Chat-Benutzerhandbuch

Amazon IVS

Amazon IVS: Chat-Benutzerhandbuch

Table of Contents

Was ist Amazon IVS Chat?

Amazon IVS Chat ist ein verwaltetes Live-Chat-Feature für Live-Videostreams. Die Dokumentation ist über die Zielseite für Amazon-IVS-Dokumentation im Abschnitt zu Amazon IVS Chat verfügbar:

- Chat-Benutzerhandbuch – Dieses Dokument zusammen mit allen anderen Seiten des Benutzerhandbuchs, die im Navigationsbereich aufgeführt sind.
- Chat-API-Referenz – Steuerebene-API (HTTPS).
- Chat-Messaging-API-Referenz – Datenebene-API (WebSocket).
- SDK-Referenzen für Chat-Clients: Android, iOS und JavaScript.

Erste Schritte mit Amazon IVS Chat

Amazon Interactive Video Service (IVS) Chat ist ein verwaltetes Live-Chat-Feature, die neben Ihren Live-Videostreams ausgeführt werden kann. (IVS Chat kann auch ohne Videostream verwendet werden.) Sie können Chatrooms erstellen und Chat-Sitzungen zwischen Ihren Benutzern aktivieren.

Mit Amazon IVS Chat können Sie sich neben Live-Videos darauf konzentrieren, maßgeschneiderte Chat-Erlebnisse zu erstellen. Sie müssen keine Infrastruktur verwalten oder Komponenten Ihrer Chat-Workflows entwickeln und konfigurieren. Amazon IVS Chat ist skalierbar, sicher, zuverlässig und kosteneffektiv.

Amazon IVS Chat funktioniert am besten, um Nachrichten zwischen Teilnehmern eines Live-Videostreams mit einem Anfang und einem Ende zu erleichtern.

Der Rest dieses Dokuments führt Sie durch die Schritte zum Erstellen Ihrer ersten Chat-Anwendung mit Amazon IVS Chat.

Beispiele: Die folgenden Demo-Apps sind verfügbar (drei Beispiel-Client-Apps und eine Backend-Server-App zur Token-Erstellung):

- Amazon-IVS-Chat-Webdemo
- Amazon-IVS-Chat für Android-Demo
- Amazon-IVS-Chat für iOS-Demo
- Amazon-IVS-Chat-Demo-Backend

Wichtig: Chatrooms, die 24 Monate lang keine neuen Verbindungen oder Updates haben, werden automatisch gelöscht.

Themen

Schritt 1: Durchführen der Ersteinrichtung

Bevor Sie fortfahren, müssen Sie folgende Aufgaben durchführen:

1. Erstellen Sie ein AWS-Konto.
2. Richten Sie Root-Benutzer und Administratoren ein.
3. Einrichten von AWS-IAM-Berechtigungen (Identity and Access Management). Verwenden Sie die unten angegebene Richtlinie.

Spezifische Schritte für die oben aufgeführten Aufgaben finden Sie unter Erste Schritte mit IVS Streaming mit niedriger Latenz im Benutzerhandbuch zu Amazon IVS. Wichtig: Verwenden Sie unter „Schritt 3: Einrichten von IAM-Berechtigungen“ diese Richtlinie für IVS Chat:

JSON

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

Schritt 2: Erstellen eines Chatrooms

Ein Amazon-IVS-Chatroom enthält damit verknüpfte Konfigurationsinformationen (z. B. maximale Nachrichtenlänge).

Die Anweisungen in diesem Abschnitt zeigen Ihnen, wie Sie die Konsole oder die AWS-CLI verwenden, um Chaträume einzurichten (einschließlich der optionalen Einrichtung für die Überprüfung von Nachrichten und/oder die Protokollierung von Nachrichten) und Räume zu erstellen.

Konsolenanweisungen zum Erstellen eines IVS-Chatrooms

Diese Schritte sind in Phasen unterteilt, die mit der ersten Einrichtung des Raums beginnen und mit der endgültigen Raumerstellung enden.

Optional können Sie einen Raum einrichten, damit Nachrichten überprüft werden. Sie können beispielsweise Nachrichteninhalt oder Metadaten aktualisieren, Nachrichten ablehnen, um zu verhindern, dass sie gesendet werden, oder die ursprüngliche Nachricht durchlassen. Weitere Informationen hierzu finden Sie unter Einrichten, um Raumnachrichten zu überprüfen (optional).

Außerdem können Sie einen Chatroom optional so einrichten, dass Nachrichten protokolliert werden. Wenn Nachrichten beispielsweise an einen Chatroom gesendet werden, können Sie sie in einem Amazon-S3-Bucket, in Amazon CloudWatch oder in Amazon Kinesis Data Firehose protokollieren. Weitere Informationen hierzu finden Sie unter Einrichtung für die Protokollierung von Nachrichten (optional).

Erstes Einrichten eines Raums

1. Öffnen Sie die Amazon-IVS-Chat-Konsole.

   (Sie können auf die Amazon-IVS-Konsole auch über die AWS-Managementkonsole zugreifen.)
2. Verwenden Sie auf der Navigationsleiste im Dropdown-Menü Auswählen einer Region, um eine Region auszuwählen. Ihr neuer Raum wird in dieser Region erstellt.
3. Wählen Sie im Feld Erste Schritte (oben rechts) Amazon-IVS-Chatroom aus. Das Fenster Raum erstellen wird angezeigt.

Create room Info

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

How Amazon IVS Chat works

Setup

Room name – *optional*

name

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

Room configuration

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

Message character limit Info

500 characters per message

Maximum message rate Info

10 messages per second

Message review handler Info

Review messages before they are sent to the room

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

Message logging Info

Automatically log chat messages

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

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

Tags Info

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

Cancel Create chat room

4. Unter Einrichtung geben Sie optional einen Raumnamen an. Raumnamen sind nicht eindeutig, aber sie bieten Ihnen eine Möglichkeit, andere Räume als den Raum-ARN (Amazon Resource Name) zu unterscheiden.
5. Unter Einrichten > Raumkonfiguration akzeptieren Sie entweder die Standardkonfiguration oder wählen Sie Benutzerdefinierte Konfiguration und konfigurieren dann die Maximale Nachrichtenlänge und/oder die Maximale Nachrichtenrate.
6. Wenn Sie Nachrichten überprüfen möchten, fahren Sie mit Einrichten, um Raum-Nachrichten zu überprüfen (optional) unten fort. Andernfalls überspringen Sie diesen Schritt (d. h. Sie akzeptieren die Einstellung Nachrichtenrezension-Handler > Deaktiviert) und fahren direkt mit Abschließende Raumerstellung fort.

Einrichten, um Raumnachrichten zu überprüfen (optional)

1. Unter Nachrichtenrezension-Handler wählen Sie Mit AWS Lambda umgehen aus. Der Bereich Message Review Handler (Nachrichtenüberprüfungs-Handler) wird erweitert, um zusätzliche Optionen anzuzeigen.
2. Konfigurieren Sie das Fallback-Ergebnis so, dass die Nachricht zugelassen oder abgelehnt wird, wenn der Handler keine gültige Antwort zurückgibt, ein Fehler auftritt oder die Leerlaufzeitlimit überschritten wird.
3. Geben Sie Ihre bestehende Lambda-Funktion an oder verwenden Sie Lambda-Funktion erstellen, um eine neue Funktion zu erstellen.

   Die Lambda-Funktion muss sich in derselben AWS-Region und demselben AWS-Konto wie der Client-VPN-Endpunkt befinden. Sie sollten dem Amazon-Chat-SDK-Service die Berechtigung erteilen, Ihre Lambda-Ressource aufzurufen. Die ressourcenbasierte Richtlinie wird automatisch für die von Ihnen ausgewählte Lambda-Funktion erstellt. Weitere Informationen zu Berechtigungen finden Sie unter Ressourcenbasierte Richtlinie für Amazon IVS Chat.

Einrichtung für die Protokollierung von Nachrichten (optional)

1. Wählen Sie unter Message logging (Nachrichtenprotokollierung) die Option Automatically log chat messages (Chatnachrichten automatisch protokollieren) aus. Der Bereich Message logging (Nachrichtenprotokollierung) wird erweitert, um zusätzliche Optionen anzuzeigen. Sie können diesem Chatroom entweder eine vorhandene Protokollierungskonfiguration hinzufügen oder eine neue erstellen, indem Sie Create logging configuration (Protokollierungskonfiguration erstellen) auswählen.

2. Wenn Sie eine vorhandene Protokollierungskonfiguration auswählen, wird ein Dropdown-Menü mit allen bereits erstellten Protokollierungskonfigurationen angezeigt. Wählen Sie eine Konfiguration in der Liste aus. Ihre Chatnachrichten werden dann automatisch am betreffenden Ziel protokolliert.
3. Wenn Sie Create logging configuration (Protokollierungskonfiguration erstellen) auswählen, wird ein Dialogfenster gezeigt, in dem Sie eine neue Protokollierungskonfiguration erstellen und anpassen können.
   a. Geben Sie optional einen Logging configuration name (Namen für die Protokollierungskonfiguration) an. Namen von Protokollierungskonfigurationen sind genau wie die Namen von Chatrooms nicht eindeutig, bieten Ihnen jedoch eine Möglichkeit, Protokollierungskonfigurationen von der Protokollierungskonfiguration mit dem ARN zu unterscheiden.
   b. Wählen Sie unter Destination (Ziel) die Option CloudWatch log group (CloudWatch-Protokollgruppe), Kinesis Firehose Delivery Stream (Kinesis-Firehose-Bereitstellungs-Stream) oder Amazon S3 bucket (Amazon-S3-Bucket) aus, um das Ziel für die Protokolle auszuwählen.
   c. Wählen Sie je nach Ziel die Option zum Erstellen einer neuen CloudWatch log group (CloudWatch-Protokollgruppe), eines neuen Kinesis firehose delivery stream (Kinesis-Firehose-Bereitstellungs-Stream) oder eines neuen Amazon S3 bucket (Amazon-S3-Bucket) bzw. zum Verwenden einer vorhandenen Gruppe, eines vorhandenen Streams oder eines vorhandenen Buckets aus.
   d. Wählen Sie nach der Überprüfung die Option Create (Erstellen) aus, um eine neue Protokollierungskonfiguration mit einer eindeutigen ARN zu erstellen. Dadurch wird die neue Protokollierungskonfiguration automatisch an den Chatroom angefügt.

Abschließende Raumerstellung

1. Wählen Sie nach der Überprüfung die Option Create chat room (Chatroom erstellen) aus, um einen neuen Chatroom mit einer eindeutigen ARN zu erstellen.

CLI-Anweisungen zum Erstellen eines IVS-Chatrooms

Dieses Dokument führt Sie durch die Schritte zur Integration des Amazon-IVS-Chatrooms mit der AWS CLI.

Erstellen eines Chatrooms

Das Erstellen eines Chatrooms mit der AWS CLI ist eine erweiterte Option und erfordert, dass Sie zuerst die CLI auf Ihrem Computer herunterladen und konfigurieren. Informationen zu den ersten Schritten finden Sie im Benutzerhandbuch für die AWS-Befehlszeilenschnittstelle.

1. Führen Sie den Chat-`create-room`-Befehl aus und und übergeben Sie einen optionalen Namen:

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

2. Dies gibt einen neuen Chatroom zurück:

```
{
   "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. Beachten Sie das Feld `arn`. Sie benötigen dies, um ein Client-Token zu erstellen und eine Verbindung mit einem Chatroom herzustellen.

Einrichten einer Protokollierungskonfiguration (optional)

Das Erstellen eines Chatrooms mit der AWS-CLI ist eine erweiterte Option. Dazu müssen Sie zunächst die CLI auf Ihrem Computer herunterladen und konfigurieren. Informationen zu den ersten Schritten finden Sie im Benutzerhandbuch für die AWS-Befehlszeilenschnittstelle.

1. Führen Sie den Chatbefehl `create-logging-configuration` aus und übergeben Sie einen optionalen Namen und eine Zielkonfiguration, die auf einen Amazon-S3-Bucket verweisen. Dieser Amazon-S3-Bucket muss vor Erstellung der Protokollierungskonfiguration vorhanden sein. (Einzelheiten zum Erstellen eines Amazon-S3-Buckets finden Sie in der Dokumentation zu Amazon S3.)

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

```
   --name "test-logging-config"
```

2. Hierdurch wird eine neue Protokollierungskonfiguration zurückgegeben:

```
{
   "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. Beachten Sie das Feld `arn`. Dieses wird benötigt, um die Protokollierungskonfiguration an den Chatroom anzufügen.

   a. Wenn Sie einen neuen Chatroom erstellen, führen Sie den Befehl `create-room` aus und übergeben Sie die Protokollierungskonfiguration `arn`:

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

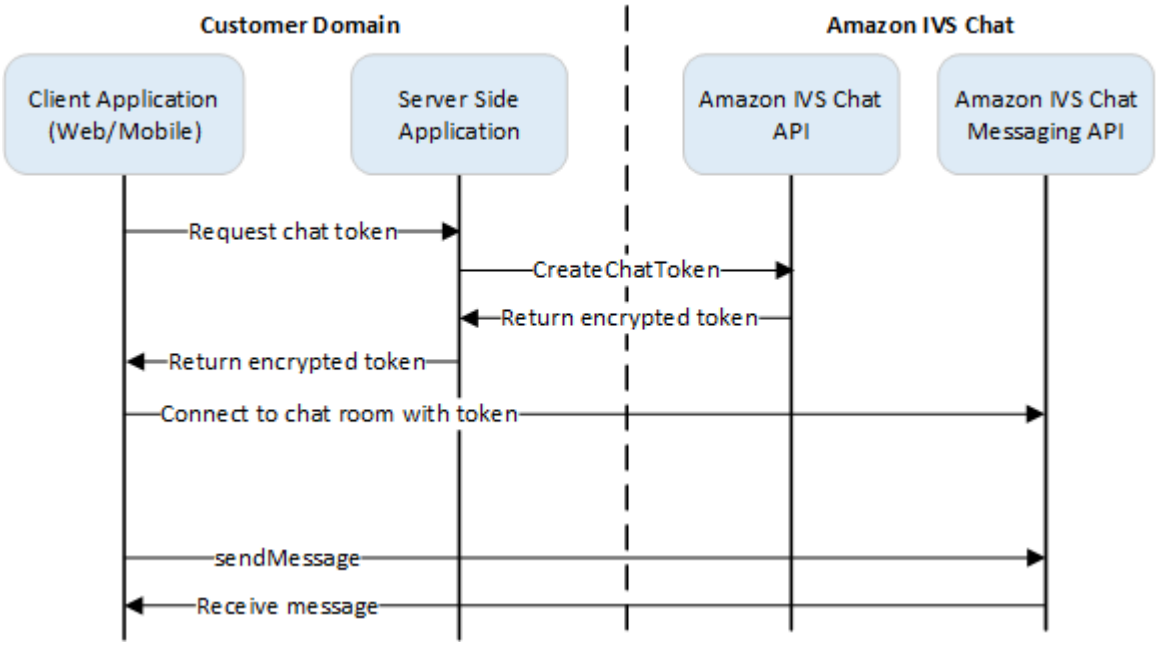
   b. Wenn Sie einen vorhandenen Chatroom aktualisieren, führen Sie den Befehl `update-room` aus und übergeben Sie die Protokollierungskonfiguration `arn`:

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

Schritt 3: Erstellen eines Chat-Tokens

Damit ein Chat-Teilnehmer eine Verbindung zu einem Raum herstellen und Nachrichten senden und empfangen kann, muss ein Chat-Token erstellt werden. Chat-Token werden verwendet, um Chat-Clients zu authentifizieren und zu autorisieren.

Dieses Diagramm veranschaulicht den Arbeitsablauf für die Erstellung eines IVS-Chat-Tokens:



Wie oben gezeigt, fragt eine Client-Anwendung Ihre serverseitige Anwendung nach einem Token, und die serverseitige Anwendung ruft CreateChatToken mithilfe eines AWS-SDK oder signierter Sigv4-Anfragen auf. Da AWS-Anmeldeinformationen zum Aufrufen der API verwendet werden, sollte das Token in einer sicheren serverseitigen Anwendung generiert werden, nicht in der clientseitigen Anwendung.

Eine Backend-Serveranwendung, die die Token-Generierung demonstriert, ist im Amazon-IVS-Chat-Demo-Backend verfügbar.

Sitzungsdauer bezieht sich darauf, wie lange eine etablierte Sitzung aktiv bleiben kann, bevor sie automatisch beendet wird. Das heißt, die Sitzungsdauer entspricht der Zeit, die der Client mit dem Chatroom verbunden bleiben kann, bevor ein neues Token generiert und eine neue Verbindung hergestellt werden muss. Während der Token-Erstellung können Sie wahlweise die Sitzungsdauer angeben.

Jedes Token kann nur einmal zum Herstellen einer Verbindung für einen Endbenutzer verwendet werden. Wenn eine Verbindung geschlossen wird, muss ein neues Token erstellt werden, bevor eine Verbindung wiederhergestellt werden kann. Das Token selbst ist bis zu dem in der Antwort enthaltenen Zeitstempel für den Ablauf des Tokens gültig.

Wenn ein Endbenutzer eine Verbindung zu einem Chatroom herstellen möchte, sollte der Client die Serveranwendung nach einem Token fragen. Die Serveranwendung erstellt ein Token und gibt es an den Client zurück. Tokens sollten auf Anfrage für Endbenutzer erstellt werden.

Befolgen Sie die Anweisungen unten, um ein Chat-Authentifizierungstoken zu erstellen. Wenn Sie ein Chat-Token erstellen, verwenden Sie die Anforderungsfelder, um Daten über den Chat-Endbenutzer und die Messaging-Funktionen des Endbenutzers zu übergeben. Einzelheiten finden Sie unter CreateChatToken in der Referenz für IVS-Chat-API.

AWS-SDK-Anweisungen

Das Erstellen eines Chat-Tokens mit dem AWS SDK erfordert, dass Sie zuerst das SDK in Ihre Anwendung herunterladen und konfigurieren. Im Folgenden finden Sie Anweisungen für das AWS SDK, das JavaScript verwendet.

Wichtig: Dieser Code muss serverseitig ausgeführt und seine Ausgabe an den Client übergeben werden.

Voraussetzung: Um das folgende Codebeispiel verwenden zu können, müssen Sie das AWS JavaScript SDK in Ihre Anwendung laden. Weitere Informationen dazu finden Sie unter Erste Schritte mit AWS SDK for JavaScript.

```
async function createChatToken(params) {
  const ivs = new AWS.Ivschat();
  const result = await ivs.createChatToken(params).promise();
  console.log("New token created", result.token);
}
/*
Create a token with provided inputs. Values for user ID and display name are
from your application and refer to the user connected to this chat session.
*/
const params = {
  "attributes": {
    "displayName": "DemoUser",
  }",
  "capabilities": ["SEND_MESSAGE"],
  "roomIdentifier": "arn:aws:ivschat:us-west-2:123456789012:room/g1H2I3j4k5L6",
  "userId": 11231234
};
createChatToken(params);
```

CLI-Anweisungen

Das Erstellen eines Chat-Tokens mit der AWS CLI ist eine erweiterte Option und erfordert, dass Sie zuerst die CLI auf Ihrem Computer herunterladen und konfigurieren. Informationen zu den

ersten Schritten finden Sie im Benutzerhandbuch für die AWS-Befehlszeilenschnittstelle. Hinweis: Das Generieren von Token mit der AWS CLI eignet sich für Testzwecke, aber für die Produktion empfehlen wir Ihnen, Token auf Serverseite mit dem AWS SDK zu generieren (siehe Anweisungen oben).

1. Führen Sie den `create-chat-token`-Befehl zusammen mit Raumkennung und Benutzer-ID für den Client aus. Fügen Sie eine der folgenden Funktionen ein: `"SEND_MESSAGE"`, `"DELETE_MESSAGE"`, `"DISCONNECT_USER"`. (Fügen Sie optional die Sitzungsdauer (in Minuten) und/oder benutzerdefinierte Attribute (Metadaten) zu dieser Chat-Sitzung ein. Diese Felder werden unten nicht angezeigt.)

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

2. Dies gibt ein Client-Token zurück:

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

3. Speichern Sie dieses Token. Sie benötigen es, um eine Verbindung mit dem Chatroom herzustellen und Nachrichten zu senden oder zu empfangen. Sie müssen ein weiteres Chat-Token generieren, bevor Ihre Sitzung endet (wie durch `sessionExpirationTime` angegeben).

Schritt 4: Senden und Empfangen Ihrer ersten Nachricht

Verwenden Sie Ihr Chat-Token, um eine Verbindung zu einem Chatroom herzustellen und Ihre erste Nachricht zu senden. Nachstehend finden Sie einen beispielhaften JavaScript-Code. IVS-Client-SDKs sind ebenfalls verfügbar: siehe Chat-SDK: Handbuch für Android, Chat-SDK: Handbuch für iOS und Chat-SDK: Handbuch für JavaScript.

Regionaler Service: Der folgende Beispielcode bezieht sich auf Ihre „unterstützte Region Ihrer Wahl“. Amazon IVS Chat bietet regionale Endpunkte, über die Sie Ihre Anforderungen stellen können. Für die Amazon-IVS-Chat-Messaging-API lautet die allgemeine Syntax eines regionalen Endpunkts:

wss://edge.ivschat.<Regionscode>.amazonaws.com

Zum Beispiel ist wss://edge.ivschat.us-west-2.amazonaws.com der Endpunkt in der Region USA West (Oregon). Eine Liste der unterstützten Regionen finden Sie in den Amazon-IVS-Chat-Informationen auf der Amazon-IVS-Seite in der Allgemeinen AWS-Referenz.

```
/*
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);
}
```

Herzlichen Glückwunsch, Sie sind fertig! Sie haben jetzt eine einfache Chat-Anwendung, die Nachrichten senden oder empfangen kann.

Schritt 5: Überprüfen Ihre Service-Quota-Limits (optional)

Ihre Chatrooms werden zusammen mit Ihrem Amazon-IVS-Live-Stream skaliert, damit alle Ihre Zuschauer Chat-Gespräche führen können. Alle Amazon-IVS-Konten haben jedoch Beschränkungen für die Anzahl der gleichzeitigen Chat-Teilnehmer und die Rate der Nachrichtenzustellung.

Stellen Sie sicher, dass Ihre Limits angemessen sind und fordern Sie bei Bedarf eine Erhöhung an, insbesondere wenn Sie ein großes Streaming-Event planen. Weitere Informationen finden Sie unter Service Quotas (Streaming mit niedriger Latenz), Service Quotas (Echtzeit-Streaming) und Service Quotas (Chat).

IVS-Chat-Protokollierung

Mit der Chatprotokollierung können Sie alle Chatnachrichten aus einem Chatroom an einem von drei Standardspeicherorten aufzeichnen: in einem Amazon-S3-Bucket, in Amazon CloudWatch Logs oder in Amazon Kinesis Data Firehose. Anschließend können die Protokolle zur Analyse oder zur Erstellung einer Chatwiedergabe genutzt werden, die mit einer Live-Videositzung verknüpft ist.

Aktivieren der Chatprotokollierung für einen Chatroom

Bei der Chatprotokollierung handelt es sich um eine erweiterte Option, die aktiviert werden kann, indem eine Protokollierungskonfiguration einem Chatroom zugeordnet wird. Eine Protokollierungskonfiguration ist eine Ressource, mit der Sie einen Speicherort angeben können (Amazon-S3-Bucket, Amazon CloudWatch Logs oder Amazon Kinesis Data Firehose), an dem Nachrichten eines Chatrooms protokolliert werden. Einzelheiten zum Erstellen und Verwalten von Protokollierungskonfigurationen finden Sie unter Erste Schritte mit Amazon IVS Chat und Referenz zur Amazon-IVS-Chat-API.

Jedem Chatroom können Sie bis zu drei Protokollierungskonfigurationen zuordnen, entweder beim Erstellen eines neuen Chatrooms (CreateRoom) oder beim Aktualisieren eines vorhandenen (UpdateRoom). Sie können mehrere Chatrooms derselben Protokollierungskonfiguration zuordnen.

Wenn mindestens eine aktive Protokollierungskonfiguration einem Chatroom zugeordnet ist, werden alle Messaging-Anforderungen, die über die Amazon-IVS-Chat-Messaging-API an diesen Chatroom gesendet werden, automatisch an den angegebenen Speicherorten aufgezeichnet. Nachfolgend sind die durchschnittlichen Übertragungsverzögerungen aufgeführt (vom Senden einer Messaging-Anforderung bis zum Zeitpunkt, zu dem sie an den angegebenen Speicherorten verfügbar ist):

- Amazon-S3-Bucket: 5 Minuten
- Amazon CloudWatch Logs oder Amazon Kinesis Data Firehose: 10 Sekunden

Nachrichteninhalt

Format

```
{
   "event_timestamp": "string",
   "type": "string",
```

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

Felder

Feld	Beschreibung
`event_timestamp`	UTC-Zeitstempel für den Empfang der Nachricht durch Amazon IVS Chat.
`payload`	Die JSON-Nutzlast für Message (Subscribe) (Nachricht (Abonnieren)) oder Event (Subscribe) (Ereignis (Abonnieren)), die Clients vom Amazon-IVS-Chat-Service erhalten.
`type`	Typ der Chatnachricht. • Zulässige Werte: `MESSAGE` | `EVENT`
`version`	Version des Formats des Nachrichteninhalts.

Amazon S3 Bucket

Format

Nachrichtenprotokolle werden mit dem folgenden S3-Präfix und -Dateiformat organisiert und gespeichert:

```
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
```

Felder

Feld	Beschreibung
`<account_id>`	ID des AWS-Kontos, in dem der Chatroom erstellt wird.

Feld	Beschreibung
`<hash>`	Ein vom System generierter Hashwert zur Gewährleistung der Eindeutigkeit.
`<region>`	Die AWS-Serviceregion, in der der Chatroom erstellt wurde.
`<resource_id>`	Die Ressourcen-ID, die Bestandteil des Chatroom-ARN ist.
`<version>`	Version des Formats des Nachrichteninhalts.
`<year> / <month> / <day> / <hours> / <minute>`	UTC-Zeitstempel für den Empfang der Nachricht durch Amazon IVS Chat.

Beispiel

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

Amazon CloudWatch Logs

Format

Nachrichtenprotokolle werden im folgenden Namensformat für Protokollstreams organisiert und gespeichert:

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

Felder

Feld	Beschreibung
`<resource_id>`	Ressourcen-ID, die Bestandteil des Chatroom-ARN ist.

Feld	Beschreibung
`<version>`	Version des Formats des Nachrichteninhalts.

Beispiel

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

Amazon Kinesis Data Firehose

Nachrichtenprotokolle werden als Echtzeit-Streaming-Daten an den Bereitstellungs-Stream gesendet, und zwar an Ziele wie Amazon Redshift, Amazon OpenSearch Service, Splunk und alle benutzerdefinierten HTTP-Endpunkte oder HTTP-Endpunkte im Besitz von unterstützten externen Serviceanbietern. Weitere Informationen finden Sie unter Was ist Amazon Kinesis Data Firehose?

Einschränkungen

- Sie müssen Eigentümer des Protokollierungsspeicherorts sein, an dem Nachrichten gespeichert werden.
- Der Chatroom, die Protokollierungskonfiguration und der Protokollierungsspeicherort müssen sich in derselben AWS-Region befinden.
- Für die Chatprotokollierung sind ausschließlich aktive Protokollierungskonfigurationen verfügbar.
- Sie können nur Protokollierungskonfigurationen löschen, die keinem Kanal mehr zugeordnet sind.

Für die Protokollierung von Nachrichten an einem Speicherort in Ihrem Besitz ist die Autorisierung mit Ihren AWS-Anmeldeinformationen erforderlich. Um IVS Chat den erforderlichen Zugriff zu gewähren, wird bei der Erstellung der Protokollierungskonfiguration automatisch eine Ressourcenrichtlinie (für einen Amazon-S3-Bucket oder CloudWatch Logs) oder eine serviceverknüpfte Rolle von AWS IAM (für Amazon Kinesis Data Firehose) generiert. Seien Sie vorsichtig bei Änderungen an Rollen oder Richtlinien, da dies Auswirkungen auf die Berechtigung zur Chatprotokollierung haben kann.

Überwachung von Fehlern mit Amazon CloudWatch

Fehler bei der Chatprotokollierung können Sie mit Amazon CloudWatch überwachen. Zudem können Sie Alarme oder Dashboards erstellen, um bestimmte Fehler anzuzeigen oder darauf zu reagieren.

Es gibt mehrere Arten von Fehlern. Weitere Informationen finden Sie unter Überwachung von Amazon IVS Chat.

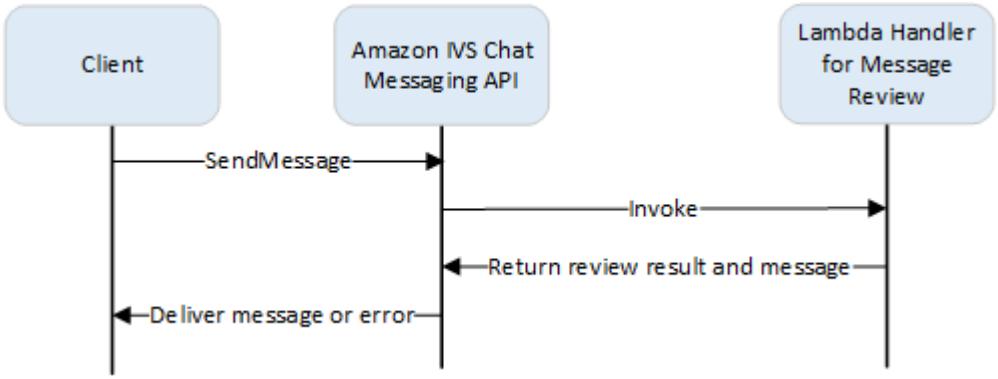
IVS-Chat-Nachrichtenüberprüfungs-Handler

Ein Nachrichtenrezension-Handler ermöglicht es Ihnen, Nachrichten zu überprüfen und/oder zu ändern, bevor sie an einen Raum geliefert werden. Wenn ein Nachrichtenrezension-Handler mit einem Raum verknüpft ist, wird er für jede SendMessage-Anforderung an diesen Raum aufgerufen. Der Handler erzwingt die Geschäftslogik Ihrer Anwendung und bestimmt, ob eine Nachricht zugelassen, verweigert oder geändert werden soll. Amazon IVS Chat unterstützt AWS-Lambda-Funktionen als Handler.

Erstellen einer Lambda-Funktion

Bevor Sie einen Nachrichtenrezension-Handler für einen Raum einrichten, müssen Sie eine Lambda-Funktion mit einer ressourcenbasierten IAM-Richtlinie erstellen. Die Lambda-Funktion muss sich im selben AWS-Konto und in derselben AWS-Region befinden wie der Raum, mit dem Sie die Funktion verwenden. Die ressourcenbasierte Richtlinie gibt Amazon IVS Chat die Berechtigung zum Aufrufen Ihrer Lambda-Funktion. Anweisungen finden Sie unter Ressourcenbasierte Richtlinie für Amazon IVS Chat.

Workflow



Anforderungssyntax

Wenn ein Client eine Nachricht sendet, ruft Amazon IVS Chat die Lambda-Funktion mit einer JSON-Nutzlast auf:

```
{
   "Content": "string",
   "MessageId": "string",
```

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

Anforderungstext

Feld	Beschreibung
`Attributes`	Attribute, die mit dem Ereignis verknüpft sind.
`Content`	Ursprünglicher Inhalt der Nachricht.
`MessageId`	Die Nachrichten-ID. Generiert von IVS Chat.
`RoomArn`	Der ARN des Raums, an den Nachrichten gesendet werden.
`Sender`	Informationen zum Absender. Dieses Objekt hat mehrere Felder: • `Attributes` – Metadaten über den Absender, die während der Authentifizierung erstellt wurden. Diese können verwendet werden, um dem Kunden weitere Informationen über den Absender zu geben, z. B. Avatar-URL, Badges, Schriftart und Farbe. • `UserId` – Eine anwendungsspezifische Kennung des Betrachters (Endbenutzers), der diese Nachricht gesendet hat. Diese kann von der Clientanwendung verwendet werden, um entweder in der Messaging-API oder in den Anwendungsdomänen auf den Benutzer zu verweisen. • `Ip` – Die IP-Adresse des Clients, der die Nachricht sendet.

Antwortsyntax

Die Handler-Lambda-Funktion muss eine JSON-Antwort mit der folgenden Syntax zurückgeben. Antworten, die nicht der folgenden Syntax entsprechen oder die Feldeinschränkungen nicht erfüllen, sind ungültig. In diesem Fall wird die Nachricht je nach `FallbackResult`-Wert, den Sie in Ihrem

Nachrichtenrezension-Handler angeben, zugelassen oder abgelehnt; siehe MessageReviewHandler in der Amazon-IVS-Chat-API-Referenz.

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

Antwortfelder

Feld	Beschreibung
`Attributes`	Attribute, die mit der Nachricht verknüpft sind, die von der Lambda-Funktion zurückgegeben wird. Wenn `ReviewResult` eine `DENY` ist, kann in `Attributes` ein `Reason` angegeben werden; z. B.: `"Attributes": {"Reason": "denied for moderation` In diesem Fall erhält der Absenderclient einen WebSocket 406-Fehler mit dem Grund in der Fehlermeldung. (Siehe WebSocket-Fehler in der Amazon-IVS-Chat-Messaging-API-Referenz.) • Größenbeschränkungen: Maximal 1 KB • Erforderlich: Nein
`Content`	Inhalt der von der Lambda-Funktion zurückgegebenen Nachricht. Er kann je nach Geschäftslogik bearbeitet oder original sein. • Längenbeschränkungen: Minimale Länge beträgt 1 Zeichen. Die maximale Länge der `MaximumMessageLength`, die Sie definiert haben, als Sie den Raum erstellt/aktualisiert haben. Weitere Informationen finden Sie in der Amazon-IVS-Chat-API-Referenz. Dies gilt nur, wenn `ReviewResult` `ALLOW` ist. • Erforderlich: Ja

Feld	Beschreibung
`ReviewResult`	Das Ergebnis der Überprüfungsverarbeitung zum Umgang mit der Nachricht. Falls zugelassen, wird die Nachricht an alle mit dem Raum verknüpften Benutzer übermittelt. Falls verweigert, wird die Nachricht an keinen Benutzer zugestellt. • Zulässige Werte: `ALLOW` | `DENY` • Erforderlich: Ja

Beispiel-Code

Unten ist ein Beispiel für Lambda-Handler in Go. Dabei wird der Nachrichteninhalt geändert, die Nachrichtenattribute werden unverändert beibehalten und die Nachricht wird zugelassen.

```
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
}
```

Verknüpfen und Trennen eines Handlers mit/von einem Raum

Sobald Sie den Lambda-Handler eingerichtet und implementiert haben, verwenden Sie die Amazon-IVS-Chat-API:

- Um den Handler einem Raum zuzuordnen, rufen Sie CreateRoom oder UpdateRoom auf und geben Sie den Handler an.
- Um den Handler von einem Raum zu trennen, rufen Sie UpdateRoom mit einem leeren Wert für `MessageReviewHandler.Uri` auf.

Überwachung von Fehlern mit Amazon CloudWatch

Sie können Fehler bei der Nachrichtenüberprüfung mit Amazon CloudWatch überwachen und Alarme oder Dashboards erstellen, um die Änderungen bestimmter Fehler anzuzeigen oder darauf zu reagieren. Wenn ein Fehler auftritt, wird die Nachricht je nach `FallbackResult`-Wert, den Sie angeben, wenn Sie den Handler einem Raum zuordnen, zugelassen oder verweigert; siehe MessageReviewHandler in der Amazon-IVS-Chat-API-Referenz.

Es gibt mehrere Arten von Fehlern:

- `InvocationErrors` treten auf, wenn Amazon IVS Chat einen Handler nicht aufrufen kann.
- `ResponseValidationErrors` treten auf, wenn ein Handler eine ungültige Antwort zurückgibt.
- AWS-Lambda-`Errors` treten auf, wenn ein Lambda-Handler einen Funktionsfehler zurückgibt, wenn er aufgerufen wurde.

Weitere Informationen zu Aufruffehlern und Fehlern bei der Antwortvalidierung (ausgegeben von Amazon IVS Chat) finden Sie in den Amazon-IVS-Chat-Informationen unter Überwachen von Amazon-IVS-Streaming mit niedriger Latenz. Weitere Informationen zu AWS Lambda-Fehlern finden Sie unter Arbeiten mit Lambda-Metriken.

Überwachen von Amazon IVS Chat

Sie können Ressourcen von Amazon Interactive Video Service (IVS) Chat mit Amazon CloudWatch überwachen. CloudWatch sammelt und verarbeitet Rohdaten von Amazon IVS Chat in lesbare Metriken, die nahezu in Echtzeit zur Verfügung stehen. Diese Statistiken werden 15 Monate lang aufbewahrt, so dass Sie einen historischen Überblick über die Leistung Ihrer Webanwendung oder Ihres Dienstes erhalten können. Sie können Alarme für bestimmte Schwellenwerte einstellen und Benachrichtigungen senden oder Aktionen durchführen, wenn diese Schwellenwerte erreicht werden. Details dazu finden Sie im CloudWatch Benutzerhandbuch.

Zugreifen auf CloudWatch-Metriken

Amazon CloudWatch sammelt und verarbeitet Rohdaten von Amazon IVS Chat in lesbare Metriken, die nahezu in Echtzeit zur Verfügung stehen. Diese Statistiken werden 15 Monate lang aufbewahrt, so dass Sie einen historischen Überblick über die Leistung Ihrer Webanwendung oder Ihres Dienstes erhalten können. Sie können Alarme für bestimmte Schwellenwerte einstellen und Benachrichtigungen senden oder Aktionen durchführen, wenn diese Schwellenwerte erreicht werden. Details dazu finden Sie im CloudWatch Benutzerhandbuch.

Beachten Sie, dass CloudWatch-Metriken im Laufe der Zeit aufgerollt werden. Die Auflösung nimmt effektiv ab, wenn die Metriken altern. Hier ist der Zeitplan:

- Metriken für 60 Sekunden sind 15 Tage lang verfügbar.
- Metriken für 5 Minuten sind 63 Tage lang verfügbar.
- 1-Stunden-Metriken stehen 455 Tage (15 Monate) lang zur Verfügung.

Aktuelle Informationen zur Datenspeicherung finden Sie unter Amazon CloudWatch – Häufig gestellte Fragen.

Anleitung für die CloudWatch-Konsole

1. Öffnen Sie die CloudWatch-Konsole unter https://console.aws.amazon.com/cloudwatch/.
2. Erweitern Sie in der Seitennavigation das Dropdown Metriken und wählen Sie dannAlle Metriken aus.
3. Wählen Sie in der Registerkarte Durchsuchen über das unbeschriftete Dropdown-Menü auf der linken Seite Ihre Heimatregion aus, in der Ihre Kanäle erstellt wurden. Weitere Informationen

zu Regionen finden Sie unter Globale Lösung, regionale Kontrolle. Eine Liste der unterstützten Regionen finden Sie auf der Amazon-IVS-Seite in der Allgemeinen AWS-Referenz.

4. Wählen Sie unten auf der Registerkarte Durchsuchen den IVSChat-Namespace aus.
5. Führen Sie eine der folgenden Aktionen aus:
   a. Geben Sie in der Suchleiste Ihre Ressourcen-ID (Teil der ARN,) ein., `arn:::ivschat:room/<resource id>`).

      Wählen Sie dann IVSChat aus.
   b. Wenn IVSChat als auswählbarer Service unter AWS Namespaces erscheint, wählen Sie ihn aus. Er wird aufgeführt, wenn Sie Amazon IVSChat verwenden und Metriken an Amazon CloudWatch senden. (Wenn IVSChat nicht aufgeführt ist, haben Sie keine Amazon IVSChat-Metriken.)

      Wählen Sie dann nach Bedarf eine Dimensionsgruppierung aus. Die verfügbaren Dimensionen sind unten in CloudWatch-Metriken aufgeführt.
6. Wählen Sie Metriken aus, die dem Diagramm hinzugefügt werden sollen. Verfügbare Metriken sind unten unter CloudWatch-Metriken aufgeführt.

Sie können auch auf der Detailseite der Chat-Sitzung auf das zugehörige CloudWatch-Diagramm zugreifen, indem Sie das Feld In CloudWatch anzeigen auswählen.

CLI-Anweisungen

Sie können auf die Metriken auch über die AWS CLI zugreifen. Dies erfordert, dass Sie zuerst die CLI auf Ihrem Computer herunterladen und konfigurieren. Informationen zu den ersten Schritten finden Sie im Benutzerhandbuch für die AWS-Befehlszeilenschnittstelle.

So greifen Sie dann über die AWS CLI auf Metriken zum Amazon-IVS-Chat-Streaming mit niedriger Latenz zu:

- Führen Sie an der Eingabeaufforderung Folgendes aus:

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

Weitere Informationen finden Sie unter Amazon CloudWatch verwenden im Amazon CloudWatch-Benutzerhandbuch.

CloudWatch-Metriken: IVS-Chat

Amazon IVS Chat bietet die folgenden Metriken im AWS/IVSChat-Namespace.

Metrik	Dimensionen	Beschreibung
`ConcurrentChatConnections`	Keine	Die Gesamtzahl der gleichzeitigen Verbindungen in einem Chatroom (maximal pro Minute gemeldet). Dies ist nützlich, um zu verstehen, wann Kunden ihr Limit für gleichzeitige Chat-Verbindungen in einer Region erreichen. Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum
`Deliveries`	Aktion	Die Anzahl der Lieferungen von Nachrichtenanforderungen eines bestimmten Aktionstypen an Chat-Verbindungen in allen Ihren Räumen in einer Region. Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum
`InvocationErrors`	`Uri`	Die Anzahl der Aufruffehler eines bestimmten Nachrichtenrezension-Handler in allen Ihren Räumen in einer Region. Ein Aufruffehler tritt auf, wenn der Nachrichtenrezension-Handler nicht aufgerufen werden kann. Aufruffehler treten auf, wenn Amazon IVS Chat einen Handler nicht aufrufen kann. Dies kann passieren, wenn der mit einem Raum verknüpfte Handler nicht mehr existiert oder wenn seine Ressourcenrichtlinie es dem Service nicht erlaubt, ihn aufzurufen.

Metrik	Dimensionen	Beschreibung
		Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum
`LogDestinationAccessDeniedError`	`LoggingConfiguration`	Die Anzahl der Zugriffsverweigerungsfehler eines Protokollziels für alle Chatrooms in einer Region. Diese Fehler treten auf, wenn Amazon IVS Chat nicht auf die Zielressource zugreifen kann, die Sie in der Protokollierungskonfiguration angegeben haben. Dies kann passieren, wenn die Richtlinie der Zielressource es dem Service nicht erlaubt, Datensätze zu übergeben. Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum
`LogDestinationErrors`	`LoggingConfiguration`	Die Anzahl der Fehler eines Protokollziels für alle Chatrooms in einer Region. Hierbei handelt es sich um eine aggregierte Metrik, die alle Arten von auftretenden Fehlern umfasst, wenn Amazon IVS Chat keine Protokolle an die Zielressource übermittelt, die Sie in der Protokollierungskonfiguration angegeben haben. Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum

Metrik	Dimensionen	Beschreibung
`LogDestinationResourceNotFoundErrors`	`LoggingConfiguration`	Die Anzahl der Fehler aufgrund nicht gefundener Ressourcen eines Protokollziels für alle Chatrooms in einer Region. Diese Fehler treten auf, wenn Amazon IVS Chat keine Protokolle an eine Zielressource übermitteln kann, die Sie in einer Protokollierungskonfiguration angegeben haben, weil die Ressource nicht vorhanden ist. Dies kann passieren, wenn die Zielressource, die einer Protokollierungskonfiguration zugeordnet ist, nicht mehr existiert. Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum
`MessagingDeliveries`	Keine	Die Anzahl der Lieferungen von Nachrichtenanforderungen an Chat-Verbindungen in allen Ihren Räumen in einer Region. Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum
`MessagingRequests`	Keine	Die Anzahl der Nachrichtenanforderungen in allen Ihren Räumen in einer Region. Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum

Metrik	Dimensionen	Beschreibung
`Requests`	Aktion	Die Anzahl der Anforderungen eines bestimmte n Aktionstyps in allen Ihren Räumen in einer Region. Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum
`ResponseV alidation Errors`	`Uri`	Die Anzahl der Antwortvalidierungsfehler eines bestimmten Nachrichtenrezension-Handlers in allen Ihren Räumen in einer Region. Ein Antwortvalidierungsfehler tritt auf, wenn die Antwort des Nachrichtenrezension-Handle r ungültig ist. Dies kann bedeuten, dass die Antwort nicht analysiert werden konnte oder Validierungsprüfungen fehlschlagen, z. B. bei einem ungültigen Prüfergebnis oder zu langen Antwortwerten. Einheit: Anzahl Gültige Statistiken: Summe, Durchschnitt, Maximum, Minimum

IVS-Chat-Client-Nachrichten-SDK

Das Amazon Interactive Video Services (IVS) Chat Client Messaging SDK ist für Entwickler gedacht, die Anwendungen mit Amazon IVS erstellen. Dieses SDK wurde entwickelt, um die Amazon-IVS-Architektur zu nutzen und bietet neben Amazon IVS Chat Aktualisierungen. Als natives SDK wurde es entwickelt, um die Leistungsauswirkungen auf Ihre Anwendung und auf die Geräte, mit denen Ihre Benutzer auf Ihre Anwendung zugreifen, zu minimieren.

Plattform-Anforderungen

Desktop-Browser

Browser	Unterstützte Versionen
Chrome	Zwei Hauptversionen (aktuelle und neueste Vorversion)
Edge	Zwei Hauptversionen (aktuelle und neueste Vorversion)
Firefox	Zwei Hauptversionen (aktuelle und neueste Vorversion)
Oper	Zwei Hauptversionen (aktuelle und neueste Vorversion)
Safari	Zwei Hauptversionen (aktuelle und neueste Vorversion)

Mobile Browser

Browser	Unterstützte Versionen
Chrome für Android	Zwei Hauptversionen (aktuelle und neueste Vorversion)
Firefox für Android	Zwei Hauptversionen (aktuelle und neueste Vorversion)
Opera für Android	Zwei Hauptversionen (aktuelle und neueste Vorversion)

Browser	Unterstützte Versionen
WebView Android	Zwei Hauptversionen (aktuelle und neueste Vorversion)
Samsung Internet	Zwei Hauptversionen (aktuelle und neueste Vorversion)
Safari für iOS	Zwei Hauptversionen (aktuelle und neueste Vorversion)

Native Plattformen

Plattform	Unterstützte Versionen
Android	5.0 und höher
iOS	13.0 und höher

Support

Wenn in Ihrem Chatroom ein Fehler oder ein anderes Problem auftritt, ermitteln Sie die eindeutige Raumkennung über die IVS-Chat-API (siehe ListRooms).

Teilen Sie diese Chatroom-Kennung dem AWS Support mit. So können sie Informationen erhalten, die Ihnen helfen, Ihr Problem zu beheben.

Hinweis: Siehe Versionshinweise zu Amazon IVS Chat für verfügbare Versionen und behobene Probleme. Aktualisieren Sie gegebenenfalls Ihre Version des SDK, bevor Sie sich an den Support wenden und prüfen Sie, ob das Problem dadurch behoben wird.

Versionsverwaltung

Die Amazon IVS Chat Client Messaging SDKs nutzen die semantische Versionsverwaltung.

Nehmen Sie für diese Diskussion an:

- Die neueste Version ist 4.1.3.

- Die neueste Version der vorherigen Hauptversion ist 3.2.4.
- Die neueste Version 1.x ist 1.5.6.

Rückwärtskompatible neue Funktionen werden als Nebenversionen der neuesten Version hinzugefügt. In diesem Fall wird der nächste Satz neuer Funktionen als Version 4.2.0 hinzugefügt.

Rückwärtskompatible, kleinere Fehlerbehebungen werden als Patch-Releases der neuesten Version hinzugefügt. Hier wird der nächste Satz von kleineren Fehlerbehebungen als Version 4.1.4 hinzugefügt.

Rückwärtskompatible, große Fehlerbehebungen werden unterschiedlich behandelt; diese werden zu mehreren Versionen hinzugefügt:

- Patch-Version der neuesten Version. Hier ist das Version 4.1.4.
- Patch-Version der vorherigen Nebenversion. Hier ist das Version 3.2.5.
- Patch-Version der neuesten Version 1.x. Hier ist das Version 1.5.7.

Wichtige Fehlerbehebungen werden vom Amazon IVS-Produktteam definiert. Typische Beispiele sind kritische Sicherheitsupdates und ausgewählte andere Korrekturen, die für Kunden erforderlich sind.

Hinweis: In den obigen Beispielen werden freigegebene Versionen inkrementiert, ohne dass Zahlen übersprungen werden (z. B. von 4.1.3 auf 4.1.4). In Wirklichkeit können eine oder mehrere Patch-Nummern intern bleiben und nicht veröffentlicht werden, so dass die freigegebene Version von 4.1.3 auf, sagen wir, 4.1.6 steigen könnte.

Außerdem wird Version 1.x bis Ende 2023 unterstützt oder wenn 3.x veröffentlicht wird, je nachdem, was später passiert.

Amazon-IVS-Chat-APIs

Auf der Serverseite (nicht von den SDKs verwaltet) gibt es zwei APIs mit jeweils eigenen Verantwortlichkeiten:

- Datenebene – Die IVS-Chat-Nachrichten-API ist eine WebSocket-API, die für die Verwendung durch Frontend-Anwendungen (iOS, Android, macOS usw.) entwickelt wurde, die von einem tokenbasierten Authentifizierungsschema gesteuert werden. Mit einem zuvor generierten Chat-Token stellen Sie über diese API eine Verbindung zu bereits vorhandenen Chatrooms her.

Die Amazon IVS Chat Client Messaging SDKs betreffen nur die Datenebene. Die SDKs gehen davon aus, dass Sie bereits Chat-Token über Ihr Backend generieren. Es wird davon ausgegangen, dass der Abruf dieser Token von Ihrer Front-End-Anwendung und nicht von den SDKs verwaltet wird.

- Steuerebene – Die IVS-Chat-Steuerebene-API bietet eine eigene Schnittstelle für Ihre eigenen Backend-Anwendungen, um Chatrooms sowie die Benutzer, die ihnen beitreten, zu verwalten und zu erstellen. Stellen Sie sich dies als das Admin-Panel für das Chat-Erlebnis Ihrer App vor, das von Ihrem eigenen Backend verwaltet wird. Es gibt Vorgänge auf Steuerebene, die für die Erstellung des Chat-Tokens verantwortlich sind, das die Datenebene benötigt, um sich bei einem Chatroom zu authentifizieren.

  Wichtig: Die Messaging-SDKs des IVS-Chat-Clients rufen keine Vorgänge der Steuerebene auf. Sie müssen Ihr Backend eingerichtet haben, um Chat-Token für Sie erstellen zu können. Ihre Front-End-Anwendung muss mit Ihrem Backend kommunizieren, um dieses Chat-Token abzurufen.

Client-Messaging-SDK für IVS Chat: Handbuch für Android

Die Amazon Interactive Video (IVS) Chat Client Messaging Android SDK bietet Schnittstellen, mit denen Sie die IVS Chat Messaging API auf Plattformen mit Android integrieren können.

Das Paket `com.amazonaws:ivs-chat-messaging` implementiert die in diesem Dokument beschriebene Schnittstelle.

Aktuelle Version von IVS Chat Client Messaging Android SDK: 1.1.0 (Versionshinweise)

Referenzdokumentation: Informationen zu den wichtigsten Methoden, die im Amazon IVS Chat Client Messaging Android SDK verfügbar sind, finden Sie in der Referenzdokumentation unter: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/

Beispiel-Code: Siehe das Android-Beispiel-Repository auf GitHub: https://github.com/aws-samples/amazon-ivs-chat-for-android-demo

Plattformanforderungen: Android 5.0 (API Level 21) oder höher ist für die Entwicklung erforderlich.

Erste Schritte mit dem IVS Chat Client Messaging Android SDK

Bevor Sie beginnen, sollten Sie mit Erste Schritte mit Amazon IVS Chat vertraut sein.

Hinzufügen des Package

Fügen Sie `com.amazonaws:ivs-chat-messaging` zu Ihren `build.gradle`-Abhängigkeiten hinzu:

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

Hinzufügen von Proguard-Regeln

Fügen Sie die folgenden Einträge zu Ihrer R8/Proguard-Regeldatei hinzu (`proguard-rules.pro`):

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

Einrichten Ihres Backends

Für diese Integration sind Endpunkte auf Ihrem Server erforderlich, die mit der Amazon-IVS-API kommunizieren. Verwenden Sie die offiziellen AWS-Bibliotheken für den Zugriff auf die Amazon-IVS-API von Ihrem Server aus. Diese sind in mehreren Sprachen aus den öffentlichen Paketen zugänglich, z. B. node.js und Java.

Erstellen Sie als Nächstes einen Serverendpunkt, der mit dem Amazon IVS Chat API kommuniziert und ein Token erstellt.

Einrichten einer Serververbindung

Erstellen Sie eine Methode, die `ChatTokenCallback` als Parameter verwendet und ein Chat-Token aus Ihrem Backend abruft. Übergeben Sie das Token an die `onSuccess`-Methode des Rückrufs. Übergeben Sie im Fehlerfall die Ausnahme an die `onError`-Methode des Rückrufs. Dies ist erforderlich, um die `ChatRoom`-Hauptentität im nächsten Schritt zu instanziieren.

Im Folgenden finden Sie Beispielcode, der das Obige mit einem `Retrofit`-Aufruf implementiert.

```
// ...

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

Verwenden des IVS Chat Client Messaging Android SDK

Dieses Dokument führt Sie durch die Schritte zur Integration des Amazon IVS Chat Client Messaging Android SDK.

Initialisieren einer Chatroom-Instance

Erstellen Sie eine Instance der `ChatRoom`-Klasse. Dazu muss `regionOrUrl` übergeben werden, was in der Regel die AWS-Region ist, in der Ihr Chatroom gehostet wird, und `tokenProvider`, die im vorherigen Schritt erstellte Methode zum Abrufen von Token.

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

Erstellen Sie als Nächstes ein Listener-Objekt, das Handler für Chat-bezogene Ereignisse implementiert, und weisen Sie es der `room.listener`-Eigenschaft zu:

```
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

// ...
```

Der letzte Schritt der grundlegenden Initialisierung besteht darin, eine Verbindung zu dem bestimmten Raum herzustellen, indem eine WebSocket-Verbindung hergestellt wird. Rufen Sie dazu die `connect()`-Methode innerhalb der Raum-Instance auf. Wir empfehlen dies in der `onResume()` -Lebenszyklusmethode zu tun, um sicherzustellen, dass eine Verbindung aufrechterhalten wird, wenn Ihre App im Hintergrund fortgesetzt wird.

```
room.connect()
```

Das SDK versucht, eine Verbindung zu einem Chatroom herzustellen, der in dem von Ihrem Server empfangenen Chat-Token codiert ist. Wenn es fehlschlägt, wird versucht, die Verbindung so oft wie in der Raum-Instance angegeben erneut herzustellen.

Durchführen von Aktionen in einem Chatroom

Die `ChatRoom`-Klasse hat Aktionen zum Senden und Löschen von Nachrichten und zum Trennen der Verbindung anderer Benutzer. Diese Aktionen akzeptieren einen optionalen Rückrufparameter, mit dem Sie Benachrichtigungen zur Bestätigung oder Ablehnung von Anfragen erhalten können.

Senden einer Nachricht

Für diese Anfrage muss die `SEND_MESSAGE`-Funktion in Ihrem Chat-Token codiert sein.

So lösen Sie eine Anfrage zum Senden einer Nachricht aus:

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

Um eine Bestätigung/Ablehnung der Anfrage zu erhalten, geben Sie einen Rückruf als zweiten Parameter an:

```
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.
   }
})
```

Löschen einer Nachricht

Für diese Anfrage muss die DELETE_MESSAGE-Funktion in Ihrem Chat-Token codiert sein.

So lösen Sie eine Anfrage zum Löschen einer Nachricht aus:

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

Um eine Bestätigung/Ablehnung der Anfrage zu erhalten, geben Sie einen Rückruf als zweiten Parameter an:

```
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.
   }
})
```

Trennen der Verbindung eines anderen Benutzers

Für diese Anfrage muss die `DISCONNECT_USER`-Funktion in Ihrem Chat-Token codiert sein.

So trennen Sie einen anderen Benutzer zu Moderationszwecken:

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

Um eine Bestätigung/Ablehnung der Anfrage zu erhalten, geben Sie einen Rückruf als zweiten Parameter an:

```
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.
   }
})
```

Trennen der Verbindung zu einem Chatroom

Um Ihre Verbindung zum Chatroom zu trennen, rufen Sie die `disconnect()`-Methode für die Raum-Instance auf:

```
room.disconnect()
```

Da die WebSocket-Verbindung nach kurzer Zeit nicht mehr funktioniert, wenn sich die Anwendung im Hintergrund befindet, empfehlen wir, dass Sie beim Übergang von/in einen Hintergrundstatus die Verbindung manuell herstellen/trennen. Stimmen Sie dazu den `room.connect()`-Aufruf

in der `onResume()`-Lebenszyklusmethode auf Android `Activity` oder `Fragment` auf einen `room.disconnect()`-Aufruf in der `onPause()`-Lebenszyklusmethode ab.

Client-Messaging-SDK für IVS Chat: Android-Tutorial Teil 1: Chaträume

Hierbei handelt es sich um den ersten Teil eines zweiteiligen Tutorials. Sie werden die Grundlagen der Arbeit mit dem SDK für Amazon IVS Chat Messaging kennenlernen, indem Sie eine voll funktionsfähige Android-Anwendung mithilfe der Kotlin-Programmiersprache entwickeln. Wir nennen die App Chatterbox.

Bevor Sie das Modul starten, nehmen Sie sich ein paar Minuten Zeit, um sich mit den Voraussetzungen, den wichtigsten Konzepten hinter Chat-Token und dem Backend-Server vertraut zu machen, der für die Erstellung von Chaträumen erforderlich ist.

Diese Tutorials sind für erfahrene Android-Entwickler gedacht, die das IVS Chat Messaging SDK noch nicht kennen. Sie müssen mit der Programmiersprache Kotlin und der Erstellung von Benutzeroberflächen auf der Android-Plattform vertraut sein.

Der vorliegende erste Teil des Tutorials ist in mehrere Abschnitte unterteilt:

1. the section called “Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers”
2. the section called “Erstellen eines Chatterbox-Projekts”
3. the section called “Mit einem Chatraum verbinden und Verbindungsupdates beobachten”
4. the section called “Erstellen eines Token-Anbieters”
5. the section called “Nächste Schritte”

Umfassende Informationen zum SDK finden Sie im Client-Messaging-SDK für Amazon IVS Chat (im vorliegenden Benutzerhandbuch zu Amazon IVS Chat) und unter Chat Client Messaging: SDK für Android-Referenz auf GitHub.

Voraussetzungen

- Sie sind vertraut mit Kotlin und der Erstellung von Anwendungen auf der Android-Plattform. Wenn Sie mit der Erstellung von Anwendungen für Android nicht vertraut sind, können Sie die Grundlagen im Leitfaden Erstellen Ihrer ersten App für Android-Entwickler erlernen.
- Lesen und verstehen Sie Erste Schritte mit IVS Chat gründlich.

- Erstellen Sie einen AWS-IAM-Benutzer mit den Fähigkeiten `CreateChatToken` und `CreateRoom`, die in einer vorhandenen IAM-Richtlinie definiert sind. (Siehe). Erste Schritte mit IVS Chat.)
- Stellen Sie sicher, dass die Geheim-/Zugriffsschlüssel für diesen Benutzer in einer Datei mit den AWS-Anmeldeinformationen gespeichert sind. Entsprechende Anweisungen finden Sie im Benutzerhandbuch zur AWS-CLI (insbesondere unter Einstellungen für Konfigurations- und Anmeldeinformationsdateien).
- Erstellen Sie einen Chatroom und speichern Sie dessen ARN. Siehe Erste Schritte mit IVS Chat. (Wenn Sie den ARN nicht speichern, können Sie ihn später über die Konsole oder die Chat-API nachschlagen.)

Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers

Ihr Backend-Server ist sowohl für die Erstellung von Chaträumen als auch für die Generierung der Chat-Token verantwortlich, die das SDK von IVS Chat Android für die Authentifizierung und Autorisierung Ihrer Kunden in Ihren Chaträumen benötigt.

Weitere Informationen finden Sie unter Erstellen eines Chat-Tokens unter Erste Schritte mit Amazon IVS Chat. Wie im dortigen Flussdiagramm gezeigt, erfolgt die Erstellung eines Chat-Tokens in Ihrem serverseitigen Code. Das bedeutet, dass Ihre App eigene Mittel zur Generierung eines Chat-Tokens bereitstellen muss, indem sie ein Token von der serverseitigen Anwendung anfordert.

Mit dem Ktor-Framework erstellen wir einen lokalen Live-Server, der die Erstellung von Chat-Token mithilfe Ihrer lokalen AWS-Umgebung verwaltet.

Zu diesem Zeitpunkt gehen wir davon aus, dass Sie Ihre AWS-Anmeldeinformationen korrekt eingerichtet haben. Schritt für Schritt Informationen dazu finden Sie unter Einrichten der AWS-Anmeldeinformationen und -Region für die Entwicklung.

Erstellen Sie ein neues Verzeichnis mit dem Namen `chatterbox` und darin ein weiteres Verzeichnis mit dem Namen `auth-server`.

Unser Server-Ordner hat die folgende Struktur:

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

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

Hinweis: Sie können den Code hier direkt in die referenzierten Dateien kopieren/einfügen.

Als Nächstes fügen wir alle notwendigen Abhängigkeiten und Plugins hinzu, damit unser Authentifizierungsserver funktioniert:

Kotlin-Skript:

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

Jetzt müssen wir die Protokollierungsfunktion für den Authentifizierungsserver einrichten. (Weitere Informationen finden Sie unter Logger konfigurieren.)

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>
```

Der Ktor-Server benötigt Konfigurationseinstellungen, die er automatisch aus der `application.*`-Datei im `resources`-Verzeichnis lädt, also fügen wir diese ebenfalls hinzu. (Weitere Informationen finden Sie unter Konfigurierung in einer Datei.)

HOCON:

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

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

Lassen Sie uns abschließend unseren Server implementieren:

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()
            )
         )
      }
   }
```

```
}
```

Erstellen eines Chatterbox-Projekts

Um ein Android-Projekt zu erstellen, installieren und öffnen Sie Android Studio.

Folgen Sie den Schritten, die in der offiziellen Android-Anleitung zum Erstellen eines Projekts aufgeführt sind.

- Wählen Sie unter Projekttyp wählen die Projektvorlage Aktivität leeren für unsere Chatterbox-App aus.
- Wählen Sie unter Projekt konfigurieren die folgenden Werte für Konfigurationsfelder aus:
  - Name: My App
  - Paketname: com.chatterbox.myapp
  - Speicherort: Zeigt auf das im vorherigen Schritt erstellte `chatterbox`-Verzeichnis
  - Sprache: Kotlin
  - API-Mindestlevel: API 21: Android 5.0 (Lollipop)

Nachdem Sie alle Konfigurationsparameter korrekt angegeben haben, sollte unsere Dateistruktur im `chatterbox`-Ordner wie folgt aussehen:

```
- 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
```

Jetzt, da wir ein funktionierendes Android-Projekt haben, können wir com.amazonaws:ivs-chat-messaging zu unseren `build.gradle`-Abhängigkeiten hinzufügen. (Weitere Informationen zum Gradle-Build-Toolkit finden Sie unter Ihren eigenen Build konfigurieren.)

Hinweis: Am Anfang jedes Codeausschnitts befindet sich ein Pfad zu der Datei, in der Sie Änderungen an Ihrem Projekt vornehmen sollten. Der Pfad ist relativ zur Root des Projekts.

Ersetzen Sie im folgenden Code *<version>* durch die aktuelle Versionsnummer des Chat Android SDKs (z. B. 1.0.0).

Kotlin:

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

plugins {
// ...
}

android {
// ...
}

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

Nachdem die neue Abhängigkeit hinzugefügt wurde, führen Sie Projekt mit Gradle-Dateien synchronisieren in Android Studio aus, um das Projekt mit der neuen Abhängigkeit zu synchronisieren. (Weitere Informationen finden Sie unter Build-Abhängigkeiten hinzufügen.)

Um unseren Authentifizierungsserver (der im vorherigen Abschnitt erstellt wurde) bequem vom Projektstammverzeichnis aus ausführen zu können, fügen wir ihn als neues Modul in `settings.gradle` hinzu. (Weitere Informationen finden Sie unter Strukturierung und Aufbau einer Softwarekomponente mit Gradle.)

Kotlin-Skript:

```
// ./settings.gradle

// ...

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

Von nun an, da `auth-server` im Android-Projekt enthalten ist, können Sie den Authentifizierungsserver mit dem folgenden Befehl aus dem Stammverzeichnis des Projekts starten:

Shell:

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

Mit einem Chatraum verbinden und Verbindungsupdates beobachten

Um eine Chatraum-Verbindung zu öffnen, verwenden wir onCreate() activity lifecycle callback, welcher ausgelöst wird, wenn die Aktivität zum ersten Mal erstellt wird. Für den ChatRoom-Constructor müssen wir `region` und `tokenProvider` bereitstellen, um eine Raumverbindung zu instanziieren.

Hinweis: Die `fetchChatToken`-Funktion im folgenden Ausschnitt wird im nächsten Abschnitt implementiert.

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)
   }

// ...
}
```

Das Anzeigen und Reagieren auf Änderungen in der Verbindung eines Chatraums sind wesentliche Bestandteile einer Chat-App wie `chatterbox`. Bevor wir anfangen können, mit dem Raum zu interagieren, müssen wir die Verbindungsstatus-Ereignisse des Chat-Raums abonnieren, um Aktualisierungen zu erhalten.

ChatRoom erwartet, dass wir eine ChatRoomListener-Schnittstellenimplementierung zum Auslösen von Lebenszyklusereignissen anhängen. Für den Moment protokollieren Listener-Funktionen nur Bestätigungsnachrichten, wenn diese aufgerufen werden:

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")
        }
    }
}
```

Nun da wir `ChatRoomListener` implementiert haben, hängen wir es an unsere Raum-Instance an:

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 {
```

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

Danach müssen wir die Möglichkeit bieten, den Raum-Verbindungsstatus zu lesen. Wir behalten sie in der Eigenschaft bei `MainActivity.kt` und initialisieren sie auf den Standardzustand DISCONNECTED für Räume (siehe `ChatRoom state` in der Referenz zu IVS Chat Android SDK). Um den lokalen Status auf dem neuesten Stand zu halten, müssen wir eine Statusaktualisierungsfunktion implementieren; nennen wir sie `updateConnectionState`:

Kotlin:

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

package com.chatterbox.myapp
// ...

enum class ConnectionState {
   CONNECTED,
   DISCONNECTED,
   LOADING
}

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

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

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

Als Nächstes integrieren wir unsere Statusaktualisierungsfunktion in die Eigenschaft ChatRoom.listener:

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)
         }
      }
   }
}
```

Da wir nun die Möglichkeit haben, den ChatRoom-Status zu speichern, abzuhören und auf Aktualisierungen zu reagieren, ist es an der Zeit, eine Verbindung zu initialisieren:

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

Erstellen eines Token-Anbieters

Es ist an der Zeit, eine Funktion zu erstellen, die für die Erstellung und Verwaltung von Chat-Token in unserer Anwendung verantwortlich ist. In diesem Beispiel verwenden wir den Retrofit-HTTP-Client für Android.

Bevor wir Netzwerkverkehr senden können, müssen wir eine Netzwerksicherheitskonfiguration für Android einrichten. (Weitere Informationen finden Sie unter Konfiguration der Netzwerksicherheit.) Wir beginnen mit dem Hinzufügen von Netzwerkberechtigungen zur App-Manifest-Datei. Beachten Sie das hinzugefügte Tag `user-permission` und das hinzugefügte Attribut `networkSecurityConfig`, die auf unsere neue Netzwerksicherheitskonfiguration verweisen. Ersetzen Sie im folgenden Code *<version>* durch die aktuelle Versionsnummer des Chat Android SDKs (z. B. 1.0.0).

XML:

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

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

// ./app/build.gradle


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

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

Deklarieren Sie die Domains `10.0.2.2` und `localhost` als vertrauenswürdig, um mit dem Nachrichtenaustausch mit unserem Backend zu beginnen:

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>
```

Als Nächstes müssen wir eine neue Abhängigkeit sowie Gson converter addition für die Analyse von HTTP-Antworten hinzufügen. Ersetzen Sie im folgenden Code *<version>* durch die aktuelle Versionsnummer des Chat Android SDKs (z. B. 1.0.0).

Kotlin-Skript:

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

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

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

Um ein Chat-Token abzurufen, müssen wir eine POST-HTTP-Anfrage von unserer `chatterbox`-App aus stellen. Wir definieren die Anfrage in einer Schnittstelle, die Retrofit implementieren soll. (Siehe Retrofit-Dokumentation. Machen Sie sich außerdem mit der Spezifikation des Vorgangs CreateChatToken vertraut.)

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>
}
```

Jetzt, da das Netzwerk eingerichtet ist, ist es an der Zeit, eine Funktion hinzuzufügen, die für die Erstellung und Verwaltung unseres Chat-Tokens verantwortlich ist. Wir fügen sie zu `MainActivity.kt` hinzu, das bei der Generierung des Projekts automatisch erstellt wurde:

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)
         }
      })
   }
}
```

Nächste Schritte

Nachdem Sie nun eine Chatraum-Verbindung hergestellt haben, fahren Sie mit Teil 2 dieses Android-Tutorials fort: Nachrichten und Ereignisse.

Client-Messaging-SDK für IVS Chat: Tutorial Teil 2: Nachrichten und Ereignisse

Der vorliegende zweite (und letzte) Teil des Tutorials ist in mehrere Abschnitte unterteilt:

1. the section called “Erstellen Sie eine Benutzeroberfläche für das Senden von Nachrichten.”
   a. the section called “Benutzeroberfläche Hauptlayout”
   b. the section called “Abstrahierte Benutzeroberflächen-Textzelle zur konsistenten Anzeige von Text”
   c. the section called “Benutzeroberfläche linke Chat-Nachricht”
   d. the section called “Benutzeroberfläche rechte Chat-Nachricht”
   e. the section called “Benutzeroberfläche zusätzliche Farbwerte”
2. the section called “View Binding anwenden”
3. the section called “Chat-Nachrichtenanfragen verwalten”
4. the section called “Letzte Schritte”

Umfassende Informationen zum SDK finden Sie im Client-Messaging-SDK für Amazon IVS Chat (im vorliegenden Benutzerhandbuch zu Amazon IVS Chat) und unter Chat Client Messaging: SDK für Android-Referenz auf GitHub.

Voraussetzung

Absolvieren Sie unbedingt Teil 1 dieses Tutorials: Chatrooms.

Erstellen Sie eine Benutzeroberfläche für das Senden von Nachrichten.

Nachdem wir die Chatraum-Verbindung erfolgreich initialisiert haben, ist es an der Zeit, unsere erste Nachricht zu senden. Für dieses Feature wird eine Benutzeroberfläche benötigt. Wir werden folgendes hinzufügen:

- `connect/disconnect`-Schaltfläche
- Nachrichteneingabe mit `send`-Schaltfläche
- Dynamische Nachrichtenliste. Um dies zu erstellen, verwenden wir das Android Jetpack RecyclerView.

Benutzeroberfläche Hauptlayout

Siehe Android Jetpack Layouts in der Android-Entwicklerdokumentation.

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>
```

Abstrahierte Benutzeroberflächen-Textzelle zur konsistenten Anzeige von Text

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>
```

Benutzeroberfläche linke Chat-Nachricht

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>
```

Benutzeroberfläche rechte Chat-Nachricht

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>
```

Benutzeroberfläche zusätzliche Farbwerte

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>
```

View Binding anwenden

Wir nutzen das Android View-Binding-Feature, um Bindungsklassen für unser XML-Layout referenzieren zu können. Um das Feature zu aktivieren, setzen Sie die `viewBinding`-Build-Option auf `true` in `./app/build.gradle`:

Kotlin-Skript:

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

android {
//    ...

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

Jetzt ist es an der Zeit, die Benutzeroberfläche mit unserem Kotlin-Code zu verbinden:

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)
    }
}
```

Wir fügen auch Methoden hinzu, um Nachrichten zu löschen und Benutzer vom Chat zu trennen. Diese können über das Kontextmenü von Chat-Nachrichten aufgerufen werden:

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

Chat-Nachrichtenanfragen verwalten

Wir benötigen eine Möglichkeit, unsere Chat-Nachrichtenanfragen in all ihren möglichen Zuständen zu verwalten:

- Ausstehend – Eine Nachricht wurde an einen Chatraum gesendet, wurde aber noch nicht bestätigt oder abgelehnt.
- Bestätigt – Der Chatraum hat eine Nachricht an alle Benutzer (einschließlich uns) gesendet.
- Abgelehnt – Eine Nachricht wurde vom Chatraum mit einem Fehlerobjekt abgelehnt.

Wir werden noch nicht aufgelöste Chat-Anfragen und Chat-Nachrichten in der Liste speichern. Die Liste benötigt eine separate Klasse, die wir nennen `ChatEntries.kt`:

Kotlin:

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

package com.chatterbox.myapp

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

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

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

    val size get() = entries.size

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

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

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

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

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

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

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

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

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

Um unsere Liste mit der Benutzeroberfläche zu verbinden, verwenden wir einen Adapter. Weitere Informationen finden Sie unter Binden an Daten mit AdapterView und generierte Bindungsklassen.

Kotlin:

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

package com.chatterbox.myapp

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


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

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

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

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

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

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

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

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

```
                viewHolder.failedMark.isGone = true

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

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

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

            is ChatEntry.PendingRequest -> {

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

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

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

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

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

Letzte Schritte

Es ist Zeit, unseren neuen Adapter zu verbinden, der die `ChatEntries`-Klasse an `MainActivity` bindet:

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
        }
    }
}
```

Da wir bereits eine Klasse haben, die dafür verantwortlich ist, unsere Chat-Anfragen zu verfolgen (`ChatEntries`), sind wir bereit, Code für die Manipulation von `entries` zu `roomListener` zu implementieren. Wir werden `entries` und `connectionState` entsprechend des Ereignisses, auf das wir reagieren, aktualisieren:

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")
        }
    }
}
```

Jetzt sollten Sie Ihre Anwendung ausführen können! (Siehe Erstellen und ausführen Ihrer App.) Denken Sie daran, dass Ihr Backend-Server in Betrieb sein muss, wenn Sie die App verwenden. Sie können ihn mit diesem Befehl vom Terminal im Stammverzeichnis unseres Projekts aus starten: `./gradlew :auth-server:run` oder indem Sie die `auth-server:run`-Gradle-Aufgabe direkt von Android Studio aus ausführen.

Client-Messaging-SDK für IVS Chat: Tutorial für Kotlin-Coroutines, Teil 1: Chaträume

Hierbei handelt es sich um den ersten Teil eines zweiteiligen Tutorials. Sie werden die Grundlagen der Arbeit mit dem SDK für Amazon IVS Chat Messaging kennenlernen, indem Sie eine voll funktionsfähige Android-Anwendung mithilfe der Kotlin-Programmiersprache und Coroutines entwickeln. Wir nennen die App Chatterbox.

Bevor Sie das Modul starten, nehmen Sie sich ein paar Minuten Zeit, um sich mit den Voraussetzungen, den wichtigsten Konzepten hinter Chat-Token und dem Backend-Server vertraut zu machen, der für die Erstellung von Chaträumen erforderlich ist.

Diese Tutorials sind für erfahrene Android-Entwickler gedacht, die das IVS Chat Messaging SDK noch nicht kennen. Sie müssen mit der Programmiersprache Kotlin und der Erstellung von Benutzeroberflächen auf der Android-Plattform vertraut sein.

Der vorliegende erste Teil des Tutorials ist in mehrere Abschnitte unterteilt:

1. the section called “Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers”
2. the section called “Erstellen eines Chatterbox-Projekts”
3. the section called “Mit einem Chatraum verbinden und Verbindungsupdates beobachten”
4. the section called “Erstellen eines Token-Anbieters”

5. the section called “Nächste Schritte”

Umfassende Informationen zum SDK finden Sie im Client-Messaging-SDK für Amazon IVS Chat (im vorliegenden Benutzerhandbuch zu Amazon IVS Chat) und unter Chat Client Messaging: SDK für Android-Referenz auf GitHub.

Voraussetzungen

- Sie sind vertraut mit Kotlin und der Erstellung von Anwendungen auf der Android-Plattform. Wenn Sie mit der Erstellung von Anwendungen für Android nicht vertraut sind, können Sie die Grundlagen im Leitfaden Erstellen Ihrer ersten App für Android-Entwickler erlernen.
- Lesen Sie sich Erste Schritte mit IVS Chat durch.
- Erstellen Sie einen AWS-IAM-Benutzer mit den Fähigkeiten `CreateChatToken` und `CreateRoom`, die in einer vorhandenen IAM-Richtlinie definiert sind. (Siehe). Erste Schritte mit IVS Chat.)
- Stellen Sie sicher, dass die Geheim-/Zugriffsschlüssel für diesen Benutzer in einer Datei mit den AWS-Anmeldeinformationen gespeichert sind. Entsprechende Anweisungen finden Sie im Benutzerhandbuch zur AWS-CLI (insbesondere unter Einstellungen für Konfigurations- und Anmeldeinformationsdateien).
- Erstellen Sie einen Chatroom und speichern Sie dessen ARN. Siehe Erste Schritte mit IVS Chat. (Wenn Sie den ARN nicht speichern, können Sie ihn später über die Konsole oder die Chat-API nachschlagen.)

Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers

Ihr Backend-Server ist sowohl für die Erstellung von Chaträumen als auch für die Generierung der Chat-Token verantwortlich, die das SDK von IVS Chat Android für die Authentifizierung und Autorisierung Ihrer Kunden in Ihren Chaträumen benötigt.

Weitere Informationen finden Sie unter Erstellen eines Chat-Tokens unter Erste Schritte mit Amazon IVS Chat. Wie im dortigen Flussdiagramm gezeigt, erfolgt die Erstellung eines Chat-Tokens in Ihrem serverseitigen Code. Das bedeutet, dass Ihre App eigene Mittel zur Generierung eines Chat-Tokens bereitstellen muss, indem sie ein Token von der serverseitigen Anwendung anfordert.

Mit dem Ktor-Framework erstellen wir einen lokalen Live-Server, der die Erstellung von Chat-Token mithilfe Ihrer lokalen AWS-Umgebung verwaltet.

Zu diesem Zeitpunkt gehen wir davon aus, dass Sie Ihre AWS-Anmeldeinformationen korrekt eingerichtet haben. Schritt-für-Schritt-Informationen dazu finden Sie unter Einrichten der temporären AWS-Anmeldeinformationen und der AWS-Region für die Entwicklung.

Erstellen Sie ein neues Verzeichnis mit dem Namen `chatterbox` und darin ein weiteres Verzeichnis mit dem Namen `auth-server`.

Unser Server-Ordner hat die folgende Struktur:

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

Hinweis: Sie können den Code hier direkt in die referenzierten Dateien kopieren/einfügen.

Als Nächstes fügen wir alle notwendigen Abhängigkeiten und Plugins hinzu, damit unser Authentifizierungsserver funktioniert:

Kotlin-Skript:

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

Jetzt müssen wir die Protokollierungsfunktion für den Authentifizierungsserver einrichten. (Weitere Informationen finden Sie unter Logger konfigurieren.)

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>
```

Der Ktor-Server benötigt Konfigurationseinstellungen, die er automatisch aus der `application.*`-Datei im `resources`-Verzeichnis lädt, also fügen wir diese ebenfalls hinzu. (Weitere Informationen finden Sie unter Konfigurierung in einer Datei.)

HOCON:

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

ktor {
    deployment {
        port = 3000
```

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

Lassen Sie uns abschließend unseren Server implementieren:

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()
                )
            )
        }
    }
}
```

Erstellen eines Chatterbox-Projekts

Um ein Android-Projekt zu erstellen, installieren und öffnen Sie Android Studio.

Folgen Sie den Schritten, die in der offiziellen Android-Anleitung zum Erstellen eines Projekts aufgeführt sind.

- Wählen Sie unter Projekt auswählen die Projektvorlage Leere Aktivität für unsere Chatterbox-App aus.

- Wählen Sie unter Projekt konfigurieren die folgenden Werte für Konfigurationsfelder aus:
  - Name: My App
  - Paketname: com.chatterbox.myapp
  - Speicherort: Zeigt auf das im vorherigen Schritt erstellte `chatterbox`-Verzeichnis
  - Sprache: Kotlin
  - API-Mindestlevel: API 21: Android 5.0 (Lollipop)

Nachdem Sie alle Konfigurationsparameter korrekt angegeben haben, sollte unsere Dateistruktur im `chatterbox`-Ordner wie folgt aussehen:

```
- 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
```

Jetzt, da wir ein funktionierendes Android-Projekt haben, können wir com.amazonaws:ivs-chat-messaging und org.jetbrains.kotlinx:kotlinx-coroutines-core zu unseren `build.gradle`-Abhängigkeiten hinzufügen. (Weitere Informationen zum Gradle-Build-Toolkit finden Sie unter Ihren eigenen Build konfigurieren.)

Hinweis: Am Anfang jedes Codeausschnitts befindet sich ein Pfad zu der Datei, in der Sie Änderungen an Ihrem Projekt vornehmen sollten. Der Pfad ist relativ zur Root des Projekts.

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'

// ...
}
```

Nachdem die neue Abhängigkeit hinzugefügt wurde, führen Sie Projekt mit Gradle-Dateien synchronisieren in Android Studio aus, um das Projekt mit der neuen Abhängigkeit zu synchronisieren. (Weitere Informationen finden Sie unter Build-Abhängigkeiten hinzufügen.)

Um unseren Authentifizierungsserver (der im vorherigen Abschnitt erstellt wurde) bequem vom Projektstammverzeichnis aus ausführen zu können, fügen wir ihn als neues Modul in `settings.gradle` hinzu. (Weitere Informationen finden Sie unter Strukturierung und Aufbau einer Softwarekomponente mit Gradle.)

Kotlin-Skript:

```
// ./settings.gradle

// ...

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

Von nun an, da `auth-server` im Android-Projekt enthalten ist, können Sie den Authentifizierungsserver mit dem folgenden Befehl aus dem Stammverzeichnis des Projekts starten:

Shell:

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

Mit einem Chatraum verbinden und Verbindungsupdates beobachten

Um eine Chatraum-Verbindung zu öffnen, verwenden wir onCreate() activity lifecycle callback, welcher ausgelöst wird, wenn die Aktivität zum ersten Mal erstellt wird. Für den ChatRoom-Constructor müssen wir `region` und `tokenProvider` bereitstellen, um eine Raumverbindung zu instanziieren.

Hinweis: Die `fetchChatToken`-Funktion im folgenden Ausschnitt wird im nächsten Abschnitt implementiert.

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)
   }

// ...
}
```

Das Anzeigen und Reagieren auf Änderungen in der Verbindung eines Chatraums sind wesentliche Bestandteile einer Chat-App wie `chatterbox`. Bevor wir anfangen können, mit dem Raum zu interagieren, müssen wir die Verbindungsstatus-Ereignisse des Chat-Raums abonnieren, um Aktualisierungen zu erhalten.

`ChatRoom` erwartet im Chat-SDK für Coroutine, dass wir die Lebenszyklusereignisse von Räumen in Flow behandeln. Für den Moment protokollieren Funktionen nur Bestätigungsnachrichten, wenn diese aufgerufen werden:

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")
                }
            }
        }
    }
}
```

Danach müssen wir die Möglichkeit bieten, den Raum-Verbindungsstatus zu lesen. Wir behalten sie in der `MainActivity.kt`-Eigenschaft bei und initialisieren sie auf den Standardzustand DISCONNECTED für Räume (siehe `ChatRoom` `state` in der IVS-Chat-Android-SDK-Referenz). Um den lokalen Status auf dem neuesten Stand zu halten, müssen wir eine Statusaktualisierungsfunktion implementieren; nennen wir sie `updateConnectionState`:

Kotlin:

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

package com.chatterbox.myapp
// ...

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

// ...

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

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

Als Nächstes integrieren wir unsere Statusaktualisierungsfunktion in die Eigenschaft ChatRoom.listener:

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)

                }
            }

      // ...

      }
   }
}
```

Da wir nun die Möglichkeit haben, den ChatRoom-Status zu speichern, abzuhören und auf Aktualisierungen zu reagieren, ist es an der Zeit, eine Verbindung zu initialisieren:

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)
      }
```

```
    }

    // ...
}
```

Erstellen eines Token-Anbieters

Es ist an der Zeit, eine Funktion zu erstellen, die für die Erstellung und Verwaltung von Chat-Token in unserer Anwendung verantwortlich ist. In diesem Beispiel verwenden wir den Retrofit-HTTP-Client für Android.

Bevor wir Netzwerkverkehr senden können, müssen wir eine Netzwerksicherheitskonfiguration für Android einrichten. (Weitere Informationen finden Sie unter Konfiguration der Netzwerksicherheit.) Wir beginnen mit dem Hinzufügen von Netzwerkberechtigungen zur App-Manifest-Datei. Beachten Sie das hinzugefügte Tag `user-permission` und das hinzugefügte Attribut `networkSecurityConfig`, die auf unsere neue Netzwerksicherheitskonfiguration verweisen. Ersetzen Sie im folgenden Code *<version>* durch die aktuelle Versionsnummer des Chat Android SDKs (z. B. 1.1.0).

XML:

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

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

// ./app/build.gradle


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

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

Deklarieren Sie Ihre lokale IP-Adresse, z. B. die Domains `10.0.2.2` und `localhost` als vertrauenswürdig, um mit dem Nachrichtenaustausch mit unserem Backend zu beginnen:

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>
```

Als Nächstes müssen wir eine neue Abhängigkeit sowie Gson converter addition für die Analyse von HTTP-Antworten hinzufügen. Ersetzen Sie im folgenden Code *<version>* durch die aktuelle Versionsnummer des Chat Android SDKs (z. B. 1.1.0).

Kotlin-Skript:

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

Um ein Chat-Token abzurufen, müssen wir eine POST-HTTP-Anfrage von unserer `chatterbox`-App aus stellen. Wir definieren die Anfrage in einer Schnittstelle, die Retrofit implementieren soll. (Siehe Retrofit-Dokumentation. Machen Sie sich außerdem mit der Spezifikation des Vorgangs CreateChatToken vertraut.)

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)
   }
}
```

Jetzt, da das Netzwerk eingerichtet ist, ist es an der Zeit, eine Funktion hinzuzufügen, die für die Erstellung und Verwaltung unseres Chat-Tokens verantwortlich ist. Wir fügen sie zu `MainActivity.kt` hinzu, das bei der Generierung des Projekts automatisch erstellt wurde:

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)
         }
      })
   }
}
```

Nächste Schritte

Nachdem Sie nun eine Chatraum-Verbindung hergestellt haben, fahren Sie mit Teil 2 dieses Tutorials für Kotlin-Coroutines fort, Nachrichten und Ereignisse.

Client-Messaging-SDK für IVS Chat: Tutorial für Kotlin-Coroutines, Teil 2: Nachrichten und Ereignisse

Der vorliegende zweite (und letzte) Teil des Tutorials ist in mehrere Abschnitte unterteilt:

1. the section called “Erstellen Sie eine Benutzeroberfläche für das Senden von Nachrichten.”
   a. the section called “Benutzeroberfläche Hauptlayout”
   b. the section called “Abstrahierte Benutzeroberflächen-Textzelle zur konsistenten Anzeige von Text”
   c. the section called “Benutzeroberfläche linke Chat-Nachricht”
   d. the section called “Benutzeroberfläche rechte Nachricht”
   e. the section called “Benutzeroberfläche zusätzliche Farbwerte”
2. the section called “View Binding anwenden”
3. the section called “Chat-Nachrichtenanfragen verwalten”
4. the section called “Letzte Schritte”

Umfassende Informationen zum SDK finden Sie im Client-Messaging-SDK für Amazon IVS Chat (im vorliegenden Benutzerhandbuch zu Amazon IVS Chat) und unter Chat Client Messaging: SDK für Android-Referenz auf GitHub.

Voraussetzung

Absolvieren Sie unbedingt Teil 1 dieses Tutorials: Chatrooms.

Erstellen Sie eine Benutzeroberfläche für das Senden von Nachrichten.

Nachdem wir die Chatraum-Verbindung erfolgreich initialisiert haben, ist es an der Zeit, unsere erste Nachricht zu senden. Für dieses Feature wird eine Benutzeroberfläche benötigt. Wir werden folgendes hinzufügen:

- `connect`/`disconnect`-Schaltfläche
- Nachrichteneingabe mit `send`-Schaltfläche
- Dynamische Nachrichtenliste. Um dies zu erstellen, verwenden wir das Android Jetpack RecyclerView.

Benutzeroberfläche Hauptlayout

Siehe Android Jetpack Layouts in der Android-Entwicklerdokumentation.

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>
```

Abstrahierte Benutzeroberflächen-Textzelle zur konsistenten Anzeige von Text

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>
```

Benutzeroberfläche linke Chat-Nachricht

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>
```

Benutzeroberfläche rechte Nachricht

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>
```

Benutzeroberfläche zusätzliche Farbwerte

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>
```

View Binding anwenden

Wir nutzen das Android View-Binding-Feature, um Bindungsklassen für unser XML-Layout referenzieren zu können. Um das Feature zu aktivieren, setzen Sie die `viewBinding`-Build-Option auf `true` in `./app/build.gradle`:

Kotlin-Skript:

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

android {
//    ...

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

Jetzt ist es an der Zeit, die Benutzeroberfläche mit unserem Kotlin-Code zu verbinden:

Kotlin:

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

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

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

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

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

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

        setUpChatView()

        updateConnectionState(ChatRoom.State.DISCONNECTED)
    }

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

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

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

}
```

Wir fügen auch Methoden hinzu, um Nachrichten zu löschen und Benutzer vom Chat zu trennen. Diese können über das Kontextmenü von Chat-Nachrichten aufgerufen werden:

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")
           }
        }
```

```
    }
}
```

Chat-Nachrichtenanfragen verwalten

Wir benötigen eine Möglichkeit, unsere Chat-Nachrichtenanfragen in all ihren möglichen Zuständen zu verwalten:

- Ausstehend – Eine Nachricht wurde an einen Chatraum gesendet, wurde aber noch nicht bestätigt oder abgelehnt.
- Bestätigt – Der Chatraum hat eine Nachricht an alle Benutzer (einschließlich uns) gesendet.
- Abgelehnt – Eine Nachricht wurde vom Chatraum mit einem Fehlerobjekt abgelehnt.

Wir werden noch nicht aufgelöste Chat-Anfragen und Chat-Nachrichten in der Liste speichern. Die Liste benötigt eine separate Klasse, die wir nennen `ChatEntries.kt`:

Kotlin:

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

package com.chatterbox.myapp

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

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

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

    val size get() = entries.size

    /**
     * Insert pending request at the end.
```

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

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

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

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

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

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

```
        }
        if (removeIndex != -1) {
            entries.removeAt(removeIndex)
            entries.add(removeIndex, ChatEntry.FailedRequest(request))
            adapter?.notifyItemChanged(removeIndex)
        } else {
            val insertIndex = entries.size
            entries.add(insertIndex, ChatEntry.FailedRequest(request))
            adapter?.notifyItemInserted(insertIndex)
        }
    }

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

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

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

Um unsere Liste mit der Benutzeroberfläche zu verbinden, verwenden wir einen Adapter. Weitere Informationen finden Sie unter Binden an Daten mit AdapterView und generierte Bindungsklassen.

Kotlin:

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

package com.chatterbox.myapp

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

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


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

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

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

    override fun getItemViewType(position: Int): Int {
        // Int 0 indicates to my message while Int 1 to other message
        val chatMessage = entries.entries[position]
```

```
        return if (chatMessage is ChatEntry.Message &&
chatMessage.message.sender.userId != userId) 1 else 0
    }

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

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

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

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

                viewHolder.failedMark.isGone = true

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

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

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

            is ChatEntry.PendingRequest -> {

viewHolder.container.setBackgroundColor(ContextCompat.getColor(context!!,
R.color.light_gray))
                viewHolder.textView.text = entry.request.content
                viewHolder.failedMark.isGone = true
```

```
                viewHolder.itemView.setOnCreateContextMenuListener(null)
                viewHolder.dateText?.text = "Sending"
            }

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

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

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

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

Letzte Schritte

Es ist Zeit, unseren neuen Adapter zu verbinden, der die `ChatEntries`-Klasse an `MainActivity` bindet:

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
        }
    }
}
```

Da wir bereits eine Klasse haben, die dafür verantwortlich ist, unsere Chat-Anfragen zu verfolgen (`ChatEntries`), sind wir bereit, Code für die Manipulation von `entries` zu roomListener zu implementieren. Wir werden `entries` und `connectionState` entsprechend des Ereignisses, auf das wir reagieren, aktualisieren:

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)
    }

// ...

}
```

Jetzt sollten Sie Ihre Anwendung ausführen können! (Siehe Erstellen und ausführen Ihrer App.) Denken Sie daran, dass Ihr Backend-Server in Betrieb sein muss, wenn Sie die App verwenden. Sie können ihn mit diesem Befehl vom Terminal im Stammverzeichnis unseres Projekts aus starten: `./gradlew :auth-server:run` oder indem Sie die `auth-server:run`-Gradle-Aufgabe direkt von Android Studio aus ausführen.

Client-Messaging-SDK für IVS Chat: Handbuch für iOS

Die Amazon Interactive Video (IVS) Chat Client Messaging iOS SDK bietet Schnittstellen, mit denen Sie unsere IVS Chat Messaging API auf Plattformen integrieren können, die die Programmiersprache Swift von Apple verwenden.

Aktuelle Version des IVS Chat Client Messaging iOS SDK: 1.0.1 (Versionshinweise)

Referenzdokumentation und Tutorials: Informationen zu den wichtigsten Methoden, die im Amazon IVS Chat Client Messaging iOS SDK verfügbar sind, finden Sie in der Referenzdokumentation unter: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.1/. Dieses Repository enthält auch verschiedene Artikel und Tutorials.

Beispiel-Code: Siehe das iOS-Beispiel-Repository auf GitHub: https://github.com/aws-samples/amazon-ivs-chat-for-ios-demo.

Plattformvoraussetzungen iOS 13.0 oder höher ist für die Entwicklung erforderlich.

Erste Schritte mit dem IVS Chat Client Messaging iOS SDK

Wir empfehlen Ihnen, das SDK über Swift Package Manager zu integrieren. Alternativ können Sie das Framework manuell integrieren.

Nach der Integration des SDK können Sie das SDK importieren, indem Sie den folgenden Code oben in Ihrer relevanten Swift-Datei hinzufügen:

```
import AmazonIVSChatMessaging
```

Swift Package Manager

Um die `AmazonIVSChatMessaging`-Bibliothek in einem Swift-Package-Manager-Projekt zu verwenden, fügen Sie sie den Abhängigkeiten für Ihr Paket und den Abhängigkeiten für Ihre relevanten Ziele hinzu:

1. Laden Sie die neueste `.xcframework` von https://ivschat.live-video.net/1.0.1/AmazonIVSChatMessaging.xcframework.zip herunter.
2. Führen Sie in Ihrem Terminal aus:

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

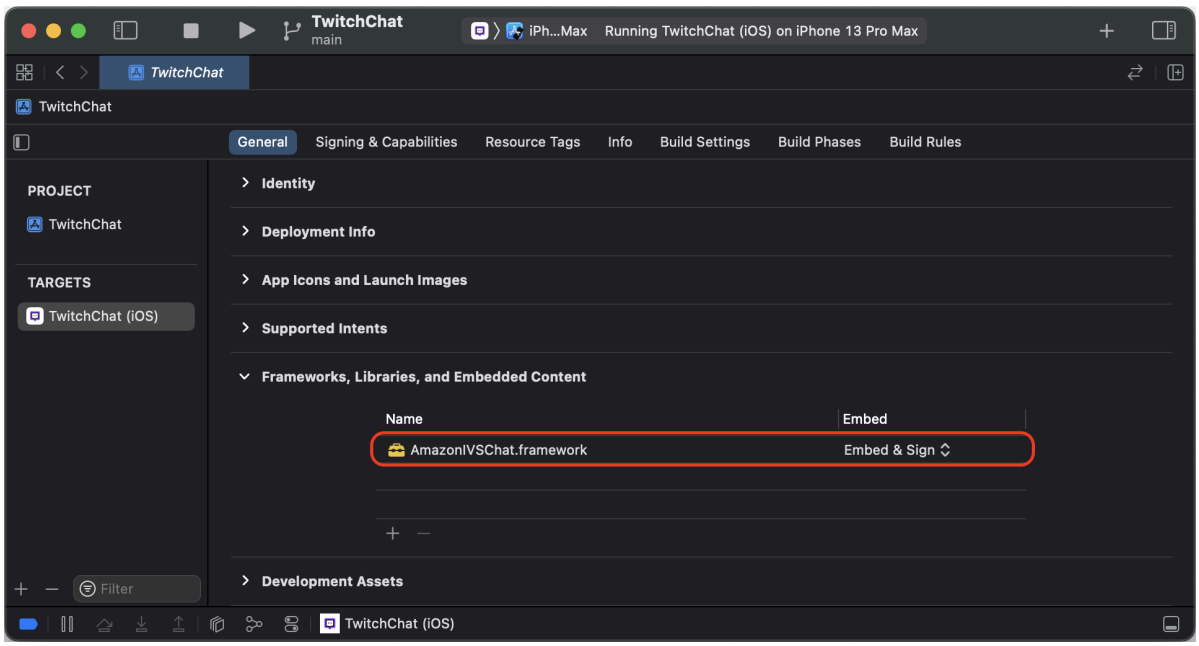
3. Nehmen Sie die Ausgabe des vorherigen Schritts und fügen Sie sie in die Prüfsummeneigenschaft von `.binaryTarget` ein, wie unten in der `Package.swift`-Datei Ihres Projekts gezeigt:

   ```
   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/1.0.1/
   AmazonIVSChatMessaging.xcframework.zip",
                  checksum: "<SHA-extracted-using-steps-detailed-above>"
               ),
               // your other dependencies
            ],
         ),
         // other targets
      ]
   ```

```
)
```

Manuelle Installation

1. Laden Sie die neueste Version von https://ivschat.live-video.net/1.0.1/AmazonIVSChatMessaging.xcframework.zip herunter.
2. Extrahieren Sie den Inhalt des Archivs. `AmazonIVSChatMessaging.xcframework` enthält das SDK für Gerät und Simulator.
3. Betten Sie das extrahierte `AmazonIVSChatMessaging.xcframework` ein, indem Sie es in den Abschnitt Rahmenbedingungen, Bibliotheken und eingebettete Inhalte auf der Registerkarte Allgemein für Ihr Anwendungsziel ziehen:



Verwenden des IVS Chat Client Messaging iOS SDK

Dieses Dokument führt Sie durch die Schritte zur Integration des Amazon IVS Chat Client Messaging iOS SDK.

Verbinden mit einem Chatroom

Bevor Sie beginnen, sollten Sie mit Erste Schritte mit Amazon IVS Chat vertraut sein. Sehen Sie sich auch die Beispiel-Apps für Web, Android und iOS an.

Um eine Verbindung zu einem Chatroom herzustellen, benötigt Ihre App eine Möglichkeit, ein von Ihrem Backend bereitgestelltes Chat-Token abzurufen. Ihre Anwendung wird wahrscheinlich ein Chat-Token über eine Netzwerkanfrage an Ihr Backend abrufen.

Um dieses abgerufene Chat-Token mit dem SDK zu kommunizieren, müssen Sie für das `ChatRoom`-Modell des SDK entweder eine `async`-Funktion oder eine Instance eines Objekts bereitstellen, das zum Zeitpunkt der Initialisierung dem bereitgestellten `ChatTokenProvider`-Protokoll entspricht. Der von einer dieser Methoden zurückgegebene Wert muss eine Instance des `ChatToken`-Modells des SDK sein.

Hinweis: Sie füllen Instances des `ChatToken`-Modells mit Daten, die aus Ihrem Backend abgerufen wurden. Die Felder, die für die Initialisierung einer `ChatToken`-Instance benötigt werden, sind die gleichen wie die Felder in der CreateChatToken-Antwort. Weitere Informationen zum Initialisieren von Instances des `ChatToken`-Modells finden Sie unter Erstellen einer Instance von ChatToken. Denken Sie daran, Ihr Backend ist verantwortlich für die Bereitstellung der Daten in der `CreateChatToken`-Antwort für Ihre App. Wie Sie sich entscheiden, mit Ihrem Backend zu kommunizieren, um Chat-Token zu generieren, hängt von Ihrer App und ihrer Infrastruktur ab.

Nachdem Sie Ihre Strategie ausgewählt haben, um ein `ChatToken` für das SDK bereitzustellen, rufen Sie `.connect()` auf, nachdem Sie eine `ChatRoom`-Instance mit Ihrem Token-Anbieter und der AWS-Region, die Ihr Backend zum Erstellen des Chatrooms verwendet hat, mit dem Sie eine Verbindung herstellen möchten, erfolgreich initialisiert haben. Beachten Sie, dass `.connect()` eine auslösende asynchrone Funktion ist:

```
import AmazonIVSChatMessaging

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

Konform mit dem ChatTokenProvider-Protokoll

Für den `tokenProvider`-Parameter im Initialisierer für `ChatRoom` können Sie eine Instance von `ChatTokenProvider` angeben. Hier ist ein Beispiel für ein Objekt, das `ChatTokenProvider` entspricht:

```
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
      )
   }
}
```

Sie können dann eine Instance dieses konformen Objekts nehmen und sie an den Initialisierer für `ChatRoom` übergeben:

```
// 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()
```

Bereitstellen einer asynchronen Funktion in Swift

Angenommen, Sie haben bereits einen Manager, mit dem Sie die Netzwerkanforderungen Ihrer Anwendung verwalten. Diese könnte wie folgt aussehen:

```
import AmazonIVSChatMessaging

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

Sie könnten einfach eine weitere Funktion in Ihrem Manager hinzufügen, um ein `ChatToken` aus Ihrem Backend abzurufen:

```
import AmazonIVSChatMessaging

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

Verwenden Sie dann den Verweis auf diese Funktion in Swift, wenn Sie einen `ChatRoom` initialisieren:

```
import AmazonIVSChatMessaging

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

Erstellen einer Instance von ChatToken

Sie können ganz einfach eine Instance von `ChatToken` mit dem im SDK bereitgestellten Initialisierer erstellen. Weitere Informationen zu den Eigenschaften von `ChatToken` finden Sie in der Dokumentation in `Token.swift`.

```
import AmazonIVSChatMessaging

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

Verwenden von Decodable

Wenn Ihr Backend während der Anbindung an die IVS-Chat-API beschließt, die CreateChatToken-Antwort einfach an Ihre Frontend-Anwendung weiterzuleiten, können Sie die Konformität von `ChatToken` mit dem `Decodable`-Protokoll von Swift nutzen. Es gibt jedoch einen Haken.

Die `CreateChatToken`-Antwortnutzlast verwendet Zeichenfolgen für Datumsangaben, die mit dem ISO 8601-Standard für Internet-Zeitstempel formatiert sind. Normalerweise

würden Sie in Swift `JSONDecoder.DateDecodingStrategy.iso8601` als Wert für die `.dateDecodingStrategy`-Eigenschaft von `JSONDecoder` angeben. Allerdings verwendet `CreateChatToken` hochpräzise Sekundenbruchteile in seinen Strings, und dies wird von `JSONDecoder.DateDecodingStrategy.iso8601` nicht unterstützt.

Für Ihre Bequemlichkeit bietet das SDK eine öffentliche Erweiterung für `JSONDecoder.DateDecodingStrategy` mit einer zusätzlichen `.preciseISO8601`-Strategie, die es Ihnen ermöglicht, `JSONDecoder` beim Dekodieren einer Instance von `ChatToken` erfolgreich zu nutzen:

```
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)
```

Trennen der Verbindung zu einem Chatroom

Um die Verbindung zu einer `ChatRoom`-Instance, mit der Sie erfolgreich verbunden sind, manuell zu trennen, rufen Sie `room.disconnect()` auf. Standardmäßig rufen Chatrooms diese Funktion automatisch auf, wenn sie freigegeben werden.

```
import AmazonIVSChatMessaging

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

// Disconnect
room.disconnect()
```

Empfangen einer Chat-Nachricht/eines Chat-Ereignisses

Um Nachrichten in Ihrem Chatroom zu senden und zu empfangen, müssen Sie ein Objekt bereitstellen, das dem `ChatRoomDelegate`-Protokoll entspricht, nachdem Sie eine Instance von `ChatRoom` erfolgreich initialisiert und `room.connect()` aufgerufen haben. Hier ist ein typisches Beispiel mit `UIViewController`:

```
import AmazonIVSChatMessaging
import Foundation
import UIKit

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

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

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

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

Erhalten von Benachrichtigungen, wenn sich die Verbindung ändert

Wie zu erwarten ist, können Sie Aktionen wie das Senden einer Nachricht in einem Raum erst ausführen, wenn der Raum vollständig verbunden ist. Die Architektur des SDK ist darauf ausgelegt, die Verbindung zu einem ChatRoom auf einem Hintergrund-Thread über asynchrone APIs zu fördern. Wenn Sie in Ihrer Benutzeroberfläche etwas erstellen möchten, das so etwas wie eine Schaltfläche zum Senden einer Nachricht deaktiviert, bietet das SDK zwei Strategien, um benachrichtigt zu werden, wenn sich der Verbindungsstatus eines Chatrooms ändert, indem Sie `Combine` oder `ChatRoomDelegate` verwenden. Diese werden im Folgenden beschrieben.

Wichtig: Der Verbindungsstatus eines Chatrooms kann sich auch aufgrund einer unterbrochenen Netzwerkverbindung ändern. Berücksichtigen Sie dies beim Erstellen Ihrer App.

Verwenden von Combine

Jede Instance von `ChatRoom` verfügt über einen eigenen `Combine`-Publisher in Form der `state`-Eigenschaft:

```
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()
}
```

Verwenden von ChatRoomDelegate

Verwenden Sie alternativ die optionalen Funktionen `roomDidConnect(_:)`, `roomIsConnecting(_:)` und `roomDidDisconnect(_:)` innerhalb eines Objekts, das `ChatRoomDelegate` entspricht. Hier finden Sie ein Beispiel mit einem `UIViewController`:

```
import AmazonIVSChatMessaging
import Foundation
import UIKit
```

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

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

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

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

Durchführen von Aktionen in einem Chatroom

Verschiedene Benutzer haben unterschiedliche Funktionen für Aktionen, die sie in einem Chatroom ausführen können, z. B. das Senden einer Nachricht, das Löschen einer Nachricht oder das Trennen der Verbindung eines Benutzers. Um eine dieser Aktionen auszuführen, rufen Sie `perform(request:)` auf einem verbundenen `ChatRoom` auf und übergeben Sie eine Instance eines der bereitgestellten `ChatRequest`-Objekte im SDK. Die unterstützten Anfragen sind in `Request.swift`.

Für einige Aktionen, die in einem Chatroom ausgeführt werden, müssen verbundene Benutzer über bestimmte Funktionen verfügen, wenn Ihre Backend-Anwendung `CreateChatToken` aufruft. Standardmäßig kann das SDK die Fähigkeiten eines verbundenen Benutzers nicht erkennen.

Während Sie also versuchen können, Moderator-Aktionen in einer verbundenen Instance von `ChatRoom` auszuführen, entscheidet die Steuerebenen-API letztendlich, ob diese Aktion erfolgreich sein wird.

Alle Aktionen, die `room.perform(request:)` durchlaufen, warten, bis der Raum die erwartete Instance eines Modells (dessen Typ dem Anforderungsobjekt selbst zugeordnet ist) erhält, das mit der `requestId` des empfangenen Modells und des Anforderungsobjekts übereinstimmt. Wenn es ein Problem mit der Anfrage gibt, löst `ChatRoom` immer einen Fehler in Form eines `ChatError` aus. Die Definition von `ChatError` ist in `Error.swift`.

Senden einer Nachricht

Um eine Chat-Nachricht zu senden, verwenden Sie eine Instance von `SendMessageRequest`:

```
import AmazonIVSChatMessaging

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

Wie oben erwähnt, gibt `room.perform(request:)` zurück, sobald ein entsprechendes `ChatMessage` vom `ChatRoom` empfangen wird. Wenn es ein Problem mit der Anfrage gibt (z. B. das Überschreiten der Nachrichtenzeichenbeschränkung für einen Raum), wird eine Instance von `ChatError` stattdessen ausgelöst. Sie können dann diese nützlichen Informationen in Ihrer Benutzeroberfläche anzeigen:

```
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)
}
```

Anhängen von Metadaten an eine Nachricht

Wenn Sie eine Nachricht senden, können Sie Metadaten anhängen, die ihr zugeordnet werden. `SendMessageRequest` hat eine `attributes`-Eigenschaft, mit der Sie Ihre Anfrage initialisieren können. Die Daten, die Sie dort anhängen, werden an die Nachricht angehängt, wenn andere Personen diese Nachricht im Raum erhalten.

Hier finden Sie ein Beispiel für das Anhängen von Emote-Daten an eine gesendete Nachricht:

```
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"
      ]
   )
)
```

Die Verwendung von `attributes` in einem `SendMessageRequest` kann äußerst nützlich sein, um komplexe Funktionen in Ihrem Chat-Produkt zu erstellen. Zum Beispiel könnte man Threading-Funktionalität mit dem `[String : String]`-Attribute-Wörterbuch in einem `SendMessageRequest` entwickeln!

Die `attributes`-Nutzlast ist sehr flexibel und leistungsstark. Verwenden Sie sie, um Informationen über Ihre Nachricht abzuleiten, die Sie ansonsten nicht ableiten könnten. Die Verwendung von

Attributen ist viel einfacher als beispielsweise das Analysieren der Zeichenfolge einer Nachricht, um Informationen über Dinge wie Emotes zu erhalten.

Löschen einer Nachricht

Das Löschen einer Chat-Nachricht ist wie das Senden einer Chat-Nachricht. Verwenden Sie die `room.perform(request:)`-Funktion auf `ChatRoom`, um dies zu erreichen, indem Sie eine Instance von `DeleteMessageRequest` erstellen.

Um einfach auf frühere Instances empfangener Chat-Nachrichten zuzugreifen, übergeben Sie den Wert von `message.id` an den Initialisierer von `DeleteMessageRequest`.

Geben Sie optional eine Ursachenzeichenfolge für `DeleteMessageRequest` an, damit Sie diese in Ihrer Benutzeroberfläche anzeigen können.

```
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!"
   )
)
```

Da es sich um eine Moderator-Aktion handelt, ist Ihr Benutzer möglicherweise nicht in der Lage, die Nachricht eines anderen Benutzers zu löschen. Sie können die auslösbare Funktionsmechanik von Swift verwenden, um eine Fehlermeldung in Ihrer Benutzeroberfläche anzuzeigen, wenn ein Benutzer versucht, eine Nachricht ohne die entsprechende Funktion zu löschen.

Wenn Ihr Backend `CreateChatToken` für einen Benutzer aufruft, muss es `"DELETE_MESSAGE"` an das `capabilities`-Feld übergeben, um diese Funktionalität für einen verbundenen Chat-Benutzer zu aktivieren.

Im Folgenden finden Sie ein Beispiel für das Abfangen eines Funktionsfehlers, der beim Versuch ausgelöst wird, eine Nachricht ohne die entsprechenden Berechtigungen zu löschen:

```
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)
}
```

Trennen der Verbindung eines anderen Benutzers

Verwenden Sie `room.perform(request:)`, um einen anderen Benutzer von einem Chatroom zu trennen. Verwenden Sie insbesondere eine Instance von `DisconnectUserRequest`. Alle von einem `ChatRoom` empfangenen `ChatMessages` haben eine `sender`-Eigenschaft, die die Benutzer-ID enthält, die Sie mit einer Instance von `DisconnectUserRequest` ordnungsgemäß initialisieren müssen. Geben Sie optional eine Ursachenzeichenfolge für die Trennungsanfrage an.

```
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
   )
)
```

Da dies ein weiteres Beispiel für eine Moderator-Aktion ist, können Sie versuchen, die Verbindung eines anderen Benutzers zu trennen, aber das ist nur möglich, wenn Sie über die `DISCONNECT_USER`-Funktion verfügen. Die Funktion wird festgelegt, wenn Ihre Backend-Anwendung `CreateChatToken` aufruft und die `"DISCONNECT_USER"`-Zeichenfolge in das `capabilities`-Feld einfügt.

Wenn Ihr Benutzer nicht in der Lage ist, die Verbindung eines anderen Benutzers zu trennen, löst `room.perform(request:)` eine Instance von `ChatError` aus, genau wie die anderen Anfragen. Sie können die `errorCode`-Eigenschaft des Fehlers überprüfen, um festzustellen, ob die Anfrage aufgrund fehlender Moderatorenberechtigungen fehlgeschlagen ist:

```
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)
}
```

Client-Messaging-SDK für IVS Chat: Tutorial für iOS

Das Amazon Interactive Video (IVS) Chat Client Messaging iOS SDK bietet Schnittstellen, mit denen Sie unsere IVS-Chat-Messaging-API auf Plattformen integrieren können, die Apples Programmiersprache Swift verwenden.

Ein Tutorial für das Chat iOS SDK finden Sie unter https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/latest/tutorials/table-of-contents/.

Client-Messaging-SDK für IVS Chat: Handbuch für JavaScript

Die Amazon Interactive Video (IVS) Chat Client Messaging JavaScript SDK ermöglicht Ihnen eine Integration unserer Amazon IVS Chat Messaging API auf Plattformen mit einem Webbrowser.

Aktuelle Version des IVS Chat Client Messaging JavaScript SDK: 1.0.2 (Versionshinweise)

Referenzdokumentation: Informationen zu den wichtigsten Methoden, die im Amazon IVS Chat Client Messaging JavaScript SDK verfügbar sind, finden Sie in der Referenzdokumentation unter: https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/

Beispielcode: Im Beispiel-Repository auf GitHub finden Sie eine webspezifische Demo mit dem JavaScript SDK: https://github.com/aws-samples/amazon-ivs-chat-web-demo

Erste Schritte mit dem IVS Chat Client Messaging JavaScript SDK

Bevor Sie beginnen, sollten Sie mit Erste Schritte mit Amazon IVS Chat vertraut sein.

Hinzufügen des Package

Verwenden Sie entweder:

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

oder:

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

React Native Support

Das IVS Chat Client Messaging JavaScript SDK hat eine `uuid`-Abhängigkeit, die die Methode `crypto.getRandomValues` verwendet. Da diese Methode in React Native nicht unterstützt wird, müssen Sie das zusätzliche Polyfill `react-native-get-random-value` installieren und es oben in der Datei `index.js` importieren:

```
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);
```

Einrichten Ihres Backends

Für diese Integration sind Endpunkte auf Ihrem Server erforderlich, die mit der Amazon-IVS-Chat-API kommunizieren. Verwenden Sie die offiziellen AWS-Bibliotheken für den Zugriff auf die Amazon-IVS-API von Ihrem Server aus. Diese Bibliotheken sind in mehreren Sprachen aus den öffentlichen Paketen zugänglich, z. B. node.js, java und go.

Erstellen Sie einen Server-Endpunkt, der mit dem Vorgang CreateChatToken der API von Amazon IVS Chat kommuniziert, um ein Chat-Token für Chat-Benutzer zu erstellen.

Verwenden des IVS Chat Client Messaging JavaScript SDK

Dieses Dokument führt Sie durch die Schritte zur Integration des Amazon IVS Chat Client Messaging JavaScript SDK.

Initialisieren einer Chatroom-Instance

Erstellen Sie eine Instance der `ChatRoom`-Klasse. Dafür muss `regionOrUrl` (die AWS-Region, in der Ihr Chatroom gehostet wird) und `tokenProvider` (die Methode zum Abrufen von Token wird im nächsten Schritt erstellt) weitergegeben werden:

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

Token-Provider-Funktion

Erstellen Sie eine asynchrone Token-Provider-Funktion, die ein Chat-Token aus Ihrem Backend abruft:

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

Die Funktion sollte keine Parameter akzeptieren und ein Promise zurückgeben, das ein Chat-Token-Objekt enthält:

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

Diese Funktion wird benötigt, um das ChatRoom-Objekt zu initialisieren. Füllen Sie unten die Felder `<token>` und `<date-time>` mit Werten aus, die Sie von Ihrem Backend erhalten haben:

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

Denken Sie daran, `tokenProvider` an den ChatRoom-Konstruktor zu übergeben. ChatRoom aktualisiert das Token, wenn die Verbindung unterbrochen wird oder die Sitzung abläuft. Verwenden Sie nicht das `tokenProvider`, um ein Token irgendwo zu speichern. Der ChatRoom erledigt es für Sie.

Empfangen von Ereignissen

Abonnieren Sie als Nächstes Chatroom-Ereignisse, um Lebenszyklusereignisse sowie Nachrichten und Ereignisse zu erhalten, die im Chatroom übermittelt werden:

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

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

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

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

/** Called when a chat event has been received. */
const unsubscribeOnEventReceived = room.addListener('event', (event) => {
 /* Example event:
  * {
  *   id: "5OPsDdX18qcJ",
  *   eventName: "customEvent,
  *   sendTime: new Date("2022-10-11T12:46:41.723Z"),
  *   requestId: "d1b511d8-d5ed-4346-b43f-49197c6e61de",
  *   attributes: { "Custom Attribute": "Custom Attribute Value" }
  * }
  */
});

/** Called when `aws:DELETE_MESSAGE` system event has been received. */
const unsubscribeOnMessageDelete = room.addListener('messageDelete',
 (deleteMessageEvent) => {
 /* Example delete message event:
  * {
```

```
  *   id: "AYk6xKitV4On",
  *   messageId: "R1BLTDN84zEO",
  *   reason: "Spam",
  *   sendTime: new Date("2022-10-11T12:56:41.113Z"),
  *   requestId: "b379050a-2324-497b-9604-575cb5a9c5cd",
  *   attributes: { MessageID: "R1BLTDN84zEO", Reason: "Spam" }
  * }
  */
});

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

Verbinden mit dem Chatroom

Der letzte Schritt der grundlegenden Initialisierung besteht darin, eine Verbindung zu dem Chatroom herzustellen, indem eine WebSocket-Verbindung hergestellt wird. Rufen Sie dazu einfach die `connect()`-Methode innerhalb der Raum-Instance ab:

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

Das SDK versucht, eine Verbindung zu dem Chatroom herzustellen, der in dem von Ihrem Server empfangenen Chat-Token codiert ist.

Nachdem Sie `connect()` aufgerufen haben, wechselt der Raum in den Status `connecting` und gibt ein `connecting`-Ereignis aus. Wenn der Raum erfolgreich verbunden ist, geht er in den Status `connected` über und sendet ein `connect`-Ereignis aus.

Ein Verbindungsfehler kann aufgrund von Problemen beim Abrufen des Tokens oder beim Herstellen einer Verbindung zu WebSocket auftreten. In diesem Fall versucht der Raum, die Verbindung

automatisch wieder herzustellen, bis zu der im Konstruktorparameter `maxReconnectAttempts` angegebenen Anzahl von Malen. Während der Wiederverbindungsversuche befindet sich der Raum im Status `connecting` und gibt keine weiteren Ereignisse aus. Nach Erschöpfung der Verbindungsversuche geht der Raum in den Zustand `disconnected` über und gibt ein `disconnect`-Ereignis aus (mit einem relevanten Trennungsgrund). Im `disconnected`-Zustand versucht der Raum nicht mehr, eine Verbindung herzustellen. Sie müssen `connect()` erneut abrufen, um den Verbindungsvorgang auszulösen.

Durchführen von Aktionen in einem Chatroom

Das Amazon IVS Chat Messaging SDK bietet Benutzeraktionen zum Senden von Nachrichten, Löschen von Nachrichten und Verbindungstrennungen zu anderen Benutzern. Diese sind auf der `ChatRoom`-Instance verfügbar. Sie geben ein `Promise`-Objekt zurück, mit dem Sie eine Bestätigung oder Ablehnung der Anfrage erhalten können.

Senden einer Nachricht

Für diese Anfrage muss eine `SEND_MESSAGE`-Kapazität in Ihrem Chat-Token codiert sein.

So lösen Sie eine Anfrage zum Senden einer Nachricht aus:

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

Um eine Bestätigung oder Ablehnung der Anfrage zu erhalten, `await` Sie das zurückgegebene Promise oder verwenden Sie die `then()`-Methode:

```
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.
}
```

Löschen einer Nachricht

Für diese Anfrage muss eine `DELETE_MESSAGE`-Kapazität in Ihrem Chat-Token codiert sein.

Um eine Nachricht zu Moderationszwecken zu löschen, rufen Sie die folgende `deleteMessage()`-Methode auf:

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

Um eine Bestätigung oder Ablehnung der Anfrage zu erhalten, `await` Sie das zurückgegebene Promise oder verwenden Sie die `then()`-Methode:

```
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.
}
```

Trennen der Verbindung eines anderen Benutzers

Für diese Anfrage muss eine `DISCONNECT_USER`-Kapazität in Ihrem Chat-Token codiert sein.

Um einen anderen Benutzer zu Moderationszwecken zu löschen, rufen Sie die `disconnectUser()`-Methode auf:

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

Um eine Bestätigung oder Ablehnung der Anfrage zu erhalten, `await` Sie das zurückgegebene Promise oder verwenden Sie die `then()`-Methode:

```
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.
}
```

Trennen der Verbindung zu einem Chatroom

Um Ihre Verbindung zum Chatroom zu trennen, rufen Sie die `disconnect()`-Methode für die `room`-Instance auf:

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

Der Aufruf dieser Methode bewirkt, dass der Raum den zugrunde liegenden WebSocket ordnungsgemäß schließt. Die Raum-Instance geht in einen `disconnected`-Zustand über und gibt ein Trennereignis aus, wobei der `disconnect`-Grund auf `"clientDisconnect"` gesetzt ist.

Client-Messaging-SDK für IVS Chat: Tutorial für JavaScript, Teil 1: Chaträume

Hierbei handelt es sich um den ersten Teil eines zweiteiligen Tutorials. Darin lernen Sie die Grundlagen der Arbeit mit dem Amazon IVS Chat Client Messaging JavaScript SDK kennen, indem Sie eine voll funktionsfähige App mit JavaScript/TypeScript entwickeln. Wir nennen die App Chatterbox.

Die Zielgruppe sind erfahrene Entwickler, die das Amazon IVS Chat Messaging SDK noch nicht kennen. Sie sollten mit der Programmiersprache JavaScript/TypeScript und der Bibliothek React vertraut sein.

Der Kürze halber bezeichnen wir das Amazon IVS Chat Client Messaging JavaScript SDK als Chat JS SDK.

Hinweis: In einigen Fällen sind die Codebeispiele für JavaScript und TypeScript identisch, daher werden sie kombiniert.

Der vorliegende erste Teil des Tutorials ist in mehrere Abschnitte unterteilt:

1. the section called “Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers”
2. the section called “Erstellen eines Chatterbox-Projekts”
3. the section called “Verbinden mit einem Chatroom”
4. the section called “Erstellen eines Token-Anbieters”
5. the section called “Beobachten von Verbindungsaktualisierungen”
6. the section called “Erstellen einer Schaltflächenkomponente zum Senden”
7. the section called “Erstellen einer Nachrichteneingabe”
8. the section called “Nächste Schritte”

Umfassende Informationen zum SDK finden Sie im Amazon IVS Chat Client Messaging SDK (im vorliegenden Benutzerhandbuch zu Amazon IVS Chat) und unter Chat Client Messaging: SDK for JavaScript Reference (Chat Client Messaging: SDK für JavaScript – Referenz) auf GitHub.

Voraussetzungen

- Sie sollten mit JavaScript/TypeScript und der Bibliothek React vertraut sein. Wenn Sie mit React nicht vertraut sind, können Sie unter Einführung in Tic-Tac-Toe die Grundlagen kennenlernen.
- Lesen Sie sich Erste Schritte mit IVS Chat durch.
- Erstellen Sie einen AWS-IAM-Benutzer mit den Fähigkeiten CreateChatToken und CreateRoom, die in einer vorhandenen IAM-Richtlinie definiert sind. (Siehe). Erste Schritte mit IVS Chat.)
- Stellen Sie sicher, dass die Geheim-/Zugriffsschlüssel für diesen Benutzer in einer Datei mit den AWS-Anmeldeinformationen gespeichert sind. Entsprechende Anweisungen finden Sie im Benutzerhandbuch zur AWS-CLI (insbesondere unter Einstellungen für Konfigurations- und Anmeldeinformationsdateien).
- Erstellen Sie einen Chatroom und speichern Sie dessen ARN. Siehe Erste Schritte mit IVS Chat. (Wenn Sie den ARN nicht speichern, können Sie ihn später über die Konsole oder die Chat-API nachschlagen.)
- Installieren Sie die Umgebung Node.js 14+ mit dem Paketmanager NPM oder Yarn.

Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers

Die Erstellung von Chatrooms und die Generierung der Chat-Token, die benötigt werden, damit das Chat JS SDK die Clients für Ihre Chatrooms authentifizieren und autorisieren kann, erfolgt in Ihrer Backend-Anwendung. Sie müssen Ihr eigenes Backend verwenden, da die AWS-Schlüssel nicht sicher in einer mobilen App gespeichert werden können. Versierte Angreifer könnten diese extrahieren und Zugriff auf Ihr AWS-Konto erlangen.

Weitere Informationen finden Sie unter Erstellen eines Chat-Tokens unter Erste Schritte mit Amazon IVS Chat. Wie im dortigen Flussdiagramm gezeigt, erfolgt die Erstellung eines Chat-Tokens in Ihrer serverseitigen Anwendung. Das bedeutet, dass Ihre App eigene Mittel zur Generierung eines Chat-Tokens bereitstellen muss, indem sie ein Token von der serverseitigen Anwendung anfordert.

In diesem Abschnitt werden die Grundlagen der Erstellung eines Token-Anbieters in Ihrem Backend vermittelt. Mit dem Express-Framework erstellen wir einen lokalen Live-Server, der die Erstellung von Chat-Token mithilfe Ihrer lokalen AWS-Umgebung verwaltet.

Erstellen Sie ein leeres `npm`-Projekt mit NPM. Erstellen Sie ein Verzeichnis für Ihre Anwendung und machen Sie dieses zu Ihrem Arbeitsverzeichnis:

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

Erstellen Sie mit `npm init` eine `package.json`-Datei für die Anwendung:

```
$ npm init
```

Durch diesen Befehl werden Sie aufgefordert, unter anderem den Namen und die Version der Anwendung einzugeben. Drücken Sie vorerst einfach RETURN, um die Standardwerte zu akzeptieren, mit der folgenden Ausnahme:

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

Drücken Sie RETURN, um den vorgeschlagenen Standarddateinamen `index.js` zu akzeptieren, oder geben Sie den gewünschten Namen für die Hauptdatei ein.

Installieren Sie nun die erforderlichen Abhängigkeiten:

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

`aws-sdk` erfordert Variablen der Konfigurationsumgebung, die automatisch aus einer Datei namens `.env` im Stammverzeichnis geladen werden. Um sie zu konfigurieren, erstellen Sie eine neue Datei mit dem Namen `.env` und tragen Sie die fehlenden Konfigurationsinformationen ein:

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

Jetzt erstellen wir im Stammverzeichnis eine Eintrittspunktdatei mit dem Namen, den Sie oben im Befehl `npm init` eingegeben haben. In diesem Fall verwenden wir `index.js` und importieren alle erforderlichen Pakete:

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

Erstellen Sie jetzt eine neue Instance von `express`:

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

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

Danach können Sie die erste Endpunkt-POST-Methode für den Token-Anbieter erstellen. Entnehmen Sie die erforderlichen Parameter dem Hauptteil der Anforderung (`roomId`, `userId`, `capabilities` und`sessionDurationInMinutes`):

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

Fügen Sie die Validierung der erforderlichen Felder hinzu:

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

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

Nach der Vorbereitung der POST-Methode integrieren wir `createChatToken` mit `aws-sdk` für die Kernfunktionalität der Authentifizierung/Autorisierung:

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

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

  ivsChat.createChatToken({ roomIdentifier, userId, capabilities,
 sessionDurationInMinutes }, (error, data) => {
    if (error) {
      console.log(error);
      res.status(500).send(error.code);
    } else if (data.token) {
      const { token, sessionExpirationTime, tokenExpirationTime } = data;
      console.log(`Retrieved Chat Token: ${JSON.stringify(data, null, 2)}`);

      res.json({ token, sessionExpirationTime, tokenExpirationTime });
    }
  });
});
```

Fügen Sie am Ende der Datei einen Port-Listener für die App `express` hinzu:

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

Nun können Sie den Server mit dem folgenden Befehl im Stammverzeichnis des Projekts ausführen:

```
$ node index.js
```

Tipp: Dieser Server akzeptiert URL-Anforderungen unter https://localhost:3000.

Erstellen eines Chatterbox-Projekts

Zunächst erstellen Sie das React-Projekt namens `chatterbox`. Führen Sie diesen Befehl aus:

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

Sie können das Chat Client Messaging JS SDK über den Node Package Manager (NPM) oder den Paketmanager Yarn integrieren:

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

Verbinden mit einem Chatroom

Hier erstellen Sie einen `ChatRoom` und stellen mit asynchronen Methoden eine Verbindung dazu her. Die Klasse `ChatRoom` verwaltet die Verbindung der Benutzer zum Chat JS SDK. Um eine Verbindung zu einem Chatroom herzustellen, müssen Sie eine Instance von `ChatToken` in Ihrer React-Anwendung angeben.

Navigieren Sie zur App-Datei, die im `chatterbox`-Standardprojekt erstellt wurde, und löschen Sie alles zwischen den beiden `<div>`-Tags. Von dem vorab eingetragenen Code wird nichts benötigt. An dieser Stelle ist unsere App ziemlich leer.

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

import * as React from 'react';

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

Erstellen Sie eine neue `ChatRoom`-Instance und übergeben Sie sie mit dem Hook `useState` an den Status. Das erfordert die Übergabe von `regionOrUrl` (die AWS-Region, in der der Chatroom gehostet wird) und `tokenProvider` (wird für den Authentifizierungs- und Autorisierungsablauf im Backend verwendet, der in nachfolgenden Schritten erstellt wird).

Wichtig: Sie müssen dieselbe AWS-Region verwenden, in der Sie den Chatroom unter Erste Schritte mit Amazon IVS Chat erstellt haben. Die API ist ein regionaler AWS-Service. Eine Liste der unterstützten Regionen und HTTPS-Service-Endpunkte für Amazon IVS Chat finden Sie auf der Seite Regionen für Amazon IVS Chat.

```
// 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>;
}
```

Erstellen eines Token-Anbieters

Als nächsten Schritt müssen wir eine parameterlose `tokenProvider`-Funktion erstellen, die vom `ChatRoom`-Konstruktor benötigt wird. Zunächst erstellen wir eine `fetchChatToken`-Funktion, die eine POST-Anforderung an die Backend-Anwendung übermittelt, die Sie unter the section called "Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers" eingerichtet haben. Chat-Token enthalten die notwendigen Informationen, damit das SDK erfolgreich eine Verbindung zum Chatroom herstellen kann. Die Chat-API nutzt diese Token als sichere Methode, um die Identität von Benutzern, die Fähigkeiten innerhalb eines Chatrooms und die Sitzungsdauer zu validieren.

Erstellen Sie im Projektnavigator eine neue TypeScript/JavaScript-Datei mit dem Namen `fetchChatToken`. Erstellen Sie eine Abrufanforderung an die `backend`-Anwendung und geben Sie das Objekt `ChatToken` aus der Antwort zurück. Fügen Sie die Eigenschaften des Anforderungshauptteils hinzu, die für die Erstellung eines Chat-Tokens erforderlich sind. Verwenden Sie die für Amazon-Ressourcennamen (ARNs) definierten Regeln. Diese Eigenschaften sind im Vorgang CreateChatToken dokumentiert.

Hinweis: Die hier verwendete URL ist dieselbe URL, die Ihr lokaler Server beim Ausführen der Backend-Anwendung erstellt hat.

TypeScript

```
// fetchChatToken.ts

import { ChatToken } from 'amazon-ivs-chat-messaging';

type UserCapability = 'DELETE_MESSAGE' | 'DISCONNECT_USER' | 'SEND_MESSAGE';

export async function fetchChatToken(
  userId: string,
  capabilities: UserCapability[] = [],
  attributes?: Record<string, string>,
```

```
  sessionDurationInMinutes?: number,
): Promise<ChatToken> {
  const response = await fetch(`${process.env.BACKEND_BASE_URL}/create_chat_token`,
 {
    method: 'POST',
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      userId,
      roomIdentifier: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
  });

  const token = await response.json();

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

JavaScript

```
// fetchChatToken.js

export async function fetchChatToken(
  userId,
  capabilities = [],
  attributes,
  sessionDurationInMinutes) {
  const response = await fetch(`${process.env.BACKEND_BASE_URL}/create_chat_token`,
 {
    method: 'POST',
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json',
    },
```

```
    body: JSON.stringify({
      userId,
      roomIdentifier: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
  });

  const token = await response.json();

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

Beobachten von Verbindungsaktualisierungen

Das Reagieren auf Änderungen im Verbindungsstatus eines Chatrooms ist ein wesentlicher Bestandteil der Entwicklung einer Chat-App. Fangen wir mit dem Abonnieren relevanter Ereignisse an:

```
// 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>;
}
```

Als Nächstes müssen wir die Möglichkeit bieten, den Verbindungsstatus zu lesen. Wir verwenden den Hook `useState`, um einen lokalen Status in `App` zu erstellen und den Verbindungsstatus in den einzelnen Listenern festzulegen.

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>;
}
```

Zeigen Sie, nachdem Sie ihn abonniert haben, den Verbindungsstatus an und stellen Sie mithilfe der Methode `room.connect` im Hook `useEffect` eine Verbindung zum Chatroom her:

```
// 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>
);

// ...
```

Damit haben Sie erfolgreich eine Verbindung zum Chatroom hergestellt.

Erstellen einer Schaltflächenkomponente zum Senden

In diesem Abschnitt erstellen Sie eine Schaltfläche zum Senden, die für jeden Verbindungsstatus anders aussieht. Diese Schaltfläche ermöglicht das Senden von Nachrichten in einem Chatroom. Sie dient auch als optisches Signal dafür, ob/wann Nachrichten gesendet werden können, z. B. bei unterbrochenen Verbindungen oder abgelaufenen Chatsitzungen.

Erstellen Sie zunächst eine neue Datei im Verzeichnis `src` Ihres Chatterbox-Projekts und geben Sie ihr den Namen `SendButton`. Erstellen Sie anschließend eine Komponente, die eine Schaltfläche für Ihre Chatanwendung anzeigt. Exportieren Sie die `SendButton` und importieren Sie sie in `App`. Fügen Sie in den leeren `<div></div>`-Tags die Zeichenfolge `<SendButton />` hinzu.

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>
);
```

Definieren Sie als Nächstes in App eine Funktion namens onMessageSend und übergeben Sie sie an die Eigenschaft SendButton onPress. Definieren Sie eine weitere Variable namens isSendDisabled (die das Senden von Nachrichten verhindert, wenn der Chatroom nicht verbunden ist) und übergeben Sie sie an die Eigenschaft SendButton disabled.

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

// ...

const onMessageSend = () => {};

const isSendDisabled = connectionState !== 'connected';

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

// ...
```

Erstellen einer Nachrichteneingabe

Die Chatterbox-Nachrichtenleiste ist die Komponente, über die Nachrichten an einen Chatroom gesendet werden. In der Regel enthält sie eine Texteingabe zum Verfassen der Nachricht und eine Schaltfläche für deren Versand.

Um eine MessageInput-Komponente zu erstellen, erstellen Sie zunächst eine neue Datei im Verzeichnis src und geben Sie ihr den Namen MessageInput. Erstellen Sie anschließend eine Eingabekomponente, die eine Eingabe für Ihre Chatanwendung anzeigt. Exportieren Sie die MessageInput und importieren Sie sie in App (oberhalb von <SendButton />).

Erstellen Sie einen neuen Status namens messageToSend mit dem Hook useState (mit einer leeren Zeichenfolge als Standardwert). Übergeben Sie im Hauptteil Ihrer App messageToSend an den value von MessageInput und setMessageToSend an die Eigenschaft onMessageChange:

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>
);
```

Nächste Schritte

Nachdem Sie nun eine Nachrichtenleiste für Chatterbox erstellt haben, fahren Sie mit Teil 2 dieses JavaScript-Tutorials fort: Nachrichten und Ereignisse.

Client-Messaging-SDK für IVS Chat: Tutorial für JavaScript, Teil 2: Nachrichten und Ereignisse

Der vorliegende zweite (und letzte) Teil des Tutorials ist in mehrere Abschnitte unterteilt:

1. the section called “Abonnieren von Chat-Nachrichtenereignissen”
2. the section called “Anzeigen empfangener Nachrichten”

   a. the section called “Erstellen einer Nachrichtenkomponente”
   b. the section called “Erkennen von Nachrichten, die vom aktuellen Benutzer gesendet wurden”
   c. the section called “Erstellen einer Nachrichtenlistenkomponente”
   d. the section called “Rendern einer Liste von Chatnachrichten”
3. the section called “Durchführen von Aktionen in einem Chatroom”
   a. the section called “Senden einer Nachricht”
   b. the section called “Löschen einer Nachricht”
4. the section called “Nächste Schritte”

Hinweis: In einigen Fällen sind die Codebeispiele für JavaScript und TypeScript identisch, daher werden sie kombiniert.

Umfassende Informationen zum SDK finden Sie im Amazon IVS Chat Client Messaging SDK (im vorliegenden Benutzerhandbuch zu Amazon IVS Chat) und unter Chat Client Messaging: SDK for JavaScript Reference (Chat Client Messaging: SDK für JavaScript – Referenz) auf GitHub.

Voraussetzung

Absolvieren Sie unbedingt Teil 1 dieses Tutorials: Chatrooms.

Abonnieren von Chat-Nachrichtenereignissen

Mithilfe von Ereignissen informiert die Instance `ChatRoom` darüber, wann Ereignisse in einem Chatroom stattfinden. Um mit der Chatimplementierung zu beginnen, müssen Sie die Benutzer darüber informieren, wenn andere in dem Chatroom, mit dem sie verbunden sind, eine Nachricht senden.

An dieser Stelle abonnieren Sie Chat-Nachrichtenereignisse. Später zeigen wir Ihnen, wie Sie eine selbst erstellte Nachrichtenliste aktualisieren, die bei jeder Nachricht und jedem Ereignis aktualisiert wird.

Abonnieren Sie in Ihrer `App` im Hook `useEffect` alle Nachrichtenereignisse:

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

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

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

Anzeigen empfangener Nachrichten

Das Empfangen von Nachrichten ist ein zentraler Bestandteil beim Chatten. Mit dem Chat JS SDK können Sie den Code so einrichten, dass Ereignisse von anderen Benutzern, die mit einem Chatroom verbunden sind, problemlos empfangen werden.

Später zeigen wir Ihnen, wie Sie Aktionen in einem Chatroom ausführen, die die hier erstellten Komponenten nutzen.

Definieren Sie in Ihrer `App` einen Status namens `messages` mit einem `ChatMessage`-Array-Typ namens `messages`:

TypeScript

```
// App.tsx

// ...

import { ChatRoom, ChatMessage, ConnectionState } from 'amazon-ivs-chat-messaging';

export default function App() {
  const [messages, setMessages] = useState<ChatMessage[]>([]);

  //...
}
```

JavaScript

```
// App.jsx

// ...

export default function App() {
  const [messages, setMessages] = useState([]);
```

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

Als Nächstes fügen Sie in der Listener-Funktion `message` die Zeichenfolge `message` an das Array `messages` an:

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

// ...

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

// ...
```

Im Folgenden gehen wir die Aufgaben zum Anzeigen empfangener Nachrichten Schritt für Schritt durch:

1. the section called “Erstellen einer Nachrichtenkomponente”
2. the section called “Erkennen von Nachrichten, die vom aktuellen Benutzer gesendet wurden”
3. the section called “Erstellen einer Nachrichtenlistenkomponente”
4. the section called “Rendern einer Liste von Chatnachrichten”

Erstellen einer Nachrichtenkomponente

Die Komponente `Message` rendert den Inhalt einer Nachricht, die im Chatroom empfangen wurde. In diesem Abschnitt erstellen Sie eine Nachrichtenkomponente zum Rendern einzelner Chatnachrichten in der `App`.

Erstellen Sie im Verzeichnis `src` eine neue Datei und geben Sie ihr den Namen `Message`. Übergeben Sie den `ChatMessage`-Typ für diese Komponente und die Zeichenfolge `content` aus den `ChatMessage`-Eigenschaften, um den Nachrichtentext anzuzeigen, der von Listenern für Chatroom-Nachrichten empfangen wurde. Wechseln Sie im Projektnavigator zu `Message`.

TypeScript

```
// Message.tsx
```

```
import * as React from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';

type Props = {
  message: ChatMessage;
}

export const Message = ({ message }: Props) => {
  return (
    <div style={{ backgroundColor: 'silver', padding: 6, borderRadius: 10, margin:
 10 }}>
      <p>{message.content}</p>
    </div>
  );
};
```

JavaScript

```
// Message.jsx

import * as React from 'react';

export const Message = ({ message }) => {
  return (
    <div style={{ backgroundColor: 'silver', padding: 6, borderRadius: 10, margin:
 10 }}>
      <p>{message.content}</p>
    </div>
  );
};
```

Tipp: Mit dieser Komponente können Sie verschiedene Eigenschaften speichern, die in den Nachrichtenzeilen gerendert werden sollen, z. B. Avatar-URLs, Benutzernamen und Zeitstempel für den Nachrichtenversand.

Erkennen von Nachrichten, die vom aktuellen Benutzer gesendet wurden

Um die vom aktuellen Benutzer gesendete Nachricht zu erkennen, ändern wir den Code und erstellen einen React-Kontext zum Speichern der `userId` des aktuellen Benutzers.

Erstellen Sie im Verzeichnis `src` eine neue Datei und geben Sie ihr den Namen `UserContext`:

TypeScript

```
// UserContext.tsx

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

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

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

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

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

  return context;
};

type UserProviderType = {
  children: ReactNode;
}

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

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

JavaScript

```
// UserContext.jsx

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

const UserContext = createContext(undefined);

export const useUserContext = () => {
```

```
  const context = useContext(UserContext);

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

  return context;
};

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

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

Hinweis: Hier haben wir den Wert `userId` mit dem Hook `useState` gespeichert. Künftig können Sie zum Ändern des Benutzerkontexts oder zum Anmelden `setUserId` verwenden.

Ersetzen Sie als Nächstes `userId` im ersten Parameter, der an `tokenProvider` übergeben wurde. Verwenden Sie dabei den zuvor erstellten Kontext:

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

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

Verwenden Sie in Ihrer `Message`-Komponente den zuvor erstellten `UserContext`, deklarieren Sie die Variable `isMine`, ordnen Sie die `userId` des Absenders der `userId` aus dem Kontext zu und wenden Sie verschiedene Nachrichtenstile für den aktuellen Benutzer an.

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>
  );
};
```

Erstellen einer Nachrichtenlistenkomponente

Die Komponente `MessageList` zeigt die Konversation eines Chatrooms im Laufe der Zeit an. Die Datei `MessageList` ist der Container, der alle Nachrichten enthält. `Message` ist eine Zeile in `MessageList`.

Erstellen Sie im Verzeichnis `src` eine neue Datei und geben Sie ihr den Namen `MessageList`. Definieren Sie `Props` mit `messages` vom Typ `ChatMessage`-Array. Ordnen Sie im Hauptteil die Eigenschaft `messages` zu und übergeben Sie `Props` an Ihre `Message`-Komponente.

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>
  );
};
```

Rendern einer Liste von Chatnachrichten

Fügen Sie jetzt Ihre neue `MessageList` in die App-Hauptkomponente ein:

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

import { MessageList } from './MessageList';
// ...

return (
  <div style={{ display: 'flex', flexDirection: 'column', padding: 10 }}>
    <h4>Connection State: {connectionState}</h4>
    <MessageList messages={messages} />
    <div style={{ flexDirection: 'row', display: 'flex', width: '100%',
 backgroundColor: 'red' }}>
      <MessageInput value={messageToSend} onValueChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </div>
  </div>
);

// ...
```

Alle Puzzleteile sind jetzt vorhanden, damit Ihre App Nachrichten rendern kann, die aus dem Chatroom empfangen wurden. Nachfolgend erfahren Sie, wie Sie in einem Chatroom Aktionen ausführen, die die von Ihnen erstellten Komponenten nutzen.

Durchführen von Aktionen in einem Chatroom

Das Senden von Nachrichten und das Durchführen von Moderatorenaktionen sind einige der wichtigsten Formen der Interaktion mit einem Chatroom. Hier erfahren Sie, wie Sie mithilfe verschiedener `ChatRequest`-Objekte allgemeine Aktionen in Chatterbox durchführen. Dazu gehören das Senden und Löschen von Nachrichten sowie das Trennen der Verbindung anderer Benutzer.

Alle Aktionen in einem Chatroom folgen einem gemeinsamen Muster: Für jede Aktion, die in einem Chatroom durchgeführt wird, gibt es ein entsprechendes Anforderungsobjekt. Für jede Anforderung gibt es ein entsprechendes Antwortobjekt, das bei Bestätigung der Anforderung empfangen wird.

Solange den Benutzern beim Erstellen eines Chat-Tokens die richtigen Berechtigungen erteilt werden, können sie die entsprechenden Aktionen erfolgreich durchführen. Mithilfe der Anforderungsobjekte lässt sich feststellen, welche Anforderungen in einem Chatroom durchgeführt werden können.

Nachfolgend erklären wir das Senden einer Nachricht und das Löschen einer Nachricht.

Senden einer Nachricht

Die Klasse `SendMessageRequest` ermöglicht das Senden von Nachrichten in einem Chatroom. Hier ändern Sie Ihre App, um mit der Komponente, die Sie unter Erstellen einer Nachrichteneingabe (in Teil 1 dieses Tutorials) erstellt haben, eine Nachrichtenanforderung zu senden.

Definieren Sie zunächst eine neue boolesche Eigenschaft namens `isSending` mit dem Hook `useState`. Schalten Sie mithilfe dieser neuen Eigenschaft den deaktivierten Status des HTML-Elements `button` um. Verwenden Sie dabei die Konstante `isSendDisabled`. Löschen Sie im Event-Handler für `SendButton` den Wert für `messageToSend` und stellen Sie `isSending` auf „true“ ein.

Da über diese Schaltfläche ein API-Aufruf getätigt wird, verhindert das Hinzufügen des booleschen Werts *`isSending`*, dass mehrere API-Aufrufe gleichzeitig ausgeführt werden. Dazu werden Benutzerinteraktionen für *`SendButton`* deaktiviert, bis die Anforderung abgeschlossen ist.

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

```
// ...

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

// ...

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

// ...

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

// ...
```

Bereiten Sie die Anforderung vor, indem Sie eine neue `SendMessageRequest`-Instance erstellen und den Nachrichteninhalt an den Konstruktor übergeben. Rufen Sie nach dem Festlegen des Status von `isSending` und `messageToSend` die Methode `sendMessage` auf, die die Anforderung an den Chatroom sendet. Löschen Sie abschließend das Flag `isSending`, sobald Sie eine Bestätigung oder Ablehnung der Anforderung erhalten haben.

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);
  }
};

// ...
```

Führen Sie Chatterbox aus: Senden Sie eine Nachricht, indem Sie eine Nachricht mit `MessageInput` verfassen und auf `SendButton` tippen. Die gesendete Nachricht sollte in der `MessageList`, die Sie zuvor erstellt haben, gerendert werden.

Löschen einer Nachricht

Um eine Nachricht aus einem Chatroom zu löschen, benötigen Sie die entsprechende Fähigkeit. Fähigkeiten werden bei der Initialisierung des Chat-Tokens gewährt, das Sie bei der Authentifizierung in einem Chatroom verwenden. Für die Zwecke dieses Abschnitts können Sie in der `ServerApp` aus Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers (in Teil 1 dieses Tutorials) die

Moderatorfähigkeiten festlegen. Dies geschieht in der App mithilfe des Objekts `tokenProvider`, das Sie unter Erstellen eines Token-Anbieters (ebenfalls in Teil 1) erstellt haben.

Hier ändern Sie Ihre `Message`, indem Sie eine Funktion zum Löschen der Nachricht hinzufügen.

Öffnen Sie zunächst `App.tsx` und fügen Sie die Fähigkeit `DELETE_MESSAGE` hinzu. (`capabilities` ist der zweite Parameter der Funktion `tokenProvider`.)

Hinweis: Auf diese Weise informiert die `ServerApp` die IVS-Chat-APIs darüber, dass der Benutzer, der mit dem resultierenden Chat-Token verknüpft wird, Nachrichten in einem Chatroom löschen kann. In einer realen Umgebung wird die Backend-Logik zur Verwaltung der Benutzerfähigkeiten in der Infrastruktur Ihrer Server-App wahrscheinlich komplexer sein.

TypeScript

```
// App.tsx

// ...

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

// ...
```

JavaScript

```
// App.jsx

// ...

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

```
// ...
```

In den nächsten Schritten aktualisieren Sie Ihre `Message`, um eine Schaltfläche zum Löschen anzuzeigen.

Öffnen Sie `Message` und definieren Sie einen neuen booleschen Status namens `isDeleting`. Verwenden Sie dabei den Hook `useState` mit dem Anfangswert `false`. Aktualisieren Sie mit diesem Status den Inhalt von `Button`, sodass die Schaltfläche je nach aktuellem Status von `isDeleting` unterschiedlich aussieht. Deaktivieren Sie die Schaltfläche, wenn `isDeleting` „true“ ist. Dadurch wird verhindert, dass zwei Anforderungen zum Löschen einer Nachricht gleichzeitig gestellt werden.

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>
  );
};
```

Definieren Sie eine neue Funktion namens onDelete, die eine Zeichenfolge als einen möglichen Parameter akzeptiert und Promise zurückgibt. Verwenden Sie im Hauptteil des Abschlusses der Aktion von Button den Code setIsDeleting, um den booleschen Wert isDeleting vor und nach einem Aufruf von onDelete umzuschalten. Übergeben Sie als Zeichenfolgenparameter die ID der Komponentennachricht.

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>
  );
};
```

Als Nächstes aktualisieren Sie die `MessageList`, damit die neuesten Änderungen an Ihrer `Message`-Komponente wiedergegeben werden.

Öffnen Sie `MessageList` und definieren Sie eine neue Funktion namens `onDelete`, die eine Zeichenfolge als einen Parameter akzeptiert und `Promise` zurückgibt. Aktualisieren Sie Ihre `Message` und übergeben Sie sie durch die Eigenschaften von `Message`. Der Zeichenfolgenparameter im neuen Abschluss ist die ID der zu löschenden Nachricht, die aus Ihrer `Message` übergeben wird.

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

Als Nächstes aktualisieren Sie die App, damit die neuesten Änderungen an der `MessageList` wiedergegeben werden.

Definieren Sie in App eine Funktion namens on`DeleteMessage` und übergeben Sie sie an die Eigenschaft `MessageList` on`Delete`:

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>
);

// ...
```

Bereiten Sie eine Anforderung vor, indem Sie eine neue Instance von `DeleteMessageRequest` erstellen und die entsprechende Nachrichten-ID an den Konstruktorparameter übergeben. Rufen Sie dann den Befehl `deleteMessage` auf, der die oben vorbereitete Anforderung akzeptiert:

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);
};

// ...
```

Als Nächstes aktualisieren Sie den Status `messages`, damit eine neue Liste von Nachrichten – ohne die gerade gelöschte Nachricht – angezeigt wird.

Warten Sie im Hook `useEffect` auf das Ereignis `messageDelete` und aktualisieren Sie das Status-Array `messages`, indem Sie die Nachricht mit einer zum Parameter `message` passenden ID löschen.

Hinweis: Das Ereignis `messageDelete` kann ausgelöst werden, wenn Nachrichten vom aktuellen Benutzer oder von anderen Benutzern im Chatroom gelöscht wurden. Wenn Sie es im Ereignishandler verarbeiten (statt neben der Anforderung `deleteMessage`), können Sie das Löschen von Nachrichten vereinheitlichen.

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

// ...

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

return () => {
```

```
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

Sie können jetzt Benutzer aus einem Chatroom in Ihrer Chat-App löschen.

Nächste Schritte

Versuchen Sie als Experiment, andere Aktionen in einem Chatroom zu implementieren, z. B. das Trennen der Verbindung eines anderen Benutzers.

Client-Messaging-SDK für IVS Chat: Tutorial für React Native, Teil 1: Chaträume

Hierbei handelt es sich um den ersten Teil eines zweiteiligen Tutorials. Darin lernen Sie die Grundlagen der Arbeit mit dem SDK für Amazon IVS Chat Client Messaging JavaScript kennen, indem Sie eine voll funktionsfähige Anwendung mit React Native entwickeln. Wir nennen die App Chatterbox.

Die Zielgruppe sind erfahrene Entwickler, die das Amazon IVS Chat Messaging SDK noch nicht kennen. Sie sollten mit der Programmiersprache TypeScript oder Java Script und der Bibliothek React vertraut sein.

Der Kürze halber bezeichnen wir das Amazon IVS Chat Client Messaging JavaScript SDK als Chat JS SDK.

Hinweis: In einigen Fällen sind die Codebeispiele für JavaScript und TypeScript identisch, daher werden sie kombiniert.

Der vorliegende erste Teil des Tutorials ist in mehrere Abschnitte unterteilt:

1. the section called “Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers”
2. the section called “Erstellen eines Chatterbox-Projekts”
3. the section called “Verbinden mit einem Chatroom”
4. the section called “Erstellen eines Token-Anbieters”

5. the section called “Beobachten von Verbindungsaktualisierungen”
6. the section called “Erstellen einer Schaltflächenkomponente zum Senden”
7. the section called “Erstellen einer Nachrichteneingabe”
8. the section called “Nächste Schritte”

Voraussetzungen

- Sie sollten mit TypeScript oder JavaScript und der Bibliothek React Native vertraut sein. Sollten Sie mit React Native nicht vertraut seien, können Sie unter Einführung in React Native mehr über die Grundlagen erfahren.
- Lesen Sie sich Erste Schritte mit IVS Chat durch.
- Erstellen Sie einen AWS-IAM-Benutzer mit den Fähigkeiten CreateChatToken und CreateRoom, die in einer vorhandenen IAM-Richtlinie definiert sind. (Siehe). Erste Schritte mit IVS Chat.)
- Stellen Sie sicher, dass die Geheim-/Zugriffsschlüssel für diesen Benutzer in einer Datei mit den AWS-Anmeldeinformationen gespeichert sind. Entsprechende Anweisungen finden Sie im Benutzerhandbuch zur AWS-CLI (insbesondere unter Einstellungen für Konfigurations- und Anmeldeinformationsdateien).
- Erstellen Sie einen Chatroom und speichern Sie dessen ARN. Siehe Erste Schritte mit IVS Chat. (Wenn Sie den ARN nicht speichern, können Sie ihn später über die Konsole oder die Chat-API nachschlagen.)
- Installieren Sie die Umgebung Node.js 14+ mit dem Paketmanager NPM oder Yarn.

Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers

Die Erstellung von Chatrooms und die Generierung der Chat-Token, die benötigt werden, damit das Chat JS SDK die Clients für Ihre Chatrooms authentifizieren und autorisieren kann, erfolgt in Ihrer Backend-Anwendung. Sie müssen Ihr eigenes Backend verwenden, da die AWS-Schlüssel nicht sicher in einer mobilen App gespeichert werden können. Versierte Angreifer könnten diese extrahieren und Zugriff auf Ihr AWS-Konto erlangen.

Weitere Informationen finden Sie unter Erstellen eines Chat-Tokens unter Erste Schritte mit Amazon IVS Chat. Wie im dortigen Flussdiagramm gezeigt, erfolgt die Erstellung eines Chat-Tokens in Ihrer serverseitigen Anwendung. Das bedeutet, dass Ihre App eigene Mittel zur Generierung eines Chat-Tokens bereitstellen muss, indem sie ein Token von der serverseitigen Anwendung anfordert.

In diesem Abschnitt werden die Grundlagen der Erstellung eines Token-Anbieters in Ihrem Backend vermittelt. Mit dem Express-Framework erstellen wir einen lokalen Live-Server, der die Erstellung von Chat-Token mithilfe Ihrer lokalen AWS-Umgebung verwaltet.

Erstellen Sie ein leeres npm-Projekt mit NPM. Erstellen Sie ein Verzeichnis für Ihre Anwendung und machen Sie dieses zu Ihrem Arbeitsverzeichnis:

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

Erstellen Sie mit `npm init` eine `package.json`-Datei für die Anwendung:

```
$ npm init
```

Durch diesen Befehl werden Sie aufgefordert, unter anderem den Namen und die Version der Anwendung einzugeben. Drücken Sie vorerst einfach RETURN, um die Standardwerte zu akzeptieren, mit der folgenden Ausnahme:

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

Drücken Sie RETURN, um den vorgeschlagenen Standarddateinamen `index.js` zu akzeptieren, oder geben Sie den gewünschten Namen für die Hauptdatei ein.

Installieren Sie nun die erforderlichen Abhängigkeiten:

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

`aws-sdk` erfordert Variablen der Konfigurationsumgebung, die automatisch aus einer Datei namens `.env` im Stammverzeichnis geladen werden. Um sie zu konfigurieren, erstellen Sie eine neue Datei mit dem Namen `.env` und tragen Sie die fehlenden Konfigurationsinformationen ein:

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

Jetzt erstellen wir im Stammverzeichnis eine Eintrittspunktdatei mit dem Namen, den Sie oben im Befehl `npm init` eingegeben haben. In diesem Fall verwenden wir `index.js` und importieren alle erforderlichen Pakete:

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

Erstellen Sie jetzt eine neue Instance von `express`:

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

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

Danach können Sie die erste Endpunkt-POST-Methode für den Token-Anbieter erstellen. Entnehmen Sie die erforderlichen Parameter dem Hauptteil der Anforderung (`roomId`, `userId`, `capabilities` und `sessionDurationInMinutes`):

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

Fügen Sie die Validierung der erforderlichen Felder hinzu:

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

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

```
  }
});
```

Nach der Vorbereitung der POST-Methode integrieren wir `createChatToken` mit `aws-sdk` für die Kernfunktionalität der Authentifizierung/Autorisierung:

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

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

  ivsChat.createChatToken({ roomIdentifier, userId, capabilities,
 sessionDurationInMinutes }, (error, data) => {
    if (error) {
      console.log(error);
      res.status(500).send(error.code);
    } else if (data.token) {
      const { token, sessionExpirationTime, tokenExpirationTime } = data;
      console.log(`Retrieved Chat Token: ${JSON.stringify(data, null, 2)}`);

      res.json({ token, sessionExpirationTime, tokenExpirationTime });
    }
  });
});
```

Fügen Sie am Ende der Datei einen Port-Listener für die App `express` hinzu:

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

Nun können Sie den Server mit dem folgenden Befehl im Stammverzeichnis des Projekts ausführen:

```
$ node index.js
```

Tipp: Dieser Server akzeptiert URL-Anforderungen unter https://localhost:3000.

Erstellen eines Chatterbox-Projekts

Zunächst erstellen Sie das React-Native-Projekt namens `chatterbox`. Führen Sie diesen Befehl aus:

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

Oder erstellen Sie ein Expo-Projekt mit einer TypeScript-Vorlage.

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

Sie können das Chat Client Messaging JS SDK über den Node Package Manager (NPM) oder den Paketmanager Yarn integrieren:

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

Verbinden mit einem Chatroom

Hier erstellen Sie einen `ChatRoom` und stellen mit asynchronen Methoden eine Verbindung dazu her. Die Klasse `ChatRoom` verwaltet die Verbindung der Benutzer zum Chat JS SDK. Um eine Verbindung zu einem Chatroom herzustellen, müssen Sie eine Instance von `ChatToken` in Ihrer React-Anwendung angeben.

Navigieren Sie zur App-Datei, die im `chatterbox`-Standardprojekt erstellt wurde, und löschen Sie alles, was eine funktionale Komponente zurückgibt. Von dem vorab eingetragenen Code wird nichts benötigt. An dieser Stelle ist unsere App ziemlich leer.

TypeScript/JavaScript:

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

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

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

Erstellen Sie eine neue `ChatRoom`-Instance und übergeben Sie sie mit dem Hook `useState` an den Status. Das erfordert die Übergabe von `regionOrUrl` (die AWS-Region, in der der Chatroom gehostet wird) und `tokenProvider` (wird für den Authentifizierungs- und Autorisierungsablauf im Backend verwendet, der in nachfolgenden Schritten erstellt wird).

Wichtig: Sie müssen dieselbe AWS-Region verwenden, in der Sie den Chatroom unter Erste Schritte mit Amazon IVS Chat erstellt haben. Die API ist ein regionaler AWS-Service. Eine Liste der unterstützten Regionen und HTTPS-Service-Endpunkte für Amazon IVS Chat finden Sie auf der Seite Regionen für Amazon IVS Chat.

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>;
}
```

Erstellen eines Token-Anbieters

Als nächsten Schritt müssen wir eine parameterlose `tokenProvider`-Funktion erstellen, die vom `ChatRoom`-Konstruktor benötigt wird. Zunächst erstellen wir eine `fetchChatToken`-Funktion, die eine POST-Anforderung an die Backend-Anwendung übermittelt, die Sie unter the section called “Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers” eingerichtet haben. Chat-Token enthalten die notwendigen Informationen, damit das SDK erfolgreich eine Verbindung zum Chatroom herstellen kann. Die Chat-API nutzt diese Token als sichere Methode, um die Identität von Benutzern, die Fähigkeiten innerhalb eines Chatrooms und die Sitzungsdauer zu validieren.

Erstellen Sie im Projektnavigator eine neue TypeScript/JavaScript-Datei mit dem Namen `fetchChatToken`. Erstellen Sie eine Abrufanforderung an die `backend`-Anwendung und

geben Sie das Objekt `ChatToken` aus der Antwort zurück. Fügen Sie die Eigenschaften des Anforderungshauptteils hinzu, die für die Erstellung eines Chat-Tokens erforderlich sind. Verwenden Sie die für Amazon-Ressourcennamen (ARNs) definierten Regeln. Diese Eigenschaften sind im Vorgang CreateChatToken dokumentiert.

Hinweis: Die hier verwendete URL ist dieselbe URL, die Ihr lokaler Server beim Ausführen der Backend-Anwendung erstellt hat.

TypeScript

```
// fetchChatToken.ts

import { ChatToken } from 'amazon-ivs-chat-messaging';

type UserCapability = 'DELETE_MESSAGE' | 'DISCONNECT_USER' | 'SEND_MESSAGE';

export async function fetchChatToken(
  userId: string,
  capabilities: UserCapability[] = [],
  attributes?: Record<string, string>,
  sessionDurationInMinutes?: number,
): Promise<ChatToken> {
  const response = await fetch(`${process.env.BACKEND_BASE_URL}/create_chat_token`,
 {
    method: 'POST',
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      userId,
      roomIdentifier: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
  });

  const token = await response.json();

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

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

JavaScript

```
// fetchChatToken.js

export async function fetchChatToken(
  userId,
  capabilities = [],
  attributes,
  sessionDurationInMinutes) {
  const response = await fetch(`${process.env.BACKEND_BASE_URL}/create_chat_token`,
 {
    method: 'POST',
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      userId,
      roomIdentifier: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
  });

  const token = await response.json();

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

Beobachten von Verbindungsaktualisierungen

Das Reagieren auf Änderungen im Verbindungsstatus eines Chatrooms ist ein wesentlicher Bestandteil der Entwicklung einer Chat-App. Fangen wir mit dem Abonnieren relevanter Ereignisse an:

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>;
}
```

Als Nächstes müssen wir die Möglichkeit bieten, den Verbindungsstatus zu lesen. Wir verwenden den Hook `useState`, um einen lokalen Status in App zu erstellen und den Verbindungsstatus in den einzelnen Listenern festzulegen.

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>;
}
```

Zeigen Sie, nachdem Sie ihn abonniert haben, den Verbindungsstatus an und stellen Sie mithilfe der Methode `room.connect` im Hook `useEffect` eine Verbindung zum Chatroom her:

TypeScript/JavaScript:

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

// ...

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

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

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

  room.connect();

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

// ...

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

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

// ...
```

Damit haben Sie erfolgreich eine Verbindung zum Chatroom hergestellt.

Erstellen einer Schaltflächenkomponente zum Senden

In diesem Abschnitt erstellen Sie eine Schaltfläche zum Senden, die für jeden Verbindungsstatus anders aussieht. Diese Schaltfläche ermöglicht das Senden von Nachrichten in einem Chatroom. Sie dient auch als optisches Signal dafür, ob/wann Nachrichten gesendet werden können, z. B. bei unterbrochenen Verbindungen oder abgelaufenen Chatsitzungen.

Erstellen Sie zunächst eine neue Datei im Verzeichnis `src` Ihres Chatterbox-Projekts und geben Sie ihr den Namen `SendButton`. Erstellen Sie anschließend eine Komponente, die eine Schaltfläche für Ihre Chatanwendung anzeigt. Exportieren Sie die `SendButton` und importieren Sie sie in `App`. Fügen Sie in den leeren `<View></View>`-Tags die Zeichenfolge `<SendButton />` hinzu.

TypeScript

```
// SendButton.tsx

import React from 'react';
import { TouchableOpacity, Text, ActivityIndicator, StyleSheet } from 'react-
native';

interface Props {
  onPress?: () => void;
  disabled: boolean;
  loading: boolean;
}

export const SendButton = ({ onPress, disabled, loading }: Props) => {
  return (
    <TouchableOpacity style={styles.root} disabled={disabled} onPress={onPress}>
      {loading ? <Text>Send</Text> : <ActivityIndicator />}
    </TouchableOpacity>
  );
```

```
};

const styles = StyleSheet.create({
  root: {
    width: 50,
    height: 50,
    borderRadius: 30,
    marginLeft: 10,
    justifyContent: 'center',
    alignContent: 'center',
  }
});

// App.tsx

import { SendButton } from './SendButton';

// ...

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

JavaScript

```
// SendButton.jsx

import React from 'react';
import { TouchableOpacity, Text, ActivityIndicator, StyleSheet } from 'react-
native';

export const SendButton = ({ onPress, disabled, loading }) => {
  return (
    <TouchableOpacity style={styles.root} disabled={disabled} onPress={onPress}>
      {loading ? <Text>Send</Text> : <ActivityIndicator />}
    </TouchableOpacity>
  );
};

const styles = StyleSheet.create({
```

```
  root: {
    width: 50,
    height: 50,
    borderRadius: 30,
    marginLeft: 10,
    justifyContent: 'center',
    alignContent: 'center',
  }
});

// App.jsx

import { SendButton } from './SendButton';

// ...

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

Definieren Sie als Nächstes in App eine Funktion namens onMessageSend und übergeben Sie sie an die Eigenschaft SendButton onPress. Definieren Sie eine weitere Variable namens isSendDisabled (die das Senden von Nachrichten verhindert, wenn der Chatroom nicht verbunden ist) und übergeben Sie sie an die Eigenschaft SendButton disabled.

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>
);

// ...
```

Erstellen einer Nachrichteneingabe

Die Chatterbox-Nachrichtenleiste ist die Komponente, über die Nachrichten an einen Chatroom gesendet werden. In der Regel enthält sie eine Texteingabe zum Verfassen der Nachricht und eine Schaltfläche für deren Versand.

Um eine `MessageInput`-Komponente zu erstellen, erstellen Sie zunächst eine neue Datei im Verzeichnis `src` und geben Sie ihr den Namen `MessageInput`. Erstellen Sie anschließend eine Eingabekomponente, die eine Eingabe für Ihre Chatanwendung anzeigt. Exportieren Sie die `MessageInput` und importieren Sie sie in `App` (oberhalb von `<SendButton />`).

Erstellen Sie einen neuen Status namens `messageToSend` mit dem Hook `useState` (mit einer leeren Zeichenfolge als Standardwert). Übergeben Sie im Hauptteil Ihrer App `messageToSend` an den `value` von `MessageInput` und `setMessageToSend` an die Eigenschaft `onMessageChange`:

TypeScript

```
// MessageInput.tsx

import * as React from 'react';

interface Props {
  value?: string;
  onValueChange?: (value: string) => void;
}

export const MessageInput = ({ value, onValueChange }: Props) => {
  return (
    <TextInput style={styles.input} value={value} onChangeText={onValueChange}
 placeholder="Send a message" />
  );
};

const styles = StyleSheet.create({
  input: {
    fontSize: 20,
    backgroundColor: 'rgb(239,239,240)',
```

```
    paddingHorizontal: 18,
    paddingVertical: 15,
    borderRadius: 50,
    flex: 1,
  }
})

// App.tsx

// ...

import { MessageInput } from './MessageInput';

// ...

export default function App() {
  const [messageToSend, setMessageToSend] = useState('');

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <View style={styles.messageBar}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </View>
  </SafeAreaView>
);

const styles = StyleSheet.create({
  root: {
    flex: 1,
  },
  messageBar: {
    borderTopWidth: StyleSheet.hairlineWidth,
    borderTopColor: 'rgb(160,160,160)',
    flexDirection: 'row',
    padding: 16,
    alignItems: 'center',
    backgroundColor: 'white',
  }
});
```

JavaScript

```
// MessageInput.jsx

import * as React from 'react';

export const MessageInput = ({ value, onValueChange }) => {
  return (
    <TextInput style={styles.input} value={value} onChangeText={onValueChange}
 placeholder="Send a message" />
  );
};

const styles = StyleSheet.create({
  input: {
    fontSize: 20,
    backgroundColor: 'rgb(239,239,240)',
    paddingHorizontal: 18,
    paddingVertical: 15,
    borderRadius: 50,
    flex: 1,
  }
})

// App.jsx

// ...

import { MessageInput } from './MessageInput';

// ...

export default function App() {
  const [messageToSend, setMessageToSend] = useState('');

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <View style={styles.messageBar}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </View>
```

```
  </SafeAreaView>
);

const styles = StyleSheet.create({
  root: {
    flex: 1,
  },
  messageBar: {
    borderTopWidth: StyleSheet.hairlineWidth,
    borderTopColor: 'rgb(160,160,160)',
    flexDirection: 'row',
    padding: 16,
    alignItems: 'center',
    backgroundColor: 'white',
  }
});
```

Nächste Schritte

Nachdem Sie nun eine Nachrichtenleiste für Chatterbox erstellt haben, fahren Sie mit Teil 2 dieses React-Native-Tutorials fort: Nachrichten und Ereignisse.

Client-Messaging-SDK für IVS Chat: Tutorial für React Native, Teil 2: Nachrichten und Ereignisse

Der vorliegende zweite (und letzte) Teil des Tutorials ist in mehrere Abschnitte unterteilt:

1. the section called “Abonnieren von Chat-Nachrichtenereignissen”
2. the section called “Anzeigen empfangener Nachrichten”
   a. the section called “Erstellen einer Nachrichtenkomponente”
   b. the section called “Erkennen von Nachrichten, die vom aktuellen Benutzer gesendet wurden”
   c. the section called “Rendern einer Liste von Chatnachrichten”
3. the section called “Durchführen von Aktionen in einem Chatroom”
   a. the section called “Senden einer Nachricht”
   b. the section called “Löschen einer Nachricht”
4. the section called “Nächste Schritte”

Hinweis: In einigen Fällen sind die Codebeispiele für JavaScript und TypeScript identisch, daher werden sie kombiniert.

Voraussetzung

Absolvieren Sie unbedingt Teil 1 dieses Tutorials: Chatrooms.

Abonnieren von Chat-Nachrichtenereignissen

Mithilfe von Ereignissen informiert die Instance `ChatRoom` darüber, wann Ereignisse in einem Chatroom stattfinden. Um mit der Chatimplementierung zu beginnen, müssen Sie die Benutzer darüber informieren, wenn andere in dem Chatroom, mit dem sie verbunden sind, eine Nachricht senden.

An dieser Stelle abonnieren Sie Chat-Nachrichtenereignisse. Später zeigen wir Ihnen, wie Sie eine selbst erstellte Nachrichtenliste aktualisieren, die bei jeder Nachricht und jedem Ereignis aktualisiert wird.

Abonnieren Sie in Ihrer `App` im Hook `useEffect` alle Nachrichtenereignisse:

TypeScript/JavaScript:

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

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

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

Anzeigen empfangener Nachrichten

Das Empfangen von Nachrichten ist ein zentraler Bestandteil beim Chatten. Mit dem Chat JS SDK können Sie den Code so einrichten, dass Ereignisse von anderen Benutzern, die mit einem Chatroom verbunden sind, problemlos empfangen werden.

Später zeigen wir Ihnen, wie Sie Aktionen in einem Chatroom ausführen, die die hier erstellten Komponenten nutzen.

Definieren Sie in Ihrer App einen Status namens `messages` mit einem `ChatMessage`-Array-Typ namens `messages`:

TypeScript

```
// App.tsx

// ...

import { ChatRoom, ChatMessage, ConnectionState } from 'amazon-ivs-chat-messaging';

export default function App() {
  const [messages, setMessages] = useState<ChatMessage[]>([]);

  //...
}
```

JavaScript

```
// App.jsx

// ...

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

export default function App() {
  const [messages, setMessages] = useState([]);

  //...
}
```

Als Nächstes fügen Sie in der Listener-Funktion `message` die Zeichenfolge `message` an das Array `messages` an:

TypeScript/JavaScript:

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

// ...

const unsubscribeOnMessageReceived = room.addListener('message', (message) => {
```

```
  setMessages((msgs) => [...msgs, message]);
});

// ...
```

Im Folgenden gehen wir die Aufgaben zum Anzeigen empfangener Nachrichten Schritt für Schritt durch:

1. the section called “Erstellen einer Nachrichtenkomponente”
2. the section called “Erkennen von Nachrichten, die vom aktuellen Benutzer gesendet wurden”
3. the section called “Rendern einer Liste von Chatnachrichten”

Erstellen einer Nachrichtenkomponente

Die Komponente `Message` rendert den Inhalt einer Nachricht, die im Chatroom empfangen wurde. In diesem Abschnitt erstellen Sie eine Nachrichtenkomponente zum Rendern einzelner Chatnachrichten in der `App`.

Erstellen Sie im Verzeichnis `src` eine neue Datei und geben Sie ihr den Namen `Message`. Übergeben Sie den `ChatMessage`-Typ für diese Komponente und die Zeichenfolge `content` aus den `ChatMessage`-Eigenschaften, um den Nachrichtentext anzuzeigen, der von Listenern für Chatroom-Nachrichten empfangen wurde. Wechseln Sie im Projektnavigator zu `Message`.

TypeScript

```
// Message.tsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';

type Props = {
  message: ChatMessage;
}

export const Message = ({ message }: Props) => {
  return (
    <View style={styles.root}>
      <Text>{message.sender.userId}</Text>
      <Text style={styles.textContent}>{message.content}</Text>
```

```
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
});
```

JavaScript

```
// Message.jsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';

export const Message = ({ message }) => {
  return (
    <View style={styles.root}>
      <Text>{message.sender.userId}</Text>
      <Text style={styles.textContent}>{message.content}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
```

```
    marginRight: 50,
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
});
```

Tipp: Mit dieser Komponente können Sie verschiedene Eigenschaften speichern, die in den Nachrichtenzeilen gerendert werden sollen, z. B. Avatar-URLs, Benutzernamen und Zeitstempel für den Nachrichtenversand.

Erkennen von Nachrichten, die vom aktuellen Benutzer gesendet wurden

Um die vom aktuellen Benutzer gesendete Nachricht zu erkennen, ändern wir den Code und erstellen einen React-Kontext zum Speichern der `userId` des aktuellen Benutzers.

Erstellen Sie im Verzeichnis `src` eine neue Datei und geben Sie ihr den Namen `UserContext`:

TypeScript

```
// UserContext.tsx

import React from 'react';

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

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

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

  return context;
};

export const UserProvider = UserContext.Provider;
```

JavaScript

```
// UserContext.jsx

import React from 'react';

const UserContext = React.createContext(undefined);

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

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

  return context;
};

export const UserProvider = UserContext.Provider;
```

Hinweis: Hier haben wir den Wert `userId` mit dem Hook `useState` gespeichert. Künftig können Sie zum Ändern des Benutzerkontexts oder zum Anmelden `setUserId` verwenden.

Ersetzen Sie als Nächstes `userId` im ersten Parameter, der an `tokenProvider` übergeben wurde. Verwenden Sie dabei den zuvor erstellten Kontext. Stellen Sie sicher, dass Sie Ihrem Token-Anbieter die `SEND_MESSAGE`-Funktion hinzufügen, wie unten angegeben. Sie ist erforderlich, um Nachrichten zu senden:

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']),
      }),
  );

  // ...
}
```

Verwenden Sie in Ihrer `Message`-Komponente den zuvor erstellten `UserContext`, deklarieren Sie die Variable `isMine`, ordnen Sie die `userId` des Absenders der `userId` aus dem Kontext zu und wenden Sie verschiedene Nachrichtenstile für den aktuellen Benutzer an.

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',
  },
});
```

Rendern einer Liste von Chatnachrichten

Listen Sie jetzt Nachrichten auf, indem Sie `FlatList` und eine `Message`-Komponente verwenden:

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>
);

// ...
```

Alle Puzzleteile sind jetzt vorhanden, damit Ihre App Nachrichten rendern kann, die aus dem Chatroom empfangen wurden. Nachfolgend erfahren Sie, wie Sie in einem Chatroom Aktionen ausführen, die die von Ihnen erstellten Komponenten nutzen.

Durchführen von Aktionen in einem Chatroom

Das Senden von Nachrichten und das Durchführen von Moderatorenaktionen sind einige der wichtigsten Formen der Interaktion. Hier erfahren Sie, wie Sie mithilfe verschiedener Chat-Anfrage Objekte allgemeine Aktionen in Chatterbox durchführen. Dazu gehören das Senden und Löschen von Nachrichten sowie das Trennen der Verbindung anderer Benutzer.

Alle Aktionen in einem Chatroom folgen einem gemeinsamen Muster: Für jede Aktion, die in einem Chatroom durchgeführt wird, gibt es ein entsprechendes Anforderungsobjekt. Für jede Anforderung gibt es ein entsprechendes Antwortobjekt, das bei Bestätigung der Anforderung empfangen wird.

Solange den Benutzern beim Erstellen eines Chat-Tokens die richtigen Fähigkeiten erteilt werden, können sie die entsprechenden Aktionen erfolgreich durchführen. Mithilfe der Anforderungsobjekte lässt sich feststellen, welche Anforderungen in einem Chatroom durchgeführt werden können.

Nachfolgend erklären wir das Senden einer Nachricht und das Löschen einer Nachricht.

Senden einer Nachricht

Die Klasse `SendMessageRequest` ermöglicht das Senden von Nachrichten in einem Chatroom. Hier ändern Sie Ihre `App`, um mit der Komponente, die Sie unter Erstellen einer Nachrichteneingabe (in Teil 1 dieses Tutorials) erstellt haben, eine Nachrichtenanforderung zu senden.

Definieren Sie zunächst eine neue boolesche Eigenschaft namens `isSending` mit dem Hook `useState`. Schalten Sie mithilfe dieser neuen Eigenschaft den deaktivierten Status des `button`-

Elements um. Verwenden Sie dabei die Konstante `isSendDisabled`. Löschen Sie im Event-Handler für `SendButton` den Wert für `messageToSend` und stellen Sie `isSending` auf „true“ ein.

Da über diese Schaltfläche ein API-Aufruf getätigt wird, verhindert das Hinzufügen des booleschen Werts *`isSending`*, dass mehrere API-Aufrufe gleichzeitig ausgeführt werden. Dazu werden Benutzerinteraktionen für *`SendButton`* deaktiviert, bis die Anforderung abgeschlossen ist.

Hinweis: Das Senden von Nachrichten funktioniert nur, wenn Sie Ihrem Token-Anbieter die `SEND_MESSAGE`-Funktion hinzugefügt haben, wie oben unter Erkennen von Nachrichten, die vom aktuellen Benutzer gesendet wurden, beschrieben.

TypeScript/JavaScript:

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

// ...

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

// ...

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

// ...

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

// ...
```

Bereiten Sie die Anforderung vor, indem Sie eine neue `SendMessageRequest`-Instance erstellen und den Nachrichteninhalt an den Konstruktor übergeben. Rufen Sie nach dem Festlegen des Status von `isSending` und `messageToSend` die Methode `sendMessage` auf, die die Anforderung an den Chatroom sendet. Löschen Sie abschließend das Flag `isSending`, sobald Sie eine Bestätigung oder Ablehnung der Anforderung erhalten haben.

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);
  }
};

// ...
```

Führen Sie Chatterbox aus: Senden Sie eine Nachricht, indem Sie eine Nachricht mit `MessageBar` verfassen und auf `SendButton` tippen. Die gesendete Nachricht sollte in der `MessageList`, die Sie zuvor erstellt haben, gerendert werden.

Löschen einer Nachricht

Um eine Nachricht aus einem Chatroom zu löschen, benötigen Sie die entsprechende Fähigkeit. Fähigkeiten werden bei der Initialisierung des Chat-Tokens gewährt, das Sie bei der Authentifizierung in einem Chatroom verwenden. Für die Zwecke dieses Abschnitts können Sie in der `ServerApp` aus Einrichten eines lokalen Authentifizierungs-/Autorisierungsservers (in Teil 1 dieses Tutorials) die Moderatorfähigkeiten festlegen. Dies geschieht in der App mithilfe des Objekts `tokenProvider`, das Sie unter Erstellen eines Token-Anbieters (ebenfalls in Teil 1) erstellt haben.

Hier ändern Sie Ihre `Message`, indem Sie eine Funktion zum Löschen der Nachricht hinzufügen.

Öffnen Sie zunächst `App.tsx` und fügen Sie die Fähigkeit `DELETE_MESSAGE` hinzu. (`capabilities` ist der zweite Parameter der Funktion `tokenProvider`.)

Hinweis: Auf diese Weise informiert die `ServerApp` die IVS-Chat-APIs darüber, dass der Benutzer, der mit dem resultierenden Chat-Token verknüpft wird, Nachrichten in einem Chatroom löschen kann.

In einer realen Umgebung wird die Backend-Logik zur Verwaltung der Benutzerfähigkeiten in der Infrastruktur Ihrer Server-App wahrscheinlich komplexer sein.

TypeScript/JavaScript:

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

// ...

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

// ...
```

In den nächsten Schritten aktualisieren Sie Ihre `Message`, um eine Schaltfläche zum Löschen anzuzeigen.

Definieren Sie eine neue Funktion namens on`Delete`, die eine Zeichenfolge als einen möglichen Parameter akzeptiert und `Promise` zurückgibt. Übergeben Sie als Zeichenfolgenparameter die ID der Komponentennachricht.

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',
  },
});
```

Als Nächstes aktualisieren Sie die `renderItem`, damit die neuesten Änderungen an Ihrer `FlatList`-Komponente wiedergegeben werden.

Definieren Sie in App eine Funktion namens `handleDeleteMessage` und übergeben Sie sie an die Eigenschaft `MessageList onDelete`:

TypeScript

```
// App.tsx

// ...

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

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

// ...
```

JavaScript

```
// App.jsx

// ...

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

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

```
// ...
```

Bereiten Sie eine Anforderung vor, indem Sie eine neue Instance von `DeleteMessageRequest` erstellen und die entsprechende Nachrichten-ID an den Konstruktorparameter übergeben. Rufen Sie dann den Befehl `deleteMessage` auf, der die oben vorbereitete Anforderung akzeptiert:

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);
};

// ...
```

Als Nächstes aktualisieren Sie den Status `messages`, damit eine neue Liste von Nachrichten – ohne die gerade gelöschte Nachricht – angezeigt wird.

Warten Sie im Hook `useEffect` auf das Ereignis `messageDelete` und aktualisieren Sie das Status-Array `messages`, indem Sie die Nachricht mit einer zum Parameter `message` passenden ID löschen.

Hinweis: Das Ereignis `messageDelete` kann ausgelöst werden, wenn Nachrichten vom aktuellen Benutzer oder von anderen Benutzern im Chatroom gelöscht wurden. Wenn Sie es im

Ereignishandler verarbeiten (statt neben der Anforderung `deleteMessage`), können Sie das Löschen von Nachrichten vereinheitlichen.

TypeScript/JavaScript:

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

Sie können jetzt Benutzer aus einem Chatroom in Ihrer Chat-App löschen.

Nächste Schritte

Versuchen Sie als Experiment, andere Aktionen in einem Chatroom zu implementieren, z. B. das Trennen der Verbindung eines anderen Benutzers.

Client-Messaging-SDK für IVS Chat: Bewährte Methoden zu React und React Native

In diesem Dokument werden die wichtigsten Methoden zur Verwendung des Amazon-IVS-Chat-Messaging-SDK für React und React Native beschrieben. Diese Informationen sollten es Ihnen ermöglichen, typische Chat-Funktionen in einer React-Anwendung zu erstellen, und Ihnen den Hintergrund geben, den Sie benötigen, um tiefer in die fortgeschritteneren Teile des IVS-Chat-Messaging-SDKs einzutauchen.

Erstellen eines ChatRoom-Initializer-Hooks

Die `ChatRoom`-Klasse enthält grundlegende Chat-Methoden und Listener zur Verwaltung des Verbindungsstatus und zum Abhören von Ereignissen wie empfangenen und gelöschten Nachrichten. Hier zeigen wir, wie man Chat-Instances richtig in einem Hook speichert.

Implementierung

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 };
};
```

Hinweis: Wir verwenden nicht die `dispatch`-Methode aus dem `setState`-Hook, da Sie die Konfigurationsparameter nicht im laufenden Betrieb aktualisieren können. Das SDK erstellt einmalig eine Instance und es ist nicht möglich, den Token-Anbieter zu aktualisieren.

Wichtig: Verwenden Sie den `ChatRoom`-Initialisierer-Hook einmal, um eine neue Chatroom-Instance zu initialisieren.

Beispiel

TypeScript/JavaScript:

```
// ...

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

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

  // ...
};

// ...
```

Auf den Verbindungsstatus warten

Optional können Sie Updates zum Verbindungsstatus in Ihrem Chatroom-Hook abonnieren.

Implementierung

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 };
};
```

Anbieter von ChatRoom-Instances

Um den Hook in anderen Komponenten zu verwenden (um Prop-Drilling zu vermeiden), können Sie mit React `context` einen Chatroom-Anbieter erstellen.

Implementierung

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;
```

Beispiel

Nach der Erstellung von `ChatRoomProvider` können Sie Ihre Instance mit `useChatRoomContext` verwenden.

Wichtig: Platzieren Sie den Anbieter nur dann in der Root-Ebene, wenn Sie Zugriff auf die `context`-Verbindung zwischen dem Chat-Bildschirm und den anderen Komponenten in der Mitte benötigen, um unnötiges erneutes Rendern zu vermeiden, wenn Sie nach Verbindungen suchen. Andernfalls stellen Sie den Anbieter so nahe an den Chatbildschirm wie möglich.

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

// ...
```

Erstellen eines Nachrichten-Listeners

Um über alle eingehenden Nachrichten auf dem Laufenden zu bleiben, sollten Sie die `message`- und `deleteMessage`-Ereignisse abonnieren. Hier ist ein Code, der Chat-Nachrichten für Ihre Komponenten bereitstellt.

Wichtig: Aus Leistungs-Gründen trennen wir `ChatMessageContext` von `ChatRoomProvider`, da wir viele Renderings erhalten können, wenn der Chat-Nachrichten-Listener den Status seiner Nachricht aktualisiert. Denken Sie daran, `ChatMessageContext` in den Komponenten zu verwenden, in denen Sie `ChatMessageProvider` verwenden werden.

Implementierung

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>;
};
```

Beispiel in React

Wichtig: Denken Sie daran, Ihren Nachrichtencontainer mit `ChatMessagesProvider` zu verpacken. Die `Message`-Zeile ist eine Beispielkomponente, die den Inhalt einer Nachricht anzeigt.

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>
  );
};
```

Beispiel in React Native

Standardmäßig enthält `ChatMessage` `id`, das automatisch als React Schlüssel in `FlatList` für jede Zeile verwendet wird; Sie müssen daher nicht an `keyExtractor` übergeben.

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} />;
};
```

Mehrere Chatroom-Instances in einer App

Wenn Sie in Ihrer App mehrere Chatrooms gleichzeitig verwenden, schlagen wir vor, jeden Anbieter für jeden Chat zu erstellen und ihn im Chat-Anbieter zu nutzen. In diesem Beispiel erstellen wir einen Hilfe-Bot und einen Kunden-Hilfe-Chat. Wir schaffen einen Anbieter für beide.

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>;
};
```

Beispiel in React

Jetzt können Sie verschiedene Chat-Anbieter verwenden, die denselben `ChatRoomProvider` verwenden. Später können Sie denselben `useChatRoomContext` in jedem Bildschirm/jeder Ansicht wiederverwenden.

TypeScript/JavaScript:

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

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

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

Beispiel in React Native

TypeScript/JavaScript:

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

const App = () => {
  return (
    <Stack.Navigator>
      <Stack.Screen name="SupportChat">
        <SupportChatProvider>
          <SupportChatScreen />
        </SupportChatProvider>
      </Stack.Screen>
      <Stack.Screen name="SalesChat">
        <SalesChatProvider>
          <SalesChatScreen />
        </SalesChatProvider>
      </Stack.Screen>
    </Stack.Navigator>
  );
};
```

TypeScript/JavaScript:

```
// SupportChatScreen.tsx / SupportChatScreen.jsx

// ...

const SupportChatScreen = () => {
  const room = useChatRoomContext();

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

  return (
    <>
      <Button title="Connect" onPress={handleConnect} />
```

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

// SalesChatScreen.tsx / SalesChatScreen.jsx

// ...

const SalesChatScreen = () => {
  const room = useChatRoomContext();

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

  return (
    <>
      <Button title="Connect" onPress={handleConnect} />
      <MessageListContainer />
    </>
  );
};
```

Sicherheit von Amazon IVS Chat

Cloud-Sicherheit hat bei AWS höchste Priorität. Als AWS-Kunde profitieren Sie von einer Rechenzentrums- und Netzwerkarchitektur, die eingerichtet wurde, um die Anforderungen der anspruchsvollsten Organisationen in puncto Sicherheit zu erfüllen.

Sicherheit ist eine übergreifende Verantwortlichkeit zwischen AWS und Ihnen. Das Modell der geteilten Verantwortung beschreibt dies als Sicherheit der Cloud und Sicherheit in der Cloud:

- Sicherheit der Cloud – AWS ist dafür verantwortlich, die Infrastruktur zu schützen, mit der AWS-Services in der AWS Cloud ausgeführt werden. AWS stellt Ihnen außerdem Services bereit, die Sie sicher nutzen können. Auditoren von Drittanbietern testen und überprüfen die Effektivität unserer Sicherheitsmaßnahmen im Rahmen der AWS-Compliance-Programme regelmäßig.
- Sicherheit in der Cloud – Ihr Verantwortungsumfang wird durch den AWS-Service bestimmt, den Sie verwenden. In Ihre Verantwortung fallen außerdem weitere Faktoren, wie z. B. die Vertraulichkeit der Daten, die Anforderungen Ihrer Organisation sowie geltende Gesetze und Vorschriften.

Diese Dokumentation hilft Ihnen zu verstehen, wie Sie das Modell der geteilten Verantwortung bei der Verwendung von Amazon IVS Chat anwenden. In den folgenden Themen erfahren Sie, wie Sie Amazon IVS Chat so konfigurieren, dass Ihre Sicherheits- und Compliance-Ziele erreicht werden.

Themen

IVS-Chat-Datenschutz

Für Daten, die an Amazon Interactive Video Service (IVS) Chat gesendet werden, sind folgende Datenschutzmaßnahmen vorhanden:

- Amazon-IVS-Chat-Datenverkehr verwendet WSS, um Daten während der Übertragung zu sichern.
- Amazon-IVS-Chat-Token werden mit vom Kunden verwalteten KMS-Schlüsseln verschlüsselt.

Amazon IVS Chat verlangt nicht, dass Sie irgendwelche Kundendaten (Endbenutzerdaten) bereitstellen. Es gibt keine Felder in Chatrooms, Eingaben oder Eingabesicherheitsgruppen, in denen erwartet wird, dass Sie Kundendaten (Endbenutzerdaten) bereitstellen.

Geben Sie keine sensiblen Informationen wie Kontonummern Ihrer Kunden (Endbenutzer) in Freiformfelder wie z. B. ein Namensfeld ein. Dies gilt auch, wenn Sie mit der Amazon-IVS-Konsole oder API, AWS-CLI oder AWS-SDKs arbeiten. Alle Daten, die Sie in Amazon IVS Chat eingeben, können in Diagnoseprotokolle aufgenommen werden.

Streams sind nicht Ende-zu-Ende verschlüsselt; ein Stream kann unverschlüsselt intern im IVS-Netzwerk zur Verarbeitung übertragen werden.

Identitäts- und Zugriffsverwaltung in IVS Chat

AWS Identity and Access Management (IAM) ist ein AWS-Service, mit dem Kontoadministratoren den Zugriff auf AWS-Ressourcen sicher steuern können. Siehe Identitäts- und Zugriffsverwaltung in IVS im Benutzerhandbuch für IVS-Streaming mit niedriger Latenz.

Zielgruppe

Wie Sie IAM verwenden, hängt von der Arbeit ab, die Sie in Amazon IVS ausführen. Siehe Zielgruppe Im Benutzerhandbuch für IVS-Streaming-mit niedriger Latenz.

Wie Amazon IVS mit IAM funktioniert

Bevor Sie Amazon IVS-API-Anfragen stellen können, müssen Sie eine oder mehrere IAM-Identitäten (Benutzer, Gruppen und Rollen) und IAM-Richtlinien erstellen und dann den Identitäten Richtlinien zuordnen. Es dauert bis zu einigen Minuten, bis die Berechtigungen weitergegeben werden. Bis dahin werden API-Anforderungen abgelehnt.

Eine Übersicht darüber, wie Amazon IVS mit IAM funktioniert, finden Sie unter AWS-Services, die mit IAM arbeiten im IAM-Benutzerhandbuch.

Identitäten

Sie können IAM-Identitäten erstellen, um die Authentifizierung für Personen und Prozesse in Ihrem AWS-Konto bereitzustellen. IAM-Gruppen sind Sammlungen von IAM-Benutzern, die Sie als eine Einheit verwalten können. Siehe Identitäten (Benutzer, Gruppen und Rollen) im IAM-Benutzerhandbuch.

Richtlinien

Richtlinien sind JSON-Berechtigungsrichtliniendokumente, die aus Elementen bestehen. Siehe Richtlinien im Benutzerhandbuch für IVS-Streaming-mit niedriger Latenz.

Amazon IVS Chat unterstützt drei Elemente:

- Aktionen – Richtlinienaktionen für Amazon IVS Chat verwenden das `ivschat`-Präfix vor der Aktion. Um beispielsweise jemandem die Berechtigung zum Erstellen eines Amazon-IVS-Chat-Channels mit der Amazon IVS Chat `CreateRoom`-API-Methode zu erteilen, nehmen Sie die `ivschat:CreateRoom`-Aktion in die Richtlinie für diese Person auf. Richtlinienanweisungen müssen entweder ein – `Action`oder ein `NotAction`-Element enthalten.
- Ressourcen – Die Amazon-IVS-Chat-Raumressource hat folgendes ARN-Format:

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

  Um z. B. Raum `VgNkEJgOVX9N` in Ihrer Anweisung anzugeben, verwenden Sie diese ARN:

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

  Einige Amazon-IVS-Chat-Aktionen, wie z. B. die zum Erstellen von Ressourcen, können nicht für eine bestimmte Ressource durchgeführt werden. In diesen Fällen müssen Sie den Platzhalter (*) verwenden. (*):

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

- Bedingungen – Amazon IVS Chat unterstützt einige globale Bedingungsschlüssel: `aws:RequestTag`, `aws:TagKeys`, und `aws:ResourceTag`.

Sie können Variablen als Platzhalter in einer Richtlinie verwenden. Sie können beispielsweise einem IAM-Benutzer nur dann die Berechtigung zum Zugriff auf eine Ressource erteilen, wenn diese mit dem IAM-Benutzernamen des Benutzers gekennzeichnet ist. Siehe Variablen und Tags im IAM-Benutzerhandbuch.

Amazon IVS stellt von AWS verwaltete Richtlinien bereit, mit denen Identitäten ein Satz vorkonfigurierter Berechtigungen gewährt werden können (nur Lesezugriff oder Vollzugriff). Sie können anstelle der unten angezeigten identitätsbasierten Richtlinien auch verwaltete Richtlinien verwenden. Einzelheiten finden Sie unter Verwaltete Richtlinien für Amazon IVS Chat.

Autorisierung auf der Basis von Amazon IVS Tags

Sie können Tags an Amazon IVS-Chat-Ressourcen anhängen oder Tags in einer Anforderung an Amazon IVS Chat übergeben. Um den Zugriff auf Basis von Tags zu steuern, geben Sie Tag-Informationen im Bedingungselement einer Richtlinie mithilfe der Bedingungsschlüssel `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` oder `aws:TagKeys` an. Weitere Informationen zum Markieren von Amazon-IVS-Chat-Ressourcen finden Sie unter „Markieren“ in der IVS-Chat-API-Referenz.

Rollen

Siehe IAM-Rollen und Temporäre Anmeldeinformationen im IAM-Benutzerhandbuch.

Eine IAM-Rolle ist eine Entität in Ihrem AWS-Konto mit spezifischen Berechtigungen.

Amazon IVS unterstützt die Verwendung temporärer Sicherheitsanmeldeinformationen. Sie können temporäre Anmeldeinformationen verwenden, um sich mit dem Verbund anzumelden, eine IAM-Rolle zu übernehmen oder eine kontoübergreifende Rolle zu übernehmen. Sie erhalten temporäre Sicherheitsanmeldeinformationen durch Aufrufen von AWS Security Token Service API-Operationen wie `AssumeRole` oder `GetFederationToken`.

Privilegierter und unprivilegierter Zugriff

API-Ressourcen haben privilegierten Zugriff. Unprivilegierter Wiedergabezugriff kann über private Kanäle eingerichtet werden; siehe Einrichten von privaten IVS-Kanälen.

Best Practices für Policen

Siehe IAM Best Practices im IAM-Benutzerhandbuch.

Identitätsbasierte Richtlinien sind sehr leistungsfähig. Sie bestimmen, ob jemand Amazon IVS-Ressourcen in Ihrem Konto erstellen, darauf zugreifen oder sie löschen kann. Dies kann zusätzliche Kosten für Ihr AWS-Konto verursachen. Befolgen Sie diese Empfehlungen:

- Gewähren Sie die geringsten Rechte – Wenn Sie benutzerdefinierte Richtlinien erstellen, gewähren Sie nur die Berechtigungen, die zum Ausführen einer Aufgabe erforderlich sind. Beginnen Sie mit einem minimalen Satz an Berechtigungen und gewähren Sie bei Bedarf mehr Berechtigungen. Dies ist sicherer, als mit zu laxen Berechtigungen zu beginnen und dann zu versuchen, diese später zu verschärfen. Insbesondere reservieren Sie `ivschat:*` für Admin-Zugriff. Verwenden Sie es nicht in Anwendungen.
- Aktivieren von MFA für sensible Vorgänge – Fordern Sie von IAM-Benutzern die Verwendung von Multi-Factor Authentication (MFA), um zusätzliche Sicherheit beim Zugriff auf sensible Ressourcen oder API-Operationen zu bieten.
- Verwenden von Richtlinienbedingungen für zusätzliche Sicherheit – Definieren Sie, soweit dies möglich ist, die Bedingungen, unter denen Ihre identitätsbasierten Richtlinien den Zugriff auf eine Ressource erlauben. Sie können z. B. Bedingungen schreiben, um einen Bereich zulässiger IP-Adressen festzulegen, von denen eine Anfrage kommen muss. Sie können auch Bedingungen schreiben, um Anfragen nur innerhalb eines bestimmten Datums oder Zeitbereichs zuzulassen oder um die Verwendung von SSL oder MFA zu verlangen.

Beispiele für identitätsbasierte Richtlinien

Verwenden Sie die Amazon IVS-Konsole

Um auf die Amazon-IVS-Konsole zuzugreifen, müssen Sie über ein Minimum an Berechtigungen verfügen, die es Ihnen ermöglichen, Details zu den Amazon-IVS-Chat-Ressourcen in Ihrem AWS-Konto aufzulisten und anzuzeigen. Wenn Sie eine identitätsbasierte Richtlinie erstellen, die restriktiver ist als die erforderlichen Mindestberechtigungen, funktioniert die Konsole für Identitäten mit dieser Richtlinie nicht wie vorgesehen. Um den Zugriff auf die Amazon IVS-Konsole sicherzustellen, fügen Sie den Identitäten die folgende Richtlinie an (siehe IAM-Berechtigungen hinzufügen und entfernen im IAM-Benutzerhandbuch).

Die Teile der folgenden Richtlinie bieten Zugriff auf:

- Alle API-Vorgänge von Amazon IVS Chat
- Ihre Service Quotas für Amazon IVS Chat

- Auflisten von Lambdas und Hinzufügen von Berechtigungen für das gewählte Lambda für Amazon-IVS-Chat-Moderation
- Amazon CloudWatch, um Metriken für Ihre Chat-Sitzung zu erhalten

JSON

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

Ressourcenbasierte Richtlinie für Amazon IVS Chat

Sie sollten dem Amazon-IVS-Chat-Service die Berechtigung erteilen, zum Überprüfen von Nachrichten Ihre Lambda-Ressource aufzurufen. Befolgen Sie hierzu die Anweisungen unter Verwenden von ressourcenbasierten Richtlinien für AWS Lambda (im AWS-Lambda-Entwicklerhandbuch) und füllen Sie die Felder wie unten angegeben aus.

Um den Zugriff auf Ihre Lambda-Ressource zu steuern, können Sie Bedingungen verwenden, die auf Folgendem basieren:

- `SourceArn` – Unsere Beispielrichtlinie verwendet einen Platzhalter (*), damit alle Räume in Ihrem Konto Lambda aufrufen können. Optional können Sie in Ihrem Konto einen Raum angeben, damit nur dieser Raum Lambda aufrufen kann.
- `SourceAccount` – Unter der folgenden Beispielrichtlinie lautet die AWS-Konto-ID `123456789012`.

JSON

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

```
}
```

Fehlerbehebung

Informationen zur Diagnose und Behebung häufiger Probleme, die bei der Arbeit mit Amazon IVS Chat und IAM auftreten können, finden Sie unter Fehlerbehebung im Benutzerhandbuch für IVS-Streaming mit niedriger Latenz.

Verwaltete Richtlinien für IVS Chat

Bei einer von AWS verwalteten Richtlinie handelt es sich um eine eigenständige Richtlinie, die von AWS erstellt und verwaltet wird. Siehe Verwaltete Richtlinien für Amazon IVS im Benutzerhandbuch für IVS-Streaming-mit niedriger Latenz.

Verwenden von serviceverknüpften Rollen für IVS Chat

Amazon IVS verwendet serviceverknüpfte AWS-IAM-Rollen. Weitere Informationen finden Sie unter Verwenden von serviceverknüpften Rollen für Amazon IVS im Benutzerhandbuch für IVS-Streaming mit niedriger Latenz.

IVS-Chat-Protokollierung und -Überwachung

Verwenden Sie Amazon CloudTrail, um Leistung und/oder Vorgänge zu protokollieren. Weitere Informationen finden Sie unter Protokollieren von Amazon-IVS-API-Aufrufen mit AWS CloudTrail im Benutzerhandbuch für IVS-Streaming mit niedriger Latenz.

IVS-Chat-Vorfallreaktion

Um Vorfälle zu erkennen oder zu warnen, können Sie den Status Ihres Streams über Amazon EventBridge Ereignisse überwachen. Siehe „Amazon EventBridge mit Amazon IVS verwenden:“ für Streaming mit niedriger Latenz und für Streaming in Echtzeit.

Benutzen Sie das AWS-Servicestatus-Dashboard für Informationen zum allgemeinen Zustand von Amazon IVS (nach Regionen).

IVS-Chat-Resilienz

Die IVS-API verwendet die globale AWS-Infrastruktur und ist um AWS-Regionen und Availability Zones herum aufgebaut. Siehe IVS-Resilienz im Benutzerhandbuch für IVS-Streaming mit niedriger Latenz.

Sicherheit der IVS-Chat-Infrastruktur

Als verwalteter Service ist Amazon IVS durch die globalen Verfahren zur Gewährleistung der Netzwerksicherheit von AWS geschützt. Diese sind in Bewährte Praktiken für Sicherheit, Identität und Compliance beschrieben.

API Calls

Sie verwenden durch AWS veröffentlichte API-Aufrufe, um über das Netzwerk auf Amazon IVS zuzugreifen. Siehe API-Aufrufe unter Infrastruktursicherheit im Benutzerhandbuch für IVS-Streaming mit niedriger Latenz.

Amazon IVS Chat

Die Aufnahme und Zustellung von Amazon-IVS-Chat-Nachrichten erfolgt über verschlüsselte WSS-Verbindungen an unserem Edge. Die Amazon-IVS-Messaging-API verwendet verschlüsselte HTTPS-Verbindungen. Wie beim Videostreaming und der Wiedergabe ist TLS Version 1.2 oder höher erforderlich, und Messaging-Daten können intern unverschlüsselt zur Verarbeitung übertragen werden.

IVS-Chat-Service-Quotas

Im Folgenden finden Sie Service Quotas und Limits für Amazon-Interactive-Video-Service-(IVS)-Chat-Endpunkte, -Ressourcen und andere Vorgänge. Service Quotas (auch als Limits bezeichnet) sind die maximale Anzahl von Service-Ressourcen oder Vorgängen für Ihr AWS-Konto. Das heißt, diese Grenzwerte gelten je AWS Konto, sofern in der Tabelle nichts anderes angegeben ist. Lesen Sie auch den Abschnitt AWS Service Quotas.

Sie verwenden einen Endpunkt, um eine Verbindung zu einem AWS-Service programmgesteuert herzustellen Lesen Sie auch den Abschnitt AWS-Service-Endpunkte.

Alle Kontingente werden pro Region erzwungen.

Erhöhte Service Quotas

Für einstellbare Kontingente können Sie eine Ratenerhöhung über die AWS-Konsole anfragen. Verwenden Sie die Konsole, um Informationen über Service Quotas anzuzeigen.

Kontingente für API-Anrufraten sind nicht anpassbar.

API-Aufrufratenquoten

Vorgangstyp	Operation	Standard
Messaging	`DeleteMessage`	100 TPS
Messaging	`DisconnectUser`	100 TPS
Messaging	`SendEvent`	100 TPS
Chat-Token	`CreateChatToken`	200 TPS
Protokollierungskonfiguration	`CreateLoggingConfiguration`	3 TPS
Protokollierungskonfiguration	`DeleteLoggingConfiguration`	3 TPS
Protokollierungskonfiguration	`GetLoggingConfiguration`	3 TPS
Protokollierungskonfiguration	`ListLoggingConfigurations`	3 TPS

Vorgangstyp	Operation	Standard
Protokollierungskonfiguration	`UpdateLoggingConfiguration`	3 TPS
Raum	`CreateRoom`	5 TPS
Raum	`DeleteRoom`	5 TPS
Raum	`GetRoom`	5 TPS
Raum	`ListRooms`	5 TPS
Raum	`UpdateRoom`	5 TPS
Tags	`ListTagsForResource`	10 TPS
Tags	`TagResource`	10 TPS
Tags	`UntagResource`	10 TPS

Andere Kontingente

Ressource oder Feature	Standard	Anpassbar	Beschreibung
Gleichzeitige Chat-Verbindungen	50 000	Ja	Maximale Anzahl gleichzeitiger Chat-Verbindungen pro Konto in allen Ihren Räumen in einer AWS-Region.
Protokollierungskonfigurationen	10	Ja	Die Höchstzahl von Protokollierungskonfigurationen, die pro Konto in der aktuellen erstellt werden können AWS-Region.
Leerlaufzeitlimit für den Nachrichtenrezension-Handler	200	Nein	Leerlaufzeitlimit in Millisekunden für alle Ihre Nachrichtenrezension-Handler in der

Ressource oder Feature	Standard	Anpassbar	Beschreibung
			aktuellen AWS-Region. Wenn dieses überschritten wird, wird die Nachricht abhängig vom Wert des Felds `fallbackResult` , das Sie für den Nachrichtenrezension-Handler konfiguriert haben, zugelassen oder abgelehnt.
Rate der DeleteMessage-Anforderungen in allen Ihren Räumen	100	Ja	Maximale Anzahl von DeleteMessage-Anforderungen, die pro Sekunde in allen Ihren Räumen gestellt werden können. Die Anforderungen können entweder von der Amazon-IVS-Chat-API oder der Amazon-IVS-Chat-Messaging-API (WebSocket) stammen.
Rate der DisconnectUser-Anforderungen in allen Ihren Räumen	100	Ja	Maximale Anzahl von DisconnectUser-Anforderungen, die pro Sekunde in allen Ihren Räumen gestellt werden können. Die Anforderungen können entweder von der Amazon-IVS-Chat-API oder der Amazon-IVS-Chat-Messaging-API (WebSocket) stammen.
Rate der Messaging-Anforderungen pro Verbindung	10	Nein	Maximale Anzahl von Messaging-Anforderungen pro Sekunde, die eine Chat-Verbindung herstellen kann.

Ressource oder Feature	Standard	Anpassbar	Beschreibung
Rate der SendMessage-Anforderungen in allen Ihren Räumen	1000	Ja	Maximale Anzahl von SendMessage-Anforderungen, die pro Sekunde in allen Ihren Räumen gestellt werden können. Diese Anforderungen stammen von der Amazon-IVS-Chat-Messaging-API (WebSocket).
Rate der SendMessage-Anforderungen pro Raum	100	Nein (aber über die API konfiguri erbar)	Maximale Anzahl von SendMessage-Anforderungen, die pro Sekunde in jedem einzelnen Ihrer Räume gestellt werden können. Dies ist über das Feld `maximumMessageRatePerSecond` von CreateRoom und UpdateRoom konfiguri erbar. Diese Anforderungen stammen von der Amazon-IVS-Chat-Messaging-API (WebSocket).
Räume	50 000	Ja	Maximale Anzahl von Chat-Räumen pro Konto und AWS-Region.

Integration von Service Quotas mit CloudWatch-Nutzungsmetriken

Verwenden von CloudWatch zum proaktiven Verwalten Ihrer Servicekontingente über CloudWatch-Nutzungsmetriken. Sie können diese Metriken verwenden, um Ihre aktuelle Servicenutzung auf CloudWatch-Diagrammen und -Dashboards zu visualisieren. Die Nutzungsmetriken von Amazon IVS Chat entsprechen den Service Quotas von Amazon IVS.

Sie können eine CloudWatch-Metrik-Mathematikfunktion verwenden, um die Service Quotas für diese Ressourcen in Ihren Diagrammen anzuzeigen. Sie können auch Alarme konfigurieren, mit denen Sie benachrichtigt werden, wenn sich Ihre Nutzung einer Service Quota nähert.

So greifen Sie auf Nutzungsmetriken zu:

1. Öffnen Sie die Service-Quotas-Konsole unter https://console.aws.amazon.com/servicequotas/
2. Wählen Sie im Navigationsbereich AWS-Services aus.
3. Suchen Sie in der AWS-Service-Liste nach Amazon Interactive Video Service Chat.
4. Wählen Sie in der Liste Service Quotas die gewünschte Service Quota aus. Es wird eine neue Seite mit Informationen über die Service Quota/die Metrik geöffnet.

Alternativ können Sie diese Metriken über die CloudWatch Konsole aufrufen. Wählen Sie unter AWS Namespaces die Option Verwendung. Wählen Sie dann in der Liste Service die Option IVS Chat aus. (Siehe Überwachen von Amazon IVS Chat.)

Im AWS/Nutzung-Namespace stellt Amazon IVS Chat die folgende Metrik bereit:

Metrikname	Beschreibung
`ResourceCount`	Die Anzahl der angegebenen Ressourcen, die in Ihrem Konto ausgeführt werden. Die Ressourcen werden durch die Dimensionen definiert, die der Metrik zugeordnet sind. Gültige Statistik: Maximalanzahl (die maximale Anzahl der Ressourcen, die während des 1-Minuten-Zeitraums verwendet werden).

Die folgenden Dimensionen werden verwendet, um die Nutzungsmetriken zu verfeinern:

Dimension	Beschreibung
Service	Der Name des AWS-Service, der die Ressource enthält. Zulässigee Wert: `IVS Chat`.
Klasse	Die Klasse der nachverfolgten Ressource. Zulässigee Wert: `None`.
Typ	Der Typ der nachverfolgten Ressource. Zulässigee Wert: `Resource`.

Dimension	Beschreibung
Ressource	Der Name der AWS-Ressource. Zulässiger Wert: `ConcurrentChatConnections` . Die Nutzungsmetrik „ConcurrentChatConnections“ ist eine Kopie derjenigen im AWS/IVSChat-Namespace (mit der None-Dimension), wie in Überwachen von Amazon IVS Chat beschrieben.

Erstellen eines CloudWatch Alarms für Nutzungsmetriken

So erstellen Sie einen CloudWatch-Alarm basierend auf einer Amazon-IVS-Chat-Nutzungsmetrik:

1. Wählen Sie in der Konsole „Service Quotas“ das gewünschte Dienstkontingent aus, wie oben beschrieben. Derzeit können Alarme nur für ConcurrentChatConnections erstellt werden.
2. Wählen Sie im Abschnitt Amazon CloudWatch-Alarme die Option Erstellen.
3. Wählen Sie bei Alarmschwellenwert den Prozentsatz des angewendeten Kontingentwerts aus, den Sie als Alarmwert festlegen möchten.
4. Geben Sie für Alarmname einen Namen für den Alarm ein.
5. Wählen Sie Erstellen.

Fehlerbehebung bei IVS Chat

In diesem Dokument werden bewährte Methoden und Tipps zur Fehlerbehebung für Amazon Interactive Video Service (IVS) Chat beschrieben. Verhaltensweisen im Zusammenhang mit IVS Chat unterscheiden sich oft von Verhaltensweisen im Zusammenhang mit IVS-Videos. Weitere Informationen finden Sie unter Erste Schritte mit Amazon IVS Chat.

Themen:

Warum wurden die IVS-Chat-Verbindungen nicht unterbrochen, als der Raum gelöscht wurde?

Wenn eine Chatroom-Ressource gelöscht wird und der Raum aktiv genutzt wird, werden die Chat-Clients, die mit dem Raum verbunden sind, nicht automatisch getrennt. Die Verbindung wird unterbrochen, falls/wenn die Chat-Anwendung das Chat-Token aktualisiert. Alternativ muss die Verbindung aller Benutzer manuell getrennt werden, um alle Benutzer aus dem Chatroom zu entfernen.

IVS-Glossar

Weitere Informationen finden Sie im AWS-Glossar. In der folgenden Tabelle steht LL für IVS-Streaming mit niedriger Latenz. RT und IVS-Echtzeit-Streaming.

Begriff	Beschreibung	LL	RT	Chat
AAC	Erweiterte Audio-Kodierung. AAC ist ein Audio-Kodierungsstandard für verlustbehaftete digitale Audiokomprimierung. Als Nachfolger des MP3-Formats konzipiert, erreicht AAC bei gleicher Bitrate im Allgemeinen eine höhere Klangqualität als MP3. AAC wurde von ISO und IEC als Teil der Spezifikationen MPEG-2 und MPEG-4 standardi siert.	✓	✓	
Streaming mit adaptiver Bitrate	Beim Streaming mit adaptiver Bitrate (ABR) kann der IVS-Player auf eine niedrigere Bitrate umschalten, wenn die Verbindungsqualität beeinträchtigt ist, und auf eine höhere Bitrate zurückschalten, wenn sich die Verbindungsqualität verbessert.	✓		
Adaptives Streaming	Weitere Informationen finden Sie unter Mehrschic htige Kodierung mit Simulcast.		✓	
Administratorbenut zer	Ein AWS-Benutzer mit Administratorzugriff auf Ressourcen und Services, die in einem AWS-Konto verfügbar sind. Weitere Informationen finden Sie unter Terminologie im Benutzerhandbuch zur AWS-Einrichtung.	✓	✓	✓
ARN	Amazon-Ressourcenname, eine eindeutige Kennung für eine AWS-Ressource. Spezifisc he ARN-Formate hängen vom Ressourcentyp ab. Informationen zu den von IVS-Ressourcen	✓	✓	✓

Begriff	Beschreibung	LL	RT	Chat
	verwendeten ARN-Formaten finden Sie in der Service-Autorisierungsreferenz.			
Seitenverhältnis	Beschreibt das Verhältnis der Rahmenbreite zur Rahmenhöhe. Beispielsweise ist 16:9 das Seitenverhältnis, das der Full-HD- oder 1080p-Auflösung entspricht.	✓	✓	
Audio-Modus	Eine voreingestellte oder benutzerdefinierte Audiokonfiguration, die für verschiedene Arten von Benutzern mobiler Geräte und die von ihnen verwendeten Geräte optimiert ist. Weitere Informationen finden Sie unter IVS Broadcast SDK: Mobile Audiomodi (Echtzeit-Streaming).		✓	
AVC, H.264, MPEG-4 Teil 10	Erweiterte Video-Kodierung, auch als H.264 oder MPEG-4 Teil 10 bezeichnet, ein Videokomprimierungsstandard für verlustbehaftete digitale Videokomprimierung.	✓	✓	
Ersetzen des Hintergrunds	Eine Art Kamerafilter, der es Livestream-Erstellern ermöglicht, ihren Hintergrund zu ändern. Weitere Informationen finden Sie unter Ersetzen des Hintergrunds in IVS Broadcast SDK: Kamera-Filter von Drittanbietern (Echtzeit-Streaming).		✓	
Bitrate	Eine Streaming-Metrik für die Anzahl der pro Sekunde übertragenen oder empfangenen Bits.	✓	✓	
Broadcast, Sender	Andere Begriffe für Stream, Streamer.	✓		

Begriff	Beschreibung	LL	RT	Chat
Pufferung	Ein Zustand, der auftritt, wenn das Wiedergab egerät den Inhalt nicht herunterladen kann, bevor der Inhalt abgespielt werden soll. Pufferung kann sich auf verschiedene Weise äußern: Inhalte können zufällig anhalten und starten (auch Stottern genannt), Inhalte können für längere Zeit anhalten (auch Einfrieren genannt) oder der IVS-Player kann die Wiedergabe anhalten.	✓	✓	
Wiedergabeliste im Bytebereich	Eine differenziertere Wiedergabeliste als die standardmäßige HLS-Wiedergabeliste. Die standardmäßige HLS-Wiedergabeliste besteht aus 10-sekündigen Mediendateien. Bei einer Wiedergabeliste mit Byte-Bereich entspricht die Segmentdauer dem Keyframe-Intervall, das für den Stream konfiguriert wurde. Die Wiedergabeliste im Bytebereich ist nur für Übertragungen verfügbar, die automatisch in einem S3-Bucket aufgezeichnet wurden. Diese wird zusätzlich zur HLS-Wiedergabeliste erstellt. Weitere Informationen finden Sie unter Wiedergab elisten im Byte-Bereich in Automatische Aufzeichn ung in Amazon S3 (Streaming mit niedriger Latenz).	✓		

Begriff	Beschreibung	LL	RT	Chat
CBR	Konstante Bitrate, eine Ratensteuerungsmethode für Encoder, die während der gesamten Wiedergabe eines Videos auf eine einheitliche Bitrate warten, unabhängig davon, was während der Übertragung passiert. Tiefpunkte im Geschehen können aufgefüllt werden, um die gewünschte Bitrate zu erreichen, und Spitzen können quantisiert werden, indem die Qualität der Kodierung an die Ziel-Bitrate angepasst wird. Wir empfehlen nachdrücklich die Verwendung von CBR anstelle von VBR.	✓	✓	
CDN	Netzwerk für die Inhaltsbereitstellung oder Netzwerk für die Inhaltsübermittlung, eine geografisch verteilte Lösung, die die Bereitstellung von Inhalten wie Streaming-Videos optimiert, indem sie diese näher an den Standort der Benutzer bringt.	✓		
Kanal	Eine IVS-Ressource, die die Konfiguration für das Streaming speichert, einschließlich eines Aufnahmeservers, eines Stream-Schlüssels, einer Wiedergabe-URL und Aufzeichnungsoptionen. Streamer verwenden den Streamschlüssel, der einem Kanal zugeordnet ist, um eine Übertragung zu starten. Alle während einer Übertragung generierten Metriken und Ereignisse werden einer Kanalressource zugeordnet.	✓		
Kanaltyp	Legt die zulässige Auflösung und Bildfrequenz für den Kanal fest. Siehe Kanaltypen in der Referenz der API von IVS-Streaming mit niedriger Latenz.	✓		
Chat-Protokollierung	Eine erweiterte Option, die durch die Zuordnung einer Protokollierungskonfiguration zu einem Chatroom ermöglicht wird.			✓

Begriff	Beschreibung	LL	RT	Chat
Chatroom	Eine IVS Ressource, die die Konfiguration für eine Chatsitzung speichert, einschließlich optionale r Features wie Handler zur Nachrichtenüberprü fung und Chat-Protokollierung. Weitere Informati onen finden Sie unter Schritt 2: Erstellen eines Chatrooms unter Erste Schritte mit Amazon IVS Chat.			✓
Clientseitige Zusammensetzung	Verwendet ein Host-Gerät zum Mischen von Audio- und Videostreams von Stage-Teilnehmern und sendet sie dann als zusammengesetzten Stream an einen IVS-Kanal. Dies ermöglicht eine bessere Kontrolle über das Erscheinungsbild der Zusammensetzung, allerdings auf Kosten einer höheren Auslastung der Client-Ressourcen und eines höheren Risikos, dass sich ein Stage- oder Host-Problem auf die Zuschauer auswirkt. Weitere Informationen finden Sie unter serversei tige Zusammensetzung.	✓	✓	
CloudFront	Ein von Amazon bereitgestellter CDN-Service.	✓		
CloudTrail	Ein AWS-Service zur Erfassung, Überwachung, Analyse und Beibehaltung von Ereignissen und Kontoaktivitäten von AWS und externen Quellen. Weitere Informationen finden Sie unter Protokoll ierung von IVS-API-Aufrufen mit AWS CloudTrail.	✓	✓	✓

Begriff	Beschreibung	LL	RT	Chat
CloudWatch	Ein AWS-Service zur Überwachung von Anwendungen, zur Reaktion auf Leistungs änderungen, zur Optimierung der Ressource nnutzung und zur Bereitstellung von Einblicken in den Betriebszustand. Sie können CloudWatc h verwenden, um IVS-Metriken zu überwachen. Weitere Informationen finden Sie unter Überwachu ng von IVS-Echtzeit-Streaming und Überwachung von IVS-Streaming mit niedriger Latenz.	✓	✓	✓
Composition	Der Prozess des Zusammenführens von Audio- und Videostreams aus mehreren Quellen zu einem einzigen Stream.	✓	✓	
Pipeline der Zusammensetzung	Eine Abfolge von Verarbeitungsschritten, die zum Kombinieren mehrerer Streams und zum Kodieren des resultierenden Streams erforderlich sind.	✓	✓	
Komprimierung	Kodierung von Informationen mit weniger Bits als in der ursprünglichen Darstellung. Jede einzelne Komprimierung ist entweder verlustfrei oder verlustbehaftet. Die verlustfreie Komprimierung reduziert die Anzahl der Bits durch die Identifiz ierung und Beseitigung statistischer Redundanz. Bei der verlustfreien Komprimierung gehen keine Informationen verloren. Bei der verlustbehafteten Komprimierung werden Bits reduziert, indem unnötige oder weniger wichtige Informationen entfernt werden.	✓	✓	
Steuerebene	Speichert Informationen zu IVS-Ressourcen wie Kanälen, Stages oder Chatrooms und stellt Schnittstellen zum Erstellen und Verwalten dieser Ressourcen bereit. Es ist regional (basierend auf AWS-Regionen).	✓	✓	✓

Begriff	Beschreibung	LL	RT	Chat
CORS	Herkunftsübergreifende Ressourcenfreigabe, ein AWS-Feature, mit dem Client-Webanwendungen, die in einer Domain geladen sind, mit Ressourcen wie S3-Buckets in einer anderen Domain interagieren können. Der Zugriff kann basierend auf Headern, HTTP-Methoden und Ursprungsdomains konfiguriert werden. Weitere Informationen finden Sie unter Nutzung der herkunftsübergreifenden Ressourcennutzung (CORS) – Amazon Simple Storage Service im Benutzerhandbuch für Amazon Simple Storage Service.	✓		
Benutzerdefinierte Bildquelle	Eine Schnittstelle, die vom IVS Broadcast SDK bereitgestellt wird und es einer Anwendung ermöglicht, ihre eigene Audioeingabe bereitzustellen, anstatt auf das integrierte Mikrofon des Geräts beschränkt zu sein.		✓	
Benutzerdefinierte Bildquelle	Eine vom IVS Broadcast SDK bereitgestellte Schnittstelle, über die eine Anwendung ihre eigene Bildeingabe bereitstellen kann, anstatt auf die voreingestellten Kameras beschränkt zu sein.	✓	✓	
Benutzerdefinierte Teilnehmeranordnung	Ermöglicht die Positionierung von Bühnenteilnehmern sowohl im Grid- als auch im PiP-Layout auf der Grundlage von benutzerdefinierten Attributwerten in Teilnehmer-Token.		✓	
Datenebene	Die Infrastruktur, die Daten von der Aufnahme bis zum Ausgang überträgt. Der Betrieb basiert auf der in der Steuerebene verwalteten Konfiguration und ist nicht auf eine AWS-Region beschränkt.	✓	✓	✓

Begriff	Beschreibung	LL	RT	Chat
Encoder, Verschlüsselung	Der Vorgang der Konvertierung von Video- und Audioinhalten in ein für Streaming geeignetes digitales Format. Die Kodierung kann hardware- oder softwarebasiert sein.	✓	✓	
E-RTMP	Verbessertes RTMP-Protokoll. IVS unterstützt die Funktionen von E-RTMP, die für Multitrack-Video erforderlich sind.	✓		
Ereignis	Eine automatische Benachrichtigung, die von IVS an den AmazonEventBridge-Überwachungsservice veröffentlicht wird. Ein Ereignis stellt eine Zustands- oder Zustandsänderung einer Streaming-Ressource dar, beispielsweise einer Stage oder einer Zusammensetzungs-Pipeline. Weitere Informationen finden Sie unter Verwendung von Amazon EventBridge mit IVS-Streaming mit niedriger Latenz und Verwendung von Amazon EventBridge mit IVS-Echtzeit-Streaming.	✓	✓	✓
FFmpeg	Ein kostenloses Open-Source-Softwareprojekt, das aus einer Reihe von Bibliotheken und Programmen zur Verarbeitung von Video- und Audiodateien und -streams besteht. FFmpeg bietet eine plattformübergreifende Lösung zum Aufzeichnen, Konvertieren und Streamen von Audio und Video.	✓		

Begriff	Beschreibung	LL	RT	Chat
Fragmentierter Stream	Wird erstellt, wenn eine Übertragung innerhalb des in der Aufzeichnungskonfiguration des Kanals angegebenen Intervalls unterbrochen und dann erneut verbunden wird. Die resultier enden mehreren Streams werden als eine einzelne Übertragung betrachtet und zu einem einzigen aufgezeichneten Stream zusammengeführt. Weitere Informationen finden Sie unter Zusammenf ühren fragmentierter Streams in Automatische Aufzeichnung in Amazon S3 (Streaming mit niedriger Latenz).	✓		
Bildrate	Eine Streaming-Metrik für die Anzahl der übertrage nen oder empfangenen Videobilder pro Sekunde.	✓	✓	
HLS	HTTP Live Streaming (HLS), ein HTTP-basiertes Streaming-Kommunikationsprotokoll mit adaptiver Bitrate, das zur Bereitstellung von IVS-Streams an Zuschauer verwendet wird.	✓		
HLS-Wiedergabelist e	Eine Liste von Mediensegmenten, aus denen ein Stream besteht. Standard-HLS-Wiedergabelisten bestehen aus 10-sekündigen Mediendateien. HLS unterstützt auch detailliertere Wiedergabelisten im Bytebereich.	✓		
Host	Ein Echtzeit-Benutzer, der eine Stage erstellt.		✓	
IAM	Identity and Access Management, ein AWS-Servi ce, der Benutzern die sichere Verwaltung von Identitäten und Zugriff auf AWS-Services und -Ressourcen, einschließlich IVS, ermöglicht.	✓	✓	✓

Begriff	Beschreibung	LL	RT	Chat
Ergest	IVS-Prozess zum Empfang von Videostreams von einem Host oder Sender zur Verarbeitung oder Bereitstellung an Zuschauer oder andere Teilnehmer.	✓	✓	
Server Ingest	Empfängt Videostreams und überträgt sie an ein Transkodierungssystem, wo Streams für die Bereitstellung an die Zuschauer in HLS transmuxiert oder transkodiert werden. Aufnahme-Server sind spezielle IVS-Komponenten, die Streams für Kanäle zusammen mit einem Aufnahmeprotokoll (RTMP, RTMPS) empfangen. Weitere Informationen zum Erstellen eines Kanals finden Sie unter Erste Schritte mit IVS-Streaming mit niedriger Latenz.	✓		
Video mit Zeilensprung	Überträgt und zeigt nur ungerade oder gerade Zeilen von aufeinanderfolgenden Frames an, um eine wahrgenommene Verdoppelung der Bildfrequenz zu erreichen, ohne zusätzliche Bandbreite zu verbrauchen. Aufgrund von Bedenken hinsichtlich der Videoqualität wird die Verwendung von Video mit Zeilensprung nicht empfohlen.	✓	✓	
JSON	JavaScript Object Notation, ein Open-Standard-Dateiformat, das für Menschen lesbaren Text verwendet, um Datenobjekte zu übertragen, die aus Attribut-Wert-Paaren und Array-Datentypen oder anderen serialisierbaren Werten bestehen.	✓	✓	✓

Begriff	Beschreibung	LL	RT	Chat
Keyframe, Delta-Frame, Keyframe-Intervall	Der Keyframe (auch als intra-kodiert oder i-Frame bezeichnet) ist ein Vollbild des Bildes in einem Video. Nachfolgende Frames, die Deltaframes (auch als prognostizierte oder p-Frames bezeichnet), enthalten nur die geänderten Informationen. Abhängig vom im Encoder definierten Keyframe-Intervall werden Keyframes innerhalb eines Streams mehrmals angezeigt.	✓	✓	
Lambda	Ein AWS-Service zum Ausführen von Code (als Lambda-Funktionen bezeichnet) ohne Bereitstellung einer Serverinfrastruktur. Lambda-Funktionen können als Reaktion auf Ereignisse und Aufrufanfragen oder basierend auf einem Zeitplan ausgeführt werden. IVS Chat verwendet beispielsweise Lambda-Funktionen, um die Nachrichtenüberprüfung für einen Chatroom zu ermöglichen.	✓	✓	✓
Latenz, Glas-zu-Glas-Latenz	Eine Verzögerung bei der Datenübertragung. IVS definiert Latenzbereiche wie folgt: • Niedrige Latenz: unter 3 Sekunden • Latenz in Echtzeit: unter 300 ms Glas-zu-Glas-Latenz bezieht sich auf die Verzögerung, wenn eine Kamera einen Livestream aufnimmt, bis zu dem Zeitpunkt, an dem der Stream auf dem Bildschirm eines Betrachters angezeigt wird.	✓	✓	
Mehrschichtige Kodierung mit Simulcast	Ermöglicht die gleichzeitige Kodierung und Veröffentlichung mehrerer Videostreams mit unterschiedlichen Qualitätsstufen. Weitere Informationen finden Sie unter Adaptives Streaming: Mehrschichtige Kodierung mit Simulcast in Streaming-Optimierungen in Echtzeit.		✓	

Begriff	Beschreibung	LL	RT	Chat
Handler für Nachrichtenüberprüfung	Ermöglicht es IVS-Chat-Kunden, Benutzer-Chat-Nachrichten automatisch zu überprüfen/zu filtern, bevor sie an den Chatroom übermittelt werden. Dies wird durch die Verknüpfung einer Lambda-Funktion mit einem Chatroom ermöglicht. Weitere Informationen finden Sie unter Erstellen einer Lambda-Funktion im Handler für Nachrichtenüberprüfung.			✓
Mischpult	Ein Feature der IVS Mobile Broadcast SDKs, die mehrere Audio- und Videoquellen aufnimmt und eine einzige Ausgabe generiert. Diese Funktion unterstützt die Verwaltung von Video- und Audioelementen auf dem Bildschirm, die Quellen wie Kameras, Mikrofone, Bildschirmaufnahmen sowie von der Anwendung generiertes Audio und Video darstellen. Die Ausgabe kann anschließend an IVS gestreamt werden. Weitere Informationen finden Sie unter Konfigurieren einer Broadcast-Sitzung zum Mischen im IVS Broadcast SDK: Mixer-Handbuch (Streaming mit niedriger Latenz).	✓		
Streaming auf mehreren Hosts	Kombiniert Streams von mehreren Hosts zu einem einzigen Stream. Dies kann entweder durch clientseitige oder serverseitige Zusammensetzung erreicht werden. Das Streaming mit mehreren Hosts ermöglicht Szenarien wie die Einladung von Zuschauern auf eine Stage für Fragen und Antworten, Wettbewerbe zwischen Hosts, Videochats und Gespräche zwischen den Hosts vor einem großen Publikum.		✓	

Begriff	Beschreibung	LL	RT	Chat
Multitrack-Video	Ermöglicht Broadcaster-Softwaretools das Kodieren und Streaming mehrerer Videoqualitäten direkt von einem GPU-gestützten Computer aus. Siehe Multitrack-Video von Amazon IVS.	✓		
Multivariante Wiedergabeliste	Ein Index aller Varianten-Streams, die für eine Übertragung verfügbar sind.	✓		
OAC	Origin Access Control, ein Mechanismus zur Einschränkung des Zugriffs auf einen S3-Bucket , sodass Inhalte wie ein aufgezeichneter Stream nur über CloudFront CDN bereitgestellt werden können.	✓		
OBS	Open Broadcaster Software, kostenlose und Open-Source-Software für Videoaufzeichnung und Live-Streaming. OBS bietet eine Alternative (zum IVS Broadcast SDK) für Desktop-Publishing. Erfahrene re Streamer, die mit OBS vertraut sind, bevorzuge n es möglicherweise aufgrund seiner erweitert en Produktionsfeatures wie Szenenübergänge, Audiomischung und Overlay-Grafiken.	✓	✓	
Teilnehmer	Ein Echtzeit-Benutzer, der als Publisher oder Subscriber mit einer Stage verbunden ist.		✓	
Teilnehmerreihenfo lge	Die Reihenfolge, in der die Teilnehmer der Phase in Grid- und PiP-Layouts positioniert sind.		✓	
Teilnehmer-Token	Authentifiziert einen Teilnehmer eines Ereigniss es in Echtzeit, wenn er einer Stage beitritt. Ein Teilnehmer-Token steuert auch, ob ein Teilnehmer Videos an die Stage senden kann.		✓	

Begriff	Beschreibung	LL	RT	Chat
Wiedergabe-Token, Wiedergabe-Schlüsselpaar	Ein Autorisierungsmechanismus, mit dem Kunden die Videowiedergabe auf privaten Kanälen beschränken können. Wiedergabe-Token werden aus einem Wiedergabe-Schlüsselpaar generiert. Ein Wiedergabe-Schlüsselpaar ist das öffentlic h-private Schlüsselpaar, das zum Signieren und Validieren des Viewer-Autorisierungs-Token für die Wiedergabe verwendet wird. Siehe Erstellen oder Importieren eines IVS-Wiedergabeschlüssels unter Einrichten von IVS-Privatkanälen und die Vorgänge für Wiedergabeschlüsselpaare unter Referenz zur API von IVS mit niedriger Latenz.	✓		
Wiedergabe-URL	Gibt die Adresse an, die ein Zuschauer verwendet , um die Wiedergabe für einen bestimmten Kanal zu starten. Diese Adresse kann global verwendet werden. Amazon IVS wählt automatisch den besten Standort im globalen Netzwerk für die Inhaltsbe reitstellung von IVS aus, um das Video an jeden Zuschauer zu übermitteln. Weitere Informationen zum Erstellen eines Kanals finden Sie unter Erste Schritte mit IVS-Streaming mit niedriger Latenz.	✓		
Privater Kanal	Ermöglicht Kunden die Einschränkung des Zugriffs auf ihre Streams mithilfe eines Autorisierungsmech anismus, der auf Wiedergabe-Tokens basiert. Siehe Workflow für private IVS-Kanäle unter Einrichten von privaten IVS-Kanälen.	✓		
Private Erfassung	Ermöglicht eine sichere private Verbindung zwischen Ihrer Amazon VPC und IVS mithilfe von VPC-Schnittstellen-Endpunkten, die von AWS PrivateLink betrieben werden. Siehe Private Erfassung in IVS	✓		

Begriff	Beschreibung	LL	RT	Chat
Progressives Video	Überträgt und zeigt alle Zeilen jedes Frames der Reihe nach an. Es empfiehlt sich die Verwendung von progressivem Video in allen Stages einer Übertragung.	✓	✓	
Publisher	Ein Teilnehmer eines Echtzeit-Ereignisses, der Video und/oder Audio auf einer Stage veröffentlicht. Weitere Informationen finden Sie unter Was ist IVS-Echtzeit-Streaming?.		✓	
Kontingente	Die maximale Anzahl von IVS-Serviceressourcen oder -Vorgängen für Ihr AWS-Konto. Das heißt, diese Grenzwerte gelten pro AWS-Konto, sofern nicht anders angegeben. Alle Kontingente werden pro Region erzwungen. Weitere Informationen finden Sie unter Endpunkten und Kontingenten von Amazon Interactive Video Service im Allgemeinen AWS-Referenzhandbuch.	✓	✓	✓

Begriff	Beschreibung	LL	RT	Chat
Regionen	Bieten Sie Zugriff auf AWS-Services, die sich physisch in einem bestimmten geografischen Gebiet befinden. Regionen bieten Fehlertoleranz, Stabilität und Ausfallsicherheit und können auch die Latenz verkürzen. Mit Regionen können Sie redundante Ressourcen erstellen, die verfügbar bleiben und von einem regionalen Ausfall nicht betroffen werden. Die meisten AWS-Serviceanfragen beziehen sich auf eine bestimmte geografische Region. Die Ressourcen, die Sie in einer Region erstellen, sind in keiner anderen Region vorhanden, es sei denn, Sie verwenden ausdrücklich ein Replikati onsfeature, die von einem AWS-Service angeboten wird. Beispielsweise unterstützt Amazon S3 die regionsübergreifende Replikation. Einige Services, wie etwa IAM, verfügen über keine regionsüb ergreifende Ressourcen.	✓	✓	✓
Auflösung	Beschreibt die Anzahl der Pixel in einem einzelnen Videobild. Full HD oder 1080p definiert beispiels weise ein Frame mit 1920x1080 Pixeln.	✓	✓	
Stammbenutzer	Der Besitzer eines AWS-Kontos. Der Root-Benu tzer hat vollständigen Zugriff auf alle AWS-Services und Ressourcen im AWS-Konto.	✓	✓	✓
RTMP, RTMPS	Real-Time Messaging Protocol, ein Branchens tandard zum Übertragen von Audio, Video und Daten über ein Netzwerk. RTMPS ist die sichere Version von RTMP, die über eine Transport Layer Security (TLS/SSL)-Verbindung ausgeführt wird.	✓	✓	

Begriff	Beschreibung	LL	RT	Chat
S3-Bucket	Eine Sammlung von Objekten, die in Amazon S3 gespeichert sind. Viele Richtlinien, einschließlich Zugriff und Replikation, werden auf Bucket-Ebene definiert und gelten für alle Objekte im Bucket. Beispielsweise wird eine IVS-Übertragung in Form mehrerer Objekte in einem S3-Bucket gespeichert.	✓		
SDK	Software Development Kit, eine Sammlung von Bibliotheken für Entwickler, die Anwendungen mit IVS erstellen. Das IVS Player SDK dient der Wiedergabe von IVS-Streams. Es nutzt die IVS-Architektur und ist für die IVS-Wiedergabe mit niedriger Latenz optimiert. IVS bietet ein Broadcast-SDK für Web, Android und iOS, das Sie in Ihre Anwendung integrieren können. Das IVS-Broadcast-SDK ist für Entwickler konzipiert, die Android- oder iOS-Anwendungen mit IVS erstellen. Dieses SDK nutzt die IVS-Architektur und wird zusammen mit IVS kontinuierlich verbessert. Als natives Broadcast-SDK wurde es entwickelt, um die Leistungsauswirkungen auf Ihre Anwendungen und auf die Geräte, mit denen Ihre Benutzer auf Ihre Anwendungen zugreifen, zu minimieren. Es gibt IVS-Broadcast-SDKs für Web, Android und iOS für Streaming mit niedriger Latenz und Echtzeit-Streaming.	✓	✓	✓

Begriff	Beschreibung	LL	RT	Chat
Selfie-Segmentation	Ermöglicht das Ersetzen des Hintergrunds in einem Live-Stream mithilfe einer Client-spezifische n Lösung. Diese akzeptiert ein Kamerabild als Eingabe und gibt eine Maske zurück, die für jedes Pixel des Bildes eine Wertung bereitste llt und angibt, ob es sich im Vordergrund oder im Hintergrund befindet. Weitere Informationen finden Sie unter Ersetzen des Hintergrunds in IVS Broadcast SDK: Kamera-Filter von Drittanbietern (Echtzeit-Streaming).		✓	
Semantische Versionsverwaltung	Ein Versionsformat in Form von Major.Minor.Patch. Fehlerkorrekturen, die sich nicht auf die API auswirken, erhöhen die Patch-Version, rückwärts kompatible API-Ergänzungen/Änderungen erhöhen die Nebenversion und rückwärtsinkompatible API-Änderungen erhöhen die Hauptversion.	✓	✓	✓
Serverseitige Zusammensetzung	Verwendet einen IVS-Server zum Mischen von Audio und Video von Teilnehmern einer Stage und sendet dieses gemischte Video dann an einen IVS-Kanal, um ein größeres Publikum zu erreichen oder um es in einem S3-Bucket zu speichern. Die serverseitige Zusammensetzung reduziert die Client-Auslastung, verbessert die Stabilität der Übertragung und ermöglicht eine effizientere Nutzung der Bandbreite. Weitere Informationen finden Sie unter Clientseitige Zusammensetzung.		✓	

Begriff	Beschreibung	LL	RT	Chat
Servicekontingente	Ein AWS-Service, mit dem Sie Ihre Kontingente für viele AWS-Services von einem Standort aus verwalten können. Sie können über die Service–Quotas-Konsole Kontingentwerte abfragen und außerdem Kontingenterhöhungen anfordern.	✓	✓	✓
Serviceverknüpfte Rolle	Ein eindeutiger IAM-Rollentyp, der direkt mit einem AWS-Service verknüpft ist. Serviceverknüpfte Rollen werden automatisch von IVS erstellt und enthalten alle Berechtigungen, die der Service benötigt, um andere AWS-Services in Ihrem Namen aufzurufen, z. B. für den Zugriff auf einen S3-Bucket. Weitere Informationen finden Sie unter Nutzung serviceverknüpfter Rollen für IVS in IVS-Sicherheit.	✓		
Stage	Eine IVS Ressource, die eine virtuelle Umgebung darstellt, in dem die Echtzeit-Teilnehmer eines Ereignisses Videos in Echtzeit austauschen können. Weitere Informationen finden Sie unter Erstellen einer Stage mit optionaler Teilnehme raufzeichnung in Erste Schritte mit IVS-Echtzeit-Streaming.		✓	
Stages-Sitzung	Beginnt, wenn der erste Teilnehmer eine Stage betritt und endet einige Minuten, nachdem der letzte Teilnehmer die Veröffentlichung in der Stage beendet hat. Eine langlebige Stage kann im Laufe ihrer Lebensdauer möglicherweise mehrere Sitzungen haben.		✓	
Stream	Daten, die Video- oder Audioinhalte darstellen und fortlaufend von einer Quelle an ein Ziel gesendet werden.	✓	✓	

Begriff	Beschreibung	LL	RT	Chat
Stream-Schlüssel	Eine von IVS beim Erstellen eines Kanals zugewiesene Kennung. Es wird verwendet, um das Streaming zum Kanal zu autorisieren. Behandeln Sie den Stream-Schlüssel wie ein Geheimnis, da jeder mit ihm berechtigt ist auf den Kanal zu streamen. Weitere Informationen finden Sie unter Erste Schritte mit IVS-Streaming mit niedriger Latenz.	✓		
Stream-Starvation	Eine Verzögerung oder ein Stopp bei der Stream-Übermittlung an IVS. Dies tritt auf, wenn IVS nicht die erwartete Anzahl an Bits empfängt, die das Kodierungsgerät angekündigt hat und die es über einen bestimmten Zeitraum senden würde. Das Auftreten einer Stream-Starvation führt zu einem Stream-Starvation-Ereignis. Aus der Sicht eines Zuschauers kann eine Stream-Starvation als verzögertes, pufferndes oder einfrierendes Video erscheinen. Die Stream-St arvation kann kurz (weniger als 5 Sekunden) oder lang (mehrere Minuten) sein, abhängig von der spezifischen Situation, die zur Stream-Starvation geführt hat. Weitere Informationen finden Sie unter Was ist Stream-Starvation in den Häufig gestellten Fragen zur Fehlerbehebung.	✓	✓	
Streamer	Eine Person oder ein Gerät, das einen Video- oder Audio-Stream an IVS sendet.	✓	✓	
Subscriber	Ein Teilnehmer eines Echtzeit-Ereignisses, der Video und/oder Audio von Stage-Publishern empfängt. Weitere Informationen finden Sie unter Was ist IVS-Echtzeit-Streaming?.		✓	

Begriff	Beschreibung	LL	RT	Chat
Tag	Ein Tag ist eine Markierung, die Sie einer AWS-Ressource zuordnen. Mithilfe von Tags können Sie Ihre AWS-Ressourcen leichter identifizieren und anordnen. Auf der Startseite der IVS-Dokumentation finden Sie den Abschnitt „Tagging“ in einer beliebigen IVS-API-Dokumentation (für Echtzeit-Streaming, Streaming mit niedriger Latenz oder Chat).	✓	✓	✓
Kamerafilter von Drittanbietern	Softwarekomponenten, die in das IVS Broadcast SDK integriert werden können, damit eine Anwendung Bilder verarbeiten kann, bevor sie dem Broadcast SDK als benutzerdefinierte Image-Quelle bereitgestellt werden. Ein Kamerafilter eines Drittanbieters kann Bilder von der Kamera verarbeiten, einen Filtereffekt anwenden usw.	✓	✓	
Miniaturansicht	Ein verkleinertes Bild, das aus einem Stream aufgenommen wurde. Standardmäßig werden Miniaturansichten alle 60 Sekunden generiert, es kann jedoch ein kürzeres Intervall konfiguriert werden. Die Auflösung der Miniaturansicht hängt vom Kanaltyp ab. Weitere Informationen finden Sie unter Aufzeichnen von Inhalten in Automatische Aufnahme in Amazon S3 (Streaming mit niedriger Latenz).	✓		

Begriff	Beschreibung	LL	RT	Chat
Zeitgesteuerte Metadaten	An bestimmte Zeitstempel innerhalb eines Streams gebundene Metadaten. Dies kann programmg esteuert mithilfe der IVS-API hinzugefügt werden und wird bestimmten Frames zugeordnet. Dadurch wird sichergestellt, dass alle Zuschauer die Metadaten an der gleichen Stelle relativ zum Stream erhalten. Zeitgesteuerte Metadaten können verwendet werden, um Aktionen auf dem Client auszulöse n, z. B. die Aktualisierung von Teamstatistiken während einer Sportveranstaltung. Weitere Informationen finden Sie unter Einbettung von Metadaten in einen Video-Stream.	✓		
Token-Austausch	Eine vom IVS Broadcast SDK bereitgestellte Schnittstelle, mit der Teilnehmer-Token-Funktionen aktualisiert oder herabgestuft und Token-Att ribute aktualisiert werden können, ohne dass die Teilnehmer erneut eine Verbindung herstelle n müssen. Dies ermöglicht Szenarien wie Co-Hosting, bei denen Teilnehmer zunächst nur über Subscribe-Funktionen verfügen und später Veröffentlichungsfunktionen benötigen.		✓	
Transkodierung	Wandelt Video und Audio von einem Format in ein anderes um. Ein eingehender Stream kann in ein anderes Format mit mehreren Bitraten und Auflösungen transkodiert werden, um eine Reihe von Wiedergabegeräten und Netzwerkbedingungen zu unterstützen.	✓	✓	

Begriff	Beschreibung	LL	RT	Chat
Transmuxing	Ein einfaches Umpacken eines aufgenommenen Streams in IVS ohne erneute Kodierung des Videostreams. „Transmux“ ist die Abkürzung für Transcode-Multiplexing, ein Prozess, der das Format einer Audio- und/oder Videodatei ändert und dabei einige oder alle der ursprünglichen Streams beibehält. Transmuxing konvertiert in ein anderes Containerformat, ohne den Dateiinhalt zu ändern. Unterscheidet sich von Transkodierung.	✓	✓	
Varianten-Streams	Eine Reihe von Kodierungen derselben Übertragung in verschiedenen Qualitätsstufen. Jeder Varianten-Stream wird als separate HLS-Wiedergabeliste kodiert. Ein Index der verfügbaren Varianten-Streams wird als multivariante Wiedergabeliste bezeichnet. Nachdem der IVS-Player eine multivariante Playlist von IVS empfangen hat, kann er während der Wiedergabe zwischen den Varianten-Streams auswählen und bei sich ändernden Netzwerkbedingungen nahtlos hin und her wechseln.	✓		
VBR	Variable Bitrate, eine Methode der Ratensteuerung für Encoder, die eine dynamische Bitrate verwendet, die sich während der Wiedergabe je nach erforderlicher Detailebene ändert. Aus Gründen der Videoqualität raten wir nachdrücklich davon ab, VBR zu verwenden. Verwenden Sie stattdessen CBR.	✓	✓	

Begriff	Beschreibung	LL	RT	Chat
Anzeigen	Eine einzelne Anzeigesitzung, die aktiv Mediendat eien herunterlädt oder abspielt. Aufrufe sind die Grundlage für das Kontingent gleichzeitiger Aufrufe. Eine Ansicht beginnt, wenn eine Anzeigesitzung die Videowiedergabe beginnt. Eine Ansicht endet, wenn eine Anzeigesitzung die Videowiedergabe stoppt. Die Wiedergabe ist der einzige Indikator für die Zuschauerschaft; Interaktionsheuristiken wie Audiopegel, Browser-Tab-Fokus und Videoqual ität werden nicht berücksichtigt. Beim Zählen der Aufrufe berücksichtigt IVS nicht die Legitimität einzelner Zuschauer und versucht auch nicht, lokalisierte Zuschauerzahlen zu deduplizieren, z. B. mehrere Videoplayer auf einem einzigen Computer. Weitere Informationen finden Sie unter Andere Kontingente in Service Quotas (Streaming mit niedriger Latenz).	✓		
Zuschauer	Eine Person, die einen Stream von IVS empfängt.	✓		

Begriff	Beschreibung	LL	RT	Chat
WebRTC	Web Real-Time Communication, ein Open-Source-Projekt, das Webbrowsern und mobilen Anwendungen Echtzeitkommunikation bietet. Es ermöglicht die Audio- und Videokommunikation innerhalb von Webseiten, indem es eine direkte Peer-to-Peer-Kommunikation erlaubt, ohne dass Plugins installiert oder native Anwendungen heruntergeladen werden müssen. Die Technologien hinter WebRTC werden als offener Webstandard implementiert und sind als reguläre JavaScript-APIs in allen gängigen Browsern oder als Bibliotheken für native Clients wie Android und iOS verfügbar.	✓	✓	

Begriff	Beschreibung	LL	RT	Chat
WHIP	WebRTC-HTTP Ingestion Protocol, ein HTTP-basiertes Protokoll, das die WebRTC-basierte Aufnahme von Inhalten in Streaming-Services und/oder CDNs ermöglicht. WHIP ist ein IETF-Entwurf, der zur Standardisierung der WebRTC-Aufnahme entwickelt wurde. WHIP ermöglicht die Kompatibilität mit Software wie OBS und bietet eine Alternative (zum IVS-Broadcast-SDK) für Desktop-Publishing. Erfahrenere Streamer, die mit OBS vertraut sind, bevorzugen es möglicherweise aufgrund seiner erweiterten Produktionsfeatures wie Szenenübergänge, Audiomischung und Overlay-Grafiken. WHIP ist auch von Vorteil, wenn die Verwendung des IVS-Broadcast-SDK nicht möglich oder nicht erwünscht ist. Beispielsweise ist das IVS-Broadcast-SDK in Setups mit Hardware-Encodern möglicherweise keine Option. Wenn der Encoder jedoch WHIP unterstützt, können Sie trotzdem direkt vom Encoder in IVS veröffentlichen. Siehe IVS-WHIP-Unterstützung (Echtzeit-Streaming).		✓	
WSS	WebSocket Secure, ein Protokoll zum Einrichten von WebSockets über eine verschlüsselte TLS-Verbindung. Dies wird zum Herstellen einer Verbindung mit IVS-Chat-Endpunkten verwendet. Weitere Informationen finden Sie unter Schritt 4: Senden und Empfangen Ihrer ersten Nachricht in Erste Schritte mit IVS-Chat.			✓

IVS-Chat-Dokumentenverlauf

In den folgenden Tabellen werden die wichtigen Änderungen an der Dokumentation für Amazon IVS Chat beschrieben. Wir aktualisieren die Dokumentation regelmäßig, um neue Versionen zu berücksichtigen und auf Ihr Feedback einzugehen.

Änderungen am Chat-Benutzerhandbuch

Änderung	Beschreibung	Datum
iOS-Integration	CocoaPods wird nicht mehr verwendet. Es wurden entsprechende Änderungen an der Dokumentation im IVS Chat Client Messaging SDK: iOS-Handbuch (unter „Erste Schritte“) vorgenommen.	13. Februar 2026
Chat-Client-Messaging-SDK: iOS 1.0.1	Versionsnummer und Artefakt-Links in den Player-SDK-Anleitungen aktualisiert: iOS. Siehe auch Versionshinweise.	8. August 2025
eine Chat-UG aufteilen	Diese Version enthält wichtige Änderungen an der Dokumentation. Wir haben die Chat-Informationen aus dem IVS-Benutzerhandbuch für Streaming mit niedriger Latenz in ein neues IVS-Chat-Benutzerhandbuch verschoben. Dieses befindet sich im vorhandenen Abschnitt IVS-Chat auf der Startseite der IVS-Dokumentation.	28. Dezember 2023

	Weitere Änderungen an der Dokumentation finden Sie unter Dokumentenverlauf (Streaming mit niedriger Latenz).	
IVS-Glossar	Das Glossar wurde um IVS-Begriffe in Echtzeit, niedriger Latenz und Chat erweitert.	20. Dezember 2023

Änderungen an der IVS-Chat-API-Referenz

API-Änderungen	Beschreibung	Datum
Update bei SendEvent	Die maximale Metadatenlänge des Felds `attributes` ist von 1 KB auf 4 KB gestiegen.	19. November 2025
Eine Chat UG aufteilen	Da es jetzt ein IVS-Chat-Benutzerhandbuch gibt (mit dieser Version), werden sich die Dokumentenverlaufseinträge für die vorhandene IVS-Chat-API-Referenz und die IVS-Chat-Messaging-API-Referenz in Zukunft hier befinden. Frühere Verlaufseinträge für diese Chat-API-Referenzen befinden sich im Dokumentenverlauf (Streaming mit niedriger Latenz).	28. Dezember 2023

IVS-Chat-Versionshinweise

Dieses Dokument enthält alle Versionshinweise zu Amazon IVS Chat, beginnend mit den neuesten, geordnet nach dem Datum ihrer Veröffentlichung.

8. August 2025

Client-Messaging-SDK für Amazon IVS Chat: iOS 1.0.1

Plattform	Downloads und Änderungen
iOS Chat Client Messaging SDK 1.0.1	Referenzdokumentation: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.1/ • Wir haben den eingebetteten Bitcode aus dem SDK entfernt.

Chat Client Messaging SDK Size: iOS

Architektur	Komprimierte Größe	Unkomprimierte Größe
ios-arm64_x86_64-simulator	256 KB	807 KB
ios-arm64	124 KB	397 KB

28. Dezember 2023

Benutzerhandbuch zu Amazon IVS Chat

Amazon Interactive Video Service (IVS) Chat ist ein verwaltetes Live-Chat-Feature, das neben Live-Videostreams genutzt werden kann. In dieser Version haben wir Chat-Informationen aus dem Benutzerhandbuch für IVS-Streaming mit niedriger Latenz in ein neues IVS-Chat-Benutzerhandbuch verschoben. Die Dokumentation ist über die Zielseite für die Amazon-IVS-Dokumentation verfügbar:

31. Januar 2023

Client-Messaging-SDK für Amazon IVS Chat: Android 1.1.0

Plattform	Downloads und Änderungen
Android Chat Client Messaging SDK 1.1.0	Referenzdokumentation: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/ • Um Kotlin-Coroutines zu unterstützen, haben wir dem Paket com.amazonaws.ivs.chat.messaging.coroutines neue IVS-Chat-Messaging-APIs hinzugefügt. Sehen Sie sich auch das neue Kotlin-Coroutines Tutorial an; Teil 1 (von 2) behandelt Chaträume.

Chat Client Messaging SDK Size: Android

Architektur	Komprimierte Größe	Unkomprimierte Größe
Alle Architekturen (Bytecode)	89 KB	92 KB

9. November 2022

Amazon IVS Chat Client Messaging SDK: JavaScript 1.0.2

Plattform	Downloads und Änderungen
JavaScript Chat Client Messaging SDK 1.0.2	Referenzdokumentation: https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/ • Es wurde ein Problem in Firefox behoben: Clients erhielten fälschlicherweise einen Socket-Fehler, wenn die Verbindung zu

Plattform	Downloads und Änderungen
	einem Chatroom über den DisconnectUser-Endpunkt getrennt wurde.

8. September 2022

Amazon IVS Chat Client Messaging: Android 1.0.0 und iOS 1.0.0

Plattform	Downloads und Änderungen
Android Chat Client Messaging SDK 1.0.0	Referenzdokumentation: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/
iOS Chat Client Messaging SDK 1.0.0	Referenzdokumentation: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/

Chat Client Messaging SDK Size: Android

Architektur	Komprimierte Größe	Unkomprimierte Größe
Alle Architekturen (Bytecode)	53 KB	58 KB

Chat Client Messaging SDK Size: iOS

Architektur	Komprimierte Größe	Unkomprimierte Größe
ios-arm64_x86_64-simulator (Bitcode)	484 KB	2,4 MB
ios-arm64_x86_64-simulator	484 KB	2,4 MB
ios-arm64 (Bitcode)	1,1 MB	3,1 MB
ios-arm64	233 KB	1,2 MB