8

Streaming Binary Data

In this chapter, we will cover the following recipes:

* Broadcasting an Image to Other Sockets
* Uploading an Image to the File System
* Uploading an Image to Amazon S3
* Streaming Audio
* Streaming Live Video

# Introduction

Socket.IO 1.0 gives us the ability to stream binary data between the server and the client. In this chapter, we will use that ability to transport various forms of binary data such as images, audio and video.

# Broadcasting an Image to Other Sockets

Typically, the src attribute for an HTML image tag will be a link to the location of the image. However, instead of a link to the image, we can actually provide the binary data for the image itself. That ability allows us to store and load actual images and not just the link to the image location.

We can actually use Socket.IO to send images from a browser to the server and then display them in another browser without ever storing them on a server or in a file system or database of any kind. In instances where we don’t need the data to be persisted, this can be really useful.

In this recipe, we will demonstrate how we can pipe an image from our file system to the browser over Web Sockets.

## Getting Ready...

In this recipe, I am using a static image called “woodchuck.jpg” to pipe into the browser. It is located at the root of the app along with the server.js and index.html files. You should be able to put any image that you want to use in that location as long as you reference the correct image in you server code. As new chunks of the image are read, they will be emitted to the browser using the “image-chunk” event.

## How To Do It...

To broadcast images over Socket.IO, follow these steps:

1. First, we will create our server.js file. This will wait for the socket connection event and immediately begin to read the image by creating a read stream using the built in fs module that comes with Node.

var express = require('express'),

app = express(),

http = require('http'),

socketIO = require('socket.io'),

fs = require('fs'),

path = require('path'),

server, io;

app.get('/', function (req, res) {

res.sendFile(\_\_dirname + '/index.html');

});

server = http.Server(app);

server.listen(5000);

io = socketIO(server);

io.on('connection', function (socket) {

var readStream = fs.createReadStream(path.resolve(\_\_dirname, './woodchuck.jpg'), {

encoding: 'binary'

}), chunks = [];

readStream.on('readable', function () {

console.log('Image loading');

});

readStream.on('data', function (chunk) {

chunks.push(chunk);

socket.emit('img-chunk', chunk);

});

readStream.on('end', function () {

console.log('Image loaded');

});

});

1. Now we need to create an index.html page to render the image in pieces as data is chunked in.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title></title>

</head>

<body>

<img id="img-stream2" src="" />

<script src="/socket.io/socket.io.js"></script>

<script type="text/javascript">

var socket = io.connect('http://localhost:5000');

var imgChunks = [];

socket.on('img-chunk', function (chunk) {

var img = document.getElementById('img-stream2');

imgChunks.push(chunk);

img.setAttribute('src', 'data:image/jpeg;base64,' + window.btoa(imgChunks));

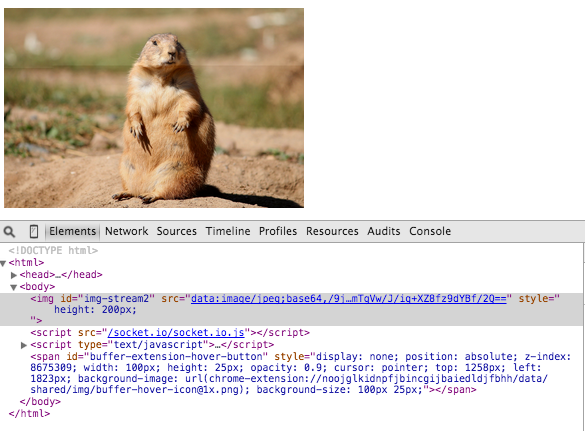
});

</script>

</body>

</html>

1. Start your server and go to localhost:5000, the image will be rendered and if you view the element in your developer console, you should see that it is prefixed with “data:image/jpeg;base64” followed by a base64-encoded string.



Insert Image B04893\_07\_01.png

## How It Works...

When a user connects to our server, we immediately begin to read the contents of the image file using a read stream. This allows us to emit parts of the image as they are read instead of emitting it all at once, which means the image will load progressively as the data is chunked.

## There’s More...

If you are on a fast connection, you may not notice that the image is loading in pieces. The best way to demonstrate that it is loading progressively is to set a timeout to delay each part of the image from loading for a set amount of time, like this:

var delay = 0;

readStream.on('data', function (chunk) {

chunks.push(chunk);

delay = delay + 5000;

setTimeout(function () {

socket.emit('img-chunk', chunk);

}, delay);

});

# Uploading An Image To The File System

With Socket.IO, we can send files to our server over Web Sockets instead of using an http POST request. This allows us to upload files in real-time and display the uploaded images as needed.

In this recipe, we will use Socket.IO to upload a file to our local file system and then send a message to the client to display the image when it is done uploading.

## Getting Ready...

For this recipe, we will be using the built-in Node fs module to upload our images.

We will be uploading files to our file system, so make sure to create a folder called “tmp” in the root of your project.

## How To Do It...

To upload an image to the file system with Socket.IO, follow these steps:

1. First, we will need to create our server.js file. This will be responsible for listening for new “upload-image” messages and uploading the file that is passed with the arguments.

var express = require('express'),

app = express(),

http = require('http'),

socketIO = require('socket.io'),

fs = require('fs'),

path = require('path'),

server, io;

app.use(express.static(\_\_dirname));

server = http.Server(app);

server.listen(5000);

console.log('Listening on port 5000');

io = socketIO(server);

io.on('connection', function (socket) {

socket.on('upload-image', function (message) {

var writer = fs.createWriteStream(path.resolve(\_\_dirname, './tmp/' + message.name), {

encoding: 'base64'

});

writer.write(message.data);

writer.end();

writer.on('finish', function () {

socket.emit('image-uploaded', {

name: '/tmp/' + message.name

});

});

});

});

1. Finally, we will need to create an index.html template for our client-side. We will add a “change” event to listen for an item to be selected in the input field and read the file and emit it when the event is triggered.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title></title>

</head>

<body>

<hr />

<input type="file" id="my-file" />

<hr />

<script src="/socket.io/socket.io.js"></script>

<script type="text/javascript">

var socket = io.connect('http://localhost:5000');

var file = document.getElementById('my-file');

file.addEventListener('change', function () {

if (!file.files.length) {

return;

}

var firstFile = file.files[0],

reader = new FileReader();

reader.onloadend = function () {

socket.emit('upload-image', {

name: firstFile.name,

data: reader.result

});

};

reader.readAsArrayBuffer(firstFile);

});

socket.on('image-uploaded', function (message) {

var img = document.createElement('img');

img.setAttribute('src', message.name);

img.setAttribute('height', '100px');

document.body.appendChild(img);

});

</script>

</body>

</html>

1. Now, if you start your server and go to localhost:5000/index.html, you should see an input field to upload a file to the server, just choose a file and it will upload to you /tmp directory and display below the file input.  
     
   

Insert Image B04893\_07\_02.png

## How It Works...

Socket.IO can pass any kind of data including binary file data. Our client-side JavaScript allows us to access the file using the FileReader() api. We can pass the data we extract from the FileReader to the server and let the server write the file. When the file is added to the file system, the server emits a message to let us know that the upload is complete and at that point we can display the newly uploaded file on the client.

# Uploading an Image to Amazon S3

Uploading images to your server-side file system is actually not a terribly good idea. If you are deploying from your repo, you don’t want to mix in user-generated media with your code. A much better and more scalable approach is to put your photos and other media in a completely separate static location, such as Amazon S3, where you can access them without letting them interfere with your core application.

In this recipe, we will upload images to Amazon S3 and display them after they are uploaded.

## Getting Ready...

For this recipe, we will be using the Amazon SDK for Node. It can be installed by running npm install aws-sdk –save in your terminal. We will also be using lodash (npm install lodash --save) and the q promise library (npm install q --save).

## How To Do It...

To upload an image to Amazon S3, follow these steps:

1. First, we will create a aws.service.js file. This will be responsible for interfacing with the Amazon AWS SDK and writing and reading binary data from Amazon. We will be using some environmental variables that we will need to set by exporting them in our terminal. For example: export AWS\_ACCESS\_KEY\_ID=”AKGAJ2PTGPBP3GAIPZ7G”. The required environmental variables are AWS\_ACCESS\_KEY\_ID, which is the key to your Amazon account, AWS\_SECRET\_ACCESS\_KEY, which is the secret key for you Amazon account, AWS\_BUCKET\_NAME, which is the name of the bucket you have set up on Amazon, which you want to write to and AWS\_BUCKET\_PATH, which is the path that you want your files to be written to.

var AWS = require('aws-sdk'),

\_ = require('lodash'),

q = require('q');

var service = {}, s3;

var AWS\_ACCESS\_KEY\_ID = process.env.AWS\_ACCESS\_KEY\_ID,

AWS\_SECRET\_ACCESS\_KEY = process.env.AWS\_SECRET\_ACCESS\_KEY,

AWS\_BUCKET\_NAME = process.env.AWS\_BUCKET\_NAME,

AWS\_BUCKET\_PATH = process.env.AWS\_BUCKET\_PATH;

AWS.config.update({

accessKeyId: AWS\_ACCESS\_KEY\_ID,

secretAccessKey: AWS\_SECRET\_ACCESS\_KEY

});

s3 = new AWS.S3();

function write (path, file) {

var deffered = q.defer();

s3.putObject({

Bucket: AWS\_BUCKET\_NAME,

Key: AWS\_BUCKET\_PATH + '/' + path,

Body: file

}, function (err, data) {

deffered.resolve(data || err);

});

return deffered.promise;

}

function readFile (path) {

var deffered = q.defer();

path = path.replace(AWS\_BUCKET\_PATH + '/', '');

s3.getObject({

Bucket: AWS\_BUCKET\_NAME,

Key: AWS\_BUCKET\_PATH + '/' + path

}, function (err, data) {

if (!data) {

deffered.resolve(err);

} else {

deffered.resolve(\_.extend(data, {

path: path

}));

}

});

return deffered.promise;

}

function read (path) {

var deffered = q.defer();

readFile(path).then(function (data) {

if (data.Body) {

var buf = new Buffer(data.Body);

deffered.resolve(buf.toString());

} else {

deffered.resolve(null);

}

});

return deffered.promise;

}

module.exports = {

write: write,

read: read,

readFile: readFile

};

1. The server.js file will be responsible for listening for Socket.IO events and uploading the images as they come in via the AWS service that we created. After each file is uploaded, we will read it as a base64-encoded string and emit the result back to the client so the client can display the image.

var express = require('express'),

app = express(),

http = require('http'),

socketIO = require('socket.io'),

fs = require('fs'),

path = require('path'),

aws = require('./aws.service'),

server, io;

app.use(express.static(\_\_dirname));

app.get('/', function (req, res) {

res.sendFile(\_\_dirname + '/index.html');

});

server = http.Server(app);

server.listen(5000);

console.log('Listening on port 5000');

io = socketIO(server);

io.on('connection', function (socket) {

socket.on('upload-image', function (message) {

var path = 'socketio/' + message.name;

aws.write(path, message.data).then(function (response) {

return aws.readFile(path);

}).then(function (response) {

var base64 = response.Body.toString('base64');

socket.emit('image-uploaded', {

name: 'data:image/jpeg;base64,' + base64

});

});

});

});

1. Now, we need to add the client-side index.html file. This will be responsible for reading files with the FileReader() method and sending the file data to the server with Socket.IO.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title></title>

</head>

<body>

<hr />

<input type="file" id="my-file" />

<hr />

<script src="/socket.io/socket.io.js"></script>

<script type="text/javascript">

var socket = io.connect('http://localhost:5000');

var file = document.getElementById('my-file');

file.addEventListener('change', function () {

if (!file.files.length) {

return;

}

var firstFile = file.files[0],

reader = new FileReader();

reader.onloadend = function () {

socket.emit('upload-image', {

name: firstFile.name,

data: reader.result

});

};

reader.readAsArrayBuffer(firstFile);

});

socket.on('image-uploaded', function (message) {

var img = document.createElement('img');

img.setAttribute('src', message.name);

img.setAttribute('height', '100px');

document.body.appendChild(img);

});

</script>

</body>

</html>

1. Now, start your server and go to localhost:5000/index.html. You should be able to upload a file to AWS and see the results render below the upload input.

## How It Works...

When we upload binary image data to the server with Socket.IO, we call the aws.write() function to save data to AWS. The service hides most of the business logic involved in writing files to Amazon, so that the Socket.IO requests and responses are able to stay slim. This also makes the AWS reading and writing functions reusable for other endpoints to call.

The AWS SDK provides methods for reading, writing, listing and deleting files, so we are able to use those methods in our service to pass files to Amazon. As long as our environmental variables are set correctly and we are sending files to Amazon S3, everything should work.

# Streaming Audio

Streaming images with Socket.IO is great, but we can use Web Sockets in combination with WebRTC to stream audio from one users microphone to another.

WebRTC (or Web Real-Time Communication) is an API that supports browser-to-browser real-time media sharing for applications such as voice calling, video chat and peer-to-peer file sharing. WebRTC is still a relatively new technology. While WebRTC has support in most browsers, at the time of this writing Internet Explorer and Safari do not yet support it.

For two browsers to directly communicate over WebRTC, there is a handshake process that needs to take place. This means that one client makes an offer containing a description of the offer; the second client must then accept the offer and pass a reciprocal description. When the first client receives the answer to their offer, it must set the remote description that is contained in the offer answer. At that point, both clients have agreed to create a WebRTC connection and they are free to openly communicate.

Socket.IO is an ideal candidate for sending offers and answers to the offers before the connection is securely established.

In this recipe, we will use Socket.IO to help establish a WebRTC connection so that we can transmit live audio from one browser to another.

## How To Do It...

To stream audio with WebRTC and Socket.IO, follow these steps:

1. First, we will create our server.js file. This will be responsible to facilitating the connection of two clients to communicate with WebRTC.

var express = require('express'),

app = express(),

http = require('http'),

socketIO = require('socket.io'),

server, io;

app.use(express.static(\_\_dirname));

server = http.Server(app);

server.listen(5000);

console.log('Listening on port 5000');

io = socketIO(server);

io.on('connection', function (socket) {

socket.on('make-offer', function (data) {

socket.broadcast.emit('offer-made', {

offer: data.offer,

socket: socket.id

});

});

socket.on('make-answer', function (data) {

socket.to(data.to).emit('answer-made', {

socket: socket.id,

answer: data.answer

});

});

});

1. Now, we will create our sender.html file. This will simply display a button to broadcast our audio to the other client.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title></title>

<link rel="stylesheet" href="/style.css" media="screen" charset="utf-8" />

</head>

<body>

<button id="broadcast" class="play">Broadcast</button>

<script src="/socket.io/socket.io.js"></script>

<script type="text/javascript" src="/shared.js"></script>

<script type="text/javascript" src="/sender.js"></script>

</body>

</html>

1. Since we will be creating two separate pages, we will use a shared.js to include common variables and functions that both pages can use. This will mostly allow the WebRTC variables to fall back on browser vendor prefixes, since support for the un-prefixed namespaces are still spotty. We will also be creating our own peer connection to transmit data over.

var peerConnection = window.RTCPeerConnection ||

window.mozRTCPeerConnection ||

window.webkitRTCPeerConnection ||

window.msRTCPeerConnection;

var sessionDescription = window.RTCSessionDescription ||

window.mozRTCSessionDescription ||

window.webkitRTCSessionDescription ||

window.msRTCSessionDescription;

navigator.getUserMedia = navigator.getUserMedia ||

navigator.webkitGetUserMedia ||

navigator.mozGetUserMedia ||

navigator.msGetUserMedia;

var socket = io.connect('http://localhost:5000');

var pc = new peerConnection({ iceServers: [{ url: ‘stun:stun.services.mozilla.com’,

username: 'myuser',

credential: 'mycreds'

}]

});

function error (err) {

console.warn(err);

}

1. Now, we will need a sender.js file to pair with our sender.html template. This will allow the user to click on the Broadcast button to establish a peer-to-peer connection with other clients and stream data once the connection is created.

var answersFrom = {};

navigator.getUserMedia({ audio: true }, function (stream) {

pc.addStream(stream);

}, error);

function createOffer () {

pc.createOffer(function(offer) {

pc.setLocalDescription(new sessionDescription(offer), function () {

socket.emit('make-offer', {

offer: offer

});

}, error);

}, error);

}

socket.on('answer-made', function (data) {

pc.setRemoteDescription(new sessionDescription(data.answer), function () {

if (!answersFrom[data.socket]) {

createOffer(data.socket);

answersFrom[data.socket] = true;

}

}, error);

});

var btn = document.getElementById('broadcast');

btn.addEventListener('click', function () {

if (btn.getAttribute('class') === 'stop') {

btn.setAttribute('class', 'play');

btn.innerHTML = 'Broadcast';

} else {

btn.setAttribute('class', 'stop');

btn.innerHTML = 'Broadcasting...';

createOffer();

}

});

1. Now, we will create a receiver.html file, which will be where users can go to listen to the stream that will be created in our sender page. The receiver will wait for a connection to be requested from the sender and then the user can click on the single button to allow the connection to be created.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title></title>

<link rel="stylesheet" href="/style.css" media="screen" charset="utf-8" />

</head>

<body>

<button id="btn" class="muted">No Station...</button>

<script src="/socket.io/socket.io.js"></script>

<script type="text/javascript" src="/shared.js"></script>

<script type="text/javascript" src="/reciever.js"></script>

</body>

</html>

1. The receiver.js will handle listening for offers to be created in Socket.IO and allowing the user to accept the offer and begin streaming the audio feed from the sender.

var offerData,

player = new Audio(),

btn = document.getElementById('btn');

btn.addEventListener('click', function () {

if (btn.getAttribute('class') === 'play') {

listen();

player.play();

} else if (btn.getAttribute('class') === 'stop') {

player.pause();

btn.setAttribute('class', 'muted');

btn.innerHTML = 'No Station...';

}

});

function listen () {

btn.setAttribute('class', 'stop');

btn.innerHTML = 'Listening';

pc.setRemoteDescription(new sessionDescription(offerData.offer), function () {

pc.createAnswer(function (answer) {

pc.setLocalDescription(new sessionDescription(answer), function () {

socket.emit('make-answer', {

answer: answer,

to: offerData.socket

});

}, error);

}, error);

}, error);

}

pc.onaddstream = function (obj) {

console.log('addStream');

player.src = window.URL.createObjectURL(obj.stream);

};

socket.on('offer-made', function (data) {

btn.setAttribute('class', 'play');

btn.innerHTML = 'Listen';

offerData = data;

});

1. Finally, we’ll add a little CSS in a style.css file to change the color of our button as needed.

button {

font-size: 2em;

border: 0px;

color: #FFF;

}

button.play {

background: green;

}

button.stop {

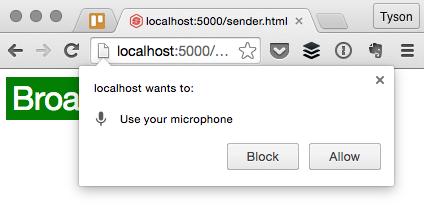
background: red;

}

button.muted {

background: #CCC;

}

1. Now, we can start our server and go to localhost:5000/sender.html in one browser and localhost:5000/receiver.html in a different browser. When you navigate to the sender.html page, it will prompt you to allow the browser to use your microphone. When you enable the microphone and click the green button, it will make an offer to establish a WebRTC connection with the other browser.  
     
   

Insert Image B04893\_07\_03.png

1. At first the receiver.html will say that no station is available

Macintosh HD:Users:tyson:Dropbox (Personal):socket.IO-Cookbook:07-streaming-binary-data:_assets:B04893_07_05.png

Insert Image B04893\_07\_05.png

1. However, once the sender initializes the connection, the button will change to read “Listen”

Macintosh HD:Users:tyson:Dropbox (Personal):socket.IO-Cookbook:07-streaming-binary-data:_assets:B04893_07_06.png

Insert Image B04893\_07\_06.png

1. Once the user clicks the “Listen” button, the connection will be fully established and they will begin to hear the streaming audio from the sender. Make sure to wear headphones if your sender and receiver are both being opened on the same computer, or you will get some nasty feedback.

## How It Works...

WebRTC and WebSockets work together really nicely to create a real-time streaming experience. In our example, we used Socket.IO to facilitate creating the connection between our clients and then we let WebRTC take it from there.

The WebRTC api allowed us to generate session descriptions, which we emitted over Socket.IO to authenticate the two browsers with each other. The descriptions are just basic JavaScript objects with some meta-data describing the type of connection we are trying to create, so they are easily transported over WebSockets.

# Streaming Live Video

While streaming audio is great, live video is even more gratifying. Using the WebRTC protocol, we can stream video in addition to audio and simply pipe it into an HTML video element instead of an audio element.

In this recipe, we will create a peer-to-peer connection where we can allow two users to chat using live video.

## How To Do It...

To stream live video with Socket.IO, follow these steps:

1. First, we will need to create a server.js file. This will be responsible for managing sockets as they join or leave and allowing the sockets to connect to one another to initiate a WebRTC session.

var express = require('express'),

app = express(),

http = require('http'),

socketIO = require('socket.io'),

fs = require('fs'),

path = require('path'),

server, io, sockets = [];

app.use(express.static(\_\_dirname));

app.get('/', function (req, res) {

res.sendFile(\_\_dirname + '/index.html');

});

server = http.Server(app);

server.listen(5000);

console.log('Listening on port 5000');

io = socketIO(server);

io.on('connection', function (socket) {

socket.emit('add-users', {

users: sockets

});

socket.broadcast.emit('add-users', {

users: [socket.id]

});

socket.on('make-offer', function (data) {

socket.to(data.to).emit('offer-made', {

offer: data.offer,

socket: socket.id

});

});

socket.on('make-answer', function (data) {

socket.to(data.to).emit('answer-made', {

socket: socket.id,

answer: data.answer

});

});

socket.on('disconnect', function () {

sockets.splice(sockets.indexOf(socket.id), 1);

io.emit('remove-user', socket.id);

});

sockets.push(socket.id);

});

1. Now, we will create the index.html template to display our client-side code.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title></title>

<link rel="stylesheet" href="/style.css" media="screen" charset="utf-8" />

</head>

<body>

<div class="container">

<video class="video-large" autoplay></video>

<div class="users-container" id="users-container">

<h4>Users</h4>

<div id="users"></div>

</div>

<div>

<script src="/socket.io/socket.io.js"></script>

<script type="text/javascript" src="/index.js"></script>

</body>

</html>

1. Next, we will create our index.js file, which will include our client-side JavaScript. This will be responsible for creating new WebRTC connections and responding to WebRTC connection requests from other users.

var socket = io.connect('http://localhost:5000');

var answersFrom = {}, offer;

var peerConnection = window.RTCPeerConnection ||

window.mozRTCPeerConnection ||

window.webkitRTCPeerConnection ||

window.msRTCPeerConnection;

var sessionDescription = window.RTCSessionDescription ||

window.mozRTCSessionDescription ||

window.webkitRTCSessionDescription ||

window.msRTCSessionDescription;

navigator.getUserMedia = navigator.getUserMedia ||

navigator.webkitGetUserMedia ||

navigator.mozGetUserMedia ||

navigator.msGetUserMedia;

var pc = new peerConnection({ iceServers: [{ url: "stun:stun.services.mozilla.com",

username: "somename",

credential: "somecredentials" }]

});

pc.onaddstream = function (obj) {

var vid = document.createElement('video');

vid.setAttribute('class', 'video-small');

vid.setAttribute('autoplay', 'autoplay');

vid.setAttribute('id', 'video-small');

document.getElementById('users-container').appendChild(vid);

vid.src = window.URL.createObjectURL(obj.stream);

}

navigator.getUserMedia({video: true}, function (stream) {

var video = document.querySelector('video');

video.src = window.URL.createObjectURL(stream);

pc.addStream(stream);

}, error);

function error (err) {

console.warn('Error', err);

}

function createOffer (id) {

pc.createOffer(function(offer) {

pc.setLocalDescription(new sessionDescription(offer), function () {

socket.emit('make-offer', {

offer: offer,

to: id

});

}, error);

}, error);

}

socket.on('answer-made', function (data) {

pc.setRemoteDescription(new sessionDescription(data.answer), function () {

document.getElementById(data.socket).setAttribute('class', 'active');

if (!answersFrom[data.socket]) {

createOffer(data.socket);

answersFrom[data.socket] = true;

}

}, error);

});

socket.on('offer-made', function (data) {

offer = data.offer;

pc.setRemoteDescription(new sessionDescription(data.offer), function () {

pc.createAnswer(function (answer) {

pc.setLocalDescription(new sessionDescription(answer), function () {

socket.emit('make-answer', {

answer: answer,

to: data.socket

});

}, error);

}, error);

}, error);

});

socket.on('add-users', function (data) {

for (var i = 0; i < data.users.length; i++) {

var el = document.createElement('div'),

id = data.users[i];

el.setAttribute('id', id);

el.innerHTML = id;

el.addEventListener('click', function () {

createOffer(id);

});

document.getElementById('users').appendChild(el);

}

});

socket.on('remove-user', function (id) {

var div = document.getElementById(id);

document.getElementById('users').removeChild(div);

});

1. Last of all, we will need to add some CSS in style.css to make our page look nice.

html, body {

padding: 0px;

margin: 0px;

}

video {

background: #CCC;

}

.container {

width: 100%;

}

.video-large {

width: 75%;

float: left;

}

.users-container {

width: 21%;

float: left;

padding: 2%;

position: relative;

}

.video-small {

margin-top: 20px;

width: 100%;

}

#users div {

color: red;

text-decoration: underline;

cursor: pointer;

}

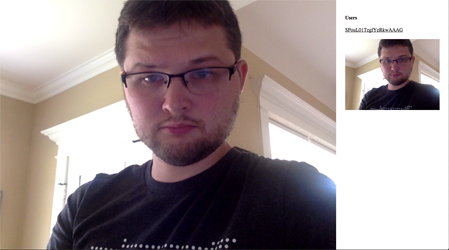
#users .active {

color: #000;

cursor: default;

}

1. Now, we can navigate to localhost:5000 in two browsers and click on the user that shows up in the right pane. This will kick off the process to create a WebRTC video connection using Socket.IO to message the connection description.



Insert Image B04893\_07\_07.png

## How It Works...

Most of the magic here is happening through WebRTC. Socket.IO simply handles the handshake process to authenticate the handshake on both ends.

Once the connection is established, we are piping data over the WebRTC connection with the addStream() method on the peer connection. We can create a data URL to pass as the src of our video element by calling window.URL.createObjectURL() and if the WebRTC session has been successfully authenticated, the video will stream as expected.