1. Frontend Architecture

The frontend is a **Single Page Application (SPA)** built with **React.js**. It is designed to be fast, responsive, and provide a seamless user experience similar to a desktop application.

- **Component-Based Structure**: The UI is broken down into small, reusable pieces called "components" (e.g.,

Task,

List,

Board,

Navbar

). This makes the code easy to maintain and test.

- **State Management**: We use React's built-in

Context API

and

Hooks

(

useState

,

useEffect

) to manage data. This ensures that when data changes (like adding a new task), the UI updates instantly without reloading the page.

- **Real-Time API**: The frontend maintains a constant connection to the server using **Socket.io-client**. It listens for events like

task_created

or

task_moved

. When these events are received, the board automatically updates, allowing multiple users to see changes instantly.

- **Routing**: We use

React Router

to handle navigation between the Login, Signup, and Dashboard pages without refreshing the browser, preserving the "app-like" feel.

- **Styling**: We use **Tailwind CSS** for styling. It provides a clean, modern, black-and-white minimal aesthetic that is fully responsive across different screen sizes.

2. Backend Architecture

The backend is a **RESTful API** built with **Node.js** and **Express**. It handles all the business logic, database interactions, and real-time communication.

- **Layered Architecture**: The code is organized into three distinct layers to keep it clean and organized:
    1. **Routes**: Define the API endpoints (URLs) that the frontend can call (e.g.,

GET /api/tasks

).

    2. **Controllers**: Contains the logic for each endpoint (e.g., verifying inputs, calculating data).
    3. **Models**: Defines the data structure (Schema) for the database.
- **Database (MongoDB)**: We use **MongoDB**, a NoSQL database, because of its flexibility. Data is stored in JSON-like documents. We have schemas for

Users

,

Boards,

Lists

,

Tasks, and

Activities

. Relationships are handled by referencing IDs (e.g., a Task contains a

boardId

to know which board it belongs to).

- **Authentication & Security**: Security is handled using **JWT (JSON Web Tokens)**. When a user logs in, they receive a token. This token must be sent with every request to prove their identity. User passwords are strictly hashed using

bcrypt

before saving to the database, so they are never stored in plain text.

- **Real-Time Engine (Socket.io)**: This is the core of the collaboration feature. The server uses **Socket.io** to create "rooms" for each Board. When a user enters a board, they join that specific room. If anyone makes a change, the server broadcasts that change *only* to the users in that specific room, ensuring efficient and targeted updates.

Database Schema Design

The application uses **MongoDB** with Mongoose for data modeling. The schema is designed to be relational through references (

ObjectId

), ensuring data integrity while maintaining MongoDB's flexibility.

Entity-Relationship Explanation

- **User**: The central entity. Users can own Boards and be assigned to Tasks.
- **Board**: Owned by a User. Contains multiple Lists.
- **List**: Belongs to a Board. Contains multiple Tasks.
- **Task**: The atomic unit of work. Belongs to a Board and a List. Can be assigned to multiple Users.
- **Activity**: An append-only log of actions (create, update, delete) linked to a Board and User.

Schema Definitions

1. User Collection

json

```
{
  "_id": "ObjectId",
  "username": "String (Unique, Required)",
  "email": "String (Unique, Required)",
  "password": "String (Hashed)",
  "createdAt": "Date"
}
```

2. Board Collection

json

```
{
  "_id": "ObjectId",
  "title": "String (Required)",
  "user": "ObjectId (Ref: User)",
  "createdAt": "Date"
}
```

3. List Collection

json

```
{
  "_id": "ObjectId",
  "title": "String (Required)",
  "board": "ObjectId (Ref: Board)",
  "position": "Number (For ordering)",
  "createdAt": "Date"
}
```

4. Task Collection

json

```
{
  "_id": "ObjectId",
  "title": "String (Required)",
  "description": "String",
  "board": "ObjectId (Ref: Board)",
  "list": "ObjectId (Ref: List)",
  "assignedTo": ["ObjectId (Ref: User)"],
  "position": "Number (For ordering within list)",
  "createdAt": "Date",
  "deadline": "Date"
}
```

5. Activity Collection

json

```
{
  "_id": "ObjectId",
  "board": "ObjectId (Ref: Board)",
  "user": "String (Username snapshot)",
  "action": "String (Enum: 'created', 'updated', 'deleted', 'moved')",
  "details": "String (Description of change)",
  "timestamp": "Date (Default: Now)"
}
```

6️⃣ API Contract Design

The API follows strict **RESTful principles**. All responses are JSON. Authentication is handled via **JWT (Bearer Token)** in the Authorization header.

Base URL:

http://localhost:5000/api

1. Authentication

Method	Endpoint	Description	Request Body	Response
POST	/auth/register	Register new user	{ username, email, password }	{ token, user: { id, username, email } }
POST	/auth/login	Login user	{ email, password }	{ token, user: { id, username, email } }
GET	/auth/user	Get current user info	N/A	{ _id, username, email }

2. Boards

Method	Endpoint	Description	Request Body	Response
GET	/boards	Get all user boards	N/A	[{ _id, title, createdAt }]
POST	/boards	Create a new board	{ title }	{ _id, title, user, createdAt }
GET	/boards/:id	Get single board details	N/A	{ _id, title, lists: [...] }

3. Lists

Method	Endpoint	Description	Request Body	Response
POST	/lists	Create a new list	{ title, boardId }	{ _id, title, board, position }

4. Tasks

Method	Endpoint	Description	Request Body	Response
GET	/tasks	Get tasks for a board	?boardId=...	[{ _id, title, list, assignedTo }]
POST	/tasks	Create a new task	{ title, description, listId, boardId }	{ _id, title, description, list, ... }
PUT	/tasks/:id	Update task (move/edit)	{ title, description, listId, position }	{ _id, title, list, position, ... }
DELETE	/tasks/:id	Delete a task	N/A	{ msg: "Task deleted" }

Standard Error Response

json

```
{
  "msg": "Error message description"
}
```

7️⃣ Real-Time Sync Strategy

The real-time synchronization is built on **Socket.io** using a **Room-based Architecture**. This ensures that updates are only sent to users who are currently viewing the relevant board, minimizing bandwidth usage and preventing unnecessary re-renders.

How it Works:

1. **Connection**: When the frontend application loads, a WebSocket connection is established with the server.
2. **Room Joining**: When a user navigates to a specific Board (e.g., Board ID

123

), the client emits a

join_board

event with that ID. The server adds their socket connection to a dedicated "room" named

board:123

.

3. **Event Broadcasting**:
    - **Action**: User A moves a task from "To Do" to "Done".
    - **API Call**: Frontend sends a standard REST

PUT

request to update the task in the database.

    - **Server Processing**: The server updates the database and **simultaneously** emits a

task_updated

event to the

board:123

room.

    - **Client Update**: User B, who is also viewing Board

123

, receives this event. Their socket listener triggers a state update, instantly moving the task on their screen without a page refresh.

Supported Events:

- task_created

: A new task card appears instantly in the correct list.

- task_updated

: Updates title, description, or position (drag-and-drop sync).

- task_deleted

: Task card disappears from the UI.

- list_created

: A new list column appears on the board.

- activity_log

: Identifying who did what (e.g., "Alice moved Task X") appears in the sidebar live.

Optimistic UI Updates

To make the app feel instant, the frontend implements **Optimistic UI**. When a user drags a task, the UI updates *immediately* before the server confirms the change. If the server request fails (e.g., internet loss), the UI automatically reverts to its previous state to ensure data consistency.