Signaling and video calling

WebRTC allows real-time, peer-to-peer, media exchange between two devices. A connection is established through a discovery and negotiation process called **signaling**. This tutorial will guide you through building a two-way video-call.

WebRTC is a fully peer-to-peer technology for the real-time exchange of audio, video, and data, with one central caveat. A form of discovery and media format negotiation must take place, as discussed elsewhere, in order for two devices on different networks to locate one another. This process is called **signaling** and involves both devices connecting to a third, mutually agreed-upon server. Through this third server, the two devices can locate one another, and exchange negotiation messages.

In this article, we will further enhance the WebSocket chat first created as part of our WebSocket documentation (this article link is forthcoming; it isn't actually online yet) to support opening a two-way video call between users. You can try out this example on Glitch, and you can remix the example to experiment with it as well. You can also look at the full project on GitHub.

Note: If you try out the example on Glitch, please note that any changes made to the code will immediately reset any connections. In addition, there is a short timeout period; the Glitch instance is for quick experiments and testing only.

The signaling server

Establishing a WebRTC connection between two devices requires the use of a **signaling server** to resolve how to connect them over the internet. A signaling server's job is to serve as an intermediary to let two peers find and establish a connection while minimizing exposure of potentially private information as much as possible. How do we create this server and how does the signaling process actually work?

First we need the signaling server itself. WebRTC doesn't specify a transport mechanism for the signaling information. You can use anything you like, from WebSocket to `XMLHttpRequest` to carrier pigeons to exchange the signaling information between the two peers.

It's important to note that the server doesn't need to understand or interpret the signaling data content. Although it's SDP, even this doesn't matter so much: the content of the message going through the signaling server is, in effect, a black box. What does matter is when the ICE subsystem instructs you to send signaling data to the other peer, you do so, and the other peer knows how to receive this information and deliver it to its own ICE subsystem. All you have to do is channel the information back and forth. The contents don't matter at all to the signaling server.

Readying the chat server for signaling

Our chat server uses the WebSocket API to send information as JSON strings between each client and the server. The server supports several message types to handle tasks, such as registering new users, setting usernames, and sending public chat messages.

To allow the server to support signaling and ICE negotiation, we need to update the code. We'll have to allow directing messages to one specific user instead of broadcasting to all connected users, and ensure unrecognized message types are passed through and delivered, without the server needing to know what they are. This lets us send signaling messages using this same server, instead of needing a separate server.

Let's take a look which changes we need to make to the chat server support WebRTC signaling. This is in the file `chatserver.js`.

First up is the addition of the function `sendToOneUser()`. As the name suggests, this sends a stringified JSON message to a particular username.

```
function sendToOneUser(target, msgString) {
  var isUnique = true;
  var i;

  for (i=0; i<connectionArray.length; i++) {
    if (connectionArray[i].username === target) {
      connectionArray[i].send(msgString);
      break;
    }
  }
}
```

This function iterates over the list of connected users until it finds one matching the specified username, then sends the message to that user. The parameter `msgString` is a stringified JSON object. We could have made it receive our original message object, but in this example it's more efficient this way. Since the message has already been stringified, we can send it with no further processing. Each entry in `connectionArray` is a `WebSocket` object, so we can just call its `send()` method directly.

Our original chat demo didn't support sending messages to a specific user. The next task is to update the main WebSocket message handler to support doing so. This involves a change near the end of the `"connection"` message handler:

```
if (sendToClients) {
  var msgString = JSON.stringify(msg);
  var i;

  if (msg.target && msg.target !== undefined && msg.target.length !==
    sendToOneUser(msg.target, msgString);
  } else {
    for (i=0; i<connectionArray.length; i++) {
      connectionArray[i].send(msgString);
    }
  }
}
```

This code now looks at the pending message to see if it has a `target` property. If that property is present, it specifies the username of the client to which the message is to be sent, and we call `sendToOneUser()` to send the message to them. Otherwise, the message is broadcast to all users by iterating over the connection list, sending the message to each user.

As the existing code allows the sending of arbitrary message types, no additional changes are required. Our clients can now send messages of unknown types to any specific user, letting them send signaling messages back and forth as desired.

That's all we need to change on the server side of the equation. Now let's consider the signaling protocol we will implement.

Designing the signaling protocol

Now that we've built a mechanism for exchanging messages, we need a protocol defining how those messages will look. This can be done in a number of ways; what's demonstrated here is just one possible way to structure signaling messages.

This example's server uses stringified JSON objects to communicate with its clients. This means our signaling messages will be in JSON format, with contents which specify what kind of messages they are as well as any additional information needed in order to handle the messages properly.

Exchanging session descriptions

When starting the signaling process, an **offer** is created by the user initiating the call. This offer includes a session description, in SDP format, and needs to be delivered to the receiving user, which we'll call the **callee**. The callee responds to the offer with an **answer** message, also containing an SDP description. Our signaling server will use WebSocket to transmit offer messages with the type `"video-offer"`, and answer messages with the type `"video-answer"`. These messages have the following fields:

`type`
: The message type; either `"video-offer"` or `"video-answer"`.

`name`
: The sender's username.

`target`
: The username of the person to receive the description (if the caller is sending the message, this specifies the callee, and vice-versa).

`sdp`
: The SDP (Session Description Protocol) string describing the local end of the connection from the perspective of the sender (or the remote end of the connection from the receiver's point of view).

At this point, the two participants know which codecs and codec parameters are to be used for this call. They still don't know how to transmit the media data itself though. This is where Interactive Connectivity Establishment (ICE) comes in.

Exchanging ICE candidates

Two peers need to exchange ICE candidates to negotiate the actual connection between them. Every ICE candidate describes a method that the sending peer is able to use to communicate. Each peer sends candidates in the order they're discovered, and keeps sending candidates until it runs out of suggestions, even if media has already started streaming.

An `icecandidate` event is sent to the `RTCPeerConnection` to complete the process of adding a local description using `pc.setLocalDescription(offer)`.

Once the two peers agree upon a mutually-compatible candidate, that candidate's SDP is used by each peer to construct and open a connection, through which media then begins to flow. If they later agree on a better (usually higher-performance) candidate, the stream may change formats as needed.

Though not currently supported, a candidate received after media is already flowing could theoretically also be used to downgrade to a lower-bandwidth connection if needed.

Each ICE candidate is sent to the other peer by sending a JSON message of type `"new-ice-candidate"` over the signaling server to the remote peer. Each candidate message include these fields:

`type`
: The message type: `"new-ice-candidate"`.

`target`
: The username of the person with whom negotiation is underway; the server will direct the message to this user only.

`candidate`
: The SDP candidate string, describing the proposed connection method. You typically don't need to look at the contents of this string. All your code needs to do is route it through to the remote peer using the signaling server.

Each ICE message suggests a communication protocol (TCP or UDP), IP address, port number, connection type (for example, whether the specified IP is the peer itself or a relay server), along with other information needed to link the two computers together. This includes NAT or other networking complexity.

Note: The important thing to note is this: the only thing your code is responsible for during ICE negotiation is accepting outgoing candidates from the ICE layer and sending them across the signaling connection to the other peer when your `onicecandidate` handler is executed, and receiving ICE candidate messages from the signaling server (when the `"new-ice-candidate"` message is received) and delivering them to your ICE layer by calling `RTCPeerConnection.addIceCandidate()`. That's it.

The contents of the SDP are irrelevant to you in essentially all cases. Avoid the temptation to try to make it more complicated than that until you really know what you're doing. That way lies madness.

All your signaling server now needs to do is send the messages it's asked to. Your workflow may also demand login/authentication functionality, but such details will vary.

Note: The `onicecandidate` Event and `createAnswer()` Promise are both async calls which are handled separately. Be sure that your signaling does not change order! For example `addIceCandidate()` with the server's ice candidates must be called after setting the answer with `setRemoteDescription()`.

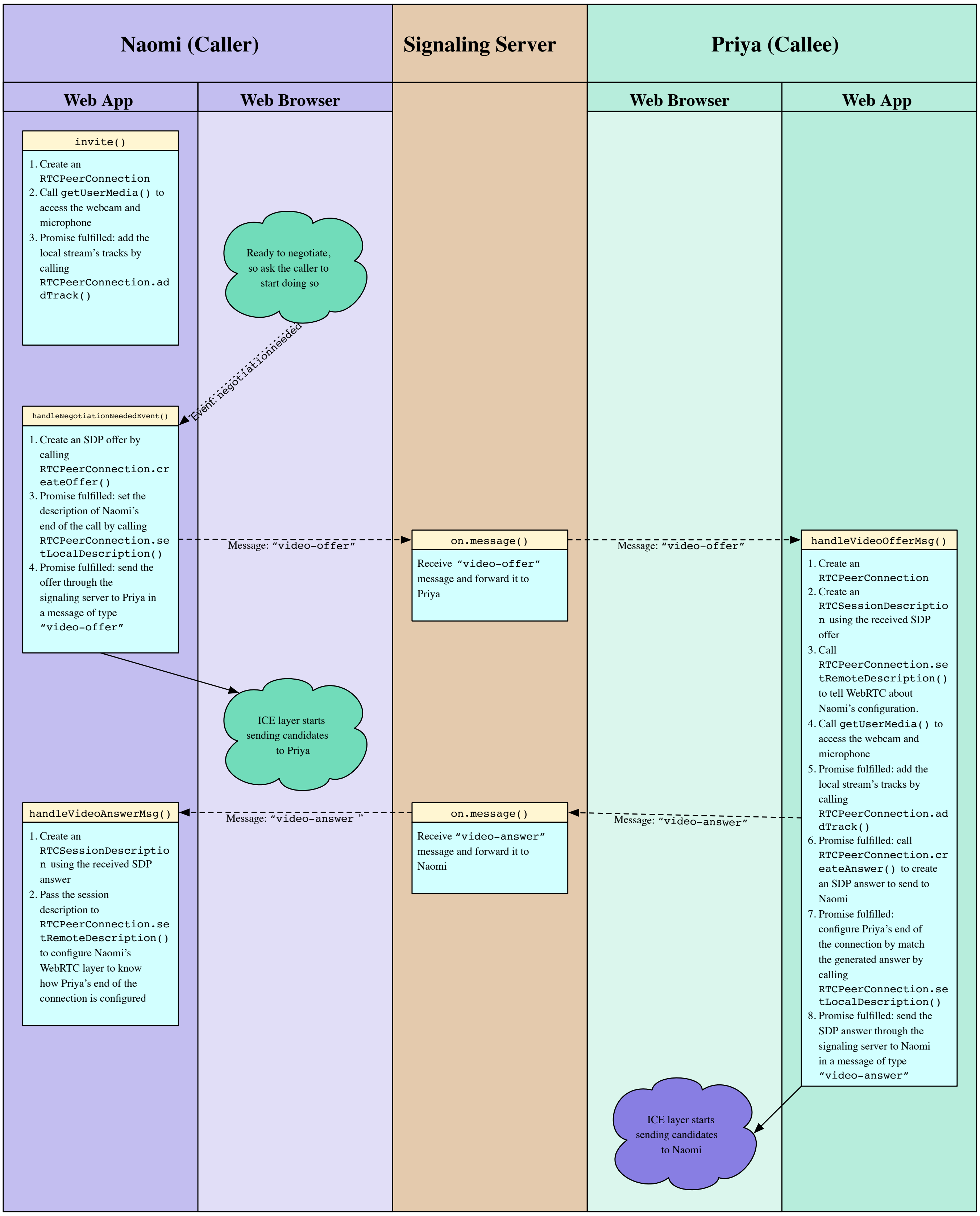
Signaling transaction flow

The signaling process involves this exchange of messages between two peers using an intermediary, the signaling server. The exact process will vary, of course, but in general there are a few key points at which signaling messages get handled:

The signaling process involves this exchange of messages among a number of points:

- Each user's client running within a web browser
- Each user's web browser
- The signaling server
- The web server hosting the chat service

Imagine that Naomi and Priya are engaged in a discussion using the chat software, and Naomi decides to open a video call between the two. Here's the expected sequence of events:

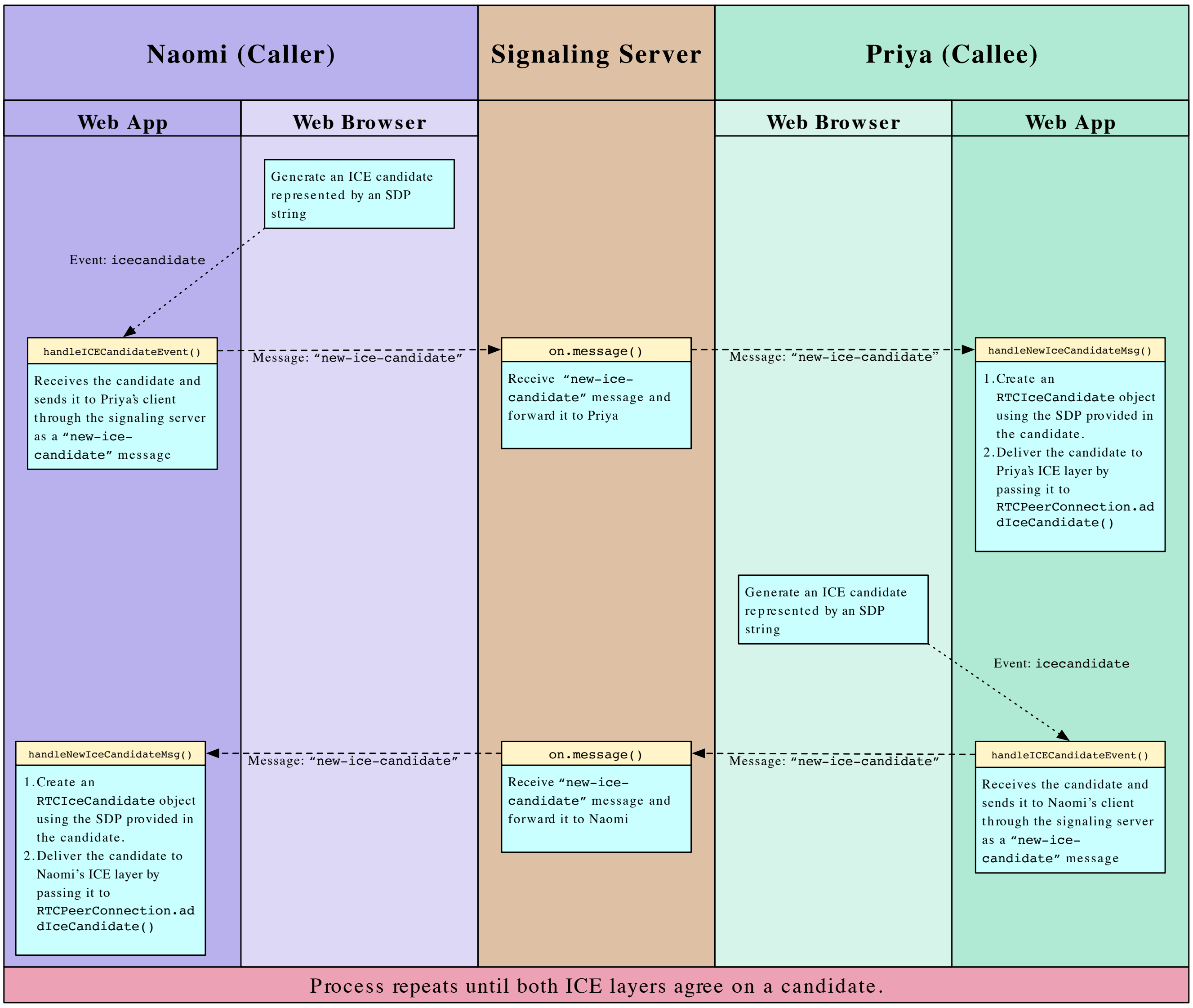


Naomi (Caller)
Signaling Server
Priya (Callee)
Web App
Web Browser
Web Browser
Web App
invite()
1. Create an RTCPeerConnection
2. Call getUserMedia() to access the webcam and microphone
3. Promise fulfilled: add the local stream's tracks by calling RTCPeerConnection.addTrack()
Ready to negotiate, so ask the caller to start doing so
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's tracks by calling RTCPeerConnection.addTrack()
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 RTCPeerConnection.setLocalDescription()
8. Promise fulfilled: send the SDP answer through the signaling server to Naomi in a message of type "video-answer"
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 WebRTC layer to know how Priya's end of the connection is configured
Message: "video-answer "
on.message()
Receive "video-answer" message and forward it to Naomi
Message: "video-answer"
ICE layer starts sending candidates to Naomi

We'll see this detailed more over the course of this article.

ICE candidate exchange process

When each peer's ICE layer begins to send candidates, it enters into an exchange among the various points in the chain that looks like this:


Naomi (Caller)
Signaling Server
Priya (Callee)
Web App
Web Browser
Web Browser
Web App
Generate an ICE candidate represented by an SDP string
Event: icecandidate
handleICECandidateEvent()
Receives the candidate and sends it to Priya's client through the signaling server as a "new-ice-candidate" message
Message: "new-ice-candidate"
on.message()
Receive "new-ice-candidate" message and forward it to Priya
Message: "new-ice-candidate"
handleNewIceCandidateMsg()
1. Create an RTCIceCandidate object using the SDP provided in the candidate.
2. Deliver the candidate to Priya's ICE layer by passing it to RTCPeerConnection.addIceCandidate()
Generate an ICE candidate represented by an SDP string
Event: icecandidate
handleICECandidateEvent()
Receives the candidate and sends it to Naomi's client through the signaling server as a "new-ice-candidate" message
Message: "new-ice-candidate"
on.message()
Receive "new-ice-candidate" message and forward it to Naomi
Message: "new-ice-candidate"
handleNewIceCandidateMsg()
1. Create an RTCIceCandidate object using the SDP provided in the candidate.
2. Deliver the candidate to Naomi's ICE layer by passing it to RTCPeerConnection.addIceCandidate()
Process repeats until both ICE layers agree on a candidate.

Each side sends candidates to the other as it receives them from their local ICE layer; there is no taking turns or batching of candidates. As soon as the two peers agree upon one candidate that they can both use to exchange the media, media begins to flow. Each peer continues to send candidates until it runs out of options, even after the media has already begun to flow. This is done in hopes of identifying even better options than the one initially selected.

If conditions change—for example the network connection deteriorates—one or both peers might suggest switching to a lower-bandwidth media resolution, or to an alternative codec. This triggers a new exchange of candidates, after which a another media format and/or codec change may take place. You can learn more about the codecs which WebRTC requires browsers to support, which additional codecs are supported by which browsers, and how to choose the best codecs to use in the guide Codecs used by WebRTC.

Optionally, see RFC 8445: Interactive Connectivity Establishment, section 2.3 ("Negotiating Candidate Pairs and Concluding ICE") if you want greater understanding of how this process is completed inside the ICE layer. You should note that candidates are exchanged and media starts to flow as soon as the ICE layer is satisfied. This is all taken care of behind the scenes. Our role is to simply send the candidates, back and forth, through the signaling server.

The client application

The core to any signaling process is its message handling. It's not necessary to use WebSockets for signaling, but it is a common solution. You should, of course, select a mechanism for exchanging signaling information that is appropriate for your application.

Let's update the chat client to support video calling.

Updating the HTML

The HTML for our client needs a location for video to be presented. This requires video elements, and a button to hang up the call:

```
<div class="flexChild" id="camera-container">
  <div class="camera-box">
    <video id="received_video" autoplay></video>
    <video id="local_video" autoplay muted></video>
    <button id="hangup-button" onclick="hangUpCall();" disabled>
      Hang Up
    </button>
  </div>
</div>
```

The page structure defined here is using `<div>` elements, giving us full control over the page layout by enabling the use of CSS. We'll skip layout detail in this guide, but take a look at the CSS on Github to see how we handled it. Take note of the two `<video>` elements, one for your self-view, one for the connection, and the `<button>` element.

The `<video>` element with the `id` "`received_video`" will present video received from the connected user. We specify the `autoplay` attribute, ensuring once the video starts arriving, it immediately plays. This removes any need to explicitly handle playback in our code. The "`local_video`" `<video>` element presents a preview of the user's camera; specifiying the `muted` attribute, as we don't need to hear local audio in this preview panel.

Finally, the "`hangup-button`" `<button>`, to disconnect from a call, is defined and configured to start disabled (setting this as our default for when no call is connected) and apply the function `hangUpCall()` on click. This function's role is to close the call, and send a signalling server notification to the other peer, requesting it also close.

The JavaScript code

We'll divide this code into functional areas to more easily describe how it works. The main body of this code is found in the `connect()` function: it opens up a `WebSocket` server on port 6503, and establishes a handler to receive messages in JSON object format. This code generally handles text chat messages as it did previously.

Sending messages to the signaling server

Throughout our code, we call `sendToServer()` in order to send messages to the signaling server. This function uses the WebSocket connection to do its work:

```
function sendToServer(msg) {
  var msgJSON = JSON.stringify(msg);

  connection.send(msgJSON);
}
```

The message object passed into this function is converted into a JSON string by calling `JSON.stringify()`, then we call the WebSocket connection's `send()` function to transmit the message to the server.

UI to start a call

The code which handles the `"userlist"` message calls `handleUserlistMsg()`. Here we set up the handler for each connected user in the user list displayed to the left of the chat panel. This function receives a message object whose `users` property is an array of strings specifying the user names of every connected user.

```
function handleUserlistMsg(msg) {
  var i;
  var listElem = document.querySelector(".userlistbox");

  while (listElem.firstChild) {
    listElem.removeChild(listElem.firstChild);
  }

  msg.users.forEach(function(username) {
    var item = document.createElement("li");
    item.appendChild(document.createTextNode(username));
    item.addEventListener("click", invite, false);

    listElem.appendChild(item);
  });
}
```

After getting a reference to the `<ul>` which contains the list of user names into the variable `listElem`, we empty the list by removing each of its child elements.

> **Note:** Obviously, it would be more efficient to update the list by adding and removing individual users instead of rebuilding the whole list every time it changes, but this is good enough for the purposes of this example.

Then we iterate over the array of user names using `forEach()`. For each name, we create a new `<li>` element, then create a new text node containing the user name using `createTextNode()`. That text node is added as a child of the `<li>` element. Next, we set a handler for the `click` event on the list item, that clicking on a user name calls our `invite()` method, which we'll look at in the next section.

Finally, we append the new item to the `<ul>` that contains all of the user names.

Starting a call

When the user clicks on a username they want to call, the `invite()` function is invoked as the event handler for that `click` event:

```
var mediaConstraints = {
  audio: true, // We want an audio track
  video: true // ...and we want a video track
};

function invite(evt) {
  if (myPeerConnection) {
    alert("You can't start a call because you already have one open!"
  } else {
    var clickedUsername = evt.target.textContent;

    if (clickedUsername === myUsername) {
      alert("I'm afraid I can't let you talk to yourself. That would
      return;
    }

    targetUsername = clickedUsername;
    createPeerConnection();

    navigator.mediaDevices.getUserMedia(mediaConstraints)
    .then(function(localStream) {
      document.getElementById("local_video").srcObject = localStream;
      localStream.getTracks().forEach(track => myPeerConnection.addTr
    })
    .catch(handleGetUserMediaError);
  }
}
```

This begins with a basic sanity check: is the user already connected? If there's already a `RTCPeerConnection`, they obviously can't make a call. Then the name of the user that was clicked upon is obtained from the event target's `textContent` property, and we check to be sure that it's not the same user that's trying to start the call.

Then we copy the name of the user we're calling into the variable `targetUsername` and call `createPeerConnection()`, a function which will create and do basic configuration of the `RTCPeerConnection`.

Once the `RTCPeerConnection` has been created, we request access to the user's camera and microphone by calling `MediaDevices.getUserMedia()`, which is exposed to us through the `Navigator.mediaDevices.getUserMedia` property. When this succeeds, fulfilling the returned promise, our `then` handler is executed. It receives, as input, a `MediaStream` object representing the stream with audio from the user's microphone and video from their webcam.

> **Note:** We could restrict the set of permitted media inputs to a specific device or set of devices by calling `navigator.mediaDevices.enumerateDevices()` to get a list of devices, filtering the resulting list based on our desired criteria, then using the selected devices' `deviceId` values in the `deviceId` field of the the `mediaConstraints` object passed into `getUserMedia()`. In practice, this is rarely if ever necessary, since most of that work is done for you by `getUserMedia()`.

We attach the incoming stream to the local preview `<video>` element by setting the element's `srcObject` property. Since the element is configured to automatically play incoming video, the stream begins playing in our local preview box.

We then iterate over the tracks in the stream, calling `addTrack()` to add each track to the `RTCPeerConnection`. Even though the connection is not fully established yet, you can begin sending data when you feel it's appropriate to do so. Media received before the ICE negotiation is completed may be used to help ICE decide upon the best connectivity approach to take, thus aiding in the negotiation process.

Note that for native apps, such as a phone application, you should not begin sending until the connection has been accepted at both ends, at a minimum, to avoid inadvertently sending video and/or audio data when the user isn't prepared for it.

As soon as media is attached to the `RTCPeerConnection`, a `negotiationneeded` event is triggered at the connection, so that ICE negotiation can be started.

If an error occurs while trying to get the local media stream, our catch clause calls `handleGetUserMediaError()`, which displays an appropriate error to the user as required.

Handling getUserMedia() errors

If the promise returned by `getUserMedia()` concludes in a failure, our `handleGetUserMediaError()` function performs.

```
function handleGetUserMediaError(e) {
  switch(e.name) {
    case "NotFoundError":
      alert("Unable to open your call because no camera and/or microp
            "were found.");
      break;
    case "SecurityError":
    case "PermissionDeniedError":
      // Do nothing; this is the same as the user canceling the call.
      break;
    default:
      alert("Error opening your camera and/or microphone: " + e.messa
      break;
  }

  closeVideoCall();
}
```

An error message is displayed in all cases but one. In this example, we ignore `"SecurityError"` and `"PermissionDeniedError"` results, treating refusal to grant permission to use the media hardware the same as the user canceling the call.

Regardless of why an attempt to get the stream fails, we call our `closeVideoCall()` function to shut down the `RTCPeerConnection`, and release any resources already allocated by the process of attempting the call. This code is designed to safely handle partially-started calls.

Creating the peer connection

The `createPeerConnection()` function is used by both the caller and the callee to construct their `RTCPeerConnection` objects, their respective ends of the WebRTC connection. It's invoked by `invite()` when the caller tries to start a call, and by `handleVideoOfferMsg()` when the callee receives an offer message from the caller.

```
function createPeerConnection() {
  myPeerConnection = new RTCPeerConnection({
      iceServers: [     // Information about ICE servers - Use your o
        {
          urls: "stun:stun.stunprotocol.org"
        }
      ]
  });

  myPeerConnection.onicecandidate = handleICECandidateEvent;
  myPeerConnection.ontrack = handleTrackEvent;
  myPeerConnection.onnegotiationneeded = handleNegotiationNeededEvent
  myPeerConnection.onremovetrack = handleRemoveTrackEvent;
  myPeerConnection.oniceconnectionstatechange = handleICEConnectionSt
  myPeerConnection.onicegatheringstatechange = handleICEGatheringStat
  myPeerConnection.onsignalingstatechange = handleSignalingStateChang
}
```

When using the `RTCPeerConnection()` constructor, we will specify an `RTCConfiguration`-compliant object providing configuration parameters for the connection. We use only one of these in this example: `iceServers`. This is an array of objects describing STUN and/or TURN servers for the ICE layer to use when attempting to establish a route between the caller and the callee. These servers are used to determine the best route and protocols to use when communicating between the peers, even if they're behind a firewall or using NAT.

> **Note:** You should always use STUN/TURN servers which you own, or which you have specific authorization to use. This example is using a known public STUN server but abusing these is bad form.

Each object in `iceServers` contains at least a `urls` field providing URLs at which the specified server can be reached. It may also provide `username` and `credential` values to allow authentication to take place, if needed.

After creating the `RTCPeerConnection`, we set up handlers for the events that matter to us.

The first three of these event handlers are required; you have to handle them to do anything involving streamed media with WebRTC. The rest aren't strictly required but can be useful, and we'll explore them. There are a few other events available that we're not using in this example, as well. Here's a summary of each of the event handlers we will be implementing:

`RTCPeerConnection.onicecandidate`

The local ICE layer calls your `icecandidate` event handler, when it needs you to transmit an ICE candidate to the other peer, through your signaling server. See Sending ICE candidates for more information and to see the code for this example.

`RTCPeerConnection.ontrack`

This handler for the `track` event is called by the local WebRTC layer when a track is added to the connection. This lets you connect the incoming media to an element to display it, for example. See Receiving new streams for details.

`RTCPeerConnection.onnegotiationneeded`

This function is called whenever the WebRTC infrastructure needs you to start the session negotiation process anew. Its job is to create and send an offer, to the callee, asking it to connect with us. See Starting negotiation to see how we handle this.

`RTCPeerConnection.onremovetrack`

This counterpart to `ontrack` is called to handle the `removetrack` event; it's sent to the `RTCPeerConnection` when the remote peer removes a track from the media being sent. See Handling the removal of tracks.

`RTCPeerConnection.oniceconnectionstatechange`

The `iceconnectionstatechange` event is sent by the ICE layer to let you know about changes to the state of the ICE connection. This can help you know when the connection has failed, or been lost. We'll look at the code for this example in ICE connection state below.

`RTCPeerConnection.onicegatheringstatechange`

The ICE layer sends you the `icegatheringstatechange` event, when the ICE agent's process of collecting candidates shifts, from one state to another (such as starting to gather candidates or completing negotiation). See ICE gathering state below.

`RTCPeerConnection.onsignalingstatechange`

The WebRTC infrastructure sends you the `signalingstatechange` message when the state of the signaling process changes (or if the connection to the signaling server changes). See Signaling state to see our code.

Starting negotiation

Once the caller has created its `RTCPeerConnection`, created a media stream, and added its tracks to the connection as shown in Starting a call, the browser will deliver a `negotiationneeded` event to the `RTCPeerConnection` to indicate that it's ready to begin negotiation with the other peer. Here's our code for handling the `negotiationneeded` event:

```
function handleNegotiationNeededEvent() {
  myPeerConnection.createOffer().then(function(offer) {
    return myPeerConnection.setLocalDescription(offer);
  })
  .then(function() {
    sendToServer({
      name: myUsername,
      target: targetUsername,
      type: "video-offer",
      sdp: myPeerConnection.localDescription
    });
  })
  .catch(reportError);
}
```

To start the negotiation process, we need to create and send an SDP offer to the peer we want to connect to. This offer includes a list of supported configurations for the connection, including information about the media stream we've added to the connection locally (that is, the video we want to send to the other end of the call), and any ICE candidates gathered by the ICE layer already. We create this offer by calling `myPeerConnection.createOffer()`.

When `createOffer()` succeeds (fulfilling the promise), we pass the created offer information into `myPeerConnection.setLocalDescription()`, which configures the connection and media configuration state for the caller's end of the connection.

> **Note:** Technically speaking, the string returned by `createOffer()` is an RFC 3264 offer.

We know the description is valid, and has been set, when the promise returned by `setLocalDescription()` is fulfilled. This is when we send our offer to the other peer by creating a new `"video-offer"` message containing the local description (now the same as the offer), then sending it through our signaling server to the callee. The offer has the following members:

`type`
: The message type: `"video-offer"`.

`name`
: The caller's username.

`target`
: The name of the user we wish to call.

`sdp`
: The SDP string describing the offer.

If an error occurs, either in the initial `createOffer()` or in any of the fulfillment handlers that follow, an error is reported by invoking our `reportError()` function.

Once `setLocalDescription()`'s fulfillment handler has run, the ICE agent begins sending `icecandidate` events to the `RTCPeerConnection`, one for each potential configuration it discovers. Our handler for the `icecandidate` event is responsible for transmitting the candidates to the other peer.

Session negotiation

Now that we've started negotiation with the other peer and have transmitted an offer, let's look at what happens on the callee's side of the connection for a while. The callee receives the offer and calls `handleVideoOfferMsg()` function to process it. Let's see how the callee handles the `"video-offer"` message.

Handling the invitation

When the offer arrives, the callee's `handleVideoOfferMsg()` function is called with the `"video-offer"` message that was received. This function needs to do two things. First, it needs to create its own `RTCPeerConnection` and add the tracks containing the audio and video from its microphone and webcam to that. Second, it needs to process the received offer, constructing and sending its answer.

```
function handleVideoOfferMsg(msg) {
  var localStream = null;

  targetUsername = msg.name;
  createPeerConnection();

  var desc = new RTCSessionDescription(msg.sdp);

  myPeerConnection.setRemoteDescription(desc).then(function () {
    return navigator.mediaDevices.getUserMedia(mediaConstraints);
  })
  .then(function(stream) {
    localStream = stream;
    document.getElementById("local_video").srcObject = localStream;

    localStream.getTracks().forEach(track => myPeerConnection.addTrac
  })
  .then(function() {
    return myPeerConnection.createAnswer();
  })
  .then(function(answer) {
    return myPeerConnection.setLocalDescription(answer);
  })
  .then(function() {
    var msg = {
      name: myUsername,
      target: targetUsername,
      type: "video-answer",
      sdp: myPeerConnection.localDescription
    };

    sendToServer(msg);
  })
  .catch(handleGetUserMediaError);
}
```

This code is very similar to what we did in the `invite()` function back in Starting a call. It starts by creating and configuring an `RTCPeerConnection` using our `createPeerConnection()` function. Then it takes the SDP offer from the received `"video-offer"` message and uses it to create a new `RTCSessionDescription` object representing the caller's session description.

That session description is then passed into `myPeerConnection.setRemoteDescription()`. This establishes the received offer as the description of the remote (caller's) end of the connection. If this is successful, the promise fulfillment handler (in the `then()` clause) starts the process of getting access to the callee's camera and microphone using `getUserMedia()`, adding the tracks to the connection, and so forth, as we saw previously in `invite()`.

Once the answer has been created using `myPeerConnection.createAnswer()`, the description of the local end of the connection is set to the answer's SDP by calling `myPeerConnection.setLocalDescription()`, then the answer is transmitted through the signaling server to the caller to let them know what the answer is

Any errors are caught and passed to `handleGetUserMediaError()`, described in Handling getUserMedia() errors.

Note: As is the case with the caller, once the `setLocalDescription()` fulfillment handler has run, the browser begins firing `icecandidate` events that the callee must handle, one for each candidate that needs to be transmitted to the remote peer.

Sending ICE candidates

The ICE negotiation process involves each peer sending candidates to the other, repeatedly, until it runs out of potential ways it can support the `RTCPeerConnection`'s media transport needs. Since ICE doesn't know about your signaling server, your code handles transmission of each candidate in your handler for the `icecandidate` event.

Your `onicecandidate` handler receives an event whose `candidate` property is the SDP describing the candidate (or is `null` to indicate that the ICE layer has run out of potential configurations to suggest). The contents of `candidate` are what you need to transmit using your signaling server. Here's our example's implementation:

```
function handleICECandidateEvent(event) {
  if (event.candidate) {
    sendToServer({
      type: "new-ice-candidate",
      target: targetUsername,
      candidate: event.candidate
    });
  }
}
```

This builds an object containing the candidate, then sends it to the other peer using the `sendToServer()` function previously described in Sending messages to the signaling server. The message's properties are:

`type`
The message type: `"new-ice-candidate"`.

`target`
The username the ICE candidate needs to be delivered to. This lets the signaling server route the message.

`candidate`
The SDP representing the candidate the ICE layer wants to transmit to the other peer.

The format of this message (as is the case with everything you do when handling signaling) is entirely up to you, depending on your needs; you can provide other information as required.

> **Note:** It's important to keep in mind that the `icecandidate` event is **not** sent when ICE candidates arrive from the other end of the call. Instead, they're sent by your own end of the call so that you can take on the job of transmitting the data over whatever channel you choose. This can be confusing when you're new to WebRTC.

Receiving ICE candidates

The signaling server delivers each ICE candidate to the destination peer using whatever method it chooses; in our example this is as JSON objects, with a `type` property containing the string `"new-ice-candidate"`. Our `handleNewICECandidateMsg()` function is called by our main WebSocket incoming message code to handle these messages:

```
function handleNewICECandidateMsg(msg) {
  var candidate = new RTCIceCandidate(msg.candidate);

  myPeerConnection.addIceCandidate(candidate)
    .catch(reportError);
}
```

This function constructs an `RTCIceCandidate` object by passing the received SDP into its constructor, then delivers the candidate to the ICE layer by passing it into `myPeerConnection.addIceCandidate()`. This hands the fresh ICE candidate to the local ICE layer, and finally, our role in the process of handling this candidate is complete.

Each peer sends to the other peer a candidate for each possible transport configuration that it believes might be viable for the media being exchanged. At some point, the two peers agree that a given candidate is a good choice and they open the connection and begin to share media. It's important to note, however, that ICE negotiation does *not* stop once media is flowing. Instead, candidates may still keep being exchanged after the conversation has begun, either while trying to find a better connection method, or simply because they were already in transport when the peers successfully established their connection.

In addition, if something happens to cause a change in the streaming scenario, negotiation will begin again, with the `negotiationneeded` event being sent to the `RTCPeerConnection`, and the entire process starts again as described before. This can happen in a variety of situations, including:

- Changes in the network status, such as a bandwidth change, transitioning from WiFi to cellular connectivity, or the like.
- Switching between the front and rear cameras on a phone.
- A change to the configuration of the stream, such as its resolution or frame rate.

Receiving new streams

When new tracks are added to the `RTCPeerConnection`— either by calling its `addTrack()` method or because of renegotiation of the stream's format—a `track` event is set to the `RTCPeerConnection` for each track added to the connection. Making use of newly added media requires implementing a handler for the `track` event. A common need is to attach the incoming media to an appropriate HTML element. In our example, we add the track's stream to the `<video>` element that displays the incoming video:

```
function handleTrackEvent(event) {
  document.getElementById("received_video").srcObject = event.streams
  document.getElementById("hangup-button").disabled = false;
}
```

The incoming stream is attached to the `"received_video"` `<video>` element, and the "Hang Up" `<button>` element is enabled so the user can hang up the call.

Once this code has completed, finally the video being sent by the other peer is displayed in the local browser window!

Handling the removal of tracks

Your code receives a `removetrack` event when the remote peer removes a track from the connection by calling `RTCPeerConnection.removeTrack()`. Our handler for `"removetrack"` is:

```
function handleRemoveTrackEvent(event) {
  var stream = document.getElementById("received_video").srcObject;
  var trackList = stream.getTracks();

  if (trackList.length == 0) {
    closeVideoCall();
  }
}
```

This code fetches the incoming video `MediaStream` from the `"received_video"` `<video>` element's `srcobject` attribute, then calls the stream's `getTracks()` method to get an array of the stream's tracks.

If the array's length is zero, meaning there are no tracks left in the stream, we end the call by calling `closeVideoCall()`. This cleanly restores our app to a state in which it's ready to start or receive another call. See Ending the call to learn how `closeVideoCall()` works.

Ending the call

There are many reasons why calls may end. A call might have completed, with one or both sides having hung up. Perhaps a network failure has occurred, or one user might have quit their browser, or had a system crash. In any case, all good things must come to an end.

Hanging up

When the user clicks the "Hang Up" button to end the call, the `hangUpCall()` function is called:

```
function hangUpCall() {
  closeVideoCall();
  sendToServer({
    name: myUsername,
    target: targetUsername,
    type: "hang-up"
  });
}
```

`hangUpCall()` executes `closeVideoCall()` to shut down and reset the connection and release resources. It then builds a `"hang-up"` message and sends it to the other end of the call to tell the other peer to neatly shut itself down.

Ending the call

The `closeVideoCall()` function, shown below, is responsible for stopping the streams, cleaning up, and disposing of the `RTCPeerConnection` object:

```
function closeVideoCall() {
  var remoteVideo = document.getElementById("received_video");
  var localVideo = document.getElementById("local_video");

  if (myPeerConnection) {
    myPeerConnection.ontrack = null;
    myPeerConnection.onremovetrack = null;
    myPeerConnection.onremovestream = null;
    myPeerConnection.onicecandidate = null;
    myPeerConnection.oniceconnectionstatechange = null;
    myPeerConnection.onsignalingstatechange = null;
    myPeerConnection.onicegatheringstatechange = null;
    myPeerConnection.onnegotiationneeded = null;

    if (remoteVideo.srcObject) {
      remoteVideo.srcObject.getTracks().forEach(track => track.stop()
    }

    if (localVideo.srcObject) {
      localVideo.srcObject.getTracks().forEach(track => track.stop())
    }

    myPeerConnection.close();
    myPeerConnection = null;
  }

  remoteVideo.removeAttribute("src");
  remoteVideo.removeAttribute("srcObject");
  localVideo.removeAttribute("src");
  remoteVideo.removeAttribute("srcObject");

  document.getElementById("hangup-button").disabled = true;
  targetUsername = null;
}
```

After pulling references to the two `<video>` elements, we check if a WebRTC connection exists; if it does, we proceed to disconnect and close the call:

1. All of the event handlers are removed. This prevents stray event handlers from being triggered while the connection is in the process of closing, potentially causing errors.
2. For both remote and local video streams, we iterate over each track, calling the `MediaStreamTrack.stop()` method to close each one.
3. Close the `RTCPeerConnection` by calling `myPeerConnection.close()`.
4. Set `myPeerConnection` to `null`, ensuring our code learns there's no ongoing call; this is useful when the user clicks a name in the user list.

Then for both the incoming and outgoing `<video>` elements, we remove their `src` and `srcobject` attributes using their `removeAttribute()` methods. This completes the disassociation of the streams from the video elements.

Finally, we set the `disabled` property to `true` on the "Hang Up" button, making it unclickable while there is no call underway; then we set `targetUsername` to `null` since we're no longer talking to anyone. This allows the user to call another user, or to receive an incoming call.

Dealing with state changes

There are a number of additional events you can set listeners for which notifying your code of a variety of state changes. We use three of them: `iceconnectionstatechange`, `icegatheringstatechange`, and `signalingstatechange`.

ICE connection state

`iceconnectionstatechange` events are sent to the `RTCPeerConnection` by the ICE layer when the connection state changes (such as when the call is terminated from the other end).

```
function handleICEConnectionStateChangeEvent(event) {
  switch(myPeerConnection.iceConnectionState) {
    case "closed":
    case "failed":
      closeVideoCall();
      break;
  }
}
```

Here, we apply our `closeVideoCall()` function when the ICE connection state changes to `"closed"` or `"failed"`. This handles shutting down our end of the connection so that we're ready start or accept a call once again.

> **Note:** We don't watch the `disconnected` signaling state here as it can indicate temporary issues and may go back to a `connected` state after some time. Watching it would close the video call on any temporary network issue.

ICE signaling state

Similarly, we watch for `signalingstatechange` events. If the signaling state changes to `closed`, we likewise close the call out.

```
function handleSignalingStateChangeEvent(event) {
  switch(myPeerConnection.signalingState) {
    case "closed":
      closeVideoCall();
      break;
  }
};
```

> **Note:** The `closed` signaling state has been deprecated in favor of the `closed` `iceConnectionState`. We are watching for it here to add a bit of backward compatibility.

ICE gathering state

`icegatheringstatechange` events are used to let you know when the ICE candidate gathering process state changes. Our example doesn't use this for anything, but it can be useful to watch these events for debugging purposes, as well as to detect when candidate collection has finished.

```
function handleICEGatheringStateChangeEvent(event) {
  // Our sample just logs information to console here,
  // but you can do whatever you need.
}
```

Next steps

You can now try out this example on Glitch to see it in action. Open the Web console on both devices and look at the logged output—although you don't see it in the code as shown above, the code on the server (and on GitHub) has a lot of console output so you can see the signaling and connection processes at work.

Another obvious improvement would be to add a "ringing" feature, so that instead of just asking the user for permission to use the camera and microphone, a "User X is calling. Would you like to answer?" prompt appears first.

See also

- WebRTC API
- Web media technologies
- Guide to media types and formats on the web
- Media Capture and Streams API
- Media Capabilities API
- MediaStream Recording API
- The Perfect Negotiation pattern

Last modified: Jul 17, 2020, by MDN contributors

Related Topics

WebRTC API

▼ WebRTC Guides

- WebRTC Architecture
- WebRTC Basics
- WebRTC Protocols
- Dealing with connectivity
- Overview of WebRTC interfaces
- Lifetime of a WebRTC Session
- Using data channels

▼ WebRTC Tutorials

- Interoperability with adapter.js
- Taking still photos from the camera
- A simple data channel example

▼ Interfaces

- `RTCPeerConnection`
- `RTCSessionDescription`
- `RTCIceCandidate`
- `RTCPeerConnectionIceEvent`
- `MessageEvent`
- `MediaStream`
- `RTCStatsReport`
- `RTCIdentityEvent`
- `RTCIdentityErrorEvent`
- `MediaStreamEvent`
- `MediaStreamTrack`
- `MediaDevices`