WebRTC Video Chat (Vite + React + Node + Socket.IO)

A minimal **WebRTC video chat** app with:

- **React (Vite)** frontend (room join UI + local/remote video + chat)
- **Node.js + Express** backend
- **Socket.IO** signaling (room join, offer/answer exchange, ICE candidates)
- Works on the **same LAN** in dev; can be deployed for **cross-network** usage with HTTPS + TURN.

Table of Contents

- What this app is
- How it works (high level)
- Repository structure
- Frontend (client) explained
- Backend (server) explained
- Local development (LAN)
- Using phones (Android/Samsung)
- Deploy for different networks
  - 1) Deploy frontend to Netlify
  - 2) Deploy backend to Render
  - 3) Add TURN (important)
- Common problems & fixes
- Security notes
- License

What this app is

This project is a **peer-to-peer (P2P) WebRTC** video chat:

- Your **video/audio** streams go directly **between clients** (browser ↔ browser).
- The backend **does NOT** relay video. The backend is only for:
  - *Signaling* (coordinating a call)
  - *Room management*
  - *Chat messages* (if implemented in signaling)

To reliably work across different networks (different Wi-Fi, mobile networks, corporate NAT), you must add:

- **HTTPS** (secure origin)
- **TURN** server (fallback relay when direct P2P fails)

How it works (high level)

WebRTC needs three things:

1. **Signaling** (via Socket.IO)
    - Client A joins a room.
    - Client B joins the same room.
    - They exchange:
        - `offer` / `answer` (SDP)
        - `ice-candidate` messages (network candidate info)
2. **ICE / STUN**
    - WebRTC tries to find the best path between peers.
    - STUN helps discover public-facing addresses.
    - Works often on home Wi-Fi, fails on strict NAT.
3. **TURN (relay)**
    - When direct connection fails, TURN relays traffic.
    - This is the “it works everywhere” part.

Repository structure

Typical layout:

```
webrtc-videochat/
├─ client/                  # React + Vite frontend
│  ├─ src/
│  │  ├─ ...               # components, pages, socket setup
│  ├─ public/
│  ├─ .env                 # VITE_SOCKET_URL (optional)
│  ├─ vite.config.js
│  ├─ package.json
│  └─ ...
└─ server/                  # Node + Express backend
   ├─ socket/
   │  └─ registerHandlers.js # all Socket.IO signaling events
   ├─ index.js              # server bootstrap
   ├─ package.json
   └─ ...
```

Frontend (client) explained

Key responsibilities

The frontend does 5 main jobs:

1. **Connect to the signaling server (Socket.IO)**
2. **Join a room**
3. **Get local camera/mic** (`navigator.mediaDevices.getUserMedia`)
4. **Create RTCPeerConnection**
5. **Exchange WebRTC messages**
    - create/send offer
    - receive/send answer
    - send/receive ICE candidates
    - attach streams to `<video>` elements

Socket initialization

A clean LAN-safe setup is:

```
import { io } from "socket.io-client";

// Default: connect to same host where the page is opened, port 3001.
const URL =
  import.meta.env.VITE_SOCKET_URL ||
  `http://${window.location.hostname}:3001`;

export const socket = io(URL, {
  transports: ["websocket"],
});
```

Why this matters

- If you open the frontend on another device using `http://192.168.x.x:5173`, then `window.location.hostname` becomes that same LAN IP.
- So Socket.IO automatically connects to `http://192.168.x.x:3001`.

Environment variable (.env)

`client/.env` (optional)

```
VITE_SOCKET_URL=http://192.168.68.109:3001
```

Only use this if you want to force a specific backend address.
If you deploy the backend later, set it to your production URL.

WebRTC connection (concept)

Inside your call logic, you will usually do:

- Create peer connection:

```
const pc = new RTCPeerConnection({ iceServers });
```

- Add local tracks:

```
localStream.getTracks().forEach(track => pc.addTrack(track,
localStream));
```

- Handle remote tracks:

```
pc.ontrack = (event) => {
  remoteVideo.srcObject = event.streams[0];
};
```

- Send ICE candidates:

```
pc.onicecandidate = (event) => {
  if (event.candidate) socket.emit("ice-candidate",
event.candidate);
};
```

- Offer/answer flow:

```
const offer = await pc.createOffer();
await pc.setLocalDescription(offer);
socket.emit("offer", offer);
```

Backend (server) explained

Key responsibilities

The backend **does not handle video**. It handles signaling:

- User connects via Socket.IO
- User joins a room
- User sends signaling events:
  - `join-room`
  - `offer`
  - `answer`

    - `ice-candidate`
- Server forwards those events to the other peer(s) in the room

Example server (bind to LAN)

`server/index.js`:

```
const express = require("express");
const http = require("http");
const cors = require("cors");
const { Server } = require("socket.io");
const { registerHandlers } = require("./socket/registerHandlers");

const app = express();

app.use(cors({ origin: true, credentials: true, methods: ["GET", "POST"]
}));
app.get("/health", (_, res) => res.json({ ok: true }));

const httpServer = http.createServer(app);

const io = new Server(httpServer, {
  cors: { origin: true, credentials: true, methods: ["GET", "POST"] },
});

registerHandlers(io);

const PORT = 3001;

// IMPORTANT: bind to all interfaces so other LAN devices can connect.
httpServer.listen(PORT, "0.0.0.0", () => {
  console.log(`Server running on http://0.0.0.0:${PORT}`);
});
```

`registerHandlers(io)`

This file should contain the Socket.IO event logic, typically:

- join a room:
    - `socket.join(roomId)`
    - notify others in room
- relay offer/answer/candidates to the other user(s)

Local development (LAN)

Step 1 — Start backend (server)

From project root:

```
cd server
npm install
node index.js
```

Check it:

- On your main machine: `http://localhost:3001/health`
- On another LAN device: `http://<YOUR_LAN_IP>:3001/health`

If `/health` does not open from another device, it is a network/firewall issue.

Step 2 — Start frontend (client)

```
cd client
npm install
npm run dev -- --host 0.0.0.0
```

Vite prints something like:

- Local: `http://localhost:5173/`
- Network: `http://192.168.xx.yy:5173/`

Step 3 — Open on devices

- On the host machine:
  `http://localhost:5173`
- On any other PC/phone in the same Wi-Fi/LAN:
  `http://<YOUR_LAN_IP>:5173`

Both devices must join the **same Room ID**.

Using phones (Android/Samsung)

Important: HTTPS requirement

Mobile browsers are stricter:

- Camera/mic and WebRTC are more reliable on **HTTPS**.
- `http://localhost` is special (allowed), but `http://192.168.x.x` may cause permissions issues.

Fast test for backend reachability

On the phone open:

- `http://<YOUR_LAN_IP>:3001/health`

Must show:

```
{"ok": true}
```

If this fails:

- phone is not reaching your backend port (router isolation / firewall / wrong network)

If Join button does nothing

Usually means:

- `getUserMedia()` is blocked/refused
- user didn't grant camera permission
- insecure origin restrictions (HTTP)

Recommended fix: deploy with HTTPS (see deployment section).

Deploy for different networks

If you want calls across **different LANs** (friends, mobile data, etc.), you must deploy:

- Frontend: **Netlify** (HTTPS)
- Backend: **Render** (or any VPS) (HTTPS/WSS)
- TURN: Twilio / Metered / Coturn

1) Deploy frontend to Netlify

Option A — Drag & Drop (your choice)

1. Build the frontend locally:

   ```
   cd client
   npm install
   npm run build
   ```

2. This creates:
   - `client/dist/` (static files)
3. Go to Netlify → **Sites** → **Deploy manually**
4. Drag & drop the **`client/dist`** folder into Netlify.

After deploy, you get:

- `https://your-site.netlify.app`

Netlify environment variable

In Netlify:

- Site settings → Environment variables
- Add:
  - `VITE_SOCKET_URL=https://your-backend.onrender.com`

Then redeploy.

---

2) Deploy backend to Render

Render is beginner friendly for Node.

1. Push your repo to GitHub
2. Render → New → Web Service
3. Select your repo
4. Root directory: `server`
5. Build command:

```
npm install
```

6. Start command:

```
node index.js
```

7. Add environment variables (optional):
   - `NODE_ENV=production`

Render gives you:

- `https://your-backend.onrender.com`

Your Socket.IO URL becomes that domain.

---

3) Add TURN (important)

What is TURN?

TURN is a relay server.
Without TURN:

- WebRTC works on easy home networks
- fails on strict NAT / corporate Wi-Fi / mobile networks

With TURN:

- WebRTC works almost everywhere
- but relaying costs bandwidth (TURN providers charge)

Add ICE servers in client

Example `iceServers`:

```
const iceServers = [
  { urls: "stun:stun.l.google.com:19302" },
  // TURN (example - replace with your provider credentials)
  {
    urls: "turn:YOUR_TURN_HOST:3478",
    username: "YOUR_USERNAME",
    credential: "YOUR_PASSWORD",
  },
];
```

Use a provider:

- Twilio (TURN credentials API)
- Metered (easy dashboard)
- Self-host coturn (advanced)

Common problems & fixes

1) Other devices can't open `http://<ip>:5173`

Fix:

- Vite must bind to all interfaces:

  ```
  npm run dev -- --host 0.0.0.0
  ```

- Ensure both devices are on same Wi-Fi
- Disable router "AP isolation / client isolation"

2) Other devices can't open `http://<ip>:3001/health`

Fix:

- Server must listen on `0.0.0.0`
- Allow incoming connections in firewall

3) Mobile join does nothing

Fix:

- Use HTTPS (deploy)
- Ensure camera permission allowed in browser settings

4) Video connects but remote stays black

Usually:

- ICE negotiation failed (needs TURN)
  Fix:
- Add TURN

Security notes

- Current CORS is open (`origin: true`) for dev.
- In production, restrict `origin` to your exact Netlify domain.

License

MIT (or choose your own)