Web Call Server - REST methods

© 2017 Flashphoner, LLC
Part I
1 REST Methods

WCS integrates with the back-end server using REST / HTTP calls or REST Methods as we call them. Therefore, you provide the WCS server with the script API, and WCS sends data on connections, calls, video streams etc. to the API.

REST methods can be used to:
- Authenticate connections to the server by token or by password
- Receive real-time information about connects, disconnects, started and ended streams, calls and so on.
- Redefine data received from the client. For example, you can rename a stream or hide its real name or redirect a call.
- Implement custom signaling with Websockets transmission of data. For example, send a text message in the chat to all connected clients.

REST methods should be implemented as simple HTTP / HTTPS scripts that work in pair with JSON in the body of an HTTP request and return JSON in the HTTP body of responses.

**Request**

<table>
<thead>
<tr>
<th>Protocol</th>
<th>HTTP / HTTPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>POST</td>
</tr>
<tr>
<td>Content-Type</td>
<td>application/json</td>
</tr>
<tr>
<td>Body</td>
<td>JSON</td>
</tr>
</tbody>
</table>

**Response**
In response, WCS should receive a standard HTTP response containing the body of JSON

<table>
<thead>
<tr>
<th>status</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason phrase</td>
<td>OK</td>
</tr>
<tr>
<td>Body</td>
<td>JSON</td>
</tr>
</tbody>
</table>
2 Invoking a REST method

Let us take a look at operation of REST method through the example of the /publishStream method. In the test interface of Streamer, a user sends a video stream to the WCS server.

Upon clicking the Start button, the interface invokes the stream.publish() function that begins broadcasting.

Here is a step-by-step diagram of publishing and invoking the publishStream REST method:

1. stream.publish() is invoked on the client side
2. WCS queries the local web server at http://localhost:9091/publishStream
3. WCS receives 200 OK response from the local server
4. WCS sends the StreamStatusEvent event to the client, status = PUBLISHING
Invoking a REST method

The final result is the PUBLISHING status displayed on the web page:

Invocation of the /publishStream method (step 2) looks as follows when the HTTP protocol is analyzed in Wireshark:
## Invoking a REST method

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /EchoApp/publishStream HTTP/1.1</td>
<td>HTTP/1.1 200 OK</td>
</tr>
<tr>
<td>Accept: application/json, application/*+js</td>
<td>Server: Apache-Coyote/1.1</td>
</tr>
<tr>
<td>Content-Type: application/json;charset=UTF8</td>
<td>Content-Type: application/json;charset=UTF8</td>
</tr>
<tr>
<td>User-Agent: Java/1.8.0_45</td>
<td>Transfer-Encoding: chunked</td>
</tr>
<tr>
<td>Host: localhost:9091</td>
<td>Date: Sat, 25 Feb 2017 05:51:11 GMT</td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td></td>
</tr>
<tr>
<td>Content-Length: 3622</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>&quot;nodeId&quot;: &quot;H4gfHeULtX6ddGGUWwZxhUNyqZHUF&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;appKey&quot;: &quot;defaultApp&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;sessionId&quot;: &quot;/192.168.1.38:52791/192.16</td>
<td></td>
</tr>
<tr>
<td>&quot;mediaSessionId&quot;: &quot;87c5ff20-fb59-11e6-81&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;name&quot;: &quot;84cac22c&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;published&quot;: true,</td>
<td></td>
</tr>
<tr>
<td>&quot;hasVideo&quot;: true,</td>
<td></td>
</tr>
<tr>
<td>&quot;hasAudio&quot;: true,</td>
<td></td>
</tr>
<tr>
<td>&quot;status&quot;: &quot;PENDING&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;record&quot;: false,</td>
<td></td>
</tr>
<tr>
<td>&quot;width&quot;: 0,</td>
<td></td>
</tr>
<tr>
<td>&quot;height&quot;: 0,</td>
<td></td>
</tr>
<tr>
<td>&quot;bitrate&quot;: 0,</td>
<td></td>
</tr>
<tr>
<td>&quot;quality&quot;: 0,</td>
<td></td>
</tr>
<tr>
<td>&quot;mediaProvider&quot;: &quot;WebRTC&quot;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
3 Four types of REST methods

1. There are four types of REST methods
   - connect
   - direct invocations
   - event
   - incoming invocations

2. The `connect` method determines behavior of all other methods.
3. The `connect` method should return in the response only those data that it really received from WCS with an exception of the case when some field in the response should be explicitly overwritten or another status must be returned.
4. Other methods are recommended to return in the response only those data that they received with an exception of the case when some field in the response should be explicitly overwritten or another status must be returned.
5. All methods should return the 200 OK HTTP status with an exception of the case when another status should be explicitly returned, for example, 403 Forbidden.

Let’s review one method of each type (the remaining methods behave similarly within the corresponding type).

Type 1 - the `connect` method
Type 2 - the `direct invoke`
Type 3 - the `event`
Type 4 - the `direct invocation`

3.1 Type 1 - the `connect` method

Scheme of work

Authentication

Configuring other REST methods

3.1.1 Scheme of work

This method is invoked only once when the browser page connects to the WCS server via the Websocket protocol. Execution of the connect method is crucial, because this method is responsible for server connection authentication, and if it won’t run or stops with an error, the WCS server will decline the connection attempt.
1. On the browser side, Flashphoner.createSession() is executed. After that the browser attempts to establish connection to WCS.
2. WCS calls the connect REST method.
3. WCS receives 200 OK response and based on the answer, authorizes this connection attempt.
4. WCS sends a confirmation to the browser as a ConnectionStatusEvent event with the ESTABLISHED status.

Example:

<table>
<thead>
<tr>
<th>POST /EchoApp/connect HTTP/1.1</th>
<th>HTTP/1.1 200 OK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept: application/json, application/*</td>
<td>Server: Apache-Coyote/1.1</td>
</tr>
<tr>
<td>Content-Type: application/json; charset=utf-8</td>
<td>Content-Type: application/json; charset=utf-8</td>
</tr>
<tr>
<td>User-Agent: Java/1.8.0_45</td>
<td>Transfer-Encoding: chunked</td>
</tr>
<tr>
<td>Host: localhost:9091</td>
<td>Date: Sun, 26 Feb 2017 23:54:06 GMT</td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td>{</td>
</tr>
<tr>
<td>Content-Length: 537</td>
<td>&quot;nodeId&quot;:&quot;H4gfHeULtX6ddGGUwZxhUNyq229&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;appKey&quot;:&quot;defaultApp&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;sessionId&quot;:&quot;/192.168.1.38:64604/192.168.1.59&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;useWsTunnel&quot;:false,</td>
</tr>
<tr>
<td></td>
<td>&quot;useWsTunnelPacketization&quot;:false,</td>
</tr>
<tr>
<td></td>
<td>&quot;useBase64BinaryEncoding&quot;:false,</td>
</tr>
<tr>
<td></td>
<td>&quot;mediaProviders&quot;:{</td>
</tr>
<tr>
<td></td>
<td>&quot;WebRTC&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;WSPlayer&quot;</td>
</tr>
<tr>
<td></td>
<td>},</td>
</tr>
<tr>
<td></td>
<td>&quot;clientVersion&quot;:&quot;0.5.16&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;clientOSVersion&quot;:&quot;5.0 (Windows NT 6.3; WOW64)&quot;</td>
</tr>
</tbody>
</table>
| | }, | "clientBrowserVersion":"Mozilla/5.0" |}
3.1.2 Authentication

By default, WCS successfully authenticates all Websocket connections. That is, the built-in server at http://localhost:9091/EchoApp/connect always returns 200 OK.

You can redefine this behavior and make your web server return 403 Forbidden, in this case WCS will abort the incoming connection.

You can send a token, a password or any other information from the web page using the custom field. WCS transfers this field to the web server as the JSON body, and the web server can decide whether or not to authenticate based on these data.

Example:

```
POST /rest/my_api/connect HTTP/1.1
Accept: application/json, application/*+json
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.8.0_111
Host: 192.168.1.101
Connection: keep-alive
Content-Length: 578

{
    "nodeId":"Hw47CFMBEchVoBMDr29IjtdnJ1sJ",
    "appKey":"defaultApp",
    "sessionId":"/192.168.1.102:60204/192.168.1.101:8443",
    "useWsTunnel":false,
    "useWsTunnelPacketization2":false,
    "useBase64BinaryEncoding":false,
    "mediaProviders":{
        "WebRTC":false
    }
}
```

```
HTTP/1.1 403 Forbidden
Date: Tue, 28 Feb 2017 09:05:56 GMT
Server: Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Content-Length: 0
Connection: close
Content-Type: text/html; charset=UTF-8
```
3.1.3 Configuring other REST methods

During authentication you can configure how should operate all other REST methods invoked after the connection is established. For example: playStream, publishStream, StreamStatusEvent, and so on. To do this, the web server should return the `restClientConfig` field in the JSON body of the 200 OK response. `restClientConfig` is a JSON object that can hold a config to any existing REST method except `connect`.

Example of the JSON body of the 200 OK response with the `restClientConfig` field.

```json
{
    "nodeId" : "kPeSwMn1PFqIMCZ1Ry6dJ0JNlZ9cqZw@89.179.119.95",
    "appKey" : "defaultApp",
    "sessionId" : "/172.16.16.139:49405/89.179.119.95:8443",
    "useWsTunnel" : false,
    "useWsTunnelPacketization2" : false,
    "useBase64BinaryEncoding" : false,
    "mediaProviders" : [ "WebRTC", "WSPlayer" ],
    "restClientConfig" : {
        "custom" : {
            "token" : "abcdef"
        }
    }
}
```
"clientVersion" : "0.5.12",
"clientOSVersion" : "5.0 (Android 5.1.1)",
"clientBrowserVersion" : "Mozilla/5.0 (Android 5.1.1; Mobile; rv:50.0) Gecko/50.0 Firefox/50.0",
"restClientConfig":
{
    "publishStream":
    {
        "restExclude":"
        "clientExclude":"
        "restOnError":"FAIL",
        "restPolicy":"NOTIFY",
        "restOverwrite":""
    }
}

This response is constructed according to the following rules:
1. We return data received from WCS unchanged.
2. Add the field:

"restClientConfig":{
    "publishStream":{
        "restExclude":"
        "clientExclude":"
        "restOnError":"FAIL",
        "restPolicy":"NOTIFY",
        "restOverwrite":""
    }
}

This configuration tells the WCS server that invocation of the publishStream REST method will comply with the following rules:

1. restExclude is empty.

This means all fields in the JSON body of the response such as name, width, height and so on will be sent from WCS to the web server and none of these fields will be excluded from this query.

2. clientExclude is empty.

Not used in the publishStream REST method.

3. restOnError: FAIL

This means that if an error occurred while sending the query to the web server or if the web server returned 4xx HTTP status, the WCS server should forbid and interrupt publishing the stream.

The default value is restOnError: LOG. This means, WCS simply logs all occurring errors, do not forbid or interrupt operations such as publishStream, even if the web server returns 403 Forbidden.

4. restPolicy: NOTIFY

This means if the web server returns data different from those it received from WCS, the data are not applied. For instance, if WCS sent the field name=stream1, and the web-server returned name=stream2 in the body of the 200 OK response, the new value will not be applied, because of the NOTIFY policy.
5. restOverwrite is empty.

This setting is only applied if the previous restPolicy is set to OVERWRITE, that is it does not allow redefining fields with data received in the JSON body of the 200 OK response. The current value is empty, which means no fields can be redefined. To redefine a field, the list of fields must be specified explicitly, for example: restOverwrite="name,width"

Example:

```json
2

POST /rest/my_api/connect HTTP/1.1
Accept: application/json, application/*+json
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.8.0_111
Host: 192.168.1.101
Connection: keep-alive
Content-Length: 550

{
  "nodeId":"Hw47CFMBEcVOpBMDr29IxjudnJ1s",
  "appKey":"defaultApp",
  "sessionId":"/192.168.1.102:26518/192.
  "useWsTunnel":false,
  "useWsTunnelPacketization2":false,
  "useBase64BinaryEncoding":false,
  "mediaProviders":
    "WebRTC",
    "WSPlayer"
},
"clientVersion":"0.5.16",
"clientOSVersion":"5.0 (Windows NT 6.3
"clientBrowserVersion":"Mozilla/5.0 (W

3

HTTP/1.1 200 OK
Date: Tue, 28 Feb 2017 10:11:03 GMT
Server: Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Content-Length: 833
Connection: close
Content-Type: application/json

{
  "nodeId":"Hw47CFMBEcVOpBMDr29IxjudnJ1s",
  "appKey":"defaultApp",
  "sessionId":"/192.168.1.102:26518/192.
  "useWsTunnel":false,
  "useWsTunnelPacketization2":false,
  "useBase64BinaryEncoding":false,
  "mediaProviders":
    "WebRTC",
    "WSPlayer"
},
"clientVersion":"0.5.16",
"clientOSVersion":"5.0 (Windows NT 6.3
"clientBrowserVersion":"Mozilla/5.0 (W
  "restClientConfig":{
    "publishStream":{
      "clientExclude":"",
      "restExclude":"recordName",
      "restOnError":"FAIL",
      "restPolicy":"NOTIFY",
      "restOverwrite":""
    },
    "playStream":{
      "clientExclude":"",
      "restExclude":"",
      "restOnError":"LOG",
      "restPolicy":"OVERWRITE",
      "restOverwrite":"height,width"
    }
  }
}
```

3.2 Type 2 - the direct invoke

Scheme of work

Authentication
Redefining fields

Excluding fields from sending

3.2.1 Scheme of work

The publishStream REST method is a direct invoke method, because a client invokes this method with
the stream.publish() command attempting to publish a video stream from the web camera.
This operation can be authorized – cancelled or permitted – and parameters of this operation can be
redefined on the side of the web server.
For example, the field name='stream1' changes to name='stream2', and if such a replace succeeds,
WCS will publish the stream with the new name stream2.
We have already described the publishStream method above, so here we simply cite the invocation
sequence again:

Example:

```
POST /rest/my_api/publishStream HTTP/1.1
Accept: application/json, application/*+json
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.8.0_111
Host: 192.168.1.101
Connection: keep-alive
Content-Length: 3611

{
    "nodeId":"Hw47CFMBEchVOpBMDr29IxjudnJ1sjOY@192.168.1.101",
    "appKey":"defaultApp",
    "sessionId":"/192.168.1.102:4388/192.168.1.101:8443",
}
```

```
HTTP/1.1 200 OK
Date: Tue, 28 Feb 2017 17:35:43 GMT
Server: Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Content-Length: 3653
Connection: close
Content-Type: application/json

{
    "nodeId":"Hw47CFMBEchVOpBMDr29IxjudnJ1sjOY@192.168.1.101",
    "appKey":"defaultApp",
    "sessionId":"/192.168.1.102:4388/192.168.1.101:8443",
}
```
3.2.2 Authentication

Authentication is similar to that of the connect method. You can send a token or a password here too, as well as permit / forbid the operation based on other parameters.

For authentication you need to configure the publishStream method at the connection stage (connect) and set restOnError:FAIL in the restClientConfig config.

Example:

```
POST /rest/my_api/publishStream HTTP/1.1
Accept: application/json, application/*+json
Content-Type: application/json;charset=UTF-8

HTTP/1.1 403 Forbidden
Date: Tue, 28 Feb 2017 13:44:39 GMT
Server: Apache/2.2.15 (CentOS)
```
### 3.2.3 Redefining fields

Redefining fields works when the method operates normally and the 200 OK response is returned. The purpose of redefining fields is, for instance, to hide the real name of the stream on the page from the end user.

For example, the user sends stream1, but the server changes the name to stream2.

To make such redefining, do the following:

1. Enable "restPolicy":"OVERWRITE" in `restClientConfig` for the publishStream when connecting to the server.
2. Put a comma-separated list of fields to overwrite in "restOverwrite":"". For example, "restOverwrite":"name" for `restClientConfig`. In this case, only the name field – the name of a stream – is overwritten.
3. In the JSON body of the 200 OK response return the modified name field, and return other fields as they were received from the WCS server.

```json
{
    "nodeId":"Hw47CFMBeChVOpBMDr29IxjudnJ1
    "appKey":"defaultApp",
    "sessionId":":192.168.1.102:17749/192.
    "mediaSessionId":":0e17ab50-fdbc-11e6-8
    "name":":3a88",
    "published":true,
    "hasVideo":true,
    "hasAudio":true,
    "status":"PENDING",
    "sdp":":.....",
    "record":false,
    "width":0,
    "height":0,
    "bitrate":0,
    "quality":0,
    "mediaProvider":"WebRTC",
    "custom":{
        "token":"abcdef"
    }
}
```

---

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-Agent: Java/1.8.0_111</td>
<td>X-Powered-By: PHP/5.3.3</td>
</tr>
<tr>
<td>Host: 192.168.1.101</td>
<td>Content-Length: 0</td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td>Connection: close</td>
</tr>
<tr>
<td>Content-Length: 3639</td>
<td>Content-Type: text/html; charset=UTF-8</td>
</tr>
</tbody>
</table>
Example:

```
POST /rest/my_api/publishStream HTTP/1.1
Accept: application/json, application/*+json
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.8.0_111
Host: 192.168.1.101
Connection: keep-alive
Content-Length: 3612

{
  "nodeId":"Hw47CFMBEchVOpBMDr29IxjudnJ1
  "appKey":"defaultApp",
  "sessionId":"/192.168.1.102:12514/192.
  "mediaSessionId":"abbe7e90-fdcd-11e6-9
  "name":"2a0c",
  "published":true,
  "hasVideo":true,
  "hasAudio":true,
  "status":"PENDING",
  "sdp":".....",
  "record":false,
  "width":0,
  "height":0,
  "bitrate":0,
  "quality":0,
  "mediaProvider":"WebRTC"
}
```

```
HTTP/1.1 200 OK
Date: Tue, 28 Feb 2017 15:50:44 GMT
Server: Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Content-Length: 3669
Connection: close
Content-Type: application/json

{
  "nodeId":"Hw47CFMBEchVOpBMDr29IxjudnJ1
  "appKey":"defaultApp",
  "sessionId":"/192.168.1.102:12514/192.
  "mediaSessionId":"abbe7e90-fdcd-11e6-9
  "name":"streamChangedName",
  "published":true,
  "hasVideo":true,
  "hasAudio":true,
  "status":"PENDING",
  "sdp":.....",
  "record":false,
  "width":0,
  "height":0,
  "bitrate":0,
  "quality":0,
  "mediaProvider":"WebRTC"
}
```
3.2.4 Excluding fields from sending

This method works in one direction only, from WCS to the web server. This means you can exclude one or more fields that the WCS server sends to the web server in the JSON body of the response on step 2. By default, publishStream sends all fields. To exclude field, put a comma-separated list of fields you want to exclude to the "restExclude":"" parameter in `restClientConfig` when establishing connection.

Operation example when the name field – the name of a stream – is excluded:

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST <code>/rest/my_api/publishStream</code> HTTP/1.1</td>
<td>HTTP/1.1 200 OK</td>
</tr>
<tr>
<td>Accept: application/json, application/&quot;+json&quot;</td>
<td>Date: Tue, 28 Feb 2017 16:43:26 GMT</td>
</tr>
<tr>
<td>Content-Type: application/json; charset=UTF-8</td>
<td>Server: Apache/2.2.15 (CentOS)</td>
</tr>
<tr>
<td>User-Agent: Java/1.8.0_111</td>
<td>X-Powered-By: PHP/5.3.3</td>
</tr>
<tr>
<td>Host: 192.168.1.101</td>
<td>Content-Length: 3649</td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td>Connection: close</td>
</tr>
<tr>
<td>Content-Length: 3602</td>
<td>Content-Type: application/json</td>
</tr>
</tbody>
</table>

```
{
  "nodeId": "Hw47CFMBEchVOpBMDr29IxjudnJ1",
  "appKey": "defaultApp",
  "sessionId": "/192.168.1.102:9232/192.168.1.101:8443",
  "mediaSessionId": "084db2f0-fdd5-11e6-9ba5-6ba06f30ad92",
  "published": true,
  "hasVideo": true,
  "hasAudio": true,
  "status": "PENDING",
  "sdp": ".....",
  "record": false,
  "width": 0,
  "height": 0,
}
```

```
{
  "nodeId": "Hw47CFMBEchVOpBMDr29IxjudnJ1",
  "appKey": "defaultApp",
  "sessionId": "/192.168.1.102:9232/192.168.1.101:8443",
  "mediaSessionId": "084db2f0-fdd5-11e6-9ba5-6ba06f30ad92",
  "published": true,
  "hasVideo": true,
  "hasAudio": true,
  "status": "PENDING",
  "sdp": ".....",
  "record": false,
  "width": 0,
  "height": 0,
}
```
3.3 **Type 3 - the event**

**Scheme of work**

**Error handling**

3.3.1 **Scheme of work**

The StreamStatusEvent REST method is an event. Events occur internally in the WCS server and are used to pass statuses of various operations.

For instance, the StreamStatusEvent event is used to pass statuses of video stream operations such as stream.play(), stream.publish(), stream.snapshot().

Indeed, if we publish a video stream, play it, or make a snapshot of it, we must know its status to manage this stream.

The web server cannot authenticate (permit or forbid) an event and simply accepts it, for example to save information about the stream in the database.
On this diagram you can see that the StreamStatusEvent event goes in two directions:
1. To the client - step 4
2. To the web-server - step 5

Example:

```
POST /rest/my_api/StreamStatusEvent HTTP/1.1
Accept: application/json, application/"*+json"
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.8.0_111
Host: 192.168.1.101
Connection: keep-alive
Content-Length: 3614

{
    "nodeId":"Hw47CFMBEchVOpBMDr29IxjudnJ1",
    "appKey":"defaultApp",
    "sessionId": "/192.168.1.102:4388/192.168.1.1102:8443"
}

HTTP/1.1 200 OK
Date: Tue, 28 Feb 2017 17:35:44 GMT
Server: Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Content-Length: 3656
Connection: close
Content-Type: application/json

{  
    "nodeId": "Hw47CFMBEchVOpBMDr29IxjudnJ1",
    "appKey": "defaultApp",
    "sessionId": "/192.168.1.102:4388/192.168.1.1102:8443"
}
```
3.3.2 Error handling

By default, WCS does not check the status of the response to invocation of the `/StreamStatusEvent` REST method.

That is, if the web server returns HTTP error status 403 or 500 or any other, the WCS simply ignores that.

To make WCS react to the error, set restOnError: FAIL in the settings of the StreamStatusEvent method, in `restClientConfig` when establishing the connection.

In this case, the client receives an additional event, ErrorEvent and is notified about the error.
On the step 6, the web server returns the HTTP status 500 Internal Error in response to invocation of the /StreamStatusEvent method. The WCS server on the step 7 notifies the client of the occurred error.

Example:

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /rest/my_api/StreamStatusEvent HTTP/1</td>
<td>HTTP/1.1 500 Internal Server Error</td>
</tr>
<tr>
<td>Accept: application/json, application/**+json</td>
<td>Date: Tue, 28 Feb 2017 18:22:49 GMT</td>
</tr>
<tr>
<td>Content-Type: application/json;charset=UTF-8</td>
<td>Server: Apache/2.2.15 (CentOS)</td>
</tr>
<tr>
<td>User-Agent: Java/1.8.0_111</td>
<td>X-Powered-By: PHP/5.3.3</td>
</tr>
<tr>
<td>Host: 192.168.1.101</td>
<td></td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td>Content-Length: 0</td>
</tr>
<tr>
<td>Content-Length: 3624</td>
<td>Connection: close</td>
</tr>
</tbody>
</table>

```json
{
    "nodeId":"Hw47CFMBeChVOpBMDr29IjjudnJ1",
    "appKey":"defaultApp",
    "sessionId":"/192.168.1.102:25301/192.
}"
```
3.4 Type 4 - the direct invocation

**Scheme of work**

**Error handling**

3.4.1 Scheme of work

A client can send an arbitrary object by invoking the sendData() direct method – step 1. On the web server the /OnDataEvent REST method is invoked – step 2. As a result, the client receives the SESSION_STATUS.SEND_DATA_STATUS_ACCEPTED status - step 4.
Example:

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /rest/my_api/OnCallEvent HTTP/1.1</td>
<td>HTTP/1.1 200 OK</td>
</tr>
<tr>
<td>Accept: application/json, application/</td>
<td>Date: Tue, 28 Feb 2017 17:35:44 GMT</td>
</tr>
<tr>
<td>application/*+json</td>
<td>Server: Apache/2.2.15 (CentOS)</td>
</tr>
<tr>
<td>Content-Type: application/json; charset=</td>
<td>X-Powered-By: PHP/5.3.3</td>
</tr>
<tr>
<td>UTF</td>
<td>Content-Length: 3656</td>
</tr>
<tr>
<td>User-Agent: Java/1.8.0_111</td>
<td>Connection: close</td>
</tr>
<tr>
<td>Host: 192.168.1.101</td>
<td>Content-Type: application/json</td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td></td>
</tr>
<tr>
<td>Content-Length: 3614</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>&quot;nodeId&quot; : &quot;ah20GKYnbYKxjJmdh1zDs6c1lWkn&quot;</td>
<td>&quot;nodeId&quot; : &quot;ah20GKYnbYKxjJmdh1zDs6c1lWkn&quot;</td>
</tr>
<tr>
<td>&quot;appKey&quot; : &quot;defaultApp&quot;,</td>
<td>&quot;appKey&quot; : &quot;defaultApp&quot;,</td>
</tr>
<tr>
<td>&quot;sessionId&quot; : &quot;/8.8.8.8:56046/46.101.108&quot;</td>
<td>&quot;sessionId&quot; : &quot;/8.8.8.8:56046/46.101.108&quot;</td>
</tr>
<tr>
<td>&quot;callId&quot; : &quot;pbx-8BD6EE910B71BE609010-343&quot;</td>
<td>&quot;callId&quot; : &quot;pbx-8BD6EE910B71BE609010-343&quot;</td>
</tr>
<tr>
<td>&quot;incoming&quot; : true,</td>
<td>&quot;incoming&quot; : true,</td>
</tr>
<tr>
<td>&quot;status&quot; : &quot;PENDING&quot;,</td>
<td>&quot;status&quot; : &quot;PENDING&quot;,</td>
</tr>
<tr>
<td>&quot;caller&quot; : &quot;PPP009&quot;,</td>
<td>&quot;caller&quot; : &quot;PPP009&quot;,</td>
</tr>
<tr>
<td>&quot;callee&quot; : &quot;PPP005&quot;,</td>
<td>&quot;callee&quot; : &quot;PPP005&quot;,</td>
</tr>
<tr>
<td>&quot;hasAudio&quot; : true,</td>
<td>&quot;hasAudio&quot; : true,</td>
</tr>
<tr>
<td>&quot;hasVideo&quot; : false,</td>
<td>&quot;hasVideo&quot; : false,</td>
</tr>
<tr>
<td>&quot;mediaProvider&quot; : &quot;WebRTC&quot;,</td>
<td>&quot;mediaProvider&quot; : &quot;WebRTC&quot;,</td>
</tr>
<tr>
<td>&quot;isMsrp&quot; : false,</td>
<td>&quot;isMsrp&quot; : false,</td>
</tr>
<tr>
<td>&quot;holdForTransfer&quot; : false</td>
<td>&quot;holdForTransfer&quot; : false</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

3.4.2 Error handling

To enable this behavior, you need to pass restOnError:FAIL to RestClientConfig for the OnDataEvent method when establishing connection using the connect method.

A client can send an arbitrary object by invoking the sendData() direct method – step 1. On the web server the /OnDataEvent REST method is invoked – step 2.

As a result, the client receives the SESSION_STATUS.SEND_DATA_STATUS FAILED status - step 4.
Example:

```
2
POST /rest/my_api/OnCallEvent HTTP/1.1
Accept: application/json, application/*+json
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.8.0_111
Host: 192.168.1.101
Connection: keep-alive
Content-Length: 3639

{  
  "nodeId" : "ah20GKYnbyXKxjMdh1zDs6cllWkn5",
  "appKey" : "defaultApp",
  "sessionId" : "/8.8.8.8:56046/46.101.108",
  "callId" : "pbx-8BD6EE910B71BE609010-343",
  "incoming" : true,
  "status" : "PENDING",
  "caller" : "PPP009",
  "callee" : "PPP005",
  "hasAudio" : true,
  "hasVideo" : false,
  "mediaProvider" : "WebRTC",
  "isMsrp" : false,
  "holdForTransfer" : false
}

3
HTTP/1.1 403 Forbidden
Date: Tue, 28 Feb 2017 13:44:39 GMT
Server: Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Content-Length: 0
Connection: close
Content-Type: text/html; charset=UTF-8
```
Part IV
4  Scheme of work restClientConfig

The below table lists all fields of the restClientConfig config and how the WCS server interprets them.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clientExclude</td>
<td>The list of fields to exclude from sending to a client (comma separated)</td>
<td>empty</td>
</tr>
<tr>
<td>restExclude</td>
<td>The list of fields to exclude from sending to REST (comma separated)</td>
<td>empty</td>
</tr>
<tr>
<td>restOnError</td>
<td>Reaction to errors or 4xx statuses:</td>
<td>LOG</td>
</tr>
<tr>
<td></td>
<td>LOG Just log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FAIL React and send the error</td>
<td></td>
</tr>
<tr>
<td>restPolicy</td>
<td>Ability to redefine fields: OVERWRITE, NOTIFY or DISABLED</td>
<td>NOTIFY</td>
</tr>
<tr>
<td></td>
<td>OVERWRITE Overwrite</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NOTIFY Do not overwrite</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DISABLED Do not send to REST</td>
<td></td>
</tr>
<tr>
<td>restOverwrite</td>
<td>The list of fields to overwrite if the option restPolicy=OVERWRITE (comma separated)</td>
<td>empty</td>
</tr>
</tbody>
</table>

Error handling

The table displays reaction of WCS to the HTTP statuses of 4xx errors while executing REST methods:

<table>
<thead>
<tr>
<th>restClientConfig settings</th>
<th>connect</th>
<th>direct invocation</th>
<th>event</th>
<th>incoming invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>restOnError = LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Declines</td>
<td></td>
<td>• Logs</td>
<td>• Logs</td>
</tr>
<tr>
<td></td>
<td>• Returns FAIL to a client</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>restOnError = FAIL</td>
<td>• Declines</td>
<td>• Declines</td>
<td>• Declines</td>
<td>• Declines</td>
</tr>
<tr>
<td></td>
<td>• Returns FAIL to a client</td>
<td>• Returns FAIL to a client</td>
<td>• Returns FAIL to a client</td>
<td>• Returns FAIL to the initiator</td>
</tr>
</tbody>
</table>
Part V
5 Controlling REST methods

Switching to your own web server

By default, all REST queries follow to the local address: http://localhost:9091/EchoApp
The invocations of REST methods look as follows then:
http://localhost:9091/EchoApp/connect
http://localhost:9091/EchoApp/playStream
http://localhost:9091/EchoApp/publishStream
and so on.

This happens because the appKey=defaultApp parameter is sent when connection to the WCS server,
and the default app for this key is this URL: http://localhost:9091/EchoApp

To change these settings, enter to the command line interface of the WCS server via SSH:

ssh -p 2000 admin@localhost

Use the password you have specified when installing WCS.
The standard password is: admin

Upon successful authorization, WCS creates a command line:

> show apps
Or enter the help command to display the list of available commands:

>help

Then, we create our own REST URL. It is created using the add app command.

Suppose, we allocated the following address on the web server for REST methods:
http://mywebserver.com/rest/connect
http://mywebserver.com/rest/playStream
http://mywebserver.com/rest/publishStream
and so on.

So we configure this as follows:

>add app myApp myAppKey "http://mywebserver.com/rest"

This command creates a new URL, http://mywebserver.com/rest, and assigns this URL for usage if a
client sent the appKey='myAppKey' during connection.

Therefore, if the newly created myAppKey is specified while creating the connection, the control is
passed to the assigned URL:

Flashphoner.createSession({appKey:'myAppKey'..});

http://mywebserver.com/rest/connect

Adding and deleting REST methods

By default, REST methods are not added to the newly configured URL. To add the REST methods, use
the following CLI commands:

>add app-rest-method -a myApp

This will add all REST methods to the myApp application.

If you add all REST methods at once, make sure your HTTP URLs: http://mywebserver.com/rest/
connect http://mywebserver.com/rest/playStream and others respond with 200 OK and return the data
they have received.

Besides, you can add one specific REST method using the following command:

>add app-rest-method myApp connect

In this case only one REST method is added: connect.

To delete REST methods, use the following commands:

remove app-rest-method -a myApp
remove app-rest-method myApp connect

In this case, REST methods will be unassigned from your URL, and the corresponding REST methods -
connect, playStream and others – will not be invoked.
The match between client invocations and REST methods

The table below lists methods, types and corresponding REST methods and events related to these REST methods. The Web SDK field illustrates how to invoke methods and handle events on the client’s side.

<table>
<thead>
<tr>
<th>Description</th>
<th>Web SDK</th>
<th>REST-method</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect</td>
<td></td>
<td>connect</td>
<td>ConnectionStatusEvent, RegistrationStatusEvent</td>
</tr>
<tr>
<td>Establish connection to WCS</td>
<td>Flashphoner.createSession()</td>
<td>connect</td>
<td></td>
</tr>
<tr>
<td>Direct invocations - Streaming</td>
<td></td>
<td>playStream</td>
<td>StreamStatusEvent</td>
</tr>
<tr>
<td>Play the stream</td>
<td>stream.play()</td>
<td>playStream</td>
<td>StreamStatusEvent</td>
</tr>
<tr>
<td>Pause playback of the stream</td>
<td>stream.pause()</td>
<td>pauseStream</td>
<td>StreamStatusEvent</td>
</tr>
<tr>
<td>Stop playback of the stream</td>
<td>stream.stop()</td>
<td>stopStream</td>
<td>StreamStatusEvent</td>
</tr>
<tr>
<td>Publish the stream</td>
<td>stream.publish()</td>
<td>publishStream</td>
<td>StreamStatusEvent</td>
</tr>
<tr>
<td>Make a snapshot of the stream</td>
<td>stream.snapshot()</td>
<td>snapshot</td>
<td>StreamStatusEvent</td>
</tr>
<tr>
<td>Stop publishing the stream</td>
<td>stream.stop()</td>
<td>unPublishStream</td>
<td>StreamStatusEvent</td>
</tr>
<tr>
<td>Direct invocations - Calls</td>
<td></td>
<td>call</td>
<td>CallStatusEvent</td>
</tr>
<tr>
<td>Make an outbound call</td>
<td>call.call()</td>
<td>call</td>
<td>CallStatusEvent</td>
</tr>
<tr>
<td>Send tone signal (DTMF)</td>
<td>call.sendDTMF()</td>
<td>sendDtmf</td>
<td>-</td>
</tr>
<tr>
<td>Answer an incoming call</td>
<td>call.answer()</td>
<td>answer</td>
<td>CallStatusEvent</td>
</tr>
<tr>
<td>Put the call on hold</td>
<td>call.hold()</td>
<td>hold</td>
<td>CallStatusEvent</td>
</tr>
<tr>
<td>Release the call from hold</td>
<td>call.unhold()</td>
<td>unhold</td>
<td>CallStatusEvent</td>
</tr>
<tr>
<td>Transfer the call to another number</td>
<td>call.transfer()</td>
<td>transfer</td>
<td>TransferStatusEvent</td>
</tr>
<tr>
<td>Hang up the call</td>
<td>call.hangup()</td>
<td>hangup</td>
<td>CallStatusEvent</td>
</tr>
<tr>
<td>Incoming invocations – Calls</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive the</td>
<td>session.on(SESSION_STATUS.INCO)</td>
<td>OnCallEvent</td>
<td>CallStatusEvent</td>
</tr>
<tr>
<td>Description</td>
<td>Web SDK</td>
<td>REST-method</td>
<td>Events</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------------------</td>
<td>------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>incoming call</td>
<td>MING_CALL, handler)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Direct and incoming invocations – Data</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send data</td>
<td>session.sendData()</td>
<td>OnDataEvent</td>
<td>DataStatusEvent</td>
</tr>
<tr>
<td>Receive data</td>
<td>session.on(SESSION_STATUS.APP_DATA, handler)</td>
<td>OnDataEvent</td>
<td>DataStatusEvent</td>
</tr>
<tr>
<td><strong>Direct invocations – Diagnostics</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Save client logs on the server</td>
<td>session.pushLogs()</td>
<td>pushLogs</td>
<td></td>
</tr>
<tr>
<td>Enable debug logs for streaming</td>
<td>session.startDebug()</td>
<td>sessionDebug</td>
<td>SessionDebugStatusEvent</td>
</tr>
<tr>
<td>Disable debug logs for streaming</td>
<td>session.stopDebug()</td>
<td>sessionDebug</td>
<td>SessionDebugStatusEvent</td>
</tr>
<tr>
<td><strong>Session events</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connection status</td>
<td>session.on(SESSION_STATUS.STATUS, handler)</td>
<td>ConnectionStatusEvent</td>
<td>ConnectionStatusEvent</td>
</tr>
<tr>
<td>Registration status</td>
<td>session.on(SESSION_STATUS.STATUS, handler)</td>
<td>RegistrationStatusEvent</td>
<td>RegistrationStatusEvent</td>
</tr>
<tr>
<td>Session debug status</td>
<td>session.on(SESSION_STATUS.DEBUG, handler)</td>
<td>SessionDebugStatusEvent</td>
<td>SessionDebugStatusEvent</td>
</tr>
<tr>
<td><strong>Stream events</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream status</td>
<td>stream.on(STREAM_STATUS.STATUS, handler)</td>
<td>StreamStatusEvent</td>
<td>StreamStatusEvent</td>
</tr>
<tr>
<td><strong>Call events</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call status</td>
<td>call.on(CALL_STATUS.STATUS, handler)</td>
<td>CallStatusEvent</td>
<td>CallStatusEvent</td>
</tr>
<tr>
<td><strong>Other events</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The event sent by the server to REST while publishing / playing the stream</td>
<td>Not used by the client</td>
<td>StreamKeepAliveEvent</td>
<td>StreamKeepAliveEvent</td>
</tr>
</tbody>
</table>
Part VII
7  REST methods object fields

The basic fields and their descriptions are listed in the documentation to Call Flow 1.0

To get the precise list of fields use:

   This log displays all REST queries sent to the web server and all received answers.

2. The tcpdump utility to record traffic and the Wireshark app to view it.

   tcpdump udp port 9091 -i lo -s 4096 -w log.pcap

   This command records traffic that is sent to and received on the port 9091. Later, you can view results in Wireshark and see how the fields are sent exactly.
## Event statuses

The event statuses are listed in the table below.

If such event is received on Web SDK (client-side), it will be processed on JavaScript in a correspondence object (stream, session, call, etc).

### Examples:

1. Handle success connection to WCS server.

   ```javascript
   session.on(SESSION_STATUS.ESTABLISHED, function(){
       trace("connection established");
   });
   ```

2. Handle success stream publishing to WCS server.

   ```javascript
   stream.on(STREAM_STATUS.PUBLISHING, function(){
       trace("stream is publishing");
   });
   ```

<table>
<thead>
<tr>
<th>Event</th>
<th>Statuses (the status field)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionStatusEvent</td>
<td>ESTABLISHED: Connection is established</td>
<td>The status of establishing connection to the WCS server via Websockets</td>
</tr>
<tr>
<td></td>
<td>FAILED: Error while establishing connection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DISCONNECTED: Connection is lost</td>
<td></td>
</tr>
<tr>
<td>RegistrationStatusEvent</td>
<td>REGISTERED: SIP account is registered</td>
<td>SIP registration status</td>
</tr>
<tr>
<td></td>
<td>UNREGISTERED: SIP account is unregistered</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AUTHENTICATION_FAIL: Authentication error</td>
<td></td>
</tr>
<tr>
<td>CallStatusEvent</td>
<td>TRYING: SIP returned TRYING</td>
<td>SIP call status</td>
</tr>
<tr>
<td></td>
<td>RING: SIP returned 183 RINGING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SESSION_PROGRESS: SIP returned 183 SESSION PROGRESS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BUSY: SIP returned 486 or 600 BUSY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ESTABLISHED: SIP returned 200 OK to INVITE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HOLD: SIP call is put on hold</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FINISH: SIP call is finished</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FAILED: SIP call has failed or ended with an error</td>
<td></td>
</tr>
<tr>
<td>TransferStatusEvent</td>
<td>ACCEPTED: SIP-call transfer request is accepted</td>
<td>Call transfer status</td>
</tr>
<tr>
<td>Event</td>
<td>Statuses (the status field)</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TRYING</td>
<td>An attempt to transfer the call</td>
<td></td>
</tr>
<tr>
<td>COMPLETED</td>
<td>Transfer is completed</td>
<td></td>
</tr>
<tr>
<td>FAILED</td>
<td>Transfer has failed</td>
<td></td>
</tr>
<tr>
<td>StreamStatusEvent</td>
<td>PUBLISHING</td>
<td>Stream is being published</td>
</tr>
<tr>
<td></td>
<td>UNPUBLISHED</td>
<td>Publishing finished</td>
</tr>
<tr>
<td></td>
<td>PLAYING</td>
<td>Stream is playing</td>
</tr>
<tr>
<td></td>
<td>STOPPED</td>
<td>Playback is stopped</td>
</tr>
<tr>
<td></td>
<td>FAILED</td>
<td>Error while publishing or playing</td>
</tr>
<tr>
<td>StreamKeepAliveEvent</td>
<td>PUBLISHING</td>
<td>Stream is being published</td>
</tr>
<tr>
<td></td>
<td>PLAYING</td>
<td>Stream is playing</td>
</tr>
<tr>
<td>SessionDebugStatusEvent</td>
<td>STARTED</td>
<td>Debug logs are enabled</td>
</tr>
<tr>
<td></td>
<td>STOPPED</td>
<td>Debug logs are disabled</td>
</tr>
<tr>
<td>DataStatusEvent</td>
<td>ACCEPTED</td>
<td>Data are accepted for processing</td>
</tr>
<tr>
<td></td>
<td>FAILED</td>
<td>An error occurred</td>
</tr>
</tbody>
</table>
9 Data exchange - sendData

Sending to the server

Data exchange sendData sending to connected client

9.1 Sending to the server

Client can send an arbitrary data-object to server using the direct method `sendData()` - step 1.
Web-server will invoke REST-method `/OnDataEvent` - step 2. As a result, client will receive status `SESSION_STATUS.SEND_DATA_STATUS ACCEPTED` - step 4.

Example:

```plaintext
POST /rest/my_api/OnDataEvent HTTP/1.1
Accept: application/json, application/*+json
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.8.0_111
Host: 192.168.1.101
Connection: keep-alive
Content-Length: 218

{
    "nodeId":"Hw47CFMBEcVOpBMDr29IjjudnJ1",
    "appKey":"defaultApp",
    "sessionId":"/192.168.1.102:55839/192.168.1.101:8443",
    "operationId":"d1999750-fde9-11e6-9f1b-913210792931",
    "payload":{
    ...
```

```
HTTP/1.1 200 OK
Date: Tue, 28 Feb 2017 19:12:14 GMT
Server: Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Content-Length: 220
Connection: close
Content-Type: application/json

{
    "nodeId":"Hw47CFMBEcVOpBMDr29IjjudnJ1",
    "appKey":"defaultApp",
    "sessionId":"/192.168.1.102:55839/192.168.1.101:8443",
    "operationId":"d1999750-fde9-11e6-9f1b-913210792931",
    "payload":{
    ...
```

```
**Error handling**

To make WCS react to the error, set restOnError: FAIL in the settings of the /OnDataEvent method, in `restClientConfig` when establishing the connection.

Client can send an arbitrary object using the direct method `sendData()` - step 1.

Web-server will invoke REST-method /OnDataEvent - step 2. As a result, client will receive status `SESSION_STATUS.SEND_DATA_STATUS FAILED` - step 4.

Example:

```
POST /rest/my_api/OnDataEvent HTTP/1.1
Accept: application/json, application/*+json
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.8.0_111
Host: 192.168.1.101
Connection: keep-alive
Content-Length: 218

{
  "nodeId":"Hw47CFMBEchVOpBMDr29IjnudnJ1",
  "appKey":"defaultApp",
  "sessionId":"/192.168.1.102:25789/192.
}
```

HTTP/1.1 403 Forbidden
Date: Tue, 28 Feb 2017 19:26:30 GMT
Server: Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Content-Length: 0
Connection: close
Content-Type: text/html; charset=UTF-8
9.2 Data exchange sendData sending to connected client

There is a way to send a direct message to a connected client http://host:9091/RESTCallback/sendData

To do this, pass the following JSON object

```
{
    "nodeId": "",
    "sessionId": "",
    "operationId": "",
    "payload": {
        "test": "test"
    }
}
```

In this case, the connected client receives the custom object where you can put any data, for example {"test": "test"} as shown above.
The client receives the SESSION_STATUS.APP_DATA event.