Setup a WebSocket Server with Cloudflare

1. What WebRTC is

WebRTC (**Web Real-Time Communication**) is a set of APIs built into modern browsers that lets two peers (e.g., two users in React apps) **connect directly** to each other to exchange:

- Audio/video streams
- Data (via a “data channel”) — chat messages, files, game state, etc.

The magic: it works **peer-to-peer**, not through a central server (though servers are still used to help them connect).

---

2. The Core Pieces

For two apps to talk over WebRTC, you need:

a) **Signaling**

- Before peers connect, they must exchange “connection setup” info (called **SDP offers/answers** and **ICE candidates**).
- This is usually done via a server using **WebSockets**, **HTTP POST**, or any other channel you choose.
- Example: your React app might send the connection info through a Node.js/Express WebSocket server.

b) **ICE / STUN / TURN**

- WebRTC peers must figure out how to reach each other across the internet (even behind NAT/firewalls).
- **STUN servers**: help discover the public IP/port of each peer.

- **TURN servers**: relay data if direct P2P fails (fallback).
- WebRTC handles this automatically if you give it the server addresses.

c) **PeerConnection**

- In code: `new RTCPeerConnection()`
- This object manages the whole connection, media, and data.

d) **Data Channel**

- In code: `peerConnection.createDataChannel("chat")`
- Lets you send arbitrary text or binary data → perfect for chat and file transfers.

---

3. Flow of a Connection

Here’s what happens when User A chats with User B:

1. **A creates an RTCPeerConnection.**
2. **A creates a data channel** (`chat`).
3. **A creates an SDP offer** (basically: "here’s what I support").
4. **A sends the offer to B** via your signaling server.
5. **B receives the offer**, creates an RTCPeerConnection, and sets it as `remoteDescription`.
6. **B creates an SDP answer** (like: "ok, here’s what I support").
7. **B sends the answer back to A** via the signaling server.
8. **Both A and B exchange ICE candidates** until they find a working route.
9. **Connection established** □□
    - Now data (chat messages, files) or media (audio/video streams) flows directly between browsers.

---

4. Chat Example (DataChannel)

```
// Peer A
const pc = new RTCPeerConnection();
const channel = pc.createDataChannel("chat");
```

```
channel.onmessage = (event) => {
  console.log("Got message:", event.data);
};

channel.onopen = () => {
  channel.send("Hello from A!");
};
```

On Peer B:

```
const pc = new RTCPeerConnection();
pc.ondatachannel = (event) => {
  const channel = event.channel;
  channel.onmessage = (event) => {
    console.log("Got message:", event.data);
  };
};
```

---

5. File Transfer Example

WebRTC DataChannels support **binary blobs**, so you can send files chunk-by-chunk:

```
// Sender
function sendFile(file, channel) {
  const chunkSize = 16384; // ~16 KB chunks
  let offset = 0;

  const reader = new FileReader();
  reader.onload = (e) => {
    channel.send(e.target.result);
    offset += e.target.result.byteLength;
    if (offset < file.size) {
      readSlice(offset);
    } else {
      channel.send("EOF"); // signal end of file
    }
  };

  function readSlice(o) {
    const slice = file.slice(o, o + chunkSize);
    reader.readAsArrayBuffer(slice);
  }

  readSlice(0);
}
```

Receiver just listens for binary data and reconstructs the file.

6. Why This Is Cool

- **Low latency** → no central server relaying chat or files.
- **Efficient file transfer** → can be faster than traditional uploads.
- Works in **all modern browsers** (React can just wrap the APIs).

□□ So in your React app, the “chat + file transfer” works because:

- You use **WebRTC DataChannels** to send text or binary data.
- A **signaling server** helps the two peers find each other and exchange connection info.
- Once connected, the peers exchange messages/files **directly**.

---

1. What signaling needs

Signaling is just about exchanging:

- **SDP offers/answers** (connection descriptions)
- **ICE candidates** (network routes)

This requires **bidirectional, asynchronous communication** between peers (via your server).

---

2. Options compared

□ WebSockets

- **Persistent, bidirectional** connection → server and client can both push messages.

- Handles **real-time** signaling well.
- Scales decently (lots of libraries and servers like Socket.IO).
- Good match for WebRTC's event-driven flow (offer → answer → ICE candidates).

⚠ HTTP (long polling / REST)

- Works, but is **clunky**:
  - Client has to poll server repeatedly for new messages.
  - Higher latency (not great if you want snappy connection setup).
  - Server load is heavier (more requests).
- Might make sense if:
  - You already have an HTTP-only environment.
  - You're connecting infrequently and can tolerate delay.

---

3. Hybrid approach

Some apps start with **HTTP POST** to register a peer (like "here's my offer") and then switch to **WebSocket** or another channel for ongoing exchange. But for your React chat/file transfer app, that's overkill — just stick with WebSockets.

---

4. Rule of thumb

- **If real-time matters (chat, file transfer, video, games): → WebSockets.**
- **If it's rare / one-off signaling (like joining a Zoom meeting via invite link): → HTTP can work, but WebSockets still nicer.**

□□ So for your use case (chat + file sharing between React apps), **WebSockets are definitely the better choice** for signaling.

Would you like me to show you a **minimal WebSocket + WebRTC signaling server (Node.js)** that you could plug into your React app as a starting point?

1. Why you can use PHP

- **WebRTC doesn’t care what language your signaling server is in.**
  All it needs is a way to exchange JSON messages (offers, answers, ICE candidates).
- WebSockets are just a protocol — PHP can handle them.

---

2. Options for PHP WebSocket servers

- **Ratchet** (most popular PHP WebSocket library)
  - Built on ReactPHP (event loop).
  - Handles multiple connections easily.
- **Workerman** (another solid async PHP framework).
- Native PHP WebSocket scripts exist, but they’re much more limited.

---

3. Minimal Ratchet Example

Install via Composer:

```
composer require cboden/ratchet
```

Server (server.php):

```
<?php
use Ratchet\MessageComponentInterface;
use Ratchet\ConnectionInterface;

require __DIR__ . '/vendor/autoload.php';

class SignalingServer implements MessageComponentInterface {
    public function onOpen(ConnectionInterface $conn) {
        echo "New connection: {$conn->resourceId}\n";
```

```
    }

    public function onMessage(ConnectionInterface $from, $msg) {
        // Broadcast to all clients except sender
        foreach ($from->httpRequest->getUri()->getHost() as $client) {
            if ($from !== $client) {
                $client->send($msg);
            }
        }
    }

    public function onClose(ConnectionInterface $conn) {
        echo "Connection {$conn->resourceId} closed\n";
    }

    public function onError(ConnectionInterface $conn, \Exception $e) {
        echo "Error: {$e->getMessage()}\n";
        $conn->close();
    }
}

$server = Ratchet\App('localhost', 8080);
$server->route('/signaling', new SignalingServer, ['*']);
$server->run();
```

Run it:

```
php server.php
```

---

4. Connecting from React

In your React app:

```
const ws = new WebSocket("ws://localhost:8080/signaling");

ws.onopen = () => {
  console.log("Connected to signaling server");
};

// Send offer/answer/ICE candidate
ws.send(JSON.stringify({ type: "offer", sdp: offer }));

ws.onmessage = (event) => {
  const message = JSON.parse(event.data);
  console.log("Got signaling message", message);
```

```
};
```

---

5. Is PHP the best choice?

- If your app is **already running PHP (LAMP stack, WordPress, Laravel, etc.)** → Ratchet is fine.
- If you want **scalability** or **lots of concurrent users** → Node.js / Go / Python WebSocket servers perform better.

---

1. Does Cloudflare support WebSockets?

 Yes.

- Cloudflare has supported **WebSocket proxying** for years on **all plans** (Free, Pro, Business, Enterprise).
- Your WebSocket server just needs to listen on a port Cloudflare allows (usually 80/443 through HTTPS/WSS).

---

2. How Cloudflare Tunnel affects it

- **Cloudflare Tunnel (cloudflared)** creates a secure outbound-only connection from your server → Cloudflare → the internet.
- If your React app connects to `wss://yourdomain.com/signal`, Cloudflare terminates TLS and then forwards the WebSocket traffic through the tunnel to your server.
- From your server's perspective, it just sees a normal WebSocket connection.

So yes, **your PHP WebSocket server will work fine through Cloudflare Tunnel**.

3. Things to watch out for

- **Long-lived connections:**
  WebSockets rely on staying open. Cloudflare supports long-lived connections (up to 24 hours on free tier, unlimited on Enterprise). If a connection drops, your app should be able to reconnect.
- **Ports:**
  If you're behind Cloudflare, you generally can't just expose random ports like `:8080` directly.
  Instead:
  - Run your WebSocket server on `:443` (wss) or `:80` (ws).
  - Or let `cloudflared` map `wss://yourdomain.com/signal` → your local server's `localhost:8080`.
- **Performance:**
  WebRTC **media/data traffic does not flow through Cloudflare** — only the signaling WebSocket does.
  Once peers connect P2P, audio/video/files bypass Cloudflare.
  (Exception: if TURN relay is needed, then that server must be public too, and Cloudflare isn't involved.)

---

4. Example Setup

1. Run your PHP Ratchet WebSocket server on `localhost:8080`.
2. Start a Cloudflare tunnel:

   `cloudflared tunnel run my-tunnel`
3. In Cloudflare dashboard, map a subdomain (e.g., `signal.example.com`) to `http://localhost:8080`.
   Cloudflare will handle upgrading HTTP → WebSocket automatically.
4. In your React app:

   `const ws = new WebSocket("wss://signal.example.com");`

 Result:

- React app → Cloudflare → tunnel → PHP WebSocket server.
- WebRTC peers connect directly (only signaling goes through Cloudflare).

---

1. Prerequisites

- A **domain** managed by Cloudflare (e.g. `example.com`)
- Cloudflare Tunnel (`cloudflared`) installed on your server
- Your **PHP WebSocket server** running locally on port `8080` (Ratchet example earlier)

---

2. Create a named tunnel

`cloudflared tunnel create signaling-tunnel`

This generates a tunnel UUID and credentials file, usually at:

`~/.cloudflared/<tunnel-uuid>.json`

---

3. Configure the tunnel (YAML)

Create a config file at `~/.cloudflared/config.yml`:

```
tunnel: signaling-tunnel
credentials-file: /home/youruser/.cloudflared/<tunnel-uuid>.json

ingress:
  - hostname: signal.example.com
    service: http://localhost:8080
  - service: http_status:404
```

Explanation:

- `hostname`: the public domain/subdomain you want (`signal.example.com`)
- `service`: the local service (your PHP WebSocket server on `localhost:8080`)
- The last catch-all rule (`http_status:404`) ensures requests not matching go nowhere

---

4. Connect the tunnel to Cloudflare

Run:

```
cloudflared tunnel route dns signaling-tunnel signal.example.com
```

This tells Cloudflare DNS to point `signal.example.com` → your tunnel.

---

5. Run the tunnel

Foreground (debugging):

```
cloudflared tunnel run signaling-tunnel
```

Background (production):

```
sudo systemctl enable cloudflared
sudo systemctl start cloudflared
```

---

6. React client connection

Now in your React app, connect securely:

```
const ws = new WebSocket("wss://signal.example.com");
```

```
ws.onopen = () => {
  console.log("Connected to signaling server via Cloudflare Tunnel");
};
```

---

7. Important notes

- **Use** `wss://` (not plain `ws://`) — Cloudflare will terminate TLS for you.
- Cloudflare automatically upgrades HTTP → WebSocket, so you don’t need any special handling in PHP.
- Your WebSocket server doesn’t need to know about Cloudflare; it just sees normal connections.
- Only **signaling traffic** goes through Cloudflare. The actual WebRTC peer-to-peer data (chat, files, media) bypasses Cloudflare once established.

□□ With this setup, you get:

- Secure **WebSocket signaling** (`wss://signal.example.com`)
- Automatic TLS from Cloudflare
- No need to expose random ports — everything runs through 443

WebRTC signaling with PHP (Ratchet) + React client

This document contains a **minimal, production-aware** example showing:

- A PHP WebSocket signaling server using **Ratchet** that supports rooms and direct peer-to-peer routing (offers/answers/ICE).
- A React client snippet (hooks) that shows how to use the signaling server to exchange SDP and ICE and establish a WebRTC DataChannel for chat + file transfer.
- Notes on Cloudflare Tunnel integration and production recommendations.

1) Server: Ratchet-based signaling server

Install:

composer require cboden/ratchet

server.php

```
<?php
// server.php
require __DIR__ . '/vendor/autoload.php';

use Ratchet\MessageComponentInterface;
use Ratchet\ConnectionInterface;
use Ratchet\Server\IoServer;
use Ratchet\Http\HttpServer;
use Ratchet\WebSocket\WsServer;

class SignalingServer implements MessageComponentInterface {
// Map of resourceId => connection
private $clients;
// rooms: roomName => [clientId => connection]
private $rooms = [];

public function __construct() {
$this->clients = new \SplObjectStorage;
echo "Signaling server started\n";
}

public function onOpen(ConnectionInterface $conn) {
$this->clients->attach($conn);
// store metadata on the connection object
```

```
$conn->clientId = null;
$conn->room = null;
echo "New connection: {$conn->resourceId}\n";
}

public function onMessage(ConnectionInterface $from, $msg) {
$data = json_decode($msg, true);
if (!$data) return;

switch ($data['type'] ?? '') {
case 'join':
// { type: 'join', room: 'room1', clientId: 'alice' }
$room = $data['room'];
$clientId = $data['clientId'];
$from->clientId = $clientId;
$from->room = $room;
if (!isset($this->rooms[$room])) $this->rooms[$room] = [];
$this->rooms[$room][$clientId] = $from;

// Notify other participants about new peer
foreach ($this->rooms[$room] as $id => $conn) {
if ($conn !== $from) {
$conn->send(json_encode([
'type' => 'peer-joined',
'clientId' => $clientId,
]));
}
```

How it works

- Clients `join` a room with a unique `clientId`.
- When sending signaling messages (SDP/ICE), clients send `type: 'signal'` and include `to` and `payload`.
- The server routes `signal` messages only to the intended recipient inside the same room.

2) React client (hooks) — minimal working flow

This is a stripped-down React hook and helper to show the signaling flow. It focuses on **DataChannel** (chat + files) but can handle media tracks too.

```
// useWebRTC.js
import { useEffect, useRef, useState } from 'react';

export default function useWebRTC({ signalingUrl, room, clientId }) {
const pcRef = useRef(null);
const wsRef = useRef(null);
const dataChannelRef = useRef(null);
const [connectedPeers, setConnectedPeers] = useState([]);

useEffect(() => {
const ws = new WebSocket(signalingUrl);
wsRef.current = ws;

ws.onopen = () => {
ws.send(JSON.stringify({ type: 'join', room, clientId }));
};

ws.onmessage = async (evt) => {
const msg = JSON.parse(evt.data);
if (msg.type === 'peer-joined') {
// a new peer arrived — you may choose to offer immediately or wait
setConnectedPeers((p) => [...p, msg.clientId]);
}

if (msg.type === 'peer-left') {
setConnectedPeers((p) => p.filter(id => id !== msg.clientId));
}

if (msg.type === 'signal') {
const { from, payload } = msg;
await handleSignal(from, payload);
}
};
```

```
ws.onclose = () => console.log('signaling closed');

return () => {
ws.close();
};
}, [signalingUrl, room, clientId]);

function sendToServer(obj) {
wsRef.current?.send(JSON.stringify(obj));
}

async function createPeerConnection(targetClientId, isInitiator = false) {
const pc = new RTCPeerConnection({
iceServers: [{ urls: 'stun:stun.l.google.com:19302' }]
});

pc.onicecandidate = (e) => {
if (e.candidate) {
sendToServer({
type: 'signal',
to: targetClientId,
from: clientId,
payload: { type: 'ice', candidate: e.candidate }
});
}
};
```

Notes on file transfer

- Use a chunked approach (e.g. 16KB slices) and send ArrayBuffers over the data channel. Include headers like `{ fileId, seq, total, meta }` in the binary protocol or send JSON control messages.
- Always respect `dataChannel.bufferedAmount` to avoid memory spikes (pause sending until it drains).

---

3) Signaling message format

Use small JSON envelopes. Examples used above:

- Join: `{ type:'join', room:'room1', clientId:'alice' }`
- Server -> peer-joined: `{ type:'peer-joined', clientId:'alice' }`
- Signal (client->server->client): `{ type:'signal', to:'bob', from:'alice', payload: { type:'offer'|'answer'|'ice', sdp?, candidate? } }`

This keeps routing simple and deterministic.

---

4) Cloudflare Tunnel reminder

- Your React client should connect to `wss://signal.example.com` (wss required).
- `cloudflared` maps `signal.example.com` -> `http://localhost:8080` (or port you choose).
- Cloudflare will proxy WebSocket frames to your Ratchet server; server code does not need Cloudflare-specific changes.

---

5) Production recommendations & extras

- **Authentication:** require clients to authenticate (JWT) before joining a room. Pass token in the initial `join` message or as a query string on the `wss://` URL (use secure cookies or headers where possible).
- **Scaling:** Ratchet on a single server is fine for modest scale. For many concurrent clients you'll need sharding or a pub/sub (Redis) to coordinate messages across multiple instances.
- **TURN server:** WebRTC media/data is P2P. If peers are behind symmetric NATs, include a TURN server (coturn) in your ICE config and make it publicly reachable.
- **Keepalive / reconnection:** implement ping/pong and auto-reconnect logic for WS.

- **Logging/metrics:** log joins/leaves and error rates; track active sockets.

---

6) How to run locally

1. Start PHP server: `php server.php` (ensure `composer install` executed).
2. Start `cloudflared tunnel run signaling-tunnel` (if using Cloudflare tunnel).
3. Serve your React app and point signalingUrl to `wss://signal.example.com` or `ws://localhost:8080` for local dev.

If you want, I can also:

- Provide a more complete React example (UI + file send/receiv

Done — I created a code + instructions document in the canvas titled **Webrtc-php-ratchet-signaling-and-react**. It includes:

- A Ratchet PHP signaling server that supports rooms and routing `offer` / `answer` / `ice` messages to specific peers.
- A compact React hook (`useWebRTC`) showing how to join, create offers/answers, exchange ICE, and open a DataChannel for chat/files.
- File-transfer tips, message formats, Cloudflare Tunnel reminders, and production recommendations.

Open the canvas (the document named **Webrtc-php-ratchet-signaling-and-react**) to copy the server and client code and follow the run instructions.

---

1. Do you need special Nginx config?

It depends how you’re exposing your WebSocket server:

- **If you're using Cloudflare Tunnel** (as we outlined earlier):
  - Nginx doesn't see the WebSocket traffic at all — `cloudflared` maps your domain directly to the Ratchet server on `localhost:8080`.
  - No special Nginx config is needed for signaling. Nginx can still serve your React app or APIs on the same domain or another subdomain.
- **If you want Nginx to proxy WebSockets** to Ratchet (without Cloudflare):
  - You need to enable **upgrade headers** in the proxy config. Example:

```
server {
    listen 443 ssl;
    server_name signal.example.com;

    ssl_certificate     /etc/letsencrypt/live/signal.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/signal.example.com/privkey.pem;

    location / {
        proxy_pass http://127.0.0.1:8080;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

    Key part is:

```
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "Upgrade";
```

    That allows Nginx to upgrade the HTTP request to a WebSocket connection.

---

2. Does the PHP Ratchet server need to be persistent?

Yes

- Ratchet is an **event loop server** (like Node.js).
- Unlike normal PHP web apps (short-lived scripts under FPM/Apache), this **must run as a long-lived process** because it maintains open WebSocket connections.

That means:

- You run it via `php server.php`, and it stays running.
- You should manage it like any daemon:
  - With **systemd** (recommended): create a unit file so it auto-restarts and runs on boot.
  - Or with **supervisord**, `pm2`, or Docker.

Example `systemd` unit (`/etc/systemd/system/ratchet.service`):

```
[Unit]
Description=Ratchet PHP WebSocket Signaling Server
After=network.target

[Service]
ExecStart=/usr/bin/php /var/www/signaling/server.php
Restart=always
User=www-data
WorkingDirectory=/var/www/signaling
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=ratchet

[Install]
WantedBy=multi-user.target
```

Then:

```
sudo systemctl daemon-reload
sudo systemctl enable ratchet
sudo systemctl start ratchet
```

---

3. Summary

- **With Cloudflare Tunnel** → no Nginx config needed.
- **With Nginx reverse proxy** → add `proxy_set_header Upgrade` + `Connection "Upgrade"` for WebSockets.
- **Ratchet must run persistently** → treat it like a daemon, not like a normal PHP-FPM request/response cycle.

□□ Why Node.js fits better than PHP Ratche for signaling

1. **Event-driven runtime**
   - Node was built for long-lived connections (like WebSockets).
   - You don’t need to think about keeping PHP alive or managing a daemon separately.
2. **Ecosystem**
   - `ws` (a very popular WebSocket library) is lightweight and fast.
   - Tons of tutorials, examples, and tools for WebRTC signaling in Node.
3. **Deployment simplicity**
   - Easier to containerize with Docker.
   - Works seamlessly with **Cloudflare Tunnel** (no Nginx needed, just tunnel directly to your Node app).
4. **Code symmetry**
   - Your React frontend is already JS/TS → using Node on the backend means you can reuse types and data structures.

□□ Minimal Node.js WebSocket signaling server

```
// server.js
import { WebSocketServer } from 'ws';

const wss = new WebSocketServer({ port: 8080 });
const rooms = new Map();

wss.on('connection', (ws) => {
  ws.on('message', (msg) => {
    const data = JSON.parse(msg);
    const { type, room, payload } = data;

    if (type === 'join') {
      if (!rooms.has(room)) rooms.set(room, new Set());
      rooms.get(room).add(ws);
      ws.room = room;
    }

    // Broadcast messages to everyone else in the same room
```

```
    if (['offer', 'answer', 'ice', 'chat', 'file'].includes(type)) {
      for (let client of rooms.get(room) || []) {
        if (client !== ws && client.readyState === ws.OPEN) {
          client.send(JSON.stringify({ type, payload }));
        }
      }
    }
  });

  ws.on('close', () => {
    if (ws.room && rooms.has(ws.room)) {
      rooms.get(ws.room).delete(ws);
      if (rooms.get(ws.room).size === 0) {
        rooms.delete(ws.room);
      }
    }
  });
});

console.log("WebSocket signaling server running on ws://localhost:8080");
```

□□ React client-side example

```
const ws = new WebSocket("wss://signal.example.com");

ws.onopen = () => {
  ws.send(JSON.stringify({ type: "join", room: "chat1" }));
};

// Send offer/answer/ice
ws.send(JSON.stringify({ type: "offer", room: "chat1", payload: offer }));
```

□□ Deployment with Cloudflare Tunnel

- Run the server on your host: `node server.js` (listening on port 8080).
- Tunnel it:

  `cloudflared tunnel --url ws://localhost:8080`
- Cloudflare handles SSL and exposure, no Nginx needed.

□ So in your case:

- Keep **Nginx + PHP** for your normal APIs / website.
- Run a **separate Node.js WebSocket signaling server** for WebRTC.
- Expose it with Cloudflare Tunnel (no special Nginx config).

□□ Options if port 8080 is already taken

1. **Pick a different local port** (simplest)
   - Run your Node signaling server on, say, `localhost:9000` or `localhost:7000`.
   - Example:

     `const wss = new WebSocketServer({ port: 9000 });`
   - Expose it with Cloudflare Tunnel:

     `cloudflared tunnel --url ws://localhost:9000`
   - Cloudflare doesn't care which local port you use — you can map it to any public subdomain.

     ```
     ingress:
       - hostname: signal.example.com
         service: ws://localhost:9000
     ```

---

2. **Run multiple services behind one Cloudflare Tunnel**
   - Cloudflare Tunnel supports **routing rules**.
   - Example `config.yml`:

     ```
     tunnel: my-tunnel-id
     credentials-file: /root/.cloudflared/my-tunnel.json

     ingress:
       - hostname: signal.example.com
         service: ws://localhost:9000
       - hostname: api.example.com
         service: http://localhost:8080
       - service: http_status:404
     ```
   - That way, you can keep your existing app on 8080 and just run WebRTC signaling on another port.

---

3. **Use Nginx locally as a multiplexer**
   - If you want to keep everything on port 8080 internally, you can put Nginx in front:
     - Nginx listens on 8080.
     - Proxies `/signal` to `localhost:9000`.
     - Proxies `/` or `/api` to your other app.
   - Then Cloudflare Tunnel points to Nginx only.
   - This way, multiple apps coexist on the same port.

   Example:

```
server {
    listen 8080;

    location /signal/ {
        proxy_pass http://127.0.0.1:9000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
    }

    location / {
        proxy_pass http://127.0.0.1:3000; # your other app
    }
}
```

Recommended

If this signaling server is dedicated, **Option 1 (different port + Cloudflare Tunnel mapping)** is the cleanest and least error-prone. You don’t even need to touch your existing app or Nginx config.

---

1. Example setup

- React frontend → `app.example.com` → served by Nginx on `localhost:3000`
- PHP API → `api.example.com` → running on `localhost:8080`

- WebRTC signaling server (Node.js WebSocket) → `signal.example.com` → running on `localhost:9000`

---

2. Cloudflare Tunnel config file (`/etc/cloudflared/config.yml`)

```
tunnel: my-tunnel-id
credentials-file: /root/.cloudflared/my-tunnel.json

ingress:
  # React frontend
  - hostname: app.example.com
    service: http://localhost:3000

  # PHP API
  - hostname: api.example.com
    service: http://localhost:8080

  # WebRTC signaling server
  - hostname: signal.example.com
    service: ws://localhost:9000

  # Default (catch-all)
  - service: http_status:404
```

> Notes:
>
> - `http://` is for normal HTTP traffic.
> - `ws://` is for raw WebSockets (Node signaling server).
> - `http_status:404` ensures unmatched requests don't hang.

---

3. Start the tunnel

If your tunnel is already created:

```
cloudflared tunnel run my-tunnel-id
```

If not, create one first:

```
cloudflared tunnel create my-tunnel
cloudflared tunnel route dns my-tunnel app.example.com
cloudflared tunnel route dns my-tunnel api.example.com
cloudflared tunnel route dns my-tunnel signal.example.com
```

---

4. DNS setup in Cloudflare

- Each hostname (`app.example.com`, `api.example.com`, `signal.example.com`) should point to the tunnel.
- Cloudflare takes care of SSL certificates for each.

---

5. Verify

- Open `https://app.example.com` → should serve your React app.
- Open `https://api.example.com` → should hit your PHP backend.
- Open `wss://signal.example.com` from your React client → should connect to your Node WebSocket server.

□□ This way you don’t worry about port clashes at all — every local service can run on its own port, and Cloudflare maps them to different hostnames.

---

1. Create the systemd unit file

Create a new file:

```
sudo nano /etc/systemd/system/cloudflared.service
```

Paste this:

```
[Unit]
Description=Cloudflare Tunnel
After=network.target

[Service]
# Adjust the path if cloudflared is elsewhere
ExecStart=/usr/bin/cloudflared tunnel run my-tunnel-id
Restart=always
RestartSec=5s
User=cloudflared
WorkingDirectory=/etc/cloudflared
Environment=LOGFILE=/var/log/cloudflared.log

# Optional logging to syslog
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=cloudflared

[Install]
WantedBy=multi-user.target
```

> □□ Replac`my-tunnel-id` with the name or UUID of your tunnel (you can see it in `cloudflared tunnel list`).
> You might also need to change `/usr/bin/cloudflared` if it's installed elsewhere (`which cloudflared` will tell you).

---

2. Create a dedicated user (optional, but safer)

```
sudo useradd -r -s /usr/sbin/nologin cloudflared
sudo chown -R cloudflared:cloudflared /etc/cloudflared
```

---

3. Enable and start

```
sudo systemctl daemon-reload
sudo systemctl enable cloudflared
sudo systemctl start cloudflared
```

---

4. Check status & logs

```
systemctl status cloudflared
journalctl -u cloudflared -f
```

---

5. Verify it’s working

- Visit your exposed hostnames (`app.example.com`, `api.example.com`, `signal.example.com`).
- Check Cloudflare dashboard → **Zero Trust → Tunnels** to confirm connections.

 With this, your tunnel will:

- Start on boot.
- Auto-restart if it crashes.
- Log to syslog (or the custom log file).

---

1. Create the service file

Save as:

```
sudo nano /etc/systemd/system/signaling.service
```

Example unit:

```
[Unit]
Description=Node.js WebRTC Signaling Server
After=network.target
```

```
[Service]
ExecStart=/usr/bin/node /var/www/signaling/server.js
Restart=always
RestartSec=5s
User=www-data
WorkingDirectory=/var/www/signaling
Environment=NODE_ENV=production

# Optional logging
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=signaling

[Install]
WantedBy=multi-user.target
```

---

2. Adjust paths and user

- `/usr/bin/node` → run `which node` to confirm the correct path.
- `/var/www/signaling/server.js` → change to your actual signaling server location.
- `User=www-data` → pick the same user that owns your app files (you can also create a dedicated user like `signaling`).

---

3. Enable and start

```
sudo systemctl daemon-reload
sudo systemctl enable signaling
sudo systemctl start signaling
```

---

4. Check logs

`systemctl status signaling`
`journalctl -u signaling -f`

---

5. Auto-restart on crash

The `Restart=always` + `RestartSec=5s` ensures that if your Node app crashes, systemd will bring it back up in 5 seconds.

 At this point:

- **cloudflared.service** handles your Cloudflare Tunnel.
- **signaling.service** keeps your Node.js WebSocket server alive.
- Both start automatically on boot and are monitored by `systemd`.

□□ Target Setup

- **Frontend React app** → served by Nginx → `app.example.com`
- **PHP backend (API)** → PHP-FPM via Nginx → `api.example.com`
- **Node.js WebRTC signaling server** → runs separately → `signal.example.com`
- **Cloudflare Tunnel** → maps each subdomain to the right local service

---

1. PHP-FPM + Nginx config (API + frontend)

Example Nginx site config (e.g. `/etc/nginx/sites-available/example.conf`):

```
server {
    listen 3000;
    server_name app.example.com;
```

```
    root /var/www/react-app/build;

    index index.html;
    location / {
        try_files $uri /index.html;
    }
}

server {
    listen 8080;
    server_name api.example.com;

    root /var/www/api/public;

    index index.php;

    location / {
        try_files $uri /index.php;
    }

    location ~ \.php$ {
        include snippets/fastcgi-php.conf;
        fastcgi_pass unix:/var/run/php/php8.2-fpm.sock;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

> Here:
>
> - React build served on **localhost:3000**
> - PHP-FPM API served on **localhost:8080**

---

2. Node.js signaling server

Say you have `/var/www/signaling/server.js`:

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

const wss = new WebSocketServer({ port: 9000 });
console.log("Signaling server running on ws://localhost:9000");

wss.on('connection', (ws) => {
  ws.on('message', (msg) => {
    const data = JSON.parse(msg);
```

```
  console.log("Received:", data);
  // Simple echo or broadcast logic here...
 });
});
```

Runs on **localhost:9000**.

---

3. Cloudflare Tunnel config (`/etc/cloudflared/config.yml`)

```
tunnel: my-tunnel-id
credentials-file: /root/.cloudflared/my-tunnel.json

ingress:
 - hostname: app.example.com
   service: http://localhost:3000

 - hostname: api.example.com
   service: http://localhost:8080

 - hostname: signal.example.com
   service: ws://localhost:9000

 - service: http_status:404
```

---

4. Services managed by `systemd`

Cloudflared

`/etc/systemd/system/cloudflared.service`

```
[Unit]
Description=Cloudflare Tunnel
After=network.target

[Service]
ExecStart=/usr/bin/cloudflared tunnel run my-tunnel-id
Restart=always
RestartSec=5s
User=cloudflared
WorkingDirectory=/etc/cloudflared
```

```
[Install]
WantedBy=multi-user.target
```

Node signaling server

`/etc/systemd/system/signaling.service`

```
[Unit]
Description=Node.js WebRTC Signaling Server
After=network.target

[Service]
ExecStart=/usr/bin/node /var/www/signaling/server.js
Restart=always
RestartSec=5s
User=www-data
WorkingDirectory=/var/www/signaling
Environment=NODE_ENV=production

[Install]
WantedBy=multi-user.target
```

---

5. Enable & start everything

```
sudo systemctl daemon-reload
sudo systemctl enable cloudflared signaling nginx php8.2-fpm
sudo systemctl start cloudflared signaling nginx php8.2-fpm
```

---

6. Result

- `https://app.example.com` → React app via Nginx
- `https://api.example.com` → PHP API via Nginx + PHP-FPM
- `wss://signal.example.com` → WebRTC signaling via Node.js
- Cloudflare Tunnel maps them securely, with SSL managed by Cloudflare

---

With this, you have an **all-in-one blueprint**: PHP + Nginx + Node + Cloudflare Tunnel living together.

You never need to expose raw ports — just map each subdomain to the right service in your `config.yml`.

□□ Enhanced Node.js Signaling + Messaging Server

```
// server.js
import { WebSocketServer } from "ws";

const wss = new WebSocketServer({ port: 9000 });
const rooms = new Map();

function broadcast(room, sender, message) {
  const clients = rooms.get(room) || new Set();
  for (let client of clients) {
    if (client !== sender && client.readyState === client.OPEN) {
      client.send(JSON.stringify(message));
    }
  }
}

wss.on("connection", (ws) => {
  ws.on("message", (raw) => {
    let data;
    try {
      data = JSON.parse(raw);
    } catch {
      return;
    }

    const { type, room, payload } = data;

    if (type === "join") {
      if (!rooms.has(room)) rooms.set(room, new Set());
      rooms.get(room).add(ws);
      ws.room = room;

      console.log(`Client joined room ${room}`);
      return;
    }
```

```
    if (!ws.room) return;

    switch (type) {
      case "offer":
      case "answer":
      case "ice":
        // WebRTC signaling
        broadcast(ws.room, ws, { type, payload });
        break;

      case "chat":
        // Chat messages
        broadcast(ws.room, ws, {
          type: "chat",
          from: ws._id || "anonymous",
          payload,
        });
        break;

      case "file-chunk":
        // File transfer (chunked)
        // payload = { filename, chunk, seq, done }
        broadcast(ws.room, ws, {
          type: "file-chunk",
          from: ws._id || "anonymous",
          payload,
        });
        break;
    }
  });

  ws.on("close", () => {
    if (ws.room && rooms.has(ws.room)) {
      rooms.get(ws.room).delete(ws);
      if (rooms.get(ws.room).size === 0) {
        rooms.delete(ws.room);
      }
    }
  });
});

console.log("Signaling server running on ws://localhost:9000");
```

□□ React Client Example

Connecting & joining

```
const ws = new WebSocket("wss://signal.example.com");

ws.onopen = () => {
  ws.send(JSON.stringify({ type: "join", room: "chatroom1" }));
};
```

Sending chat

```
function sendChat(msg) {
  ws.send(JSON.stringify({ type: "chat", room: "chatroom1", payload: msg }));
}
```

File transfer (chunked)

```
function sendFile(file) {
  const chunkSize = 16 * 1024; // 16 KB
  let offset = 0;

  const reader = new FileReader();

  reader.onload = (e) => {
    const chunk = e.target.result;
    const done = offset + chunk.byteLength >= file.size;

    ws.send(JSON.stringify({
      type: "file-chunk",
      room: "chatroom1",
      payload: {
        filename: file.name,
        chunk: Array.from(new Uint8Array(chunk)),
        seq: offset,
        done
      }
    }));

    offset += chunk.byteLength;
    if (!done) readNext();
  };

  function readNext() {
    const slice = file.slice(offset, offset + chunkSize);
    reader.readAsArrayBuffer(slice);
  }

  readNext();
}
```

Receiving chat / files

```
ws.onmessage = (event) => {
  const data = JSON.parse(event.data);

  if (data.type === "chat") {
    console.log("Chat:", data.from, data.payload);
  }

  if (data.type === "file-chunk") {
    // handle file reconstruction (buffer chunks until done)
    console.log("Received file chunk", data.payload.filename, data.payload.seq);
  }
};
```

□□ Key Points

- **WebRTC first** → Use signaling (`offer`, `answer`, `ice`) to establish P2P.
- **Fallback to WebSocket** → If NAT/firewall blocks P2P, chat and files still work over the signaling server.
- **Chunked files** → Keeps memory usage reasonable; reassemble chunks client-side.

□This way your Node server is:

- Lightweight signaling hub
- Backup chat and file transfer channel

□□ File Receiving & Reassembly in React

State for tracking incoming files

```
import { useRef } from "react";

const incomingFiles = useRef({}); // { filename: { chunks: [], received: 0, done: false } }
```

Handle incoming file chunks

```
ws.onmessage = (event) => {
  const data = JSON.parse(event.data);

  if (data.type === "file-chunk") {
    const { filename, chunk, seq, done } = data.payload;

    if (!incomingFiles.current[filename]) {
      incomingFiles.current[filename] = { chunks: [], received: 0, done: false };
    }

    const fileData = incomingFiles.current[filename];
    fileData.chunks.push({ seq, chunk });
    fileData.received += chunk.length;
    if (done) fileData.done = true;

    // If file is complete, assemble
    if (fileData.done) {
      assembleFile(filename);
    }
  }
};
```

Assemble and download file

```
function assembleFile(filename) {
  const fileData = incomingFiles.current[filename];

  // Sort chunks by seq (in case they arrive out of order)
  fileData.chunks.sort((a, b) => a.seq - b.seq);

  // Convert chunk arrays back into Uint8Array
  const buffers = fileData.chunks.map(c => new Uint8Array(c.chunk));

  // Merge all buffers
  const totalLength = buffers.reduce((sum, b) => sum + b.length, 0);
  const merged = new Uint8Array(totalLength);
```

```
  let offset = 0;
  for (const b of buffers) {
    merged.set(b, offset);
    offset += b.length;
  }

  // Create a downloadable blob
  const blob = new Blob([merged]);
  const url = URL.createObjectURL(blob);

  // Trigger download
  const a = document.createElement("a");
  a.href = url;
  a.download = filename;
  a.click();

  // Cleanup
  URL.revokeObjectURL(url);
  delete incomingFiles.current[filename];
}
```

□□ How It Works

1. Each chunk arrives as `{ filename, chunk, seq, done }`.
2. We collect them in `incomingFiles`.
3. Once `done = true`, we sort chunks and rebuild into a single `Uint8Array`.
4. Create a `Blob` and trigger a browser download.

□ Now your flow is:

- **Send file** → split into chunks, send via WebSocket.
- **Receive file** → collect chunks, assemble when complete, prompt download.
- Works whether WebRTC P2P succeeds or not.

Revision #2
Created 19 September 2025 12:07:48 by peterd
Updated 19 September 2025 12:16:11 by peterd