Guida per l'utente di Chat

Amazon IVS

Amazon IVS: Guida per l'utente di Chat

Table of Contents

Che cos'è Chat Amazon IVS?

Chat Amazon IVS è una funzionalità gestita di chat in tempo reale per accompagnare gli stream video dal vivo. La documentazione è accessibile dalla pagina di destinazione della documentazione di Amazon IVS, nella sezione Chat Amazon IVS:

- Guida per l'utente della chat: questo documento, insieme a tutte le altre pagine della Guida per l'utente elencate nel riquadro di navigazione.
- Informazioni di riferimento sull'API Chat: API del piano di controllo (control-plane) (HTTPS).
- Informazioni di riferimento sull'API Chat Messaging: API del piano dati (WebSocket).
- Riferimenti SDK per client di chat: Android, iOS e JavaScript.

Nozioni di base su Chat Amazon IVS

Amazon IVS (Interactive Video Service) Chat è una funzionalità gestita di chat in tempo reale per accompagnare gli stream video dal vivo. La chat IVS può essere utilizzata anche senza streaming video. È possibile creare chat room e abilitare le sessioni di chat tra i propri utenti.

Amazon IVS Chat consente di concentrarsi sulla creazione di esperienze di chat personalizzate insieme a video dal vivo. Non è necessario gestire l'infrastruttura o sviluppare e configurare componenti dei flussi di lavoro di chat. Amazon IVS Chat è scalabile, sicuro, affidabile e conveniente.

Amazon IVS Chat è ideale per facilitare la messaggistica tra i partecipanti di uno stream video dal vivo con un inizio e una fine.

Il resto di questo documento illustra le fasi necessarie per creare la prima applicazione di chat utilizzando Amazon IVS Chat.

Esempi: sono disponibili le seguenti app demo (tre app client di esempio e un'app server di back-end per la creazione di token):

- Demo di Amazon IVS Chat Web
- Demo di Amazon IVS Chat per Android
- Demo di Amazon IVS Chat per iOS
- Demo di back-end di Amazon IVS Chat

Importante: le chat room che non hanno nuove connessioni o aggiornamenti per 24 mesi vengono automaticamente eliminate.

Argomenti

Fase 1: Esecuzione della configurazione iniziale

Per poter procedere, è necessario:

1. Creare un account AWS
2. Configurare gli utenti root e amministrativi
3. Configura le autorizzazioni di AWS IAM (Identity and Access Management). Utilizzare la policy specificata di seguito.

Per i passaggi specifici, consulta Guida introduttiva allo streaming a bassa latenza di IVS nella Guida per l'utente di Amazon IVS. Importante: nella "Fase 3: Configurazione delle autorizzazioni IAM", utilizza questa policy per IVS Chat:

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

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

Passaggio 2: creare una chat room

A una chat room Amazon IVS sono associate le informazioni di configurazione (ad esempio, la lunghezza massima del messaggio).

Le istruzioni in questa sezione mostrano come utilizzare la console o la CLI AWS per configurare le chat room (inclusa la configurazione opzionale per la revisione e/o la registrazione dei messaggi) e creare chat room.

Istruzioni della console per la creazione di una chat room IVS

Questi passaggi sono suddivisi in fasi, a partire dalla configurazione iniziale della chat room fino alla creazione della stessa.

Facoltativamente, è possibile configurare una stanza in modo che i messaggi vengano revisionati. Ad esempio, si può aggiornare il contenuto o i metadati dei messaggi, negare i messaggi per impedirne l'invio o lasciare passare il messaggio originale. Per informazioni, consultare Configurazione per esaminare i messaggi della stanza (facoltativo).

Facoltativamente, puoi anche configurare una stanza in modo che i messaggi vengano registrati. Ad esempio, se hai messaggi inviati a una chat room, puoi registrarli su un bucket Amazon S3, Amazon CloudWatch o Amazon Kinesis Data Firehose. Questa procedura è descritta in Configurazione dei log dei messaggi (facoltativo).

Configurazione iniziale di una stanza

1. Aprire la console di Amazon IVS Chat.

   Si può accedere alla console Amazon IVS anche dalla Console di gestione AWS.
2. Dalla barra di navigazione, utilizzare il menu a discesa Seleziona una Regione per scegliere una regione. La nuova stanza verrà creata in questa regione.
3. Nella casella Get started (Nozioni di base), in alto a destra, scegliere Amazon IVS Chat Room (Chat room di Amazon IVS). Viene visualizzata la finestra Create room (Crea stanza).

Create room Info

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

How Amazon IVS Chat works

Setup

Room name – *optional*

name

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

Room configuration

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

Message character limit Info

500 characters per message

Maximum message rate Info

10 messages per second

Message review handler Info

Review messages before they are sent to the room

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

Message logging Info

Automatically log chat messages

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

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

4. In Setup (Configurazione), specificare facoltativamente un nome stanza. I nomi delle stanze non sono univoci, ma consentono di distinguere le stanze oltre all'ARN (Amazon Resource Name) della stanza.
5. In Setup > Room configuration (Configurazione > Configurazione stanza) accettare la configurazione predefinita o selezionare Custom configuration (Configurazione personalizzata), quindi configurare Maximum message length (Lunghezza massima del messaggio) e/o Maximum message rate (Frequenza massima del messaggio).
6. Se si desidera rivedere i messaggi, continuare con la sezione Configurazione per esaminare i messaggi della stanza (facoltativo) di seguito. Altrimenti, ignorare la sezione (vale a dire, accettare Gestore di revisione dei messaggi > Disabilitato) e andare direttamente a Creazione di una stanza finale.

Configurazione per esaminare i messaggi della stanza (facoltativo)

1. In Gestore di revisione dei messaggi, selezionare Gestisci con AWS Lambda. La sezione Message Review Handler (Gestore di revisione dei messaggi) si espande per mostrare opzioni aggiuntive.
2. Configurare Fallback result (Risultato fallback) per consentire o negare il messaggio se il gestore non restituisce una risposta valida, rileva un errore o supera il periodo di timeout.
3. Specificare la funzione Lambda esistente o utilizzare Create Lambda function (Crea funzione Lambda) per creare una nuova funzione.

   La funzione Lambda deve trovarsi nella stessa regione AWS e nello stesso account AWS della chat room. È necessario concedere al servizio Amazon Chat SDK l'autorizzazione per richiamare la risorsa lambda. La policy basata sulle risorse verrà creata automaticamente per la funzione lambda selezionata. Per ulteriori informazioni sulle autorizzazioni, consulta la pagina Resource-Based Policy for Amazon IVS Chat.

Configurazione dei log dei messaggi (facoltativo)

1. In Message logging (Registrazione messaggi), seleziona Automatically log chat messages (Registra automaticamente i messaggi di chat). La sezione Message logging (Registrazione di messaggi) si espande per mostrare opzioni aggiuntive. Puoi aggiungere una configurazione di registrazione esistente a questa stanza o creare una nuova configurazione di registrazione selezionando Create logging configuration (Crea configurazione di registrazione).

2. Se scegli una configurazione di registrazione esistente, viene visualizzato un menu a discesa che mostra tutte le configurazioni di registrazione che hai già creato. Selezionane uno dall'elenco e i tuoi messaggi di chat saranno registrati automaticamente in questa destinazione.
3. Se scegli Crea configurazione di registrazione, viene visualizzata una finestra modale che consente di creare e personalizzare una nuova configurazione di registrazione.
   a. Facoltativamente, specifica un nome per la configurazione di registrazione. I nomi delle configurazioni di registrazione, così come i nomi delle stanze, non sono univoci, ma consentono di distinguere le configurazioni di registrazione diverse dall'ARN di configurazione di registrazione.
   b. In Destination (Destinazione), seleziona CloudWatch log group (Gruppo di log CloudWatch), Kinesis Firehose Delivery Stream o Amazon S3 bucket (Bucket Amazon S3) per scegliere la destinazione per i log.
   c. A seconda della destinazione, seleziona l'opzione per creare un nuovo gruppo di log o utilizzare un gruppo di log CloudWatch esistente, Kinesis Firehose Delivery Stream o un bucket Amazon S3.
   d. Dopo la revisione, scegli Create (Crea) per creare una nuova configurazione di registrazione con un ARN univoco. In questo modo la nuova configurazione di registrazione viene automaticamente allegata alla chat room.

Creazione di una stanza finale

1. Dopo la revisione, scegli Create chat room (Crea chat room) per creare una nuova chat room con un ARN univoco.

Istruzioni della CLI per la creazione di una chat room IVS

Questo documento illustra i passaggi necessari per creare una Chat room Amazon IVS utilizzando AWS CLI.

Creazione di una chat room

La creazione di una chat room con la AWS CLI è un'opzione avanzata e richiede prima il download e la configurazione della CLI sul computer. Per maggiori dettagli, consultare la Guida per l'utente dell'interfaccia a riga di comando di AWS.

1. Eseguire il comando `create-room` della chat e inviare un nome facoltativo:

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

2. Viene restituita una nuova chat room:

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

3. Annotare il campo `arn`. Si avrà bisogno di questo campo per creare un token client e connettersi a una chat room.

Impostazione di una configurazione di registrazione (facoltativo)

Come per la creazione di una chat room, l'impostazione di una configurazione di registrazione con AWS CLI è un'opzione avanzata e richiede prima il download e la configurazione della CLI sul computer. Per maggiori dettagli, consultare la Guida per l'utente dell'interfaccia a riga di comando di AWS.

1. Esegui il comando `create-logging-configuration` della chat e inserisci un nome facoltativo e una configurazione di destinazione che rimandi a un bucket Amazon S3 per nome. Questo bucket Amazon S3 deve già essere presente prima di creare la configurazione di registrazione. (Per i dettagli sulla creazione di un bucket Amazon S3, consulta la documentazione di Amazon S3).

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

2. Viene restituita una nuova configurazione di registrazione:

```
{
   "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. Annotare il campo `arn`. Questa operazione è necessaria per allegare la configurazione di registrazione alla chat room.

   a. Se stai creando una nuova chat room, esegui il comando `create-room` e passa la configurazione di registrazione `arn`:

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

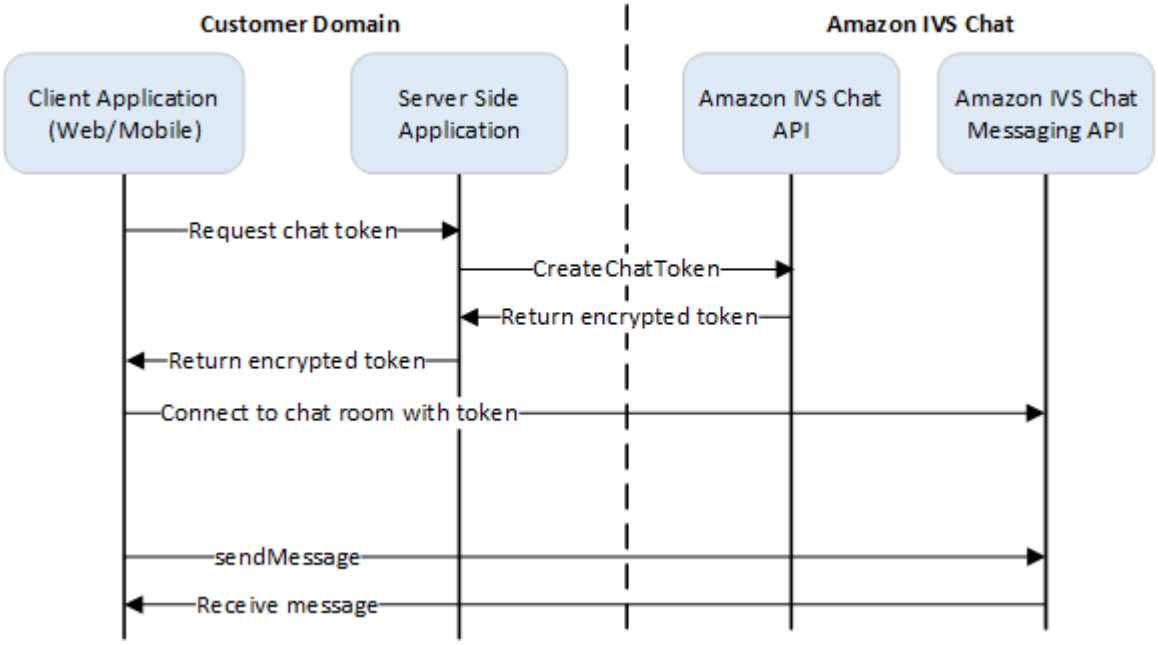
   b. Se stai aggiornando una chat room esistente, esegui il comando `update-room` e passa la configurazione di registrazione `arn`:

   ```
   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"
   ```

Passaggio 3: creare un token di chat

Affinché un partecipante alla chat possa connettersi a una stanza e iniziare a inviare e ricevere messaggi, è necessario creare un token di chat. I token di chat vengono utilizzati per autenticare e autorizzare i client delle chat.

Questo diagramma illustra il flusso di lavoro per la creazione di un token di chat IVS:



Come mostrato sopra, un'applicazione client richiede un token all'applicazione lato server e l'applicazione lato server chiama CreateChatToken utilizzando un SDK AWS o richieste firmate SigV4. Poiché le credenziali AWS vengono utilizzate per chiamare l'API, il token deve essere generato in un'applicazione sicura lato server, non nell'applicazione lato client.

Un'applicazione server di back-end che mostra come generare token è disponibile all'indirizzo Demo di back-end di Amazon IVS Chat.

Durata della sessione si riferisce alla durata per cui una sessione stabilita può rimanere attiva prima che venga terminata automaticamente. Ciò significa che la durata della sessione è la durata per cui il client può rimanere connesso alla chat room prima che sia necessario generare un nuovo token e stabilire una nuova connessione. Facoltativamente, è possibile specificare la durata della sessione durante la creazione del token.

Ogni token può essere utilizzato una sola volta per stabilire una connessione per un utente finale. Se una connessione viene interrotta, è necessario creare un nuovo token prima che la connessione possa essere ristabilita. Il token stesso è valido fino al timestamp di scadenza del token incluso nella risposta.

Quando un utente finale desidera connettersi a una chat room, il client deve chiedere all'applicazione server un token. L'applicazione server crea un token e lo ritrasmette al client. I token devono essere creati per gli utenti finali su richiesta.

Per creare un token di autenticazione per la chat, seguire le istruzioni riportate di seguito. Quando si crea un token di chat, utilizza i campi della richiesta per inviare i dati sull'utente finale della chat e sulle funzionalità di messaggistica dell'utente finale. Per i dettagli, consulta CreateChatToken nella Documentazione di riferimento dell'API Chat IVS.

Istruzioni per SDK AWS

La creazione di un token di chat con l'SDK AWS richiede prima il download e la configurazione dell'SDK sull'applicazione. Di seguito sono riportate le istruzioni per l'SDK AWS che utilizza JavaScript.

Importante: questo codice deve essere eseguito sul lato server e il relativo output passato al client.

Prerequisito: per utilizzare l'esempio di codice riportato di seguito, è necessario caricare l'SDK AWS JavaScript nell'applicazione. Per informazioni dettagliate, consultare Guida introduttiva all'SDK AWS per 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);
```

Istruzioni per la CLI

La creazione di un token di chat con la AWS CLI è un'opzione avanzata e richiede prima il download e la configurazione della CLI sul computer. Per maggiori dettagli, consultare la Guida per l'utente

dell'interfaccia a riga di comando di AWS. Nota: la generazione di token con l'AWS CLI è utile per scopi di test, ma per l'uso in produzione, si consiglia di generare token sul lato server con l'SDK AWS (vedere le istruzioni sopra).

1. Eseguire il comando `create-chat-token` insieme all'identificatore della stanza e all'ID utente per il client. Includere una delle funzionalità seguenti: `"SEND_MESSAGE"`, `"DELETE_MESSAGE"` ,`"DISCONNECT_USER"`. Facoltativamente, includere la durata della sessione (in minuti) e/o attributi personalizzati (metadati) su questa sessione di chat. Questi campi non sono mostrati di seguito.

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

2. Viene restituito un token di client:

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

3. Salvare questo token. Si avrà bisogno di questo token per connettersi alla chat room e inviare o ricevere messaggi. Sarà necessario generare un altro token di chat prima della fine della sessione (come indicato da `sessionExpirationTime`).

Passaggio 4: inviare e ricevere il primo messaggio

Usare il token di chat per connettersi a una chat room e inviare il primo messaggio. Il codice JavaScript di esempio è fornito di seguito. Sono disponibili anche gli SDK client; consulta SDK di Chat: guida per Android, SDK di Chat: guida per iOS e SDK di Chat: guida per JavaScript.

Servizio regionale: il codice di esempio riportato di seguito si riferisce alla "regione di scelta supportata". Amazon IVS Chat offre endpoint regionali utilizzabili per effettuare le richieste. Per l'Amazon IVS Chat Messaging API (API di Amazon IVS Chat Messaging), la sintassi generale di un endpoint regionale è:

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

Ad esempio, wss://edge.ivschat.us-west-2.amazonaws.com è l'endpoint della regione Stati Uniti occidentali (Oregon). Per un elenco delle regioni supportate, consultare le informazioni di Amazon IVS Chat nella pagina di Amazon IVS nella Documentazione di riferimento generale di AWS.

```
/*
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);
}
```

Congratulazioni, è tutto pronto! Ora si dispone di un'applicazione di chat semplice in grado di inviare o ricevere messaggi.

Passaggio 5: verificare i limiti delle Service Quotas (facoltativo)

Le chat room saranno scalabili insieme allo streaming live di Amazon IVS, per consentire a tutti i visualizzatori di partecipare alle conversazioni in chat. Tuttavia, tutti gli account Amazon IVS hanno dei limiti sul numero di partecipanti alla chat simultanei e sulla velocità di consegna dei messaggi.

Assicurarsi che i propri limiti siano adeguati e richiedere un aumento se necessario, specialmente se si pianifica un evento di streaming di grandi dimensioni. Per i dettagli, consulta le pagine Service Quotas (streaming a bassa latenza), Service Quotas (streaming in tempo reale) e Service Quotas (chat).

Registrazione Chat IVS

La funzione Registrazione di chat consente di registrare tutti i messaggi in una stanza in una delle tre posizioni standard: un bucket Amazon S3, File di log Amazon CloudWatch o Amazon Kinesis Data Firehose. Successivamente utilizza i log per l'analisi o per creare un riproduzione della chat che si collega a una sessione video dal vivo.

Abilitazione della registrazione delle chat per una stanza

La registrazione delle chat è un'opzione avanzata che può essere abilitata associando una configurazione di registrazione a una stanza. Una configurazione di registrazione è una risorsa che consente di specificare un tipo di posizione (bucket Amazon S3, File di log Amazon CloudWatch o Amazon Kinesis Data Firehose) in cui vengono registrati i messaggi di una stanza. Per informazioni dettagliate sulla creazione e la gestione delle configurazioni di registrazione, consulta Guida introduttiva ad Amazon IVS Chat e Documentazione di riferimento delle API di Amazon IVS Chat.

È possibile associare fino a tre configurazioni di registrazione a ciascuna stanza, quando si crea una nuova stanza (CreateRoom) o si aggiorna una stanza esistente (UpdateRoom). È possibile associare più stanze alla stessa configurazione di registrazione.

Quando almeno una configurazione di registrazione attiva è associata a una stanza, ogni richiesta di messaggistica inviata a quella stanza tramite l'API di messaggistica di Amazon IVS Chat viene registrata automaticamente nelle posizioni specificate. Questi sono i ritardi di propagazione medi (da quando viene inviata una richiesta di messaggistica a quando diventa disponibile nelle posizioni specificate):

- Bucket Amazon S3: 5 minuti
- File di log Amazon CloudWatch o Amazon Kinesis Data Firehose: 10 secondi

Contenuto del messaggio

Formato

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

```
   "payload": { "string": "string" }
}
```

Campi

Campo	Descrizione
`event_timestamp`	Timestamp UTC di quando il messaggio è stato ricevuto da Amazon IVS Chat.
`payload`	Il payload JSON del messaggio (sottoscrizione) o dell'evento (sottoscrizione) che i clienti riceveranno dal servizio Amazon IVS Chat.
`type`	Tipo di messaggio della chat. • Valori validi: `MESSAGE` | `EVENT`
`version`	La versione del formato del contenuto del messaggio.

Bucket Amazon S3

Formato

I log dei messaggi sono organizzati e archiviati con il seguente prefisso S3 e formato di file:

```
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
```

Campi

Campo	Descrizione
`<account_id>`	ID dell'account AWS da cui viene creata la stanza.
`<hash>`	Un valore hash generato dal sistema per garantire l'unicità.

Campo	Descrizione
`<region>`	La regione di servizio AWS in cui è stata creata la stanza.
`<resource_id>`	L'ID della risorsa parte dell'ARN della stanza.
`<version>`	La versione del formato del contenuto del messaggio.
`<year> / <month> / <day> / <hours> / <minute>`	Timestamp UTC di quando il messaggio è stato ricevuto da Amazon IVS Chat.

Esempio

```
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

Formato

I log dei messaggi sono organizzati e archiviati con il seguente formato di nome del flusso di log:

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

Campi

Campo	Descrizione
`<resource_id>`	L'ID della risorsa parte dell'ARN della stanza.
`<version>`	La versione del formato del contenuto del messaggio.

Esempio

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

Amazon Kinesis Data Firehose

I log dei messaggi vengono inviati al flusso di consegna come dati di streaming in tempo reale a destinazioni come Amazon Redshift, il servizio OpenSearch di Amazon, Splunk e qualsiasi endpoint HTTP personalizzato o endpoint HTTP di proprietà di provider di servizi di terze parti supportati. Per ulteriori informazioni, consulta Cos'è Amazon Kinesis Data Firehose?.

Vincoli

- È necessario essere il proprietario della posizione di registrazione in cui verranno archiviati i messaggi.
- La stanza, la configurazione di registrazione e la posizione di registrazione devono trovarsi nella stessa regione AWS.
- Per la registrazione delle chat sono disponibili solo le configurazioni di registrazione attive.
- È possibile eliminare una configurazione di registrazione che non è più associata ad alcuna stanza.

La registrazione di messaggi in una posizione di proprietà dell'utente richiede l'autorizzazione con le credenziali AWS. Per fornire a IVS Chat l'accesso richiesto, al momento della creazione della configurazione di registrazione sono generati automaticamente una policy di risorse (per un bucket Amazon S3 o File di log CloudWatch) o un ruolo collegato ai servizi (SLR) di AWS IAM (per Amazon Kinesis Data Firehose). Prestare attenzione a qualsiasi modifica al ruolo o alle policy, poiché ciò può influire sull'autorizzazione per la registrazione delle chat.

Monitoraggio degli errori con Amazon CloudWatch

Puoi monitorare gli errori che si verificano nella registrazione dei messaggi con Amazon CloudWatch e creare allarmi o pannelli di controllo per indicare o rispondere alle modifiche di errori specifici.

Esistono diversi tipi di errori. Per maggiori informazioni, consulta la pagina Monitoraggio di Chat Amazon IVS.

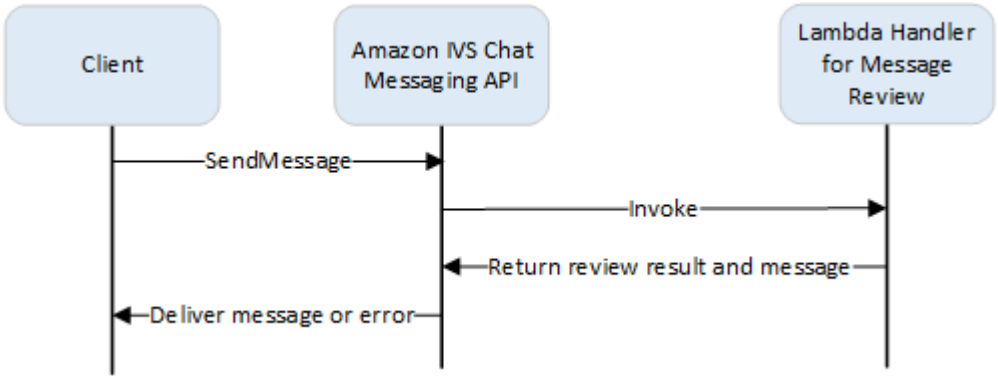
Gestore di revisione dei messaggi di Chat IVS

Un gestore di revisione dei messaggi consente di revisionare e/o modificare i messaggi prima che vengano consegnati in una stanza. Quando un gestore di revisione dei messaggi è associato a una stanza, viene richiamato per ogni richiesta SendMessage in quella stanza. Il gestore applica la logica aziendale dell'applicazione e stabilisce se consentire, negare o modificare un messaggio. Amazon IVS Chat supporta le funzioni di AWS Lambda come gestori.

Creazione di una funzione Lambda

Prima di configurare un gestore di revisione dei messaggi per una stanza, è necessario creare una funzione lambda con una policy IAM basata sulle risorse. La funzione lambda deve trovarsi nello stesso account e regione AWS della stanza con cui verrà utilizzata la funzione. La policy basata sulle risorse consente ad Amazon IVS Chat di richiamare la funzione lambda. Per le istruzioni, consulta la pagina Policy basata sulle risorse per Chat Amazon IVS.

Flusso di lavoro



Sintassi della richiesta

Quando un client invia un messaggio, Amazon IVS Chat richiama la funzione lambda con un payload JSON:

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

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

Corpo della richiesta

Campo	Descrizione
`Attributes`	Gli attributi associati al messaggio.
`Content`	Il contenuto originale del messaggio.
`MessageId`	L'ID del messaggio. Generato da IVS Chat.
`RoomArn`	L'ARN della stanza in cui vengono inviati i messaggi.
`Sender`	Informazioni sul mittente. Questo oggetto ha diversi campi: • `Attributes` : i metadati sul mittente stabiliti durante l'autenticazione. Questo valore può essere utilizzato per fornire al client ulteriori informazioni sul mittente, ad esempio URL dell'avatar, badge, carattere e colore. • `UserId`: un identificatore specificato dall'applicazione del visualizzatore (utente finale) che ha inviato questo messaggio. Questo valore può essere utilizzato dall'applicazione client per fare riferimento all'utente nell'API di messaggistica o nei domini dell'applicazione. • `Ip`: l'indirizzo IP del client che invia il messaggio.

Sintassi della risposta

La funzione lambda del gestore deve restituire una risposta JSON con la seguente sintassi. Le risposte che non corrispondono alla sintassi sottostante o soddisfano i vincoli di campo non sono valide. In questo caso, il messaggio è consentito o negato a seconda del valore `FallbackResult` specificato nel gestore di revisione dei messaggi; consulta MessageReviewHandler in Amazon IVS Chat API Reference (Documentazione di riferimento dell'API Amazon IVS Chat).

```
{
```

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

Campi di risposta

Campo	Descrizione
`Attributes`	Attributi associati al messaggio restituito dalla funzione lambda. Se `ReviewResult` è `DENY`, una `Reason` può essere fornita in `Attributes`; ad esempio: `"Attributes": {"Reason": "denied for moderation` In questo caso, il client mittente riceve un errore WebSocket 406 con il motivo nel messaggio di errore. Consulta Errori WebSocket nella Documentazione di riferimento sull'API Amazon IVS Chat Messaging. • Vincoli di dimensioni: massimo 1 KB • Campo obbligatorio: no
`Content`	Contenuto del messaggio restituito dalla funzione Lambda. Potrebbe essere modificato o originale a seconda della logica aziendale. • Limitazioni di lunghezza: lunghezza minima pari a 1. Lunghezza massima del parametro `MaximumMessageLength` che hai definito quando hai creato/aggiornato la stanza. Per ulteriori informazioni, consulta la Documentazione di riferimento sull'API Amazon IVS Chat. Questo vale solo quando `ReviewResult` è `ALLOW`. • Campo obbligatorio: sì
`ReviewResult`	Il risultato dell'elaborazione delle revisioni su come gestire il messaggio. Se consentito, il messaggio viene consegnato a tutti gli utenti connessi alla stanza. Se negato, il messaggio non viene consegnato a nessun utente. • Valori validi: `ALLOW` | `DENY` • Campo obbligatorio: sì

Codice di esempio

Di seguito è riportato un esempio di gestore lambda in Go. Modifica il contenuto del messaggio, mantiene invariati gli attributi del messaggio e consente il messaggio.

```
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
}
```

Associazione e annullamento dell'associazione di un gestore con una stanza

Una volta configurato e implementato il gestore lambda, utilizza l'API Amazon IVS Chat:

- Per associare il gestore a una stanza, chiama CreateRoom o UpdateRoom e specifica il gestore.
- Per annullare l'associazione del gestore da una stanza, chiama UpdateRoom con un valore vuoto per `MessageReviewHandler.Uri`.

Monitoraggio degli errori con Amazon CloudWatch

Puoi monitorare gli errori che si verificano nella revisione dei messaggi con Amazon CloudWatch e creare allarmi o dashboard per indicare o rispondere alle modifiche di errori specifici. Se si verifica un errore, il messaggio viene consentito o negato a seconda del valore di `FallbackResult` specificato quando associ il gestore a una stanza; consulta MessageReviewHandler in Amazon IVS Chat API Reference (Documentazione di riferimento dell'API Amazon IVS Chat).

Esistono diversi tipi di errori:

- `InvocationErrors` si verificano quando Amazon IVS Chat non può richiamare un gestore.
- `ResponseValidationErrors` si verificano quando un gestore restituisce una risposta non valida.
- `Errors` di AWS Lambda si verificano quando un gestore lambda restituisce un errore di funzione quando è stato richiamato.

Per ulteriori informazioni sugli errori di invocazione e sugli errori di convalida delle risposte (emessi da Chat Amazon IVS), consulta la pagina Monitoraggio di Chat Amazon IVS. Per ulteriori informazioni sugli errori di AWS Lambda, consulta Utilizzo dei parametri Lambda.

Monitoraggio di Chat Amazon IVS

È possibile monitorare le risorse di Amazon Interactive Video Service (IVS) tramite Amazon CloudWatch. CloudWatch raccoglie ed elabora i dati non elaborati da Chat Amazon IVS trasformandoli in parametri leggibili quasi in tempo reale. Queste statistiche vengono conservate per un periodo di 15 mesi, per permettere l'accesso alle informazioni della cronologia e per offrire una prospettiva migliore sulle prestazioni del servizio o dell'applicazione Web. Esiste la possibilità di impostare allarmi che controllano determinate soglie e inviare notifiche o intraprendere azioni quando queste soglie vengono raggiunte. Per informazioni dettagliate, consultare la Guida per l'utente di CloudWatch.

Accesso ai parametri di CloudWatch

Amazon CloudWatch raccoglie ed elabora i dati non elaborati da Chat Amazon IVS trasformandoli in parametri leggibili quasi in tempo reale. Queste statistiche vengono conservate per un periodo di 15 mesi, per permettere l'accesso alle informazioni della cronologia e per offrire una prospettiva migliore sulle prestazioni del servizio o dell'applicazione Web. Esiste la possibilità di impostare allarmi che controllano determinate soglie e inviare notifiche o intraprendere azioni quando queste soglie vengono raggiunte. Per informazioni dettagliate, consultare la Guida per l'utente di CloudWatch.

Tenere presente che i parametri di CloudWatch vengono registrati nel tempo. La risoluzione diminuisce efficacemente con l'età dei parametri. Lo schema è il seguente:

- I parametri di 60 secondi sono disponibili per 15 giorni.
- I parametri di 5 minuti sono disponibili per 63 giorni.
- I parametri di 1 ora sono disponibili per 455 giorni (15 mesi).

Per informazioni aggiornate sulla conservazione dei dati, cercare "periodo di conservazione" in Domande frequenti su Amazon CloudWatch.

Istruzioni per la console CloudWatch

1. Apri la console CloudWatch all'indirizzo https://console.aws.amazon.com/cloudwatch/.
2. Nella navigazione laterale, espandere il menu a discesa Metrics (Parametri), quindi selezionare All metrics (Tutti i parametri).

3. Nella scheda Sfoglia, utilizzando il menu a discesa senza etichetta sulla sinistra, seleziona la propria regione "di origine", ovvero dove sono stati creati i canali. Per ulteriori informazioni sulle Regioni, consultare Soluzione globale, controllo regionale. Per un elenco delle Regioni supportate, consultare la pagina di Amazon IVS nei Riferimenti generali di AWS.
4. Nella parte inferiore della scheda Sfoglia, seleziona lo spazio dei nomi IVSChat.
5. Esegui una di queste operazioni:
   a. Nella barra di ricerca digitare l'ID della risorsa (parte dell'ARN, `arn:::ivschat:room/<resource id>`).

      Quindi seleziona IVSChat.
   b. Se IVSChat viene visualizzato come servizio selezionabile in Spazi dei nomi AWS, selezionalo. Verrà specificato se si utilizza Chat Amazon IVS e se i rispettivi parametri vengono inviati ad Amazon CloudWatch. Se IVSChat non è presente nell'elenco, allora significa che non si dispone di parametri di Chat Amazon IVS.

      Selezione ora il raggruppamento di dimensioni desiderato. Le dimensioni disponibili sono elencate nei Parametri di CloudWatch qui sotto.
6. Seleziona i parametri da aggiungere al grafico. I parametri disponibili sono elencati nei Parametri di CloudWatch qui sotto.

È inoltre possibile accedere al grafico CloudWatch della sessione di streaming dalla pagina dei dettagli della sessione di chat selezionando il pulsante Visualizza in CloudWatch.

Istruzioni per la CLI

È possibile accedere ai parametri anche utilizzando l'interfaccia a riga di comando (CLI) di AWS. Ciò richiede il download e la configurazione della CLI sul computer. Per maggiori dettagli, consultare la Guida per l'utente dell'interfaccia a riga di comando di AWS.

Quindi, per accedere ai parametri della chat a bassa latenza di Amazon IVS utilizzando AWS CLI:

- Al prompt dei comandi, esegui:

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

Per ulteriori informazioni, consulta Utilizzo di parametri di Amazon CloudWatch nella Guida per l'utente di Amazon CloudWatch.

Parametri di CloudWatch: Chat IVS

Amazon IVS Chat fornisce i parametri riportati di seguito nello spazio dei nomi AWS/IVSChat.

Parametro	Dimensione	Descrizione
`ConcurrentChatConnections`	Nessuno	Il numero totale di connessioni simultanee in una chat room (numero massimo riportato al minuto). Ciò è utile per capire quando i clienti si avvicinano al limite per le connessioni di chat simultanee in una regione. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)
`Deliveries`	Azione	Il numero di consegne di richieste di messaggistica fatte da un tipo di operazione specifica per le connessioni di chat in tutte le stanze di una regione. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)
`InvocationErrors`	`Uri`	Il numero di errori di invocazione di uno specifico gestore di revisione dei messaggi in tutte le stanze di una regione. Un errore di invocazione si verifica quando non è possibile richiamare il gestore di revisione dei messaggi. Gli errori di invocazione si verificano quando Amazon IVS Chat non può richiamare un gestore. Questo può accadere se il gestore associato a una stanza non esiste più o va in

Parametro	Dimensione	Descrizione
		timeout, oppure se la policy delle risorse non consente al servizio di richiamarlo. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)
`LogDestinationAccessDeniedError`	`LoggingConfiguration`	Il numero di errori di accesso negato di una destinazione di log in tutte le stanze di una regione. Questi errori si verificano quando Amazon IVS Chat non può accedere alla risorsa di destinazione specificata nella configurazione di registrazione. Ciò può accadere se la policy delle risorse di destinazione non consente al servizio di inserire record. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)

Parametro	Dimensione	Descrizione
`LogDestinationErrors`	`LoggingConfiguration`	Il numero di tutti gli errori di una destinazione di log in tutte le stanze di una regione. Si tratta di un parametro aggregato che include tutti i tipi di errori che si verificano quando Amazon IVS Chat non riesce a consegnare i log alla risorsa di destinazione specificata nella configurazione di registrazione. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)
`LogDestinationResourceNotFoundErrors`	`LoggingConfiguration`	Il numero di errori "risorsa non trovata" di una destinazione di log in tutte le stanze di una regione. Questi errori si verificano quando Amazon IVS Chat non può consegnare i log a una risorsa di destinazione specificata nella configurazione di registrazione in quanto la risorsa non esiste. Ciò può accadere se la risorsa di destinazione associata a una configurazione di registrazione non esiste più. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)

Parametro	Dimensione	Descrizione
`Messaging Deliveries`	Nessuno	Il numero di consegne di richieste di messaggistica per le connessioni di chat in tutte le stanze di una regione. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)
`Messaging Requests`	Nessuno	Il numero di richieste di messaggistica effettuate in tutte le stanze di una regione. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)
`Requests`	Azione	Il numero di richieste effettuate di un tipo di azione specifico in tutte le stanze di una regione. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)

Parametro	Dimensione	Descrizione
`ResponseValidationErrors`	`Uri`	Il numero di errori di convalida della risposta di uno specifico gestore di revisione dei messaggi in tutte le stanze di una regione. Un errore di convalida della risposta si verifica quando la risposta del gestore di revisione dei messaggi non è valida. Ciò potrebbe significare che non è stato possibile analizzare la risposta o che la risposta non ha superato i controlli di convalida; ad esempio, un risultato di revisione non valido o valori di risposta troppo lunghi. Unità: numero Statistiche valide: Sum (Somma), Average (Media), Maximum (Massimo), Minimum (Minimo)

SDK di messaggistica per client di chat IVS

L'SDK di Amazon Interactive Video Services (IVS) Chat Client Messaging è rivolto agli sviluppatori che compilano applicazioni con Amazon IVS. Questo SDK è progettato per sfruttare l'architettura di Amazon IVS e consultarne gli aggiornamenti, così come Amazon IVS Chat. Essendo un SDK nativo, è progettato per ridurre al minimo l'impatto sulle prestazioni dell'applicazione e dei dispositivi utilizzati dagli utenti per accedere all'applicazione.

Requisiti della piattaforma

Browser desktop

Browser	Versioni supportate
Chrome	Due versioni principali (versione corrente e precedente più recente)
Edge	Due versioni principali (versione corrente e precedente più recente)
Firefox	Due versioni principali (versione corrente e precedente più recente)
Opera	Due versioni principali (versione corrente e precedente più recente)
Safari	Due versioni principali (versione corrente e precedente più recente)

Browser per dispositivi mobili

Browser	Versioni supportate
Chrome per Android	Due versioni principali (versione corrente e precedente più recente)
Firefox per Android	Due versioni principali (versione corrente e precedente più recente)
Opera per Android	Due versioni principali (versione corrente e precedente più recente)

Browser	Versioni supportate
WebView Android	Due versioni principali (versione corrente e precedente più recente)
Samsung Internet	Due versioni principali (versione corrente e precedente più recente)
Safari per iOS	Due versioni principali (versione corrente e precedente più recente)

Piattaforme native

Piattaforma	Versioni supportate
Android	5.0 e versioni successive
iOS	13.0 e versioni successive

Supporto

Se si verifica un errore o un altro problema con la chat room, determinare l'identificatore univoco della stanza tramite l'API IVS Chat (consultare ListRooms).

Condividi questo identificatore della chat room con il Supporto AWS. Grazie a questo identificatore, si possono ottenere informazioni utili per risolvere il problema.

Nota: consulta le Note di rilascio di Chat Amazon IVS per conoscere le versioni disponibili e i problemi risolti. Se necessario, prima di contattare il supporto, aggiornare la versione dell'SDK e verificare se il problema è stato risolto.

Controllo delle versioni

Gli SDK di messaggistica client di Amazon IVS Chat utilizzano il controllo semantico delle versioni.

Per questa discussione, supponiamo che:

- La versione più recente sia la 4.1.3.

- L'ultima versione della versione principale precedente sia 3.2.4.
- La versione più recente della versione 1.x sia la 1.5.6.

Le nuove funzionalità compatibili con le versioni precedenti vengono aggiunte come versioni secondarie dell'ultima versione. In questo caso, il set successivo di nuove funzionalità verrà aggiunto come versione 4.2.0.

Le correzioni di bug minori compatibili con le versioni precedenti vengono aggiunte come versioni di patch dell'ultima versione. Nel nostro caso, il set di correzioni minori di bug successivo sarà aggiunto come versione 4.1.4.

Le correzioni di bug principali compatibili con le versioni precedenti sono gestite in modo diverso, ovvero vengono aggiunte alle diverse versioni:

- Rilascio della patch dell'ultima versione. Nel nostro caso, questa è la versione 4.1.4.
- Rilascio della patch della versione secondaria precedente. Nel nostro caso, questa è la versione 3.2.5.
- Rilascio di patch dell'ultima versione 1.x. Nel nostro caso, questa è la versione 1.5.7.

Le correzioni di bug principali sono definite dal team di prodotti Amazon IVS. Esempi tipici sono gli aggiornamenti critici della sicurezza e alcune altre correzioni necessarie per i clienti.

Nota: negli esempi precedenti, le versioni rilasciate vengono incrementate senza saltare alcun numero (ad esempio, da 4.1.3 a 4.1.4). In realtà, uno o più numeri di patch possono rimanere interni e non essere rilasciati, quindi la versione rilasciata potrebbe aumentare da 4.1.3 a, ad esempio, 4.1.6.

Inoltre, la versione 1.x sarà supportata fino alla fine del 2023 o fino al rilascio della versione 3.x, a seconda di quale evento si verificherà per primo.

Amazon IVS Chat API (API di Amazon IVS Chat)

Sul lato server (non gestito dagli SDK), esistono due API, ognuna con le proprie competenze:

- Piano dati: l'API IVS Chat Messaging è un'API WebSocket progettata per essere utilizzata da applicazioni front-end (iOS, Android, macOS, ecc.) guidate da uno schema di autenticazione basato su token. Utilizzando un token di chat generato in precedenza, è possibile connettersi a chat room già esistenti utilizzando tale API.

Gli SDK di Amazon IVS Chat Client Messaging riguardano solo il piano dati. Gli SDK presuppongono che si stia già generando token di chat tramite il proprio back-end. Si presume che il recupero di questi token sia gestito dall'applicazione front-end, non dagli SDK.

- Piano di controllo (control-plane): l'API del piano di controllo (control-plane) della IVS Chat fornisce un'interfaccia delle proprie applicazioni di back-end per gestire e creare chat room e gli utenti che vi partecipano. Considerarlo il pannello di amministrazione dell'esperienza di chat dell'app, gestito dal back-end. Esistono endpoint del piano di controllo (control-plane) responsabili della creazione del token di chat che il piano dati necessita per autenticarsi in una chat room.

  Importante: gli SDK di IVS Chat Client Messaging non chiamano nessun endpoint del piano di controllo (control-plane). È necessario configurare il back-end per creare i propri token di chat. L'applicazione front-end deve comunicare con il back-end per recuperare tale token di chat.

SDK di messaggistica per client di Chat IVS: guida per Android

L'SDK per Android di Amazon Interactive Video (IVS) Chat Client Messaging fornisce interfacce che consentono di incorporare facilmente l'API IVS Chat Messaging su piattaforme che utilizzano Android.

Il pacchetto `com.amazonaws:ivs-chat-messaging` implementa l'interfaccia descritta in questo documento.

Ultima versione dell'SDK di messaggistica per client di chat IVS per Android: 1.1.0 (Note di rilascio)

Documentazione di riferimento: per informazioni sui metodi più importanti disponibili nell'SDK per Android di Amazon IVS Chat Client Messaging consultare la documentazione di riferimento all'indirizzo https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/

Codice di esempio: consultare l'archivio di esempio per Android su GitHub: https://github.com/aws-samples/amazon-ivs-chat-for-android-demo

Requisiti della piattaforma: per lo sviluppo è necessario Android 5.0 (livello API 21) o versioni successive.

Guida introduttiva all'SDK di messaggistica per client di chat IVS

Prima di iniziare, acquisire familiarità con Nozioni di base su Amazon IVS Chat.

Aggiungere il pacchetto

Aggiungere `com.amazonaws:ivs-chat-messaging` alle proprie dipendenze `build.gradle`:

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

Aggiungere le regole ProGuard

Aggiungere le seguenti voci al file delle regole di R8/ProGuard (`proguard-rules.pro`):

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

Impostazione del back-end

Questa integrazione richiede endpoint sul server che comunichino con l'API Amazon IVS. Utilizzare le librerie AWS ufficiali per accedere all'API Amazon IVS dal proprio server. Queste sono accessibili in diverse lingue dai pacchetti pubblici, ad esempio node.js e Java.

Poi creare un endpoint del server che comunichi con Amazon IVS Chat API (API di Amazon IVS Chat) creando un token.

Stabilire una connessione al server

Creare un metodo che richieda `ChatTokenCallback` come parametro e recuperi un token di chat dal proprio back-end. Passare tale token al metodo `onSuccess` del callback. In caso di errore, passare l'eccezione al metodo `onError` del callback. Questo è necessario per istanziare la principale entità della `ChatRoom` nella fase successiva.

Di seguito è disponibile un esempio di codice che implementa quanto sopra usando una chiamata di `Retrofit`.

```
// ...

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

Utilizzo dell'SDK di messaggistica per client di chat IVS su Android

Questo documento illustra i passaggi necessari per l'utilizzo dell'SDK di messaggistica per client di chat Amazon IVS su Android.

Inizializzare un'istanza di chat room

Creare un'istanza della classe `ChatRoom`. Ciò richiede il passaggio di `regionOrUrl`, che in genere è la Regione AWS in cui è ospitata la chat room, e il `tokenProvider` che è il metodo di recupero dei token creato nella fase precedente.

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

Quindi, creare un oggetto ascoltatore che implementerà i gestori per gli eventi relativi alla chat e assegnarlo alla proprietà `room.listener`:

```
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

// ...
```

L'ultimo passaggio dell'inizializzazione di base consiste nel connettersi alla stanza specifica stabilendo una connessione WebSocket. Per farlo, chiamare il metodo `connect()` all'interno dell'istanza della stanza. Consigliamo di farlo nel metodo del ciclo di vita `onResume()` per assicurarsi che mantenga la connessione se l'app riprende in background.

```
room.connect()
```

L'SDK cercherà di stabilire una connessione a una chat room codificata nel token di chat ricevuto dal server. Se fallisce, tenterà di ricollegarsi tante volte quante quelle specificate nell'istanza della stanza.

Esecuzione di azioni in una chat room

La classe `ChatRoom` dispone di azioni per inviare ed eliminare messaggi e disconnettere altri utenti. Queste azioni accettano un parametro di callback opzionale che consente di ricevere notifiche di conferma o rifiuto della richiesta.

Invio di un messaggio

Per questa richiesta è necessario disporre della funzionalità `SEND_MESSAGE` codificata nel proprio token di chat.

Per attivare una richiesta di invio di messaggi:

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

Per ottenere una conferma/rifiuto della richiesta, fornire un callback come secondo parametro:

```
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.
   }
})
```

Eliminazione di un messaggio

Per questa richiesta è necessario disporre della funzionalità DELETE_MESSAGE codificata nel proprio token di chat.

Per attivare una richiesta di eliminazione di un messaggio:

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

Per ottenere una conferma/rifiuto della richiesta, fornire un callback come secondo parametro:

```
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.
   }
})
```

Disconnessione di un altro utente

Per questa richiesta è necessario disporre della funzionalità `DISCONNECT_USER` codificata nel proprio token di chat.

Per disconnettere un altro utente a scopo di moderazione:

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

Per ottenere una conferma/rifiuto della richiesta, fornire un callback come secondo parametro:

```
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.
   }
})
```

Disconnessione da una chat room

Per chiudere la connessione alla chat room, chiamare il metodo `disconnect()` sull'istanza della stanza:

```
room.disconnect()
```

Poiché la connessione WebSocket smetterà di funzionare dopo un breve periodo di tempo, quando l'applicazione è in background, si consiglia di connettersi/disconnettersi manualmente durante la

transizione da/verso uno stato di background. A tale scopo, abbinare la chiamata `room.connect()` nel metodo del ciclo di vita `onResume()`, su Android `Activity` o `Fragment`, con la chiamata `room.disconnect()` nel metodo del ciclo di vita `onPause()`.

SDK per la messaggistica per client di Chat IVS - Tutorial per Android, parte 1: chat room

Questo è il primo di un tutorial a due parti. Scoprirai gli elementi essenziali per utilizzare l'SDK di messaggistica di chat Amazon IVS creando un'app Android funzionale e completa con il linguaggio di programmazione Kotlin. Chiameremo l'app Chatterbox.

Prima di avviare il modulo, dedica qualche minuto a familiarizzare con i prerequisiti, i concetti chiave alla base dei token di chat e il server di back-end necessario per creare le chat room.

Questi tutorial sono creati per sviluppatori Android esperti che non hanno mai utilizzato l'SDK per la messaggistica di chat IVS. Dovrai essere a tuo agio con il linguaggio di programmazione Kotlin e con la creazione di interfacce utente sulla piattaforma Android.

Questa prima parte del tutorial è suddivisa in diverse sezioni:

1. the section called “Configurazione di un server di autenticazione/autorizzazione locale”
2. the section called “Creazione di un progetto Chatterbox”
3. the section called “Connessione a una chat room e osservazione degli aggiornamenti della connessione”
4. the section called “Creazione di un provider di token”
5. the section called “Fasi successive”

Per la documentazione completa dell'SDK, inizia con l'SDK di messaggistica per client di chat Amazon IVS (qui nella Guida per l'utente di Chat Amazon IVS) e la Documentazione di riferimento dell'SDK di messaggistica per client di chat per Android su GitHub.

Prerequisiti

- Devi avere dimestichezza con Kotlin e con la creazione di applicazioni sulla piattaforma Android. Se non hai dimestichezza con la creazione di applicazioni per Android, scopri le nozioni di base nella guida Creazione della prima app per gli sviluppatori Android.
- Leggi e comprendi attentamente Nozioni di base su Chat IVS.

- Crea un utente IAM AWS con le capacità `CreateChatToken` e `CreateRoom` definite in una policy IAM esistente. Consultare Nozioni di base su Chat IVS.
- Assicurati che la chiavi di accesso segrete di questo utente siano archiviata in un file di credenziali AWS. Per istruzioni, consulta la Guida per l'utente di AWS CLI (in particolare la sezione Configurazione e impostazioni del file delle credenziali).
- Crea una chat room e salva il relativo ARN. Consultare Nozioni di base su Chat IVS. (Se non salvi l'ARN, potrai cercarlo in un secondo momento con la console o l'API di Chat.)

Configurazione di un server di autenticazione/autorizzazione locale

Il tuo server di back-end è responsabile sia della creazione di chat room sia della generazione dei token di chat necessari all'SDK di chat IVS per Android per autenticare e autorizzare i client ad accedere alle tue chat room.

Consulta Creazione di un token di chat nella Guida introduttiva ad Amazon IVS Chat. Come mostrato nel diagramma di flusso, il codice lato server è responsabile della creazione di un token di chat. Ciò significa che l'app deve fornire i propri mezzi per generare un token di chat richiedendone uno dall'applicazione lato server.

Utilizziamo il framework Ktor per creare un server locale live che gestisca la creazione di token di chat utilizzando l'ambiente AWS locale.

A questo punto, ci aspettiamo che le tue credenziali AWS siano configurate correttamente. Per istruzioni dettagliate, consulta la pagina Configurazione delle credenziali e della regione AWS per lo sviluppo.

Crea una nuova directory e chiamala `chatterbox` e al suo interno creane un'altra, denominata `auth-server`.

La cartella sul server avrà la seguente struttura:

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

```
        - application.conf
        - logback.xml
   - build.gradle.kts
```

Nota: puoi copiare/incollare direttamente il codice qui nei file di riferimento.

Successivamente, aggiungiamo tutte le dipendenze e i plugin necessari per il funzionamento del server di autenticazione:

Script Kotlin:

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

Ora dobbiamo configurare la funzionalità di registrazione per il server di autenticazione. Per ulteriori informazioni, consulta la sezione Configurazione del logger.

XML:

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

<configuration>
```

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

Il server Ktor richiede delle impostazioni di configurazione, che carica automaticamente dal file `application.*` nella directory `resources`, quindi aggiungiamo anche quelle. Per ulteriori informazioni, consulta la sezione Configurazione in un file.

HOCON:

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

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

Infine, implementiamo il server:

Kotlin:

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

package com.chatterbox.authserver

import io.ktor.http.*
import io.ktor.serialization.kotlinx.json.*
import io.ktor.server.application.*
import io.ktor.server.plugins.contentnegotiation.*
import io.ktor.server.request.*
```

```
import io.ktor.server.response.*
import io.ktor.server.routing.*
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import software.amazon.awssdk.services.ivschat.IvschatClient
import software.amazon.awssdk.services.ivschat.model.CreateChatTokenRequest

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

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

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

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

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

Creazione di un progetto Chatterbox

Per creare un progetto Android, installa e apri Android Studio.

Segui i passaggi elencati nella guida Creazione di un progetto di Android.

- In Scegli il tuo tipo di progetto, scegli il modello di progetto Attività vuota per la nostra app Chatterbox.
- In Configura il tuo progetto, scegli i seguenti valori per i campi di configurazione:
  - Nome: My App
  - Nome del pacchetto: com.chatterbox.myapp
  - Percorso di salvataggio: fai riferimento alla cartella `chatterbox` creata nel passaggio precedente
  - Lingua: Kotlin
  - Livello minimo API: API 21: Android 5.0 (Lollipop)

Dopo aver specificato correttamente tutti i parametri di configurazione, la struttura dei file all'interno della cartella `chatterbox` dovrebbe essere la seguente:

```
- 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
```

Ora che abbiamo un progetto Android funzionante, possiamo aggiungere com.amazonaws:ivs-chat-messaging alle nostre dipendenze `build.gradle`. Per ulteriori informazioni sul toolkit di compilazione Gradle, consulta la pagina Configurazione della compilazione.

Nota: nella parte superiore di ogni frammento di codice, è riportato il percorso del file in cui dovresti apportare modifiche al tuo progetto. Il percorso fa riferimento alla cartella del progetto.

Nel codice seguente, sostituisci *<version>* con il numero di versione corrente dell'SDK di chat per Android (ad esempio, 1.0.0).

Kotlin:

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

plugins {
// ...
}

android {
// ...
}

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

Dopo aver aggiunto la nuova dipendenza, esegui Sincronizzazione dei progetti con i file Gradle in Android Studio per sincronizzare il progetto con la nuova dipendenza. Per ulteriori informazioni, consulta la pagina Aggiunta di dipendenze della compilazione.

Per eseguire comodamente il server di autenticazione che abbiamo creato nella sezione precedente dalla root del progetto, lo includiamo come nuovo modulo in `settings.gradle`. Per ulteriori informazioni, consulta la pagina Strutturazione e costruzione di un componente software con Gradle.

Script Kotlin:

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

```
// ...

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

D'ora in avanti, dato che `auth-server` è incluso nel progetto Android, puoi eseguire il server di autenticazione con il seguente comando dalla root del progetto:

Shell (interprete di comandi):

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

Connessione a una chat room e osservazione degli aggiornamenti della connessione

Per aprire una connessione alla chat room, utilizziamo il callback del ciclo di vita dell'attività onCreate(), che si attiva quando l'attività viene creata per la prima volta. Il costruttore di ChatRoom ci richiede di fornire `region` e `tokenProvider` per inizializzare una connessione alla chat room.

Nota: la funzione `fetchChatToken` nel frammento di seguito verrà implementata nella sezione successiva.

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)
    }

// ...
}
```

Visualizzare i cambiamenti nello stato della connessione di una chat room e reagire a essi sono parti essenziali della creazione di un'app di chat come `chatterbox`. Prima di poter iniziare a interagire con la chat room, dobbiamo iscriverci agli eventi sullo stato di connessione della chat room per ricevere aggiornamenti.

ChatRoom si aspetta che colleghiamo un'implementazione dell'interfaccia ChatRoomListener per presentare eventi del ciclo di vita. Per ora, le funzioni dell'ascoltatore registreranno solo i messaggi di conferma, quando richiamati:

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")
        }
    }
}
```

Ora che abbiamo implementato il `ChatRoomListener`, lo colleghiamo alla nostra istanza di chat room:

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 {
// ...
```

```
}
```

Successivamente, dobbiamo fornire la capacità di leggere lo stato della connessione della chat room. Lo manterremo nella proprietà `MainActivity.kt` e lo inizializzeremo allo stato predefinito DISCONNECTED (disconnesso) per le chat room (consulta la sezione `ChatRoom state` nella Documentazione di riferimento dell'SDK di chat IVS per Android). Per poter mantenere aggiornato lo stato locale, dobbiamo implementare una funzione di aggiornamento dello stato, che chiameremo `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")
         }
      }
   }
}
```

Successivamente, integriamo la funzione di aggiornamento dello stato con la proprietà 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)
         }
      }
   }
}
```

Ora che abbiamo la possibilità di salvare, ascoltare e reagire agli aggiornamenti dello stato di ChatRoom, è il momento di inizializzare una connessione:

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

Creazione di un provider di token

È il momento di creare una funzione responsabile della creazione e della gestione dei token di chat nell'applicazione. In questo esempio utilizziamo il client HTTP Retrofit per Android.

Prima di poter inviare traffico di rete, dobbiamo impostare una configurazione di sicurezza di rete per Android. Per ulteriori informazioni, consulta la pagina Configurazione della sicurezza di rete. Iniziamo con l'aggiunta delle autorizzazioni di rete al file App Manifest. Nota che sono stati aggiunti il tag `user-permission` e l'attributo `networkSecurityConfig`, che indirizzeranno alla nostra nuova configurazione di sicurezza di rete. Nel codice seguente, sostituisci *<version>* con il numero di versione corrente dell'SDK di chat per Android (ad esempio, 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")
}
```

Dichiara i domini `10.0.2.2` e `localhost` come attendibili per iniziare a scambiare messaggi con il nostro back-end:

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>
```

Successivamente, dobbiamo aggiungere una nuova dipendenza, insieme al Gson converter addition per l'analisi delle risposte HTTP. Nel codice seguente, sostituisci *<version>* con il numero di versione corrente dell'SDK di chat per Android (ad esempio, 1.0.0).

Script Kotlin:

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

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

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

Per recuperare un token di chat, dobbiamo effettuare una richiesta HTTP POST dalla nostra app `chatterbox`. Definiamo la richiesta in un'interfaccia da implementare con Retrofit. Consulta la documentazione di Retrofit. Inoltre, acquisisci familiarità con le specifiche dell'endpoint CreateChatToken.

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>
}
```

Ora che la rete è stata configurata, è il momento di aggiungere una funzione responsabile della creazione e della gestione del token di chat. La aggiungiamo a `MainActivity.kt`, che è stato creato automaticamente quando il progetto è stato generato:

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)
         }
      })
   }
}
```

Fasi successive

Ora che hai stabilito una connessione alla chat room, vai alla parte 2 di questo tutorial per Android, Messaggi ed eventi

SDK per la messaggistica per client di Chat IVS - Tutorial per Android, parte 2: messaggi ed eventi

Questa seconda e ultima parte del tutorial è suddivisa in diverse sezioni:

1. the section called “Creazione di un'interfaccia utente per l'invio di messaggi”
   a. the section called “Layout principale dell'interfaccia utente”
   b. the section called “Cella di testo astratta dell'interfaccia utente per visualizzare il testo in modo coerente”
   c. the section called “Messaggio a sinistra dell'interfaccia utente di chat”
   d. the section called “Messaggio a destra dell'interfaccia utente di chat”
   e. the section called “Valori di colore aggiuntivi dell'interfaccia utente”
2. the section called “Applicazione dell'associazione di visualizzazione”
3. the section called “Gestione delle richieste di messaggi di chat”
4. the section called “Passaggi finali”

Per la documentazione completa dell'SDK, inizia con l'SDK di messaggistica per client di chat Amazon IVS (qui nella Guida per l'utente di Chat Amazon IVS) e la Documentazione di riferimento dell'SDK di messaggistica per client di chat per Android su GitHub.

Prerequisito

Assicurati di aver completato la prima parte di questo tutorial, Chat room.

Creazione di un'interfaccia utente per l'invio di messaggi

Ora che abbiamo inizializzato correttamente la connessione alla chat room, è il momento di inviare il primo messaggio. Per questa funzionalità è necessaria un'interfaccia utente. Aggiungeremo:

- Pulsante `connect/disconnect`
- Inserimento di messaggi con il pulsante `send`
- Elenco dei messaggi dinamici. Per realizzarlo, utilizziamo RecyclerView di Android Jetpack.

Layout principale dell'interfaccia utente

Consulta la pagina Layout di Android Jetpack nella documentazione per gli sviluppatori Android.

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>
```

Cella di testo astratta dell'interfaccia utente per visualizzare il testo in modo coerente

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>
```

Messaggio a sinistra dell'interfaccia utente di chat

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>
```

Messaggio a destra dell'interfaccia utente di chat

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>
```

Valori di colore aggiuntivi dell'interfaccia utente

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>
```

Applicazione dell'associazione di visualizzazione

Sfruttiamo la funzione di Android Visualizzazione associazione per poter fare riferimento alle classi di associazione per il nostro layout XML. Per abilitare la funzionalità, imposta l'opzione di compilazione `viewBinding` su `true` in `./app/build.gradle`:

Script Kotlin:

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

```
android {
//    ...

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

Ora è il momento di connettere l'interfaccia utente con il codice Kotlin:

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)
    }
}
```

Aggiungiamo anche dei metodi per eliminare i messaggi e disconnettere gli utenti dalla chat, che possono essere richiamati utilizzando il menu contestuale dei messaggi di chat:

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

Gestione delle richieste di messaggi di chat

Abbiamo bisogno di un modo per gestire le nostre richieste di messaggi di chat in tutti i loro stati possibili:

- In sospeso: un messaggio è stato inviato a una chat room ma non è stato ancora confermato o rifiutato.
- Confermato: un messaggio è stato inviato dalla chat room a tutti gli utenti, inclusi noi.
- Rifiutato: un messaggio è stato rifiutato dalla chat room con un oggetto di errore.

Conserveremo le richieste di chat e i messaggi di chat non risolti in un elenco. L'elenco merita una classe a parte, che chiamiamo `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()
    }
```

```
}
```

Per collegare l'elenco all'interfaccia utente, utilizziamo un adattatore. Per ulteriori informazioni, consulta le pagine Associazione dei dati con AdapterView e Classi di associazione generate.

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
}
```

Passaggi finali

È ora di collegare il nuovo adattatore, vincolando la classe `ChatEntries` a `MainActivity`:

Kotlin:

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

package com.chatterbox.myapp
// ...

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

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

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

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

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

        setUpChatView()

        updateConnectionState(ConnectionState.DISCONNECTED)
    }

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

        val recyclerViewLayoutManager = LinearLayoutManager(this@MainActivity,
 LinearLayoutManager.VERTICAL, false)
```

```
        binding.recyclerView.layoutManager = recyclerViewLayoutManager
        binding.recyclerView.adapter = adapter

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

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

Poiché abbiamo già una classe responsabile di tenere traccia delle richieste di chat (`ChatEntries`), siamo pronti a implementare il codice per la manipolazione delle `entries` in `roomListener`. Aggiorneremo `entries` e `connectionState` in base all'evento a cui stiamo rispondendo:

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")
        }
    }
}
```

Ora dovresti essere in grado di eseguire la tua applicazione. Consulta la pagina Costruzione ed esecuzione dell'app. Ricorda che il server di back-end deve essere in funzione quando usi l'app. Puoi avviarlo dal terminale alla root del progetto con il comando `./gradlew :auth-server:run` oppure eseguendo l'attività `auth-server:run` di Gradle direttamente da Android Studio.

SDK per la messaggistica per client di Chat IVS - Tutorial per le coroutine di Kotlin, parte 1: chat room

Questo è il primo di un tutorial a due parti. Scoprirai gli elementi essenziali per utilizzare l'SDK di messaggistica per chat Amazon IVS creando un'app Android funzionale e completa con il linguaggio di programmazione Kotlin e le coroutine. Chiameremo l'app Chatterbox.

Prima di avviare il modulo, dedica qualche minuto a familiarizzare con i prerequisiti, i concetti chiave alla base dei token di chat e il server di back-end necessario per creare le chat room.

Questi tutorial sono creati per sviluppatori Android esperti che non hanno mai utilizzato l'SDK per la messaggistica di chat IVS. Dovrai essere a tuo agio con il linguaggio di programmazione Kotlin e con la creazione di interfacce utente sulla piattaforma Android.

Questa prima parte del tutorial è suddivisa in diverse sezioni:

1. the section called “Configurazione di un server di autenticazione/autorizzazione locale”
2. the section called “Creazione di un progetto Chatterbox”
3. the section called “Connessione a una chat room e osservazione degli aggiornamenti della connessione”
4. the section called “Creazione di un provider di token”
5. the section called “Fasi successive”

Per la documentazione completa dell'SDK, inizia con l'SDK di messaggistica per client di chat Amazon IVS (qui nella Guida per l'utente di Chat Amazon IVS) e la Documentazione di riferimento dell'SDK di messaggistica per client di chat per Android su GitHub.

Prerequisiti

- Devi avere dimestichezza con Kotlin e con la creazione di applicazioni sulla piattaforma Android. Se non hai dimestichezza con la creazione di applicazioni per Android, scopri le nozioni di base nella guida Creazione della prima app per gli sviluppatori Android.

- Leggi e comprendi Nozioni di base su Chat IVS.
- Crea un utente IAM AWS con le capacità `CreateChatToken` e `CreateRoom` definite in una policy IAM esistente. Consultare Nozioni di base su Chat IVS.
- Assicurati che la chiavi di accesso segrete di questo utente siano archiviata in un file di credenziali AWS. Per istruzioni, consulta la Guida per l'utente di AWS CLI (in particolare la sezione Configurazione e impostazioni del file delle credenziali).
- Crea una chat room e salva il relativo ARN. Consultare Nozioni di base su Chat IVS. (Se non salvi l'ARN, potrai cercarlo in un secondo momento con la console o l'API di Chat.)

Configurazione di un server di autenticazione/autorizzazione locale

Il tuo server di back-end è responsabile sia della creazione di chat room sia della generazione dei token di chat necessari all'SDK di chat IVS per Android per autenticare e autorizzare i client ad accedere alle tue chat room.

Consulta Creazione di un token di chat nella Guida introduttiva ad Amazon IVS Chat. Come mostrato nel diagramma di flusso, il codice lato server è responsabile della creazione di un token di chat. Ciò significa che l'app deve fornire i propri mezzi per generare un token di chat richiedendone uno dall'applicazione lato server.

Utilizziamo il framework Ktor per creare un server locale live che gestisca la creazione di token di chat utilizzando l'ambiente AWS locale.

A questo punto, ci aspettiamo che le tue credenziali AWS siano configurate correttamente. Per istruzioni dettagliate, consulta la pagina Set up AWS temporary credentials and AWS Region for development.

Crea una nuova cartella e chiamala `chatterbox` e al suo interno creane un'altra, denominata `auth-server`.

La cartella sul server avrà la seguente struttura:

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

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

Nota: puoi copiare/incollare direttamente il codice qui nei file di riferimento.

Successivamente, aggiungiamo tutte le dipendenze e i plugin necessari per il funzionamento del server di autenticazione:

Script Kotlin:

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

Ora dobbiamo configurare la funzionalità di registrazione per il server di autenticazione. Per ulteriori informazioni, consulta la sezione Configurazione del logger.

XML:

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

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

Il server Ktor richiede delle impostazioni di configurazione, che carica automaticamente dal file `application.*` nella directory `resources`, quindi aggiungiamo anche quelle. Per ulteriori informazioni, consulta la sezione Configurazione in un file.

HOCON:

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

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

Infine, implementiamo il server:

Kotlin:

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

package com.chatterbox.authserver

import io.ktor.http.*
import io.ktor.serialization.kotlinx.json.*
import io.ktor.server.application.*
```

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

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

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

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

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

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

Creazione di un progetto Chatterbox

Per creare un progetto Android, installa e apri Android Studio.

Segui i passaggi elencati nella guida Creazione di un progetto ufficiale di Android.

- In Choose your project type, scegli il modello di progetto Empty Activity per la nostra app Chatterbox.

- In Configure your project, scegli i seguenti valori per i campi di configurazione:
  - Nome: My App
  - Nome del pacchetto: com.chatterbox.myapp
  - Percorso di salvataggio: fai riferimento alla cartella `chatterbox` creata nel passaggio precedente
  - Lingua: Kotlin
  - Livello minimo API: API 21: Android 5.0 (Lollipop)

Dopo aver specificato correttamente tutti i parametri di configurazione, la struttura dei file all'interno della cartella `chatterbox` dovrebbe essere la seguente:

```
- 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
```

Ora che abbiamo un progetto Android funzionante, possiamo aggiungere com.amazonaws:ivs-chat-messaging e org.jetbrains.kotlinx:kotlinx-coroutines-core alle nostre dipendenze `build.gradle`. Per ulteriori informazioni sul toolkit di compilazione Gradle, consulta la pagina Configurazione della compilazione.

Nota: nella parte superiore di ogni frammento di codice, è riportato il percorso del file in cui dovresti apportare modifiche al tuo progetto. Il percorso fa riferimento alla cartella del progetto.

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'

// ...
}
```

Dopo aver aggiunto la nuova dipendenza, esegui Sincronizzazione dei progetti con i file Gradle in Android Studio per sincronizzare il progetto con la nuova dipendenza. Per ulteriori informazioni, consulta la pagina Aggiunta di dipendenze della compilazione.

Per eseguire comodamente il server di autenticazione che abbiamo creato nella sezione precedente dalla root del progetto, lo includiamo come nuovo modulo in `settings.gradle`. Per ulteriori informazioni, consulta la pagina Strutturazione e costruzione di un componente software con Gradle.

Script Kotlin:

```
// ./settings.gradle

// ...

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

D'ora in avanti, dato che `auth-server` è incluso nel progetto Android, puoi eseguire il server di autenticazione con il seguente comando dalla root del progetto:

Shell (interprete di comandi):

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

Connessione a una chat room e osservazione degli aggiornamenti della connessione

Per aprire una connessione alla chat room, utilizziamo il callback del ciclo di vita dell'attività onCreate(), che si attiva quando l'attività viene creata per la prima volta. Il costruttore di ChatRoom ci richiede di fornire `region` e `tokenProvider` per inizializzare una connessione alla chat room.

Nota: la funzione `fetchChatToken` nel frammento di seguito verrà implementata nella sezione successiva.

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)
    }

// ...
}
```

Visualizzare i cambiamenti nello stato della connessione di una chat room e reagire a essi sono parti essenziali della creazione di un'app di chat come `chatterbox`. Prima di poter iniziare a interagire con la chat room, dobbiamo iscriverci agli eventi sullo stato di connessione della chat room per ricevere aggiornamenti.

Nell'SDK di chat per le coroutine, la `ChatRoom` si aspetta che gestiamo gli eventi del ciclo di vita delle chat room in Flow. Per ora, le funzioni dell'ascoltatore registreranno solo i messaggi di conferma, quando richiamati:

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")
                }
            }
        }
    }
}
```

Successivamente, dobbiamo fornire la capacità di leggere lo stato della connessione della chat room. Lo manterremo nella proprietà `MainActivity.kt` e lo inizializzeremo allo stato predefinito DISCONNECTED (disconnesso) per le chat room (consulta la sezione `state` in `ChatRoom` nella Documentazione di riferimento dell'SDK di chat IVS per Android). Per poter mantenere aggiornato lo stato locale, dobbiamo implementare una funzione di aggiornamento dello stato, che chiameremo `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")
          }
      }
}
```

Successivamente, integriamo la funzione di aggiornamento dello stato con la proprietà 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)

                }
```

```
            }

        // ...

        }
    }
}
```

Ora che abbiamo la possibilità di salvare, ascoltare e reagire agli aggiornamenti dello stato di ChatRoom, è il momento di inizializzare una connessione:

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)
        }
    }

    // ...
}
```

Creazione di un provider di token

È il momento di creare una funzione responsabile della creazione e della gestione dei token di chat nell'applicazione. In questo esempio utilizziamo il client HTTP Retrofit per Android.

Prima di poter inviare traffico di rete, dobbiamo impostare una configurazione di sicurezza di rete per Android. Per ulteriori informazioni, consulta la pagina Configurazione della sicurezza di rete. Iniziamo con l'aggiunta delle autorizzazioni di rete al file App Manifest. Nota che sono stati aggiunti il tag `user-permission` e l'attributo `networkSecurityConfig`, che indirizzeranno alla nostra

nuova configurazione di sicurezza di rete. Nel codice seguente, sostituisci *<version>* con il numero di versione corrente dell'SDK di chat per Android (ad esempio, 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")
}
```

Dichiara i domini IP locali, come ad esempio `10.0.2.2` e `localhost`, come attendibili per iniziare a scambiare messaggi con il nostro back-end:

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>
```

Successivamente, dobbiamo aggiungere una nuova dipendenza, insieme al Gson converter addition per l'analisi delle risposte HTTP. Nel codice seguente, sostituisci *<version>* con il numero di versione corrente dell'SDK di chat per Android (ad esempio, 1.1.0).

Script Kotlin:

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

Per recuperare un token di chat, dobbiamo effettuare una richiesta HTTP POST dalla nostra app `chatterbox`. Definiamo la richiesta in un'interfaccia da implementare con Retrofit. Consulta la documentazione di Retrofit. Inoltre, acquisisci familiarità con le specifiche dell'endpoint CreateChatToken.

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)
   }
}
```

Ora che la rete è stata configurata, è il momento di aggiungere una funzione responsabile della creazione e della gestione del token di chat. La aggiungiamo a `MainActivity.kt`, che è stato creato automaticamente quando il progetto è stato generato:

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)
         }
      })
   }
}
```

Fasi successive

Ora che hai stabilito una connessione alla chat room, vai alla parte 2 di questo tutorial per le coroutine di Kotlin, Messaggi ed eventi

SDK per la messaggistica per client di IVS Chat: Tutorial per le coroutine di Kotlin, parte 2: messaggi ed eventi

Questa seconda e ultima parte del tutorial è suddivisa in diverse sezioni:

1. the section called “Creazione di un'interfaccia utente per l'invio di messaggi”
   a. the section called “Layout principale dell'interfaccia utente”
   b. the section called “Cella di testo astratta dell'interfaccia utente per visualizzare il testo in modo coerente”
   c. the section called “Messaggio a sinistra dell'interfaccia utente di chat”
   d. the section called “Messaggio a destra dell'interfaccia utente”
   e. the section called “Valori di colore aggiuntivi dell'interfaccia utente”
2. the section called “Applicazione dell'associazione di visualizzazione”
3. the section called “Gestione delle richieste di messaggi di chat”
4. the section called “Passaggi finali”

Per la documentazione completa dell'SDK, inizia con l'SDK di messaggistica per client di chat Amazon IVS (qui nella Guida per l'utente di Chat Amazon IVS) e la Documentazione di riferimento dell'SDK di messaggistica per client di chat per Android su GitHub.

Prerequisito

Assicurati di aver completato la prima parte di questo tutorial, Chat room.

Creazione di un'interfaccia utente per l'invio di messaggi

Ora che abbiamo inizializzato correttamente la connessione alla chat room, è il momento di inviare il primo messaggio. Per questa funzionalità è necessaria un'interfaccia utente. Aggiungeremo:

- Pulsante `connect/disconnect`
- Inserimento di messaggi con il pulsante `send`

- Elenco dei messaggi dinamici. Per realizzarlo, utilizziamo RecyclerView di Android Jetpack.

Layout principale dell'interfaccia utente

Consulta la pagina Layout di Android Jetpack nella documentazione per gli sviluppatori Android.

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>
```

Cella di testo astratta dell'interfaccia utente per visualizzare il testo in modo coerente

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>
```

Messaggio a sinistra dell'interfaccia utente di chat

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>
```

Messaggio a destra dell'interfaccia utente

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>
```

Valori di colore aggiuntivi dell'interfaccia utente

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>
```

Applicazione dell'associazione di visualizzazione

Sfruttiamo la funzione di Android Visualizzazione associazione per poter fare riferimento alle classi di associazione per il nostro layout XML. Per abilitare la funzionalità, imposta l'opzione di compilazione `viewBinding` su `true` in `./app/build.gradle`:

Script Kotlin:

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

android {
//    ...

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

Ora è il momento di connettere l'interfaccia utente con il codice Kotlin:

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)
    }
// ...

}
```

Aggiungiamo anche dei metodi per eliminare i messaggi e disconnettere gli utenti dalla chat, che possono essere richiamati utilizzando il menu contestuale dei messaggi di chat:

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")
           }
        }
    }
}
```

Gestione delle richieste di messaggi di chat

Abbiamo bisogno di un modo per gestire le nostre richieste di messaggi di chat in tutti i loro stati possibili:

- In sospeso: un messaggio è stato inviato a una chat room ma non è stato ancora confermato o rifiutato.
- Confermato: un messaggio è stato inviato dalla chat room a tutti gli utenti, inclusi noi.
- Rifiutato: un messaggio è stato rifiutato dalla chat room con un oggetto di errore.

Conserveremo le richieste di chat e i messaggi di chat non risolti in un elenco. L'elenco merita una classe a parte, che chiamiamo `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()
    }
}
```

Per collegare l'elenco all'interfaccia utente, utilizziamo un adattatore. Per ulteriori informazioni, consulta le pagine Associazione dei dati con AdapterView e Classi di associazione generate.

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
}
```

Passaggi finali

È ora di collegare il nuovo adattatore, vincolando la classe `ChatEntries` a `MainActivity`:

Kotlin:

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

package com.chatterbox.myapp
// ...

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

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

    // ...

    private fun setUpChatView() {
        adapter = ChatListAdapter(entries, ::disconnectUser)
        entries.adapter = adapter

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

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

            sendButtonClick(binding.sendButton)
```

```
            return@setOnEditorActionListener true
        }
    }
}
```

Poiché abbiamo già una classe responsabile di tenere traccia delle richieste di chat (`ChatEntries`), siamo pronti a implementare il codice per la manipolazione delle `entries` in RoomListener. Aggiorneremo `entries` e `connectionState` in base all'evento a cui stiamo rispondendo:

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)
    }

// ...

}
```

Ora dovresti essere in grado di eseguire la tua applicazione. Consulta la pagina Costruzione ed esecuzione dell'app. Ricorda che il server di back-end deve essere in funzione quando usi l'app. Puoi avviarlo dal terminale alla root del progetto con il comando `./gradlew :auth-server:run` oppure eseguendo l'attività `auth-server:run` di Gradle direttamente da Android Studio.

SDK per la messaggistica client di Chat IVS - Guida per iOS

L'SDK per iOS di Amazon Interactive Video (IVS) Chat Client Messaging fornisce interfacce che consentono di incorporare facilmente l'API IVS Chat Messaging su piattaforme che utilizzano il linguaggio di programmazione Swift di Apple.

Ultima versione dell'SDK per iOS di IVS Chat Client Messaging: 1.0.0 (Note di rilascio)

Documentazione di riferimento e tutorial: per informazioni sui metodi più importanti disponibili nell'SDK per iOS di Amazon IVS Chat Client Messaging consultare la documentazione di riferimento all'indirizzo https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/. Questo archivio contiene anche vari articoli e tutorial.

Codice di esempio: consultare il repository di esempio iOS su GitHub: https://github.com/aws-samples/amazon-ivs-chat-for-ios-demo.

Requisiti della piattaforma: per lo sviluppo è richiesto iOS 13.0 o versioni successive.

Guida introduttiva all'SDK di messaggistica per client di chat IVS su iOS

Si consiglia di integrare l'SDK tramite Swift Package Manager. In alternativa, è possibile utilizzare CocoaPods o integrare il framework manualmente.

Dopo aver integrato l'SDK, è possibile importare l'SDK aggiungendo il seguente codice nella parte superiore del file Swift pertinente:

```
import AmazonIVSChatMessaging
```

Swift Package Manager

Per utilizzare la libreria `AmazonIVSChatMessaging` in un progetto Swift Package Manager, aggiungerla alle dipendenze del pacchetto e alle dipendenze per i propri obiettivi pertinenti:

1. Scaricare l'ultima versione di `.xcframework` da https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. Nel proprio Terminale, eseguire:

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

3. Prendere l'output del passaggio precedente e incollarlo nella proprietà checksum di `.binaryTarget` come mostrato di seguito all'interno del file `Package.swift` del progetto:

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

CocoaPods

I rilasci sono pubblicati tramite CocoaPods sotto il nome `AmazonIVSChatMessaging`. Aggiungere questa dipendenza al proprio Podfile:

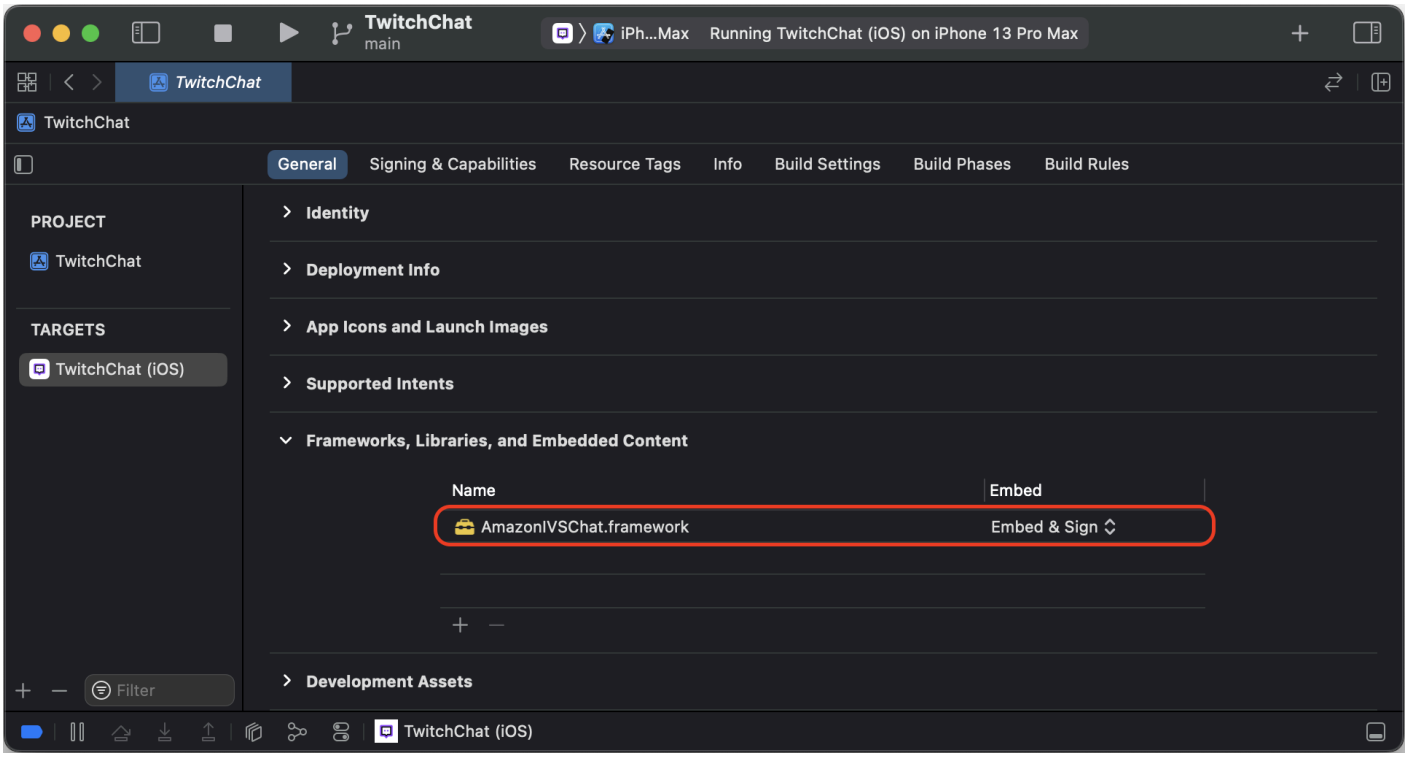
```
pod 'AmazonIVSChat'
```

Eseguire `pod install` e l'SDK sarà disponibile nel `.xcworkspace`.

Installare manualmente

1. Scaricare l'ultima versione da https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. Estrarre i contenuti dell'archivio. `AmazonIVSChatMessaging.xcframework` contiene l'SDK sia per il dispositivo sia per il simulatore.

3. Incorporare il `AmazonIVSChatMessaging.xcframework` estratto trascinandolo nella sezione Framework, librerie e contenuto incorporato della scheda General (Generale) per il target dell'applicazione:



Utilizzo dell'SDK di messaggistica per client di chat iOS IVS

Questo documento illustra i passaggi necessari per l'utilizzo dell'SDK di messaggistica per client di chat Amazon IVS su iOS.

Connessione a una chat room

Prima di iniziare, acquisire familiarità con Nozioni di base su Amazon IVS Chat. Vedere anche le app di esempio per il Web, Android e iOS.

Per connettersi a una chat room, l'app necessita di un modo per recuperare un token di chat fornito dal back-end. L'applicazione probabilmente recupererà un token di chat utilizzando una richiesta di rete al back-end.

Per comunicare questo token di chat recuperato all'SDK, il modello di `ChatRoom` dell'SDK richiede di fornire una funzione `async` o un'istanza di un oggetto conforme al protocollo del `ChatTokenProvider` fornito nel punto di inizializzazione. Il valore restituito da uno di questi metodi deve essere un'istanza del modello di `ChatToken` dell'SDK.

Nota: si popolano le istanze del modello di `ChatToken` che utilizza i dati recuperati dal back-end. I campi richiesti per inizializzare un'istanza del `ChatToken` sono uguali a quelli dei campi della risposta a CreateChatToken. Per ulteriori informazioni sull'inizializzazione delle istanze del modello di `ChatToken`, consultare Creazione di un'istanza di ChatToken. Il back-end è responsabile della fornitura dei dati nella risposta di `CreateChatToken` all'app. Il modo in cui si decide di comunicare con il proprio back-end per generare token di chat dipende dall'app e dalla sua infrastruttura.

Dopo aver scelto la strategia per fornire un `ChatToken` all'SDK, chiamare `.connect()` dopo aver correttamente inizializzato un'istanza di `ChatRoom` con il fornitore di token e la Regione AWS che il back-end ha usato per creare la chat room a cui si sta cercando di connettersi. Tenere presente che `.connect()` è una funzione asincrona di lancio:

```
import AmazonIVSChatMessaging

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

Conformità al protocollo ChatTokenProvider

Per il parametro `tokenProvider` nell'inizializzatore per la `ChatRoom` è possibile specificare un'istanza di `ChatTokenProvider`. Di seguito è illustrato un esempio di oggetto conforme a `ChatTokenProvider`:

```
import AmazonIVSChatMessaging

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

È, quindi, possibile prendere un'istanza di questo oggetto conforme e passarla all'inizializzatore per la `ChatRoom`:

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

Produzione di una funzione asincrona in Swift

Supponiamo di avere già un gestore da utilizzare per gestire le richieste di rete dell'applicazione. Potrebbe essere simile a quanto segue:

```
import AmazonIVSChatMessaging

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

Si potrebbe semplicemente aggiungere un'altra funzione nel gestore per recuperare un `ChatToken` dal back-end:

```
import AmazonIVSChatMessaging

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

Quindi, usare il riferimento a quella funzione in Swift quando si inizializza una `ChatRoom`:

```
import AmazonIVSChatMessaging
```

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

Creazione di un'istanza di ChatToken

È possibile creare facilmente un'istanza di `ChatToken` utilizzando l'inizializzatore fornito nell'SDK. Consultare la documentazione in `Token.swift` per ulteriori informazioni sulle proprietà del `ChatToken`.

```
import AmazonIVSChatMessaging

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

Utilizzo di Decodable

Se, mentre ci si interfaccia con l'API di IVS Chat, il back-end decide di inoltrare semplicemente la risposta CreateChatToken all'applicazione di front-end, è possibile sfruttare la conformità del `ChatToken` al protocollo `Decodable` di Swift. Tuttavia, esiste una condizione.

Il payload della risposta alla `CreateChatToken` utilizza stringhe per le date formattate sfruttando lo Standard ISO 8601 per i timestamp di Internet. Normalmente in Swift, si fornirebbe lo `JSONDecoder.DateDecodingStrategy.iso8601` come valore per la proprietà `.dateDecodingStrategy` di `JSONDecoder`. Tuttavia, la `CreateChatToken` utilizza frazioni di secondi ad alta precisione nelle sue stringhe e ciò non è supportato dallo `JSONDecoder.DateDecodingStrategy.iso8601`.

Per comodità, l'SDK fornisce un'estensione pubblica su `JSONDecoder.DateDecodingStrategy` con un ulteriore strategia `.preciseISO8601` che consente di utilizzare con successo `JSONDecoder` durante la decodifica di un'istanza di `ChatToken`:

```
import AmazonIVSChatMessaging
```

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

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

Disconnessione da una chat room

Per disconnettersi manualmente da un'istanza della `ChatRoom` a cui ci si è connessi con successo, chiamare `room.disconnect()`. Per impostazione predefinita, le chat room richiamano automaticamente questa funzione quando vengono deallocate.

```
import AmazonIVSChatMessaging

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

// Disconnect
room.disconnect()
```

Ricezione di un messaggio/evento di chat

Per inviare e ricevere messaggi nella propria chat room, è necessario fornire un oggetto conforme al protocollo `ChatRoomDelegate`, dopo aver inizializzato con successo un'istanza della `ChatRoom` e chiamato `room.connect()`. Ecco un tipico esempio, usando `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) { ... }
}
```

Ricezione di una notifica quando la connessione cambia

Come prevedibile, non è possibile eseguire azioni come l'invio di un messaggio in una stanza finché la stanza non è completamente connessa. L'architettura dell'SDK cerca di favorire la connessione a una ChatRoom su un thread in background tramite API asincrone. Nel caso in cui si voglia creare qualcosa nella propria interfaccia utente che disabiliti un pulsante di invio di messaggi, l'SDK fornisce due strategie per ricevere una notifica quando lo stato della connessione di una chat room cambia, utilizzando `Combine` o `ChatRoomDelegate`. Queste sono descritte di seguito.

Importante: lo stato della connessione di una chat room potrebbe anche cambiare a causa di un'interruzione di rete. Tenerlo in considerazione quando si crea l'app.

Uso di Combine

Ogni istanza della `ChatRoom` viene fornita con il suo editore `Combine`, sotto forma di proprietà `state`:

```
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()
}
```

Uso di ChatroomDelegate

In alternativa, utilizzare le funzioni opzionali `roomDidConnect(_:)`, `roomIsConnecting(_:)` e `roomDidDisconnect(_:)` all'interno di un oggetto conforme a `ChatRoomDelegate`. Di seguito è riportato un esempio che utilizza un `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!")
   }
}
```

Esecuzione di azioni in una chat room

Utenti diversi hanno capacità diverse per quanto riguarda le azioni che possono eseguire in una chat room, ad esempio inviare un messaggio, eliminare un messaggio o disconnettere un utente. Per eseguire una di queste azioni, chiamare `perform(request:)` su una `ChatRoom` connessa, passando in un'istanza di uno degli oggetti `ChatRequest` forniti nell'SDK. Le richieste supportate sono in `Request.swift`.

Alcune azioni eseguite in una chat room richiedono agli utenti connessi di disporre di funzionalità specifiche quando l'applicazione di back-end chiama `CreateChatToken`. In base alla progettazione, l'SDK non è in grado di distinguere le funzionalità di un utente connesso. Quindi, anche se è possibile provare a eseguire azioni di moderazione in un'istanza connessa della `ChatRoom`, l'API del piano di controllo decide in ultima analisi se l'azione avrà successo.

Tutte le azioni eseguite tramite `room.perform(request:)` restano in attesa fintanto che la stanza riceve l'istanza prevista di un modello (il cui tipo è associato all'oggetto richiesto stesso), corrispondente a `requestId` sia del modello ricevuto che dell'oggetto della richiesta. Se c'è un problema con la richiesta, `ChatRoom` genera sempre un errore sotto forma di un `ChatError`. La definizione di `ChatError` è in `Error.swift`.

Invio di un messaggio

Per inviare un messaggio via chat, usare un'istanza di `SendMessageRequest`:

```
import AmazonIVSChatMessaging
```

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

Come accennato in precedenza, `room.perform(request:)` restituisce un risultato quando un `ChatMessage` corrispondente viene ricevuto dalla `ChatRoom`. Se c'è un problema con la richiesta (ad esempio il superamento del limite di caratteri del messaggio per una stanza) si avvia, invece, un'istanza di `ChatError`. È quindi possibile visualizzare queste informazioni nell'interfaccia utente:

```
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)
}
```

Aggiunta di metadati a un messaggio

Quando si invia un messaggio è possibile aggiungere metadati ad esso associati. `SendMessageRequest` ha una proprietà `attributes`, con cui è possibile inizializzare la richiesta. I dati allegati vengono associati al messaggio quando altri lo ricevono nella stanza.

Ecco un esempio di come allegare dati remoti a un messaggio inviato:

```
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"
      ]
   )
)
```

Utilizzare `attributes` in una `SendMessageRequest` può essere estremamente utile per creare funzionalità complesse nel proprio prodotto chat. Ad esempio, è possibile creare funzionalità di threading utilizzando il dizionario degli attributi `[String : String]` in una `SendMessageRequest`.

Il payload di `attributes` è molto flessibile e potente. Utilizzarlo per ricavare informazioni sul proprio messaggio che altrimenti non si riuscirebbero a ottenere. Ad esempio, può essere usato per ottenere informazioni sulle emote tramite l'uso degli attributi, che è molto più semplice rispetto all'analisi della stringa di un messaggio.

Eliminazione di un messaggio

Eliminare un messaggio di chat è come inviarne uno. Utilizzare la funzione `room.perform(request:)` sulla `ChatRoom` per raggiungere questo obiettivo creando un'istanza di `DeleteMessageRequest`.

Per accedere facilmente alle istanze precedenti dei messaggi di chat ricevuti, passa il valore di `message.id` all'inizializzatore di `DeleteMessageRequest`.

Facoltativamente, fornire una motivazione di stringa per `DeleteMessageRequest`, in modo da ritrovarla nella propria interfaccia utente.

```
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!"
   )
)
```

Poiché si tratta di un'azione da moderatore, l'utente potrebbe non avere effettivamente la capacità di eliminare il messaggio di un altro utente. È possibile utilizzare la meccanica delle funzioni avviabili di Swift per far emergere un messaggio di errore nell'interfaccia utente quando un utente tenta di eliminare un messaggio senza la funzionalità appropriata.

Quando il back-end chiama `CreateChatToken` per un utente, deve passare `"DELETE_MESSAGE"` nel campo `capabilities` al fine di attivare tale funzionalità per un utente connesso alla chat.

Ecco un esempio di rilevamento di un errore di funzionalità generato quando si tenta di eliminare un messaggio senza le autorizzazioni appropriate:

```
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)
}
```

Disconnessione di un altro utente

Utilizzare `room.perform(request:)` per disconnettere un altro utente da una chat room. In particolare, usare un'istanza di `DisconnectUserRequest`. Tutti i `ChatMessage` ricevuti da una `ChatRoom` hanno la proprietà `sender`, che contiene l'ID utente e che è necessario inizializzare correttamente con un'istanza di `DisconnectUserRequest`. Facoltativamente, fornire una motivazione di stringa per la richiesta di disconnessione.

```
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
   )
)
```

Poiché questo è un altro esempio di azione da moderatore, è possibile disconnettere un altro utente solo se si dispone della funzionalità `DISCONNECT_USER`. La funzionalità viene impostata quando l'applicazione di back-end chiama `CreateChatToken` e inserisce la stringa `"DISCONNECT_USER"` nel campo `capabilities`.

Se l'utente non è in grado di disconnettere un altro utente, `room.perform(request:)` avvia un'istanza di `ChatError`, proprio come le altre richieste. È possibile controllare la proprietà dell'errore `errorCode` per determinare se la richiesta sia fallita a causa della mancanza di privilegi di moderatore:

```
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)
}
```

SDK per la messaggistica client di Chat IVS - Tutorial per iOS

L'SDK iOS per la messaggistica del client Amazon Interactive Video (IVS) Chat fornisce interfacce che consentono di incorporare facilmente l'API di messaggistica di IVS Chat su piattaforme che utilizzano il linguaggio di programmazione Swift di Apple.

Per un tutorial sull'SDK per iOS di Chat, consulta https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/latest/tutorials/table-of-contents/.

SDK per la messaggistica client di Chat IVS: guida per JavaScript

L'SDK Javascript di Amazon Interactive Video (IVS) Chat Client Messaging consente di incorporare l'API Amazon IVS Chat Messaging su piattaforme che utilizzano un browser Web.

Ultima versione dell'SDK Javascript di messaggistica client di IVS Chat: 1.0.2 (Note di rilascio)

Documentazione di riferimento: per informazioni sui metodi più importanti disponibili nell'SDK Javascript di Amazon IVS Chat Client Messaging consultare la documentazione di riferimento all'indirizzo https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/

Codice di esempio: consultare il repository di esempio su GitHub, per una demo specifica per il Web utilizzando l'SDK JavaScript: https://github.com/aws-samples/amazon-ivs-chat-web-demo

Guida introduttiva all'SDK di messaggistica per client di chat IVS su JavaScript

Prima di iniziare, acquisire familiarità con Nozioni di base su Amazon IVS Chat.

Aggiungere il pacchetto

Usa:

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

oppure:

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

Supporto React Native

L'SDK JavaScript di IVS Chat Client Messaging ha una dipendenza `uuid` che utilizza il metodo `crypto.getRandomValues`. Poiché questo metodo non è supportato in React Native, è necessario installare il polyfill aggiuntivo `react-native-get-random-value` e importarlo nella parte superiore del file `index.js`:

```
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);
```

Impostazione del back-end

Questa integrazione richiede endpoint sul server che comunichino con l'Amazon IVS Chat API. Utilizzare le librerie AWS ufficiali per accedere all'API Amazon IVS dal proprio server. Queste librerie sono accessibili in diverse lingue dai pacchetti pubblici, ad esempio node.js, java e go.

Creare un endpoint del server che comunichi con l'endpoint CreateChatToken dell'Amazon IVS Chat API per creare un token di chat per gli utenti della chat.

Utilizzo dell'SDK di messaggistica per client di chat IVS su JavaScript

Questo documento illustra i passaggi necessari per l'utilizzo dell'SDK di messaggistica per client di chat Amazon IVS su JavaScript.

Inizializzare un'istanza di chat room

Creare un'istanza della classe `ChatRoom`. Ciò richiede il trasferimento di `regionOrUrl` (la regione AWS in cui è ospitata la chat room) e `tokenProvider` (il metodo di recupero dei token verrà creato nel passaggio successivo):

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

Funzione provider di token

Creare una funzione provider di token asincrona che recuperi un token di chat dal proprio back-end:

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

La funzione non deve accettare parametri e deve restituire una Promise contenente un oggetto token di chat:

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

Questa funzione è necessaria per inizializzare l'oggetto ChatRoom. Di seguito, compilae i campi `<token>` e `<date-time>` con i valori ricevuti dal proprio back-end:

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

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

Ricordarsi di trasferire il `tokenProvider` al costruttore di ChatRoom. ChatRoom aggiorna il token quando la connessione viene interrotta o la sessione scade. Non usare il `tokenProvider` per archiviare un token da nessuna parte: ChatRoom lo gestisce per l'utente.

Ricevere eventi

Successivamente, iscriversi agli eventi della chat room per ricevere gli eventi del ciclo di vita, nonché i messaggi e gli eventi distribuiti nella chat room:

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

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

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

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

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

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

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

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

Connessione alla chat room

L'ultimo passaggio dell'inizializzazione di base consiste nel connettersi alla chat room stabilendo una connessione WebSocket. Per farlo, basta chiamare il metodo `connect()` all'interno dell'istanza della stanza:

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

L'SDK cercherà di stabilire una connessione alla chat room codificata nel token di chat ricevuto dal server.

Dopo la chiamata a `connect()`, la stanza passerà allo stato `connecting` ed emetterà un evento `connecting`. Quando la stanza si connette, passa allo stato `connected` ed emette un evento `connect`.

Un errore di connessione potrebbe verificarsi a causa di problemi durante il recupero del token o la connessione a WebSocket. In questo caso, la stanza tenta di riconnettersi automaticamente fino al numero di volte indicato dal parametro del costruttore `maxReconnectAttempts`. Durante i tentativi di riconnessione, la stanza è nello stato `connecting` e non emette eventi aggiuntivi. Dopo aver esaurito i tentativi di riconnessione, la stanza passa allo stato `disconnected` ed emette un evento `disconnect` con un motivo di disconnessione pertinente. Nello stato `disconnected`, la stanza non tenta più di connettersi. È necessario chiamare nuovamente `connect()` per attivare il processo di connessione.

Esecuzione di azioni in una chat room

L'SDK di Amazon IVS Chat Messaging offre agli utenti azioni per inviare ed eliminare messaggi e disconnettere altri utenti. Queste sono disponibili sull'istanza `ChatRoom`. Restituiscono un oggetto `Promise` che consente di ricevere la conferma o il rifiuto della richiesta.

Invio di un messaggio

Per questa richiesta è necessario disporre di una funzionalità `SEND_MESSAGE` codificata nel proprio token di chat.

Per attivare una richiesta di invio di messaggi:

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

Per ottenere una conferma o un rifiuto della richiesta, `await` la promise restituita o utilizzare il metodo `then()`:

```
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.
}
```

Eliminazione di un messaggio

Per questa richiesta è necessario disporre di una funzionalità `DELETE_MESSAGE` codificata nel proprio token di chat.

Per eliminare un messaggio per motivi di moderazione, chiamare il metodo `deleteMessage()`:

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

Per ottenere una conferma o un rifiuto della richiesta, `await` la promise restituita o utilizzare il metodo `then()`:

```
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.
}
```

Disconnessione di un altro utente

Per questa richiesta è necessario disporre di una funzionalità `DISCONNECT_USER` codificata nel proprio token di chat.

Per disconnettere un altro utente per motivi di moderazione, chiamare il metodo `disconnectUser()`:

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

Per ottenere una conferma o un rifiuto della richiesta, `await` la promise restituita o utilizzare il metodo `then()`:

```
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.
}
```

Disconnessione da una chat room

Per chiudere la connessione alla chat room, chiamare il metodo `disconnect()` sull'istanza della `room`:

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

La chiamata a questo metodo fa sì che la stanza chiuda il WebSocket sottostante in modo ordinato. L'istanza della stanza passa a uno stato `disconnected` ed emette un evento di disconnessione, con il motivo `disconnect` impostato su `"clientDisconnect"`.

SDK per la messaggistica client di Chat IVS - Tutorial per JavaScript, parte 1: chat room

Questo è il primo di un tutorial a due parti. Scoprirai gli elementi essenziali per utilizzare l'SDK JavaScript per la messaggistica client di Amazon IVS Chat creando un'app funzionale completa utilizzando JavaScript/TypeScript. Chiameremo l'app Chatterbox.

Il pubblico di riferimento è costituito da sviluppatori esperti che non conoscono l'SDK di messaggistica di Amazon IVS Chat. Dovresti essere a tuo agio con il linguaggio di programmazione JavaScript/TypeScript e la libreria React.

Per brevità, faremo riferimento all'SDK JavaScript di messaggistica del client Amazon IVS Chat come a SDK JS Chat.

Nota: in alcuni casi, gli esempi di codice per JavaScript e TypeScript sono identici, quindi vengono combinati.

Questa prima parte del tutorial è suddivisa in diverse sezioni:

1. the section called “Configurazione di un server di autenticazione/autorizzazione locale”
2. the section called “Creazione di un progetto Chatterbox”
3. the section called “Connessione a una chat room”
4. the section called “Creazione di un provider di token”
5. the section called “Osservazione degli aggiornamenti della connessione”
6. the section called “Creazione di un componente del pulsante di invio”
7. the section called “Creazione dell'input di un messaggio”
8. the section called “Fasi successive”

Per la documentazione completa dell'SDK, inizia con l'SDK di messaggistica per client di chat Amazon IVS (qui nella Guida per l'utente di Chat Amazon IVS) e Documentazione di riferimento dell'SDK di messaggistica per client di chat per JavaScript su GitHub.

Prerequisiti

- Acquisisci familiarità con JavaScript/TypeScript e la libreria React. Se non conosci React Native, scopri le nozioni basilari in questo Tutorial tris.
- Leggi e comprendi Nozioni di base su Chat IVS.
- Crea un utente AWS IAM con le capacità CreateChatToken e CreateRoom definite in una policy IAM esistente. Consultare Nozioni di base su Chat IVS.
- Assicurati che la chiavi di accesso segrete di questo utente siano archiviata in un file di credenziali AWS. Per istruzioni, consulta la Guida per l'utente di AWS CLI (in particolare la sezione Configurazione e impostazioni del file delle credenziali).
- Crea una chat room e salva il relativo ARN. Consultare Nozioni di base su Chat IVS. (Se non salvi l'ARN, potrai cercarlo in un secondo momento con la console o l'API di Chat.)
- Installa l'ambiente Node.js 14+ con il gestore di pacchetti NPM o Yarn.

Configurazione di un server di autenticazione/autorizzazione locale

La tua applicazione di backend è responsabile sia della creazione di chat room che della generazione dei token di chat necessari all'SDK JS Chat per autenticare e autorizzare i client ad accedere alle tue chat room. È necessario utilizzare il proprio backend poiché non è possibile archiviare in modo sicuro le chiavi AWS in un'app mobile; gli aggressori sofisticati potrebbero estrarle e accedere al tuo account AWS.

Consulta Creazione di un token di chat nella Guida introduttiva ad Amazon IVS Chat. Come mostrato nel diagramma di flusso, l'applicazione lato server è responsabile della creazione di un token di chat. Ciò significa che l'app deve fornire i propri mezzi per generare un token di chat richiedendone uno dall'applicazione lato server.

In questa sezione, imparerai le basi della creazione di un provider di token nel tuo backend. Utilizziamo il framework rapido per creare un server locale live che gestisca la creazione di token di chat utilizzando l'ambiente AWS locale.

Crea un progetto `npm` vuoto usando NPM. Crea una directory che possa contenere la tua applicazione e rendila la tua directory di lavoro:

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

Utilizza `npm init` per creare un file `package.json` per la tua applicazione:

```
$ npm init
```

Questo comando richiede diverse voci, tra cui il nome e la versione dell'applicazione. Per ora, è sufficiente premere INVIO per accettare i valori predefiniti per la maggior parte di essi, con la seguente eccezione:

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

Premi INVIO per accettare il nome file predefinito suggerito per `index.js` o inserisci quello che desideri sia il nome del file principale.

A questo punto installa le dipendenze richieste:

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

`aws-sdk` richiede variabili di ambiente di configurazione che vengono caricate automaticamente da un file denominato `.env` situato nella directory principale. Per configurarlo, crea un nuovo file denominato `.env` e inserisci le informazioni di configurazione mancanti:

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

Ora creiamo un file entry-point nella directory principale con il nome che hai inserito sopra nelnpm `init` comando. In questo caso, utilizziamo `index.js` e importiamo tutti i pacchetti richiesti:

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

Ora crea una nuova istanza di `express`:

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

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

Dopodiché potrai creare il tuo primo metodo POST dell'endpoint per il provider di token. Individua i parametri richiesti nel corpo della richiesta (`roomId`, `userId`, `capabilities` e `sessionDurationInMinutes`):

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

Aggiungi la convalida dei campi obbligatori:

```
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;
  }
});
```

Dopo aver preparato il metodo POST, integriamo `aws-sdk` con `createChatToken` per le funzionalità di autenticazione/autorizzazione di base:

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

Alla fine del file, aggiungi un ascoltatore di porte per la tua app `express`:

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

A questo punto puoi eseguire il server con il seguente comando dalla root del progetto:

```
$ node index.js
```

Suggerimento: questo server accetta richieste di URL all'indirizzo https://localhost:3000.

Creazione di un progetto Chatterbox

Per prima cosa creiamo il progetto React chiamato `chatterbox`. Eseguire il comando:

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

Puoi integrare l'SDK JS per la messaggistica client di Chat tramite il gestore pacchetti del nodo o il gestore pacchetti Yarn:

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

Connessione a una chat room

Qui si crea una `ChatRoom` e ci si connette ad essa utilizzando metodi asincroni. La classe `ChatRoom` gestisce la connessione dell'utente a SDK JS Chat. Per connetterti correttamente a una chat room, devi fornire un'istanza di `ChatToken` all'interno della tua applicazione React.

Passa al file App creato nel progetto `chatterbox` predefinito ed elimina tutto ciò che si trova tra i due tag `<div>`. Non è necessario alcun codice precompilato. A questo punto, il nostro App è piuttosto vuoto.

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

import * as React from 'react';

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

Crea una nuova istanza `ChatRoom` e inviala allo stato usando l'hook `useState`. Richiede l'invio di `regionOrUrl` (la regione AWS in cui è ospitata la chat room) e `tokenProvider` (utilizzato per il flusso di autenticazione/autorizzazione del backend creato nei passaggi successivi).

Importante: devi utilizzare la stessa regione AWS in cui hai creato la stanza come descritto in Guida introduttiva ad Amazon IVS Chat. L'API è un servizio regionale AWS. Per un elenco delle regioni

supportate e degli endpoint del servizio HTTPS di Amazon IVS Chat, consulta la pagina delle regioni di 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>;
}
```

Creazione di un provider di token

Come passo successivo, dobbiamo creare una funzione `tokenProvider` senza parametri richiesta dal costruttore `ChatRoom`. Innanzitutto, creeremo una funzione `fetchChatToken` che effettuerà una richiesta POST all'applicazione di backend che hai configurato in the section called “Configurazione di un server di autenticazione/autorizzazione locale”. I token di chat contengono le informazioni necessarie all'SDK per stabilire con successo una connessione alla chat room. L'API di Chat utilizza questi token come metodo sicuro per convalidare l'identità di un utente, le funzionalità all'interno di una chat room e la durata della sessione.

Nella struttura di navigazione del progetto, crea un nuovo file TypeScript/JavaScript denominato `fetchChatToken`. Crea una richiesta di recupero per l'applicazione `backend` e restituisci l'oggetto `ChatToken` dalla risposta. Aggiungi le proprietà del corpo della richiesta necessarie per creare un token di chat. Usa le regole definite per il nome della risorsa Amazon (ARN). Queste proprietà sono documentate nell'endpoint CreateChatToken.

Nota: l'URL che stai utilizzando qui è lo stesso URL creato dal tuo server locale quando hai eseguito l'applicazione di backend.

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),
  };
}
```

Osservazione degli aggiornamenti della connessione

Reagire ai cambiamenti nello stato della connessione di una chat room è una parte essenziale della creazione di un'app di chat. Cominciamo con la sottoscrizione degli eventi pertinenti:

```
// 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>;
}
```

Successivamente, dobbiamo fornire la capacità di leggere lo stato della connessione. Usiamo il nostro hook `useState` per creare uno stato locale in `App` e impostare lo stato della connessione all'interno di ciascun ascoltatore.

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>;
}
```

Dopo esserti iscritto allo stato della connessione, visualizza lo stato e connettiti alla chat room usando il metodo `room.connect` all'interno dell'hook `useEffect`:

```
// 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>
);

// ...
```

Hai implementato correttamente una connessione alla chat room.

Creazione di un componente del pulsante di invio

In questa sezione viene creato un pulsante di invio con un design diverso per ogni stato della connessione. Il pulsante di invio facilita l'invio di messaggi in una chat room. Serve anche come indicatore visivo che indica se e quando è possibile inviare messaggi, ad esempio in caso di interruzioni di connessione o sessioni di chat scadute.

Per prima cosa, crea un nuovo file nella directory `src` del tuo progetto Chatterbox e assegnagli il nome `SendButton`. Quindi, crea un componente che mostrerà un pulsante per la tua applicazione di chat. Esporta il tuo `SendButton` e importalo nell'`App`. Nel `<div></div>` vuoto, aggiungi `<SendButton />`.

TypeScript

```
// SendButton.tsx

import React from 'react';

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

```
}

export const SendButton = ({ onPress, disabled }: Props) => {
  return (
    <button disabled={disabled} onClick={onPress}>
      Send
    </button>
  );
};

// App.tsx

import { SendButton } from './SendButton';

// ...

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

JavaScript

```
// SendButton.jsx

import React from 'react';

export const SendButton = ({ onPress, disabled }) => {
  return (
    <button disabled={disabled} onClick={onPress}>
      Send
    </button>
  );
};

// App.jsx

import { SendButton } from './SendButton';

// ...
```

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

Quindi, in App definisci una funzione denominata onMessageSend e passala alla proprietà SendButton onPress. Definisci un'altra variabile denominata isSendDisabled (che impedisce l'invio di messaggi quando la stanza non è connessa) e inviala alla proprietà 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>
);

// ...
```

Creazione dell'input di un messaggio

La barra dei messaggi di Chatterbox è il componente con cui interagirai per inviare messaggi a una chat room. In genere contiene un input di testo per comporre il messaggio e un pulsante per inviarlo.

Per creare un componente MessageInput, crea prima un nuovo file nella directory src e assegnagli il nome MessageInput. Quindi, crea un componente di input controllato che mostrerà un input per la tua applicazione di chat. Esporta il tuo MessageInput e importalo nell'App (sopra il <SendButton />).

Crea un nuovo stato denominato `messageToSend` usando l'hook `useState`, con una stringa vuota come valore predefinito. Nel corpo della tua app, invia `messageToSend` al `value` di`MessageInput` e invia `setMessageToSend` alla proprietà on`MessageChange`:

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>
);
```

Fasi successive

Ora che hai terminato con la creazione di una barra dei messaggi per Chatterbox, passa alla parte 2 di questo tutorial JavaScript, Messaggi ed eventi.

SDK per la messaggistica client di Chat IVS - Tutorial per JavaScript, parte 2: messaggi ed eventi

Questa seconda e ultima parte del tutorial è suddivisa in diverse sezioni:

Nota: in alcuni casi, gli esempi di codice per JavaScript e TypeScript sono identici, quindi vengono combinati.

Per la documentazione completa dell'SDK, inizia con l'SDK di messaggistica per client di chat Amazon IVS (qui nella Guida per l'utente di Chat Amazon IVS) e Documentazione di riferimento dell'SDK di messaggistica per client di chat per JavaScript su GitHub.

Prerequisito

Assicurati di aver completato la prima parte di questo tutorial, Chat room.

Sottoscrizione a eventi di messaggi di chat

L'istanza `ChatRoom` utilizza gli eventi per comunicare quando si verificano eventi in una chat room. Per iniziare a implementare l'esperienza di chat, devi mostrare ai tuoi utenti quando altri inviano un messaggio nella stanza a cui sono connessi.

Da qui, puoi effettuare la sottoscrizione a eventi di messaggistica della chat Successivamente, ti mostreremo come aggiornare un elenco di messaggi da te creato che viene aggiornato con ogni messaggio/evento.

Nell'App, all'interno dell'hook `useEffect`, sottoscrivi tutti gli eventi di messaggistica:

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

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

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

Visualizzazione dei messaggi ricevuti

La ricezione di messaggi è una parte fondamentale dell'esperienza di chat. Utilizzando l'SDK JS Chat, puoi configurare il tuo codice per ricevere facilmente eventi da altri utenti connessi a una chat room.

Successivamente, ti mostreremo come eseguire azioni in una chat room sfruttando i componenti qui creati.

Nella tua App, definisci uno stato denominato `messages` con un tipo di array `ChatMessage` denominato `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([]);

  //...
}
```

Successivamente, nella funzione dell'ascoltatore `message`, aggiungi `message` all'array `messages`:

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

// ...

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

// ...
```

Di seguito esaminiamo le attività da completare per mostrare i messaggi ricevuti:

1. the section called “Creazione di un componente di messaggio”
2. the section called “Riconoscimento dei messaggi inviati dall'utente corrente”
3. the section called “Creazione di un componente di elenco messaggi”
4. the section called “Rendering di un elenco di messaggi di chat”

Creazione di un componente di messaggio

Il componente `Message` è responsabile della visualizzazione del contenuto di un messaggio ricevuto dalla chat room. In questa sezione, crei un componente di messaggi per il rendering di singoli messaggi di chat nell'App.

Crea un nuovo file nella directory `src` e chiamalo `Message`. Inserisci il tipo `ChatMessage` di questo componente e passa la stringa `content` dalle proprietà `ChatMessage` per visualizzare il testo del

messaggio ricevuto dagli ascoltatori dei messaggi della chat room. Nella struttura di navigazione del progetto, passa a `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>
  );
};
```

Suggerimento: utilizza questo componente per archiviare diverse proprietà da rappresentare nelle righe dei messaggi, ad esempio URL di avatar, nomi utente e timestamp del momento in cui è stato inviato il messaggio.

Riconoscimento dei messaggi inviati dall'utente corrente

Per riconoscere il messaggio inviato dall'utente corrente, modifichiamo il codice e creiamo un contesto React per memorizzare l'`userId` dell'utente corrente.

Crea un nuovo file nella directory `src` e chiamalo `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>;
};
```

Nota: qui abbiamo usato l'hook `useState` per memorizzare il valore `userId`. In futuro potrai utilizzare `setUserId` per modificare il contesto dell'utente o per scopi di accesso.

Quindi, sostituisci `userId` il primo parametro inviato a `tokenProvider`, utilizzando il contesto creato in precedenza:

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

  // ...
}
```

Nel tuo componente `Message`, usa la variabile `UserContext` creata in precedenza, dichiara la variabile `isMine`, associa `userId` del mittente con `userId` del contesto e applica diversi stili di messaggi per l'utente corrente.

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>
  );
};
```

Creazione di un componente di elenco messaggi

Il componente `MessageList` è responsabile della visualizzazione della conversazione di una chat room nel tempo. Il file `MessageList` è il contenitore che contiene tutti i nostri messaggi. `Message` è una riga in `MessageList`.

Crea un nuovo file nella directory `src` e chiamalo `MessageList`. Definisci `Props` con `messages` di tipo array `ChatMessage`. All'interno del corpo, mappa la nostra proprietà `messages` e invia `Props` al componente `Message`.

TypeScript

```
// MessageList.tsx

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

interface Props {
  messages: ChatMessage[];
}
```

```
export const MessageList = ({ messages }: Props) => {
  return (
    <div>
      {messages.map((message) => (
        <Message key={message.id} message={message}/>
      ))}
    </div>
  );
};
```

JavaScript

```
// MessageList.jsx

import React from 'react';
import { Message } from './Message';

export const MessageList = ({ messages }) => {
  return (
    <div>
      {messages.map((message) => (
        <Message key={message.id} message={message} />
      ))}
    </div>
  );
};
```

Rendering di un elenco di messaggi di chat

A questo punto inserisci il nuovo `MessageList` nel tuo componente App principale:

```
// 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>
);

// ...
```

Ora tutti i pezzi del puzzle per l'App sono a posto e puoi iniziare a renderizzare i messaggi ricevuti dalla chat room. Continua di seguito per scoprire come eseguire azioni in una chat room sfruttando i componenti appena creati.

Esecuzione di azioni in una chat room

L'invio di messaggi e l'esecuzione delle azioni dei moderatori all'interno di una chat room sono alcuni dei principali modi con cui interagire in una chat room. Qui imparerai come usare vari oggetti `ChatRequest` per eseguire azioni comuni in Chatterbox, come l'invio di un messaggio, l'eliminazione di un messaggio e la disconnessione di altri utenti.

Tutte le azioni in una chat room seguono uno schema comune: per ogni azione eseguita in una chat room, esiste un oggetto di richiesta corrispondente. Per ogni richiesta è presente un oggetto di risposta corrispondente che si riceve alla conferma della richiesta.

Se ai tuoi utenti vengono concesse le autorizzazioni corrette quando crei un token di chat, possono eseguire correttamente le azioni corrispondenti utilizzando gli oggetti della richiesta per vedere quali richieste puoi eseguire in una chat room.

Di seguito, spieghiamo come inviare un messaggio ed eliminare un messaggio.

Invio di un messaggio

La classe `SendMessageRequest` consente l'invio di messaggi in una chat room. Qui puoi modificare la tua `App` per inviare una richiesta di messaggio utilizzando il componente che hai creato in Creazione dell'input di un messaggio (nella parte 1 di questo tutorial).

Per iniziare, definisci una nuova proprietà booleana denominata `isSending` con l'hook `useState`. Usa questa nuova proprietà per attivare lo stato disabilitato dell'elemento HTML `button` usando la costante `isSendDisabled`. Nel gestore eventi per il tuo `SendButton`, cancella il valore per `messageToSend` e imposta `isSending` su true.

Poiché effettuerai una chiamata API da questo pulsante, l'aggiunta della proprietà booleana *`isSending`* consente di evitare che si verifichino più chiamate API contemporaneamente, disabilitando le interazioni utente con il *`SendButton`* fino al completamento della richiesta.

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

// ...

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

// ...

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

// ...

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

// ...
```

Prepara la richiesta creando una nuova istanza `SendMessageRequest` passando il contenuto del messaggio al costruttore. Dopo aver impostato gli stati `isSending` e `messageToSend`, chiama il metodo `sendMessage`, che invia la richiesta alla chat room. Infine, deseleziona il flag `isSending` quando ricevi la conferma o il rifiuto della richiesta.

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);
  }
};

// ...
```

Dai una chance a Chatterbox: prova a inviare un messaggio creandone una bozza con `MessageInput` e toccando quindi `SendButton`. Dovresti vedere il messaggio inviato renderizzato all'interno del `MessageList` creato in precedenza.

Eliminazione di un messaggio

Per eliminare un messaggio da una chat room, è necessario disporre delle funzionalità adeguate. Le funzionalità vengono concesse durante l'inizializzazione del token di chat utilizzato per l'autenticazione in una chat room. Ai fini di questa sezione, `ServerApp` della sezione Configurazione di un server di autenticazione/autorizzazione locale (nella parte 1 di questo tutorial) consente di specificare le funzionalità dei moderatori. Questa operazione viene eseguita nell'app utilizzando l'oggetto `tokenProvider` creato in Creazione di un provider di token (anch'esso nella parte 1 del tutorial).

Qui puoi modificare il `Message` aggiungendo una funzione per eliminare il messaggio.

Innanzitutto, apri `App.tsx` e aggiungi la funzionalità `DELETE_MESSAGE` (`capabilities` è il secondo parametro della funzione `tokenProvider`).

Nota: in questo modo `ServerApp` informa le API di IVS Chat che l'utente associato al token di chat risultante può eliminare i messaggi in una chat room. In una situazione reale, probabilmente avrai una logica di backend più complessa per gestire le funzionalità degli utenti nell'infrastruttura della tua app server.

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']),
  }),
);

// ...
```

Nei passaggi successivi, aggiorni il tuo `Message` in modo da visualizzare un pulsante di eliminazione.

Apri `Message` e definisci un nuovo stato booleano denominato `isDeleting` usando l'hook `useState` con un valore iniziale di `false`. Utilizzando questo stato, aggiorna i contenuti di `Button` in modo che siano diversi a seconda dello stato corrente di `isDeleting`. Disattiva il pulsante quando `isDeleting` è true; ciò ti impedisce di provare a effettuare due richieste di eliminazione del messaggio contemporaneamente.

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>
  );
};
```

Definisci una nuova funzione chiamata `onDelete` che accetta una stringa come uno dei suoi parametri e restituisce `Promise`. Nel corpo di chiusura dell'azione di `Button`, utilizza `setIsDeleting` per attivare la proprietà booleana `isDeleting` prima e dopo una chiamata a `onDelete`. Per il parametro string, inserisci l'ID del messaggio del componente.

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>
  );
};
```

Successivamente, aggiorna il `MessageList` per riflettere le ultime modifiche apportate al componente `Message`.

Apri `MessageList` e definisci una nuova funzione chiamata `onDelete` che accetta una stringa come parametro e restituisce `Promise`. Aggiorna il `Message` e passalo attraverso le proprietà di `Message`. Il parametro string nella nuova chiusura sarà l'ID del messaggio che desideri eliminare, che viene passato dal tuo `Message`.

TypeScript

```
// MessageList.tsx

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

interface Props {
  messages: ChatMessage[];
  onDelete(id: string): Promise<void>;
}

export const MessageList = ({ messages, onDelete }: Props) => {
  return (
    <>
```

```
      {messages.map((message) => (
        <Message key={message.id} onDelete={onDelete} content={message.content}
 id={message.id} />
      ))}
    </>
  );
};
```

JavaScript

```
// MessageList.jsx

import * as React from 'react';
import { Message } from './Message';

export const MessageList = ({ messages, onDelete }) => {
  return (
    <>
      {messages.map((message) => (
        <Message key={message.id} onDelete={onDelete} content={message.content}
 id={message.id} />
      ))}
    </>
  );
};
```

Successivamente, aggiorna il App per riflettere le ultime modifiche apportate al `MessageList`.

Quindi, in App definisci una funzione denominata on`DeleteMessage` e passala alla proprietà `MessageList onDelete`.

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>
);

// ...
```

Prepara una richiesta creando una nuova istanza di `DeleteMessageRequest`, passando l'ID messaggio pertinente al parametro del costruttore e una chiamata `deleteMessage` che accetti la richiesta preparata sopra:

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);
};

// ...
```

Successivamente, aggiorna lo stato di `messages` in modo che rifletta un nuovo elenco di messaggi che omette il messaggio appena eliminato.

Nell'hook `useEffect`, ascolta l'evento `messageDelete` e aggiorna il tuo array di stato `messages` eliminando il messaggio con un ID corrispondente al parametro `message`.

Nota: l'evento `messageDelete` potrebbe essere generato quando i messaggi vengono eliminati dall'utente corrente o da qualsiasi altro utente presente nella stanza. Gestirlo nel gestore eventi (anziché accanto alla richiesta `deleteMessage`) consente di unificare la gestione dell'eliminazione dei messaggi.

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

// ...

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

return () => {
```

```
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

A questo punto puoi eliminare gli utenti da una chat room nella tua app di chat.

Fasi successive

Come esperimento, prova a implementare altre azioni in una stanza, ad esempio la disconnessione di un altro utente.

SDK per la messaggistica per client di Chat IVS - Tutorial per React Native, parte 1: chat room

Questo è il primo di un tutorial a due parti. Scoprirai gli elementi essenziali per utilizzare l'SDK JavaScript per la messaggistica di client di chat Amazon IVS creando un'app funzionale completa con React Native. Chiameremo l'app Chatterbox.

Il pubblico di riferimento è costituito da sviluppatori esperti che non conoscono l'SDK di messaggistica di Amazon IVS Chat. È necessario conoscere i linguaggi di programmazione TypeScript e JavaScript e la libreria di React Native.

Per brevità, faremo riferimento all'SDK JavaScript di messaggistica del client Amazon IVS Chat come a SDK JS Chat.

Nota: in alcuni casi, gli esempi di codice per JavaScript e TypeScript sono identici, quindi vengono combinati.

Questa prima parte del tutorial è suddivisa in diverse sezioni:

1. the section called “Configurazione di un server di autenticazione/autorizzazione locale”
2. the section called “Creazione di un progetto Chatterbox”
3. the section called “Connessione a una chat room”
4. the section called “Creazione di un provider di token”
5. the section called “Osservazione degli aggiornamenti della connessione”
6. the section called “Creazione di un componente del pulsante di invio”

7. the section called “Creazione dell'input di un messaggio”
8. the section called “Fasi successive”

Prerequisiti

- Acquisisci familiarità con TypeScript JavaScript e la libreria React Native. Se non conosci React Native, scopri le nozioni basilari in Introduzione a React Native.
- Leggi e comprendi Nozioni di base su Chat IVS.
- Crea un utente AWS IAM con le capacità CreateChatToken e CreateRoom definite in una policy IAM esistente. Consultare Nozioni di base su Chat IVS.
- Assicurati che la chiavi di accesso segrete di questo utente siano archiviata in un file di credenziali AWS. Per istruzioni, consulta la Guida per l'utente di AWS CLI (in particolare la sezione Configurazione e impostazioni del file delle credenziali).
- Crea una chat room e salva il relativo ARN. Consultare Nozioni di base su Chat IVS. (Se non salvi l'ARN, potrai cercarlo in un secondo momento con la console o l'API di Chat.)
- Installa l'ambiente Node.js 14+ con il gestore di pacchetti NPM o Yarn.

Configurazione di un server di autenticazione/autorizzazione locale

La tua applicazione di backend è responsabile sia della creazione di chat room che della generazione dei token di chat necessari all'SDK JS Chat per autenticare e autorizzare i client ad accedere alle tue chat room. È necessario utilizzare il proprio backend poiché non è possibile archiviare in modo sicuro le chiavi AWS in un'app mobile; gli aggressori sofisticati potrebbero estrarle e accedere al tuo account AWS.

Consulta Creazione di un token di chat nella Guida introduttiva ad Amazon IVS Chat. Come mostrato nel diagramma di flusso, l'applicazione lato server è responsabile della creazione di un token di chat. Ciò significa che l'app deve fornire i propri mezzi per generare un token di chat richiedendone uno dall'applicazione lato server.

In questa sezione, imparerai le basi della creazione di un provider di token nel tuo backend. Utilizziamo il framework rapido per creare un server locale live che gestisca la creazione di token di chat utilizzando l'ambiente AWS locale.

Crea un progetto `npm` vuoto usando NPM. Crea una directory che possa contenere la tua applicazione e rendila la tua directory di lavoro:

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

Utilizza `npm init` per creare un file `package.json` per la tua applicazione:

```
$ npm init
```

Questo comando richiede diverse voci, tra cui il nome e la versione dell'applicazione. Per ora, è sufficiente premere INVIO per accettare i valori predefiniti per la maggior parte di essi, con la seguente eccezione:

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

Premi INVIO per accettare il nome file predefinito suggerito per `index.js` o inserisci quello che desideri sia il nome del file principale.

A questo punto installa le dipendenze richieste:

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

`aws-sdk` richiede variabili di ambiente di configurazione che vengono caricate automaticamente da un file denominato `.env` situato nella directory principale. Per configurarlo, crea un nuovo file denominato `.env` e inserisci le informazioni di configurazione mancanti:

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

Ora creiamo un file entry-point nella directory principale con il nome che hai inserito sopra nelnpm `init` comando. In questo caso, utilizziamo `index.js` e importiamo tutti i pacchetti richiesti:

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

Ora crea una nuova istanza di `express`:

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

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

Dopodiché potrai creare il tuo primo metodo POST dell'endpoint per il provider di token. Individua i parametri richiesti nel corpo della richiesta (`roomId`, `userId`, `capabilities` e `sessionDurationInMinutes`):

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

Aggiungi la convalida dei campi obbligatori:

```
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;
  }
});
```

Dopo aver preparato il metodo POST, integriamo `aws-sdk` con `createChatToken` per le funzionalità di autenticazione/autorizzazione di base:

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

Alla fine del file, aggiungi un ascoltatore di porte per la tua app `express`:

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

A questo punto puoi eseguire il server con il seguente comando dalla root del progetto:

```
$ node index.js
```

Suggerimento: questo server accetta richieste di URL all'indirizzo https://localhost:3000.

Creazione di un progetto Chatterbox

Crea innanzitutto il progetto React Native denominato `chatterbox`. Eseguire il comando:

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

In alternativa, crea un progetto expo con un modello TypeScript.

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

Puoi integrare l'SDK JS per la messaggistica client di Chat tramite il gestore pacchetti del nodo o il gestore pacchetti Yarn:

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

Connessione a una chat room

Qui si crea una `ChatRoom` e ci si connette ad essa utilizzando metodi asincroni. La classe `ChatRoom` gestisce la connessione dell'utente a SDK JS Chat. Per connetterti correttamente a una chat room, devi fornire un'istanza di `ChatToken` all'interno della tua applicazione React.

Passa al file App creato nel progetto `chatterbox` predefinito ed elimina tutto ciò che restituisce un componente funzionale. Non è necessario alcun codice precompilato. A questo punto, il nostro App è piuttosto vuoto.

TypeScript/JavaScript:

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

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

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

Crea una nuova istanza `ChatRoom` e inviala allo stato usando l'hook `useState`. Richiede l'invio di `regionOrUrl` (la regione AWS in cui è ospitata la chat room) e `tokenProvider` (utilizzato per il flusso di autenticazione/autorizzazione del backend creato nei passaggi successivi).

Importante: devi utilizzare la stessa regione AWS in cui hai creato la stanza come descritto in Guida introduttiva ad Amazon IVS Chat. L'API è un servizio regionale AWS. Per un elenco delle regioni supportate e degli endpoint del servizio HTTPS di Amazon IVS Chat, consulta la pagina delle regioni di 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>;
}
```

Creazione di un provider di token

Come passo successivo, dobbiamo creare una funzione `tokenProvider` senza parametri richiesta dal costruttore `ChatRoom`. Innanzitutto, creeremo una funzione `fetchChatToken` che effettuerà una richiesta POST all'applicazione di backend che hai configurato in the section called “Configurazione di un server di autenticazione/autorizzazione locale”. I token di chat contengono le informazioni necessarie all'SDK per stabilire con successo una connessione alla chat room. L'API di Chat utilizza questi token come metodo sicuro per convalidare l'identità di un utente, le funzionalità all'interno di una chat room e la durata della sessione.

Nella struttura di navigazione del progetto, crea un nuovo file TypeScript/JavaScript denominato `fetchChatToken`. Crea una richiesta di recupero per l'applicazione `backend` e restituisci l'oggetto `ChatToken` dalla risposta. Aggiungi le proprietà del corpo della richiesta necessarie per creare un token di chat. Usa le regole definite per il nome della risorsa Amazon (ARN). Queste proprietà sono documentate nell'endpoint CreateChatToken.

Nota: l'URL che stai utilizzando qui è lo stesso URL creato dal tuo server locale quando hai eseguito l'applicazione di backend.

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),
  };
}
```

Osservazione degli aggiornamenti della connessione

Reagire ai cambiamenti nello stato della connessione di una chat room è una parte essenziale della creazione di un'app di chat. Cominciamo con la sottoscrizione degli eventi pertinenti:

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>;
}
```

Successivamente, dobbiamo fornire la capacità di leggere lo stato della connessione. Usiamo il nostro hook `useState` per creare uno stato locale in App e impostare lo stato della connessione all'interno di ciascun ascoltatore.

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>;
}
```

Dopo esserti iscritto allo stato della connessione, visualizza lo stato e connettiti alla chat room usando il metodo `room.connect` all'interno dell'hook `useEffect`:

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,
  }
});

// ...
```

Hai implementato correttamente una connessione alla chat room.

Creazione di un componente del pulsante di invio

In questa sezione viene creato un pulsante di invio con un design diverso per ogni stato della connessione. Il pulsante di invio facilita l'invio di messaggi in una chat room. Serve anche come indicatore visivo che indica se e quando è possibile inviare messaggi, ad esempio in caso di interruzioni di connessione o sessioni di chat scadute.

Per prima cosa, crea un nuovo file nella directory `src` del tuo progetto Chatterbox e assegnagli il nome `SendButton`. Quindi, crea un componente che mostrerà un pulsante per la tua applicazione

di chat. Esporta il tuo `SendButton` e importalo nell'App. Nel `<View></View>` vuoto, aggiungi `<SendButton />`.

TypeScript

```
// SendButton.tsx

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

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

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

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

// App.tsx

import { SendButton } from './SendButton';

// ...

return (
```

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

JavaScript

```
// SendButton.jsx

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

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

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

// App.jsx

import { SendButton } from './SendButton';

// ...

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

```
  </SafeAreaView>
);
```

Quindi, in App definisci una funzione denominata onMessageSend e passala alla proprietà SendButton onPress. Definisci un'altra variabile denominata isSendDisabled (che impedisce l'invio di messaggi quando la stanza non è connessa) e inviala alla proprietà 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>
);

// ...
```

Creazione dell'input di un messaggio

La barra dei messaggi di Chatterbox è il componente con cui interagirai per inviare messaggi a una chat room. In genere contiene un input di testo per comporre il messaggio e un pulsante per inviarlo.

Per creare un componente MessageInput, crea prima un nuovo file nella directory src e assegnagli il nome MessageInput. Crea, quindi, un componente di input che visualizzerà un input per la tua applicazione di chat. Esporta il tuo MessageInput e importalo nell'App (sopra il <SendButton />).

Crea un nuovo stato denominato messageToSend usando l'hook useState, con una stringa vuota come valore predefinito. Nel corpo della tua app, invia messageToSend al value diMessageInput e invia setMessageToSend alla proprietà 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',
  }
});
```

Fasi successive

Una volta terminata la creazione di una barra dei messaggi per Chatterbox, passa alla Parte 2 di questo tutorial di React Native, Messaggi ed eventi.

SDK per la messaggistica per client di Chat IVS - Tutorial per JavaScript, parte 2: messaggi ed eventi

Questa seconda e ultima parte del tutorial è suddivisa in diverse sezioni:

Nota: in alcuni casi, gli esempi di codice per JavaScript e TypeScript sono identici, quindi vengono combinati.

Prerequisito

Assicurati di aver completato la prima parte di questo tutorial, Chat room.

Sottoscrizione a eventi di messaggi di chat

L'istanza `ChatRoom` utilizza gli eventi per comunicare quando si verificano eventi in una chat room. Per iniziare a implementare l'esperienza di chat, devi mostrare ai tuoi utenti quando altri inviano un messaggio nella stanza a cui sono connessi.

Da qui, puoi effettuare la sottoscrizione a eventi di messaggistica della chat Successivamente, ti mostreremo come aggiornare un elenco di messaggi da te creato che viene aggiornato con ogni messaggio/evento.

Nell'App, all'interno dell'hook `useEffect`, sottoscrivi tutti gli eventi di messaggistica:

TypeScript/JavaScript:

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

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

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

Visualizzazione dei messaggi ricevuti

La ricezione di messaggi è una parte fondamentale dell'esperienza di chat. Utilizzando l'SDK JS Chat, puoi configurare il tuo codice per ricevere facilmente eventi da altri utenti connessi a una chat room.

Successivamente, ti mostreremo come eseguire azioni in una chat room sfruttando i componenti qui creati.

Nella tua App, definisci uno stato denominato `messages` con un tipo di array `ChatMessage` denominato `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([]);

  //...
}
```

Successivamente, nella funzione dell'ascoltatore `message`, aggiungi `message` all'array `messages`:

TypeScript/JavaScript:

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

// ...

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

// ...
```

Di seguito esaminiamo le attività da completare per mostrare i messaggi ricevuti:

1. the section called “Creazione di un componente di messaggio”
2. the section called “Riconoscimento dei messaggi inviati dall'utente corrente”
3. the section called “Rendering di un elenco di messaggi di chat”

Creazione di un componente di messaggio

Il componente `Message` è responsabile della visualizzazione del contenuto di un messaggio ricevuto dalla chat room. In questa sezione, crei un componente di messaggi per il rendering di singoli messaggi di chat nell'App.

Crea un nuovo file nella directory `src` e chiamalo `Message`. Inserisci il tipo `ChatMessage` di questo componente e passa la stringa `content` dalle proprietà `ChatMessage` per visualizzare il testo del messaggio ricevuto dagli ascoltatori dei messaggi della chat room. Nella struttura di navigazione del progetto, passa a `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,
  },
});
```

Suggerimento: utilizza questo componente per archiviare diverse proprietà da rappresentare nelle righe dei messaggi, ad esempio URL di avatar, nomi utente e timestamp del momento in cui è stato inviato il messaggio.

Riconoscimento dei messaggi inviati dall'utente corrente

Per riconoscere il messaggio inviato dall'utente corrente, modifichiamo il codice e creiamo un contesto React per memorizzare l'`userId` dell'utente corrente.

Crea un nuovo file nella directory `src` e chiamalo `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;
```

Nota: qui abbiamo usato l'hook `useState` per memorizzare il valore `userId`. In futuro potrai utilizzare `setUserId` per modificare il contesto dell'utente o per scopi di accesso.

Sostituisci, quindi, `userId` nel primo parametro passato a `tokenProvider` utilizzando il contesto creato in precedenza. Assicurati di aggiungere la funzionalità `SEND_MESSAGE` al tuo provider di token, come specificato di seguito, in quanto è necessaria per l'invio di messaggi:

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']),
      }),
  );

  // ...
}
```

Nel tuo componente `Message`, usa la variabile `UserContext` creata in precedenza, dichiara la variabile `isMine`, associa `userId` del mittente con `userId` del contesto e applica diversi stili di messaggi per l'utente corrente.

TypeScript

```
// Message.tsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

type Props = {
  message: ChatMessage;
}

export const Message = ({ message }: Props) => {
  const userId = useUserContext();

  const isMine = message.sender.userId === userId;

  return (
    <View style={[styles.root, isMine && styles.mine]}>
      {!isMine && <Text>{message.sender.userId}</Text>}
      <Text style={styles.textContent}>{message.content}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
```

```
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
  mine: {
    flexDirection: 'row-reverse',
    backgroundColor: 'lightblue',
  },
});
```

JavaScript

```
// Message.jsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

export const Message = ({ message }) => {
  const userId = useUserContext();

  const isMine = message.sender.userId === userId;

  return (
    <View style={[styles.root, isMine && styles.mine]}>
      {!isMine && <Text>{message.sender.userId}</Text>}
      <Text style={styles.textContent}>{message.content}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
```

```
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
  mine: {
    flexDirection: 'row-reverse',
    backgroundColor: 'lightblue',
  },
});
```

Rendering di un elenco di messaggi di chat

A questo punto, elenca i messaggi utilizzando il componente `FlatList` e `Message`:

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>
);

// ...
```

Ora tutti i pezzi del puzzle per l'App sono a posto e puoi iniziare a renderizzare i messaggi ricevuti dalla chat room. Continua di seguito per scoprire come eseguire azioni in una chat room sfruttando i componenti appena creati.

Esecuzione di azioni in una chat room

L'invio di messaggi e l'esecuzione delle azioni dei moderatori sono alcune delle principali modalità di interazione con una chatroom. Qui imparerai come utilizzare vari oggetti di richiesta chat per eseguire azioni comuni in Chatterbox, ad esempio l'invio di un messaggio, l'eliminazione di un messaggio e la disconnessione di altri utenti.

Tutte le azioni in una chat room seguono uno schema comune: per ogni azione eseguita in una chat room, esiste un oggetto di richiesta corrispondente. Per ogni richiesta è presente un oggetto di risposta corrispondente che si riceve alla conferma della richiesta.

Se ai tuoi utenti sono offerte le funzionalità corrette quando crei un token di chat, possono eseguire correttamente le azioni corrispondenti utilizzando gli oggetti della richiesta per vedere quali richieste puoi eseguire in una chatroom.

Di seguito, spieghiamo come inviare un messaggio ed eliminare un messaggio.

Invio di un messaggio

La classe `SendMessageRequest` consente l'invio di messaggi in una chat room. Qui puoi modificare la tua `App` per inviare una richiesta di messaggio utilizzando il componente che hai creato in Creazione dell'input di un messaggio (nella parte 1 di questo tutorial).

Per iniziare, definisci una nuova proprietà booleana denominata `isSending` con l'hook `useState`. Usa questa nuova proprietà per attivare lo stato disabilitato dell'elemento `button` usando la costante `isSendDisabled`. Nel gestore eventi per il tuo `SendButton`, cancella il valore per `messageToSend` e imposta `isSending` su true.

Poiché effettuerai una chiamata API da questo pulsante, l'aggiunta della proprietà booleana *`isSending`* consente di evitare che si verifichino più chiamate API contemporaneamente, disabilitando le interazioni utente con il *`SendButton`* fino al completamento della richiesta.

Nota: l'invio di messaggi funziona solo se hai aggiunto la funzionalità `SEND_MESSAGE` al tuo provider di token, come descritto sopra in Riconoscimento dei messaggi inviati dall'utente corrente.

TypeScript/JavaScript:

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

// ...

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

// ...

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

```
};

// ...

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

// ...
```

Prepara la richiesta creando una nuova istanza `SendMessageRequest` passando il contenuto del messaggio al costruttore. Dopo aver impostato gli stati `isSending` e `messageToSend`, chiama il metodo `sendMessage`, che invia la richiesta alla chat room. Infine, deseleziona il flag `isSending` quando ricevi la conferma o il rifiuto della richiesta.

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);
  }
};

// ...
```

Dai una chance a Chatterbox: prova a inviare un messaggio creandone una bozza con `MessageBar` e toccando quindi `SendButton`. Dovresti vedere il messaggio inviato renderizzato all'interno del `MessageList` creato in precedenza.

Eliminazione di un messaggio

Per eliminare un messaggio da una chat room, è necessario disporre delle funzionalità adeguate. Le funzionalità vengono concesse durante l'inizializzazione del token di chat utilizzato per l'autenticazione in una chat room. Ai fini di questa sezione, `ServerApp` della sezione Configurazione di un server di autenticazione/autorizzazione locale (nella parte 1 di questo tutorial) consente di specificare le funzionalità dei moderatori. Questa operazione viene eseguita nell'app utilizzando l'oggetto `tokenProvider` creato in Creazione di un provider di token (anch'esso nella parte 1 del tutorial).

Qui puoi modificare il `Message` aggiungendo una funzione per eliminare il messaggio.

Innanzitutto, apri `App.tsx` e aggiungi la funzionalità `DELETE_MESSAGE` (`capabilities` è il secondo parametro della funzione `tokenProvider`).

Nota: in questo modo `ServerApp` informa le API di IVS Chat che l'utente associato al token di chat risultante può eliminare i messaggi in una chat room. In una situazione reale, probabilmente avrai una logica di backend più complessa per gestire le funzionalità degli utenti nell'infrastruttura della tua app server.

TypeScript/JavaScript:

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

// ...

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

// ...
```

Nei passaggi successivi, aggiorni il tuo `Message` in modo da visualizzare un pulsante di eliminazione.

Definisci una nuova funzione chiamata `onDelete` che accetta una stringa come uno dei suoi parametri e restituisce `Promise`. Per il parametro string, inserisci l'ID del messaggio del componente.

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',
  },
});
```

Successivamente, aggiorna il `renderItem` per riflettere le ultime modifiche apportate al componente `FlatList`.

Quindi, in App definisci una funzione denominata `handleDeleteMessage` e passala alla proprietà `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]);

// ...
```

Prepara una richiesta creando una nuova istanza di `DeleteMessageRequest`, passando l'ID messaggio pertinente al parametro del costruttore e una chiamata `deleteMessage` che accetti la richiesta preparata sopra:

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);
};

// ...
```

Successivamente, aggiorna lo stato di `messages` in modo che rifletta un nuovo elenco di messaggi che omette il messaggio appena eliminato.

Nell'hook `useEffect`, ascolta l'evento `messageDelete` e aggiorna il tuo array di stato `messages` eliminando il messaggio con un ID corrispondente al parametro `message`.

Nota: l'evento `messageDelete` potrebbe essere generato quando i messaggi vengono eliminati dall'utente corrente o da qualsiasi altro utente presente nella stanza. Gestirlo nel gestore eventi (anziché accanto alla richiesta `deleteMessage`) consente di unificare la gestione dell'eliminazione dei messaggi.

TypeScript/JavaScript:

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

A questo punto puoi eliminare gli utenti da una chat room nella tua app di chat.

Fasi successive

Come esperimento, prova a implementare altre azioni in una stanza, ad esempio la disconnessione di un altro utente.

SDK per la messaggistica per client di Chat IVS: procedure consigliate per React e React Native

Questo documento descrive le più importanti procedure di utilizzo dell'SDK di messaggistica di chat Amazon IVS per React e React Native. Queste informazioni saranno utili per creare funzionalità di chat tipiche all'interno di un'app React e ti forniranno informazioni necessarie per approfondire le parti più avanzate dell'SDK di messaggistica chat di IVS.

Creazione di un hook di inizializzazione di ChatRoom

La classe `ChatRoom` contiene metodi di chat fondamentali e ascoltatori per la gestione dello stato della connessione e l'ascolto di eventi, come la ricezione e l'eliminazione di un messaggio. Qui spieghiamo come archiviare correttamente le istanze di chat in un hook.

Implementazione

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 };
};
```

Nota: non utilizziamo il metodo `dispatch` dell'hook `setState` perché non puoi aggiornare i parametri di configurazione in modo immediato. L'SDK crea un'istanza una sola volta e non è possibile aggiornare il provider di token.

Importante: usa l'hook dell'inizializzatore `ChatRoom` una sola volta per inizializzare una nuova istanza di chatroom.

Esempio

TypeScript/JavaScript:

```
// ...

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

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

  // ...
};

// ...
```

Ascolto dello stato della connessione

Facoltativamente, puoi eseguire la sottoscrizione agli aggiornamenti dello stato della connessione nell'hook della chatroom.

Implementazione

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 };
};
```

Provider di istanze ChatRoom

Per utilizzare l'hook in altri componenti (per evitare il prop-drilling), puoi creare un provider di chatroom usando un `context` React.

Implementazione

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;
```

Esempio

Dopo la creazione del `ChatRoomProvider`, puoi utilizzare la tua istanza con `useChatRoomContext`.

Importante: colloca il provider al livello root solo se devi accedere al `context` tra la schermata della chat e gli altri componenti al centro, per evitare inutili ripetizioni dei rendering se stai ascoltando le connessioni. In alternativa, colloca il provider il più vicino possibile alla schermata della chat.

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

// ...
```

Creazione di un ascoltatore di messaggi

Per non perdere alcun messaggio in arrivo, devi eseguire la sottoscrizione a eventi `message` e `deleteMessage`. Di seguito è riportato un codice che fornisce messaggi chat per i tuoi componenti.

Importante: ai fini delle prestazioni, separiamo `ChatMessageContext` da `ChatRoomProvider`, in quanto potremmo ricevere molti rendering ripetuti quando l'ascoltatore dei messaggi della chat

aggiorna lo stato del suo messaggio. Ricordati di applicare `ChatMessageContext` nei componenti dove utilizzerai `ChatMessageProvider`.

Implementazione

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>;
};
```

Esempio in React

Importante: ricorda di eseguire il wrapping del container dei messaggi con `ChatMessagesProvider`. La riga `Message` è un componente di esempio che visualizza il contenuto di un messaggio.

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>
  );
};
```

Esempio in React Native

Per impostazione predefinita `ChatMessage` contiene `id`, che viene utilizzato automaticamente come chiavi React in `FlatList` per ogni riga, per cui non è necessario passare `keyExtractor`.

TypeScript

```
// MessageListContainer.tsx

import React from 'react';
import { ListRenderItemInfo, FlatList } from 'react-native';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useChatMessagesContext } from './ChatMessagesContext';

const MessageListContainer = () => {
  const messages = useChatMessagesContext();

  const renderItem = useCallback(({ item }: ListRenderItemInfo<ChatMessage>) =>
 <MessageRow />, []);

  return <FlatList data={messages} renderItem={renderItem} />;
};
```

JavaScript

```
// MessageListContainer.jsx

import React from 'react';
import { FlatList } from 'react-native';
import { useChatMessagesContext } from './ChatMessagesContext';

const MessageListContainer = () => {
  const messages = useChatMessagesContext();

  const renderItem = useCallback(({ item }) => <MessageRow />, []);

  return <FlatList data={messages} renderItem={renderItem} />;
};
```

Più istanze di chatroom in un'app

Se utilizzi più chatroom simultanee nella tua app, ti proponiamo di creare ogni provider per ogni chat e di utilizzarlo nel provider di chat. In questo esempio creiamo una chat Help Bot e Customer Help. Creiamo un provider per entrambi.

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>;
};
```

Esempio in React

Ora puoi utilizzare provider di chat differenti che utilizzano lo stesso `ChatRoomProvider`. In seguito, puoi riutilizzare lo stesso `useChatRoomContext` all'interno di ogni schermata/visualizzazione.

TypeScript/JavaScript:

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

const App = () => {
  return (
    <Routes>
```

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

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

Sicurezza di Chat Amazon IVS

La sicurezza del cloud in AWS ha la massima priorità. In quanto cliente AWS, puoi trarre vantaggio da un'architettura di data center e di rete progettata per soddisfare i requisiti delle aziende più esigenti a livello di sicurezza.

La sicurezza è una responsabilità condivisa tra AWS e l'utente. Il modello di responsabilità condivisa descrive questo come sicurezza del cloud e sicurezza nel cloud:

- La sicurezza del cloud: AWS è responsabile della protezione dell'infrastruttura che esegue servizi AWS nel cloud AWS. AWS fornisce anche servizi utilizzabili in maniera sicura. I revisori di terze parti testano e verificano regolarmente l'efficacia della sicurezza come parte dei programmi di conformità AWS.
- Sicurezza nel cloud: la tua responsabilità è determinata dal servizio AWS che utilizzi. L'utente è anche responsabile per altri fattori, tra cui la riservatezza dei dati, i requisiti dell'azienda e leggi e normative applicabili.

Questa documentazione aiuta a comprendere come applicare il modello di responsabilità condivisa quando si utilizza Chat Amazon IVS. Gli argomenti di seguito illustrano come configurare Chat Amazon IVS per soddisfare i propri obiettivi di sicurezza e conformità.

Argomenti

Protezione dei dati di Chat IVS

Per i dati inviati a Chat Amazon Interactive Video Service (IVS), sono disponibili le seguenti protezioni dei dati:

- Il traffico di Chat Amazon IVS utilizza WSS per proteggere i dati durante il trasporto.
- I token di Amazon IVS Chat sono crittografati utilizzando chiavi gestite dal cliente KMS.

Chat Amazon IVS non richiede di fornire dati dei clienti (utenti finali). Non sono presenti campi nelle chat room, negli input o nei gruppi di sicurezza di input per i quali è previsto che vengano forniti dati dei clienti (utenti finali).

Non inserire informazioni identificative sensibili come numeri di account dei clienti (utenti finali) in campi a formato libero come il campo Nome. Lo stesso vale quando utilizzi Amazon IVS tramite la console o l'API, la CLI AWS o gli SDK AWS. I dati immessi in Chat Amazon IVS o in altri servizi potrebbero essere inclusi nei log di diagnostica.

I flussi non sono crittografati end-to-end; un flusso può essere trasmesso in modo non crittografato internamente nella rete IVS per l'elaborazione.

Identity and Access Management in Chat IVS

AWS Identity and Access Management (IAM) è un servizio AWS che facilita all'amministratore di un account il controllo dell'accesso in alle risorse AWS in maniera sicura. Consulta Gestione dell'identità e dell'accesso in IVS nella Guida per l'utente dello streaming a bassa latenza di IVS.

Destinatari

Il modo in cui IAM viene utilizzato cambia a seconda delle operazioni da eseguire in Amazon IVS. Consulta la pagina Destinatari nella Guida per l'utente dello streaming a bassa latenza di IVS.

Come Amazon IVS funziona con IAM

Prima di poter effettuare richieste API Amazon IVS, devi creare uno o più identità IAM (utenti, gruppi e ruoli) e policy IAM, quindi collegare le policy a tali identità. Per la propagazione delle autorizzazioni sono necessari pochi minuti; fino ad allora, le richieste API vengono rifiutate.

Per una panoramica di alto livello della modalità con cui Amazon IVS utilizza IAM, consulta Servizi AWS utilizzati con IAM nella Guida per l'utente IAM.

Identità

Puoi creare identità IAM per fornire l'autenticazione a persone e processi nel tuo account AWS. I gruppi IAM sono raccolte di utenti IAM che è possibile gestire come una singola unità. Consulta Identità (utenti, gruppi e ruoli) nella Guida per l'utente di IAM.

Policy

Le policy sono documenti con autorizzazioni-policy JSON costituiti da elementi. Consulta la pagina Policy nella Guida per l'utente dello streaming a bassa latenza di IVS.

Chat Amazon IVS supporta tre elementi:

- Operazioni: le operazioni delle policy per Chat Amazon IVS utilizzano il prefisso `ivschat` prima dell'operazione. Ad esempio, per concedere a qualcuno l'autorizzazione per creare una chat room Chat Amazon IVS con il metodo API `CreateRoom` di Chat Amazon IVS, includi l'operazione `ivschat:CreateRoom` nella policy di quella persona. Le istruzioni della policy devono includere un elemento `Action` o `NotAction`.
- Risorse: la risorsa di chat room di Chat Amazon IVS dispone del seguente formato di ARN:

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

  Ad esempio, per specificare la chat room `VgNkEJgOVX9N` nell'istruzione, utilizza il seguente ARN:

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

  Alcune operazioni Chat Amazon IVS, ad esempio quelle per la creazione di risorse, non possono essere eseguite su una risorsa specifica. In questi casi, è necessario utilizzare il carattere jolly (*):

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

- Condizioni: Chat Amazon IVS supporta alcune chiavi di condizione globali: `aws:RequestTag`, `aws:TagKeys` e `aws:ResourceTag`.

È possibile utilizzare le variabili come segnaposto in una policy. Ad esempio, puoi concedere a un utente IAM l'autorizzazione per accedere a una risorsa solo se questo è stata taggata con il relativo nome utente IAM. Consulta Variabili e tag nella Guida per l'utente di IAM.

Amazon IVS fornisce policy gestite da AWS che possono essere utilizzate per concedere un insieme preconfigurato di autorizzazioni alle identità (sola lettura o accesso completo). È possibile scegliere di utilizzare policy gestite anziché basate sull'identità mostrate di seguito. Per i dettagli, consulta Policy gestite per Chat Amazon IVS.

Autorizzazione basata su tag Amazon IVS

Puoi collegare i tag alle risorse Chat Amazon IVS o inoltrarli in una richiesta a Chat Amazon IVS. Per controllare l'accesso basato su tag, fornisci informazioni sui tag nell'elemento condizione di una policy utilizzando le chiavi di condizione `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` o `aws:TagKeys`. Per ulteriori informazioni sull'applicazione di tag alle risorse Chat Amazon IVS, consulta la sezione Assegnazione di tag nella Documentazione di riferimento delle API di Amazon IVS.

Roles

Consulta le sezioni Ruoli IAM e Credenziali di sicurezza temporanee nella Guida per l'utente di IAM.

Un ruolo IAM è un'entità all'interno dell'account AWS che dispone di autorizzazioni specifiche.

Amazon IVS supporta l'uso di credenziali di sicurezza temporanee. Puoi utilizzare le credenziali temporanee per effettuare l'accesso utilizzando la federazione, assumere un ruolo IAM o assumere un ruolo tra più account. Puoi ottenere credenziali di sicurezza temporanee richiamando le operazioni dell'API AWS Security Token Service, ad esempio `AssumeRole` o `GetFederationToken`.

Accesso con privilegi e senza privilegi

Le risorse API hanno un accesso con privilegi. L'accesso alla riproduzione senza privilegi può essere configurato tramite canali privati; consulta la pagina Configurazione di canali privati IVS.

Best practice per le policy

Consulta la sezione Best practice IAM nella Guida per l'utente di IAM.

Le policy basate su identità sono molto potenti. Esse determinano se qualcuno può creare, accedere o eliminare risorse Amazon IVS nell'account. Queste operazioni possono comportare costi aggiuntivi per l'account AWS. Segui questi suggerimenti:

- Assegna il privilegio minimo: quando crei policy personalizzate, concedi solo le autorizzazioni richieste per eseguire un'attività. Inizia con un set di autorizzazioni minimo e concedi autorizzazioni aggiuntive quando necessario. Ciò è più sicuro che iniziare con autorizzazioni che sono troppo

permissive e cercare di limitarle in un secondo momento. In particolare, conserva `ivschat:*` per l'accesso amministratore; non utilizzarlo nelle applicazioni.

- Abilitare MFA (Multi-Factor Authentication) per operazioni sensibili: per una maggiore sicurezza, richiedi agli utenti IAM di utilizzare l'autenticazione a più fattori (MFA) per accedere a risorse o operazioni API sensibili.
- Utilizza le condizioni della policy per ulteriore sicurezza: per quanto possibile, definisci le condizioni per cui le policy basate su identità consentono l'accesso a una risorsa. Ad esempio, è possibile scrivere condizioni per specificare un intervallo di indirizzi IP consentiti dai quali deve provenire una richiesta. È anche possibile scrivere condizioni per consentire solo le richieste all'interno di un intervallo di date o ore specificato oppure per richiedere l'utilizzo di SSL o MFA.

Esempi di policy basate su identità

Utilizza la console Amazon IVS

Per accedere alla console Amazon IVS, devi disporre di un set minimo di autorizzazioni che consentono di elencare e visualizzare i dettagli sulle risorse Chat Amazon IVS nel tuo account AWS. Se crei una policy basata su identità più restrittiva delle autorizzazioni minime richieste, la console non funzionerà nel modo previsto per le identità associate a tale policy. Per garantire l'accesso alla console Amazon IVS, collega le seguenti policy alle identità (consulta Aggiunta e rimozione di autorizzazioni IAM nellaGuida per l'utente di IAM).

Le parti della policy riportata di seguito consentono l'accesso a:

- Tutti gli endpoint API di Chat Amazon IVS
- Le tue Service Quotas di Chat Amazon IVS
- Elenco delle lambda e aggiunta delle autorizzazioni per la lambda scelta per la moderazione di Amazon IVS Chat
- Amazon Cloudwatch per ottenere parametri per la sessione di chat

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

Policy basata su risorse per Amazon IVS Chat

Devi concedere al servizio Amazon IVS Chat l'autorizzazione per richiamare la risorsa lambda per revisionare i messaggi. A tal fine, segui le istruzioni in Utilizzo delle policy basate su risorse per AWS Lambda (nella Guida per lo sviluppatore di AWS Lambda) e compila i campi come specificato di seguito.

Per controllare l'accesso alla risorsa lambda, puoi utilizzare condizioni basate su:

- `SourceArn`: la nostra policy di esempio utilizza un carattere jolly (`*`) per consentire a tutte le stanze del tuo account di richiamare la risorsa lambda. Facoltativamente, puoi specificare una stanza nel tuo account per consentire solo a quella stanza di richiamare la risorsa lambda.
- `SourceAccount`: nella policy di esempio seguente, l'ID dell'account AWS è `123456789012`.

```
{
```

```
   "Version": "2012-10-17",
   "Statement": [
      {
         "Principal": {
            "Service": "ivschat.amazonaws.com"
         },
         "Action": [
            "lambda:InvokeFunction"
         ],
         "Effect": "Allow",
         "Resource": "arn:aws:lambda:us-west-2:123456789012:function:name",
         "Condition": {
            "StringEquals": {
               "AWS:SourceAccount": "123456789012"
            },
            "ArnLike": {
               "AWS:SourceArn": "arn:aws:ivschat:us-west-2:123456789012:room/*"
            }
         }
      }
   ]
}
```

Risoluzione dei problemi

Consulta la pagina Risoluzione dei problemi nella Guida per l'utente dello streaming a bassa latenza di IVS per informazioni sulla diagnosi e la risoluzione dei problemi comuni che possono verificarsi durante l'utilizzo di Chat Amazon IVS e IAM.

Policy gestite per Chat IVS

Una policy gestita da AWS; è una policy standalone che viene creata e amministrata da AWS. Consulta la pagina Policy gestite per Amazon IVS nella Guida per l'utente dello streaming a bassa latenza di IVS.

Utilizzo di ruoli collegati ai servizi per Chat IVS

Amazon IVS utilizza i ruoli collegati ai servizi di AWS IAM. Consulta la pagina Utilizzo di ruoli collegati ai servizi per Amazon IVS nella Guida per l'utente dello streaming a bassa latenza di IVS.

Registrazione e monitoraggio di Chat IVS

Per registrare le prestazioni e/o le operazioni, usa Amazon CloudTrail. Consulta la pagina Logging Amazon IVS API Calls with AWS CloudTrail nella Guida per l'utente dello streaming a bassa latenza di IVS.

Risposta agli eventi imprevisti in Chat IVS

Per rilevare o segnalare eventuali incidenti, puoi monitorare lo stato del tuo flusso tramite gli eventi Amazon EventBridge. Consulta Utilizzo di Amazon EventBridge con Amazon IVS per lo streaming a bassa latenza e per lo streaming in tempo reale.

Usa il Dashboard AWS Health per informazioni sull'integrità generale di Amazon IVS (per regione).

Resilienza di Chat IVS

Le API IVS utilizzano l'infrastruttura globale AWS e sono basate su Regioni e zone di disponibilità AWS. Consulta Resilienza di IVS nella Guida per l'utente dello streaming a bassa latenza di IVS.

Sicurezza dell'infrastruttura di Chat IVS

Come servizio gestito, Amazon IVS è protetto dalle procedure di sicurezza di rete globali AWS. Tali procedure sono descritte in Best practice per sicurezza, identità e conformità.

Chiamate API

Utilizza le chiamate API pubblicate AWS per accedere ad Amazon IVS tramite la rete. Consulta la pagina Chiamate API nella sezione Sicurezza dell'infrastruttura nella Guida per l'utente dello streaming a bassa latenza di IVS.

Amazon IVS Chat

L'acquisizione e la consegna dei messaggi di Amazon IVS Chat avviene tramite connessioni WSS crittografate al nostro edge. L'API Amazon IVS Messaging utilizza connessioni HTTPS crittografate. Come per lo streaming e la riproduzione video, è necessario utilizzare TLS versione 1.2 o successive e i dati di messaggistica possono essere trasmessi internamente non crittografati per l'elaborazione.

Service quotas di Chat IVS

Di seguito sono riportati i limiti e le service quotas per gli endpoint, le risorse e altre operazioni di Chat Amazon Interactive Video Service (IVS). Le service quotas (quote di servizio), a cui si fa riferimento anche come limiti, rappresentano il numero massimo possibile di risorse di servizio o operazioni per l'account AWS. In altre parole, questi limiti sono per account AWS se non diversamente indicato nella tabella. Consultare anche Service Quotas (Quote di Servizio) AWS.

Per connettersi a livello di programmazione a un servizio AWS, viene utilizzato un endpoint. Consultare anche Endpoint di servizio AWS.

Tutte le quote vengono applicate per regione.

Aumento delle quote di servizio

Per le quote regolabili, è possibile richiedere un aumento tramite la console AWS. Utilizzare la console per visualizzare informazioni anche sulle service quotas (quote di servizio).

Le quote tariffarie delle chiamate API non sono regolabili.

Quote tariffarie per le chiamate API

Tipo di endpoint	Endpoint	Predefinita
Messaggistica	`DeleteMessage`	100 TPS
Messaggistica	`DisconnectUser`	100 TPS
Messaggistica	`SendEvent`	100 TPS
Token della chat	`CreateChatToken`	200 TPS
Registrazione della configurazione	`CreateLoggingConfiguration`	3 TPS
Registrazione della configurazione	`DeleteLoggingConfiguration`	3 TPS

Tipo di endpoint	Endpoint	Predefinita
Registrazione della configurazione	`GetLoggingConfiguration`	3 TPS
Registrazione della configurazione	`ListLoggingConfigurations`	3 TPS
Registrazione della configurazione	`UpdateLoggingConfiguration`	3 TPS
Stanza	`CreateRoom`	5 TPS
Stanza	`DeleteRoom`	5 TPS
Stanza	`GetRoom`	5 TPS
Stanza	`ListRooms`	5 TPS
Stanza	`UpdateRoom`	5 TPS
Tag	`ListTagsForResource`	10 TPS
Tag	`TagResource`	10 TPS
Tag	`UntagResource`	10 TPS

Other Quotas (Altre quote)

Risorsa o funzionalità	Predefinita	Adattabile	Descrizione
Connessioni di chat simultanee	50.000	Sì	Il numero massimo di connessioni di chat simultanee per account, in tutte le stanze di una Regione AWS.
Configurazioni di registrazione	10	Sì	Il numero massimo di configurazioni di registrazione personalizzate che possono

Risorsa o funzionalità	Predefinita	Adattabile	Descrizione
			essere create per account nella Regione AWS corrente.
Periodo di timeout del gestore di revisione dei messaggi	200	No	Il periodo di timeout in millisecondi per tutti i gestori di revisione dei messaggi nella Regione AWS corrente. Se questo periodo viene superato, il messaggio viene consentito o negato a seconda del valore del campo `fallbackResult` configurato per il gestore di revisione dei messaggi.
Frequenza delle richieste DeleteMessage in tutte le stanze	100	Sì	Il numero massimo di richieste DeleteMessage che possono essere effettuate al secondo in tutte le stanze. Le richieste possono provenire dall'Amazon IVS Chat API (API di Amazon IVS Chat) o dall'Amazon IVS Chat Messaging API (API di Amazon IVS Chat Messaging) WebSocket.

Risorsa o funzionalità	Predefinita	Adattabile	Descrizione
Frequenza delle richieste DisconnectUser in tutte le stanze	100	Sì	Il numero massimo di richieste DisconnectUser che possono essere effettuate al secondo in tutte le stanze. Le richieste possono provenire dall'Amaz on IVS Chat API (API di Amazon IVS Chat) o dall'Amaz on IVS Chat Messaging API (API di Amazon IVS Chat Messaging) WebSocket.
Frequenza delle richieste di messaggistica per connessio ne	10	No	Il numero massimo di richieste di messaggistica al secondo che possono essere effettuate da una connessione chat.
Frequenza delle richieste SendMessage in tutte le stanze	1000	Sì	Il numero massimo di richieste SendMessage che possono essere effettuat e al secondo in tutte le stanze. Queste richieste provengono dall'Amazon IVS Chat Messaging (API di Amazon IVS Chat Messaging) WebSocket.

Risorsa o funzionalità	Predefinita	Adattabile	Descrizione
Frequenza delle richieste SendMessage per stanza	100	No (ma configurabile tramite l'API)	Il numero massimo di richieste SendMessage che possono essere effettuate al secondo per una qualsiasi stanza. Questo valore è configurabile con il campo `maximumMessageRatePerSecond` di CreateRoom e UpdateRoom. Queste richieste provengono dall'Amazon IVS Chat Messaging API (API di Amazon IVS Chat Messaging) WebSocket.
Stanze	50.000	Sì	Il numero massimo di chat room per account, per Regione AWS.

Integrazione di Service Quotas (Quote di Servizio) e parametri di utilizzo di CloudWatch

È possibile utilizzare CloudWatch per gestire in modo proattivo le Service Quotas tramite i parametri di utilizzo di CloudWatch. È possibile utilizzare questi parametri per visualizzare l'uso del servizio corrente su grafici e pannelli di controllo CloudWatch. I parametri di utilizzo di Chat Amazon IVS corrispondono alle Service Quotas di Chat Amazon IVS.

Esiste l'opportunità di utilizzare una funzione matematica dei parametri di CloudWatch per visualizzare le Service Quotas per tali risorse sui grafici. È possibile, inoltre, configurare gli allarmi che avvisano quando l'uso si avvicina a una quota di servizio.

Per accedere ai parametri di utilizzo:

1. Aprire la console Service Quotas all'indirizzo https://console.aws.amazon.com/servicequotas/
2. Nel riquadro di navigazione, selezionare Servizi AWS.

3. Dall'elenco dei servizi AWS, cerca e seleziona Chat Amazon Interactive Video Service.
4. Nell'elenco Service Quotas, selezionare la quota di servizio desiderata. Verrà aperta una nuova pagina con informazioni sulla quota e sui parametri del servizio.

In alternativa, si può accedere a questi parametri dalla console CloudWatch. In Spazi dei nomi AWS, selezionare Utilizzo. Quindi, dall'elenco Servizio, seleziona Chat IVS. Consulta la pagina Monitoraggio di Chat Amazon IVS.

Nello spazio dei nomi AWS/Utilizzo, Chat Amazon IVS fornisce il seguente parametro:

Nome parametro	Descrizione
`ResourceCount`	Il numero delle risorse specificate in esecuzione nel account. Le risorse sono definite dalle dimensioni associate al parametro. Statistica valida: massimo (il numero massimo di risorse utilizzate durante un periodo di 1 minuto).

Le seguenti dimensioni vengono utilizzate per perfezionare i parametri di utilizzo:

Dimensione	Descrizione
Servizio	Il nome del servizio AWS contenente la risorsa. Valore valido: `IVS Chat`.
Classe	La classe della risorsa monitorata. Valore valido: `None`.
Tipo	Il tipo di risorsa monitorata. Valore valido: `Resource`.
Risorsa	Il nome della risorsa AWS. Valore valido: `ConcurrentChatConnections` . Il parametro di utilizzo ConcurrentChatConnections è una copia di quello presente nello spazio dei nomi AWS/IVSChat (con dimensione Nessuna), come descritto in Monitoraggio di Chat Amazon IVS.

Creazione di un allarme CloudWatch per i parametri di utilizzo

Per creare un allarme CloudWatch basato su un parametro di utilizzo di Chat Amazon IVS:

1. Dalla console Service Quotas (Quote di Servizio), selezionare la quota di servizio desiderata come descritto in precedenza. Al momento gli allarmi possono essere creati solo per ConcurrentChatConnections.
2. Nella sezione Allarmi Amazon CloudWatch selezionare Crea allarme.
3. Dall'elenco a discesa Soglia di allarme selezionare la percentuale del valore della quota applicata che si desidera impostare come valore per l'allarme.
4. In Nome dell'allarme, specificare un nome per l'allarme.
5. Selezionare Crea.

Risoluzione di problemi di Chat IVS

Questo documento descrive le best practice e i suggerimenti per la risoluzione dei problemi per Chat Amazon Interactive Video Service (IVS). I fenomeni correlati a IVS Chat spesso sono diversi da quelli correlati ai video IVS. Per ulteriori informazioni, consulta Nozioni di base su Chat Amazon IVS.

Argomenti:

Perché le connessioni chat IVS non sono state disconnesse quando la chatroom è stata eliminata?

Quando una risorsa della chatroom viene eliminata, se la chatroom viene utilizzata attivamente, i client di chat connessi alla chatroom non vengono disconnessi automaticamente. La connessione viene interrotta se/quando l'applicazione di chat aggiorna il token di chat. In alternativa, è necessario eseguire una disconnessione manuale di tutti gli utenti per rimuovere tutti gli utenti dalla chatroom.

Glossario IVS

Consulta anche il glossario AWS. Nella tabella seguente, LL sta per IVS streaming a bassa latenza; RT per IVS streaming in tempo reale.

Termine	Descrizione	LL	RT	Chat
AAC	Codifica audio avanzata. AAC è uno standard di codifica audio per la compressione audio digitale con perdita. Progettato per essere il successore del formato MP3, AAC generalmente raggiunge una qualità del suono superiore rispetto all'MP3 a parità di bitrate. AAC è stato standardizzato da ISO e IEC come parte delle specifiche MPEG-2 e MPEG-4.	✓	✓	
Streaming con bitrate adattivo	Lo streaming con bitrate adattivo (ABR) consente al lettore IVS di passare a un bitrate inferiore quando la qualità della connessione ne risente e tornare a un bitrate più elevato quando la qualità migliora.	✓		
Streaming adattivo	Consulta la codifica a livelli con simulcast.		✓	
Utente amministratore	Un utente AWS con accesso amministrativo a risorse e servizi disponibili in un account AWS. Consulta la Terminologia nella Guida per l'utente alla configurazione di AWS.	✓	✓	✓
ARN	Nome della risorsa Amazon, identifica in modo univoco una risorsa AWS. I formati specifici ARN dipendono dal tipo di risorsa. Per i formati ARN utilizzati dalle risorse IVS, vedere Guida di riferimento sull'autorizzazione del servizio.	✓	✓	✓
Proporzioni	Descrive le proporzioni tra larghezza e altezza del frame. Ad esempio, 16:9 è la proporzione corrispondente alla risoluzione Full HD o 1080p.	✓	✓	

Termine	Descrizione	LL	RT	Chat
Modalità audio	Una configurazione audio preimpostata o personalizzata ottimizzata per diversi tipi di utenti di dispositivi mobili e per le apparecchiature utilizzate. Consulta SDK di trasmissione IVS: modalità audio per dispositivi mobili (streaming in tempo reale).		✓	
AVC, H.264, MPEG-4 Parte 10	Codifica video avanzata, nota anche come H.264 o MPEG-4 Parte 10, uno standard di compressione video per la compressione video digitale con perdita.	✓	✓	
Sostituzione dello sfondo	Un tipo di filtro della fotocamera che consente ai creatori di streaming live di modificare lo sfondo. Consulta Sostituzione dello sfondo in SDK di trasmissione IVS: filtri di fotocamere di terze parti (streaming in tempo reale).		✓	
Bitrate	Un parametro di streaming per il numero di bit trasmessi o ricevuti al secondo.	✓	✓	
Trasmissione, emittente	Altri termini per stream, streamer.	✓		
Buffering	Una condizione che si verifica quando il dispositivo di riproduzione non è in grado di scaricare il contenuto prima del momento in cui dovrebbe iniziare la riproduzione. Il buffering può manifestarsi in vari modi: il contenuto può interrompersi e riavviarsi in modo casuale (c.d. "stuttering"), il contenuto può interrompersi per lunghi periodi di tempo (c.d. "freezing") o il player IVS può sospendere la riproduzione.	✓	✓	

Termine	Descrizione	LL	RT	Chat
Playlist con intervalli di byte	Una playlist più granulare rispetto alla playlist HLS standard. La playlist HLS standard è composta da file multimediali di 10 secondi. Con una playlist con intervallo di byte, la durata del segmento è la stessa dell'intervallo di fotogrammi chiave configurato per lo streaming. La playlist con intervallo di byte è disponibile solo per le trasmissioni registrate automaticamente su un bucket S3. Viene creata in aggiunta alla playlist HLS. Visualizza le playlist con intervallo di byte nella Registrazione automatica su Amazon S3 (streaming a bassa latenza).	✓		
CBR	Bitrate costante, un metodo di controllo della velocità per codificatori che mantiene un bitrate costante durante l'intera riproduzione di un video, indipendentemente da ciò che accade durante la trasmissione. È possibile aggiungere interruzioni all'azione per ottenere il bitrate desiderato, mentre i picchi possono essere quantizzati regolando la qualità della codifica in modo che corrisponda al bitrate desiderato. Consigliamo vivamente di utilizzare CBR anziché VBR.	✓	✓	
CDN	Rete per la distribuzione di contenuti, una soluzione distribuita geograficamente che ottimizza la distribuzione di contenuti come lo streaming video avvicinandoli al luogo in cui si trovano gli utenti.	✓		

Termine	Descrizione	LL	RT	Chat
Canale	Una risorsa IVS che memorizza la configurazione per lo streaming, tra cui un server di acquisizione, una chiave di streaming, un URL di riproduzione e opzioni di registrazione. Gli streamer utilizzano la chiave di flusso associata a un canale per avviare una trasmissione. Tutti i parametri e gli eventi generati durante una trasmissione sono associati a una risorsa del canale.	✓		
Tipo di canale	Determina la risoluzione e la frequenza fotogrammi per il canale. Consulta Tipi di canale nella Documentazione di riferimento delle API di streaming a bassa latenza IVS.	✓		
Log di chat	Un'opzione avanzata che può essere abilitata associando una configurazione di registrazione a una chat room.			✓
Chat room	Una risorsa IVS che memorizza la configurazione per una sessione di chat, incluse funzionalità opzionali come Revisione dei messaggi di chat e Log di chat. Consulta Passaggio 2: creazione di una chat room nella Guida introduttiva a IVS Chat.			✓
Composizione lato client	Utilizza un dispositivo host per mixare i flussi audio e video dei partecipanti alla fase e quindi li invia come flusso composito a un canale IVS. Ciò consente un maggiore controllo sull'aspetto della composizione a scapito di un maggiore utilizzo delle risorse del client e di un rischio maggiore che un problema relativo a fase host influisca sugli spettatori. Consulta anche Composizione lato server.	✓	✓	
CloudFront	Un servizio CDN fornito da Amazon.	✓		

Termine	Descrizione	LL	RT	Chat
CloudTrail	Un servizio AWS per la raccolta, il monitoraggio, l'analisi e la conservazione di eventi e attività degli account da AWS e fonti esterne. Consulta Registrazione delle chiamate API IVS con AWS CloudTrail.	✓	✓	✓
CloudWatch	Un servizio AWS per il monitoraggio delle applicazioni, la risposta ai cambiamenti delle prestazioni, l'ottimizzazione dell'uso delle risorse e la fornitura di informazioni sullo stato operativo. Puoi usare CloudWatch per monitorare i parametri IVS; consulta Monitoraggio dello streaming in tempo reale di IVS e Monitoraggio dello streaming a bassa latenza di IVS.	✓	✓	✓
Composizione	Il processo di combinazione di flussi audio e video da più fonti in un unico flusso.	✓	✓	
Pipeline di composizione	Una sequenza di fasi di elaborazione necessarie per combinare più flussi e codificare il flusso risultante.	✓	✓	
Compressione	Codifica delle informazioni utilizzando un numero inferiore di bit rispetto alla rappresentazione originale. Qualsiasi compressione particolare è priva di perdita o con perdita. La compressione priva di perdita di dati riduce i bit, identificando ed eliminando la ridondanza statistica. Nessuna informazione viene persa nella compressione priva di perdita di dati. La compressione con perdita di dati riduce i bit, rimuovendo informazioni non necessarie o meno importanti.	✓	✓	

Termine	Descrizione	LL	RT	Chat
Piano di controllo (control-plane)	Memorizza informazioni sulle risorse IVS come canali, fasi o chat room e fornisce interfacce per la creazione e la gestione di tali risorse. È regionale (basato sulle regioni AWS).	✓	✓	✓
CORS	La funzionalità di condivisione delle risorse multiorigine definisce un metodo con cui le applicazioni Web dei clienti caricate in un dominio possono interagire con le risorse situate in un dominio differente, come bucket S3. L'accesso può essere configurato in base a intestazioni, metodi HTTP e domini di origine. Consulta Utilizzo delle funzionalità di condivisione di risorse multiorigine (CORS) - Amazon Simple Storage Service nella Guida per l'utente di Amazon Simple Storage Service.	✓		
Origine di immagini personalizzate	Un'interfaccia fornita dall'SDK di trasmissione IVSche consente a un'applicazione di fornire il proprio input di immagini anziché limitarsi alle fotocamere preimpostate.	✓	✓	
Piano dati	L'infrastruttura che trasporta i dati dall'acquisizione all'uscita. Funziona in base alla configurazione gestita nel piano di controllo e si limita a una regione AWS.	✓	✓	✓
Encoder, encoding	Il processo di conversione di contenuti video e audio in un formato digitale, adatto allo streaming. La codifica può essere basata su hardware o software.	✓	✓	

Termine	Descrizione	LL	RT	Chat
Evento	Una notifica automatica pubblicata da IVS al servizio di monitoraggio AmazonEventBridge. Un evento rappresenta una modifica dello stato o dello stato di salute di una risorsa di streaming, ad esempio uno stage o una pipeline di composizione. Consulta Utilizzo di Amazon EventBridge con IVS per lo streaming a bassa latenza e Utilizzo di Amazon EventBridge con IVS per lo streaming in tempo reale.	✓	✓	✓
FFmpeg	Un progetto software gratuito e open source costituito da una suite di librerie e programmi per la gestione di file e streaming video e audio. FFmpeg offre una soluzione multipiattaforma per registrare, convertire e trasmettere audio e video.	✓		
Streaming frammentato	Creato quando una trasmissione si disconnette e riconnette entro l'intervallo specificato nella configurazione di registrazione del canale. I flussi multipli risultanti vengono considerati un'unica trasmissione e uniti in un singolo flusso registrato. Consulta Unisci flussi frammentati nella Registrazione automatica su Amazon S3 (streaming a bassa latenza).	✓		
Frequenza fotogrammi	Un parametro di streaming per il numero di frame video trasmessi o ricevuti al secondo.	✓	✓	
HLS	HTTP Live Streaming (HLS), un protocollo di comunicazione di streaming bitrate adattivo basato su HTTP utilizzato per fornire streaming IVS agli spettatori.	✓		

Termine	Descrizione	LL	RT	Chat
Playlist HLS	Un elenco di segmenti multimediali che compongono uno streaming. Le playlist HLS standard sono composte da file multimediali di 10 secondi. HLS supporta anche playlist con intervallo di byte più granulari.	✓		
Host	Un partecipante all'evento in tempo reale che invia video e/o audio alla fase.		✓	
IAM	Identity and Access Management, un servizio AWS che consente agli utenti di gestire in modo sicuro le identità e l'accesso ai servizi e alle risorse AWS, incluso IVS.	✓	✓	✓
Acquisizione	Processo IVS per la ricezione di flussi video da un host o emittente per l'elaborazione o la distribuzione a spettatori o altri partecipanti.	✓	✓	
Server di acquisizione	Riceve i flussi video e li invia a un sistema di transcodifica, dove gli streaming vengono transmixati o transcodificati in HLS per essere consegnati agli spettatori. I server per l'importazione sono componenti IVS specifici che ricevono flussi per i canali, insieme a un protocollo di ingestione (RTMP, RTMPS). Consulta le informazioni sulla creazione di un canale in Guida introduttiva allo streaming a bassa latenza di IVS.		✓	
Video interlacciato	Trasmette e visualizza solo righe pari o dispari dei fotogrammi successivi per creare un raddoppio percepito della frequenza fotogrammi senza consumare ulteriore larghezza di banda. Si sconsiglia di utilizzare video interlacciati a causa di problemi di qualità video.	✓	✓	

Termine	Descrizione	LL	RT	Chat
JSON	Javascript Object Notation, un formato di file standard aperto che utilizza testo leggibile dall'utente per trasmettere oggetti dati costituiti da coppie valore-attributo e tipi di dati array o qualsiasi altro valore serializzabile.	✓	✓	✓
Fotogramma chiave, fotogramma delta, intervallo di fotogrammi chiave	Il fotogramma chiave (noto anche come intracodificato o i-frame) è un fotogramma completo dell'immagine di un video. I fotogrammi successivi, i fotogrammi delta (detti anche fotogrammi previsti o p-frame), contengono solo le informazioni che sono state modificate. I fotogrammi chiave appariranno più volte all'interno di un flusso, a seconda dell'intervallo di fotogrammi chiave definito nell'encoder.	✓	✓	
Lambda	Un servizio AWS per l'esecuzione di codice (denominato funzioni Lambda) senza effettuare il provisioning di alcuna infrastruttura server. Le funzioni Lambda possono essere eseguite in risposta a eventi e richieste di chiamata o in base a una pianificazione. Ad esempio, IVS Chat utilizza le funzioni Lambda per consentire la revisione dei messaggi per una chat room.	✓	✓	✓
Latenza, latenza glass-to-glass	Un ritardo nel trasferimento dei dati. IVS definisce gli intervalli di latenza come: • Bassa latenza: meno di 3 sec • Latenza in tempo reale: inferiore a 300 ms Una latenza glass-to-glass si riferisce al ritardo da quando una fotocamera acquisisce un live streaming a quando il flusso appare sullo schermo di uno spettatore.	✓	✓	

Termine	Descrizione	LL	RT	Chat
Codifica a livelli con simulcast	Consente la codifica e la pubblicazione simultane e di più flussi video con diversi livelli di qualità. Consulta Streaming adattivo: codifica a più livelli con Simulcast nelle ottimizzazioni dello streaming in tempo reale.		✓	
Gestore di revisione dei messaggi	Consente ai clienti di IVS Chat di esaminare/filtrare automaticamente i messaggi di chat degli utenti prima che vengano recapitati alla chat room. Viene abilitato associando una funzione Lambda a una chat room. Consulta Creazione di una funzione Lambda di Gestore di revisione dei messaggi di chat.			✓
Mixer	Una funzionalità degli SDK di trasmissione IVS per dispositivi mobili che prende più sorgenti audio e video e genera un'unica uscita. Supporta la gestione degli elementi video e audio sullo schermo che rappresentano sorgenti come fotocamere, microfoni, catture dello schermo e audio e video generati dall'applicazione. L'output può quindi essere trasmesso in streaming a IVS. Consulta Configurazione di una sessione di trasmissione per la combinazione in SDK di trasmissione IVS: guida alla combinazione (streaming a bassa latenza).	✓		
Streaming su più host	Combina i flussi provenienti da più host in un unico flusso. Può essere ottenuto utilizzando una composizione lato client o lato server. Lo streaming su più host consente scenari, come invitare gli spettatori su un palco per domande e risposte, competizioni tra host, chat video e host che conversano tra loro di fronte a un vasto pubblico.		✓	

Termine	Descrizione	LL	RT	Chat
Playlist multivariante	Un indice di tutte le varianti di streaming disponibili per una trasmissione.	✓		
OAC	Origin Access Control, un meccanismo per limitare l'accesso a un bucket S3, in modo che contenuti come uno streaming registrato possano essere serviti solo tramite CloudFront CDN.	✓		
OBS	Open Broadcaster Software, software gratuito e open source per la registrazione video e lo streaming live. OBS offre un'alternativa (all'SDK di trasmissione IVS) per la pubblicazione per desktop. Gli streamer più sofisticati che hanno familiarità con OBS potrebbero preferirlo per le sue funzionalità di produzione avanzate, come le transizioni di scena, il mixaggio audio e la grafica di sovrapposizione.	✓	✓	
Partecipante	Un utente in tempo reale connesso alla fase come host o spettatore.		✓	
Token dei partecipanti	Autentica un partecipante all'evento in tempo reale quando partecipa a un palco. Un token partecipante controlla anche se un partecipante può inviare video allo stage.		✓	

Termine	Descrizione	LL	RT	Chat
Token di riproduzione, coppia di chiavi di riproduzione	Un meccanismo di autorizzazione che consente ai clienti di limitare la riproduzione di video su canali privati. I token di riproduzione vengono generati da una coppia di chiavi di riproduzione. Una coppia di chiavi di riproduzione è la coppia di chiavi pubblica-privata utilizzata per firmare e convalidare il token di autorizzazione dello spettatore per la riproduzione. Consulta Creazione o importazione di una chiave di riproduzione IVS in Configurazione dei canali privati IVS e vedi gli endpoint della coppia di chiavi di riproduzione nel riferimento alle API a bassa latenza IVS.	✓		
URL di riproduzione	Identifica l'indirizzo utilizzato dallo spettatore per avviare la riproduzione di un canale specifico. Questo indirizzo può essere utilizzato a livello globale. IVS seleziona automaticamente la migliore posizione sulla rete globale di distribuzione dei contenuti IVS per ogni spettatore per la distribuzione del video. Consulta le informazioni sulla creazione di un canale in Guida introduttiva allo streaming a bassa latenza di IVS.	✓		
Canale privato	Consente ai clienti di limitare l'accesso ai propri streaming utilizzando un meccanismo di autorizzazione basato su token di riproduzione. Consulta Flussi di lavoro per canali privati IVS in Configurazione dei canali privati IVS.	✓		
Video progressivo	Trasmette e visualizza tutte le linee di ogni fotogramma in sequenza. Si consiglia di utilizzare il video progressivo durante tutte le fasi di una trasmissione.	✓	✓	

Termine	Descrizione	LL	RT	Chat
Quote	Il numero massimo possibile di risorse o operazioni del servizio IVS per il tuo account AWS. In altre parole, questi limiti sono per account AWS se non diversamente indicato. Tutte le quote vengono applicate per regione. Consulta endpoint e quote di Amazon Interactive Video Service in Guida di riferimento generale di AWS.	✓	✓	✓
Regioni	Consentono di accedere ai servizi AWS che risiedono fisicamente in un'area geografica specifica. Le regioni forniscono la tolleranza ai guasti, la stabilità e la resilienza e possono anche ridurre la latenza. Con le Regioni, è possibile creare risorse ridondanti che restano disponibili e non influenzate da un'interruzione a livello regionale. La maggior parte delle richieste di servizi AWS è associata a una particolare regione geografica. Le risorse create in una regione non esistono in qualsiasi altra regione, a meno che non si utilizzi in modo esplicito una funzionalità di replica offerta da un servizio AWS. Ad esempio, Amazon S3 supporta la replica tra regioni. Alcuni servizi, ad esempio IAM, non dispongono di risorse regionali.	✓	✓	✓
Risoluzione	Descrive il numero di pixel in un singolo fotogramma video, ad esempio, Full HD o 1080p definisce un fotogramma con 1920x1080 pixel.	✓	✓	
Utente root	Il proprietario dell'account AWS. L'utente root ha accesso completo a tutte le risorse e i servizi AWS in tale account.	✓	✓	✓

Termine	Descrizione	LL	RT	Chat
RTMP, RTMPS	Real-Time Messaging Protocol, uno standard di settore per la trasmissione di audio, video e dati su una rete. RTMPS è la versione sicura di RTMP, in esecuzione su una connessione Transport Layer Security (TLS/SSL).	✓	✓	
Bucket S3	Una raccolta di oggetti archiviati in Amazon S3. Molte policy, tra cui l'accesso e la replica, sono definite a livello di bucket e si applicano a tutti gli oggetti del bucket. Ad esempio, una trasmissione IVS viene archiviata come più oggetti in un bucket S3.	✓		
SDK	Software Development Kit, una raccolta di librerie per gli sviluppatori che creano applicazioni con IVS.	✓	✓	✓
Segmentazione dei selfie	Consente di sostituire lo sfondo in uno streaming live, utilizzando una soluzione specifica del client che accetta l'immagine della telecamera come input e restituisce una maschera con punteggi di affidabilità per ogni pixel dell'immagine, indicando se è in primo piano o sullo sfondo. Consulta Sostituzione dello sfondo in SDK di trasmissione IVS: filtri di fotocamere di terze parti (streaming in tempo reale).		✓	
Versione semantica	Un formato di versione sotto forma di Major.Minor.Patch. Le correzioni di bug che non influiscono sull'API incrementano la versione della patch, le aggiunte/modifiche alle API compatibili con le versioni precedenti incrementano la versione secondaria e le modifiche dell'API incompatibili con le versioni precedenti incrementano la versione principale.	✓	✓	✓

Termine	Descrizione	LL	RT	Chat
Composizione lato server	Utilizza un server IVS per mixare audio e video dei partecipanti alla fase e quindi invia questo video misto a un canale IVS per raggiungere un pubblico più ampio o per archiviarlo in un bucket S3. La composizione lato server riduce il carico del client, migliora la resilienza della trasmissione e consente un uso più efficiente della larghezza di banda. Consulta anche Composizione lato client.		✓	
Quote del servizio	Un servizio che consente di gestire le quote per numerosi servizi da un'unica posizione. Oltre a cercare i valori delle quote, nella console Service Quotas è possibile richiedere anche un aumento delle quote.	✓	✓	✓
Ruolo collegato al servizio	Un unico tipo di ruolo IAM collegato direttamente a un servizio AWS. I ruoli collegati ai servizi sono creati automaticamente da IVS e includono tutte le autorizzazioni richieste dal servizio per eseguire chiamate agli altri servizi AWS per tuo conto, per esempio, per accedere a un bucket S3. Consulta Utilizzo dei ruoli collegati ai servizi per IVS in Sicurezza IVS.	✓		
Stage	Una risorsa IVS che rappresenta uno spazio virtuale in cui i partecipanti agli eventi in tempo reale possono scambiarsi video in tempo reale. Consulta Creazione di una fase in Giuda introduttiva allo streaming in tempo reale di IVS.		✓	
Sessione di fase	Inizia quando il primo partecipante si unisce a una fase e termina pochi minuti dopo che l'ultimo ha smesso di pubblicare nella fase. Una fase di lunga durata può avere più sessioni nel corso della sua durata.		✓	

Termine	Descrizione	LL	RT	Chat
Flusso	Dati che rappresentano contenuti video o audio inviati continuamente da un'origine a una destinazione.	✓	✓	
Chiave di streaming	Un identificatore assegnato da IVS quando si crea un canale, utilizzato per autorizzare lo streaming al canale. Tratta la chiave di streaming come un segreto, poiché consente a chiunque di trasmettere in streaming al canale. Consulta Guida introduttiva allo streaming a bassa latenza di IVS.	✓		
Starvation di flussi	Ritardo o interruzione della trasmissione dello streaming a IVS. Si verifica quando IVS non riceve la quantità di bit prevista che il dispositivo di codifica avrebbe dovuto inviare in un determinato intervallo di tempo. In caso di interruzione dello streaming, si verifica un evento di starvation di flussi. Dal punto di vista dei visualizzatori, starvation di flussi può apparire come ritardo, buffering o blocco di un video. Starvation di flussi può essere breve (meno di 5 secondi) o lunga (diversi minuti), a seconda della situazione specifica che ha provocato la starvation. Consulta Cos'è la starvation di flussi nelle Domande frequenti sulla risoluzione dei problemi.	✓	✓	
Streamer	Una persona o un dispositivo che invia un streaming video o audio a IVS.	✓	✓	
Sottoscrittore	Un partecipante all'evento in tempo reale che riceve video e/o audio degli organizzatori. Consulta Cos'è lo streaming in tempo reale IVS.		✓	

Termine	Descrizione	LL	RT	Chat
Tag	Un tag è un'etichetta che viene assegnata a una risorsa AWS. I tag consentono di identificare e organizzare le risorse AWS. Nella pagina iniziale della documentazione IVS, consulta "Applicazione di tag" in qualsiasi documentazione dell'API IVS (per lo streaming in tempo reale, lo streaming a bassa latenza o la chat).	✓	✓	✓
Filtri di fotocamere di terze parti	Componenti software che possono essere integrati con SDK di trasmissione IVS per consentire a un'applicazione di elaborare le immagini prima di fornirle un SDK di trasmissione come fonte di immagini personalizzata. Un filtro di terze parti può elaborare le immagini dalla fotocamera, applicare un effetto filtro, ecc.	✓	✓	
Anteprima	Un'immagine di dimensioni ridotte presa da uno streaming. Per impostazione predefinita, le miniature vengono generate ogni 60 secondi, ma è possibile configurare un intervallo più breve. La risoluzione delle miniature dipende dal tipo di canale. Consulta Registrazione di contenuti in Registrazione automatica su Amazon S3 (streamin g a bassa latenza).	✓		

Termine	Descrizione	LL	RT	Chat
Metadati temporizzati	Metadati legati a timestamp specifici all'interno di uno streaming. Possono essere aggiunti a livello di codice utilizzando l'API IVS e vengono associati a fotogrammi specifici. Ciò garantisce che tutti gli spettatori ricevano i metadati nello stesso punto dello streaming. I metadati temporizzati possono essere utilizzati per attivare azioni sul client, come l'aggiornamento delle statistiche della squadra durante un evento sportivo. Consulta Incorporamento di metadati all'interno di un flusso video.	✓		
Transcodifica	Converte video e audio da un formato all'altro. Un flusso in ingresso può essere transcodificato in un formato diverso a più bitrate e risoluzioni in modo da supportare una serie di dispositivi di riproduzione e diverse condizioni di rete.	✓	✓	
Transmuxing	Un semplice riconfezionamento di un flusso acquisito su IVS, senza ricodifica del flusso video. "Transmux" è l'abbreviazione di transcode multiplexing, un processo che cambia il formato di un file audio e/o video mantenendo alcuni o tutti i flussi originali. Il trasmuxing esegue la conversione in un formato container diverso senza modificare il contenuto del file. Diverso dalla transcodifica.	✓	✓	

Termine	Descrizione	LL	RT	Chat
Flussi di variante	Un insieme di codifiche della stessa trasmissione in diversi livelli di qualità distinti. Ogni flusso di variante è codificato come riproduzione HLS separata. Un indice dei flussi di variante disponibili viene definito riproduzione multivariante. Dopo che il lettore IVS ha ricevuto una riproduzione multivariante da IVS, può scegliere tra i flussi di variante durante la riproduzione, passando da uno all'altro senza interruzioni al variare delle condizioni della rete.	✓		
VBR	Variable Bitrate, un metodo di controllo della velocità per codificatori che utilizza un bitrate dinamico che cambia durante la riproduzione, a seconda del livello di dettaglio richiesto. Non usare VBR per motivi di qualità video; usa invece CBR.	✓	✓	

Termine	Descrizione	LL	RT	Chat
Vista	Una sessione di visualizzazione unica che sta attivamente scaricando o riproducendo un video. Le visualizzazioni sono la base per la quota di visualizzazioni simultanee. Una vista inizia quando una sessione di visualizz azione inizia la riproduzione video. Una vista termina quando una sessione di visualizzazione interrompe la riproduzione video. La riproduzione è l'unico indicatore dello spettatore; l'euristica del coinvolgimento come i livelli audio, la messa a fuoco delle schede del browser e la qualità video non vengono prese in considerazione. Quando si contano le viste, IVS non considera la legittimi tà dei singoli spettatori né tenta di deduplicare le visualizzazioni localizzate, ad esempio più lettori video su un singolo computer. Consulta Altre quote in Service Quotas (streaming a bassa latenza).	✓		
Visualizzatore	Una persona che riceve uno streaming da IVS.	✓		
WebRTC	Web Real-Time Communication, un progetto open source che fornisce ai browser Web e alle applicazi oni mobili comunicazioni in tempo reale. Consente alle comunicazioni audio e video di funzionare all'interno delle pagine Web per una comunicazione diretta tra pari, eliminando la necessità di installare plug-in o scaricare app native. Le tecnologie alla base di WebRTC sono implementate come standard web aperto e sono disponibili come normali API JavaScript in tutti i principali browser o come librerie per client nativi, come Android e iOS.	✓	✓	

Termine	Descrizione	LL	RT	Chat
WHIP	WebRTC-HTTP Ingestion Protocol, un protocollo basato su HTTP che consente l'importazione di contenuti basata su WebRTC in servizi di streaming e/o CDN. WHIP è una bozza IETF sviluppata per standardizzare l'importazione di WebRTC. WHIP fornisce la compatibilità con software come OBS, offrendo un'alternativa per la pubblicazione per desktop quando si utilizza l'SDK di trasmissione IVS. Gli streamer più sofisticati che hanno familiarità con OBS potrebbero preferirlo per le sue funzionalità di produzione avanzate, come le transizioni di scena, il mixaggio audio e la grafica di sovrapposizione WHIP è utile anche in situazioni in cui l'utilizzo dell'SDK di trasmissione IVS non è possibile o preferibile. Ad esempio, nelle configurazioni che coinvolgono codificatori hardware, l'utilizzo dell'SDK di trasmissione IVS potrebbe non essere possibile. Tuttavia, se il codificatore supporta WHIP, è comunque possibile pubblicare direttamente dal codificatore su IVS. Vedere Supporto WHIP per IVS (streaming in tempo reale)		✓	
WSS	WebSocket Secure, un protocollo per stabilire WebSocket su una connessione TLS crittografata. Viene utilizzato per la connessione agli endpoint di IVS Chat. Consulta Passaggio 4: inviare e ricevere il primo messaggio in Guida introduttiva alla chat di IVS.			✓

Cronologia dei documenti di Chat IVS

Le tabelle seguenti descrivono le modifiche importanti alla documentazione per la Chat Amazon IVS. Aggiorniamo frequentemente la documentazione per inserire le nuove release e tenendo conto dei feedback ricevuti.

Modifiche alla Guida per l'utente di Chat

Modifica	Descrizione	Data
Dividi un Chat UG	Questa release comprende le principali modifiche alla documentazione. Abbiamo spostato le informazioni sulla chat dalla Guida per l'utente dello streaming a bassa latenza di IVS a una nuova Guida per l'utente di Chat IVS, che si trova nella sezione Chat IVS esistente della pagina di destinazione della documentazione di IVS. Per altre modifiche alla documentazione, consulta Cronologia dei documenti (streaming a bassa latenza).	28 dicembre 2023
Glossario IVS	Ha esteso il glossario, coprendo i termini IVS in tempo reale, a bassa latenza e di chat.	20 dicembre 2023

Modifiche alla Documentazione di riferimento alle API di Chat IVS

Modifiche alle API	Descrizione	Data
Dividi un Chat UG	Con la presente versione è stata creata una Guida per l'utente di Chat IVS; pertanto, d'ora in avanti, le voci della Cronologia del documento relative alle attuali Documentazione di riferimento alle API di Chat IVS e Documentazione di riferimento alle API di messaggistica di Chat IVS saranno disponibili qui. La cronologia precedente per tali Documentazioni di riferimento alle API è descritta in Cronologia dei documenti (streaming a bassa latenza).	28 dicembre 2023

Note di rilascio di Chat IVS

Questo documento contiene tutte le note di rilascio della Chat Amazon IVS iniziando dalle più recenti, organizzate in base alla data di rilascio.

28 dicembre 2023

Guida per l'utente di Chat Amazon IVS

Chat Amazon Interactive Video Service (IVS) è una funzionalità gestita di chat in tempo reale per accompagnare gli stream video dal vivo. In questa versione, abbiamo spostato le informazioni sulla chat dalla Guida per l'utente dello streaming a bassa latenza di IVS a una nuova Guida per l'utente di Chat IVS. La documentazione è accessibile dalla pagina di destinazione della documentazione di Amazon IVS.

31 gennaio 2023

SDK di messaggistica per client di chat Amazon IVS: Android 1.1.0

Piattaforma	Download e modifiche
SDK di messaggistica per client di chat IVS per Android 1.1.0	Documentazione di riferimento: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/ • Per supportare le coroutine di Kotlin, abbiamo aggiunto nuove API di messaggistica per la chat IVS nel pacchetto com.amazonaws.ivs.chat.messaging.coroutines. Consulta anche il nuovo tutorial per le coroutine di Kotlin; la parte 1 (di 2) è dedicata alle Chat room.

Dimensione dell'SDK di Chat Client Messaging: Android

Architettura	Dimensione compressa	Dimensione non compressa
Tutte le architetture (bytecode)	89 KB	92 KB

9 novembre 2022

SDK di messaggistica client di Amazon IVS Chat: JavaScript 1.0.2

Piattaforma	Download e modifiche
SDK di messaggistica client chat JavaScript 1.0.2	Documentazione di riferimento: https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/ • È stato risolto un problema che riguardava Firefox: i client ricevevano erroneamente un errore del socket quando venivano disconnessi da una chat room utilizzando l'endpoint DisconnectUser.

8 settembre 2022

Amazon IVS Chat Client Messaging SDK (SDK di Amazon IVS Chat Client Messaging): Android 1.0.0 e iOS 1.0.0

Piattaforma	Download e modifiche
SDK per Android di Chat Client Messaging 1.0.0	Documentazione di riferimento: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/

Piattaforma	Download e modifiche
SDK di iOS Chat Client Messaging 1.0.0	Documentazione di riferimento: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/

Dimensione dell'SDK di Chat Client Messaging: Android

Architettura	Dimensione compressa	Dimensione non compressa
Tutte le architetture (bytecode)	53 KB	58 KB

Dimensione dell'SDK di Chat Client Messaging: iOS

Architettura	Dimensione compressa	Dimensione non compressa
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