**PROJECT TASK REPORT**

**Project title: *Video Chatting Website***

**Abstract:**

A website where you can connect with your friends and have one to one video chat with them. It would be fun and you will get to understand how we can send video streams through a connection.

**Table of contents:**

|  |  |
| --- | --- |
| ***Sr. No.*** | ***Title*** |
| *1.* | *Introduction* |
| *2.* | *Existing Method* |
| *3.* | *Proposed method with Architecture* |
| *4.* | *Methodology* |
| *5.* | *Implementation* |
| *6.* | *Conclusion* |
| *7.* | *Sources* |

**Introduction:**

If you weren’t familiar with web video-conferencing platforms and apps before the Covid-19 pandemic, we bet you know all about them now and must have used one of them.

The ongoing COVID-19 situation has many more people working from home than ever before and companies using web conferencing as their primary means of communication between employees and clients.

And with much of the world on lockdown with social distancing measures enforced, even spending time with family and friends is now largely restricted to online video calls. From parties to business meetings, these video conferencing platforms are now commonplace for virtually everyone.

**Django-video-chat** is a demo multi-client video conferencing web app built using Django-channels and web sockets for signaling and WebRTC for p2p connections, video and display streaming. More than 2 peers can be connected. The web application was developed using Microsoft Visual Studio.

**Existing Method:**

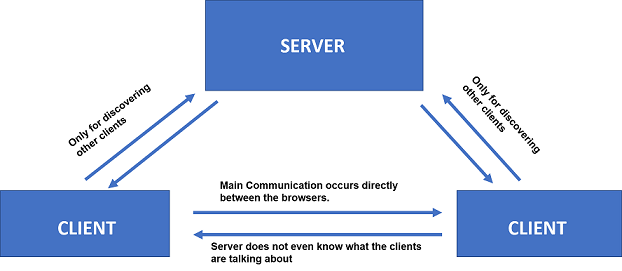
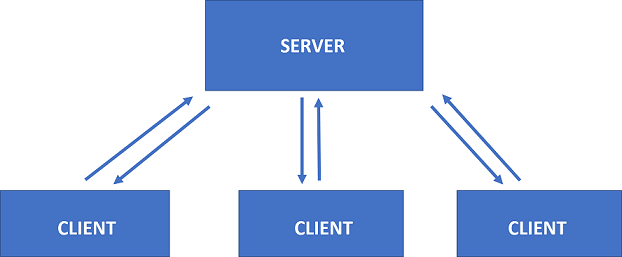
Before we begin to develop such a web app, we need to understand how an e-conferencing application is different from a simple chatting web app.

In a simple chatting web app, when two browsers need to send messages to each other, they typically need a server in between for coordination and passing the messages. But having a server in the middle results in a delay in communication between the browsers. This delay hardly affects the utility of the chatting app. Even if this delay is (say) 5 secs, we would still be able to use this chatting application.

However, in the case of a video conferencing application, this delay is significant. It will be extremely difficult to talk to someone using such an application. Imagine yourself talking to someone who receives your voice 5 secs later. You can realize how annoying it will be.

Hence, for video conferencing, we require **Real-Time Communication** between the browsers. Such communication is possible if we eliminate the server from between. This is why we will have to use **WebRTC** — an open-source framework providing web browsers and mobile applications with real-time communication via simple APIs.

## **WebRTC**

WebRTC stands for Web Real-Time Communication. It enables peer-to-peer communication without any server in between and allows the exchange of audio, video, and data between the connected peers. With WebRTC, the role of the server is limited to just helping the two peers discover each other and set up a direct connection.  


To build an application (that requires peer-to-peer communication) from scratch without WebRTC, you would need a wealth of frameworks and libraries dealing with typical issues like

* data loss
* connection dropping
* NAT traversal
* Echo cancellation
* Bandwidth adaptivity
* Dynamic jitter buffering
* Automatic gain control
* Noise reduction and suppression

With WebRTC, all of this comes **built-in into the browser out-of-the-box.** WebRTC \*\*automatically handles all of these concerns under the hood. \*\*This technology doesn’t need any plugins or third-party software. It is open-sourced and its source code is freely available at <http://www.webrtc.org/>.

Though most of the major browsers (like Chrome, Firefox etc. ) have implemented WebRTC framework and have exposed WebRTC’s APIs for the developers, it is better to confirm that your browser version supports it. You can find the list of all the browsers supporting WebRTC here.

## **WebRTC APIs**

WebRTC consists of several interrelated APIs and protocols which work together to achieve Real Time Communication. The most important APIs that we will use in this tutorial series are — click links to see demos

* getUserMedia(): capture audio and video.
* MediaRecorder: record audio and video.
* RTCPeerConnection: stream audio and video between users.
* RTCDataChannel: stream data between users.

## **Signaling**

Before the two peers can start communicating with each other, they need to know a lot of information about each other like —

* If there is any other peer available for communication.
* Network data, such as a peer’s IP address and port as seen by the outside world
* Session-control messages — used to open or close communication
* Error messages
* Media metadata, such as codecs, codec settings, bandwidth, and media types that will be sent by a peer
* Key data used to establish secure connections

Don’t worry if you do not understand what the above information represents. The important thing is to realize that a lot of information needs to be exchanged before a direct connection can be set-up. Such information can be termed as **metadata.**

Signaling refers to the mechanism which coordinates initial communication and enables sending of metadata between the peers (browsers). Hence, initially, the peers communicate with each other using the signaling mechanism — primarily, for discovering other peers and sharing the information needed to create a direct connection between them. Once the direct connection has been established, there is no role of signaling thereafter.

**Remember** — For signaling, we need a server.

**Session Description Protocol: -**

* The signaling mechanism (methods, protocols, etc.) is not specified by WebRTC. We need to build it ourselves. (Although this seems to be a complicated task, believe us — it is not. In this series, we will use Socket.IO for signaling, but there are many alternatives).
* WebRTC only requires the exchange of the media metadata mentioned above between peers as **offers and answers.** Offers and answers are communicated in **Session Description Protocol** (SDP) format which looks like the following: -  
  v=0 o=- 7614219274584779017 2 IN IP4 127.0.0.1 s=- t=0 0 a=group:BUNDLE audio video a=msid-semantic: WMS m=audio 1 RTP/SAVPF 111 103 104 0 8 107 106 105 13 126 c=IN IP4 0.0.0.0  
  ...
* If you are wondering what each line means in the above format, don’t worry. WebRTC creates this automatically according to the audio/video device present on your laptop/PC.

**Proposed method with Architecture:**

Up till now, we have described what is WebRTC, what is Signaling and what are the various APIs can be used by the developers. Now, let’s discuss how all this works together. Once we know this, we can begin writing the code.

Before we discuss, you must understand what are IP Addresses and PORTS.

* Each device that is connected to the internet, is identified using an IP address.
* Port number identifies a specific process to which an Internet or other network message is to be forwarded when it arrives at the device. The port number is used so the data from the internet is directed to the correct location within the device.

So, each device that is connected to the internet, has an IP Address and many PORTs (typically 65,536).  
RTCPeerConnection **API and signaling: Offer, answer, and candidate**

As we discussed before RTCPeerConnection API of WebRTC is used to stream audio and video between users. Hence, signaling works together with RTCPeerConnection to establish a direct connection between the browsers.

To initialize this process RTCPeerConnection has two tasks:

* Ascertain local media conditions (audio and video), such as resolution and codec capabilities. This is the metadata used for the offer-and-answer and is sent via signaling.
* Get potential network addresses (known as candidates) (which consists of an IP Address and a PORT number) for the app’s host which must be also sent by signaling

Once this local data has been ascertained, it must be exchanged through a signaling mechanism with the remote peer.

Here’s the full offer/answer mechanism in all its detail: First let’s discuss how the information will be shared regarding media conditions.

1. Create an RTCPeerConnection object.
2. Create an offer (an SDP session description) with the RTCPeerConnection createOffer() method.
3. Call setLocalDescription() to set the created offer (Session Description) as the description of local media in the connection that will be created.
4. Stringify the offer and uses a signaling mechanism.
5. Call setRemoteDescription() with offer, so that RTCPeerConnection knows about setup.
6. Call createAnswer() and the success callback function for this is passed a local session description.
7. Set her answer as the local description by calling setLocalDescription().
8. Then use the signaling mechanism to send her stringified answer.
9. Set the remote session description using setRemoteDescription().

The expression “finding candidates” refers to the process of finding network interfaces and ports (present on a peer and are available for establishing a direct connection with the other peer) using the ICE [framework](https://en.wikipedia.org/wiki/Interactive_Connectivity_Establishment).

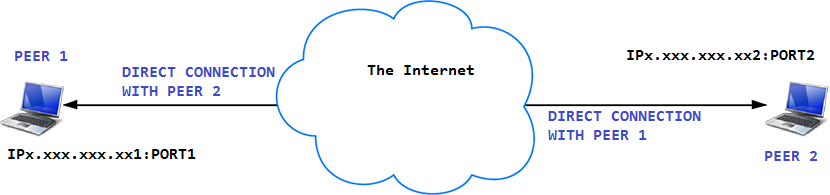
create an RTCPeerConnection object with an onicecandidate handler.

The handler is called when network candidates become available.

WebRTC supports [ICE Candidate Trickling](https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03#section-3.4.1), which allows the caller to incrementally and automatically provide candidates to the callee after the initial offer, and for the callee to automatically begin acting on the call and set up a connection without waiting for all candidates to arrive. Don’t worry if you do not understand [ICE Candidate Trickling](https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03#section-3.4.1). The important thing is WebRTC automatically creates ICE candidates (containing IP address) once a peer creates the offer. We only have to implement the methods that are required to receive and send these candidates via signaling.

Once the information regarding the media conditions and ice candidates are shared between the two peers, WebRTC automatically creates a direct connection between the peers.

## **After Signaling — Use ICE to cope with NATs and firewalls**

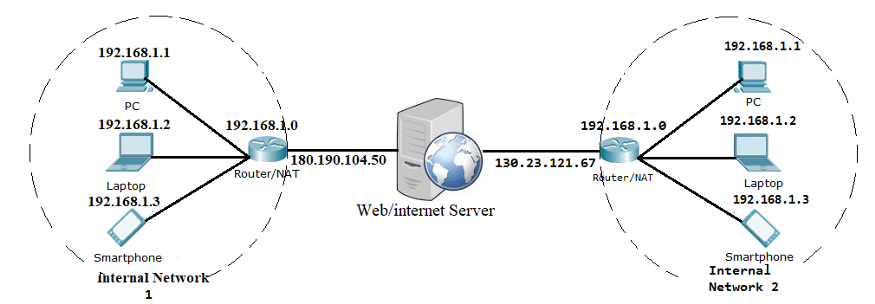
So, it is natural that you would expect that every WebRTC Connection endpoint would have a unique IP address and PORT number that it could exchange with other peers in order to communicate directly.  
But it is not so simple. There are two factors which can cause problems here. We must deal with those before we can use our web conferencing application.

**Problem 1 — NAT**

If you are familiar with Computer Networks, you would know what NAT is. If don’t know, do not worry. We will explain it here:

You already know what IP addresses are. It is an address that identifies a device connected on the internet. Logically, you would expect that each device (which is connected to the Internet) must have a UNIQUE IP Address. But this is not entirely true.

An IPv4 address is 32 bits long which implies that there are about 4 billion unique addresses (2³² = 4,294,967,296). At the end of 2018, there were about 22 billion devices connected to the internet. So, you must be wondering — if there are only 4 billion IP addresses, how can 22 billion devices be connected on the internet? The answer to this is NAT.

The guys, who maintain the internet, came up with the following solution — They divided the whole IPv4 address range into two groups — public IP Addresses and private IP Addresses. Now, each public IP address can be assigned only to one device but the same is not true for private IP addresses. See the image below for more clarification.  
In the above picture, each router has two IP addresses — one Public IP Address (facing the server) and one Private IP Address (facing the Internal Network). So, if any device inside Internal Network 1 sends a request to the server, the server will see the request coming from the same IP Address i.e., 180.190.104.50

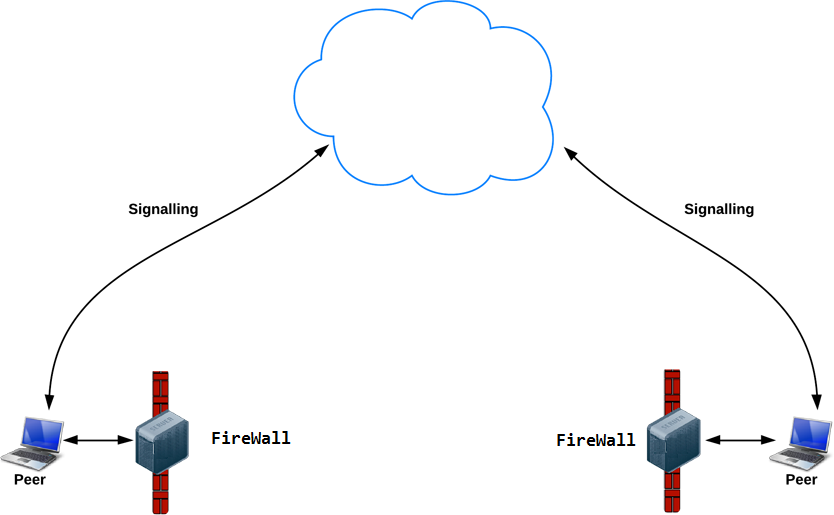
So, this implies that each router maps one Public IP Address to multiple Private IP Addresses of the devices. This also implies that each device (laptop, PC, Smartphone) only knows its private IP Address and not the public IP Address of the router. (Also, if you search on Google — my IP Address, Google will tell you the public IP Address of the router (you are connected to) because Google sees the Public IP Address of the router and not your Private IP Address.)

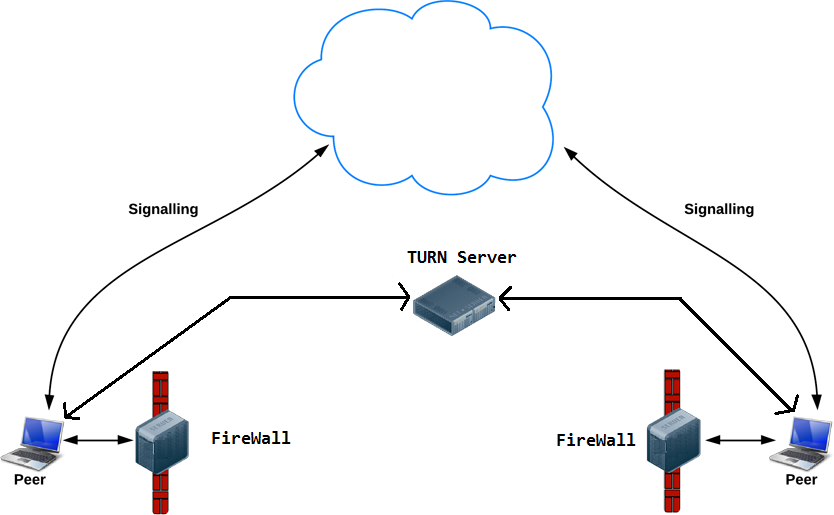
Hence, in a way, we can say that each device has two IP addresses — a Private IP address (assigned to the device) and the Public IP address (assigned to the router to which the device is connected to).

This can cause problem for WebRTC as the network ICE candidates (generated by the browser) contain the private IP address and not the public IP address of the device. Hence, we must find a way for the browser to know the Public IP Address so that it can create candidates containing the Public IP Address. The solution is a STUN (Session Traversal Utilities for NAT) server. When a device makes a request to the STUN server, the STUN server responds back with a message containing the public IP of the router to which the device is connected to. In this way, the STUN server helps the browser to generate candidates.

We will see how to integrate STUN with WebRTC later in the tutorial.

**Problem 2 — Firewall**

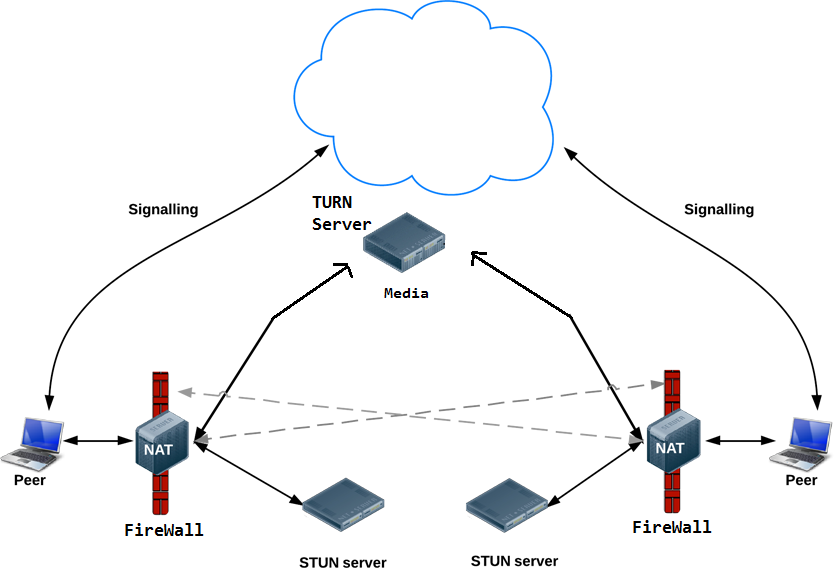
In reality, most devices live behind one or more layers of firewalls which are like antivirus software that blocks certain ports and protocols. A firewall and a NAT may in fact be implemented by the same device, such as a home WIFI router. Since WebRTC uses a number of non-standard Ports, some Firewalls do not allow a direct connection to be made between the two browsers.  


Hence, to solve this, we need a TURN (Traversal Using Relay NAT) server. TURN server basically acts a Relay Server i.e., the relay traffic directly between the two peers if direct (peer-to-peer) connection fails. Following image illustrates: -Solution

As we discussed before, we need to use STUN and TURN servers while making a peer-to-peer connection using WebRTC. To integrate TURN and STUN with webrtc, we only have to pass a object containing the URLs of TURN and STUN servers to the RTCPeerConnection() as its argument. Following code illustrates: -

//Object containing TURN/STUN URLs.var pcConfig = { 'iceServers': [ { 'urls': 'stun:stun.l.google.com:19302' }, { 'urls': 'turn:192.158.29.39:3478?transport=udp', 'credential': 'JZEOEt2V3Qb0y27GRntt2u2PAYA=', 'username': '28224511:1379330808' }, { 'urls': 'turn:192.158.29.39:3478?transport=tcp', 'credential': 'JZEOEt2V3Qb0y27GRntt2u2PAYA=', 'username': '28224511:1379330808' } ]}........//Passing the above object to RTCPeerConnectionRTCPeerConnection(pcConfig);

As illustrated above, we only have to pass the URLs. WebRTC manages everything else under the hood.

Following diagram illustrates all the connections made during a WebRTC call  


**Methodology:**

The following method is used in the development of the website:

* Open Command Prompt. Navigate to the folder in which you want to create the webapp.
* Then create a virtual environment and activate it using venv directory.
* Then install the latest version of the pip (if not installed) and then installing the required dependencies by running commands in command prompt.
* From the directory where we have installed venv, go to the mysite directory by running the command: **cd mysite**
* Start New Project in Django by using the command

**“django-admin startproject webrtc” in command prompt**

## Create New App in Django by using the command

**“python manage.py startapp in command prompt”**

## Run Django by using the command

**“python manage.py runserver**

**python manage.py runserver 8080**

**python manage.py runserver 0:8000**

**python manage.py runserver 192.168.1.4:8000”** In the command prompt

## Create Superuser using the command **python manage.py createsuperuser**

* In command prompt run this command in command prompt for permanant setting

**export DJANGO\_SETTINGS\_MODULE=webrtc.settings.local**

## Run on Local System using command prompt

**DJANGO\_SETTINGS\_MODULE=webrtc.settings.local python manage.py runserver**

## Run on Test Server using command prompt

**DJANGO\_SETTINGS\_MODULE=webrtc.settings.testing python manage.py runserver**

* Run on Production Server using command prompt **python manage.py runserver**
* Static files on Production Server using command prompt

**python manage.py collectstatic**

**python manage.py collectstatic --noinput**

**python manage.py collectstatic --noinput --clear**

* Django Channels Setup using command prompt

**pip3 install -U channels**

* Then add functionality to the website by creating a javascript file and then adding objects and implementing functions. Also implement different channels and frameworks like rtcpeerconnection api form WebRTC,implementing django, in python by creating different files in the folder for adding the following:
* rtcpeerconnection api form WebRTC,implementing django, Create an RTCPeerConnection object.
* Create an offer (an SDP session description) with the RTCPeerConnection createOffer() method.
* Call setLocalDescription() to set the created offer (Session Description) as the description of local media in the connection that will be created.
* Stringify the offer and uses a signaling mechanism.
* Call setRemoteDescription() with offer, so that RTCPeerConnection knows about setup.
* Call createAnswer() and the success callback function for this is passed a local session description.
* Set answer as the local description by calling setLocalDescription().
* Then use the signaling mechanism to send her stringified answer.
* Set the remote session description using setRemoteDescription().
* The expression “finding candidates” refers to the process of finding network interfaces and ports (present on a peer and are available for establishing a direct connection with the other peer) using the ICE [framework](https://en.wikipedia.org/wiki/Interactive_Connectivity_Establishment).
* create an RTCPeerConnection object with an onicecandidate handler.
* The handler is called when network candidates become available.
* In the handler, stringified candidate data is sent through their signaling channel.Call addIceCandidate() to add the candidate to the remote peer description.
* WebRTC supports [ICE Candidate Trickling](https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03#section-3.4.1), which allows the caller to incrementally and automatically provide candidates to the callee after the initial offer, and for the callee to automatically begin acting on the call and set up a connection without waiting for all candidates to arrive. Don’t worry if you do not understand [ICE Candidate Trickling](https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03#section-3.4.1). The important thing is WebRTC automatically creates ICE candidates (containing IP address) once a peer creates the offer. We only have to implement the methods that are required to receive and send these candidates via signaling.
* Once the information regarding the media conditions and ice candidates are shared between the two peers, WebRTC automatically creates a direct connection between the peers.
* Create a routing.py file in Project Directory Add ASGI\_APPLICATION into your settings.py file
* To start the development server, run the command: python manage.py runserver
* For testing on multiple devices in the same LAN, go to the directory where you have installed ngrok. Run the command: ngrok.exe http 8000 This will make our localhost public and provide two public URLs. However, make sure to always use the one that starts with https: and not http: as we will be accessing media devices.
* On local device, go to <http://127.0.0.1:8000/> On other devices, go to the URL from ngrok that starts with https:.
* Once the page is loaded, type a username and click join room from each device. Be sure to use different usernames for now.
* If remote video does not play, click the button that says "Click to play remote video" as some browsers require user gesture to play video.

**Implementation:**

Installation: Go to your desired folder.

Go to the directory with requirements.txt.

Run the command: python -m venv venv

After a venv directory is created, run the command for windows: venv\Scripts\activate.bat

Ensure latest version of pip by running: python -m pip install --upgrade pip

Install the dependencies by running the command: pip install -r requirements.txt

We need multiple devices in the same LAN for testing. For that we need to make our localhost public. For that, download ngrok from <https://ngrok.com/download> and install it.

Usage: From the directory where we have installed venv, go to the mysite directory by running the command: cd mysite

To start the development server, run the command: python manage.py runserver

For testing on multiple devices in the same LAN, go to the directory where you have installed ngrok. Run the command: ngrok.exe http 8000 This will make our localhost public and provide two public URLs. However, make sure to always use the one that starts with https: and not http: as we will be accessing media devices.

On local device, go to <http://127.0.0.1:8000/> On other devices, go to the URL from ngrok that starts with ‘https:’

Once the page is loaded, type a username and click join room from each device. Be sure to use different usernames for now.

If remote video does not play, click the button that says "Click to play remote video" as some browsers require user gesture to play video.

**Conclusion:**

Thus, the webapp **Django-video-chat** is created using the following components mentioned above. This webapp can now be used on different devices and also helps users to connect with more than 2 peers and can also make use of the chat functionality present in the webapp.

**Sources:**

* Coding Entrepreneurs (Django-channels): https://www.youtube.com/watch?v=RVH05S1qab8&t=0s
* Very Academy (Django-channels): https://www.youtube.com/watch?v=F4nwRQPXD8w&t=0s
* Django-channels official documentation: https://www.youtube.com/redirect?event=video\_description&redir\_token=QUFFLUhqbEdHcE5TZXhVU1gtcTY2WFpwa0EyblVGeWNSQXxBQ3Jtc0trRzlBd2pRZkVoTjdnbVJBd0Q4QjRPRDYxTVBIZ1E2TGc3dDVjVFNJTzFoQmkwUmZxTVJPZDBZOWVPbFRTdFY2QlVTSy1fTWt1VUFMR1VNczJZS3ExaGZKODhpQWNxUkROaDJ2OGhXblRhQm1aeks1cw&q=https%3A%2F%2Fchannels.readthedocs.io%2Fen%2Fstable%2F
* Hussein Nasser (WebSocket): https://www.youtube.com/watch?v=2Nt-ZrNP22A&t=0s
* Hussein Nasser (WebRTC): https://www.youtube.com/watch?v=FExZvpVvYxA&t=0s
* WebRTC official docs: <https://webrtc.org/>
* Google WebRTC code Labs: https://www.youtube.com/redirect?event=video\_description&redir\_token=QUFFLUhqbmVGZEN0Ul9nbmFLU1diTEFNZ2tJajEwOEJBUXxBQ3Jtc0trYk1icURXd1g1UXRmcndxaTd5dXZmV0JKQmdyRDc2VWRxeUhiTUNfZFZYQ2wyY05sWi1ZZnlpdmpVM0prSmM0eHBKN3NZVjl5VHNUOFNYNF95eVBsMUR5eGhNNkl1ZVItRVFuZEROTXBacEZSdEdKTQ&q=https%3A%2F%2Fcodelabs.developers.google.com%2Fcodelabs%2Fwebrtc-web%230
* WebRTC Mozilla docs[:](https://webrtc.org/) https://www.youtube.com/redirect?event=video\_description&redir\_token=QUFFLUhqa1pZMmxsZmJWX2pEY0VVSWVvenQtNTlPQWx3Z3xBQ3Jtc0tsYnhXQ3FjQ1ZCWlVRTW1QTWJwV0Rxejh2OU1lUG5mN0VQSjhId3JXY1RrVmItcjVzNmM2N094QWJxalVIZFRxRTd2M2M0UnlHc0tOeXkxQk9XV3FzQ3JXM0FTbzg0ZjJaN2hLXzhUclRfRHp3T1J2WQ&q=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FAPI%2FWebRTC\_API