Taking a PNG snapshot of the stream

Overview

WCS provides a way to take a snapshot of the published stream using REST-queries as well as using JavaScript API.

Supported protocols

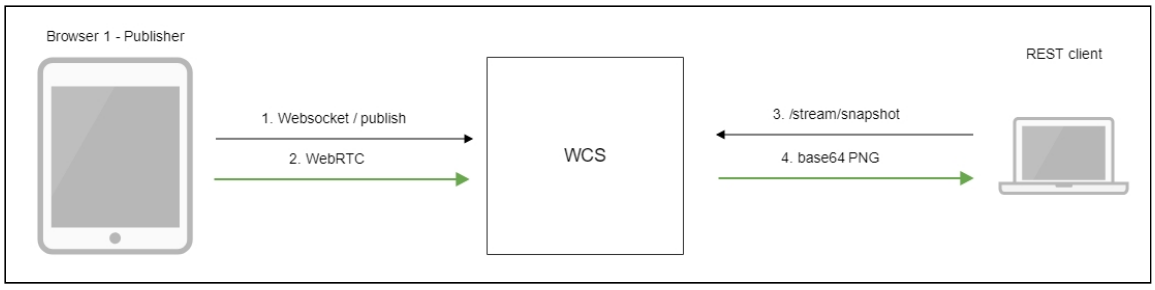
- WebRTC
- RTMP
- RTSP

Supported snapshot formats

- PNG

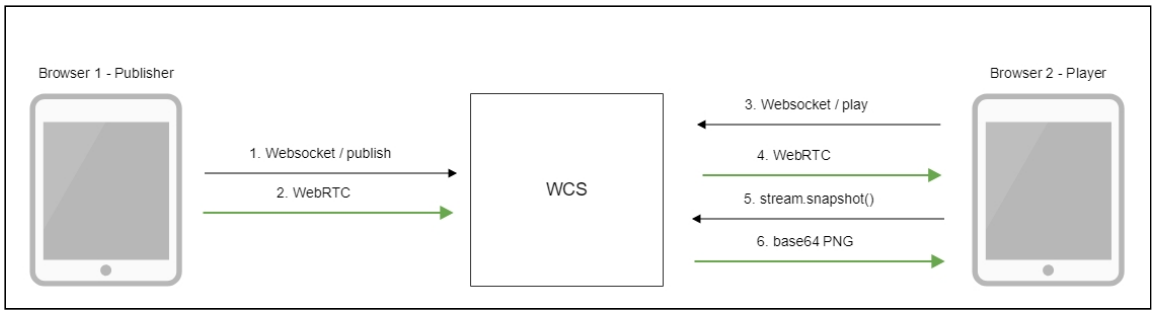
Operation flowchart

1: Using the REST query



1. The browser connects to the server via the Websocket protocol and sends the `publishStream` command.
2. The browser captures the microphone and the camera and sends the WebRTC stream to the server.
3. The REST client sends to the WCS the `/stream/snapshot` REST query.
4. The REST client receives a response with the base64-encoded snapshot of the stream.

2: Using JavaScript API



1. The browser connects to the server via the Websocket protocol and sends the `publishStream` command.
2. The browser captures the microphone and the camera and sends the WebRTC stream to the server.
3. The second browser establishes a connection also via Websocket and sends the `playStream` command.
4. The second browser receives the WebRTC stream and plays this stream on the page.
5. The second browser invokes `stream.snapshot()` to take a snapshot.
6. The second browser receives a response with the base64-encoded snapshot of the stream.

REST queries

WCS-server supports the `/stream/snapshot` REST method to take a snapshot.

A REST-query must be an HTTP/HTTPS POST request as follows:

- HTTP: `http://streaming.flashphoner.com:8081/rest-api/stream/snapshot`
- HTTPS: `https://streaming.flashphoner.com:8444/rest-api/stream/snapshot`

Here:

- `streaming.flashphoner.com` - is the address WCS server
- `8081` - is the standard REST / HTTP port of the WCS server
- `8444` - is the standard HTTPS port
- `rest-api` - is the required part of the URL
- `/stream/snapshot` - is the REST method used

REST-methods and response statuses

REST-method	Request body	Response bosy	Response status

REST-method	Request body	Response bosy	Response status
`/stream/snapshot`	`{ "streamName": "64966f33" }`	`{ "data": "iVBORw0KGgoAAAANSUhEUgAAAUAAAADwCAYAAAB...` }	200 OK 404 Stream not found 500 Internal server error

Parameters

Parameter	Description	Example
streamName	Unique stream name	`64966f33`
data	Snapshot file encoded to base64	`iVBORw0KGgoAAAANSUhEUgAAAUAAAADwCAYAAAABxLb1rAAAACXBIWXMAAAAAAAAAAQCEeRdzAAAQA...`

Sending the REST query to the WCS server

To send the REST query to the WCS server you need to use a REST-client.

Configuration

Since build 5.2.1116, a maximum snapshot taking duration, including a possible server disk I/O delay, may be configured when taking snapshot via REST API. By default, maximum duration is set to 3000 ms, and 30 checks if snapshot file is ready will be performed during this interval

```
snapshot_taking_interval_ms=3000
snapshot_taking_attempts=30
```

If the snapshot file is not ready, and the interval is expired, `/stream/snapshot` request will return the following error

```
{
  "exception":
"com.flashphoner.rest.server.exception.InternalErrorException",
  "reason": "com.flashphoner.rest.server.exception.InternalErrorException,
Internal Server Error, Snapshot response timeout, ts: 1640836780816, path:
/rest-api/stream/snapshot",
```

```
  "path": "/rest-api/stream/snapshot",
  "error": "Internal Server Error",
  "message": "Snapshot response timeout",
  "timestamp": 1640836780816,
  "status": 500
}
```

JavaScript API

The `snapshot` method of the `Stream` object in WebSDK is intended to take stream snapshots. Example of use of this method can be found in the Stream Snapshot web applications that publishes a stream and take a snapshot.

stream-snapshot.html

stream-snapshot.js

1. Creating a new stream from the published stream
   code:

   ```
   function snapshot(name) {
       setSnapshotStatus();
       var session = Flashphoner.getSessions()[0];
       session.createStream({name: name}).on(STREAM_STATUS.SNAPSHOT_COMPLETE,
   function(stream){
       ...
   }
   ```

2. Invoking the `snapshot()` method
   code:

   ```
   function snapshot(name) {
       setSnapshotStatus();
       var session = Flashphoner.getSessions()[0];
       session.createStream({name: name}).on(STREAM_STATUS.SNAPSHOT_COMPLETE,
   function(stream){
           ...
       }).snapshot();
   }
   ```

3. Upon receiving the `SNAPSHOT_COMPLETE` event, the `stream.getInfo()` function returns the base64 encoded snapshot
   code:

   ```
   function snapshot(name) {
       setSnapshotStatus();
       var session = Flashphoner.getSessions()[0];
       session.createStream({name: name}).on(STREAM_STATUS.SNAPSHOT_COMPLETE,
   function(stream){
           console.log("Snapshot complete");
   ```

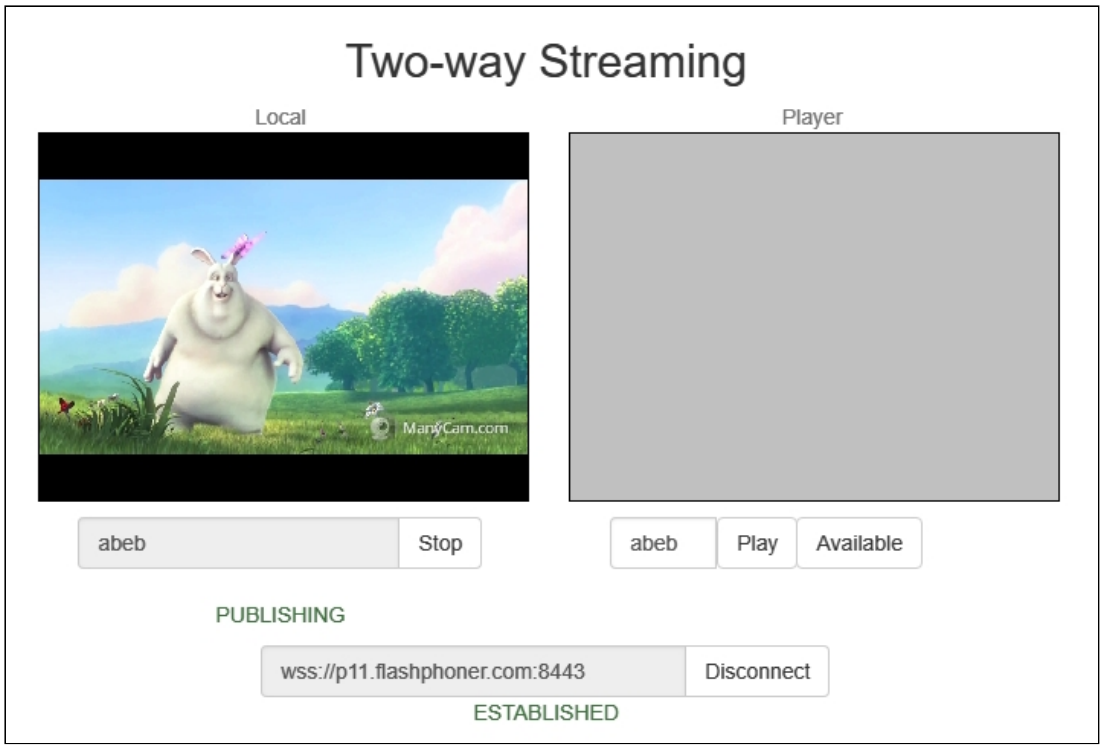
```
        setSnapshotStatus(STREAM_STATUS.SNAPSHOT_COMPLETE);
        snapshotImg.src = ""+stream.getInfo();
        ...
}
```

4. The stream stops
   code:

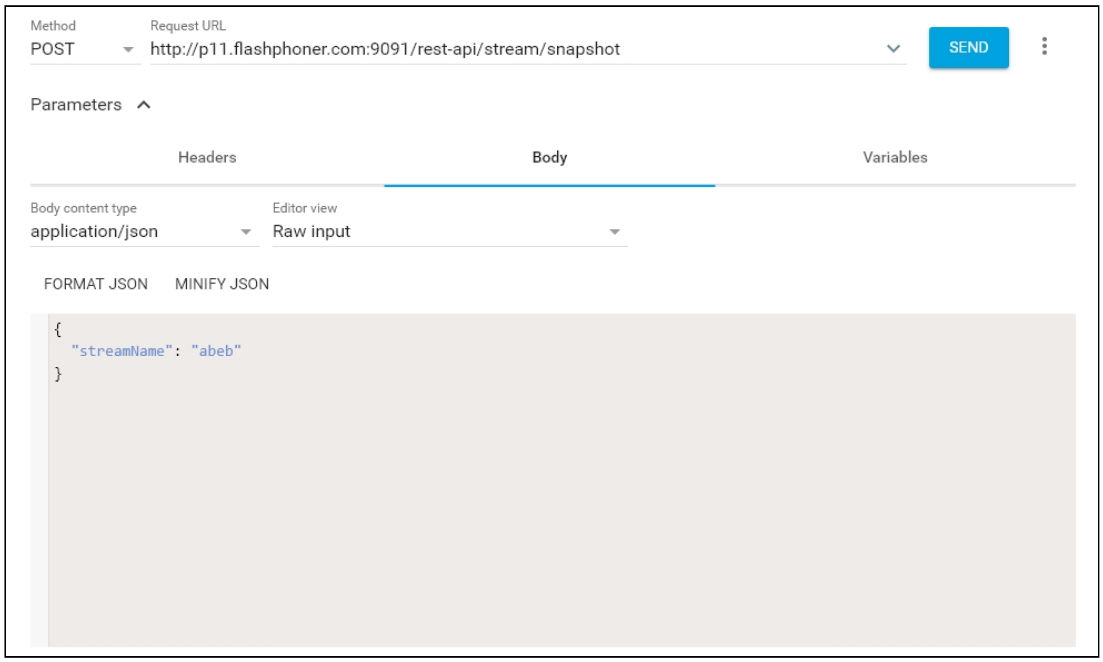
```
function snapshot(name) {
    setSnapshotStatus();
    var session = Flashphoner.getSessions()[0];
    session.createStream({name: name}).on(STREAM_STATUS.SNAPSHOT_COMPLETE,
function(stream){
        ...
        stream.stop();
    }).on(STREAM_STATUS.FAILED, function(stream){
        setSnapshotStatus(STREAM_STATUS.FAILED);
        console.log("Snapshot failed, info: " + stream.getInfo());
    }).snapshot();
}
```

Quick manual on testing

1. For the test we use:
2. the demo server `demo.flashphoner.com`;
3. the Chrome browser and the REST-client to send queries to the server;
4. the Two Way Streaming web application to publish the stream;
5. the https://www.motobit.com/util/base64-decoder-encoder.asp service to decode the snapshot.
6. Open the page of the Two Way Streaming application. Click `Connect`, then click `Publish` to publish the stream:

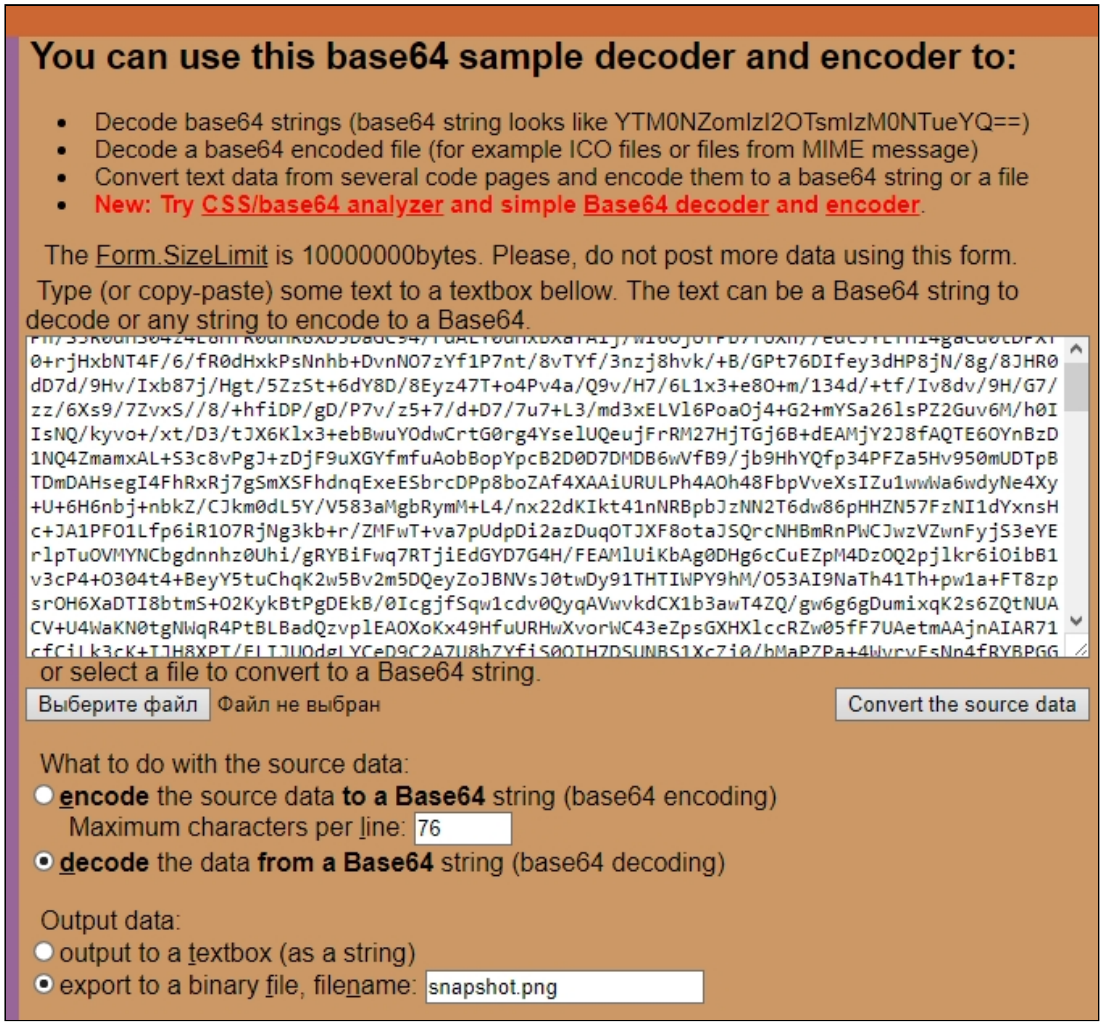


7. Open the REST-client. Send the /stream/snapshot query and pass the name of the published stream in parameters:



8. Make sure the response is received:

9. Open the online decoder and copy the response content to the form, then click `Convert the source data`:



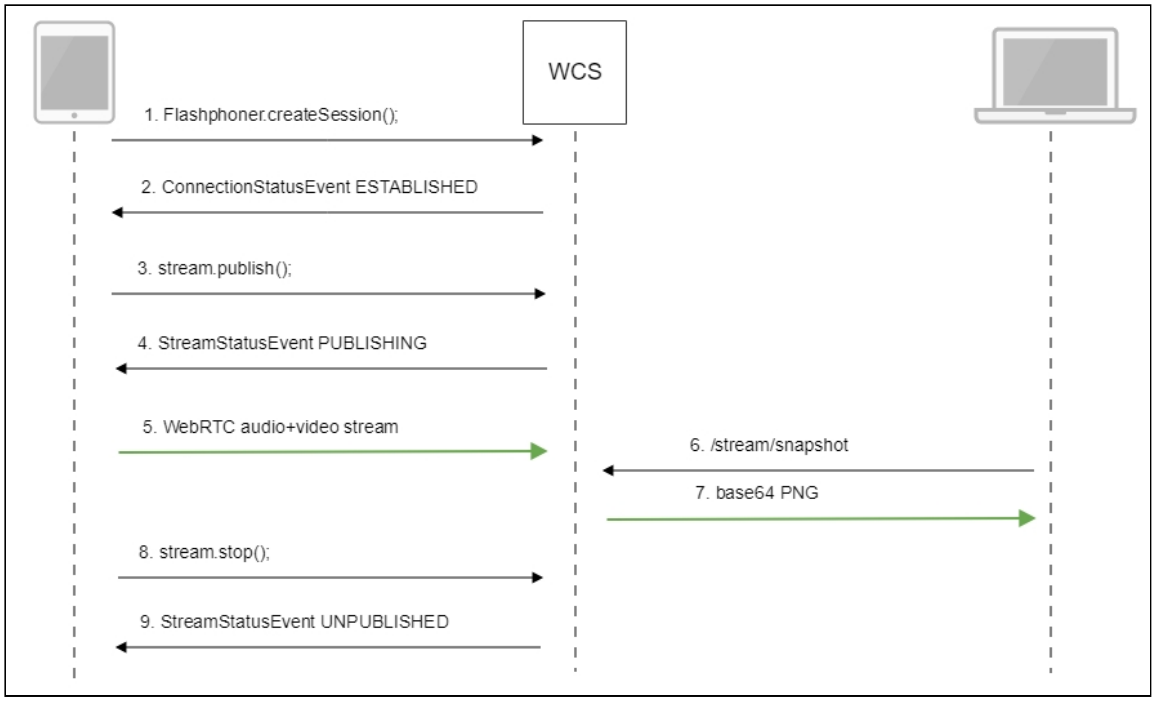
10. Here is the snapshot we have received:

Call flow

Below is the call flow when using the Stream Snapshot example to publish the stream and take a snapshot

stream-snapshot.html

stream-snapshot.js



1. Establishing a connection to the server
   `Flashphoner.createSession()` code

```
        Flashphoner.createSession({urlServer:
url}).on(SESSION_STATUS.ESTABLISHED, function(session){
            ...
        });
```

2. Receiving from the server and event confirming successful connection `SESSION_STATUS.ESTABLISHED` code

```
        Flashphoner.createSession({urlServer:
url}).on(SESSION_STATUS.ESTABLISHED, function(session){
            //session connected, start streaming
            startStreaming(session);
        }).on(SESSION_STATUS.DISCONNECTED, function(){
            ...
        }).on(SESSION_STATUS.FAILED, function(){
            ...
        });
```

3. Publishing the stream `stream.publish()` code

```
    session.createStream({
        name: streamName,
        display: localVideo,
        cacheLocalResources: true,
        receiveVideo: false,
        receiveAudio: false
        ...
    }).publish();
```

4. Receiving from the server an event confirming successful publishing of the stream `STREAM_STATUS.PUBLISHING` code

```
     session.createStream({
        name: streamName,
        display: localVideo,
        cacheLocalResources: true,
        receiveVideo: false,
        receiveAudio: false
    }).on(STREAM_STATUS.PUBLISHING, function(publishStream){
        setStatus(STREAM_STATUS.PUBLISHING);
        onPublishing(publishStream);
    }).on(STREAM_STATUS.UNPUBLISHED, function(){
        ...
    }).on(STREAM_STATUS.FAILED, function(stream){
        ...
    }).publish();
```

5. Sending the audio and video stream via WebRTC
6. Taking a snapshot of the broadcast. A new stream is created from the published one specially to take a snapshot

`stream.snapshot()` code

```
function snapshot(name) {
    setSnapshotStatus();
    var session = Flashphoner.getSessions()[0];
    session.createStream({name: name}).on(STREAM_STATUS.SNAPSHOT_COMPLETE,
function(stream){
        console.log("Snapshot complete");
        setSnapshotStatus(STREAM_STATUS.SNAPSHOT_COMPLETE);
        snapshotImg.src = ""+stream.getInfo();
        //remove failed callback
        stream.on(STREAM_STATUS.FAILED, function(){});
        //release stream object
        stream.stop();
    }).on(STREAM_STATUS.FAILED, function(stream){
        setSnapshotStatus(STREAM_STATUS.FAILED);
        console.log("Snapshot failed, info: " + stream.getInfo());
    }).snapshot();
}
```

7. Stopping publishing the stream
   `stream.stop()` code

```
function onPublishing(stream) {
    $("#publishBtn").text("Stop").off('click').click(function(){
        $(this).prop('disabled', true);
        stream.stop();
    }).prop('disabled', false);
    ...
}
```

8. Receiving from the server an event confirming unpublishing the stream
   `STREAM_STATUS.UNPUBLISHED` code

```
     session.createStream({
        name: streamName,
        display: localVideo,
        cacheLocalResources: true,
        receiveVideo: false,
        receiveAudio: false
    }).on(STREAM_STATUS.PUBLISHING, function(publishStream){
        ...
    }).on(STREAM_STATUS.UNPUBLISHED, function(){
        setStatus(STREAM_STATUS.UNPUBLISHED);
        //enable start button
        onUnpublished();
    }).on(STREAM_STATUS.FAILED, function(stream){
        ...
    }).publish();
```

Automatic stream snapshot taking

If necessary, snapshots for every stream published of supported format can be taken automatically. This feature can be enabled with the following parameter in flashphoner.properties file

```
snapshot_auto_enabled=true
```

Snapshot pictures placement can be set with the following parameter

```
snapshot_auto_dir=/usr/local/FlashphonerWebCallServer/snapshots
```

In this folder, subfolder will be created for every stream. The subfolders name is formed from stream mediasession identifier (by default)

```
snapshot_auto_naming=mediaSessionId
```

or stream name

```
snapshot_auto_naming=streamName
```

Snapshot pictures are consistently numbered and are created periodically with the following setting

```
snapshot_auto_rate=30
```

In this case, snapshot will be created from every 30 frame.

To save disk space, snspshot pictures amount can be limited using the following parameter

```
snapshot_auto_retention=20
```

In this case, last 20 snapshot pictures will be stored in stream subfolder.

Snapshot pictures numeration will be continued if stream with same name vis published.