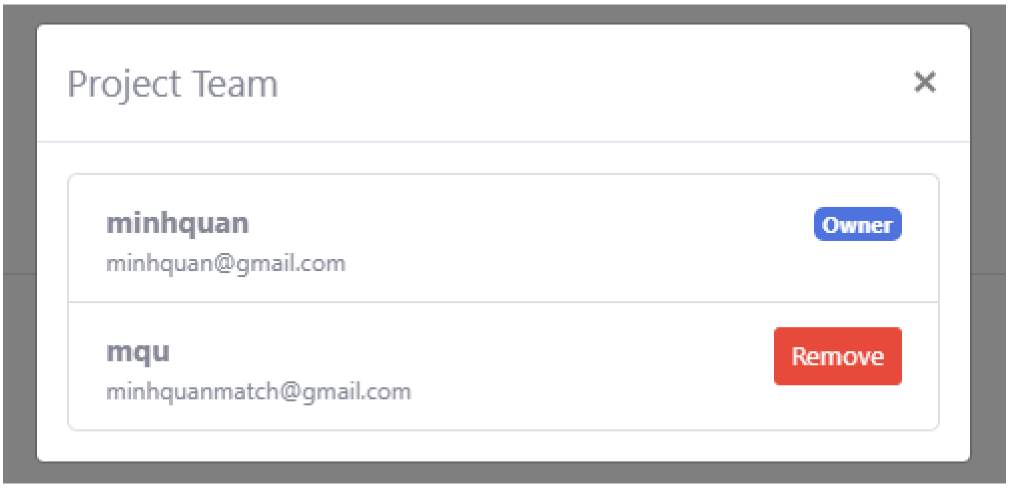
After accepting



Doc1

Today, I'm studying webapp

When joining the same project, the two members can see real-time synchronization

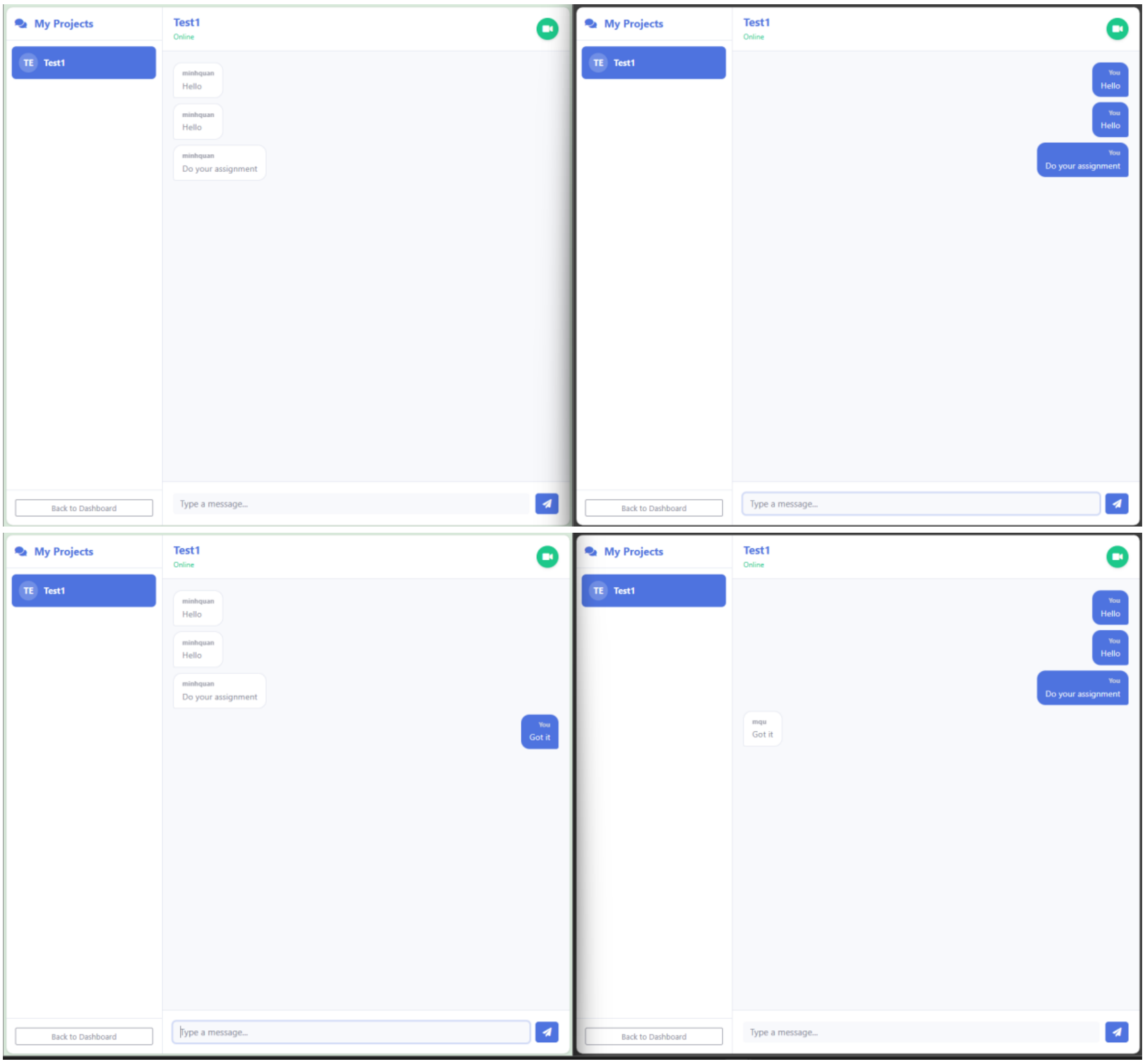


Project Team

minhquan
minhquan@gmail.com

mqu
minhquanmatch@gmail.com

The owner of the project can remove a member from the project



minhquan
Hello

minhquan
Hello

minhquan
Do your assignment

You
Hello

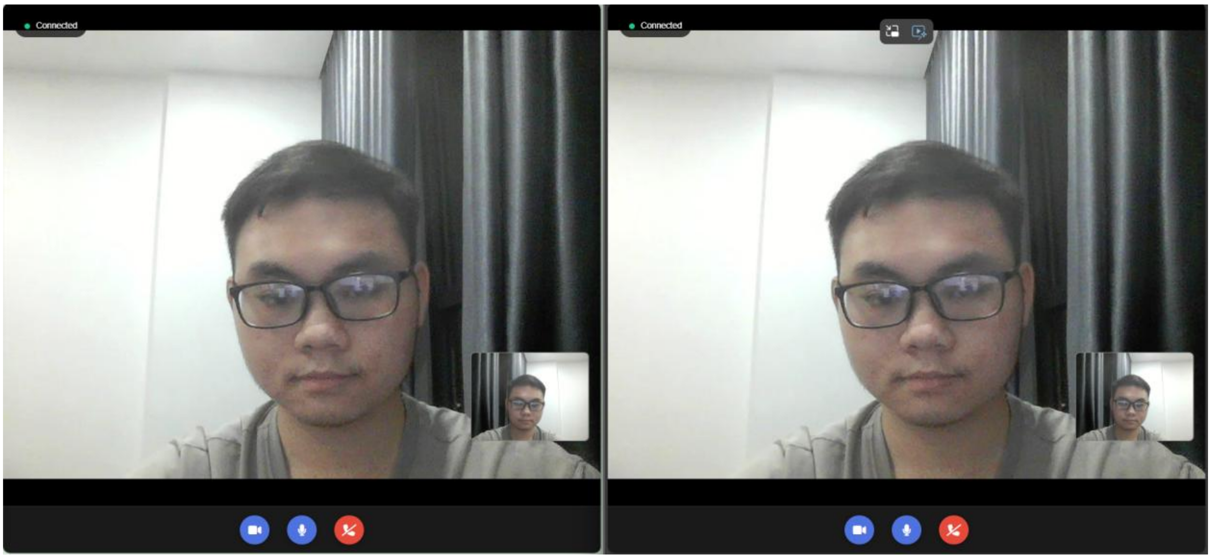
You
Hello

You
Do your assignment

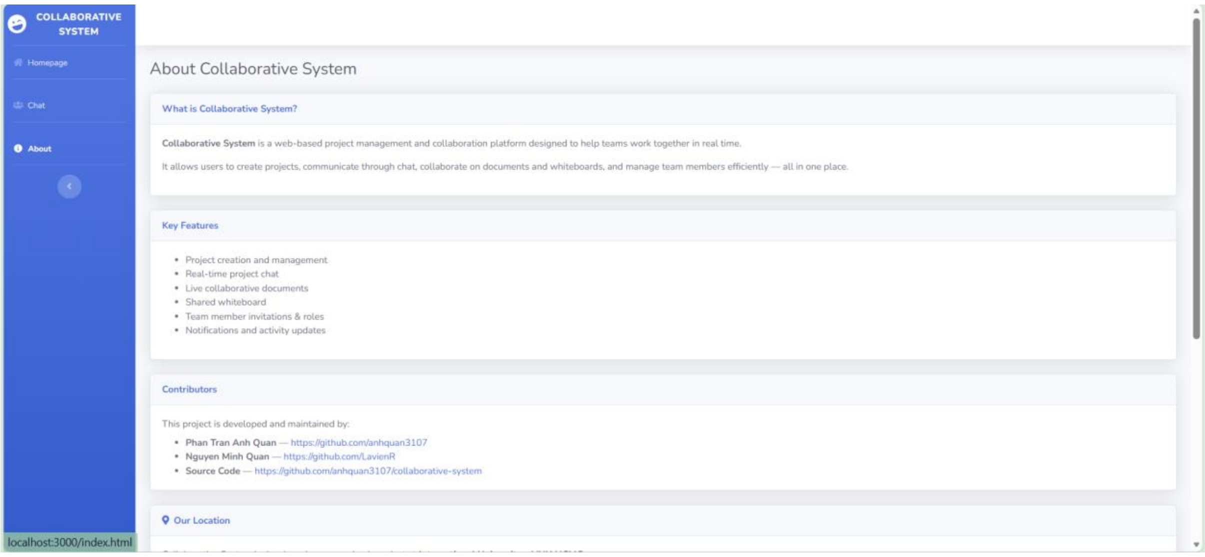
You
Got it

mqu
Got it

The members of each project can text to each other



The two members can video call with each other



About Collaborative System

What is Collaborative System?

Collaborative System is a web-based project management and collaboration platform designed to help teams work together in real time.

It allows users to create projects, communicate through chat, collaborate on documents and whiteboards, and manage team members efficiently — all in one place.

Key Features

- Project creation and management
- Real-time project chat
- Live collaborative documents
- Shared whiteboard
- Team member invitations & roles
- Notifications and activity updates

Contributors

This project is developed and maintained by:

- **Phan Tran Anh Quan** — https://github.com/anhquan3107
- **Nguyen Minh Quan** — https://github.com/LavienR
- **Source Code** — https://github.com/anhquan3107/collaborative-system

Our Location

About us

III. Code Explanation:

1. Backend Structure

Our backend is divided into many folders, each folder is responsible for how our features work.

The controllers process incoming HTTP requests, validate request data, coordinate with the models to perform the required operations, and return appropriate responses to the client. This layer contains the core business logic of the application.

The models contain SQL query implementations for the application. Each model encapsulates all database operations related to a specific entity, such as users, projects, chat messages, documents, and whiteboards.

The routes folder defines the API endpoints exposed by the backend. Each route file maps specific HTTP methods and URL paths to corresponding controller functions. This routing layer serves as the entry point for client requests and helps organize endpoints by feature, making the API easier to understand and extend.

The middleware folder contains shared logic that runs before requests reach the controllers. This includes tasks such as authentication, authorization, and request validation. Middleware ensures that only valid and authorized requests are processed, enhancing the security and reliability of the system.

The sockets folder manages real-time communication using WebSockets. It enables live features such as chat messaging, collaborative document editing, and whiteboard synchronization by broadcasting events between connected clients. This separation allows real-time logic to coexist cleanly alongside traditional REST APIs.

Finally, the config folder stores configuration-related code, including database connections and environment-based settings. Centralizing configuration simplifies deployment and allows the application to adapt easily to different environments such as development and production.

2. Features:

2.1. Authentication (Register)

The registration process begins on the frontend when the registration logic file register.js is loaded. This file attaches a submit event listener to a form identified by registerForm. When the user submits the form, the default browser behavior is prevented so that the process can be handled entirely through JavaScript.

Once the form is submitted, the script reads the values entered by the user, including the username, email address, password, and confirmation password. Before sending any data to the server, a basic validation step is performed to ensure that the password and confirmation password match. If they do not match, an error notification is displayed using Notyf and the registration request is stopped immediately.

After passing client-side validation, the interface enters a loading state. The submit button is disabled to prevent duplicate requests, and a spinner is shown to indicate that the registration process is in progress. This improves user experience and ensures that only one request is sent to the backend.

The registration request is then sent through the frontend API abstraction layer. The register function in api/auth.js is called, which internally uses the generic apiPost method from apiClient.js. This results in an HTTP POST request being sent to the /api/auth/register endpoint, with the user's registration details included in the request body as JSON.

On the backend, the Express router defined in authRoutes.js receives the incoming request and forwards it to the register controller function. The controller is responsible for handling the account creation logic, which typically includes validating the input data, checking whether the email or username already exists, securely hashing the password, and storing the new user record in the database.

If the registration is successful, the backend returns a success response to the frontend. Upon receiving this response, the frontend displays a success notification informing the user that their account has been created successfully. The system then waits briefly before redirecting the user to the login page, encouraging them to authenticate using their newly created credentials.

If an error occurs at any stage during the registration process, such as invalid input or a server-side failure, the error is caught on the frontend. The submit button is re-enabled, the loading indicator is removed, and an error notification is displayed to inform the user that the registration attempt was unsuccessful.

2.2. Authentication (Login)

The login process begins when the user accesses the login page and the login.js script is loaded. This script attaches a submit event listener to the form identified by loginForm. When the user submits the form, the default browser behavior is prevented so that the authentication process can be handled entirely through JavaScript without reloading the page.

After the form submission is intercepted, the script retrieves the email address and password entered by the user. To provide immediate feedback and prevent multiple submissions, the login button is disabled and a loading spinner is displayed, indicating that the system is attempting to authenticate the user.

The authentication request is then sent through the frontend API abstraction layer. The login function in api/auth.js is called, which internally uses the generic apiPost method from apiClient.js. This results in an HTTP POST request being sent to the /api/auth/login endpoint, containing the user's email and password in JSON format.

On the backend, the Express router defined in authRoutes.js receives the request and forwards it to the login controller function. The controller is responsible for validating the credentials, verifying the user's identity, and generating a JSON Web Token (JWT) if the authentication is successful. This token represents the user's authenticated session.

Once the backend responds successfully, the frontend receives the authentication token and stores it in the browser's local storage. This token is later attached automatically to future API requests as an authorization header, allowing the user to access protected resources without logging in again.

After storing the token, the frontend displays a success notification to inform the user that the login was successful. A short delay is applied so the message remains visible, after which the user is redirected to the main application page. This marks the completion of the login process and the start of the authenticated session.

If the authentication attempt fails at any point, the error is caught on the frontend. The login button is re-enabled, the loading indicator is removed, and an error notification is displayed to inform the user that the login attempt was unsuccessful. This ensures clear feedback while allowing the user to try again.

2.3. Project creating

The member management flow begins when a user creates a new project. On the frontend, the project creation request is sent using the createProject function in project.js, which calls the generic apiPost method with the /api/projects endpoint. This request includes the project name and optional description, and it automatically carries the user's JWT token in the authorization header.

On the backend, the request is handled by the addProject controller in projectController.js. After validating the input, the controller calls the createProject function in projectModel.js, which inserts a new record into the projects table. Immediately after creating the project, the system inserts a corresponding record into the project_members table, assigning the project creator the role of owner. This ensures that every project has at least one member and that ownership is clearly defined from the start.

2.4. Member management

a. Authentication and Access Control for Member Operations

All member-related project routes are protected using the verifyToken middleware. This middleware validates the JWT token sent from the frontend and attaches the authenticated user's information to req.user. As a result, every member-related action is tied to an authenticated identity, and unauthorized access to project data is prevented.

Before performing any member operation, the backend checks whether the requesting user has permission to access the project. This is done through helper functions such as isProjectMember, isProjectOwner, and getUserProjectRole in projectMemberModel.js. These checks ensure that only valid project members can view member lists and that only owners can perform sensitive actions such as removing members or sending invitations.

b. Loading project members

When a project workspace is opened, the frontend calls the member-loading logic during initialization. The loadMembers function triggers an API request to /api/projects/:projectId/members using the centralized path definitions. This request includes the JWT token so the backend can verify the user's identity.

On the backend, the request is handled by the listProjectMembers controller. The controller first determines the user's role in the project by calling getUserProjectRole. If the user is not a member, access is denied. If access is granted, the controller retrieves the full member list from the database using getProjectMembers, which joins the users, projects, and project_members tables. The response includes each member's role, join time, and ownership status, allowing the frontend to render an accurate member list.

c. Removing a project member

When a project owner attempts to remove a member, the frontend sends a DELETE request to /api/projects/:projectId/members/:userId. This action is only available in the UI if the current user is the project owner, but the backend still performs its own verification to ensure security.

The backend processes this request in the removeMember controller. It first confirms that the requester is the project owner by calling isProjectOwner. If the requester is not the owner, the operation is rejected. The controller also prevents owners from removing themselves to avoid leaving a project without an owner. Once all checks pass, the member is removed from the project_members table using removeProjectMember, and a success response is returned.

d. Leaving a Project as a Member

For non-owner members, the system allows users to leave a project voluntarily. The frontend triggers this action by sending a DELETE request to /api/projects/:projectId/leave. This request is authenticated and tied to the currently logged-in user.

On the backend, the leaveProject controller handles this request. The controller first checks whether the user is the project owner. If the user is the owner, the request is rejected, as owners must either transfer ownership or delete the project. If the user is a regular member, their membership record is removed from the project_members table, and the system confirms that the user has successfully left the project.

e. Invite members

The invitation feature begins on the frontend through the API abstraction defined in frontend/api/invitation.js. This file centralizes all invitation-related API calls, including sending invitations, retrieving project invitations, canceling invitations, viewing a user's pending invitations, and accepting or declining an invitation. Each function internally uses the shared apiClient methods to ensure authentication tokens are automatically attached to every request.

When a project owner wants to invite a new member, the interaction starts in frontend/js/pages/project/document/invitations.js. The setupInvitationListeners function registers an event listener on the invite button, which opens a modal dialog where the owner can enter the invitee's email address. Once submitted, the email is validated on the client side to ensure it is not empty before proceeding.

After validation, the frontend calls the inviteToProject function with the current projectId and the entered email. This sends a POST request to the backend route /api/invitations/project/:projectId. Upon success, a Notyf success notification is displayed, the modal is closed, the input field is cleared, and both the pending invitations list and project member list are refreshed to reflect the updated project state.

On the backend, the POST request is handled by the inviteToProject controller through the route defined in invitationRoutes.js. The route is protected by the verifyToken middleware, ensuring only authenticated users can send invitations. Inside the controller, the system verifies that the requesting user is a project owner before proceeding. It then generates a secure invitation token, assigns an expiration date, and stores the invitation details in the project_invitations database table with a status of pending.

Project owners can view all pending invitations for a project through the dropdown interface. This is powered by the loadPendingInvitations function, which periodically fetches data from the /api/invitations/project/:projectId endpoint every two minutes. The backend responds with all active invitations for that project, and the frontend renders them dynamically, displaying the invitee's email and the expiration date.

If the project owner decides to revoke an invitation, the cancel button triggers a DELETE request via the cancelInvitation function. This request is sent to /api/invitations/project/:projectId/:invitationId. The backend verifies ownership again before updating the invitation's status to cancelled. Once completed, the frontend reloads the pending invitations list and displays a confirmation notification.

For invited users, the system provides a separate flow to manage incoming invitations. When a user opens their dashboard or invitation view, the frontend calls getPendingInvitations, which sends a GET request to /api/invitations/pending. The backend retrieves all invitations associated with the authenticated user's email that are still in a pending state and have not expired.

When a user accepts an invitation, the frontend sends a POST request to /api/invitations/:token/accept. The backend validates the token, checks that the invitation is still valid and not expired, and confirms that the invitation email matches the authenticated user. If all checks pass, the user is added to the project by inserting a new record into the project_members table, and the invitation status is updated to accepted.

If the user chooses to decline the invitation instead, the frontend sends a POST request to /api/invitations/:token/decline. The backend performs the same validation steps but updates the invitation status to declined without adding the user to the project. This ensures that declined invitations cannot be reused or accepted later.

2.5. Document Management

a. Document Creating

The document creation process begins when a user is inside a project workspace and clicks the "Create File" button. This interaction is handled in the frontend workspace initialization, where event listeners for file creation are registered after the project loads successfully.

When the user selects the document type and enters a title, the frontend validates the input to ensure the title is not empty. Once validated, a request is prepared and sent to the document API using the project ID extracted from the URL. This request is issued through the createDocument function, which sends a POST request to the backend endpoint responsible for document creation within the selected project.

On the backend, the request first passes through authentication middleware, which verifies the user's access token and attaches the user identity to the request. The document route handler then forwards the request to the document controller, which extracts the project ID and document title from the request.

Before creating the document, the backend performs an access control check to ensure the user is a valid member of the project. This verification is done at the model layer by checking the project membership table. If the user does not belong to the project, the request is rejected to prevent unauthorized document creation.

Once access is confirmed, the backend inserts a new record into the documents table with the associated project ID, title, and an empty initial content field. After insertion, the newly created document is retrieved from the database and returned to the frontend as part of the response.

When the frontend receives the response, it immediately opens the newly created document by calling the document opening logic. The editor interface is displayed, and the document becomes the active working file for the user. At the same time, the document list in the sidebar is refreshed so that the new document appears without requiring a page reload.

Finally, a success notification is shown to the user, and the document creation modal is closed. At this point, the document is fully initialized, accessible to all project members with permission, and ready for real-time editing and collaboration.

b. Opening Document

The document opening process starts when the user clicks a document item in the sidebar. This action is detected by the event listener attached in fileList.js, where the attachFileListeners function extracts the document ID and title and calls the openDocument(id, title) function.

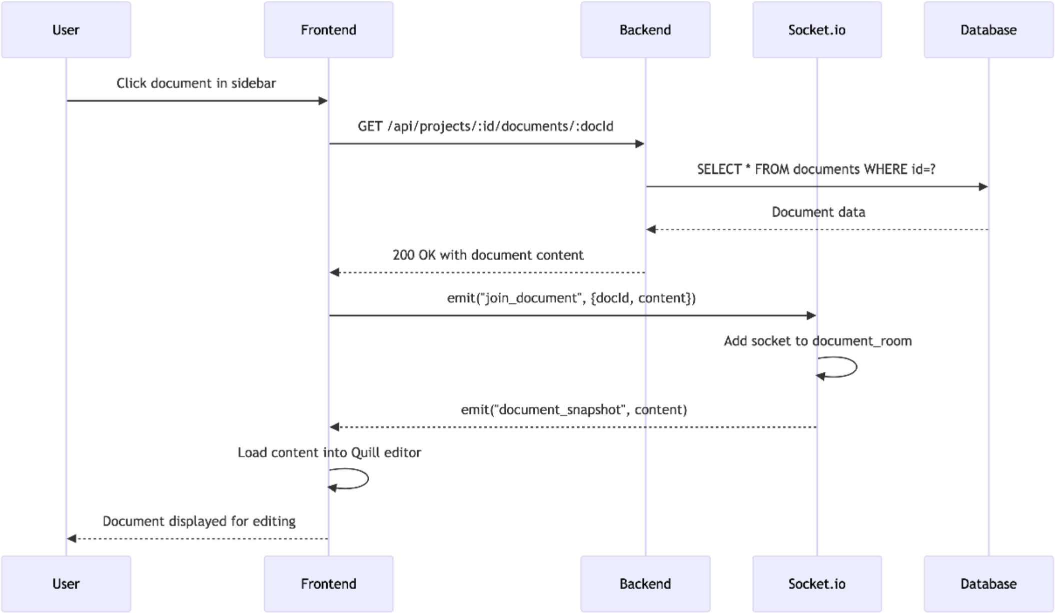
Once openDocument is triggered, the user interface switches from the placeholder view to the editor view. The showEditor() function hides the empty state and displays the editor wrapper, while the selected document ID is stored as the current active document.

After updating the interface, the frontend sends an API request to load the document content. The getDocument(projectId, docId) function in document.js sends an HTTP GET request to the backend endpoint responsible for retrieving a single document.

On the backend, the request is processed through the document routes and controller. The flow passes from documentRoutes.js to documentController.getDocument, which then calls documentModel.getDocumentById. The backend verifies access permissions and returns the document data, including its stored content.

When the response arrives on the frontend, the document content is loaded into the editor. The loadIntoQuill function checks whether the content is a Quill Delta JSON string or plain HTML. If the content is a Delta, it is applied using quill.setContents; otherwise, the content is pasted as HTML.

Finally, the client joins the real-time collaboration channel. The emitJoinDocument(docId) function is called to connect the user to the corresponding Socket.IO room, enabling live collaboration features for the opened document.


User
Frontend
Backend
Socket.io
Database
Click document in sidebar
GET /api/projects/:id/documents/:docId
SELECT * FROM documents WHERE id=?
Document data
200 OK with document content
emit("join_document", {docId, content})
Add socket to document_room
emit("document_snapshot", content)
Load content into Quill editor
Document displayed for editing

c. Real-time editing

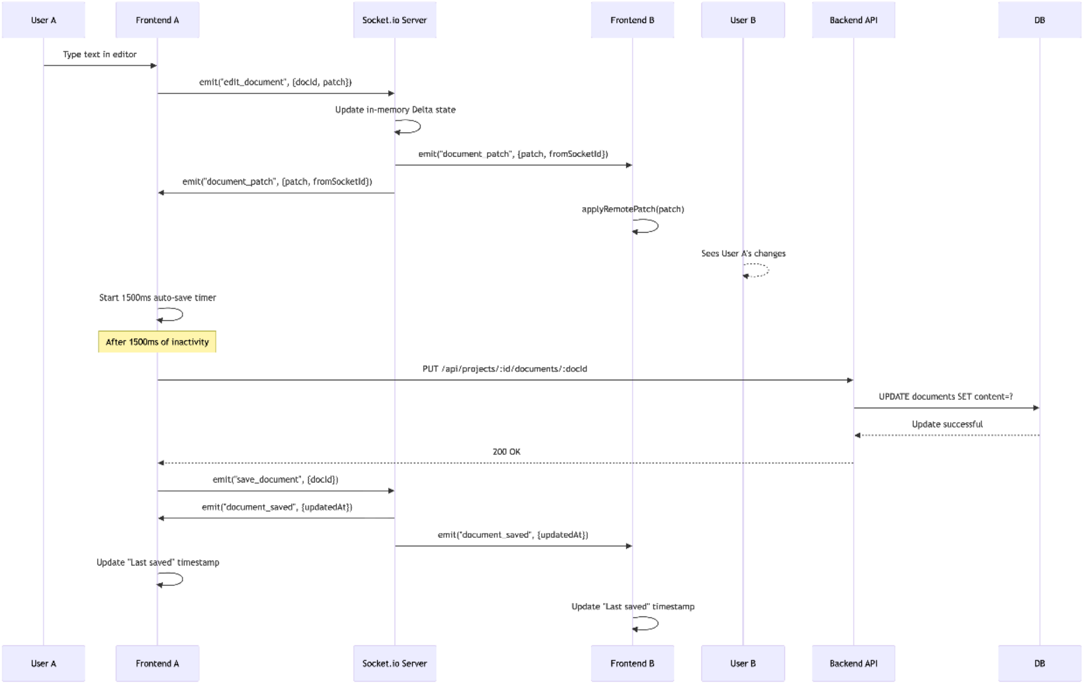
Real-time editing begins when a user types into the Quill editor. Each text modification triggers the quill.on("text-change") event inside documentEditor.js, which calls the handleRemoteApply function.

To prevent infinite update loops, the handler checks whether the change source is a user action. Only changes originating from direct user input are emitted, while programmatic updates are ignored.

When a valid change is detected, the editor emits a patch through the socket connection. The emitDocumentPatch(currentDocId, delta) function sends the Quill Delta object to the backend using a WebSocket event.

On the server side, the Socket.IO handler receives the edit event and updates its in-memory document state. The server then broadcasts the patch to all other connected users in the same document room.

On receiving a remote patch, the client applies the change to the editor. The applyRemotePatch function temporarily enables a protection flag to prevent feedback loops, then merges the incoming Delta into the editor using quill.updateContents. As a result, all users see the changes in real time without refreshing the page.


User A
Frontend A
Socket.io Server
Frontend B
User B
Backend API
DB
Type text in editor
emit("edit_document", {docId, patch})
Update in-memory Delta state
emit("document_patch", {patch, fromSocketId})
emit("document_patch", {patch, fromSocketId})
applyRemotePatch(patch)
Sees User A's changes
Start 1500ms auto-save timer
After 1500ms of inactivity
PUT /api/projects/:id/documents/:docId
UPDATE documents SET content=?
Update successful
200 OK
emit("save_document", {docId})
emit("document_saved", {updatedAt})
emit("document_saved", {updatedAt})
Update "Last saved" timestamp
Update "Last saved" timestamp

d. Save Document

The document saving process is triggered in several ways. A user may manually click the save button, allow the auto-save timer to expire after a short period of inactivity, or use a keyboard shortcut such as Ctrl + S. All of these actions call the saveCurrentDocument function.

When saving is initiated, the editor’s current state is collected using quill.getContents. The resulting Delta object is serialized into a JSON string to prepare it for storage.

The frontend then sends an HTTP PUT request to the backend using the saveDocument(projectId, currentDocId, content) function. This request targets the document save endpoint and includes the updated content in the request body.

After the backend successfully persists the document content in the database, the frontend emits a socket event to notify other collaborators. The emitSaveDocument(docId) call informs connected clients that the document has been saved.

Finally, the user interface updates the “Last saved” timestamp to reflect the successful save operation, providing immediate feedback to all active collaborators.

2.6. Whiteboard management

a. Whiteboard creating

The flow starts when the user clicks the Create button and selects Whiteboard from the file type dropdown. This interaction is handled in fileCreate.js, where the modal collects the whiteboard title and triggers the creation logic after submission.

The frontend sends a POST request using createWhiteboard(projectId, title) from frontend/api/whiteboard.js. This request is sent to the backend endpoint POST /api/projects/:projectId/whiteboards.

On the backend, whiteboardRoutes.js forwards the request to createNewWhiteboard in whiteboardController.js. The controller verifies the user's access through the model, then calls createWhiteboard in whiteboardModel.js to insert a new row into the whiteboards table with an empty strokes array.

After creation, the backend immediately fetches the newly created whiteboard and returns it as { whiteboard }. The frontend receives this response and directly calls openWhiteboard(whiteboardId, title) so the new whiteboard is opened instantly without requiring a page refresh.

b. Opening whiteboard

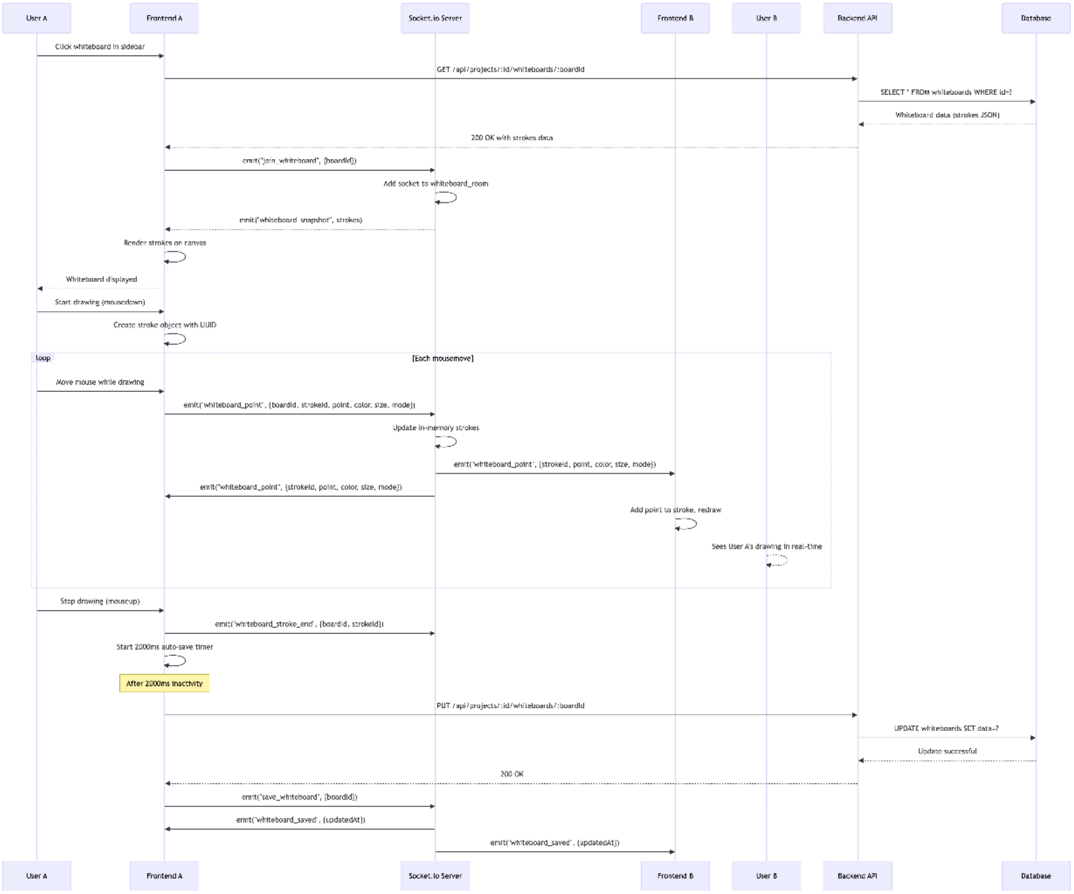
A whiteboard can be opened either right after creation or by clicking it in the sidebar file list. The click is detected in fileList.js, where the item's data-type, data-id, and data-title are extracted and passed to openWhiteboard.

The openWhiteboard function in whiteboardOpen.js first saves the currently active whiteboard (if any) to prevent data loss. It then sends a GET request via getWhiteboard(projectId, boardId) to retrieve the selected board's data from the backend.

On the backend, the request flows through whiteboardRoutes.js to getWhiteboard in the controller. The controller checks project membership and fetches the whiteboard from the database using getWhiteboardById.

Once the data reaches the frontend, the canvas is initialized, resized, and cleared. The stored stroke data is parsed from JSON and loaded into the strokes array, after which redrawCanvas() renders the whiteboard visually.

Finally, the client joins the Socket.IO room using joinWhiteboard(boardId) to enable real-time collaboration, and the selected whiteboard is highlighted in the sidebar using updateActiveFile.


User A
Frontend A
Socket.io Server
Frontend B
User B
Backend API
Database
Click whiteboard in sidebar
GET /api/projects/:id/whiteboards/:boardId
SELECT * FROM whiteboards WHERE id=?
Whiteboard data (strokes JSON)
200 OK with strokes data
emit("join_whiteboard", {boardId})
Add socket to whiteboard_room
emit("whiteboard_snapshot", strokes)
Render strokes on canvas
Whiteboard displayed
Start drawing (mousedown)
Create stroke object with UUID
loop
[Each mousemove]
Move mouse while drawing
emit('whiteboard_point', {boardId, strokeId, point, color, size, mode})
Update in-memory strokes
emit('whiteboard_point', {strokeId, point, color, size, mode})
emit("whiteboard_point", {strokeId, point, color, size, mode})
Add point to stroke, redraw
Sees User A's drawing in real-time
Stop drawing (mouseup)
emit('whiteboard_stroke_end', {boardId, strokeId})
Start 2000ms auto-save timer
After 2000ms inactivity
PUT /api/projects/:id/whiteboards/:boardId
UPDATE whiteboards SET data=?
Update successful
200 OK
emit("save_whiteboard", {boardId})
emit('whiteboard_saved', {updatedAt})
emit('whiteboard_saved', {updatedAt})
User A
Frontend A
Socket.io Server
Frontend B
User B
Backend API
Database

c. Real-time Drawing

The real-time drawing flow begins when a user interacts with the whiteboard canvas using the mouse. In whiteboardCanvas.js, the mousedown event triggers startDrawing, which initializes a new stroke object containing a unique stroke ID, the selected color, brush size, drawing mode (draw or erase), and an empty list of points. This stroke is immediately pushed into the shared strokes array so it can be rendered locally.

As the user moves the mouse while holding the button, the mousemove event repeatedly calls the draw function. Each movement calculates a new point based on the mouse position relative to the canvas and appends it to the current stroke's points array. After adding the point locally, the client emits a whiteboard_point event through Socket.IO, sending the board ID, stroke ID, point coordinates, color, size, and mode to the server.

On the server side, the Socket.IO handler receives the whiteboard_point event and broadcasts it to all other clients who have joined the same whiteboard room. The server does not modify the stroke data; it acts as a relay to ensure low latency and keep all clients synchronized in real time.

On the receiving client, whiteboardSocket.js listens for the whiteboard_point event. When a point arrives, the client first checks that the incoming board ID matches the currently opened whiteboard. It then searches for an existing stroke with the same stroke ID. If the stroke does not yet exist, a new stroke object is created using the transmitted color, size, and mode. The received point is appended to the stroke's point list.

After updating the stroke data, redrawCanvas() is called. This function clears the entire canvas and replays every stroke from the strokes array. During rendering, the drawing mode is respected: normal strokes use source-over, while eraser strokes use destination-out to remove previously drawn content. Quadratic curves are used to smooth the stroke path, ensuring visually consistent drawing across all clients.

When the user releases the mouse button or leaves the canvas area, the mouseup or mouseout event triggers stopDrawing. This ends the current stroke and emits a whiteboard_stroke_end event so other clients know the stroke is complete. At this point, the client schedules a delayed save using queueWhiteboardSave, allowing the full stroke to be persisted without interrupting the real-time drawing experience.

If a user clears the whiteboard, the clearBoardRealtime function resets the local strokes array, redraws an empty canvas, and emits a whiteboard_clear event. All connected clients receive this event, clear their local stroke data, and redraw their canvases to remain consistent.

To handle late joiners or reconnections, the socket logic supports full state synchronization through whiteboard_snapshot. When a client joins a whiteboard room, the server can send the entire strokes array, which replaces the local state and ensures the canvas accurately reflects the current whiteboard content.

d. Saving the Whiteboard

Saving happens automatically after the user stops drawing. The queueWhiteboardSave function delays saving to avoid excessive API calls, then calls saveCurrentWhiteboard.

The frontend sends a PUT request to /api/projects/:projectId/whiteboards/:whiteboardId, containing the full strokes array as a JSON string. The backend verifies access and updates the data and updated_at fields in the database.

Once saved, the UI updates the Last saved timestamp, and other connected users receive a whiteboard_saved event to keep timestamps consistent.

2.7. Chat

a. Loading chat history

When the chat page loads or a user switches to a project, the system first initializes authentication and user state. After verifying the token and fetching the current user, the client establishes a Socket.IO connection and prepares the chat environment.

When a project is selected, switchProject is called. The client clears the current chat UI, updates the active project state, and sends a join_project_chat event to the server so the user joins the correct chat room. This ensures that future messages are scoped to the selected project.

Next, the client fetches historical messages using getChatHistory(projectId), which sends a GET request to /api/projects/:projectId/messages. On the backend, chatController.getMessages verifies that the user is a member of the project before querying the database.

The database query in getProjectMessages retrieves the most recent messages, orders them correctly, and returns them with usernames attached. The frontend renders these messages in chronological order so the chat appears consistent and readable.

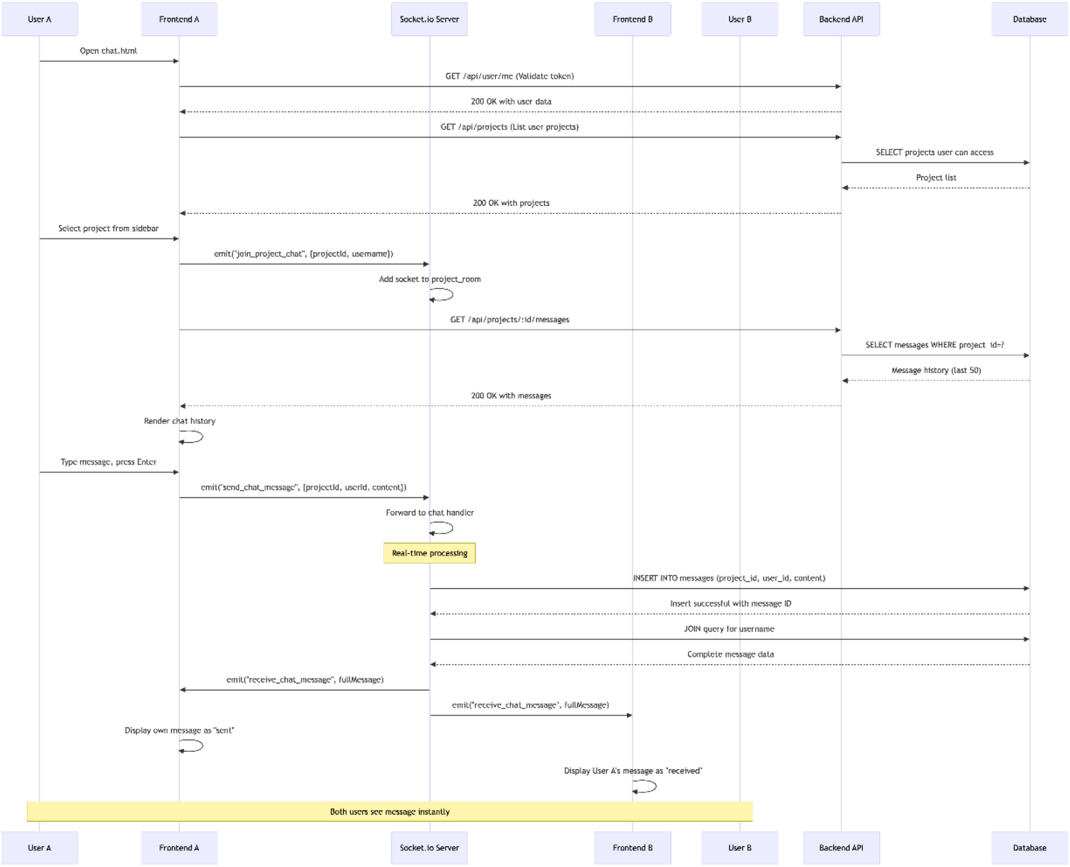
b. Send messages

When the user types a message and presses Enter or clicks the Send button, the sendMessage function is triggered. The input is trimmed to prevent empty messages, and the message content is prepared for transmission.

Instead of using an HTTP request, the client emits a send_chat_message Socket.IO event containing the project ID, user ID, and message content. This allows instant delivery without page reloads or polling.

The input field is immediately cleared to provide responsive feedback to the user, even before other clients receive the message.

c. Real-time Messgae Delivery


User A
Frontend A
Socket.io Server
Frontend B
User B
Backend API
Database
Open chat.html
GET /api/user/me (Validate token)
200 OK with user data
GET /api/projects (List user projects)
SELECT projects user can access
Project list
200 OK with projects
Select project from sidebar
emit('join_project_chat', {projectId, username})
Add socket to project_room
GET /api/projects/:id/messages
SELECT messages WHERE project_id=?
Message history (last 50)
200 OK with messages
Render chat history
Type message, press Enter
emit("send_chat_message", {projectId, userId, content})
Forward to chat handler
Real-time processing
INSERT INTO messages (project_id, user_id, content)
Insert successful with message ID
JOIN query for username
Complete message data
emit("receive_chat_message", fullMessage)
emit("receive_chat_message", fullMessage)
Display own message as "sent"
Display User A's message as "received"
Both users see message instantly
User A
Frontend A
Socket.io Server
Frontend B
User B
Backend API
Database

On the server side, the Socket.IO handler receives the send_chat_message event. The server stores the message in the database using createMessage, which inserts the message and immediately fetches the full record including the sender's username.

After saving, the server broadcasts the message to all users in the same project chat room using the receive_chat_message event. This ensures that every connected client receives the same message data.

On the frontend, each client listens for receive_chat_message. When a message arrives, the client checks whether the message's project_id matches the currently active project. This prevents messages from appearing in the wrong chat room.

If the project matches, the message is appended to the chat UI, styled differently depending on whether the sender is the current user or another member. The chat window automatically scrolls to the bottom to keep the newest message visible.

d. Real-time Notification

If a message is sent to a project that the user is not actively viewing, the system still handles it in real time through the new_message_notification event.

Multiple components listen for this event, including the dashboard message center, the project workspace popup, and the chat message manager. Each listener displays a notification using messageNotyf, showing the sender's name and message preview.

These notifications are clickable. When clicked, they redirect the user directly to chat.html with the correct projectId in the URL, allowing instant navigation to the relevant conversation.

e. Unread messages

When a new message notification is received outside the active chat, the unread message count is increased. This count is stored in localStorage so it persists across page reloads.

The unread count is reflected in multiple UI elements, including a badge icon and statistics panels. The badge remains visible at all times and updates dynamically as new messages arrive.

Each new message is also added to a recent messages list, which is limited to the latest five messages. This list is rendered inside a dropdown menu for quick access.

f. Recent messages

When the user opens the message dropdown, the system renders recent messages from localStorage. Each item includes the sender name, content preview, timestamp, and a link to the correct project chat.

Clicking a message removes it from the recent list, decreases the unread count by one, and updates both the badge and statistics immediately. This gives the user fine-grained control over what is considered “read.”

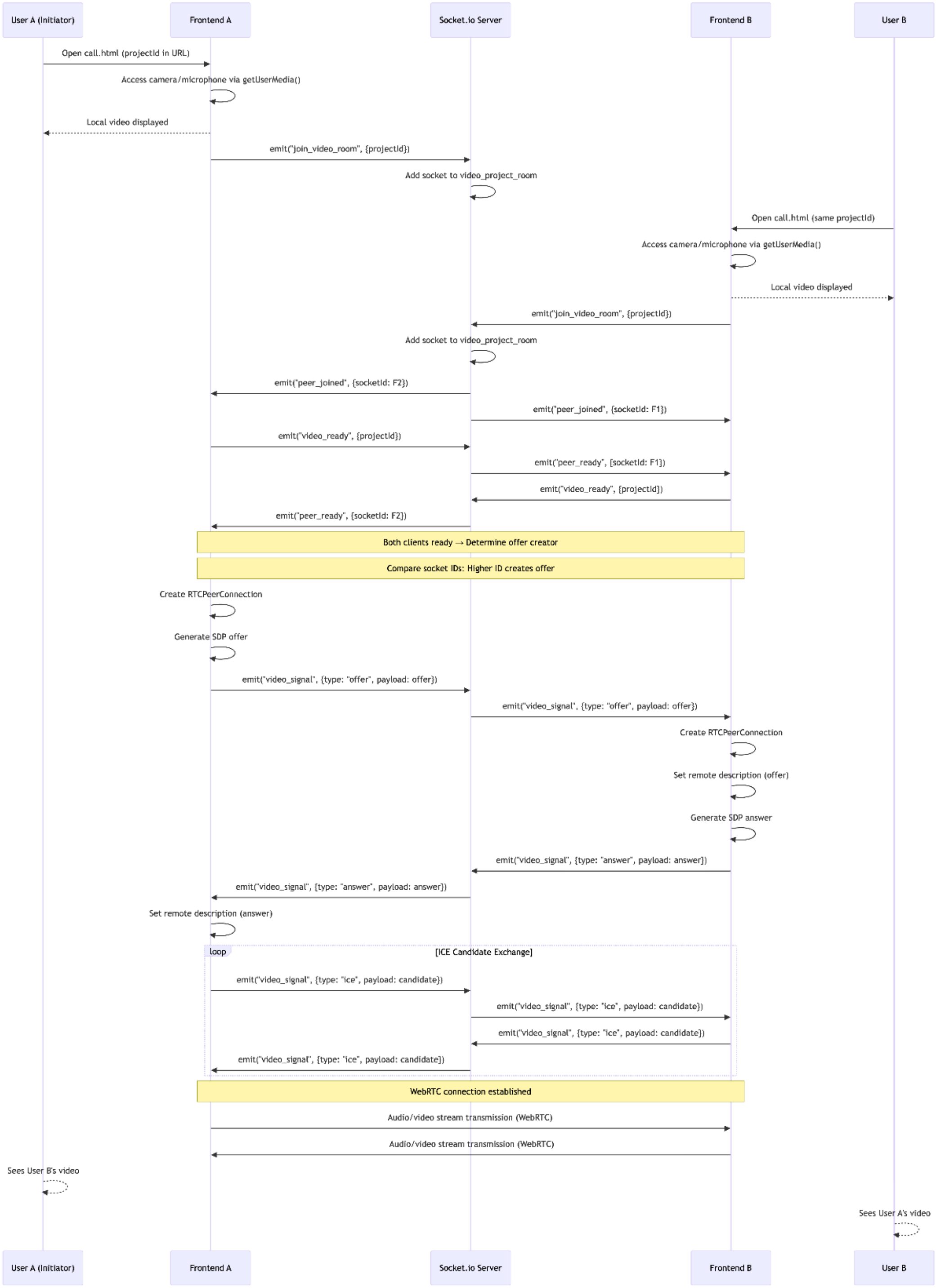
g. Switching projects

When the user switches projects, the client first leaves the previous chat room by emitting leave_project_chat. This prevents further messages from the old project from being delivered to the UI.

The client then joins the new project’s chat room and reloads the chat history for that project. The UI is fully reset so messages do not mix across projects.

This room-based architecture ensures isolation, scalability, and correct message delivery even when users are members of many projects simultaneously.

2.8. Calling management


User A (Initiator)
Frontend A
Socket.io Server
Frontend B
User B
Open call.html (projectId in URL)
Access camera/microphone via getUserMedia()
Local video displayed
emit("join_video_room", {projectId})
Add socket to video_project_room
Open call.html (same projectId)
Access camera/microphone via getUserMedia()
Local video displayed
emit("join_video_room", {projectId})
Add socket to video_project_room
emit("peer_joined", {socketId: F2})
emit("peer_joined", {socketId: F1})
emit("video_ready", {projectId})
emit("peer_ready", {socketId: F1})
emit("video_ready", {projectId})
emit("peer_ready", {socketId: F2})
Both clients ready → Determine offer creator
Compare socket IDs: Higher ID creates offer
Create RTCPeerConnection
Generate SDP offer
emit("video_signal", {type: "offer", payload: offer})
emit("video_signal", {type: "offer", payload: offer})
Create RTCPeerConnection
Set remote description (offer)
Generate SDP answer
emit("video_signal", {type: "answer", payload: answer})
emit("video_signal", {type: "answer", payload: answer})
Set remote description (answer)
loop
[ICE Candidate Exchange]
emit("video_signal", {type: "ice", payload: candidate})
emit("video_signal", {type: "ice", payload: candidate})
emit("video_signal", {type: "ice", payload: candidate})
emit("video_signal", {type: "ice", payload: candidate})
WebRTC connection established
Audio/video stream transmission (WebRTC)
Audio/video stream transmission (WebRTC)
Sees User B's video
Sees User A's video
User A (Initiator)
Frontend A
Socket.io Server
Frontend B
User B

The video call begins when the call page loads with a valid project ID. Before connecting to Socket.IO, the client requests access to the user's camera and microphone and waits until the local video is fully playing. This guarantees that signaling only starts after the media stream is ready. Once initialized, the client connects to the socket server, joins a project-specific video room, and emits a readiness signal indicating that the camera is active.

When a user joins the video room, the server notifies both the new participant and any existing peer through peer_joined events. The server also tracks camera readiness for each socket and broadcasts peer_ready when a participant finishes initializing their media devices. This synchronization ensures that WebRTC negotiation only starts when both peers are present and fully prepared.

After both peers are ready, the system deterministically selects which client creates the WebRTC offer by comparing socket IDs. The selected client initializes an RTCPeerConnection, attaches local media tracks, generates an SDP offer, and sends it through Socket.IO. The receiving peer responds by creating an SDP answer, and both sides exchange ICE candidates until a direct peer-to-peer connection is established and remote media is rendered.

During the call, users can mute their microphone or disable their camera by toggling media track states without renegotiating the connection. Call invitations can be triggered from the chat interface, allowing users to receive incoming call notifications even when they are not inside the video room.

When a call ends, the client notifies the server, stops all local media tracks, closes the peer connection, and updates the UI. If a user disconnects or reloads unexpectedly, the server cleans up the ready state and instructs the remaining peer to reset their call state, enabling automatic recovery when the user reconnects.

2.9. Password changing

The password reset process starts when a user submits their email address on the “Forgot Password” page. The frontend sends this email to the backend through the /password/forgot-password API endpoint. The server first checks whether the email exists in the database. If the email is not registered, the request is rejected immediately to prevent unnecessary processing.

If the email is valid, the backend generates a 6-digit one-time password (OTP) and stores it together with the user’s email in the server session. This OTP is then sent to the user’s email using Nodemailer and Gmail SMTP. Storing the OTP in the session ensures that it is temporary and tied to the current reset flow.

After receiving the OTP, the user enters it on the verification page. The frontend sends the entered code to the /password/verify-otp endpoint. The backend compares the provided OTP with the value stored in the session. If the values match, the OTP is considered valid and the user is allowed to proceed to the password reset step.

On the reset password page, the user submits a new password. This password is sent to the /password/reset-password endpoint. The backend hashes the new password using bcrypt and updates it in the database for the email stored in the session. Once the update is successful, the session is destroyed to prevent OTP reuse, and the user is redirected back to the login page.

2.10. Search bar

The project search feature is initialized when the dashboard loads. After the user is authenticated in main.js, the initSearchManager() function is called. This function locates the search input field in the navigation bar and attaches an input event listener to detect user typing.

When the user types into the search bar, the onSearchInput handler is triggered. Instead of calling the backend immediately on every keystroke, the input is debounced with a 300ms delay. This ensures that the search request is only sent after the user pauses typing, reducing unnecessary API calls and improving performance.

Once the debounce timer expires, searchProjects(keyword) is executed. This function calls the existing getProjects API and passes the search keyword as a query parameter. If the input is empty, it fetches all projects; otherwise, it requests only projects whose names match the search term.

On the backend, the listProjects controller receives the request. It extracts the search query parameter and forwards it, together with the authenticated user ID, to the getProjectsByUser model function. The database query filters projects owned by or shared with the user and applies the search condition before returning the results.

After the response arrives at the frontend, the project list is refreshed by calling renderProjectGrid. This replaces the current project display with the filtered results in real time, allowing the user to instantly see matching projects without reloading the page.

IV. Challenges:

1. Lack of knowledge

During the development of this project, several challenges were encountered. One major difficulty was the lack of prior knowledge in some technologies, especially real-time systems and socket-based communication, which required extra time for learning and experimentation. Understanding how different components interacted across frontend and backend was initially challenging.

2. Optimizing code structrue

Another challenge was optimizing the code structure and performance. As features increased, maintaining clean, reusable, and efficient code became more difficult. Some parts required refactoring to reduce duplication and improve readability, especially in real-time features.

3. Using github

Version control using GitHub also presented difficulties. Managing branches, resolving merge conflicts, and keeping the codebase consistent across team members required careful coordination and practice. Mistakes during merging sometimes caused temporary feature breakages.

4. Balancing new concepts

Finally, balancing learning new concepts while actively implementing features was challenging. Time had to be divided between studying unfamiliar technologies and applying them in practice, which occasionally slowed development progress.

V. Conclusion and Future work

In conclusion, this project successfully delivered a collaborative system with real-time chat, document editing, whiteboard drawing, and video calling features. It provided valuable hands-on experience in full-stack development, real-time communication, and system integration.

For future work, role-based access control can be introduced, such as editor and viewer roles, to improve collaboration management and security. Additionally, more advanced tools can be added to the whiteboard and document editor, such as shapes, comments, version history, and advanced formatting, to enhance usability and collaboration efficiency.

REFERENCES

OpenJS Foundation. (2017). *Express - Node.js web application framework*. Expressjs.com. https://expressjs.com/

MDN Contributors. (2023, September 25). *JavaScript*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/javascript

Socket.IO. (n.d.). *Introduction | Socket.IO*. Socket.io. https://socket.io/docs/v4/

WebRTC 1.0: Real-Time Communication Between Browsers. (n.d.). Www.w3.org. https://www.w3.org/TR/webrtc/