

WebRTC Guia rápido

Tutorial WebRTC

Versão PDF Guia rápido Recursos Pesquisa de emprego Discussão

Com Web Real-Time Communication (WebRTC), modernas aplicações web podem facilmente transmitir conteúdo de áudio e vídeo para milhões de pessoas. Neste tutorial, explicamos como você pode usar o WebRTC para configurar conexões peer-to-peer para outros navegadores da web de forma rápida e fácil.

**Público**

Este tutorial vai ajudar todos os desenvolvedores que gostariam de aprender como criar aplicativos, como publicidade em tempo real, jogos multiplayer, transmissão ao vivo, e-learning, para citar alguns, onde a ação ocorre em tempo real.

**Pré-requisitos**

O WebRTC é uma poderosa ferramenta que pode ser usada para infundir recursos de Comunicações em Tempo Real (RTC) em navegadores e aplicativos móveis. Este tutorial abrange apenas os conceitos básicos da WebRTC e qualquer desenvolvedor regular com algum nível de exposição ao gerenciamento de sessão em tempo real pode facilmente entender os conceitos aqui discutidos.

WebRTC – Overview

Visão Geral

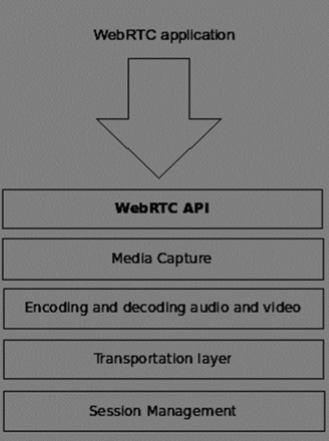
A Web não é mais um estranho para comunicação em tempo real como WebRTC (Comunicação Web em tempo real) entra em jogo. Apesar de ter sido lançado em maio de 2011, que ainda está em desenvolvimento e os seus padrões estão mudando. Um conjunto de protocolos é padronizado pelo *Real-Time Comunicação em navegadores web Grupo de Trabalho* em [http://tools.ietf.org/wg/rtcweb/](http://tools.ietf.org/wg/rtcweb/index.html) da IETF (Internet Engineering Task Force), enquanto novos conjuntos de APIs são padronizados pela Com o aparecimento do WebRTC, aplicações web modernas podem facilmente transmitir conteúdo de áudio e vídeo para milhões de pessoas.

Esquema Básico

WebRTC permite configurar conexões peer-to-peer para outros navegadores da web rapidamente e facilmente. Para construir um aplicativo, a partir do zero, você precisaria de uma grande variedade de estruturas e bibliotecas que lidam com questões típicas, como a perda de dados, conexão cair, e NAT. Com WebRTC, tudo isso vem embutido no navegador out-of-the-box. Esta tecnologia não precisa de plugins ou softwares de terceiros. É open-source e seu código fonte está livremente disponível em:

[http://www.webrtc.org/.](http://www.webrtc.org/index.html)

A API WebRTC inclui captura de mídia, codificação e decodificação de áudio e vídeo, a camada de transporte e gerenciamento de sessão.



media Capture

O primeiro passo é obter acesso à câmera e ao microfone do dispositivo do usuário. Nós detectar o tipo de dispositivos disponíveis, obter permissão do usuário para acessar esses dispositivos e gerenciar o fluxo.

Codificação e decodificação de áudio e vídeo

Ele não é uma tarefa fácil para enviar um fluxo de dados de áudio e de vídeo através da Internet. Isto é onde a codificação e descodificação são usados. Este é o processo de dividir até quadros de vídeo e ondas de áudio para pequenos pedaços e compactá-los. Este algoritmo é chamado codec.Há uma enorme quantidade de diferentes codecs, os quais são mantidos por empresas diferentes com diferentes objetivos de negócio. Há também muitos codecs dentro WebRTC como H.264, iSAC, Opus e VP8. Quando dois navegadores conectar juntos, eles escolher o codec suportado mais ideal entre dois usuários. Felizmente, WebRTC faz a maior parte da codificação nos bastidores.

Camada de transporte

A camada de transporte controla a ordem de pacotes, lidar com a perda de pacotes e se conectar a outros usuários. Mais uma vez a API WebRTC nos dá um fácil acesso a eventos que nos dizem quando há problemas com a conexão.

Gerenciamento de sessão

As ofertas de gerenciamento de sessão com a gestão, abertura e organização de conexões. Isto é comumente chamado de sinalização. Se você transferir fluxos de áudio e vídeo para o usuário, também faz sentido para transferir dados colaterais. Isto é feito pela API RTCDataChannel.

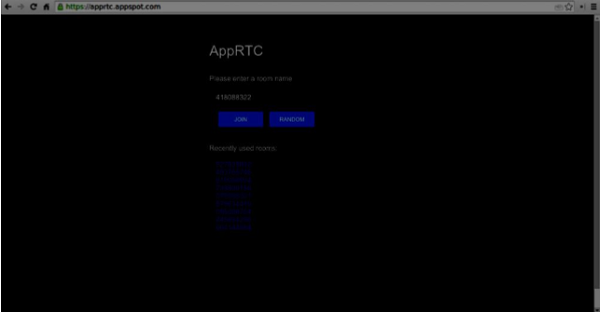
Engenheiros de empresas como Google, Mozilla, Opera e outros têm feito um grande trabalho para trazer esta experiência em tempo real para a Web.

Compatibilidade do navegador

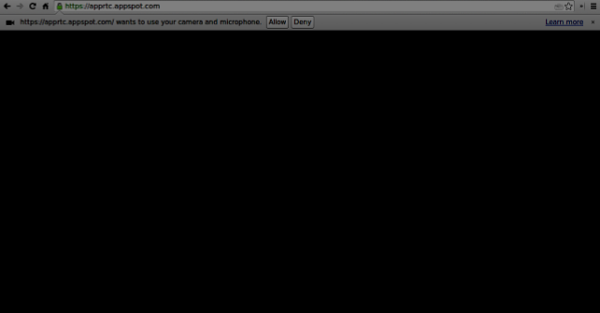
As normas WebRTC são um dos mais rápidos em evolução na web, por isso não quer dizer que cada navegador suporta todas as mesmas características ao mesmo tempo. Para verificar se o seu browser suporta WebRTC ou não, você pode visitar [http://caniuse.com/#feat=rtcpeerconnection.](http://caniuse.com/#feat=rtcpeerconnection/index.html)Ao longo de todos os tutoriais, eu recomendo que você use o Chrome para todos os exemplos.

Experimentar WebRTC

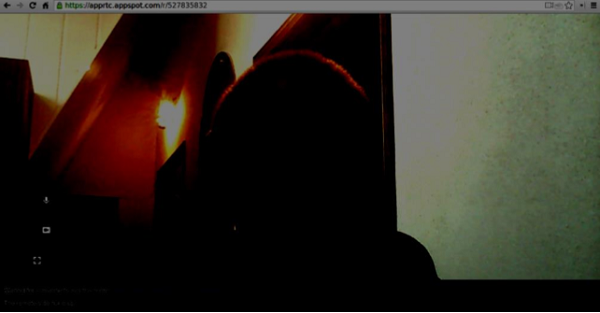
Vamos começar a usar WebRTC agora. Navegue seu navegador para o site demo em [https://apprtc.appspot.com/](http://apprtc.appspot.com/index.html)



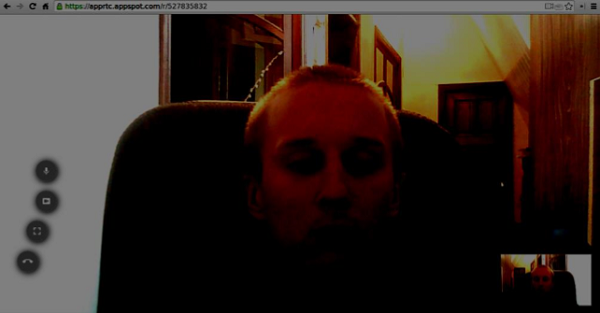
Clique no botão "Join". Você deverá ver uma notificação drop-down.



Clique no botão "Permitir" para iniciar a transmissão de seu vídeo e áudio para a página web. Você deverá ver um fluxo de vídeo de si mesmo.



Agora abra a URL que você está atualmente em uma nova aba do navegador e clique em "Cadastre-se". Você deverá ver dois fluxos de vídeo - um de seu primeiro cliente e outro da segunda.



Agora você deve entender por que WebRTC é uma ferramenta poderosa.

Casos de uso

A web em tempo real abre a porta para toda uma nova gama de aplicações, incluindo bate-papo baseado em texto, tela e compartilhamento de arquivos, jogos, vídeo chat, e muito mais. Além de comunicação que você pode usar WebRTC para outros fins, como -

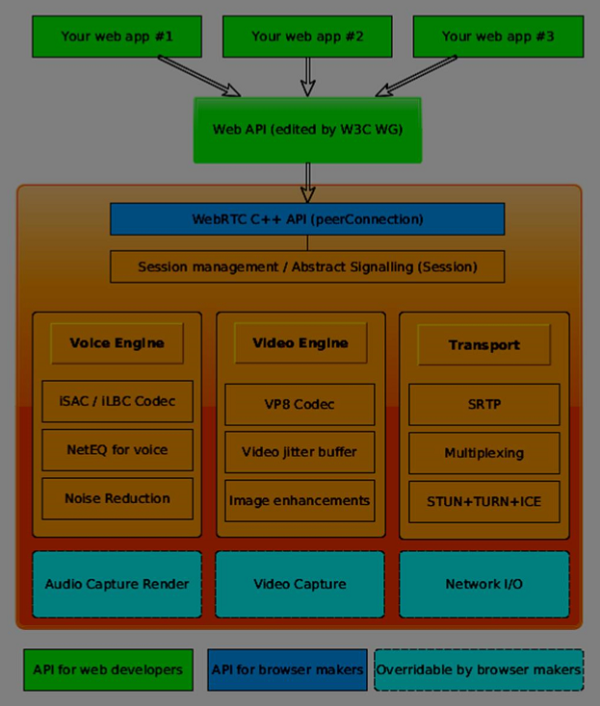
* de marketing em tempo real
* publicidade em tempo real
* comunicações de back-office (CRM, ERP, SCM, FFM)
* gestão de RH
* rede social
* serviços de namoro
* consultas médicas on-line
* serviços financeiros
* vigilância
* jogos multiplayer
* transmissão ao vivo
* e-learning

Resumo

Agora você deve ter uma compreensão clara do WebRTC prazo. Você também deve ter uma idéia do que tipos de aplicações podem ser construídos com WebRTC, como você já tentou fazê-lo no seu browser. Para resumir, WebRTC é uma tecnologia bastante útil.

WebRTC - Arquitetura

A arquitetura geral WebRTC tem um grande nível de complexidade.

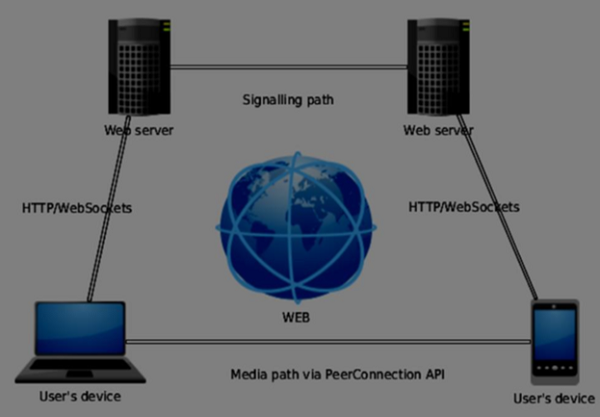


Aqui você pode encontrar três camadas diferentes -

* API para desenvolvedores web - esta camada contém todo o desenvolvedor APIs web necessário, incluindo RTCPeerConnection, RTCDataChannel e objetos MediaStrean.
* API para os fabricantes de browsers
* API substituível, que os fabricantes de browsers pode ligar.

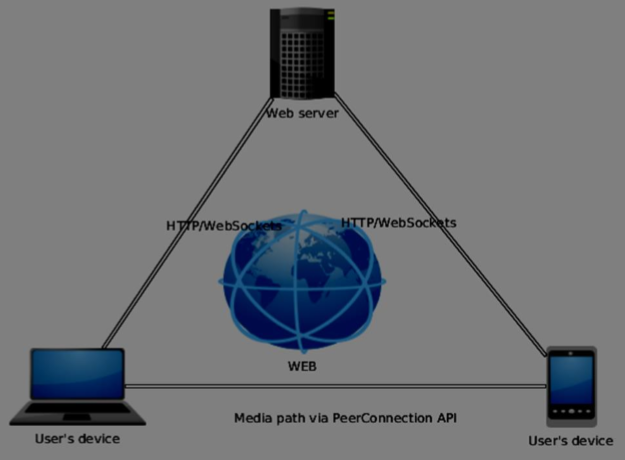
componentes de transporte permitem estabelecer conexões através de vários tipos de redes, enquanto os motores de voz e vídeo são estruturas responsáveis ​​pela transferência de áudio e vídeo a partir de uma placa de som e câmera à rede. Para os desenvolvedores da Web, a parte mais importante é API WebRTC.

Se olharmos para a arquitetura WebRTC do lado do cliente-servidor, podemos ver que um dos modelos mais utilizados é inspirado no SIP (Session Initiation Protocol) Trapézio.



Neste modelo, os dois dispositivos estão executando uma aplicação web a partir de diferentes servidores. O objeto RTCPeerConnection configura fluxos para que eles pudessem se conectar um ao outro, par peer-to-. Esta sinalização é feita através de HTTP ou WebSockets.

Mas o modelo mais utilizado é Triangle -



Neste modelo os dois dispositivos usam o mesmo aplicativo web. Dá desenvolvedor web mais flexibilidade ao gerenciamento de conexões de usuários.

O WebRTC API

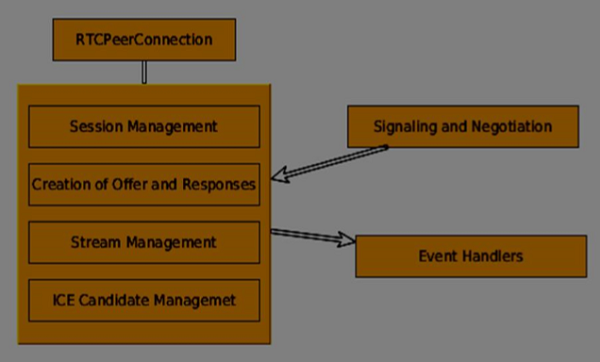
Ele consiste de algumas principais objetos JavaScript -

* RTCPeerConnection
* MediaStream
* RTCDataChannel

O objecto RTCPeerConnection

Este objeto é o principal ponto de entrada para a API WebRTC. Ele nos ajuda a conectar-se com seus pares, inicializar conexões e anexar fluxos de mídia. Ele também gerencia uma conexão UDP com outro usuário.

A principal tarefa do objeto RTCPeerConnection é configurar e criar uma conexão ponto. Nós podemos facilmente conectar chaves pontos da conexão porque este objeto dispara um conjunto de eventos quando eles aparecem. Estes acontecimentos dão-lhe acesso à configuração da nossa conexão -



O RTCPeerConnection é um simples objeto javascript, que você pode simplesmente criar desta forma -

[code]

var conn = new RTCPeerConnection(conf);

conn.onaddstream = function(stream) {

// use stream here

};

[/code]

O objeto RTCPeerConnection aceita um parâmetro *conf,* que vamos cobrir mais tarde estes tutoriais. O evento *onaddstream* é acionado quando o usuário remoto adiciona um vídeo ou stream de áudio para sua conexão ponto.

MediaStream API

Navegadores modernos dar um desenvolvedor acesso à API *getUserMedia,* também conhecida como a API *MediaStream.* Há três pontos-chave da funcionalidade -

* Ele dá um desenvolvedor acesso a um objeto de *fluxo* que representam fluxos de vídeo e áudio
* Ele gerencia a seleção de dispositivos de usuários de entrada no caso de um usuário tem múltiplas câmeras ou microfones em seu dispositivo
* Ele fornece um nível de segurança pedindo usuário o tempo todo ele quer buscar s córrego

Para testar esta API vamos criar uma página HTML simples. Ela vai mostrar um único elemento <video>, peça a permissão do usuário para usar a câmera e mostrar uma transmissão ao vivo da câmera na página. Crie um arquivo *index.html* e adicione -

[code]

<html>

<head>

<meta charset = "utf-8">

</head>

<body>

<video autoplay></video>

<script src = "client.js"></script>

</body>

</html>

[/code]

Em seguida, adicione um arquivo *Client.js* -

[code]

//checks if the browser supports WebRTC

function hasUserMedia() {

navigator.getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia

|| navigator.mozGetUserMedia || navigator.msGetUserMedia;

return !!navigator.getUserMedia;

}

if (hasUserMedia()) {

navigator.getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia

|| navigator.mozGetUserMedia || navigator.msGetUserMedia;

//get both video and audio streams from user's camera

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

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

//insert stream into the video tag

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

}, function (err) {});

}else {

alert("Error. WebRTC is not supported!");

}

[/code]

Agora, abra o *index.html* e você deve ver o fluxo de vídeo exibindo seu rosto.

Mas cuidado, porque WebRTC funciona apenas no lado do servidor. Se você simplesmente abrir esta página com o navegador não vai funcionar. Você precisa para hospedar esses arquivos no Apache ou servidores node.js, ou qual deles você prefere.

O objecto RTCDataChannel

Bem como o envio de fluxos de mídia entre os pares, você também pode enviar dados adicionais usando *DataChannel* API. Esta API é tão simples como MediaStream API. O trabalho principal é criar um canal vindo de um objeto RTCPeerConnection existente -

[code]

var peerConn = new RTCPeerConnection();

//establishing peer connection

//...

//end of establishing peer connection

var dataChannel = peerConnection.createDataChannel("myChannel", dataChannelOptions);

// here we can start sending direct messages to another peer

[/code]

Isto é tudo o que você precisava, apenas duas linhas de código. Tudo o resto é feito na camada interna do browser. Você pode criar um canal em qualquer conexão ponto até o *RTCPeerConnectionobject* está fechado.

Resumo

Agora você deve ter um firme aperto da arquitetura WebRTC. Também cobriu MediaStream, RTCPeerConnection e APIs RTCDataChannel. A API WebRTC é um alvo em movimento, por isso sempre manter-se com as mais recentes especificações.

WebRTC - Ambiente

Antes de começar a construir as nossas aplicações WebRTC, devemos definir o nosso ambiente de codificação. Primeiro de tudo, você deve ter um editor de texto ou IDE onde você pode editar HTML e Javascript. Há chances de que você já escolheu o preferido como você está lendo este tutorial. Quanto a mim, estou usando o IDE WebStorm. Você pode baixar a versão trial em [https://www.jetbrains.com/webstorm/](http://www.jetbrains.com/webstorm/index.html) . Eu também estou usando Linux Mint como meu sistema operacional de escolha.

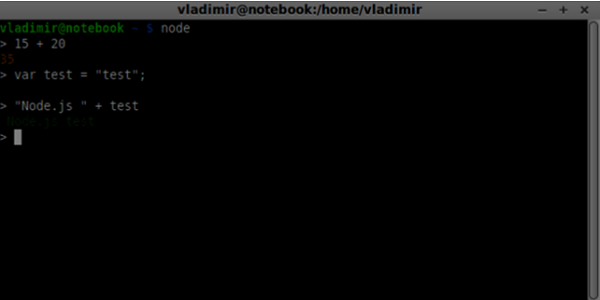
O outro requisito para aplicações WebRTC comum é ter um servidor para hospedar os arquivos HTML e Javascript. O código não irá funcionar apenas com um duplo clique nos arquivos porque o navegador não é permitido para se conectar a câmeras e microfones, a menos que os arquivos estão sendo servidos por um servidor real. Isto é feito, obviamente, devido aos problemas de segurança.

Há toneladas de diferentes servidores web, mas neste tutorial, vamos usar Node.js com nó-estático -

* Visite [https://nodejs.org/en/](http://nodejs.org/en/index.html) e baixar a versão mais recente Node.js.
* Descompacte-o para o usr / local / diretório / NodeJS.
* Abra o arquivo /home/YOUR\_USERNAME/.profile e adicione a seguinte linha ao final - *export PATH = $ PATH: / usr / local / nodejs / bin*
* O que você pode reiniciar o computador ou a fonte de executar */home/YOUR\_USERNAME/.profile*
* Agora, o comando do *nó* deve estar disponível a partir da linha de comando. O comando *NPM*também está disponível. NMP é o gerenciador de pacotes para Node.js. Você pode aprender mais em [https://www.npmjs.com/](http://www.npmjs.com/index.html) .
* Abra um terminal e executar *sudo NPM instalar -g nó-estático.* Isto irá instalar o servidor web estático para Node.js.
* Agora navegar para qualquer pasta que contém os arquivos HTML e execute o comando *estático* dentro do diretório para iniciar o seu servidor web.
* Você pode navegar para http: // localhost: 8080 para ver seus arquivos.

Há uma outra maneira de instalar nodejs. *Sudo* Basta executar *apt-get install nodejs* na janela do terminal.

Para testar a instalação Node.js abrir seu terminal e execute o comando *nó.* Digitar alguns comandos para verificar como ele funciona -



Node.js executa arquivos Javascript, bem como comandos digitados no terminal. Crie um arquivo *index.js* com o seguinte conteúdo -

console.log(“Testing Node.js”);

Em seguida, execute o comando *índice nó.* Você verá a seguinte -



Ao construir o nosso servidor de sinalização, vamos utilizar uma biblioteca de WebSockets para Node.js. Para instalar no *NPM* run *instalar ws* no terminal.

Para testar o nosso servidor de sinalização, vamos usar o utilitário wscat. Para instalá-lo correr *NPM instalar wscat -g* na sua janela de terminal.

|  |  |
| --- | --- |
| S.No | Protocolos e Descrição |
| 1 | [WebRTC Protocolos](http://www.w3ii.com/pt/webrtc/webrtc_protocols.html)  aplicações WebRTC usar UDP (User Datagram Protocol) como protocolo de transporte. A maioria das aplicações web hoje são construídos com a utilização do (Transmission Control Protocol) TCP |
| 2 | [Session Description Protocol](http://www.w3ii.com/pt/webrtc/webrtc_session_description_protocol.html)  O SDP é uma parte importante do WebRTC. É um protocolo que se destina a descrever as sessões de comunicação multimédia. |
| 3 | [Encontrar uma rota](http://www.w3ii.com/pt/webrtc/webrtc_finding_route.html)  Para se conectar a outro usuário, você deve encontrar um caminho claro em torno de sua própria rede e de rede do outro usuário. Mas há chances de que a rede está a utilizar tem vários níveis de controle de acesso para evitar problemas de segurança. |
| 4 | [Stream Control Transmission Protocol](http://www.w3ii.com/pt/webrtc/webrtc_sctp.html)  Com a conexão ponto, temos a capacidade de enviar dados de forma rápida de vídeo e áudio. O protocolo SCTP é usado hoje para enviar dados blob no topo da nossa atualmente conexão ponto a definição quando utiliza o objeto RTCDataChannel. |

Resumo

Neste capítulo, que abrangeu várias das tecnologias que permitem conexões de mesmo nível, como UDP, TCP, atordoamento, vire, ICE e SCTP. Agora você deve ter uma compreensão de nível de superfície de como SDP obras e seus casos de uso.

WebRTC - MediaStream APIs

A API MediaStream foi projetado para facilitar o acesso aos fluxos de mídia a partir de câmeras e microfones locais. O método *getUserMedia ()* é a principal maneira de acessar dispositivos de entrada locais.

A API tem alguns pontos-chave -

* Um fluxo de mídia em tempo real é representado por um objeto de *fluxo* na forma de vídeo ou áudio
* Ele fornece um nível de segurança através de permissões de usuários que pedem ao usuário antes de uma aplicação web pode começar a buscar um fluxo
* A selecção de dispositivos de entrada é tratado pela API MediaStream (por exemplo, quando há duas câmaras ou microfones conectados ao dispositivo)

Cada objeto MediaStream inclui vários objetos MediaStreamTrack. Eles representam vídeo e áudio a partir de diferentes dispositivos de entrada.

Cada objeto MediaStreamTrack pode incluir vários canais (direito e canais de áudio esquerdo). Estas são as menores partes definidas pela API MediaStream.

Há duas maneiras para a saída MediaStream objetos. Primeiro, podemos renderizar a saída em um vídeo ou elemento de áudio. Em segundo lugar, podemos enviar a saída para o objeto RTCPeerConnection, que, em seguida, enviá-lo para um ponto remoto.

Usando a API MediaStream

Vamos criar um aplicativo WebRTC simples. Ela vai mostrar um elemento de vídeo na tela, pergunte ao usuário permissão para usar a câmera, e mostrar um fluxo de vídeo ao vivo no navegador. Crie um arquivo *index.html* -

<!DOCTYPE html>

<html lang = "en">

<head>

<meta charset = "utf-8" />

</head>

<body>

<video autoplay></video>

<script src = "client.js"></script>

</body>

</html>

Em seguida, crie o *arquivo Client.js* e adicione o seguinte;

function hasUserMedia() {

//check if the browser supports the WebRTC

return !!(navigator.getUserMedia || navigator.webkitGetUserMedia ||

navigator.mozGetUserMedia);

}

if (hasUserMedia()) {

navigator.getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia

|| navigator.mozGetUserMedia;

//enabling video and audio channels

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

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

//inserting our stream to the video tag

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

}, function (err) {});

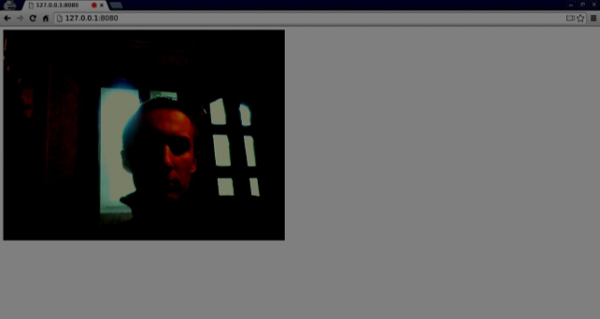
} else {

alert("WebRTC is not supported");

}

Aqui criamos a função *hasUserMedia ()* que verifica se WebRTC é suportado ou não. Em seguida, acessar a função *getUserMedia* onde o segundo parâmetro é um callback que aceitar o fluxo vindo do dispositivo do usuário. Em seguida, carregar o nosso fluxo para o elemento de *vídeo* usando *window.URL.createObjectURL* que cria uma URL que representa o objeto dado no parâmetro.

Agora atualizar sua página, clique em Permitir, e você deve ver o seu rosto na tela.



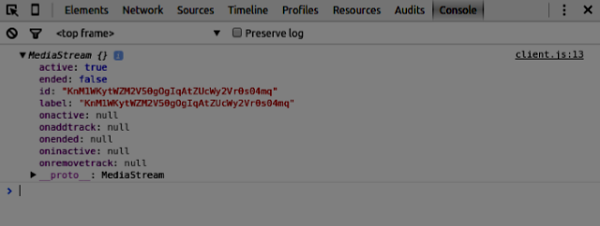
Lembre-se de executar todos os seus scripts usando o servidor web. Nós já instalou um no WebRTC Ambiente Tutorial.

MediaStream API

propriedades

* MediaStream.active (somente leitura) - Retorna true se o MediaStream está ativo, ou falso caso contrário.
* MediaStream.ended (somente leitura, obsoleto) - Retorna verdadeiro se o MediaStream.id (somente leitura) - Um identificador exclusivo para o objeto.
* MediaStream.label (somente leitura, obsoleto) - Um identificador único atribuído pelo agente do usuário.

Você pode ver como as propriedades acima olhar no meu navegador -



Manipuladores de eventos

* MediaStream.onactive - Um manipulador para um evento MediaStream.onaddtrack - Um manipulador para um evento MediaStream.onended (reprovado) - Um manipulador para um evento MediaStream.oninactive - Um manipulador para um evento MediaStream.onremovetrack - Um manipulador para um evento Métodos
  + MediaStream.addTrack () - Adiciona o objeto Se a faixa já foi adicionado, nada acontece.
  + MediaStream.clone () - Retorna um clone do objeto MediaStream com um novo ID.
  + MediaStream.getAudioTracks () - Retorna uma lista dos objetos MediaStream.getTrackById () - Retorna a pista pela ID. Se o argumento for vazio ou o ID não for encontrado, ele retorna null. Se várias faixas têm o mesmo ID, ele retorna o primeiro.
  + MediaStream.getTracks () - Retorna uma lista de todos os objetos MediaStream.getVideoTracks () - Retorna uma lista de o vídeo MediaStream.removeTrack () - Remove o objeto Se a faixa já tenha sido removido, nada acontece.

Para testar as APIs acima alterar alterar o *index.html* da seguinte maneira -

<!DOCTYPE html>

<html lang = "en">

<head>

<meta charset = "utf-8" />

</head>

<body>

<video autoplay></video>

<div><button id = "btnGetAudioTracks">getAudioTracks()

</button></div>

<div><button id = "btnGetTrackById">getTrackById()

</button></div>

<div><button id = "btnGetTracks">getTracks()</button></div>

<div><button id = "btnGetVideoTracks">getVideoTracks()

</button></div>

<div><button id = "btnRemoveAudioTrack">removeTrack() - audio

</button></div>

<div><button id = "btnRemoveVideoTrack">removeTrack() - video

</button></div>

<script src = "client.js"></script>

</body>

</html>

Nós adicionamos alguns botões para experimentar várias APIs MediaStream. Em seguida, deve adicionar manipuladores de eventos para o nosso botão recém-criado. Modificar as *Client.js* arquivo desta maneira -

var stream;

function hasUserMedia() {

//check if the browser supports the WebRTC

return !!(navigator.getUserMedia || navigator.webkitGetUserMedia ||

navigator.mozGetUserMedia);

}

if (hasUserMedia()) {

navigator.getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia

|| navigator.mozGetUserMedia;

//enabling video and audio channels

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

stream = s;

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

//inserting our stream to the video tag

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

}, function (err) {});

} else {

alert("WebRTC is not supported");

}

btnGetAudioTracks.addEventListener("click", function(){

console.log("getAudioTracks");

console.log(stream.getAudioTracks());

});

btnGetTrackById.addEventListener("click", function(){

console.log("getTrackById");

console.log(stream.getTrackById(stream.getAudioTracks()[0].id));

});

btnGetTracks.addEventListener("click", function(){

console.log("getTracks()");

console.log(stream.getTracks());

});

btnGetVideoTracks.addEventListener("click", function(){

console.log("getVideoTracks()");

console.log(stream.getVideoTracks());

});

btnRemoveAudioTrack.addEventListener("click", function(){

console.log("removeAudioTrack()");

stream.removeTrack(stream.getAudioTracks()[0]);

});

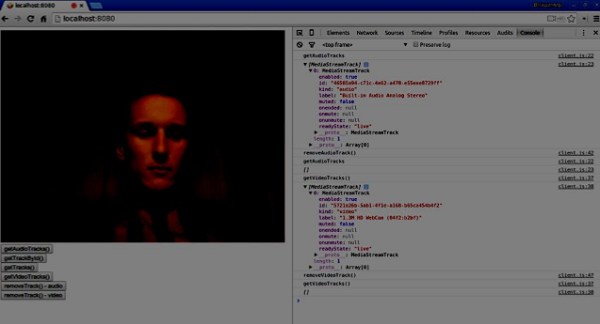
btnRemoveVideoTrack.addEventListener("click", function(){

console.log("removeVideoTrack()");

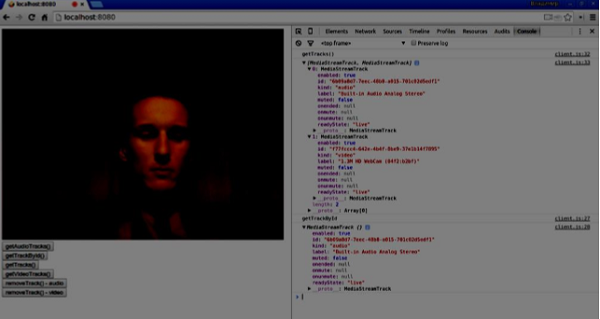
stream.removeTrack(stream.getVideoTracks()[0]);

});

Agora atualizar sua página. Clique no botão *getAudioTracks (),* em seguida, clique no *removeTrack () -* botão de *áudio.* A faixa de áudio deve agora ser removido. Em seguida, faça o mesmo para a trilha de vídeo.



Se você clicar nos *getTracks ()* botão que você deve ver todos *MediaStreamTracks* (todo o vídeo conectado e entradas de áudio). Em seguida, clique no *getTrackById ()* para obter MediaStreamTrack áudio.



Resumo

Neste capítulo, criamos um aplicativo WebRTC simples usando a API MediaStream. Agora você deve ter uma visão clara das várias APIs MediaStream que fazem o trabalho WebRTC.

WebRTC - RTCPeerConnection APIs

A API RTCPeerConnection é o núcleo da ligação ponto-a-ponto entre cada um dos navegadores. Para criar os objetos RTCPeerConnection basta escrever

var pc = RTCPeerConnection(config);

onde o argumento *de configuração* contém pelo menos na chave, iceServers. É uma matriz de objetos URL contendo informações sobre os servidores STUN e de sua vez, utilizados durante a descoberta dos candidatos ICE. Você pode encontrar uma lista de servidores STUN pública disponíveis em [code.google.com](http://code.google.com/p/natvpn/source/browse/trunk/stun_server_list/index.html)

Dependendo de se é o chamador ou o receptor do objecto RTCPeerConnection é usado de uma forma ligeiramente diferente em cada lado da ligação.

Aqui está um exemplo de fluxo do usuário -

* + Registrar o manipulador *onicecandidate.* Ele envia quaisquer candidatos ICE para os outros pares, como eles são recebidos.
  + Registrar o manipulador *onaddstream.* Ele lida com a exibição do fluxo de vídeo, uma vez que é recebida do ponto remoto.
  + Registrar o manipulador de *mensagem.* Seu servidor de sinalização também deve ter um manipulador para as mensagens recebidas a partir do outro par. Se a mensagem contém o objecto *RTCSessionDescription,* ele deve ser adicionado ao objecto *RTCPeerConnection*usando o método *setRemoteDescription ().* Se a mensagem contém o objecto *RTCIceCandidate,* ele deve ser adicionado ao objecto *RTCPeerConnection* usando o método *addIceCandidate ().*
  + Utilize *getUserMedia ()* para configurar o fluxo de mídia local e adicioná-lo ao objeto *RTCPeerConnection* usando o método *addStream ().*
  + Iniciar processo de negociação oferta / resposta. Este é o único passo em que o fluxo do chamador é diferente de um do receptor. O chamador inicia negociação utilizando o método *createOffer ()* e registra um callback que recebe o objeto *RTCSessionDescription.*Então este callback deve adicionar esse objeto *RTCSessionDescription* para o seu objeto *RTCPeerConnection* usando *setLocalDescription ().* E, finalmente, o chamador deve enviar esta *RTCSessionDescription* ao ponto remoto usando o servidor de sinalização. O receptor, por outro lado, regista o mesmo de retorno, mas no método *createAnswer ().*Note-se que o fluxo de receptor é iniciada apenas após a oferta for recebida a partir do chamador.

RTCPeerConnection API

propriedades

* + RTCPeerConnection.iceConnectionState (somente leitura) - Retorna uma enumeração RTCIceConnectionState que descreve o estado da conexão. Um evento iceconnectionstatechange é acionado quando esse valor é alterado. Os valores possíveis -
    - novo - o agente ICE está à espera de candidatos remotos ou endereços de coleta
    - verificando - o agente ICE tem candidatos remotos, mas não encontrou uma ligação ainda
    - Conectado - o agente ICE tem encontrado uma ligação utilizável, mas ainda está verificando candidato mais remota para uma melhor conexão.
    - concluída - o agente ICE tem encontrado uma ligação utilizável e parou de testar os candidatos remotos.
    - falhou - o agente ICE verificou todos os candidatos remotos, mas não encontrar uma correspondência para pelo menos um componente.
    - desconectado - pelo menos um componente não está mais vivo.
    - fechado - o agente ICE está fechado.
  + RTCPeerConnection.iceGatheringState (somente leitura) - Retorna uma enumeração RTCIceGatheringState que descreve o estado encontro ICE para a conexão -
    - novo - o objeto foi criado.
    - recolhimento - o agente ICE está no processo de coleta de candidatos
    - completar o agente ICE completou encontro.
  + RTCPeerConnection.localDescription (somente leitura) - Retorna uma RTCSessionDescription descrevendo a sessão local. Pode ser nulo, se ele ainda não foi definido.
  + RTCPeerConnection.peerIdentity (somente leitura) - Retorna uma RTCIdentityAssertion.É constituída por um IDP (nome de domínio) e um nome representando a identidade do posto remoto.
  + RTCPeerConnection.remoteDescription (somente leitura) - Retorna um RTCSessionDescription descrevendo a sessão remota. Pode ser nulo, se ele ainda não foi definido.
  + RTCPeerConnection.signalingState (somente leitura) - Retorna uma enumeração RTCSignalingState que descreve o estado de sinalização da conexão local. Este estado descreve a oferta SDP. Um evento signalingstatechange é acionado quando esse valor é alterado. Os valores possíveis -
    - estável - O estado inicial. Não há oferta SDP / troca resposta em andamento.
    - tem-local-oferta - o lado local da conexão foi aplicado localmente uma oferta SDP.
    - tem-remote-oferta - o lado remoto da conexão foi aplicado localmente uma oferta SDP.
    - tem-local-pranswer - uma oferta SDP remota tiver sido aplicada, e uma pranswer SDP aplicado localmente.
    - tem-remote-pranswer - um SDP local tem sido aplicado, e uma pranswer SDP aplicada remotamente.
    - fechada - a conexão é fechada.

Manipuladores de eventos

Dada a seguir são os manipuladores de eventos comumente usado de RTCPeerConnection.

|  |  |  |  |
| --- | --- | --- | --- |
| S.No. | | | Manipuladores de & Descrição do Evento |
| 1 | RTCPeerConnection.onaddstream  Este manipulador é chamado quando o evento addstream é acionado. Este evento é enviado quando uma MediaStream é adicionado a este respeito pela peer remoto. |
| 2 | RTCPeerConnection.ondatachannel  Este manipulador é chamado quando o evento datachannel é acionado. Este evento é enviado quando uma RTCDataChannel é adicionado a esta conexão. |
| 3 | RTCPeerConnection.onicecandidate  Este manipulador é chamado quando o evento icecandidate é acionado. Este evento é enviado quando um objeto RTCIceCandidate é adicionado ao script. |
| 4 | RTCPeerConnection.oniceconnectionstatechange  Este manipulador é chamado quando o evento iceconnectionstatechange é acionado. Este caso é enviada quando o valor de iceConnectionState muda. |
| 5 | RTCPeerConnection.onidentityresult  Este manipulador é chamado quando o evento identityresult é acionado. Este evento é enviado quando uma afirmação de identidade é gerado durante a criação de uma oferta ou uma resposta da via getIdentityAssertion (). |
| 6 | RTCPeerConnection.onidpassertionerror  Este manipulador é chamado quando o evento idpassertionerror é acionado. Este evento é enviado quando o IDP (Provedor Identitry) encontra um erro ao gerar uma afirmação de identidade. |
| 7 | RTCPeerConnection.onidpvalidation  Este manipulador é chamado quando o evento idpvalidationerror é acionado. Este evento é enviado quando o IDP (Provedor Identitry) encontra um erro ao validar uma afirmação de identidade. |
| 8 | RTCPeerConnection.onnegotiationneeded  Este manipulador é chamado quando o evento negotiationneeded é acionado. Este evento é enviado pelo navegador para informar a negociação será necessária em algum momento no futuro. |
| 9 | RTCPeerConnection.onpeeridentity  Este manipulador é chamado quando o evento peeridentity é acionado. Este evento é enviado quando uma identidade entre pares foi definido e verificado nesta conexão. |
| 10 | RTCPeerConnection.onremovestream  Este manipulador é chamado quando o evento signalingstatechange é acionado. Este caso é enviada quando o valor de signalingState muda. |
| 11 | RTCPeerConnection.onsignalingstatechange  Este manipulador é chamado quando o evento removestream é acionado. Este evento é enviado quando uma MediaStream é removido desta conexão. |

Métodos

Dada a seguir são os métodos mais utilizados de RTCPeerConnection.

|  |  |
| --- | --- |
| S.No. | Métodos e Descrição |
| 1 | RTCPeerConnection ()  Retorna um novo objeto RTCPeerConnection. |
| 2 | RTCPeerConnection.createOffer ()  Cria uma oferta (request) para encontrar um ponto remoto. Os dois primeiros parâmetros deste método são êxito e de erro chamadas de retorno. O terceiro parâmetro opcional são opções, como permitindo fluxos de áudio ou vídeo. |
| 3 | RTCPeerConnection.createAnswer ()  Cria uma resposta à oferta recebida pelo peer remoto durante o processo de negociação oferta / resposta. Os dois primeiros parâmetros deste método são êxito e de erro chamadas de retorno. O terceiro parâmetro opcional são opções para a resposta a ser criado. |
| 4 | RTCPeerConnection.setLocalDescription ()  Altera a descrição de conexão local. A descrição define as propriedades da ligação. A conexão deve ser capaz de suportar ambas as descrições antigas e novas. O método tem três parâmetros, RTCSessionDescription objeto, retorno de chamada, se a mudança de descrição bem-sucedido, chamada de retorno, se a mudança de descrição falha. |
| 5 | RTCPeerConnection.setRemoteDescription ()  Altera a descrição conexão remota. A descrição define as propriedades da ligação. A conexão deve ser capaz de suportar ambas as descrições antigas e novas. O método tem três parâmetros, RTCSessionDescription objeto, retorno de chamada, se a mudança de descrição bem-sucedido, chamada de retorno, se a mudança de descrição falha. |
| 6 | RTCPeerConnection.updateIce ()  Atualiza o processo do agente ICE de ping candidatos remotos e coleta de candidatos locais. |
| 7 | RTCPeerConnection.addIceCandidate ()  Fornece um candidato remoto para o agente ICE. |
| 8 | RTCPeerConnection.getConfiguration ()  Retorna um objeto RTCConfiguration. Ela representa a configuração do objecto RTCPeerConnection. |
| 9 | RTCPeerConnection.getLocalStreams ()  Retorna uma matriz de conexão MediaStream local. |
| 10 | RTCPeerConnection.getRemoteStreams ()  Retorna uma matriz de conexão MediaStream remoto. |
| 11 | RTCPeerConnection.getStreamById ()  Retorna MediaStream local ou remoto pela ID dado. |
| 12 | RTCPeerConnection.addStream ()  Adiciona um MediaStream como uma fonte local de vídeo ou áudio. |
| 13 | RTCPeerConnection.removeStream ()  Remove um MediaStream como uma fonte local de vídeo ou de áudio. |
| 14 | RTCPeerConnection.close ()  Fecha uma conexão. |
| 15 | RTCPeerConnection.createDataChannel ()  Cria um novo RTCDataChannel. |
| 16 | RTCPeerConnection.createDTMFSender ()  Cria um novo RTCDTMFSender, associado a um MediaStreamTrack específico. Permite enviar DTMF (Dual-tone multifrequência) de telefone de sinalização através da ligação. |
| 17 | RTCPeerConnection.getStats ()  Cria um novo RTCStatsReport que contém estatísticas sobre a conexão. |
| 18 | RTCPeerConnection.setIdentityProvider ()  Define o IDP. Leva três parâmetros - o nome, o protocolo usado para se comunicar e um nome de usuário opcional. |
| 19 | RTCPeerConnection.getIdentityAssertion ()  Reúne uma afirmação de identidade. Não é esperado para lidar com este método na aplicação. Assim, você pode chamá-lo de forma explícita apenas para antecipar a necessidade. |

Estabelecendo uma conexão

Agora vamos criar uma aplicação exemplo. Em primeiro lugar, execute o servidor de sinalização que criamos no tutorial "servidor de sinalização" via "servidor do nó".

Haverá duas entradas de texto na página, um para um login e uma para um nome de usuário que deseja se conectar. Crie um arquivo *index.html* e adicione o seguinte código -

<html lang = "en">

<head>

<meta charset = "utf-8" />

</head>

<body>

<div>

<input type = "text" id = "loginInput" />

<button id = "loginBtn">Login</button>

</div>

<div>

<input type = "text" id = "otherUsernameInput" />

<button id = "connectToOtherUsernameBtn">Establish connection</button>

</div>

<script src = "client2.js"></script>

</body>

</html>

Você pode ver que nós adicionamos a entrada de texto para um início de sessão, o botão de login, a entrada de texto para o outro nome de usuário de pares, e o botão de conexão-to-lo. Agora crie um arquivo *Client.js* e adicione o seguinte código -

var connection = new WebSocket('ws://localhost:9090');

var name = "";

var loginInput = document.querySelector('#loginInput');

var loginBtn = document.querySelector('#loginBtn');

var otherUsernameInput = document.querySelector('#otherUsernameInput');

var connectToOtherUsernameBtn = document.querySelector('#connectToOtherUsernameBtn');

var connectedUser, myConnection;

//when a user clicks the login button

loginBtn.addEventListener("click", function(event){

name = loginInput.value;

if(name.length > 0){

send({

type: "login",

name: name

});

}

});

//handle messages from the server

connection.onmessage = function (message) {

console.log("Got message", message.data);

var data = JSON.parse(message.data);

switch(data.type) {

case "login":

onLogin(data.success);

break;

case "offer":

onOffer(data.offer, data.name);

break;

case "answer":

onAnswer(data.answer);

break;

case "candidate":

onCandidate(data.candidate);

break;

default:

break;

}

};

//when a user logs in

function onLogin(success) {

if (success === false) {

alert("oops...try a different username");

} else {

//creating our RTCPeerConnection object

var configuration = {

"iceServers": [{ "url": "stun:stun.1.google.com:19302" }]

};

myConnection = new webkitRTCPeerConnection(configuration);

console.log("RTCPeerConnection object was created");

console.log(myConnection);

//setup ice handling

//when the browser finds an ice candidate we send it to another peer

myConnection.onicecandidate = function (event) {

if (event.candidate) {

send({

type: "candidate",

candidate: event.candidate

});

}

};

}

};

connection.onopen = function () {

console.log("Connected");

};

connection.onerror = function (err) {

console.log("Got error", err);

};

// Alias for sending messages in JSON format

function send(message) {

if (connectedUser) {

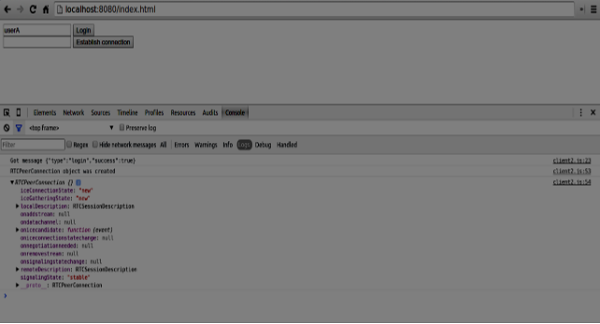
message.name = connectedUser;

}

connection.send(JSON.stringify(message));

};

Você pode ver que nós estabelecer uma conexão de soquete para o nosso servidor de sinalização. Quando um usuário clica no botão de login a aplicação envia seu nome de usuário para o servidor. Se o login for bem sucedido o aplicativo cria o manipulador onicecandidate objeto e configuração RTCPeerConnection que envia todos os icecandidates encontrados para o outro par. Agora, abra a página e tente fazer o login. Você deverá ver a seguinte saída do console -



O próximo passo é criar uma oferta para os outros pares. Adicione o seguinte código ao seu arquivo *Client.js* -

//setup a peer connection with another user

connectToOtherUsernameBtn.addEventListener("click", function () {

var otherUsername = otherUsernameInput.value;

connectedUser = otherUsername;

if (otherUsername.length > 0) {

//make an offer

myConnection.createOffer(function (offer) {

console.log();

send({

type: "offer",

offer: offer

});

myConnection.setLocalDescription(offer);

}, function (error) {

alert("An error has occurred.");

});

}

});

//when somebody wants to call us

function onOffer(offer, name) {

connectedUser = name;

myConnection.setRemoteDescription(new RTCSessionDescription(offer));

myConnection.createAnswer(function (answer) {

myConnection.setLocalDescription(answer);

send({

type: "answer",

answer: answer

});

}, function (error) {

alert("oops...error");

});

}

//when another user answers to our offer

function onAnswer(answer) {

myConnection.setRemoteDescription(new RTCSessionDescription(answer));

}

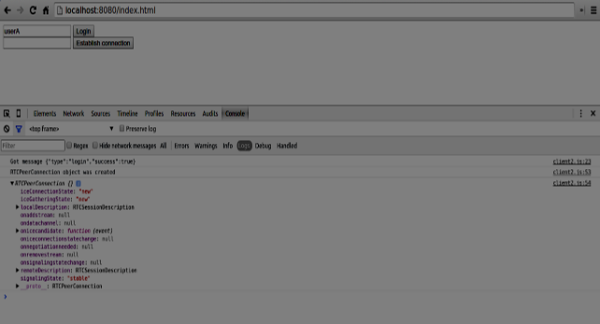
//when we got ice candidate from another user

function onCandidate(candidate) {

myConnection.addIceCandidate(new RTCIceCandidate(candidate));

}

Você pode ver que quando um usuário clica no botão "Estabelecer conexão" o aplicativo faz uma oferta SDP para o outro par. Nós também definir *onAnswer* e manipuladores *onCandidate.* Atualizar sua página, abri-lo em duas guias, faça o login com dois usuários e tentar estabelecer uma conexão entre eles. Você deverá ver a seguinte saída do console -



Agora a conexão peer-to-peer é estabelecida. Nos próximos tutoriais, vamos adicionar fluxos de vídeo e áudio, bem como apoio de chat de texto.

WebRTC - RTCDataChannel APIs

WebRTC não só é bom a transferir fluxos de áudio e vídeo, mas qualquer dado arbitrário que possa ter. Este é o lugar onde o objeto RTCDataChannel entra em jogo.

RTCDataChannel API

propriedades

* + RTCDataChannel.label (somente leitura) - Retorna uma string contendo o nome do canal de dados.
  + RTCDataChannel.ordered (somente leitura) - Retorna true se a ordem de entrega das mensagens é garantida ou falso se não for garantida.
  + RTCDataChannel.protocol (somente leitura) - Retorna um nome subprotocol string contendo usado para este canal.
  + RTCDataChannel.id (somente leitura) - Retorna uma identificação única para o canal que é definido na criação do objeto RTCDataChannel.
  + RTCDataChannel.readyState (somente leitura) - Retorna o enum RTCDataChannelState representando o estado da conexão. Os valores possíveis -
    - conectando - Indica que a conexão ainda não está ativa. Este é o estado inicial.
    - aberta - Indica que a conexão está sendo executado.
    - fechamento - Indica que a conexão está em processo de desligamento. As mensagens armazenadas em cache estão em processo de serem enviados ou recebidos, mas nenhuma tarefa recém-criada está aceitando.
    - fechado - Indica que a conexão não pôde ser estabelecida ou foi desligado.
  + RTCDataChannel.bufferedAmount (somente leitura) - Retorna a quantidade de bytes que foram enfileiradas para o envio. Esta é a quantidade de dados que não tenha sido ainda enviado através RTCDataChannel.send ().
  + RTCDataChannel.bufferedAmountLowThreshold - Retorna o número de bytes em que o RTCDataChannel.bufferedAmount é levado até um preço tão baixo. Quando o RTCDataChannel.bufferedAmount diminui abaixo deste limiar, o evento bufferedamountlow é acionado.
  + RTCDataChannel.binaryType - Retorna o tipo de dados binários transmitidos pela conexão. Pode ser "blob" ou "arraybuffer".
  + RTCDataChannel.maxPacketLifeType (somente leitura) - Retorna um curto não assinado que indica o comprimento em milissegundos da janela quando Messaging está indo no modo não confiável.
  + RTCDataChannel.maxRetransmits (somente leitura) - Retorna um curto não assinado que indica o número máximo de vezes que um canal irá retransmitir os dados se não for entregue.
  + RTCDataChannel.negotiated (somente leitura) - Retorna um booleano que indica se o canal foi negociado pelo usuário-agente, ou pela aplicação.
  + RTCDataChannel.reliable (somente leitura) - Retorna um booleano que indica da conexão pode enviar mensagens no modo não confiável.
  + RTCDataChannel.stream (somente leitura) - Sinônimo de RTCDataChannel.id

Manipuladores de eventos

* + RTCDataChannel.onopen - Este manipulador de eventos é chamado quando o evento aberto é acionado. Este caso é enviada quando a ligação de dados tenha sido estabelecida.
  + RTCDataChannel.onmessage - Este manipulador de eventos é chamado quando o evento de mensagem é acionado. O evento é enviado quando uma mensagem esteja disponível no canal de dados.
  + RTCDataChannel.onbufferedamountlow - Este manipulador de eventos é chamado quando o evento bufferedamoutlow é acionado. Este evento é enviado quando RTCDataChannel.bufferedAmount diminui abaixo da propriedade RTCDataChannel.bufferedAmountLowThreshold.
  + RTCDataChannel.onclose - Este manipulador de eventos é chamado quando o evento close é acionado. Este evento é enviado quando a conexão de dados foi fechada.
  + RTCDataChannel.onerror - Este manipulador de eventos é chamado quando o evento de erro é acionado. Este evento é enviado quando um erro foi encontrado.

Métodos

* + RTCDataChannel.close () - Fecha o canal de dados.
  + RTCDataChannel.send () - envia os dados no parâmetro através do canal. Os dados podem ser uma bolha, uma string, um ArrayBuffer ou um ArrayBufferView.

WebRTC - Envio de mensagens

Agora vamos criar um exemplo simples. Em primeiro lugar, execute o servidor de sinalização que criamos no tutorial "servidor de sinalização" via "servidor do nó".

Haverá três entradas de texto na página, uma para um início de sessão, uma para um nome de usuário e uma para a mensagem que deseja enviar para o outro ponto. Crie um arquivo *index.html* e adicione o seguinte código -

<html lang = "en">

<head>

<meta charset = "utf-8" />

</head>

<body>

<div>

<input type = "text" id = "loginInput" />

<button id = "loginBtn">Login</button>

</div>

<div>

<input type = "text" id = "otherUsernameInput" />

<button id = "connectToOtherUsernameBtn">Establish connection</button>

</div>

<div>

<input type = "text" id = "msgInput" />

<button id = "sendMsgBtn">Send text message</button>

</div>

<script src = "client.js"></script>

</body>

</html>

Nós também adicionamos três botões para login, estabelecer uma conexão e enviar uma mensagem. Agora crie um arquivo *Client.js* e adicione o seguinte código -

var connection = new WebSocket('ws://localhost:9090');

var name = "";

var loginInput = document.querySelector('#loginInput');

var loginBtn = document.querySelector('#loginBtn');

var otherUsernameInput = document.querySelector('#otherUsernameInput');

var connectToOtherUsernameBtn = document.querySelector('#connectToOtherUsernameBtn');

var msgInput = document.querySelector('#msgInput');

var sendMsgBtn = document.querySelector('#sendMsgBtn');

var connectedUser, myConnection, dataChannel;

//when a user clicks the login button

loginBtn.addEventListener("click", function(event) {

name = loginInput.value;

if(name.length > 0) {

send({

type: "login",

name: name

});

}

});

//handle messages from the server

connection.onmessage = function (message) {

console.log("Got message", message.data);

var data = JSON.parse(message.data);

switch(data.type) {

case "login":

onLogin(data.success);

break;

case "offer":

onOffer(data.offer, data.name);

break;

case "answer":

onAnswer(data.answer);

break;

case "candidate":

onCandidate(data.candidate);

break;

default:

break;

}

};

//when a user logs in

function onLogin(success) {

if (success === false) {

alert("oops...try a different username");

} else {

//creating our RTCPeerConnection object

var configuration = {

"iceServers": [{ "url": "stun:stun.1.google.com:19302" }]

};

myConnection = new webkitRTCPeerConnection(configuration, {

optional: [{RtpDataChannels: true}]

});

console.log("RTCPeerConnection object was created");

console.log(myConnection);

//setup ice handling

//when the browser finds an ice candidate we send it to another peer

myConnection.onicecandidate = function (event) {

if (event.candidate) {

send({

type: "candidate",

candidate: event.candidate

});

}

};

openDataChannel();

}

};

connection.onopen = function () {

console.log("Connected");

};

connection.onerror = function (err) {

console.log("Got error", err);

};

// Alias for sending messages in JSON format

function send(message) {

if (connectedUser) {

message.name = connectedUser;

}

connection.send(JSON.stringify(message));

};

Você pode ver que nós estabelecer uma conexão de soquete para o nosso servidor de sinalização. Quando um usuário clica no botão de login a aplicação envia seu nome de usuário para o servidor. Se o login for bem sucedido o aplicativo cria o manipulador *onicecandidate*objeto e configuração *RTCPeerConnection* que envia todos os icecandidates encontrados para o outro par. Ele também executa a função openDataChannel () que cria um dataChannel. Observe que ao criar o objeto RTCPeerConnection o segundo argumento no construtor opcionais: [{RtpDataChannels: true}] é obrigatória se você estiver usando o Chrome ou Opera. O próximo passo é criar uma oferta para os outros pares. Adicione o seguinte código ao seu *Client.js* file-

//setup a peer connection with another user

connectToOtherUsernameBtn.addEventListener("click", function () {

var otherUsername = otherUsernameInput.value;

connectedUser = otherUsername;

if (otherUsername.length > 0) {

//make an offer

myConnection.createOffer(function (offer) {

console.log();

send({

type: "offer",

offer: offer

});

myConnection.setLocalDescription(offer);

}, function (error) {

alert("An error has occurred.");

});

}

});

//when somebody wants to call us

function onOffer(offer, name) {

connectedUser = name;

myConnection.setRemoteDescription(new RTCSessionDescription(offer));

myConnection.createAnswer(function (answer) {

myConnection.setLocalDescription(answer);

send({

type: "answer",

answer: answer

});

}, function (error) {

alert("oops...error");

});

}

//when another user answers to our offer

function onAnswer(answer) {

myConnection.setRemoteDescription(new RTCSessionDescription(answer));

}

//when we got ice candidate from another user

function onCandidate(candidate) {

myConnection.addIceCandidate(new RTCIceCandidate(candidate));

}

Você pode ver que quando um usuário clica no botão "Estabelecer conexão" o aplicativo faz uma oferta SDP para o outro par. Nós também definir *onAnswer* e manipuladores *onCandidate.* Finalmente, vamos implementar a função *openDataChannel ()* que cria nossos dataChannel. Adicione o seguinte código ao seu arquivo *Client.js* -

//creating data channel

function openDataChannel() {

var dataChannelOptions = {

reliable:true

};

dataChannel = myConnection.createDataChannel("myDataChannel", dataChannelOptions);

dataChannel.onerror = function (error) {

console.log("Error:", error);

};

dataChannel.onmessage = function (event) {

console.log("Got message:", event.data);

};

}

//when a user clicks the send message button

sendMsgBtn.addEventListener("click", function (event) {

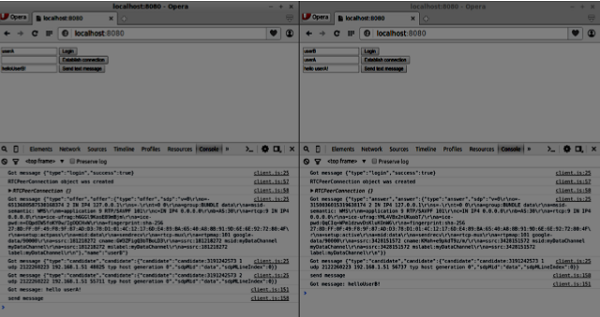
console.log("send message");

var val = msgInput.value;

dataChannel.send(val);

});

Aqui criamos o dataChannel para a nossa ligação e adicionar o manipulador de eventos para o botão "enviar mensagem". Agora abrir esta página em duas guias, faça o login com dois usuários, estabelecer uma ligação e tentar enviar mensagens. Você deve vê-los na saída do console. Note-se que o exemplo acima é testado no Opera.



Agora você pode ver que RTCDataChannel é extremamente poderoso parte da API WebRTC. Há uma série de outros casos de uso para esse objeto, como jogos peer-to-peer ou compartilhamento de arquivos baseado em torrent.

WebRTC - Sinalização

A maioria dos aplicativos WebRTC não são apenas ser capaz de se comunicar através de vídeo e áudio. Eles precisam de muitos outros recursos. Neste capítulo, vamos construir um servidor de base de sinalização.

Sinalização e Negociação

Para se conectar a outro usuário você deve saber onde ele está localizado na Web. O endereço IP do seu dispositivo permite que os dispositivos habilitados para a Internet para enviar dados diretamente entre si. O objecto *RTCPeerConnection* é responsável por isso. Assim como dispositivos de saber como encontrar uns aos outros através da Internet, eles começam trocando dados sobre quais protocolos e codecs cada dispositivo suporta.

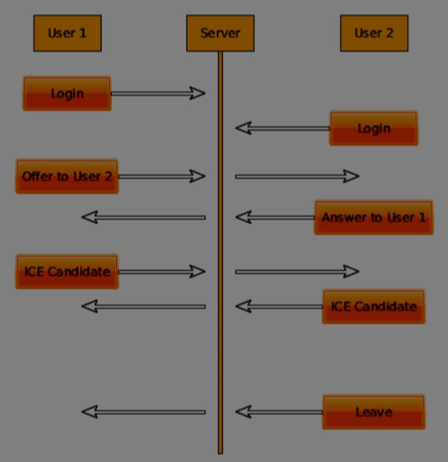
Para comunicar com outro usuário que você simplesmente precisa para trocar informações de contato eo resto será feito por WebRTC. O processo de conexão com o outro usuário também é conhecido como sinalização e negociação. Ele consiste em algumas etapas -

* + Criar uma lista de potenciais candidatos para uma conexão ponto.
  + O usuário ou um aplicativo seleciona um usuário para fazer uma conexão com.
  + A camada de sinalização notifica outro usuário que alguém deseja se conectar a ele. Ele pode aceitar ou recusar.
  + O primeiro utilizador é notificado da aceitação da oferta.
  + O primeiro usuário inicia *RTCPeerConnection* com outro usuário.
  + Ambos os usuários trocar de software e hardware informações através do servidor de sinalização.
  + Ambos os usuários trocar informações de localização.
  + A ligação bem sucedida ou não.

A especificação WebRTC não contém quaisquer normas sobre o intercâmbio de informações. Portanto, tenha em mente que o acima é apenas um exemplo de como a sinalização pode acontecer. Você pode usar qualquer protocolo ou tecnologia que você gosta.

Construindo o Servidor

O servidor que está indo para construir será capaz de se conectar dois usuários em conjunto que não estão localizados no mesmo computador. Vamos criar o nosso próprio mecanismo de sinalização. O nosso servidor de sinalização permitirá que um usuário para chamar outro. Depois que um usuário chamado outro, o servidor passa os candidatos oferta, resposta, gelo entre eles e configurar uma conexão WebRTC.



O diagrama acima é o fluxo de mensagens entre os usuários quando utilizar o servidor de sinalização. Em primeiro lugar, cada utilizador regista com o servidor. Em nosso caso, este será um simples nome de usuário string. Depois que os usuários registrados, eles são capazes de chamar uns aos outros. Usuário 1 faz uma oferta com o identificador do usuário que deseja chamar. O outro usuário deve respostas. Finalmente, os candidatos ICE são enviadas entre usuários até que eles possam fazer uma conexão.

Para criar uma conexão WebRTC clientes tem que ser capaz de transferir mensagens sem usar uma conexão ponto WebRTC. Este é o lugar onde vamos usar WebSockets HTML5 - uma conexão bidirecional tomada entre dois pontos - um servidor web e um navegador web. Agora vamos começar a usar a biblioteca WebSocket. Criar o arquivo *server.js* e insira o seguinte código -

//require our websocket library

var WebSocketServer = require('ws').Server;

//creating a websocket server at port 9090

var wss = new WebSocketServer({port: 9090});

//when a user connects to our sever

wss.on('connection', function(connection) {

console.log("user connected");

//when server gets a message from a connected user

connection.on('message', function(message){

console.log("Got message from a user:", message);

});

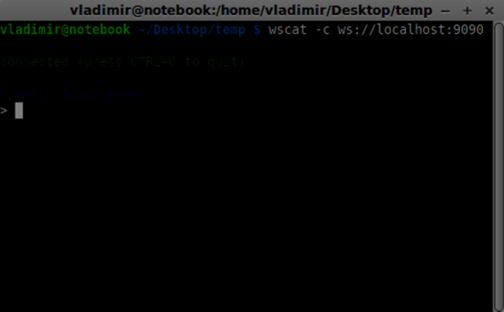
connection.send("Hello from server");

});

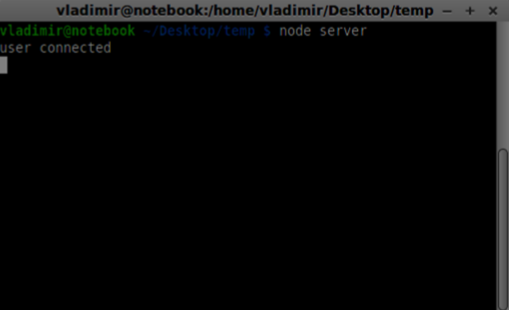
A primeira linha exige a biblioteca WebSocket que já instalado. Em seguida, criar um servidor de soquete na porta 9090. Em seguida, vamos ouvir o evento *de conexão.* Este código será executado quando um usuário faz uma conexão WebSocket para o servidor. Em seguida, ouvir as mensagens enviadas pelo usuário. Por fim, enviar uma resposta para o usuário conectado dizendo "Olá do servidor".

Agora executar o *servidor de nó* e o servidor deve começar a ouvir para conexões de soquete.

Para testar o nosso servidor, usaremos o utilitário *wscat* que também já instalado. Esta ferramenta ajuda na ligação diretamente para o servidor WebSocket e testar comandos. Executar o nosso servidor em uma janela de terminal, em seguida, abrir outra e executar os *ws wscat -c: // localhost: 9090* comando. Você deve ver o seguinte no lado do cliente -



O servidor também deve registrar o usuário conectado -



Registro do usuário

No nosso servidor de sinalização, vamos usar um nome de usuário baseada em cadeia para cada conexão por isso sabemos para onde enviar mensagens. Vamos mudar o nosso manipulador de *conexão* um pouco -

connection.on('message', function(message) {

var data;

//accepting only JSON messages

try {

data = JSON.parse(message);

} catch (e) {

console.log("Invalid JSON");

data = {};

}

});

Desta forma, aceitar apenas mensagens JSON. Em seguida, é necessário armazenar todos os usuários conectados em algum lugar. Nós vamos usar um objeto simples Javascript para ele. Alterar o topo do nosso arquivo -

//require our websocket library

var WebSocketServer = require('ws').Server;

//creating a websocket server at port 9090

var wss = new WebSocketServer({port: 9090});

//all connected to the server users

var users = {};

Estamos indo para adicionar um campo de *tipo* para cada mensagem que vem do cliente. Por exemplo, se um usuário quer acessar, ele envia a mensagem Tipo de *login.* Vamos defini-lo -

connection.on('message', function(message){

var data;

//accepting only JSON messages

try {

data = JSON.parse(message);

} catch (e) {

console.log("Invalid JSON");

data = {};

}

//switching type of the user message

switch (data.type) {

//when a user tries to login

case "login":

console.log("User logged:", data.name);

//if anyone is logged in with this username then refuse

if(users[data.name]) {

sendTo(connection, {

type: "login",

success: false

});

} else {

//save user connection on the server

users[data.name] = connection;

connection.name = data.name;

sendTo(connection, {

type: "login",

success: true

});

}

break;

default:

sendTo(connection, {

type: "error",

message: "Command no found: " + data.type

});

break;

}

});

Se o usuário envia uma mensagem com o tipo de *login,* nós -

* + Verifique se alguém já tenha iniciado sessão com esse nome de usuário
  + Se assim for, então informar ao usuário que ele não tenha login com sucesso
  + Se ninguém está usando esse nome de usuário, podemos adicionar nome de usuário como uma chave para o objeto de conexão.
  + Se um comando não é reconhecido enviamos um erro.

O código a seguir é uma função auxiliar para envio de mensagens para uma conexão. Adicioná-lo ao arquivo *server.js* -

function sendTo(connection, message) {

connection.send(JSON.stringify(message));

}

A função acima assegura que todas as nossas mensagens são enviadas no formato JSON.

Quando o usuário desconecta devemos limpar sua conexão. Podemos excluir o usuário quando o evento *close* é acionado. Adicione o seguinte código ao manipulador de *conexão* -

connection.on("close", function() {

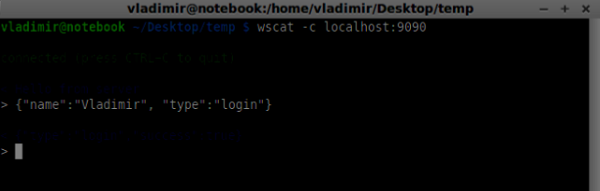
if(connection.name) {

delete users[connection.name];

}

});

Agora vamos testar nosso servidor com o comando login. Tenha em mente que todas as mensagens devem ser codificado no formato JSON. Executar o nosso servidor e tento fazer o login. Você deve ver algo como isto -



Fazer uma chamada

Depois do login do usuário quer chamar outro. Ele deve fazer uma *oferta* para outro usuário para alcançá-lo. Adicione o manipulador de *oferta* -

case "offer":

//for ex. UserA wants to call UserB

console.log("Sending offer to: ", data.name);

//if UserB exists then send him offer details

var conn = users[data.name];

if(conn != null){

//setting that UserA connected with UserB

connection.otherName = data.name;

sendTo(conn, {

type: "offer",

offer: data.offer,

name: connection.name

});

}

break;

Em primeiro lugar, temos a *conexão* do usuário que está tentando chamar. Se ele existir, enviamos-lhe *oferecer* mais detalhes. Nós também adicionar *Othername* para o objeto de *conexão.* Isto é feito para a simplicidade de encontrá-lo mais tarde.

Respondendo

Respondendo à resposta tem um padrão similar que foi utilizado no manipulador de *oferta.* O nosso servidor simplesmente passa por todas as mensagens como *resposta* a outro usuário. Adicione o seguinte código após a *oferta* hander -

case "answer":

console.log("Sending answer to: ", data.name);

//for ex. UserB answers UserA

var conn = users[data.name];

if(conn != null) {

connection.otherName = data.name;

sendTo(conn, {

type: "answer",

answer: data.answer

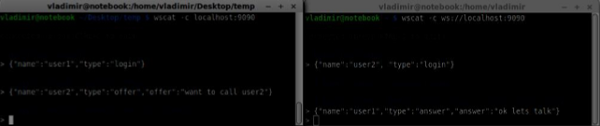
});

}

break;

Você pode ver como isso é semelhante ao processador de *oferta.* Perceba que este código segue as funções *createOffer* e *createAnswer* no objeto *RTCPeerConnection.*

Agora podemos testar o nosso mecanismo de oferta / resposta. Conectar dois clientes ao mesmo tempo e tentar fazer a oferta e resposta. Você deve ver o seguinte -



Neste exemplo, a oferta ea resposta são cadeias simples, mas em uma aplicação real que vai ser preenchido com os dados SDP.

Os candidatos ICE

A parte final é manipulação candidato ICE entre os usuários. Nós usamos a mesma técnica de passagem de mensagens entre os usuários. A principal diferença é que as mensagens de candidatos pode ocorrer várias vezes por utilizador em qualquer ordem. Adicione o manipulador *candidato* -

case "candidate":

console.log("Sending candidate to:",data.name);

var conn = users[data.name];

if(conn != null) {

sendTo(conn, {

type: "candidate",

candidate: data.candidate

});

}

break;

Ele deve funcionar de forma semelhante à *oferta* e *resposta* manipuladores.

Deixando o Connection

Para permitir que nossos usuários para se desconectar outro usuário que deve implementar a função de desligar. Ele também irá informar o servidor para apagar todas as referências do usuário. Adicione o manipulador de licença -

case "leave":

console.log("Disconnecting from", data.name);

var conn = users[data.name];

conn.otherName = null;

//notify the other user so he can disconnect his peer connection

if(conn != null) {

sendTo(conn, {

type: "leave"

});

}

break;

Isso também irá enviar a outro usuário do evento *licença* para que ele possa desligar sua conexão ponto de acordo. Nós também deve tratar o caso quando um usuário deixa cair sua conexão do servidor de sinalização. Vamos modificar nosso manipulador de *perto* -

connection.on("close", function() {

if(connection.name) {

delete users[connection.name];

if(connection.otherName) {

console.log("Disconnecting from ", connection.otherName);

var conn = users[connection.otherName];

conn.otherName = null;

if(conn != null) {

sendTo(conn, {

type: "leave"

});

}

}

}

});

Agora, se a conexão termina nossos usuários serão desconectados. O evento *close* será acionado quando um usuário fecha a janela de seu navegador enquanto ainda estão em *oferta,* *resposta* ou Estado *candidato.*

Sinalização Servidor completa

Aqui está todo o código do nosso servidor de sinalização -

//require our websocket library

var WebSocketServer = require('ws').Server;

//creating a websocket server at port 9090

var wss = new WebSocketServer({port: 9090});

//all connected to the server users

var users = {};

//when a user connects to our sever

wss.on('connection', function(connection) {

console.log("User connected");

//when server gets a message from a connected user

connection.on('message', function(message) {

var data;

//accepting only JSON messages

try {

data = JSON.parse(message);

} catch (e) {

console.log("Invalid JSON");

data = {};

}

//switching type of the user message

switch (data.type) {

//when a user tries to login

case "login":

console.log("User logged", data.name);

//if anyone is logged in with this username then refuse

if(users[data.name]) {

sendTo(connection, {

type: "login",

success: false

});

} else {

//save user connection on the server

users[data.name] = connection;

connection.name = data.name;

sendTo(connection, {

type: "login",

success: true

});

}

break;

case "offer":

//for ex. UserA wants to call UserB

console.log("Sending offer to: ", data.name);

//if UserB exists then send him offer details

var conn = users[data.name];

if(conn != null) {

//setting that UserA connected with UserB

connection.otherName = data.name;

sendTo(conn, {

type: "offer",

offer: data.offer,

name: connection.name

});

}

break;

case "answer":

console.log("Sending answer to: ", data.name);

//for ex. UserB answers UserA

var conn = users[data.name];

if(conn != null) {

connection.otherName = data.name;

sendTo(conn, {

type: "answer",

answer: data.answer

});

}

break;

case "candidate":

console.log("Sending candidate to:",data.name);

var conn = users[data.name];

if(conn != null) {

sendTo(conn, {

type: "candidate",

candidate: data.candidate

});

}

break;

case "leave":

console.log("Disconnecting from", data.name);

var conn = users[data.name];

conn.otherName = null;

//notify the other user so he can disconnect his peer connection

if(conn != null) {

sendTo(conn, {

type: "leave"

});

}

break;

default:

sendTo(connection, {

type: "error",

message: "Command not found: " + data.type

});

break;

}

});

//when user exits, for example closes a browser window

//this may help if we are still in "offer","answer" or "candidate" state

connection.on("close", function() {

if(connection.name) {

delete users[connection.name];

if(connection.otherName) {

console.log("Disconnecting from ", connection.otherName);

var conn = users[connection.otherName];

conn.otherName = null;

if(conn != null) {

sendTo(conn, {

type: "leave"

});

}

}

}

});

connection.send("Hello world");

});

function sendTo(connection, message) {

connection.send(JSON.stringify(message));

}

Assim, o trabalho é feito e nosso servidor de sinalização está pronto. Lembre-se de que fazer as coisas fora de ordem ao fazer uma conexão WebRTC pode causar problemas.

Resumo

Neste capítulo, nós construímos servidor de sinalização simples e direta. Nós caminhamos através do processo de sinalização, registro do usuário e mecanismo de oferta / resposta. Nós também implementou o envio de candidatos entre os usuários.

WebRTC - Suporte a navegadores

A Web está se movendo tão rápido e é sempre melhorar. Novos padrões são criados todos os dias. Navegadores permite que as atualizações sejam instaladas sem que o usuário nunca saber, por isso você deve manter-se com o que está acontecendo no mundo da Web e WebRTC. Aqui está um resumo do que isso é até hoje.

Suporte a navegadores

Cada navegador não tem todas as mesmas características WebRTC ao mesmo tempo. Diferentes navegadores podem estar à frente da curva, o que torna algumas características WebRTC trabalhar em um navegador, e não outra. O apoio atual para WebRTC no navegador é mostrado na figura a seguir.



Você pode verificar o status de suporte WebRTC up-to-date em [http://caniuse.com/#feat=rtcpeerconnection.](http://caniuse.com/#feat=rtcpeerconnection/index.html)

Chrome, Firefox e Opera

As versões mais recentes do Chrome, Firefox e Opera em sistemas operacionais de PC tradicionais como Mac OS X, Windows e Linux, todos WebRTC suporte out-of-the-box. E o mais importante, os engenheiros do Google Chrome e do Firefox equipes de desenvolvedores têm trabalhado em conjunto para corrigir problemas assim que estes dois navegadores pudessem se comunicar uns com os outros facilmente.

OS Android

Em sistemas operacionais Android, os aplicativos WebRTC para Chrome e Firefox deve funcionar outof-the-box. Eles são capazes de funcionar com outros navegadores após versão Sandwich Android Ice Cream (4.0). Isto é devido à partilha de código entre versões desktop e móveis.

maçã

A Apple ainda não fez qualquer anúncio sobre seus planos para apoiar WebRTC no Safari no OS X. Uma das possíveis soluções alternativas para híbridos iOS nativas aplicações para o OS para incorporar o código WebRTC diretamente no aplicativo e carregar este app em um WebView.

Internet Explorer

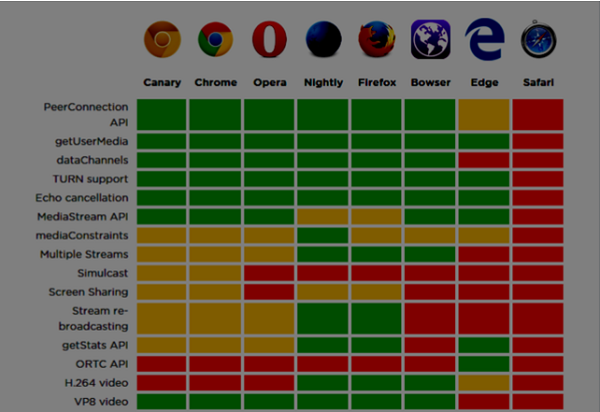
A Microsoft não suporta WebRTC em desktops. Mas eles têm oficialmente confirmou que eles estão indo para implementar ORTC (objeto em tempo real Communications) em futuras versões do IE (Edge). Eles não estão a planear apoiar WebRTC 1.0. Eles rotulou seu ORTC como WebRTC 1.1, embora seja apenas uma melhoria da comunidade e não o padrão oficial. Recentemente eles adicionaram o apoio ORTC para a última versão do Microsoft Borda. Você pode aprender mais em [https://blogs.windows.com/msedgedev/2015/09/18/ortc-api-is-now-available-in-microsoftedge/.](http://blogs.windows.com/msedgedev/2015/09/18/ortc-api-is-now-available-in-microsoft-edge/index.html)

Resumo

Observe que WebRTC é um conjunto de APIs e protocolos, nem uma única API. O apoio para cada um deles está a desenvolver em diferentes navegadores e sistemas operacionais em um nível diferente. Uma ótima maneira de verificar o mais recente nível de apoio é através [http://canisue.com.](http://canisue.com/index.html) Ele rastreia adoção de APIs modernas em vários navegadores. Você também pode encontrar as últimas informações sobre o navegador suporta, bem como demos WebRTC em [http://www.webrtc.org,](http://www.webrtc.org/index.html) que é apoiado pela Mozilla, Google e Opera.

WebRTC - Suporte Móvel

No mundo móvel, o apoio WebRTC não está no mesmo nível que ele está em desktops. Os dispositivos móveis têm sua própria maneira, de modo WebRTC também é algo diferente nas plataformas móveis.



Ao desenvolver um aplicativo WebRTC para desktop, nós consideramos usando o Chrome, Firefox ou Opera. Todos eles suportam WebRTC fora da caixa. Em geral, você só precisa de um navegador e não se preocupar com o hardware do desktop.

No mundo móvel, existem três modos possíveis para WebRTC hoje -

* + O aplicativo nativo
  + A aplicação do browser
  + O navegador nativo

andróide

Em 2013, o navegador web Firefox para Android foi apresentado com o apoio WebRTC fora da caixa. Agora você pode fazer chamadas de vídeo em dispositivos Android usando o navegador móvel Firefox.

Ele tem três componentes principais WebRTC -

* + PeerConnection - permite que as chamadas entre os navegadores
  + getUserMedia - fornece acesso à câmera e microfone
  + DataChannels - fornece a transferência de dados peer-to-peer

Google Chrome para Android fornece suporte WebRTC também. Como você já percebeu, as características mais interessantes geralmente aparecem pela primeira vez no Chrome.

No ano passado, o navegador móvel Opera apareceu com o apoio WebRTC. Assim para Android você tem Chrome, Firefox e Opera. Outros navegadores não suportam WebRTC.

iOS

Infelizmente, WebRTC não é suportado no iOS agora. Embora WebRTC funciona bem em Mac ao usar o Firefox, Opera ou Chrome, ele não é suportado no iOS.

Hoje em dia, a sua aplicação WebRTC não irá funcionar em dispositivos móveis da Apple fora da caixa. Mas há um navegador - Bowser. É um navegador web desenvolvido pela Ericsson e suporta WebRTC fora da caixa. Você pode verificar a sua homepage em [http://www.openwebrtc.org/bowser/.](http://www.openwebrtc.org/bowser/index.html)

Hoje, é a única maneira amigável para apoiar o seu pedido WebRTC no iOS. Outra maneira é desenvolver um aplicativo nativo si mesmo.

Windows Phones

A Microsoft não suporta WebRTC em plataformas móveis. Mas eles têm oficialmente confirmou que eles estão indo para implementar ORTC (objeto em tempo real Communications) em futuras versões do IE. Eles não estão a planear apoiar WebRTC 1.0. Eles rotulou seu ORTC como WebRTC 1.1, embora seja apenas uma melhoria da comunidade e não o padrão oficial.

Assim, os usuários de hoje janela de telefone não pode usar aplicações WebRTC e não há nenhuma maneira de vencer esta situação.

Amora preta

aplicações WebRTC não são suportados no Blackberry, quer, de qualquer forma.

Usando um navegador nativo WebRTC

O caso mais conveniente e confortável para os usuários a utilizar WebRTC está usando o navegador nativo do dispositivo. Neste caso, o dispositivo está pronto para trabalhar todas as configurações adicionais.

Hoje, apenas os dispositivos Android que estão versão 4 ou superior fornecem esse recurso. A Apple ainda não mostra qualquer atividade com suporte WebRTC. Assim, os usuários do Safari não pode usar aplicações WebRTC. A Microsoft também não introduzi-lo no Windows Phone 8.

Usando WebRTC via aplicativos de navegador

Isso significa usar a aplicativos de terceiros (navegadores da web não-nativos), a fim de fornecer as características WebRTC. Por enquanto, não são dois desses aplicativos de terceiros. Bowser, que é a única maneira de trazer recursos WebRTC para o dispositivo iOS e Opera, que é uma boa alternativa para a plataforma Android. O resto dos navegadores móveis disponíveis não suportam WebRTC.

Aplicações móveis nativas

Como você pode ver, WebRTC não tem um grande apoio no mundo móvel ainda. Assim, uma das possíveis soluções é o de desenvolver um aplicações nativas que utilizam a API WebRTC. Mas não é a melhor escolha porque o recurso WebRTC principal é uma solução multi-plataforma. De qualquer forma, em alguns casos, esta é a única maneira, porque um aplicativo nativo pode utilizar funções ou características específicas do dispositivo que não são suportadas pelos navegadores HTML5.

Restringindo Video Stream para dispositivos móveis e computadores

O primeiro parâmetro da API *getUserMedia* espera um objeto de chaves e valores que dizem ao navegador como processar córregos. Você pode verificar o conjunto completo de restrições na [https://tools.ietf.org/html/draft-alvestrand-constraints-resolution-03.](http://tools.ietf.org/html/draft-alvestrand-constraints-resolution-03/index.html) Você pode configurar o aspecto video de ração, frameRate, e outros parâmetros opcionais.

Suporte a dispositivos móveis é uma das maiores dores porque os dispositivos móveis têm espaço da tela limitada, juntamente com recursos limitados. Você pode querer o dispositivo móvel para capturar apenas a resolução 480x320 ou streaming de vídeo menor para economizar energia e largura de banda. Usando a seqüência do agente do usuário no navegador é uma boa maneira de testar se o usuário estiver em um dispositivo móvel ou não. Vamos ver um exemplo. Criar o arquivo *index.html* -

<!DOCTYPE html>

<html lang = "en">

<head>

<meta charset = "utf-8" />

</head>

<body>

<video autoplay></video>

<script src = "client.js"></script>

</body>

</html>

Em seguida, crie o seguinte arquivo *Client.js* -

//constraints for desktop browser

var desktopConstraints = {

video: {

mandatory: {

maxWidth:800,

maxHeight:600

}

},

audio: true

};

//constraints for mobile browser

var mobileConstraints = {

video: {

mandatory: {

maxWidth: 480,

maxHeight: 320,

}

},

audio: true

}

//if a user is using a mobile browser

if(/Android|iPhone|iPad/i.test(navigator.userAgent)) {

var constraints = mobileConstraints;

} else {

var constraints = desktopConstraints;

}

function hasUserMedia() {

//check if the browser supports the WebRTC

return !!(navigator.getUserMedia || navigator.webkitGetUserMedia ||

navigator.mozGetUserMedia);

}

if (hasUserMedia()) {

navigator.getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia ||

navigator.mozGetUserMedia;

//enabling video and audio channels

navigator.getUserMedia(constraints, function (stream) {

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

//inserting our stream to the video tag

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

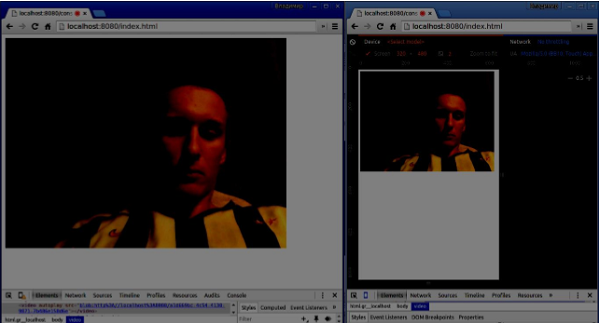
}, function (err) {});

} else {

alert("WebRTC is not supported");

}

Executar o servidor web utilizando o comando *estático* e abrir a página. Você deve ver que é 800x600. Em seguida, abra essa página em uma janela de visualização móvel usando ferramentas de cromo e verificar a resolução. Deve ser 480x320.



As restrições são a maneira mais fácil para aumentar o desempenho do seu aplicativo WebRTC.

Resumo

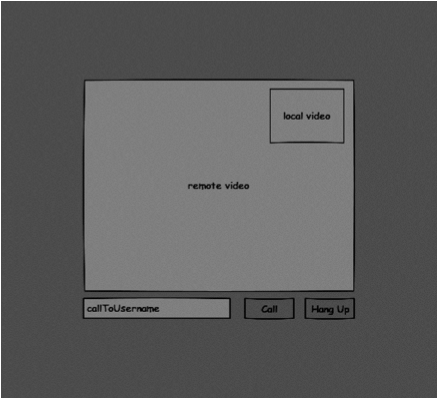
Neste capítulo, nós aprendemos sobre os problemas que podem ocorrer quando o desenvolvimento de aplicações WebRTC para dispositivos móveis. Descobrimos diferentes limitações de apoiar a API WebRTC em plataformas móveis. Nós também lançou um aplicativo de demonstração, onde vamos definir restrições diferentes para desktop e navegadores móveis.

WebRTC - Vídeo de demonstração

Neste capítulo, vamos construir uma aplicação cliente que permite que dois usuários em dispositivos separados para se comunicar usando WebRTC. Nossa aplicação terá duas páginas. Um para login e outro para chamar outro usuário.



As duas páginas serão as tags *div.* A maioria de entrada é feita através de manipuladores de eventos simples.



sinalizando Servidor

Para criar uma conexão WebRTC clientes tem que ser capaz de transferir mensagens sem usar uma conexão ponto WebRTC. Este é o lugar onde vamos usar WebSockets HTML5 - uma conexão bidirecional tomada entre dois pontos - um servidor web e um navegador web. Agora vamos começar a usar a biblioteca WebSocket. Criar o arquivo *server.js* e insira o seguinte código -

//require our websocket library

var WebSocketServer = require('ws').Server;

//creating a websocket server at port 9090

var wss = new WebSocketServer({port: 9090});

//when a user connects to our sever

wss.on('connection', function(connection) {

console.log("user connected");

//when server gets a message from a connected user

connection.on('message', function(message) {

console.log("Got message from a user:", message);

});

connection.send("Hello from server");

});

A primeira linha exige a biblioteca WebSocket que já instalado. Em seguida, criar um servidor de soquete na porta 9090. Em seguida, vamos ouvir o evento *de conexão.* Este código será executado quando um usuário faz uma conexão WebSocket para o servidor. Em seguida, ouvir as mensagens enviadas pelo usuário. Por fim, enviar uma resposta para o usuário conectado dizendo "Olá do servidor".

No nosso servidor de sinalização, vamos usar um nome de usuário baseada em cadeia para cada conexão por isso sabemos para onde enviar mensagens. Vamos mudar o nosso *manipulador de* conexão um pouco -

connection.on('message', function(message) {

var data;

//accepting only JSON messages

try {

data = JSON.parse(message);

} catch (e) {

console.log("Invalid JSON");

data = {};

}

});

Desta forma, aceitar apenas mensagens JSON. Em seguida, é necessário armazenar todos os usuários conectados em algum lugar. Nós vamos usar um objeto simples Javascript para ele. Alterar o topo do nosso arquivo -

//require our websocket library

var WebSocketServer = require('ws').Server;

//creating a websocket server at port 9090

var wss = new WebSocketServer({port: 9090});

//all connected to the server users

var users = {};

Estamos indo para adicionar um campo de *tipo* para cada mensagem que vem do cliente. Por exemplo, se um usuário quer acessar, ele envia a mensagem Tipo de *login.* Vamos defini-lo -

connection.on('message', function(message) {

var data;

//accepting only JSON messages

try {

data = JSON.parse(message);

} catch (e) {

console.log("Invalid JSON");

data = {};

}

//switching type of the user message

switch (data.type) {

//when a user tries to login

case "login":

console.log("User logged:", data.name);

//if anyone is logged in with this username then refuse

if(users[data.name]) {

sendTo(connection, {

type: "login",

success: false

});

} else {

//save user connection on the server

users[data.name] = connection;

connection.name = data.name;

sendTo(connection, {

type: "login",

success: true

});

}

break;

default:

sendTo(connection, {

type: "error",

message: "Command no found: " + data.type

});

break;

}

});

Se o usuário envia uma mensagem com o tipo de *login,* nós -

* + Verifique se alguém já tenha iniciado sessão com esse nome de usuário
  + Se assim for, então informar ao usuário que ele não tenha login com sucesso
  + Se ninguém está usando esse nome de usuário, podemos adicionar nome de usuário como uma chave para o objeto de conexão.
  + Se um comando não é reconhecido enviamos um erro.

O código a seguir é uma função auxiliar para envio de mensagens para uma conexão. Adicioná-lo ao arquivo *server.js* -

function sendTo(connection, message) {

connection.send(JSON.stringify(message));

}

Quando o usuário desconecta devemos limpar sua conexão. Podemos excluir o usuário quando o evento *close* é acionado. Adicione o seguinte código ao manipulador de *conexão* -

connection.on("close", function() {

if(connection.name) {

delete users[connection.name];

}

});

Depois do login do usuário quer chamar outro. Ele deve fazer uma *oferta* para outro usuário para alcançá-lo. Adicione o manipulador de *oferta* -

case "offer":

//for ex. UserA wants to call UserB

console.log("Sending offer to: ", data.name);

//if UserB exists then send him offer details

var conn = users[data.name];

if(conn != null) {

//setting that UserA connected with UserB

connection.otherName = data.name;

sendTo(conn, {

type: "offer",

offer: data.offer,

name: connection.name

});

}

break;

Em primeiro lugar, temos a *conexão* do usuário que está tentando chamar. Se ele existir, enviamos-lhe *oferecer* mais detalhes. Nós também adicionar *Othername* para o objeto de *conexão.* Isto é feito para a simplicidade de encontrá-lo mais tarde.

Respondendo à resposta tem um padrão similar que foi utilizado no manipulador de *oferta.* O nosso servidor simplesmente passa por todas as mensagens como *resposta* a outro usuário. Adicione o seguinte código após o manipulador de *oferta* -

case "answer":

console.log("Sending answer to: ", data.name);

//for ex. UserB answers UserA

var conn = users[data.name];

if(conn != null) {

connection.otherName = data.name;

sendTo(conn, {

type: "answer",

answer: data.answer

});

}

break;

A parte final é manipulação candidato ICE entre os usuários. Nós usamos a mesma técnica de passagem de mensagens entre os usuários. A principal diferença é que as mensagens de candidatos pode ocorrer várias vezes por utilizador em qualquer ordem. Adicione o manipulador *candidato* -

case "candidate":

console.log("Sending candidate to:",data.name);

var conn = users[data.name];

if(conn != null) {

sendTo(conn, {

type: "candidate",

candidate: data.candidate

});

}

break;

Para permitir que nossos usuários para se desconectar outro usuário que deve implementar a função de desligar. Ele também irá informar o servidor para apagar todas as referências do usuário. Adicione o manipulador de *licença* -

case "leave":

console.log("Disconnecting from", data.name);

var conn = users[data.name];

conn.otherName = null;

//notify the other user so he can disconnect his peer connection

if(conn != null) {

sendTo(conn, {

type: "leave"

});

}

break;

Isso também irá enviar a outro usuário do evento *licença* para que ele possa desligar sua conexão ponto de acordo. Nós também deve tratar o caso quando um usuário deixa cair sua conexão do servidor de sinalização. Vamos modificar nosso manipulador de *perto* -

connection.on("close", function() {

if(connection.name) {

delete users[connection.name];

if(connection.otherName) {

console.log("Disconnecting from ", connection.otherName);

var conn = users[connection.otherName];

conn.otherName = null;

if(conn != null) {

sendTo(conn, {

type: "leave"

});

}

}

}

});

A seguir está o código completo de nosso servidor de sinalização -

//require our websocket library

var WebSocketServer = require('ws').Server;

//creating a websocket server at port 9090

var wss = new WebSocketServer({port: 9090});

//all connected to the server users

var users = {};

//when a user connects to our sever

wss.on('connection', function(connection) {

console.log("User connected");

//when server gets a message from a connected user

connection.on('message', function(message) {

var data;

//accepting only JSON messages

try {

data = JSON.parse(message);

} catch (e) {

console.log("Invalid JSON");

data = {};

}

//switching type of the user message

switch (data.type) {

//when a user tries to login

case "login":

console.log("User logged", data.name);

//if anyone is logged in with this username then refuse

if(users[data.name]) {

sendTo(connection, {

type: "login",

success: false

});

} else {

//save user connection on the server

users[data.name] = connection;

connection.name = data.name;

sendTo(connection, {

type: "login",

success: true

});

}

break;

case "offer":

//for ex. UserA wants to call UserB

console.log("Sending offer to: ", data.name);

//if UserB exists then send him offer details

var conn = users[data.name];

if(conn != null) {

//setting that UserA connected with UserB

connection.otherName = data.name;

sendTo(conn, {

type: "offer",

offer: data.offer,

name: connection.name

});

}

break;

case "answer":

console.log("Sending answer to: ", data.name);

//for ex. UserB answers UserA

var conn = users[data.name];

if(conn != null) {

connection.otherName = data.name;

sendTo(conn, {

type: "answer",

answer: data.answer

});

}

break;

case "candidate":

console.log("Sending candidate to:",data.name);

var conn = users[data.name];

if(conn != null) {

sendTo(conn, {

type: "candidate",

candidate: data.candidate

});

}

break;

case "leave":

console.log("Disconnecting from", data.name);

var conn = users[data.name];

conn.otherName = null;

//notify the other user so he can disconnect his peer connection

if(conn != null) {

sendTo(conn, {

type: "leave"

});

}

break;

default:

sendTo(connection, {

type: "error",

message: "Command not found: " + data.type

});

break;

}

});

//when user exits, for example closes a browser window

//this may help if we are still in "offer","answer" or "candidate" state

connection.on("close", function() {

if(connection.name) {

delete users[connection.name];

if(connection.otherName) {

console.log("Disconnecting from ", connection.otherName);

var conn = users[connection.otherName];

conn.otherName = null;

if(conn != null) {

sendTo(conn, {

type: "leave"

});

}

}

}

});

connection.send("Hello world");

});

function sendTo(connection, message) {

connection.send(JSON.stringify(message));

}

Aplicação cliente

Uma maneira de testar esta aplicação está abrindo duas abas do navegador e tentar chamar uns aos outros.

Primeiro de tudo, precisamos de instalar a biblioteca *de bootstrap.* Bootstrap é um framework de interface para desenvolvimento de aplicações web. Você pode aprender mais em [http://getbootstrap.com/.](http://getbootstrap.com/index.html) Crie uma pasta chamada, por exemplo, "videochat". Esta será a nossa pasta raiz do aplicativo. Dentro desta pasta crie um *package.json* arquivo (que é necessário para o gerenciamento de dependências NGP) e adicione o seguinte -

{

"name": "webrtc-videochat",

"version": "0.1.0",

"description": "webrtc-videochat",

"author": "Author",

"license": "BSD-2-Clause"

}

Em seguida, execute *NPM instalação de inicialização.* Isto irá instalar a biblioteca de inicialização na pasta *videochat / node\_modules.*

Agora precisamos criar uma página HTML básico. Crie um arquivo *index.html* na pasta raiz com o seguinte código -

<html>

<head>

<title>WebRTC Video Demo</title>

<link rel = "stylesheet" href = "node\_modules/bootstrap/dist/css/bootstrap.min.css"/>

</head>

<style>

body {

background: #eee;

padding: 5% 0;

}

video {

background: black;

border: 1px solid gray;

}

.call-page {

position: relative;

display: block;

margin: 0 auto;

width: 500px;

height: 500px;

}

#localVideo {

width: 150px;

height: 150px;

position: absolute;

top: 15px;

right: 15px;

}

#remoteVideo {

width: 500px;

height: 500px;

}

</style>

<body>

<div id = "loginPage" class = "container text-center">

<div class = "row">

<div class = "col-md-4 col-md-offset-4">

<h2>WebRTC Video Demo. Please sign in</h2>

<label for = "usernameInput" class = "sr-only">Login</label>

<input type = "email" id = "usernameInput" c

lass = "form-control formgroup" placeholder = "Login"

required = "" autofocus = "">

<button id = "loginBtn" class = "btn btn-lg btn-primary btnblock">

Sign in</button>

</div>

</div>

</div>

<div id = "callPage" class = "call-page">

<video id = "localVideo" autoplay></video>

<video id = "remoteVideo" autoplay></video>

<div class = "row text-center">

<div class = "col-md-12">

<input id = "callToUsernameInput" type = "text"

placeholder = "username to call" />

<button id = "callBtn" class = "btn-success btn">Call</button>

<button id = "hangUpBtn" class = "btn-danger btn">Hang Up</button>

</div>

</div>

</div>

<script src = "client.js"></script>

</body>

</html>

Esta página deve ser familiar para você. Nós adicionamos o arquivo css *de bootstrap.* Temos também definiu duas páginas. Finalmente, criamos vários campos de texto e botões para obter informações do usuário. Você deve ver os dois elementos de vídeo para transmissões de vídeo locais e remotos. Repare que nós adicionamos um link para um arquivo *Client.js.*

Agora precisamos estabelecer uma conexão com o nosso servidor de sinalização. Criar o arquivo *Client.js* na pasta raiz com o seguinte código -

//our username

var name;

var connectedUser;

//connecting to our signaling server

var conn = new WebSocket('ws://localhost:9090');

conn.onopen = function () {

console.log("Connected to the signaling server");

};

//when we got a message from a signaling server

conn.onmessage = function (msg) {

console.log("Got message", msg.data);

var data = JSON.parse(msg.data);

switch(data.type) {

case "login":

handleLogin(data.success);

break;

//when somebody wants to call us

case "offer":

handleOffer(data.offer, data.name);

break;

case "answer":

handleAnswer(data.answer);

break;

//when a remote peer sends an ice candidate to us

case "candidate":

handleCandidate(data.candidate);

break;

case "leave":

handleLeave();

break;

default:

break;

}

};

conn.onerror = function (err) {

console.log("Got error", err);

};

//alias for sending JSON encoded messages

function send(message) {

//attach the other peer username to our messages

if (connectedUser) {

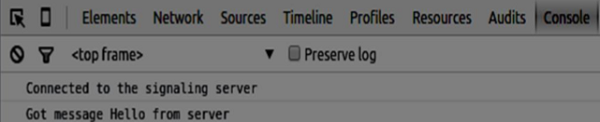
message.name = connectedUser;

}

conn.send(JSON.stringify(message));

};

Agora executar o nosso servidor de sinalização através do *servidor* de *nó.* Então, dentro da pasta raiz executar o comando *estático* e abrir a página de dentro do navegador. Você deverá ver a seguinte saída do console -



O próximo passo é implementar logar um usuário com um nome de usuário único. Nós simplesmente enviar um nome de usuário para o servidor, que então nos dizer se ele é tomado ou não. Adicione o seguinte código ao seu arquivo *Client.js* -

//\*\*\*\*\*\*

//UI selectors block

//\*\*\*\*\*\*

var loginPage = document.querySelector('#loginPage');

var usernameInput = document.querySelector('#usernameInput');

var loginBtn = document.querySelector('#loginBtn');

var callPage = document.querySelector('#callPage');

var callToUsernameInput = document.querySelector('#callToUsernameInput');

var callBtn = document.querySelector('#callBtn');

var hangUpBtn = document.querySelector('#hangUpBtn');

//hide call page

callPage.style.display = "none";

// Login when the user clicks the button

loginBtn.addEventListener("click", function (event) {

name = usernameInput.value;

if (name.length > 0) {

send({

type: "login",

name: name

});

}

});

function handleLogin(success) {

if (success === false) {

alert("Ooops...try a different username");

} else {

//display the call page if login is successful

loginPage.style.display = "none";

callPage.style.display = "block";

//start peer connection

}

};

Em primeiro lugar, vamos selecionar algumas referências aos elementos da página. O que nós ocultar a página chamada. Em seguida, adicione um ouvinte de evento no botão de login. Quando o usuário clica nele, enviamos seu nome de usuário para o servidor. Finalmente, vamos implementar a chamada de retorno handleLogin. Se o login foi bem sucedido, nós mostramos a página chamada e começar a configurar uma conexão entre pares.

Para iniciar uma conexão ponto que precisamos -

* + Obter uma corrente a partir da câmara web.
  + Criar o objeto RTCPeerConnection.

Adicione o seguinte código para os "seletores UI bloquear" -

var localVideo = document.querySelector('#localVideo');

var remoteVideo = document.querySelector('#remoteVideo');

var yourConn;

var stream;

Modificar a função *handleLogin* -

function handleLogin(success) {

if (success === false) {

alert("Ooops...try a different username");

} else {

loginPage.style.display = "none";

callPage.style.display = "block";

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Starting a peer connection

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//getting local video stream

navigator.webkitGetUserMedia({ video: true, audio: true }, function (myStream) {

stream = myStream;

//displaying local video stream on the page

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

//using Google public stun server

var configuration = {

"iceServers": [{ "url": "stun:stun2.1.google.com:19302" }]

};

yourConn = new webkitRTCPeerConnection(configuration);

// setup stream listening

yourConn.addStream(stream);

//when a remote user adds stream to the peer connection, we display it

yourConn.onaddstream = function (e) {

remoteVideo.src = window.URL.createObjectURL(e.stream);

};

// Setup ice handling

yourConn.onicecandidate = function (event) {

if (event.candidate) {

send({

type: "candidate",

candidate: event.candidate

});

}

};

}, function (error) {

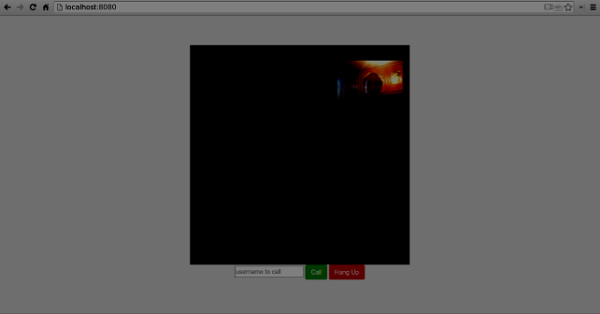
console.log(error);

});

}

};

Agora, se você executar o código, a página deve permitir que você faça o login e exibir seu fluxo de vídeo local na página.



Agora estamos prontos para iniciar uma chamada. Em primeiro lugar, enviamos uma *oferta* de outro usuário. Uma vez que um usuário recebe a oferta, ele cria uma *resposta* e iniciar a negociação candidatos ICE. Adicione o seguinte código para o arquivo *Client.js* -

//initiating a call

callBtn.addEventListener("click", function () {

var callToUsername = callToUsernameInput.value;

if (callToUsername.length > 0) {

connectedUser = callToUsername;

// create an offer

yourConn.createOffer(function (offer) {

send({

type: "offer",

offer: offer

});

yourConn.setLocalDescription(offer);

}, function (error) {

alert("Error when creating an offer");

});

}

});

//when somebody sends us an offer

function handleOffer(offer, name) {

connectedUser = name;

yourConn.setRemoteDescription(new RTCSessionDescription(offer));

//create an answer to an offer

yourConn.createAnswer(function (answer) {

yourConn.setLocalDescription(answer);

send({

type: "answer",

answer: answer

});

}, function (error) {

alert("Error when creating an answer");

});

};

//when we got an answer from a remote user

function handleAnswer(answer) {

yourConn.setRemoteDescription(new RTCSessionDescription(answer));

};

//when we got an ice candidate from a remote user

function handleCandidate(candidate) {

yourConn.addIceCandidate(new RTCIceCandidate(candidate));

};

Nós adicionar um manipulador de *clique* com o botão de Chamada, que inicia uma oferta. Então vamos implementar vários manipuladores esperados pelo manipulador *onmessage.* Eles serão processados ​​de forma assíncrona até que ambos os usuários fizeram uma conexão.

O último passo é implementar o recurso de pendurar-up. Isto irá parar a transmissão de dados e dizer ao outro usuário para fechar a chamada. Adicione o seguinte código -

//hang up

hangUpBtn.addEventListener("click", function () {

send({

type: "leave"

});

handleLeave();

});

function handleLeave() {

connectedUser = null;

remoteVideo.src = null;

yourConn.close();

yourConn.onicecandidate = null;

yourConn.onaddstream = null;

};

Quando o usuário clica no botão Hang Up -

* + Ele vai enviar uma mensagem de "licença" para o outro usuário
  + Ele vai fechar a RTCPeerConnection e destruir a ligação localmente

Agora, execute o código. Você deve ser capaz de fazer logon no servidor usando duas abas do navegador. Você pode então chamar a guia e desligar a chamada.



A seguir, todo o arquivo *Client.js* -

//our username

var name;

var connectedUser;

//connecting to our signaling server

var conn = new WebSocket('ws://localhost:9090');

conn.onopen = function () {

console.log("Connected to the signaling server");

};

//when we got a message from a signaling server

conn.onmessage = function (msg) {

console.log("Got message", msg.data);

var data = JSON.parse(msg.data);

switch(data.type) {

case "login":

handleLogin(data.success);

break;

//when somebody wants to call us

case "offer":

handleOffer(data.offer, data.name);

break;

case "answer":

handleAnswer(data.answer);

break;

//when a remote peer sends an ice candidate to us

case "candidate":

handleCandidate(data.candidate);

break;

case "leave":

handleLeave();

break;

default:

break;

}

};

conn.onerror = function (err) {

console.log("Got error", err);

};

//alias for sending JSON encoded messages

function send(message) {

//attach the other peer username to our messages

if (connectedUser) {

message.name = connectedUser;

}

conn.send(JSON.stringify(message));

};

//\*\*\*\*\*\*

//UI selectors block

//\*\*\*\*\*\*

var loginPage = document.querySelector('#loginPage');

var usernameInput = document.querySelector('#usernameInput');

var loginBtn = document.querySelector('#loginBtn');

var callPage = document.querySelector('#callPage');

var callToUsernameInput = document.querySelector('#callToUsernameInput');

var callBtn = document.querySelector('#callBtn');

var hangUpBtn = document.querySelector('#hangUpBtn');

var localVideo = document.querySelector('#localVideo');

var remoteVideo = document.querySelector('#remoteVideo');

var yourConn;

var stream;

callPage.style.display = "none";

// Login when the user clicks the button

loginBtn.addEventListener("click", function (event) {

name = usernameInput.value;

if (name.length > 0) {

send({

type: "login",

name: name

});

}

});

function handleLogin(success) {

if (success === false) {

alert("Ooops...try a different username");

} else {

loginPage.style.display = "none";

callPage.style.display = "block";

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Starting a peer connection

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//getting local video stream

navigator.webkitGetUserMedia({ video: true, audio: true }, function (myStream) {

stream = myStream;

//displaying local video stream on the page

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

//using Google public stun server

var configuration = {

"iceServers": [{ "url": "stun:stun2.1.google.com:19302" }]

};

yourConn = new webkitRTCPeerConnection(configuration);

// setup stream listening

yourConn.addStream(stream);

//when a remote user adds stream to the peer connection, we display it

yourConn.onaddstream = function (e) {

remoteVideo.src = window.URL.createObjectURL(e.stream);

};

// Setup ice handling

yourConn.onicecandidate = function (event) {

if (event.candidate) {

send({

type: "candidate",

candidate: event.candidate

});

}

};

}, function (error) {

console.log(error);

});

}

};

//initiating a call

callBtn.addEventListener("click", function () {

var callToUsername = callToUsernameInput.value;

if (callToUsername.length > 0) {

connectedUser = callToUsername;

// create an offer

yourConn.createOffer(function (offer) {

send({

type: "offer",

offer: offer

});

yourConn.setLocalDescription(offer);

}, function (error) {

alert("Error when creating an offer");

});

}

});

//when somebody sends us an offer

function handleOffer(offer, name) {

connectedUser = name;

yourConn.setRemoteDescription(new RTCSessionDescription(offer));

//create an answer to an offer

yourConn.createAnswer(function (answer) {

yourConn.setLocalDescription(answer);

send({

type: "answer",

answer: answer

});

}, function (error) {

alert("Error when creating an answer");

});

};

//when we got an answer from a remote user

function handleAnswer(answer) {

yourConn.setRemoteDescription(new RTCSessionDescription(answer));

};

//when we got an ice candidate from a remote user

function handleCandidate(candidate) {

yourConn.addIceCandidate(new RTCIceCandidate(candidate));

};

//hang up

hangUpBtn.addEventListener("click", function () {

send({

type: "leave"

});

handleLeave();

});

function handleLeave() {

connectedUser = null;

remoteVideo.src = null;

yourConn.close();

yourConn.onicecandidate = null;

yourConn.onaddstream = null;

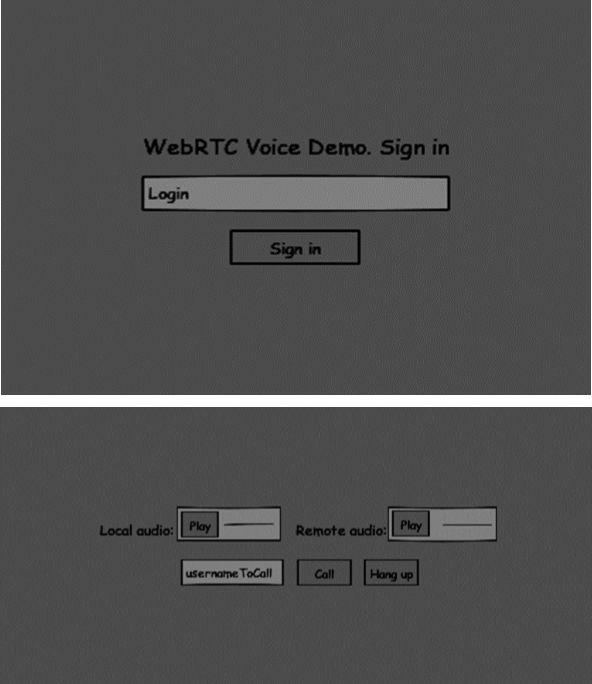
};

Resumo

Esta demo oferece uma linha de base de recursos que cada aplicação WebRTC necessita. Para melhorar esta demonstração você pode adicionar a identificação do utilizador através de plataformas como o Facebook ou Google, lidar com a entrada do usuário para dados inválidos. Also, the WebRTC connection can fail because of several reasons like not supporting the technology or not being able to traverse firewalls. A worth of work has gone into making any WebRTC application stable.

WebRTC - Voice Demo

In this chapter, we are going to build a client application that allows two users on separate devices to communicate using WebRTC audio streams. Our application will have two pages. One for login and the other for making an audio call to another user.



The two pages will be the Resumo

In this chapter, we added user authentication to our signaling server. We also learned how to create self-signed SSL certificates and use them in the scope of WebRTC applications.