SFU Player

The example shows how to play a number of streams in one WebRTC connection with simulcast. A room is considered to be a publishing unit, that is, viewers who connect to this room receive all the streams published in it.

On the screenshots below:

- `Server url` - Websocket URL of WCS server
- `Room name` - room name
- `Player` - viewer user name
- `360p, 720p, 180p send` - quality switch buttons

.

Note that audio tracks are playing in a separate `audio` tags.

Example source code

The source code consists of the following modules:

- `player.html` - HTML page
- `player.css` - HTML page styles
- `player.js` - main application logic
- `config.json` - client configuration file, contains streams publishing description

Analyzing the code

To analyze the example source code, take the file `player.js` version with hash `7bd8412` available here

1. Local variables

Local variables declaration to work with constants, SFU SDK, to display video and to work with client configuration

code

```javascript
const constants = SFU.constants;
const sfu = SFU;
let mainConfig;
let remoteDisplay;
let playState;
const PLAY = "play";
const STOP = "stop";
const PRELOADER_URL="../commons/media/silence.mp3"
```

2. Default configuration

Default room configuration to use if there is no `config.json` file found

code

```javascript
const defaultConfig = {
    room: {
        url: "ws://127.0.0.1:8080",
        name: "ROOM1",
        pin: "1234",
        nickName: "User1"
    }
};
```

3. Object to store current publishing/playback state

The object should keep Websocket session data, WebRTC connection data and room data, and shoukd form HTML tags ids to access them from code.

code

```javascript
const CurrentState = function(prefix) {
    let state = {
        prefix: prefix,
        pc: null,
        session: null,
        room: null,
        set: function(pc, session, room) {
            state.pc = pc;
            state.session = session;
            state.room = room;
        },
        clear: function() {
            state.room = null;
            state.session = null;
            state.pc = null;
        },
        buttonId: function() {
            return state.prefix + "Btn";
        },
        buttonText: function() {
            return (state.prefix.charAt(0).toUpperCase() +
```

```
state.prefix.slice(1));
        },
        inputId: function() {
            return state.prefix + "Name";
        },
        statusId: function() {
            return state.prefix + "Status";
        },
        formId: function() {
            return state.prefix + "Form";
        },
        errInfoId: function() {
            return state.prefix + "ErrorInfo";
        },
        is: function(value) {
            return (prefix === value);
        }
    };
    return state;
}
```

4. Initialization

`init()` code

The `init()` function is called on page load and:

- initializes state objects
- reads `config.json` file or default configuration
- initializes input fields

```
const init = function() {
    let configName = getUrlParam("config") || "./config.json";
    ...
    playState = CurrentState(PLAY);
    $.getJSON(configName, function(cfg){
        mainConfig = cfg;
        onDisconnected(playState);
    }).fail(function(e){
        //use default config
        console.error("Error reading configuration file " + configName + ": "
+ e.status + " " + e.statusText)
        console.log("Default config will be used");
        mainConfig = defaultConfig;
        onDisconnected(playState);
    });
    $("#url").val(setURL());
    $("#roomName").val("ROOM1-"+createUUID(4));
    $("#playName").val("Player1-"+createUUID(4));
}
```

5. Establishing server connection

`connect()`, `SFU.createRoom()` code

The `connect()` function is called by `Publish` or `Play` click:

- creates `PeerConnection` object
- cleans the previous session state displayed
- sets up room configuration and creates Websocket session
- subscribes to Websocket session events

```
const connect = function(state) {
    //create peer connection
    pc = new RTCPeerConnection();
    //get config object for room creation
    const roomConfig = getRoomConfig(mainConfig);
    roomConfig.pc = pc;
    roomConfig.url = $("#url").val();
    roomConfig.roomName = $("#roomName").val();
    roomConfig.nickname = $("#" + state.inputId()).val();
    // clean state display items
    setStatus(state.statusId(), "");
    setStatus(state.errInfoId(), "");
    // connect to server and create a room if not
    const session = sfu.createRoom(roomConfig);
    session.on(constants.SFU_EVENT.CONNECTED, function(room) {
        state.set(pc, session, room);
        onConnected(state);
        setStatus(state.statusId(), "ESTABLISHED", "green");
    }).on(constants.SFU_EVENT.DISCONNECTED, function() {
        state.clear();
        onDisconnected(state);
        setStatus(state.statusId(), "DISCONNECTED", "green");
    }).on(constants.SFU_EVENT.FAILED, function(e) {
        state.clear();
        onDisconnected(state);
        setStatus(state.statusId(), "FAILED", "red");
        setStatus(state.errInfoId(), e.status + " " + e.statusText, "red");
    });
}
```

6. Playback start after session establishing

`onConnected()` code

The `onConnected()` function:

- sets up `Stop` button click actions
- subscribes to room error events
- calls playback function

```
const onConnected = function(state) {
    $("#" + state.buttonId()).text("Stop").off('click').click(function () {
        onStopClick(state);
    }).prop('disabled', false);
    ...
    // Add errors displaying
    state.room.on(constants.SFU_ROOM_EVENT.FAILED, function(e) {
        setStatus(state.errInfoId(), e, "red");
    }).on(constants.SFU_ROOM_EVENT.OPERATION_FAILED, function (e) {
        setStatus(state.errInfoId(), e.operation + " failed: " + e.error,
"red");
    });
    playStreams(state);
}
```

7. Streams playback

`playStreams()`, `SFURoom.join()` code

The `playStreams()` function:

- initializes a base container tag to display incoming media streams
- joins to the room on server

```
const playStreams = function(state) {
    //create remote display item to show remote streams
    remoteDisplay = initRemoteDisplay(document.getElementById("remoteVideo"),
state.room, state.pc);
    state.room.join();
}
```

8. Playback stopping

`stopStreams()`, `remoteDisplay.stop()` code

```
const stopStreams = function(state) {
    if (remoteDisplay) {
        remoteDisplay.stop();
    }
}
```

9. Play click action

`onStartClick()`, `playFirstSound()`, `connect()` code

- validates input fields
- in Safari browser, calls `playFirstSound()` before playback to automatically play incoming audio

- calls `connect()` function

```
const onStartClick = function(state) {
    if (validateForm("connectionForm") && validateForm(state.formId())) {
        $("#" + state.buttonId()).prop('disabled', true);
        if (state.is(PLAY) && Browser().isSafariWebRTC()) {
            playFirstSound(document.getElementById("main"),
PRELOADER_URL).then(function () {
                connect(state);
            });
        } else {
            connect(state);
        }
    }
}
```

10. Stop click actions

`onStopClick()`, `Session.disconnect()` code

The `onStopClick()` function:

- stops playback
- disconects Websocket session

```
const onStopClick = function(state) {
    $("#" + state.buttonId()).prop('disabled', true);
    stopStreams(state);
    state.session.disconnect();
}
```

11. Websocket session disconnection actions

`onDisconnected()` code

The `onDisconnected()` functions:

- sets up `Play` click actions
- enables `Server url` and `Room name` fields access, if there's no parallel session

```
const onDisconnected = function(state) {
    $("#" +
state.buttonId()).text(state.buttonText()).off('click').click(function () {
        onStartClick(state);
    }).prop('disabled', false);
    $('#url').prop('disabled', false);
    $("#roomName").prop('disabled', false);
    $("#" + state.inputId()).prop('disabled', false);
}
```