Dialogic® PowerMedia™ XMS WebRTC JavaScript API

User's Guide

March 2016 05-2715-007

www.dialogic.com

Copyright and Legal Notice

Table of Contents

Revision History

Revision	Release Date	Notes
05-2715-007 (Updated)	March 2016	API Specification: Removed the "Send Message (offline)" section.
05-2715-007	March 2016	Updates to support PowerMedia XMS Release 3.1.
05-2715-006	September 2015	Updates to support PowerMedia XMS Release 3.0. Demos: Added more information.
05-2715-005	February 2015	Updates to support PowerMedia XMS Release 2.4. Callback Specification: Added onInfoAck callback method.
05-2715-004 (Updated)	January 2015	Updates to remove VXML support.
05-2715-004	June 2014	Updates to support PowerMedia XMS Release 2.3.
05-2715-003	May 2014	Global Change: Updated URL format to HTTP and revised link locations. API Specification: Added Set Media Options method. Callback Specification: Updated description and added parameter to onUserMediaOk.
05-2715-002	February 2014	Updates to support PowerMedia XMS Release 2.2. Register: Added userPassword parameter. Appendix: Working Example of WebRTC Demo: Updated instructions, HTML, and JavaScript for "Simple Demo".
05-2715-001	October 2013	Updates to support PowerMedia XMS Release 2.1.
05-2715-001-01	August 2013	Initial release of this document.
Last modified: March 2016		

Refer to www.dialogic.com for product updates and for information about support policies, warranty information, and service offerings.

1. Welcome

The Dialogic® PowerMedia™ Extended Media Server (also referred to herein as "PowerMedia XMS" or "XMS") WebRTC JavaScript API provides functionality to support connecting to and performing media operations on Web Real-Time Communication (WebRTC) compliant devices or endpoints (for example, browsers).

This User's Guide provides specifications for using the Dialogic® PowerMedia™ XMS WebRTC JavaScript API (also referred to herein as "WebRTC JavaScript API").

Glossary

The following table describes the meaning of the terms as they are used in this document.

Term	Description
WebRTC JavaScript API	The provided WebRTC functionality, as encapsulated by the "webrtc.js" script.
Application	Customer specific web page, designed to use the WebRTC JavaScript API with PowerMedia XMS.
Web Page	Same as "Application" description above.
WebRTC Signaling Endpoint	The WebRTC Signaling port on the PowerMedia XMS.
Controller Application	The Application running off the Application Server, which controls the call flows of the WebRTC calls. Typically uses the MSML or RESTful interface to connect to the PowerMedia XMS.

2. Overview

The WebRTC JavaScript API enables web application developers a simple JavaScript API to utilize WebRTC browsers together with PowerMedia XMS for media processing operations.

This guide covers the WebRTC signaling and the specification of the WebRTC JavaScript API library for enabling media services with WebRTC communications in a browser.

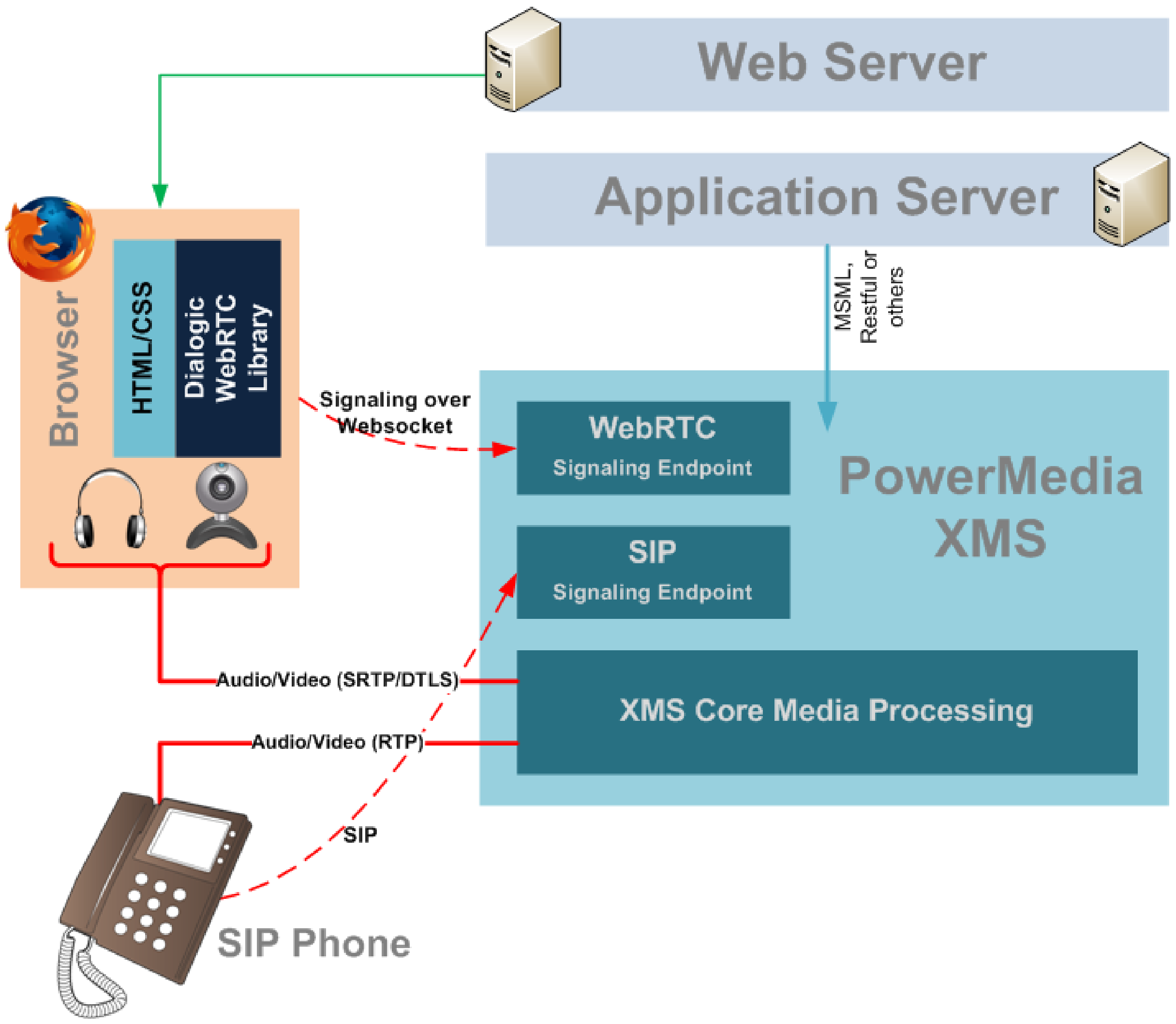
The WebRTC JavaScript API library is intended for Web Applications developed using JavaScript. It can be included as part of the rendered web page delivered to the client browser.

The WebRTC JavaScript API library handles signaling between the client and PowerMedia XMS via normal WebSocket connections. This model is commonly referred to as 1st party call control (1PCC), where signaling is handled transparently. The Application provides SDP as defined in RFC 4566, which it generates via the PeerConnect W3C API.

The WebRTC JavaScript API's are called with the SDP and a connection is established. The client browser must first create a registration binding with the PowerMedia XMS prior to initiating a call. This is accomplished through API calls. After registration is established, it can then make inbound calls to the PowerMedia XMS to perform media operations (for example, Play announcements) or connect to other registered entities.

Deployment Overview

At the heart of the WebRTC communication, is the PowerMedia XMS, which enables and manages the signaling as well as the media for the various call legs.



The Web Server hosts the Web Application, which embodies the front end of the WebRTC Application. As an example, this could be a retail website (like Amazon) which also provides users (of the website) with the ability to talk to their service representatives over an audio-visual channel in real time (for example, over WebRTC).

The Web Application in turn, builds its WebRTC communication using the WebRTC JavaScript API library, which allows rapid development of such applications.

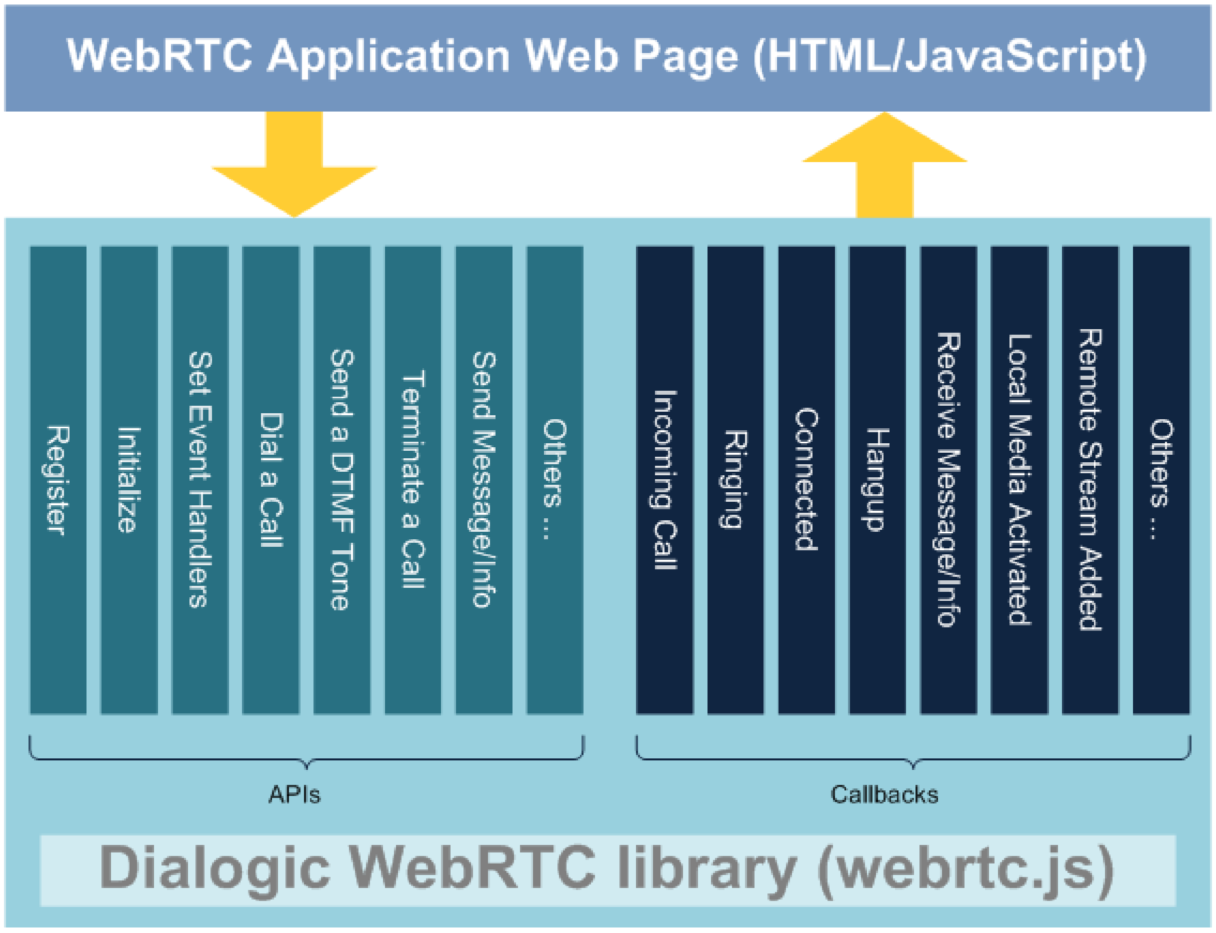
The Application Server hosts the Controller Application with the business logic to act upon communication requests, such as incoming WebRTC call requests. Typically, such applications would communicate to the PowerMedia XMS via protocols like MSML or RESTful.

WebRTC JavaScript API Overview

The WebRTC JavaScript API library is a complete and self-contained JavaScript library to enable WebRTC audio/video calls, DTMF and text chatting functionality. The functionality of this library is contained in the stand alone JavaScript - **webrtc.js**.

The WebRTC JavaScript API library is specifically designed to be easy to use. The API set contains a set of methods and callback hooks. The minimum set of APIs to get an Application up and running quickly is small. The rest of the (non-mandatory) APIs and callbacks can be implemented incrementally to refine the business logic or the user experience further. Note that the WebRTC JavaScript API library provides a default implementation for all optional callbacks, which if implemented, are overridden by the Application's method.

The library also manages the call state transitions internally, allowing the Application developer to exclusively focus on the Application's business logic and user experience.



To use the WebRTC JavaScript API library, the Application must include the **webrtc.js** in its HTML/JavaScript code, just like any other JavaScript. While the exact location of the **webrtc.js** may change depending upon the specific setup, the typical URL would be the following:

http://<xms_ip_address>/rtcweb/js/webrtc.js

The WebRTC JavaScript API library exposes its functionality through a set of APIs and callbacks. Most of the WebRTC JavaScript API's are asynchronous – therefore, the completion of the task performed is done using callbacks. This allows a purely event driven model to achieved within the Application.

The WebRTC JavaScript API library invokes the callbacks as a result of the events triggered by the user action or other entities (like a remote party trying to connect, network error, etc). The implementation of the callbacks must be supplied by the Application.

Demos

Simple Demo

The simple demo is a basic example of the HTML/JavaScript code for making WebRTC calls using the JavaScript API. It is ideal people who are unfamiliar with the JavaScript API.

The simple demo is part of the PowerMedia XMS demo repository on Github. Navigate to https://github.com/Dialogic/dialogic-html-javascript and follow the instructions in the *Readme.md* file to install and run the demo.

Refer to the *Dialogic® PowerMedia™ XMS WebRTC Demo Guide* for a list of supported browsers and how to access the browsers using HTTPS.

Verification Demo

PowerMedia XMS ships with a built-in verification demo. The verification demo's front end is more advanced than the simple demo's front end. It allows a WebRTC client to call into PowerMedia XMS to play audio and video, participate in a conference, or connect to a SIP or another WebRTC caller. Refer to the *Dialogic® PowerMedia™ XMS WebRTC Demo Guide* for more information.

The verification demo requires no installation. A WebRTC capable browser must be used to call into the verification demo. Refer to the *Dialogic® PowerMedia™ XMS WebRTC Demo Guide* for a list of supported browsers.

In the address field, enter the following URL into a WebRTC compatible browser to run the demo. The URL should be in HTTPS format:

https://<xms_ip_address>/rtcweb/webrtc.html

The source code for the demo can be found at:

<xms_ip_address>:/var/www/rtcweb/html/webrtc.html

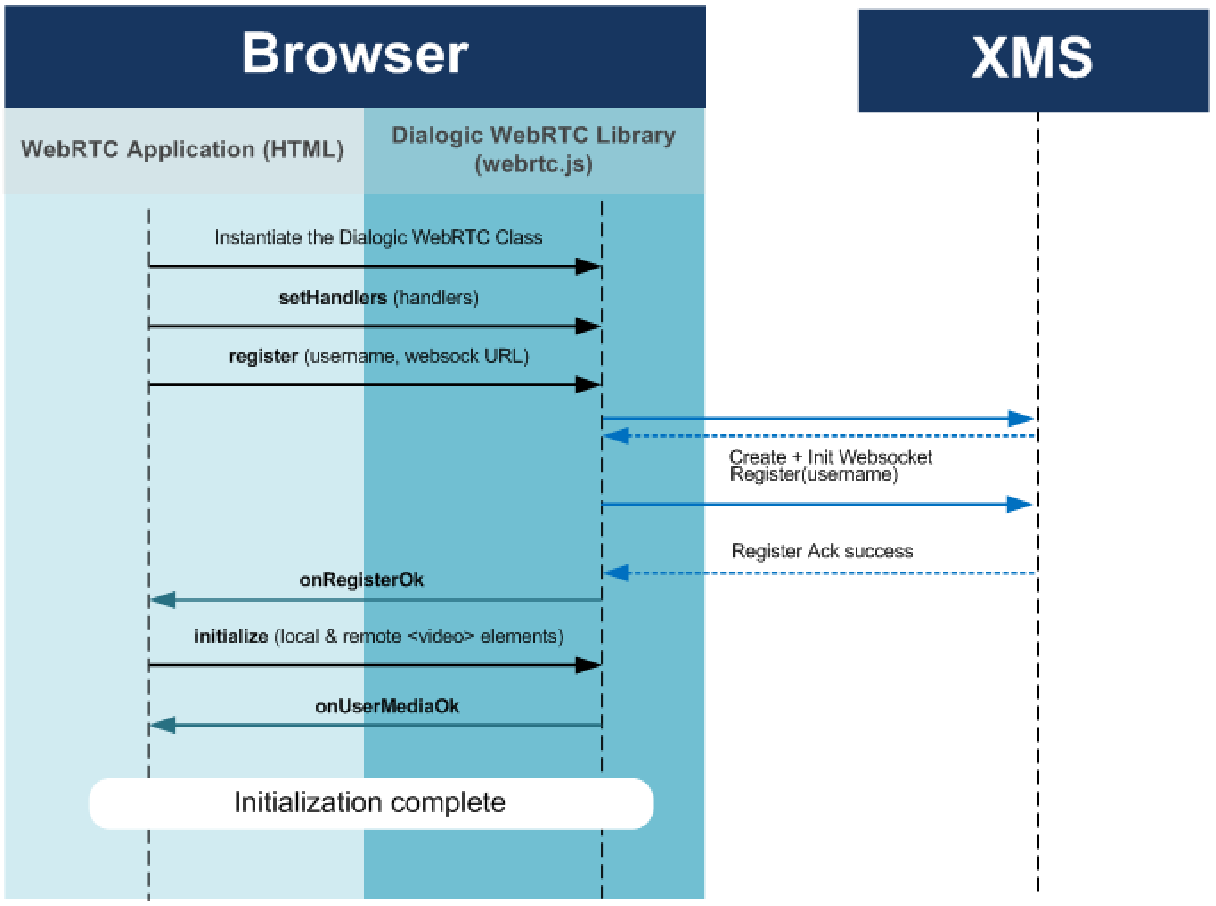
<xms_ip_address>:/var/www/rtcweb/html/js/global.js

3. API Usage Guide

Guidelines

The following provides a basic code flow of the WebRTC JavaScript API library.

Initialization


Browser
XMS
WebRTC Application (HTML)
Dialogic WebRTC Library (webrtc.js)
Instantiate the Dialogic WebRTC Class
setHandlers (handlers)
register (username, websock URL)
Create + Init Websocket
Register(username)
Register Ack success
onRegisterOk
initialize (local & remote <video> elements)
onUserMediaOk
Initialization complete

The above diagram shows the API flow for the initialization of the WebRTC JavaScript API library. The rest of the section details the code snippets corresponding to the flow.

If you are not familiar with web programming or would like to get a demo up and running quickly, refer to the Simple Demo for an example of a simple outbound WebRTC call. The example includes both HTML and JavaScript code.

Note: This is similar to "webrtc.html" demo but simplified as much as possible to illustrate programming concepts.

Instantiate the WebRTC JavaScript API Class

```
    DialogicDemo = new Dialogic ();
```

There's now a WebRTC JavaScript API instance. All further operations are performed on this object.

Set Event Handlers

```
      var userHandlers = {
            'onRegisterOk':          registerSuccess,
            'onRegisterFail':        registerFail,
            'onRinging':             ringingHandler,
            'onConnected':           null,
            'onInCall':              incomingCallHandler,
            'onHangup':              callHangupHandler,
            'onDisconnect':          disconnectHandler,
            'onUserMediaOk':         userMediaSuccessHandler,
            'onUserMediaFail':       userMediaFailHandler,
            'onRemoteStreamOk':      remoteStreamAddedHandler,
            'onMessage':             messageHandler,
            'onInfo':                infoHandler,
            'onDeregister':          null,
            'onInfoAck':             infoAckHandler
      };

      DialogicDemo.setHandlers( userHandlers );
```

All the event handlers are registered to the WebRTC JavaScript API library. They will be fired when the relevant event occurs. Note that all event handlers are not mandatory, and in this example, the "onConnected" handler has not been assigned (set to "null"). See the section Call States and Callbacks for a quick overview on events and their associated callbacks.

Register

```
    DialogicDemo.register ("John", "ws://10.40.2.33:1080");
```

This call registers this Application using the name "John" to the specified WebRTC Signaling Endpoint. Once the callback "registerSuccess" fires, move to the next step (set in Set Event Handlers).

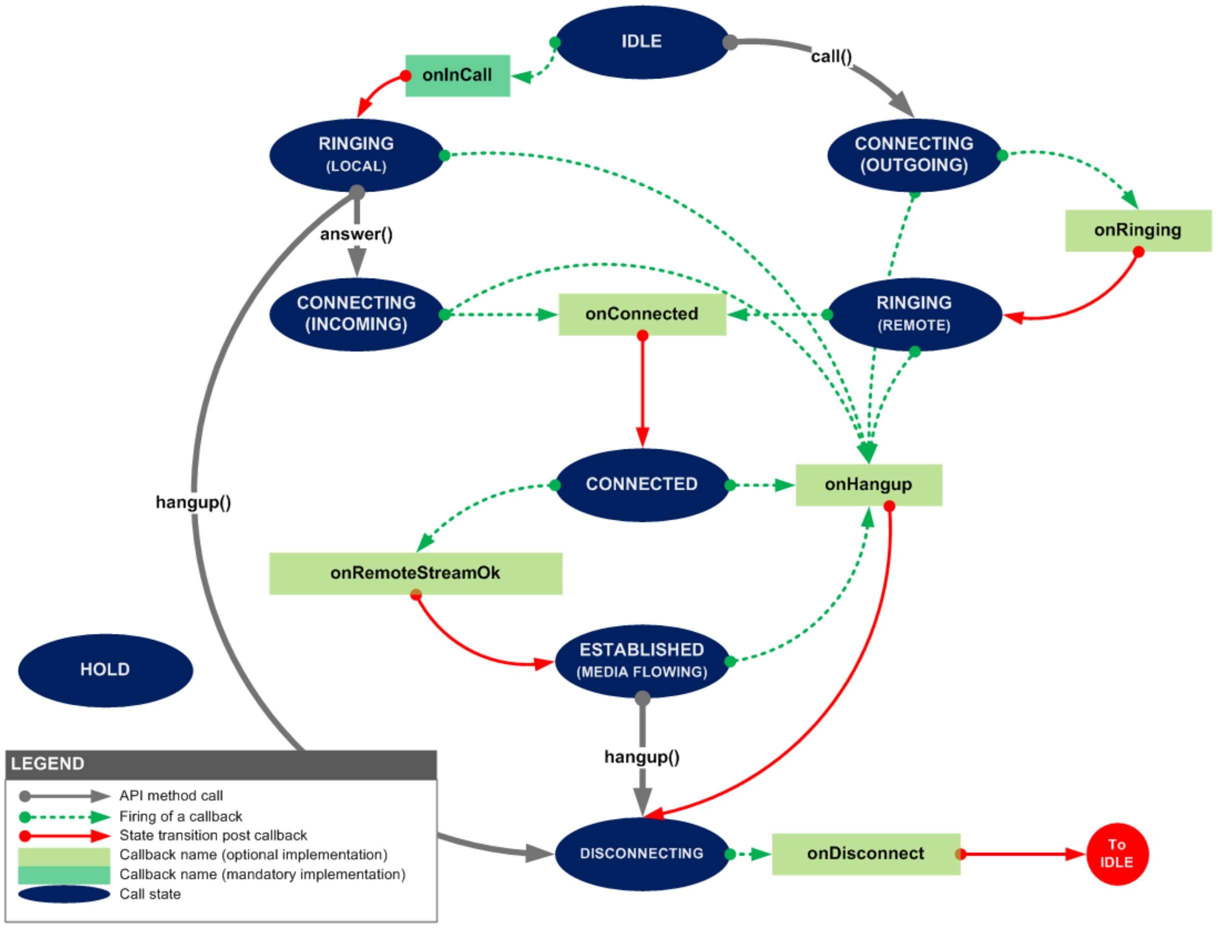
Initialize the WebRTC JavaScript API and Acquire the Local Media Method

```
    /* Initialize the WebRTC library */

    var lVideo = document.getElementById("localVideo");
    var rVideo = document.getElementById("remoteVideo");

    var spec = {
        'localVideo'  : lVideo,
        'remoteVideo' : rVideo,
        'remoteAudio' : null,
        'ipv6'        : false
    };

    /* Now invoke the initialize API with HTML video elements*/

    ret = DialogicDemo.initialize(spec);

    /*
     * Now acquire the media
     */

    var mediaConstraints = {
        'audio': true,
        'video': true
    };

    ret = DialogicDemo.acquireLocalMedia(mediaConstraints);
```

This step binds the video streams to the corresponding HTML <video> elements. Now, the WebRTC JavaScript API library will display the video streams to the local and remote <video> elements respectively. Additionally, the local media assets (like a microphone, camera, etc) are acquired. Once done successfully, the callback onUserMediaOk (set in Set Event Handlers) fires. It's now ready to make or receive a WebRTC Call.

Call States and Callbacks

The WebRTC JavaScript API library maintains the call states internally and fires various callbacks to intimate the state of progress of a call. The following diagram shows the internal call states and their relationships to the various callbacks.

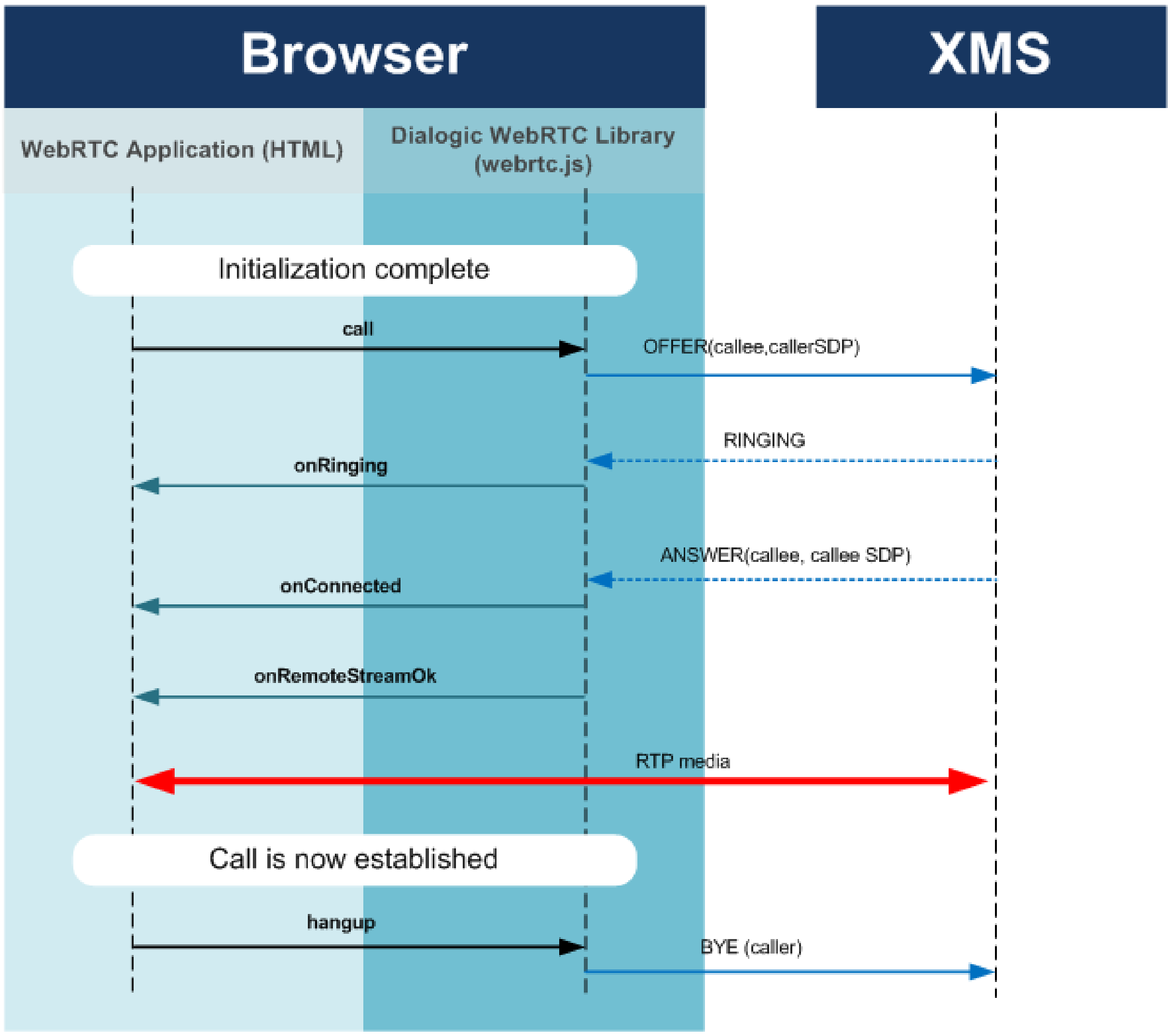


After initialization of the WebRTC JavaScript API library (see Initialization), the library is ready to make or receive a call – it is in the "IDLE" state. Once a call is disconnected, the library comes back to the "IDLE" state.

The library maintains the internal states and transits through them based on user actions (like "call" or "hangup") or events (like "incoming call"). A callback fired by the library signifies an event.

Note that all callback hooks, except for the "onInCall" hook, are optional. For example, the Application does not have to implement them. The "onInCall" hook is mandatory, since it would be impossible to act on an incoming call if this hook is not handled.

Dial a Call



Dialing an outgoing call using the WebRTC JavaScript API library involves a sequence of steps as shown in the API flow in the diagram above.

The following shows the code snippets to achieve the same.

Dial an Outgoing Call

```
// Make an outgoing call

var ret = DialogicDemo.call("Doe", "video");

if (ret != "ok") {
  // ... ( cleanup code ) ...
}
else {
  // The call was successful.
}
```

Once the "call" returns successfully, wait for the callback which is registered as the onRinging handler to fire.

Wait for Ringing

```
function ringingHandler()
{
   /** Application logic **/

   // Write some code here to play a ringing tone to the user
   // to simulate remote party ringing. Possibly other visual
   // feedbacks could also be implemented here.
   //
   // The ringing state continues until the callback onConnected
   // is fired.

   return;


}
```

The onRinging callback fires – it implies that the remote party is ringing.

Wait for Call Acceptance

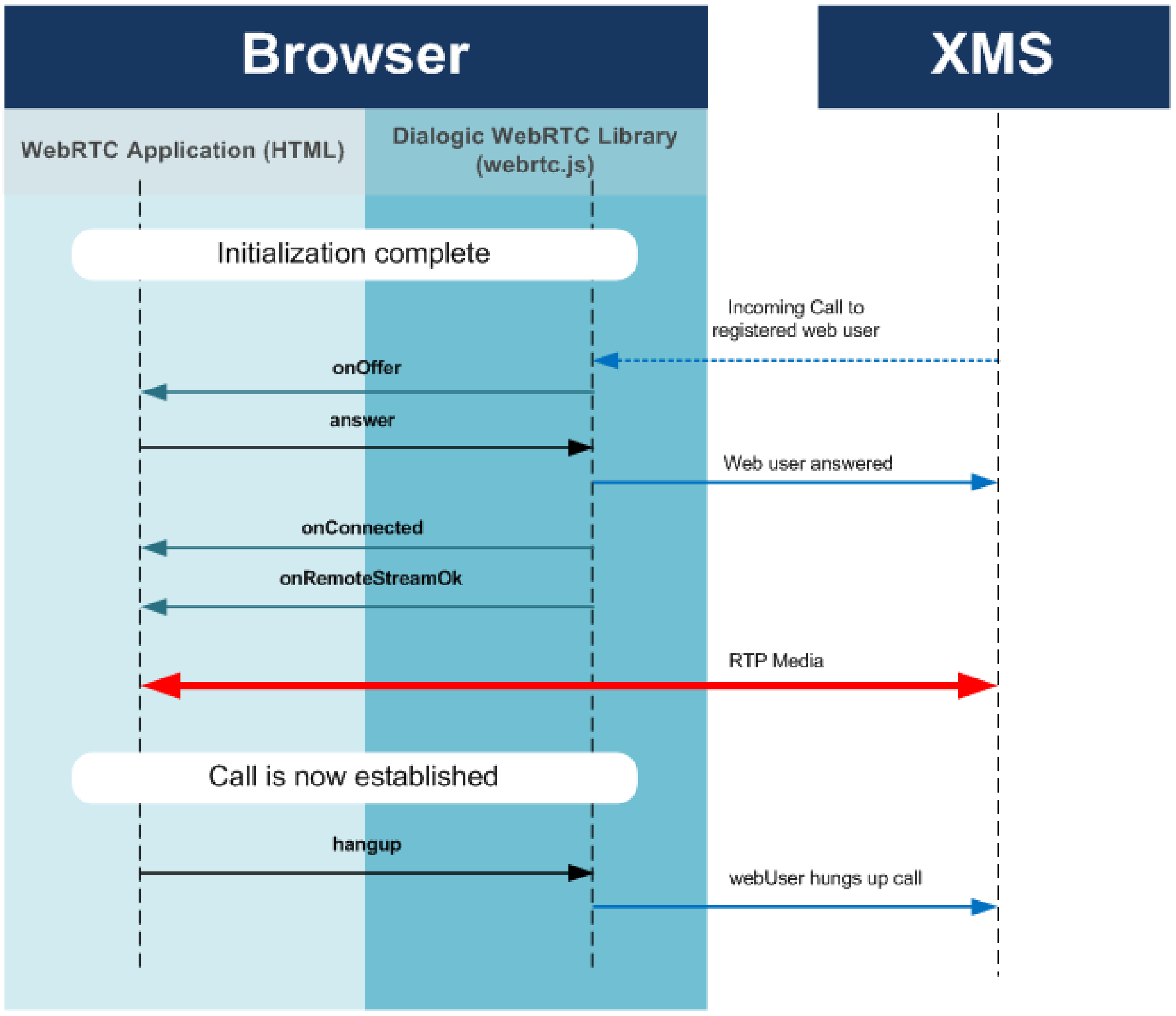
In the example, a "null" is assigned to the "onConnected" handler (see Set Event Handlers). The user will not know when the call actually gets connected. Wait for the onRemoteStreamOk callback to fire.

Wait for Remote Media Stream

```
function remoteStreamAddedHandler()
{
   /** Application logic **/

   // The call is now fully established.
   // Change or highlight some icon to indicate remote media activated

   return;
}
```

The call is now fully established. Use hangup to terminate the call. If the remote party hangs up the call, then the onHangup callback will fire.

Receive a Call



Receiving an incoming call using the WebRTC JavaScript API library involves a sequence of steps as shown in the API flow in the diagram above.

The following shows the code snippets to achieve the same.

Receive the onInCall Callback

```
function incomingCallHandler(name)
{

   /** Application logic **/

   // Write some code here to display a popup to the user
   // and informing the user of an incoming call and wait for
   // the user input.
   //
   // The "name" parameter can be used to inform ‘who’ called.
   //
   // Once the user decides (either to accept or reject the call)
   // prepare to correctly inform the WebRTC library of the decision
   // by returning the correct string as below.


   /* If the User want to accept the call ... */

          answer ();

   /* Else */

          hangup ();
   }
}
```

The onInCall callback is fired on the event of an incoming call. The user or Application has the option to either accept or reject the call. If the call is accepted, then wait for the onRemoteStreamOk callback to fire.

Wait for Remote Media Stream

```
function remoteStreamAddedHandler()
{
   /** Application logic **/

   // The call is now fully established.
   // Change or highlight some icon to indicate remote media activated

   return;
}
```

The call is now fully established. Use hangup to terminate the call. If the remote party hangs up the call, then the onHangup callback will fire.

4. API Specification

Instantiate

Prototype

```
Dialogic ();
```

Description

The JavaScript class Dialogic encapsulates the WebRTC JavaScript API functionality. It is mandatory to instantiate this class, since all the subsequent operations are performed on this instance.

Parameters

None

Usage

The Dialogic class is instantiated in a standard way by calling the JavaScript 'new' method on the class, as shown in the example code below.

Example Code

```
// Instantiate the Dialogic WebRTC Object

DialogicDemo = new Dialogic ();

// Now the object "DialogicDemo" can be used to invoke the functionality of
// the Dialogic WebRTC library
```

Set Event Handlers

Prototype

```
setHandlers ( handlers_list );
```

Description

This method sets all the required event handlers in the WebRTC JavaScript API library. The handlers are implementation defined and are fired on appropriate events. Other than the "onRegisterOk" and the "onInCall" handlers, all others are optional.

Parameters

The parameter to this method is a JavaScript associative array. Each key is a hook to an implementation defined function. The keys can take the following values:

Key	Mandatory	Description
'onRegisterOk'	Yes	Fires when registration succeeds. See onRegisterOk.
'onRegisterFail'	No	Fires when registration fails. See onRegisterFail.
'onRinging'	No	Fires on remote end ringing. See onRinging.

Key	Mandatory	Description
'onRegisterOk'	Yes	Fires when registration succeeds. See onRegisterOk.
'onRegisterFail'	No	Fires when registration fails. See onRegisterFail.
'onRinging'	No	Fires on remote end ringing. See onRinging.
'onConnected'	No	Fires when call gets connected. See onConnected.
'onInCall'	Yes	Fires on an incoming call event. See onInCall.
'onHangup'	No	Fires when the call hangs up. See onHangup.
'onDisconnect'	No	Fires when a call completely disconnects and frees all resources. See onDisconnect.
'onUserMediaOk'	No	Fires when local media initialization succeeds. See onUserMediaOk.
'onUserMediaFail'	No	Fires when local media initialization fails. See onUserMediaFail.
'onRemoteStreamOk'	No	Fires when the remote stream starts. See onRemoteStreamOk.
'onMessage'	No	Fires when a message is received. See onMessage.
'onInfo'	No	Fires when info is received. See onInfo.
'onDeregister'	No	Fires when the user is de-registered. See onDeregister.
'onInfoAck'	No	Fires when acknowledgement to an info message is received. See onInfoAck.

Usage

This method provides and sets hooks for callbacks. The associative array's keys are the 'hooks' to which the callback references (or names) are assigned. The implementation of the callbacks is Application specific, but by setting their references by this method, the Application can ensure that they are called at the appropriate events.

All hooks, except "onRegisterOk" and "onInCall", are optional and can be set to "null".

Example Code

```
      /* Set the references to the user defined callbacks */

      var userHandlers = {
            'onRegisterOk': registerSuccess,
            'onRegisterFail': registerFail,
            'onRinging': ringingHandler,
            'onConnected': null,
            'onInCall': incomingCallHandler,
            'onHangup': callHangupHandler,
            'onDisconnect': disconnectHandler,
            'onUserMediaOk': userMediaSuccessHandler,
            'onUserMediaFail': userMediaFailHandler,
            'onRemoteStreamOk': remoteStreamAddedHandler,
            'onMessage': messageHandler,
            'onInfo': infoHandler,
            'onDeregister': deRegisterHandler,
            'onInfoAck': infoAckHandler
    };

    /* Now inform the WebRTC library */
    DialogicDemo.setHandlers( userHandlers );
```

Return

Key	Type	Description
"ok"	String	Success.
"error_bad_args"	String	Either the mandatory hooks were not set, or some other improper arguments were supplied.

Register

Prototype

```
register (
        userName,
        signalingServerURL,
        userPassword
        );
```

Description

This method registers the provided user name to the WebRTC Signaling Endpoint which resides on the PowerMedia XMS (see Deployment Overview).

Parameters

Parameter	Type	Mandatory	Description
userName	String	Yes	Name to be registered with WebRTC Signaling Endpoint (on the PowerMedia XMS). Other users will be able to connect to the current user using the name specified here.
signalingServerURL	String	Yes	A WebSocket URL pointing to the WebRTC Signaling Endpoint. URL types supported are: • "ws" for normal WebSockets A "ws" URL may look like this, for example: "ws://10.40.2.33:1080" In the above example, "10.40.2.33" is the IP address of the PowerMedia XMS installation and "1080" is the TCP port configured as the WebRTC Signaling Endpoint on it.

Parameter	Type	Mandatory	Description
userName	String	Yes	Name to be registered with WebRTC Signaling Endpoint (on the PowerMedia XMS). Other users will be able to connect to the current user using the name specified here.
signalingServerURL	String	Yes	A WebSocket URL pointing to the WebRTC Signaling Endpoint. URL types supported are: • "ws" for normal WebSockets A "ws" URL may look like this, for example: "ws://10.40.2.33:1080" In the above example, "10.40.2.33" is the IP address of the PowerMedia XMS installation and "1080" is the TCP port configured as the WebRTC Signaling Endpoint on it.
userPassword	String	No	The password associated with the userName parameter.

Usage

This is the first method to be called, after instantiation of the WebRTC JavaScript API class. This method performs the first handshake with the WebRTC Signaling Endpoint. The success of this method is crucial to working of the WebRTC as this method initiates the communication with the WebRTC Signaling Endpoint and registers the current user to it.

The callbacks onRegisterOk and onRegisterFail are called in the case of success and failure of registration respectively.

A typical implementation could choose to move to the next step (see Initialize) in the callback onRegisterOk directly. If however, the onRegisterFail is fired, it is a critical situation since it is not possible to move ahead unless this error is fixed.

Typical reasons for "register" to fail could be:

- Lack of network connectivity
- Signaling Endpoint not reachable
- Controller Application specific error
- Other internal error

Fixing this error will likely require user intervention, so it is recommended that an implementation of the onRegisterFail should display an informative error message to the user, enabling the user to take corrective action.

Example Code

```
// Register the user "John" to the Dialogic WebRTC Endpoint. The callback
// handlers "registerSuccess" and "registerFail" are assumed to
// have been defined previously by using the method 'setHandlers'

DialogicDemo.register ( "John", "ws://10.40.2.33:1080" );

// Now wait for one of the above two callbacks to fire …
```

Return

Key	Type	Description
"ok"	String	Success.
"error_bad_args"	String	Either the mandatory hooks were not set, or some other improper arguments were supplied.

Acquire Local Media

Prototype

```
acquireLocalMedia ( mediaConstraints );
```

Description

This method attempts to acquire the microphone and camera attached to the machine, exclusively for its use. The method fails, if these resources are in use by some other application or even another instance of a running WebRTC session.

Parameters

The parameter "mediaConstraints" is a JavaScript associative array with keys as listed in the table below. Also, see the example code section below.

Key	Value	Description
'audio'	true/false	Specifies whether the audio component must be initialized.
'video'	true/false	Specifies whether the video component must be initialized.

Usage

This method must be called only *after* the successful completion of the Register method. A typical implementation could be, to call this method directly in the onRegisterOk callback.

This method is asynchronous, and informs its completion status through callbacks. If the initialization is successful, the callback onUserMediaOk is fired. Else, on error, the callback onUserMediaFail is invoked.

Example Code

```
// Once the registration is successful, we can go ahead and try to
// acquire the media.

function registerSuccess()
{
    var mediaConstraints = {
        'audio': true,
        'video': true
    };

    ret = DialogicDemo.acquireLocalMedia(mediaConstraints);

    if (ret != "ok") {
        /* Handle errors … */
    }
    else {
        /* Success */
    }
}
```

Return

Key	Type	Description
"ok"	String	Success.
"error_bad_args"	String	The supplied arguments could not be correctly interpreted.
"error_not_permitted"	String	The method was called in an invalid state or context.

Initialize

Prototype

```
initialize ( spec );
```

Description

This method associates the user HTML media objects with the actual media objects in the WebRTC library, and enables it to render the media into the desired HTML tags. The parameter "spec" is a JavaScript associative array object.

Parameters

The parameter "spec" is a JavaScript associative array with the keys as listed in the table below.

Key	Value	Description
'localVideo'	String	Specifies whether the HTML5 <video> tag into which the local video must be rendered
'remoteVideo'	String	Specifies whether the HTML5 <video> tag into which the remote video must be rendered
'remoteAudio'	String	Specifies whether the HTML5 <audio> tag into which the remote audio must be rendered
'ipv6'	String	Enable ipv6 ice candidates on client side. If browser has support for ipv6 candidate discovery, it would send ipv6 candidates to peer WebRTC endpoint.

Usage

This method must be called only *after* the successful completion of the Register method. A typical implementation could be to call this method directly in the onRegisterOk callback.

This method is synchronous, and its completion status is immediately informed by its return value.

Example Code

```
// Once the registration is successful, we can go ahead and initialize
// the Dialogic WebRTC library. The typical way to do it could be right
// inside the registration success callback.
//

function registerSuccess()
{
    /* Get the <video> elements from the DOM */

    var lVideo = document.getElementById("localVideo");
    var rVideo = document.getElementById("remoteVideo");

    var spec = {
        'localVideo'  : lVideo,
        'remoteVideo' : rVideo,
        'remoteAudio' : null,
        'ipv6'        : false
    };

    /* Now invoke the initialize API with HTML video elements*/

    ret = DialogicDemo.initialize(spec);

    if (ret != "ok") {
        /* Handle errors … */
    }
    else {
        /* Success */
    }
}
```

Return

Key	Type	Description
"ok"	String	Success.
"error_user_unregistered"	String	The local user is not registered with the WebRTC Signaling Endpoint.
"error_bad_args"	String	The supplied parameters were incorrect.
"error_not_permitted"	String	The method was called in an invalid state or context.

Additional Notes

The following shows an example HTML showing a video element. Note that this HTML code snippet shows the "remoteVideo" only.

```
<div class="windowBorder" style="max-width:640px;">
  <div class="innerWindow" id="videocontainer" >
    <span id="remotelive" class="live hide">LIVE</span>
    <div id="remoteVideoContainer">
      <video id="remoteVideo" controls="true"
             autoplay="autoplay" align="left"
             style="opacity: 0;
             -webkit-transition-property: opacity;
             -webkit-transition-duration: 2s;">
      </video>
    </div>
  </div>
</div>
```

Set Media Options

Prototype

```
setMediaOptions ( mediaOptions );
```

Description

This method sets media options of local video that is offered to remote side.

Parameters

The parameter "mediaOptions" is a JavaScript associative array with the keys as listed in the table below.

Key	Value	Description
'videoWidth'	Number	Video width of local video that would be offered to remote side. Default is 640.
'videoHeight'	Number	Video height of local video that would be offered to remote side. Default is 480.
'videoFrameRate'	Number	Local Video frame rate that would be offered to remote side. Default is 30.

Usage

This method must be called only *after* the successful completion of the onUserMediaOk method. A typical implementation could be to call this method directly in the onUserMediaOk callback.

This method is synchronous, and its completion status is immediately informed by its return value.

Example Code

```
// Once access to local video is successful, , we can set the media options.

//

function userMediaSuccessHandler ( localMediaType )
{

     if (localMediaType == "audiovideo" ) {
      var mediaOpt = { 'videoWidth': 640, 'videoHeight': 480,
      'videoFrameRate': 30 };
      ret = DialogicDemo.setMediaOptions(mediaOpt);
      if ( ret === 'ok' ) {
            console.log("setMediaOptions success");
      } else {
            console.log("setMediaOptions fail");
      }
     }
}
```

Return

Key	Type	Description
"ok"	String	Success.
"error_bad_args"	String	The supplied parameters were incorrect.

Dial a Call

Prototype

```
String call( calledPartyIdentifier, callType,
mime_type, text_str );
```

Description

This method initiates a call to the remote party, identified by the parameter "calledPartyIdentifier". The parameter "callType" specifies whether the call is audio only or audio/video.

Parameters

Parameter	Type	Mandatory	Description
calledPartyIdentifier	String	Yes	The identifier for the called party. The format of the identifier typically depends on the implementation of the Controller Application.
callType	String	Yes	The identifier for the called party. The format of the identifier typically depends on the implementation of the Controller Application. The type of call. Possible values are: • "audio" for audio only calls • "video" for audio/video calls
mime_type	String	No	Specifies optional mime-type that can be communicated to server side application.
text_str	String	No	Specifies optional text string that can be communicated to server side application.

Usage

This method can only be called once the WebRTC JavaScript API library has been successfully initialized (see Initialize). The invocation of this method starts the WebRTC Call dialing sequence (see Dial a Call and Call States and Callbacks).

This method will return an error if the caller is not already registered (see Register); else, the method starts off the Call dialing sequence.

If the WebRTC stack was initialized with only the "audio" component and this method is invoked with a "video" callType, then the behavior is undefined.

The progress of the call is informed to the user via the registered callbacks:

- onRinging

  The WebRTC JavaScript API library calls this (implementation defined) callback to indicate that the called party is ringing, but has not yet accepted the call.

- onConnected

  The WebRTC JavaScript API library calls this (implementation defined) callback to indicate that the called party has accepted the call and the signaling channel is established. Note that the call is still not fully established until the onRemoteStreamOk fires. If the remote party, instead, rejected the call, then the callback onHangup will fire.

- onRemoteStreamOk

  The WebRTC JavaScript API library calls this (implementation defined) callback to indicate that the media from the called party has started flowing in. The call is now fully established.

- onHangup

  This callback fires if the call gets disconnected for any reason, including a rejection from the remote party. This callback can fire at any stage during the call (see Call States and Callbacks).

- onDisconnect

  This callback fires when the call is completely disconnected. The disconnect request can be initiated by either party. The state of the WebRTC JavaScript API library after the firing of this callback is restored to "IDLE" and the library is ready to make or receive a new call (see Call States and Callbacks).

Example Code

```
// Make an outgoing call

var ret = DialogicDemo.call("Doe", "video");

if (ret != "ok") {
//
// Handle individual error codes if required.
// Since the call was not successful, cleanup and get out ...
//
//       ... ( cleanup code ) ...
//
}
else {
//
// The call was successful. We will be informed about further progress
// through the callbacks ...
//
}
```

Return

Key	Type	Description
"ok"	String	Success.
"error_user_unregistered"	String	The local user is not registered with the WebRTC Signaling Endpoint.
"error_bad_args"	String	Either the mandatory hooks were not set, or some other improper arguments were supplied.
"error_not_permitted"	String	The method was called in an invalid state or context.

Additional Notes

Any actions initiated due to this method will terminate on browser tab/page refresh or close.

Receive a Call

Description

The handling of an incoming call is done though the (implementation defined) callback referenced by onInCall. See Receive a Call for an overview of the API flow.

Accept an Incoming Call

Prototype

```
answer( mime_type, text_str );
```

Description

This method accepts an incoming call.

Parameters

Parameter	Type	Mandatory	Description
mime_type	String	No	Specifies optional mime-type that can be communicated to server side application.
text_str	String	No	Specifies optional text string that can be communicated to server side application.

Usage

This method can only be called once the WebRTC JavaScript API library has been successfully initialized (see Initialize) and must be called only in the state of an incoming call, which is indicated by the firing of the onInCall hook. The invocation of this method starts the WebRTC Call Acceptance sequence (see Dial a Call and Call States and Callbacks).

This method will return an error if the caller is not already registered (see Register) or if there is no incoming call context; else, the method starts off the Call Acceptance sequence.

The progress of the call is informed to the user via the registered callbacks:

- onConnected

  The WebRTC JavaScript API library calls this (implementation defined) callback to indicate that the signaling channel is established. Note that the call is still not fully established until the onRemoteStreamOk fires.

- onRemoteStreamOk

  The WebRTC JavaScript API library calls this (implementation defined) callback to indicate that the media from the called party has started flowing in. The call is now fully established.

- onHangup

  This callback fires if the call gets disconnected for any reason, including a rejection from the remote party. This callback can fire at any stage during the call (see Call States and Callbacks).

- onDisconnect

  This callback fires when the call is completely disconnected. The disconnect request can be initiated by either party. The state of the WebRTC JavaScript API library after the firing of this callback is restored to "IDLE" and the library is ready to make or receive a new call (see Call States and Callbacks).

Example Code

```
// In the event of an incoming call "onInCall ()", accept the call ...

DialogicDemo.answer ();
```

Return

Key	Type	Description
"ok"	String	Success.
"error_user_unregistered"	String	The local user is not registered with the WebRTC Signaling Endpoint.
"error_not_permitted"	String	The method was called in an invalid state or context.

Send DTMF Tones

Prototype

```
sendDTMF(dtmfDigit, [dtmfDuration]);
```

Description

This method sends a DTMF tone or digit to the remote party. The DTMF digits can be sent only when the call is in an established state. This method makes a best effort at transmitting the DTMF digit, but does *not* guarantee it's delivery to the remote party.

Parameters

Parameter	Type	Mandatory	Description
dtmfDigit	String	Yes	Specifies the DTMF digit to be transmitted.
dtmfDuration	Integer	No	Specifies the DTMF duration in milliseconds. The current implementation allows a maximum value of 100 milliseconds. If not set, this parameter defaults to 100 milliseconds.

Usage

This method can be invoked only if the call is. Only one digit can be sent per invocation of this method. The method makes a best effort attempt at transmitting the DTMF digit, but it’s delivery to the remote end is not guaranteed.

Example Code

```
      // Send out a DTMF digit "1"

      ret = DialogicDemo.sendDTMF("1");
      if (ret != "ok") {
            /* Handle errors … */
      }
      else {
            /* Success */
      }
```

Return

Key	Type	Description
"ok"	String	Success.
"error_user_unregistered"	String	The local user is not registered with the WebRTC Signaling Endpoint.
"error_bad_args"	String	Either the mandatory hooks were not set, or some other improper arguments were supplied.
"error_not_permitted"	String	The method was called in an invalid state or context.

Call Termination

Prototype

```
hangup( reason, mime_type, text_str );
```

Description

This method terminates an ongoing call or causes rejection of an incoming call. The remote party is informed of the termination.

Parameters

Parameter	Type	Mandatory	Description
reason	String	Yes	Specifies a reason (free flowing format). This string is communicated to the Controller Application.
mime_type	String	No	Specifies optional mime-type that can be communicated to the Controller Application.
text_str	String	No	Specifies optional text string that can be communicated to the Controller Application.

Usage

This method can be invoked anytime during the call life cycle. For an outgoing call, "hangup" can be invoked any time after the invocation of the "call". For an incoming call (notified by the onInCall hook), the invocation of "hangup" results in the rejection of the call (as opposed to accepting the call through the "answer" method). Upon the termination of the call, the (implementation defined) onDisconnect hook is fired.

Example Code

```
// Terminate an existing call

DialogicDemo.hangup( );
```

Return

Key	Type	Description
"ok"	String	Success.
"error_user_unregistered"	String	The local user is not registered with the WebRTC Signaling Endpoint.
"error_not_permitted"	String	The method was called in an invalid state or context.

Send Message (in call)

Prototype

```
sendInfo( mime_type, text_str );
```

Description

This method sends any arbitrary text string to the Controller Application *while the call is established*. The specific business function of this functionality depends on the Controller Application implementation and could be used for text based features like chatting or announcements.

Parameters

Parameter	Type	Mandatory	Description
mime_type	String	Yes	Specifies the mime-type of the message.
text_str	String	Yes	Specifies the text string that needs to be sent to the remote party.

Usage

This method must be called only if the call is in an established state. This method returns immediately after initiating the message transmission sequence. This method attempts a best effort delivery of the message and there is no callback to inform the success or failure of the delivery of this message to the remote party.

Example Code

```
// Send a text message to the Controller Application

DialogicDemo.sendInfo ("text/plain", "Hello there !");
```

Return

Key	Type	Description
"ok"	String	Success.
"error_user_unregistered"	String	The local user is not registered with the WebRTC Signaling Endpoint.
"error_bad_args"	String	Either the mandatory hooks were not set, or some other improper arguments were supplied.
"error_not_permitted"	String	The method was called in an invalid state or context.

5. Callback Specification

onRegisterOk

Prototype

```
onRegisterOk();
```

Description

This callback method indicates that the Register method call was successful. This method is implementation defined, but is invoked by the WebRTC JavaScript API library asynchronously, to indicate success in the registration process.

The implementation of this handler is mandatory.

Parameters

None

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onRegisterOk").

This function should implement logic for the next steps once the WebRTC registration succeeds. The implementation of this function could, for instance, initialize the WebRTC JavaScript API library by invoking the Initialize method.

Example Code

```
/**
 * User registration success handler. Invoke the initialize API
 **/

registerSuccess = function () {

  console.log("User Registration : Success");

  // Get the <video> elements for the Local & Remote videos

  userLocalVideo        = document.getElementById("localVideo");
  userRemoteVideo       = document.getElementById("remoteVideo");

  DialogicDemo.initialize(userLocalVideo, userRemoteVideo);
};

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onRegisterFail

Prototype

```
onRegisterFail(reason);
```

Description

This callback method indicates that an error has occurred in the execution of the Register method call. This method is implementation defined, but is invoked by the WebRTC JavaScript API library asynchronously, to indicate a failure in the registration process.

The implementation of this handler is optional.

Parameters

Parameter	Type	Description
reason	String	Indicates the reason for the registration failure. Currently the value always is "RegistrationFailed".

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onRegisterFail").

This function should implement the logic for the scenario where the WebRTC registration fails.

Example Code

```
/**
 * User registration failure handler.
 **/

registerFail = function ( reason ) {

  // Registration failed implies some error occurred and we can
  // not go ahead. Maybe the network failed ? Or some other error ?
  // In any case, the user should be informed and some corrective
  // action taken before we can go ahead …

  console.log("Registration filed. Reason : " + reason);
};

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onInCall

Prototype

```
String onInCall( fromName );
```

Description

This hook (or handler) fires if there is an incoming call event. It informs the Application of the name of the remote party which is attempting to call via the parameter "fromName".

The implementation of this handler is mandatory.

Parameters

Parameter	Type	Description
fromName	String	Indicates the identifier of the caller, and its format is Application dependent.

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onInCall").

The firing of this callback indicates an incoming call. A typical implementation could, upon receiving this callback, create a popup to inform the user of an incoming call event and providing the user with an option to accept or reject the call. Depending upon what the user chooses, either "answer" or "hangup" methods could be then invoked as the next step.

If the Application chooses to accept the call, the actual maturing of the call is informed via the hook onRemoteStreamOk.

Example Code

```
function incomingCallHandler (name)
{

   /** Application logic **/

   // Write some code here to display a popup to the user
   // and informing the user of an incoming call and wait for
   // the user input.
   //
   // The "name" parameter can be used to inform 'who' called.
   //
   // Once the user decides (either to accept or reject the call)
   // make the appropriate API call.


   if( callAccepted ) {

       /* User accepted the call ...
        *
        * call "answer ()"
        */

   } else {
       /* User rejected the call ...
        *
        * call "hangup ()"
        */
   }
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onRinging

Prototype

```
onRinging();
```

Description

This hook (or handler) fires if a call attempt to the remote party succeeds and the remote side is "ringing". This handler fires only in the case of an outgoing call.

The implementation of this handler is optional.

Parameters

None

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onRinging").

A typical implementation for this callback could play a ring tone on the user terminal to simulate remote party ringing. The "ringing" stage is cleared once the onConnected fires, which indicates that the remote party has accepted the call.

Example Code

```
function ringingHandler()
{
   /** Application logic **/

   // Write some code here to play a ringing tone to the user
   // to simulate remote party ringing. Possibly other visual
   // feedbacks could also be implemented here.
   //
   // The ringing state continues until the callback onConnected
   // is fired.

    return;

}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onConnected

Prototype

```
onConnected( );
```

Description

This hook fires when an incoming or an outgoing call gets connected. In the case of an outgoing call, this callback indicates that the remote user has accepted the call and call signaling path has been established. In the case of an incoming call, this will fire only if the local user accepted the call (see Answer a Call) and indicates that the call signaling path has been established.

The implementation of this handler is optional.

Parameters

None

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onConnected").

If the call is accepted, it implies that the signaling phase of the call is complete. However, the call is not fully established until the callback onRemoteStreamOk fires, the firing of which indicates that the remote party's media has started flowing.

Example Code

```
function callConnectedHandler ()
{
   /** Application logic **/

   // The call's signaling phase is established. Possibly the indicate this
   // to the user by some UI action, like turing an led green etc ...
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onHangup

Prototype

```
onHangup( reason );
```

Description

This hook fires up if either the remote party initiates a call termination (either by rejecting an incoming call or if it is "busy") or due to the occurrence of a network or internal error.

The call state unconditionally proceeds to termination upon the firing of this callback. This hook is not fired if the termination of the call is initiated by the local user using the "hangup" method.

The implementation of this handler is optional.

Parameters

Parameter	Type	Mandatory	Description
reason	String	Yes	Indicates the cause due to which the current call was terminated. The possible values are: "busy" The called party is busy. "error_channel_closed" Indicates a network error. "error_internal" Indicates a browser related error. "error_bad_answer" Indicates an error condition on the remote side, which inhibits it from accepting the call. "*" Indicates any other reason apart from the ones listed above. The wild card in the string indicates that this string is free flowing string. The contents of the string are Controller Application dependent.
mime_type	String	No	Specifies the optional mime-type sent by peer side or Controller application.
text_str	String	No	Specifies the text string sent by peer side or Controller application.

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onHangup").

A typical implementation could indicate the actual reason for the call termination. Note that the call is not fully disconnected until the callback onDisconnect fires.

Example Code

```
function callHangupHandler(reason)
{

   /** Application logic **/

   if (reason === "busy") {

       // Possibly play a busy tone and inform the user that the remote
       // party is busy and cannot take this call.

   }
   else {

       /** This part takes care of the all the other scenarios where
        *  the "reason" string is unspecified/Application dependent
        **/
   }

   return;
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onDisconnect

Prototype

```
onDisconnect ( );
```

Description

This hook fires on the actual termination of an ongoing call. This occurs as a result of call hangup and is the next hook to fire *after* onHangup.

The implementation of this handler is optional.

Parameters

None

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onDisconnect").

This callback indicates that all the resources associated with the current call are cleared and the library is back in the "IDLE" state and ready to make or receive a new call.

Example Code

```
function disconnectHandler ()
{
   /** Application logic **/

   // Maybe display the fact the call is truly ended and we are ready
   // to make or receive another call.

   return;
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onMessage

Prototype

```
onMessage( fromName, contentType, message );
```

Description

This hook (or handler) fires on the receipt of a text message from the Controller Application. This can occur at any time after the user is registered, even during an established call. The specific business function of this functionality depends on the Controller Application implementation and could be used for text based features like chatting or announcements.

The implementation of this handler is optional.

Parameters

Parameter	Type	Description
fromName	String	Indicates the identifier of the source of the message, and its format is Application dependent.
contentType	String	The mime-type of the message coming in. This type can be defined by the Controller Application.
message	String	The actual content of the message.

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onMessage").

Example Code

```
function messageHandler(fromName, contentType, message)
{
   /** Application logic **/

   // Maybe update the chat window with the incoming message

   return;
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onInfo

Prototype

```
onInfo( contentType, message );
```

Description

This hook fires on the receipt of a text message from the Controller Application. This can occur only during an established call. The specific business function of this functionality depends on the Controller Application implementation and could be used for text based features like chatting or announcements.

The implementation of this handler is optional.

Parameters

Parameter	Type	Description
contentType	String	The mime-type of the message coming in. This type can be defined by the Controller Application.
message	String	The actual content of the message.

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onInfo").

Example Code

```
function infoHandler(contentType, message)
{
   /** Application logic **/

   // Maybe update the chat window with the incoming message or show
   // a popup with the text of the anouncement.

   return;
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onInfoAck

Prototype

```
onInfoAck( contentType, message );
```

Description

This hook might fire as an acknowledgement to an info message send – sendInfo() from the client side. This acknowledgement to info can occur only during an established call. The specific business function of this functionality depends on the Controller Application implementation.

The implementation of this handler is optional.

Parameters

Parameter	Type	Description
contentType	String	The mime-type of the message coming in. This type can be defined by the Controller Application.
message	String	The actual content of the message.

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onInfoAck").

Example Code

```
function infoAckHandler(contentType, message)
{
   /** Application logic **/

   // Info message send by client webrtc.js is acknowledged by Controller
//Application. The acknowledgement is optional and depends on Controller
Application logic whether or not to send ack to info message received.

   return;
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onUserMediaOk

Prototype

```
onUserMediaOk( localMediaType );
```

Description

This hook fires when the WebRTC JavaScript API library’s attempt to acquire the local microphone and camera is successful and is a response to the Acquire Local Media method. Any outgoing or incoming calls can only be done once this callback has fired.

The implementation of this handler is optional.

Parameters

Parameter	Type	Description
localMediaType	String	Provides info about local media access. Possible values are "audiovideo", "audio". If webcam is not present or disabled then localMedia would be set to "audio" else "audiovideo".

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onUserMediaOk").

A typical implementation could choose to highlight some icon to indicate to the user that the Application is ready to make or receive calls.

Example Code

```
function userMediaSuccessHandler(localMediaType)
{
   /** Application logic **/

   // Change or highlight some icon to indicate readiness

   return;
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onUserMediaFail

Prototype

```
onUserMediaFail( reason );
```

Description

This hook (or handler) fires when the WebRTC JavaScript API library's attempt to acquire the local microphone and camera fails. This is an error response to the Initialize method. No outgoing or incoming calls can be made.

The implementation of this handler is optional.

Parameters

Parameter	Type	Description
reason	String	Indicates the cause due to which the current call was terminated. The actual reason string is browser specific.

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onUserMediaFail").

A typical implementation could choose to highlight the failure to the user, so that the user can take corrective action.

Example Code

```
function userMediaFailureHandler()
{
   /** Application logic **/

   // Change or highlight some icon to indicate failure

   return;
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onRemoteStreamOk

Prototype

```
onRemoteStreamOk( );
```

Description

This hook fires when the first media arrives from the remote party. This callback occurs both, in the outgoing and incoming call scenarios. Firing of this callback indicates that the call is fully established.

The implementation of this handler is optional.

Parameters

None

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onRemoteStreamOk").

A typical implementation could choose to highlight some icon to indicate to the user that the remote party can be ‘seen’ and ‘heard’.

Example Code

```
function remoteStreamAddedHandler()
{
   /** Application logic **/

   // The call is now fully established.
   // Change or highlight some icon to indicate remote media activated

   return;
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None

onDeregister

Prototype

```
onDeregister (reason);
```

Description

This hook fires when the connection with the WebRTC Signaling Endpoint is lost. This indicates a possible network error condition, and the user will need to Register and Initialize again before any calls can be made or received.

The implementation of this handler is optional.

Parameters

Parameter	Type	Mandatory	Description
reason	String	No	Optional reason parameter describing reason for deregister.

Usage

The actual implementation of this function must be defined by the user. A reference to this function is passed as a parameter to the setHandlers method (as the key "onDeregister").

A typical implementation could choose to highlight this critical situation to the user through a popup or a highlighted text.

Example Code

```
function deregisterHandler ()
{
   /** Application logic **/

   // The user has been de-registered.
   // Change or highlight some icon to indicate the situation.

   return;
}

// A reference to the above implemented handler should have been registered
// to the WebRTC library during the "setHandlers ()" method.
```

Return

None