aws

챗 사용 설명서

# Amazon IVS

# Amazon IVS: 챗 사용 설명서

# Table of Contents

# Amazon IVS 챗이란?

Amazon IVS 챗은 라이브 비디오 스트림과 함께 제공되는 관리형 라이브 채팅 기능입니다. Amazon IVS 설명서 랜딩 페이지의 Amazon IVS 챗 섹션에서 설명서에 액세스할 수 있습니다.

- 챗 사용 설명서 - 이 문서는 탐색 창에 나열된 다른 모든 사용 설명서 페이지와 함께 제공됩니다.
- Chat API 참조 - 컨트롤 플레인 API(HTTPS)입니다.
- 챗 메시징 API 참조 - 데이터 영역 API(WebSocket)입니다.
- 챗 클라이언트용 SDK 참조: Android, iOS, JavaScript.

# Amazon IVS 챗 시작하기

Amazon Interactive Video Service(IVS) Chat은 라이브 비디오 스트림과 함께 사용할 수 있는 관리형 라이브 채팅 기능입니다. (IVS 챗은 비디오 스트림 없이도 사용할 수 있습니다.) 채팅룸을 생성하고 사용자 간 채팅 세션을 활성화할 수 있습니다.

Amazon IVS 챗을 사용하면 라이브 비디오와 함께 맞춤형 채팅 경험을 구축하는 데 집중할 수 있습니다. 인프라를 관리하거나 채팅 워크플로의 구성 요소를 개발하고 구성할 필요가 없습니다. Amazon IVS 챗은 확장 가능하고 안전하고 안정적이며 비용 효율적입니다.

Amazon IVS 챗은 시작과 끝으로 라이브 비디오 스트림 참가자 간 메시징을 용이하게 하는 데 가장 적합합니다.

이 문서의 나머지 부분에서는 Amazon IVS 챗을 사용하여 첫 번째 채팅 애플리케이션을 구축하는 단계를 안내합니다.

예: 다음과 같은 데모 앱을 사용할 수 있습니다(토큰 생성을 위한 샘플 클라이언트 앱 3개와 백엔드 서버 앱 1개).

- Amazon IVS 챗 웹 데모
- Amazon IVS 챗 for Android 데모
- Amazon IVS 챗 for iOS 데모
- Amazon IVS 챗 데모 백엔드

중요: 24개월 동안 새로운 연결이나 업데이트가 없는 채팅룸은 자동으로 삭제됩니다.

주제



## 1단계: 초기 설정 수행

계속하려면 먼저 다음을 수행해야 합니다.

1. AWS 계정을 생성합니다.
2. 루트 및 관리 사용자를 설정합니다.
3. AWS IAM(Identity and Access Management) 권한을 설정합니다. 아래에 지정된 정책을 사용합니다.

위의 모든 것에 대한 구체적인 단계는 Amazon IVS 사용 설명서의 IVS 지연 시간이 짧은 스트리밍 시작하기를 참조하세요. 중요: “3단계: IAM 권한 설정”에 IVS 챗을 위한 다음 정책을 사용합니다.

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

# 2단계: 채팅룸 생성

Amazon IVS 채팅룸에 구성 정보(예: 최대 메시지 길이)가 연결됩니다.

이 섹션의 지침은 콘솔 또는 AWS CLI를 사용하여 채팅룸을 설정하고(메시지 검토 및/또는 로깅 메시지를 위한 선택적 설정 포함) 채팅룸을 만드는 방법을 보여줍니다.

## 콘솔 지침

이러한 단계는 초기 룸 설정부터 시작하여 최종 룸 생성으로 끝나는 단계로 나뉩니다.

메시지가 검토되도록 방을 설정할 수도 있습니다. 예를 들어 메시지 내용 또는 메타데이터를 업데이트하거나, 메시지를 거부하여 전송되지 않도록 하거나, 원본 메시지가 전달되도록 할 수 있습니다. 방 메시지 검토 설정(선택 사항)에서 이를 다룹니다.

또한 메시지가 로그되도록 방을 설정할 수도 있습니다. 예를 들어 채팅룸으로 메시지를 보내는 경우 Amazon S3 버킷, Amazon CloudWatch 또는 Amazon Kinesis Data Firehose에 대한 메시지를 로그할 수 있습니다. 메시지 로그 검토 설정(선택 사항)에서 이를 다룹니다.

## 초기 방 설정

1. Amazon IVS 챗 콘솔을 엽니다.

   (AWS Management Console을 통해 Amazon IVS 콘솔에 액세스할 수도 있습니다.)

2. 탐색 모음에서 리전 선택(Select a Region) 드롭다운을 사용하여 리전을 선택합니다. 이 리전에 새 방이 생성됩니다.
3. 시작하기(Get started) 상자(오른쪽 위)에서 Amazon IVS 채팅룸(Amazon IVS 챗 Room)을 선택합니다. 방 생성(Create room) 창이 나타납니다.

# 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. 설정(Setup)에서 방 이름(Room name)(선택 사항)을 지정합니다. 방 이름은 고유하지 않지만 이를 통해 방 ARN(Amazon 리소스 이름) 이외의 방을 구분할 수 있습니다.
5. 설정(Setup) > 방 구성(Room configuration)에서 기본 구성(Default configuration)을 수락하거나 사용자 지정 구성(Custom configuration)을 선택한 다음 최대 메시지 길이(Maximum message length) 및/또는 최대 메시지 속도(Maximum message rate)를 구성합니다.
6. 메시지를 검토하려면 아래의 방 메시지 검토 설정(선택 사항)으로 계속합니다. 그렇지 않으면 건너뛰고(즉, 메시지 검토 핸들러(Message Review Handler) > 비활성화됨(Disabled) 수락) 최종 방 생성(Final Room Creation)으로 바로 진행합니다.

## 방 메시지 검토 설정(선택 사항)

1. 메시지 검토 핸들러(Message Review Handler)에서 AWS Lambda로 처리(Handle with AWS Lambda)를 선택합니다. Message Review Handler(메시지 검토 핸들러) 섹션이 확장되어 추가 옵션이 표시됩니다.
2. 핸들러가 유효한 응답을 반환하지 않거나, 오류가 발생하거나, 제한 시간을 초과하는 경우 대체 결과(Fallback result)를 메시지 허용(Allow) 또는 거부(Deny)로 구성합니다.
3. 기존 Lambda 함수(Lambda function)를 지정하거나 Lambda 함수 생성(Create Lambda function)을 사용하여 새 함수를 생성합니다.

   lambda 함수는 채팅룸과 동일한 AWS 계정 및 동일한 AWS 리전에 있어야 합니다. Amazon Chat SDK 서비스에 lambda 리소스를 호출할 수 있는 권한을 부여해야 합니다. 선택한 lambda 함수에 대해 리소스 기반 정책이 자동으로 생성됩니다. 권한에 대한 자세한 내용은 Amazon IVS 챗에 대한 리소스 기반 정책을 참조하세요.

## 메시지 로그 설정(선택 사항)

1. Message logging(메시지 로깅)에서 Automatically log chat messages(채팅 메시지 자동 로그)를 선택합니다. Message logging(메시지 로깅) 섹션이 확장되어 추가 옵션이 표시됩니다. 기존 로깅 구성을 이 방에 추가하거나 Create logging configuration(로깅 구성 생성)을 선택하여 새 로깅 구성을 생성할 수 있습니다.
2. 기존 로깅 구성을 선택하면 드롭다운 메뉴가 나타나고 이미 생성한 로깅 구성이 모두 다 표시됩니다. 목록에서 하나를 선택하면 채팅 메시지가 이 대상에 자동으로 로그됩니다.
3. Create logging configuration(로깅 구성 생성)을 선택하면 새 로깅 구성을 생성하고 사용자 지정할 수 있는 모달 창이 나타납니다.

a. 선택적으로 로깅 구성 이름을 지정합니다. 방 이름과 같은 로깅 구성 이름은 고유하지 않지만 이를 통해 로깅 구성 ARN 이외의 로깅 구성을 구분할 수 있습니다.

b. 대상에서 CloudWatch 로그 그룹, Kinesis firehose 전송 스트림이나 Amazon S3 버킷을 선택하여 로그할 대상을 선택합니다.

c. 대상에 따라 기존 CloudWatch 로그 그룹, Kinesis firehose 전송 스트림이나 Amazon S3 버킷을 사용하거나 새로운 것을 생성하는 옵션을 선택합니다.

d. 검토 후 Create(생성)를 선택하여 고유한 ARN으로 새 로깅 구성을 생성합니다. 그러면 새 로깅 구성이 채팅룸에 자동으로 연결됩니다.

### 최종 방 생성

1. 검토 후 Create chat room(채팅룸 생성)을 선택하여 고유한 ARN으로 새 채팅룸을 생성합니다.

## CLI 지침

### 채팅룸 생성

AWS CLI를 사용하여 채팅룸을 생성하는 방법은 고급 옵션이며, 먼저 시스템에 CLI를 다운로드하고 구성해야 합니다. 자세한 내용은 AWS 명령줄 인터페이스 사용 설명서를 참조하세요.

1. 채팅 `create-room` 명령을 실행하고 선택적 이름을 전달합니다.

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

2. 그러면 새 채팅룸이 반환됩니다.

```
{
   "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. `arn` 필드를 기록해 둡니다. 이는 클라이언트 토큰을 생성하고 채팅룸에 연결하는 데 필요합니다.

## 로깅 구성 설정(선택 사항)

채팅룸 생성과 마찬가지로 AWS CLI를 사용하여 로깅 구성을 설정하는 것은 고급 옵션이며, 먼저 컴퓨터에 CLI를 다운로드하고 구성해야 합니다. 자세한 내용은 AWS 명령줄 인터페이스 사용 설명서를 참조하세요.

1. 채팅 `create-logging-configuration` 명령을 실행하고 선택적 이름과 Amazon S3 버킷을 가리키는 대상 구성을 이름별로 전달합니다. 이 Amazon S3 버킷은 로깅 구성을 생성하기 전에 존재해야 합니다. (Amazon S3 버킷 생성에 대한 자세한 내용은 Amazon S3 설명서를 참조하세요.)

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

2. 그러면 새 로깅 구성이 반환됩니다.

```
{
   "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. `arn` 필드를 기록해 둡니다. 채팅룸에 로깅 구성을 연결하려면 이 정보가 필요합니다.

   a. 새 채팅룸을 생성하는 경우`create-room` 명령을 실행하고 로깅 구성 `arn`을 전달합니다.

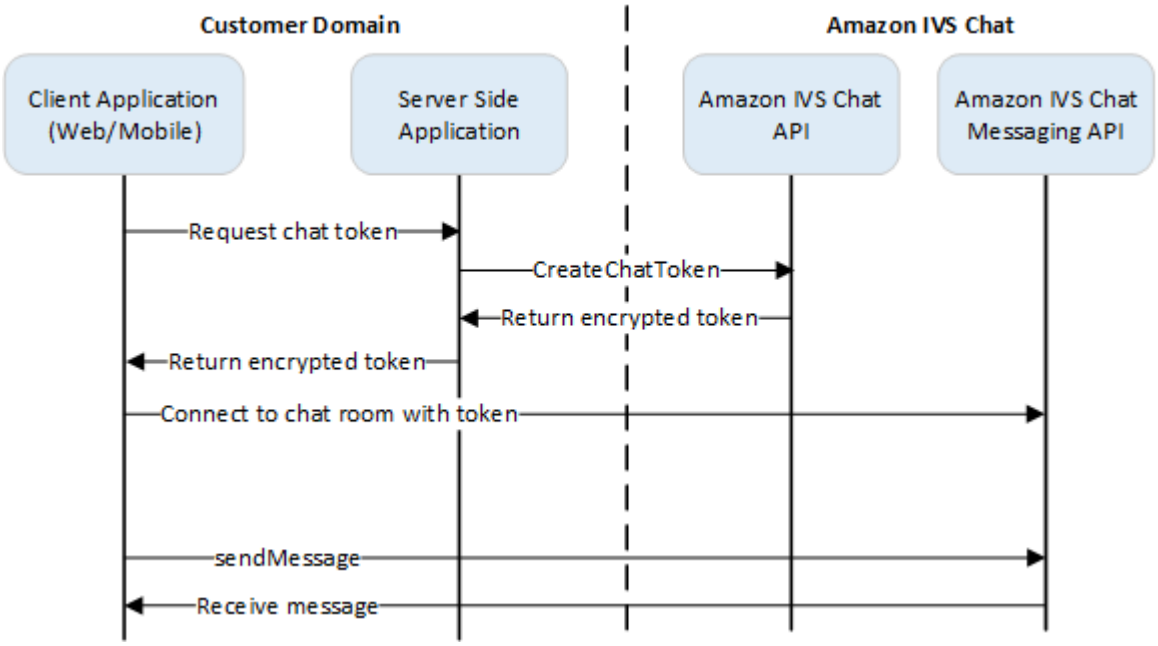
```
aws ivschat create-room --name test-room \
--logging-configuration-identifiers \
"arn:aws:ivschat:us-west-2:123456789012:logging-configuration/ABcdef34ghIJ"
```

   b. 기존 채팅룸을 업데이트하는 경우`update-room` 명령을 실행하고 로깅 구성 `arn`을 전달합니다.

```
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"
```

# 3단계: 채팅 토큰 생성




채팅 참가자가 채팅룸에 접속하여 메시지를 보내고 받기 시작하려면 채팅 토큰을 만들어야 합니다. 채팅 토큰은 채팅 클라이언트를 인증하고 권한을 부여하는 데 사용됩니다. 위에 표시된 것처럼 클라이언트 애플리케이션은 서버 측 애플리케이션에 토큰을 요청하고 서버 측 애플리케이션은 AWS SDK 또는 SigV4 서명 요청을 사용하여 CreateChatToken을 호출합니다. AWS 자격 증명은 API를 호출하는 데 사용되므로 토큰은 클라이언트 측 애플리케이션이 아닌 안전한 서버 측 애플리케이션에서 생성되어야 합니다.

토큰 생성을 보여주는 백엔드 서버 애플리케이션은 Amazon IVS 챗 데모 백엔드에서 사용할 수 있습니다.

세션 지속 시간은 설정된 세션이 자동으로 닫히기 전에 활성 상태를 유지할 수 있는 시간을 나타냅니다. 즉, 세션 지속 시간은 새 토큰이 생성되고 새 연결이 설정되기 전에 클라이언트가 채팅룸에 연결된 상태를 유지할 수 있는 기간입니다. 선택 사항으로, 토큰 생성 중 세션 기간을 지정할 수 있습니다.

각 토큰은 최종 사용자에 대한 연결을 설정하는 데 한 번만 사용할 수 있습니다. 연결이 닫히면 연결을 다시 설정하기 전에 새 토큰을 만들어야 합니다. 토큰 자체는 응답에 포함된 토큰 만료 타임스탬프까지 유효합니다.

최종 사용자가 채팅룸에 연결하려는 경우 클라이언트는 서버 애플리케이션에 토큰을 요청해야 합니다. 서버 애플리케이션은 토큰을 생성하여 클라이언트에 다시 전달합니다. 최종 사용자를 위해 필요에 따라 토큰을 만들어야 합니다.

채팅 인증 토큰을 만들려면 아래 지침을 따르세요. 채팅 토큰을 생성할 때 요청 필드를 사용하여 채팅 최종 사용자 및 최종 사용자의 메시징 기능에 대한 데이터를 전달합니다. 자세한 내용은 IVS Chat API Reference의 CreateChatToken을 참조하세요.

## AWS SDK 지침

AWS SDK를 사용하여 채팅룸을 생성하는 방법은 고급 옵션이며, 먼저 애플리케이션에 SDK를 다운로드하고 구성해야 합니다. 다음은 JavaScript를 사용하는 AWS SDK에 대한 지침입니다.

중요: 이 코드는 서버 측에서 실행되어야 하며 해당 출력은 클라이언트에 전달되어야 합니다.

전제 조건: 아래 코드 샘플을 사용하려면 애플리케이션에 AWS JavaScript SDK를 로드해야 합니다. 자세한 내용은 JavaScript용 AWS SDK 시작하기를 참조하세요.

```
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 지침

AWS CLI를 사용하여 채팅 토큰을 생성하는 방법은 고급 옵션이며, 먼저 시스템에 CLI를 다운로드하고 구성해야 합니다. 자세한 내용은 AWS 명령줄 인터페이스 사용 설명서를 참조하세요. 참고: AWS CLI

를 사용하여 토큰을 생성하는 것은 테스트 목적에 적합하지만, 프로덕션 용도의 경우 AWS SDK를 사용하여 서버 측에서 토큰을 생성하는 것이 좋습니다(위 지침 참조).

1. 클라이언트의 방 식별자 및 사용자 ID와 함께 `create-chat-token` 명령을 실행합니다. `"SEND_MESSAGE"`, `"DELETE_MESSAGE"`, `"DISCONNECT_USER"` 기능을 포함합니다. 필요에 따라 이 채팅 세션에 대한 세션 기간(분) 및/또는 사용자 지정 속성(메타데이터)을 포함합니다. 이러한 필드는 아래에 표시되지 않습니다.

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

2. 그러면 클라이언트 토큰이 반환됩니다.

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

3. 이 토큰을 저장합니다. 이는 채팅룸에 연결하고 메시지를 전송하거나 수신하는 데 필요합니다. 해당 세션이 종료되기 전에 다른 채팅 토큰을 생성해야 합니다(`sessionExpirationTime`으로 표시됨).

## 4단계: 첫 번째 메시지 전송 및 수신

클라이언트 토큰을 사용하여 채팅룸에 연결하고 첫 번째 메시지를 전송합니다. 샘플 JavaScript 코드는 아래에 나와 있습니다. IVS 클라이언트 SDK도 사용할 수 있습니다. 채팅 SDK: Android 안내서, 채팅 SDK: iOS 안내서 및 채팅 SDK: JavaScript 안내서를 참조하세요.

리전 서비스: 아래 샘플 코드는 "지원되는 선택 리전"을 나타냅니다. Amazon IVS 챗은 요청 시 사용할 수 있는 리전 엔드포인트를 제공합니다. Amazon IVS 챗 메시징 API의 경우 리전 엔드포인트의 일반 구문은 다음과 같습니다.

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

예를 들어 wss://edge.ivschat.us-west-2.amazonaws.com은 미국 서부(오레곤) 리전의 엔드포인트입니다. 지원되는 리전 목록은 AWS 일반 참조의 Amazon IVS 페이지에서 Amazon IVS 챗 정보를 참조하세요.

```
/*
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);
}
```

축하합니다. 모두 설정되었습니다. 이제 메시지를 전송하거나 수신할 수 있는 간단한 채팅 애플리케이션이 생겼습니다.

## 5단계: 서비스 할당량 한도 확인(선택 사항)

채팅룸은 Amazon IVS 라이브 스트림과 함께 확장되어 모든 시청자가 채팅 대화에 참여할 수 있습니다. 그러나 모든 Amazon IVS 계정에는 동시 채팅 참가자 수와 메시지 전달 속도에 대한 제한이 있습니다.

특히 대규모 스트리밍 이벤트를 계획하고 있는 경우 한도가 적절한지 확인하고 필요한 경우 한도를 늘려달라고 요청하세요. 자세한 내용은 Service Quotas (Low-Latency Streaming), Service Quotas (Real-Time Streaming) 및 서비스 할당량(챗)을 참조하세요.

# 채팅 로깅

채팅 로깅 기능은 방에 있는 모든 채팅 메시지를 Amazon S3 버킷, Amazon CloudWatch Logs 또는 Amazon Kinesis Data Firehose에 기록할 수 있습니다. 이후 로그는 분석이나 라이브 비디오 세션으로 연결되는 채팅 리플레이를 만드는 데 사용될 수 있습니다.

## 채팅룸에 대한 채팅 로깅 활성화

채팅 로깅은 로깅 구성을 채팅룸과 연결하여 활성화할 수 있는 고급 옵션입니다. 로깅 구성은 채팅룸의 메시지가 기록되는 위치 유형(Amazon S3 버킷, Amazon CloudWatch Logs 또는 Amazon Kinesis Data Firehose)을 지정할 수 있는 리소스입니다. 로깅 구성 생성 및 관리에 대한 자세한 내용은 Amazon IVS Chat 시작하기 및 Amazon IVS Chat API 참조를 확인하세요.

새 방을 만들 때(CreateRoom) 또는 기존 방을 업데이트할 때(UpdateRoom) 각 방에 최대 3개의 로깅 구성을 연결할 수 있습니다. 여러 방을 동일한 로깅 구성에 연결할 수 있습니다.

하나 이상의 활성 로깅 구성이 방과 연결되면 Amazon IVS Chat Messaging API를 통해 해당 방에 전송된 모든 메시징 요청이 지정한 위치에 자동으로 기록됩니다. 다음은 평균 전파 지연 시간(메시징 요청이 전송된 시점부터 지정한 위치에서 사용할 수 있게 되는 시점까지)입니다.

- Amazon S3 버킷: 5분
- Amazon CloudWatch Logs 또는 Amazon Kinesis Data Firehose: 10초

## 메시지 콘텐츠

### 형식

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

### 필드

| 필드 | 설명 |
| --- | --- |
| event_timestamp | Amazon IVS Chat에서 메시지를 받은 시점의 UTC 타임스탬프입니다. |
| payload | 클라이언트가 Amazon IVS Chat 서비스에서 받게 되는 메시지(구독) 또는 이벤트(구독) JSON 페이로드입니다. |
| type | 채팅 메시지의 유형입니다.<br>• 유효한 값: MESSAGE \| EVENT |
| version | 메시지 콘텐츠 형식의 버전입니다. |

## Amazon S3 버킷

### 형식

메시지 로그는 다음과 같은 S3 접두사와 파일 형식으로 구성되고 저장됩니다.

```
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
```

### 필드

| 필드 | 설명 |
| --- | --- |
| <account_id> | 방이 생성되는 AWS 계정 ID입니다. |
| <hash> | 고유성을 보장하기 위해 시스템에서 생성한 해시 값입니다. |
| <region> | 방이 생성된 AWS 서비스 리전입니다. |
| <resource_id> | 방 ARN의 리소스 ID 부분입니다. |

| 필드 | 설명 |
| --- | --- |
| <version> | 메시지 콘텐츠 형식의 버전입니다. |
| <year> / <month> / <day> / <hours> / <minute> | Amazon IVS Chat에서 메시지를 받은 시점의 UTC 타임스탬프입니다. |

### 예

```
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

### 형식

메시지 로그는 다음과 같은 로그 스트림 이름 형식으로 구성되고 저장됩니다.

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

### 필드

| 필드 | 설명 |
| --- | --- |
| <resource_id> | 방 ARN의 리소스 ID 부분입니다. |
| <version> | 메시지 콘텐츠 형식의 버전입니다. |

### 예

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

# Amazon Kinesis Data Firehose

메시지 로그는 Amazon Redshift, Amazon OpenSearch Service, Splunk 및 사용자 지정 HTTP 엔드포인트 또는 지원되는 서드 파티 서비스 제공업체가 소유한 HTTP 엔드포인트와 같은 대상에 실시간 스트리밍 데이터로 전송됩니다. 자세한 내용은 Amazon Kinesis Data Firehose란 무엇인가요?(What Is Amazon Kinesis Data Firehose?)를 참조하세요.

# 제약 조건

- 메시지가 저장될 로깅 위치를 소유해야 합니다.
- 방, 로깅 구성 및 로깅 위치는 동일한 AWS 리전에 있어야 합니다.
- 채팅 로깅에는 활성 로깅 구성만 사용할 수 있습니다.
- 어느 채팅룸과도 더 이상 연결되지 않는 로깅 구성만 삭제할 수 있습니다.

사용자가 소유한 위치에 메시지를 로깅하려면 AWS 보안 인증으로 권한을 부여해야 합니다. IVS Chat에 필요한 액세스 권한을 부여하려면 로깅 구성을 생성할 때 리소스 정책(Amazon S3 버킷이나 CloudWatch Logs 로그의 경우) 또는 AWS IAM 서비스 연결 역할(SLR)(Amazon Kinesis Data Firehose Firehose의 경우)이 자동으로 생성됩니다. 역할이나 정책을 수정하면 채팅 로깅 권한에 영향을 미칠 수 있으므로 주의해야 합니다.

# Amazon CloudWatch로 오류 모니터링

Amazon CloudWatch를 사용하여 채팅 로깅에서 발생하는 오류를 모니터링할 수 있으며, 특정 오류의 변경 사항을 표시하거나 이에 대응하는 경보나 대시보드를 생성할 수 있습니다.

다음과 같은 여러 유형의 오류가 있습니다. 자세한 내용은 Amazon IVS 챗 모니터링을 참조하세요.

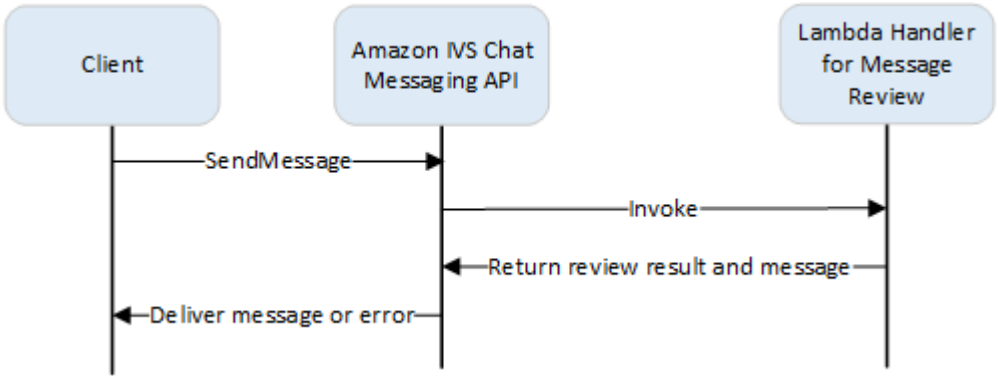
# 채팅 메시지 검토 핸들러

메시지 검토 핸들러를 사용하면 메시지가 방으로 전송되기 전에 메시지를 검토 및/또는 수정할 수 있습니다. 메시지 검토 핸들러가 방과 연결되면 해당 방에 대한 각 SendMessage 요청에 대해 호출됩니다. 핸들러는 애플리케이션의 비즈니스 로직을 적용하고 메시지를 허용, 거부 또는 수정할지 결정합니다. Amazon IVS Chat은 AWS Lambda 함수를 핸들러로 지원합니다.

## Lambda 함수 생성

방에 대한 메시지 검토 핸들러를 설정하기 전에 리소스 기반 IAM 정책으로 lambda 함수를 생성해야 합니다. lambda 함수는 함수를 사용할 방과 동일한 AWS 계정 및 AWS 리전에 있어야 합니다. 리소스 기반 정책은 Amazon IVS Chat에 lambda 함수를 호출할 수 있는 권한을 부여합니다. 지침은 Amazon IVS 챗에 대한 리소스 기반 정책을 참조하세요.

### 워크플로




### Request Syntax

클라이언트가 메시지를 전송하면 Amazon IVS Chat은 JSON 페이로드로 lambda 함수를 호출합니다.

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

```
    }
}
```

## 요청 본문

| 필드 | 설명 |
| --- | --- |
| `Attributes` | 이벤트와 연결된 속성입니다. |
| `Content` | 메시지의 원본 내용입니다. |
| `MessageId` | 메시지 ID입니다. IVS Chat에 의해 생성됩니다. |
| `RoomArn` | 메시지가 전송되는 방의 ARN입니다. |
| `Sender` | 발신자에 대한 정보입니다. 이 객체에는 여러 필드가 있습니다.<br>• `Attributes` - 인증 중 설정된 발신자에 대한 메타데이터입니다. 클라이언트에게 발신자에 대한 추가 정보(예: 아바타 URL, 배지, 글꼴 및 색상)를 제공하는 데 이 메타데이터를 사용할 수 있습니다.<br>• `UserId` - 이 메시지를 보낸 뷰어(최종 사용자)의 애플리케이션 지정 식별자입니다. 클라이언트 애플리케이션에서 메시징 API 또는 애플리케이션 도메인의 사용자를 참조하는 데 이 식별자를 사용할 수 있습니다.<br>• `Ip` - 클라이언트가 요청을 전송하는 IP 주소입니다. |

## Response Syntax

핸들러 lambda 함수는 다음 구문을 사용하여 JSON 응답을 반환해야 합니다. 아래 구문에 해당하지 않거나 필드 제약 조건을 충족하지 않는 응답은 유효하지 않습니다. 이 경우 메시지 검토 핸들러에서 지정한 `FallbackResult` 값에 따라 메시지가 허용되거나 거부됩니다. Amazon IVS Chat API Reference(Amazon IVS Chat API 참조)의 MessageReviewHandler를 참조하세요.

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

# 응답 필드

| 필드 | 설명 |
| --- | --- |
| `Attributes` | lambda 함수에서 반환된 메시지와 연결된 속성입니다.<br><br>`ReviewResult` 가 `DENY`이면 `Reason`이 `Attributes` 에 제공될 수 있습니다. 예를 들면 다음과 같습니다.<br><br>`"Attributes": {"Reason": "denied for moderation`<br><br>이 경우 발신자 클라이언트는 오류 메시지의 이유와 함께 WebSocket 406 오류를 수신합니다. Amazon IVS Chat Messaging API Reference(Amazon IVS Chat Messaging API 참조)의 WebSocket Errors(WebSocket 오류)를 참조하세요.<br><br>• 크기 제약 조건: 최대 1KB<br>• 필수 항목 여부: 아니요 |
| `Content` | Lambda 함수에서 반환된 메시지의 내용입니다. 비즈니스 로직에 따라 편집되었거나 원본일 수 있습니다.<br><br>• 길이 제약: 최소 길이 1. 방을 생성/업데이트할 때 정의한 `MaximumMessageLength` 의 최대 길이입니다. 자세한 내용을 알아보려면, Amazon IVS Chat API Reference(Amazon IVS Chat API 참조)에서 확인하세요. 이는 `ReviewResult` 가 `ALLOW`인 경우에만 적용됩니다.<br>• 필수 여부: 예 |
| `ReviewResult` | 메시지 처리 방법에 대한 검토 처리 결과입니다. 허용되면 방에 연결된 모든 사용자에게 메시지가 전송됩니다. 거부되면 사용자에게 메시지가 전송되지 않습니다.<br><br>• 유효한 값: `ALLOW \| DENY`<br>• 필수 여부: 예 |

# 샘플 코드

다음은 Go의 샘플 lambda 핸들러입니다. 메시지 내용을 수정하고, 메시지 속성을 변경하지 않고 유지하고, 메시지를 허용합니다.

```
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
}
```

# 핸들러를 방과 연결 및 연결 해제

lambda 핸들러를 설정하고 구현했으면 Amazon IVS Chat API를 사용합니다.

- 해당 핸들러를 방과 연결하려면 CreateRoom 또는 UpdateRoom을 호출하고 핸들러를 지정합니다.
- 핸들러를 방에서 연결 해제하려면 `MessageReviewHandler.Uri`에 대해 빈 값을 사용하여 UpdateRoom을 호출합니다.

# Amazon CloudWatch로 오류 모니터링

Amazon CloudWatch를 사용하여 메시지 검토에서 발생하는 오류를 모니터링할 수 있으며, 특정 오류의 변경 사항을 표시하거나 이에 대응하는 경보 또는 대시보드를 생성할 수 있습니다. 오류가 발생하면 핸들러를 방과 연결할 때 지정한 `FallbackResult` 값에 따라 메시지가 허용되거나 거부됩니다. Amazon IVS Chat API Reference(Amazon IVS Chat API 참조)의 MessageReviewHandler를 참조하세요.

다음과 같은 여러 유형의 오류가 있습니다.

- Amazon IVS Chat이 핸들러를 호출할 수 없는 경우 `InvocationErrors`가 발생합니다.
- 핸들러가 잘못된 응답을 반환하는 경우 `ResponseValidationErrors`가 발생합니다.
- lambda 핸들러가 호출되었을 때 함수 오류를 반환하는 경우 AWS Lambda `Errors`가 발생합니다.

간접 호출 오류 및 응답 검증 오류(Amazon IVS 챗에서 발생)에 대한 자세한 내용은 Amazon IVS 챗 모니터링을 참조하세요. AWS Lambda 오류에 대한 자세한 내용을 알아보려면 Lambda 함수 지표 작업을 참조하세요.

# Amazon IVS 챗 모니터링

Amazon CloudWatch를 사용하여 Amazon Interactive Video Service(IVS) 챗 리소스를 모니터링할 수 있습니다. CloudWatch는 Amazon IVS 챗에서 원시 데이터를 수집하고 실시간에 가깝게 읽기 가능한 지표로 프로세싱합니다. 이러한 통계는 15개월간 보관되므로 웹 애플리케이션 또는 서비스의 과거 수행 방식을 파악할 수 있습니다. 특정 임계값을 설정하고 해당 임계값이 충족될 때 알림을 전송하거나 조치를 취할 수 있습니다. 자세한 내용은 CloudWatch 사용 설명서를 참조하세요.

## CloudWatch 지표 액세스

Amazon CloudWatch는 Amazon IVS 챗에서 원시 데이터를 수집하고 실시간에 가깝게 읽기 가능한 지표로 프로세싱합니다. 이러한 통계는 15개월간 보관되므로 웹 애플리케이션 또는 서비스의 과거 수행 방식을 파악할 수 있습니다. 특정 임계값을 설정하고 해당 임계값이 충족될 때 알림을 전송하거나 조치를 취할 수 있습니다. 자세한 내용은 CloudWatch 사용 설명서를 참조하세요.

CloudWatch 지표는 시간이 지나면 롤업됩니다. 지표가 오래되면서 해상도가 실질적으로 감소합니다. 일정은 다음과 같습니다.

- 60초 지표는 15일 동안 사용할 수 있습니다.
- 5분 지표는 63일 동안 사용할 수 있습니다.
- 1시간 지표는 455일(15개월) 동안 사용할 수 있습니다.

데이터 보존에 대한 최신 정보를 보려면 Amazon CloudWatch FAQ에서 '보존 기간'을 검색합니다.

### CloudWatch 콘솔 지침

1. https://console.aws.amazon.com/cloudwatch/에서 CloudWatch 콘솔을 엽니다.
2. 측면 탐색에서 지표(Metrics) 드롭다운을 확장한 다음 모든 지표(All metrics)를 선택합니다.
3. 검색 탭에서 왼쪽에 있는 레이블이 없는 드롭다운을 사용하여 채널이 생성된 '홈' 리전을 선택합니다. 리전에 대한 자세한 내용은 글로벌 솔루션, 리전별 제어를 참조하세요. 지원되는 리전 목록은 AWS 일반 참조의 Amazon IVS 페이지를 참조하세요.
4. 검색 탭 하단에서 IVSChat 네임스페이스를 선택합니다.
5. 다음 중 하나를 수행하십시오.

   a. 검색 창에 리소스 ID(ARN의 일부, `arn:::ivschat:room/<resource id>`)를 입력합니다.

그런 다음 IVSChat을 선택합니다.

b. IVSChat이 AWS 네임스페이스 아래에서 선택 가능한 서비스로 나타나면 이를 선택합니다. IVSChat은 Amazon IVSChat을 사용하고 Amazon CloudWatch에 지표를 전송하는 경우에 나열됩니다. (IVSChat이 목록에 없으면 Amazon IVSChat 지표가 없는 것입니다.)

그리고 나서 필요할 경우 차원 그룹화를 선택하세요. 아래의 CloudWatch 지표에 사용 가능한 차원이 나열되어 있습니다.

6. 지표를 선택하여 그래프에 추가합니다. 아래의 CloudWatch 지표에 사용 가능한 지표가 나열되어 있습니다.

챗 세션의 세부 정보 페이지에서 CloudWatch에서 보기 버튼을 선택하여 챗 세션의 CloudWatch 차트에 액세스할 수도 있습니다.

## CLI 지침

AWS CLI를 사용하여 지표에 액세스할 수도 있습니다. 그러려면 먼저 시스템에 CLI를 다운로드하여 구성해야 합니다. 자세한 내용은 AWS 명령줄 인터페이스 사용 설명서를 참조하세요.

그런 다음 AWS CLI를 사용하여 Amazon IVS 저지연 챗 지표에 액세스하려면 다음을 수행합니다.

- 명령 프롬프트에서 다음을 실행합니다.

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

자세한 내용은 Amazon CloudWatch 사용 설명서에서 Amazon CloudWatch 지표 사용을 참조하세요.

# CloudWatch 지표: IVS 챗

Amazon IVS Chat은 AWS/IVSChat 네임스페이스에서 다음 지표를 제공합니다.

| 지표 | 차원 | 설명 |
| --- | --- | --- |
| `ConcurrentChatConnections` | None | 채팅룸의 총 동시 연결 수(분당 보고된 최대 수)입니다. 이는 고객이 리전의 동시 채팅 연결 한도에 도달하는 경우를 파악하는 데 유용합니다.<br><br>단위: 수 |

| 지표 | 차원 | 설명 |
| --- | --- | --- |
| | | 유효한 통계: 합계, 평균, 최대, 최소 |
| Deliveries | 작업 | 한 리전의 모든 방에서 특정 작업 유형으로 이루어진 채팅 연결에 대한 메시징 요청 전달 횟수입니다.<br><br>단위: 수<br><br>유효한 통계: 합계, 평균, 최대, 최소 |
| InvocationErrors | Uri | 리전의 모든 방에서 특정 메시지 검토 핸들러의 호출 오류 수입니다. 메시지 검토 핸들러를 호출할 수 없는 경우 호출 오류가 발생합니다.<br><br>Amazon IVS Chat이 핸들러를 호출할 수 없는 경우 호출 오류가 발생합니다. 방과 연결된 핸들러가 더 이상 존재하지 않거나 시간 초과된 경우 또는 리소스 정책이 서비스에서 호출하는 것을 허용하지 않는 경우 이러한 오류가 발생할 수 있습니다.<br><br>단위: 수<br><br>유효한 통계: 합계, 평균, 최대, 최소 |
| LogDestinationAccessDeniedError | LoggingConfiguration | 리전의 모든 방에서 로그 대상의 액세스 거부 오류 수입니다.<br><br>Amazon IVS Chat이 로깅 구성에서 지정한 대상 리소스에 액세스할 수 없을 때 이러한 오류가 발생합니다. 대상 리소스 정책이 서비스에서 레코드 저장을 허용하지 않는 경우 이러한 오류가 발생할 수 있습니다.<br><br>단위: 수<br><br>유효한 통계: 합계, 평균, 최대, 최소 |

| 지표 | 차원 | 설명 |
| --- | --- | --- |
| LogDestinationErrors | LoggingConfiguration | 리전의 모든 방에서 로그 대상의 모든 오류 수입니다.<br><br>이것은 Amazon IVS Chat이 로깅 구성에서 지정한 대상 리소스에 로그를 전달하지 못할 때 발생하는 모든 유형의 오류를 포함하는 집계된 지표입니다.<br><br>단위: 수<br><br>유효한 통계: 합계, 평균, 최대, 최소 |
| LogDestinationResourceNotFoundErrors | LoggingConfiguration | 리전의 모든 방에서 로그 대상의 리소스를 찾을 수 없는 오류 수입니다.<br><br>이러한 오류는 Amazon IVS Chat이 로깅 구성에서 지정한 대상 리소스가 존재하지 않아 대상 리소스에 로그를 전달할 수 없을 때 발생합니다. 이것은 로깅 구성과 연결된 대상 리소스가 더 이상 존재하지 않는 경우 발생할 수 있습니다.<br><br>단위: 수<br><br>유효한 통계: 합계, 평균, 최대, 최소 |
| MessagingDeliveries | None | 한 리전의 모든 방에서 채팅 연결에 대한 메시징 요청 전달 횟수입니다.<br><br>단위: 수<br><br>유효한 통계: 합계, 평균, 최대, 최소 |
| MessagingRequests | None | 한 리전의 모든 방에서 이루어진 메시징 요청 수입니다.<br><br>단위: 수<br><br>유효한 통계: 합계, 평균, 최대, 최소 |

| 지표 | 차원 | 설명 |
| --- | --- | --- |
| Requests | 작업 | 한 리전의 모든 방에서 특정 작업 유형으로 이루어진 요청 수입니다.<br><br>단위: 수<br><br>유효한 통계: 합계, 평균, 최대, 최소 |
| ResponseValidationErrors | Uri | 리전의 모든 방에서 특정 메시지 검토 핸들러의 응답-검증 오류 수입니다. 메시지 검토 핸들러의 응답이 유효하지 않은 경우 응답-검증 오류가 발생합니다. 이는 응답을 구문 분석할 수 없거나 검증 확인에 실패했음을 의미할 수 있습니다(예: 잘못된 검토 결과 또는 너무 긴 응답 값).<br><br>단위: 수<br><br>유효한 통계: 합계, 평균, 최대, 최소 |

# Amazon IVS 챗 Client Messaging SDK

Amazon Interactive Video Service(IVS) Chat Client Messaging SDK는 Amazon IVS로 애플리케이션을 구축하는 개발자를 위한 것입니다. 이 SDK는 Amazon IVS 아키텍처를 활용하도록 설계되었으며 Amazon IVS 챗과 함께 업데이트가 제공되고 있습니다. 이 기본 SDK는 애플리케이션 및 사용자가 애플리케이션에 액세스하는 데 사용하는 디바이스에 미치는 성능 영향을 최소화하도록 설계되었습니다.

## 플랫폼 요구 사항

### 데스크톱 브라우저

| 브라우저 | 지원되는 버전 |
| --- | --- |
| Chrome | 두 가지 주요 버전(현재 및 최신 이전 버전) |
| Edge | 두 가지 주요 버전(현재 및 최신 이전 버전) |
| Firefox | 두 가지 주요 버전(현재 및 최신 이전 버전) |
| Opera | 두 가지 주요 버전(현재 및 최신 이전 버전) |
| Safari | 두 가지 주요 버전(현재 및 최신 이전 버전) |

### 모바일 브라우저

| 브라우저 | 지원되는 버전 |
| --- | --- |
| Android용 Chrome | 두 가지 주요 버전(현재 및 최신 이전 버전) |
| Android용 Firefox | 두 가지 주요 버전(현재 및 최신 이전 버전) |
| Android용 Opera | 두 가지 주요 버전(현재 및 최신 이전 버전) |
| WebView Android | 두 가지 주요 버전(현재 및 최신 이전 버전) |

| 브라우저 | 지원되는 버전 |
| --- | --- |
| 삼성 인터넷 | 두 가지 주요 버전(현재 및 최신 이전 버전) |
| iOS용 Safari | 두 가지 주요 버전(현재 및 최신 이전 버전) |

## 기본 플랫폼

| 플랫폼 | 지원되는 버전 |
| --- | --- |
| Android | 5.0 이상 |
| iOS | 13.0 이상 |

# 지원

채팅 룸에서 오류나 기타 문제가 발생하면 IVS 챗 API를 통해 고유한 채팅 룸 식별자를 확인합니다 (ListRooms 참조).

이 채팅룸 식별자를 AWS Support와 공유합니다. 이를 통해 문제를 해결하는 데 도움이 되는 정보를 얻을 수 있습니다.

참고: Amazon IVS 챗 릴리스 정보를 참조하여 사용 가능한 버전 및 해결된 문제를 확인하세요. 해당하는 경우 Support에 문의하기 전에 SDK 버전을 업데이트하고 문제가 해결되는지 확인합니다.

## 버저닝

Amazon IVS 챗 Client Messaging SDK는 유의적 버전 관리를 사용합니다.

이를 설명하기 위해 다음을 가정합니다.

- 최신 릴리스는 버전 4.1.3입니다.
- 이전 주요 버전의 최신 릴리스는 3.2.4입니다.
- 버전 1.x의 최신 릴리스는 1.5.6입니다.

이전 버전과 호환되는 새 기능은 최신 버전의 마이너 릴리스로 추가됩니다. 이 경우 새 기능의 다음 집합이 버전 4.2.0으로 추가됩니다.

이전 버전과 호환되는 마이너 버그 수정은 최신 버전의 패치 릴리스로 추가됩니다. 여기서 마이너 버그의 다음 수정 집합은 버전 4.1.4로 추가됩니다.

이전 버전과 호환되는 메이저 버그 수정은 다르게 처리됩니다. 이러한 버그 수정은 다음과 같이 여러 버전에 추가됩니다.

- 최신 버전의 패치 릴리스에 추가되는 경우. 이 경우 버전 4.1.4입니다.
- 이전 마이너 버전의 패치 릴리스에 추가되는 경우. 이 경우 3.2.5입니다.
- 최신 버전 1.x 릴리스의 패치 릴리스에 추가되는 경우. 이 경우 버전 1.5.7입니다.

메이저 버그 수정은 Amazon IVS 제품 팀에서 정의합니다. 일반적인 예로는 중요한 보안 업데이트와 고객에게 필요한 기타 수정이 있습니다.

참고: 위의 예에서 릴리스된 버전은 숫자를 건너뛰지 않고 증가합니다(예: 4.1.3에서 4.1.4로). 실제로 하나 이상의 패치 번호가 내부에 남아 있고 릴리스되지 않을 수 있으므로, 릴리스된 버전은 4.1.3에서 예를 들어 4.1.6으로 상승할 수 있습니다.

또한, 버전 1.x는 2023년 말까지 또는 3.x가 릴리스될 때까지 중 더 나중에 도래하는 시점에 지원됩니다.

# Amazon IVS 챗 API

서버 측(SDK에서 관리하지 않음)에는 각각 고유한 책임이 있는 두 가지 API가 있습니다.

- 데이터 영역 - IVS 챗 메시징 API는 토큰 기반 인증 체계를 통해 구동되는 프런트엔드 애플리케이션(iOS, Android, macOS 등)에서 사용하도록 설계된 WebSocket API입니다. 이전에 생성된 채팅 토큰을 사용하여 이 API를 사용하는 기존 채팅 룸에 연결할 수 있습니다.

  Amazon IVS 챗 Client Messaging SDK는 데이터 영역에만 관련됩니다. SDK에서는 사용자가 이미 백엔드를 통해 채팅 토큰을 생성하고 있다고 가정합니다. 이러한 토큰의 검색은 SDK가 아닌 프런트엔드 애플리케이션에서 관리하는 것으로 간주됩니다.
- 컨트롤 플레인 - IVS 챗 Control Plane API는 고유한 백엔드 애플리케이션에 대한 인터페이스를 제공하여 채팅 룸과 채팅 룸에 참여하는 사용자를 관리하고 만들 수 있습니다. 자체 백엔드에서 관리하는 앱의 채팅 환경에 대한 관리자 패널로 생각하면 됩니다. 데이터 영역이 채팅 룸을 인증하는 데 필요로 하는 채팅 토큰 생성을 담당하는 컨트롤 플레인 엔드포인트가 있습니다.

중요: IVS 챗 Client Messaging SDK는 컨트롤 플레인 엔드포인트를 호출하지 않습니다. 채팅 토큰을 만들려면 백엔드를 설정해야 합니다. 이 채팅 토큰을 검색하려면 프런트엔드 애플리케이션이 백엔드와 통신해야 합니다.

# Amazon IVS 챗 Client Messaging SDK: Android 설명서

Amazon Interactive Video(IVS) Chat Client Messaging Android SDK는 Android를 사용하는 플랫폼에 IVS 챗 메시징 API를 쉽게 통합할 수 있는 인터페이스를 제공합니다.

`com.amazonaws:ivs-chat-messaging` 패키지는 본 문서에서 설명하는 인터페이스를 구현합니다.

최신 버전의 IVS 챗 클라이언트 메시징 Android SDK: 1.1.0(릴리스 정보)

참조 문서: Amazon IVS 챗 Client Messaging Android SDK에서 사용할 수 있는 가장 중요한 메서드에 대한 자세한 내용은 https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/의 참조 문서를 확인하세요.

샘플 코드: GitHub의 https://github.com/aws-samples/amazon-ivs-chat-for-android-demo에서 Android 샘플 리포지토리를 참조하세요.

플랫폼 요구 사항: 개발 환경에는 Android 5.0(API 레벨 21) 이상이 필요합니다.

## 시작하기

시작하기 전에 Amazon IVS 챗 시작하기의 내용을 숙지해야 합니다.

### 패키지 추가

`com.amazonaws:ivs-chat-messaging`을 `build.gradle` 종속성에 추가합니다.

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

### Proguard 규칙 추가

R8/Proguard 규칙 파일(`proguard-rules.pro`)에 다음 항목을 추가합니다.

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

## 백엔드 설정

이 통합에는 Amazon IVS API와 통신하는 서버의 엔드포인트가 필요합니다. 공식 AWS 라이브러리를 사용하여 서버에서 Amazon IVS API에 액세스합니다. 공개 패키지(예: node.js 및 Java)의 여러 언어로 액세스할 수 있습니다.

다음으로 Amazon IVS 챗 API와 통신하는 서버 엔드포인트를 생성하고 토큰을 생성합니다.

## 서버 연결 설정

`ChatTokenCallback`을 파라미터로 사용하는 메서드를 생성하고 백엔드에서 채팅 토큰을 가져옵니다. 해당 토큰을 콜백의 `onSuccess` 메서드로 전달합니다. 오류가 발생한 경우 예외를 콜백의 `onError` 메서드로 전달합니다. 이는 다음 단계에서 기본 `ChatRoom` 엔터티를 인스턴스화하는 데 필요합니다.

아래에서는 `Retrofit` 호출을 사용하여 위 사항을 구현하는 샘플 코드를 확인할 수 있습니다.

```
// ...

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

# SDK 사용

## 채팅 룸 인스턴스 초기화

ChatRoom 클래스의 인스턴스를 만듭니다. 채팅 룸이 호스팅된 AWS 리전인 regionOrUrl, 그리고 이전 단계에서 생성된 토큰 가져오기 메서드인 tokenProvider의 전달이 필요합니다.

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

다음으로 채팅 관련 이벤트의 핸들러를 구현할 리스너 객체를 만들고, 이를 room.listener 속성에 할당합니다.

```
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

// ...
```

기본 초기화의 마지막 단계는 WebSocket 연결을 설정하여 특정 룸에 연결하는 것입니다. 이를 위해 룸 인스턴스 내에서 connect() 메서드를 호출합니다. 앱이 백그라운드에서 재개되는 경우에도 연결이 유지되도록 onResume() 수명 주기 메서드에서 이를 수행하는 것이 좋습니다.

```
room.connect()
```

SDK는 서버에서 받은 채팅 토큰으로 인코딩된 채팅 룸과의 연결을 설정하려고 시도합니다. 실패하면 룸 인스턴스에 지정된 횟수만큼 다시 연결을 시도합니다.

## 채팅 룸에서 작업 수행

ChatRoom 클래스에는 메시지를 보내고 삭제하고 다른 사용자의 연결을 끊는 작업이 있습니다. 이러한 작업은 요청 확인 또는 거부 알림을 받을 수 있는 선택적 콜백 파라미터를 허용합니다.

메시지 전송

이 요청의 경우 채팅 토큰에 SEND_MESSAGE 기능이 인코딩되어 있어야 합니다.

메시지 전송 요청 트리거:

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

요청의 확인/거부를 받으려면 콜백을 두 번째 파라미터로 제공:

```
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.
   }
```

```
})
```

메시지 삭제

이 요청의 경우 채팅 토큰에 DELETE_MESSAGE 기능이 인코딩되어 있어야 합니다.

메시지 삭제 요청 트리거:

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

요청의 확인/거부를 받으려면 콜백을 두 번째 파라미터로 제공:

```
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.
   }
})
```

다른 사용자 연결 해제

이 요청의 경우 채팅 토큰에 DISCONNECT_USER 기능이 인코딩되어 있어야 합니다.

조정 목적으로 다른 사용자의 연결 해제:

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

요청의 확인/거부를 받으려면 콜백을 두 번째 파라미터로 제공:

```
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.
```

```
    }
})
```

### 채팅 룸 연결 해제

채팅 룸과의 연결을 해제하려면 룸 인스턴스에서 `disconnect()` 메서드를 호출합니다.

```
room.disconnect()
```

애플리케이션이 백그라운드 상태일 때 잠시 후 WebSocket 연결의 작동이 중지되므로 백그라운드 상태로 또는 백그라운드 상태에서 전환할 때 수동으로 연결하거나 연결을 해제하는 것이 좋습니다. 이를 위해 Android `room.connect()` 또는 `onResume()`에서 `Activity` 수명 주기 메서드의 `Fragment` 호출과 `room.disconnect()` 수명 주기 메서드의 `onPause()` 호출을 일치시킵니다.

## Amazon IVS 챗 클라이언트 메시징 SDK: Android 자습서 1부: 채팅 룸

본 문서는 두 파트로 구성된 자습서 중 첫 번째 파트에 해당하는 자습서입니다. Kotlin 프로그래밍 언어를 사용하여 완전한 기능을 갖춘 Android 앱을 구축하여 Amazon IVS 챗 메시징 SDK로 작업하기 위한 필수 사항을 알아봅니다. 여기에서 지칭하는 앱은 Chatterbox라고 합니다.

모듈을 시작하기 전에 몇 분 정도 시간을 내어 사전 조건, 채팅 토큰의 주요 개념, 채팅룸 생성에 필요한 백엔드 서버를 숙지해 두세요.

이 자습서는 IVS 챗 메시징 SDK를 처음 사용하는 숙련된 Android 개발자를 위해 만들어졌습니다. Kotlin 프로그래밍 언어와 Android 플랫폼에서 UI를 만드는 데 익숙해야 합니다.

이 자습서의 첫 번째 부분은 여러 섹션으로 나뉩니다.

1. the section called “로컬 인증/권한 부여 서버 설정”
2. the section called “Chatterbox 프로젝트 생성”
3. the section called “채팅룸에 연결 및 연결 업데이트 관찰”
4. the section called “토큰 공급자 구축”
5. the section called “다음 단계”

전체 SDK 설명서를 보려면 우선 Amazon IVS 챗 클라이언트 메시징 SDK(Amazon IVS 챗 사용 설명서에서 참조) 및 Chat Client Messaging: SDK for Android Reference(GitHub)를 참조하세요.

## 필수 조건

- Kotlin과 Android 플랫폼에서 애플리케이션을 만드는 데 익숙해야 합니다. Android용 애플리케이션을 만드는 데 익숙하지 않은 경우 Android 개발자를 위한 첫 앱 빌드 가이드에서 기본 사항을 배워 보세요.
- IVS 챗 시작하기를 철저하게 읽고 이해합니다.
- 기존 IAM 정책에 정의된 `CreateChatToken` 및 `CreateRoom` 기능을 사용하여 AWS IAM 사용자를 생성합니다. (IVS 챗 시작하기를 참조하세요.)
- 이 사용자의 비밀/액세스 키가 AWS 보안 인증 파일에 저장되어 있는지 확인합니다. 지침은 AWS CLI 사용 설명서(특히 구성 및 보안 인증 파일 설정)를 참조합니다.
- 채팅룸을 생성하고 ARN을 저장합니다. IVS 챗 시작하기를 참조하세요. (ARN을 저장하지 않은 경우 나중에 콘솔이나 Chat API를 사용하여 조회할 수 있습니다.)

## 로컬 인증/권한 부여 서버 설정

백엔드 서버는 채팅룸을 생성하고 IVS 챗 Android SDK가 채팅룸의 클라이언트를 인증하고 권한을 부여하는 데 필요한 채팅 토큰을 생성하는 일을 맡습니다.

Amazon IVS 채팅 시작하기에서 채팅 토큰 생성을 참조하세요. 플로우차트에서 볼 수 있듯이 서버 측 코드는 채팅 토큰 생성을 담당합니다 즉, 앱은 서버 측 애플리케이션에서 채팅 토큰을 요청하여 채팅 토큰을 생성하는 자체 수단을 제공해야 합니다.

저희는 Ktor 프레임워크를 사용하여 로컬 AWS 환경을 통해 채팅 토큰 생성을 관리하는 라이브 로컬 서버를 생성합니다.

이제 AWS 보안 인증 정보가 올바르게 설정되었을 것입니다. 단계별 지침은 개발을 위한 AWS 자격 증명 및 리전 설정을 참조하세요.

`chatterbox`라는 새 디렉터리를 생성하고 그 안에서 또 다른 디렉토리 `auth-server`를 생성합니다.

서버 폴더는 다음과 같은 구조를 갖습니다.

```
- auth-server
  - src
    - main
      - kotlin
        - com
          - chatterbox
            - authserver
```

```
            - Application.kt
        - resources
          - application.conf
          - logback.xml
    - build.gradle.kts
```

참고: 여기에서 코드를 참조된 파일에 직접 복사하거나 붙여넣을 수 있습니다.

다음으로 인증 서버가 작동하는 데 필요한 모든 종속 항목과 플러그인을 추가합니다.

Kotlin 스크립트:

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

이제 인증 서버의 로깅 기능을 설정해야 합니다. (자세한 정보는 로거 구성을 참조하세요.)

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 서버에는 `resources` 디렉터리의 `application.*` 파일에서 자동으로 로드되는 구성 설정이 필요하므로 구성 설정도 추가합니다. (자세한 정보는 파일으로 구성을 참조하세요.)

HOCON:

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

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

마지막으로 서버를 구현해 보겠습니다.

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()
            )
         )
      }
   }
}
```

# Chatterbox 프로젝트 생성

Android 프로젝트를 생성하려면 Android 스튜디오를 설치하고 엽니다.

공식 Android 프로젝트 생성 가이드에 나와 있는 단계를 따릅니다.

- 프로젝트 유형 선택에서 Chatterbox 앱을 위한 빈 활동 프로젝트 템플릿을 선택합니다.
- 프로젝트 구성에서 다음 구성 필드 값을 선택합니다.
  - 이름: My App
  - 패키지 이름: com.chatterbox.myapp
  - 저장 위치: 이전 단계에서 만든chatterbox 디렉터리를 지정합니다.
  - 언어: Kotlin
  - 최소 API 레벨: API 21: Android 5.0(Lollipop)

모든 구성 매개 변수를 올바르게 지정한 후 chatterbox 폴더 내의 파일 구조는 다음과 같아야 합니다.

```
- 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
```

이제 작동하는 Android 프로젝트가 있으므로 `build.gradle` 종속 항목에 com.amazonaws:ivs-chat-messaging을 추가할 수 있습니다. (Gradle 빌드 툴킷에 대한 자세한 정보는 빌드 구성을 참조하세요.)

참고: 모든 코드 조각의 맨 위에는 프로젝트에서 변경해야 하는 파일의 경로가 있습니다. 경로는 프로젝트 루트의 상대 경로입니다.

아래 코드에서 `<version>`을 챗 Android SDK의 현재 버전 번호(예: 1.0.0)로 대체하세요.

Kotlin:

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

plugins {
// ...
}

android {
// ...
}

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

새 종속 항목이 추가된 후 Android 스튜디오에서 Gradle 파일과 프로젝트 동기화를 실행하여 프로젝트를 새 종속 항목과 동기화합니다. (자세한 정보는 빌드 종속 항목 추가를 참조하세요.)

이전 섹션에서 생성한 인증 서버를 프로젝트 루트에서 편리하게 실행하기 위해 이 서버를 `settings.gradle`에 새 모듈로 포함시킵니다. (자세한 정보는 Gradle을 사용하여 소프트웨어 구성 요소 구조화 및 빌드를 참조하세요.)

Kotlin 스크립트:

```
// ./settings.gradle

// ...

rootProject.name = "Chatterbox"
include ':app'
```

```
include ':auth-server'
```

이제부터 auth-server가 Android 프로젝트에 포함되므로 프로젝트 루트에서 다음 명령으로 인증 서버를 실행할 수 있습니다.

쉘:

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

## 채팅룸에 연결 및 연결 업데이트 관찰

채팅룸 연결을 열기 위해 활동이 처음 생성될 때 실행되는 onCreate() 활동 수명 주기 콜백을 사용합니다. ChatRoom 생성자를 사용하려면 룸 연결을 인스턴스화하기 위해 region 및 tokenProvider를 제공해야 합니다.

참고: 아래 조각의 fetchChatToken 함수는 다음 섹션에서 구현됩니다.

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)
   }

// ...
```

```
}
```

채팅룸 연결의 변화를 표시하고 대응하는 것은 chatterbox와 같은 채팅 앱을 만드는 데 필수적인 부분입니다. 룸과 상호작용을 시작하기 전에 채팅룸 연결 상태 이벤트를 구독하여 업데이트를 받아야 합니다.

ChatRoom은 수명 주기 이벤트를 발생시키기 위해 ChatRoomListener 인터페이스 구현을 연결할 것으로 예상합니다. 현재 리스너 함수는 호출 시 확인 메시지만 로그합니다.

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")
        }
    }
}
```

이제 ChatRoomListener를 구현했으므로 룸 인스턴스에 연결해 보겠습니다.

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 {
// ...
}
```

이 다음으로 룸 연결 상태를 읽을 수 있는 기능을 제공해야 합니다. MainActivity.kt 속성에 이를 보관하고 룸의 기본 DISCONNECTED 상태로 초기화합니다(IVS 챗 Android SDK 참조의 ChatRoom state 참조). 로컬 상태를 최신 상태로 유지하려면 state-updater 함수를 구현해야 합니다. 이 함수를 updateConnectionState라고 해 보겠습니다.

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")
         }
      }
   }
}
```

다음으로 state-updater 함수를 ChatRoom.Listener 속성과 통합합니다.

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)
            }
        }
    }
}
```

이제 ChatRoom 상태 업데이트를 저장하고, 듣고, 반응할 수 있게 되었으므로 연결을 초기화할 차례입니다.

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

## 토큰 공급자 구축

이제 애플리케이션에서 채팅 토큰을 생성하고 관리하는 함수를 만들 차례입니다. 이 예에서는 Android용 Retrofit HTTP 클라이언트를 사용합니다.

네트워크 트래픽을 보내려면 먼저 Android용 네트워크 보안 구성을 설정해야 합니다. (자세한 정보는 네트워크 보안 구성을 참조하세요.) 앱 매니페스트 파일에 네트워크 권한을 추가하는 것부터 시작합니다. 새로운 네트워크 보안 구성을 가리키는 추가된 `user-permission` 태그와 `networkSecurityConfig` 속성에 유의하세요. 아래 코드에서 `<version>`을 챗 Android SDK의 현재 버전 번호(예: 1.0.0)로 대체하세요.

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")
}
```

`10.0.2.2` 및 `localhost` 도메인을 신뢰할 수 있는 것으로 선언하여 백엔드와 메시지 교환을 시작합니다.

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>
```

다음으로 HTTP 응답 구문 분석을 위한 Gson 변환기 추가와 함께 새로운 종속 항목을 추가해야 합니다. 아래 코드에서 `<version>`을 챗 Android SDK의 현재 버전 번호(예: 1.0.0)로 대체하세요.

Kotlin 스크립트:

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

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

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

채팅 토큰을 검색하려면 chatterbox 앱에서 POST HTTP 요청을 해야 합니다. Retrofit이 구현할 수 있도록 요청을 인터페이스로 정의합니다. (Retrofit 설명서를 참조하세요. 또한 CreateChatToken 엔드포인트 사양도 숙지하세요.)

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>
}
```

이제 네트워킹을 설정했으므로 채팅 토큰을 생성하고 관리하는 함수를 추가할 차례입니다. 프로젝트가 생성되었을 때 자동으로 생성된 MainActivity.kt에 함수를 추가합니다.

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)
            }
        })
    }
}
```

## 다음 단계

이제 채팅룸 연결을 설정했으므로 이 Android 자습서의 2부인 메시지 및 이벤트로 이동하세요.

# Amazon IVS 챗 클라이언트 메시징 SDK: Android 자습서 2부: 메시지 및 이벤트

이 자습서의 두 번째(마지막) 부분은 여러 섹션으로 나뉩니다.

1. the section called “메시지 전송을 위한 UI 만들기”
   a. the section called “UI 기본 레이아웃”
   b. the section called “텍스트를 일관되게 표시하기 위한 UI 추상화 텍스트 셀”
   c. the section called “UI 왼쪽 채팅 메시지”
   d. the section called “UI 오른쪽 채팅 메시지”
   e. the section called “UI 추가 색상 값”
2. the section called “뷰 결합 적용”
3. the section called “채팅 메시지 요청 관리”
4. the section called “최종 단계”

전체 SDK 설명서를 보려면 우선 Amazon IVS 챗 클라이언트 메시징 SDK(Amazon IVS 챗 사용 설명서에서 참조) 및 Chat Client Messaging: SDK for Android Reference(GitHub)를 참조하세요.

## 전제 조건

이 자습서의 1부인 채팅룸을 완료해야 합니다.

# 메시지 전송을 위한 UI 만들기

채팅룸 연결을 성공적으로 초기화했으므로 이제 첫 번째 메시지를 보낼 차례입니다. 이 기능에는 UI가 필요합니다. 다음을 추가합니다.

- connect/disconnect 버튼
- send 버튼으로 메시지 입력
- 동적 메시지 목록. 이를 빌드하기 위해 Android Jetpack RecyclerView를 사용합니다.

## UI 기본 레이아웃

Android 개발자 문서에서 Android 젯팩 레이아웃을 참조하세요.

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>
```

## 텍스트를 일관되게 표시하기 위한 UI 추상화 텍스트 셀

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 왼쪽 채팅 메시지

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 오른쪽 채팅 메시지

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 추가 색상 값

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>
```

## 뷰 결합 적용

Android 뷰 결합 기능을 활용하여 XML 레이아웃의 결합 클래스를 참조할 수 있습니다. 이 기능을 사용하려면 `./app/build.gradle`의 `viewBinding` 빌드 옵션을 `true`로 설정합니다.

Kotlin 스크립트:

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

android {
//    ...

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

이제 UI를 Kotlin 코드와 연결할 차례입니다.

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)
    }
}
```

또한 채팅 메시지 컨텍스트 메뉴를 사용하여 호출할 수 있는 메시지를 삭제하고 채팅에서 사용자의 연결을 끊는 메서드를 추가합니다.

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}")
                    }
                }
            }
        )
    }
}
```

## 채팅 메시지 요청 관리

가능한 모든 상태를 통해 채팅 메시지 요청을 관리할 수 있는 방법이 필요합니다.

- 보류 중(Pending) - 메시지가 채팅룸에 전송되었지만 아직 확인 또는 거부되지 않았습니다.
- 확인됨(Confirmed) - 우리를 포함한 모든 사용자에게 채팅방에 메시지를 보냈습니다.

- 거부됨(Rejected) - 채팅룸에서 오류 객체가 포함된 메시지를 거부했습니다.

확인되지 않은 채팅 요청과 채팅 메시지는 목록에 보관됩니다. 이 목록에는 `ChatEntries.kt`라는 별도의 클래스가 필요합니다.

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()
    }
}
```

목록을 UI와 연결하기 위해 어댑터를 사용합니다. 자세한 정보는 AdapterView를 사용하여 데이터에 결합 및 생성된 결합 클래스를 참조하세요.

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
}
```

## 최종 단계

이제 ChatEntries 클래스를 MainActivity에 결합하여 새 어댑터를 연결할 차례입니다.

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
        }
    }
}
```

채팅 요청을 계속 추적하는 클래스(ChatEntries)가 이미 있으므로 roomListener에 entries 조작을 위한 코드를 구현할 준비가 되었습니다. 대응 중인 이벤트에 따라 entries 및 connectionState를 업데이트합니다.

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")
        }
    }
}
```

이제 애플리케이션을 실행할 수 있을 것입니다! (앱 빌드 및 실행을 참조하세요.) 앱을 사용할 때는 반드시 백엔드 서버가 실행 중이어야 합니다. 터미널에서 `./gradlew :auth-server:run` 명령어를 프로젝트 루트에서 사용하거나 Android 스튜디오에서 `auth-server:run` Gradle 작업을 직접 실행하여 시작할 수 있습니다.

# Amazon IVS 챗 클라이언트 메시징 SDK: Kotlin 코루틴 자습서 1부: 채팅룸

본 문서는 두 파트로 구성된 자습서 중 첫 번째 파트에 해당하는 자습서입니다. Kotlin 프로그래밍 언어 및 코루틴을 사용하여 완전한 기능을 갖춘 Android 앱을 구축하여 Amazon IVS 챗 메시징 SDK로 작업하기 위한 필수 사항을 알아봅니다. 여기에서 지칭하는 앱은 Chatterbox라고 합니다.

모듈을 시작하기 전에 몇 분 정도 시간을 내어 사전 조건, 채팅 토큰의 주요 개념, 채팅룸 생성에 필요한 백엔드 서버를 숙지해 두세요.

이 자습서는 IVS 챗 메시징 SDK를 처음 사용하는 숙련된 Android 개발자를 위해 만들어졌습니다. Kotlin 프로그래밍 언어와 Android 플랫폼에서 UI를 만드는 데 익숙해야 합니다.

이 자습서의 첫 번째 부분은 여러 섹션으로 나뉩니다.

1. the section called “로컬 인증/권한 부여 서버 설정”
2. the section called “Chatterbox 프로젝트 생성”
3. the section called “채팅룸에 연결 및 연결 업데이트 관찰”
4. the section called “토큰 공급자 구축”
5. the section called “다음 단계”

전체 SDK 설명서를 보려면 우선 Amazon IVS 챗 클라이언트 메시징 SDK(Amazon IVS 챗 사용 설명서에서 참조) 및 Chat Client Messaging: SDK for Android Reference(GitHub)를 참조하세요.

## 필수 조건

- Kotlin과 Android 플랫폼에서 애플리케이션을 만드는 데 익숙해야 합니다. Android용 애플리케이션을 만드는 데 익숙하지 않은 경우 Android 개발자를 위한 첫 앱 빌드 가이드에서 기본 사항을 배워 보세요.
- IVS 챗 시작하기를 읽고 이해합니다.
- 기존 IAM 정책에 정의된 `CreateChatToken` 및 `CreateRoom` 기능을 사용하여 AWS IAM 사용자를 생성합니다. (IVS 챗 시작하기를 참조하세요.)
- 이 사용자의 비밀/액세스 키가 AWS 보안 인증 파일에 저장되어 있는지 확인합니다. 지침은 AWS CLI 사용 설명서(특히 구성 및 보안 인증 파일 설정)를 참조합니다.
- 채팅룸을 생성하고 ARN을 저장합니다. IVS 챗 시작하기를 참조하세요. (ARN을 저장하지 않은 경우 나중에 콘솔이나 Chat API를 사용하여 조회할 수 있습니다.)

## 로컬 인증/권한 부여 서버 설정

백엔드 서버는 채팅룸을 생성하고 IVS 챗 Android SDK가 채팅룸의 클라이언트를 인증하고 권한을 부여하는 데 필요한 채팅 토큰을 생성하는 일을 맡습니다.

Amazon IVS 채팅 시작하기에서 채팅 토큰 생성을 참조하세요. 플로우차트에서 볼 수 있듯이 서버 측 코드는 채팅 토큰 생성을 담당합니다 즉, 앱은 서버 측 애플리케이션에서 채팅 토큰을 요청하여 채팅 토큰을 생성하는 자체 수단을 제공해야 합니다.

저희는 Ktor 프레임워크를 사용하여 로컬 AWS 환경을 통해 채팅 토큰 생성을 관리하는 라이브 로컬 서버를 생성합니다.

이제 AWS 보안 인증 정보가 올바르게 설정되었을 것입니다. 단계별 지침은 Set up AWS temporary credentials and AWS Region for development를 참조하세요.

`chatterbox`라는 새 디렉터리를 생성하고 그 안에서 또 다른 디렉토리 `auth-server`를 생성합니다.

서버 폴더는 다음과 같은 구조를 갖습니다.

```
- auth-server
  - src
    - main
```

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

참고: 여기에서 코드를 참조된 파일에 직접 복사하거나 붙여넣을 수 있습니다.

다음으로 인증 서버가 작동하는 데 필요한 모든 종속 항목과 플러그인을 추가합니다.

Kotlin 스크립트:

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

이제 인증 서버의 로깅 기능을 설정해야 합니다. (자세한 정보는 로거 구성을 참조하세요.)

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 서버에는 `resources` 디렉터리의 `application.*` 파일에서 자동으로 로드되는 구성 설정이 필요하므로 구성 설정도 추가합니다. (자세한 정보는 파일으로 구성을 참조하세요.)

HOCON:

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

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

마지막으로 서버를 구현해 보겠습니다.

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()
            )
         )
      }
   }
```

```
}
```

## Chatterbox 프로젝트 생성

Android 프로젝트를 생성하려면 Android 스튜디오를 설치하고 엽니다.

공식 Android 프로젝트 생성 가이드에 나와 있는 단계를 따릅니다.

- 프로젝트 선택에서 Chatterbox 앱을 위한 빈 활동 프로젝트 템플릿을 선택합니다.

- 프로젝트 구성에서 다음 구성 필드 값을 선택합니다.
  - 이름: My App
  - 패키지 이름: com.chatterbox.myapp
  - 저장 위치: 이전 단계에서 만든`chatterbox` 디렉터리를 지정합니다.
  - 언어: Kotlin
  - 최소 API 레벨: API 21: Android 5.0(Lollipop)

모든 구성 매개 변수를 올바르게 지정한 후 `chatterbox` 폴더 내의 파일 구조는 다음과 같아야 합니다.

```
- 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
```

이제 작동하는 안드로이드 프로젝트가 있으므로 `build.gradle` 종속 항목에 com.amazonaws:ivs-chat-messaging 및 org.jetbrains.kotlinx:kotlinx-coroutines-core를 추가할 수 있습니다. (Gradle 빌드 툴킷에 대한 자세한 정보는 빌드 구성을 참조하세요.)

참고: 모든 코드 조각의 맨 위에는 프로젝트에서 변경해야 하는 파일의 경로가 있습니다. 경로는 프로젝트 루트의 상대 경로입니다.

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'

// ...
}
```

새 종속 항목이 추가된 후 Android 스튜디오에서 Gradle 파일과 프로젝트 동기화를 실행하여 프로젝트를 새 종속 항목과 동기화합니다. (자세한 정보는 빌드 종속 항목 추가를 참조하세요.)

이전 섹션에서 생성한 인증 서버를 프로젝트 루트에서 편리하게 실행하기 위해 이 서버를 `settings.gradle`에 새 모듈로 포함시킵니다. (자세한 정보는 Gradle을 사용하여 소프트웨어 구성 요소 구조화 및 빌드를 참조하세요.)

Kotlin 스크립트:

```
// ./settings.gradle
```

```
// ...

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

이제부터 auth-server가 Android 프로젝트에 포함되므로 프로젝트 루트에서 다음 명령으로 인증 서버를 실행할 수 있습니다.

쉘:

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

## 채팅룸에 연결 및 연결 업데이트 관찰

채팅룸 연결을 열기 위해 활동이 처음 생성될 때 실행되는 onCreate() 활동 수명 주기 콜백을 사용합니다. ChatRoom 생성자를 사용하려면 룸 연결을 인스턴스화하기 위해 region 및 tokenProvider를 제공해야 합니다.

참고: 아래 조각의 fetchChatToken 함수는 다음 섹션에서 구현됩니다.

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)
   }
```

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

채팅룸 연결의 변화를 표시하고 대응하는 것은 chatterbox와 같은 채팅 앱을 만드는 데 필수적인 부분입니다. 룸과 상호작용을 시작하기 전에 채팅룸 연결 상태 이벤트를 구독하여 업데이트를 받아야 합니다.

코루틴용 챗 SDK에서 ChatRoom은 Flow에서 룸 수명 주기 이벤트를 처리할 것으로 예상합니다. 현재 함수는 호출 시 확인 메시지만 로그합니다.

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")
                }
            }
        }
    }
}
```

이 다음으로 룸 연결 상태를 읽을 수 있는 기능을 제공해야 합니다. `MainActivity.kt` 속성에 이를 보관하고 룸의 기본 DISCONNECTED 상태로 초기화합니다(IVS 챗 Android SDK 참조의 `ChatRoom state` 참조). 로컬 상태를 최신 상태로 유지하려면 state-updater 함수를 구현해야 합니다. 이 함수를 `updateConnectionState`라고 해 보겠습니다.

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")
            }
        }
}
```

다음으로 state-updater 함수를 ChatRoom.Listener 속성과 통합합니다.

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)

                }
            }

      // ...

      }
   }
}
```

이제 ChatRoom 상태 업데이트를 저장하고, 듣고, 반응할 수 있게 되었으므로 연결을 초기화할 차례입니다.

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)
      }
   }

   // ...
}
```

## 토큰 공급자 구축

이제 애플리케이션에서 채팅 토큰을 생성하고 관리하는 함수를 만들 차례입니다. 이 예에서는 Android용 Retrofit HTTP 클라이언트를 사용합니다.

네트워크 트래픽을 보내려면 먼저 Android용 네트워크 보안 구성을 설정해야 합니다. (자세한 정보는 네트워크 보안 구성을 참조하세요.) 앱 매니페스트 파일에 네트워크 권한을 추가하는 것부터 시작합니다. 새로운 네트워크 보안 구성을 가리키는 추가된 `user-permission` 태그와 `networkSecurityConfig` 속성에 유의하세요. 아래 코드에서 *<version>*을 챗 Android SDK의 현재 버전 번호(예: 1.1.0)로 대체하세요.

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")
}
```

로컬 IP 주소(예:`10.0.2.2` 및 `localhost` 도메인)를 신뢰할 수 있는 것으로 선언하여 백엔드와 메시지 교환을 시작합니다.

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>
```

다음으로 HTTP 응답 구문 분석을 위한 Gson 변환기 추가와 함께 새로운 종속 항목을 추가해야 합니다. 아래 코드에서 *<version>*을 챗 Android SDK의 현재 버전 번호(예: 1.1.0)로 대체하세요.

Kotlin 스크립트:

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

채팅 토큰을 검색하려면 chatterbox 앱에서 POST HTTP 요청을 해야 합니다. Retrofit이 구현할 수 있도록 요청을 인터페이스로 정의합니다. (Retrofit 설명서를 참조하세요. 또한 CreateChatToken 엔드포인트 사양도 숙지하세요.)

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)
    }
}
```

이제 네트워킹을 설정했으므로 채팅 토큰을 생성하고 관리하는 함수를 추가할 차례입니다. 프로젝트가 생성되었을 때 자동으로 생성된 `MainActivity.kt`에 함수를 추가합니다.

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)
        }
    })
  }
}
```

## 다음 단계

이제 채팅룸 연결을 설정했으므로 이 Kotlin 코루틴 자습서의 2부인 메시지 및 이벤트로 이동하세요.

# Amazon IVS 챗 클라이언트 메시징 SDK: Kotlin 코루틴 자습서 2부: 메시지 및 이벤트

이 자습서의 두 번째(마지막) 부분은 여러 섹션으로 나뉩니다.

1. the section called “메시지 전송을 위한 UI 만들기”
   a. the section called “UI 기본 레이아웃”
   b. the section called “텍스트를 일관되게 표시하기 위한 UI 추상화 텍스트 셀”
   c. the section called “UI 왼쪽 채팅 메시지”
   d. the section called “UI 오른쪽 메시지”
   e. the section called “UI 추가 색상 값”
2. the section called “뷰 결합 적용”
3. the section called “채팅 메시지 요청 관리”

4. the section called “최종 단계”

전체 SDK 설명서를 보려면 우선 Amazon IVS 챗 클라이언트 메시징 SDK(Amazon IVS 챗 사용 설명서에서 참조) 및 Chat Client Messaging: SDK for Android Reference(GitHub)를 참조하세요.

## 전제 조건

이 자습서의 1부인 채팅룸을 완료해야 합니다.

## 메시지 전송을 위한 UI 만들기

채팅룸 연결을 성공적으로 초기화했으므로 이제 첫 번째 메시지를 보낼 차례입니다. 이 기능에는 UI가 필요합니다. 다음을 추가합니다.

- connect/disconnect 버튼
- send 버튼으로 메시지 입력
- 동적 메시지 목록. 이를 빌드하기 위해 Android Jetpack RecyclerView를 사용합니다.

### UI 기본 레이아웃

Android 개발자 문서에서 Android 젯팩 레이아웃을 참조하세요.

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>
```

## 텍스트를 일관되게 표시하기 위한 UI 추상화 텍스트 셀

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 왼쪽 채팅 메시지

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 오른쪽 메시지

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 추가 색상 값

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>
```

## 뷰 결합 적용

Android 뷰 결합 기능을 활용하여 XML 레이아웃의 결합 클래스를 참조할 수 있습니다. 이 기능을 사용하려면 `./app/build.gradle`의 `viewBinding` 빌드 옵션을 `true`로 설정합니다.

Kotlin 스크립트:

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

android {
//    ...

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

이제 UI를 Kotlin 코드와 연결할 차례입니다.

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)
    }
// ...

}
```

또한 채팅 메시지 컨텍스트 메뉴를 사용하여 호출할 수 있는 메시지를 삭제하고 채팅에서 사용자의 연결을 끊는 메서드를 추가합니다.

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")
           }
        }
    }
```

```
}
```

## 채팅 메시지 요청 관리

가능한 모든 상태를 통해 채팅 메시지 요청을 관리할 수 있는 방법이 필요합니다.

- 보류 중(Pending) - 메시지가 채팅룸에 전송되었지만 아직 확인 또는 거부되지 않았습니다.
- 확인됨(Confirmed) - 우리를 포함한 모든 사용자에게 채팅방에 메시지를 보냈습니다.
- 거부됨(Rejected) - 채팅룸에서 오류 객체가 포함된 메시지를 거부했습니다.

확인되지 않은 채팅 요청과 채팅 메시지는 목록에 보관됩니다. 이 목록에는 `ChatEntries.kt`라는 별도의 클래스가 필요합니다.

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()
    }
}
```

목록을 UI와 연결하기 위해 어댑터를 사용합니다. 자세한 정보는 AdapterView를 사용하여 데이터에 결합 및 생성된 결합 클래스를 참조하세요.

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
}
```

## 최종 단계

이제 ChatEntries 클래스를 MainActivity에 결합하여 새 어댑터를 연결할 차례입니다.

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
        }
    }
}
```

채팅 요청을 계속 추적하는 클래스(ChatEntries)가 이미 있으므로 roomListener에 entries 조작을 위한 코드를 구현할 준비가 되었습니다. 대응 중인 이벤트에 따라 entries 및 connectionState를 업데이트합니다.

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)
    }
```

```
// ...

}
```

이제 애플리케이션을 실행할 수 있을 것입니다! (앱 빌드 및 실행을 참조하세요.) 앱을 사용할 때는 반드시 백엔드 서버가 실행 중이어야 합니다. 터미널에서 `./gradlew :auth-server:run` 명령어를 프로젝트 루트에서 사용하거나 Android 스튜디오에서 `auth-server:run` Gradle 작업을 직접 실행하여 시작할 수 있습니다.

# Amazon IVS Chat Client Messaging SDK: iOS 설명서

Amazon Interactive Video(IVS) Chat Client Messaging iOS SDK는 Apple의 Swift 프로그래밍 언어를 사용하는 플랫폼에 IVS Chat Messaging API를 통합할 수 있는 인터페이스를 제공합니다.

IVS Chat Client Messaging iOS SDK의 최신 버전: 1.0.0(릴리스 정보)

참조 문서 및 자습서: Amazon IVS Chat Client Messaging iOS SDK에서 사용할 수 있는 가장 중요한 메서드에 대한 자세한 내용은 https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/의 참조 문서를 확인하세요. 이 리포지토리에는 다양한 기사와 자습서도 포함되어 있습니다.

샘플 코드: GitHub의 https://github.com/aws-samples/amazon-ivs-chat-for-ios-demo에서 iOS 샘플 리포지토리를 참조하세요.

플랫폼 요구 사항: 개발 환경에는 iOS 13.0 이상이 필요합니다.

## 시작하기

Swift Package Manager를 사용하여 SDK를 통합하는 것을 권장합니다. 또는 CocoaPods를 사용하거나 수동으로 프레임워크를 통합할 수 있습니다.

SDK를 통합한 후 관련 Swift 파일 상단에 다음 코드를 추가하여 SDK를 가져올 수 있습니다.

```
import AmazonIVSChatMessaging
```

### Swift Package Manager

Swift Package Manager 프로젝트에서 `AmazonIVSChatMessaging` 라이브러리를 사용하려면 이를 패키지의 종속성과 관련 대상의 종속성에 추가합니다.

1. https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip에서 최신 `.xcframework`를 다운로드합니다.
2. 터미널에서 다음을 실행합니다.

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

3. 프로젝트의 `Package.swift` 파일에 표시된 것과 같이 이전 단계의 출력을 가져와서 `.binaryTarget`의 체크섬(checksum) 속성에 붙여넣습니다.

```
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

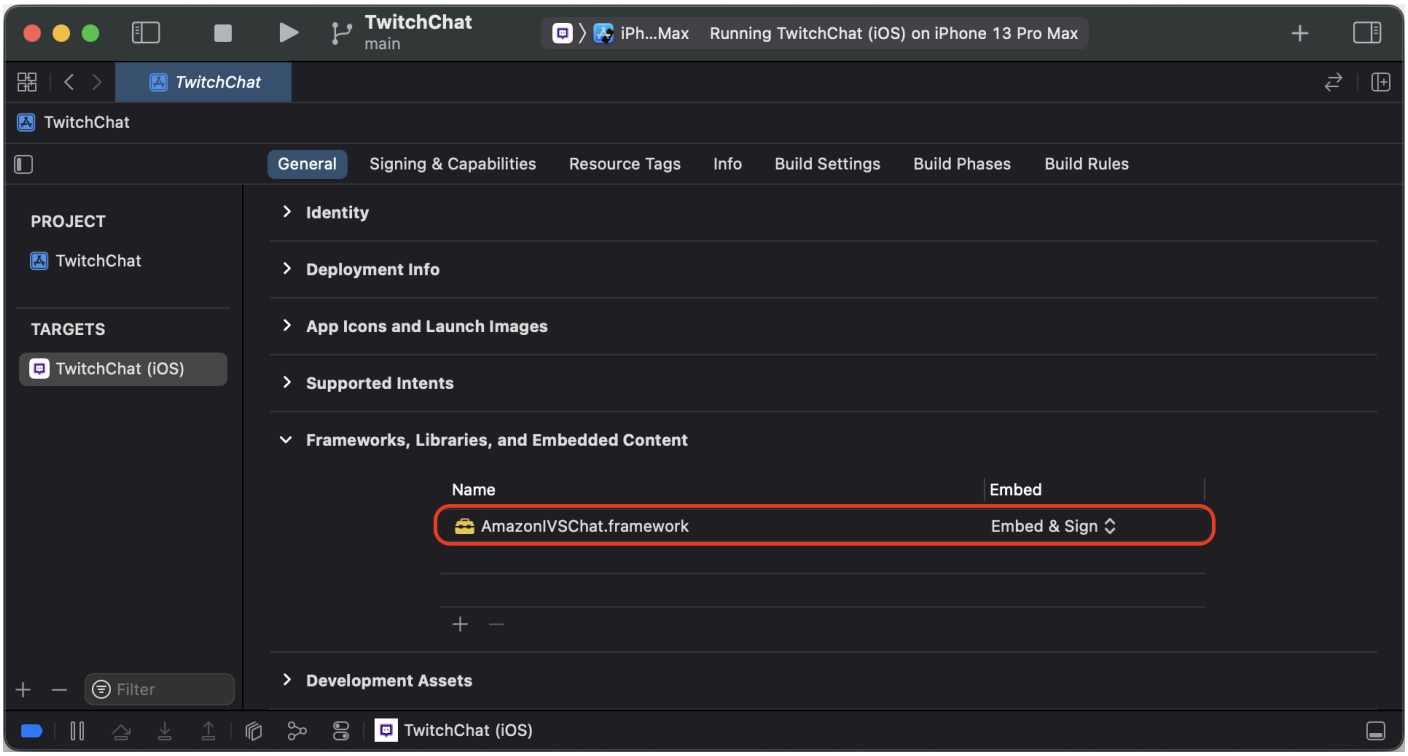
릴리스는 `AmazonIVSChatMessaging`이라는 이름으로 CoCocoaPods를 통해 게시됩니다. 이 종속성을 Podfile에 추가합니다.

```
pod 'AmazonIVSChat'
```

이후 `pod install`을 실행하면 `.xcworkspace`에서 SDK를 사용할 수 있습니다.

### 수동 설치

1. https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip에서 최신 버전을 다운로드합니다.
2. 아카이브 콘텐츠의 압축을 풉니다. `AmazonIVSChatMessaging.xcframework`에는 디바이스와 시뮬레이터 모두에 대한 SDK가 포함되어 있습니다.
3. 애플리케이션 대상에 대해 General(일반) 탭의 프레임워크, 라이브러리 및 포함된 콘텐츠(Frameworks, Libraries, and Embedded Content) 섹션으로 끌어 추출된 `AmazonIVSChatMessaging.xcframework`를 포함합니다.




## SDK 사용

### 채팅 룸에 연결

시작하기 전에 Amazon IVS Chat 시작하기의 내용을 숙지해야 합니다. 또한 웹, Android 및 iOS에 대한 예제 앱도 참조하세요.

채팅 룸에 연결하려면 앱에 백엔드에서 제공한 채팅 토큰을 검색할 수 있는 방법이 필요합니다. 애플리케이션은 아마도 백엔드에 대한 네트워크 요청을 사용하여 채팅 토큰을 검색할 것입니다.

가져온 채팅 토큰을 SDK와 통신하려면 SDK의 ChatRoom 모델에서 async 함수 또는 초기화 시점에서 제공된 ChatTokenProvider 프로토콜을 따르는 객체의 인스턴스를 제공해야 합니다. 이러한 메서드 중 하나에서 반환되는 값은 SDK의 ChatToken 모델의 인스턴스여야 합니다.

참고: 백엔드에서 검색된 데이터를 사용하여 ChatToken 모델의 인스턴스를 채웁니다. ChatToken 인스턴스의 초기화에 필요한 필드는 CreateChatToken 응답의 필드와 동일합니다. ChatToken 모델의 인스턴스 초기화에 대한 자세한 내용은 ChatToken의 인스턴스 생성을 참조하세요. 백엔드에서 앱에 대한 CreateChatToken 응답의 데이터를 제공해야 합니다. 채팅 토큰을 생성하기 위해 백엔드와 통신하는 방법은 앱과 인프라에 따라 달라집니다.

ChatToken을 SDK로 제공하는 방법을 선택한 후, 백엔드에서 연결하려는 채팅 룸을 생성하는 데 사용한 토큰 공급자와 AWS 리전으로 ChatRoom 인스턴스를 초기화하고 .connect()를 호출합니다. .connect()는 비동기 함수를 반환합니다.

```
import AmazonIVSChatMessaging

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

### ChatTokenProvider 프로토콜 준수

ChatRoom에 대한 이니셜라이저의 tokenProvider 파라미터에 대해 ChatTokenProvider의 인스턴스를 제공할 수 있습니다. 다음은 ChatTokenProvider를 준수하는 객체의 예입니다.

```
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
      )
   }
```

```
}
```

그런 다음 이 준수 객체의 인스턴스를 가져와 ChatRoom의 이니셜라이저에 전달할 수 있습니다.

```
// 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()
```

Swift에서 비동기 함수 제공

애플리케이션의 네트워크 요청을 관리하는 데 사용하는 관리자가 이미 있다고 가정해 봅시다. 값이 다음과 같을 것입니다.

```
import AmazonIVSChatMessaging

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

관리자에 다른 함수를 추가하여 백엔드에서 ChatToken을 검색할 수 있습니다.

```
import AmazonIVSChatMessaging

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

그런 다음 ChatRoom을 초기화할 때 Swift에서 해당 함수에 대한 참조를 사용합니다.

```
import AmazonIVSChatMessaging

let endpointManager: EndpointManager
```

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

## ChatToken의 인스턴스 생성

SDK에 제공된 이니셜라이저를 사용하여 ChatToken의 인스턴스를 쉽게 만들 수 있습니다. Token.swift의 설명서에서 ChatToken의 속성에 대해 자세히 알아보세요.

```
import AmazonIVSChatMessaging

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

### Decodable 사용

IVS Chat API와 인터페이스하는 동안 백엔드에서 프런트엔드 애플리케이션으로 CreateChatToken 응답을 전달하기로 결정한 경우 ChatToken의 Swift Decodable 프로토콜 준수를 활용할 수 있습니다. 하지만 한 가지 문제가 있습니다.

CreateChatToken 응답 페이로드는 ISO 8601 인터넷 타임스탬프 표준을 사용하여 형식이 지정된 날짜 문자열을 사용합니다. 일반적으로 Swift에서 JSONDecoder.DateDecodingStrategy.iso8601을 JSONDecoder의 .dateDecodingStrategy속성에 대한 값으로 제공합니다. 하지만 CreateChatToken은 문자열에 고정밀 분수 초를 사용하며, 이는 JSONDecoder.DateDecodingStrategy.iso8601에서 지원되지 않습니다.

편의를 위해 SDK는 JSONDecoder.DateDecodingStrategy에 대한 공개 확장 프로그램에 ChatToken의 인스턴스를 디코딩할 때 성공적으로 JSONDecoder를 사용할 수 있도록 하는 추가 .preciseISO8601 전략을 제공합니다.

```
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)
```

## 채팅 룸 연결 해제

성공적으로 연결한 ChatRoom 인스턴스에서 수동으로 연결을 해제하려면 room.disconnect()를 호출합니다. 기본적으로 채팅 룸은 할당이 취소되면 이 함수를 자동으로 호출합니다.

```
import AmazonIVSChatMessaging

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

// Disconnect
room.disconnect()
```

## 채팅 메시지/이벤트 받기

채팅 룸에서 메시지를 보내고 받으려면 ChatRoom의 인스턴스를 성공적으로 초기화하고 room.connect()를 호출한 후 ChatRoomDelegate 프로토콜을 준수하는 객체를 제공해야 합니다. UIViewController 사용의 일반적인 예제:

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

## 연결 변경 시 알림 받기

예상대로 완전히 연결되기 전까지는 메시지를 보내는 등의 작업을 수행할 수 없습니다. SDK의 아키텍처는 비동기 API를 통해 백그라운드 스레드에서 ChatRoom 연결을 권장하려 합니다. 메시지 전송 버튼 등을 비활성화하는 기능을 UI에 빌드하려는 경우 SDK는 채팅 룸의 연결 상태가 변경될 때 알림을 받는 두 가지 방법으로 `Combine` 또는 `ChatRoomDelegate` 사용을 제공합니다. 아래에서 설명합니다.

중요: 네트워크 연결 끊어짐 등으로 인해 채팅 룸의 연결 상태도 변경될 수 있습니다. 앱을 빌드할 때 이 점을 고려하세요.

### Combine 사용

`ChatRoom`의 모든 인스턴스에는 `state` 속성의 형식으로 자체 `Combine` 게시자가 제공됩니다.

```
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()
}
```

### ChatRoomDelegate 사용

또는 ChatRoomDelegate를 준수하는 객체 내에서 `roomDidConnect(_:)`, `roomIsConnecting(_:)` 및 `roomDidDisconnect(_:)` 함수를 사용합니다. 다음은 UIViewController 사용의 예입니다.

```
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!")
    }
}
```

## 채팅 룸에서 작업 수행

채팅 룸에서 수행할 수 있는 작업(예: 메시지 전송, 메시지 삭제, 사용자 연결 해제)이 가능한지 여부는 사용자마다 다릅니다. 이러한 작업 중 하나를 수행하려면 연결된 ChatRoom에서 perform(request:)를 호출하고, SDK에서 제공된 ChatRequest 객체 중 하나의 인스턴스를 전달합니다. 지원되는 요청은 Request.swift입니다.

채팅 룸의 일부 작업을 수행하려면 백엔드 애플리케이션이 CreateChatToken을 호출할 때 연결된 사용자에게 특정 기능을 부여해야 합니다. 설계상 SDK는 연결된 사용자의 기능을 식별할 수 없습니다. 따라서 ChatRoom의 연결된 인스턴스에서 중재자 작업을 수행해 볼 수 있습니다. 컨트롤 플레인 API는 최종적으로 해당 작업의 성공 여부를 결정합니다.

room.perform(request:)을 거치는 모든 작업은 룸에서 수신한 모델과 요청 객체 모두의 requestId와 일치하는 모델의 인스턴스(유형은 요청 객체 자체와 연결됨)를 수신할 때까지 대기합니다. 요청에 문제가 있는 경우 ChatRoom은 항상 ChatError 형식으로 오류를 반환합니다. ChatError의 정의는 Error.swift입니다.

### 메시지 전송

채팅 메시지를 보내려면 SendMessageRequest의 인스턴스를 사용합니다.

```
import AmazonIVSChatMessaging

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

위에서 언급한 바와 같이 room.perform(request:)은 ChatRoom에서 ChatMessage를 수신하면 반환됩니다. 요청에 문제가 있는 경우(예: 룸의 메시지 문자 제한 초과) ChatError의 인스턴스가 대신 반환됩니다. 이후 유용한 정보를 UI에 표시할 수 있습니다.

```
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)
}
```

메시지에 메타데이터 추가

메시지를 보낼 때 관련 메타데이터를 추가할 수 있습니다. SendMessageRequest에는 attributes 속성이 있고, 이 속성을 사용하여 요청을 초기화할 수 있습니다. 첨부한 데이터는 다른 사람들이 채팅 룸에서 해당 메시지를 받을 때 메시지에 첨부됩니다.

다음은 보내는 메시지에 이모트 데이터를 첨부하는 예시입니다.

```
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"
      ]
   )
```

```
)
```

SendMessageRequest에서 attributes를 사용하는 것이 채팅 제품에서 복잡한 기능을 빌드하는 데 매우 유용할 수 있습니다. 예를 들어 SendMessageRequest의 [String : String] 속성 사전을 사용하여 스레딩 기능을 빌드할 수 있습니다!

attributes 페이로드는 매우 유연하고 강력합니다. 이를 사용하여 다른 방법으로는 할 수 없는 메시지 관련 정보를 도출하세요. 예를 들어 메시지 문자열을 구문 분석하여 이모트 등의 정보를 가져오는 것보다 속성을 사용하는 것이 훨씬 쉽습니다.

메시지 삭제

채팅 메시지를 삭제하는 것은 메시지를 보내는 것과 같습니다. ChatRoom에서 room.perform(request:) 함수를 호출하고 DeleteMessageRequest의 인스턴스를 생성하면 됩니다.

받은 챗 메시지의 이전 인스턴스에 쉽게 액세스하려면 DeleteMessageRequest의 이니셜라이저로 message.id의 값을 전달합니다.

필요한 경우 UI에 표시할 수 있도록 이유 문자열을 DeleteMessageRequest에 제공합니다.

```
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!"
   )
)
```

이는 중재자 작업이며, 사용자는 실제로 다른 사용자의 메시지를 삭제할 수 있는 권한이 없을 수 있습니다. 사용자가 적절한 기능 없이 메시지를 삭제하려고 할 때 Swift의 반환 가능한 함수 메커니즘을 사용하여 UI에 오류 메시지를 표시할 수 있습니다.

백엔드에서 사용자에 대해 CreateChatToken을 호출할 때 "DELETE_MESSAGE"를 capabilities 필드를 전달하여 연결된 채팅 사용자에 대해 해당 기능을 활성화합니다.

다음은 적절한 권한 없이 메시지를 삭제하려고 할 때 발생하는 기능 오류의 예입니다.

```
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)
}
```

다른 사용자 연결 해제

`room.perform(request:)`을 사용하여 채팅 룸에서 다른 사용자의 연결을 해제합니다. 구체적으로 `DisconnectUserRequest`의 인스턴스를 사용합니다. `ChatRoom`에서 수신하는 모든 `ChatMessage`에는 `sender` 속성이 있으며, `DisconnectUserRequest`의 인스턴스로 적절히 초기화해야 하는 사용자 ID가 포함되어 있습니다. 연결 해제 요청에 대한 이유 문자열을 제공할 수도 있습니다.

```
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
   )
)
```

다음은 중재자 작업의 또 다른 예이며, 다른 사용자의 연결을 해제하려고 할 수 있지만 DISCONNECT_USER 기능이 없는 한 연결을 해제할 수 없습니다. 기능은 백엔드 애플리케이션이 CreateChatToken을 호출하고 "DISCONNECT_USER" 문자열을 capabilities 필드에 주입할 때 설정됩니다.

사용자에게 다른 사용자의 연결을 해제하는 기능이 없는 경우 room.perform(request:)은 다른 요청과 마찬가지로 ChatError의 인스턴스를 반환합니다. 중재자 권한 부족으로 인해 요청이 실패하는지 확인하기 위해 오류의 errorCode 속성을 검사할 수 있습니다.

```
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)
}
```

# Amazon IVS Chat Client Messaging SDK: iOS 자습서

Amazon Interactive Video(IVS) Chat Client Messaging iOS SDK는 Apple의 Swift 프로그래밍 언어를 사용하는 플랫폼에 IVS Chat Messaging API를 통합할 수 있는 인터페이스를 제공합니다.

Chat iOS SDK 자습서는 https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios를 참조하세요.

# Amazon IVS Chat Client Messaging SDK: JavaScript 설명서

Amazon Interactive Video(IVS) Chat Client Messaging JavaScript SDK를 사용하면 웹 브라우저를 사용하는 플랫폼에서 IAmazon IVS Chat Messaging API를 통합할 수 있습니다.

IVS Chat Client Messaging JavaScript SDK의 최신 버전: 1.0.2(릴리스 정보)

참조 문서: Amazon IVS Chat Client Messaging JavaScrip SDK에서 사용할 수 있는 가장 중요한 메서드에 대한 자세한 내용은 https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/의 참조 문서를 확인하세요.

샘플 코드: JavaScript SDK를 사용하는 웹 전용 데모는 GitHub의 샘플 리포지토리를 참조하십시오. https://github.com/aws-samples/amazon-ivs-chat-web-demo

## 시작하기

시작하기 전에 Amazon IVS 챗 시작하기의 내용을 숙지해야 합니다.

### 패키지 추가

다음을 사용하십시오.

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

또는:

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

### React Native Support

IVS 채팅 클라이언트 메시징 JavaScript SDK에는 `crypto.getRandomValues` 메서드를 사용하는 `uuid` 종속성이 있습니다. 이 메서드는 React Native에서 지원되지 않으므로 추가 폴리필을 설치하고 `react-native-get-random-value` `index.js` 파일 상단에서 가져와야 합니다.

```
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);
```

### 백엔드 설정

이 통합에는 Amazon IVS Chat API와 통신하는 서버의 엔드포인트가 필요합니다. 공식 AWS 라이브러리를 사용하여 서버에서 Amazon IVS API에 액세스합니다. 공개 패키지(예: node.js, java, go)의 여러 언어로 액세스할 수 있습니다.

Amazon IVS Chat API CreateChatToken 엔드포인트와 통신하는 서버 엔드포인트를 생성하여 채팅 사용자를 위한 채팅 토큰을 생성합니다.

## SDK 사용

### 채팅 룸 인스턴스 초기화

`ChatRoom` 클래스의 인스턴스를 만듭니다. 인스턴스 생성을 위해`regionOrUrl`(채팅룸이 호스팅되는 AWS 리전) 및`tokenProvider`(토큰 가져오기 메서드는 다음 단계에서 생성)를 통과해야 합니다.

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

### 토큰 공급자 함수

백엔드에서 채팅 토큰을 가져오는 비동기(asynchronous) 토큰 공급자 함수를 생성합니다.

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

함수는 매개 변수를 받지 않아야 하며 채팅 토큰 객체가 포함된 Promise를 반환해야 합니다.

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

```
}
```

이 함수는 ChatRoom 객체를 초기화하는 데 필요하므로 아래의 내용 중<token> 및<date-time> 필드에 백엔드에서 받은 값을 입력하세요.

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

반드시 ChatRoomtokenProvider 생성자에 전달하세요. ChatRoom은 연결이 중단되거나 세션이 만료되면 토큰을 새로 고침 합니다. ChatRoom에서 자동으로 토큰을 처리하므로 토큰을 다른 곳에 저장하는 데 tokenProvider를 사용하지 마세요.

## 이벤트 수신

다음으로 채팅룸 이벤트를 구독하여 대화방에서 전달되는 메시지와 이벤트뿐만 아니라 라이프사이클 이벤트를 수신하세요.

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

## 채팅룸에 연결

기본 초기화의 마지막 단계는 WebSocket 연결을 설정하여 특정 룸에 연결하는 것입니다. 이를 위해 룸 인스턴스 내에서 `connect()` 메서드를 호출합니다.

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

SDK가 서버에서 받은 채팅 토큰으로 인코딩된 채팅룸과의 연결을 설정하려고 시도합니다.

`connect()`을 호출하면 룸이`connecting` 상태로 전환되어`connecting` 이벤트가 발생합니다. 룸이 성공적으로 연결되면`connected` 상태로 전환되고`connect` 이벤트가 발생합니다.

토큰을 가져오거나 WebSocket에 연결할 때 문제가 발생하여 연결 실패가 발생할 수 있습니다. 이 경우, 룸은`maxReconnectAttempts` 생성자 매개변수에 표시된 횟수까지 자동으로 다시 연결을 시도합니다. 재연결을 시도하는 동안에는 룸이`connecting` 상태이며 추가 이벤트가 발생하지 않습니다. 재연결 시도 횟수를 모두 소진하면 해당 룸이`disconnected` 상태로 전환되고 관련 연결 해제 이유가 포함된`disconnect` 이벤트가 발생합니다. `disconnected` 상태에서는 룸이 더 이상 연결을 시도하지 않습니다. 연결 프로세스를 트리거하려면 `connect()`를 다시 호출해야 합니다.

## 채팅룸에서 작업 수행

Amazon IVS Chat Messaging SDK는 메시지 전송, 메시지 삭제 및 다른 사용자 연결 해제를 위한 사용자 작업을 제공합니다. 이들 기능은 `ChatRoom` 인스턴스에서 사용할 수 있습니다. 요청 확인 또는 거부 수신을 허용하는 `Promise` 객체를 반환합니다.

메시지 전송

이 요청의 경우 채팅 토큰에 `SEND_MESSAGE` 기능이 인코딩되어 있어야 합니다.

메시지 전송 요청 트리거:

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

요청에 대한 확인 또는 거부를 얻으려면 await 반환된 약속(promise)을 받거나 다음 then() 메서드를 사용하세요.

```
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.
}
```

메시지 삭제

이 요청의 경우 채팅 토큰에 DELETE_MESSAGE 기능이 인코딩되어 있어야 합니다.

조절 목적으로 메시지를 삭제하려면 다음deleteMessage() 메서드를 호출하십시오.

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

요청에 대한 확인 또는 거부를 얻으려면 await 반환된 약속(promise)을 받거나 다음 then() 메서드를 사용하세요.

```
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.
}
```

다른 사용자 연결 해제

이 요청의 경우 채팅 토큰에 DISCONNECT_USER 기능이 인코딩되어 있어야 합니다.

조절 목적으로 다른 사용자의 연결 해제disconnectUser():

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

요청에 대한 확인 또는 거부를 얻으려면 await 반환된 약속(promise)을 받거나 다음 then() 메서드를 사용하세요.

```
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.
}
```

### 채팅룸 연결 해제

채팅룸과의 연결을 해제하려면 `room` 인스턴스에서 `disconnect()` 메서드를 호출하세요.

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

이 메서드를 호출하면 룸이 기본 WebSocket을 순서대로 닫습니다. 룸 인스턴스가`disconnected` 상태로 전환되고 `disconnect` 이유가 `"clientDisconnect"`로 설정된 연결 해제 이벤트가 발생합니다.

# Amazon IVS Chat Client Messaging SDK: JavaScript 자습서 1부: 채팅룸

본 문서는 두 파트로 구성된 자습서 중 첫 번째 파트에 해당하는 자습서입니다. 이 자습서에서는 JavaScript/TypeScript를 통해 완전한 기능을 갖춘 앱을 구축하여 Amazon IVS Chat Client Messaging JavaScript SDK로 작업하기 위한 필수 사항을 설명하고 있습니다. 여기에서 지칭하는 앱은 Chatterbox라고 합니다.

이 자습서는 숙련된 개발자이나, Amazon IVS Chat Messaging SDK를 처음 접하는 사용자를 위해 작성되었습니다. 사용자는 이미 JavaScript/TypeScript 프로그래밍 언어와 React 라이브러리에 대한 친숙도가 있어야 합니다.

간략하게 나타내자면 Amazon IVS Chat Client Messaging JavaScript SDK를 Chat JS SDK라고 합니다.

참고: 경우에 따라 JavaScript와 TypeScript의 코드 예제가 동일하므로 서로 합쳐져 있습니다.

이 자습서의 첫 번째 부분은 여러 섹션으로 나뉩니다.

1. the section called “로컬 인증/권한 부여 서버 설정”
2. the section called “Chatterbox 프로젝트 생성”
3. the section called “채팅 룸에 연결”

4. the section called “토큰 공급자 구축”
5. the section called “연결 업데이트 관찰”
6. the section called “전송 버튼 구성 요소 생성”
7. the section called “메시지 입력 생성”
8. the section called “다음 단계”

전체 SDK 설명서를 보려면 우선 Amazon IVS 챗 클라이언트 메시징 SDK(Amazon IVS 챗 사용 설명서에서 참조) 및 Chat Client Messaging: SDK for JavaScript Reference(GitHub)를 참조하세요.

## 필수 조건

- 사용자는 JavaScript/TypeScript 및 React 라이브러리에 익숙하여야 합니다. React에 익숙하지 않다면 React 시작하기에서 기본 사항을 검토해 보시기 바랍니다.
- IVS 챗 시작하기를 읽고 이해합니다.
- 기존 IAM 정책에 정의된 CreateChatToken 및 CreateRoom 기능을 사용하여 AWS IAM 사용자를 생성합니다. (IVS 챗 시작하기를 참조하세요.)
- 이 사용자의 비밀/액세스 키가 AWS 보안 인증 파일에 저장되어 있는지 확인합니다. 지침은 AWS CLI 사용 설명서(특히 구성 및 보안 인증 파일 설정)를 참조합니다.
- 채팅룸을 생성하고 ARN을 저장합니다. IVS 챗 시작하기를 참조하세요. (ARN을 저장하지 않은 경우 나중에 콘솔이나 Chat API를 사용하여 조회할 수 있습니다.)
- NPM 또는 Yarn 패키지 관리자를 사용하여 Node.js 14+ 환경을 설치합니다.

## 로컬 인증/권한 부여 서버 설정

백엔드 애플리케이션은 채팅룸을 생성하고 Chat JS SDK가 채팅룸의 클라이언트를 인증하고 권한을 부여하는 데 필요한 채팅 토큰을 생성하는 일을 맡습니다. AWS 키는 모바일 앱에 안전하게 저장할 수 없으므로 자체 백엔드를 사용해야 합니다. 실력 좋은 공격자는 이러한 키를 추출하여 AWS 계정에 액세스할 수 있습니다.

Amazon IVS 채팅 시작하기에서 채팅 토큰 생성을 참조하세요. 순서도에서 볼 수 있듯이 서버 측 애플리케이션은 채팅 토큰 생성을 담당합니다. 즉, 앱은 서버 측 애플리케이션에서 채팅 토큰을 요청하여 채팅 토큰을 생성하는 자체 수단을 제공해야 합니다.

이 섹션에서는 백엔드에서 토큰 공급자를 생성하는 기본 사항에 대해 알아봅니다. AWS 환경을 사용하여 채팅 토큰 생성을 관리하는 로컬 서버를 만들기 위해 Express 프레임워크를 사용합니다.

NPM을 사용하여 빈 npm 프로젝트를 생성합니다. 애플리케이션을 보관할 디렉터리를 생성하고 이를 작업 디렉터리로 만듭니다.

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

npm init을 사용하여 애플리케이션의 package.json 파일을 생성합니다.

```
$ npm init
```

이 명령은 애플리케이션의 이름 및 버전을 비롯한 여러 항목을 입력하라는 메시지를 표시합니다. 지금은 RETURN 키를 눌러 대부분의 기본값을 그대로 사용할 수 있습니다. 단, 다음은 예외입니다.

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

RETURN 키를 눌러 제안된 기본 파일 이름 index.js를 그대로 사용하거나 주 파일의 이름을 원하는 대로 입력합니다.

이제 필요한 종속 항목을 설치합니다.

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

aws-sdk에는 루트 디렉터리에 있는 .env라는 파일에서 자동으로 로드되는 구성 환경 변수가 필요합니다. 이를 구성하려면 .env라는 새 파일을 생성하고 누락된 구성 정보를 입력합니다.

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

이제 npm init 명령에서 위에 입력한 이름으로 루트 디렉터리에 진입점(entry-point) 파일을 생성합니다. 이 경우 index.js를 사용하고 필요한 모든 패키지를 가져옵니다.

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

이제 express의 새 인스턴스를 생성합니다.

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

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

그런 다음 토큰 공급자를 위한 첫 번째 엔드포인트 POST 메서드를 생성할 수 있습니다. 요청 본문에서 필수 파라미터(roomId, userId, capabilities 및 sessionDurationInMinutes)를 가져옵니다.

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

필수 필드의 유효성 검사를 추가합니다.

```
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;
  }
});
```

POST 메서드를 준비한 후 인증/권한 부여의 핵심 기능을 위해 aws-sdk와 createChatToken을 통합합니다.

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

파일 끝에 express 앱의 포트 리스너를 추가합니다.

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

이제 프로젝트의 루트에서 다음 명령으로 서버를 실행할 수 있습니다.

```
$ node index.js
```

팁: 이 서버는 https://localhost:3000에서 URL 요청을 수락합니다.

## Chatterbox 프로젝트 생성

먼저 chatterbox라는 React 프로젝트를 생성합니다. 다음 명령을 실행합니다.

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

Node 패키지 관리자 또는 Yarn 패키지 관리자를 통해 Chat Client Messaging JS SDK를 통합할 수 있습니다.

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

## 채팅 룸에 연결

여기서는 ChatRoom을 생성하고 비동기 메서드를 사용하여 연결합니다. ChatRoom 클래스는 Chat JS SDK에 대한 사용자 연결을 관리합니다. 채팅룸에 성공적으로 연결하려면 React 애플리케이션 내에서 ChatToken의 인스턴스를 제공해야 합니다.

기본 chatterbox 프로젝트에서 생성한 App 파일로 이동하여 두  태그 사이의 모든 내용을 삭제합니다. 미리 입력된 코드는 필요하지 않습니다. 이 시점에서 App은 거의 비어 있습니다.

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

import * as React from 'react';

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

새 ChatRoom 인스턴스를 생성하고 useState 후크를 사용하여 상태로 전달합니다. 이를 위해서는 regionOrUrl(채팅룸이 호스팅되는 AWS 리전) 및tokenProvider(이후 단계에서 생성되는 백엔드 인증/권한 부여 흐름에 사용됨)를 전달해야 합니다.

중요: Amazon IVS Chat 시작하기에서 방을 생성한 리전과 동일한 AWS 리전을 사용해야 합니다. API는 AWS 리전 서비스입니다. 지원되는 리전 및 Amazon IVS Chat HTTPS 서비스 엔드포인트 목록은 Amazon IVS Chat 리전 페이지를 참조하세요.

```
// 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>;
}
```

## 토큰 공급자 구축

다음 단계로 ChatRoom 생성자에 필요한 파라미터 없는tokenProvider 함수를 구축해야 합니다. 먼저, the section called “로컬 인증/권한 부여 서버 설정”에서 설정한 백엔드 애플리케이션에 POST 요청을 하는 fetchChatToken 함수를 생성하겠습니다. 채팅 토큰은 SDK가 성공적으로 채팅룸 연결을 설정하는 데 필요한 정보를 포함합니다. Chat API는 사용자의 ID, 채팅룸 내 기능 및 세션 기간을 검증하는 안전한 방법으로 이러한 토큰을 사용합니다.

프로젝트 탐색기에서 fetchChatToken이라는 새 TypeScript/JavaScript 파일을 생성합니다. backend 애플리케이션에 가져오기 요청을 구축하고 응답에서 ChatToken 객체를 반환합니다. 채팅 토큰을 생성하는 데 필요한 요청 본문 속성을 추가합니다. Amazon 리소스 이름(ARN)에 정의된 규칙을 사용합니다. 이러한 속성은 CreateChatToken 엔드포인트에 문서화되어 있습니다.

참고: 여기서 사용하는 URL은 백엔드 애플리케이션을 실행했을 때 로컬 서버에서 생성한 URL과 동일한 URL입니다.

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),
  };
}
```

## 연결 업데이트 관찰

채팅 앱을 생성하기 위해서는 채팅룸 연결 상태 변화에 대해 필수적으로 대응해야 합니다. 관련 이벤트 구독부터 시작해 보겠습니다.

```
// 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>;
```

```
}
```

다음으로 연결 상태를 읽을 수 있는 기능을 제공해야 합니다. useState 후크를 사용하여 App에서 로컬 상태를 생성하고 각 리스너 내에서 연결 상태를 설정합니다.

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>;
```

```
}
```

연결 상태를 구독한 후 연결 상태를 표시하고 useEffect 후크 내의 room.connect 메서드를 사용하여 채팅룸에 연결합니다.

```
// 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>
);

// ...
```

채팅룸 연결을 성공적으로 구현했습니다.

# 전송 버튼 구성 요소 생성

이 섹션에서는 연결 상태 별로 각기 다른 디자인의 전송(send) 버튼을 생성합니다. 전송 버튼을 사용하면 채팅룸에서 메시지를 쉽게 보낼 수 있습니다. 또한 연결이 끊겼거나 채팅 세션이 만료 등과 같이 메시지를 전송 가능 여부 및 시기를 시각적으로 표시하는 역할을 합니다.

먼저, Chatterbox 프로젝트의 `src` 디렉터리에 새 파일을 생성하고 이름을 `SendButton`로 지정합니다. 그런 다음 채팅 애플리케이션용 버튼을 표시할 구성 요소를 생성합니다. `SendButton`을 내보내고 `App`으로 가져옵니다. 비어 있는 `<div></div>` 사이에 `<SendButton />`을 추가합니다.

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>
);
```

다음으로 App에서 onMessageSend라는 함수를 정의하고 이를 SendButton onPress 속성에 전달합니다. isSendDisabled라는 다른 변수를 정의하여(방이 연결되어 있지 않을 때 메시지를 보내지 못하도록 함) SendButton disabled 속성에 전달합니다.

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

// ...

const onMessageSend = () => {};

const isSendDisabled = connectionState !== 'connected';

return (
  <div>
    <div>Connection State: {connectionState}</div>
```

```
    <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
  </div>
);

// ...
```

## 메시지 입력 생성

Chatterbox 메시지 표시줄은 채팅룸에 메시지를 보내기 위해 상호 작용하는 구성 요소입니다. 일반적으로 메시지 작성을 위한 텍스트 입력과 메시지를 보내기 위한 버튼을 포함합니다.

MessageInput 구성 요소를 생성하려면 먼저 src 디렉터리에 새 파일을 생성하고 이름을 MessageInput로 지정합니다. 그런 다음 채팅 애플리케이션용 입력을 표시할 통제된 입력 구성 요소를 생성합니다. MessageInput을 내보내고 App로 가져옵니다(<SendButton /> 위로).

기본값으로 빈 문자열이 있는 useState 후크를 사용하여 messageToSend라는 새 상태를 생성합니다. 앱 본문에서 messageToSend를 MessageInput의 value로 전달하고 onMessageChange 속성에 setMessageToSend를 전달합니다.

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>
);
```

## 다음 단계

이제 Chatterbox용 메시지 표시줄 구축을 완료했으므로 이 JavaScript 자습서의 2부인 메시지 및 이벤트로 이동하세요.

# Amazon IVS Chat Client Messaging SDK: JavaScript 자습서 2부: 메시지 및 이벤트

이 자습서의 두 번째(마지막) 부분은 여러 섹션으로 나뉩니다.

1. the section called “채팅 메시지 이벤트 구독”
2. the section called “받은 메시지 보기”
   a. the section called “메시지 구성 요소 생성”
   b. the section called “현재 사용자가 전송한 메시지 인식”
   c. the section called “메시지 목록 구성 요소 생성”
   d. the section called “채팅 메시지 목록 렌더링”
3. the section called “채팅룸에서 작업 수행”
   a. the section called “메시지 전송”
   b. the section called “메시지 삭제”
4. the section called “다음 단계”

참고: 경우에 따라 JavaScript와 TypeScript의 코드 예제가 동일하므로 서로 합쳐져 있습니다.

전체 SDK 설명서를 보려면 우선 Amazon IVS 챗 클라이언트 메시징 SDK(Amazon IVS 챗 사용 설명서에서 참조) 및 Chat Client Messaging: SDK for JavaScript Reference(GitHub)를 참조하세요.

## 전제 조건

이 자습서의 1부인 채팅룸을 완료해야 합니다.

## 채팅 메시지 이벤트 구독

ChatRoom 인스턴스는 채팅룸에서 이벤트가 발생할 때 이벤트를 사용하여 통신합니다. 채팅 환경을 구현하기 위해서는 우선 연결된 방에서 다른 사람이 메시지를 보내고 있을 때 이를 사용자에게 보여줄 수 있어야 합니다.

여기서 채팅 메시지 이벤트를 구독합니다. 다음 단계에서는 생성한 메시지 목록을 업데이트하는 방법에 대해 살펴보겠습니다. 해당 목록은 각 메시지/이벤트마다 업데이트됩니다.

App의 useEffect 후크 내에서 모든 메시지 이벤트를 구독합니다.

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

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

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

## 받은 메시지 보기

메시지를 받는 것은 채팅 환경의 핵심 부분입니다. Chat JS SDK를 사용하여 채팅룸에 연결된 다른 사용자로부터 이벤트를 쉽게 수신하도록 코드를 설정할 수 있습니다.

나중에 여기서 생성한 구성 요소를 활용하여 채팅룸에서 작업을 수행하는 방법을 보여 드리겠습니다.

App에서 messages라는 ChatMessage 배열 형식으로 messages라는 상태를 정의합니다.

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([]);

  //...
}
```

다음으로 message 리스너 함수에서 messages 배열에 message를 추가합니다.

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

// ...

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

// ...
```

아래에서는 받은 메시지를 표시하는 작업을 단계별로 설명합니다.

1. the section called “메시지 구성 요소 생성”
2. the section called “현재 사용자가 전송한 메시지 인식”
3. the section called “메시지 목록 구성 요소 생성”
4. the section called “채팅 메시지 목록 렌더링”

## 메시지 구성 요소 생성

Message 구성 요소는 채팅룸에서 받은 메시지의 내용의 렌더링(rendering)을 담당합니다. 이 섹션에서는 App에서 개별 채팅 메시지를 렌더링하기 위한 메시지 구성 요소를 생성합니다.

src 디렉터리에 새 파일을 생성하고 이름을 Message로 지정합니다. 이 구성 요소의 ChatMessage 형식을 전달하고 ChatMessage 속성에서 content 문자열을 전달하여 채팅룸 메시지 리스너에서 받은 메시지 텍스트를 표시합니다. 프로젝트 탐색기에서 Message로 이동합니다.

### 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>
  );
```

```
};
```

팁: 이 구성 요소를 사용하여 메시지 행에 렌더링하려는 다양한 속성(예: 아바타 URL, 사용자 이름, 메시지가 전송된 시각의 타임스탬프)을 저장할 수 있습니다.

## 현재 사용자가 전송한 메시지 인식

현재 사용자가 전송한 메시지를 인식하려면 코드를 수정하고 현재 사용자의 `userId`를 저장하기 위한 React 컨텍스트를 만듭니다.

`src` 디렉터리에 새 파일을 생성하고 이름을 `UserContext`로 지정합니다.

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>;
};
```

참고: 여기서는 useState 후크를 사용하여 userId 값을 저장했습니다. 향후에는 사용자 컨텍스트를 변경하거나 로그인 용도로 setUserId를 사용할 수 있습니다.

다음으로 이전에 생성한 컨텍스트를 사용하여 userId를 tokenProvider에 전달된 첫 번째 파라미터로 교체합니다.

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

  // ...
}
```

Message 구성 요소에서 이전에 생성한 UserContext를 사용하고, isMine 변수를 선언하고, 발신자의 userId와 컨텍스트의 userId를 일치시키고, 현재 사용자에게 다양한 스타일의 메시지를 적용하세요.

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>
  );
};
```

## 메시지 목록 구성 요소 생성

MessageList 구성 요소는 시간이 지남에 따라 채팅룸의 대화를 표시하는 것을 담당합니다. MessageList 파일은 모든 메시지를 담는 컨테이너입니다. Message는 MessageList에 있는 한 줄입니다.

src 디렉터리에 새 파일을 생성하고 이름을 MessageList로 지정합니다. ChatMessage 배열 형식의 messages로 Props을 정의합니다. 본문 내에서 messages 속성을 매핑하고 Message 구성 요소에 Props를 전달합니다.

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>
  );
};
```

## 채팅 메시지 목록 렌더링

이제 새 `MessageList`를 기본 App 구성 요소로 가져옵니다.

```
// 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>
);

// ...
```

이제 App이 채팅룸에서 받은 메시지를 렌더링하기 위한 퍼즐 조각이 모두 다 준비되었습니다. 생성한 구성 요소를 활용하여 채팅룸에서 작업을 수행하는 방법을 보려면 다음을 계속합니다.

## 채팅룸에서 작업 수행

채팅룸 내에서 메시지를 보내고 중재자 작업을 수행하는 것은 채팅룸과 상호 작용하는 주요 방법 중 일부입니다. 여기서는 다양한 ChatRequest 객체를 사용하여 Chatterbox에서 메시지 전송, 메시지 삭제, 다른 사용자 연결 끊기와 같은 일반적인 작업을 수행하는 방법을 알아봅니다.

채팅룸의 모든 작업은 공통적인 패턴을 따릅니다. 채팅룸에서 수행하는 모든 작업에는 해당하는 요청 객체가 있습니다. 각 요청에는 요청 확인 시 받는 해당 응답 객체가 있습니다.

채팅 토큰을 생성할 때 사용자에게 올바른 권한이 부여되면 사용자는 요청 객체를 통해 이에 해당하는 작업을 성공적으로 수행하여 채팅룸에서 어떤 요청을 수행할 수 있는지 확인할 수 있습니다.

아래에서는 메시지를 전송하고 메시지를 삭제하는 방법을 설명합니다.

### 메시지 전송

SendMessageRequest 클래스를 사용하여 채팅룸에서 메시지를 보낼 수 있습니다 이 단계에서는 메시지 입력 생성(이 자습서의 1부)에서 생성한 구성 요소를 통해 메시지 요청을 보내도록 App을 수정합니다.

우선, useState 후크를 사용하여 isSending라는 새 boolean 속성을 정의합니다. 이속성을 통해 isSendDisabled 상수를 사용하여 button HTML 요소의 비활성화 상태를 전환할 수 있습니다. SendButton의 이벤트 핸들러(event handler)에서 messageToSend의 값을 지우고 isSending을 true로 설정합니다.

이 버튼으로 API를 호출하므로 isSending boolean을 추가하면 요청이 완료될 때까지 SendButton에서 사용자 상호 작용을 비활성화함으로써 여러 API 호출이 동시에 발생하는 것을 방지할 수 있습니다.

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

// ...

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

// ...

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

// ...

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

// ...
```

새 SendMessageRequest 인스턴스를 생성하고 생성자에 메시지 콘텐츠를 전달하여 요청을 준비합니다. isSending과 messageToSend 상태를 설정한 후 sendMessage 메서드를 호출하여 채팅룸으로 요청을 보냅니다. 마지막으로 요청 확인 또는 거절 수신 시isSending 플래그를 지웁니다.

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);
  }
};

// ...
```

Chatterbox를 실행해 보세요. `MessageInput`으로 초안을 작성하고 `SendButton`을 탭하여 메시지를 보내 보세요. 이전에 생성한 `MessageList` 내에 전송한 메시지가 렌더링된 것을 볼 수 있을 것입니다.

## 메시지 삭제

채팅룸에서 메시지를 삭제하려면 적절한 기능이 있어야 합니다. 기능은 채팅룸에 인증할 때 사용하는 채팅 토큰을 초기화하는 동안에 부여됩니다. 이 섹션의 목적에 따라 로컬 인증/권한 부여 서버 설정(이 자습서의 1부)의 `ServerApp`에서 중재자 기능을 지정할 수 있습니다. 이 작업은 토큰 공급자 구축(1부)에서 생성한 `tokenProvider` 객체를 사용하여 앱에서 수행됩니다.

여기서는 메시지를 삭제하는 함수를 추가하여 Message를 수정합니다.

먼저 App.tsx를 열고 DELETE_MESSAGE 기능을 추가합니다.(capabilities는 tokenProvider 함수의 두 번째 파라미터입니다.)

참고: 이는 ServerApp이 IVS Chat API에 결과 채팅 토큰과 연결된 사용자는 채팅룸에서 메시지를 삭제할 수 있음을 알리는 방법입니다. 실제 상황에서는 서버 앱 인프라의 사용자 기능을 관리하기 위한 백엔드 로직이 더 복잡할 수 있습니다.

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']),
  }),
);

// ...
```

다음 단계에서는 삭제 버튼을 표시하도록 Message를 업데이트합니다.

Message를 열고 초기 값이 false인 useState 후크를 사용하여 isDeleting라는 새 boolean 상태를 정의합니다. 이 상태를 사용하여 isDeleting의 현재 상태에 따라 Button의 내용이 달라지도록 업데이트합니다. isDeleting이 true일 때 버튼을 비활성화하면 동시에 메시지 삭제 요청을 두 번 시도할 수 없습니다.

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>
  );
};
```

문자열을 파라미터 중 하나로 받아 Promise를 반환하는 onDelete라는 새 함수를 정의합니다. Button 작업 종료의 본문에서 setIsDeleting를 사용하여 onDelete를 호출하기 전후에 isDeleting boolean을 전환합니다. 문자열 파라미터의 경우 구성 요소 메시지 ID를 전달합니다.

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>
  );
```

```
};
```

다음으로 Message 구성 요소의 최신 변경 사항을 반영하도록 MessageList를 업데이트합니다.

MessageList를 열고 문자열을 파라미터 중 하나로 받아 Promise를 반환하는 onDelete라는 새 함수를 정의합니다. Message를 업데이트하고 Message의 속성을 통해 전달합니다. 새로운 종료의 문자열 파라미터는 삭제하려는 메시지의 ID이며, 이 ID는 Message으로부터 전달됩니다.

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

다음으로 MessageList의 최신 변경 사항을 반영하도록 App을 업데이트합니다.

App에서 onDeleteMessage라는 함수를 정의하고 이를 MessageList onDelete 속성에 전달합니다.

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>
);

// ...
```

관련 메시지 ID를 생성자 파라미터에 전달하는 DeleteMessageRequest의 새 인스턴스를 생성하여 요청을 준비하고 위에서 준비한 요청을 수락하는 deleteMessage를 호출합니다.

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);
};

// ...
```

다음으로 방금 삭제한 메시지가 생략된 새 메시지 목록을 반영하도록 messages 상태를 업데이트합니다.

useEffect 후크에서messageDelete 이벤트를 수신하고 message 파라미터와 일치하는 ID가 있는 메시지를 삭제하여 messages 상태 배열을 업데이트합니다.

참고: 현재 사용자나 방에 있는 다른 사용자가 메시지를 삭제하는 경우 messageDelete 이벤트가 발생할 수 있습니다. 이벤트 핸들러에서(deleteMessage 요청 옆 대신) 이벤트를 처리하면 메시지 삭제 처리를 통합할 수 있습니다.

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

이제 채팅 앱의 채팅룸에서 사용자를 삭제할 수 있습니다.

## 다음 단계

테스트의 일환으로 방에서 다른 사용자의 연결을 끊는 등 다양한 작업을 시도해보세요.

# Amazon IVS 챗 클라이언트 메시징 SDK: React Native 자습서 1부: 채팅룸

본 문서는 두 파트로 구성된 자습서 중 첫 번째 파트에 해당하는 자습서입니다. 이 자습서에서는 React Native를 통해 완전한 기능을 갖춘 앱을 구축하여 Amazon IVS Chat Client Messaging JavaScript SDK로 작업하기 위한 필수 사항을 설명하고 있습니다. 여기에서 지칭하는 앱은 Chatterbox라고 합니다.

이 자습서는 숙련된 개발자이지만 Amazon IVS Chat Messaging SDK를 처음 접하는 사용자를 위해 작성되었습니다. 사용자는 TypeScript 또는 JavaScript 프로그래밍 언어, React Native 라이브러리에 대한 친숙도가 있어야 합니다.

간략하게 나타내자면 Amazon IVS Chat Client Messaging JavaScript SDK를 Chat JS SDK라고 합니다.

참고: 경우에 따라 JavaScript와 TypeScript의 코드 예제가 동일하므로 서로 합쳐져 있습니다.

이 자습서의 첫 번째 부분은 여러 섹션으로 나뉩니다.

1. the section called “로컬 인증/권한 부여 서버 설정”
2. the section called “Chatterbox 프로젝트 생성”
3. the section called “채팅 룸에 연결”
4. the section called “토큰 공급자 구축”
5. the section called “연결 업데이트 관찰”
6. the section called “전송 버튼 구성 요소 생성”
7. the section called “메시지 입력 생성”
8. the section called “다음 단계”

## 필수 조건

- 사용자는 TypeScript 또는 JavaScript 및 React Native 라이브러리에 익숙하여야 합니다. React Native에 익숙하지 않다면 React Native 소개에서 기본 사항을 검토해 보시기 바랍니다.
- IVS 챗 시작하기를 읽고 이해합니다.
- 기존 IAM 정책에 정의된 CreateChatToken 및 CreateRoom 기능을 사용하여 AWS IAM 사용자를 생성합니다. (IVS 챗 시작하기를 참조하세요.)
- 이 사용자의 비밀/액세스 키가 AWS 보안 인증 파일에 저장되어 있는지 확인합니다. 지침은 AWS CLI 사용 설명서(특히 구성 및 보안 인증 파일 설정)를 참조합니다.
- 채팅룸을 생성하고 ARN을 저장합니다. IVS 챗 시작하기를 참조하세요. (ARN을 저장하지 않은 경우 나중에 콘솔이나 Chat API를 사용하여 조회할 수 있습니다.)
- NPM 또는 Yarn 패키지 관리자를 사용하여 Node.js 14+ 환경을 설치합니다.

# 로컬 인증/권한 부여 서버 설정

백엔드 애플리케이션은 채팅룸을 생성하고 Chat JS SDK가 채팅룸의 클라이언트를 인증하고 권한을 부여하는 데 필요한 채팅 토큰을 생성하는 일을 맡습니다. AWS 키는 모바일 앱에 안전하게 저장할 수 없으므로 자체 백엔드를 사용해야 합니다. 실력 좋은 공격자는 이러한 키를 추출하여 AWS 계정에 액세스할 수 있습니다.

Amazon IVS 채팅 시작하기에서 채팅 토큰 생성을 참조하세요. 순서도에서 볼 수 있듯이 서버 측 애플리케이션은 채팅 토큰 생성을 담당합니다. 즉, 앱은 서버 측 애플리케이션에서 채팅 토큰을 요청하여 채팅 토큰을 생성하는 자체 수단을 제공해야 합니다.

이 섹션에서는 백엔드에서 토큰 공급자를 생성하는 기본 사항에 대해 알아봅니다. AWS 환경을 사용하여 채팅 토큰 생성을 관리하는 로컬 서버를 만들기 위해 Express 프레임워크를 사용합니다.

NPM을 사용하여 빈 npm 프로젝트를 생성합니다. 애플리케이션을 보관할 디렉터리를 생성하고 이를 작업 디렉터리로 만듭니다.

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

`npm init`을 사용하여 애플리케이션의 `package.json` 파일을 생성합니다.

```
$ npm init
```

이 명령은 애플리케이션의 이름 및 버전을 비롯한 여러 항목을 입력하라는 메시지를 표시합니다. 지금은 RETURN 키를 눌러 대부분의 기본값을 그대로 사용할 수 있습니다. 단, 다음은 예외입니다.

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

RETURN 키를 눌러 제안된 기본 파일 이름 `index.js`를 그대로 사용하거나 주 파일의 이름을 원하는 대로 입력합니다.

이제 필요한 종속 항목을 설치합니다.

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

`aws-sdk`에는 루트 디렉터리에 있는 `.env`라는 파일에서 자동으로 로드되는 구성 환경 변수가 필요합니다. 이를 구성하려면 `.env`라는 새 파일을 생성하고 누락된 구성 정보를 입력합니다.

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

이제 npm init 명령에서 위에 입력한 이름으로 루트 디렉터리에 진입점(entry-point) 파일을 생성합니다. 이 경우 index.js를 사용하고 필요한 모든 패키지를 가져옵니다.

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

이제 express의 새 인스턴스를 생성합니다.

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

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

그런 다음 토큰 공급자를 위한 첫 번째 엔드포인트 POST 메서드를 생성할 수 있습니다. 요청 본문에서 필수 파라미터(roomId, userId, capabilities 및 sessionDurationInMinutes)를 가져옵니다.

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

필수 필드의 유효성 검사를 추가합니다.

```
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;
  }
});
```

POST 메서드를 준비한 후 인증/권한 부여의 핵심 기능을 위해 aws-sdk와 createChatToken을 통합합니다.

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

파일 끝에 express 앱의 포트 리스너를 추가합니다.

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

이제 프로젝트의 루트에서 다음 명령으로 서버를 실행할 수 있습니다.

```
$ node index.js
```

팁: 이 서버는 https://localhost:3000에서 URL 요청을 수락합니다.

## Chatterbox 프로젝트 생성

먼저 chatterbox라는 React Native 프로젝트를 생성합니다. 다음 명령을 실행합니다.

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

또는 TypeScript 템플릿을 사용하여 expo 프로젝트를 생성합니다.

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

Node 패키지 관리자 또는 Yarn 패키지 관리자를 통해 Chat Client Messaging JS SDK를 통합할 수 있습니다.

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

## 채팅 룸에 연결

여기서는 ChatRoom을 생성하고 비동기 메서드를 사용하여 연결합니다. ChatRoom 클래스는 Chat JS SDK에 대한 사용자 연결을 관리합니다. 채팅룸에 성공적으로 연결하려면 React 애플리케이션 내에서 ChatToken의 인스턴스를 제공해야 합니다.

기본 chatterbox 프로젝트에서 생성한 App 파일로 이동하여 함수 구성 요소가 반환하는 모든 내용을 삭제합니다. 미리 입력된 코드는 필요하지 않습니다. 이 시점에서 App은 거의 비어 있습니다.

TypeScript/JavaScript:

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

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

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

새 ChatRoom 인스턴스를 생성하고 useState 후크를 사용하여 상태로 전달합니다. 이를 위해서는 regionOrUrl(채팅룸이 호스팅되는 AWS 리전) 및tokenProvider(이후 단계에서 생성되는 백엔드 인증/권한 부여 흐름에 사용됨)를 전달해야 합니다.

중요: Amazon IVS Chat 시작하기에서 방을 생성한 리전과 동일한 AWS 리전을 사용해야 합니다. API는 AWS 리전 서비스입니다. 지원되는 리전 및 Amazon IVS Chat HTTPS 서비스 엔드포인트 목록은 Amazon IVS Chat 리전 페이지를 참조하세요.

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>;
}
```

## 토큰 공급자 구축

다음 단계로 ChatRoom 생성자에 필요한 파라미터 없는tokenProvider 함수를 구축해야 합니다. 먼저, the section called “로컬 인증/권한 부여 서버 설정”에서 설정한 백엔드 애플리케이션에 POST 요청을 하는 fetchChatToken 함수를 생성하겠습니다. 채팅 토큰은 SDK가 성공적으로 채팅룸 연결을 설정하는 데 필요한 정보를 포함합니다. Chat API는 사용자의 ID, 채팅룸 내 기능 및 세션 기간을 검증하는 안전한 방법으로 이러한 토큰을 사용합니다.

프로젝트 탐색기에서 fetchChatToken이라는 새 TypeScript/JavaScript 파일을 생성합니다. backend 애플리케이션에 가져오기 요청을 구축하고 응답에서 ChatToken 객체를 반환합니다. 채팅

토큰을 생성하는 데 필요한 요청 본문 속성을 추가합니다. Amazon 리소스 이름(ARN)에 정의된 규칙을 사용합니다. 이러한 속성은 CreateChatToken 엔드포인트에 문서화되어 있습니다.

참고: 여기서 사용하는 URL은 백엔드 애플리케이션을 실행했을 때 로컬 서버에서 생성한 URL과 동일한 URL입니다.

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),
  };
}
```

## 연결 업데이트 관찰

채팅 앱을 생성하기 위해서는 채팅룸 연결 상태 변화에 대해 필수적으로 대응해야 합니다. 관련 이벤트 구독부터 시작해 보겠습니다.

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>;
}
```

다음으로 연결 상태를 읽을 수 있는 기능을 제공해야 합니다. `useState` 후크를 사용하여 App에서 로컬 상태를 생성하고 각 리스너 내에서 연결 상태를 설정합니다.

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>;
}
```

연결 상태를 구독한 후 연결 상태를 표시하고 `useEffect` 후크 내의 `room.connect` 메서드를 사용하여 채팅룸에 연결합니다.

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,
  }
});

// ...
```

채팅룸 연결을 성공적으로 구현했습니다.

# 전송 버튼 구성 요소 생성

이 섹션에서는 연결 상태 별로 각기 다른 디자인의 전송(send) 버튼을 생성합니다. 전송 버튼을 사용하면 채팅룸에서 메시지를 쉽게 보낼 수 있습니다. 또한 연결이 끊겼거나 채팅 세션이 만료 등과 같이 메시지를 전송 가능 여부 및 시기를 시각적으로 표시하는 역할을 합니다.

먼저, Chatterbox 프로젝트의 `src` 디렉터리에 새 파일을 생성하고 이름을 `SendButton`로 지정합니다. 그런 다음 채팅 애플리케이션용 버튼을 표시할 구성 요소를 생성합니다. `SendButton`을 내보내고 `App`으로 가져옵니다. 비어 있는 `<View></View>` 사이에 `<SendButton />`을 추가합니다.

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>
);
```

다음으로 App에서 onMessageSend라는 함수를 정의하고 이를 SendButton onPress 속성에 전달합니다. isSendDisabled라는 다른 변수를 정의하여(방이 연결되어 있지 않을 때 메시지를 보내지 못하도록 함) SendButton disabled 속성에 전달합니다.

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>
);

// ...
```

## 메시지 입력 생성

Chatterbox 메시지 표시줄은 채팅룸에 메시지를 보내기 위해 상호 작용하는 구성 요소입니다. 일반적으로 메시지 작성을 위한 텍스트 입력과 메시지를 보내기 위한 버튼을 포함합니다.

MessageInput 구성 요소를 생성하려면 먼저 src 디렉터리에 새 파일을 생성하고 이름을 MessageInput로 지정합니다. 그런 다음 채팅 애플리케이션용 입력을 표시할 입력 구성 요소를 생성합니다. MessageInput을 내보내고 App로 가져옵니다(<SendButton /> 위로).

기본값으로 빈 문자열이 있는 useState 후크를 사용하여 messageToSend라는 새 상태를 생성합니다. 앱 본문에서 messageToSend를 MessageInput의 value로 전달하고 onMessageChange 속성에 setMessageToSend를 전달합니다.

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',
  }
```

```
});
```

## 다음 단계

이제 Chatterbox용 메시지 표시줄 구축을 완료했으므로 이 React Native 자습서의 2부인 메시지 및 이벤트로 이동하세요.

# Amazon IVS 챗 클라이언트 메시징 SDK: React Native 자습서 2부: 메시지 및 이벤트

이 자습서의 두 번째(마지막) 부분은 여러 섹션으로 나뉩니다.

1. the section called “채팅 메시지 이벤트 구독”
2. the section called “받은 메시지 보기”
   a. the section called “메시지 구성 요소 생성”
   b. the section called “현재 사용자가 전송한 메시지 인식”
   c. the section called “채팅 메시지 목록 렌더링”
3. the section called “채팅룸에서 작업 수행”
   a. the section called “메시지 전송”
   b. the section called “메시지 삭제”
4. the section called “다음 단계”

참고: 경우에 따라 JavaScript와 TypeScript의 코드 예제가 동일하므로 서로 합쳐져 있습니다.

## 전제 조건

이 자습서의 1부인 채팅룸을 완료해야 합니다.

## 채팅 메시지 이벤트 구독

`ChatRoom` 인스턴스는 채팅룸에서 이벤트가 발생할 때 이벤트를 사용하여 통신합니다. 채팅 환경을 구현하기 위해서는 우선 연결된 방에서 다른 사람이 메시지를 보내고 있을 때 이를 사용자에게 보여줄 수 있어야 합니다.

여기서 채팅 메시지 이벤트를 구독합니다. 다음 단계에서는 생성한 메시지 목록을 업데이트하는 방법에 대해 살펴보겠습니다. 해당 목록은 각 메시지/이벤트마다 업데이트됩니다.

App의 `useEffect` 후크 내에서 모든 메시지 이벤트를 구독합니다.

TypeScript/JavaScript:

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

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

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

## 받은 메시지 보기

메시지를 받는 것은 채팅 환경의 핵심 부분입니다. Chat JS SDK를 사용하여 채팅룸에 연결된 다른 사용자로부터 이벤트를 쉽게 수신하도록 코드를 설정할 수 있습니다.

나중에 여기서 생성한 구성 요소를 활용하여 채팅룸에서 작업을 수행하는 방법을 보여 드리겠습니다.

App에서 `messages`라는 `ChatMessage` 배열 형식으로 `messages`라는 상태를 정의합니다.

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([]);

  //...
}
```

다음으로 message 리스너 함수에서 messages 배열에 message를 추가합니다.

TypeScript/JavaScript:

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

// ...

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

// ...
```

아래에서는 받은 메시지를 표시하는 작업을 단계별로 설명합니다.

1. the section called “메시지 구성 요소 생성”
2. the section called “현재 사용자가 전송한 메시지 인식”
3. the section called “채팅 메시지 목록 렌더링”

## 메시지 구성 요소 생성

Message 구성 요소는 채팅룸에서 받은 메시지의 내용의 렌더링(rendering)을 담당합니다. 이 섹션에서는 App에서 개별 채팅 메시지를 렌더링하기 위한 메시지 구성 요소를 생성합니다.

src 디렉터리에 새 파일을 생성하고 이름을 Message로 지정합니다. 이 구성 요소의 ChatMessage 형식을 전달하고 ChatMessage 속성에서 content 문자열을 전달하여 채팅룸 메시지 리스너에서 받은 메시지 텍스트를 표시합니다. 프로젝트 탐색기에서 Message로 이동합니다.

### 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,
  },
});
```

팁: 이 구성 요소를 사용하여 메시지 행에 렌더링하려는 다양한 속성(예: 아바타 URL, 사용자 이름, 메시지가 전송된 시각의 타임스탬프)을 저장할 수 있습니다.

## 현재 사용자가 전송한 메시지 인식

현재 사용자가 전송한 메시지를 인식하려면 코드를 수정하고 현재 사용자의 `userId`를 저장하기 위한 React 컨텍스트를 만듭니다.

`src` 디렉터리에 새 파일을 생성하고 이름을 `UserContext`로 지정합니다.

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;
```

참고: 여기서는 useState 후크를 사용하여 userId 값을 저장했습니다. 향후에는 사용자 컨텍스트를 변경하거나 로그인 용도로 setUserId를 사용할 수 있습니다.

다음으로 이전에 생성한 컨텍스트를 사용하여 userId를 tokenProvider에 전달된 첫 번째 파라미터로 교체합니다. 아래에 지정된 대로 토큰 공급자에 SEND_MESSAGE 기능을 추가해야 합니다. 메시지를 보내는 데 필요합니다.

### 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']),
      }),
  );

  // ...
}
```

Message 구성 요소에서 이전에 생성한 UserContext를 사용하고, isMine 변수를 선언(declare)하고, 발신자의 userId와 컨텍스트의 userId를 일치시킨 후 현재 사용자에게 다양한 스타일의 메시지를 적용하세요.

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',
  },
});
```

## 채팅 메시지 목록 렌더링

이제 `FlatList` 및 `Message` 구성 요소를 사용하여 메시지를 나열합니다.

### 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>
);

// ...
```

이제 App이 채팅룸에서 받은 메시지를 렌더링하기 위한 퍼즐 조각이 모두 다 준비되었습니다. 생성한 구성 요소를 활용하여 채팅룸에서 작업을 수행하는 방법을 보려면 다음을 계속합니다.

## 채팅룸에서 작업 수행

메시지를 보내고 중재자 작업을 수행하는 것은 채팅룸과 상호 작용하는 주요 방법 중 일부입니다. 여기서는 다양한 채팅 요청 객체를 사용하여 Chatterbox에서 메시지 전송, 메시지 삭제, 다른 사용자 연결 끊기와 같은 일반적인 작업을 수행하는 방법을 알아봅니다.

채팅룸의 모든 작업은 공통적인 패턴을 따릅니다. 채팅룸에서 수행하는 모든 작업에는 해당하는 요청 객체가 있습니다. 각 요청에는 요청 확인 시 받는 해당 응답 객체가 있습니다.

채팅 토큰을 생성할 때 사용자에게 올바른 기능이 부여되면 해당 사용자는 요청 객체를 사용하여 해당 작업을 성공적으로 수행하여 채팅룸에서 어떤 요청을 수행할 수 있는지 확인할 수 있습니다.

아래에서는 메시지를 전송하고 메시지를 삭제하는 방법을 설명합니다.

## 메시지 전송

SendMessageRequest 클래스를 사용하여 채팅룸에서 메시지를 보낼 수 있습니다 이 단계에서는 메시지 입력 생성(이 자습서의 1부)에서 생성한 구성 요소를 통해 메시지 요청을 보내도록 App을 수정합니다.

우선, useState 후크를 사용하여 isSending라는 새 boolean 속성을 정의합니다. 이 새 속성을 사용하면 isSendDisabled 상수를 사용하여 button 요소의 비활성화 상태를 전환할 수 있습니다. SendButton의 이벤트 핸들러(event handler)에서 messageToSend의 값을 지우고 isSending을 true로 설정합니다.

이 버튼으로 API를 호출하므로 isSending boolean을 추가하면 요청이 완료될 때까지 SendButton에서 사용자 상호 작용을 비활성화함으로써 여러 API 호출이 동시에 발생하는 것을 방지할 수 있습니다.

참고: 메시지 전송은 위의 현재 사용자가 전송한 메시지 인식에서 설명한 것처럼 토큰 공급자에 SEND_MESSAGE 기능을 추가한 경우에만 작동합니다.

TypeScript/JavaScript:

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

// ...

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

// ...

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

// ...

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

// ...
```

새 SendMessageRequest 인스턴스를 생성하고 생성자에 메시지 콘텐츠를 전달하여 요청을 준비합니다. isSending과 messageToSend 상태를 설정한 후 sendMessage 메서드를 호출하여 채팅룸으로 요청을 보냅니다. 마지막으로 요청 확인 또는 거절 수신 시isSending 플래그를 지웁니다.

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);
  }
};

// ...
```

Chatterbox를 실행해 보세요. MessageBar으로 초안을 작성하고 SendButton을 탭하여 메시지를 보내 보세요. 이전에 생성한 MessageList 내에 전송한 메시지가 렌더링된 것을 볼 수 있을 것입니다.

## 메시지 삭제

채팅룸에서 메시지를 삭제하려면 적절한 기능이 있어야 합니다. 기능은 채팅룸에 인증할 때 사용하는 채팅 토큰을 초기화하는 동안에 부여됩니다. 이 섹션의 목적에 따라 로컬 인증/권한 부여 서버 설정(이 자습서의 1부)의 ServerApp에서 중재자 기능을 지정할 수 있습니다. 이 작업은 토큰 공급자 구축(1부)에서 생성한 tokenProvider 객체를 사용하여 앱에서 수행됩니다.

여기서는 메시지를 삭제하는 함수를 추가하여 Message를 수정합니다.

먼저 App.tsx를 열고 DELETE_MESSAGE 기능을 추가합니다.(capabilities는 tokenProvider 함수의 두 번째 파라미터입니다.)

참고: 이는 ServerApp이 IVS Chat API에 결과 채팅 토큰과 연결된 사용자는 채팅룸에서 메시지를 삭제할 수 있음을 알리는 방법입니다. 실제 상황에서는 서버 앱 인프라의 사용자 기능을 관리하기 위한 백엔드 로직이 더 복잡할 수 있습니다.

TypeScript/JavaScript:

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

// ...

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

// ...
```

다음 단계에서는 삭제 버튼을 표시하도록 Message를 업데이트합니다.

문자열을 파라미터 중 하나로 받아 Promise를 반환하는 onDelete라는 새 함수를 정의합니다. 문자열 파라미터의 경우 구성 요소 메시지 ID를 전달합니다.

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',
  },
});
```

다음으로 FlatList 구성 요소의 최신 변경 사항을 반영하도록 renderItem를 업데이트합니다.

App에서 handleDeleteMessage라는 함수를 정의하고 이를 MessageList onDelete 속성에 전달합니다.

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]);

// ...
```

관련 메시지 ID를 생성자 파라미터에 전달하는 DeleteMessageRequest의 새 인스턴스를 생성하여 요청을 준비하고 위에서 준비한 요청을 수락하는 deleteMessage를 호출합니다.

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);
};

// ...
```

다음으로 방금 삭제한 메시지가 생략된 새 메시지 목록을 반영하도록 messages 상태를 업데이트합니다.

useEffect 후크에서messageDelete 이벤트를 수신하고 message 파라미터와 일치하는 ID가 있는 메시지를 삭제하여 messages 상태 배열을 업데이트합니다.

참고: 현재 사용자나 방에 있는 다른 사용자가 메시지를 삭제하는 경우 messageDelete 이벤트가 발생할 수 있습니다. 이벤트 핸들러에서(deleteMessage 요청 옆 대신) 이벤트를 처리하면 메시지 삭제 처리를 통합할 수 있습니다.

TypeScript/JavaScript:

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

이제 채팅 앱의 채팅룸에서 사용자를 삭제할 수 있습니다.

## 다음 단계

테스트의 일환으로 방에서 다른 사용자의 연결을 끊는 등 다양한 작업을 시도해보세요.

# Amazon IVS 챗 클라이언트 메시징 SDK: React 및 React Native 모범 사례

이 문서에서는 React 및 React Native용 Amazon IVS 챗 메시징 SDK를 사용하는 가장 중요한 방법을 설명합니다. 이 정보를 활용하여 React 앱 내에서 일반적인 채팅 기능을 빌드하고 IVS 챗 메시징 SDK의 고급 부분을 더 심층적으로 분석하는 데 필요한 배경지식을 얻을 수 있습니다.

## 채팅룸 이니셜라이저 후크 생성

`ChatRoom` 클래스에는 연결 상태를 관리하고 수신된 메시지 및 삭제된 메시지와 같은 이벤트를 수신하기 위한 코어 채팅 메서드와 리스너가 포함되어 있습니다. 여기서는 채팅 인스턴스를 후크에 올바르게 저장하는 방법을 보여줍니다.

## 구현

### 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 };
};
```

참고: 구성 파라미터를 즉시 업데이트할 수 없으므로 `setState` 후크의 `dispatch` 메서드를 사용하지 않습니다. SDK는 인스턴스를 한 번 생성하며, 토큰 공급자를 업데이트할 수 없습니다.

중요: `ChatRoom` 이니셜라이저 후크를 한 번 사용하여 새 채팅룸 인스턴스를 초기화하세요.

### 예

TypeScript/JavaScript:

```
// ...

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

```
  });

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

  // ...
};

// ...
```

## 연결 상태 수신 대기

원하는 경우 채팅룸 후크에서 연결 상태 업데이트를 구독할 수 있습니다.

### 구현

#### 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 };
};
```

# 채팅룸 인스턴스 공급자

prop 드릴링을 방지하기 위해 다른 구성 요소에서 후크를 사용하려면 React `context`를 사용하여 채팅룸 공급자를 생성할 수 있습니다.

## 구현

### 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;
```

## 예

ChatRoomProvider를 생성한 이후에 useChatRoomContext에서 인스턴스를 사용할 수 있습니다.

중요: 채팅 화면과 중간에 있는 다른 구성 요소 사이에 context에 액세스해야 하는 경우에만 공급자를 루트 수준으로 설정하여 연결을 수신하는 동안 불필요한 재렌더링이 발생하지 않도록 하세요. 아니면 공급자를 채팅 화면에 최대한 가깝게 배치하세요.

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

```
// ...
```

## 메시지 리스너 생성

수신되는 모든 메시지를 최신 상태로 유지하려면 message 및 deleteMessage 이벤트를 구독해야 합니다. 다음은 구성 요소에 채팅 메시지를 제공하는 몇 가지 코드입니다.

중요: 채팅 메시지 리스너가 메시지 상태를 업데이트할 때 여러 번 다시 렌더링될 수 있으므로 성능을 위해 ChatMessageContext를 ChatRoomProvider와 구분합니다. ChatMessageProvider를 사용할 구성 요소에 ChatMessageContext를 적용해야 합니다.

### 구현

#### 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>;
};
```

## React의 예

중요: 메시지 컨테이너를 ChatMessagesProvider로 래핑해야 합니다. Message 행은 메시지 내용을 표시하는 예제 구성 요소입니다.

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 Native의 예

기본적으로 `ChatMessage`에는 `FlatList`에서 각 행에 대한 React 키로 자동으로 사용되는 `id`가 포함되어 있으므로, `keyExtractor`를 전달할 필요가 없습니다.

### 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} />;
};
```

# 앱 내 여러 채팅룸 인스턴스

앱에서 여러 개의 동시 채팅룸을 사용하는 경우 채팅마다 공급자를 생성하여 해당 채팅 공급자에서 사용하는 것이 좋습니다. 이 예에서는 도움말 봇 및 고객 도움말 채팅을 생성합니다. 둘 모두를 위한 공급자를 생성합니다.

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>;
};
```

## React의 예

이제 동일한 `ChatRoomProvider`를 사용하는 다른 채팅 공급자를 사용할 수 있습니다. 나중에 각 화면/보기에서 동일한 `useChatRoomContext`를 다시 사용할 수 있습니다.

TypeScript/JavaScript:

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

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

## React Native의 예

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 챗 보안

AWS는 클라우드 보안을 가장 중요하게 생각합니다. AWS 고객은 보안에 가장 보안에 민감한 조직의 요구 사항에 부합하도록 구축된 데이터 센터 및 네트워크 아키텍처의 혜택을 누릴 수 있습니다.

보안은 AWS과 사용자의 공동 책임입니다. 공동 책임 모델은 이 사항을 클라우드 내 보안 및 클라우드의 보안으로 설명합니다.

- 클라우드의 보안 — AWS는 AWS 클라우드에서 AWS 서비스를 실행하는 인프라를 보호합니다. AWS는 또한 안전하게 사용할 수 있는 서비스를 제공합니다. 서드 파티 감사원은 정기적으로 AWS 규정 준수 프로그램의 일환으로 보안 효과를 테스트하고 검증합니다.
- 클라우드 내 보안 - 사용자의 책임은 사용하는 AWS 서비스에 의해 결정됩니다. 또한 데이터의 민감도, 조직의 요건 및 관련 법률 및 규정을 비롯한 기타 요소에 대해서도 책임이 있습니다.

이 설명서는 Amazon IVS 챗을 사용할 때 공동 책임 모델을 적용하는 방법을 이해하는 데 도움이 됩니다. 다음 주제에서는 보안 및 규정 준수 목적에 맞게 Amazon IVS 챗을 구성하는 방법을 보여줍니다.

주제



## 데이터 보호

Amazon Interactive Video Service(IVS) 챗으로 전송되는 데이터의 경우 다음과 같은 데이터 보호 기능이 적용됩니다.

- Amazon IVS Chat 트래픽은 WSS를 사용하여 전송 중인 데이터를 안전하게 유지합니다.
- Amazon IVS Chat 토큰은 KMS 고객 관리형 키를 사용하여 암호화됩니다.

Amazon IVS 챗은 고객(최종 사용자) 데이터를 제공하도록 요구하지 않습니다. 채팅방, 입력 또는 입력 보안 그룹에는 고객(최종 사용자) 데이터를 제공해야 하는 필드가 없습니다.

이름 필드와 같은 자유 형식 필드에 고객(최종 사용자) 계정 번호와 같은 중요 식별 정보를 입력하지 마세요. 여기에는 Amazon IVS 콘솔이나 API, AWS CLI 또는 AWS SDK를 사용하여 작업하는 경우가 포함됩니다. Amazon IVS 챗에 입력하는 모든 데이터는 진단 로그에 포함될 수 있습니다.

스트림은 엔드포인트 사이에서 암호화되지 않습니다. 스트림은 IVS 네트워크에서 처리를 위해 내부적으로 암호화되지 않은 상태로 전송될 수 있습니다.

# ID 및 액세스 관리

AWS Identity and Access Management (IAM)는 계정 관리자가 AWS 리소스에 대한 액세스를 안전하게 제어할 수 있도록 지원하는 AWS 서비스입니다. IVS 저지연 스트리밍 사용 설명서의 Identity and Access Management를 참조하세요.

## 고객

IAM을 사용하는 방법은 Amazon IVS에서 수행하는 작업에 따라 달라집니다. IVS 저지연 스트리밍 사용 설명서의 고객을 참조하세요.

## Amazon IVS가 IAM과 작동하는 방식

Amazon IVS API 요청을 하기 전에 먼저 하나 이상의 IAM 자격 증명(사용자, 그룹 및 역할) 및 IAM 정책을 생성한 다음, 정책을 자격 증명에 연결해야 합니다. 권한이 전파되는 데에는 몇 분 정도 소요되며 그때까지 API 요청은 거부됩니다.

Amazon IVS가 IAM과 작동하는 방식에 대한 자세한 내용은 IAM 사용 설명서에서 IAM으로 작업하는 AWS 서비스를 참조하세요.

## ID

IAM 자격 증명을 생성하여 AWS 계정의 사용자 및 프로세스에 대한 인증을 제공할 수 있습니다. IAM 그룹은 하나의 단위로 관리할 수 있는 IAM 사용자 모음입니다. IAM 사용 설명서에서 자격 증명(사용자, 그룹, 및 역할)을 참조하세요.

## 정책

정책은 JSON 권한이며, 정책 문서는 요소로 구성됩니다. IVS 저지연 스트리밍 사용 설명서에서 정책을 참조하세요.

Amazon IVS 챗은 다음 세 가지 요소를 지원합니다.

- 작업 - Amazon IVS 챗에 대한 정책 작업은 작업 앞에 `ivschat` 접두사를 사용합니다. 예를 들어, Amazon IVS 챗 `CreateRoom` API 메서드로 Amazon IVS 챗 채팅방을 생성할 권한을 누군가에게 부여하려면 해당 개인에게 적용할 정책에 `ivschat:CreateRoom` 작업을 포함합니다. 정책 문에는 `Action` 또는 `NotAction` 요소가 포함되어야 합니다.
- 리소스 - Amazon IVS 챗 채팅방 리소스에는 다음 ARN 형식이 있습니다.

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

  예를 들어, 명령문에서 VgNkEJgOVX9N 채팅방을 지정하려면 다음 ARN을 사용합니다.

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

  리소스 생성과 같은 일부 Amazon IVS 챗 작업은 특정 리소스에서 수행할 수 없습니다. 이러한 경우 와일드카드(*)를 사용해야 합니다.

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

- 조건 - Amazon IVS 챗은 `aws:RequestTag`, `aws:TagKeys`, `aws:ResourceTag` 같은 몇 가지 글로벌 조건 키를 지원합니다.

정책에서 변수를 자리 표시자로 사용할 수 있습니다. 예를 들어, 사용자의 IAM 사용자 이름으로 태그가 지정된 경우에만 IAM 사용자에게 리소스에 액세스할 수 있는 권한을 부여할 수 있습니다. IAM 사용 설명서에서 변수 및 태그를 참조하세요.

Amazon IVS에서는 미리 구성된 권한 세트를 자격 증명에 부여하는 데 사용할 수 있는 AWS 관리형 정책을 제공합니다(읽기 전용 또는 전체 액세스). 아래에 표시된 자격 증명 기반 정책 대신에 관리형 정책을 사용하도록 선택할 수 있습니다. 자세한 내용은 Amazon IVS의 관리형 정책을 참조하세요.

## Amazon IVS 태그를 기반으로 권한 부여

Amazon IVS 챗 리소스에 태그를 연결하거나 Amazon IVS 챗에 대한 요청에서 태그를 전달할 수 있습니다. 태그를 기반으로 액세스를 제어하려면 `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` 또는 `aws:TagKeys` 조건 키를 사용하여 정책의 조건 요소에서 태그 정보를 제공합니다. Amazon IVS 챗 리소스 태그 지정에 대한 자세한 내용은 Amazon IVS 챗 API 참조에서 '태그 지정'을 참조하세요.

## 역할

IAM 사용 설명서에서 IAM 역할 및 임시 보안 인증 정보를 참조하세요.

IAM 역할은 특정 권한을 가지고 있는 AWS계정 내 엔터티입니다.

Amazon IVS는 임시 보안 인증 정보를 지원합니다. 임시 보안 인증을 사용하여 페더레이션으로 로그인하거나 IAM 역할을 수임하거나 교차 계정 역할을 수임할 수 있습니다. AWS Security Token Service API 작업(예: `AssumeRole` 또는 `GetFederationToken`)을 호출하여 임시 보안 인증 정보를 가져옵니다.

## 권한 있는 액세스 및 권한 없는 액세스

API 리소스는 권한 있는 액세스 권한을 지원합니다. 권한 없는 재생 액세스 권한은 프라이빗 채널을 통해 설정할 수 있습니다(Setting Up Private Channels 참조).

## 정책에 대한 모범 사례

자세한 내용은 IAM 사용 설명서에서 IAM 모범 사례를 참조하세요.

자격 증명 기반 정책은 매우 강력합니다. 이 정책은 계정에서 사용자가 Amazon IVS 리소스를 생성, 액세스 또는 삭제할 수 있는지 여부를 결정합니다. 이 작업으로 인해 AWS 계정에 비용이 발생할 수 있습니다. 다음 권장 사항을 따릅니다.

- 최소 권한 부여 - 사용자 지정 정책을 생성할 때 오직 작업을 수행하는 데 필요한 권한만을 부여합니다. 최소한의 권한을 조합하는 것으로 시작하여 추후 필요에 따라 권한을 추가로 부여합니다. 이는 초기에 권한을 많이 부여한 후 나중에 이를 줄이는 것보다 더 안전한 방법입니다. 특히, 관리 액세스 권한을 위해 `ivschat:*`를 예약한후 이를 애플리케이션에서 사용하지 마시기 바랍니다.
- 중요한 작업에 대해 Multi-Factor Authentication(MFA) 활성화 - 보안을 강화하기 위해 IAM 사용자가 중요한 리소스 또는 API 작업에 액세스할 때 MFA를 사용하도록 합니다.
- 보안 강화를 위해 정책 조건 사용 – 실제로 가능한 경우 자격 증명 기반 정책이 리소스에 대한 액세스를 허용하는 조건을 정의합니다. 예를 들어, 요청할 수 있는 IP 주소의 범위를 지정하도록 조건을 작성할 수 있습니다. 지정된 날짜 또는 시간 범위 내에서만 요청을 허용하거나, SSL 또는 MFA를 사용해야 하는 조건을 작성할 수도 있습니다.

# 자격 증명 기반 정책 예제

## Amazon IVS 콘솔 사용

Amazon IVS 콘솔에 액세스하려면 AWS 계정의 Amazon IVS 챗 리소스에 대한 세부 정보를 나열하고 볼 수 있는 최소 권한이 있어야 합니다. 최소 필수 권한보다 더 제한적인 자격 증명 기반 정책을 생성하면 콘솔이 해당 정책에 연결된 엔터티에 대해 의도대로 작동하지 않습니다. Amazon IVS 콘솔에 대한 액세스를 보장하려면 자격 증명에 다음 정책을 연결합니다(IAM 사용 설명서에서 IAM 권한 추가 및 제거 참조).

다음 정책 부분은 다음에 대한 액세스를 제공합니다.

- 모든 Amazon IVS 챗 API 엔드포인트
- 사용자의 Amazon IVS 챗 Service quotas
- Amazon IVS Chat 중재를 위해 lambda 나열 및 선택한 lambda에 대한 권한 추가
- 챗 세션에 대한 지표를 가져오는 Amazon Cloudwatch

```
{
  "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 Chat에 대한 리소스 기반 정책

Amazon IVS Chat 서비스에 lambda 리소스를 호출하여 메시지를 검토할 수 있는 권한을 부여해야 합니다. 이를 위해 AWS Lambda에서 리소스 기반 정책 사용(AWS Lambda 개발자 안내서)의 지침을 준수하고 아래에 지정된 바와 같이 필드를 채워야 합니다.

lambda 리소스에 대한 액세스를 제어하기 위해 다음을 기반으로 조건을 사용할 수 있습니다.

- `SourceArn` - 샘플 정책은 와일드카드(*)를 사용하여 계정의 모든 방이 lambda를 호출하도록 허용합니다. 필요에 따라 계정에 방을 지정하여 해당 방만 lambda를 호출할 수 있도록 할 수 있습니다.
- `SourceAccount` - 아래 샘플 정책에서 AWS 계정 ID는 `123456789012`입니다.

```
{
   "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/*"
            }
```

```
            }
        }
    ]
}
```

## 문제 해결

Amazon IVS 챗 및 IAM에서 작업할 때 발생할 수 있는 일반적인 문제를 진단하고 수정하는 방법에 대한 자세한 내용은 IVS 저지연 스트리밍 사용 설명서의 문제 해결을 참조하세요.

# Amazon IVS에 대한 관리형 정책

AWS 관리형 정책은 AWS에서 생성 및 관리하는 독립적 정책입니다. IVS 저지연 스트리밍 사용 설명서에서 Amazon IVS에 대한 관리형 정책을 참조하세요.

# Amazon IVS에 대해 서비스 연결 역할 사용

Amazon IVS는 AWS IAM 서비스 연결 역할을 사용합니다. IVS 저지연 스트리밍 사용 설명서에서 Amazon IVS에 대해 서비스 연결 역할 사용을 참조하세요.

# 로깅 및 모니터링

성능 또는 작업을 로깅하려면 Amazon CloudTrail 을 사용합니다. IVS 저지연 스트리밍 사용 설명서에서 AWS CloudTrail을 사용하여 Amazon IVS API 호출 로깅을 참조하세요.

# 인시던트 대응

인시던트를 감지하거나 경고하기 위해 Amazon EventBridge 이벤트를 통해 스트림의 상태를 모니터링할 수 있습니다. Amazon IVS 지연 시간이 짧은 스트리밍과 함께 Amazon EventBridge 사용과 Amazon IVS 실시간 스트리밍과 함께 Amazon EventBridge 사용을 참조하세요.

Amazon IVS의 전반적인 상태(리전별)에 대한 정보를 확인하려면 AWS Health Dashboard를 사용하세요.

# 복원력

IVS API는 AWS 글로벌 인프라를 사용하며, AWS 리전 및 가용 영역을 중심으로 구축됩니다. IVS 저지연 스트리밍 사용 설명서에서 복원력을 참조하세요.

# 인프라 보안

관리형 서비스인 Amazon IVS는 AWS 글로벌 네트워크 보안 절차에 의해 보호받습니다. 이 주제는 보안, 자격 증명 및 규정 준수를 위한 모범 사례에 설명되어 있습니다.

## API 호출

AWS에서 게시한 API 호출을 사용하여 네트워크를 통해 Amazon IVS에 액세스합니다. IVS 저지연 스트리밍 사용 설명서의 인프라 보안에서 API 호출을 참조하세요.

## Amazon IVS Chat

Amazon IVS Chat 메시지 수집 및 전송은 엣지에 대한 암호화된 WSS 연결을 통해 이루어집니다. Amazon IVS Messaging API는 암호화된 HTTPS 연결을 사용합니다. 비디오 스트리밍 및 재생과 마찬가지로 TLS 버전 1.2 이상이 필요하며 메시징 데이터는 처리를 위해 내부적으로 암호화되지 않은 상태로 전송될 수 있습니다.

# Service Quotas(챗)

다음은 Amazon Interactive Video Service(IVS) 챗 엔드포인트, 리소스 및 기타 작업에 대한 서비스 할당량 및 한도입니다. 서비스 할당량(한도)은 AWS 계정의 최대 서비스 리소스 또는 작업 수입니다. 즉, 테이블에 별도로 명시되지 않는 한, 이러한 한도는 AWS 계정별로 다르게 적용됩니다. 자세한 내용은 AWS Service Quotas도 참조하세요.

AWS 서비스에 프로그래밍 방식으로 연결하려면 엔드포인트를 사용합니다. AWS 서비스 엔드포인트도 참조하세요.

모든 할당량은 리전별로 적용됩니다.

## 서비스 할당량 증가

조정 가능한 할당량과 관련하여 AWS console을 통해 비율 증가를 요청할 수 있습니다. 콘솔을 사용하여 서비스 할당량에 대한 정보도 확인합니다.

API 호출 비율 할당량은 조절할 수 없습니다.

## API 호출 비율 할당량

| 엔드포인트 타입 | 엔드포인트 | 기본값 |
|---|---|---|
| 메시지 전송 | `DeleteMessage` | 100TPS |
| 메시지 전송 | `DisconnectUser` | 100TPS |
| 메시지 전송 | `SendEvent` | 100TPS |
| 채팅 토큰 | `CreateChatToken` | 200TPS |
| 로깅 구성 | `CreateLoggingConfiguration` | 3TPS |
| 로깅 구성 | `DeleteLoggingConfiguration` | 3TPS |
| 로깅 구성 | `GetLoggingConfiguration` | 3TPS |
| 로깅 구성 | `ListLoggingConfigurations` | 3TPS |

| 엔드포인트 타입 | 엔드포인트 | 기본값 |
|---|---|---|
| 로깅 구성 | `UpdateLoggingConfiguration` | 3TPS |
| 방 | `CreateRoom` | 5TPS |
| 방 | `DeleteRoom` | 5TPS |
| 방 | `GetRoom` | 5TPS |
| 방 | `ListRooms` | 5TPS |
| 방 | `UpdateRoom` | 5TPS |
| 태그 | `ListTagsForResource` | 10TPS |
| 태그 | `TagResource` | 10TPS |
| 태그 | `UntagResource` | 10TPS |

# 기타 할당량

| 리소스 또는 기능 | 기본값 | 조정 가능 | 설명 |
|---|---|---|---|
| 동시 채팅 연결 | 50,000 | 예 | AWS 리전의 모든 방에서 계정당 최대 동시 채팅 연결 수입니다. |
| 로깅 구성 | 10 | 예 | 현재 AWS 리전에서 계정당 생성할 수 있는 최대 로깅 구성 수입니다. |
| 메시지 검토 핸들러 제한 시간 | 200 | 아니요 | 현재 AWS 리전의 모든 메시지 검토 핸들러에 대한 제한 시간(밀리초)입니다. 이 값을 초과하면 메시지 검토 핸들러에 대해 구성한 `fallbackResult` |

| 리소스 또는 기능 | 기본값 | 조정 가능 | 설명 |
| --- | --- | --- | --- |
| | | | 필드의 값에 따라 메시지가 허용되거나 거부됩니다. |
| 모든 방의 DeleteMessage 요청 비율 | 100 | 예 | 모든 방에서 1초 동안 만들 수 있는 최대 DeleteMessage 요청 수입니다. Amazon IVS 챗 API 또는 Amazon IVS 챗 메시징 API(WebSocket)에서 요청이 올 수 있습니다. |
| 모든 방의 DisconnectUser 요청 비율 | 100 | 예 | 모든 방에서 1초 동안 만들 수 있는 최대 DisconnectUser 요청 수입니다. Amazon IVS 챗 API 또는 Amazon IVS 챗 메시징 API(WebSocket)에서 요청이 올 수 있습니다. |
| 연결당 메시징 요청 비율 | 10 | 아니요 | 채팅 연결이 수행할 수 있는 초당 최대 메시징 요청 수입니다. |
| 모든 방의 SendMessage 요청 비율 | 1000 | 예 | 모든 방에서 1초 동안 만들 수 있는 최대 SendMessage 요청 수입니다. Amazon IVS 챗 메시징 API(WebSocket)에서 이러한 요청이 옵니다. |
| 방당 SendMessage 요청 비율 | 100 | 아니요(그러나 API를 통해 구성 가능) | 방 중 하나에 대해 1초 동안 만들 수 있는 최대 SendMessage 요청 수입니다. CreateRoom 및 UpdateRoom의 `maximumMessageRatePerSecond` 필드로 구성할 수 있습니다. Amazon IVS 챗 메시징 API(WebSocket)에서 이러한 요청이 옵니다. |

| 리소스 또는 기능 | 기본값 | 조정 가능 | 설명 |
| --- | --- | --- | --- |
| 방 | 50,000 | 예 | AWS 리전별 계정당 최대 채팅 룸 수입니다. |

# CloudWatch 사용량 지표와 Service Quotas 통합

CloudWatch를 사용하여 CloudWatch 사용량 지표(CloudWatch usage metrics)를 통해 서비스 할당량을 사전에 관리할 수 있습니다. 이러한 지표를 사용하여 CloudWatch 그래프 및 대시보드에서 현재 서비스 사용량을 시각화할 수 있습니다. Amazon IVS 챗 사용량 지표는 Amazon IVS 챗 서비스 할당량에 해당합니다.

CloudWatch 지표 수학 함수를 사용하여 해당 리소스에 대한 서비스 할당량을 그래프에 표시할 수 있습니다. 사용량이 서비스 할당량에 가까워지면 경고하는 경보를 구성할 수도 있습니다.

사용량 지표에 액세스하려면 다음을 수행합니다.

1. Service Quotas 콘솔 열기(https://console.aws.amazon.com/servicequotas/)
2. 탐색 창에서 AWS 서비스를 선택합니다.
3. AWS 서비스 목록에서 Amazon Interactive Video Service 챗을 선택합니다.
4. [서비스 할당량(Service quotas)] 목록에서 원하는 서비스 할당량을 선택합니다. 서비스 할당량/지표에 대한 정보를 포함한 새 페이지가 열립니다.

또는 CloudWatch 콘솔을 통해 이러한 지표를 확인할 수 있습니다. AWS 네임스페이스를 선택하고 사용량을 선택합니다. 서비스 목록에서 IVS 챗을 선택합니다. (Amazon IVS 챗 모니터링을 참조하세요.)

AWS/사용량 네임스페이스에서 Amazon IVS 챗은 다음 지표를 제공합니다.

| 지표 이름 | 설명 |
| --- | --- |
| `ResourceCount` | 계정에서 실행 중인 지정된 리소스의 수입니다. 리소스는 지표와 연결된 차원에 의해 정의됩니다.<br><br>유효한 통계: 최댓값(1분 동안 사용되는 최대 리소스 수). |

다음 차원은 사용량 지표를 세분화하는 데 사용됩니다.

| 차원 | 설명 |
| --- | --- |
| 서비스 | 리소스가 포함된 AWS 서비스의 이름 유효한 값: `IVS Chat`. |
| Class | 추적 중인 리소스의 클래스입니다. 유효한 값: `None`. |
| 유형 | 추적 중인 리소스의 유형. 유효한 값: `Resource`. |
| Resource | AWS 리소스의 이름입니다. 유효한 값: `ConcurrentChatConnections` .<br><br>ConcurrentChatConnections 사용량 지표는 Amazon IVS 챗 모니터링에 설명된 대로 AWS/IVSChat 네임스페이스(차원 없음)에 있는 지표의 사본입니다. |

## 사용량 지표에 대한 CloudWatch 경보 생성

Amazon IVS 챗 사용량 지표를 기반으로 CloudWatch 경보 생성:

1. Service Quotas 콘솔에서 위에서 설명한 대로 원하는 서비스 할당량을 선택합니다. 현재는 ConcurrentChatConnections에 대해서만 경보를 생성할 수 있습니다.
2. [Amazon CloudWatch 경보(Amazon CloudWatch alarms)] 섹션에서 [생성(Create)]을 선택합니다.
3. [경보 임계값(Alarm threshold)] 드롭다운 목록에서 경보 값으로 설정할 적용된 할당량 값의 백분율을 선택합니다.
4. [경보 이름(Alarm name)]에 경보 이름을 입력합니다.
5. [생성(Create)]을 선택합니다.

# 문제 해결 FAQ

이 문서에서는 Amazon Interactive Video Service(IVS) 챗에 대한 모범 사례와 문제 해결 팁을 설명합니다. IVS 챗과 관련된 동작은 대체로 IVS 비디오 관련 동작과 다릅니다. 자세한 내용은 Amazon IVS 챗 시작하기를 참조하세요.

주제:



## 채팅룸이 삭제될 때 IVS 챗 연결이 왜 끊겼나요?

채팅룸 리소스를 삭제해도 해당 채팅룸이 활발하게 사용되고 있는 경우 해당 채팅룸에 연결된 채팅 클라이언트의 연결이 자동으로 끊어지지 않습니다. 채팅 애플리케이션이 채팅 토큰을 새로 고치는 경우 연결이 끊어집니다. 또는 모든 사용자의 연결을 수동으로 끊어 채팅룸에서 모든 사용자를 제거해야 합니다.

# 용어집

AWS 용어집도 참조하세요. 아래 표에서 LL은 IVS 짧은 지연 시간 스트리밍을 나타내고, RT는 IVS 실시간 스트리밍을 나타냅니다.

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| AAC | 고급 오디오 코딩입니다. AAC는 손실이 발생하는 디지털 오디오 압축을 위한 오디오 코딩 표준입니다. MP3의 후속 형식으로 설계된 AAC는 일반적으로 동일한 비트레이트에서 MP3보다 음질이 우수합니다. AAC는 ISO와 IEC에서 MPEG-2 및 MPEG-4 사양의 일부로 표준화됩니다. | ✓ | ✓ | |
| 적응형 비트레이트 스트리밍 | 적응형 비트레이트(ABR) 스트리밍에서는 IVS 플레이어가 연결 품질 저하 시 더 낮은 비트레이트로 전환되고, 연결 품질 향상 시 더 높은 비트레이트로 전환될 수 있습니다. | ✓ | | |
| 적응형 스트리밍 | 동시 방송을 사용한 계층화된 인코딩을 참조하세요. | | ✓ | |
| 관리 사용자 | AWS 계정에서 사용할 수 있는 리소스 및 서비스에 대한 관리 액세스 권한이 있는 AWS 사용자입니다. AWS 설정 사용 설명서의 용어를 참조하세요. | ✓ | ✓ | ✓ |
| ARN | AWS 리소스의 고유한 식별자인 Amazon 리소스 이름입니다. 구체적인 ARN 형식은 리소스에 따라 다릅니다. IVS 리소스에서 사용하는 ARN 형식은 서비스 승인 참조에서 확인하세요. | ✓ | ✓ | ✓ |
| 가로 세로 비율 | 프레임 너비와 프레임 높이의 비율을 설명합니다. 예를 들면 16:9는 Full HD 또는 1080p 해상도에 해당하는 종횡비입니다. | ✓ | ✓ | |
| 오디오 모드 | 다양한 유형의 모바일 디바이스 사용자 및 해당 사용자가 사용하는 장비에 최적화된 사전 설정 또는 사용자 지정 오디오 구성입니다. IVS 브로드캐스트 | | ✓ | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| | SDK: 모바일 오디오 모드(실시간 스트리밍)를 참조하세요. | | | |
| AVC, H.264, MPEG-4 Part 10 | H.264 또는 MPEG-4 Part 10이라고도 하는 고급 비디오 코딩은 손실이 발생하는 디지털 비디오 압축을 위한 비디오 압축 표준입니다. | ✓ | ✓ | |
| 배경 교체 | 라이브 스트림 생성자가 배경을 변경할 수 있는 카메라 필터의 일종입니다. IVS 브로드캐스트 SDK: 타사 카메라 필터(실시간 스트리밍)의 배경 교체를 참조하세요. | | ✓ | |
| 비트 전송률 | 전송되거나 수신되는 초당 비트 수에 대한 스트리밍 지표입니다. | ✓ | ✓ | |
| 브로드캐스트, 브로드캐스터 | 스트림, 스트리머에 대한 다른 용어입니다. | ✓ | | |
| 버퍼링 | 재생 디바이스에서 재생해야 하는 콘텐츠를 다운로드할 수 없을 때 발생하는 상태입니다. 버퍼링은 여러 가지 방식으로 나타날 수 있습니다. 즉, 콘텐츠가 무작위로 중지 및 시작되거나(끊김이라고도 함) 콘텐츠가 오랫동안 중지되거나(멈춤이라고도 함) IVS 플레이어에서 재생이 일시 중지될 수 있습니다. | ✓ | ✓ | |
| 바이트 범위 재생 목록 | 표준 HLS 재생 목록보다 세분화된 재생 목록입니다. 표준 HLS 재생 목록은 10초짜리 미디어 파일로 구성됩니다. 바이트 범위 재생 목록의 경우 세그먼트 지속 시간은 스트림에 구성된 키프레임 간격과 동일합니다.<br><br>바이트 범위 재생 목록은 S3 버킷에 자동 레코딩된 브로드캐스트에만 사용할 수 있습니다. HLS 재생 목록과 함께 생성됩니다. Amazon S3에 자동 레코딩(저지연 스트리밍)의 바이트 범위 재생 목록을 참조하세요. | ✓ | | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| CBR | 고정 비트레이트는 브로드캐스트 중에 발생하는 상황과 관계없이 전체 비디오 재생에서 일관된 비트레이트를 유지하는 인코더의 속도 제어 방법입니다. 원하는 비트레이트를 달성하기 위해 작업의 빈 곳을 채울 수 있으며, 대상 비트레이트와 일치하도록 인코딩 품질을 조정하여 피크를 양자화할 수 있습니다. VBR 대신에 CBR을 사용하는 것이 좋습니다. | ✓ | ✓ | |
| CDN | 스트리밍 비디오와 같은 콘텐츠를 사용자가 있는 곳으로 가까이 가져와서 전송을 최적화하는 지리적으로 분산된 솔루션인 콘텐츠 전송 네트워크 또는 콘텐츠 배포 네트워크입니다. | ✓ | | |
| Channel | 수집 서버, 스트림 키, 재생 URL 및 레코딩 옵션을 포함하여 스트리밍의 구성을 저장하는 IVS 리소스입니다. 스트리머는 채널과 연결된 스트림 키를 사용하여 브로드캐스트를 시작합니다. 브로드캐스트 중 생성된 모든 지표 및 이벤트는 채널 리소스와 연결됩니다. | ✓ | | |
| 채널 유형 | 채널에 허용되는 해상도와 프레임 속도를 결정합니다. IVS Low-Latency Streaming API Reference의 채널 유형을 참조하세요. | ✓ | | |
| 챗 로깅 | 로깅 구성을 채팅룸과 연결하여 활성화할 수 있는 고급 옵션입니다. | | | ✓ |
| 채팅룸 | 메시지 검토 핸들러 및 챗 로깅과 같은 선택적 특성을 포함하여 채팅 세션의 구성을 저장하는 IVS 리소스입니다. IVS 챗 시작하기의 Step 2: Create a Chat Room을 참조하세요. | | | ✓ |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 클라이언트 측 구성 | 호스트 디바이스를 사용하여 스테이지 참가자의 오디오 및 비디오 스트림을 혼합한 다음 복합 스트림으로 IVS 채널에 보냅니다. 그러면 클라이언트 리소스 사용률이 높아지고 시청자에게 영향을 미치는 스테이지 또는 호스트 문제가 발생할 위험이 커지는 대신에 구성의 모양을 더 잘 제어할 수 있습니다.<br><br>서버 측 구성도 참조하세요. | ✓ | ✓ | |
| CloudFront | Amazon에서 제공하는 CDN 서비스입니다. | ✓ | | |
| CloudTrail | AWS와 외부 소스의 이벤트 및 계정 활동을 수집, 모니터링, 분석 및 유지하는 AWS 서비스입니다. AWS CloudTrail을 사용하여 API 호출 로깅을 참조하세요. | ✓ | ✓ | ✓ |
| CloudWatch | 애플리케이션을 모니터링하고, 성능 변화에 대응하고, 리소스 사용을 최적화하고, 운영 상태에 대한 인사이트를 제공하는 AWS 서비스입니다. CloudWatch를 사용하여 IVS 지표를 모니터링할 수 있습니다. IVS 실시간 스트리밍 모니터링과 IVS 저지연 스트리밍 모니터링을 참조하세요. | ✓ | ✓ | ✓ |
| 구성 | 여러 소스의 오디오 및 비디오 스트림을 단일 스트림으로 결합하는 프로세스입니다. | ✓ | ✓ | |
| 구성 파이프라인 | 여러 스트림을 결합하고 결과 스트림을 인코딩하는 데 필요한 일련의 프로세스 단계입니다. | ✓ | ✓ | |
| 압축 | 원래 표현보다 적은 비트를 사용하는 정보 인코딩입니다. 모든 특정 압축은 무손실이거나 손실 허용입니다. 무손실 압축에서는 통계적 중복성을 식별하고 제거하여 비트를 줄입니다. 무손실 압축에서는 손실되는 정보가 없습니다. 손실 허용 압축에서는 불필요하거나 덜 중요한 정보를 제거하여 비트를 줄입니다. | ✓ | ✓ | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 컨트롤 플레인 | 채널, 스테이지 또는 채팅룸과 같은 IVS 리소스에 대한 정보를 저장하고 이러한 리소스를 생성 및 관리하는 인터페이스를 제공합니다. 리전에 따라 다릅니다(AWS 리전 기준). | ✓ | ✓ | ✓ |
| CORS | 한 도메인에서 로드된 클라이언트 웹 애플리케이션이 다른 도메인의 S3 버킷과 같은 리소스와 상호작용하도록 허용하는 AWS 특성인 교차 오리진 리소스 공유(CORS)입니다. 헤더, HTTP 메서드 및 원본 도메인을 기반으로 액세스를 구성할 수 있습니다. Amazon Simple Storage Service 사용 설명서의 교차 오리진 리소스 공유(CORS) 사용 - Amazon Simple Storage Service를 참조하세요. | ✓ | | |
| 사용자 지정 이미지 소스 | 사전 설정된 카메라로 제한되지 않고 애플리케이션에서 자체 이미지 입력을 제공하도록 IVS 브로드캐스트 SDK에서 제공하는 인터페이스입니다. | ✓ | ✓ | |
| 데이터 영역 | 수집에서 송신까지 데이터를 전달하는 인프라입니다. 컨트롤 플레인에서 관리되는 구성을 기반으로 작동하며 AWS 리전에만 국한되지 않습니다. | ✓ | ✓ | ✓ |
| 인코더, 인코딩 | 비디오 및 오디오 콘텐츠를 스트리밍에 적합한 디지털 형식으로 변환하는 프로세스입니다. 인코딩은 하드웨어 또는 소프트웨어 기반일 수 있습니다. | ✓ | ✓ | |
| Event | IVS에서 AmazonEventBridge 모니터링 서비스에 게시하는 자동 알림입니다. 이벤트는 스테이지 또는 구성 파이프라인과 같은 스트리밍 리소스의 상태 변경을 나타냅니다. IVS 지연 시간이 짧은 스트리밍과 함께 Amazon EventBridge 사용과 Amazon IVS 실시간 스트리밍과 함께 Amazon EventBridge 사용을 참조하세요. | ✓ | ✓ | ✓ |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| FFmpeg | 비디오 및 오디오 파일과 스트림을 처리하는 라이브러리 및 프로그램 모음으로 구성된 무료 오픈 소스 소프트웨어 프로젝트입니다. FFmpeg에서는 오디오와 비디오를 녹음, 변환 및 스트리밍하는 교차 플랫폼 솔루션이 제공됩니다. | ✓ | | |
| 조각화된 스트림 | 채널의 레코딩 구성에 지정된 간격 내에 브로드캐스트의 연결이 해제되었다가 다시 연결될 때 생성됩니다. 여러 개의 결과 스트림이 단일 브로드캐스트로 간주되며 레코딩된 단일 스트림으로 병합됩니다. Amazon S3에 자동 레코딩(저지연 스트리밍)의 조각화된 스트림 병합을 참조하세요. | ✓ | | |
| 프레임 속도 | 전송되거나 수신되는 초당 비디오 프레임 수에 대한 스트리밍 지표입니다. | ✓ | ✓ | |
| HLS | IVS 스트림을 시청자에게 전송하는 데 사용되는 HTTP 기반 가변 비트레이트 스트리밍 통신 프로토콜인 HLS(HTTP Live Streaming)입니다. | ✓ | | |
| HLS 재생 목록 | 스트림을 구성하는 미디어 세그먼트 목록입니다. 표준 HLS 재생 목록은 10초짜리 미디어 파일로 구성됩니다. HLS에서는 더 세분화된 바이트 범위 재생 목록도 지원합니다. | ✓ | | |
| Host | 비디오 및/또는 오디오를 스테이지로 보내는 실시간 이벤트 참가자입니다. | | ✓ | |
| IAM | 사용자가 자격 증명을 안전하게 관리하고 IVS를 포함한 AWS 서비스 및 리소스에 액세스할 수 있게 하는 AWS 서비스인 Identity and Access Management입니다. | ✓ | ✓ | ✓ |
| 수집 | 호스트 또는 방송사로부터 비디오 스트림을 수신하여 처리하거나 시청자 또는 다른 참가자에게 전송하는 IVS 프로세스입니다. | ✓ | ✓ | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 수집 서버 | 시청자에게 전송하기 위해 비디오 스트림을 수신하고 트랜스코딩 시스템으로 전송하여 HLS로 트랜스먹싱하거나 트랜스코딩합니다.<br><br>수집 서버는 수집 프로토콜(RTMP, RTMPS)과 함께 채널용 스트림을 수신하는 특정 IVS 구성 요소입니다. IVS 지연 시간이 짧은 스트리밍 시작하기의 채널 생성에 대한 정보를 참조하세요. | | ✓ | |
| 인터레이스 비디오 | 후속 프레임의 홀수 또는 짝수 라인만 전송하고 표시하여 추가 대역폭을 소비하지 않고 두 배로 인식되는 프레임 속도를 생성합니다. 비디오 품질 문제 때문에 인터레이스 비디오는 사용하지 않는 것이 좋습니다. | ✓ | ✓ | |
| JSON | 사람이 읽을 수 있는 텍스트를 사용하여 속성-값 페어와 배열 데이터 유형 또는 기타 직렬화 가능한 값으로 구성된 데이터 객체를 전송하는 개방형 표준 파일 형식인 JavaScript Object Notation입니다. | ✓ | ✓ | ✓ |
| 키프레임, 델타 프레임, 키프레임 간격 | 키프레임(인트라 코딩된 프레임 또는 i-프레임이라고도 함)은 비디오 이미지의 전체 프레임입니다. 후속 프레임인 델타 프레임(예측된 프레임 또는 p-프레임이라고도 함)에는 변경된 정보만 포함됩니다. 인코더에 정의된 키프레임 간격에 따라 스트림 내에서 키프레임이 여러 번 나타납니다. | ✓ | ✓ | |
| Lambda | 서버 인프라를 프로비저닝하지 않고 코드(Lambda 함수라고 함)를 실행하는 AWS 서비스입니다. Lambda 함수는 이벤트 및 간접 호출 요청에 대한 응답으로 또는 일정에 따라 실행될 수 있습니다. 예를 들어 IVS 챗에서는 Lambda 함수를 사용하여 채팅룸에 대한 메시지 검토를 활성화합니다. | ✓ | ✓ | ✓ |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 지연 시간, 글래스 간 지연 시간 | 데이터 전송에서 발생하는 지연입니다. IVS에서는 지연 시간 범위를 다음과 같이 정의합니다.<br>• 짧은 지연 시간: 3초 미만<br>• 실시간 지연 시간: 300ms 미만<br><br>글래스 간 지연은 카메라가 라이브 스트림을 캡처할 때부터 스트림이 시청자 화면에 나타날 때까지의 지연을 말합니다. | ✓ | ✓ | |
| 동시 방송을 사용한 계층화된 인코딩 | 품질 수준이 각기 다른 여러 비디오 스트림의 동시 인코딩 및 게시를 활성화합니다. 실시간 스트리밍 최적화의 적응형 스트리밍: 동시 방송을 사용한 계층화된 인코딩을 참조하세요. | | ✓ | |
| 메시지 검토 핸들러 | 채팅룸으로 전송되기 전에 IVS 챗 고객이 사용자 채팅 메시지를 자동으로 검토/필터링할 수 있도록 합니다. Lambda 함수를 채팅룸과 연결하면 활성화됩니다. Chat Message Review Handler의 Creating a Lambda Function을 참조하세요. | | | ✓ |
| 믹서 | 여러 오디오 및 비디오 소스를 이용하여 단일 출력을 생성하는 IVS 모바일 브로드캐스트 SDK의 특성입니다. 카메라, 마이크, 화면 캡처, 애플리케이션에서 생성된 오디오 및 비디오 등의 소스를 나타내는 화면의 비디오 및 오디오 요소 관리를 지원합니다. 해당 출력은 IVS로 스트리밍될 수 있습니다. IVS 브로드캐스트 SDK: 믹서 가이드(지연 시간이 짧은 스트리밍)의 믹싱을 위한 브로드캐스트 세션 구성을 참조하세요. | ✓ | | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 다중 호스트 스트리밍 | 다중 호스트의 스트림을 단일 스트림으로 결합합니다. 이 작업은 클라이언트 측 또는 서버 측 구성을 사용하여 수행할 수 있습니다.<br><br>다중 호스트 스트리밍에서는 Q&A 스테이지로 시청자 초대, 호스트 간 경쟁, 영상 채팅, 대규모 인원 앞에서 서로 대화하는 호스트와 같은 시나리오를 지원합니다. | | ✓ | |
| 다변량 재생 목록 | 브로드캐스트에 사용할 수 있는 모든 변형 스트림의 인덱스입니다. | ✓ | | |
| OAC | 원본 액세스 제어(OAC)는 레코딩된 스트림과 같은 콘텐츠가 CloudFront CDN을 통해서만 제공될 수 있도록 S3 버킷에 대한 액세스를 제한합니다. | ✓ | | |
| OBS | Open Broadcaster Software(OBS)는 비디오 레코딩 및 라이브 스트리밍을 위한 무료 오픈 소스 소프트웨어입니다. OBS에서는 데스크톱 게시에 대한 대안을 IVS 브로드캐스트 SDK에 제공합니다. OBS에 익숙하며 더 섬세한 스트리머는 장면 전환, 오디오 믹싱, 오버레이 그래픽과 같은 고급 프로덕션 특성 때문에 OBS를 선호할 수도 있습니다. | ✓ | ✓ | |
| Participant | 호스트 또는 시청자로 스테이지에 연결된 실시간 사용자입니다. | | ✓ | |
| 참가자 토큰 | 실시간 이벤트 참가자가 스테이지에 조인할 때 인증합니다. 참가자 토큰은 스테이지에 대한 참가자의 비디오 전송 가능 여부도 제어합니다. | | ✓ | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 재생 토큰, 재생 키 페어 | 고객이 프라이빗 채널의 비디오 재생을 제한할 수 있는 인증 메커니즘입니다. 재생 토큰은 재생 키 페어에서 생성됩니다.<br><br>재생 키 페어는 재생을 위해 시청자 권한 부여 토큰에 서명하고 이를 검증하는 데 사용되는 퍼블릭-프라이빗 키 페어입니다. 프라이빗 채널 설정의 재생 키 생성 또는 가져오기를 참조하고 IVS Low-Latency API Reference의 Playback Key Pair endpoints를 참조하세요. | ✓ | | |
| 재생 URL | 시청자가 특정 채널의 재생을 시작하는 데 사용하는 주소를 식별합니다. 이 주소는 글로벌로 사용할 수 있습니다. IVS에서는 각 시청자에게 비디오를 전송하기 위해 IVS 글로벌 콘텐츠 전송 네트워크에서 최적의 위치를 자동으로 선택합니다. IVS 지연 시간이 짧은 스트리밍 시작하기의 채널 생성에 대한 정보를 참조하세요. | ✓ | | |
| 프라이빗 채널 | 고객이 재생 토큰 기반의 인증 메커니즘을 사용하여 스트림에 대한 액세스를 제한할 수 있습니다. 프라이빗 채널 설정의 프라이빗 채널 워크플로를 참조하세요. | ✓ | | |
| 프로그레시브 비디오 | 각 프레임의 모든 라인을 순서대로 전송하고 표시합니다. 브로드캐스트의 모든 스테이지에서 프로그레시브 비디오를 사용하는 것이 좋습니다. | ✓ | ✓ | |
| 할당량 | AWS 계정의 최대 IVS 서비스 리소스 또는 작업 수입니다. 즉, 별도로 명시되지 않는 한 이러한 한도는 AWS 계정별로 다르게 적용됩니다. 모든 할당량은 리전별로 적용됩니다. AWS 일반 참조 안내서의 Amazon Interactive Video Service 엔드포인트 및 할당량을 참조하세요. | ✓ | ✓ | ✓ |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 리전 | 특정 지리적 영역에 물리적으로 상주하는 AWS 서비스에 대한 액세스를 제공합니다. 리전에서는 내결함성, 안정성 및 복원성을 지원하고 지연 시간을 줄일 수도 있습니다. 리전을 통해 사용자는 가용 상태를 유지하며 리전 중단의 영향을 받지 않는 중복 리소스를 생성할 수 있습니다.<br><br>대부분의 AWS 서비스 요청은 특정한 지리적 리전과 관련이 있습니다. 한 리전에서 생성한 리소스는 AWS 서비스에서 제공하는 복제 특성을 명시적으로 사용하지 않는 한 다른 리전에 존재하지 않습니다. 예를 들어, Amazon S3에서는 교차 리전 복제를 지원합니다. IAM과 같은 일부 서비스에는 교차 리전 리소스가 없습니다. | ✓ | ✓ | ✓ |
| 해결 방법 | 단일 비디오 프레임의 픽셀 수를 설명합니다. 예를 들어 Full HD 또는 1080p는 1920x1080 픽셀로 프레임을 정의합니다. | ✓ | ✓ | |
| 루트 사용자 | AWS 계정 소유자입니다. 루트 사용자에게는 AWS 계정의 모든 AWS 서비스 및 리소스에 대한 완전한 액세스 권한이 있습니다. | ✓ | ✓ | ✓ |
| RTMP, RTMPS | 실시간 메시징 프로토콜은 네트워크를 통한 오디오, 비디오 및 데이터 전송 관련 업계 표준입니다. RTMPS는 전송 계층 보안(TLS/SSL) 연결을 통해 실행되는 안전한 RTMP 버전입니다. | ✓ | ✓ | |
| S3 버킷 | Amazon S3에 저장된 객체 모음입니다. 액세스 및 복제를 포함하여 많은 정책이 버킷 수준에서 정의되며 버킷의 모든 객체에 적용됩니다. 예를 들어 IVS 브로드캐스트는 S3 버킷에 여러 객체로 저장됩니다. | ✓ | | |
| SDK | IVS로 애플리케이션을 구축하는 개발자를 위한 라이브러리 모음인 소프트웨어 개발 키트입니다. | ✓ | ✓ | ✓ |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 셀카 분할 | 카메라 이미지를 입력으로 허용하고 이미지의 각 픽셀에 대한 신뢰도 점수를 제공하는 마스크를 반환하여 해당 이미지가 전경 또는 배경에 있는지를 나타내는 클라이언트별 솔루션을 사용하여 라이브 스트림의 배경을 교체할 수 있습니다. IVS 브로드캐스트 SDK: 타사 카메라 필터(실시간 스트리밍)의 배경 교체를 참조하세요. | | ✓ | |
| 의미 체계 버전 관리 | Major.Minor.Patch 형태의 버전 형식입니다. API에 영향을 주지 않는 버그 수정은 패치 버전을 증가시키고, 이전 버전과 호환되는 API 추가/변경은 마이너 버전을 증가시키며, 이전 버전과 호환되지 않는 API 변경은 메이저 버전을 증가시킵니다. | ✓ | ✓ | ✓ |
| 서버 측 구성 | IVS 서버를 사용하여 스테이지 참가자의 오디오와 비디오를 혼합한 다음에 이 혼합된 비디오를 IVS 채널로 보내 더 많은 대상에게 도달하거나 S3 버킷에 저장합니다. 서버 측 구성은 클라이언트 부하를 줄이고 브로드캐스트의 복원력을 개선하며 대역폭을 더 효율적으로 사용할 수 있도록 합니다.<br><br>클라이언트 측 구성도 참조하세요. | | ✓ | |
| Service quotas | 한 곳에서 많은 AWS 서비스에 대한 할당량을 관리하는 데 도움이 되는 AWS 서비스입니다. 할당량 값을 조회하는 것과 함께 Service Quotas 콘솔에서 할당량 증량을 요청할 수도 있습니다. | ✓ | ✓ | ✓ |
| 서비스 연결 역할 | AWS 서비스에 직접 연결된 고유한 유형의 IAM 역할입니다. 서비스 연결 역할은 IVS에서 자동으로 생성되며, 서비스에서 다른 AWS 서비스를 직접적으로 호출하는 데 필요한 모든 권한(예: S3 버킷 액세스 권한)을 포함합니다. IVS 보안의 IVS에 대해 서비스 연결 역할 사용을 참조하세요. | ✓ | | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 단계 | 실시간 이벤트 참가자가 실시간으로 비디오를 교환할 수 있는 가상 공간을 나타내는 IVS 리소스입니다. IVS 실시간 스트리밍 시작하기의 스테이지 생성을 참조하세요. | | ✓ | |
| 스테이지 세션 | 첫 번째 참가자가 스테이지에 조인하면 스테이지가 시작되고 마지막 참가자가 스테이지에 게시하는 것를 중지하면 몇 분 후에 종료됩니다. 수명이 긴 스테이지에는 수명 동안 여러 세션이 있을 수 있습니다. | | ✓ | |
| 스트림 | 소스에서 대상으로 지속적으로 보내지는 비디오 또는 오디오 콘텐츠를 나타내는 데이터입니다. | ✓ | ✓ | |
| 스트림 키 | 채널을 생성할 때 IVS에서 할당하는 식별자입니다. 채널에 스트리밍 권한을 부여하는 데 사용됩니다. 스트림 키는 누구나 이 키를 통해 채널로 스트리밍할 수 있으므로 암호처럼 취급합니다. IVS 지연 시간이 짧은 스트리밍 시작하기를 참조하세요. | ✓ | | |
| 스트림 결핍 | IVS로의 스트림 전송 지연 또는 중단입니다. 인코딩 디바이스에서 특정 기간에 전송할 것으로 광고한 예상 비트 양을 IVS에 수신되지 않으면 발생합니다. 스트림 결핍이 발생하면 스트림 결핍 이벤트가 됩니다.<br><br>시청자의 관점에서는 스트림 결핍이 지연, 버퍼링 또는 고정이 발생하는 비디오로 보일 수 있습니다. 스트림 결핍은 스트림 결핍을 초래한 특정 상황에 따라 짧을 수도 있고(5초 미만) 길 수도 있습니다(몇 분). 문제 해결 FAQ의 스트림 결핍이란 무엇인가요를 참조하세요. | ✓ | ✓ | |
| 스트리머 | IVS로 비디오 또는 오디오 스트림을 보내는 사람 또는 디바이스입니다. | ✓ | ✓ | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 구독자 | 호스트의 비디오 및/또는 오디오를 수신하는 실시간 이벤트 참가자입니다. IVS 실시간 스트리밍이란을 참조하세요. | | ✓ | |
| 태그 | AWS 리소스에 할당하는 메타데이터 레이블입니다. 태그를 사용하면 AWS 리소스를 식별하고 정리하는 데 도움이 됩니다. IVS 설명서 랜딩 페이지에서 실시간 스트리밍, 저지연 스트리밍 또는 채팅에 대한 IVS API 설명서의 '태그 지정'을 참조하세요. | ✓ | ✓ | ✓ |
| 타사 카메라 필터 | 애플리케이션에서 이미지를 브로드캐스트 SDK에 사용자 지정 이미지 소스로 제공하기 전에 처리할 수 있도록 IVS 브로드캐스트 SDK와 통합할 수 있는 소프트웨어 구성 요소입니다. 타사 카메라 필터에서는 카메라의 이미지를 처리하고 필터 효과를 적용하는 등의 작업을 수행할 수 있습니다. | ✓ | ✓ | |
| 썸네일 | 스트림에서 촬영한 축소된 크기의 이미지입니다. 기본적으로 60초마다 썸네일이 생성되지만, 더 짧은 간격을 구성할 수 있습니다. 썸네일 해상도는 채널 유형에 따라 다릅니다. Amazon S3에 자동 레코딩(저지연 스트리밍)의 레코딩 콘텐츠를 참조하세요. | ✓ | | |
| 시한 메타데이터 | 스트림 내 특정 타임스탬프에 연결된 메타데이터입니다. IVS API를 사용하여 프로그래밍 방식으로 추가하고 특정 프레임과 연결할 수 있습니다. 그러면 모든 시청자가 스트림을 기준으로 동일한 지점에서 메타데이터를 수신하게 됩니다.<br><br>시간 지정 메타데이터를 사용하여 스포츠 이벤트 중 팀 통계 업데이트와 같은 작업을 클라이언트에서 트리거할 수 있습니다. 비디오 스트림에 메타데이터를 포함하기를 참조하세요. | ✓ | | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 트랜스코딩 | 비디오와 오디오의 형식을 변환합니다. 수신 스트림은 다양한 재생 장치 및 네트워크 조건을 지원하기 위해 여러 비트 전송률과 해상도로 다른 형식으로 트랜스코딩될 수 있습니다. | ✓ | ✓ | |
| 트랜스먹싱 | 비디오 스트림을 다시 인코딩하지 않는 IVS에 수집한 스트림의 간단한 리패키징입니다. '트랜스먹스'는 원래 스트림의 일부 또는 모두를 유지하면서 오디오 및/또는 비디오 파일의 형식을 변경하는 프로세스인 트랜스코딩 멀티플렉싱의 약자입니다. 트랜스먹스는 파일 내용을 변경하지 않고 다른 컨테이너 형식으로 변환됩니다. 트랜스코딩과 구별됩니다. | ✓ | ✓ | |
| 변형 스트림 | 동일한 브로드캐스트를 여러 가지 품질 수준으로 인코딩한 세트입니다. 각 변형 스트림은 별도의 HLS 재생 목록으로 인코딩됩니다. 사용 가능한 변형 스트림의 인덱스를 다변량 재생 목록이라고 합니다.<br><br>IVS 플레이어에서는 IVS로부터 다변량 재생 목록을 수신한 후 재생하는 동안 네트워크 상태 변화에 따라 앞뒤로 원활하게 변경되도록 변형 스트림 중에서 선택할 수 있습니다. | ✓ | | |
| VBR | 필요한 세부 수준에 따라 재생 내내 변경되는 동적 비트레이트를 사용하는 인코더의 속도 제어 방법인 가변 비트레이트입니다. 비디오 품질 문제 때문에 VBR은 사용하지 않는 것이 좋습니다. 그 대신에 CBR을 사용하세요. | ✓ | ✓ | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| 보기 | 비디오를 적극적으로 다운로드하거나 재생하는 고유한 시청 세션입니다. 조회수는 동시 보기 할당량의 기준입니다.<br><br>보기 세션이 비디오 재생을 시작하면 보기가 시작됩니다. 보기 세션이 비디오 재생을 중지하면 보기가 종료됩니다. 재생은 시청자의 유일한 지표입니다. 오디오 레벨, 브라우저 탭 포커스 및 비디오 품질과 같은 참여 경험적 방법은 고려되지 않습니다. 조회수를 계산할 때 IVS에서는 개별 시청자의 적합성을 고려하거나, 현지화된 시청자(예: 단일 머신의 여러 비디오 플레이어)를 중복 제거하려고 시도하지 않습니다. Service Quotas(저지연 스트리밍)의 기타 할당량을 참조하세요. | ✓ | | |
| 뷰어 | IVS의 스트림을 수신하는 사람입니다. | ✓ | | |
| WebRTC | 웹 브라우저와 모바일 애플리케이션에 실시간 통신을 제공하는 오픈 소스 프로젝트인 웹 실시간 통신입니다. 플러그인 설치 또는 네이티브 앱 다운로드 필요성이 제거되도록 직접적인 P2P 통신을 허용하므로 웹 페이지 내부에서 오디오 및 비디오 통신이 작동할 수 있습니다.<br><br>WebRTC의 기반 기술은 개방형 웹 표준으로 구현되며, 모든 주요 브라우저의 기본 JavaScript API 또는 Android 및 iOS와 같은 네이티브 클라이언트의 라이브러리로 사용할 수 있습니다. | ✓ | ✓ | |

| 용어 | 설명 | LL | RT | 채팅 |
|---|---|---|---|---|
| WHIP | WebRTC-HTTP 수집 프로토콜은 WebRTC에 기반하여 스트리밍 서비스 및/또는 CDN으로 콘텐츠를 수집하도록 허용하는 HTTP 기반 프로토콜입니다. WHIP는 WebRTC 수집을 표준화하기 위해 작성된 IETF 초안입니다.<br><br>WHIP는 OBS와 같은 소프트웨어의 호환성을 지원하여 데스크톱 게시에 대한 (IVS 브로드캐스트 SDK의) 대안을 제공합니다. OBS에 익숙하며 더 섬세한 스트리머는 장면 전환, 오디오 믹싱, 오버레이 그래픽과 같은 고급 프로덕션 특성 때문에 OBS를 선호할 수도 있습니다.<br><br>WHIP는 IVS 브로드캐스트 SDK 사용이 불가능하거나 선호되지 않는 상황에서도 유용합니다. 예를 들어 하드웨어 인코더와 관련된 설정에서는 IVS 브로드캐스트 SDK가 적합하지 않을 수 있습니다. 하지만 인코더가 WHIP를 지원하는 경우에는 계속해서 인코더에서 IVS로 직접 게시할 수 있습니다.<br><br>OBS and WHIP Support를 참조하세요. | | ✓ | |
| WSS | 암호화된 TLS 연결을 통해 WebSocket을 설정하는 프로토콜인 WebSocket Secure입니다. IVS 챗 엔드포인트에 연결하는 데 사용됩니다. IVS 챗 시작하기의 Step 4: Send and Receive Your First Message를 참조하세요. | | | ✓ |

# 설명서 기록(챗)

## 챗 사용 설명서 변경 사항

| 변경 사항 | 설명 | 날짜 |
| --- | --- | --- |
| 챗 UG 분리 | 이 릴리스에는 주요 설명서 변경 사항이 포함되어 있습니다. IVS 저지연 스트리밍 사용 설명서의 챗 정보를 IVS 설명서 랜딩 페이지의 기존 IVS 챗 섹션에 있는 새로운 IVS 챗 사용 설명서로 이동했습니다.<br><br>기타 설명서 변경 사항은 문서 기록(지연 시간이 짧은 스트리밍)을 참조하세요. | 2023년 12월 28일 |
| IVS 용어집 | IVS 실시간, 저지연 및 채팅 용어를 다루는 용어집을 확장했습니다. | 2023년 12월 20일 |

## IVS 챗 API 참조 변경 사항

| API 변경 사항 | 설명 | 날짜 |
| --- | --- | --- |
| 챗 UG 분리 | 이제 IVS 챗 사용 설명서(이번 릴리스에서 작성됨)가 마련되었으므로 앞으로 기존 IVS 챗 API 참조 및 IVS 챗 메시징 API 참조에 대한 설명서 기록 내역이 여기에 작성됩니다. 이러한 챗 API 참조에 대한 이전 기록 내역은 Document History (Low-Latency Streaming)에서 확인할 수 있습니다. | 2023년 12월 28일 |

# 릴리스 노트(챗)

## 2023년 12월 28일

### Amazon IVS 챗 사용 설명서

Amazon Interactive Video Service(IVS) 챗은 라이브 비디오 스트림과 함께 사용할 수 있는 관리형 라이브 챗 기능입니다. 이번 릴리스에서는 IVS 저지연 스트리밍 사용 설명서의 챗 정보를 새로운 IVS 챗 사용 설명서로 이전했습니다. Amazon IVS 설명서 랜딩 페이지에서 설명서에 액세스할 수 있습니다.

## 2023년 1월 31일

### Amazon IVS 챗 클라이언트 메시징 SDK: Android 1.1.0

| 플랫폼 | 다운로드 및 변경 사항 |
| --- | --- |
| Android 챗 클라이언트 메시징 SDK 1.1.0 | 참조 문서: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/<br>• Kotlin 코루틴을 지원하기 위해 com.amazonaws.ivs.chat.messaging.coroutines 패키지에 새로운 IVS 챗 메시징 API를 추가했습니다. 새로운 Kotlin 코루틴 자습서도 참조하세요. 2부 중 1부는 채팅룸입니다. |

#### Chat Client Messaging SDK 크기: Android

| 아키텍처 | 압축된 크기 | 압축되지 않은 크기 |
| --- | --- | --- |
| 모든 아키텍처(바이트코드) | 89KB | 92KB |

# 2022년 11월 9일

## Amazon IVS 챗 Client Messaging SDK: JavaScript 1.0.2

| 플랫폼 | 다운로드 및 변경 사항 |
| --- | --- |
| JavaScript Chat Client Messaging SDK 1.0.2 | 참조 문서: https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/<br>• Firefox에 영향을 주던 문제를 수정했습니다. 클라이언트가 DisconnectUser 엔드포인트를 사용하여 채팅룸에서 연결을 끊었을 때 소켓 오류를 잘못 수신했습니다. |

# 2022년 9월 8일

## Amazon IVS 챗 Client Messaging SDK: Android 1.0.0 및 iOS 1.0.0

| 플랫폼 | 다운로드 및 변경 사항 |
| --- | --- |
| Android Chat Client Messaging SDK 1.0.0 | 참조 문서: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/ |
| iOS Chat Client Messaging SDK 1.0.0 | 참조 문서: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/ |

### Chat Client Messaging SDK 크기: Android

| 아키텍처 | 압축된 크기 | 압축되지 않은 크기 |
| --- | --- | --- |
| 모든 아키텍처(바이트코드) | 53KB | 58KB |

# Chat Client Messaging SDK 크기: iOS

| 아키텍처 | 압축된 크기 | 압축되지 않은 크기 |
| --- | --- | --- |
| ios-arm64_x86_64-simulator(비트코드) | 484KB | 2.4MB |
| ios-arm64_x86_64-simulator | 484KB | 2.4MB |
| ios-arm64(비트코드) | 1.1MB | 3.1MB |
| ios-arm64 | 233KB | 1.2MB |