**PARTIE III : IMPLEMENTATION**

Une des forces du WebRTC est le fait que les données s’échangent directement entre les 2 ordinateurs et que ces dernières ne passent pas par un serveur externe ! Pour réussir cet exploit, la technologie WebRTC repose sur 3 APIS web :

***getUserMedia*** : cette API permet de récupérer les flux vidéo et audio d’un ordinateur

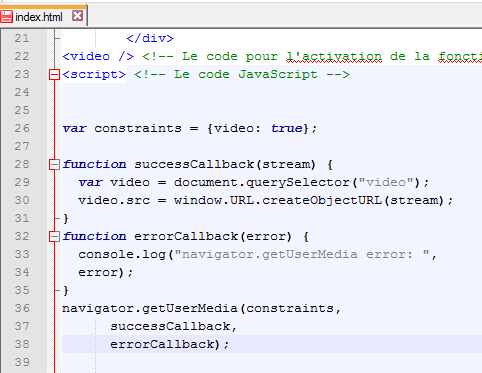
***RTCPeerConnection*** : cette API permet de faire communiquer des données entre 2 hôtes en tenant compte de tout un ensemble de contraintes telles que l’adresse IP d’une machine, ses codecs, sa connectivité, …

***RTCDataChannel*** : cette API permet de faire transiter sur une RTCPeerConnection des données textuelles ou binaires.

Pour notre projet, nous n’allons utiliser que les API getUserMedia et RTCPeerConnection.

* **getUserMedia**

Il s’agit d’une API qui permet de récupérer un ensemble de Stream de médias synchronisés. Chaque Stream peut être vidéo / audio.



Dans l’exemple ci-dessus, nous ne récupérons que la vidéo et nous injectons le résultat de l’appel de getUserMedia dans une balise vidéo.

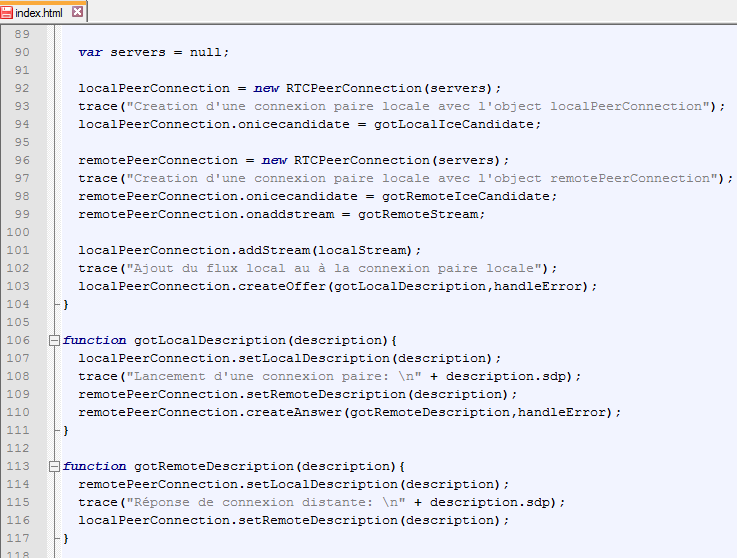
* **RTCPeerConnection**

La RTCPeerConnection permet de gérer le transport des données. Pour initialiser une RTCPeerConnection, on répond au principe de l’offre et de la demande. Il y a d’une part, une notion d’offre et de demande pour communiquer mais aussi une notion de chemin à emprunter ! Ces 2 notions s’appellent le “**Signaling**”.

Le **Signaling** a pour objectif de répondre à ces questions :

* Quel type de média et format je supporte ?
* Que puis-je envoyer ?
* Quel est mon type d’infrastructure réseau ?

Pour faire cette étape, il suffit juste de trouver un moyen de passer ces informations à l’hôte distant. Une des technologies préconisées pour faire le Signaling est “les WebSockets”. C’est donc ici qu’interviendra notre serveur de websockets.



1. **GESTION DE L’OFFRE**

Voici comment se déroule le signaling :

Aurel appelle la méthode **createOffer()**

Dans le callback, Aurel appelle **setLocalDesctiption()**

Aurel **sérialise l’offre** et l’envoie à Sylvie

Sylvie appelle la méthode **setRemoteDescription()** avec l’offre

Sylvie appelle la méthode **createAnswer()**

Sylvie appelle la méthode **setLocalDescription()** avec la réponse envoyée à Aurel

Aurel reçoit la réponse et appelle **setRemoteDescription()**

1. **Gestion du chemin ICE Candidate (ICE = Interactive Connectivity Establishement)**

Aurel & Sylvie ont leur **RTCPeerConnection**

En cas de succès de chaque côté les **IceCanditates** sont envoyées

Aurel **sérialise** ses IceCandidates et les envoie à Sylvie

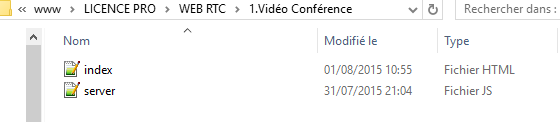
Sylvie reçoit les IceCandidates d’Aurel et appelle **addIceCandidate()**

Sylvie **sérialise** ses IceCandidates et les envoie à Aurel

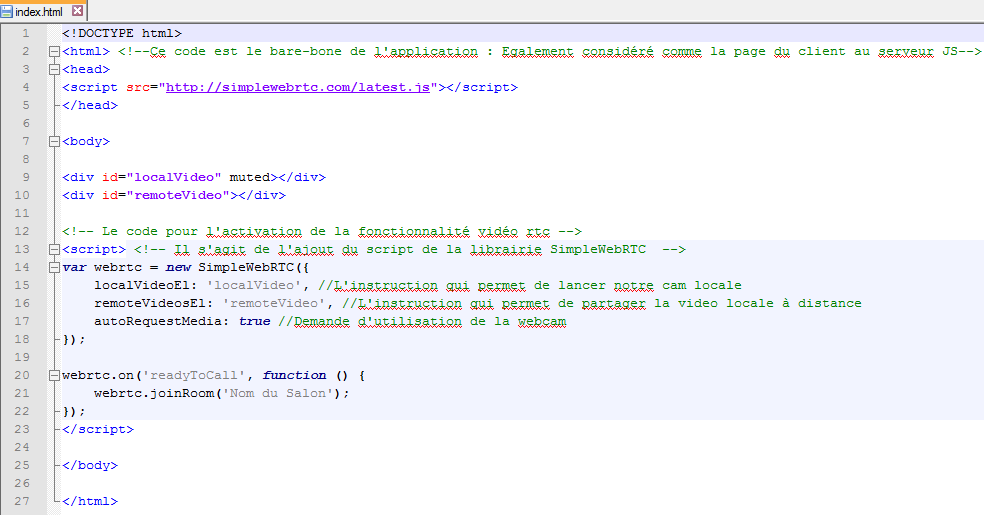
Aurel reçoit les IceCandidates de Sylvie et appelle **addIceCandidate()**

Les 2 savent comment communiquer.

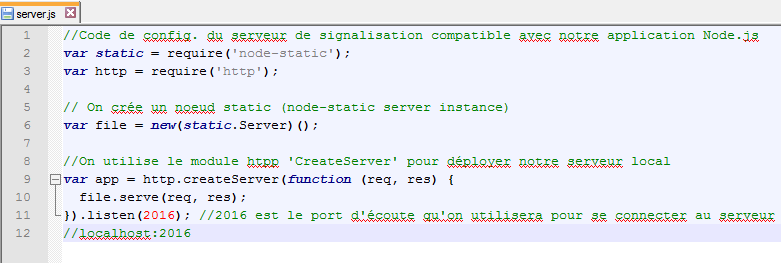
1. **LES CODES SOURCES**
   1. **Le WebRTC : Vidéoconférence**



Le code **index.html** représente le ***client*** (l’application web)

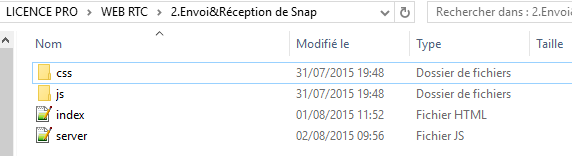


Le code **server.js** représente le ***serveur*** ***local*** (l’application Node JS)

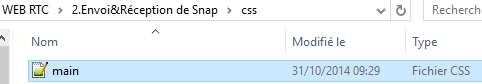


L'application Node server joue deux rôles :

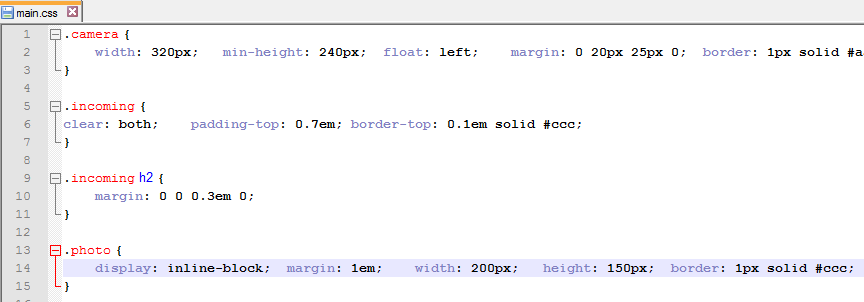
* Fonctionne comme un intermédiaire de messagerie
* Gérer les salons (rooms) de vidéoconférence
  1. **Le WebRTC : Envoi/Réception de captures d’images vidéo (Snap)**



Le dossier ***css*** contient un fichier ***main.css*** :



Le code **main.css**

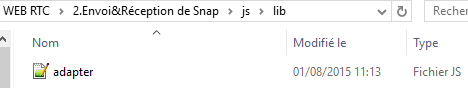


Ce code nous a permis de :

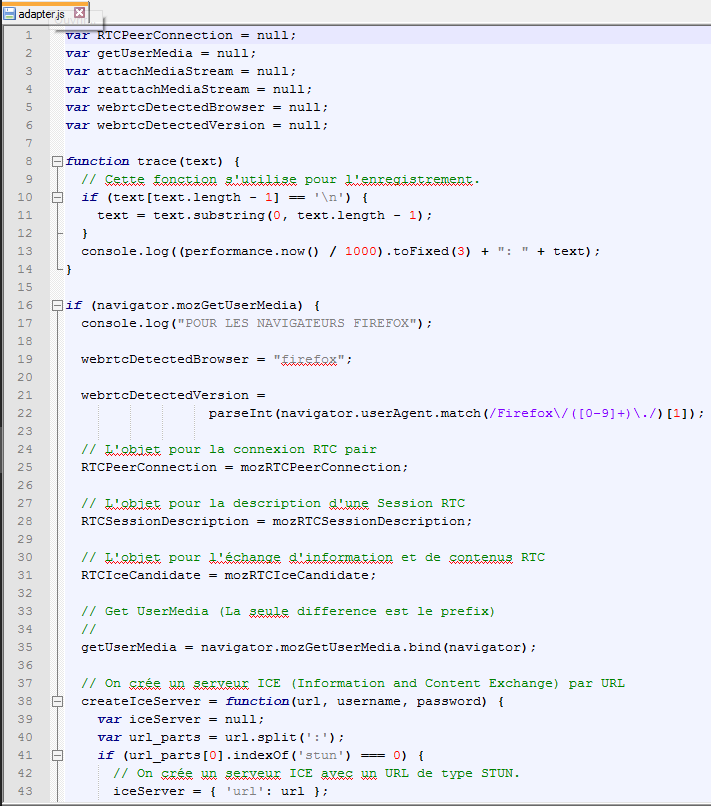
* Afficher la vidéo en grande résolution (largeur & hauteur)
* Ajouter du design à la page cliente
* Améliorer la mise en page, la structure, et ajouter un espace réservé à la zone de réception/stockage de captures d’images de vidéo.

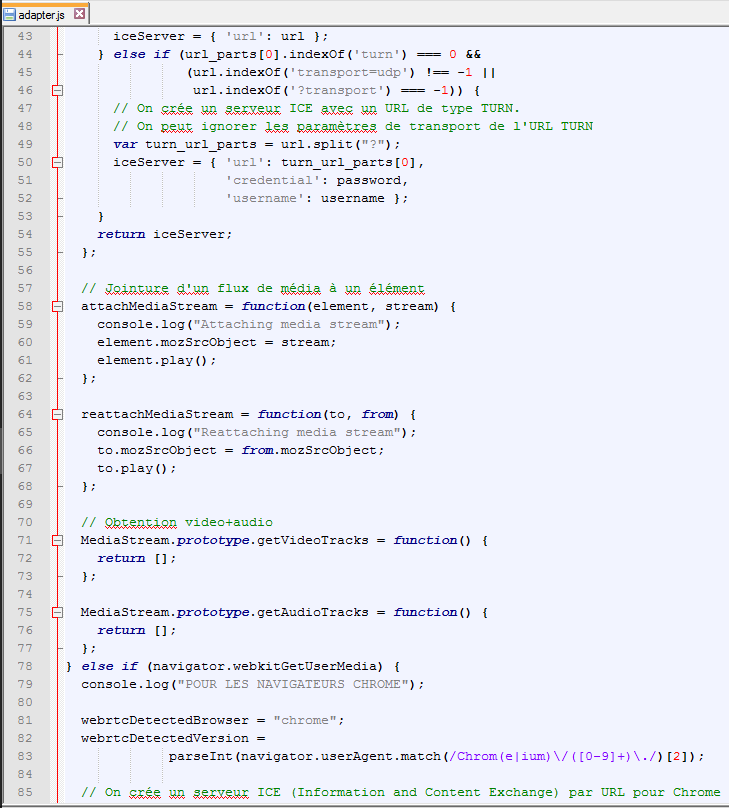
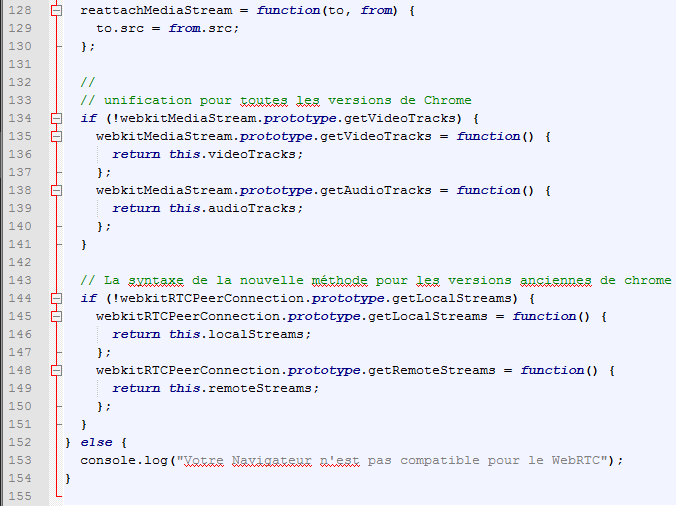
Le dossier **js** contient :

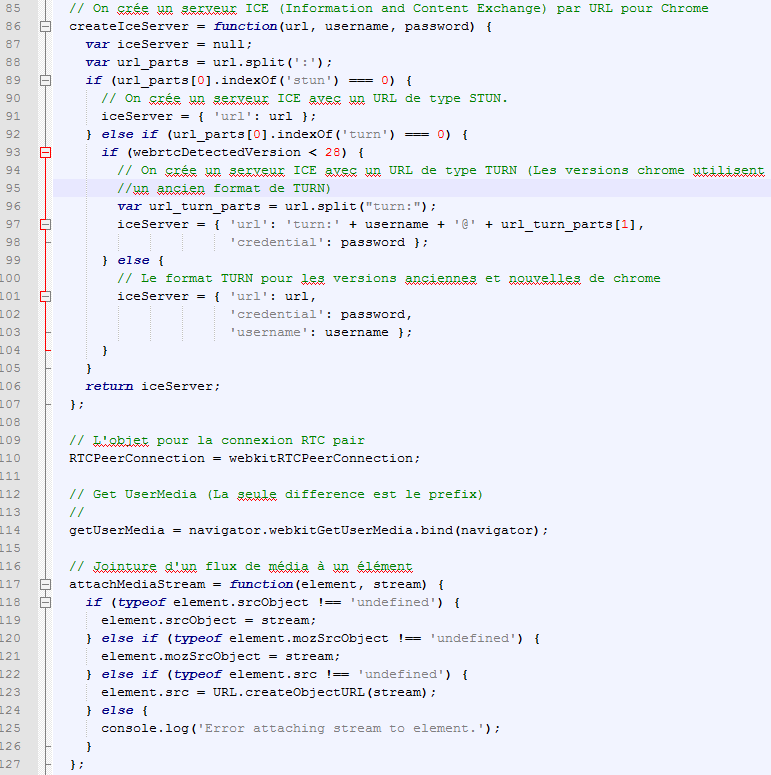
-Un dossier **lib**

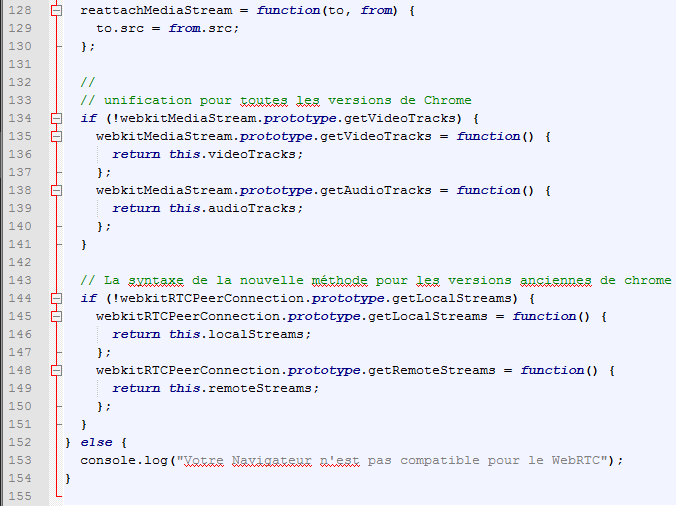


**adapter.js :** est le fichier utile dont le code source permet de masquer les différences entre Mozilla Firefox et les implémentations de Google Chrome du WebRTC.

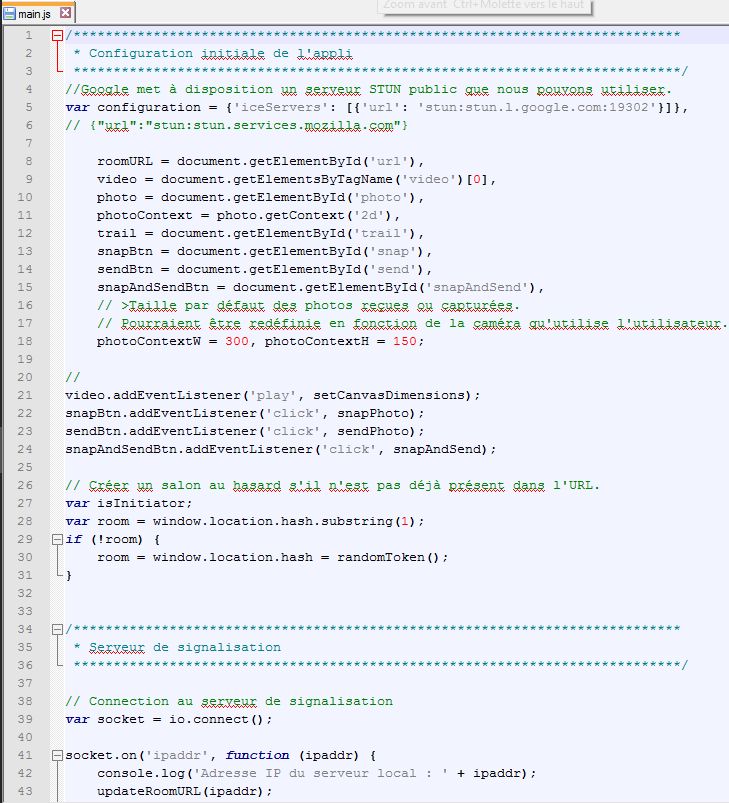


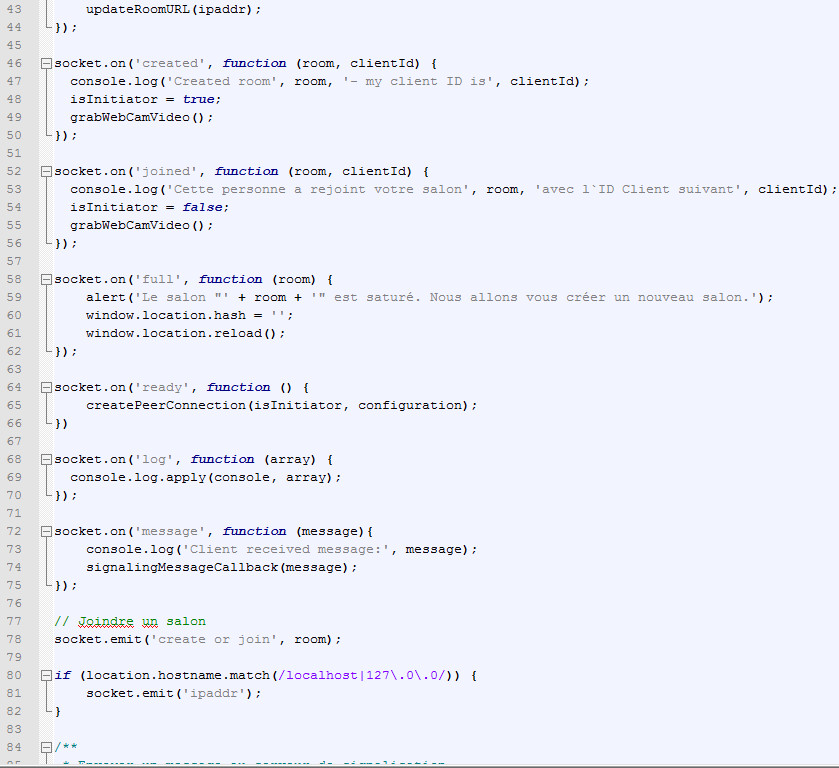
 

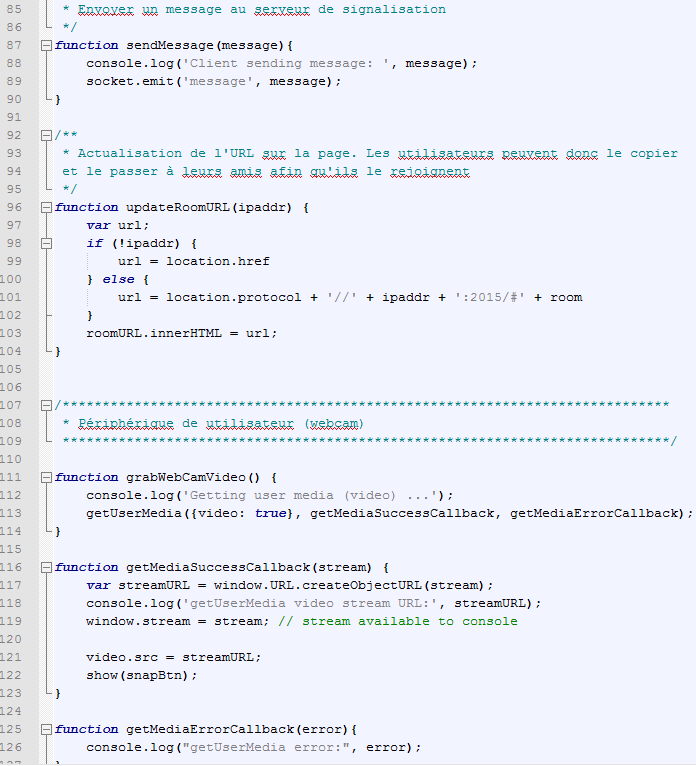




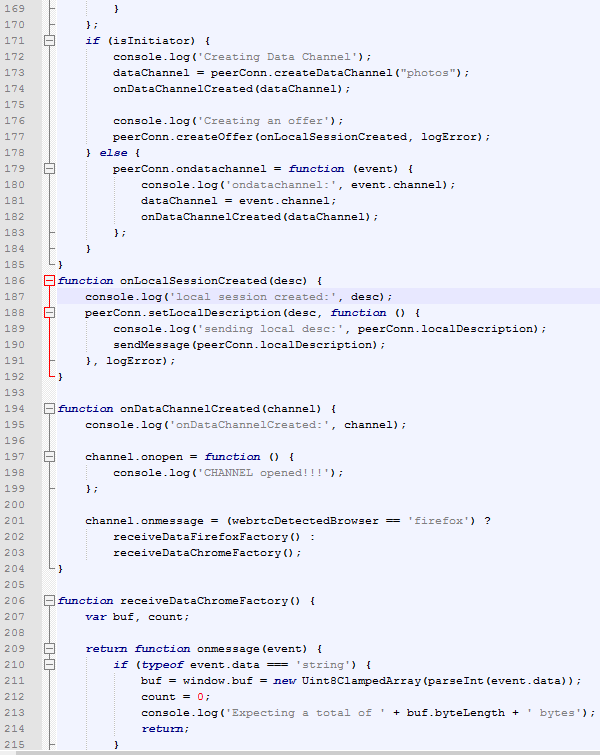
Le code **main.js** :

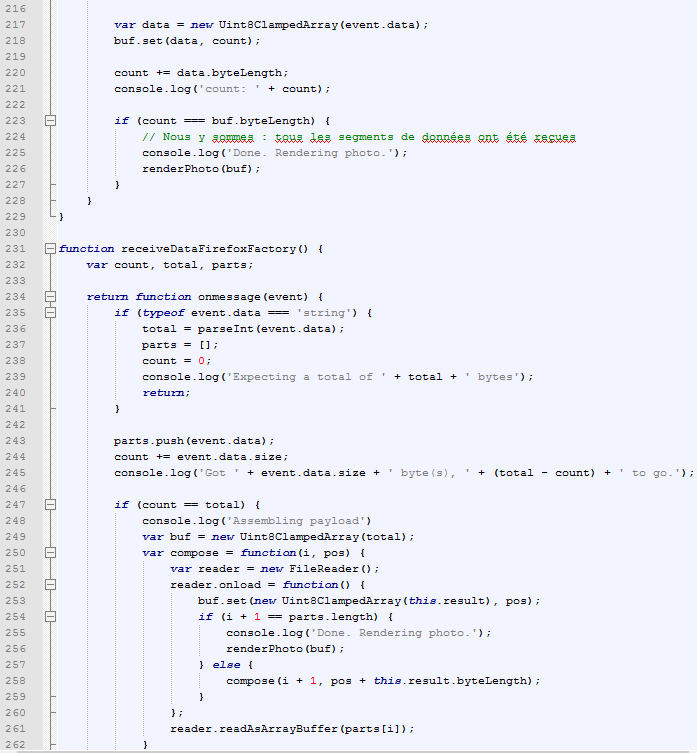


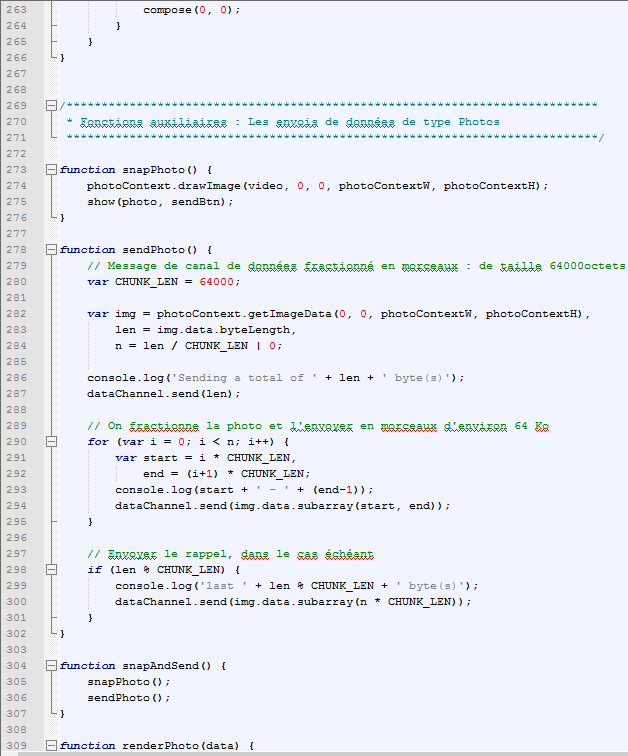


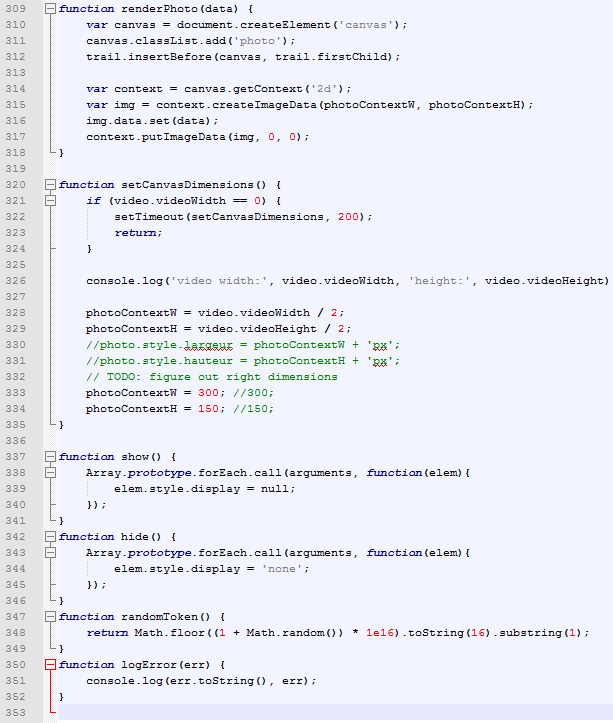




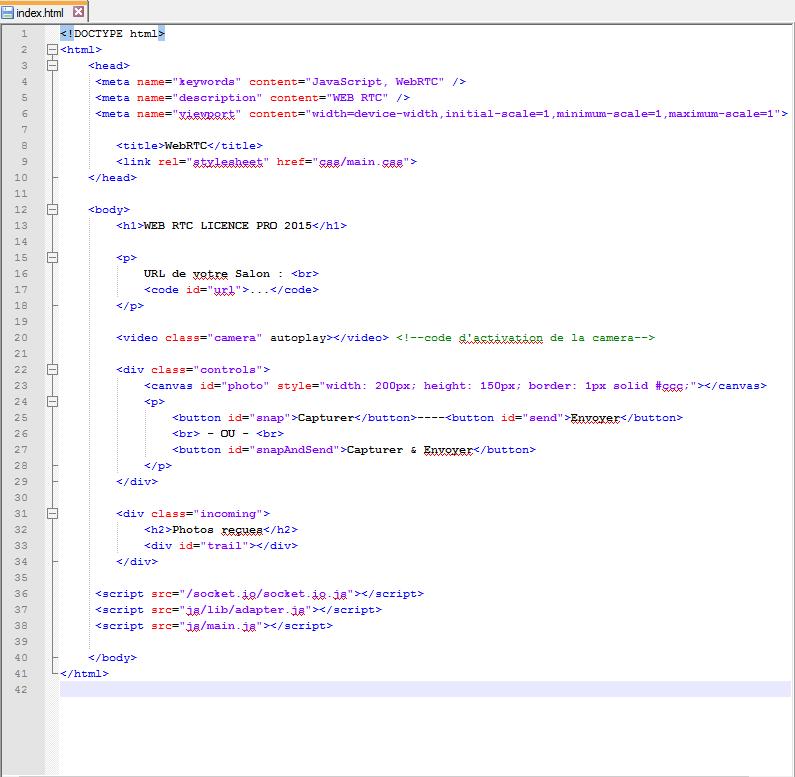




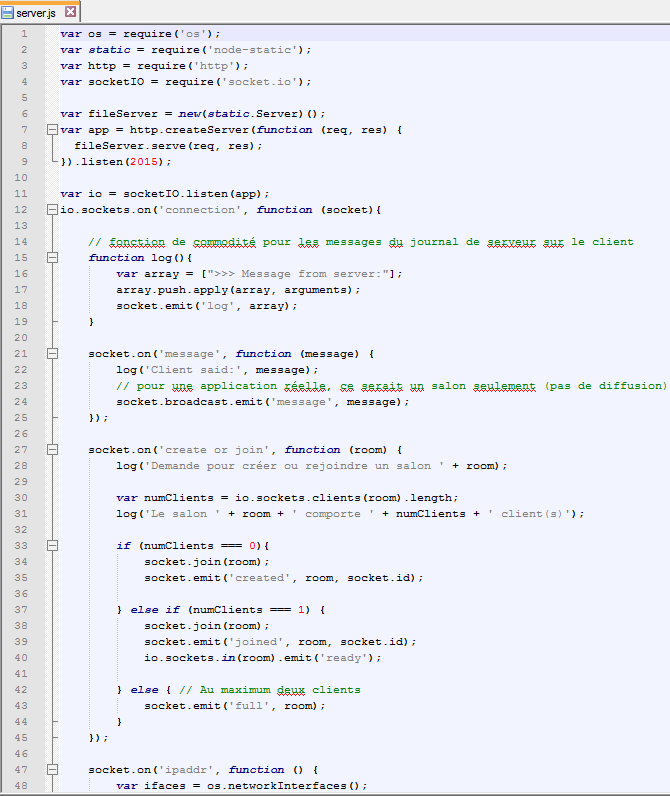


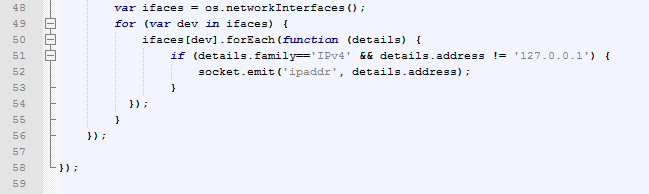


Le code **index.html** représente le ***client*** (l’application web)



Le code **server.js** représente le ***serveur*** ***local*** (l’application Node JS)

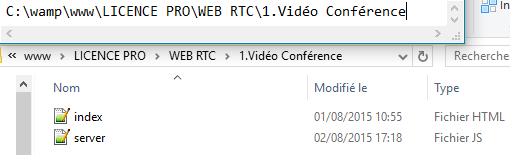




1. **TEST DE L’APPLICATION**

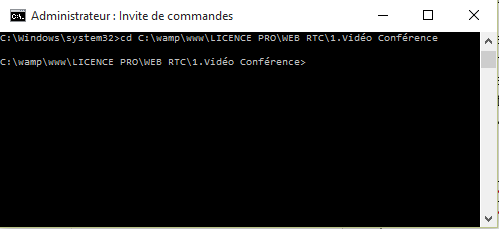
1-Nous devons nous équiper les logiciels « Node.js » et « Node-static » et les installer.

2-Ensuite, se positionner dans le dossier de l’application et copier le chemin du dossier :



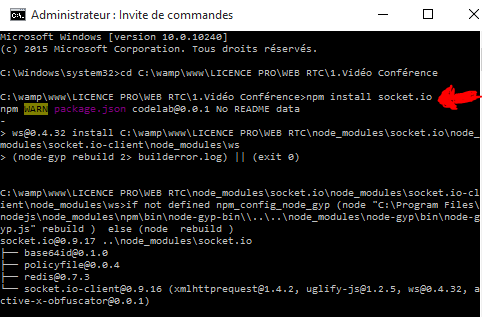
3-Lancer l’invite de commande (**cmd**) et taper la commande :

cd le\_chemin\_du\_dossier\_de\_l’application

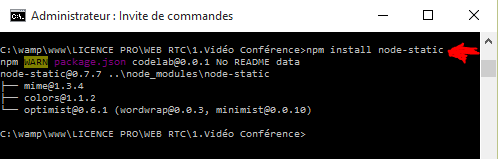


4-Puis installer **socket.io** et **node-static** en utilisant la commande **npm** comme suit :

* npm install **socket.io**

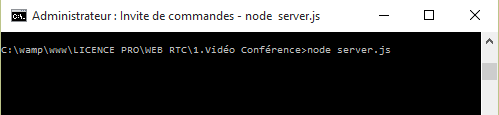


* npm install **node-static**

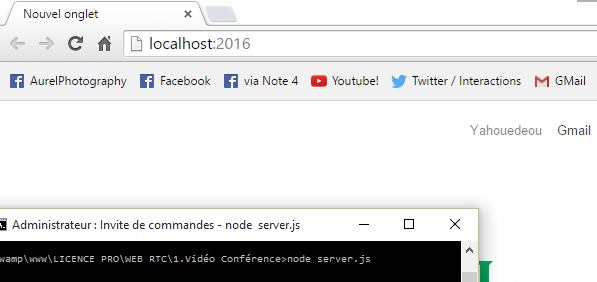


5-Mettre en marche le serveur Node JS en tapant dans le terminal la commande suivante :

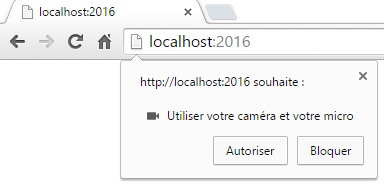
**node server.js**



6-Dans un navigateur, taper l’adresse de connexion locale ‘’localhost’’ avec notre port d’écoute ‘’ :2016’’ préenregistré dans le code ***server.js***



7-Une notification venant du navigateur nous demande si nous voulons activer la fonctionnalité vidéo chat : Cliquer sur « Autoriser »



8-Capture d’écran de deux clients communiquant