<http://forasoft.github.io/webrtc-in-plain-russian/>

<http://forasoft.github.io/kurento-media-server-overview/>

<https://openvidu.io/docs/home/>

<https://developer.mozilla.org/ru/docs/Web/API/WebRTC_API/Session_lifetime>

WebRTC плагин для поддержки в сафари и ИЕ <https://temasys.atlassian.net/wiki/spaces/TWPP/overview>

реализация в Edge <https://caniuse.com/#feat=objectrtc>

<https://en.wikipedia.org/wiki/Interactive_Connectivity_Establishment>

работает при подключении разных браузеров. но как на счет IE, Edge?

**WebRTC** (Web Real-Time Communications) - это технология, которая позволяет Web-приложениям и сайтам захватывать и вещать аудио и/или видео медиа-потоки опционально, а также передавать между браузерами произвольные данные, без обязательного использования посредников. Набор стандартов, которые включает в себя технология WebRTC, позволяет обмениваться данными и проводить телеконференции в режиме узел-узел (**peer-to-peer**) без необходимости пользователю устанавливать плагины или любое другое стороннее программное обеспечение.

Технология WebRTC делает возможной видеосвязь через окно браузера, так что для присоединения к звонку необходимо только перейти по ссылке на соответствующую веб-страницу.

### В WebRTC используются два аудиокодека, G.711 и Opus, а также видеокодеки VP8 и H.264.

### Преимущества технологии

### • проведение конференции в браузере значительно упрощает процесс проведения конференции — пользователю не нужно устанавливать для этого отдельные приложения;

### • используемые кодеки обеспечивают хорошее качество связи;

### • возможность реализации любых элементов интерфейса средствами HTML5 и JavaScript;

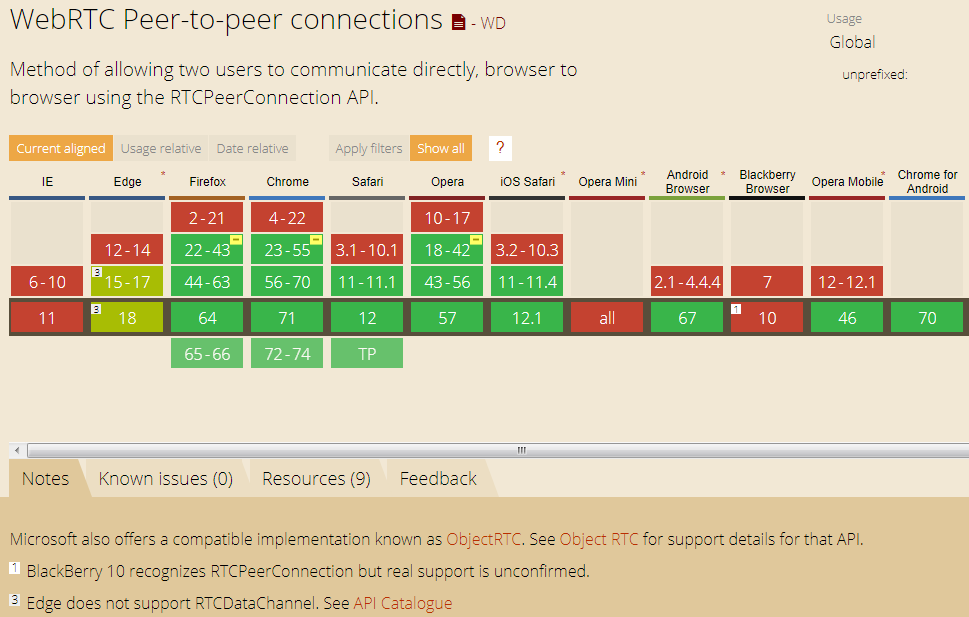
### • открытый исходный код даёт больше возможностей для использования.

### Недостатки технологии

• технология определяет только общий стандарт передачи данных (видео и звука), но отдельные решения разных браузеров относительно адресации абонентов и прочих управляющих процессов не совместимы между собой. Поэтому даже звонки между парой различных браузеров представляют отдельную сложность.

• обеспечение групповых конференций требует дополнительных разработок поверх WebRTC.

Групповой чат, проведение видео-конференций,

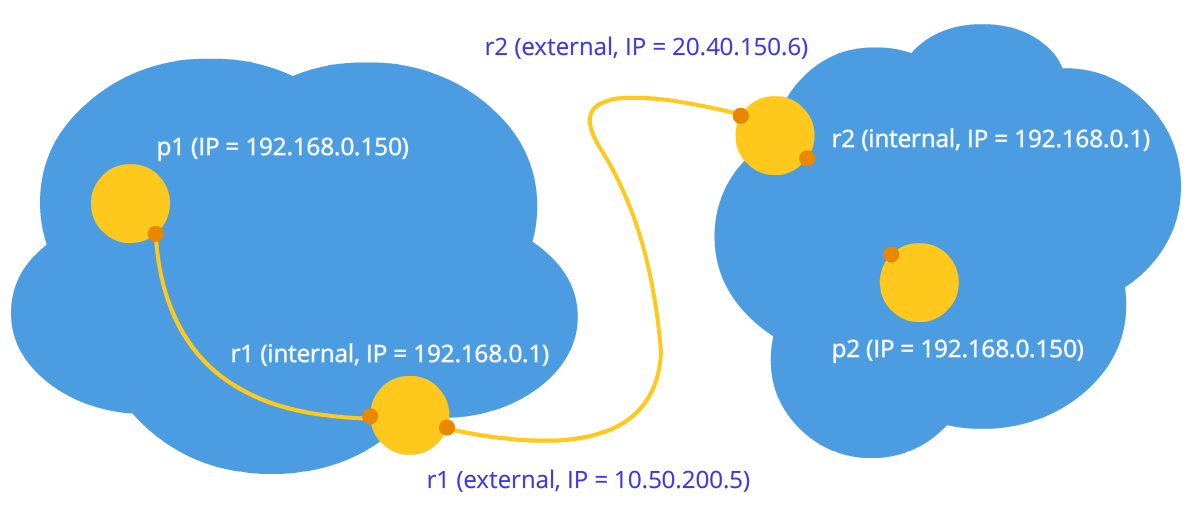


https://caniuse.com/#search=webrtc

Установить соединение **peer-to-peer** (**p2p** )– довольно трудная задача, так как компьютеры не всегда обладают публичными **IP** адресами, то есть адресами в интернете. Из-за небольшого количества **IPv4** адресов (и для целей безопасности) был разработан механизм **NAT**, который позволяет создавать приватные сети, например, для домашнего использования.

Network Address Translation ([NAT](https://developer.mozilla.org/ru/docs/Glossary/NAT)) - это стандарт, который поддерживает разделение адреса путем маршрутизации входящих и исходящих пакетов данных в и из локальной сети (LAN), которые разделяют единственный WAN (глобальный) адрес.

Многие домашние роутеры сейчас поддерживают **NAT** и благодаря этому все домашние устройства имеют выход в интернет, хотя провайдеры интернета обычно предоставляют один **IP** адрес. Публичные **IP** адреса - уникальны в интернете, а приватные нет. Поэтому соединиться **p2p** - трудно.



На рисунках выше первая буква в дву-символьных обозначениях означает тип узла (p = **peer**, r = **router**). На втором рисунке имеем две разные сети, у которых похожие нумерации узлов. Здесь появляются маршрутизаторы (роутеры), у которых есть два сетевых интерфейса – внутри своей сети и вне своей сети. Поэтому у них два **IP** адреса. Обычные узлы имеют только один интерфейс, через который они могут общаться только в своей сети. Если они передают данные кому-то вне своей сети, то только с помощью **NAT** внутри маршрутизатора (роутера) и поэтому видимы для других под **IP** адресом роутера – это их внешний **IP** адрес. Таким образом, у узла **p1** есть внутренний **IP** = 192.168.0.200 и внешний **IP** = 10.50.200.5, причем последний адрес будет внешним также и для всех других узлов в его сети[1](http://forasoft.github.io/webrtc-in-plain-russian/#fn:same_router). Похожая ситуация и для узла **p2**. Поэтому их связь невозможна, если использовать только их внутренние (свои) **IP** адреса. Можно воспользоваться внешними адресами, то есть адресами роутеров, но, так как у всех узлов в одной приватной сети один и тот же внешний адрес, то это довольно затруднительно. Это проблема решается с помощью механизма **NAT.**

Что же будет, если мы все-таки решим соединить узлы через их внутренние адреса? Данные не выйдут за пределы сети. Для усиления эффекта можно представить ситуацию, изображенную на последнем рисунке – у обоих узлов совпадают внутренние адреса. Если они будут использовать их для коммуникации, то каждый узел будет общаться с самим собой.

**WebRTC** успешно справляется с такими проблемами, используя протокол **ICE**, который, правда, требует использования дополнительных серверов (**STUN**, **TURN**).

Чтобы соединить два узла через протокол **WebRTC** необходимо провести установку соединения двух узлов (сигнальная фаза). Сигнализация - это процесс передачи управляющей информации между двумя устройствами для опредения протоколов связи, каналов, кодирования и формата медиа-данных,  методов передачи данных, а также информации, необходимой для маршрутизации. Наиболее важная вещь, о которой нужно знать о процессе сигнализации для WebRTC - это то, **этот процесс не определен в спецификации**.

Для обмена сигнальной информацией, вы можете выбрать отправку JSON-объектов через WebSocket-соединение,  можете использовать протокол XMPP/SIP через соответствующий канал, так же можете использовать [XMLHttpRequest](https://developer.mozilla.org/ru/docs/Web/API/XMLHttpRequest) через [HTTPS](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/HTTPS)  с техникой пуллинга ([HTTPS](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/HTTPS) with polling), или же другие комбинации технологий, которые вам могут прийти в голову. Вы даже можете использовать электронную почту в качестве сигнального канала.

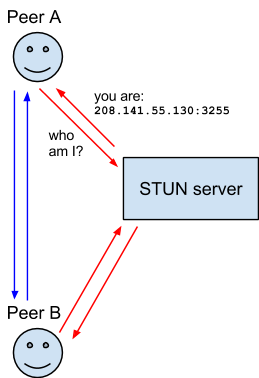
Все данные во время сигнальной фазы передаются в виде текста и делятся на два типа –**SDP (дескриптор сессии)** и **Ice Candidate**. Первый тип используется для установления логического соединения, а второй для физического. Если дескриптор сессии содержал информацию о наших установках на уровне камеры и микрофона, то кандидаты содержат информацию о нашем расположении в сети.

**Interactive Connectivity Establishment** (**ICE**) - это технология, которая используется в компьютерных сетях, чтобы найти пути для двух устройств (например, компьютеров) для подключения/передачи данных непосредственно друг другу в **p2p подключении. ICE чаще всего используется в интерактивных медиа, например: IP-телефония, p2p коммуникация, видео и система мгновенного обмена сообщениями.**

[Session Description Protocol (SDP)](http://en.wikipedia.org/wiki/Session_Description_Protocol)  - это стандарт для описания мультимедийного контента соединения,  как например: разрешение, форматы, кодеки, шифрование и т.д. Так, чтобы оба узла могли понять друг друга, после того как данные начнут передаваться. Это, в сущности, метаданные, описывающие содержимое, а не медиа контент сам по себе.

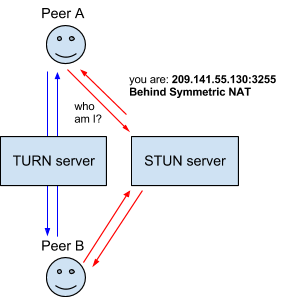
[Session Traversal Utilities for NAT (STUN)](http://en.wikipedia.org/wiki/STUN) (акроним в акрониме) это протокол для нахождения и определения вашего публичного адреса и любых ограничений в вашем маршрутизаторе, которые препятствуют прямому соединению с узлом.

Клиент отправит запрос к STUN серверу в интернете, который ответит публичным адресом клиента и, доступен ли, или нет, клиент за NAT маршрутизатором.



Некоторые маршрутизаторы, использующие NAT применяют ограничение, называемое "Симметричный NAT" (Symmetric NAT). Это означает, что маршрутизатор будет принимать соединения только от узлов, к которым вы ранее подключились.

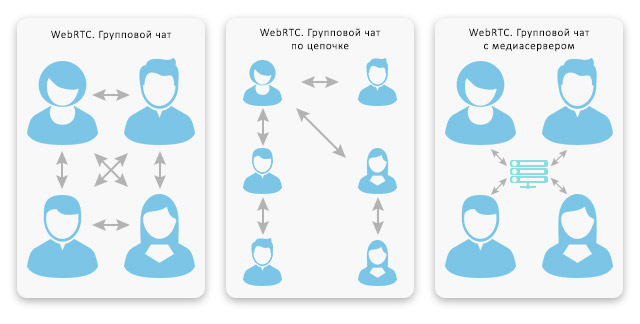
[Traversal Using Relays around NAT (TURN)](http://en.wikipedia.org/wiki/TURN) предназначен для обхода ограничения "Симметричный NAT" путём открытия соединения с TURN сервером и ретрансляции всей информации через этот сервер. Вы создадите соединение с TURN сервером и сообщите всем узлам слать пакеты этому серверу, которые затем будут переправлены вам. Очевидно, что они приходят с некоторыми накладными расходами, поэтому это используется, только если нет других альтернатив.



В каких случаях необходим **TURN** сервер? Почему не хватает **STUN** сервера? Дело в том, что существует несколько разновидностей **NAT**. Они одинаково подменяют **IP** адрес и порт, однако в некоторые из них встроена дополнительная защита от “фальсификации”. Например, в *симметричной* таблице **NAT** сохраняются еще 2 параметра - **IP** и порт удаленного узла. Пакет из внешней сети проходит через **NAT** во внутреннюю сеть только в том случае, если адрес и порт источника совпадают с записанными в таблице. Поэтому фокус со **STUN** сервером не удается - таблица **NAT** хранит адрес и порт **STUN** сервера и, когда роутер получает пакет от **WebRTC** собеседника, он его отбрасывает, так как он “фальсифицирован”. Он пришел не от **STUN** сервера.

Таким образом **TURN** сервер нужен в том случае, когда оба собеседника находятся за *симметричным* **NAT** (каждый за своим).





Архитектура групповых видеочатов. Решение с использованием медиасервера в таких случаях самое надежное, но значительно увеличивает накладные расходы на сервер.

Если мы делаем групповой чат без медиасервера, то вынуждены либо соединить каждого клиента с каждым, либо выбрать одного клиента и передавать поток через него.

Когда видео транслирует один клиент, например в приложение для проведения вебинаров, можно передавать потоки по [цепочке между всеми клиентами](http://forasoft.github.io/chain-webrtc).

Но у этих решений есть недостатки, которые станут критичными при большом числе подключенных клиентов.

* Лишняя нагрузка на ресурсы и трафик клиента из-за дублирования исходящего потока.
* Низкое качество видео при большом количестве пользователей.
* Реализация записи на клиенте — более трудоемкая задача. Если мы хотим сохранить файл на сервере, пользователю придется ждать окончания его загрузки.

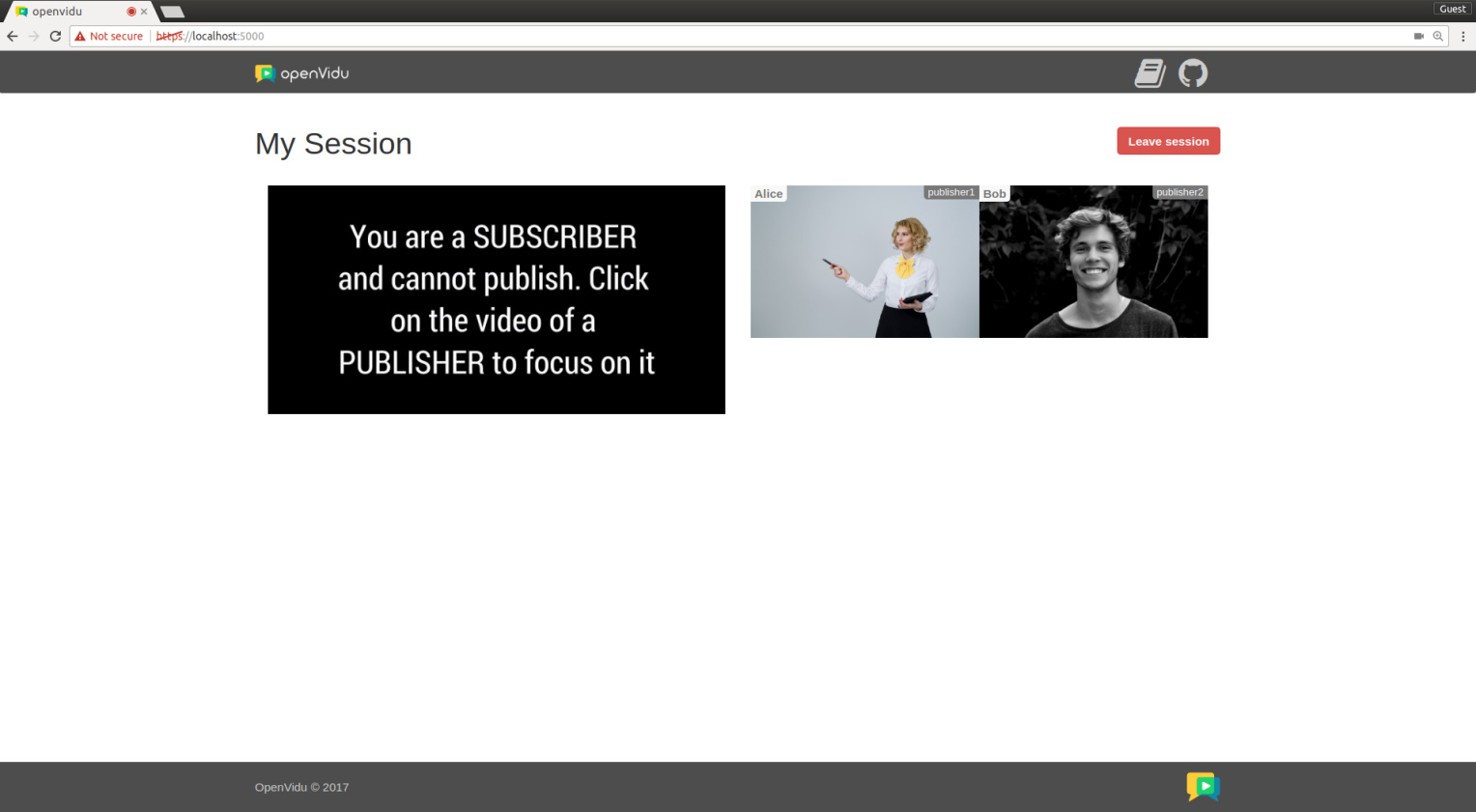
Медиасервер решает эти проблемы. Он управляет, маршрутизирует и записывает медиапотоки. С ним наше приложение будет работать быстрее у пользователей.

OpenVidu is divided into two parts:

* **OpenVidu Browser**: it is a library to use in your client side (available for JavaScript and TypeScript). It allows you to create video-calls, join users to them, send and receive video and audio, etc... All the actions available with OpenVidu are managed via OpenVidu Browser.
* **OpenVidu Server**: it is an application that handles the server-side stuff. It receives the operations from OpenVidu Browser and do whatever is necessary to establish and manage your video-calls. You will never have to explicitly use it: just to run it and know its IP address.

Существуют разные роли во время конференции:

1. Publisher - пользователь, видео/аудио которого публикуется у других пользователей (например, пользователи могут видеть стрим его вебкамеры).
2. Subscriber - пользователь, видео/аудио которого НЕ публикуется у других пользователей



<https://openvidu.io/docs/cheatsheet/join-session/>

**Присоединение к сессии**

Сначала инициализируем объект Сессия:

var OV = new OpenVidu();

var session = OV.initSession();

Затем вызывая метод session.connect мы можем присоединится к правильно инициализированной сессии:

// After retrieving a token from OpenVidu Server...

session.connect(token)

.then( ... )

.catch( ... );

You must ask OpenVidu Server for a user token. To do so:

1. **Initialize a Session in OpenVidu Server**

var openvidu = new OpenVidu(OPENVIDU\_URL, OPENVIDU\_SECRET);

var properties = {}; openVidu.createSession(properties).then(session => { ... });

1. **Generate a token for this session in OpenVidu Server**

var tokenOptions = { role: "PUBLISHER", data: "user\_data" }; session.generateToken(tokenOptions).then(token => { ... });

1. **Return this token to your client-side to use it on**Session.connect()

Мы можем додавить опционально второй параметр для передачи пользовательских метаданных, которые будут получены каждым другим пользователем подключенным к этой же сессии.

session.connect(token, DATA)

.then( ... )

.catch( ... );

<https://openvidu.io/docs/cheatsheet/leave-session/>

**Покинуть сессию**

Вызывая метод session.disconnect мы можем покинуть сессию.

session.disconnect();

Это действие будет триггером для следующих событий:

* sessionDisconnected: отправляется (dispatched) объектом Session локального пользователя, который покидает сессию. Автоматически очищает все удаленные (дистанционные, remote) видео
* streamDestroyed: отправляется объектом Издатель локального пользователя, который покидает сессию (только если публиковал видео\аудио). Автоматически очищает локальные видео.
* streamDestroyed: отправляется (dispatched) объектом Publisher каждого удаленного (дистанционного) пользователя подключенного к сессии. Автоматически очищает дистанционное видео.
* connectionDestroyed: отправляется (dispatched) объектом Session каждого удаленного (дистанционного) пользователя подключенного к сессии.

<https://openvidu.io/docs/cheatsheet/subscribe-unsubscribe/>

**Подписка\отписка (subscrive\unsubscribe) от других пользовательских стримов (потоков)**

Для получения стрима издателя в сессии:

var subscriber;

session.on('streamCreated', (event) => { subscriber = session.subscribe(event.stream, targetElement); });

Новый HTML-видео-элемент будет добавлен в дом внутрь элемента targetElement. Это может быть HTMLElement или его id атрибут, и вы можете также указать, как вставить видео в соответствии с ним ( [SubscriberProperties.insertMode](https://openvidu.io/api/openvidu-browser/interfaces/subscriberproperties.html" \l "insertmode)).

Для отключения подписки и остановки медиа:

session.unsubscribe(subscriber);

Пока объект Subscriber доступен, мы можем поочередно вызывать Session.subscribe и Session.unsubscribe столько раз, сколько хотим.

<https://openvidu.io/docs/cheatsheet/publish-unpublish/>

**Публикация и отмена публикации стрима (Publish/Unpublish a stream)**

Мы можем публиковать стрим после подключения к сессии: получите Publisher объект из OpenVidu объекта (OpenVidu.initPublisher) и публикуйте его с Session объектом.

var publisher = OV.initPublisher(targetElement);

// Register all the events you want with 'publisher.on(...)'

session.publish(publisher);

// Method Session.publish must always be called after successfully connecting to session

* OpenVidu.initPublisher  метод будет выставлять в DOM новый HTML-видеоэлемент внутрь элемента targetElement, показывая твою камеру. Это может быть HTMLElement или его id атрибут, и вы можете также указать, как вставить видео в соответствии с ним (see [PublisherProperties.insertMode](https://openvidu.io/api/openvidu-browser/interfaces/publisherproperties.html#insertmode)). Мы можем потом публиковать его в сессии, когда захотим (вероятно мы захотим чтобы пользователь подтвердил что камера работает нормально, до его публикации).
* Session.publish метод будет запускать (trigger) у [Publisher](https://openvidu.io/api/openvidu-browser/classes/publisher.html) объекта streamCreated событие у локального пользователя, который публикует стрим, и бует запускать у [Session](https://openvidu.io/api/openvidu-browser/classes/session.html) объекта streamCreated событие у каждого удаленного (дистанционного) пользователя подключенного к сессии.

Мы можем добавить еще два параметра в initPublisher метод: объект со свойствами описывающими наш публикующий стрим и callback-функцию, которая будет выполняться сразу после завершения метода и до publisher объекта отправляющего accessAllowed или accessDenied события:

OV.initPublisher(

targetElement,

{

audioSource: undefined, // The source of audio. If undefined default audio input

videoSource: undefined, // The source of video. If undefined default video input

publishAudio: true, // Whether you want to start the publishing with audio unmuted or muted

publishVideo: true, // Whether you want to start the publishing with video enabled or disabled

resolution: '640x480', // The resolution of your video

frameRate: 30, // The frame rate of your video

insertMode: 'APPEND', // How the video will be inserted according to targetElement

mirror: false // Whether to mirror your local video or not

},

(error) => { // Function to be executed when the method finishes

if (error) {

console.error('Error while initializing publisher: ', error);

} else {

console.log('Publisher successfully initialized');

}

}

);

Для отмены публикации стрима:

session.unpublish(publisher);

Пока объект Publisher доступен, мы можем поочередно вызывать Session. publish  и Session. unpublish  столько раз сколько хотим. Также можно менять Publisher динамично (например если мы хотим остановить передачу нашей вебкамеры и начать делиться (sharing) нашим экраном, все это не покидая сессию). Для этого

session.unpublish(oldPubliser);

newPublisher = OV.initPublisher(newOptions);

// Register all the events you want with 'newPublisher.on(...)'

session.publish(newPublisher);

Примечание: только пользователи с ролью PUBLISHER или MODERATOR могут вызывать Session.publish метод. Вы можете ознакомится с [OpenViduRole](https://openvidu.io/api/openvidu-node-client/enums/openvidurole.html) секцией OpenVidu Node Client для получения полного описания.

<https://openvidu.io/docs/cheatsheet/mute-audio-video/>

**Выключить/ включить аудио и видео (Mute/Unmute audio and video)**

Вызывая Publisher.publishAudio() или Publisher.publishVideo() метод мы можем включать/выключать (toggle) аудио и видео у нашего Publisher объекта. Каждый пользователь подписанный (subscribed) на нам стрим будет останавливать полученное видео или аудио.

publisher.publishAudio(audioEnabled); // true to unmute the audio track, false to mute it

publisher.publishVideo(videoEnabled); // true to enable the video track, false to disable it

Вызывая Subscriber.subscribeToAudio() или Subscriber.subscribeToVideo() метод мы можем включать/выключать (toggle) аудио и видео у нашего Subscriber объекта. Вы будете останавливать получаемое аудио или видео.

subscriber.subscribeToAudio(audioEnabled); // true to unmute the audio track, false to mute it

subscriber.subscribeToVideo(videoEnabled); // true to enable the video, false to disable it

<https://openvidu.io/docs/cheatsheet/manage-videos/>

**Управление видео**

OpenVidu делает очень простым управление видео-плеерами (video players). Вы можете позволить OpenVidu заботиться об их жизненном цикле или можете импользовать свои HTML-видеоэлементы. Каждый Publisher и Subscriber объект могут отображать медиа стрим в стольких видео-плеерах, в скольких вы хотите.

**Даем OpenVidu позаботиться он наших видеоплеерах**

##### A) Publisher videos (видео Публикующего стрим)

* Когда инициализируется Publisher объект, мы можем вставить видеоплеер, передавая валидный targetElement в метод OpenVidu.initPublisher (это может быть HTMLElement или его  id атрибут). Второй параметр позволяет нам настраивать (customize) publisher-а стрим, в том числе как OpenVidu должен вставлять видеоплеер в соответствии targetElement (more info in [OpenVidu Browser Docs](https://openvidu.io/api/openvidu-browser/interfaces/publisherproperties.html)):
* var publisher = OV.initPublisher(targetElement, {insertMode: 'APPEND'});
* После получения Publisher объекта вы можете создать больше видеоплееров и отображать их медиапоток вызывая Publisher.createVideoElement. Передаем корректный (valid ) targetElement и [insertMode](https://openvidu.io/api/openvidu-browser/enums/videoinsertmode.html) (‘AFTER’, ’APPEND’, ‘BEFORE’, ‘PREPEND’, ‘REPLACE ):
* publisher.createVideoElement(targetElement, 'APPEND');

##### B) Subscriber videos (видео Подписчика)

* Когда подписываешься на Stream, ты можешь вставить видеоплеер передавая корректный targetElement в методе Session.subscribe (это может быть HTMLElement или его id атрибут). Третий параметр позволяет нам настраивать стрим подписчика, в том числе как OpenVidu будет вставлять видеоплеер в соответствии с targetElement (more info in [OpenVidu Browser Docs](https://openvidu.io/api/openvidu-browser/interfaces/subscriberproperties.html)):
* var subscriber;
* session.on('streamCreated', event => {
* subscriber = session.subscribe(event.stream, targetElement, {insertMode: 'APPEND'});
* });
* После получения Subscriber объекта, мы можем создать больше видеоплееров для отображения их медиа стрима вызывая Subscriber.createVideoElement. Передаем корректный targetElement и [insertMode](https://openvidu.io/api/openvidu-browser/enums/videoinsertmode.html):
* subscriber.createVideoElement(targetElement, 'APPEND');

Publisher и Subscriber объекты будут отправлять videoElementCreated  событие для каждого видео, вставленного в DOM этими методами. Также эти видео будут автоматически удалены из DOM, когда это будет необходимо, Publisher и Subscriber объекты будут отправлять videoElementDestroyed события каждому из них.

### Мы заботимся об видеоплеерах

##### A) Publisher videos (видео Публикующего стрим)

* Передаем undefined как targetElement когда инициализируем Publisher объект и когда мы имеем его доступным вызываем Publisher.addVideoElement метод, передавая уже существующий HTML-видеоэлемент из DOM:
* var publisher = OV.initPublisher(undefined, publisherProperties);
* publisher.addVideoElement(videoElement);

##### B) Subscriber videos (видео Подписчика)

* Передаем  undefined  как targetElement когда подписываемся на Stream, затем когда нам доступен Subscriber объект мы вызываем Subscriber.addVideoElement метод, передавая уже существующий HTML-видеоэлемент из DOM:
* var subscriber;
* session.on('streamCreated', event => {
* subscriber = session.subscribe(event.stream, undefined);
* subscriber.addVideoElement(videoElement);
* });

Этот способ управления виеоплеерами очень полезен, когда разрабатывается наше приложение с каким-то frontend декларативным (declarative) фреймворком, например **Angular**, **React** или **Vue.js**. Это позволяет нам добавлять в необходимые Publisher or Subscriber любые видеоэлементы управляемые фреймворком. К тому же, мы можем обрабатывать обоих Publishers and Subscribers одинаковым способом касательно их местонахожения (rendering) на странице, так как они оба наслудеют от родительского класса StreamManager.

<https://openvidu.io/docs/cheatsheet/share-data/>

**Распространение (sharing) данных между пользователями**

Какое бы приложение мы не разрабатывали, нам нужно будет передать какие-то данные другим пользователям, как минимум nickname. Мы можем это сделать в двух разных местах:

* **Client-Side**: когда вызываем Session.connect метод в openvidu-browser:

session.connect(token, "USER\_DATA")

.then( ... )

.catch( ... );

* **Server-Side**: когда генерируем маркер (token):

var tokenOptions = {

role: "PUBLISHER",

data: "user\_data" };

session.generateToken(tokenOptions).then(token => { ... });

Результат будет у всех клиентов, Connection  объекты будут иметь их data  свойства с подходящим значением, которое мы предоставили для каждого пользователя. В простейшем случае получить данные ассоциированные с любым пользователем можно будет так:

session.on("streamCreated", function (event) {

session.subscribe(event.stream, "subscriber");

console.log("USER DATA: " + event.stream.connection.data);

});

Уточнения:

* Использование только первого варианта (**Client-Side**) не безопасно, так как клиент может модифицировать значение второго параметра в Session.connect  методе. Этот вариант подходит доя разработки окружения или некритических данных. Если мы хотим полностью распространяемые данные, то нужно использовать server-side way .
* Connection.data будет обычной строкой, если мы предоставляем данные одним способом, или же это будет строка в формате "CLIENT\_SIDE\_DATA%/%SERVER\_SIDE\_DATA" (разделены символами %/%), если мы предоставляем данные на **Client-Side и Server-Side**.
* Мы можем выбрать любой формат строки, который нам нравится, если мы не планируем передавать данные сложнее чем простое поле (если нам нужно передать данные не примитивного типа, то формат JSON будет подходящим решением). Метод Session.connect в Openvidu Browser явно признает данными, если в параметре будет обычный объект (он будет преобразован в строку). Другими словами в качестве параметра может быть обычный объект.

<https://openvidu.io/docs/cheatsheet/send-messages/>

**Отправка сообщений между пользователями**

Любой пользователь подключенный к сессии может отправлять сообщения любым другим участникам сессии: сообщения всем участникам (broadcast messages), сообщения определенному пользователю, сообщения группе пользователей. Для этого:

// Sender of the message (after 'session.connect')

session.signal({

data: 'My custom message', // Any string (optional)

to: [], // Array of Connection objects (optional. Broadcast to everyone if empty)

type: 'my-chat' // The type of message (optional)

})

.then(() => {

console.log('Message successfully sent');

})

.catch(error => {

console.error(error);

});

Любые пользователи подписанные на этот *type*  будут получать сообщение:

// Receiver of the message (usually before calling 'session.connect')

session.on('signal:my-chat', (event) => {

console.log(event.data); // Message

console.log(event.from); // Connection object of the sender

console.log(event.type); // The type of message ("my-chat")

});

Мы можем подписаться на **all signals**, устанавливая обработчик события:

// Receiver of all messages (usually before calling 'session.connect')

session.on('signal', (event) => {

console.log(event.data); // Message

console.log(event.from); // Connection object of the sender

console.log(event.type); // The type of message

});

Мы можем отправлять сообщения определенным пользователям, добавляя в свойство to массив соответствующих Connection объектов:

// Sender of the adressed message (after calling 'session.connect')

session.signal({

data: 'My private custom message',

to: [connection1, connection2],

type: 'my-private-chat'

});

В этом случае только участники connection1 и connection2 объектов будут получать сигнальные события, если они подписаны на них. Мы можем получить Connection объекты подписываясь на connectionCreated событие до подключения к сессии:

this.session.on('connectionCreated', (event) => {

console.log(event.connection);

});

<https://openvidu.io/docs/advanced-features/screen-share/>

**Расшаривание экрана (screen share)**

Чтобы расшарить свой экран вместо вебкамеры используется тот же процесс, что и при Публикация и отмена публикации стрима (Publish/Unpublish a stream), но только устанавливаем в свойстве videoSource значение "screen"  во время инициализации Publisher объекта:

OV.initPublisher("html-element-id", { videoSource: "screen" });

**Chrome**, **Firefox** и **Opera** поддерживают расшаривание экрана. Chrome (и Opera) нуждаются в расширении, Firefox поддерживает встроенное (native) расширение с версии 52. OpenViduError объект может быть возвращенным со свойством [OpenViduError.name](https://openvidu.io/api/openvidu-browser/enums/openviduerrorname.html) в callback-е:

* SCREEN\_SHARING\_NOT\_SUPPORTED: если браузер не поддерживает расшаривание экрана.
* SCREEN\_EXTENSION\_NOT\_INSTALLED: браузеру Chrome нужно расширение чтобы позволить расшаривание экрана. error.message содержит URL из Chrome Web Store где можно установить необходимое расширение.
* SCREEN\_EXTENSION\_DISABLED: если экранное расширение Chrome установлено, но заблокировано.
* SCREEN\_CAPTURE\_DENIED: если пользователь не дал разрешение на захват экрана, когда браузер спросил об этом.

OV.initPublisher('html-element-id', { videoSource: "screen" }, function(error) {

if (error.name == 'SCREEN\_EXTENSION\_NOT\_INSTALLED') {

showWarning(error.message);

// showWarning could show a button with href 'error.message',

// so the user can navigate to install the extension.

// A browser refresh is also needed after installation

} else if (error.name == 'SCREEN\_SHARING\_NOT\_SUPPORTED') {

alert('Your browser does not support screen sharing');

} else if (error.name == 'SCREEN\_EXTENSION\_DISABLED') {

alert('You need to enable screen sharing extension');

} else if (error.name == 'SCREEN\_CAPTURE\_DENIED') {

alert('You need to choose a window or application to share');

}

});

Для **Firefox** две разных videoSource строки позволяют сделать расширивание экрана:

* "screen": entire screen (весь экран)
* "window": specific application window (конкретное окно приложения)

В Chrome и Opera свойство "screen"  дает доступ к entire screen или specific application windows.

# Пользовательское (Custom) приложение расшаривания экрана для Chrome

OpenVidu предоставляет стандартное (default) расширение, которое будет работать на любом домене, но мы можем создать своё собственное расширение, которое будет основываться на стандартном. В таком случае наше расширение нуждается в иконке, названии, описании, пользовательских корректных (valid) доменах.  
Для использования расширения необходимо конфигурировать OpenVidu объект после его инициализации:

var OV = new OpenVidu();

OV.setAdvancedConfiguration(

{ screenShareChromeExtension: "https://chrome.google.com/webstore/detail/EXTENSION\_NAME/EXTENSION\_ID" }

);

Более детальная информация: [GitHub README](https://github.com/OpenVidu/openvidu-screen-sharing-chrome-extension)