KurentoJsonRpc Documentation

Release 6.6.1-dev

kurento.org

Sep 21, 2017

Contents


KURENTO

Part I

Description

This document describes the implementation of the *JSON-RPC* client and server in the Kurento project. A detailed introduction to the *WebSocket* protocol is beyond the scope of this document. At a minimum, however, it is important to understand that HTTP is used only for the initial handshake, which relies on a mechanism built into *HTTP*, to request a protocol upgrade (or in this case a protocol switch) to which the server can respond with HTTP status 101 (switching protocols) if it agrees. Assuming the handshake succeeds the *TCP* socket underlying the HTTP upgrade request remains open, and both client and server can use it to send messages to each other. For information about the protocol itself, please refer to this page in the project's documentation.

As the *JSON-RPC* v2.0 specification describes, the protocol implies the existence of a client issuing requests, and the presence of a server to process those requests. This comes in opposition with v1.0, which used a peer-to-peer architecture, where both peers were client and server.

Part II

Code structure

Kurento has implemented a JSON-RPC server in Java and a JSON-RPC client in Java and another in Javascript. All implementations are hosted on github:

- **Java** - https://github.com/Kurento/kurento-java/tree/master/kurento-jsonrpc
- **Javascript** - https://github.com/Kurento/kurento-jsonrpc-js

The Java implementation contains a Maven project with the following modules:

- kurento-java - reactor project
- kurento-java/kurento-jsonrpc/kurento-jsonrpc-server - Kurento's own implementation of a JSON-RPC server.
- kurento-java/kurento-jsonrpc/kurento-jsonrpc-client - Java client of the kurento-jsonrpc-server, or any other websocket server that implements the JSON-RPC protocol.
- kurento-java/kurento-jsonrpc/kurento-jsonrpc-demo-server - It is a demo application of the Kurento JsonRpc Server library. It consists of a WebSocket server that includes several test handlers of JsonRpc messages.

The Javascript implementation contains:

- kurento-jsonrpc-js - Javascript client of the kurento-jsonrpc-server, or any other websocket server that implements the JSON-RPC protocol. This library minified is available here.

Part III

Json-Rpc Server

This is a *JAVA* implementation of a *JSON-RPC* server. It supports v2.0 only, which implies that Notifications can be used. However, the only possible transport is Websockets. It is published as a Maven artifact, allowing developers to easily manage it as a dependency, by including the following dependency in their project's pom:

```
<dependency>
    <groupId>org.kurento</groupId>
    <artifactId>kurento-jsonrpc-server</artifactId>
    <version>6.6.1-SNAPSHOT</version>
</dependency>
```

The server is based on *Spring Boot* 1.3.0.RELEASE. The usage is very simple, and analogous to the creation and configuration of a WebSocketHandler from Spring. It is basically composed of the server's configuration, and a class that implements the handler for the requests received. The following code implements a handler for *JSON-RPC* requests, that contains a JsonObject as params data type. This handler will send back the params received to the client. Since the request handling always sends back a response, the library will send an automatic empty response if the programmer does not purposefully do so. In the following example, if the request does not invoke the echo method, it will send back an empty response:

```
import org.kurento.jsonrpc.DefaultJsonRpcHandler;
import org.kurento.jsonrpc.Transaction;
import org.kurento.jsonrpc.message.Request;

import com.google.gson.JsonObject;

public class EchoJsonRpcHandler extends DefaultJsonRpcHandler<JsonObject> {

    @Override
    public void handleRequest(Transaction transaction,
            Request<JsonObject> request) throws Exception {
        if ("echo".equalsIgnoreCase(request.getMethod())) {
            transaction.sendResponse(request.getParams());
        }
    }
}
```

The first argument of the method is the Transaction, which represents a message exchange between a client and the server. The methods available in this object (overloads not included), and it's different uses are:

- **sendResponse**: sends a response back to the client.
- **sendError**: sends an Error back to the client.
- **getSession**: returns the JSON-RPC session assigned to the client.
- **startAsync**: in case the programmer wants to answer the Request outside of the call to the handleRequest method, he can make use of this method to signal the server to not answer just yet. This can be used when the request requires a long time to process, and the server not be locked.
- **isNotification**: evaluates whether the message received is a notification, in which case it mustn't be answered.

Inside the method *handleRequest*, the developer can access any of the fields from a JSON-RPC Request (*method*, *params*, *id* or *jsonrpc*). This is where the methods invoked should be managed. Besides the methods processed in this class, the server handles also the following special *method* values:

- **close**: The client send this method when gracefully closing the connection. This allows the server to close connections and release resources.
- **reconnect**: A client that has been disconnected, can issue this message to be attached to an existing session. The sessionId is a mandatory param.
- **ping**: simple ping-pong message exchange to provide heartbeat mechanism.

The class *DefaultJsonRpcHandler* is generified with the payload that comes with the request. In the previous code, the payload expected is a JsonObject, but it could also be a plain String, or any other object.

To configure a WebSocket-based JSON-RPC server to use this handler, developers can use the dedicated JsonRpcConfiguration, for mapping the above websocket handler to a specific URL (http://localhost:8080/echo in this case):

```
import org.kurento.jsonrpc.internal.server.config.JsonRpcConfiguration;
import org.kurento.jsonrpc.server.JsonRpcConfigurer;
import org.kurento.jsonrpc.server.JsonRpcHandlerRegistry;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Import;

@Import(JsonRpcConfiguration.class)
public class EchoServerApplication implements JsonRpcConfigurer {

    @Override
    public void registerJsonRpcHandlers(JsonRpcHandlerRegistry registry) {
        registry.addHandler(new EchoJsonRpcHandler(), "/echo"); // "/echo" is the␣
↪path relative to the server's URL
    }

}
```

CHAPTER 1

Session control

Each client connecting to this server, will be assigned a unique sessionId. This provides a session concept, that can expand through several websocket sessions. Having this notion of JSON-RPC session, allows to bind a set of properties to one particular session. This gives the developers implementing a server the capability of having a stateful server session, which the user can recover once reconnected. The methods available in this object are

- **getSessionId**: The ID assigned to this session. It can be used to track down the session, and register it in servers and map it to other resources.
- **getRegisterInfo**: This is set by the client upon connection, and it is accessible by the server through this method.
- **isNew**: will be true if the message is the first message of the session.
- **close**: gracefully closes the connection.
- **setReconnectionTimeout**: sets the time that the server will wait for a reconnection, before closing the session.
- **getAttributes**: returns that attribute map from the session

CHAPTER 2

Handlers

Advanced properties

When registering a particular handler, there are a number of properties that can be configured. These are accessed from a fluent API in the DefaultJsonRpcHandler

- **withSockJS()** - Enables SockJS as WS library, which provides a fallback to HTTP if the upgrade fails. The client should be a SockJS capable client. There's more info here.
- **withLabel(String)** - Adds a label that is used when requests are handled. This allows having a friendly name in the log files, to track executions more easily.
- **withPingWatchdog(true|false)** - The ping watchdog is a functionality that monitors the health of the heartbeat mechanism, allowing to detect when a regular ping message is not received in the expected time. This informs the server that, though the websocket connection might still be open, the client on the other side is not working as expected.
- **withAllowedOrigins(String[])** - By default, only clients connecting from the same origin (host and port) as the application is served are allowed, limiting the clustering and load-balancing capabilities. This method takes an array of strings with the allowed origins. The official Spring-Boot documentation offers details about how this works.

Reacting to connection events

The handler offers the possibility to override some methods related to connection events. The methods available are:

```
import org.kurento.jsonrpc.DefaultJsonRpcHandler;
import com.google.gson.JsonObject;

public class EchoJsonRpcHandler extends DefaultJsonRpcHandler<JsonObject> {

    // ...
```

```
    @Override
    public void afterConnectionEstablished(Session session) throws Exception {
        // Do something useful here
    }

    @Override
    public void afterConnectionClosed(Session session, String status)
            throws Exception {
        // Do something useful here
    }

    @Override
    public void handleTransportError(Session session, Throwable exception)
            throws Exception {
        // Do something useful here
    }

    @Override
    public void handleUncaughtException(Session session, Exception exception) {
        // Do something useful here
    }
}
```

CHAPTER 3

Notifications

A Notification is a Request object without an "id" member. A *Request* object that is a Notification signifies the sender's lack of interest in the corresponding *Response* object, and as such no *Response* object needs to be returned.

Notifications are not confirmable by definition, since they do not have a Response object to be returned. As such, the sender would not be aware of any errors (like e.g. "Invalid params","Internal error")

The server is able to send notifications to connected clients using their ongoing *session* objects. For this purpose, it is needed to store the *Session* object of each client upon connection. This can be achieved by overriding the *afterConnectionEstablished* method of the handler

```
public class EchoJsonRpcHandler extends DefaultJsonRpcHandler<JsonObject> {

    public final Map<String, Session> sessions = new HashMap<>();

    @Override
    public void afterConnectionEstablished(Session session) {
        String clientId = (String) session.getAttributes().get("clientId");
        sessions.put(clientId, session);
    }

    @Override
    public void afterConnectionClosed(Session session, String status)
         throws Exception {
        String clientId = (String) session.getAttributes().get("clientId");
        sessions.remove(clientId);
    }

    // Other methods
}
```

How a session is paired with each client is something that depends on the business logic of the appllication. In this case, we are assuming that the session holds a *clientId* property, that can be used to uniquely identify each client. It is also possible to use the *sessionId*, a :term:UUID provided by the library as session identifier, but they are not meaningful for the application using the library. It is advisable to not leave sessions registered once clients disconnect, so we are overriding the *afterConnectionClosed* method and removing the stored *session* object there.

Notifications are sent to connected clients through their stablished session. Again, how to map sessions to clients in particular is out of the scope of this document, as it depends on the business logic of the application. Assuming that the *handler* object is in the same scope, the following snippet shows how a notification to a particular client would be sent

```
public void sendNotification(String clientId, String method, Object params)
    throws IOException {
  handler.sessions.get(clientId).sendNotification(method, params);
}
```

CHAPTER 4

JavaDoc

- kurento-jsonrpc-server

Part IV

Json-Rpc Client

This is the Java client of the kurento-jsonrpc-server, or any other websocket server that implements the *JSON-RPC* protocol. It allows a Java program to make JSON-RPC calls to the kurento-jsonrpc-server. Is also published as a Maven dependency, to be added to the project's pom:

```
<dependency>
    <groupId>org.kurento</groupId>
    <artifactId>kurento-jsonrpc-server</artifactId>
    <version>6.6.1-SNAPSHOT</version>
</dependency>
```

CHAPTER 5

Creating a client

Contrary to the server, the client is framework-agnostic, so it can be used in regular Java applications, Java EE, Spring… Creating a client that will send requests to a certain server is very straightforward. The URI of the server is passed to the *JsonRpcClientWebSocket* in the constructor, here assuming that it is deployed in the same machine:

```
JsonRpcClient client = new JsonRpcClientWebSocket("ws://localhost:8080/echo");
```

CHAPTER 6

Sending requests

A JSON-RPC call is represented by sending a Request object to a Server. Such object has the following members

- **jsonrpc**: a string specifying the version of the JSON-RPC protocol, "2.0" in this case
- **method**: A String containing the name of the method to be invoked
- **params**: A Structured value that holds the parameter values to be used during the invocation of the method. This member may be omitted, and the type comes defined by the server
- **id**: An identifier established by the Client. If it is not included it is assumed to be a notification. The Server replies with the same value in the Response object if included. This member is used to correlate the context between the two objects.

From all these members, users only have to set the "method" and the "params", as the other two are managed by the library.

The server defined in the previous section expects a JsonObject, and answers to the *echo* method only, bouncing back the "params" in the request. It is expected that the response to *client.sendRequest(request)* will be the wrapped *params* in the *Response<JsonElement>* object that the Server sends back to the client:

```
Request<JsonObject> request = new Request<>();
request.setMethod("echo");
JsonObject params = new JsonObject();
params.addProperty("some property", "Some Value");
request.setParams(params);
Response<JsonElement> response = client.sendRequest(request);
```

Other messages: notifications

A Notification is a Request object without an "id" member. A Request object that is a Notification signifies the Client's lack of interest in the corresponding Response object, and as such no Response object needs to be returned to the client. Notifications are not confirmable by definition, since they do not have a Response object to be returned. As such, the Client would not be aware of any errors (like e.g. "Invalid params","Internal error"):

```
client.sendNotification("echo");
```

Server responses

When the Server receives a rpc call, it will answer with a Response, except in the case of Notifications. The Response is expressed as a single JSON Object, with the following members:

- **jsonrpc**: a string specifying the version of the JSON-RPC protocol, "2.0" in this case
- **result**: this member exists only in case of success. The value is determined by the method invoked on the Server.
- **error** this member exists only in there was an error triggered during invocation. The type is an Error Object
- **id**: This is a required member, that must match the value of the id member in the Request.

Responses will have either "result" or "error" member, but not both.

Error objects

When a rpc call encounters an error, the Response Object contains the error member with a value that is a Object with the following members:

- **code**: A number that indicates the error type
- **message**: a short description of the error
- **data**: A Primitive or Structured value that contains additional information about the error. This may be omitted, and is defined by the Server (e.g. detailed error information, nested errors etc.).

CHAPTER 7

Adding connection listeners

The client offers the possibility to set-up a listener for certain connection events. A user can define a **JsonRpcWSConnectionListener** that offers overrides of certain methods. Once the connection listener is defined, it can be passed in the constructor of the client, and the client will invoke the methods once the corresponding events are produced:

```
JsonRpcWSConnectionListener listener = new JsonRpcWSConnectionListener() {

    @Override
    public void reconnected(boolean sameServer) {
        // ...
    }

    @Override
    public void disconnected() {
        // ...
    }

    @Override
    public void connectionFailed() {
        // ...
    }

    @Override
    public void connected() {
        // ...
    }
} ;
JsonRpcClient client = new JsonRpcClientWebSocket("ws://localhost:8080/echo",␣
↪listener);
```

CHAPTER 8

Managing heartbeat

As pointed out in the server, there is a heartbeat mechanism that consists in sending ping messages in regular intervals. This can controlled in the client thought the following methods:

- **enableHeartbeat**: this enables the heartbeat mechanism. The default interval is 5s, but this can be changed through the overload of this method, that receives a number as parameter.
- **disableHeartbeat**: stops the regular send of ping messages.

CHAPTER 9

Changing default timeouts

Not only the ping message interval is configurable. Other configurable timeouts are:

- **Connection timeout**: This is the time waiting for the connection to be established when the client connect to the server.
- **Idle timeout**: If no message is sent during a certain period, the connection is considered idle and closed.
- **Request timeout**: the server should answer the request under a certain response time. If the message is not answered in that time, the request is assumed not to be received by the server, and the client yields a TransportException

CHAPTER 10

JavaDoc

- kurento-jsonrpc-client

Part V

Json-Rpc Client JS

This is the Javascript client of the kurento-jsonrpc-server, or any other websocket server that implements the *JSON-RPC* protocol. It allows a Javascript program to make JSON-RPC calls to any jsonrpc-server. Is also published as a bower dependency.

CHAPTER 11

JsonRpcClient

Create client

For creating a client that will send requests, you need to create a configuration object like in the next example:

```
var configuration = {
      hearbeat: 5000,
      sendCloseMessage : false,
      ws : {
        uri : ws_uri,
        useSockJS: false,
        onconnected : connectCallback,
        ondisconnect : disconnectCallback,
        onreconnecting : disconnectCallback,
        onreconnected : connectCallback
      },
      rpc : {
        requestTimeout : 15000,
        treeStopped : treeStopped,
        iceCandidate : remoteOnIceCandidate,
      }
    };

var jsonRpcClientWs = new JsonRpcClient(configuration);
```

This configuration object has several options: in one hand, the configuration about transport on the other hand the configuration about methods that the client has to call when get a response. Also, it can configure the interval for each heartbeat and if you want send a message before closing the connection.

- **Configuration**

```
{
    heartbeat: interval in ms for each heartbeat message,
    sendCloseMessage: true / false, before closing the connection, it sends a close_
↪session message,
```

```
    ws: {
        uri: URItoconntectto,
        useSockJS: true(useSockJS)/false(useWebSocket)bydefault,
        onconnected: callback method to invoke when connection is successful,
        ondisconnect: callback method to invoke when the connection is lost,
        onreconnecting: callback method to invoke when the client is reconnecting,
        onreconnected: callback method to invoke when the client succesfully␣
↪reconnects
    },
    rpc: {
        requestTimeout: timeoutforarequest,
        sessionStatusChanged: callback method for changes in session status,
        mediaRenegotiation: mediaRenegotiation
        ...
    [Other methods you can add on rpc field are:
treeStopped : treeStopped
   iceCandidate : remoteOnIceCandidate]
    }
}
```

If heartbeat is defined, each x milliseconds the client sends a ping to the server for keeping the connection.

CHAPTER 12

Sending requests

A JSON-RPC call is represented by sending a Request object to a Server using send method. Such object has the following members:

- **method**: A String containing the name of the method to be invoked
- **params**: A Structured value that holds the parameter values to be used during the invocation of the method. This member may be omitted, and the type comes defined by the server. It is a json object.
- **callback**: A method with error and response. This method is called when the request is ended.

```
var params = {
             interval: 5000
              };

jsonrpcClient.send("ping", params , function(error, response){
         if(error) {
            ...
         } else {
            ...
         }
      });
```

Server responses

When the Server receives a rpc call, it will answer with a Response, except in the case of Notifications. The Response is expressed as a single JSON Object, with the following members:

- **jsonrpc**: a string specifying the version of the JSON-RPC protocol, "2.0" in this case
- **result**: this member exists only in case of success. The value is determined by the method invoked on the Server.
- **error**: this member exists only in there was an error triggered during invocation. The type is an Error Object
- **id**: This is a required member, that must match the value of the id member in the Request.

Responses will have either “result” or “error” member, but not both.

Error objects

When a rpc call encounters an error, the Response Object contains the error member with a value that is a Object with the following members:

- **code**: A number that indicates the error type
- **message**: a short description of the error
- **data**: A Primitive or Structured value that contains additional information about the error. This may be omitted, and is defined by the Server (e.g. detailed error information, nested errors etc.).

Other methods

- **close**: Closing jsonRpcClient explicitly by client.
- **reconnect**: Trying to reconnect the connection.
- **forceClose**: It used for testing, forcing close the connection.

CHAPTER 13

WebSocket With Reconnection

This jsonrpc client uses an implementation of websocket with reconnection. This implementation allows the connection always alive.

It is based on states and calls methods when any of next situation happens:

- **onConnected**
- **onDisconnected**
- **onReconnecting**
- **onReconnected**

It has a configuration object like next example and this object is part of jsonrpc client's configuration object.

```
{
   uri: URItoconntectto,
   useSockJS: true(useSockJS)/false(useWebSocket)bydefault,
   onconnected: callback method to invoke when connection is successful,
   ondisconnect: callback method to invoke when the connection is lost,
   onreconnecting: callback method to invoke when the client is reconnecting,
   onreconnected: callback method to invoke when the client succesfully reconnects
}
```

Part VI

Securing JSON-RPC connections

From Chrome M47, requests to *getUserMedia* are only allowed from secure origins (HTTPS or HTTP from localhost). Since Kurento relies heavily on the JSON-RPC library for the signaling part of applications, it is required that the JSON-RPC server offers a secure websocket connection (*WSS*), or the client will receive a *mixed content* error, as insecure WS connections may not be initialised from a secure HTTPS connection.

CHAPTER 14

Securing JSON-RPC Servers

Enabling secure Websocket connections is fairly easy in Spring. The only requirement is to have a certificate, either self-signed or issued by a certification authority. The certificate must be stored in a , so it can be later used by the :term:JVM. Depending on whether you have acquired a certificate or want to generate your own, you will need to perform different operations

- Certificates issued by certification authorities can be imported with the command:

```
keytool -importcert -file certificate.cer -keystore keystore.jks -alias
↪"Alias"
```

- A keystore holding a self-signed certificate can be generated with the following command:

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -
↪storepass password -validity 360 -keysize 2048
```

The file *keystore.jks* must be located the project's root path, and a file named application.properties must exist in `src/main/resources/`, with the following content:

```
server.port: 8443
server.ssl.key-store: keystore.jks
server.ssl.key-store-password: yourPassword
server.ssl.keyStoreType: JKS
server.ssl.keyAlias: yourKeyAlias
```

You can also specify the location of the properties file. Just issue the flag `-Dspring.config.location=<path-to-properties>` when launching the Spring-Boot based app. In order to change the location of the *keystore.jks* file, it is enough to change the key `server.ssl.key-store`. The complete official documentation form the Spring project can be found here

CHAPTER 15

Connecting JSON-RPC Clients to secure servers

JSON-RPC clients can connect to servers exposing a seure connection. By default, the Websocket library used will try to validate the certificate used by the server. In case of self-signed certificates, the client must be instructed to prevent skip this validation step. This can be acchieved by creating a `SslContextFactory`, and using the factory in the client.

```
SslContextFactory contextFactory = new SslContextFactory();
contextFactory.setValidateCerts(false);

JsonRpcClientWebSocket client = new JsonRpcClientWebSocket(uri, contextFactory);
```

Part VII

Glossary

This is a glossary of terms that often appear in discussion about multimedia transmissions. Most of the terms are described and linked to its wikipedia, RFC or W3C relevant documents. Some of the terms are specific to *kurento*.

HTTP The is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

See also:

RFC 2616

JAVA Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible

JSON JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is designed to be easy to understand and write for humans and easy to parse for machines.

JSON-RPC JSON-RPC is a simple remote procedure call protocol encoded in JSON. JSON-RPC allows for notifications and for multiple calls to be sent to the server which may be answered out of order.

Kurento Kurento is a platform for the development of multimedia enabled applications. Kurento is the Esperanto term for the English word 'stream'. We chose this name because we believe the Esperanto principles are inspiring for what the multimedia community needs: simplicity, openness and universality. Kurento is open source, released under Apache 2.0, and has several components, providing solutions to most multimedia common services requirements. Those components include: *Kurento Media Server*, *Kurento API*, *Kurento Protocol*, and *Kurento Client*.

Kurento API **Kurento API** is an object oriented API to create media pipelines to control media. It can be seen as and interface to Kurento Media Server. It can be used from the Kurento Protocol or from Kurento Clients.

Kurento Client A **Kurento Client** is a programming library (Java or JavaScript) used to control **Kurento Media Server** from an application. For example, with this library, any developer can create a web application that uses Kurento Media Server to receive audio and video from the user web browser, process it and send it back again over Internet. Kurento Client exposes the *Kurento API* to app developers.

Kurento Protocol Communication between KMS and clients by means of *JSON-RPC* messages. It is based on *WebSocket* that uses *JSON-RPC* V2.0 messages for making requests and sending responses.

KMS

Kurento Media Server **Kurento Media Server** is the core element of Kurento since it responsible for media transmission, processing, loading and recording.

Maven Maven is a build automation tool used primarily for Java projects.

Sphinx Documentation generation system used for Brandtalk documentation.

See also:

Easy and beautiful documentation with Sphinx

Spring Boot Spring Boot is Spring's convention-over-configuration solution for creating stand-alone, production-grade Spring based Applications that you can "just run".[17] It takes an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss

TCP A reliable IP transport protocol. communication ensures that no packets are lost in transit. As such, it is most useful in low-bandwidth or unreliable environments. Examples are slow WANs or packet radio networks.

UUID Universally Unique IDentifier, also known as Globally Unique Identifier (GUID). In the context of the distributed computing environment, unique means Practically unique. It is not guaranteed to be unique because the identifiers have a finite size (16-octet number).

WebSocket

WebSockets WebSocket specification (developed as part of the HTML5 initiative) defines a full-duplex single socket connection over which messages can be sent between client and server.

WSS

WebSockets Secure WebSocket specification (developed as part of the HTML5 initiative) defines a full-duplex single socket connection over which messages can be sent between client and server.

Index