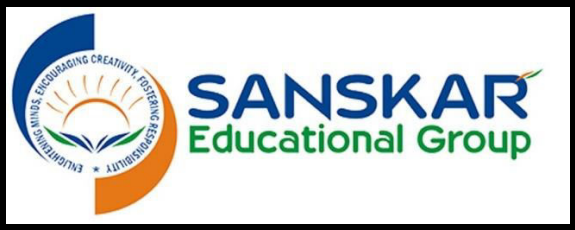
A
MAJOR PROJECT REPORT
ON
STREAMIFY



BY

ANKIT GAUTAM (2102150100015)
AREEBA ASIF (2102150100019)
SATYAM GUPTA (2102150100076)
SATYAM KUMAR (2102150100077)

Submitted to

Mr. Mahesh Sharma

(Head of Department)

Department of Computer Science & Engineering

Sanskar College of Engineering & Technology

(Affiliated to Dr. A.P.J Abdul Kalam Technical University, Lucknow)

(Session 2024-25)

A

MAJOR PROJECT REPORT

ON

STREAMIFY

BY

ANKIT GAUTAM (2102150100015)

AREEBA ASIF (2102150100019)

SATYAM GUPTA (2102150100076)

SATYAM KUMAR (2102150100077)

Submitted to the Department of Computer Science & Engineering in partial Fulfillment of the requirement for the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

UNDER GUIDASNCE OF

MISS. DEEPIKA BANSAL

ASST. PROFESSOR

(C.S.E DEPARTMENT)

SUBMITTED TO

MR. MAHESH SHARMA

HEAD OF DEPARTMENT

(C.S.E DEPARTMENT)

Sanskar College of Engineering & Technology

(Affiliated to Dr. A.P. J Abdul Kalam University, Lucknow)

(Session 2024-25)

A

MAJOR PROJECT REPORT

ON

STREAMIFY

By

ANKIT GAUTAM (2102150100015)

AREEBA ASIF (2102150100019)

SATYAM GUPTA (2102150100076)

SATYAM KUMAR (2102150100077)

Submitted to

The Department of Computer Science & Engineering

Sanskar College of Engineering & Technology

(Affiliated to Dr. A.P.J Abdual Kalam Technical University, Lucknow)

(Session 2024-25)

TABLE OF CONTENT

LIST OF FIGURES

DECLARATION

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to substantial. Extent has been accepted for the award of any degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

SIGNATURE:

NAME: ANKIT GAUTAM

ROLL NO: 2102150100015

DATE:

SIGNATURE:

NAME: AREEBA ASIF

ROLL NO: 2102150100019

DATE:

SIGNATURE:

NAME: SATYAM KUMAR

ROLL NO: 2102150100077

DATE:

SIGNATURE:

NAME: SATYAM GUPTA

ROLL NO: 2102150100076

DATE:

CERTIFICATE

This is to certify that the dissertation entitled **"STREAMIFY"** which is submitted By Ankit Gautam (2102150100015), Areeba Asif (2102150100015), Satyam Kumar (2102150100077), Satyam Gupta (2102150100076) in partial fulfilment of the requirement for the award of degree of Bachelor of technology, A.P.J Abdul Kalam Technical University, Lucknow is a record of bonafide work carried out by them under my guidance and supervision. The result embodied in this thesis has not been submitted to any other university or institution for the award of any degree.

Internal Guide

DEPARTMENT:

Asst. Prof. Miss. DEEPIKA BANSAL

(Department of CSE.)

HEAD OF

DEPARTMENT:

Asst. Prof. Mahesh Sharma

(Department of CSE.)

ACKNOWLEDGMENT

It is indeed with a great pleasure and immense sense of gratitude that we acknowledge the help of these individuals. We are highly indebted to our Director Dr. Dharmendra Singh, Sanskar College of Engineering and Technology, for the facilities provided to accomplish this main project.

We would like to thank our Asst. Prof. Mahesh Sharma, Head of the Department of Computer Science and Engineering, Sanskar College of Engineering and Technology, for this constructive criticism throughout our project.

We feel elated in manifesting our sense of gratitude to our internal project guide, Department of Computer Science and Engineering, Sanskar College of Engineering and Technology. He has been a constant source of inspiration for us and we are deeply thankful to him for his support and valuable advice.

We are extremely grateful to our Departmental staff members, Lab technicians and Non-teaching staff members for their extreme help throughout our project.

Project Associates:

ANKIT GAUTAM (2102150100015)

AREEBA ASIF (2102150100019)

SATYAM GUPTA (2102150100076)

SATYAM KUMAR (2102150100077)

ABSTRACT

Streamify is a real-time communication web application that integrates secure messaging and peer-to-peer video calling within a modern, responsive interface. Developed using a full-stack JavaScript architecture, Streamify employs React for the frontend, Node.js and Express.js for backend operations, and MongoDB for database management.

Real-time chat functionality is powered by Socket.IO, enabling instant message exchange and live notifications. For video calling, WebRTC is utilized to establish direct, browser-based peer-to-peer connections, eliminating the need for third-party software. User authentication and session management are handled securely through JWT (JSON Web Tokens).

Streamify provides users with a seamless communication experience and is deployable on cloud platforms such as Heroku or DigitalOcean, making it accessible and scalable. This project serves as both a functional application and a comprehensive learning opportunity in modern web development and real-time systems.

CHAPTER:1.1 INTRODUCTION

Streamify is a powerful, full-stack real-time communication platform designed to revolutionize how users connect, chat, and interact online. Built with modern web technologies, Streamify offers a seamless experience that combines instant messaging, peer-to-peer video calling, and secure user authentication, all in one responsive and scalable web application.

In an age where digital communication is at the heart of personal and professional interactions, Streamify delivers a reliable solution that emphasizes speed, usability, and interactivity. Whether for casual conversations, collaborative work, or virtual meetups, Streamify provides users with the tools to stay connected in real time.

At its foundation, Streamify leverages **React** for a smooth and dynamic frontend interface, ensuring a responsive user experience across devices. On the backend, **Node.js** and **Express.js** manage API requests, handle real-time connections, and ensure robust server-side logic. **MongoDB** powers the database, efficiently storing user data, chat history, and call metadata.

Real-time chat functionality is enabled through **Socket.IO**, allowing for instant message exchange, typing indicators, and live notifications. To take interaction to the next level, **WebRTC** is integrated for high-quality, peer-to-peer video calling, enabling face-to-face communication directly within the browser—no third-party apps needed.

Security is a core feature of Streamify. With JWT (JSON Web Tokens), users are authenticated securely, and sessions are managed efficiently to prevent unauthorized access and ensure data privacy.

Streamify is also deployment-ready. With support for cloud platforms like Heroku or DigitalOcean, it can be easily launched and scaled for users worldwide.

GOAL OF THIS PROJECT:

The primary goal of **Streamify** is to develop a robust, scalable, and user-friendly web application that enables real-time communication through instant messaging and peer-to-peer video calling. By integrating modern web technologies such as React, Node.js, Socket.IO, and WebRTC, the project aims to provide a seamless digital communication experience that is both secure and responsive across all devices.

1.2 OBJECTIVE OF THE SYSTEM:

The main objective of Streamify is to create an integrated real-time communication platform that supports both instant messaging and video calling with an emphasis on speed, security, and user experience. The system is intended to:

1. **Enable Seamless Communication:**
   Provide users with real-time text and video communication capabilities in a single platform.
2. **Ensure Secure Access:**
   Implement JWT-based authentication to manage user sessions and protect user data.
3. **Deliver Cross-Device Compatibility:**
   Build a fully responsive frontend using React to ensure smooth usability on desktops, tablets, and smartphones.
4. **Support Real-Time Features:**
   Use Socket.IO to enable features like typing indicators, message delivery confirmations, and instant updates.
5. **Facilitate High-Quality Video Calls:**
   Integrate WebRTC for peer-to-peer, plugin-free video calling directly in the browser.
6. **Offer a Scalable and Maintainable Architecture:**
   Design the system using modular full-stack architecture (MERN stack) to allow easy upgrades, scalability, and maintenance.

7. **Provide Deployment Readiness:**
   Ensure the system can be deployed on cloud platforms like Heroku or Digita lOcean for global access.

1.3. PRELIMINARY SYSTEM ANALYSIS

The preliminary system analysis of **Streamify** focuses on evaluating the feasibility and overall direction of building a real-time communication platform that integrates both instant messaging and video calling features. The primary motivation behind the system is to address the growing need for secure, efficient, and user-friendly digital communication, particularly in remote work, virtual collaboration, and online social interaction.

The analysis begins with identifying the limitations in existing platforms, such as dependency on third-party tools, lack of integration between chat and video functionalities, and minimal customization options for specific use cases. Streamify proposes a web-based solution built on modern technologies such as React, Node.js, MongoDB, Socket.IO, and WebRTC, which ensures real-time data exchange, secure user sessions through JWT, and a responsive interface across devices.

From a technical feasibility standpoint, the chosen tech stack is well-supported, scalable, and ideal for real-time applications. Operationally, the system is designed to offer a seamless experience to end-users with features like live messaging, typing indicators, and peer-to-peer video calls without the need for external plugins. Financially, the development can be managed using open-source tools and deployed on cost-effective cloud services like Heroku or DigitalOcean, making it accessible and maintainable.

This phase also defines the system's boundaries and ensures that it aligns with the intended goals of creating a secure, integrated, and scalable communication platform. Overall, the preliminary system analysis confirms that Streamify is a viable project with a clear purpose, achievable objectives, and strong potential for real-world application and future enhancements.

Existing System:

- Messaging and video calling are often split across multiple applications.
- Inconsistent user experience due to multiple interfaces and logins.
- Reliance on third-party services leads to latency and security concerns.
- Weak or outsourced authentication systems compromise data privacy.
- Limited customization and flexibility in closed-source platforms.
- Higher resource and bandwidth usage due to fragmented tools.
- Not optimized for cross-device responsiveness or user-friendly design.

Proposed System:

- Combines instant messaging and video calling in a single platform.
- Provides a consistent and responsive user interface using React.
- Uses Socket.IO for real-time messaging with low latency.
- Employs WebRTC for secure, peer-to-peer video communication without third-party plugins.
- Implements JWT-based authentication for secure user login and session management.
- Fully customizable and open-source, allowing flexibility for various use cases.
- Designed for scalability and easy deployment on platforms like Heroku or DigitalOcean.
- Optimized for cross-device compatibility, ensuring smooth performance on all screens.

CHAPTER 2: REQUIREMENTS SPECIFICATIONS

2.1 INTRODUCTION

The Requirements Specification document for Streamify outlines the functional and non-functional requirements necessary to design, develop, and implement a real-time communication platform that supports both instant messaging and video calling. This document serves as a blueprint for developers, designers, and stakeholders by clearly defining what the system must do, how it should perform, and under what constraints it will operate.

2.2 HARDWARE REQUIREMENTS

- **Processor:** Dual Core 2.0 GHz or higher
- **RAM:** Minimum 4 GB (8 GB recommended for smooth video calling)
- **Storage:** At least 500 MB of free space for browser cache and temporary files

2.3 SOFTWARE REQUIREMENTS

- **Operating System:** Ubuntu 20.04+, Windows Server, or any Unix-based OS
- **Runtime Environment:** Node.js (v14 or above)
- **Framework:** Express.js
- **WebSocket Library:** Socket.IO
- **Video Calling API:** WebRTC
- **Database:** MongoDB (local or cloud-hosted via MongoDB Atlas)
- **Authentication:** JSON Web Tokens (JWT) for secure session handling
- **Version Control:** Git
- **Repository Hosting:** GitHub or GitLab
- **Deployment Platform:** Heroku, DigitalOcean, or AWS
- **Database Hosting:** MongoDB Atlas (for cloud database services)

CHAPTER 3: FEASIBILITY ANALYSIS

3.1 INTRODUCTION

The feasibility analysis for **Streamify** evaluates whether the proposed real-time communication platform can be successfully developed and implemented within given constraints. This chapter examines the project from technical, operational, and economic perspectives to ensure that the system is achievable, functional, and cost-effective. By identifying potential challenges and assessing available resources, this analysis helps determine the overall viability of Streamify before proceeding to full-scale development.

3.2 TECHNICAL FEASIBILITY

The technical feasibility of Streamify confirms that the proposed system can be developed using current technologies and tools effectively and efficiently. Streamify is built on a modern and robust technology stack including React.js for the frontend, Node.js and Express.js for the backend, MongoDB for the database, and Socket.IO along with WebRTC for real-time communication and video calling features.

- **Modern Tech Stack**: Utilizes React.js, Node.js, Express.js, MongoDB, Socket.IO, and WebRTC—proven, reliable, and scalable technologies.
- **Open-Source Tools**: All core technologies are open-source, reducing cost and allowing customization.
- **Real-Time Communication**: Supports real-time chat and video calls through WebRTC and Socket.IO without third-party dependencies.
- **Cloud Deployment**: Compatible with platforms like Heroku and for easy scalable deployment.
- **Developer Resources**: Ample documentation and community support are available for each technology used.
- **Security Support**: Implements JWT for secure user authentication and session management.
- **Scalability**: Can be scaled horizontally to support more users as demand grows.

3.3 OPERATIONAL FEASIBILITY

- **User-Friendly Interface:** Streamify uses React to deliver a clean, intuitive, and responsive design, making it easy for users to navigate and interact.
- **Seamless Integration:** Combines messaging and video calling in one platform, eliminating the need for multiple apps.
- **Minimal User Training**: The familiar layout and web-based interface reduce the learning curve for end-users.
- **Real-Time Interaction**: Features like instant messaging, typing indicators
- **Low Learning Curve**: The design follows standard UI/UX practices, requiring minimal to no training for new users.
- **Secure & Reliable**: Implements JWT authentication and peer-to-peer video connections for safe, uninterrupted communication.
- **Responsive Design**: Fully functional on desktops, tablets, and smartphones, enabling flexible usage across environments.
- **Real-Time Efficiency**: Instant messaging and live video enhance decision-making, collaboration, and productivity.
- **Scalability in Operations**: Can accommodate a growing number of users without affecting usability or performance.

3.4 ECONOMICAL FEASIBILITY

Economical feasibility analyzes the cost-effectiveness of the proposed system by comparing the estimated development and operational costs with the expected benefits. For Streamify, the goal is to create a powerful real-time communication platform while keeping expenses minimal through the use of open-source technologies and scalable cloud infrastructure.

- **Low Development Cost**: Built using open-source technologies like React, Node.js, Express, and MongoDB, reducing software licensing expenses.
- **Free Communication Protocols**: WebRTC and Socket.IO provide real-time capabilities without the need for paid third-party services.
- **Affordable Deployment**: Can be hosted on low-cost cloud platforms like Heroku, DigitalOcean, or AWS free-tier, minimizing hosting charges.
- **Minimal Hardware Investment**: Requires only standard computing devices and internet access—no high-end hardware is needed.

- **Scalable Architecture**: Designed to grow with user demand, allowing costs to scale gradually with usage instead of upfront bulk spending.
- **Open-Source Ecosystem**: Access to free tools, libraries, and community support lowers maintenance and upgrade costs.
- **Cost-Efficient Training**: User-friendly interface reduces the need for extensive user training, saving time and resources.
- **Long-Term Value**: Combines multiple services (chat, video call) into one platform, lowering the need for additional subscriptions or tools.

CHAPTER 4: TECHNOLOGIES USED

The development of **Streamify** leverages a set of modern, scalable, and performance-oriented technologies suited for real-time web applications. Each technology plays a specific role in building a responsive, interactive, and reliable communication platform.

4.1 FRONTEND Development Tools

The frontend is responsible for providing users with a smooth and interactive experience. To build the user interface (UI), the following tools and frameworks were used:

React:

React is an open-source JavaScript library developed by Facebook, used for building dynamic and interactive user interfaces, especially for single-page applications (SPAs).

- **Role in Streamify:**
  React serves as the foundation of Streamify's frontend. It helps build a responsive and seamless user interface where components can update in real-time without reloading the page. This is essential for dynamic features like chat windows, video call interfaces, and live status updates.

- **Key Features in Streamify:**
  - Component-based architecture for easy code reusability and maintenance.
  - Efficient UI rendering using the virtual DOM.
  - Real-time UI updates when messages are sent or received.
  - Responsive layout adaptable to both desktop and mobile devices.
  - Smooth integration with WebRTC and Socket.IO-client for real-time functionality.

Socket.IO-client:

Socket.IO-client is the frontend library that enables real-time, bidirectional, and event-based communication between the client and the server using WebSockets.

- **Role in Streamify:**
  Socket.IO-client connects the user interface to the backend server, allowing users to send and receive messages instantly. It also powers real-time features such as typing indicators, message delivery acknowledgments, and user online/offline status updates.

- **Key Features in Streamify:**
  - Enables real-time messaging without refreshing the page.
  - Handles connection events like user join, disconnect, and reconnection seamlessly.
  - Supports custom events for notifications, message updates, and typing indicators.
  - Works in sync with the backend Socket.IO server for low-latency communication.
  - Critical for achieving a live chat experience within the application.

4.2 Backend Development Tools

The backend of Streamify is responsible for handling server-side logic, managing real-time connections, storing and retrieving user data, and ensuring secure communication. To achieve this, Streamify uses a powerful combination of Node.js, Express.js, and Socket.IO. Together, these technologies form a lightweight, high-performance backend suitable for real-time, event-driven applications.

Node.js

Node.js is a runtime environment that allows JavaScript to be run on the server side. It uses the V8 engine (the same as Chrome) and provides a non-blocking, event-driven architecture for high scalability.

- **Role in Streamify:**

  Node.js serves as the core server environment for Streamify, powering the backend logic and enabling asynchronous communication, which is crucial for handling multiple users and real-time events efficiently.

- **Key Features in Streamify:**

  - Handles concurrent connections with non-blocking I/O for real-time messaging.
  - Executes JavaScript on the server side for full-stack development in one language.
  - Supports WebSockets and integrates seamlessly with Socket.IO.
  - Offers a fast runtime, enabling low-latency interactions.

Express.js

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features to build web and mobile applications.

- **Role in Streamify:**

  Express.js handles HTTP routes, APIs, and middleware in Streamify. It manages user registration, login, chat history retrieval, and other RESTful operations, making the app responsive and secure.

- Key Features in Streamify:

  - Simplifies API routing for user authentication and data handling.
  - Middleware support for authentication, logging, and error handling.
  - Enables modular code structure for maintainability.
  - Works in tandem with MongoDB to interact with the database.

Socket.IO

Socket.IO is a library that enables real-time, bidirectional communication between clients and servers using WebSockets and fallback technologies.

- **Role in Streamify:**

  Socket.IO is the backbone of real-time communication in Streamify. It manages live chat, typing indicators, online statuses, and video call signaling.

- **Key Features in Streamify:**
  - Supports event-based communication (e.g., message sent, user typing).
  - Manages persistent WebSocket connections for low-latency messaging.
  - Enables scalable broadcast and room-based messaging (e.g., for group chats or private rooms).
  - Handles connection, reconnection, and error events gracefully.

4.3 Database Development Tools

A robust and flexible database is essential for managing user accounts, chat history, session data, and other dynamic content in a real-time application like Streamify. To meet these needs, MongoDB is used as the primary database due to its scalability, high performance, and compatibility with JavaScript-based backend environments.

MongoDB

MongoDB is a popular open-source NoSQL database that stores data in flexible, JSON-like documents called BSON (Binary JSON). Unlike traditional relational databases, it does not require a fixed schema, making it ideal for modern applications with dynamic and evolving data structures**.**

- **Role in Streamify:**

  In Streamify, MongoDB is used to store and manage user data (e.g., usernames, passwords), chat histories, user session information, and video call metadata. Its document-based model aligns well with JavaScript and makes integration with Node.js seamless.

- **Key Features in Streamify:**

- **Schema Flexibility**: Allows for dynamic fields, enabling quick iteration during development.
- **Document-Oriented Storage**: Stores chat messages, user details, and call logs as individual documents for efficient access.
- **High Performance**: Supports fast read/write operations, essential for real-time communication.
- **Scalability**: Easily scales horizontally using sharding, making it suitable for growing user bases.
- **Seamless Integration**: Works directly with Express.js and Node.js using Mongoose (ODM), simplifying data operations and validations.

CHAPTER 5: SYSTEM DESIGN

5.1 USER INTERFACE DESIGN

The user interface of the Social Media Video Chat Application is designed using React.js to be modular, intuitive, and responsive. It ensures that users can engage in seamless trading experiences while viewing real-time market activity. The core UI components include:

1. Intuitive Layout

- **Navigation Bar**: Include icons for home, chat, notifications, profile, and settings.
- **Sidebar**: Display online friends and group chats for quick access.
- **Main Chat Area**: Show active conversations and video call interfaces.
- **Video Call Window**: Implement a floating window with controls for mute, video toggle, and end call.

2. Responsive Design

- Use CSS frameworks like Tailwind CSS or Material-UI to ensure the app is mobile-friendly and adapts to various screen sizes.
- Ensure that video streams adjust dynamically to fit different device orientations and resolutions.

3. Real-Time Interaction

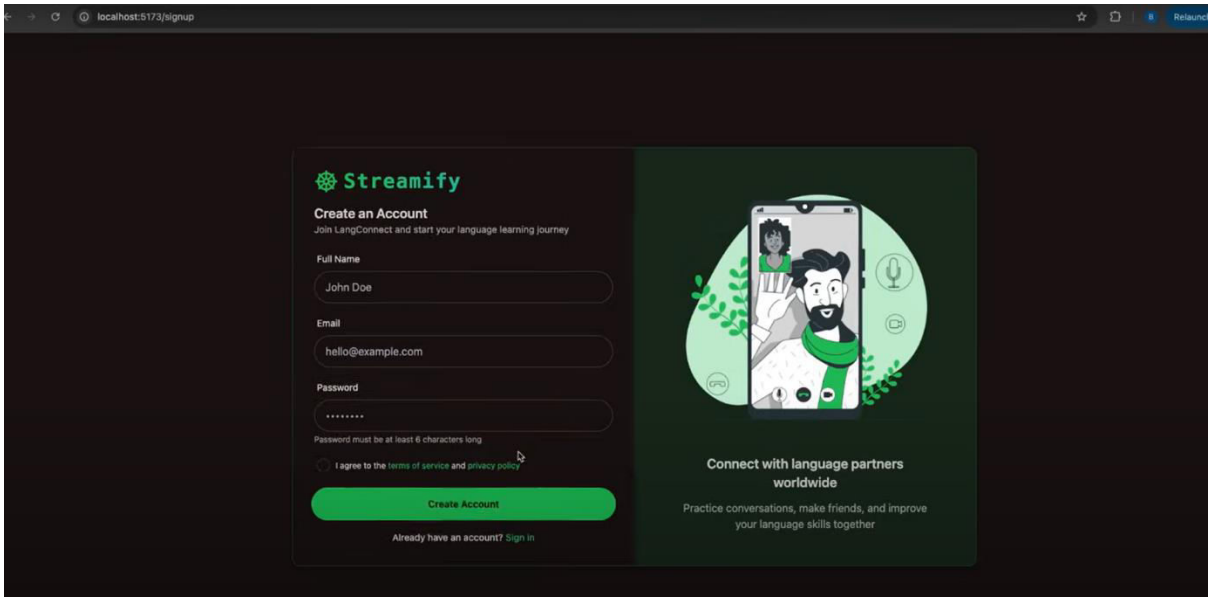
- Integrate Socket.IO for real-time messaging and notifications.
- Implement WebRTC for peer-to-peer video communication, with features like screen sharing and virtual backgrounds.

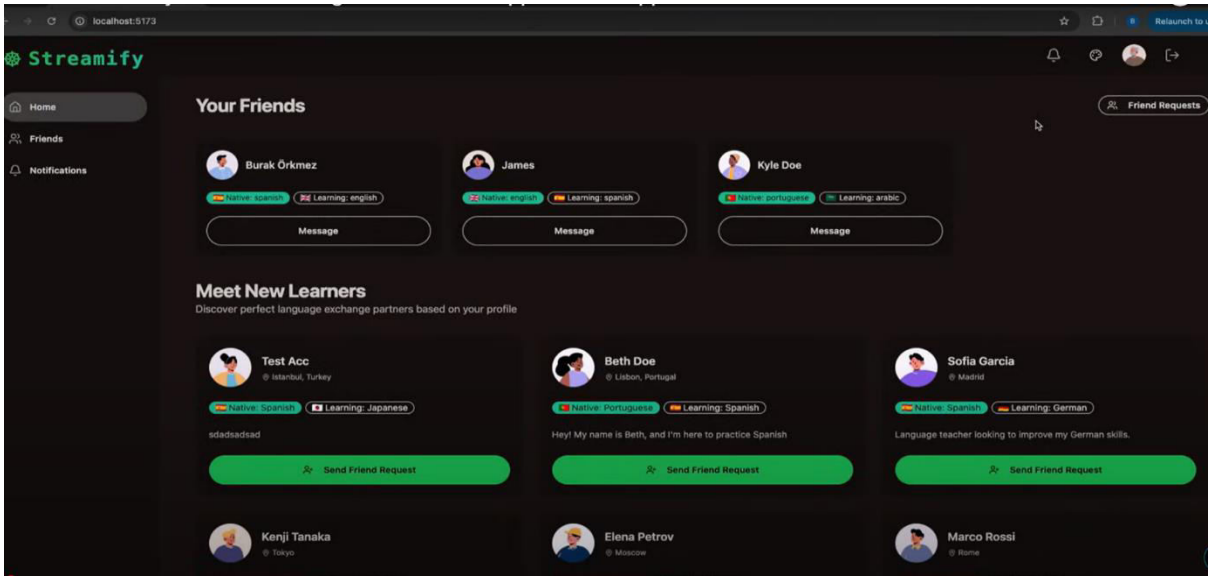
4.Accessibility

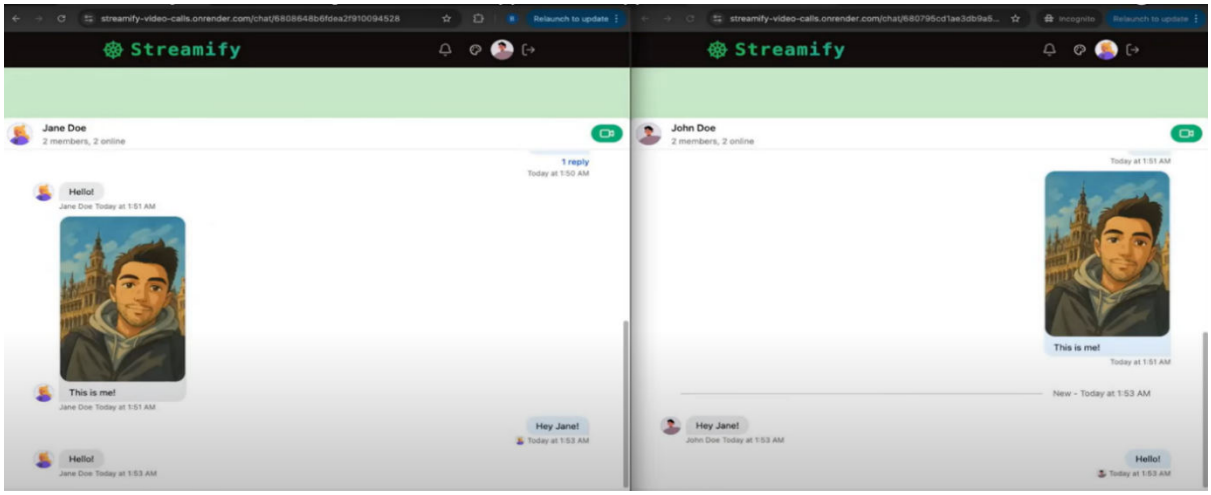
- Ensure color contrast meets accessibility standards.
- Provide keyboard shortcuts for navigation.
- Offer screen reader support and alternative text for media.

5. Real-Time Interaction

- Integrate Socket.IO for real-time messaging and notifications.
- Implement WebRTC for peer-to-peer video communication, with features like screen sharing and virtual backgrounds







5.2 VIDEO CHAT DESIGN

- **MongoDB**: NoSQL database for storing user data, messages, and call logs.
- **Express.js**: Backend framework for handling HTTP requests and API routes.
- **React.js**: Frontend library for building the user interface.
- **Node.js**: JavaScript runtime for the backend server.
- **WebRTC**: Enables peer-to-peer audio, video, and data sharing.
- **Socket.IO**: Facilitates real-time, bidirectional communication between clients and the server.
- **simple-peer**: Simplifies WebRTC peer-to-peer connections

Frontend (React.js):

- Utilizes simple-peer for establishing WebRTC connections.
- Integrates with Socket.IO for signaling and real-time messaging.
- Manages application state using Redux or Context API.

Backend (Node.js + Express.js):

- Serves as the signaling server using Socket.IO.
- Handles user authentication and authorization.
- Manages API routes for user data and message storage.

Database (MongoDB):

- Stores user profiles, messages, and call histories.
- Supports real-time data updates and retrieval

Implementation Steps

1. **Set Up the Backend**:

    - Initialize a Node.js project and install dependencies (express, socket.io, mongoose).
    - Configure MongoDB for data storage.
    - Set up Socket.IO for real-time communication

2. **Develop the Frontend**:

- Create a React application using create-react-app.
- Install necessary packages (simple-peer, socket.io-client, redux).
- Build components for login, chat interface, and video call

3. **Implement Signaling with Socket.IO**:

- Exchange ICE candidates and session descriptions between peers.
- Manage signaling events for call initiation, acceptance, and termination.

4. **Integrate WebRTC**:

- Use simple-peer to establish peer-to-peer connections.
- Implement ICE (Interactive Connectivity Establishment) for network traversal.
- Handle media streams for audio, video, and screen sharing

Source Code:-

```
import { VideoIcon } from "lucide-react";



function CallButton({ handleVideoCall }) {

  return (

    <div className="p-3 border-b flex items-center justify-end max-w-7xl mx-auto w-
full absolute top-0">

      <button onClick={handleVideoCall} className="btn btn-success btn-sm text-
white">

        <VideoIcon className="size-6" />

      </button>

    </div>

  );

}



export default CallButton;



import { LoaderIcon } from "lucide-react";



function ChatLoader() {

  return (

    <div className="h-screen flex flex-col items-center justify-center p-4">

      <LoaderIcon className="animate-spin size-10 text-primary" />

      <p className="mt-4 text-center text-lg font-mono">Connecting to chat...</p>

    </div>

  );

}



export default ChatLoader;

import { Link, useLocation } from "react-router";
```

```
import useAuthUser from "../hooks/useAuthUser";

import { BellIcon, LogOutIcon, ShipWheelIcon } from "lucide-react";

import ThemeSelector from "./ThemeSelector";

import useLogout from "../hooks/useLogout";


const Navbar = () => {

  const { authUser } = useAuthUser();

  const location = useLocation();

  const isChatPage = location.pathname?.startsWith("/chat");


  // const queryClient = useQueryClient();

  // const { mutate: logoutMutation } = useMutation({

  //   mutationFn: logout,

  //   onSuccess: () => queryClient.invalidateQueries({ queryKey: ["authUser"] }),

  // });


  const { logoutMutation } = useLogout();


  return (

    <nav className="bg-base-200 border-b border-base-300 sticky top-0 z-30 h-16
flex items-center">

      <div className="container mx-auto px-4 sm:px-6 lg:px-8">

        <div className="flex items-center justify-end w-full">

          {/* LOGO - ONLY IN THE CHAT PAGE */}

          {isChatPage && (

            <div className="pl-5">

              <Link to="/" className="flex items-center gap-2.5">

                <ShipWheelIcon className="size-9 text-primary" />

                <span className="text-3xl font-bold font-mono bg-clip-text text-
transparent bg-gradient-to-r from-primary to-secondary  tracking-wider">
```

```
            Streamify
          </span>
        </Link>
      </div>
    )}

    <div className="flex items-center gap-3 sm:gap-4 ml-auto">
      <Link to={"/notifications"}>
        <button className="btn btn-ghost btn-circle">
          <BellIcon className="h-6 w-6 text-base-content opacity-70" />
        </button>
      </Link>
    </div>

    {/* TODO */}
    <ThemeSelector />

    <div className="avatar">
      <div className="w-9 rounded-full">
        <img src={authUser?.profilePic} alt="User Avatar" rel="noreferrer" />
      </div>
    </div>

    {/* Logout button */}
    <button className="btn btn-ghost btn-circle" onClick={logoutMutation}>
      <LogOutIcon className="h-6 w-6 text-base-content opacity-70" />
    </button>
  </div>
</div>
```

```
    </nav>
  );
};
export default Navbar;
import { LoaderIcon } from "lucide-react";
import { useThemeStore } from "../store/useThemeStore";

const PageLoader = () => {
  const { theme } = useThemeStore();
  return (
    <div className="min-h-screen flex items-center justify-center" data-
theme={theme}>
      <LoaderIcon className="animate-spin size-10 text-primary" />
    </div>
  );
};
export default PageLoader;
import { Link, useLocation } from "react-router";
import useAuthUser from "../hooks/useAuthUser";
import { BellIcon, HomeIcon, ShipWheelIcon, UsersIcon } from "lucide-react";

const Sidebar = () => {
  const { authUser } = useAuthUser();
  const location = useLocation();
  const currentPath = location.pathname;

  return (
    <aside className="w-64 bg-base-200 border-r border-base-300 hidden lg:flex
flex-col h-screen sticky top-0">
      <div className="p-5 border-b border-base-300">
```

```
        <Link to="/" className="flex items-center gap-2.5">

          <ShipWheelIcon className="size-9 text-primary" />

          <span className="text-3xl font-bold font-mono bg-clip-text text-
transparent bg-gradient-to-r from-primary to-secondary  tracking-wider">

            Streamify

          </span>

        </Link>

      </div>



      <nav className="flex-1 p-4 space-y-1">

        <Link

          to="/"

          className={`btn btn-ghost justify-start w-full gap-3 px-3 normal-case ${

            currentPath === "/" ? "btn-active" : ""

          }`}

        >

          <HomeIcon className="size-5 text-base-content opacity-70" />

          <span>Home</span>

        </Link>



        <Link

          to="/friends"

          className={`btn btn-ghost justify-start w-full gap-3 px-3 normal-case ${

            currentPath === "/friends" ? "btn-active" : ""

          }`}

        >

          <UsersIcon className="size-5 text-base-content opacity-70" />

          <span>Friends</span>

        </Link>
```

```
    <Link
      to="/notifications"
      className={`btn btn-ghost justify-start w-full gap-3 px-3 normal-case ${
        currentPath === "/notifications" ? "btn-active" : ""
      }`}
    >
      <BellIcon className="size-5 text-base-content opacity-70" />
      <span>Notifications</span>
    </Link>
  </nav>

  {/* USER PROFILE SECTION */}
  <div className="p-4 border-t border-base-300 mt-auto">
    <div className="flex items-center gap-3">
      <div className="avatar">
        <div className="w-10 rounded-full">
          <img src={authUser?.profilePic} alt="User Avatar" />
        </div>
      </div>
      <div className="flex-1">
        <p className="font-semibold text-sm">{authUser?.fullName}</p>
        <p className="text-xs text-success flex items-center gap-1">
          <span className="size-2 rounded-full bg-success inline-block" />
          Online
        </p>
      </div>
    </div>
  </div>
</aside>
```

```
  );
};
export default Sidebar;


import User from "../models/User.js";
import FriendRequest from "../models/FriendRequest.js";

export async function getRecommendedUsers(req, res) {
  try {
    const currentUserId = req.user.id;
    const currentUser = req.user;

    const recommendedUsers = await User.find({
      $and: [
        { _id: { $ne: currentUserId } }, //exclude current user
        { _id: { $nin: currentUser.friends } }, // exclude current user's friends
        { isOnboarded: true },
      ],
    });
    res.status(200).json(recommendedUsers);
  } catch (error) {
    console.error("Error in getRecommendedUsers controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
}

export async function getMyFriends(req, res) {
  try {
    const user = await User.findById(req.user.id)
```

```
      .select("friends")
      .populate("friends", "fullName profilePic nativeLanguage learningLanguage");

    res.status(200).json(user.friends);
  } catch (error) {
    console.error("Error in getMyFriends controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
}

export async function sendFriendRequest(req, res) {
  try {
    const myId = req.user.id;
    const { id: recipientId } = req.params;

    // prevent sending req to yourself
    if (myId === recipientId) {
      return res.status(400).json({ message: "You can't send friend request to
yourself" });
    }

    const recipient = await User.findById(recipientId);
    if (!recipient) {
      return res.status(404).json({ message: "Recipient not found" });
    }

    // check if user is already friends
    if (recipient.friends.includes(myId)) {
      return res.status(400).json({ message: "You are already friends with this
user" });
```

```
    }

    // check if a req already exists
    const existingRequest = await FriendRequest.findOne({
      $or: [
        { sender: myId, recipient: recipientId },
        { sender: recipientId, recipient: myId },
      ],
    });

    if (existingRequest) {
      return res
        .status(400)
        .json({ message: "A friend request already exists between you and this
user" });
    }

    const friendRequest = await FriendRequest.create({
      sender: myId,
      recipient: recipientId,
    });

    res.status(201).json(friendRequest);
  } catch (error) {
    console.error("Error in sendFriendRequest controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
}

export async function acceptFriendRequest(req, res) {
```

```
  try {
    const { id: requestId } = req.params;

    const friendRequest = await FriendRequest.findById(requestId);

    if (!friendRequest) {
      return res.status(404).json({ message: "Friend request not found" });
    }

    // Verify the current user is the recipient
    if (friendRequest.recipient.toString() !== req.user.id) {
      return res.status(403).json({ message: "You are not authorized to accept this
request" });
    }

    friendRequest.status = "accepted";
    await friendRequest.save();

    // add each user to the other's friends array
    // $addToSet: adds elements to an array only if they do not already exist.
    await User.findByIdAndUpdate(friendRequest.sender, {
      $addToSet: { friends: friendRequest.recipient },
    });

    await User.findByIdAndUpdate(friendRequest.recipient, {
      $addToSet: { friends: friendRequest.sender },
    });

    res.status(200).json({ message: "Friend request accepted" });
  } catch (error) {
```

```
    console.log("Error in acceptFriendRequest controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
}

export async function getFriendRequests(req, res) {
  try {
    const incomingReqs = await FriendRequest.find({
      recipient: req.user.id,
      status: "pending",
    }).populate("sender", "fullName profilePic nativeLanguage learningLanguage");

    const acceptedReqs = await FriendRequest.find({
      sender: req.user.id,
      status: "accepted",
    }).populate("recipient", "fullName profilePic");

    res.status(200).json({ incomingReqs, acceptedReqs });
  } catch (error) {
    console.log("Error in getPendingFriendRequests controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
}

export async function getOutgoingFriendReqs(req, res) {
  try {
    const outgoingRequests = await FriendRequest.find({
      sender: req.user.id,
      status: "pending",
```

```
    }).populate("recipient", "fullName profilePic nativeLanguage
learningLanguage");



    res.status(200).json(outgoingRequests);

  } catch (error) {

    console.log("Error in getOutgoingFriendReqs controller", error.message);

    res.status(500).json({ message: "Internal Server Error" });

  }

}

import { upsertStreamUser } from "../lib/stream.js";

import User from "../models/User.js";

import jwt from "jsonwebtoken";



export async function signup(req, res) {

  const { email, password, fullName } = req.body;



  try {

    if (!email || !password || !fullName) {

      return res.status(400).json({ message: "All fields are required" });

    }



    if (password.length < 6) {

      return res.status(400).json({ message: "Password must be at least 6
characters" });

    }



    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;



    if (!emailRegex.test(email)) {

      return res.status(400).json({ message: "Invalid email format" });
```

```
    }

    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return res.status(400).json({ message: "Email already exists, please use a
diffrent one" });
    }

    const idx = Math.floor(Math.random() * 100) + 1; // generate a num between 1-
100
    const randomAvatar = `https://avatar.iran.liara.run/public/${idx}.png`;

    const newUser = await User.create({
      email,
      fullName,
      password,
      profilePic: randomAvatar,
    });

    try {
      await upsertStreamUser({
        id: newUser._id.toString(),
        name: newUser.fullName,
        image: newUser.profilePic || "",
      });
      console.log(`Stream user created for ${newUser.fullName}`);
    } catch (error) {
      console.log("Error creating Stream user:", error);
    }
```

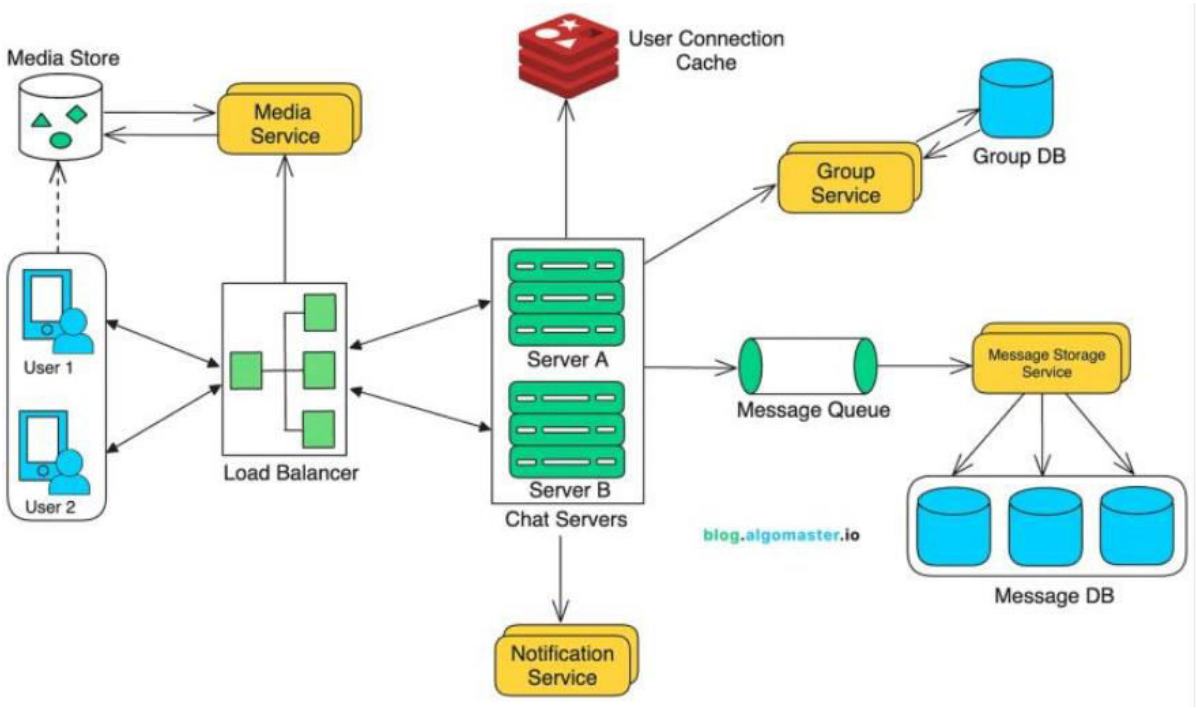
```
    const token = jwt.sign({ userId: newUser._id }, process.env.JWT_SECRET_KEY, {
      expiresIn: "7d",
    });

    res.cookie("jwt", token, {
      maxAge: 7 * 24 * 60 * 60 * 1000,
      httpOnly: true, // prevent XSS attacks,
      sameSite: "strict", // prevent CSRF attacks
      secure: process.env.NODE_ENV === "production",
    });

    res.status(201).json({ success: true, user: newUser });
  } catch (error) {
    console.log("Error in signup controller", error);
    res.status(500).json({ message: "Internal Server Error" });
  }
}

export async function login(req, res) {
  try {
    const { email, password } = req.body;

    if (!email || !password) {
      return res.status(400).json({ message: "All fields are required" });
    }

    const user = await User.findOne({ email });
    if (!user) return res.status(401).json({ message: "Invalid email or password"
});
```

```
    const isPasswordCorrect = await user.matchPassword(password);

    if (!isPasswordCorrect) return res.status(401).json({ message: "Invalid email
or password" });



    const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET_KEY, {

      expiresIn: "7d",

    });



    res.cookie("jwt", token, {

      maxAge: 7 * 24 * 60 * 60 * 1000,

      httpOnly: true, // prevent XSS attacks,

      sameSite: "strict", // prevent CSRF attacks

      secure: process.env.NODE_ENV === "production",

    });



    res.status(200).json({ success: true, user });

  } catch (error) {

    console.log("Error in login controller", error.message);

    res.status(500).json({ message: "Internal Server Error" });

  }

}



export function logout(req, res) {

  res.clearCookie("jwt");

  res.status(200).json({ success: true, message: "Logout successful" });

}



export async function onboard(req, res) {

  try {

    const userId = req.user._id;
```

```
const { fullName, bio, nativeLanguage, learningLanguage, location } = req.body;

if (!fullName || !bio || !nativeLanguage || !learningLanguage || !location) {
  return res.status(400).json({
    message: "All fields are required",
    missingFields: [
      !fullName && "fullName",
      !bio && "bio",
      !nativeLanguage && "nativeLanguage",
      !learningLanguage && "learningLanguage",
      !location && "location",
    ].filter(Boolean),
  });
}

const updatedUser = await User.findByIdAndUpdate(
  userId,
  {
    ...req.body,
    isOnboarded: true,
  },
  { new: true }
);

if (!updatedUser) return res.status(404).json({ message: "User not found" });

try {
  await upsertStreamUser({
```

```
        id: updatedUser._id.toString(),

        name: updatedUser.fullName,

        image: updatedUser.profilePic || "",

      });

      console.log(`Stream user updated after onboarding for
${updatedUser.fullName}`);

    } catch (streamError) {

      console.log("Error updating Stream user during onboarding:",
streamError.message);

    }



    res.status(200).json({ success: true, user: updatedUser });

  } catch (error) {

    console.error("Onboarding error:", error);

    res.status(500).json({ message: "Internal Server Error" });

  }
```

5.3ARCHITECTURE DIAGRAM



1. Client Applications

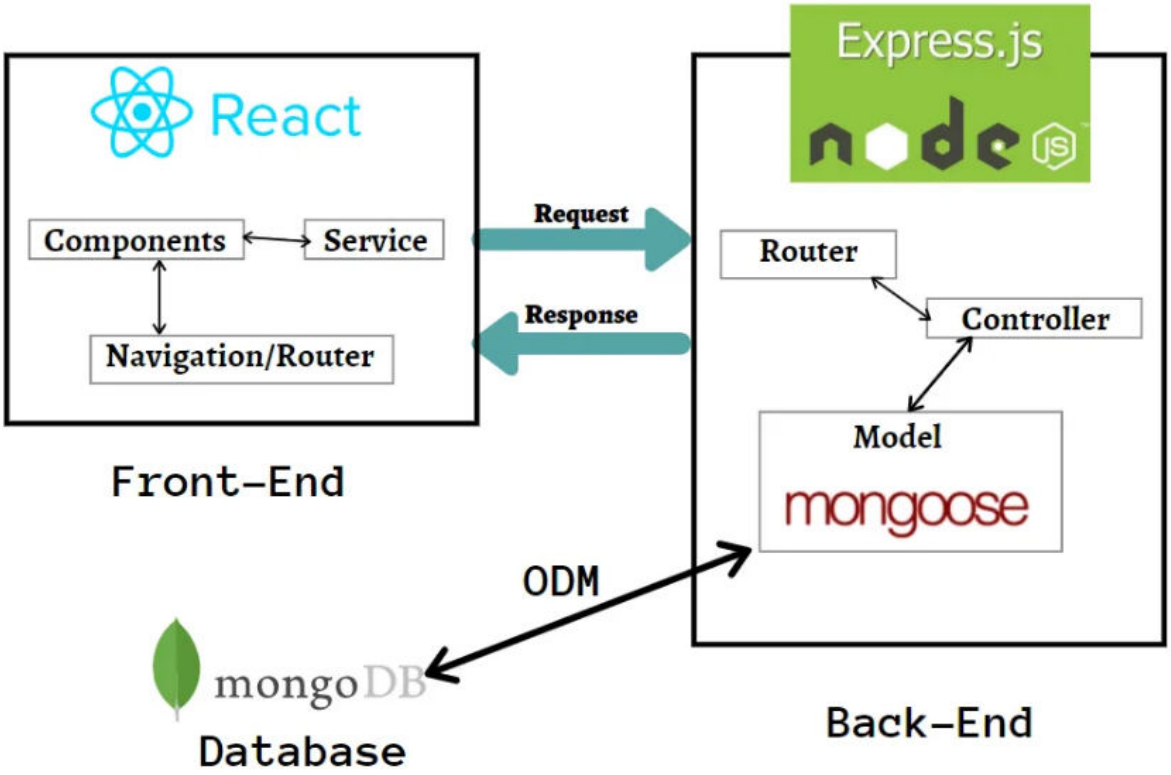
- **Platforms**: Web (React),
- **Responsibilities**:
    - User authentication and profile management
    - Initiating and receiving video/audio calls
    - Handling media streams (camera/microphone)
    - Rendering UI elements (chat interface, notifications)

2. Signaling Server

- **Purpose**: Facilitates the initial handshake between clients to establish peer-to-peer connections.
- **Common Protocols**:
    - WebSockets

- HTTP(S)
- SIP over WebSockets (e.g., JsSIP)

- **Role**:
  - Exchanges metadata (offer/answer, ICE candidates) between clients
  - Coordinates session initiation and termination



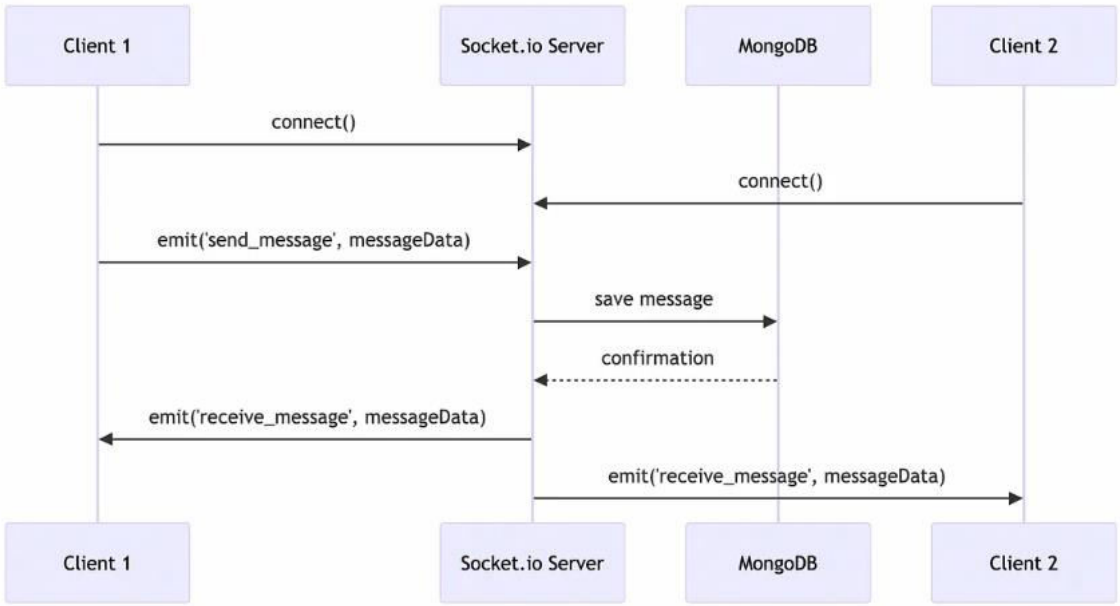
3. Media Server (Optional)

- **Selective Forwarding Unit (SFU)**:
  - Receives media streams from clients and forwards them to other participants.
  - Reduces client-side bandwidth usage and simplifies media routing.
  - Suitable for group calls with many participants.
- **Mesh Architecture**:
  - Each client sends media directly to every other client.

- Ideal for small group calls (2-5 participants) due to simplicity but less scalable.

4. Backend Infrastructure

- **User Management**:
  - Authentication (OAuth, JWT)
  - Profile storage (e.g., Firebase, MongoDB)
- **Real-Time Messaging**:
  - Chat messages, notifications
  - Services like Firebase Realtime Database or Ably for real-time data sync

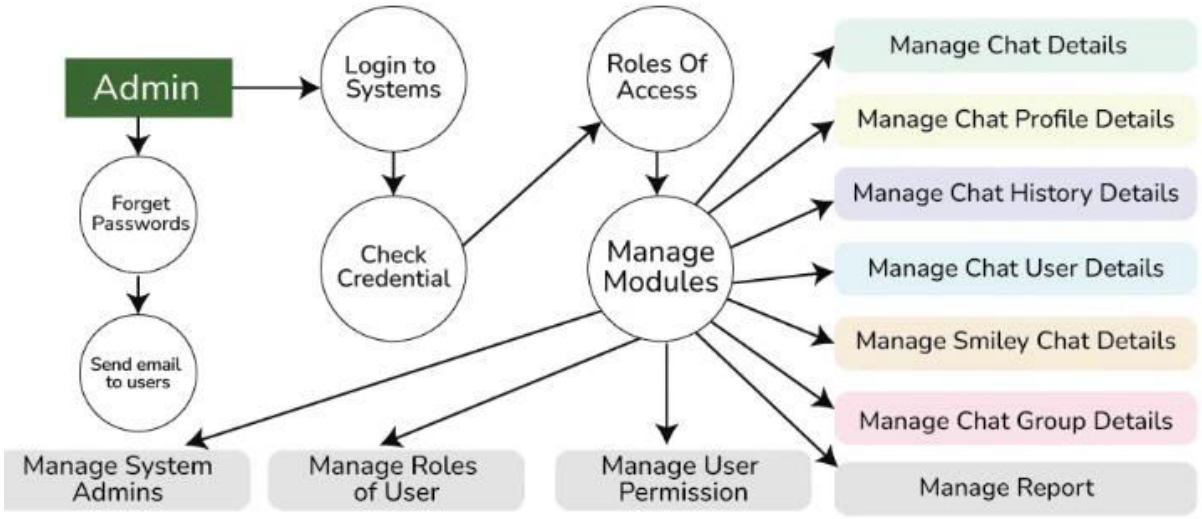


5. WebRTC (Web Real-Time Communication) and Socket.io

- **Functionality**:
  - Enables peer-to-peer audio/video communication directly between clients.

- Utilizes APIs like getUserMedia, RTCPeerConnection, and RTCDataChannel.

5.4 DATA FLOW DIAGRAM (DFD)

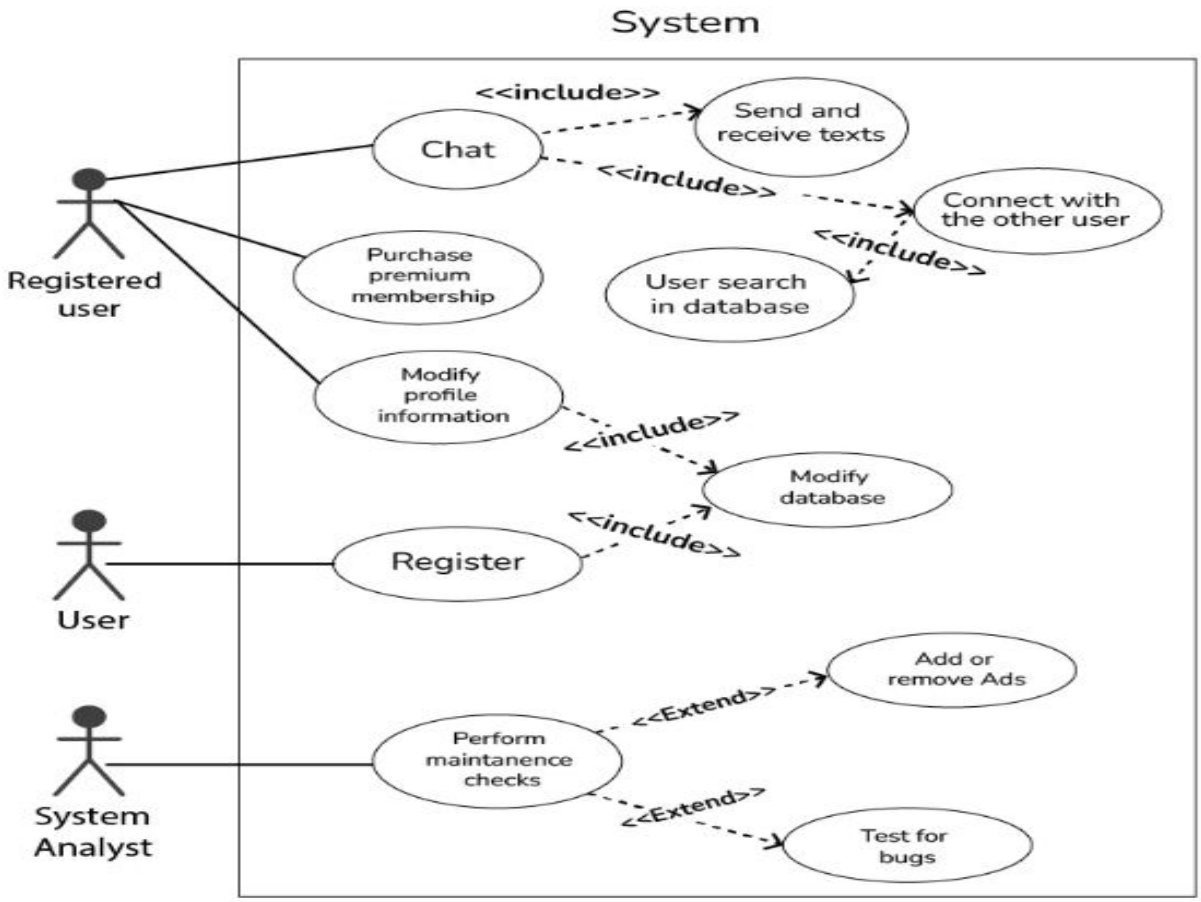


This diagram illustrates how data, such as book information, user details, and transaction records, moves between various components of the Online Chat Application.

- **Processes**, represented by circles or ovals.
- **Data stores**, depicted by rectangles, represent where information is stored.
- **Data flows**, indicated by arrows, showcase how data moves between processes.

The DFD provides a concise yet comprehensive overview of the Online Chat Application's data flow and interactions, aiding in the analysis, design, and communication of the system’s functional aspects.

5.5 USE CASE DIAGRAM



User Actions:

Sign Up: User creates a new account by providing necessary information like username, password, etc.

Sign In: User logs into the application using their credentials.

Send Message: User composes and sends a message to another user or within a chatroom.

Receive Message: User receives messages sent by other users in real-time.

Create Chatroom: User creates a new chatroom, becoming its administrator.

Join Chatroom: User joins an existing chatroom to participate in group conversations.

Leave Chatroom: User exits a chatroom, ceasing participation in its conversations.

View Chatroom Participants: User views the list of participants in a chatroom.

System Functions:

Message Management: The system handles sending and receiving messages between users and within chatrooms.

User Management: The system manages user accounts, including sign-up, sign-in, and user authentication.

Chatroom Management: The system handles creating, joining, leaving, and managing chatrooms.

Participant Management: The system manages the list of participants in each chatroom.

Real-time Communication: The system ensures real-time delivery of messages between users.

This use case diagram illustrates the primary interactions between users and the online chat application. It highlights the key functionalities that users can perform and the corresponding system functions that facilitate those actions

CHAPTER 6: FUNCTIONAL IMPLEMENTATION

1. **Introduction**

Developed a real-time communication platform combining video calling and social chat functionalities using the MERN stack (MongoDB, Express.js, React.js, Node.js) and WebRTC technology.

2. Technologies Used

- **Frontend:** React.js, Tailwind CSS
- **Backend:** Node.js, Express.js
- **Real-Time Communication:** WebRTC, Socket.IO
- **Database:** MongoDB
- **Libraries:** simple-peer, PeerJS

3. System Architecture

- **Frontend:** React components for UI, managing state with Context API or Redux.
- **Backend:** Express server handling REST APIs and WebSocket connections via Socket.IO.
- **Real-Time Communication:** WebRTC for peer-to-peer video/audio calls; signaling handled through Socket.IO.

Implementation Steps

1. Frontend Development

Set Up React App:

```
bash
Copy code
npx create-react-app client
cd client
npm install axios socket.io-client react-router-dom
tailwindcss
```

Configure Tailwind CSS:

```
bash
Copy code
npx tailwindcss init

Edit tailwind.config.js:

js
Copy code
module.exports = {
  content: ['./src/**/*.{js,jsx,ts,tsx}'],
  theme: { extend: {} },
  plugins: [],
};

Import Tailwind into index.css:

css
Copy code
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Component Development:

- **Login Component:** Handles user authentication.
- **Chat Interface:** Displays messages and allows sending/receiving.
- **Video Call Component:** Manages video streams and call controls.

2. Backend Development

Set Up Express Server:

```
bash
Copy code
npm init -y
npm install express mongoose socket.io jsonwebtoken bcryptjs
```

API Development:

- **User Authentication:** Implement JWT-based login and registration.
- **Chat Routes:** Create routes for sending and receiving messages.

Socket.IO Integration:

```
js
Copy code
```

```
const io = require('socket.io')(server, { cors: { origin: '*'
} });

io.on('connection', (socket) => {
  console.log('New client connected');
  socket.on('sendMessage', (message) => {
    io.emit('message', message);
  });
  socket.on('disconnect', () => {
    console.log('Client disconnected');
  });
});
```

3. WebRTC Integration

Install PeerJS:

```
bash
Copy code
npm install peerjs
```

Implement Signaling:

```
js
Copy code
const peer = new Peer();
peer.on('open', (id) => {
  console.log('My peer ID is: ' + id);
});
```

Handle Incoming Calls:

```
js
Copy code
peer.on('call', (call) => {
  call.answer(localStream);
  call.on('stream', (remoteStream) => {
    remoteVideo.srcObject = remoteStream;
  });
});
```

Making Outgoing Calls:

```
js
Copy code
const call = peer.call(remotePeerId, localStream);
call.on('stream', (remoteStream) => {
  remoteVideo.srcObject = remoteStream;});
```

Deployment

Frontend:

- Deploy onVercel or Netlify.

Backend:

- Deploy on Heroku or Render.

Environment Variables:

- Set MONGO_URI_DEV and JWT_SECRET in the production environment.

Challenges Faced

- **Cross-Browser Compatibility:** Ensuring WebRTC functions across different browsers.
- **Network Issues:** Handling NAT traversal and firewall restrictions.
- **Scalability:** Managing multiple peer connections efficiently.

4. Key Features

- **User Authentication:** Secure login and registration.
- **Real-Time Chat:** Instant messaging with message persistence.
- **Video Calling:** One-on-one video calls using WebRTC.
- **Social Features:** Friend requests, group chats, notifications.

5. Implementation Steps

1. **Frontend Development:**
    - Set up React app with Tailwind CSS.
    - Create components for login, chat interface, and video call.
    - Implement state management for user sessions and messages.
2. **Backend Development:**

- Set up Express server with routes for user authentication and chat functionalities.
- Implement WebSocket server using Socket.IO for real-time communication.

3. **WebRTC Integration:**

- Use simple-peer or PeerJS for establishing peer-to-peer connections.
- Implement signaling process to exchange connection information between peers.

6. Deployment

- **Frontend:** Deployed on Vercel or Netlify.
- **Backend:** Deployed on Heroku or Render.
- **Environment Variables:** Configured for production environments.

7. Challenges Faced

- **Cross-Browser Compatibility:** Ensuring WebRTC functions across different browsers.
- **Network Issues:** Handling NAT traversal and firewall restrictions.
- **Scalability:** Managing multiple peer connections efficiently.

CHAPTER 7: TESTING

7.1 INTRODUCTION TO TESTING

Web application testing involves evaluating a website or web application's functionality, usability, security, and performance to ensure it meets specified requirements and provides a seamless user experience.

- **Ensures Quality:** Identifies defects early, leading to higher-quality software.
- **Reduces Costs:** Catches issues before deployment, minimizing costly post-release fixes.
- **Enhances User Satisfaction:** Delivers a reliable product, improving user experience and trust.
- **Compliance:** Helps meet regulatory and contractual obligations

7.2 TEST CASES

Unit Testing

Frontend (React.js):

- **Jest:** A JavaScript testing framework that works out-of-the-box with React. It includes an assertion library (`expect`), mocking utilities, snapshot testing, and built-in code coverage.
- **React Testing Library:** Helps in testing UI components in a user-centric way. It encourages good testing practices by focusing on the behavior of the components rather than their implementation details

Backend (Node.js/Express):

- **Mocha:** A flexible JavaScript test framework for Node.js. It allows organizing tests into describe/it blocks with a simple API.
- **Chai:** An assertion library for Node and browser. It pairs well with Mocha to check values in tests

Integration Testing

- **Supertest:** A SuperAgent driven library that allows you to test HTTP servers in Node.js. It's commonly used to test RESTful APIs by making requests and asserting responses.
- **Socket.IO Testing:** Tools like `socket.io-client` can be used to simulate client-side socket connections in tests, ensuring that real-time communication features work as expected.

End-to-End (E2E) Testing

- **Playwright:** A Node.js library to automate Chromium, Firefox, and WebKit with a single API. It's used for E2E testing to simulate real user interactions with the application.

CHAPTER 8: CONCLUSION

The integration of the MERN stack with WebRTC has enabled the development of a robust, real-time communication platform that offers seamless video calling and instant messaging capabilities.

Key Achievements

- **Real-Time Communication**: Utilizing Socket.IO for instant messaging and WebRTC for peer-to-peer video calls ensures low-latency interactions.
- **Scalability**: MongoDB's NoSQL architecture supports flexible data models, accommodating the dynamic nature of chat and video data.
- **User Experience**: React.js facilitates the creation of a dynamic and responsive user interface, enhancing user engagement.
- **Security**: Implementing JWT-based authentication and secure WebRTC signaling ensures user data protection.

CHAPTER 9: FUTURE ENHANCEMENTS

- **Group Video Calling:** Implement multi-peer WebRTC connections.
- **End-to-End Encryption:** Enhance security for messages and calls.
- **Push Notifications:** Integrate push notifications for messages and calls
- **Screen Sharing**: Allow users to share their screens during calls, facilitating presentations and collaborative work.
- **Multi-Device Synchronization**: Ensure seamless user experience across different devices by synchronizing chats and calls.
- **Scheduling and Calendar Integration**: Allow users to schedule calls and events, integrating with their calendars.

APPENDIX

Appendix A: Code Snippets

- JWT Authentication Code used for user login and secure session handling.
- Socket.IO Integration Code to enable real-time messaging.
- WebRTC Peer Connection Code using simple-peer for video call setup.

Appendix B: Database Schema

- Collection: Fields like fullName, email, password, profilePic, friends.
- Messages Collection: Contains senderId, receiverId, message, timestamp.
- Friend Requests Collection: Stores senderId, recipientId, status.

Appendix C: UI Screenshots

- Login/Register Page
- Chat Interface
- Video Call Window
- Notification Panel and Sidebar

Appendix E: Deployment Steps

- Frontend deployed on Vercel.
- Backend deployed on Heroku.
- MongoDB hosted on MongoDB Atlas.
- Environment variables like MONGO_URI and JWT_SECRET configured securely.

REFERENCES

- Youtube (www.youtube.com)
- Mongodb (www.mongodb.com)
- WebRTC (https://webrtc.org/)
- React.js Official Docs (https://react.dev/)
- Socket.io (https://socket.io/docs/v4/)
- TanstackQuery(https://tanstack.com/query/latest/docs/framework/react)
- Online courses on Udemy, Coursera related to Web Development

Research Papers and Articles:

- Privy: A MERN Chat App with Video Conferencing & Screen Sharing (**Authors**: M Dolie Ciri, Vishnu Vardhan, M Surya Teja)
- Scalable and Secure Real-Time Chat Application Development Using MERN Stack and Socket.io(**Authors**: Abhijith Pandiri, Sai Shreyas Venishetty, Akhil Reddy Modugu, K Venkatesh Sharma).