aws

聊天功能用户指南

Amazon IVS

Amazon IVS: 聊天功能用户指南

Table of Contents

什么是 Amazon IVS 聊天功能？

Amazon IVS 聊天功能是一项托管的实时聊天功能，可与实时视频流一起使用。可从 Amazon IVS 聊天功能部分的 Amazon IVS 文档登陆页面访问文档：

- 聊天功能用户指南：此文档以及导航窗格中列出的所有其他用户指南页面。
- Chat API 参考：控制面板 API（HTTPS）。
- Chat Messaging API 参考：数据面板 API（WebSocket）。
- 聊天客户端的 SDK 参考：Android、iOS 和 JavaScript。

Amazon IVS Chat 入门

Amazon Interactive Video Service（IVS）Chat 是一项托管的实时聊天功能，可与您的实时视频流一起使用。（IVS 聊天功能也可以在没有视频流的情况下使用。）您可以创建聊天室并启用用户之间的聊天会话。

Amazon IVS Chat 让您可以专注于构建自定义聊天体验以及实时视频。您无需管理基础设施或开发和配置聊天工作流的组件。Amazon IVS Chat 具有可扩展性、安全性、可靠性及成本效益。

Amazon IVS Chat 最适合促进实时视频流（具有开头和结尾）参与者之间的消息收发。

本文档的其余部分将指导您完成使用 Amazon IVS Chat 构建第一个聊天应用程序的步骤。

示例：以下演示应用程序均可用（三个示例客户端应用程序和一个用于创建令牌的后端服务器应用程序）：

- Amazon IVS Chat Web 演示
- 适用于 Android 的 Amazon IVS Chat 演示
- 适用于 iOS 的 Amazon IVS Chat 演示
- Amazon IVS Chat 演示后端

重要提示：24 个月没有新连接或更新的聊天室将被自动删除。

主题

步骤 1：进行初始设置

在继续之前，您必须先执行以下操作：

1. 创建亚马逊云科技账户。
2. 设置根用户和管理用户。

3. 设置 Amazon IAM (Identity and Access Management) 权限。使用下面指定的策略。

有关所有上述操作的具体步骤，请参阅 Amazon IVS User Guide 中的 Getting Started with IVS Low-Latency Streaming。重要提示：在“步骤 3：设置 IAM 权限”中，请将此策略用于 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": "*"
      }
   ]
}
```

步骤 2：创建聊天室

Amazon IVS 聊天室具有与之关联的配置信息（例如，最大消息长度）。

本节中的说明向您展示如何使用控制台或 AWS CLI 设置聊天室（包括用于查看消息和/或记录消息的可选设置）和创建聊天室。

创建 IVS 聊天室的控制台说明

这些步骤分为几个阶段，第一阶段是初始房间设置，最后阶段是最终房间创建。

或者，也可以设置一个房间，以便审核消息。例如，您可以更新消息内容或元数据、拒绝消息以阻止发送消息，或者允许原始消息通过。设置以审核房间消息（可选）中介绍了此过程。

或者，也可以设置一个房间，以便记录消息。例如，如果您正在发送消息到聊天室，您可以将消息记录到 Amazon S3 存储桶、Amazon CloudWatch 或 Amazon Kinesis Data Firehose 中。设置以记录消息（可选）中介绍了此过程。

初始房间设置

1. 打开 Amazon IVS Chat 控制台。

   （您还可通过亚马逊云科技管理控制台访问 Amazon IVS 控制台。）

2. 从导航栏中，使用 Select a Region（选择区域）下拉菜单选择一个区域。您的新房间将在此区域创建。

3. 在 Get started（试用）框（右上角）中，选择 Amazon IVS Chat Room（Amazon IVS 聊天室）。此时会显示 Create room（创建房间）窗口。

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. 在 Setup（设置）下，可以选择指定一个 Room name（房间名称）。房间名称不是唯一的，但它们为您提供了一种方法来区分房间而不是房间 ARN（Amazon 资源名称）。
5. 在 Setup > Room configuration（设置 > 房间配置）下，选择接受 Default configuration（默认配置），或选择 Custom configuration（自定义配置），然后配置 Maximum message length（最大消息长度）和/或 Maximum message rate（最大消息速率）。
6. 若要审核消息，请继续执行下面的 Set Up to Review Room Messages（Optional）[设置以审核房间消息（可选）]。否则，跳过该步骤 [即接受 Message Review Handler > Disabled（消息审核处理程序 > 禁用）]，然后直接前往 Final Room Creation（最终房间创建）。

设置以审核房间消息（可选）

1. 在 Message Review Handler（消息审核处理程序）下，选择 Handle with AWS Lambda（使用 Amazon Lambda 处理）。Message Review Handler（消息审核处理程序）部分将展开以显示其他选项。
2. 如果处理程序未返回有效响应、遇到错误或超时，则会将 Fallback result（回退结果）配置为 Allow（允许）或 Deny（拒绝）消息。
3. 指定现有的 Lambda function（Lambda 函数）或使用 Create Lambda function（创建 Lambda 函数）来创建新函数。

   Lambda 函数必须与聊天室位于同一 Amazon 账户和同一 Amazon 区域中。您应该向 Amazon Chat SDK 服务授予调用 Lambda 资源的权限。将为您选择的 Lambda 函数自动创建基于资源的策略。有关权限的更多信息，请参阅 Amazon IVS 聊天功能基于资源的策略。

设置以记录消息（可选）

1. 在 Message logging（消息日志记录）下，选择 Automatically log chat messages（自动记录聊天消息）。Message logging（消息日志记录）部分将展开以显示其他选项。您可以向此房间添加现有的日志记录配置，也可以通过选择 Create logging configuration（创建日志记录配置）来创建新的日志记录配置。
2. 如果您选择现有的日志记录配置，则会出现一个下拉菜单并显示您已经创建的所有日志记录配置。从列表中选择一个日志记录配置，您的聊天消息将自动记录到该目标配置。
3. 如果您选择 Create logging configuration（创建日志记录配置），则会出现一个模态窗口，该窗口允许您创建和自定义新的日志记录配置。
   a. 可以选择指定 Logging configuration name（日志记录配置名称）。日志记录配置名称（如房间名称）不是唯一的，但它们为您提供了一种方法来区分日志记录配置而不是日志记录配置 ARN。

b. 在 Destination（目标）下，选择 CloudWatch log group（CloudWatch 日志组）、Kinesis firehose delivery stream（Kinesis firehose 传输流）或 Amazon S3 bucket（Amazon S3 存储桶）作为日志的目标配置。

c. 根据您的目标配置，选择创建新的或使用现有的 CloudWatch log group（CloudWatch 日志组）、Kinesis firehose delivery stream（Kinesis firehose 传输流）或 Amazon S3 bucket（Amazon S3 存储桶）的选项。

d. 审核完成后，选择 Create（创建）以创建具有唯一 ARN 的新日志记录配置。这会自动将新的日志记录配置附加到聊天室。

最终房间创建

1. 审核完成后，选择 Create chat room（创建聊天室）以创建具有唯一 ARN 的新聊天室。

创建 IVS 聊天室的 CLI 说明

本文档将引导您完成使用 AWS CLI 创建 Amazon IVS 聊天室所涉及的步骤。

创建聊天室

使用 Amazon CLI 创建聊天室是一个高级选项，需要先在计算机上下载并配置 CLI。有关详细信息，请参阅 Amazon 命令行界面用户指南。

1. 运行聊天 `create-room` 命令并传入一个可选名称：

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

2. 这将返回一个新的聊天室：

```
{
   "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. 请注意 arn 字段。您需要它来创建客户端令牌并连接到聊天室。

设置日志记录配置（可选）

与创建聊天室一样，使用 Amazon CLI 设置日志记录配置是一个高级选项，需要先在计算机上下载并配置 CLI。有关详细信息，请参阅 Amazon 命令行界面用户指南。

1. 运行聊天 create-logging-configuration 命令并传入可选名称和按名称指向 Amazon S3 存储桶的目标配置。在创建日志记录配置之前，此 Amazon S3 存储桶必须存在。（有关创建 Amazon S3 存储桶的详细信息，请参阅 Amazon S3 文档。）

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

2. 这将返回一个新的日志记录配置：

```
{
   "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. 请注意 arn 字段。您需要此字段来将日志记录配置附加到聊天室。

   a. 如果您要创建新的聊天室，请运行 create-room 命令并传递日志记录配置 arn：

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

   b. 如果您要更新现有的聊天室，请运行 update-room 命令并传递日志记录配置 arn：

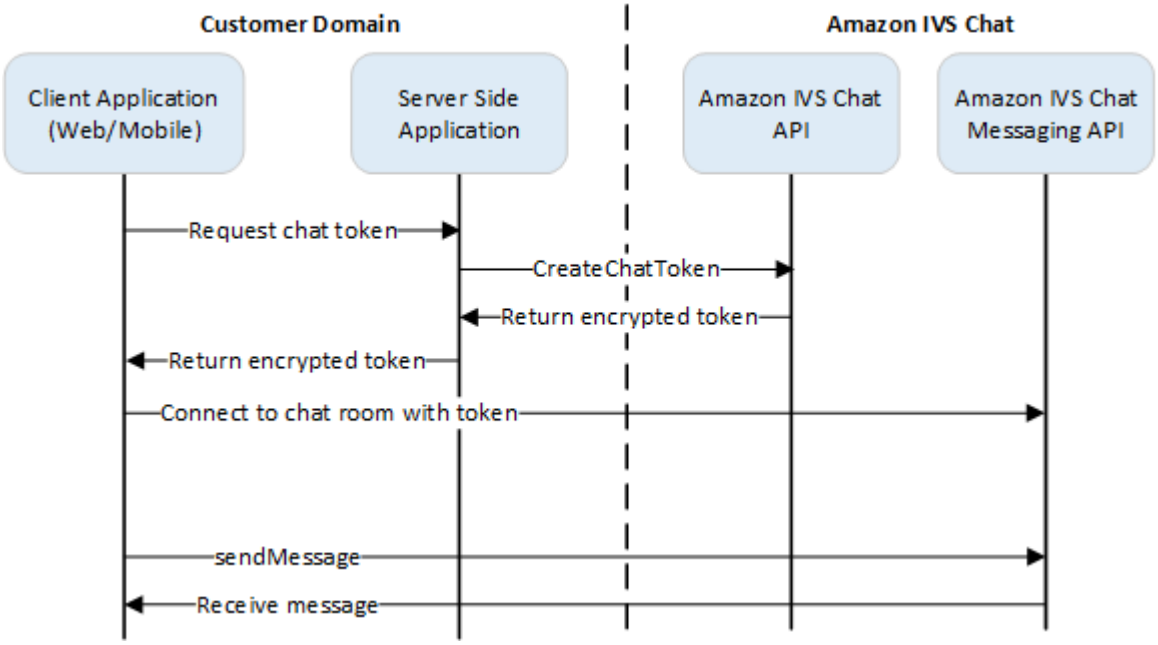
```
aws ivschat update-room --identifier \
```

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

步骤 3：创建聊天令牌

若聊天参与者要连接到房间并开始发送和接收消息，则必须创建聊天令牌。聊天令牌用于验证并授权聊天客户端。

此图表说明创建 IVS 聊天令牌的工作流程：



如上所示，客户端应用程序要求服务器端应用程序提供令牌，服务器端应用程序使用 AWS SDK 或 SigV4 签名请求调用 CreateChatToken。由于 AWS 凭证用于调用 API，因此应在安全的服务器端应用程序中生成令牌，而不是在客户端应用程序中。

Amazon IVS Chat 演示后端提供了一个演示令牌生成的后端服务器应用程序。

会话持续时间指已建立的会话在自动关闭之前可以保持活动的时间。换言之，会话持续时间是指在生成新令牌和建立新连接之前，客户端可以保持连接到聊天室的时间。您可以选择在创建令牌期间指定会话持续时间。

每个令牌只能用于为一位终端用户创建一次连接。如果连接已关闭，则必须先创建新令牌，然后才能重新建立连接。在响应中包含令牌到期时间戳之前，令牌本身一直有效。

终端用户想要连接到聊天室时，客户端应要求服务器应用程序提供令牌。服务器应用程序将创建令牌并将其传回客户端。应按需为终端用户创建令牌。

若要创建聊天验证令牌，请按照以下说明操作。创建聊天令牌时，使用请求字段传递聊天终端用户和终端用户消息传递功能的相关数据；有关详细信息，请参阅 IVS Chat API Reference 中的 CreateChatToken。

Amazon SDK 说明

使用 Amazon SDK 创建聊天令牌需要您先在应用程序上下载并配置 SDK。以下是使用 JavaScript 的 Amazon SDK 的说明。

重要提示：此代码必须在服务器端执行，并将其输出内容传递给客户端。

先决条件：要使用以下代码示例，您需要将 Amazon JavaScript SDK 加载到应用程序中。有关详细信息，请参阅 Amazon SDK for JavaScript 入门。

```
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 说明

使用 Amazon CLI 创建聊天令牌是一个高级选项，需要先在计算机上下载并配置 CLI。有关详细信息，请参阅 Amazon 命令行界面用户指南。注意：使用 Amazon CLI 生成令牌适合用于测试目的，但对于生产用途而言，我们建议您使用 Amazon SDK 在服务器端生成令牌（请参阅上述说明）。

1. 运行 create-chat-token 命令以及客户端的房间标识符和用户 ID。包括以下任何功能："SEND_MESSAGE"、"DELETE_MESSAGE"、"DISCONNECT_USER"。[可选，包括会话持续时间（以分钟为单位）和/或有关此聊天会话的自定义属性（元数据）。这些字段未在下方显示。]

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

2. 这将返回客户端令牌：

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

3. 保存该令牌。您需要它连接到聊天室并发送或接收消息。在会话结束之前，您需要生成另一个聊天令牌（如 sessionExpirationTime 所示）。

步骤 4：发送和接收第一条消息

使用聊天令牌连接到聊天室并发送第一条消息。以下是 JavaScript 示例代码。此外，还提供了 IVS 客户端 SDK：请参阅 Chat SDK: Android Guide、Chat SDK: iOS Guide 和 Chat SDK: JavaScript Guide。

区域服务：以下示例代码是指您的“受支持的选择区域”。Amazon IVS Chat 提供了可用于发出请求的区域端点。对于 Amazon IVS Chat Messaging API，区域端点的一般语法为：

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

例如，美国西部（俄勒冈州）区域的端点为 wss://edge.ivschat.us-west-2.amazonaws.com。有关受支持区域的列表，请参阅《AWS 一般参考》中 Amazon IVS 页面上的 Amazon IVS 聊天功能信息。

```
/*
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);
}
```

恭喜您，一切准备就绪！现在，您已拥有一个简单的聊天应用程序，可以发送或接收消息。

步骤 5：查看您的服务限额限制（可选）

您的聊天室将与 Amazon IVS 实时流一起扩缩，便于所有查看者参与聊天对话。但是，所有 Amazon IVS 账户对于同时参与聊天的人数和消息传送速率均有限制。

确保您的限制足够，并在需要时请求增加限额，尤其是在您计划大型流传输活动时。有关详细信息，请参阅 Service Quotas (Low-Latency Streaming)、Service Quotas (Real-Time Streaming) 和服务限额（聊天功能）。

IVS 聊天功能日志记录

聊天记录功能允许您将房间中的所有消息记录到三个标准位置中的任何一个：Amazon S3 存储桶、Amazon CloudWatch Logs 或 Amazon Kinesis Data Firehose。随后，这些日志可用于分析或构建链接到实时视频会话的聊天回放。

为某个房间启用聊天记录

聊天记录是一个高级选项，可以通过将日志记录配置与某个房间相关联来启用。日志记录配置是一种资源，允许您指定记录房间消息的位置类型（Amazon S3 存储桶、Amazon CloudWatch Logs 或 Amazon Kinesis Data Firehose）。有关创建和管理日志记录配置的详细信息，请参阅 Amazon IVS Chat 入门和 Amazon IVS Chat API 参考。

在创建新房间 (CreateRoom) 或更新现有房间 (UpdateRoom) 时，您最多可以将三个日志记录配置与每个房间相关联。您可以将多个房间与同一个日志记录配置相关联。

当至少一个处于活动状态的日志记录配置与某个房间相关联时，通过 Amazon IVS Chat 消息收发 API 发送到该房间的每个消息请求都会自动记录到指定位置。以下是平均传播延迟（从发送消息请求到消息在指定位置可用时）：

- Amazon S3 存储桶：5 分钟
- Amazon CloudWatch Logs 或 Amazon Kinesis Data Firehose：10 秒

消息内容

格式

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

字段

字段	描述
event_timestamp	Amazon IVS Chat 收到消息时的 UTC 时间戳。
payload	客户端将从 Amazon IVS Chat 服务接收的消息（订阅）或事件（订阅）JSON 负载。
type	聊天消息的类型。 • 有效值：MESSAGE | EVENT
version	消息内容格式的版本。

Amazon S3 存储桶

格式

消息日志使用以下 S3 前缀和文件格式进行组织和存储：

```
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
```

字段

字段	描述
<account_id>	创建房间的亚马逊云科技账户 ID。
<hash>	系统为确保唯一性而生成的哈希值。
<region>	创建房间的亚马逊云科技服务区域。
<resource_id>	房间 ARN 的资源 ID 部分。
<version>	消息内容格式的版本。

字段	描述
<year> / <month> / <day> / <hours> / <minute>	Amazon IVS Chat 收到消息时的 UTC 时间戳。

示例

```
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

格式

消息日志使用以下日志流名称格式进行组织和存储：

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

字段

字段	描述
<resource_id>	房间 ARN 的资源 ID 部分。
<version>	消息内容格式的版本。

示例

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

Amazon Kinesis Data Firehose

消息日志将发送到传输流，作为实时流数据传输到目标位置，例如 Amazon Redshift、Amazon OpenSearch Service、Splunk，以及任何自定义 HTTP 端点或所支持的第三方服务提供商拥有的 HTTP 端点。有关更多信息，请参阅什么是 Amazon Kinesis Data Firehose。

约束

- 您必须拥有存储消息的日志记录位置。
- 该房间、日志记录配置和日志记录位置必须位于同一亚马逊云科技区域。
- 只有处于活动状态的日志记录配置可用于聊天记录。
- 您只能删除不再与任何房间关联的日志记录配置。

将消息记录到您拥有的位置需要使用您的 Amazon 凭证进行授权。为了向 IVS Chat 提供所需的访问权限，在创建日志记录配置时会自动生成资源策略（适用于 Amazon S3 存储桶或 CloudWatch Logs）或 Amazon IAM 服务相关角色 (SLR)（适用于 Amazon Kinesis Data Firehose）。对角色或策略进行任何修改时请务必谨慎，因为这可能会影响聊天记录的权限。

使用 Amazon CloudWatch 监控错误

您可以使用 Amazon CloudWatch 监控聊天记录中出现的错误，也可以创建警报或控制面板来指示或响应对特定错误的更改。

有几种类型的错误。有关更多信息，请参阅监控 Amazon IVS 聊天功能。

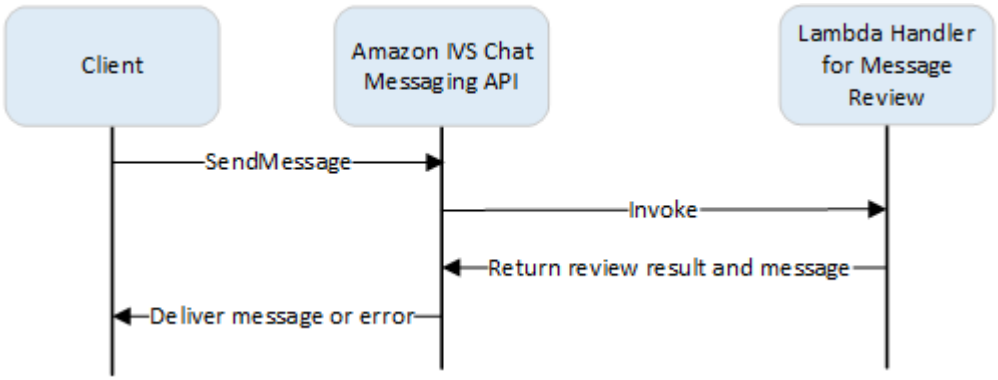
IVS 聊天功能消息审核处理程序

消息审核处理程序允许您在将消息传送到房间之前审查和/或修改消息。当消息审核处理程序与房间关联时，会对该房间的每个 SendMessage 请求调用该处理程序。处理程序会强制执行应用程序的业务逻辑，并确定是否允许、拒绝或修改消息。Amazon IVS Chat 支持将 AWS Lambda 函数作为处理程序。

创建 Lambda 函数

在为房间设置消息审核处理程序之前，您必须使用基于资源的 IAM Policy 创建 Lambda 函数。Lambda 函数必须与将要使用该函数的房间位于同一亚马逊云科技账户和亚马逊云科技区域中。基于资源的策略允许 Amazon IVS Chat 调用 Lambda 函数。有关说明，请参阅 Amazon IVS 聊天功能基于资源的策略。

工作流



请求语法

当客户端发送消息时，Amazon IVS Chat 会使用 JSON 有效负载调用 Lambda 函数：

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

请求正文

字段	描述
`Attributes`	与消息关联的属性。
`Content`	消息的原始内容。
`MessageId`	消息 ID。由 IVS Chat 生成。
`RoomArn`	发送消息的房间的 ARN。
`Sender`	有关发件人的信息。此对象具有多个字段： • `Attributes` – 身份验证期间建立的关于发件人的元数据。这可用于向客户端提供有关发件人的更多信息，例如头像 URL、徽章、字体和颜色。 • `UserId` – 发送此消息的查看者（最终用户）的应用程序指定的标识符。客户端应用程序可使用它以在消息收发 API 或应用程序域中引用用户。 • `Ip` – 发送消息的客户端的 IP 地址。

响应语法

处理程序 Lambda 函数必须返回包含以下语法的 JSON 响应。不符合以下语法或不满足字段约束的响应视为无效。在此情况下，允许或拒绝消息取决于您在消息审核处理程序中指定的 `FallbackResult` 值；请参阅《Amazon IVS Chat API 参考》中的 MessageReviewHandler。

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

响应字段

字段	描述
`Attributes`	与从 Lambda 函数返回的消息相关联的属性。

字段	描述
	如果 ReviewResult 是 DENY，则可以在 Attributes 中提供 Reason；例如： "Attributes": {"Reason": "denied for moderation 在此情况下，发件人客户端会收到 WebSocket 406 错误，错误消息中包含原因。（请参阅《Amazon IVS Chat Messaging API 参考》中的 WebSocket 错误。） • 大小限制：最大 1 KB • 必需：否
Content	从 Lambda 函数返回的消息的内容。它可以根据业务逻辑进行编辑或原创。 • 长度限制：长度下限为 1。您在创建/更新房间时定义的 MaximumMessageLength 的最大长度。有关更多信息，请参阅 Amazon IVS Chat API 参考。仅当 ReviewResult 是 ALLOW 时适用。 • 必需：是
ReviewResult	关于如何处理消息的审核处理结果。如果允许，则会将消息传送到连接到房间的所有用户。如果拒绝，则不会将消息传送给任何用户。 • 有效值：ALLOW | DENY • 必需：是

示例代码

以下是 Go 中的 Lambda 处理程序示例。它会修改消息内容，保持消息属性不变，并允许消息。

```
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
}
```

将处理程序与房间关联和取消关联

设置并实施 Lambda 处理程序后，请使用 Amazon IVS Chat API：

- 要将处理程序与房间关联，请调用 CreateRoom 或 UpdateRoom 并指定处理程序。
- 要取消处理程序与房间的关联，请使用 `MessageReviewHandler.Uri` 的空值调用 UpdateRoom。

使用 Amazon CloudWatch 监控错误

您可以使用 Amazon CloudWatch 监控消息审核中出现的错误，也可以创建告警或控制面板来指示或响应特定错误的更改。如果出现错误，则根据您在将处理程序与房间关联时指定的 `FallbackResult` 值来允许或拒绝该消息；请参阅《Amazon IVS Chat API 参考》中的 MessageReviewHandler。

有几种类型的错误：

- 当 Amazon IVS Chat 无法调用处理程序时，会出现 `InvocationErrors`。
- 当处理程序返回的响应无效时，会出现 `ResponseValidationErrors`。
- 当调用 Lambda 处理程序并返回函数错误时，会出现 AWS Lambda `Errors`。

有关调用错误和响应验证错误（由 Amazon IVS 聊天功能发出）的更多信息，请参阅监控 Amazon IVS 聊天功能。有关 AWS Lambda 错误的更多信息，请参阅使用 Lambda 指标。

监控 Amazon IVS 聊天功能

您可以使用 Amazon CloudWatch 监控 Amazon Interactive Video Service（IVS）聊天功能资源。CloudWatch 可从 Amazon IVS 聊天功能收集和处理原始数据，并将数据处理为便于读取的近乎实时的指标。这些统计数据会保存 15 个月，从而使您能够从历史角度了解您的 Web 应用程序或服务的执行情况。您可以设置用于特定阈值的警报，并在达到相应阈值时发送通知或执行操作。有关详细信息，请参阅 CloudWatch 用户指南。

访问 CloudWatch 指标

Amazon CloudWatch 可从 Amazon IVS 聊天功能收集和处理原始数据，并将数据处理为便于读取的近乎实时的指标。这些统计数据会保存 15 个月，从而使您能够从历史角度了解您的 Web 应用程序或服务的执行情况。您可以设置用于特定阈值的警报，并在达到相应阈值时发送通知或执行操作。有关详细信息，请参阅 CloudWatch 用户指南。

请注意，随着时间的推移，CloudWatch 指标会不断累积。随着指标的老化，分辨率实际有所降低。以下是计划：

- 60 秒的指标可用期为 15 天。
- 5 分钟的指标可用期为 63 天。
- 1 小时的指标可用 455 天（15 个月）。

有关数据保留的最新信息，请在 Amazon CloudWatch 常见问题中搜索“保留期”。

CloudWatch 控制台说明

1. 访问 https://console.aws.amazon.com/cloudwatch/ 打开 CloudWatch 控制台。
2. 在侧导航栏中，展开 Metrics（指标）下拉菜单，然后选择 All metrics（所有指标）。
3. 在浏览选项卡上，使用左侧未标记的下拉菜单，选择您的“主”区域，即创建通道的区域。有关区域的详细信息，请参阅全球解决方案，区域控制。有关支持区域的列表，请参阅亚马逊云科技一般参考中的 Amazon IVS 页面。
4. 在浏览选项卡的底部，选择 IVSChat 命名空间。
5. 请执行以下操作之一：

   a. 在搜索栏中，输入资源 ID（是 ARN `arn:::ivschat:room/<resource id>` 的一部分）。

然后选择 IVSChat。

b. 如果 IVSChat 显示为 Amazon 命名空间下的一个可选服务，选择该服务。如果您使用 Amazon IVSChat 并将指标发送给 Amazon CloudWatch，则将会列出 IVSChat。（如果 IVSChat 未列出，则表示您没有任何 Amazon IVSChat 指标。）

然后根据需要选择维度分组；可用维度将在下面的 CloudWatch 指标 中列出。

6. 选择要添加到图表的指标。可用维度将在下面的 CloudWatch 指标 中列出。

您还可以从聊天会话的详细信息页面访问聊天会话的 CloudWatch 图表，方法是选择在 CloudWatch 中查看按钮。

CLI 说明

您也可以使用 Amazon CLI 访问指标。这需要首先在计算机上下载并配置 CLI。有关详细信息，请参阅 Amazon 命令行界面用户指南。

然后，使用 AWS CLI 访问 Amazon IVS 低延迟聊天指标：

- 在命令提示符下，运行：

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

有关更多信息，请参阅 Amazon CloudWatch 用户指南中的使用 Amazon CloudWatch 指标。

CloudWatch 指标：IVS 聊天功能

Amazon IVS Chat 在 Amazon/IVSChat 命名空间中提供以下指标。

指标	维度	描述
`ConcurrentChatConnections`	无	聊天室中的并发连接总数（每分钟报告的最大连接数）。这有助于了解客户何时接近区域中的并发聊天连接限制。 单位：计数 有效统计数据：总数、平均值、最大值、最小值

指标	维度	描述
Deliveries	操作	由特定操作类型向某个区域中所有房间的聊天连接发送的消息请求数。 单位：计数 有效统计数据：总数、平均值、最大值、最小值
InvocationErrors	Uri	某个区域中所有房间的特定消息审核处理程序的调用错误数。无法调用消息审核处理程序时，会出现调用错误。 当 Amazon IVS Chat 无法调用处理程序时，会出现调用错误。如果与房间关联的处理程序不再存在或超时，或者其资源策略不允许服务对其进行调用，则可能会发生这种情况。 单位：计数 有效统计数据：总数、平均值、最大值、最小值
LogDestinationAccessDeniedError	LoggingConfiguration	日志目标在某个区域内所有房间的拒绝访问错误数。 当 Amazon IVS Chat 无法访问您在日志记录配置中指定的目标资源时，会出现这些错误。如果目标资源策略不允许服务放置记录，则可能会发生这种情况。 单位：计数 有效统计数据：总数、平均值、最大值、最小值

指标	维度	描述
LogDestinationErrors	LoggingConfiguration	日志目标在某个区域内所有房间的所有错误数。 这是一个聚合指标，包括当 Amazon IVS Chat 无法将日志传送到您在日志记录配置中指定的目标资源时发生的所有类型的错误。 单位：计数 有效统计数据：总数、平均值、最大值、最小值
LogDestinationResourceNotFoundErrors	LoggingConfiguration	日志目标在某个区域内所有房间的“找不到资源”错误数。 当 Amazon IVS Chat 因为资源不存在而无法将日志传送到您在日志记录配置中指定的目标资源时，会出现这些错误。如果与日志记录配置相关联的目标资源不再存在，则可能会发生这种情况。 单位：计数 有效统计数据：总数、平均值、最大值、最小值
MessagingDeliveries	无	向某个区域中所有房间的聊天连接发送的消息请求数。 单位：计数 有效统计数据：总数、平均值、最大值、最小值
MessagingRequests	无	某个区域中所有房间发出的消息请求数。 单位：计数 有效统计数据：总数、平均值、最大值、最小值

指标	维度	描述
Requests	操作	某个区域中所有房间由特定操作类型发出的请求数。 单位：计数 有效统计数据：总数、平均值、最大值、最小值
ResponseValidationErrors	Uri	某个区域中所有房间的特定消息审核处理程序的响应验证错误数。当消息审核处理程序的响应无效时，会出现响应验证错误。这可能意味着无法解析响应或验证检查失败；例如，审核结果无效或响应值过长。 单位：计数 有效统计数据：总数、平均值、最大值、最小值

IVS chat 客户端消息收发 SDK

Amazon Interactive Video Services（IVS）Chat 客户端消息收发 SDK 适用于使用 Amazon IVS 构建应用程序的开发人员。此 SDK 旨在利用 Amazon IVS 架构，并将与 Amazon IVS Chat 一起查看更新。作为本机 SDK，它旨在最大限度地减少对应用程序以及用户有权访问应用程序所在设备的性能影响。

平台要求

桌面浏览器

浏览器	受支持的版本
Chrome	两个主要版本（当前版本和最新版本）
边缘	两个主要版本（当前版本和最新版本）
Firefox	两个主要版本（当前版本和最新版本）
Opera	两个主要版本（当前版本和最新版本）
Safari	两个主要版本（当前版本和最新版本）

移动浏览器

浏览器	受支持的版本
Android 版 Chrome	两个主要版本（当前版本和最新版本）
适用于 Android 的 Firefox	两个主要版本（当前版本和最新版本）
适用于 Android 的 Opera	两个主要版本（当前版本和最新版本）

浏览器	受支持的版本
适用于 Android 的 WebView	两个主要版本（当前版本和最新版本）
Samsung Internet	两个主要版本（当前版本和最新版本）
适用于 iOS 的 Safari 浏览器	两个主要版本（当前版本和最新版本）

本机平台

平台	受支持的版本
Android	5.0 和更高版本
iOS	13.0 和更高版本

支持

如果聊天室出现错误或其他问题，请通过 IVS Chat API 确定唯一的聊天室标识符（请参阅 ListRooms）。

与 Amazon Support 共享此聊天室标识符。利用它可获得有助于解决问题的信息。

注意：请参阅 Amazon IVS 聊天功能发布说明了解可用版本和已修复问题。如果合适，请在联系支持部门之前更新您的 SDK 版本，看看这是否解决了您的问题。

版本控制

Amazon IVS Chat 客户端消息收发 SDK 使用语义版本控制。

在此讨论中，假设：

- 最新版本是 4.1.3。
- 先前主要版本的最新版本为 3.2.4。
- 版本 1.x 最新版本是 1.5.6。

最新版本的次要版本已添加向后兼容的新功能。在本例中，版本 4.2.0 已添加新功能。

最新版本的补丁版本已添加向后兼容、次要错误修复。在这里，版本 4.1.4 已添加次要错误修复。

向后兼容、主要错误修复处理方式不同；将在以下几个版本中添加：

- 最新版本补丁版本。在本例中是版本 4.1.4。
- 先前次要版本的补丁版本。在本例中是版本 3.2.5。
- 最新版本 1.x 版本的补丁版本。在本例中是版本 1.5.7。

主要错误修复由 Amazon IVS 产品团队定义。典型示例包括关键安全更新和客户所需的其他选定修复。

注意：在上面的例子中，发布的版本递增但不会跳过任何数字（例如，从 4.1.3 到 4.1.4）。实际上，一个或多个补丁编号可能保留在内部而不发布，因此发布版本可以从 4.1.3 增加到 4.1.6。

此外，版本 1.x 将一直受支持，直到 2023 年底或 3.x 发布时，以较迟者为准。

Amazon IVS Chat API

在服务器端（并非由 SDK 托管），存在两个 API，每个 API 都有自己的职责：

- 数据面板 - IVS Chat 消息收发 API 是一个 WebSocket API，旨在供由基于令牌的身份验证方案提供支持的前端应用程序（iOS、Android、macOS 等）使用。通过之前生成的聊天令牌，您可以使用此 API 连接到现有的聊天室。

  Amazon IVS Chat 客户端消息收发 SDK 仅与数据面板有关。SDK 假设您已经通过后端生成聊天令牌。假设这些令牌的检索由前端应用程序而不是 SDK 进行托管。
- 控制面板 - IVS Chat 控制面板 API 为您自己的后端应用程序提供了一个界面，用于管理和创建聊天室以及加入聊天室的用户。您可以将此界面视为应用程序聊天功能（由您自己的后端进行托管）的管理面板。有些控制面板端点负责创建数据面板对聊天室进行身份验证所需的聊天令牌。

  重要提示：IVS Chat 客户端消息收发 SDK 不调用任何控制面板端点。必须设置后端才能创建聊天令牌。前端应用程序必须与后端通信才能检索此聊天令牌。

IVS 聊天功能客户端消息收发 SDK：Android 指南

Amazon Interactive Video（IVS）Chat 客户端消息收发 Android SDK 提供界面，可让您在使用 Android 的平台上轻松整合我们的 IVS Chat 消息收发 API。

com.amazonaws:ivs-chat-messaging 软件包实现了本文档中所描述的接口。

IVS 聊天功能客户端消息收发 Android SDK 的最新版本：1.1.0（发布说明）

参考文档：有关 Amazon IVS Chat 客户端消息收发 Android SDK 中最重要方法的信息，请参阅参考文档，网址为 https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/。

示例代码：请参阅 GitHub 上的 Android 示例存储库：https://github.com/aws-samples/amazon-ivs-chat-for-android-demo

平台要求：开发需要 Android 5.0（API 级别 21）或更高版本。

IVS 聊天功能客户端消息收发 Android SDK 入门

在开始之前，应该熟悉 Amazon IVS Chat 入门。

添加程序包

将 com.amazonaws:ivs-chat-messaging 添加到 build.gradle 依赖项：

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

添加 Proguard 规则

将以下条目添加到 R8/Proguard 规则文件 (proguard-rules.pro)：

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

设置您的后端

此集成需要服务器上的端点与 Amazon IVS API 通信。使用官方亚马逊云科技库从服务器访问 Amazon IVS API。这些库可以从 node.js 和 Java 等公共程序包以多种语言进行访问。

接下来，创建一个用于与 Amazon IVS Chat API 通信的服务器端点，然后创建令牌。

设置服务器连接

创建一个将 ChatTokenCallback 作为参数的方法，然后从后端获取聊天令牌。将该令牌传递给回调的 onSuccess 方法。如果发生错误，将异常传递给回调的 onError 方法。在下一步中实例化主 ChatRoom 实体时需要执行此操作。

您可以在下面找到使用 Retrofit 调用实现上述操作的示例代码。

```
// ...

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

使用 IVS 聊天功能客户端消息收发 Android SDK

本文档将引导您完成使用 Amazon IVS 聊天功能客户端消息收发 Android SDK 涉及的步骤。

初始化聊天室实例

创建 ChatRoom 类的实例。此操作需要传递 regionOrUrl（通常是托管聊天室的 AWS 区域）和 tokenProvider（上一步中创建的令牌获取方法）。

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

接下来，创建一个侦听器对象，该对象将实现聊天相关事件的处理程序，然后将其分配给 room.listener 属性：

```
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

// ...
```

基本初始化的最后一步是通过建立 WebSocket 连接来连接到特定聊天室。为此，请在聊天室实例中调用 connect() 方法。我们建议在 onResume() lifecycle 方法中执行此操作，以确保在应用程序从后台恢复时保持连接状态。

```
room.connect()
```

SDK 将尝试与聊天室建立连接，该聊天室是使用从服务器收到的聊天令牌进行编码的。如果失败，它将尝试重新连接聊天室实例中指定的次数。

在聊天室中执行操作

ChatRoom 类包含发送和删除消息以及与其他用户断开连接的操作。这些操作接受可选的回调参数，该参数允许您获取请求确认或拒绝通知。

发送消息

对于此请求，您必须使用聊天令牌对 SEND_MESSAGE 功能进行编码。

触发发送消息请求：

```
val request = SendMessageRequest("Test Echo")
room.sendMessage(request)
```

要获得请求的确认/拒绝，请提供回调作为第二个参数：

```
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.
   }
})
```

删除消息

对于此请求，您必须在聊天令牌中编码 DELETE_MESSAGE 功能。

触发删除消息请求：

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

要获得请求的确认/拒绝，请提供回调作为第二个参数：

```
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.
    }
})
```

断开与其他用户的连接

对于此请求，您必须使用聊天令牌对 DISCONNECT_USER 功能进行编码。

出于审核目的与其他用户断开连接：

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

要获得请求的确认/拒绝，请提供回调作为第二个参数：

```
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.
   }
})
```

断开与聊天室的连接

要关闭与聊天室的连接，请对聊天室实例调用 disconnect() 方法：

```
room.disconnect()
```

由于当应用程序处于后台状态时，WebSocket 连接将在短时间后停止工作，因此我们建议您在从后台状态转换为其他状态/转换为后台状态时手动连接/断开连接。为此，请将 onResume() 生命周期方法中有关 Android Activity 或 Fragment 的 room.connect() 调用与 onPause() 生命周期方法中的 room.disconnect() 调用相匹配。

IVS 聊天功能客户端消息收发 SDK：Android 教程第 1 部分：聊天室

这是一个由两部分组成的教程的第 1 部分。通过使用 Kotlin 编程语言构建功能齐全的 Android 应用程序，您将了解使用 Amazon IVS 聊天功能消息收发 SDK 的基本知识。我们把这个应用程序称为 Chatterbox。

在开始本模块之前，请花几分钟时间熟悉先决条件、聊天令牌背后的关键概念以及创建聊天室所需的后端服务器。

这些教程是为刚接触 IVS 聊天功能消息收发 SDK 但经验丰富的 Android 开发人员创建的。您需要熟悉 Kotlin 编程语言并在 Android 平台上创建 UI。

本教程的第 1 部分分为几个章节：

1. the section called “设置本地身份验证/授权服务器”
2. the section called “创建 Chatterbox 项目”
3. the section called “连接到聊天室和观察连接更新”
4. the section called “构建令牌提供程序”
5. the section called “后续步骤”

要学习完整的 SDK 文档，请从亚马逊 IVS 聊天功能客户端消息收发 SDK（在《Amazon IVS 聊天功能用户指南》中）和 Chat Client Messaging: SDK for Android Reference（在 GitHub 上）开始。

先决条件

- 熟悉 Kotlin 并在 Android 平台上创建应用程序。如果您对创建 Android 应用程序不熟悉，请学习面向 Android 开发者的构建您的第一个应用程序指南中的基础知识。
- 仔细阅读并理解 IVS 聊天功能入门。
- 使用现有的 IAM policy 中定义的 `CreateChatToken` 和 `CreateRoom` 功能创建 AWS IAM 用户。（请参见 IVS 聊天功能入门。）
- 确保该用户的私有密钥/访问密钥存储在 Amazon 凭证文件中。有关说明，请参阅 Amazon CLI 用户指南（尤其是配置和凭证文件设置部分）。
- 创建聊天室并保存其 ARN。请参阅 IVS 聊天功能入门。（如果不保存 ARN，您可以稍后使用控制台或 Chat API 查找 ARN。）

设置本地身份验证/授权服务器

您的后端服务器负责创建聊天室和生成 IVS 聊天功能 Android SDK 所需的聊天令牌，以便在您的客户端连接聊天室时进行身份验证和授权。

请参阅《Amazon IVS Chat 入门》中的创建聊天令牌。如其中的流程图所示，您的服务器端代码负责创建聊天令牌。这意味着您的应用程序必须通过向服务器端应用程序请求聊天令牌，来提供自己生成聊天令牌的方法。

我们使用 Ktor 框架创建一个实时本地服务器，该服务器使用您的本地 Amazon 环境管理聊天令牌的创建。

此时，我们希望您已正确设置 AWS 凭证。有关分步说明，请参阅设置用于开发的 AWS 凭证和区域。

创建一个新目录并命名为 `chatterbox`，在其中再创建一个，然后命名为 `auth-server`。

我们的服务器文件夹将具有如下结构：

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

注意：您可以直接将此处的代码复制/粘贴到引用文件中。

接下来，我们添加所有必要的依赖项和插件，以便身份验证服务器正常工作：

Kotlin 脚本：

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

现在我们需要为身份验证服务器设置日志记录功能。（有关更多信息，请参阅配置记录器。）

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>
```

Ktor 服务器需要自动从 `resources` 目录的 `application.*` 文件中加载的配置设置，所以我们也添加了这些设置。（有关更多信息，请参阅文件中的配置。）

HOCON：

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

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

最后，让我们实施我们的服务器：

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()
                )
            )
        }
    }
}
```

创建 Chatterbox 项目

要创建 Android 项目，请安装并打开 Android Studio。

按照 Android 官方在创建项目指南中列出的步骤进行操作。

- 在选择您的项目类型中，为我们的 Chatterbox 应用程序选择空活动项目模板。
- 在配置您的项目中，为配置字段选择以下值：
  - 名称：My App
  - 程序包名称：com.chatterbox.myapp
  - 保存位置：指向在上一步中创建的 `chatterbox` 目录
  - 语言：Kotlin
  - 最低 API 级别：API 21：Android 5.0（Lollipop）

正确指定所有配置参数后，我们在 `chatterbox` 文件夹内的文件结构应如下所示：

```
- 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
```

现在我们有一个正在运行的 Android 项目，可以将 com.amazonaws:ivs-chat-messaging 添加到 build.gradle 依赖项中。（有关 Gradle 构建工具包的更多信息，请参阅配置您的构建。）

注意：在每个代码段的顶部，都有一个文件路径，您应该在其中对项目进行更改。路径相对于项目的根目录。

在下面的代码中，将 *<version>* 替换为聊天功能 Android SDK 的当前版本号（例如 1.0.0）。

Kotlin：

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

plugins {
// ...
}

android {
// ...
}
```

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

添加新依赖项后，在 Android Studio 中运行将项目与 Gradle 文件同步，将项目与新依赖项同步。（有关更多信息，请参阅添加构建依赖项。）

为了方便从项目根目录运行我们的身份验证服务器（在上一部分中创建），我们将其作为新模块包含在 `settings.gradle` 中。（有关更多信息，请参阅使用 Gradle 构建软件组件。）

Kotlin 脚本：

```
// ./settings.gradle

// ...

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

从现在起，由于 `auth-server` 已包含在 Android 项目中，您可以从项目的根目录使用以下命令运行身份验证服务器：

Shell：

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

连接到聊天室和观察连接更新

为了打开聊天室连接，我们使用了 onCreate() 活动生命周期回调，该回调在首次创建活动时触发。ChatRoom 构造函数要求我们提供 `region` 和 `tokenProvider` 以实例化聊天室连接。

注意：以下代码段中的 `fetchChatToken` 功能将在下一部分中实现。

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)
   }

// ...
}
```

显示聊天室连接的变化并对其做出反应是制作 chatterbox 等聊天应用程序的重要组成部分。在我们开始与聊天室互动之前，必须订阅聊天室连接状态事件以获取更新。

ChatRoom 希望我们附加 ChatroomListener 接口实现来引发生命周期事件。目前，侦听器函数在调用时只会记录确认消息：

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")
        }
    }
}
```

现在我们已经实现 ChatRoomListener，接下来将其附加到聊天室实例上：

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 {
// ...
}
```

此后，我们需要提供读取聊天室连接状态的功能。我们会将其保留在 MainActivity.kt 属性中并将其初始化为聊天室连接默认断开状态（参阅 IVS 聊天功能 Android SDK 参考中的 ChatRoom state）。为了保持最新的本地状态，我们需要实现一个状态更新器函数，我们称之为 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")
            }
```

```
        }
    }
}
```

接下来，我们将状态更新器函数与 ChatRoom.listener 属性集成：

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)
            }
        }
    }
}
```

现在我们能够保存、侦听和响应 ChatRoom 状态更新，接下来可以初始化连接了：

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

构建令牌提供程序

现在可以创建一个函数，负责在我们的应用程序中创建和管理聊天令牌。在此示例中，我们使用 Android 版 Retrofit HTTP 客户端。

在发送任何网络流量之前，我们必须设置适用于 Android 的网络安全配置。（有关更多信息，请参阅网络安全配置。）我们首先向应用程序清单文件添加网络权限。请注意，添加的 `user-permission` 标签和 `networkSecurityConfig` 属性将指向我们的新网络安全配置。在下面的代码中，将 *<version>* 替换为聊天功能 Android SDK 的当前版本号（例如 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")
}
```

声明 `10.0.2.2` 和 `localhost` 为可信域，以开始与我们的后端交换消息：

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>
```

接下来，我们需要添加一个新的依赖项以及用于解析 HTTP 响应的 Gson 转换器。在下面的代码中，将 *<version>* 替换为聊天功能 Android SDK 的当前版本号（例如 1.0.0）。

Kotlin 脚本：

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

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

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

要检索聊天令牌，我们需要从 chatterbox 应用程序发出 POST HTTP 请求。我们在 Retrofit 接口中定义请求，以便实现。（请参阅 Retrofit 文档。还要熟悉 CreateChatToken 端点规范。）

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>
}
```

现在，网络设置完成后，可以添加一个函数，负责创建和管理我们的聊天令牌。我们将其添加到 MainActivity.kt，它是在项目生成时自动创建的：

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)
         }
      })
   }
}
```

后续步骤

现在，您已经建立聊天室连接，请继续阅读本 Android 教程的第 2 部分：消息和事件。

IVS 聊天功能客户端消息收发 SDK：Android 教程第 2 部分：消息和事件

本教程的第 2 部分（也是最后一部分）分为几个章节：

1. the section called “为发送消息创建一个 UI”
   a. the section called “UI 主布局”
   b. the section called “UI 将文本单元格抽象化以便一致地显示文本”
   c. the section called “UI 左侧聊天消息”
   d. the section called “UI 右侧聊天消息”
   e. the section called “UI 其他颜色值”

2. the section called “应用视图绑定”
3. the section called “管理聊天消息请求”
4. the section called “最终步骤”

要学习完整的 SDK 文档，请从亚马逊 IVS 聊天功能客户端消息收发 SDK（在《Amazon IVS 聊天功能用户指南》中）和 Chat Client Messaging: SDK for Android Reference（在 GitHub 上）开始。

先决条件

确保您已完成本教程的第 1 部分聊天室。

为发送消息创建一个 UI

现在我们已成功初始化聊天室连接，接下来可以发送第一条消息了。要使用此功能，需要一个 UI。我们将添加：

- `connect/disconnect` 按钮
- 使用 `send` 按钮输入消息
- 动态消息列表。为了构建此内容，我们使用了 Android Jetpack RecyclerView。

UI 主布局

请参阅 Android 开发者文档中的 Android Jetpack 布局。

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 将文本单元格抽象化以便一致地显示文本

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 左侧聊天消息

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 右侧聊天消息

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 其他颜色值

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>
```

应用视图绑定

我们利用 Android 视图绑定功能来引用 XML 布局的绑定类。要启用该功能，请在 `./app/build.gradle` 中将 `viewBinding` 构建选项设置为 `true`：

Kotlin 脚本：

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

android {
//    ...

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

现在可以将 UI 与我们的 Kotlin 代码连接起来：

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)
    }
}
```

我们还添加了删除消息和断开用户聊天连接的方法，可以使用聊天消息上下文菜单调用这些方法：

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}")
                    }
                }
            }
        )
```

```
    }
}
```

管理聊天消息请求

我们需要一种用于管理不同状态的聊天消息请求的方法：

- 待处理 — 消息已发送到聊天室，但尚未被确认或拒绝。
- 已确认 — 消息已被聊天室发送给所有用户（包括我们）。
- 已拒绝 — 消息已被聊天室拒绝，并提示带有错误对象。

我们会把未解决的聊天请求和聊天消息保留在列表中。该列表属于一个单独的类，我们称之为 `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()
    }
}
```

为了将列表与 UI 连接起来，我们使用适配器。有关更多信息，请参阅使用 AdapterView 绑定到数据和生成的绑定类。

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
}
```

最终步骤

现在可以挂载我们的新适配器，将 ChatEntries 类绑定到 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
        }
    }
}
```

由于我们已经有一个用于负责跟踪我们聊天请求的类 (ChatEntries)，我们已经准备好实现用于在 roomListener 中操作 entries 的代码。我们将根据自己正在响应的事件来更新 entries 和 connectionState：

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")
        }
    }
}
```

现在应该可以运行您的应用程序了！（请参阅构建和运行您的应用程序。）使用应用程序时，请记住让您的后端服务器保持运行。您可以通过使用命令 `./gradlew :auth-server:run`，从我们项目根目录的终端启动，或是直接从 Android Studio 执行 `auth-server:run` Gradle 任务来启动。

IVS 聊天功能客户端消息收发 SDK：Kotlin 协同教程第 1 部分：聊天室

这是一个由两部分组成的教程的第 1 部分。通过使用 Kotlin 编程语言和协同例程构建功能齐全的 Android 应用程序，您将了解使用 Amazon IVS 聊天功能消息收发 SDK 的基本知识。我们把这个应用程序称为 Chatterbox。

在开始本模块之前，请花几分钟时间熟悉先决条件、聊天令牌背后的关键概念以及创建聊天室所需的后端服务器。

这些教程是为刚接触 IVS 聊天功能消息收发 SDK 但经验丰富的 Android 开发人员创建的。您需要熟悉 Kotlin 编程语言并在 Android 平台上创建 UI。

本教程的第 1 部分分为几个章节：

1. the section called “设置本地身份验证/授权服务器”
2. the section called “创建 Chatterbox 项目”
3. the section called “连接到聊天室和观察连接更新”
4. the section called “构建令牌提供程序”
5. the section called “后续步骤”

要学习完整的 SDK 文档，请从亚马逊 IVS 聊天功能客户端消息收发 SDK（在《Amazon IVS 聊天功能用户指南》中）和 Chat Client Messaging: SDK for Android Reference（在 GitHub 上）开始。

先决条件

- 熟悉 Kotlin 并在 Android 平台上创建应用程序。如果您对创建 Android 应用程序不熟悉，请学习面向 Android 开发者的构建您的第一个应用程序指南中的基础知识。
- 阅读并理解 IVS 聊天功能入门。
- 使用现有的 IAM policy 中定义的 `CreateChatToken` 和 `CreateRoom` 功能创建 AWS IAM 用户。（请参见 IVS 聊天功能入门。）
- 确保该用户的私有密钥/访问密钥存储在 Amazon 凭证文件中。有关说明，请参阅 Amazon CLI 用户指南（尤其是配置和凭证文件设置部分）。
- 创建聊天室并保存其 ARN。请参阅 IVS 聊天功能入门。（如果不保存 ARN，您可以稍后使用控制台或 Chat API 查找 ARN。）

设置本地身份验证/授权服务器

您的后端服务器负责创建聊天室和生成 IVS 聊天功能 Android SDK 所需的聊天令牌，以便在您的客户端连接聊天室时进行身份验证和授权。

请参阅《Amazon IVS Chat 入门》中的创建聊天令牌。如其中的流程图所示，您的服务器端代码负责创建聊天令牌。这意味着您的应用程序必须通过向服务器端应用程序请求聊天令牌，来提供自己生成聊天令牌的方法。

我们使用 Ktor 框架创建一个实时本地服务器，该服务器使用您的本地 Amazon 环境管理聊天令牌的创建。

此时，我们希望您已正确设置 AWS 凭证。有关分步说明，请参阅 Set up AWS temporary credentials and AWS Region for development。

创建一个新目录并命名为 chatterbox，在其中再创建一个，然后命名为 auth-server。

我们的服务器文件夹将具有如下结构：

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

注意：您可以直接将此处的代码复制/粘贴到引用文件中。

接下来，我们添加所有必要的依赖项和插件，以便身份验证服务器正常工作：

Kotlin 脚本：

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

现在我们需要为身份验证服务器设置日志记录功能。（有关更多信息，请参阅配置记录器。）

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>
```

Ktor 服务器需要自动从 resources 目录的 application.* 文件中加载的配置设置，所以我们也添加了这些设置。（有关更多信息，请参阅文件中的配置。）

HOCON：

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

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

```
}
```

最后，让我们实施我们的服务器：

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()
            )
        )
    }
  }
}
```

创建 Chatterbox 项目

要创建 Android 项目，请安装并打开 Android Studio。

按照 Android 官方在创建项目指南中列出的步骤进行操作。

- 在选择您的项目中，为我们的 Chatterbox 应用程序选择空活动项目模板。

- 在配置您的项目中，为配置字段选择以下值：
  - 名称：My App
  - 程序包名称：com.chatterbox.myapp
  - 保存位置：指向在上一步中创建的 `chatterbox` 目录
  - 语言：Kotlin
  - 最低 API 级别：API 21：Android 5.0（Lollipop）

正确指定所有配置参数后，我们在 `chatterbox` 文件夹内的文件结构应如下所示：

```
- 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
```

现在我们有一个正在运行的 Android 项目，可以将 com.amazonaws:ivs-chat-messaging 和 org.jetbrains.kotlinx:kotlinx-coroutines-core 添加到 `build.gradle` 依赖项中。（有关 Gradle 构建工具包的更多信息，请参阅配置您的构建。）

注意：在每个代码段的顶部，都有一个文件路径，您应该在其中对项目进行更改。路径相对于项目的根目录。

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'

// ...
}
```

添加新依赖项后，在 Android Studio 中运行将项目与 Gradle 文件同步，将项目与新依赖项同步。（有关更多信息，请参阅添加构建依赖项。）

为了方便从项目根目录运行我们的身份验证服务器（在上一部分中创建），我们将其作为新模块包含在 `settings.gradle` 中。（有关更多信息，请参阅使用 Gradle 构建软件组件。）

Kotlin 脚本：

```
// ./settings.gradle

// ...

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

从现在起，由于 `auth-server` 已包含在 Android 项目中，您可以从项目的根目录使用以下命令运行身份验证服务器：

Shell：

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

连接到聊天室和观察连接更新

为了打开聊天室连接，我们使用了 onCreate() 活动生命周期回调，该回调在首次创建活动时触发。ChatRoom 构造函数要求我们提供 `region` 和 `tokenProvider` 以实例化聊天室连接。

注意：以下代码段中的 `fetchChatToken` 功能将在下一部分中实现。

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)
    }

// ...
}
```

显示聊天室连接的变化并对其做出反应是制作 chatterbox 等聊天应用程序的重要组成部分。在我们开始与聊天室互动之前，必须订阅聊天室连接状态事件以获取更新。

在协同例程的聊天功能 SDK 中，ChatRoom 希望我们在流中处理聊天室生命周期事件。目前，函数在调用时只会记录确认消息：

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")
                }
            }
        }
    }
}
```

此后，我们需要提供读取聊天室连接状态的功能。我们会将其保留在 `MainActivity.kt` 属性中并将其初始化为聊天室连接默认断开状态（参阅 IVS 聊天功能 Android SDK 参考中的 `ChatRoom state`）。为了保持最新的本地状态，我们需要实现一个状态更新器函数，我们称之为 `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")
           }
       }
}
```

接下来，我们将状态更新器函数与 ChatRoom.listener 属性集成：

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)

                }
            }

      // ...

      }
   }
}
```

现在我们能够保存、侦听和响应 ChatRoom 状态更新，接下来可以初始化连接了：

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)
      }
   }

   // ...
}
```

构建令牌提供程序

现在可以创建一个函数，负责在我们的应用程序中创建和管理聊天令牌。在此示例中，我们使用 Android 版 Retrofit HTTP 客户端。

在发送任何网络流量之前，我们必须设置适用于 Android 的网络安全配置。（有关更多信息，请参阅网络安全配置。）我们首先向应用程序清单文件添加网络权限。请注意，添加的 `user-permission` 标签和 `networkSecurityConfig` 属性将指向我们的新网络安全配置。在下面的代码中，将 *<version>* 替换为聊天功能 Android SDK 的当前版本号（例如 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")
}
```

声明您的本地 IP 地址（例如 `10.0.2.2` 和 `localhost`）为可信域，开始与我们的后端交换消息：

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>
```

接下来，我们需要添加一个新的依赖项以及用于解析 HTTP 响应的 Gson 转换器。在下面的代码中，将 *<version>* 替换为聊天功能 Android SDK 的当前版本号（例如 1.1.0）。

Kotlin 脚本：

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

要检索聊天令牌，我们需要从 chatterbox 应用程序发出 POST HTTP 请求。我们在 Retrofit 接口中定义请求，以便实现。（请参阅 Retrofit 文档。还要熟悉 CreateChatToken 端点规范。）

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)
    }
}
```

现在，网络设置完成后，可以添加一个函数，负责创建和管理我们的聊天令牌。我们将其添加到 MainActivity.kt，它是在项目生成时自动创建的：

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)
         }
      })
   }
}
```

后续步骤

现在，您已经建立聊天室连接，请继续阅读本 Kotlin 协同教程的第 2 部分：消息和事件。

IVS 聊天功能客户端消息收发 SDK：Kotlin 协同教程第 2 部分：消息和事件

本教程的第 2 部分（也是最后一部分）分为几个章节：

1. the section called “为发送消息创建一个 UI”
   a. the section called “UI 主布局”
   b. the section called “UI 将文本单元格抽象化以便一致地显示文本”
   c. the section called “UI 左侧聊天消息”
   d. the section called “UI 右侧消息”
   e. the section called “UI 其他颜色值”
2. the section called “应用视图绑定”
3. the section called “管理聊天消息请求”

4. the section called “最终步骤”

要学习完整的 SDK 文档，请从亚马逊 IVS 聊天功能客户端消息收发 SDK（在《Amazon IVS 聊天功能用户指南》中）和 Chat Client Messaging: SDK for Android Reference（在 GitHub 上）开始。

先决条件

确保您已完成本教程的第 1 部分聊天室。

为发送消息创建一个 UI

现在我们已成功初始化聊天室连接，接下来可以发送第一条消息了。要使用此功能，需要一个 UI。我们将添加：

- `connect/disconnect` 按钮
- 使用 `send` 按钮输入消息
- 动态消息列表。为了构建此内容，我们使用了 Android Jetpack RecyclerView。

UI 主布局

请参阅 Android 开发者文档中的 Android Jetpack 布局。

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 将文本单元格抽象化以便一致地显示文本

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 左侧聊天消息

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 右侧消息

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 其他颜色值

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>
```

应用视图绑定

我们利用 Android 视图绑定功能来引用 XML 布局的绑定类。要启用该功能，请在 `./app/build.gradle` 中将 `viewBinding` 构建选项设置为 `true`：

Kotlin 脚本：

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

android {
//    ...

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

现在可以将 UI 与我们的 Kotlin 代码连接起来：

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)
    }
// ...

}
```

我们还添加了删除消息和断开用户聊天连接的方法，可以使用聊天消息上下文菜单调用这些方法：

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")
           }
        }
    }
}
```

管理聊天消息请求

我们需要一种用于管理不同状态的聊天消息请求的方法：

- 待处理 — 消息已发送到聊天室，但尚未被确认或拒绝。
- 已确认 — 消息已被聊天室发送给所有用户（包括我们）。
- 已拒绝 — 消息已被聊天室拒绝，并提示带有错误对象。

我们会把未解决的聊天请求和聊天消息保留在列表中。该列表属于一个单独的类，我们称之为 `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()
    }
}
```

为了将列表与 UI 连接起来，我们使用适配器。有关更多信息，请参阅使用 AdapterView 绑定到数据和生成的绑定类。

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
}
```

最终步骤

现在可以挂载我们的新适配器，将 ChatEntries 类绑定到 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
        }
    }
}
```

由于我们已经有一个用于负责跟踪我们聊天请求的类（ChatEntries），我们已经准备好实现用于在 roomListener 中操作 entries 的代码。我们将根据自己正在响应的事件来更新 entries 和 connectionState：

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)
    }

// ...
```

```
}
```

现在应该可以运行您的应用程序了！（请参阅构建和运行您的应用程序。）使用应用程序时，请记住让您的后端服务器保持运行。您可以通过使用命令 `./gradlew :auth-server:run`，从我们项目根目录的终端启动，或是直接从 Android Studio 执行 `auth-server:run` Gradle 任务来启动。

IVS 聊天功能客户端消息收发 SDK：iOS 指南

Amazon Interactive Video（IVS）Chat 客户端消息收发 iOS SDK 提供界面，可让您在使用 Apple 的 Swift 编程语言的平台上整合我们的 IVS Chat 消息收发 API。

IVS Chat 客户端消息收发 iOS SDK 的最新版本：1.0.0（发布说明）

参考文档和教程：有关 Amazon IVS Chat 客户端消息收发 iOS SDK 中最重要方法的信息，请参阅参考文档，网址为 https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/。此存储库还包含各种文章和教程。

示例代码：请参阅 GitHub 上的 iOS 示例存储库：https://github.com/aws-samples/amazon-ivs-chat-for-ios-demo。

平台要求：开发需要 iOS 13.0 或更高版本。

IVS Chat 客户端消息收发 iOS SDK 入门

我们建议您通过 Swift Package Manager 集成 SDK。或者，您可以使用 CocoaPods 或手动集成框架。

集成 SDK 后，您可以通过在相关 Swift 文件顶部添加以下代码来导入 SDK：

```
import AmazonIVSChatMessaging
```

Swift Package Manager

要在 Swift Package Manager 项目中使用 `AmazonIVSChatMessaging` 库，请将其添加到程序包的依赖项和相关目标的依赖项中：

1. 最新 `.xcframework` 的下载链接：https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip。

2. 在终端中，运行：

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

3. 获取上一步的输出并将其粘贴到 .binaryTarget 的校验和属性中，如下面项目的 Package.swift 文件中所示：

```
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

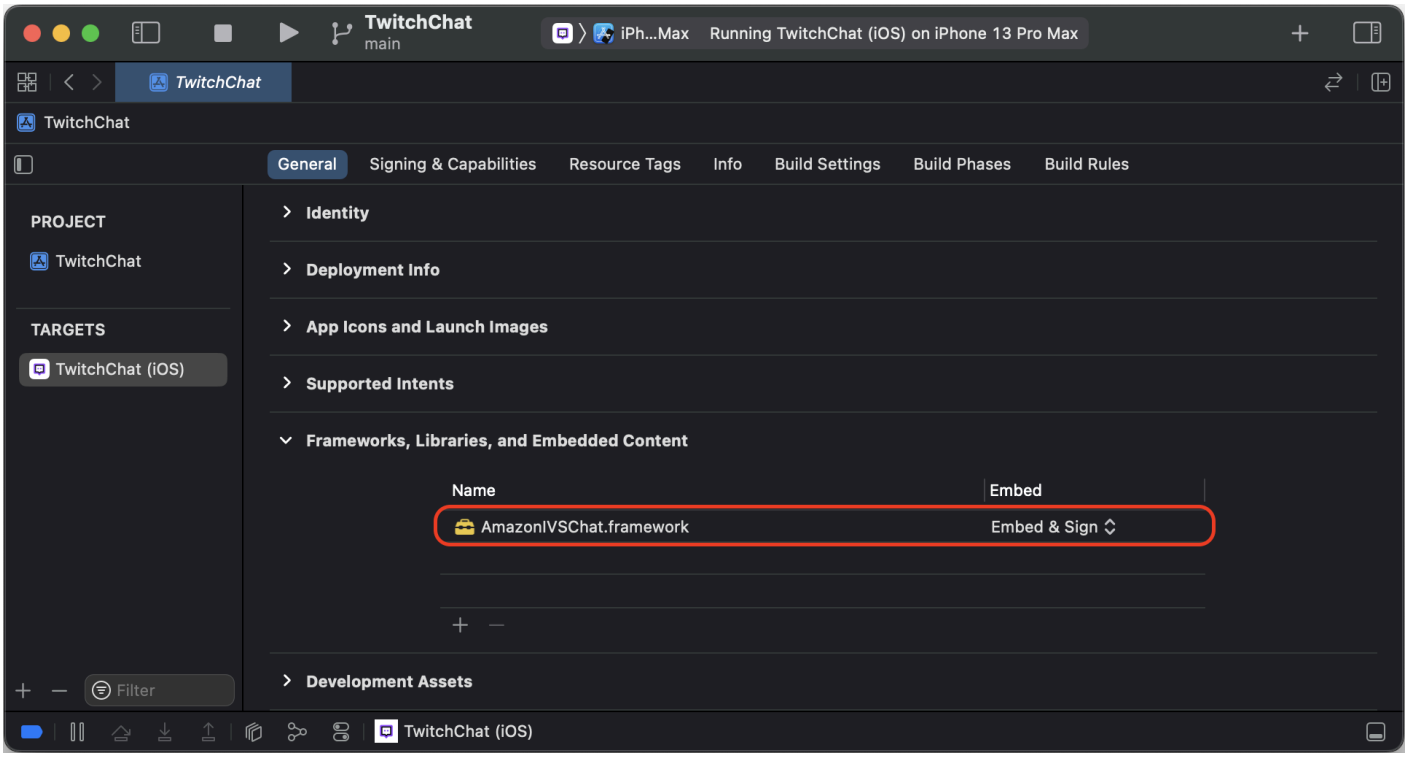
版本通过 CocoaPods 以 AmazonIVSChatMessaging 名称发布。将此依赖项添加至您的 Podfile 中：

```
pod 'AmazonIVSChat'
```

运行 pod install，开发工具包将在 .xcworkspace 中可用。

手动安装

1. 最新版本下载链接：https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip。
2. 提取归档的内容。`AmazonIVSChatMessaging.xcframework` 包含适用于设备和模拟器的开发工具包。
3. 通过以下方法嵌入提取的 `AmazonIVSChatMessaging.xcframework`：将其拖动到应用程序目标 General（常规）选项卡上的 Frameworks, Libraries, and Embedded Content（框架、库和嵌入式内容）部分：



使用 IVS 聊天功能客户端消息收发 iOS SDK

本文档将引导您完成使用 Amazon IVS 聊天功能客户端消息收发 iOS SDK 涉及的步骤。

连接到聊天室

在开始之前，应该熟悉 Amazon IVS Chat 入门。另请参阅适用于 Web、Android 和 iOS 的示例应用程序。

要连接到聊天室，您的应用程序需要以某种方式来检索后端提供的聊天令牌。应用程序可能会使用对后端的网络请求来检索聊天令牌。

要将获取的此聊天令牌与 SDK 进行通信，SDK 的 ChatRoom 模型要求您在初始化时提供 async 函数或符合所提供的 ChatTokenProvider 协议的对象实例。这两种方法返回的值都必须是 SDK ChatToken 模型实例。

注意：您使用从后端检索的数据填充 ChatToken 模型的实例。初始化 ChatToken 实例所需的字段与 CreateChatToken 响应中的字段相同。有关初始化 ChatToken 模型实例的更多信息，请参阅创建 ChatToken 实例。请记住，您的后端负责将 CreateChatToken 响应中的数据提供给您的应用程序。您决定如何与后端通信以生成聊天令牌取决于您的应用程序及其基础设施。

选择向 SDK 提供 ChatToken 的策略后，请在使用令牌提供程序和 亚马逊云科技区域（后端用于创建您要尝试连接到的聊天室）成功初始化 ChatRoom 实例后调用 .connect()。请注意，.connect() 是抛出异步函数：

```
import AmazonIVSChatMessaging

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

符合 ChatTokenProvider 协议

对于 ChatRoom 初始化程序中的 tokenProvider 参数，您可以提供 ChatTokenProvider 的实例。以下是符合 ChatTokenProvider 的对象的示例：

```
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
      )
   }
}
```

然后，您可以采用符合条件的此对象的实例并将其传递给 ChatRoom 的初始化程序：

```
// 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()
```

在 Swift 中提供异步函数

假设您已拥有用于管理应用程序的网络请求的管理器。它可能如下所示：

```
import AmazonIVSChatMessaging

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

您只需在管理器中添加另一个函数即可从后端检索 ChatToken：

```
import AmazonIVSChatMessaging

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

然后，在初始化 ChatRoom 时在 Swift 中使用对该函数的引用：

```
import AmazonIVSChatMessaging

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

```
try await room.connect()
```

创建 ChatTToken 实例

您可以使用 SDK 中提供的初始化程序轻松创建 ChatToken 实例。请参阅 Token.swift 中的文档以了解有关 ChatToken 的各个属性的更多信息。

```
import AmazonIVSChatMessaging

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

使用可解码功能

如果在与 IVS Chat API 交互时，后端决定简单地将 CreateChatToken 响应转发到前端应用程序，那么您就可以利用 ChatToken 对 Swift 的 Decodable 协议的符合性。然而，存在一个隐患。

CreateChatToken 响应有效负载使用字符串表示使用 Internet 时间戳 ISO 8601 标准设置格式的日期。通常在 Swift 中，您会提供 JSONDecoder.DateDecodingStrategy.iso8601 作为 JSONDecoder .dateDecodingStrategy 属性的值。但是，CreateChatToken 在其字符串中使用精确到小数的秒，而 JSONDecoder.DateDecodingStrategy.iso8601 不支持此精度。

为方便起见，SDK 会对 JSONDecoder.DateDecodingStrategy 提供公共扩展以及允许您在解码 ChatToken 实例时成功使用 JSONDecoder 的附加 .preciseISO8601 策略：

```
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)
```

断开与聊天室的连接

要手动断开已成功连接到的 ChatRoom 实例，请调用 room.disconnect()。默认情况下，聊天室在解除分配时会自动调用此函数。

```
import AmazonIVSChatMessaging

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

// Disconnect
room.disconnect()
```

接收聊天消息/事件

要在聊天室中发送和接收消息，您需要在成功初始化 ChatRoom 实例并调用 room.connect() 后提供符合 ChatRoomDelegate 协议的对象。以下是一个使用 UIViewController 的典型示例：

```
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) { ... }
}
```

连接发生变化时收到通知

正如所料，在聊天室完全连接之前，您无法执行诸如在聊天室中发送消息之类的操作。SDK 的架构会尝试建议您通过异步 API 在后台线程上连接到聊天室。如果您想在 UI 中构建禁用发送消息按钮之类的功能，SDK 将使用 Combine 或 ChatRoomDelegate 提供两种策略，以便您在聊天室的连接状态发生变化时收到通知。这些内容如下所述。

重要提示：聊天室的连接状态也可能由于网络连接断开等原因而发生变化。构建应用程序时应将这一点考虑在内。

使用 Combine

每个 ChatRoom 实例都会以 state 属性形式附带自己的 Combine 发布程序：

```
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()
}
```

使用 ChatRoomDelegate

或者，在符合 ChatRoomDelegate 的对象中使用可选函数 roomDidConnect(_:)、roomIsConnecting(_:) 和 roomDidDisconnect(_:)。下面是使用 UIViewController 的示例：

```
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!")
   }
}
```

在聊天室中执行操作

不同的用户能够在聊天室中执行的操作各不相同；例如，发送消息、删除消息或与用户断开连接。要执行其中一项操作，请对已连接的 ChatRoom 调用 perform(request:)，从而传递 SDK 中所提供的 ChatRequest 对象之一的实例。支持的请求位于 Request.swift 中。

当您的后端应用程序调用 CreateChatToken 时，在聊天室中执行的某些操作要求已连接的用户具有特定能力。在设计上，SDK 无法识别所连接用户具备的能力。因此，虽然您可以尝试在连接的 ChatRoom 实例中执行监管人操作，但控制面板 API 最终决定该操作是否会成功。

通过 room.perform(request:) 的所有操作都将等至聊天室收到的预期模型实例（其类型与请求对象本身相关联）与收到的模型和请求对象的 requestId 相匹配为止。如果请求存在问题，ChatRoom 将始终以 ChatError 的形式抛出错误。ChatError 的定义位于 Error.swift 中。

发送消息

要发送聊天消息，请使用 SendMessageRequest 的实例：

```
import AmazonIVSChatMessaging

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

如上所述，ChatRoom 收到相应的 ChatMessage 后就会返回 room.perform(request:)。如果请求存在问题（如超出聊天室的消息字符限制），则改为抛出 ChatError 实例。然后，您可以在 UI 中显示这些有用的信息：

```
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)
}
```

将元数据附加到消息

发送消息时，可以附加将与之关联的元数据。SendMessageRequest 具有 attributes 属性，可以使用该属性初始化请求。当其他人在聊天室中收到该消息时，您在此处附加的数据会附加到该消息中。

以下是将表情数据附加到正在发送的消息的示例：

```
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"
      ]
   )
)
```

在 SendMessageRequest 中使用 attributes 对于在聊天产品中构建复杂的功能非常有用。例如，可以使用 SendMessageRequest 中的 [String : String] 属性字典构建线程功能。

attributes 有效负载非常灵活且功能强大。使用它可以获取以其他方式所无法获取的消息的信息。例如，使用属性比解析消息字符串以获取诸如表情之类的信息要容易得多。

删除消息

删除聊天消息与发送聊天消息一样。对 ChatRoom 使用 room.perform(request:) 函数以通过创建 DeleteMessageRequest 实例来实现此目的。

要轻松访问以前收到的聊天消息实例，请将 message.id 的值传递给 DeleteMessageRequest 的初始化程序。

（可选）向 DeleteMessageRequest 提供一个原因字符串，以便在 UI 中显示该字符串。

```
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!"
   )
)
```

由于这是监管人执行的操作，因此您的用户实际上可能无法删除其他用户的消息。当用户在没有相应权限的情况下尝试删除消息时，您可以使用 Swift 的可抛出函数机制在 UI 中显示错误消息。

后端为用户调用 CreateChatToken 时，它需要将 "DELETE_MESSAGE" 传递到 capabilities 字段中，从而为已连接的聊天用户激活该功能。

以下示例说明了如何捕获在没有相应权限的情况下尝试删除消息时抛出的功能错误：

```
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)
}
```

断开与其他用户的连接

使用 room.perform(request:) 断开其他用户与聊天室的连接。具体来说，是使用 DisconnectUserRequest 的实例。ChatRoom 收到的所有 ChatMessage 均具有 sender 属性，其中包含您使用 DisconnectUserRequest 实例正确初始化所需的用户 ID。（可选）提供断开连接请求的原因字符串。

```
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
   )
)
```

由于这是监管人所执行操作的另一个示例，因此您可以尝试断开其他用户的连接，但除非您具有 DISCONNECT_USER 权限，否则无法断开连接。当后端应用程序调用 CreateChatToken 并将 "DISCONNECT_USER" 字符串注入 capabilities 字段时，您将获得该权限。

如果您的用户无权断开其他用户的连接，则 room.perform(request:) 会像抛出其他请求一样抛出 ChatError 实例。您可以检查错误的 errorCode 属性，以确定请求是否因缺少监管人权限而失败：

```
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 聊天功能客户端消息收发 SDK：iOS 教程

Amazon Interactive Video (IVS) Chat 客户端消息收发 iOS SDK 提供界面，可让您在使用 Apple Swift 编程语言的平台上整合我们的 IVS Chat 消息收发 API。

有关 Chat iOS SDK 教程，请参阅 https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/latest/tutorials/table-of-contents/。

IVS 聊天功能客户端消息收发 SDK：JavaScript 指南

Amazon Interactive Video（IVS）Chat 客户端消息收发 JavaScript SDK 可让您在使用 Web 浏览器的平台上轻松整合我们的 Amazon IVS 消息收发 API。

IVS Chat 客户端消息收发 JavaScript SDK 的最新版本：1.0.2（发布说明）

参考文档：有关 Amazon IVS Chat 客户端消息收发 JavaScript SDK 中最重要方法的信息，请参阅参考文档，网址为 https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/。

示例代码：有关使用 JavaScript SDK 的 Web 特定演示的信息，请参阅 GitHub 上的示例存储库：https://github.com/aws-samples/amazon-ivs-chat-web-demo

IVS 聊天功能客户端消息收发 JavaScript SDK 入门

在开始之前，应该熟悉 Amazon IVS Chat 入门。

添加程序包

请使用以下任一命令：

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

或者：

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

React Native 支持

IVS Chat 客户端消息收发 JavaScript SDK 具有使用 `crypto.getRandomValues` 方法的 `uuid` 依赖项。由于 React Native 中不支持此方法，因此您需要安装额外的填充代码 `react-native-get-random-value` 并在 `index.js` 文件顶部导入：

```
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);
```

设置您的后端

此集成需要服务器上的端点与 Amazon IVS Chat API 通信。使用官方亚马逊云科技库从服务器访问 Amazon IVS API。这些库可以从 node.js、java 和 go 等公共程序包以多种语言进行访问。

创建与 Amazon IVS Chat API CreateChatToken 端点通信的服务器端点，以为聊天用户创建聊天令牌。

使用 IVS 聊天功能客户端消息收发 JavaScript SDK

本文档将引导您完成使用 Amazon IVS 聊天功能客户端消息收发 JavaScript SDK 涉及的步骤。

初始化聊天室实例

创建 ChatRoom 类的实例。这需要传递 regionOrUrl（托管聊天室的 AWS 区域）和 tokenProvider（将在下一步中创建的令牌获取方式）：

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

令牌提供程序函数

创建从后端获取聊天令牌的异步令牌提供程序函数：

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

该函数不应接受任何参数，并返回包含聊天令牌对象的 Promise：

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

初始化 ChatRoom 对象需要此函数。使用从后端收到的值填写下方 <token> 和 <date-time> 字段：

```
// 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 you 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>")
  }
}
```

请记得将 tokenProvider 传递给 ChatRoom 构造函数。连接中断或会话过期时，ChatRoom 会刷新令牌。请勿在任意地方使用 tokenProvider 存储令牌；ChatRoom 会为您处理。

接收事件

下一步，订阅聊天室事件以接收生命周期事件以及聊天室中传送的消息和事件：

```
/**
* 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" }
  * }
  */
});
```

连接到聊天室

基本初始化的最后一步是通过建立 WebSocket 连接来连接到聊天室。为此，只需在聊天室实例中调用 `connect()` 方法：

```
room.connect();
```

SDK 将尝试与聊天室建立连接，该聊天室使用从服务器收到的聊天令牌进行编码。

调用 `connect()` 后，房间将转换到 `connecting` 状态并发出 `connecting` 事件。房间连接成功后，将转换到 `connected` 状态并发出 `connect` 事件。

获取令牌或连接到 WebSocket 时出现问题可能会导致连接失败。在这种情况下，房间会尝试自动重新连接，尝试连接的次数不超过 `maxReconnectAttempts` 构造函数参数所指示的次数。在尝试重新连

接期间，房间处于 connecting 状态，并不会发出其他事件。重新连接尝试次数耗尽后，房间会转换到 disconnected 状态并发出 disconnect 事件（带有相关的断开连接原因）。在 disconnected 状态下，房间不再尝试连接；您必须再次调用 connect() 才能触发连接进程。

在聊天室中执行操作

Amazon IVS Chat Messaging SDK 为用户提供发送消息、删除消息和与其他用户断开连接的操作。可在 ChatRoom 实例上执行这些操作。这些操作将返回 Promise 对象，该对象允许您接收请求确认或拒绝。

发送消息

对于此请求，您必须在聊天令牌中对 SEND_MESSAGE 功能进行编码。

触发发送消息请求：

```
const request = new SendMessageRequest('Test Echo');
room.sendMessage(request);
```

若要获得请求的确认或拒绝，await 返回的 Promise 或使用 then() 方法：

```
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.
}
```

删除消息

对于此请求，您必须在聊天令牌中对 DELETE_MESSAGE 功能进行编码。

若出于审核目的删除消息，请调用 deleteMessage() 方法：

```
const request = new DeleteMessageRequest(messageId, 'Reason for deletion');
room.deleteMessage(request);
```

若要获得请求的确认或拒绝，await 返回的 Promise 或使用 then() 方法：

```
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.
}
```

断开与其他用户的连接

对于此请求，您必须在聊天令牌中对 DISCONNECT_USER 功能进行编码。

若出于审核目的与其他用户断开连接，请调用 disconnectUser() 方法：

```
const request = new DisconnectUserRequest(userId, 'Reason for disconnecting user');
room.disconnectUser(request);
```

若要获得请求的确认或拒绝，await 返回的 Promise 或使用 then() 方法：

```
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.
}
```

断开与聊天室的连接

若要关闭与聊天室的连接，请在 room 实例上调用 disconnect() 方法：

```
room.disconnect();
```

调用此方法会导致房间按顺序关闭底层 WebSocket。房间实例转换到 disconnected 状态并发出断开连接事件，disconnect 原因设置为 "clientDisconnect"。

IVS 聊天功能客户端消息收发 SDK：JavaScript 教程第 1 部分：聊天室

这是一个由两部分组成的教程的第 1 部分。通过使用 JavaScript/TypeScript 构建功能齐全的应用程序，您将了解使用 Amazon IVS Chat 客户端消息收发 JavaScript SDK 的基本知识。我们把这个应用程序称为 Chatterbox。

目标受众是刚接触 Amazon IVS Chat 消息收发 SDK 的经验丰富的开发人员。您应该熟悉 JavaScript/TypeScript 编程语言和 React 库。

为简洁起见，我们将 Amazon IVS Chat 客户端消息收发 JavaScript SDK 称为 Chat JS SDK。

注意：在某些情况下，JavaScript 和 TypeScript 的代码示例是相同的，因此我们将它们合并在一起。

本教程的第 1 部分分为几个章节：

1. the section called “设置本地身份验证/授权服务器”
2. the section called “创建 Chatterbox 项目”
3. the section called “连接到聊天室”
4. the section called “构建令牌提供程序”
5. the section called “观察连接更新”
6. the section called “创建发送按钮组件”
7. the section called “创建消息输入”
8. the section called “后续步骤”

要学习完整的 SDK 文档，请从亚马逊 IVS 聊天功能客户端消息收发 SDK（在《Amazon IVS 聊天功能用户指南》中）和 Chat Client Messaging: SDK for JavaScript Reference（在 GitHub 上）开始。

先决条件

- 熟悉 JavaScript/TypeScript 和 React 库。如果不熟悉 React Native，请学习本井字游戏教程中的基础知识。
- 阅读并理解 IVS 聊天功能入门。
- 使用现有的 IAM policy 中定义的 CreateChatToken 和 CreateRoom 功能创建 AWS IAM 用户。（请参见 IVS 聊天功能入门。）
- 确保该用户的私有密钥/访问密钥存储在 Amazon 凭证文件中。有关说明，请参阅 Amazon CLI 用户指南（尤其是配置和凭证文件设置部分）。
- 创建聊天室并保存其 ARN。请参阅 IVS 聊天功能入门。（如果不保存 ARN，您可以稍后使用控制台或 Chat API 查找 ARN。）
- 使用 NPM 或 Yarn 程序包管理器安装 Node.js 14+ 环境。

设置本地身份验证/授权服务器

您的后端应用程序负责创建聊天室和生成 Chat JS SDK 所需的聊天令牌，以便在您的客户端连接聊天室时进行身份验证和授权。您必须使用自己的后端，因为您无法在移动应用程序中安全地存储 Amazon 密钥；老练的攻击者可以提取这些密钥并获得对您的 Amazon 账户的访问权限。

请参阅《Amazon IVS Chat 入门》中的创建聊天令牌。如其中的流程图所示，您的服务器端应用程序负责创建聊天令牌。这意味着您的应用程序必须通过向服务器端应用程序请求聊天令牌，来提供自己生成聊天令牌的方法。

在本节中，您将学习在后端创建令牌提供程序的基础知识。我们使用 Express 框架创建一个实时本地服务器，该服务器使用您的本地 Amazon 环境管理聊天令牌的创建。

使用 NPM 创建一个空的 npm 项目。创建一个用于存放应用程序的目录，并将其设为您的工作目录：

```
$ mkdir backend & cd backend
```

使用 `npm init` 为您的应用程序创建 `package.json` 文件：

```
$ npm init
```

此命令会提示您某些信息，包括您的应用程序的名称和版本。现在，只需按下 RETURN 即可接受其中大多数的默认值，但以下情况除外：

```
entry point: (index.js)
```

按下 RETURN 接受建议的默认文件名 `index.js`，或者输入任何您想要的主文件名。

现在安装所需的依赖项：

```
$ npm install express aws-sdk cors dotenv
```

`aws-sdk` 需要配置环境变量，这些变量会自动从根目录中名为 `.env` 的文件加载。要对其进行配置，请创建一个名为 `.env` 的新文件并填写缺少的配置信息：

```
# .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=...
```

现在，我们在根目录中创建一个入口点文件，其名称与您在上面的 npm init 命令中输入的名称相同。在本例中，我们使用 index.js 并导入所有必需的程序包：

```
// index.js
import express from 'express';
import AWS from 'aws-sdk';
import 'dotenv/config';
import cors from 'cors';
```

现在创建一个新的 express 实例：

```
const app = express();
const port = 3000;

app.use(express.json());
app.use(cors({ origin: ['http://127.0.0.1:5173'] }));
```

之后，您可以为令牌提供程序创建您的第一个端点 POST 方法。从请求正文中获取所需的参数（roomId、userId、capabilities 和 sessionDurationInMinutes）：

```
app.post('/create_chat_token', (req, res) => {
  const { roomIdentifier, userId, capabilities, sessionDurationInMinutes } = req.body
 || {};
});
```

添加必填字段的验证：

```
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;
  }
});
```

准备好 POST 方法后，我们将 createChatToken 与 aws-sdk 进行集成，以实现身份验证/授权的核心功能：

```
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 });
    }
  });
});
```

在文件末尾，为您的 express 应用程序添加一个端口侦听器：

```
app.listen(port, () => {
  console.log(`Backend listening on port ${port}`);
});
```

现在，您可以从项目的根目录使用以下命令运行服务器：

```
$ node index.js
```

提示：此服务器在 https://localhost:3000 接受 URL 请求。

创建 Chatterbox 项目

首先创建名为 chatterbox 的 React 项目。运行以下命令：

```
npx create-react-app chatterbox
```

您可以通过节点程序包管理器或 Yarn 程序包管理器集成 Chat 客户端消息收发 JS SDK：

- Npm：npm install amazon-ivs-chat-messaging
- Yarn：yarn add amazon-ivs-chat-messaging

连接到聊天室

在这里，您可以创建 ChatRoom 并使用异步方法对其进行连接。ChatRoom 类负责管理您的用户与 Chat JS SDK 的连接。要成功连接到聊天室，您必须在 React 应用程序中提供一个 ChatToken 实例。

导航到在默认 chatterbox 项目中创建的 App 文件，然后删除两个 标签之间的所有内容。不需要任何预填充的代码。此时，我们的 App 还很空。

```
// App.jsx / App.tsx

import * as React from 'react';

export default function App() {
  return <div>Hello!</div>;
}
```

创建一个新的 ChatRoom 实例并使用 useState 钩子将其传递给状态。该实例需要传递 regionOrUrl（托管聊天室的亚马逊云科技区域）和 tokenProvider（用于后续步骤中创建的后端身份验证/授权流程）。

重要提示：您必须使用与您在 Amazon IVS Chat 入门中创建聊天室的区域相同的亚马逊云科技区域。该 API 是一项亚马逊云科技区域服务。有关支持的区域和 Amazon IVS Chat HTTPS 服务终端节点的列表，请参阅 Amazon IVS Chat 区域页面。

```
// 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>;
}
```

构建令牌提供程序

下一步，我们需要构建 ChatRoom 构造函数所需的无参数 tokenProvider 函数。首先，我们将创建一个 fetchChatToken 函数，该函数将向您在 the section called “设置本地身份验证/授权服务器” 中设置的后端应用程序发出 POST 请求。聊天令牌包含该 SDK 成功建立聊天室连接所需的信息。Chat API 使用这些令牌作为验证用户身份、聊天室中的功能和会话持续时间的安全方式。

在项目导航器中，创建一个名为 fetchChatToken 的新 TypeScript/JavaScript 文件。构建对 backend 应用程序的提取请求，并从响应中返回 ChatToken 对象。添加创建聊天令牌所需的请求正文属性。使用为 Amazon 资源名称 (ARN) 定义的规则。这些属性记录在 CreateChatToken 端点中。

注意：您在此处使用的 URL 与运行后端应用程序时本地服务器创建的 URL 相同。

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),
  };
}
```

观察连接更新

对聊天室连接状态的变化做出反应是制作聊天应用程序的重要组成部分。让我们从订阅相关事件开始：

```
// 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>;
}
```

接下来，我们需要提供读取连接状态的功能。我们使用 useState 钩子在 App 中创建一些本地状态，并在每个侦听器内设置连接状态。

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>;
}
```

订阅连接状态后，显示连接状态并使用 useEffect 钩子内的 room.connect 方法连接到聊天室：

```
// 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>
);
```

```
// ...
```

您已成功实现聊天室连接。

创建发送按钮组件

在本节中，您将创建一个发送按钮，该按钮针对每种连接状态都有不同的设计。该发送按钮有助于在聊天室中发送消息。它还可以直观地指示是否/何时可以发送消息；例如，在连接断开或聊天会话过期的情况下。

首先，在 Chatterbox 项目的 `src` 目录中创建一个新文件并将其命名为 `SendButton`。然后，创建一个组件，该组件将显示聊天应用程序的按钮。导出您的 `SendButton` 并将其导入 `App`。在空的 `<div></div>` 中，添加 `<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>
);
```

接下来，在 App 中定义一个名为 onMessageSend 的函数，并将其传递给 SendButton onPress 属性。定义另一个名为 isSendDisabled 的变量（该变量可防止在未连接聊天室时发送消息），并将其传递给 SendButton disabled 属性。

```
// App.jsx / App.tsx

// ...

const onMessageSend = () => {};

const isSendDisabled = connectionState !== 'connected';
```

```
return (
  <div>
    <div>Connection State: {connectionState}</div>
    <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
  </div>
);

// ...
```

创建消息输入

Chatterbox 消息栏是您将与之交互以向聊天室发送消息的组件。通常，它包含用于编写消息的文本输入和用于发送消息的按钮。

要创建 MessageInput 组件，首先在 src 目录中创建一个新文件并将其命名为 MessageInput。然后，创建一个受控输入组件，该组件将显示聊天应用程序的输入。导出您的 MessageInput 并将其导入 App（在 <SendButton /> 上方）。

使用 useState 钩子创建一个名为 messageToSend 的新状态，其默认值为空字符串。在应用程序正文中，将 messageToSend 传递给 MessageInput 的 value，并将 setMessageToSend 传递给 onMessageChange 属性：

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>
);
```

后续步骤

现在，您已经为 Chatterbox 构建了一个消息栏，请继续阅读本 JavaScript 教程的第 2 部分消息和事件。

IVS 聊天功能客户端消息收发 SDK：JavaScript 教程第 2 部分：消息和事件

本教程的第 2 部分（也是最后一部分）分为几个章节：

1. the section called “订阅聊天消息事件”
2. the section called “显示收到的消息”
   a. the section called “创建消息组件”
   b. the section called “识别当前用户发送的消息”
   c. the section called “创建消息列表组件”
   d. the section called “呈现聊天消息列表”
3. the section called “在聊天室中执行操作”
   a. the section called “发送消息”
   b. the section called “删除消息”
4. the section called “后续步骤”

注意：在某些情况下，JavaScript 和 TypeScript 的代码示例是相同的，因此我们将它们合并在一起。

要学习完整的 SDK 文档，请从亚马逊 IVS 聊天功能客户端消息收发 SDK（在《Amazon IVS 聊天功能用户指南》中）和 Chat Client Messaging: SDK for JavaScript Reference（在 GitHub 上）开始。

先决条件

确保您已完成本教程的第 1 部分聊天室。

订阅聊天消息事件

当聊天室中发生事件时，ChatRoom 实例使用事件进行通信。要开始实现聊天体验，您需要向用户显示其他用户在他们连接的聊天室中发送消息的时间。

在这里，您订阅了聊天消息事件。稍后，我们将向您展示如何更新您创建的消息列表，该列表将随着每条消息/事件而更新。

在您的 App 中，请在 useEffect 钩子内订阅所有消息事件：

```
// App.tsx / App.jsx

useEffect(() => {
  // ...
  const unsubscribeOnMessageReceived = room.addListener('message', (message) => {});

  return () => {
    // ...
    unsubscribeOnMessageReceived();
  };
}, []);
```

显示收到的消息

接收消息是聊天体验的核心部分。通过使用 Chat JS SDK，您可以对代码进行设置，以轻松接收来自连接到聊天室的其他用户的事件。

稍后，我们将向您展示如何利用您在此处创建的组件在聊天室中执行操作。

在您的 App 中，使用名为 messages 的 ChatMessage 数组类型定义名为 messages 的状态：

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([]);

  //...
}
```

接下来，在 message 侦听器函数中，将 message 附加到 messages 数组中：

```
// App.jsx / App.tsx

// ...

const unsubscribeOnMessageReceived = room.addListener('message', (message) => {
  setMessages((msgs) => [...msgs, message]);
});

// ...
```

下面我们逐步完成显示收到的消息的任务：

1. the section called “创建消息组件”
2. the section called “识别当前用户发送的消息”
3. the section called “创建消息列表组件”

4. the section called “呈现聊天消息列表”

创建消息组件

Message 组件负责呈现您的聊天室收到的消息内容。在本节中，您将创建一个消息组件，用于在 App 中呈现单个聊天消息。

在 src 目录中创建一个新文件并将其命名为 Message。传入该组件的 ChatMessage 类型，然后从 ChatMessage 属性中传递 content 字符串，以显示从聊天室消息侦听器收到的消息文本。在项目导航器中，转到 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>
  );
};
```

提示：使用该组件存储您要在消息行中呈现的不同属性；例如，头像 URL、用户名和消息发送时的时间戳。

识别当前用户发送的消息

为了识别当前用户发送的消息，我们修改了代码并创建了 React 上下文来存储当前用户的 `userId`。

在 `src` 目录中创建一个新文件并将其命名为 `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>;
};
```

注意：这里我们使用了 useState 钩子来存储 userId 值。今后，您可以使用 setUserId 来更改用户上下文或用于登录目的。

接下来，使用之前创建的上下文替换传递给 tokenProvider 的第一个参数中的 userId：

```
// 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']),
      }),
  );

  // ...
}
```

在您的 Message 组件中，使用之前创建的 UserContext，声明 isMine 变量，将发件人的 userId 与上下文中的 userId 进行匹配，然后为当前用户应用不同样式的消息。

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>
  );
};
```

创建消息列表组件

MessageList 组件负责随时间显示聊天室的对话。MessageList 文件是存放所有消息的容器。Message 是 MessageList 中的一行。

在 src 目录中创建一个新文件并将其命名为 MessageList。使用类型为 ChatMessage 数组的 messages 定义 Props。在正文内，映射我们的 messages 属性并将 Props 传递给您的 Message 组件。

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>
  );
};
```

呈现聊天消息列表

现在，将新的 MessageList 导入您的主要 App 组件中：

```
// 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>
);

// ...
```

所有部分都已准备就绪，您的 App 可以开始呈现聊天室收到的消息。继续阅读下文，了解如何在聊天室中利用您创建的组件执行操作。

在聊天室中执行操作

在聊天室中发送消息和执行主持人操作是您与聊天室互动的一些主要方式。在这里，您将学习如何使用各种 ChatRequest 对象在 Chatterbox 中执行常见操作，例如发送消息、删除消息和断开其他用户的连接。

聊天室中的所有操作都遵循一个通用模式：对于您在聊天室中执行的每个操作，都有一个相应的请求对象。对于每个请求，您在请求确认时都会收到相应的响应对象。

只要在您创建聊天令牌时用户获得了正确的权限，他们就可以使用请求对象成功执行相应的操作，以查看您可以在聊天室中执行哪些请求。

下面，我们介绍如何发送消息和删除消息。

发送消息

SendMessageRequest 类允许在聊天室中发送消息。在这里，您可以修改您的 App，以使用您在创建消息输入（在本教程的第 1 部分）中创建的组件发送消息请求。

首先，使用 useState 钩子定义一个名为 isSending 的新布尔属性。使用这个新属性通过 isSendDisabled 常量切换 button HTML 元素的禁用状态。在 SendButton 的事件处理程序中，清除 messageToSend 的值并将 isSending 设置为 true。

由于您将通过此按钮进行 API 调用，因此添加 *isSending* 布尔值有助于防止同时发生多个 API 调用，方法是在请求完成之前禁用 *SendButton* 上的用户交互。

```
// App.jsx / App.tsx

// ...
```

```
const [isSending, setIsSending] = useState(false);

// ...

const onMessageSend = () => {
  setIsSending(true);
  setMessageToSend('');
};

// ...

const isSendDisabled = connectionState !== 'connected' || isSending;

// ...
```

通过创建一个新的 SendMessageRequest 实例并将消息内容传递给构造函数来准备请求。设置 isSending 和 messageToSend 状态后，调用 sendMessage 方法，将该请求发送到聊天室。最后，在收到确认或拒绝该请求的响应后清除 isSending 标志。

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);
  }
};

// ...
```

试运行 Chatterbox：尝试发送一条消息，方法是用 MessageInput 编写一条消息，然后点击 SendButton。您应该会看到您发送的消息在之前创建的 MessageList 中呈现。

删除消息

要从聊天室删除消息，您需要具有适当的权限。权限是在聊天令牌（向聊天室进行身份验证时需要使用该令牌）的初始化期间授予的。就本节而言，设置本地身份验证/授权服务器（在本教程的第 1 部分）中的 ServerApp 允许您指定主持人权限。这是在您的应用程序中使用您在构建令牌提供程序（也在第 1 部分）中创建的 tokenProvider 对象完成的。

在这里，您可以通过添加删除消息的函数来修改您的 Message。

首先，打开 App.tsx 并添加 DELETE_MESSAGE 权限。（capabilities 是 tokenProvider 函数的第二个参数。）

注意：ServerApp 通过这种方式通知 IVS Chat API，与生成的聊天令牌相关联的用户可以删除聊天室中的消息。在实际情况中，您可能需要更复杂的后端逻辑来管理服务器应用程序基础架构中的用户权限。

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']),
  }),
);

// ...
```

在接下来的步骤中，您将更新 Message 以显示删除按钮。

打开 Message 并使用初始值为 false 的 useState 钩子定义一个名为 isDeleting 的新布尔状态。使用此状态，将 Button 的内容更新为因 isDeleting 的当前状态而异。当 isDeleting 为 true 时禁用您的按钮；这样可以防止您尝试同时发出两个删除消息请求。

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>
  );
};
```

定义一个名为 onDelete 的新函数，该函数接受字符串作为其参数之一并返回 Promise。在 Button 操作闭包的正文中，使用 setIsDeleting 在调用 onDelete 之前和之后切换 isDeleting 布尔值。对于字符串参数，传入您的组件消息 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>
  );
};
```

接下来，更新 MessageList 以反映对 Message 组件的最新更改。

打开 MessageList 并定义一个名为 onDelete 的新函数，该函数接受字符串作为参数并返回 Promise。更新您的 Message 并将其传递给 Message 的属性。新闭包中的字符串参数将是您要删除的消息的 ID，该消息是从您的 Message 中传递的。

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} />
      ))}
    </>
```

```
  );
};
```

接下来，更新 App 以反映对 MessageList 的最新更改。

在 App 中，定义一个名为 onDeleteMessage 的函数，并将其传递给 MessageList onDelete 属性：

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>
);

// ...
```

通过创建一个新的 DeleteMessageRequest 实例并将相关的消息 ID 传递给构造函数参数来准备请求，然后调用接受上述准备好的请求的 deleteMessage：

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);
};

// ...
```

接下来，更新 messages 状态以反映新的消息列表，该列表忽略了您刚刚删除的消息。

在 useEffect 钩子中，侦听 messageDelete 事件，并通过删除 ID 与 message 参数相匹配的消息来更新 messages 状态数组。

注意：当消息被当前用户或聊天室中的任何其他用户删除时，可能会引发 messageDelete 事件。如果在事件处理程序中（而不是 deleteMessage 请求旁边）处理该事件，您可以统一处理删除消息。

```
// App.jsx / App.tsx

// ...

const unsubscribeOnMessageDeleted = room.addListener('messageDelete',
 (deleteMessageEvent) => {
  setMessages((prev) => prev.filter((message) => message.id !==
 deleteMessageEvent.id));
});

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

现在，您可以从聊天应用程序的聊天室中删除用户。

后续步骤

作为实验，尝试在聊天室中执行其他操作，例如断开其他用户的连接。

IVS 聊天功能客户端消息收发 SDK：React Native 教程第 1 部分：聊天室

这是一个由两部分组成的教程的第 1 部分。您将通过使用 React Native 构建功能齐全的应用程序，了解使用 Amazon IVS 聊天功能客户端消息收发 JavaScript SDK 的基本知识。我们把这个应用程序称为 Chatterbox。

目标受众是刚接触 Amazon IVS Chat 消息收发 SDK 的经验丰富的开发人员。您需要熟悉 TypeScript 或 JavaScript 编程语言以及 React Native 库。

为简洁起见，我们将 Amazon IVS Chat 客户端消息收发 JavaScript SDK 称为 Chat JS SDK。

注意：在某些情况下，JavaScript 和 TypeScript 的代码示例是相同的，因此我们将它们合并在一起。

本教程的第 1 部分分为几个章节：

1. the section called “设置本地身份验证/授权服务器”
2. the section called “创建 Chatterbox 项目”
3. the section called “连接到聊天室”
4. the section called “构建令牌提供程序”
5. the section called “观察连接更新”
6. the section called “创建发送按钮组件”
7. the section called “创建消息输入”
8. the section called “后续步骤”

先决条件

- 熟悉 TypeScript 或 JavaScript 以及 React Native 库。如果您不熟悉 React Native，请访问 Intro to React Native（React Native 简介）学习基础知识。
- 阅读并理解 IVS 聊天功能入门。
- 使用现有的 IAM policy 中定义的 CreateChatToken 和 CreateRoom 功能创建 AWS IAM 用户。（请参见 IVS 聊天功能入门。）
- 确保该用户的私有密钥/访问密钥存储在 Amazon 凭证文件中。有关说明，请参阅 Amazon CLI 用户指南（尤其是配置和凭证文件设置部分）。
- 创建聊天室并保存其 ARN。请参阅 IVS 聊天功能入门。（如果不保存 ARN，您可以稍后使用控制台或 Chat API 查找 ARN。）
- 使用 NPM 或 Yarn 程序包管理器安装 Node.js 14+ 环境。

设置本地身份验证/授权服务器

您的后端应用程序负责创建聊天室和生成 Chat JS SDK 所需的聊天令牌，以便在您的客户端连接聊天室时进行身份验证和授权。您必须使用自己的后端，因为您无法在移动应用程序中安全地存储 Amazon 密钥；老练的攻击者可以提取这些密钥并获得对您的 Amazon 账户的访问权限。

请参阅《Amazon IVS Chat 入门》中的创建聊天令牌。如其中的流程图所示，您的服务器端应用程序负责创建聊天令牌。这意味着您的应用程序必须通过向服务器端应用程序请求聊天令牌，来提供自己生成聊天令牌的方法。

在本节中，您将学习在后端创建令牌提供程序的基础知识。我们使用 Express 框架创建一个实时本地服务器，该服务器使用您的本地 Amazon 环境管理聊天令牌的创建。

使用 NPM 创建一个空的 npm 项目。创建一个用于存放应用程序的目录，并将其设为您的工作目录：

```
$ mkdir backend & cd backend
```

使用 npm init 为您的应用程序创建 package.json 文件：

```
$ npm init
```

此命令会提示您某些信息，包括您的应用程序的名称和版本。现在，只需按下 RETURN 即可接受其中大多数的默认值，但以下情况除外：

```
entry point: (index.js)
```

按下 RETURN 接受建议的默认文件名 index.js，或者输入任何您想要的主文件名。

现在安装所需的依赖项：

```
$ npm install express aws-sdk cors dotenv
```

aws-sdk 需要配置环境变量，这些变量会自动从根目录中名为 .env 的文件加载。要对其进行配置，请创建一个名为 .env 的新文件并填写缺少的配置信息：

```
# .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=...
```

现在，我们在根目录中创建一个入口点文件，其名称与您在上面的 npm init 命令中输入的名称相同。在本例中，我们使用 index.js 并导入所有必需的程序包：

```
// index.js
import express from 'express';
import AWS from 'aws-sdk';
import 'dotenv/config';
import cors from 'cors';
```

现在创建一个新的 express 实例：

```
const app = express();
const port = 3000;

app.use(express.json());
app.use(cors({ origin: ['http://127.0.0.1:5173'] }));
```

之后，您可以为令牌提供程序创建您的第一个端点 POST 方法。从请求正文中获取所需的参数（roomId、userId、capabilities 和 sessionDurationInMinutes）：

```
app.post('/create_chat_token', (req, res) => {
  const { roomIdentifier, userId, capabilities, sessionDurationInMinutes } = req.body
 || {};
});
```

添加必填字段的验证：

```
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;
  }
});
```

准备好 POST 方法后，我们将 createChatToken 与 aws-sdk 进行集成，以实现身份验证/授权的核心功能：

```
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 });
    }
  });
});
```

在文件末尾，为您的 express 应用程序添加一个端口侦听器：

```
app.listen(port, () => {
  console.log(`Backend listening on port ${port}`);
});
```

现在，您可以从项目的根目录使用以下命令运行服务器：

```
$ node index.js
```

提示：此服务器在 https://localhost:3000 接受 URL 请求。

创建 Chatterbox 项目

首先创建名为 chatterbox 的 React Native 项目。运行以下命令：

```
npx create-expo-app
```

或者使用 TypeScript 模板创建一个 expo 项目。

```
npx create-expo-app -t expo-template-blank-typescript
```

您可以通过节点程序包管理器或 Yarn 程序包管理器集成 Chat 客户端消息收发 JS SDK：

- Npm：`npm install amazon-ivs-chat-messaging`
- Yarn：`yarn add amazon-ivs-chat-messaging`

连接到聊天室

在这里，您可以创建 ChatRoom 并使用异步方法对其进行连接。ChatRoom 类负责管理您的用户与 Chat JS SDK 的连接。要成功连接到聊天室，您必须在 React 应用程序中提供一个 ChatToken 实例。

导航到在默认 chatterbox 项目中创建的 App 文件，并删除功能组件返回的所有内容。不需要任何预填充的代码。此时，我们的 App 还很空。

TypeScript/JavaScript：

```
// App.tsx / App.jsx

import * as React from 'react';
import { Text } from 'react-native';

export default function App() {
  return <Text>Hello!</Text>;
}
```

创建一个新的 ChatRoom 实例并使用 useState 钩子将其传递给状态。该实例需要传递 regionOrUrl（托管聊天室的亚马逊云科技区域）和 tokenProvider（用于后续步骤中创建的后端身份验证/授权流程）。

重要提示：您必须使用与您在 Amazon IVS Chat 入门中创建聊天室的区域相同的亚马逊云科技区域。该 API 是一项亚马逊云科技区域服务。有关支持的区域和 Amazon IVS Chat HTTPS 服务终端节点的列表，请参阅 Amazon IVS Chat 区域页面。

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>;
}
```

构建令牌提供程序

下一步，我们需要构建 ChatRoom 构造函数所需的无参数 tokenProvider 函数。首先，我们将创建一个 fetchChatToken 函数，该函数将向您在 the section called “设置本地身份验证/授权服务器” 中设置的后端应用程序发出 POST 请求。聊天令牌包含该 SDK 成功建立聊天室连接所需的信息。Chat API 使用这些令牌作为验证用户身份、聊天室中的功能和会话持续时间的安全方式。

在项目导航器中，创建一个名为 fetchChatToken 的新 TypeScript/JavaScript 文件。构建对 backend 应用程序的提取请求，并从响应中返回 ChatToken 对象。添加创建聊天令牌所需的请求正文属性。使用为 Amazon 资源名称 (ARN) 定义的规则。这些属性记录在 CreateChatToken 端点中。

注意：您在此处使用的 URL 与运行后端应用程序时本地服务器创建的 URL 相同。

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),
  };
}
```

观察连接更新

对聊天室连接状态的变化做出反应是制作聊天应用程序的重要组成部分。让我们从订阅相关事件开始：

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>;
}
```

接下来，我们需要提供读取连接状态的功能。我们使用 useState 钩子在 App 中创建一些本地状态，并在每个侦听器内设置连接状态。

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>;
}
```

订阅连接状态后，显示连接状态并使用 useEffect 钩子内的 room.connect 方法连接到聊天室：

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,
  }
});

// ...
```

您已成功实现聊天室连接。

创建发送按钮组件

在本节中，您将创建一个发送按钮，该按钮针对每种连接状态都有不同的设计。该发送按钮有助于在聊天室中发送消息。它还可以直观地指示是否/何时可以发送消息；例如，在连接断开或聊天会话过期的情况下。

首先，在 Chatterbox 项目的 `src` 目录中创建一个新文件并将其命名为 `SendButton`。然后，创建一个组件，该组件将显示聊天应用程序的按钮。导出您的 `SendButton` 并将其导入 `App`。在空的 `<View></View>` 中，添加 `<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>
);
```

接下来，在 App 中定义一个名为 onMessageSend 的函数，并将其传递给 SendButton onPress 属性。定义另一个名为 isSendDisabled 的变量（该变量可防止在未连接聊天室时发送消息），并将其传递给 SendButton disabled 属性。

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>
);

// ...
```

创建消息输入

Chatterbox 消息栏是您将与之交互以向聊天室发送消息的组件。通常，它包含用于编写消息的文本输入和用于发送消息的按钮。

要创建 MessageInput 组件，首先在 src 目录中创建一个新文件并将其命名为 MessageInput。然后创建一个输入组件，这将会显示聊天应用程序的输入。导出您的 MessageInput 并将其导入 App（在 <SendButton /> 上方）。

使用 useState 钩子创建一个名为 messageToSend 的新状态，其默认值为空字符串。在应用程序正文中，将 messageToSend 传递给 MessageInput 的 value，并将 setMessageToSend 传递给 onMessageChange 属性：

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',
  }
});
```

后续步骤

现在，您已经为 Chatterbox 构建了一个消息栏，请继续阅读本 React Native 教程第 2 部分：消息和事件。

IVS 聊天功能客户端消息收发 SDK：React Native 教程第 2 部分：消息和事件

本教程的第 2 部分（也是最后一部分）分为几个章节：

1. the section called “订阅聊天消息事件”
2. the section called “显示收到的消息”
   a. the section called “创建消息组件”
   b. the section called “识别当前用户发送的消息”
   c. the section called “呈现聊天消息列表”
3. the section called “在聊天室中执行操作”
   a. the section called “发送消息”
   b. the section called “删除消息”
4. the section called “后续步骤”

注意：在某些情况下，JavaScript 和 TypeScript 的代码示例是相同的，因此我们将它们合并在一起。

先决条件

确保您已完成本教程的第 1 部分聊天室。

订阅聊天消息事件

当聊天室中发生事件时，ChatRoom 实例使用事件进行通信。要开始实现聊天体验，您需要向用户显示其他用户在他们连接的聊天室中发送消息的时间。

在这里，您订阅了聊天消息事件。稍后，我们将向您展示如何更新您创建的消息列表，该列表将随着每条消息/事件而更新。

在您的 App 中，请在 `useEffect` 钩子内订阅所有消息事件：

TypeScript/JavaScript：

```
// App.tsx / App.jsx

useEffect(() => {
  // ...
  const unsubscribeOnMessageReceived = room.addListener('message', (message) => {});

  return () => {
    // ...
    unsubscribeOnMessageReceived();
  };
}, []);
```

显示收到的消息

接收消息是聊天体验的核心部分。通过使用 Chat JS SDK，您可以对代码进行设置，以轻松接收来自连接到聊天室的其他用户的事件。

稍后，我们将向您展示如何利用您在此处创建的组件在聊天室中执行操作。

在您的 App 中，使用名为 `messages` 的 `ChatMessage` 数组类型定义名为 `messages` 的状态：

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([]);

  //...
}
```

接下来，在 message 侦听器函数中，将 message 附加到 messages 数组中：

TypeScript/JavaScript：

```
// App.tsx / App.jsx

// ...

const unsubscribeOnMessageReceived = room.addListener('message', (message) => {
  setMessages((msgs) => [...msgs, message]);
});

// ...
```

下面我们逐步完成显示收到的消息的任务：

1. the section called “创建消息组件”

2. the section called “识别当前用户发送的消息”
3. the section called “呈现聊天消息列表”

创建消息组件

Message 组件负责呈现您的聊天室收到的消息内容。在本节中，您将创建一个消息组件，用于在 App 中呈现单个聊天消息。

在 src 目录中创建一个新文件并将其命名为 Message。传入该组件的 ChatMessage 类型，然后从 ChatMessage 属性中传递 content 字符串，以显示从聊天室消息侦听器收到的消息文本。在项目导航器中，转到 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,
  },
});
```

提示：使用该组件存储您要在消息行中呈现的不同属性；例如，头像 URL、用户名和消息发送时的时间戳。

识别当前用户发送的消息

为了识别当前用户发送的消息，我们修改了代码并创建了 React 上下文来存储当前用户的 `userId`。

在 `src` 目录中创建一个新文件并将其命名为 `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;
```

注意：这里我们使用了 useState 钩子来存储 userId 值。今后，您可以使用 setUserId 来更改用户上下文或用于登录目的。

然后使用之前创建的上下文替换传递给 tokenProvider 的第一个参数中的 userId。务必要按下文的规定，将 SEND_MESSAGE 功能添加到您的令牌提供者；发送消息将需要此功能：

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']),
      }),
  );

  // ...
}
```

在您的 Message 组件中，使用之前创建的 UserContext，声明 isMine 变量，将发件人的 userId 与上下文中的 userId 进行匹配，然后为当前用户应用不同样式的消息。

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',
  },
});
```

呈现聊天消息列表

现在使用 `FlatList` 和 `Message` 组件列出消息：

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>
);

// ...
```

所有部分都已准备就绪，您的 App 可以开始呈现聊天室收到的消息。继续阅读下文，了解如何在聊天室中利用您创建的组件执行操作。

在聊天室中执行操作

发送消息和执行主持人操作是您与聊天室互动的一些主要方式。在这里，您将学习如何使用各种聊天请求对象在 Chatterbox 中执行常见操作，例如发送消息、删除消息和断开其他用户的连接。

聊天室中的所有操作都遵循一个通用模式：对于您在聊天室中执行的每个操作，都有一个相应的请求对象。对于每个请求，您在请求确认时都会收到相应的响应对象。

只要您在创建聊天令牌时向用户授予了正确的功能，他们就可以使用请求对象成功执行相应的操作，以查看您可以在聊天室中执行的请求。

下面，我们介绍如何发送消息和删除消息。

发送消息

SendMessageRequest 类允许在聊天室中发送消息。在这里，您可以修改您的 App，以使用您在创建消息输入（在本教程的第 1 部分）中创建的组件发送消息请求。

首先，使用 useState 钩子定义一个名为 isSending 的新布尔属性。借助这一新属性可使用 isSendDisabled 常量切换 button 元素的禁用状态。在 SendButton 的事件处理程序中，清除 messageToSend 的值并将 isSending 设置为 true。

由于您将通过此按钮进行 API 调用，因此添加 *isSending* 布尔值有助于防止同时发生多个 API 调用，方法是在请求完成之前禁用 *SendButton* 上的用户交互。

注意：只有当您向令牌提供者添加了该 SEND_MESSAGE 功能时，发送消息才能正常工作，详见上文识别当前用户发送的消息。

TypeScript/JavaScript：

```
// App.tsx / App.jsx

// ...

const [isSending, setIsSending] = useState(false);

// ...

const onMessageSend = () => {
  setIsSending(true);
  setMessageToSend('');
};

// ...

const isSendDisabled = connectionState !== 'connected' || isSending;

// ...
```

通过创建一个新的 SendMessageRequest 实例并将消息内容传递给构造函数来准备请求。设置 isSending 和 messageToSend 状态后，调用 sendMessage 方法，将该请求发送到聊天室。最后，在收到确认或拒绝该请求的响应后清除 isSending 标志。

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);
  }
};

// ...
```

试运行 Chatterbox：尝试发送一条消息，方法是用 MessageBar 编写一条消息，然后点击 SendButton。您应该会看到您发送的消息在之前创建的 MessageList 中呈现。

删除消息

要从聊天室删除消息，您需要具有适当的权限。权限是在聊天令牌（向聊天室进行身份验证时需要使用该令牌）的初始化期间授予的。就本节而言，设置本地身份验证/授权服务器（在本教程的第 1 部分）中的 ServerApp 允许您指定主持人权限。这是在您的应用程序中使用您在构建令牌提供程序（也在第 1 部分）中创建的 tokenProvider 对象完成的。

在这里，您可以通过添加删除消息的函数来修改您的 Message。

首先，打开 App.tsx 并添加 DELETE_MESSAGE 权限。（capabilities 是 tokenProvider 函数的第二个参数。）

注意：ServerApp 通过这种方式通知 IVS Chat API，与生成的聊天令牌相关联的用户可以删除聊天室中的消息。在实际情况中，您可能需要更复杂的后端逻辑来管理服务器应用程序基础架构中的用户权限。

TypeScript/JavaScript：

```
// App.tsx / App.jsx

// ...

const [room] = useState(() =>
    new ChatRoom({
      regionOrUrl: process.env.REGION,
      tokenProvider: () => tokenProvider(userId, ['SEND_MESSAGE', 'DELETE_MESSAGE']),
    }),
);

// ...
```

在接下来的步骤中，您将更新 Message 以显示删除按钮。

定义一个名为 onDelete 的新函数，该函数接受字符串作为其参数之一并返回 Promise。对于字符串参数，传入您的组件消息 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',
  },
});
```

接下来，更新 renderItem 以反映对 FlatList 组件的最新更改。

在 App 中，定义一个名为 handleDeleteMessage 的函数，并将其传递给 MessageList onDelete 属性：

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]);

// ...
```

通过创建一个新的 DeleteMessageRequest 实例并将相关的消息 ID 传递给构造函数参数来准备请求，然后调用接受上述准备好的请求的 deleteMessage：

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);
};

// ...
```

接下来，更新 messages 状态以反映新的消息列表，该列表忽略了您刚刚删除的消息。

在 useEffect 钩子中，侦听 messageDelete 事件，并通过删除 ID 与 message 参数相匹配的消息来更新 messages 状态数组。

注意：当消息被当前用户或聊天室中的任何其他用户删除时，可能会引发 messageDelete 事件。如果在事件处理程序中（而不是 deleteMessage 请求旁边）处理该事件，您可以统一处理删除消息。

TypeScript/JavaScript：

```
// App.tsx / App.jsx

// ...

const unsubscribeOnMessageDeleted = room.addListener('messageDelete',
 (deleteMessageEvent) => {
  setMessages((prev) => prev.filter((message) => message.id !==
 deleteMessageEvent.id));
});

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

现在，您可以从聊天应用程序的聊天室中删除用户。

后续步骤

作为实验，尝试在聊天室中执行其他操作，例如断开其他用户的连接。

IVS 聊天功能客户端消息收发 SDK：React 和 React Native 最佳实践

此文档说明了使用适用于 React 和 React Native 的 Amazon IVS 聊天功能消息收发 SDK 的最重要实践。利用这些信息，您应该能够在 React 应用程序中构建典型的聊天功能，此外还包括相关的背景知识，以帮助您更深入了解 IVS 聊天功能消息收发 SDK 的更高级部分。

创建聊天室初始化程序挂钩

`ChatRoom` 类包含核心的聊天方法和侦听器，用于管理连接状态和侦听各种事件，例如已收到消息和已删除消息。我们将在本教程中演示如何在挂钩中正确地存储聊天实例。

实施

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 };
};
```

注意：我们不会使用来自 setState 挂钩的 dispatch 方法，因为您无法即时更新配置参数。SDK 会一次性创建实例，并且无法更新令牌提供者。

重要提示：一次使用 ChatRoom 初始化程序挂钩来初始化一个新的聊天室实例。

示例

TypeScript/JavaScript：

```
// ...

const MyChatScreen = () => {
  const userId = 'Mike';
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(ROOM_ID, ['SEND_MESSAGE']),
```

```
  });

  const handleConnect = () => {
    room.connect();
  };

  // ...
};

// ...
```

侦听连接状态

您还可以在聊天室挂钩中订阅连接状态更新。

实施

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 };
};
```

聊天室实例提供者

要在其他组件中使用挂钩（以避免道具钻取），您可以使用 React `context` 创建聊天室提供者。

实施

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;
```

示例

创建 ChatRoomProvider 后，您可以通过 useChatRoomContext 来使用您的实例。

重要提示：请仅在您需要访问聊天屏幕和中间的其他组件之间的 context 时，才将提供者置于根级别，以避免在侦听连接时进行不必要的重新渲染。否则，应将提供者放置在尽可能靠近聊天屏幕的位置。

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();
  };
  // ...
};

// ...
```

创建消息侦听器

要及时了解所有传入的消息，您需要订阅 message 和 deleteMessage 事件。以下是一些为您的组件提供聊天消息的代码。

重要提示：为确保理想的性能，我们将 ChatMessageContext 与 ChatRoomProvider 分开，因为当聊天消息侦听器更新其消息状态时，我们可能会进行多次重新渲染。请记住要在您要使用 ChatMessageProvider 的组件中应用 ChatMessageContext。

实施

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>;
};
```

React 示例

重要提示：请记住要用 ChatMessagesProvider 打包您的消息容器。Message 行是一个显示消息内容的示例组件。

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>
  );
};
```

React Native 示例

`ChatMessage` 会默认包含 `id`，后者将在 `FlatList` 中自动用作每行的 React 键；因此，您无需传递 `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} />;
};
```

一个应用程序中的多个聊天室实例

如果您在应用程序中使用多个并发聊天室，我们建议为每个聊天创建一个提供者，并在聊天提供者中使用该提供者。在此例中，我们将创建一个帮助机器人和客户帮助聊天。我们将为两者分别创建了一个提供者。

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>;
};
```

React 示例

您现在可以使用具有相同 ChatRoomProvider 的不同聊天提供者。稍后您可以在每个屏幕/视图中重复使用相同的 useChatRoomContext。

TypeScript/JavaScript：

```
// App.tsx / App.jsx

const App = () => {
  return (
    <Routes>
      <Route
        element={
          <SupportChatProvider>
            <SupportChatScreen />
          </SupportChatProvider>
        }
      />
      <Route
        element={
          <SalesChatProvider>
            <SalesChatScreen />
          </SalesChatProvider>
        }
      />
    </Routes>
  );
};
```

React Native 示例

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 聊天功能安全性

AWS 十分重视云安全性。作为 AWS 客户，您将从专为满足大多数安全敏感型组织的要求而打造的数据中心和网络架构中受益。

安全性是 AWS 和您的共同责任。责任共担模式将其描述为云的 安全性和云中 的安全性：

- 云的安全性 – AWS 负责保护在 AWS 云中运行 AWS 服务的基础设施。AWS 还向您提供可安全使用的服务。作为 AWS 合规性计划的一部分，第三方审核人员将定期测试和验证安全性的有效性。
- 云中的安全性 – 您的责任由您使用的 AWS 服务决定。您还需要对其他因素负责，包括您的数据的敏感性、您组织的要求以及适用的法律法规。

此文档将帮助您了解如何在使用 Amazon IVS 聊天功能时应用责任共担模式。以下主题说明如何配置 Amazon IVS 聊天功能以实现您的安全性和合规性目标。

主题

IVS 聊天功能数据保护

对于发送到 Amazon Interactive Video Service（IVS）聊天功能的数据，将应用以下数据保护措施：

- Amazon IVS Chat 流量使用 WSS 来确保传输过程中的数据安全。
- Amazon IVS Chat 令牌使用 KMS 客户管理的密钥进行加密。

Amazon IVS 聊天功能不会要求您提供任何客户（最终用户）数据。当需要您提供客户（最终用户）数据时，聊天室、输入或输入安全组中没有任何字段。

请勿将敏感的识别信息（如客户（最终用户）账号）放入自由格式字段（如 Name（名称）字段）。在使用 Amazon IVS 控制台或 API、AWS CLI 或 AWS SDK 的情况下也不例外。您输入到 Amazon IVS 聊天功能中的任何数据都可能包含在诊断日志中。

流不是端到端加密的；流可以在 IVS 网络内部以不加密形式传输，以供处理。

IVS 聊天功能中的身份和访问管理

AWS Identity and Access Management（IAM）是一项 AWS 服务，可以帮助账户管理员安全地控制对 AWS 资源的访问。请参阅《IVS Low-Latency Streaming User Guide》中的 Identity and Access Management in IVS。

受众

如何使用 IAM 因您可以在 Amazon IVS 中执行的操作而异。请参阅《IVS Low-Latency Streaming User Guide》中的 Audience。

Amazon IVS 如何与 IAM 配合使用

在您发出 Amazon IVS API 请求之前，您必须创建一个或多个 IAM身份（用户、组和角色）和 IAM策略，然后向身份附加策略。传播权限最多需要几分钟时间；在此之前，API 请求会被拒绝。

要简要了解 Amazon IVS 如何与 IAM 结合使用，请参阅《IAM 用户指南》中的 与 IAM 配合使用的 AWS 服务。

身份

您可以创建 IAM 身份，以便为向您的 AWS 账户中的人员和进程提供身份验证。IAM 组是 IAM 用户的集合，可以将其作为一个单位进行管理。请参阅《IAM 用户指南》中的身份（用户、组和角色）。

策略

策略是由元素组成的 JSON 权限策略文档。请参阅《IVS Low-Latency Streaming User Guide》中的 Policies。

Amazon IVS 聊天功能支持三个元素：

- 操作：Amazon IVS 聊天功能的策略操作在操作前使用 `ivschat` 前缀。例如，要授予某人使用 Amazon IVS 聊天功能 `CreateRoom` API 方法创建 Amazon IVS 聊天室的权限，您应针对此人员将 `ivschat:CreateRoom` 操作纳入其策略中。策略语句必须包含 `Action` 或 `NotAction` 元素。

- 资源：Amazon IVS 聊天室资源采用以下 ARN 格式：

```
arn:aws:ivschat:${Region}:${Account}:room/${roomId}
```

例如，要在语句中指定 VgNkEJgOVX9N 聊天室，请使用以下 ARN：

```
"Resource": "arn:aws:ivschat:us-west-2:123456789012:room/VgNkEJgOVX9N"
```

无法对特定资源执行某些 Amazon IVS 聊天功能操作，例如，用于创建资源的操作。在这些情况下，您必须使用通配符 (*)：

```
"Resource":"*"
```

- 条件：Amazon IVS 聊天功能支持部分全局条件键：`aws:RequestTag`、`aws:TagKeys` 和 `aws:ResourceTag`。

在策略中，您可以使用变量作为占位符。例如，只有在使用用户的 IAM 用户名标记 IAM 用户时，您才能为其授予访问资源的权限。请参阅 IAM 用户指南中的变量和标签。

Amazon IVS 提供 AWS 托管式策略，可用于向身份授予一组预配置的权限（只读或完全访问权限）。您可以选择使用托管式策略，而不是下面所示的基于身份的策略。有关详细信息，请参阅 Managed Policies for Amazon IVS Chat。

基于 Amazon IVS 标签的授权

您可以将标签附加到 Amazon IVS 聊天功能资源，或者在请求中将标签传递给 Amazon IVS 聊天功能。要基于标签控制访问，您需要使用 `aws:ResourceTag/key-name`、`aws:RequestTag/key-name` 或 `aws:TagKeys` 条件键在策略的条件元素中提供标签信息。有关标记 Amazon IVS 聊天功能资源的更多信息，请参阅 IVS Chat API Reference 中的“Tagging”。

角色

请参阅《IAM 用户指南》中的 IAM 角色和临时安全凭证。

IAM 角色是 AWS 账户中具有特定权限的实体。

Amazon IVS 支持使用临时安全凭证。您可以使用临时凭证进行联合身份登录，担任 IAM 角色或担任跨账户角色。您可以调用 AWS Security Token Service API 操作（如 `AssumeRole` 或 `GetFederationToken`）以获取临时安全凭证。

特权访问和非特权访问

API 资源具有特权访问权限。非特权播放访问权限可通过私有通道进行设置；请参阅 Setting Up IVS Private Channels。

使用策略的最佳实践

请参阅 IAM 用户指南中的 IAM 最佳实践。

基于身份的策略非常强大。它们确定某个人是否可以创建、访问或删除您账户中的 Amazon IVS 资源。这些操作可能会使 AWS 账户产生成本。请遵循以下建议：

- 授予最低权限 – 创建自定义策略时，仅授予执行任务所需的许可。最开始只授予最低权限，然后根据需要授予其他权限。这样做比起一开始就授予过于宽松的权限而后再尝试收紧权限来说更为安全。具体而言，预留 `ivschat:*` 供管理员访问使用；请勿在应用程序中使用它。
- 为敏感操作启用多重身份验证（MFA）– 为了提高安全性，要求 IAM 用户使用 MFA 访问敏感资源或 API 操作。
- 使用策略条件来增强安全性 – 在切实可行的范围内，定义基于身份的策略在哪些情况下允许访问资源。例如，您可编写条件来指定请求必须来自允许的 IP 地址范围。您也可以编写条件，以便仅允许指定日期或时间范围内的请求，或者要求使用 SSL 或 MFA。

基于身份的策略示例

使用 Amazon IVS 控制台

要访问 Amazon IVS 控制台，您必须具有最低权限集，以允许您列出和查看有关 AWS 账户中 Amazon IVS 聊天功能资源的详细信息。如果创建比必需的最低权限更为严格的基于身份的策略，对于附加了该策略的实体，控制台将无法按预期正常运行。为确保对 Amazon IVS 控制台的访问权限，请将以下策略附加到身份（参阅《IAM 用户指南》中的添加和删除 IAM 权限）。

以下策略的各个部分提供了对以下内容的访问权限：

- 所有 Amazon IVS 聊天功能 API 端点
- 您的 Amazon IVS 聊天功能服务限额
- 列出 Lambdas 并为所选 Lambda 添加权限以进行 Amazon IVS Chat 审核
- Amazon Cloudwatch 用于获取聊天会话的指标

```
{
  "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": "*"
    }
  ]
}
```

Amazon IVS Chat 基于资源的策略

您必须向 Amazon IVS Chat 服务授予权限才能调用 Lambda 资源来审查消息。为此，请按照 将基于资源的策略用于 AWS Lambda（在《AWS Lambda 开发人员指南》）中的说明操作并填写以下指定的字段。

要控制对 Lambda 资源的访问，您可以使用基于以下前提的条件：

- SourceArn – 我们的示例策略使用通配符（*），允许账户中的所有房间调用 Lambda。或者，您也可以在账户中指定一个房间，仅允许该房间调用 Lambda。
- SourceAccount – 在以下示例策略中，AWS 账户 ID 为 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/*"
            }
         }
      }
   ]
}
```

故障排除

有关诊断和修复在使用 Amazon IVS 聊天功能和 IAM 时可能遇到的常见问题，请参阅《IVS Low-Latency Streaming User Guide》中的 Troubleshooting。

IVS 聊天功能的托管式策略

AWS 托管式策略是由 AWS 创建和管理的独立策略。请参阅《IVS Low-Latency Streaming User Guide》中的 Managed Policies for Amazon IVS。

对 IVS 聊天功能使用服务相关角色

Amazon IVS 使用 AWS IAM 服务相关角色。请参阅《IVS Low-Latency Streaming User Guide》中的 Using Service-Linked Roles for Amazon IVS。

IVS 聊天功能日志记录和监控

要记录性能和/或操作，请使用 Amazon CloudTrail。请参阅《IVS Low-Latency Streaming User Guide》中的 Logging Amazon IVS API Calls with AWS CloudTrail。

IVS 聊天功能事件响应

要检测事件或发出事件警报，您可以通过 Amazon EventBridge 事件监控流的运行状况。请参阅适用于低延迟直播功能和适用于实时直播功能的 Using Amazon EventBridge with Amazon IVS。

使用 AWS Health Dashboard 获取有关 Amazon IVS 整体运行状况的信息（按区域）。

IVS 聊天功能恢复能力

IVS API 使用 AWS 全球基础设施，并围绕 AWS 区域和可用区构建。请参阅《IVS Low-Latency Streaming User Guide》中的 IVS Resilience。

IVS 聊天功能基础设施安全性

作为一项托管式服务，Amazon IVS 都受 AWS 全球网络安全流程的保护。这些流程的说明详见安全性、身份和合规性的最佳实践。

API 调用

您可以使用 AWS 发布的 API 调用通过网络访问 Amazon IVS。请参阅《IVS Low-Latency Streaming User Guide》中 Infrastructure Security 下的 API Calls。

Amazon IVS 聊天功能

Amazon IVS Chat 消息提取和传送通过与我们边缘连接的加密 WSS 进行。Amazon IVS Messaging API 使用加密的 HTTPS 连接。与视频流和播放一样，它需要 TLS 1.2 或更高版本，并且可在内部以未加密的方式传输消息收发数据以进行处理。

IVS 聊天功能服务配额

以下是 Amazon Interactive Video Service（IVS）聊天功能端点、资源和其他操作的服务限额和限制。服务配额（也称为限制）是您的亚马逊云科技账户使用的服务资源或操作的最大数量。也就是说，除非表中另有说明，否则这些限制针对每个亚马逊云科技账户。另请参阅 Amazon Service Quotas。

要以编程方式连接到亚马逊云科技服务，您需要使用端点。另请参阅亚马逊云科技服务端点。

所有限额都是按区域强制执行的。

服务限额增加

对于可调配额，您可以通过 Amazon 管理控制台请求提高速率。也可以使用控制台查看有关服务限额的信息。

API 调用速率限额不可调整。

API 调用速率限额

端点类型	端点	默认
消息收发	DeleteMessage	100 TPS
消息收发	DisconnectUser	100 TPS
消息收发	SendEvent	100 TPS
聊天令牌	CreateChatToken	200 TPS
日志记录配置	CreateLoggingConfiguration	3 TPS
日志记录配置	DeleteLoggingConfiguration	3 TPS
日志记录配置	GetLoggingConfiguration	3 TPS
日志记录配置	ListLoggingConfigurations	3 TPS
日志记录配置	UpdateLoggingConfiguration	3 TPS
房间	CreateRoom	5 TPS

端点类型	端点	默认
房间	DeleteRoom	5 TPS
房间	GetRoom	5 TPS
房间	ListRooms	5 TPS
房间	UpdateRoom	5 TPS
标签	ListTagsForResource	10 TPS
标签	TagResource	10 TPS
标签	UntagResource	10 TPS

其他限额

资源或功能	默认	可调整	描述
并发聊天连接	50000	是	每个账户在 AWS 区域 的所有房间中的最大并发聊天连接数。
日志记录配置	10	是	在当前的 AWS 区域 中，每个账户可以创建的日志记录配置的最大数量。
消息审核处理程序超时时段	200	否	当前 AWS 区域 中所有消息审核处理程序的超时时段（以毫秒为单位）。如果超过此值，则允许或拒绝该消息，具体取决于为消息审核处理程序配置的 fallbackResult 字段的值。
所有房间的 DeleteMessage 请求速率	100	是	所有房间每秒可以发出的最大 DeleteMessage 请求数。请

资源或功能	默认	可调整	描述
			求可以来自 Amazon IVS Chat API，也可以来自 Amazon IVS Chat Messaging API（WebSocket）。
所有房间的 DisconnectUser 请求速率	100	是	所有房间每秒可以发出的最大 DisconnectUser 请求数。请求可以来自 Amazon IVS Chat API，也可以来自 Amazon IVS Chat Messaging API（WebSocket）。
每个连接的消息收发请求速率	10	否	聊天连接每秒可发出的最大消息收发请求数。
所有房间的 SendMessage 请求速率	1000	是	所有房间每秒可发出的最大 SendMessage 请求数。这些请求来自 Amazon IVS Chat Messaging API（WebSocket）。
每个房间的 SendMessage 请求速率	100	否（但可通过 API 进行配置）	任一房间每秒可发出的最大 SendMessage 请求数。这可通过 CreateRoom 和 UpdateRoom 的 `maximumMessageRatePerSecond` 字段进行配置。这些请求来自 Amazon IVS Chat Messaging API（WebSocket）。
房间	50000	是	每个账户在每个 AWS 区域 的最大聊天室数量。

服务限额与 CloudWatch 使用量指标的集成

您可以通过 CloudWatch 使用量指标主动管理您的服务限额。您可以使用这些指标在 CloudWatch 图表和控制面板上直观显示当前服务用量。Amazon IVS 聊天功能用量指标与 Amazon IVS 聊天功能服务限额对应。

您可以使用 CloudWatch 指标数学函数在图表上显示这些资源的服务配额。还可以配置警报，以在用量接近服务配额时向您发出警报。

要访问用量指标，请执行以下操作：

1. 打开服务配额控制台：https://console.aws.amazon.com/servicequotas/
2. 在导航窗格中，选择 Amazon services（亚马逊云科技服务）。
3. 从 AWS 服务列表中，搜索并选择 Amazon Interactive Video Service 聊天功能。
4. 在 Service quotas（服务限额）列表中，选择感兴趣的服务限额。系统将打开一个新页面，其中包含有关服务配额/指标的信息。

或者，您可以通过 CloudWatch 控制台访问这些指标。在 AWS Namespaces（Amazon 命名空间）中，选择 Usage（用量）。然后，从服务列表中选择 IVS 聊天功能。（请参阅监控 Amazon IVS 聊天功能。）

在 AWS/用量命名空间中，Amazon IVS 聊天功能将提供以下指标：

指标名称	描述
`ResourceCount`	您账户中运行的指定资源的数量。资源由与指标关联的维度定义。 有效统计数据：最大数量（1 分钟期间内使用的最大资源数）。

以下维度用于优化用量指标：

维度	描述
服务	包含资源的 AWS 服务的名称。有效值：`IVS Chat`。
类	所跟踪的资源的类。有效值：`None`。

维度	描述
类型	所跟踪的资源的类型。有效值：Resource。
资源	亚马逊云科技资源的名称。有效值：ConcurrentChatConnections 。 ConcurrentChatConnections 用量指标是 AWS/IVSChat 命名空间（使用“无”维度）中相应指标的副本，详见 监控 Amazon IVS 聊天功能。

为用量指标创建 CloudWatch 警报

要基于 Amazon IVS 聊天功能用量指标创建 CloudWatch 警报，请执行以下操作：

1. 从 Service Quotas 控制台中选择感兴趣的服务配额，如上所述。目前，只能为 ConcurrentChatConnections 创建警报。
2. 在 Amazon CloudWatch 警报部分中，选择 Create (创建)。
3. 从 Alarm threshold (警报阈值) 下拉清单中，选择要设置为警报值的已应用配额值的百分比。
4. 对于 Alarm name (警报名称)，输入警报的名称。
5. 选择 Create (创建)。

IVS 聊天功能故障排除

本文档说明了 Amazon Interactive Video Service（IVS）聊天功能的最佳实践和故障排除提示。与 IVS 聊天功能相关的行为通常不同于 IVS 视频相关的行为。有关更多信息，请参阅 Amazon IVS 聊天功能入门。

主题：

删除房间后，为什么 IVS 聊天连接没有断开？

删除聊天室资源时，如果聊天室正在使用，连接到该房间的聊天客户端不会自动断开连接。如果/当聊天应用程序刷新聊天令牌时，连接将会断开。或者，必须手动断开所有用户的连接才能将所有用户从聊天室中删除。

IVS 术语表

另请参阅 AWS 术语表。在下表中，LL 代表 IVS 低延迟直播；RT 代表 IVS 实时直播。

租期	描述	LL	RT	聊天
AAC	高级音频编码。AAC 是有损数字音频压缩的音频编码标准。AAC 旨在成为 MP3 格式的继任者，在相同的比特率下，其音质通常比 MP3 更高。ISO 和 IEC 已将 ACC 标准化，作为 MPEG-2 和 MPEG-4 规范的一部分。	✓	✓	
自适应比特率流	自适应比特率（ABR）流允许 IVS 播放器在连接质量下降时切换到较低的比特率，并在连接质量提高时切换回较高的比特率。	✓		
自适应流	请参阅通过联播分层编码。		✓	
管理用户	对 AWS 账户中可用的资源和服务具有管理权限的 AWS 用户。请参阅《AWS 设置用户指南》中的术语。	✓	✓	✓
ARN	Amazon 资源名称，这是 AWS 资源的唯一标识符。具体的 ARN 格式取决于资源类型。有关 IVS 资源使用的 ARN 格式，请参阅《服务授权参考》。	✓	✓	✓
纵横比	描述帧宽度与帧高的比率。例如，16:9 是与全高清或 1080p 分辨率相对应的宽高比。	✓	✓	
音频模式	针对不同类型的移动设备用户及其使用的设备进行优化的预设或自定义音频配置。请参阅 IVS 广播 SDK：移动音频模式（实时直播）。		✓	
AVC、H.264、MPEG-4 第 10 部分	高级视频编码，也称为 H.264 或 MPEG-4 第 10 部分，是有损数字视频压缩的视频压缩标准。	✓	✓	

租期	描述	LL	RT	聊天
背景替换	一种相机滤镜，让实时流创作者能够更改其背景。请参阅 IVS 广播 SDK：第三方相机滤镜（实时直播）中的背景替换。		✓	
比特率	每秒传输或接收的比特数的直播指标。	✓	✓	
广播，广播者	流、直播工具的其他术语。	✓		
缓冲	播放设备在需要播放内容之前无法下载该内容时出现的一种情况。缓冲可以通过多种方式表现出来：内容可能随机停止并开始（也称为卡顿），内容也可能长时间停止（也称为冻结），或者 IVS 播放器可能进入了暂停状态。	✓	✓	
字节范围播放列表	比标准 HLS 播放列表更精细的播放列表。标准 HLS 播放列表由 10 秒的媒体文件组成。对于字节范围播放列表，片段持续时间与为流配置的关键帧间隔相同。 字节范围播放列表仅适用于自动录制到 S3 存储桶的广播。其是在 HLS 播放列表之外创建的。请参阅自动录制到 Amazon S3（低延迟直播）中的字节范围播放列表。	✓		
CBR	恒定比特率，编码器的一种速率控制方法，无论广播期间发生什么情况，都能在视频的整个播放过程中保持一致的比特率。可以填充操作中的间歇以达到所需的比特率，并且可以通过调整编码质量以匹配目标比特率来量化峰值。我们强烈建议使用 CBR 而不是 VBR。	✓	✓	
CDN	内容分发网络，一种地理分布式解决方案，通过让直播视频等内容更靠近用户所在位置来优化内容的交付。	✓		

租期	描述	LL	RT	聊天
频道	存储直播配置的 IVS 资源，包括摄取服务器、流密钥、播放 URL 和录制选项。流传输工具使用与通道关联的流密钥来启动广播。广播期间生成的所有指标和事件都与通道资源相关联。	✓		
通道类型	确定通道允许的分辨率和帧速率。请参阅 IVS Low-Latency Streaming API Reference 中的 Channel Types（通道类型）。	✓		
聊天记录	一个高级选项，可以通过将日志记录配置与某个聊天室相关联来启用。			✓
聊天室	一种 IVS 资源，用于存储聊天会话的配置，包括消息审核处理程序和聊天记录等可选功能。请参阅 Getting Started with IVS Chat 中的 Step 2: Create a Chat Room。			✓
客户端合成	使用主机设备混合舞台参与者的音频和视频流，然后将其作为复合流发送到 IVS 通道。这样可以更好地控制合成的外观，代价是客户端资源的利用率更高，舞台或主机问题影响查看者的风险也更高。 另请参阅服务器端合成。	✓	✓	
CloudFront	Amazon 提供的 CDN 服务。	✓		
CloudTrail	一项 AWS 服务，用于收集、监控、分析和保留来自 AWS 和外部来源的事件和账户活动。请参阅使用 AWS CloudTrail 记录 IVS API 调用。	✓	✓	✓
CloudWatch	一项 AWS 服务，用于监控应用程序、响应性能变化、优化资源使用并提供有关运行状况的见解。您可以使用 CloudWatch 来监控 IVS 指标；请参阅监控 IVS 实时直播和监控 IVS 低延迟直播。	✓	✓	✓
合成	将来自多个来源的音频和视频流合并为单个流的过程。	✓	✓	

租期	描述	LL	RT	聊天
合成管道	合并多个流并对生成的流进行编码所需的一系列处理步骤。	✓	✓	
压缩	使用比原始表示形式更少的比特数对信息进行编码。任何特定的压缩都是无损或有损的。无损压缩通过识别和消除统计冗余来减少比特数。无损压缩不会丢失任何信息。有损压缩通过删除不必要或不太重要的信息来减少比特数。	✓	✓	
控制层面	存储有关 IVS 资源（例如通道、舞台或聊天室）的信息，并提供用于创建和管理这些资源的界面。其具有区域性（基于 AWS 区域）。	✓	✓	✓
CORS	跨源资源共享（CORS），这是一项 AWS 功能，允许在一个域中加载的客户端 Web 应用程序与另一个域中的资源（例如 S3 存储桶）进行交互。可以根据标头、HTTP 方法和源域配置访问权限。请参阅《Amazon Simple Storage Service 用户指南》中的使用跨源资源共享（CORS）–Amazon Simple Storage Service。	✓		
自定义图像源	IVS 广播 SDK 提供的界面，允许应用程序提供自己的图像输入，而不仅限于预设相机。	✓	✓	
数据层面	将数据从摄取传输到出口的基础设施。其根据控制面板中管理的配置运行，不限于 AWS 区域。	✓	✓	✓
编码器、编码	将视频和音频内容转换为适合直播的数字格式的过程。编码可以基于硬件，也可以基于软件。	✓	✓	
事件	IVS 向 AmazonEventBridge 监控服务发布的自动通知。事件代表直播资源（例如舞台或合成管道）的状态或运行状况变化。请参阅将 Amazon EventBridge 与 IVS 低延迟直播结合使用和将 Amazon EventBridge 与 IVS 实时直播结合使用。	✓	✓	✓

租期	描述	LL	RT	聊天
FFmpeg	一个免费的开源软件项目，其中包含一套用于处理视频和音频文件以及流的库和程序。FFmpeg 提供了一种跨平台解决方案来录制、转换以及直播音频和视频。	✓		
片段化的流	当广播断开连接，然后在通道录制配置中指定的间隔内重新连接时创建。生成的多个流被视为单个广播，并合并到录制流中。请参阅自动录制到 Amazon S3（低延迟直播）中合并片段化的流。	✓		
帧率	每秒传输或接收的视频帧数的直播指标。	✓	✓	
HLS	HTTP 实时流（HLS），一种基于 HTTP 的自适应比特率流通信协议，用于向查看者传送 IVS 流。	✓		
HLS 播放列表	组成流的媒体片段列表。标准 HLS 播放列表由 10 秒的媒体文件组成。HLS 还支持更精细的字节范围播放列表。	✓		
Host	向舞台发送视频和/或音频的实时事件参与者。		✓	
IAM	Identity and Access Management，这是一项 AWS 服务，允许用户安全地管理身份和访问 AWS 服务和资源，包括 IVS。	✓	✓	✓
提取	用于从主机或广播者接收视频流以进行处理或传送给查看者或其他参与者的 IVS 过程。	✓	✓	
提取服务器	接收视频流并将其传送到转码系统，在该系统中，视频流被转码多路复用或转码为 HLS 以传送给查看者。 摄取服务器是特定的 IVS 组件，用于接收通道流以及摄取协议（RTMP、RTMPS）。有关创建通道的信息，请参阅 IVS 低延迟直播入门。		✓	

租期	描述	LL	RT	聊天
隔行视频	仅传输和显示后续帧的奇数行或偶数行，从而在不消耗额外带宽的情况下实现帧速率的翻倍。出于视频质量方面的考虑，我们不建议使用隔行视频。	✓	✓	
JSON	JavaScript 对象表示法，这是一种开放标准文件格式，使用人类可读文本来传输由属性值对和数组数据类型或其他可序列化值组成的数据对象。	✓	✓	✓
关键帧、增量帧、关键帧间隔	关键帧（也称为帧内编码或 i 帧）是视频中图像的全帧。后续帧，即增量帧（也称为预测帧或 p 帧），仅包含已更改的信息。关键帧将在一个流中多次出现，具体取决于编码器中定义的关键帧间隔。	✓	✓	
Lambda	一项 AWS 服务，用于运行代码（称为 Lambda 函数），无需预置任何服务器基础设施。Lambda 函数可以响应事件和调用请求运行，也可以根据计划运行。例如，IVS 聊天功能使用 Lambda 函数来启用聊天室的消息审核。	✓	✓	✓
延迟、玻璃到玻璃的延迟	数据传输中的延迟。IVS 将延迟范围定义如下： • 低延迟：低于 3 秒 • 实时延迟：低于 300 毫秒 玻璃到玻璃延迟是指从摄像机捕获实时流到流显示在查看者屏幕上之间的延迟。	✓	✓	
通过联播分层编码	支持同时编码并发布具有不同质量级别的多个视频流。请参阅实时直播优化中的自适应直播：通过联播分层编码。		✓	
消息审核处理程序	允许 IVS 聊天功能客户能够在用户聊天消息传送到聊天室之前自动查看/筛选这些消息。将 Lambda 函数与聊天室关联来将其启用。请参阅 Chat Message Review Handler 中的 Creating a Lambda Function。			✓

租期	描述	LL	RT	聊天
混合器	IVS 移动广播 SDK 的一项功能，可接收多个音频和视频源并生成单个输出。其支持对代表源的屏幕视频和音频元素进行管理，例如相机、麦克风、屏幕截图以及应用程序生成的音频和视频。然后可以将输出直播到 IVS。请参阅 IVS 广播 SDK：混合器指南（低延迟直播）中的配置广播会话以进行混合。	✓		
多主机直播	将来自多个主机的流合并为一个流。可通过使用客户端或服务器端合成来实现。 多主机直播支持诸如邀请查看者到舞台进行问答、主机之间的竞赛、视频聊天以及主机当众对话等场景。		✓	
多变体播放列表	可用于广播的所有变体流的索引。	✓		
OAC	来源访问控制，一种用于限制对 S3 存储桶访问的机制，因此只能通过 CloudFront CDN 提供录制流等内容。	✓		
OBS	Open Broadcaster Software（OBS），一款用于视频录制和实时直播的免费开源软件。OBS 为桌面发布提供了另一种选择（IVS 广播 SDK）。熟悉 OBS 的更精密的流传输工具可能会更喜欢这种选择，因为其具有高级制作功能，例如场景过渡、音频混音和图形叠加。	✓	✓	
参与者	以主机或查看者身份连接到舞台的实时用户。		✓	
参与者令牌	在实时事件参与者加入舞台时对其进行身份验证。参与者令牌还能控制参与者是否可以向舞台发送视频。		✓	

租期	描述	LL	RT	聊天
播放令牌、播放密钥对	一种授权机制，允许客户限制在私有通道上播放视频。播放令牌由播放密钥对生成。 播放密钥对是用于签名和验证查看者播放授权令牌的公有-私有密钥对。请参阅 Setting up IVS Private Channels 中的 Create or Import an IVS Playback Key 以及 IVS Low-Latency API Reference 中的播放密钥对端点。	✓		
播放 URL	标识查看者用于开始特定通道播放的地址。此地址可以在全球范围使用。IVS 会自动为每个查看者选择 IVS 全球内容分发网络上的最佳位置，以便向每个查看者传送视频。有关创建通道的信息，请参阅 IVS 低延迟直播入门。	✓		
私有通道	允许客户使用基于播放令牌的授权机制来限制对其流的访问。请参阅 Setting up IVS Private Channels 中的 Workflow for IVS Private Channels。	✓		
逐行视频	按顺序传输并显示每帧的所有行。我们建议在广播的所有阶段使用渐进式视频。	✓	✓	
配额	AWS 账户使用的 IVS 服务资源或操作的最大数量。也就是说，除非另有说明，否则这些限制针对每个 AWS 账户。所有限额都是按区域强制执行的。请参阅 AWS General Reference Guide中的 Amazon Interactive Video Service endpoints and quotas。	✓	✓	✓

租期	描述	LL	RT	聊天
区域	提供对实际位于特定地理区域的 AWS 服务的访问权限。区域提供容错能力、稳定性和弹性，还可以减少延迟。使用区域，您能够创建保持可用且不受区域中断影响的冗余资源。 大多数 AWS 服务请求都与特定的地理区域相关联。除非您明确使用 AWS 服务提供的复制功能，否则在一个区域中创建的资源在任何其他区域中都不存在。例如，Amazon S3 支持跨区域复制。某些服务（例如 IAM）没有跨区域资源。	✓	✓	✓
解决方案	描述单个视频帧中的像素数，例如，全高清或 1080p 定义了具有 1920x1080 像素的帧。	✓	✓	
根用户	AWS 账户的所有者。根用户对 AWS 账户中的所有 AWS 服务和资源具有完全访问权限。	✓	✓	✓
RTMP、RTMPS	实时消息协议，一种通过网络传输音频、视频和数据的行业标准。RTMPS 是 RTMP 的安全版本，通过传输层安全性协议（TLS/SSL）连接运行。	✓	✓	
S3 存储桶	存储在 Amazon S3 中对象的集合。许多策略（包括访问和复制）都是在存储桶级别定义的，适用于存储桶中的所有对象。例如，IVS 广播作为多个对象存储在 S3 存储桶中。	✓		
SDK	软件开发工具包，供开发人员使用 IVS 构建应用程序的库的集合。	✓	✓	✓
自拍分割	允许使用特定于客户端的解决方案替换实时流中的背景，该解决方案接受相机图像作为输入，并返回一个掩码，该掩码为图像的每个像素提供置信度分数，指示图像是在前景还是背景中。请参阅 IVS 广播 SDK：第三方相机滤镜（实时直播）中的背景替换。		✓	

租期	描述	LL	RT	聊天
Semantic 版本控制	Major.Minor.Patch 形式的版本格式。不影响 API 的错误修复会增加补丁版本，向后兼容的 API 添加/更改会增加次要版本，不向后兼容的 API 更改会增加主要版本。	✓	✓	✓
服务器端合成	使用 IVS 服务器混合舞台参与者的音频和视频，然后将此混合视频发送到 IVS 通道，以服务于更多观众或将其存储在 S3 存储桶中。服务器端合成减少了客户端负载，提高了广播的弹性，并可以更有效地使用带宽。 另请参阅客户端合成。		✓	
服务限额	一项 AWS 服务，可帮助您从一个位置管理多个 AWS 服务的限额。除了查找配额值，您也可以从 Service Quotas 控制台请求提高配额。	✓	✓	✓
服务相关角色	与 AWS 服务直接关联的一种独特类型的 IAM 角色。服务相关角色由 IVS 自动创建，并包含该服务代表您调用其他 AWS 服务所需的一切权限，例如访问 S3 存储桶。请参阅 IVS 安全性中的对 IVS 使用服务相关角色。	✓		
舞台	IVS 资源，代表实时事件参与者可以在其中实时交换视频的虚拟空间。请参阅 IVS 实时直播入门中的创建舞台。		✓	
舞台会话	第一个参与者加入舞台时舞台会话开始，最后一个参与者停止向舞台发布几分钟后舞台会话结束。长期存在的舞台在其生命周期内可能有多个会话。		✓	
流	表示从源持续发送到目的地的视频或音频内容的数据。	✓	✓	
流密钥	创建通道时由 IVS 分配的标识符；用于授权直播到该通道。将流密钥视为秘密，因为任何拥有它的人都可以直播到通道。请参阅 IVS 低延迟直播入门。	✓		

租期	描述	LL	RT	聊天
流匮乏	向 IVS 的直播延迟或停止。当 IVS 未收到编码设备宣传的其将在特定时间范围内发送的预期比特量时，就会发生这种情况。出现流匮乏会导致流匮乏事件。 从查看者的角度来看，流匮乏可能表现为视频延迟、缓冲或冻结。流匮乏的持续时间可能较为短暂（小于 5 秒），也可能较长（几分钟），取决于导致流匮乏的具体情况。请参阅问题排查常见问题中的什么是流匮乏。	✓	✓	
流传输工具	向 IVS 发送视频或音频流的个人或设备。	✓	✓	
订阅者	接收主机视频和/或音频的实时事件参与者。请参阅什么是 IVS 实时直播。		✓	
标签	分配给 AWS 资源的元数据标签。标签可帮助您标识和组织 AWS 资源。在 IVS 文档登录页面上，请参阅任何 IVS API 文档中的“标记”（适用于实时直播、低延迟直播或聊天）。	✓	✓	✓
第三方相机滤镜	可以与 IVS 广播 SDK 集成的软件组件，允许应用程序在将图像作为自定义图像源提供给广播 SDK 之前对其进行处理。第三方相机滤镜可以处理来自相机的图像、应用滤镜效果等。	✓	✓	
缩略图	从流中拍摄的缩小尺寸的图像。默认情况下，每 60 秒生成一次缩略图，但可以配置更短的时间间隔。缩略图分辨率取决于通道类型。请参阅自动录制到 Amazon S3（低延迟直播）中的录制内容。	✓		

租期	描述	LL	RT	聊天
定时元数据	与流中特定时间戳相关联的元数据。可以使用 IVS API 以编程方式进行添加，并与特定帧相关联。这样可以确保所有查看者在与视频流相关的同一时间点上接收元数据。 定时元数据可用于触发客户端上的操作，例如在体育赛事期间更新球队统计数据。请参阅将元数据嵌入视频流中。	✓		
转码	将视频和音频从一种格式转换为另一种格式。传入流可以多个比特率和分辨率将其转码为不同格式，以支持一系列播放设备和网络条件。	✓	✓	
转码多路复用	将摄取的流简单地重新打包到 IVS，无需重新编码视频流。“Transmux”是转码多路复用的缩写，这是一个改变音频和/或视频文件格式的过程，同时保留部分或全部原始流。转码多路复用转换为不同的容器格式，而不会更改文件内容。与转码不同。	✓	✓	
变体流	多个不同的质量级别的同一广播的一组编码。每个变体流都编码为单独的 HLS 播放列表。可用变体流的索引称为多变体播放列表。 IVS 播放器从 IVS 接收到多变体播放列表后，可以在播放期间在变体流之间进行选择，随着网络条件的变化，可以无缝地来回转换。	✓		
VBR	可变比特率，编码器的一种速率控制方法，使用动态比特率，根据所需的详情级别在整个播放过程中发生变化。出于视频质量方面的考虑，我们强烈建议不要使用 VBR；改用 CBR。	✓	✓	

租期	描述	LL	RT	聊天
查看	会主动下载或播放视频的独特观看会话。观看次数是并发观看次数限额的基础。 视图会在查看会话开始播放视频时开始。视图会在查看会话停止视频播放时结束。播放是收视率的唯一指标；不考虑音频级别、浏览器选项卡焦点和视频质量等互动启发式算法。在计算观看次数时，IVS 不会考虑个别查看者的合法性，也不会尝试重复计算本地化的收视率，例如单台计算机上的多个视频播放器。请参阅服务限额（低延迟直播）中的其他限额。	✓		
查看者	从接收 IVS 流的人。	✓		
WebRTC	Web 实时通信，一个为 Web 浏览器和移动应用程序提供实时通信的开源项目。。允许直接点对点通信，从而允许音频和视频在网页内进行通信，无需安装插件或下载本机应用程序。 WebRTC 背后的技术是作为开放的 Web 标准实现的，在所有主流浏览器中都可以作为常规 JavaScript API 使用，也可以作为本地客户端（例如 Android 和 iOS）的库使用。	✓	✓	

租期	描述	LL	RT	聊天
WHIP	WebRTC-HTTP 摄取协议，一种基于 HTTP 的协议，允许以基于 WebRTC 的方式将内容摄取到流式传输服务和/或 CDN 中。WHIP 是一份 IETF 草案，旨在对 WebRTC 摄取进行标准化。 WHIP 与 OBS 等软件兼容，为桌面发布提供了另一种选择（替代 IVS 广播 SDK）。熟悉 OBS 的经验丰富的流媒体参与者可能会更喜欢这种选择，因为其具有高级制作功能，例如场景过渡、音频混音和图形叠加 在无法使用或不宜使用 IVS 广播 SDK 的情况下，WHIP 也很有用。例如，在涉及硬件编码器的情况下，可能无法选择 IVS 广播 SDK。但是，如果编码器支持 WHIP，您仍然可以直接从编码器发布到 IVS。 请参阅 IVS WHIP 支持（实时流）。		✓	
WSS	WebSocket Secure，通过在加密的 TLS 连接上建立 WebSocket 的协议。用于连接到 IVS 聊天端点。请参阅 Getting Started with IVS Chat中的 Step 4: Send and Receive Your First Message。			✓

IVS 聊天功能文档历史记录

下表介绍了对 Amazon IVS 聊天功能的文档所做的重要更改。我们经常更新文档以发布新版本以及处理您发给我们的反馈。

聊天功能用户指南更改

变更	说明	日期
拆分聊天 UG	此发行版附带主要的文档更改。我们将聊天信息从《IVS Low-Latency Streaming User Guide》移至新的《IVS Chat User Guide》，该指南位于 IVS 文档登录页面的现有 IVS 聊天功能部分。 有关其他文档更改，请参阅 Document History（Low-Latency Streaming）。	2023 年 12 月 28 日
IVS 术语表	扩展了术语表，涵盖 IVS 实时、低延迟和聊天术语。	2023 年 12 月 20 日

IVS 聊天功能 API 参考更改

API 更改	描述	日期
拆分聊天 UG	现在有了 IVS 聊天功能用户指南（在本版本中创建），接下来，现有 IVS Chat API Reference 和 IVS Chat Messaging API Reference 的文档历史记录条目将位于此处。这些聊天功能 API 参考以往的历史记录条目位于 Document History (Low-Latency Streaming) 中。	2023 年 12 月 28 日

IVS 聊天功能发布说明

此文档包含所有 Amazon IVS 聊天功能发布说明（按发布日期排列，最新的发布说明显示在最前面）。

2023 年 12 月 28 日

Amazon IVS 聊天功能用户指南

Amazon Interactive Video Service（IVS）聊天功能是一项托管式实时聊天功能，可与实时视频直播一起使用。在此发布版本中，我们将聊天功能信息从《IVS Low-Latency Streaming User Guide》移至新的《IVS 聊天功能用户指南》。可从 Amazon IVS 文档登陆页面访问文档。

2023 年 1 月 31 日

Amazon IVS Chat 客户端消息收发 SDK：Android 1.1.0

平台	下载和更改
Android 聊天功能客户端消息收发 SDK 1.1.0	参考文档：https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/ • 为了支持 Kotlin 协同例程，我们在 com.amazonaws.ivs.chat.messaging.coroutines 程序包中添加了新的 IVS 聊天功能消息收发 API。另请参阅新的 Kotlin 协同教程；第 1 部分（共 2 部分）是聊天室。

Chat 客户端消息收发 SDK 大小：Android

架构	压缩大小	未压缩大小
所有架构（字节码）	89KB	92KB

2022 年 11 月 9 日

Amazon IVS Chat 客户端消息收发 SDK：JavaScript 1.0.2

平台	下载和更改
JavaScript Chat 客户端消息收发 SDK 1.0.2	参考文档：https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/ • 修复了影响 Firefox 的问题：客户端在使用 DisconnectUser 端点与聊天室断开连接时错误地收到了套接字错误。

2022 年 9 月 8 日

Amazon IVS Chat 客户端消息收发 SDK：Android 1.0.0 和 iOS 1.0.0

平台	下载和更改
Android Chat 客户端消息收发 SDK 1.0.0	参考文档：https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/
iOS Chat 客户端消息收发 SDK 1.0.0	参考文档：https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/

Chat 客户端消息收发 SDK 大小：Android

架构	压缩大小	未压缩大小
所有架构（字节码）	53KB	58KB

Chat 客户端消息收发 SDK 大小：iOS

架构	压缩大小	未压缩大小
ios-arm64_x86_64 模拟器（位码）	484KB	2.4MB
ios-arm64_x86_64 模拟器	484KB	2.4MB
ios-arm64（位码）	1.1MB	3.1MB
ios-arm64	233KB	1.2MB