# Oracle® Communications WebRTC Session Controller

Application Developer's Guide

Release 7.2

**E69517-02**

December 2016

ORACLE®

Oracle Communications WebRTC Session Controller Application Developer's Guide, Release 7.2

E69517-02

# Contents

## 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 Using the WebRTC Browser Extension



## 10 Extending Your Applications Using WebRTC Session Controller JavaScript API

## 11 Debugging and Troubleshooting Your WebRTC-Enabled Applications

## 12 Developing WebRTC-Enabled Android Applications

## 13 Developing WebRTC-Enabled iOS Applications

## 14 WebRTC Session Controller JavaScript API Error Codes and Errors

# Preface

This document provides an overview of the Oracle Communications WebRTC Session Controller application programming interfaces (API) for JavaScript, Android, and iOS, that support 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, Android, or iOS APIs to create WebRTC enabled applications.

## 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 Android API Reference*
- *Oracle Communications WebRTC Session Controller iOS API Reference*
- *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 that have purchased support 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 support real-time communications (RTC) capabilities. The WebRTC standardization effort employs standardized browser capabilities, JavaScript application program interface (API), and HTML5 to support real-time multimedia communication in applications without the use of any browser plug-ins. For more information, see the WebRTC website at `https://www.webrtc.org`.

The WebRTC Session Controller JavaScript API enables your web applications to communicate with WebRTC Session Controller so that applications can allow the user to make calls, configure 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, when used in conjunction with WebRTC Session Controller, 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 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 network services endpoints. 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 *Oracle Communications 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 *Oracle Communications WebRTC Session Controller Extension Developer's Guide*.

## About Supported WebRTC-Enabled Browsers

WebRTC Session Controller works with any WebRTC-enabled browser. For a listing of browser versions that have been tested with WebRTC Session Controller, see `http://www.oracle.com/technetwork/developer-tools/webrtc/documentation/index.html`.

In addition, you can create WebRTC applications that work with Microsoft Internet Explorer version 11. For more information, see Chapter 9, "Using the WebRTC Browser Extension".

## 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, see http://www.webrtc.org/native-code/native-apis.

The WebRTC Session Controller JavaScript API library communicates with WebRTC Session Controller using the JSONRTC protocol for communication-related functions such as call control, file transfer, and message notification. The JSONRTC protocol is a sub protocol of the MessageBroker WebSocket protocol. For more information on JSONRTC, see Appendix A of the *Oracle Communications 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 Sample Applications

WebRTC Session Controller provides a set of sample applications that show the use of the Web SDK, the Android SDK, and the iOS SDK. These samples can be found in the *Oracle_home*/**wsc/samples** directory, where *Oracle_home* is the storage location for Oracle product files created during the installation of WebRTC Session Controller.

The following sample applications are included:

- **Android Sample App**: implementing the following use cases:
  - Audio/video calls
  - Data-channel chat
  - Call upgrades and downgrades
  - Standalone messaging
  - Device handover
  - Push notifications
- **iOS Sample App**: implementing the following use cases:
  - Audio/video calls
  - Data-channel chat
  - Call upgrades and downgrades
  - Standalone messaging
  - Device handover
  - Push notifications
- **Web Sample App**: implementing the following use cases:
  - Audio/video calls
  - Data-channel chat

  - Call upgrades and downgrades
  - Message Session Relay Protocol (MSRP) chat and file transfer
  - Standalone messaging
  - Device handover
  - Push notifications (Chrome only)
- **Video Conferencing Sample App**: implementing a video conferencing interface that integrates with Cisco Unified Communications Manager (CUCM) and Dialogic PowerMedia XMS.

Configuration and usage information for each sample application is available in **README.md** files in the respective sample directories.

---

**Note:** The **README.md** files are authored using markdown syntax which enables plain text to be formatted as styled HTML in Web browsers. For more information on markdown, see https://daringfireball.net/projects/markdown/.

---

## 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

# 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".
- REST-based authentication. See "Login Using REST 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 *Oracle Communications 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.

### Understanding OAuth 2.0 Concepts

OAuth is an open standard for authorization that allows subscribers to share their private resources with a third party without having to provide their own security credentials. These resources could be photos, videos, contact lists, location and billing capability, and so on, and are usually stored with another service provider. For example, photos stored on a dedicated photo Web storage site.

OAuth does this by granting requesting (client) applications a token, once access is approved by the resource owner. Each token grants access to a specific resource for a specific period. The requesting application uses the token for access to resources stored with another service provider, instead of the owner's credentials.

A resource can be:

- A single file such as a photo or video.
- Access to a website, such as the services of a video editing website.
- Personal information such as their location or billing capability.

Application can use these Security Assertion Markup Language (SAML) access tokens for single sign-on (SSO) authentication or to provide enhanced security profiles required for using derived values such as signatures or hash-method authentication codes (HMACs), or for client-server integration scenarios where the subscriber may not be present.

### Understanding OAuth Terminology

Table 2–1 lists OAuth terminology and definitions.

***Table 2–1 OAuth Terminology and Definitions***

| Term | Definition |
|---|---|
| Application Client | An application making protected resource requests on behalf of the resource owner and with the resource owner's authorization.<br>The term client does not imply any particular implementation characteristics (for example, whether the application executes on a server, a desktop, or other devices). |
| Application Instance ID | String that uniquely identifies the Application Instance. One applicationInstanceId can be mapped with one OAuth2 Client Identifier, so that SLA can be proceed for OAuth2 based traffic. |
| Authentication Server | Server that validates resource owner identity (defined by WebRTC Session Controller?). |
| Authorization Endpoint | Used to obtain authorization from the resource owner using user-agent redirection. |
| Authorization Grant | Represents the authorization given by the resource owner to a client application. An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token. |
| Authorization Server | Server that issues authorization codes and access token. |
| Access Token | Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. |
| Client Identifier | A unique string representing the registration information provided by the client. |
| Custom Subscriber Manager | Component that authenticates resource owner's username and password with a custom identity store (such as LDAP). |
| Delegated Authentication | Authentication mode supported by WSC to integrate with 3rd party authentication systems. |
| Grant Endpoint | URI supported by WSC to post the authentication result to issue the authorization code (defined by WSC). |
| Group Owner | Owner of the Group URI. Issues authorization token on behalf of the group members. In the example, Google. |
| Group URI | URI that represents a group of Resource Owners. |
| OAuth | Open Authorization Protocol |

*Table 2–1 (Cont.) OAuth Terminology and Definitions*

| Term | Definition |
|---|---|
| Protocol Endpoint | Network URI representing the location of a service to:<br>■ obtain an authorization code and other values<br>■ obtain an access token<br>■ submit a grant.<br>■ access a resource |
| Redirection Endpoint | After completing its interaction with the resource owner, the Authorization Server directs the resource owner's user-agent back to the client. The Authorization Server redirects the user-agent to the client's redirection endpoint previously established with the Authorization Server during the client registration process or when making the authorization request. |
| Refresh Token | Refresh tokens are credentials used to re-obtain access tokens. |
| Resource | The Web resource protected by OAuth Protocol. |
| Resource Owner | An entity capable of granting access to a protected resource. In the operator context this is defined as Resource owner URI (tel: or sip:) |
| Resource Server | Server that hosts protected resources and validates access token during resource access. |
| scopeId | Unique string that identifies a resource and used as part of scope-token by application client. |
| Subscriber Manager | Component in WSC to validate subscribers provisioned in its database. |

## About the OAuth/WSC Entities and Their Relationships

*Figure 2–1 OAuth Flow and Entity Relationships*

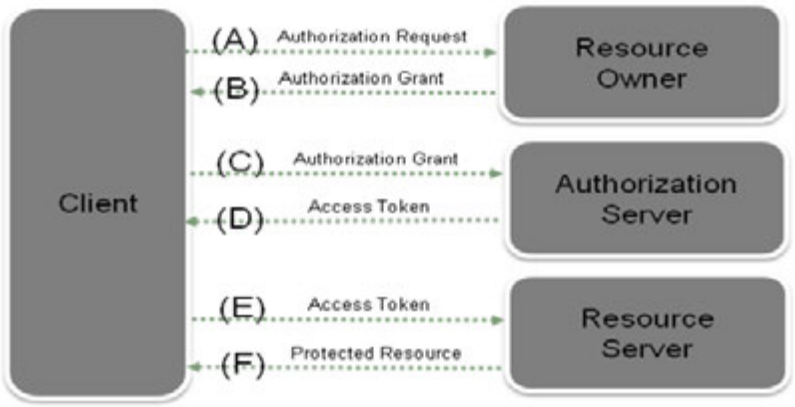



Figure 2–1 shows an example OAuth protocol flow for a resource access request:

(A) The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly using the Authorization Server as an intermediary.
(B) The client receives an authorization grant, including a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an Extension grant type. The authorization grant type depends

on the method used by the client to request authorization and the types supported by the Authorization Server.
(C) The client requests an access token by authenticating with the Authorization Server and presenting the authorization grant.
(D) The Authorization Server authenticates the client and validates the authorization grant, and if valid issues an access token.
(E) The client requests the protected resource from the resource server and authenticates by re-sending the access token.
(F) The resource server validates the access token, and if valid, serves the request.

### About the OAuth Protocol Endpoints

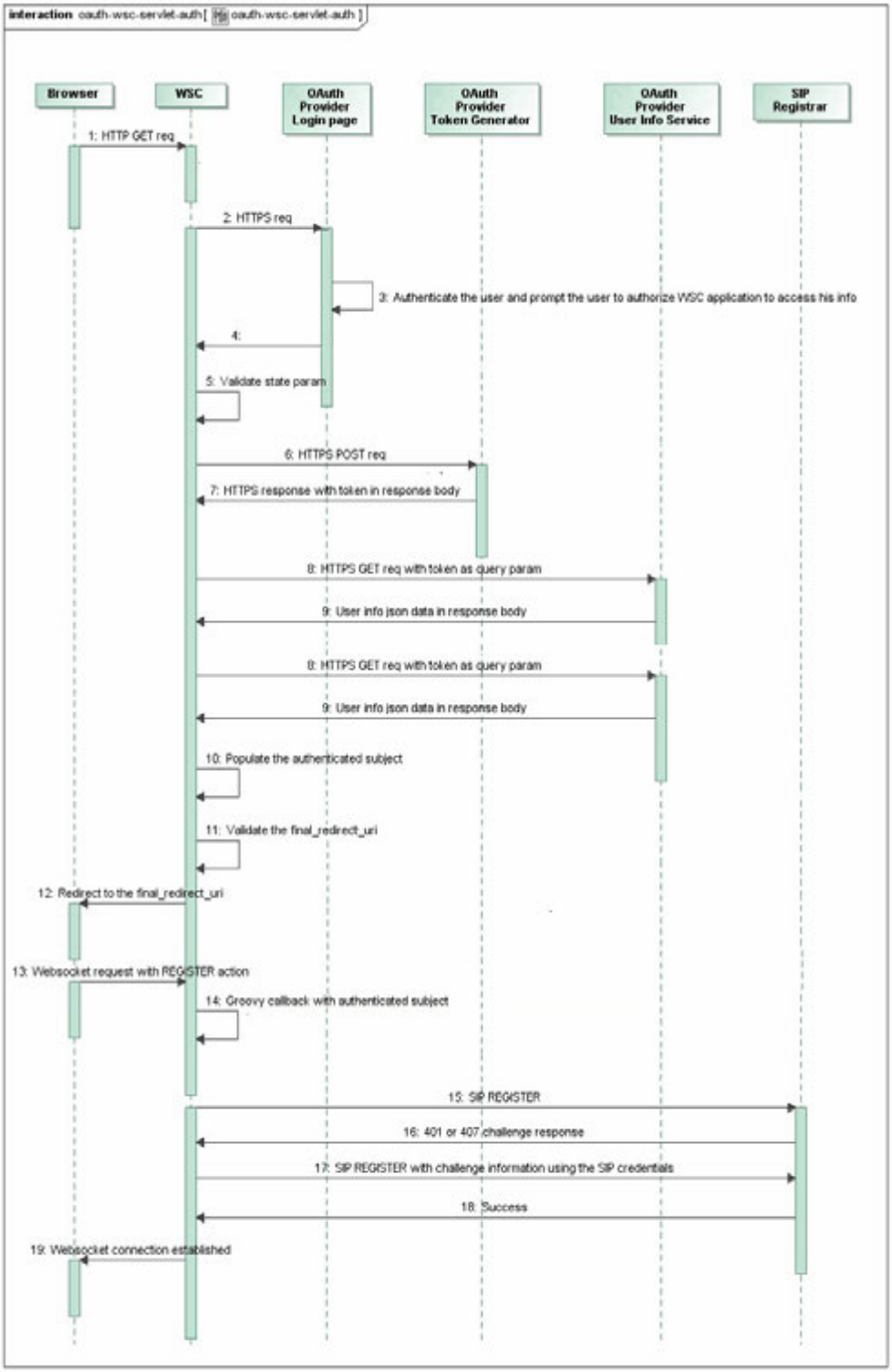
The OAuth specification defines three types of protocol endpoints.

- **Redirection Endpoint**: The redirection endpoint is a URI used by Authorization Server to return authorization credentials responses from the Authorization Server to the client using the resource owner user-agent.
- **Authorization Endpoint**: The authorization endpoint interacts with the resource owner (typically the subscriber) and obtain an authorization grant which will be issued to an application client by the resource owner. The Authorization Server must first verify the identity of the resource owner before granting the authorization grant. This authorization grant will be exchanged by the application client for an access token.
- **Token Endpoint**: The client uses the token endpoint to obtain an access token by presenting its authorization grant (the authorization code) or refresh token. The token endpoint is used with every authorization grant except for the implicit grant type (since an access token is issued directly).

## The OAuth2 Authentication Process

Figure 2–2 shows how OAuth2 authentication process from the client side proceeds when the user clicks the **Login** button on the web page:

***Figure 2–2 OAuth2 Authentication Process***




The following process shows the two tasks a WebRTC application performs in the required sequence when a user logs in to the application. Steps 1-12 deal with the OAuth2 login authentication and steps 13-19 with the obtaining of a WebSocket connection to a SIP server.

1. Your WebRTC application sends a HTTPS GET request to the WebRTC Session Controller Signaling Engine (WSC in the figure).

***Example 2–1 HTTPS Login Request from a Chrome Browser***

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. In this case, *google*.
- **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.

2. WSC does the following:
   a. Based on the URL query parameters, identifies that the request is for the OAuth login mechanism.
   b. Sends a HTTPS request to the OAuth provider's login dialog page.

      This HTTPS request contains the **client_id**, the **redirect_uri**, **response_type**, **scope**, and **state**, a recommended security practice in the OAuth specification. WSC generates a unique string as the **state** parameter and sends it to the OAuth2 provider. In turn, the OAuth2 provider returns this string to WSC.
3. The user provides his or her credentials to the OAuth provider. In the case of Chrome example, the Gmail login and password. This allows the WSC application to access the user information.
4. The OAuth provider authenticates the user. It redirects the user back to WebRTC Session Controller with the OAuth access **code** information along with **state** as a query parameter.

5. WSC validates the **state** that it received from the OAuth provider.
6. WSC sends a HTTPS POST request to obtain the access token. The request includes the **client_id**, **client_secret**, **redirect_uri**, **grant_type** set to access_code, and **code** with code_received.
7. The OAuth2 provider responds by placing the access token in the response body.
8. WSC sends a HTTPS GET request with the access token as the query parameter to OAuth Provider. This is a request authorization to obtain details about the user (not the password).
9. The OAuth2 provider responds with the user information as a JSON data.
10. WSC sets up the authenticated subject for use in mapping Web credentials to SIP credentials. It populates this authenticated subject with the user name (user's mail with the OAuth provider), and a group name configured in the OAuth security provider, for example "google".

    At this point, you can customize the security provision. For example, you can chain additional security providers such that they can add more credential information for the authenticated subject.
11. WSC verifies the location to which Signaling Engine should redirect the user after a successful OAuth login by validating the value of the **final_redirect_uri**.
12. WSC sends the verified **final_redirect_uri** to the browser where your WebRTC application resides.
13. The WebRTC application sends a WebSocket request to the Signaling Engine, to register for message events.
14. The WSC Signaling Engine sets up a Groovy callback with data from the authenticated subject.
15. The WSC Signaling Engine attempts to register with the SIP Server by calling the SIP REGISTER method.
16. The registrar for the SIP Server sends back a challenge.
17. The WSC Signaling Engine sends the SIP credentials in the SIP REGISTER method back to the SIP Server.
18. The SIP SERVER validates the user's credentials. It registers the user in its database and sends back a success response.
19. The WebSocket connection with your WebRTC application is now established.

## 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 *Oracle Communications WebRTC Session Controller Extension Developer's Guide*.

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

## Login Using REST Authentication

REST stands for *representational state transfer*, a style of network architecture that complies with the following constraints:

- A uniform separation between client and server so that they need be concerned only about the interface between them.
- Stateless client and server with each request containing all the necessary information so that neither side needs to store context. The state contains links that the client can use in the future to begin a new state-transition.
- Cachable responses that are defined to prevent clients from reusing stale or inappropriate data in responding to further requests.
- A layered system in which the client does not know if it's connected to an end or intermediary server. Layered systems can improve scalability, providing load balancing, shared caching, and so on.
- Optional code-on-demand that enables a server to transfer Java applets or JavaScript code to a client to temporarily enhance its capabilities.
- A uniform interface that enables each part of the architecture to evolve independently.

To enable REST authentication, you must install a REST service provider and enable it through the WLS administration console. See the section on configuring WSC authentication in the *WebRTC Session Controller System Administrator's Guide* for information on installing and enabling a REST service provider.

When a client logs in, REST authentication occurs as follows:

1. The client sends the request URI to the WebRTC Session Controller, indicating that the request type is REST authentication:

   ```
   http://wsc-host:wsc-port/login?wsc-app-uri=/ws/webrtc/restauth&redirect-uri=http://successpage.html
   ```

   Where:

   - *wsc-host* is the host name where WebRTC Session Controller is running
   - *wsc-port* is the listening port for WebRTC Session Controller
   - *successpage.html* is the landing page where the browser should be directed after a successful login

2. WebRTC Session Controller responds and triggers a client authentication pop-up window that prompts the user for a login name and password.
3. The client sends the login name and password to WebRTC Session Controller.
4. The WebRTC Session Controller forwards the login name and password to the REST service provider, which validates them.
5. The REST service provider returns one of the following results:
   - A 200 message to indicate that the client authentication was successful

- A 401 message to indicate that the client was not successfully authenticated

6. On successful authentication, WebRTC Session Controller populates the authentication context, AuthenticateContext, which is defined in the Groovy script library.

## 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 *Oracle Communications WebRTC Session Controller System Administrator's Guide* for information on session timeout if the WebSocket loses its connection.

# 3

# About Using the WebRTC Session Controller JavaScript API

This chapter presents a general overview of the Oracle Communications WebRTC Session Controller JavaScript SDK 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 APIs 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.
- Supporting Web Client Notifications in Your Applications.
- Managing the Sessions in Your Application.
- Handling Session Rehydration on the Same Client Device.
- Handling Session Rehydration When the User Moves to Another Device.
- Configuring Screen Sharing.

For information about:

- Debugging and troubleshooting your WebRTC-enabled applications, see "Debugging and Troubleshooting Your WebRTC-Enabled Applications".
- Error codes used by the WebRTC Session Controller JavaScript SDK, see "WebRTC Session Controller JavaScript API Error Codes and Errors".
- Service provider Interface (SPI) functions supported by this SDK, see "Extending Your Applications Using WebRTC Session Controller JavaScript API".

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

For more information about the individual WebRTC Session Controller JavaScript API classes, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

# About the wsc Namespace

The **wsc** namespace exposes the WebRTC Session Controller JavaScript SDK. You can access its objects and methods in your applications.

For more information about the **wsc** namespace, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

## About Using the WebRTC Session Controller JavaScript API Library

The WebRTC Session Controller JavaScript SDK can be used to provide real-time communication-related functionality in your applications. For example, you can support audio calls, video calls, data transfers, and message notifications.

### About the API to Use for General Tasks

The following WebRTC Session Controller JavaScript APIs are used for general tasks in your application.

- **wsc.Session**

  Manage the application session with **wsc.Session**. See "Managing Sessions with wsc.Session".

- **wsc.AuthHandler**

  Authenticate users **wsc.AuthHandler**. See "Authenticating Users with wsc.AuthHandler".

- **wsc.SESSIONSTATE**

  Manage changes in the state of the application session with **wsc.SESSIONSTATE**. See "Handling Session State Changes".

- **wsc.LOGLEVEL**

  Set the logging level with **wsc.LOGLEVEL**. See "Debugging Your Application with wsc.LOGLEVEL".

- **wsc.ERRORCODE**

  Respond to errors with **wsc.ERRORCODE**. See "WebRTC Session Controller JavaScript API Error Codes and Errors".

In addition, you can 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**

  Manage call applications with **wsc.CallPackage**. See "Managing Calls with wsc.CallPackage".

- **wsc.Call**

  Manage calls with **wsc.Call**. See "Managing a Call with wsc.Call".

- **wsc.CallConfig**

  Set up the capabilities of the call with **wsc.CallConfig**. See "Specifying the Media Stream for Calls in the CallConfig Object".

- **wsc.CALLSTATE**

  Manage the changes in the call state with **wsc.CALLSTATE**. See "Handling Changes in Call States".

- **wsc.MEDIASTREAMEVENT**

  Manage the changes in the media stream with **wsc.MEDIASTREAMEVENT**. See "Handling Changes in Media Stream States".

- **wsc.DataTransfer**

  Manage a data channel between two peers with **wsc.DataTransfer**. See "Transferring Data With wsc.DataTransfer".

- Send raw data over the data channel with **wsc.DataSender**. See "Sending Data Using wsc.DataSender".

- **wsc.DataReceiver**

  Receive raw data over the data channel with **wsc.DataReceiver**. 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 are described in "Setting Up Message Alert Notifications".

### About Extending WebRTC Session Controller JavaScript API

You can extend your applications by extending WebRTC Session Controller JavaScript API classes. See "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 using the **wsc.Session** class. It represents a persistent association 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 an account holder accesses your application on the web page, create an instance of the **Session** class before you use any of the other WebRTC Session Controller JavaScript API objects.

You provide the following information when you create a session object:

- The user name.
- The WebSocket URI.
- The callback function in your application to call when the session is created.
- The callback function in your application to call when there is an error in creating the session.

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

Your application 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 is successfully created, provide the settings to manage the connection. See "About Monitoring Your Application WebSocket Connection".

For information about the various session states and how your application can manage changes in its session state, see "Handling Session State Changes".

At times, your application must maintain session transfers in a device handover. For example, Alice is on a cellphone call to a sales representative. During the call, Alice continues the call on the softphone on the laptop. See "Handling Session Rehydration When the User Moves to Another Device", for more information.

### 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 do not disrupt message flow during the life of the session.

For example, based on the action of your user, your application sends a request to a target uniform resource locator (URL). WebRTC Session Controller Signaling Engine forwards the request to the target URL. The Session Initialization Protocol (SIP) proxy or registrar server checks the user credentials. If the user credential information is inadequate, the SIP server does not allow that request to go further in the target environment. It sends back a challenge to WebRTC Session Controller Signaling Engine asking for more credential information.

The challenge can be from a 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 SDK. The WebRTC Session Controller JavaScript SDK calls the **refresh** event of the **wsc.AuthHandler** object.

Your application can respond to such a challenge by retrieving the user credentials and returning that information as a JSON object. On receiving this JSON object, WebRTC Session Controller Signaling Engine can then send the response from your application to the SIP proxy or 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. WebRTC Session Controller can determine it
// by the cookie of the request.
wscSession = new wsc.Session(null, webSocketUri, sessionSuccessHandler,
sessionErrorHandler);
// Register a wsc.AuthHandler with the session,
// which 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 as one of the following:

- **wsc.AUTHTYPE.TURN**: The type of authentication that allows for the client to authenticate with a TURN server. TURN servers facilitate communication between clients residing behind network address translator (NAT) routers or firewalls.

- **wsc.AUTHTYPE.SERVICE**: The type of authentication used when a back-end SIP application, such as the Proxy Registrar, requires user authentication.

To obtain the authentication information from the *authHeaders*, use the value in *authType*. Return the authentication information as a JSON object, as shown in Example 4–8, "Template for the refreshAuth Function()".

### How Your Application Saves Session Information

By default, the WebRTC Session Controller JavaScript API SDK stores your application data in JSON format in the **sessionStorage** object associated with the browser. It saves all the session information associated with the following objects in your application session:

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

> **Note:** If you extend any WebRTC Session Controller JavaScript API class object, save the session identifier (*sessionID*). Use this value when you create the current session after your application page reloads.

### Handling Session State Changes

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

Whenever the state of your application session changes, the WebRTC Session Controller JavaScript SDK calls the **Session.onSessionStateChange** event handler in your application and provides the new state. Set up a callback function in your application session object, and assign that function to the **Session.onSessionStateChange** event handler in your application.

In the callback function, 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, take appropriate action in the application logic.

For a more information about the **wsc.SESSIONSTATE** enumerator, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

### Debugging Your Application with wsc.LOGLEVEL

Set up the type of records your application must log with the **wsc.LOGLEVEL** enumerator object. The supported log levels are:

- DEBUG (0)
- INFO (1)
- WARN (2)
- ERROR (3)
- OFF (4)

To set up the log level for debugging, pass the required constant to the **setLogLevel** method at the start of your JavaScript application:

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

Alternatively, you can directly input the associated numeric value, in this case, 0, corresponding to the DEBUG log level:

```
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 communication, video communication, and data transfers in calls. The WebRTC Session Controller JavaScript SDK handles the messaging and call flow for all calls created through the **wsc.CallPackage** object for an application session.

In the following example code, an application creates an instance of **wsc.CallPackage** named *callPackage* for *wscSession*, the application 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. Filter the call to reject blacklisted numbers or respond when the callee accepts or declines the call. For information about 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.

To create outgoing calls, use the **CallPackage** object. See "Managing a Call with wsc.Call".

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

### Managing a Call with wsc.Call

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

Set up the **Call** object in your application in the following ways:

- For the caller, create the call object using the **CallPackage.createCall** method and provide the 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 SDK calls the **CallPackage.onIncomingCall** event handler and provides the incoming call object and its call configuration, as shown in

Example 4–13, "Sample onIncomingCall Function".

Manage changes in the state of the call and its associated audio, media, and data channel with the event handlers of the **wsc.Call** class. Implement the appropriate 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".
- dataTransfer object creation, use the **onDataTransfer** event handler. See "Transferring Data With wsc.DataTransfer".

### Specifying the Media Stream for Calls in the CallConfig Object

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

When you create the **wsc.CallConfig** object, set up the direction of the audio and video elements in the local media stream. To specify the direction of the local media stream, use the **wsc.MEDIADIRECTION** enumerator values:

- **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.

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;
```

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

### Defining Data Transfers with dataChannelConfig Parameter

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.

The **dataChannelConfig** parameter is an array of JavaScript Object Notation (JSON) objects in the format:

```
{"label":"DataLabel", "ordered": true }
```

The **dataChannelConfig** parameter supports the following keys:

- **label**

  The key is required. It represents a label that can be used to distinguish this **RTCDataChannel** object from other **RTCDataChannel** objects.

- **ordered**

Optional key. By default, initialized to **true**, to indicate that the **RTCDataChannel** is ordered.

- **maxPacketLifeTime**

  Optional key. The length of the time window (in milliseconds) during which transmissions and retransmissions could occur in unreliable mode. By default, initialized to **null**.

- **maxRetransmits**

  Optional key. The maximum number of retransmissions that are attempted in unreliable mode. By default, initialized to **null**.

- **protocol**

  Optional key. The name of the sub-protocol used with this RTCDataChannel, if any. By default, initialized to an empty string **""**.

- **negotiated**

  Optional key. By default, initialized to **false**. This value is **true** if this RTCDataChannel was negotiated by the application.

- **id**

  The ID for this **RTCDataChannel**, set when the **RTCDataChannel** was created. The ID was either assigned by the user agent at channel creation time or selected by the script.

Set up a call configuration object called *callConfig* defining the local media stream and data transfers, as shown in this example:

```
var audioMediaDirection = wsc.MEDIADIRECTION.SENDRECV;
var videoMediaDirection = wsc.MEDIADIRECTION.NONE;
var dtConfigs = new Array();
dtConfigs[0] = {"label":"DataLabel", "ordered": true };
var callConfig = new wsc.CallConfig(audioMediaDirection, videoMediaDirection,
dtConfigs);
```

### Handling Changes in Call States

Configure your application 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 about the **wsc.CALLSTATE** enumerator, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

Process changes to the current call within the callback function for the **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. The local component in the browser associates with your application. The remote component relates to the browser associated with the other party.

In your application, handle the changes in the media stream states of the call, whether it is voice or video. The WebRTC Session Controller JavaScript SDK 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**)

For information about how to use the **wsc.MEDIASTREAMEVENT** enumerator, see "Setting Up the Event Handler for Call State Changes".

For a more information about the **wsc.MEDIASTREAMEVENT** enumerator, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

### Transferring Data With wsc.DataTransfer

Support text messaging, chat sessions, and file transfers in your application through the corresponding data channels of the data transfer objects. 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** for the data channel. For information about **dataChannelConfig**, the data channel configuration parameter in the **CallConfig** object, see "Specifying the Media Stream for Calls in the CallConfig Object".

You can retrieve the following information from the **DataTransfer** object:

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

Manage the open, closed, and error states of a data transfer with the **onOpen**, **onClose**, and **onError** event handlers associated with the **DataTransfer** object.

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

### Sending Data Using wsc.DataSender

The **wsc.DataSender** object represents the sender of a data transfer.

Send a raw data object in the data channel of a data transfer as a string or a binary large object (BLOB) in the **DataTransfer** object. Retrieve the identity of the sender by calling the **getSender()** method of the **DataSender** object. Set up the data object and send it using the **send** method of the **DataSender** object in your application. For more information, see "Sample Send Function".

### Receiving Data Using wsc.DataReceiver

The **wsc.DataReceiver** class represents the receiver of a data transfer.

If your application supports receiving raw data, set up a callback function for the **onMessage** event handler of **wsc.DataReceiver**. 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,

- *receiver* is the instance of **wsc.DataReceiver** in the application.
- *evt* is the raw data in its entirety such as a text string, a BLOB, or array data.

## About the Code Segments Displayed in This Guide

The example code segments in this guide focus on the features of the WebRTC Session Controller JavaScript SDK. They 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 use the **console.log** method to display debug messages. To assist you in your application development process, use the JavaScript Console API methods supported in the web browser.

# About the Application HTML File

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



## About Web Applications Using WebRTC Session Controller JavaScript API

Audio, media stream, or data transfer (such as chat sessions) in WebRTC applications are associated with a call or a subscription in the application. For web applications using the WebRTC Session Controller JavaScript SDK, 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 consists of the following:

- Enabling a caller who is logged in to your application to start a call.

  To do so, ensure that your application does the following:

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

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

  To do so, ensure that your application does the following:

  - Sets up the necessary elements to respond to the incoming call request and perhaps filters the incoming call.
  - If necessary, your application 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, it completes the steps to establish the call session.

  - If the callee declines the call, it takes appropriate steps.
  - If the caller cancels the call before it is established, it takes appropriate steps.
- Monitoring the established call session until one of the parties ends the call.

  To do so, ensure that your application does the following:

  - 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 an account holder who is logged in to subscribe to receiving notifications.

  To do so, ensure that your application does the following:

  - Sets up the elements necessary for the account holder to subscribe for notifications.
  - On receiving the subscription target from the subscriber, it sets up the subscriptions.
- Enabling the account holder to access and process the received notifications.

  To do so, ensure that your application does the following:

  - Sets up the elements necessary to receive the incoming notification and displays it for the user.
  - Sets up the logic to respond to actions taken by the account holder.

## WebRTC Session Controller Support Libraries

WebRTC Session Controller provides a set of libraries that you include in your application to support various WebRTC functions. Table 3–1 lists the provided JavaScript files and details the functions each file supports.

***Table 3–1 WebRTC Session Controller JavaScript Files***

| File name | Description |
|---|---|
| **wsc-call.js** | Call Package API implementation.<br>To support audio and video calls, include this file in your application. For more information, see "Setting Up Audio Calls in Your Applications". |
| **wsc-capability.js** | Capability exchange API implementation.<br>To support capability exchange, include this file in your application. For more information, see "Capabilities Exchange". |
| **wsc-chat.js** | Message Session Relay Protocol (MSRP) chat API implementation.<br>To support one-to-one and one-to-many chat functionality, include this file in your application. For more information, see "Creating an RCS Chat Application". |

*Table 3–1 (Cont.) WebRTC Session Controller JavaScript Files*

| File name | Description |
|---|---|
| **wsc-common.js** | Common utilities and functionality required by other files. **This file is always required** unless you are including **wsc.js**. |
| **wsc-filetransfer.js** | File transfer API implementation.<br>To support file transfers in your application, include this file. For more information, see "Implementing File Transfer". |
| **wsc-messaging.js** | Stand alone messaging API implementation.<br>To support basic messaging, include this file. For more information, see "Sending a Standalone Message". |
| **wsc-msgalert.js** | Message notification API implementation.<br>To support message alerts, include this file. For more information, see "Setting Up Message Alert Notifications". |
| **wsc.js** | A combined file including all the WebRTC Session Controller APIs. |

Insert references to the appropriate libraries in the *head* element of your application HTML file.

***Example 3–1 Referencing WebRTC Session Controller JavaScript Libraries***

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

In Example 3–1, *wsc_context_root* represents the HTTP location where the WebRTC Session Controller is provisioned. The libraries **wsc-common.js** and **wsc-call.js** are included to support audio and video calling. You can include other libraries from Table 3–1 depending on the requirements of your application.

**Note:** The utilities file, **wsc-common.js**, is required for all applications. Include this file, unless you are using the composite **wsc.js** file which includes all the APIs as well as the utility functions.

**Note:** To access the entire WebRTC Session Controller JavaScript API, include the composite **wsc.js** file. To improve application performance, include the files supporting the specific functionality you require only.

## Including WebRTC Browser Support

For information on an additional JavaScript library that is needed to provide WebRTC browser support for Internet Explorer and Safari, see "Using the WebRTC Browser Extension."

### Verifying Browser Capabilities

At the start of your application logic, check your browser to see if it can access the local media stream, including audio and 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, attach the media stream to the appropriate video/audio HTML5 media element.

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 perform a graceful exit.
      reportBrowserIssue();
     };
```

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. The WebRTC Session Controller JavaScript SDK monitors this connection between your application and WebRTC Session Controller Signaling Engine.

When you instantiate your session object, configure how the WebRTC Session Controller JavaScript APIs check the WebSocket connection. Monitor the state of the connection, by setting the following values in the **Session** object:

- **Session.ackInterval**, the interval between receiving acknowledgement messages.
- How often the WebRTC Session Controller JavaScript SDK 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 timeout retry count. After this count is reached, the WebRTC Session Controller JavaScript SDK reconnects to the WebRTC Session Controller Signaling Engine. The default value is **2**.
- **Session.reconnectTime**, which specifies the maximum time for the interval during which the WebRTC Session Controller JavaScript SDK attempts 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 called 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 WebSocket connection.

When there is a device handover, your application suspends the application session. The WebSocket connection closes abnormally. See "Handling Session Rehydration When the User Moves to Another Device".

# 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.

## 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.

## About Handling Events in the Application Environment

Configure your application to handle the following events in your deployment environment:

- Client rehydration

  At times, the account holder reloads the application page, or the application reloads as a result of the browser. See "Handling Session Rehydration on the Same Client Device".

- Network switchover

  A network switchover takes place when the IP address of the account holder changes. For example, the associated device switches from a WI-FI network to 4G network. When a network switchover takes place, the WebRTC Session Controller JavaScript SDK reconnects your application session with WebRTC Session Controller Signaling Engine. It then tries to resurrect all the subsessions inside the **Session** object, such as the call and the subscription.

  For information about handling a call after a network switchover, see "Restoring a Call Session". For information about handling a subscription after a network switchover, see "Restoring a Subscription Session".

- Clustered server shut down

  In a cluster environment, the server to which your application browser web browser is connected shuts down. In this scenario, the WebRTC Session Controller JavaScript SDK tries to reconnect your application session to the corresponding failover server. If the connection is re-established, the WebRTC Session Controller JavaScript SDK attempts to resurrect all the subsessions inside the **Session** object.

  For information about handling a call after a server failover, see "Restoring a Call Session". For information about handling a subscription after a server failover, see "Restoring a Subscription Session".

- Session Hibernation

  When there is a period of inactivity, your applications can go into hibernation and disconnect from the WebRTC Session Controller server. It is awakened when there is a notification, and handles that event. See "Supporting Web Client Notifications in Your Applications".

- Session Rehydration following a Device Handover

  In a device handover, a customer account is associated with more than one device. The customer starts a call on one device. During the call, the customer switches to another device continue the call. For example, Alice is on a cellphone call to a sales representative. During the call, Alice switches attempts to continue the call on the softphone on the laptop. See "Handling Session Rehydration When the User Moves to Another Device".

## Supporting Web Client Notifications in Your Applications

Configure support for notifications in your applications. A client application running on a browser retrieves a registration ID from its notification provider. With this registration, your application users are notified of various events when the application hibernates. The notifications are related to web events, such as a promotion offered by a website to which the account holder has subscribed.

An application session for an account holder (Bob) is in hibernation. Then an event, such as an incoming call for Bob, occurs. The WebRTC Session Controller server sends

an HTTP message to the cloud messaging server. The response to this message contains the registration ID for the application and the payload. On being woken up on that browser, your application reconnects with the server. It uses the saved session ID to resurrect the session data, and handles the incoming event.

If no event occurs during the specified hibernation period, there are no notifications to process. The hibernation period expires and the WebRTC Session Controller server cleans up the session.

Such support of web client notifications enables your applications to lower Firewall licensing costs by closing connections associated with hibernating sessions. After storing the session state in passive storage, your application can release server resources such as open files and network connections.

Preliminary configurations and registration actions are necessary to support client notifications in your applications. They provide necessary information about the browser, the APIs, the application, and so on to the WebRTC Session Controller server and the cloud messaging provider.

## About the WebRTC Session Controller Notification Service

The WebRTC Session Controller Notification Service manages the external connectivity with the respective notification providers. It implements the Cloud Messaging Provider specific protocol such as GCM, APNS. The WebRTC Session Controller Notification Service ensures that all notification messages are transported to the appropriate notification providers.

---

**Important:** To activate the WebRTC Session Controller Notification Service, ensure that you configure the service using the Configuration tab in the WebRTC Session Controller Administration Console. See the description about "Configuring WebRTC Session Controller Notification Service" in *WebRTC Session Control System Administrator's Guide*.

---

The WebRTC Session Controller server constructs a message to trigger the notification provider to deliver a push notification to the web application. It constructs the message using the following:

- The received message payload from your application.
- The payload configured in the application settings.
- The application provider settings you provide WebRTC Session Controller.

### About Employing Your Current Notification System

At this point, verify if your current installation has an existing notification server that talks to the Cloud Messaging system and that the installation supports applications for your users through this server. If you currently have such a notification server successfully associated with a cloud messaging system, you can use the pre-existing notification system to send notifications using the REST interface.

## How the Notification Process Works

Let us take the WebRTC call as an example. The notification process works in this manner:

1. Bob, a valid account holder, accesses a web browser, such as Google Chrome or Mozilla Firefox.
2. The web browser registers with the notification service. The response from the browser contains the device token. It is the registration ID from the notification provider such as Google Cloud Messaging system or Apple Push Notification Service. This registration ID identifies browser instance.
3. Bob logs in to your application on the web browser. In this scenario, your audio call application features on that web browser.
4. The WebRTC Session Controller client SDK connects with the WebRTC Session Controller server and a session is created.

   A WebSocket connection is opened.
5. When there is inactivity on the part of the account holder (Bob), your application goes into the background. Your application sends a message to the WebRTC Session Controller server informing the server of its intent to hibernate and specifies a time duration for the hibernation.

   The WebSocket connection closes.
6. During the hibernation period, an event occurs. For example, Alice calls Bob, a call that is targeted to the audio call feature on your application.
7. WebRTC Session Controller server receives this call request from Alice. It checks the session state and determines that the request came during the hibernation period. The WebRTC Session Controller server uses its notification service to send a notification to the notification provider.
8. The notification provider delivers the notification to your audio call application on the web browser.
9. On receiving this notification:
   a. Your application uses WebRTC Session Controller JavaScript APIs to issue a request to rehydrate the session and reconnect with the WebRTC Session Controller server.
   b. The call starts. Your application user interface logic informs Bob appropriately. In this example, the soft phone rings on the browser.
10. Bob accepts the call.
11. Your application logic manages the call to its completion.

> **Note:** If the time set for the hibernation period completes with no event, the WebRTC Session Controller server closes the session. The Session ID and its data are destroyed.
>
> In this scenario, your application cannot use the session ID to restore the data. It must create a session.

### Handling Multiple Sessions

If you have defined multiple applications in WebRTC Session Controller, then it is likely that customers access more than one such WebRTC Session Controller application. As a result, multiple WebRTC Session Controller-associated sessions in the device can be active.

Where more than one session data is involved, all the associated session data is stored appropriately. Your application instances can retrieve all the data.

## The Process Workflow for Your Web Application

The process workflow to support notifications in your WebRTC application are:

1. The prerequisites to using the notification service are complete. See "About the General Requirements to Provide Notifications".
2. An account holder accesses your application on the web browser. Your application on the browser sends the **registration_Id** (also called the **deviceToken**) to the Client SDK. The Client SDK then sends it to the WebRTC Session Controller server and saves it locally.
3. To wake up a hibernating session, the WebRTC Session Controller server sends a message with the **registration_id** to the notification provider.
4. The notification provider delivers this notification to the web browser.
5. Your application on the browser is awakened. It re-establishes communication with the WebRTC Session Controller server and handles the event.

## Before You Proceed

The WSC Android SDK API, **hibernate**(), enables your applications to handle notifications related to session hibernation. For information about the supported WebRTC Session Controller Web APIs, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

Please review the following sections before you implement web client notifications in your application:

- How Your Application Saves Session Information
- Recreating the Session When the Application Page Reloads
- How WebRTC Session Controller Restores Application Data on the Same Device
- Restoring a Call Session
- Restoring a Subscription Session
- Resuming Your Application Operation

## About the General Requirements to Provide Notifications

Complete the following tasks as required for your application. Some are performed outside of your application:

- Register with the Cloud Messaging System
- Enable Your Applications to Use WebRTC Session Controller Notification Service
- Obtain the Registration ID to Allow Push Notifications
- In your application, complete all actions with respect to changes in the activity life cycle, creating, updating, and removing notifications.
- General Tasks to Implement Session Rehydration
- Handling Hibernation Requests from the Server

### Register with the Cloud Messaging System

Register your WSC installation with the appropriate system:

- Google API Console:

Create a project and obtain the following:

a. Project ID

b. API Key

For information about how to complete this task, refer to the *Google Developers Console Help* documentation.

- Apple Push Notification Service

  Set up the following:

  a. SSL certificate to communicate with the APN service.

  b. Provisioning profile for the application.

  For information about completing these tasks, refer to the *Local and Remote Notification Programming Guide* in the iOS Developer Library at

  https://developer.apple.com/library/ios/navigation/.

For information about how to complete this task, refer to the *Google Developers Console Help* documentation.

### Enable Your Applications to Use WebRTC Session Controller Notification Service

This step is performed in the WebRTC Session Controller Administration Console.

Access the **Notification Service** tab in the WebRTC Session Controller Administration Console and configure the use of WebRTC Session Controller Notification Service. For each application, enter the application setting such as the application ID, API Key, the cloud provider for the API service. For more information about completing this task, see "Creating Applications for the Notification Service" in *WebRTC Session Controller System Administrator's Guide*.

### Obtain the Registration ID to Allow Push Notifications

This step is performed within your WebRTC application.

Your application registers for push notifications from a notification provider (such as Google Cloud Messaging system or Apple Push Notification Service) and receives a **registrationId**. To register for the notification, use the Push API or Notification API, as appropriate.

- Google

  For information about how to complete this task for the Chrome browser, see the discussion about *Push Notifications on the Open Web* in the *Google Developers* documentation.

- Mozilla

  For information about how to complete this task for the Firefox browser, see the discussion about *Push API* in the *Mozilla Developer Network* documentation.

### General Tasks to Implement Session Rehydration

The WebRTC Session Controller Web SDK requires a session ID to support the rehydration of a session. To implement session rehydration in your application:

- Persist Session IDs

  To provide your customers with uninterrupted quality in service, persist the session ID value in your applications. To do so, use the various standard storage mechanisms offered by the browser platform. See "Store the Session ID".

- Use the appropriate Session ID

  Provide the same session ID that the client last successfully connected with when it hibernated.

- Set up the logic to trigger hydration for more than one session object.

  This scenario occurs when you have multiple applications defined in WSC. Your customer creates a session with each one of WSC applications in their application. In such a scenario, the client application uses as many session objects.

### Handling Hibernation Requests from the Server

When your application receives a request to hibernate from WebRTC Session Controller, provide the necessary logic to handle the user interface and other elements in your application.

See "Responding to Hibernation Requests from the Server" for information on how to set up the callbacks to the specific WebRTC Session Controller JavaScript SDK event handlers.

## Tasks that Use WebRTC Session Controller Web SDK APIs

The following tasks use WebRTC Session Controller Web SDK APIs:

- Provide the Device Token and Application Information when Creating the Session Object
- Store the Session ID
- Implement Session Hibernation and Handle its Scenarios
- Send Notifications to the Callee when Callee Client Session is in Hibernated State
- Rehydrate the Session with the Session ID

For information about the supported WebRTC Session Controller Web SDK APIs, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

### Provide the Device Token and Application Information when Creating the Session Object

When you create the **Session** object, provide the device token you received from the notification provider. In the following syntax used to create the session object, the **extPayloads** parameter is used as the extensions payload for push notification service:

```
new Session(userName, webSocketUri, successCallback, failureCallback, extHeaders,
extPayloads)
```

***Example 3–2 Providing a Device Token and Application Information when Creating a Session***

```
...
var sessionId = null;
...
var mediaOptions = {};
...
// Set up mediaOptions
...
wscSession = new wsc.Session(null, wsUri, sessionSuccessHandler, sessionFailureHandler, null,
                             {'X-MediaOptions': mediaOptions},
                             {"capability": {"appid": "oracle.wsc.samples.web",
```

```
                                        "appversion": "0.1"}, "devicetoken":
"appDeviceToken"});
```

In Example 3–2, the sample code excerpt creates an instance of the **Session** class object called *wscSession*, with:

- *username*, the first parameter has a value *null*. This code obtains the user name from the authentication handler for the session.
- *wsURI* as its WebSocket connection.
- *sessionSuccessHandler* as the callback function for a successful creation of the session.
- *sessionFailureHandler* as the callback function for a successful creation of the session.
- *null* for the *sessionId* parameter when you create the session. The Web client SDK assigns the session ID.
- The **extHeaders** in the JSON object using *mediaOptions* to specify the media options for Temasys Plug-in.
- The **extPayloads** in the JSON object that contains application information in *capability*, and the device token in *appDeviceToken*.

  Web client SDK adds the contents of **extPayloads** to the payload it sends to the WebRTC Session Controller server.

See Example 4–3 for a complete example of registering an authentication handler for the session and configuring other attributes of the newly created session.

### Store the Session ID

The **getSessionId()** method of **wsc.Session** object returns the session identifier as a String object. Retrieve the session ID and store it in your application, for example in the *sessionSuccessHandler*, the callback function for a successful creation of the session.

> **Tip:** You can create an extensible session using the **ExtensibleSession** class. Its **saveToStorage** method uses the HTML **sessionStorage** object stores data for one session. (If the browser tab closes, the data is lost.)

***Example 3–3 Storing the userName and sessionId***

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

   ...
    // Store user name in sessionStorage.
    userName = wscSession.getUserName();
    sessionStorage.setItem("userName", userName);

    // Store sessionId in sessionStorage.
    sessionStorage.setItem("sessionId", wscSession.getSessionId());
   ...
}
```

See Example 4–3 for the example function.

To persist the session ID, use the various standard storage mechanisms offered by the browser platform.

Your application uses this session ID to immediately present "Bob" (the user) with the last current state of Bob's session with your application. For more information, see the description of **wsc.Session** and **wsc.ExtensibleSession** in *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

### Implement Session Hibernation and Handle its Scenarios

When your web application is in the background, your application must send a request back to WebRTC Session Controller stating that it wants to hibernate the session. Use the appropriate method to release shared resources, invalidate timers. Store the state information necessary to restore your application to its current state, in case it is terminated at a later time.

The WebRTC Session Controller Web Client SDK provides the following method in the **wsc.Session** object.

```
hibernate(timeToLiveInSecs, onSuccess, onError)
```

Where:

- **timeToLiveInSecs**, the time (in seconds) for which the session is alive. After this time, the session cannot be hibernated.

  For **timeToLiveInSecs**, provide the maximum period for which the client session is to be kept alive on the server. All notifications received within this period are sent to the client device.

  The WebRTC Session Controller server maintains a maximum interval depending on the policy set for each type of client device. If your application sets an interval greater than this period, the server uses the policy-supported maximum interval.

- **onSuccess**, the callback for a successful hibernation. Set up a function for this callback and provide the logic your application requires.

- **onFailure**, the callback for an error in hibernating the session. Set up a function for this callback and provide the logic your application requires.

The WebRTC Session Controller server identifies the client device (going into hibernation) by the **deviceToken** you provided when building the session object. When the **hibernate** method completes, the **wscConst.SessionState** for the session is **HIBERNATED**. The session with the WebRTC Session Controller server closes. Your application can take no action, such as making a call request, until the session is resurrected.

For information about **hibernate** method, see the description about **WSCSession** in *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

### Send Notifications to the Callee when Callee Client Session is in Hibernated State

If the callee client session is in a hibernated state, any incoming event for that client session requires some time for the call setup so that the callee can accept the call. In your web application, add the logic to handle push notification event when the callee session is in a hibernated state.

In Example 3–4, the application uses the JavaScript **addEventListener** method to attach an function as the event handler to the specified element, '*push*' for the push event.

***Example 3–4 Setting up a Function to handle a Push Notification***

```
self.addEventListener('push', function(event) {
      console.log('Push Event Received');
```

```
  //Set up a notification to indicate a push event received with the msg payload
  var title = 'Push notification: Someone is calling you';
  var body = 'Please click this notification to rehydrate web if you accept the call.';
  var icon = 'imagelocation.format';
  var tag = 'ImageNametoUseAsTag';

 // Chain the waitUntil method to the event to ensure the code inside waitUntil() occurs.
  event.waitUntil(
    self.registration.showNotification(title, {
      body: body,
      icon: icon,
      tag: tag
     })
   );
 });
```

### Rehydrate the Session with the Session ID

To rehydrate an existing session, use the stored session ID that you stored earlier.

***Example 3–5 Rehydrating the Session***

```
// If sessionId is stored in SessionStorage, set up the rehydration process.
  if (sessionStorage.getItem("sessionId")) {
// SessionId is stored in SessionStorage. Reconnect and handle the call event.
    connect(sessionStorage.getItem("sessionId"), sessionStorage.getItem("userName"));
    ...
  } else {
// Cannot resurrect session. Take appropriate action.
    ...
  }
```

The connect function used by the sample excerpt above is set up as:

```
function connect(sessionId, username) {
    var wsUri = protocol.replace(/^http/, 'ws') + '//' + address + requestUri;
    var mediaOptions = {};

    if (tenant) {
// Update the wsUri with the tenant profile key.
      wsUri += '?tenant_profile_key=' + tenant;
    }
// Retrieve the Flash settings.
    if (typeof wsc_flash != 'undefined') {
      mediaOptions = getMediaOptions();
    }
    if (sessionId) {
// Session exists.
      wscSession = new wsc.Session(username, wsUri, onSuccess, onFailed, sessionId,
{'X-MediaOptions': mediaOptions});
      wscSession.isRefresh = true;
    } else {
// Create aplug-in session. Session ID is assigned on successful creation.
      wscSession = new wsc.Session(username, wsUri, onSuccess, onFailed, undefined,
{'X-MediaOptions': mediaOptions});
    }

    var authHandler = new wsc.AuthHandler(wscSession);
    authHandler.refresh = function(authType, authHeaders) {
      var authInfo = null;
```

```
    if(authType == wsc.AUTHTYPE.SERVICE) {
      authInfo = getServiceAuth(authHeaders);
    } else if (authType==wsc.AUTHTYPE.TURN) {
      console.log("Turn server auth challenge");
    }
    return authInfo;
  };
}
```

Your application can repopulate the data associated with the session by doing the following:

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

### Responding to Hibernation Requests from the Server

When the server has to force your application to hibernate, it calls the **onHibernationRequest** event handler in the **Session** interface.

To handle the user interface and other elements in your application, provide the necessary logic in your implementation of the **onHibernationRequest** callback.

***Example 3–6 Handling Server-originated Hibernation Requests***

```
...
wscSession.onHibernationRequest = function(wscSession) {
  wscSession.handleHibernateRequest(response_code, reason);
}
...
```

# Managing the Sessions in Your Application

When you use the WebRTC Session Controller JavaScript SDK 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 SDK 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.

---

## About Session Rehydration Scenarios

Many types of events occur during the life cycle of an application session. Set up your application to manage the session data appropriately. The rehydration scenarios associated with your application Session can be:

- Within the same client device.

  See "Handling Session Rehydration on the Same Client Device".

- On another device.

  See "Handling Session Rehydration When the User Moves to Another Device".

# Handling Session Rehydration on the Same Client Device

A network switchover, or a clustered server shut down affects an application session. Configure the logic in your application to recover from the event. Present the account holder with the current state of the session and its sub sessions on the same device.

## How WebRTC Session Controller Restores Application Data on the Same Device

When there is a client rehydration, a network switchover, or a clustered server shut down, the WebRTC Session Controller JavaScript SDK attempts to restore your application. To do so, it uses the *sessionId* as the key and loads the saved session data from the **sessionStorage** object associated with the browser. 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 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 SDK 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 SDK fails to load data, it calls the *failureCallback* function associated with the session creation step.

## Recreating the Session When the Application Page Reloads

Ensure that you save the session identifier for the created session in your application. To recreate the **Session** object when your application page reloads, use that session identifier for the previously created session.

In Example 3–7, an application checks the *isPageReload* variable. If 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 3–7 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 is called when the session is created successfully is not shown here. This function calls the *savePageInfo* function.

After recreating the session, your application can repopulate the data 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 the **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, define the actions to handle 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 the **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, 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. For more information, see "Extending Objects Using the wsc.extend Method".

## Restoring a Call Session

If a call is recovered following a page reload, a network switchover, or a server failover, the WebRTC Session Controller JavaScript SDK calls the **CallPackage.onResurrect** event handler in your application. The resurrected call is provided as the parameter to the callback function. To perform the actions required on the rehydrated call and resume the rehydrated call, use the callback function. See "Resuming Your Application Operation".

In Example 3–8, 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 3–8 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

When a subscription is recovered following a page reload, a network switchover, or a server failover, the WebRTC Session Controller JavaScript SDK calls the **MessageAlertPackage.onResurrect** event handler. The rehydrated subscription is provided as the parameter to the callback function. To perform the required action, use the corresponding callback function. See "Resuming Your Application Operation".

In Example 3–9, 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 current **Subscription** object.

***Example 3–9 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 about page ...
}
```

## Resuming Your Application Operation

To resume your application operation following a page reload, a network switchover, or a server failover:

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

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

# Handling Session Rehydration When the User Moves to Another Device

At times, actions taken by customers in the personal, local, or wide area network result in the application session shifting from one device to another. This happens when a

customer using your application on one device (a cellphone) moves to your application on another device (a laptop softphone that uses the same account authenticated by WebRTC Session Controller). For example, your customer Alice, accesses a web browser from a cellphone to talk about a purchase selection with Bob, a customer support representative. While Alice is on the call, she switches over to a laptop to look at the purchase selection in greater detail.

You can use the WebRTC Session Controller to configure applications that support handovers of session information between devices successfully. Your application then manages the rehydration of the session and all its data on the target device (in this example, the laptop). All handovers of application sessions from the client device take place seamlessly and maintain the quality of service.

This section described how your application can work to present the customer with the session state recreated on another device.

---

**Note:** In a device-handover scenario, WebRTC Session Controller manages the data associated with the subsessions of your application session. It keeps their states intact through the handovers that occur during the life of an application session.

The focus of the handover logic in your application is the Session within which a call, a message, or a video session is alive.

---

## About the Supported Operating Systems

You can design your applications using WebRTC Session Controller JavaScript API such that you support handover to your applications programmed for Android, and the iOS systems.

---

**Note:** For such a handover to be successful, all the following are necessary:

- WebRTC Session Controller authenticates the user name and account.
- The various operating systems support your application.
- Your application is active on the various devices belonging to a user.

---

This chapter deals with setting up your WebRTC application to support handing using the WSC JavaScript SDK. For information about supporting handovers to:

- Android applications, see "Developing WebRTC-Enabled Android Applications".
- iOS applications, see "Developing WebRTC-Enabled iOS Applications".

## Configuring WebRTC Session Controller to Support Transfer of Session Data

In a device handover, the same WebSocket sessionID is used to transfer an application session state that is active in the current client device (for example, *DeviceA-1*, a cellphone belonging to Alice) and present that state on the subsequent device (*DeviceA-4*, her laptop).

When one client uses another client's websocket sessionID to connect with WebRTC Session Controller, the server checks the value in the system property, **allowSessionTransfer**. The default value of **allowSessionTransfer** is **false**. This value

causes WebRTC Session Controller to consider the request as a hacking attack and reject the request.

In order to allow the same account holder or tenant to connect with the WebRTC Session Controller server using the same WebSocket session ID, set the startup command option **allowSessionTransfer** to **true**. For more information about how to do so, see "Supporting Session Rehydration for Device Handover Scenarios" in *WebRTC Session Controller System Administrator's Guide*.

## About the WebSocket Disconnection

When the device handover takes place, the WebSocket connection immediately closes.

The WSC signaling engine keeps the session alive for a time period specified as **WebSocket Disconnect Time Limit** in the WebRTC Session Controller Administration Console.

**Note:** If the target device fails to pick up the session within the **WebSocket Disconnect Time Limit** period, the device handover fails.

## About the Normalized Session Data User to Support Handovers

WebRTC Session Controller supports a normalized uniform session data format to transfer the session state information between the following systems.

- Web
- Android
- iOS

Example 3–10 shows the session state information sent in a sample handover request by Web SDK.

***Example 3–10 Session State Data in a Sample Handover Request***

```
{
   "sessionId":"pQAAAU+78M8vPdzNtkVUbg4nrgwAAAAD_182",
   "userName":"alice@example.com",
   "unACKedMsgQueue":{
      "lowerSeq":3,
      "upperSeq":3,
      "msgQueue":{
         "array":[
            {
               "control":{
                  "type":"message",
                  "package_type":"call",
                  "session_id":"pQAAAU+78M8vPdzNtkVUbg4nrgwAAAAD_182",
                  "subsession_id":"0a74f9cf-dd75-4bdd-8fe6-3b4886b9c365",
                  "sequence":3,
                  "correlation_id":"c2",
                  "version":"1.0"
               },
               "header":{
                  "action":"complete"
               },
               "payload":{
               }
            }
```

```
      ],
      "offset":0
   }
},
"lastOutboundSeq":3,
"lastInboundSeq":3,
"lastUnHandledInboundReq":null,
"subSessionsMap":{
   "mapSize":4,
   "entry":{
      "call":{
         "mapSize":1,
         "entry":{
            "0a74f9cf-dd75-4bdd-8fe6-3b4886b9c365":{
               "caller":"alice@example.com",
               "callee":"bob@example.com",
               "callConfig":{
                  "audioConfig":"SENDRECV",
                  "videoConfig":"SENDRECV",
                  "dataChannelConfig":null
               },
               "callState":{
                  "state":"ESTABLISHED",
                  "status":{
                     "code":null,
                     "reason":"sent complete"
                  }
               },
               "earlyMediaState":null,
               "subSessionId":"0a74f9cf-dd75-4bdd-8fe6-3b4886b9c365",
               "remoteMedias":[
                  {
                     "type":"audio",
                     "mode":"sendrecv",
                     "port":"53261"
                  },
                  {
                     "type":"video",
                     "mode":"sendrecv",
                     "port":"53261"
                  }
               ],
               "offerAnswerManager":{
                  "trickledCandidatesMap":{
                     "mapSize":0,
                     "entry":{
                     }
                  },
                  "masterState":"Ack sent",
                  "slaveState":"Initial state"
               }
            }
         }
      },
      "flash":{
         "mapSize":0,
         "entry":{
         }
      },
      "chat":{
```

```
            "mapSize":0,
            "entry":{
            }
         },
         "file_transfer":{
            "mapSize":0,
            "entry":{
            }
         }
      }
   },
   "packagesMap":{
      "mapSize":5,
      "entry":{
         "register":{
            "packageType":"register"
         },
         "call":{
            "packageType":"call",
            "trickleIceMode":"off"
         },
         "flash":{
            "packageType":"flash",
            "trickleIceMode":"off"
         },
         "chat":{
            "packageType":"chat"
         },
         "file_transfer":{
            "packageType":"file_transfer"
         }
      }
   }
}
```

## About the Handover Scenario on the Original Device

When the original device (*Device A-1*) detects a handover, the following events occur:

1. **On the original device:**

   a. Your application on *DeviceA-1* suspends the active session on the WebRTC Session Controller server. The Web Client SDK API returns the session data to your application. The application session on the original device closes.

   See "Suspending the Session on the Original Device".

   b. Your application transfers the session data (**stateInfo**) for use by the application on the device *Device A-4*.

   Your application on the original device (*Device A-1*) sends the session state information in a handover request to the Application Service (your application, a web application, or a RESTful service).

   You can configure how your application performs this task in the way that suits your environment. For example, your application can push the **stateInfo** to other device or allow the other device to pull the **stateInfo**.

   See "Sending the Handover Request with Your Session Data".

The WSC Signaling engine keeps this session alive for a time period specified as **WebSocket Disconnect Time Limit** in the WebRTC Session Controller Administration Console.

2. **On the device receiving the handover:**

   The subsequent device (*Device A-4* in our example), pulls the device handover data actively. Your application on *Device A-4* is awakened. See "Recreating the Application Session for the Handover Recipient".

## Completing the Tasks to Support Session Rehydration in Another Supported Device

This section describes the tasks to complete in your application to hand over a session to and to receive a session handed over from another device. To support session transfers and rehydration with the transferred session state information to another device, complete the following tasks:

- Suspending the Session on the Original Device.
- Sending the Handover Request with Your Session Data.
- Recreating the Application Session for the Handover Recipient.

### Suspending the Session on the Original Device

In order to implement a handover, your application on the original device *DeviceA-1* suspends the active session on the WebRTC Session Controller server. One scenario would be to set up a *handover* function and suspend the session within its logic.

> **Note:** The logic surrounding the detection of the actual device handover is beyond the scope of this document.

The WSC JavaScript API method to suspend a session is **suspend()**. This method does both of the following:

- It returns the session data in JSON format.
- It calls the **onSessionStateChange** event for **wsc.Session** in your application.

In your application logic that handles the user interface related to the handover, invoke the WSC JavaScript API method **Session.suspend()**.

***Example 3–11 Suspending a Session***

```
...
// Handover detected
function handover() {
...
    var sessionData = JSON.parse(wscSession.suspend());
...
}
```

In Example 3–11, the sample code excerpt parses the JSON string object into JSON format.

### Sending the Handover Request with Your Session Data

When the **suspend()** method completes, the session state is **SUSPENDED**. In the event handler for the **onSessionStateChange** event for **wsc.Session**, set up the information to transfer to the Application service. Convert the session data that is now in JSON

format into a JSON string object by calling the **JSON.stringify()** method. Include this JSON string object and any other relevant information in the data you send with the handover request to the application service.

In the following code excerpt shown in Example 3–12, the application normalizes the session data it received from the suspension. It then sends the session data along with other relevant information to the target device using the jQuery $.post() method.

***Example 3–12 Posting the Session Data from the Initial Device (Example)***

```
// The handover REST service is deployed on the host server
...
  var restServerBaseUrl = protocol + '//' + address + "/handoverservice/v1";
  var addSessionDataUrl = restServerBaseUrl + "/add/";
...
    $.post(addSessionDataUrl + webId, JSON.stringify(sessionData), function(){}, "text")
      .done(function() {
        console.log("Send session data to REST server successfully.");
      })
      .fail(function(evt) {
        console.warn("Failed to send session data to REST server with error", evt);
      });
...
  });
```

In Example 3–12:

- The **wsc.Session.suspend** method returns the session data parsed in JSON format.
- The user name *webId* was authenticated earlier in the sample code.
- The $.post() method sends the data to the RESTful service on the host server. The *function(){}* indicates that the application does not run any other function.

### Recreating the Application Session for the Handover Recipient

When the subsequent device also supports Web Client SDK, WebRTC Session Controller manages the interim actions such as reconnecting with the WebRTC Session Controller server, sending the invite requests, and resuming the call (in our example case).

The handover ends with the successful continuation of the ongoing session for the customer. Verify that your application state is fully maintained by addressing the effect of device handovers on all other application and user interface logic that does not involve WebRTC Session Controller APIs.

This chapter dealt with setting up your WebRTC application to support handovers using the WebRTC Session Controller JavaScript SDK. For information about supporting handovers in:

- Android applications, see "Developing WebRTC-Enabled Android Applications".
- iOS applications, see "Developing WebRTC-Enabled iOS Applications".

## Configuring Screen Sharing

You can set up your WebRTC-enabled application to support the sharing of a screen between a caller and a callee in a call session. You can share the screen between any two browsers.

## About the Requirements

Verify that the following requirements for screen sharing are satisfied and available in your environment:

- WebRTC Session Controller
- The browser-specific plug-in using the browser-provided APIs for WebRTC.

## How the Screen Sharing Process Works

When you have configured your WebRTC-enabled application for screen sharing your environment:

1. A caller establishes a WebRTC-enabled application session on one browser.

   Two Chrome browsers are involved. Both are video-enabled and compliant with the necessary elements.

   A caller (Alice) accesses your application on her browser in which your application is installed. Your application on her browser gets an alert based on the plug-in user interface you created.

2. Alice sets up a call with a callee (Bob) and Bob accepts the call.

   Bob accepts the call on the call session user interface of your application on the browser. The call session is established.

   The browser for both parties displays your application screen-sharing user interface element where the customer can make some selections.

3. Screen Sharing request from one of the parties in the call.

   One of the parties, Alice, or Bob, clicks the screen-sharing user interface element to request screen sharing.

   WebRTC Session Controller SDK sends an alert to the plug-in informing it about the start of the screen-sharing session.

   WebRTC Session Controller JavaScript SDK provides **mediaOptions**, an object which specifies the types of media to request and the requirements for each type.

   A message goes out to the other party's browser plug-in.

4. The receiver of the screen share request is alerted.

   The other party, Alice, or Bob, accepts or rejects the screen sharing request.

5. If the request to share the screen is accepted:

   a. The screen sharing begins.

   b. Screen Sharing ends or is canceled by one or the other parties in the call. Shared screens video element is null and the shared screens close. The call continues.

   c. The call ends. Shared screens video element is null and the shared screens close.

6. If the request to share the screen is rejected:

   The shared screen video element is null. Based on the user interface setup, the requester is prompted to close the video streaming in the user interface element. If the call session has not ended, the audio call continues.

## Verifying Browser Capabilities

When the customer accesses a browser in which your application is installed, WebRTC Controller JavaScript SDK verifies that your application extension is installed.

## Providing the Required User Interfaces and Event handlers

Set up your screen-sharing plug-in user interface to support the actions a customer may want to take, such as:

- Request to share the screen.
- Accept screen-sharing request.
- Decline a screen-sharing request.
- Cancel a screen-sharing request.
- End a screen-sharing session.

## Verifying the Availability of the Plug-in Interface

When the user accesses the browser where your application is installed, your application gets an alert that you configured on the plug-in user interface. Set up the logic to respond to this event.

In the excerpt in Example 3–13, WebRTC Controller JavaScript SDK gives the example application a "ping" to say that the extension is installed. In the code excerpt, *extensionInstalled* is set to *true* when the ping is received.

***Example 3–13 Verifying the Availability of the Extension***

```
// content-script sends a 'OCWSC_PING' msg if extension is installed
    if (event.data.type && (event.data.type === 'OCWSC_PING')) {
        extensionInstalled = true;
    }

    if(!extensionInstalled) {
        alert("Screen sharing plug-in is not installed");
    }
```

## Providing the logic to Initiate a Screen Sharing Session

To start screen sharing with the other party in the call, the customer makes the appropriate selection based on the design of your plug-in user interface. Set up the logic to respond to this event to send the request as a message to your plug-in on the other browser.

Set up *mediaOptions*, an object which specifies the types of media to request and the requirements for each type. You can obtain the MediaStreamConstraints object using the **getUserMedia** method:

```
getUserMedia(constraints, successCallback, errorCallback);
```

For example, your application can request the audio and camera capabilities it requires. In Example 3–14 the code specifies 1280by720 for the camera resolution.

***Example 3–14 Setting up the Media Options***

```
mediaOptions = {
    audio: false,
    video: {
```

```
        mandatory: {
            chromeMediaSource: 'desktop',
            chromeMediaSourceId: streamId,
            maxWidth: 1280,
            maxHeight: 720
        },
        optional: []
    }
};
```

# 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 *Oracle Communications 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.

## 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's requirements.

- The WebRTC Session Controller JavaScript API support libraries:
  - **wsc-common.js**
  - **wsc-call.js**

  If your application uses other supporting libraries, reference them, as well.

### 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.
var wsUri = "ws://" + window.location.hostname  + ":" + window.location.port + "/ws/webrtc/sample";
   ...
```

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 Session Object
- Setting Up the Call Package for the Session
- Handling Session State Changes
- Handling Errors Related to Sessions
- Obtaining the Callee Information

### 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 Media Stream for Calls in the CallConfig Object".

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.
    // WebRTC Session Controller can determine it using the cookie of the request.
    wscSession = new wsc.Session(null, wsUri, sessionSuccessHandler,
sessionErrorHandler);
    // Register a wsc.AuthHandler with session.
    // This handler 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 *Oracle Communications 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:<br>■ **CallConfig**, which defines the calling capability, is configured.<br>■ When the page loads, the *wscSession* object is created and configured.<br>■ The session is now in a CONNECTED state.<br>■ Controls are displayed on the application page. For the audio call, they consist of a callee input field, **Call**, **Cancel** and **Logout** buttons.<br>■ The call package is created inside the callback for the session success event handler.<br>The example code retrieves the user Id for debugging purposes. | `Created CallConfig with audio stream only.`<br><br>`Page has loaded. Setting up the Session.`<br><br>`In setSessionUp(). Session configured with authhandler, intervals and sessionStateChange handler.`<br><br>`sessionState : CONNECTED`<br><br>`In displayControls(). Waiting for Callee Input.`<br><br>`In sessionSuccesshandler. Created CallPackage..`<br><br>`Our user is bob1@example.com` | `Created CallConfig with audio stream only.`<br><br>`Page has loaded. Setting up the Session.`<br><br>`In setSessionUp(). Session configured with authhandler, intervals and sessionStateChange handler.`<br><br>`sessionState : CONNECTED`<br><br>`In displayControls(). Waiting for Callee Input.`<br><br>`In sessionSuccesshandler. Created CallPackage..`<br><br>`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.
- 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 the caller.
    callee = document.getElementById("callee").value;
    caller = userName;
    console.log ("Name entered is " + callee);

    // Check to see if user gave a valid input.
    ...
    // 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":
"YXBwLTNjOHFlaHR2eGRhcnxiYWNkMTIxMWFmZDlkNmUyMThmZmI0ZDc4ZmY3ZmY1YUAxMC4xODIuMTMuM
Th8Mzc3N2E3Nzc0ODYyMGY4",
    "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 *Oracle Communications 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 *Oracle Communications 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.<br>The call workflow starts. | For the caller (*bob1*) side, the application does the following in the *onCallSomeOne()* callback function:<br>■ Creates a *call* object with the callee's id, the configuration for calls in this browser, and the necessary call error handler function.<br>■ Sets up the general event handler to handle changes in the call.<br>■ Issues the command *call*.**start**.<br>■ Enables the controls to cancel the call before it is set up.<br>■ Defines the call and media state change handlers.<br>The browser requests the user to allow access to audio media. If the user gives permission, the local media stream is added. | `In onCallSomeOne()`<br>`Name entered is bob2`<br>`Adding string to name`<br>`Caller, bob1@example.com, wants to call bob2@example.com, the Callee.`<br>`Creating call object to call bob2@example.com`<br>`Created the call.`<br><br>`Calling setEventHandlers from onCallSomeOne() with call data.`<br>`In setEventHandlers`<br><br>`In onCallSomeOne(). Starting Call.`<br>`Enabled bob1@example.com to cancel call.`<br><br>`In mediaStreamEventHandler.`<br>`mediastate : LOCAL_STREAM_ADDED`<br><br>`In callStateChangeHandler().`<br>`callstate :`<br>`{"state":"STARTED","status":`<br>`{"code":null,"reason":"start call"}}`<br>`In callStateChangeHandler()`<br>`callstate :`<br>`{"state":"RESPONSED","status":`<br>`{"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.<br>If the user accepts the call, Signaling Engine asks the user for permission to use the microphone.<br>*When permission is given, the local and remote streams are added.* | For the callee (*bob2*) side:<br>Signaling Engine, on receiving the call invitation from the caller, triggers the function configured in the application to handle incoming calls.<br>This is the *call* object's **onIncomingCall()** callback function that was assigned in Example 4–4.<br>The application does the following:<br>■ Sets up the actions in the callback function to handle changes in the call.<br>■ 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.`<br>`Calling setEventHandlers from onIncomingCall() with callObj`<br>`In setEventHandlers`<br><br>`User Accepted the call.`<br>`In callStateChangeHandler().`<br>`callstate : {"state":"STARTED","status": {"code":null,"reason":"receive call"}}`<br>`Invoking getTurnAuthInfo`<br>`In mediaStreamEventHandler.`<br>`mediastate : LOCAL_STREAM_ADDED`<br><br>`In mediaStreamEventHandler.`<br>`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)<br>*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.<br>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"}}`<br>`In callStateChangeHandler(). callstate : {"state":"ESTABLISHED","status": {"code":null,"reason":"sent complete"}}` | `In callStateChangeHandler(). callstate : {"state":"RESPONSED","status": {"code":200,"reason":"sent success response"}}`<br>`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.
    ...
    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.<br>The call takes place.<br>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:<br>▪ Sets up the controls to enable the caller to end the call.<br>▪ Adds the remote media stream enabling the caller to hear the "Hello?"<br>On the callee's (*bob2*) side:<br>Sets up the controls to enable the callee to end the call. | In callStateChangeHandler(). callstate : {"state":"ESTABLISHED","status": {"code":null,"reason":"sent complete"}}<br>Call is established. Calling callMonitor.<br><br>In callMonitor. Monitoring the call. Setting up controls to Hang Up.<br>In mediaStreamEventHandler. mediastate : REMOTE_STREAM_ADDED | In callStateChangeHandler(). callstate : {"state":"ESTABLISHED","status": {"code":null,"reason":"got complete"}}<br>Calling callMonitor.<br><br>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.<br>*In this example, bob1, the caller, ended the call.*<br>*The console log for the caller from the callMonitor() function specifies who ended the call.*<br>*At this point note the differences in the console log entries for the caller and callee.*<br>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.<br>■ The state of the call changes to ENDED.<br>■ The local media stream for the browser is disconnected.<br>■ At this point, your application's logic may vary.<br>■ 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.`<br><br>`In callStateChangeHandler(). callstate : {"state":"ENDED","status" : {"code":null,"reason":"st op call"}} Call ended. Displaying controls again.`<br><br>`In displayControls(). Waiting for Callee Input.`<br><br>`In mediaStreamEventHandler. mediastate : LOCAL_ STREAM_REMOVED` | `In callStateChangeHandler(). callstate : {"state":"ENDED","status" : {"code":null,"reason":"st op call"}} Call ended. Displaying controls again.`<br><br>`In displayControls(). Waiting for Callee Input.`<br><br>`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.

## 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.

### 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.

## Enabling Trickle ICE to Improve Application Performance

To improve performance when Web applications negotiate connections from private networks behind Network Address Translation (NAT)-enabled routers using the

Interactive Connectivity Establishment (ICE) protocol, WebRTC Session Controller supports the draft IETF Trickle ICE specification (http://tools.ietf.org/html/draft-ietf-mmusic-trickle-ice-01).

Trickle ICE, when enabled, allows ICE host Traversal Using Relays around NAT (TURN) candidates to be exchanged incrementally rather than requiring a full protocol negotiation which can take some time depending upon network conditions.

**Note:** Firefox does not support Trickle ICE.

To enable Trickle ICE functionality, after you have instantiated a **CallPackage** object, use the **setTrickleIceMode** method with the appropriate value:

```
callPackage = new wsc.CallPackage(wscSession);
callPackage.setTricleIceMode(mode);
```

The *mode* variable is one of the following values:

- **off**: Trickle ICE is disabled (default)
- **half**: for use in cases where full Trickle ICE support cannot be assumed on the receiving endpoint
- **full**: full Trickle ICE support

## 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.

## Handling Dual Tone Multi Frequency in Calls

RFC4733 (defined in WebRTC) and SIP INFO method of the SIP protocol are two different ways of sending dual multi tone frequencies (DTMFs) when a call is made from your WebRTC application to a SIP-supporting endpoints. The touch tones in the (non-rotary) push-button phones support DTMFs.

This section describes how you can set up your WebRTC application to support calls to legacy networks where the other party is in SIP-supporting destination, by using the SIP INFO method. SIP INFO messages can send indications of DTMF audio tones between peers as part of the signaling path of the call. They separate DTMF digits from the voice stream and send them in their own signaling message.Upon receipt of a SIP INFO message with DTMF content, the gateway generates the specified DTMF tone on the receiving end of the call.

At the caller's end, your WebRTC application sets up an INFO message and sends it using the **sendInfoMessage** method of **wscCall**.

Example 4–16 shows how a WebRTC application sets up the code to do this. It extends the **wsc.Call** and **wsc.CallPackage** to support sending and receiving INFO messages.

***Example 4–16 WebRTC Application Actions to Send DTMF Information***

```
/*
 *Function that sends the DTMF
  */
function SendDTMF()  {
  wscCall.sendInfoMessage(document.getElementById("DTMF").value);
  document.getElementById("DTMF").value = ""; };

/*
 *Extending the Call Package
  */
function CallExtension() {
  console.log("In CallExtension constructor.");
  CallExtension.superclass.constructor.apply(this, arguments) }

CallExtension.prototype.onMessage = function(message) {
  console.log("In CallExtension onMessage method.");

  if (this.isInfoMessage(message)) {
    console.log("Got an INFO message...");

    var infoText = message.payload.content;
    var sender = message.header.initiator;
    var receiver = message.header.receiver;

    console.log("Message received from " + sender + " to " + receiver + " with content " + text);
  } else {
    console.log("Not an INFO message, invoke onMessage of Call.");

    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"; };

wsc.extend(CallExtension, wsc.Call);


CallExtension.prototype.sendInfoMessage = function(text) {
  console.log("Send an INFO message.");

  var target;
  var initiator = this.session.userName;
  if (initiator == this.getCaller()) {
    target = this.getCallee();
  } else {
    target = this.getCaller();
  }
  var msg = {
    "control" : {
      "type" : "message",
      "version" : "1.0",
      "sequence" : this.session.getLastOutboundSeq() + 1,
      "ack_sequence" : this.session.lastServerSequence,
      "subsession_id" : this.subSessionId,


      "package_type" : "call"
    },
    "header" : {

      "action" : "info",
      "initiator" : initiator,
      "target" : target
    },
    "payload" : {
      "content" : text
    }
  };
  wscSession.sendMessage(new wsc.Message(msg), true);

  console.log("Message sent from " + initiator + " to " + target + " with content " + text); }

var CallPackageExtension = function() {
  console.log("In CallPackageExtension constructor.");

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

CallPackageExtension.prototype.prepareCall = function(session, callConfig,
    caller, callee) {
  console.log("In CallPackageExtension, prepareCall method.");
  return new CallExtension(session, callConfig, caller, callee); };

wsc.extend(CallPackageExtension, wsc.CallPackage);

/*
 *When starting the session
  */
function sessionCreated(mSession) {
  callPackage = new CallPackageExtension(mSession);
(…)
}
```

To complete the requirements for your application to support DTMFs, you need to configure WebRTC Session Controller Signaling Engine (Signaling Engine) to process the Sip Request variable with details from the INFO message received from the WebRTC application. For information about how to do so, see "Customizing SIP Requests to Support DTMFs" in *WebRTC Session Controller Extension Developer's Guide*.

# 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 *Oracle Communications 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 APIs.

## 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 *Oracle Communications 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:

- Setting Up the General Elements for the Data Transfer Feature
- 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

## Setting Up the General Elements for the Data Transfer Feature

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

- The WebRTC Session Controller JavaScript API support libraries:
  - **wsc-common.js**
  - **wsc-call.js**

  If your application uses other supporting libraries, reference them, as well.

## 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. See "Defining Data Transfers with dataChannelConfig Parameter" for more information.

### Defining the Data Transfer in the CallConfig Object

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 a user to cancel the call before it is set up.
        // Disable "Start a Chat Session" button and enable "Cancel" button.
        // If a 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 *Oracle Communications 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:

- Include the following WebRTC Session Controller JavaScript API support libraries in the <head> element of your HTML page:
  - **wsc-common.js**
  - **wsc-msgalert.js**

If your application uses other supporting libraries, reference them, as well.

- 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

# Developing Rich Communication Services Applications

This chapter shows how you can use the Oracle Communications WebRTC Session Controller JavaScript application programming interface (API) library to develop Rich Communication Services (RCS) applications.

## About Rich Communication Services

Rich Communications Services is a system created by the Groupe Speciale Mobile Association (GSMA) which allows telecommunication providers running IP Media Subsystem (IMS) networks to extend those networks with a variety of features, including enhanced phone book capabilities, messaging options, and expanded options during calls. For more details on RCS, see http://www.gsma.com/network2020/rcs/.

## About WebRTC Session Controller RCS Support

The WebRTC Session Controller JavaScript SDK provides support for integrating RCS functionality into your WebRTC Session Controller web-based applications. For more details on any of the APIs described in this chapter, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

The following RCS services are implemented:

- **Capabilities Exchange**: Determine the capabilities of a remote endpoint, such as audio, video or file transfer support. For more information, see "Capabilities Exchange".
- **Stand Alone Messaging**: Send simple messages between two endpoints. For more information, see "Sending a Standalone Message".
- **One on One and Group Chat**: Create real-time chat sessions between one or more participants. For more information, see "Creating an RCS Chat Application".
- **File Transfer**: Transfer files of any type between two endpoints. For more information, see "Implementing File Transfer".

## Prerequisites

Before continuing, make sure you thoroughly review and understand the JavaScript API discussed in the following chapters:

- About Using the WebRTC Session Controller JavaScript API

- Setting Up Security

# About the Examples in This Chapter

The examples and descriptions in this chapter are kept intentionally straightforward in order to illustrate the functionality of the WebRTC Session Controller JavaScript API without obscuring it with user interface code and other abstractions and indirections. Since it is likely that use cases for production applications will take many forms, the examples assume no pre-existing interface schemas except when absolutely necessary, and then, only with the barest minimum of code. For example, if a particular method requires arguments such as a user name, a code example will show a plain string *username* such as "*bob@example.com*" being passed to the method. It is assumed that in a production application, you would retrieve information using some sort of form-based user interface.

---

**Note:** The examples in this chapter are not intended to be "plug-and-play" examples; error checking and security are ignored in favor of concept illustration.

---

# Capabilities Exchange

WebRTC Session Controller implements support for RCS capabilities exchange which lets two endpoints describe their capabilities such as video streaming, audio streaming, and file transfer so that they can negotiate support within a shared application session.

## Sample Capability Exchange HTML File

In Example 8–1, the sample HTML file contains three check boxes which let you set local client "capabilities," a button and an associated input text box to submit query enquiries. Two div elements serve as targets for displaying capabilities from the remote host, **queryResult**, and for showing incoming query requests, **queryFromResult**.

---

**Note:** The capabilities in this example are simple arbitrary strings and do not actually represent the capabilities of the host browser. They are provided for example purposes only. For information on RCS capabilities strings, see http://www.gsma.com/network2020/rcs/specs-and-product-docs/.

---

The required SDK files that must be included for this sample are:

- **wsc-common.js**: Shared common library utility functions.
- **wsc-capability.js**: Capabilities SDK functions.

***Example 8–1 Capability Exchange Sample HTML File***

```
<!DOCTYPE HTML>
<html>
<head>
    <title>WebRTC Session Controller Capabilties Exchange Example</title>
    <script type="text/javascript" src="/api/wsc-common.js"></script>
    <script type="text/javascript" src="/api/wsc-capability.js"></script>
</head>
```

```
<body>
  <h1>WebRTC Session Controller Capabilties Exchange Example</h1>
  <div id="mainPanel">
      <div class="container" id="myCapabilities">
        <p>My capabilities:</p>
        <input type="checkbox" id="audio" checked>IM/Chat (IM)
        <input type="checkbox" id="video" checked>Video share (VS)
        <input type="checkbox" id="file">File transfer (FT)
      </div>

      <div class="container" id="query">
        <button id="queryBtn">Query: </button>
        <input type="text" id="target" size="40">
        <div id="queryResult" class="content"></div>
      </div>

      <div class="container" id="queryFrom">
        <p>Capabilties query received from: </p>
        <div id="queryFromResult" class="content"></div>
      </div>
  </div>
</body>
</html>
```

## Initiate a Capability Exchange Query

In Example 8–2, the following occurs:

1. A set of global variables are initialized for a variety of utility values.
2. A new **capabilityPackage** object is instantiated using the current Session as an argument.
3. A new **capabilityExchange** object is instantiated using the **capabilityPackage** object's **createCapabilityExchange** method.
4. The event handler, **onQueryRequest**, is bound to the **capabilityExchange** object's **onQueryRequest** listener. This event handler processes the capabilities of the remote host. See Example 8–4.
5. The event handler, **onQueryResponse**, is bound to the **capabilityExchange** object's **onQueryResponse** listener. This event handler processes queries from the remote host. See Example 8–3.
6. The event handler, **onErrorResponse**, is bound to the **capabilityExchange** object's **onErrorResponse** listener. This event handler is triggered when any errors occur during the capabilities exchange. See Example 8–5.
7. The **queryCapability** utility function is bound to the click event of the query button, **queryBtn**, and the function **getMyCapabilties** reads the state of each of the three check boxes and creates an array of values.

***Example 8–2 Initiating a Capability Exchange Query***

```
var
  capabilityPackage,
  capabilityExchange,
  queryCounter = 0,
  queryFromCounter = 0,
  _A = "IM/Chat",
  _V = "Video share",
  _F = "File transfer";
```

```
capabilityPackage = new wsc.CapabilityPackage(wscSession);
capabilityExchange = capabilityPackage.createCapabilityExchange();
capabilityExchange.onQueryResponse = onQueryResponse;
capabilityExchange.onQueryRequest = onQueryRequest;
capabilityExchange.onErrorReponse = onErrorResponse;

var queryButton = document.getElementById("queryBtn");
queryButton.addEventListener("click", queryCapability);

function queryCapability() {
  var target = document.getElementById("target").value.trim();
  console.log("Query capability of:", target);
  capabilityExchange.enquiry(getMyCapabilities(), target);
}

function getMyCapabilities() {
  var myCs = [];
  if (document.getElementById("audio").checked) {
    myCs.push(_A);
  }
  if (document.getElementById("video").checked) {
    myCs.push(_V);
  }
  if (document.getElementById("file").checked) {
    myCs.push(_F);
  }
  return myCs.join(";");
}
```

The actual query itself is sent using the **enquiry** method of the **capabilityExchange** object.

## Handle a Capability Query Response

You define the **onQueryResponse** event handler to process query responses from remote hosts. In Example 8–3, the variable, capabilities, is returned from the incoming query object's **targetCapability** method. If capabilities exists, the array is unwrapped using a for loop, and the various capabilities are parsed and displayed along with the target name using the **queryResult** div element as a container. The **queryCounter** variable is used to prevent naming collisions during multiple requests.

***Example 8–3 onQueryResponse Sample Code***

```
function onQueryResponse(query) {
  var capabilities = query.targetCapability;
  var from = query.target;

  var queryResultEle = document.getElementById("queryResult");
  queryResultEle.innerHTML = queryItemEle;
  document.getElementById("queryName" + queryCounter).textContent = from + ": ";

  if (capabilities) {
    var cArray = capabilities.split(";");
    for (var i = 0; i < cArray.length; i++) {
      var c = cArray[i];
      if (c == _A) {
        document.getElementById("queryCoa" + queryCounter).className += " cEnable";
        showElement("queryCoa"+ queryCounter);
```

```
      } else if (c == _V) {
        document.getElementById("queryCov" + queryCounter).className += " cEnable";
        showElement("queryCov"+ queryCounter);
      } else if (c == _F) {
        document.getElementById("queryCof" + queryCounter).className += " cEnable";
        showElement("queryCof"+ queryCounter);
      } else {
        console.log(from, "has capability:", c);
      }
    }
  }
}
```

## Handle an Incoming Capability Query

You define the **onQueryRequest** event handler to process incoming query requests. In Example 8–4, span elements are concatenated together using the **queryFromCounter** value to make sure that they are distinct between multiple queries. The span elements are bound to the **queryFromResult** content div element, and the same for loop from Example 8–3 is used to display each capability entry if it is defined in the capabilities array. Finally, the **capabilityExchange** object's **respond** method is called to return the local capabilities to the remote endpoint.

***Example 8–4 onQueryRequest Sample Code***

```
function onQueryRequest(query) {
    var capabilities = query.initiatorCapability;
    var from = query.initiator;

    queryFromCounter = queryFromCounter + 1;

    var nameS = '<span id="queryFromName' + queryFromCounter + '"></span>  ';
    var coaS = '<span id="queryFromCoa' + queryFromCounter + '"
                                        hidden="true">IM</span>  ';
    var covS = '<span id="queryFromCov' + queryFromCounter + '"
                                        hidden="true">VS</span>  ';
    var cofS = '<span id="queryFromCof' + queryFromCounter + '"
                                        hidden="true">FT</span>';
    var queryFromItemEle = nameS + coaS + covS + cofS;

    var queryFromResultEle = document.getElementById("queryFromResult");
    queryFromResultEle.innerHTML = queryFromItemEle;
    document.getElementById("queryFromName" + queryFromCounter).textContent = from + ": ";

  if (capabilities) {
    var cArray = capabilities.split(";");
    for (var i = 0; i < cArray.length; i++) {
      var c = cArray[i];
      if (c == _A) {
        document.getElementById("queryFromCoa" + queryFromCounter).className += " cEnable";
        showElement("queryFromCoa"+ queryFromCounter);
      } else if (c == _V) {
        document.getElementById("queryFromCov" + queryFromCounter).className += " cEnable";
        showElement("queryFromCov"+ queryFromCounter);
      } else if (c == _F) {
        document.getElementById("queryFromCof" + queryFromCounter).className += " cEnable";
        showElement("queryFromCof"+ queryFromCounter);
      } else {
        console.log(from, "has capability:", c);
      }
```

```
    }
  }
  capabilityExchange.respond(query, getMyCapabilities());
}
```

## Handle Capability Exchange Errors

You define the **onErrorResponse** event handler to process error conditions.

***Example 8–5 onErrorResponse Sample Code***

```
function onErrorResponse (error) {
  console.log("Error action: "+error.action);
  console.log("Error code: "+error.errorCode);
  console.log("Error reason: "+error.errorReason);
}
```

## Initiate a Capability Exchange Request in a Call

To initiate a capability exchange request in an active call, you simply initiate the **capabilityPackage** using a **Call** object instead of a session object:

***Example 8–6 Initiating a Capability Exchange in a Call***

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

var myCall = callPackage.createCall("alice@example.com", callConfig);
myCall.start();

capabilityPackage = new wsc.CapabilityPackage(myCall);
capabilityExchange = capabilityPackage.createCapabilityExchange();
capabilityExchange.onQueryResponse = onQueryResponse;
capabilityExchange.onQueryRequest = onQueryRequest;
capabilityExchange.onErrorReponse = onErrorResponse;

var queryButton = document.getElementById("queryBtn");
queryButton.addEventListener("click", queryCapability);

// Continue with your workflow...
```

# Sending a Standalone Message

The RCS standard defines a simple way an application can send text messages between two endpoints, which is implemented in WebRTC Session Controller in the wsc.MessagingPackage class.

## Messaging Sample HTML File

The sample HTML file for messaging examples contains the following elements:

- A div element, **statusArea**, used to display application status messages.
- A form input text box, **msgTarget**, used to input a message recipient's ID.
- A div element, **history**, used as a container to store message history.
- A form input text box, **msgContent**, used to input a message for the recipient.

- A form input button, **msgSend**, used to send the content of the **msgContent** text box to the recipient.

The required SDK files that must be included for this sample are:

- **wsc-common.js**: Shared common library utility functions.
- **wsc-messaging.js**: Messaging SDK functions.

***Example 8–7 Messaging Sample HTML File***

```
<!DOCTYPE html>
<html>
  <head>
    <title>WebRTC Session Controller Messaging Example</title>
    <script type="text/javascript" src="/api/wsc-common.js"></script>
    <script type="text/javascript" src="/api/wsc-messaging.js"></script>
  </head>
  <body>
    <h1>WebRTC Session Controller Messaging Example</h1>
    <div id="statusArea"></div>
    <h4>Recipient:</h4>
    <p><input type="text" name="msgTarget" id="msgTarget" size="40"/></p>
    <div id="history"></div>
    <h4>Message Content:</h4>
    <p>
      <input type="text" name="msgContent" id="msgContent" size="50"/>
      <input type="button" name="msgSend" id="msgSend" value="Send"/>
    </p>
  </body>
</html>
```

## Send a Message

In Example 8–8, the following occurs:

1. A new **messagePackage** object is instantiated using the current Session as an argument.
2. A new **messaging** object is instantiated using the **messagePackage** object's **createMessaging** method.
3. The event handler, **onNewMessage**, is bound to the **messaging** object's **onNewMessage** listener. This event handler processes incoming messages. See Example 8–9.
4. The event handler, **onSuccessResponse**, is bound to the **messaging** object's **onSuccessResponse** listener. This event handler processes success responses. See Example 8–10.
5. The event handler, **onErrorResponse**, is bound to the **messaging** object's **onErrorResponse** listener. This event handler is triggered when any errors occur during the message exchange. See Example 8–11.
6. The **sendMsg** utility function is bound to the click event of the button, **msgSend**, and the contents of the **msgTarget** edit box is read, and assigned as the message recipient, unless the edit box is empty, in which case the **statusArea** div element is updated with an error.

***Example 8–8 Sending a Message***

```
messagePackage = new wsc.MessagingPackage(wscSession);
```

```
messaging = messagePackage.createMessaging();
messaging.onNewMessage = onNewMessage;
messaging.onSuccessResponse = onSuccessResponse;
messaging.onErrorResponse = onErrorResponse;

document.getElementById("msgSend").onclick = function() {
  sendMsg();
};

function sendMsg() {
  var msg = document.getElementById("msgContent").value;
  var target = document.getElementById("msgTarget").value;
  if (msg && msg != "" && target && target != "") {
    var msgId = messaging.send(msg, target);
    console.log("The sent message ID is: " + msgId);
  } else if (target == "") {
    document.getElementById("statusArea").innerHTML = "Please enter a recipient.";
  }
}
```

## Handle an Incoming Message

You define the **onNewMessage** event handler to process incoming messages. When a new message comes in, use the **accept** method to accept the message or the **reject** method to reject it. In Example 8–9, depending on the value of the **msgRejected** boolean which would be handled in additional user interface code you would supply, the message is either accepted or rejected, and, if accepted, the message is added to the **history** div element using the **updateHistory** utility function. A message rejection returns an SIP 603 error with the status Decline.

***Example 8–9 onNewMessage Sample Code***

```
function onNewMessage(chatMessage) {
  var
    initiator = chatMessage.initiator,
    msg = chatMessage.content,

  if (msgRejected) {
    messaging.reject(chatMessage, 603, "Decline");
  } else {
    messaging.accept(chatMessage);
    updateHistory(initiator, msg, true);
  }
}

function updateHistory(initiator, msg, newMessage) {
  var history = document.getElementById("history");
  var d = new Date();
  var ds = d.toLocaleTimeString();
  var newMsg = "(" + ds + ") " + initiator + ": " + msg;
  if (newMessage) {
    newMsg = "<div id='inChatMessage'>" + newMsg + "</div>";
    document.getElementById("msgTarget").value = initiator;
  } else {
    newMsg = "<div id='outChatMessage'>" + newMsg + "</div>";
  }
  history.innerHTML = history.innerHTML + newMsg;
};
```

## Handle Messaging Success Events

You define the **onSuccessResponse** event handler to process a successful message transmission, and update the **statusArea** div element accordingly.

***Example 8–10 onSuccessResponse Sample Code***

```
function onSuccessResponse(message) {
  var content = message.content;
  document.getElementById("statusArea").innerHTML = "Send message \"" + content
                                                    + "\" succeeded.";
}
```

## Handle Messaging Error Events

You define the **onErrorResponse** event handler to process a message transmission failure and update the **statusArea** div element accordingly.

***Example 8–11 onErrorResponse Sample Code***

```
function onErrorResponse(message, extHeaders) {
  var content = message.content;
  document.getElementById("statusArea").innerHTML = "Send message \"" + content
                                                    + "\" failed.";
}
```

# Creating an RCS Chat Application

The WebRTC Session Controller JavaScript SDK lets you implement a one to one or one to many chat application as defined by the RCS specification. For more information on the RCS chat specification, see http://www.gsma.com/network2020/rcs/specs-and-product-docs/.

## Chat Sample HTML File

The sample HTML file for chat examples contains the following elements:

- A div element, **statusArea**, used to display application status messages.
- A form input text box, **target**, used to input a chat request recipient's ID.
- A form input text box, **participants**, used to add additional recipients to a group chat.
- A form input button, **chatButton**, used to initiate a chat session request.
- A form input button, **endChatButton**, used to terminate a chat session.
- A div element, **history**, used as a container to display chat message history.
- A form input text box, **msgContent**, used to enter a chat message.
- A form input button, **msgSend**, used to send the content of the **msgContent** text box to the recipient.

The required SDK files that must be included for this sample are:

- **wsc-common.js**: Shared common library utility functions.
- **wsc-chat.js**: Chat SDK functions.

***Example 8–12 Chat Sample HTML File***

```
<!DOCTYPE HTML>
<html>
<head>
  <title>WebRTC Session Controller Chat Example</title>
  <script type="text/javascript" src="/api/wsc-common.js"></script>
  <script type="text/javascript" src="/api/wsc-chat.js"></script>
</head>
<body>
  <div id="mainPanel">
    <div id="statusArea"></div>
    <br/>
    <p>To: <input type='text' name='target' id='target' size='30'/></p>
    <p>Add participants: <input type='text' name='participants' id='participants' size='70'/></p>
    <p>
      <input type='button' name='chatButton' id='chatButton' value='Chat'
                                                          onclick='startChat()'/> 
      <input type='button' name='endCallButton' id='endChatButton' value='End Chat'
                                                          onclick='end()'/>
    </p>
    <div id='history'></div>
    <p>
      <input type='text' name='msgContent' id='msgContent'/> 
      <input type='button' name='msgSend' id='msgSend' value='Send'/>
    </p>
  </div>
</body>
</html>
```

## Implementing Chat

To determine whether a remote endpoint is a WebRTC compliant browser or an RCS compliant application, you can configure a capabilities exchange as described in "Capabilities Exchange". For browser to RCS client, the WebRTC Session Controller JavaScript SDK uses Message Session Relay Protocol (MSRP) and Session Description Protocol (SDP) to negotiate with the target and configure the WebSocket connection with WebRTC Session Controller Media Engine (Media Engine). For browser to browser chat sessions, your application can use the CallPackage data channel functions as described in "Setting Up Data Transfers in Your Applications".

### Initiate the Chat Session

In Example 8–13, the following occurs:

1. Global variables for a chat object, **wscChat**, and a chat package, **chatPackage** are initialized and the contents of the target text box are retrieved and assigned to the **target** variable.
2. A new **chatPackage** object is instantiated using the current Session as an argument.
3. The event handler, **onIncomingChat**, is bound to the **chatPackage** object's **onIncomingChat** listener. This event handler processes incoming chat requests. See Example 8–15.
4. The target variable is tested to make sure it is not zero length, and a **chatConfig** object, along with an empty array of **acceptTypes** are declared.

5. The **acceptTypes** array is initialized, the **chatConfig.setMaxSize** instance variable is initialized, as is the **chatConfig.acceptTypes** instance variable is initialized with the **acceptTypes** array. The possible values for a chatConfig object are as follows:
   - **acceptTypes**: An array of media types the endpoint is willing to accept. May contain zero or more media types or a wildcard, "*".
   - **acceptWrappedTypes**: An array of media types that an endpoint is willing to receive in an MSRP message with multipart content. May not be used as the outermost type of the message. May contain zero or more media types or a wildcard, "*".
   - **maxSize**: A whole numeric value indicating the maximum message size specified in octets the endpoint is capable of receiving.
6. A chat object, **wscChat**, is created using the **chatPackage** object's **createChat** method with the **target** as an argument.
7. The event handler, **onStateChange**, is bound to the **chat** object's **onStateChange** listener. This event handler handles changes in the state of the chat object. See Example 8–16.
8. The event handler, **onConnectionStateChange**, is bound to the **chat** object's **onConnectionStateChange** listener. This event handler handles changes in the connection state of the chat object. See Example 8–17.
9. The event handler, **onChatMessage**, is bound to the **chat** object's **onChatMessage** listener. This event handler handles incoming chat messages. See Example 8–18.
10. The event handler, **onMessageSendSuccess**, is bound to the **chat** object's **onMessageSendSuccess** listener. This event handler handles successful message transmission events. See Example 8–19.
11. The event handler, **onMessageSendFailure**, is bound to the **chat** object's **onMessageSendFailure** listener. This event handler handles failed message transmission events. See Example 8–20.
12. The event handler, **onMessageTyping**, is bound to the **chat** object's **onMessageTyping** listener. This event handler is triggered when the remote party is actively typing. See Example 8–21.
13. The event handler, **onMessageTypingStop**, is bound to the **chat** object's **onMessageTypingStop** listener. This event handler is triggered when the remote party stops typing. See Example 8–22.
14. Add additional chat participants to the chat using the chat object's **addParticipants** method.
15. Toggle the security of the chat session's transport layer using the chat object's **setSecure** method.
16. Finally, the **chat** object's **start** method is used to start the chat session taking the **chatConfig** object as an argument.

***Example 8–13 Initiating the Chat Session***

```
var
    wscChat,
    chatPackage;
    target = document.getElementById("target").value;

chatPackage = new wsc.ChatPackage(wscSession);
chatPackage.onIncomingChat = onIncomingChat;
```

```
if (target != "") {
  var
     chatConfig = {},
     acceptTypes = [];

  acceptTypes.push('text/plain');
  acceptTypes.push('message/cpim');

  chatConfig.selfMaxSize = 1024;
  chatConfig.acceptTypes = acceptTypes;

  wscChat = chatPackage.createChat(target);

  wscChat.onStateChange = onStateChange;
  wscChat.onConnectionStateChange = onConnectionStateChange;
  wscChat.onChatMessage = onChatMessage;
  wscChat.onMessageSendSuccess = onMessageSendSuccess;
  wscChat.onMessageSendFailure = onMessageSendFailure;
  wscChat.onMessageTyping = onMessageTyping;
  wscChat.onMessageTypingStop = onMessageTypingStop;

  wscChat.start(chatConfig);
}
```

Within the chat session, use the following methods as required:

- **send**: Send a message in the chat session. See "Send a Chat Message" for more information.
- **end**: End the chat session.

Once the chat session is ended using the **wsc.Chat** object **end** method, use the **wsc.ChatPackage close** method to terminate all sessions and release resources.

## Send a Chat Message

Send a chat message to a target address using the Chat object's send method.

In Example 8–14, the form send button, **msgSend**, has its **onclick** event bound to an anonymous function that retrieves the text value from the **msgContent** form text box, and, if the value is not empty, passes it to the **sendChatMessage** function.

The **sendChatMessage** function checks to see if the message is a Common Profile for Instant Messaging (CPIM) format and creates the payload accordingly, or creates a plain text payload if not.

> **Note:** The **isCpimMessage** as well as the values of the **cpim** variable must be set and retrieved using your own mechanisms in your application.

Finally, the message is sent using the Chat objects send method, and a utility function, **updateHistory** is called to append the message along with a date and message initiator to the **history** content div element.

***Example 8–14 Sending a Chat Message***

```
document.getElementById("msgSend").onclick = function() {
  var msg = document.getElementById("msgContent").value;
  if (msg && msg != "") {
```

```
    sendChatMessage(msg);
  }
};

function sendChatMessage(msg) {

  var chatMessage;

  if (isCpimMessage) {
    var cpim = "From: alice@example.com\r\n" +
               "To: bob@example.com\r\n" +
               "DateTime: 2015-12-08T00:00:00-00:00\r\n" +
               "Content-Type: text/plain\r\n" +
               "Content-Length: 1\r\n\r\n" +
               msg;
    chatMessage =
        {
          contentType : 'message/cpim',
          content : cpim
        };
  } else {
    chatMessage =
    {
      contentType : 'text/plain',
      content : msg
    };
  }

  wscChat.send(chatMessage);

  updateHistory(wscSession.userName, msg);

  function updateHistory(initiator, msg) {
    var
      d = new Date(),
      ds = d.toLocaleTimeString(),
      title = "(" + ds + ") " + initiator + "\r\n:";
      title = "<div id='outChatMessage'>" + title + "</div>";

    var newMsg = title + msg;
    document.getElementById("history").innerHTML += newMsg;
  }
}
```

## Handle Incoming Chat Requests

Define the **onIncomingChat** event handler to process incoming chat session requests.

When responding to an incoming chat session request, use the following methods as required:

- **accept**: Accept the chat invitation.
- **decline**: Decline the chat invitation.
- **getInitiator**: Return the initiator of the chat session request.

In Example 8–15, the **statusArea** div element is used as the target of a **showRequest** function that creates a status message and an interface that allows a user to accept or decline the chat invitation. The status message and user interface are then displayed in the **statusArea** div element. If the session is accepted, the chat event handlers are

initialized in the same manner as Example 8–13.

***Example 8–15 onIncomingChat Sample Code***

```
function onIncomingChat(chat) {
  document.getElementById("statusArea").innerHTML = showRequest(chat);

  document.getElementById("acceptButton").onclick = function() {
    chat.accept(chatConfig);
  };
  document.getElementById("declineButton").onclick = function() {
    chat.decline();
  };

  function showRequest(chat) {
    var
      initiator = chat.getInitiator(),
      message = initiator + " is requesting a chat session.";

    var display = message +
      "<input type='button' name='acceptButton' id='acceptButton' value='Accept' onclick=''/>" +
      "<input type='button' name='declineButton' id='declineButton' value='Decline' onclick=''/>";
    return display;
  }

  chat.onStateChange = onStateChange;
  chat.onConnectionStateChange = onChatConnectionStateChange;
  // Continue configuring chat callbacks...
}
```

## Handle Chat Signaling State Changes

Define the **onStateChange** event handler to process changes in the chat signaling state. The **wsc.CALLSTATE** enum defines the possible call states. For a complete list of **wsc.CALLSTATE** values see the *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

***Example 8–16 onStateChange Sample Code***

```
function onStateChange(chat, callState) {
  console.log("Chat state: " + callState.state);
  switch (callState.state) {
    case wsc.CALLSTATE.ESTABLISHED:
      // Handle an established call (chat) state as required...
      break;
    case wsc.CALLSTATE.UPDATED:
      // Handle an updated call (chat) state as required...
      break;
    case wsc.CALLSTATE.UPDATE_FAILED:
      // Handle an update failed call (chat) state as required...
      break;
    case wsc.CALLSTATE.ENDED:
      // Handle an ended call (chat) state as required...
      break;
    case wsc.CALLSTATE.FAILED:
      // Handle a failed call (chat) state as required...
      break;
    default:
      break;
  }
```

```
}
```

## Handle Chat Connection State Changes

Define the **onConnectionStateChange** to process changes in chat connection state over MSRP. The **ConnectionStateEnum** defines the possible connection states.

***Example 8–17 onConnectionStateChange Sample Code***

```
function onConnectionStateChange(state) {
  console.log("Chat state: " + state.state);
  switch (state.state) {
    case wsc.ConnectionStateEnum.INIT:
      // Handle the INIT state...
      break;
    case wsc.ConnectionStateEnum.ESTABLISHED:
      // Handle the ESTABLISHED state...
      break;
    case wsc.ConnectionStateEnum.ERROR:
      // Handle the ERROR state...
      break;
    case wsc.ConnectionStateEnum.CLOSED:
      // Handle the CLOSED state...
      break;
  }
}
```

## Handle Incoming Chat Messages

Define the **onChatMessage** event handler to process incoming chat messages. In Example 8–18, after initializing a set of variables, the function examines the content type of the message, **cType**, to see whether it is in a CPIM or plain text, and sets the **textContent** variable accordingly. It next pulls the initiating user name from either the MSRP message or the JSON header. It replaces angle brackets with their named elements in order to preserve the integrity of the HTML, and then calls the **updateHistory** function to append an entry to the history **content** div element, including a date.

***Example 8–18 onChatMessage Sample Code***

```
function onChatMessage(msg) {
  console.log("Received a chat message: " + msg.content);
  var
    content = msg.content,
    textContent = null,
    cType = msg.contentType,
    initator = null;

  if (cType == 'message/cpim') {
    initator = extractText(content, "From:", "\r\n");
    var textContent = content.substring(content.indexOf("Content-Type"));
    var startIndex = textContent.indexOf("\r\n\r\n");
    textContent = textContent.substring(startIndex);
    if (textContent.indexOf("\r\n\r\n") == 0) {
      textContent.replace("\r\n\r\n", "");
    }
  } else {
    textContent = content;
  }
```

```
  if (!initator) {
    if (wscChat.getInitiator() === wscChat.session.getUserName()) {
      initator = wscChat.getTarget();
    } else {
      initator = wscChat.getInitiator();
    }
  }

  textContent = textContent.replace("<", "&lt;");
  textContent = textContent.replace(">", "&gt;");

  updateHistory(initator, textContent, true);

  function updateHistory(initiator, msg) {
    var
      d = new Date(),
      ds = d.toLocaleTimeString(),
      title = "(" + ds + ") " + initiator + "\r\n:";

    title = "<div id='inChatMessage'>" + title + "</div>";
    var newMsg = title + msg;

    document.getElementById("history").innerHTML += newMsg;
  }
}
```

## Handle Message Transmission Success and Failure Events

Define the **onMessageSendSuccess** event handler to process notification of a successful message transmission. In Example 8–19, the only result is a log notification, but you might wish to update a status area or provide a more dynamic notification system.

***Example 8–19 onMessageSendSuccess Sample Code***

```
function onMessageSendSuccess(msgId) {
  console.log("Message successfully sent. ID: " + msgId);
}
```

Define the **onMessageSendFailure** event handler to process notification of a failed message transmission. As with Example 8–19, the only result is a log notification, but, again, you may wish to create a customized notification in your own application.

***Example 8–20 onMessageSendFailure Sample Code***

```
function onMessageSendFailure(msgId) {
  console.log("Message transfer failed. ID: " + msgId);
}
```

## Handle Participant Typing Notifications

Optionally, you can define **onMessageTyping** and **onMessageTypingStop** event handlers to process changes in user interface state when tracking a remote user's data input. In Example 8–21 and Example 8–22, only log messages are generated, but you could also update your chat history window or other user interface device with the changing statuses.

***Example 8–21 onMessageTyping Sample Code***

```
function onMessageTyping() {
  console.log(wscChat.getTarget() + " is typing...");
}
```

***Example 8–22 onMessageTypingStop Sample Code***

```
function onMessageTypingStop() {
  console.log(wscChat.getTarget() + " has stopped typing...");
}
```

# Implementing File Transfer

The WebRTC Session Controller JavaScript SDK lets you implement a one to one or one to many file transfer application as defined by the RCS specification. For more information on the RCS chat specification, see http://www.gsma.com/network2020/rcs/specs-and-product-docs/.

**Note:** Multiple file transfer in a single session is not supported.

## File Exchange Example HTML File

The sample HTML file for messaging examples contains the following elements:

- A div element, **statusArea**, used to display application status messages.
- A form text input box, **target**, into which a recipient address is entered.
- A file input button, **selectFilesButton**, used to select files to transfer to a remote party.
- A button input, **filesButton**, used to start a file transfer, the **onclick** event of which is bound to a **sendFile** function.
- A button input, **endFtButton**, used to cancel a file transfer, the **onclick** event of which is bound to the **endFt** function.
- A button input, **cleanButton**, used to hide the file transfer progress bar, the **onclick** event of which is bound to the **hideFileTransferProgress** function.
- A form input button, **msgSend**, used to send the content of the **msgContent** text box to the recipient.

The required SDK files that must be included for this sample are:

- **wsc-common.js**: Shared common library utility functions.
- **wsc-filetransfer.js**: Messaging SDK functions.

***Example 8–23 File Transfer HTML Sample***

```
<!DOCTYPE HTML>
<html>
<head>
  <title>WebRTC Session Controller File Transfer Example</title>
  <script type="text/javascript" src="/api/wsc-common.js"></script>
  <script type="text/javascript" src="/api/wsc-filetransfer.js"></script>
</head>
<body>
  <div id="mainPanel">
    <div id="statusArea"></div>
```

```
    <br/>
    <p>To: <input type='text' name='target' id='target' size='30'/></p>
    <p>
      <input type='file' multiple name='selectFileButton' id='selectFileButton'/>
      <input type='button' name='filesButton' id='fileButton' value='Send'
                                          onclick='sendFile()'/> 
      <input type='button' name='endFtButton' id='endFtButton' value='Cancel transfers'
                                          onclick='endFt()'/> 
      <input type='button' name='cleanButton' id='cleanButton' value='Clear progress'
                                          onclick='hideFileTransferProgress()'/>
    </p>
    <div id="messageArea">
    </div>
  </div>
</body>
</html>
```

## Setup a File Transfer Session

In Example 8–13, the following occurs:

1. Global variables for a file transfer object, **wscFileTransfer**, a file transfer package, **fileTransferPackage**, and a **fileConfig** array are initialized and the contents of the target text box are retrieved and assigned to the **target** variable.
2. A new **fileTransferPackage** object is instantiated using the current Session as an argument.
3. The event handler, **onFileTransfer**, is bound to the **fileTransferPackage** object's **onFileTransfer** listener. This event handler processes incoming file transfer requests. See Example 8–26.
4. A message transfer object, **wscFileTransfer**, is created using the **fileTransferPackage** object's **createFileTransfer** method with the **target** as an argument.
5. The event handler, **onStateChange**, is bound to the **wscFileTransfer** object's **onStateChange** listener. This event handler handles changes in the state of the file transfer object. See Example 8–16.
6. The event handler, **onSessionStateChange**, is bound to the **wscFileTransfer** object's **onSessionStateChange** listener. This event handler handles changes in the session state of the file transfer object. See Example 8–17.
7. The event handler, **onFileData**, is bound to the **wscFileTransfer** object's **onFileData** listener. This event handler handles incoming file data. See Example 8–18.
8. The event handler, **onProgress**, is bound to the **wscFileTransfer** object's **onProgress** listener. This event handler handles file transfer progress events. See Example 8–19.
9. The event handler, **onFileTransferSuccess**, is bound to the **wscFileTransfer** object's **onFileTransferSuccess** listener. This event handler handles successful file transmission events. See Example 8–20.
10. The event handler, **onFileTransferFailure**, is bound to the **wscFileTransfer** object's **onFileTransferFailure** listener. This event handler handles failed file transmission events. See Example 8–20.
11. The **fileConfig** object is initialized. The **file** field is the actual instance of the file itself. The **props** object can have the following properties:

- **name**: A string containing the file name.
- **size**: An integer indicating the file size in octets.
- **type**: A string indicating the MIME type of the file.
- **hashes**: An array containing the hash computation of the file: {algorithmName: "value: *xxxx*"}.
- **disposition**: A string with value **render** or **attachment**. The disposition value tells the receiving endpoint how to handle the file. The value **render** indicates that the file should be automatically rendered by the endpoint, for example a GIF or JPEG image file. The value **attachment** indicates that the file should not be rendered and should be treated as a downloadable attachment, for example an EXE or other such BLOB. If the disposition attribute is not specified, **render** is implied.
- **description**: A string description of the file.
- **creationTime**: A string containing the file creation date and time.
- **modificationTime**: A string containing the file modification date and time.
- **readTime**: A string containing the time and date the file was last read.
- **icon**: A string containing the Content ID URL (cid:content-id) for the file. In the case of images, usually renders as a file icon.
- **startOffset**: A string indicating the octet position in the file where the file transfer should start. The first octet of a file is indicated by the ordinal number 1.
- **stopOffset**: A string indicating the octet position in the file where the file transfer should end, including the specified octet. If the total file size is not known, use the "*" wildcard.
- **direction**: A string indicating the transfer direction for the file. To push a file, use **send** and to pull a file use **receive**.

12. Add additional file transfer recipients to the **fileTransfer** session using the **fileTransfer** object's **addParticipants** method.
13. Toggle the security of the **fileTransfer** session's transport layer using the **fileTransfer** object's **setSecure** method.
14. Finally, the **wscFileTransfer** object's **start** method is used to start the file transfer session taking the **fileConfig** object as an argument.

***Example 8–24 Instantiating a File Transfer Session***

```
var
    wscFileTransfer,
    fileTransferPackage,
    fileConfigs = [],
    target = document.getElementById("target").value;

fileTransferPackage = new wsc.FileTransferPackage(wscSession);
fileTransferPackage.onFileTransfer = onFileTransfer;

wscFileTransfer = fileTransferPackage.createFileTransfer(target);

wscFileTransfer.onStateChange = onStateChange
wscFileTransfer.onSessionStateChange = onSessionStateChange
wscFileTransfer.onFileData = onFileData;
```

```
wscFileTransfer.onProgress = onFileProgress;
wscFileTransfer.onFileTransferSuccess = onFileTransferSuccess;
wscFileTransfer.onFileTransferFailure = onFileTransferFailure;

fileConfig.file = myFileName;
fileConfig.props = null;

wscFileTransfer.start(fileConfig);
```

### Control and Return Information on the File Transfer

Within the file transfer session, use the following methods as required:

- **abort**: Abort the file transfer session.
- **getInitiator**: Return the initiating address of the file transfer session.
- **getTarget**: Return the file transfer session recipient.

### Terminate the File Transfer Session

Once the file transfer session is ended using the **wscFileTransfer** object **end** method, use the **wscFileTransferPackage** object **close** method to terminate all sessions and release resources.

## Send a File from Your Application

In Example 8–25, the selectedFiles variable is initialized, the recipient's address is retrieved from the text box, **target**, and assigned to the variable, **recipient**, and the **selectFiles** function is bound to the **onchange** event of the **selectFilesButton** input button. When the **onchange** event fires, the file selected in the file browser of the selectFilesButton input button is assigned to **selectedFile**.

Next, when the input button, **fileButton**, is clicked, the **sendFile** function bound to it is triggered. In the **sendFile** function, both the **recipient** and **selectedFile** variables are checked to make sure they are not empty, and, if valid a new **wscFileTransfer** object is created and initialized, and the various event handlers for the **wscFileTransfer** object are bound to their respective listeners as in "Setup a File Transfer Session".

Then, **selectedFile** is retrieved from the input user interface, using the document object's **querySelector** method, and a fileConfig object is created.

Finally, the file transfer is started using the **wscFileTransfer** object's **start** method.

***Example 8–25 Sending One or More Files***

```
var
    selectedFile,
    recipient = document.getElementById("target").value;

document.querySelector("#selectFilesButton").onchange = selectFile;

function selectFile() {
  var
    fileInput = document.querySelector("#selectFileButton");

  selectedFile = fileInput.file;
  }
}

function sendFile() {
  if (recipient != "") {
```

```
    if (selectedFile) {
      wscFileTransfer = fileTransferPackage.createFileTransfer(target);

      wscFileTransfer.onCallStateChange = onStateChange
      wscFileTransfer.onSessionStateChange = onSessionStateChange
      wscFileTransfer.onFileData = onFileData;
      wscFileTransfer.onProgress = onFileProgress;
      wscFileTransfer.onFileTransferSuccess = onFileTransferSuccess;
      wscFileTransfer.onFileTransferFailure = onFileTransferFailure;

      var fileConfigs = [];
      var file = selectedFile[0], fileConfig = {};

      fileConfig.file = file;
      fileConfig.props = null;
      fileConfigs.push(fileConfig);

      wscFileTransfer.start(fileConfigs);

    }
  }
}
```

## Handle Incoming File Transfer Requests

Define the **onFileTransfer** event handler to process incoming file transfer session requests.

When responding to an incoming file transfer session request, use the following methods as required:

- **accept**: Accept the file transfer invitation.
- **decline**: Decline the file transfer invitation.
- **getInitiator**: Return the initiator of the file transfer session request.

In Example 8–15, the **statusArea** div element is used as the target of a **showRequest** function that retrieves the file names for the incoming fileTransfer objects, creates a status message, and an interface that allows a user to accept or decline the file transfer invitation. The status message and interface are rendered in the statusArea div element. If the session is accepted, the event handlers are initialized in the same manner as Example 8–24.

***Example 8–26 onFileTransfer Sample Code***

```
function onFileTransfer(fileTransfer) {
  document.getElementById("statusArea").innerHTML = showFileRequest(fileTransfer);

  document.getElementById("acceptButton").onclick = function() {
    fileTransfer.accept();
  };

  document.getElementById("declineButton").onclick = function() {
    fileTransfer.decline();
  };

  function showFileRequest(fileTransfer) {
    var
      initiator = fileTransfer.getInitiator(),
      fileConfigs = fileTransfer.getFileConfigs();
```

```
      if (fileConfigs.length > 1) {
        message = initiator + " wants to send you some files.";
      } else {
        message = initiator + " wants to send you a file.";
      }

    for (var i=0; i<fileConfigs.length; i++) {
      message += fileConfigs[i].props.name;
    }

    var display = message +
      "<input type='button' name='acceptButton' id='acceptButton' value='Accept' onclick=''/>" +
      "<input type='button' name='declineButton' id='declineButton' value='Decline' onclick=''/>";

    return display;
  }

  wscFileTransfer.onCallStateChange = onStateChange
  wscFileTransfer.onSessionStateChange = onSessionStateChange
  wscFileTransfer.onFileData = onFileData;
  // Continue intializing wscFileTransfer...
}
```

## Handle File Transfer Signaling State Changes

Define the **onStateChange** event handler to process changes in the chat signaling state. The **wsc.CALLSTATE** enum defines the possible call states. For a complete list of **wsc.CALLSTATE** values see the *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

***Example 8–27 onStateChange Sample Code***

```
function onStateChange(ft, callState) {
  console.log("File transfer state: " + callState.state);
  switch (callState.state) {
    case wsc.CALLSTATE.ESTABLISHED:
      // Handle an established call (file transfer) state as required...
      break;
    case wsc.CALLSTATE.UPDATED:
      // Handle an updated call (file transfer) state as required...
      break;
    case wsc.CALLSTATE.RESPONDED:
      // Handle an responded call (file transfer) state as required...
      break;
    case wsc.CALLSTATE.ENDED:
      // Handle an ended call (file transfer) state as required...
      break;
    case wsc.CALLSTATE.FAILED:
      // Handle an failed call (file transfer) state as required...
      break;
    default:
      break;
  }
}
```

## Handle File Transfer Connection State Changes

Define the **onConnectionStateChange** to process changes in file transfer connection state over MSRP. The **ConnectionStateEnum** defines the possible connection states.

***Example 8–28 onConnectionStateChange Sample Code***

```
function onConnectionStateChange(state) {
  console.log("File transfer state: " + state.state);
  switch (state.state) {
    case wsc.ConnectionStateEnum.INIT:
      // Handle the INIT state...
      break;
    case wsc.ConnectionStateEnum.ESTABLISHED:
      // Handle the ESTABLISHED state...
      break;
    case wsc.ConnectionStateEnum.ERROR:
      // Handle the ERROR state...
      break;
    case wsc.ConnectionStateEnum.CLOSED:
      // Handle the CLOSED state...
      break;
  }
}
```

## Handle Message Transmission Success and Failure Events

Define the **onFileTransferSuccess** event handler to process notification of a successful file transmission. In Example 8–29, the only result is a log notification, but you might wish to update a status area or provide a more dynamic notification system.

***Example 8–29 onFileTransferSuccess Sample Code***

```
function onFileTransferSuccess(fileTransferId) {
  console.log("File successfully sent. ID: " + fileTransferId);
}
```

Define the **onFileTransferFailure** event handler to process notification of a failed file transmission. As with Example 8–30, the only result is a log notification, but, again, you may wish to create a customized notification in your own application.

***Example 8–30 onFileTransferFailure Sample Code***

```
function onFileTransferFailure(fileTransferId) {
  console.log("File transfer failed. ID: " + fileTransferId);
}
```

## Handle File Data Transmission

Define the **onFileData** event handler to process the data received from a file transmission.

Data is received in a **fileData** object comprising the following elements:

- **fileTransferID**: A string containing the file transfer ID (defined by WebRTC Session Controller).
- **range**: A **FileDataRange** object containing the total range of data in the file. This object comprises the following properties:
  - **start**: A number indicating the first byte of the data.
  - **end**: A number indicating the final byte of the data.
  - **total**: A number indicating the total size of the data in bytes.
- **content**: An 8-bit unsigned integer array containing the actual file data.

In Example 8–31, the data object is processed and assigned to **cachedFileData** as it comes in, pushed onto the **cachedFileData** array if it is text, or concatenated to the array if it is binary. As the data comes in, the progress bar percentage is calculated using the **range.end** and **range.total** properties of the **data** object. If the progress bar for the currently transferring file has not already been added to the HTML document, the **showFileTransferProgress** function is called.

The **showFileTransferProgress** function finds the current **fileConfig** using its **fileTransferId** argument and comparing that to the **fileTransferIDs** of the fileTransfer object. If a **fileTransferArea** div element does not already exist in the HTML page, one is create and inserted before the **buttonArea** div element. The **insertFileTransferProgress** function adds both a progress bar and a cancel button to the **fileTransferArea** div element, either inserting or appending depending upon the state of the div element.

With the progress bar added, it is updated as the file transfer progresses.

Once the file transfer is complete, the **cachedFileData** array is assigned to a Binary Large Object (BLOB) file, the file name is pulled from the **fileConfigs** array, and the **saveToDisk** function is used to save the BLOB to the user's local disk.

***Example 8–31 onFileData Sample Code***

```
var
    cachedFileData = [];

function onFileData(data) {
  var fileTransferId = data.fileTransferId;
  console.log("Received file data. ID: " +
                              fileTransferId + ", range: " + JSON.stringify(data.range));

  if (!cachedFileData[fileTransferId]) {
    cachedFileData[fileTransferId] = [];
  }

  if (data.content instanceof String) {
    cachedFileData[fileTransferId].push(data.content);
  } else {
    cachedFileData[fileTransferId] = cachedFileData[fileTransferId].concat(data.content);
  }

  // Update the progress bar...
  var progressPercent = Math.ceil(data.range.end/data.range.total*100);
  if (!document.getElementById(fileTransferId)) {
    showFileTransferProgress(fileTransferId);
  }

  var progressBar = document.getElementById(fileTransferId).childNodes[1];

  if (progressBar) {
    progressBar.style.width = progressPercent+"%";
  }

  // File transfer finished...
  if (data.range.end == data.range.total) {
    var blob = new Blob(cachedFileData[fileTransferId]),
    fileConfigs = wscFileTransfer.getFileConfigs(),
    fileName = null;

    for (var i=0; i<fileConfigs.length; i++) {
```

```
      if (fileConfigs[i].props.fileTransferId == fileTransferId) {
        fileName = fileConfigs[i].props.name;
        break;
      }
    }

    blob.url = (window.URL || window.webkitURL).createObjectURL(blob);
    if (fileName) {
      saveToDisk(blob.url, fileName);
    }
    var fileTransferProgressElem = document.getElementById(fileTransferId);
    if (fileTransferProgressElem) {
      fileTransferProgressElem.hidden = true;
      var cancelBtn = fileTransferProgressElem.nextSibling;
      if (cancelBtn.type == "button") {
        cancelBtn.hidden = true;
      }
    }
  }
}

function showFileTransferProgress(fileTransferId) {
  var
    fileTransferDisplayArea = document.getElementById("fileTransferArea"),
    fileConfigs = wscFileTransfer.getFileConfigs(),
    fileConfig = null;

  for (var i=0; i<fileConfigs.length; i++) {
    if (fileConfigs[i].props.fileTransferId == fileTransferId) {
      fileConfig = fileConfigs[i];
      break;
    }
  }

  if (fileTransferDisplayArea == null) {
    var fileTransferArea = document.createElement("div");
    fileTransferArea.id = 'fileTransferArea';
    var buttonArea = document.getElementById("buttonArea");
    buttonArea.parentNode.insertBefore(fileTransferArea, buttonArea.nextSibling);
    insertFileTransferProgress(fileTransferArea, fileConfig);
  } else {
    insertFileTransferProgress(fileTransferDisplayArea, fileConfig);
  }

  function insertFileTransferProgress(fileTransferDisplayArea, fileConfig) {
    var
      fileNameNode,
      progressBar,
      lastProgressChild,
      cancelBtn,

    fileTransferProgressDisplay = document.createElement("div");
    fileTransferProgressDisplay.id = fileTransferId;
    fileTransferProgressDisplay.className = "progress";
    fileTransferProgressDisplay.style.width = "85%";
    fileTransferProgressDisplay.style.float = "left";

    fileNameNode = document.createElement("div");
    fileNameNode.innerText = fileConfig.props.name;
    fileNameNode.style.position = "absolute";
```

```
    fileNameNode.style.textAlign = "center";
    fileNameNode.style.left = "3%";
    fileNameNode.style.color = "white";
    fileTransferProgressDisplay.appendChild(fileNameNode);

    // Append progress bar
    progressBar = document.createElement("div");
    progressBar.className = "bar";
    progressBar.style.width = "0%";
    fileTransferProgressDisplay.appendChild(progressBar);

    // Append the cancel button
    cancelBtn = document.createElement("button");
    cancelBtn.id = "cancelBtn";
    cancelBtn.type = "button";
    cancelBtn.innerText = "Cancel";
    cancelBtn.style.marginLeft = "2%";
    cancelBtn.style.marginBottom = "20px";
    cancelBtn.onclick = function() {
      wscFileTransfer.abort(fileConfig);
      cancelBtn.disabled = 'true';
    };

    if (fileTransferDisplayArea.hasChildNodes()) {
      lastProgressChild = fileTransferDisplayArea.lastChild;
      fileTransferDisplayArea.insertBefore(fileTransferProgressDisplay,
                                                    lastProgressChild.nextSibling);
      fileTransferDisplayArea.appendChild(cancelBtn);
    } else {
      fileTransferDisplayArea.appendChild(fileTransferProgressDisplay);
      fileTransferDisplayArea.appendChild(cancelBtn);
    }
  }
}
```

## Handle File Transfer Progress Updates

Define the **onProgress** event handler for handling file transmission progress data. The **onProgress** event handler returns a **fileProgressData** object which is the same as the **fileData** object but lacks the actual file content data:

- **fileTransferID**: A string containing the file transfer ID (defined by WebRTC Session Controller).
- **range**: A **FileDataRange** object containing the total range of data in the file. This object comprises the following properties:
  - **start**: A number indicating the first byte of the data.
  - **end**: A number indicating the final byte of the data.
  - **total**: A number indicating the total size of the data in bytes.

In Example 8–32, when a progress event occurs, the fileTransferId data object attribute is passed to the **showFileTransferProgress** function described in "Handle File Data Transmission", and the progress bar updated in the same manner.

***Example 8–32 onProgress Sample Code***

```
function onProgress(data) {
  var
    fileTransferId = data.fileTransferId;
```

```
  if (!document.getElementById(fileTransferId)) {
    showFileTransferProgress(fileTransferId);
  }

  var
    progressPercent = Math.ceil(data.range.end/data.range.total*100),
    progressBar = document.getElementById(fileTransferId).childNodes[1];

  if (progressBar) {
    progressBar.style.width = progressPercent+"%";
  }
}

function showFileTransferProgress(fileTransferId) {
  // See Example 8-31...
}
```

# 9

# Using the WebRTC Browser Extension

This chapter describes how to use the WebRTC browser extension with WebRTC Session Controller.

## About WebRTC Session Controller Browser Plug-in Support

For many enterprise customers for whom the Internet Explorer (IE) and Safari browsers remain integral to their business operations, a single complete solution to supporting WebRTC technology is critical.

The Temasys plug-in, version **0.8.884**, certified by WebRTC Session Controller, supports WebRTC in desktop versions of IE, version 11, and Safari, versions 9 and 10. You must also use the **adapter.js** helper file for the plug-in to develop WebRTC applications for these browsers. Load the **adapter.js** file after all the WebRTC Session Controller web SDK .js files are loaded. See "WebRTC Session Controller Support Libraries" for more information.

With the Temasys WebRTC plug-in, your WebRTC applications can support audio and video calls, data channel, and HTTP proxy.

### About the WebRTC Plug-in

See the links that accompany the following topics for more information regarding the Temasys plugin:

- Public documentation, including information on supported audio and video codecs:

  http://confluence.temasys.com.sg/display/TWPP

  **Note:** Before you proceed, review the documentation links provided at the bottom section of this website and verify that the plug-in meets your requirements

- Downloading and installing:

  http://confluence.temasys.com.sg/display/TWPP/Downloading+and+Installing

- Downloading the version of AdapterJS provided with the plug-in installer:

  https://github.com/Temasys/AdapterJS

- Determining browser support and all other details about using the Temasys plug-in:

http://confluence.temasys.com.sg/display/TWPP/How+to+integrate+the+Temasys+WebRTC+Plugin+into+your+website

This website provides helpful guidelines to assist you in integrating the Temasys WebRTC plug-in with your WebRTC-based web application.

For general information about monitoring browsers and connections, see "Debugging and Troubleshooting Your WebRTC-Enabled Applications".

## Integrating WebRTC Session Controller with Temasys WebRTC Plug-in

To enable WebRTC Session Controller SDK to work with the Temasys plug-in, load the Temasys **adapter.js** file after all the WSC SDK **.js** files are loaded. In Example 9–1, the *js/adapter.js* file, obtained from the Temasys website, is the last entry.:

***Example 9–1 Loading the Temasys adapter.js file***

```
<head>
...
<script type="text/javascript" src="/api/wsc-common.js"></script>
  <script type="text/javascript" src="/api/wsc-call.js"></script>
  <script type="text/javascript" src="js/adapter.js"></script>
...
<head>
```

### Application Logic for Media Streams

You can now configure audio and video support for your Android and IOS -based applications.

**Android Applications** For information about configuring the audio and video support for your Android applications, see:

- Setting Up Audio Calls in Your Applications
- Setting Up Video Calls in Your Applications
- Adding WebRTC Voice Support to your Android Application
- Adding WebRTC Video Support to your Android Application

**iOS Applications** For information about configuring the audio and video support for your iOS applications, see:

- Setting Up Audio Calls in Your Applications
- Setting Up Video Calls in Your Applications
- Adding WebRTC Voice Support to your iOS Application
- Adding WebRTC Video Support to your iOS Application

# 10

# 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 about the individual WebRTC Session Controller JavaScript API classes.

## About the Default Messaging Mechanism Used by Your Applications

When your application creates a session, a call or message alert package, starts 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 the signaling messages for all the resulting default commands.

In order to broaden or extend your application logic, 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 more communication-related services in audio, video, and data transfer flows. For example:

- Custom calls

  You can implement logic to prepare custom calls. Set 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 can 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 that extends 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 employ 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 10–1 creates the *wsc.CallExtension* object which extends the **wsc.Call** object.

***Example 10–1 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);
```

Here, 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. To access and retrieve information about a subsession, use the methods in **wsc.ExtensibleSession**. To do so, use its identifier (**sessionId)**, retrieve a specific package by its type and manage the object. You can also 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 tasks:

- 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 **CallPackage.onMessage** event handler in the application. When call-related messages come in to your application, this callback function is called, 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, the **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 insert the call in the flow 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. To do so, extend the **Call** object. 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 call the methods that support extension headers. See "Handling Extra 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 the **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 the **MessageAlertPackage** object in your application manages an array of subscriptions, then, when a notification comes to your application, the **MessageAlertPackage.onMessage** event handler is called. In the callback function you assign to this event handler, you can process the incoming message notification to identify the subscription object and call 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 Extra Headers in Messages

You can set up your application to send or receive extra data in the form of an extra header field.

## About Extra Headers in Messages

Some methods in the **Call** and **CallPackage** class objects can accept an extra argument, as long as it is a JSON object. This extra 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, your application page is designed for an auto dealership. Your application has gathered data on the preferences that a customer has for the make, model, and deal preferences, such as *carMaker*, *Convertible*), and *Lease*.

When the customer 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 called 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 displays the information for the dealer.

## Handling Extra 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 *Oracle Communications WebRTC Session Controller Extension Developer's Guide*.

### Managing Calls with Extra 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. To start the call, place the JSON object as the last parameter when your application calls the **start** method of the outgoing call object.

   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 the **CallPackage.onIncomingCall** event handler. The extension header is in JSON format.
2. Perform any actions based on the extra data in the extension header.
3. If the call is accepted, do the following:
   a. Set up the extension header as a JSON object.
   b. To start the call, include the JSON object when your application calls the **start** method of the outgoing call object.

For example, if *call* is the call object, *callConfig* the local media stream and the 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 a callee declines the call, do the following:
   a. Obtain the reason why the callee declined the call.
   b. Set up the extension header as a JSON object.
   c. Include the JSON object when your application calls the **decline** method of the outgoing call object.

      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)
      ```

Set up your application to handle incoming messages and set up outgoing messages associated with the following events. In each case, set up the extension header and place it as the last parameter when you call the associated function for the extended call.

- The caller or callee ends the call.

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

- The call is updated.

  If the caller or callee:

  – Requests the update.

    Set up the extension header as described earlier and include it when you call 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 call 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 extra 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 customers to subscribe to the presence of, chat with, or set up calls that can be transferred to other individuals. 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 WebSocket 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 in your application, use the following objects:

- The **ExtensibleSession**:

  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 the **ExtensibleSession** object.

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

  For more information about subsessions, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

- 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 the **ExtensibleSession** object.

  – Retrieving the package by its type by using the **getPackage** method of the **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 the **ExtensibleSession** object.

  – Handling an incoming message using the **Call.onMessage** method in the **ExtensibleSession** object.

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

    The generated correlation ID is based on the current outbound **sequence** number sequence of this session. For information about **sequence**, see *Oracle Communications WebRTC Session Controller Extension Developer's Guide*.

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

To handle the data associated with custom sessions or subsessions, save the corresponding data in the HTML **SessionStorage** area using the **ExtensibleSession.saveToStorage** method. Store the session data in JSON format. To maintain the session's current state for use in dealing with connectivity issues, ensure that your application captures the necessary changes.

---

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

---

## Sending And Receiving Custom Messages

At times you create messages independent of the default call or message alert package. Set up logic to handle the flow of such custom messages and the resulting actions in your application.

You can send custom messages within a subsession by providing a **Message** object as an argument when you call the **ExtensibleSession#sendMessage** method. Ensure that the **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 *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

### About the API Classes Used to Create Custom Message

To create custom packages and use custom message flows in your applications, use the following objects:

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

- wsc.Message#payload

> **Note:** For information about 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 10–2 shows the header sections of a message object which starts a WebSocket connection:

***Example 10–2 The Header Sections of a Message Object***

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

> **Note:** To create messages independent of the default call or message alert package, use the **wsc.Message** object. 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 about the headers supported in the Control section, see *Oracle Communications WebRTC Session Controller JavaScript API Reference* and *Oracle Communications 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 launched the request, for whom it is intended, and so on. Your application can add extra headers to this section. A gateway server can map such headers to a SIP header or a parameter. For information about the headers supported in the Header section, see *Oracle Communications WebRTC Session Controller JavaScript API Reference* and *Oracle Communications 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 contains the presence information.

## Managing Custom Message Data Flows

To send and receive messages for custom flows 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 the **ExtensibleSession** object in your application.
- To take further action, monitor the message flow.
- 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

You can customize your application by extending the default call and message alert package API objects in WebRTC Session Controller JavaScript API library.

## 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 10–3 Creating an Extended Call Package***

```
var CallPackageExtension = function() {

    //sub-class must call the constructor of the superclass
    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 an extended CallPackage.

extcallPackage = new CallPackageExtension(wscSession);
```

Call objects created from this extended call package can handle extra 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 calls 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 10–4 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 10–4 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 he objects.
//The objects initialized in the constructor code for the parent are 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 it 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.
//A CallExtension object must be 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 calls 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.
- To handle the overridden **onMessage** event handler of the extended **MessageAlertPackage** object, assign a callback function. 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 stores 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 about the new notification messages.

# 11

# Debugging and Troubleshooting Your WebRTC-Enabled Applications

This chapter describes how to set up runtime environment checks and debug the applications you develop using the Oracle Communications WebRTC Session Controller Javascript application programming interface (API) library.

## About the Runtime Checks on the Environment

How well your applications perform in the runtime depend partly on browser compatibility, media device availability and network connectivity. This section deals with how you can check these elements in your runtime environment.

### Browsers and Connections

Your WebRTC-enabled applications are supported on browsers such as Google Chrome, Microsoft Firefox, Opera, and Edge. Within your application, you need to verify and monitor the following:

- Browsers. See "Testing Browser Compatibility."
- Network connections. See "Monitoring Network Connections."
- **WebSocket** connection. See "Monitoring the WebSocket."
- **RTCPeerConnection** interface. See "Verifying the WebRTC Connection to a Peer."
- Media devices, including microphones and camera. For:
  - Microphones, see "Selecting Appropriate Source for the Audio Elements."
  - Camera, see "Selecting Appropriate Source for the Video Elements."

#### Testing Browser Compatibility

To verify the browser compatibility, call the **testBrowser** method of the **wsc** namespace. Its syntax is:

```
<static> testBrowser()
```

The **testBrowser** method returns an object whose format is:

```
{
     compatible: boolean,
     details: {
         browserName: String,
         browserVersion: String,
         supportedMinimumVersion: String,
```

```
            supportedMaximumVersion: String,
            supportWebRTC: boolean,
            supportWebSocket: boolean,
            supportSessionStorage: boolean
        }
}
```

For more information about **wsc.testBrowser**, see *Oracle Communications WebRTC Session Controller Cloud Edition JavaScript API Reference*.

### Monitoring Network Connections

To monitor your network connections call the **testNetwork** method of the **wsc** namespace. Its syntax is:

```
<static> testNetwork(turnURI, username, password)
```

Where all three input parameters are required:

- *turnURI* contains the URI of the TURN server. For example, `turn:myserver.com:3478`.
- *username* contains the user name for the TURN server.
- *password* contains the password for the TURN server.

The **wsc.testNetwork** method returns a **Promise** whose value is an object in the format shown in:

***Example 11–1 Object format of Promise Returned by wsc.testNetwork***

```
{
  connectivity: boolean,    // true if (udpConnectivity || tcpConnectivity) && (dataThroughput > 0
&& videoBandwidth > 0)
  details: {
    udpConnectivity: boolean,
    tcpConnectivity: boolean,
    dataThroughput: Number // unit: kbps
    videoBandwidth: Number // unit: kbps
  }
}
```

For more information on **wsc.testNetwork**, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

### Monitoring the WebSocket

WebRTC Session Controller Javascript SDK uses WebSocket connections to send and receive real-time communication (RTC) and JSON messages to and from the Signaling Engine server. The rapid but disparate rates in the evolution of the supported browsers makes the configuration and monitoring of WebSockets an important task.

For more information on monitoring WebSocket connections, see the description about "Using the WebSocket Protocol in WebLogic Server" in *Oracle Fusion Middleware Developing Applications for Oracle WebLogic Server*.

### Verifying the WebRTC Connection to a Peer

**RTCPeerConnection** is an interface that enables you to manage a WebRTC connection between a local computer and a remote peer. Information on how to use the **RTCPeerConnection** is available at

http://www.w3.org/TR/webrtc/

This guide contains an example of how to configure local video and audio MediaStream objects using the WebRTC **PeerConnectionFactory**, for an Android application. See "Configure the Local MediaStream for Audio".

# Media Devices

Media devices include microphone and camera. There may be multiple microphones and cameras on one terminal. To check the given device, WebRTC Session Controller uses the device source ID. If no device source ID is provided, the default device is used.

## Selecting Appropriate Source for the Audio Elements

When your applications support the use of audio devices, it is important to know the state of the audio devices at a terminal. At times, there may be multiple microphones at a terminal. To check a given device, Web RTC Session Controller uses the source ID of the device. If no device source ID is provided, it uses the default device.

To verify and test microphone availability in the runtime environment, use the following:

- **getuserMedia**

  The **getUserMedia** method uses constraints to help select an appropriate source for a track and configure it. When you use the **getUserMedia** method, a track is not connected to a source if its initial constraints cannot be satisfied.

  For more information on the **getUserMedia** method, see

  http://w3c.github.io/mediacapture-main/getusermedia.html

  To troubleshoot a microphone, you can use the WebRTC Troubleshooter at

  https://test.webrtc.org/

- **wsc.testMicrophone**

  To test the microphone availability, call the **testMicrophone** method of the **wsc** namespace. Its syntax is:

  ```
  <static> testMicrophone()
  ```

  The **wsc.testMicrophone** method returns a **Promise** whose value is an object in the format shown in:

***Example 11–2 Object format of Promise Returned by wsc.testNetwork***

```
{
  available: boolean,    // true if one source is available
  details: [
    {
      available: boolean,
      sourceId: String,
      label: String,
      channelNumber: Number,
      isStereo: boolean
    },
      ...
    {
      available: boolean,
```

```
      sourceId: String,
      label: String,
      channelNumber: Number,
      isStereo: boolean
    }
  ]
}
```

For more information on **wsc.testNetwork**, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

- **Web Audio API**

  The Web Audio API has been designed so that applications can support the changing face of modern desktop audio software.

  For more information on the **Web Audio API**, see

  http://www.w3.org/TR/webaudio/

## Selecting Appropriate Source for the Video Elements

When your applications support the use of media devices, it is important to know the state of the media device at a terminal. At times, there may be multiple microphones and cameras at one terminal. To check a given device, Web RTC Session Controller uses the source ID of the device. If no device source ID is provided, it uses the default device.

To verify, and test camera availability in the runtime environment, use the following:

- **getuserMedia**

  The **getUserMedia** method uses constraints to help select an appropriate source for a track and configure it. When you use the **getUserMedia** method, a track is not connected to a source if its initial constraints cannot be satisfied.

  For more information on the **getUserMedia** method, see

  http://w3c.github.io/mediacapture-main/getusermedia.html

  To troubleshoot a camera, you can use the WebRTC Troubleshooter at

  https://test.webrtc.org/

- **wsc.testCamera**

  To test the microphone availability, call the **testCamera** method of the **wsc** namespace. Its syntax is:

  ```
  <static> testCamera()
  ```

  The **wsc.testCamera** method returns a **Promise** whose value is an object in the format shown in:

***Example 11–3 Object format of Promise Returned by wsc.testCamera***

```
{
  available: boolean,    // true if one source is available
  details: [
    {
      available: boolean,
      sourceId: String,
      label: String,
      availableResolutions: [String]
```

```
    },
      ...
    {
      available: boolean,
      sourceId: String,
      label: String,
      availableResolutions: [String]
    }
  ]
}
```

For more information on **wsc.testCamera**, see *Oracle Communications WebRTC Session Controller JavaScript API Reference*.

# About Debugging Your Applications

You can enable the debug mode for an application at the server level. When you do so, all the sessions of that application are in the debug mode and logs are collected for those sessions.

To run the application in the debug mode, select the **Debug** check box for your application entry in the WebRTC Session Controller Administration Console.

To do so, access the **Application Profiles** tab, select the application name. In the **Profile** tab for the application, select the **Debug** check box. For more information about configurable runtime parameters, see the table under "Global Integration Parameters of the Signaling Engine" in *WebRTC Session Controller System Administrator's Guide*.

## About the Log Information

When you enable this API in your application, the following logs are recorded for the specific application:

- All JsonRTC messages from the client and to the client.
- All SIP message from the client and to the client.
- All session or sub-session status.
- All the error and exception messages encountered.

### About the Log Recording Points During the Life Cycle of a Session

When the debug mode is enabled for an application, WebRTC Session Controller provides information over the life cycle of the session and its sub-sessions. WebRTC Session Controller logs errors and exceptions at the following events:

- For a Session:
  - onSessionCreate
  - onSessionDestroy
- For a Sub Session
  - onSubSessionStarted
  - onSubSessionEstablished
  - onSubSessionFailed
- For SIP

- onSipRequestReceived
- onSipResponseReceived
- onSipRequestSent
- onSipReesponseReceived
- onSipSessionEnd

### Sample of a Log Output

Example 11–4 shows a sample debug log file:

***Example 11–4 Sample Debug Log File***

```
DEBUG 2015-07-03 17:31:56,001 [ (self-tuning)'] (sc.core.debug.LogRecorder:  43) -
 ------------------------------------------------
## Application : guest
## SessionId : pQAAAU5TN6Ml6hLzQqzR71zJqnwAAAAD_4
## MessageState : JSON_RECEIVED
## LogContent : Registration message : Frame(control=Control(version=1.0, verb=connect,
type=REQUEST, messageState=null, sequence=1, sessionId=null, subSessionId=null, correlationId=null,
ackSequence=0, packagetype=register, wscId=null, connectSlr=null, connectSlw=null, connectSuw=null,
connectSslr=null), normalized={control={}, header={}, payload={}})

DEBUG 2015-07-03 17:31:56,005 [ (self-tuning)'] (sc.core.debug.LogRecorder:  43) -
 ------------------------------------------------
## Application : guest
## SessionId : pQAAAU5TN6Ml6hLzQqzR71zJqnwAAAAD_4
## MessageState : SUBSESSION_CREATED
## LogContent :
```

## About the Configuration of Application Debug Log File

When an application runs in the debug mode, WebRTC Session Controller stores the output to a file named, **wsc-debug-client.log**. Look for this file in the **domain/server/logs** directory.

Example 11–5 shows the setup of the debug log file:

***Example 11–5 Application Debug Log File Configuration***

```
<Appenders>
    <RollingFile
            name="debug"
            fileName="servers/${sys:weblogic.Name}/logs/wsc-debug-server.log"
            filePattern="servers/${sys:weblogic.Name}/logs/wsc-debug-server.log-%d{yyyy-MM-dd}-%i"
            immediateFlush="true"
            append="true">

      <PatternLayout pattern="%5p %d [%-15.15t] (%-25.25c:%4L) - %m%n" />
      <Policies>
        <OnStartupTriggeringPolicy />
        <TimeBasedTriggeringPolicy interval="24" modulate="true" />
        <SizeBasedTriggeringPolicy size="100 MB"/>
      </Policies>
      <DefaultRolloverStrategy max="10"/>
    </RollingFile>
  </Appenders>

  <Loggers>
```

```
  <Logger name="oracle.wsc.core.threat.log" level="info" additivity="false">
    <AppenderRef ref="threat"/>
  </Logger>
  <Logger name="oracle.wsc.core.debug" level="debug" additivity="false">
    <AppenderRef ref="debug"/>
  </Logger>
  <Root level="warn">
    <AppenderRef ref="file" />
  </Root>
</Loggers>
```

# 12

# Developing WebRTC-Enabled Android Applications

This chapter shows how you can develop WebRTC-enabled Android applications with the Oracle Communications WebRTC Session Controller Android application programming interface (API) library.

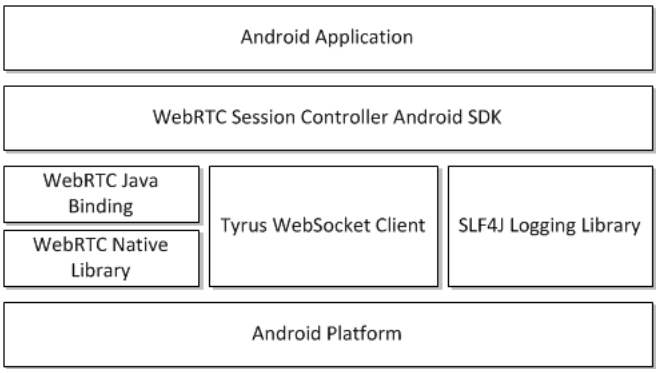
## About the Android SDK

The WebRTC Session Controller Android SDK enables you to integrate your Android applications with core WebRTC Session Controller functions. You can use the Android SDK to implement the following features:

- Audio calls between an Android application and any other WebRTC-enabled application, a Session Initialization Protocol (SIP) endpoint, or a public switched telephone network endpoint using a SIP trunk.
- Video calls between an Android application and any other WebRTC-enabled application, with suitable support for video conferencing.
- Seamless upgrading of an audio call to a video call and downgrading of a video call to an audio call.
- Support for Interactive Connectivity Establishment (ICE) server configuration, including support for Trickle ICE.
- Transparent session reconnection following network connectivity interruption.

The WebRTC Session Controller Android SDK is built upon several other libraries and modules as shown in Figure 12–1.

***Figure 12–1 Android SDK Architecture***



The WebRTC Java binding enables Java access to the native WebRTC library which itself provides WebRTC support. The Tyrus WebSocket client enables the WebSocket access required to communicate with WebRTC Session Controller. Finally, the SLF4J logging library enables you to plug in a logging framework of your choice to create persistent log files for application monitoring and troubleshooting.

For more information about any of the APIs described in this document, see *Oracle Communications WebRTC Session Controller Android API Reference*.

## About the Android SDK WebRTC Call Workflow

The general workflow for using the WebRTC Session Controller Android SDK is:

1. Authenticate against WebRTC Session Controller using the **HttpContext** class. You initialize the HttpContext with necessary HTTP headers and optional SSLContext information in the following manner:
   a. Send an HTTP GET request to the login URI of WebRTC Session Controller.
   b. Complete the authentication process based on your authentication scheme.
   c. Proceed with the WebSocket handshake on the established authentication context.
2. Establish a WebRTC Session Controller session using the **WSCSession** class. Two more classes must be implemented:
   - **ConnectionCallback**: An interface that reports on the success or failure of the session creation.
   - **WSCSession.Observer**: An abstract class that signals on various session state changes, including **CLOSED**, **CONNECTED**, **FAILED**, and others.
3. Once a session is established, create a **CallPackage** which manages **Call** objects in a **WSCSession**.
4. Create a **Call** using the **CallPackage createCall** method with a callee ID as its argument, for example, alice@example.com.
5. To monitor call events such as **ACCEPTED**, **REJECTED**, **RECEIVED**, create a Call.Observer class which attaches to the **Call**.
6. To determine the nature of the WebRTC call, whether bi or mono-directional audio or video or both, create a **CallConfig** object.
7. Create and configure a new **PeerConnectionFactory** object and start the **Call** using the **start** method of the call.
8. When the call is complete, terminate the **Call** object using its **end** method.

## Prerequisites

Before continuing, make sure you thoroughly review and understand the JavaScript API discussed in the following chapters:

- About Using the WebRTC Session Controller JavaScript API
- Setting Up Security
- Setting Up Audio Calls in Your Applications
- Setting Up Video Calls in Your Applications

The WebRTC Session Controller Android SDK is closely aligned in concept and functionality with the JavaScript SDK to ensure a seamless transition.

In addition to an understanding of the WebRTC Session Controller JavaScript API, you are expected to be familiar with:

- Java and object oriented programming concepts
- General Android SDK programming concepts including event handling, and activities.

There are many excellent online resources for learning Java programming, and, for a practical introduction to Android programming, see http://developer.android.com/guide/index.html.

## Android SDK System Requirements

In order to develop applications with the WebRTC Session Controller SDK, you must meeting the following software/hardware requirements:

- Java Development Kit (JDK) 1.6 or higher installed with all available security patches: http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase6-419409.html

  **Note:** OpenJDK is not supported.

- The latest version of the Android SDK available from http://developer.android.com/sdk/installing/index.html, running on a supported version of Windows, Mac OS X, or Linux.
- If you are using the Android SDK command line tools, you must have Apache Ant 1.8 or later: http://ant.apache.org/.
- A installed and fully configured WebRTC Session Controller installation. See the WebRTC Session Controller Installation Guide.
- An actual Android hardware device. You can test the general flow and function of your Android WebRTC Session Controller application using the Android emulator. However, a physical Android device such as a phone or tablet is required to utilize audio or video functionality.

## About the Examples in This Chapter

The examples and descriptions in this chapter are kept intentionally straightforward. They illustrate the functionality of the WebRTC Session Controller Android SDK API without obscuring it with user interface code and other abstractions and indirections. It is likely that use cases for production applications will take many forms. Therefore, the examples assume no pre-existing interface schemes except when necessary, and then, only with the barest minimum of code. For example, if a particular method requires arguments such as a user name, a code example will show a plain string *username* such as "*alice@example.com*" being passed to the method. It is assumed that in a production application, you would interface with the contact manager of the Android device.

## General Android SDK Best Practices

When designing and implementing your WebRTC-enabled Android application, keep the following best practices in mind:

- Following Android application development general guidelines, do not call any networking operations in the main Application UI thread. Instead, run network operations on a separate background thread, using the supplied Observer mechanisms to handle any necessary responses.
- The Observers themselves run on a separate background thread. Your application must not make any user interface updates on that thread, since the Android user interface toolkit is not thread safe. For more information, see https://developer.android.com/training/multiple-threads/communicate-ui.html.
- In any class that extends or uses the android.app.Application class or any initial Activity class, initialize the WebRTC PeerConnectionFactory only once during its lifetime:

```
PeerConnectionFactory.initializeAndroidGlobals(context, true /* initializeAudio
*/, true /* initializeVideo */);
```

- The signaling communications take place over a background thread. To prevent communications disruption, initialize and create WebRTC Session Controller sessions using an Android background service.

  The background service can maintain a reference to the Session object and share that among the activities, fragments, and other components of your Android application. The service can also be run at a higher priority and be used to handle notifications. For more information, see https://developer.android.com/training/best-background.html.

## Installing the Android SDK

To install the WebRTC Session Controller Android SDK, do the following:

1. After you have installed your Android development environment, use the Android SDK Manager to download the required SDK tools and platform support: http://developer.android.com/sdk/installing/adding-packages.html.

   **Note:** Android API level 17 (4.2.2 *Jellybean*) is the minimum required by the WebRTC Session Controller Android SDK for full functionality. To ensure the broadest application compatibility, target the lowest API level possible.

2. Configure virtual and hardware devices as required for your application: http://developer.android.com/tools/devices/index.html and http://developer.android.com/tools/device.html.
3. Create a Android project using the Android development environment of your choice: http://developer.android.com/tools/projects/index.html.
4. Download and extract the libs folder from the WebRTC Session Controller Android SDK ZIP file into the libs folder of your Android application. Create the libs folder if it does not exist.

   **Note:** Both debug and release versions of the WebRTC peer connection library are included. Choose the correct one for the development state of your project.

5. Depending on your Android development environment, add the path to the libs folder to your Android project as indicated in your Android development environment documentation.

### WebRTC Session Controller SDK Required Permissions

The WebRTC Session Controller SDK requires the following Android permissions to function correctly:

- android.permission.**INTERNET**
- android.permission.**ACCESS_NETWORK_STATE**
- android.permission.**CAMERA**
- android.permission.**RECORD_AUDIO**

Also, if your logging subsystem requires access to an external SD card (or a different storage volume) also grant the android.permission.**WRITE_EXTERNAL_STORAGE** permission.

## Configuring Logging

The WebRTC Session Controller Android SDK includes support for the Simple Logging Facade for Java (SLF4J) which lets you plug in your preferred logging framework.

Examples in this chapter use the popular Log4j logging framework which requires the addition of the following libraries to your project, where *n* indicates a version number:

- slf4j-log4j*n*-*n.n.n*.jar
- log4j-*n.n.n*.jar
- android-logging-log4j-*n.n.n*.jar

***Example 12–1 Configuring Log4j***

```
public class ConfigureLog4j {
  public void configureLogging() {
    Log.i(MyApp.TAG, "Configuring the Log4j logging framework...");
    final LogConfigurator logConfigurator = new LogConfigurator();
    logConfigurator.setFileName(Environment.getExternalStorageDirectory()
              + File.separator
              + "sample_android_app.log");
    logConfigurator.setRootLevel(Level.DEBUG);
    logConfigurator.setFilePattern("%d %-5p [%c{2}]-[%L] %m%n");
    logConfigurator.setMaxFileSize(1024 * 1024 * 5);
    logConfigurator.setImmediateFlush(true);
    logConfigurator.configure();
  }
}
```

**Note:** To write log files to any location other than the internal storage of an Android device, grant the **WRITE_EXTERNAL_STORAGE** permission.

For more information about configuring and using Log4j, see http://logging.apache.org/log4j/.

# Authenticating with WebRTC Session Controller

Use the class **HttpContext** to set up an authentication context. The authentication context contains the necessary HTTP headers and **SSLContext** information, and is used when setting up a **wsc.Session**.

## Initialize the CookieManager

To handle storage of authentication headers and URIs, initialize the cookie manager. For more information about the Android CookieManager class, see http://developer.android.com/reference/android/webkit/CookieManager.html.

***Example 12–2 Initializing the CookieManager***

```
Log.i(MyApp.TAG, "Initialize the cookie manager...");
CookieManager cookieManager = new CookieManager(null, CookiePolicy.ACCEPT_ALL);
java.net.CookieHandler.setDefault(cookieManager);
```

## Initialize a URL Connection

Create a URL object using the URI to your WebRTC Session Controller endpoint. Open a urlConnection using the URL object openConnection method.

***Example 12–3 Initializing a URL Connection***

```
try {
  url = new URL("http://server:port/login?wsc_app_uri=/ws/webrtc/myapp");
} catch (MalformedURLException e1) {
  Log.i(MyApp.TAG, "Malformed URL.");
}
try {
  urlConnection = (HttpURLConnection) url.openConnection();
} catch (IOException e) {
  Log.i(MyApp.TAG, "IO Exception.");
}
```

**Note:** The default WebRTC Session Controller port is 7001.

## Configure Authorization Headers if Required

Configure authorization headers as required by your authentication scheme. The following example uses Basic authentication; OAuth and other authentication schemes are similarly configured. For more information about WebRTC Session Controller authentication, see "Setting Up Security".

***Example 12–4 Initializing Basic Authentication Headers***

```
String name = "username";
String password = "password";
        String authString = "Basic " + name + ":" + password;
byte[] authEncBytes = Base64.encode(authString.getBytes(), 0);
String authHeader = new String(authEncBytes);
urlConnection.setRequestProperty(HttpContext.AUTHORIZATION_HEADER, authHeader);
```

**Note:** If you are using Guest authentication, no headers are required.

## Configure the SSL Context if Required

If you are using Secure Sockets Layer (SSL), configure the SSL context, including the TrustManager if necessary. See Example 12–5.

***Example 12–5 Configuring the SSL Context***

```
if (HTTPS.equals(url.getProtocol())) {
  Log.i(MyApp.TAG, "Configuring SSL context...");
  HttpsURLConnection.setDefaultHostnameVerifier(getNullHostVerifier());
  SSLContext ctx = null;
  try {
    ctx = SSLContext.getInstance("TLS");
  } catch (NoSuchAlgorithmException e) {
    Log.i(MyApp.TAG, "No Such Algorithm.");
  }
  try {
    ctx.init(null, getTrustAllManager(), new SecureRandom());
  } catch (KeyManagementException e) {
    Log.i(MyApp.TAG, "Key Management Exception.");
  }
  final SSLSocketFactory sslFactory = ctx.getSocketFactory();
  HttpsURLConnection.setDefaultSSLSocketFactory(sslFactory);
}
```

Example 12–6 is a stub method. In it, you can implement a routine to test the validity of the input URL object and handle program flow based on HTTP return codes. This method expects a URL object (shown in Example 12–5). It passes that object to a custom **getNullHostVerifier** method, whose job is to validate that the URL is live.

***Example 12–6 Host Name Verification***

```
private HostnameVerifier getNullHostVerifier() {
  return new HostnameVerifier() {
    @Override
    public boolean verify(final String hostname, final SSLSession session) {
      Log.i(MyApp.TAG, "Stub verification for " + hostname +
                              " for session: " + session);
      return true;
    }
  };
}
```

Finally, if your implementation depends upon a Java Secure Socket Extension implementation, configure the Android TrustManager class as required (and shown in Example 12–7). For more information about the Android TrustManager class, see http://developer.android.com/reference/android/webkit/CookieManager.html.

***Example 12–7 Configuring the TrustManager***

```
public static TrustManager[] getTrustAllManager() {
  return new X509TrustManager[] { new X509TrustManager() {
    @Override
    public java.security.cert.X509Certificate[] getAcceptedIssuers() {
      return null;
    }

    @Override
    public void checkClientTrusted(
      java.security.cert.X509Certificate[] certs, String authType) {
```

```
    }

    @Override
    public void checkServerTrusted(
      java.security.cert.X509Certificate[] certs, String authType) {
    }
  } };
}
```

## Build the HTTP Context

Next, build the HTTP context, as shown in Example 12–8. Retrieve the authorization headers using the CookieManager class you instantiated in "Initialize the CookieManager".

***Example 12–8 Building the HTTP Context***

```
Log.i(MyApp.TAG, "Building the HTTP context...");
Map<String, List<String>> headers = new HashMap<String, List<String>>();

HttpContext httpContext = null;

try {
  httpContext = HttpContext.Builder.create()
                .withHeaders(cookieManager.get(url.toURI(), headers))
                .build();
} catch (IOException e) {
  e.printStackTrace();
} catch (URISyntaxException e) {
  e.printStackTrace();
}
```

## Connect to the URL

With your authentication parameters configured, you can now connect to the WebRTC Session Controller URL using the **connect** method of the **UrlConnection** object, as shown in Example 12–9.

***Example 12–9 Connecting to the WebRTC Session Controller URL***

```
try {
  urlConnection.connect();
} catch (IOException e) {
  e.printStackTrace();
}
```

# Configuring Interactive Connectivity Establishment (ICE)

If you have access to one or more STUN/TURN ICE servers, implement the **IceServerConfig** interface, as shown in Example 12–10. For information about ICE, see "Managing Interactive Connectivity Establishment Interval".

***Example 12–10 Configuring the ICE Server Config Class***

```
class MyIceServerConfig implements IceServerConfig {
  public Set<IceServer> getIceServers() {
    Log.i(MyApp.TAG, "Setting up ICE servers...");
    Set<IceServer> iceServers = new HashSet<IceServer>();
    iceServers.add(new IceServerConfig.IceServer(
```

```
            "stun:stun-relay.example.net:3478", "admin", "password"));
        iceServers.add(new IceServerConfig.IceServer(
            "turn:turn-relay.example.net:3478", "admin", "password"));
        return iceServers;
      }
    }
```

## 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 Android API library monitors this connection.

When you instantiate your session object, configure how the functionality in WebRTC Session Controller Android API library checks the WebSocket connection of your application, by setting the following values in the **WSCSession** object:

- **Session.PROP_ACK_INTERVAL**, which specifies the acknowledgement interval. The default is 60,000 milliseconds (ms).
- How often the WebRTC Session Controller Android API library must ping the WebRTC Session Controller Signaling Engine:
  - **WSCSession.PROP_BUSY_PING_INTERVAL**, when there are subsessions inside the session. The default is 3,000 ms.
  - **WSCSession.PROP_IDLE_PING_INTERVAL**, when there are no subsessions inside the session. The default is 10,000 ms.
- **Session.PROP_RECONNECT_INTERVAL**, which specifies the interval between attempts to reconnect to the WebRTC Session Controller Signaling Engine. The default is 2000 ms.
- **Session.PROP_RECONNECT_TIME**, which specifies the maximum time for the interval during which the WebRTC Session Controller Android API library attempts 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 called 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, monitor these values to check the state of the connection. When there is a device handover, your application suspends the application session. The WebSocket connection closes abnormally. See "Suspending the Session on the Original Device".

## Configuring Support for Notifications

Set up client notifications to enable your applications to operate without impacting the battery life and data consumption associated with the associated mobile devices.

With such a setup, whenever a user (for example, Bob) is not actively using your application, your application hibernates the client session. It does so after informing the WebRTC Session Controller server. The WebSocket connection to the WebRTC Session Controller server closes. During that hibernation period, to alert Bob of an

event (such as a call from Alice on the Call feature of your Android application), the WebRTC Session Controller server sends a message (about the call invite) to the cloud messaging server.

The cloud messaging server uses a push notification, a short message that it delivers to a device (such as a mobile phone registered to Bob). This message contains the registration ID for the application and the payload. On being woken up on that device, your application reconnects with the server, uses the saved session ID to resurrect the session data, and handles the incoming event.

If no event occurs during the specified hibernation period and the period expires, there are no notifications to process. The WebRTC Session Controller server cleans up the session.

The preliminary configurations and registration actions that you perform to support client notifications in your applications provide the WebRTC Session Controller server and the cloud messaging provider the necessary information such as the device, the APIs and the application. The client application running on the mobile device or browser retrieves a registration ID from its notification provider, the Google Cloud Messaging service.

## About the WebRTC Session Controller Notification Service

The WebRTC Session Controller Notification Service manages the external connectivity with the respective notification providers. It implements the Cloud Messaging Provider specific protocol such as GCM and APNS. The WebRTC Session Controller Notification Service ensures that all notification messages are transported to the appropriate notification providers.

The WebRTC Session Controller server constructs the payload in the push notification it sends by combining the received message payload from your application with the pay load configured in the application settings or the application provider settings you provide WebRTC Session Controller.

If you plan to use the WebRTC Session Controller server to communicate with the Google Cloud Messaging system, register it with the GCM. See "Enable Your Applications to Use the WebRTC Session Controller Notification Service".

### About Employing Your Current Notification System

At this point, verify if your current installation has an existing notification server that talks to the Cloud Messaging system and that the installation supports applications for your users through this server.

If you currently have such a notification server successfully associated with a cloud messaging system, you can use the pre-existing notification system to send notifications using the REST interface. For more information on the REST interface, see the *Oracle Communications WebRTC Session Controller Extension Developer's Guide*.

## How the Notification Process Works

In its simplest form, the notification process works in this manner:

1. Bob, an end user, accesses your application on a mobile device. In this scenario, your Android Audio Call application.
2. The client application running on the device/browser fetches a registration ID from its notification provider.

3. WebRTC Session Controller Android client SDK sends the information about the client device and the application settings to the WebRTC Session Controller server.

   A WebSocket connection is opened.

4. When there is inactivity on the part of the end user (Bob), your application goes into the background. Your application sends a message to the WebRTC Session Controller server informing the server of its intent to hibernate and specifies a time duration for the hibernation.

   The WebSocket connection closes.

5. During the hibernation period, an event occurs. For example, Alice places a call to Bob on your Android Audio Call application.

6. WebRTC Session Controller server receives this call request from Alice. It checks the session state. Since the call invite request came during the time interval set as the hibernation period, the WebRTC Session Controller server uses its notification service to send a notification to the GCM server.

7. The GCM server delivers the notification to your Android Call application on the mobile device registered to Bob.

8. On receiving this notification,

   - Your Android application reconnects with the notification service using the last session-id and receives the incoming call.
   - WebRTC Session Controller client SDK once again establishes the connection to the server WebRTC Session Controller server.

9. WebRTC Session Controller sends the appropriate notification to your application. The user interface logic in your application informs Bob appropriately.

10. Bob accepts the call.

11. Your application logic manages the call to its completion.

> **Note:** If the time set for the hibernation period completes with no event (such as a call from Alice to Bob), then, the WebRTC Session Controller server closes the session. The Session ID and its data are destroyed.
>
> Your application must create a session. It cannot use the session ID to restore the data.

### Handling Multiple Sessions

If you have defined multiple applications in WebRTC Session Controller and your customer can access more than one such application. As a result, there can be multiple WebRTC Session Controller-associated sessions in the mobile application registered to the customer.

In such a scenario where more than one session data is involved, all the associated session data is stored appropriately and available to your applications.

## The Process Workflow for Your Android Application

The process workflow to support notifications in your Android application are:

1. The prerequisites to using the notification service are complete. See "About the General Requirements to Provide Notifications".

2. Your application on the Android device sends the **registration_Id** to the WebRTC Session Controller Android client SDK, which then sends it to the WebRTC Session Controller server and saves it locally.
3. When a notification is to be sent, the WebRTC Session Controller server sends a message with the **registration_id** to the GCM notification provider.
4. The notification provider delivers this notification to the device.
5. When the notification is clicked on the device, your application is awakened. It re-establishes communication with the WebRTC Session Controller server again and handles the event.

## About the WebRTC Session Controller Android APIs for Client Notifications

The following WebRTC Session Control Android APIs enable your applications to handle notifications related to session hibernation:

- **hibernate**

  The **hibernate** method of the **WSCSession** object starts a hibernate request to the WebRTC Session Controller server.

- **HIBERNATED**

  This enum value of the **SessionState** object indicates that the session is in hibernation.

- **HibernateParams**

  The **HibernateParams** object stores the parameters for the hibernating session.

- **HibernationHandler**

  The **HibernationHandler** interface is associated with a **Session** object. It contains the callback methods for the requests and responses to the session hibernation.

- **withDeviceToken** parameter of **WSCSession.Builder**

  When you provide the **withDeviceToken** parameter, the session is built with the device token obtained from GCM.

- **withSessionId** parameter of **WSCSession.Builder**

  The parameter is used for rehydration. When you provide the **withSessionId** parameter, the session is built with the input session ID.

- **withHibernationHandler** parameter of **WSCSession.Builder**

  When you provide the **withHibernationHandler** parameter, the session is built to handle hibernation.

For more on these and other WebRTC Session Controller Android API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller Android API Reference*.

## About the General Requirements to Provide Notifications

Complete the following tasks as required for your application. Some are performed outside of your application:

- Register with Google
- Obtain the Registration ID for your Application
- Enable Your Applications to Use the WebRTC Session Controller Notification Service

- Inform the Device to Deliver Push Notifications to Your Application
- In your application, complete all actions associated with changes in the activity life cycle, creating, updating, and removing notifications.
- Store the Session ID
- Implement Session Rehydration
- Handling Hibernation Requests from the Server

### Register with Google

Register your WebRTC Session Controller installation with the Google API Console and create a project to receive the following:

1. Project ID
2. API Key

For information about how to complete this task, refer to the *Google Developers Console Help* documentation.

### Obtain the Registration ID for your Application

To obtain a **registrationId**, register your application with GCM. For information about how to complete this task, refer to the *Google Developers Console Help* documentation.

### Enable Your Applications to Use the WebRTC Session Controller Notification Service

This step is performed in the WebRTC Session Controller Administration Console.

Access the **Notification Service** tab in the WebRTC Session Controller Administration Console and enter the information about each application for the use of the WebRTC Session Controller Notification Service. For each application, enter the application setting such as the application ID, API Key, the cloud provider for the API service. For more information about completing this task, see "Creating Applications for the Notification Service" in *WebRTC Session Controller System Administration Guide*.

### Inform the Device to Deliver Push Notifications to Your Application

This step is performed within your Android application.

Ensure that, after your application launches successfully, your application informs the device that it requires push notifications. For information about how to complete this task, refer to the appropriate *Google Developers* documentation.

### Store the Session ID

To persist the Session ID in your application, use the various standard storage mechanisms offered by the Android platforms. Your Android application can use this session ID to immediately present "Bob" (the end user) with the last current state of the application session. The **WSCSession.getSessionId()** method returns the session ID as a String.

For more information, see the description of **WSCSession** in *Oracle Communications WebRTC Session Controller Android API Reference*.

### Implement Session Rehydration

To implement session rehydration in your application:

- Persist Session IDs

  To provides your end users with a seamless user experience, persist the session ID value in your Android applications. Use the various standard storage mechanisms offered by Android Platform do so.

- Use the appropriate Session ID

  Provide the same session ID that the client last successfully connected with when it hibernated. The WebRTC Session Controller Android SDK supports rehydration of its session, when given a session ID.

- Provide the capability to trigger hydration for more than one session object.

  This scenario occurs when you have multiple applications defined in WebRTC Session Controller and your customer creates a session with more than one of these applications in their mobile device. In such a scenario, the client application is using more than one **WSCSession** object.

### Handling Hibernation Requests from the Server

At times your application receives a request to hibernate from the WebRTC Session Controller server. To respond to such a request, provide the necessary logic to handle the user interface and other elements in your application.

See "Responding to Hibernation Requests from the Server" for information about how to set up the callbacks to the specific WebRTC Session Controller Android SDK event handlers.

## Tasks that Use WebRTC Session Controller Android APIs

Use WebRTC Session Controller Android APIs to do the following:

- Associate the Device Token when Building the WebRTC Session.
- Associate the Hibernation Handler for the Session.
- Implement the HibernationHandler Interface.
- Implement Session Hibernation.
- Send Notifications to the Callee when Callee Client Session is in Hibernated State.
- Provide the Session ID to Rehydrate the Session.
- Respond to Hibernation Requests from the Server.

For information about the supported WebRTC Session Controller Android APIs, see *Oracle Communications WebRTC Session Controller Android API Reference*.

### Associate the Device Token when Building the WebRTC Session

Associate the device token when you build a WebRTC Session Controller session using the **WSCSession.Builder** object. To input the device token obtained from GCM, use **withdeviceToken (String** *token*) method.

For example:

```
WSCSession.Builder builder = WSCSession.Builder.create(new URI(webSocketURL))
            ...
            .withDeviceToken("ASDAKSDHUWE12329KDA1233");
WSCSession session = builder.build();
```

See Example 12–18.

For information about **WSCSession.Builder**, see *Oracle Communications WebRTC Session Controller Android API Reference*.

### Associate the Hibernation Handler for the Session

Set up the hibernation handling function when you build a WebRTC Session Controller session. Use the **withHibernationHandler** method of **WSCSession.Builder** as shown here:

```
WSCSession.Builder builder = WSCSession.Builder.create(new URI(webSocketURL))
            ...
            .withHiberationHandler(new MyHibernationHandler());
WSCSession session = builder.build();
```

### Implement the HibernationHandler Interface

Implement the **HibernationHandler** interface to handle the hibernation requests that originate from the server or the client. This interface has the following event handlers:

- **onFailure**: called when a Hibernate request from the client fails.
- **onSuccess**: called when a Hibernate request from the client succeeds.
- **onRequest**: called when there is a request from the server. Returns an instance of **session.HibernateParams**.
- **onRequestCompleted**: called when the request from the server end completes. This event handler uses a **StatusCode** enum value as input parameter.

***Example 12–11 Implementing the HibernationHandler Interface***

```
// Handle hibernation for the session.
 class MyHibernationHandler implements HibernationCallback {

   // On success response for Hibernate requests originated from Client
   public void onSuccess() {
     // perform other cleanup
   }

   // On failure response for Hibernate requests originated from Client
   public void onFailure(StatusCode code) {
     // Hibernate request rejected..
   }

   // On request for Hibernate originated from Server.
   public HibernateData onRequest() {
     // fetch device token if not already
     return HibernateData.of(registrationId, timeToLive);
   }

   // On completion of request for Hibernate originated from Server end.
   public void onRequestCompleted(StatusCode code) {
     // process status code and clean up if OK.
   }
 }
```

For information about **HibernationHandler** and **StatusCode**, see *Oracle Communications WebRTC Session Controller Android API Reference*.

### Implement Session Hibernation

When your Android application is in the background, your application must send a request back to WebRTC Session Controller stating that it wishes to hibernate the session.

Take appropriate steps to release shared resources, invalidate timers, and store the state information necessary to restore your application to its current state, in case it is terminated later.

To start a hibernate request to the WebRTC Session Controller server, call the **WSCSession.hibernate** method. The **HibernateParams** object contains the parameters for hibernating a session. Provide this object when you call the **hibernate** method. Example 12–12 shows how an example application creates a holder for **HibernatedParams** with the **HibernatedParams.of** method when that application starts the hibernate request.

***Example 12–12 Hibernating the Session***

```
WSCSession session = sessionbuilder.build();
...
// Hibernate the session
int timeToLiveInSecs = 3600;
session.hibernate(HibernateParams.of(timeToLiveInSecs, TimeUnit.SECONDS));
```

The WebRTC Session Controller server identifies the client device (going into hibernation) by the **deviceToken** you provided when you built the session object (Example 12–18).

When you call the **hibernate** method, provide the maximum period for which the client session is kept alive on the server. All notifications received within this period are sent to the client device. In Example 12–12, the session called the hibernation method and sets the hibernation period to 3600 seconds. The WebRTC Session Controller server maintains a maximum interval depending on the policy set for each type of client device. If your application sets an interval greater than this period, the server uses the policy-supported maximum interval.

When the **WSCSession.hibernate** method completes, the **SessionState** for the session is **HIBERNATED**. The session with the WebRTC Session Controller server closes. Your application can take no action, such as a call request.

For information about **WSCSession.hibernate** method, see *Oracle Communications WebRTC Session Controller Android API Reference*.

### Send Notifications to the Callee when Callee Client Session is in Hibernated State

If the client session of the callee is in a hibernated state, any incoming event for that client session requires some time for the call setup so that the callee can accept the call. In your Android application, add the logic to the callback function to handle incoming call event when the callee session is in a hibernated state.

---

**Note:** This section describes how to use the WebRTC Session Controller notification API to send the notification. For more information about how the payload is constructed see, "Message Payloads" in *WebRTC Session Controller Extensions Developer's Guide*.

If your application connects to a notification system that exposes a REST API, you can use the REST API Callouts instead.

---

Set up a function to handle the **onWSHibernated** method in the Groovy Script library. This method takes **NotificationContext** object as a parameter.

The **NotificationContext** object serves as a cache and way for notifications and allows notifications to be marked for consumption after the render life cycle has completed. It allows equal access to notifications across multiple interfaces on a page. You can do the following with the **NotificationContext** object:

- Retrieve
  - information about the triggering message, (such as the initiator, the target, package type).
  - Information about the application (ID, version, platform, platform version).
  - The device token.
  - The incoming message that triggered this notification, as a normalized message.
  - The REST client instance for submitting outbound REST requests (synchronized call outs only).
- Dispatch the messages through the internal notification service, if configured.

For more information about **NotificationContext**, see All Classes in *Oracle Communications WebRTC Session Controller Configuration API Reference*.

Example 12–13 shows a sample code excerpt that creates the JSON message in the *msg_payload* object. It uses the **context.dispatch** method to dispatch the message payload through the local notification service.

***Example 12–13 Using Groovy Method to Define the Notification Payload***

```
/**
 * This function gets called when the client end-point is in a hibernated state when an incoming
event arrives for it.
 * A typical action would be to send some trigger/Notification to wake up the client.
 *
 * @param context the notification context
 */
void onWSHibernated(NotificationContext context) {
  // Define the notification payload.
  def msg_payload = "{\"data\" : {\"wsc_event\": \"Incoming " + context.getPackageType() +
          "\", \"wsc_from\": \"" + context.getInitiator() + "\"}}"
  if (Log.debugEnabled) {
    Log.debug("Notification Payload: " + msg_payload)
  }
  // Using local notification gateway
  context.dispatch(msg_payload)
}
```

### Provide the Session ID to Rehydrate the Session

To rehydrate an existing session, use the **withSessionID** property of **WSCSession.Builder**. You can set up an observer for incoming notifications in your application. Pass the stored session ID into the session builder.

The session builder rehydrates the session by retrieving the hibernated session out of persisted storage using the passed session ID as the key.

**Important:** Call this method when attempting to rehydrate an existing session only.

***Example 12–14 Rehydrating an Existing Session***

```
WSCSession.Builder builder = WSCSession.Builder.create(...)
    .withUserName(userName)
       ...
    .withSessionID("S123");

WSCSession session = sessionbuilder.build();
```

### Respond to Hibernation Requests from the Server

When the server has to force your application to hibernate, it calls the **onRequest** method in the **HibernationHandler** interface. When the hibernation request from the server completes, it calls the **onRequestCompleted** method in that **WSCHibernationHandler** interface.

To handle the user interface and other elements in your application, provide the necessary logic in your implementation of **HibernationHandler**.

***Example 12–15 Handling Server-originated Hibernation Requests***

```
WSCSession.Builder builder = WSCSession.Builder.create(new URI(webSocketURL))
             ...
             .withDeviceToken("....")
             .withHiberation(new MyHibernationHandler())
...
// Handle hibernation for the session.
 class MyHibernationHandler implements HibernationCallback {

   // On success response for Hibernate requests originated from Client
   public void onSuccess() {
     // perform other cleanup
   }

   // On failure response for Hibernate requests originated from Client
   public void onFailure(StatusCode code) {
     // Hibernate request rejected..
   }

   // On request for Hibernate originated from Server.
   public HibernateData onRequest() {
     // fetch device token if not already
     return HibernateData.of(registrationId, timeToLive);
   }

   // On completion of request for Hibernate originated from Server end.
   public void onRequestCompleted(StatusCode code) {
     // process status code and clean up if OK.
   }
 }
```

# Creating a WebRTC Session Controller Session

Once you have configured your authentication method and connected to your WebRTC Session Controller endpoint, instantiate a WebRTC Session Controller session object. Before instantiating a session object, configure the following elements:

- To handle the results of a session creation request, you "Implement the ConnectionCallback Interface".
- To monitor and respond to changes in session state, you "Create a Session Observer Object".
- To set up the session, you "Build the Session Object".
- To configure specific session object behaviors and performance parameters, you "Configure Session Properties".

## Implement the ConnectionCallback Interface

You must implement the ConnectionCallback interface to handle the results of your session creation request. The ConnectionCallback interface has two event handlers:

- **onSuccess**: Triggered upon a successful session creation.
- **onFailure**: Returns an enum of type StatusCode. Triggered when session creation fails. For a listing of status code, see *Oracle Communications WebRTC Session Controller Android API Reference*.

***Example 12–16 Implementing the ConnectionCallback Interface***

```
public class MyConnectionCallback implements ConnectionCallback {
  @Override
  public void onFailure(StatusCode arg0) {
    Log.i(MyApp.TAG, "Handle a connection failure...");
  }

  @Override
  public void onSuccess() {
    Log.i(MyApp.TAG, "Handle a connection success...");
  }
}
```

## Create a Session Observer Object

You must create a session Observer object to monitor and respond to changes in session state.

***Example 12–17 Instantiating a Session Observer***

```
public class MySessionObserver extends Observer {
  @Override
  public void stateChanged(final SessionState state) {
    runOnUiThread(new Runnable() {
      @Override
      public void run() {
        Log.i(MyApp.TAG, "Session state changed to " + state);
        switch (state) {
          case CONNECTED:
            break;
          case RECONNECTING:
            break;
```

```
          case FAILED:
            Log.i(MyApp.TAG,
                  "Send events to various active activities as required...");
            shutdownCall();
            break;
          case CLOSED:
          default:
            break;
        }
      }
    });
  }
}
```

## Build the Session Object

With the ConnectionCallback and Session Observer configured, you now build a WebRTC Session Controller session using the session **Builder** method.

***Example 12–18 Building the Session Object***

```
Log.i(MyApp.TAG, "Creating a WebRTC Session Controller session...");
WSCSession.Builder builder = null;
try {
  builder = WSCSession.Builder.create(new java.net.URI(webSocketURL))
           .withUserName(userName)
           .withPackage(new CallPackage())
           .withHttpContext(httpContext)
           .withConnectionCallback(new MyConnectionCallback())
           .withIceServerConfig(new MyIceServerConfig())
           .withObserver(new MySessionObserver());
           .withDeviceToken("MyDeviceToken")
           .withHiberation(new MyHibernationHandler())
} catch (URISyntaxException e) {
  e.printStackTrace();
}

WSCSession session = builder.build();
```

In Example 12–18, the **withPackage** method registers a new **CallPackage** with the session that is instantiated when creating voice or video calls. The device token, **ConnectionCallback**, **IceServerConfig**, **HibernationHandler** and **SessionObserver** objects (created earlier) are also registered.

## Configure Session Properties

You can configure more properties when creating a session using the **withProperty** method.

For a complete list of properties and their descriptions, see the *Oracle Communications WebRTC Session Controller Android SDK API Reference*.

***Example 12–19 Configuring Session Properties***

```
WSCSession.Builder builder = WSCSession.Builder.create(...)
    .withUserName(userName)
       ...
    .withProperty(WSCSession.PROP_RECONNECT_INTERVAL, 5000)
    .withProperty(WSCSession.PROP_IDLE_PING_INTERVAL, 15000));
WSCSession session = sessionbuilder.build();
```

# Adding WebRTC Voice Support to your Android Application

This section describes adding WebRTC voice support to your Android application.

## Initialize the CallPackage Object

When you created your Session, you registered a new **CallPackage** object using the **withPackage** method of the Session object. You now instantiate that **CallPackage**.

***Example 12–20 Initializing the CallPackage***

```
String callType = CallPackage.PACKAGE_TYPE;
CallPackage callPackage = (CallPackage) session.getPackage(callType);
```

---

**Note:** Use the default **PACKAGE_TYPE** call type unless you have defined a custom call type.

---

## Place a WebRTC Voice Call from Your Android Application

Once you have configured your authentication scheme, created a **Session**, and initialized a **CallPackage**, you can place voice calls from your Android application.

### Initialize the Call Object

With the CallPackage object created, initialize a Call object, passing the callee ID as an argument.

---

**Note:** In a production application, integrate with the Android contacts provider or another enterprise directory system, rather than passing a bare string to the **createCall** method. For more information about integrating with the Android contacts provider, see http://developer.android.com/guide/topics/providers/contacts-provider.html.

---

***Example 12–21 Initializing the Call Object***

```
String calleeId = "bob@example.com";
call = callPackage.createCall(calleeId);
```

### Configure Trickle ICE

To improve ICE candidate gathering performance, enable Trickle ICE in your application using the **setTrickleIceMode** method of the Call object. For more information, see "Enabling Trickle ICE to Improve Application Performance".

***Example 12–22 Configuring Trickle ICE***

```
Log.i(MyApp.TAG, "Configure Trickle ICE options, OFF, HALF, or FULL...");
call.setTrickleIceMode(Call.TrickleIceMode.FULL);
```

### Create a Call Observer Object

You next create a **CallObserver** object so you can respond to Call events. Example 12–23 provides a skeleton with the appropriate call update, media, and call states. You can use it to handle updates to, and input from your application accordingly.

***Example 12–23 Creating a CallObserver Object***

```
Create a Call Observer Object
You next create a CallObserver object so you can respond to Call events. Example 12-18 provides a
skeleton with the appropriate call update, media, and call states, which you can use to handle
updates to, and input from, your application accordingly.
Creating a CallObserver Object
public class MyCallObserver extends oracle.wsc.android.call.Call.Observer {
  @Override
  public void callUpdated(final CallUpdateEvent state, final CallConfig callConfig, Cause cause) {
    Log.i(MyApp.TAG, "Call updated: " + state);
    runOnUiThread(new Runnable() {

      @Override
      public void run() {
        switch (state) {
          case SENT:
            break;
          case RECEIVED:
            break;
          case ACCEPTED:
            break;
          case REJECTED:
            break;
          default:
           break;
        }
      }
    });
  }

  @Override
  public void mediaStateChanged(MediaStreamEvent mediaStreamEvent, MediaStream mediaStream) {
    Log.i(MyApp.TAG, "Media State " + mediaStreamEvent
                                          + " for media stream " + mediaStream.label());
  }

  @Override
  public void stateChanged(final CallState state, Cause cause) {
    runOnUiThread(new Runnable() {
      @Override
      public void run() {
        switch (state) {
          case ESTABLISHED:
          Log.i(MyApp.TAG, "Update the UI to indicate that the call has been accepted...");
          break;
        case ENDED:
          Log.i(MyApp.TAG, "Update the UI and possibly close the activity...");
          break;
        case REJECTED:
          break;
        case FAILED:
          break;
        default:
          break;
        }
      }
    });
  }
}
```

### Register the CallObserver with the Call Object

Once you've implemented the **CallObserver**, register it with the Call object.

***Example 12–24 Registering a Call Observer***

```
call.setObserver(new MyCallObserver());
```

### Create a CallConfig Object

You create a **CallConfig** object to determine the type of call you wish to make. The **CallConfig** constructor takes two parameters, both named **MediaDirection**. The first parameter configures an audio call while the second configures a video call:

```
CallConfig(MediaDirection audioMediaDirection, MediaDirection videoMediaDirection)
```

The values for each **MediaDirection** parameter are:

- **NONE**: No direction; media support disabled.
- **RECV_ONLY**: The media stream is receive only.
- **SEND_ONLY**: The media stream is send only.
- **SEND_RECV**: The media stream is bi-directional.

Example 12–25 shows the configuration for a bi-directional, audio-only call.

***Example 12–25 Creating an Audio CallConfig Object***

```
CallConfig callConfig = new CallConfig(MediaDirection.SEND_RECV,
                                       MediaDirection.NONE);
```

### Configure the Local MediaStream for Audio

With the **CallConfig** object created, you configure the local audio **MediaStream** using the WebRTC **PeerConnectionFactory**. For information about the WebRTC SDK API, see https://webrtc.org/native-code/native-apis/.

***Example 12–26 Configuring the Local MediaStream for Audio***

```
Log.i(MyApp.TAG, "Get the local media streams...");
PeerConnectionFactory pcf = call.getPeerConnectionFactory();
mediaStream = pcf.createLocalMediaStream("ARDAMS");
AudioSource audioSource = pcf.createAudioSource(new MediaConstraints());
mediaStream.addTrack(pcf.createAudioTrack("ARDAMSa0", audioSource));
```

### Start the Audio Call

Finally, you start the audio call using the start method of the **Call** object and passing it the **CallConfig** object and the **MediaStream** object.

***Example 12–27 Starting the Audio Call***

```
Log.i(MyApp.TAG, "Start the audio call...");
call.start(callConfig, mediaStream);
```

### Terminating the Audio Call

To terminate the audio call, use the **end** method of the **Call** object:

```
call.end()
```

**Note:** To reclaim any resources that the **MediaStream** object is using, explicitly set the **MediaStream** object to null.

## Receiving a WebRTC Voice Call in Your Android Application

This section configuring your Android application to receive WebRTC voice calls.

### Create a CallPackage Observer

To be notified of an incoming call, create a **CallPackageObserver** and attach it to your **CallPackage**. The CallPackageObserver lets you intercept and respond to changes in the state of the CallPackage.

***Example 12–28 A CallPackage Observer***

```
public class MyCallPackageObserver extends oracle.wsc.android.call.CallPackage.Observer {
  @Override
  public void callArrived(Call call, CallConfig callConfig, Map<String, ?> extHeaders) {

    Log.i(MyApp.TAG, "Registering a call observer...");
    call.setObserver(new MyCallObserver());

    Log.i(MyApp.TAG, "Getting the local media stream...");
    PeerConnectionFactory pcf = call.getPeerConnectionFactory();
    MediaStream mediaStream = pcf.createLocalMediaStream("ARDAMS");
    AudioSource audioSource = pcf.createAudioSource(new MediaConstraints());
    mediaStream.addTrack(pcf.createAudioTrack("ARDAMSa0", audioSource));

    Log.i(MyApp.TAG, "Accept or reject the call...");
    if (answerTheCall) {
      Log.i(MyApp.TAG, "Answering the call...");
      call.accept(callConfig, mediaStream);
    } else {
      Log.i(MyApp.TAG, "Declining the call...");
      call.decline(StatusCode.DECLINED.getCode());
    }
  }
}
```

In Example 12–28, the **callArrived** event handler processes an incoming call request:

1. The method registers a **CallObserver** for the incoming call. In this case, it uses the same CallObserver, **myCallObserver**, from the example in "Create a Call Observer Object".
2. The method then configures the local media stream, in the same manner as the example in "Configure the Local MediaStream for Audio".
3. The **accept** or **decline** method of the Call object is called based on the boolean value of **answerTheCall**.

**Note:** The boolean value of **answerTheCall** can be set by a user interface element in your application such as a button or link.

### Bind the CallPackage Observer to the CallPackage

With the **CallPackageObserver** object created, you bind it to your **CallPackage** object:

```
callPackage.setObserver(new MyCallPackageObserver());
```

# Adding WebRTC Video Support to your Android Application

This section describes how you can add WebRTC video support to your Android application. While the methods are almost identical to adding voice call support to an Android application, more preparation is required.

## Initializing the PeerConnectionFactory Object

You can use the org.webrtc.VideoRendererGui class to initialize the components of the PeerConnectionFactory in the following way:

***Example 12–29 Initializing Android Globals***

```
//initialize Android globals
  PeerConnectionFactory.initializeAndroidGlobals(
      /** Context */this,
      /** enableAudio */true,
      /** enableVideo */true,
      /** hw acceleration */true,
      /** egl context */null
  );

  //create the peerConnectionFactory
  pcf = new PeerConnectionFactory();


  //video controls
  mVideoView = (GLSurfaceView) findViewById(R.id.video_view);

  //set the view on the renderer
  VideoRendererGui.setView(mVideoView, null);

  //set remote and local renderers as follows (for example)
  final VideoRendererGui.ScalingType scalingType =
VideoRendererGui.ScalingType.SCALE_ASPECT_FILL;
  remoteRender = VideoRendererGui.create(0, 0, 100, 100, scalingType, false);
  localRender = VideoRendererGui.create(70, 70, 25, 25, scalingType, true);
```

## Find and Return the Video Capture Device

Before your application tries to initialize a video calling session, verify that the Android device it is running on actually has a video capture device available. Find the video capture device and return a **VideoCapturer** object. For more information about handling the camera of an Android device, see http://developer.android.com/guide/topics/media/camera.html.

***Example 12–30 Finding a Video Capture Device***

```
private VideoCapturer getVideoCapturer() {
  Log.i(MyApp.TAG,
      "Cycle through likely device names for a camera and return the first "
      + "available capture device. Throw an exception if none exists.");

  final String[] cameraFacing = { "front", "back" };
  final int[] cameraIndex = { 0, 1 };
  final int[] cameraOrientation = { 0, 90, 180, 270 };

  for (final String facing : cameraFacing) {
    for (final int index : cameraIndex) {
```

```
      for (final int orientation : cameraOrientation) {
        final String name = "Camera " + index + ", Facing "
                          + facing + ", Orientation " + orientation;
        final VideoCapturer capturer = VideoCapturer.create(name);
        if (capturer != null) {
           Log.i(MyApp.TAG, "Using camera: " + name);
           return capturer;
        }
      }
    }
  }
  throw new RuntimeException("Failed to open a capture device.");
}
```

> **Note:** Example 12–30 is not a robust algorithm for video capturer detection and is not recommended for production use.

## Create a GLSurfaceView in Your User Interface Layout

Your application must provide a container to display a local or remote video feed. To do that, you add an OpenGL **SurfaceView** container to your user interface layout. In Example 12–31, a GLSurfaceView container is created with the ID, *video_view*. For more information about GLSurfaceView containers, see http://developer.android.com/reference/android/opengl/GLSurfaceView.html.

> **Note:** You, of course, customize the GLSurfaceView container for the requirements of your specific application.

***Example 12–31 A Layout Containing a GLSurfaceView Element***

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingBottom="@dimen/activity_vertical_margin"
  android:paddingLeft="@dimen/activity_horizontal_margin"
  android:paddingRight="@dimen/activity_horizontal_margin"
  android:paddingTop="@dimen/activity_vertical_margin"
  tools:context=".MyActivity"
  android:orientation="vertical" >

  <android.opengl.GLSurfaceView
    android:id="@+id/video_view"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="0dp"
    android:layout_weight="1" />
</LinearLayout>
```

## Initialize the GLSurfaceView Control

Next, you initialize the **GLSurfaceView** container by finding its ID in the resource list in your Android application, *video_view*, and creating a **VideoRendererGui** object using the control ID as an argument.

***Example 12–32 Initializing the GLSurfaceView Control***

```
Log.i(MyApp.TAG,"Initialize the video view control in your main layout...");
//video controls
  mVideoView = (GLSurfaceView) findViewById(R.id.video_view);

  //set the view on the renderer
  VideoRendererGui.setView(mVideoView, null);

  //set remote and local renderers as follows (for example)
  final VideoRendererGui.ScalingType scalingType =
VideoRendererGui.ScalingType.SCALE_ASPECT_FILL;
  remoteRender = VideoRendererGui.create(0, 0, 100, 100, scalingType, false);
  localRender = VideoRendererGui.create(70, 70, 25, 25, scalingType, true);
```

**Note:** The **VideoRendererGUI** class is freely available. Use Google Code search to find the latest version.

## Placing a WebRTC Video Call from Your Android Application

To place a video call from your Android application, complete the coding tasks described in the earlier sections:

- Authenticating with WebRTC Session Controller
- Configuring Interactive Connectivity Establishment (ICE) (if required)
- Creating a WebRTC Session Controller Session

In addition, complete the coding tasks for an audio call contained in the sections:

- Initialize the Call Object
- Configure Trickle ICE (if required)
- Create a Call Observer Object
- Register the CallObserver with the Call Object

**Note:** Audio and video call work flows are identical with the exception of media directions, local media stream configuration, and the extra considerations described earlier in this section.

### Create a CallConfig Object

You create a **CallConfig** object as described in "Create a CallConfig Object", in the audio call section, setting both arguments to **MediaDirection.SEND_RECV**.

***Example 12–33 Creating an Audio/Video CallConfig Object***

```
CallConfig callConfig = new CallConfig(MediaDirection.SEND_RECV,
                                       MediaDirection.SEND_RECV);
```

### Configure the Local MediaStream for Audio and Video

With the CallConfig object created, you then configure the local video and audio MediaStream objects using the WebRTC **PeerConnectionFactory**. For information about the WebRTC SDK API, see https://webrtc.org/native-code/native-apis/.

***Example 12–34 Configuring the Local MediaStream for Video***

```
Log.i(MyApp.TAG, "Get the local media streams...");
private MediaStream getLocalMediaStreams(PeerConnectionFactory pcf) {
    if (mediaStream == null) {
      // Create audioSource, audiotrack
      AudioSource audioSource = pcf.createAudioSource(new MediaConstraints());
      AudioTrack localAudioTrack = pcf.createAudioTrack("ARDAMSa0", audioSource);
      // get frontfacingcam
      String frontFacingCam = VideoCapturerAndroid.getNameOfFrontFacingDevice();
      // get video capturer from cam above
      VideoCapturer videoCapturer = VideoCapturerAndroid.create(frontFacingCam);
      // Create videoSource, videoTrack
      localVideoSource = pcf.createVideoSource(videoCapturer, getConstraintsFromConfig());
      VideoTrack localVideoTrack = pcf.createVideoTrack("ARDAMSv0", localVideoSource);
      // get localstreams, add audio/video tracks to it
      mediaStream = pcf.createLocalMediaStream("ARDAMS");
      mediaStream.addTrack(localVideoTrack);
      mediaStream.addTrack(localAudioTrack);
      //render local video
      localVideoTrack.addRenderer(new VideoRenderer(localRender));
    }
    return mediaStream;
  }
```

In Example 12–34, the WebRTC SDK **PeerConnectionFactory** adds both an audio and a video stream to the **MediaStream** object.

## Start the Video Call

Finally, start the audio/video call using the start method of the Call object and passing it the **CallConfig** object and the **MediaStream** object.

***Example 12–35 Starting the Video Call***

```
Log.i(MyApp.TAG, "Start the video call...");
call.start(callConfig, mediaStream);
```

## Terminate the Video Call

To terminate the video call, reinitialize the appropriate objects to reclaim their resources, and use the **end** method of the **Call** object (as with an audio-only call).

***Example 12–36 Terminating the Video Call***

```
Log.i(MyApp.TAG, "Shutting down the call...");
if (videoCapturer != null) {
  videoCapturer.dispose();
  videoCapturer = null;
  videoSource.dispose();
  videoSource = null;
}

call.end();
mVideoView = null;
localRender = null;
mediaStream = null;
```

## Receiving a WebRTC Video Call in Your Android Application

Receiving a video call is identical to receiving an audio call as described here, "Receiving a WebRTC Voice Call in Your Android Application". The only difference is the configuration of the **MediaStream** object, as described in "Configure the Local MediaStream for Audio and Video".

# Supporting SIP-based Messaging in Your Android Application

You can design your Android application to send and receive SIP-based messages using the messaging package in WebRTC Session Controller Android SDK.

To support messaging, define the logic for the following in your application:

- Setup and management of the various activities associated with the states of the various objects, such as the session and the message transfer.
- Enabling users to send or receive messages.
- Handling the incoming and outgoing message data.
- Managing the required user interface elements to display the message content throughout the call session.

## About the Major Classes Used to Support SIP-based Messaging

The following major classes and protocols of the WebRTC Session Controller Android SDK enable you to provide SIP-based messaging support in your Android application:

- **MessagingPackage**

  This package handler enables messaging applications. You can send SIP-based messages to any logged-in user with an object of the **MessagingPackage** class. This object also dispatches received messages to the registered observer.

- **MessagingPackage.Observer**

  This class acts as a listener for incoming messages and their acknowledgements. It holds the following event handlers:

  - **onNewMessage**

    This event handler is called when your application receives a new SIP-based message.

  - **onSuccessResponse**

    This event handler is called when your application receives an accept/positive acknowledgment for a sent message.

  - **onErrorResponse**

    This event handler is called when your application receives a reject/negative acknowledgment for a sent message.

- **MessagingMessage**

  This class is used to hold the payload for SIP-based messaging.

- **withPackage**

  This method belongs to the **WSCSession.Builder** class. It is used to build a session that supports a package, such as the messaging package.

For more on these and other WebRTC Session Controller Android API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller Android API Reference*.

## Setting up the SIP-based Messaging Support in Your Android Application

Complete the following tasks to setup SIP-based messaging support in your Android applications:

1. Enabling SIP-based Messaging
2. Sending SIP-based Messages
3. Handling Incoming SIP-based Messages

### Enabling SIP-based Messaging

To enable SIP-based messaging in your Android application, create and assign an instance of a messaging package.

When you set up the builder for the **WSCSession** class, pass this messaging package in the **withPackage** parameter of the WSCSession builder API, as shown in Example 12–37.

***Example 12–37 Building a Session with a Messaging Package***

```
WSCSession.Builder builder = WSCSession.Builder.create(new java.net.URI(webSocketURL))
            ...
            .withPackage(new MessagingPackage())
            ...;
WSCSession session = sessionbuilder.build();
```

Ensure that you implement the logic for **onSuccess** and **OnFailure** event handlers in the **WSCSession.ConnectionCallback** object. WebRTC Session Controller Android SDK sends asynchronous messages to these event handlers, based on its success or failure to build the session.

### Sending SIP-based Messages

To send a SIP-based message, call the **send** method of the **MessagePackage** object. The **send** method takes two arguments, the string text, and the target.

```
send(String content,String target)
```

***Example 12–38 Sending a SIP-based Message***

```
...
MessagingPackage msgPackage = (MessagingPackage) session.getPackage(MessagingPackage.PACKAGE_TYPE);
 ...
    sendMessage(String text, String destination) {
      msgPackage.send(text, destination, null);
      return true;
    }
 ...
```

In Example 12–38, if *destination* is "bob@example.com" and the *text* is "Hi there Bob!", Bob sees this message from the sending party. No external headers are sent with the message.

### Handling Incoming SIP-based Messages

Set up your application to handle incoming messages and acknowledgements. Register a **MessagingPackage.Observer** to be notified when a new message is received. Use the **setObserver** method of the **MessagePackage** object, as shown in Example 12–39:

***Example 12–39 Registering the Observer for the Message Package***

```
...
WSCSession session;
// Register an observer for listening to incoming messaging events.
MessagingPackage msgPackage = (MessagingPackage) session.getPackage(MessagingPackage.PACKAGE_TYPE);
msgPackage.setObserver(new MyMessagingObserver());
...
```

When a new message comes in, the **onNewMessage** event handler of the **MessagingPackage.Observer** object is called. In the callback function you implement, accept or reject the message received using the appropriate APIs.

Set up the logic to handle the acknowledgements appropriately:

- The **accept** method of the **MessagePackage** object. When the receiver of the message accepts the message, the **onSuccessResponse** event is triggered on the sender's side (that originated the message).
- The **reject** method of the **MessagePackage** object. When the receiver of the message rejects the message, the **onErrorResponse** event is triggered on the sender's side (that originated the message).

***Example 12–40 Example of an Observer Set up for a Message Package***

```
// Class that observes for incoming messages from Messaging.
// This class either accepts or rejects the incoming message using the accept() or
reject() api.
class MyMessagingObserver extends MessagingPackage.Observer {

 public void onNewMessage(MessagingMessage messagingMessage) {
    // Process message contents
    String messageContent = messagingMessage.getContent();

    // Accept the payload
    msgPackage.accept(messagingMessage);
 }

 public void onSuccessResponse(MessagingMessage messagingMessage) {
    // Message got accepted from other side.
 }

 public void onErrorResponse(MessagingMessage messagingMessage, StatusCode
statusCode, String s) {
    // Message got rejected from other side.
 }
}
```

# Adding WebRTC Data Channel Support to Your Android Application

This section describes how you can add WebRTC data channel support to the calls you enable in your Android application. For information about adding voice call support

to an Android application, see "Adding WebRTC Voice Support to your Android Application".

To support calls with data channels, define the logic for the following in your application:

- Setup and management of the various activities associated with the states of the various objects, such as the session and the data transfer.
- Enabling users to make or receive calls with data channels set up with or without the audio and video streams
- Handling the incoming and outgoing data
- Managing the required user interface elements to display the data content throughout the call session.

For more on these and other WebRTC Session Controller Android API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller Android API Reference*.

## About the Major Classes and Protocols Used to Support Data Channels

The following major classes and protocols enable you to provide data channel support in your Android application:

- **Call**

  This object represents a call with any combination of audio, video, and data channel capabilities. It creates a data channel and initializes the **DataTransfer** object for the data channel when the call starts or when accepting the Call if the Call has capability of data channel.

- **CallConfig**

  The **CallConfig** object represents a call configuration. It describes the audio, video, or data channel capabilities of a call.

- **DataChannelOption**

  The **DataChannelOption** object describes the configuration items in the data channel of a call such as whether ordered delivery is required, the stream id, maximum number of retransmissions and so on.

- **DataChannelConfig**

  The **DataChannelConfig** object describes the data channel of a call, including its label and **DataChannelOption**.

- **DataTransfer**

  The **DataTransfer** object manages the data channel. If the **CallConfig** object includes the data channel, the **Call** object creates an instance of the **DataTransfer** object.

  Each **DataTransfer** object manages a **DataChannel** object which is identified by a string label.

- **DataSender**

  A nested class of **DataTransfer**, the **DataSender** object exposes the capability of a **DataTransfer** to send raw data over a data channel. The instance is created by **DataTransfer**.

- **DataReceiver**

A nested class of **DataTransfer**, the **DataReceiver** object exposes the capability of a **DataTransfer** to receive raw data over the established data channel. The instance is created by **DataTransfer**.

- **DataTransfer.Observer**

  The **DataTransfer.Observer** interface acts as an observer of incoming data and state changes for the **DataTransfer** object.

  Your application must implement the **onMessage** method of **DataTransfer.Observer** to be informed of changes in **DataTransfer**.

- **DataTransfer.DataTransferState**

  The **DataTransfer.DataTransferState** stores the status of the **DataTransfer** object as **NONE**, **STARTING**, **OPEN**, or **CLOSED**.

  Your application must implement the **onStateChange** method of **DataTransfer.Observer** to be informed of changes in **DataTransfer**.

For more on these and other WebRTC Session Controller Android API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller Android API Reference*.

## Initialize the CallPackage Object

If, when you created your Session, you registered a new **CallPackage** object using the Session object's **withPackage** method, you now instantiate that **CallPackage**.

***Example 12–41 Initializing the ChatPackage***

```
String callType = CallPackage.PACKAGE_TYPE;
CallPackage callPackage = (CallPackage) session.getPackage(callType);
```

Use the default PACKAGE_TYPE call type unless you have defined a custom call type.

## Sending Data from Your Android Application

To send data from your Android application, complete the coding tasks contained in the following sections.

- Authenticating with WebRTC Session Controller
- Configuring Interactive Connectivity Establishment (ICE) (if required)
- Creating a WebRTC Session Controller Session

Complete the coding tasks for an audio call contained in the following sections:

- Initialize the Call Object
- Configure Trickle ICE (if required)

### Create a Call Observer

You next set up a **CallObserver** object in your application so that you can set up the callback function to handle the response to changes in the **Call**.

***Example 12–42 Create a Call Observer***

```
call.setObserver(new CallObserver());
```

For information about creating the **CallObserver** object, see Example 12–23.

### Configure the Data Channel for the Data Transfers

Configure the data channel to use before you set up the **CallConfig** object.

If your application supports only one data channel in a call, then, set up the label for the data channel using the **DataChannelOption** as shown in Example 12–43.

***Example 12–43 Configuring the Single Data Channel of the Call***

```
DataChannelOption dataChannelOption = new DataChannelOption();
DataChannelConfig dataChannelConfig = new DataChannelConfig("testDataChannel",
dataChannelOption);
```

If your application supports multiple data channels in a call, then, define the **dataChannelConfig** array as a variable parity parameter, commonly known as **varargs**. You can add as many data channels to the **dataChannelConfig** array in your application.

Set up a label for each the data channels. Example 12–44 defines two data channels:

***Example 12–44 Configuring Two Data Channels for the Call***

```
DataChannelConfig dataChannelConfig1 = new DataChannelConfig("testDataChannel_1",
new DataChannelOption());
DataChannelConfig dataChannelConfig2 = new DataChannelConfig("testDataChannel_2",
new DataChannelOption());
```

### Create a CallConfig Object

Having defined the data channel setup for the call, you can now create a **CallConfig** object to determine the type of call you wish to make.

The following constructor sets up the **CallConfig** object to support local audio and video media streams and multiple data channels:

***Example 12–45 Constructor to Support Multiple Calls in CallConfig***

```
public CallConfig(final MediaDirection audioMediaDirection, final MediaDirection
videoMediaDirection, DataChannelConfig... dataChannelConfigs);
```

The following code sample creates a **CallConfig** object for use with the channels defined in Example 12–45 (and no audio or video media stream):

```
CallConfig callConfig = new CallConfig(null, null, dataChannelConfig1,
dataChannelConfig2);
```

If your application supports only one data channel and no audio or video, use the following statement to set up the **CallConfig** object

```
CallConfig callConfig = new CallConfig(null, null, dataChannelConfig);
```

where *dataChannelConfig* is previously defined, as seen in Example 12–43.

If in addition to the data channel, your application must support an audio and/or video stream, configure the local video and audio **MediaStream** objects accordingly. See Example 12–34.

### Register the Observer for the Data Channel

Register the observer for the data channel in the **Call** object. Call the **registerDataTransferObserver** method for the call. Provide the label of the data channel when you do register the observer:

```
...
call.registerDataTransferObserver("testDataChannel", observer);
```

### Set Up the Data Transfer Observer to Send Data

Implement the **onMessage** method of **DataTransfer.Observer** interface to handle received raw data before starting a data channel call, as shown in Example 12–46.

***Example 12–46 Setting Up the Callback Function Before Starting a Call***

```
DataTransfer.Observer observer = new DataTransfer.Observer() {
    @Override
    public void onMessage(ByteBuffer byteBuffer) {
      //handle the raw data;
      System.out.println("handle the raw data!");
    }

    @Override
    public void onStateChange(DataTransfer.DataTransferState state) {
      //Set up logic for the DataTransfer states: CLOSED, OPEN, NONE, STARTING
    ...
    }
  };
call.registerDataTransferObserver("testDataChannel", observer); //call is an
object of Call class
  //Start the data channel call
  call.start(callConfig, mediaStream);
```

### Handle Changes in the State of the Data Transfer

Whenever there is a change in the **DataTransferState**, the **onStateChange** method of **DataTransfer.Observer** is called. In your application, provide the logic to handle the states of the data transfer, represented by the following Enum constants:

- NONE
- STARTING
- OPEN
- CLOSED

See Example 12–46.

### Start the Call

Start the call using the **start** method of the **Call** object and passing it the **CallConfig** object and the **MediaStream** object, as shown in Example 12–47.

***Example 12–47 Starting the Call***

```
Log.i(MyApp.TAG, "Start the data channel...");
call.start(callConfig, mediaStream);
```

### Send the Data Content

You can send data using the **send** method of the **DataSender** in the **DataTransfer** object. The data can be raw data as one of the following

- **ByteBuffer**, using:

```
send(ByteBuffer data)
```

- **byte** array, using:

```
send(byte[] data)
```

- **String**, using:

```
send(String data)
```

Use the label for the data channel to retrieve the **DataTransfer** object from the **Call** object. Set up the **DataSender** object. Verify that the status of the **DataTransfer** object is OPEN, by calling the **getState** method. Call the **send** method of this **DataSender** object to send data. Example 12–48 shows a text message sent by the sample code.

> **Note:** Call the send method, when the status of the **DataTransfer** object is **OPEN**.

***Example 12–48 Sending Data***

```
DataTransfer dataTransfer = call.getDataTransfer(DATA_CHANNEL_LABEL);
// Send Data after verifying that DataTransferState is OPEN.
if(dataTransfer != null && dataTransfer.getState() ==
DataTransfer.DataTransferState.OPEN) {
  dataTransfer.getDataSender().send(content);
  ... // Handle any user interface related activity
} else {
  System.out.println("Data Channel not ready, please wait.");
  }
```

### Terminate the Data Channel in the Call

To terminate the audio call, use the **end** method of the **Call** object:

```
call.end();
```

## Receiving Data Content in Your Android Application

This section describes the steps specific to configuring your Android application to receive WebRTC data transfers.

### Register the Observer for the Receiver of the Data Channel

Register the observer for the data channel in the **Call** object.

```
call.registerDataTransferObserver("testDataChannel", observer);
```

### Set Up the Data Receiver to Receive Incoming Data

To set up the **DataReceiver**, retrieve the **DataTransfer** object by the data channel label in **CallConfig** Object. Then create an observer to register it to the **DataReceiver** to receive the raw data. Implement the **onMessage** method of **DataTransfer.Observer**

interface to handle received raw data before accepting a data channel call, as shown in Example 12–49.

***Example 12–49 Setting Up the Callback Before Accepting a Data Channel Call***

```
DataTransfer.Observer observer = new DataTransfer.Observer() {
    @Override
    public void onMessage(ByteBuffer byteBuffer) {
      //handle the raw data;
      System.out.println("handle the raw data!");
    }

    @Override
    public void onStateChange(DataTransfer.DataTransferState state) {
      //Set up logic for the DataTransfer states: CLOSED, OPEN, NONE, STARTING
    ...
    }

  };
call.registerDataTransferObserver("testDataChannel", observer); //call is an
object of Call class
  //Accept the data channel call
  call.accept(callConfig, mediaStream);
```

### Accept the Call

The command to accept the call is:

```
call.accept(callConfig, mediaStream);
```

See Example 12–49.

## Upgrading and Downgrading Calls

This section describes how you can handle upgrading an audio call to an audio video call and downgrading a video call to an audio-only call in your Android application.

### Handle Upgrade and Downgrade Requests from Your Application

To upgrade from a voice call to a video call, you can bind a user interface element such as a button or link to a class containing the Call update logic using the **setOnClickListener** method of the interface object:

```
myButton.setOnClickListener(new CallUpdateHandler());
```

You handle the upgrade or downgrade workflow in the **onClick** event handler of the **CallUpdateHandler** class. In Example 12–50 the **myButton** object simply serves to toggle video support on and off for the current call object. Once the **CallConfig** object is reconfigured, the actual state change for the call is started using the **update** method of the **Call** object.

***Example 12–50 Handling Upgrade Downgrade Requests from Your Application***

```
class CallUpdateHandler implements View.OnClickListener {
  @Override
  public void onClick(final View v) {
    // Toggle between video on/off
    MediaDirection videoDirection;
    if (call.getCallConfig().shouldSendVideo()) {
```

```
      videoDirection = MediaDirection.NONE;
    } else {
      videoDirection = MediaDirection.SEND_RECV;
    }

    Log.i(MyApp.TAG, "Toggle Video");
    CallConfig callConfig = new CallConfig(MediaDirection.SEND_RECV,
                                           videoDirection);
    MediaStream mediaStream = getLocalMediaStreams(call
                                    .getPeerConnectionFactory());
    try {
       call.update(callConfig, mediaStream);
    } catch (IllegalStateException e) {
       Log.e(MyApp.TAG, "Invalid state", e);
    }
  }
}
```

## Handle Incoming Upgrade Requests

You configure the **callUpdated** method of your **CallObserver** class to handle incoming upgrade requests in the case of a **RECEIVED** state change. See Example 12–23 for the complete CallObserver framework.

In Example 12–51, when the **CallUpdateEventState** is RECEIVED, the application:

- Handles data channel activity with *YourActivityClassName*, an extension of the **Activity** class.
- Creates an **AlertDialog.Builder** with a Yes/No click dialog interface determine the user preference for the upgrade.
- When the Yes button is clicked, the code performs the upgrade.
- When the No button is clicked, the code responds accordingly.

***Example 12–51 Handling an Incoming Upgrade Request***

```
case RECEIVED:
  String mediaConfig = "Video - " + callConfig.getVideoConfig().name();
  new AlertDialog.Builder(YourActivityClassName.this)
    .setIcon(android.R.drawable.ic_dialog_alert)
    .setTitle("Call Update Notfication")
    .setMessage("Do you wish you accept this update: " + mediaConfig + " ?")
    .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
        @Override

        public void onClick(DialogInterface dialog, int which) {
          MediaStream mediaStream =
getLocalMediaStreams(call.getPeerConnectionFactory());

          //Setup the call back to handle received raw data.
          DataTransfer.Observer observer = new DataTransfer.Observer() {
            @Override

            public void onMessage(ByteBuffer byteBuffer) {
              //handle the raw data
            }

            @Override
            public void onStateChange(DataTransfer.DataTransferState state) {
              //Do some handling when Datatransfer state change
```

```
            }
          };

          call.registerDataTransferObserver(DATA_CHANNEL_LABEL, observer);

          //Accept the data channel call
          call.accept(callConfig, mediaStream);
        }
      })

// Update rejected.
  .setNegativeButton("No", new DialogInterface.OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
        call.decline(StatusCode.DECLINED.getCode());
      }
    })
  .show();
break;
```

# Handling Session Rehydration When the User Moves to Another Device

When your customer is using your application on one device (a cellphone), the customer could move to another device (a laptop softphone that uses the same account and is authenticated by WebRTC Session Controller). A Session (along with its subsessions) currently active in your application on one device belonging to your customer becomes active on your application on another device belonging to the same customer.

For example, your customer Alice, accesses a web browser from a cellphone to talk about a purchase selection with Bob, a customer support representative active in that browser session. While Alice is on the call, she switches over to her laptop to look at the purchase selection in greater detail.

You can use the WebRTC Session Controller to configure applications that support handovers of session information between devices successfully. Your application then manages the rehydration of the session and all its data on the target device (in this example, the tablet laptop) such that the call from Alice to Bob continues in an uninterrupted fashion.

This section described how your application can work to present the customer with the session state recreated on another device.

---

**Note:** In a device-handover scenario, WebRTC Session Controller manages the data associated with the subsessions of your application session. It keeps their states intact through the handovers that occur during the life of an application session.

The focus of the handover logic in your application is the Session within which a call, a message, or a video session is alive.

---

## About the Supported Operating Systems

You can design your applications using WebRTC Session Controller such that you support handover in your applications programmed for the Android, Web, and iOS systems.

**Note:** For such a handover to be successful, your application must be active on the various devices belonging to a user, the associated user name and account be authenticated by WebRTC Session Controller, and the applications supported on the various operating systems.

This chapter deals with setting up your Android application to support handing using the WebRTC Session Controller Android SDK. For information about supporting handovers to:

- Web applications, see "About Using the WebRTC Session Controller JavaScript API".
- iOS applications, see "Developing WebRTC-Enabled iOS Applications".

## Configuring WebRTC Session Controller to Support Transfer of Session Data

In a device handover, the same WebSocket sessionID is used to transfer an application session state that is active in the current client device (for example, the cellphone registered to Alice) and present that state on the subsequent device (her laptop).

When one client uses another client's WebSocket sessionID to connect with WebRTC Session Controller, the WSC server checks the value in the system property, **allowSessionTransfer**. The default value of **allowSessionTransfer** is **false**. This value causes WebRTC Session Controller to consider the request as a hacking attack and reject the request.

In order to allow the same user or tenant to connect with the WebRTC Session Controller server using the same WebSocket session ID, set the startup command option **allowSessionTransfer** to **true** in the WebRTC Session Controller. For more information, see the description about "Supporting Session Rehydration for Device Handover Scenarios" in *WebRTC Session Controller System Administrator's Guide*.

## About the WebSocket Disconnection

When the device handover occurs, the WebSocket connection immediately closes.

The WebRTC Session Controller signaling engine keeps the session alive for a time period specified as **WebSocket Disconnect Time Limit** in the WebRTC Session Controller Administration Console.

**Note:** If the target device fails to pick up the session within the **WebSocket Disconnect Time Limit** period, the device handover fails.

## About the Normalized Session Data User to Support Handovers

A user could move from an application where your application uses one type of SDK to a device where your application uses a different SDK. The supported Client SDKs are:

- Web
- Android
- iOS

WebRTC Session Controller supports a normalized uniform session data format to transfer the session state information between these systems. The session state information is sent as a binary large object (BLOB).

## About the Handover Scenario on the Original Device

When the original device (for example, the cellphone *Device A-1* registered to Alice) triggers a handover, the following events occur:

1. **On the Original Device:**

   a. Your application on *DeviceA-1* suspends the active session on the WebRTC Session Controller server.

      See "Suspending the Session on the Original Device".

   b. Your application transfers the session data (**stateInfo**) to be received and processed by the application on the subsequent device *Device A-2*.

      See "Sending the Session Data to the Application Service".

   The WebRTC Session Controller Signaling engine keeps this session alive for a time period specified as **WebSocket Disconnect Time Limit** in the WebRTC Session Controller Administration Console.

2. **On the device receiving the handover**

   The subsequent device that receives the handover is laptop *DeviceA-2* (registered to Alice) in this discussion. The following events occur:

   a. Your application on the subsequent device retrieves the **stateInfo** string from the Restful server. See "Requesting for the Session Data from the Application Service".

   b. Your application uses the session state information to recreate the session. See "Recreating the Application Session with the StateInfo Object".

   c. Your application rehydrates the call with WebRTC Session Controller Android SDK.

      See "Rehydrating a WebRTC Call After a Device Handover".

   d. The active call resumes on the subsequent device.

## About the WebRTC Session Controller Android APIs for Device Handover

The WebRTC Session Control Android APIs that enable your applications to handle notifications related to session Rehydration in another device are:

- **withStateInfo** parameter of **WSCSession.Builder**

  When you provide the **withStateInfo** parameter, the WebRTC Session Controller Android SDK rehydrates the session using the StateInfo object. If there is a call subsession in the StateInfo object, WebRTC Session Controller Android SDK rehydrates the call.

- **suspend**

  The **suspend** method of the **WSCSession** object suspends the active session. This method return JSON string object containing the session data to use in session rehydration

For more on these and other WebRTC Session Controller Android API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller Android API Reference*.

## Completing the Tasks to Support Session Rehydration in Another Supported Device

This section described the tasks to complete in your application to hand over a session to another device and to receive a session handed over from another device. Complete the following tasks to support session transfers and rehydration with the transferred session state information:

- Suspending the Session on the Original Device
- Sending the Session Data to the Application Service
- Requesting for the Session Data from the Application Service
- Recreating the Application Session with the StateInfo Object
- Rehydrating a WebRTC Call After a Device Handover

### Suspending the Session on the Original Device

In order to implement a handover, your application on the original device *DeviceA-1* suspends the active session on the WebRTC Session Controller server. One scenario would be to set up a *handover* function and suspend the session within its logic.

> **Note:** The logic surrounding the detection of the actual device handover is beyond the scope of this document.

The WebRTC Session Controller Android API method to suspend a session is **suspend()**. The WebRTC Session Controller Android SDK API returns the session data to your application in JSON string format. The WebSocket connection closes.

In your application logic that handles the user interface related to the handover, call the WebRTC Session Controller Android API method **WSCSession.suspend()**, as shown in Example 12–52.

***Example 12–52 Suspending a Session***

```
...
// Handover Triggered
public void handOver handover() {
...
    String sessionData = currentSession.suspend();
...
}
```

### Sending the Session Data to the Application Service

Your application on the original device (for example, the cellphone called *Device A-1* registered to Alice) sends the session state information in a handover request to the Application Service (your application, a Web application, or a RESTful service).

You can configure how your application performs this task in the way that suits your environment. For example, your application can push the **stateInfo** to the other device or allow the other device to pull the **stateInfo**.

When the **suspend** method completes, your application has the session data with which to rehydrate the session. The session data is in JSON format. In your implementation of the logic for the **WSCSessionHandedOver** state of **WSCSession**, set up the information to transfer to the Application service.

Include this JSON string object and any other relevant information in the data you send with the handover request to the application service.

### Requesting for the Session Data from the Application Service

In the application logic that handles this trigger, send a request to the application service asking for the session state information, as shown in Example 12–53. The application service returns the **stateInfo**, the session data for rehydration in a JSON String format.

***Example 12–53 Requesting for StateInfo from the Application Service***

```
StringBuilder urlSB = new StringBuilder();
    urlSB.append(SDKHelper.getInstance().getHandoverServerURL());
    urlSB.append("/");
    urlSB.append(operation);
    urlSB.append("/");
    urlSB.append(userName);
    String url = urlSB.toString();
    final HttpClient client = new DefaultHttpClient();
    HttpResponse response = null;
    try {
      final HttpGet httpGet = new HttpGet(url);
      response = client.execute(httpGet);
    } catch (IOException e) {
      Log.e(TAG, "Got exception when handling HandOver state info for: " +
userName, e);
    }
String resp = null;
    if(response == null || response.getStatusLine().getStatusCode() !=
HttpStatus.SC_OK) {
      Log.d(TAG, "Failed to " + KEY_OPERATION_ADD_ID + " HandOver state info for:
" + userName);
    } else {
      try {
        resp = EntityUtils.toString(response.getEntity());
      } catch (IOException e) {
        Log.e(TAG, "Got exception when handling HandOver state info for: " +
userName, e);
      }
    }
```

### Recreating the Application Session with the StateInfo Object

Your application on the subsequent device sends the session state information to the WebRTC Session Controller Android SDK.

In your application logic that handles session rehydration following a device handover, set up a session object with *dhSessionStateInfo*. This is the session state configuration you received from the application service in Example 12–53.

Set up the Builder for the WSCSession by providing the *dhSessionStateInfo* in the **withStateInfo** method of the **WSCSession.Builder** object. Then call the **build** method to recreate the session, as shown in Example 12–54.

***Example 12–54 Building the Session with StateInfo***

```
...
WSCSession.Builder builder = WSCSession.Builder.create(new URI(webSocketURL))
             ...
             .withStateInfo(dhSessionStateInfo);
```

```
WSCSession session = builder.build();
...
```

Ensure that you implement the logic for **onSuccess** and **OnFailure** event handlers in the **WSCSession.ConnectionCallback** object. WebRTC Session Controller Android SDK sends asynchronous messages to these event handlers, based on its success or failure to build the session.

### Rehydrating a WebRTC Call After a Device Handover

A call session that was part of the application session on the original device, is also suspended when that device suspends the application session. In such a scenario, WebRTC Session Controller Android SDK creates the subsession objects for your application. For example, if there is a **Call** object, it passes the call configuration object to the **CallObserver** object in your application.

> **Note:** After the new device connects to the session, WebRTC Session Controller does not send out a new incoming call request to connect a call that is part of the handover. To recreate the call connection, the WebRTC Session Controller Android SDK uses the information in the **stateInfo** object.
>
> Your application can rehydrate a session where the call between the two users is in an already established state on the device receiving the handover. If there is no "re-invite" flow, the ongoing call is not established, because the client ip, port, or codec has changed.
>
> In your application, complete the candidates re-negotiation with the peer side, as dictated by the Android system.

Implement the following logic in your Android application:

- To be informed of the updated call configuration that results from the handover, implement the **CallUpdated** method in the **Call.Observer** object.
- To handle the rehydrated call, implement the **callResurrected(Call rehydratedCall)** method in the **CallPackage.Observer** object.

***Example 12–55 Handling a Rehydrated Call***

```
@Override
    public void callResurrected(final Call call) {
      call.setObserver(new CallObserver());
      call.setPeerConnectionFactory(pcf);
      call.start(call.getCallConfig(),
getLocalMediaStreams(call.getPeerConnectionFactory()));
    }
```

Following a device handover, the general workflow for rehydrating a call with video streams or data channels is identical to the workflow for rehydrating a call with audio stream. Any specificity lies in how your application logic handles the **CallConfig** object to maintain the video streams and data transfers associated with the call.

# Extending Your Applications with WebRTC Session Controller Android SDK

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

> **Note:** Before you proceed, review the discussion about how the WebRTC Session Control JavaScript APIs assist in extending WebRTC applications. See "Extending Your Applications Using WebRTC Session Controller JavaScript API" for more information.

You can extend the your Android applications by adding a sub session to a WSC Session. See "About the Classes and Methods Used to Extend Android Applications".

## About the Classes and Methods Used to Extend Android Applications

The following class and methods enable you to extend Android applications:

- **Frame**

  This class is the master data transfer object class for all JSON messages.

- **FrameFactory**

  This class is the helper class for creating JSON Frame instances.

- **Headers**

  This class is the master data transfer object class for sending a JSON string as the Message section.

- **Payload**

  This class is the master data transfer object class for sending a JSON string as the Payload section.

- **Control**

  This class is the master data transfer object class for sending a JSON string as the Control section.

- **WSCSession**
- **Call**
- **MessagingPackage**

## Extending WebRTC Session Controller Android Applications

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

- **Frame**
- **FrameFactory**

  This class is the helper class for creating JSON Frame instances.

### Extending Your Session Application Using the Session Object

Your application session can have one or more subsessions. In order to send a message, you can add a subsession to a session. The following methods enable you to extend your application session.

- **sendMessage**

  This method sends the message as a **Frame** object to WebRTC Session Controller server over a WebSocket.

- **putSubSession**

  This method adds a sub session to the WSCSession.

- **generateSubSessionId**

  This method Generates random UUID according to RFC 4122 v4.

- **getSubSession**

  This method retrieves a subsession, when given the subsession id.

- **getSubSessions**

  This method retrieves a Collection of subsessions that belong to the session.

- **removeSubSession**

  This method removes a sub session with the input session ID from **WSCSession**.

***Example 12–56 Sending a Message Using a SubSession***

```
self.wscSession = [self createWSCSession];

MySubSession *mySubSession = [self createSubSession];
[self.wscSession putSubSession:mySubSession];

WSCFrame *myFrame = [self frameFromFactory:mySubSession];
[self.wscSession sendMessage:myFrame];
```

### Extending Your Application Using Extension Headers

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

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

This formatted object is paced in the message formatted as:

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

For more information, see "About Extra Headers in Messages".

Place the extension header when you call the methods that support extension headers. The extension headers are inserted into the JSON message.

The following class elements support extension headers

- **CallPackage.Observer**

  The **callArrived** event of the **CallPackage.Observer** supports extension headers.

- **Call**

  – **accept**

  - **decline**
  - **start**
  - **update**
  - **end**
- **MessagePackage**
  - **accept**
  - **reject**
  - **send**

# 13

# Developing WebRTC-Enabled iOS Applications

This chapter shows how you can use the Oracle Communications WebRTC Session Controller iOS application programming interface (API) library to develop WebRTC-enabled iOS applications. The library is delivered in a WebRTC Session Controller SDK framework.

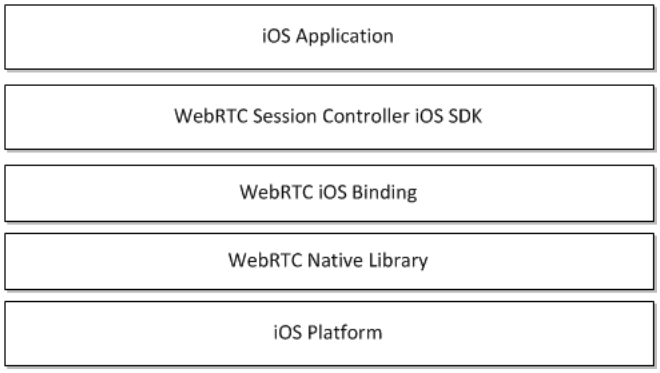
## About the iOS SDK

The WebRTC Session Controller iOS SDK enables you to integrate your iOS applications with core WebRTC Session Controller functions. You can use the iOS SDK to implement the following features:

- Audio calls between an iOS application and any other WebRTC-enabled application, a Session Initialization Protocol (SIP) endpoint, or a public switched telephone network endpoint using a SIP trunk.
- Video calls between an iOS application and any other WebRTC-enabled application, with suitable video conferencing support.
- Seamless upgrading of an audio call to a video call and downgrading of a video call to an audio call.
- Peer to peer data transfers between an iOS application and any other WebRTC-enabled application.
- Support client notifications when the device goes into hibernation. When the server sends a notification to wake up the client, the application can reestablish the connection, rehydrate the session.
- Support for Interactive Connectivity Establishment (ICE) server configuration, including support for Trickle ICE.
- Transparent session reconnection following network connectivity interruption.
- Session rehydration following a device handover.

The WebRTC Session Controller iOS SDK is built upon several additional libraries and modules as shown in Figure 13–1.

***Figure 13–1 iOS SDK Architecture***




The WebRTC iOS binding enables the WebRTC Session Controller iOS SDK access to the native WebRTC library which itself provides WebRTC support. The Socket Rocket WebSocket library enables the WebSocket access required to communicate with WebRTC Session Controller.

For additional information about any of the APIs used in this document, see *Oracle Communications WebRTC Session Controller iOS API Reference*.

## Supported Architectures

The WebRTC Session Controller iOS SDK is compliant with iOS9 (arm64) mobile operating system.

## About the iOS SDK WebRTC Call Workflow

The general workflow for using the WebRTC Session Controller iOS SDK to place a call is:

1. Authenticate against WebRTC Session Controller using the **WSCHttpContext** class. You initialize the **WSCHttpContext** with necessary HTTP headers and optional **SSLContextRef** in the following manner:
   a. Send an HTTP GET request to the login URI of WebRTC Session Controller.
   b. Complete the authentication process based on your authentication scheme.
   c. Proceed with the WebSocket handshake on the established authentication context.
2. Establish a WebRTC Session Controller session using the **WSCSession** class.

   Two protocols must be implemented:

   - **WSCSessionConnectionDelegate**: A delegate that reports on the success or failure of the session creation.
   - **WSCSessionObserverDelegate**: A delegate that signals on various session state changes, including **CLOSED**, **CONNECTED**, **FAILED**, and others.
3. Once a session is established, create a **WSCCallPackage** class which manages **WSCCall** objects in the **WSCSession**.
4. Create a **WSCCall** using the **WSCCallPackage createCall** method with a callee ID as its argument, for example, alice@example.com.

5. To monitor call events such as **ACCEPTED**, **REJECTED**, **RECEIVED**, implement a **WSCCallObserver** protocol which attaches to the **WSCCall**.
6. To maintain the nature of the WebRTC call, create a **WSCCallConfig** object with one of the following settings:
   - Bi-directional or mono-directional audio or video.
   - Bi-directional or mono-directional audio and video.
   - Message exchanges containing raw data.
7. Create and configure a **RTCPeerConnectionFactory** object and start the **WSCCall** using the **start** method.
8. When the call is complete, terminate the call using the **end** method of the **WSCCall** object.

## Prerequisites

Before continuing, make sure you thoroughly review and understand the JavaScript API discussed in the chapters listed below:

- About Using the WebRTC Session Controller JavaScript API
- Setting Up Security
- Setting Up Audio Calls in Your Applications
- Setting Up Video Calls in Your Applications
- Setting Up Data Transfers in Your Applications

The WebRTC Session Controller iOS SDK is closely aligned in concept and functionality with the JavaScript SDK to ensure a seamless transition.

In addition to an understanding of the WebRTC Session Controller JavaScript API, you are expected to be familiar with:

- Objective C and general object-oriented programming concepts
- General iOS SDK programming concepts including event handling, delegates, and views.
- The functionality and use of XCode.

For an introduction to programming iOS applications using XCode and for more background on all areas of iOS application development, see:

`https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/WhereToGoFromHere.html#/apple_ref/doc/uid/TP40011343-CH12-SW1`

## iOS SDK System Requirements

In order to develop applications with the WebRTC Session Controller SDK, complete the following software/hardware requirements:

- An installed and fully configured WebRTC Session Controller installation. See the *Oracle Communications WebRTC Session Controller Installation Guide*.
- A Macintosh computer capable of running XCode version 5.1 or later.
- An actual iOS hardware device.

You can test the general flow and function of your iOS WebRTC Session Controller application using the iOS simulator. To utilize audio and video functionality fully, a physical iOS device such as an iPhone or an iPad is required.

## About the Examples in This Chapter

In order to illustrate the functionality of the WebRTC Session Controller iOS SDK API, the examples and descriptions in this chapter are kept intentionally straightforward. The examples assume no pre-existing interface schemas except when necessary, and then, only with the barest minimum of code. For example, if a particular method requires arguments such as a user name, the code examples show a plain string *userName* such as "*bob@example.com*" being passed to the method. It is assumed that in a production application, you would interface with the contact manager of the iOS device.

# Installing the iOS SDK

To install the WebRTC Session Controller iOS SDK, do the following:

1. Install XCode from the Apple App store:

   https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/index.html#/apple_ref/doc/uid/TP40011343-CH2-SW1

   **Note:** The WebRTC Session Controller iOS SDK requires XCode version 6 or higher.

2. Create an iOS project using xCode, adding any required targets:

   https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/FirstTutorial.html#/apple_ref/doc/uid/TP40011343-CH3-SW1

   **Note:** iOS version 7 is the minimum required by the WebRTC Session Controller iOS SDK for full functionality.

3. Download and extract the WebRTC Session Controller iOS SDK compressed (.zip) file. There are three subfolders in the archive, **debug**, **docs** and **release**.
   - The **debug** folder contains the debug frameworks.
   - The **docs** folder contains the iOS API Reference.
   - The **release** folder contains the release frameworks.
4. Add the WebRTC Session Controller SDK frameworks to your project.
   a. Select your application target in XCode project navigator.
   b. Select the **Build Phases** tab in the top of the editor pane.
   c. Expand **Link Binary With Libraries**.
   d. Depending on your requirements, drag the framework files from either **release** or **debug** to **Link Binary with Libraries**.

   **Note:** The **webrtc** folder contains a single library that supports iOS devices and the iOS simulator.

5. Import any other system frameworks you require. The following frameworks are recommended:
   - **CFNetwork.framework**: zero-configuration networking services. For more information, see:

     https://developer.apple.com/library/ios/documentation/CFNetwork/Reference/CFNetwork_Framework/index.html
   - **Security.framework**: General interfaces for protecting and controlling security access. For more information, see:

     https://developer.apple.com/library/ios/documentation/Security/Reference/SecurityFrameworkReference/index.html
   - **CoreMedia.framework**: Interfaces for playing audio and video assets in an iOS application. For more information, see:

     https://developer.apple.com/library/mac/documentation/CoreMedia/Reference/CoreMediaFramework/index.html
   - **GLKit.framework**: Library that facilitates and simplifies creating shader-based iOS applications (useful for video rendering). For more information, see:

     .https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/DrawingWithOpenGLES/DrawingWithOpenGLES.html
   - **AVFoundation.framework**: A framework that facilities managing and playing audio and video assets in iOS applications. For more information, see:

     https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/00_Introduction.html#/apple_ref/doc/uid/TP40010188
   - **AudioToolbox.framework**: A framework containing interfaces for audio playback, recording, and media stream parsing. For more information, see:

     https://developer.apple.com/library/ios/documentation/MusicAudio/Reference/CAAudioTooboxRef/index.html#/apple_ref/doc/uid/TP40002089
   - **libicucore.dylib**: A unicode support library. For more information, see:

     http://icu-project.org/apiref/icu4c40/
   - **libsqlite3.dylib**: A framework providing a SQLite interface. For more information, see:

     https://developer.apple.com/technologies/ios/data-management.html
6. If you are targeting iOS version 8 or above, add the **libstdc++.6.dylib.a** framework to prevent linking errors.

## Authenticating with WebRTC Session Controller

You use the **WSCHttpContext** class to set up an authentication context. The authentication context contains the necessary HTTP headers and SSLContext information, and is used when setting up a wsc.Session.

## Initialize a URL Object

You then create an NSURL object using the URL to your WebRTC Session Controller endpoint.

***Example 13–1 Initializing a URL Object***

```
NSString *urlString=*"http://server:port/login?wsc_app_uri=/ws/webrtc/myapp";
NSURL authUrl=[NSURL URLWithString:urlString];
```

## Configure Authorization Headers

Configure authorization headers as required by your authentication scheme. The following example uses Basic authentication; OAuth and other authentication schemes are similarly configured.

***Example 13–2 Initializing Basic Authentication Headers***

```
NSString *authType = @"Basic ";
NSString *username = @"username";
NSString *password = @"password";
NSString * authString = [authType stringByAppendingString:[username
                                 stringByAppendingString:[@":"
                                 stringByAppendingString:[password]]];
```

**Note:** If you are using Guest authentication, no headers are required.

## Connect to the URL

With your authentication parameters configured, you can now connect to the WebRTC Session Controller URL using **sendSynchronousRequest**, or **NSURlRequest**, and **NSURlConnection**, in which case the error and response are returned in delegate methods.

***Example 13–3 Connecting to the WebRTC Session Controller URL***

```
NSHTTPURLResponse * response;
NSError * error;authUrlNSMutableURLRequest *loginRequest = [NSMutableURLRequest requestWithURL:];
[loginRequest setValue:authString forHTTPHeaderField:@"Authorization"];
[NSURLConnection sendSynchronousRequest:loginRequest returningResponse:&response error:&error];
```

## Configure the SSL Context

If you are using Secure Sockets Layer (SSL), configure the SSL context, using the **SSLCreateContext** method, depending upon whether the URL connection was successful. For more information about **SSLCreateContext**, see the following Apple developer content:

https://developer.apple.com/library/mac/documentation/Security/Reference/secureTransportRef/index.html#/apple_ref/c/func/SSLCreateContext

***Example 13–4 Configuring the SSLContext***

```
if (error) {
      // Handle an error..
       NSLog("The following error occurred: %@", error.description);
} else {

  // Configure the SSLContext if necessary...
```

```
  SSLContextRef sslContext = SSLCreateContext(NULL, kSSLClientSide, kSSLStreamType);
  // Copy the SSLContext configuration to the httpContext builder...
  [builder withSSLContextRef:&sslContext];

  ...
}
```

## Retrieve the Response Headers from the Request

Depending upon the results of the authentication request, you retrieve the response headers from the URL request and copy the cookies to the httpContext builder.

***Example 13–5 Retrieving the Response Headers from the URL Request***

```
if (error) {
      // Handle an error..
       NSLog("The following error occurred: %@", error.description);
} else {

  // Configure the SSLContext if necessary, from Example 13-4...
  SSLContextRef sslContext = SSLCreateContext(NULL, kSSLClientSide, kSSLStreamType);
  // Copy the SSLContext configuration to the httpContext builder...
  [builder withSSLContextRef:&sslContext];

  // Retrieve all the response headers...
  NSDictionary *respHeaders = [response allHeaderFields];
  WSCHttpContextBuilder *builder = [WSCHttpContextBuilder create];
  // Copy all cookies from respHeaders to the httpContext builder...
  [builder withHeader:key value:headerValue];

  ...
}
```

## Build the HTTP Context

Depending upon the results of the authentication request, you then build the **WSCHttpContext** using **WSCHttpContextBuilder**.

***Example 13–6 Building the HttpContext***

```
if (error) {
      // Handle an error..
       NSLog("The following error occurred: %@", error.description);
} else {

  // Configure the SSLContext if necessary, from Example 13-4...
  SSLContextRef sslContext = SSLCreateContext(NULL, kSSLClientSide, kSSLStreamType);
  // Copy the SSLContext configuration to the httpContext builder...
  [builder withSSLContextRef:&sslContext];

  // Retrieve all the response headers from Example 13-5...

  // Build the httpContext...
  WSCHttpContext *httpContext = [builder build];

  ...
}
```

# Configure Interactive Connectivity Establishment (ICE)

If you have access to one or more STUN/TURN ICE servers, you can initialize the **WSCIceServer** class. For details on ICE, see "Managing Interactive Connectivity Establishment Interval".

***Example 13–7 Configuring the WSCIceServer Class***

```
WSCIceServer *iceServer1 = [[WSCIceServer alloc] initWithUrl:@"stun:stun-server:port"];
WSCIceServer *iceServer2 = [[WSCIceServer alloc] initWithUrl:@"turn:turn-server:port",
                                                        @"admin", @"password"];
WSCIceServerConfig *iceServerConfig = [[WSCIceServerConfig alloc]
                                          initWithIceServers: iceServer1, iceServer2, NIL];
```

# Configuring Support for Notifications

Set up client notifications to enable your applications to operate without impacting the battery life and data consumption with the associated mobile devices.

Whenever a user (for example, Bob) is not actively using your application, your application can hibernate the client session after informing the WebRTC Session Controller server. The WebSocket connecting your application to the WebRTC Session Controller server closes. During that hibernation period, if Bob needs to be alerted of an event (such as a call from Alice on the Call feature of your iOS application), the WebRTC Session Controller server sends a message (about the call invite) to the cloud messaging server.

The cloud messaging server uses a push notification, a short message that it delivers to the specific device (such as a mobile phone). This message contains the registration ID for the application and the payload. On being woken up on that device, your application reconnects with the server, uses the saved session ID to resurrect the session data, and handles the incoming event.

If no event occurs during the specified hibernation period and the period expires, there are no notifications to process and the WebRTC Session Controller server cleans up the session.

The preliminary configurations and registration actions that you perform to support client notifications in your applications provide the WebRTC Session Controller server and the cloud messaging provider the necessary information about the device, the APIs, the application, and so on. The client application running on the mobile device or browser retrieves a registration ID from its notification provider.

## About the WebRTC Session Controller Notification Service

The WebRTC Session Controller Notification Service manages the external connectivity with the respective notification providers. It implements the Cloud Messaging Provider specific protocol such as Apple Push Notification service (APNs). The WebRTC Session Controller Notification Service ensures that all notification messages are transported to the appropriate notification providers.

The WebRTC Session Controller server constructs the payload in the push notification it sends by combining the received message payload from your application with the pay load configured in the application settings or the application provider settings you provide to WebRTC Session Controller.

If you plan to use the WebRTC Session Controller server to communicate with the APNs system, then you must register it with Apple. See "The Notification Process Workflow for Your iOS Application".

### About Employing Your Current Notification System

At this point, verify if your current installation has an existing notification server that talks to the Cloud Messaging system and that the installation supports applications for your users through this server.

If you currently have such a notification server successfully associated with a cloud messaging system, you can use the pre-existing notification system to send notifications using the REST interface. For more information, see the *Oracle Communications WebRTC Session Controller Extension Developer's Guide*.

## How the Notification Process Works

In its simplest form, the notification process works in this manner:

1. Bob, a customer, accesses your application on his mobile device. For example, assume this is your iOS Audio Call application.
2. The client application running on the device/browser fetches a device token from its notification provider.
3. WebRTC Session Controller iOS SDK sends the information about the client device and the application settings to the WebRTC Session Controller server.

   A WebSocket connection is opened.
4. When there is no activity on the part of the customer (Bob), your application goes into the background. Your application sends a message to the WebRTC Session Controller server informing the server of its intent to hibernate and specifies a time duration for the hibernation.

   The WebSocket connection closes.
5. During the hibernation period an event occurs. For example, Alice makes a call to Bob on your iOS Audio Call application.
6. WebRTC Session Controller server receives this call request from Alice and checks the session state. Since the call invite request came during the time interval set as the hibernation period for that session, the WebRTC Session Controller server uses its notification service to send a notification to the APNs server.
7. The APNs server delivers the notification to your ioS Call application on the mobile device.
8. On receiving this notification,
   - Your iOS application reconnects with the notification service using the last session-id and receives the incoming call.
   - WebRTC Session Controller iOS SDK once again establishes the connection to the server WebRTC Session Controller server.
9. WebRTC Session Controller sends the appropriate notification to your application. The user interface logic in your application informs Bob appropriately.
10. Bob accepts the call.
11. Your application logic manages the call to its completion.

**Note:** If the time set for the hibernation period completes with no event (such as the call from Alice to Bob), then, the WebRTC Session Controller Server closes the session.

The Session Id and its data are destroyed. Your application must create another session. Your application cannot use that session ID cannot be used to restore the data.

### Handling Multiple Sessions

If you have defined multiple applications in WebRTC Session Controller, your customer may have accessed more than one such application. As a result, there may be multiple WebRTC Session Controller-associated sessions associated with the application.

In such a scenario where data for more than one session is involved, all of the associated session data is stored appropriately and can be retrieved by your application instances.

## The Notification Process Workflow for Your iOS Application

The process workflow to support notifications in your iOS application are:

1. The prerequisites to using the notification service are complete. See "About the General Requirements to Provide Notifications".
2. Your application on the iOS device sends the **registration_Id** to the WebRTC Session Controller iOS SDK, which then sends it to the WebRTC Session Controller server and saves it locally for future use.
3. When a notification is to be sent, the WebRTC Session Controller server sends a message with the **deviceToken** to the appropriate notification provider (APNs).

   Internally, the WebRTC Session Controller iOS SDK passes the device and operating system information about the client to the server to determine what features the client is able to support.
4. The notification provider (APNs) forwards the notification to the device.
5. When the notification is clicked on the device, your application is awakened. It re-establishes communication with the WebRTC Session Controller server again and handles the event.

**Note:** Apple allows a payload up to 256 bytes for pre- iOS8 systems and 2Kilobytes for later iOS8 systems).

Additionally, the notifications can do one of the following:

- Display a short text message
- Play a brief sound
- Display a number in a badge on the application icon
- Display a short text message

## About the WebRTC Session Controller APIs for Client Notifications

The WebRTC Session Control iOS API objects that enable your applications to handle notifications related to session hibernation are:

- **hibernate**

  The **hibernate** method of the **WSCSession** object sends a hibernate request to the WebRTC Session Controller server.

- **WSCSessionStateHibernated**

  The enum value of the **WSCSessionState** object indicating that the session is in hibernation.

- **WSCHibernateParams**

  The **WSCHibernateParams** object stores the parameters for the hibernating session.

- **withDeviceToken** parameter of **WSCSessionBuilder**

  When you provide the **withDeviceToken** parameter, the session is built with the device token obtained from APNs.

- **withHibernationHandler** parameter of **WSCSessionBuilder**

  When you provide the **withHibernationHandler** parameter, the session is built to handle hibernation.

- **withSessionId** method for the **WSCSessionBuilder** object

  Used for rehydration. When you provide the **withSessionId** parameter, the session is built with the input session ID.

- **WSCHibernationHandler.h**, a new protocol file

For more on these and other WebRTC Session Controller iOS API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller iOS API Reference*.

## About the General Requirements to Provide Notifications

Complete the following tasks as required for your application. Some are performed outside of your application:

- Registering with Apple Push Notification Service
- Obtaining the Device Token
- Enabling Your Applications to Use the WebRTC Session Controller Notification Service
- Informing the Device to Deliver Push Notifications to Your Application
- All actions with respect to changes in the activity life cycle, creating, updating, and removing notifications and so on.
- Storing the Device Token
- Storing the Session ID
- Implement Session Rehydration

### Registering with Apple Push Notification Service

Register your WebRTC Session Controller installation with the Apple Push Notification service to set up the following:

1. SSL certificate to communicate with the APN service.
2. Provisioning profile for the application.

For information about completing these tasks, refer to the *Local and Remote Notification Programming Guide* in the iOS Developer Library at

https://developer.apple.com/library/ios/navigation/

### Obtaining the Device Token

The device token is similar to a phone number and is used in the push notification. The Apple Push Notification service uses this token to locate the specific device on which your iOS application is installed.

Your application should register with Apple Push Notification service to obtain a **deviceToken**.

For information about how to register with the notification service and obtain a device token, refer to the *Local and Remote Notification Programming Guide* in the *iOS Developer Library* documentation.

### Enabling Your Applications to Use the WebRTC Session Controller Notification Service

Access the **Notification Service** tab in the WebRTC Session Controller Administration Console and enter the information about each application for the use of the WebRTC Session Controller Notification Service. For each application, enter the application settings such as the application ID, API Key, the cloud provider for the API service. For more information about completing this task, see "Creating Applications for the Notification Service" in *WebRTC Session Controller System Administrator's Guide*.

### Informing the Device to Deliver Push Notifications to Your Application

Ensure that, after your application launches successfully, your application informs the device that it requires push notifications.

In Example 13–8, the application registers for push notifications.

***Example 13–8 Registering for Push Notifications***

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions
{

// Application lets the device know it wants to receive push notifications
  [[UIApplication sharedApplication] registerForRemoteNotificationTypes:
   (UIRemoteNotificationTypeBadge | UIRemoteNotificationTypeSound |
UIRemoteNotificationTypeAlert)];

NSDictionary *appDefaults = [NSDictionary
                              dictionaryWithObject:[NSNumber numberWithBool:YES]
                              forKey:@"CacheDataAgressively"];
  [[NSUserDefaults standardUserDefaults] registerDefaults:appDefaults];
  [self registerDefaultsFromSettingsBundle];

  if (launchOptions != nil)
  {
    NSDictionary *dictionary = [launchOptions
objectForKey:UIApplicationLaunchOptionsRemoteNotificationKey];
    if (dictionary != nil)
    {
      NSLog(@"Launched from push notification: %@", dictionary);
      //[self addMessageFromRemoteNotification:dictionary updateUI:NO];
    }
```

```
  }
  return YES;
}
```

**Note:** iOS7.0 and later versions support silent remote notification where the silent notification wakes up the application in the background so that it can get new data from the server.

If you remove the Sound, Badge and Alert from the **registerForRemoteNotificationTypes** call by sending in an empty set, the push service should send your notifications silently.

### Storing the Device Token

After you have successfully registered with Apple Push Notification service, store the device token in your application.

The Example 13–9 code excerpt follows from Example 13–8 and shows how an application stores the device token and reports any failure to do so.

***Example 13–9 Storing the Token Device***

```
(BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  // Let the device know we want to receive push notifications
 ...
return YES;
}
 ...
-(void)application:(UIApplication*)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken {
  DLog(@"My token is: %@", deviceToken);
  [SampleIOSUtils setDeviceToken:deviceToken];
}

- (void)application:(UIApplication*)application
didFailToRegisterForRemoteNotificationsWithError:(NSError*)error {
  NSLog(@"Failed to get token, error: %@", error);
}
```

The following excerpt from an application shows sets up the logic to get and set device tokens.

```
...
static NSString * const DEVICE_TOKEN_KEY = @"DeviceToken";
static NSData *deviceToken;
...
+(NSData *)getDeviceToken {
  return [[NSUserDefaults standardUserDefaults] objectForKey:DEVICE_TOKEN_KEY];
}
+(void)setDeviceToken:(NSData *)token {
  [[NSUserDefaults standardUserDefaults] setObject:token forKey:DEVICE_TOKEN_KEY];
  [[NSUserDefaults standardUserDefaults] synchronize];
}
```

### Storing the Session ID

Your application should use the various standard storage mechanisms offered by the iOS platforms to persist the session ID. Your application can use this session ID to immediately present "Bob" (the customer) with the last current state of Bob's session with your application.

The **getSessionId()** method of **WSCSession** object returns the session identifier as an NSString object. For more information, see the description of **WSCSession** in *Oracle Communications WebRTC Session Controller iOS API Reference*.

### Implement Session Rehydration

To implement session rehydration in your application:

- Persist Session Ids

  To provides your customers with a seamless user experience, persist the session ID value in your application. Use the various standard storage mechanisms offered by iOS platform to store user credentials in your iOS applications.

- Use the appropriate Session ID

  Provide the same session ID that the client last successfully connected with when it hibernated. The WebRTC Session Controller iOS SDK uses the session ID to rehydrate the session. It uses stored credentials to authenticate the client session.

- Provide for the ability to trigger hydration for more than one session object.

  This scenario occurs when you have multiple applications defined in WebRTC Session Controller and your customer creates a session with more than one of the WebRTC Session Controller applications in their mobile application. In such a scenario, the client application is using more than one WSCSession(s), each with its own session ID.

### Handling Hibernation Requests from the Server

At times your application receives a request to hibernate from the WebRTC Session Controller server. To respond to such a request, provide the necessary logic to handle the user interface and other elements in your application.

See "Responding to Hibernation Requests from the Server" for information on how to set up the callbacks to the specific WebRTC Session Controller iOS SDK event handlers.

## Tasks that Use WebRTC Session Controller iOS APIs

Use WebRTC Session Controller iOS APIs to do the following:

- Associate the Device Token when Building the WebRTC Session
- Associate the Hibernation Handler for the Session
- Implement the HibernationHandler Interface
- Implement Session Rehydration
- Send Notifications to the Callee when the Client Session is in Hibernated State
- Provide the Session ID to Rehydrate the Session
- Responding to Hibernation Requests from the Server

For information about the WebRTC Session Controller iOS APIs used in the following sections, see *Oracle Communications WebRTC Session Controller iOS API Reference*.

### Associate the Device Token when Building the WebRTC Session

Associate the device token when you build a WebRTC Session Controller session. The **withdeviceToken** method returns a **WSCSessionBuilder** object with a device token configuration:

```
-(WSCSessionBuilder *)withDeviceToken:(NSData *)token;
```

Pass the stored device token when you initialize the session, as shown in Example 13–17.

In the following sample code excerpt, the application provides a stored device token that it retrieves from its *SampleIOSUtils* interface.

```
...
self.wscSession = [[[[[[[[[[[[[[WSCSessionBuilder create:url]
...
                              withDeviceToken:[SampleIOSUtils getDeviceToken]
...
```

For information about **WSCSessionBuilder**, see *Oracle Communications WebRTC Session Controller iOS API Reference*.

### Associate the Hibernation Handler for the Session

Set up the hibernation handling function when you build a WebRTC Session Controller session. The **withHibernationHandler** method returns a **WSCSessionBuilder** object with a hibernation handler configuration:

```
- (WSCSessionBuilder *)withHibernationsHandler:(id<WSCHibernationHandler>)handler
```

See Example 13–17.

In the following sample code excerpt, the application allocates and initializes a hibernation handler:

```
...
self.wscSession = [[[[[[[[[[[[[[WSCSessionBuilder create:url]
...
                       withHibernationsHandler:[[WSCHibernationHandler alloc] init]
...
```

### Implement the HibernationHandler Interface

Implement the **HibernationHandler** interface to handle the hibernation requests that originate from the server or the client. The **HibernationHandler** interface has the following event handlers:

- **onFailure**: Called when a hibernate request from the client fails.
- **onSuccess**: Called when a hibernate request from the client succeeds.
- **onRequest**: Called when there is a request from the server to hibernate the client. Returns an instance of **WSCHibernateParams**.
- **onRequestCompleted**: Called when the request from the server end completes. This event handler uses a **WSCStatusCode** enum value as input parameter.

***Example 13–10 Sample HibernationHandler Implementation***

```
-(void)onSuccess {
// Hibernate success. Store the recent sessionId
  DLog("Success Hibernating");
```

```
  WSCSession *session = [SampleIOSUtils getActiveSession];
  [SampleIOSUtils setLastSessionId:[session getSessionId]];
  [SampleIOSUtils logout];
}

-(void)onFailure:(WSCStatusCode)code {
// Hibernate failed. Issue alert
  DLog("Error Hibernating, status code: %ld",(long)code);
  //[SampleIOSUtils showAlertBox];
  [SampleIOSUtils logout];

}

// Return the timetolive param set for the hibernation
-(WSCHibernateParams *)onRequest {
  WSCHibernateParams *params = [[WSCHibernateParams alloc] initWithTTL:12000];
  return params;
}

-(void)onRequestCompleted:(WSCStatusCode)code {

  if(code == WSCStatusCodeOk ){
// WebRTC Session Controller Status 200.. Store the recent sessionId
    WSCSession *session = [SampleIOSUtils getActiveSession];
    [SampleIOSUtils setLastSessionId:[session getSessionId]];
    DLog("Success Hibernating, status code:%ld",(long)code);
  } else {
// WebRTC Session Controller Not OK.. Show Alert
    DLog("Error Hibernating, status code: %ld",(long)code);
    //[SampleIOSUtils showAlertBox];
  }
  [SampleIOSUtils logout];

}
```

For information about **WSCHibernationHandler** and **WSCStatusCode**, see *Oracle Communications WebRTC Session Controller iOS API Reference*.

### Implement Session Hibernation

When your iOS application is in the background, your application must send a request back to WebRTC Session Controller stating that it wants to hibernate the session.

Use the appropriate method to release shared resources, invalidate timers, and store the state information necessary to restore your application to its current state, in case it is terminated later. For information about handling the applications life cycle see the UIApplicationDelegate reference section of the *iOS Developer Library*.

The WebRTC Session Controller iOS SDK provides the following method in the **WSCSession** object.

```
-(void)hibernate:(WSCHibernateParams *) params;
```

Provide the hibernation period with the **WSCHibernateParams** object, using its static **of** method to create a holder for the time interval and its unit. For information about the **hibernate** method, see *Oracle Communications WebRTC Session Controller iOS API Reference*.

Example 13–11 illustrates how to initiate a hibernate request to the WebRTC Session Controller server.

***Example 13–11 Hibernating the Session***

```
-(void)applicationDidEnterBackground:(UIApplication *)application
{
  WSCSession *session = //get current active session
  // Hibernate the session
  [session hibernate:[session.hibernationHandler]];
}
```

The WebRTC Session Controller server identifies the client device (going into hibernation) by the **deviceToken** you provided when building the session object (Example 13–17).

When you implement the **hibernate** method, provide the maximum period for which the client session should be kept alive on the server. All notifications received within this period are sent to the client device. The WebRTC Session Controller server maintains a maximum interval depending on the policy set for each type of client device. If your application sets an interval greater than this period, the server uses the policy-supported maximum interval.

When the **hibernate** method completes, the **WSCSessionState** for the session is **WSCSessionStateHibernated**. The session with the WebRTC Session Controller server closes. Your application can take no action, such as making a call request.

For information about **hibernate** method, see the description about **WSCSession** in *Oracle Communications WebRTC Session Controller iOS API Reference*.

### Send Notifications to the Callee when the Client Session is in Hibernated State

If the client session for the callee is in a hibernated state, any incoming event for that client session may require some time for the call setup so that the callee can accept the call. In your iOS application, add the logic to the callback function to handle incoming call event when the session for the callee is in a hibernated state.

---

**Note:** This section describes how to use the WebRTC Session Controller notification API to send the notification.

If your application connects to a notification system that exposes a REST API, you can use REST API Callouts instead.

---

Set up a function to handle the **onWSHibernated** method provided by the Groovy Script library. This method takes **NotificationContext** object as a parameter.

The **NotificationContext** object serves as a cache and way for notifications and allows notifications to be marked for consumption after the render life cycle has completed. It allows equal access to notifications across multiple interfaces on a page. Use this object to do the following:

- Retrieve
  - information about the triggering message, (such as the initiator, the target, package type)
  - Information about the application (Id, version, platform, platform version)
  - The device token
  - The incoming message that triggered this notification, as a normalized message

  - The REST client instance for submitting outbound REST requests (synchronized call outs only).
- Dispatch the messages through the internal notification service, if configured.

For more information about **NotificationContext**, see All Classes in *Oracle Communications WebRTC Session Controller Configuration API Reference*.

Example 13–12 shows a sample code excerpt that creates the JSON message in the *msg_payload* object. It uses the **context.dispatch** method to dispatch the message payload through the local notification service.

***Example 13–12 Using Groovy Method to Define the Notification Payload***

```
/**
 * This function gets invoked when the client end-point is in a hibernated state when an incoming
event arrives for it.
 * A typical action would be to send some trigger/Notification to wake up the client.
 *
 * @param context the notification context
 */
void onWSHibernated(NotificationContext context) {
  // Define the notification payload.
  def msg_payload = "{\"data\" : {\"wsc_event\": \"Incoming " + context.getPackageType() +
          "\", \"wsc_from\": \"" + context.getInitiator() + "\"}}"
  if (Log.debugEnabled) {
    Log.debug("Notification Payload: " + msg_payload)
  }
  // Using local notification gateway
  context.dispatch(msg_payload)
}
```

### Provide the Session ID to Rehydrate the Session

To rehydrate an existing session, use the stored session ID. The **withSessionId** method to create a session with a stored session ID (*value*) is:

```
-(WSCSessionBuilder *)withSessionId:(NSString *)value;
```

> **Important:** Invoke this method when attempting to rehydrate an existing session only.

As shown in Example 13–13, you can set up a listener for push notification in your application. Pass the session ID received in the push notification into the session builder. The session builder rehydrates the session by retrieving the hibernated session out of persisted storage using the passed **sessionId** as the key.

***Example 13–13 Rehydrating an Existing Session***

```
-(void)application:(UIApplication*)application didReceiveRemoteNotification:(NSDictionary*)userInfo
{
  NSLog(@"Received notification: %@", userInfo);

  NSString *sessionId = //Parse sessionID from userInfo object

  ...

  // Build a new session object containing the sessionId pulled from userInfo object
  WSCSessionBuilder *builder = [[WSCSessionBuilder alloc] init:webSocketURL]];
```

```
  [[[[[builder withHibernationsHandler:hibernationHandler]
                                  ...
                          withSessionId: sessionId];
  ...
  // Build the new session object
  WSCSession *session = [builder build];

}
```

### Responding to Hibernation Requests from the Server

When the server has to force your application to hibernate, it calls the **onRequest** method in the **WSCHibernationHandler**. When the hibernation request from the server completes, it calls the **onRequestCompleted** method in that **WSCHibernationHandler**.

Provide the necessary logic in your implementation of **WSCHibernationHandler** to handle the user interface and other elements in your application, as shown in Example 13–14.

***Example 13–14 Handling Server-originated Hibernation Requests***

```
@protocol WSCHibernationHandler <NSObject>

-(void)onSuccess;

-(void)onFailure:(WSCStatusCode)code;

-(WSCHibernateParams *)onRequest;

-(void)onRequestCompleted:(WSCStatusCode)code;

@end
```

# Creating a WebRTC Session Controller Session

Once you have configured your authentication method and connected to your WebRTC Session Controller endpoint, you can instantiate a WebRTC Session Controller session object.

## Implement the WSCSessionConnectionDelegate Protocol

You must implement the **WSCSessionConnectionDelegate** protocol to handle the results of your session creation request. See Example 13–15. The **WSCSessionConnectionDelegate** protocol has two event handlers:

- **onSuccess**: Triggered upon a successful session creation.
- **onFailure**: Returns a failure status code. Triggered when session creation fails.

***Example 13–15 Implementing the WSCSessionConnectionDelegate Protocol***

```
#pragma mark WSCSessionConnectionDelegate
-(void)onSuccess {
  NSLog(@"WebRTC Session Controller session connected.");
    NSLog(@"Connection succeeded. Continuing...");
 }
```

```
-(void)onFailure:(enum WSCStatusCode)code {
  switch (code) {
    case WSCStatusCodeUnauthorized:
      NSLog(@"Unable to connect. Please check your credentials.");
      break;
    case WSCStatusCodeResourceUnavailable:
      NSLog(@"Unable to connect. Please check the URL.");
      break;
    default:
      // Handle other cases as required...
      break;
  }
}
```

## Implement the WSCSession Connection Observer Protocol

Create a **WSCSessionConnectionObserver** protocol to monitor and respond to changes in session state, as shown in Example 13–16.

***Example 13–16 Implementing the WSCSessionConnectionObserver Protocol***

```
#pragma mark WSCSessionConnectionDelegate
-(void)stateChanged:(WSCSessionState) sessionState {
   switch (sessionState) {
     case WSCSessionStateConnected:
       NSLog(@"Session is connected.");
       break;
     case WSCSessionStateReconnecting:
       NSLog(@"Session is attempting reconnection.");
       break;
     case WSCSessionStateFailed:
       NSLog(@"Session connection attempt failed.");
       break;
     case WSCSessionStateClosed:
       NSLog(@"Session connection has been closed.");
       break;
     default:
       break;
   }
}
```

## Build the Session Object and Open the Session Connection

With the connection delegate and connection observer configured, you now build a WebRTC Session Controller session and open a connection with the server, as shown in Example 13–17.

***Example 13–17 Building the Session Object and Opening the Session Connection***

```
if (error) {
      // Handle an error..
       NSLog("The following error occurred: %@", error.description);
} else {

  // Configure the SSLContext if necessary, from Example 13-4...
  ...

  // Retrieve all the response headers from Example 13-5...
  ...
```

```
  // Build the httpContext from Example 13-6...
  ...

  NSString *userName = @"username";
  self.wscSession = [[[[[[[[[[[[WSCSessionBuilder create:urlString]
                                                withConnectionDelegate:WSCSessionConnectionDelegate]
                                               withUserName:userName]
                                              withObserverDelegate:WSCSessionConnectionObserverDelegate]
                                             withPackage:[[WSCCallPackage alloc] init]]
                                            withHttpContext:httpContext]
                                           withIceServerConfig:iceServerConfig]
                                           withObserverDelegate:[WSCSessionObserverDelegate]
                                           withHibernationsHandler:[WSCHibernationHandler]
                                           withDeviceToken:["MydeviceToken"]
                                         build];
  // Open a connection to the server...
  [self.wscSession open];
```

In Example 13–17, note that the **withPackage** method registers a new **WSCCallPackage** with the session that will be instantiated when creating voice or video calls.

## Configure Additional WSCSession Properties

You can configure additional properties when creating a session using the **WSCSessionBuilder withProperty** method, as shown in Example 13–18.

***Example 13–18 Configuring WSCSession Properties***

```
if (error) {
      // Handle an error..
       NSLog("The following error occurred: %@", error.description);
} else {

  // Configure the SSLContext if necessary, from Example 13-4...
  ...

  // Retrieve all the response headers from Example 13-5...
  ...

  // Build the httpContext from Example 13-6...
  ...

  self.wscSession = [[[[[[[[[[[[WSCSessionBuilder create:urlString]
                ...
              withProperty:WSC_PROP_IDLE_PING_INTERVAL value:[NSNumber numberWithInt: 20]]
            withProperty:WSC_PROP_RECONNECT_INTERVAL value:[NSNumber numberWithInt:10000]]
           ...
         build];
       [self.wscSession open];
}
```

For a complete list of properties see the *Oracle Communications WebRTC Session Controller iOS SDK API Reference*.

# Adding WebRTC Voice Support to your iOS Application

This section describes how you can add WebRTC voice support to your iOS application.

## Initialize the CallPackage Object

When you created your Session, you registered a new **WSCCallPackage** object using the **withPackage** method of the Session object. You now instantiate that **WSCCallPackage**, as shown in Example 13–19.

***Example 13–19 Initializing the CallPackage***

```
WSCCallPackage *callPackage = (WSCCallPackage*)[wscSession getPackage:PACKAGE_TYPE_CALL];
```

**Note:** Use the default **PACKAGE_TYPE_CALL** call type unless you have defined a custom call type.

## Place a WebRTC Voice Call from Your iOS Application

Once you have configured your authentication scheme, and created a Session, you can place voice calls from your iOS application.

### Add the Audio Capture Device to Your Session

Before continuing, in order to stream audio from your iOS device you initialize a capture session and add an audio capture device, as shown in Example 13–20.

***Example 13–20 Adding an Audio Capture Device to Your Session***

```
- (instancetype)initAudioDevice
{
  self = [super initAudioDevice];
  if (self) {
   self.captureSession = [[AVCaptureSession alloc] initAudioDevice];
   [self.captureSession setSessionPreset:AVCaptureSessionPresetLow];

   // Get the audio capture device and add to our session.
   self.audioCaptureDevice = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeAudio];
   NSError *error = nil;
   AVCaptureDeviceInput *audioInput = [AVCaptureDeviceInput
                                deviceInputWithDevice:self.audioCaptureDevice error:&error];
   if (audioInput) {
    [self.captureSession addInput:audioInput];
   }
   else {
    NSLog(@"Unable to find audio capture device : %@", error.description);
   }

  return self;
}
```

### Initialize the Call Object

Now, with the **WSCCallPackage** object created, you then initialize a **WSCCall** object, passing the callee ID as an argument, as shown in Example 13–21.

***Example 13–21 Initializing the Call Object***

```
String callee = @"bob@example.com";
WSCCall *call = [callPackage createCall:callee];
```

### Configure Trickle ICE

To improve ICE candidate gathering performance, you can choose to enable Trickle ICE in your application using the **setTrickleIceMode** method of the **WSCCall** object, as shown in Example 13–22.

***Example 13–22 Configuring Trickle ICE***

```
NSLog(@"Configure Trickle ICE options, WSCTrickleIceModeOFF,
WSCTrickleIceModeHalf, or WSCTrickleIceModeFull...");
[call setTrickleIceMode: WSCTrickleIceModeFull];
```

For more information, see "Enabling Trickle ICE to Improve Application Performance".

### Create a WSCCallObserverDelegate Protocol

You create a **WSCCallObserverDelegate** protocol, as shown in Example 13–23, so you can respond to the following **WSCCall** events:

- **callUpdated**: Triggered on incoming and outgoing call update requests.
- **mediaStateChanged**: Triggered on changes to the **WSCCall** media state.
- **stateChanged**: Triggered on changes to the **WSCCall** state.
- **onDataTransfer**: Triggered when a **WSCDataTransfer** object is created.

***Example 13–23 Creating a WSCCallObserverDelegate Protocol***

```
#pragma mark WSCCallObserverDelegate

-(void)callUpdated:(WSCCallUpdateEvent)event
                   callConfig:(WSCCallConfig *)callConfig
                   cause:(WSCCause *)cause
{
  NSLog("callUpdate request with config: %@", callConfig.description);
  switch(event){
    case WSCCallUpdateEventSent:
      break;
    case WSCCallUpdateEventReceived:
      NSLog("Call Update event received for config: %@", callConfig.description);
      break;
    case WSCCallUpdateEventAccepted:
      NSLog("Call Update accepted for config: %@", callConfig.description);
      break;
    case WSCCallUpdateEventRejected:
      NSLog("Call Update event rejected for config: %@", callConfig.description);
      break;
    default:
    break;
  }
}

-(void)mediaStateChanged:(WSCMediaStreamEvent)mediaStreamEvent
     mediaStream:(RTCMediaStream *)mediaStream
{
  NSLog(@"mediaStateChanged : %u", mediaStreamEvent);
}

-(void)stateChanged:(WSCCallState)callState
     cause:(WSCCause *)cause
{
```

```
   NSLog(@"Call State changed : %u", callState);
  switch (callState) {
     NSLog(@"stateChanged: %u", callState);
    case WSCCallStateNone:
       NSLog(@"stateChanged: %@", @"WSC_CS_NONE");
      break;
    case WSCCallStateStarted:
       NSLog(@"stateChanged: %@", @"WSC_CS_STARTED");
      break;
    case WSCCallStateResponded:
       NSLog(@"stateChanged: %@", @"WSC_CS_RESPONDED");
      break;
    case WSCCallStateEstablished:
       NSLog(@"stateChanged: %@", @"WSC_CS_ESTABLISHED");
      break;
    case WSCCallStateFailed:
       NSLog(@"stateChanged: %@", @"WSC_CS_FAILED");
      break;
    case WSCCallStateRejected:
       NSLog(@"stateChanged: %@", @"WSC_CS_REJECTED");
      break;
    case WSCCallStateEnded:
       NSLog(@"stateChanged: %@", @"WSC_CS_ENDED");
      break;
    default:
      break;
  }
}
```

### Register the WSCCallObserverDelegate Protocol with the Call Object

You register the **WSCCallObserverDelegate** protocol with the **WSCCall** object, as shown in Example 13–24.

***Example 13–24 Registering a WSCCallObserverDelegate Protocol***

```
call.observerDelegate = WSCCallObserverDelegate;
```

### Create a WSCCallConfig Object

You create a **WSCCallConfig** object to determine the type of call you wish to make. The **WSCCallConfig** constructor takes two parameters, **audioMediaDirection** and **videoMediaDirection**. The first parameter configures an audio call while the second configures a video call.

The values for **audioMediaDirection** and **videoMediaDirection** parameters are:

- **WSCMediaDirectionNone**: No direction; media support disabled.
- **WSCMediaDirectionRecvOnly**: The media stream is receive only.
- **WSCMediaDirectionSendOnly**: The media stream is send only.
- **WSCMediaDirectionSendRecv**: The media stream is bi-directional.

Example 13–25 shows the configuration for a bi-directional, audio-only call.

***Example 13–25 Creating an Audio CallConfig Object***

```
WSCCallConfig *callConfig = [[WSCCallConfig alloc]
initWithAudioVideoDirection:WSCMediaDirectionSendRecv
video:WSCMediaDirectionNone];
```

### Configure the Local MediaStream for Audio

With the **WSCCallConfig** object created, you then configure the local audio MediaStream using the WebRTC **PeerConnectionFactory**, as shown in Example 13–26.

***Example 13–26 Configuring the Local MediaStream for Audio***

```
RTCPeerConnectionFactory *)pcf = [call getPeerConnectionFactory];
RTCMediaStream* localMediaStream = [pcf mediaStreamWithLabel:@"ARDAMS"];
[localMediaStream addAudioTrack:[pcf audioTrackWithID:@"ARDAMSa0"]];
NSArray *streamArray = [[NSArray alloc] initWithObjects:localStream, nil];
```

For information about the WebRTC SDK API, see https://webrtc.org/native-code/native-apis/.

### Start the Audio Call

Finally, you start the audio call using the **start** method of the **WSCCall** object and passing it the **WSCCallConfig** object and the **streamArray**, as shown in Example 13–27.

***Example 13–27 Starting the Audio Call***

```
[call start:callConfig streams:streamArray];
```

### Terminating the Audio Call

To terminate the audio call, use the **WSCCall** object **end** method:

```
[call end];
```

## Receiving a WebRTC Voice Call in Your iOS Application

This section describes configuring your iOS application to receive WebRTC voice calls.

### Create a WSCCallPackageObserverDelegate

To be notified of an incoming call, create a **WSCCallPackageObserverDelegate** and attach it to your **WSCCallPackage**, as shown in Example 13–28.

***Example 13–28 Creating a CallPackageObserver Delegate***

```
Creating a CallPackageObserver Delegate
#pragma mark WSCCallPackageObserverDelegate
-(void)callArrived:(WSCCall *)call
        callConfig:(WSCCallConfig *)callConfig
        extHeaders:(NSDictionary *)extHeaders {

    NSLog(@"Registering a WSCCallObserverDelegate...");
    call.setObserverDelegate = WSCCallObserverDelegate;

    NSLog(@"Configuring the media streams...");
    RTCPeerConnectionFactory *)pcf = [call getPeerConnectionFactory];
    RTCMediaStream* localMediaStream = [pcf mediaStreamWithLabel:@"ARDAMS"];
    [localMediaStream addAudioTrack:[pcf audioTrackWithID:@"ARDAMSa0"]];

    if (answerTheCall) {
      NSLog(@"Answering the call...");
      [call accept:self.callConfig streams:localMediaStream];
    } else {
      NSLog(@"Declining the call...");
```

```
      [call decline:WSCStatusCodeBusyHere];
    }
  }
}
```

In Example 13–28, the **callArrived** event handler processes an incoming call request:

1. The method registers a **WSCCallObserverDelegate** for the incoming call. In this case, it uses the same **WSCCallObserverDelegate**, from the example in "Create a WSCCallObserverDelegate Protocol".
2. The method then configures the local media stream, in the same manner as "Configure the Local MediaStream for Audio".
3. The method determines whether to accept or reject the call based on the value of the **answerTheCall** boolean using the **accept** or **decline** methods of the **WSCCall** object.

> **Note:** The **answerTheCall** boolean will most likely be set by a user interface element in your application such as a button or link.

### Bind the CallPackage Observer to the CallPackage

With the **WSCCallPackageObserverDelegate** object created, you bind it to your **WSCCallPackage** object:

```
[callPackage setObserverDelegate:WSCCallPackageObserverDelegate;
```

# Adding WebRTC Video Support to your iOS Application

This section describes how you can add WebRTC video support to your iOS application. While the methods are almost completely identical to adding voice call support to an iOS application, additional preparation is required.

## Add the Audio and Video Capture Devices to Your Session

As with an audio call, you initialize the audio capture device as shown in Example 13–20. In addition, you initialize the video capture device and add it to your session, as shown below in Example 13–29.

***Example 13–29 Adding the Audio and Video Capture Devices to Your Session***

```
- (instancetype)initAudioVideo
{
  self = [super initAudioVideo];
  if (self) {
   self.captureSession = [[AVCaptureSession alloc] initAudioVideo];
   [self.captureSession setSessionPreset:AVCaptureSessionPresetLow];

   // Get the audio capture device and add to our session.
   self.audioCaptureDevice = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeAudio];
   NSError *error = nil;
   AVCaptureDeviceInput *audioInput = [AVCaptureDeviceInput
                                deviceInputWithDevice:self.audioCaptureDevice error:&error];
   if (audioInput) {
    [self.captureSession addInput:audioInput];
   }
   else {
    NSLog(@"Unable to find audio capture device : %@", error.description);
```

```
    }

    // Get the video capture devices and add to our session.
    for (AVCaptureDevice* videoCaptureDevice in [AVCaptureDevice
                                                 devicesWithMediaType:AVMediaTypeVideo]) {
     if (videoCaptureDevice.position == AVCaptureDevicePositionFront) {
       self.frontVideCaptureDevice = videoCaptureDevice;
       AVCaptureDeviceInput *videoInput = [AVCaptureDeviceInput
                                           deviceInputWithDevice:videoCaptureDevice error:&error];
       if (videoInput) {
         [self.captureSession addInput:videoInput];
       } else {
         NSLog(@"Unable to get front camera input : %@", error.description);
       }
     } else if (videoCaptureDevice.position == AVCaptureDevicePositionBack) {
       self.backVideCaptureDevice = videoCaptureDevice;
     }
    }
   }
   return self;
}
```

## Configure a View Controller to Display Incoming Video

You add a view object to a view controller to display the incoming video. In Example 13–30, when the **MyWebRTCApplicationViewController** view controller is created, its view property is nil, which triggers the **loadView** method

***Example 13–30 Creating a View to Display the Video Stream***

```
@implementation MyWebRTCApplicationViewController

- (void)loadView {

  // Create the view, videoView...
  CGRect frame = [UIScreen mainScreen].bounds;
  MyWebRTCApplicationView *videoView = [[MyWebRTCApplication alloc] initWithFrame:frame];

  // Set videoView as the main view of the view controller...
  self.view = videoView;
}

@end
```

Next you set the view controller as the rootViewController, which adds **videoView** as a subview of the window, and automatically resizes **videoView** to be the same size as the window, as shown in Example 13–31.

***Example 13–31 Setting the Root View Controller***

```
#import "MyWebRTCApplicationAppDelegate.h"
#import "MyWebRTCApplicationViewController.h"

@implementation MyWebRTCApplicationAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
                                                (NSDictionary *)launchOptions {
  self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

  MyWebRTCApplicationViewController *myvc = [[MyWebRTCApplicaitonViewController alloc] init];
  self.window.rootVewController = myvc;
```

```
  self.window.backgroundColor = [UIColor grayColor];
  return YES;
}
```

## Placing a WebRTC Video Call from Your iOS Application

To place a video call from your iOS application, complete the coding tasks contained in the following sections:

- Authenticating with WebRTC Session Controller
- Configure Interactive Connectivity Establishment (ICE) (if necessary)
- Configuring Support for Notifications

In addition, complete the coding tasks for an audio call contained in the following sections:

- Initialize the Call Object
- Configure Trickle ICE (if necessary)
- Create a WSCCallObserverDelegate Protocol
- Register the WSCCallObserverDelegate Protocol with the Call Object

> **Note:** Audio and video call work flows are identical with the exception of media directions, local media stream configuration, and the additional considerations described earlier in this section.

### Create a WSCCallConfig Object

You create a **WSCCallConfig** object as described in "Create a WSCCallConfig Object", in the audio call section. Set both arguments to **WSCMediaDirectionSendRecv**, as shown in Example 13–32.

***Example 13–32 Creating an Audio/Video WSCCallConfig Object***

```
WSCCallConfig *callConfig = [[WSCCallConfig alloc]
initWithAudioVideoDirection:WSCMediaDirectionSendRecv
video:WSCMediaDirectionSendRecv];
```

### Configure the Local WSCMediaStream for Audio and Video

With the **CallConfig** object created, you then configure the local video and audio **MediaStream** objects using the WebRTC **PeerConnectionFactory**. In Example 13–33, the **PeerConnectionFactory** is used to first configure a video stream using optional constraints and mandatory constraints (as defined in the **getMandatoryConstraints** method), and is then added to the **localMediaStream** using its **addVideoTrack** method. Two boolean arguments, hasAudio and hasVideo, enable the calling function to specify whether audio or video streams are supported in the current call. The audioTrack is added as well and the **localMediaStream** is returned to the calling function.

For information about the WebRTC **PeerConnectionFactory** and mandatory and optional constraints, see https://webrtc.org/native-code/native-apis/.

***Example 13–33 Configuring the Local MediaStream for Audio and Video***

```
-(RTCMediaStream *)getLocalMediaStreams:(RTCPeerConnectionFactory *)pcf
```

```
                                        enableAudio:(BOOL)hasAudio enableVideo:(BOOL)hasVideo {
  NSLog(@"Getting local media streams");
  if (!localMediaStream) {
      NSLog("PeerConnectionFactory: createLocalMediaStream() with pcf : %@", pcf);
      localMediaStream = [pcf mediaStreamWithLabel:@"ALICE"];
      NSLog(@"MediaStream1 = %@", localMediaStream);
  }

  if(hasVideo && (localMediaStream.videoTracks.count <= 0)){
    if (hasVideo) {
      RTCVideoCapturer* capturer = [RTCVideoCapturer
                          capturerWithDeviceName:[avManager.frontVideCaptureDevice localizedName]];
      RTCPair *dtlsSrtpKeyAgreement = [[RTCPair alloc] initWithKey:@"DtlsSrtpKeyAgreement"
                                        value:@"true"];
      NSArray * optionalConstraints = @[dtlsSrtpKeyAgreement];
      NSArray *mandatoryConstraints = [self getMandatoryConstraints];
      RTCMediaConstraints *videoConstraints = [[RTCMediaConstraints alloc]
                                               initWithMandatoryConstraints:mandatoryConstrainta
                                               optionalConstraints:optionalConstraints];
      RTCVideoSource *videoSource = [pcf videoSourceWithCapturer:capturer
                                                constraints:videoConstraints];
      RTCVideoTrack *videoTrack = [pcf videoTrackWithID:@"ALICEv0" source:videoSource];
      if (videoTrack) {
        [localMediaStream addVideoTrack:videoTrack];
      }
    }
  }

  if (localMediaStream.audioTracks.count <= 0 && hasAudio) {
    [localMediaStream addAudioTrack:[pcf audioTrackWithID:@"ALICEa0"]];
  }

  if (!hasVideo && localMediaStream.videoTracks.count > 0) {
    for (RTCVideoTrack *videoTrack in localMediaStream.videoTracks) {
      [localMediaStream removeVideoTrack:videoTrack];
    }
  }

  if (!hasAudio && localMediaStream.audioTracks.count > 0) {
    for (RTCAudioTrack *audioTrack in localMediaStream.audioTracks) {
      [localMediaStream removeAudioTrack:audioTrack];
    }
  }

  NSLog(@"MediaStream = %@", localMediaStream);
  return localMediaStream;
}

-(NSArray *)getMandatoryConstraints {

  RTCPair *localVideoMaxWidth = [[RTCPair alloc] initWithKey:@"maxWidth" value:@"640"];
  RTCPair *localVideoMinWidth = [[RTCPair alloc] initWithKey:@"minWidth" value:@"192"];
  RTCPair *localVideoMaxHeight = [[RTCPair alloc] initWithKey:@"maxHeight" value:@"480"];
  RTCPair *localVideoMinHeight = [[RTCPair alloc] initWithKey:@"minHeight" value:@"144"];
  RTCPair *localVideoMaxFrameRate = [[RTCPair alloc] initWithKey:@"maxFrameRate" value:@"30"];
  RTCPair *localVideoMinFrameRate = [[RTCPair alloc] initWithKey:@"minFrameRate" value:@"5"];
  RTCPair *localVideoGoogLeakyBucket = [[RTCPair alloc]
                                              initWithKey:@"googLeakyBucket" value:@"true"];

  return @[localVideoMaxHeight,
```

```
        localVideoMaxWidth,
        localVideoMinHeight,
        localVideoMinWidth,
        localVideoMinFrameRate,
        localVideoMaxFrameRate,
        localVideoGoogLeakyBucket];
}
```

### Bind the Video Track to the View Controller

As shown in Example 13–34, bind the video track to the view controller you created in "Configure a View Controller to Display Incoming Video".

***Example 13–34 Binding the Video Track to the View Controller***

```
if(localMediaStream.videoTracks.count >0) {
  [MyWebRTCApplicationViewController
                              localVideoConnected:localMediaStream.videoTracks[0]];
}
```

### Start the Video Call

Finally, start the audio/video call using the Call object's start method and passing it the **WSCCallConfig** object and the MediaStream stream array.

***Example 13–35 Starting the Video Call***

```
[call start:callConfig streams:streamArray];
```

### Terminate the Video Call

To terminate the video call, use the **WSCCall** object's end method:

***Example 13–36 Terminating the Video Call***

```
[self.call end];
```

## Receiving a WebRTC Video Call in Your iOS Application

Receiving a video call is identical to receiving an audio call as described here, "Receiving a WebRTC Voice Call in Your iOS Application". The only difference is the configuration of the **WSCMediaStream** object, as described in "Configure the Local WSCMediaStream for Audio and Video".

# Supporting SIP-based Messaging in Your iOS Application

You can design your iOS application to send and receive SIP-based messages using the messaging package in WebRTC Session Controller iOS SDK.

To support messaging, define the logic for the following in your application:

- Setup and management of the various activities associated with the states of the various objects, such as the session and the message transfer.
- Enabling users to send or receive messages
- Handling the incoming and outgoing message data
- Managing the required user interface elements to display the message content throughout the call session.

## About the Major Classes Used to Support SIP-based Messaging

The following major classes and protocols of the WebRTC Session Controller iOS SDK enable you to provide data channel support in your iOS application:

- **WSCMessagingPackage**

  This package handler enables messaging applications. You can send SIP-based messages to any logged-in user with an object of the **WSCMessagingPackage** class. This object also dispatches received messages to the registered delegate.

- **WSCMessagingDelegate**

  This class acts as a listener for incoming messages and their acknowledgements. It holds the following event handlers:

  – **onNewMessage**

    This event handler is called when your application receives a new SIP-based message.

  – **onSuccessResponse**

    This event handler is called when your application receives an accept/positive acknowledgment for a sent message.

  – **onErrorResponse**

    This event handler is called when your application receives a reject/negative acknowledgment for a sent message.

- **WSCMessagingMessage**

  This class is used to hold the payload for SIP-based messaging.

- **withPackage**

  This method belongs to the **WSCSessionBuilder** class. It is used to hold to build a session that supports a package, such as the messaging package.

For more on these and other WebRTC Session Controller iOS API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller iOS API Reference*.

## Setting up the SIP-based Messaging Support in Your iOS Application

Complete the following tasks to setup SIP-based messaging support in your iOS applications:

1. Enabling SIP-based Messaging
2. Sending SIP-based Messages
3. Handling Incoming SIP-based Messages

### Enabling SIP-based Messaging

To enable SIP-based messaging in your iOS application, create and assign an instance of a messaging package.

When you set up the **WSCSession** class, pass this messaging package in the **withPackage** parameter of the WSCSession builder API, as shown in Example 13–37.

***Example 13–37 Building a Session with a Messaging Package***

```
#import "WSCMessagingPackage.h"
 ...
```

```
WSCSession *wscSession = [[[[[[[[WSCSessionBuilder create: wsUrl]
                            withConnectionDelegate: self]
                                      withUserName: userName]
                              withObserverDelegate: self]
                                       withPackage: [[WSCCallPackage alloc] init]]
                                       withPackage: [[WSCMessagingPackage alloc] init]]
                                  withHttpContext: httpContext]
                                                  build];
```

Ensure that your application implements the **onSuccess** and **OnFailure** event handlers in the **WSCSessionConnectionDelegate** object. WebRTC Session Controller iOS SDK sends asynchronous messages to these event handlers, based on its success or failure to build the session.

### Sending SIP-based Messages

To send a SIP-based message, invoke the **send** method of the **WSCMessagePackage** object. The signature of the **send** method is:

```
(NSString *)send:(NSString *)textMessage target:(NSString *)target extHeaders:(NSDictionary
*)extHeaders
```

In Example 13–38, *destination* is "bob@example.com" and the *text* is "Hi There." Bob sees the message from the sending party.

***Example 13–38 Sending a SIP-based Message***

```
...
WSCMessagingPackage *msgPackage = (WSCMessagingPackage *)[self.wscSession getPackage:PACKAGE_TYPE_
MESSAGING];

NSString *peer = @"alice@example.com";
[msgPackage send:@"Hi There" target:peer];
...
```

### Handling Incoming SIP-based Messages

Set up your application to handle incoming messages and acknowledgements. Register a **WSCMessagingDelegate** to be notified when a new message is received. Set the **ObserverDelegate** property of the **WSCMessagePackage** object, as shown in Example 13–39:

***Example 13–39 Registering the Observer for the Message Package***

```
...
WSCSession session;
// Register an observer for listening to incoming messaging events.
WSCMessagingPackage *msgPackage = (WSCMessagingPackage *)[self.wscSession getPackage:PACKAGE_TYPE_
MESSAGING];
[msgPackage setObserverDelegate:self];
...
```

When a new message comes in, the **onNewMessage** event handler of the **WSCMessagingDelegate** object is called. In the callback function you implement for the **onNewMessage** event handler, accept or reject the message received using the appropriate APIs.

Set up the logic to handle the acknowledgements appropriately:

- The **accept** method of the **WSCMessagePackage** object. When the receiver of the message accepts the message, the **onSuccessResponse** event is triggered on the sender's side (that originated the message).
- The **reject** method of the **WSCMessagePackage** object. When the receiver of the message rejects the message, the **onErrorResponse** event is triggered on the sender's side (that originated the message).

***Example 13–40 Example of a Delegate Set up for a Message Package***

```
// Class that observes for incoming messages from Messaging.
#pragma mark WSCMessagingDelegate

- (void) onNewMessage:(WSCMessagingMessage *)message {
  NSLog(@"Got Messaging: onNewMessage with content:%@", message.content);

  // Show message content in some UITextView

  // Accept message as received
  [self.wscMsging accept:message];
}

-(void) onSuccessResponse:(WSCMessagingMessage *)message{
  NSLog(@"Messaging: onSuccessResponse");
}

-(void) onErrorResponse:(WSCMessagingMessage *)message cause:(WSCCause *)cause reason:(NSString
*)reason {
  NSLog(@"Messaging: onErrorResponse");
}
```

# Adding WebRTC Data Channel Support to Your iOS Application

This section describes how you can add WebRTC data channel support to the calls you enable in your iOS application. For information about adding voice call support to an iOS application, see "Adding WebRTC Voice Support to your iOS Application".

To support calls with data channels, define the logic for the following in your application:

- Setup and management of the various activities associated with the states of the various objects such as the session, the data transfer.
- Enabling users to make or receive calls with data channels set up with or without the audio and video streams
- Handling the incoming and outgoing data
- Managing the required user interface elements to display the data content throughout the call session.

## About the Major Classes and Protocols Used to Support Data Channels

The following major classes and protocols enable you to provide data channel support in your iOS application:

- **WSCCall**

  This object represents a call with any combination of audio, video, and data channel capabilities. It creates a data channel and initializes the **WSCDataTransfer**

object for the data channel when the call starts or when accepting the Call if the Call has capability of data channel.

- **WSCCallConfig**

  The **WSCCallConfig** object represents a call configuration. It describes the audio, video, or data channel capabilities of a call.

- **WSCDataChannelOption**

  The **WSCDataChannelOption** object describes the configuration items in the data channel of a call such as whether ordered delivery is required, the stream id, maximum number of retransmissions and so on.

- **WSCDataChannelConfig**

  The **WSCDataChannelConfig** object describes the data channel of a call, including its label and **WSCDataChannelOption**.

- **WSCDataTransfer**

  The **WSCDataTransfer** object manages the data channel. If the **WSCCallConfig** object includes the data channel, the **WSCCall** object creates an instance of the **WSCDataTransfer** object.

- **WSCDataSender**

  The **WSCDataSender** object exposes the capability of a WSCDataTransfer to send raw data over a data channel. The instance is created by **WSCDataTransfer**.

- **WSCDataReceiver**

  The **WSCDataReceiver** object exposes the capability of a WSCDataTransfer to receive raw data over the established data channel. The instance is created by **WSCDataTransfer**.

- **onDataTransfer**

  The **onDataTransfer** method associated with WSCCallObserverDelegate indicates that a **WSCDataTransfer** is created.

- **WSCDataTransferObserverDelegate**

  The **WSCDataTransferObserverDelegate** acts as an observer protocol for **WSCDataTransfer**.

  Your application can implement this protocol to be informed of changes in **WSCDataTransfer**.

- **WSCDataReceiverObserverDelegate**

  The **WSCDataReceiverObserverDelegate** acts as an observer protocol for **WSCDataReceiver**, the receiver of the data transfer.

For more information on these and other WebRTC Session Controller iOS API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller iOS API Reference*.

### About the Sample Code Excerpts in This Section

The sample code excerpts shown in this section are taken from a sample iOS application which supports data-channels. The sample interface extends the **WSCDataTransferObserverDelegate** and **WSCDataReceiverObserverDelegate** Observers in addition to existing **WSCCallObserverDelegate** observers. Each excerpt is simple and attempts to illustrate the functionality under that discussion only.

## About the Data Transfers and Data Channels

If the data channel is enabled, then, for both incoming and outgoing calls, a **WSCDataTransfer** object is created and passed back to your iOS application in the callback for the **onDataTransfer** method of the **WSCCallObserverDelegate** protocol.

### Setting Up DataTransferObserverDelegate Protocol to Handle Data Transfers

When the **onDataTransfer** method of the **WSCCallObserverDelegate** protocol object is called, set the data transfer observer delegate to be informed of changes in the data transfer.

```
-(void)onDataTransfer:(WSCDataTransfer *)dataTransfer {

  // register delegate to listen to the state change
  dataTransfer.observerDelegate = self;

  // keep this data transfer object for later use
  _dataTransfer = dataTransfer;
}
```

In order for your application to respond to the various states of the data channel, implement the following methods of the **WSCDataTransferObserverDelegate** protocol:

- **onOpen**

  ```
  - (void)onOpen:(WSCDataTransfer *)dataTransfer
  ```

  This method is called when the data channel of the **WSCDataTransfer** object is open. The state of the **WSCDataTransfer** object is denoted by the enum value **WSCDataTransferOpen**.

  Your application can send and receive messages, as appropriate.

- **onClose**

  The method is called when the data channel of the **WSCDataTransfer** object is closed. The state of the **WSCDataTransfer** object is denoted by the enum value **WSCDataTransferClosed**.

- **onError**

  ```
  - (void)onError:(WSCDataTransfer *)dataTransfer
  ```

  This method is called when the data channel of the **WSCDataTransfer** object encounters an error. The state of the **WSCDataTransfer** object is denoted by the enum value **WSCDataTransferError**.

## Initialize the CallPackage Object

When you created your Session, you registered a new **WSCCallPackage** object using the Session object's **withPackage** method. You now instantiate that **WSCCallPackage**, as shown in Example 13–41.

***Example 13–41 Initializing the CallPackage***

```
WSCCallPackage *callPackage = (WSCCallPackage*)[wscSession getPackage:PACKAGE_
TYPE_CALL];
```

Use the default PACKAGE_TYPE call type unless you have defined a custom call type. For more information, see "Initialize the CallPackage Object" under the audio call section of this chapter.

## Sending Data from Your iOS Application

To send data from your iOS application, complete the coding tasks contained in the following sections.

- Authenticating with WebRTC Session Controller
- Configure Interactive Connectivity Establishment (ICE) (if necessary)
- Creating a WebRTC Session Controller Session

Complete the coding tasks for an audio call contained in the following sections:

- Initialize the Call Object
- Configure Trickle ICE (if necessary)
- Create a WSCCallObserverDelegate Protocol
- Register the WSCCallObserverDelegate Protocol with the Call Object

### Configure the Data Channel for the Data Transfers

Configure the data channel with **WSCDataChannelConfig** before you set up the **WSCCallConfig** object. The WebRTC Session Controller client iOS SDK supports multiple data channels in a call.

If you create one **WSCDataChannelConfig** object for a call, then one instance of the **WSCDataTransfer** object is created to support the call. Assign a label for the data channel configuration object, to allow your application to access the corresponding **WSDDataTransfer** with this label.

Example 13–42 shows one data channel that is assigned the label, *Sample*.

***Example 13–42 Configuring a Single Data Channel for the Call***

```
...
// Set up WSCDataChannelOption
  WSCDataChannelOption *option = [[WSCDataChannelOption alloc]init];
  // Set various options on WSCDataChannelOption
  //For example, option.maxRetransmits = 5;

// create WSCDataChannelConfig
WSCDataChannelConfig *dcConfig = [[WSCDataChannelConfig alloc] initWithLabel:@"sample"
withOption:option];
  NSArray *dcConfigs = [[NSArray alloc] initWithObjects:dcConfig, nil]
...
```

You can create multiple **WSCDataChannelConfig** objects. When you do so, the WebRTC Session Controller iOS SDK creates the required number of **WSCDataTransfer** objects to support your requirement.

Example 13–43 shows the code sample defining two data channels, each with its own label and placing them in *myDataChannelConfig*, its **DataChannelConfig** object.

***Example 13–43 Configuring Multiple Data Channels for the Call***

```
  // This code sample sets up 2 different data channels both using default values.
  WSCDataChannelOption *firstDcOption = [[WSCDataChannelOption alloc] init];
  WSCDataChannelConfig *firstDcConfig = [[WSCDataChannelConfig alloc]
initWithLabel:@"firstDataChannel" withOption:firstDcOption];

  WSCDataChannelOption *secondDcOption = [[WSCDataChannelOption alloc] init];
```

```
  WSCDataChannelConfig *secondDcConfig = [[WSCDataChannelConfig alloc]
initWithLabel:@"secondDataChannel" withOption:secondDcOption];

  // Create an array containing both data channel definitions.
  NSArray *myDataChannelConfigs = [[NSArray alloc] initWithObjects:firstDcConfig, secondDcConfig,
nil];

  // Finally initialise the call configuration by including both data channels.
  WSCCallConfig* myCallConfig = [[WSCCallConfig alloc] initWithAudioDirection:audioMediaDirection
                                                withVideoDirection:videoMediaDirection
                                                   withDataChannel:myDataChannelConfigs];
```

### Handling the Data Channel States

Implement the **onOpen**, **onClose**, and **onError** methods of the **DataTransferObserverDelegate** protocol so you can respond to changes in the states of the data channel.

For a description of the methods, see "Setting Up DataTransferObserverDelegate Protocol to Handle Data Transfers".

### Create a WSCCallConfig Object with Data Channel Option

Having defined the data channel setup for the call, you can now create a **WSCCallConfig** object to determine the type of call you wish to make.

In Example 13–44, the sample application uses the IBAction return type to use the method as the action of a UI control.

***Example 13–44 Configuring WSCCallConfig with a DataChannel***

```
...
    NSArray *dataChannelConfigs = nil;
    WSCDataChannelOption *dcOption = [[WSCDataChannelOption alloc] init];
    WSCDataChannelConfig *dcConfig = [[WSCDataChannelConfig alloc] initWithLabel:@"sample"
withOption:dcOption];
    dataChannelConfigs = [[NSArray alloc] initWithObjects:dcConfig, nil];

    WSCCallConfig* newConfig = [[WSCCallConfig alloc]
initWithAudioDirection:self.callConfig.audioConfig

withVideoDirection:self.callConfig.videoConfig
                                                        withDataChannel:dataChannelConfigs];
```

### Configure the Local MediaStream for Audio and Video

If the calls in your application support an audio and/or video stream also, configure the local video and audio MediaStream objects using the WebRTC **PeerConnectionFactory**.

For more information about configuring the audio and video streams, see:

- Configure the Local MediaStream for Audio
- Configure the Local WSCMediaStream for Audio and Video

### Set Up Your Application to Receive Incoming Data

Set up a function to process the incoming data as shown in Example 13–45.

The **onMessage** method of **DataReceiverObserverDelegate** protocol is invoked when a message is received.

> **Tip:** Process the message inside the function handling **onMessage**, or copy it for later use.

***Example 13–45 Receiving Data from WSCDataReceiver***

```
-(void)onMessage:(RTCDataBuffer *)buffer
{
/**
 * Invoked when there is message received.
 *
 * @param data RTCDataBuffer which is received by this data channel.
 */
-(void) onMessage:(RTCDataBuffer *)data {
  DLog(@"DataTransfer %@ receive message %@", self.dataTransfer.label, data);
  if (data.isBinary) {
    ALog(@"DataTransfer %@ receive binary message", self.dataTransfer.label);
  } else {
    NSString *received = [[NSString alloc] initWithData:data.data encoding:NSUTF8StringEncoding];
    DLog(@"DataTransfer %@ receive text message: %@", self.dataTransfer.label, received);
    DLog(@"content of the data chage view: %@", self.dataChatView.text);
    [self appendIncomingMessageToHistory:received];
  }

}
```

The **WSCDataReceiver** object created by the **WSCDataTransfer** object, can receive raw data over the underlying data channel.

### Start the Data Channel Call

Start the call using the **WSCCall** object's **start** method. When you call this method, provide the instance of the **WSCCallConfig** object that contains the call's capabilities and the **streams** object with the local media streams to be attached to call.

Invoke the appropriate method:

- Without extension headers, as shown in Example 13–46:

```
-(void)start:(WSCCallConfig *)config
     headers:(NSDictionary *)headers
     streams:(NSArray *)localStreams;
```

***Example 13–46 Start a Call***

```
[call start:callConfig streams:streamArray];
```

- With extension headers:

```
-(void)start:(WSCCallConfig *)config
     headers:(NSDictionary *)headers
     streams:(NSArray *)localStreams;
```

  See "About Extension Headers and JSON Messages".

### Send the Data Content

Send data using the **send** method of the **sender** in the **WSCDataTransfer** object.

Use the label for the data channel to retrieve the **WSCDataTransfer** object from the **WSCCall** object. Set up the **WSCDataSender** object. Verify that the state of the **WSCDataTransferState** object is WSCDataTransferOpen. Invoke the **send** method of this **WSCDataSender** object to send data. Example 13–47 shows a text message sent by the sample code.

> **Note:** Invoke the **send:** method, when the data channel of the data transfer object is in an open state.

***Example 13–47 Sending Text Strings as Data***

```
-(void)sendData:(NSString *)message {

  RTCDataBuffer *buffer = [[RTCDataBuffer alloc] initWithData:[message
dataUsingEncoding:NSUTF8StringEncoding] isBinary:NO];

// Send Data after verifying that the data channel is in WSCDataTransferOpen state.
...
  // send the message through WSCDataSender
  [dataTransfer.sender send:buffer];
}
```

### Terminate the Data Channel in the Call

To terminate the audio call, use the appropriate method:

- `- (void)end`
- `- (void)end:(NSDictionary *)headers`

  See "About Extension Headers and JSON Messages".

## Receiving Data Content in Your iOS Application

This section describes the steps specific to configuring your iOS application to receive WebRTC data transfers.

### Implement WSCCallPackageObserverDelegate Protocol to Verify Data Channel Capability

Use the **callArrived:callConfig:extHeaders:** method of **WSCCallPackageObserverDelegate** to initialize the **WSCCallObserverDelegate** protocol that acts as the **ObserverDelegate** for the **Call**.

As Example 13–48 shows, you can also verify from the **CallConfig** object that a data channel is enabled for this call.

***Example 13–48 Initializing the ObserverDelegate for a Data Channel Call***

```
-(void)callArrived:(WSCCall *)call callConfig:(WSCCallConfig *)callConfig
extHeaders:(NSDictionary *)extHeaders {
  if (callConfig.dataChannelConfigs) {
    // This is data channel enabled call
  }
  call.observerDelegate = self;
}
```

### Handling the Data Channel States

Implement the **onOpen**, **onClose**, and **onError** methods of the **DataTransferObserverDelegate** protocol so you can respond to changes in the states of the data channel.

For a description of the methods, see "Setting Up DataTransferObserverDelegate Protocol to Handle Data Transfers".

### Implement the DataReceiverObserverDelegate Protocol to Listen for Messages

When the data channel is in an open state, initialize the **ObserverDelegate** of **receiver**, the receiving object of the **DataTransfer**, as shown in Example 13–49. This object serves to listen for incoming messages.

***Example 13–49 The ObserverDelegate for the Receiver of the Data Transfer***

```
-(void)onOpen:(WSCDataTransfer *)dataTransfer {

  // after the callback completes, this data transfer is ready to send/receive
message.

  // register delegate on receiver to listen to the incoming message
  dataTransfer.receiver.observerDelegate = self;

}
```

### Accept the Call

To accept an incoming call, invoke the appropriate method:

- **accept:streams:** method, invoked as

  ```
  - (void)accept:(WSCCallConfig *)config streams:(RTCMediaStream
  *)localStreams
  ```

- **accept:extHeaders:streams:** method when you support extension headers. This is invoked as

  ```
  - (void)accept:(WSCCallConfig *)config streams:(RTCMediaStream
  *)localStreams
  ```

  See "About Extension Headers and JSON Messages".

### Receiving Data

Obtain the incoming message from the data channel by setting up the logic for the **onMessage** method of **WSCDataReceiverObserverDelegate**. This method is invoked when there is an incoming message, as shown in Example 13–50.

***Example 13–50 Receiving Data***

```
-(void)onMessage:(RTCDataBuffer *)buffer
{
  if (data.isBinary) {
    // raw data
  } else {
   NSString *received = [[NSString alloc] initWithData:data.data encoding:NSUTF8StringEncoding];
  }
}
```

# Upgrading and Downgrading Calls

This section describes how you can handle upgrading an audio call to an audio video call and downgrading a video call to an audio-only call in your iOS application.

## Handle Upgrade and Downgrade Requests from Your Application

To upgrade from a voice call to a video call as a request from your application, you can bind a user interface element such as a button class containing the **WSCCall** update logic using the **forControlEvents** action:

```
[requestUpgradeButton addTarget:self action:@selector(videoUpgrade)
                                forControlEvents:UIControlEventTouchUpInside];
[requestDowngradeButton addTarget:self action:@selector(videoDowngrade)
                                forControlEvents:UIControlEventTouchUpInside];
```

You handle the upgrade or downgrade workflow in the **videoUpgrade** and **videoDowngrade** event handlers for each button instance, as shown in Example 13–51.

***Example 13–51 Sending Upgrade/Downgrade Requests from Your Application***

```
- (void) videoUpgrade: {
  // Set the criteria for the current call...
  self.hasVideo = NO;
  self.hasAudio = YES;

  // Fetch local streams using the the getLocalMediaStreams function from Example 13-33
  [self getLocalMediaStreams:[self.call getPeerConnectionFactory] enableVideo:hasVideo
                                                                  enableAudio:hasAudio];

  // Bind the video stream to the view controller as in Example 13-34
  if(localMediaStream.videoTracks.count >0) {
    [MyWebRTCApplicationViewController
                            localVideoConnected:localMediaStream.videoTracks[0]];
  }

  // Audio -> Video upgrade
  WSCCallConfig *newConfig = [[WSCCallConfig alloc]
                              initWithAudioVideoDirection:WSCMediaDirectionSendRecv
                              video:WSCMediaDirectionSendRecv]];
  [call update:newConfig headers:nil streams:@[localMediaStream]];
}

- (void) videoDowngrade: {
  // Set the criteria for the current call...
  self.hasVideo = YES;
  self.hasAudio = YES;

  // Fetch local streams using the the getLocalMediaStreams function from Example 13-33
  [self getLocalMediaStreams:[self.call getPeerConnectionFactory] enableVideo:hasVideo
                                                                  enableAudio:hasAudio];

  // Bind the video stream to the view controller as in Example 13-34
  if(localMediaStream.videoTracks.count >0) {
    [MyWebRTCApplicationViewController
                            localVideoConnected:localMediaStream.videoTracks[0]];
  }

  // Video -> Audio downgrade
```

```
  WSCCallConfig *newConfig = [[WSCCallConfig alloc]
                              initWithAudioVideoDirection:WSCMediaDirectionSendRecv
                              video:WSCMediaDirectionNone]];
  [call update:newConfig headers:nil streams:@[localMediaStream]];
}
```

## Handle Incoming Upgrade Requests

You configure the **callUpdated** method of your **WSCCallObserverDelegate** class to handle incoming upgrade requests based on the **WSCCallUpdateEventReceived** state change. See Example 13–52.

**Note:** The **declineUpgrade** boolean must be set by some other part of your application's user interface.

***Example 13–52 Handling an Incoming Upgrade Request***

```
- (void)callUpdated:(WSCCallUpdateEvent)event
         callConfig:(WSCCallConfig *)callConfig
              cause:(WSCCause *)cause
{
  NSLog("callUpdate request with config: %@", callConfig.description);
  switch(event){
    case WSCCallUpdateEventSent:
      break;
    case WSCCallUpdateEventReceived:
      if(declineUpgrade) {
         NSLog(@"Declining upgrade.");
         [self.call decline:WSCStatusCodeDeclined];
      } else {
          NSLog(@"Accepting upgrade.");
          NSLog(@"Call config: %@", updateConfig.description);
          BOOL hasAudio;
          BOOL hasVideo;
          if (updateConfig.audioConfig == WSCMediaDirectionNone) {
            hasAudio = NO;
          }
          if (updateConfig.videoConfig == WSCMediaDirectionNone) {
            hasVideo = NO;
          }
          self.callConfig = updateConfig;
          [self getLocalMediaStreams:[self.call getPeerConnectionFactory] enableAudio:hasAudio
                                                                          enableVideo:hasVideo];
          [self.call accept:updateConfig streams:localMediaStream];
          [callViewController updateView:self.callConfig];
      }
    case WSCCallUpdateEventAccepted:
      break;
    case WSCCallUpdateEventRejected:
      break;
    default:
      break;
  }
```

# Handling Session Rehydration When the User Moves to Another Device

When your customer is using your application on one device (a cellphone), the customer may move to another device (a laptop softphone that uses the same account

and is authenticated by WebRTC Session Controller). A Session (along with its subsessions) currently active in your application on one device belonging to your customer becomes active on your application on another device belonging to the same customer.

For example, your customer Alice, accesses a web browser from a cellphone to talk about a purchase selection with Bob, a customer support representative active in that browser session. While Alice is on the call, she switches over to her laptop to look at the purchase selection in greater detail.

You can use the WebRTC Session Controller to configure applications that support such handovers of session information between devices successfully. Your application then manages the rehydration of the session and all its data on the target device (in this example, the tablet laptop) such that Alice's call to Bob continues in an uninterrupted fashion.

This section described how your application can work to present the customer with the session state recreated on another device.

> **Note:** In a device-handover scenario, WebRTC Session Controller manages the data associated with the subsessions of your application session. It keeps their states intact through the handovers that may occur during the life of an application session.
>
> The focus of the handover logic in your application is the Session within which a call, a message, or a video session lives.

## About the Supported Operating Systems

You can design your applications using WebRTC Session Controller such that you support handover to your applications programmed for iOS, Web and Android systems.

> **Note:** For such a handover to be successful, your application must be active on the various devices belonging to a user, the associated user name and account be authenticated by WebRTC Session Controller, and the applications supported on the various operating systems.

This chapter deals with setting up your iOS application to support handovers using the WebRTC Session Controller iOS SDK. For information about supporting handovers to:

- Web applications, see "About Using the WebRTC Session Controller JavaScript API".
- Android applications, see "Developing WebRTC-Enabled Android Applications".

## Configuring WebRTC Session Controller to Support Transfer of Session Data

In a device handover, the same WebSocket sessionID is used to transfer an application session state that is active in the current client device (for example Alice's cellphone) and present that state on the subsequent device (Alice's laptop).

When one client uses another client's websocket sessionID to connect with WebRTC Session Controller, the WSC server checks the value in the system property, **allowSessionTransfer**. The default value of **allowSessionTransfer** is **false**. This value

causes WebRTC Session Controller to consider the request as a hacking attack and reject the request.

In order to allow the same user or tenant to connect with the WebRTC Session Controller server using the same WebSocket session Id, set the startup command option **allowSessionTransfer** to **true** in the WebRTC Session Controller Administration Console. For more information on how to set the startup option **allowSessionTransfer** to **true** in WebRTC Session Controller, see "Supporting Session Rehydration for Device Handover Scenarios" in *Oracle Communication WebRTC Session Controller System Administrator's Guide*.

## About the WebSocket Disconnection

When the device handover occurs, the WebSocket connection immediately closes.

The WebRTC Session Controller signaling engine keeps the session alive for a time period specified as **WebSocket Disconnect Time Limit** in the WebRTC Session Controller Administration Console.

---

**Note:** If the target device fails to pick up the session within the **WebSocket Disconnect Time Limit** period, the device handover fails.

---

## About the Normalized Session Data User to Support Handovers

A user may move from an application where your application uses one type of SDK to a device where your application uses a different SDK. The supported Client SDKs are:

- Web
- Android
- iOS

WebRTC Session Controller supports a normalized uniform session data format to transfer the session state information between these systems. The session state information is sent as a binary large object (BLOB).

## About the Handover Scenario on the Original Device

When the original device (for example, Alice's cellphone *Device A-1*) triggers a handover, the following events occur:

1. **On the Original Device:**
   a. Your application on *DeviceA-1* suspends the active session on the WebRTC Session Controller server. The WebRTC Session Controller iOS SDK API returns the session data to your application. The WebSocket connection closes.

      See "Suspending the Session on the Original Device".

   b. Your application transfers the session data (**stateInfo**) to be received and processed by the application on the subsequent device *Device A-2*.

      See "Sending the Session Data to the Application Service".

   The WebRTC Session Controller Signaling engine keeps this session alive for a time period specified as **WebSocket Disconnect Time Limit** in the WebRTC Session Controller Administration Console.

2. **On the device receiving the handover**

The subsequent device that receives the handover is Alice's laptop *DeviceA-2* in this discussion. The following events occur:

**a.** Your application on the subsequent device retrieves the **stateInfo** string from the Restful server. See "Requesting for the Session Data from the Application Service".

**b.** Your application uses the session state information to recreate the session. See "Recreating the Application Session with the StateInfo Object".

**c.** Your application rehydrates the call with WebRTC Session Controller iOS SDK.

See "Rehydrating a WebRTC Call After a Device Handover".

**d.** The active call resumes on the subsequent device.

## About the WebRTC Session Controller iOS APIs for Device Handover

The WebRTC Session Control iOS APIs that enable your applications to handle notifications related to session Rehydration in another device are:

- **withStateInfo** parameter of **WSCSessionBuilder**

  When you provide the **withStateInfo** parameter, the WebRTC Session Controller iOS SDK rehydrates the session using the StateInfo object. If there is a call subsession in the StateInfo object, WebRTC Session Controller iOS SDK rehydrates the call.

- **suspend**

  The **suspend** method of the **WSCCallPackage** object suspends the active session. This method returns a JSON string object containing the session data to use in session rehydration.

- **WSCSessionStateSuspended**

  The **WSCSessionStateSuspended** constant represents the state of **WSCSession** after the session is handed over.

- **callResurrected**

  The **callResurrected** method of the **WSCCallPackageObserverDelegate** is called when the call is resurrected after a session is rehydrated. Its parameters are **WSCCall** (the call that was resurrected) and the **WSCCallConfig** of the resurrected call.

For more on these and other WebRTC Session Controller iOS API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller iOS API Reference*.

## Completing the Tasks to Support Session Rehydration in Another Supported Device

This section described the tasks to complete in your application to hand over a session to another device and to receive a session handed over from another device. Complete the following tasks to support session transfers and rehydration with the transferred session state information:

- Suspending the Session on the Original Device
- Sending the Session Data to the Application Service
- Requesting for the Session Data from the Application Service
- Recreating the Application Session with the StateInfo Object

- Rehydrating a WebRTC Call After a Device Handover

### Suspending the Session on the Original Device

In order to implement a handover, your application on the original device *DeviceA-1* suspends the active session on the WebRTC Session Controller server. One scenario would be to set up a *handover* function and suspend the session within its logic.

> **Note:** The logic surrounding the detection of the actual device handover is beyond the scope of this document.

The WebRTC Session Controller iOS API method to suspend a session is **suspend**. The WebRTC Session Controller iOS SDK API returns the session data to your application in JSON string format. The WebSocket connection closes.

In your application logic that handles the user interface related to the handover, call the WebRTC Session Controller iOS API method, **WSCSession.suspend()**.

***Example 13–53 Suspending a Session***

```
...
// Handover detected
...
/**
 * Shut down the currently open socket and return StateInfo Json String
 *
 * @return NSString
 */
-(NSString *)suspend;
...
```

### Sending the Session Data to the Application Service

Your application on the original device (for example, Alice's cellphone called *Device A-1*) sends the session state information in a handover request to the Application Service (your application, a Web application, or a RESTful service).

You can configure how your application performs this task in the way that suits your environment. For example, your application can push the **stateInfo** to the other device or allow the other device to pull the **stateInfo**.

When the **suspend** method completes, your application has the session data with which to rehydrate the session. The session data is in JSON format. In your implementation of the logic for the **WSCSessionHandedOver** state of **WSCSession**, set up the information to transfer to the Application service.

Include this JSON string object and any other relevant information in the data you send with the handover request to the application service.

### Requesting for the Session Data from the Application Service

In the application logic that handles this trigger, send a request to the application service asking for the session state information. The application service returns the **stateInfo**, the session data for rehydration in a JSON String format.

### Recreating the Application Session with the StateInfo Object

Your application on the subsequent device sends the session state information to the WebRTC Session Controller iOS SDK.

In your application logic that handles session rehydration following a device handover, set up a session object with the session state data you received from the application service.

Set up the Builder for the WSCSession by providing the session state data in the **withStateInfo** method of the **WSCSessionBuilder** object. This method returns a **WSCSessionBuilder** object with a stateInfo configuration:

```
- (WSCSessionBuilder *)withStateInfo:(NSString *)stateInfo
```

In the following sample code excerpt, the application provides the session data in *dhSessionStateInfo* and builds the session.

***Example 13–54 Building the Session with StateInfo***

```
...
self.wscSession = [[[[[[[[[[[[[[WSCSessionBuilder create:url]
...
                           withStateInfo:[NSString dhSessionStateInfo]
...
                            build];
```

For information about **WSCSessionBuilder**, see *Oracle Communications WebRTC Session Controller iOS API Reference*.

Ensure that you implement the logic for **onSuccess** and **OnFailure** event handlers in the **WSCSessionConnectionDelegate** object. WebRTC Session Controller iOS SDK sends asynchronous messages to these event handlers, based on its success or failure to build the session.

A subsession, such as a call session that was part of the application session on the original device, is also suspended when that device suspends the application session. In such a scenario, WebRTC Session Controller iOS SDK creates the subsession (for example, the WSCCall object) and passes the object to your application's **WSCCallObserver**.

### Rehydrating a WebRTC Call After a Device Handover

A call session that was part of the application session on the original device, is also suspended when that device suspends the application session. In such a scenario, WebRTC Session Controller iOS SDK creates the subsession objects for your application. For example, if there is a **WSCCall** object, it passes the call configuration object to your application's **WSCCallObserverDelegate**.

---

**Note:** After the new device connects to the session, WebRTC Session Controller does not send out a new incoming call request to connect a call that is part of the handover. To recreate the call connection, the WebRTC Session Controller iOS SDK uses the information in the **stateInfo** object.

If there is no "re-invite" flow, the ongoing call is not established, because the client IP, port, or codec has changed.

In your application, complete the candidates re-negotiation with the peer side, as dictated by the iOS system.

---

If the rehydrated session contains a call session, WebRTC Session Controller iOS SDK will attempt to create the **WSCCall** object from the stored **stateInfo** object.

On successful creation of the **WSCCall** object, WebRTC Session Controller iOS SDK triggers the **callResurrected** delegate of the **WSCCallPackageObserverDelegate** protocol object and provides the **WSCCall** object and the **WSCCallConfig** objects for the resurrected call.

Implement the following logic in your iOS application:

- In your application logic that handles the **callResurrected** event of the **WSCCallPackageObserverDelegate** protocol object, rehydrate the call using this **WSCCall** object and its **WSCCallConfig** object.
- To be informed of the updated call, implement the **CallUpdated** method in the **WSCCallObserverDelegate** protocol object.

Following a device handover, the general workflow for rehydrating a call with video streams or data channels is identical to the workflow for rehydrating a call with audio stream. Any specificity lies in how your application logic handles the **WSCCallConfig** object to maintain the video streams and data transfers associated with the call.

# Extending Your Applications with WebRTC Session Controller iOS SDK

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

---

**Note:** Before you proceed, review the discussion about how the WebRTC Session Control JavaScript APIs assist in extending WebRTC applications. See "Extending Your Applications Using WebRTC Session Controller JavaScript API" for more information.

---

You can extend your iOS applications by doing the following:

- Extending an existing package. See "Extending an Existing Package".
- Adding a new package. See "Adding a New Package".
- Extending an existing package. See "About Extension Headers and JSON Messages".

## Extending an Existing Package

To extend existing packages in your iOS application, extend the associated iOS classes. The supported classes are:

- **WSCSession**:

  Your application can create only one session with the WebRTC Session Controller server for each user. However, you can do the following:

  - Extend the session when you build it. See "Building an Extended Session".
  - Create a WSC message frame. See "Creating a WSC Frame".
  - Send a WSC message frame. See "Sending a WSC message Frame to the Session".
  - Add a subsession. See "Adding a SubSession to the Session".

- **WSCMessagingPackage**

To extend the SIP-based messaging, your application can create a **WSCMessagingPackage** class and register it when you create a session. See "Supporting SIP-based Messaging in Your iOS Application".

- **WSCFrame**

  **WSCFrame** is the master data transfer object class for all JSON messages.

  - **initWithJson**

    Constructor with a given JSON string. See "Creating a WSC Frame".

- **WSCFrameFactory**

  Helper class to create JSON WSCFrame instances.

  - **createFrame**

    Creates a JSON frame with arguments and a correlation id. See "Creating a WSC Frame".

- **WSCCall**

  - Accept a call with specific configuration.

    ```
    - (void)accept:(WSCCallConfig *)config extHeaders:(NSDictionary
    *)extHeaders streams:(RTCMediaStream *)localStreams
    ```

  - Decline a call with specific configuration.

    ```
    - (void)decline:(NSInteger)code headers:(NSDictionary *)headers
    ```

    The available reason codes are **486** (busy here), **603** (decline), and **600** (busy everywhere).

  - Start a call with specific configuration.

    ```
    - (void)start:(WSCCallConfig *)config headers:(NSDictionary *)headers
    streams:(NSArray *)localStreams
    ```

  - Update the audio and video capabilities of a call.

    ```
    - (void)update:(WSCCallConfig *)config headers:(NSDictionary *)headers
    streams:(NSArray *)localStreams
    ```

  - End a call with specific configuration.

    ```
    - (void)end:(NSDictionary *)headers
    ```

For more information on these and other WebRTC Session Controller iOS API classes, see **AllClasses** at *Oracle Communications WebRTC Session Controller iOS API Reference*.

### Building an Extended Session

Use extension headers to extend the session when you build it with the **WSCSessionBuilder**. The following entry creates a new **WSCSessionBuilder** with extension headers which are sent as part of the session connection:

```
- (WSCSessionBuilder *)withExtHeaders:(NSDictionary *)value
```

Where, *value* contains the headers.

See "About Extension Headers and JSON Messages".

### Creating a WSC Frame

You can create a **WSCFrame** object with one of the following:

- **initWithJson**

  Use the **initiwithJson** method when you have a JSON string that contains the required data. In Example 13–55, the code excerpt passes *content* encoded as a JSON string with the **initiwithJson** parameter.

***Example 13–55 Creating a WSCFrame from File Name***

```
+ (WSCFrame*)frameFromFileName:(NSString*)fileName{
  NSString *path = [[NSBundle bundleForClass:[self class]] pathForResource:fileName
ofType:@"json"];
  NSString *content = [NSString stringWithContentsOfFile:path encoding:NSUTF8StringEncoding
error:nil];
  WSCFrame* frame = [[WSCFrame alloc]initWithJson:content];
  return frame;
}
```

- **createFrame**

  Use the **createFrame** method of the **WSCFrameFactory** class, as shown in Example 13–56.

***Example 13–56 Creating a WSCFrame from Factory***

```
+ (WSCFrame*)frameFromFactory:(MySubSession *)mySubSession {
   WSCFrame *frame = [WSCFrameFactory createFrame: WSC_MT_MESSAGE
                                             verb:@"myMadeUpVerb"
                                             pack:MY_PACKAGE_TYPE
                                        sessionId:[self.session getSessionId]
                                    subSessionId:[mySubSession getSubSessionId]
                                   correlationId:[mySubSession getCorrelationId]];
  return frame;
}
```

### Sending a WSC message Frame to the Session

To send a WSC message frame to a session, use the following syntax:

```
- (void)sendMessage:(WSCFrame *)frame
```

Where, *frame* is the WSC message frame that is sent to the session. See Example 13–57.

### Adding a SubSession to the Session

The **putSubSession** method adds a sub session to the WSCSession object. Example 13–57 shows how to add a sub session to a session (*MySubSession*) and send a WSC message frame to the session.

***Example 13–57 Adding a WSCSubSession and Sending a WSC Message Frame***

```
self.wscSession = [self createWSCSession];

MySubSession *mySubSession = [self createSubSession];
[self.wscSession putSubSession:mySubSession];

WSCFrame *myFrame = [self frameFromFactory:mySubSession];
[self.wscSession sendMessage:myFrame];
```

## Adding a New Package

"New packages can be added in both SDKs". How?

## About Extension Headers and JSON Messages

When you provide extension header to a method that supports the *extHeaders* parameter, the contents are inserted into the JSON message. For example, suppose that you input an extension header formatted as:

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

It is formatted into the JSON message as:

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

# 14

# 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 *Oracle Communications 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 14–1 lists the possible error codes and their descriptions.

***Table 14–1 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 14–1 (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 14–1.

***Example 14–1 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 14–2.

***Example 14–2 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 14–3 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 14–3 Subscription Creation Error***

```
function onSubscribeError(errorObj) {
    console.log("Error code: "+errorObj.code);
    removeSubscriptionInfoElem();
  };
```