Università di Pisa

Master Of Science in Computer Engineering

Distributed Systems and Middleware Technologies

Distributed Shared Whiteboard

Project members:
Marco Imbelli Cai
Niccolò Mulè
Nicolò Picchi

Contents

1 Project Specifications

1.1 Introduction

Shared Whiteboards is a distributed web application that enables registered users to create, edit, and share work pages with others. With Shared Whiteboards, you can collaborate with colleagues, share information, or unleash your creativity.

1.2 Functional Requirements

- **Actors**
  - Unregistered user
  - Registered user
- **Unregistered users can:**
  - Create an account
- **Registered users can:**
  - Log in/out
  - Browse shared whiteboards
  - Create a new whiteboard, choosing a title, a description and whether to make the whiteboard read-only or not (which means that only the creator will be able to draw on it)
  - Share created whiteboards
  - View collaborators on a whiteboard
  - Draw on the whiteboard and view other users' changes
  - Remove whiteboard strokes using the eraser tool
  - Undo or redo whiteboard actions
  - Invite other users to join the collaboration
  - Join/leave accessible whiteboards
  - Remove users from the whiteboards they own
  - Delete whiteboards they own, thus removing all participants from it

1.3 Non-Functional Requirements

- Concurrent service access management
- Ensure consistency during live drawing sessions
- Ensure high availability
- Ensure high horizontal scalability

1.4 Synchronization and Communication Issues

- Manage concurrent edits from multiple users, ensuring minimal data loss.
- Ensure all users have synchronized views of the whiteboards through real-time updates, utilizing technologies like WebSockets or server-sent events.
- Notify users promptly when they are invited to collaborate on a new whiteboard, facilitating in-app notifications.
- Implement a mechanism to track and display current collaborators on the whiteboard, which may involve monitoring user activity or maintaining an active user list.

2 System Architecture

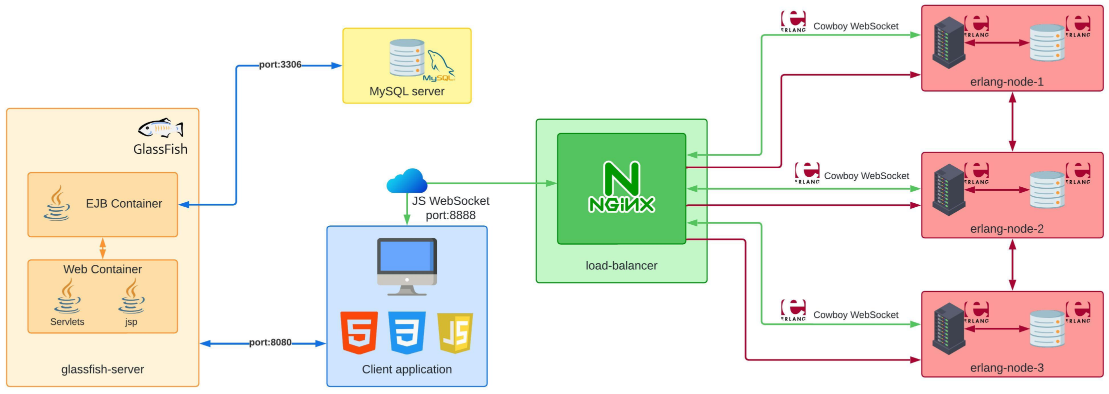


Figure 1: System architecture schema

The web application architecture consists of a Client Application, a Jakarta Application Server, a NGINX load balancer and multiple Erlang servers. The Jakarta EE Application Server, leveraging Glassfish 7.0.10 as its reference implementation, primarily handles the application's functionality and communicates with a MySQL relational database located on the same machine.

The Erlang Servers are responsible for managing operations on the whiteboards through HTTP WebSocket communication. Additionally, Remote Procedure Call (RPC) mechanisms have been implemented to facilitate direct communication between Glassfish and the Erlang nodes. This enables efficient management of whiteboards, including tasks such as deletion, addition, or participant management. For more detailed insights into each module's functionality and interactions, further analysis of individual components will be provided.

2.1 NGINX

As previously mentioned, we employ a NGINX proxy server to properly distribute load balance among Erlang servers. We decided to deploy the NGINX server with a custom configuration, that overrides the normal round-robin based load balancing offered by NGINX. Our load balancing policy redirects all WebSocket connections to a specific whiteboard to the same Erlang node based on the whiteboardId parameter sent by the client upon opening the WebSocket. This approach significantly reduces latency for users as actions on the whiteboard are processed by a single node, eliminating the delays that would occur if messages had to be sent across multiple servers. Although each node independently handles specific whiteboards, the Erlang nodes still exchange whiteboard states with one another, providing redundancy in case the primary node for a given set of whiteboards fails. However, we are aware that this load balancing policy may become problematic if a subset of whiteboards assigned to a specific node has a number of users that is substantially bigger than the other ones.

2.2 Erlang

The exchange of information between the various whiteboard operations is performed using Erlang. To take full advantage of Erlang's support for concurrency and scalability and to meet the requirements for high availability and fault tolerance, we deployed the necessary code that realizes the logic to make the whiteboards work on three Erlang nodes. Leveraging Cowboy, an HTTP server written in Erlang, each individual node exposes an endpoint to which clients will be able to connect to via a WebSocket. In this way, the server can receive whiteboard information from a user and forward it to other online users, meanwhile the user is not required to refresh the webpage. On the client-side, WebSocket communication has been implemented in JavaScript. Every message exchanged between client and server is serialized in JSON format.

2.2.1 Mnesia Database

We leveraged Mnesia Database to keep some persistent data in the Erlang nodes. More specifically:

- `whiteboard_access`: This table records which whiteboards each user can access and the level of permissions they have. This allows the Erlang server to enforce access control and prevent unauthorized access to whiteboards.
- `whiteboard_users`: This table tracks which users are currently connected to specific whiteboards and records the process ID of the Erlang process hosting the WebSocket for each user on a given whiteboard. Users can connect to multiple whiteboards simultaneously, but they cannot connect to the same whiteboard using different clients at the same time; the server will close the older WebSocket connection in such cases.
- `whiteboard_strokes_log`: This table logs actions performed by users on the whiteboards they can access. It details the type of action (whether a new stroke was added or an existing one was deleted), the timestamp, the strokeId the action refers to, and, if the action is for a new stroke, the serialized data of that stroke.
- `redo_stack`: This table functions exactly like a stack and it was necessary in order to implement the undo and redo functions, of which we provide the details in the following section.

2.2.2 Whiteboard Logic

The logic implemented in Erlang to realize the whiteboard functions has several responsibilities:

1. Upon user connection, the Erlang servers regenerate the whiteboard state on the client by sending all the active strokes, i.e., the strokes that do not have a corresponding delete action and are not in the `redo_stack` table
2. The server processes the following commands received from the client via WebSocket:
   (a) `addStroke`: This command is sent when a user adds a new stroke. The server generates a unique ID for that stroke and notifies all the other connected users. The stroke identifier is necessary to reference the stroke in future delete actions.
   (b) `deleteStroke`: This indicates that the user has deleted a stroke using the eraser tool. The stroke to be deleted is identified by the stroke ID in the message payload.

   (c) `updateUserCursor`: The client app regularly samples the coordinates of the cursor of the user on the whiteboard and sends it to the server, which simply forwards them to other clients. This allows all participants to see the cursor positions of collaborators in real time. The server does not log these positions.

   (d) `undo`: This command is used when a user wishes to undo their most recent action. The server moves the latest action from the `whiteboard_strokes_log` to the `redo_stack`. To inform other users, the server sends a WebSocket message with the inverted action. For example, if a user undoes an addStroke action, the server issues a deleteStroke action, and vice versa.

   (e) `redo`: This command is used when a user wants to redo an action they previously undid. The server moves the latest undone action from the `redo_stack` back to the `whiteboard_strokes_log` and sends a message to connected users that replicates the action being redone. For instance, redoing an addStroke action results in another addStroke command, with the stroke data retrieved from the `redo_stack`.

3. As partially anticipated, the server can issue the following commands via the WebSocket towards the clients:

   (a) `addStroke`: notifies the clients that a new stroke was added by another participant, including the stroke identifier and serialized data necessary to reconstruct the stroke on the client side.

   (b) `deleteStroke`: notifies that a stroke was deleted by another participant. The message will contain the stroke identifier of the stroke being deleted.

   (c) `updateUserList`: Upon user connection or disconnection, the server updates all connected users with the current list of participants.

   (d) `updateUserCursor`: Shares the cursor positions of other participants.

4. Finally, the Erlang server also exposes a function (`handle_whiteboard_change`) that allows our Java application to update the whiteboard data contained in the Erlang nodes whenever a whiteboard is created, deleted or a participant is removed by the owner.

2.2.3 Action broadcasting

When an action requiring propagation is received (for example a new stroke is deleted or added), the main Erlang process sends a message to another Erlang process (the `RPC Manager`), which attempts to perform an RPC on other active nodes to ensure their data is up to date, utilizing a retry-fail mechanism. The list of active nodes is managed by another Erlang process, the `Node Monitor`, which periodically pings the other nodes, filtering out those that do not respond.

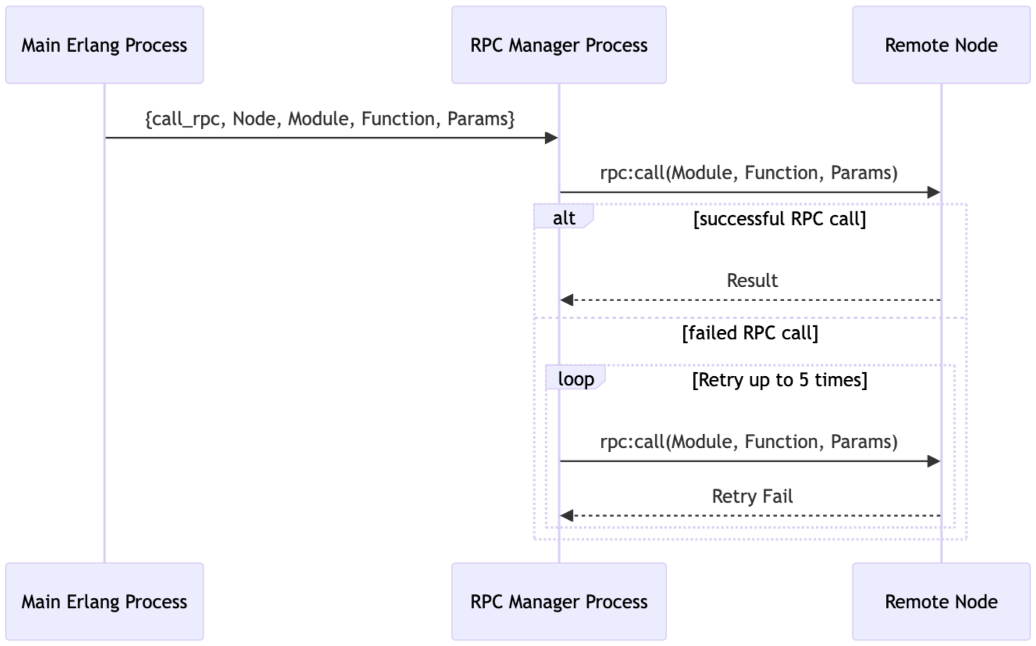


Figure 2: RPC retry-fail mechanism

However, when less critical messages, such as `updateUserCursor`, are sent among nodes and some data loss is tolerable, `rpc:cast` is used instead, as there is no need to ensure that the remote procedure call was executed successfully.

2.2.4 Authenticating users on Erlang using JWTs

Erlang itself does not store information required to verify the user's identity, as that would require further coordination between Erlang and our Java application. To address this, we decided to employ Json Web Tokens to verify user identities on Erlang, relying only on a shared encryption key between Java and Erlang. Upon login, the user's client is given a JWT cookie by the Java application. The same JWT is sent to the Erlang server when entering a whiteboard, so that the Erlang can identify the user by decrypting the JWT payload encrypted by the Java application and perform the necessary permission checks.

2.2.5 Client-side: JavaScript WebSocket

As anticipated, message exchange occurs between the browser client and the WebSocket handler to facilitate real-time communication. The browser client sends and receives messages via a WebSocket implemented using JavaScript. This WebSocket leverages the WebSocket API, which offers a set of methods and events tailored for WebSocket connections.

These methods and events provided by the WebSocket API are mapped to corresponding functions responsible for handling them within the application.

For instance, drawing actions initiated by users are transmitted as messages over the WebSocket connection, allowing for seamless synchronization of whiteboard content across multiple clients. Similarly, functions are implemented to handle collaboration invitations, ensuring that users receive timely notifications and can seamlessly join collaborative sessions.

The whiteboard's front-end was built using Paper.js, a JavaScript library that made it remarkably easier to create, delete and serialize the strokes on the canvas.

2.3 Glassfish

In our project, the web application functionality is implemented using the Java EE application server, Glassfish. This part of the project is structured into three sub-projects, each serving a distinct purpose to ensure modularity and separation of concerns:

1. **web project**: This component contains servlets and Java Server Pages (JSP) and serves as the entry point for incoming requests. Servlets within this project handle incoming HTTP requests and invoke Enterprise JavaBeans (EJBs) to perform business logic operations. The JSP technology is utilized to dynamically generate web pages based on the data retrieved or processed by the servlets.

2. **ejb-interfaces**: This component houses the interfaces for invoking EJBs from servlets, as well as the definition of Data Transfer Objects (DTOs) and enumerated types used across the application. The interfaces define the contract between the web layer and the EJB layer, enabling seamless communication and decoupling of the presentation layer from the business logic.

3. **ejb**: The EJB component contains the implementation of all Enterprise JavaBeans responsible for executing the core business logic of the application. These EJBs encapsulate various functionalities, such as user management, whiteboard creation, collaboration features, and data persistence. By centralizing the business logic within EJBs, we ensure scalability, maintainability, and reusability across different modules of the application.

2.3.1 Web

In the Maven project **web**, which serves as the front end of our application, we have implemented Servlets and Java Server Pages (JSP) to handle dynamic content generation. Each web page is associated with a Servlet responsible for gathering the necessary data and a corresponding JSP to construct the HTML page using JSP technology. This approach allows us to maintain separation of concerns and keep the logic of each web page isolated.

For example, when a user requests a particular page, the corresponding Servlet fetches the relevant data from the backend, processes it, and forwards it to the associated JSP. The JSP then utilizes this data to dynamically generate the HTML content, which is returned to the user's browser for display.

Additionally, the **web** project contains all the resources required for displaying content, including CSS stylesheets, JavaScript files, and images. These assets are organized within the `webapp/assets` folder, ensuring easy accessibility and management. They are utilized to enhance the visual presentation and functionality of the web pages, providing a seamless user experience.

The servlets implemented within the application are organized into folders based on their respective functionalities:

- **Whiteboard Operations:** Servlets responsible for managing whiteboard operations are in the `whiteboardManager` folder. These servlets handle tasks like creating, editing, sharing whiteboards, and managing collaboration features.
- **User Access Management:** Servlets that manage user access to the application, including authentication, authorization, and session management, are in the `accessManager` folder. These servlets ensure secure and seamless user interaction with the platform.
- **User Profile Page:** The `UserProfileServlet` handles requests related to the user profile page, allowing users to view and update their personal information.
- **Homepage:** The `HomepageServlet` manages requests related to the homepage, where users can view their whiteboards and create new ones. This servlet handles operations such as displaying whiteboard previews and filtering whiteboards based on user preferences.
- **Whiteboard Interaction:** The `WhiteboardServlet` facilitates interactions with individual whiteboards, allowing users to draw, erase, and collaborate in real-time. This servlet manages user input and updates to the whiteboard interface.
- **WebSocket Communication:** The `WebSocketServerEndpoint` servlet manages WebSocket communication. Unlike the Erlang implementation, which handles updates related to stroke data, this servlet's WebSocket serves as a notification channel for informing about additions to or removals from a whiteboard.

Each servlet, except for those handling login and signup, accesses session information to check if the current user is already logged in; otherwise, the user is redirected to the login page.

Table 1: Servlet Implementation Methods

Servlet Name	**Implemented Methods**
LoginServlet	POST
SignupServlet	GET/POST
LogoutServlet	GET/POST
DeleteWhiteboardServlet	POST
InsertWhiteboardServlet	POST
ShareWhiteboardServlet	POST
WhiteboardSnapshotServlet	GET/POST
HomepageServlet	GET
UserProfileServlet	GET
WhiteboardServlet	GET

2.3.2 EJB Interfaces

In the `ejb-interfaces` Maven project, we have consolidated all the interfaces essential for communication with Enterprise JavaBeans (EJBs) within the application. Additionally, the `ejb-interfaces` project houses Data Transfer Objects (DTOs) and enumerated types, which play crucial roles in easing data transfer and ensuring type safety within the application. The implemented interfaces are:

1. **UserEJB**
2. **WhiteboardEJB**

2.3.3 EJB

The `ejb` Maven project serves as the implementation repository for the Enterprise JavaBeans (EJBs) defined in the `ejb-interfaces` project, housing the classes that contain the business logic necessary to provide the required functionalities within the application.

The EJBs implemented in this project are of the stateless type, chosen for their scalability, performance, and efficient resource management characteristics. Statelessness ensures that each EJB instance serves multiple clients concurrently, optimizing resource utilization and enhancing system responsiveness.

As previously introduced in the **EJB Interfaces** section, two EJBs were utilized to implement the core business logic of the application:

1. **UserEJBImplementation**: This EJB implementation encapsulates the business logic related to user management within the application. It handles operations such as user authentication, registration, profile management, and access control, ensuring seamless user interaction and secure access to application resources.
2. **WhiteboardEJBImplementation:** Responsible for implementing the business logic associated with whiteboard management and collaboration features. This EJB facilitates operations such as whiteboard creation, editing, sharing, and real-time collaboration among users. It

ensures synchronized access to whiteboard resources and facilitates seamless communication between users during collaborative sessions.

2.3.4 POJOs: DTOs and Enums

- In the Maven projects `ejb-interfaces` and `ejb`, we introduced some Plain Old Java Objects (POJOs) to efficiently share data between EJBs and Servlets and to separate different concerns.
- **Package DTO (project ejb-interfaces):** This package contains all the Data Transfer Objects (DTOs) necessary to transfer information from the business logic tier to the presentation tier. We chose to develop a separate DTO for each piece of information to minimize the transferred data and to better separate each concern.
- **Package enums (project ejb-interfaces):** This package contains the several enum classes, such as the `ParticipantOperationStatus` enum class, which is needed to correctly identify the status of a user in terms of being registered or unregistered and participating or not participating in a certain whiteboard. This package also includes the `SignupStatus` enum class to represent the result of a signup request.

2.4 MySQL

To ensure consistency, permanence, and availability of data, the MySQL server was utilized as the backend database management system.

The final structure of our database can be observed in figure 3 and consists of the following tables:

- **Users**: This table stores information required for user login, including credentials such as username and password, as well as additional personal information to identify the user.
- **Whiteboards**: The "Whiteboards" table contains comprehensive information about each whiteboard within the application. This includes metadata such as whiteboard name, description, creation date, and any other pertinent details. Additionally, the table may store the latest snapshot of each whiteboard for reference.
- **WhiteboardParticipants**: This table establishes the relationship between whiteboards and their associated members. It serves as a junction table, connecting whiteboards to users and indicating their respective roles, such as owner or participant. This enables tracking of whiteboard ownership and membership details.

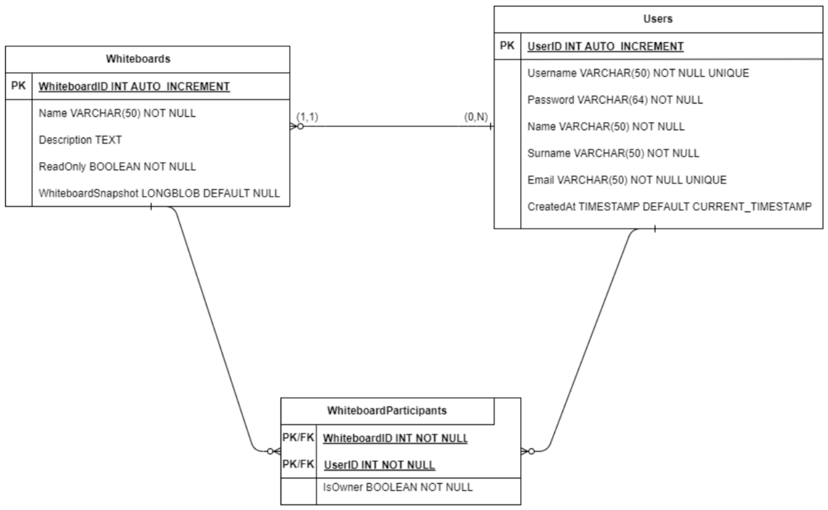

Whiteboards
PK
WhiteboardID INT AUTO_INCREMENT
Name VARCHAR(50) NOT NULL
Description TEXT
ReadOnly BOOLEAN NOT NULL
WhiteboardSnapshot LONGBLOB DEFAULT NULL
(1,1)
(0,N)
Users
PK
UserID INT AUTO_INCREMENT
Username VARCHAR(50) NOT NULL UNIQUE
Password VARCHAR(64) NOT NULL
Name VARCHAR(50) NOT NULL
Surname VARCHAR(50) NOT NULL
Email VARCHAR(50) NOT NULL UNIQUE
CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
WhiteboardParticipants
PK/FK
WhiteboardID INT NOT NULL
PK/FK
UserID INT NOT NULL
IsOwner BOOLEAN NOT NULL

Figure 3: MySQL tables structure

3 User Manual

3.1 Login Page

The login page allows users to enter their username and password to access the platform. An option is also available for unregistered users, directing them to the sign-up page.

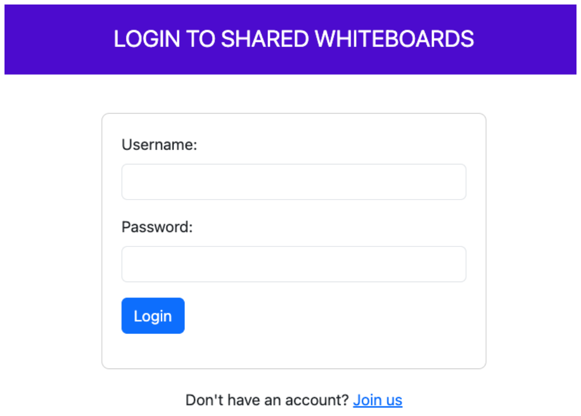
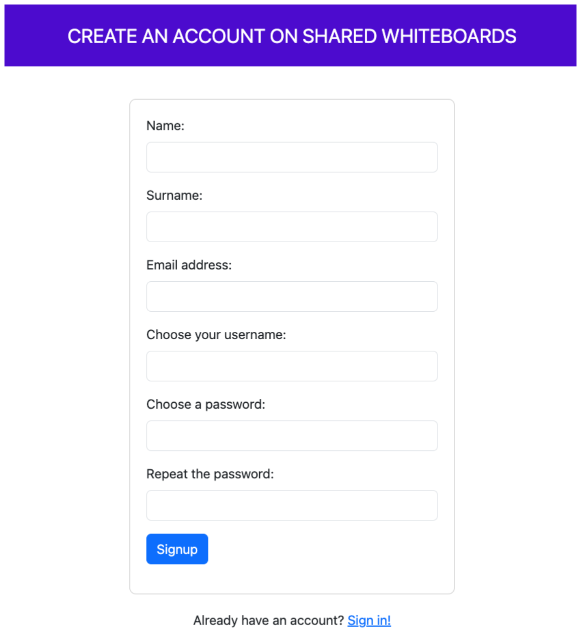


Figure 4: Login page

3.2 Sign-up Page

The sign-up page enables unregistered users to create an account. Users are required to fill in personal information fields. Upon successful sign-up, users are redirected to the login page. If an error occurs during sign-up, an error message will be displayed. The sign-up page enables unregistered users to create an account. Users are required to fill in personal information fields. Upon successful sign-up, users are redirected to the login page. If an error occurs during sign-up, an error message will be displayed.


CREATE AN ACCOUNT ON SHARED WHITEBOARDS

Name:

Surname:

Email address:

Choose your username:

Choose a password:

Repeat the password:

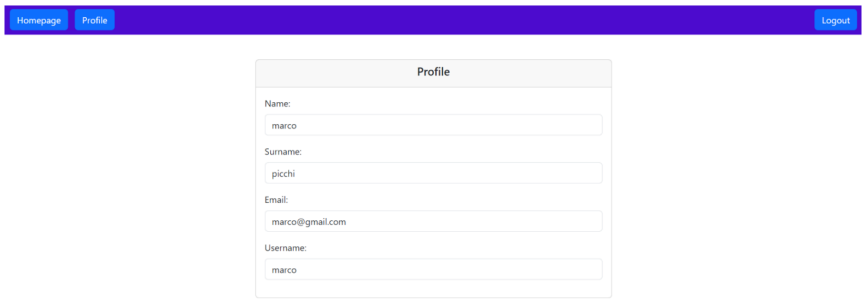
Signup

Already have an account? Sign in!

Figure 5: Sign-up page

3.3 Profile page

The profile page displays the user's personal information, including name, surname, email, and username.



Profile

Name:
marco

Surname:
picchi

Email:
marco@gmail.com

Username:
marco

Figure 6: Profile page

3.4 Homepage

The homepage displays a preview of all whiteboards associated with the user, including those they own and those shared with them. Users can search for whiteboards by name or filter only shared whiteboards. A button allows users to create a new whiteboard by providing a name, description, and indicating if it is read-only. Pressing a whiteboard will redirect you to the whiteboard page. By pressing the "x" button inside the whiteboard name area, the user will either delete the whiteboard or leave it according to whether the user owns or not that whiteboard.

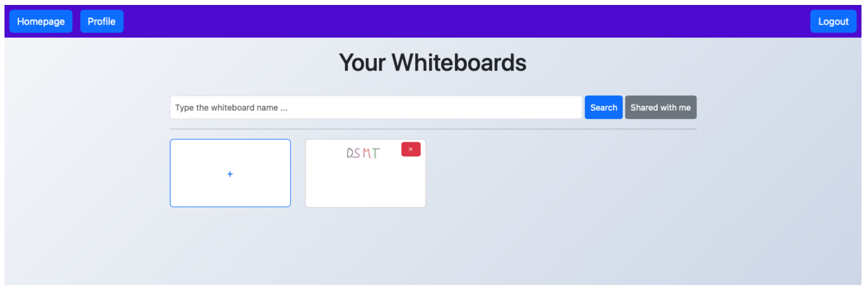


Figure 7: Homepage

3.5 Whiteboard page

The whiteboard page allows users to draw lines with the pencil tool, erase them with the eraser, and undo or redo actions. At any time, you can see in real time, through a colorful slider, the participants "active" to the whiteboard. In addition, pressing the "participant" button will open a modal where you can see all the participants, even those not active at that time. From this modal you can also remove participants, excluding the owner, from the whiteboard. You can also share the whiteboard with other users by pressing the share key and entering the username of the user with whom you want to share the whiteboard. Please note that sharing or removing participants is only allowed to the whiteboard owner.

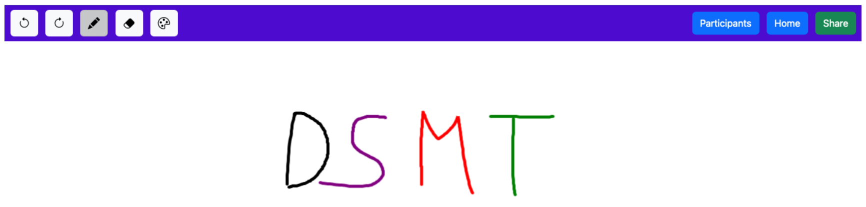


Figure 8: Whiteboard Page