ROTOM Corporation
Discussion of Implementation

Plotline

Version 1.0

Authors: Patrick Hwang, Julia Chaidez,
Emily Panfilo, Julian Cabello, Julianna Arzola

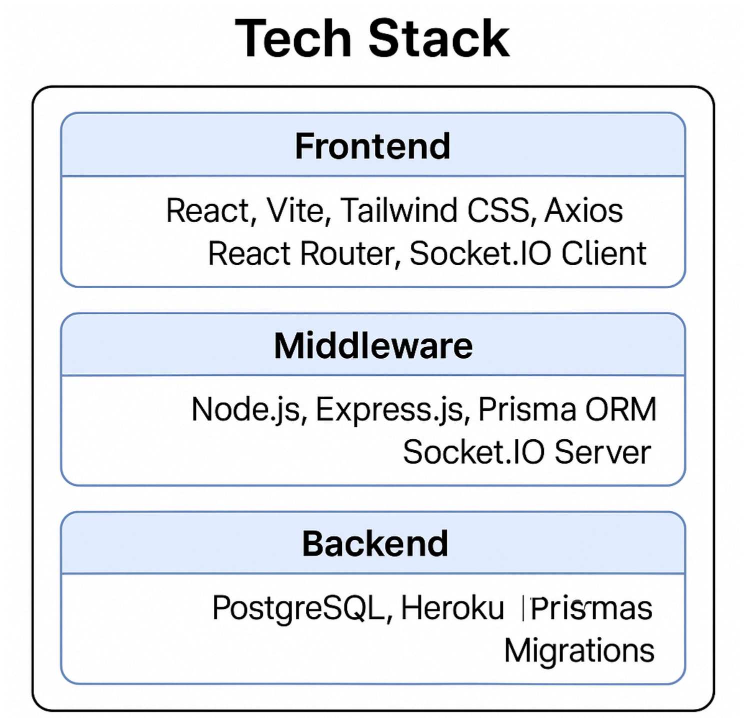
Table of Contents

1. Brief Introduction

1.1 The Product and Its Goals

Plotline is a full social media platform built specifically for book clubs. It's a place where readers join clubs, discuss books, chat in real time, track progress, follow friends, and participate in a community based on reading. Clubs are centered around genres, authors, or specific books. People can also discover new clubs, either by browsing the list of book clubs or by recommendations based on their past reads.

1.2 Tech Stack


Tech Stack
Frontend
React, Vite, Tailwind CSS, Axios
React Router, Socket.IO Client
Middleware
Node.js, Express.js, Prisma ORM
Socket.IO Server
Backend
PostgreSQL, Heroku | Prismas
Migrations

Frontend: React, Vite, Tailwind CSS, Axios, React Router, Socket.IO Client

React serves as the foundation for the frontend. It allows the team to build modular, component-based interfaces, enabling reusable UI elements. In addition, React allows for efficient rendering, which is essential for dynamic features like real-time feeds and interactive chat. Along with the use of React, Vite is used as the frontend build tool and development server, because it offers extremely fast hot-module replacement and optimized production builds. This allows for reduced development turnaround time compared to older tools. Tailwind CSS is also used for styling the project's interface. This was chosen since Tailwind makes it easy to maintain a cohesive design system across all pages, such as the home feed, club pages, and DM interface, without writing custom CSS files.

Paired with the style interfaces, Axios handles all HTTP communication from the frontend to the backend server. Axios handles login and signup requests, retrieves user profiles, fetches book club data, and submits threads, posts, and messages. React Router is also used to manage all client-side navigation, enabling Plotline to behave like a multi-page application while still functioning as a single-page app. Finally, Socket.IO Client library enables real-time communication features such as general club chat, direct messaging, and live notifications.

Middleware / Server Layer: Node.js, Express.js, Prisma ORM, Socket.IO Server

Node.js provides the runtime environment for the server, and its event-driven architecture is ideal for applications requiring high I/O throughput. Express serves as the main backend framework and API gateway. Also, it provides routing, middleware handling, and HTTP utilities. Prisma ORM acts as the database access layer. This translates JavaScript operations into SQL queries and ensures type-safe interaction with PostgreSQL. Socket.IO Server powers Plotline's real-time features.

Backend: PostgreSQL, Heroku, Prisma Migrations

PostgreSQL is the relational database used to store persistent data, and it was chosen because of its reliability, strong relational integrity, and compatibility with Prisma ORM. This then stores users and profiles, book clubs, membership, threads and replies, messages and direct messages, notifications, and user reading progress. Heroku hosts the deployed system since it provides automatic build pipelines, environment variable management, horizontal scaling options, and log aggregation. The schema is version-controlled through Prisma Migrate, ensuring that all developers and production environments share identical database structures.

2. Implementation of the Frontend

2.1 Why We Chose React

React was chosen as the primary framework for Plotline's frontend due to its component-based architecture, efficiency, and extensive ecosystem. Since Plotline is a highly interactive social platform, with my real-time aspects such as live chat, friend interactions, threaded discussions, and frequent UI updates, React's virtual DOM and declarative rendering model make it a strong fit for our purposes. In addition, one of React's main benefits is its reusability. Key interface elements such as club cards, thread previews, chat messages, user profile cards, and friend request components are implemented as reusable components. This allows for consistency across all interfaces and reduces the chances of having duplicate logic in the codebase.

The large ecosystem around React was also a major factor in our decision to choose a frontend development system. The availability of React Router, Tailwind CSS integrations, and Socket.IO client support meant that the team could implement critical features quickly and reliably. In addition to the widely documented and supported videos and web-based sources for React, it made it significantly easier for members who had not previously worked with it to get up to date with its structure and implementation quickly.

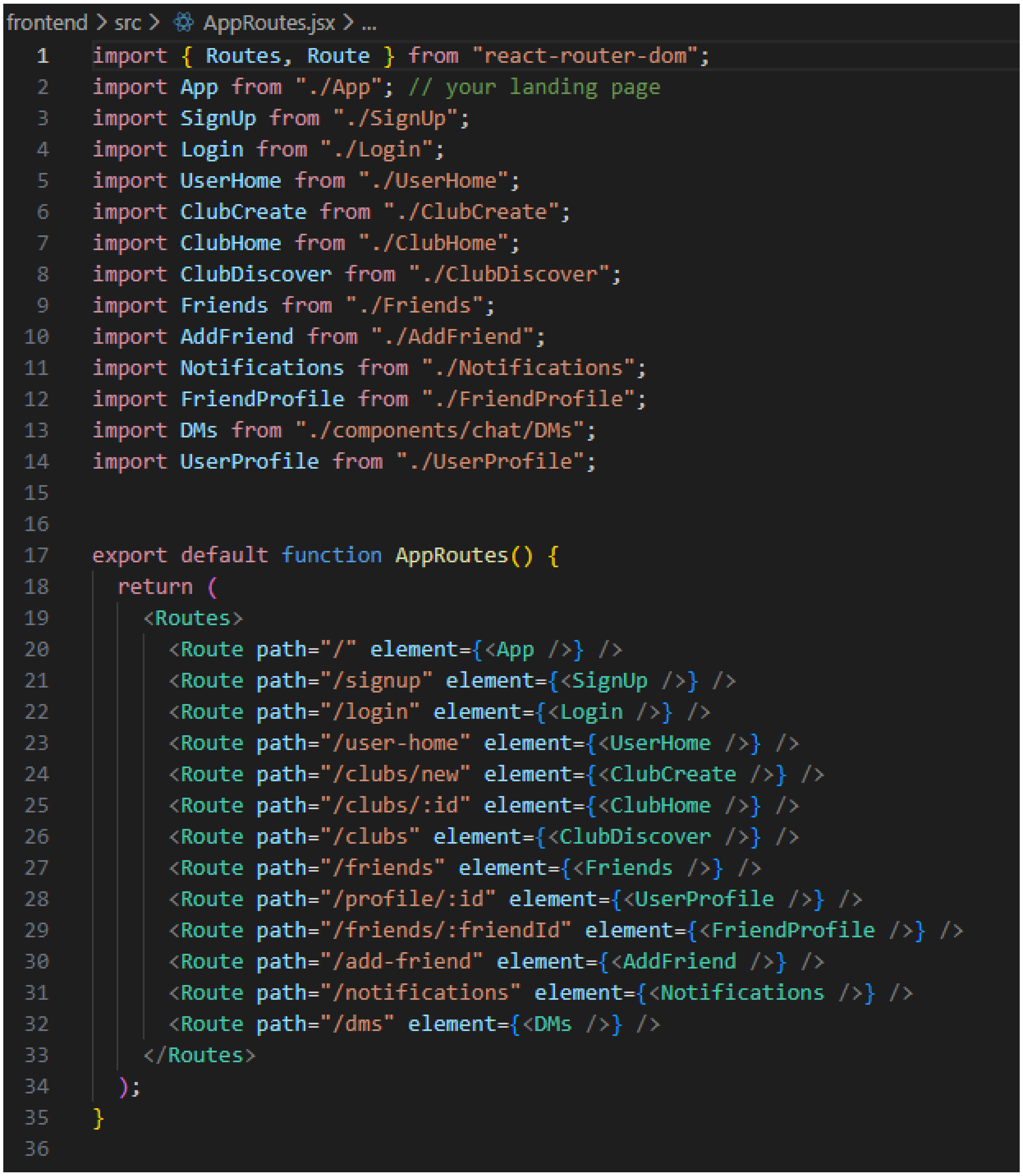
Lastly, choosing React ensured long-term compatibility with modern deployment and build tools, including Vite, which dramatically speeds up development with hot-module replacement and lightweight builds.

2.2 The Router

Routing in Plotline is handled by React Router, which enables the application to function as a fully client-side Single Page Application (SPA). Plotline loads a single HTML page and dynamically switches React components based on URL paths, allowing users to navigate instantly between pages without dealing with full-page reloads.

Inside App.jsx, which serves as the central assembly point for all high-level application pages, is where the router is defined. This file imports the major view components- Home, Club Page, Threads, DMs, Notifications, Profile, Discover, Login, Signup, and Settings, and registers them as routes using < BrowserRouter> and <Routes>.

The structure of the router ensures that navigation remains consistent across every page. There are shared layout components, such as the navigation bar, sidebar, and global notifications listener are mounted outside of the page-level routes so that they appear across the entire application. Plotline also uses lazy loading for several heavier routes, including the Club Page and Thread Page, to reduce initial bundle size. Within lazy loading, it ensures that expensive components, such as the large lists for the messages and threads, are only loaded when the user navigates to the pages associated with them. This improves overall performance for the webpage and allows the homepage and authentication pages to load quickly.


frontend > src > AppRoutes.jsx > ...

```jsx
import { Routes, Route } from "react-router-dom";
import App from "./App"; // your landing page
import SignUp from "./SignUp";
import Login from "./Login";
import UserHome from "./UserHome";
import ClubCreate from "./ClubCreate";
import ClubHome from "./ClubHome";
import ClubDiscover from "./ClubDiscover";
import Friends from "./Friends";
import AddFriend from "./AddFriend";
import Notifications from "./Notifications";
import FriendProfile from "./FriendProfile";
import DMs from "./components/chat/DMs";
import UserProfile from "./UserProfile";


export default function AppRoutes() {
  return (
    <Routes>
      <Route path="/" element={<App />} />
      <Route path="/signup" element={<SignUp />} />
      <Route path="/login" element={<Login />} />
      <Route path="/user-home" element={<UserHome />} />
      <Route path="/clubs/new" element={<ClubCreate />} />
      <Route path="/clubs/:id" element={<ClubHome />} />
      <Route path="/clubs" element={<ClubDiscover />} />
      <Route path="/friends" element={<Friends />} />
      <Route path="/profile/:id" element={<UserProfile />} />
      <Route path="/friends/:friendId" element={<FriendProfile />} />
      <Route path="/add-friend" element={<AddFriend />} />
      <Route path="/notifications" element={<Notifications />} />
      <Route path="/dms" element={<DMs />} />
    </Routes>
  );
}

```

Figure 2.1 — Core routing configuration implemented using React Router, mapping URL paths to major application pages.

When the user navigates to any of these routes, React Router extracts the exact path parameters, which are then passed to API calls through Axios to retrieve the correct data. Additionally, certain pages automatically redirect users based on their state. For example, if a user is in a logged-in state, the router redirects them to the Home page. This is required to prevent authenticated users from accessing pages intended only for new or returning users.

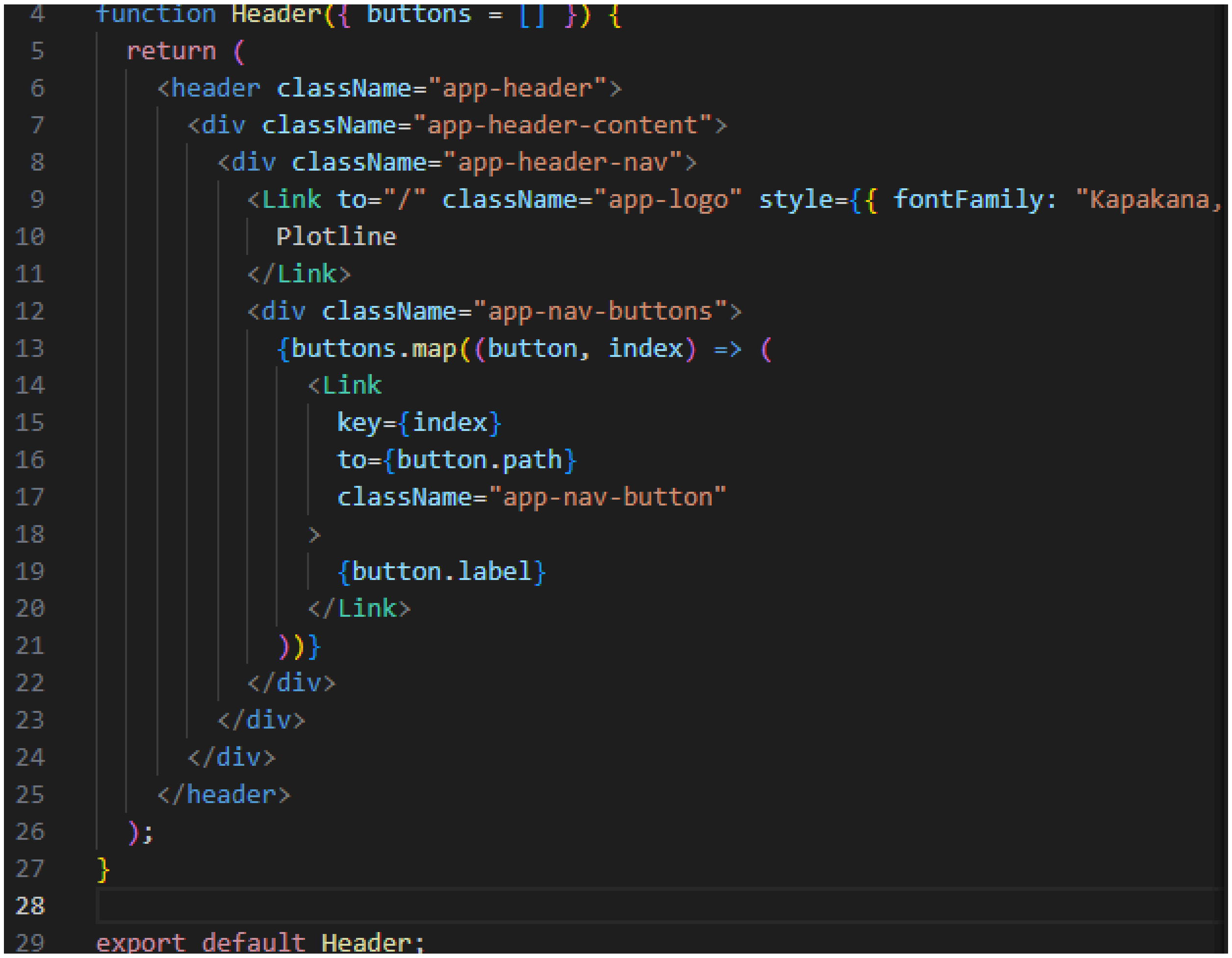
Overall, the router is the backbone of Plotline's navigation system, which enables fast, intuitive movement between features while enforcing authentication rules, lazy loading optimization, and dynamic data retrieval.

2.3 Navigation Bar / Layout

The Navigation Bar is another core component of Plotline's user experience, appearing persistently across nearly every page of the application. The Plotline Navbar is mounted at the top of the level of the application layout so that users can navigate quickly between all the major features of the webpage, including Home, Clubs, DMs, and Profile, regardless of their current page.

This functionality is implemented as a standalone React component and uses Tailwind CSS utility classes to maintain a consistent, responsive design. It has two main purposes: (1) to provide global app navigation and (2) to display dynamic elements that depend on authentication and application state.

The Navbar consists of the following key elements: Logo, Home link, Primary Navigation Links, Search Bar, Real-Time Indicators, and Profile Button. The Logo and Home link are positioned on the far left and double as a shortcut back to the homepage. The Primary Navigation Links are displayed as a hidden horizontal menu containing links to Home, Notifications, Edit Profile, and Logout. Each link again uses React Router components, enabling instant UI updates without full page reloads. The Search Bar is another global element that allows users to look up clubs, users, or books. When the user types a query and hits enter, the application will try its best to navigate to where the results are rendered. It will also populate with suggestions using the debounded frontend state. Real-Time Indicators for features like DMs and notifications, displayed in the Navbar, use dynamic badges that update in real time. The bell icon highlights when new notifications arrive. These updates are handled from Socket.IO events that trigger state updates in the Navbar component.



```
function Header({ buttons = [] }) {
  return (
    <header className="app-header">
      <div className="app-header-content">
        <div className="app-header-nav">
          <Link to="/" className="app-logo" style={{ fontFamily: "Kapakana,
            Plotline
          </Link>
          <div className="app-nav-buttons">
            {buttons.map((button, index) => (
              <Link
                key={index}
                to={button.path}
                className="app-nav-button"
              >
                {button.label}
              </Link>
            ))}
          </div>
        </div>
      </div>
    </header>
  );
}

export default Header;
```

Figure 2.2 — Navigation header displaying the Plotline logo and dynamic navigation links.

To determine whether a user is logged in, the Navbar checks a localStorage token and performs a validation request on mount. If the user is not authenticated, DMs, Notifications, and Profile buttons redirect to login. However, if they are authenticated, all buttons behave normally.

2.4 Home Page (User Home / Club Feed)

The home page is implemented in UserHome.jsx and serves as the main landing page for authenticated users. This is due to it acting as a personalized dashboard, combining information about the user's profile.

UserHome uses the useUser context to access the current user, authentication state, and profile update functions. Once the user enters the site, a useEffet hook checks whether the user is authenticated. If not, the component redirects to /login. This enforces that all Home page features are restricted to only logged-in users.

The layout is composed of three main sections arranged using a responsive grid:

Left Sidebar

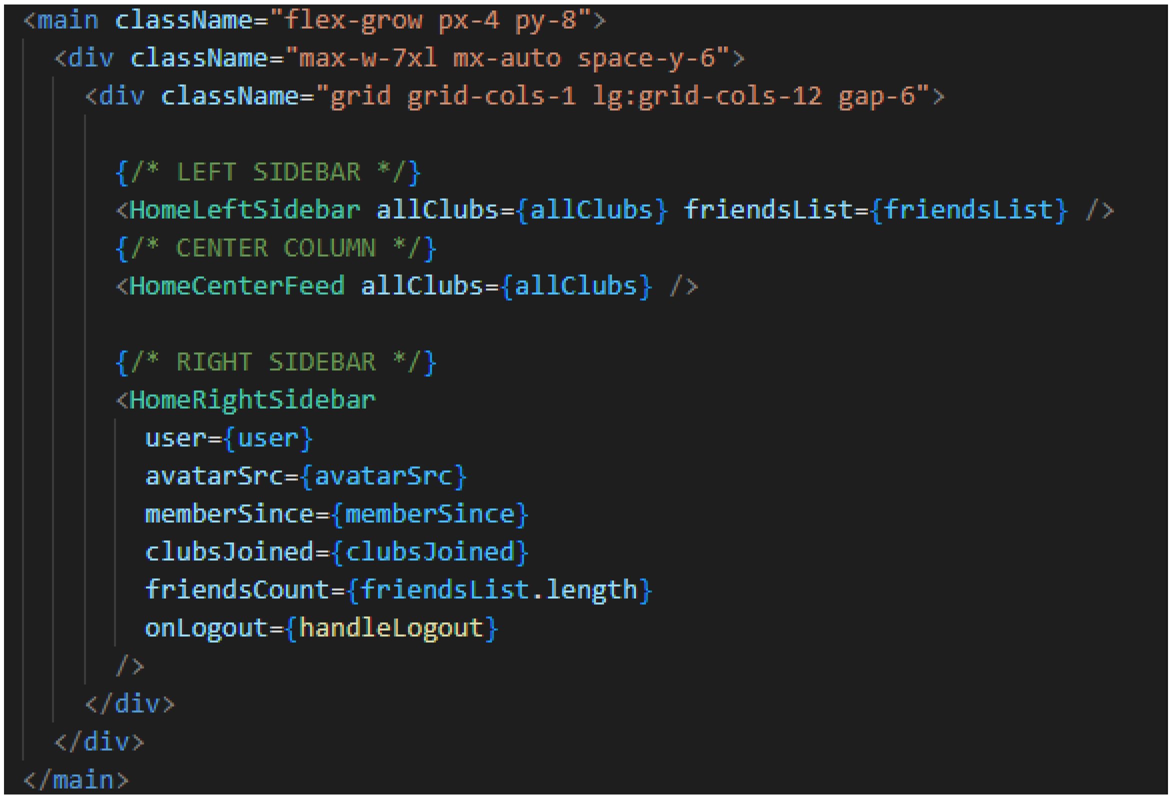
- This feature shows "My Book Clubs", listing clubs that the user is a member of or has created. In addition, within each club entry is rendered as a link, which allows for quick navigation into the club space. This sidebar also includes shortcuts for creating a new club and navigating to the Discover page and DMs.

Center Feed

- As for the Center Feed, this acts as the primary activity area of the webpage. This displays a high-level notification summary by periodically fetching pending friend requests and club invitations using getReceivedFriendRequests and getClubInvitations

Right Sidebar

- Lastly, the right sidebar focuses on the user's reading activity. This includes information such as "My Bookshelf" which is a preview of the current or recently finished books. It also includes the preading progress indicators and optional goal summaries for clubs. There is also a feature that routes to the full bookshelf view.



```
<main className="flex-grow px-4 py-8">
  <div className="max-w-7xl mx-auto space-y-6">
    <div className="grid grid-cols-1 lg:grid-cols-12 gap-6">

      {/* LEFT SIDEBAR */}
      <HomeLeftSidebar allClubs={allClubs} friendsList={friendsList} />
      {/* CENTER COLUMN */}
      <HomeCenterFeed allClubs={allClubs} />

      {/* RIGHT SIDEBAR */}
      <HomeRightSidebar
        user={user}
        avatarSrc={avatarSrc}
        memberSince={memberSince}
        clubsJoined={clubsJoined}
        friendsCount={friendsList.length}
        onLogout={handleLogout}
      />
    </div>
  </div>
</main>
```

Figure 2.3 — Three-column User Home layout composed of sidebars and the central feed.

2.5 Login, Signup, and Profile

The login page allows existing users to sign in using their email and password. This is achieved by using the useState to track form fields and submission status and useUser().login to trigger the actual authentication request. When the formal is submitted, the component sets a logging in message and it calls login(email, password) from the UserContext. This function sends a POST request to the backend, validates credentials, and stores a JWT plus user data in local storage/context.

The Signup page implements account creation. This collects name, email, and password, then calls useUser() signup on form submission. This process is similar to Login.

The profile page allows users to view and manage their account details more deeply than the Home page. Firstly, it gets the current user and authentication status. It then fetches the full profile from the backend into a local full Profile state. Next, it integrates the ProfileEdit modal.

2.6 Club Page (General Chat and Threads)

The main club page, built in ClubHome,jsx, serves as the central hub for each book club on Plotline. This system begins by pulling the club ID from the URL with useParams() and then loads all relevant information through a custom useClubData hook. This call would include the club's description, rules, current book, roles, membership details, reading goals, and the user's individual progress. Again, this page is also divided into three parts.

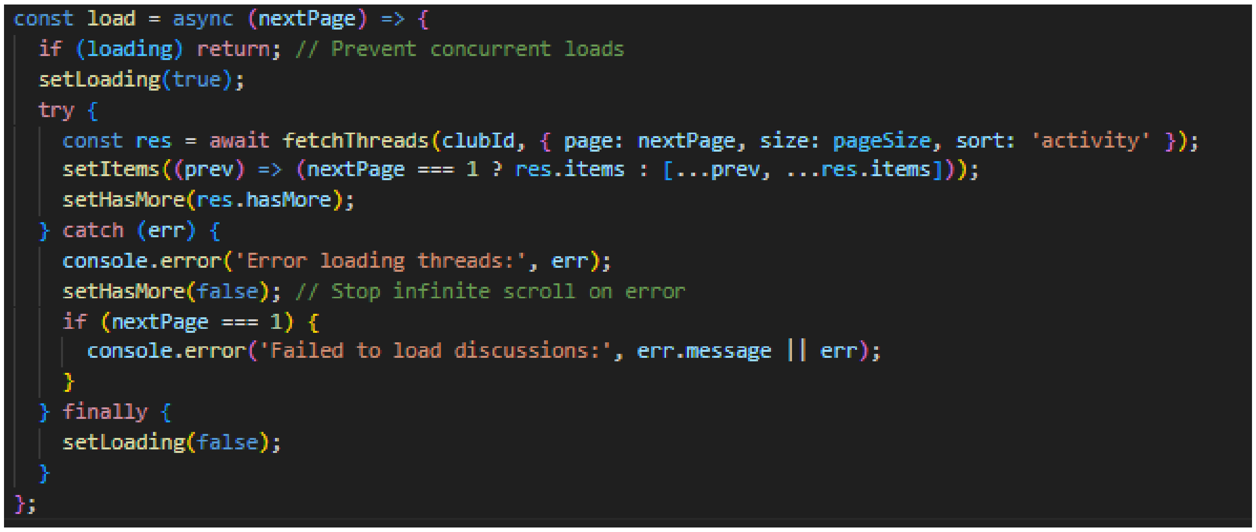
At the top, the combined features of the ClubHeader and ClubTitileBar display the club name, the assigned book, and the role badges for host and moderators. Included in this section are also options to join, leave, or manage the club, all depending on the user's permissions. The left sidebar, which of course is handled by ClubLeftSidebar, presents the club's description, reading schedule, tags, membership roles, and an action car with buttons for tasks like setting reading goals or assigning a book. The center area hosts the live general chat, through LiveChat, and a separate discussions section managed by DiscussionsPanel. On the right side, CLubRightSidebar highlights the user's reading progress and shows their overall progress statistics across members.

2.7 Thread Creation and Replies

Threaded discussions are another major feature for spoiler-conscious, long-form conversation. This is implemented by a set of components under src/components/threads/ and corresponding service modules in src/services/discussions.js.

Within ThreadList, it is responsible for listing threads for a given club and managing infinite scroll and creation. The system accepts clubId, currentUser, isHost, and isMember as props and then computes whether the current user can create a thread using a helper can CreateThread.

The ThreadDetaik.jsx component manages the display and interaction for an individual discussion thread and all of its replies. It starts off by loading the thread's main content and its associated replies through fetchThread. It then processes these replies into a hierarchical structure using buildTree. Each reply is displayed through the ReplyNode component. The component can also call setLastSeen(threadId) to log when a user last viewed the conversation, allowing the interface to display "new" indicators through isNewBadge. As replies are added, edited, or deleted, the reply tree is rebuilt so that the structure on the page always remains consistent with the server's data.



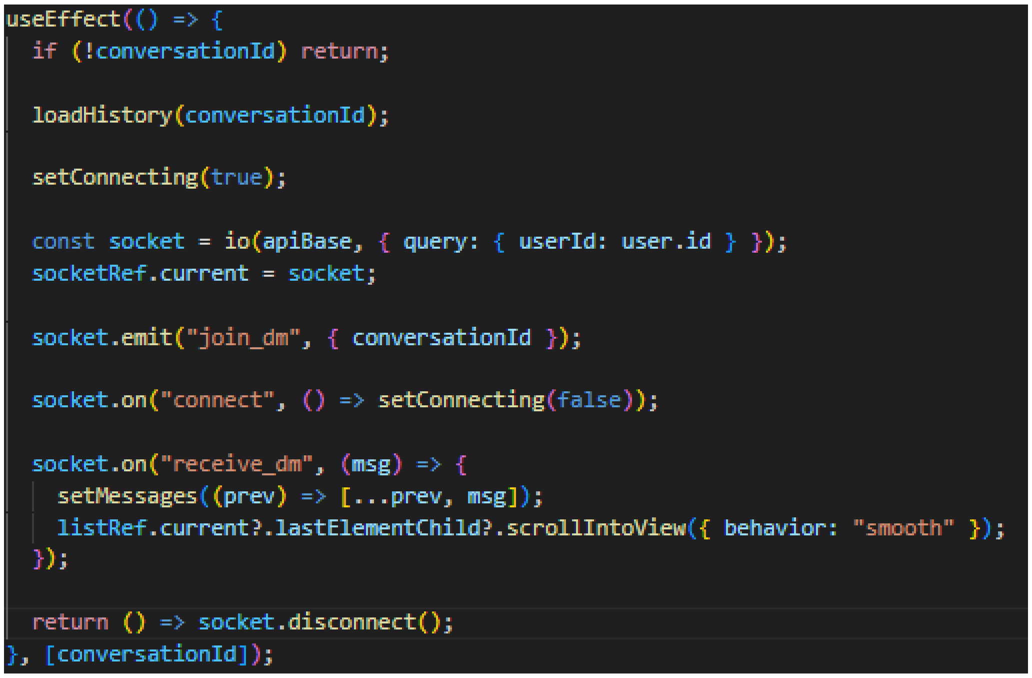
```
const load = async (nextPage) => {
  if (loading) return; // Prevent concurrent loads
  setLoading(true);
  try {
    const res = await fetchThreads(clubId, { page: nextPage, size: pageSize, sort: 'activity' });
    setItems((prev) => (nextPage === 1 ? res.items : [...prev, ...res.items]));
    setHasMore(res.hasMore);
  } catch (err) {
    console.error('Error loading threads:', err);
    setHasMore(false); // Stop infinite scroll on error
    if (nextPage === 1) {
      console.error('Failed to load discussions:', err.message || err);
    }
  } finally {
    setLoading(false);
  }
};
```

Figure 2.4 — Thread loading function implementing pagination, infinite scrolling, and server-side activity sorting.

2.8 Direct Messages (DMs)

The Direct Messages system is implemented through components in src/components/chat/, and the main interface is handled by DMs.jsx, which manages both the user's list of friends and the currently active conversation. The first step of this process is retrieving the logged-in user either from local storage or the backend, and then fetches that user's friend list using the /api/friends/:userId endpoint. Following this, the UI system was designed into two columns: the list of friends on the left and the active chat window on the right.

The DMChat.jsx component is responsible for all real-time messaging and message history within a specific direct message thread. It tracks several values such as the active conversationId returned by the backend, the list of messages, the user's current input text, and connection states. Whenever the component mounts or the selected friend changes, it sends a request to /api/dm/conversation to either fetch or create a DM thread for the two users, then loads the existing message history. After obtaining the conversationId, the component sets up a Socket.IO connection and joins a dedicated room (formatted as dm:<conversationId>). All incoming messages are appended to the chat, and all outgoing messages are filtered through Socket.IO.



```
useEffect(() => {
  if (!conversationId) return;

  loadHistory(conversationId);

  setConnecting(true);

  const socket = io(apiBase, { query: { userId: user.id } });
  socketRef.current = socket;

  socket.emit("join_dm", { conversationId });

  socket.on("connect", () => setConnecting(false));

  socket.on("receive_dm", (msg) => {
    setMessages((prev) => [...prev, msg]);
    listRef.current?.lastElementChild?.scrollIntoView({ behavior: "smooth" });
  });

  return () => socket.disconnect();
}, [conversationId]);
```

Figure 2.5 — Real-time DM socket subscription managing incoming messages and scrolling behavior.

2.9 Friends System (Friends Page and related UI)

The friends system, which is primarily structured in Friends.jsx, gives users a centralized place to manage their social connections outside of individual clubs. First step to verify authentication using useUser(), and then fetches the user's friends with a helper such as getFriends(user.id). Friend-related actions all appear across the app in places like home sidebars, club pages, and notifications. These actions all rely on service functions such as getReceivedFriendRequests, respondToFriendRequest, and similar helpers which allows the logic to remain consistent across multiple pages.

2.10 Notifications System

Notifications are handled through Notifications.jsx, which consolidates updates about friend requests and club invitations. After confirming that the user is authenticated, the page loads three sets of information: incoming friend requests, pending outgoing requests, and any invitations to join clubs. Each item is stored in dedicated state variables and rendered with buttons that let the user accept or decline, depending on the situation. The component keeps a processing set to prevent duplicate submissions. Whenever an action completes, the corresponding list is updated so the page always reflects the newest data. The app also surfaces a notification count through both the HomeCenterFeed and the navigation bar, using a periodic polling cycle (every ten seconds) to display a small badge that alerts the user to pending actions

2.11 Search and Discover (Club Discover Page)

Search and club discovery are handled in ClubDiscover.jsx, which gives users a way to browse new book clubs or locate specific ones. The page relies on useUser() to determine if the visitor is authenticated; if not, they may be redirected or shown limited functionality. When the component mounts, it requests a list of clubs—typically from an endpoint like /api/clubs/discover—and displays each club with information such as its name, description, host, and number of members. Each card includes a button to view or join the club. The global search bar in the navigation menu ties into this page: submitting a query triggers navigation to a URL containing the search term, and ClubDiscover either filters the loaded clubs on the frontend or requests filtered results from the backend. Because of this setup, the same Discover page supports both open browsing and targeted searching without duplicating code.

3. Backend (Middleware) Implementation

3.1 Why We Chose Node.js and Express

For the backend layer, we chose Node.js with Express as our middleware framework due to its JavaScript ecosystem consistency with our React frontend, good real-time capabilities with Socket.io and seamless integration with modern databases like Postgres through Prisma ORM. Express acts as the middleman between the frontend React interface and the database, managing HTTP requests, processing business logic, and ensuring secure and structured data flow.

One of the primary reasons that we chose to use Node.js/Express was for its support of real-time communication through Socket.io. This made it simple for beginners to implement the live chat functionality for each book club and the direct messages between friends. The ability to use JavaScript for both the frontend and the backend simplified development and code sharing between layers. This kept the process straightforward for beginners to minimize context switching.

We also leveraged Prisma and our Object-Relational Mapping (ORM) tool, which significantly simplified database operations. Prisma provided type-safe database queries, automatic migrations, and an intuitive query API that makes fetching and managing data much easier than writing in raw SQL. The Prisma schema file serves as the single source of truth for our database structure and Prisma is able to automatically generate TypeScript types that ensures type safety throughout the application.

By using Express as our middleware, we were able to ensure that our application was secure, scalable, and easy to maintain, while also supporting rapid development and clear separation between frontend and backend responsibilities. This helped us complete this full project in a short amount of time. Additionally, using packages such as bcrypt for password hashing, multer for file uploads, CORS for cross-origin requests, and dotenv for environment variable management were essential for streamlining our development process.

3.2 API Design/Structure

Our Express backend is organized into several main API sections, all defined in the main server.js file with some routes extracted into separate route files for better organization. It should be noted that we aim to further separate server.js into multiple files as we recognize it is too long, but in the interest of time, we have left that to be completed at the end of the project. The main APIs include:

- Authentication API: Handles user signup, login, and session management
- User API: Manages user profiles, account information, and user data
- Club API: Handles book club creation, membership, book assignments, and club management
- Discussion API: Manages discussion posts, replies, and the voting functionality
- Friends API: Handles friend requests, friendships, and friend-related features
- Direct Messages, Handles DM conversations between friends and message history

- Real-time Chat API: Uses Socket.io handlers to live club chat and direct messages

3.3 Authentication API

Our authentication API manages user registration, login, and password handling. The Sign Up route receives a payload that includes name, email address, and password. It uses that information to create a new user account. The password is automatically hashed using bcrypt before being stored in the database. If the email is already registered or there is an issue with the database, an appropriate error is returned. When a user is created, a profile is automatically created with a randomly generated username.

```
// Sign Up Route
app.post('/api/signup', async (req, res) => {
  try {
    const { name, email, password } = req.body;

    if (!name || !email || !password) {
      return res.status(400).json({ error: "All fields are required" });
    }
    const existingUser = await prisma.user.findUnique({ where: { email }
});
    if (existingUser) {
      return res.status(400).json({ error: 'Email already registered' });
    }
    const hashedPassword = await bcrypt.hash(password, 10);
    const user = await prisma.user.create({
      data: {
        name,
        email,
        password: hashedPassword,
        profile: {
          create: {
            username: randomUsername(),
            fullName: name,
            bio: null,
            profilePicture: null,
          },
        },
      },
      include: { profile: true },
    });
    console.log(`✅ New user registered: ${user.email}`);
    res.json({
      message: 'User registered successfully!',
      user: serializeUser(user),
    });
  } catch (error) {
    console.error('❌ Signup error:', error);
    res.status(500).json({ error: 'Server error during signup' });
  }
});
```

The Login route authenticates a user by checking the provided email and password against the database. It uses bcrypt to compare the entered password against the hashed password stored in the database. The code also includes support for plaintext passwords (if they were made before incorporating bcrypt) to be automatically upgraded to have bcrypt hashes.

```
      // User Login Route
      app.post('/api/login', async (req, res) => {
        try {
          const { email, password } = req.body;

          if (!email || !password) {
            return res.status(400).json({ error: "Email and password are 
required" });
          }

          const user = await prisma.user.findUnique({
            where: { email },
            include: { profile: true },
          });
          if (!user) {
            return res.status(401).json({ error: "Invalid email or password" 
});
          }
          // Handle legacy plaintext passwords gracefully and upgrade to 
bcrypt
          const stored = user.password || '';
          let isPasswordValid = false;
          if (typeof stored === 'string' && stored.startsWith('$2')) {
            // Bcrypt hash present
            isPasswordValid = await bcrypt.compare(password, stored);
          } else {
            // Legacy plaintext stored; compare directly then upgrade hash on 
success
            if (password === stored) {
              isPasswordValid = true;
              try {
                const newHash = await bcrypt.hash(password, 10);
                await prisma.user.update({ where: { id: user.id }, data: { 
password: newHash } });
              } catch (e) {
                console.warn('Password hash upgrade failed for user', user.id, 
e?.message);
              }
            } else {
              isPasswordValid = false;
            }
          }
          if (!isPasswordValid) {
            return res.status(401).json({ error: "Invalid email or password" 
});
          }
          res.json({
```

```
        message: "Login successful!",
        user: serializeUser(user),
      });
    } catch (error) {
      console.error("❌ Login error:", error);
      res.status(500).json({ error: "Server error during login" });
    }
  });
```

3.4 User API

The User API handles fetching user data, updating profiles, and managing user information. The Get User Profile route retrieves a single user's information including their profile data. This uses the Prisma's findUnique method with the included option to automatically fetch profile data.

```
// Get single user
app.get('/api/users/:id', async (req, res) => {
  try {
    let userId;
    try {
      userId = parseUserId(req.params.id);
    } catch {
      return res.status(400).json({ error: 'Invalid user id' });
    }
    const user = await prisma.user.findUnique({
      where: { id: userId },
      include: { profile: true },
    });
    if (!user) return res.status(404).json({ error: 'User not found' });
    res.json(serializeUser(user));
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Failed to fetch user' });
  }
});
```

The Update User Profile route allows users to update their profile information including name, email, bio, and profile picture. The code uses Prisma's upsert functionality to either update an existing profile or create a new one if it doesn't exist.

```
// Update user profile
app.put('/api/users/:id', async (req, res) => {
  try {
    let userId;
    try {
      userId = parseUserId(req.params.id);
    } catch {
      return res.status(400).json({ error: 'Invalid user id' });
    }
    const { name, email, profile = {} } = req.body;
    const existingUser = await prisma.user.findUnique({
          where: { id: userId },
          select: { name: true },
        });
        if (!existingUser) {
          return res.status(404).json({ error: 'User not found' });
        }
        const userUpdates = {};
        if (typeof name === 'string') userUpdates.name = name;
        if (typeof email === 'string') userUpdates.email = email;
        const profileUpdates = {};
        if (typeof profile.bio === 'string' || profile.bio === null)
profileUpdates.bio = profile.bio;
        if (typeof profile.profilePicture === 'string' ||
profile.profilePicture === null) {
          profileUpdates.profilePicture = profile.profilePicture;
        }
        if (typeof profile.fullName === 'string') profileUpdates.fullName =
profile.fullName;
        if (profile.username !== undefined) {
          const parsedUsername = String(profile.username).trim();
          if (parsedUsername.length > 0) {
            profileUpdates.username = parsedUsername;
          }
        }
        const data = { ...userUpdates };
        if (Object.keys(profileUpdates).length > 0) {
          const createProfile = {
            username:
              typeof profileUpdates.username === 'string' &&
profileUpdates.username.length > 0
                ? profileUpdates.username
                : randomUsername(),
            fullName:
              profileUpdates.fullName ||
              userUpdates.name ||
              req.body.name ||
              existingUser.name ||
              'New User',
            bio: profileUpdates.bio ?? null,
            profilePicture: profileUpdates.profilePicture ?? null,
          };
          data.profile = {
            upsert: {
              create: createProfile,
              update: profileUpdates,
            },
          };
        }
        if (Object.keys(data).length === 0) {
          return res.status(400).json({ error: 'No valid fields provided'
});
        }
        const user = await prisma.user.update({
          where: { id: userId },
          data,
          include: { profile: true },
        });
```

```
          res.json({ message: 'Profile updated', user: serializeUser(user) });
        } catch (err) {
          console.error(err);
          res.status(500).json({ error: 'Failed to update user' });
        }
      });
```

The Upload Avatar route handles profile picture uploads using multer for file handling. The uploaded file is saved to the server's public/uploads directory with a unique filename, and the relative path is stored in the user's profile.

```
      // Upload avatar
      app.post(
        '/api/users/:id/avatar',
        upload
          ? upload.single('avatar')
          : (req, res, next) => {
              req.file = null;
              next();
            },
        async (req, res) => {
          try {
            if (!upload) {
              return res.status(503).json({ error: 'Avatar uploads unavailable:
multer not installed' });
            }
            if (!req.file) return res.status(400).json({ error: 'No file
uploaded' });
            let userId;
            try {
              userId = parseUserId(req.params.id);
            } catch {
              return res.status(400).json({ error: 'Invalid user id' });
            }
            const relativePath = `/uploads/${req.file.filename}`;
            const existingUser = await prisma.user.findUnique({
              where: { id: userId },
              select: { name: true },
            });
            if (!existingUser) {
              return res.status(404).json({ error: 'User not found' });
            }
            const user = await prisma.user.update({
              where: { id: userId },
              data: {
                profile: {
                  upsert: {
                    create: {
                      username: randomUsername(),
                      fullName: existingUser.name || 'New User',
                      profilePicture: relativePath,
                    },
                    update: {
                      profilePicture: relativePath,
                    },
                  },
                },
              },
              include: { profile: true },
            });
            res.json({
              message: 'Avatar uploaded',
              avatar: relativePath,
              user: serializeUser(user),
            });
          } catch (err) {
            console.error(err);
            res.status(500).json({ error: err.message || 'Failed to upload
avatar' });
          }
        }
      );
```

3.5 Club API

The Club API manages book clubs including their creation, membership, book assignments, and club-related operations. The Create Club route allos for users to create new book clubs. When a club is created, the creator is automatically added as a member so they can participate in the reading goals and discussions.

```
      // Create a new book club
      app.post("/api/clubs", async (req, res) => {
        try {
          const { name, description, creatorId } = req.body;
          // Basic validation
          if (!name || !creatorId) {
            return res.status(400).json({ error: "Name and creatorId are
required." });
          }
          // Create club and include creator info for convenience
          const newClub = await prisma.club.create({
            data: {
              name,
              description,
              creatorId,
            },
            include: {
              creator: {
                include: { profile: true },
              },
            },
          });
          // Automatically add the creator as a member so they can participate
in reading goals
          await prisma.clubMember.create({
            data: {
              clubId: newClub.id,
              userId: Number(creatorId),
              progress: 0,
            }
          });
          // Send structured response
          res.status(201).json({
            message: "Club created successfully!",
            club: newClub,
          });
        } catch (error) {
          console.error("❌ Error creating club:", error);
          res.status(500).json({
            error: error.message || "Server error while creating club.",
          });
        }
      });
```

The Assign Book to Club route allows club creators to assign books to their clubs. When a new book is assigned, any existing current book is automatically archived the club's past read history.

```
      // Assign book to club
      app.put("/api/clubs/:id/book", async (req, res) => {
        try {
          const { id } = req.params;
          const { userId, bookData, readingGoal, goalDeadline } = req.body;
          const club = await prisma.club.findUnique({ where: { id: Number(id)
} });
          if (!club) {
            return res.status(404).json({ error: "Club not found" });
          }
          // Only the creator can assign books
          if (club.creatorId !== Number(userId)) {
            return res.status(403).json({ error: "You are not authorized to
assign books to this club." });
          }
          // If there's a current book, archive it
          if (club.currentBookId && club.currentBookData) {
            await prisma.clubBookHistory.create({
              data: {
                clubId: club.id,
                bookId: club.currentBookId,
                bookData: club.currentBookData,
                assignedAt: club.assignedAt ?? new Date(),
                finishedAt: new Date(),
              },
            });
          }
          // Normalize + clean the book data
          const normalizedBookData = {
            title: bookData.title,
            authors: bookData.authors || bookData.author || "Unknown Author",
            cover: bookData.cover || "",
            description: bookData.description || "No description available.",
            year: bookData.year || null,
            genre: bookData.genre || null
          };
          const updatedClub = await prisma.club.update({
            where: { id: Number(id) },
            data: {
              currentBookId: bookData.title || "",
              currentBookData: normalizedBookData,
              assignedAt: new Date(),
              readingGoal: readingGoal || null,
              goalDeadline: goalDeadline ? new Date(goalDeadline) : null,
            },
          });
          res.json(updatedClub);
        } catch (error) {
          console.error("❌ Error assigning book:", error);
          res.status(500).json({ error: "Server error while assigning book."
});
        }
      });
```

The Join Club route allows for users to join existing book clubs. The code will check if the user is already a member before creating a new membership record for that user.

```
      // Join a club
      app.post("/api/clubs/:id/join", async (req, res) => {
        try {
          const { id } = req.params;
          const { userId } = req.body;
          const club = await prisma.club.findUnique({ where: { id: Number(id)
} });
          if (!club) {
            return res.status(404).json({ error: "Club not found" });
          }
          // Check if user is already a member
          const existingMember = await prisma.clubMember.findUnique({
            where: { clubId_userId: { clubId: Number(id), userId:
Number(userId) } }
          });
          if (existingMember) {
            return res.status(400).json({ error: "User is already a member of
this club" });
          }
          const member = await prisma.clubMember.create({
            data: {
              clubId: Number(id),
              userId: Number(userId),
              progress: 0,
            },
            include: {
              user: {
                select: { id: true, name: true, email: true }
              }
            }
          });
          res.json(member);
        } catch (error) {
```

```
        console.error("❌ Error joining club:", error);
        res.status(500).json({ error: "Server error while joining club." });
      }
    });
```

3.6 Discussion API

The Discussion API manages all the discussion posts, replies, and voting functionalities for each book club discussion. The Create Discussion route allows club members to create new discussion posts. It should be noted that this part of the code will be updated so that only hosts or assigned moderators have this ability.

```
    // Create new discussion post
    app.post('/api/discussion', async (req, res) => {
      try {
        const bodyClubId = req.body.clubId ?? req.body.bookClubID;
        const bodyUserId = req.body.userId ?? req.body.userID;
        const { message, media = [], title, chapterIndex = null, tags = [] }
= req.body;
        const clubId = Number(bodyClubId);
        const userId = Number(bodyUserId);
        if (!clubId || !userId || !message || !title) {
          return res.status(400).json({ error: 'clubId, userId, title and
message are required.' });
        }
        const club = await prisma.club.findUnique({ where: { id: clubId }
});
        if (!club) {
          return res.status(404).json({ error: 'Club not found.' });
        }
        let membership = await prisma.clubMember.findUnique({
          where: { clubId_userId: { clubId, userId } },
          select: { id: true },
        });
        // If not a member yet, auto-join so they can post
        if (!membership) {
          try {
            const created = await prisma.clubMember.create({
              data: { clubId, userId, progress: 0 },
              select: { id: true },
            });
            membership = created;
          } catch (_) {
            // If creation fails, keep membership as null and block
          }
        }
        if (!membership) {
          return res.status(403).json({ error: 'You must be a club member to
create a discussion.' });
        }
        const newDiscussion = await prisma.discussionPost.create({
          data: {
            // Explicitly connect required relations to satisfy Prisma's
checked create input
            club: { connect: { id: clubId } },
            user: { connect: { id: userId } },
            hasMedia: Array.isArray(media) && media.length > 0,
            title: String(title).slice(0, 120),
            chapterIndex: chapterIndex != null ? Math.max(1,
Number(chapterIndex) || 1) : null,
            tags: Array.isArray(tags) ? tags.slice(0, 5) : [],
            content: {
              create: {
                message: String(message).slice(0, 10000),
              },
            },
            media: Array.isArray(media) && media.length > 0
              ? {
                  create: media.map((file) => ({
                    file: file?.path || file?.url || String(file),
                    fileType: file?.type || 'unknown',
                  })),
                }
              : undefined,
          },
          include: {
            user: { select: { id: true, name: true, email: true } },
            content: true,
            media: true,
          },
        });

        res.status(201).json({
          message: 'Discussion post created.',
          discussion: serializeDiscussion(newDiscussion),
        });
```

```
        } catch (error) {
          console.error('Error creating discussion post:', error);
          res.status(500).json({ error: error?.message || 'Failed to create
discussion' });
        }
      });
```

The Vote on Discussion route allows for users to upvote or downvote discussion posts already created. The code handles the vote toggling to ensure they can only up or down vote once. If the user votes the same way twice, the vote is removed.

```
      app.post('/api/discussion/:id/vote', async (req, res) => {
        try {
          const discussionId = Number(req.params.id);
          const { userId, value } = req.body;
          const v = clampVote(value);
          if (!discussionId || !userId || v === null) {
            return res.status(400).json({ error: 'discussionId (in URL),
userId and value (-1,0,1) are required' });
          }
          const discussion = await prisma.discussionPost.findUnique({ where: {
id: discussionId }, select: { id: true, locked: true } });
          if (!discussion) return res.status(404).json({ error: 'Discussion
not found' });
          const summary = await setDiscussionVote(prisma, discussionId,
Number(userId), v);
          res.json(summary);
        } catch (error) {
          console.error('Error casting discussion vote:', error);
          res.status(500).json({ error: 'Failed to cast vote' });
        }
      });
```

3.7 Friends API

The Friend API manages all friend requests and friendships between users. The Send Friend Request route creates a pending friend request between two users. The code will check that the users are not already friends before doing so.

```
//create friendship/send friend request (user sends request to friend)
      app.post('/api/friends', async(req, res) => {
      try {
        const {userId, friendId} = req.body;
        if (!userId || !friendId) {
            return res.status(400).json({error: 'userId and friendId are
required'});
        }
        if (userId == friendId) {
           return res.status(400).json({error: "You cannot send a friend
request to yourself."});
```

```
        }
        const existingRequest = await prisma.friend.findFirst ({
           where: {
                  OR: [
                   {userID: Number(userId), friendID: Number(friendId)},
                   {userID: Number(friendId), friendID: Number(userId)},
                   ],
                  },
        });
        if(existingRequest) {
           return res.status(400).json({error: "Friend request or friendship
already exists."});
        }
      const friendship = await prisma.friend.create ({
            data: {
                   userID: Number(userId),
                   friendID: Number(friendId),
                   status: "PENDING",
             },
             });
           res.json({message: "Friend request sent.", friendship});
           } catch (error) {
           console.error('Error sending friend request:', error);
           res.status(500).json({error: 'Failed to send friend request'});
           }
      });
```

The Respond to Friend Request route allows users to accept or decline pending friend requests. An accept will create a bidirectional friendship between the users.

```
      //accept or decline friend request
      app.post("/api/friends/respond", async (req, res) => {
          try {
              const {userId, friendId, friendStatus} = req.body;
              if (!userId || !friendId || !friendStatus) {
                 return res.status(400).json({error: "userId, friendId, and
friendStatus are required."});
              }
              const request = await prisma.friend.findFirst({
                where: {
                        userID: Number(friendId),
                        friendID: Number(userId),
                        status: "PENDING",
                        }
              });
              if (!request) {
                 return res.status(404).json({error: "No pending friend
request."});
             }
             const updatedStatus = await prisma.friend.update ({
                 where: {id: request.id},
                 data: {status: friendStatus},
                });
                if (friendStatus == "ACCEPTED") {
                   const reverseExists = await prisma.friend.findFirst({
                       where: {
                          userID: Number(userId),
                          friendID: Number(friendId),
                          },
                    });
                if(!reverseExists) {
                   await prisma.friend.create({
                   data: {
                         userID: Number(userId),
                         friendID: Number(friendId),
                          status: "ACCEPTED",
                          },
                   });
                }
               // Get user names for notification message
               const [accepter, requester] = await Promise.all([
               prisma.user.findUnique({ where: { id: Number(userId) }, select:
{ name: true } }),
               prisma.user.findUnique({ where: { id: Number(friendId) },
select: { name: true } }),
               ]);
               res.json({
               message: "Friend request accepted.",
               accepterName: accepter?.name || "User",
               requesterName: requester?.name || "User",
               });
               }
               if (friendStatus == "DECLINED") {
                // Delete the pending request when declined
                await prisma.friend.delete({
                    where: { id: request.id },
                    });
                res.json({ message: "Friend request declined." });
                }
                } catch (error) {
                  console.error("Error responding to request: ", error);
                 res.status(500).json({error: "Server error while responding
to friend request."});
                }
      });
```

3.8 Direct Messages (DM) API

The Direct Messages API manages DM conversations and message history between any users that are friends on the website. This is currently in a separate route file for better organization.The Get or Create Conversation route is used to find existing DM conversations between two users or create one if it doesn't already exist.

```
      router.get("/conversation", async (req, res) => {
        try {
          const userId = Number(req.query.userId);
          const friendId = Number(req.query.friendId);
          if (!userId || !friendId) {
            return res.status(400).json({ error: "Missing userId or friendId"
  });
          }
          // Check for existing DM conversation
          let convo = await prisma.directMessage.findFirst({
            where: {
              OR: [
                { user1Id: userId, user2Id: friendId },
                { user1Id: friendId, user2Id: userId }
              ],
            },
          });
          // If no conversation exists, create it
          if (!convo) {
            convo = await prisma.directMessage.create({
              data: {
                user1Id: userId,
                user2Id: friendId,
              },
            });
          }
          res.json({ conversationId: convo.id });
        } catch (err) {
          console.error("Error loading DM conversation:", err);
          res.status(500).json({ error: "Server error" });
        }
      });
```

The Get DM Message route retrieves the message history for a specific conversation. It is ordered chronologically.

```
    // GET DM message history
    router.get("/messages/:conversationId", async (req, res) => {
      try {
        const conversationId = req.params.conversationId;

        const messages = await prisma.dMMessage.findMany({
          where: {
            convoId: conversationId
          },
          orderBy: { createdAt: "asc" },
          include: {
            sender: {
              select: {
                id: true,
                name: true,
                profile: { select: { profilePicture: true } }
              }
            }
          }
        });

        res.json(messages);
      } catch (err) {
        console.error("Error loading DM messages:", err);
```

```
          res.status(500).json({ error: "Failed to load messages" });
        }
      });
```

3.9 Real-time Chat API

The Real-time Chat API uses Socket.io to provide instant messaging functionalities for both club chat and direct messaging. This is implemented in a separate socket.js file.

The Socket.io server is initialized in server.js and uses middleware to authenticate connections by requiring a userId in the connection query parameters. Once a user is authenticated, the socket connection is set up with handlers for various real-time events on the website.

The Join Club Chat handler allows users to join a club’s chat room. The code will verify that the user is actually a member of the club before allowing them to join the room.

```
      socket.on("joinClub", async ({ clubId }) => {
            try {
              const cId = Number(clubId);
              if (!cId || !userId) return;
              const isMember = await isClubMember(cId, userId);
              if (!isMember) {
                console.warn(`User ${userId} tried joining club ${cId} but is
not a member`);
                return;
              }
              socket.join(clubRoom(cId));
              console.log(`User ${userId} joined room ${clubRoom(cId)}`);
            } catch (err) {
              console.error("Error joining club room:", err);
            }
          });
```

The Send Club Message handler processes messages sent in the club chat. It validates the message, checks membership, and saves the message to the database. It will then broadcast it to all the members in the club’s chat room.

```
      socket.on("sendMessage", async ({ clubId, content }, ack) => {
            try {
              const cId = Number(clubId);
              const text = (content || "").trim();
              if (!cId || !userId || !text) {
                return ack?.({ ok: false, error: "Invalid club message data"
});
              }
              const isMember = await isClubMember(cId, userId);
              if (!isMember) {
                return ack?.({ ok: false, error: "Not a member of this club"
});
```

```
       }
       const saved = await prisma.message.create({
         data: {
           content: text.slice(0, 2000),
           userId,
           clubId: cId,
         },
         include: {
           user: {
             select: {
               id: true,
               name: true,
               profile: { select: { profilePicture: true } },
             },
           },
         },
       });
       const baseUrl =
         process.env.BASE_URL || "http://localhost:3001";
       const message = {
         id: saved.id,
         clubId: saved.clubId,
         content: saved.content,
         createdAt: saved.createdAt,
         user: {
           id: saved.user.id,
           name: saved.user.name,
           profilePicture: saved.user.profile?.profilePicture
             ?
`${baseUrl}${saved.user.profile.profilePicture.startsWith("/") ? "" :
"/"}${saved.user.profile.profilePicture}`
             : null,
         },
       };
       io.to(clubRoom(cId)).emit("newMessage", message);
       ack?.({ ok: true, message });
      } catch (error) {
       console.error("Error sending club message:", error);
       ack?.({ ok: false, error: "Failed to send message" });
      }
     });
```

The Send Direct Message handler processes direct messages between friends. It validates the conversation, checks if the users are friends, saves the messages, and then broadcasts them to both users in the conversation.

```
    socket.on("send_dm", async ({ conversationId, senderId, content }, ack)
=> {
      try {
       if (!conversationId || !senderId || !content) {
         return ack?.({ ok: false, error: "Invalid DM payload" });
       }
       const convo = await prisma.directMessage.findUnique({
         where: { id: conversationId },
       });
```

```
        if (!convo) {
          return ack?.({ ok: false, error: "Conversation not found" });
        }
        const receiverId =
          convo.user1Id === senderId ? convo.user2Id : convo.user1Id;
        const areFriends = await prisma.friend.findFirst({
          where: {
            status: "ACCEPTED",
            OR: [
              { userID: senderId, friendID: receiverId },
              { userID: receiverId, friendID: senderId },
            ],
          },
        });

        if (!areFriends) {
          return ack?.({ ok: false, error: "Users are not friends" });
        }
        const message = await prisma.dMMessage.create({
          data: {
            content,
            convoId: conversationId,
            senderId,
            receiverId,
          },
          include: {
            sender: {
              select: {
                id: true,
                name: true,
                profile: { select: { profilePicture: true } },
              },
            },
          },
        });
        io.to(`dm_${conversationId}`).emit("receive_dm", message);
        ack?.({ ok: true, message });
      } catch (err) {
        console.error("Error sending DM:", err);
        ack?.({ ok: false, error: "Server error sending DM" });
      }
    });
```

By using Socket.io for real-time communication we elevate our website to provide instant connection opportunities for our users. They won't need to constantly refresh for new messages, but can see them easily appear like other common social media applications. It was able to quickly get this feature working and provide a seamless chat experience that is intuitive and responsive.

4. Database Design

4.1 Why We Selected PostgreSQL

For our backend, we selected PostgresSQL, an open source relational database management system known for its reliability, scalability, and support of complex data types and relationships. It is great for structured data and can provide performance while still handling complex queries such as searching, filtering, or real-time operations. This makes it an ideal choice for a social application like Plotline.

The schema for our databases was designed to support essential functionality outlined in our Software Requirement Document. This includes functionality for book club applications such as user accounts, book clubs, discussion posts, replies, direct messages, friendships, and club memberships. Each model was defined in our Prisma schema file, which serves as a single source of truth for our database structure. Prisma automatically generates migrations that create the corresponding tables in PostgreSQL, allowing us to maintain normalized relationships between users and their interactions.

PostgreSQL's support for JSON data types was particularly useful for storing flexible book data structures, allowing us to store book information from external APIs without requiring rigid schema changes. The database also supports efficient indexing, which we leveraged for frequently queried fields such as club IDs, user IDs, and conversation IDs in our messaging system.

Overall, using PostgreSQL as our database enabled us to create a structured, secure, scalable, and functioning backend that supports all the core features of Plotline while allowing for future growth and maintainability.

4.2 Overview of Tables

4.2.1 Club Table

Attributes	Field Type
id	integer
name	text
description	text
createdAt	timestamp without time zone
creatorId	integer

currentBookId	text
currentBookData	jsonb
readingGoal	text
goalDeadline	timestamp without time zone
assignedAt	timestamp without time zone

4.2.2 ClubBookHistory Table

Attribute	Field Type
id	integer
clubId	integer
bookId	text
bookData	jsonb
assignedAt	timestamp without time zone
finishedAt	timestamp without time zone

4.2.3 ClubInvitation Table

Attribute	Field Type
id	integer
clubId	integer
inviterId	integer
inviteeId	integer
status	USER-DEFINED
createdAt	timestamp without time zone
updatedAt	timestamp without time zone

4.2.4 ClubMember Table

Attribute	Field Type
id	integer
clubId	integer
userId	integer
progress	integer
joinedAt	timestamp without time zone

4.2.5 DMMessage Table

Attribute	Field Type
id	integer
content	text
senderId	integer
receiverId	integer
convoId	text
createdAt	timestamp without time zone

4.2.6 DirectMessage Table

Attribute	Field Type
id	integer
user1Id	integer
user2Id	integer
createdAt	timestamp without time zone
updatedAt	timestamp without time zone

4.2.7 DiscussionPost Table

Attribute	Field Type
id	integer
datePosted	timestamp without time zone
dateEdited	timestamp without time zone
hasMedia	boolean
title	text
chapterIndex	integer
locked	boolean
pinned	boolean
tags	jsonb
clubId	integer
userId	integer
contentId	integer

4.2.8 DiscussionPostContent Table

Attribute	Field Type
id	integer
message	text

4.2.9 DiscussionPostVote Table

Attribute	Field Type
id	integer
userId	integer

discussionId	integer
value	integer

4.2.10 DiscussionReply Table

Attribute	Field Type
id	integer
discussionId	integer
parentId	integer
userId	integer
body	text
createdAt	timestamp without time zone
updatedAt	timestamp without time zone

4.2.11 DiscussionReplyVote Table

Attribute	Field Type
id	integer
userId	integer
replyId	integer
value	integer

4.2.12 Friend Table

Attribute	Field Type
id	integer
userID	integer
friendID	integer

status	USER-DEFINED

4.2.13 Message Table

Attribute	Field Type
id	integer
clubId	integer
userId	integer
content	text
createdAt	timestamp without time zone

4.2.14 Profile Table

Attribute	Field Type
id	integer
username	text
fullName	text
profilePicture	text
bio	text
joinDate	timestamp without time zone
userId	integer

4.2.15 User Table

Attribute	Field Type
id	integer
name	text
email	text

password	text
createdAt	timestamp without time zone

5. Deployment Strategy

5.1 Full-Stack Deployment via Heroku

To ensure that Plotline was accessible, performant, and scalable, we deployed our application using Heroku. This is a platform-as-a-service (PaaS) that simplifies deployment and scaling of web applications. Our deployment strategy focused on ensuring a smooth user experience across all components while minimizing operational overhead. We chose to deploy both the frontend and the backend of Plotline on a single Heroku dyno. Heroku's build system automatically builds our React/Vite frontend during deployment and serves it as static files through our Express backend. The Heroku-postbuild script in our package.json handles the frontend build process by copying all the built files into the backend's public directory where Express serves them. The steps for our deployment process are as follows:

1. Heroku installs dependencies for both frontend and backend
2. The frontend is built using Vite, creating optimized production assets
3. Built frontend files are copied to backend/public directory
4. Prisma Client is generated to ensure database types are up to date
5. The Express server starts and serves both API routes and static frontend files

In all honesty another main reason for choosing Heroku was for its free student option and easy ability to deploy a Github branch. However, we found that the single-dyno approach really simplified the deployment process for us and reduced the costs while still providing us with a functioning deployed website.

5.2 PostgreSQL via Heroku Postgres

We hosted our Postgres database through Heroku Postgres which automatically provisioned as an add-on when deploying to Heroku. This helped manage our Postgres database with automatic backups, connection pooling, and seamless integration with our application through the DATABASE_URL environment variable.

The database connection is managed through Prisma which reads the DATABASE_URL from the .env file and then automatically migrates during deployment though the release phase of our Procfile. This separation of data storage and application server allows for better scalability and data persistence.

5.3 Media Storage via Local File System

The only source of media we handle (as specified in our SRS document) is profile photo images uploaded by the user. To handle the profile picture uploads, we store the image files locally on the Heroku's dyno's filesystem in the backend/public/uploads directory. Files are served statically through Express, making them accessible via URLs such as /uploads/filename.jpg.

While this approach is currently working okay for our small scale project, we recognized that it has limitations. The files are lost when the dyno restarts and do not persist after the restart. We plan to migrate to AWS S3 bucket or something more production grade for future development. The file upload implementation uses multer for handling requests, validates file types (JPG/PNG only), and generates unique filenames to prevent any conflicts.

5.4 Security and Configuration

We implemented HTTPS for secure communication through Heroku's automatic SSL certificate management. All traffic to our application is automatically encrypted, ensuring secure transmission of sensitive data like passwords and user information.

Additionally, we used environment variables (a common practice) for managing secrets like our database connection string. These variables are stored securely in Heroku's config vars and loaded using dotenv in development. This very importantly keeps our sensitive data safe. We also used bcrypt for hashing user passwords, meaning that passwords are never stored in plaintext. CORS (Cross-Origin Resource Sharing) is configured to allow requests from our frontend origin. Socket.io connections require authentication through a userId query parameter, ensuring only authenticated users can establish real-time connections.

6. Programming Patterns

6.1 Decorator Pattern (Structural Pattern)

According to the notes, the Decorator pattern “dynamically adds features/functionality to a component by wrapping the core component with another component that adds extra functionality.” In Plotline, we use the Decorator pattern extensively in React components, where higher-order UI wrappers extend or modify the behavior of core components without changing their internal logic.

Frontend Example — Layout and Page Wrappers

Many pages in Plotline wrap their main component with shared “decorator” components such as:

- Navigation bar
- Sidebars
- User dropdown
- Auth guard wrappers
- Modal containers

For example, pages like ClubHome.jsx, UserHome.jsx, and Friends.jsx wrap their central content with layout components such as:

```
return (

  <>

    <UserDropdown />   // added functionality (logout, profile editing)

    <ProfileEdit />    // added UI functionality

    <div className="home-container">

      <HomeLeftSidebar />

      <HomeCenterFeed />

      <HomeRightSidebar />
    </div>
  </>
);
```

These decorators *wrap* the page to add cross-cutting behavior (navigation, modals, UI features) without modifying the page's main logic. This matches the structural definition in the notes: adding extra behavior by wrapping the original component.

6.2 Command Pattern (Behavior Pattern)

Frontend Example — Service Modules as Command Objects

All actions in Plotline that modify server state are abstracted into service modules stored under src/services/:

```
export function createThread(data) {
  return api.post('/threads', data);
}
export function createReply(data) {
  return api.post('/replies', data);
}
```

Each service function encapsulates:

- The **data** (arguments: thread title, reply text, IDs)
- The **behavior** (the actual HTTP call)

React components do not perform network operations themselves—they simply *invoke commands*.

7. Challenges and Solutions

7.1 Real-Time Messaging and Synchronization Issues

Challenge:
Implementing a real-time messaging system for both general club chat and direct messages (DMs) introduced issues such as duplicated messages, missing events, and socket listeners not properly joining conversation rooms. When multiple clients joined or left conversations, inconsistent UI states were observed, especially when switching between DM threads.

Solution:
The team standardized the socket event workflow by:

- Ensuring every DM conversation had a unique room (dm_<conversationId>).
- Emitting join_dm immediately after establishing the socket connection.
- Broadcasting messages only to the appropriate room using io.to(room).emit(...).
- Adding cleanup logic to disconnect the socket on component unmount.
- Implementing optimistic UI updates so messages appear instantly, regardless of network delay.

7.2 Club Membership Logic and Permission Enforcement

Challenge:
Permission logic inside book clubs (host, moderator, member, non-member) became complicated when implementing actions such as thread creation, replying, assigning books, and updating reading progress. Several UI components incorrectly enabled or disabled features based on outdated membership state.

Solution:
The team moved all membership and role checks into centralized backend handlers. The frontend now uses a single source of truth returned by the backend (membershipStatus) instead of deriving its own permissions. This eliminated UI inconsistencies and ensured that unauthorized actions were fully blocked at the API layer.

8. Future Improvements

8.1 Migrate Image and File Storage to AWS S3 or Cloud Storage

Currently, profile photos are stored on Heroku's ephemeral filesystem, which is not persistent. A cloud storage provider (e.g., AWS S3, Google Cloud Storage, Supabase Storage) would enable:

- Permanent image hosting
- Faster content delivery through CDNs
- Scalable bandwidth for large user bases
- This is the highest-priority improvement for production-readiness. Explanation here.

8.2 Enhanced Notification System

Possible upgrades include:

- Push notifications (mobile)
- Email summaries of club activity
- Notification categorization (social, club, system)
- Real-time read/unread sync

This would make the platform feel more polished and responsive.

9. Statistics

This section summarizes the quantitative aspects of the Plotline implementation, including file count, lines of code, number of major components, and overall architecture sizing. These statistics help demonstrate the scope, complexity, and effort involved in building a full-stack social platform with real-time capabilities.

9.1 Lines of Code (Approximate)

These counts exclude third-party dependencies, build artifacts, autogenerated files, and compiled assets. Only project-authored source files are included.

Frontend (React + Vite + Tailwind)

Component Area	LOC
Page components (/src)	~4,200
Club-related components (/components/club)	~1,600
Thread + Reply system (/components/threads)	~830
Direct Messages (/components/chat)	~390
Home components (/components/home)	~375
Global components (/components)	~330
Context Providers	~170
Services + API adapters (/services)	~640
Custom Hooks	~200
Total Frontend LOC	~8,900
Total Project LOC (Frontend + Backend)	

≈ 12,200 lines of code

This reflects the substantial scope of the application, including real-time messaging, a complex relational schema, modular frontend routing, and multi-layered UI components.

9.2 File Count

Frontend:
~53 React component files
~18 service and API modules
~3 CSS files
~8 utility/hook files

Total: ~82 authored frontend files

Backend:
~4 main JS files (server, routes, socket)
~3 Prisma files (schema + migrations)
Total: ~7 authored backend files

9.3 Function Count (Estimated):

A function count was estimated by scanning .jsx and .js files for both named and arrow function signatures.

Frontend: ~574 functions (React components, handlers, effect logic, services)
Backend: ~111 functions (route handlers, middleware logic, socket listeners, Prisma actions)

Total: ~685 functions throughout the codebase.

This reflects the highly modular design required for a complex social reading platform.

9.4 Architectural Complexity Overview

- Database Models: 11+ (User, Profile, Club, Membership, Thread, Reply, Message, DMConversation, Notification, ReadingProgress, etc.)
- Routes and Endpoints: 40+ total API endpoints
- Real-Time Event Types: ~8 (DM send/receive, chat send/receive, notification emit, join room events, etc.)
- Pages: ~12 major pages (Login, Signup, Home, Profile, Club Home, Threads, Discover, Friends, Notifications, DM, Settings, etc.)
- Modals: 12+ (Thread creation, reply edit/delete, reading goal modal, invite friends modal, edit profile modal, assign book modal)

Plotline's implementation demonstrates a full-scale, production-style architecture with layered components, real-time infrastructure, relational consistency, and a strong separation of concerns.

10. Conclusion

The development of Plotline resulted in a complete, fully functional social platform designed specifically for book clubs. The project integrates real-time chat, threaded discussions, friend connections, notifications, reading progress tracking, and robust profile management into one cohesive system.

Despite initial challenges—including schema synchronization, role/permission enforcement, real-time synchronization, and state consistency—the team successfully designed and deployed a scalable and maintainable system. The use of programming patterns (Decorator, Command, Adapter) strengthened the software architecture and aligned the implementation with best practices taught in class.

Overall, Plotline serves as a strong example of an end-to-end full-stack application: feature-rich, modular, scalable, and grounded in real-world social platform design. The final system not only meets the project requirements but provides a strong foundation for future growth, such as cloud storage integration, mobile support, richer discussion tools, and recommendation systems.