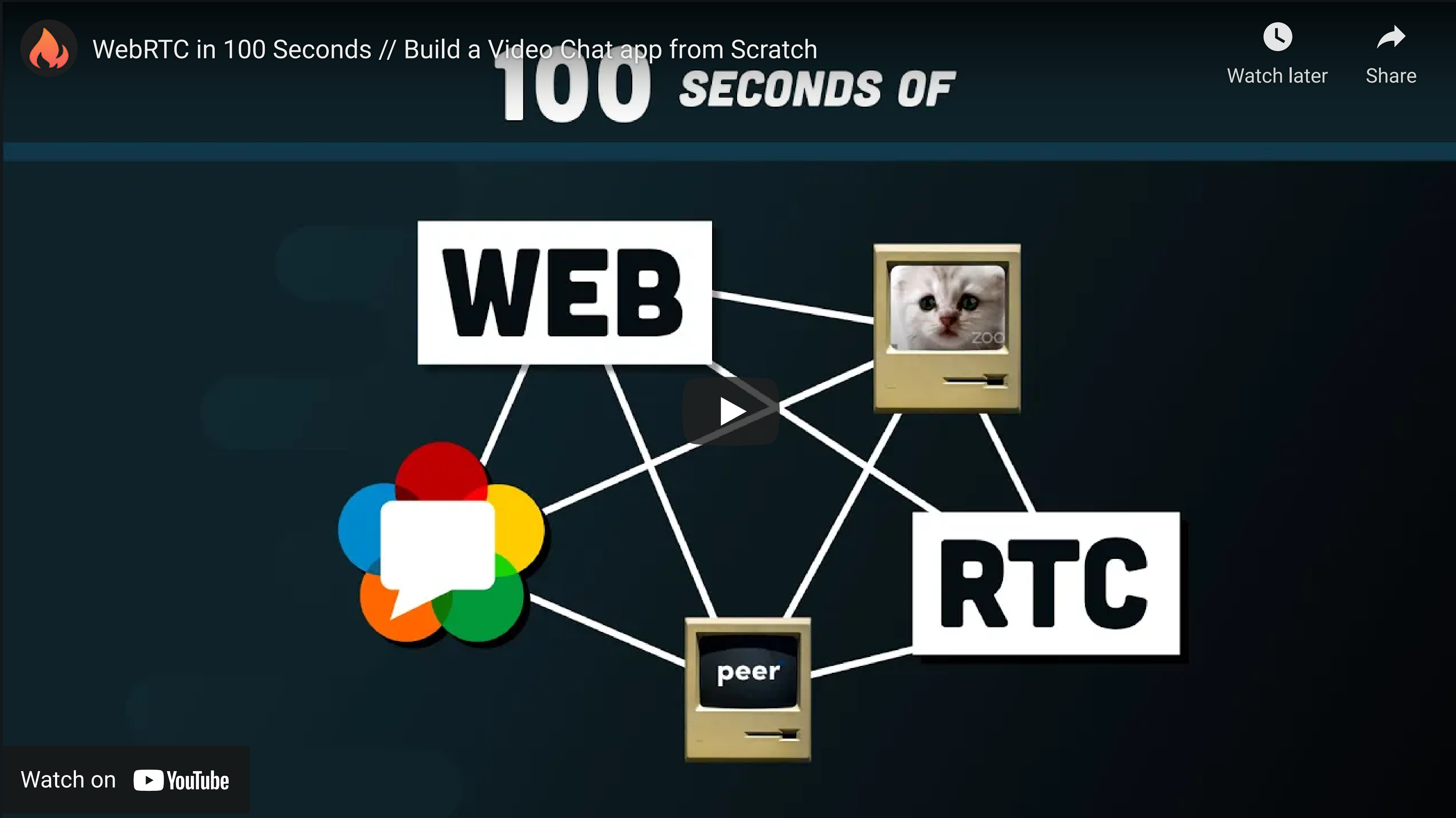
WEBRTC VIDEO CHAT ON FIREBASE

Answer a Call



CODE SLACK

Build a Zoom-like video chat app using

WebRTC and Firebase 859 words.

By Jeff Delaney

Created Mar 2, 2021

Last Updated Mar 2, 2021

#webrtc #firebase #javascript

WebRTC facilities realtime audio/video communication on the web using a peer-to-peer protocol, allowing you to build apps like Zoom, Skype, etc.

The following lesson builds a 1-to-1 video chat, where each peer streams directly to the other peer - there is no need for a middle-man server to handle video content. However, a 3rd party server is required for ***signaling*** that stores shared data for stream negotiation. Firestore is an excellent choice for WebRTC because it is easy to listen to updates to the database in realtime.

Additional Resources:

- Firebase WebRTC Codelab

- Demo with Firebase RealtimeDB

HOW WEBRTC WORKS

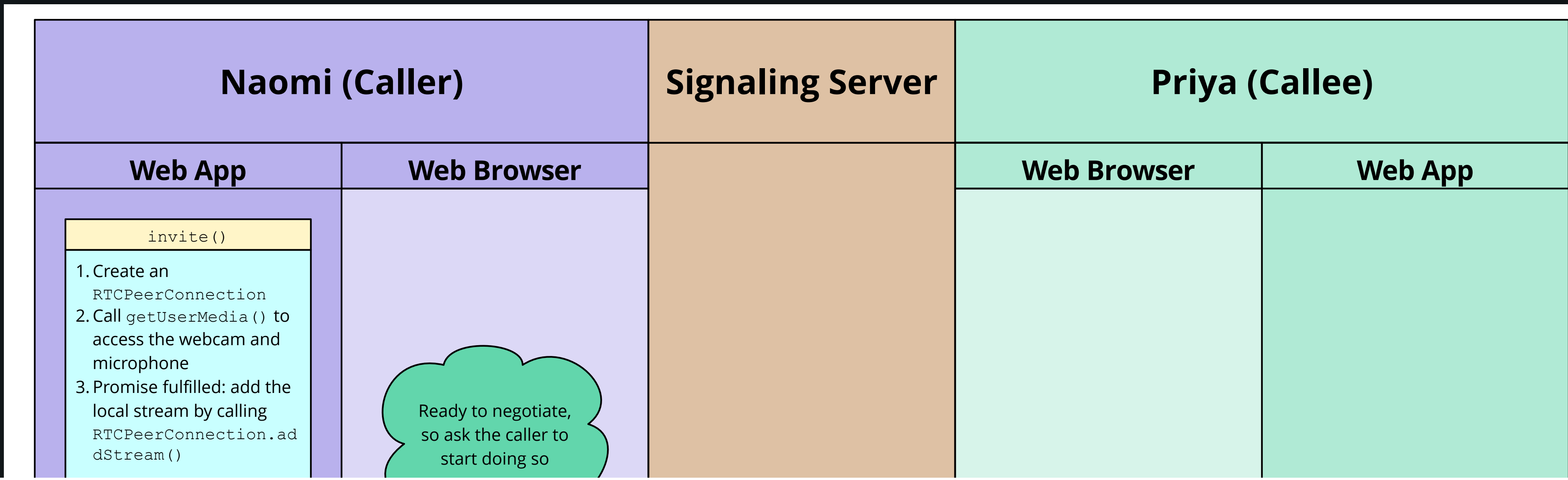
It helps to first understand the steps involved in a WebRTC video chat app from the perspective of the caller and callee.

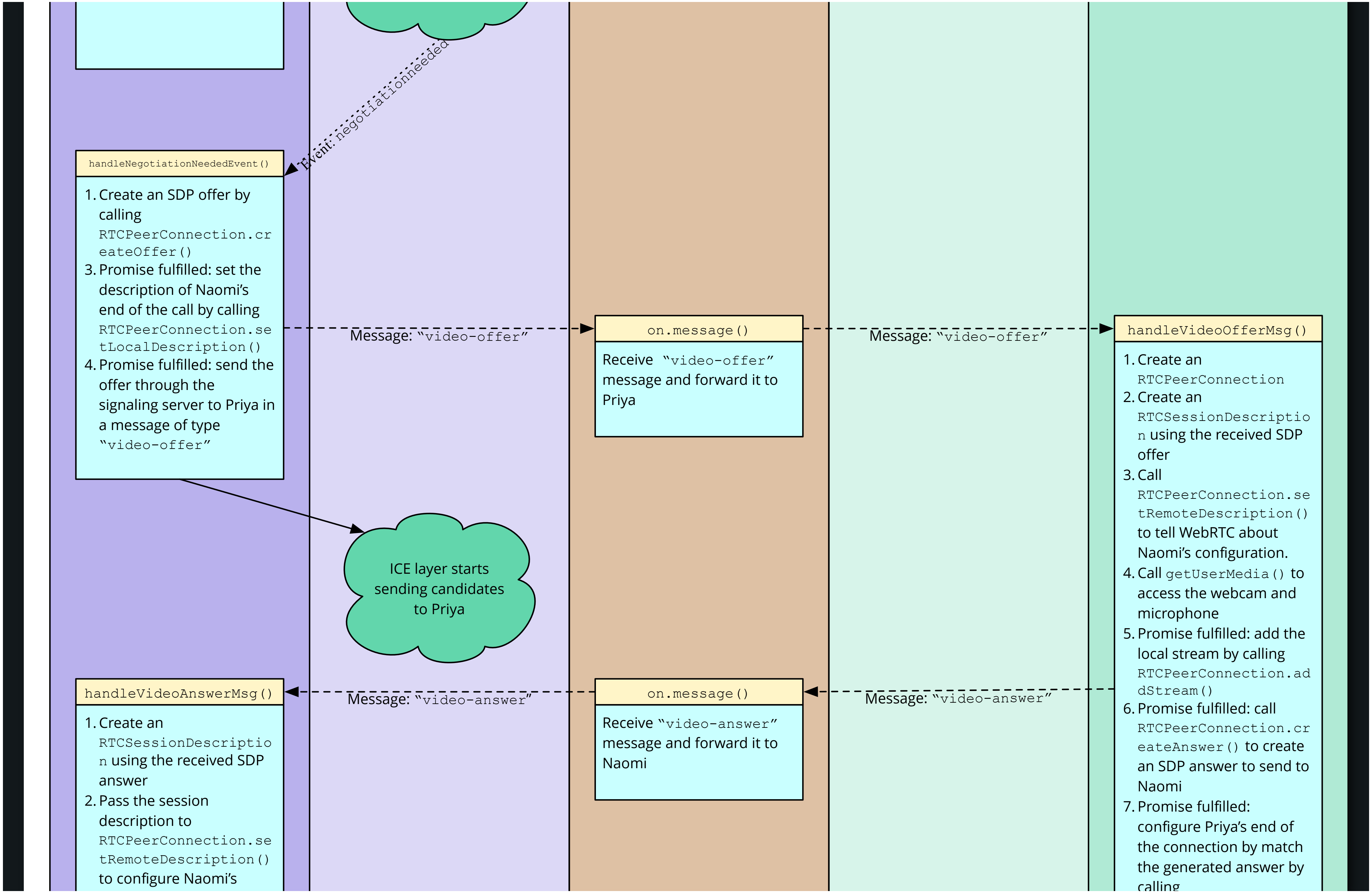
🧑‍🎤 **Caller (Peer 1)**

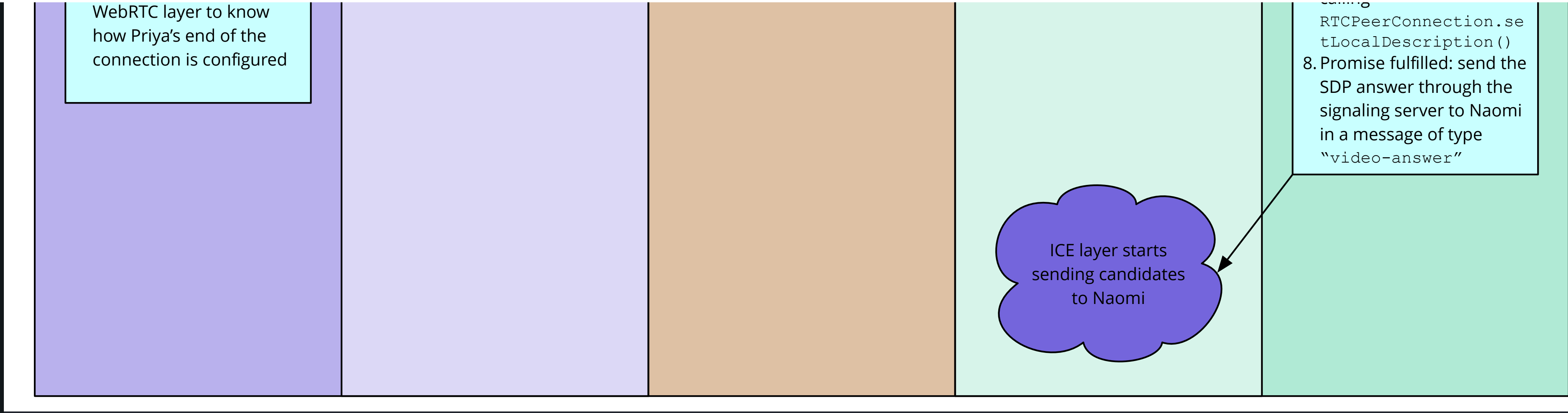
1. Start a webcam feed
2. Create an ‘RTCPeerConnection` connection
3. Call `createOffer()` and write the offer to the database
4. Listen to the database for an answer
5. Share ICE candidates with other peer
6. Show remote video feed

🧑‍🚀 **Callee (Peer 2)**

1. Start a webcam feed
2. Create an ‘RTCPeerConnection` connection
3. Fetch database document with the offer.
4. Call `createAnswer()`, then write answer to database.
5. Share ICE candidates with other peer
6. Show remote video feed




Event: negotiationneeded
handleNegotiationNeededEvent()
1. Create an SDP offer by calling RTCPeerConnection.createOffer()
3. Promise fulfilled: set the description of Naomi's end of the call by calling RTCPeerConnection.setLocalDescription()
4. Promise fulfilled: send the offer through the signaling server to Priya in a message of type "video-offer"
Message: "video-offer"
on.message()
Receive "video-offer" message and forward it to Priya
Message: "video-offer"
handleVideoOfferMsg()
1. Create an RTCPeerConnection
2. Create an RTCSessionDescription using the received SDP offer
3. Call RTCPeerConnection.setRemoteDescription() to tell WebRTC about Naomi's configuration.
4. Call getUserMedia() to access the webcam and microphone
5. Promise fulfilled: add the local stream by calling RTCPeerConnection.addStream()
6. Promise fulfilled: call RTCPeerConnection.createAnswer() to create an SDP answer to send to Naomi
7. Promise fulfilled: configure Priya's end of the connection by match the generated answer by calling
ICE layer starts sending candidates to Priya
handleVideoAnswerMsg()
1. Create an RTCSessionDescription using the received SDP answer
2. Pass the session description to RTCPeerConnection.setRemoteDescription() to configure Naomi's
Message: "video-answer"
on.message()
Receive "video-answer" message and forward it to Naomi
Message: "video-answer"



Breakdown of WebRTC signaling process. Source Mozilla

💡 Mozilla provides an excellent breakdown of the **WebRTC signaling**.

INITIAL SETUP

Setup a basic JS project using **Vite**, then install Firebase.

>_ command line

```
npm init @vitejs/app
```

```
npm install firebase
```

Initialize Firebase with Firestore:

main.js

```
import firebase from 'firebase/app';
import 'firebase/firestore';

const firebaseConfig = {
    // your config
};

if (!firebase.apps.length) {
  firebase.initializeApp(firebaseConfig);
}
const firestore = firebase.firestore();
```

VIDEO CHAT FEATURE

The following code snippets break down the most important concepts when building a video chat feature. Reference the full **source code** for the complete project.

PEER CONNECTION

Create global variables for the **peer connection** and video streams. Notice how the peer connection makes a reference to STUN servers hosted by Google - a STUN server is used to discover a suitable IP/port candidates for establishing a P2P connection.

JS main.js

```
const servers = {
  iceServers: [
    {
      urls: ['stun:stun1.l.google.com:19302', 'stun:stun2.l.google.com:19302'],
    },
  ],
  iceCandidatePoolSize: 10,
};

const pc = new RTCPeerConnection(servers);
let localStream = null;
let remoteStream = null;
```

LOCAL VIDEO STREAM

Create a video feed from a webcam using the **MediaStream** interface. Add the stream tracks to the peer connection.

```
webcamButton.onclick = async () => {
    localStream = await navigator.mediaDevices.getUserMedia({ video: true, audio: true });

    // Push tracks from local stream to peer connection
    localStream.getTracks().forEach((track) => {
        pc.addTrack(track, localStream);
    });

    // Show stream in HTML video
    webcamVideo.srcObject = localStream;
}
```

REMOTE VIDEO STREAM

Initialize a remote video feed with an empty stream. Eventually, the stream will be populated when tracks are added to the peer connection.

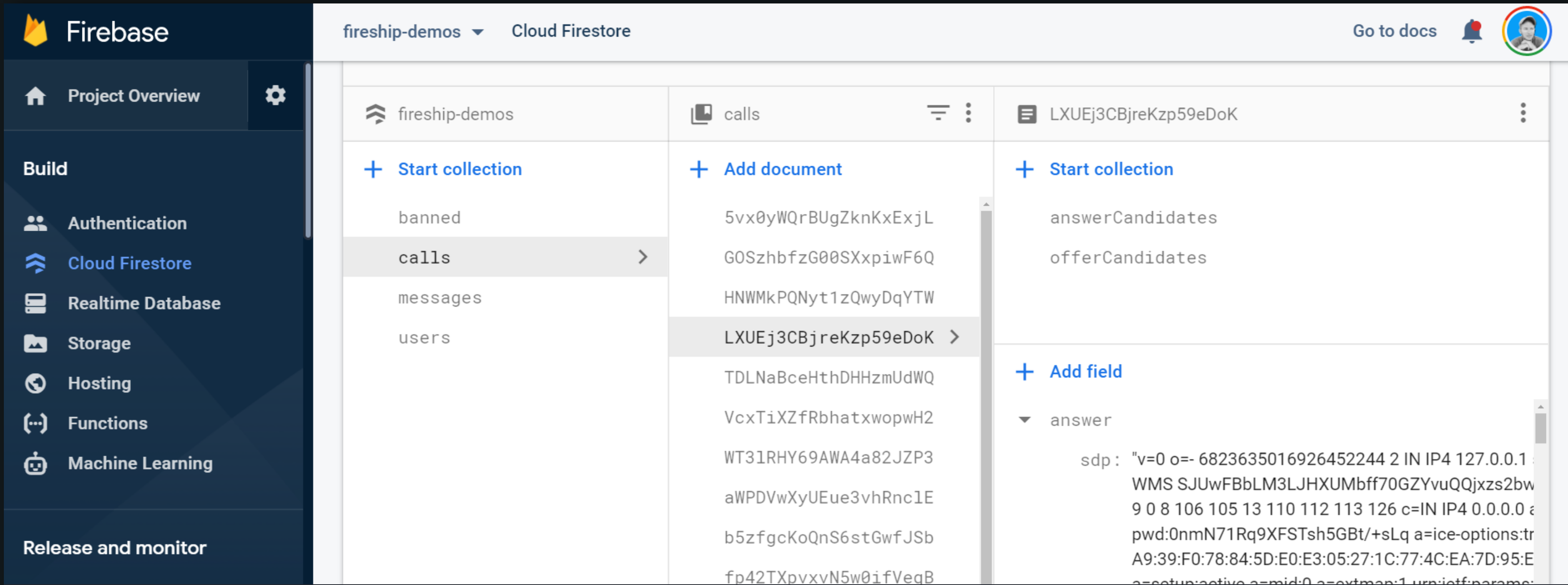
```
remoteStream = new MediaStream();

// Pull tracks from remote stream, add to video stream
pc.ontrack = event => {
    event.streams[0].getTracks().forEach(track => {
        remoteStream.addTrack(track);
    });
};

remoteVideo.srcObject = remoteStream;
```

SIGNALING

To connect two or more peers via WebRTC, each clientside app needs to provide an ICE (Internet Connectivity Establishment) server configuration. The apps must ***signal*** to each other how they should connect, which requires a backend server or database (like Firebase). The database allows the peers to relay the required information to establish a connection.

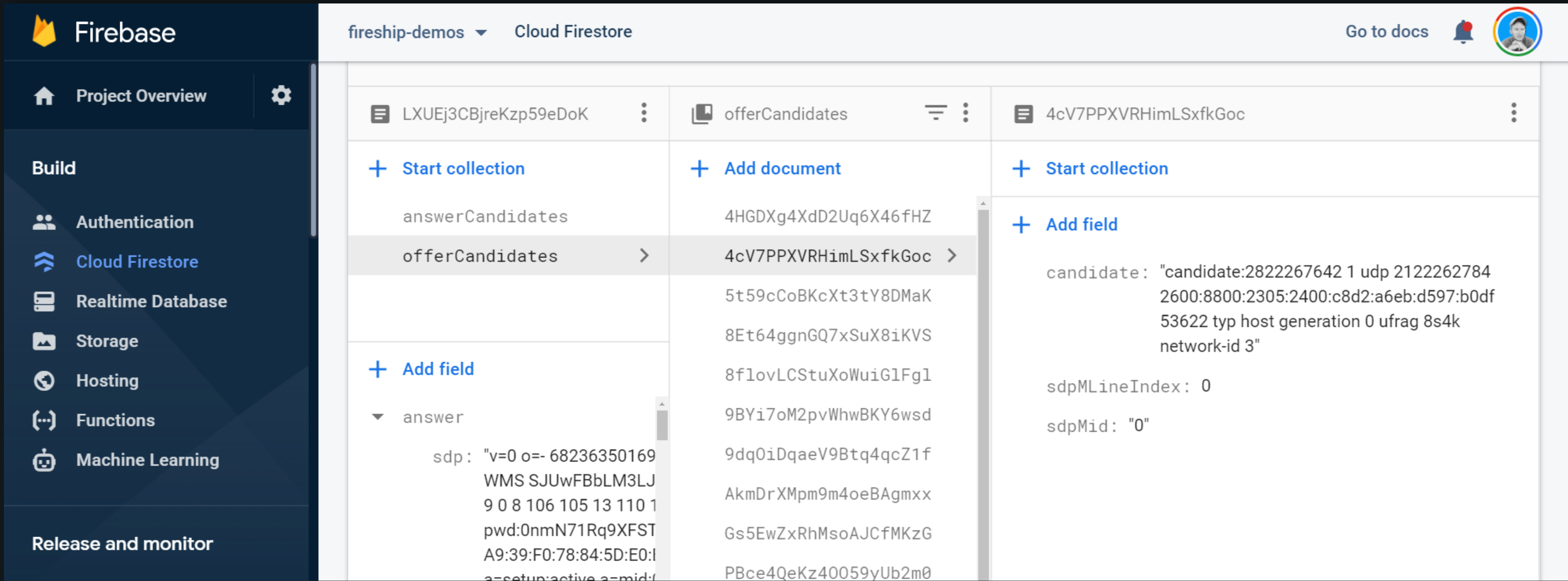
DATA MODEL

The database contains a `calls` collection that stores the offer/answer objects from each peer.



Call document in Firestore

Each call document contains a subcollection for `answerCandidates` and `offerCandidates`.



Candidates subcollections

CREATE A CALL

The caller will reference a new Firestore document with a random ID. The WebRTC peer connection can then create an offer and write the result to the database.

create an offer and write the result to the database.

Once created, the caller will then wait for the document to be updated with an answer from the other user.

```
callButton.onclick = async () => {
// Reference Firestore collections for signaling
  const callDoc = firestore.collection('calls').doc();
  const offerCandidates = callDoc.collection('offerCandidates');
  const answerCandidates = callDoc.collection('answerCandidates');

  callInput.value = callDoc.id;

  // Get candidates for caller, save to db
  pc.onicecandidate = event => {
    event.candidate && offerCandidates.add(event.candidate.toJSON());
  };

  // Create offer
  const offerDescription = await pc.createOffer();
  await pc.setLocalDescription(offerDescription);

  const offer = {
    sdp: offerDescription.sdp,
    type: offerDescription.type,
```

```
  };


  await callDoc.set({ offer });

  // Listen for remote answer
  callDoc.onSnapshot((snapshot) => {
    const data = snapshot.data();
    if (!pc.currentRemoteDescription && data?.answer) {
      const answerDescription = new RTCSessionDescription(data.answer);
      pc.setRemoteDescription(answerDescription);
    }
  });

  // Listen for remote ICE candidates
  answerCandidates.onSnapshot(snapshot => {
    snapshot.docChanges().forEach((change) => {
      if (change.type === 'added') {
        const candidate = new RTCIceCandidate(change.doc.data());
        pc.addIceCandidate(candidate);
      }
    });
  });
}
```

ANSWER A CALL

The process to answer a call is similar, but will reference the existing Firestore document ID, instead of creating a new document.

```
answerButton.onclick = async () => {
  const callId = callInput.value;
  const callDoc = firestore.collection('calls').doc(callId);
  const offerCandidates = callDoc.collection('offerCandidates');
  const answerCandidates = callDoc.collection('answerCandidates');

  pc.onicecandidate = event => {
    event.candidate && answerCandidates.add(event.candidate.toJSON());
  };

  // Fetch data, then set the offer & answer

  const callData = (await callDoc.get()).data();

  const offerDescription = callData.offer;
  await pc.setRemoteDescription(new RTCSessionDescription(offerDescription));
```

```
    const answerDescription = await pc.createAnswer();
    await pc.setLocalDescription(answerDescription);


    const answer = {
      type: answerDescription.type,
      sdp: answerDescription.sdp,
    };

    await callDoc.update({ answer });

    // Listen to offer candidates

    offerCandidates.onSnapshot((snapshot) => {
      snapshot.docChanges().forEach((change) => {
        console.log(change)
        if (change.type === 'added') {
          let data = change.doc.data();
          pc.addIceCandidate(new RTCIceCandidate(data));
        }
      });
    });
  };
```

QUESTIONS?

Ask questions via GitHub below OR chat on Slack #questions

3 Comments - *powered by utteranc.es*

SuryaDaiv commented 2 weeks ago

How to host this on a website. So I execute on two different websites?

siddhant221 commented 4 days ago

what to do after creating answerCandidates and offerCandidates, its not working on my html page

SuryaDaiv commented 4 days ago

> what to do after creating answerCandidates and offerCandidates, its not working on my html page

from what I executed I just created the first level collection (don't remember the name) in Firebase and others got created automatically as we connect the calls.

Write Preview

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub