aws

Guia do usuário do Chat

Amazon IVS

Amazon IVS: Guia do usuário do Chat

Table of Contents

O que é o Chat do Amazon IVS?

O Chat do Amazon IVS é um recurso gerenciado de chat ao vivo desenvolvido para acompanhar suas transmissões de vídeo ao vivo. A documentação está acessível na página inicial da documentação do Amazon IVS na seção Chat do Amazon IVS:

- Guia do usuário do Chat: este documento, junto com todas as outras páginas do Guia do usuário listadas no painel de navegação.
- Referência de API de chat: API do ambiente de gerenciamento (HTTPS).
- Referência da API do Chat Messaging: API de plano de dados (WebSocket).
- Referências do SDK para clientes de chat: Android, iOS e JavaScript.

Conceitos básicos do Amazon IVS Chat

O Amazon Interactive Video Service (IVS) Chat é um recurso gerenciado de chat ao vivo para acompanhar suas transmissões de vídeo ao vivo. (O Chat do IVS também poderá ser usado sem uma transmissão de vídeo.) É possível criar salas de chat e habilitar sessões de chat entre seus usuários.

Com o Amazon IVS Chat, você pode se concentrar na criação de experiências de chat personalizadas junto com vídeos ao vivo. Não é necessário gerenciar a infraestrutura ou desenvolver e configurar componentes de seus fluxos de trabalho de chat. O Amazon IVS Chat é escalável, seguro, confiável e econômico.

O Amazon IVS Chat funciona melhor para promover o envio de mensagens entre os participantes de uma transmissão de vídeo ao vivo com um início e um fim.

O restante deste documento orienta as etapas para criar sua primeira aplicação de chat usando o Amazon IVS Chat.

Exemplos: as seguintes aplicações de demonstração estão disponíveis (três exemplos de aplicações clientes e uma aplicação de servidor de backend para criação de tokens):

- Demonstração do Amazon IVS Chat Web
- Demonstração do Amazon IVS Chat para Android
- Demonstração do Amazon IVS Chat para iOS
- Demonstração do backend do Amazon IVS Chat

Importante: as salas de chat que não tiverem novas conexões ou atualizações num período de 24 meses serão excluídas automaticamente.

Tópicos

Etapa 1: realizar a configuração inicial

Antes de prosseguir, é necessário:

1. Criar uma conta da AWS.
2. Configurar os usuários raiz e administrativo.
3. Configurar as permissões do AWS IAM (Identity and Access Management). Usar a política especificada abaixo.

Para obter as etapas específicas para todo o exposto, consulte Conceitos básicos do streaming de baixa latência do IVS no Guia do usuário do Amazon IVS. Importante: na “Etapa 3: configurar permissões do IAM”, use esta política para o 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": "*"
      }
   ]
}
```

Etapa 2: criar uma sala de chat

Uma sala de chat do Amazon IVS tem informações de configuração associadas a ela (por exemplo, tamanho máximo da mensagem).

As instruções nesta seção mostram como usar o console ou a AWS CLI para configurar salas de chat (incluindo a configuração opcional para revisar e/ou registrar mensagens) e criar salas.

Instruções do console para criar uma sala de chat do IVS

Essas etapas são divididas em fases, começando pela configuração inicial da sala e terminando com a criação final da sala.

Se preferir, você pode configurar uma sala para que as mensagens sejam revisadas. Por exemplo, você pode atualizar o conteúdo da mensagem ou metadados, negar mensagens para evitar que sejam enviadas ou deixar a mensagem original passar. Isso é abordado em Configuração para revisar as mensagens da sala (opcional).

Opcionalmente, você também pode configurar uma sala para que as mensagens sejam registradas em log. Por exemplo, se você tiver mensagens sendo enviadas para uma sala de chat, poderá registrá-las em log em um bucket do Amazon S3, no Amazon CloudWatch ou no Amazon Kinesis Data Firehose. Isso é abordado em Configuração para registro de mensagens em log (opcional).

Configuração inicial da sala

1. Abra o console do Amazon IVS Chat.

   (Você também pode acessar o console do Amazon IVS via Console de Gerenciamento da AWS .)
2. Na barra de navegação, use o menu suspenso Select a Region (Selecione uma região) para escolher uma região. Sua nova sala será criada nessa região.
3. Na caixa Get started (Comece a usar), no canto superior direito, escolha Amazon IVS Chat Room (Sala do Amazon IVS Chat). Será exibida a janela Create room (Criar sala).

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. Em Setup (Configuração), opcionalmente, especifique um Room name (Nome da sala). Os nomes de sala não são exclusivos, mas oferecem um modo de distinguir salas diferentes do ARN (nome do recurso da Amazon) da sala.
5. Em Setup > Room configuration (Configuração > Configuração da sala), aceite a Default configuration (Configuração padrão) ou selecione Custom configuration (Configuração personalizada) e configure o Maximum message length (Tamanho máximo da mensagem) e/ou Maximum message rate (Taxa máxima de mensagens).
6. Se você quiser revisar as mensagens, continue com Configuração para revisar as mensagens da sala (opcional) abaixo. Caso contrário, pule essa parte (ou seja, aceite Message Review Handler > Disabled [Manipulador de revisão de mensagens > Desabilitado]) e prossiga diretamente para Final Room Creation (Criação da sala final).

Configuração para revisar as mensagens da sala (opcional)

1. Em Message Review Handler (Manipulador de revisão de mensagens), selecione Handle with AWS Lambda (Manipular com o AWS Lambda). A seção Message Review Handler (Manipulador de revisão de mensagem) se expandirá para exibir outras opções.
2. Configure Fallback result (Resultado de fallback) para Allow (Permitir) ou Deny (Negar) a mensagem se o manipulador não retornar uma resposta válida, encontrar um erro ou exceder o período de tempo limite.
3. Especifique a função Lambda existente ou use Create Lambda function (Criar função Lambda) para criar uma nova função.

   A função Lambda deve estar na mesma conta da AWS e nas mesmas regiões da AWS que a sala de chat. É necessário dar ao serviço Amazon Chat SDK permissão para invocar seu recurso do Lambda. A política baseada em recursos será criada automaticamente para a função Lambda selecionada. Para obter mais informações sobre permissões, consulte Política baseada em recursos para Chat do Amazon IVS.

Configuração para registro de mensagens em log (opcional)

1. Em Message logging (Registro de mensagens em log), selecione Automatically log chat messages (Registrar mensagens em log automaticamente). A seção Message logging (Registro de mensagens em log) se expandirá para exibir outras opções. É possível adicionar uma configuração de log existente a essa sala ou criar uma nova configuração de log selecionando Create logging configuration (Criar configuração de log).

2. Se você escolher uma configuração de log existente, um menu suspenso será exibido e mostrará todas as configurações de log que você já criou. Selecione uma na lista e suas mensagens de chat serão automaticamente registradas em log nesse destino.
3. Se você escolher Create logging configuration (Criar configuração de log), uma janela modal será exibida, permitindo que você crie e personalize uma nova configuração de log.
   a. Opcionalmente, especifique um Logging configuration name (Nome da configuração de log). Os nomes das configuração de log, como os nomes das sala, não são exclusivos, mas oferecem um modo de distinguir as configurações de log diferentes do ARN da configuração de log.
   b. Em Destination (Destino), selecione CloudWatch log group (Grupo de logs do CloudWatch), Kinesis firehose delivery stream (Stream de entrega do Kinesis Firehose) ou Amazon S3 bucket (Bucket do Amazon S3) para escolher o destino para seus logs.
   c. Dependendo do seu destino, selecione a opção de criar um novo CloudWatch log group (Grupo de logs do CloudWatch), Kinesis firehose delivery stream (Stream de entrega do Kinesis Firehose) ou Amazon S3 bucket (Bucket do Amazon S3), ou usar um já existente.
   d. Depois de revisar, escolha Create (Criar) para criar uma nova configuração de log com um ARN exclusivo. Isso anexa automaticamente a nova configuração de log à sala de chat.

Criação da sala final

1. Depois de revisar, escolha Create chat room (Criar sala de chat) para criar uma nova sala de chat com um ARN exclusivo.

Instruções da CLI para criar uma sala de chat do IVS

Este documento orienta você pelas etapas envolvidas na criação de uma sala de Chat do Amazon IVS usando a AWS CLI.

Criar uma sala de chat

A criação de uma sala de chat com a AWS CLI é uma opção avançada e exige que você baixe e configure a CLI em sua máquina primeiro. Para obter mais detalhes, consulte o Guia do usuário da Interface de Linhas de Comando da AWS.

1. Execute o comando `create-room` e envie um nome opcional:

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

2. Isso retorna uma nova sala de chat:

```
{
   "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. Observe o campo `arn`. Você precisará dele para criar um token de cliente e se conectar a uma sala de chat.

Definição de uma configuração de log (opcional)

Assim como na criação de uma sala de chat, a definição de uma configuração de log com a AWS CLI é uma opção avançada e exige que você baixe e configure a CLI em sua máquina primeiro. Para obter mais detalhes, consulte o Guia do usuário da Interface de Linhas de Comando da AWS.

1. Execute o comando `create-logging-configuration` de chat e insira um nome opcional e uma configuração de destino apontando para um bucket do Amazon S3 pelo nome. Esse bucket do Amazon S3 deve existir antes da criação da configuração de log. (Para obter detalhes sobre como criar um bucket do Amazon S3, consulte a Documentação do Amazon S3.)

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

2. Isso retornará uma nova configuração de log:

```
{
   "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. Observe o campo `arn`. Você precisará disso para anexar a configuração de log à sala de chat.

   a. Se você estiver criando uma nova sala de chat, execute o comando `create-room` e passe a configuração de log `arn`:

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

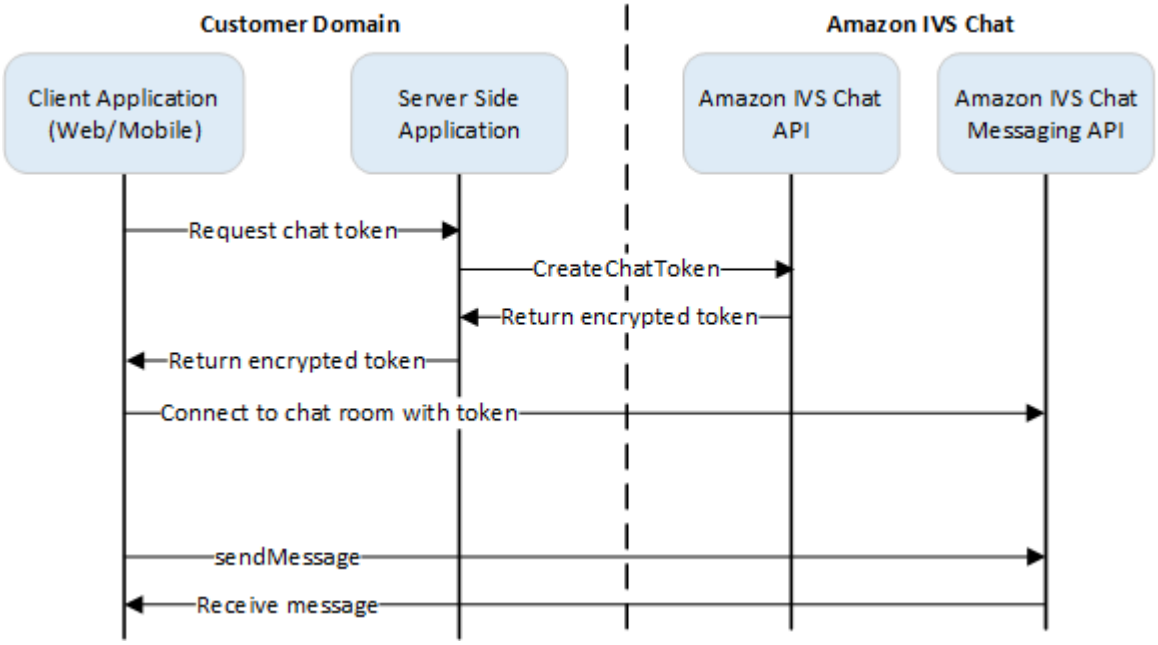
   b. Se você estiver atualizando uma sala de chat existente, execute o comando `update-room` e passe a configuração de log `arn`:

   ```
   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"
   ```

Etapa 3: criar uma token de chat

Para que um participante do chat se conecte a uma sala e comece a enviar e receber mensagens, um token de chat deve ser criado. Os tokens de chat são usados para autenticar e autorizar clientes de chat.

Este diagrama ilustra o fluxo de trabalho para a criação de um token de chat do IVS:



Conforme mostrado acima, um aplicativo cliente solicita um token ao aplicativo do lado do servidor, e o aplicativo do lado do servidor chama CreateChatToken usando um SDK da AWS ou solicitações assinadas pelo SIGv4. Como as credenciais da AWS são usadas para chamar a API, o token deve ser gerado em um aplicativo seguro do lado do servidor, não no aplicativo do lado do cliente.

Uma aplicação de servidor de backend que demonstra a geração de tokens está disponível em Demonstração de backend do Amazon IVS Chat.

A duração da sessão informa por quanto tempo uma sessão estabelecida pode permanecer ativa antes de ser encerrada automaticamente. Ou seja, a duração da sessão significa por quanto tempo o cliente pode permanecer conectado à sala de chat antes que um novo token seja gerado e seja necessário estabelecer uma nova conexão. Durante a criação do token, também é possível especificar a duração da sessão.

Cada token pode ser usado para estabelecer uma conexão somente uma vez. Se a conexão for encerrada, será necessário criar um novo token para que a conexão possa ser restabelecida. O token em si é válido até o timestamp de expiração do token incluído na resposta.

Quando um usuário final quiser se conectar a uma sala de chat, o cliente deve solicitar um token ao aplicativo do servidor. O aplicativo do servidor cria um token e o repassa para o cliente. Os tokens devem ser criados para usuários finais sob demanda.

Para criar um token de autenticação de dados, siga as instruções abaixo. Ao criar um token de chat, use os campos de solicitação para transmitir dados sobre o usuário final do chat e as funcionalidades

de mensagens do usuário final. Para obter mais detalhes, consulte CreateChatToken na Referência de API do Chat do IVS.

Instruções do AWS SDK

Criar um token de chat com o AWS SDK exige que você baixe e configure o SDK no aplicativo primeiro. As instruções para o AWS SDK usando JavaScript são descritas a seguir.

Importante: esse código deve ser executado no lado do servidor e sua saída transferida para o cliente.

Pré-requisito: para usar o exemplo de código abaixo, você precisa carregar o AWS JavaScript SDK em sua aplicação. Para obter mais detalhes, consulte .Conceitos básicos do AWS SDK for JavaScript.

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

Instruções da CLI

A criação de um token de chat com a AWS CLI é uma opção avançada e exige que você baixe e configure a CLI em sua máquina primeiro. Para obter mais detalhes, consulte o Guia do usuário da Interface de Linhas de Comando da AWS. Observação: gerar tokens com a AWS CLI é bom para fins de teste, mas para uso em produção, recomendamos gerar tokens no lado do servidor com o AWS SDK (consulte as instruções acima).

1. Execute o comando `create-chat-token` junto com o identificador da sala e o ID do usuário para o cliente. Inclua qualquer um destes recursos: `"SEND_MESSAGE"`, `"DELETE_MESSAGE"`, `"DISCONNECT_USER"`. (Opcionalmente, inclua a duração da sessão (em minutos) ou atributos personalizados (metadados) sobre a sessão de chat. Esses campos não são exibidos abaixo.)

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

2. Isso retorna um token de cliente:

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

3. Salve esse token. Você precisará dele para se conectar à sala de chat e enviar ou receber mensagens. Será necessário gerar outro token de chat antes que sua sessão termine (conforme indicado por `sessionExpirationTime`).

Etapa 4: enviar e receber sua primeira mensagem

Use o token do chat para se conectar a uma sala de chat e enviar sua primeira mensagem. Veja um exemplo de código JavaScript abaixo. Os SDKs do cliente do IVS também estão disponíveis: consulte Chat SDK: Android Guide, Chat SDK: iOS Guide e Chat SDK: JavaScript Guide.

Serviço regional: o exemplo de código abaixo se refere à “região de sua preferência compatível”. O Amazon IVS Chat oferece endpoints regionais que podem ser usados para fazer as solicitações. Para a API do Amazon IVS Chat Messaging, a sintaxe geral de um endpoint regional é:

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

Por exemplo, wss://edge.ivschat.us-west-2.amazonaws.com é o endpoint da região Oeste dos EUA (Oregon). Para obter uma lista das regiões compatíveis, consulte as informações sobre o Amazon IVS Chat na página do Amazon IVS na AWS General Reference (Referência geral da 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);
}
```

Parabéns, já está tudo pronto! Agora você tem uma aplicação de chat simples que pode enviar ou receber mensagens.

Etapa 5: verificar limites de cota de serviço (opcional)

Suas salas de chat serão escaladas junto com sua transmissão ao vivo do Amazon IVS, para permitir que todos os espectadores se envolvam em conversas por chat. Porém, todas as contas do Amazon IVS têm limites de número de participantes do chat simultâneo e na taxa de entrega de mensagens.

Certifique-se de que seus limites são adequados e, se necessário, solicite um aumento, especialmente se você estiver planejando um grande evento de streaming. Para obter detalhes, consulte Service Quotas (streaming de baixa latência), Service Quotas (streaming em tempo real) e Service Quotas (Chat).

Registro em log do Chat do IVS

O recurso de registro de logs de chat permite a você gravar todas as mensagens de chat em uma sala em qualquer um dos três locais padrão: um bucket do Amazon S3, no Amazon CloudWatch Logs ou no Amazon Kinesis Data Firehose. Posteriormente, os logs podem ser usados para análise ou criação de uma repetição de chat vinculada a uma sessão de vídeo ao vivo.

Ativar o log de chat para uma sala

O log de chat é uma opção avançada que pode ser ativada pela associação de uma configuração de log a uma sala. Uma configuração de log é um recurso que permite especificar um tipo de local (bucket do Amazon S3, Amazon CloudWatch Logs ou Amazon Kinesis Data Firehose) em que as mensagens de uma sala serão registradas em log. Para obter detalhes sobre como criar e gerenciar configurações de log, consulte Conceitos básicos do Amazon IVS Chat e Referência de API do Amazon IVS Chat.

É possível associar até três configurações de log a cada sala, seja ao criar uma nova sala (CreateRoom) ou ao atualizar uma sala existente (UpdateRoom). É possível associar várias salas à mesma configuração de log.

Quando pelo menos uma configuração ativa de log é associada a uma sala, todas as solicitações de mensagens enviadas para essa sala por meio da API Amazon IVS Chat Messaging são automaticamente registradas nos locais especificados. Estes são os atrasos médios de propagação (desde quando uma solicitação de mensagem é enviada até o momento em que ela se torna disponível em seus locais especificados):

- Bucket do Amazon S3: 5 minutos
- Amazon CloudWatch Logs ou Amazon Kinesis Data Firehose: 10 segundos

Conteúdo da mensagem

Formato

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

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

Campos

Campo	Descrição
`event_timestamp`	Carimbo de data/hora em UTC de quando a mensagem foi recebida pelo Amazon IVS Chat.
`payload`	A carga JSON de Mensagem (inscrever) ou Evento (inscrever) que os clientes receberão do serviço Amazon IVS Chat.
`type`	Tipo da mensagem de chat. • Valores válidos: `MESSAGE` | `EVENT`
`version`	Versão do formato do conteúdo da mensagem.

Bucket do Amazon S3

Formato

Os logs de mensagens são organizados e armazenados com o seguinte prefixo S3 e formato de arquivo:

```
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
```

Campos

Campo	Descrição
`<account_id>`	ID da conta da AWS a partir da qual a sala é criada.

Campo	Descrição
`<hash>`	Um valor de hash gerado pelo sistema para garantir exclusividade.
`<region>`	A região de serviço da AWS em que a sala foi criada.
`<resource_id>`	Parte de ID de recurso do ARN da sala.
`<version>`	Versão do formato do conteúdo da mensagem.
`<year> / <month> / <day> / <hours> / <minute>`	Carimbo de data/hora em UTC de quando a mensagem foi recebida pelo Amazon IVS Chat.

Exemplo

```
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

Os logs de mensagens são organizados e armazenados com o seguinte formato de nome de fluxo de logs:

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

Campos

Campo	Descrição
`<resource_id>`	Parte de ID de recurso do ARN da sala.
`<version>`	Versão do formato do conteúdo da mensagem.

Exemplo

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

Amazon Kinesis Data Firehose

Os logs de mensagem são enviados ao stream de entrega como dados de streaming em tempo real para destinos como o Amazon Redshift, Amazon OpenSearch Service, Splunk e quaisquer endpoints de HTTP personalizados de propriedade de provedores de serviços de terceiros compatíveis. Para obter mais informações, consulte O que é o Amazon Kinesis Data Firehose.

Restrições

- Você deve ser o proprietário do local de log em que as mensagens serão armazenadas.
- A sala, a configuração de log e o local do log devem estar na mesma região da AWS.
- Somente as configurações ativas de log estão disponíveis para o log de chat.
- Somente é possível excluir uma configuração de log que não esteja mais associada a nenhuma sala.

Para registrar em log mensagens em um local de sua propriedade, é necessária autorização com suas credenciais da AWS. Para dar ao IVS Chat o acesso necessário, uma política de recursos (para um bucket do Amazon S3 ou o CloudWatch Logs) ou uma função vinculada ao serviço (SLR) do AWS IAM (para o Amazon Kinesis Data Firehose) é gerada automaticamente quando a configuração de log é criada. Tenha cuidado com qualquer modificação na função ou nas políticas, pois isso pode afetar a permissão para o log de chat.

Monitoramento de erros com o Amazon CloudWatch

É possível monitorar erros que ocorrem no log de chat com o Amazon CloudWatch e criar alarmes ou painéis para indicar ou responder às alterações de erros específicos.

Há vários tipos de erros. Para obter mais informações, consulte Monitorar o Chat do Amazon IVS.

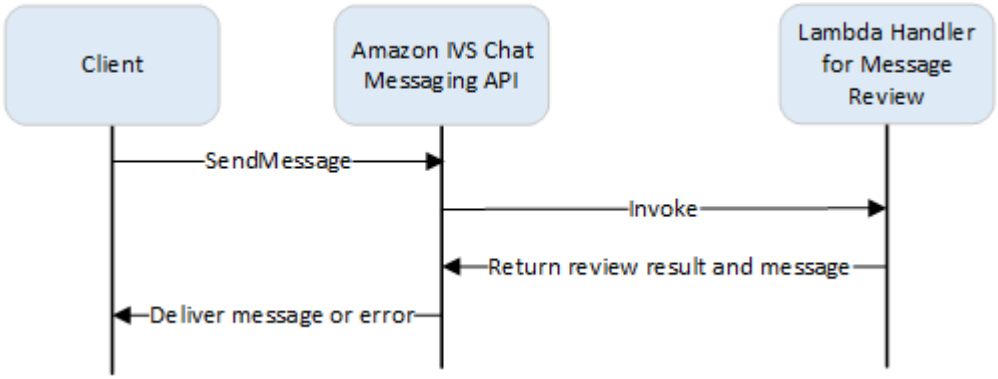
Manipulador de revisão de mensagem do Chat do IVS

Um manipulador de revisão de mensagens permite revisar ou modificar mensagens antes que elas sejam entregues em uma sala. Quando um manipulador de revisão de mensagens é associado a uma sala, ele é chamado para cada solicitação SendMessage para a sala. O manipulador aplica a lógica de negócios da aplicação e determina se deseja permitir, negar ou modificar uma mensagem. O Amazon IVS Chat oferece suporte a funções do AWS Lambda, como manipuladores.

Criação de uma função do Lambda

Antes de configurar um manipulador de revisão de mensagens para uma sala, é necessário criar uma função Lambda com uma política do IAM baseada em recursos. A função Lambda deve estar na mesma conta da AWS e na mesma região da AWS que a sala com a qual você usará a função. A política baseada em recurso dá ao Amazon IVS Chat permissão para invocar sua função Lambda. Para obter instruções, consulte Política baseada em recurso para o Chat do Amazon IVS.

Fluxo de trabalho



Sintaxe da Solicitação

Quando um cliente envia uma mensagem, o Amazon IVS Chat invoca a função Lambda com uma carga útil JSON:

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

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

Corpo da Solicitação

Campo	Descrição
`Attributes`	Atributos associados à mensagem.
`Content`	Conteúdo original da mensagem.
`MessageId`	O ID da mensagem. Gerada pelo IVS Chat.
`RoomArn`	O ARN da sala onde as mensagens são enviadas.
`Sender`	Informações sobre o remetente. Esse objeto tem vários campos: • `Attributes` : metadados sobre o remetente estabelecido durante a autenticação. Pode ser usado para fornecer ao cliente mais informações sobre o remetente; por exemplo, URL do avatar, selos, fonte e cor. • `UserId`: um identificador especificado pela aplicação do visualizador (usuário final) que enviou a mensagem. Pode ser usado pela aplicação cliente para se referir ao usuário na API de mensagens ou nos domínios da aplicação. • `Ip`: o endereço IP do cliente que está enviando a mensagem.

Sintaxe da resposta

O manipulador da função Lambda deve retornar uma resposta JSON com a sintaxe a seguir. As respostas que não correspondem à sintaxe abaixo ou não satisfazem as restrições de campo são inválidas. Nesse caso, a mensagem é permitida ou negada conforme o valor de `FallbackResult` especificado no manipulador de revisão de mensagens; consulte MessageReviewHandler na Referência da API do Amazon IVS Chat.

```
{
```

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

Campos de resposta

Campo	Descrição
`Attributes`	Atributos associados à mensagem retornada da função Lambda. Se `ReviewResult` for `DENY`, poderá ser fornecido um `Reason` em `Attributes`; por exemplo: `"Attributes": {"Reason": "denied for moderation` Nesse caso, o cliente remetente receberá um erro WebSocket 406 com o motivo na mensagem de erro. (Consulte Erros WebSocket na Referência da API de mensagens do Amazon IVS Chat.) • Restrições de tamanho: máximo de 1 KB • Obrigatório: Não
`Content`	O conteúdo da mensagem retornada da função Lambda. Pode ser editado ou original, conforme a lógica de negócios. • Restrições de Tamanho: tamanho mínimo 1. Tamanho máximo do `MaximumMessageLength` que você definiu ao criar ou atualizar a sala. Consulte a Referência da API do Amazon IVS Chat para obter mais informações. Isso se aplica somente quando `ReviewResult` é `ALLOW`. • Obrigatório: sim
`ReviewResult`	O resultado do processamento de revisão sobre como manipular a mensagem. Se permitido, a mensagem será entregue a todos os usuários conectados à sala. Se negado, a mensagem não será entregue a nenhum usuário. • Valores válidos: `ALLOW` | `DENY` • Obrigatório: sim

Código de exemplo

Veja abaixo um exemplo de manipulador do Lambda em Go. Modifica o conteúdo da mensagem, mantém os atributos da mensagem inalterados e permite a mensagem.

```
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
}
```

Associar e desassociar um manipulador de uma sala

Depois de configurar e implementar o manipulador do Lambda, use a API do Amazon IVS Chat:

- Para associar o manipulador a uma sala, chame CreateRoom ou UpdateRoom e especifique o manipulador.
- Para desassociar o manipulador de uma sala, chame UpdateRoom com um valor vazio para `MessageReviewHandler.Uri`.

Monitoramento de erros com o Amazon CloudWatch

É possível monitorar erros que ocorrem na revisão de mensagens com o Amazon CloudWatch e criar alarmes ou painéis para indicar ou responder às alterações de erros específicos. Se ocorrer um erro, a mensagem será permitida ou negada conforme o valor de `FallbackResult` especificado quando você associa o manipulador a uma sala; consulte MessageReviewHandler na Referência da API do Amazon IVS Chat.

Há vários tipos de erros:

- `InvocationErrors` ocorrem quando o Amazon IVS Chat não pode chamar um manipulador.
- `ResponseValidationErrors` ocorrem quando um manipulador retorna uma resposta inválida.
- Os `Errors` do AWS Lambda ocorrem quando um manipulador do Lambda retorna um erro de função quando foi invocado.

Para obter mais informações sobre erros de invocação e erros de validação de resposta (emitidos pelo Chat do Amazon IVS), Monitoramento do Chat do Amazon IVS. Para obter mais informações sobre erros do AWS Lambda, consulte Trabalhar com métricas de funções do Lambda.

Monitoramento do Chat do Amazon IVS

É possível monitorar os recursos de Chat do Amazon Interactive Video Service (IVS) usando o Amazon CloudWatch. O CloudWatch coleta e processa dados brutos do Chat do Amazon IVS e os transforma em métricas legíveis quase em tempo real. Essas estatísticas são mantidas por 15 meses, de maneira que você pode obter uma perspectiva histórica de como a aplicação da Web ou o serviço está se saindo. É possível definir alarmes para determinados limites e enviar notificações ou realizar ações quando esses limites são atingidos. Para obter mais detalhes, consulte o Guia do usuário do CloudWatch.

Acessar métricas do CloudWatch

O Amazon CloudWatch coleta e processa dados brutos do Chat do Amazon IVS e os transforma em métricas legíveis quase em tempo real. Essas estatísticas são mantidas por 15 meses, de maneira que você pode obter uma perspectiva histórica de como a aplicação da Web ou o serviço está se saindo. É possível definir alarmes para determinados limites e enviar notificações ou realizar ações quando esses limites são atingidos. Para obter mais detalhes, consulte o Guia do usuário do CloudWatch.

Observe que as métricas do CloudWatch são acumuladas ao longo do tempo. A resolução diminui efetivamente à medida que as métricas envelhecem. A programação é:

- Métricas de 60 segundos permanecem disponíveis por 15 dias.
- Métricas de 5 minutos permanecem disponíveis por 63 dias.
- Métricas de 1 hora permanecem disponíveis por 455 dias (15 meses).

Para obter informações atuais sobre retenção de dados, procure por “período de retenção” nas Perguntas frequentes sobre o Amazon CloudWatch.

Instruções do console do CloudWatch

1. Abra o console do CloudWatch em https://console.aws.amazon.com/cloudwatch/.
2. Na navegação lateral, expanda a lista suspensa Metrics (Métricas) e, em seguida, selecione All metrics (Todas as métricas).
3. Na guia Procurar, usando o menu suspenso sem rótulo à esquerda, selecione a sua região “inicial”, onde os seus canais foram criados. Para obter mais informações sobre regiões, consulte

Solução global, controle regional. Para obter uma lista das regiões compatíveis, consulte a página do Amazon IVS na Referência geral da AWS.

4. Na parte inferior da guia Procurar, selecione o namespace IVSChat.
5. Execute um destes procedimentos:
   a. Na barra de pesquisa, insira o ID do recurso (parte do ARN, `arn:::ivschat:room/<resource id>`).

      Em seguida, selecione IVSChat.
   b. Se IVSChat aparecer como um serviço selecionável em Namespaces da AWS, selecione-o. Ele será listado se você usar o Chat do Amazon IVS e estiver enviando métricas para o Amazon CloudWatch. (Se IVSChat não estiver listado, você não terá nenhuma métrica do Chat do Amazon IVS).

      Em seguida, escolha um agrupamento de dimensões, conforme desejado. As dimensões disponíveis estão listadas em Métricas do CloudWatch abaixo.
6. Escolha as métricas a serem adicionadas ao gráfico. As métricas disponíveis estão listadas em Métricas do CloudWatch abaixo.

Você também pode acessar o gráfico CloudWatch da sessão de chat na página de detalhes da sessão de chat selecionando o botão Visualizar no CloudWatch.

Instruções da CLI

Você também pode acessar as métricas usando a AWS CLI. Isso exige que você primeiro faça o download e configure a CLI em sua máquina. Para obter mais detalhes, consulte o Guia do usuário da Interface de Linhas de Comando da AWS.

Depois, para acessar as métricas do chat de baixa latência do Amazon IVS usando a CLI da AWS:

- Em um prompt de comando, execute:

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

Para obter mais informações, consulte Como usar métricas do Amazon CloudWatch no Guia do usuário do Amazon CloudWatch.

Métricas do CloudWatch: Chat do IVS

O Amazon IVS Chat fornece as seguintes métricas no namespace da AWS/IVSChat.

Métrica	Dimensão	Descrição
`ConcurrentChatConnections`	Nenhum	O total de conexões simultâneas em uma sala de chat (máximo por minuto informado). Isso é útil para entender quando os clientes se aproximam do limite para conexões de chat simultâneas em uma região. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo
`Deliveries`	Ação	O número de entregas de solicitações de mensagens feitas de um tipo específico de ação para conexões por chat em todas as salas de uma região. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo
`InvocationErrors`	`Uri`	O número de erros de invocação de um manipulador de revisão de mensagens específico em todas as suas salas em uma região. Um erro de invocação ocorre quando não é possível invocar o manipulador de revisão de mensagens. Os erros de invocação ocorrem quando o Amazon IVS Chat não pode chamar um manipulador. Isso pode acontecer se o manipulador associado a uma sala não existir

Métrica	Dimensão	Descrição
		mais ou expirar ou se a política de recursos não permitir que o serviço o invoque. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo
`LogDestinationAccessDeniedError`	`LoggingConfiguration`	O número de erros de acesso negado de um destino de log em todas as suas salas em uma região. Esses erros ocorrem quando o Amazon IVS Chat não pode acessar o recurso de destino que você especificou na configuração de log. Isso poderá acontecer se a política de recursos de destino não permitir que o serviço adicione registros. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo
`LogDestinationErrors`	`LoggingConfiguration`	O número de todos os erros de um destino de log em todas as suas salas em uma região. Essa é uma métrica agregada que inclui todos os tipos de erros que ocorrem quando o Amazon IVS Chat não entrega logs para o recurso de destino que você especificou na configuração de log. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo

Métrica	Dimensão	Descrição
`LogDestinationResourceNotFoundErrors`	`LoggingConfiguration`	O número de erros de recurso não encontrado de um destino de log em todas as suas salas em uma região. Esses erros ocorrem quando o Amazon IVS Chat não pode entregar os logs a um recurso de destino que você especificou em uma configuração de log porque o recurso não existe. Isso poderá acontecer se o recurso de destino associado a uma configuração de log não existir mais. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo
`MessagingDeliveries`	Nenhum	O número de entregas de solicitações de mensagens para conexões de chat em todas as suas salas em uma região. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo
`MessagingRequests`	Nenhum	O número de solicitações de mensagens feitas em todas as suas salas em uma região. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo

Métrica	Dimensão	Descrição
`Requests`	Ação	O número de solicitações feitas de um tipo de ação específico em todas as suas salas em uma região. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo
`ResponseValidationErrors`	`Uri`	O número de erros de resposta-validação de um manipulador de revisão de mensagens específico em todas as suas salas em uma região. Um erro de validação de resposta ocorre quando a resposta do manipulador de revisão de mensagens é inválida. Isso pode significar que a resposta não pôde ser analisada ou não passou nas verificações de validação; por exemplo, um resultado de revisão inválido ou valores de resposta que são muito longos. Unidade: contagem Estatísticas válidas: soma, média, máximo, mínimo

SDK de Mensagens para Clientes do Chat do IVS

O Amazon Interactive Video Services (IVS) Chat Client Messaging SDK destina-se a desenvolvedores que estão criando aplicações com o Amazon IVS. Este SDK foi desenvolvido para aproveitar a arquitetura do Amazon IVS e receberá atualizações, juntamente com o Amazon IVS Chat. Na condição um SDK nativo, ele foi projetado para minimizar o impacto na performance em sua aplicação e nos dispositivos com os quais seus usuários acessam sua aplicação.

Requisitos da plataforma

Navegadores desktop

Navegador	Versões compatíveis
Chrome	Duas versões principais (versão anterior atual e mais recente)
Borda	Duas versões principais (versão anterior atual e mais recente)
Firefox	Duas versões principais (versão anterior atual e mais recente)
Opera	Duas versões principais (versão anterior atual e mais recente)
Safari	Duas versões principais (versão anterior atual e mais recente)

Navegadores móveis

Navegador	Versões compatíveis
Chrome para Android	Duas versões principais (versão anterior atual e mais recente)
Firefox para Android	Duas versões principais (versão anterior atual e mais recente)
Opera para Android	Duas versões principais (versão anterior atual e mais recente)

Navegador	Versões compatíveis
WebView Android	Duas versões principais (versão anterior atual e mais recente)
Internet da Samsung	Duas versões principais (versão anterior atual e mais recente)
Safari para iOS	Duas versões principais (versão anterior atual e mais recente)

Plataformas nativas

Plataforma	Versões compatíveis
Android	5.0 e posterior
iOS	13.0 e posterior

Suporte

Em caso de erro ou outro problema com a sala de chat, determine o identificador exclusivo da sala via API do IVS Chat (consulte ListRooms).

Compartilhe esse identificador de sala de transmissão com o AWS Support. Com ele, a equipe de suporte poderá obter informações para ajudar a solucionar seu problema.

Observação: consulte as Notas de versão do Chat do Amazon IVS para obter informações sobre versões disponíveis e problemas corrigidos. Se apropriado, antes de entrar em contato com o suporte, atualize sua versão do SDK de transmissão e veja se isso resolve seu problema.

Versionamento

Os SDKs do Amazon IVS Chat Client Messaging usam versionamento semântico.

Para esta discussão, suponha que:

- A versão mais recente é 4.1.3.
- A versão mais recente da versão principal anterior é 3.2.4.

- A versão mais recente da versão 1.x é 1.5.6.

Novos recursos compatíveis com versões anteriores são adicionados como versões secundárias da versão mais recente. Nesse caso, o próximo conjunto de novos recursos vai ser adicionado como versão 4.2.0.

Compatíveis com versões anteriores, pequenas correções de bugs são adicionadas como lançamentos de patch da versão mais recente. Aqui, o próximo conjunto de pequenas correções de bugs vai ser adicionado como versão 4.1.4.

Compatíveis com versões anteriores, as principais correções de bugs são tratadas de forma diferente; estas são adicionadas a várias versões:

- Versão do patch da versão mais recente. Aqui, esta é a versão 4.1.4.
- Lançamento do patch da versão secundária anterior. Aqui, esta é a versão 3.2.5.
- Versão do patch da versão 1.x mais recente. Aqui, esta é a versão 1.5.7.

As principais correções de bugs são definidas pela equipe de produtos do Amazon IVS. Exemplos típicos são atualizações de segurança críticas e outras correções selecionadas necessárias para os clientes.

Observação: nos exemplos acima, versões lançadas incrementam sem ignorar nenhum número (por exemplo, de 4.1.3 para 4.1.4). Na realidade, um ou mais números de patch podem permanecer internos e não ser liberados, de modo que a versão lançada pode ser incrementada de 4.1.3 para, digamos, 4.1.6.

Além disso, o suporte à versão 1.x será oferecido até o final de 2023 ou quando a versão 3.x for lançada, o que ocorrer por último.

APIs do Amazon IVS Chat

No lado do servidor (não gerenciado pelos SDKs), há duas APIs, cada uma com suas próprias responsabilidades:

- Plano de dados: a API de mensagens do IVS Chat é uma API de WebSocket projetada para ser usada por aplicações front-end (iOS, Android, macOS, etc.) que são acionadas por um esquema de autenticação baseado em tokens. Usando um token de chat gerado anteriormente, é possível se conectar a salas de chat já existentes usando essa API.

Os Amazon IVS Chat Client Messaging SDKs atuam somente no plano de dados. Os SDKs presumem que você já está gerando tokens de chat por meio do seu backend. Supõe-se que a recuperação desses tokens seja gerenciada pela aplicação front-end, e não pelos SDKs.

- Ambiente de gerenciamento: a API de ambiente de gerenciamento do IVS Chat fornece uma interface para suas próprias aplicações de backend para gerenciar e criar salas de chat, bem como os usuários que se juntam a elas. Pense nisso como o painel de administração da experiência de chat da sua aplicação, gerenciado pelo seu próprio backend. Existem operações do ambiente de gerenciamento que são responsáveis por criar o token de chat de que o plano de dados precisa para ser autenticado em uma sala de chat.

  Importante: os IVS Chat Client Messaging SDKs não chamam nenhuma operações do plano de ambiente de gerenciamento. O backend deve ser configurado para criar tokens de chat para você. A aplicação front-end deve se comunicar com o backend para recuperar esse token de chat.

SDK de Mensagens para Clientes do Chat do IVS: Guia para Android

O Amazon Interactive Video (IVS) Chat Client Messaging Android SDK fornece interfaces que permitem a você incorporar facilmente nossa API do IVS Chat Messaging em plataformas que utilizam Android.

O pacote `com.amazonaws:ivs-chat-messaging` implementa a interface descrita neste documento.

Versão mais recente do SDK de Mensagens para Clientes do Chat do IVS para Android: 1.1.0 (Notas de lançamento)

Documentação de referência: para obter informações sobre os métodos mais importantes disponíveis no Amazon IVS Chat Client Messaging Android SDK, consulte a documentação de referência em https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/.

Exemplo de código: consulte o repositório de exemplos para Android no GitHub: https://github.com/aws-samples/amazon-ivs-chat-for-android-demo

Requisitos de plataforma: o Android 5.0 (API nível 21) ou superior é necessário para desenvolvimento.

Introdução ao SDK para Android de mensagens para clientes do Chat do IVS

Antes de começar, você deve se familiarizar com os Conceitos básicos do Amazon IVS Chat.

Adicionar o pacote

Adicione `com.amazonaws:ivs-chat-messaging` às dependências de `build.gradle`:

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

Adicionar regras do Proguard

Adicione as seguintes entradas ao arquivo de regras do R8/Proguard (`proguard-rules.pro`):

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

Configurar seu backend

Esta integração exige endpoints em seu servidor que conversem com a API do Amazon IVS. Use as bibliotecas oficiais da AWS para acessar a API do Amazon IVS via servidor. Elas podem ser acessadas em várias linguagens via pacotes públicos, por exemplo, node.js e Java.

Em seguida, crie um endpoint de servidor que se comunique com a API do Amazon IVS Chat e crie um token.

Configurar uma conexão com o servidor

Crie um método que use `ChatTokenCallback` como um parâmetro e busque um token de chat do seu backend. Passe esse token para o método `onSuccess` do retorno de chamada. Em caso de erro, passe a exceção para o método `onError` do retorno de chamada. Isso é necessário para instanciar a entidade na `ChatRoom` principal na próxima etapa.

Um exemplo de código que implementa o código acima usando uma chamada `Retrofit` é mostrado a seguir.

```
// ...
```

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

Usar o SDK para Android de mensagens para clientes do Chat do IVS

Este documento descreve as etapas envolvidas no uso do SDK para Android de mensagens para clientes do Chat do Amazon IVS

Inicializar uma instância de sala de chat

Crie uma instância da classe `ChatRoom`. Isso requer passar `regionOrUrl`, que normalmente é a região da AWS na qual sua sala de chat está hospedada, e `tokenProvider`, que é o método de busca de tokens criado na etapa anterior.

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

Em seguida, crie um objeto ouvinte que implementará manipuladores para eventos relacionados ao chat e o atribuirá à propriedade `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

// ...
```

A última etapa da inicialização básica é conectar à sala específica estabelecendo uma conexão WebSocket. Para fazer isso, chame o método `connect()` dentro da instância da sala. Recomendamos fazer isso no método de ciclo de vida `onResume()` para garantir que ele mantenha uma conexão se a aplicação for retomada em segundo plano.

```
room.connect()
```

O SDK tentará estabelecer uma conexão com uma sala de chat codificada no token de chat recebido do servidor. Se falhar, ele tentará reconectar o número de vezes especificado na instância da sala.

Executar ações em uma sala de chat

A classe `ChatRoom` tem ações para enviar e excluir mensagens e desconectar outros usuários. Essas ações aceitam um parâmetro opcional de retorno de chamada que permite que você receba notificações de confirmação ou rejeição da solicitação.

Enviar uma mensagem

Para esta solicitação, é necessário ter a capacidade `SEND_MESSAGE` codificada em seu token de chat.

Para acionar uma solicitação de envio de mensagem:

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

Para obter uma confirmação/rejeição da solicitação, forneça um retorno de chamada como segundo parâmetro:

```
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.
   }
})
```

Excluir mensagem

Para esta solicitação, é necessário ter a capacidade DELETE_MESSAGE codificada em seu token de chat.

Para acionar uma solicitação delete-message:

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

Para obter uma confirmação/rejeição da solicitação, forneça um retorno de chamada como segundo parâmetro:

```
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.
   }
})
```

Desconectar outro usuário

Para esta solicitação, é necessário ter a capacidade `DISCONNECT_USER` codificada em seu token de chat.

Para desconectar outro usuário para fins de moderação:

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

Para obter a confirmação/rejeição da solicitação, forneça um retorno de chamada como segundo parâmetro:

```
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.
   }
})
```

Desconectar de uma sala de chat

Para fechar sua conexão com a sala de chat, chame o método `disconnect()` na instância da sala:

```
room.disconnect()
```

Como a conexão WebSocket deixará de funcionar após um curto período quando a aplicação estiver em segundo plano, recomendamos se conectar/desconectar manualmente ao fazer a transição de/para um estado de segundo plano. Para fazer isso, combine a chamada `room.connect()` no método de ciclo de vida `onResume()`, em `Activity` ou `Fragment` do Android, com uma chamada `room.disconnect()` no método de ciclo de vida `onPause()`.

SDK de Mensagens para Clientes do Chat do IVS: Tutorial para Android, parte 1: salas de chat

Esta é a primeira de um tutorial de duas partes. Você aprenderá os fundamentos do trabalho com o SDK de Mensagens do Chat do Amazon IVS ao desenvolver uma aplicação Android totalmente funcional usando a linguagem de programação Kotlin. Chamamos a aplicação de Chatterbox.

Antes de iniciar o módulo, dedique alguns minutos para se familiarizar com os pré-requisitos, os principais conceitos por trás dos tokens de chat e o servidor de backend necessários para criar salas de chat.

Esses tutoriais são criados para desenvolvedores de Android experientes que são iniciantes no SDK de Mensagens para Clientes do Chat do IVS. Você precisará estar familiarizado com a linguagem de programação Kotlin e com a criação de interfaces de usuário na plataforma Android.

Esta primeira parte do tutorial está dividida em várias seções:

1. the section called “Configure um servidor local de autenticação/autorização”
2. the section called “Crie um projeto de Chatterbox”
3. the section called “Conecte-se a uma sala de chat e observe as atualizações de conexão”
4. the section called “Crie um provedor de tokens”
5. the section called “Próximas etapas”

Para obter a documentação completa do SDK, comece com o SDK de Mensagens para Clientes do Chat do Amazon IVS (aqui no Guia de usuário do Chat do Amazon IVS) e a Referência de Mensagens para Clientes do Chat: SDK para Android (no Github).

Pré-requisitos

- Ter familiaridade com Kotlin e com a criação de aplicações na plataforma Android. Se você não tiver familiaridade com a criação de aplicações para Android, aprenda o básico no guia Crie sua primeira aplicação para desenvolvedores de Android.
- Ler e compreender completamente os Conceitos básicos do Chat do IVS.
- Crie um usuário do AWS IAM com os recursos `CreateChatToken` e `CreateRoom` definidos em uma política do IAM existente. (Consulte Conceitos básicos do Chat do IVS.)
- Certifique-se de que as chaves secretas/de acesso desse usuário estejam armazenadas em um arquivo de credenciais da AWS. Para obter instruções, consulte o Guia do usuário da AWS CLI (especialmente Configuração e definições do arquivo de credenciais).
- Crie uma sala de chat e salve seu ARN. Consulte Conceitos básicos do Chat do IVS. (Se você não salvar o ARN, poderá consultá-lo posteriormente com o console ou a API do Chat.)

Configure um servidor local de autenticação/autorização

Seu servidor de backend será responsável por criar salas de chat e gerar os tokens de chat necessários para que o SDK do Chat do IVS para Android autentique e autorize seus clientes nas salas de chat.

Consulte Criar um token de chat em Introdução ao Amazon IVS Chat. Conforme mostrado no fluxograma, o código do lado do servidor é responsável por criar um token de chat. Isso significa que sua aplicação deve fornecer seu próprio meio de gerar um token de chat solicitando-o da sua aplicação a partir do lado do servidor.

Usamos a estrutura Ktor para criar um servidor local ativo que gerencia a criação de tokens de chat usando seu ambiente local da AWS.

Neste momento, esperamos que você tenha as credenciais da AWS configuradas corretamente. Para obter instruções detalhadas, consulte Configurar as credenciais e a região da AWS para o desenvolvimento.

Crie um novo diretório e chame-o de `chatterbox`. Nele, crie outro, chamado `auth-server`.

A pasta do nosso servidor terá a seguinte estrutura:

```
- auth-server
```

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

Observação: é possível copiar e colar este código diretamente nos arquivos referenciados.

Em seguida, adicionaremos todas as dependências e plug-ins necessários para que o servidor de autenticação funcione:

Script de 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")
}
```

Agora, é necessário configurar a funcionalidade de registro em log para o servidor de autenticação. (Para obter mais informações, consulte Configurar 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>
```

O servidor Ktor requer definições de configuração, que são carregadas automaticamente do arquivo `application.*` no diretório `resources`, então adicionaremos isso também. (Para obter mais informações, consulte Configuração em um arquivo.)

HOCON:

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

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

Por fim, vamos implementar o servidor:

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()
```

```
            )
        )
    }
  }
}
```

Crie um projeto de Chatterbox

Para criar um projeto Android, instale e abra o Android Studio.

Siga as etapas listadas no guia oficial Criar um projeto do Android.

- Em Escolher o tipo de projeto, selecione o modelo de projeto Atividade em branco para a aplicação Chatterbox.
- Em Configurar o projeto, escolha os valores a seguir para os campos de configuração:
  - Nome: My App
  - Nome do pacote: com.chatterbox.myapp
  - Salvar localização: direcione para o diretório `chatterbox` criado na etapa anterior
  - Linguagem: Kotlin
  - Nível mínimo de API: API 21: Android 5.0 (Lollipop)

Após especificar todos os parâmetros de configuração corretamente, a estrutura de arquivos na pasta `chatterbox` deve se assemelhar a:

```
- 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
```

Agora que temos um projeto Android em funcionamento, é possível adicionar com.amazonaws:ivs-chat-messaging às dependências `build.gradle`. (Para obter mais informações sobre o kit de ferramentas de compilação Gradle, consulte Configurar sua compilação.)

Observação: na parte superior de cada trecho de código, há um caminho para o arquivo em que você deve fazer alterações em seu projeto. O caminho é relativo à raiz do projeto.

No código abaixo, substitua *<version>* pelo número da versão atual do SDK do Chat para Android (por exemplo, 1.0.0).

Kotlin:

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

plugins {
// ...
}

android {
// ...
}

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

Após a nova dependência ser adicionada, execute Sincronizar projeto com arquivos do Gradle no Android Studio para sincronizar o projeto com a nova dependência. (Para obter mais informações, consulte Adicionar dependências de compilação.)

Para executar o servidor de autenticação (criado na seção anterior) de forma conveniente a partir da raiz do projeto, nós o incluímos como um novo módulo em `settings.gradle`. (Para obter mais informações, consulte Como estruturar e criar um componente de software com o Gradle.)

Script de Kotlin:

```
// ./settings.gradle

// ...

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

A partir de agora, como `auth-server` está incluso no projeto Android, é possível executar o servidor de autenticação com o seguinte comando da raiz do projeto:

Shell:

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

Conecte-se a uma sala de chat e observe as atualizações de conexão

Para abrir uma conexão de sala de chat, usamos o retorno de chamada do ciclo de vida da atividade onCreate(), que é acionado quando a atividade é criada pela primeira vez. O construtor do ChatRoom exige que forneçamos `region` e `tokenProvider` para instanciar uma conexão de sala.

Observação: a função `fetchChatToken` apresentada no trecho abaixo será implementada na próxima seção.

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)
   }

// ...
}
```

Exibir e reagir a mudanças na conexão de uma sala de chat são partes essenciais da criação de uma aplicação de chat como o `chatterbox`. Antes de começar a interagir com a sala, é necessário se inscrever em eventos de estado de conexão da sala de chat para obter atualizações.

O ChatRoom espera que vinculemos uma implementação da interface ChatRoomListener para a geração de eventos de ciclo de vida. Por enquanto, as funções do receptor registrarão em log somente mensagens de confirmação, quando invocadas:

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")
        }
    }
}
```

Agora que implementamos o `ChatRoomListener`, vamos vinculá-lo à instância da sala:

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 {
// ...
}
```

Depois disso, é necessário fornecer a capacidade de ler o estado de conexão da sala. Vamos mantê-lo na propriedade `MainActivity.kt` e inicializá-lo com o estado padrão DISCONNECTED para salas (consulte `ChatRoom state` na Referência do SDK do Chat do IVS para Android). Para ser possível manter o estado local atualizado, é necessário implementar uma função de atualização de estado. Vamos chamá-la de `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")
         }
      }
   }
```

```
}
```

Em seguida, integraremos a função de atualização de estado com a propriedade 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)
         }
      }
   }
}
```

Agora que temos a capacidade de salvar, ouvir e reagir às atualizações de estado do ChatRoom, é o momento de inicializar uma conexão:

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

Crie um provedor de tokens

É hora de criar uma função responsável pela criação e pelo gerenciamento de tokens de chat na aplicação. Neste exemplo, usamos o cliente HTTP Retrofit para Android.

Antes de ser possível enviar qualquer tráfego de rede, é necessário definir uma configuração de segurança de rede para o Android. (Para obter mais informações, consulte Configuração de segurança de rede.) Começamos adicionando permissões de rede ao arquivo Manifesto da aplicação. Observe a etiqueta `user-permission` e o atributo `networkSecurityConfig` adicionados, que direcionarão para a nova configuração de segurança de rede. No código abaixo, substitua *<version>* pelo número da versão atual do SDK do Chat para Android (por exemplo, 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")
}
```

Declare os domínios `10.0.2.2` e `localhost` como confiáveis para começar a trocar mensagens com o backend:

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>
```

Em seguida, é necessário adicionar uma nova dependência, em conjunto com a adição do conversor Gson para analisar as respostas HTTP. No código abaixo, substitua *<version>* pelo número da versão atual do SDK do Chat para Android (por exemplo, 1.0.0).

Script de Kotlin:

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

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

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

Para recuperar um token de chat, precisamos realizar uma solicitação POST HTTP da aplicação `chatterbox`. Definimos a solicitação em uma interface para implementação do Retrofit. Consulte a documentação do Retrofit. Familiarize-se também com a especificação da operação 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>
}
```

Agora, com a rede configurada, é o momento de adicionar uma função responsável pela criação e pelo gerenciamento do token de chat. Nós o adicionamos ao `MainActivity.kt`, que foi criado automaticamente quando o projeto foi gerado:

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)
         }
      })
   }
}
```

Próximas etapas

Agora que você estabeleceu uma conexão com a sala de chat, prossiga para a parte 2 deste tutorial para Android: mensagens e eventos.

SDK de Mensagens para Clientes do Chat do IVS: Tutorial para Android, parte 2: mensagens e eventos

Esta segunda e última parte do tutorial é dividida em várias seções:

1. the section called “Crie uma IU para enviar mensagens”
   a. the section called “Layout principal da IU”
   b. the section called “Célula de texto abstrato da IU para exibição do texto de forma consistente”
   c. the section called “Mensagem à esquerda do chat da IU”
   d. the section called “Mensagem à direita do chat da IU”
   e. the section called “Valores de cores adicionais da IU”
2. the section called “Aplicação de vinculação de visualizações”
3. the section called “Gerenciamento de solicitações de mensagens de chat”
4. the section called “Etapas finais”

Para obter a documentação completa do SDK, comece com o SDK de Mensagens para Clientes do Chat do Amazon IVS (aqui no Guia de usuário do Chat do Amazon IVS) e a Referência de Mensagens para Clientes do Chat: SDK para Android (no Github).

Pré-requisito

Certifique-se de ter concluído a parte 1 deste tutorial, salas de chat.

Crie uma IU para enviar mensagens

Agora que inicializamos com sucesso a conexão da sala de chat, é hora de enviar nossa primeira mensagem. Para esse recurso, uma IU é necessária. Nós adicionaremos:

- Botão `connect/disconnect`
- Entrada de mensagem com o botão `send`
- Lista de mensagens dinâmicas. Para desenvolver isso, usamos o RecyclerView do Android Jetpack.

Layout principal da IU

Consulte os layouts do Android Jetpack na documentação do desenvolvedor do 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>
```

Célula de texto abstrato da IU para exibição do texto de forma consistente

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>
```

Mensagem à esquerda do chat da IU

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>
```

Mensagem à direita do chat da IU

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>
```

Valores de cores adicionais da IU

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>
```

Aplicação de vinculação de visualizações

Aproveitamos o recurso Vinculação de visualizações do Android para poder referenciar as classes de vinculação para o layout XML. Para ativar o recurso, defina a opção de desenvolvimento `viewBinding` como `true` em `./app/build.gradle`:

Script de Kotlin:

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

android {
//    ...

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

Agora é o momento de conectar a IU com o código 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)
    }
}
```

Também adicionamos métodos para excluir mensagens e desconectar usuários do chat, que podem ser invocados usando o menu de contexto da mensagem de 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}")
                    }
                }
            }
        )
    }
}
```

Gerenciamento de solicitações de mensagens de chat

Precisamos de uma maneira de gerenciar as solicitações de mensagens de chat em todos os estados possíveis:

- Pendente: uma mensagem foi enviada para uma sala de chat, mas ainda não foi confirmada ou rejeitada.
- Confirmada: uma mensagem foi enviada pela sala de chat para todos os usuários (inclusive nós).
- Rejeitada: uma mensagem foi rejeitada pela sala de chat com um objeto de erro.

Manteremos solicitações de chat e mensagens de chat não resolvidas em uma lista. A lista merece uma classe separada, que denominaremos `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()
    }
}
```

Para conectar a lista com a IU, usaremos um Adaptador. Para obter mais informações, consulte Vinculação de dados com o AdapterView e Classes de vinculação geradas.

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
}
```

Etapas finais

É o momento de conectar o novo adaptador, vinculando a classe `ChatEntries` à `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
        }
    }
}
```

Como já temos uma classe responsável por rastrear as solicitações de chat (`ChatEntries`), está tudo pronto para implementar o código para manipulação de `entries` na `roomListener`. Atualizaremos `entries` e `connectionState` de acordo com os eventos que estamos respondendo:

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")
        }
    }
}
```

Agora você deveria poder executar a aplicação. (Consulte Construir e executar sua aplicação.) Lembre-se de ter o servidor de backend em execução ao usar a aplicação. É possível ativá-lo no terminal na raiz do projeto com o comando: `./gradlew :auth-server:run` ou ao executar a tarefa `auth-server:run` do Gradle diretamente do Android Studio.

SDK de Mensagens para Clientes do Chat do IVS: Tutorial de Kotlin Coroutines, parte 1: salas de chat

Esta é a primeira de um tutorial de duas partes. Você aprenderá os fundamentos do trabalho com o SDK de Mensagens do Chat do Amazon IVS ao desenvolver uma aplicação Android totalmente funcional usando a linguagem de programação e as coroutines do Kotlin. Chamamos a aplicação de Chatterbox.

Antes de iniciar o módulo, dedique alguns minutos para se familiarizar com os pré-requisitos, os principais conceitos por trás dos tokens de chat e o servidor de backend necessários para criar salas de chat.

Esses tutoriais são criados para desenvolvedores de Android experientes que são iniciantes no SDK de Mensagens para Clientes do Chat do IVS. Você precisará estar familiarizado com a linguagem de programação Kotlin e com a criação de interfaces de usuário na plataforma Android.

Esta primeira parte do tutorial está dividida em várias seções:

1. the section called “Configure um servidor local de autenticação/autorização”
2. the section called “Crie um projeto de Chatterbox”
3. the section called “Conecte-se a uma sala de chat e observe as atualizações de conexão”
4. the section called “Crie um provedor de tokens”
5. the section called “Próximas etapas”

Para obter a documentação completa do SDK, comece com o SDK de Mensagens para Clientes do Chat do Amazon IVS (aqui no Guia de usuário do Chat do Amazon IVS) e a Referência de Mensagens para Clientes do Chat: SDK para Android (no Github).

Pré-requisitos

- Ter familiaridade com Kotlin e com a criação de aplicações na plataforma Android. Se você não tiver familiaridade com a criação de aplicações para Android, aprenda o básico no guia Crie sua primeira aplicação para desenvolvedores de Android.
- Leia e compreenda Conceitos básicos do Chat do IVS.
- Crie um usuário do AWS IAM com os recursos `CreateChatToken` e `CreateRoom` definidos em uma política do IAM existente. (Consulte Conceitos básicos do Chat do IVS.)
- Certifique-se de que as chaves secretas/de acesso desse usuário estejam armazenadas em um arquivo de credenciais da AWS. Para obter instruções, consulte o Guia do usuário da AWS CLI (especialmente Configuração e definições do arquivo de credenciais).
- Crie uma sala de chat e salve seu ARN. Consulte Conceitos básicos do Chat do IVS. (Se você não salvar o ARN, poderá consultá-lo posteriormente com o console ou a API do Chat.)

Configure um servidor local de autenticação/autorização

Seu servidor de backend será responsável por criar salas de chat e gerar os tokens de chat necessários para que o SDK do Chat do IVS para Android autentique e autorize seus clientes nas salas de chat.

Consulte Criar um token de chat em Introdução ao Amazon IVS Chat. Conforme mostrado no fluxograma, o código do lado do servidor é responsável por criar um token de chat. Isso significa que sua aplicação deve fornecer seu próprio meio de gerar um token de chat solicitando-o da sua aplicação a partir do lado do servidor.

Usamos a estrutura Ktor para criar um servidor local ativo que gerencia a criação de tokens de chat usando seu ambiente local da AWS.

Neste momento, esperamos que você tenha as credenciais da AWS configuradas corretamente. Para obter instruções detalhadas, consulte Configurar credenciais temporárias da AWS e a região da AWS para desenvolvimento.

Crie um novo diretório e chame-o de `chatterbox`. Nele, crie outro, chamado `auth-server`.

A pasta do nosso servidor terá a seguinte estrutura:

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

Observação: é possível copiar e colar este código diretamente nos arquivos referenciados.

Em seguida, adicionaremos todas as dependências e plug-ins necessários para que o servidor de autenticação funcione:

Script de 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")
}
```

Agora, é necessário configurar a funcionalidade de registro em log para o servidor de autenticação. (Para obter mais informações, consulte Configurar 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>
```

O servidor Ktor requer definições de configuração, que são carregadas automaticamente do arquivo `application.*` no diretório `resources`, então adicionaremos isso também. (Para obter mais informações, consulte Configuração em um arquivo.)

HOCON:

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

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

Por fim, vamos implementar o servidor:

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()
            )
        )
    }
  }
}
```

Crie um projeto de Chatterbox

Para criar um projeto Android, instale e abra o Android Studio.

Siga as etapas listadas no guia oficial Criar um projeto do Android.

- Em Escolher o projeto, selecione o modelo de projeto Atividade em branco para a aplicação Chatterbox.

- Em Configurar o projeto, escolha os valores a seguir para os campos de configuração:
  - Nome: My App
  - Nome do pacote: com.chatterbox.myapp
  - Salvar localização: direcione para o diretório `chatterbox` criado na etapa anterior
  - Linguagem: Kotlin
  - Nível mínimo de API: API 21: Android 5.0 (Lollipop)

Após especificar todos os parâmetros de configuração corretamente, a estrutura de arquivos na pasta `chatterbox` deve se assemelhar a:

```
- 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
```

Agora que temos um projeto Android em funcionamento, é possível adicionar com.amazonaws:ivs-chat-messaging e org.jetbrains.kotlinx:kotlinx-coroutines-core às dependências `build.gradle`. (Para obter mais informações sobre o kit de ferramentas de compilação Gradle, consulte Configurar sua compilação.)

Observação: na parte superior de cada trecho de código, há um caminho para o arquivo em que você deve fazer alterações em seu projeto. O caminho é relativo à raiz do projeto.

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'

// ...
}
```

Após a nova dependência ser adicionada, execute Sincronizar projeto com arquivos do Gradle no Android Studio para sincronizar o projeto com a nova dependência. (Para obter mais informações, consulte Adicionar dependências de compilação.)

Para executar o servidor de autenticação (criado na seção anterior) de forma conveniente a partir da raiz do projeto, nós o incluímos como um novo módulo em `settings.gradle`. (Para obter mais informações, consulte Como estruturar e criar um componente de software com o Gradle.)

Script de Kotlin:

```
// ./settings.gradle

// ...

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

A partir de agora, como `auth-server` está incluso no projeto Android, é possível executar o servidor de autenticação com o seguinte comando da raiz do projeto:

Shell:

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

Conecte-se a uma sala de chat e observe as atualizações de conexão

Para abrir uma conexão de sala de chat, usamos o retorno de chamada do ciclo de vida da atividade onCreate(), que é acionado quando a atividade é criada pela primeira vez. O construtor do ChatRoom exige que forneçamos `region` e `tokenProvider` para instanciar uma conexão de sala.

Observação: a função `fetchChatToken` apresentada no trecho abaixo será implementada na próxima seção.

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)
   }

// ...
}
```

Exibir e reagir a mudanças na conexão de uma sala de chat são partes essenciais da criação de uma aplicação de chat como o `chatterbox`. Antes de começar a interagir com a sala, é necessário se inscrever em eventos de estado de conexão da sala de chat para obter atualizações.

No SDK do Chat para coroutine, `ChatRoom` espera que lidemos com eventos do ciclo de vida da sala no Fluxo. Por enquanto, as funções registrarão em log somente mensagens de confirmação, quando invocadas:

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")
                }
            }
        }
    }
}
```

Depois disso, é necessário fornecer a capacidade de ler o estado de conexão da sala. Vamos mantê-lo na propriedade `MainActivity.kt` e inicializá-lo com o estado padrão DISCONNECTED para salas (consulte `ChatRoom` `state` na Referência do SDK do Chat do IVS para Android). Para ser possível manter o estado local atualizado, é necessário implementar uma função de atualização de estado. Vamos chamá-la de `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")
          }
      }
}
```

Em seguida, integraremos a função de atualização de estado com a propriedade 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)
```

```
                }
            }

        // ...

        }
    }
}
```

Agora que temos a capacidade de salvar, ouvir e reagir às atualizações de estado do ChatRoom, é o momento de inicializar uma conexão:

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)
        }
    }

    // ...
}
```

Crie um provedor de tokens

É hora de criar uma função responsável pela criação e pelo gerenciamento de tokens de chat na aplicação. Neste exemplo, usamos o cliente HTTP Retrofit para Android.

Antes de ser possível enviar qualquer tráfego de rede, é necessário definir uma configuração de segurança de rede para o Android. (Para obter mais informações, consulte Configuração de segurança de rede.) Começamos adicionando permissões de rede ao arquivo Manifesto da

aplicação. Observe a etiqueta `user-permission` e o atributo `networkSecurityConfig` adicionados, que direcionarão para a nova configuração de segurança de rede. No código abaixo, substitua *<version>* pelo número da versão atual do SDK do Chat para Android (por exemplo, 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")
}
```

Declare seu endereço IP local, por exemplo, os domínios `10.0.2.2` e `localhost` como confiáveis, para começar a trocar mensagens com o backend:

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>
```

Em seguida, é necessário adicionar uma nova dependência, em conjunto com a adição do conversor Gson para analisar as respostas HTTP. No código abaixo, substitua *<version>* pelo número da versão atual do SDK do Chat para Android (por exemplo, 1.1.0).

Script de 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")
}
```

Para recuperar um token de chat, precisamos realizar uma solicitação POST HTTP da aplicação `chatterbox`. Definimos a solicitação em uma interface para implementação do Retrofit. Consulte a documentação do Retrofit. Familiarize-se também com a especificação da operação 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)
   }
}
```

Agora, com a rede configurada, é o momento de adicionar uma função responsável pela criação e pelo gerenciamento do token de chat. Nós o adicionamos ao `MainActivity.kt`, que foi criado automaticamente quando o projeto foi gerado:

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)
         }
      })
   }
}
```

Próximas etapas

Agora que você estabeleceu uma conexão com a sala de chat, prossiga para a parte 2 deste tutorial de Kotlin Coroutines: mensagens e eventos.

SDK de Mensagens para Clientes do Chat do IVS: Tutorial de Kotlin Coroutines, parte 2: mensagens e eventos

Esta segunda e última parte do tutorial é dividida em várias seções:

1. the section called “Crie uma IU para enviar mensagens”
   a. the section called “Layout principal da IU”
   b. the section called “Célula de texto abstrato da IU para exibição do texto de forma consistente”
   c. the section called “Mensagem à esquerda do chat da IU”
   d. the section called “Mensagem à direita da IU”
   e. the section called “Valores de cores adicionais da IU”
2. the section called “Aplicação de vinculação de visualizações”
3. the section called “Gerenciamento de solicitações de mensagens de chat”
4. the section called “Etapas finais”

Para obter a documentação completa do SDK, comece com o SDK de Mensagens para Clientes do Chat do Amazon IVS (aqui no Guia de usuário do Chat do Amazon IVS) e a Referência de Mensagens para Clientes do Chat: SDK para Android (no Github).

Pré-requisito

Certifique-se de ter concluído a parte 1 deste tutorial, salas de chat.

Crie uma IU para enviar mensagens

Agora que inicializamos com sucesso a conexão da sala de chat, é hora de enviar nossa primeira mensagem. Para esse recurso, uma IU é necessária. Nós adicionaremos:

- Botão `connect/disconnect`
- Entrada de mensagem com o botão `send`

- Lista de mensagens dinâmicas. Para desenvolver isso, usamos o RecyclerView do Android Jetpack.

Layout principal da IU

Consulte os layouts do Android Jetpack na documentação do desenvolvedor do 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>
```

Célula de texto abstrato da IU para exibição do texto de forma consistente

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>
```

Mensagem à esquerda do chat da IU

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>
```

Mensagem à direita da IU

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>
```

Valores de cores adicionais da IU

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>
```

Aplicação de vinculação de visualizações

Aproveitamos o recurso Vinculação de visualizações do Android para poder referenciar as classes de vinculação para o layout XML. Para ativar o recurso, defina a opção de desenvolvimento `viewBinding` como `true` em `./app/build.gradle`:

Script de Kotlin:

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

android {
//    ...

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

Agora é o momento de conectar a IU com o código 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)
    }
// ...

}
```

Também adicionamos métodos para excluir mensagens e desconectar usuários do chat, que podem ser invocados usando o menu de contexto da mensagem de 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")
           }
        }
    }
}
```

Gerenciamento de solicitações de mensagens de chat

Precisamos de uma maneira de gerenciar as solicitações de mensagens de chat em todos os estados possíveis:

- Pendente: uma mensagem foi enviada para uma sala de chat, mas ainda não foi confirmada ou rejeitada.
- Confirmada: uma mensagem foi enviada pela sala de chat para todos os usuários (inclusive nós).
- Rejeitada: uma mensagem foi rejeitada pela sala de chat com um objeto de erro.

Manteremos solicitações de chat e mensagens de chat não resolvidas em uma lista. A lista merece uma classe separada, que denominaremos `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()
    }
}
```

Para conectar a lista com a IU, usaremos um Adaptador. Para obter mais informações, consulte Vinculação de dados com o AdapterView e Classes de vinculação geradas.

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
}
```

Etapas finais

É o momento de conectar o novo adaptador, vinculando a classe `ChatEntries` à `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
        }
    }
}
```

Como já temos uma classe responsável por acompanhar as solicitações de chat (`ChatEntries`), está tudo pronto para implementar o código para manipulação de `entries` na roomListener. Atualizaremos `entries` e `connectionState` de acordo com os eventos que estamos respondendo:

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)
    }

// ...

}
```

Agora você deveria poder executar a aplicação. (Consulte Construir e executar sua aplicação.) Lembre-se de ter o servidor de backend em execução ao usar a aplicação. É possível ativá-lo no terminal na raiz do projeto com o comando: `./gradlew :auth-server:run` ou ao executar a tarefa `auth-server:run` do Gradle diretamente do Android Studio.

SDK de Mensagens para Clientes do Chat do IVS: Guia para iOS

O Amazon Interactive Video (IVS) Chat Client Messaging iOS SDK fornece interfaces que permitem a você incorporar nossa API do IVS Chat Messaging em plataformas que utilizam a linguagem de programação Swift da Apple.

Versão mais recente do IVS Chat Client Messaging iOS SDK: 1.0.0 (Notas de versão)

Documentação de referência e tutoriais: para obter informações sobre os métodos mais importantes disponíveis no Amazon IVS Chat Client Messaging iOS SDK, consulte a documentação de referência em https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/. Este repositório também contém vários artigos e tutoriais.

Código de amostra: consulte o repositório de amostra do iOS no GitHub: https://github.com/aws-samples/amazon-ivs-chat-for-ios-demo.

Requisitos de plataforma: o iOS 13.0 ou superior é necessário para o desenvolvimento.

Introdução ao SDK para iOS de mensagens para clientes do Chat do IVS

Recomendamos integrar o SDK via Swift Package Manager. Opcionalmente, é possível usar CocoaPods ou integrar o framework manualmente.

Após a integração do SDK, ele poderá ser importado com a adição do seguinte código na parte superior do arquivo Swift relevante:

```
import AmazonIVSChatMessaging
```

Swift Package Manager

Para usar a biblioteca `AmazonIVSChatMessaging` em um projeto do Swift Package Manager, adicione-a às dependências do pacote e às dependências dos alvos relevantes:

1. Faça download da versão mais recente do `.xcframework` de https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. No seu terminal, execute:

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

3. Pegue a saída da etapa anterior e cole-a na propriedade checksum de `.binaryTarget` conforme mostrado abaixo no arquivo `Package.swift` do seu projeto:

```
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

Os lançamentos são publicados via CocoaPods sob o nome `AmazonIVSChatMessaging`. Adicione esta dependência ao seu Podfile:

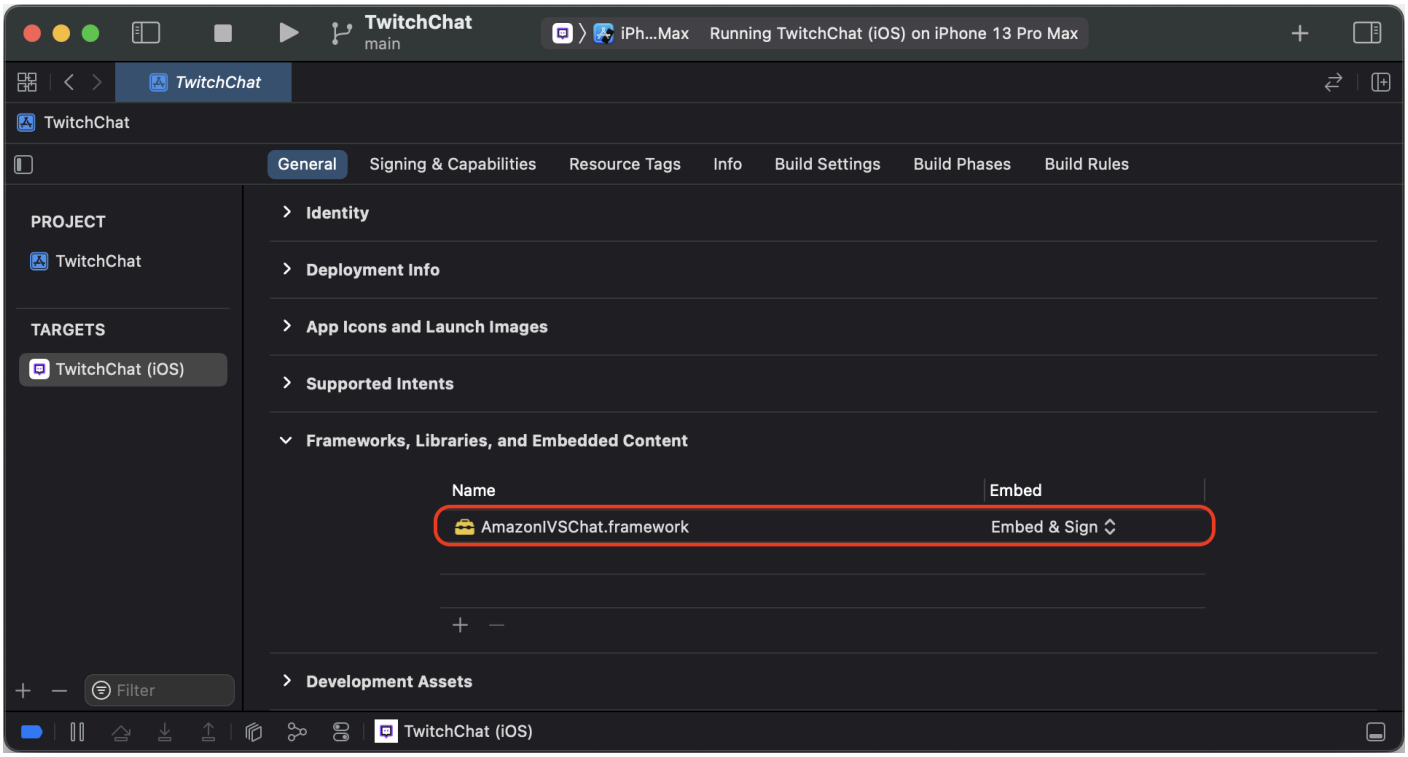
```
pod 'AmazonIVSChat'
```

A execução do `pod install` e do SDK estará disponível em seu `.xcworkspace`.

Instalação manual

1. Faça download da versão mais recente de https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. Extraia o conteúdo do arquivo. `AmazonIVSChatMessaging.xcframework` contém o SDK para dispositivo e para o simulador.

3. Incorpore o `AmazonIVSChatMessaging.xcframework` extraído arrastando-o para a seção Frameworks, Libraries, and Embedded Content (Estruturas, bibliotecas e conteúdo incorporado) da guia General (Geral) para o destino da aplicação:



Usar o SDK para iOS de mensagens para clientes do Chat do IVS

Este documento descreve as etapas envolvidas no uso do SDK para iOS de mensagens para clientes do Chat do Amazon IVS

Conectar a uma sala de chat

Antes de começar, você deve se familiarizar com os Conceitos básicos do Amazon IVS Chat. Veja também os exemplos de aplicações para Web, Android e iOS.

Para se conectar a uma sala de chat, sua aplicação precisa de alguma forma de recuperar um token de chat fornecido pelo backend. Sua aplicação provavelmente recuperará um token de chat usando uma solicitação de rede para seu backend.

Para comunicar esse token de chat buscado com o SDK, o modelo `ChatRoom` do SDK exige que você forneça uma função `async` ou uma instância de um objeto em conformidade com o protocolo `ChatTokenProvider` fornecido no ponto de inicialização. O valor retornado por qualquer um desses métodos precisa ser uma instância do modelo `ChatToken` do SDK.

Observação: as instâncias do modelo `ChatToken` devem ser preenchidas com dados recuperados do seu backend. Os campos necessários para inicializar uma instância do `ChatToken` são os mesmos que os campos na resposta CreateChatToken. Para obter mais informações sobre como inicializar instâncias do modelo `ChatToken`, consulte Criar uma instância de ChatToken. Lembre-se, seu backend é responsável por fornecer os dados na resposta `CreateChatToken` para sua aplicação. A forma como você decide se comunicar com seu backend para gerar tokens de chat depende da sua aplicação e da sua infraestrutura.

Depois de escolher sua estratégia para fornecer um `ChatToken` para o SDK, chame `.connect()` depois de inicializar com êxito uma instância do `ChatRoom` com seu provedor de token e a Região da AWS que seu backend usou para criar a sala de chat à qual você está tentando se conectar. Observe que `.connect()` é uma função assíncrona de lançamento:

```
import AmazonIVSChatMessaging

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

Em conformidade com o protocolo ChatTokenProvider

Para o parâmetro `tokenProvider` no inicializador para `ChatRoom`, é possível fornecer uma instância de `ChatTokenProvider`. Aqui está um exemplo de um objeto em conformidade com `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
      )
   }
```

```
}
```

Em seguida, é possível pegar uma instância desse objeto em conformidade e passá-la para o inicializador para `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()
```

Fornecendo uma função assíncrona no Swift

Suponha que você já tenha um gerenciador que utiliza para gerenciar as solicitações de rede da sua aplicação. A aparência poderá ser semelhante a esta:

```
import AmazonIVSChatMessaging

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

Você poderia simplesmente adicionar outra função em seu gerenciador para recuperar um `ChatToken` do seu backend:

```
import AmazonIVSChatMessaging

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

Em seguida, use a referência a essa função no Swift ao inicializar um `ChatRoom`:

```
import AmazonIVSChatMessaging
```

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

Criar uma instância de ChatToken

É possível criar facilmente uma instância do `ChatToken` usando o inicializador fornecido no SDK. Consulte a documentação em `Token.swift` para saber mais sobre as propriedades no `ChatToken`.

```
import AmazonIVSChatMessaging

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

Usar Decodable

Se, durante a interface com a API do IVS Chat, o backend decidir simplesmente encaminhar a resposta CreateChatToken à aplicação frontend, você poderá aproveitar a conformidade de `ChatToken` com o protocolo `Decodable` da linguagem Swift. No entanto, há um detalhe.

O payload da resposta `CreateChatToken` usa strings para datas que são formatadas usando a norma ISO 8601 para carimbos de data/hora da Internet. Em Swift, normalmente você forneceria `JSONDecoder.DateDecodingStrategy.iso8601` como um valor para a propriedade `.dateDecodingStrategy` de `JSONDecoder`. No entanto, `CreateChatToken` usa segundos fracionários de alta precisão em suas strings, e isso não é aceito por `JSONDecoder.DateDecodingStrategy.iso8601`.

Para sua conveniência, o SDK fornece uma extensão pública em `JSONDecoder.DateDecodingStrategy` com uma estratégia `.preciseISO8601` adicional que permite que você use `JSONDecoder` com sucesso ao decodificar uma instância de`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)
```

Desconectar de uma sala de chat

Para se desconectar manualmente de uma instância do `ChatRoom` à qual você se conectou com êxito, chame `room.disconnect()`. Por padrão, as salas de chat chamam automaticamente essa função quando elas são desalocadas.

```
import AmazonIVSChatMessaging

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

// Disconnect
room.disconnect()
```

Receber uma mensagem/evento de chat

Para enviar e receber mensagens em sua sala de chat, é necessário fornecer um objeto que esteja em conformidade com o protocolo `ChatRoomDelegate` após inicializar com êxito uma instância de `ChatRoom` e chamar `room.connect()`. Aqui está um exemplo típico 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) { ... }
}
```

Seja notificado quando a conexão mudar

Como seria de se esperar, não será possível realizar ações como enviar uma mensagem em uma sala até que a sala esteja totalmente conectada. A arquitetura do SDK tenta incentivar a conexão a um ChatRoom em um thread em segundo plano por meio de APIs assíncronas. Caso deseje criar algo em sua interface de usuário que desative algum objeto, como um botão de envio de mensagem, o SDK fornece duas estratégias para ser notificado quando o estado da conexão de uma sala de chat mudar usando `Combine` ou `ChatRoomDelegate`. Eles são descritos abaixo.

Importante: o estado da conexão de uma sala de chat também pode mudar em função de fatores como uma conexão de rede interrompida. Leve isso em consideração ao criar sua aplicação.

Usar Combine

Cada instância de `ChatRoom` é acompanhada pelo próprio editor `Combine` na forma da propriedade `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()
}
```

Usar ChatRoomDelegate

Como alternativa, use as funções opcionais `roomDidConnect(_:)`, `roomIsConnecting(_:)` e `roomDidDisconnect(_:)` dentro de um objeto que esteja em conformidade com `ChatRoomDelegate`. Exemplo de uso de `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!")
    }
}
```

Executar ações em uma sala de chat

Usuários diferentes têm recursos diferentes para ações que podem realizar em uma sala de chat; por exemplo, enviar mensagens, excluir mensagens ou desconectar usuários. Para realizar uma dessas ações, chame `perform(request:)` em um `ChatRoom` conectado, passando em uma instância de um dos objetos `ChatRequest` fornecidos no SDK. As solicitações compatíveis estão em `Request.swift`.

Algumas ações realizadas em uma sala de chat exigem que os usuários conectados tenham recursos específicos concedidos a eles quando a aplicação de backend chama `CreateChatToken`. Por design, o SDK não consegue discernir os recursos de um usuário conectado. Portanto, embora seja possível tentar realizar ações de moderador em uma instância conectada do `ChatRoom`, a API do ambiente de gerenciamento é que decide se essa ação será bem-sucedida.

Todas as ações que passam por `room.perform(request:)` aguardam até que a sala receba a instância esperada de um modelo (cujo tipo está associado ao próprio objeto de solicitação) compatível com o `requestId` do modelo recebido e do objeto da solicitação. Se houver um problema com a solicitação, `ChatRoom` sempre gera um erro na forma de um `ChatError`. A definição de `ChatError` está em `Error.swift`.

Enviar uma mensagem

Para enviar uma mensagem de chat, use uma instância do `SendMessageRequest`:

```
import AmazonIVSChatMessaging
```

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

Como mencionado acima, `room.perform(request:)` retorna quando um `ChatMessage` correspondente é recebido pelo `ChatRoom`. Se houver um problema com a solicitação (como exceder o limite de caracteres da mensagem para uma sala), uma instância de `ChatError` será iniciada em vez disso. Em seguida, você pode exibir essas informações úteis em sua interface de usuário:

```
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)
}
```

Anexar metadados a uma mensagem

Ao enviar uma mensagem, é possível acrescentar metadados que serão associados a ela. A propriedade `attributes` de `SendMessageRequest` pode ser usada para inicializar sua solicitação. Os dados anexados aqui são anexados à mensagem quando outras pessoas a recebem na sala.

Aqui está um exemplo de como anexar dados remotos a uma mensagem que está sendo enviada:

```
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"
      ]
   )
)
```

Usar `attributes` em um `SendMessageRequest` pode ser extremamente útil para criar recursos complexos em seu produto de chat. Por exemplo, é possível criar a funcionalidade de encadeamento usando o dicionário de atributos `[String : String]` em um `SendMessageRequest`!

A payload `attributes` é muito flexível e poderosa. Use-a para obter informações sobre sua mensagem que você não conseguiria fazer de outra forma. Usar atributos é muito mais fácil do que, por exemplo, analisar a sequência de uma mensagem para obter informações sobre coisas como emotes.

Excluir mensagem

Excluir uma mensagem de chat é como enviar uma. Use a função `room.perform(request:)` em `ChatRoom` para conseguir isso criando uma instância de `DeleteMessageRequest`.

Para acessar facilmente instâncias anteriores de mensagens de chat recebidas, passe o valor de `message.id` ao inicializador de `DeleteMessageRequest`.

Opcionalmente, forneça uma sequência de motivos para `DeleteMessageRequest` para que você possa revelar isso na sua interface de usuário.

```
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!"
   )
)
```

Como essa é uma ação do moderador, seu usuário pode não ter a capacidade de excluir a mensagem de outro usuário. É possível usar a mecânica de funções executáveis do Swift para exibir uma mensagem de erro em sua interface de usuário quando um usuário tentar excluir uma mensagem sem a capacidade apropriada.

Quando o backend chamar `CreateChatToken` para um usuário, ele precisará passar `"DELETE_MESSAGE"` para o campo `capabilities` para ativar essa funcionalidade para um usuário de chat conectado.

Aqui está um exemplo de como detectar um erro de capacidade gerado ao tentar excluir uma mensagem sem as permissões apropriadas:

```
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)
}
```

Desconectar outro usuário

Use `room.perform(request:)` para desconectar outro usuário de uma sala de chat. Especificamente, use uma instância de `DisconnectUserRequest`. Todos os `ChatMessages` recebidos por um `ChatRoom` têm uma propriedade `sender` que contém o ID do usuário que deve ser inicializado adequadamente com uma instância do `DisconnectUserRequest`. Opcionalmente, forneça uma string de motivo para a solicitação de desconexão.

```
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
   )
)
```

Como esse é outro exemplo de ação do moderador, você pode tentar desconectar outro usuário, mas não poderá fazer isso a menos que tenha o recurso `DISCONNECT_USER`. O recurso é definido quando seu aplicativo de backend chama `CreateChatToken` e injeta a string `"DISCONNECT_USER"` no campo `capabilities`.

Se seu usuário não tiver a capacidade de desconectar outro usuário, `room.perform(request:)` iniciará uma instância de `ChatError`, assim como as outras solicitações. Inspesione a propriedade `errorCode` do erro para determinar se a solicitação falhou devido à falta de privilégios de moderador:

```
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 de Mensagens para Clientes do Chat do IVS: Tutorial do iOS

O SDK do Amazon Interactive Video (IVS) Chat Client Messaging iOS fornece interfaces para permitir que você incorpore nossa API do IVS Chat Messaging em plataformas que utilizem a linguagem de programação Swift da Apple.

Para ver um tutorial do SDK do Chat para iOS, consulte https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/latest/tutorials/table-of-contents/.

SDK de Mensagens para Clientes do Chat do IVS: Guia para JavaScript

O Amazon Interactive Video (IVS) Chat Client Messaging JavaScript SDK permite a você incorporar facilmente nossa API do Amazon IVS Chat Messaging em plataformas que utilizam um navegador Web.

Versão mais recente do SDK JavaScript do IVS Chat Client Messaging: 1.0.2 (notas de release)

Documentação de referência: para obter informações sobre os métodos mais importantes disponíveis no Amazon IVS Chat Client Messaging JavaScript SDK, consulte a documentação de referência em https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/

Código de exemplo: consulte o repositório de amostra no GitHub, para uma demonstração específica da Web usando o SDK do JavaScript: https://github.com/aws-samples/amazon-ivs-chat-web-demo

Introdução ao SDK para JavaScript de mensagens para clientes do Chat do IVS

Antes de começar, você deve se familiarizar com os Conceitos básicos do Amazon IVS Chat.

Adicionar o pacote

Use:

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

ou:

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

Suporte ao React Native

O SDK JavaScript do IVS Chat Client Messaging tem uma dependência `uuid` que usa o método `crypto.getRandomValues`. Como esse método não é suportado no React Native, você precisa instalar o polyfill adicional `react-native-get-random-value` e importá-lo na parte superior do arquivo `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);
```

Configurar seu backend

Esta integração exige endpoints em seu servidor que conversem com a API do Amazon IVS Chat. Use as bibliotecas oficiais da AWS para acessar a API do Amazon IVS via servidor. Essas bibliotecas podem ser acessadas em várias linguagens via pacotes públicos, por exemplo, node.js, java e go.

Crie um endpoint de servidor que se comunique com a operação CreateChatToken da API do Chat do Amazon IVS para criar um token de chat para usuários de chat.

Usar o SDK para JavaScript de mensagens para clientes do Chat do IVS

Este documento descreve as etapas envolvidas no uso do SDK para JavaScript de mensagens para clientes do Chat do Amazon IVS

Inicializar uma instância de sala de chat

Crie uma instância da classe `ChatRoom`. Isso requer aprovação de `regionOrUrl` (a região da AWS em que sua sala de chat está hospedada) e `tokenProvider` (o método de busca de tokens será criado na próxima etapa):

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

Função de provedor de tokens

Crie uma função assíncrona de provedor de token que busca um token de chat do seu backend:

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

A função não deve aceitar parâmetros e retornar uma promessa contendo um objeto de token de chat:

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

Essa função é necessária para inicializar o objeto ChatRoom. Abaixo, preencha os campos `<token>` e `<date-time>` com os valores recebidos do seu backend:

```
// You will need to fetch a fresh token each time this method is called by
// the IVS Chat Messaging SDK, since each token is only accepted once.
function tokenProvider(): Promise<ChatToken> {
  // Call your backend to fetch chat token from IVS Chat endpoint:
```

```
  // e.g. const token = await appBackend.getChatToken()
  return {
    token: "<token>",
    sessionExpirationTime: new Date("<date-time>"),
    tokenExpirationTime: new Date("<date-time>")
  }
}
```

Lembre-se de passar o `tokenProvider` para o construtor do ChatRoom. O ChatRoom atualiza o token quando a conexão é interrompida ou a sessão expira. Não use o `tokenProvider` para armazenar um token em qualquer lugar; o ChatRoom cuida disso para você.

Receber eventos

Em seguida, inscreva-se nos eventos da sala de chat para receber eventos do ciclo de vida, bem como mensagens e eventos entregues na sala de chat:

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

Conectar a uma sala de chat

A última etapa da inicialização básica é conectar à sala de chat estabelecendo uma conexão WebSocket. Para fazer isso, basta chamar o método `connect()` dentro da instância da sala.

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

O SDK tentará estabelecer uma conexão com uma sala de chat codificada no token de chat recebido do servidor.

Depois de ligar para `connect()`, a sala fará a transição para o estado `connecting` e emitirá um evento `connecting`. Quando a sala se conecta com sucesso, ela passa para o estado `connected` e emite um evento `connect`.

Uma falha na conexão pode ocorrer devido a problemas ao buscar o token ou ao se conectar ao WebSocket. Nesse caso, a sala tenta se reconectar automaticamente até o número de vezes indicado pelo parâmetro do construtor `maxReconnectAttempts`. Durante as tentativas de reconexão, a sala está no estado `connecting` e não emite eventos adicionais. Depois de esgotar as tentativas de reconexão, a sala passa para o estado `disconnected` e emite um evento `disconnect` (com um motivo de desconexão relevante). No estado `disconnected`, a sala não tenta mais se conectar; você deve ligar para `connect()` novamente para acionar o processo de conexão.

Executar ações em uma sala de chat

O SDK do Amazon IVS Chat Messaging fornece ações ao usuário para enviar mensagens, excluir mensagens e desconectar outros usuários. Eles estão disponíveis na instância do `ChatRoom`. Eles devolvem um objeto `Promise` que permite que você receba a confirmação ou rejeição da solicitação.

Enviar uma mensagem

Para esta solicitação, é necessário ter a capacidade `SEND_MESSAGE` codificada em seu token de chat.

Para acionar uma solicitação de envio de mensagem:

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

Para obter uma confirmação ou rejeição da solicitação,`await` a promessa devolvida ou use o método `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.
}
```

Excluir mensagem

Para esta solicitação, é necessário ter a capacidade `DELETE_MESSAGE` codificada em seu token de chat.

Para excluir uma mensagem para fins de moderação, chame o método `deleteMessage()`:

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

Para obter uma confirmação ou rejeição da solicitação,`await` a promessa devolvida ou use o método `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.
}
```

Desconectar outro usuário

Para esta solicitação, é necessário ter a capacidade `DISCONNECT_USER` codificada em seu token de chat.

Para desconectar outro usuário para fins de moderação, chame o método `disconnectUser()`:

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

Para obter uma confirmação ou rejeição da solicitação,`await` a promessa devolvida ou use o método `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.
}
```

Desconectar de uma sala de chat

Para fechar sua conexão com a sala de chat, chame o método `disconnect()` na instância `room`:

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

Chamar esse método faz com que a sala feche o WebSocket subjacente de maneira ordenada. A instância da sala faz a transição para um estado `disconnected` e emite um evento de desconexão, com o motivo `disconnect` definido como`"clientDisconnect"`.

SDK de Mensagens para Clientes do Chat do IVS: Tutorial de JavaScript, parte 1: salas de chat

Esta é a primeira de um tutorial de duas partes. Você aprenderá os fundamentos do trabalho com o SDK JavaScript do Amazon IVS Chat Client Messaging criando uma aplicação totalmente funcional usando JavaScript/TypeScript. Chamamos a aplicação de Chatterbox.

O público-alvo são desenvolvedores experientes, mas iniciantes no SDK Amazon IVS Chat Messaging. Você deve se sentir confortável com a linguagem de programação JavaScript/TypeScript e a biblioteca React.

Para resumir, vamos nos referir ao SDK JavaScript do Amazon IVS Chat Client Messaging como o SDK do Chat JS.

Observação: em alguns casos, os exemplos de código para JavaScript e TypeScript são idênticos, então eles são combinados.

Esta primeira parte do tutorial está dividida em várias seções:

1. the section called “Configure um servidor local de autenticação/autorização”
2. the section called “Crie um projeto de Chatterbox”
3. the section called “Conectar a uma sala de chat”
4. the section called “Crie um provedor de tokens”
5. the section called “Observe as atualizações de conexão”
6. the section called “Crie um componente do botão Enviar”
7. the section called “Crie uma entrada de mensagem”
8. the section called “Próximas etapas”

Para obter a documentação completa do SDK, comece com o SDK de Mensagens para Clientes do Chat do Amazon IVS (aqui no Guia do usuário do Chat do Amazon IVS) e a Referência de Mensagens para Clientes do Chat: SDK para JavaScript (no GitHub).

Pré-requisitos

- Familiarize-se com o JavaScript/TypeScript e a biblioteca React. Se você não estiver familiarizado com o React, aprenda o básico neste Tic-Tac-Toe Tutorial.
- Leia e compreenda Conceitos básicos do Chat do IVS.
- Crie um usuário do AWS IAM com os recursos createChatToken e createRoom definidos em uma política do IAM existente. (Consulte Conceitos básicos do Chat do IVS.)
- Certifique-se de que as chaves secretas/de acesso desse usuário estejam armazenadas em um arquivo de credenciais da AWS. Para obter instruções, consulte o Guia do usuário da AWS CLI (especialmente Configuração e definições do arquivo de credenciais).
- Crie uma sala de chat e salve seu ARN. Consulte Conceitos básicos do Chat do IVS. (Se você não salvar o ARN, poderá consultá-lo posteriormente com o console ou a API do Chat.)
- Instale o ambiente Node.js 14+ com o gerenciador de pacotes NPM ou Yarn.

Configure um servidor local de autenticação/autorização

Sua aplicação de backend é responsável por criar salas de chat e gerar os tokens de chat necessários para que o SDK do Chat JS autentique e autorize seus clientes em suas salas de chat. Você deverá usar seu próprio backend, pois não é possível armazenar com segurança as chaves da AWS em uma aplicação móvel. Invasores sofisticados podem extraí-las e obter acesso à sua conta da AWS.

Consulte Criar um token de chat em Introdução ao Amazon IVS Chat. Conforme mostrado no fluxograma, sua aplicação do lado do servidor é responsável por criar um token de chat. Isso significa que sua aplicação deve fornecer seu próprio meio de gerar um token de chat solicitando-o da sua aplicação a partir do lado do servidor.

Nesta seção, você aprenderá os fundamentos da criação de um provedor de tokens em seu backend. Usamos a estrutura expressa para criar um servidor local ativo que gerencia a criação de tokens de chat usando seu ambiente local da AWS.

Crie um projeto npm vazio usando o NPM. Crie um diretório para manter sua aplicação e torne-o seu diretório de trabalho:

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

Use `npm init` para criar um arquivo `package.json` para sua aplicação:

```
$ npm init
```

Esse comando solicita várias coisas, incluindo o nome e a versão da sua aplicação. Por enquanto, basta pressionar RETURN para aceitar os padrões da maioria deles, com a seguinte exceção:

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

Pressione RETURN para aceitar o nome de arquivo padrão sugerido `index.js` ou digite o que você quiser que seja o nome do arquivo principal.

Agora instale as dependências necessárias:

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

`aws-sdk` requer variáveis de ambiente de configuração que são carregadas automaticamente de um arquivo chamado `.env` localizado no diretório raiz. Para configurá-lo, crie um novo arquivo chamado `.env` e preencha as informações de configuração ausentes:

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

Agora, criamos um arquivo de ponto de entrada no diretório raiz com o nome que você inseriu acima no comando `npm init`. Nesse caso, usamos `index.js` e importamos todos os pacotes necessários:

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

Agora, crie uma nova instância de `express`:

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

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

Depois disso, será possível criar seu primeiro método POST de endpoint para o provedor do token. Pegue os parâmetros necessários do corpo da solicitação (`roomId`, `userId`, `capabilities` e `sessionDurationInMinutes`):

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

Adicione a validação dos campos obrigatórios:

```
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;
  }
});
```

Depois de preparar o método POST, integramos `createChatToken` com `aws-sdk` para a funcionalidade principal de autenticação/autorização:

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

No final do arquivo, adicione um receptor de portas para sua aplicação `express`:

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

Agora é possível executar o servidor com a linha de comando a seguir na raiz do projeto:

```
$ node index.js
```

Dica: este servidor aceita solicitações de URL em https://localhost:3000.

Crie um projeto de Chatterbox

Primeiro você cria o projeto React chamado `chatterbox`. Execute este comando:

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

É possível integrar o SDK do Chat Client Messaging JS por meio do Gerenciador de pacotes de nó ou do Gerenciador de pacotes Yarn:

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

Conectar a uma sala de chat

Aqui você cria uma `ChatRoom` e se conecta a ela usando métodos assíncronos. A classe `ChatRoom` gerencia a conexão do usuário com o SDK do Chat JS. Para se conectar com sucesso a uma sala de chat, você deve fornecer uma instância de `ChatToken` dentro da sua aplicação React.

Navegue até o arquivo App criado no projeto `chatterbox` padrão e exclua tudo entre as duas etiquetas `<div>`. Nenhum código pré-preenchido é necessário. Neste ponto, a nossa App está bem vazia.

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

import * as React from 'react';

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

Crie uma nova instância de `ChatRoom` e passe-a para o estado usando o gancho `useState`. Ela exige a passagem de `regionOrUrl` (a região da AWS na qual sua sala de chat está hospedada) e `tokenProvider` (usado para o fluxo de autenticação/autorização de backend criado nas etapas subsequentes).

Importante: você deve usar a mesma região da AWS em que criou a sala em Conceitos básicos do Amazon IVS Chat. A API é um serviço regional da AWS. Para obter uma lista das regiões com

suporte e dos endpoints do serviço HTTPS do Amazon IVS Chat, consulte a página de Regiões do 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>;
}
```

Crie um provedor de tokens

Como próxima etapa, precisamos criar uma função `tokenProvider` sem parâmetros que seja exigida pelo construtor `ChatRoom`. Primeiro, criaremos uma função `fetchChatToken` que fará uma solicitação POST para a aplicação de backend que você configurou em the section called "Configure um servidor local de autenticação/autorização". Os tokens de chat contêm as informações necessárias para que o SDK estabeleça uma conexão com a sala de chat com êxito. A API do Chat usa esses tokens como uma forma segura de validar a identidade, os recursos de um usuário em uma sala de chat e a duração da sessão.

No navegador do projeto, crie um novo arquivo TypeScript/JavaScript chamado `fetchChatToken`. Crie uma solicitação de busca para a aplicação `backend` e retorne o objeto `ChatToken` da resposta. Adicione as propriedades do corpo da solicitação necessárias para a criação de um token de chat. Use as regras definidas para nomes do recurso da Amazon (ARNs). Essas propriedades estão documentadas na operação CreateChatToken.

Observação: o URL que você está usando aqui é o mesmo URL que seu servidor local criou quando você executou a aplicação de 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),
  };
}
```

Observe as atualizações de conexão

Reagir às mudanças no estado da conexão de uma sala de chat é parte essencial da criação de uma aplicação de chat. Vamos começar assinando eventos relevantes:

```
// 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>;
}
```

Em seguida, precisamos fornecer a capacidade de ler o estado da conexão. Usamos nosso hook `useState` para criar algum estado local em `App` e definir o estado da conexão dentro de cada receptor.

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>;
}
```

Depois de se inscrever no estado da conexão, exiba o estado da conexão e conecte-se à sala de chat usando o método `room.connect` dentro do gancho `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>
);

// ...
```

Você implementou com êxito uma conexão de sala de chat.

Crie um componente do botão Enviar

Nesta seção, você criará um botão de envio que tem um design diferente para cada estado de conexão. O botão de envio facilita enviar mensagens em uma sala de chat. Ele também serve como um indicador visual de se/quando as mensagens podem ser enviadas; por exemplo, em caso de conexões perdidas ou sessões de chat expiradas.

Primeiro, crie um novo arquivo no diretório `src` do seu projeto do Chatterbox e dê a ele o nome `SendButton`. Em seguida, crie um componente que exibirá um botão para sua aplicação de chat. Exporte seu `SendButton` e importe-o para `App`. No `<div></div>` vazio, adicione `<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>
);
```

Em seguida, em App, defina uma função chamada onMessageSend e passe-a para a propriedade SendButton onPress. Defina outra variável chamada isSendDisabled (que impede o envio de mensagens quando a sala não estiver conectada) e passe-a para a propriedade 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>
);

// ...
```

Crie uma entrada de mensagem

A barra de mensagens do Chatterbox é o componente com o qual você interagirá para enviar mensagens para uma sala de chat. Normalmente, ela contém uma entrada de texto para redigir sua mensagem e um botão para enviar sua mensagem.

Para criar um componente MessageInput, primeiro crie um novo arquivo no diretório src e dê a ele o nome MessageInput. Em seguida, crie um componente de entrada controlada que exibirá uma entrada para sua aplicação de chat. Exporte sua MessageInput e importe-a para App (acima do <SendButton />).

Crie um novo estado chamado `messageToSend` usando o hook `useState`, com uma string vazia como valor padrão. No corpo da sua aplicação, passe `messageToSend` para o `value` de `MessageInput` e passe `setMessageToSend` para a propriedade `onMessageChange`:

TypeScript

```
// MessageInput.tsx

import * as React from 'react';

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

export const MessageInput = ({ value, onValueChange }: Props) => {
  return (
    <input type="text" value={value} onChange={(e) => onValueChange?.
(e.target.value)} placeholder="Send a message" />
  );
};


// App.tsx

// ...

import { MessageInput } from './MessageInput';

// ...

export default function App() {
  const [messageToSend, setMessageToSend] = useState('');

// ...

return (
  <div>
    <h4>Connection State: {connectionState}</h4>
    <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
    <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
  </div>
);
```

JavaScript

```
// MessageInput.jsx

import * as React from 'react';

export const MessageInput = ({ value, onValueChange }) => {
  return (
    <input type="text" value={value} onChange={(e) => onValueChange?.
(e.target.value)} placeholder="Send a message" />
  );
};

// App.jsx

// ...

import { MessageInput } from './MessageInput';

// ...

export default function App() {
  const [messageToSend, setMessageToSend] = useState('');

// ...


return (
  <div>
    <h4>Connection State: {connectionState}</h4>
    <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
    <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
  </div>
);
```

Próximas etapas

Agora que você terminou de criar uma barra de mensagens para o Chatterbox, vá para a parte 2 deste tutorial de JavaScript, mensagens e eventos.

SDK do IVS Chat Client Messaging: Tutorial de JavaScript, parte 2: mensagens e eventos

Esta segunda e última parte do tutorial é dividida em várias seções:

1. the section called “Inscreva-se em eventos de mensagens de chat”
2. the section called “Exibir mensagens recebidas”
   a. the section called “Criação de um componente de mensagem”
   b. the section called “Reconhecimento das mensagens enviadas pelo usuário atual”
   c. the section called “Criação de um componente de lista mensagens”
   d. the section called “Renderização de uma lista de mensagens de chat”
3. the section called “Executar ações em uma sala de chat”
   a. the section called “Enviar uma mensagem”
   b. the section called “Excluir mensagem”
4. the section called “Próximas etapas”

Observação: em alguns casos, os exemplos de código para JavaScript e TypeScript são idênticos, então eles são combinados.

Para obter a documentação completa do SDK, comece com o SDK de Mensagens para Clientes do Chat do Amazon IVS (aqui no Guia do usuário do Chat do Amazon IVS) e a Referência de Mensagens para Clientes do Chat: SDK para JavaScript (no GitHub).

Pré-requisito

Certifique-se de ter concluído a parte 1 deste tutorial, salas de chat.

Inscreva-se em eventos de mensagens de chat

A instância `ChatRoom` usa eventos para se comunicar, quando os eventos ocorrem em uma sala de chat. Para começar a implementar a experiência de chat, você precisa mostrar aos usuários quando as outras pessoas enviam uma mensagem na sala à qual estão conectados.

Aqui, você se inscreve em eventos de mensagens de chat Posteriormente, mostraremos como atualizar uma lista de mensagens que você criou e é atualizada com cada mensagem/evento.

Em seu `App`, dentro do hook `useEffect`, inscreva-se em todos os eventos de mensagens:

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

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

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

Exibir mensagens recebidas

Receber mensagens é parte essencial da experiência de chat. Usando o SDK do Chat JS, é possível configurar seu código para receber facilmente eventos de outros usuários conectados a uma sala de chat.

Posteriormente, mostraremos como realizar ações em uma sala de chat que utilizam os componentes criados por você aqui.

Em sua `App`, defina um estado chamado `messages`, com um tipo de matriz `ChatMessage` chamado `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([]);

  //...
}
```

Em seguida, na função de receptor da `message`, acrescente `message` à matriz `messages`:

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

// ...

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

// ...
```

Abaixo, analisamos as tarefas para mostrar as mensagens recebidas:

1. the section called “Criação de um componente de mensagem”
2. the section called “Reconhecimento das mensagens enviadas pelo usuário atual”
3. the section called “Criação de um componente de lista mensagens”
4. the section called “Renderização de uma lista de mensagens de chat”

Criação de um componente de mensagem

O componente `Message` é responsável por renderizar o conteúdo de uma mensagem recebida pela sua sala de chat. Nesta seção, você cria um componente de mensagens para renderizar mensagens de chat individuais na `App`.

Crie um novo arquivo no diretório `src` e atribua a ele o nome `Message`. Passe o tipo `ChatMessage` para esse componente e, em seguida, passe a string `content` das propriedades de `ChatMessage` para exibir o texto da mensagem recebida dos receptores de mensagens da sala de chat. No Project Navigator, acesse `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>
  );
};
```

Dica: use este componente para armazenar propriedades diferentes que você deseja renderizar em suas linhas de mensagens; por exemplo, URLs de avatar, nomes de usuário e carimbos de data e hora de quando a mensagem foi enviada.

Reconhecimento das mensagens enviadas pelo usuário atual

Para reconhecer a mensagem enviada pelo usuário atual, modificamos o código e criamos um contexto do React para armazenar o `userId` do usuário atual.

Crie um novo arquivo no diretório `src` e atribua a ele o nome `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>;
};
```

Observação: aqui usamos o hook `useState` para armazenar o valor `userId`. No futuro, será possível usar `setUserId` para alterar o contexto do usuário ou para fins de login.

Em seguida, substitua `userId` no primeiro parâmetro passado para `tokenProvider` usando o contexto criado anteriormente:

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

  // ...
}
```

Em seu componente `Message`, use o `UserContext` criado antes, declare a variável `isMine`, corresponda o `userId` do remetente com o `userId` do contexto e aplique estilos diferentes de mensagens para o usuário atual.

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>
  );
};
```

Criação de um componente de lista mensagens

O componente `MessageList` é responsável por exibir a conversa de uma sala de chat ao longo do tempo. O arquivo `MessageList` é o contêiner que contém todas as nossas mensagens. `Message` é uma linha de `MessageList`.

Crie um novo arquivo no diretório `src` e atribua a ele o nome `MessageList`. Defina `Props` com `messages` do tipo matriz de `ChatMessage`. Dentro do corpo, mapeie nossa propriedade `messages` e passe `Props` para seu 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>
  );
};
```

Renderização de uma lista de mensagens de chat

Agora, coloque o novo `MessageList` em seu componente `App` principal:

```
// 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>
);

// ...
```

Todas as peças do quebra-cabeça estão prontas para que sua App comece a renderizar as mensagens recebidas pela sua sala de chat. Continue abaixo para ver como realizar ações em uma sala de chat que aproveitem os componentes que você criou.

Executar ações em uma sala de chat

Enviar mensagens e realizar ações de moderador em uma sala de chat são algumas das principais formas de interagir com uma sala de chat. Aqui você aprenderá como usar vários objetos `ChatRequest` para realizar ações comuns no Chatterbox, como enviar uma mensagem, excluir uma mensagem e desconectar outros usuários.

Todas as ações em uma sala de chat seguem um padrão comum: para cada ação executada em uma sala de chat, há um objeto de solicitação correspondente. Para cada solicitação, há um objeto de resposta correspondente que você recebe na confirmação da solicitação.

Desde que seus usuários recebam as permissões corretas quando você criar um token de chat, eles poderão realizar com êxito as ações correspondentes usando os objetos de solicitação para ver quais solicitações são possíveis de ser realizadas em uma sala de chat.

Abaixo, explicamos como enviar uma mensagem e excluir uma mensagem.

Enviar uma mensagem

A classe `SendMessageRequest` permite o envio de mensagens em uma sala de chat. Aqui, você modifica sua App para enviar uma solicitação de mensagem usando o componente que criou em Criar uma entrada de mensagem (na parte 1 deste tutorial).

Para começar, defina uma nova propriedade booleana chamada de `isSending` com o hook `useState`. Use essa nova propriedade para alternar o estado desativado do seu elemento HTML `button` usando a constante `isSendDisabled`. No manipulador de eventos do seu `SendButton`, limpe o valor de `messageToSend` e defina `isSending` como verdadeiro.

Como você fará uma chamada de API a partir desse botão, adicionar o booleano *`isSending`* ajuda a evitar que várias chamadas de API ocorram ao mesmo tempo, desativando as interações do usuário no seu *`SendButton`* até que a solicitação seja concluída.

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

// ...

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

// ...

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

// ...

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

// ...
```

Prepare a solicitação criando uma nova instância `SendMessageRequest`, passando o conteúdo da mensagem para o construtor. Depois de definir os estados `isSending` e `messageToSend`, chame o método `sendMessage`, que envia a solicitação para a sala de chat. Por fim, limpe o sinalizador `isSending` ao receber a confirmação ou rejeição da solicitação.

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);
  }
};

// ...
```

Experimente o Chatterbox: tente enviar uma mensagem redigindo uma com a sua `MessageInput` e tocando no seu `SendButton`. Você deve ver sua mensagem enviada renderizada dentro da `MessageList` que você criou anteriormente.

Excluir mensagem

Para excluir uma mensagem de uma sala de chat, você precisa ter a capacidade adequada. As capacidades são concedidas durante a inicialização do token de chat que você usa ao se autenticar em uma sala de chat. Para os propósitos desta seção, a `ServerApp` de Configure um servidor local de autenticação/autorização (na parte 1 deste tutorial) permite que você especifique as capacidades de moderador. Isso é feito em sua aplicação usando o objeto `tokenProvider` que você criou em Crie um provedor de tokens (também na parte 1).

Aqui você modifica sua `Message` adicionando uma função para excluir a mensagem.

Primeiro, abra `App.tsx` e adicione a capacidade `DELETE_MESSAGE`. (`capabilities` é o segundo parâmetro da sua função `tokenProvider`.)

Observação: é assim que sua `ServerApp` informa às APIs do IVS Chat que o usuário associado ao token de chat resultante pode excluir mensagens em uma sala de chat. Em uma situação real, você provavelmente terá uma lógica de backend mais complexa para gerenciar os recursos do usuário na infraestrutura da sua aplicação de servidor.

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']),
  }),
);

// ...
```

Nas próximas etapas, você atualizará sua `Message` para exibir um botão de exclusão.

Abra `Message` e defina um novo estado booleano chamado `isDeleting` usando o hook `useState` com um valor inicial de `false`. Usando esse estado, atualize o conteúdo do seu `Button` para ser diferente, dependendo do estado atual de `isDeleting`. Desative seu botão quando `isDeleting` for verdadeiro. Isso evita que você tente fazer duas solicitações de exclusão de mensagens ao mesmo tempo.

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>
  );
};
```

Defina uma nova função chamada onDelete que aceite uma string como um de seus parâmetros e retorne Promise. No corpo do encerramento da ação do seu Button, use setIsDeleting para alternar seu booleano isDeleting antes e depois de uma chamada para onDelete. Para o parâmetro de string, passe o ID da mensagem do 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>
  );
};
```

Em seguida, atualize sua `MessageList` para refletir as alterações mais recentes em seu componente `Message`.

Abra `MessageList` e defina uma nova função chamada `onDelete` que aceite uma string como um parâmetro e retorne `Promise`. Atualize a `Message` e passe-a pelas propriedades da `Message`. O parâmetro de string em seu novo encerramento será o ID da mensagem que você deseja excluir, que será passada a partir da sua `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} />
      ))}
    </>
  );
};
```

Em seguida, você atualiza sua App para refletir as alterações mais recentes em sua `MessageList`.

Em App, defina uma função chamada on`DeleteMessage` e passe-a para a propriedade `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>
);

// ...
```

Prepare uma solicitação criando uma nova instância de `DeleteMessageRequest`, passando o ID da mensagem relevante para o parâmetro do construtor e chame `deleteMessage`, que aceita a solicitação preparada acima:

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);
};

// ...
```

Em seguida, atualize o estado de `messages` para refletir uma nova lista de mensagens que omita a mensagem que você acabou de excluir.

No hook `useEffect`, receba o evento `messageDelete` e atualize sua matriz de estados de `messages` excluindo a mensagem com um ID correspondente ao parâmetro `message`.

Observação: o evento `messageDelete` pode ser gerado para que as mensagens sejam excluídas pelo usuário atual ou por qualquer outro usuário na sala. Manipulá-lo no manipulador de eventos (em vez de junto à solicitação `deleteMessage`) permite unificar o tratamento da exclusão de mensagens.

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

// ...

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

return () => {
  // ...
```

```
  unsubscribeOnMessageDeleted();
};

// ...
```

Agora é possível excluir usuários de uma sala de chat na sua aplicação de chat.

Próximas etapas

A título de experimento, tente implementar outras ações em uma sala, como desconectar um outro usuário.

SDK de Mensagens para Clientes do Chat do IVS: Tutorial do React Native, parte 1: salas de chat

Esta é a primeira de um tutorial de duas partes. Você aprenderá os fundamentos do trabalho com o SDK de Mensagens do JavaScript para Clientes do Chat do Amazon IVS ao criar uma aplicação totalmente funcional usando o React Native. Chamamos a aplicação de Chatterbox.

O público-alvo são desenvolvedores experientes, mas iniciantes no SDK Amazon IVS Chat Messaging. Você deve ficar confortável com as linguagens de programação TypeScript ou JavaScript e a biblioteca React Native.

Para resumir, vamos nos referir ao SDK JavaScript do Amazon IVS Chat Client Messaging como o SDK do Chat JS.

Observação: em alguns casos, os exemplos de código para JavaScript e TypeScript são idênticos, então eles são combinados.

Esta primeira parte do tutorial está dividida em várias seções:

1. the section called “Configure um servidor local de autenticação/autorização”
2. the section called “Crie um projeto de Chatterbox”
3. the section called “Conectar a uma sala de chat”
4. the section called “Crie um provedor de tokens”
5. the section called “Observe as atualizações de conexão”

6. the section called “Crie um componente do botão Enviar”
7. the section called “Crie uma entrada de mensagem”
8. the section called “Próximas etapas”

Pré-requisitos

- Familiarize-se com o TypeScript ou o JavaScript e com a biblioteca React Native. Se você não estiver familiarizado com o React Native, aprenda o básico em Intro to React Native (Introdução ao React Native).
- Leia e compreenda Conceitos básicos do Chat do IVS.
- Crie um usuário do AWS IAM com os recursos createChatToken e createRoom definidos em uma política do IAM existente. (Consulte Conceitos básicos do Chat do IVS.)
- Certifique-se de que as chaves secretas/de acesso desse usuário estejam armazenadas em um arquivo de credenciais da AWS. Para obter instruções, consulte o Guia do usuário da AWS CLI (especialmente Configuração e definições do arquivo de credenciais).
- Crie uma sala de chat e salve seu ARN. Consulte Conceitos básicos do Chat do IVS. (Se você não salvar o ARN, poderá consultá-lo posteriormente com o console ou a API do Chat.)
- Instale o ambiente Node.js 14+ com o gerenciador de pacotes NPM ou Yarn.

Configure um servidor local de autenticação/autorização

Sua aplicação de backend é responsável por criar salas de chat e gerar os tokens de chat necessários para que o SDK do Chat JS autentique e autorize seus clientes em suas salas de chat. Você deverá usar seu próprio backend, pois não é possível armazenar com segurança as chaves da AWS em uma aplicação móvel. Invasores sofisticados podem extraí-las e obter acesso à sua conta da AWS.

Consulte Criar um token de chat em Introdução ao Amazon IVS Chat. Conforme mostrado no fluxograma, sua aplicação do lado do servidor é responsável por criar um token de chat. Isso significa que sua aplicação deve fornecer seu próprio meio de gerar um token de chat solicitando-o da sua aplicação a partir do lado do servidor.

Nesta seção, você aprenderá os fundamentos da criação de um provedor de tokens em seu backend. Usamos a estrutura expressa para criar um servidor local ativo que gerencia a criação de tokens de chat usando seu ambiente local da AWS.

Crie um projeto npm vazio usando o NPM. Crie um diretório para manter sua aplicação e torne-o seu diretório de trabalho:

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

Use `npm init` para criar um arquivo `package.json` para sua aplicação:

```
$ npm init
```

Esse comando solicita várias coisas, incluindo o nome e a versão da sua aplicação. Por enquanto, basta pressionar RETURN para aceitar os padrões da maioria deles, com a seguinte exceção:

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

Pressione RETURN para aceitar o nome de arquivo padrão sugerido `index.js` ou digite o que você quiser que seja o nome do arquivo principal.

Agora instale as dependências necessárias:

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

`aws-sdk` requer variáveis de ambiente de configuração que são carregadas automaticamente de um arquivo chamado `.env` localizado no diretório raiz. Para configurá-lo, crie um novo arquivo chamado `.env` e preencha as informações de configuração ausentes:

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

Agora, criamos um arquivo de ponto de entrada no diretório raiz com o nome que você inseriu acima no comando `npm init`. Nesse caso, usamos `index.js` e importamos todos os pacotes necessários:

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

Agora, crie uma nova instância de `express`:

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

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

Depois disso, será possível criar seu primeiro método POST de endpoint para o provedor do token. Pegue os parâmetros necessários no corpo da solicitação (`roomId`, `userId`, `capabilities` e `sessionDurationInMinutes`):

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

Adicione a validação dos campos obrigatórios:

```
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;
  }
});
```

Depois de preparar o método POST, integramos `createChatToken` com `aws-sdk` para a funcionalidade principal de autenticação/autorização:

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

No final do arquivo, adicione um receptor de portas para sua aplicação `express`:

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

Agora é possível executar o servidor com a linha de comando a seguir na raiz do projeto:

```
$ node index.js
```

Dica: este servidor aceita solicitações de URL em https://localhost:3000.

Crie um projeto de Chatterbox

Primeiro, você cria o projeto do React denominado `chatterbox`. Execute este comando:

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

Ou crie um projeto de exposição com um modelo do TypeScript.

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

É possível integrar o SDK do Chat Client Messaging JS por meio do Gerenciador de pacotes de nó ou do Gerenciador de pacotes Yarn:

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

Conectar a uma sala de chat

Aqui você cria uma `ChatRoom` e se conecta a ela usando métodos assíncronos. A classe `ChatRoom` gerencia a conexão do usuário com o SDK do Chat JS. Para se conectar com sucesso a uma sala de chat, você deve fornecer uma instância de `ChatToken` dentro da sua aplicação React.

Navegue até o arquivo App criado no projeto `chatterbox` padrão e exclua tudo que for retornado por um componente funcional. Nenhum código pré-preenchido é necessário. Neste ponto, a nossa App está bem vazia.

TypeScript/JavaScript:

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

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

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

Crie uma nova instância de `ChatRoom` e passe-a para o estado usando o hook `useState`. Ela exige a passagem de `regionOrUrl` (a região da AWS na qual sua sala de chat está hospedada) e `tokenProvider` (usado para o fluxo de autenticação/autorização de backend criado nas etapas subsequentes).

Importante: você deve usar a mesma região da AWS em que criou a sala em Conceitos básicos do Amazon IVS Chat. A API é um serviço regional da AWS. Para obter uma lista das regiões com

suporte e dos endpoints do serviço HTTPS do Amazon IVS Chat, consulte a página de Regiões do 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>;
}
```

Crie um provedor de tokens

Como próxima etapa, precisamos criar uma função `tokenProvider` sem parâmetros que seja exigida pelo construtor `ChatRoom`. Primeiro, criaremos uma função `fetchChatToken` que fará uma solicitação POST para a aplicação de backend que você configurou em the section called "Configure um servidor local de autenticação/autorização". Os tokens de chat contêm as informações necessárias para que o SDK estabeleça uma conexão com a sala de chat com êxito. A API do Chat usa esses tokens como uma forma segura de validar a identidade, os recursos de um usuário em uma sala de chat e a duração da sessão.

No navegador do projeto, crie um novo arquivo TypeScript/JavaScript chamado `fetchChatToken`. Crie uma solicitação de busca para a aplicação `backend` e retorne o objeto `ChatToken` da resposta. Adicione as propriedades do corpo da solicitação necessárias para a criação de um token de chat. Use as regras definidas para nomes do recurso da Amazon (ARNs). Essas propriedades estão documentadas na operação CreateChatToken.

Observação: o URL que você está usando aqui é o mesmo URL que seu servidor local criou quando você executou a aplicação de 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),
  };
}
```

Observe as atualizações de conexão

Reagir às mudanças no estado da conexão de uma sala de chat é parte essencial da criação de uma aplicação de chat. Vamos começar assinando eventos relevantes:

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>;
}
```

Em seguida, precisamos fornecer a capacidade de ler o estado da conexão. Usamos nosso hook `useState` para criar algum estado local em `App` e definir o estado da conexão dentro de cada receptor.

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>;
}
```

Depois de se inscrever no estado da conexão, exiba o estado da conexão e conecte-se à sala de chat usando o método `room.connect` dentro do 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,
  }
});

// ...
```

Você implementou com êxito uma conexão de sala de chat.

Crie um componente do botão Enviar

Nesta seção, você criará um botão de envio que tem um design diferente para cada estado de conexão. O botão de envio facilita enviar mensagens em uma sala de chat. Ele também serve como um indicador visual de se/quando as mensagens podem ser enviadas; por exemplo, em caso de conexões perdidas ou sessões de chat expiradas.

Primeiro, crie um novo arquivo no diretório `src` do seu projeto do Chatterbox e dê a ele o nome `SendButton`. Em seguida, crie um componente que exibirá um botão para sua aplicação de chat. Exporte seu `SendButton` e importe-o para `App`. No `<View></View>` vazio, adicione `<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>
);
```

Em seguida, em App, defina uma função chamada onMessageSend e passe-a para a propriedade SendButton onPress. Defina outra variável chamada isSendDisabled (que impede o envio de mensagens quando a sala não estiver conectada) e passe-a para a propriedade 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>
);

// ...
```

Crie uma entrada de mensagem

A barra de mensagens do Chatterbox é o componente com o qual você interagirá para enviar mensagens para uma sala de chat. Normalmente, ela contém uma entrada de texto para redigir sua mensagem e um botão para enviar sua mensagem.

Para criar um componente MessageInput, primeiro crie um novo arquivo no diretório src e dê a ele o nome MessageInput. Em seguida, crie um componente de entrada que exibirá uma entrada para sua aplicação de chat. Exporte sua MessageInput e importe-a para App (acima do <SendButton />).

Crie um novo estado chamado `messageToSend` usando o hook `useState`, com uma string vazia como valor padrão. No corpo da sua aplicação, passe `messageToSend` para o `value` de `MessageInput` e passe `setMessageToSend` para a propriedade 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 (
    <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',
  }
```

```
});
```

Próximas etapas

Agora que você terminou de criar uma barra de mensagens para o Chatterbox, vá para a parte 2 deste tutorial do React Native, Mensagens e eventos.

SDK de Mensagens para Clientes do Chat do IVS: Tutorial do React Native, parte 2: mensagens e eventos

Esta segunda e última parte do tutorial é dividida em várias seções:

1. the section called “Inscreva-se em eventos de mensagens de chat”
2. the section called “Exibir mensagens recebidas”
   a. the section called “Criação de um componente de mensagem”
   b. the section called “Reconhecimento das mensagens enviadas pelo usuário atual”
   c. the section called “Renderização de uma lista de mensagens de chat”
3. the section called “Executar ações em uma sala de chat”
   a. the section called “Enviar uma mensagem”
   b. the section called “Excluir mensagem”
4. the section called “Próximas etapas”

Observação: em alguns casos, os exemplos de código para JavaScript e TypeScript são idênticos, então eles são combinados.

Pré-requisito

Certifique-se de ter concluído a parte 1 deste tutorial, salas de chat.

Inscreva-se em eventos de mensagens de chat

A instância `ChatRoom` usa eventos para se comunicar, quando os eventos ocorrem em uma sala de chat. Para começar a implementar a experiência de chat, você precisa mostrar aos usuários quando as outras pessoas enviam uma mensagem na sala à qual estão conectados.

Aqui, você se inscreve em eventos de mensagens de chat Posteriormente, mostraremos como atualizar uma lista de mensagens que você criou e é atualizada com cada mensagem/evento.

Em seu App, dentro do hook `useEffect`, inscreva-se em todos os eventos de mensagens:

TypeScript/JavaScript:

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

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

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

Exibir mensagens recebidas

Receber mensagens é parte essencial da experiência de chat. Usando o SDK do Chat JS, é possível configurar seu código para receber facilmente eventos de outros usuários conectados a uma sala de chat.

Posteriormente, mostraremos como realizar ações em uma sala de chat que utilizam os componentes criados por você aqui.

Em sua App, defina um estado chamado `messages`, com um tipo de matriz `ChatMessage` chamado `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([]);

  //...
}
```

Em seguida, na função de receptor da `message`, acrescente `message` à matriz `messages`:

TypeScript/JavaScript:

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

// ...

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

// ...
```

Abaixo, analisamos as tarefas para mostrar as mensagens recebidas:

1. the section called “Criação de um componente de mensagem”
2. the section called “Reconhecimento das mensagens enviadas pelo usuário atual”
3. the section called “Renderização de uma lista de mensagens de chat”

Criação de um componente de mensagem

O componente `Message` é responsável por renderizar o conteúdo de uma mensagem recebida pela sua sala de chat. Nesta seção, você cria um componente de mensagens para renderizar mensagens de chat individuais na `App`.

Crie um novo arquivo no diretório `src` e atribua a ele o nome `Message`. Passe o tipo `ChatMessage` para esse componente e, em seguida, passe a string `content` das propriedades de `ChatMessage` para exibir o texto da mensagem recebida dos receptores de mensagens da sala de chat. No Project Navigator, acesse `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,
  },
});
```

Dica: use este componente para armazenar propriedades diferentes que você deseja renderizar em suas linhas de mensagens; por exemplo, URLs de avatar, nomes de usuário e carimbos de data e hora de quando a mensagem foi enviada.

Reconhecimento das mensagens enviadas pelo usuário atual

Para reconhecer a mensagem enviada pelo usuário atual, modificamos o código e criamos um contexto do React para armazenar o `userId` do usuário atual.

Crie um novo arquivo no diretório `src` e atribua a ele o nome `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;
```

Observação: aqui usamos o hook `useState` para armazenar o valor `userId`. No futuro, será possível usar `setUserId` para alterar o contexto do usuário ou para fins de login.

Em seguida, substitua `userId` no primeiro parâmetro passado para `tokenProvider` usando o contexto criado anteriormente. Adicione o recurso `SEND_MESSAGE` ao seu provedor de token, conforme especificado abaixo; é necessário enviar mensagens:

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']),
      }),
  );

  // ...
}
```

Em seu componente `Message`, use o `UserContext` criado antes, declare a variável `isMine`, corresponda o `userId` do remetente com o `userId` do contexto e aplique estilos diferentes de mensagens para o usuário atual.

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',
  },
});
```

Renderização de uma lista de mensagens de chat

Agora, liste as mensagens usando um 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>
);

// ...
```

Todas as peças do quebra-cabeça estão prontas para que sua App comece a renderizar as mensagens recebidas pela sua sala de chat. Continue abaixo para ver como realizar ações em uma sala de chat que aproveitem os componentes que você criou.

Executar ações em uma sala de chat

Enviar mensagens e realizar ações de moderador são algumas das principais formas de interagir com uma sala de chat. Aqui, você aprenderá como usar vários objetos de solicitação de chat para realizar ações comuns no Chatterbox, como enviar uma mensagem, excluir uma mensagem e desconectar outros usuários.

Todas as ações em uma sala de chat seguem um padrão comum: para cada ação executada em uma sala de chat, há um objeto de solicitação correspondente. Para cada solicitação, há um objeto de resposta correspondente que você recebe na confirmação da solicitação.

Uma vez que seus usuários recebam as permissões corretas quando você criar um token de chat, eles poderão realizar com êxito as ações correspondentes usando os objetos de solicitação para ver quais solicitações são possíveis de serem realizadas em uma sala de chat.

Abaixo, explicamos como enviar uma mensagem e excluir uma mensagem.

Enviar uma mensagem

A classe `SendMessageRequest` permite o envio de mensagens em uma sala de chat. Aqui, você modifica sua `App` para enviar uma solicitação de mensagem usando o componente que criou em Criar uma entrada de mensagem (na parte 1 deste tutorial).

Para começar, defina uma nova propriedade booleana chamada de `isSending` com o hook `useState`. Use essa nova propriedade para alternar o estado desabilitado do seu elemento `button` usando a constante `isSendDisabled`. No manipulador de eventos do seu `SendButton`, limpe o valor de `messageToSend` e defina `isSending` como verdadeiro.

Como você fará uma chamada de API a partir desse botão, adicionar o booleano *`isSending`* ajuda a evitar que várias chamadas de API ocorram ao mesmo tempo, desativando as interações do usuário no seu *`SendButton`* até que a solicitação seja concluída.

Observação: o envio de mensagens só funcionará se você adicionar o recurso `SEND_MESSAGE` ao seu provedor de token, conforme descrito acima em Reconhecer mensagens enviadas pelo usuário atual.

TypeScript/JavaScript:

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

```
// ...

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

// ...

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

// ...

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

// ...
```

Prepare a solicitação criando uma nova instância `SendMessageRequest`, passando o conteúdo da mensagem para o construtor. Depois de definir os estados `isSending` e `messageToSend`, chame o método `sendMessage`, que envia a solicitação para a sala de chat. Por fim, limpe o sinalizador `isSending` ao receber a confirmação ou rejeição da solicitação.

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);
```

```
  }
};

// ...
```

Experimente o Chatterbox: tente enviar uma mensagem redigindo uma com a sua `MessageBar` e tocando no seu `SendButton`. Você deve ver sua mensagem enviada renderizada dentro da `MessageList` que você criou anteriormente.

Excluir mensagem

Para excluir uma mensagem de uma sala de chat, você precisa ter a capacidade adequada. As capacidades são concedidas durante a inicialização do token de chat que você usa ao se autenticar em uma sala de chat. Para os propósitos desta seção, a `ServerApp` de Configure um servidor local de autenticação/autorização (na parte 1 deste tutorial) permite que você especifique as capacidades de moderador. Isso é feito em sua aplicação usando o objeto `tokenProvider` que você criou em Crie um provedor de tokens (também na parte 1).

Aqui você modifica sua `Message` adicionando uma função para excluir a mensagem.

Primeiro, abra `App.tsx` e adicione a capacidade `DELETE_MESSAGE`. (`capabilities` é o segundo parâmetro da sua função `tokenProvider`.)

Observação: é assim que sua `ServerApp` informa às APIs do IVS Chat que o usuário associado ao token de chat resultante pode excluir mensagens em uma sala de chat. Em uma situação real, você provavelmente terá uma lógica de backend mais complexa para gerenciar os recursos do usuário na infraestrutura da sua aplicação de servidor.

TypeScript/JavaScript:

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

// ...

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

```
// ...
```

Nas próximas etapas, você atualizará sua `Message` para exibir um botão de exclusão.

Defina uma nova função chamada on`Delete` que aceite uma string como um de seus parâmetros e retorne `Promise`. Para o parâmetro de string, passe o ID da mensagem do 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',
  },
});
```

Em seguida, atualize sua `renderItem` para refletir as alterações mais recentes em seu componente `FlatList`.

Em App, defina uma função chamada `handleDeleteMessage` e passe-a para a propriedade `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]);

// ...
```

Prepare uma solicitação criando uma nova instância de `DeleteMessageRequest`, passando o ID da mensagem relevante para o parâmetro do construtor e chame `deleteMessage`, que aceita a solicitação preparada acima:

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);
};

// ...
```

Em seguida, atualize o estado de `messages` para refletir uma nova lista de mensagens que omita a mensagem que você acabou de excluir.

No hook `useEffect`, receba o evento `messageDelete` e atualize sua matriz de estados de `messages` excluindo a mensagem com um ID correspondente ao parâmetro `message`.

Observação: o evento `messageDelete` pode ser gerado para que as mensagens sejam excluídas pelo usuário atual ou por qualquer outro usuário na sala. Manipulá-lo no manipulador de eventos (em vez de junto à solicitação `deleteMessage`) permite unificar o tratamento da exclusão de mensagens.

TypeScript/JavaScript:

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

// ...

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

return () => {
  // ...
```

```
  unsubscribeOnMessageDeleted();
};

// ...
```

Agora é possível excluir usuários de uma sala de chat na sua aplicação de chat.

Próximas etapas

A título de experimento, tente implementar outras ações em uma sala, como desconectar um outro usuário.

SDK de Mensagens para Clientes do Chat do IVS: Práticas recomendadas do React e do React Native

Este documento descreve as práticas mais importantes de uso do SDK de Mensagens do Chat do Amazon IVS para React e React Native. Essas informações devem permitir que você crie uma funcionalidade típica de chat dentro de uma aplicação React e forneça a base de que você precisa para se aprofundar nas partes mais avançadas do SDK de Mensagens do Chat do IVS.

Criar um gancho do inicializador do ChatRoom

A classe `ChatRoom` contém os principais métodos de chat e receptores para gerenciar o estado da conexão e receber eventos, como mensagem recebida e mensagem excluída. Aqui, mostramos como armazenar adequadamente as instâncias de chat em um gancho.

Implementação

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 };
};
```

Observação: não usamos o método `dispatch` do gancho `setState` porque não é possível atualizar parâmetros de configuração em tempo real. O SDK cria uma instância uma vez e não é possível atualizar o provedor do token.

Importante: use o gancho do inicializador do `ChatRoom` uma vez para inicializar uma nova instância de sala de chat.

Exemplo

TypeScript/JavaScript:

```
// ...

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

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

  // ...
};
```

```
// ...
```

Receptor para o estado da conexão

Opcionalmente, você pode se inscrever para receber atualizações do estado da conexão no gancho da sua sala de chat.

Implementação

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 };
};
```

Provedor de instâncias do ChatRoom

Para usar o gancho em outros componentes (para evitar prop drilling), você pode criar um provedor de sala de chat usando o `context` React.

Implementação

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;
```

Exemplo

Depois de criar o `ChatRoomProvider`, você pode consumir sua instância com `useChatRoomContext`.

Importante: só coloque o provedor no nível raiz se precisar acessar o `context` entre a tela de chat e os outros componentes intermediários para evitar novas renderizações desnecessárias se você estiver recebendo conexões. Caso contrário, coloque o provedor o mais próximo possível da tela de 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();
  };
  // ...
};

// ...
```

Criar um receptor de mensagem

Para se manter atualizado com todas as mensagens recebidas, você deve se inscrever em eventos de `message` e `deleteMessage`. Veja alguns códigos que fornecem mensagens de chat para seus componentes.

Importante: para fins de desempenho, separamos `ChatMessageContext` de `ChatRoomProvider`, pois podemos receber muitas novas renderizações quando o receptor do chat atualiza o estado da mensagem. Lembre-se de aplicar `ChatMessageContext` nos componentes em que você usará `ChatMessageProvider`.

Implementação

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>;
};
```

Exemplo no React

Importante: lembre-se de encapsular seu contêiner de mensagens com o `ChatMessagesProvider`. A linha `Message` é um exemplo de componente que exibe o conteúdo de uma mensagem.

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>
```

```
  );
};
```

Exemplo no React Native

Por padrão, `ChatMessage` contém `id`, que é usado automaticamente como chaves do React em `FlatList` para cada linha; portanto, não é preciso passar `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} />;
};
```

Várias instâncias de sala de chat em uma aplicação

Se você usa várias salas de chat simultâneas na sua aplicação, propomos a criação de cada provedor para cada chat e consumi-lo no provedor de chat. Neste exemplo, estamos criando um chat de bot de ajuda e de suporte ao cliente. Criamos um provedor para ambos.

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>;
};
```

Exemplo no React

Agora, você pode usar diferentes provedores de chat que usam o mesmo `ChatRoomProvider`. Posteriormente, você poderá reutilizar o mesmo `useChatRoomContext` dentro de cada tela/visualização.

TypeScript/JavaScript:

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

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

Exemplo no 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 />
    </>
  );
```

```
};
```

Segurança do Chat do Amazon IVS

A segurança na nuvem da AWS é nossa maior prioridade. Como cliente da AWS, você se beneficiará de um data center e de uma arquitetura de rede criados para atender aos requisitos das empresas com as maiores exigências de segurança.

A segurança é uma responsabilidade compartilhada entre a AWS e você. O modelo de responsabilidade compartilhada descreve isto como segurança da nuvem e segurança na nuvem:

- Segurança da nuvem: a AWS é responsável pela proteção da infraestrutura que executa produtos da AWS na Nuvem AWS. A AWS também fornece serviços que podem ser usados com segurança. Auditores de terceiros testam e verificam regularmente a eficácia da nossa segurança como parte dos programas de conformidade da AWS.
- Segurança na nuvem: sua responsabilidade é determinada pelo serviço da AWS que você usa. Você também é responsável por outros fatores, como a confidencialidade de seus dados, os requisitos da sua organização, leis e regulamentos aplicáveis.

Esta documentação ajuda você a entender como aplicar o modelo de responsabilidade compartilhada ao usar o Chat do Amazon IVS. Os tópicos a seguir mostram como configurar o Chat do Amazon IVS para atender aos seus objetivos de segurança e conformidade.

Tópicos

Proteção de dados do Chat to IVS

Para dados enviados para o Chat do Amazon Interactive Video Service (IVS), as seguintes proteções de dados estão em vigor:

- O tráfego do Amazon IVS Chat usa o WSS para manter os dados protegidos em trânsito.
- Os tokens do Amazon IVS Chat são criptografados usando chaves do KMS gerenciadas pelo cliente.

O Chat do Amazon IVS não exige que você forneça todos os dados do cliente (usuário final). Não há campos em salas de chat, entradas nem em grupos de segurança de entrada nos quais haja uma expectativa de que você vai fornecer dados do cliente.

Não coloque informações confidenciais de identificação, como números de conta do cliente em campos de texto livre, como o campo Name (Nome). Isso vale para situações em que você trabalha com a API ou console do Amazon IVS, a AWS CLI ou AWS SDKs. Todos os dados inseridos no Chat do Amazon IVS podem ser incluídos em logs de diagnóstico.

Os streams não são criptografados de ponta a ponta; um stream pode ser transmitido sem criptografia internamente na rede IVS para processamento.

Gerenciamento de Identidade e Acesso no Chat do IVS

O AWS Identity and Access Management (IAM) é um serviço da AWS que ajuda o administrador de uma conta a controlar de forma segura o acesso aos recursos da AWS. Consulte Identity and Access Management in IVS no Guia do usuário do streaming de baixa latência do IVS.

Público

O uso do IAM varia, dependendo do trabalho realizado no Amazon IVS. Consulte Público no Guia do usuário do streaming de baixa latência do IVS.

Como a Amazon IVS funciona com o IAM

Ants de fazer solicitações de API do Amazon IVS, você deve criar uma ou mais identidades do IAM (usuários, grupos e funções) e políticas do IAM, depois anexar as políticas às identidades. Leva alguns minutos para que as permissões sejam propagadas; até então, as solicitações de API são rejeitadas.

Para uma visualização de alto nível de como o Amazon IVS funciona com o IAM, consulte Serviços da AWS que funcionam com o IAM no Guia do usuário do IAM.

Identidades

Você pode criar identidades do IAM para fornecer autenticação a pessoas e processos na sua conta da AWS. Os grupos do IAM são conjuntos de usuários do IAM que podem ser gerenciados como uma unidade. Consulte Identidades (usuários, grupos e funções) no Guia do usuário do IAM.

Políticas

As políticas são documentos de política de permissão de JSON compostos de elementos. Consulte Políticas no Guia do usuário do streaming de baixa latência do IVS.

O Chat do Amazon IVS oferece suporte a três elementos:

- Ações: as ações de política para o Chat do Amazon IVS usam o prefixo `ivschat` antes da ação. Por exemplo, para conceder permissão a alguém para criar uma sala de Chat do Amazon IVS com o método da API `CreateRoom` do Chat do Amazon IVS, inclua a ação `ivschat:CreateRoom` na política dessa pessoa. As instruções de política devem incluir um elemento `Action` ou `NotAction`.
- Recursos: o recurso de sala do Chat do Amazon IVS tem o seguinte formato de ARN:

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

  Por exemplo, para especificar a sala `VgNkEJg0VX9N` em sua instrução, use este ARN:

  ```
  "Resource": "arn:aws:ivschat:us-west-2:123456789012:room/VgNkEJg0VX9N"
  ```

  Algumas ações do Chat do Amazon IVS, como as de criação de recursos, não podem ser executadas em um recurso específico. Nesses casos, você deve usar o caractere curinga (*):

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

- Condições: o Chat do Amazon IVS oferece suporte a algumas chaves de condição globais: `aws:RequestTag`, `aws:TagKeys` e `aws:ResourceTag`.

Você pode usar variáveis como espaços reservados em uma política. Por exemplo, será possível conceder a um usuário do IAM permissão para acessar um recurso somente se ele estiver marcado com o nome de usuário do IAM. Consulte Variáveis e tags no Guia do usuário do IAM.

O Amazon IVS fornece políticas gerenciadas pela AWS que podem ser usadas para conceder um conjunto predefinido de permissões para identidades (somente leitura ou acesso total). Você pode optar por usar políticas gerenciadas em vez das políticas baseadas em identidade mostradas abaixo. Para obter detalhes, consulte Managed Policies for Amazon IVS Chat.

Autorização baseada em tags do Amazon IVS

É possível anexar tags a recursos do Chat do Amazon IVS ou informar tags em uma solicitação para o Chat do Amazon IVS. Para controlar o acesso baseado em tags, forneça informações sobre as tags no elemento de condição de uma política usando as chaves de condição `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` ou `aws:TagKeys`. Para obter mais informações sobre como marcar recursos do Chat do Amazon IVS, consulte “Marcação” na Referência da API do Chat do IVS.

Perfis

Consulte Funções do IAM eCredenciais de segurança temporárias no Guia do usuário do IAM.

Perfil do IAM é uma entidade dentro da sua conta da AWS que tem permissões específicas.

O Amazon IVS oferece suporte ao uso de credenciais de segurança temporárias. É possível usar credenciais temporárias para fazer login com federação, assumir uma função do IAM ou assumir uma função entre contas. Obtenha credenciais de segurança temporárias chamando operações de API do AWS Security Token Service, como `AssumeRole` ou `GetFederationToken`.

Acesso privilegiado e sem privilégios

Os recursos da API têm acesso privilegiado. O acesso de reprodução sem privilégios pode ser configurado por meio de canais privados; consulte Setting Up IVS Private Channels.

Práticas recomendadas para políticas

Consulte Práticas recomendadas do IAM no Guia do usuário do IAM.

As políticas baseadas em identidade são muito eficientes. Elas determinam se alguém pode criar, acessar ou excluir recursos do Amazon IVS em sua conta. Essas ações podem incorrer em custos para sua conta da AWS. Siga estas recomendações:

- Conceder privilégios mínimos: ao criar políticas personalizadas, conceda apenas as permissões necessárias para executar uma tarefa. Comece com um conjunto mínimo de permissões e conceda permissões adicionais, conforme necessário. Fazer isso é mais seguro do que começar com permissões que são muito lenientes, e tentar restringi-las posteriormente. Especificamente, reservar `ivschat:*` para acesso de administrador; não usá-lo em aplicações.
- Habilitar autenticação multifator (MFA) para operações confidenciais: para aumentar a segurança, exija que os usuários do IAM usem MFA para acessar recursos ou operações de API confidenciais.
- Usar condições de política para segurança adicional: na medida do possível, defina as condições sob as quais suas políticas baseadas em identidade permitem o acesso a um recurso. Por exemplo, você pode gravar condições para especificar um intervalo de endereços IP permitidos do qual a solicitação deve partir. Você também pode escrever condições para permitir somente solicitações em uma data especificada ou período ou para exigir o uso de SSL ou MFA.

Exemplos de políticas baseadas em identidade

Use o console do Amazon IVS.

Para acessar o console do Amazon IVS, é necessário ter um conjunto mínimo de permissões que permitam listar e visualizar detalhes sobre os recursos do Chat do Amazon IVS em sua conta da AWS. Se você criar uma política de permissões baseada em identidade que seja mais restritiva que as permissões mínimas necessárias, o console não vai funcionar como pretendido para entidades com essa política. Para garantir o acesso ao console do Amazon IVS, anexe a seguinte política às identidades (consulte Adicionar e remover permissões do IAM no Guia do usuário do IAM).

As partes da política a seguir fornecem acesso a:

- Todas as operações da API do Chat do Amazon IVS
- Suas cotas de serviço do Chat do Amazon IVS
- Listar lambdas e adicionar permissões para o lambda escolhido para moderação do Amazon IVS Chat
- Amazon CloudWatch para obter métricas para sua sessão de 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": "*"
    }
  ]
}
```

Política baseada em recurso para o Amazon IVS Chat

É necessário dar ao serviço Amazon IVS Chat a permissão para invocar seu recurso do Lambda para revisar mensagens. Para isso, siga as instruções em Usar políticas baseadas em recursos para o AWS Lambda (no Guia do desenvolvedor do AWS Lambda) e preencha os campos, conforme especificado abaixo.

Para controlar o acesso a seu recursos do Lambda, use condições baseadas em:

- `SourceArn`: nossa política de exemplo usa um curinga (*) para permitir que todas as salas de sua conta invoquem o lambda. Opcionalmente, você pode especificar uma sala da conta para permitir que apenas essa sala invoque o lambda.

- `SourceAccount`: na política de exemplo abaixo, o ID da conta da AWS é `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/*"
            }
         }
      }
   ]
}
```

Solução de problemas

Consulte Solução de problemas no Guia do usuário do streaming de baixa latência do IVS para obter informações sobre como diagnosticar e corrigir problemas comuns que poderiam ocorrer ao trabalhar com o Chat do Amazon IVS e o IAM.

Politicas gerenciadas para o Chat do IVS

Uma política gerenciada pela AWS é uma política independente que é criada e administrada por ela. Consulte Políticas gerenciadas para Amazon IVS no Guia do usuário do streaming de baixa latência do IVS.

Uso de funções vinculadas ao serviço para o Chat do IVS

O Amazon IVS usa perfis vinculados ao serviço do AWS IAM. Consulte Usar perfis vinculados ao serviço para o Amazon IVS no Guia do usuário do streaming de baixa latência do IVS.

Registro em log e monitoramento do Chat do IVS

Para registrar em log a performance e/ou as operações, use o Amazon CloudTrail. Consulte Registro em log de chamadas de API do Amazon IVS com o AWS AWS CloudTrail no Guia do usuário do streaming de baixa latência do IVS.

Resposta a incidentes no Chat do IVS

Para detectar ou alertar sobre incidentes, você pode monitorar a integridade do seu stream por meio de eventos do Amazon EventBridge. Consulte Uso do Amazon EventBridge com o Amazon IVS para o streaming de baixa latência e para o streaming em tempo real.

Use o AWS Health Dashboard para obter informações sobre a integridade geral do Amazon IVS (por região).

Resiliência do chat do IVS

As APIs do IVS usam a infraestrutura global da AWS e são criadas em torno das regiões e zonas de disponibilidade da AWS. Consulte IVS Resilience no Guia do usuário do streaming de baixa latência do IVS.

Segurança da infraestrutura do Chat do IVS

Como um serviço gerenciado, o Amazon IVS é protegido pelos procedimentos de segurança da rede global da AWS. Eles estão descritos em Práticas recomendadas de segurança, identidade e conformidade.

Chamadas de API

Você usa chamadas de API publicadas pela AWS para acessar o Amazon IVS por meio da rede. Consulte Chamadas de API em Segurança da infraestrutura no Guia do usuário do streaming de baixa latência do IVS.

Amazon IVS Chat

A ingestão e entrega de mensagens do Amazon IVS Chat ocorrem por meio de conexões WSS criptografadas para nossa borda. A API do Amazon IVS Messaging usa conexões HTTPS criptografadas. Assim como acontece com a transmissão e reprodução de vídeo, é necessário o TLS versão 1.2 ou superior, e os dados de mensagens podem ser transmitidos sem criptografia interna para processamento.

Service Quotas do Chat do IVS

Veja a seguir cotas de serviço e limites para endpoints, recursos e outras operações de chat do Amazon Interactive Video Service (IVS). As cotas de serviço (também chamadas de limites) são o número máximo de recursos ou operações de serviço para sua conta da AWS. Ou seja, esses limites são por conta da AWS, salvo indicação em contrário na tabela. Além disso, consulte AWS Service Quotas.

Para se conectar a um produto da AWS de forma programática, use um endpoint. Além disso, consulte AWS Service Endpoints (Endpoints de produto da AWS).

Todas as cotas são aplicadas por região.

Aumentos de cota de serviço

Para cotas que são ajustáveis, você pode solicitar um aumento de taxa por meio do Console da AWS. Use o console para também visualizar informações sobre cotas de serviço.

As cotas para taxa de chamadas da API não são ajustáveis.

Cotas de taxa de chamada de API

Tipo de operação	Operação	Padrão
Sistema de mensagens	`DeleteMessage`	100 TPS
Sistema de mensagens	`DisconnectUser`	100 TPS
Sistema de mensagens	`SendEvent`	100 TPS
Token de chat	`CreateChatToken`	200 TPS
Configuração de log	`CreateLoggingConfiguration`	3 TPS
Configuração de log	`DeleteLoggingConfiguration`	3 TPS
Configuração de log	`GetLoggingConfiguration`	3 TPS
Configuração de log	`ListLoggingConfigurations`	3 TPS

Tipo de operação	Operação	Padrão
Configuração de log	`UpdateLoggingConfiguration`	3 TPS
Sala	`CreateRoom`	5 TPS
Sala	`DeleteRoom`	5 TPS
Sala	`GetRoom`	5 TPS
Sala	`ListRooms`	5 TPS
Sala	`UpdateRoom`	5 TPS
Tags	`ListTagsForResource`	10 TPS
Tags	`TagResource`	10 TPS
Tags	`UntagResource`	10 TPS

Outras cotas

Recurso ou atributo	Padrão	Ajustável	Descrição
Conexões de chat simultâneas	50.000	Sim	Máximo de conexões de chat simultâneas por conta, em todas as suas salas em uma Região da AWS.
Configurações de log	10	Sim	O número máximo de configurações de log que podem ser criadas por conta na Região da AWS atual.
Período de tempo limite para o manipulador de revisão	200	Não	Período de tempo limite em milissegundos para todos os manipuladores de revisão de mensagens na Região

Recurso ou atributo	Padrão	Ajustável	Descrição
			da AWS atual. Se isso for excedido, a mensagem será permitida ou negada, conforme o valor do campo `fallbackResult` que você configurou para o manipulador de revisão de mensagens.
Taxa de solicitações DeleteMessage em todas as suas salas	100	Sim	Máximo de solicitações DeleteMessage que podem ser feitas por segundo em todas suas salas. As solicitações podem vir da API do Amazon IVS Chat ou da API do Amazon IVS Chat Messaging (WebSocket).
Taxa de solicitações DisconnectUser em todas as suas salas	100	Sim	Máximo de solicitações DisconnectUser que podem ser feitas por segundo em todas suas salas. As solicitações podem vir da API do Amazon IVS Chat ou da API do Amazon IVS Chat Messaging (WebSocket).
Taxa de solicitações de mensagens por conexão	10	Não	Máximo de solicitações de mensagens por segundo que uma conexão de chat pode fazer.

Recurso ou atributo	Padrão	Ajustável	Descrição
Taxa de solicitações de SendMessage em todas as suas salas	1000	Sim	Máximo de solicitações SendMessage que podem ser feitas por segundo em todas as suas salas. Essas solicitações vêm da API do Amazon IVS Chat Messaging (WebSocket).
Taxa de solicitações SendMessage por sala	100	Não (mas é possível configurar pela API)	Máximo de solicitações SendMessage que podem ser feitas por segundo para qualquer uma de suas salas. É possível configurar isso com o campo `maximumMessageRatePerSecond` de CreateRoom e UpdateRoom. Essas solicitações vêm da API do Amazon IVS Chat Messaging (WebSocket).
Salas	50.000	Sim	Máximo de salas de chat por conta e por Região da AWS.

Integração do Service Quotas a métricas de uso do CloudWatch

Você pode usar o CloudWatch para gerenciar proativamente suas cotas de serviço por meio das métricas de uso do CloudWatch. É possível usar essas métricas para visualizar o uso do serviço atual nos gráficos e painéis do CloudWatch. As métricas de uso do Chat do Amazon IVS correspondem às cotas de serviço do Chat do Amazon IVS.

É possível usar uma função matemática métrica do CloudWatch para exibir as cotas de serviço desses recursos nos gráficos. Também é possível configurar alarmes que alertam você quando o uso se aproxima de uma cota de serviço.

Para acessar as métricas de uso:

1. Abra o console do Service Quotas em https://console.aws.amazon.com/servicequotas/
2. No painel de navegação, escolha AWS services (produtos da AWS).
3. Na lista de serviços da AWS, procure e selecione Chat do Amazon Interactive Video Service.
4. Na lista Service quotas (Cotas de serviço), selecione a cota de serviço de seu interesse. Uma nova página é aberta com informações sobre a cota/métrica de serviço.

Como alternativa, você pode acessar essas métricas por meio do console do CloudWatch. Em Namespaces da AWS, escolha Use (Uso). Em seguida, na lista Serviço, escolha IVS. (Consulte Monitoramento do Chat do Amazon IVS.)

No namespace AWS/Uso, o Chat do Amazon IVS fornece a seguinte métrica:

Nome da métrica	Descrição
`ResourceCount`	A contagem de recursos especificados em execução em sua conta. Os recursos são definidos pelas dimensões associadas à métrica. Estatística válida: máximo (número máximo de recursos usados durante o período de um minuto).

As dimensões a seguir são usadas para refinar a métrica de uso.

Dimensão	Descrição
Serviço	O nome do produto da AWS que contém o recurso. Valor válido: `IVS Chat`.
Classe	A classe do recurso sob acompanhamento. Valor válido: `None`.
Tipo	O tipo de recurso que está sendo acompanhado. Valor válido: `Resource`.
Recurso	O nome do recurso da AWS. Valor válido: `ConcurrentChatConnections` .

Dimensão	Descrição
	A métrica de utilização do ConcurrentChatConnections é uma cópia da métrica no namespace AWS/IVSChat (com a dimensão Nenhuma), conforme descrito em Monitoramento do Chat do Amazon IVS.

Criando um alarme do CloudWatch para Métricas de uso

Para criar um alarme do CloudWatch com base em uma métrica de uso do Chat do Amazon IVS:

1. No console do Service Quotas, selecione a cota de serviço de seu interesse, conforme descrito acima. No momento, só é possível criar alarmes para ConcurrentChatConnections.
2. Na seção Amazon CloudWatch alarms (Alarmes do Amazon CloudWatch), escolha Create alarm (Criar alarme).
3. Em Alarm threshold (Limite do alarme), escolha a porcentagem do valor da cota aplicada que você deseja definir como o valor do alarme.
4. Em Name of alarm (Nome do alarme), digite um nome para o alarme.
5. Escolha Create (Criar).

Solução de problemas do Chat do IVS

Este documento descreve as melhores práticas e dicas de solução de problemas do Chat do Amazon Interactive Video Service (IVS). Os comportamentos relacionados ao chat do IVS muitas vezes são diferentes dos comportamentos relacionados ao vídeo do IVS. Para obter mais informações, consulte Conceitos básicos do Chat do Amazon IVS.

Tópicos:

Por que as conexões do chat do IVS não foram desconectadas quando a sala foi excluída?

Quando um recurso de sala de chat for excluído, se a sala estiver sendo usada ativamente, os clientes de chat conectados à sala não serão automaticamente desconectados. A conexão será interrompida se/quando a aplicação de chat atualizar o token de chat. Como alternativa, deve ser feita uma desconexão manual de todos os usuários para remover todos os usuários da sala de chat.

Glossário do IVS

Consulte também o Glossary da AWS. Na tabela abaixo, LL significa streaming de baixa latência do IVS; RT, streaming em tempo real do IVS.

Prazo	Descrição	LL	RT	Chat
AAC	Codificação de áudio avançado. O AAC é um padrão de codificação de áudio para compressão de áudio digital com perdas. Projetado para ser o sucessor do formato MP3, o AAC geralmente alcança uma qualidade de som superior ao MP3 com a mesma taxa de bits. O AAC foi padronizado pela ISO e pela IEC como parte das especificações MPEG-2 e MPEG-4.	✓	✓	
Streaming com taxa de bits adaptável	O streaming com taxa de bits adaptável (ABR) permite que o reprodutor do IVS mude para uma taxa de bits mais baixa quando a qualidade da conexão piora e volte para uma taxa de bits mais alta quando a qualidade da conexão melhora.	✓		
Streaming adaptável	Consulte Codificação em camadas com a transmissão simultânea.		✓	
Usuário administrativo	Um usuário da AWS com acesso administrativo aos recursos e serviços disponíveis em uma conta da AWS. Consulte Terminologia no Guia do usuário da configuração da AWS.	✓	✓	✓
ARN	Nome do recurso da Amazon, um identificador exclusivo de um recurso da AWS. Os formatos específicos do ARN dependem do tipo do recurso. Para conhecer os formatos de ARN usados pelos recursos do IVS, consulte a Referência de autorização do serviço.	✓	✓	✓

Prazo	Descrição	LL	RT	Chat
Taxa de proporção	Descreve a proporção entre a largura e a altura do quadro. Por exemplo, 16:9 é a proporção que corresponde à resolução Full HD ou 1080p.	✓	✓	
Modo de áudio	Uma configuração de áudio predefinida ou personalizada otimizada para diferentes tipos de usuários de dispositivos móveis e o equipamento que eles usam. Consulte SDK de Transmissão do IVS: modos de áudio móvel (streaming em tempo real).		✓	
AVC, H.264, MPEG-4 Parte 10	Codificação de vídeo avançado, também conhecida como H.264 ou MPEG-4 Parte 10, um padrão de compressão para vídeo digital com perdas.	✓	✓	
Substituição de plano de fundo	Um tipo de filtro de câmera que permite que criadores de fluxo ao vivo alterem seus planos de plano de fundo. Consulte Substituição de plano de fundo em SDK de Transmissão do IVS: filtros de câmera de terceiros (streaming em tempo real).		✓	
Taxa de bits	Uma métrica de streaming para o número de bits transmitidos ou recebidos por segundo.	✓	✓	
Transmissão, transmissores	São outros termos para fluxo, streamer.	✓		

Prazo	Descrição	LL	RT	Chat
Armazenamento em buffer	Uma condição que ocorre quando o dispositivo de reprodução não consegue baixar o conteúdo antes que ele deva ser reproduzido. O armazenamento em buffer pode se manifestar de várias maneiras: o conteúdo pode parar e começar aleatoria mente (também conhecido como engasgo), o conteúdo pode parar por longos períodos (também conhecido como congelamento) ou o reprodutor do IVS pode pausar a reprodução.	✓	✓	
Lista de reproduçã o de intervalo de bytes	Uma lista de reprodução mais granular do que a lista de reprodução HLS padrão. A lista de reprodução HLS padrão é composta de arquivos de mídia de dez segundos. Com uma lista de reprodução de intervalo de bytes, a duração do segmento é a mesma do intervalo de quadros-c have configurado para o fluxo. A lista de reprodução com intervalo de bytes está disponível somente para as transmissões que foram gravadas automaticamente em um bucket do S3. Ela é criada além da lista de reprodução HLS. Consulte Listas de reprodução de intervalo de bytes em Gravação automática no Amazon S3 (streaming de baixa latência).	✓		

Prazo	Descrição	LL	RT	Chat
CBR	Taxa de bits constante, um método de controle de taxa para codificadores que mantém uma taxa de bits consistente durante toda a reprodução de um vídeo, independentemente do que esteja acontecendo durante a transmissão. As pausas na ação podem ser preenchidas para atingir a taxa de bits desejada e os picos podem ser quantizados pelo ajuste da qualidade da codificação para corresponder à taxa de bits desejada. É altamente recomendável usar CBR em vez de VBR.	✓	✓	
CDN	Rede de entrega de conteúdo ou Rede de distribuição de conteúdo, uma solução distribuída geograficamente que otimiza a entrega de conteúdo, como streaming de vídeo, ao aproximá-lo de onde os usuários estão localizados.	✓		
Canal	Um recurso do IVS que armazena a configuração para streaming, incluindo um servidor de ingestão, uma chave de fluxo, um URL de reprodução e opções de gravação. Os streamers usam a chave de stream associada a um canal para iniciar uma transmissão. Todas as métricas e eventos gerados durante uma transmissão são associados a um recurso de canal.	✓		
Tipo de canal	Determina a resolução e a taxa de quadros permitidas para o canal. Consulte Channel Types em IVS Low-Latency Streaming API Reference.	✓		
Registro em log de chat	Uma opção avançada que pode ser habilitada pela associação de uma configuração de registro em log com uma sala de chat.			✓

Prazo	Descrição	LL	RT	Chat
Sala de chat	Um recurso do IVS que armazena a configuração de uma sessão de conversa, incluindo recursos opcionais, como Processador de análise de mensagens e Registro em log de chat. Consulte Step 2: Create a Chat Room em Getting Started with IVS Chat.			✓
Composição do cliente	Usa um dispositivo host para mixar fluxos de áudio e vídeo dos participantes do palco e depois os envia como um fluxo composto para um canal do IVS. Isso permite mais controle sobre o aspecto da composição à custa de uma maior utilização dos recursos do cliente e de um risco maior de que um problema em um palco ou em um host afete os espectadores. Consulte também composição do servidor.	✓	✓	
CloudFront	Um serviço de CDN fornecido pela Amazon.	✓		
CloudTrail	Um serviço da AWS para coletar, monitorar, analisar e reter eventos e atividades da conta da AWS e de fontes externas. Consulte Registro de chamadas de API do IVS com o AWS CloudTrail.	✓	✓	✓
CloudWatch	Um serviço da AWS para monitorar aplicações, responder a alterações de mudanças de performance, otimizar o uso de recursos e fornecer insights sobre integridade operacional. Você pode usar o CloudWatch para monitorar métricas do IVS; consulte Monitoramento do streaming em tempo real do IVS e Monitoramento do streaming de baixa latência do IVS.	✓	✓	✓
Composição	O processo de combinar fluxos de áudio e vídeo de várias fontes em um único fluxo.	✓	✓	

Prazo	Descrição	LL	RT	Chat
Pipeline de composição	Uma sequência de etapas de processamento necessárias para combinar vários fluxos e codificar o fluxo resultante.	✓	✓	
Compactação	Codificação de informações usando menos bits do que a representação original. Qualquer compactação específica pode ser sem perdas ou com perdas. A compactação sem perdas reduz os bits ao identificar e eliminar a redundância estatística. Nenhuma informação é perdida na compactação sem perdas. A compactação com perdas reduz os bits ao remover informações desnecessárias ou menos importantes.	✓	✓	
Ambiente de gerenciamento	Armazena informações sobre os recursos do IVS, como canais, palcos ou salas de chat, e fornece interfaces para criar e gerenciar esses recursos. É regional (baseado nas regiões da AWS).	✓	✓	✓
CORS	O compartilhamento de recursos de origem cruzada é um recurso da AWS que permite que as aplicações Web clientes carregadas em um domínio interajam com recursos, como buckets do S3, em outro domínio. O acesso pode ser configurado com base em cabeçalhos, métodos HTTP e domínios de origem. Consulte Usar o compartilhamento de recursos de origem cruzada (CORS): Amazon Simple Storage Service no Guia do usuário do Amazon Simple Storage Service.	✓		
Fonte de imagem personalizada	Uma interface fornecida pelo SDK de Transmissão do IVS que permite que uma aplicação forneça sua própria entrada de imagem em vez de ficar limitada a câmeras predefinidas.	✓	✓	

Prazo	Descrição	LL	RT	Chat
Plano de dados	A infraestrutura que transporta os dados da ingestão para a saída. Ele opera com base na configuração gerenciada no ambiente de gerenciamento e não está restrito a uma região da AWS.	✓	✓	✓
Codificador, codificação	O processo de conversão de conteúdo de vídeo e áudio em formato digital, adequado para streaming . A codificação pode ser baseada em hardware ou software.	✓	✓	
Evento	Uma notificação automática publicada pelo IVS para o serviço de monitoramento do AmazonEventBridge. Um evento representa uma alteração no estado ou na integridade de um recurso de streaming, como um palco ou um pipeline de composição. Consulte Uso do Amazon EventBridge com o streaming de baixa latência do IVS e Uso do Amazon EventBridge com o streaming em tempo real do IVS.	✓	✓	✓
FFmpeg	Um projeto de software gratuito e de código aberto que consiste em um conjunto de bibliotecas e programas para o tratamento de arquivos e fluxos de vídeo e áudio. O FFmpeg fornece uma solução entre plataformas para gravar, converter e transmitir áudio e vídeo.	✓		

Prazo	Descrição	LL	RT	Chat
Fluxo fragmentado	Criado quando uma transmissão se desconecta e depois se reconecta no intervalo especificado na configuração de gravação do canal. Os vários fluxos resultantes são considerados uma única transmissão e são mesclados em um único fluxo gravado. Consulte Mesclar streams fragmentados em Gravação automática no Amazon S3 (streamin g de baixa latência).	✓		
Taxa de quadros	Uma métrica de streaming para o número de quadros de vídeo transmitidos ou recebidos por segundo.	✓	✓	
HLS	HTTP Live Streaming (HLS), um protocolo de comunicações de streaming com taxa de bits adaptável baseado em HTTP usado para entregar fluxos do IVS aos espectadores.	✓		
Lista de reprodução HLS	Uma lista dos segmentos de mídia que compõem um fluxo. As listas de reprodução HLS padrão são compostas de arquivos de mídia de dez segundos. O HLS também é compatível com listas de reprodução de intervalo de bytes mais granulare s.	✓		
Host	O participante de um evento em tempo real que envia vídeo ou áudio para o palco.		✓	
IAM	Identity and Access Management, um serviço da AWS que permite que os usuários gerenciem com segurança identidades e acesso aos serviços e recursos da AWS, incluindo o IVS.	✓	✓	✓

Prazo	Descrição	LL	RT	Chat
Ingestão	Processo do IVS para receber fluxos de vídeo de um host ou transmissor para processamento ou entrega para visualizadores ou outros participantes.	✓	✓	
Servidor de ingestão	Recebe fluxos de vídeo e os envia para um sistema de transcodificação, em que os fluxos são submetidos a transmux ou são transcodificados para HLS para entrega aos visualizadores. Os servidores de ingestão são componentes específicos do IVS que recebem fluxos para canais, junto com um protocolo de ingestão (RTMP, RTMPS). Consulte as informações sobre como criar um canal em Conceitos básicos do streaming de baixa latência do IVS.		✓	
Vídeo entrelaçado	Transmite e exibe somente linhas pares ou ímpares de quadros subsequentes para criar uma duplicação percebida da taxa de quadros sem consumir largura de banda adicional. Não recomendamos o uso de vídeo entrelaçado devido a preocupações com a qualidade do vídeo.	✓	✓	
JSON	JavaScript Object Notation é um formato de arquivo de padrão aberto que usa texto legível por humanos para transmitir objetos de dados que constituam pares atributo-valor e tipos de dados de matriz ou outro valor que possa ser serializado.	✓	✓	✓

Prazo	Descrição	LL	RT	Chat
Quadro-chave, quadro delta, intervalo de quadros-chave	O quadro-chave (também conhecido como intracodificado ou i-frame) é um quadro completo da imagem em um vídeo. Os quadros subsequentes, os quadros delta (também chamados de quadros previstos ou p-frame), contêm apenas informações que foram alteradas. Os quadros-chave aparecerão várias vezes em um fluxo, dependendo do intervalo do quadro-chave definido no codificador.	✓	✓	
Lambda	Um serviço da AWS para executar código (denominado funções do Lambda) sem provisionar qualquer infraestrutura de servidor. As funções do Lambda podem ser executadas em resposta a eventos e solicitações de invocação ou com base em um cronograma. Por exemplo, o Chat do IVS usa funções do Lambda para permitir a revisão de mensagens em uma sala de chat.	✓	✓	✓
Latência, latência de vidro para vidro	É um atraso na transferência de dados. O IVS define os intervalos de latência como: • Baixa latência: menos de três segundos • Latência em tempo real: menos de 300 ms A latência de vidro para vidro refere-se ao atraso que ocorre entre o momento em que uma câmera captura um streaming ao vivo e esse streaming chega à tela de um visualizador.	✓	✓	
Codificação em camadas com transmissão simultânea	Permite a codificação e publicação simultâneas de vários fluxos de vídeo com diferentes níveis de qualidade. Consulte Streaming adaptável: codificação em camadas com a transmissão simultânea em Otimizações de streaming em tempo real.		✓	

Prazo	Descrição	LL	RT	Chat
Manipulador de revisão de mensagem	Permite que os clientes do Chat do IVS analisem e filtrem automaticamente as mensagens de chat do usuário antes que elas sejam enviadas para a sala de chat. Ele é habilitado pela associação de uma função do Lambda com uma sala de chat. Consulte Creating a Lambda Function em Chat Message Review Handler.			✓
Mesclador	Um recurso dos SDKs de Transmissão Móvel do IVS que usa várias fontes de áudio e vídeo e gera uma única saída. Ele oferece suporte ao gerenciamento de vídeo na tela e a elementos de áudio que representam fontes, como câmeras, microfones, capturas de tela e áudio e vídeo gerados pela aplicação. A saída pode então ser transmitida para o IVS. Consulte Configuração de uma sessão de transmissão para mixagem no SDK de Transmissão do IVS: Guia de mixagem (streaming de baixa latência).	✓		
Streaming de vários hosts	Combina fluxos de vários hosts em um único fluxo. Isso pode ser feito usando a composição do cliente ou do servidor. O streaming de vários hosts permite diversos cenários, como convidar visualizadores para um palco para perguntas e respostas, competições entre hosts, chat por vídeo e hosts conversando entre si na frente de um grande público.		✓	
Lista de reprodução multivariante	Um índice de todos os fluxos variantes disponíveis para uma transmissão.	✓		

Prazo	Descrição	LL	RT	Chat
OAC	Controle de acesso de origem, um mecanismo para restringir o acesso a um bucket do S3 para que conteúdo, como um fluxo gravado, possa ser disponibilizado somente por meio da CDN do CloudFront.	✓		
OBS	Open Broadcaster Software, software gratuito e de código aberto para gravação de vídeo e streaming ao vivo. O OBS oferece uma alternativa (ao SDK de Transmissão do IVS) para editoração eletrônica. Streamers mais sofisticados familiarizados com o OBS podem preferi-lo por seus recursos avançados de produção, como transições de cena, mixagem de áudio e gráficos de sobreposição.	✓	✓	
Participante	Um usuário em tempo real conectado ao palco como host ou visualizador.		✓	
Token de participante	Autentica o participante de um evento em tempo real quando ele entra em um palco. Um token de participante também controla se um participante pode enviar vídeo para o palco.		✓	
Token de reprodução, par de chaves de reprodução	Um mecanismo de autorização que permite que os clientes restrinjam a reprodução de vídeo em canais privados. Os tokens de reprodução são gerados em um par de chaves de reprodução. Um par de chaves de reprodução é o par de chaves públicas e privadas usado para assinar e validar o token de autorização do visualizador para reprodução. Consulte Criar ou importar uma chave de reprodução do IVS em Configuração de canais privados IVS e consulte as operações do par de chaves de reprodução na IVS Low-Latency Streaming API Reference..	✓		

Prazo	Descrição	LL	RT	Chat
URL de reprodução	Identifica o endereço que um visualizador usa para iniciar a reprodução de um canal específico. E esse endereço pode ser usado em todo o mundo. O IVS automaticamente seleciona a melhor localizaç ão na rede de entrega de conteúdo global do IVS para entregar o vídeo a cada visualizador. Consulte as informações sobre como criar um canal em Conceitos básicos do streaming de baixa latência do IVS.	✓		
Canal privado	Permite que os clientes restrinjam o acesso aos fluxos usando um mecanismo de autorização baseado em tokens de reprodução. Consulte Workflow for IVS Private Channels em Workflow for IVS Private Channels.	✓		
Vídeo progressivo	Transmite e exibe todas as linhas de cada quadro em sequência. Recomendamos o uso de vídeo progressivo em todas as etapas de uma transmiss ão.	✓	✓	
Cotas	Número máximo de recursos ou operações de serviço do IVS para sua conta da AWS. Ou seja, esses limites são para cada conta da AWS, salvo indicação em contrário. Todas as cotas são aplicadas por região. Consulte Amazon Interacti ve Video Service endpoints and quotas no Guia de referência geral da AWS.	✓	✓	✓

Prazo	Descrição	LL	RT	Chat
Regiões	Permitem acesso aos serviços da AWS que residam fisicamente em uma área geográfica específica. As regiões fornecem tolerância a falhas, estabilidade e resiliência e também podem reduzir a latência. Com as regiões, você pode criar recursos redundantes que permanecem disponíveis e não afetados por uma interrupção regional. A maioria das solicitações de serviços da AWS está associada a uma região geográfica específica. Os recursos que você cria em uma região não existe em qualquer outra região, a menos que você use explicitamente um recurso de replicação oferecido por um serviço da AWS. Por exemplo, o Amazon S3 oferece suporte à replicação entre regiões. Alguns serviços, como o IAM, não têm recursos entre regiões.	✓	✓	✓
Resolução	Descreve o número de pixels em um único quadro de vídeo, por exemplo, Full HD ou 1080p define um quadro com 1920x1080 pixels.	✓	✓	
Usuário raiz	O proprietário de uma conta da AWS. O usuário raiz tem acesso completo a todos os serviços e recursos da AWS na conta da AWS.	✓	✓	✓
RTMP, RTMPS	O Real-Time Messaging Protocol é um padrão do setor para transmissão de vídeo em uma rede. O RTMPS é a versão segura do RTMP, sendo executado por uma conexão Transport Layer Security (TLS/SSL).	✓	✓	

Prazo	Descrição	LL	RT	Chat
Bucket do S3	Uma coleção de objetos armazenados no Amazon S3. Muitas políticas, incluindo acesso e replicação, são definidas no bucket e se aplicam a todos os objetos no bucket. Por exemplo, uma transmissão do IVS é armazenada como vários objetos em um bucket do S3.	✓		
SDK	Kit de desenvolvimento de software, uma coleção de bibliotecas para desenvolvedores que criam aplicações com o IVS.	✓	✓	✓
Segmentação de selfies	Permite substituir o plano de fundo em uma transmissão ao vivo, usando uma solução específica do cliente que aceita uma imagem da câmera como entrada e retorna uma máscara que fornece uma pontuação de confiança para cada pixel da imagem, indicando se ela está em primeiro ou segundo plano. Consulte Substituição de plano de fundo em SDK de Transmissão do IVS: filtros de câmera de terceiros (streaming em tempo real).		✓	
Versionamento semântico	Um formato de versão na forma de Major.Minor.Patch. Correções de bugs que não afetam a API incrementam a versão do patch, as adições/alterações da API compatíveis com versões anteriores incrementam a versão secundária e as alterações da API incompatíveis com versões anteriores incrementam a versão principal.	✓	✓	✓

Prazo	Descrição	LL	RT	Chat
Composição do servidor	Usa um servidor do IVS para mixar áudio e vídeo dos participantes do palco e, em seguida, enviar esse vídeo mixado para um canal do IVS para alcançar um público maior ou armazená-lo em um bucket do S3. A composição do servidor reduz a carga do cliente, melhora a resiliência da transmissão e permite um uso mais eficiente da largura de banda. Consulte também a composição do cliente.		✓	
Cotas de serviço	Serviço da AWS que ajuda a gerenciar as cotas para muitos serviços da AWS em um único local. Além de pesquisar os valores de cotas, é possível solicitar o aumento delas no console do Cotas de serviço.	✓	✓	✓
Perfil vinculado a serviço	Tipo exclusivo de perfil do IAM que é vinculado diretamente a um serviço da AWS. Os perfis vinculados a serviços são automaticamente criados pelo IVS e incluem todas as permissões que o serviço exige para chamar outros serviços da AWS em seu nome, por exemplo, acessar um bucket do S3. Consulte Uso de funções vinculadas ao serviço para o IVS em Segurança do IVS.	✓		
Estágio	Recurso do IVS que representa um espaço virtual no qual os participantes do evento em tempo real podem trocar vídeos em tempo real. Consulte Criar um palco em Conceitos básicos do streaming em tempo real do IVS.		✓	

Prazo	Descrição	LL	RT	Chat
Sessão de palco	Começa quando o primeiro participante entra em um palco e termina alguns minutos após o último participante parar de publicar no palco. Um palco de longa duração pode ter várias sessões ao longo de sua vida útil.		✓	
Fluxo	Dados que representam conteúdo de vídeo ou áudio que são enviados continuamente de uma origem para um destino.	✓	✓	
Chave de stream	Identificador atribuído pelo IVS quando você cria um canal. Usado para autorizar streaming para o canal. Trate a chave de fluxo como um segredo, pois qualquer pessoa que a tenha poderá fazer streaming para o canal. Consulte Conceitos básicos do streaming de baixa latência do IVS.	✓		
Privação de fluxo	Um atraso ou uma interrupção na entrega de fluxo ao IVS. Isso ocorre quando o IVS não recebe a quantidade esperada de bits que o dispositivo de codificação anunciou que enviaria em um determinado período. Uma ocorrência de privação de fluxo resulta em um evento de privação de fluxo. Do ponto de vista de um visualizador, a privação de fluxo pode ser um vídeo com atrasos, buffers ou congelamentos. A privação de fluxo pode ser breve (menos de cinco segundos) ou longa (vários minutos), dependendo da situação específica que resultou na privação de fluxo. Consulte O que é privação de fluxo? em Perguntas frequentes sobre solução de problemas.	✓	✓	
Streamer	Uma pessoa ou um dispositivo que envia um fluxo de vídeo ou áudio para o IVS.	✓	✓	

Prazo	Descrição	LL	RT	Chat
Assinante	Participante de um evento em tempo real que recebe vídeo ou áudio dos hosts. Consulte O que é o Streaming em tempo real do IVS?.		✓	
Tag	É um rótulo de metadados atribuído a um recurso da AWS. As tags ajudam a identificar e organizar os recursos da AWS. Na página de destino da documentação do IVS, consulte “Marcação” em qualquer documentação da API do IVS (para streaming em tempo real, streaming de baixa latência ou chat).	✓	✓	✓
Filtros de câmera de terceiros	Componentes de software que podem ser integrados ao SDK de Transmissão do IVS para permitir que um aplicação processe imagens antes de fornecê-las ao SDK de Transmissão como uma fonte de imagem personalizada. Um filtro de câmera de terceiros pode processar imagens da câmera, aplicar um efeito de filtro etc.	✓	✓	
Miniatura	Uma imagem de tamanho reduzido obtida de um fluxo. Por padrão, as miniaturas são geradas a cada 60 segundos, mas um intervalo menor pode ser configurado. A resolução da miniatura depende do tipo de canal. Consulte Conteúdo do registro em Gravação automática no Amazon S3 (streaming de baixa latência).	✓		

Prazo	Descrição	LL	RT	Chat
Metadados temporizados	Metadados vinculados a carimbos de data e hora específicos em um fluxo. Eles podem ser adicionados programaticamente por meio da API do IVS e tornam-se associados a quadros específicos. Isso garante que todos os visualizadores recebam os metadados no mesmo ponto em relação ao fluxo. Metadados sincronizados podem ser usados para acionar ações no cliente, como atualizar as estatísticas da equipe durante um evento esportivo. Consulte Como inserir metadados em um stream de vídeo.	✓		
Transcodificação	Converte vídeo e áudio de um formato para outro. Um fluxo de entrada pode ser transcodificado para um formato diferente em várias taxas de bits e resoluções para oferecer suporte a uma variedade de dispositivos de reprodução e condições de rede.	✓	✓	
Transmux	Repetição simples de um empacotamento de um fluxo ingerido para o Amazon IVS, sem recodificação do fluxo de vídeo. “Transmux” é a abreviação de multiplexação de transcodificação, um processo que altera o formato de um arquivo de áudio ou vídeo, mantendo alguns ou todos os fluxos originais. Realizar transmux resulta na conversão para um formato de contêiner diferente sem alterar o conteúdo do arquivo. Diferente de transcodificação.	✓	✓	

Prazo	Descrição	LL	RT	Chat
Fluxos variantes	Um conjunto de codificações da mesma transmissão em vários níveis de qualidade distintos. Cada fluxo variante é codificado como uma lista de reprodução HLS separada. Um índice dos fluxos variantes disponíveis é chamado de lista de reprodução multivariante. Depois que o reprodutor do IVS recebe uma lista de reprodução multivariante do IVS, ele pode escolher entre os fluxos variantes durante a reprodução, mudando continuamente de um para outro à medida que as condições da rede mudam.	✓		
VBR	Taxa de bits variável, um método de controle de taxa para codificadores que usa uma taxa de bits dinâmica que muda durante a reprodução, dependendo do nível de detalhe necessário. É altamente recomendável não usar a VBR devido a problemas de qualidade de vídeo. Em vez disso, use CBR.	✓	✓	

Prazo	Descrição	LL	RT	Chat
Exibição	Trata-se de uma sessão de exibição exclusiva fazendo download ou reproduzindo vídeo de forma ativa. As visualizações são a base para a cota de visualizações simultâneas. Uma exibição se inicia quando uma sessão de visualização dá início à uma reprodução de vídeo. Uma exibição termina quando uma sessão de visualização interrompe a reprodução de vídeo. A reprodução é o único indicador de audiência; não são consideradas heurísticas de engajamento, como níveis de áudio, foco da guia do navegador e qualidade do vídeo. Ao contabilizar as visualizações, o Amazon IVS não considera a legitimidade dos visualizadores nem tenta desduplicar visualizações localizadas, como a execução de vários reprodutores de vídeo em uma única máquina. Consulte Outras cotas em Service Quotas (streaming de baixa latência).	✓		
Visualizador	Uma pessoa que recebe um fluxo do IVS.	✓		

Prazo	Descrição	LL	RT	Chat
WebRTC	Comunicação em tempo real na Web, um projeto de código aberto que fornece comunicação em tempo real aos navegadores da Web e às aplicações móveis. Ele permite que a comunicaç ão de áudio e vídeo funcione dentro de páginas da Web, permitindo a comunicação direta entre pares, eliminando a necessidade de instalar plug-ins ou baixar aplicações nativas. As tecnologias por trás do WebRTC são implement adas como um padrão aberto da Web e estão disponíveis como APIs do JavaScript regulares em todos os principais navegadores ou como bibliotec as para clientes nativos, como Android e iOS.	✓	✓	

Prazo	Descrição	LL	RT	Chat
WHIP	WebRTC-HTTP Ingestion Protocol, um protocolo baseado em HTTP que permite a ingestão de conteúdo com base em WebRTC em serviços de streaming e/ou CDNs. O WHIP é um esboço do IETF desenvolvido para padronizar a ingestão de WebRTC. O WHIP possibilita a compatibilidade com softwares como o OBS, oferecendo uma alternati va (ao SDK de transmissão do IVS) para editoraçã o eletrônica. Streamers mais sofisticados familiari zados com o OBS podem preferi-lo por seus recursos avançados de produção, como transições de cena, mixagem de áudio e gráficos de sobreposi ção. O WHIP também é benéfico em situações em que o uso do SDK de transmissão do IVS não é viável ou preferido. Por exemplo, em configurações que envolvem codificadores de hardware, o SDK de transmissão do IVS pode não ser uma opção. No entanto, se o codificador for compatível com WHIP, ainda será possível publicar diretamente do codificador para o IVS. Consulte Suporte a WHIP no IVS (streaming em tempo real).		✓	
WSS	WebSocket Secure, um protocolo para estabelec er WebSockets por meio de uma conexão TLS criptografada. Ele está sendo usado para se conectar aos endpoints do Chat do IVS. Consulte Step 4: Send and Receive Your First Message em Getting Started with IVS Chat.			✓

Histórico de documentos do Chat do IVS

As tabelas a seguir descrevem as alterações importantes na documentação do Chat do Amazon IVS. Atualizamos a documentação com frequência, para novas versões e para atender aos comentários que vocês nos enviam.

Alterações no Guia do usuário do Chat

Alteração	Descrição	Data
Dividir um Guia do usuário do Chat	As principais alterações na documentação acompanham esta versão. Transferimos as informações de chat do Guia do usuário de streaming de baixa latência do IVS para um novo Guia do usuário do Chat do IVS, localizado na seção existente do Chat do IVS da página de destino da documentação do IVS. Para visualizar outras alterações na documentação, consulte Histórico do documento (streaming de baixa latência).	28 de dezembro de 2023
Glossário do IVS	Ampliou o glossário, abordando termos do IVS em tempo real, baixa latência e bate-papo.	20 de dezembro de 2023

Alterações na Referência da API do Chat do IVS

Alterações de API	Descrição	Data
Dividir um Guia do usuário do Chat	Agora que há um Guia do Usuário do Chat do IVS (criado nesta versão), as entradas do Histórico do documento para a Referência da API do Chat do IVS e a Referência da API de mensagens do Chat do IVS existentes estarão localizadas aqui a partir de agora. As entradas anteriores do histórico para aquelas Referências da API do Chat podem ser encontrad as no Histórico do documento (streaming de baixa latência).	28 de dezembro de 2023

Notas de versão do Chat do IVS

Este documento contém todas as notas de lançamento do Chat do Amazon IVS, começando com as mais recentes, organizadas por data de lançamento.

28 de dezembro de 2023

Guia do usuário do Chat do Amazon IVS:

O Chat do Amazon Interactive Video Service (IVS) é um recurso gerenciado de chat em tempo real para acompanhar suas transmissões de vídeo ao vivo. Nesta versão, transferimos as informações de chat do Guia do usuário do streaming de baixa latência do IVS para um novo Guia do usuário do Chat do IVS. A documentação está acessível na página inicial da documentação do Amazon IVS.

31 de janeiro de 2023

SDK de Mensagens para Clientes do Chat do Amazon IVS: Android 1.1.0

Plataforma	Downloads e alterações
SDK de Mensagens para Clientes do Chat para Android 1.1.0	Documentação de referência: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/ • Para oferecer suporte ao Kotlin Coroutines, adicionamos novas APIs de Mensagens de Chat do IVS no pacote com.amazonaws.ivs.chat.messaging.coroutines. Consulte também o novo tutorial de Kotlin Coroutines: a parte 1 (de 2) é referente às Salas de chat.

Tamanho do Chat Client Messaging SDK: Android

Arquitetura	Tamanho compactado	Tamanho descompactado
Todas as arquiteturas (código de bytes)	89 KB	92 KB

9 de novembro de 2022

Amazon IVS Chat Client Messaging SDK: JavaScript 1.0.2

Plataforma	Downloads e alterações
JavaScript Chat Client Messaging SDK 1.0.2	Documentação de referência: https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/ • Corrigido um problema que afetava o Firefox: os clientes recebiam erroneamente um erro de soquete quando eram desconectados de uma sala de chat usando o endpoint DisconnectUser.

8 de setembro de 2022

Amazon IVS Chat Client Messaging SDK: Android 1.0.0 e iOS 1.0.0

Plataforma	Downloads e alterações
Android Chat Client Messaging SDK 1.0.0	Documentação de referência: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/
iOS Chat Client Messaging SDK 1.0.0	Documentação de referência: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/

Tamanho do Chat Client Messaging SDK: Android

Arquitetura	Tamanho compactado	Tamanho descompactado
Todas as arquiteturas (código de bytes)	53 KB	58 KB

Tamanho do Chat Client Messaging SDK: iOS

Arquitetura	Tamanho compactado	Tamanho descompactado
ios-arm64_x86_64-simulator (código de bits)	484 KB	2,4 MB
ios-arm64_x86_64-simulator	484 KB	2,4 MB
ios-arm64 (código de bits)	1,1 MB	3,1 MB
ios-arm64	233 KB	1,2 MB