

# display.js

The functions to create and destroy HTML5 tags to capture and display video and audio are moved to `display.js` module

## Analyzing the source code

To analyze the source code take the `display.js` module version available [here](#)

### Local video capturing and displaying

#### 1. Initialization

`initLocalDisplay()` [code](#)

The `initLocalDisplay()` returns the object to work with HTML5 tags to capture and display local video and audio

```
const initLocalDisplay = function (localDisplayElement) {
  const localDisplayDiv = localDisplayElement;
  const localDisplays = {};

  const removeLocalDisplay = function (id) {
    ...
  }

  const getAudioContainer = function () {
    ...
  }

  const onMuteClick = function (button, stream, type) {
    ...
  }

  const add = function (id, name, stream, type) {
    ...
  }

  const stop = function () {
    ...
  }

  const audioStateText = function (stream) {
    ...
  }

  return {
```

```
    add: add,  
    stop: stop  
  }  
}
```

## 2. Adding HTML tags to capture and display local video/audio

### 2.1. ADD AUDIO TRACK TO HTML5 VIDEO TAG

`add()` code

Where:

- audio track is added to video tag
- `onended` event handler is added to audio track
- click event handler for the audio mute/unmute button is added

```
if (stream.getAudioTracks().length > 0) {  
  let videoElement = getAudioContainer();  
  if (videoElement) {  
    let track = stream.getAudioTracks()[0];  
    videoElement.video.srcObject.addTrack(track);  
    videoElement.audioStateDisplay.innerHTML = audioStateText(stream) + "  
" + type;  
    videoElement.audioStateDisplay.addEventListener("click", function ()  
{  
      onMuteClick(videoElement.audioStateDisplay, stream, type);  
    });  
    track.addEventListener("ended", function () {  
      videoElement.video.srcObject.removeTrack(track);  
      videoElement.audioStateDisplay.innerHTML = "No audio";  
      //check video element has no tracks left  
      for (const [key, vTrack] of  
Object.entries(videoElement.video.srcObject.getTracks())) {  
        if (vTrack.readyState !== "ended") {  
          return;  
        }  
      }  
      removeLocalDisplay(videoElement.id);  
    });  
    return;  
  }  
}
```

### 2.2. CONTAINER CREATION TO DISPLAY LOCAL VIDEO

`add()` code

Where:

- container `div` tag to display local video is created
- `div` tag to display video information is created

```
const coreDisplay = createContainer(null);
coreDisplay.id = stream.id;
const publisherNameDisplay = createInfoDisplay(coreDisplay, name + " " +
type);
```

### 2.3. BUTTON CREATION TO MUTE/UNMUTE LOCAL AUDIO

`add()` code

Where:

- button to mute/unmute local audio is created

```
const audioStateDisplay = document.createElement("button");
coreDisplay.appendChild(audioStateDisplay);
```

### 2.4. TAG CREATION TO DISPLAY LOCAL VIDEO

`add()` code

Where:

- container tag which can be resized to a parent node is created
- HTML5 video tag is created (considering Safari publishing)

```
const streamDisplay = createContainer(coreDisplay);
streamDisplay.id = "stream-" + id;
const video = document.createElement("video");
video.muted = true;
if(Browser().isSafariWebRTC()) {
    video.setAttribute("playsinline", "");
    video.setAttribute("webkit-playsinline", "");
}
streamDisplay.appendChild(video);
video.srcObject = stream;
```

### 2.5. VIDEO TAG EVENT HANDLERS CREATION

`add()` code

Where:

- local video playback is started
- `onended` event handler is set up for video track
- `onresize` event handler is set up for local video to adjust video displaying size to the container dimensions

```
video.onloadedmetadata = function (e) {
    video.play();
};
stream.getTracks().forEach(function(track){
```

```

    track.addEventListener("ended", function() {
        video.srcObject.removeTrack(track);
        //check video element has no tracks left
        for (const [key, vTrack] of
Object.entries(video.srcObject.getTracks())) {
            if (vTrack.readyState !== "ended") {
                return;
            }
        }
        removeLocalDisplay(id);
    });
});
if (stream.getVideoTracks().length > 0) {
    // Resize only if video displayed
    video.addEventListener('resize', function (event) {
        publisherNameDisplay.innerHTML = name + " " + type + " " +
video.videoWidth + "x" + video.videoHeight;
        resizeVideo(event.target);
    });
} else {
    // Hide audio only container
    hideItem(streamDisplay);
    // Set up mute button for audio only stream
    audioStateDisplay.innerHTML = audioStateText(stream) + " " + type;
    audioStateDisplay.addEventListener("click", function() {
        onMuteClick(audioStateDisplay, stream, type);
    });
}
}

```

## 2.6. VIDEO CONTAINER ADDITION TO HTML PAGE

`add()` code

```

localDisplays[id] = coreDisplay;
localDisplayDiv.appendChild(coreDisplay);
return coreDisplay;

```

## 3. Stop video and audio capturing

`stop()` code

```

const stop = function () {
    for (const [key, value] of Object.entries(localDisplays)) {
        removeLocalDisplay(value.id);
    }
}

```

## Room streams published displaying

### 1. Initialization

`initRemoteDisplay()` code

The `initRemoteDisplay()` function returns the object to work with HTML5 tags to display remote video and audio streams

```
/*
display options:
autoAbr      - choose abr by default
quality      - show quality buttons
showAudio    - show audio elements
*/
const initRemoteDisplay = function (room, div, displayOptions, abrOptions,
meetingController, meetingModel, meetingView, participantFactory) {
  // Validate options first
  if (!div) {
    throw new Error("Main div to place all the media tag is not
defined");
  }
  if (!room) {
    throw new Error("Room is not defined");
  }

  const dOptions = displayOptions || {quality: true, type: true, showAudio:
false};
  let abrFactory;
  if (abrOptions) {
    abrFactory = abrManagerFactory(room, abrOptions);
  }
  participantFactory.abrFactory = abrFactory;
  participantFactory.displayOptions = dOptions;
  return meetingController(room, meetingModel(meetingView(div),
participantFactory));
}
```

## 2. Objects factory creation to control WebRTC ABR playback

`abrManagerFactory()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      ...
      return abr;
    }
  }
}
```

### 2.1. ABR MANAGER OBJECT INITIALIZING

`createAbrManager()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        track: null,
        interval: abrOptions.interval,

```

```

        thresholds: abrOptions.thresholds,
        qualities: [],
        currentQualityName: null,
        statTimer: null,
        paused: false,
        manual: false,
        keepGoodTimeout: abrOptions.abrKeepOnGoodQuality,
        keepGoodTimer: null,
        tryUpperTimeout: abrOptions.abrTryForUpperQuality,
        tryUpperTimer: null,
        ...
    }
    return abr;
}
}
}
}

```

## 2.2. START AUTOMATIC ABR QUALITY SELECTION

`abr.start()` code

```

const abrManagerFactory = function (room, abrOptions) {
    return {
        createAbrManager: function () {
            let abr = {
                ...
                start: function () {
                    this.stop();
                    console.log("Start abr interval")
                    if (abr.interval) {
                        const thresholds = Thresholds();
                        for (const threshold of abr.thresholds) {
                            thresholds.add(threshold.parameter,
threshold.maxLeap);
                        }
                        abr.statsTimer = setInterval(() => {
                            if (abr.track) {
                                room.getStats(abr.track.track,
constants.SFU_RTC_STATS_TYPE.INBOUND, (stats) => {
                                    if (thresholds.isReached(stats)) {
                                        abr.shiftDown();
                                    } else {
                                        abr.useGoodQuality();
                                    }
                                }
                            }
                        }, abr.interval);
                    }
                },
                ...
            }
            return abr;
        }
    }
}
}
}
}

```

### 2.3. STOP AUTOMATIC ABR QUALITY SELECTION

`abr.stop()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        stop: function () {
          console.log("Stop abr interval")
          abr.stopKeeping();
          abr.stopTrying();
          if (abr.statsTimer) {
            clearInterval(abr.statsTimer);
            abr.statsTimer = null;
          }
        },
        ...
      }
      return abr;
    }
  }
}
```

### 2.4. SETTING VIDEO TRACK INFO

`abr.setTrack()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        setTrack: function (track) {
          abr.track = track;
        },
        ...
      }
      return abr;
    }
  }
}
```

### 2.5. ABR QUALITY ADDITION TO SELECTION LIST

`abr.addQuality()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        addQuality: function (name) {
          abr.qualities.push({name: name, available: false, good:

```

```

true});
    },
    ...
  }
  return abr;
}
}
}

```

## 2.6. SET ABR QUALITY AS AVAILABLE TO SELECT

`abr.setQualityAvailable()` [code](#)

```

const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        setQualityAvailable: function (name, available) {
          for (let i = 0; i < abr.qualities.length; i++) {
            if (name === abr.qualities[i].name) {
              abr.qualities[i].available = available;
            }
          }
        },
        ...
      }
      return abr;
    }
  }
}

```

## 2.7. SET GOODNESS FOR CURRENT ABR QUALITY

`abr.setQualityGood()` [code](#)

```

const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        setQualityGood: function (name, good) {
          if (name) {
            for (let i = 0; i < abr.qualities.length; i++) {
              if (name === abr.qualities[i].name) {
                abr.qualities[i].good = good;
              }
            }
          }
        },
        ...
      }
      return abr;
    }
  }
}

```



---

## 2.8. GET THE FIRST AVAILABLE QUALITY

`abr.getFirstAvailableQuality()` [code](#)

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        getFirstAvailableQuality: function () {
          for (let i = 0; i < abr.qualities.length; i++) {
            if (abr.qualities[i].available) {
              return abr.qualities[i];
            }
          }
          return null;
        },
        ...
      }
      return abr;
    }
  }
}
```

## 2.9. GET A NEXT LOWER QUALITY

`abr.getLowerQuality()` [code](#)

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        getLowerQuality: function (name) {
          let quality = null;
          if (!name) {
            // There were no switching yet, return a first
            available quality
            return abr.getFirstAvailableQuality();
          }
          let currentIndex = abr.qualities.map(item =>
            item.name).indexOf(name);
          for (let i = 0; i < currentIndex; i++) {
            if (abr.qualities[i].available) {
              quality = abr.qualities[i];
            }
          }
          return quality;
        },
        ...
      }
      return abr;
    }
  }
}
```

## 2.10. GET A NEXT UPPER QUALITY

`abr.getUpperQuality()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        getUpperQuality: function (name) {
          let quality = null;
          if (!name) {
            // There were no switching yet, return a first
            available quality
            return abr.getFirstAvailableQuality();
          }
          let currentIndex = abr.qualities.map(item =>
            item.name).indexOf(name);
          for (let i = currentIndex + 1; i < abr.qualities.length;
            i++) {
            if (abr.qualities[i].available) {
              quality = abr.qualities[i];
              break;
            }
          }
          return quality;
        },
        ...
      }
      return abr;
    }
  }
}
```

## 2.11. SWITCH TO A LOWER QUALITY

`abr.shiftDown()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        shiftDown: function () {
          if (!abr.manual && !abr.paused) {
            abr.stopKeeping();
            abr.setQualityGood(abr.currentQualityName, false);
            let quality =
            abr.getLowerQuality(abr.currentQualityName);
            if (quality) {
              console.log("Switching down to " + quality.name +
              " quality");
              abr.setQuality(quality.name);
            }
          }
        },
        ...
      }
    }
  }
}
```

```

        ...
    }
    return abr;
}
}
}

```

## 2.12. SWITCH TO AN UPPER QUALITY

`abr.shiftUp()` code

```

const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        shiftUp: function () {
          if (!abr.manual && !abr.paused) {
            let quality =
abr.getUpperQuality(abr.currentQualityName);
            if (quality) {
              if (quality.good) {
                console.log("Switching up to " + quality.name
+ " quality");
                abr.setQuality(quality.name);
              } else {
                abr.tryUpper();
              }
            }
          }
        },
        ...
      }
      return abr;
    }
  }
}

```

## 2.13. USE A CURRENT QUALITY AS THE GOOD

`abr.useGoodQuality()` code

```

const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        useGoodQuality: function () {
          if (!abr.manual && !abr.paused) {
            if (!abr.currentQualityName) {
              let quality = abr.getFirstAvailableQuality();
              abr.currentQualityName = quality.name;
            }
            abr.setQualityGood(abr.currentQualityName, true);
            abr.keepGoodQuality();
          }
        }
      }
    }
  }
}

```

```

        },
        ...
    }
    return abr;
}
}
}

```

## 2.14. SET A TIMEOUT TO KEEP ON THE LAST GOOD QUALITY

`abr.keepGoodQuality()` [code](#)

```

const abrManagerFactory = function (room, abrOptions) {
    return {
        createAbrManager: function () {
            let abr = {
                ...
                keepGoodQuality: function () {
                    if (abr.keepGoodTimeout && !abr.keepGoodTimer &&
abr.getUpperQuality(abr.currentQualityName)) {
                        console.log("start keepGoodTimer");
                        abr.keepGoodTimer = setTimeout(() => {
                            abr.shiftUp();
                            abr.stopKeeping();
                        }, abr.keepGoodTimeout);
                    }
                },
                ...
            }
            return abr;
        }
    }
}
}

```

## 2.15. STOP KEEPING THE CURRENT QUALITY

`abr.stopKeeping()` [code](#)

```

const abrManagerFactory = function (room, abrOptions) {
    return {
        createAbrManager: function () {
            let abr = {
                ...
                stopKeeping: function () {
                    if (abr.keepGoodTimer) {
                        clearTimeout(abr.keepGoodTimer);
                        abr.keepGoodTimer = null;
                    }
                },
                ...
            }
            return abr;
        }
    }
}
}

```

## 2.16. TRY TO PLAY AN UPPER QUALITY

`abr.tryUpper()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        tryUpper: function () {
          let quality =
abr.getUpperQuality(abr.currentQualityName);
          if (abr.tryUpperTimeout && !abr.tryUpperTimer && quality)
{
              abr.tryUpperTimer = setTimeout(() => {
                abr.setQualityGood(quality.name, true);
                abr.stopTrying();
              }, abr.tryUpperTimeout);
            }
          },
          ...
        }
      return abr;
    }
  }
}
```

## 2.17. STOP TRYING THE UPPER QUALITY

`abr.stopTrying()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
    createAbrManager: function () {
      let abr = {
        ...
        stopTrying: function () {
          if (abr.tryUpperTimer) {
            clearTimeout(abr.tryUpperTimer);
            abr.tryUpperTimer = null;
          }
        },
        ...
      }
      return abr;
    }
  }
}
```

## 2.18. SWITCH TO A SELECTED QUALITY

`abr.setQuality()` code

```
const abrManagerFactory = function (room, abrOptions) {
  return {
```

```

    createAbrManager: function () {
      let abr = {
        ...
        setQuality: async function (name) {
          console.log("set quality name");
          // Pause switching until a new quality is received
          abr.pause();
          abr.currentQualityName = name;
          abr.track.setPreferredQuality(abr.currentQualityName);
        }
      }
      return abr;
    }
  }
}

```

### 3. Meeting room controller creation

`createDefaultMeetingController()` code

```

const createDefaultMeetingController = function (room, meetingModel) {
  ...

  return {
    stop: stop
  }
}

```

#### 3.1. PARTICIPANT\_LIST EVENT HANDLING

`createDefaultMeetingController()` code

```

const createDefaultMeetingController = function (room, meetingModel) {
  const constants = SFU.constants;
  room.on(constants.SFU_ROOM_EVENT.PARTICIPANT_LIST, async function (e) {
    for (const idName of e.participants) {
      meetingModel.addParticipant(idName.userId, idName.name);
    }
    ...
  });
  ...
}

```

#### 3.2. JOINED EVENT HANDLING

`createDefaultMeetingController()` code

```

const createDefaultMeetingController = function (room, meetingModel) {
  const constants = SFU.constants;
  room.on(constants.SFU_ROOM_EVENT.PARTICIPANT_LIST, async function (e) {
    ...
  }).on(constants.SFU_ROOM_EVENT.JOINED, async function (e) {
    meetingModel.addParticipant(e.userId, e.name);
    ...
  });
}

```

```
    ...  
  }
```

### 3.3. LEFT EVENT HANDLING

`createDefaultMeetingController()` code

```
const createDefaultMeetingController = function (room, meetingModel) {  
  const constants = SFU.constants;  
  room.on(constants.SFU_ROOM_EVENT.PARTICIPANT_LIST, async function (e) {  
    ...  
  }).on(constants.SFU_ROOM_EVENT.LEFT, function (e) {  
    meetingModel.removeParticipant(e.userId);  
    ...  
  });  
  ...  
}
```

### 3.4. ADD\_TRACKS EVENT HANDLING

`createDefaultMeetingController()` code

```
const createDefaultMeetingController = function (room, meetingModel) {  
  const constants = SFU.constants;  
  room.on(constants.SFU_ROOM_EVENT.PARTICIPANT_LIST, async function (e) {  
    ...  
  }).on(constants.SFU_ROOM_EVENT.ADD_TRACKS, async function (e) {  
    meetingModel.addTracks(e.info.userId, e.info.info);  
    ...  
  });  
  ...  
}
```

### 3.5. REMOVE\_TRACKS EVENT HANDLING

`createDefaultMeetingController()` code

```
const createDefaultMeetingController = function (room, meetingModel) {  
  const constants = SFU.constants;  
  room.on(constants.SFU_ROOM_EVENT.PARTICIPANT_LIST, async function (e) {  
    ...  
  }).on(constants.SFU_ROOM_EVENT.REMOVE_TRACKS, async function (e) {  
    meetingModel.removeTracks(e.info.userId, e.info.info);  
    ...  
  });  
  ...  
}
```

### 3.6. TRACK\_QUALITY\_STATE EVENT HANDLING

`createDefaultMeetingController()` code

```
const createDefaultMeetingController = function (room, meetingModel) {  
  const constants = SFU.constants;
```

```

    room.on(constants.SFU_ROOM_EVENT.PARTICIPANT_LIST, async function (e) {
      ...
    }).on(constants.SFU_ROOM_EVENT.TRACK_QUALITY_STATE, async function (e) {
      meetingModel.updateQualityInfo(e.info.userId, e.info.tracks);
      ...
    });
    ...
  }

```

### 3.7. ENDED EVENT HANDLING

`createDefaultMeetingController()` code

```

const createDefaultMeetingController = function (room, meetingModel) {
  const constants = SFU.constants;
  room.on(constants.SFU_ROOM_EVENT.PARTICIPANT_LIST, async function (e) {
    ...
  }).on(constants.SFU_ROOM_EVENT.ENDED, function (e) {
    meetingModel.end();
  });
  ...
}

```

### 3.8. STOPPING THE MEETING ROOM CONTROL

`createDefaultMeetingController()` code

```

const createDefaultMeetingController = function (room, meetingModel) {
  ...
  const stop = function () {
    meetingModel.end();
  };

  return {
    stop: stop
  }
}

```

## 4. Meeting room model creation

`createDefaultMeetingModel()` code

```

const createDefaultMeetingModel = function (meetingView, participantFactory,
displayOptions, abrFactory) {
  ...
}

```

### 4.1. ADD A PARTICIPANT

`addParticipant()` code

```

const createDefaultMeetingModel = function (meetingView, participantFactory,
displayOptions, abrFactory) {

```



```

return {
  ...
  addParticipant: function (userId, participantName) {
    if (this.participants.get(userId)) {
      return;
    }
    const [participantModel, participantView, participant] =
participantFactory.createParticipant(userId, participantName, displayOptions,
abrFactory);
    this.participants.set(userId, participant);
    meetingView.addParticipant(userId, participantName,
participantView.rootDiv);
  },
  ...
}
}

```

#### 4.2. REMOVE A PARTICIPANT

`removeParticipant()` [code](#)

```

const createDefaultMeetingModel = function (meetingView, participantFactory,
displayOptions, abrFactory) {
  return {
    ...
    removeParticipant: function (userId) {
      const participant = this.participants.get(userId);
      if (participant) {
        this.participants.delete(userId);
        meetingView.removeParticipant(userId);
        participant.dispose();
      }
    },
    ...
  }
}

```

#### 4.3. RENAME A PARTICIPANT

`renameParticipant()` [code](#)

```

const createDefaultMeetingModel = function (meetingView, participantFactory,
displayOptions, abrFactory) {
  return {
    ...
    renameParticipant: function (userId, newNickname) {
      const participant = this.participants.get(userId);
      if (participant) {
        participant.setNickname(newNickname);
      }
    },
    ...
  }
}

```

#### 4.4. ADD PARTICIPANTS TRACKS TO DISPLAY

`addTracks()` code

```
const createDefaultMeetingModel = function (meetingView, participantFactory,
displayOptions, abrFactory) {
  return {
    ...
    addTracks: function (userId, tracks) {
      const participant = this.participants.get(userId);
      if (!participant) {
        return;
      }

      for (const track of tracks) {
        if (track.type === "VIDEO") {
          participant.addVideoTrack(track);
        } else if (track.type === "AUDIO") {
          participant.addAudioTrack(track);
        }
      }
    },
    ...
  }
}
```

#### 4.5. REMOVE PARTICIPANTS TRACKS DISPLAYED

`removeTracks()` code

```
const createDefaultMeetingModel = function (meetingView, participantFactory,
displayOptions, abrFactory) {
  return {
    ...
    removeTracks: function (userId, tracks) {
      const participant = this.participants.get(userId);
      if (!participant) {
        return;
      }
      for (const track of tracks) {
        if (track.type === "VIDEO") {
          participant.removeVideoTrack(track);
        } else if (track.type === "AUDIO") {
          participant.removeAudioTrack(track);
        }
      }
    },
    ...
  }
}
```

#### 4.6. UPDATE PARTICIPANTS TRACKS QUALITY INFO

`updateQualityInfo()` code

```

const createDefaultMeetingModel = function (meetingView, participantFactory,
displayOptions, abrFactory) {
  return {
    ...
    updateQualityInfo: function (userId, tracksInfo) {
      const participant = this.participants.get(userId);
      if (!participant) {
        return;
      }
      participant.updateQualityInfo(tracksInfo);
    },
    ...
  }
}

```

#### 4.7. STOP PARTICIPANTS DISPLAYING WHEN STOPPING THE ROOM

`end()` code

```

const createDefaultMeetingModel = function (meetingView, participantFactory,
displayOptions, abrFactory) {
  return {
    ...
    end: function () {
      console.log("Meeting " + this.meetingName + " ended");
      meetingView.end();
      this.participants.forEach((participant, id) => {
        participant.dispose();
      });
      this.participants.clear();
    },
    ...
  }
}

```

#### 4.8. RENAME THE ROOM

`setMeetingName()` code

```

const createDefaultMeetingModel = function (meetingView, participantFactory,
displayOptions, abrFactory) {
  return {
    ...
    setMeetingName: function (id) {
      this.meetingName = id;
      meetingView.setMeetingName(id);
    }
  }
}

```

### 5. Meeting room streams view object creation

`createDefaultMeetingView()` code

```
const createDefaultMeetingView = function (entryPoint) {
  ...
}
```

### 5.1. HTML5 TAGS INITIALIZING

`createDefaultMeetingView()` code

```
const createDefaultMeetingView = function (entryPoint) {
  const rootDiv = document.createElement("div");
  rootDiv.setAttribute("class", "grid-item");
  entryPoint.appendChild(rootDiv);
  const title = document.createElement("label");
  title.setAttribute("style", "display:block; border: solid; border-width:
1px");
  rootDiv.appendChild(title);
  return {
    ...
  }
}
```

### 5.2. PARTICIPANTS VIEWS INITIALIZING

`participantViews()` code

```
const createDefaultMeetingView = function (entryPoint) {
  ...
  return {
    participantViews: new Map(),
    ...
  }
}
```

### 5.3. SET MEETING ROOM NAME TO DISPLAY

`setMeetingName()` code

```
const createDefaultMeetingView = function (entryPoint) {
  ...
  return {
    ...
    setMeetingName: function (id) {
      title.innerText = "Meeting: " + id;
    },
    ...
  }
}
```

### 5.4. ADDING PARTICIPANT DISPLAYING TAGS

`addParticipant()` code

```

const createDefaultMeetingView = function (entryPoint) {
  ...
  return {
    ...
    addParticipant: function (userId, participantName, cell) {
      const participantDiv = createContainer(rootDiv);
      participantDiv.appendChild(cell);
      this.participantViews.set(userId, participantDiv);
    },
    ...
  }
}

```

#### 5.5. REMOVING PARTICIPANT DISPLAYING TAGS

`removeParticipant()` code

```

const createDefaultMeetingView = function (entryPoint) {
  ...
  return {
    ...
    removeParticipant: function (userId) {
      const cell = this.participantViews.get(userId);
      if (cell) {
        this.participantViews.delete(userId);
        cell.remove();
      }
    },
    ...
  }
}

```

#### 5.6. REMOVING ROOT DIV TAG

`end()` code

```

const createDefaultMeetingView = function (entryPoint) {
  ...
  return {
    ...
    end: function () {
      rootDiv.remove();
    }
  }
}

```

### 6. Participants objects factory creation

`createParticipantFactory()` code

Here a participant model, view and controller objects are created for the certain participant

```

const createParticipantFactory = function (remoteTrackFactory,
createParticipantView, createParticipantModel) {

```

```

return {
  displayOptions: null,
  abrFactory: null,
  createParticipant: function (userId, nickname) {
    const view = createParticipantView();
    const model = createParticipantModel(userId, nickname, view,
remoteTrackFactory, this.abrFactory, this.displayOptions);
    const controller = createParticipantController(model);
    return [model, view, controller];
  }
}
}
}

```

## 7. Participant controller object creation

`createParticipantController()` [code](#)

The object calls an appropriate participant model methods

```

const createParticipantController = function (model) {
  return {
    addVideoTrack: function (track) {
      model.addVideoTrack(track);
    },
    removeVideoTrack: function (track) {
      model.removeVideoTrack(track);
    },
    addAudioTrack: function (track) {
      model.addAudioTrack(track);
    },
    removeAudioTrack: function (track) {
      model.removeAudioTrack(track);
    },
    updateQualityInfo: function (qualityInfo) {
      model.updateQualityInfo(qualityInfo);
    },
    setNickname: function (nickname) {
      model.setNickname(nickname);
    },
    dispose: function () {
      model.dispose();
    }
  }
}
}

```

## 8. Participant model object creation for one-to-one meeting room

`createOneToOneParticipantModel()` [code](#)

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  const instance = {
    userId: userId,
    nickname: nickname,
    remoteVideoTracks: new Map(),

```

```

    remoteAudioTracks: new Map(),
    audioTracks: new Map(),
    videoTracks: new Map(),
    abrManagers: new Map(),
    disposed: false,
    dispose: async function () {
        ...
    },
    addVideoTrack: function (track) {
        ...
    },
    removeVideoTrack: function (track) {
        ...
    },
    addAudioTrack: function (track) {
        ...
    },
    removeAudioTrack: function (track) {
        ...
    },
    setUserId: function (userId) {
        ...
    },
    setNickname: function (nickname) {
        ...
    },
    updateQualityInfo: function (remoteTracks) {
        ...
    },
    requestVideoTrack: async function (track, remoteTrack) {
        ...
    },
    pickQuality: async function (track, qualityName) {
        ...
    },
    muteVideo: async function (track) {
        ...
    },
    unmuteVideo: async function (track) {
        ...
    }
  };
  instance.setUserId(userId);
  instance.setNickname(nickname);
  return instance;
}

```

## 8.1. ADDING VIDEO TRACK TO DISPLAY

`addVideoTrack()` code

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  const instance = {
    ...
    addVideoTrack: function (track) {
      this.videoTracks.set(track.mid, track);
    }
  };
}

```

```

        if (!track.quality) {
            track.quality = [];
        }
        participantView.addVideoTrack(track);
        const self = this;
        remoteTrackFactory.getVideoTrack().then((remoteTrack) => {
            if (remoteTrack) {
                if (self.disposed || !self.videoTracks.get(track.mid)) {
                    remoteTrack.dispose();
                    return;
                }

                participantView.addVideoSource(remoteTrack.track, track,
() => {

                    const abrManager = self.abrManagers.get(track.id);
                    if (!abrManager) {
                        return;
                    }
                    if (abrManager.isAuto()) {
                        abrManager.resume();
                    }
                }, (mute) => {
                    if (mute) {
                        return self.muteVideo(track);
                    } else {
                        return self.unmuteVideo(track);
                    }
                });
                self.requestVideoTrack(track, remoteTrack).then(() => {
                    participantView.showVideoTrack(track);
                }, (ex) => {
                    participantView.removeVideoSource(track);
                    remoteTrack.dispose();
                });
            }
        }, (ex) => {
            console.log("Failed to get remote track " + ex);
        });
    },
    ...
};
...
return instance;
}

```

## 8.2. REMOVING VIDEO TRACK DISPLAYED

`removeVideoTrack()` code

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
    const instance = {
        ...
        removeVideoTrack: function (track) {
            if (this.videoTracks.delete(track.mid)) {
                const remoteTrack = this.remoteVideoTracks.get(track.mid);
                if (remoteTrack) {

```



```

        this.remoteVideoTracks.delete(track.mid);
        remoteTrack.dispose();
    }
    participantView.removeVideoTrack(track);

    const abrManager = this.abrManagers.get(track.id);
    if (abrManager) {
        this.abrManagers.delete(track.id);
        abrManager.clearQualityState();
        abrManager.stop();
    }
    }
    },
    ...
};
...
return instance;
}

```

### 8.3. ADDING AUDIO TRACK TO DISPLAY

`addAudioTrack()` code

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
    const instance = {
        ...
        addAudioTrack: function (track) {
            this.audioTracks.set(track.mid, track);
            const self = this;
            remoteTrackFactory.getAudioTrack().then((remoteTrack) => {
                if (remoteTrack) {
                    if (self.disposed || !self.audioTracks.get(track.mid)) {
                        remoteTrack.dispose();
                        return;
                    }
                    this.remoteAudioTracks.set(track.mid, remoteTrack);
                    remoteTrack.demandTrack(track.id).then(() => {
                        if (!self.audioTracks.get(track.mid)) {
                            remoteTrack.dispose();
                            self.remoteAudioTracks.delete(track.mid);
                            return;
                        }
                    })
                    participantView.addAudioTrack(track,
remoteTrack.track, displayOptions.showAudio);
                }, (ex) => {
                    console.log("Failed demand track " + ex);
                    remoteTrack.dispose();
                    self.remoteAudioTracks.delete(track.mid);
                });
            });
        }
    }, (ex) => {
        console.log("Failed to get audio track " + ex);
    });
    },
    ...
};

```

```

    ...
    return instance;
}

```

#### 8.4. REMOVING AUDIO TRACK DISPLAYED

`removeAudioTrack()` code

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  const instance = {
    ...
    removeAudioTrack: function (track) {
      if (!this.audioTracks.delete(track.mid)) {
        return
      }

      participantView.removeAudioTrack(track);
      const remoteTrack = this.remoteAudioTracks.get(track.mid);
      if (remoteTrack) {
        this.remoteAudioTracks.delete(track.mid);
        remoteTrack.dispose();
      }
    },
    ...
  };
  ...
  return instance;
}

```

#### 8.5. REQUEST VIDEO TRACK TO DISPLAY

`requestVideoTrack()` code

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  const instance = {
    ...
    requestVideoTrack: async function (track, remoteTrack) {
      return new Promise((resolve, reject) => {
        if (!remoteTrack || !track) {
          reject(new Error("Remote and local track must be
defined"));
          return;
        }
        const self = this;
        remoteTrack.demandTrack(track.id).then(() => {
          if (!self.videoTracks.get(track.mid)) {
            reject(new Error("Video track already removed from
model"));
            return;
          }
        })
        let abrManager = self.abrManagers.get(track.id);

        if (abrManager) {
          abrManager.clearQualityState();
        }
      });
    },
    ...
  };
  ...
  return instance;
}

```

```

        } else if (abrFactory) {
            abrManager = abrFactory.createAbrManager();
            self.abrManagers.set(track.id, abrManager);
        }

        if (abrManager) {
            abrManager.setTrack(remoteTrack);
            abrManager.stop();
            if (track.quality.length > 0) {
                participantView.addQuality(track, "Auto", true,
async () => {
                    const manager =
self.abrManagers.get(track.id);
                    if (!manager) {
                        return;
                    }
                    manager.start();
                    manager.setAuto();
                    participantView.pickQuality(track, "Auto");
                });
                if (displayOptions.autoAbr) {
                    abrManager.setAuto();
                    abrManager.start();
                    participantView.pickQuality(track, "Auto");
                }
            }
        }
        for (const qualityDescriptor of track.quality) {
            if (abrManager) {
                abrManager.addQuality(qualityDescriptor.quality);

                abrManager.setQualityAvailable(qualityDescriptor.quality,
                qualityDescriptor.available);
            }
            if (displayOptions.quality) {
                participantView.addQuality(track,
                qualityDescriptor.quality, qualityDescriptor.available, async () => {
                    const manager =
self.abrManagers.get(track.id);
                    if (manager) {
                        manager.setManual();

                manager.setQuality(qualityDescriptor.quality);
                }
                return self.pickQuality(track,
                qualityDescriptor.quality);
            });
        }
        }
        self.remoteVideoTracks.delete(track.mid);
        self.remoteVideoTracks.set(track.mid, remoteTrack);
        resolve();
    }, (ex) => {
        reject(ex);
    });
});
},
...

```

```

};
...
return instance;
}

```

## 8.6. UPDATE AVAILABLE TRACK QUALITIES INFO

### updateQualityInfo() code

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  const instance = {
    ...
    updateQualityInfo: function (remoteTracks) {
      for (const remoteTrackQuality of remoteTracks) {
        const track = this.videoTracks.get(remoteTrackQuality.mid);
        if (!track) {
          continue;
        }
        if (!this.remoteVideoTracks.get(track.mid)) {
          // update model and return, view not changed
          for (const remoteQualityInfo of
remoteTrackQuality.quality) {
            const quality = track.quality.find((q) => q.quality
=== remoteQualityInfo.quality);
            if (quality) {
              quality.available = remoteQualityInfo.available;
            } else {
              track.quality.push(remoteQualityInfo);
            }
          }
          return;
        }
        let abrManager = this.abrManagers.get(track.id);
        if (abrManager && track.quality.length === 0 &&
remoteTrackQuality.quality.length > 0) {
          const self = this;
          participantView.addQuality(track, "Auto", true, async ()
=> {
            const manager = self.abrManagers.get(track.id);
            if (!manager) {
              return;
            }
            manager.start();
            manager.setAuto();
            participantView.pickQuality(track, "Auto");
          })
          if (displayOptions.autoAbr) {
            abrManager.setAuto();
            abrManager.start();
            participantView.pickQuality(track, "Auto");
          }
        }
        for (const remoteQualityInfo of remoteTrackQuality.quality) {
          const localQuality = track.quality.find((q) => q.quality
=== remoteQualityInfo.quality);
          if (localQuality) {

```

```

        localQuality.available = remoteQualityInfo.available;
        if (abrManager) {

abrManager.setQualityAvailable(remoteQualityInfo.quality,
remoteQualityInfo.available);
        }
        if (displayOptions.quality) {
            participantView.updateQuality(track,
localQuality.quality, localQuality.available);
        }
        } else {
            track.quality.push(remoteQualityInfo);
            if (abrManager) {
                abrManager.addQuality(remoteQualityInfo.quality);

abrManager.setQualityAvailable(remoteQualityInfo.quality,
remoteQualityInfo.available)
            }
            if (displayOptions.quality) {
                const self = this;
                participantView.addQuality(track,
remoteQualityInfo.quality, remoteQualityInfo.available, async () => {
                    const manager =
self.abrManagers.get(track.id);
                    if (manager) {
                        manager.setManual();

manager.setQuality(remoteQualityInfo.quality);
                    }
                    return self.pickQuality(track,
remoteQualityInfo.quality);
                });
            }
        }
    }
},
...
};
...
return instance;
}

```

## 8.7. PICK A QUALITY TO PLAY

`pickQuality()` code

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
    const instance = {
        ...
        pickQuality: async function (track, qualityName) {
            let remoteVideoTrack = this.remoteVideoTracks.get(track.mid);
            if (remoteVideoTrack) {
                return
remoteVideoTrack.setPreferredQuality(qualityName).then(() => {

```

```

        participantView.pickQuality(track, qualityName);
    });
    }
    },
    ...
};
...
return instance;
}

```

## 8.8. MUTE PARTICIPANT VIDEO

`muteVideo()` code

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
    const instance = {
        ...
        muteVideo: async function (track) {
            const remoteTrack = this.remoteVideoTracks.get(track.mid);
            if (remoteTrack) {
                return remoteTrack.mute();
            } else {
                return new Promise((resolve, reject) => {
                    reject(new Error("Remote track not defined"));
                });
            }
        },
        ...
    };
    ...
    return instance;
}

```

## 8.9. UNMUTE PARTICIPANT VIDEO

`unmuteVideo()` code

```

const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
    const instance = {
        ...
        unmuteVideo: async function (track) {
            const remoteTrack = this.remoteVideoTracks.get(track.mid);
            if (remoteTrack) {
                return remoteTrack.unmute();
            } else {
                return new Promise((resolve, reject) => {
                    reject(new Error("Remote track not defined"));
                });
            }
        },
        ...
    };
    ...
    return instance;
}

```

---

## 8.10. DISPOSE THE OBJECT

`dispose()` code

```
const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  const instance = {
    ...
    dispose: async function () {
      this.disposed = true;
      participantView.dispose();
      this.remoteVideoTracks.forEach((track, id) => {
        track.dispose();
      })
      this.remoteVideoTracks.clear();

      this.remoteAudioTracks.forEach((track, id) => {
        track.dispose();
      })
      this.remoteAudioTracks.clear();

      this.abrManagers.forEach((abrManager, id) => {
        abrManager.stop();
      })
      this.abrManagers.clear();

    },
  };
  ...
  return instance;
}
```

## 9. Participant model object creation for one-to-many meeting room

`createOneToManyParticipantModel()` code

```
const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  ...
  const instance = {
    userId: userId,
    nickname: nickname,
    videoEnabled: false,
    currentTrack: null,
    remoteVideoTrack: null,
    remoteAudioTracks: new Map(),
    audioTracks: new Map(),
    videoTracks: new Map(),
    abr: null,
    disposed: false,
    dispose: async function () {
      ...
    },
    addVideoTrack: function (track) {
      ...
    }
  };
  ...
  return instance;
}
```

```

    },
    removeVideoTrack: function (track) {
        ...
    },
    addAudioTrack: function (track) {
        ...
    },
    removeAudioTrack: function (track) {
        ...
    },
    setUserId: function (userId) {
        ...
    },
    setNickname: function (nickname) {
        ...
    },
    updateQualityInfo: function (remoteTracks) {
        ...
    },
    requestVideoTrack: async function (track, remoteTrack) {
        ...
    },
    pickQuality: async function (track, qualityName) {
        ...
    },
    muteVideo: async function (track) {
        ...
    },
    unmuteVideo: async function (track) {
        ...
    }
}
};
instance.setUserId(userId);
instance.setNickname(nickname);
if (abrFactory) {
    instance.abr = abrFactory.createAbrManager();
}
return instance;
}

```

### 9.1. ADDING VIDEO TRACK TO DISPLAY

`addVideoTrack()` [code](#)

```

const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
    ...
    const instance = {
        ...
        addVideoTrack: function (track) {
            this.videoTracks.set(track.mid, track);
            if (!track.quality) {
                track.quality = [];
            }
        }
        const self = this;
        participantView.addVideoTrack(track, () => {
            if (self.disposed) {

```



```

        return new Promise((resolve, reject) => {
            reject(new Error("Model disposed"));
        });
    }

    if (self.remoteVideoTrack) {
        return new Promise((resolve, reject) => {
            self.requestVideoTrack(track,
self.remoteVideoTrack).then(() => {
                resolve();
            }, (ex) => {
                reject(ex);
            });
        });
    } else {
        return new Promise((resolve, reject) => {
            reject(new Error("Remote track is null"));
            requestTrackAndPick(self, track);
        });
    }
});
requestTrackAndPick(this, track);
},
...
};
...
return instance;
}

```

## 9.2. REMOVING VIDEO TRACK DISPLAYING

`removeVideoTrack()` [code](#)

```

const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
    ...
    const instance = {
        ...
        removeVideoTrack: function (track) {
            this.videoTracks.delete(track.mid);
            participantView.removeVideoTrack(track);
            if (this.currentTrack && this.currentTrack.mid === track.mid) {
                repickTrack(this, track);
            }
        },
        ...
    };
    ...
    return instance;
}

```

## 9.3. ADDING AUDIO TRACK TO DISPLAY

`addAudioTrack()` [code](#)

```

const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  ...
  const instance = {
    ...
    addAudioTrack: async function (track) {
      this.audioTracks.set(track.mid, track);
      const self = this;
      remoteTrackFactory.getAudioTrack().then((remoteTrack) => {
        if (!remoteTrack) {
          return;
        }
        if (self.disposed || !self.audioTracks.get(track.mid)) {
          remoteTrack.dispose();
          return;
        }
        this.remoteAudioTracks.set(track.mid, remoteTrack);
        remoteTrack.demandTrack(track.id).then(() => {
          if (!self.audioTracks.get(track.mid)) {
            remoteTrack.dispose();
            self.remoteAudioTracks.delete(track.mid);
            return;
          }
          participantView.addAudioTrack(track, remoteTrack.track,
displayOptions.showAudio);
        }, (ex) => {
          console.log("Failed demand track " + ex);
          remoteTrack.dispose();
          self.remoteAudioTracks.delete(track.mid);
        });
      }, (ex) => {
        console.log("Failed to get audio track " + ex);
      });
    },
    ...
  };
  ...
  return instance;
}

```

#### 9.4. Remove audio track displayed

`removeAudioTrack()` code

```

const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  ...
  const instance = {
    ...
    removeAudioTrack: function (track) {
      if (!this.audioTracks.delete(track.mid)) {
        return
      }

      participantView.removeAudioTrack(track);
    }
  }
}

```

```

        const remoteTrack = this.remoteAudioTracks.get(track.mid);
        if (remoteTrack) {
            this.remoteAudioTracks.delete(track.mid);
            remoteTrack.dispose();
        }

        },
        ...
    };
    ...
    return instance;
}

```

## 9.5. REQUEST VIDEO TRACK TO DISPLAY

### `requestVideoTrack()` code

```

const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
    ...
    const instance = {
        ...
        requestVideoTrack: async function (track, remoteTrack) {
            return new Promise((resolve, reject) => {
                if (!remoteTrack || !track) {
                    reject(new Error("Remote and local track must be
defined"));
                }
                return;
            }
            const self = this;
            remoteTrack.demandTrack(track.id).then(() => {
                // channels reordering case, must be removed after
channels unification
                if (!self.videoTracks.get(track.mid)) {
                    reject(new Error("Video track already removed from
model"));
                }
                return;
            }
            self.currentTrack = track;
            participantView.clearQualityState(track);
            if (self.abr) {
                self.abr.stop();
                self.abr.clearQualityState();
                self.abr.setTrack(remoteTrack);

                if (track.quality.length > 0) {
                    participantView.addQuality(track, "Auto", true,
async () => {
                        if (!self.abr) {
                            return;
                        }
                        self.abr.start();
                        self.abr.setAuto();
                        participantView.pickQuality(track, "Auto");
                    })
                }
            }
            if (displayOptions.autoAbr) {

```

```

        self.abr.setAuto();
        self.abr.start();
        participantView.pickQuality(track, "Auto");
    }
}
for (const qualityDescriptor of track.quality) {
    if (self.abr) {
        self.abr.addQuality(qualityDescriptor.quality);

self.abr.setQualityAvailable(qualityDescriptor.quality,
qualityDescriptor.available);
    }
    if (displayOptions.quality) {
        participantView.addQuality(track,
qualityDescriptor.quality, qualityDescriptor.available, async () => {
            if (self.abr) {
                self.abr.setManual();

self.abr.setQuality(qualityDescriptor.quality);
            }
            return self.pickQuality(track,
qualityDescriptor.quality);
        });
    }
}
    resolve();
}, (ex) => reject(ex));
});
},
...
};
...
return instance;
}

```

## 9.6. UPDATE AVAILABLE TRACK QUALITIES INFO

`updateQualityInfo()` code

```

const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
    ...
    const instance = {
        ...
        updateQualityInfo: function (remoteTracks) {
            for (const remoteTrackQuality of remoteTracks) {
                const track = this.videoTracks.get(remoteTrackQuality.mid);
                if (!track) {
                    continue;
                }
                if (!this.currentTrack || this.currentTrack.mid !==
track.mid) {
                    // update model and return, view not changed
                    for (const remoteQualityInfo of
remoteTrackQuality.quality) {
                        const quality = track.quality.find((q) => q.quality
=== remoteQualityInfo.quality);

```

```

        if (quality) {
            quality.available = remoteQualityInfo.available;
        } else {
            track.quality.push(remoteQualityInfo);
        }
    }
    return;
}
if (this.abr && track.quality.length === 0 &&
remoteTrackQuality.quality.length > 0) {
    const self = this;
    participantView.addQuality(track, "Auto", true, async ()
=> {
        if (!self.abr) {
            return;
        }
        self.abr.start();
        self.abr.setAuto();
        participantView.pickQuality(track, "Auto");
    })
    if (displayOptions.autoAbr && this.abr) {
        this.abr.setAuto();
        this.abr.start();
        participantView.pickQuality(track, "Auto");
    }
}
for (const remoteQualityInfo of remoteTrackQuality.quality) {
    const localQuality = track.quality.find((q) => q.quality
=== remoteQualityInfo.quality);
    if (localQuality) {
        localQuality.available = remoteQualityInfo.available;
        if (this.abr) {
            this.abr.setQualityAvailable(remoteQualityInfo.quality,
remoteQualityInfo.available)
        }
        if (displayOptions.quality) {
            participantView.updateQuality(track,
localQuality.quality, localQuality.available);
        }
    } else {
        track.quality.push(remoteQualityInfo);
        if (this.abr) {
            this.abr.addQuality(remoteQualityInfo.quality);
            this.abr.setQualityAvailable(remoteQualityInfo.quality,
remoteQualityInfo.available)
        }
        if (displayOptions.quality) {
            const self = this;
            participantView.addQuality(track,
remoteQualityInfo.quality, remoteQualityInfo.available, async () => {
                if (self.abr) {
                    self.abr.setManual();
                }
            })
            self.abr.setQuality(remoteQualityInfo.quality);
        }
        return self.pickQuality(track,

```

```

remoteQualityInfo.quality);
    });
  }
}
},
...
};
...
return instance;
}

```

## 9.7. PICK QUALITY TO DISPLAY

`pickQuality()` code

```

const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  ...
  const instance = {
    ...
    pickQuality: async function (track, qualityName) {
      if (this.remoteVideoTrack) {
        return
this.remoteVideoTrack.setPreferredQuality(qualityName).then(() =>
participantView.pickQuality(track, qualityName));
      }
    },
    ...
  };
  ...
  return instance;
}

```

## 9.8. MUTE PARTICIPANT VIDEO

`muteVideo()` code

```

const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  ...
  const instance = {
    ...
    muteVideo: async function (track) {
      if (this.remoteVideoTrack) {
        return this.remoteVideoTrack.mute();
      } else {
        return new Promise((resolve, reject) => {
          reject(new Error("Remote track not defined"));
        });
      }
    },
    ...
  };
  ...
}

```

```
    return instance;
  }
```

## 9.9. UNMUTE PARTICIPANT VIDEO

`unmuteVideo()` [code](#)

```
const createOneToManyParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  ...
  const instance = {
    ...
    unmuteVideo: async function (track) {
      if (this.remoteVideoTrack) {
        return this.remoteVideoTrack.unmute();
      } else {
        return new Promise((resolve, reject) => {
          reject(new Error("Remote track not defined"));
        });
      }
    }
  };
  ...
  return instance;
}
```

## 9.10. DISPOSE THE OBJECT

`dispose()` [code](#)

```
const createOneToOneParticipantModel = function (userId, nickname,
participantView, remoteTrackFactory, abrFactory, displayOptions) {
  const instance = {
    ...
    dispose: async function () {
      this.disposed = true;
      participantView.dispose();
      this.remoteVideoTracks.forEach((track, id) => {
        track.dispose();
      })
      this.remoteVideoTracks.clear();

      this.remoteAudioTracks.forEach((track, id) => {
        track.dispose();
      })
      this.remoteAudioTracks.clear();

      this.abrManagers.forEach((abrManager, id) => {
        abrManager.stop();
      })
      this.abrManagers.clear();

    },
  };
  ...
}
```

```
    return instance;
  }
```

## 10. Participant view object creation for one-to-one meeting room

`createOneToOneParticipantView()` code

```
const createOneToOneParticipantView = function () {
  const participantDiv = createContainer(null);
  const audioDisplay = createContainer(participantDiv);
  const participantNicknameDisplay = createInfoDisplay(participantDiv,
  "Name: ")
  const videoPlayers = new Map();
  const audioElements = new Map();

  return {
    rootDiv: participantDiv,
    dispose: function () {
      ...
    },
    addVideoTrack: function (track) {
      ...
    },
    removeVideoTrack: function (track) {
      ...
    },
    addVideoSource: function (remoteVideoTrack, track, onResize,
muteHandler) {
      ...
    },
    removeVideoSource: function (track) {
      ...
    },
    showVideoTrack: function (track) {
      ...
    },
    addAudioTrack: function (track, audioTrack, show) {
      ...
    },
    removeAudioTrack: function (track) {
      ...
    },
    setNickname: function (userId, nickname) {
      ...
    },
    updateQuality: function (track, qualityName, available) {
      ...
    },
    addQuality: function (track, qualityName, available, onQualityPick) {
      ...
    },
    pickQuality: function (track, qualityName) {
      ...
    }
  }
}
```



```
}  
}  
}
```

### 10.1. ADDING VIDEO TRACK TO PLAY

`addVideoTrack()` [code](#)

```
const createOneToOneParticipantView = function () {  
  ...  
  return {  
    ...  
    addVideoTrack: function (track) {  
      const player = createVideoPlayer(participantDiv);  
      videoPlayers.set(track.mid, player);  
    },  
    ...  
  }  
}
```

### 10.2. REMOVING VIDEO TRACK PLAYING

`removeVideoTrack()` [code](#)

```
const createOneToOneParticipantView = function () {  
  ...  
  return {  
    ...  
    removeVideoTrack: function (track) {  
      const player = videoPlayers.get(track.mid);  
      if (player) {  
        player.dispose();  
      }  
    },  
    ...  
  }  
}
```

### 10.3. ADDING MEDIA SOURCE TO VIDEO TAG

`addVideoSource()` [code](#)

```
const createOneToOneParticipantView = function () {  
  ...  
  return {  
    ...  
    addVideoSource: function (remoteVideoTrack, track, onResize,  
muteHandler) {  
      const player = videoPlayers.get(track.mid);  
      if (player) {  
        player.setVideoSource(remoteVideoTrack, onResize,  
muteHandler);  
      }  
    },  
    ...  
  }  
}
```

```
}  
}
```

#### 10.4. REMOVING MEDIA SOURCE FROM VIDEO TAG

`removeVideoSource()` [code](#)

```
const createOneToOneParticipantView = function () {  
  ...  
  return {  
    ...  
    removeVideoSource: function (track) {  
      const player = videoPlayers.get(track.mid);  
      if (player) {  
        player.removeVideoSource();  
      }  
    },  
    ...  
  }  
}
```

#### 10.5. SHOW VIDEO TRACK

`showVideoTrack()` [code](#)

```
const createOneToOneParticipantView = function () {  
  ...  
  return {  
    ...  
    showVideoTrack: function (track) {  
      const player = videoPlayers.get(track.mid);  
      if (player) {  
        player.showVideoTrack(track);  
      }  
    },  
    ...  
  }  
}
```

#### 10.6. ADDING AUDIO TRACK

`addAudioTrack()` [code](#)

```
const createOneToOneParticipantView = function () {  
  ...  
  return {  
    ...  
    addAudioTrack: function (track, audioTrack, show) {  
      const stream = new MediaStream();  
      stream.addTrack(audioTrack);  
      const audioElement = document.createElement("audio");  
      if (!show) {  
        hideItem(audioElement);  
      }  
      audioElement.controls = "controls";  
    }  
  }  
}
```

```

        audioElement.muted = true;
        audioElement.autoplay = true;
        audioElement.onloadedmetadata = function (e) {
            audioElement.play().then(function () {
                if (Browser().isSafariWebRTC() && Browser().isiOS()) {
                    console.warn("Audio track should be manually unmuted
in iOS Safari");
                } else {
                    audioElement.muted = false;
                }
            });
        };
        audioElements.set(track.mid, audioElement);
        audioDisplay.appendChild(audioElement);
        audioElement.srcObject = stream;
    },
    ...
}
}

```

## 10.7. REMOVING AUDIO TRACK

`removeAudioTrack()` [code](#)

```

const createOneToOneParticipantView = function () {
    ...
    return {
        ...
        removeAudioTrack: function (track) {
            const audioElement = audioElements.get(track.mid);
            if (audioElement) {
                audioElement.remove();
                audioElements.delete(track.mid);
            }
        },
        ...
    }
}

```

## 10.8. SET PARTICIPANT NICKNAME TO DISPLAY

`setNickname()` [code](#)

```

const createOneToOneParticipantView = function () {
    ...
    return {
        ...
        setNickname: function (userId, nickname) {
            const additionalUserId = userId ? "#" + getShortUserId(userId) :
"";
            participantNicknameDisplay.innerHTML = "Name: " + nickname +
additionalUserId;
        },
        ...
    }
}

```

---

## 10.9. UPDATE TRACK QUALITY INFO

`updateQuality()` code

```
const createOneToOneParticipantView = function () {
  ...
  return {
    ...
    updateQuality: function (track, qualityName, available) {
      const player = videoPlayers.get(track.mid);
      if (player) {
        player.updateQuality(qualityName, available);
      }
    },
    ...
  }
}
```

## 10.10. ADD TRACK QUALITY INFO

`addQuality()` code

```
const createOneToOneParticipantView = function () {
  ...
  return {
    ...
    addQuality: function (track, qualityName, available, onQualityPick) {
      const player = videoPlayers.get(track.mid);
      if (player) {
        player.addQuality(qualityName, available, onQualityPick);
      }
    },
    ...
  }
}
```

## 10.11. PICK A QUALITY TO DISPLAY

`pickQuality()` code

```
const createOneToOneParticipantView = function () {
  ...
  return {
    ...
    pickQuality: function (track, qualityName) {
      const player = videoPlayers.get(track.mid);
      if (player) {
        player.pickQuality(qualityName);
      }
    },
    ...
  }
}
```

## 10.12. DISPOSE THE OBJECT

`dispose()` code

```
const createOneToOneParticipantView = function () {
  ...
  return {
    ...
    dispose: function () {
      for (const player of videoPlayers.values()) {
        player.dispose();
      }
      videoPlayers.clear();
      for (const element of audioElements.values()) {
        element.remove();
      }
      audioElements.clear();
    },
    ...
  }
}
```

## 11. Participant view object creation for one-to-many meeting room

`createOneToManyParticipantView()` code

```
const createOneToManyParticipantView = function () {

  const participantDiv = createContainer(null);

  const audioDisplay = createContainer(participantDiv);

  const participantNicknameDisplay = createInfoDisplay(participantDiv,
  "Name: ")

  const audioElements = new Map();
  const player = createVideoPlayer(participantDiv);

  return {
    rootDiv: participantDiv,
    currentTrack: null,
    dispose: function () {
      ...
    },
    addVideoTrack: function (track) {
      ...
    },
    removeVideoTrack: function (track) {
      ...
    },
    addVideoSource: function (remoteVideoTrack, track, onResize,
muteHandler) {
      ...
    },
    removeVideoSource: function (track) {
      ...
    }
  }
}
```

```

    },
    showVideoTrack: function (track) {
        ...
    },
    addAudioTrack: function (track, audioTrack, show) {
        ...
    },
    removeAudioTrack: function (track) {
        ...
    },
    setNickname: function (userId, nickname) {
        ...
    },
    updateQuality: function (track, qualityName, available) {
        ...
    },
    addQuality: function (track, qualityName, available, onQualityPick) {
        ...
    },
    clearQualityState: function (track) {
        ...
    },
    pickQuality: function (track, qualityName) {
        ...
    }
}
}

```

### 11.1. ADDING VIDEO TRACK TO PLAY

`addVideoTrack()` code

```

const createOneToManyParticipantView = function () {
    ...
    return {
        ...
        addVideoTrack: function (track, requestVideoTrack) {
            player.addVideoTrack(track, async () => {
                return requestVideoTrack();
            });
        },
        ...
    }
}

```

### 11.2. REMOVING VIDEO TRACK PLAYING

`removeVideoTrack()` code

```

const createOneToManyParticipantView = function () {
    ...
    return {
        ...
        removeVideoTrack: function (track) {
            player.removeVideoTrack(track);
        },
    }
}

```

```
    ...
  }
}
```

### 11.3. ADDING MEDIA SOURCE TO VIDEO TAG

`addVideoSource()` [code](#)

```
const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    addVideoSource: function (remoteVideoTrack, track, onResize,
muteHandler) {
      this.currentTrack = track;
      player.setVideoSource(remoteVideoTrack, onResize, muteHandler);
    },
    ...
  }
}
```

### 11.4. REMOVING MEDIA SOURCE FROM VIDEO TAG

`removeVideoSource()` [code](#)

```
const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    removeVideoSource: function (track) {
      if (this.currentTrack && this.currentTrack.mid === track.mid) {
        player.removeVideoSource();
      }
    },
    ...
  }
}
```

### 11.5. SHOW VIDEO TRACK

`showVideoTrack()` [code](#)

```
const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    showVideoTrack: function (track) {
      player.showVideoTrack(track);
    },
    ...
  }
}
```

### 11.6. ADDING AUDIO TRACK

### addAudioTrack() code

```
const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    addAudioTrack: function (track, audioTrack, show) {
      const stream = new MediaStream();
      stream.addTrack(audioTrack);
      const audioElement = document.createElement("audio");
      if (!show) {
        hideItem(audioElement);
      }
      audioElement.controls = "controls";
      audioElement.muted = true;
      audioElement.autoplay = true;
      audioElement.onloadedmetadata = function (e) {
        audioElement.play().then(function () {
          if (Browser().isSafariWebRTC() && Browser().isiOS()) {
            console.warn("Audio track should be manually unmuted
in iOS Safari");
          } else {
            audioElement.muted = false;
          }
        });
      });
      audioElements.set(track.mid, audioElement);
      audioDisplay.appendChild(audioElement);
      audioElement.srcObject = stream;
    },
    ...
  }
}
```

### 11.7. REMOVING AUDIO TRACK

#### removeAudioTrack() code

```
const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    removeAudioTrack: function (track) {
      const audioElement = audioElements.get(track.mid);
      if (audioElement) {
        audioElement.remove();
        audioElements.delete(track.mid);
      }
    },
    ...
  }
}
```

### 11.8. SET PARTICIPANT NICKNAME TO DISPLAY



### setNickname() code

```
const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    setNickname: function (userId, nickname) {
      const additionalUserId = userId ? "#" + getShortUserId(userId) :
      "";
      participantNicknameDisplay.innerText = "Name: " + nickname +
      additionalUserId;
    },
    ...
  }
}
```

## 11.9. UPDATE TRACK QUALITY INFO

### updateQuality() code

```
const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    updateQuality: function (track, qualityName, available) {
      player.updateQuality(qualityName, available);
    },
    ...
  }
}
```

## 11.10. ADD TRACK QUALITY INFO

### addQuality() code

```
const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    addQuality: function (track, qualityName, available, onQualityPick) {
      player.addQuality(qualityName, available, onQualityPick);
    },
    ...
  }
}
```

## 11.11. PICK A QUALITY TO DISPLAY

### pickQuality() code

```
const createOneToManyParticipantView = function () {
  ...
  return {
    ...
```

```

    pickQuality: function (track, qualityName) {
      player.pickQuality(qualityName);
    }
    ...
  }
}

```

#### 11.12. CLEAR QUALITY STATE DISPLAYED

`clearQualityState()` code

```

const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    clearQualityState: function (track) {
      player.clearQualityState();
    },
    ...
  }
}

```

#### 11.13. DISPOSE THE OBJECT

`dispose()` code

```

const createOneToManyParticipantView = function () {
  ...
  return {
    ...
    dispose: function () {
      player.dispose();
      for (const element of audioElements.values()) {
        element.remove();
      }
      audioElements.clear();
    },
    ...
  }
}

```

## 12. Video player object creation

`createVideoPlayer()` code

```

const createVideoPlayer = function (participantDiv) {

  const streamDisplay = createContainer(participantDiv);

  const resolutionLabel = createInfoDisplay(streamDisplay, "0x0");
  hideItem(resolutionLabel);

  const trackNameDisplay = createInfoDisplay(streamDisplay, "track not
set");
  hideItem(trackNameDisplay);

```

```
const videoMuteDisplay = createContainer(streamDisplay);

const qualityDisplay = createContainer(streamDisplay);

const trackDisplay = createContainer(streamDisplay);

let videoElement;

const trackButtons = new Map();
const qualityButtons = new Map();

const lock = function () {
  ...
}

const unlock = function () {
  ...
}

const setWebkitEventHandlers = function (video) {
  ...
}
const setEventHandlers = function (video) {
  ...
}

const repickQuality = function (qualityName) {
  ...
}

return {
  rootDiv: streamDisplay,
  muteButton: null,
  autoButton: null,
  dispose: function () {
    ...
  },
  clearQualityState: function () {
    ...
  },
  addVideoTrack: function (track, asyncCallback) {
    ...
  },
  removeVideoTrack: function (track) {
    ...
  },
  setVideoSource: function (remoteVideoTrack, onResize, onMute) {
    ...
  },
  removeVideoSource: function () {
    ...
  },
  showVideoTrack: function (track) {
    ...
  },
  updateQuality: function (qualityName, available) {
    ...
  }
}
```

```

    },
    addQuality: function (qualityName, available, onPickQuality) {
        ...
    },
    pickQuality: function (qualityName) {
        ...
    }
}
}

```

## 12.1. LOCK AND UNLOCK PLAYER BUTTONS TO WAIT FOR ASYNCHRONOUS OPERATIONS

`lock()`, `unlock()` [code](#)

```

const createVideoPlayer = function (participantDiv) {
    ...
    const lock = function () {
        for (const btn of trackButtons.values()) {
            btn.disabled = true;
        }
        for (const state of qualityButtons.values()) {
            state.btn.disabled = true;
        }
    }

    const unlock = function () {
        for (const btn of trackButtons.values()) {
            btn.disabled = false;
        }
        for (const state of qualityButtons.values()) {
            state.btn.disabled = false;
        }
    }

    ...
    return {
        ...
    }
}

```

## 12.2. SAFARI EVENT HANDLERS SET UP

`setWebkitEventHandlers()` [code](#)

```

const createVideoPlayer = function (participantDiv) {
    ...
    const setWebkitEventHandlers = function (video) {
        let needRestart = false;
        let isFullscreen = false;
        // Use webkitbeginfullscreen event to detect full screen mode in iOS
        Safari
        video.addEventListener("webkitbeginfullscreen", function () {
            isFullscreen = true;
        });
        video.addEventListener("pause", function () {
            if (needRestart) {
                console.log("Media paused after fullscreen, continue...");
            }
        });
    };
}

```

```

        video.play();
        needRestart = false;
    } else {
        console.log("Media paused by click, continue...");
        video.play();
    }
});
video.addEventListener("webkitendfullscreen", function () {
    video.play();
    needRestart = true;
    isFullscreen = false;
});
}
...
return {
    ...
}
}

```

### 12.3. OTHER BROWSERS EVENT HANDLERS SET UP

`setEventHandlers()` [code](#)

```

const createVideoPlayer = function (participantDiv) {
    ...
    const setEventHandlers = function (video) {
        // Ignore play/pause button
        video.addEventListener("pause", function () {
            console.log("Media paused by click, continue...");
            video.play();
        });
    }
    ...
    return {
        ...
    }
}

```

### 12.4. CHANGE QUALITY BUTTONS COLORS WHEN PICKING A QUALITY

`repickQuality()` [code](#)

```

const createVideoPlayer = function (participantDiv) {
    ...
    const repickQuality = function (qualityName) {
        for (const [quality, state] of qualityButtons.entries()) {
            if (quality === qualityName) {
                state.btn.style.color = QUALITY_COLORS.SELECTED;
            } else if (state.btn.style.color === QUALITY_COLORS.SELECTED) {
                if (state.available) {
                    state.btn.style.color = QUALITY_COLORS.AVAILABLE;
                } else {
                    state.btn.style.color = QUALITY_COLORS.UNAVAILABLE;
                }
            }
        }
    }
    ...
}

```

```

    }
    ...
    return {
        ...
    }
}

```

## 12.5. REMOVE QUALITY BUTTONS

`clearQualityState()` [code](#)

```

const createVideoPlayer = function (participantDiv) {

    ...
    return {
        ...
        clearQualityState: function () {
            qualityButtons.forEach((state, qName) => {
                state.btn.remove();
            });
            qualityButtons.clear();
        },
        ...
    }
}

```

## 12.6. ADDING A VIDEO TRACK TO PLAY

`addVideoTrack()` [code](#)

```

const createVideoPlayer = function (participantDiv) {

    ...
    return {
        ...
        addVideoTrack: function (track, asyncCallback) {
            const trackButton = document.createElement("button");
            trackButtons.set(track.mid, trackButton);
            trackButton.innerText = "Track №" + track.mid + ": " +
track.contentType;
            trackButton.setAttribute("style", "display:inline-block; border:
solid; border-width: 1px");
            trackButton.style.color = QUALITY_COLORS.AVAILABLE;
            const self = this;
            trackButton.addEventListener('click', async function () {
                console.log("Clicked on track button track.mid " +
track.mid);
                if (trackButton.style.color === QUALITY_COLORS.SELECTED) {
                    return
                }

                lock();
                asyncCallback().then(() => {
                    self.showVideoTrack(track);
                }).finally(() => {
                    unlock();
                });
            });
        }
    }
}

```

```

        });
    });
    trackDisplay.appendChild(trackButton);
},
...
}
}

```

## 12.7. REMOVING VIDEO TRACK PLAYED

`removeVideoTrack()` [code](#)

```

const createVideoPlayer = function (participantDiv) {

    ...
    return {
        ...
        removeVideoTrack: function (track) {
            const trackButton = trackButtons.get(track.mid);
            if (trackButton) {
                trackButton.remove();
                trackButtons.delete(track.mid);
            }
        },
        ...
    }
}

```

## 12.8. ADDING A VIDEO TAG AND SETTING VIDEO TRACK AS A SOURCE

`setVideoSource()` [code](#)

```

const createVideoPlayer = function (participantDiv) {

    ...
    return {
        ...
        setVideoSource: function (remoteVideoTrack, onResize, onMute) {
            if (!this.muteButton) {
                const newVideoMuteBtn = document.createElement("button");
                this.muteButton = newVideoMuteBtn;
                newVideoMuteBtn.innerText = "mute";
                newVideoMuteBtn.setAttribute("style", "display:inline-block;
border: solid; border-width: 1px");
                newVideoMuteBtn.addEventListener('click', async function () {
                    newVideoMuteBtn.disabled = true;
                    try {
                        if (newVideoMuteBtn.innerText === "mute") {
                            await onMute(true);
                            newVideoMuteBtn.innerText = "unmute";
                        } else if (newVideoMuteBtn.innerText === "unmute") {
                            await onMute(false);
                            newVideoMuteBtn.innerText = "mute";
                        }
                    }
                } finally {
                    newVideoMuteBtn.disabled = false;
                }
            }
        }
    }
}

```

```

    }
    });
    videoMuteDisplay.appendChild(newVideoMuteBtn);
  }

  if (videoElement) {
    videoElement.remove();
    videoElement = null;
  }

  if (!remoteVideoTrack) {
    return;
  }

  videoElement = document.createElement("video");
  hideItem(videoElement);
  videoElement.setAttribute("style", "display:none; border: solid;
border-width: 1px");

  const stream = new MediaStream();

  streamDisplay.appendChild(videoElement);
  videoElement.srcObject = stream;
  videoElement.onloadedmetadata = function (e) {
    videoElement.play();
  };
  videoElement.addEventListener("resize", function (event) {
    showItem(resolutionLabel);
    if (videoElement) {
      resolutionLabel.innerText = videoElement.videoWidth + "x"
+ videoElement.videoHeight;
      resizeVideo(event.target);
      onResize();
    }
  });
  stream.addTrack(remoteVideoTrack);
  if (Browser().isSafariWebRTC()) {
    videoElement.setAttribute("playsinline", "");
    videoElement.setAttribute("webkit-playsinline", "");
    setWebkitEventHandlers(videoElement);
  } else {
    setEventHandlers(videoElement);
  }
},
...
}
}

```

## 12.9. REMOVING THE VIDEO TAG

`removeVideoSource()` [code](#)

```

const createVideoPlayer = function (participantDiv) {
  ...
  return {
    ...
  }
}

```



```

    removeVideoSource: function () {
      if (videoElement) {
        videoElement.remove();
        videoElement = null;
      }
      if (this.muteButton) {
        this.muteButton.remove();
        this.muteButton = null;
      }
      hideItem(resolutionLabel);
      trackNameDisplay.innerText = "track not set";
    },
    ...
  }
}

```

## 12.10. VIDEO TAG AND TRACK INFO DISPLAYING

`showVideoTrack()` code

```

const createVideoPlayer = function (participantDiv) {

  ...
  return {
    ...
    showVideoTrack: function (track) {
      if (videoElement) {
        showItem(videoElement);
      }
      for (const [mid, btn] of trackButtons.entries()) {
        if (mid === track.mid) {
          btn.style.color = QUALITY_COLORS.SELECTED;
        } else if (btn.style.color === QUALITY_COLORS.SELECTED) {
          btn.style.color = QUALITY_COLORS.AVAILABLE;
        }
      }
      trackNameDisplay.innerText = "Current video track: " + track.mid;
      showItem(trackNameDisplay);
    },
    ...
  }
}

```

## 12.11. UPDATE AVAILABLE AUALITY INFO

`updateQuality()` code

```

const createVideoPlayer = function (participantDiv) {

  ...
  return {
    ...
    updateQuality: function (qualityName, available) {
      const value = qualityButtons.get(qualityName);
      if (value) {
        const qualityButton = value.btn;

```

```

        value.available = available;
        if (qualityButton.style.color === QUALITY_COLORS.SELECTED) {
            return;
        }
        if (available) {
            qualityButton.style.color = QUALITY_COLORS.AVAILABLE;
        } else {
            qualityButton.style.color = QUALITY_COLORS.UNAVAILABLE;
        }
    }
},
...
}
}

```

## 12.12. ADD QUALITY BUTTON

`addQuality()` code

```

const createVideoPlayer = function (participantDiv) {
    ...
    return {
        ...
        addQuality: function (qualityName, available, onPickQuality) {
            const qualityButton = document.createElement("button");
            qualityButtons.set(qualityName, {btn: qualityButton, available:
available});
            qualityButton.innerText = qualityName;
            qualityButton.setAttribute("style", "display:inline-block;
border: solid; border-width: 1px");
            if (available) {
                qualityButton.style.color = QUALITY_COLORS.AVAILABLE;
            } else {
                qualityButton.style.color = QUALITY_COLORS.UNAVAILABLE;
            }
            qualityDisplay.appendChild(qualityButton);
            qualityButton.addEventListener('click', async function () {
                console.log("Clicked on quality button " + qualityName);
                if (qualityButton.style.color === QUALITY_COLORS.SELECTED ||
qualityButton.style.color === QUALITY_COLORS.UNAVAILABLE || !videoElement) {
                    return;
                }
                lock();
                onPickQuality().finally(() => unlock());
            });
        },
        ...
    }
}

```

## 12.13. HANDLE A QUALITY BUTTON CLICK

`pickQuality()` code

```

const createVideoPlayer = function (participantDiv) {
  ...
  return {
    ...
    pickQuality: function (qualityName) {
      repickQuality(qualityName);
    }
    ...
  }
}

```

#### 12.14. DISPOSE THE OBJECT

`dispose()` [code](#)

```

const createVideoPlayer = function (participantDiv) {
  ...
  return {
    ...
    dispose: function () {
      streamDisplay.remove();
    },
    ...
  }
}

```

#### 13. GET A TRACK TO DISPLAY FROM THE MEETING ROOM

`remoteTrackProvider()` [code](#)

```

const remoteTrackProvider = function (room) {
  return {
    getVideoTrack: async function () {
      return await room.getRemoteTrack("VIDEO", false);
    },
    getAudioTrack: async function () {
      return await room.getRemoteTrack("AUDIO", true);
    }
  }
}

```

### 14. Helper functions

#### 14.1. RE-SCALE VIDEO TO PLAYER VIDEO TAG SIZE

`resizeVideo()`, `downScaleToFitSize()` [code](#)

```

const resizeVideo = function (video, width, height) {
  // TODO: fix
  if (video) {
    return;
  }
}

```

```

    if (!video.parentNode) {
        return;
    }
    if (video instanceof HTMLCanvasElement) {
        video.videoWidth = video.width;
        video.videoHeight = video.height;
    }
    const display = video.parentNode;
    const parentSize = {
        w: display.parentNode.clientWidth,
        h: display.parentNode.clientHeight
    };
    let newSize;
    if (width && height) {
        newSize = downScaleToFitSize(width, height, parentSize.w,
parentSize.h);
    } else {
        newSize = downScaleToFitSize(video.videoWidth, video.videoHeight,
parentSize.w, parentSize.h);
    }
    display.style.width = newSize.w + "px";
    display.style.height = newSize.h + "px";

    //vertical align
    let margin = 0;
    if (parentSize.h - newSize.h > 1) {
        margin = Math.floor((parentSize.h - newSize.h) / 2);
    }
    display.style.margin = margin + "px auto";
    console.log("Resize from " + video.videoWidth + "x" + video.videoHeight +
" to " + display.offsetWidth + "x" + display.offsetHeight);
}

const downScaleToFitSize = function (videoWidth, videoHeight, dstWidth,
dstHeight) {
    var newWidth, newHeight;
    var videoRatio = videoWidth / videoHeight;
    var dstRatio = dstWidth / dstHeight;
    if (dstRatio > videoRatio) {
        newHeight = dstHeight;
        newWidth = Math.floor(videoRatio * dstHeight);
    } else {
        newWidth = dstWidth;
        newHeight = Math.floor(dstWidth / videoRatio);
    }
    return {
        w: newWidth,
        h: newHeight
    };
}

```

## 14.2. CREATE A TAG TO DISPLAY A TEXTUAL INFO

`createInfoDisplay()` [code](#)

```

const createInfoDisplay = function (parent, text) {
    const div = document.createElement("div");

```

```
    if (text) {
      div.innerHTML = text;
    }
    div.setAttribute("style", "width:auto; height:30px;");
    div.setAttribute("class", "text-center");
    if (parent) {
      parent.appendChild(div);
    }
    return div;
  }
}
```

#### 14.3. CREATE A CONTAINER TAG

`createContainer()` [code](#)

```
const createContainer = function (parent) {
  const div = document.createElement("div");
  div.setAttribute("style", "width:auto; height:auto;");
  div.setAttribute("class", "text-center");
  if (parent) {
    parent.appendChild(div);
  }
  return div;
}
```

#### 14.4. SHOW AND HIDE THE TAG ON THE PAGE

`showItem()`, `hideItem()` [code](#)

```
const showItem = function (tag) {
  if (tag) {
    tag.style.display = "block";
  }
}

const hideItem = function (tag) {
  if (tag) {
    tag.style.display = "none";
  }
}
```