aws

Chat User Guide

Amazon IVS

Amazon IVS: Chat User Guide

Table of Contents

What is Amazon IVS Chat?

Amazon IVS Chat is a managed, live-chat feature to go alongside live video streams. Documentation is accessible from the Amazon IVS documentation landing page, in the Amazon IVS Chat section:

- Chat User Guide — This document, along with all the other User Guide pages listed on the navigation pane.
- Chat API Reference — Control-plane API (HTTPS).
- Chat Messaging API Reference — Data-plane API (WebSocket).
- SDK References for chat clients: Android, iOS, and JavaScript.

Getting Started with Amazon IVS Chat

Amazon Interactive Video Service (IVS) Chat is a managed, live-chat feature to go alongside your live video streams. (IVS Chat also can be used without a video stream.) You can create chat rooms and enable chat sessions between your users.

Amazon IVS Chat lets you focus on building customized chat experiences alongside live video. You don't need to manage infrastructure or develop and configure components of your chat workflows. Amazon IVS Chat is scalable, secure, reliable, and cost effective.

Amazon IVS Chat works best to facilitate messaging between participants of a live-video stream with a beginning and an end.

The rest of this document takes you through the steps to build your first chat application using Amazon IVS Chat.

Examples: The following demo apps are available (three sample clients apps and a backend server app for token creation):

- Amazon IVS Chat Web Demo
- Amazon IVS Chat for Android Demo
- Amazon IVS Chat for iOS Demo
- Amazon IVS Chat Demo Backend

Important: Chat rooms that have no new connections or updates for 24 months are automatically deleted.

Topics

Step 1: Do Initial Setup

Before proceeding, you must:

1. Create an AWS account.
2. Set up root and administrative users.
3. Set up AWS IAM (Identity and Access Management) permissions. Use the policy specified below.

For specific steps for all the above, see Getting Started with IVS Low-Latency Streaming in the *Amazon IVS User Guide*. **Important**: In "Step 3: Set Up IAM Permissions," use this policy for IVS Chat:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "ivschat:CreateChatToken",
            "ivschat:CreateLoggingConfiguration",
            "ivschat:CreateRoom",
            "ivschat:DeleteLoggingConfiguration",
            "ivschat:DeleteMessage",
            "ivschat:DeleteRoom",
            "ivschat:DisconnectUser",
            "ivschat:GetLoggingConfiguration",
            "ivschat:GetRoom",
            "ivschat:ListLoggingConfigurations",
            "ivschat:ListRooms",
            "ivschat:ListTagsForResource",
            "ivschat:SendEvent",
            "ivschat:TagResource",
            "ivschat:UntagResource",
            "ivschat:UpdateLoggingConfiguration",
            "ivschat:UpdateRoom"
         ],
         "Resource": "*"
      },
      {
         "Effect": "Allow",
         "Action": [
```

```
            "servicequotas:ListServiceQuotas",
            "servicequotas:ListServices",
            "servicequotas:ListAWSDefaultServiceQuotas",
            "servicequotas:ListRequestedServiceQuotaChangeHistoryByQuota",
            "servicequotas:ListTagsForResource",
            "cloudwatch:GetMetricData",
            "cloudwatch:DescribeAlarms"
         ],
         "Resource": "*"
      },
      {
         "Effect": "Allow",
         "Action": [
            "logs:CreateLogDelivery",
            "logs:GetLogDelivery",
            "logs:UpdateLogDelivery",
            "logs:DeleteLogDelivery",
            "logs:ListLogDeliveries",
            "logs:PutResourcePolicy",
            "logs:DescribeResourcePolicies",
            "logs:DescribeLogGroups",
            "s3:PutBucketPolicy",
            "s3:GetBucketPolicy",
            "iam:CreateServiceLinkedRole",
            "firehose:TagDeliveryStream"
         ],
         "Resource": "*"
      }
   ]
}
```

Step 2: Create a Chat Room

An Amazon IVS chat room has configuration information associated with it (e.g., maximum message length).

The instructions in this section show you how to use the console or AWS CLI to set up chat rooms (including optional setup for reviewing messages and/or logging messages) and create rooms.

Console Instructions for Creating an IVS Chat Room

These steps are divided into phases, starting with initial room setup and ending with final room creation.

Optionally, you can set up a room so messages are reviewed. For example, you can update message content or metadata, deny messages to prevent them from being sent, or let the original message through. This is covered in Set Up to Review Room Messages (Optional).

Also optionally, you can set up a room so that messages are logged. For example, if you have messages being sent to a chat room, you can log them to an Amazon S3 bucket, Amazon CloudWatch, or Amazon Kinesis Data Firehose. This is covered in Set Up to Log Messages (Optional).

Initial Room Setup

1. Open the Amazon IVS Chat console.

   (You also can access the Amazon IVS console through the AWS Management Console.)
2. From the navigation bar, use the **Select a Region** drop-down to choose a region. Your new room will be created in this region.
3. In the **Get started** box (top right), choose **Amazon IVS Chat Room**. The **Create room** window appears.

Create room Info

Rooms are the central Amazon IVS Chat resource. Clients can connect to a room to exchange messages with other clients who are connected to the room. Rooms that are inactive for 24 months will be automatically deleted. Learn more

How Amazon IVS Chat works

Setup

Room name – *optional*

name

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

Room configuration

- Default configuration
  Use the default maximum value of message limits
- Custom configuration
  Specify your own chat message limits

Message character limit Info

500 characters per message

Maximum message rate Info

10 messages per second

Message review handler Info

Review messages before they are sent to the room

- Disabled
  Messages will not be reviewed
- Handle with AWS Lambda
  Create or select an AWS Lambda function

Message logging Info

Automatically log chat messages

When enabled, messages from the chat room are logged automatically. Logged content can be managed directly in the destination services.

- Disabled
  Chat messages will not be logged
- Automatically log messages and events
  Create or select logging configurations

4. Under **Setup**, optionally specify a **Room name**. Room names are not unique, but they provide a way for you to distinguish rooms other than the room ARN (Amazon Resource Name).
5. Under **Setup > Room configuration**, either accept the **Default configuration**, or select **Custom configuration** and then configure the **Maximum message length** and/or **Maximum message rate**.
6. If you want to review messages, continue with Set Up to Review Room Messages (Optional) below. Otherwise, skip that (i.e., accept **Message Review Handler > Disabled**) and proceed directly to Final Room Creation.

Set Up to Review Room Messages (Optional)

1. Under **Message Review Handler**, select **Handle with AWS Lambda**. The **Message Review Handler** section expands to show additional options.
2. Configure the **Fallback result** to **Allow** or **Deny** the message if the handler does not return a valid response, encounters an error, or exceeds the timeout period.
3. Specify your existing **Lambda function** or use **Create Lambda function** to create a new function.

   The lambda function must be in the same AWS account and the same AWS regions as the chat room. You should give the Amazon Chat SDK service permission to invoke your lambda resource. The resource-based policy will be automatically created for the lambda function you selected. For more information about permissions, see Resource-Based Policy for Amazon IVS Chat.

Set Up to Log Messages (Optional)

1. Under **Message logging**, select **Automatically log chat messages**. The **Message logging** section expands to show additional options. You can either add an existing logging configuration to this room or create a new logging configuration by selecting **Create logging configuration**.
2. If you choose an existing logging configuration, a dropdown menu appears and shows all logging configurations that you already created. Select one from the list and your chat messages automatically will log to this destination.
3. If you choose **Create logging configuration**, a modal window appears which allows you to create and customize a new logging configuration.

a. Optionally specify a **Logging configuration name**. Logging-configuration names, like room names, are not unique, but they provide a way for you to distinguish logging configurations other than the logging configuration ARN.

b. Under **Destination**, select **CloudWatch log group**, **Kinesis firehose delivery stream**, or **Amazon S3 bucket** to choose the destination for your logs.

c. Depending on your Destination, select the option to create a new or use an existing **CloudWatch log group**, **Kinesis firehose delivery stream**, or **Amazon S3 bucket**.

d. After reviewing, choose **Create** to create a new logging configuration with a unique ARN. This automatically attaches the new logging configuration to the chat room.

Final Room Creation

1. After reviewing, choose **Create chat room** to create a new chat room with a unique ARN.

CLI Instructions for Creating an IVS Chat Room

This document takes you through the steps involved in creating an Amazon IVS chat room using the AWS CLI.

Creating a Chat Room

Creating a chat room 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. Run the chat `create-room` command and pass in an optional name:

```
aws ivschat create-room --name test-room
```

2. This returns a new chat room:

```
{
   "arn": "arn:aws:ivschat:us-west-2:123456789012:room/g1H2I3j4k5L6",
   "id": "string",
   "createTime": "2021-06-07T14:26:05-07:00",
   "maximumMessageLength": 200,
   "maximumMessageRatePerSecond": 10,
   "name": "test-room",
```

```
   "tags": {},
   "updateTime": "2021-06-07T14:26:05-07:00"
}
```

3. Note the `arn` field. You will need this to create a client token and connect to a chat room.

Setting Up a Logging Configuration (Optional)

As with creating a chat room, setting up a logging configuration 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. Run the chat `create-logging-configuration` command and pass in an optional name and a destination configuration pointing to an Amazon S3 bucket by name. This Amazon S3 bucket must exist before creating the logging configuration. (For details on creating an Amazon S3 bucket, see Amazon S3 Documentation.)

```
aws ivschat create-logging-configuration \
   --destination-configuration s3={bucketName=demo-logging-bucket} \
   --name "test-logging-config"
```

2. This returns a new logging configuration:

```
{
   "Arn": "arn:aws:ivschat:us-west-2:123456789012:logging-configuration/
ABcdef34ghIJ",
   "createTime": "2022-09-14T17:48:00.653000+00:00",
   "destinationConfiguration": {
      "s3": {"bucketName": "demo-logging-bucket"}
   },
   "id": "ABcdef34ghIJ",
   "name": "test-logging-config",
   "state": "ACTIVE",
   "tags": {},
   "updateTime": "2022-09-14T17:48:01.104000+00:00"
}
```

3. Note the `arn` field. You need this to attach the logging configuration to the chat room.
   a. If you are creating a new chat room, run the `create-room` command and pass the logging-configuration `arn`:

```
aws ivschat create-room --name test-room \
--logging-configuration-identifiers \
"arn:aws:ivschat:us-west-2:123456789012:logging-configuration/ABcdef34ghIJ"
```

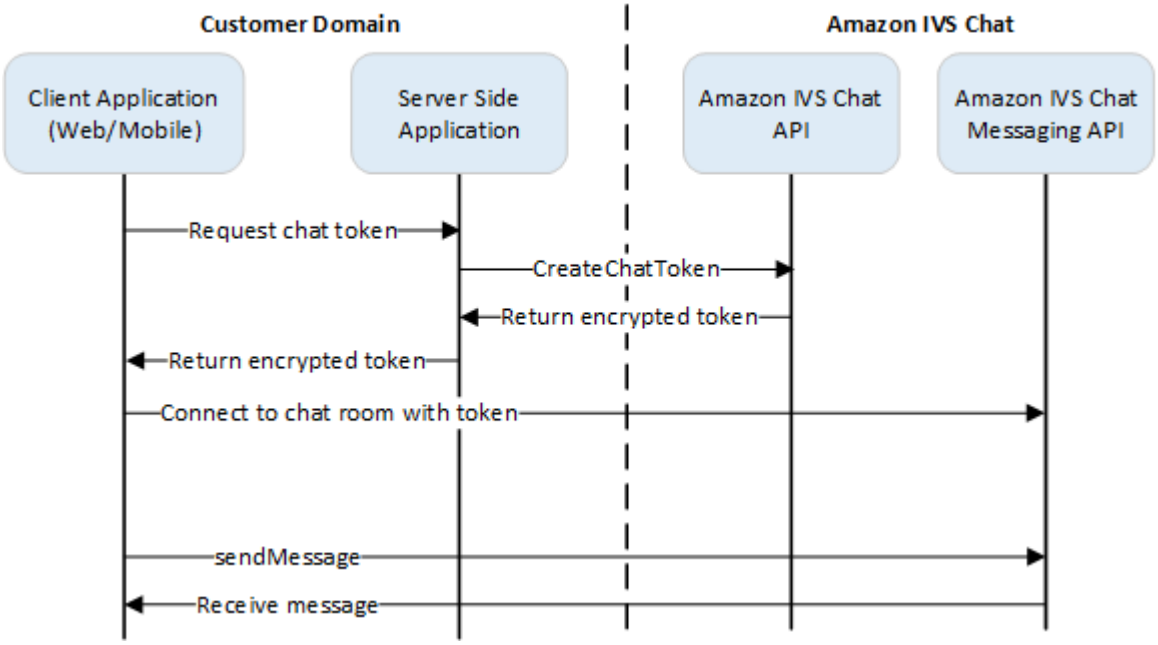
b. If you are updating an existing chat room, run the `update-room` command and pass the logging-configuration `arn`:

```
aws ivschat update-room --identifier \
"arn:aws:ivschat:us-west-2:12345689012:room/g1H2I3j4k5L6" \
--logging-configuration-identifiers \
"arn:aws:ivschat:us-west-2:123456789012:logging-configuration/ABcdef34ghIJ"
```

Step 3: Create a Chat Token

For a chat participant to connect to a room and start sending and receiving messages, a chat token must be created. Chat tokens are used to authenticate and authorize chat clients.

This diagram illustrates the workflow for creating an IVS chat token:



As shown above, a client application asks your server-side application for a token, and the server-side application calls CreateChatToken using an AWS SDK or SigV4 signed requests. 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.

A backend server application that demonstrates token generation is available at Amazon IVS Chat Demo Backend.

Session duration refers to how long an established session can remain active before it is automatically closed. That is, the session duration is how long the client can remain connected to the chat room before a new token must be generated and a new connection must be established. Optionally, you can specify session duration during token creation.

Each token can be used only once to establish a connection for an end user. If a connection is closed, a new token must be created before a connection can be re-established. The token itself is valid until the token-expiration timestamp included in the response.

When an end user wants to connect to a chat room, the client should ask the server application for a token. The server application creates a token and passes it back to the client. Tokens should be created for end users on demand.

To create a chat auth token, follow the instructions below. When you create a chat token, use the request fields to pass data about the chat end user and the end user's messaging capabilities; for details, see CreateChatToken in the *IVS Chat API Reference*.

AWS SDK Instructions

Creating a chat token with the AWS SDK requires that you first download and configure the SDK on your application. 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 load the AWS JavaScript SDK into your application. For details, see Getting started with the AWS SDK for JavaScript.

```
async function createChatToken(params) {
  const ivs = new AWS.Ivschat();
  const result = await ivs.createChatToken(params).promise();
  console.log("New token created", result.token);
}
/*
Create a token with provided inputs. Values for user ID and display name are
from your application and refer to the user connected to this chat session.
*/
const params = {
```

```
  "attributes": {
    "displayName": "DemoUser",
  }",
  "capabilities": ["SEND_MESSAGE"],
  "roomIdentifier": "arn:aws:ivschat:us-west-2:123456789012:room/g1H2I3j4k5L6",
  "userId": 11231234
};
createChatToken(params);
```

CLI Instructions

Creating a chat token 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. Note: 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 above).

1. Run the `create-chat-token` command along with room identifier and user ID for the client. Include any of the following capabilities: `"SEND_MESSAGE"`, `"DELETE_MESSAGE"`, `"DISCONNECT_USER"`. (Optionally, include session duration (in minutes) and/or custom attributes (metadata) about this chat session. These fields are not shown below.)

   ```
   aws ivschat create-chat-token --room-identifier "arn:aws:ivschat:us-
   west-2:123456789012:room/g1H2I3j4k5L6" --user-id "11231234" --capabilities
    "SEND_MESSAGE"
   ```

2. This returns a client token:

   ```
   {
     "token":
    "abcde12345FGHIJ67890_klmno1234PQRS567890uvwxyz1234.abcd12345EFGHI67890_jklmno123PQRS567890u
     "sessionExpirationTime": "2022-03-16T04:44:09+00:00",
     "tokenExpirationTime": "2022-03-16T03:45:09+00:00"
   }
   ```

3. Save this token. You will need this to connect to the chat room and send or receive messages. You will need to generate another chat token before your session ends (as indicated by `sessionExpirationTime`).

Step 4: Send and Receive Your First Message

Use your chat token to connect to a chat room and send your first message. Sample JavaScript code is provided below. IVS client SDKs also are available: see Chat SDK: Android Guide, Chat SDK: iOS Guide, and Chat SDK: JavaScript Guide.

Regional service: The sample code below refers to your "supported region of choice." Amazon IVS Chat offers regional endpoints that you can use to make your requests. For the Amazon IVS Chat Messaging API, the general syntax of a regional endpoint is:

wss://edge.ivschat.<region-code>.amazonaws.com

For example, wss://edge.ivschat.us-west-2.amazonaws.com is the endpoint in the US West (Oregon) region. For a list of supported regions, see the Amazon IVS Chat information on the Amazon IVS page in the *AWS General Reference*.

```
/*
1. To connect to a chat room, you need to create a Secure-WebSocket connection
using the client token you created in the previous steps. Use one of the provided
endpoints in the Chat Messaging API, depending on your AWS region.
*/
const chatClientToken = "GENERATED_CHAT_CLIENT_TOKEN_HERE";
const socket = "wss://edge.ivschat.us-west-2.amazonaws.com"; // Replace “us-west-2”
 with supported region of choice.
const connection = new WebSocket(socket, chatClientToken);

/*
2. You can send your first message by listening to user input
in the UI and sending messages to the WebSocket connection.
*/
const payload = {
  "Action": "SEND_MESSAGE",
  "RequestId": "OPTIONAL_ID_YOU_CAN_SPECIFY_TO_TRACK_THE_REQUEST",
  "Content": "text message",
  "Attributes": {
    "CustomMetadata": "test metadata"
  }
}
connection.send(JSON.stringify(payload));

/*
```

```
3. To listen to incoming chat messages from this WebSocket connection
and display them in your UI, you must add some event listeners.
*/
connection.onmessage = (event) => {
  const data = JSON.parse(event.data);
  displayMessages({
    display_name: data.Sender.Attributes.DisplayName,
    message: data.Content,
    timestamp: data.SendTime
  });
}

function displayMessages(message) {
  // Modify this function to display messages in your chat UI however you like.
  console.log(message);
}

/*
4. Delete a chat message by sending the DELETE_MESSAGE action to the WebSocket
connection. The connected user must have the "DELETE_MESSAGE" permission to
perform this action.
*/

function deleteMessage(messageId) {
  const deletePayload = {
    "Action": "DELETE_MESSAGE",
    "Reason": "Deleted by moderator",
    "Id": "${messageId}"
  }
  connection.send(deletePayload);
}
```

Congratulations, you are all set! You now have a simple chat application that can send or receive messages.

Step 5: Check Your Service-Quota Limits (Optional)

Your chat rooms will scale along with your Amazon IVS live stream, to enable all your viewers to engage in chat conversations. However, all Amazon IVS accounts have limits on the number of concurrent chat participants and the rate of message delivery.

Ensure that your limits are adequate and request an increase if needed, especially if you are planning a large streaming event. For details, see Service Quotas (Low-Latency Streaming), Service Quotas (Real-Time Streaming), and Service Quotas (Chat).

IVS Chat Logging

The Chat Logging feature allows you to record all messages in a room to any of three standard locations: an Amazon S3 bucket, Amazon CloudWatch Logs, or Amazon Kinesis Data Firehose. Subsequently, the logs can be used for analysis or building a chat replay that links to a live-video session.

Enable Chat Logging for a Room

Chat Logging is an advanced option that can be enabled by associating a logging configuration with a room. A logging configuration is a resource that allows you to specify a type of location (Amazon S3 bucket, Amazon CloudWatch Logs, or Amazon Kinesis Data Firehose) where messages of a room are logged. For details on creating and managing logging configurations, see Getting Started with Amazon IVS Chat and Amazon IVS Chat API Reference.

You can associate up to three logging configurations with each room, either when creating a new room (CreateRoom) or updating an existing room (UpdateRoom). You can associate multiple rooms with the same logging configuration.

When at least one active logging configuration is associated with a room, every messaging request sent to that room via the Amazon IVS Chat Messaging API is automatically recorded to the specified location(s). These are the average propagation delays (from when a messaging request is sent to when it becomes available in your specified locations):

- Amazon S3 bucket: 5 minutes
- Amazon CloudWatch Logs or Amazon Kinesis Data Firehose: 10 seconds

Message Content

Format

```
{
   "event_timestamp": "string",
   "type": "string",
   "version": "string",
   "payload": { "string": "string" }
```

```
}
```

Fields

Field	Description
`event_timestamp`	UTC timestamp of when the message was received by Amazon IVS Chat.
`payload`	The Message (Subscribe) or Event (Subscribe) JSON payload that clients will receive from the Amazon IVS Chat service.
`type`	Type of the chat message. • Valid Values: `MESSAGE` | `EVENT`
`version`	Version of the message-content format.

Amazon S3 Bucket

Format

Message logs are organized and stored with the following S3 prefix and file format:

```
AWSLogs/<account_id>/IVSChatLogs/<version>/<region>/room_<resource_id>/<year>/<month>/
<day>/<hours>/
<account_id>_IVSChatLogs_<version>_<region>_room_<resource_id>_<year><month><day><hours><minute
```

Fields

Field	Description
`<account_id>`	AWS account ID from which the room is created.
`<hash>`	A hash value generated by the system to ensure uniqueness.

Field	Description
<region>	The AWS service region where the room was created.
<resource_id>	The resource ID part of the room ARN.
<version>	Version of the message-content format.
<year> / <month> / <day> / <hours> / <minute>	UTC timestamp of when the message was received by Amazon IVS Chat.

Example

```
AWSLogs/123456789012/IVSChatLogs/1.0/us-west-2/
room_abc123DEF456/2022/10/14/17/123456789012_IVSChatLogs_1.0_us-
west-2_room_abc123DEF456_20221014T1740Z_1766dcbc.log.gz
```

Amazon CloudWatch Logs

Format

Message logs are organized and stored with the following log-stream name format:

```
aws/IVSChatLogs/<version>/room_<resource_id>
```

Fields

Field	Description
<resource_id>	Resource ID part of the room ARN.
<version>	Version of the message-content format.

Example

```
aws/IVSChatLogs/1.0/room_abc123DEF456
```

Amazon Kinesis Data Firehose

Message logs are sent to the delivery stream as real-time streaming data to destinations like Amazon Redshift, Amazon OpenSearch Service, Splunk, and any custom HTTP endpoint or HTTP endpoints owned by supported third-party service providers. For more information, see What Is Amazon Kinesis Data Firehose.

Constraints

- You must own the logging location where messages will be stored.
- The room, logging configuration, and logging location must be in the same AWS region.
- Only active logging configurations are available for Chat Logging.
- You can only delete a logging configuration that is no longer associated with any rooms.

Logging messages to a location that you own requires authorization with your AWS credentials. To give IVS Chat the required access, a resource policy (for an Amazon S3 bucket or CloudWatch Logs) or an AWS IAM Service-Linked Role (SLR) (for Amazon Kinesis Data Firehose) is generated automatically when the logging configuration is created. *Be cautious about any modification to the role or policies, as that can impact permission for chat logging.*

Monitoring Errors with Amazon CloudWatch

You can monitor errors occurring in chat logging with Amazon CloudWatch, and you can create alarms or dashboards to indicate or respond to the changes of specific errors.

There are several types of errors. For more information, see Monitoring Amazon IVS Chat.

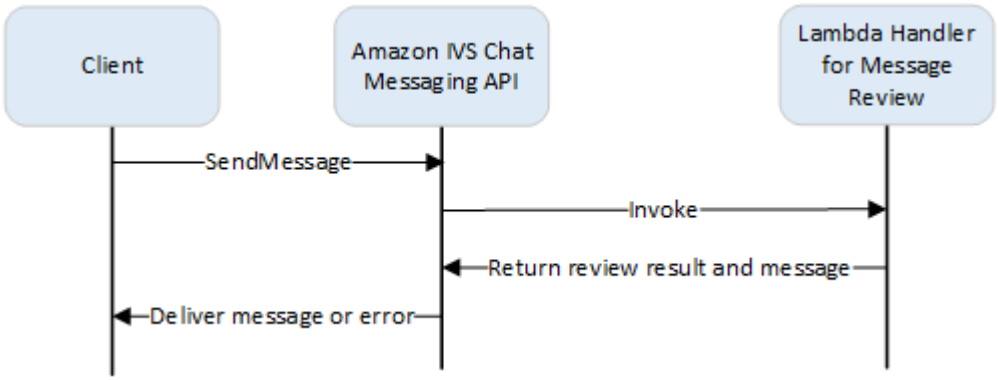
IVS Chat Message Review Handler

A message review handler allows you to review and/or modify messages before they are delivered to a room. When a message review handler is associated with a room, it is invoked for each SendMessage request to that room. The handler enforces your application's business logic and determines whether to allow, deny, or modify a message. Amazon IVS Chat supports AWS Lambda functions as handlers.

Creating a Lambda Function

Before setting up a message review handler for a room, you must create a lambda function with a resource-based IAM policy. The lambda function must be in the same AWS account and AWS region as the room with which you will use the function. The resource-based policy gives Amazon IVS Chat permission to invoke your lambda function. For instructions, see Resource-Based Policy for Amazon IVS Chat.

Workflow



Request Syntax

When a client sends a message, Amazon IVS Chat invokes the lambda function with a JSON payload:

```
{
   "Content": "string",
   "MessageId": "string",
   "RoomArn": "string",
   "Attributes": {"string": "string"},
   "Sender": {
      "Attributes": { "string": "string" },
```

```
      "UserId": "string",
      "Ip": "string"
   }
}
```

Request Body

Field	Description
`Attributes`	Attributes associated with the message.
`Content`	Original content of the message.
`MessageId`	The message ID. Generated by IVS Chat.
`RoomArn`	The ARN of the room where messages are sent.
`Sender`	Information about the sender. This object has several fields: • `Attributes` — Metadata about the sender established during authentic ation. This can be used to give the client more information about the sender; e.g., avatar URL, badges, font, and color. • `UserId` — An application-specified identifier of the viewer (end user) who sent this message. This can be used by the client application to refer to the user in either the messaging API or application domains. • `Ip` — The IP address of the client sending the message.

Response Syntax

The handler lambda function must return a JSON response with the following syntax. Responses that do not correspond to the syntax below or satisfy the field constraints are invalid. In this case, the message is allowed or denied depending on the `FallbackResult` value that you specify in your message review handler; see MessageReviewHandler in the *Amazon IVS Chat API Reference*.

```
{
   "Content": "string",
   "ReviewResult": "string",
   "Attributes": {"string": "string"},
```

```
}
```

Response Fields

Field	Description
`Attributes`	Attributes associated with the message returned from the lambda function. If `ReviewResult` is DENY, a `Reason` may be provided in `Attributes` ; e.g.: `"Attributes": {"Reason": "denied for moderation` In this case, the sender client receives a WebSocket 406 error with the reason in the error message. (See WebSocket Errors in the *Amazon IVS Chat Messaging API Reference*.) • Size Constraints: Maximum 1 KB • Required: No
`Content`	Content of the message returned from the Lambda function. It could be edited or original depending on the business logic. • Length Constraints: Minimum length of 1. Maximum length of the `MaximumMessageLength` you defined when you created/updated the room. See the Amazon IVS Chat API Reference for more information. This applies only when `ReviewResult` is `ALLOW`. • Required: Yes
`ReviewResult`	The result of review processing on how to handle the message. If allowed, the message is delivered to all users connected to the room. If denied, the message is not delivered to any user. • Valid Values: `ALLOW` | DENY • Required: Yes

Sample Code

Below is a sample lambda handler in Go. It modifies the message content, keeps the message attributes unchanged, and allows the message.

```
package main

import (
   "context"
   "github.com/aws/aws-lambda-go/lambda"
)

type Request struct {
   MessageId string
   Content string
   Attributes map[string]string
   RoomArn string
   Sender Sender
}

type Response struct {
   ReviewResult string
   Content string
   Attributes map[string]string
}

type Sender struct {
   UserId string
   Ip string
   Attributes map[string]string
}

func main() {
   lambda.Start(HandleRequest)
}

func HandleRequest(ctx context.Context, request Request) (Response, error) {
   content := request.Content + "modified by the lambda handler"
   return Response{
       ReviewResult: "ALLOW",
       Content: content,
   }, nil
}
```

Associating and Dissociating a Handler with a Room

Once you have the lambda handler set up and implemented, use the Amazon IVS Chat API:

- To associate the handler with a room, call CreateRoom or UpdateRoom and specify the handler.
- To disassociate the handler from a room, call UpdateRoom with an empty value for `MessageReviewHandler.Uri`.

Monitoring Errors with Amazon CloudWatch

You can monitor errors occurring in message review with Amazon CloudWatch, and you can create alarms or dashboards to indicate or respond to the changes of specific errors. If an error occurs, the message is allowed or denied depending on the `FallbackResult` value you specify when you associate the handler with a room; see MessageReviewHandler in the *Amazon IVS Chat API Reference*.

There are several types of errors:

- `InvocationErrors` occur when Amazon IVS Chat cannot invoke a handler.
- `ResponseValidationErrors` occur when a handler returns a response that is invalid.
- AWS Lambda `Errors` occur when a lambda handler returns a function error when it has been invoked.

For more information on invocation errors and response-validation errors (emitted by Amazon IVS Chat), see Monitoring Amazon IVS Chat. For more information on AWS Lambda errors, see Working with Lambda Metrics.

Monitoring Amazon IVS Chat

You can monitor Amazon Interactive Video Service (IVS) Chat resources using Amazon CloudWatch. CloudWatch collects and processes raw data from Amazon IVS Chat into readable, near real-time metrics. These statistics are kept for 15 months, so you can gain a historical perspective on how your web application or service performs. You can set alarms for certain thresholds and send notifications or take actions when those thresholds are met. For details, see the CloudWatch User Guide.

Access CloudWatch Metrics

Amazon CloudWatch collects and processes raw data from Amazon IVS Chat into readable, near-real-time metrics. These statistics are kept for 15 months, so you can gain a historical perspective on how your web application or service performs. You can set alarms for certain thresholds and send notifications or take actions when those thresholds are met. For details, see the CloudWatch User Guide.

Note that CloudWatch metrics are rolled up over time. Resolution effectively decreases as the metrics age. Here is the schedule:

- 60-second metrics are available for 15 days.
- 5-minute metrics are available for 63 days.
- 1-hour metrics are available for 455 days (15 months).

For current information on data retention, search for "retention period" in Amazon CloudWatch FAQs.

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 **IVSChat** namespace.

5. Do one of the following:

   a. In the search bar, enter your resource ID (part of the ARN, `arn:::ivschat:room/<resource id>`).

      Then select **IVSChat**.

   b. If **IVSChat** appears as a selectable service under **AWS Namespaces**, select it. It will be listed if you use Amazon IVSChat and it is sending metrics to Amazon CloudWatch. (If **IVSChat** is not listed, you do not have any Amazon IVSChat 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 chat session's CloudWatch chart from the chat 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 low-latency chat metrics using the AWS CLI:

- At a command prompt, run:

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

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

CloudWatch Metrics: IVS Chat

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

Metric	Dimensions	Description
`ConcurrentChatConnections`	None	The total number of concurrent connections in a chat room (reported max per minute). This is useful to understand when customers approach their limit for concurrent chat connections in a region. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum
`Deliveries`	Action	The number of deliveries of messaging requests made of a specific action type to chat connections across all your rooms in a region. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum
`InvocationErrors`	`Uri`	The number of invocation errors of a specific message review handler across all your rooms in a region. An invocation error occurs when the message review handler cannot be invoked. Invocation errors occur when Amazon IVS Chat cannot invoke a handler. This may happen if the handler associated with a room no longer exists or times out, or if its resource policy does not allow the service to invoke it. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum

Metric	Dimensions	Description
`LogDestinationAccessDeniedError`	`LoggingConfiguration`	The number of access-denied errors of a log destination across all your rooms in a region. These errors occur when Amazon IVS Chat cannot access the destination resource you specified in the logging configuration. This may happen if the destination resource policy does not allow the service to put records. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum
`LogDestinationErrors`	`LoggingConfiguration`	The number of all errors of a log destination across all your rooms in a region. This is an aggregated metric which includes all types of errors that occur when Amazon IVS Chat fails to deliver logs to the destination resource you specified in the logging configuration. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum

Metric	Dimensions	Description
`LogDestinationResourceNotFoundErrors`	`LoggingConfiguration`	The number of resource-not-found errors of a log destination across all your rooms in a region. These errors occur when Amazon IVS Chat cannot deliver logs to a destination resource you specified in a logging configuration because the resource does not exist. This may happen if the destination resource associated with a logging configuration no longer exists. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum
`MessagingDeliveries`	None	The number of deliveries of messaging requests to chat connections across all your rooms in a region. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum
`MessagingRequests`	None	The number of messaging requests made across all your rooms in a region. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum

Metric	Dimensions	Description
Requests	Action	The number of requests made of a specific action type across all your rooms in a region. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum
ResponseValidationErrors	Uri	The number of response-validation errors of a specific message review handler across all your rooms in a region. A response-validation error occurs when the response from the message review handler is invalid. This may mean that the response could not be parsed or fails validation checks; e.g., an invalid review result or response values that are too long. Unit: Count Valid statistics: Sum, Average, Maximum, Minimum

IVS Chat Client Messaging SDK

The Amazon Interactive Video Services (IVS) Chat Client Messaging SDK is for developers who are building applications with Amazon IVS. This SDK is designed to leverage the Amazon IVS architecture and will see updates, alongside Amazon IVS Chat. As a native SDK, it is designed to minimize the performance impact on your application and on the devices with which your users access your application.

Platform Requirements

Desktop Browsers

Browser	Supported Versions
Chrome	Two major versions (current and most recent prior version)
Edge	Two major versions (current and most recent prior version)
Firefox	Two major versions (current and most recent prior version)
Opera	Two major versions (current and most recent prior version)
Safari	Two major versions (current and most recent prior version)

Mobile Browsers

Browser	Supported Versions
Chrome for Android	Two major versions (current and most recent prior version)
Firefox for Android	Two major versions (current and most recent prior version)
Opera for Android	Two major versions (current and most recent prior version)

Browser	Supported Versions
WebView Android	Two major versions (current and most recent prior version)
Samsung Internet	Two major versions (current and most recent prior version)
Safari for iOS	Two major versions (current and most recent prior version)

Native Platforms

Platform	Supported Versions
Android	5.0 and later
iOS	13.0 and later

Support

If you encounter an error or other issue with your chat room, determine the unique room identifier via the IVS Chat API (see ListRooms).

Share this chat room identifier with AWS support. With it, they can get information to help troubleshoot your issue.

Note: See Amazon IVS Chat Release Notes for available versions and fixed issues. If appropriate, before contacting support, update your version of the SDK and see if that resolves your issue.

Versioning

The Amazon IVS Chat Client Messaging 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.

Also, version 1.x will be supported until the end of 2023 or when 3.x is released, whichever happens later.

Amazon IVS Chat APIs

On the server side (not managed by the SDKs), there are two APIs, each with its own responsibilities:

- **Data plane** — The IVS Chat Messaging API is a WebSockets API designed to be used by front-end applications (iOS, Android, macOS, etc) that are driven by a token-based authentication scheme. Using a previously generated chat token, you connect to already existing chat rooms using this API.

  The Amazon IVS Chat Client Messaging SDKs are concerned only with the data plane. The SDKs assume that you are already generating chat tokens through your backend. Retrieval of these tokens is assumed to be managed by your front-end application, not the SDKs.

- **Control plane** — The IVS Chat Control Plane API provides an interface for your own *backend applications* to manage and create chat rooms as well as the users who join them. Think of this as the admin panel for your app's chat experience that is managed by *your own backend*. There are control-plane operations that are responsible for creating the *chat token* that the data plane needs to authenticate to a chat room.

  Important: *The IVS Chat Client Messaging SDKs do not call any control-plane operations. You must have your backend set up to create chat tokens for you. Your front-end application must communicate with your backend to retrieve this chat token.*

IVS Chat Client Messaging SDK: Android Guide

The Amazon Interactive Video (IVS) Chat Client Messaging Android SDK provides interfaces that allow you to easily incorporate our IVS Chat Messaging API on platforms using Android.

The `com.amazonaws:ivs-chat-messaging` package implements the interface described in this document.

Latest version of IVS Chat Client Messaging Android SDK: 1.1.0 (Release Notes)

Reference documentation: For information on the most important methods available in the Amazon IVS Chat Client Messaging Android SDK, see the reference documentation at: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/

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

Platform requirements: Android 5.0 (API level 21) or greater is required for development.

Getting Started with the IVS Chat Client Messaging Android SDK

Before starting, you should be familiar with Getting Started with Amazon IVS Chat.

Add the Package

Add `com.amazonaws:ivs-chat-messaging` to your `build.gradle` dependencies:

```
dependencies {
   implementation 'com.amazonaws:ivs-chat-messaging'
}
```

Add Proguard Rules

Add the following entries to your R8/Proguard rules file (`proguard-rules.pro`):

```
-keep public class com.amazonaws.ivs.chat.messaging.** { *; }
-keep public interface com.amazonaws.ivs.chat.messaging.** { *; }
```

Set Up Your Backend

This integration requires endpoints on your server that talk to the Amazon IVS API. Use the official AWS libraries for access to the Amazon IVS API from your server. These are accessible within several languages from the public packages; e.g., node.js and Java.

Next, create a server endpoint that talks to the Amazon IVS Chat API and creates a token.

Set Up a Server Connection

Create a method that takes `ChatTokenCallback` as a param and fetches a chat token from your backend. Pass that token to the `onSuccess` method of the callback. In case of error, pass the exception to the `onError` method of the callback. This is needed to instantiate the main `ChatRoom` entity in the next step.

Below you can find sample code that implements the above using a `Retrofit` call.

```
// ...

private fun fetchChatToken(callback: ChatTokenCallback) {
    apiService.createChatToken(userId, roomId).enqueue(object : Callback<ChatToken> {
        override fun onResponse(call: Call<ExampleResponse>, response:
 Response<ExampleResponse>) {
            val body = response.body()
            val token = ChatToken(
                body.token,
                body.sessionExpirationTime,
                body.tokenExpirationTime
            )
            callback.onSuccess(token)
        }

        override fun onFailure(call: Call<ChatToken>, throwable: Throwable) {
            callback.onError(throwable)
        }
```

```
    })
}
// ...
```

Using the IVS Chat Client Messaging Android SDK

This document takes you through the steps involved in using the Amazon IVS chat client messaging Android SDK.

Initialize a Chat Room Instance

Create an instance of the `ChatRoom` class. This requires passing `regionOrUrl`, which typically is the AWS region in which your chat room is hosted, and `tokenProvider` which is the token-fetching method created in the previous step.

```
val room = ChatRoom(
    regionOrUrl = "us-west-2",
    tokenProvider = ::fetchChatToken
)
```

Next, create a listener object that will implement handlers for chat related events, and assign it to the `room.listener` property:

```
private val roomListener = object : ChatRoomListener {
    override fun onConnecting(room: ChatRoom) {
      // Called when room is establishing the initial connection or reestablishing
 connection after socket failure/token expiration/etc
    }

    override fun onConnected(room: ChatRoom) {
        // Called when connection has been established
    }

    override fun onDisconnected(room: ChatRoom, reason: DisconnectReason) {
        // Called when a room has been disconnected
    }

    override fun onMessageReceived(room: ChatRoom, message: ChatMessage) {
        // Called when chat message has been received
    }
```

```
    override fun onEventReceived(room: ChatRoom, event: ChatEvent) {
        // Called when chat event has been received
    }

    override fun onDeleteMessage(room: ChatRoom, event: DeleteMessageEvent) {
       // Called when DELETE_MESSAGE event has been received
    }
}

val room = ChatRoom(
    region = "us-west-2",
    tokenProvider = ::fetchChatToken
)

room.listener = roomListener // <- add this line

// ...
```

The last step of basic initialization is connecting to the specific room by establishing a WebSocket connection. To do this, call the `connect()` method within the room instance. We recommend doing so in the `onResume()` lifecycle method to make sure it keeps a connection if your app resumes from the background.

```
room.connect()
```

The SDK will try to establish a connection to a chat room encoded in the chat token received from your server. If it fails, it will try to reconnect the number of times specified in the room instance.

Perform Actions in a Chat Room

The `ChatRoom` class has actions for sending and deleting messages and disconnecting other users. These actions accept an optional callback parameter that allows you to get request confirmation or rejection notifications.

Sending a Message

For this request, you must have the `SEND_MESSAGE` capability encoded in your chat token.

To trigger a send-message request:

```
val request = SendMessageRequest("Test Echo")
```

```
room.sendMessage(request)
```

To get a confirmation/rejection of the request, provide a callback as a second parameter:

```
room.sendMessage(request, object : SendMessageCallback {
   override fun onConfirmed(request: SendMessageRequest, response: ChatMessage) {
      // Message was successfully sent to the chat room.
   }
   override fun onRejected(request: SendMessageRequest, error: ChatError) {
      // Send-message request was rejected. Inspect the `error` parameter for details.
   }
})
```

Deleting a Message

For this request, you must have the DELETE_MESSAGE capability encoded in your chat token.

To trigger a delete-message request:

```
val request = DeleteMessageRequest(messageId, "Some delete reason")
room.deleteMessage(request)
```

To get a confirmation/rejection of the request, provide a callback as a second parameter:

```
room.deleteMessage(request, object : DeleteMessageCallback {
   override fun onConfirmed(request: DeleteMessageRequest, response:
 DeleteMessageEvent) {
      // Message was successfully deleted from the chat room.
   }
   override fun onRejected(request: DeleteMessageRequest, error: ChatError) {
      // Delete-message request was rejected. Inspect the `error` parameter for
 details.
   }
})
```

Disconnecting Another User

For this request, you must have the DISCONNECT_USER capability encoded in your chat token.

To disconnect another user for moderation purposes:

```
val request = DisconnectUserRequest(userId, "Reason for disconnecting user")
```

```
room.disconnectUser(request)
```

To get confirmation/rejection of the request, provide a callback as a second parameter:

```
room.disconnectUser(request, object : DisconnectUserCallback {
   override fun onConfirmed(request: SendMessageRequest, response: ChatMessage) {
      // User was disconnected from the chat room.
   }
   override fun onRejected(request: SendMessageRequest, error: ChatError) {
      // Disconnect-user request was rejected. Inspect the `error` parameter for
 details.
   }
})
```

Disconnect from a Chat Room

To close your connection to the chat room, call the `disconnect()` method on the room instance:

```
room.disconnect()
```

Because the WebSocket connection will stop working after a short time when the application is in a background state, we recommend that you manually connect/disconnect when transitioning from/to a background state. To do so, match the `room.connect()` call in the `onResume()` lifecycle method, on Android `Activity` or `Fragment`, with a `room.disconnect()` call in the `onPause()` lifecycle method.

IVS Chat Client Messaging SDK: Android Tutorial Part 1: Chat Rooms

This is the first of a two-part tutorial. You will learn the essentials of working with the Amazon IVS Chat Messaging SDK by building a fully functional Android app using the Kotlin programming language. We call the app *Chatterbox*.

Before you start the module, take a few minutes to familiarize yourself with the prerequisites, key concepts behind chat tokens, and the backend server necessary for creating chat rooms.

These tutorials are created for experienced Android developers who are new to the IVS Chat Messaging SDK. You will need to be comfortable with the Kotlin programming language and creating UIs on the Android platform.

This first part of the tutorial is broken up into several sections:

1. the section called “Set Up a Local Authentication/Authorization Server”
2. the section called “Create a Chatterbox Project”
3. the section called “Connect to a Chat Room and Observe Connection Updates”
4. the section called “Build a Token Provider”
5. the section called “Next Steps”

For full SDK documentation, start with Amazon IVS Chat Client Messaging SDK (here in the *Amazon IVS Chat User Guide*) and the Chat Client Messaging: SDK for Android Reference (on GitHub).

Prerequisites

- Be familiar with Kotlin and creating applications on the Android platform. If you are unfamiliar with creating applications for Android, learn the basics in the Build your first app guide for Android developers.
- Thoroughly read and understand *Getting Started with IVS Chat*.
- Create an AWS IAM user with the `CreateChatToken` and `CreateRoom` capabilities defined in an existing IAM policy. (See *Getting Started with IVS Chat*.)
- Ensure that the secret/access keys for this user are stored in an AWS credentials file. For instructions, see the AWS CLI User Guide (especially Configuration and credential file settings).
- Create a chat room and save its ARN. See *Getting Started with IVS Chat*. (If you don’t save the ARN, you can look it up later with the console or Chat API.)

Set Up a Local Authentication/Authorization Server

Your backend server will be responsible for both creating chat rooms and generating the chat tokens needed for the IVS Chat Android SDK to authenticate and authorize your clients for your chat rooms.

See Create a Chat Token in *Getting Started with Amazon IVS Chat*. As shown in the flowchart there, your server-side code is responsible for creating a chat token. This means your app must provide its own means of generating a chat token by requesting one from your server-side application.

We use the Ktor framework to create a live local server that manages the creation of chat tokens using your local AWS environment.

At this point, we expect you have your AWS credentials set up correctly. For step-by-step instructions, see Set up AWS Credentials and Region for Development.

Create a new directory and call it `chatterbox` and inside it, another one, called `auth-server`.

Our server folder will have the following structure:

```
- auth-server
  - src
    - main
      - kotlin
        - com
          - chatterbox
            - authserver
              - Application.kt
       - resources
         - application.conf
         - logback.xml
   - build.gradle.kts
```

Note: you can directly copy/paste the code here into the referenced files.

Next, we add all the necessary dependencies and plugins for our auth server to work:

Kotlin Script:

```
// ./auth-server/build.gradle.kts

plugins {
   application
   kotlin("jvm")
   kotlin("plugin.serialization").version("1.7.10")
}

application {
   mainClass.set("io.ktor.server.netty.EngineMain")
}

dependencies {
   implementation("software.amazon.awssdk:ivschat:2.18.1")
   implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.7.20")
```

```
    implementation("io.ktor:ktor-server-core:2.1.3")
    implementation("io.ktor:ktor-server-netty:2.1.3")
    implementation("io.ktor:ktor-server-content-negotiation:2.1.3")
    implementation("io.ktor:ktor-serialization-kotlinx-json:2.1.3")

    implementation("ch.qos.logback:logback-classic:1.4.4")
}
```

Now we need to set up logging functionality for the auth server. (For more information, see Configure logger.)

XML:

```
// ./auth-server/src/main/resources/logback.xml

<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{YYYY-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</
pattern>
        </encoder>
    </appender>
    <root level="trace">
        <appender-ref ref="STDOUT"/>
    </root>
    <logger name="org.eclipse.jetty" level="INFO"/>
    <logger name="io.netty" level="INFO"/>
</configuration>
```

The Ktor server requires configuration settings, which it automatically loads from the `application.*` file in the `resources` directory, so we add that as well. (For more information, see Configuration in a file.)

HOCON:

```
// ./auth-server/src/main/resources/application.conf

ktor {
    deployment {
        port = 3000
    }
    application {
```

```
      modules = [ com.chatterbox.authserver.ApplicationKt.main ]
   }
}
```

Finally, let's implement our server:

Kotlin:

```
// ./auth-server/src/main/kotlin/com/chatterbox/authserver/Application.kt

package com.chatterbox.authserver

import io.ktor.http.*
import io.ktor.serialization.kotlinx.json.*
import io.ktor.server.application.*
import io.ktor.server.plugins.contentnegotiation.*
import io.ktor.server.request.*
import io.ktor.server.response.*
import io.ktor.server.routing.*
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import software.amazon.awssdk.services.ivschat.IvschatClient
import software.amazon.awssdk.services.ivschat.model.CreateChatTokenRequest

@Serializable
data class ChatTokenParams(var userId: String, var roomIdentifier: String)

@Serializable
data class ChatToken(
   val token: String,
   val sessionExpirationTime: String,
   val tokenExpirationTime: String,
)

fun Application.main() {
   install(ContentNegotiation) {
      json(Json)
   }

   routing {
      post("/create_chat_token") {
         val callParameters = call.receive<ChatTokenParams>()
         val request =
 CreateChatTokenRequest.builder().roomIdentifier(callParameters.roomIdentifier)
```

```
            .userId(callParameters.userId).build()
         val token = IvschatClient.create()
            .createChatToken(request)

         call.respond(
            ChatToken(
                token.token(),
                token.sessionExpirationTime().toString(),
                token.tokenExpirationTime().toString()
            )
         )
      }
   }
}
```

Create a Chatterbox Project

To create an Android project, install and open Android Studio.

Follow the steps listed in the official Android Create a Project guide.

- In Choose your project type, choose the **Empty Activity** project template for our Chatterbox app.
- In Configure your project, choose the following values for configuration fields:
  - **Name**: My App
  - **Package name**: com.chatterbox.myapp
  - **Save location**: point at the created `chatterbox` directory created in the previous step
  - **Language**: Kotlin
  - **Minimum API level**: API 21: Android 5.0 (Lollipop)

After specifying all the configuration parameters correctly, our file structure inside the `chatterbox` folder should look like the following:

```
- app
  - build.gradle
  ...
- gradle
- .gitignore
- build.gradle
- gradle.properties
```

```
- gradlew
- gradlew.bat
- local.properties
- settings.gradle
- auth-server
  - src
    - main
      - kotlin
        - com
          - chatterbox
            - authserver
              - Application.kt
       - resources
         - application.conf
         - logback.xml
   - build.gradle.kts
```

Now that we have a working Android project, we can add com.amazonaws:ivs-chat-messaging to our `build.gradle` dependencies. (For more information on the Gradle build toolkit, see Configure your build.)

Note: At the top of every code snippet, there is a path to the file where you should be making changes in your project. The path is relative to the project's root.

In the code below, replace `<version>` with the current version number of the Chat Android SDK (e.g., 1.0.0).

Kotlin:

```
// ./app/build.gradle

plugins {
// ...
}

android {
// ...
}

dependencies {
   implementation("com.amazonaws:ivs-chat-messaging:<version>")
// ...
```

```
}
```

After the new dependency is added, run **Sync Project with Gradle Files** in Android Studio to synchronize the project with the new dependency. (For more information, see Add build dependencies.)

To conveniently run our auth server (created in the previous section) from the project root, we include it as a new module in `settings.gradle` . (For more information, see Structuring and Building a Software Component with Gradle.)

Kotlin Script:

```
// ./settings.gradle

// ...

rootProject.name = "Chatterbox"
include ':app'
include ':auth-server'
```

From now on, as `auth-server` is included in the Android project, you can run the auth server with the following command from the project's root:

Shell:

```
./gradlew :auth-server:run
```

Connect to a Chat Room and Observe Connection Updates

To open a chat-room connection, we use the onCreate() activity lifecycle callback, which fires when the activity is first created. The ChatRoom constructor requires us to provide `region` and `tokenProvider` to instantiate a room connection.

Note: The `fetchChatToken` function in the snippet below will be implemented in the next section.

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt
```

```
package com.chatterbox.myapp

// ...
import androidx.appcompat.app.AppCompatActivity
// ...

// AWS region of the room that was created in Getting Started with Amazon IVS Chat
const val REGION = "us-west-2"

class MainActivity : AppCompatActivity() {
    private var room: ChatRoom? = null
    // ...

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

      // Create room instance
      room = ChatRoom(REGION, ::fetchChatToken)
   }

// ...
}
```

Displaying and reacting to changes in a chat room's connection are essential parts of making a chat app like `chatterbox`. Before we can start interacting with the room, we must subscribe to chat-room connection-state events, to get updates.

ChatRoom expects us to attach a ChatRoomListener interface implementation for raising lifecycle events. For now, listener functions will log only confirmation messages, when invoked:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

// ...
package com.chatterbox.myapp
// ...
const val TAG = "IVSChat-App"

class MainActivity : AppCompatActivity() {
// ...
```

```
    private val roomListener = object : ChatRoomListener {
        override fun onConnecting(room: ChatRoom) {
            Log.d(TAG, "onConnecting")
        }

        override fun onConnected(room: ChatRoom) {
            Log.d(TAG, "onConnected")
        }

        override fun onDisconnected(room: ChatRoom, reason: DisconnectReason) {
            Log.d(TAG, "onDisconnected $reason")
        }

        override fun onMessageReceived(room: ChatRoom, message: ChatMessage) {
            Log.d(TAG, "onMessageReceived $message")
        }

        override fun onMessageDeleted(room: ChatRoom, event: DeleteMessageEvent) {
            Log.d(TAG, "onMessageDeleted $event")
        }

        override fun onEventReceived(room: ChatRoom, event: ChatEvent) {
            Log.d(TAG, "onEventReceived $event")
        }

        override fun onUserDisconnected(room: ChatRoom, event:    DisconnectUserEvent)
 {
            Log.d(TAG, "onUserDisconnected $event")
        }
    }
}
```

Now that we have `ChatRoomListener` implemented, we attach it to our room instance:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   binding = ActivityMainBinding.inflate(layoutInflater)
```

```
    setContentView(binding.root)

    // Create room instance
    room = ChatRoom(REGION, ::fetchChatToken).apply {
        listener = roomListener
    }
}

private val roomListener = object : ChatRoomListener {
// ...
}
```

After this, we need to provide the ability to read the room-connection state. We will keep it in the `MainActivity.kt` property and initialize it to the default DISCONNECTED state for rooms (see `ChatRoom state` in the IVS Chat Android SDK Reference). To be able to keep local state up to date, we need to implement a state-updater function; let's call it `updateConnectionState`:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

enum class ConnectionState {
    CONNECTED,
    DISCONNECTED,
    LOADING
}

class MainActivity : AppCompatActivity() {
    private var connectionState = ConnectionState.DISCONNECTED
// ...

    private fun updateConnectionState(state: ConnectionState) {
        connectionState = state

        when (state) {
            ConnectionState.CONNECTED -> {
                Log.d(TAG, "room connected")
            }
            ConnectionState.DISCONNECTED -> {
                Log.d(TAG, "room disconnected")
```

```
            }
            ConnectionState.LOADING -> {
                Log.d(TAG, "room loading")
            }
        }
    }
}
```

Next, we integrate our state-updater function with the ChatRoom.listener property:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
// ...

    private val roomListener = object : ChatRoomListener {
        override fun onConnecting(room: ChatRoom) {
            Log.d(TAG, "onConnecting")
            runOnUiThread {
                updateConnectionState(ConnectionState.LOADING)
            }
        }

        override fun onConnected(room: ChatRoom) {
            Log.d(TAG, "onConnected")
            runOnUiThread {
                updateConnectionState(ConnectionState.CONNECTED)
            }
        }

        override fun onDisconnected(room: ChatRoom, reason: DisconnectReason) {
            Log.d(TAG, "[${Thread.currentThread().name}] onDisconnected")
            runOnUiThread {
                updateConnectionState(ConnectionState.DISCONNECTED)
            }
        }
    }
}
```

Now that we have the ability to save, listen, and react to ChatRoom state updates, it's time to initialize a connection:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

enum class ConnectionState {
   CONNECTED,
   DISCONNECTED,
   LOADING
}

class MainActivity : AppCompatActivity() {
   private var connectionState = ConnectionState.DISCONNECTED
// ...

   private fun connect() {
      try {
         room?.connect()
      } catch (ex: Exception) {
         Log.e(TAG, "Error while calling connect()", ex)
      }
   }

   private val roomListener = object : ChatRoomListener {
      // ...
      override fun onConnecting(room: ChatRoom) {
         Log.d(TAG, "onConnecting")
         runOnUiThread {
            updateConnectionState(ConnectionState.LOADING)
         }
      }

      override fun onConnected(room: ChatRoom) {
         Log.d(TAG, "onConnected")
         runOnUiThread {
            updateConnectionState(ConnectionState.CONNECTED)
         }
      }
      // ...
```

```
    }
}
```

Build a Token Provider

It's time to create a function responsible for creating and managing chat tokens in our application. In this example we use the Retrofit HTTP client for Android.

Before we can send any network traffic, we must set up a network-security configuration for Android. (For more information, see Network security configuration.) We begin with adding network permissions to the App Manifest file. Note the added `user-permission` tag and `networkSecurityConfig` attribute, which will point to our new network-security configuration. *In the code below, replace `<version>` with the current version number of the Chat Android SDK (e.g., 1.0.0).*

XML:

```
// ./app/src/main/AndroidManifest.xml

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.chatterbox.myapp">
    <uses-permission android:name="android.permission.INTERNET" />
    <application
        android:allowBackup="true"
        android:fullBackupContent="@xml/backup_rules"
        android:label="@string/app_name"
        android:networkSecurityConfig="@xml/network_security_config"
// ...

// ./app/build.gradle


dependencies {
   implementation("com.amazonaws:ivs-chat-messaging:<version>")
// ...

   implementation("com.squareup.retrofit2:retrofit:2.9.0")
}
```

Declare `10.0.2.2` and `localhost` domains as trusted, to start exchanging messages with our backend:

XML:

```
// ./app/src/main/res/xml/network_security_config.xml

<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">10.0.2.2</domain>
        <domain includeSubdomains="true">localhost</domain>
    </domain-config>
</network-security-config>
```

Next, we need to add a new dependency, along with Gson converter addition for parsing HTTP responses. *In the code below, replace `<version>` with the current version number of the Chat Android SDK (e.g., 1.0.0).*

Kotlin Script:

```
// ./app/build.gradle

dependencies {
   implementation("com.amazonaws:ivs-chat-messaging:<version>")
// ...

   implementation("com.squareup.retrofit2:retrofit:2.9.0")
}
```

To retrieve a chat token, we need to make a POST HTTP request from our `chatterbox` app. We define the request in an interface for Retrofit to implement. (See Retrofit documentation. Also familiarize yourself with the CreateChatToken operation specification.)

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/network/ApiService.kt

package com.chatterbox.myapp.network
// ...
```

```
import androidx.annotation.Keep
import com.amazonaws.ivs.chat.messaging.ChatToken
import retrofit2.Call
import retrofit2.http.Body
import retrofit2.http.POST

data class CreateTokenParams(var userId: String, var roomIdentifier: String)

interface ApiService {
   @POST("create_chat_token")
   fun createChatToken(@Body params: CreateTokenParams): Call<ChatToken>
}
```

Now, with networking set up, it's time to add a function responsible for creating and managing our chat token. We add it to `MainActivity.kt`, which was automatically created when the project was generated:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt


package com.chatterbox.myapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import com.amazonaws.ivs.chat.messaging.*
import com.chatterbox.myapp.network.CreateTokenParams
import com.chatterbox.myapp.network.RetrofitFactory
import retrofit2.Call
import java.io.IOException
import retrofit2.Callback
import retrofit2.Response

// custom tag for logging purposes
const val TAG = "IVSChat-App"

// any ID to be associated with auth token
const val USER_ID = "test user id"
// ID of the room the app wants to access. Must be an ARN. See Amazon Resource
 Names(ARNs)
const val ROOM_ID = "arn:aws:..."
```

```
// AWS region of the room that was created in Getting Started with Amazon IVS Chat
const val REGION = "us-west-2"

class MainActivity : AppCompatActivity() {
   private val service = RetrofitFactory.makeRetrofitService()
   private lateinit var userId: String

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

   private fun fetchChatToken(callback: ChatTokenCallback) {
      val params = CreateTokenParams(userId, ROOM_ID)
      service.createChatToken(params).enqueue(object : Callback<ChatToken> {
         override fun onResponse(call: Call<ChatToken>, response: Response<ChatToken>)
 {
            val token = response.body()
            if (token == null) {
               Log.e(TAG, "Received empty token response")
               callback.onFailure(IOException("Empty token response"))
               return
            }

            Log.d(TAG, "Received token response $token")
            callback.onSuccess(token)
         }

         override fun onFailure(call: Call<ChatToken>, throwable: Throwable) {
            Log.e(TAG, "Failed to fetch token", throwable)
            callback.onFailure(throwable)
         }
      })
   }
}
```

Next Steps

Now that you've established a chat-room connection, proceed to Part 2 of this Android tutorial, Messages and Events.

IVS Chat Client Messaging SDK: Android Tutorial Part 2: Messages and Events

This second (and last) part of the tutorial is broken up into several sections:

1. the section called “Create a UI for Sending Messages”
   a. the section called “UI Main Layout”
   b. the section called “UI Abstracted Text Cell to Display Text Consistently”
   c. the section called “UI Left Chat Message”
   d. the section called “UI Right Chat Message”
   e. the section called “UI Additional Color Values”
2. the section called “Apply View Binding”
3. the section called “Manage Chat-Message Requests”
4. the section called “Final Steps”

For full SDK documentation, start with Amazon IVS Chat Client Messaging SDK (here in the *Amazon IVS Chat User Guide*) and the Chat Client Messaging: SDK for Android Reference (on GitHub).

Prerequisite

Be sure you have completed Part 1 of this tutorial, Chat Rooms.

Create a UI for Sending Messages

Now that we successfully initialized the chat room connection, it’s time to send our first message. For this feature, a UI is needed. We will add:

- `connect/disconnect` button
- Message input with `send` button
- Dynamic messages list. To build this, we use Android Jetpack RecyclerView.

UI Main Layout

See Android Jetpack Layouts in the Android developer documentation.

XML:

```
// ./app/src/main/res/layout/activity_main.xml


<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout 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">

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
                  xmlns:app="http://schemas.android.com/apk/res-auto"
                  android:id="@+id/connect_view"
                  android:layout_width="match_parent"
                  android:layout_height="match_parent"
                  android:gravity="center"
                  android:orientation="vertical">

        <androidx.cardview.widget.CardView
                android:id="@+id/connect_button"
                android:layout_width="match_parent"
                android:layout_height="48dp"
                android:layout_gravity=""
                android:layout_marginStart="16dp"
                android:layout_marginTop="4dp"
                android:layout_marginEnd="16dp"
                android:clickable="true"
                android:elevation="16dp"
                android:focusable="true"
                android:foreground="?android:attr/selectableItemBackground"
                app:cardBackgroundColor="@color/purple_500"
                app:cardCornerRadius="10dp">

            <TextView
                    android:id="@+id/connect_text"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
```

```
                android:layout_alignParentEnd="true"
                android:layout_gravity="center"
                android:layout_weight="1"
                android:paddingHorizontal="12dp"
                android:text="Connect"
                android:textColor="@color/white"
                android:textSize="16sp"/>

        <ProgressBar
                android:id="@+id/activity_indicator"
                android:layout_width="20dp"
                android:layout_height="20dp"
                android:layout_gravity="center"
                android:layout_marginHorizontal="20dp"
                android:indeterminateOnly="true"
                android:indeterminateTint="@color/white"
                android:indeterminateTintMode="src_atop"
                android:keepScreenOn="true"
                android:visibility="gone"/>
    </androidx.cardview.widget.CardView>

</LinearLayout>

<androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/chat_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:clipToPadding="false"
        android:visibility="visible"
        tools:context=".MainActivity">

    <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            app:layout_constraintBottom_toTopOf="@+id/layout_message_input"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent">

        <androidx.recyclerview.widget.RecyclerView
                android:id="@+id/recycler_view"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:clipToPadding="false"
```

```
                    android:paddingTop="70dp"
                    android:paddingBottom="20dp"/>
        </RelativeLayout>

        <RelativeLayout
                android:id="@+id/layout_message_input"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:background="@android:color/white"
                android:clipToPadding="false"
                android:drawableTop="@android:color/black"
                android:elevation="18dp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintStart_toStartOf="parent">

            <EditText
                    android:id="@+id/message_edit_text"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:layout_centerVertical="true"
                    android:layout_marginStart="16dp"
                    android:layout_toStartOf="@+id/send_button"
                    android:background="@android:color/transparent"
                    android:hint="Enter Message"
                    android:inputType="text"
                    android:maxLines="6"
                    tools:ignore="Autofill"/>

            <Button
                    android:id="@+id/send_button"
                    android:layout_width="84dp"
                    android:layout_height="48dp"
                    android:layout_alignParentEnd="true"
                    android:background="@color/black"
                    android:foreground="?android:attr/selectableItemBackground"
                    android:text="Send"
                    android:textColor="@color/white"
                    android:textSize="12dp"/>
        </RelativeLayout>
    </androidx.constraintlayout.widget.ConstraintLayout>


</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

UI Abstracted Text Cell to Display Text Consistently

XML:

```
// ./app/src/main/res/layout/common_cell.xml

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:id="@+id/layout_container"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:background="@color/light_gray"
              android:minWidth="100dp"
              android:orientation="vertical">

    <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="horizontal">

        <TextView
                android:id="@+id/card_message_me_text_view"
                android:layout_width="wrap_content"
                android:layout_height="match_parent"
                android:layout_marginBottom="8dp"
                android:maxWidth="260dp"
                android:paddingLeft="12dp"
                android:paddingTop="8dp"
                android:paddingRight="12dp"
                android:text="This is a Message"
                android:textColor="#ffffff"
                android:textSize="16sp"/>

        <TextView
                android:id="@+id/failed_mark"
                android:layout_width="40dp"
                android:layout_height="match_parent"
                android:paddingRight="5dp"
                android:src="@drawable/ic_launcher_background"
                android:text="!"
                android:textAlignment="viewEnd"
                android:textColor="@color/white"
                android:textSize="25dp"
```

```
                android:visibility="gone"/>
    </LinearLayout>

</LinearLayout>
```

UI Left Chat Message

XML:

```
// ./app/src/main/res/layout/card_view_left.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              xmlns:app="http://schemas.android.com/apk/res-auto"
              android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:layout_marginStart="8dp"
              android:layout_marginBottom="12dp"
              android:orientation="vertical">

    <TextView
            android:id="@+id/username_edit_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="UserName"/>

    <androidx.constraintlayout.widget.ConstraintLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content">

        <androidx.cardview.widget.CardView
                android:id="@+id/card_message_other"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_gravity="left"
                android:layout_marginBottom="4dp"
                android:foreground="?android:attr/selectableItemBackground"
                app:cardBackgroundColor="@color/light_gray_2"
                app:cardCornerRadius="10dp"
                app:cardElevation="0dp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintStart_toStartOf="parent">
```

```
            <include layout="@layout/common_cell"/>
        </androidx.cardview.widget.CardView>

        <TextView
                android:id="@+id/dateText"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginLeft="4dp"
                android:layout_marginBottom="4dp"
                android:text="10:00"
                app:layout_constraintBottom_toBottomOf="@+id/card_message_other"
                app:layout_constraintLeft_toRightOf="@+id/card_message_other"/>
    </androidx.constraintlayout.widget.ConstraintLayout>


</LinearLayout>
```

UI Right Chat Message

XML:

```
// ./app/src/main/res/layout/card_view_right.xml

<?xml version="1.0" encoding="utf-8"?>

<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://
schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
                android:layout_width="match_parent"
               android:layout_height="wrap_content"
android:layout_marginEnd="8dp">

    <androidx.cardview.widget.CardView
            android:id="@+id/card_message_me"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="right"
            android:layout_marginBottom="10dp"
            android:foreground="?android:attr/selectableItemBackground"
            app:cardBackgroundColor="@color/purple_500"
            app:cardCornerRadius="10dp"
            app:cardElevation="0dp"
            app:cardPreventCornerOverlap="false"
```

```
            app:cardUseCompatPadding="true"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent">

        <include layout="@layout/common_cell"/>

    </androidx.cardview.widget.CardView>

    <TextView
            android:id="@+id/dateText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginRight="12dp"
            android:layout_marginBottom="4dp"
            android:text="10:00"
            app:layout_constraintBottom_toBottomOf="@+id/card_message_me"
            app:layout_constraintRight_toLeftOf="@+id/card_message_me"/>

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

UI Additional Color Values

XML:

```
// ./app/src/main/res/values/colors.xml

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!--    ...-->
    <color name="dark_gray">#4F4F4F</color>
    <color name="blue">#186ED3</color>
    <color name="dark_red">#b30000</color>
    <color name="light_gray">#B7B7B7</color>
    <color name="light_gray_2">#eef1f6</color>
</resources>
```

Apply View Binding

We leverage the Android View Binding feature to be able to reference binding classes for our XML layout. To enable the feature, set the `viewBinding` build option to `true` in `./app/build.gradle`:

Kotlin Script:

```
 // ./app/build.gradle

android {
//    ...

    buildFeatures {
        viewBinding = true
    }
//    ...
}
```

Now it's time to connect the UI with our Kotlin code:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt
package com.chatterbox.myapp
// ...
const val TAG = "Chatterbox-MyApp"


class MainActivity : AppCompatActivity() {
//    ...

    private fun sendMessage(request: SendMessageRequest) {
        try {
            room?.sendMessage(
                request,
                object : SendMessageCallback {
                    override fun onRejected(request: SendMessageRequest, error:
 ChatError) {
                        runOnUiThread {
                            entries.addFailedRequest(request)
                            scrollToBottom()
                            Log.e(TAG, "Message rejected: ${error.errorMessage}")
                        }
                    }
                }
            )

            entries.addPendingRequest(request)
```

```
            binding.messageEditText.text.clear()
            scrollToBottom()
        } catch (error: Exception) {
            Log.e(TAG, error.message ?: "Unknown error occurred")
        }
    }

    private fun scrollToBottom() {
        binding.recyclerView.smoothScrollToPosition(entries.size - 1)
    }

    private fun sendButtonClick(view: View) {
        val content = binding.messageEditText.text.toString()
        if (content.trim().isEmpty()) {
            return
        }

        val request = SendMessageRequest(content)
        sendMessage(request)
    }
}
```

We also add methods to delete messages and disconnect users from the chat, which can be invoked using the chat-message context menu:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
//    ...

class MainActivity : AppCompatActivity() {
//    ...

    private fun deleteMessage(request: DeleteMessageRequest) {
        room?.deleteMessage(
            request,
            object : DeleteMessageCallback {
                override fun onRejected(request: DeleteMessageRequest, error:
 ChatError) {
                    runOnUiThread {
```

```
                        Log.d(TAG, "Delete message rejected: ${error.errorMessage}")
                    }
                }
            }
        )
    }

    private fun disconnectUser(request: DisconnectUserRequest) {
        room?.disconnectUser(
            request,
            object : DisconnectUserCallback {
                override fun onRejected(request: DisconnectUserRequest, error:
 ChatError) {
                    runOnUiThread {
                        Log.d(TAG, "Disconnect user rejected: ${error.errorMessage}")
                    }
                }
            }
        )
    }
}
```

Manage Chat-Message Requests

We need a way to manage our chat-message requests through all their possible states:

- Pending — A message was sent to a chat room but is not yet confirmed or rejected.
- Confirmed — A message was sent by the chat room to all users (including us).
- Rejected — A message was rejected by the chat room with an error object.

We will keep unresolved chat requests and chat messages in a list. The list merits a separate class, which we call `ChatEntries.kt`:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/ChatEntries.kt

package com.chatterbox.myapp

import com.amazonaws.ivs.chat.messaging.entities.ChatMessage
import com.amazonaws.ivs.chat.messaging.requests.SendMessageRequest
```

```
sealed class ChatEntry() {
    class Message(val message: ChatMessage) : ChatEntry()
    class PendingRequest(val request: SendMessageRequest) : ChatEntry()
    class FailedRequest(val request: SendMessageRequest) : ChatEntry()
}

class ChatEntries {
    /* This list is kept in sorted order. ChatMessages are sorted by date, while
 pending and failed requests are kept in their original insertion point. */
    val entries = mutableListOf<ChatEntry>()
    var adapter: ChatListAdapter? = null

    val size get() = entries.size

    /**
     * Insert pending request at the end.
     */
    fun addPendingRequest(request: SendMessageRequest) {
        val insertIndex = entries.size
        entries.add(insertIndex, ChatEntry.PendingRequest(request))
        adapter?.notifyItemInserted(insertIndex)
    }

    /**
     * Insert received message at proper place based on sendTime. This can cause
 removal of pending requests.
     */
    fun addReceivedMessage(message: ChatMessage) {
        /* Skip if we have already handled that message. */
        val existingIndex = entries.indexOfLast { it is ChatEntry.Message &&
 it.message.id == message.id }
        if (existingIndex != -1) {
            return
        }

        val removeIndex = entries.indexOfLast {
            it is ChatEntry.PendingRequest && it.request.requestId == message.requestId
        }
        if (removeIndex != -1) {
            entries.removeAt(removeIndex)
        }
```

```
        val insertIndexRaw = entries.indexOfFirst { it is ChatEntry.Message &&
it.message.sendTime > message.sendTime }
        val insertIndex = if (insertIndexRaw == -1) entries.size else insertIndexRaw
        entries.add(insertIndex, ChatEntry.Message(message))

        if (removeIndex == -1) {
            adapter?.notifyItemInserted(insertIndex)
        } else if (removeIndex == insertIndex) {
            adapter?.notifyItemChanged(insertIndex)
        } else {
            adapter?.notifyItemRemoved(removeIndex)
            adapter?.notifyItemInserted(insertIndex)
        }
    }

    fun addFailedRequest(request: SendMessageRequest) {
        val removeIndex = entries.indexOfLast {
            it is ChatEntry.PendingRequest && it.request.requestId == request.requestId
        }
        if (removeIndex != -1) {
            entries.removeAt(removeIndex)
            entries.add(removeIndex, ChatEntry.FailedRequest(request))
            adapter?.notifyItemChanged(removeIndex)
        } else {
            val insertIndex = entries.size
            entries.add(insertIndex, ChatEntry.FailedRequest(request))
            adapter?.notifyItemInserted(insertIndex)
        }
    }

    fun removeMessage(messageId: String) {
        val removeIndex = entries.indexOfFirst { it is ChatEntry.Message &&
it.message.id == messageId }
        entries.removeAt(removeIndex)
        adapter?.notifyItemRemoved(removeIndex)
    }

    fun removeFailedRequest(requestId: String) {
        val removeIndex = entries.indexOfFirst { it is ChatEntry.FailedRequest &&
it.request.requestId == requestId }
        entries.removeAt(removeIndex)
        adapter?.notifyItemRemoved(removeIndex)
    }
```

```
    fun removeAll() {
        entries.clear()
    }
}
```

To connect our list with the UI, we use an Adapter. For more information, see Binding to Data with AdapterView and Generated binding classes.

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/ChatListAdapter.kt

package com.chatterbox.myapp

import android.content.Context
import android.graphics.Color
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.LinearLayout
import android.widget.TextView
import androidx.core.content.ContextCompat
import androidx.core.view.isGone
import androidx.recyclerview.widget.RecyclerView
import com.amazonaws.ivs.chat.messaging.requests.DisconnectUserRequest
import java.text.DateFormat


class ChatListAdapter(
    private val entries: ChatEntries,
    private val onDisconnectUser: (request: DisconnectUserRequest) -> Unit,
) :
    RecyclerView.Adapter<ChatListAdapter.ViewHolder>() {
    var context: Context? = null
    var userId: String? = null

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val container: LinearLayout = view.findViewById(R.id.layout_container)
        val textView: TextView = view.findViewById(R.id.card_message_me_text_view)
        val failedMark: TextView = view.findViewById(R.id.failed_mark)
        val userNameText: TextView? = view.findViewById(R.id.username_edit_text)
        val dateText: TextView? = view.findViewById(R.id.dateText)
    }
```

```
   override fun onCreateViewHolder(viewGroup: ViewGroup, viewType: Int): ViewHolder {
       if (viewType == 0) {
           val rightView =
LayoutInflater.from(viewGroup.context).inflate(R.layout.card_view_right, viewGroup,
false)
           return ViewHolder(rightView)
       }
       val leftView =
LayoutInflater.from(viewGroup.context).inflate(R.layout.card_view_left, viewGroup,
false)
       return ViewHolder(leftView)
   }

   override fun getItemViewType(position: Int): Int {
       // Int 0 indicates to my message while Int 1 to other message
       val chatMessage = entries.entries[position]
       return if (chatMessage is ChatEntry.Message &&
chatMessage.message.sender.userId != userId) 1 else 0
   }

   override fun onBindViewHolder(viewHolder: ViewHolder, position: Int) {
       return when (val entry = entries.entries[position]) {
           is ChatEntry.Message -> {
               viewHolder.textView.text = entry.message.content

               val bgColor = if (entry.message.sender.userId == userId) {
                   R.color.purple_500
               } else {
                   R.color.light_gray_2
               }

viewHolder.container.setBackgroundColor(ContextCompat.getColor(context!!, bgColor))

               if (entry.message.sender.userId != userId) {
                   viewHolder.textView.setTextColor(Color.parseColor("#000000"))
               }

               viewHolder.failedMark.isGone = true

               viewHolder.itemView.setOnCreateContextMenuListener { menu, _, _ ->
                   menu.add("Kick out").setOnMenuItemClickListener {
                       val request =
DisconnectUserRequest(entry.message.sender.userId, "Some reason")
```

```
                        onDisconnectUser(request)
                        true
                    }
                }

                viewHolder.userNameText?.text = entry.message.sender.userId
                viewHolder.dateText?.text =

 DateFormat.getTimeInstance(DateFormat.SHORT).format(entry.message.sendTime)
            }

            is ChatEntry.PendingRequest -> {

 viewHolder.container.setBackgroundColor(ContextCompat.getColor(context!!,
 R.color.light_gray))
                viewHolder.textView.text = entry.request.content
                viewHolder.failedMark.isGone = true
                viewHolder.itemView.setOnCreateContextMenuListener(null)
                viewHolder.dateText?.text = "Sending"
            }

            is ChatEntry.FailedRequest -> {
                viewHolder.textView.text = entry.request.content

 viewHolder.container.setBackgroundColor(ContextCompat.getColor(context!!,
 R.color.dark_red))
                viewHolder.failedMark.isGone = false
                viewHolder.dateText?.text = "Failed"
            }
        }
    }

    override fun onAttachedToRecyclerView(recyclerView: RecyclerView) {
        super.onAttachedToRecyclerView(recyclerView)
        context = recyclerView.context
    }

    override fun getItemCount() = entries.entries.size
}
```

Final Steps

It is time to hook up our new adapter, binding `ChatEntries` class to `MainActivity`:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

import com.chatterbox.myapp.databinding.ActivityMainBinding
import com.chatterbox.myapp.ChatListAdapter
import com.chatterbox.myapp.ChatEntries

class MainActivity : AppCompatActivity() {
    // ...
    private var entries = ChatEntries()
    private lateinit var adapter: ChatListAdapter
    private lateinit var binding: ActivityMainBinding

    /* see https://developer.android.com/topic/libraries/data-binding/generated-
binding#create */
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        /* Create room instance. */
        room = ChatRoom(REGION, ::fetchChatToken).apply {
            listener = roomListener
        }

        binding.sendButton.setOnClickListener(::sendButtonClick)
        binding.connectButton.setOnClickListener { connect() }

        setUpChatView()

        updateConnectionState(ConnectionState.DISCONNECTED)
    }

    private fun setUpChatView() {
        /* Setup Android Jetpack RecyclerView - see https://developer.android.com/
develop/ui/views/layout/recyclerview.*/
        adapter = ChatListAdapter(entries, ::disconnectUser)
        entries.adapter = adapter
```

```
        val recyclerViewLayoutManager = LinearLayoutManager(this@MainActivity,
 LinearLayoutManager.VERTICAL, false)
        binding.recyclerView.layoutManager = recyclerViewLayoutManager
        binding.recyclerView.adapter = adapter

        binding.sendButton.setOnClickListener(::sendButtonClick)
        binding.messageEditText.setOnEditorActionListener { _, _, event ->
            val isEnterDown = (event.action == KeyEvent.ACTION_DOWN) && (event.keyCode
 == KeyEvent.KEYCODE_ENTER)
            if (!isEnterDown) {
                return@setOnEditorActionListener false
            }

            sendButtonClick(binding.sendButton)
            return@setOnEditorActionListener true
        }
    }
}
```

As we already have a class responsible for keeping track of our chat requests (`ChatEntries`), we are ready to implement code for manipulating `entries` in `roomListener`. We will update `entries` and `connectionState` accordingly to the event we are responding to:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
    //...


    private fun sendMessage(request: SendMessageRequest) {
    //...

    }

    private fun scrollToBottom() {
        binding.recyclerView.smoothScrollToPosition(entries.size - 1)
    }
```

```
    private val roomListener = object : ChatRoomListener {
        override fun onConnecting(room: ChatRoom) {
            Log.d(TAG, "[${Thread.currentThread().name}] onConnecting")
            runOnUiThread {
                updateConnectionState(ConnectionState.LOADING)
            }
        }

        override fun onConnected(room: ChatRoom) {
            Log.d(TAG, "[${Thread.currentThread().name}] onConnected")
            runOnUiThread {
                updateConnectionState(ConnectionState.CONNECTED)
            }
        }

        override fun onDisconnected(room: ChatRoom, reason: DisconnectReason) {
            Log.d(TAG, "[${Thread.currentThread().name}] onDisconnected")
            runOnUiThread {
                updateConnectionState(ConnectionState.DISCONNECTED)
                entries.removeAll()
            }
        }

        override fun onMessageReceived(room: ChatRoom, message: ChatMessage) {
            Log.d(TAG, "[${Thread.currentThread().name}] onMessageReceived $message")
            runOnUiThread {
                entries.addReceivedMessage(message)
                scrollToBottom()
            }
        }

        override fun onEventReceived(room: ChatRoom, event: ChatEvent) {
            Log.d(TAG, "[${Thread.currentThread().name}] onEventReceived $event")
        }

        override fun onMessageDeleted(room: ChatRoom, event: DeleteMessageEvent) {
            Log.d(TAG, "[${Thread.currentThread().name}] onMessageDeleted $event")
        }

        override fun onUserDisconnected(room: ChatRoom, event: DisconnectUserEvent) {
            Log.d(TAG, "[${Thread.currentThread().name}] onUserDisconnected $event")
        }
    }
```

```
}
```

Now you should be able to run your application! (See Build and run your app.) Remember to have your backend server running when using the app. You can spin it up from the terminal at the root of our project with this command: `./gradlew :auth-server:run` or by executing the `auth-server:run` Gradle task directly from Android Studio.

IVS Chat Client Messaging SDK: Kotlin Coroutines Tutorial Part 1: Chat Rooms

This is the first of a two-part tutorial. You will learn the essentials of working with the Amazon IVS Chat Messaging SDK by building a fully functional Android app using the Kotlin programming language and coroutines. We call the app *Chatterbox*.

Before you start the module, take a few minutes to familiarize yourself with the prerequisites, key concepts behind chat tokens, and the backend server necessary for creating chat rooms.

These tutorials are created for experienced Android developers who are new to the IVS Chat Messaging SDK. You will need to be comfortable with the Kotlin programming language and creating UIs on the Android platform.

This first part of the tutorial is broken up into several sections:

1. the section called “Set Up a Local Authentication/Authorization Server”
2. the section called “Create a Chatterbox Project”
3. the section called “Connect to a Chat Room and Observe Connection Updates”
4. the section called “Build a Token Provider”
5. the section called “Next Steps”

For full SDK documentation, start with Amazon IVS Chat Client Messaging SDK (here in the *Amazon IVS Chat User Guide*) and the Chat Client Messaging: SDK for Android Reference (on GitHub).

Prerequisites

- Be familiar with Kotlin and creating applications on the Android platform. If you are unfamiliar with creating applications for Android, learn the basics in the Build your first app guide for Android developers.

- Read and understand *Getting Started with IVS Chat*.
- Create an AWS IAM user with the `CreateChatToken` and `CreateRoom` capabilities defined in an existing IAM policy. (See *Getting Started with IVS Chat*.)
- Ensure that the secret/access keys for this user are stored in an AWS credentials file. For instructions, see the AWS CLI User Guide (especially Configuration and credential file settings).
- Create a chat room and save its ARN. See *Getting Started with IVS Chat*. (If you don't save the ARN, you can look it up later with the console or Chat API.)

Set Up a Local Authentication/Authorization Server

Your backend server will be responsible for both creating chat rooms and generating the chat tokens needed for the IVS Chat Android SDK to authenticate and authorize your clients for your chat rooms.

See Create a Chat Token in *Getting Started with Amazon IVS Chat*. As shown in the flowchart there, your server-side code is responsible for creating a chat token. This means your app must provide its own means of generating a chat token by requesting one from your server-side application.

We use the Ktor framework to create a live local server that manages the creation of chat tokens using your local AWS environment.

At this point, we expect you have your AWS credentials set up correctly. For step-by-step instructions, see Set up AWS temporary credentials and AWS Region for development.

Create a new directory and call it `chatterbox` and inside it, another one, called `auth-server`.

Our server folder will have the following structure:

```
- auth-server
  - src
    - main
      - kotlin
        - com
          - chatterbox
            - authserver
              - Application.kt
       - resources
         - application.conf
         - logback.xml
   - build.gradle.kts
```

Note: you can directly copy/paste the code here into the referenced files.

Next, we add all the necessary dependencies and plugins for our auth server to work:

Kotlin Script:

```
// ./auth-server/build.gradle.kts

plugins {
   application
   kotlin("jvm")
   kotlin("plugin.serialization").version("1.7.10")
}

application {
   mainClass.set("io.ktor.server.netty.EngineMain")
}

dependencies {
   implementation("software.amazon.awssdk:ivschat:2.18.1")
   implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.7.20")

   implementation("io.ktor:ktor-server-core:2.1.3")
   implementation("io.ktor:ktor-server-netty:2.1.3")
   implementation("io.ktor:ktor-server-content-negotiation:2.1.3")
   implementation("io.ktor:ktor-serialization-kotlinx-json:2.1.3")

   implementation("ch.qos.logback:logback-classic:1.4.4")
}
```

Now we need to set up logging functionality for the auth server. (For more information, see Configure logger.)

XML:

```
// ./auth-server/src/main/resources/logback.xml

<configuration>
   <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
      <encoder>
         <pattern>%d{YYYY-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</
pattern>
      </encoder>
```

```
   </appender>
   <root level="trace">
      <appender-ref ref="STDOUT"/>
   </root>
   <logger name="org.eclipse.jetty" level="INFO"/>
   <logger name="io.netty" level="INFO"/>
</configuration>
```

The Ktor server requires configuration settings, which it automatically loads from the `application.*` file in the `resources` directory, so we add that as well. (For more information, see Configuration in a file.)

HOCON:

```
// ./auth-server/src/main/resources/application.conf

ktor {
   deployment {
      port = 3000
   }
   application {
      modules = [ com.chatterbox.authserver.ApplicationKt.main ]
   }
}
```

Finally, let's implement our server:

Kotlin:

```
// ./auth-server/src/main/kotlin/com/chatterbox/authserver/Application.kt

package com.chatterbox.authserver

import io.ktor.http.*
import io.ktor.serialization.kotlinx.json.*
import io.ktor.server.application.*
import io.ktor.server.plugins.contentnegotiation.*
import io.ktor.server.request.*
import io.ktor.server.response.*
import io.ktor.server.routing.*
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
```

```
import software.amazon.awssdk.services.ivschat.IvschatClient
import software.amazon.awssdk.services.ivschat.model.CreateChatTokenRequest

@Serializable
data class ChatTokenParams(var userId: String, var roomIdentifier: String)

@Serializable
data class ChatToken(
   val token: String,
   val sessionExpirationTime: String,
   val tokenExpirationTime: String,
)

fun Application.main() {
   install(ContentNegotiation) {
      json(Json)
   }

   routing {
      post("/create_chat_token") {
         val callParameters = call.receive<ChatTokenParams>()
         val request =
 CreateChatTokenRequest.builder().roomIdentifier(callParameters.roomIdentifier)
            .userId(callParameters.userId).build()
         val token = IvschatClient.create()
            .createChatToken(request)

         call.respond(
            ChatToken(
                token.token(),
                token.sessionExpirationTime().toString(),
                token.tokenExpirationTime().toString()
            )
         )
      }
   }
}
```

Create a Chatterbox Project

To create an Android project, install and open Android Studio.

Follow the steps listed in the official Android Create a Project guide.

- In Choose your project, choose the **Empty Activity** project template for our Chatterbox app.

- In Configure your project, choose the following values for configuration fields:
  - **Name**: My App
  - **Package name**: com.chatterbox.myapp
  - **Save location**: point at the created `chatterbox` directory created in the previous step
  - **Language**: Kotlin
  - **Minimum API level**: API 21: Android 5.0 (Lollipop)

After specifying all the configuration parameters correctly, our file structure inside the `chatterbox` folder should look like the following:

```
- app
  - build.gradle
  ...
- gradle
- .gitignore
- build.gradle
- gradle.properties
- gradlew
- gradlew.bat
- local.properties
- settings.gradle
- auth-server
  - src
    - main
      - kotlin
        - com
          - chatterbox
            - authserver
              - Application.kt
       - resources
         - application.conf
         - logback.xml
   - build.gradle.kts
```

Now that we have a working Android project, we can add com.amazonaws:ivs-chat-messaging and org.jetbrains.kotlinx:kotlinx-coroutines-core to our `build.gradle` dependencies. (For more information on the Gradle build toolkit, see Configure your build.)

Note: At the top of every code snippet, there is a path to the file where you should be making changes in your project. The path is relative to the project's root.

Kotlin:

```
// ./app/build.gradle

plugins {
// ...
}

android {
// ...
}

dependencies {
    implementation 'com.amazonaws:ivs-chat-messaging:1.1.0'
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4'

// ...
}
```

After the new dependency is added, run **Sync Project with Gradle Files** in Android Studio to synchronize the project with the new dependency. (For more information, see Add build dependencies.)

To conveniently run our auth server (created in the previous section) from the project root, we include it as a new module in `settings.gradle`. (For more information, see Structuring and Building a Software Component with Gradle.)

Kotlin Script:

```
// ./settings.gradle

// ...

rootProject.name = "My App"
include ':app'
include ':auth-server'
```

From now on, as `auth-server` is included in the Android project, you can run the auth server with the following command from the project's root:

Shell:

```
./gradlew :auth-server:run
```

Connect to a Chat Room and Observe Connection Updates

To open a chat-room connection, we use the onCreate() activity lifecycle callback, which fires when the activity is first created. The ChatRoom constructor requires us to provide `region` and `tokenProvider` to instantiate a room connection.

Note: The `fetchChatToken` function in the snippet below will be implemented in the next section.

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

// AWS region of the room that was created in Getting Started with Amazon IVS Chat
const val REGION = "us-west-2"

class MainActivity : AppCompatActivity() {
    private var room: ChatRoom? = null
    // ...

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

      // Create room instance
      room = ChatRoom(REGION, ::fetchChatToken)
   }

// ...
}
```

Displaying and reacting to changes in a chat room's connection are essential parts of making a chat app like `chatterbox`. Before we can start interacting with the room, we must subscribe to chat-room connection-state events, to get updates.

In the Chat SDK for coroutine, ChatRoom expects us to handle room lifecycle events in Flow. For now, the functions will log only confirmation messages, when invoked:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

const val TAG = "Chatterbox-MyApp"

class MainActivity : AppCompatActivity() {
// ...

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...

        // Create room instance
        room = ChatRoom(REGION, ::fetchChatToken).apply {
            lifecycleScope.launch {
                stateChanges().collect { state ->
                    Log.d(TAG, "state change to $state")
                }
            }

            lifecycleScope.launch {
                receivedMessages().collect { message ->
                    Log.d(TAG, "messageReceived $message")
                }
            }

            lifecycleScope.launch {
                receivedEvents().collect { event ->
                    Log.d(TAG, "eventReceived $event")
                }
            }

            lifecycleScope.launch {
                deletedMessages().collect { event ->
                    Log.d(TAG, "messageDeleted $event")
                }
            }
```

```
            lifecycleScope.launch {
                disconnectedUsers().collect { event ->
                    Log.d(TAG, "userDisconnected $event")
                }
            }
        }
    }
}
```

After this, we need to provide the ability to read the room-connection state. We will keep it in the `MainActivity.kt` property and initialize it to the default DISCONNECTED state for rooms (see `ChatRoom state` in the IVS Chat Android SDK Reference). To be able to keep local state up to date, we need to implement a state-updater function; let's call it `updateConnectionState`:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
   private var connectionState = ChatRoom.State.DISCONNECTED

// ...

   private fun updateConnectionState(state: ChatRoom.State) {
      connectionState = state

     when (state) {
          ChatRoom.State.CONNECTED -> {
              Log.d(TAG, "room connected")
          }
          ChatRoom.State.DISCONNECTED -> {
              Log.d(TAG, "room disconnected")
          }
          ChatRoom.State.CONNECTING -> {
              Log.d(TAG, "room connecting")
          }
      }
}
```

Next, we integrate our state-updater function with the ChatRoom.listener property:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
// ...

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...

        // Create room instance
        room = ChatRoom(REGION, ::fetchChatToken).apply {
            lifecycleScope.launch {
                stateChanges().collect { state ->
                    Log.d(TAG, "state change to $state")
                    updateConnectionState(state)

                }
            }

      // ...

      }
   }
}
```

Now that we have the ability to save, listen, and react to ChatRoom state updates, it's time to initialize a connection:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
// ...

   private fun connect() {
```

```
      try {
         room?.connect()
      } catch (ex: Exception) {
         Log.e(TAG, "Error while calling connect()", ex)
      }
   }

   // ...
}
```

Build a Token Provider

It's time to create a function responsible for creating and managing chat tokens in our application. In this example we use the Retrofit HTTP client for Android.

Before we can send any network traffic, we must set up a network-security configuration for Android. (For more information, see Network security configuration.) We begin with adding network permissions to the App Manifest file. Note the added `user-permission` tag and `networkSecurityConfig` attribute, which will point to our new network-security configuration. *In the code below, replace `<version>` with the current version number of the Chat Android SDK (e.g., 1.1.0).*

XML:

```
// ./app/src/main/AndroidManifest.xml

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.chatterbox.myapp">
    <uses-permission android:name="android.permission.INTERNET" />
    <application
        android:allowBackup="true"
        android:fullBackupContent="@xml/backup_rules"
        android:label="@string/app_name"
        android:networkSecurityConfig="@xml/network_security_config"
// ...

// ./app/build.gradle


dependencies {
```

```
   implementation("com.amazonaws:ivs-chat-messaging:<version>")
// ...

   implementation("com.squareup.retrofit2:retrofit:2.9.0")
   implementation("com.squareup.retrofit2:converter-gson:2.9.0")
}
```

Declare your local IP address, e.g. `10.0.2.2` and `localhost` domains as trusted, to start exchanging messages with our backend:

XML:

```
// ./app/src/main/res/xml/network_security_config.xml

<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">10.0.2.2</domain>
        <domain includeSubdomains="true">localhost</domain>
    </domain-config>
</network-security-config>
```

Next, we need to add a new dependency, along with Gson converter addition for parsing HTTP responses. *In the code below, replace `<version>` with the current version number of the Chat Android SDK (e.g., 1.1.0).*

Kotlin Script:

```
// ./app/build.gradle

dependencies {
   implementation("com.amazonaws:ivs-chat-messaging:<version>")
// ...

   implementation("com.squareup.retrofit2:retrofit:2.9.0")
   implementation("com.squareup.retrofit2:converter-gson:2.9.0")
}
```

To retrieve a chat token, we need to make a POST HTTP request from our `chatterbox` app. We define the request in an interface for Retrofit to implement. (See Retrofit documentation. Also familiarize yourself with the CreateChatToken operation specification.)

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/network/ApiService.kt

package com.chatterbox.myapp.network

import com.amazonaws.ivs.chat.messaging.ChatToken
import retrofit2.Call
import retrofit2.http.Body
import retrofit2.http.POST

data class CreateTokenParams(var userId: String, var roomIdentifier: String)

interface ApiService {
   @POST("create_chat_token")
   fun createChatToken(@Body params: CreateTokenParams): Call<ChatToken>
}


// ./app/src/main/java/com/chatterbox/myapp/network/RetrofitFactory.kt

package com.chatterbox.myapp.network

import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object RetrofitFactory {
   private const val BASE_URL = "http://10.0.2.2:3000"

   fun makeRetrofitService(): ApiService {
       return Retrofit.Builder()
           .baseUrl(BASE_URL)
           .addConverterFactory(GsonConverterFactory.create())
           .build().create(ApiService::class.java)
   }
}
```

Now, with networking set up, it's time to add a function responsible for creating and managing our chat token. We add it to `MainActivity.kt`, which was automatically created when the project was generated:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import androidx.lifecycle.lifecycleScope
import kotlinx.coroutines.launch
import com.amazonaws.ivs.chat.messaging.*
import com.amazonaws.ivs.chat.messaging.coroutines.*
import com.chatterbox.myapp.network.CreateTokenParams
import com.chatterbox.myapp.network.RetrofitFactory
import retrofit2.Call
import java.io.IOException
import retrofit2.Callback
import retrofit2.Response

// custom tag for logging purposes
const val TAG = "Chatterbox-MyApp"

// any ID to be associated with auth token
const val USER_ID = "test user id"
// ID of the room the app wants to access. Must be an ARN. See Amazon Resource
 Names(ARNs)
const val ROOM_ID = "arn:aws:..."
// AWS region of the room that was created in Getting Started with Amazon IVS Chat
const val REGION = "us-west-2"

class MainActivity : AppCompatActivity() {

   private val service = RetrofitFactory.makeRetrofitService()
   private var userId: String = USER_ID

// ...

   private fun fetchChatToken(callback: ChatTokenCallback) {
      val params = CreateTokenParams(userId, ROOM_ID)
      service.createChatToken(params).enqueue(object : Callback<ChatToken> {
         override fun onResponse(call: Call<ChatToken>, response: Response<ChatToken>)
 {
            val token = response.body()
            if (token == null) {
```

```
                Log.e(TAG, "Received empty token response")
                callback.onFailure(IOException("Empty token response"))
                return
            }

            Log.d(TAG, "Received token response $token")
            callback.onSuccess(token)
        }

        override fun onFailure(call: Call<ChatToken>, throwable: Throwable) {
            Log.e(TAG, "Failed to fetch token", throwable)
            callback.onFailure(throwable)
        }
    })
  }
}
```

Next Steps

Now that you've established a chat-room connection, proceed to Part 2 of this Kotlin Coroutines tutorial, Messages and Events.

IVS Chat Client Messaging SDK: Kotlin Coroutines Tutorial Part 2: Messages and Events

This second (and last) part of the tutorial is broken up into several sections:

1. the section called "Create a UI for Sending Messages"
   a. the section called "UI Main Layout"
   b. the section called "UI Abstracted Text Cell to Display Text Consistently"
   c. the section called "UI Left Chat Message"
   d. the section called "UI Right Message"
   e. the section called "UI Additional Color Values"
2. the section called "Apply View Binding"
3. the section called "Manage Chat-Message Requests"
4. the section called "Final Steps"

For full SDK documentation, start with Amazon IVS Chat Client Messaging SDK (here in the *Amazon IVS Chat User Guide*) and the Chat Client Messaging: SDK for Android Reference (on GitHub).

Prerequisite

Be sure you have completed Part 1 of this tutorial, Chat Rooms.

Create a UI for Sending Messages

Now that we successfully initialized the chat room connection, it's time to send our first message. For this feature, a UI is needed. We will add:

- `connect/disconnect` button
- Message input with `send` button
- Dynamic messages list. To build this, we use Android Jetpack RecyclerView.

UI Main Layout

See Android Jetpack Layouts in the Android developer documentation.

XML:

```
// ./app/src/main/res/layout/activity_main.xml


<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout 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">

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
                  xmlns:app="http://schemas.android.com/apk/res-auto"
                  android:id="@+id/connect_view"
                  android:layout_width="match_parent"
```

```
                android:layout_height="match_parent"
                android:gravity="center"
                android:orientation="vertical">

        <androidx.cardview.widget.CardView
                android:id="@+id/connect_button"
                android:layout_width="match_parent"
                android:layout_height="48dp"
                android:layout_gravity=""
                android:layout_marginStart="16dp"
                android:layout_marginTop="4dp"
                android:layout_marginEnd="16dp"
                android:clickable="true"
                android:elevation="16dp"
                android:focusable="true"
                android:foreground="?android:attr/selectableItemBackground"
                app:cardBackgroundColor="@color/purple_500"
                app:cardCornerRadius="10dp">

            <TextView
                    android:id="@+id/connect_text"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_alignParentEnd="true"
                    android:layout_gravity="center"
                    android:layout_weight="1"
                    android:paddingHorizontal="12dp"
                    android:text="Connect"
                    android:textColor="@color/white"
                    android:textSize="16sp"/>

            <ProgressBar
                    android:id="@+id/activity_indicator"
                    android:layout_width="20dp"
                    android:layout_height="20dp"
                    android:layout_gravity="center"
                    android:layout_marginHorizontal="20dp"
                    android:indeterminateOnly="true"
                    android:indeterminateTint="@color/white"
                    android:indeterminateTintMode="src_atop"
                    android:keepScreenOn="true"
                    android:visibility="gone"/>
        </androidx.cardview.widget.CardView>
```

```
    </LinearLayout>

    <androidx.constraintlayout.widget.ConstraintLayout
            android:id="@+id/chat_view"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:clipToPadding="false"
            android:visibility="visible"
            tools:context=".MainActivity">

        <RelativeLayout
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:orientation="vertical"
                app:layout_constraintBottom_toTopOf="@+id/layout_message_input"
                app:layout_constraintEnd_toEndOf="parent"
                app:layout_constraintStart_toStartOf="parent">

            <androidx.recyclerview.widget.RecyclerView
                    android:id="@+id/recycler_view"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:clipToPadding="false"
                    android:paddingTop="70dp"
                    android:paddingBottom="20dp"/>
        </RelativeLayout>

        <RelativeLayout
                android:id="@+id/layout_message_input"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:background="@android:color/white"
                android:clipToPadding="false"
                android:drawableTop="@android:color/black"
                android:elevation="18dp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintStart_toStartOf="parent">

            <EditText
                    android:id="@+id/message_edit_text"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:layout_centerVertical="true"
                    android:layout_marginStart="16dp"
```

```
                    android:layout_toStartOf="@+id/send_button"
                    android:background="@android:color/transparent"
                    android:hint="Enter Message"
                    android:inputType="text"
                    android:maxLines="6"
                    tools:ignore="Autofill"/>

            <Button
                    android:id="@+id/send_button"
                    android:layout_width="84dp"
                    android:layout_height="48dp"
                    android:layout_alignParentEnd="true"
                    android:background="@color/black"
                    android:foreground="?android:attr/selectableItemBackground"
                    android:text="Send"
                    android:textColor="@color/white"
                    android:textSize="12dp"/>
        </RelativeLayout>
    </androidx.constraintlayout.widget.ConstraintLayout>


</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

UI Abstracted Text Cell to Display Text Consistently

XML:

```
// ./app/src/main/res/layout/common_cell.xml

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:id="@+id/layout_container"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:background="@color/light_gray"
              android:minWidth="100dp"
              android:orientation="vertical">

    <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="horizontal">
```

```
        <TextView
                android:id="@+id/card_message_me_text_view"
                android:layout_width="wrap_content"
                android:layout_height="match_parent"
                android:layout_marginBottom="8dp"
                android:maxWidth="260dp"
                android:paddingLeft="12dp"
                android:paddingTop="8dp"
                android:paddingRight="12dp"
                android:text="This is a Message"
                android:textColor="#ffffff"
                android:textSize="16sp"/>

        <TextView
                android:id="@+id/failed_mark"
                android:layout_width="40dp"
                android:layout_height="match_parent"
                android:paddingRight="5dp"
                android:src="@drawable/ic_launcher_background"
                android:text="!"
                android:textAlignment="viewEnd"
                android:textColor="@color/white"
                android:textSize="25dp"
                android:visibility="gone"/>
    </LinearLayout>

</LinearLayout>
```

UI Left Chat Message

XML:

```
// ./app/src/main/res/layout/card_view_left.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              xmlns:app="http://schemas.android.com/apk/res-auto"
              android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:layout_marginStart="8dp"
              android:layout_marginBottom="12dp"
              android:orientation="vertical">
```

```
    <TextView
            android:id="@+id/username_edit_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="UserName"/>

    <androidx.constraintlayout.widget.ConstraintLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content">

        <androidx.cardview.widget.CardView
                android:id="@+id/card_message_other"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_gravity="left"
                android:layout_marginBottom="4dp"
                android:foreground="?android:attr/selectableItemBackground"
                app:cardBackgroundColor="@color/light_gray_2"
                app:cardCornerRadius="10dp"
                app:cardElevation="0dp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintStart_toStartOf="parent">

            <include layout="@layout/common_cell"/>
        </androidx.cardview.widget.CardView>

        <TextView
                android:id="@+id/dateText"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginLeft="4dp"
                android:layout_marginBottom="4dp"
                android:text="10:00"
                app:layout_constraintBottom_toBottomOf="@+id/card_message_other"
                app:layout_constraintLeft_toRightOf="@+id/card_message_other"/>
    </androidx.constraintlayout.widget.ConstraintLayout>


</LinearLayout>
```

UI Right Message

XML:

```
// ./app/src/main/res/layout/card_view_right.xml

<?xml version="1.0" encoding="utf-8"?>

<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://
schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
                android:layout_width="match_parent"
               android:layout_height="wrap_content"
android:layout_marginEnd="8dp">

    <androidx.cardview.widget.CardView
            android:id="@+id/card_message_me"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="right"
            android:layout_marginBottom="10dp"
            android:foreground="?android:attr/selectableItemBackground"
            app:cardBackgroundColor="@color/purple_500"
            app:cardCornerRadius="10dp"
            app:cardElevation="0dp"
            app:cardPreventCornerOverlap="false"
            app:cardUseCompatPadding="true"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent">

        <include layout="@layout/common_cell"/>

    </androidx.cardview.widget.CardView>

    <TextView
            android:id="@+id/dateText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginRight="12dp"
            android:layout_marginBottom="4dp"
            android:text="10:00"
            app:layout_constraintBottom_toBottomOf="@+id/card_message_me"
            app:layout_constraintRight_toLeftOf="@+id/card_message_me"/>
```

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

UI Additional Color Values

XML:

```
// ./app/src/main/res/values/colors.xml

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!--    ...-->
    <color name="dark_gray">#4F4F4F</color>
    <color name="blue">#186ED3</color>
    <color name="dark_red">#b30000</color>
    <color name="light_gray">#B7B7B7</color>
    <color name="light_gray_2">#eef1f6</color>
</resources>
```

Apply View Binding

We leverage the Android View Binding feature to be able to reference binding classes for our XML layout. To enable the feature, set the `viewBinding` build option to `true` in `./app/build.gradle`:

Kotlin Script:

```
 // ./app/build.gradle

android {
//    ...

    buildFeatures {
        viewBinding = true
    }
//    ...
}
```

Now it's time to connect the UI with our Kotlin code:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt
```

```
package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
    // ...
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Create room instance
        room = ChatRoom(REGION, ::fetchChatToken).apply {
            // ...
        }

        binding.sendButton.setOnClickListener(::sendButtonClick)
        binding.connectButton.setOnClickListener {connect()}

        setUpChatView()

        updateConnectionState(ChatRoom.State.DISCONNECTED)
    }

    private fun sendMessage(request: SendMessageRequest) {
        lifecycleScope.launch {
           try {
               binding.messageEditText.text.clear()
               room?.awaitSendMessage(request)
           } catch (exception: ChatException) {
               Log.e(TAG, "Message rejected: ${exception.message}")
           } catch (exception: Exception) {
               Log.e(TAG, exception.message ?: "Unknown error occurred")
           }
        }
    }

    private fun sendButtonClick(view: View) {
        val content = binding.messageEditText.text.toString()
        if (content.trim().isEmpty()) {
            return
        }
```

```
        val request = SendMessageRequest(content)
        sendMessage(request)
    }
// ...

}
```

We also add methods to delete messages and disconnect users from the chat, which can be invoked using the chat-message context menu:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
//    ...

class MainActivity : AppCompatActivity() {
//    ...

    private fun deleteMessage(request: DeleteMessageRequest) {
        lifecycleScope.launch {
           try {
               room?.awaitDeleteMessage(request)
           } catch (exception: ChatException) {
               Log.e(TAG, "Delete message rejected: ${exception.message}")
           } catch (exception: Exception) {
               Log.e(TAG, exception.message ?: "Unknown error occurred")
           }
        }
    }

    private fun disconnectUser(request: DisconnectUserRequest) {
        lifecycleScope.launch {
           try {
               room?.awaitDisconnectUser(request)
           } catch (exception: ChatException) {
               Log.e(TAG, "Disconnect user rejected: ${exception.message}")
           } catch (exception: Exception) {
               Log.e(TAG, exception.message ?: "Unknown error occurred")
           }
        }
```

```
    }
}
```

Manage Chat-Message Requests

We need a way to manage our chat-message requests through all their possible states:

- Pending — A message was sent to a chat room but is not yet confirmed or rejected.
- Confirmed — A message was sent by the chat room to all users (including us).
- Rejected — A message was rejected by the chat room with an error object.

We will keep unresolved chat requests and chat messages in a list. The list merits a separate class, which we call `ChatEntries.kt`:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/ChatEntries.kt

package com.chatterbox.myapp

import com.amazonaws.ivs.chat.messaging.entities.ChatMessage
import com.amazonaws.ivs.chat.messaging.requests.SendMessageRequest

sealed class ChatEntry() {
    class Message(val message: ChatMessage) : ChatEntry()
    class PendingRequest(val request: SendMessageRequest) : ChatEntry()
    class FailedRequest(val request: SendMessageRequest) : ChatEntry()
}

class ChatEntries {
    /* This list is kept in sorted order. ChatMessages are sorted by date, while
 pending and failed requests are kept in their original insertion point. */
    val entries = mutableListOf<ChatEntry>()
    var adapter: ChatListAdapter? = null

    val size get() = entries.size

    /**
     * Insert pending request at the end.
     */
    fun addPendingRequest(request: SendMessageRequest) {
```

```
        val insertIndex = entries.size
        entries.add(insertIndex, ChatEntry.PendingRequest(request))
        adapter?.notifyItemInserted(insertIndex)
    }

    /**
     * Insert received message at proper place based on sendTime. This can cause
removal of pending requests.
     */
    fun addReceivedMessage(message: ChatMessage) {
        /* Skip if we have already handled that message. */
        val existingIndex = entries.indexOfLast { it is ChatEntry.Message &&
it.message.id == message.id }
        if (existingIndex != -1) {
            return
        }

        val removeIndex = entries.indexOfLast {
            it is ChatEntry.PendingRequest && it.request.requestId == message.requestId
        }
        if (removeIndex != -1) {
            entries.removeAt(removeIndex)
        }

        val insertIndexRaw = entries.indexOfFirst { it is ChatEntry.Message &&
it.message.sendTime > message.sendTime }
        val insertIndex = if (insertIndexRaw == -1) entries.size else insertIndexRaw
        entries.add(insertIndex, ChatEntry.Message(message))

        if (removeIndex == -1) {
            adapter?.notifyItemInserted(insertIndex)
        } else if (removeIndex == insertIndex) {
            adapter?.notifyItemChanged(insertIndex)
        } else {
            adapter?.notifyItemRemoved(removeIndex)
            adapter?.notifyItemInserted(insertIndex)
        }
    }

    fun addFailedRequest(request: SendMessageRequest) {
        val removeIndex = entries.indexOfLast {
            it is ChatEntry.PendingRequest && it.request.requestId == request.requestId
        }
        if (removeIndex != -1) {
```

```
            entries.removeAt(removeIndex)
            entries.add(removeIndex, ChatEntry.FailedRequest(request))
            adapter?.notifyItemChanged(removeIndex)
        } else {
            val insertIndex = entries.size
            entries.add(insertIndex, ChatEntry.FailedRequest(request))
            adapter?.notifyItemInserted(insertIndex)
        }
    }

    fun removeMessage(messageId: String) {
        val removeIndex = entries.indexOfFirst { it is ChatEntry.Message &&
 it.message.id == messageId }
        entries.removeAt(removeIndex)
        adapter?.notifyItemRemoved(removeIndex)
    }

    fun removeFailedRequest(requestId: String) {
        val removeIndex = entries.indexOfFirst { it is ChatEntry.FailedRequest &&
 it.request.requestId == requestId }
        entries.removeAt(removeIndex)
        adapter?.notifyItemRemoved(removeIndex)
    }

    fun removeAll() {
        entries.clear()
    }
}
```

To connect our list with the UI, we use an Adapter. For more information, see Binding to Data with AdapterView and Generated binding classes.

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/ChatListAdapter.kt

package com.chatterbox.myapp

import android.content.Context
import android.graphics.Color
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
```

```
import android.widget.LinearLayout
import android.widget.TextView
import androidx.core.content.ContextCompat
import androidx.core.view.isGone
import androidx.recyclerview.widget.RecyclerView
import com.amazonaws.ivs.chat.messaging.requests.DisconnectUserRequest
import java.text.DateFormat


class ChatListAdapter(
    private val entries: ChatEntries,
    private val onDisconnectUser: (request: DisconnectUserRequest) -> Unit,
) :
    RecyclerView.Adapter<ChatListAdapter.ViewHolder>() {
    var context: Context? = null
    var userId: String? = null

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val container: LinearLayout = view.findViewById(R.id.layout_container)
        val textView: TextView = view.findViewById(R.id.card_message_me_text_view)
        val failedMark: TextView = view.findViewById(R.id.failed_mark)
        val userNameText: TextView? = view.findViewById(R.id.username_edit_text)
        val dateText: TextView? = view.findViewById(R.id.dateText)
    }

    override fun onCreateViewHolder(viewGroup: ViewGroup, viewType: Int): ViewHolder {
        if (viewType == 0) {
            val rightView =
 LayoutInflater.from(viewGroup.context).inflate(R.layout.card_view_right, viewGroup,
 false)
            return ViewHolder(rightView)
        }
        val leftView =
 LayoutInflater.from(viewGroup.context).inflate(R.layout.card_view_left, viewGroup,
 false)
        return ViewHolder(leftView)
    }

    override fun getItemViewType(position: Int): Int {
        // Int 0 indicates to my message while Int 1 to other message
        val chatMessage = entries.entries[position]
        return if (chatMessage is ChatEntry.Message &&
 chatMessage.message.sender.userId != userId) 1 else 0
    }
```

```
   override fun onBindViewHolder(viewHolder: ViewHolder, position: Int) {
       return when (val entry = entries.entries[position]) {
           is ChatEntry.Message -> {
               viewHolder.textView.text = entry.message.content

               val bgColor = if (entry.message.sender.userId == userId) {
                   R.color.purple_500
               } else {
                   R.color.light_gray_2
               }

viewHolder.container.setBackgroundColor(ContextCompat.getColor(context!!, bgColor))

               if (entry.message.sender.userId != userId) {
                   viewHolder.textView.setTextColor(Color.parseColor("#000000"))
               }

               viewHolder.failedMark.isGone = true

               viewHolder.itemView.setOnCreateContextMenuListener { menu, _, _ ->
                   menu.add("Kick out").setOnMenuItemClickListener {
                       val request =
DisconnectUserRequest(entry.message.sender.userId, "Some reason")
                       onDisconnectUser(request)
                       true
                   }
               }

               viewHolder.userNameText?.text = entry.message.sender.userId
               viewHolder.dateText?.text =

DateFormat.getTimeInstance(DateFormat.SHORT).format(entry.message.sendTime)
           }

           is ChatEntry.PendingRequest -> {

viewHolder.container.setBackgroundColor(ContextCompat.getColor(context!!,
R.color.light_gray))
               viewHolder.textView.text = entry.request.content
               viewHolder.failedMark.isGone = true
               viewHolder.itemView.setOnCreateContextMenuListener(null)
               viewHolder.dateText?.text = "Sending"
           }
```

```
            is ChatEntry.FailedRequest -> {
                viewHolder.textView.text = entry.request.content

 viewHolder.container.setBackgroundColor(ContextCompat.getColor(context!!,
 R.color.dark_red))
                viewHolder.failedMark.isGone = false
                viewHolder.dateText?.text = "Failed"
            }
        }
    }

    override fun onAttachedToRecyclerView(recyclerView: RecyclerView) {
        super.onAttachedToRecyclerView(recyclerView)
        context = recyclerView.context
    }

    override fun getItemCount() = entries.entries.size
}
```

Final Steps

It is time to hook up our new adapter, binding `ChatEntries` class to `MainActivity`:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

import com.chatterbox.myapp.databinding.ActivityMainBinding
import com.chatterbox.myapp.ChatListAdapter
import com.chatterbox.myapp.ChatEntries

class MainActivity : AppCompatActivity() {
    // ...
    private var entries = ChatEntries()
    private lateinit var adapter: ChatListAdapter

    // ...

    private fun setUpChatView() {
```

```
        adapter = ChatListAdapter(entries, ::disconnectUser)
        entries.adapter = adapter

        val recyclerViewLayoutManager = LinearLayoutManager(this@MainActivity,
 LinearLayoutManager.VERTICAL, false)
        binding.recyclerView.layoutManager = recyclerViewLayoutManager
        binding.recyclerView.adapter = adapter

        binding.sendButton.setOnClickListener(::sendButtonClick)
        binding.messageEditText.setOnEditorActionListener { _, _, event ->
            val isEnterDown = (event.action == KeyEvent.ACTION_DOWN) && (event.keyCode
 == KeyEvent.KEYCODE_ENTER)
            if (!isEnterDown) {
                return@setOnEditorActionListener false
            }

            sendButtonClick(binding.sendButton)
            return@setOnEditorActionListener true
        }
    }
}
```

As we already have a class responsible for keeping track of our chat requests (`ChatEntries`), we are ready to implement code for manipulating `entries` in roomListener. We will update `entries` and `connectionState` accordingly to the event we are responding to:

Kotlin:

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt

package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
// ...


    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Create room instance
```

```
        room = ChatRoom(REGION, ::fetchChatToken).apply {
            lifecycleScope.launch {
                stateChanges().collect { state ->
                    Log.d(TAG, "state change to $state")
                    updateConnectionState(state)
                    if (state == ChatRoom.State.DISCONNECTED) {
                       entries.removeAll()
                    }
                }
            }

            lifecycleScope.launch {
                receivedMessages().collect { message ->
                    Log.d(TAG, "messageReceived $message")
                    entries.addReceivedMessage(message)
                }
            }

            lifecycleScope.launch {
                receivedEvents().collect { event ->
                    Log.d(TAG, "eventReceived $event")
                }
            }

            lifecycleScope.launch {
                deletedMessages().collect { event ->
                    Log.d(TAG, "messageDeleted $event")
                    entries.removeMessage(event.messageId)
                }
            }

            lifecycleScope.launch {
                disconnectedUsers().collect { event ->
                    Log.d(TAG, "userDisconnected $event")
                }
            }
        }

        binding.sendButton.setOnClickListener(::sendButtonClick)
        binding.connectButton.setOnClickListener {connect()}

        setUpChatView()

        updateConnectionState(ChatRoom.State.DISCONNECTED)
```

```
    }

// ...

}
```

Now you should be able to run your application! (See Build and run your app.) Remember to have your backend server running when using the app. You can spin it up from the terminal at the root of our project with this command: `./gradlew :auth-server:run` or by executing the `auth-server:run` Gradle task directly from Android Studio.

IVS Chat Client Messaging SDK: iOS Guide

The Amazon Interactive Video (IVS) Chat Client Messaging iOS SDK provides interfaces that allow you to incorporate our IVS Chat Messaging API on platforms using Apple's Swift programming language.

Latest version of IVS Chat Client Messaging iOS SDK: 1.0.0 (Release Notes)

Reference documentation and tutorials: For information on the most important methods available in the Amazon IVS Chat Client Messaging iOS SDK, see the reference documentation at: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/. This repository also contains various articles and tutorials.

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

Platform requirements: iOS 13.0 or greater is required for development.

Getting Started with the IVS Chat Client Messaging iOS SDK

We recommend that you integrate the SDK via Swift Package Manager. Alternatively, you can use CocoaPods or integrate the framework manually.

After integrating the SDK, you can import the SDK by adding the following code at the top of your relevant Swift file:

```
import AmazonIVSChatMessaging
```

Swift Package Manager

To use the `AmazonIVSChatMessaging` library in a Swift Package Manager project, add it to the dependencies for your package and the dependencies for your relevant targets:

1. Download the latest `.xcframework` from https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. In your Terminal, run:

```
shasum -a 256 path/to/downloaded/AmazonIVSChatMessaging.xcframework.zip
```

3. Take the output of the previous step and paste it into the checksum property of `.binaryTarget` as shown below within your project's `Package.swift` file:

```
let package = Package(
   // name, platforms, products, etc.
   dependencies: [
      // other dependencies
   ],
   targets: [
      .target(
         name: "<target-name>",
         dependencies: [
            // If you want to only bring in the SDK
            .binaryTarget(
               name: "AmazonIVSChatMessaging",
               url: "https://ivschat.live-video.net/1.0.0/
AmazonIVSChatMessaging.xcframework.zip",
               checksum: "<SHA-extracted-using-steps-detailed-above>"
            ),
            // your other dependencies
         ],
      ),
      // other targets
   ]
)
```

CocoaPods

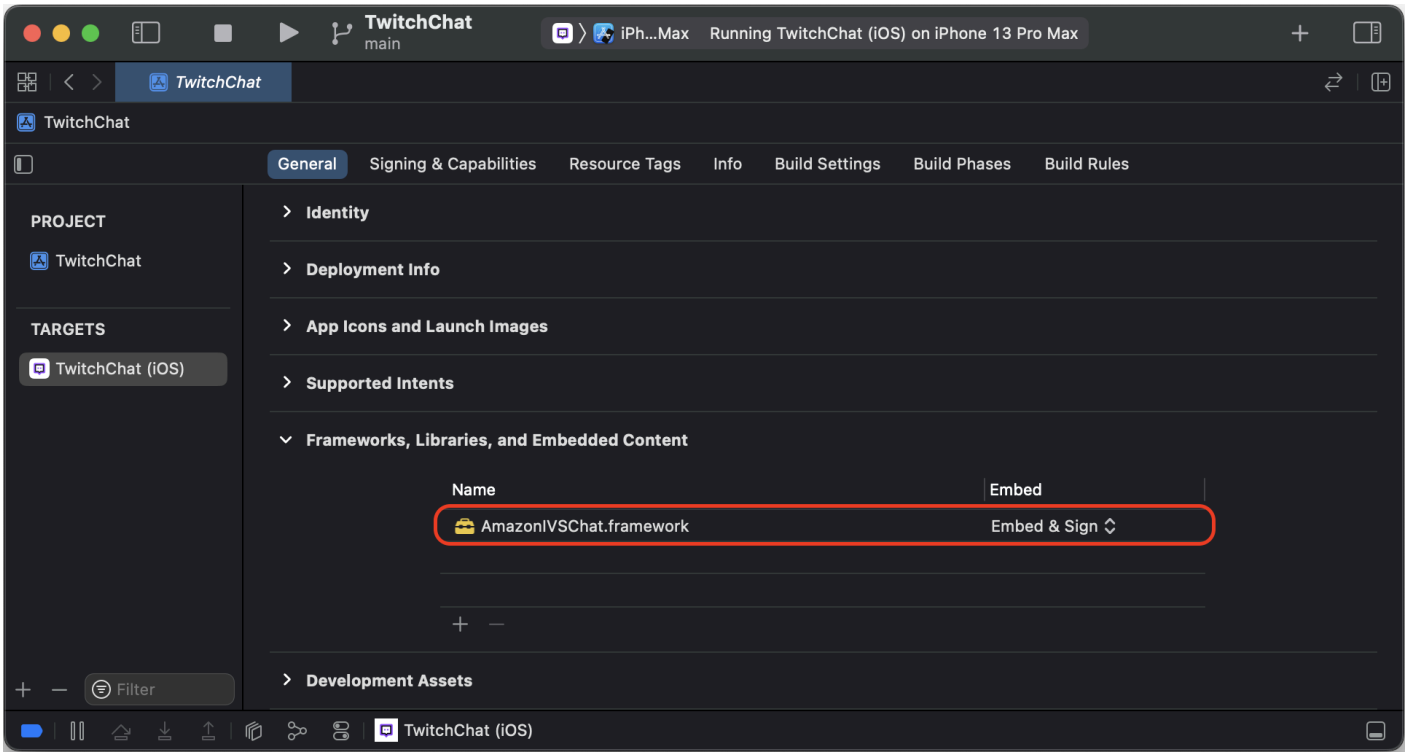
Releases are published via CocoaPods under the name `AmazonIVSChatMessaging`. Add this dependency to your Podfile:

```
pod 'AmazonIVSChat'
```

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

Manual Installation

1. Download the latest version from https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. Extract the contents of the archive. `AmazonIVSChatMessaging.xcframework` contains the SDK for both device and simulator.
3. Embed the extracted `AmazonIVSChatMessaging.xcframework` by dragging it into the **Frameworks, Libraries, and Embedded Content** section of the **General** tab for your application target:



Using the IVS Chat Client Messaging iOS SDK

This document takes you through the steps involved in using the Amazon IVS chat client messaging iOS SDK.

Connect to a Chat Room

Before starting, you should be familiar with Getting Started with Amazon IVS Chat. Also see the example apps for Web, Android, and iOS.

To connect to a chat room, your app needs some way of retrieving a chat token provided by your backend. Your application probably will retrieve a chat token using a network request to your backend.

To communicate this fetched chat token with the SDK, the SDK's `ChatRoom` model requires you to provide either an `async` function or an instance of an object conforming to the provided `ChatTokenProvider` protocol at the point of initialization. The value returned by either of these methods needs to be an instance of the SDK's `ChatToken` model.

Note: You populate instances of the `ChatToken` model using data retrieved from your backend. The fields required to initialize a `ChatToken` instance are the same as the fields in the CreateChatToken response. For more information on initializing instances of the `ChatToken` model, see Create an instance of ChatToken. Remember, *your backend* is responsible for providing the data in the `CreateChatToken` response to your app. How you decide to communicate with your backend to generate chat tokens is up to your app and its infrastructure.

After choosing your strategy to provide a `ChatToken` to the SDK, call `.connect()` after successfully initializing a `ChatRoom` instance with your token provider and the *AWS region* that your backend used to create the chat room you are trying to connect to. Note that `.connect()` is a throwing async function:

```
import AmazonIVSChatMessaging

let room = ChatRoom(
   awsRegion: <region-your-backend-created-the-chat-room-in>,
   tokenProvider: <your-chosen-token-provider-strategy>
)
try await room.connect()
```

Conforming to the ChatTokenProvider Protocol

For the `tokenProvider` parameter in the initializer for `ChatRoom`, you can provide an instance of `ChatTokenProvider`. Here is an example of an object conforming to `ChatTokenProvider`:

```
import AmazonIVSChatMessaging
```

```
// This object should exist somewhere in your app
class ChatService: ChatTokenProvider {
   func getChatToken() async throws -> ChatToken {
      let request = YourApp.getTokenURLRequest
      let data = try await URLSession.shared.data(for: request).0
      ...
      return ChatToken(
         token: String(data: data, using: .utf8)!,
         tokenExpirationTime: ..., // this is optional
         sessionExpirationTime: ... // this is optional
      )
   }
}
```

You can then take an instance of this conforming object and pass it into the initializer for `ChatRoom`:

```
// This should be the same AWS Region that you used to create
// your Chat Room in the Control Plane
let awsRegion = "us-west-2"
let service = ChatService()
let room = ChatRoom(
   awsRegion: awsRegion,
   tokenProvider: service
)
try await room.connect()
```

Providing an async Function in Swift

Suppose you already have a manager that you use to manage your application's network requests. It might look like this:

```
import AmazonIVSChatMessaging

class EndpointManager {
   func getAccounts() async -> AppUser {...}
   func signIn(user: AppUser) async {...}
   ...
}
```

You could just add another function in your manager to retrieve a `ChatToken` from your backend:

```
import AmazonIVSChatMessaging

class EndpointManager {
   ...
   func retrieveChatToken() async -> ChatToken {...}
}
```

Then, use the reference to that function in Swift when initializing a `ChatRoom`:

```
import AmazonIVSChatMessaging

let endpointManager: EndpointManager
let room = ChatRoom(
   awsRegion: endpointManager.awsRegion,
   tokenProvider: endpointManager.retrieveChatToken
)
try await room.connect()
```

Create an Instance of ChatToken

You can easily create an instance of `ChatToken` using the initializer provided in the SDK. See the documentation in `Token.swift` to learn more about the properties on `ChatToken`.

```
import AmazonIVSChatMessaging

let chatToken = ChatToken(
   token: <token-string-retrieved-from-your-backend>,
   tokenExpirationTime: nil, // this is optional
   sessionExpirationTime: nil // this is optional
)
```

Using Decodable

If, while interfacing with the IVS Chat API, your backend decides to simply forward the CreateChatToken response to your frontend application, you can take advantage of `ChatToken` 's conformance to Swift's `Decodable` protocol. However, there is a catch.

The `CreateChatToken` response payload uses strings for dates that are formatted using the ISO 8601 standard for internet timestamps. Normally in Swift, you would provide `JSONDecoder.DateDecodingStrategy.iso8601` as a value to `JSONDecoder`'s `.dateDecodingStrategy` property. However, `CreateChatToken`

uses high-precision fractional seconds in its strings, and this is not supported by `JSONDecoder.DateDecodingStrategy.iso8601`.

For your convenience, the SDK provides a public extension on `JSONDecoder.DateDecodingStrategy` with an additional `.preciseISO8601` strategy that allows you to successfully use `JSONDecoder` when decoding a instance of `ChatToken`:

```
import AmazonIVSChatMessaging

// The CreateChatToken data forwarded by your backend
let responseData: Data

let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .preciseISO8601
let token = try decoder.decode(ChatToken.self, from: responseData)
```

Disconnect from a Chat Room

To manually disconnect from a `ChatRoom` instance to which you successfully connected, call `room.disconnect()`. By default, chat rooms automatically call this function when they are deallocated.

```
import AmazonIVSChatMessaging

let room = ChatRoom(...)
try await room.connect()

// Disconnect
room.disconnect()
```

Receive a Chat Message/Event

To send and receive messages in your chat room, you need to provide an object that conforms to the `ChatRoomDelegate` protocol, after you successfully initialize an instance of `ChatRoom` and call `room.connect()`. Here is a typical example using `UIViewController`:

```
import AmazonIVSChatMessaging
import Foundation
import UIKit

class ViewController: UIViewController {
```

```
    let room: ChatRoom = ChatRoom(
        awsRegion: "us-west-2",
        tokenProvider: EndpointManager.shared
    )

    override func viewDidLoad() {
        super.viewDidLoad()
        Task { try await setUpChatRoom() }
    }

    private func setUpChatRoom() async throws {
        // Set the delegate to start getting notifications for room events
        room.delegate = self
        try await room.connect()
    }
}

extension ViewController: ChatRoomDelegate {
    func room(_ room: ChatRoom, didReceive message: ChatMessage) { ... }
    func room(_ room: ChatRoom, didReceive event: ChatEvent) { ... }
    func room(_ room: ChatRoom, didDelete message: DeletedMessageEvent) { ... }
}
```

Get Notified when the Connection Changes

As is to be expected, you cannot perform actions like sending a message in a room until the room is fully connected. The architecture of the SDK tries to encourage connecting to a ChatRoom on a background thread through async APIs. In case you want to build something in your UI that disables something like a send-message button, the SDK provides two strategies for getting notified when the connection state of a chat room changes, using `Combine` or `ChatRoomDelegate`. These are described below.

Important: A chat room's connection state also could change due to things like a dropped network connection. Take this into account when building your app.

Using Combine

Every instance of `ChatRoom` comes with its own `Combine` publisher in the form of the `state` property:

```
import AmazonIVSChatMessaging
import Combine
```

```
var cancellables: Set<AnyCancellable> = []

let room = ChatRoom(...)
room.state.sink { state in
   switch state {
   case .connecting:
      let image = UIImage(named: "antenna.radiowaves.left.and.right")
      sendMessageButton.setImage(image, for: .normal)
      sendMessageButton.isEnabled = false
   case .connected:
      let image = UIImage(named: "paperplane.fill")
      sendMessageButton.setImage(image, for: .normal)
      sendMessageButton.isEnabled = true
   case .disconnected:
      let image = UIImage(named: "antenna.radiowaves.left.and.right.slash")
      sendMessageButton.setImage(image, for: .normal)
      sendMessageButton.isEnabled = false
   }
}.assign(to: &cancellables)

// Connect to `ChatRoom` on a background thread
Task(priority: .background) {
   try await room.connect()
}
```

Using ChatRoomDelegate

Alternately, use the optional functions `roomDidConnect(_:)`, `roomIsConnecting(_:)`, and `roomDidDisconnect(_:)` within an object that conforms to `ChatRoomDelegate`. Here is an example using a `UIViewController`:

```
import AmazonIVSChatMessaging
import Foundation
import UIKit

class ViewController: UIViewController {
   let room: ChatRoom = ChatRoom(
      awsRegion: "us-west-2",
      tokenProvider: EndpointManager.shared
   )

   override func viewDidLoad() {
```

```
        super.viewDidLoad()
        Task { try await setUpChatRoom() }
    }

    private func setUpChatRoom() async throws {
        // Set the delegate to start getting notifications for room events
        room.delegate = self
        try await room.connect()
    }
}

extension ViewController: ChatRoomDelegate {
    func roomDidConnect(_ room: ChatRoom) {
        print("room is connected!")
    }
    func roomIsConnecting(_ room: ChatRoom) {
        print("room is currently connecting or fetching a token")
    }
    func roomDidDisconnect(_ room: ChatRoom) {
        print("room disconnected!")
    }
}
```

Perform Actions in a Chat Room

Different users have different capabilities for actions they can perform in a chat room; e.g., sending a message, deleting a message, or disconnecting a user. To perform one of these actions, call `perform(request:)` on a connected `ChatRoom`, passing in an instance of one of the provided `ChatRequest` objects in the SDK. The supported requests are in `Request.swift`.

Some actions performed in a chat room require connected users to have specific capabilities granted to them when your backend application calls `CreateChatToken`. By design, the SDK cannot discern the capabilities of a connected user. Hence, while you can try to perform moderator actions in a connected instance of `ChatRoom`, the control-plane API ultimately decides whether that action will succeed.

All actions that go through `room.perform(request:)` wait until the room receives the expected instance of a model (the type of which is associated with the request object itself) matched to the `requestId` of both the received model and the request object. If there is an issue with the request, `ChatRoom` always throws an error in the form of a `ChatError`. The definition of `ChatError` is in `Error.swift`.

Sending a Message

To send a chat message, use an instance of `SendMessageRequest`:

```
import AmazonIVSChatMessaging

let room = ChatRoom(...)
try await room.connect()
try await room.perform(
   request: SendMessageRequest(
      content: "Release the Kraken!"
   )
)
```

As mentioned above, `room.perform(request:)` returns once a corresponding `ChatMessage` is received by the `ChatRoom`. If there is an issue with the request (like exceeding the message character limit for a room), an instance of `ChatError` is thrown instead. You can then surface this useful information in your UI:

```
import AmazonIVSChatMessaging

do {
   let message = try await room.perform(
      request: SendMessageRequest(
         content: "Release the Kraken!"
      )
   )
   print(message.id)
} catch let error as ChatError {
   switch error.errorCode {
   case .invalidParameter:
      print("Exceeded the character limit!")
   case .tooManyRequests:
      print("Exceeded message request limit!")
   default:
      break
   }

   print(error.errorMessage)
}
```

Appending Metadata to a Message

When sending a message, you can append metadata that will be associated with it. `SendMessageRequest` has an `attributes` property, with which you can initialize your request. The data you attach there is attached to the message when others receive that message in the room.

Here is an example of attaching emote data to a message being sent:

```
import AmazonIVSChatMessaging

let room = ChatRoom(...)
try await room.connect()
try await room.perform(
   request: SendMessageRequest(
      content: "Release the Kraken!",
      attributes: [
         "messageReplyId" : "<other-message-id>",
         "attached-emotes" : "krakenCry,krakenPoggers,krakenCheer"
      ]
   )
)
```

Using `attributes` in a `SendMessageRequest` can be extremely useful for building complex features in your chat product. For example, one could build threading functionality using the `[String : String]` attributes dictionary in a `SendMessageRequest`!

The `attributes` payload is very flexible and powerful. Use it to derive information about your message you would not be able to do otherwise. Using attributes is much easier than, for instance, parsing the string of a message to get information about things like emotes.

Deleting a Message

Deleting a chat message is just like sending one. Use the `room.perform(request:)` function on `ChatRoom` to achieve this by creating an instance of `DeleteMessageRequest`.

To easily access previous instances of received Chat messages, pass in the value of `message.id` to the initializer of `DeleteMessageRequest`.

Optionally, provide a reason string to `DeleteMessageRequest` so you can surface that in your UI.

```
import AmazonIVSChatMessaging
```

```
let room = ChatRoom(...)
try await room.connect()
try await room.perform(
   request: DeleteMessageRequest(
      id: "<other-message-id-to-delete>",
      reason: "Abusive chat is not allowed!"
   )
)
```

As this is a moderator action, your user may not actually have the capability of deleting another user's message. You can use Swift's throwable function mechanic to surface an error message in your UI when a user tries to delete a message without the appropriate capability.

When your backend calls `CreateChatToken` for a user, it needs to pass `"DELETE_MESSAGE"` into the `capabilities` field to activate that functionality for a connected chat user.

Here is an example of catching a capability error thrown when attempting to delete a message without the appropriate permissions:

```
import AmazonIVSChatMessaging

do {
   // `deleteEvent` is the same type as the object that gets sent to
   // `ChatRoomDelegate`'s `room(_:didDeleteMessage:)` function
   let deleteEvent = try await room.perform(
      request: DeleteMessageRequest(
         id: "<other-message-id-to-delete>",
         reason: "Abusive chat is not allowed!"
      )
   )
   dataSource.messages[deleteEvent.messageID] = nil
   tableView.reloadData()
} catch let error as ChatError {
   switch error.errorCode {
   case .forbidden:
      print("You cannot delete another user's messages. You need to be a mod to do
 that!")
   default:
      break
   }

   print(error.errorMessage)
```

```
}
```

Disconnecting Another User

Use `room.perform(request:)` to disconnect another user from a chat room. Specifically, use an instance of `DisconnectUserRequest`. All `ChatMessages` received by a `ChatRoom` have a `sender` property, which contains the user ID that you need to properly initialize with an instance of `DisconnectUserRequest`. Optionally, provide a reason string for the disconnect request.

```
import AmazonIVSChatMessaging

let room = ChatRoom(...)
try await room.connect()

let message: ChatMessage = dataSource.messages["<message-id>"]
let sender: ChatUser = message.sender
let userID: String = sender.userId
let reason: String = "You've been disconnected due to abusive behavior"

try await room.perform(
   request: DisconnectUserRequest(
      id: userID,
      reason: reason
   )
)
```

As this is another example of a moderator action, you may try to disconnect another user, but you will be unable to do so unless you have the `DISCONNECT_USER` capability. The capability gets set when your backend application calls `CreateChatToken` and injects the `"DISCONNECT_USER"` string into the `capabilities` field.

If your user does not have the capability to disconnect another user, `room.perform(request:)` throws an instance of `ChatError`, just like the other requests. You can inspect the error's `errorCode` property to determine if the request failed because of the lack of moderator privileges:

```
import AmazonIVSChatMessaging

do {
   let message: ChatMessage = dataSource.messages["<message-id>"]
   let sender: ChatUser = message.sender
```

```
    let userID: String = sender.userId
    let reason: String = "You've been disconnected due to abusive behavior"

    try await room.perform(
        request: DisconnectUserRequest(
            id: userID,
            reason: reason
        )
    )
} catch let error as ChatError {
    switch error.errorCode {
    case .forbidden:
        print("You cannot disconnect another user. You need to be a mod to do that!")
    default:
        break
    }

    print(error.errorMessage)
}
```

IVS Chat Client Messaging SDK: iOS Tutorial

The Amazon Interactive Video (IVS) Chat Client Messaging iOS SDK provides interfaces to allow you to incorporate our IVS Chat Messaging API on platforms using Apple's Swift programming language.

For a Chat iOS SDK tutorial, see https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/latest/tutorials/table-of-contents/.

IVS Chat Client Messaging SDK: JavaScript Guide

The Amazon Interactive Video (IVS) Chat Client Messaging JavaScript SDK allows you to incorporate our Amazon IVS Chat Messaging API on platforms using a Web browser.

Latest version of IVS Chat Client Messaging JavaScript SDK: 1.0.2 (Release Notes)

Reference documentation: For information on the most important methods available in the Amazon IVS Chat Client Messaging JavaScript SDK, see the reference documentation at: https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/

Sample code: See the sample repository on GitHub, for a Web-specific demo using the JavaScript SDK: https://github.com/aws-samples/amazon-ivs-chat-web-demo

Getting Started with the IVS Chat Client Messaging JavaScript SDK

Before starting, you should be familiar with Getting Started with Amazon IVS Chat.

Add the Package

Use either:

```
$ npm install --save amazon-ivs-chat-messaging
```

or:

```
$ yarn add amazon-ivs-chat-messaging
```

React Native Support

The IVS Chat Client Messaging JavaScript SDK has a `uuid` dependency which uses the `crypto.getRandomValues` method. Since this method is not supported in React Native, you need to install the additional polyfill `react-native-get-random-value` and import it at the top of the `index.js` file:

```
import 'react-native-get-random-values';
import {AppRegistry} from 'react-native';
import App from './src/App';
import {name as appName} from './app.json';

AppRegistry.registerComponent(appName, () => App);
```

Set Up Your Backend

This integration requires endpoints on your server that talk to the Amazon IVS Chat API. Use the official AWS libraries for access to the Amazon IVS API from your server. These libraries are accessible within several languages from the public packages; e.g., node.js, java, and go.

Create a server endpoint that talks to the Amazon IVS Chat API CreateChatToken operation, to create a chat token for chat users.

Using the IVS Chat Client Messaging JavaScript SDK

This document takes you through the steps involved in using the Amazon IVS chat client messaging JavaScript SDK.

Initialize a Chat Room Instance

Create an instance of the `ChatRoom` class. This requires passing `regionOrUrl` (the AWS region where your chat room is hosted) and `tokenProvider` (the token-fetching method will be created in the next step):

```
const room = new ChatRoom({
  regionOrUrl: 'us-west-2',
  tokenProvider: tokenProvider,
});
```

Token Provider Function

Create an asynchronous token-provider function that fetches a chat token from your backend:

```
type ChatTokenProvider = () => Promise<ChatToken>;
```

The function should accept no parameters and return a Promise containing a chat token object:

```
type ChatToken = {
  token: string;
  sessionExpirationTime?: Date;
  tokenExpirationTime?: Date;
}
```

This function is needed to initialize the ChatRoom object. Below, fill in the `<token>` and `<date-time>` fields with values received from your backend:

```
// You will need to fetch a fresh token each time this method is called by
// the IVS Chat Messaging SDK, since each token is only accepted once.
function tokenProvider(): Promise<ChatToken> {
  // Call your backend to fetch chat token from IVS Chat endpoint:
  // e.g. const token = await appBackend.getChatToken()
  return {
    token: "<token>",
```

```
    sessionExpirationTime: new Date("<date-time>"),
    tokenExpirationTime: new Date("<date-time>")
  }
}
```

Remember to pass the `tokenProvider` to the ChatRoom constructor. ChatRoom refreshes the token when the connection is interrupted or session expires. Do not use the `tokenProvider` to store a token anywhere; the ChatRoom handles it for you.

Receive Events

Next, subscribe to chat-room events to receive lifecycle events, as well as messages and events delivered in the chat room:

```
/**
* Called when room is establishing the initial connection or reestablishing
* connection after socket failure/token expiration/etc
*/
const unsubscribeOnConnecting = room.addListener('connecting', () => { });

/** Called when connection has been established. */
const unsubscribeOnConnected = room.addListener('connect', () => { });

/** Called when a room has been disconnected. */
const unsubscribeOnDisconnected = room.addListener('disconnect', () => { });

/** Called when a chat message has been received. */
const unsubscribeOnMessageReceived = room.addListener('message', (message) => {
 /* Example message:
  * {
  *   id: "5OPsDdX18qcJ",
  *   sender: { userId: "user1" },
  *   content: "hello world",
  *   sendTime: new Date("2022-10-11T12:46:41.723Z"),
  *   requestId: "d1b511d8-d5ed-4346-b43f-49197c6e61de"
  * }
  */
});

/** Called when a chat event has been received. */
const unsubscribeOnEventReceived = room.addListener('event', (event) => {
 /* Example event:
  * {
```

```
  *   id: "5OPsDdX18qcJ",
  *   eventName: "customEvent,
  *   sendTime: new Date("2022-10-11T12:46:41.723Z"),
  *   requestId: "d1b511d8-d5ed-4346-b43f-49197c6e61de",
  *   attributes: { "Custom Attribute": "Custom Attribute Value" }
  * }
  */
});

/** Called when `aws:DELETE_MESSAGE` system event has been received. */
const unsubscribeOnMessageDelete = room.addListener('messageDelete',
 (deleteMessageEvent) => {
 /* Example delete message event:
  * {
  *   id: "AYk6xKitV4On",
  *   messageId: "R1BLTDN84zEO",
  *   reason: "Spam",
  *   sendTime: new Date("2022-10-11T12:56:41.113Z"),
  *   requestId: "b379050a-2324-497b-9604-575cb5a9c5cd",
  *   attributes: { MessageID: "R1BLTDN84zEO", Reason: "Spam" }
  * }
  */
});

/** Called when `aws:DISCONNECT_USER` system event has been received. */
const unsubscribeOnUserDisconnect = room.addListener('userDisconnect',
 (disconnectUserEvent) => {
 /* Example event payload:
  * {
  *   id: "AYk6xKitV4On",
  *   userId": "R1BLTDN84zEO",
  *   reason": "Spam",
  *   sendTime": new Date("2022-10-11T12:56:41.113Z"),
  *   requestId": "b379050a-2324-497b-9604-575cb5a9c5cd",
  *   attributes": { UserId: "R1BLTDN84zEO", Reason: "Spam" }
  * }
  */
});
```

Connect to the Chat Room

The last step of basic initialization is connecting to the chat room by establishing a WebSocket connection. To do this, simply call the `connect()` method within the room instance:

```
room.connect();
```

The SDK will try to establish a connection to the chat room encoded in the chat token received from your server.

After you call `connect()`, the room will transition to the `connecting` state and emit a `connecting` event. When the room successfully connects, it transitions to the `connected` state and emits a `connect` event.

A connection failure might happen due to issues when fetching the token or when connecting to WebSocket. In this case, the room tries to reconnect automatically up to the number of times indicated by the `maxReconnectAttempts` constructor parameter. During the reconnection attempts, the room is in the `connecting` state and does not emit additional events. After exhausting the reconnect attempts, the room transitions to the `disconnected` state and emits a `disconnect` event (with a relevant disconnect reason). In the `disconnected` state, the room no longer tries to connect; you must call `connect()` again to trigger the connection process.

Perform Actions in a Chat Room

The Amazon IVS Chat Messaging SDK provides user actions for sending messages, deleting messages, and disconnecting other users. These are available on the `ChatRoom` instance. They return a `Promise` object which allows you to receive request confirmation or rejection.

Sending a Message

For this request, you must have a `SEND_MESSAGE` capacity encoded in your chat token.

To trigger a send-message request:

```
const request = new SendMessageRequest('Test Echo');
room.sendMessage(request);
```

To get a confirmation or rejection of the request, `await` the returned promise or use the `then()` method:

```
try {
  const message = await room.sendMessage(request);
  // Message was successfully sent to chat room
} catch (error) {
```

```
  // Message request was rejected. Inspect the `error` parameter for details.
}
```

Deleting a Message

For this request, you must have a `DELETE_MESSAGE` capacity encoded in your chat token.

To delete a message for moderation purposes, call the `deleteMessage()` method:

```
const request = new DeleteMessageRequest(messageId, 'Reason for deletion');
room.deleteMessage(request);
```

To get a confirmation or rejection of the request, `await` the returned promise or use the `then()` method:

```
try {
  const deleteMessageEvent = await room.deleteMessage(request);
  // Message was successfully deleted from chat room
} catch (error) {
  // Delete message request was rejected. Inspect the `error` parameter for details.
}
```

Disconnecting Another User

For this request, you must have a `DISCONNECT_USER` capacity encoded in your chat token.

To disconnect another user for moderation purposes, call the `disconnectUser()` method:

```
const request = new DisconnectUserRequest(userId, 'Reason for disconnecting user');
room.disconnectUser(request);
```

To get a confirmation or rejection of the request, `await` the returned promise or use the `then()` method:

```
try {
  const disconnectUserEvent = await room.disconnectUser(request);
  // User was successfully disconnected from the chat room
} catch (error) {
  // Disconnect user request was rejected. Inspect the `error` parameter for details.
}
```

Disconnect from a Chat Room

To close your connection to the chat room, call the `disconnect()` method on the `room` instance:

```
room.disconnect();
```

Calling this method causes the room to close the underlying WebSocket in an orderly manner. The room instance transitions to a `disconnected` state and emits a disconnect event, with the `disconnect` reason set to `"clientDisconnect"`.

IVS Chat Client Messaging SDK: JavaScript Tutorial Part 1: Chat Rooms

This is the first of a two-part tutorial. You will learn the essentials of working with the Amazon IVS Chat Client Messaging JavaScript SDK by building a fully functional app using JavaScript/TypeScript. We call the app *Chatterbox*.

The intended audience is experienced developers who are new to the Amazon IVS Chat Messaging SDK. You should be comfortable with the JavaScript/TypeScript programming language and React library.

For brevity, we'll refer to the Amazon IVS Chat Client Messaging JavaScript SDK as the Chat JS SDK.

Note: In some cases, code examples for JavaScript and TypeScript are identical, so they are combined.

This first part of the tutorial is broken up into several sections:

1. the section called "Set Up a Local Authentication/Authorization Server"
2. the section called "Create a Chatterbox Project"
3. the section called "Connect to a Chat Room"
4. the section called "Build a Token Provider"
5. the section called "Observe Connection Updates"
6. the section called "Create a Send Button Component"
7. the section called "Create a Message Input"
8. the section called "Next Steps"

For full SDK documentation, start with Amazon IVS Chat Client Messaging SDK (here in the *Amazon IVS Chat User Guide*) and the Chat Client Messaging: SDK for JavaScript Reference (on GitHub).

Prerequisites

- Be familiar with JavaScript/TypeScript and the React library. If you're unfamiliar with React, learn the basics in this Tic-Tac-Toe Tutorial.
- Read and understand *Getting Started with IVS Chat*.
- Create an AWS IAM user with the CreateChatToken and CreateRoom capabilities defined in an existing IAM policy. (See *Getting Started with IVS Chat*.)
- Ensure that the secret/access keys for this user are stored in an AWS credentials file. For instructions, see the AWS CLI User Guide (especially Configuration and credential file settings).
- Create a chat room and save its ARN. See *Getting Started with IVS Chat*. (If you don't save the ARN, you can look it up later with the console or Chat API.)
- Install the Node.js 14+ environment with the NPM or Yarn package manager.

Set Up a Local Authentication/Authorization Server

Your backend application is responsible for both creating chat rooms and generating the chat tokens that are needed for the Chat JS SDK to authenticate and authorize your clients for your chat rooms. You must use your own backend since you cannot securely store AWS keys in a mobile app; sophisticated attackers could extract these and gain access to your AWS account.

See Create a Chat Token in *Getting Started with Amazon IVS Chat*. As shown in the flowchart there, your server-side application is responsible for creating a chat token. This means your app must provide its own means of generating a chat token by requesting one from your server-side application.

In this section, you will learn the basics of creating a token provider in your backend. We use the express framework to create a live local server that manages the creation of chat tokens using your local AWS environment.

Create an empty `npm` project using NPM. Create a directory to hold your application, and make that your working directory:

```
$ mkdir backend & cd backend
```

Use `npm init` to create a `package.json` file for your application:

```
$ npm init
```

This command prompts you for several things, including the name and version of your application. For now, just press **RETURN** to accept the defaults for most of them, with the following exception:

```
entry point: (index.js)
```

Press **RETURN** to accept the suggested default filename of `index.js` or enter whatever you want the name of the main file to be.

Now install required dependencies:

```
$ npm install express aws-sdk cors dotenv
```

`aws-sdk` requires configuration-environment variables, which automatically load from a file named `.env` located in the root directory. To configure it, create a new file named `.env` and fill in the missing configuration information:

```
# .env

# The region to send service requests to.
AWS_REGION=us-west-2

# Access keys use an access key ID and secret access key
# that you use to sign programmatic requests to AWS.

# AWS access key ID.
AWS_ACCESS_KEY_ID=...

# AWS secret access key.
AWS_SECRET_ACCESS_KEY=...
```

Now we create an entry-point file in the root directory with the name you entered above in the `npm init` command. In this case, we use `index.js`, and import all required packages:

```
// index.js
import express from 'express';
import AWS from 'aws-sdk';
```

```
import 'dotenv/config';
import cors from 'cors';
```

Now create a new instance of `express`:

```
const app = express();
const port = 3000;

app.use(express.json());
app.use(cors({ origin: ['http://127.0.0.1:5173'] }));
```

After that you can create your first endpoint POST method for the token provider. Take the required parameters from the request body (`roomId`, `userId`, `capabilities` and `sessionDurationInMinutes`):

```
app.post('/create_chat_token', (req, res) => {
  const { roomIdentifier, userId, capabilities, sessionDurationInMinutes } = req.body
 || {};
});
```

Add validation of required fields:

```
app.post('/create_chat_token', (req, res) => {
  const { roomIdentifier, userId, capabilities, sessionDurationInMinutes } = req.body
 || {};

  if (!roomIdentifier || !userId) {
    res.status(400).json({ error: 'Missing parameters: `roomIdentifier`, `userId`' });
    return;
  }
});
```

After preparing the POST method, we integrate `createChatToken` with `aws-sdk` for the core functionality of authentication/authorization:

```
app.post('/create_chat_token', (req, res) => {
  const { roomIdentifier, userId, capabilities, sessionDurationInMinutes } = req.body
 || {};

  if (!roomIdentifier || !userId || !capabilities) {
```

```
    res.status(400).json({ error: 'Missing parameters: `roomIdentifier`, `userId`,
 `capabilities`' });
    return;
  }

  ivsChat.createChatToken({ roomIdentifier, userId, capabilities,
 sessionDurationInMinutes }, (error, data) => {
    if (error) {
      console.log(error);
      res.status(500).send(error.code);
    } else if (data.token) {
      const { token, sessionExpirationTime, tokenExpirationTime } = data;
      console.log(`Retrieved Chat Token: ${JSON.stringify(data, null, 2)}`);

      res.json({ token, sessionExpirationTime, tokenExpirationTime });
    }
  });
});
```

At the end of the file, add a port listener for your `express` app:

```
app.listen(port, () => {
  console.log(`Backend listening on port ${port}`);
});
```

Now you can run the server with the following command from the project's root:

```
$ node index.js
```

Tip: This server accepts URL requests at https://localhost:3000.

Create a Chatterbox Project

First you create the React project called `chatterbox`. Run this command:

```
npx create-react-app chatterbox
```

You can integrate the Chat Client Messaging JS SDK via Node Package Manager or Yarn Package Manager:

- Npm: `npm install amazon-ivs-chat-messaging`

- Yarn: `yarn add amazon-ivs-chat-messaging`

Connect to a Chat Room

Here you create a `ChatRoom` and connect to it using asynchronous methods. The `ChatRoom` class manages your user's connection to the Chat JS SDK. To successfully connect to a chat room, you must provide an instance of `ChatToken` within your React application.

Navigate to the `App` file that's created in the default `chatterbox` project and delete everything between the two `<div>` tags. None of the pre-populated code is needed. At this point, our `App` is pretty empty.

```
// App.jsx / App.tsx

import * as React from 'react';

export default function App() {
  return <div>Hello!</div>;
}
```

Create a new `ChatRoom` instance and pass it to state using the `useState` hook. It requires passing `regionOrUrl` (the AWS region in which your chat room is hosted) and `tokenProvider` (used for the backend authentication/authorization flow that is created in subsequent steps).

Important: You must use the same AWS region as the one in which you created the room in Getting Started with Amazon IVS Chat . The API is an AWS regional service. For a list of supported regions and Amazon IVS Chat HTTPS service endpoints, see the Amazon IVS Chat regions page.

```
// App.jsx / App.tsx

import React, { useState } from 'react';
import { ChatRoom } from 'amazon-ivs-chat-messaging';

export default function App() {
  const [room] = useState(() =>
    new ChatRoom({
      regionOrUrl: process.env.REGION as string,
      tokenProvider: () => {},
    }),
  );
```

```
  return <div>Hello!</div>;
}
```

Build a Token Provider

As the next step, we need to build a parameterless `tokenProvider` function that is required by the `ChatRoom` constructor. First, we will create a `fetchChatToken` function that will make a POST request to the backend application that you set up in the section called "Set Up a Local Authentication/Authorization Server". Chat tokens contain the information needed for the SDK to successfully establish a chat-room connection. The Chat API uses these tokens as a secure way of validating a user's identity, capabilities within a chat room, and session duration.

In the Project navigator, create a new TypeScript/JavaScript file named `fetchChatToken`. Build a fetch request to the `backend` application and return the `ChatToken` object from the response. Add the request body properties needed for creating a chat token. Use the rules defined for Amazon Resource Names (ARNs). These properties are documented in the CreateChatToken operation.

Note: The URL you're using here is the same URL that your local server created when you ran the backend application.

TypeScript

```
// fetchChatToken.ts

import { ChatToken } from 'amazon-ivs-chat-messaging';

type UserCapability = 'DELETE_MESSAGE' | 'DISCONNECT_USER' | 'SEND_MESSAGE';

export async function fetchChatToken(
  userId: string,
  capabilities: UserCapability[] = [],
  attributes?: Record<string, string>,
  sessionDurationInMinutes?: number,
): Promise<ChatToken> {
  const response = await fetch(`${process.env.BACKEND_BASE_URL}/create_chat_token`,
 {
    method: 'POST',
    headers: {
      Accept: 'application/json',
```

```
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      userId,
      roomIdentifier: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
  });

  const token = await response.json();

  return {
    ...token,
    sessionExpirationTime: new Date(token.sessionExpirationTime),
    tokenExpirationTime: new Date(token.tokenExpirationTime),
  };
}
```

JavaScript

```
// fetchChatToken.js

export async function fetchChatToken(
  userId,
  capabilities = [],
  attributes,
  sessionDurationInMinutes) {
  const response = await fetch(`${process.env.BACKEND_BASE_URL}/create_chat_token`,
 {
    method: 'POST',
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      userId,
      roomIdentifier: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
```

```
  });

  const token = await response.json();

  return {
    ...token,
    sessionExpirationTime: new Date(token.sessionExpirationTime),
    tokenExpirationTime: new Date(token.tokenExpirationTime),
  };
}
```

Observe Connection Updates

Reacting to changes in a chat room's connection state are essential parts of making a chat app. Let's start with subscribing to relevant events:

```
// App.jsx / App.tsx

import React, { useState, useEffect } from 'react';
import { ChatRoom } from 'amazon-ivs-chat-messaging';
import { fetchChatToken } from './fetchChatToken';

export default function App() {
  const [room] = useState(
    () =>
      new ChatRoom({
        regionOrUrl: process.env.REGION as string,
        tokenProvider: () => fetchChatToken('Mike', ['SEND_MESSAGE']),
      }),
  );

  useEffect(() => {
    const unsubscribeOnConnecting = room.addListener('connecting', () => {});
    const unsubscribeOnConnected = room.addListener('connect', () => {});
    const unsubscribeOnDisconnected = room.addListener('disconnect', () => {});

    return () => {
      // Clean up subscriptions.
      unsubscribeOnConnecting();
      unsubscribeOnConnected();
      unsubscribeOnDisconnected();
    };
```

```
  }, [room]);

  return <div>Hello!</div>;
}
```

Next, we need to provide the ability to read the connection state. We use our `useState` hook to create some local state in App and set the connection state inside each listener.

TypeScript

```
// App.tsx

import React, { useState, useEffect } from 'react';
import { ChatRoom, ConnectionState } from 'amazon-ivs-chat-messaging';
import { fetchChatToken } from './fetchChatToken';

export default function App() {
  const [room] = useState(
    () =>
      new ChatRoom({
        regionOrUrl: process.env.REGION as string,
        tokenProvider: () => fetchChatToken('Mike', ['SEND_MESSAGE']),
      }),
  );
  const [connectionState, setConnectionState] =
 useState<ConnectionState>('disconnected');

  useEffect(() => {
    const unsubscribeOnConnecting = room.addListener('connecting', () => {
      setConnectionState('connecting');
    });

    const unsubscribeOnConnected = room.addListener('connect', () => {
      setConnectionState('connected');
    });

    const unsubscribeOnDisconnected = room.addListener('disconnect', () => {
      setConnectionState('disconnected');
    });

    return () => {
      unsubscribeOnConnecting();
      unsubscribeOnConnected();
```

```
      unsubscribeOnDisconnected();
    };
  }, [room]);

  return <div>Hello!</div>;
}
```

JavaScript

```
// App.jsx

import React, { useState, useEffect } from 'react';
import { ChatRoom } from 'amazon-ivs-chat-messaging';
import { fetchChatToken } from './fetchChatToken';

export default function App() {
  const [room] = useState(
    () =>
      new ChatRoom({
        regionOrUrl: process.env.REGION,
        tokenProvider: () => fetchChatToken('Mike', ['SEND_MESSAGE']),
      }),
  );
  const [connectionState, setConnectionState] = useState('disconnected');

  useEffect(() => {
    const unsubscribeOnConnecting = room.addListener('connecting', () => {
      setConnectionState('connecting');
    });

    const unsubscribeOnConnected = room.addListener('connect', () => {
      setConnectionState('connected');
    });

    const unsubscribeOnDisconnected = room.addListener('disconnect', () => {
      setConnectionState('disconnected');
    });

    return () => {
      unsubscribeOnConnecting();
      unsubscribeOnConnected();
      unsubscribeOnDisconnected();
    };
```

```
  }, [room]);

  return <div>Hello!</div>;
}
```

After subscribing to the connection state, display the connection state and connect to the chat room using the `room.connect` method inside the `useEffect` hook:

```
// App.jsx / App.tsx

// ...

useEffect(() => {
  const unsubscribeOnConnecting = room.addListener('connecting', () => {
    setConnectionState('connecting');
  });

  const unsubscribeOnConnected = room.addListener('connect', () => {
    setConnectionState('connected');
  });

  const unsubscribeOnDisconnected = room.addListener('disconnect', () => {
    setConnectionState('disconnected');
  });

  room.connect();

  return () => {
    unsubscribeOnConnecting();
    unsubscribeOnConnected();
    unsubscribeOnDisconnected();
  };
}, [room]);

// ...

return (
  <div>
    <h4>Connection State: {connectionState}</h4>
  </div>
);
```

```
// ...
```

You have successfully implemented a chat-room connection.

Create a Send Button Component

In this section you create a send button that has a different design for each connection state. The send button facilitates the sending of messages in a chat room. It also serves as a visual indicator of whether/when messages can be sent; e.g., in the face of dropped connections or expired chat sessions.

First, create a new file in the `src` directory of your Chatterbox project and name it `SendButton`. Then, create a component that will display a button for your chat application. Export your `SendButton` and import it to `App`. In the empty `<div></div>`, add `<SendButton />`.

TypeScript

```
// SendButton.tsx

import React from 'react';

interface Props {
  onPress?: () => void;
  disabled?: boolean;
}

export const SendButton = ({ onPress, disabled }: Props) => {
  return (
    <button disabled={disabled} onClick={onPress}>
      Send
    </button>
  );
};

// App.tsx

import { SendButton } from './SendButton';

// ...

return (
  <div>
```

```
    <div>Connection State: {connectionState}</div>
    <SendButton />
  </div>
);
```

JavaScript

```
// SendButton.jsx

import React from 'react';

export const SendButton = ({ onPress, disabled }) => {
  return (
    <button disabled={disabled} onClick={onPress}>
      Send
    </button>
  );
};

// App.jsx

import { SendButton } from './SendButton';

// ...

return (
  <div>
    <div>Connection State: {connectionState}</div>
    <SendButton />
  </div>
);
```

Next, in App define a function named onMessageSend and pass it to the SendButton onPress property. Define another variable named isSendDisabled (which prevents sending messages when the room is not connected) and pass it to the SendButton disabled property.

```
// App.jsx / App.tsx

// ...

const onMessageSend = () => {};
```

```
const isSendDisabled = connectionState !== 'connected';

return (
  <div>
    <div>Connection State: {connectionState}</div>
    <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
  </div>
);

// ...
```

Create a Message Input

The Chatterbox message bar is the component that you will interact with to send messages to a chat room. Typically it contains a text input for composing your message and a button to send your message.

To create a `MessageInput` component, first create a new file in the `src` directory and name it `MessageInput`. Then, create a controlled input component that will display an input for your chat application. Export your `MessageInput` and import it to `App` (above the `<SendButton />`).

Create a new state named `messageToSend` using the `useState` hook, with an empty string as its default value. In the body of your app, pass `messageToSend` to the `value` of `MessageInput` and pass the `setMessageToSend` to the on`MessageChange` property:

TypeScript

```
// MessageInput.tsx

import * as React from 'react';

interface Props {
  value?: string;
  onValueChange?: (value: string) => void;
}

export const MessageInput = ({ value, onValueChange }: Props) => {
  return (
    <input type="text" value={value} onChange={(e) => onValueChange?.
(e.target.value)} placeholder="Send a message" />
  );
};
```

```
// App.tsx

// ...

import { MessageInput } from './MessageInput';

// ...

export default function App() {
  const [messageToSend, setMessageToSend] = useState('');

// ...

return (
  <div>
    <h4>Connection State: {connectionState}</h4>
    <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
    <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
  </div>
);
```

JavaScript

```
// MessageInput.jsx

import * as React from 'react';

export const MessageInput = ({ value, onValueChange }) => {
  return (
    <input type="text" value={value} onChange={(e) => onValueChange?.
(e.target.value)} placeholder="Send a message" />
  );
};

// App.jsx

// ...

import { MessageInput } from './MessageInput';

// ...
```

```
export default function App() {
  const [messageToSend, setMessageToSend] = useState('');

// ...


return (
  <div>
    <h4>Connection State: {connectionState}</h4>
    <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
    <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
  </div>
);
```

Next Steps

Now that you finished building a message bar for Chatterbox, proceed to Part 2 of this JavaScript tutorial, Messages and Events.

IVS Chat Client Messaging SDK: JavaScript Tutorial Part 2: Messages and Events

This second (and last) part of the tutorial is broken up into several sections:

1. the section called “Subscribe to Chat Message Events”
2. the section called “Show Received Messages”
   a. the section called “Creating a Message Component”
   b. the section called “Recognizing Messages Sent by the Current User”
   c. the section called “Creating a Message List Component”
   d. the section called “Rendering a List of Chat Messages”
3. the section called “Perform Actions in a Chat Room”
   a. the section called “Sending a Message”
   b. the section called “Deleting a Message”
4. the section called “Next Steps”

Note: In some cases, code examples for JavaScript and TypeScript are identical, so they are combined.

For full SDK documentation, start with Amazon IVS Chat Client Messaging SDK (here in the *Amazon IVS Chat User Guide*) and the Chat Client Messaging: SDK for JavaScript Reference (on GitHub).

Prerequisite

Be sure you have completed Part 1 of this tutorial, Chat Rooms.

Subscribe to Chat Message Events

The `ChatRoom` instance uses events to communicate when events occur in a chat room. To start implementing the chat experience, you need to show your users when others send a message in the room to which they're connected.

Here, you subscribe to chat message events. Later, we'll show you how to update a message list you create, which updates with every message/event.

In your App, inside the `useEffect` hook, subscribe to all message events:

```
// App.tsx / App.jsx

useEffect(() => {
  // ...
  const unsubscribeOnMessageReceived = room.addListener('message', (message) => {});

  return () => {
    // ...
    unsubscribeOnMessageReceived();
  };
}, []);
```

Show Received Messages

Receiving messages is a core part of the chat experience. Using the Chat JS SDK, you can set up your code to easily receive events from other users connected to a chat room.

Later, we'll show you how to perform actions in a chat room that leverage the components you create here.

In your App, define a state named `messages` with a `ChatMessage` array type named `messages`:

TypeScript

```
// App.tsx

// ...

import { ChatRoom, ChatMessage, ConnectionState } from 'amazon-ivs-chat-messaging';

export default function App() {
  const [messages, setMessages] = useState<ChatMessage[]>([]);

  //...
}
```

JavaScript

```
// App.jsx

// ...

export default function App() {
  const [messages, setMessages] = useState([]);

  //...
}
```

Next, in the `message` listener function, append `message` to the `messages` array:

```
// App.jsx / App.tsx

// ...

const unsubscribeOnMessageReceived = room.addListener('message', (message) => {
  setMessages((msgs) => [...msgs, message]);
});

// ...
```

Below we step through the tasks to show received messages:

1. the section called “Creating a Message Component”

2. the section called “Recognizing Messages Sent by the Current User”
3. the section called “Creating a Message List Component”
4. the section called “Rendering a List of Chat Messages”

Creating a Message Component

The `Message` component is responsible for rendering the contents of a message received by your chat room. In this section, you create a messages component for rendering individual chat messages in the App.

Create a new file in the `src` directory and name it `Message`. Pass in the `ChatMessage` type for this component, and pass the `content` string from `ChatMessage` properties to display message text received from chat-room message listeners. In the Project Navigator, go to `Message`.

TypeScript

```
// Message.tsx

import * as React from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';

type Props = {
  message: ChatMessage;
}

export const Message = ({ message }: Props) => {
  return (
    <div style={{ backgroundColor: 'silver', padding: 6, borderRadius: 10, margin:
 10 }}>
      <p>{message.content}</p>
    </div>
  );
};
```

JavaScript

```
// Message.jsx

import * as React from 'react';

export const Message = ({ message }) => {
```

```
  return (
    <div style={{ backgroundColor: 'silver', padding: 6, borderRadius: 10, margin:
 10 }}>
      <p>{message.content}</p>
    </div>
  );
};
```

Tip: Use this component to store different properties that you want to render in your message rows; for example, avatar URLs, user names, and timestamps of when the message was sent.

Recognizing Messages Sent by the Current User

To recognize the message sent by the current user, we modify the code and create a React context for storing the `userId` of the current user.

Create a new file in the `src` directory and name it `UserContext`:

TypeScript

```
// UserContext.tsx

import React, { ReactNode, useState, useContext, createContext } from 'react';

type UserContextType = {
  userId: string;
  setUserId: (userId: string) => void;
};

const UserContext = createContext<UserContextType | undefined>(undefined);

export const useUserContext = () => {
  const context = useContext(UserContext);

  if (context === undefined) {
    throw new Error('useUserContext must be within UserProvider');
  }

  return context;
};

type UserProviderType = {
```

```
  children: ReactNode;
}

export const UserProvider = ({ children }: UserProviderType) => {
  const [userId, setUserId] = useState('Mike');

  return <UserContext.Provider value={{ userId, setUserId }}>{children}</
UserContext.Provider>;
};
```

JavaScript

```
// UserContext.jsx

import React, { useState, useContext, createContext } from 'react';

const UserContext = createContext(undefined);

export const useUserContext = () => {
  const context = useContext(UserContext);

  if (context === undefined) {
    throw new Error('useUserContext must be within UserProvider');
  }

  return context;
};

export const UserProvider = ({ children }) => {
  const [userId, setUserId] = useState('Mike');

  return <UserContext.Provider value={{ userId, setUserId }}>{children}</
UserContext.Provider>;
};
```

Note: Here we used the `useState` hook to store the `userId` value. In the future you can use `setUserId` to change user context or for login purposes.

Next, replace `userId` in the first parameter passed to `tokenProvider`, using the previously created context:

```
// App.jsx / App.tsx
```

```
// ...

import { useUserContext } from './UserContext';

// ...


export default function App() {
  const [messages, setMessages] = useState<ChatMessage[]>([]);
  const { userId } = useUserContext();
  const [room] = useState(
    () =>
      new ChatRoom({
        regionOrUrl: process.env.REGION,
        tokenProvider: () => tokenProvider(userId, ['SEND_MESSAGE']),
      }),
  );

  // ...
}
```

In your `Message` component, use the `UserContext` created before, declare the `isMine` variable, match the sender's `userId` with the `userId` from the context, and apply different styles of messages for the current user.

TypeScript

```
// Message.tsx

import * as React from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

type Props = {
  message: ChatMessage;
}

export const Message = ({ message }: Props) => {
  const { userId } = useUserContext();

  const isMine = message.sender.userId === userId;
```

```
  return (
    <div style={{ backgroundColor: isMine ? 'lightblue' : 'silver', padding: 6,
 borderRadius: 10, margin: 10 }}>
      <p>{message.content}</p>
    </div>
  );
};
```

JavaScript

```
// Message.jsx

import * as React from 'react';
import { useUserContext } from './UserContext';

export const Message = ({ message }) => {
  const { userId } = useUserContext();

  const isMine = message.sender.userId === userId;

  return (
    <div style={{ backgroundColor: isMine ? 'lightblue' : 'silver', padding: 6,
 borderRadius: 10, margin: 10 }}>
      <p>{message.content}</p>
    </div>
  );
};
```

Creating a Message List Component

The `MessageList` component is responsible for displaying a chat room's conversation over time. The `MessageList` file is the container that holds all our messages. `Message` is one row in `MessageList`.

Create a new file in the `src` directory and name it `MessageList`. Define `Props` with `messages` of type `ChatMessage` array. Inside the body, map our `messages` property and pass `Props` to your `Message` component.

TypeScript

```
// MessageList.tsx
```

```
import React from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { Message } from './Message';

interface Props {
  messages: ChatMessage[];
}

export const MessageList = ({ messages }: Props) => {
  return (
    <div>
      {messages.map((message) => (
        <Message key={message.id} message={message}/>
      ))}
    </div>
  );
};
```

JavaScript

```
// MessageList.jsx

import React from 'react';
import { Message } from './Message';

export const MessageList = ({ messages }) => {
  return (
    <div>
      {messages.map((message) => (
        <Message key={message.id} message={message} />
      ))}
    </div>
  );
};
```

Rendering a List of Chat Messages

Now bring your new `MessageList` into your main App component:

```
// App.jsx / App.tsx
```

```
import { MessageList } from './MessageList';
// ...

return (
  <div style={{ display: 'flex', flexDirection: 'column', padding: 10 }}>
    <h4>Connection State: {connectionState}</h4>
    <MessageList messages={messages} />
    <div style={{ flexDirection: 'row', display: 'flex', width: '100%',
 backgroundColor: 'red' }}>
      <MessageInput value={messageToSend} onValueChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </div>
  </div>
);

// ...
```

All the puzzle pieces are now in place for your App to start rendering messages received by your chat room. Continue below to see how to perform actions in a chat room that leverage the components you have created.

Perform Actions in a Chat Room

Sending messages and performing moderator actions within a chat room are some of the primary ways you interact with a chat room. Here you will learn how to use various `ChatRequest` objects to perform common actions in Chatterbox, such as sending a message, deleting a message, and disconnecting other users.

All actions in a chat room follow a common pattern: for every action you perform in a chat room, there is a corresponding request object. For each request there is a corresponding response object that you receive on request confirmation.

As long as your users are granted the correct permissions when you create a chat token, they can successfully perform the corresponding action(s) using the request objects to see what requests you can perform in a chat room.

Below, we explain how to send a message and delete a message.

Sending a Message

The `SendMessageRequest` class enables sending messages in a chat room. Here, you modify your App to send a message request using the component you created in Create a Message Input (in Part 1 of this tutorial).

To start, define a new boolean property named `isSending` with the `useState` hook. Use this new property to toggle the disabled state of your `button` HTML element, using the `isSendDisabled` constant. In the event handler for your `SendButton`, clear the value for `messageToSend` and set `isSending` to true.

Since you will be making an API call from this button, adding the `isSending` boolean helps prevent multiple API calls from occuring at the same time, by disabling user interactions on your `SendButton` until the request is complete.

```
// App.jsx / App.tsx

// ...

const [isSending, setIsSending] = useState(false);

// ...

const onMessageSend = () => {
  setIsSending(true);
  setMessageToSend('');
};

// ...

const isSendDisabled = connectionState !== 'connected' || isSending;

// ...
```

Prepare the request by creating a new `SendMessageRequest` instance, passing message content to the constructor. After setting the `isSending` and `messageToSend` states, call the `sendMessage` method, which sends the request to the chat room. Finally, clear the `isSending` flag on receiving confirmation or rejection of the request.

TypeScript

```
// App.tsx

// ...
import { ChatMessage, ChatRoom, ConnectionState, SendMessageRequest } from 'amazon-
ivs-chat-messaging'
// ...

const onMessageSend = async () => {
  const request = new SendMessageRequest(messageToSend);
  setIsSending(true);
  setMessageToSend('');

  try {
    const response = await room.sendMessage(request);
  } catch (e) {
    console.log(e);
    // handle the chat error here...
  } finally {
    setIsSending(false);
  }
};

// ...
```

JavaScript

```
// App.jsx

// ...
import { ChatRoom, SendMessageRequest } from 'amazon-ivs-chat-messaging'
// ...

const onMessageSend = async () => {
  const request = new SendMessageRequest(messageToSend);
  setIsSending(true);
  setMessageToSend('');

  try {
    const response = await room.sendMessage(request);
  } catch (e) {
    console.log(e);
```

```
    // handle the chat error here...
  } finally {
    setIsSending(false);
  }
};

// ...
```

Give Chatterbox a run: try sending a message by drafting one with your `MessageInput` and tapping your `SendButton`. You should see your sent message rendered within the `MessageList` that you created earlier.

Deleting a Message

To delete a message from a chat room, you need to have the proper capability. Capabilities are granted during the initialization of the chat token that you use when authenticating to a chat room. For the purposes of this section, the `ServerApp` from Set Up a Local Authentication/ Authorization Server (in Part 1 of this tutorial) lets you specify moderator capabilities. This is done in your app using the `tokenProvider` object that you created in Build a Token Provider (also in Part 1).

Here you modify your `Message` by adding a function to delete the message.

First, open the `App.tsx` and add the `DELETE_MESSAGE` capability. (`capabilities` is the second parameter of your `tokenProvider` function.)

Note: This is how your `ServerApp` informs the IVS Chat APIs that the user being associated with the resulting chat token can delete messages in a chat room. In a real-world situation you probably will have more complex backend logic to manage user capabilities in your server app's infrastructure.

TypeScript

```
// App.tsx

// ...

const [room] = useState( () =>
    new ChatRoom({
      regionOrUrl: process.env.REGION as string,
```

```
      tokenProvider: () => tokenProvider(userId, ['SEND_MESSAGE',
 'DELETE_MESSAGE']),
    }),
);

// ...
```

JavaScript

```
// App.jsx

// ...

const [room] = useState( () =>
  new ChatRoom({
    regionOrUrl: process.env.REGION,
    tokenProvider: () => tokenProvider(userId, ['SEND_MESSAGE', 'DELETE_MESSAGE']),
  }),
);

// ...
```

In the next steps, you update your `Message` to display a delete button.

Open `Message` and define a new boolean state named `isDeleting` using the `useState` hook with an initial value of `false`. Using this state, update the contents of your `Button` to be different depending on the current state of `isDeleting`. Disable your button when `isDeleting` is true; this prevents you from attempting to make two delete message requests at the same time.

TypeScript

```
// Message.tsx

import React, { useState } from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

type Props = {
  message: ChatMessage;
}
```

```
export const Message = ({ message }: Props) => {
  const { userId } = useUserContext();
  const [isDeleting, setIsDeleting] = useState(false);

  const isMine = message.sender.userId === userId;

  return (
    <div style={{ backgroundColor: isMine ? 'lightblue' : 'silver', padding: 6,
 borderRadius: 10, margin: 10 }}>
      <p>{message.content}</p>
      <button disabled={isDeleting}>Delete</button>
    </div>
  );
};
```

JavaScript

```
// Message.jsx

import React from 'react';
import { useUserContext } from './UserContext';

export const Message = ({ message }) => {
  const { userId } = useUserContext();
  const [isDeleting, setIsDeleting] = useState(false);

  return (
    <div style={{ backgroundColor: isMine ? 'lightblue' : 'silver', padding: 6,
 borderRadius: 10, margin: 10 }}>
      <p>{message.content}</p>
      <button disabled={isDeleting}>Delete</button>
    </div>
  );
};
```

Define a new function called `onDelete` that accepts a string as one of its parameters and returns `Promise`. In the body of your `Button`'s action closure, use `setIsDeleting` to toggle your `isDeleting` boolean before and after a call to `onDelete`. For the string parameter, pass in your component message ID.

TypeScript

```
// Message.tsx

import React, { useState } from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

export type Props = {
  message: ChatMessage;
  onDelete(id: string): Promise<void>;
};

export const Message = ({ message onDelete }: Props) => {
  const { userId } = useUserContext();
  const [isDeleting, setIsDeleting] = useState(false);
  const isMine = message.sender.userId === userId;
  const handleDelete = async () => {
    setIsDeleting(true);
    try {
      await onDelete(message.id);
    } catch (e) {
      console.log(e);
      // handle chat error here...
    } finally {
      setIsDeleting(false);
    }
  };

  return (
    <div style={{ backgroundColor: isMine ? 'lightblue' : 'silver', padding: 6,
 borderRadius: 10, margin: 10 }}>
      <p>{content}</p>
      <button onClick={handleDelete} disabled={isDeleting}>
        Delete
      </button>
    </div>
  );
};
```

JavaScript

```
// Message.jsx
```

```
import React, { useState } from 'react';
import { useUserContext } from './UserContext';

export const Message = ({ message, onDelete }) => {
  const { userId } = useUserContext();
  const [isDeleting, setIsDeleting] = useState(false);
  const isMine = message.sender.userId === userId;
  const handleDelete = async () => {
    setIsDeleting(true);
    try {
      await onDelete(message.id);
    } catch (e) {
      console.log(e);
      // handle the exceptions here...
    } finally {
      setIsDeleting(false);
    }
  };

  return (
    <div style={{ backgroundColor: 'silver', padding: 6, borderRadius: 10, margin:
 10 }}>
      <p>{message.content}</p>
      <button onClick={handleDelete} disabled={isDeleting}>
        Delete
      </button>
    </div>
  );
};
```

Next, you update your `MessageList` to reflect the latest changes to your `Message` component.

Open `MessageList` and define a new function called `onDelete` that accepts a string as a parameter and returns `Promise`. Update your `Message` and pass it through the properties of `Message`. The string parameter in your new closure will be the ID of the message that you want to delete, which gets passed from your `Message`.

TypeScript

```
// MessageList.tsx
```

```
import * as React from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { Message } from './Message';

interface Props {
  messages: ChatMessage[];
  onDelete(id: string): Promise<void>;
}

export const MessageList = ({ messages, onDelete }: Props) => {
  return (
    <>
      {messages.map((message) => (
        <Message key={message.id} onDelete={onDelete} content={message.content}
 id={message.id} />
      ))}
    </>
  );
};
```

JavaScript

```
// MessageList.jsx

import * as React from 'react';
import { Message } from './Message';

export const MessageList = ({ messages, onDelete }) => {
  return (
    <>
      {messages.map((message) => (
        <Message key={message.id} onDelete={onDelete} content={message.content}
 id={message.id} />
      ))}
    </>
  );
};
```

Next, you update your App to reflect the latest changes to your `MessageList`.

In App, define a function named onDeleteMessage and pass it to the `MessageList` onDelete property:

TypeScript

```
// App.tsx

// ...

const onDeleteMessage = async (id: string) => {};

return (
  <div style={{ display: 'flex', flexDirection: 'column', padding: 10 }}>
    <h4>Connection State: {connectionState}</h4>
    <MessageList onDelete={onDeleteMessage} messages={messages} />
    <div style={{ flexDirection: 'row', display: 'flex', width: '100%' }}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onSendPress={onMessageSend} />
    </div>
  </div>
);

// ...
```

JavaScript

```
// App.jsx

// ...

const onDeleteMessage = async (id) => {};

return (
  <div style={{ display: 'flex', flexDirection: 'column', padding: 10 }}>
    <h4>Connection State: {connectionState}</h4>
    <MessageList onDelete={onDeleteMessage} messages={messages} />
    <div style={{ flexDirection: 'row', display: 'flex', width: '100%' }}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onSendPress={onMessageSend} />
    </div>
  </div>
);

// ...
```

Prepare a request by creating a new instance of `DeleteMessageRequest`, passing the relevant message ID to the constructor parameter, and call `deleteMessage` that accepts the prepared request above:

TypeScript

```
// App.tsx

// ...

const onDeleteMessage = async (id: string) => {
  const request = new DeleteMessageRequest(id);
  await room.deleteMessage(request);
};

// ...
```

JavaScript

```
// App.jsx

// ...

const onDeleteMessage = async (id) => {
  const request = new DeleteMessageRequest(id);
  await room.deleteMessage(request);
};

// ...
```

Next, you update your `messages` state to reflect a new list of messages that omits the message you just deleted.

In the `useEffect` hook, listen for the `messageDelete` event and update your `messages` state array by deleting the message with a matching ID to the `message` parameter.

Note: The `messageDelete` event might be raised for messages being deleted by the current user or any other user in the room. Handling it in the event handler (instead of next to the `deleteMessage` request) allows you to unify delete-message handling.

```
// App.jsx / App.tsx
```

```
// ...

const unsubscribeOnMessageDeleted = room.addListener('messageDelete',
 (deleteMessageEvent) => {
  setMessages((prev) => prev.filter((message) => message.id !==
 deleteMessageEvent.id));
});

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

You are now able to delete users from a chat room in your chat app.

Next Steps

As an experiment, try implementing other actions in a room like disconnecting another user.

IVS Chat Client Messaging SDK: React Native Tutorial Part 1: Chat Rooms

This is the first of a two-part tutorial. You will learn the essentials of working with the Amazon IVS Chat Client Messaging JavaScript SDK by building a fully functional app using React Native. We call the app *Chatterbox*.

The intended audience is experienced developers who are new to the Amazon IVS Chat Messaging SDK. You should be comfortable with the TypeScript or JavaScript programming languages and React Native library.

For brevity, we'll refer to the Amazon IVS Chat Client Messaging JavaScript SDK as the Chat JS SDK.

Note: In some cases, code examples for JavaScript and TypeScript are identical, so they are combined.

This first part of the tutorial is broken up into several sections:

1. the section called “Set Up a Local Authentication/Authorization Server”
2. the section called “Create a Chatterbox Project”
3. the section called “Connect to a Chat Room”
4. the section called “Build a Token Provider”
5. the section called “Observe Connection Updates”
6. the section called “Create a Send Button Component”
7. the section called “Create a Message Input”
8. the section called “Next Steps”

Prerequisites

- Be familiar with TypeScript or JavaScript and the React Native library. If you're unfamiliar with React Native, learn the basics in Intro to React Native.
- Read and understand *Getting Started with IVS Chat*.
- Create an AWS IAM user with the CreateChatToken and CreateRoom capabilities defined in an existing IAM policy. (See *Getting Started with IVS Chat*.)
- Ensure that the secret/access keys for this user are stored in an AWS credentials file. For instructions, see the AWS CLI User Guide (especially Configuration and credential file settings).
- Create a chat room and save its ARN. See *Getting Started with IVS Chat*. (If you don’t save the ARN, you can look it up later with the console or Chat API.)
- Install the Node.js 14+ environment with the NPM or Yarn package manager.

Set Up a Local Authentication/Authorization Server

Your backend application is responsible for both creating chat rooms and generating the chat tokens that are needed for the Chat JS SDK to authenticate and authorize your clients for your chat rooms. You must use your own backend since you cannot securely store AWS keys in a mobile app; sophisticated attackers could extract these and gain access to your AWS account.

See Create a Chat Token in *Getting Started with Amazon IVS Chat*. As shown in the flowchart there, your server-side application is responsible for creating a chat token. This means your app must provide its own means of generating a chat token by requesting one from your server-side application.

In this section, you will learn the basics of creating a token provider in your backend. We use the express framework to create a live local server that manages the creation of chat tokens using your local AWS environment.

Create an empty npm project using NPM. Create a directory to hold your application, and make that your working directory:

```
$ mkdir backend & cd backend
```

Use `npm init` to create a `package.json` file for your application:

```
$ npm init
```

This command prompts you for several things, including the name and version of your application. For now, just press **RETURN** to accept the defaults for most of them, with the following exception:

```
entry point: (index.js)
```

Press **RETURN** to accept the suggested default filename of `index.js` or enter whatever you want the name of the main file to be.

Now install required dependencies:

```
$ npm install express aws-sdk cors dotenv
```

`aws-sdk` requires configuration-environment variables, which automatically load from a file named `.env` located in the root directory. To configure it, create a new file named `.env` and fill in the missing configuration information:

```
# .env

# The region to send service requests to.
AWS_REGION=us-west-2

# Access keys use an access key ID and secret access key
# that you use to sign programmatic requests to AWS.

# AWS access key ID.
```

```
AWS_ACCESS_KEY_ID=...

# AWS secret access key.
AWS_SECRET_ACCESS_KEY=...
```

Now we create an entry-point file in the root directory with the name you entered above in the `npm init` command. In this case, we use `index.js`, and import all required packages:

```
// index.js
import express from 'express';
import AWS from 'aws-sdk';
import 'dotenv/config';
import cors from 'cors';
```

Now create a new instance of `express`:

```
const app = express();
const port = 3000;

app.use(express.json());
app.use(cors({ origin: ['http://127.0.0.1:5173'] }));
```

After that you can create your first endpoint POST method for the token provider. Take the required parameters from the request body (`roomId`, `userId`, `capabilities`, and `sessionDurationInMinutes`):

```
app.post('/create_chat_token', (req, res) => {
  const { roomIdentifier, userId, capabilities, sessionDurationInMinutes } = req.body
 || {};
});
```

Add validation of required fields:

```
app.post('/create_chat_token', (req, res) => {
  const { roomIdentifier, userId, capabilities, sessionDurationInMinutes } = req.body
 || {};

  if (!roomIdentifier || !userId) {
    res.status(400).json({ error: 'Missing parameters: `roomIdentifier`, `userId`' });
    return;
```

```
  }
});
```

After preparing the POST method, we integrate `createChatToken` with `aws-sdk` for the core functionality of authentication/authorization:

```
app.post('/create_chat_token', (req, res) => {
  const { roomIdentifier, userId, capabilities, sessionDurationInMinutes } = req.body
 || {};

  if (!roomIdentifier || !userId || !capabilities) {
    res.status(400).json({ error: 'Missing parameters: `roomIdentifier`, `userId`,
 `capabilities`' });
    return;
  }

  ivsChat.createChatToken({ roomIdentifier, userId, capabilities,
 sessionDurationInMinutes }, (error, data) => {
    if (error) {
      console.log(error);
      res.status(500).send(error.code);
    } else if (data.token) {
      const { token, sessionExpirationTime, tokenExpirationTime } = data;
      console.log(`Retrieved Chat Token: ${JSON.stringify(data, null, 2)}`);

      res.json({ token, sessionExpirationTime, tokenExpirationTime });
    }
  });
});
```

At the end of the file, add a port listener for your `express` app:

```
app.listen(port, () => {
  console.log(`Backend listening on port ${port}`);
});
```

Now you can run the server with the following command from the project's root:

```
$ node index.js
```

Tip: This server accepts URL requests at https://localhost:3000.

Create a Chatterbox Project

First you create the React Native project called `chatterbox`. Run this command:

```
npx create-expo-app
```

Or create an expo project with a TypeScript template.

```
npx create-expo-app -t expo-template-blank-typescript
```

You can integrate the Chat Client Messaging JS SDK via Node Package Manager or Yarn Package Manager:

- Npm: `npm install amazon-ivs-chat-messaging`
- Yarn: `yarn add amazon-ivs-chat-messaging`

Connect to a Chat Room

Here you create a `ChatRoom` and connect to it using asynchronous methods. The `ChatRoom` class manages your user's connection to the Chat JS SDK. To successfully connect to a chat room, you must provide an instance of `ChatToken` within your React application.

Navigate to the App file that's created in the default `chatterbox` project and delete everything that a functional component returns. None of the pre-populated code is needed. At this point, our App is pretty empty.

TypeScript/JavaScript:

```
// App.tsx / App.jsx

import * as React from 'react';
import { Text } from 'react-native';

export default function App() {
  return <Text>Hello!</Text>;
}
```

Create a new `ChatRoom` instance and pass it to state using the `useState` hook. It requires passing `regionOrUrl` (the AWS region in which your chat room is hosted) and `tokenProvider` (used for the backend authentication/authorization flow that is created in subsequent steps).

Important: You must use the same AWS region as the one in which you created the room in Getting Started with Amazon IVS Chat. The API is an AWS regional service. For a list of supported regions and Amazon IVS Chat HTTPS service endpoints, see the Amazon IVS Chat regions page.

TypeScript/JavaScript:

```
// App.jsx / App.tsx

import React, { useState } from 'react';
import { Text } from 'react-native';
import { ChatRoom } from 'amazon-ivs-chat-messaging';

export default function App() {
  const [room] = useState(() =>
    new ChatRoom({
      regionOrUrl: process.env.REGION,
      tokenProvider: () => {},
    }),
  );

  return <Text>Hello!</Text>;
}
```

Build a Token Provider

As the next step, we need to build a parameterless `tokenProvider` function that is required by the `ChatRoom` constructor. First, we will create a `fetchChatToken` function that will make a POST request to the backend application that you set up in the section called "Set Up a Local Authentication/Authorization Server". Chat tokens contain the information needed for the SDK to successfully establish a chat-room connection. The Chat API uses these tokens as a secure way of validating a user's identity, capabilities within a chat room, and session duration.

In the Project navigator, create a new TypeScript/JavaScript file named `fetchChatToken`. Build a fetch request to the `backend` application and return the `ChatToken` object from the response. Add the request body properties needed for creating a chat token. Use the rules defined for Amazon Resource Names (ARNs). These properties are documented in the CreateChatToken operation.

Note: The URL you're using here is the same URL that your local server created when you ran the backend application.

TypeScript

```
// fetchChatToken.ts

import { ChatToken } from 'amazon-ivs-chat-messaging';

type UserCapability = 'DELETE_MESSAGE' | 'DISCONNECT_USER' | 'SEND_MESSAGE';

export async function fetchChatToken(
  userId: string,
  capabilities: UserCapability[] = [],
  attributes?: Record<string, string>,
  sessionDurationInMinutes?: number,
): Promise<ChatToken> {
  const response = await fetch(`${process.env.BACKEND_BASE_URL}/create_chat_token`,
 {
    method: 'POST',
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      userId,
      roomIdentifier: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
  });

  const token = await response.json();

  return {
    ...token,
    sessionExpirationTime: new Date(token.sessionExpirationTime),
    tokenExpirationTime: new Date(token.tokenExpirationTime),
  };
}
```

JavaScript

```
// fetchChatToken.js

export async function fetchChatToken(
  userId,
  capabilities = [],
  attributes,
  sessionDurationInMinutes) {
  const response = await fetch(`${process.env.BACKEND_BASE_URL}/create_chat_token`,
 {
    method: 'POST',
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      userId,
      roomIdentifier: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
  });

  const token = await response.json();

  return {
    ...token,
    sessionExpirationTime: new Date(token.sessionExpirationTime),
    tokenExpirationTime: new Date(token.tokenExpirationTime),
  };
}
```

Observe Connection Updates

Reacting to changes in a chat room's connection state are essential parts of making a chat app. Let's start with subscribing to relevant events:

TypeScript/JavaScript:

```
// App.tsx / App.jsx
```

```
import React, { useState, useEffect } from 'react';
import { Text } from 'react-native';
import { ChatRoom } from 'amazon-ivs-chat-messaging';
import { fetchChatToken } from './fetchChatToken';

export default function App() {
  const [room] = useState(
    () =>
      new ChatRoom({
        regionOrUrl: process.env.REGION,
        tokenProvider: () => fetchChatToken('Mike', ['SEND_MESSAGE']),
      }),
  );

  useEffect(() => {
    const unsubscribeOnConnecting = room.addListener('connecting', () => {});
    const unsubscribeOnConnected = room.addListener('connect', () => {});
    const unsubscribeOnDisconnected = room.addListener('disconnect', () => {});

    return () => {
      // Clean up subscriptions.
      unsubscribeOnConnecting();
      unsubscribeOnConnected();
      unsubscribeOnDisconnected();
    };
  }, [room]);

  return <Text>Hello!</Text>;
}
```

Next, we need to provide the ability to read the connection state. We use our `useState` hook to create some local state in App and set the connection state inside each listener.

TypeScript/JavaScript:

```
// App.tsx / App.jsx

import React, { useState, useEffect } from 'react';
import { Text } from 'react-native';
import { ChatRoom, ConnectionState } from 'amazon-ivs-chat-messaging';
import { fetchChatToken } from './fetchChatToken';
```

```
export default function App() {
  const [room] = useState(
    () =>
      new ChatRoom({
        regionOrUrl: process.env.REGION,
        tokenProvider: () => fetchChatToken('Mike', ['SEND_MESSAGE']),
      }),
  );
  const [connectionState, setConnectionState] =
 useState<ConnectionState>('disconnected');

  useEffect(() => {
    const unsubscribeOnConnecting = room.addListener('connecting', () => {
      setConnectionState('connecting');
    });

    const unsubscribeOnConnected = room.addListener('connect', () => {
      setConnectionState('connected');
    });

    const unsubscribeOnDisconnected = room.addListener('disconnect', () => {
      setConnectionState('disconnected');
    });

    return () => {
      unsubscribeOnConnecting();
      unsubscribeOnConnected();
      unsubscribeOnDisconnected();
    };
  }, [room]);

  return <Text>Hello!</Text>;
}
```

After subscribing to the connection state, display the connection state and connect to the chat room using the `room.connect` method inside the `useEffect` hook:

TypeScript/JavaScript:

```
// App.tsx / App.jsx

// ...
```

```
useEffect(() => {
  const unsubscribeOnConnecting = room.addListener('connecting', () => {
    setConnectionState('connecting');
  });

  const unsubscribeOnConnected = room.addListener('connect', () => {
    setConnectionState('connected');
  });

  const unsubscribeOnDisconnected = room.addListener('disconnect', () => {
    setConnectionState('disconnected');
  });

  room.connect();

  return () => {
    unsubscribeOnConnecting();
    unsubscribeOnConnected();
    unsubscribeOnDisconnected();
  };
}, [room]);

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
  </SafeAreaView>
);

const styles = StyleSheet.create({
  root: {
    flex: 1,
  }
});

// ...
```

You have successfully implemented a chat-room connection.

Create a Send Button Component

In this section you create a send button that has a different design for each connection state. The send button facilitates the sending of messages in a chat room. It also serves as a visual indicator of whether/when messages can be sent; e.g., in the face of dropped connections or expired chat sessions.

First, create a new file in the `src` directory of your Chatterbox project and name it `SendButton`. Then, create a component that will display a button for your chat application. Export your `SendButton` and import it to `App`. In the empty `<View></View>`, add `<SendButton />`.

TypeScript

```
// SendButton.tsx

import React from 'react';
import { TouchableOpacity, Text, ActivityIndicator, StyleSheet } from 'react-
native';

interface Props {
  onPress?: () => void;
  disabled: boolean;
  loading: boolean;
}

export const SendButton = ({ onPress, disabled, loading }: Props) => {
  return (
    <TouchableOpacity style={styles.root} disabled={disabled} onPress={onPress}>
      {loading ? <Text>Send</Text> : <ActivityIndicator />}
    </TouchableOpacity>
  );
};

const styles = StyleSheet.create({
  root: {
    width: 50,
    height: 50,
    borderRadius: 30,
    marginLeft: 10,
    justifyContent: 'center',
    alignContent: 'center',
  }
```

```
});

// App.tsx

import { SendButton } from './SendButton';

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <SendButton />
  </SafeAreaView>
);
```

JavaScript

```
// SendButton.jsx

import React from 'react';
import { TouchableOpacity, Text, ActivityIndicator, StyleSheet } from 'react-
native';

export const SendButton = ({ onPress, disabled, loading }) => {
  return (
    <TouchableOpacity style={styles.root} disabled={disabled} onPress={onPress}>
      {loading ? <Text>Send</Text> : <ActivityIndicator />}
    </TouchableOpacity>
  );
};

const styles = StyleSheet.create({
  root: {
    width: 50,
    height: 50,
    borderRadius: 30,
    marginLeft: 10,
    justifyContent: 'center',
    alignContent: 'center',
  }
});

// App.jsx
```

```
import { SendButton } from './SendButton';

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <SendButton />
  </SafeAreaView>
);
```

Next, in App define a function named onMessageSend and pass it to the SendButton onPress property. Define another variable named isSendDisabled (which prevents sending messages when the room is not connected) and pass it to the SendButton disabled property.

TypeScript/JavaScript:

```
// App.jsx / App.tsx

// ...

const onMessageSend = () => {};

const isSendDisabled = connectionState !== 'connected';

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
  </SafeAreaView>
);

// ...
```

Create a Message Input

The Chatterbox message bar is the component that you will interact with to send messages to a chat room. Typically it contains a text input for composing your message and a button to send your message.

To create a `MessageInput` component, first create a new file in the `src` directory and name it `MessageInput`. Then, create an input component that will display an input for your chat application. Export your `MessageInput` and import it to `App` (above the `<SendButton />`).

Create a new state named `messageToSend` using the `useState` hook, with an empty string as its default value. In the body of your app, pass `messageToSend` to the `value` of `MessageInput` and pass the `setMessageToSend` to the on`MessageChange` property:

TypeScript

```
// MessageInput.tsx

import * as React from 'react';

interface Props {
  value?: string;
  onValueChange?: (value: string) => void;
}

export const MessageInput = ({ value, onValueChange }: Props) => {
  return (
    <TextInput style={styles.input} value={value} onChangeText={onValueChange}
 placeholder="Send a message" />
  );
};

const styles = StyleSheet.create({
  input: {
    fontSize: 20,
    backgroundColor: 'rgb(239,239,240)',
    paddingHorizontal: 18,
    paddingVertical: 15,
    borderRadius: 50,
    flex: 1,
  }
})

// App.tsx

// ...

import { MessageInput } from './MessageInput';
```

```
// ...

export default function App() {
  const [messageToSend, setMessageToSend] = useState('');

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <View style={styles.messageBar}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </View>
  </SafeAreaView>
);

const styles = StyleSheet.create({
  root: {
    flex: 1,
  },
  messageBar: {
    borderTopWidth: StyleSheet.hairlineWidth,
    borderTopColor: 'rgb(160,160,160)',
    flexDirection: 'row',
    padding: 16,
    alignItems: 'center',
    backgroundColor: 'white',
  }
});
```

JavaScript

```
// MessageInput.jsx

import * as React from 'react';

export const MessageInput = ({ value, onValueChange }) => {
  return (
    <TextInput style={styles.input} value={value} onChangeText={onValueChange}
 placeholder="Send a message" />
  );
};
```

```
const styles = StyleSheet.create({
  input: {
    fontSize: 20,
    backgroundColor: 'rgb(239,239,240)',
    paddingHorizontal: 18,
    paddingVertical: 15,
    borderRadius: 50,
    flex: 1,
  }
})

// App.jsx

// ...

import { MessageInput } from './MessageInput';

// ...

export default function App() {
  const [messageToSend, setMessageToSend] = useState('');

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <View style={styles.messageBar}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </View>
  </SafeAreaView>
);

const styles = StyleSheet.create({
  root: {
    flex: 1,
  },
  messageBar: {
    borderTopWidth: StyleSheet.hairlineWidth,
    borderTopColor: 'rgb(160,160,160)',
    flexDirection: 'row',
    padding: 16,
```

```
    alignItems: 'center',
    backgroundColor: 'white',
  }
});
```

Next Steps

Now that you finished building a message bar for Chatterbox, proceed to Part 2 of this React Native tutorial, Messages and Events.

IVS Chat Client Messaging SDK: React Native Tutorial Part 2: Messages and Events

This second (and last) part of the tutorial is broken up into several sections:

1. the section called “Subscribe to Chat Message Events”
2. the section called “Show Received Messages”
   a. the section called “Creating a Message Component”
   b. the section called “Recognizing Messages Sent by the Current User”
   c. the section called “Rendering a List of Chat Messages”
3. the section called “Perform Actions in a Chat Room”
   a. the section called “Sending a Message”
   b. the section called “Deleting a Message”
4. the section called “Next Steps”

Note: In some cases, code examples for JavaScript and TypeScript are identical, so they are combined.

Prerequisite

Be sure you have completed Part 1 of this tutorial, Chat Rooms.

Subscribe to Chat Message Events

The `ChatRoom` instance uses events to communicate when events occur in a chat room. To start implementing the chat experience, you need to show your users when others send a message in the room to which they're connected.

Here, you subscribe to chat message events. Later, we'll show you how to update a message list you create, which updates with every message/event.

In your App, inside the `useEffect` hook, subscribe to all message events:

TypeScript/JavaScript:

```
// App.tsx / App.jsx

useEffect(() => {
  // ...
  const unsubscribeOnMessageReceived = room.addListener('message', (message) => {});

  return () => {
    // ...
    unsubscribeOnMessageReceived();
  };
}, []);
```

Show Received Messages

Receiving messages is a core part of the chat experience. Using the Chat JS SDK, you can set up your code to easily receive events from other users connected to a chat room.

Later, we'll show you how to perform actions in a chat room that leverage the components you create here.

In your App, define a state named `messages` with a `ChatMessage` array type named `messages`:

TypeScript

```
// App.tsx

// ...
```

```
import { ChatRoom, ChatMessage, ConnectionState } from 'amazon-ivs-chat-messaging';

export default function App() {
  const [messages, setMessages] = useState<ChatMessage[]>([]);

  //...
}
```

JavaScript

```
// App.jsx

// ...

import { ChatRoom, ConnectionState } from 'amazon-ivs-chat-messaging';

export default function App() {
  const [messages, setMessages] = useState([]);

  //...
}
```

Next, in the `message` listener function, append `message` to the `messages` array:

TypeScript/JavaScript:

```
// App.tsx / App.jsx

// ...

const unsubscribeOnMessageReceived = room.addListener('message', (message) => {
  setMessages((msgs) => [...msgs, message]);
});

// ...
```

Below we step through the tasks to show received messages:

1. the section called “Creating a Message Component”
2. the section called “Recognizing Messages Sent by the Current User”

3. the section called “Rendering a List of Chat Messages”

Creating a Message Component

The `Message` component is responsible for rendering the contents of a message received by your chat room. In this section, you create a messages component for rendering individual chat messages in the App.

Create a new file in the `src` directory and name it `Message`. Pass in the `ChatMessage` type for this component, and pass the `content` string from `ChatMessage` properties to display message text received from chat-room message listeners. In the Project Navigator, go to `Message`.

TypeScript

```
// Message.tsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';

type Props = {
  message: ChatMessage;
}

export const Message = ({ message }: Props) => {
  return (
    <View style={styles.root}>
      <Text>{message.sender.userId}</Text>
      <Text style={styles.textContent}>{message.content}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
```

```
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
});
```

JavaScript

```
// Message.jsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';

export const Message = ({ message }) => {
  return (
    <View style={styles.root}>
      <Text>{message.sender.userId}</Text>
      <Text style={styles.textContent}>{message.content}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
});
```

Tip: Use this component to store different properties that you want to render in your message rows; for example, avatar URLs, user names, and timestamps of when the message was sent.

Recognizing Messages Sent by the Current User

To recognize the message sent by the current user, we modify the code and create a React context for storing the `userId` of the current user.

Create a new file in the `src` directory and name it `UserContext`:

TypeScript

```
// UserContext.tsx

import React from 'react';

const UserContext = React.createContext<string | undefined>(undefined);

export const useUserContext = () => {
  const context = React.useContext(UserContext);

  if (context === undefined) {
    throw new Error('useUserContext must be within UserProvider');
  }

  return context;
};

export const UserProvider = UserContext.Provider;
```

JavaScript

```
// UserContext.jsx

import React from 'react';

const UserContext = React.createContext(undefined);

export const useUserContext = () => {
  const context = React.useContext(UserContext);

  if (context === undefined) {
    throw new Error('useUserContext must be within UserProvider');
  }

  return context;
```

```
};

export const UserProvider = UserContext.Provider;
```

Note: Here we used the `useState` hook to store the `userId` value. In the future you can use `setUserId` to change user context or for login purposes.

Next, replace `userId` in the first parameter passed to `tokenProvider`, using the previously created context. Make sure you add the `SEND_MESSAGE` capability to your token provider, as specified below; it is required to send messages.:

TypeScript

```
// App.tsx

// ...

import { useUserContext } from './UserContext';

// ...


export default function App() {
  const [messages, setMessages] = useState<ChatMessage[]>([]);
  const userId = useUserContext();
  const [room] = useState(
    () =>
      new ChatRoom({
        regionOrUrl: process.env.REGION,
        tokenProvider: () => tokenProvider(userId, ['SEND_MESSAGE']),
      }),
  );

  // ...
}
```

JavaScript

```
// App.jsx

// ...
```

```
import { useUserContext } from './UserContext';

// ...


export default function App() {
  const [messages, setMessages] = useState([]);
  const userId = useUserContext();
  const [room] = useState(
    () =>
      new ChatRoom({
        regionOrUrl: process.env.REGION,
        tokenProvider: () => tokenProvider(userId, ['SEND_MESSAGE']),
      }),
  );

  // ...
}
```

In your `Message` component, use the `UserContext` created before, declare the `isMine` variable, match the sender's `userId` with the `userId` from the context, and apply different styles of messages for the current user.

TypeScript

```
// Message.tsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

type Props = {
  message: ChatMessage;
}

export const Message = ({ message }: Props) => {
  const userId = useUserContext();

  const isMine = message.sender.userId === userId;

  return (
```

```
    <View style={[styles.root, isMine && styles.mine]}>
      {!isMine && <Text>{message.sender.userId}</Text>}
      <Text style={styles.textContent}>{message.content}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
  mine: {
    flexDirection: 'row-reverse',
    backgroundColor: 'lightblue',
  },
});
```

JavaScript

```
// Message.jsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

export const Message = ({ message }) => {
  const userId = useUserContext();

  const isMine = message.sender.userId === userId;

  return (
    <View style={[styles.root, isMine && styles.mine]}>
```

```
      {!isMine && <Text>{message.sender.userId}</Text>}
      <Text style={styles.textContent}>{message.content}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
  mine: {
    flexDirection: 'row-reverse',
    backgroundColor: 'lightblue',
  },
});
```

Rendering a List of Chat Messages

Now list messages using `FlatList` and `Message` component:

TypeScript

```
// App.tsx

// ...

const renderItem = useCallback<ListRenderItem<ChatMessage>>(({ item }) => {
  return (
    <Message key={item.id} message={item} />
  );
}, []);
```

```
return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <FlatList inverted data={messages} renderItem={renderItem} />
    <View style={styles.messageBar}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </View>
  </SafeAreaView>
);

// ...
```

JavaScript

```
// App.jsx

// ...

const renderItem = useCallback(({ item }) => {
  return (
    <Message key={item.id} message={item} />
  );
}, []);

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <FlatList inverted data={messages} renderItem={renderItem} />
    <View style={styles.messageBar}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </View>
  </SafeAreaView>
);

// ...
```

All the puzzle pieces are now in place for your App to start rendering messages received by your chat room. Continue below to see how to perform actions in a chat room that leverage the components you have created.

Perform Actions in a Chat Room

Sending messages and performing moderator actions are some of the primary ways you interact with a chat room. Here you will learn how to use various chat request objects to perform common actions in Chatterbox, such as sending a message, deleting a message, and disconnecting other users.

All actions in a chat room follow a common pattern: for every action you perform in a chat room, there is a corresponding request object. For each request there is a corresponding response object that you receive on request confirmation.

As long as your users are granted the correct capabilities when you create a chat token, they can successfully perform the corresponding action(s) using the request objects to see what requests you can perform in a chat room.

Below, we explain how to send a message and delete a message.

Sending a Message

The `SendMessageRequest` class enables sending messages in a chat room. Here, you modify your App to send a message request using the component you created in Create a Message Input (in Part 1 of this tutorial).

To start, define a new boolean property named `isSending` with the `useState` hook. Use this new property to toggle the disabled state of your `button` element, using the `isSendDisabled` constant. In the event handler for your `SendButton`, clear the value for `messageToSend` and set `isSending` to true.

Since you will be making an API call from this button, adding the `isSending` boolean helps prevent multiple API calls from occuring at the same time, by disabling user interactions on your `SendButton` until the request is complete.

Note: Sending messages works only if you added the `SEND_MESSAGE` capability to your token provider, as covered above in Recognizing Messages Sent by the Current User.

TypeScript/JavaScript:

```
// App.tsx / App.jsx
```

```
// ...

const [isSending, setIsSending] = useState(false);

// ...

const onMessageSend = () => {
  setIsSending(true);
  setMessageToSend('');
};

// ...

const isSendDisabled = connectionState !== 'connected' || isSending;

// ...
```

Prepare the request by creating a new `SendMessageRequest` instance, passing message content to the constructor. After setting the `isSending` and `messageToSend` states, call the `sendMessage` method, which sends the request to the chat room. Finally, clear the `isSending` flag on receiving confirmation or rejection of the request.

TypeScript/JavaScript:

```
// App.tsx / App.jsx

// ...
import { ChatRoom, ConnectionState, SendMessageRequest } from 'amazon-ivs-chat-
messaging'
// ...

const onMessageSend = async () => {
  const request = new SendMessageRequest(messageToSend);
  setIsSending(true);
  setMessageToSend('');

  try {
    const response = await room.sendMessage(request);
  } catch (e) {
    console.log(e);
    // handle the chat error here...
  } finally {
    setIsSending(false);
```

```
  }
};

// ...
```

Give Chatterbox a run: try sending a message by drafting one with your `MessageBar` and tapping your `SendButton`. You should see your sent message rendered within the `MessageList` that you created earlier.

Deleting a Message

To delete a message from a chat room, you need to have the proper capability. Capabilities are granted during the initialization of the chat token that you use when authenticating to a chat room. For the purposes of this section, the `ServerApp` from Set Up a Local Authentication/ Authorization Server (in Part 1 of this tutorial) lets you specify moderator capabilities. This is done in your app using the `tokenProvider` object that you created in Build a Token Provider (also in Part 1).

Here you modify your `Message` by adding a function to delete the message.

First, open the `App.tsx` and add the `DELETE_MESSAGE` capability. (`capabilities` is the second parameter of your `tokenProvider` function.)

Note: This is how your `ServerApp` informs the IVS Chat APIs that the user being associated with the resulting chat token can delete messages in a chat room. In a real-world situation you probably will have more complex backend logic to manage user capabilities in your server app's infrastructure.

TypeScript/JavaScript:

```
// App.tsx / App.jsx

// ...

const [room] = useState(() =>
    new ChatRoom({
      regionOrUrl: process.env.REGION,
      tokenProvider: () => tokenProvider(userId, ['SEND_MESSAGE', 'DELETE_MESSAGE']),
    }),
);
```

```
// ...
```

In the next steps, you update your `Message` to display a delete button.

Define a new function called `onDelete` that accepts a string as one of its parameters and returns `Promise`. For the string parameter, pass in your component message ID.

TypeScript

```
// Message.tsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

export type Props = {
  message: ChatMessage;
  onDelete(id: string): Promise<void>;
};

export const Message = ({ message, onDelete }: Props) => {
  const userId = useUserContext();

  const isMine = message.sender.userId === userId;
  const handleDelete = () => onDelete(message.id);

  return (
    <View style={[styles.root, isMine && styles.mine]}>
      {!isMine && <Text>{message.sender.userId}</Text>}
      <View style={styles.content}>
        <Text style={styles.textContent}>{message.content}</Text>
        <TouchableOpacity onPress={handleDelete}>
          <Text>Delete<Text/>
        </TouchableOpacity>
      </View>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
```

```
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  content: {
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'space-between',
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
  mine: {
    flexDirection: 'row-reverse',
    backgroundColor: 'lightblue',
  },
});
```

JavaScript

```
// Message.jsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

export const Message = ({ message, onDelete }) => {
  const userId = useUserContext();

  const isMine = message.sender.userId === userId;
  const handleDelete = () => onDelete(message.id);

  return (
    <View style={[styles.root, isMine && styles.mine]}>
      {!isMine && <Text>{message.sender.userId}</Text>}
      <View style={styles.content}>
        <Text style={styles.textContent}>{message.content}</Text>
        <TouchableOpacity onPress={handleDelete}>
```

```
          <Text>Delete<Text/>
        </TouchableOpacity>
      </View>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  content: {
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'space-between',
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
  mine: {
    flexDirection: 'row-reverse',
    backgroundColor: 'lightblue',
  },
});
```

Next, you update your `renderItem` to reflect the latest changes to your `FlatList` component.

In App, define a function named `handleDeleteMessage` and pass it to the `MessageList` on`Delete` property:

TypeScript

```
// App.tsx

// ...
```

```
const handleDeleteMessage = async (id: string) => {};

const renderItem = useCallback<ListRenderItem<ChatMessage>>(({ item }) => {
  return (
    <Message key={item.id} message={item} onDelete={handleDeleteMessage} />
  );
}, [handleDeleteMessage]);

// ...
```

JavaScript

```
// App.jsx

// ...

const handleDeleteMessage = async (id) => {};

const renderItem = useCallback(({ item }) => {
  return (
    <Message key={item.id} message={item} onDelete={handleDeleteMessage} />
  );
}, [handleDeleteMessage]);

// ...
```

Prepare a request by creating a new instance of `DeleteMessageRequest`, passing the relevant message ID to the constructor parameter, and call `deleteMessage` that accepts the prepared request above:

TypeScript

```
// App.tsx

// ...

const handleDeleteMessage = async (id: string) => {
  const request = new DeleteMessageRequest(id);
  await room.deleteMessage(request);
};
```

```
// ...
```

JavaScript

```
// App.jsx

// ...

const handleDeleteMessage = async (id) => {
  const request = new DeleteMessageRequest(id);
  await room.deleteMessage(request);
};

// ...
```

Next, you update your `messages` state to reflect a new list of messages that omits the message you just deleted.

In the `useEffect` hook, listen for the `messageDelete` event and update your `messages` state array by deleting the message with a matching ID to the `message` parameter.

Note: The `messageDelete` event might be raised for messages being deleted by the current user or any other user in the room. Handling it in the event handler (instead of next to the `deleteMessage` request) allows you to unify delete-message handling.

TypeScript/JavaScript:

```
// App.tsx / App.jsx

// ...

const unsubscribeOnMessageDeleted = room.addListener('messageDelete',
 (deleteMessageEvent) => {
  setMessages((prev) => prev.filter((message) => message.id !==
 deleteMessageEvent.id));
});

return () => {
  // ...

  unsubscribeOnMessageDeleted();
```

```
};

// ...
```

You are now able to delete users from a chat room in your chat app.

Next Steps

As an experiment, try implementing other actions in a room like disconnecting another user.

IVS Chat Client Messaging SDK: React & React Native Best Practices

This document describes the most important practices of using the Amazon IVS Chat Messaging SDK for React and React Native. This information should enable you to build typical chat functionality inside a React app, and give you the background you need to dive deeper into the more advanced parts of the IVS Chat Messaging SDK.

Creating a ChatRoom Initializer Hook

The `ChatRoom` class contains core chat methods and listeners for managing connection state and listening for events like message received and message deleted. Here, we show how to properly store chat instances in a hook.

Implementation

TypeScript

```
// useChatRoom.ts

import React from 'react';
import { ChatRoom, ChatRoomConfig } from 'amazon-ivs-chat-messaging';

export const useChatRoom = (config: ChatRoomConfig) => {
  const [room] = React.useState(() => new ChatRoom(config));

  return { room };
};
```

JavaScript

```
import React from 'react';
import { ChatRoom } from 'amazon-ivs-chat-messaging';

export const useChatRoom = (config) => {
  const [room] = React.useState(() => new ChatRoom(config));

  return { room };
};
```

Note: We don't use the `dispatch` method from the `setState` hook, because you can't update configuration parameters on the fly. The SDK creates an instance once, and it is not possible to update the token provider.

Important: Use the `ChatRoom` initializer hook once to initialize a new chat-room instance.

Example

TypeScript/JavaScript:

```
// ...

const MyChatScreen = () => {
  const userId = 'Mike';
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(ROOM_ID, ['SEND_MESSAGE']),
  });

  const handleConnect = () => {
    room.connect();
  };

  // ...
};

// ...
```

Listening for Connection State

Optionally, you can subscribe to connection-state updates in your chat-room hook.

Implementation

TypeScript

```
// useChatRoom.ts

import React from 'react';
import { ChatRoom, ChatRoomConfig, ConnectionState } from 'amazon-ivs-chat-
messaging';

export const useChatRoom = (config: ChatRoomConfig) => {
  const [room] = useState(() => new ChatRoom(config));

  const [state, setState] = React.useState<ConnectionState>('disconnected');

  React.useEffect(() => {
    const unsubscribeOnConnecting = room.addListener('connecting', () => {
      setState('connecting');
    });

    const unsubscribeOnConnected = room.addListener('connect', () => {
      setState('connected');
    });

    const unsubscribeOnDisconnected = room.addListener('disconnect', () => {
      setState('disconnected');
    });

    return () => {
      unsubscribeOnConnecting();
      unsubscribeOnConnected();
      unsubscribeOnDisconnected();
    };
  }, []);

  return { room, state };
};
```

JavaScript

```
// useChatRoom.js

import React from 'react';
```

```
import { ChatRoom } from 'amazon-ivs-chat-messaging';

export const useChatRoom = (config) => {
  const [room] = useState(() => new ChatRoom(config));

  const [state, setState] = React.useState('disconnected');

  React.useEffect(() => {
    const unsubscribeOnConnecting = room.addListener('connecting', () => {
      setState('connecting');
    });

    const unsubscribeOnConnected = room.addListener('connect', () => {
      setState('connected');
    });

    const unsubscribeOnDisconnected = room.addListener('disconnect', () => {
      setState('disconnected');
    });

    return () => {
      unsubscribeOnConnecting();
      unsubscribeOnConnected();
      unsubscribeOnDisconnected();
    };
  }, []);

  return { room, state };
};
```

ChatRoom Instance Provider

To use the hook in other components (to avoid prop drilling), you can create a chat-room provider using React `context`.

Implementation

TypeScript

```
// ChatRoomContext.tsx

import React from 'react';
```

```
import { ChatRoom } from 'amazon-ivs-chat-messaging';

const ChatRoomContext = React.createContext<ChatRoom | undefined>(undefined);

export const useChatRoomContext = () => {
  const context = React.useContext(ChatRoomContext);

  if (context === undefined) {
    throw new Error('useChatRoomContext must be within ChatRoomProvider');
  }

  return context;
};

export const ChatRoomProvider = ChatRoomContext.Provider;
```

JavaScript

```
// ChatRoomContext.jsx

import React from 'react';
import { ChatRoom } from 'amazon-ivs-chat-messaging';

const ChatRoomContext = React.createContext(undefined);

export const useChatRoomContext = () => {
  const context = React.useContext(ChatRoomContext);

  if (context === undefined) {
    throw new Error('useChatRoomContext must be within ChatRoomProvider');
  }

  return context;
};

export const ChatRoomProvider = ChatRoomContext.Provider;
```

Example

After creating `ChatRoomProvider`, you can consume your instance with `useChatRoomContext`.

Important: Put the provider in the root level only if you need access to the `context` between the chat screen and the other components in the middle, to avoid unnecessary re-renders if you are listening for connections. Otherwise, put the provider as close as possible to the chat screen.

TypeScript/JavaScript:

```
// AppContainer

const AppContainer = () => {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(ROOM_ID, ['SEND_MESSAGE']),
  });

  return (
    <ChatRoomProvider value={room}>
      <MyChatScreen />
    </ChatRoomProvider>
  );
};

// MyChatScreen

const MyChatScreen = () => {
  const room = useChatRoomContext();

  const handleConnect = () => {
    room.connect();
  };
  // ...
};

// ...
```

Creating a Message Listener

To stay up to date with all incoming messages, you should subscribe to `message` and `deleteMessage` events. Here is some code that provides chat messages for your components.

Important: For performance purposes, we separate `ChatMessageContext` from `ChatRoomProvider`, as we may get many re-renders when the chat-message listener updates its

message's state. Remember to apply `ChatMessageContext` in components where you will use `ChatMessageProvider`.

Implementation

TypeScript

```
// ChatMessagesContext.tsx

import React from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useChatRoomContext } from './ChatRoomContext';

const ChatMessagesContext = React.createContext<ChatMessage[] |
 undefined>(undefined);

export const useChatMessagesContext = () => {
  const context = React.useContext(ChatMessagesContext);

  if (context === undefined) {
    throw new Error('useChatMessagesContext must be within ChatMessagesProvider);
  }

  return context;
};

export const ChatMessagesProvider = ({ children }: { children: React.ReactNode }) =>
 {
  const room = useChatRoomContext();

  const [messages, setMessages] = React.useState<ChatMessage[]>([]);

  React.useEffect(() => {
    const unsubscribeOnMessageReceived = room.addListener('message', (message) => {
      setMessages((msgs) => [message, ...msgs]);
    });

    const unsubscribeOnMessageDeleted = room.addListener('messageDelete',
 (deleteEvent) => {
      setMessages((prev) => prev.filter((message) => message.id !==
 deleteEvent.messageId));
    });
```

```
    return () => {
      unsubscribeOnMessageDeleted();
      unsubscribeOnMessageReceived();
    };
  }, [room]);

  return <ChatMessagesContext.Provider value={messages}>{children}</
ChatMessagesContext.Provider>;
};
```

JavaScript

```
// ChatMessagesContext.jsx

import React from 'react';
import { useChatRoomContext } from './ChatRoomContext';

const ChatMessagesContext = React.createContext(undefined);

export const useChatMessagesContext = () => {
  const context = React.useContext(ChatMessagesContext);

  if (context === undefined) {
    throw new Error('useChatMessagesContext must be within ChatMessagesProvider);
  }

  return context;
};

export const ChatMessagesProvider = ({ children }) => {
  const room = useChatRoomContext();

  const [messages, setMessages] = React.useState([]);

  React.useEffect(() => {
    const unsubscribeOnMessageReceived = room.addListener('message', (message) => {
      setMessages((msgs) => [message, ...msgs]);
    });

    const unsubscribeOnMessageDeleted = room.addListener('messageDelete',
 (deleteEvent) => {
      setMessages((prev) => prev.filter((message) => message.id !==
 deleteEvent.messageId));
```

```
    });

    return () => {
      unsubscribeOnMessageDeleted();
      unsubscribeOnMessageReceived();
    };
  }, [room]);

  return <ChatMessagesContext.Provider value={messages}>{children}</
ChatMessagesContext.Provider>;
};
```

Example in React

Important: Remember to wrap your message container with `ChatMessagesProvider`. The `Message` row is an example component that displays the content of a message.

TypeScript/JavaScript:

```
// your message list component...

import React from 'react';
import { useChatMessagesContext } from './ChatMessagesContext';

const MessageListContainer = () => {
  const messages = useChatMessagesContext();

  return (
    <React.Fragment>
      {messages.map((message) => (
        <MessageRow message={message} />
      ))}
    </React.Fragment>
  );
};
```

Example in React Native

By default, `ChatMessage` contains `id`, which is used automatically as React keys in `FlatList` for each row; therefore, you don't need to pass `keyExtractor`.

TypeScript

```
// MessageListContainer.tsx

import React from 'react';
import { ListRenderItemInfo, FlatList } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useChatMessagesContext } from './ChatMessagesContext';

const MessageListContainer = () => {
  const messages = useChatMessagesContext();

  const renderItem = useCallback(({ item }: ListRenderItemInfo<ChatMessage>) =>
 <MessageRow />, []);

  return <FlatList data={messages} renderItem={renderItem} />;
};
```

JavaScript

```
// MessageListContainer.jsx

import React from 'react';
import { FlatList } from 'react-native';
import { useChatMessagesContext } from './ChatMessagesContext';

const MessageListContainer = () => {
  const messages = useChatMessagesContext();

  const renderItem = useCallback(({ item }) => <MessageRow />, []);

  return <FlatList data={messages} renderItem={renderItem} />;
};
```

Multiple Chat Room Instances in an App

If you use multiple concurrent chat rooms in your app, we propose creating each provider for each chat and consuming it in the chat provider. In this example, we're creating a Help Bot and Customer Help chat. We create a provider for both.

TypeScript

```
// SupportChatProvider.tsx

import React from 'react';
import { SUPPORT_ROOM_ID, SOCKET_URL } from '../../config';
import { tokenProvider } from '../tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SupportChatProvider = ({ children }: { children: React.ReactNode }) =>
 {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SUPPORT_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};

// SalesChatProvider.tsx

import React from 'react';
import { SALES_ROOM_ID, SOCKET_URL } from '../../config';
import { tokenProvider } from '../tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SalesChatProvider = ({ children }: { children: React.ReactNode }) => {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SALES_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};
```

JavaScript

```
// SupportChatProvider.jsx

import React from 'react';
import { SUPPORT_ROOM_ID, SOCKET_URL } from '../../config';
```

```
import { tokenProvider } from '../tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SupportChatProvider = ({ children }) => {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SUPPORT_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};

// SalesChatProvider.jsx

import React from 'react';
import { SALES_ROOM_ID, SOCKET_URL } from '../../config';
import { tokenProvider } from '../tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SalesChatProvider = ({ children }) => {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SALES_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};
```

Example in React

Now you can use different chat providers that use the same `ChatRoomProvider`. Later on, you can reuse the same `useChatRoomContext` inside each screen/view.

TypeScript/JavaScript:

```
// App.tsx / App.jsx

const App = () => {
  return (
    <Routes>
```

```
      <Route
        element={
          <SupportChatProvider>
            <SupportChatScreen />
          </SupportChatProvider>
        }
      />
      <Route
        element={
          <SalesChatProvider>
            <SalesChatScreen />
          </SalesChatProvider>
        }
      />
    </Routes>
  );
};
```

Example in React Native

TypeScript/JavaScript:

```
// App.tsx / App.jsx

const App = () => {
  return (
    <Stack.Navigator>
      <Stack.Screen name="SupportChat">
        <SupportChatProvider>
          <SupportChatScreen />
        </SupportChatProvider>
      </Stack.Screen>
      <Stack.Screen name="SalesChat">
        <SalesChatProvider>
          <SalesChatScreen />
        </SalesChatProvider>
      </Stack.Screen>
    </Stack.Navigator>
  );
};
```

TypeScript/JavaScript:

```
// SupportChatScreen.tsx / SupportChatScreen.jsx

// ...

const SupportChatScreen = () => {
  const room = useChatRoomContext();

  const handleConnect = () => {
    room.connect();
  };

  return (
    <>
      <Button title="Connect" onPress={handleConnect} />
      <MessageListContainer />
    </>
  );
};

// SalesChatScreen.tsx / SalesChatScreen.jsx

// ...

const SalesChatScreen = () => {
  const room = useChatRoomContext();

  const handleConnect = () => {
    room.connect();
  };

  return (
    <>
      <Button title="Connect" onPress={handleConnect} />
      <MessageListContainer />
    </>
  );
};
```

Amazon IVS Chat Security

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** — AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs.
- **Security in the cloud** — Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Amazon IVS Chat. The following topics show you how to configure Amazon IVS Chat to meet your security and compliance objectives.

Topics

IVS Chat Data Protection

For data sent to Amazon Interactive Video Service (IVS) Chat, the following data protections are in place:

- Amazon IVS Chat traffic uses WSS to keep data secure in transit.
- Amazon IVS Chat tokens are encrypted using KMS customer-managed keys.

Amazon IVS Chat does not require that you supply any customer (end user) data. There are no fields in chat rooms, inputs, or input security groups where there is an expectation that you will provide customer (end user) data.

Do not put sensitive identifying information such as your customer (end user) account numbers into free-form fields such as a Name field. This includes when you work with the Amazon IVS console or API, AWS CLI, or AWS SDKs. Any piece of data that you enter into Amazon IVS Chat might be included in diagnostic logs.

Streams are not end-to-end encrypted; a stream may be transmitted unencrypted internally in the IVS network, for processing.

Identity and Access Management in IVS Chat

AWS Identity and Access Management (IAM) is an AWS service that helps an account administrator securely control access to AWS resources. See Identity and Access Management in IVS in the *IVS Low-Latency Streaming User Guide*.

Audience

How you use IAM differs, depending on the work you do in Amazon IVS. See Audience in the *IVS Low-Latency Streaming User Guide*.

How Amazon IVS Works with IAM

Before you can make Amazon IVS API requests, you must create one or more IAM *identities* (users, groups, and roles) and IAM *policies*, then attach policies to identities. It takes up to a few minutes for the permissions to propagate; until then, API requests are rejected.

For a high-level view of how Amazon IVS works with IAM, see AWS Services That Work with IAM in the *IAM User Guide*.

Identities

You can create IAM identities to provide authentication for people and processes in your AWS account. IAM groups are collections of IAM users that you can manage as a unit. See Identities (Users, Groups, and Roles) in the *IAM User Guide*.

Policies

Policies are JSON permissions-policy documents made up of *elements*. See Policies in the *IVS Low-Latency Streaming User Guide*.

Amazon IVS Chat supports three elements:

- **Actions** — Policy actions for Amazon IVS Chat use the `ivschat` prefix before the action. For example, to grant someone permission to create an Amazon IVS Chat room with the Amazon IVS Chat `CreateRoom` API method, you include the `ivschat:CreateRoom` action in the policy for that person. Policy statements must include either an `Action` or `NotAction` element.
- **Resources** — The Amazon IVS Chat room resource has the following ARN format:

  ```
  arn:aws:ivschat:${Region}:${Account}:room/${roomId}
  ```

  For example, to specify the `VgNkEJgOVX9N` room in your statement, use this ARN:

  ```
  "Resource": "arn:aws:ivschat:us-west-2:123456789012:room/VgNkEJgOVX9N"
  ```

  Some Amazon IVS Chat actions, such as those for creating resources, cannot be performed on a specific resource. In those cases, you must use the wildcard (*):

  ```
  "Resource":"*"
  ```

- **Conditions** — Amazon IVS Chat supports some global condition keys: `aws:RequestTag`, `aws:TagKeys`, and `aws:ResourceTag`.

You can use variables as placeholders in a policy. For example, you can grant an IAM user permission to access a resource only if it is tagged with the user's IAM username. See Variables and Tags in the *IAM User Guide*.

Amazon IVS provides AWS managed policies that can be used to grant a preconfigured set of permissions to identities (read only or full access). You can choose to use managed policies instead of the identity-based policies shown below. For details, see Managed Policies for Amazon IVS Chat.

Authorization Based on Amazon IVS Tags

You can attach tags to Amazon IVS Chat resources or pass tags in a request to Amazon IVS Chat. To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys. For more information about tagging Amazon IVS Chat resources, see "Tagging" in the IVS Chat API Reference.

Roles

See IAM Roles and Temporary Security Credentials in the *IAM User Guide*.

An IAM *role* is an entity within your AWS account that has specific permissions.

Amazon IVS supports using *temporary security credentials*. You can use temporary credentials to sign in with federation, assume an IAM role, or assume a cross-account role. You obtain temporary security credentials by calling AWS Security Token Service API operations such as `AssumeRole` or `GetFederationToken`.

Privileged and Unprivileged Access

API resources have privileged access. Unprivileged playback access can be set up through private channels; see Setting Up IVS Private Channels.

Best Practices for Policies

See IAM Best Practices in the *IAM User Guide*.

Identity-based policies are very powerful. They determine whether someone can create, access, or delete Amazon IVS resources in your account. These actions can incur costs for your AWS account. Follow these recommendations:

- **Grant least privilege** — When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant more permissions as needed. Doing so is more secure than starting with permissions that are too lenient, then

trying to tighten them later. Specifically, reserve `ivschat:*` for admin access; do not use it in applications.

- **Enable multi-factor authentication (MFA) for sensitive operations** — For extra security, require IAM users to use MFA to access sensitive resources or API operations.
- **Use policy conditions for extra security** — To the extent practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses from which a request must come. You also can write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA.

Identity-Based Policy Examples

Use the Amazon IVS Console

To access the Amazon IVS console, you must have a minimum set of permissions which allow you to list and view details about the Amazon IVS Chat resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console will not function as intended for identities with that policy. To ensure access to the Amazon IVS console, attach the following policy to the identities (see Adding and Removing IAM Permissions in the *IAM User Guide*).

The parts of the following policy provide access to:

- All Amazon IVS Chat API operations
- Your Amazon IVS Chat service quotas
- Listing lambdas and adding permissions for the chosen lambda for Amazon IVS Chat moderation
- Amazon Cloudwatch to get metrics for your chat session

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "ivschat:*",
      "Effect": "Allow",
      "Resource": "*"
    },
    {
```

```
      "Action": [
        "servicequotas:ListServiceQuotas"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Action": [
        "cloudwatch:GetMetricData"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Action": [
        "lambda:AddPermission",
        "lambda:ListFunctions"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

Resource-Based Policy for Amazon IVS Chat

You must give the Amazon IVS Chat service permission to invoke your lambda resource to review messages. To do that, follow the instructions in Using resource-based policies for AWS Lambda (in the *AWS Lambda Developer Guide*) and fill out the fields as specified below.

To control access to your lambda resource, you can use conditions based on:

- `SourceArn` — Our sample policy uses a wildcard (`*`) to allow all rooms in your account to invoke the lambda. Optionally, you can specify a room in your account to allow only that room to invoke the lambda.
- `SourceAccount` — In the sample policy below, the AWS account ID is `123456789012`.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
```

```
         "Principal": {
            "Service": "ivschat.amazonaws.com"
         },
         "Action": [
            "lambda:InvokeFunction"
         ],
         "Effect": "Allow",
         "Resource": "arn:aws:lambda:us-west-2:123456789012:function:name",
         "Condition": {
            "StringEquals": {
               "AWS:SourceAccount": "123456789012"
            },
            "ArnLike": {
               "AWS:SourceArn": "arn:aws:ivschat:us-west-2:123456789012:room/*"
            }
         }
      }
   ]
}
```

Troubleshooting

See Troubleshooting in the *IVS Low-Latency Streaming User Guide* for information about diagnosing and fixing common issues that you might encounter when working with Amazon IVS Chat and IAM.

Managed Policies for IVS Chat

An AWS managed policy is a standalone policy that is created and administered by AWS. See Managed Policies for Amazon IVS in the *IVS Low-Latency Streaming User Guide*.

Using Service-Linked Roles for IVS Chat

Amazon IVS uses AWS IAM service-linked roles. See Using Service-Linked Roles for Amazon IVS in the *IVS Low-Latency Streaming User Guide*.

IVS Chat Logging and Monitoring

To log performance and/or operations, use Amazon CloudTrail. See Logging Amazon IVS API Calls with AWS CloudTrail in the *IVS Low-Latency Streaming User Guide* .

IVS Chat Incident Response

To detect or alert for incidents, you can monitor your stream's health via Amazon EventBridge events. See Using Amazon EventBridge with Amazon IVS: for Low-Latency Streaming and for Real-Time Streaming.

Use the AWS Health Dashboard for information on the overall health of Amazon IVS (by region).

IVS Chat Resilience

IVS APIs use the AWS global infrastructure and is built around AWS Regions and Availability Zones. See IVS Resilience in the *IVS Low-Latency Streaming User Guide*.

IVS Chat Infrastructure Security

As a managed service, Amazon IVS is protected by the AWS global network security procedures. These are described in Best Practices for Security, Identity, & Compliance.

API Calls

You use AWS published API calls to access Amazon IVS through the network. See API Calls under Infrastructure Security in the *IVS Low-Latency Streaming User Guide*.

Amazon IVS Chat

Amazon IVS Chat message ingestion and delivery occurs over encrypted WSS connections to our edge. The Amazon IVS Messaging API uses encrypted HTTPS connections. As with video streaming and playback, TLS version 1.2 or later is required and messaging data may be transmitted unencrypted internally for processing.

IVS Chat Service Quotas

The following are service quotas and limits for Amazon Interactive Video Service (IVS) chat 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

Operation Type	Operation	Default
Messaging	`DeleteMessage`	100 TPS
Messaging	`DisconnectUser`	100 TPS
Messaging	`SendEvent`	100 TPS
Chat token	`CreateChatToken`	200 TPS
Logging Configuration	`CreateLoggingConfiguration`	3 TPS
Logging Configuration	`DeleteLoggingConfiguration`	3 TPS
Logging Configuration	`GetLoggingConfiguration`	3 TPS
Logging Configuration	`ListLoggingConfigurations`	3 TPS

Operation Type	Operation	Default
Logging Configuration	`UpdateLoggingConfiguration`	3 TPS
Room	`CreateRoom`	5 TPS
Room	`DeleteRoom`	5 TPS
Room	`GetRoom`	5 TPS
Room	`ListRooms`	5 TPS
Room	`UpdateRoom`	5 TPS
Tags	`ListTagsForResource`	10 TPS
Tags	`TagResource`	10 TPS
Tags	`UntagResource`	10 TPS

Other Quotas

Resource or Feature	Default	Adjustable	Description
Concurrent chat connections	50,000	Yes	Maximum number of concurrent chat connections per account, across all your rooms in an AWS Region.
Logging configurations	10	Yes	The maximum number of logging configurations that can be created per account in the current AWS Region.
Message review handler timeout period	200	No	Timeout period in milliseconds for all your message review handlers in the current AWS Region.

Resource or Feature	Default	Adjustable	Description
			If this is exceeded, the message is allowed or denied depending on the value of the `fallbackResult` field you configured for the message review handler.
Rate of DeleteMessage requests across all your rooms	100	Yes	Maximum number of DeleteMessage requests that can be made per second across all your rooms. The requests can come from either the Amazon IVS Chat API or the Amazon IVS Chat Messaging API (WebSocket).
Rate of DisconnectUser requests across all your rooms	100	Yes	Maximum number of DisconnectUser requests that can be made per second across all your rooms. The requests can come from either the Amazon IVS Chat API or the Amazon IVS Chat Messaging API (WebSocket).
Rate of messaging requests per connection	10	No	Maximum number of messaging requests per second that a chat connection can make.

Resource or Feature	Default	Adjustable	Description
Rate of SendMessage requests across all your rooms	1000	Yes	Maximum number of SendMessage requests that can be made per second across all your rooms. These requests come from the Amazon IVS Chat Messaging API (WebSocket).
Rate of SendMessage requests per room	100	No (but configurable through the API)	Maximum number of SendMessage requests that can be made per second for any one of your rooms. This is configurable with the `maximumMessageRatePerSecond` field of CreateRoom and UpdateRoom. These requests come from the Amazon IVS Chat Messaging API (WebSocket).
Rooms	50,000	Yes	Maximum number of chat rooms per account, per AWS Region.

Service Quotas Integration with CloudWatch Usage Metrics

You can use CloudWatch to proactively manage your service quotas, via CloudWatch *usage metrics*. You can use these metrics to visualize your current service usage on CloudWatch graphs and dashboards. Amazon IVS Chat usage metrics correspond to Amazon IVS Chat service quotas.

You can use a CloudWatch metric math function to display the service quotas for those resources on your graphs. You can also configure alarms that alert you when your usage approaches a service quota.

To access usage metrics:

1. Open the Service Quotas console at https://console.aws.amazon.com/servicequotas/
2. In the navigation pane, select **AWS services**.
3. From the AWS services list, search for and select **Amazon Interactive Video Service Chat**.
4. In the **Service quotas** list, select the service quota of interest. A new page opens with information about the service quota/metric.

Alternately, you can get to these metrics through the CloudWatch console. Under **AWS Namespaces**, choose **Usage**. Then, from the **Service** list, choose **IVS Chat**. (See Monitoring Amazon IVS Chat.)

In the **AWS/Usage** namespace, Amazon IVS Chat provides the following metric:

Metric Name	Description
`ResourceCount`	A count of the specified resources running in your account. The resources are defined by the dimensions associated with the metric. Valid statistic: Maximum (the maximum number of resources used during the 1-minute period).

The following dimensions are used to refine the usage metric:

Dimension	Description
Service	The name of the AWS service containing the resource. Valid value: `IVS Chat`.
Class	The class of resource being tracked. Valid value: `None`.
Type	The type of resource being tracked. Valid value: `Resource`.
Resource	The name of the AWS resource. Valid value: `ConcurrentChatConnections` . The ConcurrentChatConnections usage metric is a copy of the one in the **AWS/IVSChat** namespace (with the None dimension), as described in Monitoring Amazon IVS Chat.

Creating a CloudWatch Alarm for Usage Metrics

To create a CloudWatch alarm based on an Amazon IVS Chat usage metric:

1. From the Service Quotas console, select the service quota of interest, as described above. Currently, alarms can be created only for ConcurrentChatConnections.
2. In the **Amazon CloudWatch alarms** section, choose **Create**.
3. From the **Alarm threshold** dropdown list, choose the percentage of your applied quota value that you want to set as the alarm value.
4. For **Alarm name**, enter a name for the alarm.
5. Select **Create**.

Troubleshooting IVS Chat

This document describes best practices and troubleshooting tips for Amazon Interactive Video Service (IVS) Chat. Behaviors related to IVS Chat often are distinct from behaviors related to IVS video. For more information, see Getting Started with Amazon IVS Chat.

Topics:

Why were IVS chat connections not disconnected when the room was deleted?

When a chat-room resource is deleted, if the room is actively being used, the chat clients that are connected to the room are not automatically disconnected. The connection is dropped if/when the chat application refreshes the chat token. Alternately, a manual disconnect of all users must be done to remove all users from the chat room.

IVS 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 composition 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 operational 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 user who creates a 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-coded 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 Optimizations*.		✓	

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 management of on-screen video and audio elements representing 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, competitions 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 restricting 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 publisher or subscriber.		✓	
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 an IVS Playback Key in *Setting up IVS Private Channels* and see the Playback Key Pair operations 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 IVS Private Channels in *Setting up IVS 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.	✓	✓	
Publisher	A real-time event participant who publishes video and/or audio to a stage. See What is IVS Real-Time Streaming.		✓	
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.	✓	✓	✓

Term	Description	LL	RT	Chat
Resolution	Describes the number of pixels in a single video frame, for example, Full HD or 1080p defines a frame with 1920x1080 pixels.	✓	✓	
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)*.		✓	

Term	Description	LL	RT	Chat
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.	✓	✓	✓
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 composition 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 automatically created by IVS and include all the permissions 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 with Optional Participant Recording in *Getting Started with IVS Real-Time Streaming*.		✓	

Term	Description	LL	RT	Chat
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.	✓	✓	
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 stage publishers. See What is IVS Real-Time Streaming.		✓	

Term	Description	LL	RT	Chat
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-latency streaming, or 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 configured. 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.	✓		

Term	Description	LL	RT	Chat
Transcoding	Converts video and audio from one format to another. An incoming stream may be transcoded to a different format at multiple bitrates and resolutions, to support a range of playback devices and network conditions.	✓	✓	
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 IVS WHIP Support (Real-Time Streaming).		✓	
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*.			✓

IVS Chat Document History

The following tables describe the important changes to the documentation for Amazon IVS Chat. We update the documentation frequently, for new releases and to address the feedback that you send us.

Chat User Guide Changes

Change	Description	Date
Split out a Chat UG	Major documentation changes accompany this release. We moved chat information from the IVS Low-Latency Streaming User Guide to a new IVS Chat User Guide, located in the existing IVS Chat section of the IVS documentation landing page. For other documentation changes, see Document History (Low-Latency Streaming).	December 28, 2023
IVS Glossary	Extended the glossary, covering IVS real-time, low-latency, and chat terms.	December 20, 2023

IVS Chat API Reference Changes

API Change	Description	Date
Split out a Chat UG	Now that there is an IVS Chat User Guide (created in this release), the Document History entries for	Dec 28, 2023

API Change	Description	Date
	the existing IVS Chat API Reference and IVS Chat Messaging API Reference will be located here, moving forward. Prior history entries for those Chat API References are in Document History (Low-Latency Streaming).	

IVS Chat Release Notes

This document contains all Amazon IVS Chat release notes, latest first, organized by date of release.

December 28, 2023

Amazon IVS Chat User Guide

Amazon Interactive Video Service (IVS) Chat is a managed, live-chat feature to go alongside live video streams. In this release, we moved chat information from the IVS Low-Latency Streaming User Guide to a new IVS Chat User Guide. Documentation is accessible from the Amazon IVS documentation landing page.

January 31, 2023

Amazon IVS Chat Client Messaging SDK: Android 1.1.0

Platform	Downloads and Changes
Android Chat Client Messaging SDK 1.1.0	**Reference documentation:** https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/ • To support Kotlin Coroutines, we added new IVS Chat Messaging APIs in the com.amazonaws.ivs.chat.messaging.coroutines package. Also see the new Kotlin Coroutines tutorial; part 1 (of 2) is Chat Rooms.

Chat Client Messaging SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
All architectures (bytecode)	89 KB	92 KB

November 9, 2022

Amazon IVS Chat Client Messaging SDK: JavaScript 1.0.2

Platform	Downloads and Changes
JavaScript Chat Client Messaging SDK 1.0.2	**Reference documentation:** https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/ • Fixed an issue that affected Firefox: clients erroneously received a socket error when they were disconnected from a chat room using the DisconnectUser endpoint.

September 8, 2022

Amazon IVS Chat Client Messaging SDK: Android 1.0.0 and iOS 1.0.0

Platform	Downloads and Changes
Android Chat Client Messaging SDK 1.0.0	**Reference documentation:** https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/
iOS Chat Client Messaging SDK 1.0.0	**Reference documentation:** https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/

Chat Client Messaging SDK Size: Android

Architecture	Compressed Size	Uncompressed Size
All architectures (bytecode)	53 KB	58 KB

Chat Client Messaging SDK Size: iOS

Architecture	Compressed Size	Uncompressed Size
ios-arm64_x86_64-simulator (bitcode)	484 KB	2.4 MB
ios-arm64_x86_64-simulator	484 KB	2.4 MB
ios-arm64 (bitcode)	1.1 MB	3.1 MB
ios-arm64	233 KB	1.2 MB