Star 688 Fork 310

How to use RTCPeerConnection.js? (v1.5) ® Muaz Khan

 » @WebRTC Experiments » Google+ » What's New?

Explains how to

1. use WebRTC peer connection API (RTCPeerConnection.js)
2. write one-to-one video sharing application
3. use socket.io or websockets for signaling

Suggestions

1. If you're newcomer, newbie or beginner; you're suggested to try RTCMultiConnection.js or DataChannel.js libraries.

2. Remember: RTCPeerConnection.js is documented here.
3. Another recommended tutorial is: How to use WebRTC PeerConnection?

Start here..

First of all; you need to reference RTCPeerConnection.js library:

```
<script src="https://www.webrtc-experiment.com/RTCPeerConnection-v1.5.js"></script>
```

Offerer

Now assume that you are creating "offer"...you need to use this code to create offer sdp:

```
var peer = RTCPeerConnection({
    attachStream        : clientStream,
    onICE               : function (candidate) {},
```

```
    onRemoteStream       : function (stream) {},
    onOfferSDP           : function(sdp) {}
});
```

Here is the short explanation of above code ↑

1. attachStream: client stream that you need to share with other peer!

   > A few days later; you may want to attach multiple streams; e.g. one audio+video stream; and one screen sharing stream; in such case, you can use attachStreams object to attach multiple streams:
   >
   > attachStreams = [MediaStream1, MediaStream2, MediaStream3]

---

2. onICE: it returns locally gathered ICE so you can share them with other end.

   candidate object contains two properties:

    1. candidate.sdpMLineIndex
    2. candidate.candidate

3. onRemoteStream: returns remote stream attached by other peer.
4. onOfferSDP: returns offer sdp; so you can send it to other peer to get answer sdp.

Now assume that other peer generated answer sdp and sent to you; you can pass that sdp to this function:

```
peer.addAnswerSDP( answer_sdp );
```

Now assume that other peer gathered ICE and sent to you; you can pass that ICE to this function:

```
peer.addICE({
    sdpMLineIndex       : candidate.sdpMLineIndex,
    candidate           : candidate.candidate
});
```

Answerer

99% process is the same for peer who "creates answer sdp"; the only difference is: for that peer you don't need "onOfferSDP" and also you don't need to call "peer.addAnswerSDP(answer_sdp);". What extra you need to do is here:

```
var peer = RTCPeerConnection({
    attachStream        : clientStream,

    onICE               : function (candidate) {},
    onRemoteStream      : function (stream) {},

    // see below two additions ↓
    offerSDP            : offer_sdp,
    onAnswerSDP         : function(sdp) {}
```

```
});
```

Let me elaborate:

1. offerSDP: you need to pass offer sdp sent by other peer.
2. onAnswerSDP: returns answer sdp so you can send it to other end.

For ICE sent by other peer; you need to do same thing:

```
peer.addICE({
    sdpMLineIndex       : candidate.sdpMLineIndex,
    candidate           : candidate.candidate
});
```

Some quick tips:

1. You MUST get client stream before opening sockets or XHR requests.
2. Offerer can create offer sdp any time; but other peer should start creating answer sdp only when it has offer sdp.
3. Before creating answer; you MUST not add any ICE sent by other peer.

How to write a realtime WebRTC app using socket.io?

First of all; you need to reference socket.io.js:

```
<script src="https://www.webrtc-experiment.com/dependencies/socket.io.js"></script>
```

Now, open socket and transmit your request (e.g. room) until a participant found:

```
var socket = io.connect('http://pubsub.pubnub.com/webrtc-app', {
    publish_key         : 'demo',
    subscribe_key       : 'demo',
```

```
    ssl                 : true,                   /* <<< for HTTPS */
    channel             : 'WebRTC App'
});

socket.on('connect', onconnect);
socket.on('message', oncallback);

/* socket is opened: it is your time to transmit request! */
function onconnect() {}

/* got response */
function oncallback(response) {}
```

Above code is same for both: offerer and answerer.

Offerer

Now, assume that it is "offerer" who transmits request for participant to join him. He will not create "offer sdp" until he receive "join request" from his participant.

Following code is for offerer (95% part of this code can be used for Answerer):

```
function onconnect()
{
    transmitRequest();
}

var userID = 'offerer';             /* unique ID to identify this user */
var foundParticipant = false;

function transmitRequest()
{
    socket.send({
        userID  : userID,
        type    : 'request to join'
    });

    // Transmit "join request" until participant found
    !foundParticipant && setTimeout(transmitRequest, 1000);
}
```

```
function oncallback(response)
{
    // Don't get his own messages
    if(response.userID == userID) return;

    // if participant found
    if(response.participant)
    {
        foundParticipant = true;

        // create offer and send him offer sdp
        createOffer();
    }

    // answer sdp sent to you: complete handshake
    if(response.firstPart || response.secondPart)
    {
        processAnswerSDP(response);
    }
}
```

```
var peer;

function createOffer()
{
    peer = RTCPeerConnection({

        /* function(offer_sdp) {}, */
        onOfferSDP: sendOfferSDP,

        onICE: function(candidate) {
            socket && socket.send({
                userID: userID,
                candidate: {
                    sdpMLineIndex: candidate.sdpMLineIndex,
                    candidate: JSON.stringify(candidate.candidate)
                }
            });
        },
        onRemoteStream: function(stream) {
            if(stream) video.src = webkitURL.createObjectURL(stream);
```

```
        },
        attachStream: clientStream
    });
}

// send offer sdp
function sendOfferSDP(sdp)
{
    var sdp = JSON.stringify(sdp);

    /* because sdp size is larger than what pubnub supports for single request...
    /* that's why it is splitted in two parts */
    var firstPart = sdp.substr(0, 700),
        secondPart = sdp.substr(701, sdp.length - 1);

    /* transmitting first sdp part */
    socket.send({
        userID: userID,
        firstPart: firstPart
    });
```

```
    /* transmitting second sdp part */
    socket.send({
        userID: userID,
        secondPart: secondPart
    });
}

var answerSDP = {};

// got answer sdp, process it
function processAnswerSDP(response)
{
    if (response.firstPart) {
        answerSDP.firstPart = response.firstPart;
        if (answerSDP.secondPart) {
            var fullSDP = JSON.parse(answerSDP.firstPart + answerSDP.secondPart);
            peer.addAnswerSDP(fullSDP);
        }
    }

    if (response.secondPart) {
```

```
            answerSDP.secondPart = response.secondPart;
            if (answerSDP.firstPart) {
                var fullSDP = JSON.parse(answerSDP.firstPart + answerSDP.secondPart);
                peer.addAnswerSDP(fullSDP);
            }
        }
    }
```

That was all you need to do for offerer side.

Answerer

For answerer: you write 95% same code like offerer; there is just a little bit difference.

1. For answerer, you don't use "onOfferSDP" instead you use "onAnswerSDP"
2. For answerer, you've to pass "offerSDP" object containing "offer sdp" sent by offerer
3. For answerer, you don't need to call "peer.addAnswerSDP" because this function is for "offerer" only.

On the "Answerer" side; when user click "join" button; send a message to Offerer to tell him that you're ready to get "offer sdp" from him.

```
var userID = 'answerer';

socket && socket.send({
    participant: true,
    userID: userID
});
```

Here is the function that creates "answer sdp":

```
function createAnswer(offer_sdp)
{
    peer = RTCPeerConnection({
        /* you need to pass offer sdp sent by offerer */
        offerSDP: offer_sdp,
        onAnswerSDP: sendAnswerSDP,
```

```
        onICE: onICE,
        onRemoteStream: onRemoteStream,
        attachStream: clientStream
    });
}
```

For answerer: socket "callback" function will be like this:

```
function oncallback(response)
{
    if(response.userID == userID) return;

    // you can show a "join" button or you can send participant request
    if(response.type && response.type == 'request to join') {}

    // offer sdp sent to you by offerer
    if(response.firstPart || response.secondPart)
    {
        processAnswerSDP(response);
    }
}
```

Remember: you don't need to call "peer.addAnswerSDP(fullSDP)" in processAnswerSDP function; instead call createAnswer like this:

```
createAnswer(fullSDP);
```

You MUST get client stream before creating offer or answer.

How to write same app in WebSocket?

For WebSocket, you need to reference websocket.js instead of referencing socket.io.js

```
<script src="https://www.webrtc-experiment.com/dependencies/websocket.js"></script>
```

You need to create socket like this:

```
"use strict"
```

```
var socket = new WebSocket('wss://pubsub.pubnub.com/demo/demo/webrtc-app');
socket.onmessage = onconnect;
socket.onopen = function(event)
{
    oncallback(event.data);
};
```

All other things are 100% same like above code.

You may also like this guide: WebRTC for Beginners: A getting stared guide!

Latest Updates

RTCMultiConnection audio/video recording support added.

/RTCMultiConnection-v1.4-Demos/RecordRTC-and-RTCMultiConnection.html

Session Reinitation also fixed:

/RTCMultiConnection-v1.4-Demos/session-reinitiation.html

See more demos:

/RTCMultiConnection-v1.4-Demos/

muaz-khan 24 days ago

RecordRTC updated: /RecordRTC/

muaz-khan 24 days ago

RTCall.js updated: /RTCall/

muaz-khan 24 days ago

Updates—Socket.io / WebScockets / RTCMultiConnection-v1.4 / RecordRTC

muaz-khan 1 months ago

RecordRTC-to-PHP demo added. #48

/RecordRTC/PHP/

muaz-khan

1 months ago

RTCMultiConnection-v1.4 fixed for session re-initiation (close/rejoin/leave)

/RTCMultiConnection-v1.4-Demos/session-reinitiation.html

#89 / you can close/open/join unlimited rooms without page refresh. You can join/leave many rooms too. Remember, you must override "onNewSession" to make sure RTCMultiConnection doesn't auto-join first available room. Read more here:

github.com/muaz-kh...

You can prefer TURN candidates by disabling reflexive and host candidates:

connection.candidates.host = false;
connection.candidates.reflexive = false;

You can also prefer using reliable (SCTP-based) data channels:
connection.reliable = true;

muaz-khan 1 months ago

tabCapture API new demo added.

/screen-broadcast/

github.com/muaz-kh...

code.google.com/p/...

muaz-khan 1 months ago

tinny fixes; a little bit updates and some new stuff.

muaz-khan 2 months ago

"removeStream" and "disableDtlsSrtp" added. File-sharing extended.

github.com/muaz-kh...

muaz-khan 2 months ago

No file-sharing crash anymore; also file-sharing extended.

DataChannel.js

github.com/muaz-kh...

File-Hangout:

github.com/muaz-kh...

RTCMultiConnection: (v1.4)

github.com/muaz-kh...

muaz-khan 2 months ago

RTCMultiConnection Changes Log added.

github.com/muaz-kh...

muaz-khan 2 months ago

RecordRTC — MediaRecorder API relevant bug fixed.

/RecordRTC/

muaz-khan 2 months ago

screen.js & meeting.js - "leave" method added. #82

// to stop sharing screen

screen.leave();

/screen-sharing/

// to leave a meeting room

meeting.leave();

/meeting/

muaz-khan 2 months ago

RTCMultiConnection — v1.4, v.15, and v1.6

RTCMultiConnection-v1.4 is standard version. However, it requires multi-sockets for peers connectivity.

RTCMultiConnection-v1.5 is 100% same like v1.4; however it doesn't require multi-sockets. It is capable to work with each and every signaling gateway even if it is SIP, XMPP, XHR, WebSockets, Socket.io etc.

RTCMultiConnection-v1.6 is experimental release. It will be removed when v1.5 gets stable.

RTCMultiConnection v.1.4 and v1.5 allows you be and admin; and invite/eject users in/from your room. As and administrator, you can manage users! Read more here:

github.com/muaz-kh...

muaz-khan 2 months ago

video-conferencing — an important bug fixed.

/video-conferencing/

muaz-khan 2 months ago

Feedback

Have any message? Suggestions or something went wrong?

Send Message

Enter your email too; if you want "direct" reply!