Web Real-Time Communications Developer's Guide

Using the API

12/12/2025

Contents

Using the API

The Genesys WebRTC JavaScript API is easy to use, requiring only a few simple steps to register with the WebRTC Gateway and establish media sessions.

Configuration

For information on how to configure your application, see Grtc.Client Instance Attributes.

Support Scripts

To use WebRTC on a browser page, you must include the following JavaScript in your HTML source:

```
<script type="text/javascript" src="<path>/grtc.js"></script>
```

Call Handling

This section describes the things you must do to conduct a call.

Connect to and Register with the Gateway

Follow these steps in order to connect to and register with the WebRTC Gateway:

1. Create an instance of `Grtc.Client`. For example:

   ```
   var configuration = {
      'webrtc_gateway': 'http://WebRTC.genesyslab.com:8086',
      'stun_server': 'stun.genesyslab.com:3478',
      'dtls_srtp' : true
   };
   var grtcClient = new Grtc.Client(configuration);
   ```

   Note: The HTTP port of the WebRTC Gateway is configurable using the parameter `rsmp.http-port`.

2. Invoke either the `connect()` method to connect anonymously or the `register(myDN)` method to establish a connection using a DN that is registered with the SIP Server. The following example demonstrates the `register` method:

   ```
   // set the callback to handle the event when registration is successful
   grtcClient.onRegister.add(onRegisterCallback);
   // set the call back to handle the event when registration fails
   grtcClient.onFailed.add(onFailedCallback);
   // now the client tries to register as DN 1020
   grtcClient.register("1020");
   ```

The result of the `register` call is indicated by two events: the `onRegister` event indicating the success of registration, and the `onFailed` event indicating an error. So, as the example shows, you need to add event handlers (callbacks) if you want to perform specific tasks when such events happen.

Enable Local Media Stream

You can enable user media either before or after connecting to the gateway, but it is normally done afterwards. Use these two methods:

- enableMediaSource()
- setViewFromStream()

Here is an example:

```
// set the callback to handle the success event
grtcClient.onMediaSuccess.add(function (obj) {
    // on success, port the local media in an HTML5 video element
    grtcClient.setViewFromStream(document.getElementById("localView"), obj.stream);
});
// set the callback to handle the failure event
grtcClient.onMediaFailure.add(function (obj) {
    window.alert(obj.message);
});
// enable local media source, with audio set to true, and video
// set by a constraint object where the width of video is specified
grtcClient.enableMediaSource(true, {mandatory:{minWidth:360}});
```

The result of the `enableMediaSource` call is indicated by two events: the `onMediaSuccess` event indicating the success of accessing the local media stream, and the `onMediaFailure` event indicating an error. This means you need to add event handlers (callbacks) to properly deal with these events. Typically, you should handle the `onMediaSuccess` event by retrieving the local media stream from the argument object, and then call the `setViewFromStream` method to attach the stream to an HTML5 <video> or <audio> element. The HTML code for the video element might look something like this:

```
<video width="160" height="120" id="localView" autoplay="autoplay">
```

Establish the Call

In order to establish a call, both peers must have carried out the two steps mentioned above. After that, the caller can invoke `makeCall(remoteDN)` and the callee can invoke `acceptCall()`, as described in the following sections.

Caller makeCall()

1. Create a `Grtc.MediaSession` object before making your call.
2. Set up a callback to handle the event that is triggered when the remote stream arrives. This is normally done by calling `setViewFromStream` to attach the remote stream to an HTML5 audio or video element.
3. You can also attach data to the call, such as a customer name and phone number to be sent to an agent.
4. Now you can make the call.

Here is an example:

```
// create a new session (passing the client object as argument)
var grtcSession = new Grtc.MediaSession(grtcClient);
// set callback to port remote media stream when it comes
grtcSession.onRemoteStream.add(function (data) {
    grtcClient.setViewFromStream(document.getElementById("remoteView"), data.stream);
});
// attach data if available: the data is an array of objects
// each object contains two properties named “key” and “value”
var dataToAttach = [
    {
        "key": "Name",
        "value": $('#login_name').val()
    },
    {
        "key": "Phone",
        "value": $('#phone_number').val()
    },
    {
        "key": "Email",
        "value": $('#email').val()
    }
];
grtcSession.setData(dataToAttach);
// preparation is ready, now make the call
grtcSession.makeCall(“1021”);
```

Callee acceptCall()

The JavaScript API fires an `onIncomingCall` event when an `OFFER` message is received. The callee web app is expected to handle this event by creating a `Grtc.MediaSession` instance, informing the user of the incoming call, and if the user authorizes, invoking `Grtc.MediaSession.acceptCall()` to establish a call session.

Here is an example:

```
grtcClient.onIncomingCall.add(function (data) {
    // create a session
    var grtcSession = new Grtc.MediaSession(grtcClient);
    // register a handler when remote stream is available
    grtcSession.onRemoteStream.add(function (data2) {
        grtcClient.setViewFromStream(document.getElementById("remoteView"), data2.stream);
    });
    // inform user of incoming call
    var userResponse = window.confirm(“Accept call from “ + data.peer + “?”);
    if (userResponse === true) {
        // accept the call
        grtcSession.acceptCall();

        // process attached data if available
        var dataAttached = grtcSession.getData();
        if (dataAttached) {
            // dataAttached is an array of objects, each having 2 properties "key" and "value"
            for (var i=0; i<dataAttached.length; ++i) {
                var obj = dataAttached[i];
                console.log("dataAttached[" + i + "]:" + obj.key + ", " + obj.value);
        }
    }
} else {
    grtcSession.rejectCall();
```

```
}
});
```

Terminate the Call

You can explicitly terminate a call by invoking `Grtc.MediaSession.terminateCall()`, followed by `Grtc.Client.disableMediaSource()`, as shown in this example:

```
grtcSession.terminateCall();
grtcClient.disableMediaSource();
// ... other processing, for example, call status update on the web page
```

If the remote peer terminates a call, the local peer needs to respond accordingly. You can prepare for this action by registering the `onPeerClosing` event on the client right after the client instance has been created, as shown in this example:

```
var grtcClient = new Grtc.Client(configuration);
// Set up other callbacks, as shown above...
grtcClient.onPeerClosing.add(function () {
    grtcClient.disableMediaSource();
    // ... other processing, for example, call status update on the web page
});
```

Backbone Example

This section provides two HTML files, one as the caller and the other as the callee, in order to give a full picture of how to use the WebRTC JavaScript API to establish a simple WebRTC call. What this example illustrates is very simple: the callee connects to the WebRTC Gateway using DN 1020, and the caller connects anonymously and calls the callee.

caller.html:

```
<html>
<head>
<style> video { border: 5px solid gray; } </style>
<script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
<script type="text/javascript" src="../../src/jsapi/classes/grtc.js"></script>
<script type="text/javascript">
    var conf = {
        "webrtc_gateway": "http://WebRTC.genesyslab.com:8086",
        "stun_server": "stun.genesyslab.com:3478",
        "dtls_srtp" : true
    };
    // construct a Grtc.Client instance
    var grtcClient = new Grtc.Client(conf);
    var grtcSession = null;

    // add a handler to do some work when the peer closes
    grtcClient.onPeerClosing.add(function () {
        $("#remoteStatus").empty();
        if (grtcSession) grtcSession = null;
    });

    // add a handler to disconnect when window is closed
    window.onbeforeunload = function() {
        grtcClient.disconnect();
```

```
    };

    grtcClient.onMediaSuccess.add(function (obj) {
        grtcClient.setViewFromStream(document.getElementById("localView"), obj.stream);
        grtcClient.onConnect.add(function () {
            $("#localStatus").empty();
            $("#localStatus").append("connected anonymously");
            // create a MediaSession instance and make a call on it
            grtcSession = new Grtc.MediaSession(grtcClient);
            grtcSession.onRemoteStream.add(function (data) {
                grtcClient.setViewFromStream(document.getElementById("remoteView"),
data.stream);
            });
            grtcSession.makeCall("1020");
        });
        grtcClient.onFailed.add(function (e) { window.alert(e.message); });
        grtcClient.connect();
    });
    grtcClient.onMediaFailure.add(function (obj) {
        window.alert(obj.message);
    });
    // enable microphone and camera
    grtcClient.enableMediaSource();

    function terminateCall() {
        grtcSession.terminateCall();
        grtcSession = null;
        $("#remoteStatus").empty();
    }

</script>
</head>

<body>
<div>
<input type="button" style="text-align:left;width:100px;" value="Terminate Call"
onClick="terminateCall();">
</div>

<div>
    <table>
        <tr> <td> local view </td> <td> remote view </td> </tr>
        <tr>
            <td> <video width="160" height="120" id="localView" autoplay="autoplay" controls>
</td>
            <td> <video width="160" height="120"id="remoteView" autoplay="autoplay" controls>
</td>
        </tr>
        <tr> <td> <span id="localStatus"></span> </td> <td> <span id="remoteStatus"></span>
</td> </tr>
    </table>
</div>

</body>
</html>
```

callee.html:

```
<html>
<head>
<style> video { border: 5px solid gray; } </style>
<script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
<script type="text/javascript" src="../../src/jsapi/classes/grtc.js"></script>
```

```
<script type="text/javascript">
    var conf = {
        "webrtc_gateway": "http://WebRTC.genesyslab.com:8086",
        "stun_server": "stun.genesyslab.com:3478",
        "dtls_srtp" : true
    };
    // construct a Grtc.Client instance
    var grtcClient = new Grtc.Client(conf);
    var grtcSession = null;

    // callee needs to register a handler to deal with incoming call
    grtcClient.onIncomingCall.add(function (data1) {
        // create a MediaSession to handle incoming call
        grtcSession = new Grtc.MediaSession(grtcClient);

        // register a handler when remote stream is available
        grtcSession.onRemoteStream.add(function (data2) {
            grtcClient.setViewFromStream(document.getElementById("remoteView"), data2.stream);
        });

        // ask user to confirm whether to accept or reject call
        var user_said = window.confirm("Do you want to accept the call from " + data1.peer +
"?");
        if (user_said === true) {
            $("#remoteStatus").empty();
            $("#remoteStatus").append("call from " + data1.peer);
            grtcSession.acceptCall();
        } else {
            grtcSession.rejectCall();
            grtcSession = null;
        }
    });

    // add a handler to do some work when the peer closes
    grtcClient.onPeerClosing.add(function () {
        $("#remoteStatus").empty();
        if (grtcSession) grtcSession = null;
    });

    // add a handler to disconnect when window is closed
    window.onbeforeunload = function() {
        grtcClient.disconnect();
    };

    grtcClient.onMediaSuccess.add(function (obj) {
        grtcClient.setViewFromStream(document.getElementById("localView"), obj.stream);
        // once microphone and camera are enabled, connect to gateway
        grtcClient.onRegister.add(function () {
            $("#localStatus").empty();
            $("#localStatus").append("connected as 1020");
        });
        grtcClient.onFailed.add(function (e) { window.alert(e.message); });
        grtcClient.register("1020");

    });
    grtcClient.onMediaFailure.add(function (obj) {
        window.alert(obj.message);
    });
    // enable microphone and camera
    grtcClient.enableMediaSource();

    function terminateCall() {
        grtcSession.terminateCall();
```

```
        grtcSession = null;
        $("#remoteStatus").empty();
    }
</script>
</head>

<body>
<div>
<input type="button" style="text-align:left;width:100px;" value="Terminate Call"
onClick="terminateCall();">
</div>

<div>
    <table>
        <tr> <td> local view </td> <td> remote view </td> </tr>
        <tr>
            <td> <video width="160" height="120" id="localView" autoplay="autoplay" controls>
</td>
            <td> <video width="160" height="120" id="remoteView" autoplay="autoplay"
controls> </td>
        </tr>
        <tr> <td> <span id="localStatus"></span> </td> <td> <span id="remoteStatus"></span>
</td> </tr>
    </table>
</div>

</body>
</html>
```

Browser Interoperability

Vendors tend to add their own prefixes to public API interfaces before the API standard has been finalized. For example, for the `getUserMedia` API specified by the W3C, Chrome provides this API as `webkitGetUserMedia`, while Firefox provides it as `mozGetUserMedia`. Other vendors may use other names. Refer to https://webrtc.org/web-apis/interop/ for a quick summary of the naming issues. The Genesys WebRTC JavaScript API has integrated the "polyfill" library suggested on that page in order to take care of some of the interoperability issues and allow developers to write to the unprefixed W3C standard names.

Known Issues

Firefox Renegotiation Issue

Firefox does not currently support renegotiation of an ongoing media session: once a media session has been set up, its parameters are fixed. For all practical purposes, this means that you cannot, for example, start an audio-only call and then add video to that same `PeerConnection` later in that session.

In order to add video mid-call, the recommended workaround is to destroy the audio-only `PeerConnection` and create a new `PeerConnection` that uses both audio and video. This is demonstrated in Demo Number 3, which comes with the WebRTC JSAPI IP. If you wish to track this issue, the current Firefox bug can be found at https://bugzilla.mozilla.org/show_bug.cgi?id=857115.

Note: This issue has now been resolved by Firefox. However, the default behavior in JSAPI has not changed, so it will still create a new `PeerConnection` on every renegotiation. This behavior in JSAPI can be overridden to reuse the same `PeerConnection` by calling the `Grtc.Client` method `setRenewSessionOnNeed(false)` with the value `false` during the initialization part in the client application. Also, the WebRTC Gateway option `rsmp.new-pc-support` must be set to `0` for this to work.

Mobile Browser Support

- Google Chrome for Android supports WebRTC in version 29 and higher.
- Mozilla Firefox for Android supports WebRTC in version 24 and higher.
- Opera for Android supports WebRTC in version 20 and higher.

Note that the following problems may be noticed when using WebRTC in Android browsers:

- Audio may sound choppy and warbled, especially on devices where the CPU is under a heavy load
- Acoustic echo cancellation on mobile platforms may not work well
- For Chrome, DTLS on Android can fail and may need to be disabled
- There may be general stability and complexity issues

Tips on making calls with WebRTC for Android:

- DTLS can be disabled for Chrome and Opera
- The default resolution used WebRTC is 640 x 480, which may be too complex for certain mobile devices. Therefore, if you notice a low frame rate or a loaded CPU, use a lower resolution, such as 320 x 240.