Agora Web SDK

Reference Manual v1.2

support@agora.io

March 2016

Contents

Introduction to Agora Web SDK

Agora Communications as a Service (CaaS) provides ensured Quality of Experience for worldwide Internet-based voice and video communications through a virtual Agora Global Network that is especially optimized for real-time and mobile-to-mobile communications. Agora CaaS solves quality of experience challenges for mobile devices, 3G/4G/Wi-Fi networks with varying performance, and global Internet bottlenecks.

Agora CaaS includes mobile-optimized Agora Native SDKs for iOS and Android smartphones that provide access to the Agora Global Network along with device-specific mobile optimizations. Agora Native SDK is also available for Windows, and will be available for Mac in the future. Applications using the Native SDK link it directly into the application executable when they are built. These other SDKs are documented elsewhere in separate Reference Manuals.

The **Agora Web SDK** documented here is an additional Agora CaaS capability that provides open access to the Agora Global Network from any device that supports a standard WebRTC-compliant web browser, without requiring any downloads or plugins.

Agora Web SDK is a JavaScript library that is loaded by an HTML web page, as with any other JavaScript library. It also provides a set of simple high-level JavaScript APIs for establishing voice and video communications with other users across the Agora Global Network. The Agora Web SDK library uses the primitive WebRTC APIs in the browser to establish connections and control the voice and video media, while providing a simpler high-level interface for developers.

The Agora Web SDK allows JavaScript code to:

- Join and leave shared Agora sessions (identified by unique channel names) where there may be many global users on a conference together.
- Set various voice and video parameters which help the Agora SDK optimize communications. Most of the SDK operations are automated and do not require any developer intervention if these parameter settings are not provided.
- Manipulate voice and video media streams, controlling whether they are muted or seen and where within the page's HTML framework they will be seen or heard.
- Support multiple voice and video streams simultaneously which will occur in multi-party conference applications.

The Web SDK provides three straightforward JavaScript classes to deliver these features, which support:

- A single **Client** object for establishing and controlling sessions.
- Multiple **Stream** objects for managing different voice and video media streams.
- Top-level **AgoraRTC** object for creating the appropriate **Client** and **Stream** objects.

Getting Started with Agora Web SDK

This section explains how to use the Agora Web SDK, including related documentation, supported web browsers, release content, deployment processes, how to run the sample code, dynamic keys, and issues and limitations.

Compatibility and Related Documentation

This v1.2 release of the Agora Web SDK is compatible with the use of the Agora Native SDK v1.2, which supports compatible voice and video capabilities. The Native SDK is used to build native applications for Android, iOS and Windows, while the Web SDK described here is used for browser-apps running within WebRTC-compatible browsers.

If developing both native and web browser applications, please also refer to:

- Agora Native SDK for iOS Reference Manual - v1.2 or later
- Agora Native SDK for Android Reference Manual - v1.2 or later
- Agora Native SDK for Windows Reference Manual - v1.2 or later

This documentation, as well as complete SDK packages for all platforms, can all be downloaded via www.agora.com/developer.

Supported Web Browsers

The Agora Web SDK requires a WebRTC-compliant web browser and has been tested and verified on all the browser types and versions listed below, which are available across a wide range of platforms. Note that Microsoft Internet Explorer and Apple Safari do not support WebRTC and so do not support Agora Web SDK. Due to Apple limitations there are currently no WebRTC-compliant browsers for Apple iOS, which therefore requires native applications to implement embedded real-time communications.

Recent browser releases, starting with Chrome v47 and Opera v34, have increased security by not allowing WebRTC use from HTTP domains. We recommend that developers follow Google's lead by moving to HTTPS domains for all WebRTC and Agora Web SDK applications.

	Chrome v42-46	**Chrome v47**	**Firefox v42**	**Opera v34**	**QQ v9.0**	**360 v8.5**	**Baidu v7.6**	**Sougou v6.0**
HTTP	Y	N	Y	N	Y	Y	Y	Y
HTTPS	Y	Y	Y	Y	Y	Y	Y	Y

Note: All the browser versions listed have been tested and verified. Earlier versions of these browsers that support WebRTC may work, but have not been verified by Agora.io.

Release Content

The Agora Web SDK is delivered as a package of the following files that can be simply unpacked on your development systems and on your production web server when you are ready to deploy.

Component	Description
./doc	Agora Web SDK Documentation: Agora_Web_SDK_Release_Notes_v1_EN.pdf Agora_Web_SDK_API_Reference_v1_EN.pdf
./client	Sample web application.
./server	Web server-side sample code and libraries for dynamic key generation.

Obtain your Vendor Key

Developers must obtain a **vendor key** and a **sign key** from Agora.io (accessible on your Dashboard page after logging in at Agora.io, or contact support@agora.io), which are needed in using the API to access the Agora Global Network. Either the vendor key is used directly or, for more security, it is combined with the sign key within your web server code to generate a secure **dynamic key**. See the section below on dynamic keys, and the Sample Application for examples of these two uses.

To obtain your **vendor key**:

a) Sign up for a new account at agora.io (top left of page as shown):

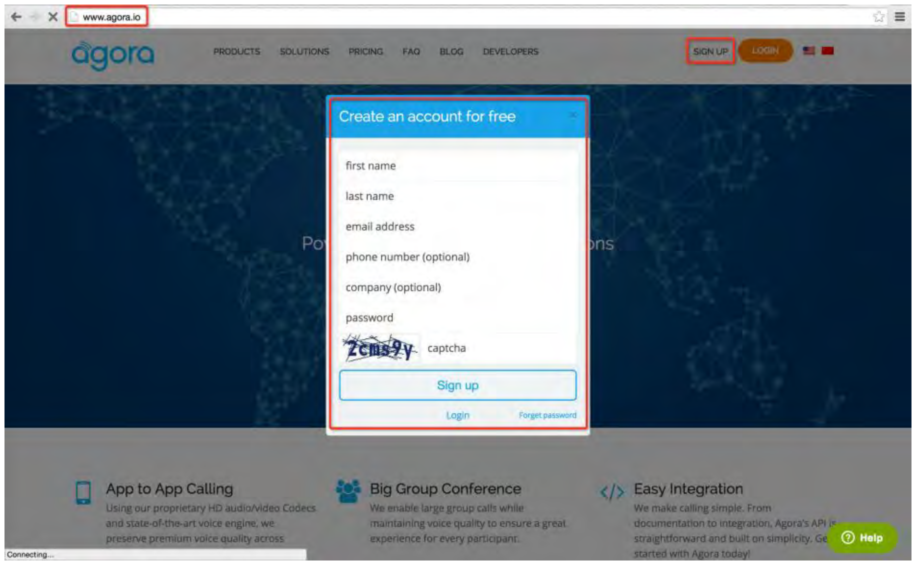

Figure 1

b) Complete the sign-up process and you will find your API vendor key, once it has been created, on the completion page at [http://dashboard.agora.io/,](http://dashboard.agora.io/) as shown in the following screenshot. You will also receive an email with your API vendor key.

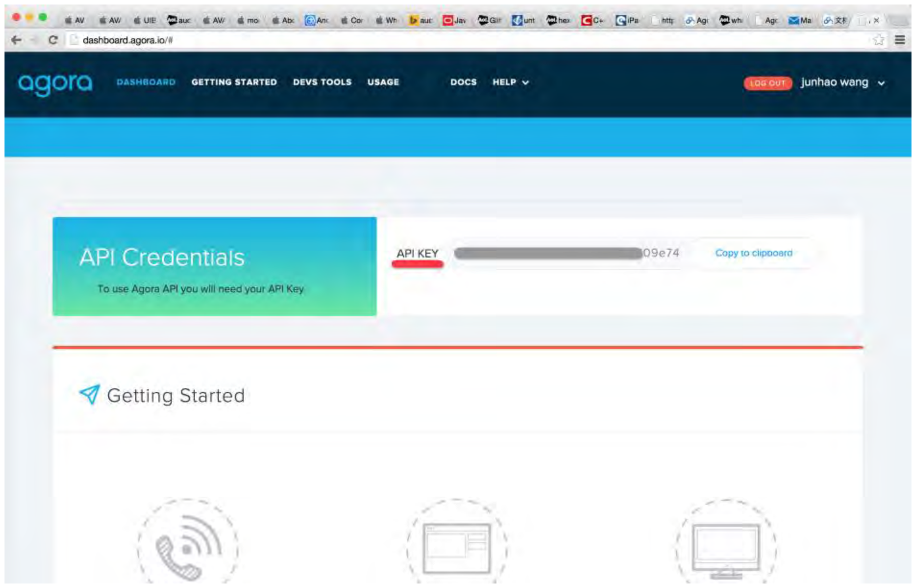

Figure 2

c) Once your **vendor key** is assigned, you can start using Agora.io services with this key.

Application Development and Deployment

Applications using the Agora Web SDK are completely standard JavaScript applications that just need to load the Agora JS library, which will also need access to the JS extension libraries provided with the SDK. Just follow your normal JavaScript hosting procedures when deploying these JavaScript libraries.

Note: 1) Load the Agora JS library AgoraRTCSDK-1.2.0.js, you can download it from

https://rtcsdk.agora.io/AgoraRTCSDK-1.2.0.js or http://rtcsdk.agora.io/AgoraRTCSDK-1.2.0.js

2) Required JS extension libraries (can be found under the **/client/js/** folder in the release package):

jquery.js – version >= v2.0.0

socket,io.js – version >= 1.3.6

adapter.js – version >= 0.2.5

How to Run the Sample Web Application

Client:

To run the provided sample web application, you need to have a local web server installed, such as Apache, NginX or Node.js. Simply deploy the files under **./client/** to your web server and launch your http/https service. Access the sample application page on your web server using one of the browsers listed in the **Supported Browsers** section. Google Chrome is recommended.

The default sample code only requires your static **vendor key** (which is simply entered into the sample application web page). Follow the steps below if you want to try the sample application with a **dynamic key**:

- Uncomment the code marked by 'for dynamic key' in index.html
- Replace the 'ip:port' with your dynamic key-generation server URL (below)
- Set up and launch the key-generation server as described below

Server:

This code is only needed if you want to experiment with using more secure **dynamic keys**. In production use, you (developers) should integrate this logic into your own server-side applications and (re)code this in the programming languages you are already using for your server-based functionality. The sample code is in JavaScript and requires a standard Node.js server.

- Install a standard Node.js server within your server or cloud infrastructure
- Run 'npm install' under ../server/nodejs/
- Fill in your VENDOR_KEY and SIGN_KEY values in ../server/nodejs/DemoServer.js.
- Launch the server with 'node DemoServer.js'

Note: Please contact support@agora.io to obtain your unique vendor key and sign key if you do not have them.

Use of Dynamic Keys for Increased Security

Each "vendor" organization using the Agora SDK (across platforms) has a unique **vendor key** that identifies that organization. All the communication sessions that are created across the Agora Global Network for one **vendor key** are isolated from all other sessions for other **vendor keys**. Therefore, communication sessions will not be connected together across vendors. Statistics, management and billing are all separated according to your **vendor key**. If an organization has multiple applications that should be kept completely separate (perhaps they are built by completely different teams), then they would use multiple **vendor keys** for this. If applications need to communicate with each other, then a single **vendor key** should be used.

Vendor keys are used in the Client **init()** method to identify your organization to the Agora Global Network and then a subsequent **join()** method is used to join a one-to-one or conferencing session using a unique channel name for your Vendor Key. Using your **vendor key** directly is easy and works well for the initial development of applications.

However, if someone else illicitly obtains your **vendor key,** they will be able to perform the same operations in their own client applications, allowing them to join sessions that belong to you and are billed to you. To prevent this and make your applications more secure you should consider using **dynamic keys** for your large-scale production applications.

With this more secure approach, an additional **dynamic key** is used instead of the **vendor key** in the Client init() method. **Dynamic keys** must be constructed within the server-side logic for your application for each init() the client application performs. On your server, the channel name you want to use is cryptographically encoded using your Vendor Key, an additional unique **sign key** provided by Agora.io that should never be made visible to any client, and additional timestamp and random number information. This encoded **dynamic key** should then be passed back down to the client application and used in the init() call. When the Agora Global Network receives this information it can determine that it must have come from your application because only your server-code and Agora.io know the unique **sign key**.

When using **dynamic keys**, the join() call must specify the same channel name previously encoded into the dynamic key passed to the earlier init() call. To join another different session after a leave() it is necessary to call init() again with a new dynamic key that encodes the new channel name.

Note: This new-channel sequence is slightly different in the Web SDK versus the Native SDK where dynamic keys are passed to the join instead. This change has the benefit for the Web SDK that both the **vendor key** and the **sign key** can be kept only on your server and never passed down to the client browser, so neither is visible to any client.

Also note: the sample code is an example to show the concept. In production use you should incorporate this logic into your secure server code so you do not simply have a general purpose “key constructor” service openly visible on the Internet.

Please contact support@agora.io to discuss use of **dynamic keys** if required for your production applications and to get your **sign key** and additional details. Agora.io can provide sample server-side code for the encoding operation in JavaScript for Node.js (included with the Web SDK sample app), in Java, and in C. The encoding uses the industry standard HMAC/SHA1 approach (see https://en.wikipedia.org/wiki/Hash-based_message_authentication_code) for which there are libraries on most common server-side development platforms, such as Node.js, PHP, Python, Ruby and others.

Known Issues and Limitations

1. Agora Web SDK is based on WebRTC technology. Certain browsers, including Microsoft Internet Explore and Apple Safari, and platforms, including Apple iOS, currently do not support WebRTC. See the **Supported Browsers** section above.
2. Agora Web SDK supports video profiles up to 1080p resolution if the client has a true HD camera installed. However, the maximum resolution will be limited by the camera device capabilities.
3. The maximum video bandwidth setting is not currently supported in Mozilla Firefox.
4. Agora Web SDK v1.2 does not currently provide certain functionality that is available in the Agora Native SDK. This includes quality indicators, testing services, providing feedback ratings for sessions, recording and logging.

Agora Web SDK API Reference – JavaScript Classes

The Agora Web SDK library includes the following classes:

AgoraRTC	Use the AgoraRTC object to create Client(s) and Stream objects.
Client	Represents the web client object which provides access to core AgoraRTC functionality.
Stream	Represents the local/remote media streams in a session.

1. AgoraRTC API Methods

`createClient()`

This method creates and returns a Client object. It should be called only once per session.

Sample code:

```
var client = AgoraRTC.createClient();
```

`getDevices(callback)`

This method enumerates platform camera/microphone devices.

Parameter Name	Type	Description
callback	Function	The callback function to retrieve the devices information. For ex. callback(devices): devices[0].deviceId: device ID devices[0].label: device name in string devices[0].kind: ‘audioinput’, ‘videoinput’

Sample code:

```
AgoraRTC.getDevices (function(devices) {
    var dev_count = devices.length;
    var id = devices[0].deviceId;
});
```

`createStream(spec)`

This method creates and returns a Stream object.

Parameter Name	Type	Description

Parameter Name	**Type**	**Description**
spec	Object	This object contains the following properties: streamID: represents the stream ID, normally set to uid which can be retrieved from the `client.join` callback. audio: (flag) true/false, marks whether this stream contains an audio track. video: (flag) true/false, marks whether this stream contains a video track. screen: (flag) true/false, marks whether this stream contains a screen sharing track. It should be set to false in the current version. (optional) cameraId: the camera device ID retrieved from the getDevices method. (optional) microphoneId: the microphone device ID retrieved from the getDevices method.

Sample code:

```
var stream = AgoraRTC.createStream({streamID: uid, audio:true, video:true, screen:false});
```

2. Client API Methods and Callback Events

Represents the web client object that provides access to core AgoraRTC functionality.

Methods

`init(key, onSuccess, onFailure)`

This method initializes the Client object.

Parameter Name	Type	Description
key	String	Key can be one of the following: - vendor key: provided by Agora during registration. - dynamic key: the token generated with vendor key and sign key. A NodeJS implementation of token-gen algorithm is provided. This is the safest and recommended way to access the Agora Global Network.
onSuccess	function	(optional) the function to be called when the method succeeds.
onFailure	function	(optional) the function to be called when the method fails.

Sample code:

```
client.init(vendorKey, function() {
    log(“client initialized”);
    //join channel
    ……
}, function(err) {
    log(“client init failed ”, err);
    //error handling
});
```

`join(channel, uid, onSuccess, onFailure)`

This method allows the user to join an AgoraRTC channel.

Parameter Name	Type	Description
channel	String	A string providing the unique channel name for the AgoraRTC session.
uid	String	The user ID: a 32-bit unsigned integer ranges from 1 to (2^32-1). It must be unique. If set to ‘undefined’, the server will allocate one and returns it in `onSuccess` callback.

Parameter Name	Type	Description
onSuccess	function	(optional) The function to be called when the method succeeds, will return the uid which represents the identity of the user.
onFailure	function	(optional) the function to be called when the method fails.

Sample code:

```
client.join('1024', undefined, function(uid) {
   log("client " + uid + "" joined channel");
   //create local stream
   ……
}, function(err) {
   log("client join failed ", err);
   //error handling
});
```

`leave(onSuccess, onFailure)`

This method allows the user to leave an AgoraRTC channel.

Parameter Name	Type	Description
onSuccess	function	(optional) The function to be called when the method succeeds.
onFailure	function	(optional) The function to be called when the method fails.

Sample code:

```
client.leave(function() {
   log("client leaves channel");
   ……
}, function(err) {
   log("client leave failed ", err);
   //error handling
});
```

`publish(stream, onFailure)`

This method publishes a local stream to the server.

Parameter Name	Type	Description
stream	object	Stream object, which represents the local stream.

Parameter Name	Type	Description
onFailure	function	(optional) The function to be called when the method fails.

Sample code:

```
        client.publish(stream, function(err) {
            log.("stream published");
            ……
        })
```

`unpublish(stream, onFailure)`

This method unpublishes the local stream.

Parameter Name	Type	Description
stream	object	Stream object which represents local stream.
onFailure	function	(optional) the function to be called when the method fails.

Sample code:

```
        client.unpublish(stream, function(err) {
            log.("stream unpublished");
            ……
        })
```

`subscribe(stream, onFailure)`

This method subscribes remote stream from the server.

Parameter Name	Type	Description
stream	object	Stream object, which represents the remote stream.
onFailure	function	(optional) The function to be called when the method fails.

Sample code:

```
        client.subscribe(stream, function(err) {
            log.("stream unpublished");
            ……
        })
```

`unsubscribe(stream, onFailure)`

This method unsubscribes the remote stream.

Parameter Name	Type	Description
stream	object	Stream object, which represents remote stream.
onFailure	function	(optional) The function to be called when the method fails.

Sample code:

```
        client.unsubscribe(stream, function(err) {
           log.("stream unpublished");
           ……
        })
```

Events

`stream-published`

Notify the application that the local stream has been published.

Sample code:

```
client.on('stream-published', function(evt) {
   log.("local stream published");
   ……
        })
```

`stream-added`

Notify the application that the remote stream has been added.

Sample code:

```
client.on('stream-added', function(evt) {
   var stream = evt.stream;
   log("new stream added ", stream.getId());
   //subscribe the stream
   ……
        })
```

`stream-removed`

Notifies the application that the remote stream has been removed, (i.e., a peer user called unpublish stream).

Sample code:

```
client.on('stream-removed, function(evt) {
    var stream = evt.stream;
    log.("remote stream was removed", stream.getId());
    ……
        })
```

`stream-subscribed`

Notifies the application that the remote stream has been subscribed.

Sample code:

```
        client.on('stream-subscribed', function(evt) {
            var stream = evt.stream;
            log("new stream subscribed ", stream.getId());
            //play the stream
            ……
        })
```

`peer-leave`

Notifies the application that the peer user has left the room, (i.e., peer user called client.leave())

Sample code:

```
client.on('peer-leave', function(evt) {
    var uid = evt.uid;
    log("remote user left ", uid);
    ……
        })
```

Error Code

The onFailure callback function returns the following error codes:

Value	Description
10	Invalid user key.
11	Invalid user operation.
12	Invalid local stream.
13	Invalid remote stream.
100	Socket connection error.
101	Connect to web service failed.

Value	**Description**
102	Peer connection lost.
1000	Service not available.
1001	Join channel failed.
1002	Publish stream failed.
1003	Unpublish stream failed.
1004	Subscribe stream failed.
1005	Unsubscribe stream failed.

3. Stream API Methods

`init(onSuccess, onFailure)`

This method initializes the Stream object.

Parameter Name	Type	Description
onSuccess	function	(optional) The function to be called when the method succeeds.
onFailure	function	(optional) The function to be called when the method fails.

Sample code:

```
stream.init(function() {
   log(“local stream initialized”);
   // publish the stream
   ……
}, function(err) {
   log(“local stream init failed ”, err);
   //error handling
        });
```

`getId()`

Retrieves the stream id.

`getAttributes()`

Retrieves the stream attributes.

`hasVideo()`

Retrieves video flag.

`hasAudio()`

Retrieves the audio flag.

`enableVideo()`

Enables video track in the stream; it only works when video flag was set to true in `stream.create` and acts like video resume.

`disableVideo()`

Disables video track in the stream, only works when video flag was set to true in `stream.create` and acts like video pause.

`enableAudio()`

Enables audio track in the stream and acts like audio resume.

`disableAudio()`

Disables audio track in the stream and acts like audio pause.

`setVideoProfile(profile)`

Sets the video profile. It is optional and only works before calling stream.init().

Parameter Name	Type	Description
profile	String	The video profile. It can be set to one of the following: '120p_1' : 160x120, 15fps, 80kbps '240p_1' : 320x240, 15fps, 200kbps '480p_1' : 640x480, 15fps, 500kbps (default) '480p_2' : 640x480, 30fps, 1Mbps '720p_1' : 1280x720, 15fps, 1Mbps '720p_2' : 1280x720, 30fps, 2Mbps '1080p_1' : 1920x1080, 15fps, 1.5Mbps '1080p_2' : 1920x1080, 30fps, 3Mbps

Sample code:

```
stream.setVideoProfile('480p_1');
```

`play(elementID, assetsURL)`

Plays the video/audio stream.

Parameter Name	Type	Description
elementID	String	Represents the html element id.
assetsURL	String	(optional) The URL of the resource file. By default the files are located under /assets/ folder.

Sample code:

```
stream.play('div_id', '/res'); // stream will be played in div_id element, resource files
under /res/assets/
```

`stop()`

Stops the video/audio stream and clears the display DOM tree under elementID (specified in the play method).

`close()`

Closes the video/audio stream. The camera and microphone authorization will be cancelled after calling this method.