1. What are WebSockets?

WebSockets is a **communication protocol** that provides a persistent, full-duplex (two-way) connection between a client (e.g., browser, mobile app) and a server. Unlike traditional HTTP, which is **request–response** based, WebSockets allow both the server and client to send data to each other **at any time**.

- **Traditional HTTP**: Client must always initiate a request, server responds, connection is closed.

- **WebSocket**: Connection is established once (the handshake) and then kept open, enabling continuous exchange of messages.

This makes WebSockets perfect for **real-time applications** such as chat apps, video calls, collaborative whiteboards, live dashboards, gaming platforms, etc.

2. How WebSockets Work Under the Hood

1. **Handshake**:
    - Starts as an HTTP(S) request.
    - Client sends an “Upgrade” header to switch the protocol to WebSocket.
    - Server responds with a confirmation, and the connection is upgraded.

2. **Persistent Connection**:
    - Once established, the connection stays open.
    - Messages can now flow both ways without needing repeated requests.

3. **Message Frames**:
    - Data is exchanged in small “frames” (text or binary).
    - This makes it lightweight compared to polling or long-polling.

3. WebSocket Servers in Node.js

In Node.js, WebSocket servers can be created using packages such as:

- **`ws`**: A widely used, minimal WebSocket library for Node.js.

- **Socket.IO**: Built on top of WebSockets (and fallbacks), provides additional features like rooms, namespaces, and auto-reconnect.

Example (with ws):

```
import { WebSocketServer } from "ws";

// Create a WebSocket server on port 8080
const wss = new WebSocketServer({ port: 8080 });

wss.on("connection", (ws) => {
  console.log("New client connected!");

  // Listen for messages from client
  ws.on("message", (message) => {
    console.log(`Received: ${message}`);

    // Echo back to client
    ws.send(`Server says: ${message}`);
  });

  // Handle disconnection
  ws.on("close", () => {
    console.log("Client disconnected");
  });
});
```

4. Types of WebSocket Connections / Use Cases

- **One-to-One Communication**
  - Example: Direct private chat between two users.

- **One-to-Many (Broadcast)**
  - Example: Server broadcasts a message (e.g., “User joined the group”) to all connected clients.

- **Many-to-Many (Groups/Rooms)**
  - Example: Group chats or collaborative video rooms where multiple participants exchange messages and streams simultaneously.

5. How This Applies to *Streamify*

In Streamify, WebSockets power the **real-time interactivity**:

- **Chat Messaging**:
    - When a user sends a message, it travels via a WebSocket connection to the server.
    - Server routes it instantly to the target recipient(s), ensuring messages appear in real time.

- **Video/Audio Calling**:
    - Before a call starts, WebSockets are used to exchange signaling data (like offer/answer/ICE candidates in WebRTC).
    - This sets up a peer-to-peer media stream between participants.

- **Presence and Typing Indicators**:
    - Users can see who is online or when someone is typing, thanks to real-time updates sent over WebSockets.

- **Scalability with Groups**:
    - Public and private groups map to “rooms” in the WebSocket server.
    - Messages or events are delivered only to members of those rooms.

6. Advantages of WebSockets

- **Low Latency**: Faster than polling or long-polling.

- **Bi-Directional**: Server can push updates instantly.

- **Reduced Overhead**: No repeated HTTP headers with every message.

- **Real-Time**: Perfect for collaborative and interactive apps.

Code:

backend > JS server.js > ...

```
34
35  //routes
36  app.get('/',(req,res)=>{
37      res.send('hello from server !');
38  })
39
40  app.use("/api/v1/auth",AuthRoutes);
41  app.use("/api/v1/user",UserRoutes);
42  app.use("/api/v1/chat", chatRoutes);
43  app.use("/api/v1/group",GroupRoutes);
44
45  //running:
46  const port=process.env.PORT;
47  const mongourl=process.env.MONGO_URI;
48  app.listen(port, async()=>{
49      try {
50          const connection=await mongosse.connect(mongourl);
51          if(connection)
52          {
53              console.log(`Server running on http://localhost:${port}`);
54              console.log('MongoDB connected successfully')
55          }
```

Fig 1: API endpoints defined in server.js using REST principles with mongoose connection

```
export function attachWebSocketServer(server) {

                    const history = await Chat.find({ sosId }).sort({ createdAt: 1 });
                    ws.send(JSON.stringify({
                        type: "chat_history",
                        payload: history.map(msg => ({
                            sender: msg.senderType,
                            text: msg.message,
                            timestamp: msg.createdAt,
                        }))
                    }));
                }

            } catch (e) {
                console.error("Failed to process message:", e);
                ws.send(JSON.stringify({ type: "error", message: "Invalid message format" }));
            }
        });

        ws.on("close", () => {
            if (userId && connectedUsers.get(userId)?.dataIntervalId) {
                clearInterval(connectedUsers.get(userId).dataIntervalId);
            }
            connectedUsers.delete(userId);
            console.log(`❌ User disconnected: ${userId}`);
        });

        ws.on("error", (err) => console.error("WS Error:", err));
    });

    console.log("🚀 WebSocket Server is running.");
}
```

Fig. 2: ws server connection setup in backend

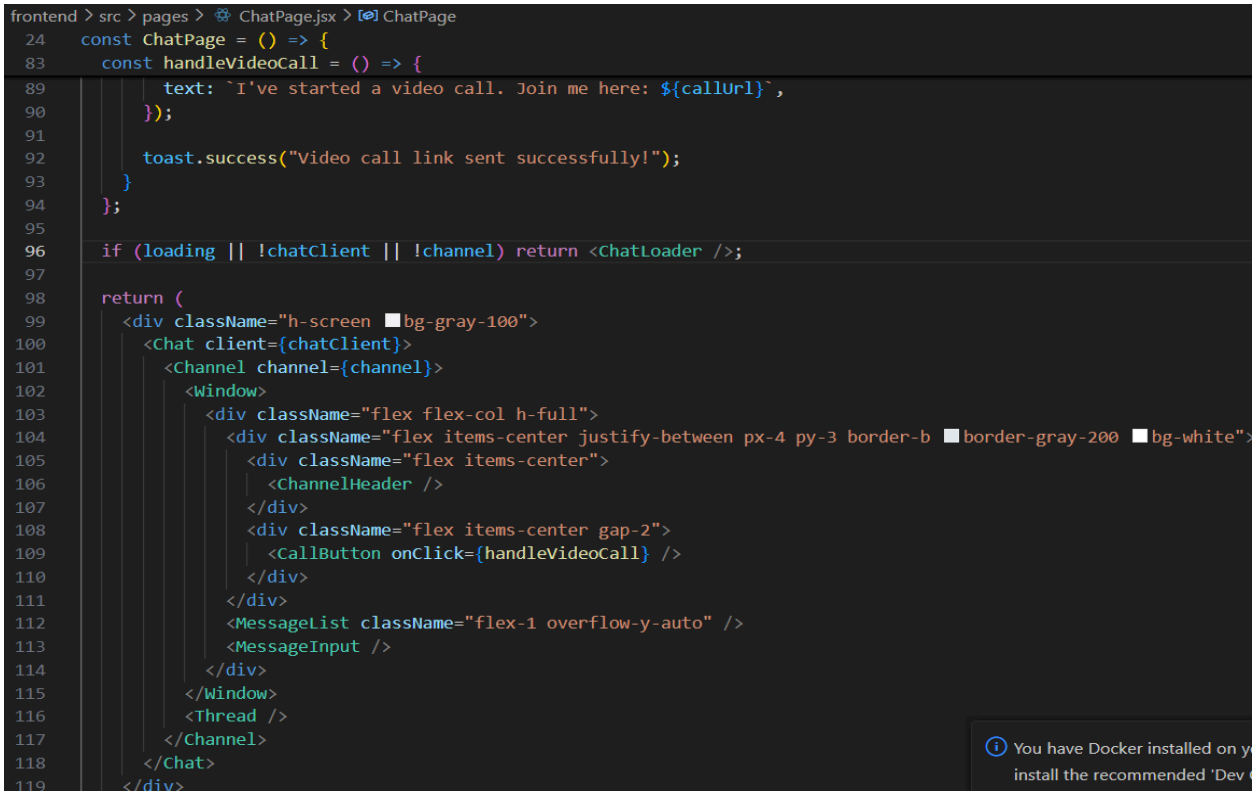

Fig 3: Frontend Chatscreen UI setup.

Output:

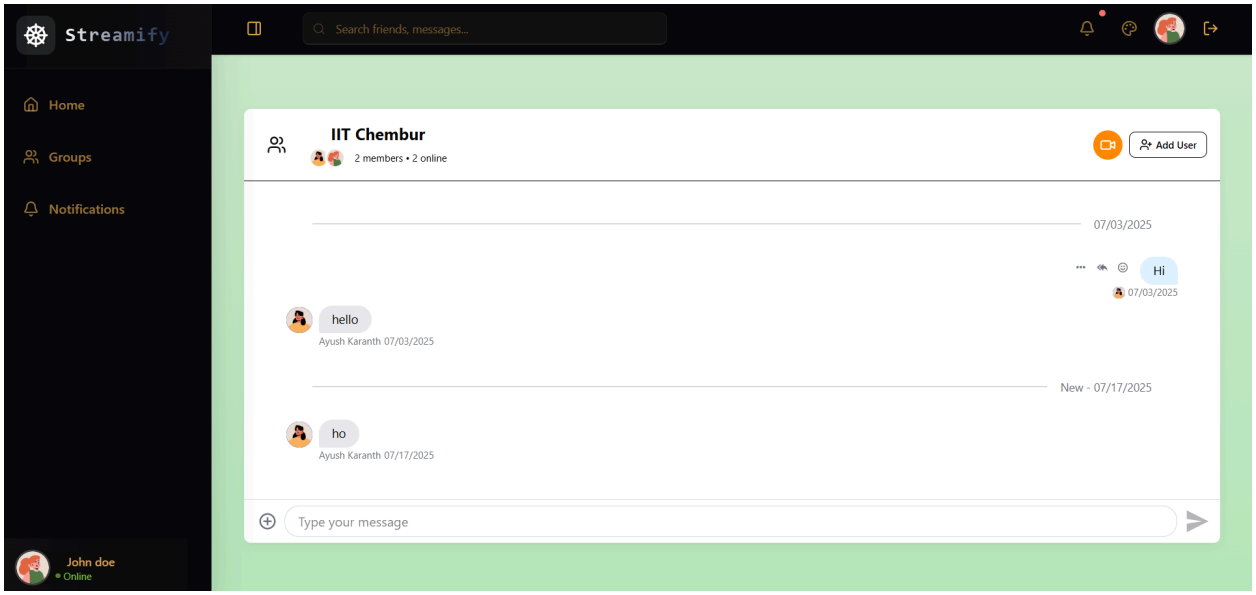

Fig 4: Group Chat Screen

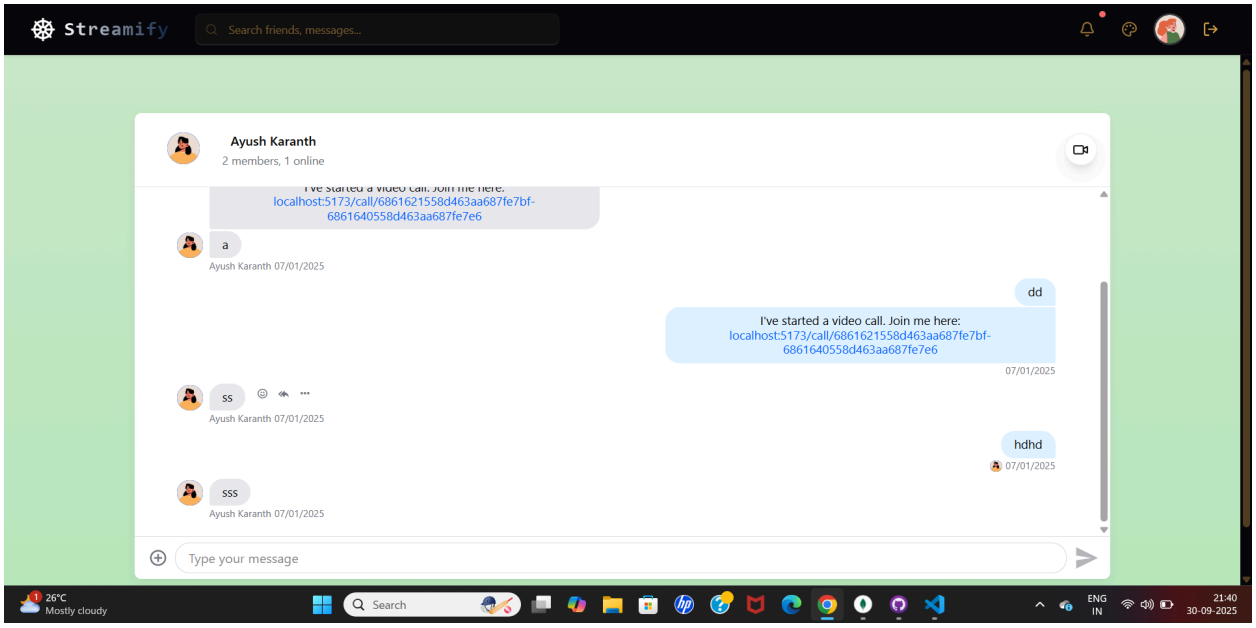

Fig 5: One to One chat screen.

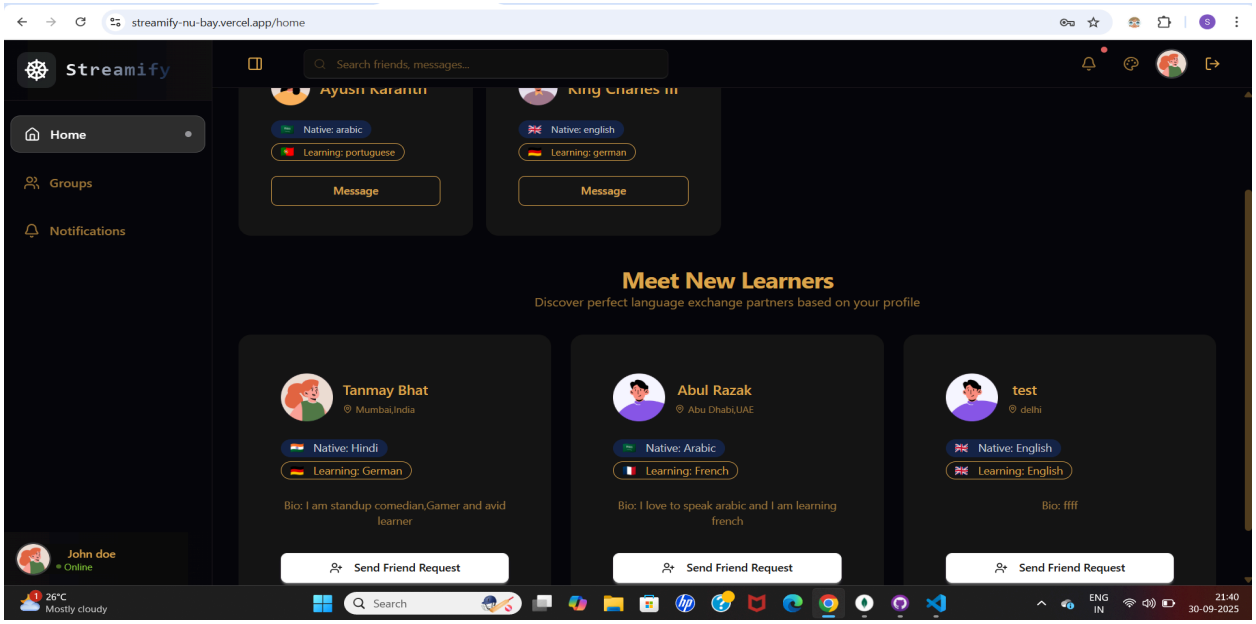

Fig 6: Home Page of Application