



チャットのユーザーガイド

Amazon IVS



Amazon IVS: チャットのユーザーガイド

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon の商標およびトレードドレスは、Amazon 以外の製品およびサービスに使用することはできません。また、お客様に誤解を与える可能性がある形式で、または Amazon の信用を損なう形式で使用することもできません。Amazon が所有していないその他のすべての商標は、Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

Table of Contents

IVS チャットとは	1
IVS Chat の開始方法	2
ステップ 1: 初期設定を行う	3
ステップ 2: チャットルームを作成する	4
コンソールでの手順	5
CLI の手順	8
ステップ 3: チャットトークンを作成する	10
AWS SDK での手順	11
CLI の手順	12
ステップ 4: 最初のメッセージを送受信する	13
ステップ 5: Service-Quota 制限の確認 (オプション)	15
チャットのログ記録	16
ルームでチャットのログ記録を有効にする	16
メッセージの内容	16
[形式]	16
フィールド	17
Amazon S3 バケット	17
[形式]	17
フィールド	17
例	18
Amazon CloudWatch Logs	18
[形式]	18
フィールド	18
例	18
Amazon Kinesis Data Firehose	19
制約	19
Amazon CloudWatch によるエラーのモニタリング	19
チャットメッセージレビューハンドラ	20
Lambda 関数の作成	20
ワークフロー	20
リクエストの構文	20
リクエスト本文	21
レスポンスの構文	21
レスポンスフィールド	22

サンプルコード	23
ハンドラとルームとの関連付けと関連付け解除	24
Amazon CloudWatch によるエラーのモニタリング	24
モニタリング	25
CloudWatch メトリクスへのアクセス	25
CloudWatch コンソールでの手順	25
CLI の手順	26
CloudWatch メトリクス: IVS Chat	27
IVS Chat Client Messaging SDK	31
プラットフォームの要件	31
デスクトップブラウザ	31
モバイルブラウザ	31
ネイティブプラットフォーム	32
サポート	32
バージョンニング	32
Amazon IVS Chat API	33
Android ガイド	34
概要	34
SDK の使用	36
Android のチュートリアルパート 1: チャットルーム	40
前提条件	40
ローカル認証サーバーおよび認可サーバーのセットアップ	41
Chatterbox プロジェクトの作成	44
チャットルームに接続し、接続の更新を確認する	47
トークンプロバイダーの作成	52
次のステップ	56
Android のチュートリアルパート 2: メッセージとイベント	56
前提条件	57
メッセージ送信のために UI を作成する	57
ビューバインディングを適用する	64
チャットメッセージのリクエストを管理する	67
最後のステップ	72
Kotlin コルーチンのチュートリアルパート 1: チャットルーム	75
前提条件	76
ローカル認証サーバーおよび認可サーバーのセットアップ	77
Chatterbox プロジェクトの作成	80

チャットルームに接続し、接続の更新を確認する	83
トークンプロバイダーの作成	87
次のステップ	91
Kotlin コルーチンのチュートリアルパート 2: メッセージとイベント	91
前提条件	92
メッセージ送信のために UI を作成する	92
ビューインデントを適用する	99
チャットメッセージのリクエストを管理する	102
最後のステップ	107
iOS ガイド	110
概要	110
SDK の使用	112
iOS のチュートリアル	124
JavaScript ガイド	124
概要	125
SDK の使用	126
JavaScript のチュートリアルパート 1: チャットルーム	131
前提条件	132
ローカル認証サーバーおよび認可サーバーのセットアップ	132
Chatterbox プロジェクトの作成	136
チャットルームに接続する	136
トークンプロバイダーの作成	137
接続の更新を確認する	139
送信ボタンコンポーネントの作成	143
メッセージ入力の作成	145
次のステップ	147
JavaScript チュートリアルパート 2: メッセージとイベント	148
前提条件	148
チャットメッセージイベントへのサブスクライブ	148
受信済みメッセージの表示	149
チャットルームでアクションを実行する	157
次のステップ	167
React Native チュートリアルパート 1: Chat Rooms (チャットルーム)	168
前提条件	168
ローカル認証サーバーおよび認可サーバーのセットアップ	169
Chatterbox プロジェクトの作成	172

チャットルームに接続する	173
トークンプロバイダーの作成	174
接続の更新を確認する	176
送信ボタンコンポーネントの作成	179
メッセージ入力の作成	182
次のステップ	185
React Native チュートリアルパート 2: Messages and Events (メッセージとイベント)	185
前提条件	186
チャットメッセージイベントへのサブスクライブ	186
受信済みメッセージの表示	187
チャットルームでアクションを実行する	196
次のステップ	204
React と React Native のベストプラクティス	204
チャットルームイニシャライザーフックの作成	205
ChatRoom インスタンスプロバイダー	208
メッセージリスナーの作成	210
アプリ内の複数のチャットルームインスタンス	214
セキュリティ	219
チャット データ保護	220
ID とアクセス管理	220
対象者	220
Amazon IVS で IAM を使用する方法	220
ID	221
ポリシー	221
Amazon IVS タグに基づく承認	222
ロール	222
特権アクセスと非特権アクセス	222
ポリシーのベストプラクティス	222
アイデンティティベースのポリシーの例	223
Amazon IVS Chat のリソースベースのポリシー	224
トラブルシューティング	225
IVS Chat 用マネージドポリシー	225
IVS Chat のサービスにリンクされたロールの使用	225
ログ記録とモニタリング	226
インシデントへの対応	226
レジリエンス	226

インフラストラクチャセキュリティ	226
API 呼び出し	226
Amazon IVS Chat	226
Service Quotas	228
Service Quotas の引き上げ	228
API コールレートクォータ	228
その他のクォータ	229
Service Quotas と CloudWatch 使用量メトリクスの統合	231
使用量メトリクスの CloudWatch アラームを作成する	232
トラブルシューティングに関するよくある質問	234
ルームが削除されたときに IVS チャットの接続が切断されなかったのはなぜですか?	234
用語集	235
ドキュメント履歴	256
チャットのユーザーガイドの変更点	256
IVS チャット API リファレンスの変更点	256
リリースノート	258
2023 年 12 月 28 日	258
Amazon IVS Chat ユーザーガイド	258
2023 年 1 月 31 日	258
Amazon IVS Chat Client Messaging SDK: Android 1.1.0	258
2022 年 11 月 9 日	259
Amazon IVS Chat Client Messaging SDK: JavaScript 1.0.2	259
2022 年 9 月 8 日	259
Amazon IVS Chat Client Messaging SDK: Android 1.0.0 および iOS 1.0.0	259

Amazon IVS Chat とは何ですか？

Amazon IVS Chat は、ライブ動画ストリームと一緒に使用できるマネージドライブチャット機能です。ドキュメントは、[Amazon IVS ドキュメントのランディングページ](#)の「Amazon IVS Chat」セクションからアクセスできます。

- [チャットユーザーガイド](#) — このドキュメントおよびナビゲーションペインに一覧表示されている他のすべてのユーザーガイドページ。
- [チャット API リファレンス](#) — コントロールプレーン API (HTTPS)。
- [チャットメッセージング API リファレンス](#) — データプレーン API (WebSocket)。
- [チャットクライアント向け SDK リファレンス](#): Android、iOS、JavaScript

Amazon IVS Chat の開始方法

Amazon Interactive Video Service (IVS) チャットは、ライブ動画ストリームに付随するマネージド型のライブチャット機能です。(IVS Chat はビデオストリームなしでも使用できます。) チャットルームを作成し、ユーザー間でチャットセッションを有効にできます。

Amazon IVS Chat では、ライブ動画に付随するカスタマイズされたチャット体験を構築することに注力できます。インフラストラクチャの管理や、チャットワークフローのコンポーネントの開発および設定は不要です。Amazon IVS Chat は、スケーラブルかつ安全で、信頼性が高く、コスト効率に優れています。

Amazon IVS Chat は、ライブ動画ストリームの参加者間の開始と終了時のメッセージングを容易にするために最適です。

このドキュメントの残りの部分では、Amazon IVS Chat を使用して最初のチャットアプリケーションを構築する手順を説明します。

例: 以下のデモアプリが利用可能です (3 つのサンプルクライアントアプリとトークン作成用のバックエンドサーバーアプリ)。

- [Amazon IVS Chat ウェブ用デモ](#)
- [Amazon IVS Chat Android 用デモ](#)
- [Amazon IVS Chat iOS 用デモ](#)
- [Amazon IVS Chat デモバックエンド](#)

重要: 24 か月間新しい接続や更新がないチャットルームは自動的に削除されます。

トピック

- [ステップ 1: 初期設定を行う](#)
- [ステップ 2: チャットルームを作成する](#)
- [ステップ 3: チャットトークンを作成する](#)
- [ステップ 4: 最初のメッセージを送受信する](#)
- [ステップ 5: Service-Quota 制限の確認 \(オプション\)](#)

ステップ 1: 初期設定を行う

先に進む前に、次のことを行う必要があります。

1. AWS アカウント作成します。
2. ルートユーザーと管理ユーザーを設定します。
3. AWS IAM (Identity and Access Management) のアクセス許可を設定します。次のポリシーを使用してください。

前述の具体的なステップについては、「Amazon IVS ユーザーガイド」の「[Amazon IVS 低レイテンシーストリーミングの開始](#)」を参照してください。重要: 「ステップ 3: IAM アクセス許可の設定」では、次の IVS チャットポリシーを使用します。

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

ステップ 2: チャットルームを作成する

Amazon IVS Chat ルームには、それに関連付けられた設定情報 (メッセージの最大長など) があります。

このセクションの手順では、コンソールまたは AWS CLI を使用してチャットルームをセットアップし (メッセージを確認したり、メッセージをログ記録したりするためのオプション設定を含む)、チャットルームを作成する方法を示します。

コンソールでの手順

これらのステップは、初期ルームの設定から最終ルームの作成まで、いくつかのフェーズに分かれています。

オプションで、メッセージがレビューされるようにルームを設定できます。例えば、メッセージの内容やメタデータを更新したり、メッセージを拒否して送信されないようにしたり、元のメッセージを通過させたりすることができます。これについては、[ルームメッセージのレビュー設定 \(オプション\)](#)で説明しています。

オプションで、メッセージがログに記録されるようにルームを設定することもできます。例えば、チャットルームに送信するメッセージがある場合は、Amazon S3 バケット、Amazon CloudWatch、または Amazon Kinesis Data Firehose にそのメッセージをログとして記録できます。これについては、[メッセージのログ設定 \(オプション\)](#)で説明しています。


初期ルーム設定

1. [Amazon IVS Chat コンソール](#)を開きます。

([AWS マネジメントコンソール](#)から Amazon IVS コンソールにアクセスすることもできます。)

2. ナビゲーションバーから、[Select a Region (リージョンの選択)] ドロップダウンをクリックして、リージョンを選択します。新しいルームがこのリージョンに作成されます。
3. [Get started] (使用開始) ボックス (右上) で、[Amazon IVS Chat Room] (Amazon IVS Chat ルーム) を選択します。[Create room] (ルームの作成) ウィンドウが表示されます。

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*

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

4. [Setup] (セットアップ) を選択し、オプションで [Room name] (ルーム名) を指定します。ルーム名は一意ではありませんが、ルーム ARN (Amazon リソースネーム) 以外のルームを区別するのに役立ちます。
5. [Setup > Room configuration] (設定 > ルーム設定) で、[Default configuration] (デフォルト設定) を受け入れるか、[Custom configuration] (カスタム設定) を選択して、[Maximum message length] (最大メッセージ長) および/または [Maximum message rate] (最大メッセージレート) を設定することができます。
6. メッセージをレビューする場合は、以下の「[ルームメッセージのレビュー設定 \(オプション\)](#)」に進みます。それ以外の場合は、その手順をスキップして (つまり、[Message Review Handler > Disabled] (メッセージレビューハンドラ > 無効) を受け入れ)、直接 [\[Final Room Creation\]](#) (最終ルームの作成) に進みます。

ルームメッセージのレビュー設定 (オプション)

1. [Message Review Handler] (メッセージレビューハンドラ) で、[Handle with AWS Lambda] (AWS Lambda で処理する) を選択します。[Message Review Handler] (メッセージレビューハンドラ) セクションが展開され、追加のオプションが表示されます。
2. ハンドラが有効な応答を返さない場合、エラーが発生した場合、またはタイムアウト期間を超えた場合に、メッセージを許可または拒否するようにフォールバック結果を設定します。
3. 既存の [Lambda function] (Lambda 関数) を指定するか、[Create Lambda function] (Lambda 関数の作成) を使用して新しい関数を作成します。

Lambda 関数は、チャットルームと同じ AWS アカウントおよび同じ AWS リージョンに存在する必要があります。Amazon Chat SDK サービスに Lambda リソースを呼び出すアクセス許可を与える必要があります。リソースベースのポリシーは、選択した Lambda 関数に対して自動的に作成されます。アクセス許可の詳細については、「[Resource-Based Policy for Amazon IVS Chat](#)」を参照してください。

メッセージのログ設定 (オプション)

1. [Message logging] (メッセージログ) で、[Automatically log chat messages] (チャットメッセージを自動的に記録する) を選択します。[Message logging] (メッセージログ) セクションが展開され、追加のオプションが表示されます。既存のログ記録設定をこのルームに追加するか、[Create logging configuration] (ログ記録設定の作成) を選択して、新しいログ記録設定を作成できます。

2. 既存のログ記録設定を選択すると、ドロップダウンメニューが開き、作成済みのログ記録設定がすべて表示されます。リストから 1 つ選択すると、自動的にチャットメッセージがこの送信先に記録されます。
3. [Create logging configuration] (ログ記録設定の作成) を選択すると、新しくモーダルウィンドウが表示され、ログ記録設定の作成とカスタマイズができます。
 - a. オプションで [Logging configuration name] (ログ記録設定名) を指定します。ログ記録設定名は、ルーム名のように一意ではありませんが、ログ記録設定 ARN 以外のログ記録設定を区別するのに役立ちます。
 - b. [Destination] (送信先) で、[CloudWatch log group] (CloudWatch ロググループ)、[Kinesis firehose delivery stream] (Kinesis Firehose 配信ストリーム)、または [Amazon S3 bucket] (Amazon S3 バケット) を選択して、ログの送信先を選択します。
 - c. 送信先に応じて、既存の CloudWatch ロググループ、Kinesis Firehose 配信ストリーム、または Amazon S3 バケットを使用するか、これらを新規に作成するかを選択します。
 - d. レビュー後、[Create] (作成) をクリックして、一意の ARN で新しいログ記録設定を作成します。これにより新しいログ記録設定が、自動的にチャットルームにアタッチされます。

最終ルームの作成

1. レビュー後、[Create chat room] (チャットルームの作成) をクリックして、一意の ARN で新しいチャットルームを作成します。

CLI の手順

チャットルームを作成する

AWS CLI を使用してチャットルームを作成することは詳細オプションであり、最初に CLI をダウンロードしてマシンに設定する必要があります。詳細については、「[AWS Command Line Interface のユーザーガイド](#)」を参照してください。

1. チャット create-room コマンドを実行し、オプション名を渡します。

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

2. これにより、新しいチャットルームが返されます。

```
{
  "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 フィールドに注意してください。クライアントトークンを作成してチャットルームに接続するには、これが必要です。

ログ記録設定のセットアップ (オプション)

AWS CLI を使用してチャットルームを作成し、ログ記録設定をセットアップすることは詳細オプションであり、最初に CLI をダウンロードしてマシンに設定する必要があります。詳細については、「[AWS Command Line Interface のユーザーガイド](#)」を参照してください。

1. チャットの create-logging-configuration コマンドを実行し、オプションとしてログ設定の名前と、Amazon S3 バケットを指定する送信先設定の名前を渡します。この Amazon S3 バケットは、ログ記録設定の作成前に存在している必要があります。(Amazon S3 バケットの作成について詳しくは、[Amazon S3 ドキュメント](#)を参照してください。)

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

2. これで新しいログ記録設定が返ってきます。

```
{
  "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 フィールドに注意してください。チャットルームにログ記録設定をアタッチするには、このフィールドが必要です。

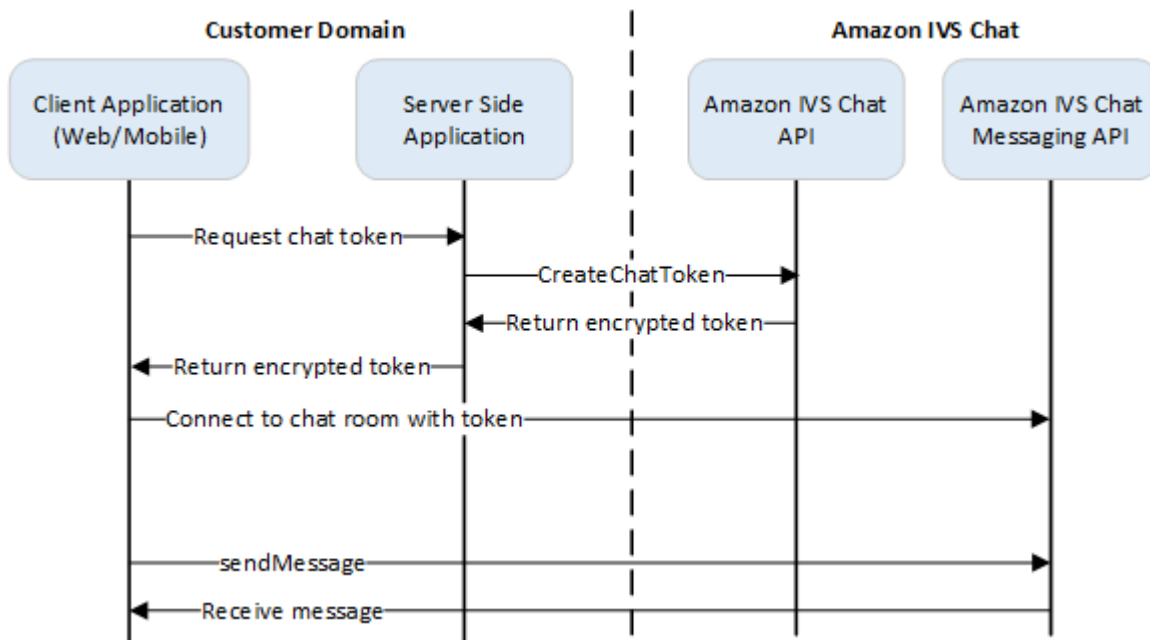
- a. 新しくチャットルームを作成する場合は、create-room コマンドを実行して、ログ記録設定 arn を渡します。

```
aws ivschat create-room --name test-room \
--logging-configuration-identifiers \
"arn:aws:ivschat:us-west-2:123456789012:logging-configuration/ABCdef34ghIJ"
```

- b. 既存のチャットルームを更新する場合は、update-room コマンドを実行して、ログ記録設定 arn を渡します。

```
aws ivschat update-room --identifier \
"arn:aws:ivschat:us-west-2:123456789012:room/g1H2I3j4k5L6" \
--logging-configuration-identifiers \
"arn:aws:ivschat:us-west-2:123456789012:logging-configuration/ABCdef34ghIJ"
```

ステップ 3: チャットトークンを作成する



チャット参加者がルームに接続してメッセージの送受信を開始するには、チャットトークンを作成する必要があります。チャットトークンは、チャットクライアントの認証と承認に使用されます。

上記のように、クライアントアプリケーションはサーバー側のアプリケーションにトークンを要求し、サーバー側のアプリケーションは AWS SDK または [SigV4](#) 署名付きリクエストを使用して `CreateChatToken` を呼び出します。AWS 認証情報が API の呼び出しに使用されるため、トークンはクライアント側のアプリケーションではなく、安全なサーバー側のアプリケーションで生成する必要があります。

トークン生成のデモンストレーションを行うバックエンドサーバーアプリケーションは、[Amazon IVS Chat デモバックエンド](#) で入手可能です。

セッション期間とは、確立されたセッションが自動的に終了されるまでのアクティブな状態を維持できる期間を指します。つまり、セッション期間は、新しいトークンを生成して新しい接続を確立する必要が生じる前に、クライアントがチャットルームに接続したままにできる期間です。オプションでは、トークンの作成中に、セッション期間を指定できます。

エンドユーザーの接続を確立するために、各トークンは一度だけ使用できます。接続が終了した場合、接続を再確立する前に、新しいトークンを作成する必要があります。トークン自体は、応答に含まれるトークンの有効期限のタイムスタンプまで有効です。

エンドユーザーがチャットルームに接続する場合、クライアントはサーバーアプリケーションにトークンを要求する必要があります。サーバーアプリケーションはトークンを作成し、それをクライアントに返します。トークンはオンデマンドでエンドユーザー向けに作成する必要があります。

チャット認証トークンを作成するには、以下の手順に従います。チャットトークンを作成する際は、リクエストフィールドを使用してチャットエンドユーザーとエンドユーザーのメッセージング機能に関するデータを渡します。詳細については、「IVS チャット API リファレンス」の「[CreateChatToken](#)」を参照してください。

AWS SDK での手順

AWS SDK を使用してチャットトークンを作成するには、最初にアプリケーションに SDK をダウンロードして設定する必要があります。以下は、JavaScript による AWS SDK の手順です。

重要: このコードは、サーバー側で実行し、その出力をクライアントに渡す必要があります。

前提条件: 以下のコードサンプルを使用するには、AWS JavaScript SDK をアプリケーションにロードする必要があります。詳細については、[Getting started with the AWS SDK for JavaScript](#) を参照してください。

```
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 の手順

AWS CLI を使用してチャットトークンを作成することは詳細オプションであり、最初に CLI をダウンロードしてマシンに設定する必要があります。詳細については、「[AWS Command Line Interface のユーザーガイド](#)」を参照してください。注: AWS CLI を使用したトークンの生成はテスト目的の場合には問題ありませんが、本番環境で使用する場合は、AWS SDK を使用してサーバー側でトークンを生成することをお勧めします (上記の手順を参照)。

1. クライアントのルーム識別子およびユーザ ID とともに `create-chat-token` のコマンドを実行します。以下のいずれかの機能を含めます:

"SEND_MESSAGE"、"DELETE_MESSAGE"、"DISCONNECT_USER"。(オプションで、このチャットセッションに関するセッション期間 (分単位) やカスタム属性 (メタデータ) を含めます。これらのフィールドは以下には表示されません。)

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

2. これはクライアントトークンを返します。

```
{
  "token":
  "abcde12345FGHIJ67890_klmno1234PQRS567890uvwxyz1234.abcd12345EFGHI67890_jklmno123PQRS567890",
  "sessionExpirationTime": "2022-03-16T04:44:09+00:00",
  "tokenExpirationTime": "2022-03-16T03:45:09+00:00"
```

```
}
```

- このトークンを保存します。チャットルームに接続し、メッセージを送受信するには、これが必要です。セッションが終了する前に、別のチャットトークンを生成する必要があります (sessionExpirationTime で示されています)。

ステップ 4: 最初のメッセージを送受信する

チャットトークンを使用してチャットルームに接続し、最初のメッセージを送信します。サンプルの JavaScript コードを以下に示します。IVS クライアント SDK も利用可能です。「[チャット SDK: Android ガイド](#)」、「[チャット SDK: iOS ガイド](#)」、「[チャット SDK: JavaScript ガイド](#)」をご覧ください。

リージョンサービス: 以下のサンプルコードは、お客様の「サポート対象リージョン」を示しています。Amazon IVS Chat では、リクエストの実行に使用できるリージョンのエンドポイントを提供しています。Amazon IVS Chat メッセージング API の場合、リージョンエンドポイントの一般的な構文は次のとおりです。

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

例えば、米国西部 (オレゴン) リージョンのエンドポイントは `wss://edge.ivschat.us-west-2.amazonaws.com` です。サポートされているリージョンのリストについては、「AWS 全般のリファレンス」の [Amazon IVS ページ](#)にある Amazon IVS Chat 情報を参照してください。

```
/*
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);
}
```

おめでとうございます、これで準備万端です！これで、メッセージを送受信できるシンプルなチャットアプリケーションができました。

ステップ 5: Service-Quota 制限の確認 (オプション)

チャットルームは Amazon IVS ライブストリームに合わせて拡張され、すべての視聴者がチャットの会話に参加できるようになります。ただし、すべての Amazon IVS アカウントには、同時チャット参加者数およびメッセージ配信速度に制限があります。

制限が適切であることを確認し、必要に応じて引き上げをリクエストします (特に大規模なストリーミングイベントを計画している場合)。詳細については、「[Service Quotas \(Low-Latency Streaming\)](#)」、「[Service Quotas \(Real-Time Streaming\)](#)」、および「[Service Quotas \(Chat\)](#)」を参照してください。

IVS Chat のログ記録

チャットのログ記録機能を使用すると、ルーム内のすべてのメッセージを Amazon S3 バケット、Amazon CloudWatch Logs、または Amazon Kinesis Data Firehose のいずれかに記録できます。その後、ライブビデオセッションにリンクするチャットリプレイの作成や分析にログを使用できます。

ルームでチャットのログ記録を有効にする

チャットのログ記録は、ログ記録設定をルームに関連付けることで有効にできる詳細オプションです。ログ記録設定は、ルームのメッセージが記録される場所のタイプ (Amazon S3 バケット、Amazon CloudWatch Logs、または Amazon Kinesis Data Firehose) を指定できるリソースです。ログ記録設定の作成と管理の詳細については、「[Amazon IVS Chatの開始方法](#)」および「[Amazon IVS Chat API リファレンス](#)」を参照してください。

新しいルームの作成 ([CreateRoom](#))、または既存のルームの更新 ([UpdateRoom](#)) のいずれかで、各ルームに最大 3 つのログ記録設定を関連付けることができます。複数のルームを同じログ記録設定に関連付けることができます。

アクティブなログ記録設定がルームに少なくとも 1 つ関連付けられている場合、そのルームに送信されたすべてのメッセージリクエストが、[Amazon IVS Chat メッセージング API](#) を介して指定された場所に自動的に記録されます。伝播遅延の平均値 (メッセージリクエストが送信されてから指定した場所で利用可能になるまで) は次の通りです。

- Amazon S3 バケット: 5 分
- Amazon CloudWatch Logs または Amazon Kinesis Data Firehose: 10 秒

メッセージの内容

[形式]

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

}

フィールド

フィールド	説明
event_timestamp	Amazon IVS Chat でメッセージが受信された時刻の UTC タイムスタンプ。
payload	クライアントが Amazon IVS Chat サービスから受け取る メッセージ (サブスクライブ) または イベント (サブスクライブ) の JSON ペイロード。
type	チャットメッセージのタイプ。 • 有効な値: MESSAGE EVENT
version	メッセージおよびコンテンツ形式のバージョン。

Amazon S3 バケット

[形式]

メッセージログは、次の S3 プレフィックスとファイル形式で整理および保存されます。

```
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>
```

フィールド

フィールド	説明
<account_id>	ルームの作成元となる AWS アカウントの ID。
<hash>	一意性を確保するためシステムにより生成されるハッシュ値。

フィールド	説明
<region>	ルームが作成された AWS サービスのリージョン。
<resource_id>	ルーム ARN のリソース ID 部分。
<version>	メッセージおよびコンテンツ形式のバージョン。
<year> / <month> / <day> / <hours> / <minute>	Amazon IVS Chat でメッセージが受信された時刻の UTC タイムスタンプ。

例

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

Amazon CloudWatch Logs

[形式]

メッセージログは、次のログストリーム名の形式で整理および保存されます。

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

フィールド

フィールド	説明
<resource_id>	ルーム ARN のリソース ID 部分。
<version>	メッセージおよびコンテンツ形式のバージョン。

例

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

Amazon Kinesis Data Firehose

メッセージログは、リアルタイムのストリーミングデータとして配信ストリームに送信されます。送信先は、Amazon Redshift、Amazon OpenSearch Service、Splunk、およびカスタム HTTP エンドポイントまたはサードパーティのサービスプロバイダーが所有する HTTP エンドポイントなどです。詳細については、「[What Is Amazon Kinesis Data Firehose](#)」(Amazon Kinesis Data Firehose とは何ですか?) を参照してください。

制約

- メッセージが保存されるログ記録の場所を所有している必要があります。
- ルーム、ログ記録設定、およびログ記録の場所は、同じ AWS リージョン内にある必要があります。
- チャットのログ記録に使用できるのは、アクティブなログ記録設定だけです。
- どのルームにも関連付けられていないログ記録設定は削除できます。

所有する場所にメッセージをログ記録するには、AWS 認証情報を使用した承認が必要です。IVS チャットに必要なアクセス権を付与するため、ログ記録設定の作成時に、リソースポリシー (Amazon S3 バケットまたは CloudWatch Logs の場合) または AWS IAM の[サービスにリンクされたルール](#) (SLR) (Amazon Kinesis Data Firehose の場合) が自動的に生成されます。ルールやポリシーを変更すると、チャットのログ記録のアクセス許可に影響する可能性があることに注意してください。

Amazon CloudWatch によるエラーのモニタリング

Amazon CloudWatch を使用して、チャットのログ記録で発生するエラーをモニタリングできます。また、アラームやダッシュボードを作成して、特定のエラーの変化を表示したり対応できます。

エラーにはいくつかの種類があります。詳細については、「[Monitoring Amazon IVS Chat](#)」を参照してください。

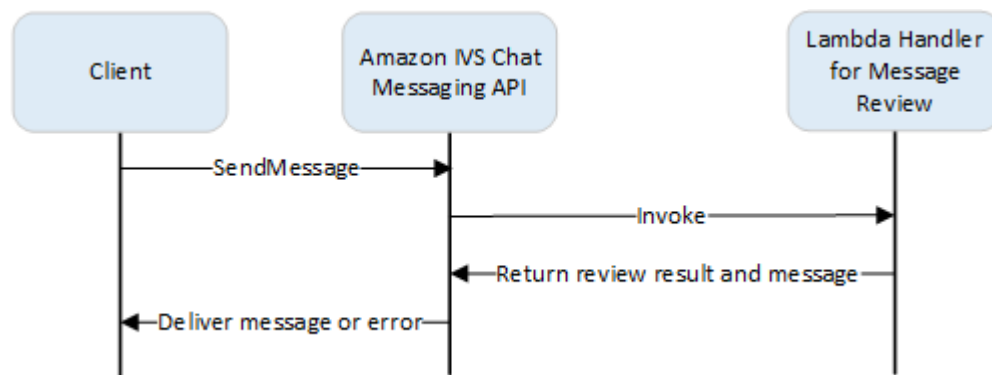
IVS Chat メッセージレビューハンドラ

メッセージレビューハンドラを使用すると、ルームに配信される前にメッセージをレビューおよび/または修正することができます。メッセージレビューハンドラがルームに関連付けられている場合、そのルームへの SendMessage リクエストごとに呼び出されます。ハンドラは、アプリケーションのビジネスロジックを実行し、メッセージを許可、拒否、または修正するかどうかを決定します。Amazon IVS Chat は AWS Lambda 関数をハンドラとしてサポートしています。

Lambda 関数の作成

ルームのメッセージレビューハンドラを設定する前に、リソースベースの IAM ポリシーを使用して Lambda 関数を作成する必要があります。Lambda 関数は、関数を使用するルームと同じ AWS アカウントおよび AWS リージョン内に存在する必要があります。リソースベースのポリシーは、Lambda 関数を呼び出すアクセス許可を Amazon IVS Chat に付与します。手順については、「[Resource-Based Policy for Amazon IVS Chat](#)」を参照してください。

ワークフロー



リクエストの構文

クライアントがメッセージを送信すると、Amazon IVS Chat は JSON ペイロードを使用して Lambda 関数を呼び出します。

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

```

    "Attributes": { "string": "string" },
    "UserId": "string",
    "Ip": "string"
  }
}

```

リクエスト本文

フィールド	説明
Attributes	メッセージに関連付けられた属性。
Content	メッセージの元のコンテンツ。
MessageId	メッセージ ID。IVS チャットによって生成されます。
RoomArn	メッセージが送信されるルームの ARN。
Sender	送信者に関する情報。このオブジェクトにはいくつかのフィールドがあります。 <ul style="list-style-type: none"> Attributes — 認証中に確立された送信者に関するメタデータ。これは、アバター URL、バッジ、フォント、色など、送信者に関する詳細情報をクライアントに提供するために使用できます。 UserId — このメッセージを送信した視聴者 (エンドユーザー) のアプリケーションで指定した識別子。これは、クライアントアプリケーションがメッセージング API またはアプリケーションドメインでユーザーを参照するために使用できます。 Ip - メッセージを送信するクライアントの IP アドレス。

レスポンスの構文

ハンドラ Lambda 関数は、次の構文で JSON レスポンスを返す必要があります。以下の構文に対応しない、またはフィールド制約を満たすレスポンスは無効です。この場合、メッセージレビューハンドラで指定した `FallbackResult` の値に応じて、メッセージが許可または拒否されます。「Amazon IVS Chat API リファレンス」の「[MessageReviewHandler](#)」を参照してください。

```
{
```

```

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

```

レスポンスフィールド

フィールド	説明
Attributes	<p>Lambda 関数から返されたメッセージに関連付けられた属性。</p> <p>ReviewResult が DENY の場合、Attributes には Reason を指定できません。例:</p> <pre>"Attributes": {"Reason": "denied for moderation"}</pre> <p>この場合、送信側クライアントはエラーメッセージに理由を記載した WebSocket 406 エラーを受信します。(「Amazon IVS Chat メッセージング API リファレンス」の「WebSocket エラー」を参照してください。)</p> <ul style="list-style-type: none"> • サイズの制約: 最大 1 KB • 必須: いいえ
Content	<p>Lambda 関数から返されたメッセージの内容。ビジネスロジックに応じて、編集したり、元のメッセージにしたりできます。</p> <ul style="list-style-type: none"> • 長さの制限: 最小長は 1 です。ルームを作成/更新したときに定義した MaximumMessageLength の最大長。詳細については、「Amazon IVS Chat API リファレンス」を参照してください。これは、ReviewResult が ALLOW の場合にのみ適用されます。 • 必須: はい
ReviewResult	<p>メッセージの処理方法に関するレビュー処理の結果。許可されている場合、メッセージはルームに接続されているすべてのユーザーに配信されます。拒否された場合、メッセージはどのユーザーにも配信されません。</p> <ul style="list-style-type: none"> • 有効な値: ALLOW DENY • 必須: はい

サンプルコード

以下は、Go の Lambda ハンドラのサンプルです。メッセージの内容を修正し、メッセージ属性を変更せずにメッセージを許可します。

```
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
}
```

ハンドラとルームとの関連付けと関連付け解除

Lambda ハンドラのセットアップと実装が完了したら、[Amazon IVS Chat API](#) を使用します。

- ハンドラをルームに関連付けるには、CreateRoom または UpdateRoom を呼び出して、ハンドラを指定します。
- ハンドラをルームとの関連付けを解除するには、MessageReviewHandler.Uri に空の値を使用して UpdateRoom を呼び出します。

Amazon CloudWatch によるエラーのモニタリング

Amazon CloudWatch を使用して、メッセージレビューで発生するエラーをモニタリングできます。また、アラームやダッシュボードを作成し、特定のエラーの変化を表示したり、対応したりすることができます。エラーが発生した場合、ハンドラをルームに関連付ける際に指定した FallbackResult の値に応じて、メッセージが許可または拒否されます。「Amazon IVS Chat API リファレンス」の「[MessageReviewHandler](#)」を参照してください。

エラーにはいくつかの種類があります。

- InvocationErrors は、Amazon IVS Chat がハンドラを呼び出すことができない場合に発生します。
- ResponseValidationErrors は、ハンドラが無効なレスポンスを返したときに発生します。
- AWS Lambda Errors は、Lambda ハンドラが呼び出されたときに関数エラーを返した場合に発生します。

Amazon IVS Chat によって発行される呼び出しエラーとレスポンス検証エラーの詳細については、「[Monitoring Amazon IVS Chat](#)」を参照してください。AWS Lambda のエラーの詳細については、「[Lambda 関数のメトリクスの使用](#)」を参照してください。

Amazon IVS Chat のモニタリング

Amazon CloudWatch を使用して、Amazon Interactive Video Service (IVS) Chat のリソースをモニタリングできます。CloudWatch は Amazon IVS Chat から未加工データを収集し、リアルタイムに近い読み取り可能なメトリクスに加工します。これらの統計は 15 か月間保持されるため、ウェブアプリケーションやサービスの動作に関する履歴情報を取得できます。特定のしきい値にアラームを設定し、これらのしきい値に達したときに通知を送信したりアクションを実行したりできます。詳細については、[CloudWatch ユーザーガイド](#)を参照してください。

CloudWatch メトリクスへのアクセス

Amazon CloudWatch は Amazon IVS Chat から未加工データを収集し、ほぼリアルタイムの読み取り可能なメトリクスに加工します。これらの統計は 15 か月間保持されるため、ウェブアプリケーションやサービスの動作に関する履歴情報を取得できます。特定のしきい値にアラームを設定し、これらのしきい値に達したときに通知を送信したりアクションを実行したりできます。詳細については、[CloudWatch ユーザーガイド](#)を参照してください。

CloudWatch メトリクスは時間の経過とともにロールアップされることに注意してください。メトリクスの経過とともに解像度は実質的に低下します。スケジュールは次のとおりです。

- 60 秒のメトリクスは 15 日間使用できます。
- 5 分間のメトリクスは 63 日間使用できます。
- 1 時間のメトリクスは、455 日 (15 か月) 間使用できます。

データ保持に関する最新情報については、[Amazon CloudWatch のよくある質問](#)の「保持期間」を検索してください。

CloudWatch コンソールでの手順

1. CloudWatch コンソール (<https://console.aws.amazon.com/cloudwatch/>) を開きます。
2. サイドナビゲーションで、[メトリクス] ドロップダウンをクリックし、[すべてのメトリクス] を選択します。
3. [参照] タブで、左側のラベルなしのドロップダウンを使用して、チャンネルが作成された「ホーム」リージョンを選択します。リージョンの詳細については、「[グローバルソリューション、リージョナルコントロール](#)」を参照してください。対応するリージョンの一覧については、「[AWS 全般のリファレンス](#)」の Amazon IVS のページを参照してください。

4. [参照] タブの下部で IVSChat 名前空間を選択します。
5. 次のいずれかを行います。
 - a. 検索バーに、リソース ID (ARN の一部、arn::*ivschat:room/<resource id>*) を入力します。

次に、[IVSChat] を選択します。

- b. [AWS の名前空間]に [IVSChat] が選択可能なサービスとして表示される場合、それを選択します。Amazon IVSChat を使用していて、Amazon CloudWatch にメトリクスを送信している場合にリスト表示されます。(IVSChat がリスト表示されていない場合、Amazon IVSChat メトリクスはありません)。

次に、必要に応じてディメンショングループを選択します。使用可能なディメンションは、以下の「[CloudWatch メトリクス](#)」にリストされています。

6. グラフに追加するメトリクスを選択します。利用可能なメトリックスは、以下の「[CloudWatch メトリクス](#)」にリストされています。

チャットセッションの詳細ページで [CloudWatch で表示] ボタンを選択することで、チャットセッションの CloudWatch グラフにアクセスすることもできます。

CLI の手順

AWS CLI を使用してメトリクスにアクセスすることもできます。そのためには、まず CLI をマシンにダウンロードして設定する必要があります。詳細については、[AWS コマンドラインインターフェイスのユーザーガイド](#)を参照してください。

次に、AWS CLI を使用して Amazon IVS 低レイテンシーチャットメトリクスにアクセスするために、次の操作を実行します。

- コマンドプロンプトで、次のコマンドを実行します。

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

詳細については、Amazon CloudWatch ユーザーガイドの「[Amazon CloudWatch メトリクスの使用](#)」を参照してください。

CloudWatch メトリクス: IVS Chat

Amazon IVS Chat は、AWS/IVSChat の名前空間で以下のメトリクスを提供します。

メトリクス	ディメンション	説明
ConcurrentChatConnections	なし	<p>チャットルームでの同時接続の総数 (1 分あたりの最大レポート数)。これは、お客様がリージョン内の同時チャット接続の上限に近づいたときに、それを把握するのに役立ちます。</p> <p>単位: 個</p> <p>有効な統計: 合計、平均、最大、最小</p>
Deliveries	アクション	<p>リージョン内のすべてのルームでのチャット接続への特定のアクションタイプで行われたメッセージングリクエストの配信数。</p> <p>単位: 個</p> <p>有効な統計: 合計、平均、最大、最小</p>
InvocationErrors	Uri	<p>リージョン内のすべてのルームにおける特定のメッセージレビューハンドラの呼び出しエラーの数。呼び出しエラーは、メッセージレビューハンドラーを呼び出すことができない場合に発生します。</p> <p>呼び出しエラーは、Amazon IVS Chat がハンドラーを呼び出すことができない場合に発生します。これは、ルームに関連付けられているハンドラーが存在しないかタイムアウトした場合、またはそのリソースポリシーでサービスがルームを呼び出すことを許可していない場合に発生する可能性があります。</p> <p>単位: 個</p>

メトリクス	ディメンション	説明
LogDestinationAccessDeniedError	LoggingConfiguration	<p>リージョン内のすべてのルームにおけるログ送信先のアクセス拒否エラーの数。</p> <p>これらのエラーは、Amazon IVS Chat がログ記録設定で指定された送信先リソースにアクセスできない場合に発生します。これは、送信先リソースポリシーで、サービスにレコードの配置が許可されていない場合に発生する可能性があります。</p> <p>単位: 個</p> <p>有効な統計: 合計、平均、最大、最小</p>
LogDestinationErrors	LoggingConfiguration	<p>リージョン内のすべてのルームにおけるログ送信先のすべてのエラーの数。</p> <p>これは、あらゆる種類のエラーを含む集計されたメトリクスで、Amazon IVS Chat がログ記録設定で指定された送信先リソースにログを配信できなかった場合に発生するエラーなどが含まれます。</p> <p>単位: 個</p> <p>有効な統計: 合計、平均、最大、最小</p>

メトリクス	ディメンション	説明
LogDestinationResourceNotFoundErrors	LoggingConfiguration	<p>リージョン内のすべてのルームにおけるログ送信先のリソースが見つからないエラーの数。</p> <p>これらのエラーは、リソースが存在しないために Amazon IVS Chat がログ記録設定で指定された送信先リソースにログを配信できない場合に発生します。これは、ログ記録設定に関連する送信先リソースが存在しなくなった場合に発生する可能性があります。</p> <p>単位: 個</p> <p>有効な統計: 合計、平均、最大、最小</p>
MessagingDeliveries	なし	<p>リージョン内のすべてのルームにわたるチャット接続へのメッセージングリクエストの配信数。</p> <p>単位: 個</p> <p>有効な統計: 合計、平均、最大、最小</p>
MessagingRequests	なし	<p>リージョン内のすべてのルームで行われたメッセージリクエストの数。</p> <p>単位: 個</p> <p>有効な統計: 合計、平均、最大、最小</p>
Requests	アクション	<p>リージョン内のすべてのルームで特定のアクションタイプで行われたリクエストの数。</p> <p>単位: 個</p> <p>有効な統計: 合計、平均、最大、最小</p>

メトリクス	ディメンション	説明
ResponseValidationErrors	Uri	<p>リージョン内のすべてのルームにおける特定のメッセージレビューハンドラーのレスポンス検証エラーの数。メッセージレビューハンドラーからの応答が無効であると、レスポンス検証エラーが発生します。これは、レスポンスを解析できなかったり、検証チェックに失敗した可能性があります。たとえば、無効なレビュー結果やレスポンス値が長すぎる可能性があります。</p> <p>単位: 個</p> <p>有効な統計: 合計、平均、最大、最小</p>

IVS Chat Client Messaging SDK

Amazon Interactive Video Service (IVS) Chat Client Messaging SDK は、Amazon IVS を使用してアプリケーションを構築するデベロッパー向けのものです。この SDK は、Amazon IVS アーキテクチャを活用するように設計されており、Amazon IVS Chat と更新されます。ネイティブの SDK として、アプリケーションおよびユーザーがアプリケーションにアクセスするデバイスに対してパフォーマンスへの影響を最小限に抑えるように設計されています。

プラットフォームの要件

デスクトップブラウザ

ブラウザ	サポートされているバージョン
Chrome	2 つのメジャーバージョン (最新バージョンと 1 つ前のバージョン)
Edge	2 つのメジャーバージョン (最新バージョンと 1 つ前のバージョン)
Firefox	2 つのメジャーバージョン (最新バージョンと 1 つ前のバージョン)
Opera	2 つのメジャーバージョン (最新バージョンと 1 つ前のバージョン)
Safari	2 つのメジャーバージョン (最新バージョンと 1 つ前のバージョン)

モバイルブラウザ

ブラウザ	サポートされるバージョン
Chrome for Android	2 つのメジャーバージョン (最新バージョンと 1 つ前のバージョン)
Android 向け Firefox	2 つのメジャーバージョン (最新バージョンと 1 つ前のバージョン)
Android 向け Opera	2 つのメジャーバージョン (最新バージョンと 1 つ前のバージョン)

ブラウザ	サポートされるバージョン
WebView Android	2つのメジャーバージョン (最新バージョンと1つ前のバージョン)
サムスン・インターネット	2つのメジャーバージョン (最新バージョンと1つ前のバージョン)
iOS 向け Safari	2つのメジャーバージョン (最新バージョンと1つ前のバージョン)

ネイティブプラットフォーム

プラットフォーム	サポートされているバージョン
Android	5.0 以降
iOS	13.0 以降

サポート

チャットルームのエラーやその他の問題が発生した場合は、IVS チャット API を介してチャットルームの識別子を特定します ([LisListRooms](#) を参照)。

このチャットルームの識別子を AWS サポートに伝えます。それにより、サポートはトラブルシューティングに役立つ情報を入手できます。

注意: 利用可能なバージョンと修正済みの問題については、「[Amazon IVS Chat Release Notes](#)」を参照してください。必要な場合、サポートに連絡する前にお使いの SDK のバージョンを更新し、問題が解決するかどうか確認してください。

バージョンニング

Amazon IVS Chat Client Messaging SDK は、[セマンティックバージョンニング](#)を使用しています。

以下の解説は、次を前提としています。

- 最新リリースは 4.1.3。

- 1つ前のメジャーバージョンの最新リリースは 3.2.4。
- バージョン 1.x の最新リリースは 1.5.6。

最新バージョンのマイナーリリースとして、下位互換性のある新機能が追加されています。この場合、次回の新機能のセットは、バージョン 4.2.0 として追加されます。

下位互換性のあるマイナーなバグ修正が、最新バージョンのパッチリリースとして追加されています。ここでは、次回のマイナーなバグ修正のセットは、バージョン 4.1.4 として追加されます。

下位互換性のあるメジャーなバグ修正は異なる方法で処理されます。これらはいくつかのバージョンに追加されています。

- 最新バージョンのパッチリリース。こちらは、バージョン 4.1.4 です。
- 1つ前のマイナーバージョンのパッチリリース。こちらは、バージョン 3.2.5 です。
- 最新バージョン 1.x リリースのパッチリリース。こちらは、バージョン 1.5.7 です。

メジャーなバグ修正は、Amazon IVS 製品チームによって定義されています。典型的な例に、重要なセキュリティ更新のほか、お客様に必要な選別された修正があります。

注: 上記の例では、リリースされたバージョンの数字は、連番でインクリメントされています(4.1.3 → 4.1.4、など)。実際は、1つ以上のパッチ番号が内部に残り、リリースされないままになることもあります。そのため、リリースされたバージョンは 4.1.3 から (例えば) 4.1.6 に増えることもあります。

また、バージョン 1.x は 2023 年末まで、または 3.x がリリースされるまでのいずれか遅い方までサポートされます。

Amazon IVS Chat API

サーバー側 (SDK によって管理されていない) には 2 つの API があり、それぞれに次の役割があります。

- データプレーン - [IVS チャットメッセージ API](#) は、トークンベースの認証スキームによって駆動されるフロントエンドアプリケーション (iOS、Android、macOS など) で使用するよう設計された WebSocket API です。以前に生成されたチャットトークンを使用して、この API を使用して既存のチャットルームに接続します。

Amazon IVS Chat Client Messaging SDK はデータプレーンのみに関係します。SDK は、バックエンドですでにチャットトークンを生成していることを前提としています。これらのトークンの取得は、SDK ではなくフロントエンドアプリケーションによって管理されることを前提としています。

- コントロールプレーン - [IVS チャットコントロールプレーン API](#) は独自のバックエンドアプリケーションにインターフェイスを提供して、チャットルームとそこに参加するユーザーを管理および作成します。これは、独自のバックエンドによって管理されるアプリのチャットエクスペリエンスの管理パネルと考えてください。データプレーンがチャットルームに対して認証するために必要なチャットトークンの作成を担当する control-plane エンドポイントがあります。

重要: IVS Chat Client Messaging SDK SDK は、control-plane エンドポイントを呼び出しません。チャットトークンを作成するには、バックエンドを設定する必要があります。このチャットトークンを取得するには、フロントエンドアプリケーションがバックエンドと通信する必要があります。

IVS Chat Client Messaging SDK: Android ガイド

Amazon Interactive Video (IVS) Chat Client Messaging Android SDK は、Android を使用するプラットフォームに [IVS チャットメッセージング API](#) を簡単に組み込むことができるインターフェイスを提供します。

com.amazonaws:ivs-chat-messaging パッケージは、このドキュメントで説明されているインターフェイスを実装します。

IVS Chat Client Messaging Android SDK の最新バージョン: 1.1.0 ([リリースノート](#))

リファレンスドキュメント: Amazon IVS Chat Client Messaging Android SDK で使用できる最も重要なメソッドについては、リファレンスドキュメント (<https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/>) を参照してください。

サンプルコード: GitHub の Android サンプルリポジトリ (<https://github.com/aws-samples/amazon-ivs-chat-for-android-demo>) を参照してください。

プラットフォームの要件: 開発には Android 5.0 (API レベル 21) 以上が必要です。

概要

開始する前に、「[Amazon IVS Chat の開始方法](#)」を理解しておく必要があります。

Package の追加

次の build.gradle 依存関係を com.amazonaws:ivs-chat-messaging に追加します。

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

ProGuard ルールの追加

R8/Proguard ルールファイル (proguard-rules.pro) に次のエントリを追加してください。

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

バックエンドのセットアップ

この統合には、[Amazon IVS API](#) と通信するサーバ上のエンドポイントが必要です。サーバーから Amazon IVS API へのアクセスに[公式の AWS ライブラリ](#)を使用します。これらはパブリックパッケージ (node.js や Java など) から、複数の言語でアクセスできます。

次に、[Amazon IVS Chat API](#)と通信するサーバーエンドポイントを作成し、トークンを作成します。

サーバー接続を設定する

ChatTokenCallback をパラメータとし、バックエンドからチャットトークンをフェッチするメソッドを作成します。そのトークンをコールバックの onSuccess メソッドに渡します。エラーが発生した場合、コールバックの onError メソッドへ例外を渡します。これは、次のステップでメイン ChatRoom エンティティをインスタンス化するために必要です。

以下に、Retrofit 呼び出しを使用して上記を実装するサンプルコードを示します。

```
// ...  
  
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 の使用

チャットルームインスタンスを初期化する

ChatRoom クラスのインスタンスを作成します。これには、通常、チャットルームがホストされている AWS リージョンである tokenProvider と、前のステップで作成されたトークン取得方法である regionOrUrl を渡す必要があります。

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

次に、チャットに関連したイベントのハンドラーを実装するリスナーオブジェクトを作成し、それを room.listener プロパティに割り当てます。

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

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

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

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

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

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

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

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

// ...
```

基本的な初期化の最後のステップは、WebSocket 接続を確立することによって特定のルームに接続することです。これを行うには、ルームインスタンス内の `connect()` メソッドを呼び出します。アプリがバックグラウンドから再開した場合に接続を維持するには、`onResume()` ライフサイクルメソッドでこれを行うことをお勧めします。

```
room.connect()
```

SDK は、サーバーから受信したチャットトークンにエンコードされたチャットルームへの接続を確立しようとします。失敗した場合、ルームインスタンスで指定された回数だけ再接続を試みます。

チャットルームでアクションを実行する

`ChatRoom` クラスには、メッセージの送信と削除、他のユーザーの接続解除を行うアクションがあります。これらのアクションは、リクエストの確認または拒否の通知を受け取ることができるオプションのコールバックパラメータを受け入れます。

メッセージの送信

このリクエストには、チャットトークンにエンコードされた SEND_MESSAGE 機能が必要です。

send-message リクエストをトリガーする方法

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

リクエストの確認/拒否を取得するには、2 つ目のパラメータとして次のコールバックを指定します。

```
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.
    }
})
```

メッセージの削除

このリクエストでは、チャットトークンに DELETE_MESSAGE 機能がエンコードされている必要があります。

delete-message リクエストをトリガーするには

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

リクエストの確認/拒否を取得するには、2 つ目のパラメータとして次のコールバックを指定します。

```
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.
    }
})
```

他のユーザーの接続を切断する

このリクエストには、チャットトークンにエンコードされた DISCONNECT_USER 機能が必要です。

モデレーションを目的として他のユーザーとの接続を切断するには

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

リクエストの確認/拒否を取得するには、2 つ目のパラメータとして次のコールバックを指定します。

```
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.
    }
})
```

チャットルームの接続を切断する

チャットルームへの接続を閉じるには、ルームインスタンスで次の disconnect() メソッドを呼び出します。

```
room.disconnect()
```

アプリケーションがバックグラウンド状態になると、WebSocket 接続はしばらくすると機能しなくなります。バックグラウンド状態から移行またはバックグラウンド状態への移行時には、手動で接続/切断することをおすすめします。Android Activity または Fragment の onResume() ライフサイクルメソッドの room.connect() 呼び出しと、 onPause() ライフサイクルメソッドの room.disconnect() 呼び出しを一致させます。

IVS Chat Client Messaging SDK: Android のチュートリアルパート 1: チャットルーム

これは 2 部構成のチュートリアルの第 1 部です。[Kotlin](#) プログラミング言語を使用して完全に機能する Android アプリを構築することにより、Amazon IVS Chat Messaging SDK の操作の要点を学びます。このアプリケーションは、Chatterbox と呼ばれます。

モジュールを開始する前に、数分の時間を割いて、前提条件、チャットトークンの主な概念、チャットルームの作成に必要なバックエンドサーバーについて理解しておいてください。

これらのチュートリアルは、IVS Chat Messaging SDK を初めて利用する、経験豊富な Android デベロッパー向けに作成されています。Kotlin プログラミング言語と Android プラットフォームでの UI の作成に慣れている必要があります。

このチュートリアルの最初の部分は、いくつかのセクションに分かれています。

1. [the section called “ローカル認証サーバーおよび認可サーバーのセットアップ”](#)
2. [the section called “Chatterbox プロジェクトの作成”](#)
3. [the section called “チャットルームに接続し、接続の更新を確認する”](#)
4. [the section called “トークンプロバイダーの作成”](#)
5. [the section called “次のステップ”](#)

すべての SDK ドキュメントについては、まずこの「Amazon IVS Chat ユーザーガイド」の「[Amazon IVS Chat Client Messaging SDK](#)」および GitHub の「[Chat Client Messaging: SDK for Android リファレンス](#)」を参照してください。

前提条件

- Kotlin と Android プラットフォームでのアプリケーションの作成に慣れておいてください。Android 向けのアプリケーションの作成に慣れていない場合は、Android デベロッパー向けの「[初めての Android アプリを作成する](#)」のガイドで基本を学んでください。
- [IVS Chat の開始方法](#) をよく読み、理解しておいてください。
- 既存の IAM ポリシーで定義されている CreateChatToken および CreateRoom 機能を持つ、AWS IAM ユーザーを作成してください。(「[IVS Chat の開始方法](#)」を参照してください。)

- このユーザーのシークレットキーまたはアクセスキーが、AWS 認証情報ファイルに保存されていることを確認してください。手順については、「[AWS CLI ユーザーガイド](#)」(特に「[設定ファイルと認証情報ファイルの設定](#)」)を参照してください。
- チャットルームを作成し、その ARN を保存してください。「[IVS Chat の開始方法](#)」を参照してください。(ARN を保存しない場合、後でコンソールまたは Chat API で参照できます。)

ローカル認証サーバーおよび認可サーバーのセットアップ

バックエンドサーバーは、チャットルームの作成と、IVS Chat Android SDK がチャットルーム用のクライアントを認証および承認するために必要なチャットトークンの生成の両方を行います。

「Amazon IVS Chat の開始方法」の「[チャットトークンを作成する](#)」を参照してください。フローチャートで示されているように、チャットトークンの作成はサーバー側のコードで行われます。つまり、サーバー側のアプリケーションからリクエストし、独自の方法でチャットトークンを生成する必要があります。

[Ktor](#) フレームワークを使用して、ローカルの AWS 環境でチャットトークンの作成を管理するライブローカルサーバーを作成します。

この時点で、AWS 認証情報が正しく設定されている必要があります。詳しい手順については、「[開発用に一時的な AWS 認証情報とリージョンを設定する](#)」を参照してください。

chatterbox という新しいディレクトリを作成し、その中に auth-server という別のディレクトリを作成します。

サーバーフォルダは次の構造になります。

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


注: このコードを参照ファイルに直接コピーして貼り付けることができます。

次に、認証サーバーが動作するために必要なすべての依存関係とプラグインを追加します。

Kotlin スクリプト:

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

ここで、認証サーバー用にログ記録機能を設定する必要があります。(詳細については、「[Configure logger](#)」を参照してください。)

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>
</configuration>
```

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

[Ktor](#) サーバーには、resources ディレクトリ内の application.* ファイルから自動的にロードされる構成設定が必要なので、それも追加します。(詳細については、「[ファイル内の設定](#)」を参照してください。)

HOCON:

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

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

最後に、サーバーを実装します。

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 プロジェクトの作成

Android プロジェクトを作成するには、[Android Studio](#) をインストールして開きます。

Android の公式な「[プロジェクトを作成する](#)」のガイドに記載されている手順に従います。

- 「[プロジェクトタイプを選択する](#)」で、Chatterbox アプリ用に [空のアクティビティ] プロジェクトテンプレートを選択します。
- 「[プロジェクトを構成する](#)」で、設定フィールドに次の値を選択します。
 - [Name]: My App
 - [Package name]: com.chatterbox.myapp
 - [Save location]: 前の手順で作成した chatterbox ディレクトリを指します
 - [Language]: Kotlin
 - [Minimum API level]: API 21: Android 5.0 (Lollipop)

すべての設定パラメータを正しく指定すると、chatterbox フォルダ内のファイル構造は次のようになります。

```
- 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
```

これで Android プロジェクトが動作するようになったので、build.gradle 依存関係に [com.amazonaws:ivs-chat-messaging](#) を追加できます。(Gradle ビルドツールキットの詳細については、「[Configure your build](#)」を参照してください)。

注: すべてのコードスニペットの先頭には、プロジェクトで変更を加える必要があるファイルへのパスがあります。パスは、プロジェクトのルートに対する相対パスです。

以下のコードでは、<version> を、Chat Android SDK の現在のバージョン番号 (1.0.0 など) に置き換えてください。

Kotlin:

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

plugins {
// ...
}

android {
// ...
}

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

新しい依存関係を追加したら、Android Studio で [Sync Project with Gradle Files] を実行して、プロジェクトを新しい依存関係と同期させます。(詳細については、「[ビルド依存関係を追加する](#)」を参照してください。)

(前のセクションで作成した) 認証サーバーをプロジェクトルートから便利に実行するために、settings.gradle で新しいモジュールとして組み込みます。(詳細については、「[Structuring and Building a Software Component with Gradle](#)」を参照してください。)

Kotlin スクリプト:

```
// ./settings.gradle

// ...

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

これで、`auth-server` が Android プロジェクトに含まれたため、そのプロジェクトのルートから次のコマンドを実行して認証サーバーを実行できます。

シェル:

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

チャットルームに接続し、接続の更新を確認する

チャットルーム接続を開くには、アクティビティが最初に作成されたときに実行される [onCreate\(\)](#) [アクティビティライフサイクルコールバック](#)を使用します。[ChatRoom コンストラクター](#)では、ルーム接続をインスタンス化するために `region` と `tokenProvider` を提供する必要があります。

注: 以下のスニペットの `fetchChatToken` 関数は、[次のセクション](#)で実装されます。

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)
    }

    // ...
}
```

chatterbox のようなチャットアプリを作成するには、チャットルーム接続の変化を表示して、これに対応することが不可欠です。ルームとのインタラクションを開始する前に、チャットルームの接続状態のイベントをサブスクライブして、最新情報を入手する必要があります。

[ChatRoom](#) では、ライフサイクルイベントを発生させるための [ChatRoomListener インターフェイス](#) 実装をアタッチすることが想定されています。現時点では、リスナー関数は呼び出されたときに確認メッセージのみをログ記録します。

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")
    }
}
}
```

ChatRoomListener の実装が完了したので、ルームインスタンスにアタッチします。

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 {
// ...
}
```

この後、ルーム接続状態を読み取る機能を提供する必要があります。これを MainActivity.kt [プロパティ](#) に保持し、ルームのデフォルトの [DISCONNECTED] 状態に初期化します (「[IVS Chat Android SDK リファレンス](#)」の「ChatRoom state」を参照してください)。ローカルの状態を最新に保つには、state-updater 関数を実装する必要があります。これを updateConnectionState と呼ぶことにします。

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")
            }
        }
    }
}
```

次に、state-updater 関数を [ChatRoom.listener](#) プロパティと統合します。

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)
    }
}
}
```

これで、[ChatRoom](#) の状態更新を保存、リッスンし、これに対応できるようになりました。次は接続を初期化します。

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

トークンプロバイダーの作成

ここで、アプリケーションでチャットトークンの作成と管理を行う関数を作成します。この例では、[Android 用の Retrofit HTTP クライアント](#)を使用しています。

ネットワークトラフィックを送信する前に、Android 用にネットワークセキュリティ構成を設定する必要があります。(詳細については、「[ネットワークセキュリティ構成](#)」を参照してください。) まず、[アプリマニフェスト](#)ファイルにネットワーク許可を追加します。user-permission タグと networkSecurityConfig 属性が追加されていることに注意してください。これらは新しいネットワークセキュリティ設定を指します。以下のコードでは、<version> を、Chat Android SDK の現在のバージョン番号 (1.0.0 など) に置き換えてください。

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 および localhost ドメインを信頼済みとして宣言して、バックエンドとのメッセージ交換を開始します。

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>
```

次に、HTTP レスポンスを解析するための [Gson コンバーターを追加](#)するとともに、新しい依存関係を追加する必要があります。以下のコードでは、<version> を、Chat Android SDK の現在のバージョン番号 (1.0.0 など) に置き換えてください。

Kotlin スクリプト:

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

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

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

チャットトークンを取得するには、chatterbox アプリから POST HTTP リクエストを実行する必要があります。Retrofit が実装するインターフェイスでリクエストを定義します。([Retrofit のドキュメント](#) を参照してください。また、 [CreateChatToken](#) エンドポイントの仕様についてもよく理解しておいてください。)

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>
}
```

ネットワークが設定されたので、チャットトークンの作成と管理を行う関数を追加します。プロジェクトが [生成](#) されたときに自動的に作成された MainActivity.kt に追加します。

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)
}
})
}
```

次のステップ

これで、チャットルーム接続を確立できました。次に、この Android のチュートリアルパート 2「[メッセージとイベント](#)」に進んでください。

IVS Chat Client Messaging SDK: Android のチュートリアルパート 2: メッセージとイベント

本チュートリアルパート 2 (最後のパート) は、複数のセクションに分かれています。

1. [the section called “メッセージ送信のために UI を作成する”](#)
 - a. [the section called “UI メインレイアウト”](#)
 - b. [the section called “テキストを一貫して表示するための UI 抽象化テキストセル”](#)
 - c. [the section called “UI 左側チャットメッセージ”](#)
 - d. [the section called “UI 右側チャットメッセージ”](#)
 - e. [the section called “UI その他の色の値”](#)
2. [the section called “ビューバインディングを適用する”](#)
3. [the section called “チャットメッセージのリクエストを管理する”](#)
4. [the section called “最後のステップ”](#)

すべての SDK ドキュメントについては、まずこの「Amazon IVS Chat ユーザーガイド」の「[Amazon IVS Chat Client Messaging SDK](#)」および GitHub の「[Chat Client Messaging: SDK for Android リファレンス](#)」を参照してください。

前提条件

このチュートリアルパート 1 である「[チャットルーム](#)」を完了してから、このパートに進んでください。

メッセージ送信のために UI を作成する

チャットルーム接続が正常に初期化されたので、最初のメッセージを送信します。この機能には、UI が必要です。次を追加します。

- connect/disconnect ボタン
- send ボタンを使用したメッセージ入力
- 動的メッセージリスト。これを構築するために、Android Jetpack [RecyclerView](#) を使用します。

UI メインレイアウト

Android デベロッパー向けドキュメントの Android Jetpack の「[Layouts](#)」を参照してください。

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>
```

テキストを一貫して表示するための UI 抽象化テキストセル

XML:

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

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

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

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

        <TextView
```

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

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

```
</LinearLayout>
```

```
</LinearLayout>
```

UI 左側チャットメッセージ

XML:

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

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

    <TextView
```

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

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

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

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

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

</LinearLayout>
```

UI 右側チャットメッセージ

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 その他の色の値

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>
```

ビューバインディングを適用する

XML レイアウト用にバインディングクラスを参照できるように、Android の [ビューバインディング機能](#) を活用します。この機能を有効にするには、./app/build.gradle で viewBinding ビルドオプションを true に設定します。

Kotlin スクリプト:

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

android {
    // ...

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

ここで、UI を Kotlin コードに接続します。

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)
    }
}
```



```
}
```

また、メッセージを削除したり、ユーザーをチャットから切断したりするメソッドも追加します。これらのメソッドは、チャットメッセージのコンテキストメニューを使用して呼び出すことができます。

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

```
}
```

チャットメッセージのリクエストを管理する

考えられるすべての状態を通じてチャットメッセージ リクエストを管理する方法が必要です。

- Pending (保留中) — メッセージはチャットルームに送信されましたが、まだ確認または拒否されていません。
- Confirmed (確認済み) — チャットルームから (私たちを含む) すべてのユーザーにメッセージが送信されました。
- Rejected (拒否) — メッセージはエラーオブジェクトによってチャットルームに拒否されました。

未解決のチャットリクエストとチャットメッセージは [リスト](#) に保持されます。このリストは別のクラスにする価値があり、私たちは `ChatEntries.kt` と呼んでいます。

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()
    }
}
```

リストを UI に接続するには、[アダプター](#)を使用します。詳細については、「[AdapterView を使用したデータへのバインディング](#)」と「[生成されたバインディングクラス](#)」を参照してください。

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

            is ChatEntry.PendingRequest -> {

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

```
        }

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

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

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

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

最後のステップ

新しいアダプターを接続して、ChatEntries クラスを MainActivity にバインドします。

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
    }
}
```



```
}
```

チャットリクエストの追跡を行うクラス (ChatEntries) が既にあるので、roomListener で entries を操作するためのコードを実装する準備が整いました。対応しているイベントに応じて entries と connectionState を更新します。

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")
    }
}
}
```

これでアプリケーションを実行できます。(「[アプリをビルドして実行する](#)」を参照してください。) アプリを使用する際には、必ずバックエンドサーバーが稼働しているようにしてください。プロジェクトのルートにあるターミナルから起動するには、このコマンド (`./gradlew :auth-server:run`) を使用するが、または Android Studio から `auth-server:run` Gradle タスクを直接実行します。

IVS Chat Client Messaging SDK: Kotlin コルーチンのチュートリアル パート 1: チャットルーム

これは 2 部構成のチュートリアルの第 1 部です。[Kotlin](#) プログラミング言語と[コルーチン](#)を使用してフル機能の Android アプリを構築することで、Amazon IVS Chat Messaging SDK を使用する際の要点を学びます。このアプリケーションは、Chatterbox と呼ばれます。

モジュールを開始する前に、数分の時間を割いて、前提条件、チャットトークンの主な概念、チャットルームの作成に必要なバックエンドサーバーについて理解しておいてください。

これらのチュートリアルは、IVS Chat Messaging SDK を初めて利用する、経験豊富な Android デベロッパー向けに作成されています。Kotlin プログラミング言語と Android プラットフォームでの UI の作成に慣れている必要があります。

このチュートリアルの最初の部分は、いくつかのセクションに分かれています。

1. [the section called “ローカル認証サーバーおよび認可サーバーのセットアップ”](#)
2. [the section called “Chatterbox プロジェクトの作成”](#)
3. [the section called “チャットルームに接続し、接続の更新を確認する”](#)
4. [the section called “トークンプロバイダーの作成”](#)
5. [the section called “次のステップ”](#)

すべての SDK ドキュメントについては、まずこの「Amazon IVS Chat ユーザーガイド」の「[Amazon IVS Chat Client Messaging SDK](#)」および GitHub の「[Chat Client Messaging: SDK for Android リファレンス](#)」を参照してください。

前提条件

- Kotlin と Android プラットフォームでのアプリケーションの作成に慣れておいてください。Android 向けのアプリケーションの作成に慣れていない場合は、Android デベロッパー向けの「[初めてのアプリを作成する](#)」のガイドで基本を学んでください。
- [IVS Chat の開始方法](#) を読んで理解してください。
- 既存の IAM ポリシーで定義されている CreateChatToken および CreateRoom 機能を持つ、AWS IAM ユーザーを作成してください。(「[IVS Chat の開始方法](#)」を参照してください。)
- このユーザーのシークレットキーまたはアクセスキーが、AWS 認証情報ファイルに保存されていることを確認してください。手順については、「[AWS CLI ユーザーガイド](#)」(特に「[設定ファイルと認証情報ファイルの設定](#)」)を参照してください。
- チャットルームを作成し、その ARN を保存してください。「[IVS Chat の開始方法](#)」を参照してください。(ARN を保存しない場合、後でコンソールまたは Chat API で参照できます。)

ローカル認証サーバーおよび認可サーバーのセットアップ

バックエンドサーバーは、チャットルームの作成と、IVS Chat Android SDK がチャットルーム用のクライアントを認証および承認するために必要なチャットトークンの生成の両方を行います。

「Amazon IVS Chat の開始方法」の「[チャットトークンを作成する](#)」を参照してください。フローチャートで示されているように、チャットトークンの作成はサーバー側のコードで行われます。つまり、サーバー側のアプリケーションからリクエストし、独自の方法でチャットトークンを生成する必要があります。

[Ktor](#) フレームワークを使用して、ローカルの AWS 環境でチャットトークンの作成を管理するライブローカルサーバーを作成します。

この時点で、AWS 認証情報が正しく設定されている必要があります。詳しい手順については、「[開発用の AWS 認証情報と AWS リージョンのセットアップ](#)」を参照してください。

chatterbox という新しいディレクトリを作成し、その中に auth-server という別のディレクトリを作成します。

サーバーフォルダは次の構造になります。

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

注: このコードを参照ファイルに直接コピーして貼り付けることができます。

次に、認証サーバーが動作するために必要なすべての依存関係とプラグインを追加します。

Kotlin スクリプト:

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

ここで、認証サーバー用にログ記録機能を設定する必要があります。(詳細については、「[Configure logger](#)」を参照してください。)

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](#) サーバーには、resources ディレクトリ内の application.* ファイルから自動的にロードされる構成設定が必要なため、それも追加します。(詳細については、「[ファイル内の設定](#)」を参照してください。)

HOCON:

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

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

最後に、サーバーを実装します。

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 roomIdIdentifier: 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().roomIdIdentifier(callParameters.roomIdentifier)
                    .userId(callParameters.userId).build()
            val token = IvschatClient.create()
                .createChatToken(request)

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

Chatterbox プロジェクトの作成

Android プロジェクトを作成するには、[Android Studio](#) をインストールして開きます。

Android の公式な「[プロジェクトを作成する](#)」のガイドに記載されている手順に従います。

- [プロジェクトを選択](#) で、Chatterbox アプリ用に [Empty Activity] プロジェクトテンプレートを選択します。
- [プロジェクトを構成する](#) で、設定フィールドに次の値を選択します。
 - [Name]: My App
 - [Package name]: com.chatterbox.myapp

- [Save location]: 前の手順で作成した chatterbox ディレクトリを指します
- [Language]: Kotlin
- [Minimum API level]: API 21: Android 5.0 (Lollipop)

すべての設定パラメータを正しく指定すると、chatterbox フォルダ内のファイル構造は次のようになります。

```
- 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
```

これで Android プロジェクトが動作するようになったので、[com.amazonaws:ivs-chat-messaging](#) と [org.jetbrains.kotlinx:kotlinx-coroutines-core](#) を build.gradle 依存関係に追加できます。(Gradle ビルドツールキットの詳細については、「[Configure your build](#)」を参照してください)。

注: すべてのコードスニペットの先頭には、プロジェクトで変更を加える必要があるファイルへのパスがあります。パスは、プロジェクトのルートに対する相対パスです。

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'  
  
    // ...  
}
```

新しい依存関係を追加したら、Android Studio で [Sync Project with Gradle Files] を実行して、プロジェクトを新しい依存関係と同期させます。(詳細については、「[ビルド依存関係を追加する](#)」を参照してください。)

(前のセクションで作成した) 認証サーバーをプロジェクトルートから便利に実行するために、settings.gradle で新しいモジュールとして組み込みます。(詳細については、「[Structuring and Building a Software Component with Gradle](#)」を参照してください。)

Kotlin スクリプト:

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

これで、auth-server が Android プロジェクトに含まれたため、そのプロジェクトのルートから次のコマンドを実行して認証サーバーを実行できます。

シェル:

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

チャットルームに接続し、接続の更新を確認する

チャットルーム接続を開くには、アクティビティが最初に作成されたときに実行される [onCreate\(\)](#) [アクティビティライフサイクルコールバック](#)を使用します。[ChatRoom コンストラクター](#)では、ルーム接続をインスタンス化するために `region` と `tokenProvider` を提供する必要があります。

注: 以下のスニペットの `fetchChatToken` 関数は、[次のセクション](#)で実装されます。

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)
    }

    // ...
}
```

`chatterbox` のようなチャットアプリを作成するには、チャットルーム接続の変化を表示して、これに対応することが不可欠です。ルームとのインタラクションを開始する前に、チャットルームの接続状態のイベントをサブスクライブして、最新情報を入手する必要があります。

コルーチン用の Chat SDK では、[ChatRoom](#) は、[Flow](#) でルームのライフサイクルイベントが処理されることを想定しています。現時点では、関数は呼び出されたときに確認メッセージのみをログ記録します。

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")
                }
            }
        }
    }
}
```

```
        }  
    }  
}
```

この後、ルーム接続状態を読み取る機能を提供する必要があります。これを MainActivity.kt プロパティに保持し、ルームのデフォルトの [DISCONNECTED] 状態に初期化します (「[IVS Chat Android SDK リファレンス](#)」の「ChatRoom state」を参照してください)。ローカルの状態を最新に保つには、state-updater 関数を実装する必要があります。これを updateConnectionState と呼ぶことにします。

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")  
            }  
        }  
    }  
}
```

次に、state-updater 関数を [ChatRoom.listener](#) プロパティと統合します。

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)
                }
            }
        }

        // ...
    }
}
```

これで、[ChatRoom](#) の状態更新を保存、リッスンし、これに対応できるようになりました。次は接続を初期化します。

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)
    }
}

// ...
}
```

トークンプロバイダーの作成

ここで、アプリケーションでチャットトークンの作成と管理を行う関数を作成します。この例では、[Android 用の Retrofit HTTP クライアント](#)を使用しています。

ネットワークトラフィックを送信する前に、Android 用にネットワークセキュリティ構成を設定する必要があります。(詳細については、「[ネットワークセキュリティ構成](#)」を参照してください。) まず、[アプリマニフェスト](#)ファイルにネットワーク許可を追加します。user-permission タグと networkSecurityConfig 属性が追加されていることに注意してください。これらは新しいネットワークセキュリティ設定を指します。以下のコードでは、<version> を、Chat Android SDK の現在のバージョン番号 (1.1.0 など) に置き換えてください。

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.myapplication">
    <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")
}
```

ローカル IP アドレス (10.0.2.2 など) と localhost ドメインを信頼できるドメインとして宣言して、バックエンドとのメッセージ交換を開始します。

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>
```

次に、HTTP レスポンスを解析するための [Gson コンバーターを追加](#)するとともに、新しい依存関係を追加する必要があります。以下のコードでは、`<version>` を、Chat Android SDK の現在のバージョン番号 (1.1.0 など) に置き換えてください。

Kotlin スクリプト:

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

チャットトークンを取得するには、chatterbox アプリから POST HTTP リクエストを実行する必要があります。Retrofit が実装するインターフェイスでリクエストを定義します。([Retrofit のドキュメント](#) を参照してください。また、 [CreateChatToken](#) エンドポイントの仕様についてもよく理解しておいてください。)

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)
    }
}
```

ネットワークが設定されたので、チャットトークンの作成と管理を行う関数を追加します。プロジェクトが[生成](#)されたときに自動的に作成された MainActivity.kt に追加します。

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)
    }
})
}
```

次のステップ

これで、チャットルーム接続を確立できました。次に、この Kotlin コルーチンのチュートリアル
のパート 2「[メッセージとイベント](#)」に進んでください。

IVS Chat Client Messaging SDK: Kotlin コルーチンのチュートリアル パート 2: メッセージとイベント

本チュートリアルのパート 2 (最後のパート) は、複数のセクションに分かれています。

1. [the section called “メッセージ送信のために UI を作成する”](#)
 - a. [the section called “UI メインレイアウト”](#)
 - b. [the section called “テキストを一貫して表示するための UI 抽象化テキストセル”](#)
 - c. [the section called “UI 左側チャットメッセージ”](#)
 - d. [the section called “UI 右側メッセージ”](#)
 - e. [the section called “UI その他の色の値”](#)
2. [the section called “ビューバインディングを適用する”](#)
3. [the section called “チャットメッセージのリクエストを管理する”](#)
4. [the section called “最後のステップ”](#)

すべての SDK ドキュメントについては、まずこの「Amazon IVS Chat ユーザーガイド」の
「[Amazon IVS Chat Client Messaging SDK](#)」および GitHub の「[Chat Client Messaging: SDK for Android リファレンス](#)」を参照してください。

前提条件

このチュートリアルパート 1 である「[チャットルーム](#)」を完了してから、このパートに進んでください。

メッセージ送信のために UI を作成する

チャットルーム接続が正常に初期化されたので、最初のメッセージを送信します。この機能には、UI が必要です。次を追加します。

- connect/disconnect ボタン
- send ボタンを使用したメッセージ入力
- 動的メッセージリスト。これを構築するために、Android Jetpack [RecyclerView](#) を使用します。

UI メインレイアウト

Android デベロッパー向けドキュメントの Android Jetpack の「[Layouts](#)」を参照してください。

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>
```

テキストを一貫して表示するための UI 抽象化テキストセル

XML:

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

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

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

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

        <TextView
```

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

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

```
</LinearLayout>
```

```
</LinearLayout>
```

UI 左側チャットメッセージ

XML:

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

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

    <TextView
```

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

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

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

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

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

</LinearLayout>
```

UI 右側メッセージ

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 その他の色の値

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>
```

ビューバインディングを適用する

XML レイアウト用にバインディングクラスを参照できるように、Android の [ビューバインディング](#) 機能を活用します。この機能を有効にするには、./app/build.gradle で viewBinding ビルドオプションを true に設定します。

Kotlin スクリプト:

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

android {
//    ...

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

ここで、UI を Kotlin コードに接続します。

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.sendMessage.setOnClickListener(::sendMessageClick)
        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 sendMessageClick(view: View) {
        val content = binding.messageEditText.text.toString()
        if (content.trim().isEmpty()) {
            return
        }

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

```
        sendMessage(request)
    }
    // ...
}
```

また、メッセージを削除したり、ユーザーをチャットから切断したりするメソッドも追加します。これらのメソッドは、チャットメッセージのコンテキストメニューを使用して呼び出すことができます。

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")
            }
        }
    }
}
```

```
}
```

チャットメッセージのリクエストを管理する

考えられるすべての状態を通じてチャットメッセージ リクエストを管理する方法が必要です。

- Pending (保留中) — メッセージはチャットルームに送信されましたが、まだ確認または拒否されていません。
- Confirmed (確認済み) — チャットルームから (私たちを含む) すべてのユーザーにメッセージが送信されました。
- Rejected (拒否) — メッセージはエラーオブジェクトによってチャットルームに拒否されました。

未解決のチャットリクエストとチャットメッセージは [リスト](#) に保持されます。このリストは別のクラスにする価値があり、私たちは `ChatEntries.kt` と呼んでいます。

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()
    }
}
```

リストを UI に接続するには、[アダプター](#)を使用します。詳細については、「[AdapterView を使用したデータへのバインディング](#)」と「[生成されたバインディングクラス](#)」を参照してください。

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

        is ChatEntry.PendingRequest -> {

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

```
        }

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

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

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

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

最後のステップ

新しいアダプターを接続して、ChatEntries クラスを MainActivity にバインドします。

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
    }
}
}
```

チャットリクエストの追跡を行うクラスが既にあるので (ChatEntries)、roomListener で entries を操作するためのコードを実装する準備が整いました。対応しているイベントに応じて entries と connectionState を更新します。

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)
    }

// ...

}
```

これでアプリケーションを実行できます。(「[アプリをビルドして実行する](#)」を参照してください。) アプリを使用する際には、必ずバックエンドサーバーが稼働しているようにしてください。プロジェクトのルートにあるターミナルから起動するには、このコマンド (`./gradlew :auth-server:run`) を使用するか、または Android Studio から `auth-server:run` Gradle タスクを直接実行します。

IVS Chat Client Messaging SDK: iOS ガイド

Amazon Interactive Video (IVS) Chat Client Messaging iOS SDK は、Apple の [Swift プログラミング言語](#) を使用したプラットフォームに [IVS Chat Messaging API](#) を組み込むことができるインターフェイスを提供します。

IVS Chat Client Messaging iOS SDK の最新バージョン: 1.0.0 ([リリースノート](#))

リファレンスドキュメントおよびチュートリアル: Amazon IVS Chat Client Messaging SDK で使用できる最も重要なメソッドについては、リファレンスドキュメント (<https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/>) を参照してください。また、このリポジトリには、さまざまな記事やチュートリアルも含まれています。

サンプルコード: GitHub の iOS サンプルリポジトリ (<https://github.com/aws-samples/amazon-ivs-chat-for-ios-demo>) を参照してください。

プラットフォームの要件: 開発には iOS 13.0 以上が必要です。

概要

[Swift Package Manager](#) を介して SDK を統合することをお勧めします。または、[CocoaPods](#) を使用するか、[フレームワークを手動で統合する](#)ことができます。

SDK を統合後、関連する Swift ファイルの先頭に次のコードを追加して SDK をインポートできます。

```
import AmazonIVSChatMessaging
```

Swift Package Manager

Swift Package Manager プロジェクトで AmazonIVSChatMessaging ライブラリを使用するには、パッケージの依存関係と関連するターゲットの依存関係にライブラリを追加します。

1. 最新の .xcframework を <https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip> からダウンロードします。
2. ターミナルで以下を実行します。

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

3. 前のステップの出力を取得し、以下に示すように、プロジェクトの Package.swift ファイル内の .binaryTarget のチェックサムプロパティに貼り付けます。

```
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

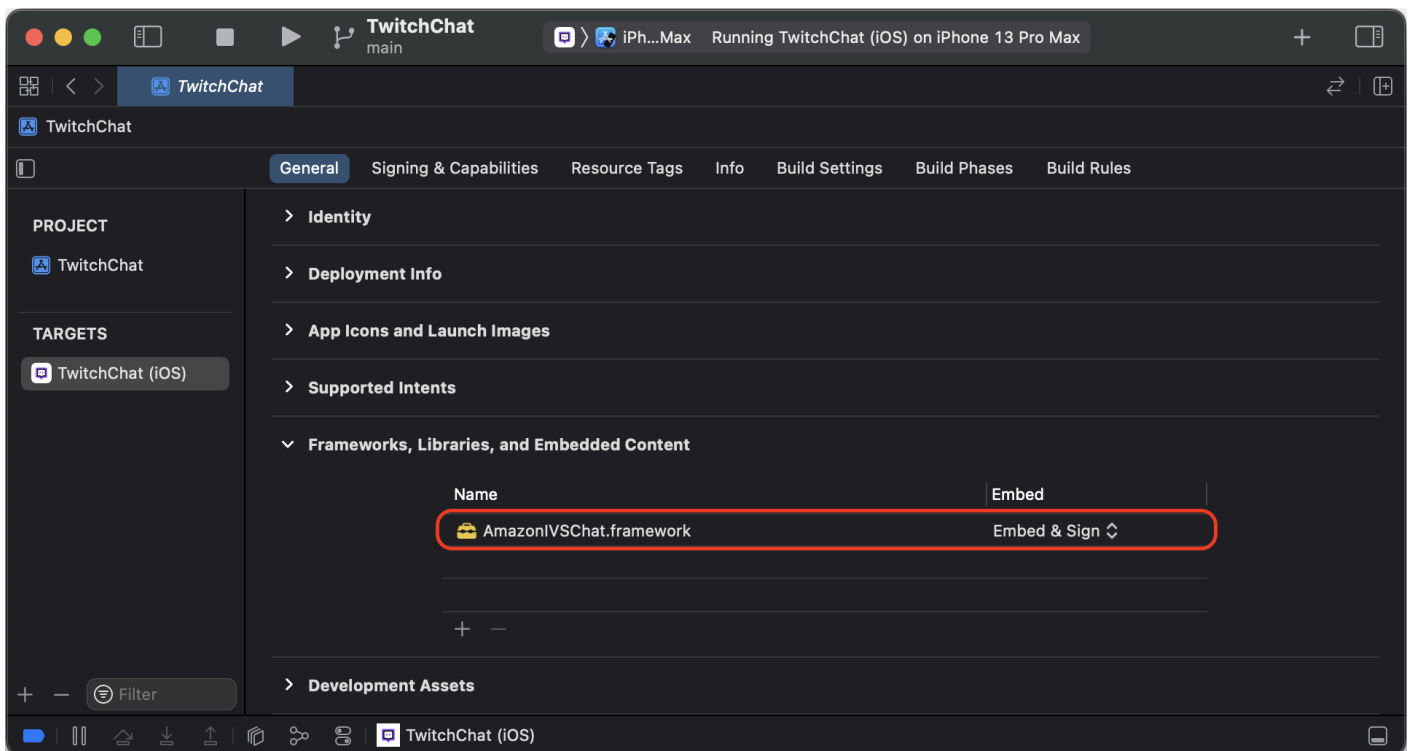
リリースは、CocoaPods から AmazonIVSChatMessaging という名前で公開されます。この依存関係を自分の Podfile に追加します。

```
pod 'AmazonIVSChat'
```

pod install を実行すると、SDK が .xcworkspace で利用できるようになります。

手動インストール

1. 次のリンクから最新バージョンをダウンロードします。 <https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip>
2. アーカイブの内容を抽出します。AmazonIVSChatMessaging.xcframework には、デバイスとシミュレータの両方の SDK が含まれています。
3. アプリケーションターゲットの [General] (全般) タブの [Frameworks, Libraries, and Embedded Content] (フレームワーク、ライブラリ、埋め込みコンテンツ) のセクションに抽出した AmazonIVSChatMessaging.xcframework をドラッグして埋め込みます。



SDK の使用

チャットルームに接続する

開始する前に、「[Amazon IVS Chat の開始方法](#)」を理解しておく必要があります。[ウェブ](#)、[Android](#)、および [iOS](#) のサンプルアプリもご覧ください。

チャットルームに接続するには、アプリがバックエンドによって提供されたチャットトークンを取得する何らかの方法が必要です。アプリケーションはおそらく、バックエンドへのネットワークリクエストを使用してチャットトークンを取得します。

このフェッチされたチャットトークンを SDK と通信するには、SDK の ChatRoom モデルで、`async` 関数、または初期化の時点で提供された ChatTokenProvider プロトコルに準拠するオブジェクトのインスタンスを提供する必要があります。これらのメソッドのいずれかによって返される値は、SDK ChatToken モデルのインスタンスである必要があります。

注意: バックエンドから取得したデータを使用して、ChatToken モデルのインスタンスを設定します。ChatToken インスタンスの初期化に必要なフィールドは、[CreateChatToken](#) レスポンスのフィールドと同じです。ChatToken モデルのインスタンスの初期化の詳細については、「[ChatToken のインスタンスの作成](#)」を参照してください。バックエンドは、アプリへの CreateChatToken リスponseでデータを提供する責任を担っていることに留意してください。バックエンドと通信してチャットトークンを生成する方法は、アプリとそのインフラストラクチャによります。

ChatToken を SDK へ提供する戦略を選んだ後、トークンプロバイダーと、接続しようとしているチャットルームを作成するためにバックエンドが使用した AWS リージョンで ChatRoom インスタンスを正常に初期化した後に `.connect()` を呼び出します。`.connect()` は次の非同期関数のスローであることに注意してください。

```
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 プロトコルに準拠

ChatRoom のイニシャライザの `tokenProvider` パラメータには、ChatTokenProvider のインスタンスを指定できます。ChatTokenProvider に準拠するオブジェクトの例を次に示します。

```
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
)
}
}
```

次に、この準拠オブジェクトのインスタンスを取得し、それを ChatRoom のイニシャライザに渡すことができます。

```
// 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()
```

Swift で非同期関数を提供する

アプリケーションのネットワークリクエストを管理するマネージャーが、既にあるとします。次のように指定します。

```
import AmazonIVSChatMessaging

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

マネージャーに別の関数を追加して、バックエンドから ChatToken を取得できます。

```
import AmazonIVSChatMessaging
```

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

次に、ChatRoom を初期化する際に、Swift でその関数への参照を使用します。

```
import AmazonIVSChatMessaging

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

ChatToken のインスタンスの作成

SDK で提供されているイニシャライザを使用して、ChatToken のインスタンスを簡単に作成できます。ChatToken のプロパティの詳細については、Token.swift にあるドキュメントを参照してください。

```
import AmazonIVSChatMessaging

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

Decodable を使用する

IVS チャット API とのインターフェイス中に、バックエンドが [CreateChatToken](#) レスポンスをフロントエンドアプリケーションにシンプルに転送することを決定した場合、Swift の Decodable プロトコルへの ChatToken の準拠を利用できます。ただし、注意点があります。

CreateChatToken レスポンスペイロードは、[インターネットタイムスタンプの ISO 8601 標準](#)を使用してフォーマットされた日付の文字列を使用します。通常 Swift では、JSONDecoder.DateDecodingStrategy.iso8601 を JSONDecoder の .dateDecodingStrategy プロパティへ値として[提供します](#)。

ただし、`CreateChatToken` は文字列で高精度の小数秒を使用しますが、これは `JSONDecoder.DateDecodingStrategy.iso8601` ではサポートされていません。

便宜上、SDK は `JSONDecoder.DateDecodingStrategy` にパブリック拡張機能を提供し、追加の `.preciseISO8601` ストラテジーを使用して、`ChatToken` のインスタンスをデコードするときに `JSONDecoder` を正常に使用できるようにします。

```
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)
```

チャットルームの接続を切断する

正常に接続した `ChatRoom` インスタンスから手動で切断するには、`room.disconnect()` を呼び出します。デフォルトでは、チャットルームは割り当てが解除されると自動的にこの関数を呼び出します。

```
import AmazonIVSChatMessaging

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

// Disconnect
room.disconnect()
```

チャットメッセージ/イベントを受信する

チャットルームでメッセージを送受信するには、`ChatRoom` のインスタンスを正常に初期化し、`room.connect()` を呼び出した後、`ChatRoomDelegate` プロトコルに準拠するオブジェクトを提供する必要があります。UIViewController を使用した典型的な例を次に示します。

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

接続が変更されたときに通知を受け取る

当然のことながら、ルームが完全に接続されるまで、ルームでメッセージを送信などのアクションは実行できません。SDK のアーキテクチャは、非同期 API を介してバックグラウンドスレッドの ChatRoom への接続を推奨しようとしています。メッセージ送信ボタンなどを無効にする何かを UI に構築する場合、SDK は Combine または ChatRoomDelegate を使用して、チャットルームの接続状態が変化したときに通知を受け取るための 2 つの戦略を提供します。これについて以下に説明します。

重要: チャットルームの接続状態は、ネットワーク接続の切断などによっても変化する可能性があります。アプリをビルトするときには、この点を考慮してください。

Combine を使用する

ChatRoom のすべてのインスタンスには、state プロパティの形式で独自の Combine パブリッシャーが付属しています。

```
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 を使用する

また、ChatRoomDelegate に準拠するオブジェクト内でオプション関数の `roomDidConnect(_:)`、`roomIsConnecting(_:)`、および `roomDidDisconnect(_:)` を使用することもできます。UIViewController を使用する例を次に示します。

```
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!")
    }
}
```

チャットルームでアクションを実行する

チャットルームで実行できる機能 (メッセージの送信、メッセージの削除、ユーザーの接続解除など) は、ユーザーごとに異なります。これらのアクションのいずれかを実行するには、接続されている ChatRoom で `perform(request:)` を呼び出し、SDK で提供されている ChatRequest オブジェクトのいずれかのインスタンスを渡します。対応しているリクエストは `Request.swift` にあります。

チャットルームで実行されるアクションの中には、バックエンドアプリケーションが `CreateChatToken` を呼び出す場合に、接続しているユーザーに特定の権限を付与する必要があるものがあります。設計上、SDK は接続しているユーザーの機能を識別できません。したがって、接続された ChatRoom のインスタンスでモデレーターアクションを実行しようとすることはできますが、最終的にそのアクションが成功するかどうかは、control-plane API が決定します。

`room.perform(request:)` を通過するすべてのアクションは、受信したモデルとリクエストオブジェクトの両方の `requestId` に一致するモデルの予期されたインスタンス (そのタイプはリクエストオブジェクト自体に関連付けられています) をルームが受信するまで待機します。リクエストに問

題がある場合、ChatRoom は常に ChatError の形でエラーをスローします。ChatError の定義は Error.swift にあります。

メッセージの送信

チャットメッセージを送信するには、SendMessageRequest インスタンスを使用してください。

```
import AmazonIVSChatMessaging

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

前述のとおり、room.perform(request:) は ChatRoom が対応する ChatMessage を受け取る と返されます。リクエストに問題がある場合 (ルームのメッセージの文字数制限を超えるなど)、代わりに ChatError のインスタンスがスローされます。その場合、次の有用な情報を UI に表示できません。

```
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)
```

```
}
```

メッセージへのメタデータの追加

[メッセージを送信する](#) 場合、関連するメタデータを追加できます。SendMessageRequest は attributes プロパティがあり、これを使用してリクエストを初期化できます。そこにアタッチしたデータは、他のユーザーがルームでそのメッセージを受信したときにメッセージにアタッチされます。

送信中のメッセージにエモートデータを添付する例を以下に示します。

```
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"
        ]
    )
)
```

SendMessageRequest で attributes を使用することは、チャット製品で複雑な機能を構築するのに非常に役立ちます。たとえば、SendMessageRequest の [String : String] 属性ディクショナリを使用して、スレッド機能を構築できます。

attributes ペイロードは非常に柔軟で強力です。これを使うと、他の方法では得られないようなメッセージについての情報を導き出すことができます。属性を使用する方が、たとえばメッセージの文字列を解析してエモートなどの情報を取得するよりもはるかに簡単です。

メッセージの削除

チャットメッセージの削除は、チャットメッセージの送信と同じです。メッセージを削除するには、ChatRoom で room.perform(request:) 関数を利用して、DeleteMessageRequest インスタンスを作成します。

受信したチャットメッセージの以前のインスタンスに簡単にアクセスするために、message.id の値を DeleteMessageRequest のイニシャライザに渡します。

必要に応じて、DeleteMessageRequest へ理由文字列を提供し、UI で表示できます。

```
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!"
    )
)
```

これはモデレーターのアクションであるため、ユーザーは実際には別のユーザーのメッセージを削除できない場合があります。Swift のスロー可能な関数メカニズムを使用すると、適切な権限のないユーザーがメッセージを削除しようとしたときに UI にエラーメッセージを表示することができます。

バックエンドがユーザーに対して CreateChatToken を呼び出す場合、接続されたチャットユーザーに対してその機能を有効にするには、capabilities フィールドに "DELETE_MESSAGE" を渡す必要があります。

適切な権限がない状態でメッセージを削除しようとした場合、スローされる機能エラーをキャッチする例を次に示します。

```
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)
}
```

他のユーザーの接続を切断する

チャットルームから他のユーザーの接続を切断するには、`room.perform(request:)` を使用します。具体的には、`DisconnectUserRequest` のインスタンスを使用してください。ChatRoom が受信するすべての `ChatMessage` には `sender` プロパティがあり、これには `DisconnectUserRequest` のインスタンスで適切に初期化する必要があるユーザー ID が含まれています。必要に応じて、接続解除リクエストの理由文字列を指定します。

```
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
    )
)
```

これはモデレーターアクションであるため、`DISCONNECT_USER` の権限がない限り別のユーザーの接続を切断することはできません。バックエンドアプリケーションが `CreateChatToken` を呼び出し `"DISCONNECT_USER"` 文字列を `capabilities` フィールドに挿入すると、権限が設定されます。

ユーザーが別のユーザーとの接続を切断する権限がない場合は、他のリクエストと同様に `room.perform(request:)` は `ChatError` のインスタンスをスローします。エラーの

`errorCode` プロパティを調べ、モデレーター権限がないためにリクエストが失敗したかどうかを判断できます。

```
import AmazonIVSChatMessaging

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

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

    print(error.errorMessage)
}
```

IVS Chat Client Messaging SDK: iOS のチュートリアル

Amazon Interactive Video (IVS) Chat Client Messaging iOS SDK は、Apple の [Swift プログラミング言語](#) を使用したプラットフォームに [IVS Chat Messaging API](#) を組み込むことができるインターフェイスを提供します。

チャット iOS SDK のチュートリアルについては、<https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/latest/tutorials/table-of-contents/> を参照してください。

IVS Chat Client Messaging SDK: JavaScript ガイド

Amazon Interactive Video (IVS) Chat Client Messaging JavaScript SDK は、ウェブブラウザを使用するプラットフォームに [Amazon IVS Chat メッセージング API](#) を組み込むことができます。

IVS Chat Client Messaging JavaScript SDK の最新バージョン: 1.0.2 ([リリースノート](#))

リファレンスドキュメント: Amazon IVS Chat Client Messaging JavaScript SDK で使用できる最も重要なメソッドについては、リファレンスドキュメント (<https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/>) を参照してください。

サンプルコード: JavaScript SDK を使用した Web 固有のデモについては、GitHub のサンプルリポジトリを参照してください。 <https://github.com/aws-samples/amazon-ivs-chat-web-demo>

概要

開始する前に、「[Amazon IVS Chat の開始方法](#)」を理解しておく必要があります。

Package の追加

次のいずれかを使用してください。

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

または

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

React Native Support

IVS Chat Client Messaging JavaScript SDK には、`crypto.getRandomValues` メソッドを使用する `uuid` 依存関係があります。このメソッドは React Native ではサポートされていないため、追加のポリフィル `react-native-get-random-value` をインストールして、`index.js` ファイルの先頭でインポートする必要があります。

```
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);
```

バックエンドのセットアップ

この統合には、[Amazon IVS Chat API](#) と通信するサーバ上のエンドポイントが必要です。サーバーから Amazon IVS API へのアクセスに[公式の AWS ライブラリ](#)を使用します。これらのライブラリはパブリックパッケージ ([node.js](#)、[java](#) や [go](#) など) から、複数の言語でアクセスできます。

Amazon IVS Chat API [CreateChatToken](#) エンドポイントと通信するサーバーエンドポイントを作成し、チャットユーザーのチャットトークンを作成します。

SDK の使用

チャットルームインスタンスを初期化する

ChatRoom クラスのインスタンスを作成します。これには、regionOrUrl (チャットルームがホストされている AWS リージョン) と tokenProvider (次のステップで作成するトークンの取得方法) を渡す必要があります。

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

トークンプロバイダー機能

バックエンドからチャットトークンをフェッチする非同期トークンプロバイダー関数を作成します。

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

この関数はパラメーターを受け入れず、チャットトークンオブジェクトを含む [Promise](#) を返す必要があります。

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

この関数は、[ChatRoom オブジェクトを初期化するため](#)に必要です。以下の、<token> および <date-time> フィールドに、バックエンドから受け取った値を入力します。

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

ChatRoom コンストラクターに `tokenProvider` を必ず渡してください。ChatRoom は、接続が中断したり、セッションが期限切れになったりする場合には、トークンをリフレッシュします。tokenProvider を使用してトークンをどこにも保存しないでください。チャットルームが代わりに処理します。

イベントを受信する

次に、チャットルームイベントにサブスクライブして、ライフサイクルイベントのほか、チャットルームで配信されるメッセージやイベントを受信します。

```
/**
 * 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: "50PsDdX18qcJ",
  *   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: "50PsDdX18qcJ",
  *   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: "AYk6xKitV40n",
  *   messageId: "R1BLTDN84zE0",
  *   reason: "Spam",
  *   sendTime: new Date("2022-10-11T12:56:41.113Z"),
  *   requestId: "b379050a-2324-497b-9604-575cb5a9c5cd",
  *   attributes: { MessageID: "R1BLTDN84zE0", Reason: "Spam" }
  * }
  */
});

/** Called when `aws:DISCONNECT_USER` system event has been received. */
const unsubscribeOnUserDisconnect = room.addListener('userDisconnect',
(disconnectUserEvent) => {
  /* Example event payload:
  * {
  *   id: "AYk6xKitV40n",
  *   userId: "R1BLTDN84zE0",
  *   reason: "Spam",
  *   sendTime: new Date("2022-10-11T12:56:41.113Z"),
  *   requestId: "b379050a-2324-497b-9604-575cb5a9c5cd",
  *   attributes: { UserId: "R1BLTDN84zE0", Reason: "Spam" }
  * }
  */
});
```

```
});
```

チャットルームに接続する

基本的な初期化の最後のステップは、WebSocket 接続を確立することによってチャットルームに接続することです。これを行うには、ルームインスタンス内の `connect()` メソッドを呼び出します。

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

SDK は、サーバーから受信したチャットトークンにエンコードされたチャットルームへの接続を確立しようとします。

`connect()` に呼び出すと、ルームは `connecting` 状態に移行して `connecting` イベントを発行します。ルームが正常に接続されると、`connected` 状態に移行して `connect` イベントを発行します。

トークンの取得時または WebSocket への接続時の問題により、接続障害が発生する可能性があります。この場合、ルームは `maxReconnectAttempts` コンストラクターパラメーターに指定された回数まで自動的に再接続を試みます。再接続の試行中、ルームは `connecting` 状態であり、追加のイベントは発生しません。試行回数以内に接続できないと、ルームは `disconnected` 状態に移行し、接続解除の理由とともに `disconnect` イベントを発行します。`disconnected` 状態では、ルームは接続を試行しなくなります。接続プロセスを開始するには、`connect()` を再度呼び出す必要があります。

チャットルームでアクションを実行する

Amazon IVS Chat Messaging SDK には、メッセージの送信、メッセージの削除、および他のユーザーの接続を切断するためのユーザアクションが用意されています。これらは、`ChatRoom` インスタンスで使用できます。ユーザアクションは、リクエストの確認または拒否を受け取ることができる `Promise` オブジェクトを返します。

メッセージの送信

このリクエストには、チャットトークンにエンコードされた `SEND_MESSAGE` キャパシティーが必要です。

send-message リクエストをトリガーする方法

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



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

リクエストの確認や拒否を取得するには、`await` から返却される `promise` を待つか、`then()` メソッドを使用します。

```
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.
}
```

メッセージの削除

このリクエストには、チャットトークンにエンコードされた `DELETE_MESSAGE` キャパシティーが必要です。

モデレーションを目的としてメッセージを削除するには、`deleteMessage()` メソッドを呼び出します。

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

リクエストの確認や拒否を取得するには、`await` から返却される `promise` を待つか、`then()` メソッドを使用します。

```
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.
}
```

他のユーザーの接続を切断する

このリクエストには、チャットトークンにエンコードされた `DISCONNECT_USER` キャパシティーが必要です。

モデレーションを目的として他のユーザーとの接続を切断するには、`disconnectUser()` メソッドを呼び出します。

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

リクエストの確認や拒否を取得するには、`await` から返却される `promise` を待つか、`then()` メソッドを使用します。

```
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.
}
```

チャットルームの接続を切断する

チャットルームへの接続を閉じるには、`room` インスタンスで `disconnect()` メソッドを呼び出します。

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

このメソッドを呼び出すと、チャットルームは基盤となっていた `WebSocket` を正常終了させます。ルームインスタンスは `disconnected` 状態へと移行し、`disconnect` の理由に `"clientDisconnect"` が設定された状態で `disconnect` イベントを発行します。

IVS Chat Client Messaging SDK: JavaScript のチュートリアルパート 1: チャットルーム

これは 2 部構成のチュートリアルの第 1 部です。JavaScript および TypeScript を使用してフル機能のアプリケーションを構築することで、Amazon IVS Chat Client Messaging JavaScript SDK を使用する際の基本を学びます。このアプリケーションは、Chatterbox と呼ばれます。

本チュートリアルは、デベロッパーとしての経験があり、Amazon IVS Chat Messagin SDK を初めて利用する人を対象としています。JavaScript や TypeScript のプログラミング言語と React ライブラリに精通していることが望まれます。

便宜上、Amazon IVS Chat Client Messaging JavaScript SDK は Chat JS SDK とします。

注: JavaScript と TypeScript のコード例が同じ内容である場合は、共通の例として示しています。

このチュートリアル最初の部分は、いくつかのセクションに分かれています。

1. [the section called “ローカル認証サーバーおよび認可サーバーのセットアップ”](#)
2. [the section called “Chatterbox プロジェクトの作成”](#)
3. [the section called “チャットルームに接続する”](#)
4. [the section called “トークンプロバイダーの作成”](#)
5. [the section called “接続の更新を確認する”](#)
6. [the section called “送信ボタンコンポーネントの作成”](#)
7. [the section called “メッセージ入力の作成”](#)
8. [the section called “次のステップ”](#)

すべての SDK ドキュメントについては、まず「Amazon IVS Chat ユーザーガイド」の「[Amazon IVS Chat Client Messaging SDK](#)」および GitHub の「[Chat Client Messaging: SDK for JavaScript Reference](#)」を参照してください。

前提条件

- JavaScript または TypeScript、および React ライブラリを事前に習得しておいてください。React に慣れていない場合は、この [Tic-Tac-Toe チュートリアル](#) の基本を学習してください。
- [IVS Chat の開始方法](#) を読んで理解してください。
- 既存の IAM ポリシーで定義されている CreateChatToken および CreateRoom 機能を持つ、AWS IAM ユーザーを作成してください。(「[IVS Chat の開始方法](#)」を参照してください。)
- このユーザーのシークレットキーまたはアクセスキーが、AWS 認証情報ファイルに保存されていることを確認してください。手順については、「[AWS CLI ユーザーガイド](#)」(特に「[設定ファイルと認証情報ファイルの設定](#)」)を参照してください。
- チャットルームを作成し、その ARN を保存してください。「[IVS Chat の開始方法](#)」を参照してください。(ARN を保存しない場合、後でコンソールまたは Chat API で参照できます。)
- NPM または Yarn パッケージマネージャーを使用して、Node.js 14+ 環境をインストールしてください。

ローカル認証サーバーおよび認可サーバーのセットアップ

バックエンドアプリケーションは、チャットルームのクライアントを認証および認可するために、Chat JS SDK に必要なチャットトークンの生成とチャットルームの作成の両方を行います。モ

バイルアプリでは、巧妙な攻撃者により AWS キーが抽出され AWS アカウントにアクセスされる可能性があるため、キーを安全に保存することはできません。そのため、独自のバックエンドを使用する必要があります。

「Amazon IVS Chat の開始方法」の「[チャットトークンを作成する](#)」を参照してください。フローチャートで示されているように、チャットトークンの作成はサーバー側のアプリケーションで行われます。つまり、サーバー側のアプリケーションからリクエストし、独自の方法でチャットトークンを生成する必要があります。

このセクションでは、バックエンドでトークンプロバイダーを作成するための基本について説明します。Express フレームワークを使用して、ローカルの AWS 環境でチャットトークンの作成を管理するライブローカルサーバーを作成します。

NPM を使用して、空の npm プロジェクトを作成します。アプリケーションを格納するディレクトリを作成し、そのディレクトリを作業ディレクトリにします。

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

npm init を使用して、アプリケーション用の package.json ファイルを作成します。

```
$ npm init
```

このコマンドでは、アプリケーションの名前やバージョンなど、いくつかの入力が求められます。ここでは、RETURN を押して、次を除くほとんどの項目をデフォルト値のままにします。

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

RETURN を押して推奨されるデフォルトのファイル名 index.js を受け入れるか、希望するメインファイル名を入力します。

必要な依存関係をインストールします。

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

aws-sdk には環境変数の設定が必要です。この変数は、ルートディレクトリにある .env という名前のファイルから自動的にロードされます。これを設定するには、.env という名前の新しいファイルを作成し、不足している設定情報を入力します。

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

次に、上記の `npm init` コマンドに入力した名前でルートディレクトリにエントリポイントファイルを作成します。ここでは、`index.js` を使用して、必要なパッケージをすべてインポートします。

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

次に、`express` の新しいインスタンスを作成します。

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

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

その後、トークンプロバイダー用に最初のエンドポイントの POST メソッドを作成できます。リクエスト本文から、必要なパラメーター (`roomId`、`userId`、`capabilities` および `sessionDurationInMinutes`) を取得します。

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

必須フィールドの検証を追加します。

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

  if (!roomIdIdentifier || !userId) {
    res.status(400).json({ error: 'Missing parameters: `roomIdIdentifier`, `userId`' });
    return;
  }
});
```

POST メソッドを作成したら、コア機能の認証および認可のため、`createChatToken` を `aws-sdk` と統合します。

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

  if (!roomIdIdentifier || !userId || !capabilities) {
    res.status(400).json({ error: 'Missing parameters: `roomIdIdentifier`, `userId`,
`capabilities`' });
    return;
  }

  ivsChat.createChatToken({ roomIdIdentifier, 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 });
    }
  });
});
```

ファイルの最後に、`express` アプリのポートリスナーを追加します。

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

これで、プロジェクトのルートから次のコマンドを使用してサーバーを実行できます。

```
$ node index.js
```

ヒント: このサーバーは、`https://localhost:3000` で URL のリクエストを受け付けます。

Chatterbox プロジェクトの作成

まず、`chatterbox` という名前の React プロジェクトを作成します。次のコマンドを実行します。

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

Chat Client Messaging JS SDK は、[Node Package Manager](#) または [Yarn Package Manager](#) を使用して統合できます。

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

チャットルームに接続する

ここでは、`ChatRoom` を作成し、非同期メソッドを使用して接続します。Chat JS SDK へのユーザーの接続は、`ChatRoom` クラスで管理されます。チャットルームに正常に接続するには、React アプリケーション内に `ChatToken` のインスタンスを提供する必要があります。

デフォルトの `chatterbox` プロジェクトで作成された `App` ファイルに移動し、2 つの `<div>` タグの間にあるものをすべて削除します。事前入力されているコードは必要ありません。この時点で、`App` はほとんど空です。

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

import * as React from 'react';

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

新しい `ChatRoom` インスタンスを作成し、`useState` フックを使用してそのインスタンスをステートに渡します。`regionOrUrl` (チャットルームがホストされている AWS リージョン)

と `tokenProvider` (後続のステップで作成されるバックエンドの認証および認可フローに使用されます) が必要です。

重要: [Amazon IVS Chat の開始方法](#) でチャットルームを作成したときと同じ AWS リージョンを使用する必要があります。API は AWS リージョナルサービスです。サポートされているリージョンと Amazon IVS Chat HTTPS サービスエンドポイントのリストについては、[Amazon IVS Chat リージョン](#) のページを参照してください。

```
// 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>;
}
```

トークンプロバイダーの作成

次のステップとして、`ChatRoom` コンストラクターに必要なパラメータなしの `tokenProvider` 関数を作成する必要があります。まず、[the section called “ローカル認証サーバーおよび認可サーバーのセットアップ”](#) で設定したバックエンドアプリケーションに POST リクエストを送信する `fetchChatToken` 関数を作成します。チャットトークンには、SDK がチャットルームへの接続を正常に確立するために必要な情報が含まれています。Chat API では、ユーザーの ID、チャットルーム内の機能、およびセッション時間を検証する安全な方法としてこれらのトークンを使用します。

プロジェクトナビゲーターで、`fetchChatToken` という名前の新しい TypeScript または JavaScript ファイルを作成します。backend アプリケーションへのフェッチリクエストを作成し、レスポンスから `ChatToken` オブジェクトを返します。チャットトークンの作成に必要なリクエスト本文のプロパティを追加します。[Amazon リソースネーム \(ARN\)](#) に定義されているルールを使用します。これらのプロパティは [CreateChatToken エンドポイント](#) で説明されています。

注: ここで使用する URL は、バックエンドアプリケーションを実行したときにローカルサーバーが作成した URL と同じものです。

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,
      roomId: 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,
      roomId: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
  });

  const token = await response.json();

  return {
    ...token,
    sessionExpirationTime: new Date(token.sessionExpirationTime),
    tokenExpirationTime: new Date(token.tokenExpirationTime),
  };
}
```

接続の更新を確認する

チャットアプリを作る上では、チャットルームの接続状態の変化に対応することが重要です。まずは関連イベントをサブスクライブします。

```
// 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>;
}
```

次に、接続状態を読み取る機能を提供する必要があります。useState フックを使用して App でローカルステートを作成し、各リスナー内で接続状態を設定します。

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>;
}
```

接続状態をサブスクライブしたらそれを表示し、`useEffect` フック内の `room.connect` メソッドを使用してチャットルームに接続します。

```
// 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>
);

// ...
```

チャットルームへの接続が正常に実装されました。

送信ボタンコンポーネントの作成

このセクションでは、接続状態ごとに異なるデザインの送信ボタンを作成します。送信ボタンを使用すると、チャットルームでのメッセージの送信が容易になります。また、接続が切断されたりチャットセッションが期限切れになった場合に、メッセージを送信できるかどうか、いつ送信できるかを視覚的に示す役割もあります。

まず、Chatterbox プロジェクトの `src` ディレクトリに新しいファイルを作成し、`SendButton` と名前を付けます。次に、チャットアプリケーションのボタンを表示するコンポーネントを作成します。`SendButton` をエクスポートし、`App` にインポートします。空の `<div></div>` に、`<SendButton />` を追加します。

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>
);
```

次に、App で `onMessageSend` という名前の関数を定義して `SendButton` `onPress` プロパティに渡します。 `isSendDisabled` という名前の別の変数 (ルームが接続されていないときにメッセージが送信されないようにします) を定義し、それを `SendButton` `disabled` プロパティに渡します。

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

// ...

const onMessageSend = () => {};

const isSendDisabled = connectionState !== 'connected';

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

// ...
```

メッセージ入力の作成

Chatterbox のメッセージバーは、チャットルームにメッセージを送信するときに操作するコンポーネントです。通常、メッセージを作成するためのテキスト入力とメッセージを送信するためのボタンが含まれています。

`MessageInput` コンポーネントを作成するには、まず `src` ディレクトリに新しいファイルを作成し、`MessageInput` と名前を付けます。次に、チャットアプリケーションでの入力を表示する制御された入力コンポーネントを作成します。`MessageInput` をエクスポートして、App (`<SendButton />` の上) にインポートします。

useState フックを使用して、messageToSend という新しいステートを作成します。空の文字列がデフォルト値です。アプリケーションの本文で、messageToSend を MessageInput の value に渡し、setMessageToSend を onMessageChange プロパティに渡します。

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>
);
```

次のステップ

Chatterbox のメッセージバーが作成できました。この JavaScript のチュートリアル第二部分 2、[Messages and Events](#) (メッセージとイベント) に進んでください。

IVS Chat Client Messaging SDK: JavaScript チュートリアルパート 2: メッセージとイベント

本チュートリアルのパート 2 (最後のパート) は、複数のセクションに分かれています。

1. [the section called “チャットメッセージイベントへのサブスクライブ”](#)
2. [the section called “受信済みメッセージの表示”](#)
 - a. [the section called “メッセージコンポーネントの作成”](#)
 - b. [the section called “現在のユーザーにより送信されたメッセージの認識”](#)
 - c. [the section called “メッセージリストコンポーネントの作成”](#)
 - d. [the section called “チャットメッセージのリストのレンダリング”](#)
3. [the section called “チャットルームでアクションを実行する”](#)
 - a. [the section called “メッセージの送信”](#)
 - b. [the section called “メッセージの削除”](#)
4. [the section called “次のステップ”](#)

注: JavaScript と TypeScript のコード例が同じ内容である場合は、共通の例として示しています。

すべての SDK ドキュメントについては、まず「Amazon IVS Chat ユーザーガイド」の「[Amazon IVS Chat Client Messaging SDK](#)」および GitHub の「[Chat Client Messaging: SDK for JavaScript Reference](#)」を参照してください。

前提条件

このチュートリアルのパート 1 である「[チャットルーム](#)」を完了してから、このパートに進んでください。

チャットメッセージイベントへのサブスクライブ

チャットルームでイベントが発生した際、ChatRoom インスタンスはイベントを使用して通信を行います。チャットによるエクスペリエンスを開始するには、ルーム内で、そこに接続しているユーザーに対し、他のユーザーからメッセージを送信されたことを知らせる必要があります。

このためにユーザーは、チャットメッセージイベントをサブスクライブします。メッセージ/イベントごとに、作成済みのメッセージリストを更新する方法については、この後半で説明します。

App の useEffect フック内で、すべてのメッセージイベントにサブスクライブします。

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

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

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

受信済みメッセージの表示

メッセージの受信は、チャットエクスペリエンスにとって中心的な要素です。Chat JS SDKを使用してコードを作成することで、チャットルームに接続している他のユーザーからのイベントを簡単に受信できます。

後半部分で、ここで作成したコンポーネントを活用してチャットルームでアクションを実行する方法を説明します。

App で、messages という名前の ChatMessage 配列型を使用して、messages という名前の状態を定義します。

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([]);  
  
  //...  
}
```

次に、message リスナー関数内で、messages 配列に message を追加します。

```
// App.jsx / App.tsx  
  
// ...  
  
const unsubscribeOnMessageReceived = room.addListener('message', (message) => {  
  setMessages((msgs) => [...msgs, message]);  
});  
  
// ...
```

以下に、受信したメッセージを表示するタスクを順を追って説明します。

1. [the section called “メッセージコンポーネントの作成”](#)
2. [the section called “現在のユーザーにより送信されたメッセージの認識”](#)
3. [the section called “メッセージリストコンポーネントの作成”](#)
4. [the section called “チャットメッセージのリストのレンダリング”](#)

メッセージコンポーネントの作成

Message コンポーネントにより、チャットルームで受信したメッセージ内容のレンダリングが行われます。このセクションでは、個々のチャットメッセージを App 内でレンダリングするための、メッセージコンポーネントを作成します。

src ディレクトリで Message という名前の新しいファイルを作成します。このコンポーネントに ChatMessage 型を渡し、さらに ChatMessage プロパティからの content 文字列を渡すことで、チャットルームのメッセージリスナーから受信したメッセージテキストを表示します。プロジェクトナビゲーターで、Message に移動します。

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>
  );
};
```

ヒント: このコンポーネントを使用して、メッセージ行に表示するさまざまなプロパティ (アバター URL、ユーザー名、メッセージ送信時のタイムスタンプなど) を保存できます。

現在のユーザーにより送信されたメッセージの認識

現在のユーザーから送信されたメッセージを認識するために、そのユーザーの `userId` を格納する React コンテキストを作成するように、コードを変更します。

src ディレクトリに、UserContext という名前で新しいファイルを作成します。

TypeScript

```
// UserContext.tsx

import React, { ReactNode, useState, useContext, createContext } from 'react';

type UserContextType = {
  userId: string;
  setUserId: (userId: string) => void;
};

const UserContext = createContext<UserContextType | undefined>(undefined);

export const useUserContext = () => {
  const context = useContext(UserContext);

  if (context === undefined) {
    throw new Error('useUserContext must be within UserProvider');
  }

  return context;
};

type UserProviderType = {
  children: ReactNode;
}

export const UserProvider = ({ children }: UserProviderType) => {
  const [userId, setUserId] = useState('Mike');

  return <UserContext.Provider value={{ userId, setUserId }}>{children}</
UserContext.Provider>;
};
```

JavaScript

```
// UserContext.jsx

import React, { useState, useContext, createContext } from 'react';

const UserContext = createContext(undefined);
```

```
export const useUserContext = () => {
  const context = useContext(UserContext);

  if (context === undefined) {
    throw new Error('useUserContext must be within UserProvider');
  }

  return context;
};

export const UserProvider = ({ children }) => {
  const [userId, setUserId] = useState('Mike');

  return <UserContext.Provider value={{ userId, setUserId }}>{children}</
UserContext.Provider>;
};
```

注: ここでは、useState フックを使用して userId 値を保存しています。将来的には、ユーザーコンテキストの変更や、ログインのためにも setUserId を使用できます。

次に、tokenProvider に渡された最初のパラメータの userId を、先に作成済みのコンテキストにより置き換えます。

```
// 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 コンポーネント内では、既に作成済みの `UserContext` を使用して `isMine` 変数を宣言し、送信者の `userId` とコンテキストからの `userId` を照合し、現在のユーザーにさまざまなスタイルのメッセージを適用します。

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>
);
};
```

メッセージリストコンポーネントの作成

MessageList コンポーネントは、チャットルームの会話を時系列で表示する役割を果たします。MessageList ファイルは、すべてのメッセージを格納するコンテナであり、Message は MessageList の中の 1 つの行です。

src ディレクトリで MessageList という名前の新しいファイルを作成します。ChatMessage 型の配列の messages で Props を定義します。本文内で messages プロパティをマッピングし、Message コンポーネントに Props を渡します。

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>
  );
};
```

チャットメッセージのリストのレンダリング

次に、メインの App のコンポーネント内に、新しい MessageList を取り込みます。

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

// ...
```

以上で、チャットルームで受信したメッセージを App でレンダリングする準備ができました。作成したコンポーネントを活用して、チャットルーム内のアクションを実行する方法について、この後で説明します。

チャットルームでアクションを実行する

チャットルームに対する主な操作例としては、チャットルーム内でのメッセージの送信や、モデレーターのアクション実行などが挙げられます。ここでは、さまざまな ChatRequest オブジェクトを使用し、メッセージの送信や削除、他のユーザーの接続の切断といった一般的なアクションを Chatterbox 内で実行する方法を説明します。

チャットルームのアクションは、すべて共通のパターンに従っており、それぞれ対応するリクエストオブジェクトを持っています。各リクエストは、それが確認された際に、対応するレスポンスオブジェクトを受け取ります。

チャットトークンの作成時に適切な権限が付与されているユーザーであれば、チャットルームで実行できるリクエストを確認するためのアクションを、対応するリクエストオブジェクトを使用して正常に実行できます。

以下で、[メッセージを送信](#)する方法と、[メッセージを削除](#)する方法について説明します。

メッセージの送信

SendMessageRequest クラスにより、チャットルームでのメッセージ送信が有効化されます。ここでは、「[メッセージ入力の作成](#)」(このチュートリアルパート 1) で作成したコンポーネントを使用して、メッセージリクエストを送信するように App を変更します。

まず、isSending という名前を付けた新しいブール値のプロパティを、useState フックで定義します。この新しいプロパティで isSendDisabled 定数を使用すると、button HTML 要素の無効状態を切り替えることができます。SendButton のイベントハンドラーで messageToSend の値をクリアし、isSending を true に設定します。

API 呼び出しはこのボタンにより実行されるため、isSending ブール値を追加することで、リクエストが完了するまで SendButton でのユーザー操作を無効にし、複数の API 呼び出しが同時に発生するのを防ぐことができます。

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

// ...

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

```
// ...

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

// ...

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

// ...
```

新しい `SendMessageRequest` インスタンスを作成してリクエストを準備し、メッセージの内容をコンストラクターに渡してます。 `isSending` および `messageToSend` の状態を設定したら、 `sendMessage` メソッドを呼び出してリクエストをチャットルームに送信します。最後に、リクエストの確認または拒否を受け取った時点で、 `isSending` フラグをクリアします。

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);
  }
};

// ...
```

ここで、Chatterbox を実行してみます。MessageInput を使用してメッセージの下書きを作成し、SendButton をタップしてメッセージを送信します。先に作成済みの MessageList 内にレンダリングされた、送信済みメッセージが表示されるはずですが。

メッセージの削除

チャットルームからメッセージを削除するには、適切な権限が必要です。これらの権限は、チャットルームへの認証時に使用するチャットトークンの初期化中に付与されます。このセクションでは、「[ローカル認証/承認サーバーのセットアップ](#)」(このチュートリアルパート 1)にある ServerApp により、モデレーターの機能を指定しています。この処理は、「[トークンプロバイダーの作成](#)」(同じくパート 1)で作成した tokenProvider オブジェクトを使用して、アプリ内で実行されます。

ここでは、Message を変更し、メッセージを削除する関数を追加します。

まず、App.tsx を開き DELETE_MESSAGE の機能を追加します。(capabilities は tokenProvider 関数の 2 番目のパラメータです。)

注: これにより、生成されたチャットトークンに関連付けられているユーザーがチャットルーム内のメッセージを削除できることが、ServerApp から IVS Chat API に対し伝達されます。実際には、サーバーアプリのインフラストラクチャでユーザーの権限を管理するためのバックエンドロジックは、さらに複雑なものになるでしょう。

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']),
  }),
);

// ...
```

以降のステップでは、削除ボタンを表示するために、Message を変更していきます。

Message を開き、初期値として false を指定した useState フックを使用して、isDeleting の名前で新しいブール値の状態を定義します。この状態を使用して、isDeleting の現在の状態に応じた内容になるように Button を更新します。isDeleting が true の場合はボタンを無効にします。これにより、2つのメッセージ削除リクエストが、同時に生成されるのを防ぐことができます。

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>  
);  
};
```

文字列をパラメータの1つとして受け取り Promise を返す新しい関数を、onDelete の名前で定義します。Button のアクションクロージャの本文で setIsDeleting を使用し、onDelete に対する呼び出しの前後で isDeleting のブール値を切り替えます。文字列パラメータには、コンポーネントのメッセージ ID を渡します。

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>  
  );  
};
```

```
);  
};
```

次に、Message コンポーネントに加えられた最新の変更を反映するように MessageList を更新します。

MessageList を開き、文字列をパラメータとして受け取り Promise を返す新しい関数を、onDelete の名前で定義します。Message を更新し、Message のプロパティを介してこれを渡します。新しいクロージャール内の文字列パラメータは、削除するメッセージの ID になります。これは Message から渡されたものです。

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

次に、App を更新して、MessageList に対する最新の変更を反映させます。

App の中で onDeleteMessage という名前の関数を定義し、それを MessageList onDelete プロパティに渡します。

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>
);

// ...
```

DeleteMessageRequest のインスタンスを新しく作成して、関連するメッセージ ID をコンストラクタのパラメータに渡し、リクエストを用意します。用意したリクエストを受け入れるために deleteMessage を呼び出します。

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);
  };

  // ...
```

次に、messages の状態を更新し、削除済みメッセージが消去された新しいメッセージリストを反映させます。

useEffect フックでは messageDelete イベントを監視し、message パラメーターと一致する ID を持つメッセージを削除して、messages 状態配列を更新します。

注: messageDelete イベントは、現在のユーザー、またはチャットルーム内の他のユーザーによってメッセージが削除された場合に発生します。これを、deleteMessage リクエストの後ではなくイベントハンドラーで処理することで、メッセージ削除の処理をまとめることができます。

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

この段階で、チャットアプリのチャットルームからユーザーを削除することが可能になりました。

次のステップ

実験として、他のユーザーの接続切断など追加のアクションも、ルームに実装してみてください。

IVS Chat Client Messaging SDK: React Native のチュートリアル

パート 1: Chat Rooms (チャットルーム)

これは 2 部構成のチュートリアルの第 1 部です。React Native を使用してフル機能のアプリケーションを構築し、Amazon IVS Chat Client Messaging JavaScript SDK を使用する際の基本を学びます。このアプリケーションは、Chatterbox と呼ばれます。

本チュートリアルは、デベロッパーとしての経験があり、Amazon IVS Chat Messaging SDK を初めて利用する人を対象としています。TypeScript または JavaScript のプログラミング言語と React Native ライブラリに精通していることが望まれます。

便宜上、Amazon IVS Chat Client Messaging JavaScript SDK は Chat JS SDK とします。

注: JavaScript と TypeScript のコード例が同じ内容である場合は、共通の例として示しています。

このチュートリアルの最初の部分は、いくつかのセクションに分かれています。

1. [the section called “ローカル認証サーバーおよび認可サーバーのセットアップ”](#)
2. [the section called “Chatterbox プロジェクトの作成”](#)
3. [the section called “チャットルームに接続する”](#)
4. [the section called “トークンプロバイダーの作成”](#)
5. [the section called “接続の更新を確認する”](#)
6. [the section called “送信ボタンコンポーネントの作成”](#)
7. [the section called “メッセージ入力の作成”](#)
8. [the section called “次のステップ”](#)

前提条件

- TypeScript または JavaScript、および React Native ライブラリに精通しておいてください。React Native に慣れていない場合は、「[Intro to React Native](#)」(React Native の概要) で基本を学んでください。
- [IVS Chat の開始方法](#) を読んで理解してください。
- 既存の IAM ポリシーで定義されている CreateChatToken および CreateRoom 機能を持つ、AWS IAM ユーザーを作成してください。(「[IVS Chat の開始方法](#)」を参照してください。)

- このユーザーのシークレットキーまたはアクセスキーが、AWS 認証情報ファイルに保存されていることを確認してください。手順については、「[AWS CLI ユーザーガイド](#)」(特に「[設定ファイルと認証情報ファイルの設定](#)」)を参照してください。
- チャットルームを作成し、その ARN を保存してください。「[IVS Chat の開始方法](#)」を参照してください。(ARN を保存しない場合、後でコンソールまたは Chat API で参照できます。)
- NPM または Yarn パッケージマネージャーを使用して、Node.js 14+ 環境をインストールしてください。

ローカル認証サーバーおよび認可サーバーのセットアップ

バックエンドアプリケーションは、チャットルームのクライアントを認証および認可するために、Chat JS SDK に必要なチャットトークンの生成とチャットルームの作成の両方を行います。モバイルアプリでは、巧妙な攻撃者により AWS キーが抽出され AWS アカウントにアクセスされる可能性があるため、キーを安全に保存することはできません。そのため、独自のバックエンドを使用する必要があります。

「Amazon IVS Chat の開始方法」の「[チャットトークンを作成する](#)」を参照してください。フローチャートで示されているように、チャットトークンの作成はサーバー側のアプリケーションで行われます。つまり、サーバー側のアプリケーションからリクエストし、独自の方法でチャットトークンを生成する必要があります。

このセクションでは、バックエンドでトークンプロバイダーを作成するための基本について説明します。Express フレームワークを使用して、ローカルの AWS 環境でチャットトークンの作成を管理するライブローカルサーバーを作成します。

NPM を使用して、空の npm プロジェクトを作成します。アプリケーションを格納するディレクトリを作成し、そのディレクトリを作業ディレクトリにします。

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

npm init を使用して、アプリケーション用の package.json ファイルを作成します。

```
$ npm init
```

このコマンドでは、アプリケーションの名前やバージョンなど、いくつかの入力が求められます。ここでは、RETURN を押して、次を除くほとんどの項目をデフォルト値のままにします。

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


RETURN を押して推奨されるデフォルトのファイル名 `index.js` を受け入れるか、希望するメインファイル名を入力します。

必要な依存関係をインストールします。

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

`aws-sdk` には環境変数の設定が必要です。この変数は、ルートディレクトリにある `.env` という名前のファイルから自動的にロードされます。これを設定するには、`.env` という名前の新しいファイルを作成し、不足している設定情報を入力します。

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

次に、上記の `npm init` コマンドに入力した名前でルートディレクトリにエントリポイントファイルを作成します。ここでは、`index.js` を使用して、必要なパッケージをすべてインポートします。

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

次に、`express` の新しいインスタンスを作成します。

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

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

その後、トークンプロバイダー用に最初のエンドポイントの POST メソッドを作成できません。リクエスト本文から、必要なパラメータ (roomId、userId、capabilities、および sessionDurationInMinutes) を取得します。

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

必須フィールドの検証を追加します。

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

  if (!roomIdIdentifier || !userId) {
    res.status(400).json({ error: 'Missing parameters: `roomIdIdentifier`, `userId`' });
    return;
  }
});
```

POST メソッドを作成したら、コア機能の認証および認可のため、createChatToken を aws-sdk と統合します。

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

  if (!roomIdIdentifier || !userId || !capabilities) {
    res.status(400).json({ error: 'Missing parameters: `roomIdIdentifier`, `userId`,
`capabilities`' });
    return;
  }

  ivsChat.createChatToken({ roomIdIdentifier, 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 });
  }
});
});
```

ファイルの最後に、express アプリのポートリスナーを追加します。

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

これで、プロジェクトのルートから次のコマンドを使用してサーバーを実行できます。

```
$ node index.js
```

ヒント: このサーバーは、https://localhost:3000 で URL のリクエストを受け付けます。

Chatterbox プロジェクトの作成

まず、chatterbox という名前の React Native プロジェクトを作成します。次のコマンドを実行します。

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

または、TypeScript テンプレートを使用してエキスポプロジェクトを作成します。

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

Chat Client Messaging JS SDK は、[Node Package Manager](#) または [Yarn Package Manager](#) を使用して統合できます。

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

チャットルームに接続する

ここでは、ChatRoom を作成し、非同期メソッドを使用して接続します。Chat JS SDK へのユーザーの接続は、ChatRoom クラスで管理されます。チャットルームに正常に接続するには、React アプリケーション内に ChatToken のインスタンスを提供する必要があります。

デフォルトの chatterbox プロジェクトで作成された App ファイルに移動し、機能コンポーネントが返すすべてのものを削除します。事前入力されているコードは必要ありません。この時点で、App はほとんど空です。

TypeScript/JavaScript:

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

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

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

新しい ChatRoom インスタンスを作成し、useState フックを使用してそのインスタンスをステートに渡します。regionOrUrl (チャットルームがホストされている AWS リージョン) と tokenProvider (後続のステップで作成されるバックエンドの認証および認可フローに使用されます) が必要です。

重要: [Amazon IVS Chat の開始方法](#) でチャットルームを作成したときと同じ AWS リージョンを使用する必要があります。API は AWS リージョナルサービスです。サポートされているリージョンと Amazon IVS Chat HTTPS サービスエンドポイントのリストについては、[Amazon IVS Chat リージョン](#) のページを参照してください。

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>;
}
```

トークンプロバイダーの作成

次のステップとして、ChatRoom コンストラクターに必要なパラメータなしの tokenProvider 関数を作成する必要があります。まず、[the section called “ローカル認証サーバーおよび認可サーバーのセットアップ”](#) で設定したバックエンドアプリケーションに POST リクエストを送信する fetchChatToken 関数を作成します。チャットトークンには、SDK がチャットルームへの接続を正常に確立するために必要な情報が含まれています。Chat API では、ユーザーの ID、チャットルーム内の機能、およびセッション時間を検証する安全な方法としてこれらのトークンを使用します。

プロジェクトナビゲーターで、fetchChatToken という名前の新しい TypeScript または JavaScript ファイルを作成します。backend アプリケーションへのフェッチリクエストを作成し、レスポンスから ChatToken オブジェクトを返します。チャットトークンの作成に必要なリクエスト本文のプロパティを追加します。[Amazon リソースネーム \(ARN\)](#) に定義されているルールを使用します。これらのプロパティは [CreateChatToken エンドポイント](#) で説明されています。

注: ここで使用する URL は、バックエンドアプリケーションを実行したときにローカルサーバーが作成した URL と同じものです。

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,
    roomIdIdentifier: 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),
};
}
```

接続の更新を確認する

チャットアプリを作る上では、チャットルームの接続状態の変化に対応することが重要です。まずは関連イベントをサブスクライブします。

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>;
}
```

次に、接続状態を読み取る機能を提供する必要があります。useState フックを使用して App でローカルステートを作成し、各リスナー内で接続状態を設定します。

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>;
}
```

接続状態をサブスクライブしたらそれを表示し、`useEffect` フック内の `room.connect` メソッドを使用してチャットルームに接続します。

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();
  };
}, [room]);
```

```
    unsubscribeOnDisconnected();
  };
}, [room]));

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
  </SafeAreaView>
);

const styles = StyleSheet.create({
  root: {
    flex: 1,
  }
});

// ...
```

チャットルームへの接続が正常に実装されました。

送信ボタンコンポーネントの作成

このセクションでは、接続状態ごとに異なるデザインの送信ボタンを作成します。送信ボタンを使用すると、チャットルームでのメッセージの送信が容易になります。また、接続が切断されたりチャットセッションが期限切れになった場合に、メッセージを送信できるかどうか、いつ送信できるかを視覚的に示す役割もあります。

まず、Chatterbox プロジェクトの `src` ディレクトリに新しいファイルを作成し、`SendButton` と名前を付けます。次に、チャットアプリケーションのボタンを表示するコンポーネントを作成します。`SendButton` をエクスポートし、`App` にインポートします。空の `<View></View>` に、`<SendButton />` を追加します。

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',
    alignItems: '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>
);
```

次に、App で `onMessageSend` という名前の関数を定義して `SendButton` `onPress` プロパティに渡します。 `isSendDisabled` という名前の別の変数 (ルームが接続されていないときにメッセージが送信されないようにします) を定義し、それを `SendButton` `disabled` プロパティに渡します。

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>
);

// ...
```

メッセージ入力の作成

Chatterbox のメッセージバーは、チャットルームにメッセージを送信するときに操作するコンポーネントです。通常、メッセージを作成するためのテキスト入力とメッセージを送信するためのボタンが含まれています。

MessageInput コンポーネントを作成するには、まず src ディレクトリに新しいファイルを作成し、MessageInput と名前を付けます。その後、チャットアプリケーションでの入力を表示する入力コンポーネントを作成します。MessageInput をエクスポートして、App (<SendButton /> の上) にインポートします。

useState フックを使用して、messageToSend という新しいステートを作成します。空の文字列がデフォルト値です。アプリケーションの本文で、messageToSend を MessageInput の value に渡し、setMessageToSend を onMessageChange プロパティに渡します。

TypeScript

```
// MessageInput.tsx

import * as React from 'react';

interface Props {
  value?: string;
  onChange?: (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',
    }
  });
});
```

次のステップ

Chatterbox のメッセージバーの作成が完了したので、この React Native のチュートリアルパート 2「[Messages and Events](#)」(メッセージとイベント)に進んでください。

IVS Chat Client Messaging SDK: React Native チュートリアルパート 2: メッセージとイベント

本チュートリアルパート 2 (最後のパート) は、複数のセクションに分かれています。

1. [the section called “チャットメッセージイベントへのサブスクリブ”](#)

2. [the section called “受信済みメッセージの表示”](#)
 - a. [the section called “メッセージコンポーネントの作成”](#)
 - b. [the section called “現在のユーザーにより送信されたメッセージの認識”](#)
 - c. [the section called “チャットメッセージのリストのレンダリング”](#)
3. [the section called “チャットルームでアクションを実行する”](#)
 - a. [the section called “メッセージの送信”](#)
 - b. [the section called “メッセージの削除”](#)
4. [the section called “次のステップ”](#)

注: JavaScript と TypeScript のコード例が同じ内容である場合は、共通の例として示しています。

前提条件

このチュートリアルパート 1 である「[チャットルーム](#)」を完了してから、このパートに進んでください。

チャットメッセージイベントへのサブスクライブ

チャットルームでイベントが発生した際、ChatRoom インスタンスはイベントを使用して通信を行います。チャットによるエクスペリエンスを開始するには、ルーム内で、そこに接続しているユーザーに対し、他のユーザーからメッセージを送信されたことを知らせる必要があります。

このためにユーザーは、チャットメッセージイベントをサブスクライブします。メッセージ/イベントごとに、作成済みのメッセージリストを更新する方法については、この後半で説明します。

App の `useEffect` フック内で、すべてのメッセージイベントにサブスクライブします。

TypeScript/JavaScript:

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

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

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

```
}, []);
```

受信済みメッセージの表示

メッセージの受信は、チャットエクスペリエンスにとって中心的な要素です。Chat JS SDKを使用してコードを作成することで、チャットルームに接続している他のユーザーからのイベントを簡単に受信できます。

後半部分で、ここで作成したコンポーネントを活用してチャットルームでアクションを実行する方法を説明します。

App で、messages という名前の ChatMessage 配列型を使用して、messages という名前の状態を定義します。

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([]);

  //...
}
```

次に、message リスナー関数内で、messages 配列に message を追加します。

TypeScript/JavaScript:

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

// ...

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

// ...
```

以下に、受信したメッセージを表示するタスクを順を追って説明します。

1. [the section called “メッセージコンポーネントの作成”](#)
2. [the section called “現在のユーザーにより送信されたメッセージの認識”](#)
3. [the section called “チャットメッセージのリストのレンダリング”](#)

メッセージコンポーネントの作成

Message コンポーネントにより、チャットルームで受信したメッセージ内容のレンダリングが行われます。このセクションでは、個々のチャットメッセージを App 内でレンダリングするための、メッセージコンポーネントを作成します。

src ディレクトリで Message という名前の新しいファイルを作成します。このコンポーネントに ChatMessage 型を渡し、さらに ChatMessage プロパティからの content 文字列を渡すことで、チャットルームのメッセージリスナーから受信したメッセージテキストを表示します。プロジェクトナビゲーターで、Message に移動します。

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,
  },
});
```

ヒント: このコンポーネントを使用して、メッセージ行に表示するさまざまなプロパティ (アバター URL、ユーザー名、メッセージ送信時のタイムスタンプなど) を保存できます。

現在のユーザーにより送信されたメッセージの認識

現在のユーザーから送信されたメッセージを認識するために、そのユーザーの `userId` を格納する React コンテキストを作成するように、コードを変更します。

`src` ディレクトリに、`UserContext` という名前で新しいファイルを作成します。

TypeScript

```
// UserContext.tsx

import React from 'react';

const UserContext = React.createContext<string | undefined>(undefined);

export const useUserContext = () => {
  const context = React.useContext(UserContext);

  if (context === undefined) {
    throw new Error('useUserContext must be within UserProvider');
  }
}
```

```
    return context;
  };

export const UserProvider = UserContext.Provider;
```

JavaScript

```
// UserContext.jsx

import React from 'react';

const UserContext = React.createContext(undefined);

export const useUserContext = () => {
  const context = React.useContext(UserContext);

  if (context === undefined) {
    throw new Error('useUserContext must be within UserProvider');
  }

  return context;
};

export const UserProvider = UserContext.Provider;
```

注: ここでは、`useState` フックを使用して `userId` 値を保存しています。将来的には、ユーザーコンテキストの変更や、ログインのためにも `setUserId` を使用できます。

次に、`tokenProvider` に渡された最初のパラメータの `userId` を、先に作成済みのコンテキストにより置き換えます。以下に示すように、トークンプロバイダーに `SEND_MESSAGE` 権限を追加してください。これは、メッセージを送信するために必要です。

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 コンポーネント内では、既に作成済みの `UserContext` を使用して `isMine` 変数を宣言し、送信者の `userId` とコンテキストからの `userId` を照合し、現在のユーザーにさまざまなスタイルのメッセージを適用します。

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',
},
});
```

チャットメッセージのリストのレンダリング

ここで、FlatList および Message コンポーネントを使用してメッセージを一覧表示します。

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>
);

// ...
```

以上で、チャットルームで受信したメッセージを App でレンダリングする準備ができました。作成したコンポーネントを活用して、チャットルーム内のアクションを実行する方法について、この後で説明します。

チャットルームでアクションを実行する

チャットルームに対する主な操作例としては、メッセージの送信や、モデレーターのアクション実行などが挙げられます。ここでは、さまざまな ChatRequest オブジェクトを使用し、メッセージの送信や削除、他のユーザーの接続の切断といった一般的なアクションを Chatterbox 内で実行する方法を説明します。

チャットルームのアクションは、すべて共通のパターンに従っており、それぞれ対応するリクエストオブジェクトを持っています。各リクエストは、それが確認された際に、対応するレスポンスオブジェクトを受け取ります。

チャットトークンの作成時に適切な権限が付与されているユーザーであれば、チャットルームで実行できるリクエストを確認するためのアクションを、対応するリクエストオブジェクトを使用して正常に実行できます。

以下で、[メッセージを送信](#)する方法と、[メッセージを削除](#)する方法について説明します。

メッセージの送信

SendMessageRequest クラスにより、チャットルームでのメッセージ送信が有効化されます。ここでは、「[メッセージ入力の作成](#)」(このチュートリアルパート 1) で作成したコンポーネントを使用して、メッセージリクエストを送信するように App を変更します。

まず、isSending という名前を付けた新しいブール値のプロパティを、useState フックで定義します。この新しいプロパティで isSendDisabled 定数を使用すると、button 要素の無効状態を切り替えることができます。SendButton のイベントハンドラーで messageToSend の値をクリアし、isSending を true に設定します。

API 呼び出しはこのボタンにより実行されるため、isSending ブール値を追加することで、リクエストが完了するまで SendButton でのユーザー操作を無効にし、複数の API 呼び出しが同時に発生するのを防ぐことができます。

注: メッセージの送信は、上記の「[現在のユーザーにより送信されたメッセージの認識](#)」にあるように、トークンプロバイダーに SEND_MESSAGE 権限を追加した場合にのみ機能します。

TypeScript/JavaScript:

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

// ...

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

// ...

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

// ...

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

// ...
```

新しい SendMessageRequest インスタンスを作成してリクエストを準備し、メッセージの内容をコンストラクターに渡してます。isSending および messageToSend の状態を設定した

ら、sendMessage メソッドを呼び出してリクエストをチャットルームに送信します。最後に、リクエストの確認または拒否を受け取った時点で、isSending フラグをクリアします。

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);
  }
};

// ...
```

ここで、Chatterbox を実行してみます。MessageBar を使用してメッセージの下書きを作成し、SendButton をタップしてメッセージを送信します。先に作成済みの MessageList 内にレンダリングされた、送信済みメッセージが表示されるはずですが。

メッセージの削除

チャットルームからメッセージを削除するには、適切な権限が必要です。これらの権限は、チャットルームへの認証時に使用するチャットトークンの初期化中に付与されます。このセクションでは、「[ローカル認証/承認サーバーのセットアップ](#)」(このチュートリアルパート 1)にある ServerApp により、モデレーターの機能を指定しています。この処理は、「[トークンプロバイダーの作成](#)」(同じくパート 1)で作成した tokenProvider オブジェクトを使用して、アプリ内で実行されます。

ここでは、Message を変更し、メッセージを削除する関数を追加します。

まず、App.tsx を開き DELETE_MESSAGE の機能を追加します。(capabilities は tokenProvider 関数の 2 番目のパラメータです。)

注: これにより、生成されたチャットトークンに関連付けられているユーザーがチャットルーム内のメッセージを削除できることが、ServerApp から IVS Chat API に対し伝達されます。実際には、サーバーアプリのインフラストラクチャでユーザーの権限を管理するためのバックエンドロジックは、さらに複雑なものになるでしょう。

TypeScript/JavaScript:

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

// ...

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

// ...
```

以降のステップでは、削除ボタンを表示するために、Message を変更していきます。

文字列をパラメータの 1 つとして受け取り Promise を返す新しい関数を、onDelete の名前で定義します。文字列パラメータには、コンポーネントのメッセージ ID を渡します。

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',
  },
});
```

次に、FlatList コンポーネントに加えられた最新の変更を反映するように renderItem を更新します。

App の中で handleMessage という名前の関数を定義し、それを MessageList onDelete プロパティに渡します。

TypeScript

```
// App.tsx

// ...

const handleMessage = async (id: string) => {};

const renderItem = useCallback<ListRenderItem<ChatMessage>>(({ item }) => {
  return (
    <Message key={item.id} message={item} onDelete={handleDeleteMessage} />
  );
}, [handleDeleteMessage]);

// ...
```

JavaScript

```
// App.jsx

// ...

const handleMessage = async (id) => {};

const renderItem = useCallback(({ item }) => {
  return (
    <Message key={item.id} message={item} onDelete={handleDeleteMessage} />
  );
}, [handleDeleteMessage]);
```

```
    );  
  }, [handleDeleteMessage]);  
  
  // ...
```

`DeleteMessageRequest` のインスタンスを新しく作成して、関連するメッセージ ID をコンストラクタのパラメータに渡し、リクエストを用意します。用意したリクエストを受け入れるために `deleteMessage` を呼び出します。

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

次に、`messages` の状態を更新し、削除済みメッセージが消去された新しいメッセージリストを反映させます。

`useEffect` フックでは `messageDelete` イベントを監視し、`message` パラメーターと一致する ID を持つメッセージを削除して、`messages` 状態配列を更新します。

注: `messageDelete` イベントは、現在のユーザー、またはチャットルーム内の他のユーザーによってメッセージが削除された場合に発生します。これを、`deleteMessage` リクエストの後ではなくイベントハンドラーで処理することで、メッセージ削除の処理をまとめることができます。

TypeScript/JavaScript:

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

この段階で、チャットアプリのチャットルームからユーザーを削除することが可能になりました。

次のステップ

実験として、他のユーザーの接続切断など追加のアクションも、ルームに実装してみてください。

IVS Chat Client Messaging SDK: React と React Native のベストプラクティス

このドキュメントは、React と React Native のために Amazon IVS Chat Messaging SDK を使用する際の最も重要なプラクティスについて記載しています。この情報により、React アプリ内で一般的なチャット機能を構築できるようになり、IVS Chat Messaging SDK のより高度な部分を深く掘り下げるために必要な背景知識が得られるはずです。

チャットルームイニシャライザーフックの作成

ChatRoom クラスには、接続状態を管理し、メッセージの受信やメッセージの削除などのイベントをリスンするためのコアチャットメソッドとリスナーが含まれています。ここでは、チャットインスタンスをフックに適切に格納する方法を示します。

実装

TypeScript

```
// useChatRoom.ts

import React from 'react';
import { ChatRoom, ChatRoomConfig } from 'amazon-ivs-chat-messaging';

export const useChatRoom = (config: ChatRoomConfig) => {
  const [room] = React.useState(() => new ChatRoom(config));

  return { room };
};
```

JavaScript

```
import React from 'react';
import { ChatRoom } from 'amazon-ivs-chat-messaging';

export const useChatRoom = (config) => {
  const [room] = React.useState(() => new ChatRoom(config));

  return { room };
};
```

注: 設定パラメータをその場で更新できないため、setState フックからの dispatch メソッドは使用しません。SDK はインスタンスを一度作成します。また、トークンプロバイダーを更新することはできません。

重要: ChatRoom イニシャライザーフックを 1 回使用して、新しいチャットルームインスタンスを初期化します。

例

TypeScript/JavaScript:

```
// ...

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

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

  // ...
};

// ...
```

接続状態のリスニング

オプションで、チャットルームフックで接続状態の更新をサブスクライブできます。

実装

TypeScript

```
// useChatRoom.ts

import React from 'react';
import { ChatRoom, ChatRoomConfig, ConnectionState } from 'amazon-ivs-chat-messaging';

export const useChatRoom = (config: ChatRoomConfig) => {
  const [room] = useState(() => new ChatRoom(config));

  const [state, setState] = React.useState<ConnectionState>('disconnected');

  React.useEffect(() => {
    const unsubscribeOnConnecting = room.addListener('connecting', () => {
```

```
    setState('connecting');
  });

  const unsubscribeOnConnected = room.addListener('connect', () => {
    setState('connected');
  });

  const unsubscribeOnDisconnected = room.addListener('disconnect', () => {
    setState('disconnected');
  });

  return () => {
    unsubscribeOnConnecting();
    unsubscribeOnConnected();
    unsubscribeOnDisconnected();
  };
}, []);

return { room, state };
};
```

JavaScript

```
// useChatRoom.js

import React from 'react';
import { ChatRoom } from 'amazon-ivs-chat-messaging';

export const useChatRoom = (config) => {
  const [room] = useState(() => new ChatRoom(config));

  const [state, setState] = React.useState('disconnected');

  React.useEffect(() => {
    const unsubscribeOnConnecting = room.addListener('connecting', () => {
      setState('connecting');
    });

    const unsubscribeOnConnected = room.addListener('connect', () => {
      setState('connected');
    });

    const unsubscribeOnDisconnected = room.addListener('disconnect', () => {
```

```
        setState('disconnected');
    });

    return () => {
        unsubscribeOnConnecting();
        unsubscribeOnConnected();
        unsubscribeOnDisconnected();
    };
}, []);

return { room, state };
};
```

ChatRoom インスタンスプロバイダー

(Prop Drilling (バケツリレー) を避けるために) 他のコンポーネントでフックを使用するには、React context を使用してチャットルームプロバイダーを作成できます。

実装

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;
```

例

`ChatRoomProvider` を作成したら、インスタンスを `useChatRoomContext` で使用できます。

重要: 接続をリッスンしている場合に不要な再レンダリングを避けるために、チャット画面と中間の他のコンポーネントの間で `context` へのアクセスが必要な場合にのみ、プロバイダーをルートレベルに配置してください。それ以外の場合は、チャット画面のできるだけ近くにプロバイダーを配置してください。

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

// ...
```

メッセージリスナーの作成

すべての着信メッセージについての最新の状態を知るには、`message` および `deleteMessage` イベントをサブスクライブする必要があります。コンポーネントにチャットメッセージを提供するコードを次に示します。

重要: チャットメッセージリスナーがメッセージの状態を更新するときに、多くの再レンダリングが発生する可能性があるため、パフォーマンス上の理由から、`ChatMessageContext` と `ChatRoomProvider` は分離されています。`ChatMessageProvider` を使用するコンポーネントで `ChatMessageContext` を適用することを忘れないでください。

実装

TypeScript

```
// ChatMessagesContext.tsx

import React from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useChatRoomContext } from './ChatRoomContext';

const ChatMessagesContext = React.createContext<ChatMessage[] |
  undefined>(undefined);
```

```
export const useChatMessagesContext = () => {
  const context = React.useContext(ChatMessagesContext);

  if (context === undefined) {
    throw new Error('useChatMessagesContext must be within ChatMessagesProvider');
  }

  return context;
};

export const ChatMessagesProvider = ({ children }: { children: React.ReactNode }) => {
  const room = useChatRoomContext();

  const [messages, setMessages] = React.useState<ChatMessage[]>([]);

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

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

    return () => {
      unsubscribeOnMessageDeleted();
      unsubscribeOnMessageReceived();
    };
  }, [room]);

  return <ChatMessagesContext.Provider value={messages}>{children}</
ChatMessagesContext.Provider>;
};
```

JavaScript

```
// ChatMessagesContext.jsx

import React from 'react';
import { useChatRoomContext } from './ChatRoomContext';
```

```
const ChatMessagesContext = React.createContext(undefined);

export const useChatMessagesContext = () => {
  const context = React.useContext(ChatMessagesContext);

  if (context === undefined) {
    throw new Error('useChatMessagesContext must be within ChatMessagesProvider');
  }

  return context;
};

export const ChatMessagesProvider = ({ children }) => {
  const room = useChatRoomContext();

  const [messages, setMessages] = React.useState([]);

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

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

    return () => {
      unsubscribeOnMessageDeleted();
      unsubscribeOnMessageReceived();
    };
  }, [room]);

  return <ChatMessagesContext.Provider value={messages}>{children}</
ChatMessagesContext.Provider>;
};
```

React の例

重要: メッセージコンテナを `ChatMessagesProvider` で必ずラップしてください。Message 行は、メッセージの内容を表示するコンポーネントの例です。

TypeScript/JavaScript:

```
// your message list component...

import React from 'react';
import { useChatMessagesContext } from './ChatMessagesContext';

const MessageListContainer = () => {
  const messages = useChatMessagesContext();

  return (
    <React.Fragment>
      {messages.map((message) => (
        <MessageRow message={message} />
      ))}
    </React.Fragment>
  );
};
```

React Native の例

デフォルトでは、`ChatMessage` には `id` が含まれており、これは各行の `FlatList` の React キーとして自動的に使用されます。したがって、`keyExtractor` を渡す必要はありません。

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} />;  
};
```

アプリ内の複数のチャットルームインスタンス

アプリで複数のチャットルームを同時に使用する場合は、チャットごとに各プロバイダーを作成し、チャットプロバイダーを使用することをお勧めします。この例では、ヘルプボットとカスタマーヘルプチャットを作成しています。両方のためにプロバイダーを作成します。

TypeScript

```
// SupportChatProvider.tsx  
  
import React from 'react';  
import { SUPPORT_ROOM_ID, SOCKET_URL } from '../././config';  
import { tokenProvider } from './tokenProvider';  
import { ChatRoomProvider } from './ChatRoomContext';  
import { useChatRoom } from './useChatRoom';  
  
export const SupportChatProvider = ({ children }: { children: React.ReactNode }) =>  
{  
  const { room } = useChatRoom({  
    regionOrUrl: SOCKET_URL,  
    tokenProvider: () => tokenProvider(SUPPORT_ROOM_ID, ['SEND_MESSAGE']),  
  });  
}
```

```
});

return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};

// SalesChatProvider.tsx

import React from 'react';
import { SALES_ROOM_ID, SOCKET_URL } from '../././config';
import { tokenProvider } from '.././tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SalesChatProvider = ({ children }: { children: React.ReactNode }) => {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SALES_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};
```

JavaScript

```
// SupportChatProvider.jsx

import React from 'react';
import { SUPPORT_ROOM_ID, SOCKET_URL } from '../././config';
import { tokenProvider } from '.././tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SupportChatProvider = ({ children }) => {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SUPPORT_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};

// SalesChatProvider.jsx
```

```
import React from 'react';
import { SALES_ROOM_ID, SOCKET_URL } from '../././config';
import { tokenProvider } from '.././tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SalesChatProvider = ({ children }) => {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SALES_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};
```

React の例

同じ ChatRoomProvider を使用する別のチャットプロバイダーを使用できるようになりました。後で、各画面/ビュー内で同じ useChatRoomContext を再利用できます。

TypeScript/JavaScript:

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

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

```
);  
};
```

React Native の例

TypeScript/JavaScript:

```
// App.tsx / App.jsx  
  
const App = () => {  
  return (  
    <Stack.Navigator>  
      <Stack.Screen name="SupportChat">  
        <SupportChatProvider>  
          <SupportChatScreen />  
        </SupportChatProvider>  
      </Stack.Screen>  
      <Stack.Screen name="SalesChat">  
        <SalesChatProvider>  
          <SalesChatScreen />  
        </SalesChatProvider>  
      </Stack.Screen>  
    </Stack.Navigator>  
  );  
};
```

TypeScript/JavaScript:

```
// SupportChatScreen.tsx / SupportChatScreen.jsx  
  
// ...  
  
const SupportChatScreen = () => {  
  const room = useChatRoomContext();  
  
  const handleConnect = () => {  
    room.connect();  
  };  
  
  return (  
    <>  
      <Button title="Connect" onPress={handleConnect} />  
      <MessageListContainer />  
    </>  
  );  
};
```



```
    </>
  );
};

// SalesChatScreen.tsx / SalesChatScreen.jsx

// ...

const SalesChatScreen = () => {
  const room = useChatRoomContext();

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

  return (
    <>
      <Button title="Connect" onPress={handleConnect} />
      <MessageListContainer />
    </>
  );
};
```

Amazon IVS Chat のセキュリティ

AWS では、クラウドのセキュリティが最優先事項です。AWS のお客様は、セキュリティを最も重視する組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャから利点を得られます。

セキュリティは、AWS とお客様の間の共有責任です。[責任共有モデル](#)では、この責任がクラウドのセキュリティおよびクラウド内のセキュリティとして説明されています。

- クラウドのセキュリティ - AWS は、AWS クラウドで AWS サービスを実行するインフラストラクチャを保護する責任を負います。また、AWS は、使用するサービスを安全に提供します。[AWS コンプライアンスプログラム](#)の一環として、サードパーティーの監査が定期的にセキュリティの有効性をテストおよび検証しています。
- クラウドにおけるセキュリティ - お客様の責任は、使用する AWS のサービスに応じて判断されます。お客様は、データの機密性、組織の要件、および適用法令と規制などのその他要因に対する責任も担います。

このドキュメントは、Amazon IVS Chat を使用する際の責任共有モデルの適用方法を理解するのに役立ちます。以下のトピックでは、セキュリティおよびコンプライアンスの目的を達成するように Amazon IVS Chat を設定する方法について説明します。

トピック

- [IVS Chat データ保護](#)
- [IVS Chat での Identity and Access Management](#)
- [IVS Chat 用マネージドポリシー](#)
- [IVS Chat のサービスにリンクされたロールの使用](#)
- [IVS Chat のログ記録とモニタリング](#)
- [IVS Chat のインシデントへの対応](#)
- [IVS Chat の耐障害性](#)
- [IVS Chat インフラストラクチャセキュリティ](#)

IVS Chat データ保護

Amazon Interactive Video Service (IVS) Chat に送信されるデータに対して、以下のデータ保護が施されています。

- Amazon IVS Chat のトラフィックは WSS を使用して、転送中のデータを安全に保ちます。
- Amazon IVS Chat トークンは、KMS カスタマー管理キーを使用して暗号化されます。

Amazon IVS Chat はお客様 (エンドユーザー) データの提供を必要としません。チャットルーム、入力、入力セキュリティグループにお客様 (エンドユーザー) データが求められるフィールドはありません。

[Name (名前)] フィールドのような自由記述のフィールドに、お客様 (エンドユーザー) のアカウント番号などの機密扱いの識別情報を入力しないでください。Amazon IVS コンソール、API、AWS CLI、AWS SDK を操作する場合も同様です。Amazon IVS Chat に入力したデータは、診断ログに含まれる場合があります。

ストリームはエンドツーエンドで暗号化されません。ストリームは、IVS ネットワーク内で暗号化されずに送信され、処理されます。

IVS Chat での Identity and Access Management

AWS Identity and Access Management (IAM) は、AWS リソースへのアクセスをアカウント管理者が安全に制御するために役立つ AWS のサービスです。「IVS Low-Latency Streaming User Guide」の「[Identity and Access Management in IVS](#)」を参照してください。

対象者

IAM の用途は、Amazon IVS で行う作業によって異なります。「IVS Low-Latency Streaming User Guide」の「[Audience](#)」を参照して下さい。

Amazon IVS で IAM を使用する方法

Amazon IVS API リクエストを作成する前に、IAM ID (ユーザー、グループ、ロール) および IAM ポリシーを 1 つ以上作成する必要があります。次に、ポリシーを ID にアタッチします。アクセス許可が反映されるまでに数分かかります。それまでは、API リクエストは拒否されます。

Amazon IVS を IAM と連携させる方法の概要は、「IAM ユーザーガイド」の「[IAM と連携する AWS のサービス](#)」を参照してください。

ID

IAM アイデンティティを作成して、AWS アカウントのユーザーとプロセスに認証を提供できます。IAM グループは、1つのユニットとして管理できる IAM ユーザーのコレクションです。IAM ユーザーガイドの [ID \(ユーザー、グループ、ロール\)](#) を参照してください。

ポリシー

ポリシーは、要素から成る JSON のアクセス許可ポリシードキュメントです。「IVS Low-Latency Streaming User Guide」の「[Policies](#)」を参照して下さい。

Amazon IVS Chat は次の 3 つの要素をサポートしています。

- アクション — Amazon IVS Chat のポリシーアクションは、アクションの前に `ivschat` プレフィックスを使用します。例えば、Amazon IVS Chat `CreateRoom` API メソッドを使用して Amazon IVS Chat ルームを作成するアクセス許可を誰かに付与するには、`ivschat:CreateRoom` アクションのポリシーにその誰かを含めます。ポリシーステートメントには、`Action` または `NotAction` 要素を含める必要があります。
- リソース — Amazon IVS Chat ルームリソースには次のような [ARN](#) 形式があります。

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

例えば、ステートメントで `VgNkEJg0VX9N` ルームを指定するには、この ARN を使用します。

```
"Resource": "arn:aws:ivschat:us-west-2:123456789012:room/VgNkEJg0VX9N"
```

リソースの作成など、一部の Amazon IVS Chat アクションは、特定のリソースで実行できません。このような場合は、ワイルドカード (*) を使用する必要があります。

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

- 条件 — Amazon IVS Chat は、`aws:RequestTag`、`aws:TagKeys`、および `aws:ResourceTag` のグローバル条件キーをサポートしています。

変数をポリシーのプレースホルダーとして使用できます。例えば、ユーザーの IAM ユーザー名でタグ付けされている場合のみ、リソースにアクセスする IAM ユーザーアクセス許可を付与できます。IAM ユーザーガイドの [変数とタグ](#) を参照してください。

Amazon IVS には、事前設定された一連のアクセス許可をアイデンティティに付与する (読み取り専用またはフルアクセス) ために使用できる AWS マネージドポリシーが用意されています。以下に示す ID ベースのポリシーの代わりにマネージドポリシーを使用することもできます。詳細については、「[Amazon IVS Chat 用マネージドポリシー](#)」を参照してください。

Amazon IVS タグに基づく承認

タグは、Amazon IVS Chat リソースにアタッチすることも、Amazon IVS Chat へのリクエストで渡すこともできます。タグに基づいてアクセスを制御するには、`aws:ResourceTag/key-name`、`aws:RequestTag/key-name`、または `aws:TagKeys` の条件キーを使用して、ポリシーの条件要素でタグ情報を提供します。Amazon IVS Chat リソースのタグ付けの詳細については、「[IVS Chat API Reference](#)」の「Tagging」を参照してください。

ロール

IAM ユーザーガイドの [IAM ロール](#) および [一時的セキュリティ認証情報](#) を参照してください。

IAM ロールは AWS アカウント内のエンティティで、特定の許可を持っています。

Amazon IVS では、一時認証情報の使用をサポートしています。一時的な認証情報を使用して、フェデレーションでサインインする、IAM ロールを引き受ける、またはクロスアカウントロールを引き受けることができます。一時的なセキュリティ認証情報を取得するには、`AssumeRole` や `GetFederationToken` などの [AWS Security Token Service](#) API オペレーションを呼び出します。

特権アクセスと非特権アクセス

API リソースには特権アクセスがあります。非特権再生アクセスは、プライベートチャンネルを介してセットアップできます。「[Setting Up IVS Private Channels](#)」を参照してください。

ポリシーのベストプラクティス

IAM ユーザーガイドの [IAM ベストプラクティス](#) を参照してください。

アイデンティティベースのポリシーは非常に強力です。アカウント内で、Amazon IVS リソースを作成、アクセス、削除できるかどうかを決定します。これらのアクションを実行すると、AWS アカウントに追加料金が発生する可能性があります。次の推奨事項に従ってください。

- 最小権限を付与する – カスタムポリシーを作成するときは、タスクの実行に必要なアクセス許可のみを付与します。最小限のアクセス許可から開始し、必要に応じて追加のアクセス許可を付与します。この方法は、寛容なアクセス許可で開始し、後でそれらを強化しようとするよりも安全で

す。具体的には、`ivschat:*` を管理アクセスのために使用し、アプリケーションでは使用しないでください。

- 機密性の高いオペレーションに Multi-Factor Authentication (MFA) を有効にする – 追加セキュリティとして、機密性の高いリソースまたは API オペレーションにアクセスする IAM ユーザーに対して、MFA を使用するよう求めます。
- 追加のセキュリティとしてポリシー条件を使用する – 実行可能な範囲内で、ID ベースのポリシーでリソースへのアクセスを許可する条件を定義します。例えば、条件を記述して、リクエストが発生する IP アドレスの範囲を指定することができます。指定された日付や時間範囲内でのみリクエストを許可したり、SSL や MFA の使用を要求したりする条件を記述できます。

アイデンティティベースのポリシーの例

Amazon IVS コンソールを使用する

Amazon IVS コンソールにアクセスするには、AWS アカウントの Amazon IVS Chat リソースに関する詳細をリスト化および表示することを許可する最小限の許可セットが必要です。最小限必要なアクセス許可よりも制限された ID ベースのポリシーを作成すると、そのポリシーをアタッチした ID に対してコンソールが意図したとおりに機能しません。Amazon IVS コンソールへのアクセスを確実にするには、以下のポリシーを ID にアタッチします (IAM ユーザーガイドの [IAM アクセス許可の追加および削除](#) を参照してください)。

次のポリシーの部分は、以下へのアクセスを提供します。

- すべての Amazon IVS Chat API エンドポイント
- ユーザーの Amazon IVS Chat の [Service Quotas](#)
- Lambda を一覧表示し、Amazon IVS Chat のモデレーションのために選択したラムダのアクセス許可を追加する
- チャットセッションのメトリクスを取得する 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 Chat のリソースベースのポリシー

メッセージをレビューするために Lambda リソースを呼び出すには、Amazon IVS Chat サービスにアクセス許可を与える必要があります。これを行うには、「[AWS Lambda のリソースベースポリシーの使用](#)」(「AWS Lambda 開発デベロッパーガイド」内)の手順に従って、以下のように指定されたフィールドに記入します。

Lambda リソースへのアクセスを制御するために、以下に基づく条件を使用することができます。

- **SourceArn** — サンプルポリシーではワイルドカード (*) を使用して、アカウント内のすべてのルームが Lambda を呼び出すことを許可します。オプションで、アカウント内のルームを指定して、そのルームのみが Lambda を呼び出すことを許可できます。
- **SourceAccount** — 以下のサンプルポリシーでは、AWS アカウント ID は 123456789012 です。

```
{
```

```
"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/*"
      }
    }
  }
]
```

トラブルシューティング

Amazon IVS Chat と IAM の使用に伴って発生する可能性がある一般的な問題の診断と修正については、「IVS Low-Latency Streaming User Guide」の「[Troubleshooting](#)」を参照してください。

IVS Chat 用マネージドポリシー

AWS 管理ポリシーは、AWS が作成および管理するスタンドアロンポリシーです。「IVS Low-Latency Streaming User Guide」の「[Managed Policies for Amazon IVS](#)」を参照して下さい。

IVS Chat のサービスにリンクされたロールの使用

Amazon IVS は、AWS IAM [サービスリンクロール](#)を使用しています。「IVS Low-Latency Streaming User Guide」の「[Using Service-Linked Roles for Amazon IVS](#)」を参照してください。

IVS Chat のログ記録とモニタリング

パフォーマンスおよび/またはオペレーションを記録するには、Amazon CloudTrail を使用します。「IVS Low-Latency Streaming User Guide」の「[Logging Amazon IVS API Calls with AWS CloudTrail](#)」を参照してください。

IVS Chat のインシデントへの対応

インシデントを検出またはアラートするために、Amazon EventBridge イベントを介してストリームの状態をモニタリングできます。「[Amazon IVS 低レイテンシーストリーミングで Amazon EventBridge を使用する](#)」および「[Amazon IVS リアルタイムストリーミングで Amazon EventBridge を使用する](#)」を参照してください。

Amazon IVS の全体的な状態 (リージョン別) に関する情報を得るには、[AWS Health Dashboard](#) を使用します。

IVS Chat の耐障害性

IVS API は、AWS グローバルインフラストラクチャを使用し、AWS リージョンとアベイラビリティゾーンを中心として構築されます。「IVS Low-Latency Streaming User Guide」の「[IVS Resilience](#)」を参照して下さい。

IVS Chat インフラストラクチャセキュリティ

マネージドサービスである Amazon IVS は AWS グローバルネットワークセキュリティ手順で保護されています。これらは、[Best Practices for Security, Identity, & Compliance](#) に記載されています。

API 呼び出し

ネットワーク経由で Amazon IVS にアクセスするには、AWS が発行した API 呼び出しを使用します。「IVS Low-Latency Streaming User Guide」の「インフラストラクチャセキュリティ」の「[API コール](#)」を参照してください。

Amazon IVS Chat

Amazon IVS Chat メッセージの取り込みと配信は、エッジへの暗号化された WSS 接続を介して行われます。Amazon IVS Messaging API は、暗号化された HTTPS 接続を使用します。動画のスト

リーミングや再生と同様、TLS バージョン 1.2 以降が必要です。メッセージングデータは、内部で暗号化されずに転送、処理されることがあります。

IVS Chat Service Quotas

以下は、Amazon Interactive Video Service (IVS) チャットのエンドポイント、リソース、およびその他のオペレーションの Service Quotas と制限です。Service Quotas (制限とも呼ばれます) とは、AWS アカウントのサービスリソースまたはオペレーションの最大数のことです。つまり、これらの制限は、表に明記されていない限り AWS のアカウントごとに適用されます。[AWS Service Quotas](#) を参照してください。

AWS のサービスにプログラムで接続するには、エンドポイントを使用します。[AWS サービスエンドポイント](#) も参照してください。

すべてのクォータはリージョンごとに適用されます。

Service Quotas の引き上げ

引き上げ可能なクォータについては、[AWS コンソール](#) からレートの引き上げをリクエストできます。コンソールでは、Service Quotas に関する情報も閲覧できます。

API コールレートクォータは調整できません。

API コールレートクォータ

エンドポイントタイプ	エンドポイント	デフォルト
メッセージング	DeleteMessage	100 TPS
メッセージング	DisconnectUser	100 TPS
メッセージング	SendEvent	100 TPS
チャットトークン	CreateChatToken	200 TPS
ログ記録設定	CreateLoggingConfiguration	3 TPS
ログ記録設定	DeleteLoggingConfiguration	3 TPS
ログ記録設定	GetLoggingConfiguration	3 TPS
ログ記録設定	ListLoggingConfigurations	3 TPS

エンドポイントタイプ	エンドポイント	デフォルト
ログ記録設定	UpdateLoggingConfiguration	3 TPS
ルーム	CreateRoom	5 TPS
ルーム	DeleteRoom	5 TPS
ルーム	GetRoom	5 TPS
ルーム	ListRooms	5 TPS
ルーム	UpdateRoom	5 TPS
タグ	ListTagsForResource	10 TPS
タグ	TagResource	10 TPS
タグ	UntagResource	10 TPS

その他のクォータ

リソースまたは機能	デフォルト	引き上げ可能	説明
同時チャット接続	50,000	あり	アカウントごとに、AWS リージョンのすべてのルームで同時接続できるチャットの最大数。
ロギング設定	10	[Yes (はい)]	現在の AWS リージョンのアカウントごとに作成できるロギング設定の最大数。
メッセージレビューハンドラーのタイムアウト期間	200	なし	現在の AWS リージョンのすべてのメッセージレビューハンドラーのタイムアウト期間 (ミリ秒単位)。これを超えると、メッセージレビュー

リソースまたは機能	デフォルト	引き上げ可能	説明
			ハンドラー用に設定した <code>fallbackResult</code> フィールドの値に応じて、メッセージが許可または拒否されます。
すべてのルームでの <code>DeleteMessage</code> リクエストのレート	100	あり	すべてのルームで 1 秒あたりに実行できる <code>DeleteMessage</code> リクエストの最大数。リクエストは、Amazon IVS Chat API または Amazon IVS Chat メッセージング API (WebSocket) のいずれかから送信できます。
すべてのルームでの <code>DisconnectUser</code> リクエストのレート	100	あり	すべてのルームで 1 秒あたりに実行できる <code>DisconnectUser</code> リクエストの最大数。リクエストは、Amazon IVS Chat API または Amazon IVS Chat メッセージング API (WebSocket) のいずれかから送信できます。
接続あたりのメッセージングリクエストのレート	10	なし	チャット接続が実行できる 1 秒あたりのメッセージングリクエストの最大数。
すべてのルームでの <code>SendMessage</code> リクエストのレート	1000	あり	すべてのルームで 1 秒あたりに実行できる <code>SendMessage</code> リクエストの最大数。これらのリクエストは、Amazon IVS Chat メッセージング API (WebSocket) から送信されます。

リソースまたは機能	デフォルト	引き上げ可能	説明
ルームあたりの SendMessage リクエストのレート	100	いいえ (ただし、API を使用して設定可能)	1 つのルームに対して 1 秒あたりに実行できる SendMessage リクエストの最大数。これは、 CreateRoom および UpdateRoom の maximumMessageRatePerSecond フィールドで設定できます。これらのリクエストは、Amazon IVS Chat メッセージング API (WebSocket) から送信されます。
ルーム	50,000	あり	AWS リージョンごとのアカウントあたりのチャットルームの最大数。

Service Quotas と CloudWatch 使用量メトリクスの統合

CloudWatch 使用量メトリクスを使用すると、Service Quotas をプロアクティブに管理できます。これらのメトリクスを使用すると、現在のサービスの使用状況を CloudWatch のグラフやダッシュボードを使って可視化できます。Amazon IVS Chat の使用量メトリクスは、Amazon IVS Chat の Service Quotas に対応しています。

CloudWatch のメトリクスの数学関数を使用すると、それらリソースの Service Quotas をグラフ化できます。また、使用量が Service Quotas に近づいたときに警告するアラームも設定できます。

使用量メトリクスにアクセスするには

1. Service Quotas のコンソールを開きます。 <https://console.aws.amazon.com/servicequotas/>
2. ナビゲーションペインで、[AWS services (AWS のサービス)] を選択します。
3. AWS のサービス一覧から、[Amazon Interactive Video Service Chat] を探し、選択します。
4. Service Quotas の一覧から、目的のサービスクォータを選択します。新しいページが開き、Service Quotas/メトリクスに関する情報が表示されます。

または、CloudWatch コンソールからこれらのメトリクスにアクセスすることも可能です。[AWS Namespaces (AWS の名前空間)] で、[Usage (使用量)] を選択します。次に、[サービス] の一覧から [IVS Chat] を選択します。(「[Monitoring Amazon IVS Chat](#)」を参照してください。)

AWS/使用量 の名前空間の場合、Amazon IVS Chat には次のメトリクスが表示されます。

メトリクス名	説明
ResourceCount	<p>お使いのアカウントで実行されている特定のリソースの数。リソースは、メトリクスに関連付けられたディメンションによって定義されます。</p> <p>有効な統計: 最大 (1 分間に使用されるリソースの最大数)。</p>

次のディメンションは、使用量メトリクスを絞り込むために使用されます。

ディメンション	説明
Service	リソースが含まれる AWS のサービスの名前。有効な値: IVS Chat。
Class	追跡されているリソースのクラス。有効な値: None。
Type	追跡されるリソースのタイプ。有効な値: Resource。
Resource	<p>AWS リソースの名前。有効な値: ConcurrentChatConnections 。</p> <p>ConcurrentChatConnections 使用状況メトリクスは、「Monitoring Amazon IVS Chat」に説明されているように、AWS/IVSChat 名前空間にあるもの (ディメンションは None) のコピーです。</p>

使用量メトリクスの CloudWatch アラームを作成する

Amazon IVS Chat 使用量メトリクスに基づいて CloudWatch アラームを作成するには

1. Service Quotas のコンソールで、上記の説明に従って目的のサービスクォータを選択します。現在、アラームを作成できるのは、ConcurrentChatConnections に対してのみです。
2. Amazon CloudWatch アラームのセクションで [Create (作成)] を選択します。

3. [Alarm threshold (アラームのしきい値)] のドロップダウンリストから、アラーム値として設定する、適用されたクォータ値のパーセンテージを選択します。
4. [Alarm name (アラーム名)] に、アラームの名前を入力します。
5. [Create (作成)] を選択します。

IVS Chat のトラブルシューティング

このドキュメントでは、Amazon Interactive Video Service (IVS) Chat のベストプラクティスおよびトラブルシューティングのヒントを説明します。IVS Chat に関連する動作は、多くの場合、IVS 動画に関連する動作とは異なります。詳細については、「[Amazon IVS Chat の開始方法](#)」を参照してください。

トピック:

- [the section called “ルームが削除されたときに IVS チャットの接続が切断されなかったのはなぜですか?”](#)

ルームが削除されたときに IVS チャットの接続が切断されなかったのはなぜですか？

チャットルームリソースが削除されても、そのルームがアクティブに使用されている場合、そのルームに接続しているチャットクライアントは自動的に切断されません。チャットアプリケーションがチャットトークンを更新すると、その時点で接続が切断されます。または、すべてのユーザーをチャットルームから削除するには、すべてのユーザーの接続を手動で切断する必要があります。

IVS の用語集

「[AWS 用語集](#)」も参照してください。下の表では、LL は IVS 低レイテンシーストリーミング、RT、IVS リアルタイムストリーミングを表します。

言葉	説明	LL	RT	チャット
AAC	高度なオーディオコーディング。AAC は、非可逆デジタルオーディオ 圧縮 用のオーディオコーディング規格です。MP3 形式の後継として設計された AAC は、通常、同じビットレートで MP3 よりも高い音質を実現します。AAC は MPEG-2 および MPEG-4 の仕様の一部として ISO と IEC によって標準化されています。	✓	✓	
アダプティブビットレートのストリーミング	アダプティブビットレート (ABR) ストリーミングにより、IVS プレーヤーは接続品質が低下した場合は低い ビットレート に切り替え、接続品質が向上した場合は高いビットレートに戻すことができます。	✓		
アダプティブストリーミング	「 サイマルキャストによるレイヤードエンコーディング 」を参照してください。		✓	
管理ユーザー。	AWS アカウントで利用可能なリソースとサービスへの管理アクセス権を持つ AWS ユーザー。 「AWS セットアップユーザーガイド」の「 用語 」を参照してください。	✓	✓	✓
ARN	AWS リソースに固有の識別子である Amazon リソースネーム 。具体的な ARN 形式は、リソースの種類によって異なります。IVS リソースで使用される ARN 形式については、「サービス認証リファレンス」を参照してください。	✓	✓	✓

言葉	説明	LL	RT	チャット
アスペクト比	フレーム幅とフレーム高さの比率について説明します。たとえば、16:9 はフル HD または 1080p の 解像度 に対応するアスペクト比です。	✓	✓	
オーディオモード	さまざまなタイプのモバイルデバイスユーザーや使用する機器に合わせて最適化された、プリセットまたはカスタムのオーディオ設定。「 IVS Broadcast SDK: モバイルオーディオモード (リアルタイムストリーミング) 」を参照してください。		✓	
AVC、H.264、MPEG-4 Part 10	アドバンスドビデオコーディング (H.264 または MPEG-4 Part 10 と呼ばれる) は、非可逆デジタルビデオ 圧縮 用のビデオ圧縮規格です。	✓	✓	
背景置換	ライブストリームのクリエイターが背景を変更できるようにする カメラフィルター の一種。「IVS Broadcast SDK: サードパーティーのカメラフィルター (リアルタイムストリーミング)」の「 背景の置換 」を参照してください。		✓	
ビットレート	1 秒あたりに送受信されるビット数のストリーミングメトリック。	✓	✓	
ブロードキャスト、配信者	ストリーム 、 ストリーマー のためのその他の用語。	✓		
バッファリング	コンテンツが再生されることになっている時点よりも前に再生デバイスがコンテンツをダウンロードできない場合に発生する状態。バッファリングは、コンテンツがランダムに停止および開始する (「途切れ」とも呼ばれます)、コンテンツが長時間にわたって停止する (「フリーズ」とも呼ばれます)、または IVS プレイヤーが再生を一時停止するなど、いくつかの態様で現れる可能性があります。	✓	✓	

言葉	説明	LL	RT	チャット
バイト範囲プレイリスト	<p>標準の HLS プレイリスト よりも詳細なプレイリスト。標準 HLS プレイリストは 10 秒のメディアファイルで構成されています。バイト範囲のプレイリストでは、セグメント再生時間は ストリーム に設定された キーフレーム間隔 と同じです。</p> <p>バイトレンジプレイリストは、S3 バケット に自動記録されたブロードキャストでのみ使用できます。HLS プレイリスト に加えて作成されます。「Amazon S3 への自動レコーディング (低レイテンシーストリーミング)」の「バイトレンジプレイリスト」を参照してください。</p>	✓		
CBR	<p>コンスタントビットレートとは、ブロードキャスト中に起こる事象に関わらず、動画の再生中ずっと一定のビットレートを維持するエンコーダー用のレート制御方法です。アクション中の小康状態は希望のビットレートになるようにパディングされ、ピークはターゲットビットレートに合うようにエンコーディングの品質を調整することで量子化できます。VBR ではなく CBR を使用することを強くお勧めします。</p>	✓	✓	
CDN	<p>コンテンツ配信ネットワーク (Content Delivery Network または Content Distribution Network) は、ストリーミングビデオなどのコンテンツをユーザーのいる場所に近づけることで配信を最適化する、地理的に分散したソリューションです。</p>	✓		

言葉	説明	LL	RT	チャット
Channel	インジェストサーバー 、 ストリームキー 、 再生 URL 、録画オプションなど、ストリーミングの設定を保存する IVS リソース。ストリーマーは、チャンネルに関連付けられたストリームキーを使用してブロードキャストを開始します。すべてのメトリクスとブロードキャスト中に生成される イベント は、チャンネルリソースに関連付けられています。	✓		
チャンネルタイプ	チャンネル の許容 解像度 と フレームレート を決定します。「IVS 低レイテンシーストリーミング API リファレンス」の「 チャンネルタイプ 」を参照してください。	✓		
チャットのログ記録	ログ記録設定を チャットルーム に関連付けることで有効にできる詳細オプション。			✓
チャットルーム	メッセージレビューハンドラー や チャットロギング などのオプション機能を含む、チャットセッションの設定を保存する IVS リソース。「IVS Chat の開始方法」の「 ステップ 2: チャットルームを作成する 」を参照してください。			✓
クライアントサイドコンポジション	ホスト デバイスを使用してステージ参加者からのオーディオストリームとビデオストリームをミックスし、それらをコンポジットストリームとして IVS チャンネル に送信します。これにより、クライアントリソースの使用率が高くなり、 ステージ や ホスト の問題が視聴者に影響を与えるリスクは高まりますが、 コンポジション の外観をより細かく制御できます。 「 サーバーサイドコンポジション 」も参照してください。	✓	✓	

言葉	説明	LL	RT	チャット
CloudFront	Amazon が提供する CDN サービス。	✓		
CloudTrail	AWS や外部ソースからのイベントやアカウントアクティビティを収集、監視、分析、保持するための AWS サービス。「 AWS CloudTrail を使用した IVS.API コールログ作成 」を参照してください。	✓	✓	✓
CloudWatch	アプリケーションの監視、パフォーマンスの変化への対応、リソース使用の最適化、および運用状況に関するインサイトの提供を行う AWS サービス。CloudWatch を使用して IVS メトリクスをモニタリングできます。「 IVS リアルタイムストリーミングのモニタリング 」と「 IVS 低レイテンシーストリーミングのモニタリング 」を参照してください。	✓	✓	✓
コンポジション	複数のソースからのオーディオストリームとビデオストリームを 1 つのストリームにまとめるプロセス。	✓	✓	
コンポジションパイプライン	複数のストリームを結合し、結果のストリームをエンコードするために必要な一連の処理ステップ。	✓	✓	
圧縮	元の表示よりも少ないビット数で情報をエンコードします。いずれの圧縮も、可逆圧縮または非可逆圧縮です。可逆圧縮は、統計上の冗長性を特定して排除することでビット数を削減します。可逆圧縮では情報が失われることはありません。非可逆圧縮は、不要な情報や重要度の低い情報を削除することでビット数を削減します。	✓	✓	

言葉	説明	LL	RT	チャット
コントロールプレーン	チャンネル 、 ステージ 、 チャットルーム などの IVS リソースに関する情報を保存し、これらのリソースを作成および管理するためのインターフェースを提供します。リージョナルであり、AWS リージョン に基づきます。	✓	✓	✓
CORS	Cross-Origin Resource Sharing は、特定のドメインにロードされたクライアントウェブアプリケーションが異なるドメイン内 S3 バケット などのリソースと通信する方法を定義します。アクセスはヘッダー、HTTP メソッド、オリジンドメインに基づいて設定できます。「Amazon Simple Storage Service ユーザーガイド」の「 クロスオリジンリソース共有 (CORS) の使用 – Amazon Simple Storage Service 」を参照してください。	✓		
カスタム画像ソース	IVS Broadcast SDK が提供するインターフェース。プリセットカメラに限定されず、アプリケーションが独自の画像入力を行えるようにします。	✓	✓	
データプレーン	データを 取り込み から出力まで伝送するインフラストラクチャ。 コントロールプレーン で管理される設定に基づいて動作し、AWS リージョンに限定されません。	✓	✓	✓
エンコーダ。エンコーディングします	動画やオーディオコンテンツをストリーミングに適したデジタル形式に変換するプロセス。エンコーディングはハードウェアベースでもソフトウェアベースでもかまいません。	✓	✓	

言葉	説明	LL	RT	チャット
イベント	IVS が AmazonEventBridge モニタリングサービスに発行する自動通知。イベントは、 ステージやコンポジションパイプライン などのストリーミングリソースの状態や状態の変化を表します。「 IVS 低レイテンシーストリーミングで Amazon EventBridge を使用する 」および「 IVS リアルタイムストリーミングで Amazon EventBridge を使用する 」を参照してください。	✓	✓	✓
FFmpeg	動画やオーディオのファイルやストリームを処理するためのライブラリとプログラム群で構成される、無料でオープンソースのソフトウェアプロジェクト。 FFmpeg は、オーディオと動画を録画、変換、ストリーミングするためのクロスプラットフォームソリューションを提供します。	✓		
断片化されたストリーム	ブロードキャストが切断され、 チャンネル の録画設定で指定された時間内に再接続されるときに作成されます。生成された複数のストリームは、単一のブロードキャストと見なされ、マージされ単一の録画ストリームになります。「Amazon S3 への自動記録(低レイテンシーストリーミング)」の「 フラグメント化されたストリームのマージ 」を参照してください。	✓		
フレームレート	1 秒あたりに送受信される動画フレーム数のストリーミングメトリック。	✓	✓	
HLS	HTTP ライブストリーミング (HLS) は、IVS ストリームを視聴者に配信するために使用される HTTP ベースの アダプティブビットレートストリーミング 通信プロトコルです。	✓		

言葉	説明	LL	RT	チャット
HLS のプレイリスト	ストリームを構成するメディアセグメントのリスト。標準 HLS プレイリストは 10 秒のメディアファイルで構成されています。HLS は、より詳細な バイト範囲のプレイリスト をサポートしています。	✓		
ホスト	ビデオやオーディオをステージに送信するリアルタイムのイベント 参加者 。		✓	
IAM	Identity and Access Management は、ユーザーが ID を管理し、IVS を含む AWS のサービスとリソースへのアクセスを安全に管理できるようにする AWS サービスです。	✓	✓	✓
取り込み	IVS は、ホストまたはブロードキャストから動画ストリームを受信して処理または視聴者や他の参加者に配信するためのプロセスです。	✓	✓	
取り込みサーバー	ビデオストリームを受信してトランスコーディングシステムに配信します。トランスコーディングシステムでは、ストリームが トランスミックス されるか、 HLS にトランスコーディング され、視聴者に配信されます。 インジェストサーバーは、取り込みプロトコル (RTMP 、 RTMPS) とともに チャンネル のストリームを受信する特定の IVS コンポーネントです。チャンネルの作成については、「 IVS 低レイテンシーストリーミングの開始 」を参照してください。		✓	
インターレースビデオ	後続のフレームの奇数行または偶数行のみを送信して表示し、余分な帯域幅を消費せずに、体感 フレームレート を倍増させます。ビデオ品質の問題から、インターレースビデオの使用はお勧めしません。	✓	✓	

言葉	説明	LL	RT	チャット
JSON	JavaScript オブジェクト表記とは、人が読み取れるテキストを使用して、属性と値のペアと配列データ型またはその他の直列化可能な値で構成されるデータオブジェクトを送信する、オープンな標準ファイル形式。	✓	✓	✓
キーフレーム、デルタフレーム、キーフレーム間隔	キーフレーム (イントラコードまたは i フレームとも呼ばれます) は、動画内の画像のフルフレームです。後続のフレームであるデルタフレーム (予測フレームまたは P フレームとも呼ばれます) には、変更された情報のみが含まれます。キーフレームは、エンコーダーで定義されているキーフレーム間隔に応じて、 ストリーム 内に複数回表示されます。	✓	✓	
Lambda	サーバーインフラストラクチャをプロビジョニングせずにコード (Lambda 関数と呼ばれる) を実行するための AWS サービス。Lambda 関数は、イベントや呼び出しリクエストに応答して実行することも、スケジュールに基づいて実行することもできます。たとえば、IVS Chat は Lambda 関数を使用して チャットルームのメッセージレビュー を有効にします。	✓	✓	✓
レイテンシー、グラストウグラスのレイテンシー	<p>データ転送の遅延。IVS はレイテンシー範囲を次のように定義しています。</p> <ul style="list-style-type: none"> 低レイテンシー: 3 秒未満 リアルタイムレイテンシー: 300 ms 未満 <p>グラストウグラスレイテンシーとは、カメラがライブストリームをキャプチャしてから視聴者の画面に表示されるまでの遅延を指します。</p>	✓	✓	

言葉	説明	LL	RT	チャット
サイマルキャストによるレイヤードエンコーディング。	品質レベルの異なる複数のビデオストリームを同時にエンコードして公開できます。「リアルタイムストリーミングによる最適化」の「 アダプティブストリーミング: サイマルキャストによるレイヤードエンコーディング 」。		✓	
メッセージレビューハンドラ	IVS Chat の顧客に、 チャットルーム に配信される前にユーザーチャットメッセージを自動的に確認/フィルタリングする機能を付与します。 Lambda 関数をチャットルームに関連付けることで有効になります。「チャットメッセージレビューハンドラ」の「 Lambda 関数の作成 」を参照してください。			✓
ミキサー	IVS Mobile Broadcast SDK の機能の 1 つとして、複数のオーディオおよびビデオソースを受け取り、単一の出力を生成します。カメラ、マイク、スクリーンキャプチャ、アプリケーションで生成されたオーディオと動画などのソースを表す画面上のビデオとオーディオ要素の管理をサポートします。その後、出力を IVS にストリーミングできます。「IVS Broadcast SDK: Mixer ガイド (低レイテンシーストリーミング)」の「 ミキシング用のブロードキャストセッションの設定 」を参照してください。	✓		

言葉	説明	LL	RT	チャット
マルチホストストリーミング	<p>複数のホストからのストリームを1つのストリームにまとめます。これは、クライアント側またはサーバー側のコンポジションを使用して実現できます。</p> <p>マルチホストストリーミングでは、視聴者をステージに招待して質疑応答、ホスト同士の競争、ビデオチャット、大勢の視聴者の前でホスト同士が会話するなどのシナリオが可能になります。</p>		✓	
マルチバリエーションプレイリスト	ブロードキャストで利用できるすべての バリエーションストリーム のインデックス。	✓		
OAC	オリジンアクセスコントロールは S3 バケットへのアクセスを制限するメカニズムで、記録されたストリームなどのコンテンツを CloudFront CDN 経由でのみ提供できるようにします。	✓		
OBS	オープンブロードキャストソフトウェア (OBS) – 動画の録画とライブストリーミング用のオープンソースの無料ソフトウェア。 OBS はデスクトップパブリッシング用の (IVS ブロードキャスト SDK に代わる) 代替手段を提供します。シーントランジション、オーディオミキシング、オーバーレイグラフィックなどの高度な制作機能を備えているため、OBS に精通している上級ストリーマーには OBS が好ましいかもしれません。	✓	✓	
Participant	ホスト または 視聴者 としてステージに接続したリアルタイムのユーザー。		✓	
参加者トークン	イベント 参加者 が ステージ に参加すると、その参加者をリアルタイムで認証します。参加者トークンは、参加者がステージに動画を送信できるかどうかを制御します。		✓	

言葉	説明	LL	RT	チャット
再生トークン、再生キーペア	<p>顧客がプライベートチャンネルでの動画再生を制限できるようにする認証メカニズム。再生トークンは、再生 キーペアから生成されます。</p> <p>再生キーペアは、再生用の視聴者認証トークンに署名して検証するために使用されるパブリックキーとプライベートキーのペアです。「IVS プライベートチャンネルの設定」の「IVS 再生キーの作成またはインポート」と、「IVS 低レイテンシー API リファレンス」の「再生キーペアのエンドポイント」を参照してください。</p>	✓		
再生 URL	<p>視聴者が特定のチャンネルの再生を開始するために使用するアドレスを識別します。このアドレスはグローバルに使用できます。IVS は、IVS グローバルコンテンツ配信ネットワーク上で最適な場所を自動的に選択し、各視聴者に動画を配信します。チャンネルの作成については、「IVS 低レイテンシー ストリーミングの開始」を参照してください。</p>	✓		
¥プライベートチャンネル	<p>再生トークンに基づく認証メカニズムを使用して、お客様がストリームへのアクセスを制限できるようにします。「IVS プライベートチャンネルの設定」の「IVS プライベートチャンネルのワークフロー」を参照してください。</p>	✓		
プログレッシブビデオ	<p>各フレームのすべての行を順番に送信して表示します。ブロードキャストのすべての段階でプログレッシブビデオを使用することをお勧めします。</p>	✓	✓	

言葉	説明	LL	RT	チャット
クォータ	AWS アカウントの IVS サービスリソースまたはオペレーションの最大数。つまり、これらの制限は、特に断りのない限り、AWS のアカウントごとに適用されます。すべてのクォータはリージョンごとに適用されます。「AWS 一般リファレンスガイド」の「 Amazon インタラクティブビデオサービスのエンドポイントとクォータ 」を参照してください。	✓	✓	✓
リージョン	<p>リージョンを使用すると、特定の地域に物理的に存在する AWS のサービスにアクセスすることができます。リージョンでは耐障害性や安定性が提供され、レイテンシーを低減することもできます。これにより、リージョンの障害の影響を受けずに利用できる冗長リソースを作成できます。</p> <p>ほとんどの AWS サービスのリクエストは特定の地域に関するものです。あるリージョンで作成したリソースは、AWS サービスで提供されるレプリケーション機能を明示的に使用しないかぎり、他のリージョンに存在することはありません。たとえば、Amazon S3 はクロスリージョンのレプリケーションをサポートしています。IAM などの一部のサービスには、リージョン間のリソースがありません。</p>	✓	✓	✓
解決方法	1 つの動画フレームのピクセル数を示します。たとえば、フル HD または 1080p は 1920 x 1080 ピクセルのフレームを定義します。	✓	✓	
ルートユーザー	AWS アカウントの所有者。ルートユーザーは、AWS アカウントのすべての AWS サービスとリソースへの完全なアクセス権を持ちます。	✓	✓	✓

言葉	説明	LL	RT	チャット
RTMP、RTMPS	Real-Time Messaging Protocol。ネットワーク上でオーディオ、動画、データを送信するための業界標準。RTMPS は、Transport Layer Security (TLS/SSL) 接続上で実行される RTMP の安全なバージョンです。	✓	✓	
S3 バケット	Amazon S3 に格納されたオブジェクトのコレクション。アクセスやレプリケーションを含む多くのポリシーがバケットレベルで定義され、バケット内のすべてのオブジェクトに適用されます。たとえば、IVS ブロードキャストは S3 バケット内の複数のオブジェクトとして保存されます。	✓		
SDK	ソフトウェア開発キットは、IVS を使用してアプリケーションを構築する開発者向けのライブラリ集です。	✓	✓	✓
自撮りセグメンテーション	カメラ画像を入力として受け取り、画像の各ピクセルがフォアグラウンドかバックグラウンドかを示す信頼度スコアを提供するマスクを返す、お客様固有のソリューションを使用して、ライブストリームのバックグラウンドを置き換えることができます。「IVS Broadcast SDK: サードパーティーのカメラフィルター (リアルタイムストリーミング)」の「 背景の置換 」を参照してください。		✓	
セマンティックバージョンニング	Major.Minor.Patch 形式のバージョンフォーマット。API に影響しないバグ修正ではパッチバージョンが上がり、下位互換性のある API を追加または変更するとマイナーバージョンが上がり、下位互換性のない API の変更ではメジャーバージョンが上がります。	✓	✓	✓

言葉	説明	LL	RT	チャット
サーバーサイドコンポジション	<p>IVS サーバーを使用してステージ参加者全員からの音声と動画を合成し、IVS チャンネル に送信し、より多くの視聴者や S3 バケット に保存します。サーバーサイドコンポジションにより、クライアントの負荷が軽減され、ブロードキャストの耐性が向上し、帯域幅をより効率的に使用できるようになります。</p> <p>「クライアント側コンポジション」も参照してください。</p>		✓	
Service Quotas	<p>AWS は、多くの AWS サービスの クォータ を 1 か所から管理できるサービスです。クォータ値を確認できるだけでなく、Service Quotas コンソールからクォータの引き上げをリクエストすることもできます。</p>	✓	✓	✓
サービスリンクロール	<p>AWS サービスに直接リンクされた一意のタイプの IAM ロールです。サービスにリンクされたロールは、IVS で自動作成され、サービスがユーザーの代わりに、S3 バケット へのアクセスなど、その他の AWS サービスを呼び出すために必要なすべてのアクセス権限が付与されます。「IVS Security」の IVS の「サービスにリンクされたロールの使用」を参照してください。</p>	✓		
ステージ	<p>リアルタイムのイベント参加者がリアルタイムでビデオを送受信できる仮想スペースを提示する IVS リソース。「IVS リアルタイムストリーミング入門」の「ステージの作成」を参照してください。</p>		✓	

言葉	説明	LL	RT	チャット
ステージセッション	最初の参加者が ステージ に参加したときに始まり、最後の参加者がステージへの公開を停止した数分後に終了します。長期間有効なステージでは、その存続期間中に複数のセッションが発生する場合があります。		✓	
ストリーム	ソースから宛先に継続的に送信される動画またはオーディオコンテンツを表すデータ。	✓	✓	
ストリームキー	チャンネル 作成時に IVS によって割り当てられる識別子で、チャンネルへのストリーミングの認証に使用されます。ストリームキーを使うとすべてのユーザーがチャンネルにストリーミングできるため、ストリームキーは機密情報として扱ってください。「 IVS 低レイテンシーストリーミングを開始する 」を参照してください。	✓		
ストリームスタベーション	IVS へのストリーム配信の遅延または停止。IVS が、エンコーディングデバイスがアダプタイズした想定されたビット数を、一定期間にわたって取り込まなかった場合に発生します。ストリーム不足が発生すると、ストリームスタベーション イベント が発生します。 視聴者の観点から見ると、ストリームスタベーションは、動画における遅延、バッファリング、フリーズとして現れる場合があります。ストリームスタベーションは、ストリーム不足の原因となった特定の状況に応じて、短い (5 秒未満) 場合もあれば、長い (数分) 場合もあります。「 トラブルシューティング FAQ 」の「 ストリームスタベーションとは 」を参照してください。	✓	✓	
ストリーマー	IVS にビデオまたはオーディオ ストリーム を送信するユーザーまたはデバイス。	✓	✓	

言葉	説明	LL	RT	チャット
サブスクライバー	ホストの動画やオーディオを受信するリアルタイムのイベント参加者。「 IVS リアルタイムストリーミングとは 」を参照してください。		✓	
タグ	AWS リソースに割り当てるメタデータラベル。タグを使用すると、AWS リソースの識別や整理ができます。 IVS ドキュメントのランディングページ では、IVS API ドキュメント (リアルタイムストリーミング、低レイテンシーストリーミング、チャット) の「タグ付け」を参照してください。	✓	✓	✓
サードパーティーのカメラフィルター	IVS Broadcast SDK と統合できるソフトウェアコンポーネント。これにより、アプリケーションは画像を カスタム画像ソース として Broadcast SDK に送信する前に処理できます。サードパーティーのカメラフィルターは、カメラからの画像を処理したり、フィルター効果を適用したりできます。	✓	✓	
サムネイル	ストリームから取得した縮小サイズの画像。デフォルトでは、サムネイルは 60 秒ごとに生成されますが、より短い間隔を設定することもできます。サムネイルの解像度は チャンネルタイプ によって異なります。「Amazon S3 への自動記録 (低レイテンシーストリーミング)」の「 コンテンツの記録 」を参照してください。	✓		

言葉	説明	LL	RT	チャット
時間指定メタデータ	<p>ストリーム内の特定のタイムスタンプに関連付けられたメタデータ。IVS API を使用してプログラムで追加でき、特定のフレームに関連付けられます。これにより、すべての視聴者は、ストリームに対してメタデータを同じ時点で受信できます。</p> <p>時限メタデータを使用して、スポーツイベント中にチームの統計を更新するなど、クライアント側でアクションをトリガーできます。「動画ストリーム内にメタデータを埋め込む」を参照してください。</p>	✓		
トランスコーディング	<p>動画とオーディオを、あるフォーマットから別のフォーマットに変換します。入力ストリームが複数のビットレートと解像度が異なるフォーマットにコード変換され、各種の再生デバイスとネットワーク条件をサポートします。</p>	✓	✓	
トランスマックス	<p>ビデオストリームの再エンコードを行わずに、取り込んだストリームを IVS に簡単に再パッケージ化できます。トランスマックスは、「transcode multiplexing」(トランスコード多重化) の略称で、オーディオおよび/または動画ファイルを、元のストリームの一部またはすべてを保持しながら、フォーマット変更するプロセスです。トランスマキシングは、ファイルの内容を変更せずに別のコンテナ形式に変換します。トランスコーディングとは区別されます。</p>	✓	✓	

言葉	説明	LL	RT	チャット
バリエーションストリーム	<p>同じブロードキャストを複数の異なる品質レベルでエンコードしたものです。各バリエーションストリームは個別の HLS プレイリスト としてエンコードされます。利用可能なバリエーションストリームのインデックスは、マルチバリエーションプレイリスト と呼ばれます。</p> <p>IVS プレーヤーは IVS からマルチバリエーションプレイリストを受信すると、再生中にバリエーションストリームの中から選択でき、ネットワークの状況の変化に応じてシームレスに切り替わります。</p>	✓		
VBR	<p>可変ビットレート。必要なディテールのレベルに応じて再生中に変化するダイナミックビットレートを使用するエンコーダのレート制御方法です。動画品質の問題から VBR は使用しないことを強くお勧めします。代わりに CBR を使用してください。</p>	✓	✓	

言葉	説明	LL	RT	チャット
ビュー	<p>アクティブに動画をダウンロードまたは再生している固有の視聴セッション。視聴回数は同時視聴クォータの基準です。</p> <p>視聴セッションの動画再生開始により、視聴が始まります。視聴セッションが動画の再生を停止すると、視聴が終了します。再生は視聴率の唯一の指標です。オーディオレベル、ブラウザのタブフォーカス、動画の品質などのエンゲージメントヒューリスティクスは考慮されません。IVS は視聴回数をカウントする際、個々の視聴者の正当性を考慮せず、また、1 台のマシンに複数の動画プレーヤーがあるなど、ローカライズされた視聴者の重複排除をしません。「Service Quotas (低レイテンシーストリーミング)」の「その他のクォータ」を参照してください。</p>	✓		
表示者	IVS から ストリーム を受信しているユーザ。	✓		
WebRTC	<p>ウェブリアルタイムコミュニケーションは、ウェブブラウザとモバイルアプリケーションにリアルタイムのコミュニケーションを提供するオープンソースプロジェクトです。直接ピアツーピア通信が可能になるため、ウェブページ内でオーディオや動画の通信が可能になり、プラグインをインストールしたりネイティブアプリをダウンロードしたりする必要がなくなります。</p> <p>WebRTC の背後にある技術はオープンなウェブ規格として実装されており、すべての主要なブラウザで通常の JavaScript API として、または Android や iOS などのネイティブクライアント用のライブラリとして利用できます。</p>	✓	✓	

言葉	説明	LL	RT	チャット
WHIP	<p>WebRTC-HTTP Ingestion Protocol。ストリーミングサービスや CDN へのコンテンツの WebRTC ベースの 取り込み を可能にする HTTP ベースの プロトコルです。WHIP は、WebRTC 取り込みを標準化するために開発された IETF ドラフトです。</p> <p>WHIP は OBS などのソフトウェアとの互換性を可能にし、デスクトップ公開用の (IVS Broadcast SDK との) 代替を提供します。シーントランジション、オーディオミキシング、オーバーレイグラフィックなどの高度な制作機能を備えているため、OBS に精通している上級ストリーマーには OBS が好ましいかもしれません。</p> <p>WHIP は、IVS Broadcast SDK の使用が実行可能でない場合や推奨されない場合にも有益です。例えば、ハードウェアエンコーダーを含むセットアップでは、IVS Broadcast SDK は使用できない場合があります。ただし、エンコーダーが WHIP をサポートしている場合でも、エンコーダーから IVS に直接公開できます。</p> <p>「OBS and WHIP Support」を参照してください。</p>		✓	
WSS	<p>WebSocket Secure は、暗号化された TLS 接続を介して WebSockets を確立するためのプロトコルです。IVS Chat エンドポイントへの接続に使用されています。「IVS Chat の開始方法」の「ステップ 4: 最初のメッセージの送受信」を参照してください。</p>			✓

IVS Chat ドキュメント履歴

チャットのユーザーガイドの変更点

変更	説明	日付
チャット UG を分割する	<p>このリリースに伴い、ドキュメントに大幅な変更が行われました。チャット情報を IVS 低レイテンシーストリーミングユーザーガイドから、IVS ドキュメントのランディングページの既存の IVS Chat セクションにある新しい IVS Chat ユーザーガイドに移動しました。</p> <p>その他のドキュメントの変更については、「ドキュメント履歴 (低レイテンシーストリーミング)」を参照してください。</p>	2023 年 12 月 28 日
IVS の用語集	用語集を拡張し、IVS のリアルタイム、低レイテンシー、チャット用語を網羅しました。	2023 年 12 月 20 日

IVS チャット API リファレンスの変更点

API の変更	説明	日付
チャット UG を分割する	IVS チャットのユーザーガイド (このリリースで作成) ができたので、今後は既存の IVS Chat API Reference と IVS Chat Messaging API Reference のドキュメ	2023 年 12 月 28 日

API の変更	説明	日付
	<p>ント履歴エントリはこちらに移動しました。これらのチャット API リファレンスの以前の履歴エントリは、Document History (Low-Latency Streaming) にあります。</p>	

IVS Chat リリースノート

2023 年 12 月 28 日

Amazon IVS Chat ユーザーガイド

Amazon Interactive Video Service (IVS) Chat は、ライブ動画ストリームに付随するマネージド型のライブチャット機能です。このリリースでは、「IVS Low-Latency Streaming User Guide」から新しい「IVS Chat User Guide」にチャット情報を移動しました。ドキュメントは、[Amazon IVS ドキュメントのランディングページ](#)からアクセスできます。

2023 年 1 月 31 日

Amazon IVS Chat Client Messaging SDK: Android 1.1.0

プラットフォーム	ダウンロードおよび変更
Android Chat Client Messaging SDK 1.1.0	<p>リファレンスドキュメント: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/</p> <ul style="list-style-type: none"> Kotlin コルーチンをサポートするために、新しい IVS Chat Messaging API を <code>com.amazonaws.ivs.chat.messaging.coroutines</code> パッケージに追加しました。新しい Kotlin コルーチンのチュートリアルも参照してください。パート 1 (パートは全部で 2 つあります) は「チャットルーム」です。

Chat Client Messaging SDK サイズ: Android

アーキテクチャ	圧縮サイズ	非圧縮サイズ
すべてのアーキテクチャ (バイトコード)	89 KB	92 KB

2022 年 11 月 9 日

Amazon IVS Chat Client Messaging SDK: JavaScript 1.0.2

プラットフォーム	ダウンロードおよび変更
JavaScript Chat Client Messaging SDK 1.0.2	<p>リファレンスドキュメント: https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/</p> <ul style="list-style-type: none"> Firefox に影響する問題を修正しました。クライアントが DisconnectUser エンドポイントによりチャットルームから切断された場合に、誤ってソケットエラーを受信する問題が修正されました。

2022 年 9 月 8 日

Amazon IVS Chat Client Messaging SDK: Android 1.0.0 および iOS 1.0.0

プラットフォーム	ダウンロードおよび変更
Android Chat Client Messaging SDK 1.0.0	リファレンスドキュメント: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/
iOS Chat Client Messaging SDK 1.0.0	リファレンスドキュメント: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/

Chat Client Messaging SDK サイズ: Android

アーキテクチャ	圧縮サイズ	非圧縮サイズ
すべてのアーキテクチャ (バイトコード)	53 KB	58 KB

Chat Client Messaging SDK サイズ: iOS

アーキテクチャ	圧縮サイズ	非圧縮サイズ
ios-arm64_x86_64-simulator (ビットコード)	484 KB	2.4 MB
ios-arm64_x86_64-simulator	484 KB	2.4 MB
ios-arm64 (ビットコード)	1.1 MB	3.1 MB
ios-arm64	233 KB	1.2 MB