Oracle® Communications WebRTC Session Controller

Web Application Developer's Guide

Release 7.0

E40978-01

November 2013

ORACLE®

Oracle Communications WebRTC Session Controller Web Application Developer's Guide, Release 7.0

E40978-01

Contents

3 Setting Up Security

4 Setting Up Audio Calls in Your Applications

5 Setting Up Video Calls in Your Applications

6 Setting Up Data Transfers in Your Applications

9 WebRTC Session Controller JavaScript API Error Codes and Errors

10 Sample Audio Call Application

Preface

This document provides an overview of the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) that supports multimedia and data stream communications in multiple platforms running under multiple protocols.

Audience

This document is intended for developers who use WebRTC Session Controller JavaScript API to create multimedia applications that run in WebRTC-enabled browsers.

Related Documents

For more information, see the following documents:

- *Oracle Communications WebRTC Session Controller Concepts*
- *Oracle Communications WebRTC Session Controller Extension Developer's Guide*
- *Oracle Communications WebRTC Session Controller JavaScript API Reference*
- *Oracle Communications WebRTC Session Controller Release Notes*
- *Oracle Communications WebRTC Session Controller System Administrator's Guide*
- *Oracle Communications WebRTC Session Controller Security Guide*

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

1

Creating HTML5 Applications for WebRTC-Enabled Browsers

This chapter presents an overview of how you can use the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library to create multimedia applications that run in WebRTC-enabled browsers.

About Applications for WebRTC-Enabled Browsers

WebRTC-enabled browsers are web browsers that have real-time communications (RTC) capabilities. The WebRTC standardization effort employs standardized browser capabilities, JavaScript API, and HTML5 to support real-time multimedia communication in applications without the use of any browser plugins. For more information, see the WebRTC website at `http://www.webrtc.org`.

WebRTC Session Controller JavaScript API enables your web applications to communicate with WebRTC Session Controller so that the applications can allow the user to make calls, configure the callbacks to handle incoming calls, notifications, media state in the call, session state changes, and so on.

Web applications developed for WebRTC-enabled browsers can establish real-time communication with each other and with legacy network services. To access such applications, a subscriber needs to be connected to the Internet and use a device (such as a mobile phone, a laptop, a tablet or a desktop computer) equipped with a WebRTC-enabled browser.

Such applications enable end users to perform a multitude of tasks. Suppose that you create an application for the web pages of a real-estate company. When an interested party, such as a buyer's agent, accesses the company's web page, your application starts to respond to the agent's actions while the agent is on that web page. The content of the session managed by your application could include:

- A call session when the buyer's agent uses the calling feature in your application to contact and communicate with the seller's agent
- An online video chat between the two agents, where your application manages the audio and video synchronization
- Sending and/or receiving text or data files, such as a data sheet about the property with a photo of the house
- Sending and/or receiving video data, such as an online tour of the house
- Signing of some initial terms using electronic signatures

About Your Application Development Environment

WebRTC Session Controller supports the following building blocks required for your web application development:

- WebRTC Session Controller Signaling Engine. See "About WebRTC Session Controller Signaling Engine".
- WebRTC Session Controller. See "About the WebRTC Session Controller and Your Applications".
- WebRTC-enabled browsers. See "About the Supported WebRTC-Enabled Browsers".
- JavaScript. See "About JavaScript".
- JavaScript Object Notation. See "About the Browser Protocols and Your Applications".

About WebRTC Session Controller Signaling Engine

WebRTC Session Controller Signaling Engine manages the connectivity between the browser and the end network services. Sitting between the browser and the telecommunication network, it does the following:

- Acts as an intermediary between the web browser and the telecommunication network services, thereby making the browser a client of the network services.
- Provides security to the interactions between your applications and the telecommunication network services.
- Provides the JavaScript API enabling you to develop applications targeted for WebRTC-enabled browsers.

For more information on Signaling Engine, see *WebRTC Session Controller Concepts*.

About the WebRTC Session Controller and Your Applications

The WebSocket uniform resource identifier (URI) your application uses to connect to the WebRTC Session Controller identifies your application, its configuration, and extensions to that default configuration (when present). All interactions between the WebRTC Session Controller and your application take place within that default or extended configuration.

For more information on WebRTC Session Controller, see *WebRTC Session Controller Extension Developer's Guide*.

About the Supported WebRTC-Enabled Browsers

WebRTC Session Controller works with any WebRTC-enabled browser. Currently, it is certified with the Google Chrome browser and the Mozilla Firefox browser. See *WebRTC Session Controller Installation Guide* for more information.

About JavaScript

The business logic of a web application is implemented in JavaScript along with HTML and CSS for the presentation layer. Applications written in JavaScript can interact with the user, control the browser, communicate asynchronously, and alter the content displayed on the browser page.

About the Browser Protocols and Your Applications

WebRTC-enabled browsers are equipped with the WebRTC API. For more information, go to the WebRTC website at http://www.webrtc.org/reference/native-apis.

The WebRTC Session Controller JavaScript API library communicates with WebRTC Session Controller using JsonRTC protocol for communication-related functions such as call control, file transfer, and message notification. JsonRTC protocol is a sub protocol of the MessageBroker WebSocket protocol. For more information on JsonRTC, see Appendix A of *WebRTC Session Controller Extension Developer's Guide*.

Your applications can use the WebRTC Session Controller JavaScript API to set up and manage communication-related functions associated with calls and subscriptions. See "About Using the WebRTC Session Controller JavaScript API" for a description of the components of the WebRTC Session Controller JavaScript API.

About the Conventions Used in This Guide

This guide uses the following conventions:

- Whenever the term "application" is used, it refers to a WebRTC-enabled web application.
- The WebRTC Session Controller JavaScript API class objects, their events, and methods are shown in bold font. For example:

  Session, **CallConfig**, **onIncomingCall**, and **getValue**

- Italicized words are placeholders. For example:

  wscSession, *callObj*, *callConfig*, and so on.

2

About Using the WebRTC Session Controller JavaScript API

This chapter presents a general overview of the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library for the use of web application developers.

> **Note:** This document assumes that you have experience with developing applications using HTML5 features designed for WebRTC-enabled browsers. Its focus is restricted to how you can use the WebRTC Session Controller JavaScript API library to manage real-time communication featuring media stream and data transfers.

This chapter covers the following topics:

- About the wsc Namespace
- About the Application HTML File
- About Monitoring Your Application WebSocket Connection
- About Handling Events in the Application Environment
- Managing the Sessions in Your Application

For information on error codes used by this API library, see "WebRTC Session Controller JavaScript API Error Codes and Errors".

For information on the Service Provider Interface (SPI) functions supported by this API library, see "Extending Your Applications Using WebRTC Session Controller JavaScript API".

> **Note:** Creating and implementing the design of the application's page, the appearance of its user interface and display elements are beyond the scope of this document.

See *WebRTC Session Controller JavaScript API Reference* for more information on the individual WebRTC Session Controller JavaScript API classes.

About the wsc Namespace

The **wsc** namespace exposes the WebRTC Session Controller JavaScript API library so that you can extend specific objects or methods in your applications.

About Using the WebRTC Session Controller JavaScript API Library

The WebRTC Session Controller JavaScript API library enables you to set up and manage real-time communication-related functionality associated with a call session, for example, audio and video call management, file transfers, message notifications, and so on.

About the API Used for General Tasks

The following WebRTC Session Controller JavaScript API classes are used to perform general tasks in your application:

- **wsc.Session** to manage the application session. See "Managing Sessions with wsc.Session".
- **wsc.AuthHandler** to authenticate users. See "Authenticating Users with wsc.AuthHandler".
- **wsc.SESSIONSTATE** to manage changes in the state of the application session. See "Handling Session State Changes".
- **wsc.LOGLEVEL** to set the logging level. See "Debugging Your Application with wsc.LOGLEVEL".
- **wsc.ERRORCODE** to respond to errors. See "WebRTC Session Controller JavaScript API Error Codes and Errors".

In addition, your application needs to verify the media streaming capabilities of the browser. See "Verifying Browser Capabilities".

About the API Used for Call-Related Tasks

The following WebRTC Session Controller JavaScript API classes are used to perform call-related tasks in your application:

- **wsc.CallPackage** to manage call applications. See "Managing Calls with wsc.CallPackage".
- **wsc.Call** to manage a call. See "Managing a Call with wsc.Call".
- **wsc.CallConfig** to set up the capabilities of the call. See "Specifying the Configuration for Calls with wsc.CallConfig".
- **wsc.CALLSTATE** to manage the changes in the call state. See "Handling Changes in Call States".
- **wsc.MEDIASTREAMEVENT** to manage the changes in the media stream. See "Handling Changes in Media Stream States".
- **wsc.DataTransfer** to manage a data channel between two peers. See "Transferring Data With wsc.DataTransfer".
- **wsc.DataSender** to send raw data over the data channel. See "Sending Data Using wsc.DataSender".
- **wsc.DataReceiver** to receive raw data over the data channel. See "Receiving Data Using wsc.DataReceiver".

About the API Used for Message-Related Tasks

The WebRTC Session Controller JavaScript API classes used to perform message and notification-related tasks in your application are described in "Setting Up Message Alert Notifications".

About Extending WebRTC Session Controller JavaScript API

The ways in which you can extend your applications by extending WebRTC Session Controller JavaScript API classes are described in "Extending Your Applications Using WebRTC Session Controller JavaScript API".

Managing Sessions with wsc.Session

Manage the WebSocket connection between your application and WebRTC Session Controller by using the **wsc.Session** class. It represents a persistent association required for the exchange of information between your application and WebRTC Session Controller Signaling Engine. All the media streams, data transfers, and message alert notifications in your application take place within the scope of **wsc.Session**.

When your application page loads and the user logs in to your application, you first create an instance of the **Session** class such as *wscSession*, before you use any of the other WebRTC Session Controller JavaScript API objects associated with calls, data transfers, or message notifications.

Provide the following information when you create your application's session object:

- The user's name
- The WebSocket URI
- The callback function in your application that should be invoked when the session is created
- The callback function in your application that should be invoked when there is an error in creating the session

See "Setting Up the Session Object" for more information.

Your application's session object is associated with a unique session identifier. If you are creating the session object for the first time, WebRTC Session Controller Signaling Engine assigns the session identifier. You can use this identifier to refresh the session, for example, when the application page reloads. When the session has been successfully created, provide the settings to manage the connection. See "About Monitoring Your Application WebSocket Connection".

Your application's session may have many changes in its state. See "Handling Session State Changes" for information on session states and how your application can manage session state changes.

Authenticating Users with wsc.AuthHandler

Authenticate your application users with the **wsc.AuthHandler** class. This class enables your application to ensure that the user credentials are appropriate and will not disrupt message flow during the life of the session.

For example, based on your user's action, your application may send a request to a target Uniform Resource Locator (URL). WebRTC Session Controller Signaling Engine forwards the request to the target URL. The SIP proxy or registrar server may not allow that request to go further in the target environment if it does not have enough user credential information to do so. When this happens, the server sends back a challenge to WebRTC Session Controller Signaling Engine asking for more credential information.

On receiving the challenge from the SIP server, Traversal Using Relays around Network address translation (TURN) server, or proxy or registrar server, WebRTC Session Controller Signaling Engine forwards the challenge to the WebRTC Session

Controller JavaScript API library. The WebRTC Session Controller JavaScript API library invokes the **refresh** event in your application's authentication handler. Your application can respond to such a challenge by retrieving the user credentials and returning that information as a JSON object. On receiving this information, WebRTC Session Controller Signaling Engine can then send your application's response to the SIP proxy/registrar server.

When you create an instance of the **wsc.AuthHandler** class in your application, set up a callback function to handle the **AuthHandler.refresh** event. In the following example, an application creates an authentication handler called *authHandler* and assigns a callback function called *refreshAuth* to its **refresh** event:

```
// Create the session.
// Here, userName is null. WSC can determine it by the cookie of the request.
wscSession = new wsc.Session(null, webSocketUri, sessionSuccessHandler,
sessionErrorHandler);
// Register a wsc.AuthHandler with the session.
// It provides customized authentication info, such as username/password.
var authHandler = new wsc.AuthHandler(wscSession);
authHandler.refresh = refreshAuth;
```

The callback function has two parameters, *authType* and *authHeaders*. The *authType* entry indicates the authentication type and is one of the following entries:

- **wsc.AUTHTYPE.TURN**: The type of authentication that allows for the client to authenticate with a TURN server. Turn servers facilitate clients behind a network address translator (NAT) or Firewall to communicate with each other in certain network topologies.
- **wsc.AUTHTYPE.SERVICE**: The type of authentication when a back-end SIP application, such as the proxy/registrar, requires user authentication.

Use the value in *authType* to obtain the authentication information from the *authHeaders* and return it as a JSON object, as shown in Example 4–8.

Handling Session State Changes

Session state values are constants, such as CLOSED or CONNECTED. Session states are defined in the **wsc.SESSIONSTATE** enumerator.

If *wscSession* is your application's session object, you can set up a callback function and assign that function to your application's **Session.onSessionStateChange** event handler. Whenever the state of your application's session changes, the WebRTC Session Controller JavaScript API library invokes your application's **Session.onSessionStateChange** event handler and provides the new state.

In the callback function, you can check the new session state against the defined constants and set up appropriate actions to respond to the new state. For example, a change in the value of **wsc.SESSIONSTATE** from RECONNECTING to CONNECTED indicates that the attempt to reconnect succeeded and that the application can proceed. Or if the state changes from RECONNECTING to FAILED, the attempt to reconnect failed. In each case, your application may need to take appropriate action with respect to the user.

For a more information on the **wsc.SESSIONSTATE** enumerator, see *Web Session Controller JavaScript API Reference*.

Debugging Your Application with wsc.LOGLEVEL

Use the **wsc.LOGLEVEL** enumerator object to set up the type of records your application must log. The supported log levels are indicated by the data constants DEBUG (0), INFO (1), WARN (2), ERROR (3), and OFF (4).

To set up the log level for debugging, input the constant at the start of your JavaScript application:

```
wsc.setLogLevel(wsc.LOGLEVEL.DEBUG);
```

Alternatively, you can input the value (0):

```
wsc.setLogLevel(0);
```

See "Sample Setup of Global Variables and WebSocket URI".

Managing Calls with wsc.CallPackage

Use the **wsc.CallPackage** class to manage audio or video communication and/or data transfers in calls made from or received by your application. When you create an instance of **wsc.CallPackage** class, the WebRTC Session Controller JavaScript API library handles the messaging and call flow for all calls created through that object for that application session.

In the following example code, an application creates an instance of **wsc.CallPackage** called *callPackage*. Here, *wscSession* is the application's session with WebRTC Session Controller Signaling Engine.

```
var callPackage;
...
callPackage = new wsc.CallPackage(wscSession);
```

After creating an instance of the **CallPackage** class in your application, assign a callback function to handle each of the following events:

- An incoming call, using the **onIncomingCall** event handler.

  In this callback function, implement the logic to process the incoming call, such as filtering to reject calls from blacklisted numbers or responding when the user accepts or declines the call. For information on how the default **CallPackage** class can be used in your applications, see "Setting Up Audio Calls in Your Applications".

- A reconnected call, using the **onResurrect** event handler.

  In this callback function, implement the logic to handle the call that was dropped momentarily. See "Reconnecting Dropped Calls" for more information.

You use your application's **CallPackage** object to create outgoing calls. See "Managing a Call with wsc.Call".

See "Extending and Overriding WebRTC Session Controller JavaScript API Object Methods" for more information on extending the **Call** and **CallPackage** API classes.

Managing a Call with wsc.Call

Manage all audio, video streams, or data transfers that are associated with a single call session, by using the **wsc.Call** class. The **wsc.Call** object represents a single call and is used within a call package.

You can set up your application's **Call** object in the following ways:

- When your application user initiates a call, create the call object using the **CallPackage.createCall** method and provide your application's call configuration.
- When your application accepts an incoming call, use the incoming call object and the remote call configuration for the resulting call session. The WebRTC Session Controller JavaScript API library invokes the **CallPackage.onIncomingCall** event handler and provides the incoming call object and the caller's call configuration, as shown in Example 4–13.

Manage changes in the state of the call and its associated audio, media or data channel with the event handlers of the **wsc.Call** class, by implementing the logic in the callback function you assign for each event. For:

- Call state changes, use the **onCallStateChange** event handler. See "Handling Changes in Call States".
- Media state changes, use the **onMediaStreamEvent** event handler. See "Handling Changes in Media Stream States".
- Data transfer object creation, use the **onDataTransfer** event handler. See "Transferring Data With wsc.DataTransfer".

Specifying the Configuration for Calls with wsc.CallConfig

Specify the audio, video, and data channel capability for calls made from your application, with the **wsc.CallConfig** class.

When you create an instance of the **wsc.CallConfig** class in your application, set up the direction of the audio and video elements in the local media stream. Use the **wsc.MEDIADIRECTION** enumerator to specify the direction of the local media stream as one of the following:

- **wsc.MEDIADIRECTION.SENDRECV** which indicates that the local media stream can send and receive the media stream.
- **wsc.MEDIADIRECTION.SENDONLY** which indicates that the local media stream can send the media stream.
- **wsc.MEDIADIRECTION.RECVONLY** which indicates that the local media stream can receive the media stream.
- **wsc.MEDIADIRECTION.NONE** which indicates that media is not supported.

Set up the configuration for the data transfers in the **dataChannelConfig** parameter with key-value pairs in JSON format. Input the settings for the media stream and data transfers when you create the call configuration object in your application.

In the following example, an application sets up the local media stream to send and receive audio calls only:

```
var audioMediaDirection = wsc.MEDIADIRECTION.SENDRECV;
var videoMediaDirection = wsc.MEDIADIRECTION.NONE;
```

The application sets up the configuration for data transfers in *dtConfigs* with:

```
var dtConfigs = new Array();
dtConfigs[0] = {"label":"DataLabel", "reliable" : false };
```

Finally, the application uses these parameters to create a call configuration object called *callConfig*:

```
var callConfig = new wsc.CallConfig(audioMediaDirection,videoMediaDirection,
dtConfigs);
```

See "Verifying Browser Capabilities" for information on how to verify the browser support for media streams.

Handling Changes in Call States

Your application needs to respond to the changes in the state of a call. The fields of the **wsc.CALLSTATE** enumerator object hold the various states of a call, such as STARTED, RESPONSED, and ENDED. For more information on the **wsc.CALLSTATE** enumerator, see *Web Session Controller JavaScript API Reference*.

The WebRTC Session Controller JavaScript API library also provides the **wsc.CallState** class which represents the state of the call. To process changes to the current call within the callback function you assign to your application's **Call.onCallStateChange** event handler, use the **wsc.Callstate#status** class and determine the call state status, the code for the current state, and the reason for the current state.

See "Setting Up the Event Handler for Call State Changes".

Handling Changes in Media Stream States

The media stream associated with your application is made up of two media components, local (in your application's browser) and remote (the other party's browser).

Your application needs to respond to changes in the media stream states of the call, whether it is voice or video. The WebRTC Session Controller JavaScript API library provides the **wsc.MEDIASTREAMEVENT** enumerator which defines the following three states each for the local and remote streams.

- Added (**LOCAL_STREAM_ADDED** or **REMOTE_STREAM_ADDED**)
- Removed (**LOCAL_STREAM_REMOVED** or **REMOTE_STREAM_REMOVED**)
- In error (**LOCAL_STREAM_ERROR** or **REMOTE_STREAM_ERROR**)

See "Setting Up the Event Handler for the Media Streams" for information on how to use the **wsc.MEDIASTREAMEVENT** enumerator.

For a more information on the **wsc.MEDIASTREAMEVENT** enumerator, see *Web Session Controller JavaScript API Reference*.

Transferring Data With wsc.DataTransfer

If your application supports features such as gaming, text messaging, chat sessions, and file transfers, you can set up data transfer objects to manage the corresponding data channels in your application. Additionally, you can use the data channels with or without the audio or video streams.

The **wsc.DataTransfer** class manages a data channel between two peers. Each data transfer has a **label**. See "Specifying the Configuration for Calls with wsc.CallConfig" for information on **dataChannelConfig**, the data channel configuration parameter in your application's **CallConfig** object.

You can retrieve the following from your application's **DataTransfer** object:

- The sender of the data transfer as an instance of **wsc.DataSender** class, by using the **getSender** method.
- The receiver of the data transfer as an instance of **wsc.DataReceiver** class, by using the **getReceiver** method.
- The state of the data transfer, by using the **getState** method.

To manage the open, closed, and error states of a data transfer, use the **onOpen**, **onClose**, and **onError** event handlers associated with your application's **DataTransfer** object.

See "Setting Up Data Transfers in Your Applications" for more information on how you can create applications that support data transfers.

Sending Data Using wsc.DataSender

The **wsc.DataSender** API works in connection with **wsc.DataTransfer** and **wsc.DataReceiver** API class objects.

If your application supports the sending of raw data, it can send a raw data object in the data channel of a data transfer as a string or a binary large object (BLOB) in your application's **DataTransfer** object. You can retrieve the identity of the sender by using the **getSender()** method of your application's **DataSender** object. Set up the data object that is to be sent and send it using the **send** method of the **DataSender** object in your application. See "Sample Send Function" for more information.

Receiving Data Using wsc.DataReceiver

The **wsc.DataReceiver** class works in connection with **wsc.DataTransfer** and **wsc.DataSender** classes.

If your application supports receiving raw data, and *receiver* is the instance of **wsc.DataReceiver** in your application, set up a callback function for the application's *receiver*.**onMessage** event handler. In that callback function, retrieve and handle the retrieved data as required by your application. For example:

```
receiver.onMessage = function(evt) {
    var rcvdDataElm = document.getElementById("rcvData");
    rcvdDataElm.value = evt.data;
 ...
}
```

Here, *evt* is the raw data in its entirety such as a text string, BLOBs, or array data.

About the Code Segments Displayed in This Guide

The example code segments shown in this guide focus on the features of the WebRTC Session Controller JavaScript API library and use minimal HTML5 elements for display aspects such as messages and control buttons. Any description of the display aspects of your applications and their CSS elements are beyond the scope of this guide.

The sample applications described in this guide show the use of the **console.log** method for displaying debug messages. When you create your applications, use the JavaScript Console API methods supported in your application's web browser to assist you in your application development process.

About the Application HTML File

This section describes the following aspects of your application's HTML file:

- About Web Applications Using WebRTC Session Controller JavaScript API
- Supporting Libraries
- Verifying Browser Capabilities
- WebRTC Session Controller JavaScript API Error Codes and Errors

Note: Each of the examples used in this document was written up as a single application html file with **wsc.js** as the sole supporting library.

About Web Applications Using WebRTC Session Controller JavaScript API

Every web application that uses audio, media stream, or data transfer (such as chat sessions) does so because of a call or a subscription associated with the application user. For web applications using the WebRTC Session Controller JavaScript API library, the **Call** object or the **Subscription** object is the critical element for providing communication functionality.

General Call Logic of Your Applications

The general logic associated with calls in web applications comprises the following:

- Enabling a user who is logged in to your application to place a call.

  To do so, your application:

  - Sets up the logic necessary to obtain information on the recipient of the call (the callee's identifier).
  - On receiving the number to call from the user, performs the actions necessary to establish the call session between the caller and callee.

- Enabling a user to accept or decline an audio or video call invitation.

  To do so, your application:

  - Sets up the necessary elements to respond to the incoming call request and perhaps filters the incoming call.
  - If necessary, provides the controls for the callee to accept or decline the audio or video call invitation from the caller.
  - If the callee accepts the call, completes the steps to establish the call session.
  - If the callee declines the call, takes appropriate steps.
  - If the caller cancels the call before it is established, takes appropriate steps.

- Monitoring the established call session until one of the parties ends the call.

  To do so, your application:

  - Provides the logic necessary to end the call.
  - Takes appropriate action based on whether the call was ended or a party logged out (thus ending the session).

General Notifications Logic of Your Applications

The general logic associated with message alert notifications in web applications comprises the following:

- Enabling a user who is logged in to subscribe to receiving notifications.

  To do so, your application:

  - Sets up the elements necessary for the user to subscribe for notifications.
  - On receiving the subscription target from the subscriber, sets up the subscriptions.

- Enabling the user to access and process the received notifications.

To do so, your application:

- Sets up the elements necessary to receive the incoming notification and displays it for the user.
- Sets up the logic to respond to the user's actions on his subscriptions.

Supporting Libraries

The **wsc.js** file contains the WebRTC Session Controller JavaScript API library. Reference the **wsc.js** file at the start of your HTML5 application code along with all other JavaScript files used by your application.

Example 2–1 Referencing the WSC.js Library

```
<script type="text/JavaScript" src="wsc_context_root/api/wsc.js"></script>
```

In Example 2–1, *wsc_context_root* represents the HTTP location where the WebRTC Session Controller is provisioned.

Ensure that all the supporting libraries used by your application are referenced appropriately.

Verifying Browser Capabilities

At the start of your application logic, check your browser's capabilities to access local media including audio and/or video media. If your browser does not appear to support the streaming needs of your application, enable your application to perform a graceful exit. If your browser can access the local media and your application obtains the media stream, your application can attach the media stream to the video/audio HTML5 media element, as appropriate.

In the following example, an application checks its browser to see if it can access the local media. If it cannot, the application calls a local function to perform a graceful exit.

```
if (!navigator.mozGetUserMedia && !navigator.webkitGetUserMedia) {
      // Cannot access media. Call function to exit gracefully.
      reptBrowserIssue();
     };
```

Below, the same application employs a utility function named *attachMediaStream* to attach a media stream to a video/audio element, when necessary:

```
var attachMediaStream = null;
attachMediaStream = function(element, stream) {
    element.src = URL.createObjectURL(stream);
    }
```

About Monitoring Your Application WebSocket Connection

The state of the application session depends on the state of the WebSocket connection between your application and WebRTC Session Controller Signaling Engine. The WebRTC Session Controller JavaScript API library monitors this connection.

When you instantiate your session object, you configure how the functionality in WebRTC Session Controller JavaScript API library checks your application's WebSocket connection. Monitor the state of the connection, by setting the following values in your application's **Session** object:

- How often the WebRTC Session Controller JavaScript API library must ping the WebRTC Session Controller Signaling Engine:
  - **Session.busyPingInterval**, when there are subsessions inside the session. The default is 3,000 milliseconds (ms).
  - **Session.idlePingInterval**, when there are no subsessions inside the session. The default is 10,000 ms.
- **Session.reconnectInterval**, which specifies the interval between attempts to reconnect to the WebRTC Session Controller Signaling Engine. The default is 2000 ms.
- **Session.retryCount** which specifies the number of retry attempts to check the ping-pong time out after which the WebRTC Session Controller JavaScript API library reconnects to the WebRTC Session Controller Signaling Engine.
- **Session.reconnectTime**, which specifies the maximum time for the interval during which the WebRTC Session Controller JavaScript API library should attempt to reconnect to the server. If the specified time is reached and the connection still fails, no further attempt is made to reconnect to the WebRTC Session Controller Signaling Engine. Instead, the session *failureCallback* event handler is invoked in your application. The default value is 60,000 ms.

  Note: Verify that the **Session.reconnectTime** value does not exceed the value configured for "WebSocket Disconnect Time Limit" in WebRTC Session Controller.

When your application is active, these values are used to check the state of the web socket connection.

About Handling Events in the Application Environment

Your application may be affected by the following events in your deployment environment, if the application is in operation when the events occur:

- Client rehydration

  At times, the local application page state may be reinitialized if your application user reloaded the page or because the application reloaded itself (perhaps to update to a new version). See "Managing the Sessions in Your Application".

- Network switch over

  A network switch over takes place when the end-user's IP address changes for example, when the end-user's device switches from a WIFI network to 4G network. Issues may arise when there is a switch over in the network environment. When a network switch over takes place, the WebRTC Session Controller JavaScript API library reconnects your application's session with WebRTC Session Controller Signaling Engine and tries to resurrect all the subsessions inside your application's **Session** object, such as the call and the subscription.

  For information on handling a call after a network switch over, see "Restoring a Call Session". For information on handling a subscription after a network switch over, see "Restoring a Subscription Session".

- Clustered server shut down

If the WebRTC Session Controller runs in a cluster environment, the server to which your application browser web browser is connected may shut down. In this scenario, the WebRTC Session Controller JavaScript API library tries to reconnect your application's session to the corresponding fail-over server. If the connection is reestablished, the WebRTC Session Controller JavaScript API library attempts to resurrect all the subsessions inside your application's **Session** object.

For information on handling a call after a server fail over, see "Restoring a Call Session". For information on handling a subscription after a server fail over, see "Restoring a Subscription Session".

Managing the Sessions in Your Application

When you use the WebRTC Session Controller JavaScript API library in your application, your application can create an instance of the **Session** object and its subsessions, such as a call session or a subscription session. This section describes how the WebRTC Session Controller JavaScript API library manages the session and subsession information.

Note: Any discussion on the management of arbitrary session or subsession data is beyond the scope of this document.

How Your Application Saves Session Information

By default, when your application uses the WebRTC Session Controller JavaScript API libraary and your application's session changes, the WebRTC Session Controller JavaScript API library saves all of the session information associated with the following objects in your application's session:

- **CallPackage**
- **Call**
- **MessageAlertPackage**
- **Subscription**

The WebRTC Session Controller JavaScript API library stores this application data in JSON format in the browser's **sessionStorage** object.

Note: If you extend any WebRTC Session Controller JavaScript API class object, save the session identifier (**sessionId**) for use when you attempt to create the current session after your application page reloads.

Recreating the Session Using the Session ID

When your application page reloads, if you have saved the session identifier for the previously created session in your application, use that session identifier to recreate the **Session** object.

In Example 2–2, an application uses the *isPageReload* variable to set up the session as necessary. If the value in *isPageReload* indicates that the page has been reloaded, the application attempts to create the current session by providing the *sessionId* it had previously stored. Otherwise, it creates a new session.

In this application:

- *userName* is the user name.
- *webSocketUri* is the WebSocket connection defined earlier. See "Sample Setup of Global Variables and WebSocket URI".
- *successCallback* is the function to call if the session object was created successfully.
- *failureCallback* is the function to call if the session object was not created.
- *sessionId* is the Session ID stored by the application.

Example 2–2 Recreating the Session Using SessionId Example

```
var isPageReload = false;
var sessionId = null;
...
...
// Create the session. If the page is reloaded, recreate the current session.
if (isPageReload) {
    // Aplication page reload scenario. Input the saved sessionId.
    wseSession = new wse.Session(userName, webSocketUri, successCallback,
failureCallback, sessionId);
    } else {
    // This is a new session. Save sessionId if you have extended any API.
        wseSession = new wse.Session( userName, webSocketUri, successCallback,
failureCallback);
        }
...
```

In addition, this application uses the following functions:

- The *onPageLoad* function sets the *isPageLoad* variable to true if the application page is the result of a page reload.

```
function onPageLoad() {
    if (getSavedPageInfo()) {
        isPageReload = true;
        register();
       }
    }
```

- The *savePageInfo* function saves the *sessionId* in the HTML sessionStorage object.

```
function savePageInfo() {
    sessionStorage.setItem("sessionId", wseSession.getSessionId());
    }
```

- The *getSavedPageInfo* function attempts to retrieve the *sessionId* from the HTML sessionStorage object.

```
function getSavedPageInfo() {
    sessionId = sessionStorage.getItem("sessionId");
    if (sessionId != null) {
        return true;
    }
    return null;
    }
```

- The *successCallback* function invoked when the session is created successfully is not shown here. This function calls the *savePageInfo* function.

How WebRTC Session Controller JavaScript API Library Restores Application Data

When there is a client rehydration, a network switch over, or a clustered server shut down, the WebRTC Session Controller JavaScript API library attempts to restore your application using the *sessionId* you provide when you attempt to recreate your current application session. It uses *sessionId* as the key and loads the saved session data from browser's sessionStorage. It then sends a reconnect message to the WebRTC Session controller Signaling Engine.

If the reconnection request receives a success response from WebRTC Session Controller Signaling Engine, your application's session state goes from **wsc.SESSIONSTATE.RECONNECTING** to **wsc.SESSIONSTATE.CONNECTED**. Monitor this change in the callback function you assign to the **Session.onSessionStateChange** event handler.

When the session regains its **wsc.SESSIONSTATE.CONNECTED** state, the WebRTC Session Controller JavaScript API library provides the following to your application:

- The rehydrated package data to the appropriate **onRehydration** event handler in your application:
  - **CallPackage.onRehydration**. See "Restoring CallPackage Data After Pages Reload"
  - **MessageAlertPackage.onRehydration**. See "Restoring Extended MessageAlertPackage Data After Pages Reload"

  In each case, your application receives the rehydrated data as the parameter in the callback function.

  Important: If you create a custom package, be sure to implement a custom **onRehydration** event handler in that package.

- The resurrected call or subscription to the appropriate **onResurrect** event handler in your application:
  - **CallPackage.onResurrect**. See "Restoring a Call Session".
  - **MessageAlertPackage.onResurrect**. See "Restoring a Subscription Session".

If the WebRTC Session Controller JavaScript API library fails to load data, it invokes the *failureCallback* function associated with the session creation step.

Restoring Your Application Package Data After Your Application Pages Reload

After your application pages successfully reload and the WebSocket connection has been successfully restored, your application can repopulate the data associated with its WebRTC Session Controller JavaScript API objects by doing the following:

- Restoring CallPackage Data After Pages Reload
- Restoring Extended MessageAlertPackage Data After Pages Reload

Restoring CallPackage Data After Pages Reload

To restore the call package data after your application pages successfully reload, set up appropriate actions within the callback function assigned to your application's **CallPackage.onRehydration** event handler. This callback function has one parameter, **rehydratedData**, which contains the data about the call stored in the sessionStorage object of the web browser. Within the callback function you can define the actions with

respect to the recovered call. See "Restoring a Call Session".

When you extend the **CallPackage** object in your application, you can override its **onRehydration** event handler. See "Extending Objects Using the wsc.extend Method" for more information.

Restoring Extended MessageAlertPackage Data After Pages Reload

To handle the subscription session data after your application pages successfully reload, set up appropriate actions within the callback function assigned to your application's **MessageAlertPackage.onRehydration** event handler. This callback function has one parameter, **rehydratedData** which contains the data about the subscription stored in the sessionStorage object of the web browser. Within the callback function you can define the actions with respect to the recovered subscription. See "Restoring a Subscription Session".

When you extend the **MesageAlertPackage** object, you can override its **onRehydration** event handler and use this function to refresh the current subscription with the recovered data. See "Extending Objects Using the wsc.extend Method" for more information.

Restoring a Call Session

If a call is recovered following a page reload, network switch over or server fail over, the WebRTC Session Controller JavaScript API library invokes the **CallPackage.onResurrect** event handler in your application. The resurrected call is provided as the parameter to the callback function. Use the callback function to perform the actions required on the rehydrated call and resume the rehydrated call. See "Resuming Your Application Operation".

In Example 2–3, an application sets up *callHandler* as an instance of the **CallPackage** class object and assigns *onResurrect* as the callback function for its **CallPackage.onResurrect** event handler. The *onResurrect* callback function uses the *rehydratedCall* object.

Example 2–3 Sample onResurrect Function for a CallPackage

```
callHandler = new wsc.CallPackage(wscSession);
if(callHandler){
  callHandler.onResurrect = onResurrect;
}
...
function onResurrect(rehydratedCall) {
  // set callback for call state changed
  rehydratedCall.onCallStateChange = function(newState) {
    ...
  }
  // set callback for media state changed
  rehydratedCall.onMediaStreamEvent= function(mediaStreamEvent, stream) {
    ...
  }
  // resume the call with setting related callback functions.
  rehydratedCall.resume(onResumeCallSuccess, doCallError);
}
```

Restoring a Subscription Session

If your application user's subscription is recovered following a page reload, network switch over or server fail over, the WebRTC Session Controller JavaScript API library

invokes your application's **MessageAlertPackage.onResurrect** event handler. The rehydrated subscription is provided as the parameter to the callback function. Use the corresponding callback function to perform any action it requires. See "Resuming Your Application Operation".

In Example 2–4, an application sets up *subscribeHandler* as an instance of the **MessageAlertPackage** class object and assigns *onResurrect* as the callback function for its **MessageAlertPackage.onResurrect** event handler. The *onResurrect* callback function uses the *rehydratedSubscription* object to restore the application's current **Subscription** object.

Example 2–4 Sample onResurrect Function for a MessageAlertPackage

```
subscribeHandler = new wsc.MessageAlertPackage(wscSession);
if (subscribeHandler) {
    subscribeHandler.onResurrect = onResurrect;
}
...
function onResurrect(rehydratedSubscription) {
    // reset related callback functions
    subscription = rehydratedSubscription;
    subscription.onNotification = onNotification;
    subscription.onEnd = onEnd;
    // initialize other information on page ...
}
```

Resuming Your Application Operation

To resume your application operation following a page reload, a network switch over, or a server fail over:

- For calls: Resume the current call by invoking its **resume** method. See "Reconnecting Dropped Calls".
- For subscriptions: Call the subscription's **isValid** method and take further action. If the user prefers to end the subscription, use the subscription's **end** method to do so. See "Restoring a Subscription".

Important: If you create a custom package, be sure to implement the appropriate logic to resume its call or subscription operation.

3

Setting Up Security

This chapter describes how to set up security for the applications you develop using the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library. For information about configuring security on the web server, see *WebRTC Session Controller Security Guide.*

Handling Login to WebRTC Session Controller

By default, the WebRTC Session Controller JavaScript API library supports the following authentication methods for your web applications:

- Basic authentication. See "Login Using Basic Authentication".
- OAuth authentication. See "Login Using OAuth Authentication".
- Form-based authentication. See "Login Using Form-Based Authentication".

Please refer to your server-side configuration documentation to configure authentication-based login and logout on the web server side.

Login Using Basic Authentication

Basic authentication implements access controls using static, standard HTTP headers. This is one of the default authentication methods supported by WebRTC Session Controller Signaling Engine. For more information on Basic authentication, see the Internet Engineering Task Force website at `http://tools.ietf.org/html/rfc2617`.

When a user attempts to access your application, your application can initiate a login request by sending an HTTPS request to the following uniform resource locator (URL):

`https://`*`wsc-host`*`:`*`wsc-port`***`/login`**

Where:

- *wsc-host* is the host name where WebRTC Session Controller is running.
- *wsc-port* is the listening port for WebRTC Session Controller.

When such a request is made to the URL:

1. The web browser displays a basic authentication dialog window.
2. The user enters his credentials.
3. One of the following occurs:
   - If the credentials are valid, the user is authenticated.
   - If the credentials are not valid, an error response is displayed.

Redirecting After a Successful Login

You can add the following two optional request parameters to the login URL you define in your application to redirect the user after a successful login:

- **redirect_uri**: Specify the uniform resource identifier (URI) of the page the browser should be redirected to after a successful login.
- **wsc_app_uri**: Specify the URI of the WebRTC Session Controller application configured in WebRTC Session Controller which will be invoked by this client after logging in, such as **/ws/webrtc/sample**. The WebRTC Session Controller configuration contains a set of domain names valid for the application. This data is used to validate the domain name in the **redirect_uri**.

The following is an example of an HTTPS login request that redirects the user to a new web page:

https://*wsc-host*:*wsc-port***/login?redirect_uri=https://***name_of_theDomain.com***/index.html&wsc_app_uri=/ws/webrtc/sample**

When you redirect users, WebRTC Session Controller Signaling Engine checks to see if the domain name for **redirect_uri** is set to one of the configured domains for the WebRTC Session Controller application that will be invoked following successful authentication. Your WebRTC Session Controller administrator will have the information you need to access the application.

See *WebRTC Session Controller Extension Developer's Guide* for more information.

Login Using OAuth Authentication

OAuth is an open standard for authentication. OAuth 2.0 is one of the default authentication methods supported by WebRTC Session Controller Signaling Engine.

End users attempting to access your application are redirected to supported third-party websites for authentication. Facebook OAuth token authentication is one example. For information on OAuth authentication, see the OAuth website at http://oauth.net/.

You can set up your web applications to employ the end user's OAuth identity by using the OAuth login mechanism. In this scenario, your web applications can use the user's external OAuth identity, such as Facebook or Google identity, to enable the user to log in to WebRTC Session Controller Signaling Engine. Configure OAuth authentication based on the requirements of the selected OAuth security provider.

From the client side, when the user clicks the **Login** button on the web page, the OAuth process works in the following way:

1. Your application sends an HTTPS request to the Signaling Engine login URL.
2. WebRTC Session Controller Signaling Engine does the following:
   a. Based on the URL query parameters, identifies that the request is for the OAuth login mechanism.
   b. Redirects the browser to the OAuth provider's login dialog page.
3. The user provides his or her credentials to the OAuth provider.
4. The OAuth provider authenticates the user and redirects the user back to WebRTC Session Controller with the OAuth access code information.
5. WebRTC Session Controller Signaling Engine retrieves the OAuth token and user information from the OAuth access code and does the following:

a. Logs the user in to the Signaling Engine domain.

b. Redirects the user back to the final redirect URI specified in your application.

For example, the following is a sample authentication/authorization request generated by an example web application. The end user is redirected to the **loginRedirect.html** page, at the host and port location where your application resides. The request is shown here with carriage returns added to promote its readability:

```
https://wsc-host:wsc-port/login/google?
client_id=12349876.apps.googleusercontent.com&
redirect_uri=http://wsc-host:wsc-port/login/google&
wsc_app_uri=/ws/webrtc/sample&
response_type=code&
scope=email&
oauth_url=https://accounts.google.com/o/oauth2/auth&
final_redirect_uri=http://custapp-host:custapp-port/wscsample/loginRedirect.html
```

In the above request:

- **client_id** specifies the OAuth client ID for the registered WebRTC Session Controller application. This is the client ID you received when you first created and registered the WebRTC Session Controller application with the OAuth provider.
- **redirect_uri** is the configured login URI in WebRTC Session Controller for a specific OAuth provider.
- **wsc_app_uri** is your application's URI. In this example, the application is named **sample** and resides at **/ws/webrtc/**.
- **response_type** specifies the supported OAuth response type. The entry code indicates that your server expects to receive an authorization code.
- **scope** specifies the OAuth scope, indicating which parts of the user's account you wish to access. The value of **scope** depends on the OAuth provider. This example uses **email**.
- **oauth_url** specifies the URL of the OAuth provider's login dialog page.
- **final_redirect_uri** specifies the location to which Signaling Engine should redirect the user after a successful OAuth login.

Login Using Form-Based Authentication

Form-based authentication requires your application to implement the logic to obtain and authenticate the username and password from the application user.

If you plan to use form-based authentication in your applications, do the following:

1. Create a separate web application which enforces form-based authentication for login. For more information on how to create such an application, see *Oracle Fusion Middleware Programming Security for Oracle WebLogic Server* at:

   http://docs.oracle.com/cd/E24329_01/web.1211/e24485/thin_client.htm#autoId11

2. Deploy this application in the WebRTC Session Controller nodes. For more information, see *WebRTC Session Controller Extension Developer's Guide*.

Any one who logs in to this specific web application is also logged in to WebRTC Session Controller Signalling Engine application.

Handling Logout from WebRTC Session Controller

When your user logs out of your application or leaves the browser page, your application also logs out of WebRTC Session Controller. You can optionally redirect the user to the web page from where he was directed to your application.

In your application, send an HTTPS request to:

https://*wsc-host*:*wsc-port*/**logout?redirect_uri=***url_to_redirect_to_after_logout*

The above request logs out the user from the WebRTC Session Controller domain and redirects the browser to the URI specified by **redirect_uri**. If **redirect_uri** is not specified, a message saying that the user has been logged out is displayed.

When an application logs into WebRTC Session Controller, the login is valid for one hour. If the WebSocket is disconnected for any reason within that hour, the application can reconnect without logging in again.After one hour, the user needs to login again for new WebSocket connections to be set up. See *WebRTC Session Controller System Administrator's Guide* for information on session timeout if the WebSocket loses its connection.

4

Setting Up Audio Calls in Your Applications

This chapter shows how you can use the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library to enable your applications users to make and receive audio calls from your applications when your applications run on WebRTC-enabled browsers.

> **Note:** See *WebRTC Session Controller JavaScript API Reference* for more information on the individual WebRTC Session Controller JavaScript API classes.

About Implementing the Audio Call Feature in Your Applications

The WebRTC Session Controller JavaScript API for audio calls enables your web applications to support audio calls made to and received from phones located on applications hosted on other WebRTC-enabled browsers, Session Initialization Protocol (SIP) protocol based applications, and public switched telephone network (PSTN) phones.

To support audio calls in your application, implement the logic to do the following:

- For calls made from by your application user, obtain the callee information and start the process to set up the call session between the caller and callee.
- For calls received by your application user, obtain the callee's response to the incoming call request and respond to the callee accepting or rejecting the incoming call invitation.
- Monitor the call session, taking action to respond to any change in the state of the application session, call session or media stream.
- Take appropriate action when one of the parties ends the call.

This basic logic can be used to support calls with video and data transfers.

About the WebRTC Session Controller JavaScript API Used in Implementing Audio Calls

The following WebRTC Session Controller JavaScript API classes are used to implement audio calls in your web applications:

- **wsc.Session** for the session object
- **wsc.CallPackage** for the call package object
- **wsc.Call** for the call object

- **wsc.CallConfig** for the media configuration in the calls
- The constants defined in the following enumerators:
  - **wsc.SESSIONSTATE**
  - **wsc.CALLSTATE**
  - **wsc.MEDIADIRECTION**
  - **wsc.MEDIASTREAMEVENT**
  - **wsc.ERRORCODE**
  - **wsc.LOGLEVEL**

You can extend the audio call feature in your application to perform custom tasks by extending these API classes.

Setting Up Audio Calls in Your Applications

Use the WebRTC Session Controller JavaScript API library to set up the audio call feature in your application to suit your deployment environment. The specific logic, web application elements, and controls you implement for the audio call feature in your applications are predicated upon how the audio call feature is used in your web application.

To illustrate the basic logic in setting up call capability in web applications using the WebRTC Session Controller JavaScript API library, this section uses a sample application in which the audio call is its primary and sole feature.

About the Sample Audio Call Application

The sample audio call application referenced in this chapter provides the logic necessary to enable two users to place a call to each other. It uses WebRTC Session Controller JavaScript API and supports audio calls only. The sample audio call application obtains the call information from an input field it provides on the application page. The steps in the development process described below refer to this sample audio call application. See "Sample Audio Call Application" to view the complete code.

Overview of Setting Up the Audio Call Feature in Your Application

To set up an audio call feature in your application, requires implementing logic for the following:

1. Setting Up the General Elements for the Audio Call Feature
2. Enabling Users to Make Audio Calls From Your Application
3. Implementing the Logic to Set Up the Call Session
4. Enabling Your Application Users to Receive Calls
5. Monitoring the Call
6. Ending the Call

Setting Up the General Elements for the Audio Call Feature

To set up the audio call feature in your application, include the following in the <head> section of your application:

- The <audio> element

  Set up the <audio> element for the local and remote audio according to your browser.

- The WebRTC Session Controller JavaScript API library (**wsc.js**)

  Reference the **wsc.js** file. If your application uses other supporting libraries, reference them, as well.

See "Sample Audio Call Application".

Setting Up the Main Objects and Values

Use the WebRTC Session Controller JavaScript API to set up the main objects and values at the start of your application:

- Declare a **Session** object, a **CallPackage** object, and a variable for the user name.
- Set the log level as required as described in "Debugging Your Application with wsc.LOGLEVEL".
- Set up the web Socket uniform resource identifier (URI) for the WebLogic Server and the login and logout URIs, if your application uses them. The WebSocket URI is required when you create a session object in your application.

Example 4–1 shows how the sample application described in this chapter sets up the WebSocket URI and global variables.

Example 4–1 Sample Setup of Global Variables and WebSocket URI

```
var wscSession, callPackage, userName, caller, callee;
wsc.setLogLevel(wsc.LOGLEVEL.DEBUG);

// Save the location from where the user accessed this application.
var savedUrl = window.location;

// This application is deployed on WebRTC Session Controller (WSC).
var wsUri = "ws://" + window.location.hostname  + ":" + window.location.port + "/ws/webrtc/sample";

//  loginURI is the location from where the user accesses the application.
//  logoutURI is the location to which the user is redirected after logout.
   ...
```

Here:

- **window.location.hostname** and **window.location.port** define the location for Signaling Engine associated with the audio call application.
- */ws/webrtc/sample* indicates that the sample application is deployed in WebRTC Session Controller.

Current Stage in the Development of the Audio Call Feature

At this point, you have completed the setup for the general elements required for an audio call application. You now need to enable users to make a call from the audio call application.

Enabling Users to Make Audio Calls From Your Application

To enable users to make a call from your application, complete the following tasks:

- Setting Up the Configuration for Calls Supported by the Application

Setting Up the Configuration for Calls Supported by the Application

The WebRTC Session Controller JavaScript API library provides the **CallConfig** class object to define the audio, video, and data channel capabilities of a call. To create a **CallConfig** class object, use the syntax:

```
wsc.CallConfig(audioMediaDirection, videoMediaDirection, dataChannelConfig)
```

When you create your application's **CallConfig** class object, specify the configuration for the local audio media stream in **audioMediaDirection** and video media stream in **videoMediaDirection** as described in "Specifying the Configuration for Calls with wsc.CallConfig".

The **dataChannelConfigs** parameter is used to define data transfers (as in text messaging sessions), and is an array of JavaScript Object Notation (JSON) objects that describe the configuration of the data channel. See "Setting Up the Configuration for Data Transfers in Chat Sessions" for more information on setting up the configuration for data transfers.

Set the local audio, video stream, and data transfer configuration for calls in your application based on your browser properties and your web application's requirements.

The sample audio call application supports audio calls in both directions and creates a call configuration object called *callConfig*, as shown below in Example 4–2:

Example 4–2 Sample Call Configuration Object

```
// Create a CallConfig object.
var audioMediaDirection = wsc.MEDIADIRECTION.SENDRECV;
var videoMediaDirection = wsc.MEDIADIRECTION.NONE;
var callConfig = new wsc.CallConfig(audioMediaDirection, videoMediaDirection);
```

Setting Up the Session Object

The WebRTC Session Controller JavaScript API library provides the **wsc.Session** class object to encapsulate the session between your web application and WebRTC Session Controller Signaling Engine. To create an instance of the **Session** class, use the following syntax:

```
wsc.Session(userName, webSocketUri, successCallback, failureCallback, sessionId)
```

Where:

- *userName* is the user name.
- *webSocketUri* is the WebSocket connection defined earlier in Example 4–1.
- *successCallback* is the function to call if the session object was created successfully.
- *failureCallback* is the function to call if the session object was not created.
- *sessionId* is the Session Id. It is needed if you are refreshing an existing session.

To set up a session object in your application:

- Create an instance of the **wsc.Session** object.
- Set up the logic for the *successCallback* and *failureCallback* functions.
- If your application authenticates its users before allowing them to make calls:
  - Set up an authentication handler for that session. Input the session object when you instantiate the **wsc.AuthHandler** class.
  - Assign the callback function to the **refresh** field of your application's authentication handler.
  - Set up the logic for the callback function. See Example 4–8.
- Specify the values for **busyPingInterval**, **idlePingInterval**, and **reconnectTime**. These settings determine how your application's session is monitored. See "About Monitoring Your Application WebSocket Connection".
- Manage the changes in the state of your application session in the following way:
  - Assign a callback function to your application's **Session.onSessionStateChange** event handler.
  - Set up the actions to be performed by the callback function. See "Handling Session State Changes".

The sample audio call application performs these tasks inside a function called *setSessionUp*. When the sample audio call application page loads, the JavaScript onPageLoad function runs and it calls the *setSessionUp* function as shown below.

```
// The onPageLoad event handler.
function onPageLoad() {
    setSessionUp();
}
```

Within the *setSessionUp* function, the sample audio call application:

- Creates an instance of the **Session** class object called *wscSession*, with:
  - *wsURI* as its WebSocket connection.
  - *sessionSuccessHandler* as the callback function for a successful creation of the session.
  - *sessionErrorHandler* as the callback function for a successful creation of the session.
- Registers an authentication handler called *authHandler* with *wscSession*.
- Configures the monitoring time intervals for *wscSession*.
- Assigns a callback function called *sessionStateChangeHandler* to the application's **onSessionStateChange** event handler. This callback function manages the changes in the application's session state.

Example 4–3 shows the *setSessionUp* function implemented in the sample audio call application:

Example 4–3 Sample Session Object Setup

```
// This function sets up and configures the WebSocket connection.
function setSessionUp() {
    console.log("In setSessionUp().");
```

```
    // Create the session. Here, userName is null.
    // WSC can determine it using the cookie of the request.
    wscSession = new wsc.Session(null, wsUri, sessionSuccessHandler,
sessionErrorHandler);
    // Register a wsc.AuthHandler with session.
    // It provides customized authentication information, such as
    // username and password.
    var authHandler = new wsc.AuthHandler(wscSession);
    authHandler.refresh = refreshAuth;

    // Configure the session.
    wscSession.setBusyPingInterval(2 *1000);
    wscSession.setIdlePingInterval(6 * 1000);
    wscSession.setReconnectTime(2 * 1000);
    wscSession.onSessionStateChange = sessionStateChangeHandler;
    console.log("Session configured with authhandler, intervals and
sessionStateChange handler.\n");
}
```

Setting Up the Call Package for the Session

The WebRTC Session Controller JavaScript API library provides the **CallPackage** class to manage the calls and all the messaging workflow with WebRTC Session Controller Signaling Engine. To create an instance of the **CallPackage** class, use the following syntax:

```
wsc.CallPackage(session)
```

Where *session* is the instance of the **Session** object in your application.

To configure the call package to manage the audio calls in your application:

- Create an instance of the **CallPackage** class object for the application session.
- Implement your application logic for incoming calls in the following way:
  - Assign a callback function for the **CallPackage.onIncomingCall** event handler.
  - Set up the actions to be performed by the callback function.
- Implement your application logic to refresh a call that was dropped momentarily:
  - Assign a callback function for the **CallPackage.onResurrect** event handler.
  - Set up the actions to be performed by the callback function.

The sample audio call application sets up a call package called *callPackage*. It sets up the call package within a callback function called *sessionSuccessHandler* which is called when the application session is created. To process incoming calls, the sample audio call application assigns a function named *onIncomingCall* to the **Call.onIncomingCall** event handler for incoming calls. This callback function is describer later in "Responding to Your User's Actions on an Incoming Call". Additionally, the sample audio call application retrieves the name of the user.

Example 4–4 shows the *sessionSuccessHandler* callback function.

Example 4–4 Sample CallPackage Setup

```
function sessionSuccessHandler() {
    console.log(" In sessionSuccesshandler.");

    // Create a CallPackage.
```

```
    callPackage = new wsc.CallPackage(wscSession);
    // Bind the event handler of incoming call.
    if(callPackage){
        callPackage.onIncomingCall = onIncomingCall;
    }
    console.log(" Created CallPackage..\n");
    // Get user Id.
    userName = wscSession.getUserName();
    console.log (" Our user is " + userName);
}
```

Handling Session State Changes

When your application's session state changes, the WebRTC Session Controller JavaScript API Library invokes the application session object's **Session.onSessionStateChange** event handler. The new session state for the call is provided as input to your application.

Monitor the different states in the callback function you assigned to your application session object's **Session.onSessionStateChange** event handler. Specify the actions your application must take for each of the state changes you include.

The **wsc.SESSIONSTATE** enumerator contains the different states of a session defined as constants such as **NONE** when the session is created, **CONNECTED** when the session connects with the server, **CLOSED** when the session closes normally, and so on. See *WebRTC Session Controller JavaScript API Reference* for more information.

The sample audio call application assigns a callback function named *sessionStateChangeHandler* to its application session object's **Session.onSessionStateChange** event handler. In that callback function, the sample audio call application monitors and implements logic for three session states, **CONNECTED**, **FAILED**, and **RECONNECTING**. When the session state is **CONNECTED**, the sample audio call application calls a function named *displayInitialControls* to obtain the callee's name.

Example 4–5 shows the *sessionStateChangeHandler* callback function.

Example 4–5 Sample Session State Handler Callback Function

```
function sessionStateChangeHandler(sessionState) {
    console.log("sessionState : " + sessionState);
    switch (sessionState) {
        case wsc.SESSIONSTATE.RECONNECTING:
        setControls("<h1>Network is unstable, please wait...</h1>");
        break;
        case wsc.SESSIONSTATE.CONNECTED:
        if (wscSession.getAllSubSessions().length == 0) {
            displayInitialControls();
        }
        break;
        case wsc.SESSIONSTATE.FAILED:
        setControls("<h1>Session Failed, please logout and try again.</h1>");
        break;
    }
}
```

Obtaining the Callee Information

Your application can obtain the callee information in a number of ways. Ensure that, if the user is given a choice of controls such as canceling the operation or logging out, the corresponding callback functions are invoked in your application.

The sample audio call application uses a function called *displayInitialControls* to obtain the callee information. In it, the sample audio call application defines a simple user interface consisting of input fields and control buttons to receive the callee's name. The 'onclick'='*functionName()*' for each button triggers the next step for that event. For example, the *onCallSomeOne()* function is invoked when the *Call* button is selected.

Example 4–6 shows the *displayInitialControls* callback function.

Example 4–6 Sample displayInitialControls Function

```
function displayInitialControls() {
    console.log ("In displayControls().");
    var controls = "Enter Your Callee: <input type='text' name='callee' id='callee'/><br><hr>"
                    + "<input type='button' name='callButton' id='btnCall'  value='Call'
onclick='onCallSomeOne()'/>"
                + "<input type='button' name='cancelButton' id='btnCancel'  value='Cancel'
onclick='' disabled ='true'/><br><br><hr>"
                + "<input type='button' name='logoutButton' id='Logout'  value='Logout'
onclick='logout()'/>"
                + "<br><br><hr>";
    setControls(controls);
    var calleeInput = document.getElementById("callee");

    if (calleeInput) {
        console.log (" Waiting for Callee Input.");
        console.log (" ");
        if(userName != calleeInput) {
            calleeInput.focus();
        }

    }
}
```

Current Stage in the Development of the Audio Call Feature in Your Application

At this stage in the development of the audio call feature in your application:

- The general elements required for audio calls are set.
- Your application can obtain the callee information.
- The application logic for the following functions is implemented:
  - *successCallback* function invoked when the application's session object is created
  - *failureCallback* function invoked when the application's session object is not created
  - The callback function assigned to the **Session.onSessionStateChange** event handler
  - The callback function assigned to the **CallPackage.onIncomingCall** event handler
  - The callback function assigned to the **CallPackage.onResurrect** event handler

Your application now needs the logic to handle both end points, the caller's side which must handle connecting the caller to the callee; and the callee's side which must respond to the callee accepting or declining the incoming call.

Initial Actions of the Sample Audio Call Application

Table 4–1 reports on the sample audio call's actions in enabling a user to make a call from the application. It describes the events that occur on the sample audio call application page, the actions taken by the sample audio call application, and the messages logged by the **console.log** method for this segment of the application code.

Table 4–1 Initial Actions Performed by the Sample Audio Call Application

Sample Audio Call Application Page Events	Actions Taken by the Sample Audio Call Application	Console Log for the Caller (*bob1*)	Console Log for the Callee (*bob2*)
When the page loads, the page displays the control buttons and input fields to allow the user to make a call.	The initial actions taken by the audio call application before the user starts the call or receives a call: ■ **CallConfig**, which defines the calling capability, is configured. ■ When the page loads, the *wscSession* object is created and configured. ■ The session is now in a CONNECTED state. ■ Controls are displayed on the application page. For the audio call, they consist of a callee input field, **Call**, **Cancel** and **Logout** buttons. ■ The call package is created inside the callback for the session success event handler. The example code retrieves the user Id for debugging purposes.	`Created CallConfig with audio stream only.` `Page has loaded. Setting up the Session.` `In setSessionUp(). Session configured with authhandler, intervals and sessionStateChange handler.` `sessionState : CONNECTED` `In displayControls(). Waiting for Callee Input.` `In sessionSuccesshandler. Created CallPackage..` `Our user is bob1@example.com`	`Created CallConfig with audio stream only.` `Page has loaded. Setting up the Session.` `In setSessionUp(). Session configured with authhandler, intervals and sessionStateChange handler.` `sessionState : CONNECTED` `In displayControls(). Waiting for Callee Input.` `In sessionSuccesshandler. Created CallPackage..` `Our user is bob2@example.com`

Implementing the Logic to Set Up the Call Session

When your application has obtained the callee information, it can start the process to establish a call session between the caller and the callee.

To implement the logic to start a call from your application, complete the following tasks:

- Start the call. See "Starting a Call From Your Application"

- Set up the callback function to handle any failure in creating the call. See "Handling Errors Related to Calls".
- If the browser does not support the media stream, set up your application to respond appropriately. See "Handling Changes in Media Stream States" for more information.
- Set up the authentication handler based on whether your application supports Traversal Using Relays around Network address translation (TURN) or SERVICE authentication. See "Retrieving the Appropriate Authentication Headers".
- Provide the logic for the call state event handler. See "Setting Up the Event Handler for Call State Changes".
- Provide the logic for the media stream event handler. See "Setting Up the Event Handler for the Media Streams".

Starting a Call From Your Application

The WebRTC Session Controller JavaScript API library provides the **wsc.Call** class object to represent a call with any combination of audio/video/data channel capability. Use the **createCall** method of the **CallPackage** class to create your application's call object. The syntax to create your application's **Call** object is:

```
callPackage.createCall(target, callConfig, errorCallback)
```

Where:

- *target* is the callee.
- *callConfig* is audio/video/data channel capability of calls defined earlier in Example 4–2.
- *errorCallback* is the function to call if the call was not created.

When you obtain the callee information, implement the logic to start the call in the following way:

- Create an instance of the **wsc.Call** object.
- To handle changes in the call session state:
  - Assign a callback function for the **Call.onCallStateChange** event handler.
  - Set up the actions to be performed by the callback function.
- To handle changes in the state of the media stream:
  - Assign a callback function for the **Call.onMediaStreamEvent** event handler.
  - Set up the actions to be performed by the callback function.
- To handle any updates to the call:
  - Assign a callback function for the **Call.onUpdate** event handler.
  - Set up the actions to be performed by the callback function.
- To handle any error in the call creation:
  - Set up the actions to be performed by your application's *errorCallback* function.
- Start the call with the **Call.start** method.
- Set up other actions as dictated by the environment in which your application is deployed.

The sample audio call application invokes a function called *onCallSomeOne*, when it receives the callee information. In this *onCallSomeOne* function, the sample audio call application does the following:

- Sets up a call object named *call*.
- Configures one function called *setEventHandlers* which handles the changes to the call states and the media stream states in its *call* object.

  The *setEventHandlers* function invokes *callStateChangeHandler* for changes in the call state and *mediaStreamEventHandler* for media stream or data transfer changes in the call. See "Sample Audio Call Application" for more information on the *setEventHandlers* function.
- Starts the call using the **start** method of the *call* object.
- Sets up the controls which allow the user to hang up or cancel the call.
- If the user prematurely ends the call, ends the call using the **end** method of its **Call** object.

Example 4–7 Sample Function to Set Up Call for Caller

```
function onCallSomeOne() {

    // Need the caller callee name. Also storing caller.
    callee = document.getElementById("callee").value;
    caller = userName;
    console.log ("Name entered is " + callee);

    // Check to see if user gave a valid input. Omitted here. See "Sample Audio
Call Application".
    ...
    // To call someone, create a Call object first.
    var call = callPackage.createCall(callee, callConfig, doCallError);
    console.log ("Created the call.");
    console.log (" ");

    if (call != null) {
        console.log ("Calling setEventHandlers from onCallSomeOne() with call
data.");
        console.log (" ");
        setEventHandlers(call);
        // Then start the call.
        console.log ("In onCallSomeOne(). Starting Call. ");
        call.start();
        ...
    }
}
```

Retrieving the Appropriate Authentication Headers

This section applies to your application if it uses an authentication mechanism before allowing users access to its audio call feature.

If an authentication handler has been assigned to your application's **Session** object and your application starts a call or receives a call, the authentication function assigned to the **AuthHandler.refresh** event is called. See Example 4–3.

Set up logic in the callback function assigned to your application's **AuthHandler.refresh** event.

The sample audio call application uses Representational State Transfer (REST) based authentication. The *refreshAuth* function shown in Example 4–8 is for your reference. See "Setting Up Security" for more information on the SERVICE and Traversal Using Relays around Network address translation (TURN) authentication seen in the code below.

Example 4–8 Template for the refreshAuth Function()

```
function refreshAuth(authType, authHeaders) {
    //Set up the response object by calling a function.
    var authInfo = null;

    if(authType==wsc.AUTHTYPE.SERVICE){
        //Return JSON object according to the content of the "authHeaders".
        // For the digest authentication implementation, refer to RFC2617.
        authInfo = getSipAuth(authHeaders);

    } else if(authType==wsc.AUTHTYPE.TURN){

        //Return JSON object in this format:
        // {"iceServers" : [ {"url":"turn:test@<aHost>:<itsPort>",
"credential":"nnnn"} ]}.
        authInfo = getTurnAuth();
    }
    return authInfo;
};
```

If your application uses Digest access authentication, ensure that it sets up the response using the headers in the **authHeaders** object it retrieves. For more information on Digest access authentication, see http://www.ietf.org/rfc/rfc2617.txt.

About Digest Access Authentication

If a Session Initiation Protocol (SIP) network does not support an identity mapping between a web identity and a SIP identity, it might choose to challenge the messages from the application using a WWW-authenticate header as stipulated by RFC 2617. On receiving the WWW-authenticate header, WebRTC Session Controller Signaling Engine sends a JavaScript Object Notation (JSON) form of this header to the WebRTC Session Controller JavaScript API library. In turn, the WebRTC Session Controller JavaScript API library invokes the callback function assigned to your application's **AuthHandler.refresh** event handler.

To provide the appropriate challenge response, do the following in the callback function assigned to your application's **AuthHandler.refresh** event handler:

- Retrieve the appropriate credentials from the user, using your application-specific logic.
- Create your application's challenge response in JSON format and constructed, as stipulated by RFC 2617.
- Return the challenge response to the WebRTC Session Controller JavaScript API library.

This challenge response is used to authenticate your application user with the SIP network.

Example 4–9 shows a sample **authHeader** received by an application that uses Digest authentication. The **authHeader** object is in JSON format.

Example 4–9 Digest Authentication Header Received by an Application

```
{
    "scheme": "Digest",
    "nonce": "a12e8f74-af01-4e74-9714-4d65bae4e024",
    "realm": "example.com",
    "qop": "auth",
    "challenge_code": "407",
    "opaque":
"YXBwLTNjOHFlaHR2eGRhcnxiYWNkMTIxMWFmZDlkNmUyMThmZmI0ZDc4ZmY3ZmY1YUAxMC4xODIuMTMuMTh8Mzc3N2E3Nzc0ODYyMGY4",
    "charset": "utf-8",
    "method": "REGISTER",
    "uri": "sip:<host>:<port>"
}
```

Where:

- <*host*> is the host name for the SIP registrar.
- <*port*> is the listening port for the SIP registrar.

Creating the authHeader Object for the Response

Example 4–10 shows a sample function used by an application to set up the *authHeaders* in its response.

Example 4–10 Sample createResponseHeaders Function

```
function createResponseHeaders(authHeaders) {
// cnonce is the string provided by the client.
//The application MUST implement the MD5 algorithm.
    var
        userName = "alice@example.com",
        password = "********",
        realm = authHeaders['realm'],
        method = authHeaders['method'],
        uri = authHeaders['uri'],
        qop = authHeaders['qop'],
        nc = '00000001',
        nonce = authHeaders['nonce'],
        cnonce = "",
        ha1 = hex_md5(userName + ":" + realm + ":" + password),
        ha2 = hex_md5(method + ":" + uri),
        response;

    if(!qop){
        response = hex_md5(ha1 + ":" + nonce + ":" + ha2);
    } else if(qop=="auth") {
        response = hex_md5(ha1 + ":" + nonce + ":" + nc + ":" + cnonce + ":" + qop
+ ":" + ha2);
    }

    // add client calculated header to the headers.
    authHeaders['username'] = username;
    authHeaders['cnonce'] = cnonce;
    authHeaders['response'] = response;
    authHeaders['nc'] = nc;
    return authHeaders;
};
```

Setting Up the Event Handler for Call State Changes

When your application's call state changes, the WebRTC Session Controller JavaScript API Library invokes your application's **Call.onSessionStateChange** event handler. The new state for the call is provided as input to your application.

The many states of a call, such as **ESTABLISHED**, **ENDED**, and **FAILED** are defined as constants in the **wsc.CALLSTATE** enumerator. See *WebRTC Session Controller JavaScript API Reference* for more information.

Use as many of the constants in **wsc.CALLSTATE** to meet your application's needs. Specify the actions your application must take for each of the state changes you include in the callback function you assigned to your application's **Call.onCallStateChange** event handler, as described in "Starting a Call From Your Application".

Example 4–11 shows how the sample audio call application handles call state changes. It sets up a callback function called *callStateChangeHandler* to monitor for three call states, **wsc.CALLSTATE.ESTABLISHED**, **wsc.CALLSTATE.ENDED**, and **wsc.CALLSTATE.FAILED**. When the sample audio call application's callback function in invoked with **wsc.CALLSTATE.ESTABLISHED** as the new call state, it calls a function called *callMonitor* to monitor the call. See Example 4–14. For the remaining two states, this callback function merely displays the user interface required to place a call.

Example 4–11 Sample Call State Change Handler

```
function callStateChangeHandler(callObj, callState) {
    console.log (" In callStateChangeHandler().");
    console.log("callstate : " + JSON.stringify(callState));
    if (callState.state == wsc.CALLSTATE.ESTABLISHED) {
        console.log (" Call is established. Calling callMonitor. ");
        console.log (" ");
        callMonitor(callObj);
    } else if (callState.state == wsc.CALLSTATE.ENDED) {
        console.log (" Call ended. Displaying controls again.");
        console.log (" ");
        displayInitialControls();
    } else if (callState.state == wsc.CALLSTATE.FAILED) {
        console.log (" Call failed. Displaying controls again.");
        console.log (" ");
        displayInitialControls();
    }
}
```

Setting Up the Event Handler for the Media Streams

When there is a change in the state of the local or remote media stream, the WebRTC Session Controller JavaScript API Library invokes your application's **Call.onMediaStreamEvent** event handler. The new state for the media stream is provided as input to your application.

The **wsc.MEDIASTREAMEVENT** enumerator defines the states of the local or remote media stream as **LOCAL_STREAM_ADDED**, **REMOTE_STREAM_REMOVED**, **LOCAL _STREAM_ERROR**, and so on. See *WebRTC Session Controller JavaScript API Reference* for more information.

Use as many of the constants in **wsc.MEDIASTREAMEVENT** to meet your application's needs. Specify the actions your application must take for each of the state changes you include in the callback function you assigned to your application's

Call.onMediaStreamEvent event handler. Whenever this callback function is invoked with a new state for the media stream, your application logic should perform the action required for the new state.

Example 4–11 shows how the sample audio call application handles media stream state changes using a callback function called *mediaStreamEventHandler*.

Example 4–12 Sample Media Stream Event Handler

```
// This event handler is invoked when a  media stream event is fired.
// Attach media stream to HTML5 audio element.
function mediaStreamEventHandler(mediaState, stream) {
    console.log (" In mediaStreamEventHandler.");
    console.log("mediastate : " + mediaState);
    console.log (" ");

    if (mediaState == wsc.MEDIASTREAMEVENT.LOCAL_STREAM_ADDED) {
        attachMediaStream(document.getElementById("selfAudio"), stream);
    } else if (mediaState == wsc.MEDIASTREAMEVENT.REMOTE_STREAM_ADDED) {
        attachMediaStream(document.getElementById("remoteAudio"), stream);
    }
}
```

Current Stage in the Development of the Audio Call Feature in Your Application

At this stage in the development of the audio call feature in your application:

- The general elements required for audio calls are set.
- Your application can obtain the callee information.
- Your application can retrieve the call information and start a call.
- The application logic for the following functions is implemented:
  - *errorCallback* function invoked when the call is not created
  - The callback function assigned to the **Call.onCallStateChange** event handler
  - The callback function assigned to the **Call.onMediaStreamEvent** event handler
  - The callback function assigned to the **Call.onDataTransfer** event handler
  - The callback function assigned to the **Call.onUpdate** event handler

You can now provide the logic to handle an incoming call.

How the Sample Audio Call Application Starts a Call

Table 4–2 reports on the sample audio call's actions in setting up a call session. It describes the events that occur on the sample audio call application page, the actions taken by the sample audio call application, and the messages logged by the **console.log** method for this segment of the application code. The focus of actions for this part of the application is the caller.

Table 4–2 Sample Audio Call Application Actions in Setting Up a Call

Sample Audio Call Application Page Events	Actions Taken by the Sample Audio Call Application	Console Log for the Caller (bob1)
Signaling Engine asks the user for permission to use the microphone. The call workflow starts.	For the caller (*bob1*) side, the application does the following in the *onCallSomeOne()* callback function: ■ Creates a *call* object with the callee's id, the configuration for calls in this browser, and the necessary call error handler function. ■ Sets up the general event handler to handle changes in the call. ■ Issues the command *call*.**start**. ■ Enables the controls to cancel the call before it is set up. ■ Defines the call and media state change handlers. The browser requests the user to allow access to audio media. If the user gives permission, the local media stream is added.	`In onCallSomeOne()` `Name entered is bob2` `Adding string to name` `Caller, bob1@example.com, wants to call bob2@example.com, the Callee.` `Creating call object to call bob2@example.com` `Created the call.` `Calling setEventHandlers from onCallSomeOne() with call data.` `In setEventHandlers` `In onCallSomeOne(). Starting Call.` `Enabled bob1@example.com to cancel call.` `In mediaStreamEventHandler.` `mediastate : LOCAL_STREAM_ADDED` `In callStateChangeHandler().` `callstate :` `{"state":"STARTED","status":` `{"code":null,"reason":"start call"}}` `In callStateChangeHandler()` `callstate :` `{"state":"RESPONSED","status":` `{"code":180,"reason":"Ringing"}}`

Enabling Your Application Users to Receive Calls

The focus of the actions taken in this section is the callee.

To enable application users to receive calls, do the following:

1. Provide the logic to respond to the callee's actions with respect to the incoming call. See "Responding to Your User's Actions on an Incoming Call".
2. Verify that you have defined the logic for the following tasks with respect to the callee:
   - Setting Up the Event Handler for Call State Changes
   - Setting Up the Event Handler for the Media Streams

Responding to Your User's Actions on an Incoming Call

When a user is logged in to your application and WebRTC Session Controller Signaling Engine receives a call for the user, the WebRTC Session Controller JavaScript API library invokes the **CallPackage.onIncomingCall** event handler in your application. It sends the incoming call object and the call configuration for that incoming call object as parameters to the **CallPackage.onIncomingCall** event handler.

Define the actions to process the incoming call in the callback function assigned to the **onIncomingCall** event handler in the following way:

- Provide the interface and logic necessary for the callee to accept or decline the call.
- Provide logic for the following events in association with the incoming call object:
  - User accepts the call. Run the **accept** method for the incoming call object. This will return the success response to the caller.
  - User declines the call. Run the **decline** method for the incoming call object. This will return the failure response to the caller.
- Assign the callback functions to the event handlers of the incoming call object. These should already have been defined earlier. See "Starting a Call From Your Application".

Example 4–13 shows the *onIncomingCall* callback function used by the sample audio call application:

Note: Example 4–13 uses the simplest set of controls embedded in the *onIncomingCall()* function to inform the user that there is an incoming call.

You can set up your application to filter the information in the remote call object and its configuration to determine how to handle the incoming call, prior to informing the user about the call.

Example 4–13 Sample onIncomingCall Function

```
function onIncomingCall(callObj, callConfig) {

// Draw two buttons for users to accept or decline the incoming call.
// Attach onclick event handlers to these two buttons.
    console.log ("In onIncomingCall(). Drawing up Control buttons to accept or decline the call.");
    var controls = "<input type='button' name='acceptButton' id='btnAccept' value='Accept "
    + callObj.getCaller()
    + " Incoming Audio Call' onclick=''/><input type='button' name='declineButton' id='btnDecline'
value='Decline Incoming Audio Call' onclick=''/>"
    + "<br><br><hr>";
    setControls(controls);

    document.getElementById("btnAccept").onclick = function() {
        // User accepted the call.

        //  Store the caller and callee names.
        callee = userName;
        caller = callObj.getCaller;
        console.log (callee + " accepted the call from caller " + caller);
        console.log (" ");

        // Send the message back.
        callObj.accept(callConfig);
    }
    document.getElementById("btnDecline").onclick = function() {
        // User declined the call. Send a message back.

        // Get the caller name.
        callee = userName;
        caller = callObj.getCaller;
        console.log (callee + " declined the call from caller, " + caller);
```

```
        console.log (" ");

        // Send the message back.
        callObj.decline();
    }

    // User accepted the call. Bind the event handlers for the call and media stream.
    console.log ("Calling setEventHandlers from onIncomingCall() with remote call object ");
    setEventHandlers(callObj);
}
```

Current Stage in the Development of the Audio Call Feature in Your Application

At this stage in the development of the audio call feature in your application:

- The general elements required for audio calls are set.
- Your application can obtain the callee information.
- Your application can retrieve the call information and start a call.
- Your application can alert the user about an incoming call and respond appropriately to the user accepting or declining the incoming call.
- The application logic for the following functions is implemented:
  - Callback functions assigned to the **Session** Object's event handlers
  - The success and error callback functions invoked when a **Session** object is not created
  - Callback functions assigned to the **CallPackage** Object's event handlers
  - Callback functions assigned to the **Call** Object's event handlers
  - The error callback function invoked when a **Call** object is not created

How the Sample Audio Call Application Handles Incoming Calls

Table 4–3 reports on the sample audio call's actions in enabling a user to receive a call. It describes the events that occur on the sample audio call application page, the actions taken by the sample audio call application, and the messages logged by the **console.log** method for this segment of the application code. The focus here is on the callee.

Table 4–3 A breakdown of the Application Actions Needed to Receive a Call

Sample Audio Call Application Page Events	Actions Taken by the Sample Audio Call Application	Console Log for the Callee (bob2)
A call is received. If the user accepts the call, Signaling Engine asks the user for permission to use the microphone. *When permission is given, the local and remote streams are added.*	For the callee (*bob2*) side: Signaling Engine, on receiving the call invitation from the caller, triggers the function configured in the application to handle incoming calls. This is the *call* object's **onIncomingCall()** callback function that was assigned in Example 4–4. The application does the following: ■ Sets up the actions in the callback function to handle changes in the call. ■ Displays control buttons to enable the callee to accept or decline the call.	`In onIncomingCall(). Drawing up Control buttons to accept or deny the call.` `Calling setEventHandlers from onIncomingCall() with callObj` `In setEventHandlers` `User Accepted the call.` `In callStateChangeHandler().` `callstate : {"state":"STARTED","status": {"code":null,"reason":"receive call"}}` `Invoking getTurnAuthInfo` `In mediaStreamEventHandler.` `mediastate : LOCAL_STREAM_ADDED` `In mediaStreamEventHandler.` `mediastate : REMOTE_STREAM_ADDED`

How a Call is Established in the Sample Audio Call Application

This section uses the sample audio call application as an example to describe what happens during the interval between the caller and callee requesting and accepting the call and when the call actually starts.

At the start of the flow, the sample audio call application on the caller's side sends the START or INVITE message to WebRTC Session Controller Signaling Engine which routes the message through the network to the receiving end point. For more information on this, please see *WebRTC Session Controller Extension Developer's Guide*.

At appropriate points in the message flow, the caller and callee are requested to allow access to the audio element in the browser.

The log output taken from the console when the sample audio call application was run is shown in Table 4–4. Note the log output from the call state and media state transfer event handlers. All the action is done by Signaling Engine and the sample audio call application merely receives the final state (ESTABLISHED or FAILED).

Table 4–4 A Log of the Call Flow

Sample Audio Call Application Page Events	Actions Taken by the Sample Audio Call Application	Console Log for the Caller (bob1)	Console Log for the Callee (bob2)
(Activity that takes place behind the browser activity) *For the caller, the media state changes to include the remote media stream only after the call is established.*	The console log describes the flow of the call to the point where the two parties are connected and can hear each other. The application displays the control button enabling either party to conclude the call.	`In callStateChangeHandler(). callstate : {"state":"RESPONSED","status": {"code":200,"reason":"got success response"}}` `In callStateChangeHandler(). callstate : {"state":"ESTABLISHED","status": {"code":null,"reason":"sent complete"}}`	`In callStateChangeHandler(). callstate : {"state":"RESPONSED","status": {"code":200,"reason":"sent success response"}}` `In callStateChangeHandler(). callstate : {"state":"ESTABLISHED","status": {"code":null,"reason":"got complete"}}`

Monitoring the Call

The call is established when the callee accepts the call. However, your application needs to provide some way for both parties to end the call.

Note: A call can be ended by either party (caller/callee).

When a call is ended by one party, the other party will receive a message from the browser that the call has ended and this ENDED state will trigger the message stream event handler to release the local media stream.

See the **Console Log for the Caller** and **Console Log for the Callee** columns in Table 4–6.

To monitor the call and take action, do the following in your application:

- Display the user interface necessary for the user to end the call.
- Provide the logic for the caller or the callee to end the call.
- Take appropriate actions for the following events:
  - A user actively ends the call.
  - The other party ends the call.

As shown in Example 4–14, the sample audio call application does the following:

- Displays two control buttons for the users: "Hang Up" and "Logout".
- Responds to the selection:
  - If *Hang Up* is clicked, ends the call (which ends the call session and releases the call resources).
  - If *Logout* is selected, ends the session (which ends the call and releases the session's resources).

Example 4–14 Monitoring the Established Call

```
function callMonitor(callObj) {
    console.log ("In callMonitor");
    console.log ("Monitoring the call. Setting up controls to Hang Up.");
    console.log (" ");

    // Draw 2 buttons.
    // "Hang Up" button ends the call, but user stays on the application page.
    // "Logout" button ends the session, and user leaves the application.
    // For the complete code, see "Sample Audio Call Application".
    ...
    document.getElementById("btnHangup").onclick = function() {
        ....
        callObj.end();
    };
}
```

How the Sample Audio Call Application Monitors a Call

Table 4–5 reports on the sample audio call application's actions in monitoring a call session. It describes the events that occur on the sample audio call application page, the actions taken by the sample audio call application, and the messages logged by the **console.log** method for this segment of the application code.

Table 4–5 How the Sample Audio Call Application Monitors the Call

Sample Audio Call Application Page Events	Actions Taken by the Sample Audio Call Application	Console Log for the Caller (bob1)	Console Log for the Callee (bob2)
The remote stream is added for the caller. The call takes place. Control buttons are displayed to enable either party to end the call.	When the call state is ESTABLISHED, the application does the following on the caller's (*bob1*) side: ▪ Sets up the controls to enable the caller to end the call. ▪ Adds the remote media stream enabling the caller to hear the "Hello?" On the callee's (*bob2*) side: Sets up the controls to enable the callee to end the call.	In callStateChangeHandler(). callstate : {"state":"ESTABLISHED","status": {"code":null,"reason":"sent complete"}} Call is established. Calling callMonitor. In callMonitor. Monitoring the call. Setting up controls to Hang Up. In mediaStreamEventHandler. mediastate : REMOTE_STREAM_ADDED	In callStateChangeHandler(). callstate : {"state":"ESTABLISHED","status": {"code":null,"reason":"got complete"}} Calling callMonitor. Call established. Setting up controls to Hang Up.

Ending the Call

When either the callee or caller ends the call, the call state goes to ENDED which triggers the browser to stop the call. The local media stream is removed from each browser application.

Set up the next action according to your application's requirements.

In the sample audio call application as shown in Example 4–11, the application calls the *displayInitialControls()* function which renders the controls to make calls.

Table 4–6 reports on the sample audio call application's actions in ending a call session. It describes the events that occur on the sample audio call application page, the actions taken by the sample audio call application, and the messages logged by the **console.log** method for this segment of the application code.

Table 4–6 A Breakdown of How the Sample Audio Call Ends

Sample Audio Call Application Page Events	Actions Taken by the Sample Audio Call Application	Console Log for the Caller (bob1)	Console Log for the Callee (bob2)
One or the other party can end the call. *In this example, bob1, the caller, ended the call.* *The console log for the caller from the callMonitor() function specifies who ended the call.* *At this point note the differences in the console log entries for the caller and callee.* The example code also once again displays the input buttons for the user to make a call.	▪ Either the caller or the callee clicks the control button to end the call. ▪ The state of the call changes to ENDED. ▪ The local media stream for the browser is disconnected. ▪ At this point, your application's logic may vary. ▪ In this example, the controls to make a call are displayed once again.	`In callMonitor. Caller, bob1@example.com, clicked the Hang Up button. Calling call.end now.` `In callStateChangeHandler(). callstate : {"state":"ENDED","status" : {"code":null,"reason":"st op call"}} Call ended. Displaying controls again.` `In displayControls(). Waiting for Callee Input.` `In mediaStreamEventHandler. mediastate : LOCAL_ STREAM_REMOVED`	`In callStateChangeHandler(). callstate : {"state":"ENDED","status" : {"code":null,"reason":"st op call"}} Call ended. Displaying controls again.` `In displayControls(). Waiting for Callee Input.` `In mediaStreamEventHandler. mediastate : LOCAL_ STREAM_REMOVED`

Current Stage in the Development of the Audio Call Feature in Your Application

At this stage in the development of the audio call feature in your application:

- The general elements required for audio calls are set.
- Your application can obtain the callee information.
- Your application can retrieve the call information and start a call.
- Your application can alert the user about an incoming call and respond appropriately to the user accepting or declining the incoming call.
- The application logic for the following functions should be implemented:
  - Callback functions assigned to the **Session** Object's event handlers
  - The success and error callback functions invoked when a **Session** object is not created
  - Callback functions assigned to the **CallPackage** Object's event handlers
  - Callback functions assigned to the **Call** Object's event handlers
  - The error callback function invoked when a **Call** object is not created
- Your application can monitor the established call, take action as necessary when there is a change to the call in any way.

- When one user ends the call, our application can close the call connection successfully.

Closing the Session When the User Logs Out

The **close()** method of the **Session** API is used to close a session with WebRTC Session Controller Signaling Engine. The syntax is:

```
wscSession.close();
```

Set up the logic to close the session according to your application's requirements.

In the sample audio call application, when the user clicks the Logout button, the application calls the *logout* function to close the session as shown in Example 4–15. Additionally, the user is sent back to the location specified in **logoutUri** (which was defined in Example 4–1 at the start of this sample code.

Example 4–15 Sample Logout Function

```
function logout() {
    if (wscSession) {
        wscSession.close();
    }
    // Send the user back to where he came from.
    window.location.href = logoutUri;
}
```

In your environment, the call feature may be one of the many features of your application. For this example, and at this point, the sample audio call application has completed its task. All that remains is to provide the closing entries for the HTML element tags.

The code for the sample audio call application discussed in this chapter can be seen under "Sample Audio Call Application".

Other Actions on Calls

This section describes some of the other actions your application can take on calls.

Gathering Information on the Current Call

You can obtain the following data about the current call by using the methods of the **Call** object:

- The caller or the callee by using the **Call.getCaller** or **Call.getCallee** method respectively.
- The call configuration by using the **Call.getCallConfig** method.
- The call state by using the **Call.getCallState** method.
- The data transfer object by its label using the **Call.getDataTransfer(label)** method.
- The RTCPeerConnection (peer-to-peer connection) of the current call by using the **Call.getPeerConnection** method. For example, when the call employs dual-tone multi-frequency (DTMF) signal tones, use its **getPeerConnection** method to perform operations directly on the WebRTC **PeerConnection** connection.

Note: The peer connection for the current call may change. Always retrieve its current value using the **getPeerConnection** method for your call object, and then use the result.

Supporting Multiple Calls Using CallPackage

Since the **CallPackage** class object can handle an array of calls, you can configure your application to set up and manage an array of calls (both incoming and outgoing). The basic logic outlined in Overview of Setting Up the Audio Call Feature in Your Application can be used in this scenario. Update this logic so that your application properly manages each specific call session in the array of calls with respect to maintaining the details of the call details, handling changes to the call, media or session states.

See "Extending Your Applications Using WebRTC Session Controller JavaScript API" for more information on extending the **Call** and **CallPackage** API.

Managing Interactive Connectivity Establishment Interval

Your application can configure the time period within which the WebRTC Session Controller JavaScript API library uses the Interactive Connectivity Establishment (ICE) protocol to set up the call session. This procedure comes into play when your application is the caller and your application starts the call setup with its **Call.start** command.

About the Use of ICE and ICE Candidate Trickling

ICE is a technique which determines the best possible pairing of the local IP address and the remote IP address that can be used to establish the call session between the two applications associated with the caller and the callee. Each user agent (caller or callee's browser) has an entity (such as WebRTC Session Controller Signaling Engine) which acts as the ICE agent and collects and shares possible IP addresses. The final pair of IP addresses is elected after gathering and checking possible candidates (IP addresses) and taking into account the security of the end point applications and of the call connection. The media connection is established only after the ICE procedure finds an appropriate pair of IP addresses with which to communicate.

ICE candidate trickling is an extension of ICE. In this technique, a caller's ICE agent may incrementally provide candidates to the callee's ICE agent after the initial offer (the request which requires a response) has been dispatched. This ICE candidate trickling process allows the callee's application to begin acting upon the call and setting up the necessary protocol connections immediately, without waiting for the caller to gather all possible candidates. Doing so results in faster call startup in cases where gathering is not performed prior to initiating the call.

For more information on Interactive Connectivity Establishment, see `http://tools.ietf.org/html/draft-rescorla-mmusic-ice-trickle`

About WebRTC Session Controller Signaling Engine and the ICE Interval

WebRTC Session Controller Signaling Engine enables your applications to limit the time taken by the ICE agent to set up a call session by enabling you to specifying the ICE interval your application allows for this deliberation process.

The default value ICE interval for a call setup is 2000 milliseconds.

Signaling Engine checks the status of the ICE candidate periodically. If new candidates are gathered, the ICE agent will attempt to send this information in JSON format in the START message to the other peer.

Retrieving the Current ICE Interval for the Call

To retrieve the current ICE interval, use the **getIceCheckInterval** method of your application's *call* object. The interval is returned in milliseconds.

Setting Up the ICE Interval for the Call

To set the current ICE interval, provide the time interval in milliseconds when you call the **setIceCheckInterval** method of your application's **Call** object.

Updating a Call

When a call is in an ESTABLISHED state, the caller or the callee may wish to update the call in one of a set of supported or configured ways. For example, one or the other party may select or deselect the mute button on a call, or move from an audio to a video format for the call. As a result, your application may need to update the call for the specific reason.

In order to handle this scenario,

- Set up the necessary interface to capture the information your application user provides on:
  - The type of update the user wishes to make
  - The accept or decline response to the update request
- From the point of view of the person initiating the update:
  - Set up the callback function to invoke when your application user requests the update.
  - Configure the parameters (*CallConfig*, and *localStreams*) required for the update.
  - Invoke the **Call.update** method with the *CallConfig*, and *localStreams* parameters.
  - Provide the required logic in the callback function assigned to your application's **Call.onCallStateChange** event handler for each of the possible call state changes relating to updates, **wsc.CALLSTATE.UPDATED** and **wsc.CALLSTATE.UPDATE_FAILED**.
  - Save any data specific to your application.
  - Set up the actions in response to the other party declining the update.
- From the point of view of the person receiving the update:
  - Set up the callback function you assign to the **Call.onUpdate** event handler when your application receives the update request from Signaling Engine.
  - Process the parameters (*CallConfig*, and *localStreams*) required for the update.
  - Invoke the **Call.accept** method with *CallConfig*, and *localStreams* parameters.
  - Set up the required logic in the callback function assigned to your application's **Call.onCallStateChange** for each of the possible call state changes relating to updates, **wsc.CALLSTATE.UPDATED** and **wsc.CALLSTATE.UPDATE_FAILED**.

– Save any data specific to your application.

Reconnecting Dropped Calls

At times, a drop in reception quality or some other event may cause a call that is in progress to be momentarily dropped and reconnected. When a call has been recovered, the WebRTC Session Controller JavaScript API library invokes your application's **CallPackage.onResurrect** event handler with the rehydrated call as the parameter. Your application can handle this scenario by providing the logic in the callback function assigned to the **CallPackage.onResurrect** event handler to use the rehydrated call object and resume the call.

Important: If you create a custom call package, be sure to implement the appropriate logic to resume your application operation and reconnect calls.

To reconnect the call, do the following in your application:

1. If *callPackage* is the name of your application's **CallPackage** object, add the following statement to assign a callback function to its **onResurrect** event handler:

   ```
   callPackage.onResurrect = onResurrect;
   ```

2. Set up the callback function (*onResurrect* in this case).

   In this callback function, be sure to resume the call after you perform any necessary actions. For example,

   ```
   function onResurrect(resurrectedCall) {
       ...
       resurrectedCall.resume(onResumeCallSuccess, doCallError);
   }
   ```

3. Set up the *onResumeCallsuccess* success callback for the **Call.resume** method.

   For example,

   ```
   function onResumeCallSuccess(callObj) {
       // Is the call in an established state?
       if (callObj.getCallState().state == wsc.CALLSTATE.ESTABLISHED) {
           // Call is in established state. Take action.
           ...
       } else {
           // Call is not in established state. Take action.
           ...
       }
   }
   ```

   The *doCallError* callback function should have been defined earlier when the application's **Call** object was created.

5

Setting Up Video Calls in Your Applications

This chapter shows how you can use the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library to enable your applications users to make and receive video calls from your applications, when your applications run on WebRTC-enabled browsers.

> **Note:** See *WebRTC Session Controller JavaScript API Reference* for more information on the individual WebRTC Session Controller JavaScript API classes.

About Implementing the Video Call Feature in Your Applications

The WebRTC Session Controller JavaScript API associated with video calls enables your web applications to support video calls made to and received from other WebRTC-enabled browsers and Session Initiation Protocol (SIP)-based applications.

To support the video call feature in your application, update the application logic you used to set up audio calls in the following way:

- Setting up the `<video>` element for the video stream to display optimally on the application page.
- Enabling a user to make or receive a video call.
- Monitoring the video display for the duration of the video call.
- Adjusting the display element when the video call ends.

About the WebRTC Session Controller JavaScript API Used in Implementing Video Calls

The WebRTC Session Controller JavaScript API objects and methods you use in implementing video calls are the same API objects you would use to implement the audio call feature in your applications. See "About the WebRTC Session Controller JavaScript API Used in Implementing Audio Calls". You can extend the video call feature in your application to perform custom tasks by extending these API.

Setting Up Video Calls in Your Applications

You can use WebRTC Session Controller JavaScript API to set up the video call feature in your application to suit your deployment environment. The specific logic, web application elements, and controls you implement for the video call feature in your

applications are predicated upon how the video call feature is used in your web application.

The logic to set up video calls in your applications is based on the basic logic described in "Overview of Setting Up the Audio Call Feature in Your Application". Supporting video calls becomes a matter of modifying that basic logic to set up, manage, and close video calls using the WebRTC Session Controller JavaScript API library and providing the associated display elements and controls on the application page.

When you have the basic code to place and receive audio calls using the WebRTC Session Controller JavaScript API library, update that application logic by doing the following:

- Setting Up the Video Display
- Specifying the Video Direction in the Call Configuration
- Managing the Video Display on Your Application Page
- Managing the Video Streams in the Media Stream Event Handler
- Provide the associated display elements and controls on the application page as required by your application and its deployment environment.

Setting Up the Video Display

After assessing your browser's support for video, set up the video display settings based on the requirements of your application and the deployment environment.

In Example 5–1, an application sets up the video interface using the attributes of the HTML <video> tag. It uses the width attribute to specify the display area in percentages and the autoplay attribute to specify that the video should start playing as soon as it is ready.

Example 5–1 Sample Video Display Settings

```
</table>
 ...
    <!-- HTML5 audio element. -->
    <tr>
        <td width="15%"><video id="selfVideo" autoplay></video></td>
        <td width="15%"><video id="remoteAudio" autoplay></video></td>
</tr>
</table>
```

Specifying the Video Direction in the Call Configuration

The WebRTC Session Controller JavaScript API library provides the **videoMediaDirection** parameter to specify the video capability for calls in the **CallConfig** class object.

Enable the video stream in your application when you create the **CallConfig** object by setting the video media direction variable (**videoMediaDirection**). See "Setting Up the Configuration for Calls Supported by the Application".

In Example 5–2, an application enables the user to send and receive video objects by setting the video media direction variable to **wsc.MEDIADIRECTION.SENDRECV** when it creates its **CallConfig** object.

Example 5–2 Call Configuration Updated to Include Video

```
// Create a CallConfig object.
var audioMediaDirection = wsc.MEDIADIRECTION.SENDRECV;
var videoMediaDirection = wsc.MEDIADIRECTION.SENDRECV;
var callConfig = new wsc.CallConfig(audioMediaDirection, videoMediaDirection);
console.log("Created CallConfig with video stream.");
console.log(" ");
```

Managing the Video Display on Your Application Page

Set up the video to display or be hidden as required by your application and your deployment environment. One way to manage your application page optimally would be to enable the video element in your application when the call is in the required state and not otherwise. When your application deals with a new state in the call, specify the hidden attribute for the media element and set it to the required display state of the video media.

In Example 5–3, an application has a callback function called *callStateChangeHandler* assigned to its **Call.onCallStateChange** event handler. The application uses this callback function to manage the video display based on the call state changes. The application sets the media.hidden value to:

- **false** when the call is established
- **true** for all other call states

Example 5–3 Including Video Display State

```
function callStateChangeHandler(callObj, callState) {
    console.log (" In callStateChangeHandler().");
    console.log("callstate : " + JSON.stringify(callState));
    if (callState.state == wsc.CALLSTATE.ESTABLISHED) {
        console.log (" Call is established. Calling callMonitor. ");
        console.log (" ");
        callMonitor(callObj);
        media.hidden = false;
    } else if (callState.state == wsc.CALLSTATE.ENDED) {
        console.log (" Call ended. Displaying controls again.");
        console.log (" ");
        displayInitialControls();
        media.hidden = true;
    } else if (callState.state == wsc.CALLSTATE.FAILED) {
        console.log (" Call failed. Displaying controls again.");
        console.log (" ");
        displayInitialControls();
        media.hidden = true;
    }
}
```

Managing the Video Streams in the Media Stream Event Handler

When the media state changes, the WebRTC session Controller JavaScript API library invokes the event handler you assigned to **Call.onMediaStreamEvent** in your application and provides it with the new media state. Use this new state to take action on the media stream, attaching or removing it as required.

In Example 5–4, an application has a callback function called *mediaStreamEventHandler* assigned to its **Call.onMediaStreamEvent** event handler. The application uses this callback function to manage the video media stream based on the value in *mediaState*,

the new media state the application receives from the WebRTC session Controller JavaScript API library. The callback function retrieves the appropriate video element from *document*, the Document Object Model (DOM) object and attaches the stream to that video element, using the WebRTC **attachmediastream** function.

Example 5–4 Attaching Video Streams in the Media Stream Event Handler

```
// Attach media stream to HTML5 audio element.
function mediaStreamEventHandler(mediaState, stream) {
    console.log (" In mediaStreamEventHandler.");
    console.log("mediastate : " + mediaState);
    console.log (" ");

    if (mediaState == wsc.MEDIASTREAMEVENT.LOCAL_STREAM_ADDED) {
        attachMediaStream(document.getElementById("selfVideo"), stream);
    } else if (mediaState == wsc.MEDIASTREAMEVENT.REMOTE_STREAM_ADDED) {
        attachMediaStream(document.getElementById("remoteVideo"), stream);
    }
}
```

6

Setting Up Data Transfers in Your Applications

This chapter shows how you can use the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library to send and receive data over the data channel established in calls from your applications, when your applications run on WebRTC-enabled browsers.

> **Note:** See *WebRTC Session Controller JavaScript API Reference* for more information on the individual WebRTC Session Controller JavaScript API classes.

About Data Transfers and Signaling Engine

The WebRTC Session Controller JavaScript data transfer API sets up peer to peer data channels based on the WebRTC data channel definition and manages the workflow of the message exchanges such as chat sessions in web applications.

The data being transferred could be a raw data object such as a binary large object (BLOB), DOMString, ArrayBuffer, ArrayBufferView. The WebRTC Session Controller JavaScript API library manages the data transfer only. It does not access the contents of the data object that it transfers.

This chapter describes how you can use the WebRTC Session Controller JavaScript API library to set up and manage call sessions that support data transfers by managing the flow of the data element in the communication, detecting changes in the data flow state, and responding accordingly to the changes.

About Setting Up Data Transfers in Your Applications

The WebRTC Session Controller JavaScript API related to data transfers support text-based communications such as text messaging and chat sessions when data channels are configured in calls connecting browser phones located on web applications hosted at other WebRTC-enabled browsers.

To support data transfers, do the following in your application:

- Set up the required user interface elements, such as for the chat session to display optimally on the application page.
- Enable users to make or receive data transfers in calls.
- Manage calls with data transfers by doing the following:
  - Monitoring the state of the data channel.
  - Handling the incoming data; and displaying it, if necessary.

  - Sending the data object provided by the user.
- Adjust the display elements when the call with data transfer ends.

About the API Used to Manage the Transfer of Data

The following WebRTC Session Controller JavaScript API classes are used to manage the transfer of data in calls made from or received by your web application:

- The **dataChannelConfigs** parameter associated with the **CallConfig** class
- The **CallPackage.onIncomingCall** event handler
- The **Call.onDataTransfer** event handler
- The data transfer object, **wsc.DataTransfer**. See "Managing Data Channels Using wsc.DataTransfer".
- The data sender object, **wsc.DataSender**. See "Sending Data Using wse.DataSender".
- The data receiver object, **wsc.DataReceiver**. See "Handling Incoming Data Using wsc.DataReceiver".

Managing Data Channels Using wsc.DataTransfer

Set up an instance of the data transfer object, **wsc.DataTransfer**, to manage data channels between two peers. Table 6–1 lists its states.

Table 6–1 Data Transfer States

Setting for State	Description
none	The **DataTransfer** object has been created but no data channel has been established or is initializing.
starting	The data channel of **DataTransfer** object is initializing or is in negotiation to be established.
open	The data channel of **DataTransfer** object is established. The data channel is ready to send or receive data and data transfers can take place.
closed	The data channel of the **DataTransfer** object is closed.

Obtain information on the state of the data transfer object using the following:

- **DataTransfer.getReceiver()**

  This method returns data on the receiver as an instance of the **DataReceiver** class.

- **DataTransfer.onOpen**

  This event handler is invoked when the data channel of the **DataTransfer** object is open. Data can be sent over or received from the data channel. Assign and define a callback function to this event handler.

- **DataTransfer.onClose**

  This event handler is invoked when the data channel of the **DataTransfer** object is closed. Data cannot be transferred. Assign and define a callback function to this event handler.

- **DataTransfer.onError**

This event handler is invoked when the data channel of the **DataTransfer** object has an error. Assign and define a callback function to this event handler.

See *WebRTC Session Controller JavaScript API Reference* for more information on the **wsc.DataTransfer** class.

Sending Data Using wse.DataSender

Use the **wsc.DataSender** class object to send raw data.

Obtain the **DataSender** object from the **DataTransfer** object by calling the application's **DataTransfer.getSender** method. This method returns the **DataSender** object to your application.

Use the **DataSender.send** method to send raw data such as a text string or a BLOB using the data channel in the data transfer object.

Handling Incoming Data Using wsc.DataReceiver

Use the **wsc.DataReciever** class object to handle incoming raw data.

Obtain the **DataReceiver** object from the **DataTransfer** object by calling the application's **DataTransfer.getReceiver** method. This method returns the **DataReceiver** object to your application.

Assign and define a callback function to the **DataReceiver.onMessage** event handler. This event handler is called when a raw data object is received by the data channel of the **DataTransfer** object.

In the following example, an application retrieves an instance of the **DataTransfer** class in *receiver*. The callback function assigned to the **onMessage** event handler of the *receiver* object processes the incoming data.

```
onDCOpen = function(){
    // Set up the receiver object
    receiver = dataTransfer.getReceiver();
    if(receiver){
        receiver.onMessage = function (evt){
        // Retrieve the data and assign it.
        var rcvdDataElm = document.getElementById("rcvData");
        rcvdDataElm.value = evt.data
    }
}
```

See "Sample Event Handler Invoked When the Data Channel is Open" for more information.

The next section uses a chat session to show how the WebRTC Session Controller JavaScript API library can be used to support data transfers in your web applications.

Setting up Data Transfers in Your Application

You can use the WebRTC Session Controller JavaScript API library to set up data transfers in your application to suit your deployment environment. The specific logic, web application elements, and controls you implement for calls with data transfers in your applications are predicated upon how the data transfer feature is used in your web application.

The logic to set up data transfers in calls is based on the basic logic described in "Overview of Setting Up the Audio Call Feature in Your Application". Supporting data transfers in calls becomes a matter of modifying that basic logic to set up, manage, and

close data channels using the WebRTC Session Controller JavaScript API library and providing the associated display elements and controls on the application page.

If the callee is not available, you can implement additional logic in your application to store the incoming data transfer (such as a text message) and provide a notification for the receiver. See "Setting Up Message Alert Notifications".

To support video calls in your applications, augment your application's audio call logic by implementing the following logic specific to data transfers:

- Declaring Variables Specific to the Chat Sessions
- Setting Up the Configuration for Data Transfers in Chat Sessions
- Assigning the Data Transfer Event Handler to the Call Package
- Obtaining the Callee Information
- Starting the Call with the Data Transfer Feature in the Call
- Responding to Your User's Actions on an Incoming Call
- Setting Up the Chat Session User Interface
- Setting Up the Data Transfer State Event Handler for the Chat Session
- Managing the Flow of Data
- Monitoring the Chat Session

Declaring Variables Specific to the Chat Sessions

For each data channel configured in your application's **CallConfig** object, your application needs a **wsc.DataTransfer** class object. And each **wsc.DataTransfer** class object is associated with a **wsc.DataSender** and a **wsc.DataReceiver** class object. Declare the variables necessary for the data channels supported by the calls made from or received by your application.

In Example 6–1, an application declares these objects at the start of the application.

Example 6–1 Sample Data Transfer Variables

```
var dataTransfer, sender, receiver;
var target, buddy1, buddy2;
```

Setting Up the Configuration for Data Transfers in Chat Sessions

The WebRTC Session Controller JavaScript API library provides the **dataChannelConfigs** parameter to define the data channel for calls in the **CallConfig** class object. This **dataChannelConfigs** parameter is an array of JavaScript Object Notation (JSON) objects that describe the configuration of the data channel.

In order to define just the data channel in the call configuration for the application, input the data channel capability in the **dataChannelConfigs** parameter when you create the **CallConfig** object in your application.

In Example 6–2, an application enables the user to send and receive data by setting the data channel capability in the **dataChannelConfigs** object when it creates its **CallConfig** object:

Example 6–2 Sample Call Configuration Object for Data Transfers

```
// create a CallConfig object.
 var dtConfigs = new Array();
```

```
dtConfigs[0] = {"label":"ChatOverDataChannel", "reliable" : false };
var callConfig = new wsc.CallConfig(null,null,dtConfigs);
```

Assigning the Data Transfer Event Handler to the Call Package

When the WebRTC Session Controller JavaScript API library receives data transfer object for the user, it invokes the **CallPackage.onIncomingCall** event handler in your application. Assign a callback function to handle the data transfer object received by your application.

Alternatively, you can assign a single callback function to your application's **CallPackage.onIncomingCall** event handler and within that callback function implement the logic to handle the incoming audio, video calls or data transfers.

In Example 6–3, an application creates its **CallPackage** object within a callback function called *sessionSuccessHandler* which is called when the application **Session** object is created. In the *sessionSuccessHandler* function, the application assigns a callback function named *onIncomingDataTransferCall* to its **Call.onIncomingCall** event handler.

Example 6–3 Sample sessionSuccessHandler for Data Transfers

```
function sessionSuccessHandler() {
    // Create the CallPackage.
    callPackage = new wsc.CallPackage(wseSession);
    // Bind event handler of incoming call.
    if(callPackage){
      callPackage.onIncomingCall = onIncomingDataTransferCall;
    }
    // Other application-specific logic.
    ...
  }
```

See Example 6–6 for a description of the *onIncomingDataTransferCall* function.

Obtaining the Callee Information

Define the user interface to enable the caller to input the callee ID of the person with whom the chat session is to be established. And provide the underlying logic for the appropriate functions to be called when the caller enters text or selects the control buttons. You need to set up the function that will start the chat session.

In Example 6–4, an application calls a function named *displayInitialControls* to provide the user interface and controls for calls with data transfers. As with the sample audio call application, this application calls the *displayInitialControls* function when the session state is CONNECTED.

In this function:

- An input field is provided for the callee ID along with the *Start a Chat Session* control button.
- The onclick= action for the *Start a Chat Session* control button triggers a function called *startDataTransfer* to start the setup for the chat session. In addition, it also defines other functions to invoke when the user selects to cancel or log out.

Example 6–4 Sample Code to Receive Callee Information

```
function displayInitialControls() {
    ...
    var controls = "Enter the Name of Your Chat Buddy: <input type='text' name='dataTarget'
id='dataTarget'/><br>"
    + "<input type='button' name='startDataTransfer' id='startDataTransfer' value='Start a Chat
Session ' onclick='startDataTransfer()'/><br><br>"
    + "<input type='button' name='cancelButton' id='cancelButton' value='Cancel Chat' onclick=''
disabled ='false'/>"
    + "<br><br><br>"
    + "<input type='button' name='logoutButton' id='logoutButton' value='Logout'
onclick='logout()'/>"
    + "<hr>";

    setControls(controls);
    // Verify the input is not blank or invalid number..
    ...
   ...
}
```

Starting the Call with the Data Transfer Feature in the Call

When you receive the callee information, you need to start the setup for the call by creating the **Call** object, implementing the logic to handle events associated with the **Call** object, invoking **Call.Start**, and setting up the required user interface and controls for the chat session.

In Example 6–5, an application uses a function called *startDataTransfer* to perform these actions. The basic logic is similar to the *onCallSomeOne* function used in the sample audio call application. See "Starting a Call From Your Application". The application invokes *startDataTransfer* when it receives the callee information and the user's request to start a chat session.

Example 6–5 Sample startDataTransfer Function

```
function startDataTransfer() {
   // Store the caller and callee names.
   ...
   // Check to see if the user gave a valid input. Omitted here.
   ...
   // Create the  call object.
   var call = callPackage.createCall(target, callConfig, doCallError);
   // Set up the call object's components.
   if (call != null) {

       //Call object is valid. Call the required event handlers.
       setEventHandlers(call);
       ....

       //Set the event handler to call when a data transfer object is created.
       call.onDataTransfer = onDataTransfer;

       // Then start the call.
       call.start();

       // Allow the user to cancel call before it is set up.
       // Disable "Start a Chat Session" button and enable "Cancel" button.
       // If user clicks Cancel, call end() for the call object.
       // Call displayInitialControls() to display the initial input fields.
       ....
    }
```

```
}
```

Responding to Your User's Actions on an Incoming Call

When a user who is logged in to your application receives an in-browser call from another user, the WebRTC Session Controller JavaScript API library invokes the **CallPackage.onIncomingCall** event handler in your application. It sends the incoming call object and the call configuration for that incoming call object as parameters to your application's **CallPackage.onIncomingCall** event handler.

Define the actions to the incoming call in the callback function assigned to your application's **CallPackage.onIncomingCall** event handler in the following way:

- Provide the user interface and logic necessary for the callee to accept or decline the call.
- Provide logic for the following events:
  - User accepts the call. Run the **accept** method for the incoming call object. This will return the success response to the caller.
  - User declines the call. Run the **decline** method for the incoming call object. This will return the failure response to the caller.
- Set up the user interface for the data transfers in the call.
- Assign the callback functions to the event handlers of the incoming call object. These should already have been defined.

In Example 6–6, an application uses the *onIncomingDataTransferCall* callback function assigned to its **Call.onDataTransfer** event handler. The basic logic is similar to the *onCallSomeOne* function used in the sample audio call application. See "Responding to Your User's Actions on an Incoming Call". The application uses the incoming call object (*dtCall*) and the call configuration for that incoming call object (*callConfig*):

Example 6–6 Sample onIncomingDataTransferCall Function

```
function onIncomingDataTransferCall(dTCall, callConfig) {

    //assign the event handler onDataTransfer to the call object
    dTCall.onDataTransfer = onDataTransfer;

    var dElement = document.getElementById("dataTarget");

    // We need the user's response.
    //Display an interface that lets a user decline or accept a call
    // Attach event handlers to these events.
    ...
    document.getElementById("acceptDTBtn").onclick = function() {
        // Chat session accepted
        dTCall.accept(callConfig, null);

        // At this point, update the user interface for the callee
        // Display the fields, controls for text and ending the chat session.
        ...
    }
    document.getElementById("declineDTBtn").onclick = function() {
        dTCall.decline(null);
    }
    setEventHandlers(dTCall);
    call = dTCall;
```

```
}
```

Setting Up the Chat Session User Interface

The design of your application's page determines the type of user interface for the chat session. Use your application's **DataSender** object and its **send** method to send the data entered by the user. Assign callback functions for the controls you use in the interface and set up the actions within those callback functions.

Setting Up the Data Transfer State Event Handler for the Chat Session

The **DataTransfer** class object contains three event handlers:

- **onClose** which indicates that the data channel of a **DataTransfer** object is closed.
- **onOpen** which indicates that the data channel of a **DataTransfer** object is open.
- **onError** which indicates that the data channel of a **DataTransfer** object is in error.

Assign the callback function to handle each of the events and provide the logic for each callback function.

In Example 6–7, an application shows the callback function *onDataTransfer* which was assigned to **Call.onDataTransfer** event handler. In this *onDataTransfer* function, the application assigns a callback function to each of the **onOpen**, **onClose** and **onError** event handlers of the data transfer object.

Example 6–7 Sample onDataTransfer Callback Function

```
onDataTransfer = function(dT) {
    dataTransfer = dT;
    dataTransfer.onOpen = onDCOpen;
    dataTransfer.onError = onDCError;
    dataTransfer.onClose = onDCClose;
};
```

Managing the Flow of Data

To maintain and manage the flow of data in the data channel, implement the logic in the callback functions to handle the **Open**, **Close**, and **Error** states of the data transfer object.

To do so, provide the logic required to handle the following in your application:

1. The **Open** state for the data channel. See "Handling the Open State of the Data Channel".
2. The received text. See "Handling the Received Text".
3. Sending the text entered in the text field. See "Sending the Text".
4. The **Close** state for the data channel. See "Handling the Closed State of the Data Channel".
5. The **Error** state for the data channel. See "Handling Errors Related to Data Transfers".

Handling the Open State of the Data Channel

When the data channel is open, your application can do the following:

- Retrieve its **Datasender** and **DataReceiver** objects from its **DataTransfer** object.

  The **DataSender** and **DataReceiver** objects are returned when you call your application's **DataTransfer.getSender** and **DataTransfer.getReceiver** methods. For the receiver, the data is from the remote peer of the current data channel session.

- For the receiver of the data, retrieve the data from the remote peer of current data channel session, and display it.

- Other actions as necessary. Your application sets up the logic necessary to save the incoming and outgoing text to display the chat session appropriately.

Provide the logic for your application's **DataTransfer.onOpen** event handler as required by your application.

In Example 6–8, when the data channel is open in an application, the callback function assigned as the event handler assigned to the application's **DataTransfer.onOpen** event handler processes the data transfer object.

Example 6–8 Sample Event Handler Invoked When the Data Channel is Open

```
onDCOpen = function(){
    // Set up the receiver object
    receiver = dataTransfer.getReceiver();
    if(receiver){
        receiver.onMessage = function (evt){
            // Retrieve the data and assign it.
            var rcvdDataElm = document.getElementById("rcvData");
            rcvdDataElm.value = evt.data;
        }
    }

    // Retrieve the sender
    sender = dataTransfer.getSender();

    var dcReadyState = dataTransfer.state;

    // Set up the control buttons appropriately
    var sendDataBtn = document.getElementById("sendData");
    sendDataBtn.hidden = false;
    var dataForChannel = document.getElementById("dataForChannel");
    dataForChannel.hidden = false;
    dataForChannel.value="";
    var endButton = document.getElementById("endDataChannel");
    endButton.hidden = false;
    var rcvdDataElm = document.getElementById("rcvData");
    rcvdDataElm.value = "";
    rcvdDataElm.hidden = false;

    var acceptBtn = document.getElementById("acceptDTBtn");
    if(acceptBtn){
        acceptBtn.hidden = true;
        var declineBtn = document.getElementById("declineDTBtn");
        declineBtn.hidden = true;
    }
}
```

Handling the Received Text

Set up the logic to handle the text that is received by your application user.

In Example 6–9, an application uses a utility function named *onReceiveDTMsg* to handle the received text.

Example 6–9 Sample Function to Assign Received Text

```
onReceiveDTMsg = function(data) {
    var rcvdDataElm = document.getElementById("rcvData");
    rcvdDataElm.value = data;
}
```

Sending the Text

When the user enters text and clicks the control to send the text, provide the logic to send the text entered in the text field using your application's **DataSender.send** method.

In Example 6–10, an application uses a utility function named *send* to retrieve the data from the Document Object Model (DOM) object. It calls the method of the application's **DataSender.send** method with this data.

Example 6–10 Sample Send Function

```
 send = function(){
    // get the data from the text field
    var data = document.getElementById("dataForChannel").value;
    if(sender) {
        sender.send(data);
        document.getElementById("dataForChannel").value = "";
    } else {
       console.log("sender is null");
    }
}
```

Handling the Closed State of the Data Channel

Set up the logic to handle the closed state of the data channel.

Example 6–11 shows the function expression for *onDCClose* to handle the **Close** state for the data channel.

Example 6–11 Sample Event Handler Invoked When the Data Channel is Closed

```
onDCClose = function(){
    var dcReadyState = dataTransfer.state;
}
```

Monitoring the Chat Session

Update the logic used to monitor the call by providing some way for both parties to end the chat session.

In the audio call application described in "Setting Up Audio Calls in Your Applications", a function named *callMonitor* is called by the callback function assigned to the application's **Call.onCallStateChange** event handler. As shown in Example 4–11, the *callMonitor* function is called when the call state is **WSC.CALLSTATE.ESTABLISHED**.

At this point, do the following in your application:

- Display the user interface necessary for the chat session.
- Provide the logic for the actions to take when the chat session ends.

> **Note:** A chat session can be ended by the caller or the callee.

In Example 6–12, an application does the following:

- Displays the user interface for the chat session.
- Responds to the selection:
  - If *End Chat Session* is clicked, ends the call (which ends the chat session and releases the call resources).
  - If *Logout* is selected, ends the session (which releases the session's resources).

Example 6–12 Monitoring the Chat Session

```
function callMonitor(callObj) {

    // Draw up the user interface for the callee.
    ...

    //set the button of ending a dataTransfer call
    var endBtn = document.getElementById("endDataChannel");
    if (endBtn){
       //set the event handler when clicking the end button
       endBtn.onclick = function() {
           if(dataTransfer != null) {
               // There is some data.
               // This function merely sets the text to blank.
               document.getElementById("dataForChannel").value = "";
           }

           // End the data channel call.
           callObj.end();
        };
    }
}
```

7

Setting Up Message Alert Notifications

This chapter shows how you can use the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library to enable your application users to subscribe to and receive message alert notifications from your applications.

> **Note:** See *WebRTC Session Controller JavaScript API Reference* for more information on the individual WebRTC Session Controller JavaScript API classes.

About Message Alert Notifications and Signaling Engine

Message alert notifications consist of text, pager, fax, voice, and multimedia message notifications that are useful ways to enable users to access and retrieve such communication at a later time. These messages could be stored on designated message servers for a configurable time to be accessed by the respective recipients.

You can use WebRTC Session Controller JavaScript API to enable your application users to subscribe to notifications.

> **Note:**
>
> - The web user interface aspects required to enable the user to subscribe to or to retrieve notifications are beyond the scope of this document.
> - All other aspects of a stored message service system such as creating the message, storing it, accessing the message server, retrieving the message, and forwarding, storing or destroying the message are dependent on the environment where your application is deployed.

Handling Message Notifications in Your Web Applications

To handle message alert notifications, the logic in your application must accomplish the following tasks:

- Enable a user to subscribe to receiving notifications.

  To do so, in your application:

  – Provide the interface elements necessary for the user to specify the service target.

  - Set up the subscription when the service target is received from the subscriber.
- Enable the user to access and process the received notifications.

  To do so, in your application:

  - Set up the elements necessary to receive the incoming notification.
  - Process the information and display it for the user.
  - Set up the logic to respond to the user's actions on the subscriptions.
- Respond to the end of a subscription resulting from the following events:
  - The user stops the current subscription to notifications.
  - The provider of the notification ends the notifications to this subscriber. The WebRTC Session Controller JavaScript API library invokes the event handler which indicates to your application that the subscription has ended.

About the API Used to Manage Message Alert Notifications

In addition to the general WebRTC Session Controller JavaScript API objects, the following API objects are used to manage message alert notifications in your applications:

- **wsc.MessageAlertPackage** to enable message alert notifications. See "Managing Message Alert Notifications with wsc.MessageAlertPackage".
- **wsc.Notification** to manage notifications. See "Handling Notifications with wsc.Notification".
- **wsc.Subscription** to manage subscriptions. See "Subscribing to Notifications with wsc.Subscription".
- **wsc.MessageSummary** to obtain message summary information. See "Getting Message Summary Information".
- **wsc.MessageCounts** to obtain the number of messages by the type of message. See "Retrieving Message Counts from Message-Summary Notifications".

See "Extending Your Applications Using WebRTC Session Controller JavaScript API" for information on extending these objects.

Managing Message Alert Notifications with wsc.MessageAlertPackage

Manage messaging alert notifications for pending voice mails, fax messages, and so on during the specified session with an instance of the **wsc.MessageAlertPackage** class. This object enables message alert notification applications. Use it to create new subscriptions for notifications, manage active subscriptions, and handle received message notifications. When you use **wsc.MessageAlertPackage**, the WebRTC Session Controller JavaScript API library handles the messaging flow for the notifications.

Create an instance of **wsc.MessageAlertPackage**, such as *messageAlertPackage*, using your application's **Session** object. You can create an array of subscriptions in your application and use the *messageAlertPackage* object to manage the user's alert and message notifications for each subscription.

If the web page reloads, set up the logic to restore failed subscriptions. To do so, assign a callback function to your application's **MessageAlertPackage.onResurrect** event handler. The WebRTC Session Controller JavaScript API library provides the rehydrated subscription data as a parameter to the callback function assigned to your application's **MessageAlertPackage.onResurrect** event handler.

See "Extending and Overriding WebRTC Session Controller JavaScript API Object Methods" for more information on extending the **MessageAlertPackage** API.

Handling Notifications with wsc.Notification

Use the **wsc.Notification** class to obtain the information associated with a notification, such as the identity of the sender, the content of the notification, and the identity of the receiver.

If your application receives a message notification for a subscription, the WebRTC Session Controller JavaScript API library provides the notification when it invokes the **onNotification** event handler for your application's **Subscription** object.

Inspect the incoming notification in the callback function you assigned to your application's **Subscription.onNotification** event handler to process the information using:

- The **wsc.MessageSummary** class, derived from the **wsc.Notification** class.

  It holds the message summary of the incoming notification.

- The **wsc.MessageCounts** class which holds the number of new and old messages retrieved from the message summary, grouped as regular or urgent.

You can retrieve the following information:

- The number of a specific type of message, such as the number of new and/or old voice message messages in your application's **MessageSummary** object.

  To retrieve the number of a specific type of message, call the **getMessageCounts** method of your application's **MessageSummary** object and input the type of message as *msgClassType*.

- The message content in the incoming notification, by using the **getContent** method of your application's **Notification** object. The message content is returned as a JavaScript Object Notation (JSON) object.

- The identity of the receiver of the incoming notification, by using the **getReceiver** method of your application's **Notification** object.

- The identity of the sender of the incoming notification, by using the **getSender** method of your application's **Notification** object.

Subscribing to Notifications with wsc.Subscription

The **wsc.Subscription** class can be used to enable your application users to subscribe to notifications.

When your application user provides the service target such as *voice_mail@example.com*, you can create a **Subscription** object (for example, *subscription*) using the **MessageAlertPackage.createNewSubscription** method. The service target *voice_mail@example.com* represents a service in the telecommunication network that can send message alert notifications for such a subscription. Note that, WebRTC Session Controller must be configured in order to route the subscription requests to such a service.

Manage subscriptions by providing logic for the following:

- The **Subscription.onNotification** event handler

  Assign and set up the callback function for your application's **Subscription.onNotification** event handler to handle incoming message notifications for the current subscription.

- The **Subscription.onEnd** event handler

  Assign and set up the callback function for your application's **Subscription.onEnd** event handler to handle the end message for a subscription.

- The validity of a subscription

  Use the **Subscription.isValid** method to check and take action based on the validity of the current subscription.

- The ending of the current subscription

  Use the **Subscription.end** method to stop the current subscription and end message notifications for it.

Getting Message Summary Information

The **wsc.MessageSummary** class is extended from **wsc.Notification**. When the WebRTC Session Controller JavaScript API library receives a notification whose event type is **message-summary**, the **MessageAlertPackage** API creates an instance of the **MessageSummary** object. It provides the **MessageSummary** object as input to the callback function you assigned to your application's **Subscription.onNotification** event handler.

In the callback function, you can use the following methods:

- **MessageSummary.getMessageAccount**

  Use the **MessageSummary.getMessageAccount** method to retrieve the message account for the current message summary notification in a String format.

- **MessageSummary.getMessageCounts(***msgClassType***)**

  where *msgClassType* represents the message class type. The count of the number of *msgClassType* messages is returned to your application in an instance of the **wsc.MessageCounts** object. Use the methods of **wsc.MessageCounts** to get more details on the notification.

  For example, input the string "voice_message" or "fax_summary" to retrieve the count of the number of voice or fax messages for this subscription. This message class type corresponds to the Session Initiation Protocol (SIP) notification message class type.

- **MessageSummary.isMessageWaiting**

  If there is a message waiting in the incoming notification, the **MessageSummary.isMessageWaiting** method returns the boolean value **true**.

Retrieving Message Counts from Message-Summary Notifications

The **wsc.MessageCounts** class contains the number of *msgClassType* messages in a message-summary type of notification.

To obtain the message count when your application receives a message-summary notification as a **MessageSummary** object (for example, *msgSummary*, do the following in your application:

1. Retrieve the message count by invoking the **MessageSummary.getMessageCounts** method.

   This method returns the message counts as an instance of **wsc.MessageCounts**, for example, *msgCounts*.

2. Retrieve the messages by age and urgency. Use:

- **MessageCounts.getUrgentNew** to retrieve the urgent messages that are new
- **MessageCounts.getNew** to retrieve the normal messages that are new
- **MessageCounts.getUrgentOld** to retrieve the urgent messages that are old
- **MessageCounts.getOld** to retrieve the normal messages that are old

3. Provide the information to the user as required.

In Example 7–1, an application uses a callback function called *onNotify* to process an incoming notification called *incomingNotification*. If the instance of the **MessageCounts** object retrieved from *incomingNotification* is not null, the function retrieves the number of normal and urgent messages that are new in *newnormal* and *newurgent*.

Example 7–1 Obtaining Number of Messages by Type

```
function onNotify(incomingNotification) {
    var msgCount = incomingNotification.getMessageCounts("voice_message");
     if(msgCount != null){
         // Deal with the New and Old normal and Urgent messages
         var newurgent = msgCount.getNewUrgent();
         var newnormal = msgCount.getNew();
    ...
    };
}
```

The application in the above example can then update the display for the device for example, update the audio or visual display for the message-waiting indicator on the browser page.

Managing Subscriptions

Managing user subscriptions to notifications in your applications consists of the following tasks:

- Enabling the User to Subscribe to Notifications
- Setting Up a Subscription
- Handling the Ending of a Subscription
- Restoring a Subscription

Enabling the User to Subscribe to Notifications

To enable your application user to subscribe to notifications:

- Set up your application's message alert notification package using your application's session. In the following example, an application sets up a message alert notification package called *MsgAlertHandler* with reference its application session, *wscSession*.

  ```
  MsgAlertHandler = new wsc.MessageAlertPackage(wscSession);
  ```

- Set up the interface for the user to enter the information on the service target.
- Define the logic in the callback functions to respond to the user's actions.

Setting Up a Subscription

To implement the logic to support subscriptions, your application needs to create a subscription when the user enters a target for a subscription service. The target could be for an identity of a service or an account. It should be in the format **user@domain**. WebRTC Session Controller adds **sip:** to the target from a web subscribe user (for example, **sip:user@domain**), if that target is determined to be a SIP notification application. Your application can then notify the user whenever there is a change in the message status for the account.

Creating a Subscription

Use the **MessageAlertPackage.createNewSubscription** method to create a new subscription. The syntax for the method is:

```
createNewSubscription(target, subscriber, onSuccess, onError, onNotification,
onEnd, extheaders)
```

Where:

- *target* is the service target you obtained from the user, the device or the service the user wishes to monitor.
- *subscriber* is the user identity of this subscriber.
- **onSuccess** is the event handler called when the application creates the subscription.
- **onError** is the event handler called when the application fails to create the subscription.
- **onNotification** is the event handler for a notify message.
- **onEnd** is the event handler called when the provider of the notification notifies Signaling Engine that this subscription has ended.
- *extHeaders* is extension header. Extension headers are inserted into the JSON message.

Example 7–2 Creating a Subscription

```
// Create a message alert package.
msgAlertHandler = new wsc.MessageAlertPackage(wscSession);
    ...
// Create a new subscription for this target.
subscription = msgAlertHandler.createNewSubscription(target,
userIdentity,onSubscribeSuccess,onSubscribeError,onNotify,onEnd);
// Assign the onNotification event handler for the subscription.
 subscription.onNotification = onNotify;
 ...
```

If the application user subscribes to notifications from multiple targets such as phones and voice mail storage devices, set up the subscriptions and store the information on the service targets accordingly. Implement the logic to verify and deliver the incoming notification to the appropriate subscription.

You can optionally include extension headers as the last parameter *extHeaders* when you invoke the **MessageAlertPackage.createNewSubscription** method shown in Example 7–2. The *extHeaders* you input must be in JSON format to be inserted in the outgoing message. Here is an example of an extension header:

```
{'customerKey1':'value1','customerKey2':'value2'}
```

When you include the extension header as the last parameter in the **MessageAlertPackage.createNewSubscription** method, it is placed in the header section of the message in the following way:

```
{ "control" : {}, "header" :
{...,'customerKey1':'value1','customerKey2':'value2'}, "payload" : {}}
```

Verifying that a Subscription is Active

To verify whether a current subscription is active, use the **Subscription.isValid()** method. The function returns **true**, if the subscription is active.

Handling the Ending of a Subscription

Set up the functions to handle the following scenarios:

- The user ends the current subscription.

  Use the **Subscription.end** method to stop the current subscription. If your application uses extension headers, input them when you call this method.

- The WebRTC Session Controller JavaScript API library invokes the **Subscription.onEnd** event handler in your application.

  The WebRTC Session Controller JavaScript API library invokes this event handler when it is notified about the end of that specific subscription for the user. Assign a callback function to the **Subscription.onEnd** event handler. Set up the logic within this function to inform the user appropriately and take action accordingly.

Restoring a Subscription

When a subscription has been recovered, the WebRTC Session Controller JavaScript API library invokes your application's **MessageAlertPackage.onResurrect** event handler with the rehydrated subscription as the parameter. Your application can call the **Subscription.isValid** method and take further action. If the user prefers to end the subscription, use the **Subscription.end** method.

> **Important:** If you create a custom message alert package, be sure to implement the appropriate logic to resume your application operation and restore subscription operations.

In Example 7–3, an application resets the callback functions to the rehydrated subscription object and proceeds with its actions.

Example 7–3 Sample onResurrect Function for a Subscription

```
subscribeHandler = new wsc.MessageAlertPackage(wscSession);
if (subscribeHandler) {
    subscribeHandler.onResurrect = onResurrect;
}
...
function onResurrect(rehydratedSubscription) {
    ...
    // Reset related callback functions
    subscription = rehydratedSubscription;
    subscription.onSuccess = onSubscribeSuccess;
    subscription.onError = onSubscribeError;
    subscription.onNotification = onNotification;
```

```
    subscription.onShutdown = onShutdown;
    // Initialize other parts of the application, such as page
...
}
```

Managing Notifications

Your application needs to set up the logic required to support the message alert notifications for the event types that must be supported in the environment in which your application is deployed.

In order to do so, implement and extend the **wsc.Notification** class to handle the notifications for the required event types. Process the incoming notification to update the information on the message counts (for example, how many new and old) and process it to identify the identity of the person or service providing the notification and subscriber of the service and take action accordingly.

Handling Message Notifications

To handle an incoming notification that is not a message summary, set up the callback function to retrieve the message content and the identities of the sender and receiver:

- **Notification.getContent**

  Use the **Notification.getContent** method to retrieve the contents of the notification as a JSON object.

- **Notification.getSender**

  Use the **Notification.getSender** method to retrieve the identity of the sender (service) of the notification as a string object.

- **Notification.getReceiver**

  Use the **Notification.getRetriever** method to retrieve the identity of the receiver of the notification as a string object.

Set up the logic to retrieve handle the information as necessary.

See "Handling Custom Message Notifications" for information on how to handle custom message notifications in your web applications.

8

Extending Your Applications Using WebRTC Session Controller JavaScript API

This chapter describes how you can extend the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library.

> **Note:** See *WebRTC Session Controller JavaScript API Reference* for more information on the individual WebRTC Session Controller JavaScript API classes.

About the Default Messaging Mechanism Used by Your Applications

When your application needs to perform an action such as creating a session, a call or message alert package, starting a call or responding to a notification, it sends a JavaScript message associated with that action to the WebRTC Session Controller JavaScript API library. For its part, the WebRTC Session Controller JavaScript API library converts these messages (for example, **Call**.**start**) into signaling messages using a protocol based on JavaScript Object Notation (JSON). For more information, see *WebRTC Session Controller Extension Developer's Guide*.

The WebRTC Session Controller JavaScript API library contains classes and methods that have a default behavior and others that can be extended. When you use the default classes and methods, the WebRTC Session Controller JavaScript API library handles all of the signaling messages for all the resulting default commands.

When you need to broaden or extend your application logic, you may need to extend the JavaScript message sent by your application. To do so, use those objects and methods in the WebRTC Session Controller JavaScript API that are extensible.

About Extending the WSC Namespace

Your applications can support additional communication-related services in audio, video, and data transfer flows. For example:

- Custom calls

  In order to handle custom call flows you can implement logic to prepare the calls, setting up the logic to accept prepared calls, and manage the sequence of messages associated with the prepared calls.

- Custom packages

  To support custom services in calls of custom call flows, your application may need to extend the application session.

The WebRTC Session Controller JavaScript API library provides the following objects and functions for this purpose:

- **wsc.extend**. See "Extending Objects Using the wsc.extend Method".
- **wsc.ExtensibleSession**. See "Extending Sessions with wsc.ExtensibleSession Class".
- Other extensible methods. See "Extending and Overriding WebRTC Session Controller JavaScript API Object Methods".

> **Note:** WebRTC Session Controller JavaScript API Reference uses the term "For extensibility" to identify such extensible objects and methods.

Extending Objects Using the wsc.extend Method

The **wsc.extend** method is a utility you use when you wish to extend a WebRTC Session Controller JavaScript API class object exposed through **wsc** namespace. The **wsc.extend** method takes two parameters, *child* and *parent*, in that order. The syntax is:

```
wsc.extend(child, parent);
```

When you call the **wsc.extend** method, the constructor of the child object calls the constructor of the parent. All the members that are attached to the prototype object of the parent entry are copied to the prototype object of the child entry. The objects initialized in the parent's constructor code become available and the child can now make use of the objects in the parent class object. You can then override any function in the child object without impacting the parent.

The code sample in Example 8–1 creates the *wsc.CallExtension* object which extends the **wsc.Call** object.

Example 8–1 Creating CallExtension from the Call Object

```
function CallExtension() {
    //chain constructor
    CallExtension.superclass.constructor.apply(this, arguments)
}

//The following statement makes the CallExtension object a child of wsc.Call
wsc.extend(CallExtension, wsc.Call);
```

At this point, the inherited members of *CallExtension* can be overridden without affecting the corresponding members in the parent *Call* object.

See "Working with Extended Calls" for a description of how the **onMessage** function is overridden in this newly-created *CallExtension* class object.

Extending Sessions with wsc.ExtensibleSession Class

The **wsc.ExtensibleSession** class object provides many critical functions required to extend packages. Use the methods in **wsc.ExtensibleSession** to access and retrieve information about a subsession using its identifier (**sessionId)**, retrieve a specific package by its type and manage it, and configure custom packages in your application to handle specific set of tasks. See "Creating Custom Packages Using the ExtensibleSession Object".

Extending and Overriding WebRTC Session Controller JavaScript API Object Methods

You can override and extend methods in WebRTC Session Controller JavaScript API objects to do the following:

- Handling Extended Call Sessions with CallPackage.onMessage
- Preparing Custom Calls with CallPackage.prepareCall
- Inserting Calls into a Session with CallPackage.putCall
- Processing Custom Messages for a Call with Call.onMessage
- Extending Headers in Call Messages
- Handling Custom Message Notifications
- Handling Extensions to Notifications with MessageAlertPackage.onMessage

Handling Extended Call Sessions with CallPackage.onMessage

If your application logic uses extended call sessions, set up the required actions in a callback function for the application's **CallPackage.onMessage** event handler. When call-related messages come in to your application, this callback function will be invoked enabling you to inspect the incoming message and take further action on the call.

When you extend the **CallPackage** object, you can override the **CallPackage.onMessage** event handler. See "Extending Objects Using the wsc.extend Method" for more information.

Preparing Custom Calls with CallPackage.prepareCall

By default, your application's **Call** object is created with reference to the default **Session** object.

The **CallPackage.prepareCall** method is a Service Provider interface function which prepares a call with reference to a session. For example:

```
mycall = callPackage.prepareCall(mysession,CallConfig, caller, callee);
```

Here, an application has set up the *caller*, *callee*, and *callConfig* objects and uses the **CallPackage.prepareCall** method to prepare a call called *mycall* with reference to a specific session, *mysession*.

Inserting Calls into a Session with CallPackage.putCall

Use the **CallPackage.putCall** method to place a prepared call object in a specific point in the flow for that call session. To do so, you need:

- The subsession Id (*id*) for the call session
- The prepared *call* object.

You can now place the call with the following statement:

```
putCall(id, call);
```

Processing Custom Messages for a Call with Call.onMessage

Process custom message content that your application receives for the current call by using the **Call.onMessage** method. Extend the **Call** object to do so. See "Working with Extended Calls".

Extending Headers in Call Messages

When you use an extension header in a call session, set up the extension header in the following JavaScript format:

```
{'label1':'value1','label2':'value2'}
```

Place the extension header as the last parameter when you invoke the methods that support extension headers. See "Handling Additional Headers" for the complete of objects and methods that support extension headers.

Handling Custom Message Notifications

If the received notification message is not a message summary, your application receives a **wsc.Notification** object as the parameter to its **Subscription.onNotification** event handler.

In the callback function you assign to your application's **Subscription.onNotification** event handler, use this incoming notification to instantiate an extended **wsc.Notification** class object. Use the methods of the extended class object to parse the supported types of notification messages.

Handling Extensions to Notifications with MessageAlertPackage.onMessage

If your application's **MessageAlertPackage** object manages an array of subscriptions, then, when a notification comes to your application, the **MessageAlertPackage.onMessage** event handler is invoked. In the callback function you assign to this event handler, you can process the incoming message notification to identify the subscription object and invoke the appropriate **Subscription.onNotification** event handler for further processing of that notification.

The **MessageAlertPackage.onMessage** function can be overridden to handle custom message events. See "Working with Extended CallPackage Objects".

Handling Additional Headers in Messages

Your application may need to allow users to send or receive additional data in the form of an extra header field.

About Additional Headers in Messages

Some methods in the **Call** and **CallPackage** class objects can accept an additional argument, as long as it is a JSON object. This additional data is sent as an extension header in the message and received as an extra parameter by the event handler of the incoming call.

For example, when a user navigates your application page designed for an auto dealership, your application may have gathered data on the user's preferences for the make, model, and deal preferences, such as *carMaker*, *Convertible*), and *Lease*.

When the user calls the dealership from your page, your application can pass this information to the dealer in the call. Your application sets up this information as a JSON object.

```
{'custprefKey1':'BMW','custprefKey2':'Convertible', 'custprefKey2':'Lease'}
```

This JSON object is now sent in the message as:

```
{ "control" : {}, "header" : {'custprefKey1':'BMW','custprefKey2':'Convertible',
'custprefKey2':'Lease'}, "payload" : {}, }
```

At the dealership, when the dealer receives the call, the appropriate function is invoked with the extra information as the last argument, for example,

```
callObj.accept(callConfig, null, extheader);
```

In that function, your application takes this *extheader* JSON object, retrieves the information and takes the actions necessary to display the information for the dealer's use.

Handling Additional Headers

The WebRTC Session Controller JavaScript API library supports extension headers as the parameter in the following:

- Call Methods:
  - **Call.accept**
  - **Call.decline**
  - **Call.end**
  - **Call.start**
  - **Call.update**
- Event Handlers:
  - **CallPackage.onIncomingCall**
  - **Call.onCallStateChange**
  - **Call.onUpdate**

See the discussion on customizing messages for new Session Initialization Protocol (SIP) or JSON data in *WebRTC Session Controller Extension Developer's Guide*.

Managing Calls with Additional Headers

The **Call** API object can be used to send or receive an extension header in its call flow.

For your application to start a call with extension headers:

1. Set up the extension header as a JSON object.
2. Place the JSON object as the last parameter when your application invokes the outgoing call object's **start** method to start the call.

   For example, if *call* is the call object, *lmedstrm* is the local media stream object, and *extHeader* is the extension header in your application, use the following statement to start the call:

   ```
   call.start(lmedstrm,extHeader)
   ```

When your application receives a request for a call with extension headers:

1. Retrieve the extension header from your application's **CallPackage.onIncomingCall** event handler. The extension header is in JSON format.
2. Perform any actions based on the additional data in the extension header.
3. If the application user accepts the call, do the following:
   a. Set up the extension header as a JSON object.

b. Include the JSON object when your application invokes the outgoing call object's **start** method to start the call.

   For example, if *call* is the call object, *callConfig* the local media stream and data transfer capability for calls, *lmedstrm* is the local media stream object, and *extHeader* is the extension header in your application, use the following statement to accept the call:

   ```
   call.accept(callConfig, lmedstrm, extHeader)
   ```

4. If the application user declines the call, do the following:

   a. Obtain the reason the call was declined by the user.

   b. Set up the extension header as a JSON object.

   c. Include the JSON object when your application invokes the outgoing call object's **decline** method.

      For example, if *extCall* is the extended call object, *reason* the reason the call was declined, and *extHeader* is the extension header:

      ```
      extCall.decline(reason,extHeader)
      ```

Whether your application user is the caller or the callee, your application may need to handle incoming messages and set up outgoing messages associated with the following events. In each case, set up the extension header as described and place it as the last parameter when you invoke the associated function for the extended call.

- Your application user ends the call.

  Set up the extension header as described earlier and include it when you invoke the **end** method for the extended call.

- The call is updated.

  If your application user:

  – Requests the update.

    Set up the extension header as described earlier and include it when you invoke the **update** method for the extended call.

  – Receives the update request.

    Process any data in the extension header in the **onUpdate** event handler. Set up the extension header as described earlier and include it when you invoke the **accept** or **decline** method of the extended call object, as appropriate.

- The call state changes.

  Process any data in the extension header in the **Call.onCallStateChange** event handler. Set up the additional data as the extension header in the method for the outgoing call.

Working with wsc.ExtensibleSession

Your applications can use custom session objects to enable users to subscribe to the presence of other users, chat with SIP users, or set up calls that can be transferred to other users. Configure and manage custom flows in your applications by using the **wsc.ExtendedSession** object.

Creating an Extensible Session in Your Application

To create an extensible session, use the syntax:

```
wsc.ExtensibleSession(useName, webSocketUri, successCallback, failureCallback, sessionId)
```

Where:

- *useName*, is the user name.
- *webSocketUri*, the predefined web socket connection.
- *successCallback*, the function to call if the session object was created successfully.
- *failureCallback*, the function to call if the session object was not created.
- *sessionId*, if you are refreshing an existing session.

Creating Custom Packages Using the ExtensibleSession Object

You can create custom packages in your application to handle specific set of tasks and expand the scope of your application. To add custom packages, your application needs to use the following objects:

- The **ExtensibleSession**:

  Use this object to do one or more of the following

  – Creating an extended session object using the **ExtensibleSession** method of **wsc**.

  – Retrieving all sub-sessions by using the **getAllSubSessions** method of your application's **ExtensibleSession** object.

  – Saving session data to the web browser's **sessionStorage** by using the **saveToStorage** method of your application's **ExtensibleSession** object.

  For more information on subsessions, see *Oracle Communications WebRTC Session Controller JavaScript Extension Developer's Guide*.

- The custom package object in the session:

  Set up this object by doing the following:

  – Creating the custom package object.

  – Registering the package by using the **registerPackage** method of your application's **ExtensibleSession** object.

  – Retrieving the package by its type by using the **getPackage** method of your application's **ExtensibleSession** object.

- The message object that the custom package sends or receives:

  Manage the message object by doing the following:

  – Defining the message object with **wsc.Message**

  – Sending the message using the **sendMessage** method of your application's **ExtensibleSession** object.

  – Handling an incoming message using the **Call.onMessage** method in your application's **ExtensibleSession** object.

  – Generating a correlation ID for the message using the **genNewCorrelationId** method of your application's **ExtensibleSession** object.

The generated correlation ID is based on the current outbound **sequence** number sequence of this session. For information on **sequence**, see *WebRTC Session Controller Extensions Developer Guide*.

- The message flow as required by the requirements of the extended session. See "Sending And Receiving Custom Messages".
- The subsession of the **ExtensibleSession** by:
  - Retrieving the session Id of the subsession using **getSubSessionId** of your application's **ExtensibleSession** object.
  - Retrieving all subsessions that belong to a specific package using **getSubSessionsByPackageType** method of your application's **ExtensibleSession** object.
  - Placing a subsession object into the session with **putSubSession** method of your application's **ExtensibleSession** object.
  - Removing subsession object giving its id using **removeSubSession** method of your application's **ExtensibleSession** object.

Saving Your Custom Session

When you create applications using the default or custom behavior of WebRTC Session Controller JavaScript API, the library automatically saves the data for the sessions.

If you need your application to handle the data associated with custom sessions or subsessions, save the corresponding data in the HTML **SessionStorage** area using the **ExtensibleSession.saveToStorage** method. Your session data should be stored in JSON format. Ensure that your application saves the session data such that it captures the changes so as to maintain the session's current state for use in dealing with connectivity issues.

Important: When your application uses a custom package and/or subsession object save the subsession's state to support rehydration. Monitor the change in the subsession's state and call your application's **ExtensibleSession.saveToStorage()** method to save the data.

Sending And Receiving Custom Messages

When you create messages independent of the default call or message alert package, you need to set up logic to handle the flow of such messages and the resulting actions your application needs to take.

You can send custom messages within a sub-session by providing a **Message** object as an argument when you call your application's **ExtensibleSession#sendMessage** method. Ensure that your application's **Message** object has the control, header, and payload blocks. For example, to send INFO messages as part of an ongoing call, your application can extend its **Call** and **CallPackage** objects and use them to support sending and receiving INFO messages while delegating all other functionality to the existing **Call** and **CallPackage**. See the discussion on extension points in *WebRTC Session Controller JavaScript API Reference*.

About the API Classes Used to Create Custom Message

At times, you may need to create you create custom packages and use custom message flows in your applications. Use the following:

- wsc.Message
- wsc.Message#control
- wsc.Message#header
- wsc.Message#payload

> **Note:** For information on the **wsc.Map** utility you can use when you set up custom messages, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

wsc.Message

The **wsc.Message** class object encapsulates a message and contains two sections of headers and the payload, if necessary. All messages between your application and WebRTC Session Controller are sent in this format. Create the control header, general header and the payload sections of *message* object in your application. Example 8–2 shows the header sections of a message object which initiates a WebSocket connection:

Example 8–2 The Header Sections of a Message Object

```
{
   "control": {
    "type":"request",
    "sequence":"1",
    "version":"1.0"
   },
   "header": {
    "action":"connect",
    "initiator":"bob@someCompany.com",
   }
}
```

> **Note:** When you need to create messages independent of the default call or message alert package, use the **wsc.Message** object and manage the messaging workflow using the **wsc.ExtensibleSession** object. See "Working with wsc.ExtensibleSession".

wsc.Message#control

Use the **wsc.Message#control** object to define the control header in a message.

A control header contains information required for WebSocket reconnection, reliability, timeouts, error, the state of the message, type of the message, and so on. For information on the headers supported in the Control section, see *WebRTC Session Controller JavaScript API Reference* and *WebRTC Session Controller Extension Developer's Guide*.

wsc.Message#header

Use the **wsc.Message#header** object to specify the specific action involved in the message. For example, for a START request, such information would contain who

initiated the request, for whom it is intended, and so on. Your application can add additional headers to the this section. Such headers may be mapped by a gateway server to a SIP header or a parameter. For information on the headers supported in the Header section, see *WebRTC Session Controller JavaScript API Reference* and *WebRTC Session Controller Extension Developer's Guide*.

wsc.Message#payload

Use the **wsc.Message#payload** object to specify the payload section of the protocol specific to the "package". For:

- **CallPackage**, the payload contains the offer or answer in Session Description Protocol (SDP)
- **MessageAlertPackage**, the payload contains the exact message alerts in JSON format.

If you create a "Presence" package, the payload for messages associated with this package should contain the presence information.

Managing Custom Message Data Flows

When you use custom message flows, set up your application with the appropriate logic required to send and receive messages from Signaling Engine. Ensure that the correlation Ids, the sequencing and other details of the outgoing message are appropriate.

Sending a Custom Message to Signaling Engine

Complete the following tasks to send a custom message to Signaling Engine:

- Set up the data as "key" : "value" pairs in
  - **wsc.Message#control()**
  - **wsc.Message#header()**
- Set up the payload using **wsc.Message#payload**
- Use **JSON.stringify** method to set up the message data in *msg*.
- Create the message to be sent using **wsc.Message(***msg***)**, where *msg* is message data.
- Send the message using the **sendMessage** method of your application's **ExtensibleSession** object.
- Monitor the message flow to take further action.
- Save the session and subsession data, as required. See "Saving Your Custom Session".

Processing an Incoming Custom Message

Process a custom message that your application receives from Signaling Engine in the following way:

- Set up the callback function for the **onMessage** event handler of the extended **CallPackage** object in your application. The custom message is provided in the event handler as a **wsc.Message** object.
- Take appropriate action. Set up your application's response in the outgoing message.

- Monitor the message flow to take further action.
- Save the session and subsession data, as required. See "Saving Your Custom Session".

Customizing Your Applications by Extending the Package Objects

This section describes how you can customize your application by extending the WebRTC Session Controller JavaScript API library's default call and message alert package API objects.

Working with Extended CallPackage Objects

Working with extended CallPackage objects involves the following:

- Creating an Extended Call Package
- Registering the Extended Package with the Session
- Extending the Methods and Event Handlers in the Extended Call Package
- Working with Extended Calls

Creating an Extended Call Package

You can create an extended call package when you instantiate a session, such as *wscSession* as shown below:

Example 8–3 Creating an Extended Call Package

```
var CallPackageExtension = function() {

    //sub-class must invoke the superclass's constructor
    CallPackageExtension.superclass.constructor.apply(this, arguments);
};
    CallPackageExtension.prototype.prepareCall = function(wseSession, callConfig,
caller, callee) {
        return new CallExtension(session, callConfig, caller, callee);
    };
wsc.extend(CallPackageExtension, wsc.CallPackage);
```

Registering the Extended Package with the Session

Register the extended call package with the **Session** object you instantiated. For example:

```
// Create a extended CallPackage.

extcallPackage = new CallPackageExtension(wscSession);
```

Call objects created from this extended call package can handle additional headers.

Extending the Methods and Event Handlers in the Extended Call Package

When you extend the call package, extend the required methods and event handlers:

- **prepareCall**
- **putCall**
- **onMessage**

- **onRehydration**. Extend this event handler so that your application can re-create the subsession object based on the rehydrated data your application receives through this event handler. When the subsession object is recreated, WebRTC Session Controller JavaScript API library invokes the **onResurrect** event handler of the call object.

Working with Extended Calls

To work with extended calls:

1. Extend the **CallPackage** object:

   ```
   wsc.extend(CallPackageExtension, wsc.CallPackage);
   ```

2. Create an instance of the extended call package **CallPackageExtension** and register it with the session.

   ```
   CallPackage = new CallPackageExtension(wscSession);
   ```

   Use this instance of the call package to expand the way your application handles calls.

The code sample in Example 8–4 adds support to handle INFO messages as part of a call by extending the **Call** and **CallPackage** objects. If the incoming message has extra data, the **prepareCall** function is overridden.

Example 8–4 Extending the Call and CallPackage Objects

```
//CallExtension is the child object which extends Call object and overrides a function
//The constructor of the child object calls the constructor of the parent so that the objects
//initialized in the parent's constructor code is available to the child.

function CallExtension() {
    //chain constructor
    CallExtension.superclass.constructor.apply(this, arguments)
}

//The following statement makes the CallExtension object as a child of wsc.Call
wsc.extend(CallExtension, wsc.Call);

   //override the method onMessage to support handling INFO messages

CallExtension.prototype.onMessage = function (message) {
    //check if this is an INFO message, if so, handle it here
    if (this.isInfoMessage(message)) {
        handleInfoMessage(message);
    } else {
        // delegate the handling to the base class
        CallExtension.superclass.onMessage.call(this, message)
    }
};

CallExtension.prototype.isInfoMessage = function (message) {
    var action = message.header.action,
    type = message..control.type;
    return action === "info" && type === "message";
};

//Extend and CallPackage object and override the prepareCall function such that
//A CallExtension object is created instead of the default Call object
function CallPackageExtension() {
```

```
    CallPackageExtension.superclass.constructor.apply(this, arguments)
}

wsc.extend(CallPackageExtension, wsc.CallPackage);

//override prepareCall function
CallPackageExtension.prototype.prepareCall = function (session, callConfig, caller, callee) {
    return new CallExtension(session, callConfig, caller, callee);
};
```

Working with Extended MessageAlertPackage Objects

Working with extended CallPackage objects involves the following:

- Creating an extended message alert package. The process is similar to creating an extended call package. See "Creating an Extended Call Package".
- Registering the extended package with the session. The process is similar to registering an extended call package. See "Registering the Extended Package with the Session".
- Extending the Methods and Event Handlers
- Extending the MessageAlertPackage to Support Other Message Events

Extending the Methods and Event Handlers

When you extend the **MessageAlertPackage** class object, extend the required methods and event handlers:

- **onMessage**
- **onRehydration**. Extend this event handler so that your application can re-create the subsession object based on the rehydrated data your application receives through this event handler. When the subsession object is recreated, WebRTC Session Controller JavaScript API library invokes the **onResurrect** event handler of the rehydrated **Subscription** object.

Extending the MessageAlertPackage to Support Other Message Events

You can extend the **wsc.MessageAlertPackage** class object to support other message event types.

To do so:

- Use **wsc.extend** method to set up an extended **MessageAlertPackage** object.
- Override the **onMessage** event handler of the extended **MessageAlertPackage** object. Your application can now handle notifications other than the default **MessageSummary** type.
- Assign a callback function to handle the overridden **onMessage** event handler of the extended **MessageAlertPackage** object. In this callback function, process the new type of notification message.
- Define a custom class extended from the **Notification** class object. This new type of notification object will store the notification messages made available by the overridden **onMessage** event handler.
- Define a new class similar to **MessageCounts** and set it up to store the information on the new notification messages.

9

WebRTC Session Controller JavaScript API Error Codes and Errors

This chapter describes the error handlers and error codes provided in the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library.

> **Note:** See *WebRTC Session Controller JavaScript API Reference* for more information on the individual WebRTC Session Controller JavaScript API classes.

About wsc.ERRORCODE

The WebRTC Session Controller JavaScript API library provides the **wsc.ERRORCODE** enumerator object for the possible error codes. When there is an error, the appropriate error handler is called with the specific error code.

About the Error Codes

Table 9–1 lists the possible error codes and their descriptions.

Table 9–1 Error Codes and Their Descriptions

Error Code	Error Constant	Description
401	UNAUTHORIZED	The request requires user authentication.
403	FORBIDDEN	The server understood the request, but is refusing to fulfill it.
404	RESOURCE_UNAVAILABLE	The server has definitive information that the user does not exist at the domain specified in the Request-URI.
407	PROXYAUTH_REQUIRED	This code is similar to 401 (UNAUTHORIZED), but indicates that the client MUST first authenticate itself with the proxy.
480	TEMPORARILY_UNAVAILABLE	The callee's end system was contacted successfully but the callee is currently unavailable.
486	BUSY_HERE	The callee's end system was contacted successfully, but the callee is currently not willing or able to take additional calls at this end system.
487	REQUEST_TERMINATED	The request was terminated.
500	SYSTEM_ERROR	The server encountered an unexpected condition that prevented it from fulfilling the request.

Table 9–1 (Cont.) Error Codes and Their Descriptions

Error Code	Error Constant	Description
600	BUSY_EVERYWHERE	The callee's end system was contacted successfully but the callee is busy and does not wish to take the call at this time.
603	DECLINED	The callee's machine was successfully contacted but the user explicitly does not wish to or cannot participate.
1001	WEBSOCKET_ERROR	The websocket has an error or the connection has failed.
1101	PEERCONNECTION_ERROR	The peerConnection has encountered an error.
1201	MEDIA_ERROR	The media stream has an error.
1301	RESTORE_FAILED	The session state could not be reloaded.
1302	SAVE_FAILED	The session state could not be saved.

Using wsc.ErrorInfo

The **wsc.ErrorInfo** object enables you to handle error scenarios in your application. use the **code** and **reason** properties to retrieve the error code and reason and process the failure scenario accordingly.

About the Error Handlers

Assign callback functions and implement the logic to perform the following tasks:

- Handling Errors Related to Sessions
- Handling Errors Related to Calls
- Handling Errors Related to Data Transfers
- Handling Errors Related to Subscriptions

Handling Errors Related to Sessions

In our base case example, we created the *wseSession* object using the following statement:

```
wseSession = new wsc.Session(null, wsUri, sessionSuccessHandler, sessionErrorHandler);
```

When *wseSession* has an error, WebRTC Session Controller JavaScript API library invokes the callback function *sessionErrorHandler()* and provides the error as *error*, the argument in the *sessionErrorHandler()* callback function.

In the callback function assigned in your application to handle session-related errors, use the *error***.code** property to display the error code and *error***.reason** property to display the reason for the specific error as shown below in Example 9–1.

Example 9–1 Session Creation Error Handler

```
function sessionErrorHandler(error) {
    console.log("onSessionError: error code=" + error.code + ", reason=" + error.reason);
    setControls("<h1>Session Failed, please logout and try again.</h1>");
    ...
  }
```

Take any other action as appropriate for the session-related error.

Handling Errors Related to Calls

Suppose your application uses the following statement is used to create an instance of a **Call** object named *call*:

```
var call = callPackage.createCall(callee, callConfig, failureCallback);
```

Your application may be required to handle errors related to calls the following scenarios:

- If the call object named *call* is not created for some reason, Signaling Engine invokes the callback function *failureCallback* and provides the error as *error*, the argument in the *failureCallback* callback function.
- Your application invokes the **Call.start** method for this call and the WebRTC Session Controller JavaScript API library attempts to send the request to start the call. If an exception occurs:
  - Before the WebRTC Session Controller JavaScript API library sends the request to start the call, then the *failureCallback* function is invoked.

    In the callback function assigned in your application to handle call-related errors, use the **ErrorInfo.reason** property to display the reason for the specific error as shown below in Example 9–2.

Example 9–2 Handling Call-Related Error

```
function failureCallback(error) {
    alert('Call error reason:'+error.reason);
}
```

  - After the WebRTC Session Controller JavaScript API library sends the request to start the call, then the Signaling Engine invokes the **Call.onCallStateChange** event handler of the call with the call state as **wsc.CALLSTATE_FAILED**.

    Set up the appropriate actions in the callback function assigned in your application to **Call.onCallStateChange** to handle this call state.

Handling Errors Related to Data Transfers

In "Setting Up the Data Transfer State Event Handler for the Chat Session", the logic in the *onDataTransfer* callback function assigns *onDCError* as the callback function for data channel errors with the following statement:

```
dataTransfer.onError = onDCError;
```

If there is an issue in a chat session, in sending a text message or a data file, the WebRTC Session Controller JavaScript API library triggers *onDCError*, the error event handler and provides the appropriate error constant from the **WSC.ERRORCODE** enumerator object.

In the callback function assigned in your application to handle call-related errors, use the **ErrorInfo.reason** property to display the reason for the specific error. Take any other action as appropriate for the error.

Handling Errors Related to Subscriptions

If there is an issue in creating a subscription, Signaling Engine triggers the error event handler **onError** with the appropriate constant defined in the **WSC.ERRORCODE** enumerator object.

The following statement creates an instance of the **Subscription** class called *subscription*:

```
subscription = MsgAlertHandler.createNewSubscription(
      target,subscriber,onSubscribeSuccess,onSubscribeError,onNotification,onEnd,
extHeaders);
```

Where:

- *target* is the service target you obtained from the user, the device or the service the user wishes to monitor.
- *subscriber* is the user identity of this subscriber.
- *onSubscribeSuccess* is the event handler called when the application creates the subscription.
- *onSubscribeError* is the event handler called when the application fails to create the subscription.
- *onNotification* is the event handler for a notify message.
- *onEnd*, is the event handler called when the provider of the notification notifies Signaling Engine that this subscription has ended.
- *extHeaders* are the extension headers.

Example 9–3 shows the error callback function *onSubscribeError* called by an application. This function processes the error by calling *removeSubscriptionInfoElem()*. In this case, the *removeSubscriptionInfoElem()* function removes the information element for the subscription from the web application page.

Example 9–3 Subscription Creation Error

```
function onSubscribeError(errorObj) {
    console.log("Error code: "+errorObj.code);
    removeSubscriptionInfoElem();
  };
```

10

Sample Audio Call Application

This chapter shows the sample audio call application developed using the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library.

> **Note:** See *WebRTC Session Controller JavaScript API Reference* for more information on the individual WebRTC Session Controller JavaScript API classes.

About the Sample Audio Call Application

The sample audio call application supports audio calls only.

This application provides the logic necessary to enable two users who are in the same domain to place a call to each other. In this application, two users *bob1* and *bob2*, access our application from the *example.com* domain.

See "Setting Up Audio Calls in Your Applications" for more information on logic underlying this application logic.

> **Note:** This sample audio call application does not use extension headers. As a result, the **extHeaders** parameter is not used in this application code.

The Sample Audio Call Application Code

```
<!DOCTYPE HTML>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Audio Demo</title>


    <!-- Load wsc.js.
    It is assumed that this app is deployed on container which hosts wsc.
    Otherwise, the value of src might be http://my-wsc.com/api/wsc.js -->
    <script type="text/javascript" src="/api/wsc.js"></script>

</head>

<body onload="onPageLoad()">
```

```
        <h2 id="heading">Welcome to SDP InfoDev Demo 1 -- Audio Call</h2>
        <hr>

        <!-- Button for login. This is a start point of this page.
        Method register is invoked when the button is clicked. -->
        <div id="controlsArea">

        </div>

        <br>
        <br>
        <table hidden="true" id="media">
            <tr>
                <td>You</td>
                <td>Remote</td>
            </tr>
            <!-- HTML5 audio element. -->
            <tr>
                <td width="15%"><audio id="selfAudio" autoplay></audio></td>
                <td width="15%"><audio id="remoteAudio" autoplay></audio></td>
            </tr>
        </table>

        <script type="text/javascript">

            /** This app checks media stream support in Chrome or Firefox 23+  */
            // In your application, please use appropriate API to verify media stream support.
            var attachMediaStream = null;
            if (navigator.mozGetUserMedia) {
                console.log("Attaching media stream");
                // Attach a media stream to an element.
                attachMediaStream = function(element, stream) {
                    console.log("Application using Mozilla browser");
                    element.mozSrcObject = stream;
                    element.play();
                };
            } else if (navigator.webkitGetUserMedia) {
                console.log("Application using Chrome browser");
                // Attach a media stream to an element.
                attachMediaStream = function(element, stream) {
                    element.src = webkitURL.createObjectURL(stream);
                };
            } else {
                // The browser does not support media streams
                reptBrowserIssue();
            }

            //*************security login***************
            var demoName = " Audio Call Demo ";
            var wscSession, callPackage, userName, caller, callee;
            wsc.setLogLevel(wsc.LOGLEVEL.DEBUG);

            // Save where the user came from.
            var savedUrl = window.location;

            // This application is deployed on WebRTC Session Controller.
            var wsUri = "ws://" + window.location.hostname  + ":" + window.location.port +
"/ws/webrtc/sample";

            // login and logout URI to redirect the user.
```

```
            //var loginUri = "http://" + window.location.hostname + ":" + window.location.port +
"/infodev/wscdemo.html";
            var logoutUri = "http://" + window.location.hostname + ":" + window.location.port +
"/infodev/demos/wscdemo.html";

            // Configuring the audio and video settings in the CallConfig object.
            var audioMediaDirection = wsc.MEDIADIRECTION.SENDRECV;
            var videoMediaDirection = wsc.MEDIADIRECTION.NONE;
            var callConfig = new wsc.CallConfig(audioMediaDirection, videoMediaDirection);
            console.log("Created CallConfig with audio stream only.");
            console.log(" ");

            // The onPageLoad event handler.
            function onPageLoad() {
                console.log("Page has loaded. Setting up the Session.");
                setSessionUp();
            }

            // This function sets up and configures the WebSocket connection.
            function setSessionUp() {
                console.log("In setSessionUp().");

                // Create the session. Here, userName is null.
                // wsc can determine userName using the cookie of the request.
                wscSession = new wsc.Session(null, wsUri, sessionSuccessHandler,
sessionErrorHandler);
                // Register a wsc.AuthHandler with session.
                // It provides customized info of authentication, such as username/password.
                var authHandler = new wsc.AuthHandler(wscSession);
                authHandler.refresh = refreshAuth;
                // Configure the session.
                wscSession.setBusyPingInterval(2 *1000);
                wscSession.setIdlePingInterval(6 * 1000);
                wscSession.setReconnectTime(2 * 1000);
                wscSession.onSessionStateChange = sessionStateChangeHandler;

                console.log("Session configured with authhandler, intervals and sessionStateChange
handler.");
                console.log(" ");
            }

            // The function called when a session is instantiated.
            // The next steps are processed here.
            function sessionSuccessHandler() {
                console.log(" In sessionSuccesshandler.");

                // Create a CallPackage.
                callPackage = new wsc.CallPackage(wscSession);
                // Bind the event handler of incoming call.
                if(callPackage){
                    callPackage.onIncomingCall = onIncomingCall;
                }
                console.log(" Created CallPackage..");
                console.log (" ");
                // Get user Id.
                userName = wscSession.getUserName();
                console.log (" Our user is " + userName);
                console.log (" ");
            }
```

```
            // The function called when a session is not instantiated.
            function sessionErrorHandler(error) {
                console.log("onSessionError: error code=" + error.code + ", reason=" +
error.reason);
                setControls("<h1>Session Failed, please logout and try again.</h1>");
            }

            // This is a sample function. It requests the number to call.
            // 'onclick'='functionName()' for each button triggers the next step for the code.

            function displayInitialControls() {
                console.log ("In displayControls().");
                var controls = "Enter Your Callee: <input type='text' name='callee'
id='callee'/><br><hr>"
                + "<input type='button' name='callButton' id='btnCall'  value='Call'
onclick='onCallSomeOne()'/>"
                + "<input type='button' name='cancelButton' id='btnCancel'  value='Cancel'
onclick='' disabled ='true'/><br><br><hr>"
                + "<input type='button' name='logoutButton' id='Logout'  value='Logout'
onclick='logout()'/>"
                + "<br><br><hr>";
                setControls(controls);
                var calleeInput = document.getElementById("callee");

                if (calleeInput) {
                    console.log (" Waiting for Callee Input.");
                    console.log (" ");
                    if( userName != calleeInput) {
                        calleeInput.focus();
                    }

                }
            }

            // This example does not use either TURN or SERVICE authentication.
            // This function is provided as a reference for your use.
            function refreshAuth(authType, authHeaders) {
                var authInfo = null;

                if(authType==wsc.AUTHTYPE.SERVICE){
                //Return JSON object according to the content of the "authHeaders".
                // For the digest authentication implementation, refer to RFC2617.
                    authInfo = getSipAuth(authHeaders);

                } else if(authType==wsc.AUTHTYPE.TURN){

                    //Return JSON object in this format:
                    // {"iceServers" : [ {"url":"turn:test@<aHost>:<itsPort>", "credential":"nnnn"}
]}.
                    authInfo = getTurnAuth();
                }
                return authInfo;
            };

            // This function manages the session states.
            function sessionStateChangeHandler(sessionState) {
                console.log("sessionState : " + sessionState);
                switch (sessionState) {
                    case wsc.SESSIONSTATE.RECONNECTING:
                    setControls("<h1>Network is unstable, please wait...</h1>");
```

```
                    break;
                    case wsc.SESSIONSTATE.CONNECTED:
                    if (wscSession.getAllSubSessions().length == 0) {
                        displayInitialControls();
                    }
                    break;
                    case wsc.SESSIONSTATE.FAILED:
                    setControls("<h1>Session Failed, please logout and try again.</h1>");
                    break;
                }
            }

            // This function is the incoming call callback
            // wsc triggers this function when it receives the invite from the remote caller.
            function onIncomingCall(callObj, callConfig) {
                // We need the user's response. In this example code, we do the following:
                // We draw two buttons for users to accept or decline the incoming call.
                // Attach onclick event handlers to these two buttons.
                console.log ("In onIncomingCall(). Drawing up Control buttons to accept or decline
the call.");
                var controls = "<input type='button' name='acceptButton' id='btnAccept'
value='Accept "
                + callObj.getCaller()
                + " Incoming Audio Call' onclick=''/><input type='button' name='declineButton'
id='btnDecline'  value='Decline Incoming Audio Call' onclick=''/>"
                + "<br><br><hr>";
                setControls(controls);

                document.getElementById("btnAccept").onclick = function() {
                    // User accepted the call.

                    //  Store the caller and callee names.
                    callee = userName;
                    caller = callObj.getCaller;
                    console.log (callee + " accepted the call from caller " + caller);
                    console.log (" ");

                    // Send the message back.
                    callObj.accept(callConfig);
                }
                document.getElementById("btnDecline").onclick = function() {
                    // User declined the call. Send a message back.

                    // Get the caller name.
                    callee = userName;
                    caller = callObj.getCaller;
                    console.log (callee + " declined the call from caller, " + caller);
                    console.log (" ");

                    // Send the message back.
                    callObj.decline();
                }

                // User accepted the call. Bind the event handlers for the call and media stream.
                console.log ("Calling setEventHandlers from onIncomingCall() with remote call
object ");
                setEventHandlers(callObj);
            }
```

```
            // This function binds the call and media state event handlers to the call object.
            // It is called by when user is the caller or the callee.
            function setEventHandlers(callobj) {
                console.log ("In setEventHandlers");
                console.log (" ");
                callobj.onCallStateChange = function(newState){
                    callStateChangeHandler(callobj, newState);

                };
                callobj.onMediaStreamEvent= mediaStreamEventHandler;
            }

            // This function is an event handler for changes of call state.
            function callStateChangeHandler(callObj, callState) {
                console.log (" In callStateChangeHandler().");
                console.log("callstate : " + JSON.stringify(callState));
                if (callState.state == wsc.CALLSTATE.ESTABLISHED) {
                    console.log (" Call is established. Calling callMonitor. ");
                    console.log (" ");
                    callMonitor(callObj);
                } else if (callState.state == wsc.CALLSTATE.ENDED) {
                    console.log (" Call ended. Displaying controls again.");
                    console.log (" ");
                    displayInitialControls();
                } else if (callState.state == wsc.CALLSTATE.FAILED) {
                    console.log (" Call failed. Displaying controls again.");
                    console.log (" ");
                    displayInitialControls();
                }
            }

            // This event handler is invoked when "Call" button is clicked.
            function onCallSomeOne() {
                console.log ("In onCallSomeOne()");

                // Need the caller callee name. Also storing caller.
                callee = document.getElementById("callee").value;
                caller = userName;
                console.log ("Name entered is " + callee);

                // Did the user enter a blank ?
                if (callee === "") {
                    setControls("<h1>Invalid entry. Please enter the number you wish to
call.</h1>");
                    setTimeout(function(){ displayInitialControls();}, 2000)
                    return;
                }

                // Same domain case. The caller may not have given the entire name.
                if (callee.indexOf("@") < 0) {
                    console.log (" ");
                    callee = callee + "@example.com";
                    console.log("Complete caller ID is " + callee);
                }
                // Did the user enter his own number?
                if (callee == userName) {
                    setControls("<h1>You cannot call yourself. Please enter the number you wish to
call.</h1>");
                    return;
                }
```

```
                console.log(" Caller, " + caller + ", wants to call " + callee + ", the Callee.");
                console.log (" ");


                console.log("Creating call object to call  " + callee);

                // To call someone, create a Call object first.
                var call = callPackage.createCall(callee, callConfig, doCallError);
                console.log ("Created the call.");
                console.log (" ");

                if (call != null) {
                    console.log ("Calling setEventHandlers from onCallSomeOne() with call data.");
                    console.log (" ");
                    setEventHandlers(call);
                    // Then start the call.
                    console.log ("In onCallSomeOne(). Starting Call. ");
                    call.start();
                    // Allow the user to cancel call before it is set up. End the call.
                    // Disable "Call" button and enable "Cancel" button.
                    var btnCall = document.getElementById("btnCall");
                    btnCall.disabled = true;
                    var btnCancel = document.getElementById("btnCancel");
                    btnCancel.disabled = false;
                    console.log ("Enabled " + caller + " to cancel call.");
                    btnCancel.onclick = function() {
                        console.log ("In onCallSomeOne().");
                        console.log (caller + " clicked the Cancel button. Ending call. ");
                        console.log (" ");
                        call.end();
                        console.log (" If user logs out, the user will be sent back to " +
logoutUri);

                    }
                }
            }

            // This function monitors the call when call is established.
            function callMonitor(callObj) {
                console.log ("In callMonitor");
                console.log ("Monitoring the call. Setting up controls to Hang Up.");
                console.log (" ");

                // We need the user's response.
                //In this example code, we draw 2 buttons.
                // "Hang Up" button ends the call, but user stays on the application page.
                // "Logout" button ends the session, and user leaves the application.
                // Attach onclick event handler to each button.
                var controls = "<input type='button' name='hangup' id='btnHangup' value='Hang Up'
onclick=' '/><br><br>"
                                + "<input type='button' name='logoutButton' id='Logout'
value='Logout' onclick='logout()'/>"
                                + "<br><br><hr>";
                setControls(controls);
                document.getElementById("btnHangup").onclick = function() {
                    console.log (" In callMonitor.");
                    // Who ended the call?
                    if (userName == caller) {
```

```
                        console.log ("Caller, " + caller + ", clicked the Hang Up button. Calling
call.end now.");
                        console.log (" ");
                      } else {
                        console.log ("Callee, " + callee + ", clicked the Hang Up button. Calling
call.end now.");
                        console.log (" ");
                      }
                    callObj.end();
                };

            }

            // This event handler is invoked when a  media stream event is fired.
            // Attach media stream to HTML5 audio element.
            function mediaStreamEventHandler(mediaState, stream) {
                console.log (" In mediaStreamEventHandler.");
                console.log("mediastate : " + mediaState);
                console.log (" ");

                if (mediaState == wsc.MEDIASTREAMEVENT.LOCAL_STREAM_ADDED) {
                    attachMediaStream(document.getElementById("selfAudio"), stream);
                } else if (mediaState == wsc.MEDIASTREAMEVENT.REMOTE_STREAM_ADDED) {
                    attachMediaStream(document.getElementById("remoteAudio"), stream);
                }
            }

            // This function displays the controls set by the application.
            function setControls(controls) {
                var controlsArea = document.getElementById("controlsArea");
                controlsArea.innerHTML = controls;
            }

            // This function is called when the call is not created.
            function doCallError(error) {
                alert('Call error reason:' + error.reason);
            }

            // The browser does not support media streams
            // Use this function to exit gracefully.
            function reptBrowserIssue() {
                console.log("In reptBrowserIssue");
                console.log("Browser does not appear to be WebRTC-capable");
                logout();
            }

            // This function logs the user out.
            // For 3rd party authentication use login uri to send user back to where he came from.
            function logout() {
                console.log("In logout(). Closing session.");
                console.log (" ");
                if (wscSession) {
                    wscSession.close();
                }
                // Send the user back to where he came from.
                console.log (" In logout(). Sending user back to " + logoutUri);
                window.location.href = logoutUri;
            }
</script>
</body>
```

```
</html>
```