Oracle® Communications WebRTC Session Controller

Extension Developer's Guide

Release 7.2

E69518-01

May 2016

ORACLE®

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

E69518-01

Contents

3 Creating WebRTC Session Controller Applications, Packages, and Criteria

4 Customizing Messages for New SIP or JSON Data

5 Using Policy Data in Messages

6 Anchoring Media Sessions

A JSONRTC Protocol Reference

Preface

This document describes the developer extensions for Oracle Communications WebRTC Session Controller product.

Audience

This document is intended for developers who use WebRTC Session Controller to make their SIP-based services available to users using WebRTC-enabled web browsers. WebRTC Session Controller does this by making your web-based JSON messages understandable to a SIP network, and your SIP messages understandable to JSON-based browsers and applications. This document also explains the points where the SIP to JSON translation is extendable to add new features and incorporate other features and technologies. This document further explains how to take advantage of the WebRTC Session Controller Media Engine media anchoring features, and how to incorporate WebRTC Session Controller Quality of Service (QoS) restrictions in your implementation.

Related Documents

For more information, see the following documents in the Oracle Communications WebRTC Session Controller documentation set:

- *Oracle Communications WebRTC Session Controller Concepts*
- *Oracle Communications WebRTC Session Controller System Administrator's Guide*
- *Oracle Communications WebRTC Session Controller Security Guide*
- *Oracle Communications WebRTC Session Controller Application Developer's Guide*
- *Oracle Communications WebRTC Session Controller Configuration API Reference*
- *Oracle Communications WebRTC Session Controller JavaScript API Reference*
- *Oracle Fusion Middleware Securing Resources and Policies for Oracle Weblogic Server*

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

About Extending WebRTC Session Controller

This chapter introduces the Oracle Communications WebRTC Session Controller Signaling Engine (Signaling Engine) and WebRTC Session Controller Media Engine (Media Engine) features that you use to establish communication between WebRTC-enabled browsers and SIP-based network services.

In order to use WebRTC Session Controller functionality, you must be familiar with

- The Session Initiation Protocol (SIP) transport protocol, and how it builds up and tears down calls.
- The JavaScript Object Notation (JSON) data format.
- JavaScript (JS), and how it communicates with web servers.
- The Java and Groovy Programming Languages that you use to create scripts that perform JSON to SIP and SIP to JSON translations within Signaling Engine.

About Extending WebRTC Session Controller Functionality

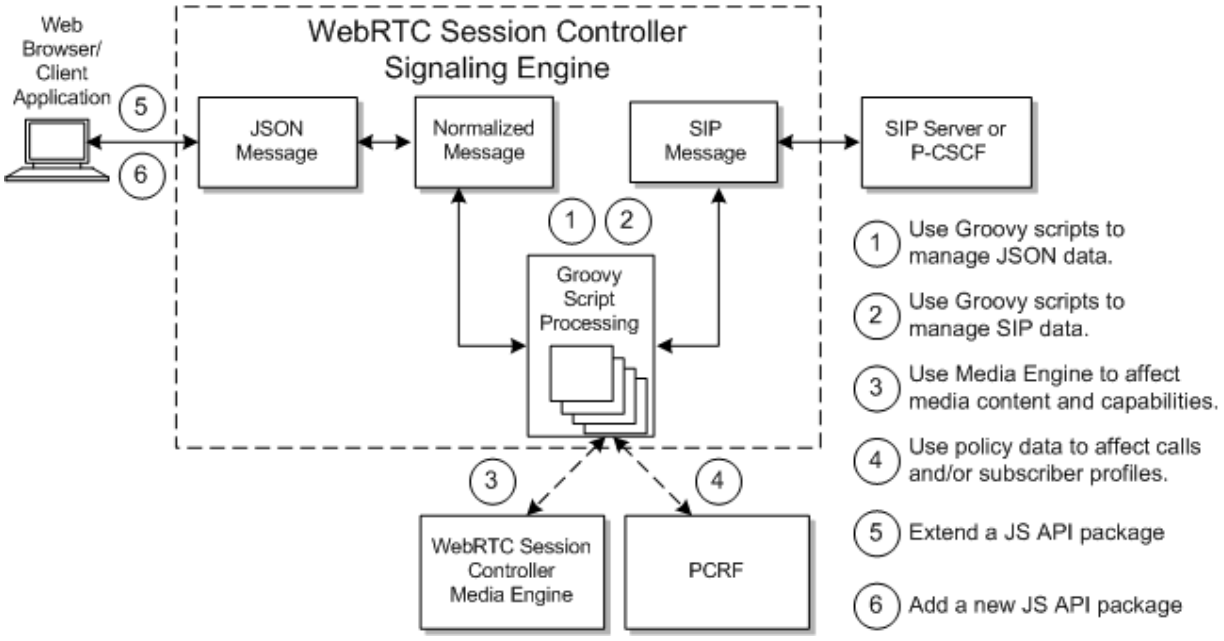
WebRTC Session Controller builds up and tears down real-time multimedia calls between client applications on WebRTC-enabled web browsers, and your SIP multimedia services. WebRTC Session Controller does this by translating telecom messages between the JSON data structure used by the client applications and the SIP protocol used by your IMS core.

See "WebRTC Session Controller Software and Protocol Conformance" for details on the protocol levels this release uses.

Figure 1–1 illustrates the WebRTC Session Controller architecture, lists the developer extension points you use to customize the JSON to SIP communication, and shows how JSON messages are translated to SIP and SIP messages to JSON.

Client applications send JSON messages to Signaling Engine, which uses Groovy scripts to process them and translate them to SIP messages and send them on to your SIP servers and IMS core. During Groovy processing you can use interact with a Policy Control and Rules Function (PCRF) to include policy (QoS) information, and Media Engine to manage the call's media session.

Figure 1–1 WebRTC Session Controller Components and Developer Extension Points



This guide explains how to customize Signaling Engine Groovy scripts to:

- Change the JSON data used in messages. For example, to modify your JSON data if the JSON specifications change.
- Change the SIP data used in messages. You can then add any additional SIP header data that your SIP implementation requires.
- Use Media Engine to affect the media parameters (SDP data) of media sessions. For example, you can use Media Engine to negotiate a codec supported by both call parties.
- Send and receive policy information from your PCRF. You can exchange information with a PCRF and affect the call (or the subscriber's account) either before or after the call's media session.
- Streamline communication between client applications and WebRTC Session Controller to provide a more lightweight and efficient protocol. You can tune the network traffic by filtering or aggregating messages to limit the call flow. For example, you could remove some or all of the SIP 1xx informational messages from the call flow as they arrive at WebRTC Session Controller. Or you could aggregate related SIP messages and forward them to the client application as a single combined message once WebRTC Session Controller has received them all. This reduces the amount of traffic that the client application must process, enabling you to simplify the client applications logic.

The *WebRTC Session Controller Application Developer's Guide* explains the extension points not covered in this document, including:

- How to extend the default WebRTC Session Controller JavaScript API package
- How to create a WebRTC Session Controller JavaScript API package

During the JSON to SIP translation you have a lot of flexibility in what you can do with individual messages. You can manipulate messages:

- At the field level, by adding new fields, creating a new field layout, adding optional data, and creating a new data representation.

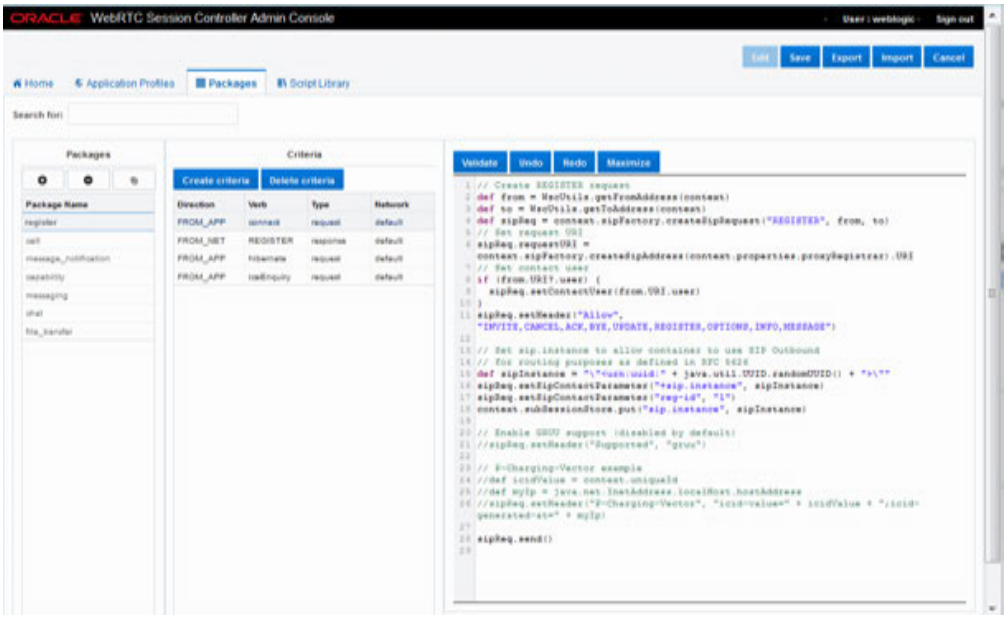
- At the header level, by separating a SIP message header from the message content, and pass on one or the other in a new message (for a different service). You can then pass one or both to a new service.

As shown in Figure 1–1, calls can originate either from a WebRTC-enabled web browser (client application), or from the SIP IMS core itself. Browser-originating messages are first translated from JSON to a normalized format by Signaling Engine. Then the normalized message is translated to the SIP protocol by Signaling Engine Groovy scripts calling methods from the WebRTC Session Controller API.

Signaling Engine processes SIP server-originated messages in the opposite direction. They are first translated from SIP to a normalized format using your Groovy scripts. Then Signaling Engine translates them again from the normalized format to the JSON format that your web browser can parse.

Figure 1–2 shows the WebRTC Session Controller console graphical interface that you use to create, select, and modify the Groovy scripts that translate and customize messages. In it, the Packages tab is selected, showing one of the default Groovy scripts provided with this release.

Figure 1–2 The WebRTC Session Controller Window



See "Customizing Messages for New SIP or JSON Data" for details on how to use the WebRTC Session Controller console.

About the WebRTC Session Controller Console Components

WebRTC Session Controller uses these concepts and components:

- WebRTC Session Controller applications - Each application represents a single WebRTC-enabled client application and all of its capabilities. For example, an application could be a website that offers a video and audio chat capabilities. Each application uses a set of WebRTC Session Controller packages.

  See "About Applications" for more details and "Creating Applications" for instructions on how to create them.

- WebRTC Session Controller packages - Each WebRTC Session Controller package is a unit of real time communication capability that WebRTC Session Controller supports. Each package is a collection of the individual Groovy scripts (criteria) that actually do the message processing and translation from SIP to JSON and back. Each JavaScript client application must reference the packages that it is allowed to use.

  Each package typically includes the logical set of message processing behaviors for an application. For example you might put all message translation scripts for video chat calls between a web browser and a SIP phone in one package. You would probably define a separate package for audio chat calls, and another for a sending SMSs, and another for file transfers.

  See "About Packages" for more details and "Creating Packages" for details on how to create one.

- WebRTC Session Controller criteria - Each criteria includes information to identify a SIP or JSON message to translate, and a Groovy script to do the translation. The translation can be from SIP to JSON or JSON to SIP. For example one criteria accepts the BYE message from a SIP phone to stop a media stream, and translates it to a JSON shutdown message.

  You use one criteria for each type of message that you expect to receive during the process of setting up or tearing down the WebRTC multimedia calls that WebRTC Session Controller processes. The Groovy scripts that you create and use are where you add code to process any new or custom SIP or JSON data that your implementation requires.

  See "About Criteria" for more details on WebRTC Session Controller criteria, and "Creating Criteria" for details on creating criteria.

- Script Library - The script library is a collection of useful groovy code and examples that you can use or reuse in any of the criteria Groovy scripts that you create.

  See "About the Groovy Scripts" for details on using the script library.

- WebRTC Session Controller console - A Graphical User Interface (GUI) that you use to create WebRTC Session Controller applications, packages, and criteria. This GUI also contains the Script Library, and a Configuration tab that use to configure Signaling Engine and Media Engine.

  See "About the WebRTC Session Controller Console" for information about this GUI. See the *WebRTC Session Controller System Administrator's Guide* for information on the configuration settings.

- Web RTC Session Controller console configuration settings - Used to set performance limits for each Signaling Controller implementation. You use these settings to balance WebRTC Session Controller capabilities with your network load and hardware capabilities. For example you use this tab to enable/disable glare handling (avoiding race conditions caused by concurrent requests), and set a maximum number of WebSocket connections.

  WebSocket is a protocol that provides bidirectional communication between a web client and a server over a Transmission Control Protocol (TCP) connection. RFC 6455 describes the protocol in detail. The World Wide Web Consortium (W3C) defines the WebSocket API.

About the WebRTC Session Controller Groovy Scripts

The WebRTC Signaling Controller Groovy scripts translate WebRTC messages between JSON messages to SIP, and also serve as extension points that you use to customize behavior during the translation. During the translation process you can:

- Make your WebRTC-based client web applications communicate with your SIP servers and IMS core.
- Make the client-to-SIP communication more efficient by removing unimportant messages. For example, your IMS core probably includes media servers, which send status messages that may by unimportant to a client application. You can add instructions to the Groovy scripts to remove any unnecessary messages from traffic, to save bandwidth.
- Redirect the messages to different SIP services. For example, during a shopping session, offer a video chat with a customer representative. As the chat session is being set up, you can automatically provision the chat session with subscriber information.
- Intercept, modify, or replace individual messages as they are being processed, within the same session. For example, you could modify or replace functionality based on the values included in the message parameters. If your message includes subscriber information, you could apply a special deal to all residents of a specific city; or shut off service to all residents of a city; or redirect suspicious messages to a quarantine area. You can make these decisions based on any information contained in the messages from your SIP servers or client application. You are only limited by the information in your JSON and SIP messages and the capabilities of the Groovy scripting language.
- Use policy control and charging rules function (PCRF) information to affect the call or the subscriber's account. You can use policy information to affect the call either before the media stream is set up or afterward.
- Use Media Engine data to change the call's Session Description Protocol (SDP) parameters. for example, to negotiate a codec that two web browsers support.
- Make arbitrary Representational State Transformation (REST) calls to remote REST endpoints.
- Add other features or behaviors that your implementation requires.

About Creating Client Applications Using the JavaScript API

You use the WebRTC Session Controller JavaScript API library to create multimedia applications that run on WebRTC-enabled browsers. These client applications use this API to communicate with the Signaling Engine, and Signaling Engine, in turn, translates the JSON data format to one the SIP nodes can use.

See *WebRTC Session Controller Application Developer's Guide* for details on how to use this API to create client applications.

About Translating Calls Using the Configuration API

You use the WebRTC Session Controller Configuration API, documented in the *WebRTC Session Controller Configuration API Reference* and *WebRTC Session Controller JavaScript API Reference* documents to translate call messages between the JSON data format that the client application uses, and the SIP language that your IMS core understands. It contains separate packages for:

- Creating, sending, and receiving SIP messages
- Creating, sending, and receiving JSON messages
- Sending and receiving policy (Diameter Rx) messages
- Creating and using Java **Future** interface objects to delay processing until computations by Media Engine or a PCRF are complete. For example you can delay establishing a media session until your PCRF confirms that the subscriber is entitled to the resources.
- Using Media Engine to translate between WebRTC-enabled browsers and entities that do not support the same codecs. These entities can be SIP nodes, or web browsers that do not support the codec sent in the call request.
- Administering Signaling Engine. WebRTC Session Controller API contains a configuration MBean. See *WebRTC Session Controller System Administrator's Guide* for details on using this MBean.

About Extending WebRTC Session Controller Using the JSONRTC Protocol

WebRTC Session Controller includes the JSONRTC WebSocket subprotocol that it uses to communicate with client applications and extend the default JavaScript capabilities. JSONRTC uses the JSON data interchange format and the MBWS subprotocol as the basis for message reliability.

If you use the WebRTC Session Controller console to create applications and packages, then you do not need to understand this protocol. However, if your implementation requires that you extend WebRTC Session Controller with new software packages, you use this protocol to do so.

You can use this protocol to change the extend the JSON and SIP data structures and add new fields and headers as necessary using the optional fields in the JSONRTC protocol. However, you must use a Groovy script to validate and use the data as necessary. WebRTC Session Controller accepts new data freely, but ignores if you do not process it in your Groovy scripts.

See "Extending WebRTC Session Controller Functionality" for guidelines for creating a custom package, and "JSONRTC Protocol Reference" for details on the WebRTC Session Controller JSONRTC Protocol.

WebRTC Session Controller Software and Protocol Conformance

WebRTC Session Controller uses the following revision levels of the software tools and protocols:

- The JSON data format for communicating with web browsers and other HTTP nodes.
- Session Description Protocol (SDP) RFC 4566 for communicating information about message media streams. The specification is available at the SDP specification website: `http://tools.ietf.org/html/rfc4566`
- The default JDK version - 1.7 plus any security updates.
- The Groovy scripting engine version 2.1.3.
- The SIP protocol RFC 3261 for building up, tearing down calls.

- A Groovy scripting language. See this Groovy website for information and documentation:

  http://groovy.codehaus.org/JSR+223+Scripting+with+Groovy

Prerequisites for Extending WebRTC Session Controller Functionality

Before using the instructions in this guide to configure and customize your WebRTC to SIP communication, you need to know:

- How to program in the Groovy scripting language. This Groovy website can get you started: http://groovy.codehaus.org. There are also third party tutorials and books available.
- Details of your WebRTC client application message requirements.
- Details of your SIP message requirements.
- Details of any policy information that you provide to a PCRF to affect subscriber profiles or accounts, and any policy information that you intend to use to affect calls.
- Details of the security groups that your WebLogic server uses. For details on using security roles, see the discussion on users, groups, and security roles in *Oracle Fusion Middleware Securing Resources and Policies for Oracle Weblogic Server*.
- Details on the other security considerations that your network requires. See *WebRTC Session Controller Security Guide* for information about setting up a secure WebRTC Session Controller implementation.

2

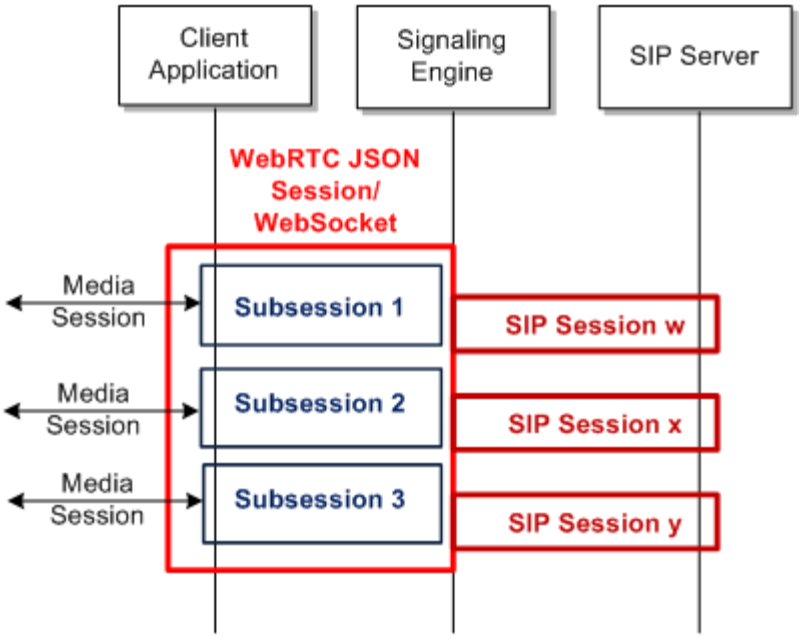
About Building JSON to SIP Communication

This chapter explains how you use WebRTC Session Controller Signaling Engine (Signaling Engine) to build up and tear down calls, and translate them between the JavaScript Object Notation (JSON) data format and the Session Initiation Protocol (SIP) protocol.

About Building JSON to SIP Communication

Figure 2–1 shows the sessions and subsessions used in making a simple JSON to SIP call flow. Client applications first set up JSON sessions that include WebSocket connections to start communicating with Signaling Engine. Signaling Engine then starts subsessions to communicate with the client application, and SIP sessions to communicate with your IP Multimedia Subsystem (IMS) core. The sections that follow explain information you need to know to set up this communication.

Figure 2–1 Signaling Engine Call Flow Overview



Securing Signaling Engine Connections

See the *WebRTC Session Controller Security Guide* for information on securing:

- Signaling Engine-SIP connections.

- Client application to Signaling Engine connections.
- Internal Signaling Engine internal components and processes.
- WebRTC to SIP connections.

About Connecting to a Client Application

Signaling Engine uses the JSONRTC protocol to communicate between client applications and the WebRTC Session Controller console. Signaling Engine establishes communication using an HTTP/HTTPS handshake message using a value of **webrtc.oracle.com** for **Sec-WebSocket-Protocol**.

See "JSONRTC Protocol Reference" and *WebRTC Session Controller Application Developer's Guide* for details on this protocol and how to use it to develop client applications.

About Sessions and Subsessions

Figure 2–1 shows an overview of how Signaling Engine handles JSON sessions, WebSockets, and subsessions, and how they relate to SIP sessions. First, Signaling Engine opens a protocol session using the WebRTC Session Controller Configuration API, and within that session Signaling Engine then opens a WebSocket connection. Inside the WebSocket connection, Signaling Engine uses the JSONRTC protocol to open a subsession to pass messages between the browser and the Signaling Engine. Finally Signaling Engine opens a SIP session to communicate with your IMS core.

There is one WebSocket connection for one WSCSession/WebRTC JSON session. However if a network problem interrupts the WebRTC session, the WebSocket connection can be reestablished with session information (rehydrated) if the problem is fixed before the connection timeout limit is reached. If the time limit is reached, the WebRTC session exits. See *WebRTC Session Controller System Administrator's Guide* for details.

Each subsession is associated with a WebRTC Session Controller package, which defines the allowable actions for the WebSocket and its subsessions. A subsession is the scope in which a particular package operates. There are generally several subsessions in a session.

Each subsession is responsible for maintaining the media session between the client application and the peer (media server, SIP phone, web client, and so on). When call is torn down, the media stream and the subsession are closed.

Usually each Signaling Engine subsession has one corresponding SIP session. However, a SIP session may not always required. If for example, a SubSession that just sends a SIP MESSAGE outside of a dialog would not create a media session or require a SIP session.

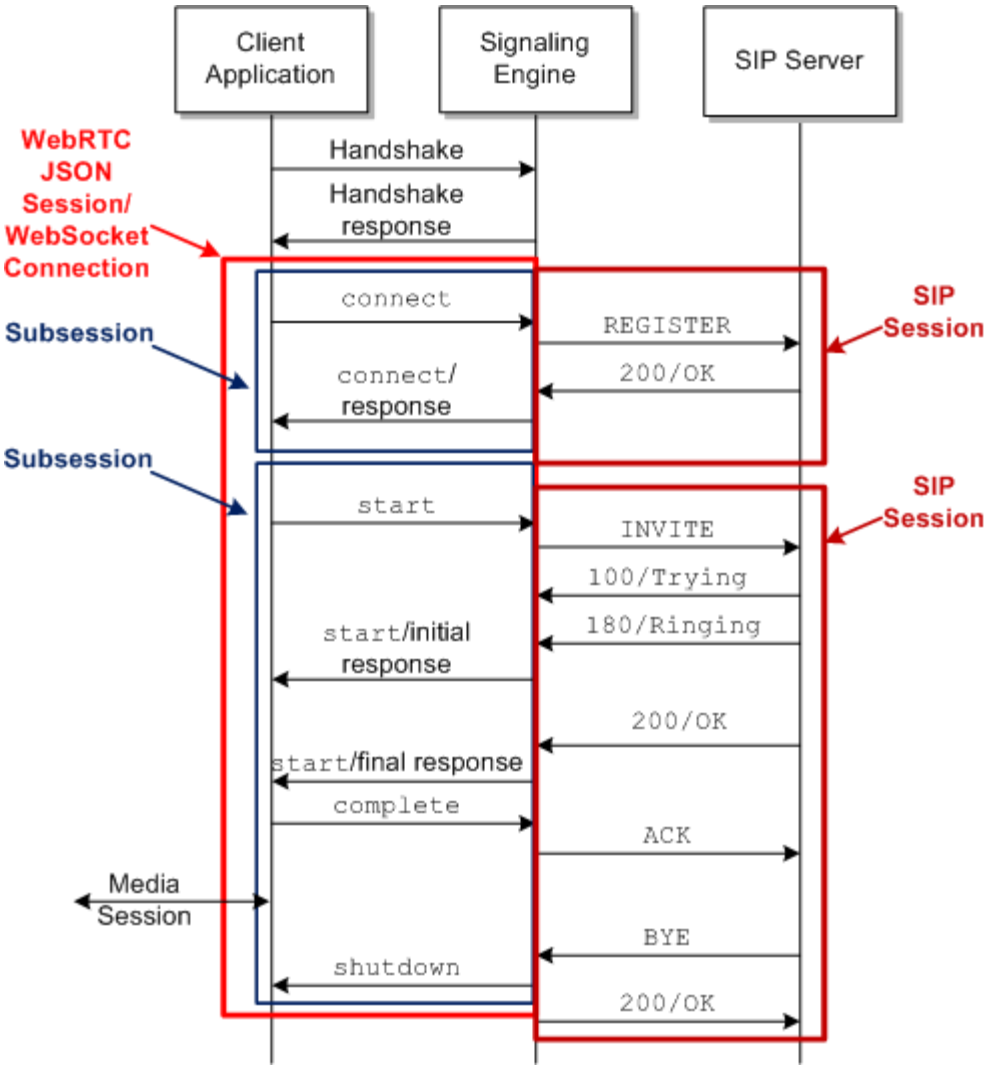
If a WebSocket is disconnected unexpectedly, Signaling Engine can start a new one to continue using the existing subsessions. This enables you to continue sessions if a WebSocket connection is unexpectedly terminated by a network failure, HTML refresh, or other service interruption.

A session has several states from INITIAL to TERMINATED. When the client sets up the WebSocket session with the server, the session is in an ESTABLISHED state. When the session is closed, it is in a TERMINATED state.

About JSON to SIP Communication

Figure 2–2 shows the JSON and SIP message flow for a call originating from the client application and how the messages relate to JSON sessions, WebSocket connections, subsessions, and SIP sessions.

Figure 2–2 Default WebRTC Session Controller FROM_APP Call Flow Detail



The client application initiates communication with Signaling Engine by sending a **wsc.session** object (not shown), which includes the handshake and **connect** message.

The call shown in Figure 2–2 originates from a client application (WebRTC-enabled browser) using the RFC WebSocket protocol. In this case Signaling Engine translates the JSON data into SIP protocol messages for the SIP server to respond do. The call recipient (not shown) may be a SIP device served by the SIP server itself, or another WebRTC browser using another Signaling Engine implementation.

As Figure 2–2 shows, Signaling Engine translates the JSON messages to SIP and passes them to the SIP server, and translates the SIP messages to JSON and passes them back to the client. Signaling Engine groups the Groovy-based translation code segments into *applications*, *packages*, and *criteria*. Each application represents all JSON to SIP communication for a collection of related Signaling Engine features. Each of the

features is composed of a package, which is a collection of individual criteria that each perform a single translation action. These criteria contain the individual Groovy scripts that perform each action.

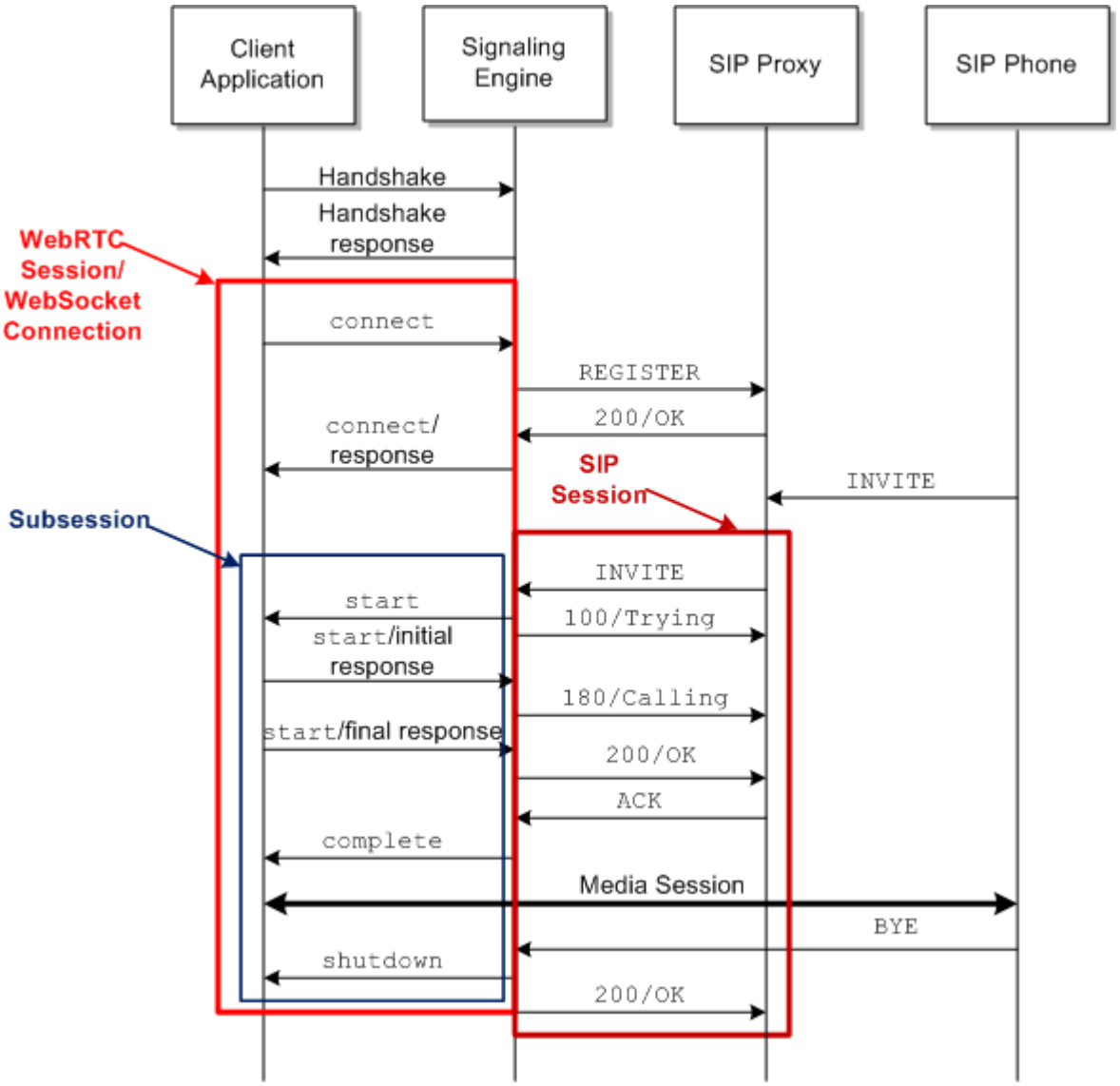
See "About SIP to Client Communication" for a description of how Signaling Engine processes messages that originate in your IMS core.

About SIP to Client Communication

This section explains how Signaling Engine processes messages what originate from your IMS core. For example, if a subscriber using a SIP phone attempts to communicate with a Signaling Engine client application. The call could originate from any SIP device or another Signaling Engine implementation. From the Signaling Engine perspective, these calls originate from a SIP server.

Figure 2–3 shows a sample call flow initiated by a SIP phone and how Signaling Engine translates the SIP messages from the phone's SIP server to the JSON data format that the client application can use for communication.

Figure 2–3 Default Signaling Engine FROM_NET Call Flow Detail



About Storing Data Within Sessions

As you are building up and tearing down calls, you will probably need to store data within Signaling Engine for different messages to use, such as customer subscriber data. Signaling Engine provides an attribute store to hold data that you use within sessions. You use the **getSessionStore()** class in the **oracle.wsc.feature.webrtc.template** package (**TemplateContext** interface) to retrieve data from the attribute store. The attribute store is created when a WebSocket session or SIP session is created within Signaling Engine, and is persistent until that session is torn down.

Understanding the WebRTC Session Controller Components

Using the WebRTC Session Controller console, you equate one WebRTC-enabled client application to a WebRTC Session Controller application. Each Signaling Engine application in turn, is a collection of WebRTC Session Controller packages that each roughly equate to a single real time WebRTC feature. Each package contains any number of Groovy scripts, called criteria, that each perform a single translation function on a single type of call message. The sections below provide more detail on applications, packages, and criteria.

About Applications

A WebRTC Session Controller application represents a single client application or service that sends messages between a browser and a SIP server through WebRTC Session Controller. Each application must have at least two packages, one for translating from the SIP server to the application and one for translating from the application to the SIP server. Each application has:

- An informal name.
- An active/inactive setting.
- An informal description.
- References to the WebRTC Session Controller packages that contain the Groovy scripts that performs message processing and translation.
- A Request URI that the application uses to communicate with a SIP server or proxy.
- The WebLogic security group. The security groups is required, and the Weblogic Server contains some default security groups that you can use. For details on using security groups, see the discussion on users, groups, and security roles in *Oracle Fusion Middleware Securing Resources and Policies for Oracle Weblogic Server*.
- A list of Allowed Domains that serve as a white list of domains the application is allowed to contact.
- Resource limits that allow you to protect system performance by limiting an application's impact on Signaling Engine. These resource limits can also serve as application white- and black-lists for individual applications.

See "About Packages" for details on the packages that comprise an application.

About Packages

A package defines a service provided by the JSONRTC protocol. It consists of a group of messages such as requests and responses that are sent between the client and the server. Table 2–1 lists the packages that are available through the WebRTC Session

Controller console on the **Packages** tab.

Table 2–1 WebRTC Session Controller Packages

Package	Description
call	Provides the ability to start a voice or video call, upgrade from a voice to a video call, establish a DataChannel.
capability	Provides exchange capability such as file transfer or video sharing.
chat	Provides one-to-one and group chat service.
file_transfer	Provides the ability to transmit files, share images, and so on.
message_notification	Provides the ability to notify the client of information such as pending voice mails, fax messages, and so on.
messaging	Provides a standalone message capability.
register	Registers a SIP session request.

Each WebRTC Session Controller package contains a group of criteria that contain the Groovy scripts to translate a logical group of messages from JSON to SIP or SIP to JSON. For example, Signaling Engine provides an example **call** package that includes all of the JSON (FROM_APP) and SIP (FROM_NET) criteria to build up and tear down a JSON to SIP call, including:

- JSON start messages (request, response, and error)
- JSON complete message
- JSON prack message
- JSON shutdown message
- SIP INVITE messages (request and response)
- SIP CANCEL message
- SIP ACK messages (request and response)
- SIP UPDATE messages (request and response)
- SIP PRACK message
- SIP BYE messages (request and response)

A package generally contains criteria for a single client application. Each package can be used by any number of Signaling Engine applications.

See the WebRTC Session Controller console to inspect the default packages provided, and "About Criteria" for details about the criteria that comprise a package.

About Criteria

Each WebRTC Session Controller criteria matches one kind of JSON or SIP message and runs the code in a Groovy script against it. For example, one criteria translates an INVITE request message from SIP to JSON, and another translates the response back from JSON to your SIP format.

Each criteria uses this information to identify the messages it translates:

- A **FROM_APP** or **FROM_NET** direction that specifies whether the message originated in a WebRTC-enabled browser, or your SIP IMS core.

- A verb matching the type of JSON or SIP request or response. For example, an UPDATE verb matches SIP UPDATE requests and response messages, and a **complete** verb matches a JSON **complete** request or response message.
- A type of message for the criteria to match. For example **request** or **response**. See "JSONRTC Protocol Reference" for the list of supported type values.
- A Network Service name. Each default criteria uses a Script Library call to return the network service of the call. A **default** network service name is the default.

See "About the Groovy Scripts" for details on the Groovy scripts.

By default, WebRTC Session Controller contains useful criteria that you can use to build and tear down calls, and use as stubs to add functionality that your implementation requires. You can use these default criteria as provided but you will probably modify them to fit your implementation's needs.

This is the default FROM_NET/INVITE/request/default criteria Groovy script:

```
def sipRequest = context.originatingSipMessage
def webMessage = context.webFactory.createWebRequest("start")
webMessage.header = [
  initiator : sipAddressToString(sipRequest.from),
  target    : sipAddressToString(sipRequest.to)
]
// SDP
if (sipRequest.sdp) {
  webMessage.payload = [sdp : sipRequest.sdp]
}
def sdpString = sipRequest.sdp

if(context.mediaFactory.isAvailable() && sdpString!=null) {
  def sdpOffer = context.mediaFactory.createSdpOffer("1", sdpString,
Constants.ME_CONFIG_NAME, null, sipAddressToString(sipRequest.to),
sipAddressToString(sipRequest.from));
  def ascFuture = sdpOffer.send()
  context.getTaskBuilder("processMediaResponseToSendWebMsg").withArg("ascFuture",
ascFuture).withArg("webMessage",webMessage).onSuccess(ascFuture).build();
}
else{
  webMessage.send()
}
```

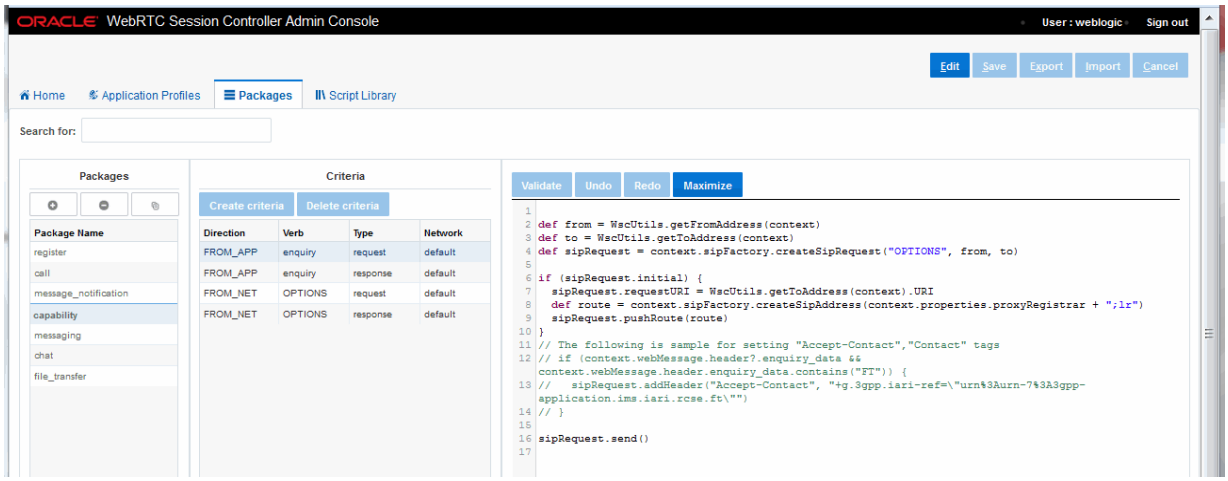
About the WebRTC Session Controller Console

You use the WebRTC Session Controller console to create, organize, use, and extend the applications, packages, and criteria for your implementation.

At the highest level, you use the **Applications** tab to create and manage applications, that roughly equate to a single web client application. See "Creating Applications" for details on creating applications.

Figure 2–4 shows the WebRTC Session Controller console with the **Packages** tab exposed. Each package is listed with its direction, verb, type, and network service. The Groovy script used in each criteria is shown on the bottom right of the pane. You enter or change the Groovy script for each package in this pane. You can also reference any existing Groovy code stored in the **Script Library** tab. See "Creating Packages" for details on creating packages for your applications.

Figure 2–4 WebRTC Session Controller Console Packages Tab



The **Script Library** tab is a repository of validated Groovy scripting code that you can reference in any of your packages.

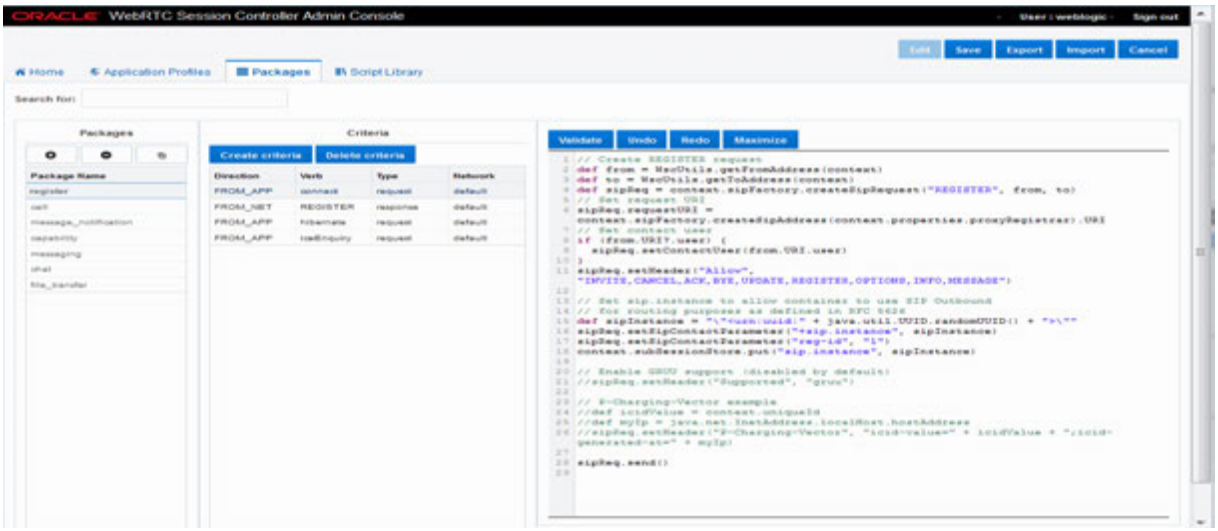
About the Groovy Scripts

Groovy is a scripting languages based on, and very similar to the Java programming language. Each Signaling Engine package contains its own individual Groovy scripts that translate messages matching its criteria specifications. The translation is either from WebRTC Session Controller's normalized format to SIP, or SIP to the normalized format depending on the direction you set. Packages require that you set up one criteria for translating in one direction, and a corresponding criteria for translating messages in the other direction. Signaling Engine then translates the normalized format to a format that your browser/application uses and back again.

Most packages use synchronous communication, so most criteria are created in pairs; one that translates messages from the client application (JSON) to SIP, and the other to translate from SIP to JSON. However the traffic can be asynchronous, as with Short Message Service (SMS) messages for example.

Figure 2–5 shows how criteria use Signaling Engine components to translate messages. The **register** package shown contains two criteria, one for **FROM_NET** messages and the other for **FROM_APP** messages.

Figure 2–5 Signaling Engine Criteria Components



At run time, Signaling Engine converts the criteria identifying information (**FROM_APP** or **FROM_NET**, the verb, message type, and network service) to a method name. The **TemplateContext** interface of the **oracle.wsc.feature.webrtc.template** package in the WebRTC Session Controller JSONRTC protocol is called to act on the Groovy script. This interface includes all of the other JSONRTC protocol interfaces that specify the specific Java methods that you can use to get information from or set information to a message.

See *WebRTC Session Controller Configuration API Reference* for details on this API.

Finally, every criteria contains a Groovy script that acts on the information obtained from the context. It is in these scripts that you add any new functionality or redirection instructions that change the behavior or destination of the message. In addition to simple translation, you can add any other processing that your implementation requires to each message. For example, you could:

- Map JSON information to SIP fields so that your SIP server accepts it.
- Map SIP header information to a form that your web application can use.
- Route the message to a specific URL based on its JSON information.
- Route the message to a specific URL based on its SIP information.
- Incorporate features such as redirecting a message to a different URL for example, to prevent bill shock.

See *WebRTC Session Controller Configuration API Reference* for details on the packages and methods of the API.

About Accessing the Parameters Using Groovy Scripts

This section describes the integration parameters associated with the Signaling Engine that are configured through the WebRTC Session. Table 2–2 shows the system integration parameters that you can access from your Groovy scripts.

Table 2–2 Accessing Integration and Package Filter Parameters

Parameter	Description
Proxy Registrar URI	Enter a SIP proxy server/Registrar URI. The value you enter in this field becomes the default SIP proxy server/Registrar URI for any new application you create. Access this parameter in Groovy as **context.properties.proxyRegistrar** using **SipContext**, **AuthenticationContext**, **TemplateContext**, or **WebContext**.
Dynamic Media Anchoring Type	Select a media anchoring option supported by WebRTC Session Controller. The possible selections are: ■ Disabled The application should not connect to the Media Engine. ■ web-to-web-anchor-conditional web to web conditional anchoring is used in a session when WebRTC-enabled browsers are allowed to communicate directly. If for some reason the browsers cannot communicate directly, they can communicate through WebRTC Session Controller. ■ web-to-web-anchored web to web forced anchoring is used in a session when all media flows through Media Engine. The supported Media Engine session type, is assigned to the Groovy constant ME_CONFIG_NAME_DMA, in the Groovy library
Media Engine MSRP	Select a message Session Relay Protocol (MSRP) to **Media Engine** from **Signaling Engine**. The possible selections are: ■ msrpwss-to-msrptcp ■ msrpws-to-msrptcp Access this parameter in Groovy as **context.properties.webMSRP** using **SipContext**, **AuthenticationContext**, **TemplateContext**, or **WebContext**.

Table 2–2 (Cont.) Accessing Integration and Package Filter Parameters

Parameter	Description
Signaling Engine MSRP	Select an MSRP to **Signaling Engine** from Media Engine. ▪ msrptcp-to-msrpwss This denotes the media engine configuration for MSRP to the WebRTC side. ▪ msrptcp-to-msrpws This denotes the media engine configuration for MSRP from the WebRTC side. Access this parameter in Groovy as **context.properties.netMSRP** using **SipContext**, **AuthenticationContext**, **TemplateContext**, or **WebContext**.
File Transfer	Enter a pattern for resolving file_transfer packages by SDP. The default pattern is: `a.*=.*file-selector` Access this parameter in Groovy as **context.properties.fileTransferPattern** using SipContext, AuthenticationContext, TemplateContext, or WebContext
MSRP	Enter a pattern for resolving MSRP packages by the Session Description Protocol (SDP). The default pattern is: `m.*=.*message.*MSRP` MSRP signaling is carried in SIP INVITE requests. When WebRTC Session Controller receives a SIP INVITE, it determines whether the request should be processed as a call, msrp chat or msrp file transfer. To do so, it looks at these regex expressions. Access this parameter in Groovy as **context.properties.msrpPattern** using **SipContext**, **AuthenticationContext**, **TemplateContext**, or **WebContext**.

For a description about configuring these parameters through the Administration console, see "Global Integration Parameters of the Signaling Engine" in *WebRTC Session Controller System Administrator's Guide*.

About the Contexts

You can access the parameters listed in through a number of contexts. In the access strings, the *context* entry can be **SipContext**, **AuthenticationContext**, **TemplateContext**, or **WebContext** interface. The code sample in Example 2–1 accesses a SIP package using the **SipContext** Interface.

Example 2–1 Accessing SipContext Interface of the SIP Package

```
@groovy.transform.CompileStatic
Map<String, Object> resolveProcessingParameters(final SipContext sipContext) {
  final WscSipMessage sipMessage = sipContext.wscSipMessage
  final def proxy = sipContext.properties.proxyRegistrar
  return [
          package_type    : resolvePackageType(sipMessage),
          network_service : resolveNetworkService(sipMessage)

}
```

For a description about these WSC API interfaces, see *WebRTC Session Controller Configuration API Reference*.

About the Script Library

The Groovy scripts you create for a package will probably use some functionality provided in the *script library*. The script library contains a set of useful methods that you can add to as required by your implementation. To use one of these methods, select the **Script Library** tab, click **Edit**, make your changes, then click **Save**. Example methods in the **Script Library** include code to:

- Return a user address based on the characteristics of a SIP string.
- Set the SIP routing URI.
- Set the SIP message contact parameter.
- Copy SDP data to a SIP header.

The Groovy code that you create in a package's criteria is appended to the code in the script library at run time. So you can reuse any of the library code in the scripts that you create for each individual package. You can also add more code to the library as needed, and use it in other individual packages.

After changing the code in either a package or the library, you need to validate it to ensure that it compiles correctly. You use the **Validate** button on the **Package** and **Script Library** tabs to validate individual package scripts or the script library.

Note: WebRTC Session Controller does not support global variables in the Groovy script library.

About the Normalized Data Format

As pictured in Figure 1–1 Signaling Engine converts messages into a normalized data format in the process of translating them between SIP and JSON. All messages are converted to the normalized format, regardless of whether they originated in a WebRTC client application, or from your IMS core.

The syntax for the normalized format is straightforward:

```
Map<String, Object>
```

For example, this JSON data format message:

```
{
"age":25,
"name":{
"first":"joe",
"last":"smith"
},
"messages":["msg 1","msg 2","msg 3"]
}
```

Is translated to this normalized message format, which is a hash map representation of the message:

```
{"age"=25,
{"name"={"first"="joe","last"="smith"}},
{"messages"=["msg1","msg2","msg3"]}}
```

Notice that the **messages** array is a nested hash map of its own.

Table 2–3 lists a variety of actions that you might perform while creating a Groovy translation script and the Groovy code that performs the action.

Table 2–3 Java and Groovy Actions on the Normalized Format

Translation Action	Groovy Code
Get the age	`def age = map.age`
Get the first name	`def firstName = map.name.first`
Get the list of messages	`def messages = map.messages`
Get message 2 from the list	`def message2 = messages[1]`
Modify the last name	`map.name.last = "doe"`
Add a middle name	`map.name.middle = "bob"`

3

Creating WebRTC Session Controller Applications, Packages, and Criteria

This chapter explains how to create the applications, packages, and criteria that WebRTC Session Controller Signaling Engine (Signaling Engine) uses to establish and modify communication between your client applications and IMS core.

You use the WebRTC Session Controller console graphical user interface (GUI) to create and manage the applications, packages, and criteria that Signaling Controller uses to translate and modify messages between client applications and your IP Multimedia Subsystem (IMS) core.

This procedure requires a running WebLogic server, and that you know the WebLogic username and password that you created for the domain. See the discussion on getting started in *WebRTC Session Controller System Administrator's Guide* for instructions on creating and starting a WebLogic domain.

To establish and modify communication between your client applications and IMS core, you do the following:

- Create the signaling engine criteria. See "Creating Criteria."

Creating Criteria

Each Signaling Engine criteria contains a single Groovy script that performs all translation and processing tasks for a single type of JavaScript Object Notation (JSON) or Session Initiation Protocol (SIP) message. You must create separate criteria for all possible JSON or SIP message that your Signaling Engine implementation processes. In synchronous request/response communication, you must create a separate criteria for each request and response message.

Before creating a new criteria, look through the Groovy code in the default packages, and in the **Script Library** to see whether there is already some code that accomplishes what your message requires.

Criteria are applied to messages based on this information included in each criteria, such as the direction from which a message originates, the SIP method or JSON action that the criteria matches, the type of message, and an identifier for the application that the message is associated with.

To create the criteria and Groovy script processing necessary to implement your new package, access the WebRTC Session Controller console. For details on creating criteria, see the description about "Managing Package Criteria" in *WebRTC Session Controller System Administrator's Guide*.

Creating Packages

A package is a collection of all the criteria (Groovy scripts) necessary to translate the telecom messages in a session from JSON to SIP and back. So creating a new package really just creates a shell that you fill with criteria. This procedure assumes that you have already created the criteria required.

To create the new package, access the WebRTC Session Controller console. For details on creating criteria, see the description about "Configuring Messaging Packages" in *WebRTC Session Controller System Administrator's Guide*.

Creating Applications

Each application is a collection of packages that contain the criteria that translate (and probably change) WebRTC application to SIP network communication for a single program. This procedure assumes that you have already created the criteria and packages required.

Applications reference your WebLogic security groups. Create any security groups your implementation requires before following this procedure.

To create applications, access Application Profiles tab in the WebRTC Session Controller console. For details on creating and registering applications, see the description about "Managing WebRTC Session Controller Application Profiles" in *WebRTC Session Controller System Administrator's Guide*.

Exporting and Importing a Configuration

You can export your current configuration settings to a file or import a set of configuration settings from a file to which a configuration instance was previously saved. For details on creating and registering applications, see the description about "Exporting and Importing a Configuration" in *WebRTC Session Controller System Administrator's Guide*.

Debugging Groovy Script Run Time Errors

You can diagnose Groovy script problems using the stack trace in the *domain_home*/**wsc.log** file, where *domain_home* is the name of the WebRTC Session Controller domain. This file contains the Signaling Engine stack trace messages. You identify the individual Groovy script by searching for the individual criteria method name that contains the criteria information.

For details on debugging your groovy scripts, see the description about "Debugging Groovy Script Run Time Errors" in *WebRTC Session Controller System Administrator's Guide*.

About the WebRTC Session Controller Console Validation Tests

The WebRTC Session Controller console runs validation tests to confirm that your Groovy scripts, Groovy library, packages, and applications are all valid. It runs the validation tests each time you commit changes to an application, package, or criteria, or click the **Validate** button.

For the complete list of the error types and the messages, see "About the WebRTC Session Controller Console Validation Tests" in *WebRTC Session Controller System Administrator's Guide*.

4

Customizing Messages for New SIP or JSON Data

This chapter contains examples of how to use Oracle Communications WebRTC Session Controller Signaling Engine (Signaling Engine) to process customized Session Initiation Protocol (SIP) data in messages, and add new JavaScript Object Notation (JSON) data to support protocol changes.

Processing Messages With Custom SIP Data

This section provides some examples for how to translate SIP messages which contain custom SIP data.

Example SIP Request Variable

The examples in this chapter assume that you have created a custom **sipReq** variable as shown in this example:

```
// Create REGISTER request
def from = getFromAddress(context)
def to = getToAddress(context)
def sipReq = context.sipFactory.createSipRequest("REGISTER", from, to)

// Set request URI
sipReq.requestURI = context.sipFactory.createSipAddress(Constants.PROXY_SIP_
URI).URI

// Set contact user
if (from.URI?.user) {
  sipReq.setContactUser(from.URI.user)
}

// Set sip.instance to allow container to use SIP Outbound
// for routing purposes as defined in RFC 5626
def sipInstance = "\"<urn:uuid:" + java.util.UUID.randomUUID() + ">\""
sipReq.setSipContactParameter("+sip.instance", sipInstance)
sipReq.setSipContactParameter("reg-id", "1")
context.subSessionStore.put("sip.instance", sipInstance)

sipReq.send()
```

Propagating Custom Headers to SIP and Browser Endpoints

To propagate custom headers to SIP or Browser endpoints, you need to implement the following additional logic in the WSC groovy scripts:

- Adding the Custom Header to the SIP endpoint

  Retrieve the extra header from the JSON message and add it as a SIP header. To do so, modify the FROM_APP_START_REQUEST, as shown here:

```
 if(context.webMessage.header?.NAME_OF_THE_HEADER) {
    sipRequest.setHeader(NAME_OF_THE_HEADER,context.webMessage.header.NAME_OF_
THE_HEADER)
}
```

- Adding the Custom Header to the Browser endpoint

  Retrieve the extra header from the SIP INVITE message and add it to the JSON message. To do so, modify the FROM_NET_INVITE_REQUEST, as shown here:

```
webMessage.header = [
  initiator : sipAddressToString(sipRequest.from),
  target    : sipAddressToString(sipRequest.to),
  test      : sipRequest.getHeader(NAME_OF_THE_HEADER)
]
```

Extending SIP Messages with New Headers

This Groovy code snippet from the default **register** package, in the **FROM_APP/connect/request/default** criteria (commented out) adds support for a Globally Routable User agent URI (GRUU).

```
sipReq.setHeader("Supported", "gruu")
// P-Charging-Vector example
def icidValue = context.uniqueId
def myIp = java.net.InetAddress.localHost.hostAddress
sipReq.setHeader("P-Charging-Vector", "icid-value=" + icidValue +
";icid-generated-at=" + myIp)
```

Protecting System Performance by Removing SIP Messages

You can save network bandwidth by removing unimportant messages during processing. For example, you would use this code snippet to remove provisional SIP responses (the 1xx SIP messages). You would put this in the Groovy script for the FROM_NET/INVITE/response criteria:

```
if (sipResponse.status < 200) {
  // Ignore provisional responses
} else if (sipResponse.status < 300)
  // Proceed with processing
}
{...
}
```

Removing a SIP Header in a Message

Use this Groovy code snippet to remove a header. Headers cannot be renamed.

```
sipReq.removeHeader("headername")
```

Replacing a SIP Header in a Message

You use the **setHeader** method to replace a header in a SIP message. Setting a header overwrites its value.

Conditionally Passing SIP Headers in Messages

This example Groovy code snippet probes for a JSON parameter called **myWebParmeter** and if present it copies the value to a SIP header.

```
def myWebParameter = context.webMessage?.header.?myParameter
if (myWebParameter) {
  sipRequest.setHeader("MyHeader", myWebParameter)
}
```

You pass SIP headers as extension headers (**extHeader**) in the JSON API. See *WebRTC Session Controller Application Developer's Guide* for examples of using extension headers.

Changing JSON Data to Support Protocol Changes

If the JSON protocol specification changes, you can add processing for additional data in your Groovy scripts. WebRTC Session Controller ignores new JSON data if you do not use it in processing.

Retrieving Session Addressing Information from Groovy

In certain instances, you may need to retrieve connection information from a particular WebRTC Session Controller session, in order, for instance, to provide media service route lookups to a WebRTC Session Controller Media Engine. To facilitate such communication, an **authenticationContext** object is passed to the **buildSecurityContext** method of the Groovy script library. Table 4–1 lists the properties of the **authenticationContext** object.

Table 4–1 authenticationContext Properties

Property	Description
authContext.properties.connection.remote_ip	Returns the Internet Protocol (IP) address of the client or the last proxy that sent the request.
authContext.properties.connection.remote_port	Returns the IP source port number of the client or the last proxy that sent the request.
authContext.properties.connection.local_ip	Returns the IP address of the interface on which the request was received.
authContext.properties.connection.local_port	Returns the IP port number of the interface on which the request was received.
authContext.properties.connection.server_name	Returns the host name of the server to which the request was sent.
authContext.properties.connection.headers	Returns the Websocket headers such as origin, host and others. For a complete listing, see https://tools.ietf.org/html/rfc6455#page-25

The **authenticationContext** properties are initiated during the session handshake upon client connection and are available to all WebRTC Session Controller Groovy packages.

Example 4–1 shows how to retrieve the **authenticationContext** properties from Groovy.

Example 4–1 Retrieving authenticationContext Properties

```
// Retrieve the properties from the authenticationContext...
def properties = authContext.properties
// Retrieve the connection associated with the properties...
def connection = properties.connection

// Print the properties to the WebLogic console...
println "Remote IP address: "+connection.remote_ip
println "Remote port: "+connection.remote_port
println "Local IP address: "+connection.local_ip
println "Local port "+connection.local_port
println "Server name: "+connection.server_name
println "Websocket headers "+connection.headers
```

Initiating REST Calls from Groovy

This section describes how you can initiate arbitrary Representational State Transformation (REST) calls to external network endpoints.

The WebRTC Session Controller REST call functionality supports the following features:

- Asynchronous and synchronous callback responses
- Support for HTTP and HTTPS
- Support for all standard REST methods:
  - **GET**
  - **POST**
  - **PUT**
  - **DELETE**
  - **HEAD**
  - **OPTIONS**
- Support for REST calls during message processing or WebSocket connection establishment

For complete details on the Groovy REST API, see *WebRTC Session Controller Configuration API Reference*.

Adding a REST URI Endpoint Constant

As a matter of convenience and to simplify maintenance, you should define a Groovy constant for your REST endpoint URI. In the global constants block of the WebRTC Session Controller Groovy script library, add a line similar to Example 4–2, replacing *server*, *port* and *rest_endpoint* with the correct values for your configuration.

Example 4–2 Defining a REST URL Endpoint Constant

```
public static final MY_REST_URL = "http://server:port/rest_endpoint"
```

With the constant defined, you can reference from the script library or WebRTC Session Controller packages similar to Example 4–3.

Example 4–3 Referencing the URL Constant

```
def restRequest = context.restClient.createRequest(Constants.MY_REST_URL...);
```

Creating a REST Request in Groovy

To create a REST request in Groovy, you use the WebRTC Session Controller **restClient** object's **createRequest** method:

```
def restRequest = context.restClient.createRequest(rest_url[, http_method][, synchronous]);
```

The *rest_url* parameter is required and represents a valid REST endpoint. The *http_method* parameter is a valid REST HTTP method, while the *synchronous* parameter is a boolean value indicating, if true, that the REST invocation is synchronous. Both the *http_method* and *synchronous* parameters are optional, and, if omitted, a REST request is created using the provided *rest_url* and the HTTP **GET** method by default.

> **Note:** REST requests created in WebRTC Session Controller packages must always be asynchronous. If you initiate REST calls from the **buildSecurityContext** Groovy library script, they must be synchronous.

In Example 4–4, an asynchronous REST request is created using the REST endpoint constant from Example 4–2 and the HTTP **PUT** method.

Example 4–4 Creating a REST Request

```
def myRestRequest = context.restClient.createRequest(Constants.MY_REST_URL, "PUT");
```

Configuring the REST Request

Once the REST request is created, you can customize it using the following methods:

- **addHeader**(string *name*, string *value*): add an arbitrary header to the REST request
- **setAccept**(RestMediaType *mediatypes*): define the data types the REST endpoint accepts, defined as a **RestMediaType** enum:
  - **APPLICATION_JSON**: an application/json content type
  - **APPLICATION_XML**: an application/xml content type
  - **TEXT_PLAIN**: a text/plain content type
- **setAcceptLanguage**(string *locales*): a string defining the acceptable locales
- **setEntity**(object *entity*, RestMediaType *mediatype*): set the request entity for REST POST and PUT methods

Note: Depending upon the **RestMediaType**, the following requirements apply to the **setEntity** method:

- **APPLICATION_JSON**: the entity object should be an object expected by the call method of groovy.json.JsonBuilder, or a groovy.json.JsonBuilder object which is a groovy.lang.Writable.
- **APPLICATION_XML**: the entity object should be a groovy.lang.Closure object as expected by the bind method of groovyl.xml.StreamingMarkupBuilder or a groovy.lang.Writable object obtained from groovy.xml.StreamingMarkupBuilder.
- **TEXT_PLAIN**: the entity object should be a java.lang.string.

For more details on Groovy objects and data types, see the Groovy documentation at http://groovy-lang.org/documentation.html.

- **setStackConfiguration**(string *name*, object *value*): adds a property supported by the underlying REST stack implementation

Example 4–5 configures some basic parameters for the REST request object created in Example 4–4.

Example 4–5 Configuring a REST Request Object

```
myRestRequest.setAccept(APPLICATION_XML);
myRestRequest.setAcceptLanguage("en-us", "de-de", "fr-fr");
myRestRequest.addHeader("My Key", "My Value");
```

Sending the REST Request

In Example 4–6, using the REST request created Example 4–4, you use the request object's **send** method to send the REST request. Example 4–6 provides as arguments to the **send** method an optional *entity*, in this case an XML snippet, and also provides the *RestMediaType* of the entity. The **send** method may also be called with no arguments, and returns the future of the REST invocation, here stored in the variable *myRestFuture*.

Example 4–6 Sending the REST Request

```
def xml = {
  mkp.xmlDeclaration()
  fish {
    name("salmon")
    price("10")
    }
};

def myRestFuture = myRestRequest.send(xml, APPLICATION_XML);
```

Note: Specifying the *entity* and *RestMediaType* arguments for the send method is equivalent to using the **setEntity** method described in "Configuring the REST Request."

Handling REST Responses

In order to handle REST responses from asynchronous REST requests, using the **context** object's **taskBuilder** method, you bind the **restClient** context to a Groovy callback function that will handle the REST response.

In Example 4–7, *myRestFuture* from Example 4–6 is bound to the Groovy function **processResponse** for **getTaskBuilder**'s **onSuccess** and **onError** methods.

Example 4–7 Binding the REST Response to a Groovy Callback

```
context.getTaskBuilder("processResponse").withArg("myRestFuture", restFuture)
            .onSuccess(restFuture).build();
context.getTaskBuilder("processResponse").withArg("myRestFuture", restFuture)
            .onError(restFuture).build();
```

> **Note:** While the **onError** and **onSuccess** methods in Example 4–7, are bound to the same **processResponse** function, you can choose different functions for each depending upon your requirements.

The **processResponse** Groovy function referenced in Example 4–7 can be defined in the WebRTC Session Controller script library to process the REST response. Example 4–8 shows a basic example using the XML document defined in Example 4–6.

Example 4–8 REST Response Handler

```
void processResponse(TemplateContext context) {
  def result = context.taskArgs.restFuture.get()
  if (result.status == 200) {
    def fish = result.value()
    if (fish.name.text() == "salmon") {
      // Continue processing...
    }
  } else {
    // Handle any errors...
  }
}
```

The **RestResult** variable, *resp*, itself provides the following utility methods that you can use when processing the REST response:

- **getAllow**
- **getCookies**
- **getEntityTag**
- **getHeaders**
- **getLanguage**
- **getLastModified**
- **getLength**
- **getLocation**
- **getResponseData**
- **getStatus**
- **hasEntity**

- **value**

For details on those methods, see *WebRTC Session Controller Configuration API Reference*.

REST Authentication

The WebRTC Session Controller REST API supports basic and digest authentication schemes. You use the methods **setCredentials** and **setDigestCredentials** to specify username and password for basic and digest authentication respectively:

- **setCredentials**(string *username*, byte[] *password*)
- **setDigestCredentials**(string *username*, byte[] *password*)

> **Note:** The API accepts encrypted passwords that you can retrieve using the **weblogic.security.Encrypt** utility.

In Example 4–9, a synchronous REST request is created, *myRestAuthRequest*, and the **setCredentials** method is used to initialize a *username* and *password*.

Example 4–9 Creating a Basic Authentication REST Request

```
def myRestAuthRequest = context.restClient.createRequest(Constants.MY_REST_URL, "PUT", true);
def username = "myuserid";
def password = [231, 245, 675, 232, 123] as byte[];
myRestAuthRequest.setCredentials(username, password);
```

Useful XML Groovy Utilities for REST Calls

If you are using XML markup in your REST requests and responses, there are Groovy utilities that can streamline much of your work:

- **StreamingMarkupBuilder**: a utility that simplifies building XML documents
- **XmlSlurper**: a utility that simplifies reading and formatting XML documents

For more details on those and other Groovy utilities, see http://groovy-lang.org/documentation.html#apidocumentation.

Extending WebRTC Session Controller Functionality

If your implementation requires client application logic that WebRTC Session Controller or Javascript does not support by default, you need to create new software packages to implement it. The procedure below offers guidelines for creating a new package. The exact steps and sequence depend on your requirements.

See "JSONRTC Protocol Reference" for details on the JSONRTC protocol that WebRTC uses to communicate with client applications. Also, see "Prerequisites for Extending WebRTC Session Controller Functionality" for information on other protocols you may need to understand.

To create a new package:

1. Design your new package.

   Include the new JSON to SIP message mapping and any new JSON and SIP data, formats, and headers.

2. To create the criteria and Groovy script processing necessary to implement your new package, access the WebRTC Session Controller console.

   For details on creating criteria, see the description about "Configuring Package Criteria" in *WebRTC Session Controller System Administrator's Guide*.

3. Create or extend the tools necessary to use the package with a client application.

   - If you use the JavaScript Development Environment client operating system, see the *WebRTC Session Controller Application Developer's Guide* for more information.
   - If you use a different client operating system, see that operating system documentation for details. You may also find the *WebRTC Session Controller Application Developer's Guide* helpful.

4. Write the client application.

   - To develop JavaScript client applications see the *WebRTC Session Controller Application Developer's Guide* for more information.
   - To develop client applications in another operating system, see that operating system documentation for information on how to communicate with WebRTC Session Controller.

5

Using Policy Data in Messages

This chapter explains how Oracle Communications WebRTC Session Controller Signaling Engine (Signaling Engine) uses policy data from policy charging rule functions (PCRFs) to affect subscriber calls and profiles.

About Using Policy Control Data with Signaling Engine

Signaling Engine supports using its Groovy script translation capability to make policy (QoS) decisions by using the policy information contained in Diameter Rx interface messages. Signaling Engine acts as a Diameter application function (AF) by exchanging Diameter Rx messages with your policy control and charging rules function (PCRF) in a 3GPP architecture.

Signaling Engine supports sending AA Request (AAR) and Session Termination Request (STR) Diameter Rx messages from Signaling Engine to your PCRF, and using the data from AA Answer (AAA) and Session Termination Answer (STA) messages that it receives in return.

The Diameter Rx messages and their responses are frequently used with the **pcrfFuture** interface that enables you to delay processing until a later message arrives. Oracle expects that most implementations will send Diameter AAR requests and then delay the media session until they receive an AAA confirming that the subscriber is entitled to the service.

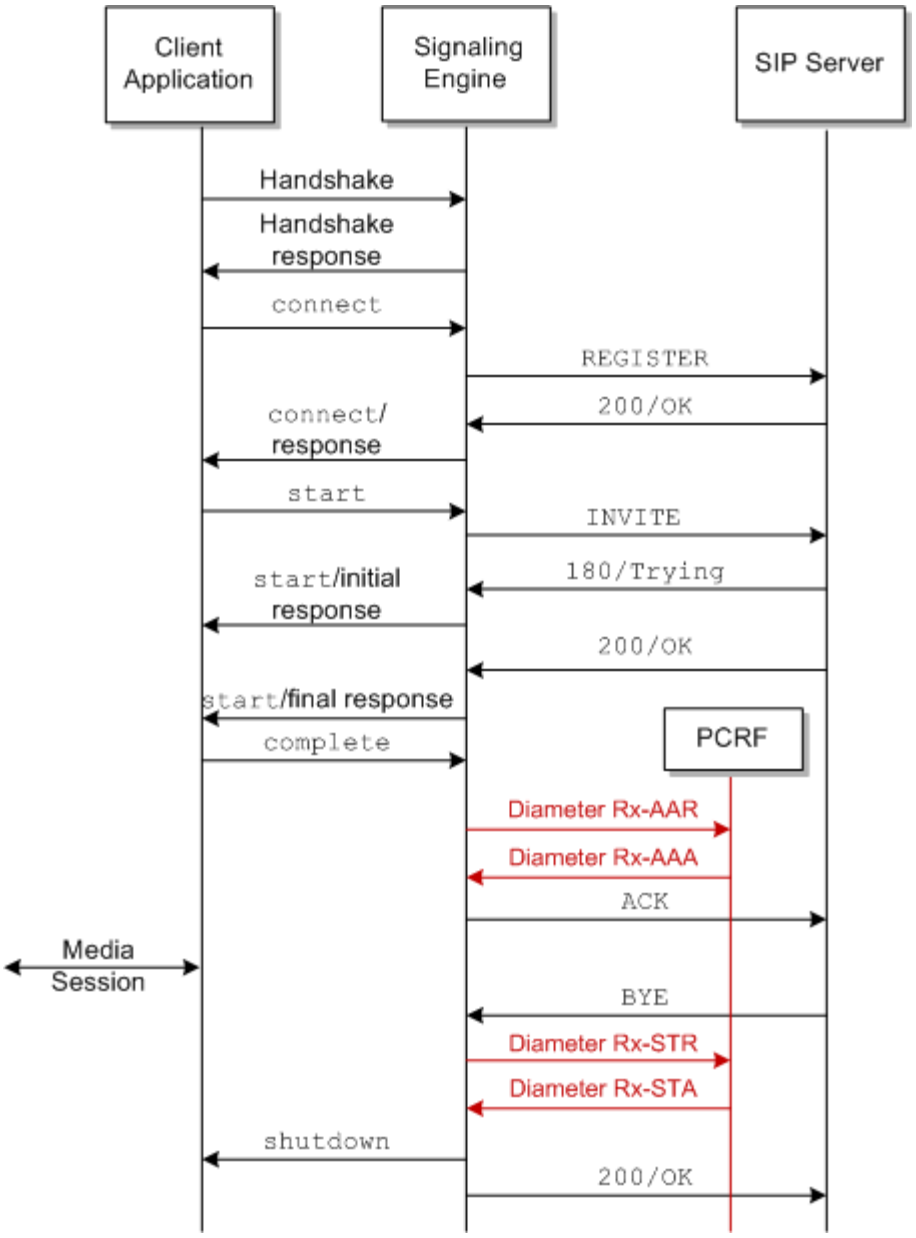
The AAR and AAA messages can be exchanged any time before a call's media stream, and the STR and STA messages are exchanged after the stream. So you can affect your PCRF and PCEF affect the subscriber profile before the media stream resources are used, update the subscriber's profile after the media stream resources have been consumed, or both.

See *WebRTC Session Controller Statement of Compliance* for the complete list of Diameter Rx commands and AVPs that Signaling Engine Supports.

Before the AAR and STR messages can be useful, you must configure your PCRF to accept and make policy decisions based on the AVPs that you send them. If your implementation requires it, you must also configure a PCEF to enforce those decisions.

Figure 5–1 shows an example call flow in which Signaling Engine exchanges messages with a PCRF both before and after the call's multimedia stream. Diameter Rx AAR, AAA, STR, and STA messages are shown in red in the call flow.

Figure 5–1 Signaling Engine Call Flow with PCRF Support



Creating and Sending Diameter Rx Request messages

You use the **createRxAAR** and **createRxSTR** methods in the **WscDiameterFactory** interface of the **oracle.wsc.feature.webrtc.template.diameter** package to create AAR and STR messages. These methods accept a map of AVPs that you create, and adds them to a Diameter Rx message that your PCRF can parse. Your PCRF then accepts the AVPs and take whatever action that you have configured it.

These AVPs are automatically added to each outgoing request and need not be specified in a Groovy script:

- Session-Id
- Origin-Host
- Origin-Realm

- Auth-Application-Id
- Destination-Realm

You must specify any other AVPs that your implementation requires in your Groovy scripts. See *WebRTC Session Controller Statement of Compliance* for details on the AVPs supported.

This example defines an AAR message and specifically defines the AVPs used (for example: Subscription-Id, Subscription-Id-Type, and Subscription-Id-Data):

```
 def avps = [
      'Subscription-Id':[
        'Subscription-Id-Type':2, //END_USER_SIP_URI
        'Subscription-Id-Data':"bob@example.com"
      ],
      'Framed-IP-Address':[
        0x84,
        0x08,
        0x88,
        0x65] as byte[],
      'AF-Application-Identifier':"WSE".getBytes("utf-8"),
      'Media-Type':0, //Audio
      'AF-Charging-Identifier':'charing-id-55'.getBytes("utf-8"), //Audio
      'Media-Component-Description':[
        'Media-Component-Number':[0, 1],
        'Media-Sub-Component': [
          [
            'Flow-Number':1,
            'Flow-Description':'permit out 8001 from assigned 34 to 24.2.1.6/18
8000'
          ],
          [
            'Flow-Number':1,
            'Flow-Description':'permit out 8005 from assigned 36 to 24.2.1.6/18
8001'
          ]
        ],
        'Flow-Status':2
      ]
    ]

   def aar = context.diameterFactory.createRxAAR(avps)
```

After creating a Diameter request message, you must explicitly send it using a **send** method call. **send** is a method in the **WscDiameterRequest** interface in the **oracle.wsc.feature.webrtc.template.diameter** package. This example sends an AAR message, and provides example success and error conditions:

```
def pcrfFuture = aar.send();

//success
context.getTaskBuilder("processSuccessFromPcrf").withArg("sipRequest",sipRequest)
       .withArg("pcrfFuture", pcrfFuture).onSuccess(pcrfFuture).build();

//error
context.getTaskBuilder("processErrorFromPcrf").withArg("sipRequest",sipRequest)
       .withArg("pcrfFuture", pcrfFuture).onError(pcrfFuture).build();
```

This example lists the **pcrfSuccessHandler** and **pcrfErrorHandler** methods that you would define to handle the success and failure conditions.

You use the methods in the **PcrfFuture** interface in the **oracle.wscfeature.webrtc.template.diameter** package to determine if any future objects are ready for use by your Groovy scripts. This interface extends the **oracle.wsc.feature.webrtc.template.future** interface.

This example checks the AVP values in the response to confirm that the subscriber **bob@example.com** uses a media type of **0**.

```
def avps = context.taskArgs.pcrfFuture.get().getAvps()

if(avps.'Subscription-Id'?.'Subscription-Id-Data'=='bob@example.com"){
  //add logic here.
}else if(avps.'Media-Type'==0){
//provide alternative
}
```

Accepting and Using Diameter Rx Answer Messages

You use the **getAvps**, **getCommandCode**, and **getResultCode** methods in the **WscDiameterResponse** interface of the **oracle.wscfeature.webrtc.template.diameter** package to process the Diameter Rx AAA and STA messages returned by your PCRF. **getCommandCode**, returns the command code identifying the type of message (265 for AAR and AAA, and 275 for STR and STA). **getResultCode** returns the integer values for the Result-Code AVP. **getAvps** returns a map of all the AVPs in the AAA or STA message. You use this method in groovy scripts you create to obtain the data necessary to perform policy actions, and take those actions.

6

Anchoring Media Sessions

This chapter explains how to use the Oracle Communications WebRTC Session Controller Media Engine (Media Engine) features to anchor media sessions.

About the WebRTC Session Controller Media Server

You use Media Engine to:

- Establish communication between a WebRTC-enabled browser and a Session Initiation Protocol (SIP) or a public switched telephone network (PSTN) device.
- Establish communication between two end points (WebRTC-enabled browsers, or SIP or PSTN based devices) that do not share a common codec they can use to communicate directly.
- Enable a content service provider to forcibly anchor a call for example, to lawfully intercept it.

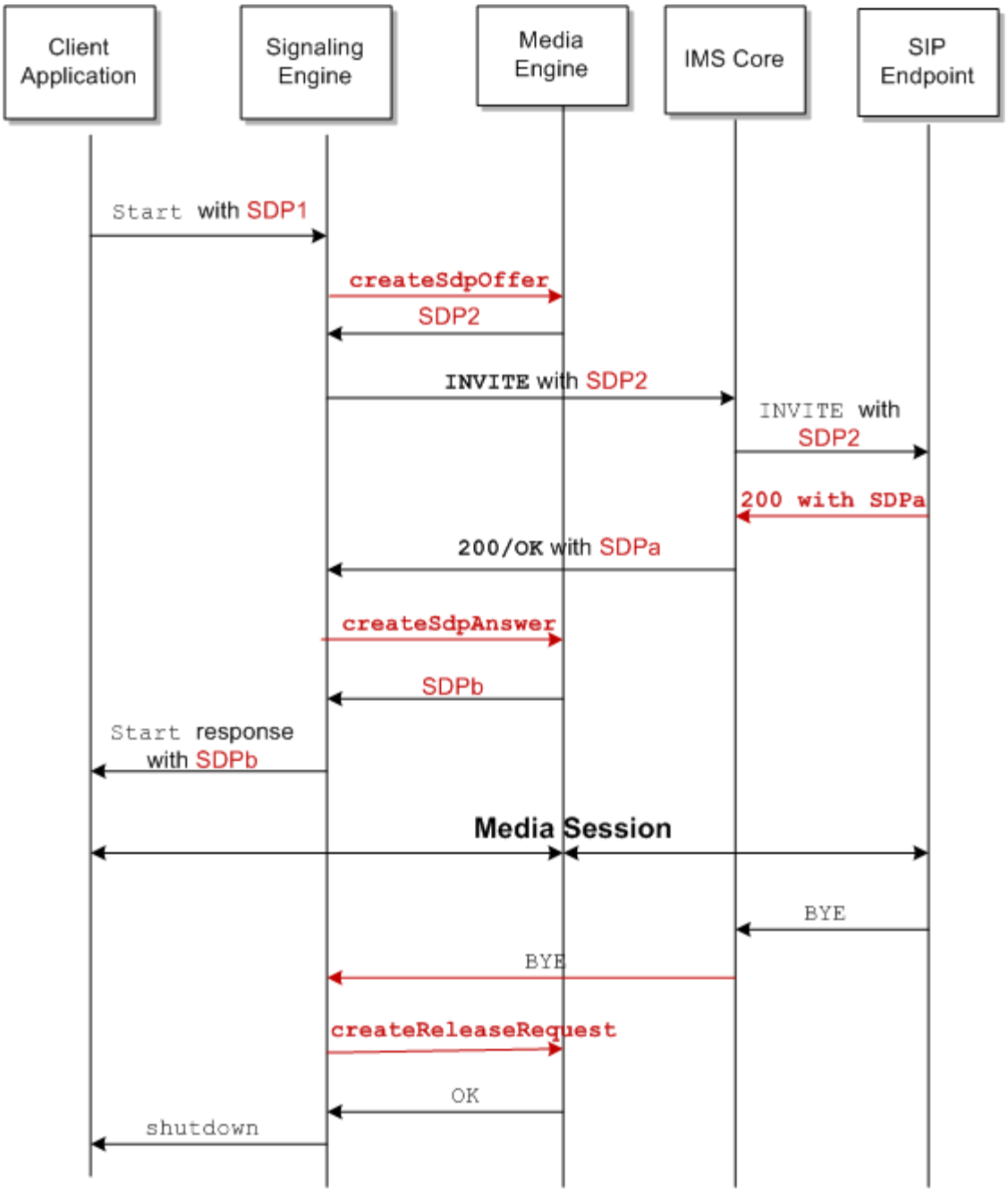
In the WebRTC Session Controller JsonRTC protocol, you use the **WscMediaFactory** interface in the **oracle.wsc.feature.webrtc.template.media** package to interact with Media Engine. It includes these methods:

- **createSdpOffer** - Can contain the media session ID, SDP data, **fromMediaConfigName**, and **toMediaConfigName** to use, and the From and To URLs to use for communication. See "About Media Engine Sessions" for details on the supported sessions.
- **createSdpAnswer** - Contains the media session ID and SDP data.
- **createReleaseRequest** - Contains the media session ID to release. This method releases the media or resources currently being used by the callee.
- **isAvailable** - Confirms that a Media Engine can be used. This is useful in cases where your Groovy script uses the Media Engine functionality if one is available, or does its own internal processing (attempts to connect the two client directly) if not.

See *WebRTC Session Controller Configuration API Reference* for details on this interface and these methods.

Figure 6–1 shows a flow of SDP data between two clients, in this case a WebRTC-enabled browser and a SIP endpoint. The two Signaling Engines may be different nodes in a clustered implementation, or they may be the same instance. This flow also shows where the **processSdpOffer**, **processSdpAnswer**, and **createReleaseRequest** actions occur.

Figure 6–1 Media Engine SDP Flow



In a typical scenario, Signaling Engine sends a **createSdpOffer** message to the Media Engine that includes all possible codecs that the caller supports. The Media Engine then returns modified SDP data including a list of the codecs that it supports and allows.

Further, the callee's SDP data, including a list of supported codecs, is sent from the SIP proxy to Signaling Engine in a 200/OK message, as shown in Figure 6–1. Signaling Engine then sends a **createSdpAnswer** to Media Engine with the list of codecs. If any codecs sent by Signaling Engine match the codecs supported by the Media Engine, the Media Engine returns the codecs it supports. Or, if Media Engine is configured to do so, it may attempt to convert the media stream to a alternate codec that the callee can use.

Once the media session has terminated, you send a **createReleaseRequest** message to the media server to release any resources the media server has allocated.

This code snippet from the Signaling Engine default **call** package, **FROM_NET/INVITE/request** criteria shows how to set up media anchoring:

```
if(Constants.ME_CONFIG_NAME_NET && sdpString!=null) {
  def sdpOffer = context.mediaFactory.createSdpOffer("1", sdpString, Constants.ME_
CONFIG_NAME_NET, null, sipAddressToString(sipRequest.to),
sipAddressToString(sipRequest.from));
  def ascFuture = sdpOffer.send()
  context.getTaskBuilder("processMediaResponseToSendWebMsg").withArg("ascFuture",
ascFuture).withArg("webMessage",webMessage).onSuccess(ascFuture).build();
}
else{
  webMessage.send()
}
```

This Groovy code tests whether a Media Engine is available, and if so sends a **createSdpOffer** request to the Media Engine with SDP data. If no Media Engine is available sends a **webMessage**.

This code snippet from the Script Library shows one example of handling a reply from Media Engine:

```
void processMediaResponseToSendWebMsg(TemplateContext context) {
  def resp = context.taskArgs.ascFuture.get();
  def newSdp = resp.getSdp();
  def webMessage = context.taskArgs.webMessage
  if (webMessage.payload) {
    webMessage.payload.sdp = newSdp
  } else {
    webMessage.payload = [sdp : newSdp]
  }
  webMessage.send()
}
```

It processes the response and sends the new SDP data back to the original caller.

About Media Engine Sessions

Table 6–1 lists the supported Media Engine session types, as assigned to the Groovy constant, **ME_CONFIG_NAME_DMA**, in the Groovy library, lists their Media Engine config names, and describes how they are used.

Table 6–1 Media Engine Session Types

Session Type	Config Name	Description
Web to Web Conditional Anchoring	**web-to-web-anchor-conditional**	Used when WebRTC-enabled browsers are allowed to communicate directly. If for some reason they cannot communicate directly, they can communicate through WebRTC Session Controller
Web to Web Forced Anchoring	**web-to-web-anchored**	Forces all media flows through Media Engine.

About Using createSdpOffer to Modify INVITE SDP Data

You use the **createSdpOffer** method to direct Media Engine to process SDP data sent by the calling end point. You either send SDP data with this method for Media Engine to process, or send the name of a media configuration that the node uses to determine for itself which SDP data to use. The Media Engine replies to Signaling Engine with the new or modified SDP data. Signaling Engine then uses the SDP data returned in the call's media session.

createSdpOffer includes these parameters:

- A set of SDP data to use.
- A media configuration name. The Media Engine uses the media configuration to select SDP data to return to the Signaling Engine. Media configuration names must be preconfigured on the Media Engine. See "About Media Engine Sessions" for details.
- A **fromURI**.
- A **toURI**.

You can send a **session_id** value with **createSdpOffer** to identify a specific media session. For example, **createSdpAnswer** requires a **session_id** to function.

You use the **send()** method from the **oracle.wsc.feature.webrtc.template** interface, **WscMessage** package to send **createSdpOffer**. See *WebRTC Session Controller Configuration API Reference* for details on **send()**.

About Using createSdpAnswer to Process 200 Message SDP Data

A SIP 200/OK message that accepts a session invitation contains SDP data to use in that session. You use the **createSdpAnswer** method in a Groovy script to accept and process that SDP.

About Using createReleaseRequest to Explicitly Release Media

All media sessions are released automatically when the call terminates. You can also force Media Engine to release all media for a session immediately by sending the session ID to the **createReleaseRequest** method in a Groovy script.

A

JSONRTC Protocol Reference

This appendix provides reference information for the WebRTC Session Controller JSONRTC Protocol used by WebRTC Session Controller Signaling Engine (Signaling Engine).

About the JSONRTC Protocol

WebRTC Session Controller uses this protocol to communicate with WebRTC-enabled browser client applications. It establishes the sessions and subsessions that you use to pass messages between WebRTC Session Controller and its client applications inside WebSocket connections.

You can also use this protocol to create new WebRTC Session Controller packages for your WebRTC Session Controller implementation.

See "About Building JSON to SIP Communication" for more information about how WebRTC Session Controller handles WebSocket connections, sessions, and subsessions.

While WebRTC Session Controller uses this protocol to communicate with JavaScript-based applications by default, this protocol also communicates with client applications based on different operating systems. Your client application opens the WebSockets necessary for the JSONRTC protocol subsessions to communicate with.

The JSON protocol operates between a WebRTC client and WebRTC server which can be an application server or a gateway. The WebRTC client can be a WebRTC-enabled browser (Chrome and Firefox), Android or iOS native applications.

Initiating a HTTP/HTTPS Handshake with Signaling Engine

The JSONRTC protocol is a sub protocol of the WebSocket protocol. You establish a handshake with a WebSocket protocol to initiate communication between the two. The handshake establishes a connection between the client (usually an application in a browser) and the Signaling Engine server inside HTTP/HTTPS. Once the client receives the handshake response, communication can proceed. The handshake is an HTTP `GET /chat` message using **webrtc.oracle.com** as the value for `Sec-WebSocket-Protocol`. For Example:

```
GET /chat HTTP/1.1
Host: server.wsc_IP.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://client_IP.com
Sec-WebSocket-Protocol: webrtc.oracle.com
Sec-WebSocket-Version: 13
```

Where:

wsc_IP is the domain name of the Signaling Engine server.

client_IP is the domain name of the client.

The handshake includes a `101 Switching Protocols` entry to allow the connection, as shown in this example handshake reply:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: webrtc.oracle.com
```

Once the client receives the handshake response, the client and Signaling Engine server can communicate further.

Immediately after establishing a WebSocket connection, the client sends a JSONRTC `connect` message to establish the WebRTC Session Controller JSONRTC session. Once WebRTC Signaling Controller accepts the `connect` message, it responds by sending back a **session_id**. If the WebSocket connection is broken unexpectedly, for example by a network problem, the client can re-establish the session by starting a new websocket connection with the original **session_id** in a `connect` message.

Closing a JSONRTC Session

You close a JSONRTC session by invoking the `session.close()` function.

About JSONRTC Sessions and SubSessions

See "About Sessions and Subsessions" for details on how this protocol establishes and manipulates sessions and subsessions.

JSONRTC uses a **session_id** field instead of a Message Broker WebSocket Subprotocol (MBWS) **connection_name** to identify the WebSocket session. The **session_id** field value must be unique across time and space to work with geographically redundant clusters.

The **subsession_id** is the **session_id** value with a **c** or **s** prefix added to it.

Also see "Initiating a HTTP/HTTPS Handshake with Signaling Engine" for more information about using **session_id** to reconnect a session.

About Message Reliability

This protocol uses the MessageBroker WebSocket Subprotocol (MBWS) as basis for message reliability. For more information on MBWS, see the MBWS specification:

`http://tools.ietf.org/html/draft-hapner-hybi-messagebroker-subprotocol-03`

About the JSONRTC Session Controller Messages

The basic communication unit used between a WebRTC-enabled client application and the WebRTC Session Controller JSONRTC protocol is a message. Signaling Engine communication can be synchronous or asynchronous.

About Messages

Each message executes a part of the whole message flow in a package. The **action** header defines the specific action each message carries out in a subsession in a specific package. For example, a "START" action will start a "call" or start a "message-notification". A SHUTDOWN action will end a "call" subsession or "messageNotification" subsession. The client and server do not make any assumptions of a particular **action** outside the defined behavior of the package.

All messages with **type=message** are asynchronous in nature and do not expect to receive a response message. Such messages are acknowledged. In case of an error, you receive an error message.

If you want to cancel a START message, send the CANCEL message before sending the START final response.

The CONNECT action is not associated with any package. CONNECT represents establishing a logical session. All other actions are specific to subsessions.

About Acknowledgements and Error Messages

An Acknowledgement message indicates that the message (and all the messages lower than the sequence) has reached the other side. An Error Message indicates that the message with the specified sequence met with an error.

Acknowledgement and error messages are applicable for requests, responses and messages. One side receives an acknowledgement with a sequence that is higher than a particular message and has not received an error message yet. This sequence indicates that the message sent has been received successfully by the other side. You can configure your applications to receive acknowledgement messages for every message.

About the Message Components

Each messages includes these components:

- Control Headers
- General Headers
- Message Payloads

This section also includes "Example Message Bodies" that you can use for reference.

Control Headers

The control header specifies information that the client and server use to handle (control) the message. It includes information required for WebSocket reconnect reliability, error, the message type, session ID, message state, and so on. Typically Signaling Engine uses this information itself, not applications or Groovy scripts.

type
The control type of JSON message. Can be one of:

request
A message, such as an offer message, that requires a response. A protocol frame with control type **request** may also contain a payload header.

response
This message is a response to a request message. For example, an answer or a provisional answer, pranswer in JavaScript Session Establishment Protocol (JSEP). A protocol frame with control type **request** may also contain a payload header.

message
A message that does not require a response. For example **notification**, or **publish**. A protocol frame with control type **request** may also contain a payload header.

acknowledgement
A message that acknowledges another message. Cannot contain a payload header.

error
Indicates that an error has arrived. Cannot contain a payload header.

package_type
Optional. The package is the type of service or functionality that the message handles and identifies the Signaling Engine package that the message applies to. The **register, call**, **flash**, **message_notification**, **capability**, **messaging**, **chat**, and **file_transfer** types are defined by default. If no **package_type** is specified, Signaling Engine assumes that the default **call** package is used for all messages except messages with a `connect` action. The `connect` action messages attempt to establish a session and are not associated with a package. See "Creating Packages" for details about the Signaling Engine packages.

session_id
Identifies a WebSocket session. The server creates the session ID and returns it to the client in the CONNECT response. A CONNECT message containing a session ID reestablishes a JSONRTC session. This value must be completely unique so that it may be used across redundant clusters. A session ID has the same role as the MessageBroker WebSocket Subprotocol (MBWS) **connection-name**.

sequence
A serial number that uniquely identifies a message in a JSONRTC session. Each side of the WebSocket connection maintains its own serial number counts, starting with 1.

Note the following exceptions for the **sequence** header:

- If a message is for WebSocket re-connection, the sequence number will not be set in the message.
- In an **acknowledge** message, this header indicates the specific message that came from the peer and does not indicate the serial number of this message.

ack_sequence

Optional. It identifies a particular message within a JSONRTC session. This header acknowledges that a specified server message has been received by the client or, likewise, that a specified client message has been received by the server. For example, if a client receives a message with **ack_sequence=2**, it means that the server has received the second message sent by the client. All messages with a lower value than specified by **ack_sequence** are also considered acknowledged.

If the sender has not received any message from its peer, the value of **ack_sequence** is **0**.

> **Note:** Do not set the **ack_sqeuence** header for an acknowledgement message. An acknowledgement message uses the sequence header to confirm the peer side.

subsession_id

Identifies a subsession within a session. The sequence numbers are incremented each time a new session is started. The client and server keep separate subsession ID counts. The subsession ID typically includes a **c** prefix if the subsession originated with the client and an **s** prefix if it originated with the server. An implementation can also choose to use a globally unique identifier as the subsession ID.

For example, the second client-originated subsession has the value **"subsession_ id":"c2"**. The seventh server-originated session uses the value **"subsession_id":"s7"**.

correlation_id

A string that identifies a specific message within a session. It can simply be a sequence number incremented each time a new message is sent. When the control header **type** is **response**, **acknowledge**, or **error**, the **correlation_id** associates it with the actual message.

The client and server keep separate message counts. When a client request does not have a **correlation_id**, server may assign one for response. Messages from the client have a "**c**" suffix and messages from the server have an "**s**" suffix. For example, the third client-originated message has the value **"correlation_id":"c3"**. The sixth server-originated message has the value **"correlation_id":"s6"**.

When a client request does not have a **correlation_id**, the server assigns one for its response.

message_state

Identifies the message state as **subsequent**, or **final**. Only subsequent or final response messages need specify the message_state.

For example: **"message_state":"final"**

version

Identifies the JSONRTC protocol version that message sender supports (client or server). If none is present in the message version, it is assumed to be **"3.0"**.

General Headers

The general header contains information related to the specific action involved in the message. For example, for a START request, such information would contain who initiated the request, for whom it is intended, and so on.

The general header includes fields that Signaling Engine uses to build up and tear down calls. These fields are specific to one or more packages and are available to use in both client applications and Groovy scripts. Your application can add additional headers to this section. Such headers may be mapped by a gateway server to a SIP header or a parameter.

action
The purpose of the message. Can be one of:

connect
Establishes a session with the server. These general headers are only used with the CONNECT action.

cslr
Optional. Sent with the **session_id** of a session to reconnect. Uses the sequence number of the last message received from the client to identify the session.

cslw
Optional. Sent with the **session_id** of a session to reconnect. Uses the lower bound of the messages in the client's retained window to identify the session.

csuw
Optional. Sent with the **session_id** of a session to reconnect. Uses the upper bound of the messages in the client's retained window to identify the session.

sslr
Optional. Sent with the **session_id** of a session to reconnect. Uses the sequence number of the last message received by the server to identify the session.

start
Starts a session with a specific package.

complete
Announces that the media session has been established.

hibernate
Announces that the session is going to hibernate. Hibernates a protocol level session with the server.

notify
Equivalent to Notification of Notification Server.

shutdown
Shuts down a session opened by a specific request.

prack
Pre-acknowledges provisional responses.

enquiry
Queries information (about capabilities) from the peer side.

send
Sends out data.

trickle
Sends out ICE candidates. Trickle SDP candidate information from the peer.

initiator
Optional. Identifies the URI of the user initiating the HTTP request. If this value exists, it may be set by the client or the HTTP session. In certain cases, a value may not even exist (such as when a random user clicks on a web page to talk with customer care).

target
Optional. Identifies the URI of the Signaling Engine server that is targeted by the message. Can be obtained from the HTTP session.

error_code
Optional. In error type messages, lists the error message.

reason
Optional. The description of the error.

response_code
Optional. Specifies the result for a response message, especially in the `call` package. For example, "180", "200", and so on.

enquiry_data
Optional. This is the enquiry data that is the result of an `enquiry` action. In most cases it is for the `capability` package.

wsc_id
Optional. Used in a re-connect response message only. It identifies the server id with which WebSocket is connecting.

Package-Specific General Headers

Some headers are specified only for certain packages.

message_notification, expiry (xp)
Optional. Represents the subscription's expiration time for receiving message-summary notifications.

Authentication-Specific General Headers

The authentication headers are specific to client authentication and authorization.

authenticate
Optional. Represents authentication information from the server. If the server leverages DIGEST for authentication, this header contains {`scheme`, `username`, `realm`, `qop`, `opaque`, `nonce`, `cnonce`, `ha1`, `challenge_code`, `algorithm`}.

authorization
Optional. This header represents the authorization response to the server. If DIGEST is leveraged, this header contains {`scheme, username, realm, qop, opaque, nonce, cnonce, ha1, challenge_code, algorithm`}. For more details on this header, see `https://www.ietf.org/rfc/rfc2617.txt`.

The `ha1` value is calculated with the following steps:

1. A1 calculation:

- If `algorithm`=MD5 or is unspecified

  ```
  A1 = username-value ":" realm-value ":" password
  ```

- If algorithm=MD5-sess

  ```
  A1 = H(username-value ":" realm-value ":" password) ":" nonce-value ":"
  cnonce-value
  ```

2. `ha1`=H(A1)

Where `username-value`, `realm-value`, `nonce-value`, and `cnonce-value` are strings without quote marks. H means the string obtained by applying the checksum algorithm to A1.

Message Payloads

The message payload is specific to the Signaling Engine package for which the message is used.

For:

- The **call**, **chat,** and **file_transfer** packages, the default payload is an SDP offer or answer.
- The **messaging** package, the payload is content that represents the text message.
- The **message-notification** or a **register** with the **hibernate** action, the payload is JSON data with the exact message alerts.

Providing Client Information as a Payload

The mechanism to register with the Cloud Notification system is entirely up to your (Android or iOS) application and should follow the Android or iOS guidelines or the Customer's own notification registration APIs. For more information on the registration step, see the Android or iOS reference documents, as appropriate.

The WSC Client SDK captures the information required to identify the client and its capability as shown in the following table:

Table A–1 Client Capability Identification Data

Parameter	Value	Description
family	android/iOS/Chrome/Firefox	Device family
version	41/37/21/8.1	Current version of the Client
appid	com.enterprise.*enterpriseId*.wsc	Unique application Id. For example, com.enterprise.*xyz*.wsc
appversion	3.1	Application version

When the SDK sends it initial registration request, it sends this client identification information to the WSC server as part of that **register** request.

Here is an example used by a sample Android application:

Example A–1 Payload Data Sent in the Initial Register Request (Android)

```
{
  "control": {
    "type": "request",
    "package_type": "register",
    "sequence": 1,
    "version": "1.0"
  },
  "header": {
    "action": "connect"
  },
  "payload": {
    "capability": {
      "family": "android",
      "version": 21,
      "appid": "com.enterprise.xyz.wsc"
      "appversion": "3.1"
    },
    "devicetoken": "APA91bFlmPxDGNWp42wCJE8_r09YECG-dEWtzNU1DXACl1IaqFSJdOlxCvO_
4K-mkiQO6CS-jVnVxDVXiCQwK5F0dosPMa2bZpiBc6vo";
  }
}
```

As shown in Example A–1, the device token identifying the client device is sent. This value stored as part of the Session data in the Cluster state and is used later for sending notifications.

Notification Payloads

The notification sent to the client mobile device contains user parameters and is limited by the cloud notification system.

Add custom data in JSON format. For example:

```
 "data": {
    "time": "15:16.2342",
    "message": "Incoming Call"
  }
```

WebRTC Session Controller server converts the message data to suit the cloud messaging system. For the Google cloud messaging system (GCM), this notification contains the registration id for the device as shown here:

Example A–2 Notification Payload (for GCM)

```
{
  "collapse_key": "wsc_notify",
  "time_to_live": 300,
  "delay_while_idle": true,
  "data": {
    "time": "15:16.2342",
    "message": "Incoming Call"
  },
  "registration_ids":["APA91bFlmPxDGNWp42wCJE8_
r09YECG-dEWtzNU1DXACl1IaqFSJdOlxCvO_4K-mkiQO6CS"]
}
```

Note: The additional custom data is passed along without any changes.

Example Message Bodies

The following sections show message body examples.

Connect Request Message

```
{
   "control": {
    "type":"request",
    "sequence":"1"
   },
   "header": {
    "action":"connect",
    "initator":"bob@example.com",
   }
}
```

CONNECT Response Message

```
{
   "control": {
    "type":"response",
    "sequence":"1",
    "correlation_id":"c1",
    "subsession_id":"c1",
    "session_id":"Hyi89JUThhjjR"
   },
   "header": {
    "action":"connect"
   }
}
```

CONNECT Request with Device Token

```
{
   "control":{
    "type":"request",
    "package_type":"register",
    "sequence":1,
    "version":"3.0"
   },

 "header":{
    "action":"connect",
    "initiator":"alice@example.com",
    "target":"alice@example.com"
   },

  "payload":{
   "capability":{
    "appid":"oracle.wsc.samples.web",
    "appversion":"0.1",
    "family":"Chrome",
    "version":"45.0.2454.101"
   },

"devicetoken":"APA91bHIx2iEtOjulPrVQxDQPIwwHscTXRpkzTJZJoYr7ajWH0XPglQ10Y8J7pNn3Sh
8RnchKNno-BhmfvJqrO_8EFx-AGilpG8YlV9wbI2C9OlOmw5jAstOtpxkuj0IMBHViX4y3uQaRAqFB_
```

```
YvprHBHqRzBTZ6hvqaDwN1OXiyANk-LYTIVWXKep-Hp03K-5VpFEY3zbBg"
  }
}
```

START Request Message (with initiator/target)

```
{
   "control": {
    "package_type": "call",
    "type":"request",
    "sequence":"2",
   },
   "header": {
    "action":"start",
    "initator":"bob@example.com",
    "target":"alice@example.com",
   },
   "payload": {
     "<offer_sdp>"
   }
}
```

START Response Message (with initiator/target)

```
{
   "control": {
    "package_type": "call",
    "type":"response",
    "message_state":"final",
    "sequence":"2",
    "correlation_id":"c2"
    "subsession_id":"c2"
   },
   "header": {
    "action":"start"
    "initator":"bob@att.com",
    "target":"alice@att.com",
   },
   "payload": {
     "<answer_sdp>"
   }
}
```

START Request Message (without initiator/target)

```
{
"control": {
    "package_type": "call",
    "type":"request",
    "sequence":"2"
   },
   "header": {
    "action":"start"
   },
   "payload": {
     "<offer_sdp>"
   }
}
```

START Request Message (without initiator/target)

```
{
   "control": {
    "package_type": "call",
    "type":"response"
    "message_state":"final"
    "sequence":"2",
    "correlation_id":"c2"
    "subsession_id":"c2"
   },
   "header": {
    "action":"start"
   },
   "payload": {
     "<pranswer_sdp>"
   }
}
```

START Offer with Changed Media

```
{
"control": {
    "package_type": "call",
    "type":"request",
    "sequence":"3",
    "subsession_id":"c2"
   },
   "header": {
    "action":"start"
   },
   "payload": {}
     "<offer_sdp>"
   }
}
```

HIBERNATE Request Message

```
{
"control": {
        "type": "request",
        "package_type": "register",
        "session_id": "pQAAAVBltniS54z1CuCiXp9AffIAAAAD_23",
        "subsession_id": "50d890f2-d357-4847-9278-442d5964fc32",
        "correlation_id": "c2",
        "version": "1.0"
    },
    "header": {
        "action": "hibernate",
        "ttl": 3600
    },
    "payload": {}
}
```

HIBERNATE Response Message

```
{
"header": {
        "response_code": 200,
        "action": "hibernate"
```

```
    },
    "control": {
        "sequence": 2,
        "subsession_id": "50d890f2-d357-4847-9278-442d5964fc32",
        "message_state": "final",
        "session_id": "pQAAAVBltniS54z1CuCiXp9AffIAAAAD_23",
        "correlation_id": "c2",
        "type": "response",
        "package_type": "register",
        "version": "1.0"
    }
}
```

Note that:

- The **response_code** indicates if the other side accepted or rejected the request.
- The **ttl** parameter in the Hibernate response contains the value the WSC server finally chose for the Session-Alive interval. The WSC server maintains a maximum interval depending on the policy set for each type of client device.

  If the requested value was more than the accepted maximum for the server, the value will revert to the server's maximum value. If the value is too lower than the default value for the server, the server reverts to its default value.

SHUTDOWN Message

```
{
   "control": {
    "package_type": "call",
    "type":"message"
    "sequence":"4",
    "subsession_id":"c2"
   },
   "header": {
    "action":"shutdown"
   }
}
```

ACKNOWLEGEMENT Message

```
{
"control": {
    "type":"acknowledgement"
    "sequence":"5"
   }
}
```

ERROR Message

```
{
   "control": {
    "package_type": "call"
    "type":"error"
    "sequence":"6",
    "correlation_id":"c2"
    "subsession_id":"c2"
    "error_code":"480"
   }
}
```

ENQUIRY Request Message

```
{
    "package_type": "capability"
    "type":"request",
    "sequence":"1",
    "correlation_id":"c1"
    "subsession_id":"c1"
  }
  "header": {
   "action":"enquiry",
   "enquiry_data":"IM/CHAT,VS",
   "initiator":"bob@att.com",
   "target":"alice@att.com",
  }
}
```

Enquiry Response Message

```
{
   "control": {
    "package_type": "capability",
    "type":"response",
    "message_state":"final",
    "sequence":"1",
    "correlation_id":"c1"
    "subsession_id":"c1"
   }
   "header": {
    "action":"enquiry",
    "enquiry_data""IM/CHAT, FT",
    "initator":"bob@att.com",
    "target":"alice@att.com",
   }
}
```

SEND Message

```
{
   "control": {
    "package_type": "messaging"
    "type":"message"
    "sequence":"1",
    "correlation_id":"c1"
    "subsession_id":"c1"
   }
   "header": {
    "action":"send"
    "initiator":"bob@att.com",
    "target":"alice@att.com",
   }
   "payload": {
    "content": <message content>
   }
}
```

TRICKLE Message

```
{
    "control": {
```

```
        "type": "message",
        "package_type": "call",
        "session_id": "pQAAAVBlDjliO2V9HrgoM9QDd4EAAAAF_255",
        "subsession_id": "9356a8a5-8b48-4b8d-9355-0c43315114d3",
        "sequence": 4,
        "ack_sequence": 3,
        "version": "3.0"
    },
    "header": {
        "action": "trickle",
        "initiator": "alice@example.com",
        "target": "bob@example.com"
    },
    "payload": {
        "candidates": "a=mid:audio\r\na=candidate:134500521 1 ..."
    }
}
```