Setup https:
1. npm install mkcert -g
2. mkcert create-ca
3. mkcert create-cert
4. OPTIONAL: to run it locally, update the files with your local IP

Steps in a WebRTC app
1. getUserMedia() is run - CLIENT1/Init/Caller/Offerer
2. CLIENT1 creates an RTCPeerConnection object called peerConnection
3. the new peerConnection needs STUN servers so other clients can find
    - they will create ICE candidates, for use later
4. CLIENT1 adds localstream tracks (from GUM) to peerConnection
    - we need to associate CLIENT1 feed with peerConnection
5. CLIENT1 creates an offer
    - offer requires a peerConnection with tracks
    - offer = RTCSessionDescription, an object with 2 properties
        1. SDP - codec and other information
        2. Type (offer)
6. CLIENT1 passes offer to peerConnection.setLocalDescription
7. (ASYNC)ICE candidates can now start coming in

Signal server needs to be running for 8 on
- signal server is a node server, it enables the browsers to find/talk each other

8. CLIENT1 emits offer to the signal server (socket.io/node)
    - socket.io server holds it for the other browser
    - associate with CLIENT1
9. (ASYNC) As 7 happens, emit ICE candidates up to the signaling server
    - socket.io server holds it for when another client responds
    - associate the ICE candidates with CLIENT1

CLIENT1 and Signaling server wait for CLIENT2

10. CLIENT2 loads up the webpage
    - io.connect() runs and connects to the socket.io server
11. socket.io emits out the RTCSessionDescription to the new client
    - RTCSessionDescription = an offer
12. CLIENT2 runs getUserMedia()
13. CLIENT2 creates a peerConnection
    - pass in the STUN servers
14. CLIENT2 adds its localstream tracks to peerconnection
15. CLIENT2 creates an answer with createAnswer()
    - createAnswer = RTCSessionDescription
    - same as #5, but with a type of "answer"
16. CLIENT2 hands answer to peerconnection.setLocalDescription()
17. CLIENT2 has the offer (CLIENT1's SDP)
    - CLIENT2 can pass the offer to peerconnection.setRemoteDescription()
18. (ASYNC) Once #16 runs, CLIENT2 can start collecting ICE candidates

Signaling server (socket.io) has been waiting...

19. CLIENT2 emits answer (RTCSessionDesc - sdp/type) up to signaling server
20. (ASYNC) CLIENT2 will listen for tracks/ICE from remote.
    - and is done.
    - waiting on ICE candidates from CLIENT1
    - waiting on tracks from CLIENT1
21. signaling server has been listening for answer. On arrival
    - emit CLIENT2 answer to CLIENT1 (RTCSessionDesc - sdp/type)
22. CLIENT1 takes the answer and passes it to pc.setRemoteDescription()
23. (ASYNC) CLIENT1 waits for ICE candidates and tracks

21 & 23 are waiting for ICE candidates.
    - Once they are exchanged, tracks will exchange
CONNECTED!!