Real-Time Streaming User Guide

Amazon IVS

Amazon IVS: Real-Time Streaming User Guide

Table of Contents

What is Amazon IVS Real-Time Streaming?

Amazon Interactive Video Service (IVS) Real-Time Streaming gives you everything you need to add real-time audio and video to your applications.

Strengths:

- Real-time latency — Build applications for latency-sensitive use cases, helping your viewers stay connected and engaged with IVS real-time streaming. Deliver live streams with a latency that can be under 300 milliseconds from host to viewer.
- High concurrency — Unlock the potential of large-scale interactions with IVS real-time streaming. Accommodate audiences of up to 10,000 viewers and enable up to 12 hosts to take the virtual stage.
- Mobile optimized — IVS real-time streaming is optimized for mobile use cases, catering to a diverse range of devices and network capabilities. By integrating the Amazon IVS broadcast SDKs for Android and iOS, your users can engage as hosts or viewers, enjoying high-quality live streams on their mobile devices.

Use cases:

- Guest spots — Create applications that allow hosts to promote guests "on stage," turning viewers into hosts for real-time interactions.
- Versus (VS) mode — Produce experiences with side-by-side competitions and let viewers watch hosts compete in real-time.
- Audio rooms — Invite listeners to join the conversation as guests and foster deeper engagement in your audio rooms.
- Live video auctions — Turn auctions into interactive video events and maintain their excitement and integrity with real-time latency.

In addition to the product documentation here, see https://ivs.rocks/, a dedicated site to browse published content (demos, code samples, blog posts), estimate cost, and experience Amazon IVS through live demos.

Global Solution, Regional Control

Streaming and Viewing are Global

You can use Amazon IVS to stream to viewers worldwide:

- When you stream, Amazon IVS automatically ingests video at a location near you.
- Viewers can watch your live streams globally.

Another way of saying this is that the "data plane" is global. The data plane refers to streaming/ ingesting and viewing.

Control is Regional

While the Amazon IVS data plane is global, the "control plane" is regional. The control plane refers to the Amazon IVS console, API, and resources (stages).

Another way of saying this is that Amazon IVS is a "regional AWS service." That is, Amazon IVS resources in each region are independent of similar resources in other regions. For example, a stage that you create in one region is independent of stages you create in other regions.

When you use resources (e.g., create a stage), you must specify the region in which it will be created. Subsequently, when you manage resources, you must do so from the same region where they were created.

If you use the ...	You specify the region by ...
Amazon IVS console	Using the **Select a Region** drop-down in the top right of the navigation bar.
Amazon IVS API	Using the appropriate service endpoint. See the Amazon IVS Real-Time Streaming API Reference. (If you access the API through an SDK, set up the SDK's `region` parameter. See Tools to Build on AWS.)
AWS CLI	Either: • Appending `--region <aws-region>` to your CLI command.

If you use the ...	You specify the region by ...
	• Putting the region in your local AWS configuration file.

Remember, regardless of the region in which a stage was created, you can stream to Amazon IVS from anywhere, and viewers can watch from anywhere.

Getting Started with IVS Real-Time Streaming

This document takes you through the steps involved in integrating Amazon IVS Real-Time Streaming into your app.

Topics

Introduction

Prerequisites

Before you use Real-Time Streaming for the first time, complete the following tasks. For instructions, see Getting Started with IVS Low-Latency Streaming.

- Create an AWS Account
- Set Up Root and Administrative Users

Other References

- IVS Web Broadcast SDK Reference
- IVS Android Broadcast SDK Reference
- IVS iOS Broadcast SDK Reference
- IVS Real-Time Streaming API Reference

Real-Time Streaming Terminology

Term	Description	
Stage	A virtual space where participants can exchange video in real time.	
Host	A participant that sends local video to the stage.	
Viewer	A participant that receives video of the hosts.	
Participant	A user connected to the stage as a host or viewer.	
Participant token	A token that authenticates a participant when they join a stage.	
Broadcast SDK	A client library that enables participants to send and receive video.	

Overview of Steps

1. the section called “Set Up IAM Permissions” — Create an AWS Identity and Access Management (IAM) policy that gives users a basic set of permissions and assign that policy to users.
2. Create a stage — Create a virtual space where participants can exchange video in real time.
3. Distribute participant tokens — Send tokens to participants so they can join your stage.
4. Integrate the IVS Broadcast SDK — Add the broadcast SDK to your app to enable participants to send and receive video: the section called “Web”, the section called “Android”, and the section called “iOS”.
5. Publish and subscribe to video — Send your video to the stage and receive video from other hosts: the section called “Web”, the section called “Android”, and the section called “iOS”.

Set Up IAM Permissions

Next, you must create an AWS Identity and Access Management (IAM) policy that gives users a basic set of permissions (e.g., to create an Amazon IVS stage and create participant tokens) and assign that policy to users. You can either assign the permissions when creating a new user or add permissions to an existing user. Both procedures are given below.

For more information (for example, to learn about IAM users and policies, how to attach a policy to a user, and how to constrain what users can do with Amazon IVS), see:

- Creating an IAM User in the *IAM User Guide*
- The information in Amazon IVS Security on IAM and "Managed Policies for IVS."
- The IAM information in Amazon IVS Security

You can either use an existing AWS managed policy for Amazon IVS or create a new policy that customizes the permissions you want to grant to a set of users, groups, or roles. Both approaches are described below.

Use an Existing Policy for IVS Permissions

In most cases, you will want to use an AWS managed policy for Amazon IVS. They are described fully in the Managed Policies for IVS section of *IVS Security*.

- Use the `IVSReadOnlyAccess` AWS managed policy to give your application developers access to all IVS Get and List API endpoints (for both low-latency and real-time streaming).
- Use the `IVSFullAccess` AWS managed policy to give your application developers access to all IVS API endpoints (for both low-latency and real-time streaming).

Optional: Create a Custom Policy for Amazon IVS Permissions

Follow these steps:

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Policies**, then choose **Create policy**. A **Specify permissions** window opens..

3. In the **Specify permissions** window, choose the **JSON** tab, and copy and paste the following IVS policy to the **Policy editor** text area. (The policy does not include all Amazon IVS actions. You can add/delete (Allow/Deny) endpoint access permissions as needed. See IVS Real-Time Streaming API Reference for details on IVS endpoints.)

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "ivs:CreateStage",
            "ivs:CreateParticipantToken",
            "ivs:GetStage",
            "ivs:GetStageSession",
            "ivs:ListStages",
            "ivs:ListStageSessions",
            "ivs:CreateEncoderConfiguration",
            "ivs:GetEncoderConfiguration",
            "ivs:ListEncoderConfigurations",
            "ivs:GetComposition",
            "ivs:ListCompositions",
            "ivs:StartComposition",
            "ivs:StopComposition"
          ],
          "Resource": "*"
      },
      {
         "Effect": "Allow",
         "Action": [
            "cloudwatch:DescribeAlarms",
            "cloudwatch:GetMetricData",
            "s3:DeleteBucketPolicy",
            "s3:GetBucketLocation",
            "s3:GetBucketPolicy",
            "s3:PutBucketPolicy",
            "servicequotas:ListAWSDefaultServiceQuotas",
            "servicequotas:ListRequestedServiceQuotaChangeHistoryByQuota",
            "servicequotas:ListServiceQuotas",
            "servicequotas:ListServices",
            "servicequotas:ListTagsForResource"
         ],
         "Resource": "*"
```

```
        }
    ]
}
```

4. Still in the **Specify permissions** window, choose **Next** (scroll to the bottom of the window to see this). A **Review and create** window opens.
5. On the **Review and create** window, enter a **Policy name** and optionally add a **Description**. Make a note of the policy name, as you will need it when creating users (below). Choose **Create policy** (at the bottom of the window).
6. You are returned to the IAM console window, where you should see a banner confirming that your new policy was created.

Create a New User and Add Permissions

IAM User Access Keys

IAM access keys consist of an access key ID and a secret access key. They are used to sign programmatic requests that you make to AWS. If you don't have access keys, you can create them from the AWS Management Console. As a best practice, do not create root-user access keys.

The only time that you can view or download a secret access key is when you create access keys. You cannot recover them later. However, you can create new access keys at any time; you must have permissions to perform the required IAM actions.

Always store access keys securely. Never share them with third parties (even if an inquiry seems to come from Amazon). For more information, see Managing access keys for IAM users in the *IAM User Guide*.

Procedure

Follow these steps:

1. In the navigation pane, choose **Users**, then choose **Create user**. A **Specify user details** window opens.
2. In the **Specify user details** window:
    a. Under **User details**, type the new **User name** to be created.
    b. Check **Provide user access to the AWS Management Console**.
    c. Under **Console password**, select **Autogenerated password**.

   d. Check **Users must create a new password at next sign-in**.

   e. Choose **Next**. A **Set permissions** window opens.

3. Under **Set permissions**, select **Attach policies directly**. A **Permissions policies** window opens.
4. In the search box, enter an IVS policy name (either an AWS managed policy or your previously created custom policy). When it is found, check the box to select the policy.
5. Choose **Next** (at the bottom of the window). A **Review and create** window opens.
6. On the **Review and create** window, confirm that all user details are correct, then choose **Create user** (at the bottom of the window).
7. The **Retrieve password** window opens, containing your **Console sign-in details**. *Save this information securely for future reference*. When you are done, choose **Return to users list**.

Add Permissions to an Existing User

Follow these steps:

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Users**, then choose an existing user name to be updated. (Choose the name by clicking on it; do not check the selection box.)
3. On the **Summary** page, on the **Permissions** tab, choose **Add permissions**. An **Add permissions** window opens.
4. Select **Attach existing policies directly**. A **Permissions policies** window opens.
5. In the search box, enter an IVS policy name (either an AWS managed policy or your previously created custom policy). When the policy is found, check the box to select the policy.
6. Choose **Next** (at the bottom of the window). A **Review** window opens.
7. On the **Review** window, select **Add permissions** (at the bottom of the window).
8. On the **Summary** page, confirm that the IVS policy was added.

Create a Stage

A stage is a virtual space where participants can exchange video in real time. It is the foundational resource of the Real-Time Streaming API. You can create a stage using either the console or the CreateStage endpoint.

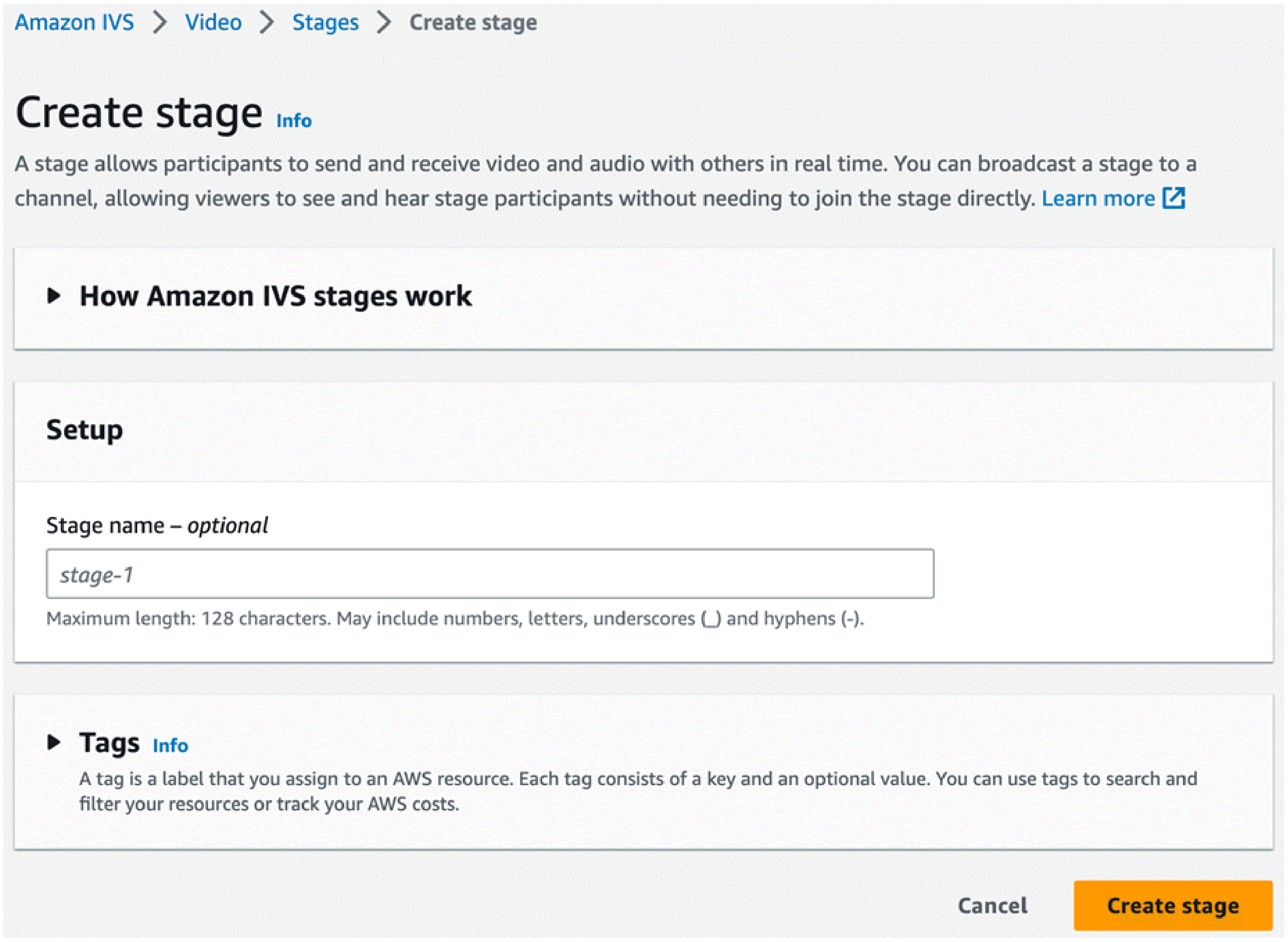
We recommend that where possible, you create a new stage for each logical session and delete it when done, rather than keeping around old stages for possible reuse. If stale resources (old stages, not to be reused) are not cleaned up, you're likely to hit the limit of the maximum number of stages faster.

Console Instructions

1. Open the Amazon IVS console.

   (You also can access the Amazon IVS console through the AWS Management Console.)

2. On the left navigation pane, select **Stages**, then select **Create stage**. The **Create stage** window appears.


Amazon IVS > Video > Stages > Create stage
Create stage Info

A stage allows participants to send and receive video and audio with others in real time. You can broadcast a stage to a channel, allowing viewers to see and hear stage participants without needing to join the stage directly. Learn more

▶ How Amazon IVS stages work
Setup

Stage name – *optional*

stage-1

Maximum length: 128 characters. May include numbers, letters, underscores (_) and hyphens (-).

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

3. Optionally enter a **Stage name**. Select **Create stage** to create the stage. The stage details page appears, for the new stage.

CLI Instructions

To install the AWS CLI, see Install or update the latest version of the AWS CLI.

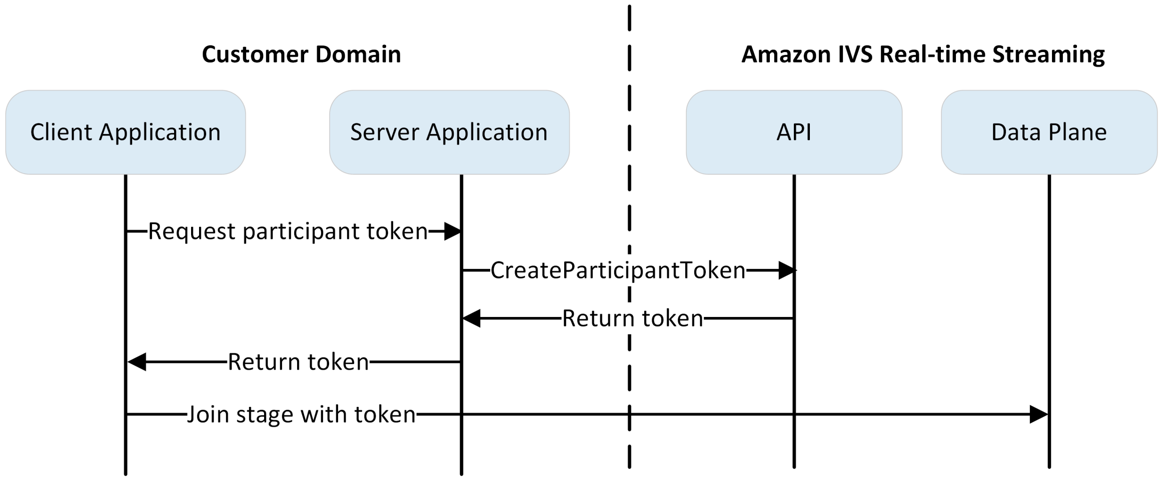
Now you can use the CLI to create and manage resources. The stage API is under the ivs-realtime namespace. For example, to create a stage:

```
aws ivs-realtime create-stage --name "test-stage"
```

The response is:

```
{
   "stage": {
      "arn": "arn:aws:ivs:us-west-2:376666121854:stage/VSWjvX5XOkU3",
      "name": "test-stage"
   }
}
```

Distribute Participant Tokens



Now that you have a stage, you need to create and distribute tokens to participants to enable them to join the stage and start sending and receiving video.

As shown above, a client application asks your server application for a token, and the server application calls CreateParticipantToken using an AWS SDK or SigV4 signed request. Since AWS credentials are used to call the API, the token should be generated in a secure server-side application, not the client-side application.

When creating a participant token, you can optionally specify the capabilities enabled by that token. The default is `PUBLISH` and `SUBSCRIBE`, which allows the participant to send and receive audio and video, but you could issue tokens with a subset of capabilities. For example, you could issue a token with only the `SUBSCRIBE` capability for moderators. In that case, the moderators could see the participants that are sending video but not send their own video.

You can create participant tokens via the console or CLI for testing and development, but most likely you will want to create them with the AWS SDK in your production environment.

You will need a way to distribute tokens from your server to each client (e.g., via an API request). We do not provide this functionality. For this guide, you can simply copy and paste the tokens into client code in the following steps.

Important: Treat tokens as opaque; i.e., do not build functionality based on token contents. The format of tokens could change in the future.

Console Instructions

1. Navigate to the stage you created in the prior step.
2. Select **Create a participant token**. **The Create a participant token** window appears.
3. Enter a user ID to be associated with the token. This can be any UTF-8 encoded text.
4. Select **Create a participant token**.
5. Copy the token. *Important: Be sure to save the token; IVS does not store it and you cannot retrieve it later*.

CLI Instructions

Creating a token with the AWS CLI requires that you first download and configure the CLI on your machine. For details, see the AWS Command Line Interface User Guide. Note that generating tokens with the AWS CLI is good for testing purposes, but for production use, we recommend that you generate tokens on the server side with the AWS SDK (see instructions below).

1. Run the `create-participant-token` command with the stage ARN. Include any or all of the following capabilities: `"PUBLISH"`, `"SUBSCRIBE"`.

```
aws ivs-realtime create-participant-token --stage-arn arn:aws:ivs:us-
west-2:376666121854:stage/VSWjvX5XOkU3 --capabilities '["PUBLISH", "SUBSCRIBE"]'
```

2. This returns a participant token:

```
{
    "participantToken": {
        "capabilities": [
            "PUBLISH",
            "SUBSCRIBE"
        ],
        "expirationTime": "2023-06-03T07:04:31+00:00",
        "participantId": "tU06DT5jCJeb",
        "token":
 "eyJhbGciOiJLTVMiLCJ0eXAiOiJKV1QifQ.eyJleHAiOjE2NjE1NDE0MjAsImp0aSI6ImpGcFdtdmVFTm9sUyIsInJ
TaKjllW9Qac6c5xBrdAk"    }
}
```

3. Save this token. You will need this to join the stage and send and receive video.

AWS SDK Instructions

You can use the AWS SDK to create tokens. Below are instructions for the AWS SDK using JavaScript.

Important: This code must be executed on the server side and its output passed to the client.

Prerequisite: To use the code sample below, you need to install the aws-sdk/client-ivs-realtime package. For details, see Getting started with the AWS SDK for JavaScript.

```
import { IVSRealTimeClient, CreateParticipantTokenCommand } from "@aws-sdk/client-ivs-
realtime";

const ivsRealtimeClient = new IVSRealTimeClient({ region: 'us-west-2' });
const stageArn = 'arn:aws:ivs:us-west-2:123456789012:stage/L210UYabcdef';
const createStageTokenRequest = new CreateParticipantTokenCommand({
  stageArn,
});
const response = await ivsRealtimeClient.send(createStageTokenRequest);
console.log('token', response.participantToken.token);
```

Integrate the IVS Broadcast SDK

IVS provides a broadcast SDK for web, Android, and iOS that you can integrate into your application. The broadcast SDK is used for both sending and receiving video. In this section, we write a simple application that enables two or more participants to interact in real time. The steps below guide you through creating an app called BasicRealTime. The full app code is on CodePen and GitHub:

- Web: https://codepen.io/amazon-ivs/pen/ZEqgrpo/cbe7ac3b0ecc8c0f0a5c0dc9d6d36433
- Android: https://github.com/aws-samples/amazon-ivs-real-time-streaming-android-samples
- iOS: https://github.com/aws-samples/amazon-ivs-real-time-streaming-ios-samples

Web

Set Up Files

To start, set up your files by creating a folder and an initial HTML and JS file:

```
mkdir realtime-web-example
cd realtime-web-example
touch index.html
touch app.js
```

You can install the broadcast SDK using a script tag or npm. Our example uses the script tag for simplicity but is easy to modify if you choose to use npm later.

Using a Script Tag

The Web broadcast SDK is distributed as a JavaScript library and can be retrieved at https://web-broadcast.live-video.net/1.11.0/amazon-ivs-web-broadcast.js.

When loaded via `<script>` tag, the library exposes a global variable in the window scope named `IVSBroadcastClient`.

Using npm

To install the npm package:

```
npm install amazon-ivs-web-broadcast
```

You can now access the IVSBroadcastClient object:

```
const { Stage } = IVSBroadcastClient;
```

Android

Create the Android Project

1. In Android Studio, create a **New Project**.
2. Choose **Empty Views Activity**.

   Note: In some older versions of Android Studio, the View-based activity is called **Empty Activity**. If your Android Studio window shows **Empty Activity** and does *not* show **Empty Views** Activity, select **Empty Activity**. Otherwise, don't select **Empty Activity**, since we'll be using View APIs (not Jetpack Compose).
3. Give your project a **Name**, then select **Finish**.

Install the Broadcast SDK

To add the Amazon IVS Android broadcast library to your Android development environment, add the library to your module's `build.gradle` file, as shown here (for the latest version of the Amazon IVS broadcast SDK). In newer projects the `mavenCentral` repository may already be included in your `settings.gradle` file, if that is the case you can omit the `repositories` block. For our sample, we'll also need to enable data binding in the `android` block.

```
android {
    dataBinding.enabled true
}

repositories {
    mavenCentral()
}

dependencies {
     implementation 'com.amazonaws:ivs-broadcast:1.17.0:stages@aar'
}
```

Alternately, to install the SDK manually, download the latest version from this location:

https://search.maven.org/artifact/com.amazonaws/ivs-broadcast

iOS

Create the iOS Project

1. Create a new Xcode project.
2. For **Platform**, select **iOS**.
3. For **Application**, select **App**.
4. Enter the **Product Name** of your app, then select **Next**.
5. Choose (navigate to) a directory in which to save the project, then select **Create**.

Next you need to bring in the SDK. We recommend that you integrate the broadcast SDK via CocoaPods. Alternatively, you can manually add the framework to your project. Both methods are described below.

Recommended: Install the Broadcast SDK (CocoaPods)

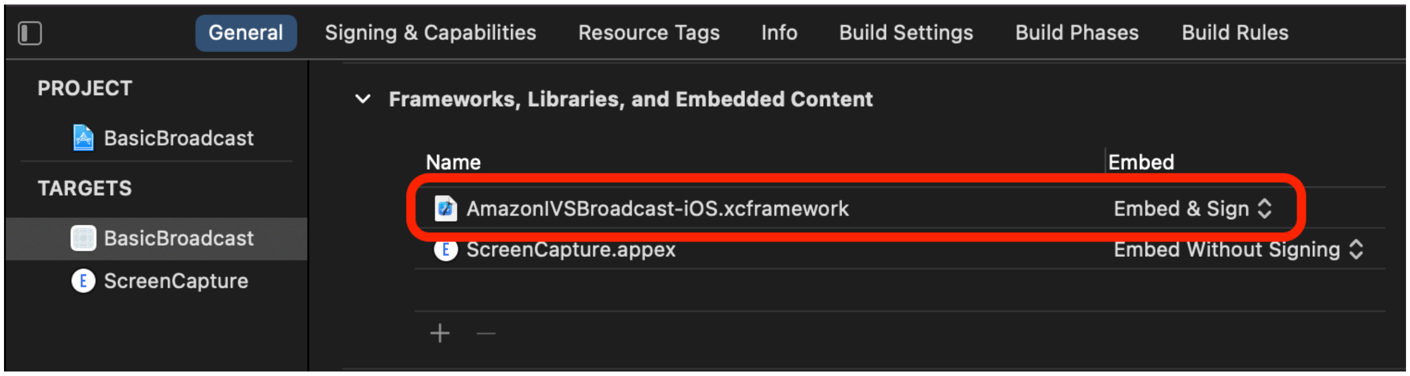
Assuming your project name is `BasicRealTime`, create a `Podfile` in the project folder with the following contents and then run `pod install`:

```
target 'BasicRealTime' do
  # Comment the next line if you don't want to use dynamic frameworks
  use_frameworks!

  # Pods for BasicRealTime
  pod 'AmazonIVSBroadcast/Stages'
end
```

Alternate Approach: Install the Framework Manually

1. Download the latest version from https://broadcast.live-video.net/1.17.0/AmazonIVSBroadcast-Stages.xcframework.zip.
2. Extract the contents of the archive. `AmazonIVSBroadcast.xcframework` contains the SDK for both device and simulator.

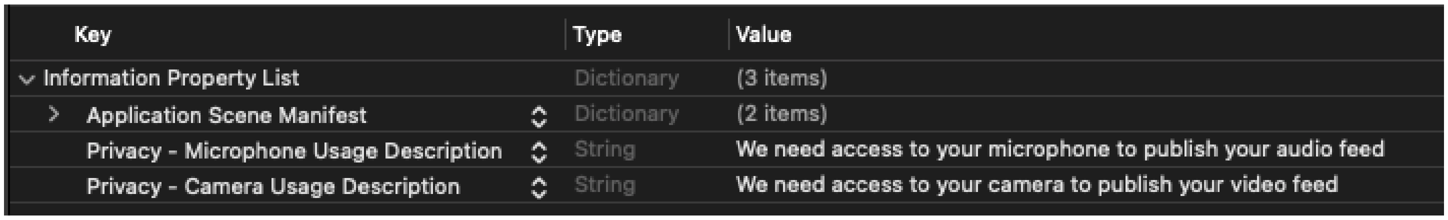
3. Embed `AmazonIVSBroadcast.xcframework` by dragging it into the **Frameworks, Libraries, and Embedded Content** section of the **General** tab for your application target:



Name	Embed
AmazonIVSBroadcast-iOS.xcframework	Embed & Sign
ScreenCapture.appex	Embed Without Signing

Configure Permissions

You need to update your project's `Info.plist` to add two new entries for `NSCameraUsageDescription` and `NSMicrophoneUsageDescription`. For the values, provide user-facing explanations of why your app is asking for camera and microphone access.



Key	Type	Value
Information Property List	Dictionary	(3 items)
Application Scene Manifest	Dictionary	(2 items)
Privacy - Microphone Usage Description	String	We need access to your microphone to publish your audio feed
Privacy - Camera Usage Description	String	We need access to your camera to publish your video feed

Publish and Subscribe to Video

See the details below for web, Android, and iOS.

Web

Create HTML Boilerplate

First let's create the HTML boilerplate and import the library as a script tag:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <!-- Import the SDK -->
  <script src="https://web-broadcast.live-video.net/1.11.0/amazon-ivs-web-
broadcast.js"></script>
</head>

<body>

<!-- TODO - fill in with next sections -->
<script src="./app.js"></script>

</body>
</html>
```

Accept Token Input and Add Join/Leave Buttons

Here we fill in the body with our input controls. These take as input the token, and they set up **Join** and **Leave** buttons. Typically applications will request the token from your application's API, but for this example you'll copy and paste the token into the token input.

```
<h1>IVS Real-Time Streaming</h1>
<hr />

<label for="token">Token</label>
<input type="text" id="token" name="token" />
<button class="button" id="join-button">Join</button>
<button class="button" id="leave-button" style="display: none;">Leave</button>
<hr />
```

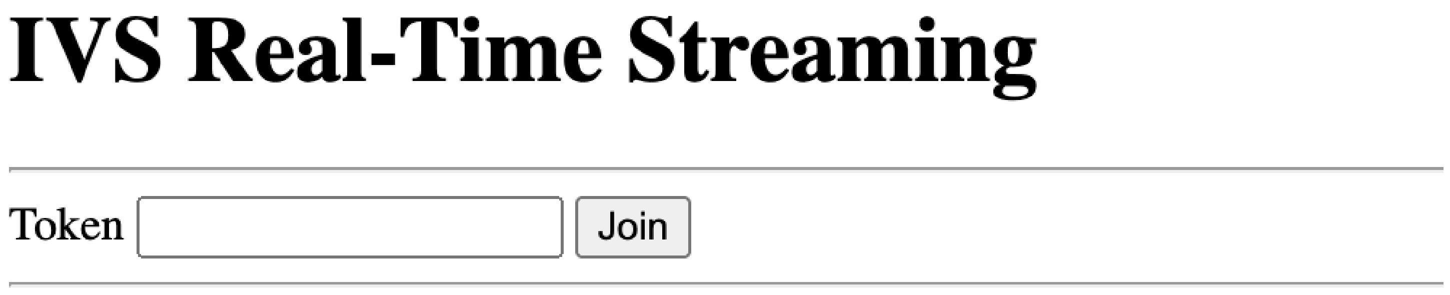
Add Media Container Elements

These elements will hold the media for our local and remote participants. We add a script tag to load our application's logic defined in `app.js`.

```
<!-- Local Participant -->
<div id="local-media"></div>

<!-- Remote Participants -->
<div id="remote-media"></div>
```

```
<!-- Load Script -->
<script src="./app.js"></script>
```

This completes the HTML page and you should see this when loading `index.html` in a browser:



Create app.js

Let's move to defining the contents of our `app.js` file. Begin by importing all the requisite properties from the SDK's global:

```
const {
  Stage,
  LocalStageStream,
  SubscribeType,
  StageEvents,
  ConnectionState,
  StreamType
} = IVSBroadcastClient;
```

Create Application Variables

Establish variables to hold references to our **Join** and **Leave** button HTML elements and store state for the application:

```
let joinButton = document.getElementById("join-button");
let leaveButton = document.getElementById("leave-button");

// Stage management
let stage;
let joining = false;
let connected = false;
let localCamera;
let localMic;
let cameraStageStream;
let micStageStream;
```

Create joinStage 1: Define the Function and Validate Input

The `joinStage` function takes the input token, creates a connection to the stage, and begins to publish video and audio retrieved from `getUserMedia`.

To start, we define the function and validate the state and token input. We'll flesh out this function in the next few sections.

```
const joinStage = async () => {
  if (connected || joining) {
    return;
  }
  joining = true;

  const token = document.getElementById("token").value;

  if (!token) {
    window.alert("Please enter a participant token");
    joining = false;
    return;
  }

  // Fill in with the next sections
};
```

Create joinStage 2: Get Media to Publish

Here is the media that will be published to the stage:

```
async function getCamera() {
  // Use Max Width and Height
  return navigator.mediaDevices.getUserMedia({
    video: true,
    audio: false
  });
}

async function getMic() {
  return navigator.mediaDevices.getUserMedia({
    video: false,
    audio: true
  });
}

// Retrieve the User Media currently set on the page
localCamera = await getCamera();
localMic = await getMic();

// Create StageStreams for Audio and Video
cameraStageStream = new LocalStageStream(localCamera.getVideoTracks()[0]);
micStageStream = new LocalStageStream(localMic.getAudioTracks()[0]);
```

Create joinStage 3: Define the Stage Strategy and Create the Stage

This stage strategy is the heart of the decision logic that the SDK uses to decide what to publish and which participants to subscribe to. For more information on the function's purpose, see Strategy.

This strategy is simple. After joining the stage, publish the streams we just retrieved and subscribe to every remote participant's audio and video:

```
const strategy = {
  stageStreamsToPublish() {
    return [cameraStageStream, micStageStream];
  },
  shouldPublishParticipant() {
    return true;
  },
  shouldSubscribeToParticipant() {
    return SubscribeType.AUDIO_VIDEO;
  }
};

stage = new Stage(token, strategy);
```

Create joinStage 4: Handle Stage Events and Render Media

Stages emit many events. We'll need to listen to the `STAGE_PARTICIPANT_STREAMS_ADDED` and `STAGE_PARTICIPANT_LEFT` to render and remove media to and from the page. A more exhaustive set of events are listed in Events.

Note that we create four helper functions here to assist us in managing necessary DOM elements: `setupParticipant`, `teardownParticipant`, `createVideoEl`, and `createContainer`.

```
stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {
  connected = state === ConnectionState.CONNECTED;

  if (connected) {
    joining = false;
    joinButton.style = "display: none";
    leaveButton.style = "display: inline-block";
  }
});

stage.on(
  StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED,
  (participant, streams) => {
    console.log("Participant Media Added: ", participant, streams);

    let streamsToDisplay = streams;

    if (participant.isLocal) {
      // Ensure to exclude local audio streams, otherwise echo will occur
      streamsToDisplay = streams.filter(
        (stream) => stream.streamType === StreamType.VIDEO
      );
    }

    const videoEl = setupParticipant(participant);
    streamsToDisplay.forEach((stream) =>
      videoEl.srcObject.addTrack(stream.mediaStreamTrack)
    );
  }
);

stage.on(StageEvents.STAGE_PARTICIPANT_LEFT, (participant) => {
  console.log("Participant Left: ", participant);
  teardownParticipant(participant);
});


// Helper functions for managing DOM

function setupParticipant({ isLocal, id }) {
  const groupId = isLocal ? "local-media" : "remote-media";
  const groupContainer = document.getElementById(groupId);

  const participantContainerId = isLocal ? "local" : id;
  const participantContainer = createContainer(participantContainerId);
  const videoEl = createVideoEl(participantContainerId);

  participantContainer.appendChild(videoEl);
  groupContainer.appendChild(participantContainer);

  return videoEl;
}

function teardownParticipant({ isLocal, id }) {
  const groupId = isLocal ? "local-media" : "remote-media";
  const groupContainer = document.getElementById(groupId);
  const participantContainerId = isLocal ? "local" : id;

  const participantDiv = document.getElementById(
    participantContainerId + "-container"
  );
  if (!participantDiv) {
    return;
  }
  groupContainer.removeChild(participantDiv);
}

function createVideoEl(id) {
  const videoEl = document.createElement("video");
  videoEl.id = id;
  videoEl.autoplay = true;
  videoEl.playsInline = true;
  videoEl.srcObject = new MediaStream();
  return videoEl;
}

function createContainer(id) {
  const participantContainer = document.createElement("div");
  participantContainer.classList = "participant-container";
  participantContainer.id = id + "-container";

  return participantContainer;
}
```

Create joinStage 5: Join the Stage

Let's complete our `joinStage` function by finally joining the stage!

```
try {
  await stage.join();
} catch (err) {
  joining = false;
  connected = false;
  console.error(err.message);
}
```

Create leaveStage

Define the `leaveStage` function which the leave button will invoke.

```
const leaveStage = async () => {
  stage.leave();

  joining = false;
  connected = false;
};
```

Initialize Input-Event Handlers

We'll add one last function to our `app.js` file. This function is invoked immediately when the page loads and establishes event handlers for joining and leaving the stage.

```
const init = async () => {
  try {
    // Prevents issues on Safari/FF so devices are not blank
    await navigator.mediaDevices.getUserMedia({ video: true, audio: true });
  } catch (e) {
    alert(
      "Problem retrieving media! Enable camera and microphone permissions."
    );
  }

  joinButton.addEventListener("click", () => {
    joinStage();
  });
  leaveButton.addEventListener("click", () => {
    leaveStage();
    joinButton.style = "display: inline-block";
    leaveButton.style = "display: none";
  });
};

init(); // call the function
```

Run the Application and Provide a Token

At this point you can share the web page locally or with others, open the page, and put in a participant token and join the stage.

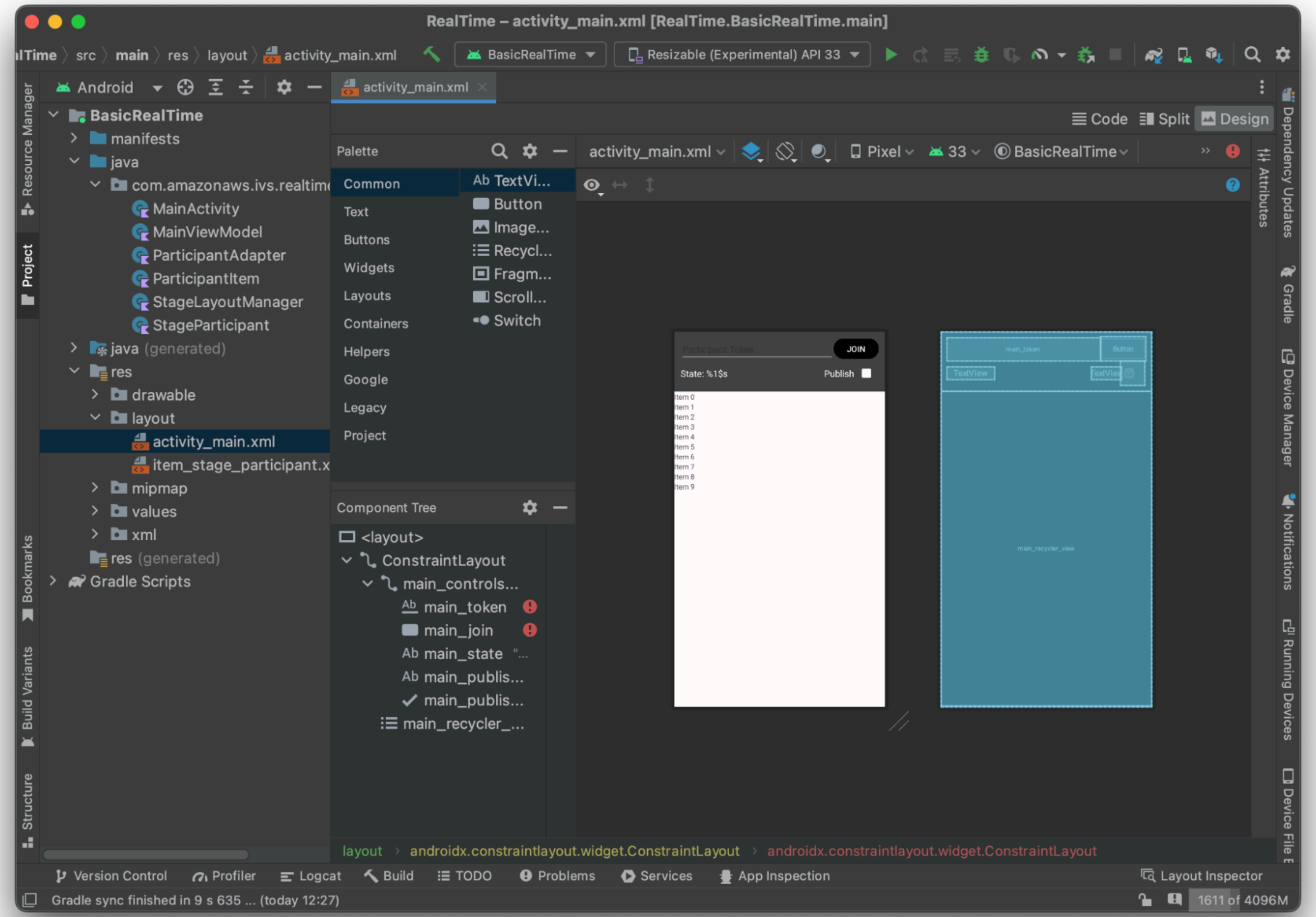
What's Next?

For more detailed examples involving npm, React, and more, see the IVS Broadcast SDK: Web Guide (Real-Time Streaming Guide).

Android

Create Views

We start by creating a simple layout for our app using the auto-created `activity_main.xml` file. The layout contains an `EditText` to add a token, a Join `Button`, a `TextView` to show the stage state, and a `CheckBox` to toggle publishing.



Here is the XML behind the view:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:keepScreenOn="true"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".BasicActivity">

        <androidx.constraintlayout.widget.ConstraintLayout
            android:id="@+id/main_controls_container"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:background="@color/cardview_dark_background"
            android:padding="12dp"
            app:layout_constraintTop_toTopOf="parent">

            <EditText
                android:id="@+id/main_token"
                android:layout_width="0dp"
                android:layout_height="wrap_content"
                android:autofillHints="@null"
                android:backgroundTint="@color/white"
                android:hint="@string/token"
                android:imeOptions="actionDone"
                android:inputType="text"
                android:textColor="@color/white"
                app:layout_constraintEnd_toStartOf="@id/main_join"
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toTopOf="parent" />

            <Button
                android:id="@+id/main_join"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:backgroundTint="@color/black"
                android:text="@string/join"
                android:textAllCaps="true"
                android:textColor="@color/white"
                android:textSize="16sp"
                app:layout_constraintBottom_toBottomOf="@+id/main_token"
                app:layout_constraintEnd_toEndOf="parent"
                app:layout_constraintStart_toEndOf="@id/main_token" />

            <TextView
                android:id="@+id/main_state"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/state"
                android:textColor="@color/white"
                android:textSize="18sp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toBottomOf="@id/main_token" />

            <TextView
```

```
                android:id="@+id/main_publish_text"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/publish"
                android:textColor="@color/white"
                android:textSize="18sp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintEnd_toStartOf="@id/main_publish_checkbox"
                app:layout_constraintTop_toBottomOf="@id/main_token" />

            <CheckBox
                android:id="@+id/main_publish_checkbox"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:buttonTint="@color/white"
                android:checked="true"
                app:layout_constraintBottom_toBottomOf="@id/main_publish_text"
                app:layout_constraintEnd_toEndOf="parent"
                app:layout_constraintTop_toTopOf="@id/main_publish_text" />

        </androidx.constraintlayout.widget.ConstraintLayout>

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/main_recycler_view"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            app:layout_constraintTop_toBottomOf="@+id/main_controls_container"
            app:layout_constraintBottom_toBottomOf="parent" />

    </androidx.constraintlayout.widget.ConstraintLayout>
<layout>
```

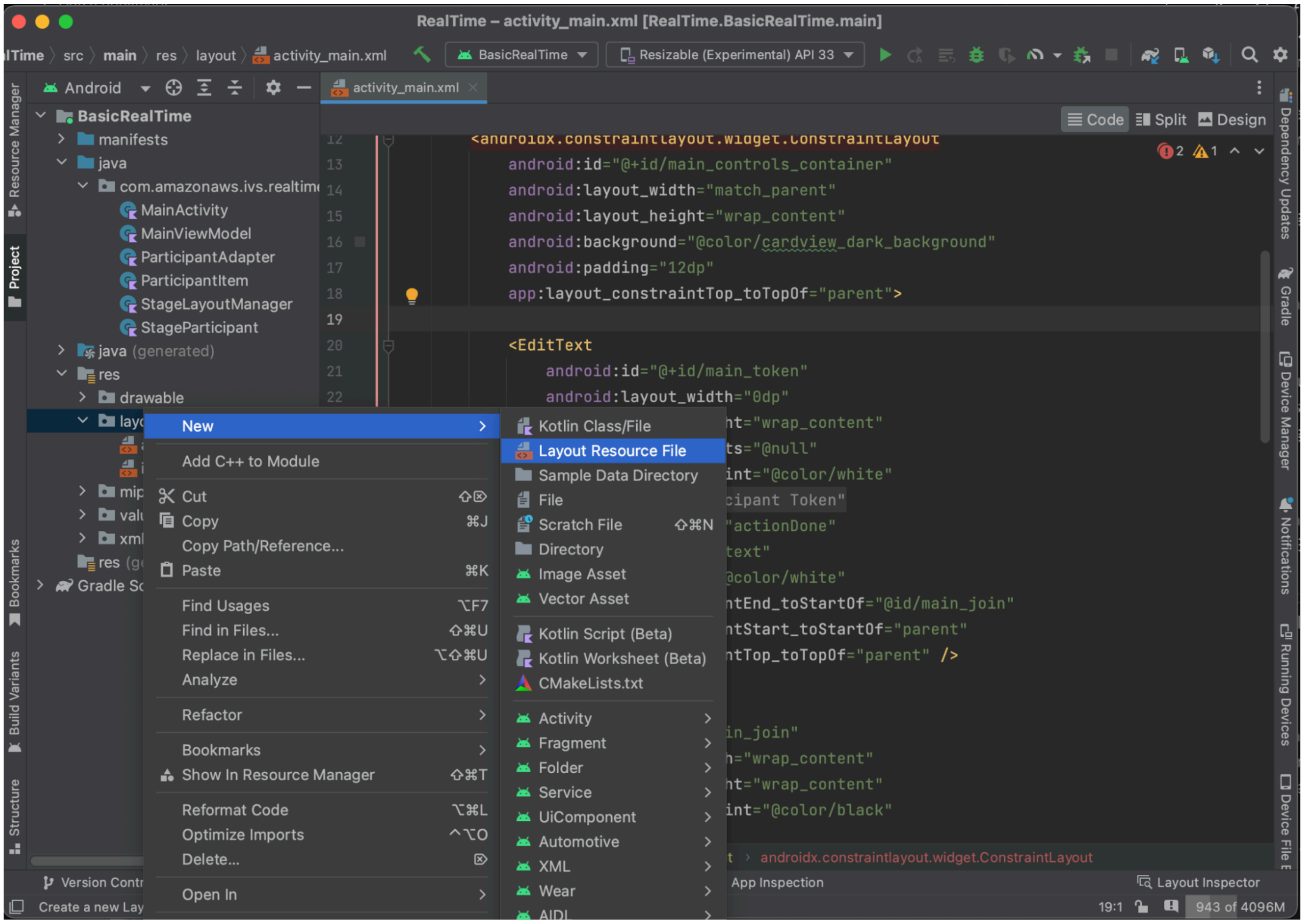
We referenced a couple of string IDs here, so we'll create our entire `strings.xml` file now:

```
<resources>
    <string name="app_name">BasicRealTime</string>
    <string name="join">Join</string>
    <string name="leave">Leave</string>
    <string name="token">Participant Token</string>
    <string name="publish">Publish</string>
    <string name="state">State: %1$s</string>
</resources>
```

Let's link those views in the XML to our `MainActivity.kt`:

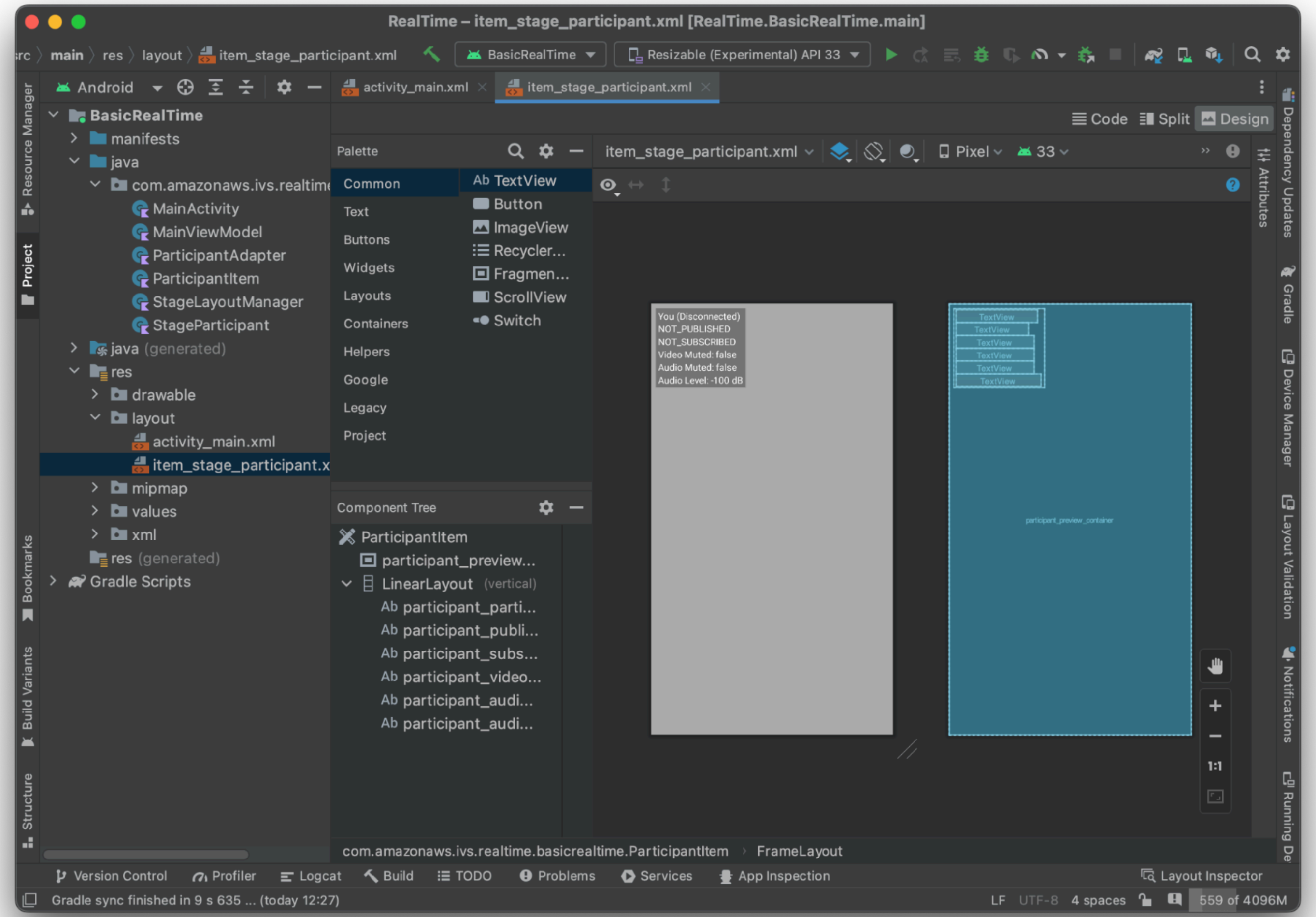
```
import android.widget.Button
import android.widget.CheckBox
import android.widget.EditText
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

private lateinit var checkboxPublish: CheckBox
private lateinit var recyclerView: RecyclerView
private lateinit var buttonJoin: Button
private lateinit var textViewState: TextView
private lateinit var editTextToken: EditText

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    checkboxPublish = findViewById(R.id.main_publish_checkbox)
    recyclerView = findViewById(R.id.main_recycler_view)
    buttonJoin = findViewById(R.id.main_join)
    textViewState = findViewById(R.id.main_state)
    editTextToken = findViewById(R.id.main_token)
}
```

Now we create an item view for our `RecyclerView`. To do this, right-click your `res/layout` directory and select **New > Layout Resource File**. Name this new file `item_stage_participant.xml`.



```
<androidx.constraintlayout.widget.ConstraintLayout
    android:id="@+id/main_controls_container"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/cardview_dark_background"
    android:padding="12dp"
    app:layout_constraintTop_toTopOf="parent">

    <EditText
        android:id="@+id/main_token"
        android:layout_width="0dp"
nt="wrap_content"
ts="@null"
int="@color/white"
cipant Token"
"actionDone"
text"
@color/white"
ntEnd_toStartOf="@id/main_join"
ntStart_toStartOf="parent"
ntTop_toTopOf="parent" />

in_join"
n="wrap_content"
nt="wrap_content"
int="@color/black"
```

The layout for this item is simple: it contains a view for rendering a participant's video stream and a list of labels for displaying information about the participant:



Here is the XML:

```
<?xml version="1.0" encoding="utf-8"?>
<com.amazonaws.ivs.realtime.basicrealtime.ParticipantItem xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <FrameLayout
        android:id="@+id/participant_preview_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:background="@android:color/darker_gray" />
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:background="#50000000"
        android:orientation="vertical"
        android:paddingLeft="4dp"
        android:paddingTop="2dp"
        android:paddingRight="4dp"
        android:paddingBottom="2dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <TextView
            android:id="@+id/participant_participant_id"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="You (Disconnected)" />

        <TextView
            android:id="@+id/participant_publishing"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="NOT_PUBLISHED" />

        <TextView
            android:id="@+id/participant_subscribed"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="NOT_SUBSCRIBED" />

        <TextView
            android:id="@+id/participant_video_muted"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="Video Muted: false" />

        <TextView
            android:id="@+id/participant_audio_muted"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="Audio Muted: false" />

        <TextView
            android:id="@+id/participant_audio_level"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="Audio Level: -100 dB" />

    </LinearLayout>

</com.amazonaws.ivs.realtime.basicrealtime.ParticipantItem>
```

This XML file inflates a class we haven't created yet, `ParticipantItem`. Because the XML includes the full namespace, be sure to update this XML file to your namespace. Let's create this class and set up the views, but otherwise leave it blank for now.

Create a new Kotlin class, `ParticipantItem`:

```
package com.amazonaws.ivs.realtime.basicrealtime

import android.content.Context
import android.util.AttributeSet
import android.widget.FrameLayout
import android.widget.TextView
import kotlin.math.roundToInt

class ParticipantItem @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0,
    defStyleRes: Int = 0,
) : FrameLayout(context, attrs, defStyleAttr, defStyleRes) {
    private lateinit var previewContainer: FrameLayout
    private lateinit var textViewParticipantId: TextView
    private lateinit var textViewPublish: TextView
    private lateinit var textViewSubscribe: TextView
    private lateinit var textViewVideoMuted: TextView
    private lateinit var textViewAudioMuted: TextView
    private lateinit var textViewAudioLevel: TextView

    override fun onFinishInflate() {
        super.onFinishInflate()
        previewContainer = findViewById(R.id.participant_preview_container)
        textViewParticipantId = findViewById(R.id.participant_participant_id)
        textViewPublish = findViewById(R.id.participant_publishing)
        textViewSubscribe = findViewById(R.id.participant_subscribed)
        textViewVideoMuted = findViewById(R.id.participant_video_muted)
        textViewAudioMuted = findViewById(R.id.participant_audio_muted)
        textViewAudioLevel = findViewById(R.id.participant_audio_level)
    }
}
```

Permissions

To use the camera and microphone, you need to request permissions from the user. We follow a standard permissions flow for this:

```
override fun onStart() {
    super.onStart()
    requestPermission()
}

private val requestPermissionLauncher =
    registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions())
 { permissions ->
        if (permissions[Manifest.permission.CAMERA] == true &&
 permissions[Manifest.permission.RECORD_AUDIO] == true) {
            viewModel.permissionGranted() // we will add this later
        }
    }

private val permissions = listOf(
    Manifest.permission.CAMERA,
    Manifest.permission.RECORD_AUDIO,
)
```

```
private fun requestPermission() {
    when {
        this.hasPermissions(permissions) -> viewModel.permissionGranted() // we will
 add this later
        else -> requestPermissionLauncher.launch(permissions.toTypedArray())
    }
}

private fun Context.hasPermissions(permissions: List<String>): Boolean {
    return permissions.all {
        ContextCompat.checkSelfPermission(this, it) ==
 PackageManager.PERMISSION_GRANTED
    }
}
```

App State

Our application keeps track of the participants locally in a `MainViewModel.kt` and the state will be communicated back to the `MainActivity` using Kotlin's StateFlow.

Create a new Kotlin class `MainViewModel`:

```
package com.amazonaws.ivs.realtime.basicrealtime

import android.app.Application
import androidx.lifecycle.AndroidViewModel

class MainViewModel(application: Application) : AndroidViewModel(application),
 Stage.Strategy, StageRenderer {

}
```

In `MainActivity.kt` we manage our view model:

```
import androidx.activity.viewModels

private val viewModel: MainViewModel by viewModels()
```

To use `AndroidViewModel` and these Kotlin `ViewModel` extensions, you'll need to add the following to your module's `build.gradle` file:

```
implementation 'androidx.core:core-ktx:1.10.1'
implementation "androidx.activity:activity-ktx:1.7.2"
implementation 'androidx.appcompat:appcompat:1.6.1'
implementation 'com.google.android.material:material:1.10.0'
implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"

def lifecycle_version = "2.6.1"
implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
```

RecyclerView Adapter

We'll create a simple `RecyclerView.Adapter` subclass to keep track of our participants and update our `RecyclerView` on stage events. But first, we need a class that represents a participant. Create a new Kotlin class `StageParticipant`:

```
package com.amazonaws.ivs.realtime.basicrealtime

import com.amazonaws.ivs.broadcast.Stage
import com.amazonaws.ivs.broadcast.StageStream

class StageParticipant(val isLocal: Boolean, var participantId: String?) {
    var publishState = Stage.PublishState.NOT_PUBLISHED
    var subscribeState = Stage.SubscribeState.NOT_SUBSCRIBED
    var streams = mutableListOf<StageStream>()

    val stableID: String
        get() {
            return if (isLocal) {
                "LocalUser"
            } else {
                requireNotNull(participantId)
            }
        }
}
```

We'll use this class in the `ParticipantAdapter` class that we'll create next. We start by defining the class and creating a variable to track the participants:

```
package com.amazonaws.ivs.realtime.basicrealtime
```

```
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView

class ParticipantAdapter : RecyclerView.Adapter<ParticipantAdapter.ViewHolder>() {

    private val participants = mutableListOf<StageParticipant>()
```

We also have to define our `RecyclerView.ViewHolder` before implementing the rest of the overrides:

```
class ViewHolder(val participantItem: ParticipantItem) :
 RecyclerView.ViewHolder(participantItem)
```

Using this, we can implement the standard `RecyclerView.Adapter` overrides:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    val item = LayoutInflater.from(parent.context)
        .inflate(R.layout.item_stage_participant, parent, false) as ParticipantItem
    return ViewHolder(item)
}

override fun getItemCount(): Int {
    return participants.size
}

override fun getItemId(position: Int): Long =
    participants[position]
        .stableID
        .hashCode()
        .toLong()

override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    return holder.participantItem.bind(participants[position])
}

override fun onBindViewHolder(holder: ViewHolder, position: Int, payloads:
 MutableList<Any>) {
    val updates = payloads.filterIsInstance<StageParticipant>()
    if (updates.isNotEmpty()) {
        updates.forEach { holder.participantItem.bind(it) // implemented later }
    } else {
        super.onBindViewHolder(holder, position, payloads)
    }
}
```

Finally, we add new methods that we will call from our `MainViewModel` when changes to participants are made. These methods are standard CRUD operations on the adapter.

```
fun participantJoined(participant: StageParticipant) {
    participants.add(participant)
    notifyItemInserted(participants.size - 1)
}

fun participantLeft(participantId: String) {
    val index = participants.indexOfFirst { it.participantId == participantId }
    if (index != -1) {
        participants.removeAt(index)
        notifyItemRemoved(index)
    }
}

fun participantUpdated(participantId: String?, update: (participant: StageParticipant)
 -> Unit) {
    val index = participants.indexOfFirst { it.participantId == participantId }
    if (index != -1) {
        update(participants[index])
        notifyItemChanged(index, participants[index])
    }
}
```

Back in `MainViewModel` we need to create and hold a reference to this adapter:

```
internal val participantAdapter = ParticipantAdapter()
```

Stage State

We also need to track some stage state within `MainViewModel`. Let's define those properties now:

```
private val _connectionState = MutableStateFlow(Stage.ConnectionState.DISCONNECTED)
val connectionState = _connectionState.asStateFlow()

private var publishEnabled: Boolean = false
    set(value) {
        field = value
        // Because the strategy returns the value of `checkboxPublish.isChecked`, just
 call `refreshStrategy`.
        stage?.refreshStrategy()
    }

private var deviceDiscovery: DeviceDiscovery? = null
private var stage: Stage? = null
private var streams = mutableListOf<LocalStageStream>()
```

To see your own preview before joining a stage, we create a local participant immediately:

```
init {
    deviceDiscovery = DeviceDiscovery(application)

    // Create a local participant immediately to render our camera preview and
 microphone stats
    val localParticipant = StageParticipant(true, null)
    participantAdapter.participantJoined(localParticipant)
}
```

We want to make sure we clean up these resources when our `ViewModel` is cleaned up. We override `onCleared()` right away, so we don't forget to clean these resources.

```
override fun onCleared() {
    stage?.release()
    deviceDiscovery?.release()
    deviceDiscovery = null
    super.onCleared()
}
```

Now we populate our local `streams` property as soon as permissions are granted, implementing the `permissionsGranted` method that we called earlier:

```
internal fun permissionGranted() {
    val deviceDiscovery = deviceDiscovery ?: return
    streams.clear()
    val devices = deviceDiscovery.listLocalDevices()
    // Camera
    devices
        .filter { it.descriptor.type == Device.Descriptor.DeviceType.CAMERA }
        .maxByOrNull { it.descriptor.position == Device.Descriptor.Position.FRONT }
        ?.let { streams.add(ImageLocalStageStream(it)) }
    // Microphone
    devices
        .filter { it.descriptor.type == Device.Descriptor.DeviceType.MICROPHONE }
        .maxByOrNull { it.descriptor.isDefault }
        ?.let { streams.add(AudioLocalStageStream(it)) }

    stage?.refreshStrategy()

    // Update our local participant with these new streams
    participantAdapter.participantUpdated(null) {
        it.streams.clear()
        it.streams.addAll(streams)
    }
}
```

Implementing the Stage SDK

Three core concepts underlie real-time functionality: stage, strategy, and renderer. The design goal is minimizing the amount of client-side logic necessary to build a working product.

Stage.Strategy

Our `Stage.Strategy` implementation is simple:

```
override fun stageStreamsToPublishForParticipant(
    stage: Stage,
    participantInfo: ParticipantInfo
): MutableList<LocalStageStream> {
    // Return the camera and microphone to be published.
    // This is only called if `shouldPublishFromParticipant` returns true.
    return streams
}

override fun shouldPublishFromParticipant(stage: Stage, participantInfo:
 ParticipantInfo): Boolean {
    return publishEnabled
}

override fun shouldSubscribeToParticipant(stage: Stage, participantInfo:
 ParticipantInfo): Stage.SubscribeType {
    // Subscribe to both audio and video for all publishing participants.
    return Stage.SubscribeType.AUDIO_VIDEO
}
```

To summarize, we publish based on our internal `publishEnabled` state, and if we publish we will publish the streams we collected earlier. Finally for this sample, we always subscribe to other participants, receiving both their audio and video.

StageRenderer

The `StageRenderer` implementation also is fairly simple, though given the number of functions it contains quite a bit more code. The general approach in this renderer is to update our `ParticipantAdapter` when the SDK notifies us of a change to a participant. There are certain scenarios where we handle local participants differently, because we have decided to manage them ourselves so they can see their camera preview before joining.

```
override fun onError(exception: BroadcastException) {
    Toast.makeText(getApplication(), "onError ${exception.localizedMessage}",
 Toast.LENGTH_LONG).show()
    Log.e("BasicRealTime", "onError $exception")
}

override fun onConnectionStateChanged(
    stage: Stage,
    connectionState: Stage.ConnectionState,
    exception: BroadcastException?
) {
    _connectionState.value = connectionState
}

override fun onParticipantJoined(stage: Stage, participantInfo: ParticipantInfo) {
    if (participantInfo.isLocal) {
        // If this is the local participant joining the stage, update the participant
 with a null ID because we
        // manually added that participant when setting up our preview
        participantAdapter.participantUpdated(null) {
            it.participantId = participantInfo.participantId
        }
    } else {
        // If they are not local, add them normally
        participantAdapter.participantJoined(
            StageParticipant(
                participantInfo.isLocal,
                participantInfo.participantId
            )
        )
    }
```

```
}

override fun onParticipantLeft(stage: Stage, participantInfo: ParticipantInfo) {
    if (participantInfo.isLocal) {
        // If this is the local participant leaving the stage, update the ID but keep
 it around because
        // we want to keep the camera preview active
        participantAdapter.participantUpdated(participantInfo.participantId) {
            it.participantId = null
        }
    } else {
        // If they are not local, have them leave normally
        participantAdapter.participantLeft(participantInfo.participantId)
    }
}

override fun onParticipantPublishStateChanged(
    stage: Stage,
    participantInfo: ParticipantInfo,
    publishState: Stage.PublishState
) {
    // Update the publishing state of this participant
    participantAdapter.participantUpdated(participantInfo.participantId) {
        it.publishState = publishState
    }
}

override fun onParticipantSubscribeStateChanged(
    stage: Stage,
    participantInfo: ParticipantInfo,
    subscribeState: Stage.SubscribeState
) {
    // Update the subscribe state of this participant
    participantAdapter.participantUpdated(participantInfo.participantId) {
        it.subscribeState = subscribeState
    }
}

override fun onStreamsAdded(stage: Stage, participantInfo: ParticipantInfo, streams:
 MutableList<StageStream>) {
    // We don't want to take any action for the local participant because we track
 those streams locally
    if (participantInfo.isLocal) {
        return
    }
    // For remote participants, add these new streams to that participant's streams
 array.
    participantAdapter.participantUpdated(participantInfo.participantId) {
        it.streams.addAll(streams)
    }
}

override fun onStreamsRemoved(stage: Stage, participantInfo: ParticipantInfo, streams:
 MutableList<StageStream>) {
    // We don't want to take any action for the local participant because we track
 those streams locally
    if (participantInfo.isLocal) {
        return
    }
    // For remote participants, remove these streams from that participant's streams
 array.
    participantAdapter.participantUpdated(participantInfo.participantId) {
        it.streams.removeAll(streams)
    }
}

override fun onStreamsMutedChanged(
    stage: Stage,
    participantInfo: ParticipantInfo,
    streams: MutableList<StageStream>
) {
    // We don't want to take any action for the local participant because we track
 those streams locally
    if (participantInfo.isLocal) {
        return
    }
    // For remote participants, notify the adapter that the participant has been
 updated. There is no need to modify
    // the `streams` property on the `StageParticipant` because it is the same
 `StageStream` instance. Just
    // query the `isMuted` property again.
    participantAdapter.participantUpdated(participantInfo.participantId) {}
}
```

Implementing a Custom RecyclerView LayoutManager

Laying out different numbers of participants can be complex. You want them to take up the entire parent view's frame but you don't want to handle each participant configuration independently. To make this easy, we'll walk through implementing a `RecyclerView.LayoutManager`.

Create another new class, `StageLayoutManager`, which should extend `GridLayoutManager`. This class is designed to calculate the layout for each participant based on the number of participants in a flow-based row/column layout. Each row is the same height as the others, but columns can be different widths per row. See the code comment above the `layouts` variable for a description of how to customize this behavior.

```
package com.amazonaws.ivs.realtime.basicrealtime

import android.content.Context
import androidx.recyclerview.widget.GridLayoutManager
import androidx.recyclerview.widget.RecyclerView

class StageLayoutManager(context: Context?) : GridLayoutManager(context, 6) {

    companion object {
        /**
         * This 2D array contains the description of how the grid of participants
 should be rendered
         * The index of the 1st dimension is the number of participants needed to
 active that configuration
         * Meaning if there is 1 participant, index 0 will be used. If there are 5
 participants, index 4 will be used.
         *
         * The 2nd dimension is a description of the layout. The length of the array is
 the number of rows that
         * will exist, and then each number within that array is the number of columns
 in each row.
         *
         * See the code comments next to each index for concrete examples.
         *
         * This can be customized to fit any layout configuration needed.
         */
        val layouts: List<List<Int>> = listOf(
            // 1 participant
            listOf(1), // 1 row, full width
            // 2 participants
            listOf(1, 1), // 2 rows, all columns are full width
            // 3 participants
            listOf(1, 2), // 2 rows, first row's column is full width then 2nd row's
 columns are 1/2 width
            // 4 participants
            listOf(2, 2), // 2 rows, all columns are 1/2 width
            // 5 participants
            listOf(1, 2, 2), // 3 rows, first row's column is full width, 2nd and 3rd
 row's columns are 1/2 width
            // 6 participants
            listOf(2, 2, 2), // 3 rows, all column are 1/2 width
            // 7 participants
            listOf(2, 2, 3), // 3 rows, 1st and 2nd row's columns are 1/2 width, 3rd
 row's columns are 1/3rd width
            // 8 participants
            listOf(2, 3, 3),
            // 9 participants
            listOf(3, 3, 3),
            // 10 participants
            listOf(2, 3, 2, 3),
            // 11 participants
            listOf(2, 3, 3, 3),
            // 12 participants
            listOf(3, 3, 3, 3),
        )
    }

    init {
        spanSizeLookup = object : SpanSizeLookup() {
            override fun getSpanSize(position: Int): Int {
                if (itemCount <= 0) {
                    return 1
                }
                // Calculate the row we're in
                val config = layouts[itemCount - 1]
                var row = 0
                var curPosition = position
                while (curPosition - config[row] >= 0) {
                    curPosition -= config[row]
                    row++
                }
                // spanCount == max spans, config[row] = number of columns we want
                // So spanCount / config[row] would be something like 6 / 3 if we want
 3 columns.
```

```
                // So this will take up 2 spans, with a max of 6 is 1/3rd of the view.
                return spanCount / config[row]
            }
        }
    }

    override fun onLayoutChildren(recycler: RecyclerView.Recycler?, state:
 RecyclerView.State?) {
        if (itemCount <= 0 || state?.isPreLayout == true) return

        val parentHeight = height
        val itemHeight = parentHeight / layouts[itemCount - 1].size // height divided
 by number of rows.

        // Set the height of each view based on how many rows exist for the current
 participant count.
        for (i in 0 until childCount) {
            val child = getChildAt(i) ?: continue
            val layoutParams = child.layoutParams as RecyclerView.LayoutParams
            if (layoutParams.height != itemHeight) {
                layoutParams.height = itemHeight
                child.layoutParams = layoutParams
            }
        }
        // After we set the height for all our views, call super.
        // This works because our RecyclerView can not scroll and all views are always
 visible with stable IDs.
        super.onLayoutChildren(recycler, state)
    }

    override fun canScrollVertically(): Boolean = false
    override fun canScrollHorizontally(): Boolean = false
}
```

Back in `MainActivity.kt` we need to set the adapter and layout manager for our `RecyclerView`:

```
// In onCreate after setting recyclerView.
recyclerView.layoutManager = StageLayoutManager(this)
recyclerView.adapter = viewModel.participantAdapter
```

Hooking Up UI Actions

We are getting close; there are just a few UI actions that we need to hook up.

First we'll have our `MainActivity` observe the `StateFlow` changes from `MainViewModel`:

```
// At the end of your onCreate method
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.CREATED) {
        viewModel.connectionState.collect { state ->
            buttonJoin.setText(if (state == ConnectionState.DISCONNECTED) R.string.join
 else R.string.leave)
            textViewState.text = getString(R.string.state, state.name)
        }
    }
}
```

Next we add listeners to our Join button and Publish checkbox:

```
buttonJoin.setOnClickListener {
    viewModel.joinStage(editTextToken.text.toString())
}
checkboxPublish.setOnCheckedChangeListener { _, isChecked ->
    viewModel.setPublishEnabled(isChecked)
}
```

Both of the above call functionality in our `MainViewModel`, which we implement now:

```
internal fun joinStage(token: String) {
    if (_connectionState.value != Stage.ConnectionState.DISCONNECTED) {
        // If we're already connected to a stage, leave it.
        stage?.leave()
    } else {
        if (token.isEmpty()) {
            Toast.makeText(getApplication(), "Empty Token", Toast.LENGTH_SHORT).show()
            return
        }
        try {
            // Destroy the old stage first before creating a new one.
            stage?.release()
            val stage = Stage(getApplication(), token, this)
            stage.addRenderer(this)
            stage.join()
            this.stage = stage
        } catch (e: BroadcastException) {
            Toast.makeText(getApplication(), "Failed to join stage
 ${e.localizedMessage}", Toast.LENGTH_LONG).show()
            e.printStackTrace()
        }
    }
}

internal fun setPublishEnabled(enabled: Boolean) {
    publishEnabled = enabled
}
```

Rendering the Participants

Finally, we need to render the data we receive from the SDK onto the participant item that we created earlier. We already have the `RecyclerView` logic finished, so we just need to implement the `bind` API in `ParticipantItem`.

We'll start by adding the empty function and then walk through it step by step:

```
fun bind(participant: StageParticipant) {

}
```

First we'll handle the easy state, the participant ID, publish state, and subscribe state. For these, we just update our `TextViews` directly:

```
val participantId = if (participant.isLocal) {
    "You (${participant.participantId ?: "Disconnected"})"
} else {
    participant.participantId
}
textViewParticipantId.text = participantId
textViewPublish.text = participant.publishState.name
textViewSubscribe.text = participant.subscribeState.name
```

Next we'll update the audio and video muted states. To get the muted state, we need to find the `ImageDevice` and `AudioDevice` from the streams array. To optimize performance, we remember the last attached device IDs.

```
// This belongs outside the `bind` API.
private var imageDeviceUrn: String? = null
private var audioDeviceUrn: String? = null

// This belongs inside the `bind` API.
val newImageStream = participant
    .streams
    .firstOrNull { it.device is ImageDevice }
textViewVideoMuted.text = if (newImageStream != null) {
    if (newImageStream.muted) "Video muted" else "Video not muted"
} else {
    "No video stream"
}

val newAudioStream = participant
    .streams
    .firstOrNull { it.device is AudioDevice }
textViewAudioMuted.text = if (newAudioStream != null) {
    if (newAudioStream.muted) "Audio muted" else "Audio not muted"
} else {
    "No audio stream"
}
```

Finally we want to render a preview for the `imageDevice`:

```
if (newImageStream?.device?.descriptor?.urn != imageDeviceUrn) {
    // If the device has changed, remove all subviews from the preview container
    previewContainer.removeAllViews()
    (newImageStream?.device as? ImageDevice)?.let {
        val preview = it.getPreviewView(BroadcastConfiguration.AspectMode.FIT)
        previewContainer.addView(preview)
        preview.layoutParams = FrameLayout.LayoutParams(
            FrameLayout.LayoutParams.MATCH_PARENT,
            FrameLayout.LayoutParams.MATCH_PARENT
        )
    }
}
imageDeviceUrn = newImageStream?.device?.descriptor?.urn
```

And we display audio stats from the `audioDevice`:

```
if (newAudioStream?.device?.descriptor?.urn != audioDeviceUrn) {
    (newAudioStream?.device as? AudioDevice)?.let {
        it.setStatsCallback { _, rms ->
            textViewAudioLevel.text = "Audio Level: ${rms.roundToInt()} dB"
        }
    }
}
audioDeviceUrn = newAudioStream?.device?.descriptor?.urn
```

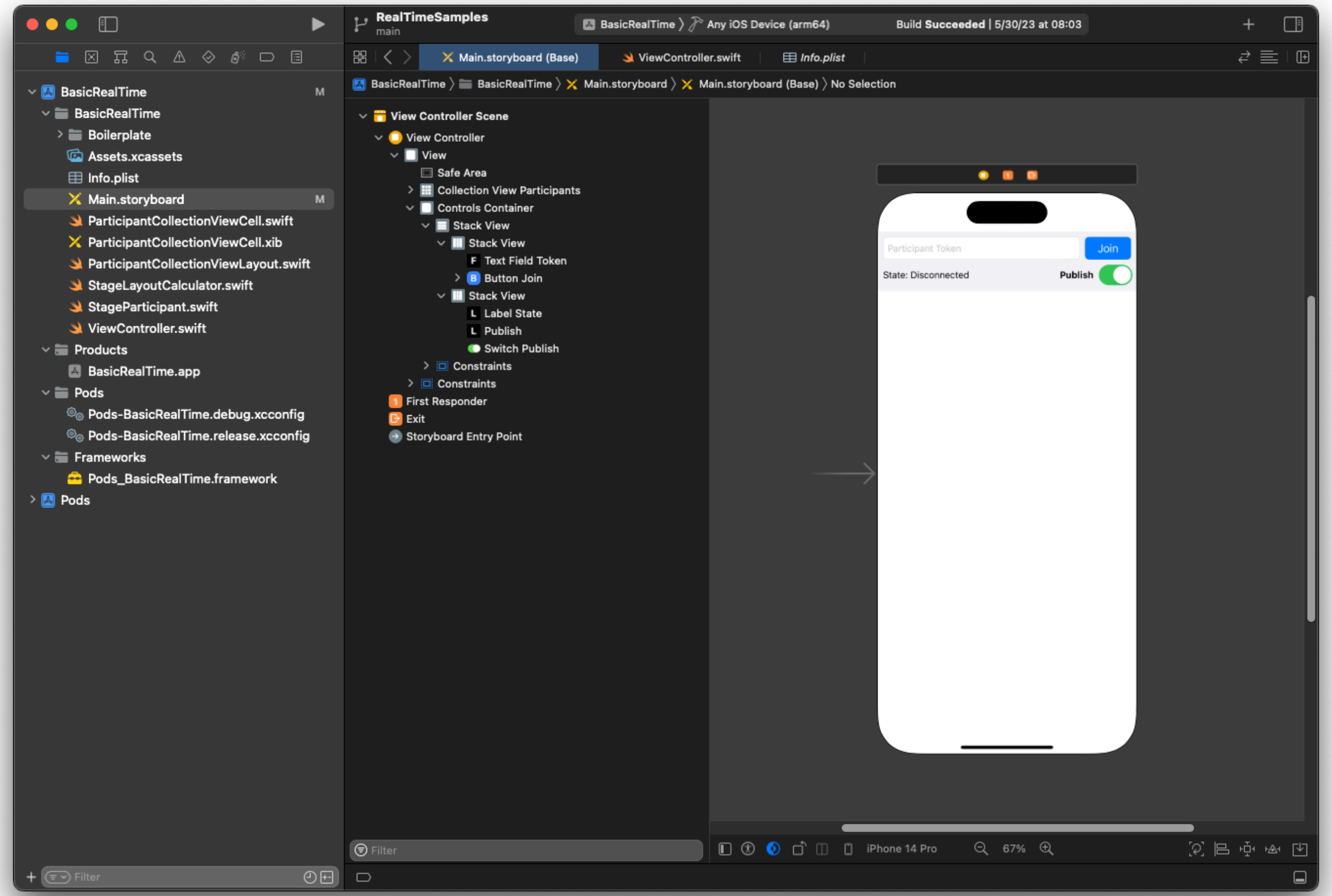
iOS

Create Views

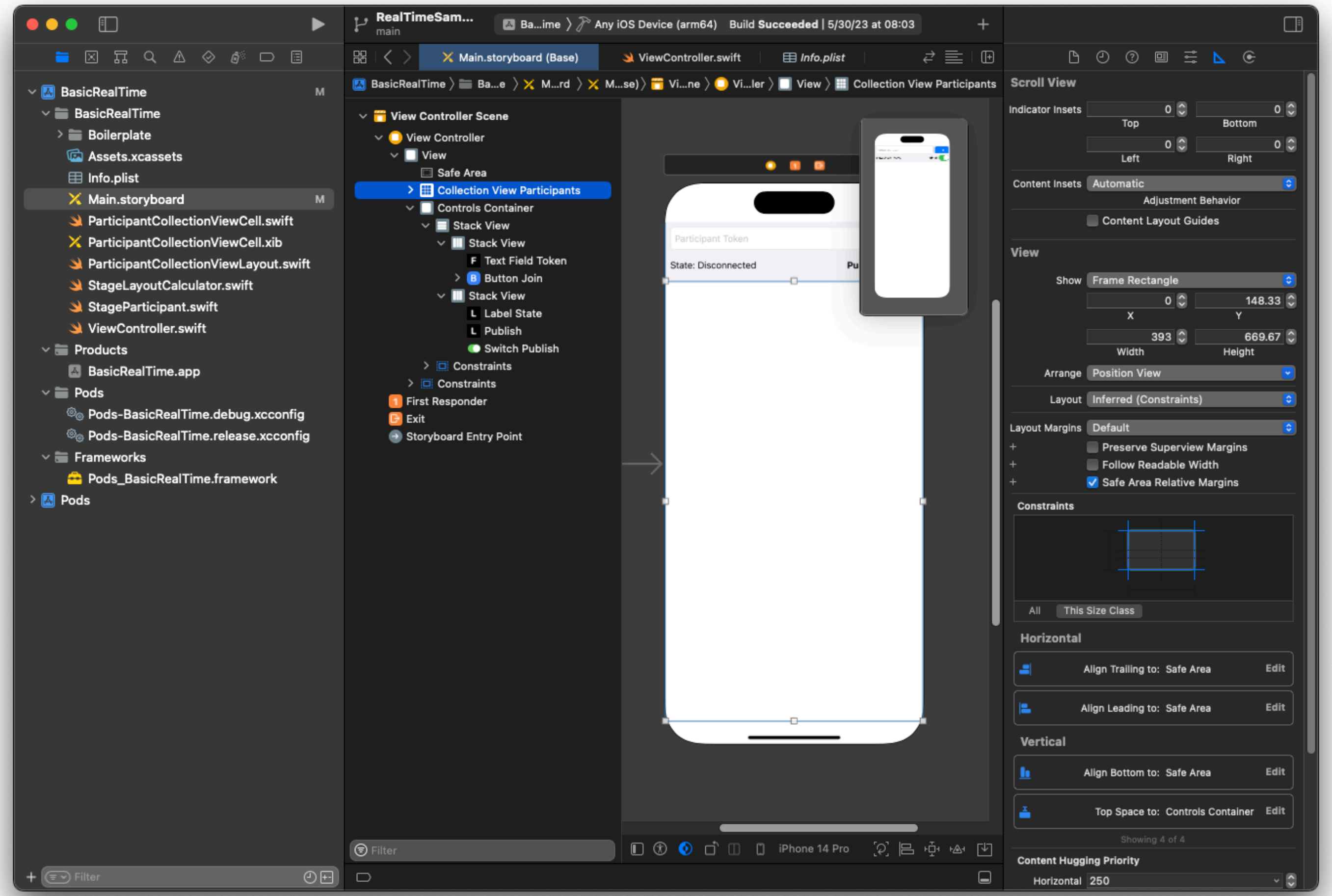
We start by using the auto-created `ViewController.swift` file to import `AmazonIVSBroadcast` and then add some `@IBOutlets` to link:

```
import AmazonIVSBroadcast

class ViewController: UIViewController {

    @IBOutlet private var textFieldToken: UITextField!
    @IBOutlet private var buttonJoin: UIButton!
    @IBOutlet private var labelState: UILabel!
    @IBOutlet private var switchPublish: UISwitch!
    @IBOutlet private var collectionViewParticipants: UICollectionView!
```

Now we create those views and link them up in `Main.storyboard`. Here is the view structure that we'll use:

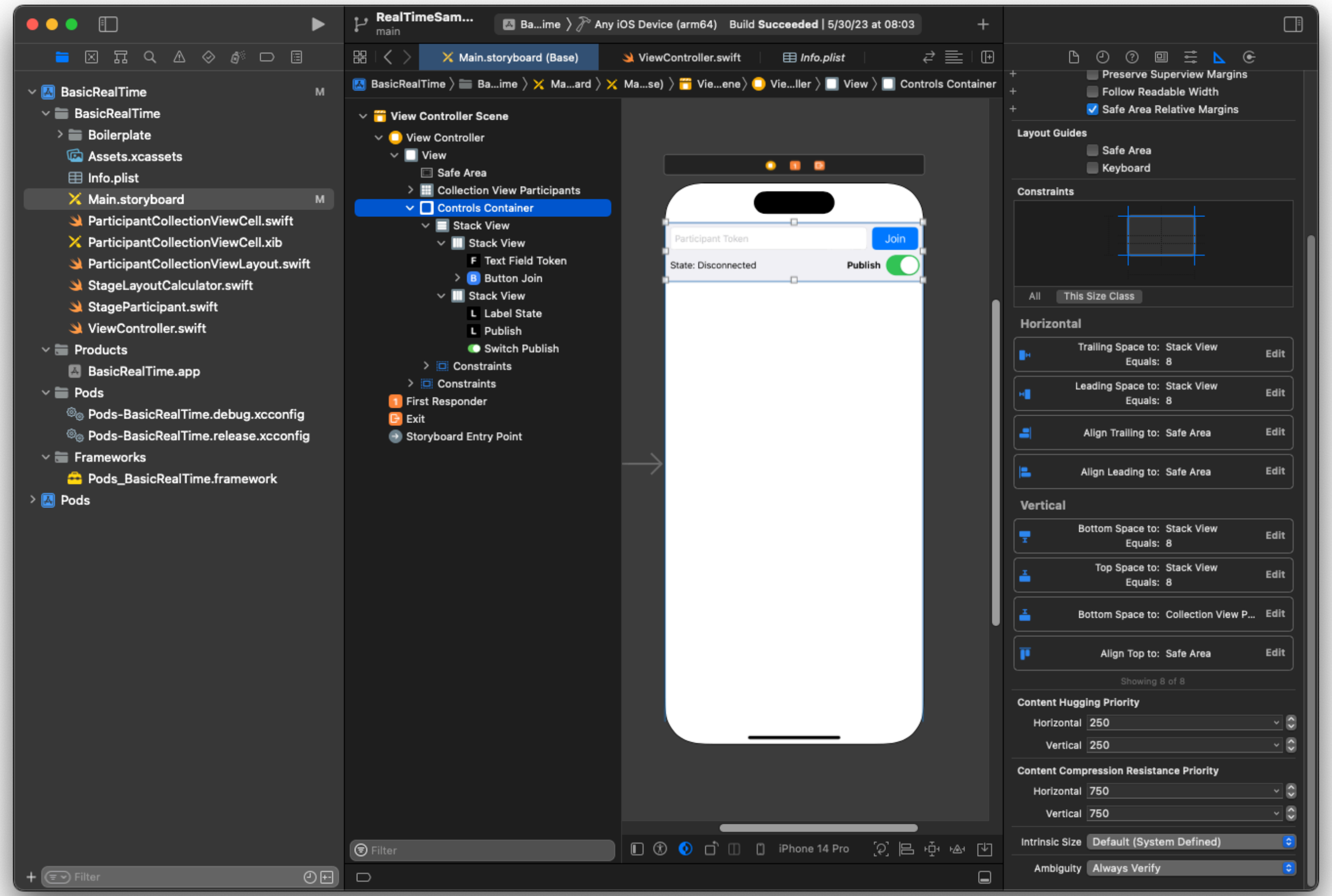


For AutoLayout configuration, we need to customize three views. The first view is **Collection View Participants** (a `UICollectionView`). Bound **Leading**, **Trailing**, and **Bottom** to **Safe Area**. Also bound **Top** to **Controls Container**.



Indicator Insets: Top 0, Bottom 0, Left 0, Right 0
Content Insets: Automatic (Adjustment Behavior)
Content Layout Guides: unchecked
Show: Frame Rectangle
X 0, Y 148.33, Width 393, Height 669.67
Arrange: Position View
Layout: Inferred (Constraints)
Layout Margins: Default
Preserve Superview Margins: unchecked
Follow Readable Width: unchecked
Safe Area Relative Margins: checked
Horizontal: Align Trailing to: Safe Area — Edit; Align Leading to: Safe Area — Edit
Vertical: Align Bottom to: Safe Area — Edit; Top Space to: Controls Container — Edit
Content Hugging Priority: Horizontal 250

The second view is **Controls Container**. Bound **Leading**, **Trailing**, and **Top** to **Safe Area**:



Preserve Superview Margins
Follow Readable Width
Safe Area Relative Margins

Layout Guides
Safe Area
Keyboard

Constraints

Horizontal
- Trailing Space to: Stack View Equals: 8
- Leading Space to: Stack View Equals: 8
- Align Trailing to: Safe Area
- Align Leading to: Safe Area

Vertical
- Bottom Space to: Stack View Equals: 8
- Top Space to: Stack View Equals: 8
- Bottom Space to: Collection View P...
- Align Top to: Safe Area

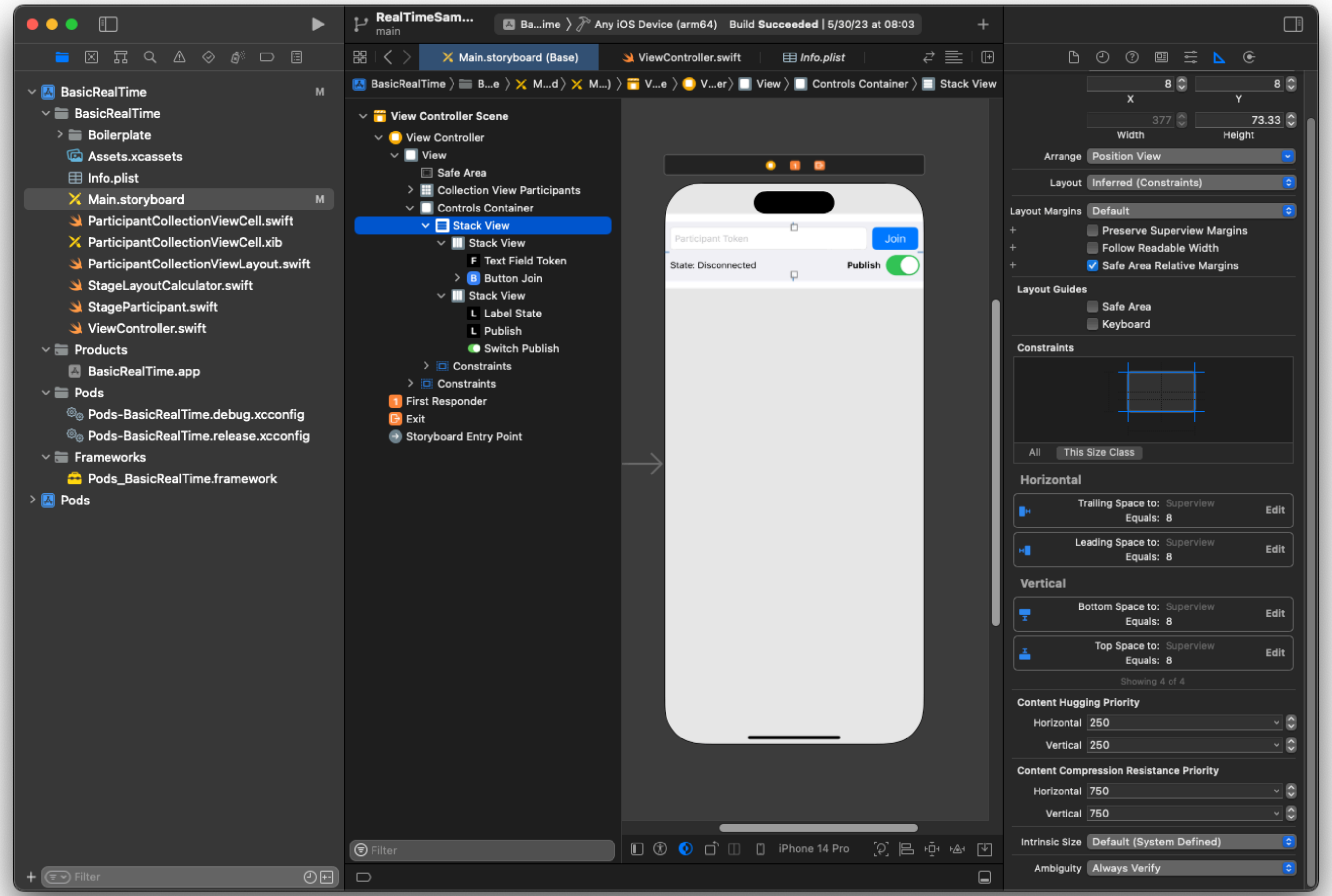
Showing 8 of 8

Content Hugging Priority
Horizontal 250
Vertical 250

Content Compression Resistance Priority
Horizontal 750
Vertical 750

Intrinsic Size: Default (System Defined)
Ambiguity: Always Verify

The third and last view is **Vertical Stack View**. Bound **Top**, **Leading**, **Trailing**, and **Bottom** to **Superview**. For styling, set the spacing to 8 instead of 0.



X: 8
Y: 8
Width: 377
Height: 73.33
Arrange: Position View
Layout: Inferred (Constraints)
Layout Margins: Default
Preserve Superview Margins
Follow Readable Width
Safe Area Relative Margins ✓

Layout Guides
Safe Area
Keyboard

Constraints
All | This Size Class

Horizontal
Trailing Space to: Superview Equals: 8 Edit
Leading Space to: Superview Equals: 8 Edit

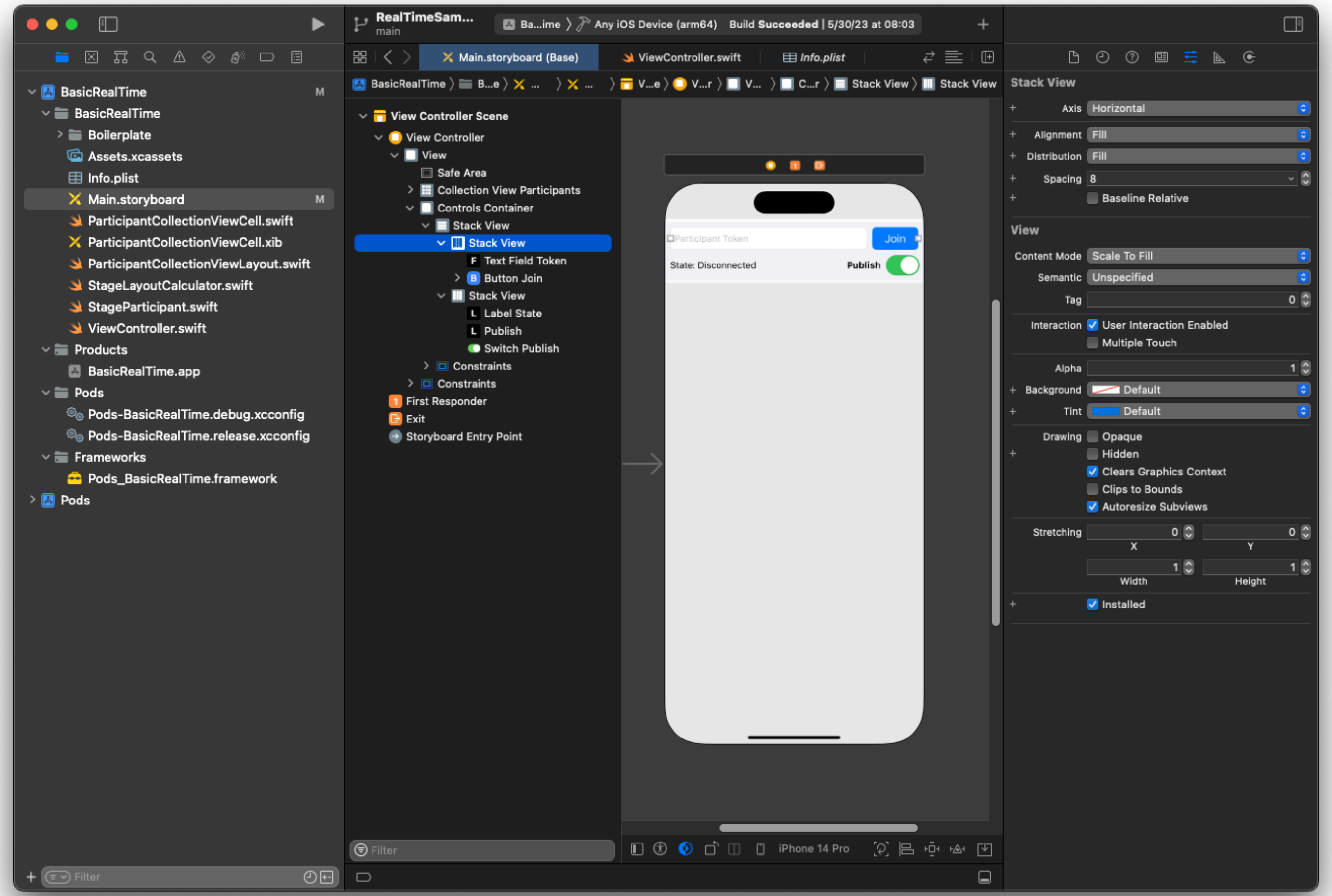
Vertical
Bottom Space to: Superview Equals: 8 Edit
Top Space to: Superview Equals: 8 Edit
Showing 4 of 4

Content Hugging Priority
Horizontal 250
Vertical 250

Content Compression Resistance Priority
Horizontal 750
Vertical 750

Intrinsic Size: Default (System Defined)
Ambiguity: Always Verify

The **UIStackViews** will handle the layout of the remaining views. For all three **UIStackViews**, use **Fill** as the **Alignment** and **Distribution**.



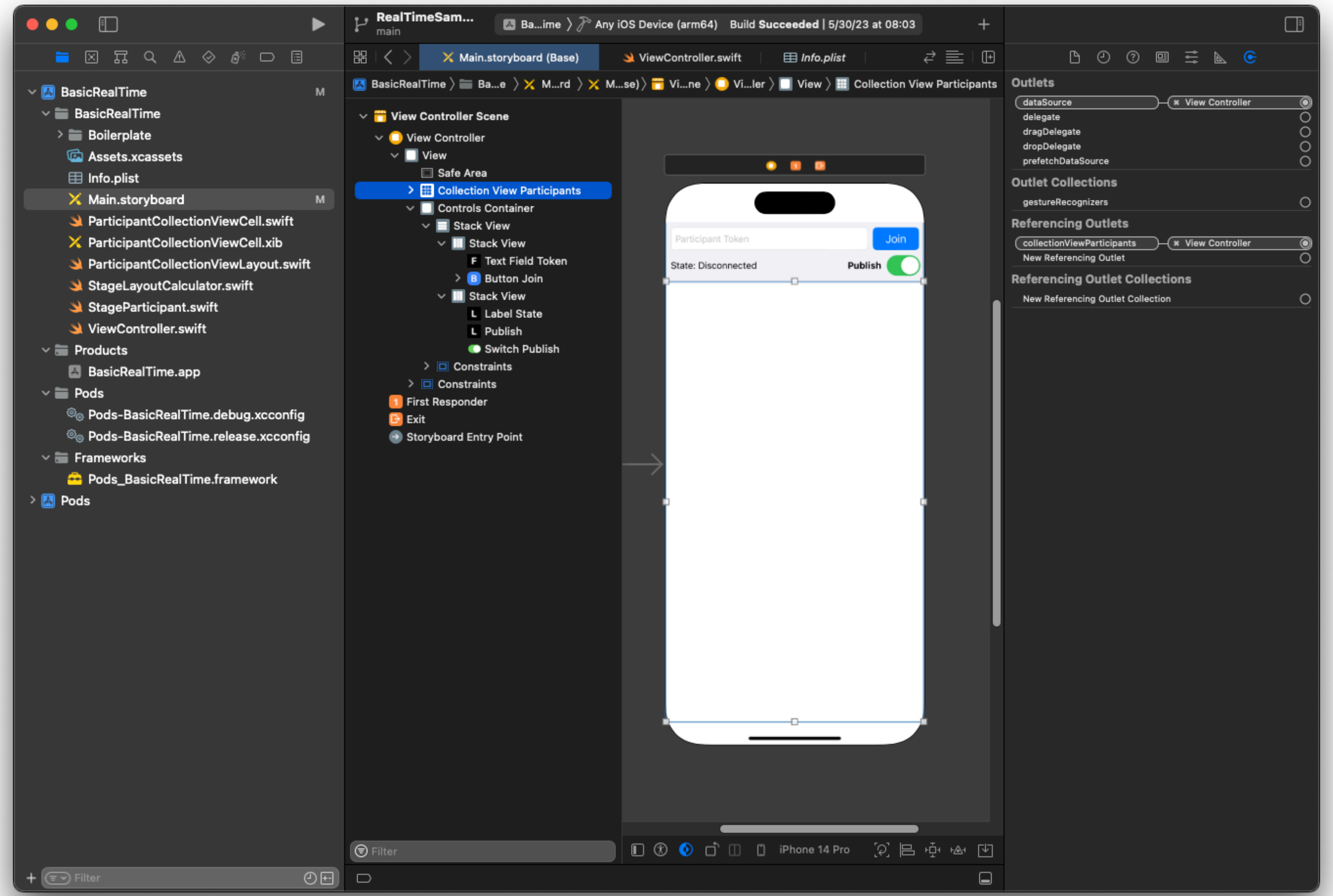
Stack View
Axis: Horizontal
Alignment: Fill
Distribution: Fill
Spacing: 8
Baseline Relative: ☐

View
Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: ✓ User Interaction Enabled; ☐ Multiple Touch
Alpha: 1
Background: Default
Tint: Default
Drawing: ☐ Opaque; ☐ Hidden; ✓ Clears Graphics Context; ☐ Clips to Bounds; ✓ Autoresize Subviews
Stretching: X 0, Y 0, Width 1, Height 1
✓ Installed

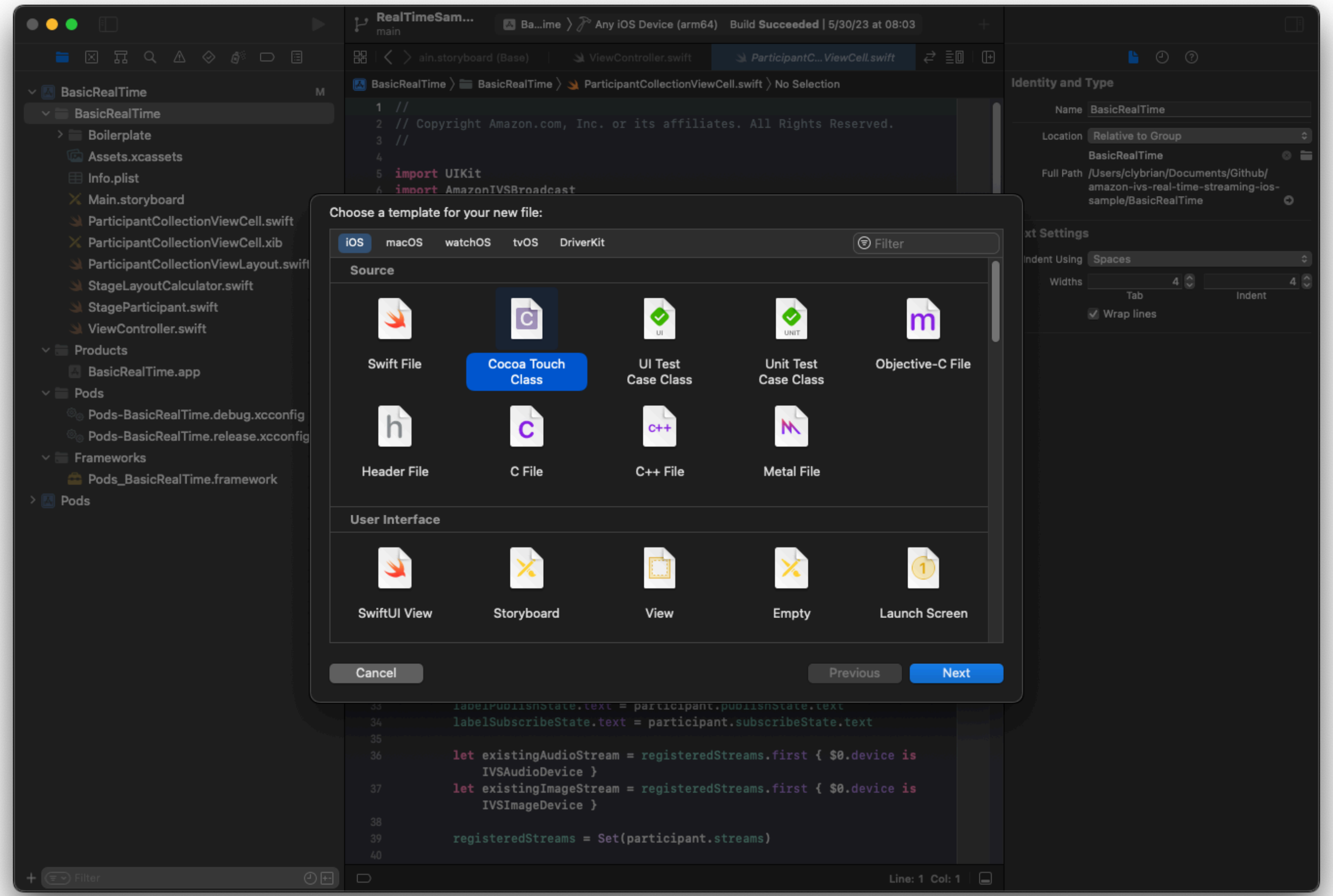
Finally, let's link these views to our `ViewController`. From above, map the following views:

- **Text Field Join** binds to `textFieldToken`.
- **Button Join** binds to `buttonJoin`.
- **Label State** binds to `labelState`.
- **Switch Publish** binds to `switchPublish`.
- **Collection View Participants** binds to `collectionViewParticipants`.

Also use this time to set the `dataSource` of the **Collection View Participants** item to the owning `ViewController`:



Now we create the `UICollectionViewCell` subclass in which to render the participants. Start by creating a new **Cocoa Touch Class** file:

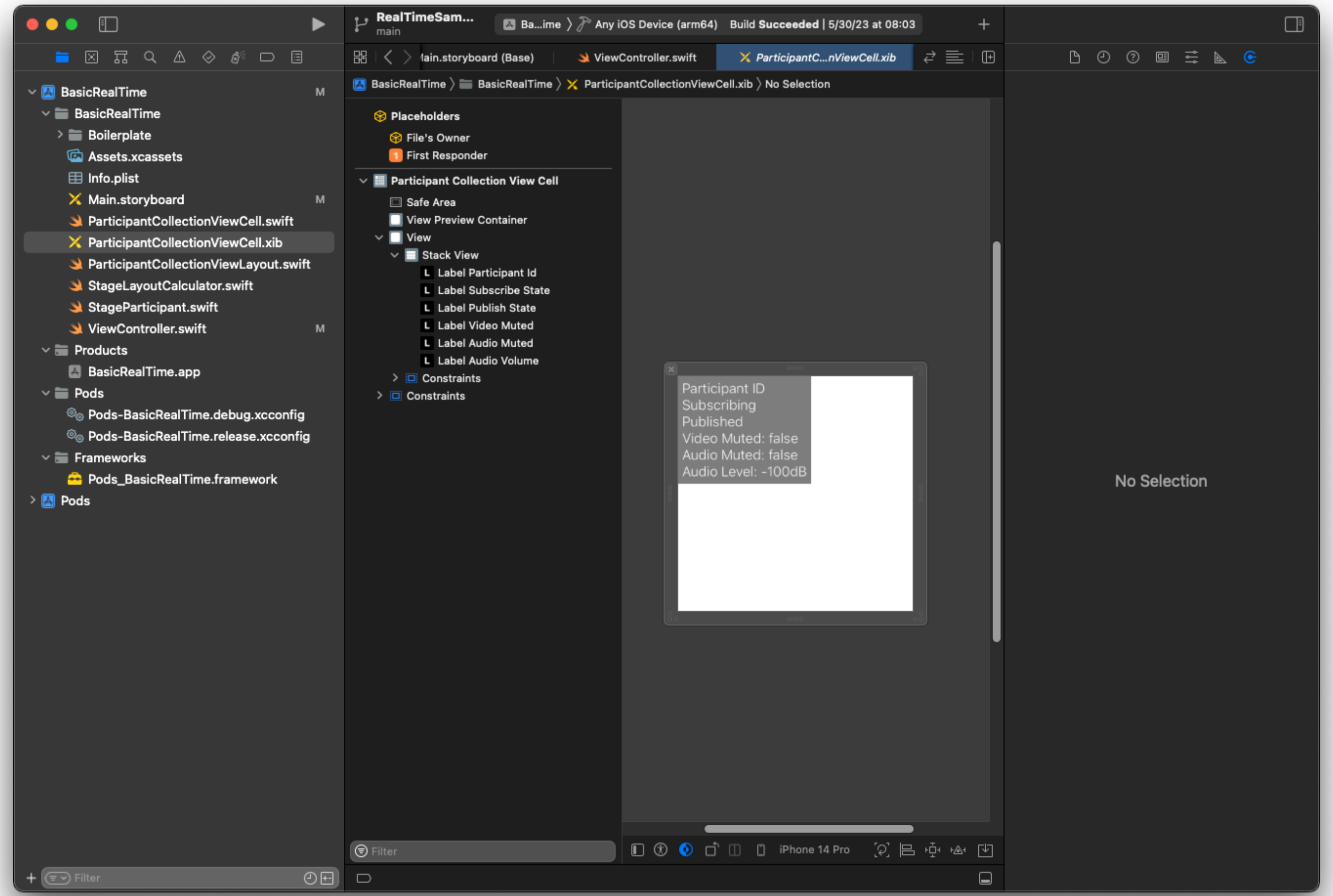

```
//
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
//

import UIKit
import AmazonIVSBroadcast
```

```
        labelPublishState.text = participant.publishState.text
        labelSubscribeState.text = participant.subscribeState.text

        let existingAudioStream = registeredStreams.first { $0.device is
            IVSAudioDevice }
        let existingImageStream = registeredStreams.first { $0.device is
            IVSImageDevice }

        registeredStreams = Set(participant.streams)
```

Name it `ParticipantUICollectionViewCell` and make it a subclass of `UICollectionViewCell` in Swift. We start in the Swift file again, creating our `@IBOutlets` to link:

```
import AmazonIVSBroadcast

class ParticipantCollectionViewCell: UICollectionViewCell {

    @IBOutlet private var viewPreviewContainer: UIView!
    @IBOutlet private var labelParticipantId: UILabel!
    @IBOutlet private var labelSubscribeState: UILabel!
    @IBOutlet private var labelPublishState: UILabel!
    @IBOutlet private var labelVideoMuted: UILabel!
    @IBOutlet private var labelAudioMuted: UILabel!
    @IBOutlet private var labelAudioVolume: UILabel!
```

In the associated XIB file, create this view hierarchy:



RealTimeSam... main | Ba...ime 〉 Any iOS Device (arm64) | Build Succeeded | 5/30/23 at 08:03

Main.storyboard (Base) | ViewController.swift | ParticipantC...nViewCell.xib

BasicRealTime 〉 BasicRealTime 〉 ParticipantCollectionViewCell.xib 〉 No Selection

```
BasicRealTime                                M
  BasicRealTime
    Boilerplate
    Assets.xcassets
    Info.plist
    Main.storyboard                          M
    ParticipantCollectionViewCell.swift
    ParticipantCollectionViewCell.xib
    ParticipantCollectionViewLayout.swift
    StageLayoutCalculator.swift
    StageParticipant.swift
    ViewController.swift                     M
  Products
    BasicRealTime.app
  Pods
    Pods-BasicRealTime.debug.xcconfig
    Pods-BasicRealTime.release.xcconfig
  Frameworks
    Pods_BasicRealTime.framework
Pods
```

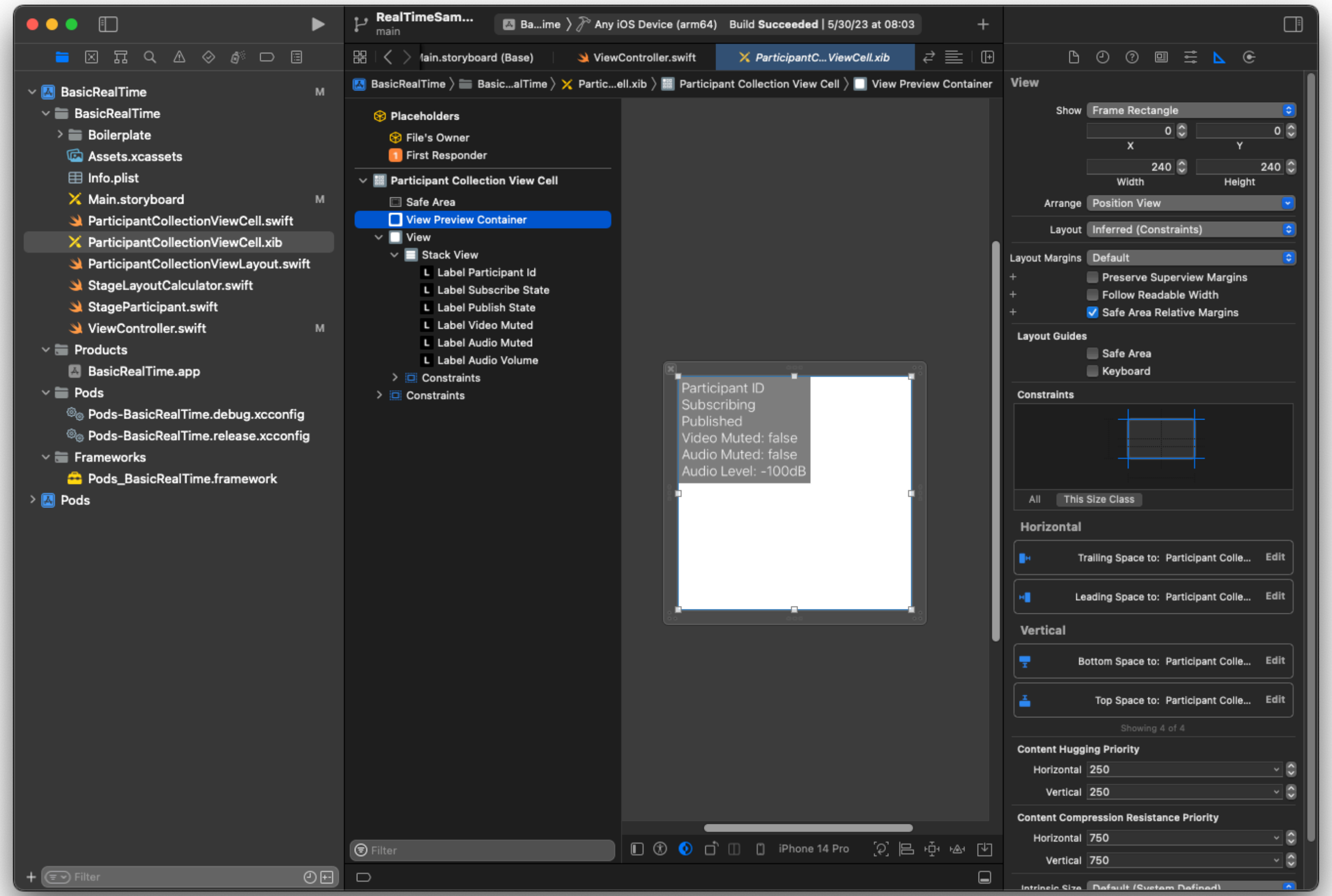
```
Placeholders
  File's Owner
  First Responder
Participant Collection View Cell
  Safe Area
  View Preview Container
  View
    Stack View
      Label Participant Id
      Label Subscribe State
      Label Publish State
      Label Video Muted
      Label Audio Muted
      Label Audio Volume
    Constraints
  Constraints
```

```
Participant ID
Subscribing
Published
Video Muted: false
Audio Muted: false
Audio Level: -100dB
```

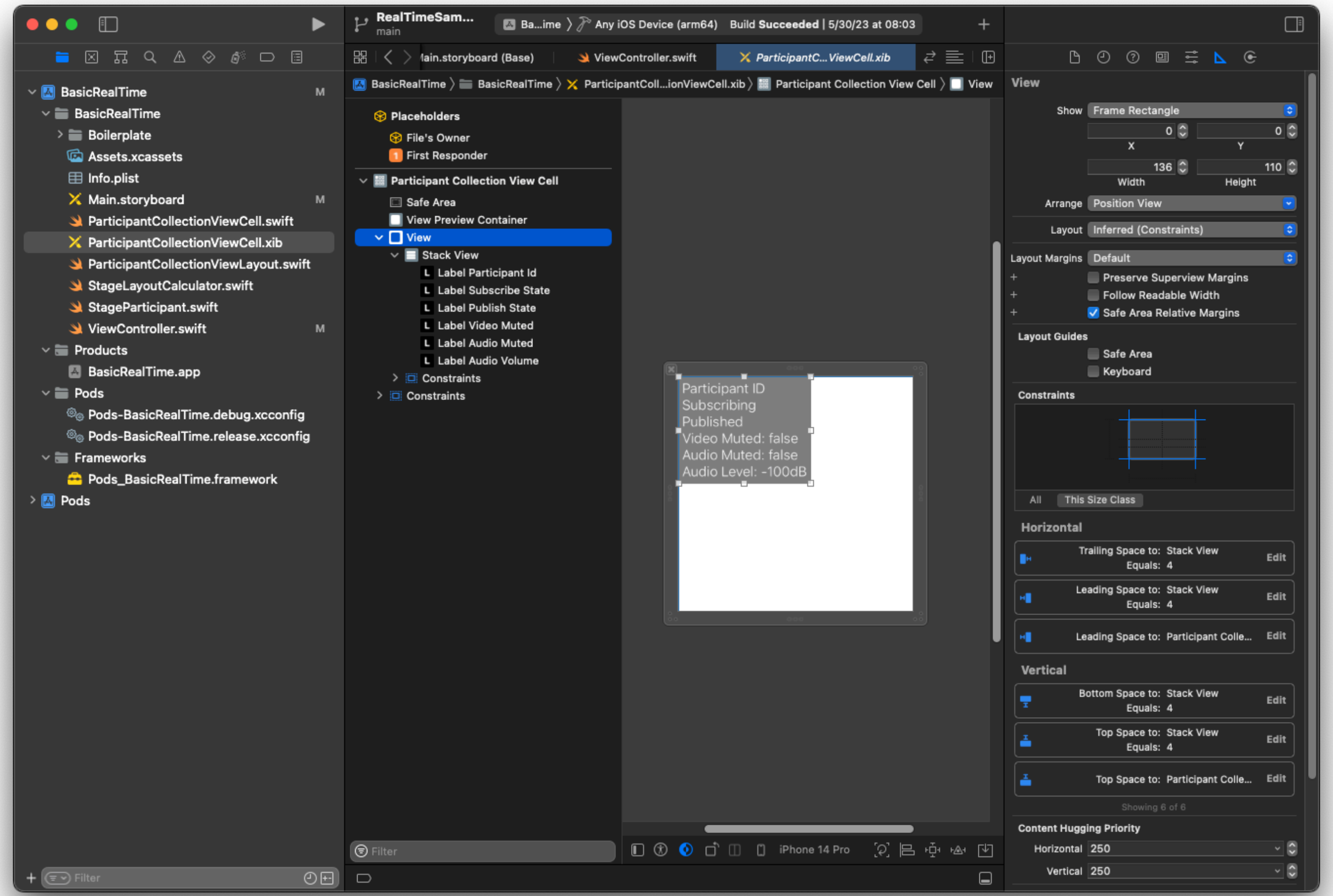
iPhone 14 Pro

No Selection

For AutoLayout, we’ll modify three views again. The first view is **View Preview Container**. Set **Trailing**, **Leading**, **Top**, and **Bottom** to **Participant Collection View Cell**.



The second view is **View**. Set **Leading** and **Top** to **Participant Collection View Cell** and change the value to 4.



View
Show: Frame Rectangle
X: 0 Y: 0
Width: 136 Height: 110
Arrange: Position View
Layout: Inferred (Constraints)
Layout Margins: Default
Preserve Superview Margins
Follow Readable Width
Safe Area Relative Margins ✓
Layout Guides
Safe Area
Keyboard
Constraints
All | This Size Class

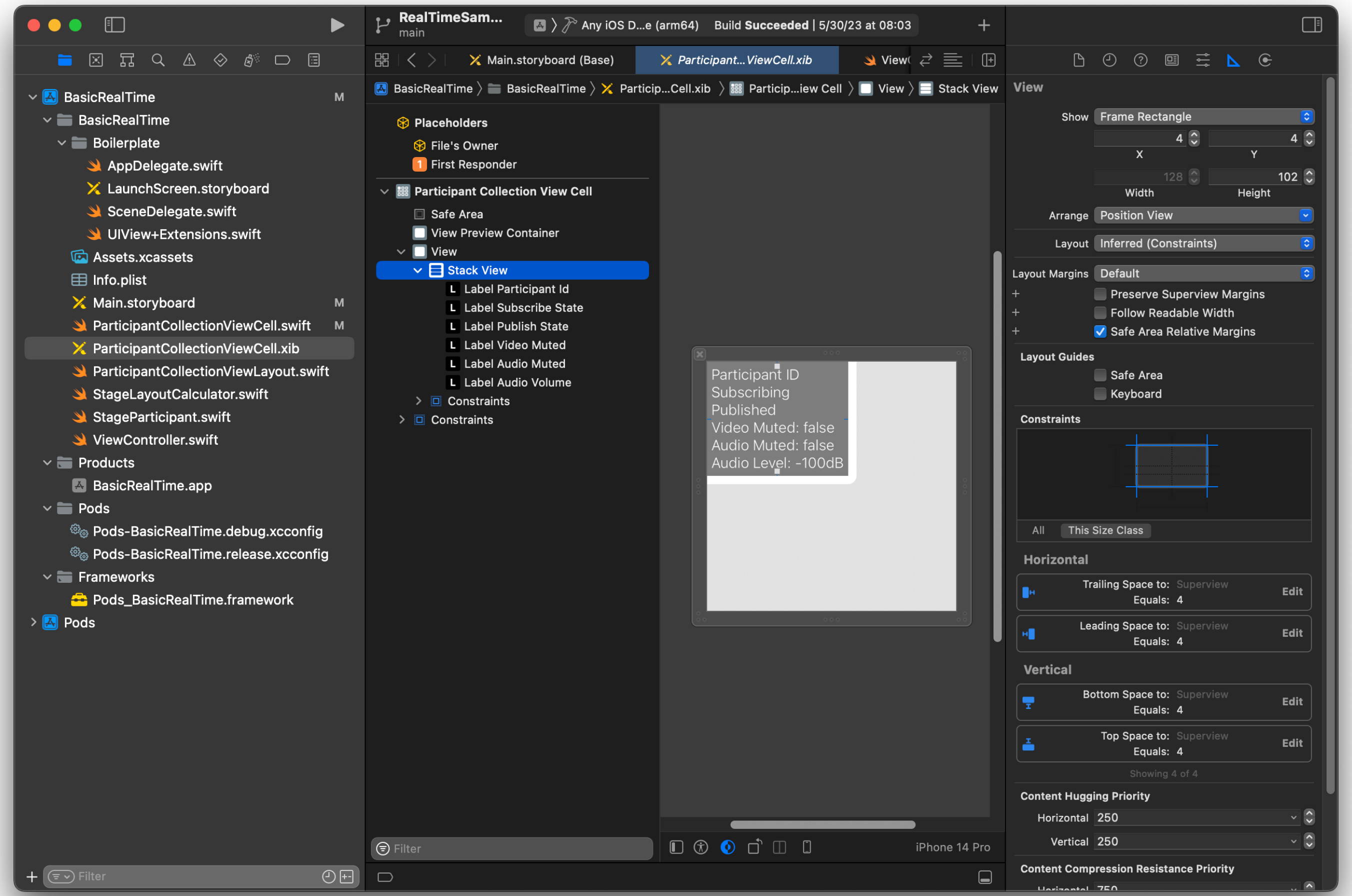
Horizontal
- Trailing Space to: Stack View Equals: 4 — Edit
- Leading Space to: Stack View Equals: 4 — Edit
- Leading Space to: Participant Colle... — Edit

Vertical
- Bottom Space to: Stack View Equals: 4 — Edit
- Top Space to: Stack View Equals: 4 — Edit
- Top Space to: Participant Colle... — Edit

Showing 6 of 6

Content Hugging Priority
Horizontal: 250
Vertical: 250

The third view is **Stack View**. Set **Trailing**, **Leading**, **Top**, and **Bottom** to **Superview** and change the value to 4.



```
BasicRealTime                                M
  BasicRealTime
    Boilerplate
      AppDelegate.swift
      LaunchScreen.storyboard
      SceneDelegate.swift
      UIView+Extensions.swift
    Assets.xcassets
    Info.plist
    Main.storyboard                          M
    ParticipantCollectionViewCell.swift      M
    ParticipantCollectionViewCell.xib
    ParticipantCollectionViewLayout.swift
    StageLayoutCalculator.swift
    StageParticipant.swift
    ViewController.swift
  Products
    BasicRealTime.app
  Pods
    Pods-BasicRealTime.debug.xcconfig
    Pods-BasicRealTime.release.xcconfig
  Frameworks
    Pods_BasicRealTime.framework
Pods
```

```
Placeholders
  File's Owner
  First Responder
Participant Collection View Cell
  Safe Area
  View Preview Container
  View
    Stack View
      Label Participant Id
      Label Subscribe State
      Label Publish State
      Label Video Muted
      Label Audio Muted
      Label Audio Volume
    Constraints
  Constraints
```

```
Participant ID
Subscribing
Published
Video Muted: false
Audio Muted: false
Audio Level: -100dB
```

```
View
Show            Frame Rectangle
X 4   Y 4   Width 128   Height 102
Arrange         Position View
Layout          Inferred (Constraints)
Layout Margins  Default
  Preserve Superview Margins
  Follow Readable Width
  ✓ Safe Area Relative Margins
Layout Guides
  Safe Area
  Keyboard
Constraints
Horizontal
  Trailing Space to: Superview   Equals: 4   Edit
  Leading Space to: Superview    Equals: 4   Edit
Vertical
  Bottom Space to: Superview     Equals: 4   Edit
  Top Space to: Superview        Equals: 4   Edit
Showing 4 of 4
Content Hugging Priority
  Horizontal 250
  Vertical 250
Content Compression Resistance Priority
```

Permissions and Idle Timer

Going back to our `ViewController`, we will disable the system idle timer to prevent the device from going to sleep while our application is being used:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    // Prevent the screen from turning off during a call.
    UIApplication.shared.isIdleTimerDisabled = true
}

override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
    UIApplication.shared.isIdleTimerDisabled = false
}
```

Next we request camera and microphone permissions from the system:

```
private func checkPermissions() {
    checkOrGetPermission(for: .video) { [weak self] granted in
        guard granted else {
            print("Video permission denied")
            return
        }
        self?.checkOrGetPermission(for: .audio) { [weak self] granted in
            guard granted else {
                print("Audio permission denied")
                return
            }
            self?.setupLocalUser() // we will cover this later
        }
    }
}

private func checkOrGetPermission(for mediaType: AVMediaType, _ result: @escaping
 (Bool) -> Void) {
    func mainThreadResult(_ success: Bool) {
        DispatchQueue.main.async {
            result(success)
        }
    }
    switch AVCaptureDevice.authorizationStatus(for: mediaType) {
    case .authorized: mainThreadResult(true)
    case .notDetermined:
        AVCaptureDevice.requestAccess(for: mediaType) { granted in
            mainThreadResult(granted)
        }
    case .denied, .restricted: mainThreadResult(false)
    @unknown default: mainThreadResult(false)
    }
}
```

App State

We need to configure our `collectionViewParticipants` with the layout file that we created earlier:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // We render everything to exactly the frame, so don't allow scrolling.
    collectionViewParticipants.isScrollEnabled = false
    collectionViewParticipants.register(UINib(nibName: "ParticipantCollectionViewCell",
 bundle: .main), forCellWithReuseIdentifier: "ParticipantCollectionViewCell")
}
```

To represent each participant, we create a simple struct called `StageParticipant`. This can be included in the `ViewController.swift` file, or a new file can be created.

```
import Foundation
import AmazonIVSBroadcast

struct StageParticipant {
    let isLocal: Bool
    var participantId: String?
    var publishState: IVSParticipantPublishState = .notPublished
    var subscribeState: IVSParticipantSubscribeState = .notSubscribed
    var streams: [IVSStageStream] = []

    init(isLocal: Bool, participantId: String?) {
        self.isLocal = isLocal
        self.participantId = participantId
    }
}
```

To track those participants, we keep an array of them as a private property in our `ViewController`:

```
private var participants = [StageParticipant]()
```

This property will be used to power our `UICollectionViewDataSource` that was linked from the storyboard earlier:

```
extension ViewController: UICollectionViewDataSource {

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection
 section: Int) -> Int {
        return participants.count
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath:
 IndexPath) -> UICollectionViewCell {
        if let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
 "ParticipantCollectionViewCell", for: indexPath) as? ParticipantCollectionViewCell {
```

```
            cell.set(participant: participants[indexPath.row])
            return cell
        } else {
            fatalError("Couldn't load custom cell type
 'ParticipantCollectionViewCell'")
        }
    }

}
```

To see your own preview before joining a stage, we create a local participant immediately:

```
override func viewDidLoad() {
    /* existing UICollectionView code */
    participants.append(StageParticipant(isLocal: true, participantId: nil))
}
```

This results in a participant cell being rendered immediately once the app is running, representing the local participant.

Users want to be able to see themselves before joining a stage, so next we implement the `setupLocalUser()` method that gets called from the permissions-handling code earlier. We store the camera and microphone reference as `IVSLocalStageStream` objects.

```
private var streams = [IVSLocalStageStream]()
private let deviceDiscovery = IVSDeviceDiscovery()

private func setupLocalUser() {
    // Gather our camera and microphone once permissions have been granted
    let devices = deviceDiscovery.listLocalDevices()
    streams.removeAll()
    if let camera = devices.compactMap({ $0 as? IVSCamera }).first {
        streams.append(IVSLocalStageStream(device: camera))
        // Use a front camera if available.
        if let frontSource = camera.listAvailableInputSources().first(where:
 { $0.position == .front }) {
            camera.setPreferredInputSource(frontSource)
        }
    }
    if let mic = devices.compactMap({ $0 as? IVSMicrophone }).first {
        streams.append(IVSLocalStageStream(device: mic))
    }
    participants[0].streams = streams
    participantsChanged(index: 0, changeType: .updated)
}
```

Here we've found the device's camera and microphone through the SDK and stored them in our local `streams` object, then assigned the `streams` array of the first participant (the local participant that we created earlier) to our `streams`. Finally we call `participantsChanged` with an `index` of 0 and `changeType` of `updated`. That function is a helper function for updating our `UICollectionView` with nice animations. Here's what it looks like:

```
private func participantsChanged(index: Int, changeType: ChangeType) {
    switch changeType {
    case .joined:
        collectionViewParticipants?.insertItems(at: [IndexPath(item: index, section:
 0)])
    case .updated:
        // Instead of doing reloadItems, just grab the cell and update it ourselves. It
 saves a create/destroy of a cell
        // and more importantly fixes some UI flicker. We disable scrolling so the
 index path per cell
        // never changes.
        if let cell = collectionViewParticipants?.cellForItem(at: IndexPath(item:
 index, section: 0)) as? ParticipantCollectionViewCell {
            cell.set(participant: participants[index])
        }
    case .left:
        collectionViewParticipants?.deleteItems(at: [IndexPath(item: index, section:
 0)])
    }
}
```

Don't worry about `cell.set` yet; we'll get to that later, but that's where we will render the cell's contents based on the participant.

The `ChangeType` is a simple enum:

```
enum ChangeType {
    case joined, updated, left
}
```

Finally, we want to keep track of whether the stage is connected. We use a simple `bool` to track that, which will automatically update our UI when it is updated itself.

```
private var connectingOrConnected = false {
    didSet {
        buttonJoin.setTitle(connectingOrConnected ? "Leave" : "Join", for: .normal)
        buttonJoin.tintColor = connectingOrConnected ? .systemRed : .systemBlue
    }
}
```

Implement the Stage SDK

Three core concepts underlie real-time functionality: stage, strategy, and renderer. The design goal is minimizing the amount of client-side logic necessary to build a working product.

IVSStageStrategy

Our `IVSStageStrategy` implementation is simple:

```
extension ViewController: IVSStageStrategy {
    func stage(_ stage: IVSStage, streamsToPublishForParticipant participant:
 IVSParticipantInfo) -> [IVSLocalStageStream] {
        // Return the camera and microphone to be published.
        // This is only called if `shouldPublishParticipant` returns true.
        return streams
    }

    func stage(_ stage: IVSStage, shouldPublishParticipant participant:
 IVSParticipantInfo) -> Bool {
        // Our publish status is based directly on the UISwitch view
        return switchPublish.isOn
    }

    func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType {
        // Subscribe to both audio and video for all publishing participants.
        return .audioVideo
    }
}
```

To summarize, we only publish if the publish switch is in the “on” position, and if we publish we will publish the streams that we collected earlier. Finally, for this sample, we always subscribe to other participants, receiving both their audio and video.

IVSStageRenderer

The `IVSStageRenderer` implementation also is fairly simple, though given the number of functions it contains quite a bit more code. The general approach in this renderer is to update our `participants` array when the SDK notifies us of a change to a participant. There are certain scenarios where we handle local participants differently, because we have decided to manage them ourselves so they can see their camera preview before joining.

```
extension ViewController: IVSStageRenderer {

    func stage(_ stage: IVSStage, didChange connectionState: IVSStageConnectionState,
 withError error: Error?) {
        labelState.text = connectionState.text
        connectingOrConnected = connectionState != .disconnected
    }

    func stage(_ stage: IVSStage, participantDidJoin participant: IVSParticipantInfo) {
        if participant.isLocal {
            // If this is the local participant joining the Stage, update the first
 participant in our array because we
            // manually added that participant when setting up our preview
            participants[0].participantId = participant.participantId
            participantsChanged(index: 0, changeType: .updated)
        } else {
            // If they are not local, add them to the array as a newly joined
 participant.
            participants.append(StageParticipant(isLocal: false, participantId:
 participant.participantId))
            participantsChanged(index: (participants.count - 1), changeType: .joined)
        }
    }

    func stage(_ stage: IVSStage, participantDidLeave participant: IVSParticipantInfo)
 {
        if participant.isLocal {
            // If this is the local participant leaving the Stage, update the first
 participant in our array because
            // we want to keep the camera preview active
            participants[0].participantId = nil
            participantsChanged(index: 0, changeType: .updated)
        } else {
            // If they are not local, find their index and remove them from the array.
            if let index = participants.firstIndex(where: { $0.participantId ==
participant.participantId }) {
                participants.remove(at: index)
                participantsChanged(index: index, changeType: .left)
            }
        }
    }

    func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChange
publishState: IVSParticipantPublishState) {
        // Update the publishing state of this participant
        mutatingParticipant(participant.participantId) { data in
            data.publishState = publishState
        }
    }

    func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChange
subscribeState: IVSParticipantSubscribeState) {
        // Update the subscribe state of this participant
        mutatingParticipant(participant.participantId) { data in
            data.subscribeState = subscribeState
        }
    }

    func stage(_ stage: IVSStage, participant: IVSParticipantInfo,
didChangeMutedStreams streams: [IVSStageStream]) {
        // We don't want to take any action for the local participant because we track
those streams locally
        if participant.isLocal { return }
        // For remote participants, notify the UICollectionView that they have updated.
There is no need to modify
        // the `streams` property on the `StageParticipant` because it is the same
`IVSStageStream` instance. Just
        // query the `isMuted` property again.
        if let index = participants.firstIndex(where: { $0.participantId ==
participant.participantId }) {
            participantsChanged(index: index, changeType: .updated)
        }
    }
    func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didAdd streams:
 [IVSStageStream]) {
        // We don't want to take any action for the local participant because we track
 those streams locally
        if participant.isLocal { return }
        // For remote participants, add these new streams to that participant's streams
 array.
        mutatingParticipant(participant.participantId) { data in
            data.streams.append(contentsOf: streams)
        }
    }

    func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didRemove streams:
 [IVSStageStream]) {
        // We don't want to take any action for the local participant because we track
 those streams locally
        if participant.isLocal { return }
        // For remote participants, remove these streams from that participant's
 streams array.
        mutatingParticipant(participant.participantId) { data in
            let oldUrns = streams.map { $0.device.descriptor().urn }
            data.streams.removeAll(where: { stream in
                return oldUrns.contains(stream.device.descriptor().urn)
            })
        }
    }

    // A helper function to find a participant by its ID, mutate that participant, and
 then update the UICollectionView accordingly.
    private func mutatingParticipant(_ participantId: String?, modifier: (inout
 StageParticipant) -> Void) {
        guard let index = participants.firstIndex(where: { $0.participantId ==
 participantId }) else {
            fatalError("Something is out of sync, investigate if this was a sample app
 or SDK issue.")
        }

        var participant = participants[index]
        modifier(&participant)
        participants[index] = participant
        participantsChanged(index: index, changeType: .updated)
    }
}
```

This code uses an extension to convert the connection state into human-friendly text:

```
extension IVSStageConnectionState {
    var text: String {
        switch self {
        case .disconnected: return "Disconnected"
        case .connecting: return "Connecting"
        case .connected: return "Connected"
        @unknown default: fatalError()
        }
    }
}
```

Implementing a Custom UICollectionViewLayout

Laying out different numbers of participants can be complex. You want them to take up the entire parent view's frame but you don't want to handle each participant configuration independently. To make this easy, we'll walk through implementing a `UICollectionViewLayout`.

Create another new file, `ParticipantCollectionViewLayout.swift`, which should extend `UICollectionViewLayout`. This class will use another class called `StageLayoutCalculator`, which we'll cover soon. The class receives calculated frame values for each participant and then generates the necessary `UICollectionViewLayoutAttributes` objects.

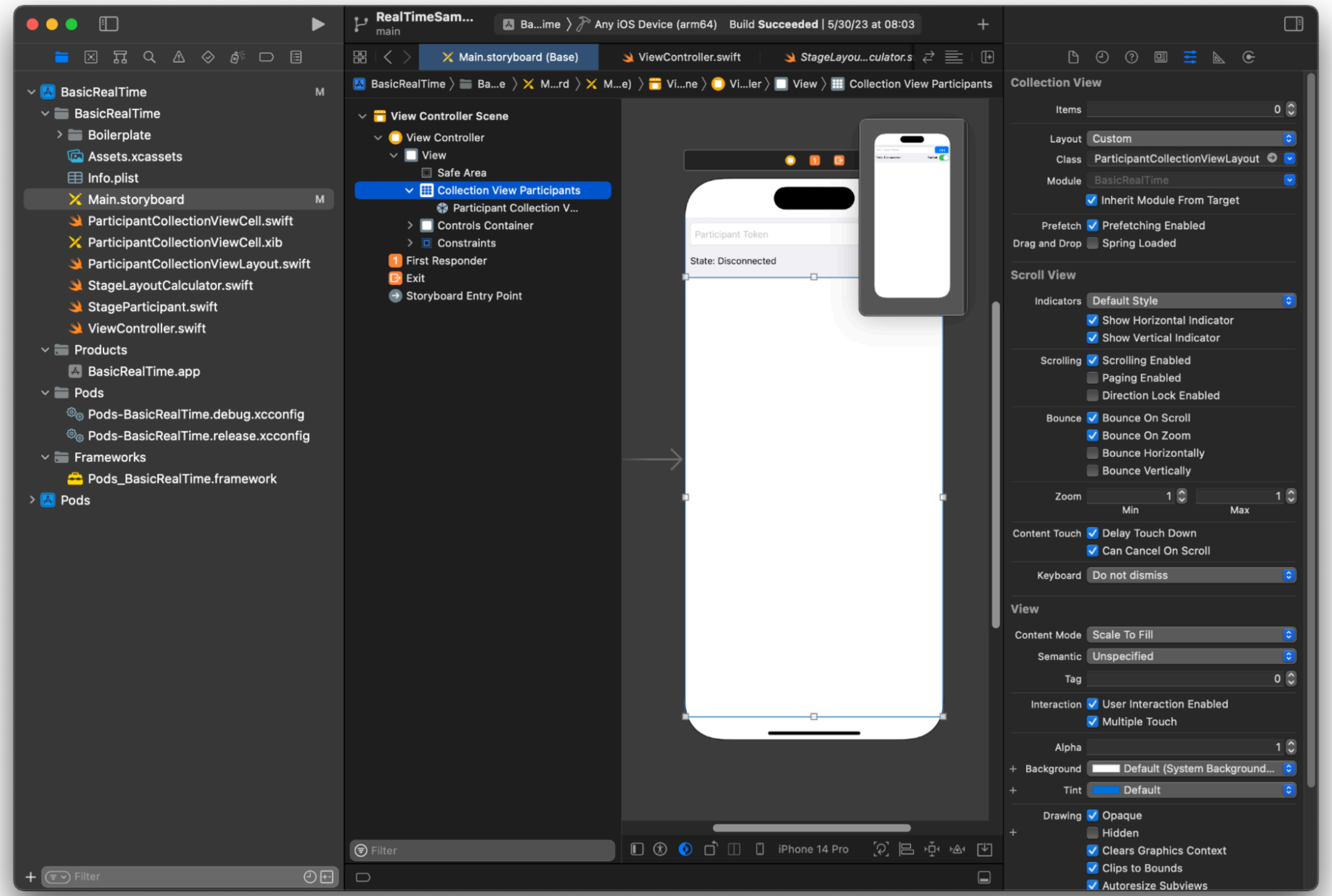
```
import Foundation
import UIKit

/**
 Code modified from https://developer.apple.com/documentation/uikit/views_and_controls/
collection_views/layouts/customizing_collection_view_layouts?language=objc
 */
class ParticipantCollectionViewLayout: UICollectionViewLayout {

    private let layoutCalculator = StageLayoutCalculator()

    private var contentBounds = CGRect.zero
    private var cachedAttributes = [UICollectionViewLayoutAttributes]()

    override func prepare() {
        super.prepare()

        guard let collectionView = collectionView else { return }
        cachedAttributes.removeAll()
        contentBounds = CGRect(origin: .zero, size: collectionView.bounds.size)

        layoutCalculator.calculateFrames(participantCount:
collectionView.numberOfItems(inSection: 0),
                                        width: collectionView.bounds.size.width,
                                        height: collectionView.bounds.size.height,
                                        padding: 4)
        .enumerated()
        .forEach { (index, frame) in
            let attributes = UICollectionViewLayoutAttributes(forCellWith:
IndexPath(item: index, section: 0))
            attributes.frame = frame
            cachedAttributes.append(attributes)
            contentBounds = contentBounds.union(frame)
        }
    }

    override var collectionViewContentSize: CGSize {
        return contentBounds.size
    }

    override func shouldInvalidateLayout(forBoundsChange newBounds: CGRect) -> Bool {
        guard let collectionView = collectionView else { return false }
        return !newBounds.size.equalTo(collectionView.bounds.size)
    }

    override func layoutAttributesForItem(at indexPath: IndexPath) ->
UICollectionViewLayoutAttributes? {
        return cachedAttributes[indexPath.item]
    }

    override func layoutAttributesForElements(in rect: CGRect) ->
[UICollectionViewLayoutAttributes]? {
        var attributesArray = [UICollectionViewLayoutAttributes]()

        // Find any cell that sits within the query rect.
        guard let lastIndex = cachedAttributes.indices.last, let firstMatchIndex =
binSearch(rect, start: 0, end: lastIndex) else {
            return attributesArray
        }
        // Starting from the match, loop up and down through the array until all the
 attributes
        // have been added within the query rect.
        for attributes in cachedAttributes[..<firstMatchIndex].reversed() {
            guard attributes.frame.maxY >= rect.minY else { break }
            attributesArray.append(attributes)
        }

        for attributes in cachedAttributes[firstMatchIndex...] {
            guard attributes.frame.minY <= rect.maxY else { break }
            attributesArray.append(attributes)
        }

        return attributesArray
    }

    // Perform a binary search on the cached attributes array.
    func binSearch(_ rect: CGRect, start: Int, end: Int) -> Int? {
        if end < start { return nil }

        let mid = (start + end) / 2
        let attr = cachedAttributes[mid]

        if attr.frame.intersects(rect) {
            return mid
        } else {
            if attr.frame.maxY < rect.minY {
                return binSearch(rect, start: (mid + 1), end: end)
            } else {
                return binSearch(rect, start: start, end: (mid - 1))
            }
        }
    }
}
```

More important is the `StageLayoutCalculator.swift` class. It is designed to calculate the frames for each participant based on the number of participants in a flow-based row/column layout. Each row is the same height as the others, but the columns can be different widths per row. See the code comment above the `layouts` variable for a description of how to customize this behavior.

```
import Foundation
import UIKit
```

```
class StageLayoutCalculator {

    /// This 2D array contains the description of how the grid of participants should
 be rendered
    /// The index of the 1st dimension is the number of participants needed to active
 that configuration
    /// Meaning if there is 1 participant, index 0 will be used. If there are 5
 participants, index 4 will be used.
    ///
    /// The 2nd dimension is a description of the layout. The length of the array is
 the number of rows that
    /// will exist, and then each number within that array is the number of columns in
 each row.
    ///
    /// See the code comments next to each index for concrete examples.
    ///
    /// This can be customized to fit any layout configuration needed.
    private let layouts: [[Int]] = [
        // 1 participant
        [ 1 ], // 1 row, full width
        // 2 participants
        [ 1, 1 ], // 2 rows, all columns are full width
        // 3 participants
        [ 1, 2 ], // 2 rows, first row's column is full width then 2nd row's columns
 are 1/2 width
        // 4 participants
        [ 2, 2 ], // 2 rows, all columns are 1/2 width
        // 5 participants
        [ 1, 2, 2 ], // 3 rows, first row's column is full width, 2nd and 3rd row's
 columns are 1/2 width
        // 6 participants
        [ 2, 2, 2 ], // 3 rows, all column are 1/2 width
        // 7 participants
        [ 2, 2, 3 ], // 3 rows, 1st and 2nd row's columns are 1/2 width, 3rd row's
 columns are 1/3rd width
        // 8 participants
        [ 2, 3, 3 ],
        // 9 participants
        [ 3, 3, 3 ],
        // 10 participants
        [ 2, 3, 2, 3 ],
        // 11 participants
        [ 2, 3, 3, 3 ],
        // 12 participants
        [ 3, 3, 3, 3 ],
    ]

    // Given a frame (this could be for a UICollectionView, or a Broadcast Mixer's
canvas), calculate the frames for each
    // participant, with optional padding.
    func calculateFrames(participantCount: Int, width: CGFloat, height: CGFloat,
padding: CGFloat) -> [CGRect] {
        if participantCount > layouts.count {
            fatalError("Only \(layouts.count) participants are supported at this time")
        }
        if participantCount == 0 {
            return []
        }
        var currentIndex = 0
        var lastFrame: CGRect = .zero

        // If the height is less than the width, the rows and columns will be flipped.
        // Meaning for 6 participants, there will be 2 rows of 3 columns each.
        let isVertical = height > width

        let halfPadding = padding / 2.0

        let layout = layouts[participantCount - 1] // 1 participant is in index 0, so
`-1`.
        let rowHeight = (isVertical ? height : width) / CGFloat(layout.count)

        var frames = [CGRect]()
        for row in 0 ..< layout.count {
            // layout[row] is the number of columns in a layout
            let itemWidth = (isVertical ? width : height) / CGFloat(layout[row])
            let segmentFrame = CGRect(x: (isVertical ? 0 : lastFrame.maxX) +
halfPadding,
                                      y: (isVertical ? lastFrame.maxY : 0) +
halfPadding,
                                      width: (isVertical ? itemWidth : rowHeight) -
padding,
                                      height: (isVertical ? rowHeight : itemWidth) -
padding)

            for column in 0 ..< layout[row] {
                var frame = segmentFrame
                if isVertical {
                    frame.origin.x = (itemWidth * CGFloat(column)) + halfPadding
                } else {
                    frame.origin.y = (itemWidth * CGFloat(column)) + halfPadding
                }
                frames.append(frame)
                currentIndex += 1
            }

            lastFrame = segmentFrame
            lastFrame.origin.x += halfPadding
            lastFrame.origin.y += halfPadding
        }
        return frames
    }

}
```

Back in `Main.storyboard`, be sure to set the layout class for the `UICollectionView` to the class we just created:


RealTimeSam...
main
Ba...ime 〉 Any iOS Device (arm64) Build Succeeded | 5/30/23 at 08:03
Main.storyboard (Base)
ViewController.swift
StageLayou...culator.s
BasicRealTime 〉 Ba...e 〉 M...rd 〉 M...e) 〉 Vi...ne 〉 Vi...ler 〉 View 〉 Collection View Participants
BasicRealTime
BasicRealTime
Boilerplate
Assets.xcassets
Info.plist
Main.storyboard
ParticipantCollectionViewCell.swift
ParticipantCollectionViewCell.xib
ParticipantCollectionViewLayout.swift
StageLayoutCalculator.swift
StageParticipant.swift
ViewController.swift
Products
BasicRealTime.app
Pods
Pods-BasicRealTime.debug.xcconfig
Pods-BasicRealTime.release.xcconfig
Frameworks
Pods_BasicRealTime.framework
Pods
View Controller Scene
View Controller
View
Safe Area
Collection View Participants
Participant Collection V...
Controls Container
Constraints
First Responder
Exit
Storyboard Entry Point
Participant Token
State: Disconnected
iPhone 14 Pro
Filter

Collection View
Items 0
Layout Custom
Class ParticipantCollectionViewLayout
Module BasicRealTime
✓ Inherit Module From Target
Prefetch ✓ Prefetching Enabled
Drag and Drop Spring Loaded

Scroll View
Indicators Default Style
✓ Show Horizontal Indicator
✓ Show Vertical Indicator
Scrolling ✓ Scrolling Enabled
Paging Enabled
Direction Lock Enabled
Bounce ✓ Bounce On Scroll
✓ Bounce On Zoom
Bounce Horizontally
Bounce Vertically
Zoom 1 Min 1 Max
Content Touch ✓ Delay Touch Down
✓ Can Cancel On Scroll
Keyboard Do not dismiss

View
Content Mode Scale To Fill
Semantic Unspecified
Tag 0
Interaction ✓ User Interaction Enabled
✓ Multiple Touch
Alpha 1
Background Default (System Background...
Tint Default
Drawing ✓ Opaque
Hidden
✓ Clears Graphics Context
✓ Clips to Bounds
✓ Autoresize Subviews

Hooking Up UI Actions

We are getting close, there are a few IBActions that we need to create.

First we'll handle the join button. It responds differently based on the value of connectingOrConnected. When it is already connected, it just leaves the stage. If it is disconnected, it reads the text from the token UITextField and creates a new IVSStage with that text. Then we add our ViewController as the strategy, errorDelegate, and renderer for the IVSStage, and finally we join the stage asynchronously.

```
@IBAction private func joinTapped(_ sender: UIButton) {
    if connectingOrConnected {
        // If we're already connected to a Stage, leave it.
        stage?.leave()
    } else {
        guard let token = textFieldToken.text else {
            print("No token")
            return
        }
        // Hide the keyboard after tapping Join
        textFieldToken.resignFirstResponder()
        do {
            // Destroy the old Stage first before creating a new one.
            self.stage = nil
            let stage = try IVSStage(token: token, strategy: self)
            stage.errorDelegate = self
            stage.addRenderer(self)
            try stage.join()
            self.stage = stage
        } catch {
            print("Failed to join stage - \(error)")
        }
    }
}
```

The other UI action we need to hook up is the publish switch:

```
@IBAction private func publishToggled(_ sender: UISwitch) {
    // Because the strategy returns the value of `switchPublish.isOn`, just call
 `refreshStrategy`.
    stage?.refreshStrategy()
}
```

Rendering the Participants

Finally, we need to render the data we receive from the SDK onto the participant cell that we created earlier. We already have the `UICollectionView` logic finished, so we just need to implement the `set` API in `ParticipantCollectionViewCell.swift`.

We'll start by adding the `empty` function and then walk through it step by step:

```
func set(participant: StageParticipant) {

}
```

First we handle the easy state, the participant ID, publish state, and subscribe state. For these, we just update our `UILabels` directly:

```
labelParticipantId.text = participant.isLocal ? "You (\(participant.participantId ??
 "Disconnected"))" : participant.participantId
labelPublishState.text = participant.publishState.text
labelSubscribeState.text = participant.subscribeState.text
```

The text properties of the publish and subscribe enums come from local extensions:

```
extension IVSParticipantPublishState {
    var text: String {
        switch self {
        case .notPublished: return "Not Published"
        case .attemptingPublish: return "Attempting to Publish"
        case .published: return "Published"
        @unknown default: fatalError()
        }
    }
}

extension IVSParticipantSubscribeState {
    var text: String {
        switch self {
        case .notSubscribed: return "Not Subscribed"
        case .attemptingSubscribe: return "Attempting to Subscribe"
        case .subscribed: return "Subscribed"
        @unknown default: fatalError()
        }
    }
}
```

Next we update the audio and video muted states. To get the muted states we need to find the `IVSImageDevice` and `IVSAudioDevice` from the `streams` array. To optimize performance, we will remember the last devices attached.

```
// This belongs outside `set(participant:)`
private var registeredStreams: Set<IVSStageStream> = []
private var imageDevice: IVSImageDevice? {
    return registeredStreams.lazy.compactMap { $0.device as? IVSImageDevice }.first
}
private var audioDevice: IVSAudioDevice? {
    return registeredStreams.lazy.compactMap { $0.device as? IVSAudioDevice }.first
}
```

```
// This belongs inside `set(participant:)`
let existingAudioStream = registeredStreams.first { $0.device is IVSAudioDevice }
let existingImageStream = registeredStreams.first { $0.device is IVSImageDevice }

registeredStreams = Set(participant.streams)

let newAudioStream = participant.streams.first { $0.device is IVSAudioDevice }
let newImageStream = participant.streams.first { $0.device is IVSImageDevice }

// `isMuted != false` covers the stream not existing, as well as being muted.
labelVideoMuted.text = "Video Muted: \(newImageStream?.isMuted != false)"
labelAudioMuted.text = "Audio Muted: \(newAudioStream?.isMuted != false)"
```

Finally we want to render a preview for the `imageDevice` and display audio stats from the `audioDevice`:

```
if existingImageStream !== newImageStream {
    // The image stream has changed
    updatePreview() // We'll cover this next
}

if existingAudioStream !== newAudioStream {
    (existingAudioStream?.device as? IVSAudioDevice)?.setStatsCallback(nil)
    audioDevice?.setStatsCallback( { [weak self] stats in
        self?.labelAudioVolume.text = String(format: "Audio Level: %.0f dB", stats.rms)
    })
    // When the audio stream changes, it will take some time to receive new stats.
 Reset the value temporarily.
    self.labelAudioVolume.text = "Audio Level: -100 dB"
}
```

The last function we need to create is `updatePreview()`, which adds a preview of the participant to our view:

```
private func updatePreview() {
    // Remove any old previews from the preview container
    viewPreviewContainer.subviews.forEach { $0.removeFromSuperview() }
    if let imageDevice = self.imageDevice {
        if let preview = try? imageDevice.previewView(with: .fit) {
            viewPreviewContainer.addSubviewMatchFrame(preview)
        }
    }
}
```

The above uses a helper function on `UIView` to make embedding subviews easier:

```
extension UIView {
    func addSubviewMatchFrame(_ view: UIView) {
        view.translatesAutoresizingMaskIntoConstraints = false
        self.addSubview(view)
        NSLayoutConstraint.activate([
            view.topAnchor.constraint(equalTo: self.topAnchor, constant: 0),
            view.bottomAnchor.constraint(equalTo: self.bottomAnchor, constant: 0),
            view.leadingAnchor.constraint(equalTo: self.leadingAnchor, constant: 0),
            view.trailingAnchor.constraint(equalTo: self.trailingAnchor, constant: 0),
        ])
    }
}
```

Monitoring Amazon IVS Real-Time Streaming

What is a Stage Session?

A stage *session* begins when the first participant joins a stage and ends a few minutes after the last participant stops publishing to the stage. Stage sessions help with debugging long-lived stages by separating out events and participants into short-lived sessions.

View Stage Sessions and Participants

Console Instructions

1. Open the Amazon IVS console.

   (You also can access the Amazon IVS console through the AWS Management Console.)
2. On the navigation pane, choose **Stages**. (If the nav pane is collapsed, first open it by choosing the hamburger icon.)
3. Choose the stage to go to its details page.
4. Scroll down the page until you see the **Stage sessions** section, then select a stage session to view its details page.
5. To view participants in the session, scroll down until you see the **Participants** section, then select a participant to view its details page, including charts for Amazon CloudWatch metrics.

View Events for a Participant

Events are sent when a participant's status in a stage changes, such as joining a stage or encountering an error trying to publish to a stage. Not all errors cause events; e.g., client-side network errors and token-signature errors are not sent as events. To handle these errors in your client application, use the IVS broadcast SDKs.

Console Instructions

1. Navigate to the participant details page as instructed above.

2. Scroll down until you see the **Events** section. This displays an ordered list of participant events. See Using Amazon EventBridge with Amazon IVS for details on events that are emitted for participants.

CLI Instructions

Accessing stage-session events with the AWS CLI is an advanced option and requires that you first download and configure the CLI on your machine. For details, see the AWS Command Line Interface User Guide.

1. List stage sessions to find a stage session:

```
aws ivs-realtime list-stage-sessions --stage-arn <arn>
```

2. List participants for a stage session to find a participant:

```
aws ivs-realtime list-participants --stage-arn <arn> -session-id <sessionId>
```

3. List events for a stage session and participant:

```
aws ivs-realtime list-participant-events --stage-arn <arn> --session-id <sessionId>
 --participant-id <participantId>
```

Here is a sample response to the `list-participant-events` call:

```
{
    "events": [
        {
            "eventTime": "2023-04-04T22:48:41+00:00",
            "name": "JOINED",
            "participantId": "AdRezBl021t0"
        },
        {
            "eventTime": "2023-04-04T22:48:41+00:00",
            "name": "SUBSCRIBE_STARTED",
            "participantId": "AdRezBl021t0",
            "remoteParticipantId": "Ou5b5n5XLMdC"
        },
        {
            "eventTime": "2023-04-04T22:49:45+00:00",
            "name": "SUBSCRIBE_STOPPED",
            "participantId": "AdRezBl021t0",
            "remoteParticipantId": "Ou5b5n5XLMdC"
        },
        {
            "eventTime": "2023-04-04T22:49:45+00:00",
            "name": "LEFT",
            "participantId": "AdRezBl021t0"
        }
    ]
}
```

Access CloudWatch Metrics

For CloudWatch metrics to be available, the following IVS Broadcast SDK versions are required: Web 1.5.0 or later, Android 1.12.0 or later, or iOS 1.12.0 or later.

CloudWatch Console Instructions

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the side navigation, expand the **Metrics** dropdown, then select **All metrics**.
3. On the **Browse** tab, using the unlabeled dropdown at the left, select your “home” region, where your channel(s) was(were) created. For more on regions, see Global Solution, Regional Control. For a list of supported regions, see the Amazon IVS page in the *AWS General Reference*.
4. At the bottom of the **Browse** tab, select the **IVSRealTime** namespace.
5. Do one of the following:
   a. In the search bar, enter your resource ID (part of the ARN, `arn::ivs:stage/<resource id>`).

      Then select **IVSRealTime > Stage Metrics**.
   b. If **IVSRealTime** appears as a selectable service under **AWS Namespaces**, select it. It will be listed if you use Amazon IVS Real-Time Streaming and it is sending metrics to Amazon CloudWatch. (If **IVSRealTime** is not listed, you do not have any Amazon IVS metrics.)

      Then choose a *dimension* grouping as desired; available dimensions are listed in CloudWatch Metrics below.
6. Choose metrics to add to the graph. Available metrics are listed in CloudWatch Metrics below.

You also can access your stream session's CloudWatch chart from the stream session's details page, by selecting the **View in CloudWatch** button.

CLI Instructions

You also can access the metrics using the AWS CLI. This requires that you first download and configure the CLI on your machine. For details, see the AWS Command Line Interface User Guide.

Then, to access Amazon IVS real-time streaming metrics using the AWS CLI:

- At a command prompt, run:

```
aws cloudwatch list-metrics --namespace AWS/IVSRealTime
```

For more information, see Using Amazon CloudWatch Metrics in the *Amazon CloudWatch User Guide*.

CloudWatch Metrics: IVS Real-Time Streaming

Amazon IVS provides the following metrics in the **AWS/IVSRealTime** namespace.

For CloudWatch metrics to be available, Web Broadcast SDK 1.5.2 or later must be used.

The dimension can have the following valid values:

- The `Stage` dimension is a resource ID (part of the ARN, `arn:::stage/<resource id>`).
- The `Participant` dimension is a `participantID`.
- The `SimulcastLayer` is "hi", "mid", "low", or "no-rid" for a `MediaType` of "video" or "disabled" for a `MediaType` of "audio." This value also can be empty.
- The `MediaType` dimension is "video" or "audio" (string).

Metric	Dimension	Description
`DownloadPacketLoss`	`Stage`	Each sample represents the percentage of packets that were lost by a given subscriber while downloading from the IVS server.

Metric	Dimension	Description
		Unit: Percent Valid statistics: Average, Maximum, Minimum — Average number, largest number, or smallest number (respecti vely) of packet loss over the configured interval
`DownloadP acketLoss`	`Stage,Par ticipant`	Filters `DownloadPacketLoss` by participant, for subscribers who are also publishers. Samples represent the percentage of packets that were lost by the subscriber while downloading from the IVS server. Samples are emitted only when the participant is also a publisher. Unit: Percent Valid statistics: Average, Maximum, Minimum — Average number, largest number, or smallest number (respecti vely) of dropped frames over the configured interval
`DroppedFr ames`	`Stage`	Each sample represents the percentage of frames that were dropped by a given subscriber. Unit: Percent Valid statistics: Average, Maximum, Minimum — Average number, largest number, or smallest number (respecti vely) of dropped frames over the configured interval

Metric	Dimension	Description
DroppedFrames	Stage,Participant	Filters DroppedFrames by participant, for subscribers who are also publishers. Samples represent the percentage of frames that were dropped between the subscribing participant and all publishers in the stage. Samples are emitted only when the participant is also a publisher. Unit: Percent Valid statistics: Average, Maximum, Minimum — Average number, largest number, or smallest number (respectively) of dropped frames over the configured interval
PublishBitrate	Stage	Samples emitted represent the total rate at which a given publisher is sending both video and audio data (summed across all simulcast layers). Unit: Bits/second Valid statistics: Average, Maximum, Minimum — Average number, largest number, or smallest number (respectively) of bitrate over the configured interval
PublishBitrate	Stage, Participant, SimulcastLayer, MediaType	Filters PublishBitrate by participant, simulcast layer, and media type. The simulcast layer ID is set by the broadcast SDK. When simulcast is disabled, this layer ID will be set to "disabled". The media type is either video or audio. Unit: Bits/second Valid statistics: Average, Maximum, Minimum — Average number, largest number, or smallest number (respectively) of bitrate over the configured interval

Metric	Dimension	Description
`Publishers`	`Stage`	Number of participants publishing to the stage. Unit: Count Valid statistics: Average, Maximum, Minimum
`PublishResolution`	`Stage, Participant, Simulcast Layer, MediaType`	Number of pixels across the smaller of the width or height of the frame. For example, for a landscape frame of size 1920x1080, the PublishResolution is 1080. For a portrait frame of size 720x1280, the PublishResolution is 720. Unit: Count Valid statistics: Average, Maximum, Minimum
`Subscribe Bitrate`	`Stage`	Samples emitted represent the total rate at which a given subscriber is receiving both video and audio data. Unit: Bits/second Valid statistics: Average, Maximum, Minimum — Average number, largest number, or smallest number (respectively) of bitrate over the configured interval
`Subscribe Bitrate`	`Stage,Participant, MediaType`	Filters `SubscribeBitrate` by participant, for subscribers who are also publishers. Samples represent the bitrate at which a given subscriber is receiving the given `MediaType`. Samples are only emitted while the subscribing participant is publishing. Unit: Bits/second Valid statistics: Average, Maximum, Minimum — Average number, largest number, or smallest number (respectively) of bitrate over the configured interval

Metric	Dimension	Description
`Subscribers`	`Stage`	Number of participants subscribed to the stage. Note that participants that are actively publishing and subscribing are counted as both publishers and subscribers. Unit: Count Valid statistics: Average, Maximum, Minimum

IVS Broadcast SDK (Real-Time Streaming)

The Amazon Interactive Video Services (IVS) Real-Time Streaming broadcast SDK is for developers who are building applications with Amazon IVS. This SDK is designed to leverage the Amazon IVS architecture and will see continual improvement and new features, alongside Amazon IVS. As a native broadcast SDK, it is designed to minimize the performance impact on your application and on the devices with which your users access your application.

Note that the broadcast SDK is used for both sending and receiving video; i.e., you use the same SDK for hosts and viewers. No separate player SDK needed.

Your application can leverage the key features of the Amazon IVS broadcast SDK:

- **High quality streaming** — The broadcast SDK supports high quality streaming. Capture video from your camera and encode it at up to 720p.
- **Automatic Bitrate Adjustments** — Smartphone users are mobile, so their network conditions can change throughout the course of a broadcast. The Amazon IVS broadcast SDK automatically adjusts the video bitrate to accommodate changing network conditions.
- **Portrait and Landscape Support** — No matter how your users hold their devices, the image appears right-side up and properly scaled. The broadcast SDK supports both portrait and landscape canvas sizes. It automatically manages the aspect ratio when the users rotate their device away from the configured orientation.
- **Secure Streaming** — Your user's broadcasts are encrypted using TLS, so they can keep their streams secure.
- **External Audio Devices** — The Amazon IVS broadcast SDK supports audio jack, USB, and Bluetooth SCO external microphones.

Platform Requirements

Native Platforms

Platform	Supported Versions
Android	9.0 and later -- note customers can build with version 5.0 but will not be able to use real-time streaming functionality.

Platform	Supported Versions
iOS	14 and later

IVS supports a minimum of 4 major iOS versions and 6 major Android versions. Our current version support may extend beyond these minimums. Customers will be notified via SDK release notes at least 3 months in advance of a major version no longer being supported.

Desktop Browsers

Browser	Supported Platforms	Supported Versions
Chrome	Windows, macOS	Two major versions (current and most recent prior version)
Firefox	Windows, macOS	Two major versions (current and most recent prior version)
Edge	Windows 8.1 and later	Two major versions (current and most recent prior version) Excludes Edge Legacy
Safari	macOS	Two major versions (current and most recent prior version)

Mobile Browsers (iOS and Android)

Browser	Supported Platforms	Supported Versions
Chrome	iOS, Android	Two major versions (current and most recent prior version)

Browser	Supported Platforms	Supported Versions
Firefox	Android	Two major versions (current and most recent prior version)
Safari	iOS	Two major versions (current and most recent prior version)

Known Limitations

- On all mobile devices, we do not recommend publishing/subscribing with four or more participants at the same time, due to issues with video artifacts and black screens. If you require more participants, configure audio-only publish and subscribe.
- We do not recommend compositing a stage and broadcasting it to a channel on Android Mobile Web, due to performance considerations and potential crashes. If broadcast functionality is required, integrate the IVS real-time streaming Android broadcast SDK.

Webviews

The Web broadcast SDK does not provide support for webviews or weblike environments (TVs, consoles, etc). For mobile implementations, see the Real-Time Streaming Broadcast SDK Guide for Android and for iOS.

Required Device Access

The broadcast SDK requires access to the device's cameras and microphones, both those built into the device and those connected through Bluetooth, USB, or audio jack.

Support

The broadcast SDK is continually improved. See Amazon IVS Release Notes for available versions and fixed issues. If appropriate, before contacting support, update your version of the broadcast SDK and see if that resolves your issue.

Versioning

The Amazon IVS broadcast SDKs use semantic versioning.

For this discussion, suppose:

- The latest release is 4.1.3.
- The latest release of the prior major version is 3.2.4.
- The latest release of version 1.x is 1.5.6.

Backward-compatible new features are added as minor releases of the latest version. In this case, the next set of new features will be added as version 4.2.0.

Backward-compatible, minor bug fixes are added as patch releases of the latest version. Here, the next set of minor bug fixes will be added as version 4.1.4.

Backward-compatible, major bug fixes are handled differently; these are added to several versions:

- Patch release of the latest version. Here, this is version 4.1.4.
- Patch release of the prior minor version. Here, this is version 3.2.5.
- Patch release of the latest version 1.x release. Here, this is version 1.5.7.

Major bug fixes are defined by the Amazon IVS product team. Typical examples are critical security updates and selected other fixes necessary for customers.

Note: In the examples above, released versions increment without skipping any numbers (e.g., from 4.1.3 to 4.1.4). In reality, one or more patch numbers may remain internal and not be released, so the released version could increment from 4.1.3 to, say, 4.1.6.

IVS Broadcast SDK: Web Guide (Real-Time Streaming)

The IVS real-time streaming Web broadcast SDK gives developers the tools to build interactive, real-time experiences on the web. This SDK is for developers who are building web applications with Amazon IVS.

The Web broadcast SDK enables participants to send and receive video. The SDK supports the following operations:

- Join a stage
- Publish media to other participants in the stage
- Subscribe to media from other participants in the stage
- Manage and monitor video and audio published to the stage
- Get WebRTC statistics for each peer connection
- All operations from the IVS low-latency streaming Web broadcast SDK

Latest version of Web broadcast SDK: 1.11.0 (Release Notes)

Reference documentation: For information on the most important methods available in the Amazon IVS Web Broadcast SDK, see https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference. Make sure the most current version of the SDK is selected.

Sample code: The samples below are a good place to get started quickly with the SDK:

- HTML and JavaScript
- React

Platform requirements: See Amazon IVS Broadcast SDK for a list of supported platforms

Getting Started

Imports

The building blocks for real-time are located in a different namespace than the root broadcasting modules.

Using a Script Tag

Using the same script imports, the classes and enums defined in the examples below can be found on the global object `IVSBroadcastClient`:

```
const { Stage, SubscribeType } = IVSBroadcastClient;
```

Using npm

The classes, enums, and types also can be imported from the package module:

```
import { Stage, SubscribeType, LocalStageStream } from 'amazon-ivs-web-broadcast'
```

Server-Side Rendering Support

The Web Broadcast SDK Stages library cannot be loaded in a server-side context, as it references browser primitives necessary to the functioning of the library when loaded. To work around this, load the library dynamically, as demonstrated in the Web Broadcast Demo using Next and React.

Request Permissions

Your app must request permission to access the user's camera and microphone, and it must be served using HTTPS. (This is not specific to Amazon IVS; it is required for any website that needs access to cameras and microphones.)

Here's an example function showing how you can request and capture permissions for both audio and video devices:

```
async function handlePermissions() {
   let permissions = {
       audio: false,
       video: false,
   };
   try {
       const stream = await navigator.mediaDevices.getUserMedia({ video: true, audio:
 true });
       for (const track of stream.getTracks()) {
           track.stop();
       }
       permissions = { video: true, audio: true };
   } catch (err) {
       permissions = { video: false, audio: false };
       console.error(err.message);
   }
   // If we still don't have permissions after requesting them display the error
 message
   if (!permissions.video) {
       console.error('Failed to get video permissions.');
   } else if (!permissions.audio) {
       console.error('Failed to get audio permissions.');
   }
}
```

For additional information, see the Permissions API and MediaDevices.getUserMedia().

List Available Devices

To see what devices are available to capture, query the browser's MediaDevices.enumerateDevices() method:

```
const devices = await navigator.mediaDevices.enumerateDevices();
window.videoDevices = devices.filter((d) => d.kind === 'videoinput');
window.audioDevices = devices.filter((d) => d.kind === 'audioinput');
```

Retrieve a MediaStream from a Device

After acquiring the list of available devices, you can retrieve a stream from any number of devices. For example, you can use the `getUserMedia()` method to retrieve a stream from a camera.

If you'd like to specify which device to capture the stream from, you can explicitly set the `deviceId` in the `audio` or `video` section of the media constraints. Alternately, you can omit the `deviceId` and have users select their devices from the browser prompt.

You also can specify an ideal camera resolution using the `width` and `height` constraints. (Read more about these constraints here.) The SDK automatically applies width and height constraints that correspond to your maximum broadcast resolution; however, it's a good idea to also apply these yourself to ensure that the source aspect ratio is not changed after you add the source to the SDK.

For real-time streaming, ensure that media is constrained to 720p resolution. Specifically, your `getUserMedia` and `getDisplayMedia` constraint values for width and height must not exceed 921600 (1280*720) when multiplied together.

```
const videoConfiguration = {
  maxWidth: 1280,
  maxHeight: 720,
  maxFramerate: 30,
}

window.cameraStream = await navigator.mediaDevices.getUserMedia({
   video: {
       deviceId: window.videoDevices[0].deviceId,
       width: {
           ideal: videoConfiguration.maxWidth,
       },
       height: {
           ideal:videoConfiguration.maxHeight,
       },
   },
});
window.microphoneStream = await navigator.mediaDevices.getUserMedia({
   audio: { deviceId: window.audioDevices[0].deviceId },
});
```

Publishing and Subscribing

Concepts

Three core concepts underlie real-time functionality: stage, strategy, and events. The design goal is minimizing the amount of client-side logic necessary to build a working product.

Stage

The `Stage` class is the main point of interaction between the host application and the SDK. It represents the stage itself and is used to join and leave the stage. Creating and joining a stage requires a valid, unexpired token string from the control plane (represented as `token`). Joining and leaving a stage are simple:

```
const stage = new Stage(token, strategy)

try {
    await stage.join();
} catch (error) {
    // handle join exception
}

stage.leave();
```

Strategy

The `StageStrategy` interface provides a way for the host application to communicate the desired state of the stage to the SDK. Three functions need to be implemented: `shouldSubscribeToParticipant`, `shouldPublishParticipant`, and `stageStreamsToPublish`. All are discussed below.

To use a defined strategy, pass it to the `Stage` constructor. The following is a complete example of an application using a strategy to publish a participant's webcam to the stage and subscribe to all participants. Each required strategy function's purpose is explained in detail in the subsequent sections.

```
const devices = await navigator.mediaDevices.getUserMedia({
   audio: true,
   video: {
        width: { max: 1280 },
        height: { max: 720 },
    }
});
const myAudioTrack = new LocalStageStream(devices.getAudioTracks()[0]);
const myVideoTrack = new LocalStageStream(devices.getVideoTracks()[0]);

// Define the stage strategy, implementing required functions
const strategy = {
   audioTrack: myAudioTrack,
   videoTrack: myVideoTrack,

   // optional
   updateTracks(newAudioTrack, newVideoTrack) {
      this.audioTrack = newAudioTrack;
      this.videoTrack = newVideoTrack;
   },

   // required
   stageStreamsToPublish() {
      return [this.audioTrack, this.videoTrack];
   },

   // required
   shouldPublishParticipant(participant) {
      return true;
   },

   // required
   shouldSubscribeToParticipant(participant) {
      return SubscribeType.AUDIO_VIDEO;
   }
};

// Initialize the stage and start publishing
```

```
const stage = new Stage(token, strategy);
await stage.join();


// To update later (e.g. in an onClick event handler)
strategy.updateTracks(myNewAudioTrack, myNewVideoTrack);
stage.refreshStrategy();
```

Subscribing to Participants

```
shouldSubscribeToParticipant(participant: StageParticipantInfo): SubscribeType
```

When a remote participant joins the stage, the SDK queries the host application about the desired subscription state for that participant. The options are `NONE`, `AUDIO_ONLY`, and `AUDIO_VIDEO`. When returning a value for this function, the host application does not need to worry about the publish state, current subscription state, or stage connection state. If `AUDIO_VIDEO` is returned, the SDK waits until the remote participant is publishing before it subscribes, and it updates the host application by emitting events throughout the process.

Here is a sample implementation:

```
const strategy = {

   shouldSubscribeToParticipant: (participant) => {
      return SubscribeType.AUDIO_VIDEO;
   }

   // ... other strategy functions
}
```

This is the complete implementation of this function for a host application that always wants all participants to see each other; e.g., a video chat application.

More advanced implementations also are possible. Use the `userInfo` property on `ParticipantInfo` to selectively subscribe to participants based on server-provided attributes:

```
const strategy = {

   shouldSubscribeToParticipant(participant) {
      switch (participant.info.userInfo) {
         case 'moderator':
            return SubscribeType.NONE;
         case 'guest':
            return SubscribeType.AUDIO_VIDEO;
         default:
            return SubscribeType.NONE;
      }
   }
   // . . . other strategies properties
}
```

This can be used to create a stage where moderators can monitor all guests without being seen or heard themselves. The host application could use additional business logic to let moderators see each other but remain invisible to guests.

Publishing

```
shouldPublishParticipant(participant: StageParticipantInfo): boolean
```

Once connected to the stage, the SDK queries the host application to see if a particular participant should publish. This is invoked only on local participants that have permission to publish based on the provided token.

Here is a sample implementation:

```
const strategy = {

   shouldPublishParticipant: (participant) => {
      return true;
   }

   // . . . other strategies properties
}
```

This is for a standard video chat application where users always want to publish. They can mute and unmute their audio and video, to instantly be hidden or seen/heard. (They also can use publish/unpublish, but that is much slower. Mute/unmute is preferable for use cases where changing visibility often is desirable.)

Choosing Streams to Publish

```
stageStreamsToPublish(): LocalStageStream[];
```

When publishing, this is used to determine what audio and video streams should be published. This is covered in more detail later in [Publish a Media Stream](#).

Updating the Strategy

The strategy is intended to be dynamic: the values returned from any of the above functions can be changed at any time. For example, if the host application does not want to publish until the end user taps a button, you could return a variable from `shouldPublishParticipant` (something like `hasUserTappedPublishButton`). When that variable changes based on an interaction by the end user, call `stage.refreshStrategy()` to signal to the SDK that it should query the strategy for the latest values, applying only things that have changed. If the SDK observes that the `shouldPublishParticipant` value has changed, it starts the publish process. If the SDK queries and all functions return the same value as before, the `refreshStrategy` call does not modify the stage.

If the return value of `shouldSubscribeToParticipant` changes from `AUDIO_VIDEO` to `AUDIO_ONLY`, the video stream is removed for all participants with changed returned values, if a video stream existed previously.

Generally, the stage uses the strategy to most efficiently apply the difference between the previous and current strategies, without the host application needing to worry about all the state required to manage it properly. Because of this, think of calling `stage.refreshStrategy()` as a cheap operation, because it does nothing unless the strategy changes.

Events

A `Stage` instance is an event emitter. Using `stage.on()`, the state of the stage is communicated to the host application. Updates to the host application's UI usually can be supported entirely by the events. The events are as follows:

```
stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {})
stage.on(StageEvents.STAGE_PARTICIPANT_JOINED, (participant) => {})
stage.on(StageEvents.STAGE_PARTICIPANT_LEFT, (participant) => {})
stage.on(StageEvents.STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED, (participant, state) =>
 {})
stage.on(StageEvents.STAGE_PARTICIPANT_SUBSCRIBE_STATE_CHANGED, (participant, state) =>
 {})
stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED, (participant, streams) => {})
stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_REMOVED, (participant, streams) => {})
stage.on(StageEvents.STAGE_STREAM_MUTE_CHANGED, (participant, stream) => {})
```

For most of these events, the corresponding `ParticipantInfo` is provided.

It is not expected that the information provided by the events impacts the return values of the strategy. For example, the return value of `shouldSubscribeToParticipant` is not expected to change when `STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED` is called. If the host application wants to subscribe to a particular participant, it should return the desired subscription type regardless of that participant's publish state. The SDK is responsible for ensuring that the desired state of the strategy is acted on at the correct time based on the state of the stage.

Publish a Media Stream

Local devices like microphones and cameras are retrieved using the same steps as outlined above in Retrieve a MediaStream from a Device. In the example we use `MediaStream` to create a list of `LocalStageStream` objects used for publishing by the SDK:

```
try {
    // Get stream using steps outlined in document above
    const stream = await getMediaStreamFromDevice();

    let streamsToPublish = stream.getTracks().map(track => {
        new LocalStageStream(track)
    });

    // Create stage with strategy, or update existing strategy
    const strategy = {
        stageStreamsToPublish: () => streamsToPublish
    }
}
```

Publish a Screenshare

Applications often need to publish a screenshare in addition to the user's web camera. Publishing a screenshare necessitates creating an additional `Stage` with its own unique token.

```
// Invoke the following lines to get the screenshare's tracks
const media = await navigator.mediaDevices.getDisplayMedia({
   video: {
      width: {
         max: 1280,
      },
      height: {
         max: 720,
      }
   }
});
const screenshare = { videoStream: new LocalStageStream(media.getVideoTracks()[0]) };
const screenshareStrategy = {
   stageStreamsToPublish: () => {
      return [screenshare.videoStream];
   },
   shouldPublishParticipant: (participant) => {
      return true;
   },
   shouldSubscribeToParticipant: (participant) => {
      return SubscribeType.AUDIO_VIDEO;
   }
}
const screenshareStage = new Stage(screenshareToken, screenshareStrategy);
await screenshareStage.join();
```

Display and Remove Participants

After subscribing is completed, you receive an array of `StageStream` objects through the `STAGE_PARTICIPANT_STREAMS_ADDED` event. The event also gives you participant info to help when displaying media streams:

```
stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED, (participant, streams) => {
    const streamsToDisplay = streams;

    if (participant.isLocal) {
        // Ensure to exclude local audio streams, otherwise echo will occur
        streamsToDisplay = streams.filter(stream => stream.streamType ===
 StreamType.VIDEO)
    }

    // Create or find video element already available in your application
    const videoEl = getParticipantVideoElement(participant.id);

    // Attach the participants streams
    videoEl.srcObject = new MediaStream();
    streamsToDisplay.forEach(stream =>
 videoEl.srcObject.addTrack(stream.mediaStreamTrack));
})
```

When a participant stops publishing or is unsubscribed from a stream, the `STAGE_PARTICIPANT_STREAMS_REMOVED` function is called with the streams that were removed. Host applications should use this as a signal to remove the participant's video stream from the DOM.

`STAGE_PARTICIPANT_STREAMS_REMOVED` is invoked for all scenarios in which a stream might be removed, including:

- The remote participant stops publishing.
- A local device unsubscribes or changes subscription from `AUDIO_VIDEO` to `AUDIO_ONLY`.
- The remote participant leaves the stage.
- The local participant leaves the stage.

Because `STAGE_PARTICIPANT_STREAMS_REMOVED` is invoked for all scenarios, no custom business logic is required around removing participants from the UI during remote or local leave operations.

Mute and Unmute Media Streams

`LocalStageStream` objects have a `setMuted` function that controls whether the stream is muted. This function can be called on the stream before or after it is returned from the `stageStreamsToPublish` strategy function.

Important: If a new `LocalStageStream` object instance is returned by `stageStreamsToPublish` after a call to `refreshStrategy`, the mute state of the new stream object is applied to the stage. Be careful when creating new `LocalStageStream` instances to make sure the expected mute state is maintained.

Monitor Remote Participant Media Mute State

When participants change the mute state of their video or audio, the `STAGE_STREAM_MUTE_CHANGED` event is triggered with a list of streams that have changed. Use the `isMuted` property on `StageStream` to update your UI accordingly:

```
stage.on(StageEvents.STAGE_STREAM_MUTE_CHANGED, (participant, stream) => {
   if (stream.streamType === 'video' && stream.isMuted) {
       // handle UI changes for video track getting muted
   }
})
```

Also, you can look at StageParticipantInfo for state information on whether audio or video is muted:

```
stage.on(StageEvents.STAGE_STREAM_MUTE_CHANGED, (participant, stream) => {
   if (participant.videoStopped || participant.audioMuted) {
       // handle UI changes for either video or audio
   }
})
```

Get WebRTC Statistics

To get the latest WebRTC statistics for a publishing stream or subscribing stream, use `getStats` on `StageStream`. This is an asynchronous method with which you can retrieve statistics either via await or by chaining a promise. The result is an `RTCStatsReport` which is a dictionary containing all standard statistics.

```
try {
   const stats = await stream.getStats();
} catch (error) {
   // Unable to retrieve stats
}
```

Optimizing Media

It's recommended to limit `getUserMedia` and `getDisplayMedia` calls to the following constraints for the best performance:

```
const CONSTRAINTS = {
    video: {
        width: { ideal: 1280 }, // Note: flip width and height values if portrait is
 desired
        height: { ideal: 720 },
        framerate: { ideal: 30 },
    },
};
```

You can further constrain the media through additional options passed to the `LocalStageStream` constructor:

```
const localStreamOptions = {
    minBitrate?: number;
    maxBitrate?: number;
    maxFramerate?: number;
    simulcast: {
        enabled: boolean
    }
}
const localStream = new LocalStageStream(track, localStreamOptions)
```

In the code above:

- `minBitrate` sets a minimum bitrate that the browser should be expected to use. However, a low complexity video stream may push the encoder to go lower than this bitrate.
- `maxBitrate` sets a maximum bitrate that the browser should be expected to not exceed for this stream.
- `maxFramerate` sets a maximum frame rate that the browser should be expected to not exceed for this stream.
- The `simulcast` option is usable only on Chromium-based browsers. It enables sending three rendition layers of the stream.
  - This allows the server to choose which rendition to send to other participants, based on their networking limitations.
  - When `simulcast` is specified along with a `maxBitrate` and/or `maxFramerate` value, it is expected that the highest rendition layer will be configured with these values in mind, provided the `maxBitrate` does not go below the internal SDK's second highest layer's default `maxBitrate` value of 900 kbps.
  - If `maxBitrate` is specified as too low compared to the second highest layer's default value, `simulcast` will be disabled.
  - `simulcast` cannot be toggled on and off without republishing the media through a combination of having `shouldPublishParticipant` return `false`, calling `refreshStrategy`, having `shouldPublishParticipant` return `true` and calling `refreshStrategy` again.

Get Participant Attributes

If you specify attributes in the `CreateParticipantToken` endpoint request, you can see the attributes in `StageParticipantInfo` properties:

```
stage.on(StageEvents.STAGE_PARTICIPANT_JOINED, (participant) => {
   console.log(`Participant ${participant.id} info:`, participant.attributes);
})
```

Handling Network Issues

When the local device's network connection is lost, the SDK internally tries to reconnect without any user action. In some cases, the SDK is not successful and user action is needed.

Broadly the state of the stage can be handled via the `STAGE_CONNECTION_STATE_CHANGED` event:

```
stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {
   switch (state) {
      case StageConnectionState.DISCONNECTED:
         // handle disconnected UI
         break;
      case StageConnectionState.CONNECTING:
         // handle establishing connection UI
         break;
      case StageConnectionState.CONNECTED:
         // SDK is connected to the Stage
         break;
      case StageConnectionState.ERRORED:
         // unrecoverable error detected, please re-instantiate
         Break;
})
```

In general, encountering errors after successfully joining a stage indicates that the SDK lost the connection and was unsuccessful in reestablishing a connection. Create a new `Stage` object and try to join when network conditions improve.

Broadcast the Stage to an IVS Channel

To broadcast a stage, create a separate `IVSBroadcastClient` session and then follow the usual instructions for broadcasting with the SDK, described above. The list of `StageStream` exposed via `STAGE_PARTICIPANT_STREAMS_ADDED` can be used to retrieve the participant media streams which can be applied to the broadcast stream composition, as follows:

```
// Setup client with preferred settings
const broadcastClient = getIvsBroadcastClient();
```

```
stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED, (participant, streams) => {
    streams.forEach(stream => {
        const inputStream = new MediaStream([stream.mediaStreamTrack]);
        switch (stream.streamType) {
            case StreamType.VIDEO:
                broadcastClient.addVideoInputDevice(inputStream, `video-
${participant.id}`, {
                    index: DESIRED_LAYER,
                    width: MAX_WIDTH,
                    height: MAX_HEIGHT
                });
                break;
            case StreamType.AUDIO:
                broadcastClient.addAudioInputDevice(inputStream, `audio-
${participant.id}`);
                break;
        }
    })
})
```

Optionally, you can composite a stage and broadcast it to an IVS low-latency channel, to reach a larger audience. See Enabling Multiple Hosts on an Amazon IVS Stream in the IVS Low-Latency Streaming User Guide.

Known Issues and Workarounds

- When closing browser tabs or exiting browsers without calling `stage.leave()`, users can still appear in the session with a frozen frame or black screen for up to 10 seconds.

  Workaround: None.

- Safari sessions intermittently appear with a black screen to users joining after a session has begun.

  Workaround: Refresh the browser and reconnect the session.

- Safari does not recover gracefully from switching networks.

  Workaround: Refresh the browser and reconnect the session.

- The developer console repeats an `Error: UnintentionalError at StageSocket.onClose` error.

Workaround: Only one stage can be created per participant token. This error occurs when more than one `Stage` instance is created with the same participant token, regardless of whether the instance is on one device or multiple devices.

- You may have trouble maintaining a `StageParticipantPublishState.PUBLISHED` state and may receive repeated `StageParticipantPublishState.ATTEMPTING_PUBLISH` states when listening to the `StageEvents.STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED` event.

  Workaround: Constrain video resolution to 720p when invoking `getUserMedia` or `getDisplayMedia`. Specifically, your `getUserMedia` and `getDisplayMedia` constraint values for width and height must not exceed 921600 (1280*720) when multiplied together.

Safari Limitations

- Denying a permissions prompt requires resetting the permission in Safari website settings at the OS level.
- Safari does not natively detect all devices as effectively as Firefox or Chrome. For example, OBS Virtual Camera does not get detected.

Firefox Limitations

- System permissions need to be enabled for Firefox to screen share. After enabling them, the user must restart Firefox for it to work correctly; otherwise, if permissions are perceived as blocked, the browser will throw a NotFoundError exception.
- The `getCapabilities` method is missing. This means users cannot get the media track's resolution or aspect ratio. See this bugzilla thread.
- Several `AudioContext` properties are missing; e.g., latency and channel count. This could pose a problem for advanced users who want to manipulate the audio tracks.
- Camera feeds from `getUserMedia` are restricted to a 4:3 aspect ratio on MacOS. See bugzilla thread 1 and bugzilla thread 2.
- Audio capture is not supported with `getDisplayMedia`. See this bugzilla thread.
- Framerate in screen capture is suboptimal (approximately 15fps?). See this bugzilla thread.

Mobile Web Limitations

- getDisplayMedia screen sharing is unsupported on mobile devices.

  Workaround: None.
- Participant takes 15-30 seconds to leave when closing a browser without calling `leave()`.

  Workaround: Add a UI that encourages users to properly disconnect.
- Backgrounding app causes publishing video to stop.

  Workaround: Display a UI slate when the publisher is paused.
- Video framerate drops for approximately 5 seconds after unmuting a camera on Android devices.

  Workaround: None.
- The video feed is stretched on rotation for iOS 16.0.

  Workaround: Display a UI outlining this known OS issue.
- Switching the audio-input device automatically switches the audio-output device.

  Workaround: None.
- Backgrounding the browser causes the publishing stream to go black and produce only audio.

  Workaround: None. This is for security reasons.

Error Handling

This section is an overview of error conditions, how the Web Broadcast SDK reports them to the application, and what an application should do when those errors are encountered. There are four categories of errors:

```
try {
  stage = new Stage(token, strategy);
} catch (e) {
  // 1) stage instantiation errors
}

try {
  await stage.join();
} catch (e) {
  // 2) stage join errors
}

stage.on(StageEvents.STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED, (participantInfo, state)
 => {
  if (state === StageParticipantPublishState.ERRORED) {
    // 3) stage publish errors
  }
});

stage.on(StageEvents.STAGE_PARTICIPANT_SUBSCRIBE_STATE_CHANGED, (participantInfo,
 state) => {
  if (state === StageParticipantSubscribeState.ERRORED) {
    // 4) stage subscribe errors
  }
});
```

Stage Instantiation Errors

Stage instantiation does not remotely validate tokens, but it does check for some basic token issues that can be validated on the client-side. As a result, the SDK may throw an error.

Malformed Participant Token

This occurs when the stage token is malformed. When instantiating a Stage, the SDK throws an error with this message: "Error parsing Stage Token."

Action: Create a valid token and retry instantiating.

Stage Join Errors

These are the errors that may occur when initially attempting to join a stage.

Stage was Deleted

This occurs when joining a stage (associated with a token) which was deleted. The `join` SDK method throws an error with this message: "Operation timed out."

Action: Create a valid token with a new stage and retry joining.

Expired Participant Token

This occurs when the token is expired. The `join` SDK method throws an error with this message: "Token expired and is no longer valid."

Action: Create a new token and retry joining.

Invalid or Revoked Participant Token

This occurs when the token is not valid or was revoked/disconnected. The `join` SDK method throws an error with this message: "Operation timed out."

Action: Create a new token and retry joining.

Disconnected Token

This occurs when the stage token is not malformed but is rejected by the Stages server. The `join` SDK method throws an error with this message: "Operation timed out."

Action: Create a valid token and retry joining.

Network Errors for Initial Join

This occurs when the SDK cannot contact the Stages server to establish a connection. The `join` SDK method throws an error with this message: "Operation timed out."

Action: Wait for the device's connectivity to recover and retry joining.

Network Errors when Already Joined

If the device's network connection goes down, the SDK may lose its connection to Stage servers. You may see errors in the console because the SDK can no longer reach backend services. POSTs to https://broadcast.stats.live-video.net will fail.

If you are publishing and/or subscribing, you will see errors in the console related to attempts to publish/subscribe.

Internally the SDK will try to reconnect with an exponential backoff strategy.

Action: Wait for the device's connectivity to recover. If publishing or subscribing, refresh the strategy to ensure republication of your media stream(s).

Publish and Subscribe Errors

Publish Error: Publish States

The SDK reports `ERRORED` when a publish fails. This can occur due to network conditions or if a stage is at capacity for publishers.

```
stage.on(StageEvents.STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED, (participantInfo, state)
 => {
  if (state === StageParticipantPublishState.ERRORED) {
      // Handle
  }
});
```

Action: Refresh the strategy to attempt republication of your media stream(s).

Subscribe Errors

The SDK reports ERRORED when a subscribe fails. This can occur due to network conditions or if a stage is at capacity for subscribers.

```
stage.on(StageEvents.STAGE_PARTICIPANT_SUBSCRIBE_STATE_CHANGED, (participantInfo,
 state) => {
  if (state === StageParticipantSubscribeState.ERRORED) {
    // 4) stage subscribe errors
  }
});
```

Action: Refresh the strategy to try a new subscribe.

IVS Broadcast SDK: Android Guide (Real-Time Streaming)

The IVS real-time streaming Android broadcast SDK enables participants to send and receive video on Android.

The `com.amazonaws.ivs.broadcast` package implements the interface described in this document. The SDK supports the following operations:

- Join a stage
- Publish media to other participants in the stage
- Subscribe to media from other participants in the stage
- Manage and monitor video and audio published to the stage
- Get WebRTC statistics for each peer connection
- All operations from the IVS low-latency streaming Android broadcast SDK

Latest version of Android broadcast SDK: 1.17.0 (Release Notes)

Reference documentation: For information on the most important methods available in the Amazon IVS Android broadcast SDK, see the reference documentation at https://aws.github.io/amazon-ivs-broadcast-docs/1.17.0/android/.

Sample code: See the Android sample repository on GitHub: https://github.com/aws-samples/amazon-ivs-broadcast-android-sample.

Platform requirements: Android 9.0 and later.

Getting Started

Install the Library

To add the Amazon IVS Android broadcast library to your Android development environment, add the library to your module's `build.gradle` file, as shown here (for the latest version of the Amazon IVS broadcast SDK):

```
repositories {
    mavenCentral()
}

dependencies {
     implementation 'com.amazonaws:ivs-broadcast:1.17.0:stages@aar'
}
```

Add the following permission to your manifest to allow the SDK to enable and disable the speakerphone:

```
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
```

Alternately, to install the SDK manually, download the latest version from this location:

https://search.maven.org/artifact/com.amazonaws/ivs-broadcast

Be sure to download the `aar` with `-stages` appended.

Request Permissions

Your app must request permission to access the user's camera and mic. (This is not specific to Amazon IVS; it is required for any application that needs access to cameras and microphones.)

Here, we check whether the user has already granted permissions and, if not, ask for them:

```
final String[] requiredPermissions =
         { Manifest.permission.CAMERA, Manifest.permission.RECORD_AUDIO };

for (String permission : requiredPermissions) {
    if (ContextCompat.checkSelfPermission(this, permission)
                != PackageManager.PERMISSION_GRANTED) {
        // If any permissions are missing we want to just request them all.
        ActivityCompat.requestPermissions(this, requiredPermissions, 0x100);
        break;
    }
}
```

Here, we get the user's response:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode,
               permissions, grantResults);
    if (requestCode == 0x100) {
        for (int result : grantResults) {
            if (result == PackageManager.PERMISSION_DENIED) {
                return;
            }
        }
        setupBroadcastSession();
    }
}
```

Publishing and Subscribing

Concepts

Three core concepts underlie real-time functionality: stage, strategy, and renderer. The design goal is minimizing the amount of client-side logic necessary to build a working product.

Stage

The `Stage` class is the main point of interaction between the host application and the SDK. It represents the stage itself and is used to join and leave the stage. Creating and joining a stage requires a valid, unexpired token string from the control plane (represented as `token`). Joining and leaving a stage are simple.

```
Stage stage = new Stage(context, token, strategy);

try {
 stage.join();
} catch (BroadcastException exception) {
 // handle join exception
}

stage.leave();
```

The `Stage` class is also where the `StageRenderer` can be attached:

```
stage.addRenderer(renderer); // multiple renderers can be added
```

Strategy

The `Stage.Strategy` interface provides a way for the host application to communicate the desired state of the stage to the SDK. Three functions need to be implemented: `shouldSubscribeToParticipant`, `shouldPublishFromParticipant`, and `stageStreamsToPublishForParticipant`. All are discussed below.

Subscribing to Participants

```
Stage.SubscribeType shouldSubscribeToParticipant(@NonNull Stage stage, @NonNull
 ParticipantInfo participantInfo);
```

When a remote participant joins the stage, the SDK queries the host application about the desired subscription state for that participant. The options are `NONE`, `AUDIO_ONLY`, and `AUDIO_VIDEO`. When returning a value for this function, the host application does not need to worry about the publish state, current subscription state, or stage connection state. If `AUDIO_VIDEO` is returned, the SDK waits until the remote participant is publishing before subscribing, and it updates the host application through the renderer throughout the process.

Here is a sample implementation:

```
@Override
Stage.SubscribeType shouldSubscribeToParticipant(@NonNull Stage stage, @NonNull
 ParticipantInfo participantInfo) {
 return Stage.SubscribeType.AUDIO_VIDEO;
}
```

This is the complete implementation of this function for a host application that always wants all participants to see each other; e.g., a video chat application.

More advanced implementations also are possible. Use the `userInfo` property on `ParticipantInfo` to selectively subscribe to participants based on server-provided attributes:

```
@Override
Stage.SubscribeType shouldSubscribeToParticipant(@NonNull Stage stage, @NonNull
 ParticipantInfo participantInfo) {
 switch(participantInfo.userInfo.get(“role”)) {
  case “moderator”:
   return Stage.SubscribeType.NONE;
  case “guest”:
   return Stage.SubscribeType.AUDIO_VIDEO;
  default:
   return Stage.SubscribeType.NONE;
 }
}
```

This can be used to create a stage where moderators can monitor all guests without being seen or heard themselves. The host application could use additional business logic to let moderates see each other but remain invisible to guests.

Publishing

```
boolean shouldPublishFromParticipant(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo);
```

Once connected to the stage, the SDK queries the host application to see if a particular participant should publish. This is invoked only on local participants that have permission to publish based on the provided token.

Here is a sample implementation:

```
@Override
boolean shouldPublishFromParticipant(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo) {
 return true;
}
```

This is for a standard video chat application where users always want to publish. They can mute and unmute their audio and video, to instantly be hidden or seen/heard. (They also can use publish/unpublish, but that is much slower. Mute/unmute is preferable for use cases where changing visibility often is desirable.)

Choosing Streams to Publish

```
@Override
List<LocalStageStream> stageStreamsToPublishForParticipant(@NonNull Stage stage,
 @NonNull ParticipantInfo participantInfo);
}
```

When publishing, this is used to determine what audio and video streams should be published. This is covered in more detail later in Publish a Media Stream.

Updating the Strategy

The strategy is intended to be dynamic: the values returned from any of the above functions can be changed at any time. For example, if the host application does not want to publish until the end user taps a button, you could return a variable from `shouldPublishFromParticipant` (something like `hasUserTappedPublishButton`). When that variable changes based on an interaction by the end user, call `stage.refreshStrategy()` to signal to the SDK that it should query the strategy for the latest values, applying only things that have changed. If the SDK observes that the `shouldPublishFromParticipant` value has changed, it will start the publish process. If the SDK queries and all functions return the same value as before, the `refreshStrategy` call will not perform any modifications to the stage.

If the return value of `shouldSubscribeToParticipant` changes from `AUDIO_VIDEO` to `AUDIO_ONLY`, the video stream will be removed for all participants with changed returned values, if a video stream existed previously.

Generally, the stage uses the strategy to most efficiently apply the difference between the previous and current strategies, without the host application needing to worry about all the state required to manage it properly. Because of this, think of calling `stage.refreshStrategy()` as a cheap operation, because it does nothing unless the strategy changes.

Renderer

The `StageRenderer` interface communicates the state of the stage to the host application. Updates to the host application's UI usually can be powered entirely by the events provided by the renderer. The renderer provides the following functions:

```
void onParticipantJoined(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo);

void onParticipantLeft(@NonNull Stage stage, @NonNull ParticipantInfo participantInfo);

void onParticipantPublishStateChanged(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo, @NonNull Stage.PublishState publishState);

void onParticipantSubscribeStateChanged(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo, @NonNull Stage.SubscribeState subscribeState);

void onStreamsAdded(@NonNull Stage stage, @NonNull ParticipantInfo participantInfo,
 @NonNull List<StageStream> streams);

void onStreamsRemoved(@NonNull Stage stage, @NonNull ParticipantInfo participantInfo,
 @NonNull List<StageStream> streams);

void onStreamsMutedChanged(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo, @NonNull List<StageStream> streams);

void onError(@NonNull BroadcastException exception);

void onConnectionStateChanged(@NonNull Stage stage, @NonNull Stage.ConnectionState
 state, @Nullable BroadcastException exception);
```

For most of these methods, the corresponding `Stage` and `ParticipantInfo` are provided.

It is not expected that the information provided by the renderer impacts the return values of the strategy. For example, the return value of `shouldSubscribeToParticipant` is not expected to change when `onParticipantPublishStateChanged` is called. If the host application wants to subscribe to a particular participant, it should return the desired subscription type regardless of that participant's publish state. The SDK is responsible for ensuring that the desired state of the strategy is acted on at the correct time based on the state of the stage.

The `StageRenderer` can be attached to the stage class:

```
stage.addRenderer(renderer); // multiple renderers can be added
```

Note that only publishing participants trigger `onParticipantJoined`, and whenever a participant stops publishing or leaves the stage session, `onParticipantLeft` is triggered.

Publish a Media Stream

Local devices such as built-in microphones and cameras are discovered via `DeviceDiscovery`. Here is an example of selecting the front-facing camera and default microphone, then return them as `LocalStageStreams` to be published by the SDK:

```
DeviceDiscovery deviceDiscovery = new DeviceDiscovery(context);

List<Device> devices = deviceDiscovery.listLocalDevices();
List<LocalStageStream> publishStreams = new ArrayList<LocalStageStream>();

Device frontCamera = null;
Device microphone = null;

// Create streams using the front camera, first microphone
for (Device device : devices) {
 Device.Descriptor descriptor = device.getDescriptor();
 if (!frontCamera && descriptor.type == Device.Descriptor.DeviceType.Camera &&
 descriptor.position = Device.Descriptor.Position.FRONT) {
  front Camera = device;
 }
 if (!microphone && descriptor.type == Device.Descriptor.DeviceType.Microphone) {
  microphone = device;
 }
}

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera);
AudioLocalStageStream microphoneStream = new AudioLocalStageStream(microphoneDevice);

publishStreams.add(cameraStream);
publishStreams.add(microphoneStream);

// Provide the streams in Stage.Strategy
@Override
@NonNull List<LocalStageStream> stageStreamsToPublishForParticipant(@NonNull Stage
 stage, @NonNull ParticipantInfo participantInfo) {
```

```
 return publishStreams;
}
```

Display and Remove Participants

After subscribing is completed, you will receive an array of `StageStream` objects through the renderer's `onStreamsAdded` function. You can retrieve the preview from an `ImageStageStream`:

```
ImagePreviewView preview = ((ImageStageStream)stream).getPreview();

// Add the view to your view hierarchy
LinearLayout previewHolder = findViewById(R.id.previewHolder);
preview.setLayoutParams(new LinearLayout.LayoutParams(
  LinearLayout.LayoutParams.MATCH_PARENT,
  LinearLayout.LayoutParams.MATCH_PARENT));
previewHolder.addView(preview);
```

You can retrieve the audio-level stats from an `AudioStageStream`:

```
((AudioStageStream)stream).setStatsCallback((peak, rms) -> {
 // handle statistics
});
```

When a participant stops publishing or is unsubscribed from, the `onStreamsRemoved` function is called with the streams that were removed. Host applications should use this as a signal to remove the participant's video stream from the view hierarchy.

`onStreamsRemoved` is invoked for all scenarios in which a stream might be removed, including:

- The remote participant stops publishing.
- A local device unsubscribes or changes subscription from `AUDIO_VIDEO` to `AUDIO_ONLY`.
- The remote participant leaves the stage.
- The local participant leaves the stage.

Because `onStreamsRemoved` is invoked for all scenarios, no custom business logic is required around removing participants from the UI during remote or local leave operations.

Mute and Unmute Media Streams

`LocalStageStream` objects have a `setMuted` function that controls whether the stream is muted. This function can be called on the stream before or after it is returned from the `streamsToPublishForParticipant` strategy function.

Important: If a new `LocalStageStream` object instance is returned by `streamsToPublishForParticipant` after a call to `refreshStrategy`, the mute state of the new stream object is applied to the stage. Be careful when creating new `LocalStageStream` instances to make sure the expected mute state is maintained.

Monitor Remote Participant Media Mute State

When a participant changes the mute state of their video or audio stream, the renderer `onStreamMutedChanged` function is invoked with a list of streams that have changed. Use the `getMuted` method on `StageStream` to update your UI accordingly.

```
@Override
void onStreamsMutedChanged(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo, @NonNull List<StageStream> streams) {
 for (StageStream stream : streams) {
  boolean muted = stream.getMuted();
  // handle UI changes
 }
}
```

Get WebRTC Statistics

To get the latest WebRTC statistics for a publishing stream or a subscribing stream, use `requestRTCStats` on `StageStream`. When a collection is completed, you will receive statistics through the `StageStream.Listener` which can be set on `StageStream`.

```
stream.requestRTCStats();

@Override
void onRTCStats(Map<String, Map<String, String>> statsMap) {
 for (Map.Entry<String, Map<String, string>> stat : statsMap.entrySet()) {
  for(Map.Entry<String, String> member : stat.getValue().entrySet()) {
   Log.i(TAG, stat.getKey() + “ has member “ + member.getKey() + “ with value “ +
 member.getValue());
```

```
   }
  }
 }
```

Get Participant Attributes

If you specify attributes in the `CreateParticipantToken` endpoint request, you can see the attributes in `ParticipantInfo` properties:

```
@Override
void onParticipantJoined(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo) {
 for (Map.Entry<String, String> entry : participantInfo.userInfo.entrySet()) {
  Log.i(TAG, “attribute: “ + entry.getKey() + “ = “ + entry.getValue());
 }
}
```

Continue Session in the Background

When the app enters the background, you may want to stop publishing or subscribe only to other remote participants' audio. To accomplish this, update your `Strategy` implementation to stop publishing, and subscribe to `AUDIO_ONLY` (or `NONE`, if applicable).

```
// Local variables before going into the background
boolean shouldPublish = true;
Stage.SubscribeType subscribeType = Stage.SubscribeType.AUDIO_VIDEO;

// Stage.Strategy implementation
@Override
boolean shouldPublishFromParticipant(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo) {
 return shouldPublish;
}

@Override
Stage.SubscribeType shouldSubscribeToParticipant(@NonNull Stage stage, @NonNull
 ParticipantInfo participantInfo) {
 return subscribeType;
}

// In our Activity, modify desired publish/subscribe when we go to background, then
 call refreshStrategy to update the stage
```

```
@Override
void onStop() {
 super.onStop();
 shouldPublish = false;
 subscribeTpye = Stage.SubscribeType.AUDIO_ONLY;
 stage.refreshStrategy();
}
```

Enable/Disable Layered Encoding with Simulcast

When publishing a media stream, the SDK transmits high-quality and low-quality video streams, so remote participants can subscribe to the stream even if they have limited downlink bandwidth. Layered encoding with simulcast is on by default. You can disable it by using the `StageVideoConfiguration.Simulcast` class:

```
// Disable Simulcast
StageVideoConfiguration config = new StageVideoConfiguration();
config.simulcast.setEnabled(false);

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera, config);

// Other Stage implementation code
```

Video-Configuration Limitations

The SDK does not support forcing portrait mode or landscape mode using `StageVideoConfiguration.setSize(BroadcastConfiguration.Vec2 size)`. In portrait orientation, the smaller dimension is used as the width; in landscape orientation, the height. This means that the following two calls to `setSize` have the same effect on the video configuration:

```
StageVideo Configuration config = new StageVideo Configuration();

config.setSize(BroadcastConfiguration.Vec2(720f, 1280f);
config.setSize(BroadcastConfiguration.Vec2(1280f, 720f);
```

Handling Network Issues

When the local device’s network connection is lost, the SDK internally tries to reconnect without any user action. In some cases, the SDK is not successful and user action is needed. There are two main errors related to losing the network connection:

- Error code 1400, message: "PeerConnection is lost due to unknown network error"
- Error code 1300, message: "Retry attempts are exhausted"

If the first error is received but the second is not, the SDK is still connected to the stage and will try to reestablish its connections automatically. As a safeguard, you can call `refreshStrategy` without any changes to the strategy method's return values, to trigger a manual reconnect attempt.

If the second error is received, the SDK's reconnect attempts have failed and the local device is no longer connected to the stage. In this case, try to rejoin the stage by calling `join` after your network connection has been reestablished.

In general, encountering errors after joining a stage successfully indicates that the SDK was unsuccessful in reestablishing a connection. Create a new `Stage` object and try to join when network conditions improve.

Using Bluetooth Microphones

To publish using Bluetooth microphone devices, you must start a Bluetooth SCO connection:

```
Bluetooth.startBluetoothSco(context);
// Now bluetooth microphones can be used
…
// Must also stop bluetooth SCO
Bluetooth.stopBluetoothSco(context);
```

Known Issues and Workarounds

- When an Android device goes to sleep and wakes up, it is possible for the preview to be in a frozen state.

  Workaround: Create and use a new `Stage`.
- When a participant joins with a token that is being used by another participant, the first connection is disconnected without a specific error.

  Workaround: None.
- There is a rare issue where the publisher is publishing but the publish state that subscribers receive is `inactive`.

Workaround: Try leaving and then joining the session. If the issue remains, create a new token for the publisher.

- A rare audio-distortion issue may occur intermittently during a stage session, typically on calls of longer durations.

  Workaround: The participant with distorted audio can either leave and rejoin the session, or unpublish and republish their audio to fix the issue.
- External microphones are not supported when publishing to a stage.

  Workaround: Do not use an external microphone connected via USB for publishing to a stage.
- Publishing to a stage with screen share using `createSystemCaptureSources` is not supported.

  Workaround: Manage the system capture manually, using custom image-input sources and custom audio-input sources.
- When an `ImagePreviewView` is removed from a parent (e.g., `removeView()` is called at the parent), the `ImagePreviewView` is released immediately. The `ImagePreviewView` does not show any frames when it is added to another parent view.

  Workaround: Request another preview using `getPreview`.
- When joining a stage with a Samsung Galaxy S22/+ with Android 12, you may encounter a 1401 error and the local device fails to join the stage or joins but has no audio.

  Workaround: Upgrade to Android 13.
- When joining a stage with a Nokia X20 on Android 13, the camera may fail to open and an exception is thrown.

  Workaround: None.
- Devices with the MediaTek Helio chipset may not render video of remote participants properly.

  Workaround: None.
- On a few devices, the device OS may choose a different microphone than what's selected through the SDK. This is because the Amazon IVS Broadcast SDK cannot control how the `VOICE_COMMUNICATION` audio route is defined, as it varies according to different device manufacturers.

  Workaround: None.

- Some Android video encoders cannot be configured with a video size less than 176x176. Configuring a smaller size causes an error and prevents streaming.

  Workaround: Do not configure the video size to be less than 176x176.

Error Handling

Fatal vs. Non-Fatal Errors

The error object has an "is fatal" boolean field of `BroadcastException`.

In general, fatal errors are related to connection to the Stages server (either a connection cannot be established or is lost and cannot be recovered). The application should re-create the stage and re-join, possibly with a new token or when the device's connectivity recovers.

Non-fatal errors generally are related to the publish/subscribe state and are handled by the SDK, which retries the publish/subscribe operation.

You can check this property:

```
try {
  stage.join(...)
} catch (e: BroadcastException) {
  If (e.isFatal) {
    // the error is fatal
```

Join Errors

Malformed Token

This happens when the stage token is malformed.

The SDK throws a Java exception from a call to `stage.join`, with error code = 1000 and fatal = true.

Action: Create a valid token and retry joining.

Expired Token

This happens when the stage token is expired.

The SDK throws a Java exception from a call to `stage.join`, with error code = 1001 and fatal = true.

Action: Create a new token and retry joining.

Invalid or Revoked Token

This happens when the stage token is not malformed but is rejected by the Stages server. This is reported asynchronously through the application-supplied stage renderer.

The SDK calls `onConnectionStateChanged` with an exception, with error code = 1026 and fatal = true.

Action: Create a valid token and retry joining.

Network Errors for Initial Join

This happens when the SDK cannot contact the Stages server to establish a connection. This is reported asynchronously through the application-supplied stage renderer.

The SDK calls `onConnectionStateChanged` with an exception, with error code = 1300 and fatal = true.

Action: Wait for the device's connectivity to recover and retry joining.

Network Errors when Already Joined

If the device's network connection goes down, the SDK may lose its connection to Stage servers. This is reported asynchronously through the application-supplied stage renderer.

The SDK calls `onConnectionStateChanged` with an exception, with error code = 1300 and fatal = true.

Action: Wait for the device's connectivity to recover and retry joining.

Publish/Subscribe Errors

Initial

There are several errors:

- MultihostSessionOfferCreationFailPublish (1020)
- MultihostSessionOfferCreationFailSubscribe (1021)

- MultihostSessionNoIceCandidates (1022)
- MultihostSessionStageAtCapacity (1024)
- SignallingSessionCannotRead (1201)
- SignallingSessionCannotSend (1202)
- SignallingSessionBadResponse (1203)

These are reported asynchronously through the application-supplied stage renderer.

The SDK retries the operation for a limited number of times. During retries, the publish/subscribe state is `ATTEMPTING_PUBLISH` / `ATTEMPTING_SUBSCRIBE`. If the retry attempts succeed, the state changes to `PUBLISHED` / `SUBSCRIBED`.

The SDK calls `onError` with the relevant error code and fatal = false.

Action: No action is needed, as the SDK retries automatically. Optionally, the application can refresh the strategy to force more retries.

Already Established, Then Fail

A publish or subscribe can fail after it is established, most likely due to a network error. The error code for a "peer connection lost due to network error" is 1400.

This is reported asynchronously through the application-supplied stage renderer.

The SDK retries the publish/subscribe operation. During retries, the publish/subscribe state is `ATTEMPTING_PUBLISH` / `ATTEMPTING_SUBSCRIBE`. If the retry attempts succeed, the state changes to `PUBLISHED` / `SUBSCRIBED`.

The SDK calls `onError` with the error code = 1400 and fatal = false.

Action: No action is needed, as the SDK retries automatically. Optionally, the application can refresh the strategy to force more retries. In the event of total connectivity loss, it’s likely that the connection to Stages will fail too.

IVS Broadcast SDK: iOS Guide (Real-Time Streaming)

The IVS real-time streaming iOS broadcast SDK enables participants to send and receive video on iOS.

The `AmazonIVSBroadcast` module implements the interface described in this document. The following operations are supported:

- Join a stage
- Publish media to other participants in the stage
- Subscribe to media from other participants in the stage
- Manage and monitor video and audio published to the stage
- Get WebRTC statistics for each peer connection
- All operations from the IVS low-latency streaming iOS broadcast SDK

Latest version of iOS broadcast SDK: 1.17.0 (Release Notes)

Reference documentation: For information on the most important methods available in the Amazon IVS iOS broadcast SDK, see the reference documentation at https://aws.github.io/amazon-ivs-broadcast-docs/1.17.0/ios/.

Sample code: See the iOS sample repository on GitHub: https://github.com/aws-samples/amazon-ivs-broadcast-ios-sample.

Platform requirements: iOS 14 or greater

Getting Started

Install the Library

We recommend that you integrate the broadcast SDK via CocoaPods. (Alternatively, you can manually add the framework to your project.)

Recommended: Integrate the Broadcast SDK (CocoaPods)

Real-time functionality is published as a subspec of the iOS Low-Latency Streaming broadcast SDK. This is so customers can choose to include or exclude it based on their feature needs. Including it increases the package size.

Releases are published via CocoaPods under the name `AmazonIVSBroadcast`. Add this dependency to your Podfile:

```
pod 'AmazonIVSBroadcast/Stages'
```

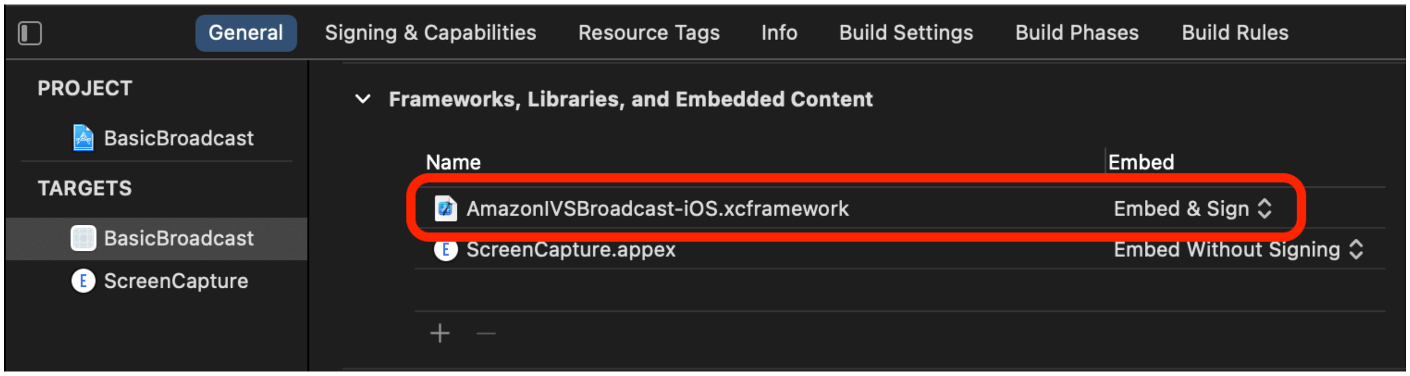
Run `pod install` and the SDK will be available in your `.xcworkspace`.

Important: The IVS real-time streaming broadcast SDK (i.e., with the stage subspec) includes all features of the IVS low-latency streaming broadcast SDK. *It is not possible to integrate both SDKs in the same project.* If you add the stage subspec via CocoaPods to your project, be sure to remove any other lines in the Podfile containing `AmazonIVSBroadcast`. For example, do not have both these lines in your Podfile:

```
pod 'AmazonIVSBroadcast'
pod 'AmazonIVSBroadcast/Stages'
```

Alternate Approach: Install the Framework Manually

1. Download the latest version from https://broadcast.live-video.net/1.17.0/AmazonIVSBroadcast-Stages.xcframework.zip.
2. Extract the contents of the archive. `AmazonIVSBroadcast.xcframework` contains the SDK for both device and simulator.
3. Embed `AmazonIVSBroadcast.xcframework` by dragging it into the **Frameworks, Libraries, and Embedded Content** section of the **General** tab for your application target.



Name	Embed
AmazonIVSBroadcast-iOS.xcframework	Embed & Sign
ScreenCapture.appex	Embed Without Signing

Request Permissions

Your app must request permission to access the user's camera and mic. (This is not specific to Amazon IVS; it is required for any application that needs access to cameras and microphones.)

Here, we check whether the user has already granted permissions and, if not, we ask for them:

```
switch AVCaptureDevice.authorizationStatus(for: .video) {
case .authorized: // permission already granted.
case .notDetermined:
   AVCaptureDevice.requestAccess(for: .video) { granted in
       // permission granted based on granted bool.
   }
case .denied, .restricted: // permission denied.
@unknown default: // permissions unknown.
}
```

You need to do this for both `.video` and `.audio` media types, if you want access to cameras and microphones, respectively.

You also need to add entries for `NSCameraUsageDescription` and `NSMicrophoneUsageDescription` to your `Info.plist`. Otherwise, your app will crash when trying to request permissions.

Disable the Application Idle Timer

This is optional but recommended. It prevents your device from going to sleep while using the broadcast SDK, which would interrupt the broadcast.

```
override func viewDidAppear(_ animated: Bool) {
   super.viewDidAppear(animated)
   UIApplication.shared.isIdleTimerDisabled = true
}
override func viewDidDisappear(_ animated: Bool) {
   super.viewDidDisappear(animated)
   UIApplication.shared.isIdleTimerDisabled = false
}
```

Publishing and Subscribing

Concepts

Three core concepts underlie real-time functionality: stage, strategy, and renderer. The design goal is minimizing the amount of client-side logic necessary to build a working product.

Stage

The `IVSStage` class is the main point of interaction between the host application and the SDK. The class represents the stage itself and is used to join and leave the stage. Creating or joining a stage requires a valid, unexpired token string from the control plane (represented as `token`). Joining and leaving a stage are simple.

```
let stage = try IVSStage(token: token, strategy: self)

try stage.join()

stage.leave()
```

The `IVSStage` class also is where the `IVSStageRenderer` and `IVSErrorDelegate` can be attached:

```
let stage = try IVSStage(token: token, strategy: self)
stage.errorDelegate = self
stage.addRenderer(self) // multiple renderers can be added
```

Strategy

The `IVSStageStrategy` protocol provides a way for the host application to communicate the desired state of the stage to the SDK. Three functions need to be implemented: `shouldSubscribeToParticipant`, `shouldPublishParticipant`, and `streamsToPublishForParticipant`. All are discussed below.

Subscribing to Participants

```
func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType
```

When a remote participant joins a stage, the SDK queries the host application about the desired subscription state for that participant. The options are `.none`, `.audioOnly`, and `.audioVideo`. When returning a value for this function, the host application does not need to worry about the publish state, current subscription state, or stage connection state. If `.audioVideo` is returned, the SDK waits until the remote participant is publishing before subscribing, and it updates the host application through the renderer throughout the process.

Here is a sample implementation:

```
func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType {
    return .audioVideo
}
```

This is the complete implementation of this function for a host application that always wants all participants to see each other; e.g., a video-chat application.

More advanced implementations also are possible. Use the `attributes` property on `IVSParticipantInfo` to selectively subscribe to participants based on server-provided attributes:

```
func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType {
    switch participant.attributes["role"] {
    case "moderator": return .none
    case "guest": return .audioVideo
    default: return .none
    }
}
```

This can be used to create a stage where moderators can monitor all guests without being seen or heard themselves. The host application could use additional business logic to let moderators see each other but remain invisible to guests.

Publishing

```
func stage(_ stage: IVSStage, shouldPublishParticipant participant: IVSParticipantInfo)
 -> Bool
```

Once connected to the stage, the SDK queries the host application to see if a particular participant should publish. This is invoked only on local participants that have permission to publish based on the provided token.

Here is a sample implementation:

```
func stage(_ stage: IVSStage, shouldPublishParticipant participant: IVSParticipantInfo)
 -> Bool {
    return true
}
```

This is for a standard video chat application where users always want to publish. They can mute and unmute their audio and video, to instantly be hidden or seen/heard. (They also can use publish/unpublish, but that is much slower. Mute/unmute is preferable for use cases where changing visibility often is desirable.)

Choosing Streams to Publish

```
func stage(_ stage: IVSStage, streamsToPublishForParticipant participant:
 IVSParticipantInfo) -> [IVSLocalStageStream]
```

When publishing, this is used to determine what audio and video streams should be published. This is covered in more detail later in Publish a Media Stream.

Updating the Strategy

The strategy is intended to be dynamic: the values returned from any of the above functions can be changed at any time. For example, if the host application does not want to publish until the end user taps a button, you could return a variable from `shouldPublishParticipant` (something like `hasUserTappedPublishButton`). When that variable changes based on an interaction by the end user, call `stage.refreshStrategy()` to signal to the SDK that it should query the strategy for the latest values, applying only things that have changed. If the SDK observes that the `shouldPublishParticipant` value has changed, it will start the publish process. If the SDK queries and all functions return the same value as before, the `refreshStrategy` call will not make any modifications to the stage.

If the return value of `shouldSubscribeToParticipant` changes from `.audioVideo` to `.audioOnly`, the video stream will be removed for all participants with changed returned values, if a video stream existed previously.

Generally, the stage uses the strategy to most efficiently apply the difference between the previous and current strategies, without the host application needing to worry about all the state required to manage it properly. Because of this, think of calling `stage.refreshStrategy()` as a cheap operation, because it does nothing unless the strategy changes.

Renderer

The `IVSStageRenderer` protocol communicates the state of the stage to the host application. Updates to the host application's UI usually can be powered entirely by the events provided by the renderer. The renderer provides the following functions:

```
func stage(_ stage: IVSStage, participantDidJoin participant: IVSParticipantInfo)
func stage(_ stage: IVSStage, participantDidLeave participant: IVSParticipantInfo)

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChange publishState:
 IVSParticipantPublishState)

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChange
 subscribeState: IVSParticipantSubscribeState)

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didAdd streams:
 [IVSStageStream])

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didRemove streams:
 [IVSStageStream])

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChangeMutedStreams
 streams: [IVSStageStream])

func stage(_ stage: IVSStage, didChange connectionState: IVSStageConnectionState,
 withError error: Error?)
```

It is not expected that the information provided by the renderer impacts the return values of the strategy. For example, the return value of `shouldSubscribeToParticipant` is not expected to change when `participant:didChangePublishState` is called. If the host application wants to subscribe to a particular participant, it should return the desired subscription type regardless of that participant's publish state. The SDK is responsible for ensuring that the desired state of the strategy is acted on at the correct time based on the state of the stage.

Note that only publishing participants trigger `participantDidJoin`, and whenever a participant stops publishing or leaves the stage session, `participantDidLeave` is triggered.

Publish a Media Stream

Local devices such as built-in microphones and cameras are discovered via `IVSDeviceDiscovery`. Here is an example of selecting the front-facing camera and default microphone, then returning them as `IVSLocalStageStreams` to be published by the SDK:

```
let devices = IVSDeviceDiscovery().listLocalDevices()

// Find the camera virtual device, choose the front source, and create a stream
let camera = devices.compactMap({ $0 as? IVSCamera }).first!
let frontSource = camera.listAvailableInputSources().first(where: { $0.position
 == .front })!
```

```
camera.setPreferredInputSource(frontSource)
let cameraStream = IVSLocalStageStream(device: camera)

// Find the microphone virtual device and create a stream
let microphone = devices.compactMap({ $0 as? IVSMicrophone }).first!
let microphoneStream = IVSLocalStageStream(device: microphone)

// Configure the audio manager to use the videoChat preset, which is optimized for bi-
directional communication, including echo cancellation.
IVSStageAudioManager.sharedInstance().setPreset(.videoChat)

// This is a function on IVSStageStrategy
func stage(_ stage: IVSStage, streamsToPublishForParticipant participant:
 IVSParticipantInfo) -> [IVSLocalStageStream] {
    return [cameraStream, microphoneStream]
}
```

Display and Remove Participants

After subscribing is completed, you will receive an array of `IVSStageStream` objects through the renderer's `didAddStreams` function. To preview or receive audio level stats about this participant, you can access the underlying `IVSDevice` object from the stream:

```
if let imageDevice = stream.device as? IVSImageDevice {
    let preview = imageDevice.previewView()
    /* attach this UIView subclass to your view */
} else if let audioDevice = stream.device as? IVSAudioDevice {
    audioDevice.setStatsCallback( { stats in
        /* process stats.peak and stats.rms */
    })
}
```

When a participant stops publishing or is unsubscribed from, the `didRemoveStreams` function is called with the streams that were removed. Host applications should use this as a signal to remove the participant's video stream from the view hierarchy.

`didRemoveStreams` is invoked for all scenarios in which a stream might be removed, including:

- The remote participant stops publishing.
- A local device unsubscribes or changes subscription from `.audioVideo` to `.audioOnly`.
- The remote participant leaves the stage.

- The local participant leaves the stage.

Because `didRemoveStreams` is invoked for all scenarios, no custom business logic is required around removing participants from the UI during remote or local leave operations.

Mute and Unmute Media Streams

`IVSLocalStageStream` objects have a `setMuted` function that controls whether the stream is muted. This function can be called on the stream before or after it is returned from the `streamsToPublishForParticipant` strategy function.

Important: If a new `IVSLocalStageStream` object instance is returned by `streamsToPublishForParticipant` after a call to `refreshStrategy`, the mute state of the new stream object is applied to the stage. Be careful when creating new `IVSLocalStageStream` instances to make sure the expected mute state is maintained.

Monitor Remote Participant Media Mute State

When a participant changes the mute state of its video or audio stream, the renderer `didChangeMutedStreams` function is invoked with an array of streams that have changed. Use the `isMuted` property on `IVSStageStream` to update your UI accordingly:

```
func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChangeMutedStreams
 streams: [IVSStageStream]) {
    streams.forEach { stream in
        /* stream.isMuted */
    }
}
```

Create a Stage Configuration

To customize the values of a stage's video configuration, use `IVSLocalStageStreamVideoConfiguration`:

```
let config = IVSLocalStageStreamVideoConfiguration()
try config.setMaxBitrate(900_000)
try config.setMinBitrate(100_000)
try config.setTargetFramerate(30)
try config.setSize(CGSize(width: 360, height: 640))
config.degradationPreference = .balanced
```

Get WebRTC Statistics

To get the latest WebRTC statistics for a publishing stream or a subscribing stream, use `requestRTCStats` on `IVSStageStream`. When a collection is completed, you will receive statistics through the `IVSStageStreamDelegate` which can be set on `IVSStageStream`. To continually collect WebRTC statistics, call this function on a `Timer`.

```
func stream(_ stream: IVSStageStream, didGenerateRTCStats stats: [String : [String :
 String]]) {
    for stat in stats {
      for member in stat.value {
         print("stat \(stat.key) has member \(member.key) with value \(member.value)")
      }
   }
}
```

Get Participant Attributes

If you specify attributes in the `CreateParticipantToken` endpoint request, you can see the attributes in `IVSParticipantInfo` properties:

```
func stage(_ stage: IVSStage, participantDidJoin participant: IVSParticipantInfo) {
    print("ID: \(participant.participantId)")
    for attribute in participant.attributes {
        print("attribute: \(attribute.key)=\(attribute.value)")
    }
}
```

Continue Session in the Background

When the app enters the background, you can continue to be in the stage while hearing remote audio, though it is not possible to continue to send your own image and audio. You will need to update your `IVSStrategy` implementation to stop publishing and subscribe to `.audioOnly` (or `.none`, if applicable):

```
func stage(_ stage: IVSStage, shouldPublishParticipant participant: IVSParticipantInfo)
 -> Bool {
    return false
}
func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType {
    return .audioOnly
}
```

Then make a call to `stage.refreshStrategy()`.

Enable/Disable Layered Encoding with Simulcast

When publishing a media stream, the SDK transmits high-quality and low-quality video streams, so remote participants can subscribe to the stream even if they have limited downlink bandwidth. Layered encoding with simulcast is on by default. You can disable it with `IVSSimulcastConfiguration`:

```
// Disable Simulcast
let config = IVSLocalStageStreamVideoConfiguration()
config.simulcast.enabled = false

let cameraStream = IVSLocalStageStream(device: camera, configuration: config)

// Other Stage implementation code
```

Broadcast the Stage to an IVS Channel

To broadcast a stage, create a separate `IVSBroadcastSession` and then follow the usual instructions for broadcasting with the SDK, described above. The `device` property on `IVSStageStream` will be either an `IVSImageDevice` or `IVSAudioDevice` as shown in the snippet above; these can be connected to the `IVSBroadcastSession.mixer` to broadcast the entire stage in a customizable layout.

Optionally, you can composite a stage and broadcast it to an IVS low-latency channel, to reach a larger audience. See Enabling Multiple Hosts on an Amazon IVS Stream in the IVS Low-Latency Streaming User Guide.

How iOS Chooses Camera Resolution and Frame Rate

The camera managed by the broadcast SDK optimizes its resolution and frame rate (frames-per-second, or FPS) to minimize heat production and energy consumption. This section explains how the resolution and frame rate are selected to help host applications optimize for their use cases.

When creating an `IVSLocalStageStream` with an `IVSCamera`, the camera is optimized for a frame rate of `IVSLocalStageStreamVideoConfiguration.targetFramerate` and a resolution of `IVSLocalStageStreamVideoConfiguration.size`. Calling `IVSLocalStageStream.setConfiguration` updates the camera with newer values.

Camera Preview

If you create a preview of an `IVSCamera` without attaching it to a `IVSBroadcastSession` or `IVSStage`, it defaults to a resolution of 1080p and a frame rate of 60 fps.

Broadcasting a Stage

When using an `IVSBroadcastSession` to broadcast an `IVSStage`, the SDK tries to optimize the camera with a resolution and frame rate that meet the criteria of both sessions.

For example, if the broadcast configuration is set to have a frame rate of 15 FPS and a resolution of 1080p, while the Stage has a frame rate of 30 FPS and a resolution of 720p, the SDK will select a camera configuration with a frame rate of 30 FPS and a resolution of 1080p. The `IVSBroadcastSession` will drop every other frame from the camera, and the `IVSStage` will scale the 1080p image down to 720p.

If a host application plans on using both `IVSBroadcastSession` and `IVSStage` together, with a camera, we recommend that the `targetFramerate` and `size` properties of the respective configurations match. A mismatch could cause the camera to reconfigure itself while capturing video, which will cause a brief delay in video-sample delivery.

If having identical values does not meet the host application's use case, creating the higher quality camera first will prevent the camera from reconfiguring itself when the lower quality session is added. For example, if you broadcast at 1080p and 30 FPS and then later join a Stage set to 720p and 30 FPS, the camera will not reconfigure itself and video will continue uninterrupted. This is because 720p is less than or equal to 1080p and 30 FPS is less than or equal to 30 FPS.

Arbitrary Frame Rates, Resolutions, and Aspect Ratios

Most camera hardware can exactly match common formats, such as 720p at 30 FPS or 1080p at 60 FPS. However, it is not possible to exactly match all formats. The broadcast SDK chooses the camera configuration based on the following rules (in priority order):

1. The width and height of the resolution are greater than or equal to the desired resolution, but within this constraint, width and height are as small as possible.
2. The frame rate is greater than or equal to the desired frame rate, but within this constraint, frame rate is as low as possible.

3. The aspect ratio matches the desired aspect ratio.
4. If there are multiple matching formats, the format with the greatest field of view is used.

Here are two examples:

- The host application is trying to broadcast in 4k at 120 FPS. The selected camera supports only 4k at 60 FPS or 1080p at 120 FPS. The selected format will be 4k at 60 FPS, because the resolution rule is higher priority than the frame-rate rule.
- An irregular resolution is requested, 1910x1070. The camera will use 1920x1080. *Be careful: choosing a resolution like 1921x1080 will causes the camera to scale up to the next available resolution (such as 2592x1944), which incurs a CPU and memory-bandwidth penalty*.

What about Android?

Android does not adjust its resolution or frame rate on the fly like iOS does, so this does not impact the Android broadcast SDK.

Known Issues and Workarounds

- Changing Bluetooth audio routes can be unpredictable. If you connect a new device mid-session, iOS may or may not automatically change the input route. Also, it is not possible to choose between multiple Bluetooth headsets that are connected at the same time. This happens in both regular broadcast and stage sessions.

  Workaround: If you plan to use a Bluetooth headset, connect it before starting the broadcast or stage and leave it connected throughout the session.
- Participants using an iPhone 14, iPhone 14 Plus, iPhone 14 Pro, or iPhone 14 Pro Max may cause an audio echo issue for other participants.

  Workaround: Participants using the affected devices can use headphones to prevent the echo issue for other participants.
- When a participant joins with a token that is being used by another participant, the first connection is disconnected without a specific error.

  Workaround: None.
- There is a rare issue where the publisher is publishing but the publish state that subscribers receive is `inactive`.

Workaround: Try leaving and then joining the session. If the issue remains, create a new token for the publisher.

- When a participant is publishing or subscribing, it is possible to receive an error with code 1400 that indicates disconnection due to a network issue, even when the network is stable.

  Workaround: Try republishing / resubscribing.
- A rare audio-distortion issue may occur intermittently during a stage session, typically on calls of longer durations.

  Workaround: The participant with distorted audio can either leave and rejoin the session, or unpublish and republish their audio to fix the issue.

Error Handling

Fatal vs. Non-Fatal Errors

The error object has an "is fatal" boolean. This is a dictionary entry under `IVSBroadcastErrorIsFatalKey` which contains a boolean.

In general, fatal errors are related to connection to the Stages server (either a connection cannot be established or is lost and cannot be recovered). The application should re-create the stage and re-join, possibly with a new token or when the device's connectivity recovers.

Non-fatal errors generally are related to the publish/subscribe state and are handled by the SDK, which retries the publish/subscribe operation.

You can check this property:

```
let nsError = error as NSError
if nsError.userInfo[IVSBroadcastErrorIsFatalKey] as? Bool == true {
  // the error is fatal
}
```

Join Errors

Malformed Token

This happens when the stage token is malformed.

The SDK throws a Swift exception with error code = 1000 and IVSBroadcastErrorIsFatalKey = YES.

Action: Create a valid token and retry joining.

Expired Token

This happens when the stage token is expired.

The SDK throws a Swift exception with error code = 1001 and IVSBroadcastErrorIsFatalKey = YES.

Action: Create a new token and retry joining.

Invalid or Revoked Token

This happens when the stage token is not malformed but is rejected by the Stages server. This is reported asynchronously through the application-supplied stage renderer.

The SDK calls `stage(didChange connectionState, withError error)` with error code = 1026 and IVSBroadcastErrorIsFatalKey = YES.

Action: Create a valid token and retry joining.

Network Errors for Initial Join

This happens when the SDK cannot contact the Stages server to establish a connection. This is reported asynchronously through the application-supplied stage renderer.

The SDK calls `stage(didChange connectionState, withError error)` with error code = 1300 and IVSBroadcastErrorIsFatalKey = YES.

Action: Wait for the device's connectivity to recover and retry joining.

Network Errors when Already Joined

If the device's network connection goes down, the SDK may lose its connection to Stage servers. This is reported asynchronously through the application-supplied stage renderer.

The SDK calls `stage(didChange connectionState, withError error)` with error code = 1300 and IVSBroadcastErrorIsFatalKey value = YES.

Action: Wait for the device's connectivity to recover and retry joining.

Publish/Subscribe Errors

Initial

There are several errors:

- MultihostSessionOfferCreationFailPublish (1020)
- MultihostSessionOfferCreationFailSubscribe (1021)
- MultihostSessionNoIceCandidates (1022)
- MultihostSessionStageAtCapacity (1024)
- SignallingSessionCannotRead (1201)
- SignallingSessionCannotSend (1202)
- SignallingSessionBadResponse (1203)

These are reported asynchronously through the application-supplied stage renderer.

The SDK retries the operation for a limited number of times. During retries, the publish/subscribe state is `ATTEMPTING_PUBLISH` / `ATTEMPTING_SUBSCRIBE`. If the retry attempts succeed, the state changes to `PUBLISHED` / `SUBSCRIBED`.

The SDK calls `IVSErrorSourceDelegate:didEmitError` with the relevant error code and IVSBroadcastErrorIsFatalKey = NO.

Action: No action is needed, as the SDK retries automatically. Optionally, the application can refresh the strategy to force more retries.

Already Established, Then Fail

A publish or subscribe can fail after it is established, most likely due to a network error. The error code for a "peer connection lost due to network error" is 1400.

This is reported asynchronously through the application-supplied stage renderer.

The SDK retries the publish/subscribe operation. During retries, the publish/subscribe state is `ATTEMPTING_PUBLISH` / `ATTEMPTING_SUBSCRIBE`. If the retry attempts succeed, the state changes to `PUBLISHED` / `SUBSCRIBED`.

The SDK calls `didEmitError` with error code = 1400 and IVSBroadcastErrorIsFatalKey = NO.

Action: No action is needed, as the SDK retries automatically. Optionally, the application can refresh the strategy to force more retries. In the event of total connectivity loss, it's likely that the connection to Stages will fail too.

IVS Broadcast SDK: Custom Image Sources (Real-Time Streaming)

Custom image-input sources allow an application to provide its own image input to the broadcast SDK, instead of being limited to the preset cameras. A custom image source can be as simple as a semi-transparent watermark or static "be right back" scene, or it can allow the app to do additional custom processing like adding beauty filters to the camera.

When you use a custom image-input source for custom control of the camera (such as using beauty-filter libraries that require camera access), the broadcast SDK is no longer responsible for managing the camera. Instead, the application is responsible for handling the camera's lifecycle correctly. See official platform documentation on how your application should manage the camera.

Android

After you create a `DeviceDiscovery` session, create an image-input source:

```
CustomImageSource imageSource = deviceDiscovery.createImageInputSource(new
 BroadcastConfiguration.Vec2(1280, 720));
```

This method returns a `CustomImageSource`, which is an image source backed by a standard Android Surface. The sublcass `SurfaceSource` can be resized and rotated. You also can create an `ImagePreviewView` to display a preview of its contents.

To retrieve the underlying `Surface`:

```
Surface surface = surfaceSource.getInputSurface();
```

This `Surface` can be used as the output buffer for image producers like Camera2, OpenGL ES, and other libraries. The simplest use case is directly drawing a static bitmap or color into the Surface's Canvas. However, many libraries (such as beauty-filter libraries) provide a method that allows an application to specify an external `Surface` for rendering. You can use such a method to pass this `Surface` to the filter library, which allows the library to output processed frames for the broadcast session to stream.

This `CustomImageSource` can be wrapped in a `LocalStageStream` and returned by the `StageStrategy` to publish to a `Stage`.

iOS

After you create a `DeviceDiscovery` session, create an image-input source:

```
let customSource = broadcastSession.createImageSource(withName: "customSourceName")
```

This method returns an `IVSCustomImageSource`, which is an image source that allows the application to submit `CMSampleBuffers` manually. For supported pixel formats, see the iOS Broadcast SDK Reference; a link to the most current version is in the Amazon IVS Release Notes for the latest broadcast SDK release.

Samples submitted to the custom source will be streamed to the Stage:

```
customSource.onSampleBuffer(sampleBuffer)
```

For streaming video, use this method in a callback. For example, if you're using the camera, then every time a new sample buffer is received from an `AVCaptureSession`, the application can forward the sample buffer to the custom image source. If desired, the application can apply further processing (like a beauty filter) before submitting the sample to the custom image source.

The `IVSCustomImageSource` can be wrapped in an `IVSLocalStageStream` and returned by the `IVSStageStrategy` to publish to a `Stage`.

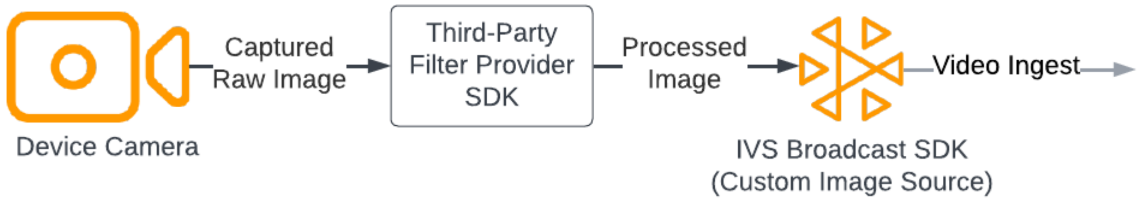
IVS Broadcast SDK: Third-Party Camera Filters (Real-Time Streaming)

This guide assumes you are already familiar with custom image sources as well as integrating the IVS real-time streaming broadcast SDK into your application.

Camera filters enable live-stream creators to augment or alter their facial or background appearance. This potentially can increase viewer engagement, attract viewers, and enhance the live-streaming experience.

Integrating Third-Party Camera Filters

You can integrate third-party camera filter SDKs with the IVS broadcast SDK by feeding the filter SDK's output to a custom image input source. A custom image-input source allows an application to provide its own image input to the Broadcast SDK. A third-party filter provider's SDK may manage the camera's lifecycle to process images from the camera, apply a filter effect, and output it in a format that can be passed to a custom image source.


Device Camera
Captured Raw Image
Third-Party Filter Provider SDK
Processed Image
Video Ingest
IVS Broadcast SDK (Custom Image Source)

Consult your third-party filter provider's documentation for built-in methods to convert a camera frame, with the filter effect, applied to a format that can be passed to a custom image-input source. The process varies, depending on which version of the IVS broadcast SDK is used:

- **Web** — The filter provider must be able to render its output to a canvas element. The captureStream method can then be used to return a MediaStream of the canvas's contents. The MediaStream can then be converted to an instance of a LocalStageStream and published to a Stage.
- **Android** — The filter provider's SDK can either render a frame to an Android `Surface` provided by the IVS broadcast SDK or convert the frame to a bitmap. If using a bitmap, it can then be rendered to the underlying `Surface` provided by the custom image source, by unlocking and writing to a canvas.
- **iOS** — A third-party filter provider's SDK must provide a camera frame with a filter effect applied as a `CMSampleBuffer`. Refer to your third-party filter vendor SDK's documentation for information on how to get a `CMSampleBuffer` as the final output after a camera image is processed.

BytePlus

Android

Install and Set Up the BytePlus Effects SDK

See the BytePlus Android Access Guide for details on how to install, initialize, and set up the BytePlus Effects SDK.

Set Up the Custom Image Source

After initializing the SDK, feed processed camera frames with a filter effect applied to a custom-image input source. To do that, create an instance of a `DeviceDiscovery` object and create a custom image source. Note that when you use a custom image input source for custom control of the camera, the broadcast SDK is no longer responsible for managing the camera. Instead, the application is responsible for handling the camera's lifecycle correctly.

Java

```
var deviceDiscovery = DeviceDiscovery(applicationContext)
var customSource = deviceDiscovery.createImageInputSource( BroadcastConfiguration.Vec2(
720F, 1280F
))
var surface: Surface = customSource.inputSurface
var filterStream = ImageLocalStageStream(customSource)
```

Convert Output to a Bitmap and Feed to Custom Image Input Source

To enable camera frames with a filter effect applied from the BytePlus Effect SDK to be forwarded directly to the IVS broadcast SDK, convert the BytePlus Effects SDK's output of a texture to a bitmap. When an image is processed, the `onDrawFrame()` method is invoked by the SDK. The `onDrawFrame()` method is a public method of Android's GLSurfaceView.Renderer interface. In the Android sample app provided by BytePlus, this method is called on every camera frame; it outputs a texture. Concurrently, you can supplement the `onDrawFrame()` method with logic to convert this texture to a bitmap and feed it to a custom image input source. As shown in the following code sample, use the `transferTextureToBitmap` method provided by the BytePlus SDK to do this conversion. This method is provided by the com.bytedance.labcv.core.util.ImageUtil library from the BytePlus Effects SDK, as shown in the following code sample.You can then render to the underlying Android `Surface` of a `CustomImageSource` by writing the resulting bitmap to a Surface's canvas. Many successive invocations of `onDrawFrame()` results in a sequence of bitmaps, and when combined, creates a stream of video.

Java

```
import com.bytedance.labcv.core.util.ImageUtil;
...
protected ImageUtil imageUtility;
...
```

```
@Override
public void onDrawFrame(GL10 gl10) {
  ...
  // Convert BytePlus output to a Bitmap
  Bitmap outputBt = imageUtility.transferTextureToBitmap(output.getTexture(),ByteEffect

  Constants.TextureFormat.Texture2D,output.getWidth(), output.getHeight());

  canvas = surface.lockCanvas(null);
  canvas.drawBitmap(outputBt, 0f, 0f, null);
  surface.unlockCanvasAndPost(canvas);
```

DeepAR

Android

See the Android Integration Guide from DeepAR for details on how to integrate the DeepAR SDK with the Android IVS broadcast SDK.

iOS

See the iOS Integration Guide from DeepAR for details on how to integrate the DeepAR SDK with the iOS IVS broadcast SDK.

Snap

Web

This section assumes you are already familiar with publishing and subscribing to video using the Web Broadcast SDK.

To integrate Snap's Camera Kit SDK with the IVS real-time streaming Web broadcast SDK, you need to:

1. Install the Camera Kit SDK and Webpack. (Our example uses Webpack as the bundler, but you can use any bundler of your choice.)
2. Create `index.html`.
3. Add setup elements.
4. Display and set up participants.
5. Display connected cameras and microphones.

6. Create a Camera Kit session.
7. Fetch and apply a Lens.
8. Render the output from a Camera Kit session to a canvas.
9. Provide Camera Kit with a media source for rendering and publish a `LocalStageStream`.
10. Create a Webpack config file.

Each of these steps is described below.

Install the Camera Kit SDK and Webpack

```
npm i @snap/camera-kit webpack webpack-cli
```

Create index.html

Next, create the HTML boilerplate and import the Web broadcast SDK as a script tag. In the following code, be sure to replace `<SDK version>` with the broadcast SDK version that you are using.

JavaScript

```
<!--
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */
-->
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <title>Amazon IVS Real-Time Streaming Web Sample (HTML and JavaScript)</title>

  <!-- Fonts and Styling -->
  <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Roboto:300,300italic,700,700italic" />
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/
normalize.css" />
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/milligram/1.4.1/
milligram.css" />
```

```
  <link rel="stylesheet" href="./index.css" />

  <!-- Stages in Broadcast SDK -->
  <script src="https://web-broadcast.live-video.net/<SDK version>/amazon-ivs-web-
broadcast.js"></script>
</head>

<body>
  <!-- Introduction -->
  <header>
    <h1>Amazon IVS Real-Time Streaming Web Sample (HTML and JavaScript)</h1>

    <p>This sample is used to demonstrate basic HTML / JS usage. <b><a href="https://
docs.aws.amazon.com/ivs/latest/userguide/multiple-hosts.html">Use the AWS CLI</
a></b> to create a <b>Stage</b> and a corresponding <b>ParticipantToken</b>.
 Multiple participants can load this page and put in their own tokens. You can <b><a
 href="https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-guides/stages#glossary"
 target="_blank">read more about stages in our public docs.</a></b></p>
  </header>
  <hr />

  <!-- Setup Controls -->

  <!-- Local Participant -->

  <hr style="margin-top: 5rem"/>

  <!-- Remote Participants -->

  <!-- Load all Desired Scripts -->

</body>

</html>
```

Add Setup Elements

Create the HTML for selecting a camera and microphone and specifying a participant token:

JavaScript

```
<!-- Setup Controls -->
  <div class="row">
    <div class="column">
```

```
      <label for="video-devices">Select Camera</label>
      <select disabled id="video-devices">
        <option selected disabled>Choose Option</option>
      </select>
    </div>
    <div class="column">
      <label for="audio-devices">Select Microphone</label>
      <select disabled id="audio-devices">
        <option selected disabled>Choose Option</option>
      </select>
    </div>
    <div class="column">
      <label for="token">Participant Token</label>
      <input type="text" id="token" name="token" />
    </div>
    <div class="column" style="display: flex; margin-top: 1.5rem">
      <button class="button" style="margin: auto; width: 100%" id="join-button">Join
 Stage</button>
    </div>
    <div class="column" style="display: flex; margin-top: 1.5rem">
      <button class="button" style="margin: auto; width: 100%" id="leave-button">Leave
 Stage</button>
    </div>
  </div>
```

Add additional HTML beneath that to display camera feeds from local and remote participants:

JavaScript

```
 <!-- Local Participant -->
<div class="row local-container">
    <canvas id="canvas"></canvas>

    <div class="column" id="local-media"></div>
    <div class="static-controls hidden" id="local-controls">
      <button class="button" id="mic-control">Mute Mic</button>
      <button class="button" id="camera-control">Mute Camera</button>
    </div>
  </div>


  <hr style="margin-top: 5rem"/>

  <!-- Remote Participants -->
  <div class="row">
    <div id="remote-media"></div>
  </div>
```

Load additional logic, including helper methods for setting up the camera and the bundled JavaScript file. (Later in this section, you will create these JavaScript files and bundle them into a single file, so you can import Camera Kit as a module. The bundled JavaScript file will contain the logic for setting up Camera Kit, applying a Lens, and publishing the camera feed with a Lens applied to a stage.)

JavaScript

```
<!-- Load all Desired Scripts -->
  <script src="./helpers.js"></script>
  <script src="./media-devices.js"></script>
  <!-- <script type="module" src="./stages-simple.js"></script> -->
  <script src="./dist/bundle.js"></script>
```

Display and Set Up Participants

Next, create `helpers.js`, which contains helper methods that you will use to display and set up participants:

JavaScript

```
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */

function setupParticipant({ isLocal, id }) {
  const groupId = isLocal ? 'local-media' : 'remote-media';
  const groupContainer = document.getElementById(groupId);

  const participantContainerId = isLocal ? 'local' : id;
  const participantContainer = createContainer(participantContainerId);
  const videoEl = createVideoEl(participantContainerId);

  participantContainer.appendChild(videoEl);
  groupContainer.appendChild(participantContainer);

  return videoEl;
}
```

```
function teardownParticipant({ isLocal, id }) {
  const groupId = isLocal ? 'local-media' : 'remote-media';
  const groupContainer = document.getElementById(groupId);
  const participantContainerId = isLocal ? 'local' : id;

  const participantDiv = document.getElementById(
    participantContainerId + '-container'
  );
  if (!participantDiv) {
    return;
  }
  groupContainer.removeChild(participantDiv);
}

function createVideoEl(id) {
  const videoEl = document.createElement('video');
  videoEl.id = id;
  videoEl.autoplay = true;
  videoEl.playsInline = true;
  videoEl.srcObject = new MediaStream();
  return videoEl;
}

function createContainer(id) {
  const participantContainer = document.createElement('div');
  participantContainer.classList = 'participant-container';
  participantContainer.id = id + '-container';

  return participantContainer;
}
```

Display Connected Cameras and Microphones

Next, create `media-devices.js`, which contains helper methods for displaying cameras and microphones connected to your device:

JavaScript

```
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */

/**
 * Returns an initial list of devices populated on the page selects
 */
async function initializeDeviceSelect() {
  const videoSelectEl = document.getElementById('video-devices');
  videoSelectEl.disabled = false;

  const { videoDevices, audioDevices } = await getDevices();
  videoDevices.forEach((device, index) => {
    videoSelectEl.options[index] = new Option(device.label, device.deviceId);
  });

  const audioSelectEl = document.getElementById('audio-devices');

  audioSelectEl.disabled = false;
  audioDevices.forEach((device, index) => {
    audioSelectEl.options[index] = new Option(device.label, device.deviceId);
  });
}

/**
 * Returns all devices available on the current device
 */
async function getDevices() {
  // Prevents issues on Safari/FF so devices are not blank
  await navigator.mediaDevices.getUserMedia({ video: true, audio: true });

  const devices = await navigator.mediaDevices.enumerateDevices();
  // Get all video devices
  const videoDevices = devices.filter((d) => d.kind === 'videoinput');
  if (!videoDevices.length) {
    console.error('No video devices found.');
  }

  // Get all audio devices
  const audioDevices = devices.filter((d) => d.kind === 'audioinput');
  if (!audioDevices.length) {
    console.error('No audio devices found.');
  }

  return { videoDevices, audioDevices };
}

async function getCamera(deviceId) {
  // Use Max Width and Height
  return navigator.mediaDevices.getUserMedia({
    video: {
      deviceId: deviceId ? { exact: deviceId } : null,
    },
    audio: false,
  });
}

async function getMic(deviceId) {
  return navigator.mediaDevices.getUserMedia({
    video: false,
    audio: {
      deviceId: deviceId ? { exact: deviceId } : null,
    },
  });
}
```

Create a Camera Kit Session

Create `stages.js`, which contains the logic for applying a Lens to the camera feed and publishing the feed to a stage. In the first part of this file, we import the broadcast SDK and Camera Kit Web SDK and initialize the variables we will use with each SDK. We create a Camera Kit session by calling `createSession` after bootstrapping the Camera Kit Web SDK. Note that a canvas element object is passed to a session; this tells Camera Kit to render into that canvas.

Java

```
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */

// All helpers are expose on 'media-devices.js' and 'dom.js'
// const { setupParticipant } = window;
// const { initializeDeviceSelect, getCamera, getMic } = window;
// require('./helpers.js');
// require('./media-devices.js');

const {
  Stage,
  LocalStageStream,
  SubscribeType,
  StageEvents,
  ConnectionState,
  StreamType,
} = IVSBroadcastClient;
```

```
import {
  bootstrapCameraKit,
  createMediaStreamSource,
  Transform2D,
} from '@snap/camera-kit';

let cameraButton = document.getElementById('camera-control');
let micButton = document.getElementById('mic-control');
let joinButton = document.getElementById('join-button');
let leaveButton = document.getElementById('leave-button');

let controls = document.getElementById('local-controls');
let videoDevicesList = document.getElementById('video-devices');
let audioDevicesList = document.getElementById('audio-devices');

// Stage management
let stage;
let joining = false;
let connected = false;
let localCamera;
let localMic;
let cameraStageStream;
let micStageStream;

const liveRenderTarget = document.getElementById('canvas');

const init = async () => {
  await initializeDeviceSelect();

  const cameraKit = await bootstrapCameraKit({
    apiToken: INSERT_API_TOKEN_HERE,
  });

  const session = await cameraKit.createSession({ liveRenderTarget });
```

Fetch and Apply a Lens

To fetch your Lenses, insert your Lens Group ID, which can be found in the Camera Kit Developer Portal. In this example, we keep it simple by applying the first Lens in the Lens array that is returned.

JavaScript

```
const { lenses } = await cameraKit.lensRepository.loadLensGroups([
    INSERT_LENS_GROUP_ID_HERE,
  ]);

  session.applyLens(lenses[0]);
```

Render the Output from a Camera Kit Session to a Canvas

Use the captureStream method to return a `MediaStream` of the canvas's contents. The canvas will contain a video stream of the camera feed with a Lens applied. Also, add event listeners for buttons to mute the camera and microphone as well as event listeners for joining and leaving a stage. In the event listener for joining a stage, we pass in a Camera Kit session and the `MediaStream` from the canvas so it can be published to a stage.

JavaScript

```
const snapStream = liveRenderTarget.captureStream();

  cameraButton.addEventListener('click', () => {
    const isMuted = !cameraStageStream.isMuted;
    cameraStageStream.setMuted(isMuted);
    cameraButton.innerText = isMuted ? 'Show Camera' : 'Hide Camera';
  });

  micButton.addEventListener('click', () => {
    const isMuted = !micStageStream.isMuted;
    micStageStream.setMuted(isMuted);
    micButton.innerText = isMuted ? 'Unmute Mic' : 'Mute Mic';
  });

  joinButton.addEventListener('click', () => {
    joinStage(session, snapStream);
  });

  leaveButton.addEventListener('click', () => {
    leaveStage();
  });
};
```

Provide Camera Kit with a Media Source for Rendering and Publish a LocalStageStream

To publish a video stream with a Lens applied, create a function called `setCameraKitSource` to pass in the `MediaStream` captured from the canvas earlier. The `MediaStream` from the canvas isn't doing anything at the moment because we have not incorporated our local camera feed yet. We can incorporate our local camera feed by calling the `getCamera` helper method and assigning it to `localCamera` . We can then pass in our local camera feed (via `localCamera`) and the session object to `setCameraKitSource`. The `setCameraKitSource` function converts our local camera feed to a source of media for CameraKit by calling `createMediaStreamSource`. The media source for `CameraKit` is then transformed to mirror the front-facing camera. The Lens effect is then applied to the media source and rendered to the output canvas by calling `session.play()`.

With Lens now applied to the `MediaStream` captured from the canvas, we can then proceed to publishing it to a stage. We do that by creating a `LocalStageStream` with the video tracks from the `MediaStream`. An instance of `LocalStageStream` can then be passed in to a `StageStrategy` to be published.

JavaScript

```
async function setCameraKitSource(session, mediaStream) {
  const source = createMediaStreamSource(mediaStream);
  await session.setSource(source);
  source.setTransform(Transform2D.MirrorX);
  session.play();
}

const joinStage = async (session, snapStream) => {
  if (connected || joining) {
    return;
  }
  joining = true;

  const token = document.getElementById('token').value;

  if (!token) {
    window.alert('Please enter a participant token');
    joining = false;
    return;
  }

  // Retrieve the User Media currently set on the page
  localCamera = await getCamera(videoDevicesList.value);
  localMic = await getMic(audioDevicesList.value);
  await setCameraKitSource(session, localCamera);
  // Create StageStreams for Audio and Video
  // cameraStageStream = new LocalStageStream(localCamera.getVideoTracks()[0]);
  cameraStageStream = new LocalStageStream(snapStream.getVideoTracks()[0]);
  micStageStream = new LocalStageStream(localMic.getAudioTracks()[0]);

  const strategy = {
    stageStreamsToPublish() {
      return [cameraStageStream, micStageStream];
    },
    shouldPublishParticipant() {
      return true;
    },
    shouldSubscribeToParticipant() {
      return SubscribeType.AUDIO_VIDEO;
    },
  };
```

The remaining code below is for creating and managing our stage:

JavaScript

```
stage = new Stage(token, strategy);

  // Other available events:
  // https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-guides/stages#events

  stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {
    connected = state === ConnectionState.CONNECTED;

    if (connected) {
      joining = false;
      controls.classList.remove('hidden');
    } else {
      controls.classList.add('hidden');
    }
  });

  stage.on(StageEvents.STAGE_PARTICIPANT_JOINED, (participant) => {
    console.log('Participant Joined:', participant);
  });

  stage.on(
    StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED,
    (participant, streams) => {
      console.log('Participant Media Added: ', participant, streams);

      let streamsToDisplay = streams;

      if (participant.isLocal) {
        // Ensure to exclude local audio streams, otherwise echo will occur
        streamsToDisplay = streams.filter(
          (stream) => stream.streamType === StreamType.VIDEO
        );
      }

      const videoEl = setupParticipant(participant);
      streamsToDisplay.forEach((stream) =>
        videoEl.srcObject.addTrack(stream.mediaStreamTrack)
      );
    }
  );

  stage.on(StageEvents.STAGE_PARTICIPANT_LEFT, (participant) => {
    console.log('Participant Left: ', participant);
    teardownParticipant(participant);
  });

  try {
    await stage.join();
  } catch (err) {
    joining = false;
    connected = false;
    console.error(err.message);
  }
};

const leaveStage = async () => {
  stage.leave();

  joining = false;
  connected = false;

  cameraButton.innerText = 'Hide Camera';
  micButton.innerText = 'Mute Mic';
  controls.classList.add('hidden');
};
```

```
init();
```

Create a Webpack Config File

Create `webpack.config.js` and add the following code. This bundles the logic above so that you can use the import statement to use Camera Kit.

JavaScript

```
const path = require('path');
module.exports = {
  entry: ['./stage.js'],
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

Finally, run `npm run build` to bundle your JavaScript as defined in the Webpack config file. You can then serve HTML and JavaScript from a web server. For example, you can use Python's HTTP server and open localhost:8000 to see the result:

```
# Run this from the command line and the directory containing index.html
python3 -m http.server -d ./
```

Android

To integrate Snap's Camera Kit SDK with the IVS Android broadcast SDK, you must install the Camera Kit SDK, initialize a Camera Kit session, apply a Lens and feed the Camera Kit session's output to the custom-image input source.

To install the Camera Kit SDK, add the following to your module's `build.gradle` file. Replace `$cameraKitVersion` with the latest Camera Kit SDK version.

Java

```
implementation "com.snap.camerakit:camerakit:$cameraKitVersion"
```

Initialize and obtain a `cameraKitSession`. Camera Kit also provides a convenient wrapper for Android's CameraX APIs, so you don't have to write complicated logic to use CameraX with Camera

Kit. You can use the `CameraXImageProcessorSource` object as a Source for ImageProcessor, which allows you to start camera-preview streaming frames.

Java

```
protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        // Camera Kit support implementation of ImageProcessor that is backed by
CameraX library:
        // https://developer.android.com/training/camerax
        CameraXImageProcessorSource imageProcessorSource = new
CameraXImageProcessorSource(
            this /*context*/, this /*lifecycleOwner*/
        );
        imageProcessorSource.startPreview(true /*cameraFacingFront*/);

        cameraKitSession = Sessions.newBuilder(this)
                .imageProcessorSource(imageProcessorSource)
                .attachTo(findViewById(R.id.camerakit_stub))
                .build();
    }
```

Fetch and Apply Lenses

You can configure Lenses and their ordering in the carousel on the Camera Kit Developer Portal:

Java

```
// Fetch lenses from repository and apply them
 // Replace LENS_GROUP_ID with Lens Group ID from https://camera-kit.snapchat.com
cameraKitSession.getLenses().getRepository().get(new Available(LENS_GROUP_ID),
 available -> {
     Log.d(TAG, "Available lenses: " + available);
     Lenses.whenHasFirst(available, lens ->
 cameraKitSession.getLenses().getProcessor().apply(lens, result -> {
          Log.d(TAG,  "Apply lens [" + lens + "] success: " + result);
      }));
});
```

To broadcast, send processed frames to the underlying `Surface` of a custom image source. Use a `DeviceDiscovery` object and create a `CustomImageSource` to return a `SurfaceSource`. You can then render the output from a `CameraKit` session to the underlying `Surface` provided by the `SurfaceSource`.

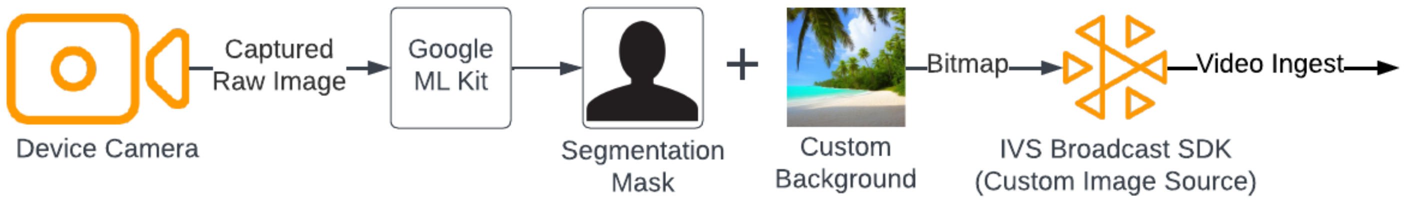
Java

```
val publishStreams = ArrayList<LocalStageStream>()

val deviceDiscovery = DeviceDiscovery(applicationContext)
val customSource =
 deviceDiscovery.createImageInputSource(BroadcastConfiguration.Vec2(720f, 1280f))

cameraKitSession.processor.connectOutput(outputFrom(customSource.inputSurface))
val customStream = ImageLocalStageStream(customSource)

// After rendering the output from a Camera Kit session to the Surface, you can
// then return it as a LocalStageStream to be published by the Broadcast SDK
val customStream: ImageLocalStageStream = ImageLocalStageStream(surfaceSource)
publishStreams.add(customStream)

@Override
fun stageStreamsToPublishForParticipant(stage: Stage, participantInfo:
 ParticipantInfo): List<LocalStageStream> = publishStreams
```

Background Replacement

Background replacement is a type of camera filter that enables live-stream creators to change their backgrounds. As shown in the following diagram, replacing your background involves:

1. Getting a camera image from the live camera feed.
2. Segmenting it into foreground and background components using Google ML Kit.
3. Combining the resulting segmentation mask with a custom background image.
4. Passing it to a Custom Image Source for broadcast.



Web

This section assumes you are already familiar with publishing and subscribing to video using the Web Broadcast SDK.

To replace the background of a live stream with a custom image, use the selfie segmentation model with MediaPipe Image Segmenter. This is a machine-learning model that identifies which pixels in the video frame are in the foreground or background. You can then use the results from the model to replace the background of a live stream, by copying foreground pixels from the video feed to a custom image representing the new background.

To integrate background replacement with the IVS real-time streaming Web broadcast SDK, you need to:

1. Install MediaPipe and Webpack. (Our example uses Webpack as the bundler, but you can use any bundler of your choice.)
2. Create `index.html`.
3. Add media elements.
4. Add a script tag.
5. Create `app.js`.
6. Load a custom background image.
7. Create an instance of `ImageSegmenter`.
8. Render the video feed to a canvas.
9. Create background replacement logic.
10. Create Webpack config File.
11. Bundle Your JavaScript file.

Install MediaPipe and Webpack

To start, install the `@mediapipe/tasks-vision` and `webpack` npm packages. The example below uses Webpack as a JavaScript bundler; you can use a different bundler if preferred.

JavaScript

```
npm i @mediapipe/tasks-vision webpack webpack-cli
```

Make sure to also update your `package.json` to specify `webpack` as your build script:

JavaScript

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack"
  },
```

Create index.html

Next, create the HTML boilerplate and import the Web broadcast SDK as a script tag. In the following code, be sure to replace `<SDK version>` with the broadcast SDK version that you are using.

JavaScript

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <!-- Import the SDK -->
  <script src="https://web-broadcast.live-video.net/<SDK version>/amazon-ivs-web-
broadcast.js"></script>
</head>

<body>

</body>
</html>
```

Add Media Elements

Next, add a video element and two canvas elements within the body tag. The video element will contain your live camera feed and will be used as input to the MediaPipe Image Segmenter. The first canvas element will be used to render a preview of the feed that will be broadcast. The second canvas element will be used to render the custom image that will be used as a background. Since the second canvas with the custom image is used only as a source to programmatically copy pixels from it to the final canvas, it is hidden from view.

JavaScript

```
<div class="row local-container">
      <video id="webcam" autoplay style="display: none"></video>
    </div>
    <div class="row local-container">
      <canvas id="canvas" width="640px" height="480px"></canvas>

      <div class="column" id="local-media"></div>
      <div class="static-controls hidden" id="local-controls">
        <button class="button" id="mic-control">Mute Mic</button>
        <button class="button" id="camera-control">Mute Camera</button>
      </div>
    </div>
    <div class="row local-container">
      <canvas id="background" width="640px" height="480px" style="display: none"></
canvas>
    </div>
```

Add a Script Tag

Add a script tag to load a bundled JavaScript file that will contain the code to do the background replacement and publish it to a stage:

```
<script src="./dist/bundle.js"></script>
```

Create app.js

Next, create a JavaScript file to get the element objects for the canvas and video elements that were created in the HTML page. Import the `ImageSegmenter` and `FilesetResolver` modules. The `ImageSegmenter` module will be used to perform the segmentation task.

JavaScript

```
const canvasElement = document.getElementById("canvas");
const background = document.getElementById("background");
const canvasCtx = canvasElement.getContext("2d");
const backgroundCtx = background.getContext("2d");
const video = document.getElementById("webcam");

import { ImageSegmenter, FilesetResolver } from "@mediapipe/tasks-vision";
```

Next, create a function called `init()` to retrieve the MediaStream from the user's camera and invoke a callback function each time a camera frame finishes loading. Add event listeners for the buttons to join and leave a stage.

Note that when joining a stage, we pass in a variable named `segmentationStream`. This is a video stream that is captured from a canvas element, containing a foreground image overlaid on the custom image representing the background. Later, this custom stream will be used to create an instance of a `LocalStageStream`, which can be published to a stage.

JavaScript

```
const init = async () => {
  await initializeDeviceSelect();

  cameraButton.addEventListener("click", () => {
    const isMuted = !cameraStageStream.isMuted;
    cameraStageStream.setMuted(isMuted);
    cameraButton.innerText = isMuted ? "Show Camera" : "Hide Camera";
  });

  micButton.addEventListener("click", () => {
    const isMuted = !micStageStream.isMuted;
    micStageStream.setMuted(isMuted);
    micButton.innerText = isMuted ? "Unmute Mic" : "Mute Mic";
  });

  localCamera = await getCamera(videoDevicesList.value);
  const segmentationStream = canvasElement.captureStream();

  joinButton.addEventListener("click", () => {
    joinStage(segmentationStream);
  });

  leaveButton.addEventListener("click", () => {
    leaveStage();
  });
};
```

Load a Custom Background Image

At the bottom of the `init` function, add code to call a function named `initBackgroundCanvas`, which loads a custom image from a local file and renders it onto a canvas. We will define this function in the next step. Assign the `MediaStream` retrieved from the user's camera to the video object. Later, this video object will be passed to the Image Segmenter. Also, set a function named `renderVideoToCanvas` as the callback function to invoke whenever a video frame has finished loading. We will define this function in a later step.

JavaScript

```
initBackgroundCanvas();

  video.srcObject = localCamera;
  video.addEventListener("loadeddata", renderVideoToCanvas);
```

Let's implement the `initBackgroundCanvas` function, which loads an image from a local file. In this example, we use an image of a beach as the custom background. The canvas containing the custom image will be hidden from display, as you will merge it with the foreground pixels from the canvas element containing the camera feed.

JavaScript

```
const initBackgroundCanvas = () => {
  let img = new Image();
  img.src = "beach.jpg";

  img.onload = () => {
    backgroundCtx.clearRect(0, 0, canvas.width, canvas.height);
    backgroundCtx.drawImage(img, 0, 0);
  };
};
```

Create an Instance of ImageSegmenter

Next, create an instance of `ImageSegmenter`, which will segment the image and return the result as a mask. When creating an instance of an `ImageSegmenter`, you will use the selfie segmentation model.

JavaScript

```
const createImageSegmenter = async () => {
  const audio = await FilesetResolver.forVisionTasks("https://cdn.jsdelivr.net/npm/
@mediapipe/tasks-vision@0.10.2/wasm");

  imageSegmenter = await ImageSegmenter.createFromOptions(audio, {
    baseOptions: {
      modelAssetPath: "https://storage.googleapis.com/mediapipe-models/image_segmenter/
selfie_segmenter/float16/latest/selfie_segmenter.tflite",
      delegate: "GPU",
    },
    runningMode: "VIDEO",
    outputCategoryMask: true,
  });
};
```

Render the Video Feed to a Canvas

Next, create the function that renders the video feed to the other canvas element. We need to render the video feed to a canvas so we can extract the foreground pixels from it using the Canvas 2D API. While doing this, we also will pass a video frame to our instance of `ImageSegmenter`, using the segmentforVideo method to segment the foreground from the background in the video frame. When the segmentforVideo method returns, it invokes our custom callback function, `replaceBackground`, for doing the background replacement.

JavaScript

```
const renderVideoToCanvas = async () => {
  if (video.currentTime === lastWebcamTime) {
    window.requestAnimationFrame(renderVideoToCanvas);
    return;
  }
  lastWebcamTime = video.currentTime;
  canvasCtx.drawImage(video, 0, 0, video.videoWidth, video.videoHeight);

  if (imageSegmenter === undefined) {
    return;
  }

  let startTimeMs = performance.now();

  imageSegmenter.segmentForVideo(video, startTimeMs, replaceBackground);
};
```

Create Background Replacement Logic

Create the `replaceBackground` function, which merges the custom background image with the foreground from the camera feed to replace the background. The function first retrieves the underlying pixel data of the custom background image and the video feed from the two canvas elements created earlier. It then iterates through the mask provided by `ImageSegmenter`, which indicates which pixels are in the foreground. As it iterates through the mask, it selectively copies pixels that contain the user's camera feed to the corresponding background pixel data. Once that is done, it converts the final pixel data with the foreground copied on to the background and draws it to a Canvas.

JavaScript

```
function replaceBackground(result) {
  let imageData = canvasCtx.getImageData(0, 0, video.videoWidth,
 video.videoHeight).data;
  let backgroundData = backgroundCtx.getImageData(0, 0, video.videoWidth,
 video.videoHeight).data;
  const mask = result.categoryMask.getAsFloat32Array();
  let j = 0;

  for (let i = 0; i < mask.length; ++i) {
    const maskVal = Math.round(mask[i] * 255.0);

    j += 4;
  // Only copy pixels on to the background image if the mask indicates they are in the
 foreground
    if (maskVal < 255) {
      backgroundData[j] = imageData[j];
      backgroundData[j + 1] = imageData[j + 1];
      backgroundData[j + 2] = imageData[j + 2];
      backgroundData[j + 3] = imageData[j + 3];
    }
  }

 // Convert the pixel data to a format suitable to be drawn to a canvas
  const uint8Array = new Uint8ClampedArray(backgroundData.buffer);
  const dataNew = new ImageData(uint8Array, video.videoWidth, video.videoHeight);
  canvasCtx.putImageData(dataNew, 0, 0);
  window.requestAnimationFrame(renderVideoToCanvas);
}
```

For reference, here is the complete `app.js` file containing all the logic above:

JavaScript

```
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */

// All helpers are expose on 'media-devices.js' and 'dom.js'
const { setupParticipant } = window;

const { Stage, LocalStageStream, SubscribeType, StageEvents, ConnectionState,
 StreamType } = IVSBroadcastClient;
const canvasElement = document.getElementById("canvas");
const background = document.getElementById("background");
const canvasCtx = canvasElement.getContext("2d");
const backgroundCtx = background.getContext("2d");
const video = document.getElementById("webcam");

import { ImageSegmenter, FilesetResolver } from "@mediapipe/tasks-vision";

let cameraButton = document.getElementById("camera-control");
let micButton = document.getElementById("mic-control");
let joinButton = document.getElementById("join-button");
let leaveButton = document.getElementById("leave-button");

let controls = document.getElementById("local-controls");
let audioDevicesList = document.getElementById("audio-devices");
let videoDevicesList = document.getElementById("video-devices");

// Stage management
let stage;
let joining = false;
let connected = false;
let localCamera;
let localMic;
let cameraStageStream;
let micStageStream;
let imageSegmenter;
let lastWebcamTime = -1;

const init = async () => {
  await initializeDeviceSelect();

  cameraButton.addEventListener("click", () => {
    const isMuted = !cameraStageStream.isMuted;
    cameraStageStream.setMuted(isMuted);
    cameraButton.innerText = isMuted ? "Show Camera" : "Hide Camera";
  });

  micButton.addEventListener("click", () => {
    const isMuted = !micStageStream.isMuted;
    micStageStream.setMuted(isMuted);
    micButton.innerText = isMuted ? "Unmute Mic" : "Mute Mic";
  });

  localCamera = await getCamera(videoDevicesList.value);
  const segmentationStream = canvasElement.captureStream();

  joinButton.addEventListener("click", () => {
    joinStage(segmentationStream);
  });

  leaveButton.addEventListener("click", () => {
    leaveStage();
  });

  initBackgroundCanvas();

  video.srcObject = localCamera;
  video.addEventListener("loadeddata", renderVideoToCanvas);
};

const joinStage = async (segmentationStream) => {
  if (connected || joining) {
    return;
  }
  joining = true;

  const token = document.getElementById("token").value;

  if (!token) {
    window.alert("Please enter a participant token");
    joining = false;
    return;
  }

  // Retrieve the User Media currently set on the page
  localMic = await getMic(audioDevicesList.value);

  cameraStageStream = new LocalStageStream(segmentationStream.getVideoTracks()[0]);
  micStageStream = new LocalStageStream(localMic.getAudioTracks()[0]);

  const strategy = {
    stageStreamsToPublish() {
      return [cameraStageStream, micStageStream];
    },
    shouldPublishParticipant() {
      return true;
    },
    shouldSubscribeToParticipant() {
      return SubscribeType.AUDIO_VIDEO;
    },
  };

  stage = new Stage(token, strategy);

  // Other available events:
  // https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-guides/stages#events
  stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {
    connected = state === ConnectionState.CONNECTED;

    if (connected) {
      joining = false;
      controls.classList.remove("hidden");
    } else {
      controls.classList.add("hidden");
    }
  });

  stage.on(StageEvents.STAGE_PARTICIPANT_JOINED, (participant) => {
    console.log("Participant Joined:", participant);
  });

  stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED, (participant, streams) => {
    console.log("Participant Media Added: ", participant, streams);

    let streamsToDisplay = streams;

    if (participant.isLocal) {
      // Ensure to exclude local audio streams, otherwise echo will occur
      streamsToDisplay = streams.filter((stream) => stream.streamType ===
 StreamType.VIDEO);
    }
    const videoEl = setupParticipant(participant);
    streamsToDisplay.forEach((stream) =>
 videoEl.srcObject.addTrack(stream.mediaStreamTrack));
  });

  stage.on(StageEvents.STAGE_PARTICIPANT_LEFT, (participant) => {
    console.log("Participant Left: ", participant);
    teardownParticipant(participant);
  });

  try {
    await stage.join();
  } catch (err) {
    joining = false;
    connected = false;
    console.error(err.message);
  }
};

const leaveStage = async () => {
  stage.leave();

  joining = false;
  connected = false;

  cameraButton.innerText = "Hide Camera";
  micButton.innerText = "Mute Mic";
  controls.classList.add("hidden");
};

function replaceBackground(result) {
  let imageData = canvasCtx.getImageData(0, 0, video.videoWidth,
 video.videoHeight).data;
  let backgroundData = backgroundCtx.getImageData(0, 0, video.videoWidth,
 video.videoHeight).data;
  const mask = result.categoryMask.getAsFloat32Array();
  let j = 0;

  for (let i = 0; i < mask.length; ++i) {
    const maskVal = Math.round(mask[i] * 255.0);

    j += 4;
    if (maskVal < 255) {
      backgroundData[j] = imageData[j];
      backgroundData[j + 1] = imageData[j + 1];
      backgroundData[j + 2] = imageData[j + 2];
      backgroundData[j + 3] = imageData[j + 3];
    }
  }
  const uint8Array = new Uint8ClampedArray(backgroundData.buffer);
  const dataNew = new ImageData(uint8Array, video.videoWidth, video.videoHeight);
  canvasCtx.putImageData(dataNew, 0, 0);
  window.requestAnimationFrame(renderVideoToCanvas);
}

const createImageSegmenter = async () => {
  const audio = await FilesetResolver.forVisionTasks("https://cdn.jsdelivr.net/npm/
@mediapipe/tasks-vision@0.10.2/wasm");

  imageSegmenter = await ImageSegmenter.createFromOptions(audio, {
    baseOptions: {
      modelAssetPath: "https://storage.googleapis.com/mediapipe-models/image_segmenter/
selfie_segmenter/float16/latest/selfie_segmenter.tflite",
      delegate: "GPU",
    },
    runningMode: "VIDEO",
    outputCategoryMask: true,
  });
};

const renderVideoToCanvas = async () => {
  if (video.currentTime === lastWebcamTime) {
    window.requestAnimationFrame(renderVideoToCanvas);
    return;
  }
  lastWebcamTime = video.currentTime;
  canvasCtx.drawImage(video, 0, 0, video.videoWidth, video.videoHeight);

  if (imageSegmenter === undefined) {
    return;
  }

  let startTimeMs = performance.now();

  imageSegmenter.segmentForVideo(video, startTimeMs, replaceBackground);
};

const initBackgroundCanvas = () => {
  let img = new Image();
  img.src = "beach.jpg";

  img.onload = () => {
    backgroundCtx.clearRect(0, 0, canvas.width, canvas.height);
    backgroundCtx.drawImage(img, 0, 0);
  };
};

createImageSegmenter();
init();
```

Create a Webpack Config File

Add this configuration to your Webpack config file to bundle `app.js`, so the import calls will work:

JavaScript

```
const path = require("path");
module.exports = {
  entry: ["./app.js"],
  output: {
    filename: "bundle.js",
    path: path.resolve(__dirname, "dist"),
  },
};
```

Bundle Your JavaScript files

```
npm run build
```

Start a simple HTTP server from the directory containing `index.html` and open `localhost:8000` to see the result:

```
python3 -m http.server -d ./
```

Android

To replace the background in your live stream, you can use the selfie segmentation API of Google ML Kit. The selfie segmentation API accepts a camera image as input and returns a mask that provides a confidence score for each pixel of the image, indicating whether it was in the foreground or the background. Based on the confidence score, you can then retrieve the corresponding pixel color from either the background image or the foreground image. This process continues until all confidence scores in the mask have been examined. The result is a new array of pixel colors containing foreground pixels combined with pixels from the background image.

To integrate background replacement with the IVS real-time streaming Android broadcast SDK, you need to:

1. Install CameraX libraries and the Google ML kit.
2. Initialize boilerplate variables.
3. Create a custom image source.
4. Manage camera frames.
5. Pass camera frames to Google ML Kit.
6. Overlay camera frame foreground onto your custom background.
7. Feed the new image to a custom image source.

Install CameraX Libraries and Google ML Kit

To extract images from the live camera feed, use Android's CameraX library. To install the CameraX library and Google ML Kit, add the following to your module's `build.gradle` file. Replace `${camerax_version}` and `${google_ml_kit_version}` with the latest version of the CameraX and Google ML Kit libraries, respectively.

Java

```
implementation "com.google.mlkit:segmentation-selfie:${google_ml_kit_version}"
implementation "androidx.camera:camera-core:${camerax_version}"
implementation "androidx.camera:camera-lifecycle:${camerax_version}"
```

Import the following libraries:

Java

```
import androidx.camera.core.CameraSelector
import androidx.camera.core.ImageAnalysis
import androidx.camera.core.ImageProxy
import androidx.camera.lifecycle.ProcessCameraProvider
import com.google.mlkit.vision.segmentation.selfie.SelfieSegmenterOptions
```

Initialize Boilerplate Variables

Initialize an instance of `ImageAnalysis` and an instance of an `ExecutorService`:

Java

```
private lateinit var binding: ActivityMainBinding
private lateinit var cameraExecutor: ExecutorService
private var analysisUseCase: ImageAnalysis? = null
```

Initialize a Segmenter instance in STREAM_MODE:

Java

```
private val options =
        SelfieSegmenterOptions.Builder()
            .setDetectorMode(SelfieSegmenterOptions.STREAM_MODE)
            .build()

private val segmenter = Segmentation.getClient(options)
```

Create a Custom Image Source

In the `onCreate` method of your activity, create an instance of a `DeviceDiscovery` object and create a custom image source. The `Surface` provided by the Custom Image Source will receive the final image, with the foreground overlaid on a custom background image. You will then create an instance of a `ImageLocalStageStream` using the Custom Image Source. The instance of a `ImageLocalStageStream` (named `filterStream` in this example) can then be published to a stage. See the IVS Android Broadcast SDK Guide for instructions on setting up a stage. Finally, also create a thread that will be used to manage the camera.

Java

```
var deviceDiscovery = DeviceDiscovery(applicationContext)
var customSource = deviceDiscovery.createImageInputSource( BroadcastConfiguration.Vec2(
720F, 1280F
))
var surface: Surface = customSource.inputSurface
var filterStream = ImageLocalStageStream(customSource)

cameraExecutor = Executors.newSingleThreadExecutor()
```

Manage Camera Frames

Next, create a function to initialize the camera. This function uses the CameraX library to extract images from the live camera feed. First, you create an instance of a `ProcessCameraProvider` called `cameraProviderFuture`. This object represents a future result of obtaining a camera provider. Then you load an image from your project as a bitmap. This example uses an image of a beach as a background, but it can be any image you want.

You then add a listener to `cameraProviderFuture`. This listener is notified when the camera becomes available or if an error occurs during the process of obtaining a camera provider.

Java

```
private fun startCamera(surface: Surface) {
        val cameraProviderFuture = ProcessCameraProvider.getInstance(this)
        val imageResource = R.drawable.beach
        val bgBitmap: Bitmap = BitmapFactory.decodeResource(resources, imageResource)
        var resultBitmap: Bitmap;


        cameraProviderFuture.addListener({
            val cameraProvider: ProcessCameraProvider = cameraProviderFuture.get()


                if (mediaImage != null) {
                    val inputImage =
                        InputImage.fromMediaImage(mediaImage,
 imageProxy.imageInfo.rotationDegrees)

                            resultBitmap = overlayForeground(mask, maskWidth,
 maskHeight, inputBitmap, backgroundPixels)
                            canvas = surface.lockCanvas(null);
                            canvas.drawBitmap(resultBitmap, 0f, 0f, null)

                            surface.unlockCanvasAndPost(canvas);

                        }
                        .addOnFailureListener { exception ->
                            Log.d("App", exception.message!!)
                        }
                        .addOnCompleteListener {
                            imageProxy.close()
                        }

                }
            };

            val cameraSelector = CameraSelector.DEFAULT_FRONT_CAMERA

            try {
                // Unbind use cases before rebinding
                cameraProvider.unbindAll()

                // Bind use cases to camera
                cameraProvider.bindToLifecycle(this, cameraSelector, analysisUseCase)

            } catch(exc: Exception) {
                Log.e(TAG, "Use case binding failed", exc)
            }

        }, ContextCompat.getMainExecutor(this))
    }
```

Within the listener, create `ImageAnalysis.Builder` to access each individual frame from the live camera feed. Set the back-pressure strategy to `STRATEGY_KEEP_ONLY_LATEST`. This guarantees that only one camera frame at a time is delivered for processing. Convert each individual camera frame to a bitmap, so you can extract its pixels to later combine it with the custom background image.

Java

```
val imageAnalyzer = ImageAnalysis.Builder()
analysisUseCase = imageAnalyzer
    .setTargetResolution(Size(360, 640))
    .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
    .build()

analysisUseCase?.setAnalyzer(cameraExecutor) { imageProxy: ImageProxy ->
    val mediaImage = imageProxy.image
    val tempBitmap = imageProxy.toBitmap();
    val inputBitmap = tempBitmap.rotate(imageProxy.imageInfo.rotationDegrees.toFloat())
```

Pass Camera Frames to Google ML Kit

Next, create an `InputImage` and pass it to the instance of Segmenter for processing. An `InputImage` can be created from an `ImageProxy` provided by the instance of `ImageAnalysis`. Once an `InputImage` is provided to Segmenter, it returns a mask with confidence scores indicating the likelihood of a pixel being in the foreground or background. This mask also provides width and height properties, which you will use to create a new array containing the background pixels from the custom background image loaded earlier.

Java

```
if (mediaImage != null) {
        val inputImage =
            InputImage.fromMediaImag


segmenter.process(inputImage)
    .addOnSuccessListener { segmentationMask ->
        val mask = segmentationMask.buffer
        val maskWidth = segmentationMask.width
        val maskHeight = segmentationMask.height
        val backgroundPixels = IntArray(maskWidth * maskHeight)
        bgBitmap.getPixels(backgroundPixels, 0, maskWidth, 0, 0, maskWidth, maskHeight)
```

Overlay the Camera Frame Foreground onto Your Custom Background

With the mask containing the confidence scores, the camera frame as a bitmap, and the color pixels from the custom background image, you have everything you need to overlay the foreground onto your custom background. The `overlayForeground` function is then called with the following parameters:

Java

```
resultBitmap = overlayForeground(mask, maskWidth, maskHeight, inputBitmap,
 backgroundPixels)
```

This function iterates through the mask and checks the confidence values to determine whether to get the corresponding pixel color from the background image or the camera frame. If the confidence value indicates that a pixel in the mask is most likely in the background, it will get the corresponding pixel color from the background image; otherwise, it will get the corresponding pixel color from the camera frame to build the foreground. Once the function finishes iterating through the mask, a new bitmap is created using the new array of color pixels and returned. This new bitmap contains the foreground overlaid on the custom background.

Java

```
private fun overlayForeground(
        byteBuffer: ByteBuffer,
        maskWidth: Int,
        maskHeight: Int,
        cameraBitmap: Bitmap,
        backgroundPixels: IntArray
    ): Bitmap {
        @ColorInt val colors = IntArray(maskWidth * maskHeight)
        val cameraPixels = IntArray(maskWidth * maskHeight)

        cameraBitmap.getPixels(cameraPixels, 0, maskWidth, 0, 0, maskWidth, maskHeight)

        for (i in 0 until maskWidth * maskHeight) {
            val backgroundLikelihood: Float = 1 - byteBuffer.getFloat()

            // Apply the virtual background to the color if it's not part of the
 foreground
            if (backgroundLikelihood > 0.9) {
                // Get the corresponding pixel color from the background image
                // Set the color in the mask based on the background image pixel color
                colors[i] = backgroundPixels.get(i)
            } else {
                // Get the corresponding pixel color from the camera frame
                // Set the color in the mask based on the camera image pixel color
                colors[i] = cameraPixels.get(i)
            }
        }

        return Bitmap.createBitmap(
            colors, maskWidth, maskHeight, Bitmap.Config.ARGB_8888
        )
    }
```

Feed the New Image to a Custom Image Source

You can then write the new bitmap to the `Surface` provided by a custom image source. This will broadcast it to your stage.

Java

```
resultBitmap = overlayForeground(mask, inputBitmap, mutableBitmap, bgBitmap)
canvas = surface.lockCanvas(null);
canvas.drawBitmap(resultBitmap, 0f, 0f, null)
```

Here is the complete function for getting the camera frames, passing it to Segmenter, and overlaying it on the background:

Java

```
@androidx.annotation.OptIn(androidx.camera.core.ExperimentalGetImage::class)
    private fun startCamera(surface: Surface) {
        val cameraProviderFuture = ProcessCameraProvider.getInstance(this)
        val imageResource = R.drawable.clouds
        val bgBitmap: Bitmap = BitmapFactory.decodeResource(resources, imageResource)
        var resultBitmap: Bitmap;

        cameraProviderFuture.addListener({
            // Used to bind the lifecycle of cameras to the lifecycle owner
            val cameraProvider: ProcessCameraProvider = cameraProviderFuture.get()

            val imageAnalyzer = ImageAnalysis.Builder()
            analysisUseCase = imageAnalyzer
                .setTargetResolution(Size(720, 1280))
                .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
                .build()

            analysisUseCase!!.setAnalyzer(cameraExecutor) { imageProxy: ImageProxy ->
                val mediaImage = imageProxy.image
                val tempBitmap = imageProxy.toBitmap();
                val inputBitmap =
 tempBitmap.rotate(imageProxy.imageInfo.rotationDegrees.toFloat())

                if (mediaImage != null) {
                    val inputImage =
                        InputImage.fromMediaImage(mediaImage,
 imageProxy.imageInfo.rotationDegrees)

                    segmenter.process(inputImage)
                        .addOnSuccessListener { segmentationMask ->
                            val mask = segmentationMask.buffer
                            val maskWidth = segmentationMask.width
                            val maskHeight = segmentationMask.height
                            val backgroundPixels = IntArray(maskWidth * maskHeight)
                            bgBitmap.getPixels(backgroundPixels, 0, maskWidth, 0, 0,
maskWidth, maskHeight)

                            resultBitmap = overlayForeground(mask, maskWidth,
maskHeight, inputBitmap, backgroundPixels)
                            canvas = surface.lockCanvas(null);
                            canvas.drawBitmap(resultBitmap, 0f, 0f, null)

                            surface.unlockCanvasAndPost(canvas);

                        }
                        .addOnFailureListener { exception ->
                            Log.d("App", exception.message!!)
                        }
                        .addOnCompleteListener {
                            imageProxy.close()
                        }

                }
            };

            val cameraSelector = CameraSelector.DEFAULT_FRONT_CAMERA

            try {
                // Unbind use cases before rebinding
                cameraProvider.unbindAll()

                // Bind use cases to camera
                cameraProvider.bindToLifecycle(this, cameraSelector, analysisUseCase)

            } catch(exc: Exception) {
                Log.e(TAG, "Use case binding failed", exc)
            }

        }, ContextCompat.getMainExecutor(this))
    }
```

IVS Broadcast SDK: Mobile Audio Modes (Real-Time Streaming)

Audio quality is an important part of any real-team media experience, and there isn't a one-size-fits-all audio configuration that works best for every use case. To ensure that your users have the best experience when listening to an IVS real-time stream, our mobile SDKs provide several preset audio configurations, as well as more powerful customizations as needed.

Introduction

The IVS mobile broadcast SDKs provide a `StageAudioManager` class. This class is designed to be the single point of contact for controlling the underlying audio modes on both platforms. On Android, this controls the AudioManager, including the audio mode, audio source, content type, usage, and communication devices. On iOS, it controls the application AVAudioSession, as well as whether voiceProcessing is enabled.

Important: Do not interact with `AVAudioSession` or `AudioManager` directly while the IVS real-time broadcast SDK is active. Doing so could result in the loss of audio, or audio being recorded from or played back on the wrong device.

Before you create your first `DeviceDiscovery` or `Stage` object, the `StageAudioManager` class must be configured.

Android (Kotlin)

```
StageAudioManager.getInstance(context).setPreset(StageAudioManager.UseCasePreset.VIDEO_CHAT)
 The default value

val deviceDiscovery = DeviceDiscovery(context)
val stage = Stage(context, token, this)

// Other Stage implementation code
```

iOS (Swift)

```
IVSStageAudioManager.sharedInstance().setPreset(.videoChat) // The default value

let deviceDiscovery = IVSDeviceDiscovery()
let stage = try? IVSStage(token: token, strategy: self)

// Other Stage implementation code
```

If nothing is set on the `StageAudioManager` before initialization of a `DeviceDiscovery` or `Stage` instance, the `VideoChat` preset is applied automatically.

Audio Mode Presets

The real-time broadcast SDK provides three presets, each tailored to common use cases, as described below. For each preset, we cover five key categories that differentiate the presets from each other.

The **Volume Rocker** category refers to the type of volume (media volume or call volume) that is used or changed via the physical volume rockers on the device. Note that this impacts volume when switching audio modes. For example, suppose the device volume is set to the maximum value while using the Video Chat preset. Switching to the Subscribe Only preset causes a different volume level from the operating system, which could lead to a significant volume change on the device.

Video Chat

This is the default preset, designed for when the local device is going to have a real-time conversation with other participants.

Known issue on iOS: Using this preset and not attaching a microphone causes audio to play through the earpiece instead of the device speaker. Use this preset only in combination with a microphone.

Category	Android	iOS
Echo Cancellation	Enabled	Enabled
Volume Rocker	Call Volume	Call Volume
Microphone Selection	Limited based on the OS. USB microphones may not be available .	Limited based on the OS. USB and Bluetooth microphones may not be available. Bluetooth headsets that handle both input and output together should work; e.g., AirPods.

Category	Android	iOS
Audio Output	Any output device should work.	Limited based on the OS. Wired headsets may not be available.
Audio Quality	Medium / Low. It will sound like a phone call, not like media playback.	Medium / Low. It will sound like a phone call, not like media playback.

Subscribe Only

This preset is designed for when you plan to subscribe to other publishing participants but not publish yourself. It focuses on audio quality and supporting all available output devices.

Category	Android	iOS
Echo Cancellation	Disabled	Disabled
Volume Rocker	Media Volume	Media Volume
Microphone Selection	N/A, this preset is not designed for publishing.	N/A, this preset is not designed for publishing.
Audio Output	Any output device should work.	Any output device should work.
Audio Quality	High. Any media type should come through clearly, including music.	High. Any media type should come through clearly, including music.

Studio

This preset is designed for high quality subscribing while maintaining the ability to publish. It requires the recording and playback hardware to provide echo cancellation. A use case here would be using a USB microphone and a wired headset. The SDK will maintain the highest quality audio while relying on the physical separation of those devices from causing echo.

Category	Android	iOS
Echo Cancellation	Disabled	Disabled
Volume Rocker	Media Volume in most cases. Call Volume when a Bluetooth microphone is connected.	Media Volume
Microphone Selection	Any microphone should work.	Any microphone should work.
Audio Output	Any output device should work.	Any output device should work.
Audio Quality	High. Both sides should be able to send music and hear it clearly on the other side. When a Bluetooth headset is connected, audio quality will drop due to Bluetooth SCO mode being enabled.	High. Both sides should be able to send music and hear it clearly on the other side. When a Bluetooth headset is connected, audio quality may drop due to Bluetooth SCO mode being enabled, depending on the headset.

Advanced Use Cases

Beyond the presets, both the iOS and Android real-time streaming broadcast SDKs allow configuring the underlying platform audio modes:

- On Android, set the AudioSource, Usage, and ContentType.
- On iOS, use AVAudioSession.Category, AVAudioSession.CategoryOptions, AVAudioSession.Mode, and the ability to toggle if voice processing is enabled or not while publishing.

Note: When using these audio SDK methods, it is possible to incorrectly configure the underlying audio session. For example, using the `.allowBluetooth` option on iOS in combination with the `.playback` category creates an invalid audio configuration and the SDK cannot record or play back audio. These methods are designed to be used only when an application has specific audio-session requirements that have been validated.

Android (Kotlin)

```
// This would act similar to the Subscribe Only preset, but it uses a different
 ContentType.
StageAudioManager.getInstance(context)
    .setConfiguration(StageAudioManager.Source.GENERIC,
                      StageAudioManager.ContentType.MOVIE,
                      StageAudioManager.Usage.MEDIA);

val stage = Stage(context, token, this)

// Other Stage implementation code
```

iOS (Swift)

```
// This would act similar to the Subscribe Only preset, but it uses a different mode
 and options.
IVSStageAudioManager.sharedInstance()
    .setCategory(.playback,
                 options: [.duckOthers, .mixWithOthers],
                 mode: .default)

let stage = try? IVSStage(token: token, strategy: self)

// Other Stage implementation code
```

iOS Echo Cancellation

Echo cancellation on iOS can be independently controlled via `IVSStageAudioManager` as well using its `echoCancellationEnabled` method. This method controls whether voice processing is enabled on the input and output nodes of the underlying `AVAudioEngine` used by the SDK. It is important to understand the effect of changing this property manually:

- The `AVAudioEngine` property is honored only if the SDK's microphone is active; this is necessary due to the iOS requirement that voice processing be enabled on both the input and output nodes simultaneously. Normally this is done by using the microphone returned by `IVSDeviceDiscovery` to create an `IVSLocalStageStream` to publish. Alternately, the microphone can be enabled, without being used to publish, by attaching an

IVSAudioDeviceStatsCallback to the microphone itself. This alternate approach is useful if echo cancellation is needed while using a custom audio-source-based microphone instead of the IVS SDK's microphone.

- Enabling the AVAudioEngine property requires a mode of .videoChat or .voiceChat. Requesting a different mode causes iOS's underlying audio framework to fight the SDK, causing audio loss.
- Enabling AVAudioEngine automatically enables the .allowBluetooth option.

Behaviors can differ depending on the device and iOS version.

iOS Custom Audio Sources

Custom audio sources can be used with the SDK by using IVSDeviceDiscovery.createAudioSource. When connecting to a Stage, the IVS real-time streaming broadcast SDK still manages an internal AVAudioEngine instance for audio playback, even if the SDK's microphone is not used. As a result, the values provided to IVSStageAudioManager must be compatible with the audio being provided by the custom audio source.

If the custom audio source being used to publish is recording from the microphone but managed by the host application, the echo-cancellation SDK above will not work unless the SDK-managed microphone is activated. To work around that requirement, see iOS Echo Cancellation.

Publishing with Bluetooth on Android

The SDK automatically reverts to the VIDEO_CHAT preset on Android when the following conditions are met:

- The assigned configuration does not use the VOICE_COMMUNICATION usage value.
- A Bluetooth microphone is connected to the device.
- The local participant is publishing to a Stage.

This is a limitation of the Android operating system in regard to how Bluetooth headsets are used for recording audio.

Integrating with Other SDKs

Because both iOS and Android support only one active audio mode per application, it is common to run into conflicts if your application uses multiple SDKs that require control of the audio mode. When you run into these conflicts, there are some common resolution strategies to try, explained below.

Match Audio Mode Values

Using either the IVS SDK's advanced audio-configuration options or the other SDK's functionality, have the two SDKs align on the underlying values.

Agora

iOS

On iOS, telling the Agora SDK to keep the `AVAudioSession` active will prevent it from deactivating while the IVS real-time streaming broadcast SDK is using it.

```
myRtcEngine.SetParameters("{\"che.audio.keep.audiosession\":true}");
```

Android

Avoid calling `setEnableSpeakerphone` on `RtcEngine`, and call `enableLocalAudio(false)` while publishing with the IVS real-time streaming broadcast SDK. You can call `enableLocalAudio(true)` again when the IVS SDK is not publishing.

Using Amazon EventBridge with IVS Real-Time Streaming

You can use Amazon EventBridge to monitor your Amazon Interactive Video Service (IVS) streams.

Amazon IVS sends change events about the status of your streams to Amazon EventBridge. All events that are delivered are valid. However, events are sent on a best-effort basis, which means there is no guarantee that:

- Events are delivered — A designated event can occur (e.g., a participant published) but it is possible that Amazon IVS will not send a corresponding event to EventBridge. Amazon IVS tries to deliver events for several hours before giving up.
- Events that are delivered will arrive in a specified timeframe — You may receive events up to a few hours old.
- Events are delivered in order — Events may be out of order, especially if they are sent within a short time of each other. For example, you could see Participant Unpublished before Participant Published.

While it's rare for events to be missing, late, or out of sequence, you should handle these possibilities if you write business-critical programs that depend on the order or existence of notification events.

You can create EventBridge rules for any of the following events.

Event Type	Event	Sent When ...
IVS Composition State Change	Destination Failure	An attempt to output to a Destinati on failed. For example, broadcast ing to a channel failed because there was no stream key or another broadcast was happening.
IVS Composition State Change	Destination Start	Output to a Destination successfully started.
IVS Composition State Change	Destination End	Output to a Destination finished.

Event Type	Event	Sent When ...
IVS Composition State Change	Destination Reconnecting	Output to a Destination was interrupted and a reconnect is being attempted.
IVS Composition State Change	Session Start	A Composition session was created. This event fires when a Composition process pipeline successfully initializ es. At this point, the Composition pipeline has successfully subscribe d to a Stage and is receiving media and able to compose video.
IVS Composition State Change	Session End	A Composition session completed.
IVS Composition State Change	Session Failure	A Composition pipeline failed to initialize due to Stage resources not being available, or any other internal error.
IVS Stage Update	Participant Published	A participant begins publishing to a stage.
IVS Stage Update	Participant Unpublished	A participant has stopped publishing to a stage.

Creating Amazon EventBridge Rules for Amazon IVS

You can create a rule that triggers on an event emitted by Amazon IVS. Follow the steps in Create a rule in Amazon EventBridge in the *Amazon EventBridge User Guide*. When selecting a service, choose **Interactive Video Service (IVS)**.

Examples: Composition State Change

Destination Failure: This event is sent when an attempt to output to a Destination failed. For example, broadcasting to a channel failed because there was no stream key or another broadcast was happening.

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Destination Failure",
     "stage_arn": "<stage-arn>",
     "id": "<Destination-id>",
     "reason": "eg. stream key invalid"
   }
}
```

Destination Start: This event is sent when output to a Destination successfully started.

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Destination Start",
     "stage_arn": "<stage-arn>",
     "id": "<destination-id>",
   }
}
```

Destination End: This event is sent when output to a Destination finished.

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Destination End",
     "stage_arn": "<stage-arn>",
     "id": "<Destination-id>",
   }
}
```

Destination Reconnecting: This event is sent when output to a Destination was interrupted and a reconnect is being attempted.

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Destination Reconnecting",
     "stage_arn": "<stage-arn>",
     "id": "<Destination-id>",
   }
}
```

Session Start: This event is sent when a Composition session was created. This event fires when a Composition process pipeline successfully initializes. At this point, the Composition pipeline has successfully subscribed to a Stage and is receiving media and able to compose video.

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Session Start",
     "stage_arn": "<stage-arn>"
   }
}
```

Session End: This event is sent when a Composition session completed and all resources were deleted.

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Session End",
     "stage_arn": "<stage-arn>"
   }
}
```

Session Failure: This event is sent when a Composition pipeline failed to initialize due to Stage resources not being available, no participants being in the stage, or any other internal error.

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Session Failure",
     "stage_arn": "<stage-arn>",
     "reason": "eg. no participants in the stage"
   }
}
```

Examples: Stage Update

Stage update events include an event name (which classifies the event) and metadata about the event. The metadata includes the participant ID which triggered the event, the associated stage and session IDs, and the user ID.

Participant Published: This event is sent when a participant begins publishing to a stage.

```
{
    "version": "0",
    "id": "12345678-1a23-4567-a1bc-1a2b34567890",
    "detail-type": "IVS Stage Update",
    "source": "aws.ivs",
    "account": "123456789012",
    "time": "2020-06-23T20:12:36Z",
    "region": "us-west-2",
    "resources": [
        "arn:aws:ivs:us-west-2:123456789012:stage/AbCdef1G2hij"
    ],
    "detail": {
        "session_id": "st-1234567890",
        "event_name": "Participant Published",
        "user_id": "Your User Id",
        "participant_id": "xYz1c2d3e4f"
    }
}
```

Participant Unpublished: This event is sent when a participant has stopped publishing to a stage.

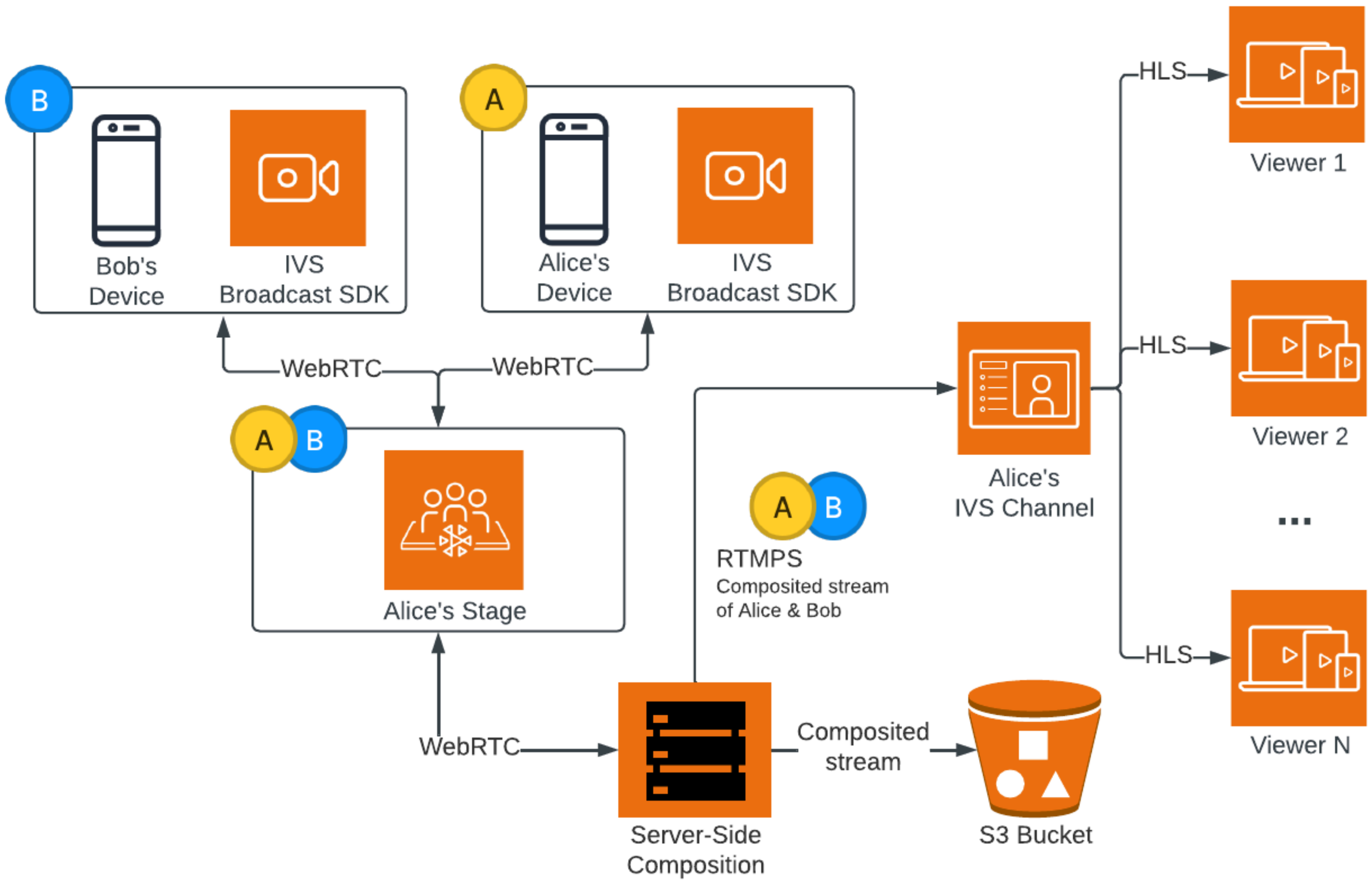
```
{
    "version": "0",
    "id": "12345678-1a23-4567-a1bc-1a2b34567890",
    "detail-type": "IVS Stage Update",
    "source": "aws.ivs",
    "account": "123456789012",
    "time": "2020-06-23T20:12:36Z",
    "region": "us-west-2",
    "resources": [
        "arn:aws:ivs:us-west-2:123456789012:stage/AbCdef1G2hij"
    ],
    "detail": {
        "session_id": "st-1234567890",
        "event_name": "Participant Unpublished",
        "user_id": "Your User Id",
        "participant_id": "xYz1c2d3e4f"
    }
}
```

Server-Side Composition (Real-Time Streaming)

Server-side composition uses an IVS server to mix audio and video from all stage participants and then sends this mixed video to an IVS channel (e.g., to reach a larger audience) or an S3 bucket. Server-side composition is invoked through IVS control-plane endpoints in the stage's home region.

Broadcasting or recording a stage using server-side composition offers numerous benefits, making it an attractive choice for users seeking efficient and reliable cloud-based video workflows.

This diagram illustrates how server-side composition works:



Benefits

Compared to client-side composition, server-side composition has the following benefits:

- **Reduced client load** — With server-side composition, the burden of processing and combining audio and video sources is shifted from individual client devices to the server itself. Server-side composition eliminates the need for client devices to use their CPU and network resources for compositing the view and transmitting it to IVS. This means viewers can watch the broadcast without their devices having to handle resource-intensive tasks, which can lead to improved battery life and smoother viewing experiences.

- **Consistent quality** — Server-side composition allows for precise control over the quality, resolution, and bitrate of the final stream. This ensures a consistent viewing experience for all viewers, regardless of their individual devices' capabilities.
- **Resilience** — By centralizing the composition process on the server, the broadcast becomes more robust. Even if a publisher device experiences technical limitations or fluctuations, the server can adapt and provide a smoother stream to all audience members.
- **Bandwidth efficiency** — Since the server handles the composition, stage publishers do not have to spend extra bandwidth broadcasting the video to IVS.

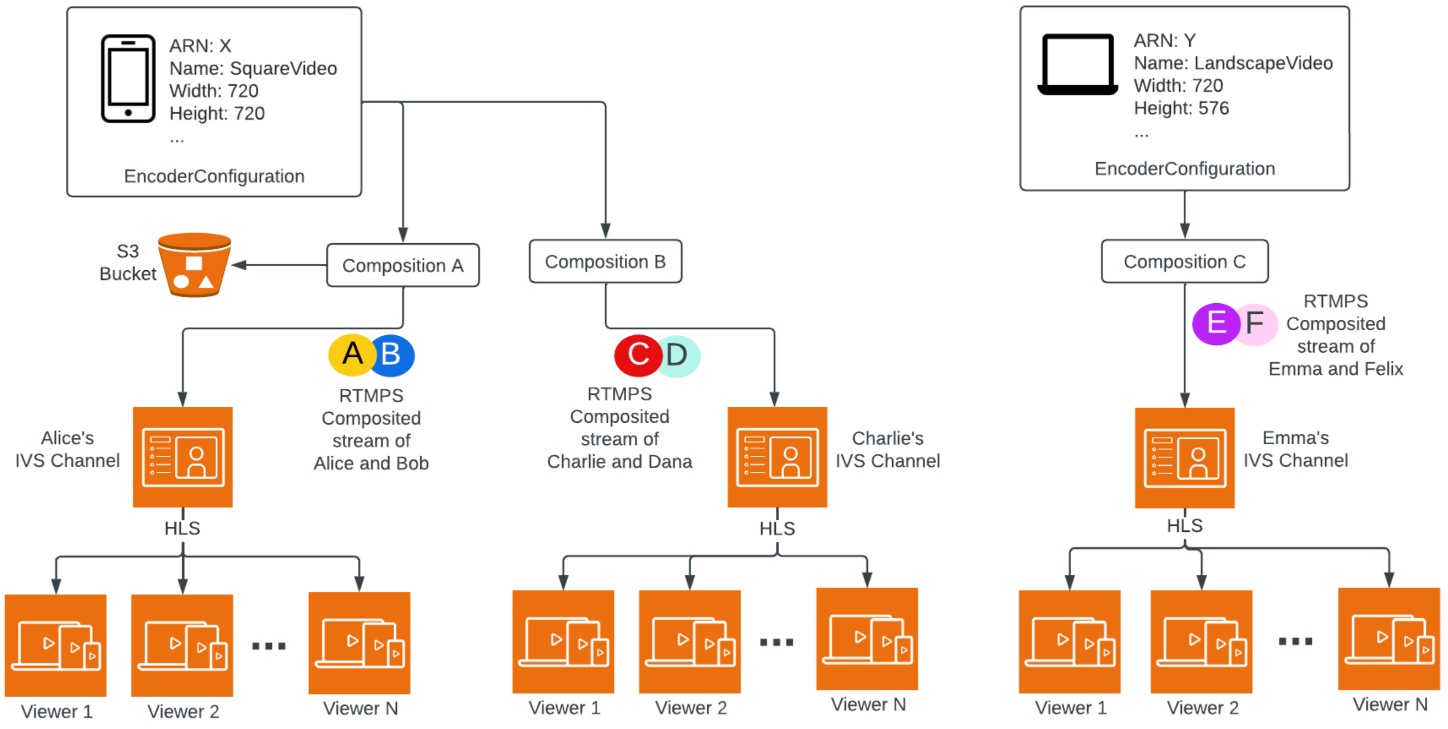
Alternatively, to broadcast a stage to an IVS channel, you can do the composition client side; see Enabling Multiple Hosts on an IVS Stream in the *IVS Low-Latency Streaming User Guide*.

IVS API

Server-side composition uses these key API elements:

- An *EncoderConfiguration* object allows you to customize the format of the video to be generated (height, width, bitrate, and other streaming parameters). You can reuse an EncoderConfiguration every time you call the StartComposition endpoint.
- *Composition* endpoints track the video composition and output to an IVS channel.
- *StorageConfiguration* tracks the S3 bucket where compositions are recorded.

To use server-side composition, you need to create an EncoderConfiguration and attach it when calling the StartComposition endpoint. In this example, the SquareVideo EncoderConfiguration is used in two Compositions:



For complete information, see IVS Real-Time Streaming API Reference.

Layouts

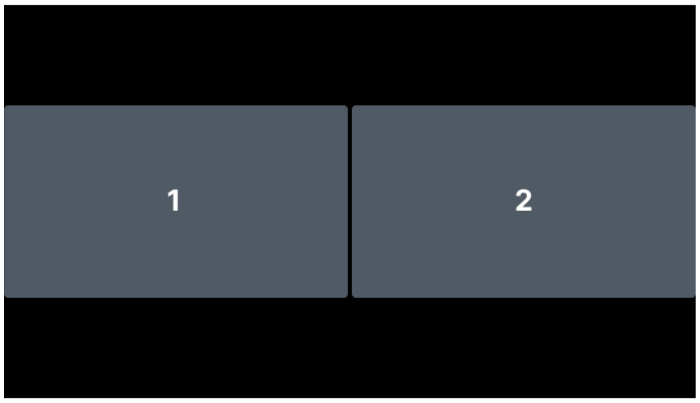
The StartComposition endpoint offers two layout options: grid and pip (Picture-in-Picture).

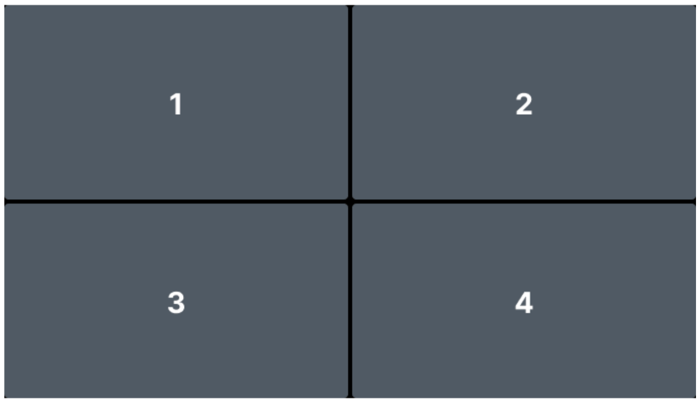
Grid Layout

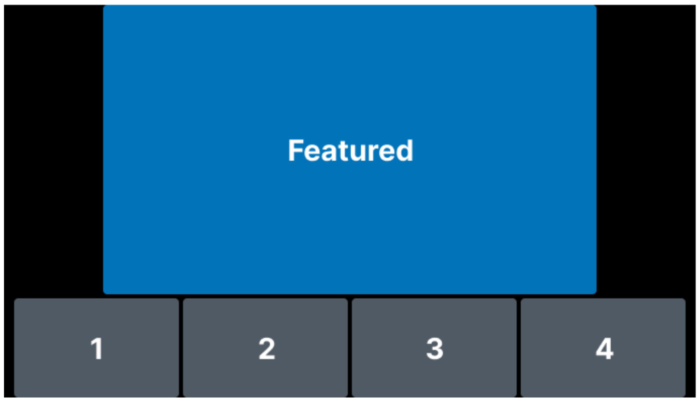
The grid layout arranges stage participants in a grid of equally sized slots. It provides several customizable properties:

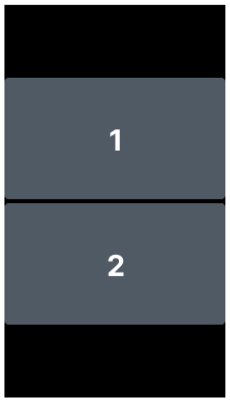
- `videoAspectRatio` sets the participant display mode to control the aspect ratio of video tiles.
- `videoFillMode` defines how video content fits within the participant tile.
- `gridGap` specifies the spacing between participant tiles in pixels.
- `omitStoppedVideo` allows excluding stopped video streams from the composition.
- `featuredParticipantAttribute` identifies the featured slot. When this is set, the featured participant is displayed in a larger slot on the main screen, with other participants shown below it.

For details on grid layout (including valid values and defaults for all fields), see the GridConfiguration data type.









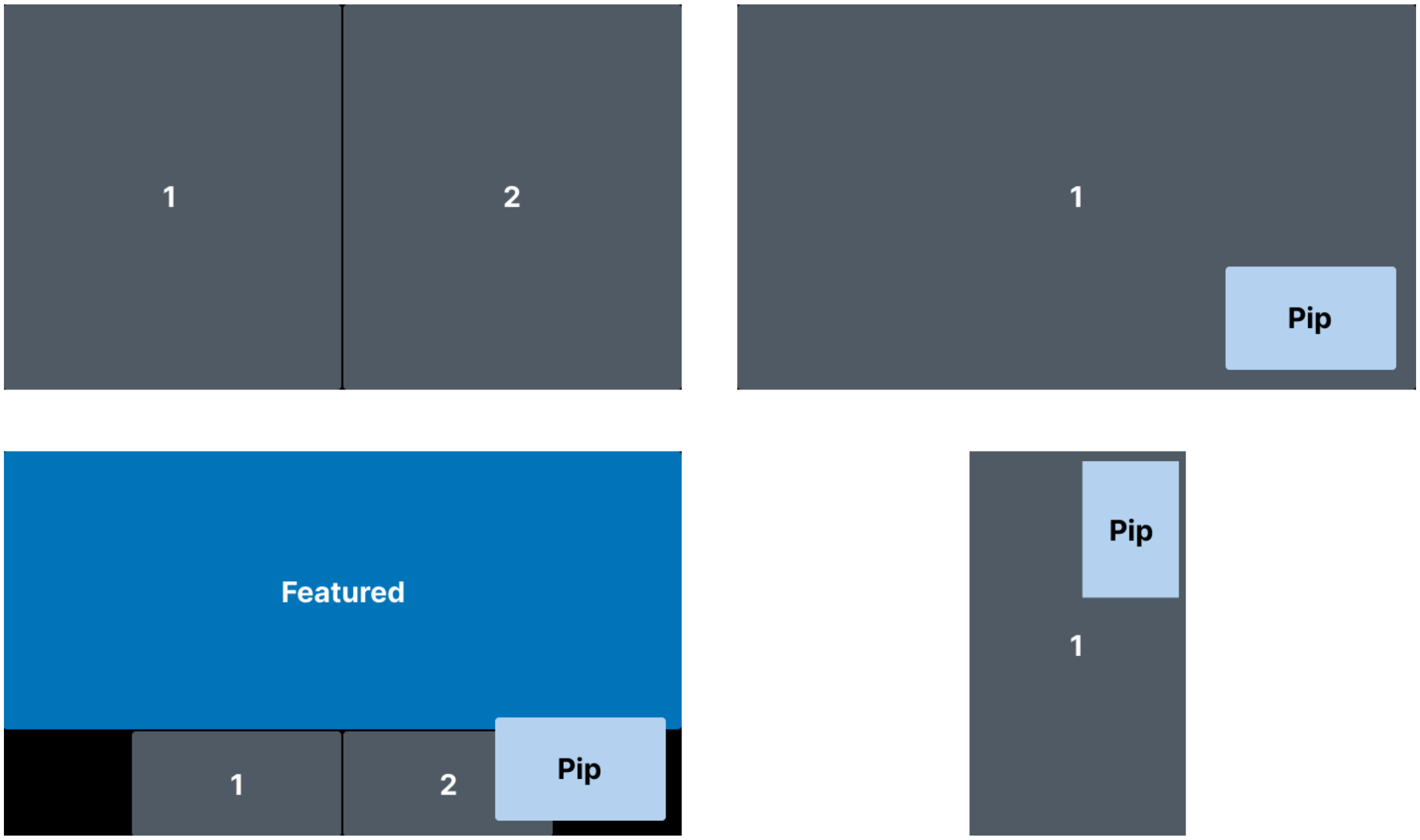
Picture-in-Picture (PiP) Layout

The PiP layout enables displaying a participant in an overlay window with configurable size, position, and behavior. Key properties include:

- `pipParticipantAttribute` specifies the participant for the PiP window.
- `pipPosition` determines the corner position of the PiP window.
- `pipWidth` and `pipHeight`configure the width and height of the PiP window.
- `pipOffset` sets the offset position of the PiP window in pixels from the closest edges.
- `pipBehavior` defines PiP behavior when all other participants have left.

Like the grid layout, the PiP supports `featuredParticipantAttribute`, `omitStoppedVideo`, `videoFillMode`, and `gridGap` to further customize the composition.

For details on PiP layout (including valid values and defaults for all fields), see the PipConfiguration data type.



Note: The maximum resolution supported by a stage publisher on server-side composition is 1080p. If a publisher sends video higher than 1080p, the publisher will be rendered as an audio-only participant.

Important: Ensure your application does not depend on the specific features of the current layout, such as size and position of tiles. *Visual improvements to layouts can be introduced at any time*.

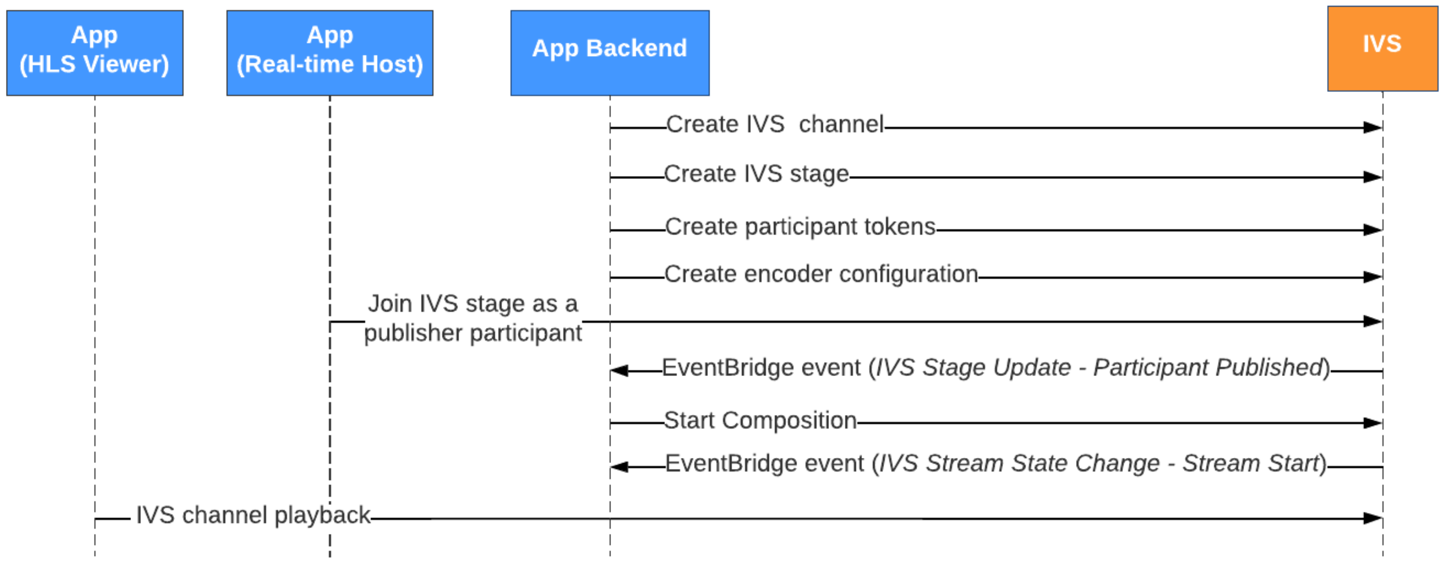
Getting Started

Prerequisites

To use server-side composition, you must have a stage with active publishers and use an IVS channel and/or an S3 bucket as the composition destination. Below, we describe one possible workflow that uses EventBridge events to start a composition that broadcasts the stage to an IVS channel when a participant publishes. Alternatively, you can start and stop compositions based on your own app logic. See Composite Recording for another example which showcases the use of server-side composition to record a stage directly to an S3 bucket.

1. Create an IVS channel. See Getting Started with Amazon IVS Low-Latency Streaming.

2. Create an IVS stage and participant tokens for each publisher.
3. Create an EncoderConfiguration.
4. Join the stage and publish to it. (See the "Publishing and Subscribing" sections of the real-time streaming broadcast SDK guides: Web, Android, and iOS.)
5. When you receive a Participant Published EventBridge event, call StartComposition with your desired layout configuration.
6. Wait for a few seconds and see the composited view in the channel playback.


App (HLS Viewer)
App (Real-time Host)
App Backend
IVS
Create IVS channel
Create IVS stage
Create participant tokens
Create encoder configuration
Join IVS stage as a publisher participant
EventBridge event (*IVS Stage Update - Participant Published*)
Start Composition
EventBridge event (*IVS Stream State Change - Stream Start*)
IVS channel playback

Note: A Composition performs auto-shutdown after 60 seconds of inactivity from publisher participants on the stage. At that point, the Composition is terminated and transitions to a STOPPED state. A Composition is automatically deleted after a few minutes in the STOPPED state.

CLI Instructions

Using the AWS CLI is an advanced option and requires that you first download and configure the CLI on your machine. For details, see the AWS Command Line Interface User Guide.

Now you can use the CLI to create and manage resources. The Composition endpoints are under the `ivs-realtime` namespace.

Create the EncoderConfiguration Resource

An EncoderConfiguration is an object that allows you to customize the format of the generated video (height, width, bitrate, and other streaming parameters). You can reuse an EncoderConfiguration every time you call the Composition endpoint, as explained in the next step.

The command below creates an EncoderConfiguration resource that configures server-side video composition parameters like video bitrate, frame rate and resolution:

```
aws ivs-realtime create-encoder-configuration --name "MyEncoderConfig" --video
 "bitrate=2500000,height=720,width=1280,framerate=30"
```

The response is:

```
{
   "encoderConfiguration": {
      "arn": "arn:aws:ivs:us-east-1:927810967299:encoder-configuration/9W59OBY2M8s4",
      "name": "MyEncoderConfig",
      "tags": {},
      "video": {
         "bitrate": 2500000,
         "framerate": 30,
         "height": 720,
         "width": 1280
      }
   }
}
```

Start a Composition

Using the EncoderConfiguration ARN provided in the response above, create your Composition resource:

Grid Layout Example

```
aws ivs-realtime start-composition --stage-arn "arn:aws:ivs:us-
east-1:927810967299:stage/8faHz1SQp0ik" --destinations '[{"channel":
 {"channelArn": "arn:aws:ivs:us-east-1:927810967299:channel/
DOlMW4dfMR8r", "encoderConfigurationArn": "arn:aws:ivs:us-
east-1:927810967299:encoder-configuration/9W59OBY2M8s4"}}]' --layout '{"grid":
{"featuredParticipantAttribute":"isFeatured","videoFillMode":"COVER","gridGap":0}}'
```

PiP Layout Example

```
aws ivs-realtime start-composition --stage-arn "arn:aws:ivs:us-
east-1:927810967299:stage/8faHz1SQp0ik" --destinations '[{"channel": {"channelArn":
 "arn:aws:ivs:us-east-1:927810967299:channel/DOlMW4dfMR8r", "encoderConfigurationArn":
 "arn:aws:ivs:us-east-1:927810967299:encoder-configuration/DEkQHWPVaOwO"}}]' --layout
 '{"pip":{"pipParticipantAttribute":"isPip","pipOffset":10,"pipPosition":"TOP_RIGHT"}}'
```

Note: You can use this tool to more easily generate the `--layout` configuration based on your layout choices.

The response will show that the Composition is created with a `STARTING` state. Once the Composition starts publishing the composition, the state transitions to `ACTIVE`. (You can see the state by calling the ListCompositions or GetComposition endpoint.)

Once a Composition is `ACTIVE`, the composite view of the IVS stage is visible on the IVS channel, using ListCompositions:

```
aws ivs-realtime list-compositions
```

The response is:

```
{
   "compositions": [
      {
         "arn": "arn:aws:ivs:us-east-1:927810967299:composition/YVoaXkKdEdRP",
         "destinations": [
            {
               "id": "bD9rRoN91fHU",
               "startTime": "2023-09-21T15:38:39+00:00",
               "state": "ACTIVE"
            }
         ],
         "stageArn": "arn:aws:ivs:us-east-1:927810967299:stage/8faHz1SQp0ik",
         "startTime": "2023-09-21T15:38:37+00:00",
         "state": "ACTIVE",
         "tags": {}
      }
   ]
}
```

Note: You need to have publisher participants actively publishing to the stage to keep the composition alive. For more information, see the "Publishing and Subscribing" sections of the real-time streaming broadcast SDK guides: Web, Android, and iOS. You must create a distinct stage token for each participant.

Enable Screen Share

To use a fixed screen-share layout, follow the steps below.

Create the EncoderConfiguration Resource

The command below creates an EncoderConfiguration resource that configures server-side composition parameters (video bitrate, framerate, and resolution).

```
aws ivs-realtime create-encoder-configuration --name "test-ssc-with-screen-share" --
video={bitrate=2000000,framerate=30,height=720,width=1280}
```

Create a stage participant token with a `screen-share` attribute. Since we will specify `screen-share` as the name of the `featured` slot, we need to create a stage token with the `screen-share` attribute set to `true`:

```
aws ivs-realtime create-participant-token --stage-arn "arn:aws:ivs:us-
east-1:123456789012:stage/u9OiE29bT7Xp" --attributes screen-share=true
```

The response is:

```
{
   "participantToken": {
      "attributes": {
         "screen-share": "true"
      },
      "expirationTime": "2023-08-04T05:26:11+00:00",
      "participantId": "E813MFklPWLF",
      "token":
 "eyJhbGciOiJLTVMiLCJ0eXAiOiJKV1QifQ.eyJleHAiOjE2OTExMjY3NzEsImlhdCI6MTY5MTA4MzU3MSwianRpIjoiRT
   }
}
```

Start the Composition

To start the composition using the screen-share feature, we use this command:

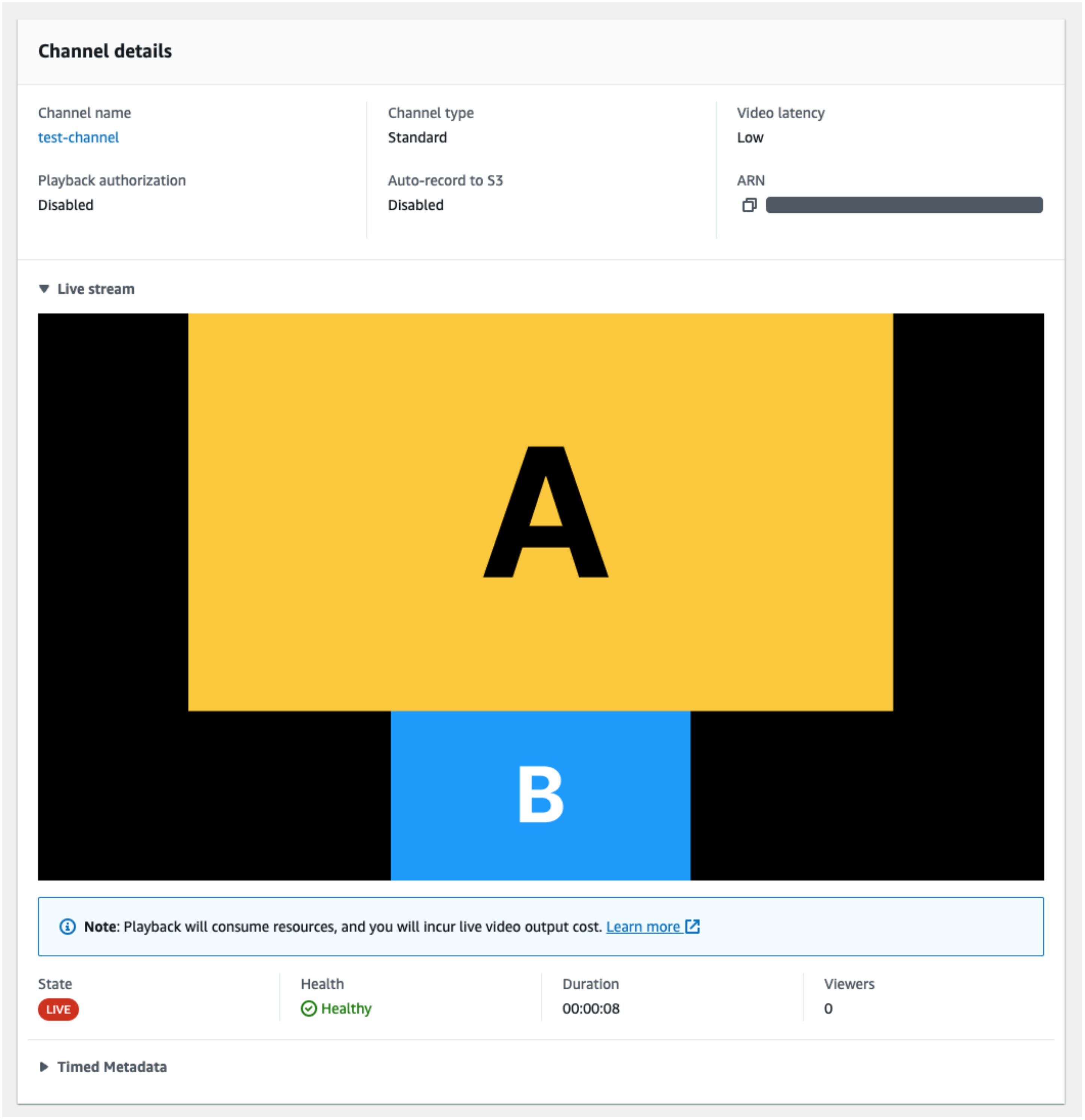
```
aws ivs-realtime start-composition --stage-arn "arn:aws:ivs:us-
east-1:927810967299:stage/8faHz1SQp0ik" --destinations  '[{"channel": {"channelArn":
 "arn:aws:ivs:us-east-1:927810967299:channel/DOlMW4dfMR8r", "encoderConfigurationArn":
```

```
"arn:aws:ivs:us-east-1:927810967299:encoder-configuration/DEkQHWPVaOwO"}}]' --layout
'{"grid":{"featuredParticipantAttribute":"screen-share"}}'
```

The response is:

```
{
   "composition" : {
      "arn" : "arn:aws:ivs:us-east-1:927810967299:composition/B19tQcXRgtoz",
      "destinations" : [ {
         "configuration" : {
            "channel" : {
               "channelArn" : "arn:aws:ivs:us-east-1:927810967299:channel/
DOlMW4dfMR8r",
               "encoderConfigurationArn" : "arn:aws:ivs:us-east-1:927810967299:encoder-
configuration/DEkQHWPVaOwO"
            },
            "name" : ""
         },
         "id" : "SGmgBXTULuXv",
         "state" : "STARTING"
      } ],
      "layout" : {
         "grid" : {
            "featuredParticipantAttribute" : "screen-share",
            "gridGap": 2,
            "omitStoppedVideo": false,
            "videoAspectRatio": "VIDEO"
         }
      },
      "stageArn" : "arn:aws:ivs:us-east-1:927810967299:stage/8faHz1SQp0ik",
      "startTime" : "2023-09-27T21:32:38Z",
      "state" : "STARTING",
      "tags" : { }
   }
}
```

When the stage participant `E813MFklPWLF` joins the stage, that participant's video will be displayed in the featured slot, and all other stage publishers will be rendered below the slot:



Channel details

Channel name
test-channel

Channel type
Standard

Video latency
Low

Playback authorization
Disabled

Auto-record to S3
Disabled

ARN
[redacted]

▼ Live stream

Note: Playback will consume resources, and you will incur live video output cost. Learn more

State
LIVE

Health
Healthy

Duration
00:00:08

Viewers
0

▶ Timed Metadata

Stop the Composition

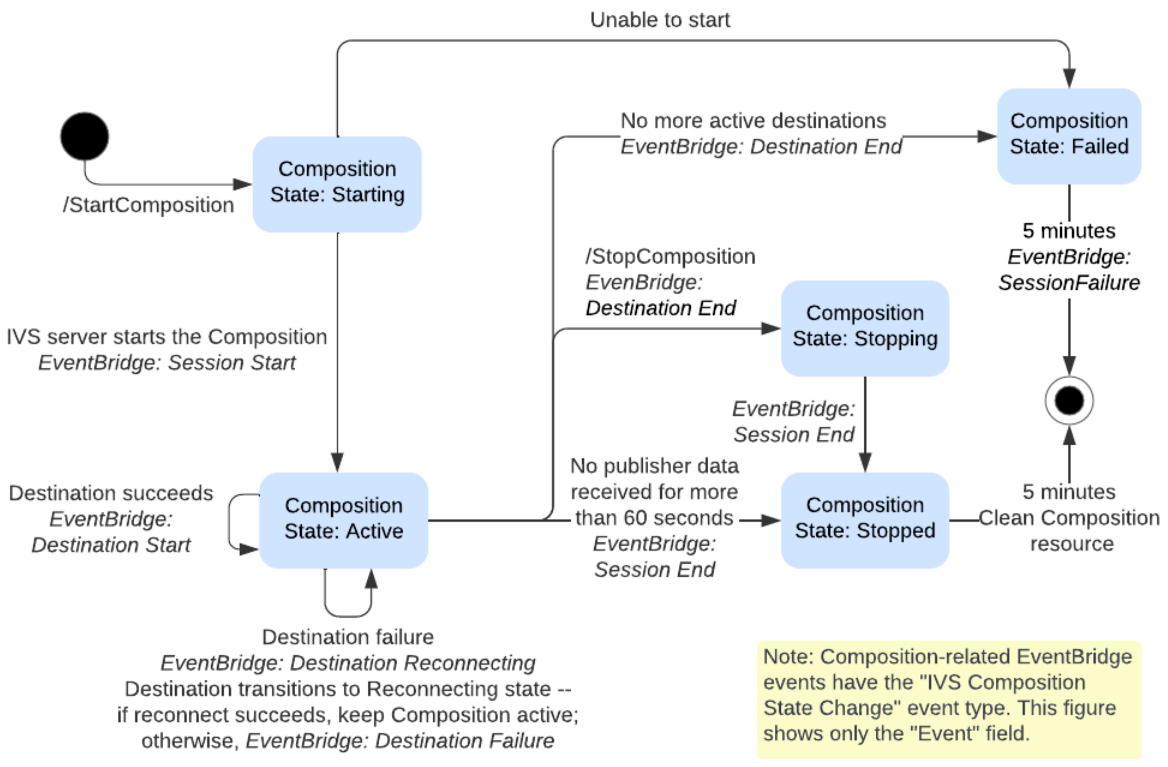
To stop a composition at any point, call the StopComposition endpoint:

```
aws ivs-realtime stop-composition --arn arn:aws:ivs:us-east-1:927810967299:composition/
B19tQcXRgtoz
```

Composition Lifecycle

Use the diagram below to understand the state transitions of a Composition. At a high level, the life cycle of a Composition is as follows:

1. A Composition resource is created when the user calls the StartComposition endpoint
2. Once IVS successfully starts the Composition, an “IVS Composition State Change (Session Start)” EventBridge event is sent. See Using EventBridge with IVS Real-Time Streaming for details about events.
3. Once a Composition is in an active state, the following can happen:
    - User stops the Composition — If the StopComposition endpoint is called, IVS initiates a graceful shutdown of the Composition, sending "Destination End" events followed by a "Session End" event.
    - Composition performs auto-shutdown — If no participant is actively publishing to the IVS stage, the Composition is finalized automatically after 60 seconds and EventBridge events are sent.
    - Destination failure — If a destination unexpectedly fails (e.g., the IVS channel gets deleted), the destination transitions to the `RECONNECTING` state and a “Destination Reconnecting” event is sent. If recovery is impossible, IVS transitions the destination to the `FAILED` state and a “Destination Failure” event is sent. IVS keeps the composition alive if at least one of its destinations is active.
4. Once the composition is in the `STOPPED` or `FAILED` state, it is automatically cleaned up after five minutes. (Then it no longer is retrieved by ListCompositions or GetComposition.)


Unable to start
/StartComposition
Composition State: Starting
No more active destinations
EventBridge: Destination End
Composition State: Failed
5 minutes
EventBridge: SessionFailure
/StopComposition
EvenBridge: Destination End
Composition State: Stopping
IVS server starts the Composition
EventBridge: Session Start
EventBridge: Session End
Destination succeeds
EventBridge: Destination Start
Composition State: Active
No publisher data received for more than 60 seconds
EventBridge: Session End
Composition State: Stopped
5 minutes
Clean Composition resource
Destination failure
EventBridge: Destination Reconnecting
Destination transitions to Reconnecting state -- if reconnect succeeds, keep Composition active; otherwise, *EventBridge: Destination Failure*
Note: Composition-related EventBridge events have the "IVS Composition State Change" event type. This figure shows only the "Event" field.

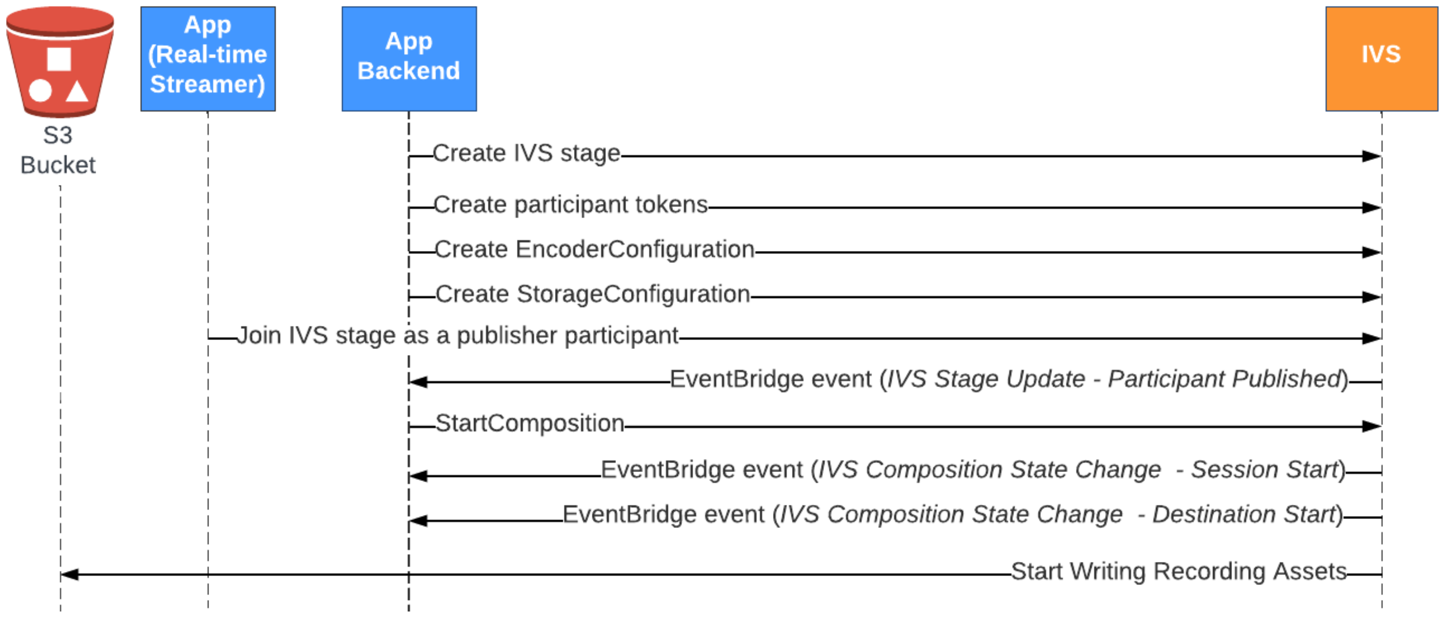
Composite Recording (Real-Time Streaming)

This document explains how to use the composite-recording feature within server-side composition. Composite recording allows you to generate HLS recordings of an IVS stage by effectively combining all stage publishers into one view using an IVS server, and then saving the resulting video to an S3 bucket.

Prerequisites

To use composite recording, you must have a stage with active publishers and an S3 bucket to use as the recording destination. Below, we describe one possible workflow that uses EventBridge events to record a composition to an S3 bucket. Alternatively, you can start and stop compositions based on your own app logic.

1. Create an IVS stage and participant tokens for each publisher.
2. Create an EncoderConfiguration (an object representing how the recorded video should be rendered).
3. Create an S3 bucket and a StorageConfiguration (where the recording contents will be stored).
4. Join the stage and publish to it.
5. When you receive a Participant Published EventBridge event, call StartComposition with an S3 DestinationConfiguration object as the destination
6. After a few seconds, you should be able to see the HLS segments being persisted to your S3 buckets.


S3 Bucket
App (Real-time Streamer)
App Backend
IVS
Create IVS stage
Create participant tokens
Create EncoderConfiguration
Create StorageConfiguration
Join IVS stage as a publisher participant
EventBridge event (IVS Stage Update - Participant Published)
StartComposition
EventBridge event (IVS Composition State Change - Session Start)
EventBridge event (IVS Composition State Change - Destination Start)
Start Writing Recording Assets

Note: A composition performs auto-shutdown after 60 seconds of inactivity from publisher participants on the stage. At that point, the composition is terminated and transitions to a `STOPPED` state. A composition is automatically deleted after a few minutes in the `STOPPED` state. For details, see Composition Lifecycle in *Server-Side Composition*.

Composite Recording Example: StartComposition with an S3 Bucket Destination

The example below shows a typical call to the StartComposition endpoint, specifying S3 as the only destination for the composition. Once the composition transitions to an `ACTIVE` state, video segments and metadata will start to be written to the S3 bucket specified by the `storageConfiguration` object. To create compositions with different layouts, see "Layouts" in Server-Side Composition and the IVS Real-Time Streaming API Reference.

Request

```
POST /StartComposition HTTP/1.1
Content-type: application/json

{
   "destinations": [
      {
         "s3": {
            "encoderConfigurationArns": [
              "arn:aws:ivs:ap-northeast-1:927810967299:encoder-configuration/
PAAwglkRtjge"
            ],
            "storageConfigurationArn": "arn:aws:ivs:ap-
northeast-1:927810967299:storage-configuration/ZBcEbgbE24Cq"
         }
      }
   ],
   "idempotencyToken": "db1i782f1g9",
   "stageArn": "arn:aws:ivs:ap-northeast-1:927810967299:stage/WyGkzNFGwiwr"
}
```

Response

```
{
    "composition": {
        "arn": "arn:aws:ivs:ap-northeast-1:927810967299:composition/s2AdaGUbvQgp",
        "destinations": [
            {
                "configuration": {
                    "name": "",
                    "s3": {
                        "encoderConfigurationArns": [
                            "arn:aws:ivs:ap-northeast-1:927810967299:encoder-
configuration/PAAwglkRtjge"
                        ],
                        "recordingConfiguration": {
                            "format": "HLS"
                        },
                        "storageConfigurationArn": "arn:aws:ivs:ap-
northeast-1:927810967299:storage-configuration/ZBcEbgbE24Cq"
                    }
                },
                "detail": {
                    "s3": {
                        "recordingPrefix": "MNALAcH9j2EJ/s2AdaGUbvQgp/2pBRKrNgX1ff/
composite"
                    }
                },
                "id": "2pBRKrNgX1ff",
                "state": "STARTING"
            }
        ],
        "layout": null,
        "stageArn": "arn:aws:ivs:ap-northeast-1:927810967299:stage/WyGkzNFGwiwr",
        "startTime": "2023-11-01T06:25:37Z",
        "state": "STARTING",
        "tags": {}
    }
}
```

The `recordingPrefix` field, present in the StartComposition response can be used to determine where the recording contents will be stored.

Recording Contents

When the composition transitions to an `ACTIVE` state, you will start to see HLS video segments and metadata files being written to the S3 bucket that was provided when calling StartComposition. These contents are available for post-processing or playback as on-demand video.

Note that after a composition becomes live, an “IVS Composition State Change” event is emitted, and it may take a little time before the manifest files and video segments are written. We recommend that you play back or process recorded streams only after the “IVS Composition State Change (Session End)” event is received. For details, see Using EventBridge with IVS Real-Time Streaming .

The following is a sample directory structure and contents of a recording of a live IVS session:

```
MNALAcH9j2EJ/s2AdaGUbvQgp/2pBRKrNgX1ff/composite
   events
      recording-started.json
      recording-ended.json
   media
      hls
```

The `events` folder contains the metadata files corresponding to the recording event. JSON metadata files are generated when recording starts, ends successfully, or ends with failures:

- `events/recording-started.json`
- `events/recording-ended.json`
- `events/recording-failed.json`

A given `events` folder will contain `recording-started.json` and either `recording-ended.json` or `recording-failed.json`.

These contain metadata related to the recorded session and its output formats. JSON details are given below.

The `media` folder contains the supported media contents. The `hls` subfolder contains all media and the manifest files generated during the composition session and is playable with the IVS player. The HLS manifest is located in the `multivariant.m3u8` folder.

Bucket Policy for StorageConfiguration

When a StorageConfiguration object is created, IVS will get access to write content to the specified S3 bucket. This access is granted by making modifications to the S3 bucket's policy. *If the policy for the bucket is altered in a way that removes IVS's access, ongoing and new recordings will fail.*

The example below shows an S3 bucket policy that allows IVS to write to the S3 bucket:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CompositeWrite-y1d212y",
            "Effect": "Allow",
            "Principal": {
                "Service": "ivs-composite.ap-northeast-1.amazonaws.com"
            },
            "Action": [
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Resource": "arn:aws:s3:::my-s3-bucket/*",
            "Condition": {
                "StringEquals": {
                    "s3:x-amz-acl": "bucket-owner-full-control"
                },
                "Bool": {
                    "aws:SecureTransport": "true"
                }
            }
        }
    ]
}
```

JSON Metadata Files

This metadata is in JSON format. It comprises the following information:

Field	Type	Required	Description
`stage_arn`	string	Yes	ARN of the stage being used as the source of the composition.
`media`	object	Yes	Object that contains the enumerated objects of media content available for this recording. Valid values: `"hls"`.
`hls`	object	Yes	Enumerated field that describes the Apple HLS format output.
`duration_ms`	integer	Conditional	Duration of the recorded HLS content in milliseconds. This is available only when `recording_status` is `"RECORDING_ENDED"` or `"RECORDING_ENDED_WITH_FAILURE"`. If a failure occurred before any recording was done, this is 0.
`path`	string	Yes	Relative path from the S3 prefix where HLS content is stored.
`playlist`	string	Yes	Name of the HLS master playlist file.
`renditions`	object	Yes	Array of renditions (HLS variant) of metadata objects. There always is at least one rendition.

Field	Type	Required	Description
`path`	string	Yes	Relative path from the S3 prefix where HLS content is stored for this rendition.
`playlist`	string	Yes	Name of the media playlist file for this rendition.
`resolution_height`	int	Conditional	Pixel resolution height of the encoded video. This is available only when the rendition contains a video track.
`resolution_width`	int	Conditional	Pixel resolution width of the encoded video. This is available only when the rendition contains a video track.
`recording_ended_at`	string	Conditional	RFC 3339 UTC timestamp when the recording ended. This is available only when `recording_status` is `"RECORDING_ENDED"` or `"RECORDING_ENDED_WITH_FAILURE"`. `recording_started_at` and `recording_ended_at` are timestamps when these events are generated and may not exactly match the HLS video-segment timestamps. To accurately determine the duration of a recording, use the `duration_ms` field.

Field	Type	Required	Description
`recording_started_at`	string	Conditional	RFC 3339 UTC timestamp when the recording started. This is unavailable when `recording_status` is `RECORDING_START_FAILED`. See the note above for `recording_ended_at`.
`recording_status`	string	Yes	Status of the recording. Valid values: `"RECORDING_STARTED"`, `"RECORDING_ENDED"`, `"RECORDING_START_FAILED"`, `"RECORDING_ENDED_WITH_FAILURE"`.
`recording_status_message`	string	Conditional	Descriptive information on the status. This is available only when `recording_status` is `"RECORDING_ENDED"` or `"RECORDING_ENDED_WITH_FAILURE"`.
`version`	string	Yes	The version of the metadata schema.

Example: recording-started.json

```
{
  "version": "v1",
  "stage_arn": "arn:aws:ivs:ap-northeast-1:123456789012:stage/aAbBcCdDeE12",
  "recording_started_at": "2023-11-01T06:01:36Z",
  "recording_status": "RECORDING_STARTED",
  "media": {
    "hls": {
      "path": "media/hls",
      "playlist": "multivariant.m3u8",
      "renditions": [
        {
          "path": "720p30-abcdeABCDE12",
          "playlist": "playlist.m3u8",
          "resolution_width": 1280,
          "resolution_height": 720
        }
      ]
    }
  }
}
```

Example: recording-ended.json

```
{
  "version": "v1",
  "stage_arn": "arn:aws:ivs:ap-northeast-1:123456789012:stage/aAbBcCdDeE12",
  "recording_started_at": "2023-10-27T17:00:44Z",
  "recording_ended_at": "2023-10-27T17:08:24Z",
  "recording_status": "RECORDING_ENDED",
  "media": {
    "hls": {
      "duration_ms": 460315,
      "path": "media/hls",
      "playlist": "multivariant.m3u8",
      "renditions": [
        {
          "path": "720p30-abcdeABCDE12",
          "playlist": "playlist.m3u8",
          "resolution_width": 1280,
          "resolution_height": 720
        }
      ]
    }
  }
}
```

Example: recording-failed.json

```
{
  "version": "v1",
  "stage_arn": "arn:aws:ivs:ap-northeast-1:123456789012:stage/aAbBcCdDeE12",
  "recording_started_at": "2023-10-27T17:00:44Z",
  "recording_ended_at": "2023-10-27T17:08:24Z",
  "recording_status": "RECORDING_ENDED_WITH_FAILURE",
  "media": {
    "hls": {
      "duration_ms": 460315,
      "path": "media/hls",
      "playlist": "multivariant.m3u8",
      "renditions": [
        {
          "path": "720p30-abcdeABCDE12",
          "playlist": "playlist.m3u8",
          "resolution_width": 1280,
          "resolution_height": 720
        }
      ]
    }
  }
}
```

Playback of Recorded Content from Private Buckets

By default, the recorded content is private; hence, these objects are inaccessible for playback using the direct S3 URL. If you try to open the HLS multivariate playlist (m3u8 file) for playback using the IVS player or another player, you will get an error (e.g., “You do not have permission to access the requested resource”). Instead, you can play back these files with the Amazon CloudFront CDN (Content Delivery Network).

CloudFront distributions can be configured to serve content from private buckets. Typically this is preferable to having openly accessible buckets where reads bypass the controls offered by CloudFront. You can set up your distribution to be served from a private bucket by creating an origin access control (OAC), which is a special CloudFront user that has read permissions on the private origin bucket. You can create the OAC after you create your distribution, through the CloudFront console or API. See Creating a new origin access control in the *Amazon CloudFront Developer Guide*.

Setting Up Playback using CloudFront with CORS Enabled

This example covers how a developer can set up a CloudFront distribution with CORS enabled, enabling playback of their recordings from any domain. This is especially useful during the development phase, but you can modify the example below to match your production needs.

Step 1: Create an S3 Bucket

Create an S3 bucket that will be used to store the recordings. Note that the bucket needs to be in the same region that you use for your IVS workflow.

Add a permissive CORS policy to the bucket:

1. In the AWS console, go to the **S3 Bucket Permissions** tab.
2. Copy the CORS policy below and paste it under **Cross-origin resource sharing (CORS)**. This will enable CORS access on the S3 bucket.

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "PUT",
            "POST",
            "DELETE",
            "GET"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": [
            "x-amz-server-side-encryption",
            "x-amz-request-id",
            "x-amz-id-2"
        ]
    }
]
```

Step 2: Create a CloudFront Distribution

See [Creating a CloudFront distribution] in the *CloudFront Developer Guide*.

Using the AWS console, enter the following information:

For this field ...	Choose this ...
Origin Domain	The S3 bucket created in the previous step
Origin Access	Origin access control settings (recommended), using default parameters
Default cache behavior: Viewer Protocol Policy	Redirect HTTP to HTTPS
Default cache behavior: Allowed HTTP methods	GET, HEAD and OPTIONS
Default cache behavior: Cache key and origin requests	CachingDisabled policy
Default cache behavior: Origin request policy	CORS-S3Origin
Default cache behavior: Response headers policy	SimpleCORS
Web Application Firewall	Enable security protections

Then save the CloudFront distribution.

Step 3: Set Up the S3 Bucket Policy

1. Delete any StorageConfiguration that you have set up for the S3 bucket. This will remove any bucket policies that were automatically added when creating the policy for that bucket.
2. Go to your CloudFront Distribution, make sure all distribution fields are in the states defined in the previous step, and **Copy the Bucket Policy** (use the **Copy policy** button).
3. Go to your S3 bucket. On the **Permissions** tab, select **Edit Bucket Policy** and paste the bucket policy that you copied in the previous step. After this step, the bucket policy should have the CloudFront policy exclusively.

4. Create a StorageConfiguration, specifying the S3 bucket.

After the StorageConfiguration is created, you will see two items in the S3 bucket policy, one allowing CloudFront to read contents and another one allowing IVS to write contents. An example of a final bucket policy, with CloudFront and IVS access, is shown in Example: S3 Bucket Policy with CloudFront and IVS Access.

Step 4: Play Back Recordings

After you successfully set up the CloudFront distribution and update the bucket policy, you should be able to play back recordings using the IVS player:

1. Successfully start a Composition and make sure you have a recording stored on the S3 bucket.
2. After following the Step 1 through Step 3 in this example, the video files should be available for consumption through the CloudFront URL. Your CloudFront URL is the **Distribution domain name** on the **Details** tab in the Amazon CloudFront console. It should be something like this:

   ```
   a1b23cdef4ghij.cloudfront.net
   ```
3. To play the recorded video through the CloudFront distribution, find the object key for your `multivariant.m3u8` file under the s3 bucket. It should be something like this:

   ```
   FDew6Szq5iTt/9NIpWJHj0wPT/fjFKbylPb3k4/composite/media/hls/
   multivariant.m3u8
   ```
4. Append the object key to the end of your CloudFront URL. Your final URL will be something like this:

   ```
   https://a1b23cdef4ghij.cloudfront.net/FDew6Szq5iTt/9NIpWJHj0wPT/
   fjFKbylPb3k4/composite/media/hls/multivariant.m3u8
   ```
5. You can now add the final URL to the source attribute of an IVS player to watch the full recording. To watch the recorded video, you can use the demo in Getting Started in the *IVS Player SDK: Web Guide*.

Example: S3 Bucket Policy with CloudFront and IVS Access

The snippet below illustrates an S3 bucket policy that allows CloudFront to read content to the private bucket and IVS to write content to the bucket. **Note: Do not copy and paste the**

snippet below to your own bucket. Your policy should contain the IDs that are relevant to your CloudFront distribution and StorageConfiguration.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CompositeWrite-7eiKaIGkC9DO",
      "Effect": "Allow",
      "Principal": {
        "Service": "ivs-composite.ap-northeast-1.amazonaws.com"
      },
      "Action": [
        "s3:PutObject",
        "s3:PutObjectAcl"
      ],
      "Resource": "arn:aws:s3:::eicheane-test-1026-2-ivs-recordings/*",
      "Condition": {
        "StringEquals": {
          "s3:x-amz-acl": "bucket-owner-full-control"
        },
        "Bool": {
          "aws:SecureTransport": "true"
        }
      }
    },
    {
      "Sid": "AllowCloudFrontServicePrincipal",
      "Effect": "Allow",
      "Principal": {
        "Service": "cloudfront.amazonaws.com"
      },
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::eicheane-test-1026-2-ivs-recordings/*",
      "Condition": {
        "StringEquals": {
          "AWS:SourceArn": "arn:aws:cloudfront::844311324168:distribution/
E1NG4YMW5MN25A"
        }
      }
    }
  ]
}
```

Troubleshooting

- **The composition is not written to the S3 bucket —** Ensure that the S3 bucket and StorageConfiguration objects are created and in the same region. Also ensure that IVS has access to the bucket by checking your bucket policy; see Bucket Policy for StorageConfiguration.
- **I can't find a composition when performing *ListCompositions* —** Compositions are ephemeral resources. Once they transition to a final state, they are deleted automatically after a few minutes.
- **My composition stops automatically —** A composition will stop automatically if there is no publisher on the stage for more than 60 seconds.

Known Issue

The media playlist written by composite recording has the tag `#EXT-X-PLAYLIST-TYPE:EVENT` while the composition is ongoing. When composition is done, the tag is updated to `#EXT-X-PLAYLIST-TYPE:VOD`. For a smooth playback experience, we recommend that you use this playlist only after the composition finalizes successfully.

OBS and WHIP Support (Real-Time Streaming)

This document explains how to use WHIP-compatible encoders like OBS to publish to IVS real-time streaming. WHIP (WebRTC-HTTP Ingestion Protocol) is an IETF draft developed to standardize WebRTC ingestion.

WHIP enables compatibility with software like OBS, offering an alternative (to the IVS broadcast SDK) for desktop publishing. More sophisticated streamers familiar with OBS may prefer it for its advanced production features, such as scene transitions, audio mixing, and overlay graphics. This provides developers with a versatile option: use the IVS web broadcast SDK for direct browser publishing or allow streamers to use OBS on their desktop for more powerful tools.

Also, WHIP is beneficial in situations where using the IVS broadcast SDK isn't feasible or preferred. For example, in setups involving hardware encoders, the IVS broadcast SDK might not be an option. However, if the encoder supports WHIP, you can still publish directly from the encoder to IVS.

OBS Guide

OBS supports WHIP as of version 30. To start, download OBS v30 or newer: https://obsproject.com/.

To publish to an IVS stage using OBS via WHIP, follow these steps:

1. Generate a participant token with publish capability. In WHIP terms, a participant token is a bearer token. By default, participant tokens expire in 12 hours, but you can extend the duration up to 14 days.
2. Click **Settings**. In the **Stream** section of the **Settings** panel, select **WHIP** from the **Service** dropdown.
3. For the **Server**, enter https://global.whip.live-video.net.
4. For the **Bearer Token**, enter the participant token that you generated in step 2.
5. Configure your video settings as you normally would, with a few restrictions:
    a. IVS real-time streaming supports input up to 720p at 8.5 Mbps. If you exceed either of these limits, your stream will be disconnected.
    b. We recommend setting your **Keyframe Interval** in the **Output** panel to 1s or 2s. A low keyframe interval allows video playback to start more quickly for viewers. We also

   recommend setting **CPU Usage Preset** to ultrafast and **Tune** to zerolatency, to enable the lowest latency.

   c. Because OBS does not support simulcast, we recommend keeping your bitrate below 2.5 Mbps. This enables viewers on lower-bandwidth connections to watch.

6. Press **Start Streaming**.

Note: We are aware of quality issues (like intermittent video freezing) that can occur with WHIP in OBS. These typically arise when the broadcaster's network is unstable. We recommend testing WHIP in OBS before using it for production live streams. Lowering your broadcast bitrate also may help reduce the occurrence of these issues.

Service Quotas (Real-Time Streaming)

The following are service quotas and limits for Amazon Interactive Video Service (IVS) real-time endpoints, resources, and other operations. Service quotas (also known as limits) are the maximum number of service resources or operations for your AWS account. That is, these limits are per AWS account, unless noted otherwise in the table. Also see AWS Service Quotas.

You use an endpoint to connect programmatically to an AWS service. Also see AWS Service Endpoints.

All quotas are enforced per region.

Service Quota Increases

For quotas that are adjustable, you can request a rate increase through the AWS console. Use the console to view information about service quotas too.

API call rate quotas are not adjustable.

API Call Rate Quotas

Endpoint Type	Endpoint	Default
Composition	`GetComposition`	5 TPS
Composition	`ListCompositions`	5 TPS
Composition	`StartComposition`	5 TPS
Composition	`StopComposition`	5 TPS
MediaEncoder	`CreateEncoderConfiguration`	5 TPS
MediaEncoder	`DeleteEncoderConfiguration`	5 TPS
MediaEncoder	`GetEncoderConfiguration`	5 TPS
MediaEncoder	`ListEncoderConfigurations`	5 TPS

Endpoint Type	Endpoint	Default
Stage	`CreateParticipantToken`	50 TPS
Stage	`CreateStage`	5 TPS
Stage	`DeleteStage`	5 TPS
Stage	`DisconnectParticipant`	5 TPS
Stage	`GetParticipant`	5 TPS
Stage	`GetStage`	5 TPS
Stage	`GetStageSession`	5 TPS
Stage	`ListStages`	5 TPS
Stage	`UpdateStage`	5 TPS
Stage	`ListParticipants`	5 TPS
Stage	`ListParticipantEvents`	5 TPS
Stage	`ListStageSessions`	5 TPS
StorageConfiguration	`CreateStorageConfiguration`	5 TPS
StorageConfiguration	`DeleteStorageConfiguration`	5 TPS
StorageConfiguration	`GetStorageConfiguration`	5 TPS
StorageConfiguration	`ListStorageConfigurations`	5 TPS
Tags	`ListTagsForResource`	10 TPS
Tags	`TagResource`	10 TPS
Tags	`UntagResource`	10 TPS

Other Quotas

Resource or Feature	Default	Adjustable	Description
EncoderConfigurations	20	Yes	Maximum number of EncoderConfiguration objects per account.
Composition destinations	2	No	Maximum number of Destination objects in a Composition object.
Composition: max duration	24	No	Maximum amount of time a composition can exist, in hours.
Compositions	5	Yes	Maximum concurrent Composition objects per account.
Participant publish or subscribe duration	24	No	Maximum length of time a participant can publish or remain subscribed to a stage, in hours.
Participant publish resolution	720p	No	Maximum resolution of video published by participants.
Participant download bitrate	8.5 Mbps	No	Maximum aggregate download bitrate across all of a participant’s subscriptions.
Stage participants (publishe rs)	12	No	Maximum number of participants who can be publishing to a stage at once.
Stage participants (subscrib ers)	10,000	Yes	Maximum number of participants who can be subscribing to a stage at once.

Resource or Feature	Default	Adjustable	Description
Stages	100	Yes	Maximum number of stages, per AWS Region.
StorageConfigurations	5	Yes	Maximum number of StorageConfiguration objects per account.

Real-Time Streaming Optimizations

To ensure that your users have the best experience when streaming and viewing video using IVS real-time streaming, there are several ways you can improve or optimize for parts of the experience, using features that we offer today.

Introduction

When optimizing for a user's quality of experience, it’s important to consider their desired experience, which can change depending on the content they are watching and network conditions.

Throughout this guide we focus on users who are either *publishers* of streams or *subscribers* of streams, and we consider the desired actions and experiences of those users.

Adaptive Streaming: Layered Encoding with Simulcast

This feature is supported only in the following client versions:

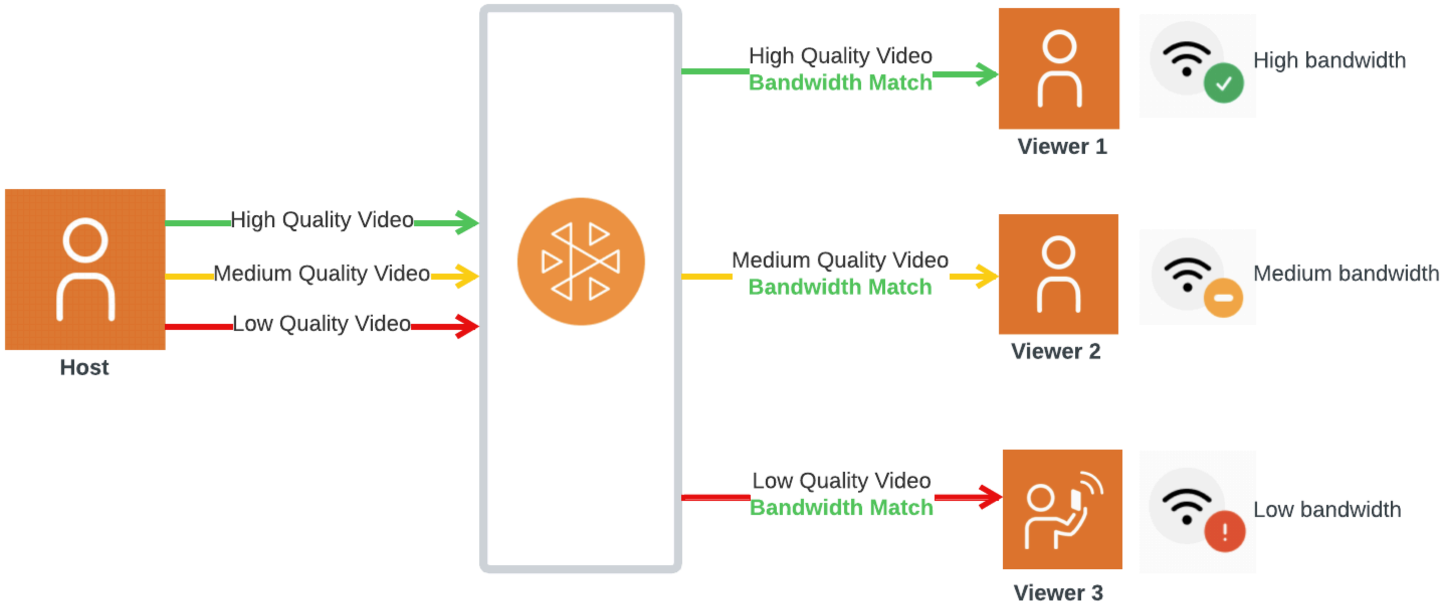
- iOS and Android 1.12.0+
- Web 1.5.1+

You must email amazon-ivs-simulcast@amazon.com to opt-in to this feature for your account. Enabling simulcast via the SDK configuration will have no effect unless you are opted in.

Once you have opted into the feature, when using IVS real-time broadcast SDKs, publishers encode multiple layers of video and subscribers automatically adapt or change to the quality best suited for their network. We call this *layered encoding with simulcast*.

Layered encoding with simulcast is supported on Android and iOS, and on Chrome desktop browsers (for Windows and macOS). We do not support layered encoding on other browsers.

In the diagram below, the host is sending three video qualities (high, medium, and low). IVS forwards the highest quality video to each viewer based on available bandwidth; this provides an optimal experience for each viewer. If Viewer's 1 network connection changes from good to bad, IVS automatically starts sending Viewer 1 lower quality video, so Viewer 1 can keep watching the stream uninterrupted (with the best quality possible).



Default Layers, Qualities, and Framerates

The default qualities and layers provided for mobile and web users are as follows:

Mobile (Android, iOS)	Web (Chrome)
High layer (or custom): • Max bitrate: 900,000 bps • Framerate: 15 fps • Resolution: 360x640	High layer (or custom): • Max bitrate: 1,700,000 bps • Framerate: 30 fps • Resolution: 1280x720
Mid layer: none (not needed, because the difference between the high- and low-layer bitrates on mobile is narrow)	Mid layer: • Max bitrate: 700,000 bps • Framerate: 20 fps • Resolution: 640x360
Low layer: • Max bitrate: 150,000 bps • Framerate: 15 fps • Resolution: 180x320	Low layer: • Max bitrate: 200,000 bps • Framerate: 15 fps • Resolution: 320x180

Configuring Layered Encoding with Simulcast

To use layered encoding with simulcast, you must have opted into the feature, and enabled this on the client. If you enable it, you will see an increase in overall bitrate transmitted, with the benefit of less video freezing.

Android

```
// Opt-out of Simulcast
StageVideoConfiguration config = new StageVideoConfiguration();
config.simulcast.setEnabled(true);

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera, config);

// Other Stage implementation code
```

iOS

```
// Opt-out of Simulcast
let config = IVSLocalStageStreamVideoConfiguration()
config.simulcast.enabled = true

let cameraStream = IVSLocalStageStream(device: camera, configuration: config)

// Other Stage implementation code
```

Web

```
// Opt-out of Simulcast
let cameraStream = new LocalStageStream(cameraDevice, {
   simulcast: { enabled: true }
})

// Other Stage implementation code
```

Streaming Configurations

This section explores other configurations you can make to your video and audio streams.

Changing Video Stream Bitrate

To change the bitrate of your video stream, use the following configuration samples.

Android

```
StageVideoConfiguration config = new StageVideoConfiguration();

// Update Max Bitrate to 1.5mbps
config.setMaxBitrate(1500000);

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera, config);

// Other Stage implementation code
```

iOS

```
let config = IVSLocalStageStreamVideoConfiguration();

// Update Max Bitrate to 1.5mbps
try! config.setMaxBitrate(1500000);

let cameraStream = IVSLocalStageStream(device: camera, configuration: config);

// Other Stage implementation code
```

Web

```
let cameraStream = new LocalStageStream(camera.getVideoTracks()[0], {
   // Update Max Bitrate to 1.5mbps or 1500kbps
   maxBitrate: 1500
})

// Other Stage implementation code
```

Changing Video Stream Framerate

To change the framerate of your video stream, use the following configuration samples.

Android

```
StageVideoConfiguration config = new StageVideoConfiguration();
```

```
// Update target framerate to 10fps
config.targetFramerate(10);

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera, config);

// Other Stage implementation code
```

iOS

```
let config = IVSLocalStageStreamVideoConfiguration();

// Update target framerate to 10fps
try! config.targetFramerate(10);

let cameraStream = IVSLocalStageStream(device: camera, configuration: config);

// Other Stage implementation code
```

Web

```
// Note: On web it is also recommended to configure the framerate of your device from
 userMedia
const camera = await navigator.mediaDevices.getUserMedia({
   video: {
      frameRate: {
         ideal: 10,
         max: 10,
      },
   },
});

let cameraStream = new LocalStageStream(camera.getVideoTracks()[0], {
   // Update Max Framerate to 10fps
   maxFramerate: 10
})
// Other Stage implementation code
```

Optimizing Audio Bitrate and Stereo Support

To change the bitrate and stereo settings of your audio stream, use the following configuration samples.

Web

```
// Note: Disable autoGainControl, echoCancellation, and noiseSuppression when enabling
 stereo.
const camera = await navigator.mediaDevices.getUserMedia({
   audio: {
      autoGainControl: false,
      echoCancellation: false,
      noiseSuppression: false
   },
});

let audioStream = new LocalStageStream(camera.getAudioTracks()[0], {
   // Optional: Update Max Audio Bitrate to 96Kbps. Default is 64Kbps
   maxAudioBitrateKbps: 96,

   // Signal stereo support. Note requires dual channel input source.
   stereo: true
})

// Other Stage implementation code
```

Android

```
StageAudioConfiguration config = new StageAudioConfiguration();

// Update Max Bitrate to 96Kbps. Default is 64Kbps.
config.setMaxBitrate(96000);

AudioLocalStageStream microphoneStream = new AudioLocalStageStream(microphone, config);

// Other Stage implementation code
```

iOS

```
let config = IVSLocalStageStreamConfiguration();

// Update Max Bitrate to 96Kbps. Default is 64Kbps.
try! config.audio.setMaxBitrate(96000);

let microphoneStream = IVSLocalStageStream(device: microphone, config: config);
```

```
// Other Stage implementation code
```

Suggested Optimizations

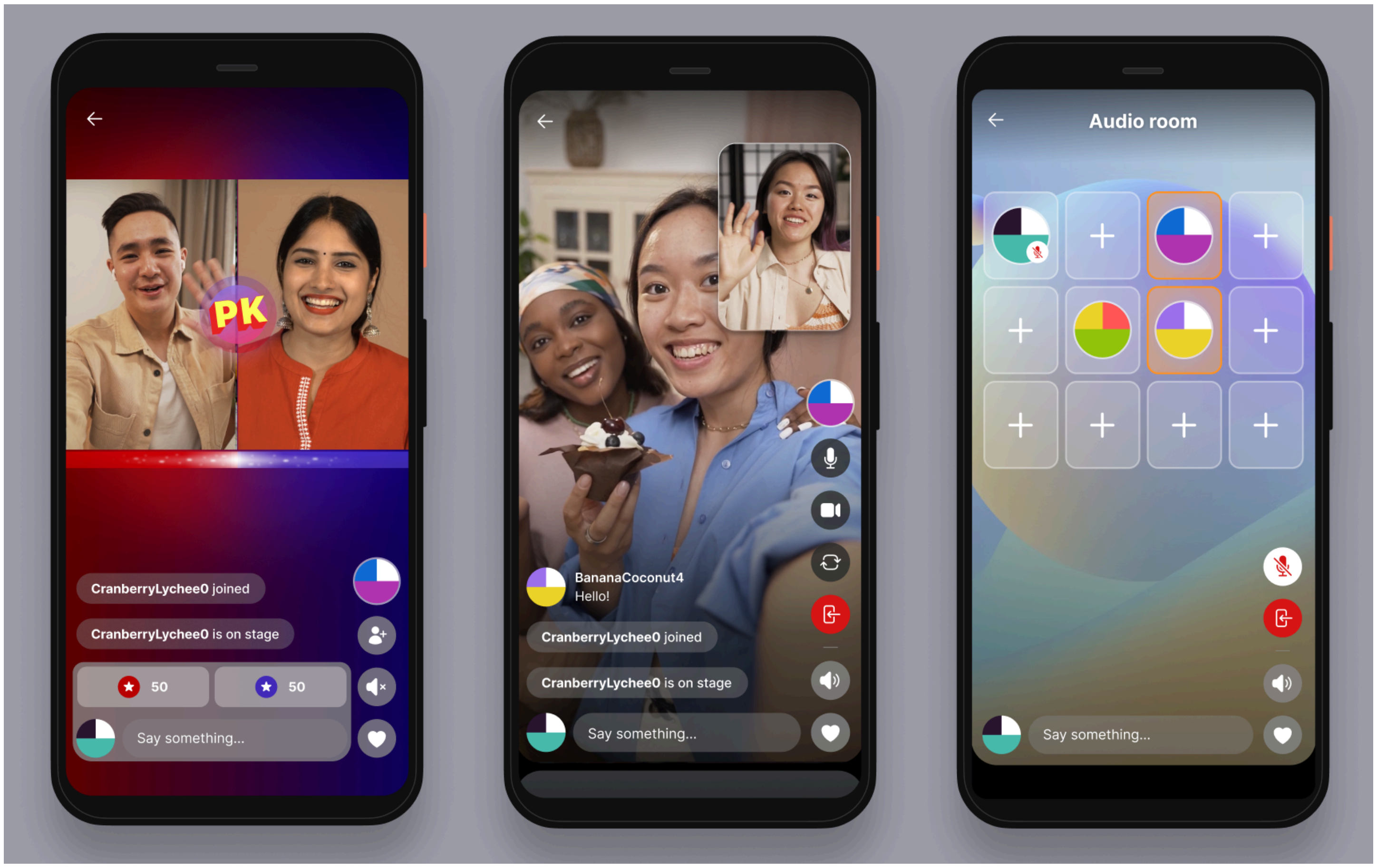
Scenario	Recommendations
Streams with text, or slow moving content, like presentations or slides	Use layered encoding with simulcast or configure streams with lower framerate.
Streams with action or a lot of movement	Use layered encoding with simulcast.
Streams with conversation or little movement	Use layered encoding with simulcast or choose audio-only (see "Subscribing to Participants" in the Real-Time Streaming Broadcast SDK Guides: Web, Android, and iOS).
Users streaming with limited data	Use layered encoding with simulcast or, if you want lower data usage for everyone, configure a lower framerate and lower the bitrate manually.

Resources and Support (Real-Time Streaming)

Resources

https://ivs.rocks/ is a dedicated site to browse published content (demos, code samples, blog posts), estimate cost, and experience Amazon IVS through live demos.

Demos



The IVS real-time streaming demo for iOS and Android shows developers how to use Amazon IVS to build a compelling real-time, social-user-generated content application. This application features a scrollable feed of user-generated real-time streams. Users can create video streams and audio-only rooms. Video-stream guests can join in guest spot or versus (VS) mode. Instructions on how to deploy the required backend and build the application are available in the following GitHub repositories:

- iOS: https://github.com/aws-samples/amazon-ivs-real-time-for-ios-demo/

- Android: https://github.com/aws-samples/amazon-ivs-real-time-for-android-demo/
- Backend: https://github.com/aws-samples/amazon-ivs-real-time-serverless-demo/

Support

The AWS Support Center offers a range of plans that provide access to tools and expertise to support your AWS solutions. All support plans provide 24/7 access to customer service. For technical support and more resources to plan, deploy, and improve your AWS environment, choose a support plan that best aligns with your AWS use case.

AWS Premium Support is a one-on-one, fast-response support channel to help you build and run applications on AWS.

AWS re:Post is a community-based Q&A site for developers to discuss technical questions related to Amazon IVS.

Contact Us has links for nontechnical inquiries about your billing or account. For technical questions, use the discussion forums or support links above.

Glossary

Also see the AWS glossary. In the table below, LL stands for IVS low-latency streaming; RT, IVS real-time streaming.

Term	Description	LL	RT	Chat
AAC	Advanced Audio Coding. AAC is an audio coding standard for lossy digital audio compression. Designed to be the successor of the MP3 format, AAC generally achieves higher sound quality than MP3 at the same bitrate. AAC has been standardi zed by ISO and IEC as part of the MPEG-2 and MPEG-4 specifications.	✓	✓	
Adaptive bitrate streaming	Adaptive Bitrate (ABR) streaming allows the IVS player to switch to a lower bitrate when connectio n quality suffers, and to switch back to a higher bitrate when connection quality improves.	✓		
Adaptive streaming	See Layered encoding with simulcast.		✓	
Administrative user	An AWS user with administrative access to resources and services available in an AWS account. See Terminology in *AWS Setup User Guide*.	✓	✓	✓
ARN	Amazon Resource Name, a unique identifier for an AWS resource. Specific ARN formats depend on the resource type. For ARN formats used by IVS resources, see in *Service Authorization Reference*.	✓	✓	✓
Aspect ratio	Describes the ratio of frame width to frame height. For example, 16:9 is the aspect ratio that corresponds to the Full HD or 1080p resolution.	✓	✓	
Audio mode	A preset or custom audio configuration optimized for different types of mobile device users and the		✓	

Term	Description	LL	RT	Chat
	equipment that they use. See IVS Broadcast SDK: Mobile Audio Modes (Real-Time Streaming).			
AVC, H.264, MPEG-4 Part 10	Advanced Video Coding, also referred to as H.264 or MPEG-4 Part 10, a video compression standard for lossy digital video compression.	✓	✓	
Background replacement	A type of camera filter that enables live-stre am creators to change their backgrounds. See Background Replacement in *IVS Broadcast SDK: Third-Party Camera Filters (Real-Time Streaming).*		✓	
Bitrate	A streaming metric for the number of bits transmitted or received per second.	✓	✓	
Broadcast, broadcaster	Other terms for stream, streamer.	✓		
Buffering	A condition that occurs when the playback device is unable to download the content before the content is supposed to be played. Buffering can manifest in several ways: content may randomly stop and start (also known as stuttering), content may stop for long periods of time (also known as freezing), or the IVS player may pause playback.	✓	✓	
Byte-range playlist	A more granular playlist than the standard HLS playlist. The standard HLS playlist is made up of 10-second media files. With a byte-range playlist, the segment duration is the same as the keyframe interval configured for the stream. Byte-range playlist is available only for the broadcasts that were auto-recorded to an S3 bucket. It is created in addition to the HLS playlist. See Byte-Range Playlists in *Auto-Record to Amazon S3 (Low-Latency Streaming).*	✓		

Term	Description	LL	RT	Chat
CBR	Constant Bitrate, a rate-control method for encoders that maintains a consistent bitrate throughout the entire playback of a video, regardless of what is happening during the broadcast. Lulls in the action may be padded to achieve the desired bitrate, and peaks may be quantized by adjusting the quality of encoding to match the target bitrate. *We strongly recommend using CBR instead of VBR*.	✓	✓	
CDN	Content Delivery Network or Content Distribut ion Network, a geographically distributed solution that optimizes delivery of content such as streaming video by bringing it closer to where users are located.	✓		
Channel	An IVS resource that stores configuration for streaming, including an ingest server, a stream key, a playback URL, and recording options. Streamers use the stream key associated with a channel to start a broadcast. All metrics and events generated during a broadcast are associate d with a channel resource.	✓		
Channel type	Determines the allowable resolution and frame rate for the channel. See Channel Types in the *IVS Low-Latency Streaming API Reference*.	✓		
Chat logging	An advanced option that can be enabled by associating a logging configuration with a chat room.			✓

Term	Description	LL	RT	Chat
Chat room	An IVS resource that stores configuration for a chat session, including optional features such as Message Review Handler and Chat Logging. See Step 2: Create a Chat Room in *Getting Started with IVS Chat*.			✓
Client-side composition	Uses a host device to mix audio and video streams from stage participants and then sends them as a composite stream to an IVS channel. This allows more control over the look of the compositi on at the cost of higher utilization of client resources and a higher risk of a stage or a host issue impacting the viewers. Also see server-side composition.	✓	✓	
CloudFront	A CDN service provided by Amazon.	✓		
CloudTrail	An AWS service for collecting, monitoring, analyzing, and retaining events and account activity from AWS and external sources. See Logging IVS API Calls with AWS CloudTrail.	✓	✓	✓
CloudWatch	An AWS service for monitoring applications, responding to performance changes, optimizing resource use, and providing insights into operation al health. You can use CloudWatch to monitor IVS metrics; see Monitoring IVS Real-Time Streaming and Monitoring IVS Low-Latency Streaming.	✓	✓	✓
Composition	The process of combining audio and video streams from multiple sources into a single stream.	✓	✓	
Composition pipeline	A sequence of processing steps required to combine multiple streams and encode the resulting stream.	✓	✓	

Term	Description	LL	RT	Chat
Compression	Encoding of information using fewer bits than the original representation. Any particular compression is either lossless or lossy. Lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression. Lossy compression reduces bits by removing unnecessary or less important information.	✓	✓	
Control plane	Stores information about IVS resources such as channels, stages, or chat rooms and provides interfaces for creating and managing these resources. It is regional (based on AWS regions).	✓	✓	✓
CORS	Cross-Origin Resource Sharing, an AWS feature that allows client web applications that are loaded in one domain to interact with resources such as S3 buckets in a different domain. Access can be configured based on headers, HTTP methods, and origin domains. See Using cross-origin resource sharing (CORS) - Amazon Simple Storage Service in *Amazon Simple Storage Service User Guide*.	✓		
Custom image source	An interface provided by the IVS Broadcast SDK that allows an application to provide its own image input instead of being limited to the preset cameras.	✓	✓	
Data plane	The infrastructure that carries data from ingest to egress. It operates based on the configuration managed in the control plane and is not restricted to an AWS region.	✓	✓	✓
Encoder, encoding	The process of converting video and audio content into a digital format, suitable for streaming. Encoding can be hardware or software based.	✓	✓	

Term	Description	LL	RT	Chat
Event	An automatic notification published by IVS to the AmazonEventBridge monitoring service. An event represents a state or health change of a streaming resource such as a stage or a composition pipeline. See Using Amazon EventBridge with IVS Low-Latency Streaming and Using Amazon EventBridge with IVS Real-Time Streaming.	✓	✓	✓
FFmpeg	A free and open-source software project consisting of a suite of libraries and programs for handling video and audio files and streams. FFmpeg provides a cross-platform solution to record, convert and stream audio and video.	✓		
Fragmented stream	Created when a broadcast disconnects and then reconnects within the interval specified in the channel's recording configuration. The resulting multiple streams are considered a single broadcast and merged together into a single recorded stream. See Merge Fragmented Streams in *Auto-Record to Amazon S3 (Low-Latency Streaming)*.	✓		
Frame rate	A streaming metric for the number of video frames transmitted or received per second.	✓	✓	
HLS	HTTP Live Streaming (HLS), an HTTP-based adaptive bitrate streaming communications protocol used to deliver IVS streams to viewers.	✓		
HLS playlist	A list of media segments that make up a stream. Standard HLS playlists are made up of 10-second media files. HLS also supports more granular byte-range playlists.	✓		
Host	A real-time event participant who sends video and/or audio to the stage.		✓	

Term	Description	LL	RT	Chat
IAM	Identity and Access Management, an AWS service that allows users to securely manage identitie s and access to AWS services and resources, including IVS.	✓	✓	✓
Ingest	IVS process for receiving video streams from a host or broadcaster for processing or delivery to viewers or other participants.	✓	✓	
Ingest server	Receives video streams and delivers them to a transcoding system, where streams are transmuxe d or transcoded into HLS for delivery to viewers. Ingest servers are specific IVS components that receive streams for channels, along with an ingestion protocol (RTMP, RTMPS). See the information on creating a channel in Getting Started with IVS Low-Latency Streaming.		✓	
Interlaced video	Transmits and displays only odd or even lines of subsequent frames to create perceived doubling of frame rate without consuming extra bandwidth. We do not recommend using interlaced video due to the video quality concerns.	✓	✓	
JSON	JavaScript Object Notation, an open-standard file format that uses human-readable text to transmit data objects consisting of attribute-value pairs and array data types or other serializable values.	✓	✓	✓

Term	Description	LL	RT	Chat
Keyframe, delta frame, keyframe interval	The keyframe (also referred to as intra-cod ed or i-frame) is a full frame of the image in a video. Subsequent frames, the delta frames (also referred to as predicted or p-frames), only contain the information that has changed. Keyframes will appear multiple times within a stream, depending on the keyframe interval defined in the encoder.	✓	✓	
Lambda	An AWS service for running code (referred to as Lambda functions) without provisioning any server infrastructure. Lambda functions can run in response to events and invocation requests, or based on a schedule. For example, IVS Chat uses Lambda functions to enable message review for a chat room.	✓	✓	✓
Latency, glass-to-glass latency	A delay in data transfer. IVS defines latency ranges as: • Low latency: under 3 sec • Real-time latency: under 300 ms *Glass-to-glass* latency refers to the delay from when a camera captures a live stream to when the stream appears on a viewer's screen.	✓	✓	
Layered encoding with simulcast	Enables simultaneous encoding and publishing of multiple video streams with different quality levels. See Adaptive Streaming: Layered Encoding with Simulcast in *Real-Time Streaming Optimizat ions*.		✓	

Term	Description	LL	RT	Chat
Message review handler	Enables IVS Chat customers to automatically review/filter user chat messages before they are delivered to the chat room. It is enabled by associating a Lambda function with a chat room. See Creating a Lambda Function in *Chat Message Review Handler*.			✓
Mixer	A feature of the IVS Mobile Broadcast SDKs that takes multiple audio and video sources and generates a single output. It supports managemen t of on-screen video and audio elements represent ing sources such as cameras, microphones, screen captures, and audio and video generated by the application. The output can then be streamed to IVS. See Configuring a Broadcast Session for Mixing in *IVS Broadcast SDK: Mixer Guide (Low-Latency Streaming)*.	✓		
Multi-host streaming	Combines streams from multiple hosts into a single stream. This can be accomplished using either client-side or server-side composition. Multi-host streaming enables scenarios such as inviting viewers onto a stage for Q&A, competiti ons between hosts, video chat, and hosts conversing with each other in front of a large audience.		✓	
Multivariant playlist	An index of all the variant streams available for a broadcast.	✓		
OAC	Origin Access Control, a mechanism for restricti ng access to an S3 bucket, so that content such as a recorded stream can be served only through CloudFront CDN.	✓		

Term	Description	LL	RT	Chat
OBS	Open Broadcaster Software, free and open source software for video recording and live streaming . OBS offers an alternative (to the IVS broadcast SDK) for desktop publishing. More sophisticated streamers familiar with OBS may prefer it for its advanced production features, such as scene transitions, audio mixing, and overlay graphics.	✓	✓	
Participant	A real-time user connected to a stage as a host or viewer.		✓	
Participant token	Authenticates a real-time event participant when they join a stage. A participant token also controls whether a participant can send video to the stage.		✓	
Playback token, playback key pair	An authorization mechanism that allows customers to restrict video playback on private channels. Playback tokens are generated from a playback key pair. A playback key pair is the public-private pair of keys used to sign and validate the viewer authorization token for playback. See Create or Import a Playback Key in *Setting up Private Channels* and see the Playback Key Pair endpoints in the IVS Low-Latency API Reference.	✓		
Playback URL	Identifies the address a viewer uses to start playback for a specific channel. This address can be used globally. IVS automatically selects the best location on the IVS global content delivery network for delivering the video to each viewer. See the information on creating a channel in Getting Started with IVS Low-Latency Streaming.	✓		

Term	Description	LL	RT	Chat
Private channel	Allows customers to restrict access to their streams using an authorization mechanism based on playback tokens. See Workflow for Private Channels in *Setting up Private Channels*.	✓		
Progressive video	Transmits and displays all lines of each frame in sequence. We recommend using progressive video during all stages of a broadcast.	✓	✓	
Quotas	The maximum numbers of IVS service resources or operations for your AWS account. That is, these limits are per AWS account, unless noted otherwise. All quotas are enforced per region. See Amazon Interactive Video Service endpoints and quotas in *AWS General Reference Guide*.	✓	✓	✓
Regions	Provide access to AWS services that physicall y reside in a specific geographic area. Regions provide fault tolerance, stability, and resilience, and can also reduce latency. With Regions, you can create redundant resources that remain available and unaffected by a regional outage. Most AWS service requests are associated with a particular geographic region. The resources that you create in one region do not exist in any other region unless you explicitly use a replication feature offered by an AWS service. For example, Amazon S3 supports cross-region replication. Some services, such as IAM, do not have cross-reg ional resources.	✓	✓	✓
Resolution	Describes the number of pixels in a single video frame, for example, Full HD or 1080p defines a frame with 1920x1080 pixels.	✓	✓	

Term	Description	LL	RT	Chat
Root user	The owner of an AWS account. The root user has complete access to all AWS services and resources in the AWS account.	✓	✓	✓
RTMP, RTMPS	Real-Time Messaging Protocol, an industry standard for transmitting audio, video, and data over a network. RTMPS is the secure version of RTMP, running over a Transport Layer Security (TLS/SSL) connection.	✓	✓	
S3 bucket	A collection of objects stored in Amazon S3. Many policies, including access and replication, are defined at the bucket level and apply to all objects in the bucket. For example, an IVS broadcast is stored as multiple objects in an S3 bucket.	✓		
SDK	Software Development Kit, a collection of libraries for the developers building applications with IVS.	✓	✓	✓
Selfie segmentation	Enables replacing the background in a live stream, using a client-specific solution that accepts a camera image as input and returns a mask that provides a confidence score for each pixel of the image, indicating whether it is in the foreground or the background. See Background Replacement in *IVS Broadcast SDK: Third-Party Camera Filters (Real-Time Streaming)*.		✓	
Semantic versioning	A version format in the form of Major.Minor.Patch. Bug fixes not affecting the API increment the patch version, backward compatible API additions/changes increment the minor version, and backward incompatible API changes increment the major version.	✓	✓	✓

Term	Description	LL	RT	Chat
Server-side composition	Uses an IVS server to mix audio and video from stage participants and then sends this mixed video to an IVS channel to reach a larger audience or to store it in an S3 bucket. Server-side compositi on reduces client load, improves resilience of the broadcast, and enables more efficient use of bandwidth. Also see client-side composition.		✓	
Service quotas	An AWS service that helps you manage your quotas for many AWS services from one location. Along with looking up the quota values, you can also request a quota increase from the Service Quotas console.	✓	✓	✓
Service-linked role	A unique type of IAM role that is linked directly to an AWS service. Service-linked roles are automatic ally created by IVS and include all the permissio ns that the service requires to call other AWS services on your behalf, for example, to access an S3 bucket. See Using Service-Linked Roles for IVS in *IVS Security*.	✓		
Stage	An IVS resource that represents a virtual space where real-time event participants can exchange video in real time. See Create a Stage in *Getting Started with IVS Real-Time Streaming*.		✓	
Stage session	Begins when the first participant joins a stage and ends a few minutes after the last participant stops publishing to the stage. A long-lived stage may have multiple sessions over its lifetime.		✓	
Stream	Data representing video or audio content being sent continuously from a source to a destination.	✓	✓	

Term	Description	LL	RT	Chat
Stream key	An identifier assigned by IVS when you create a channel; it is used to authorize streaming to the channel. **Treat the stream key like a secret, since anyone with it can stream to the channel.** See Getting Started with IVS Low-Latency Streaming.	✓		
Stream starvation	A delay or halt in stream delivery to IVS. It occurs when IVS does not receive the expected amount of bits that the encoding device advertise d it would send over a certain timeframe. An occurrence of stream starvation results in a stream starvation event. From a viewer's perspective, stream starvatio n may appear as video that lags, buffers, or freezes. Stream starvation can be brief (less than 5 seconds) or long (several minutes), depending on the specific situation that resulted in stream starvation. See What is Stream Starvation in *Troubleshooting FAQ*.	✓	✓	
Streamer	A person or a device sending a video or audio stream to IVS.	✓	✓	
Subscriber	A real-time event participant who receives video and/or audio of the hosts. See What is IVS Real-Time Streaming.		✓	
Tag	A metadata label that you assign to an AWS resource. Tags can help you identify and organize your AWS resources. On the IVS documentation landing page, see “Tagging” in any of the IVS API documentation (for real-time streaming, low-laten cy streaming, or chat).	✓	✓	✓

Term	Description	LL	RT	Chat
Third-party camera filters	Software components that can be integrated with the IVS Broadcast SDK to allow an application to process images before providing them to the Broadcast SDK as a custom image source. A third-party camera filter may process images from the camera, apply a filter effect, etc.	✓	✓	
Thumbnail	A reduced-size image taken from a stream. By default, thumbnails are generated every 60 seconds, but a shorter interval can be configure d. Thumbnail resolution depends on the channel type. See Recording Contents in *Auto-Record to Amazon S3 (Low-Latency Streaming)*.	✓		
Timed metadata	Metadata tied to specific timestamps within a stream. It can be added programmatically using the IVS API and becomes associated with specific frames. This ensures that all viewers receive the metadata at the same point relative to the stream. Timed metadata can be used to trigger actions on the client such as updating team statistics during a sporting event. See Embedding Metadata within a Video Stream.	✓		
Transcoding	Converts video and audio from one format to another. An incoming stream may be transcode d to a different format at multiple bitrates and resolutions, to support a range of playback devices and network conditions.	✓	✓	

Term	Description	LL	RT	Chat
Transmuxing	A simple repackaging of an ingested stream to IVS, with no re-encoding of the video stream. “Transmux” is short for transcode multiplexing, a process that changes the format of an audio and/or video file while keeping some or all of the original streams. Transmuxing converts to a different container format without changing the file contents. Distinguished from transcoding.	✓	✓	
Variant streams	A set of encodings of the same broadcast in several distinct quality levels. Each variant stream is encoded as a separate HLS playlist. An index of the available variant streams is referred to as a multivariant playlist. After the IVS player receives a multivariant playlist from IVS, it can then choose between the variant streams during playback, changing back and forth seamlessly as network conditions change.	✓		
VBR	Variable Bitrate, a rate-control method for encoders that uses a dynamic bitrate that changes throughout playback, depending on the level of detail needed. We strongly recommend against using VBR due to video-quality concerns; use CBR instead.	✓	✓	

Term	Description	LL	RT	Chat
View	A unique viewing session which is actively downloading or playing video. Views are the basis for the concurrent views quota. A view starts when a viewing session begins video playback. A view ends when a viewing session stops video playback. Playback is the sole indicator of viewership; engagement heuristic s such as audio levels, browser tab focus, and video quality are not considered. When counting views, IVS does not consider the legitimacy of individual viewers or try to deduplicate localized viewership, such as multiple video players on a single machine. See Other Quotas in *Service Quotas (Low-Latency Streaming)*.	✓		
Viewer	A person receiving a stream from IVS.	✓		
WebRTC	Web Real-Time Communication, an open-sour ce project providing web browsers and mobile applications with real-time communication. It allows audio and video communication to work inside web pages by allowing direct peer-to-peer communication, eliminating the need to install plugins or download native apps. The technologies behind WebRTC are implement ed as an open web standard and are available as regular JavaScript APIs in all major browsers or as libraries for native clients, like Android and iOS.	✓	✓	

Term	Description	LL	RT	Chat
WHIP	WebRTC-HTTP Ingestion Protocol, an HTTP based protocol that allows WebRTC based ingestion of content into streaming services and/or CDNs. WHIP is an IETF draft developed to standardize WebRTC ingestion. WHIP enables compatibility with software like OBS, offering an alternative (to the IVS broadcast SDK) for desktop publishing. More sophisticated streamers familiar with OBS may prefer it for its advanced production features, such as scene transitions, audio mixing, and overlay graphics WHIP is also beneficial in situations where using the IVS broadcast SDK isn't feasible or preferred . For example, in setups involving hardware encoders, the IVS broadcast SDK might not be an option. However, if the encoder supports WHIP, you can still publish directly from the encoder to IVS. See OBS and WHIP Support.		✓	
WSS	WebSocket Secure, a protocol for establishing WebSockets over an encrypted TLS connection. It is being used for connecting to IVS Chat endpoints . See Step 4: Send and Receive Your First Message in *Getting Started with IVS Chat*.			✓

Document History (Real-Time Streaming)

Real-Time Streaming User Guide Changes

Change	Description	Date
Broadcast SDK: Web 1.11.0	Updated version number and artifact links on the IVS documentation landing page and in the real-time-streamin g broadcast SDK guide: Web. Also see the Release Notes.	May 6, 2024
Broadcast SDK: Web 1.10.1	Updated version number and artifact links on the IVS documentation landing page and in the real-time-streamin g broadcast SDK guide: Web. Also see the Release Notes.	April 30, 2024
Broadcast SDK: Android 1.15.2, iOS 1.15.2	Updated version number and artifact links on the IVS documentation landing page and in the real-time-streamin g broadcast SDK guides: Android and iOS. Also see the Release Notes.	April 30, 2024
Broadcast SDK: iOS Guide	In Publish a Media Stream, we updated the code example.	April 26, 2024
Broadcast SDK: Android 1.17.0, iOS 1.17.0	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guides: Android and iOS. On the Amazon IVS documenta	April 22, 2024

	tion landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release.	
Server-side composition	In SSC, made various changes, especially in "Layout," to explain PiP and grid layouts. In the Web Broadcast SDK Guide, added Server-Side Rendering Support.	March 26, 2024
OBS and WHIP Support	Added a note about quality issues (like intermittent video freezing) that can occur with WHIP in OBS.	March 22, 2024
Broadcast SDK: Android 1.16.0, iOS 1.16.0, Web 1.10.0	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guides: Android, iOS, and Web. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release.	March 21, 2024

Broadcast SDK: Android 1.15.1, iOS 1.15.1	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guides: Android and iOS. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release.	March 13, 2024
Broadcast SDK: Mobile Audio Modes	In "Audio Mode Presets," added information on the Volume Rocker preset category and an iOS known issue with the Video Chat preset. In "Advanced Use Cases," added a note on avoiding incorrect configurations, and added sections on "iOS Echo Cancellation" and "iOS Custom Audio Sources."	March 1, 2024

Broadcast SDK: Android 1.15.0, iOS 1.15.0, Web 1.9.0	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guides: Android, iOS, and Web. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release.	February 22, 2024
OBS and WHIP Support	Added a new page. This document explains how to use WHIP-compatible encoders like OBS to publish to IVS real-time streaming . WHIP (WebRTC-HTTP Ingestion Protocol) is an IETF draft developed to standardi ze WebRTC ingestion.	February 6, 2024

Broadcast SDK: Android 1.14.1, iOS 1.14.1, Web 1.8.0	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guides: Android, iOS, and Web. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release. For the Android Guide, we added a new Known Issue (video size less than 176x176). For the Web Guide, we added a new Known Issue. The workaround is constraining video resolution to 720p when invoking `getUserMedia` or `getDisplayMedia` . In *Real-Time Streaming Optimizations* we updated Configuring Layered Encoding with Simulcast; now this is disabled by default.	February 1, 2024

Broadcast SDK: Android 1.13.4, iOS 1.13.4, Web 1.7.0	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guides: Android, iOS, and Web. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release.	January 3, 2024
IVS Glossary	Extended the glossary, covering IVS real-time, low-latency, and chat terms.	December 20, 2023
Stage Health: New CloudWatch Metrics	Renamed the PacketLoss (Stage) metric to be DownloadPacketLoss (Stage) and released additional CloudWatch metrics for IVS real-time streaming: • DownloadPacketLoss (Stage,Participant) • DroppedFrames (Stage,Participant) • SubscribeBitrate (Stage,Participant,MediaType) See Monitoring IVS Real-Time Streaming.	December 7, 2023

IAM managed policies	Added two managed policies, IVSReadOnlyAccess and IVSFullAccess. See: • The new section on Managed Policies for Amazon IVS on the *Security* page. • Changes to Step 3: Set Up IAM Permissions in *Getting Started with IVS Low-Laten cy Streaming*.	December 5, 2023
Broadcast SDK: Android 1.13.2, iOS 1.13.2	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guides: Android and iOS. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release.	December 4, 2023

Broadcast SDK: Android 1.13.1	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guide: Android. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release.	November 21, 2023
Service Quotas	Changed "Participant publish resolution" from 1080p to 720p.	November 18, 2023

Broadcast SDK: Android 1.13.0, iOS 1.13.0	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guides: Android and iOS. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release. We also made various updates to Streaming Optimizations. Among other things, the "Adaptive Streaming: Layered Encoding with Simulcast" feature now requires explicit opt-in and is supported only in recent versions of the SDK.	November 17, 2023
Composite Recording	Made the following changes: • Added a Composite Recording page for this new feature. • Updated Getting Started with IVS Real-Time Streaming with S3 endpoints in the policy in "Set Up IAM Permissions." • Updated Service Quotas with call-rate quotas for the new endpoints.	November 16, 2023

Server-side composition (SSC)	IVS server-side composition enables clients to offload the composition and broadcast ing of an IVS stage to an IVS-managed service. SSC and RTMP broadcast to a channel are invoked through IVS control-plane endpoints in the stage's home region. See: • Getting Started – We added SSC endpoints to the policy in "Set Up IAM Permissio ns." • Using Amazon EventBridge with IVS – We added new metrics. • Server-Side Compositi on – This new document includes an overview and setup instructions. • Service Quotas – We added new call-rate limits and other quotas. Also see: • Changes listed below in IVS Real-Time Streaming API Reference Changes. • Changes listed in Document History (Low-Latency Streaming).	November 16, 2023

IVS broadcast SDK	In the Broadcast SDK overview, we updated Platform Requirements > Native Platforms to clarify which SDK versions are supported and we added "Mobile Browsers (iOS and Android)." In the Broadcast Web Guide, we added "Mobile Web Limitations."	November 9, 2023
IVS broadcast SDK	We added a new page on Third-Party Camera Filters.	November 9, 2023
Getting Started with IVS Real-Time Streaming	We updated procedures in Set Up IAM Permissions.	October 20, 2023
Monitoring Real-Time Streaming	In CloudWatch Metrics: IVS Real-Time Streaming, we added sample values for dimensions.	October 17, 2023
Broadcast SDK: Web Guide	We made several changes to Monitor Remote Participant Media Mute State.	October 17, 2023

Broadcast SDK: Web 1.6.0	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guide: Web. The Amazon IVS documentation landing page points to the current version of Broadcast SDK References. Also see the Amazon IVS Release Notes for this release. In the Web Guide, in "Retrieve a MediaStream from a Device," we also deleted the two `max` lines; best practice is to specify only `ideal`. In Real-Time Streaming Optimizations, we added a new section, Optimizing Audio Bitrate and Stereo Support.	October 16, 2023
Stage Health: New CloudWatch Metrics	Released CloudWatch metrics for IVS real-time streaming. See Monitoring IVS Real-Time Streaming.	October 12, 2023

Broadcast SDK: Android 1.12.1	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guide: Android. Also added a new section, Using Bluetooth Microphones. The Amazon IVS documentation landing page points to the current version of Broadcast SDK References. Also see the Amazon IVS Release Notes for this release.	October 12, 2023
Broadcast SDK: Web 1.5.2	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guide: Web. The Amazon IVS documentation landing page points to the current version of Broadcast SDK References. Also see the Amazon IVS Release Notes for this release.	September 14, 2023
Getting Started with IVS Real-Time Streaming	In Android > Install the Broadcast SDK, added data binding.	September 12, 2023
Broadcast SDK error handling	Added "Error Handling" sections to the Broadcast SDK Guides: Web, Android, and iOS.	September 12, 2023

Getting Started with IVS Real-Time Streaming	In Distribute Participant Tokens, added an **Important** note about not building functionality based on current token format.	September 1, 2023
Getting Started with IVS Real-Time Streaming	In Set Up IAM Permissions, updated the set of permissio ns.	August 31, 2023
Broadcast SDK: Web 1.5.1, Android 1.12.0, and iOS 1.12.0	Updated version number and artifact links for the new release, in the real-time-streaming broadcast SDK guides: Web, Android, and iOS. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release.	August 23, 2023

Real-time streaming launch	Major documentation changes accompany this release. We renamed the previous documentation to be IVS Low-Latency Streaming and published new IVS Real-Time Streaming documenta tion. The IVS documenta tion landing page now has separate sections for real-time streaming and low-laten cy streaming. Each section has its own User Guide and API Reference. For other documentation changes, see Document History (Low-Latency Streaming).	August 7, 2023
Broadcast SDK: Web 1.5.0, Android 1.11.0, and iOS 1.11.0	Updated version number and artifact links for the new release, in the broadcast SDK guides: Web, Android, and iOS. On the Amazon IVS documentation landing page, updated the broadcast SDK Reference links to point to the new version. Also see the Amazon IVS Release Notes for this release.	August 7, 2023

IVS Real-Time Streaming API Reference Changes

API Change	Description	Date
Remove `svs` from ARN patterns	ARN patterns which specified `[is]vs` were updated to specify `ivs`. This affects all three Tag endpoints and the `ChannelDestinationConfiguration$channelArn` field.	April 25, 2024
Server-side composition updates	We added one object: PipConfiguration. We modified two objects (LayoutConfiguration, GridConfiguration). This affects the GetComposition response and the StartComposition request and response.	March 13, 2024
Composite recording	We added 4 StorageConfiguration endpoints and 7 objects (DestinationDetail, RecordingConfiguration, S3DestinationConfiguration, S3Detail, S3StorageConfiguration, StorageConfiguration, StorageConfigurationSummary). We modified 3 objects (Composition, Destination, DestinationConfiguration). This affects the GetComposition response and the StartComposition request and response.	November 16, 2023
Server-side composition	We added 8 Composition and EncoderConfiguration endpoints and 11 objects (ChannelDestinationConfiguration, Composition, CompositionSummary, Destination, DestinationConfiguration, DestinationSummary, EncoderConfiguration, EncoderConfigurationSummary, GridConfiguration, LayoutConfiguration, and Video).	November 16, 2023
Stage Health: New Participant Data	Added six fields to the Participant object: `browserName` , `browserVersion` , `ispName`,	October 12, 2023

API Change	Description	Date
	`osName`, `osVersion`, and `sdkVersion`. This affects the GetParticipant response.	
Participant Token	Added an **Important** note about not building functionality based on current token format.	September 1, 2023
IVS Real-Time Streaming launch	Major documentation changes accompany this release. We renamed the previous documentation to be IVS Low-Latency Streaming and published new IVS Real-Time Streaming documentation. The IVS documentation landing page now has separate sections for real-time streaming and low-latency streaming. Each section has its own User Guide and API Reference. IVS Real-Time Streaming API Reference is part of IVS real-time streaming documentation. Previously it was titled IVS Stage API Reference. Its prior history is described in Document History (Low-Latency Streaming).	August 7, 2023

Release Notes (Real-Time Streaming)

May 6, 2024

IVS Broadcast SDK: Web 1.11.0 (Real-Time Streaming)

Platform	Downloads and Changes
Web Broadcast SDK 1.11.0	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Fixed an edge case where the SDK did not attempt to recover on a stage `DISCONNECT` . • Updated the error message for a `join()` timeout error. Instead of "InitialConnectTimedOut after 10 seconds," the SDK now returns "Operation timed out."

April 30, 2024

IVS Broadcast SDK: Web 1.10.1 (Real-Time Streaming)

Platform	Downloads and Changes
Web Broadcast SDK 1.10.1	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Minor bug fixes.

April 30, 2024

Amazon IVS Broadcast SDK: Android 1.15.2, iOS 1.15.2 (Real-Time Streaming)

Platform	Downloads and Changes
Android Broadcast SDK 1.15.2	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.15.2/android • Minor bug fixes. Upgrade to this version only if you have a specific reason to do so; otherwise, use the highest version that is released.
iOS Broadcast SDK 1.15.2	**Download for real-time streaming:** https://broadcast.live-video.net/1.15.2/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.15.2/ios • Minor bug fixes. Upgrade to this version only if you have a specific reason to do so; otherwise, use the highest version that is released.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.244 MB	13.198 MB
armeabi-v7a	4.543 MB	9.192 MB
x86_64	5.437 MB	14.051 MB

Architecture	Compressed Size	Uncompressed Size
x86	5.631 MB	14.461 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	3.359 MB	7.836 MB

April 22, 2024

Amazon IVS Broadcast SDK: Android 1.17.0, iOS 1.17.0 (Real-Time Streaming)

Platform	Downloads and Changes
Android Broadcast SDK 1.17.0	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.17.0/android • Fixed a rare crash that can occur while publishing.
iOS Broadcast SDK 1.17.0	**Download for real-time streaming:** https://broadcast.live-video.net/1.17.0/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.17.0/ios • The `AmazonIVSBroadcast` framework now includes a privacy manifest, as required by Apple.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.273 MB	13.275 MB
armeabi-v7a	4.571 MB	9.251 MB
x86_64	5.468 MB	14.137 MB
x86	5.662 MB	14.549 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	3.388 MB	7.916 MB

March 21, 2024

Amazon IVS Broadcast SDK: Android 1.16.0, iOS 1.16.0, Web 1.10.0 (Real-Time Streaming)

Platform	Downloads and Changes
Web Broadcast SDK 1.10.0	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Fixed an intermittent error when cleaning up connections after unsubscribing or leaving a stage.
Android Broadcast SDK 1.16.0	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.16.0/android

Platform	Downloads and Changes
	• Fixed a previews freeze on the Exynos variant of Samsung devices with Android 14. • Added a function for querying camera zoom capabilities and setting the zoom factor.
iOS Broadcast SDK 1.16.0	**Download for real-time streaming:** https://broadcast.live-video.net/1.16.0/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.16.0/ios • Minor bug fixes.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.253 MB	13.21 MB
armeabi-v7a	4.551 MB	9.204 MB
x86_64	5.447 MB	14.070 MB
x86	5.640 MB	14.480 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	3.361 MB	7.836 MB

March 13, 2024

Amazon IVS Broadcast SDK: Android 1.15.1, iOS 1.15.1 (Real-Time Streaming)

Platform	Downloads and Changes
Android Broadcast SDK 1.15.1	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.15.1/android • Fixed a rare crash when subscribing to a remote participant.
iOS Broadcast SDK 1.15.1	**Download for real-time streaming:** https://broadcast.live-video.net/1.15.1/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.15.1/ios • Fixed a rare crash when subscribing to a remote participant.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.243 MB	13.194 MB
armeabi-v7a	4.541 MB	9.188 MB
x86_64	5.628 MB	14.455 MB
x86	5.434 MB	14.046 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	3.358 MB	7.820 MB

March 13, 2024

Server-Side Composition API Updates

We introduced new properties to the GridConfiguration and a new picture-in-picture layout, enhancing the customization options for compositions. For specific documentation changes, see the Document History (see the table of API Reference changes).

Important: Ensure your application does not depend on the specific features of the current layout, such as size and position of tiles. *Visual improvements to layouts can be introduced at any time*.

March 8, 2024

Server-Side Composition Layout Updates

Today we enabled the changes to the default grid layout that are described in the February 7, 2024 entry.

February 22, 2024

Amazon IVS Broadcast SDK: Android 1.15.0, iOS 1.15.0, Web 1.9.0 (Real-Time Streaming)

Platform	Downloads and Changes
Web Broadcast SDK 1.9.0	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Improved internal error handling.

Platform	Downloads and Changes
Android Broadcast SDK 1.15.0	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.15.0/android • Minor bug fixes.
iOS Broadcast SDK 1.15.0	**Download for real-time streaming:** https://broadcast.live-video.net/1.15.0/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.15.0/ios • Added an `AVPictureInPictureController` extension to allow creating a new instance with an `IVSImagePreviewView`. • Added a new API on `IVSImageDevice` to create an `AVSampleBufferDisplayLayer` to which the device renders. • Fixed a low bitrate issue on devices running iOS 17 and later. • Minor bug fixes.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.243 MB	13.194 MB
armeabi-v7a	4.541 MB	9.188 MB
x86_64	5.628 MB	14.455 MB
x86	5.434 MB	14.046 MB

Broadcast SDK Size: iOS

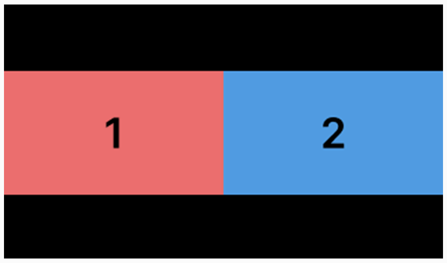
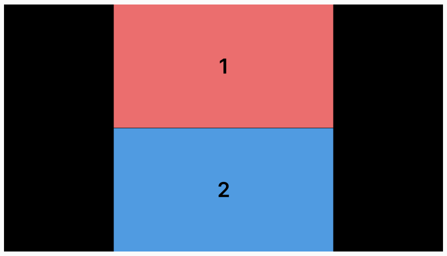
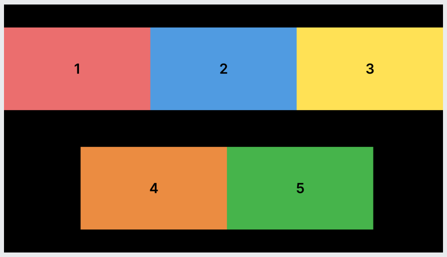
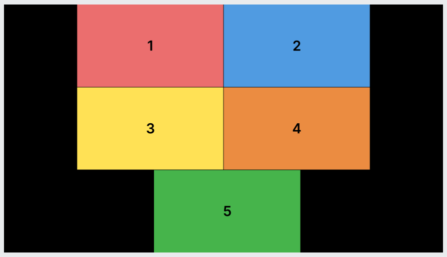
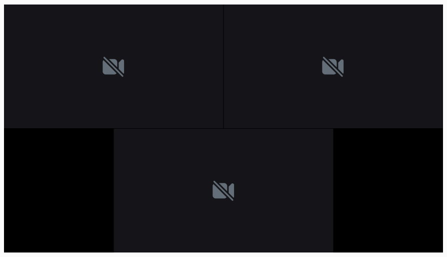
Architecture	Compressed Size	Uncompressed Size
arm64	3.358 MB	7.820 MB

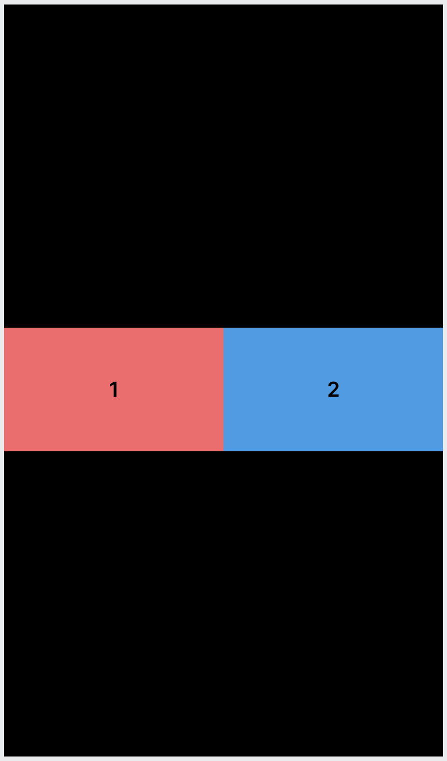
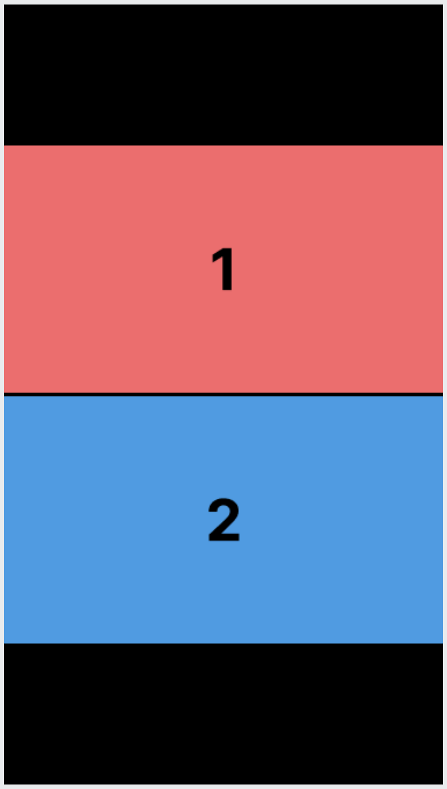
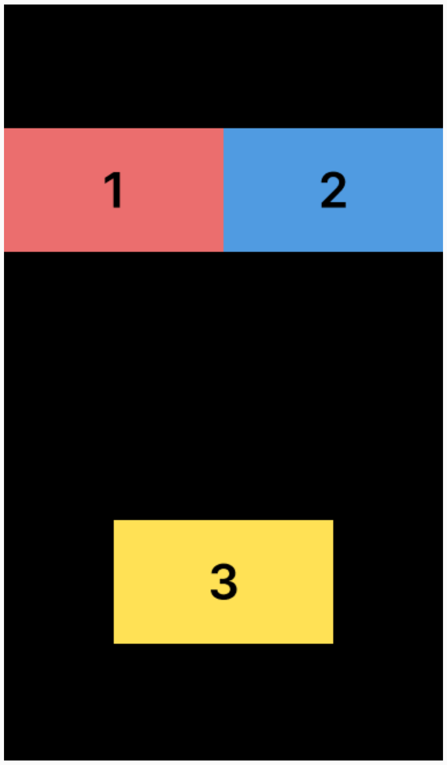
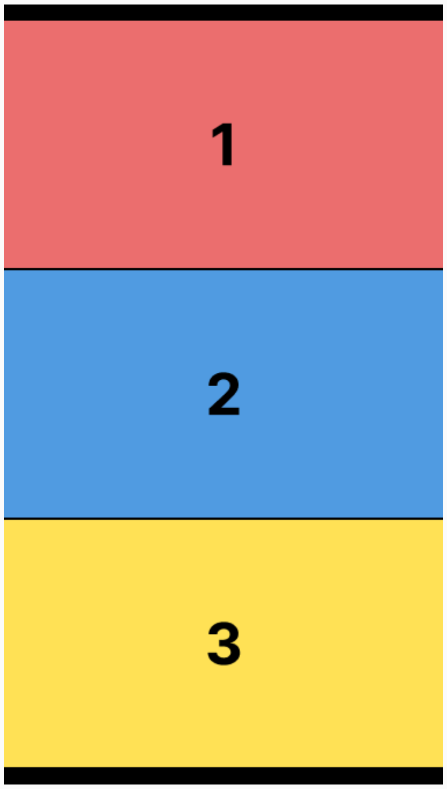
February 7, 2024

Server-Side Composition Layout Updates

This release introduces visual improvements to the default grid layout. These changes will optimize how video is displayed and reduce blank space. These changes will be enabled on March 7, 2024.

Important: Ensure your application does not depend on the specific features of the current layout, such as size and position of tiles. *Visual improvements to layouts can be introduced at any time.*

Description of the Change	Old	New
Automatically selects the optimal placement of participants to maximize video size.		
Enhances space utilization by reducing gaps and minimizing black bars.		
Adds a new “camera off” indicator for clear visibility of participants not sharing video.		

Description of the Change	Old	New
Improves space utilization and proportions for portrait use cases.		
Enhances space utilizati on in portrait use cases by minimizing spacing between participants and reducing letterboxing or pillarboxing.		

February 6, 2024

OBS and WHIP Support

IVS can be used with WHIP-compatible encoders like OBS to publish to IVS real-time streaming. WHIP (WebRTC-HTTP Ingestion Protocol) is an IETF draft developed to standardize WebRTC ingestion. See the new page on OBS and WHIP Support.

February 1, 2024

Amazon IVS Broadcast SDK: Android 1.14.1, iOS 1.14.1, Web 1.8.0 (Real-Time Streaming)

Platform	Downloads and Changes
Web Broadcast SDK 1.8.0	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Layered encoding with simulcast is now disabled by default. • Fixed an issue where a Stage instance would not cleanly disconnect when a Stage was deleted, or when a participant was disconnected from the server. The SDK now emits a `STAGE_CONNECTION_STATE_CHANGED` event with a state of `DISCONNECTED` (instead of `ERRORED` and then `CONNECTING`). • Fixed issue where publishing would fail when updating the strategy with empty audio or video tracks.
Android Broadcast SDK 1.14.1	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.14.1/android

Platform	Downloads and Changes
	• Layered encoding with simulcast is now disabled by default. • Updated `libWebRTC` from M108 to M119. • Fixed several crashes to improve overall stability. • Added support for stereo publishing. This can be enabled through the `StageAudioConfiguration` object. • Fixed a bug causing a black feed from participants after joining a session. • Updated internal `libWebRTC` references to avoid symbol conflicts when other `libWebRTC` versions are included in the same host application.
iOS Broadcast SDK 1.14.1	**Download for real-time streaming:** https://broadcast.live-video.net/1.14.1/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.14.1/ios • Layered encoding with simulcast is now disabled by default. • Updated `libWebRTC` from M108 to M119. • Fixed several crashes to improve overall stability. • Added support for stereo publishing. This can be enabled through `IVSLocalStageStreamAudioConfiguration` . • Fixed a crash when enabling audio-only mode for other participants. • Improved TTV and reduced binary size.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.223 MB	13.118 MB
armeabi-v7a	4.524 MB	9.134 MB
x86_64	5.418 MB	13.955 MB
x86	5.61 MB	14.369 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	3.350 MB	7.790 MB

January 3, 2024

Amazon IVS Broadcast SDK: Android 1.13.4, iOS 1.13.4, Web 1.7.0 (Real-Time Streaming)

Platform	Downloads and Changes
Web Broadcast SDK 1.7.0	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Improved time-to-video for subscribers joining stages. • Removed the `minAudioBitrateKbps` property (it was unused). • Improved network recovery during internet outages or changes.

Platform	Downloads and Changes
Android Broadcast SDK 1.13.4	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.13.4/android • StageAudioConfiguration now supports setting whether echo cancellation should be enabled.
iOS Broadcast SDK 1.13.4	**Download for real-time streaming:** https://broadcast.live-video.net/1.13.4/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.13.4/ios • On iOS, we improved the audio engine for both recording and playback with a focus on stability and recoverability. This enhances support for route changes while in use, improves battery recovery for edge cases, and reduces the amount of main thread blocking. • Fixed an issue where the microphone might stay active even after it was detached from a stage, leaving the iOS privacy indicator on. (The SDK was not processing incoming audio at the time.)

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.187 MB	13.025 MB
armeabi-v7a	4.491 MB	9.056 MB

Architecture	Compressed Size	Uncompressed Size
x86_64	5.359 MB	13.829 MB
x86	5.553 MB	14.214 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	3.45 MB	7.84 MB

December 7, 2023

New CloudWatch Metrics

We renamed the PacketLoss (Stage) metric to be DownloadPacketLoss (Stage). We also released additional CloudWatch metrics for IVS real-time streaming:

- DownloadPacketLoss (Stage,Participant)
- DroppedFrames (Stage,Participant)
- SubscribeBitrate (Stage,Participant,MediaType)

For details, see Monitoring IVS Real-Time Streaming.

December 4, 2023

Amazon IVS Broadcast SDK: Android 1.13.2 and iOS 1.13.2 (Real-Time Streaming)

Platform	Downloads and Changes
All mobile (Android and iOS)	• Noise-suppression configuration is available for developers to enable/disable for publishing.
Android Broadcast SDK 1.13.2	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.13.2/android • Improved the time it takes to load the video (TTV) when joining the first stage in a session.
iOS Broadcast SDK 1.13.2	**Download for real-time streaming:** https://broadcast.live-video.net/1.13.2/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.13.2/ios • No changes in the real-time SDK.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.177 MB	13.01 MB
armeabi-v7a	4.485 MB	9.045 MB
x86_64	5.352 MB	13.808 MB

Architecture	Compressed Size	Uncompressed Size
x86	5.547 MB	14.192 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	3.45 MB	7.82 MB

November 21, 2023

Amazon IVS Broadcast SDK: Android 1.13.1 (Real-Time Streaming)

Platform	Downloads and Changes
Android Broadcast SDK 1.13.1	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.13.1/android • Fixed an issue that caused a crash when quickly leaving, releasing, and rejoining the same stage.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.177 MB	13.102 MB
armeabi-v7a	4.485 MB	9.046 MB
x86_64	5.353 MB	13.809 MB
x86	5.547 MB	14.192 MB

November 17, 2023

Amazon IVS Broadcast SDK: Android 1.13.0 and iOS 1.13.0 (Real-Time Streaming)

Platform	Downloads and Changes
All mobile (Android and iOS)	• Updated Streaming Optimizations. Among other things, the "Adaptive Streaming : Layered Encoding with Simulcast" feature now requires explicit opt-in and is supported only in recent versions of the SDK. • Improved the stability of stages by reducing occurrences of rare crashes. • Improved the time it takes to load the video (TTV) when joining a stage. • Improved the experience with Bluetooth devices. • Optimized SDK CPU and memory usage, and reduced the library size. • Added the `StageAudioManager` class, which can be used to set audio capture and playback parameters, including presets for voice communication, media playback and more. For details, see the new page, IVS Broadcast SDK: Mobile Audio Modes. • Added a new `requestQualityStats` function to display structured quality events from WebRTC stats. • Added a new function to update the audio bitrate. It is set on `LocalStageStream` objects just like the video configuration, but through a new audio configuration object.

Platform	Downloads and Changes
Android Broadcast SDK 1.13.0	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.13.0/android • All methods on the `StageRenderer` interface are now optional. • Added support to `Surfaceview` -based preview for better performance. The existing `getPreview` methods in `Session` and `StageStream` continue to return a subclass of `TextureView` , but this may change in a future SDK version. ◦ If your application depends on `TextureView` specifically, you can continue with no changes. You also can switch from `getPreview` to `getPreviewTextureView` to prepare for the eventual change of what the default `getPreview` returns. ◦ If your application does not require `TextureView` specifically, we recommend switching to `getPreviewSurfaceView` for lower CPU and memory usage. • The SDK now implements a new type of preview called `ImagePreviewSurfaceTarget` which works with the application-provided Android Surface object. It is not a subclass of Android View, which provides better flexibility. • Fixed the case where `onFrame` callback for remote participant is called at the wrong time with the wrong size.

Platform	Downloads and Changes
	• `SurfaceSource # getInputSurface` is now annotated with `@Nullable`. Your code should check it before using it. • Added `UserId` and `attributes` to `ParticipantInfo`. The `UserId` and `attributes` properties are embedded in the token and applications can retrieve them via `ParticipantInfo` whenever a participant joins. • Camera capture and preview rendering now defaults to 720 x 1280 or publish resolution (whichever is greater) at 15 fps. You can adjust the resolution and/or the fps using `StageVideoConfiguration # setCameraCaptureQuality`. • `IllegalArgumentException` thrown when setting configuration properties now includes the provided value in the exception message.

Platform	Downloads and Changes
iOS Broadcast SDK 1.13.0	**Download for real-time streaming:** https://broadcast.live-video.net/1.13.0/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.13.0/ios • Fixed the issue where the SDK does not change video configuration if the video configuration is updated before publishing. • Incorporated the Google fix for a LibVPX security vulnerability (CVE-2023-5217). (Note that the Android SDK did not require any changes for this issue.) • Applications using other libraries that include `libWebRTC` will no longer have conflicts with the IVS Broadcast SDK. • All methods on the `IVSStageRenderer` protocol are now marked `@optional`. • Microphones and cameras returned by our SDKs now have a guaranteed sorting order, as documented in the SDKs themselves. • Multiple cameras can now have a value of `true` for their `isDefault` property, one for each position as determined by the operating system. • Added `IVSStageAudioManager`, which allows precise control over the underlying `AVAudioSession` to enable a wider variety of use cases for Stages functionality. • Added `UserId` to `ParticipantInfo`.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.17 MB	13.00 MB
armeabi-v7a	4.48 MB	9.04 MB
x86_64	5.35 MB	13.80 MB
x86	5.54 MB	14.18 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	3.45 MB	7.84 MB

November 16, 2023

Composite Recording

This new feature enables recording the composited view of an IVS Stage to an Amazon S3 bucket. For more information, see:

- Composite Recording – This is a new page.
- Getting Started with IVS Real-Time Streaming – We added S3 endpoints to the policy in "Set Up IAM Permissions."
- Service Quotas – We added call-rate quotas for the new endpoints.
- IVS Real-Time Streaming API Reference – We added 4 StorageConfiguration endpoints and 7 objects (DestinationDetail, RecordingConfiguration, S3DestinationConfiguration, S3Detail, S3StorageConfiguration, StorageConfiguration, StorageConfigurationSummary). We also modified 3 objects (Composition, Destination, DestinationConfiguration); this affects the GetComposition response and the StartComposition request and response.

November 16, 2023

Server-Side Composition

IVS server-side composition enables clients to offload the composition and broadcasting of an IVS stage to an IVS-managed service. Server-side composition and RTMP broadcast to a channel are invoked through IVS control plane endpoints in the stage's home region. For more information, see:

- Getting Started with IVS Real-Time Streaming – We added SSC endpoints to the policy in "Set Up IAM Permissions."
- Using Amazon EventBridge with IVS Real-Time Streaming – We added new metrics.
- Server-Side Composition – This new document includes an overview and setup instructions.
- Service Quotas (Real-Time Streaming) – We added new call-rate limits and other quotas.
- Real-Time Streaming API Reference – We added 8 Composition and EncoderConfiguration endpoints and 11 objects (ChannelDestinationConfiguration, Composition, CompositionSummary, Destination, DestinationConfiguration, DestinationSummary, EncoderConfiguration, EncoderConfigurationSummary, GridConfiguration, LayoutConfiguration, and Video).

In the *IVS Low-Latency Streaming User Guide*, see:

- Enabling Multiple Hosts on an IVS Stream – We added "Broadcasting a Stage: Client-Side versus Server-Side Composition" and updated "4. Broadcast the Stage."

October 16, 2023

Amazon IVS Broadcast SDK: Web 1.6.0 (Real-Time Streaming)

Platform	Downloads and Changes
Web Broadcast SDK 1.6.0	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Improved Time-To-Video (TTV).

Platform	Downloads and Changes
	• Added `maxAudioBitrate` configuration, supporting up to 128kbps of mono or stereo audio channels.

October 12, 2023

New CloudWatch Metrics and Participant Data

We released CloudWatch metrics for IVS real-time streaming. For details, see Monitoring IVS Real-Time Streaming.

We also added six fields to the Participant API object: `browserName`, `browserVersion`, `ispName`, `osName`, `osVersion`, and `sdkVersion`. This affects the GetParticipant response. See the IVS Real-Time Streaming API Reference.

October 12, 2023

Amazon IVS Broadcast SDK: Android 1.12.1 (Real-Time Streaming)

Platform	Downloads and Changes
Android Broadcast SDK 1.12.1	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.12.1/android • Fixed a bug where calling `BroadcastSession.setListener` resulted in an error.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.853 MB	16.375 MB

Architecture	Compressed Size	Uncompressed Size
armeabi-v7a	4.895 MB	10.803 MB
x86_64	6.149 MB	17.318 MB
x86	6.328 MB	17.186 MB

September 14, 2023

Amazon IVS Broadcast SDK: Web 1.5.2 (Real-Time Streaming)

Platform	Downloads and Changes
Web Broadcast SDK 1.5.2	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Fixed a bug that prevented republishing with `refreshStrategy` when the published state enters an `ERRORED` state.

August 23, 2023

Amazon IVS Broadcast SDK: Web 1.5.1, Android 1.12.0, and iOS 1.12.0 (Real-Time Streaming)

Platform	Downloads and Changes
Web Broadcast SDK 1.5.1	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Fixed a bug with internal Maybe types on TypeScript 5.

Platform	Downloads and Changes
	• Added better detection for Simulcast support. • Fixed two race conditions with `refreshStrategy` when trying to publish. • Fixed a race condition with `refreshStrategy` when trying to update participants to subscribe to.
All mobile (Android and iOS)	• Fixed a rare issue where publishing action is never completed. • Improved the stability of stages by reducing occurrences of rare crashes. • Improved the stability of stages by resolving race-condition issues caused by rapid join / leave. • Added a new `setOnFrameCallback` method on `ImageDevice` . This allows observation as frames pass through the device itself, giving insight into the aspect ratio of the latest images. This method also can be used to detect when the first frame is rendered for a remote participant in a stage.
Android Broadcast SDK 1.12.0	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.12.0/android • Android 9 is now supported. • Improved CPU usage and performance.

Platform	Downloads and Changes
iOS Broadcast SDK 1.12.0	**Download for real-time streaming:** https://broadcast.live-video.net/1.12.0/AmazonIVSBroadcast-Stages.xcframework.zip **Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.12.0/ios • Corrected the signature of `IVSDeviceDiscovery.createAudioSourceWithName` to return an `IVSCustomAudioSource` instead of `IVSCustomImageSource`.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.853 MB	16.375 MB
armeabi-v7a	4.895 MB	10.803 MB
x86_64	6.149 MB	17.318 MB
x86	6.328 MB	17.186 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	5.06 MB	10.92 MB

August 7, 2023

Amazon IVS Broadcast SDK: Web 1.5.0, Android 1.11.0, and iOS 1.11.0

Platform	Downloads and Changes
Web Broadcast SDK 1.5.0	**Reference documentation:** https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-reference • Added Simulcast – When enabled, this feature allows the publisher to send high- and low-quality layers of video. Subscribers automatically select their optimal quality based on their network conditions. See Optimizing Media.
All mobile (Android and iOS)	Added Simulcast – When enabled, this feature allows the publisher to send high- and low-quality layers of video. Subscribers automatically select their optimal quality based on their network conditions. See “Enable/Disable Layered Encoding with Simulcast” in the Android and iOS Broadcast SDK Guides.
Android Broadcast SDK 1.11.0	**Reference documentation:** https://aws.github.io/amazon-ivs-broadcast-docs/1.11.0/android • Fixed an issue where creating many stages eventually results in a crash. (The exact number of stages depends on the device.)
iOS Broadcast SDK 1.11.0	**Download for real-time streaming:** https://broadcast.live-video.net/1.11.0/AmazonIVSBroadcast-Stages.xcframework.zip

Platform	Downloads and Changes
	Reference documentation: https://aws.github.io/amazon-ivs-broadcast-docs/1.11.0/ios • Corrected the signature of `IVSDeviceDiscovery.createAudioSourceWithName` to return `IVSCustomAudioSource` instead of `IVSCustomImageSource`.

Broadcast SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
arm64-v8a	5.811 MB	16.186 MB
armeabi-v7a	4.857 MB	10.646 MB
x86_64	6.108 MB	17.122 MB
x86	6.289 MB	16.994 MB

Broadcast SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
arm64	5.030 MB	10.810 MB

August 7, 2023

Real-Time Streaming

Amazon Interactive Video Service (IVS) Real-Time Streaming enables you to deliver live streams with a latency that can be under 300 milliseconds from host to viewer.

Major documentation changes accompany this release. The IVS documentation landing page now has separate sections for real-time streaming and low-latency streaming. Each section has its own User Guide and API Reference. For documentation details, see the Document History (for both real-time and low-latency documentation changes). For real-time streaming, start with the IVS Real-Time Streaming User Guide and IVS Real-Time Streaming API Reference.