codelab
webrtc

complete			
README.md	14.1 KB	52 minutes ago	last minute tweaks

WebRTC codelab

Overview

WebRTC enables real-time communication in the browser.

This tutorial explains how to build a simple video and text chat application.

For more information about WebRTC, see Getting started with WebRTC on HTML5 Rocks.

Prerequisites

Basic knowledge:

1. HTML, CSS and JavaScript
2. git
3. Chrome DevTools

Experience of Node.js and socket.io would also be useful

Installed on your development machine:

1. Google Chrome
2. Code editor
3. Web server such as MAMP or XAMPP -- or just run `python -m SimpleHTTPServer` in your application directory
4. Web cam
5. git, in order to get the source code
6. The source code
7. Node.js with socket.io and node-static. (Node.js hosting would also be an advantage -- see below for some options.)

It would also be useful to have an Android device with Google Chrome Beta installed in order to try out the examples on mobile. To run WebRTC APIs on Chrome for Android, you must enable WebRTC from the chrome://flags page.

Step 0: Get the code

Using git, clone the codelab repository onto your development computer. If you haven't used git before, there are several tutorials and reference guides available from the git website.

Step 1: Create a blank HTML5 document

Complete example: complete/step1.html.

1. Create a bare-bones HTML document.
2. Run that from localhost (see server suggestions above).

Step 2: Get video from your webcam

Complete example: complete/step2.html.

1. Add a video element to your page.
2. Add the following JavaScript to the script element on your page, to enable getUserMedia() to set the source of the video from the web cam:

```
navigator.getUserMedia = navigator.getUserMedia ||
  navigator.webkitGetUserMedia || navigator.mozGetUserMedia;
```

```
var constraints = {video: true};

function successCallback(localMediaStream) {
  window.stream = localMediaStream; // stream available to console
  var video = document.querySelector("video");
  video.src = window.URL.createObjectURL(localMediaStream);
  video.play();
}

function errorCallback(error){
  console.log("navigator.getUserMedia error: ", error);
}

navigator.getUserMedia(constraints, successCallback, errorCallback);
```

3. View your page from *localhost*.

Explanation

`getUserMedia` is called like this:

```
navigator.getUserMedia(constraints, successCallback, errorCallback);
```

The constraints argument allows us to specify the media to get, in this case video only:

```
var constraints = {"video": true}
```

If successful, the video stream from the webcam is set as the source of the video element:

```
function successCallback(localMediaStream) {
  window.stream = localMediaStream; // stream available to console
  var video = document.querySelector("video");
  video.src = window.URL.createObjectURL(localMediaStream);
  video.play();
}
```

Bonus points

1. Inspect the stream object from the console.
2. Try calling `stream.stop()`.
3. What does `stream.getVideoTracks()` return?
4. Look at the constraints object: what happens when you change it to `{audio: true, video: true}`?
5. What size is the video element? How can you get the video's natural size from JavaScript? Use the Chrome Dev Tools to check. Use CSS to make the video full width. How would you ensure the video is no higher than the viewport?
6. Try adding CSS filters to the video element (more ideas here).
7. Try changing constraints: see the sample at simpl.info/getusermedia/constraints.

For example:

```
video {
  filter: hue-rotate(180deg) saturate(200%);
  -moz-filter: hue-rotate(180deg) saturate(200%);
  -webkit-filter: hue-rotate(180deg) saturate(200%);
}
```

Step 3: Stream video with RTCPeerConnection

Complete example: complete/step3.html.

RTCPeerConnection is the WebRTC API for video and audio calling.

This example sets up a connection between two peers on the same page. Not much use, but good for understanding how RTCPeerConnection works!

1. Get rid of the JavaScript you've entered so far -- we're going to do something different!
2. Edit the HTML so there are two video elements and three buttons: Start, Call and Hang Up:

```
<video id="vid1" autoplay></video>
<video id="vid2" autoplay></video>

<div>
  <button id="startButton">Start</button>
  <button id="callButton">Call</button>
  <button id="hangupButton">Hang Up</button>
</div>
```

3. Add the JavaScript from complete/step3.html.

Explanation

This code does a lot!

- Get and share local and remote descriptions: metadata (in SDP format) of local media conditions.
- Get and share ICE candidates: network information.
- Pass the local stream to the remote *RTCPeerConnection*.

Bonus points

1. Take a look at *chrome://webrtc-internals*. (There is a full list of Chrome URLs at *chrome://about*.)
2. Style the page with CSS:
    - Put the videos side by side.
    - Make the buttons the same width, with bigger text.
    - Make sure it works on mobile.
3. From the Chrome Dev Tools console, inspect *localStream*, *localPeerConnection* and *remotePeerConnection*.
4. Take a look at *localPeerConnection.localDescription*. What does SDP format look like?

Step 4: Stream arbitrary data with RTCDataChannel

Complete example: complete/step4.html.

For this step, we'll use RTCDataChannel to send text between two textareas on the same page: not very useful, except to demonstrate how the API works.

1. Create a new document and add the following HTML:

```
<textarea id="dataChannelSend" disabled></textarea>
<textarea id="dataChannelReceive" disabled></textarea>

<div id="buttons">
  <button id="startButton">Start</button>
  <button id="sendButton">Send</button>
  <button id="closeButton">Stop</button>
</div>
```

2. Add the JavaScript from complete/step4.html.

Explanation

This code uses RTCPeerConnection to enable exchange of text messages.

A lot of the code is the same as for the RTCPeerConnection example. Additional code is as follows:

```
function sendData(){
  var data = document.getElementById("dataChannelSend").value;
  sendChannel.send(data);
}
...
localPeerConnection = new webkitRTCPeerConnection(servers,
  {optional: [{RtpDataChannels: true}]});
sendChannel = localPeerConnection.createDataChannel("sendDataChannel",
  {reliable: false});
sendChannel.onopen = handleSendChannelStateChange;
sendChannel.onclose = handleSendChannelStateChange;
...
remotePeerConnection = new webkitRTCPeerConnection(servers,
  {optional: [{RtpDataChannels: true}]});
function gotReceiveChannel(event) {
  receiveChannel = event.channel;
  receiveChannel.onmessage = gotMessage;
}
...
remotePeerConnection.ondatachannel = gotReceiveChannel;
function gotMessage(event) {
  document.getElementById("dataChannelReceive").value = event.data;
}
```

Notice the use of constraints.

Bonus points

1. Try out RTCDataChannel file sharing with Sharefest. When would RTCDataChannel need to be reliable, and when might performance be more important?
2. Use CSS to improve page layout, and add a placeholder attribute to the *dataChannelReceive* textarea.

3. Test the page on a mobile device.

Step 5: Set up a signalling server and exchange messages

Complete example: complete/step5.

In the examples already completed, signalling between RTCPeerconnection objects happens on the same page: the process of exchanging candidate information and offer/answer messages.

In the real world, a server is required to enable signalling between WebRTC clients on different devices.

In this step we'll build a simple Node.js messaging server, using socket.io Node module and JavaScript library for messaging. Experience of Node.js and socket.io will be useful, but not crucial. In this example, the server is *server.js* and the client is *index.html*.

The server application in this step has two tasks.

To act as a messaging intermediary:

```
socket.on('message', function (message) {
  log('Got message: ', message);
  socket.broadcast.emit('message', message); // should be room only
});
```

To manage 'rooms':

```
if (numClients == 0){
  socket.join(room);
  socket.emit('created', room);
} else if (numClients == 1) {
  io.sockets.in(room).emit('join', room);
  socket.join(room);
  socket.emit('joined', room);
} else { // max two clients
  socket.emit('full', room);
}
```

Our simple WebRTC application will only permit a maximum of two peers to share a room.

1. Ensure you have Node, socket.io and node-static installed.
2. Using the code from the *step 5* directory, run the server (*server.js*). To start the server, from your application directory run the following:

   ```
   node server.js
   ```

3. From your browser, open *localhost:2013*. Open a new tab page or window in any browser and open *localhost:2013* again, then repeat.
4. To see what's happening, check the Chrome DevTools console (Command-Option-J, or Ctrl-Shift-J).

Bonus points

1. Try deploying your messaging server so you can access it via a public URL. (Free trials and easy deployment options for Node are available on several hosting sites including nodejitsu, heroku and nodester.)
2. What alternative messaging mechanisms are available? (Take a look at apprtc.appspot.com.) What problems might we encounter using 'pure' WebSocket? (Take a look at Arnout Kazemier's presentation, WebSuckets.)
3. What issues might be involved with scaling this application? Can to develop a method for testing thousands or millions of simultaneous room requests.
4. Try out Remy Sharp's tool nodemon. This monitors any changes in your Node.js application and automatically restarts the server when changes are saved.
5. This app uses a JavaScript prompt to get a room name. Work out a way to get the room name from the URL, for example *localhost:2013/foo* would give the room name *foo*.

Step 6: RTCPeerConnection with messaging

Complete example: complete/step6.

In this step, we build a video chat client, using the signalling server we created in Step 5 and the RTCPeerConnection code from Step 3.

1. Ensure you have Node, socket.io and node-static installed and working. If in doubt, try the code in Step 5.
2. Using the code from the *step 6* directory, run the server (*server.js*). To start the server, from your application directory run the following:

   ```
   node server.js
   ```

3. From your browser, open *localhost:2013*. Open a new tab page or window in any browser and open *localhost:2013*.
4. View logging from the Chrome DevTools console and WebRTC debug information from chrome://webrtc-internals.

Bonus points

1. This application only supports one-to-one video chat. How might you change the design to enable more than one person to share the same video chat room? (Look at conversat.io for an example of this in action.)
2. The example has the room name *foo* hard coded. What would be the best way to enable other room names?
3. Does the app work on mobile? Try it out on a phone, on a 7" and a 10" tablet. What layout, UI and UX changes would be required to ensure a good mobile experience?
4. Deploy your app at a public URL (see above for hosting options). Try different network configurations, for example with one user on wifi and another on 3G. Any problems?
5. How would users share the room name? Try to build an alternative to sharing room names.

Step 7: Putting it all together: RTCPeerConnection + RTCDataChannel + signalling

This is a DIY step!

1. Take a look at the app you built in step 4.
2. Add the RTCDataChannel code to your Step 6 app to create a complete application.

Bonus points

1. The app hasn't had any work done on layout. Sort it out! Make sure your app works well on different devices.

Step 8: Use a WebRTC library: SimpleWebRTC

Complete example: complete/step8.html.

Abstraction libraries such as SimpleWebRTC make it simple to create WebRTC applications.

1. Create a new document using the code from complete/step8.html.
2. Open the document in multiple windows or tab.

Bonus points

1. Find a WebRTC library for RTCDataChannel. (Hint: there's one named PeerJS!)
2. Set up your own signalling server using the SimpleWebRTC server signalmaster.
