Guía del usuario de Chat

Amazon IVS

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

Amazon IVS: Guía del usuario de Chat

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

Las marcas comerciales y la imagen comercial de Amazon no se pueden utilizar en relación con ningún producto o servicio que no sea de Amazon, de ninguna manera que pueda causar confusión entre los clientes y que menosprecie o desacredite a Amazon. Todas las demás marcas comerciales que no son propiedad de Amazon son propiedad de sus respectivos propietarios, que pueden o no estar afiliados, relacionados o patrocinados por Amazon.

Table of Contents

¿Qué es Chat de IVS? ... 1
Introducción al Chat de IVS ... 2
Paso 1: realizar la configuración inicial ... 3
Paso 2: crear una sala de chat ... 4
Instrucciones de la consola ... 5
Instrucciones de la CLI ... 8
Paso 3: crear un token de chat ... 10
Instrucciones del SDK de AWS ... 11
Instrucciones de la CLI ... 12
Paso 4: enviar y recibir su primer mensaje ... 13
Paso 5: verificar sus limites de Service-Quota (opcional) ... 15
Registro de chat ... 16
Habilitar el registro de chat para una sala ... 16
Contenido del mensaje ... 16
Formato ... 16
Campos ... 17
Bucket de Amazon S3 ... 17
Formato ... 17
Campos ... 17
Ejemplo ... 18
Registros de Amazon CloudWatch ... 18
Formato ... 18
Campos ... 18
Ejemplo ... 19
Amazon Kinesis Data Firehose ... 19
Restricciones ... 19
Monitoreo de errores con Amazon CloudWatch ... 19
Controlador de revisión de mensajes de chat ... 20
Creación de una función de Lambda ... 20
Flujo de trabajo ... 20
Sintaxis de la solicitud ... 20
Cuerpo de la solicitud ... 21
Sintaxis de la respuesta ... 21
Campos de respuesta ... 22

Código de muestra 23
Asociación y disociación de un controlador con una sala 24
Monitorieo de errores con Amazon CloudWatch 24
Supervisión 25
Acceso a métricas de CloudWatch 25
Instrucciones de la consola de CloudWatch 25
Instrucciones de la CLI 26
Métricas de CloudWatch: Chat de IVS 27
SDK de mensajería para clientes de Chat de IVS 31
Requisitos de la plataforma 31
Navegadores de escritorio 31
Navegadores en dispositivos móviles 31
Plataformas nativas 32
Soporte 32
Control de versiones 32
API de chat de IVS para Amazon 33
Guía de Android 34
Introducción 35
Uso del SDK 36
Parte 1 del tutorial de Android: salas de chat 40
Requisitos previos 40
Configuración de un servidor local de autenticación y autorización 41
Creación de un proyecto de Chatterbox 45
Conexión a una sala de chat y observación de actualizaciones de conexión 47
Cree un proveedor de tokens 52
Siguientes pasos 56
Parte 2 del tutorial de Android: mensajes y eventos 56
Requisito previo 57
Creación de una interfaz de usuario para enviar mensajes 57
Aplicación de la vinculación de vista 64
Administrar solicitudes de mensajes de chat 67
Pasos finales 72
Parte 1 del tutorial de las corrutinas de Kotlin: salas de chat 76
Requisitos previos 76
Configuración de un servidor local de autenticación y autorización 77
Creación de un proyecto de Chatterbox 81

- Conexión a una sala de chat y observación de actualizaciones de conexión 83
- Cree un proveedor de tokens 87
- Siguientes pasos 91
- Parte 2 del tutorial de corrutinas de Kotlin: mensajes y eventos 92
- Requisito previo 92
- Creación de una interfaz de usuario para enviar mensajes 92
- Aplicación de la vinculación de vista 100
- Administrar solicitudes de mensajes de chat 102
- Pasos finales 108
- Guía para iOS 111
- Introducción 111
- Uso del SDK 113
- Tutorial para iOS 125
- Guía para JavaScript 125
- Introducción 126
- Uso del SDK 127
- Parte 1 del tutorial de JavaScript: salas de chat 132
- Requisitos previos 133
- Configuración de un servidor local de autenticación y autorización 133
- Creación de un proyecto de Chatterbox 137
- Conectarse a una sala de chat 137
- Creación de un proveedor de tokens 138
- Visualización de las actualizaciones de conexión 140
- Creación de un componente de botón de envío 144
- Creación de una entrada de mensajes 146
- Siguientes pasos 148
- Parte 2 del tutorial de JavaScript: mensajes y eventos 149
- Requisito previo 149
- Suscribirse a los eventos de mensajes de chat 149
- Mostrar los mensajes recibidos 150
- Realizar acciones en una sala de chat 158
- Siguientes pasos 169
- Parte 1 del tutorial de React Native: salas de chat 169
- Requisitos previos 170
- Configuración de un servidor local de autenticación y autorización 170
- Creación de un proyecto de Chatterbox 173

Conectarse a una sala de chat ... 174
Creación de un proveedor de tokens ... 175
Visualización de las actualizaciones de conexión ... 177
Creación de un componente de botón de envío ... 180
Creación de una entrada de mensajes ... 183
Siguientes pasos ... 186
Parte 2 del tutorial de React Native: mensajes y eventos ... 187
Requisito previo ... 187
Suscribirse a los eventos de mensajes de chat ... 187
Mostrar los mensajes recibidos ... 188
Realizar acciones en una sala de chat ... 197
Siguientes pasos ... 205
Prácticas recomendadas para React y React Native ... 206
Creación de un enlace inicializador de sala de chat ... 206
Proveedor de instancias de la sala de chat ... 209
Creación de un oyente de mensajes ... 211
Varias instancias de sala de chat en una aplicación ... 215
Seguridad ... 220
Protección de los datos ... 221
Identity and Access Management ... 221
Público ... 221
Cómo funciona Amazon IVS con IAM ... 221
Identidades ... 222
Políticas ... 222
Autorización basada en etiquetas de Amazon IVS ... 223
Roles ... 223
Acceso privilegiado y sin privilegios ... 223
Prácticas recomendadas para utilizar las políticas ... 223
Ejemplos de políticas basadas en identidad ... 224
Política basada en recursos para chat de Amazon IVS ... 225
Solución de problemas ... 226
Políticas administradas para Amazon IVS ... 227
Uso de roles vinculados a servicios para Amazon IVS ... 227
Registro y monitorización ... 227
Respuesta frente a incidencias ... 227
Resiliencia ... 227

- Seguridad de infraestructuras ... 227
  - API Calls (Llamadas a la API) ... 228
  - Chat de Amazon IVS ... 228
- Service Quotas ... 229
  - Aumentos en la cuota de servicio ... 229
  - Cuotas de tarifa de llamadas a la API ... 229
  - Otras cuotas ... 230
  - Integración de Service Quotas con las métricas de uso de CloudWatch ... 233
    - Creación de una alarma de CloudWatch para métricas de uso ... 234
- Preguntas frecuentes de solución de problemas ... 235
  - ¿Por qué no se desconectaron las conexiones del chat de IVS cuando se eliminó la sala? ... 235
- Glosario ... 236
- Historial de documentos ... 258
  - Cambios en la Guía del usuario de Chat ... 258
  - Cambios en la Referencia de la API de Chat de IVS ... 259
- Notas de la versión ... 260
  - 28 de diciembre de 2023 ... 260
    - Guía del usuario de Chat de Amazon IVS ... 260
  - 31 de enero de 2023 ... 260
    - SDK de mensajería para clientes de Chat de Amazon IVS para Android 1.1.0 ... 260
  - 9 de noviembre de 2022 ... 261
    - SDK de mensajería para clientes de Chat de Amazon IVS: JavaScript 1.0.2 ... 261
  - 8 de septiembre de 2022 ... 261
    - SDK de mensajería para clientes de Chat de Amazon IVS: Android 1.0.0 e iOS 1.0.0 ... 261

Qué es Chat de Amazon IVS

Chat de Amazon IVS es una característica de chat en directo administrada que acompaña a las transmisiones de video en directo. Se puede acceder a la documentación desde la página de inicio de la documentación de Amazon IVS, en la sección de Chat de Amazon IVS:

- Guía del usuario de Chat: este documento, junto con todas las demás páginas de la Guía del usuario que aparecen en el panel de navegación.
- Referencia de la API de chat: API de plano de control (HTTPS).
- Referencia de API de mensajería de chat: API de plano de datos (WebSocket).
- Referencias del SDK para clientes de chat: Android, iOS y JavaScript.

Introducción al Chat de Amazon IVS

El chat de Amazon Interactive Video Service (IVS) es una función de chat en directo administrada para acompañar sus transmisiones de vídeo en directo. (El Chat de IVS también se puede utilizar sin transmisión de video). Puede crear salas de chat y habilitar las sesiones de chat entre sus usuarios.

El chat de Amazon IVS le permite centrarse en crear experiencias de chat personalizadas junto con vídeos en directo. No necesita administrar la infraestructura ni desarrollar y configurar componentes de sus flujos de trabajo de chat. El chat de Amazon IVS es escalable, seguro, fiable y rentable.

El chat de Amazon IVS funciona mejor para facilitar la mensajería entre los participantes de una transmisión de vídeo en directo con un principio y un final.

El resto de este documento lo guiará a través de los pasos necesarios para crear su primera aplicación de chat mediante el chat de Amazon IVS.

Ejemplos: están disponibles las siguientes aplicaciones de demostración (tres aplicaciones de cliente de muestra y una aplicación de servidor de fondo para la creación de tokens):

- Demostración web de chat de Amazon IVS
- Demo de chat de Amazon IVS para Android
- Demostración de chat de Amazon IVS para iOS
- Backend de demostración de chat de Amazon IVS

Importante: Las salas de chat que no tienen conexiones o actualizaciones nuevas durante 24 meses se eliminan automáticamente.

Temas

- Paso 1: realizar la configuración inicial
- Paso 2: crear una sala de chat
- Paso 3: crear un token de chat
- Paso 4: enviar y recibir su primer mensaje
- Paso 5: verificar sus limites de Service-Quota (opcional)

Paso 1: realizar la configuración inicial

Antes de continuar, debe realizar lo siguiente:

1. Crear una cuenta de AWS
2. Configurar los usuarios raíz y administrativo
3. Configurar los permisos de AWS Identity and Access Management (IAM) Utilizar la política que se especifica a continuación

Para conocer los pasos específicos sobre lo que se ha mencionado anteriormente, consulte Introducción al streaming de baja latencia de IVS en la Guía del usuario de Amazon IVS. Importante: En el "Paso 3: configurar permisos de IAM", utilice esta política para el chat de IVS:

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

Paso 2: crear una sala de chat

Una sala de chat de Amazon IVS tiene asociada información de configuración (por ejemplo, longitud máxima del mensaje).

En las instrucciones de esta sección, se muestra cómo utilizar la consola o AWS CLI para configurar salas de chat (incluida la configuración opcional para revisar o registrar los mensajes) y crear salas.

Instrucciones de la consola

Estos pasos se dividen en fases, que empiezan con la configuración de la sala inicial y terminan con la creación de la sala final.

Opcionalmente, puede configurar una sala para que se revisen los mensajes. Por ejemplo, puede actualizar el contenido de los mensajes o los metadatos, denegar mensajes para evitar que se envíen o dejar pasar el mensaje original. Esto se explica en Configuración para revisar mensajes de sala (opcional).

De forma opcional, puede configurar una sala para que se registren los mensajes. Por ejemplo, si sus mensajes se envían a una sala de chat, puede registrarlos en un bucket de Amazon S3, en Amazon CloudWatch o Amazon Kinesis Data Firehose. Esto se explica en Configuración para registrar los mensajes (opcional).

Configuración inicial de sala

1. Abra la consola de chat de Amazon IVS.

   (También puede acceder a la consola de Amazon IVS a través de la consola de administración de AWS).
2. En la barra de navegación, utilice el menú desplegable Select a Region (Seleccionar una región) para elegir una región. Su nueva sala se creará en esta región.
3. En la casilla Introducción (parte superior derecha), elija Sala de chat de Amazon IVS. Aparece la ventana Crear 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. Abajo de Configuración, especifique opcionalmente un Nombre de sala. Los nombres de las salas no son únicos, pero proporcionan una forma de distinguir otras salas además del ARN (nombre de recurso de Amazon) del canal.
5. Abajo de Configuración > Configuración de salas, acepte la Configuración predeterminada o bien, seleccione Configuración personalizada y, a continuación, configure la Longitud máxima del mensaje o Velocidad máxima de mensajes.
6. Si desea revisar los mensajes, continúe con Configuración para revisar mensajes de sala (opcional) a continuación. De lo contrario, omita eso (es decir, acepte el Controlador de revisión de mensajes > Deshabilitado) y proceda directamente a Creación final de sala.

Configurar para revisar mensajes de sala (opcional)

1. Abajo de Controlador de revisión de mensajes, seleccione Controlar con AWS Lambda. La sección del Controlador de revisión de mensajes se expande para mostrar opciones adicionales.
2. Configuración del Resultado alternativo para Permitir o Negar el mensaje si el controlador no devuelve una respuesta válida, encuentra un error o supera el período de espera.
3. Especifique su Función de Lambda existente o utilice Creación de una función de Lambda para crear una nueva función.

   La función Lambda debe estar en la misma cuenta región de AWS y en la misma cuenta de AWS que la sala de chat. Debe dar permiso al servicio SDK de Amazon Chat para invocar su recurso lambda. La política basada en recursos se creará automáticamente para la función lambda que ha seleccionado. Para obtener más información sobre los permisos, consulte Resource-Based Policy for Amazon IVS Chat.

Configurar para registrar mensajes (opcional)

1. En Registro de mensajes, seleccione Registrar los mensajes de chat de forma automática. La sección del Registro de mensajes se expande para mostrar opciones adicionales. Puede agregar una configuración de registro actual a esta sala o crear una nueva seleccionando Crear configuración de registro.
2. Si elige una configuración de registro actual, aparece un menú desplegable que muestra todas las que ya creó. Seleccione una de la lista y sus mensajes de chat se registrarán en este destino de forma automática.

3. Si elige Create logging configuration (Crear configuración de registro), aparece una ventana modal que le permite crear y personalizar una configuración de registro nueva.
    a. Si lo desea, especifique un nombre para la configuración de registro. Estos nombres, como los de las salas, no son únicos, pero permiten distinguir las configuraciones de registro diferentes a las del ARN.
    b. En Destination (Destino), seleccione el grupo de registros de CloudWatch, el flujo de entrega de Kinesis Firehose o el bucket de Amazon S3 para elegir el destino de sus registros.
    c. Según el Destino, seleccione la opción para crear un grupo de registro de CloudWatch nuevo o utilizar uno actual, un flujo de entrega de Kinesis Firehose o un bucket de Amazon S3.
    d. Después de revisar, elija Create (Crear) para crear una configuración de registro nueva un ARN único. Esto adjunta la configuración de registro nueva a la sala de chat automáticamente.

Creación final de sala

1. Después de revisar, elija Create chat room (Crear sala de chat) para crear una nueva con un ARN único.

Instrucciones de la CLI

Creación de una sala de chat

Crear una sala de chat con la AWS CLI es una opción avanzada y requiere que primero descargue y configure la CLI en su equipo. Para obtener más información, consulte la Guía del usuario de la interfaz de línea de comandos de AWS.

1. Ejecute el comando `create-room` de chat y pase un nombre opcional:

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

2. Esto devuelve una nueva 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. Tenga en cuenta el campo `arn`. Lo necesitará para crear un token de cliente y conectarse a una sala de chat.

Preparación de una configuración de registro (opcional)

Tal como sucede con la creación de una sala de chat, la preparación de la configuración de registro es una opción avanzada y requiere que primero descargue y configure la CLI en su equipo. Para obtener más información, consulte la Guía del usuario de la interfaz de línea de comandos de AWS.

1. Ejecute el comando `create-logging-configuration` de chat y pase un nombre opcional y una configuración de destino que apunte a un bucket de Amazon S3 por nombre. Este bucket de Amazon S3 debe existir antes de crear la configuración de registro. (Para obtener información sobre la creación de un bucket de Amazon S3, consulte la Documentación de Amazon S3).

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

2. Esto devuelve una configuración de registro nueva:

```
{
   "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. Tenga en cuenta el campo `arn`. Lo necesita para adjuntar la configuración de registro a la sala de chat.

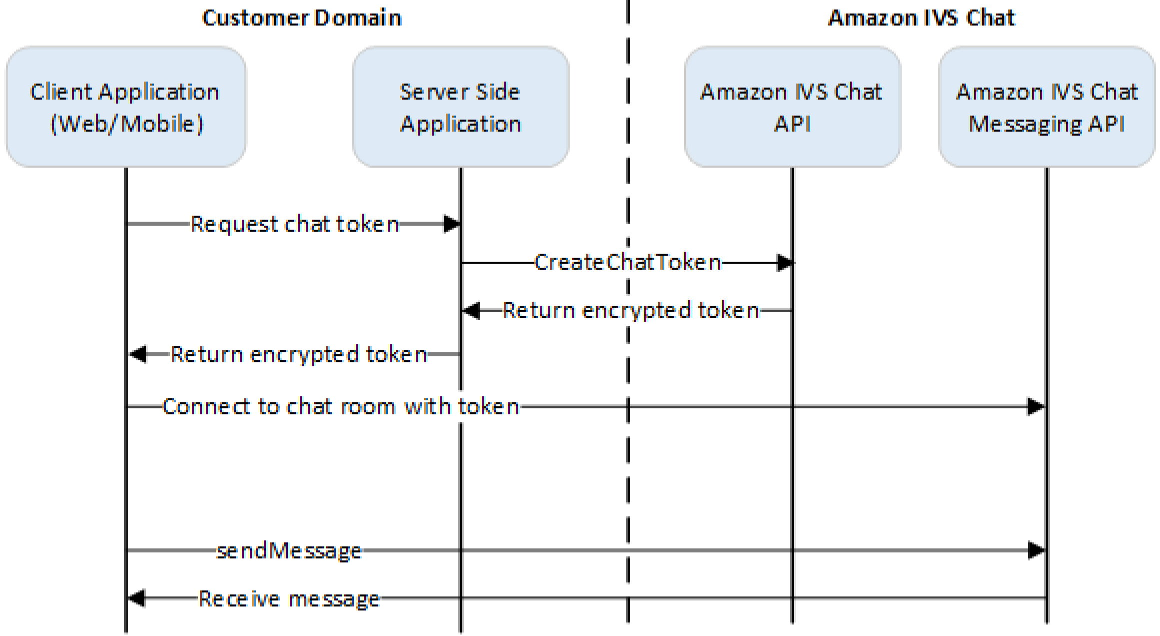
a. Si está creando una sala de chat nueva, ejecute el comando `create-room` y pase el `arn` de la configuración de registro:

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

b. Si está actualizando una sala de chat actual, ejecute el comando `update-room` y pase el `arn` de la configuración de registro:

```
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"
```

Paso 3: crear un token de chat



Para que un participante del chat se conecte a una sala y comience a enviar y recibir mensajes, se debe crear un token de chat. Los tokens de chat se utilizan para autenticar y autorizar a los clientes de chat. Como se muestra arriba, una aplicación cliente solicita un token a su aplicación del lado del servidor y la aplicación del lado del servidor llama a CreateChatToken mediante solicitudes firmadas por AWS SDK o Sigv4. Como las credenciales de AWS se utilizan para llamar a la API, el token debe generarse en una aplicación segura del lado del servidor, no en la aplicación del lado del cliente.

Una aplicación de servidor de fondo que muestra la generación de tokens está disponible en Backend de demostración de chat de Amazon IVS.

Duración de la sesión hace referencia a cuánto tiempo puede permanecer activa una sesión establecida antes de que finalice automáticamente. Es decir, la duración de la sesión es el tiempo que el cliente puede permanecer conectado a la sala de chat antes que se deba generar un nuevo token y establecer una nueva conexión. Durante la creación de tokens, puede especificar la duración de la sesión como opción.

Cada token solo se puede utilizar una vez para establecer una conexión para un usuario final. Si una conexión está cerrada, se debe crear un nuevo token antes de restablecer una conexión. El token en sí es válido hasta la fecha de caducidad del token incluida en la respuesta.

Cuando un usuario final quiere conectarse a una sala de chat, el cliente debe solicitar un token a la aplicación del servidor. La aplicación de servidor crea un token y lo envía al cliente. Los tokens deben crearse para los usuarios finales a pedido.

Para crear un token de autenticación de chat, siga las instrucciones debajo. Al crear un token de chat, utilice los campos de solicitud para transferir datos sobre el usuario final del chat y las capacidades de mensajería del usuario final; para obtener más información, consulte CreateChatToken en la Referencia de la API del Chat de IVS.

Instrucciones del SDK de AWS

Crear un token de chat con el SDK de AWS requiere que descargue y configure primero el SDK en su aplicación. A continuación, se muestran las instrucciones para el SDK de AWS mediante JavaScript.

Importante: Este código debe ejecutarse en el lado del servidor y su salida se debe pasar al cliente.

Requisito previo: para utilizar el ejemplo de código siguiente, debe cargar el SDK de JavaScript de AWS en su aplicación. Para obtener más información, consulte Introducción a SDK de AWS para 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);
```

Instrucciones de la CLI

Crear un token de chat con la AWS CLI es una opción avanzada y requiere que primero descargue y configure la CLI en su equipo. Para obtener más información, consulte la Guía del usuario de la interfaz de línea de comandos de AWS. Nota: La generación de tokens con la CLI de AWS es buena para realizar pruebas, pero para uso en producción, le recomendamos que genere tokens en el lado del servidor con el SDK de AWS (consulte las instrucciones anteriores).

1. Ejecute el comando `create-chat-token` junto con el identificador de sala y el identificador de usuario del cliente. Incluya cualquiera de las siguientes capacidades: `"SEND_MESSAGE"`, `"DELETE_MESSAGE"`, `"DISCONNECT_USER"`. (Opcionalmente, incluya la duración de la sesión [en minutos] o atributos personalizados [metadatos] sobre esta sesión de chat. Estos campos no se muestran a continuación.)

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

2. Esto devuelve un 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. Guarde este token. Lo necesitará para conectarse a la sala de chat y enviar o recibir mensajes. Tendrá que generar otro token de chat antes de que finalice la sesión (como se indica en `sessionExpirationTime`).

Paso 4: enviar y recibir su primer mensaje

Utilice el token de chat para conectarse a una sala de chat y enviar su primer mensaje. A continuación, se proporciona un código JavaScript de prueba. Los SDK de los clientes de IVS también están disponibles: consulte SDK de chat: guía para Android, SDK de chat: guía para iOS y SDK de chat: guía para JavaScript.

Servicio regional: el código de prueba que aparece a continuación se refiere a la “región de elección admitida”. El chat de Amazon IVS ofrece puntos de conexión regional que puede utilizar para realizar sus solicitudes. Para la API de mensajería de chat de Amazon IVS, la sintaxis general de un punto de conexión regional es:

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

Por ejemplo, el punto de conexión de la región Oeste de EE. UU. (Oregón) es wss://edge.ivschat.us-west-2.amazonaws.com. Para obtener una lista de las regiones admitidas, consulte la información de chat de Amazon IVS en Página de Amazon IVS en la Referencia general de 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);
}
```

¡Enhorabuena, está listo! Ahora tiene una aplicación de chat sencilla que puede enviar o recibir mensajes.

Paso 5: verificar sus limites de Service-Quota (opcional)

Sus salas de chat se ampliarán junto con la transmisión en directo de Amazon IVS para permitir que todos los espectadores participen en conversaciones de chat. Sin embargo, todas las cuentas de Amazon IVS tienen límites en el número de participantes en el chat simultáneo y la velocidad de entrega de mensajes.

Asegúrese de que sus límites son adecuados y solicite un aumento si es necesario, especialmente si está planificando un evento de streaming grande. Para obtener más información, consulte Service Quotas (Low-Latency Streaming), Service Quotas (Real-Time Streaming) y Cuotas de servicio (Chat).

Registro de chat

La función de registro de chat permite registrar todos los mensajes de una sala en cualquiera de las tres ubicaciones estándar: un bucket de Amazon S3, Registros de Amazon CloudWatch o Amazon Kinesis Data Firehose. Luego, puede utilizar los registros para analizarlos o crear una repetición del chat que se vincule a una sesión de video en directo.

Habilitar el registro de chat para una sala

El registro de chat es una opción avanzada que se puede habilitar mediante la vinculación de una configuración de registro con una sala. La configuración de registro es un recurso que permite especificar el tipo de ubicación (el bucket de Amazon S3, los registros de Amazon CloudWatch o Amazon Kinesis Data Firehose) en la que se registran los mensajes de una sala. Para obtener más información sobre la creación y la administración de las configuraciones de registro, consulte la Introducción al chat de Amazon IVS y la Referencia de la API de chat de Amazon IVS.

Puede asociar hasta tres configuraciones de registro a cada sala, ya sea cuando crea una nueva (CreateRoom) o cuando actualiza una actual (UpdateRoom). Puede asociar varias salas con la misma configuración de registro.

Cuando se asocia al menos una configuración de registro activa a una sala, todas las solicitudes de mensajería enviadas a esa sala a través de la API de mensajería de chat de Amazon IVS se graban automáticamente en las ubicaciones especificadas. Estos son los retrasos de propagación promedio (desde que se envía una solicitud de mensaje hasta que está disponible en las ubicaciones especificadas):

- Bucket de Amazon S3: 5 minutos
- Registros de Amazon CloudWatch o Amazon Kinesis Data Firehose: 10 segundos

Contenido del mensaje

Formato

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

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

Campos

Campo	Descripción
`event_timestamp`	Marca de tiempo UTC de cuando el chat de Amazon IVS recibió el mensaje.
`payload`	La carga JSON de Mensaje (suscripción) o Evento (suscripción) que los clientes recibirán del servicio de chat de Amazon IVS.
`type`	Tipo de mensaje de chat. • Valores válidos: `MESSAGE` | `EVENT`
`version`	Versión del formato del contenido del mensaje.

Bucket de Amazon S3

Formato

Los registros de mensajes se organizan y almacenan con el siguiente prefijo S3 y formato de archivo:

```
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	Descripción
`<account_id>`	ID de la cuenta de AWS a partir de la que se creó la sala.

Campo	Descripción
`<hash>`	Un valor de hash que genera el sistema para garantizar la unicidad.
`<region>`	La región del servicio de AWS en la que se creó la sala.
`<resource_id>`	La parte de identificación del recurso del ARN de la sala.
`<version>`	Versión del formato del contenido del mensaje.
`<year> / <month> / <day> / <hours> / <minute>`	Marca de tiempo UTC de cuando el chat de Amazon IVS recibió el mensaje.

Ejemplo

```
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
```

Registros de Amazon CloudWatch

Formato

Los registros de mensajes se organizan y almacenan con el siguiente formato de nombre de flujo de registro:

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

Campos

Campo	Descripción
`<resource_id>`	Parte de identificación del recurso del ARN de la sala.
`<version>`	Versión del formato del contenido del mensaje.

Ejemplo

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

Amazon Kinesis Data Firehose

Los registros de mensajes se envían al flujo de entrega como datos de transmisión en tiempo real a destinos como Amazon Redshift, Amazon OpenSearch Service, Splunk y cualquier punto de conexión HTTP personalizado o que sea de proveedores de servicios de terceros compatibles. Para obtener más información, consulte ¿Qué es Amazon Kinesis Data Firehose?.

Restricciones

- Debe ser el propietario de la ubicación de registro en la que se almacenarán los mensajes.
- La sala, la configuración del registro y su ubicación deben estar en la misma región de AWS.
- Solo las configuraciones de registro activas están disponibles para el registro de chat.
- Solo puede eliminar una configuración de registro que ya no esté asociada a ningún canal.

El registro de mensajes en una ubicación de su propiedad requiere autorización con sus credenciales de AWS. Para otorgarle al chat de IVS el acceso necesario, se genera automáticamente una política de recursos (para un bucket de Amazon S3 o los registros de CloudWatch) o un rol vinculado a servicios (SLR) de AWS IAM (para Amazon Kinesis Data Firehose) cuando se crea la configuración de registro. Tenga cuidado con las modificaciones del rol o las políticas, ya que eso puede afectar el permiso de registro del chat.

Monitoreo de errores con Amazon CloudWatch

Puede monitorear los errores que se producen en el registro del chat con Amazon CloudWatch y crear alarmas o paneles de mando para indicar o responder a los cambios de errores específicos.

Existen varios tipos de errores. Para obtener más información, consulte Supervisión de Chat de Amazon IVS.

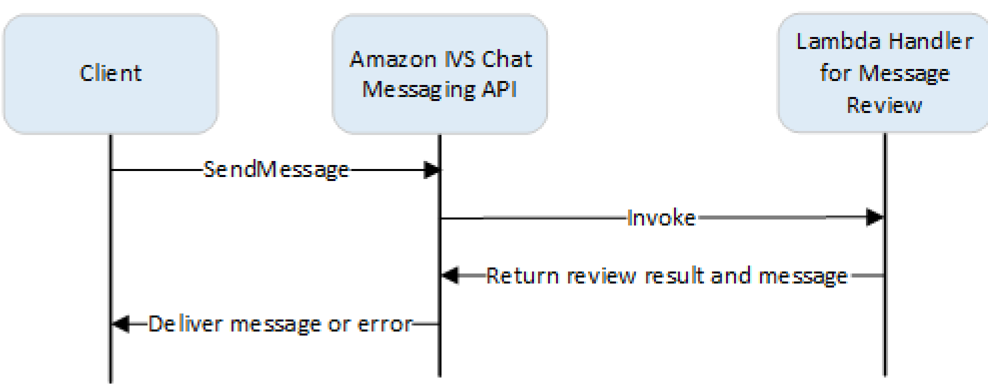
Controlador de revisión de mensajes de chat

Un controlador de revisión de mensajes le permite revisar o modificar los mensajes antes de entregarlos a una sala. Cuando un controlador de revisión de mensajes está asociado a una sala, se invoca para cada solicitud de SendMessage de esa sala. El controlador de revisión de mensajes hace cumplir la lógica empresarial de la aplicación y determina si permitir, denegar o modificar un mensaje. El chat de Amazon IVS admite las funciones de AWS Lambda como controladores.

Creación de una función de Lambda

Antes de configurar un controlador de revisión de mensajes para una sala, debe crear una función lambda con una política de IAM basada en recursos. La función lambda debe estar en la misma cuenta y región de AWS que la sala con la que utilizará la función. La política basada en recursos otorga permiso al chat de Amazon IVS para invocar la función lambda. Para obtener instrucciones, consulte Resource-Based Policy for Amazon IVS Chat.

Flujo de trabajo



Sintaxis de la solicitud

Cuando un cliente envía un mensaje, el chat de Amazon IVS invoca la función lambda con una carga útil JSON:

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

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

Cuerpo de la solicitud

Campo	Descripción
`Attributes`	Atributos asociados al mensaje.
`Content`	Contenido original del mensaje.
`MessageId`	El ID del mensaje. Generado por el chat de IVS.
`RoomArn`	El ARN de la sala a la que se envían los mensajes.
`Sender`	Información sobre el remitente. Este objeto tiene varios campos: • `Attributes` : metadatos sobre el remitente establecido durante la autenticación. Esto se puede utilizar para proporcionar al cliente más información sobre el remitente; por ejemplo, URL de avatar, insignias, fuente y color. • `UserId`: un identificador especificado por la aplicación del espectador (usuario final) que envió este mensaje. La aplicación cliente puede utilizar esto para hacer referencia al usuario en la API de mensajería o en los dominios de aplicación. • `Ip`: la dirección IP del cliente que envía el mensaje.

Sintaxis de la respuesta

La función lambda del controlador debe devolver una respuesta JSON con la siguiente sintaxis. Las respuestas que no corresponden con la siguiente sintaxis o que no satisfagan las restricciones de campo no son válidas. En este caso, el mensaje es permitido o denegado según el valor `FallbackResult` que especifique en el controlador de revisión de mensajes; consulte MessageReviewHandler en la Amazon IVS Chat API Reference (Referencia de la API de chat de Amazon IVS).

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

Campos de respuesta

Campo	Descripción
`Attributes`	Atributos asociados al mensaje devuelto desde la función lambda. Si `ReviewResult` es `DENY`, un `Reason` puede proporcionarse en `Attributes` ; por ejemplo: `"Attributes": {"Reason": "denied for moderation` En este caso, el cliente remitente recibe un error de WebSocket 406 con el motivo del mensaje de error. (Consulte WebSocket Errors [Errores WebSocket] en la Amazon IVS Chat Messaging API Reference [Referencia de la API de mensajería de chat de Amazon IVS]). • Restricciones de tamaño: máximo 1 KB • Requerido: no
`Content`	Contenido del mensaje devuelto desde la función Lambda. Podría editarse o ser original según la lógica empresarial. • Limitaciones de longitud: longitud mínima de 1. Longitud máxima del `MaximumMessageLength` que definió al crear o actualizar la sala. Para obtener más información, consulte la Referencia de la API de chat de Amazon IVS. Esto se aplica solo cuando `ReviewResult` es `ALLOW`. • Obligatorio: sí
`ReviewResult`	El resultado del procesamiento de revisiones sobre cómo manejar el mensaje. Si es permitido, el mensaje se entrega a todos los usuarios conectados a la sala. Si se deniega, el mensaje no se entrega a ningún usuario. • Valores válidos: `ALLOW` | `DENY`

Campo	Descripción
	• Obligatorio: sí

Código de muestra

A continuación se muestra un controlador lambda de muestra en Go. Este modifica el contenido del mensaje, mantiene los atributos del mensaje sin cambios y permite el mensaje.

```
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
}
```

Asociación y disociación de un controlador con una sala

Una vez que haya configurado e implementado el controlador lambda, utilice la API de chat de Amazon IVS:

- Para asociar el controlador a una sala, llame a CreateRoom o UpdateRoom y especifique el controlador.
- Para desasociar el controlador de una sala, llame a UpdateRoom con un valor vacío para `MessageReviewHandler.Uri`.

Monitorieo de errores con Amazon CloudWatch

Puede monitorear los errores que se producen en la revisión de mensajes con Amazon CloudWatch y crear alarmas o paneles de mando para indicar o responder a los cambios de errores específicos. Si se produce un error, el mensaje se permite o deniega según el `FallbackResult` valor que especifique al asociar el controlador a una sala; consulte Handler de revisión de mensajes en la Referencia de la API de chat de Amazon IVS.

Existen varios tipos de errores:

- `InvocationErrors` ocurre cuando Amazon IVS Chat no puede invocar un controlador.
- `ResponseValidationErrors` ocurre cuando un controlador devuelve una respuesta que no es válida.
- AWS Lambda `Errors` ocurre cuando un controlador lambda devuelve un error de función cuando se ha invocado.

Para obtener más información sobre los errores de invocación y de validación de respuestas (que emite Chat de Amazon IVS), consulte Supervisión de Chat de Amazon IVS. Para obtener más información sobre los errores de AWS Lambda, consulte Trabajar con métricas de Lambda.

Supervisión de Chat de Amazon IVS

Puede supervisar los recursos de Chat de Amazon Interactive Video Service (IVS) mediante Amazon CloudWatch. CloudWatch recopila y procesa los datos sin formato de Chat de Amazon IVS en métricas legibles y casi en tiempo real. Estas estadísticas se mantienen durante 15 meses, de forma que pueda obtener una perspectiva histórica sobre el rendimiento de su aplicación web o servicio. Puede establecer alarmas que vigilen determinados umbrales y enviar notificaciones o realizar acciones cuando se alcancen dichos umbrales. Para obtener más información, consulte la Guía del usuario de CloudWatch.

Acceso a métricas de CloudWatch

Amazon CloudWatch recopila y procesa los datos sin procesar de Chat de Amazon IVS en métricas legibles y casi en tiempo real. Estas estadísticas se mantienen durante 15 meses, de forma que pueda obtener una perspectiva histórica sobre el rendimiento de su aplicación web o servicio. Puede establecer alarmas que vigilen determinados umbrales y enviar notificaciones o realizar acciones cuando se alcancen dichos umbrales. Para obtener más información, consulte la Guía del usuario de CloudWatch.

Tenga en cuenta que las métricas de CloudWatch se acumulan a lo largo del tiempo. En la práctica, la resolución disminuye a medida que las métricas envejecen. El esquema es el siguiente:

- Las métricas de 60 segundos están disponibles durante 15 días.
- Las métricas de 5 minutos están disponibles durante 63 días.
- Las métricas de 1 hora están disponibles durante 455 días (15 meses).

Para obtener información actualizada sobre la retención de datos, busque el “periodo de retención” en Preguntas frecuentes sobre Amazon CloudWatch.

Instrucciones de la consola de CloudWatch

1. Abra la consola de CloudWatch en https://console.aws.amazon.com/cloudwatch/.
2. En el panel de navegación lateral, expanda el menú desplegable Metrics (Métricas) y, a continuación, seleccione All metrics (Todas las métricas).
3. En la pestaña Explorar, mediante el menú desplegable sin etiqueta de la izquierda, seleccione su región de “inicio”, donde se crearon los canales. Para obtener más información sobre las regiones,

consulte Solución global, control regional. Para obtener una lista de las regiones admitidas, consulte la Página de Amazon IVS en la Referencia general de AWS.

4. En la parte inferior de la pestaña Explorar, seleccione el espacio de nombres IVSChat.
5. Realice una de las acciones siguientes:
   a. En la barra de búsqueda, ingrese el ID de recurso (parte del ARN, `arn::ivschat:room/<resource id>`).

      A continuación, seleccione IVSChat.
   b. Si IVSChat aparece como servicio seleccionable en Espacios de nombres de AWS, selecciónelo. Se mostrará si utiliza Amazon IVSChat y envía métricas a Amazon CloudWatch. (Si IVSChat no está en la lista, no tiene ninguna métrica de Amazon IVSChat).

      Luego, elija la agrupación de dimensiones que desee; las dimensiones disponibles se muestran a continuación en Métricas de CloudWatch.
6. Elija métricas para agregarlas al gráfico. Las métricas disponibles se muestran a continuación en Métricas de CloudWatch.

También puede acceder al gráfico de CloudWatch de la sesión de chat desde la página de detalles de la sesión de chat; para ello, seleccione el botón Ver en CloudWatch.

Instrucciones de la CLI

También puede obtener acceso a las métricas mediante la AWS CLI. Esto requiere que primero descargue y configure la CLI en su equipo. Para obtener más información, consulte la Guía del usuario de la interfaz de línea de comandos de AWS.

A continuación, para acceder a las métricas de chat de baja latencia de Amazon IVS mediante AWS CLI:

- En el símbolo del sistema, ejecute:

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

Para obtener más información, consulte Uso de las métricas de Amazon CloudWatch en la Guía del usuario de Amazon CloudWatch.

Métricas de CloudWatch: Chat de IVS

El chat de Amazon IVS proporciona las siguientes métricas en el espacio de nombres AWS/IVSChat.

Métrica	Dimensión	Descripción
`ConcurrentChatConnections`	Ninguna	El número total de conexiones simultáneas en una sala de chat (se informa como máximo por minuto). Esto resulta útil para comprender cuándo los clientes se acercan a su límite de conexiones de chat simultáneas en una región. Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo
`Deliveries`	Acción de	El número de entregas de solicitudes de mensajería hechas de un tipo de acción específica a las conexiones de chat en todas las salas de una región. Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo
`InvocationErrors`	`Uri`	El número de errores de invocación de un controlador de revisión de mensajes específico en todas las salas de una región. Se produce un error de invocación cuando no se puede invocar el controlador de revisión de mensajes. Los errores de invocación se producen cuando el chat de Amazon IVS no puede invocar un controlador. Esto puede ocurrir si el controlador asociado a una sala ya no existe o se agota el tiempo, o si su política de recursos no permite que el servicio la invoque.

Métrica	Dimensión	Descripción
		Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo
`LogDestinationAccessDeniedError`	`LoggingConfiguration`	La cantidad de errores de acceso denegado de un destino de registro en todas las salas de una región. Estos errores se producen cuando el chat de Amazon IVS no puede acceder al recurso de destino que especificó en la configuración de registro. Puede ocurrir si la política del recurso de destino no permite que el servicio coloque registros. Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo
`LogDestinationErrors`	`LoggingConfiguration`	La cantidad de todos los errores de un destino de registro en todas las salas de una región. Se trata de una métrica agregada que incluye todos los tipos de errores que se producen cuando el chat de Amazon IVS no entrega los registros al recurso de destino que especificó en la configuración de registro. Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo

Métrica	Dimensión	Descripción
`LogDestinationResourceNotFoundErrors`	`LoggingConfiguration`	La cantidad de errores recurso no encontrado de un destino de registro en todas las salas de una región. Estos errores se producen cuando el chat de Amazon IVS no puede enviar los registros al recurso de destino que especificó en la configuración de registro porque no existe. Esto puede suceder si el recurso de destino asociado a una configuración de registro ya no existe. Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo
`MessagingDeliveries`	Ninguna	El número de entregas de solicitudes de mensajería a las conexiones de chat en todas las salas de una región. Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo
`MessagingRequests`	Ninguna	El número de solicitudes de mensajería realizadas en todas las salas de una región. Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo

Métrica	Dimensión	Descripción
`Requests`	Acción de	El número de solicitudes realizadas de un tipo de acción específico en todas las salas de una región. Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo
`ResponseValidationErrors`	`Uri`	El número de errores de validación de respuesta de un controlador de revisión de mensajes específico en todas las salas de una región. Se produce un error de validación de respuestas cuando la respuesta del controlador de revisión de mensajes no es válida. Esto puede significar que no se ha podido analizar la respuesta o no se han podido realizar comprobaciones de validación; por ejemplo, un resultado de revisión no válido o valores de respuesta demasiado largos. Unidad: recuento Estadísticas válidas: suma, promedio, máximo, mínimo

SDK de mensajería para clientes de Chat de Amazon IVS

El SDK de mensajería del cliente de chat de Amazon Interactive Video Service (IVS) está dedicado a desarrolladores que crean aplicaciones con Amazon IVS. Este SDK está diseñado a fin de aprovechar la arquitectura de Amazon IVS y ver mejoras continuas y nuevas características, junto con Amazon IVS. Como SDK de transmisión nativo, está diseñado para minimizar el impacto en el rendimiento de la aplicación y en los dispositivos con los que los usuarios acceden a la aplicación.

Requisitos de la plataforma

Navegadores de escritorio

Navegador	Versiones compatibles
Chrome	Dos versiones principales (la versión actual y la anterior más reciente)
Ubicaciones	Dos versiones principales (la versión actual y la anterior más reciente)
Firefox	Dos versiones principales (la versión actual y la anterior más reciente)
Opera	Dos versiones principales (la versión actual y la anterior más reciente)
Safari	Dos versiones principales (la versión actual y la anterior más reciente)

Navegadores en dispositivos móviles

Navegador	Versiones compatibles
Chrome para Android	Dos versiones principales (la versión actual y la anterior más reciente)
Firefox para Android	Dos versiones principales (la versión actual y la anterior más reciente)
Opera para Android	Dos versiones principales (la versión actual y la anterior más reciente)

Navegador	Versiones compatibles
WebView Android	Dos versiones principales (la versión actual y la anterior más reciente)
Internet de Samsung	Dos versiones principales (la versión actual y la anterior más reciente)
Safari para iOS	Dos versiones principales (la versión actual y la anterior más reciente)

Plataformas nativas

Plataforma	Versiones compatibles
Android	5.0 y versiones posteriores
iOS	13.0 y versiones posteriores

Soporte

Si encuentra un error u otro problema en la sala de chat, determine el identificador único a través de la API de chat de IVS (consulteListRooms).

Comparta este identificador de sesión de chat con AWS Support. Con él, pueden obtener información para ayudar a solucionar el problema.

Nota: Consulte Notas de la versión de Chat de Amazon IVS para ver las versiones disponibles y los problemas solucionados. Si procede, antes de contactar con el soporte técnico, actualice su versión del SDK de transmisión y compruebe si se resuelve el problema.

Control de versiones

Los SDK de mensajería del cliente de chat de Amazon IVS utilizan el control de versiones semántico.

Para este análisis, suponga:

- La última versión es la 4.1.3.

- La última versión de la versión principal anterior es la 3.2.4.
- La última versión de la versión 1.x es la 1.5.6.

Las características nuevas compatibles con versiones anteriores se agregan como versiones secundarias de la última versión. En este caso, el siguiente conjunto de características nuevas se agregará como la versión 4.2.0.

Se agregan correcciones de errores menores compatibles con versiones anteriores como parches de la última versión. Aquí, el siguiente conjunto de correcciones de errores menores se agregará como la versión 4.1.4.

Las correcciones de errores principales compatibles con versiones anteriores se manejan de manera diferente; estas se agregan a varias versiones:

- Versión del parche de la última versión. Aquí, esta es la versión 4.1.4.
- Versión del parche de la versión secundaria anterior. Aquí, esta es la versión 3.2.5.
- Versión del parche de la última versión 1.x. Aquí, esta es la versión 1.5.7.

El equipo de productos de Amazon IVS define las principales correcciones de errores. Las actualizaciones de seguridad críticas y otras correcciones seleccionadas necesarias para los clientes son ejemplos típicos.

Nota: En los ejemplos anteriores, las versiones publicadas aumentan sin omitir ningún número (por ejemplo, de 4.1.3 a 4.1.4). En realidad, uno o más números de parche pueden permanecer internos y no ser lanzados, por lo que la versión publicada podría aumentar de 4.1.3 a 4.1.6.

Además, la versión 1.x será compatible hasta finales de 2023 o cuando se lance la 3.x, que será más tarde.

API de chat de IVS para Amazon

En el lado del servidor (no gestionadas por los SDK), hay dos API, cada una con sus propias responsabilidades:

- Plano de datos: la API de mensajería de chat de IVS es una API de WebSockets diseñada para ser utilizada por aplicaciones frontales (iOS, Android, macOS, etc.) que se basan en un esquema de autenticación basado en tokens. Con un token de chat generado anteriormente, se puede conectar a salas de chat ya existentes mediante esta API.

Los SDK de mensajería del cliente de chat de Amazon IVS se refieren únicamente al plano de datos. Los SDK asumen que ya está generando tokens de chat a través de su backend. Se asume que la recuperación de estos tokens la gestiona su aplicación de front-end, no los SDK.

- Plano de control: la API de plano de control de IVS Chatproporciona una interfaz propia aplicaciones de backend para gestionar y crear salas de chat, así como los usuarios que se unen a ellas. Considérelo el panel de administración de la experiencia de chat de su aplicación, gestionado por su propio backend. Hay puntos de conexión del plano de control que son responsables de crear el token de chat que el plano de datos necesita autenticarse en una sala de chat.

  Importante: Los SDK de mensajería del cliente de chat de IVS no llaman a ningún punto de conexión del plano de control. Debe tener su backend configurado para crear sus tokens de chat. Su aplicación de interfaz debe comunicarse con su servidor para recuperar este token de chat.

SDK de mensajería del cliente de chat de Amazon IVS: guía para Android

El SDK de mensajería del cliente de chat de Amazon Interactive Video Service (IVS) proporciona interfaces que le permiten incorporar fácilmente nuestras API de mensajería de chat de IVS en plataformas que utilizan Android.

El paquete de `com.amazonaws:ivs-chat-messaging` implementa la interfaz descrita en este documento.

Versión más reciente del SDK de mensajería para clientes de Chat de IVS para Android: 1.1.0 (Notas de la versión)

Documentación de referencia: a fin de obtener información sobre los métodos más importantes disponibles en el SDK de mensajería del cliente de chat de Amazon IVS para Android, consulte la documentación de referencia en: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/

Código de muestra: consulte el repositorio de muestra de Android en GitHub: https://github.com/aws-samples/amazon-ivs-chat-for-android-demo

Requisitos de la plataforma: se requiere Android 5.0 (API nivel 21) o posterior para el desarrollo.

Introducción

Antes de comenzar, debe estar familiarizado con Primeros pasos en el chat de Amazon IVS.

Añadir el paquete

Añada `com.amazonaws:ivs-chat-messaging` a sus dependencias `build.gradle`:

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

Añadir las normas de Proguard

Añada las siguientes entradas a su archivo de reglas de R8/Proguard (`proguard-rules.pro`):

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

Configuración del backend

Esta integración requiere puntos de conexión en su servidor que se comuniquen con la API de Amazon IVS. Utilice las bibliotecas oficiales de AWS para acceder a la API de Amazon IVS desde su servidor. Se puede acceder a ellas en varios idiomas desde los paquetes públicos, por ejemplo, node.js y Java.

A continuación, cree un punto de conexión de servidor que se comunique con la API de chat de Amazon IVS y cree un token.

Configurar una conexión de servidor

Cree un método que tome `ChatTokenCallback` como parámetro y obtenga un token de chat de su backend. Pase ese token al método `onSuccess` de devolución de llamada. En caso de error, pase la excepción al método `onError` de devolución de llamada. Esto es necesario para instanciar la entidad `ChatRoom` principal en el siguiente paso.

A continuación, puede encontrar un código de ejemplo que implementa lo anterior mediante una llamada `Retrofit`.

```
// ...
```

```
private fun fetchChatToken(callback: ChatTokenCallback) {
    apiService.createChatToken(userId, roomId).enqueue(object : Callback<ChatToken> {
        override fun onResponse(call: Call<ExampleResponse>, response:
 Response<ExampleResponse>) {
            val body = response.body()
            val token = ChatToken(
                body.token,
                body.sessionExpirationTime,
                body.tokenExpirationTime
            )
            callback.onSuccess(token)
        }

        override fun onFailure(call: Call<ChatToken>, throwable: Throwable) {
            callback.onError(throwable)
        }
    })
}
// ...
```

Uso del SDK

Inicialización de una instancia de sala de chat

Cree una instancia de la clase `ChatRoom`. Esto requiere pasar `regionOrUrl`, que normalmente es la región de AWS en la que está alojada la sala de chat, y `tokenProvider`, que es el método de obtención de tokens creado en el paso anterior.

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

A continuación, cree un objeto oyente que implemente controladores para los eventos relacionados con el chat y asígnelo a la propiedad `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

// ...
```

El último paso de la inicialización básica es conectarse a la sala específica mediante el establecimiento de una conexión WebSocket. Para ello, llame al método `connect()` dentro de la instancia de la sala. Le recomendamos que lo haga en el método de ciclo de vida `onResume()` para asegurarse de que mantiene una conexión si su aplicación se reanuda desde el segundo plano.

```
room.connect()
```

El SDK intentará establecer una conexión con una sala de chat codificada en el token de chat recibido de su servidor. Si falla, intentará volver a conectarse el número de veces especificado en la instancia de la sala.

Realizar acciones en una sala de chat

La clase `ChatRoom` tiene acciones para enviar y eliminar mensajes y desconectar a otros usuarios. Estas acciones aceptan un parámetro de devolución de llamada opcional que le permite recibir notificaciones de confirmación o rechazo de solicitudes.

Envío de un mensaje

Para esta solicitud, debe tener la capacidad `SEND_MESSAGE` codificada en su token de chat.

Para activar una solicitud de envío de mensajes:

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

Para obtener una confirmación/rechazo de la solicitud, proporcione una devolución de llamada como segundo parámetro:

```
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.
   }
})
```

Eliminar mensajes

Para esta solicitud, debe tener la función DELETE_MESSAGE (eliminar mensaje) codificada en su token de chat.

Para activar una solicitud de eliminación de mensajes:

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

Para obtener una confirmación/rechazo de la solicitud, proporcione una devolución de llamada como segundo parámetro:

```
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 otro usuario

Para esta solicitud, debe tener la capacidad `DISCONNECT_USER` codificada en tu token de chat.

Para desconectar a otro usuario con fines de moderación:

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

Para obtener la confirmación/rechazo de la solicitud, proporcione una devolución de llamada como segundo parámetro:

```
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.
    }
})
```

Desconectarse de una sala de chat

Para cerrar la conexión con la sala de chat, llame al método `disconnect()` en la instancia de la sala:

```
room.disconnect()
```

Dado que la conexión WebSocket dejará de funcionar al cabo de un breve período de tiempo cuando la aplicación esté en segundo plano, le recomendamos que se conecte o desconecte manualmente

al pasar de un estado de segundo plano o a uno en segundo plano. Para ello, haga coincidir la llamada `room.connect()` en el método de ciclo de vida `onResume()`, en Android `Activity` o `Fragment`, con una llamada `room.disconnect()` en el método de ciclo de vida `onPause()`.

SDK de mensajería para clientes de Chat de Amazon IVS, parte 1 del tutorial de Android: salas de chat

Esta es la primera parte del tutorial de dos partes. Aprenderá los aspectos básicos de trabajar con el SDK de mensajería del Chat de Amazon IVS mediante la creación de una aplicación de Android totalmente funcional con el lenguaje de programación Kotlin. Denominamos a la aplicación Chatterbox.

Antes de comenzar el módulo, dedique unos minutos a familiarizarse con los requisitos previos, los conceptos clave de los tokens de chat y el servidor backend necesario para crear salas de chat.

Estos tutoriales se crearon para desarrolladores de Android con experiencia que nunca hayan utilizado el SDK de mensajería del Chat de IVS. Deberá sentirse cómodo con el lenguaje de programación Kotlin y con la creación de interfaces de usuario en la plataforma Android.

Esta primera parte del tutorial se divide en varias secciones:

1. the section called “Configuración de un servidor local de autenticación y autorización”
2. the section called “Creación de un proyecto de Chatterbox”
3. the section called “Conexión a una sala de chat y observación de actualizaciones de conexión”
4. the section called “Cree un proveedor de tokens”
5. the section called “Siguientes pasos”

Para consultar la documentación completa del SDK, comience por Amazon IVS Chat Client Messaging SDK (aquí en la Guía del usuario de Chat de Amazon IVS) y Chat Client Messaging: SDK for Android Reference (en GitHub).

Requisitos previos

- Familiarícese con Kotlin y con la creación de aplicaciones en la plataforma Android. Si no conoce bien la creación de aplicaciones para Android, consulte los conceptos básicos en la guía Cómo crear tu primera app para desarrolladores de Android.
- Lea y comprenda minuciosamente la sección Introducción al Chat de IVS.

- Cree un usuario de AWS IAM con las funcionalidades `CreateChatToken` y `CreateRoom` definidas en una política de IAM existente. (Consulte Introducción al Chat de IVS).
- Asegúrese de que las claves secretas o de acceso para este usuario estén almacenadas en un archivo de credenciales de AWS. Para obtener instrucciones, consulte la Guía del usuario de la AWS CLI (especialmente las Opciones de los archivos de configuración y credenciales).
- Cree una sala de chat y guarde su ARN. Consulte Introducción al Chat de IVS. (Si no guarda el ARN, puede buscarlo luego en la consola o en la API de chat.)

Configuración de un servidor local de autenticación y autorización

Su servidor backend se encargará tanto de crear las salas de chat como de generar los tokens de chat necesarios para que el SDK de Android del Chat de IVS autentique y autorice a sus clientes para entrar a sus salas.

Consulte Crear un token de chat en Introducción al chat de Amazon IVS. Como se muestra en ese diagrama de flujo, la aplicación del lado del servidor se encarga de crear el token de chat. Esto significa que su aplicación debe generarlo por cuenta propia al solicitarle uno a la aplicación del lado del servidor.

Utilizamos el marco de Ktor para crear un servidor local activo que administre la creación de tokens de chat con el entorno local de AWS.

En este momento, se espera que ya haya configurado correctamente sus credenciales de AWS. Para ver instrucciones paso a paso, consulte Configuración de credenciales y regiones para desarrollo de AWS.

Cree un directorio nuevo y póngale el nombre `chatterbox` y dentro de él, otro, llamado `auth-server`.

Nuestra carpeta de servidor tendrá la siguiente estructura:

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

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

Nota: Puede copiar y pegar directamente el código aquí en los archivos de referencia.

Luego, agregamos todas las dependencias y complementos necesarios para que nuestro servidor de autenticación 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")
}
```

Ahora debemos configurar la funcionalidad de registro para el servidor de autenticación. (Para obtener más información, consulte Configuración de los registradores).

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>
```

El servidor Ktor requiere ajustes de configuración, que se cargan automáticamente desde el archivo `application.*` del directorio `resources`, por lo que también los agregamos. (Para obtener más información sobre los archivos de configuración, consulte Configuración en un archivo).

HOCON:

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

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

Por último, implementemos nuestro 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()
            )
         )
      }
   }
}
```

Creación de un proyecto de Chatterbox

Para crear un proyecto de Android, instale y abra Android Studio.

Siga los pasos que se indican en la guía oficial de Creación de un proyecto de Android.

- En Elegir tipo de proyecto, elija la plantilla de proyecto Actividad vacía para nuestra aplicación Chatterbox.
- En Configure su proyecto, elija los siguientes valores para los campos de configuración:
  - Nombre: My App
  - Nombre del paquete: com.chatterbox.myapp
  - Ubicación de guardado: elija la ruta al directorio `chatterbox` creado en el paso anterior
  - Idioma: Kotlin
  - Nivel de API mínimo: API 21: Android 5.0 (Lollipop)

Después de especificar todos los parámetros de configuración correctamente, nuestra estructura de archivos dentro de la carpeta `chatterbox` debería verse así:

```
- 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
```

Ahora que tenemos un proyecto de Android en funcionamiento, podemos agregar com.amazonaws:ivs-chat-messaging a nuestras dependencias `build.gradle`. (Para obtener más información sobre el kit de herramientas de compilación de Gradle, consulte Cómo configurar tu compilación).

Nota: En la parte superior de cada fragmento de código, hay una ruta al archivo donde debería realizar cambios en su proyecto. La ruta depende de la raíz del proyecto.

En el código siguiente, sustituya *<version>* por el número de la versión actual del SDK del chat para Android (p. ej., 1.0.0).

Kotlin:

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

plugins {
// ...
}

android {
// ...
}

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

Después de agregar la nueva dependencia, ejecute Sincronizar proyecto con archivos de Gradle en Android Studio para sincronizar el proyecto con la nueva dependencia. (Para obtener más información, consulte Agregar dependencias de compilación).

Para ejecutar nuestro servidor de autenticación de forma simple (creado en la sección anterior) desde la raíz del proyecto, lo incluimos como un módulo nuevo en `settings.gradle`. (Para obtener más información, consulte Estructuración y creación de un componente de software con Gradle).

Script de Kotlin:

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

```
// ...

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

A partir de ahora, como `auth-server` se incluye en el proyecto de Android, puede ejecutar el servidor de autenticación con el siguiente comando desde la raíz del proyecto:

Intérprete de comandos:

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

Conexión a una sala de chat y observación de actualizaciones de conexión

Para abrir una conexión a una sala de chat, utilizamos la Devolución de llamada del ciclo de vida de la actividad onCreate(), que se activa cuando la actividad se crea por primera vez. El constructor de ChatRoom requiere que proporcionemos `region` y `tokenProvider` para instanciar una conexión de sala.

Nota: La función `fetchChatToken` del siguiente fragmento se implementará en la siguiente sección.

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)
    }

// ...
}
```

Mostrar y reaccionar a los cambios en el estado de conexión de la sala de chat es una parte esencial en la creación de una aplicación de chat como `chatterbox`. Antes de poder empezar a interactuar con la sala, debemos suscribirnos a los eventos del estado de conexión de la sala de chat para recibir actualizaciones.

ChatRoom espera que adjuntemos una implementación de la interfaz ChatRoomListener para aumentar los eventos del ciclo de vida. Por ahora, las funciones del oyente realizan un registro solo de los mensajes de confirmación cuando se invoquen:

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")
        }
    }
}
```

Ahora que hemos implementado `ChatRoomListener`, lo adjuntamos a nuestra instancia de 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 {
// ...
}
```

Luego, debemos establecer la capacidad de leer el estado de la conexión. Lo mantendremos en la propiedad `MainActivity.kt` y lo inicializaremos en el estado DESCONECTADO predeterminado para las salas (consulte `ChatRoom state` en la referencia del SDK de Android para el Chat de IVS). Para poder mantener actualizado el estado local, necesitamos implementar una función de actualización de estados; llamémosla `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")
         }
      }
   }
}
```

A continuación, integramos nuestra función de actualización de estados con la propiedad 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)
         }
      }
   }
}
```

Ahora que podemos guardar y escuchar las actualizaciones del estado de ChatRoom, y reaccionar a ellas, es el momento de inicializar la conexión:

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

Cree un proveedor de tokens

Es hora de crear una función que se encargue de generar y gestionar los tokens de chat en nuestra aplicación. En este ejemplo utilizamos el cliente HTTP Retrofit para Android.

Antes de poder enviar cualquier tráfico de red, debemos definir una configuración de seguridad de red para Android. (Para obtener más información, consulte Configuraciones de seguridad de la red). Empezamos por agregar permisos de red al archivo App Manifest. Tenga en cuenta la etiqueta `user-permission` y el atributo `networkSecurityConfig` agregados, que apuntarán a nuestra nueva configuración de seguridad de red. En el código siguiente, sustituya *<version>* por el número de la versión actual del SDK del chat para Android (p. ej., 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 los dominios `10.0.2.2` y `localhost` como de confianza para empezar a intercambiar mensajes con nuestro 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>
```

Luego, debemos agregar una dependencia nueva, junto con la adición del convertidor Gson para analizar las respuestas HTTP. En el código siguiente, sustituya `<version>` por el número de la versión actual del SDK del chat para Android (p. ej., 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 un token de chat, necesitamos realizar una solicitud HTTP POST desde nuestra aplicación `chatterbox`. Definimos la solicitud en una interfaz para que Retrofit la implemente. (Consulte la documentación de Retrofit. Además, familiarícese con la especificación de punto de conexión 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>
}
```

Ahora, con la red configurada, es el momento de agregar una función que se encargue de crear y administrar nuestro token de chat. Lo agregamos a `MainActivity.kt`, que se creó automáticamente cuando se [generó](#) el proyecto:

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)
            }
        })
    }
}
```

Siguientes pasos

Ahora que estableció la conexión a la sala de chat, continúe con la parte 2 de este tutorial de Android, Mensajes y eventos.

SDK de mensajería para clientes de Chat de Amazon IVS, parte 2 del tutorial de Android: mensajes y eventos

Esta segunda (y última) parte del tutorial se divide en varias secciones:

1. the section called “Creación de una interfaz de usuario para enviar mensajes”
   a. the section called “Diseño principal de la interfaz de usuario”

   b. the section called “Celda de texto abstracto de la interfaz de usuario para mostrar texto de forma coherente”
   c. the section called “Mensaje de chat izquierdo de la interfaz de usuario”
   d. the section called “Mensaje de chat derecho de la interfaz de usuario”
   e. the section called “Valores de color adicionales de la interfaz de usuario”
2. the section called “Aplicación de la vinculación de vista”
3. the section called “Administrar solicitudes de mensajes de chat”
4. the section called “Pasos finales”

Para consultar la documentación completa del SDK, comience por Amazon IVS Chat Client Messaging SDK (aquí en la Guía del usuario de Chat de Amazon IVS) y Chat Client Messaging: SDK for Android Reference (en GitHub).

Requisito previo

Asegúrese de haber completado la parte 1 de este tutorial: salas de chat.

Creación de una interfaz de usuario para enviar mensajes

Ahora que hemos iniciado correctamente la conexión a la sala de chat, es el momento de enviar nuestro primer mensaje. Para esta función, se necesita una interfaz de usuario. Agregaremos:

- Botón `connect/disconnect`
- Entrada de mensajes con botón `send`
- Lista dinámica de mensajes. Para crear esto, utilizamos RecyclerView de Android Jetpack.

Diseño principal de la interfaz de usuario

Consulte los Diseños de Android Jetpack en la documentación para desarrolladores de 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>
```

Celda de texto abstracto de la interfaz de usuario para mostrar texto de forma coherente

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>
```

Mensaje de chat izquierdo de la interfaz de usuario

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>
```

Mensaje de chat derecho de la interfaz de usuario

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 color adicionales de la interfaz de usuario

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>
```

Aplicación de la vinculación de vista

Aprovechamos la función Vinculación de vista de Android para poder hacer referencia a las clases de enlace para nuestro diseño XML. Para habilitar la función, defina la opción de compilación `viewBinding` a `true` en `./app/build.gradle`:

Script de Kotlin:

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

```
android {
//    ...

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

Ahora es el momento de conectar la interfaz de usuario con nuestro código de 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)
    }
}
```

También agregamos métodos para eliminar mensajes y desconectar a los usuarios del chat, que se pueden invocar con el menú contextual de mensajes 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}")
                    }
                }
            }
        )
    }
}
```

Administrar solicitudes de mensajes de chat

Necesitamos una forma de gestionar nuestras solicitudes de mensajes de chat en todos sus estados posibles:

- Pendiente: se envió un mensaje a una sala de chat, pero aún no se ha confirmado ni se ha rechazado.
- Confirmado: la sala de chat envió un mensaje a todos los usuarios (incluidos nosotros).
- Rechazado: la sala de chat rechazó un mensaje con un objeto de error.

Mantendremos las solicitudes de chat y los mensajes de chat sin resolver en una lista. La lista requiere una clase aparte, que llamamos `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 nuestra lista con la interfaz de usuario, utilizamos un Adaptador. Para obtener más información, consulte Vinculación a datos con AdapterView y Clases de vinculación generadas.

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
}
```

Pasos finales

Es hora de conectar nuestro adaptador nuevo, al vincular la clase `ChatEntries` a `MainActivity`:

Kotlin:

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

package com.chatterbox.myapp
// ...

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

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

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

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

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

        setUpChatView()

        updateConnectionState(ConnectionState.DISCONNECTED)
    }

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

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

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

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

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

Como ya tenemos una clase que se encarga de llevar un registro de nuestras solicitudes de chat (`ChatEntries`), estamos listos para implementar el código para manipular `entries` en `roomListener`. Actualizaremos `entries` y `connectionState` en concordancia con el evento al que respondemos:

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")
        }
    }
}
```

¡Ahora debería poder ejecutar su aplicación! (Consulte Cree y ejecute su aplicación). Recuerde tener su servidor backend en ejecución cuando utilice la aplicación. Puede activarlo desde la terminal que está en la raíz de nuestro proyecto con este comando: `./gradlew :auth-server:run` o si ejecuta la tarea de Gradle `auth-server:run` directamente desde Android Studio.

SDK de mensajería para clientes de Chat de Amazon IVS, parte 1 del tutorial de las corrutinas de Kotlin: salas de chat

Esta es la primera parte del tutorial de dos partes. Conocerá los aspectos básicos de trabajar con el SDK de mensajería para el Chat de Amazon IVS cuando cree una aplicación de Android totalmente funcional con el lenguaje de programación y las corrutinas de Kotlin. Denominamos a la aplicación Chatterbox.

Antes de comenzar el módulo, dedique unos minutos a familiarizarse con los requisitos previos, los conceptos clave de los tokens de chat y el servidor backend necesario para crear salas de chat.

Estos tutoriales se crearon para desarrolladores de Android con experiencia que nunca hayan utilizado el SDK de mensajería del Chat de IVS. Deberá sentirse cómodo con el lenguaje de programación Kotlin y con la creación de interfaces de usuario en la plataforma Android.

Esta primera parte del tutorial se divide en varias secciones:

1. the section called “Configuración de un servidor local de autenticación y autorización”
2. the section called “Creación de un proyecto de Chatterbox”
3. the section called “Conexión a una sala de chat y observación de actualizaciones de conexión”
4. the section called “Cree un proveedor de tokens”
5. the section called “Siguientes pasos”

Para consultar la documentación completa del SDK, comience por Amazon IVS Chat Client Messaging SDK (aquí en la Guía del usuario de Chat de Amazon IVS) y Chat Client Messaging: SDK for Android Reference (en GitHub).

Requisitos previos

- Familiarícese con Kotlin y con la creación de aplicaciones en la plataforma Android. Si no conoce bien la creación de aplicaciones para Android, consulte los conceptos básicos en la guía Cómo crear tu primera app para desarrolladores de Android.

- Lea y conozca Introducción al Chat de IVS.
- Cree un usuario de AWS IAM con las funcionalidades `CreateChatToken` y `CreateRoom` definidas en una política de IAM existente. (Consulte Introducción al Chat de IVS).
- Asegúrese de que las claves secretas o de acceso para este usuario estén almacenadas en un archivo de credenciales de AWS. Para obtener instrucciones, consulte la Guía del usuario de la AWS CLI (especialmente las Opciones de los archivos de configuración y credenciales).
- Cree una sala de chat y guarde su ARN. Consulte Introducción al Chat de IVS. (Si no guarda el ARN, puede buscarlo luego en la consola o en la API de chat.)

Configuración de un servidor local de autenticación y autorización

Su servidor backend se encargará tanto de crear las salas de chat como de generar los tokens de chat necesarios para que el SDK de Android del Chat de IVS autentique y autorice a sus clientes para entrar a sus salas.

Consulte Crear un token de chat en Introducción al chat de Amazon IVS. Como se muestra en ese diagrama de flujo, la aplicación del lado del servidor se encarga de crear el token de chat. Esto significa que su aplicación debe generarlo por cuenta propia al solicitarle uno a la aplicación del lado del servidor.

Utilizamos el marco de Ktor para crear un servidor local activo que administre la creación de tokens de chat con el entorno local de AWS.

En este momento, se espera que ya haya configurado correctamente sus credenciales de AWS. Para ver instrucciones paso a paso, consulte Configurar credenciales temporales de AWS y región de AWS para desarrollo.

Cree un directorio nuevo y póngale el nombre `chatterbox` y dentro de él, otro, llamado `auth-server`.

Nuestra carpeta de servidor tendrá la siguiente estructura:

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

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

Nota: Puede copiar y pegar directamente el código aquí en los archivos de referencia.

Luego, agregamos todas las dependencias y complementos necesarios para que nuestro servidor de autenticación 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")
}
```

Ahora debemos configurar la funcionalidad de registro para el servidor de autenticación. (Para obtener más información, consulte Configuración de los registradores).

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>
```

El servidor Ktor requiere ajustes de configuración, que se cargan automáticamente desde el archivo `application.*` del directorio `resources`, por lo que también los agregamos. (Para obtener más información sobre los archivos de configuración, consulte Configuración en un archivo).

HOCON:

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

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

Por último, implementemos nuestro 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()
            )
         )
      }
   }
}
```

Creación de un proyecto de Chatterbox

Para crear un proyecto de Android, instale y abra Android Studio.

Siga los pasos que se indican en la guía oficial de Creación de un proyecto de Android.

- En Elija su proyecto, elija la plantilla de proyecto Actividad vacía para nuestra aplicación Chatterbox.

- En Configure su proyecto, elija los siguientes valores para los campos de configuración:
  - Nombre: My App
  - Nombre del paquete: com.chatterbox.myapp
  - Ubicación de guardado: elija la ruta al directorio `chatterbox` creado en el paso anterior
  - Idioma: Kotlin
  - Nivel de API mínimo: API 21: Android 5.0 (Lollipop)

Después de especificar todos los parámetros de configuración correctamente, nuestra estructura de archivos dentro de la carpeta `chatterbox` debería verse así:

```
- 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
```

Ahora que tenemos un proyecto de Android en funcionamiento, podemos agregar com.amazonaws:ivs-chat-messaging y org.jetbrains.kotlinx:kotlinx-coroutine-core a nuestras dependencias `build.gradle`. (Para obtener más información sobre el kit de herramientas de compilación de Gradle, consulte Cómo configurar tu compilación).

Nota: En la parte superior de cada fragmento de código, hay una ruta al archivo donde debería realizar cambios en su proyecto. La ruta depende de la raíz del proyecto.

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'

// ...
}
```

Después de agregar la nueva dependencia, ejecute Sincronizar proyecto con archivos de Gradle en Android Studio para sincronizar el proyecto con la nueva dependencia. (Para obtener más información, consulte Agregar dependencias de compilación).

Para ejecutar nuestro servidor de autenticación de forma simple (creado en la sección anterior) desde la raíz del proyecto, lo incluimos como un módulo nuevo en `settings.gradle`. (Para obtener más información, consulte Estructuración y creación de un componente de software con Gradle).

Script de Kotlin:

```
// ./settings.gradle

// ...

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

A partir de ahora, como `auth-server` se incluye en el proyecto de Android, puede ejecutar el servidor de autenticación con el siguiente comando desde la raíz del proyecto:

Intérprete de comandos:

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

Conexión a una sala de chat y observación de actualizaciones de conexión

Para abrir una conexión a una sala de chat, utilizamos la Devolución de llamada del ciclo de vida de la actividad onCreate(), que se activa cuando la actividad se crea por primera vez. El constructor de ChatRoom requiere que proporcionemos `region` y `tokenProvider` para instanciar una conexión de sala.

Nota: La función `fetchChatToken` del siguiente fragmento se implementará en la siguiente sección.

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)
    }

// ...
}
```

Mostrar y reaccionar a los cambios en el estado de conexión de la sala de chat es una parte esencial en la creación de una aplicación de chat como `chatterbox`. Antes de poder empezar a interactuar con la sala, debemos suscribirnos a los eventos del estado de conexión de la sala de chat para recibir actualizaciones.

En el SDK de chat para corrutinas, `ChatRoom` espera que gestionemos los eventos del ciclo de vida de la sala en Flujo. Por ahora, las funciones realizarán un registro solo de los mensajes de confirmación cuando se invoquen:

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")
                }
            }
        }
    }
}
```

Luego, debemos establecer la capacidad de leer el estado de la conexión. Lo mantendremos en la propiedad `MainActivity.kt` y lo inicializaremos en el estado DESCONECTADO predeterminado para las salas (consulte `ChatRoom state` en la referencia del SDK de Android para el Chat de IVS). Para poder mantener actualizado el estado local, necesitamos implementar una función de actualización de estados; llamémosla `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")
        }
    }
}
```

A continuación, integramos nuestra función de actualización de estados con la propiedad 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)

                }
            }

      // ...

      }
```

```
    }
}
```

Ahora que podemos guardar y escuchar las actualizaciones del estado de ChatRoom, y reaccionar a ellas, es el momento de inicializar la conexión:

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)
      }
   }

   // ...
}
```

Cree un proveedor de tokens

Es hora de crear una función que se encargue de generar y gestionar los tokens de chat en nuestra aplicación. En este ejemplo utilizamos el cliente HTTP Retrofit para Android.

Antes de poder enviar cualquier tráfico de red, debemos definir una configuración de seguridad de red para Android. (Para obtener más información, consulte Configuraciones de seguridad de la red). Empezamos por agregar permisos de red al archivo App Manifest. Tenga en cuenta la etiqueta `user-permission` y el atributo `networkSecurityConfig` agregados, que apuntarán a nuestra nueva configuración de seguridad de red. En el código siguiente, sustituya *<version>* por el número de la versión actual del SDK del chat para Android (p. ej., 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 su dirección IP local, por ejemplo los dominios `10.0.2.2` y `localhost`, como de confianza para empezar a intercambiar mensajes con nuestro 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>
```

Luego, debemos agregar una dependencia nueva, junto con la adición del convertidor Gson para analizar las respuestas HTTP. En el código siguiente, sustituya *`<version>`* por el número de la versión actual del SDK del chat para Android (p. ej., 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 un token de chat, necesitamos realizar una solicitud HTTP POST desde nuestra aplicación `chatterbox`. Definimos la solicitud en una interfaz para que Retrofit la implemente. (Consulte la documentación de Retrofit. Además, familiarícese con la especificación de punto de conexión 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)
   }
}
```

Ahora, con la red configurada, es el momento de agregar una función que se encargue de crear y administrar nuestro token de chat. Lo agregamos a `MainActivity.kt`, que se creó automáticamente cuando se generó el proyecto:

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)
         }
      })
   }
}
```

Siguientes pasos

Ahora que estableció una conexión a la sala de chat, continúe con la parte 2 de este tutorial de corrutinas de Kotlin, Mensajes y eventos.

SDK de mensajería para clientes de Chat de Amazon IVS, parte 2 del tutorial de corrutinas de Kotlin: mensajes y eventos

Esta segunda (y última) parte del tutorial se divide en varias secciones:

1. the section called “Creación de una interfaz de usuario para enviar mensajes”
   a. the section called “Diseño principal de la interfaz de usuario”
   b. the section called “Celda de texto abstracto de la interfaz de usuario para mostrar texto de forma coherente”
   c. the section called “Mensaje de chat izquierdo de la interfaz de usuario”
   d. the section called “Mensaje derecho de la interfaz de usuario”
   e. the section called “Valores de color adicionales de la interfaz de usuario”
2. the section called “Aplicación de la vinculación de vista”
3. the section called “Administrar solicitudes de mensajes de chat”
4. the section called “Pasos finales”

Para consultar la documentación completa del SDK, comience por Amazon IVS Chat Client Messaging SDK (aquí en la Guía del usuario de Chat de Amazon IVS) y Chat Client Messaging: SDK for Android Reference (en GitHub).

Requisito previo

Asegúrese de haber completado la parte 1 de este tutorial: salas de chat.

Creación de una interfaz de usuario para enviar mensajes

Ahora que hemos iniciado correctamente la conexión a la sala de chat, es el momento de enviar nuestro primer mensaje. Para esta función, se necesita una interfaz de usuario. Agregaremos:

- Botón `connect` o `disconnect`
- Entrada de mensajes con botón `send`
- Lista dinámica de mensajes. Para crear esto, utilizamos RecyclerView de Android Jetpack.

Diseño principal de la interfaz de usuario

Consulte los Diseños de Android Jetpack en la documentación para desarrolladores de 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>
```

Celda de texto abstracto de la interfaz de usuario para mostrar texto de forma coherente

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>
```

Mensaje de chat izquierdo de la interfaz de usuario

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>
```

Mensaje derecho de la interfaz de usuario

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 color adicionales de la interfaz de usuario

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>
```

Aplicación de la vinculación de vista

Aprovechamos la función Vinculación de vista de Android para poder hacer referencia a las clases de enlace para nuestro diseño XML. Para habilitar la función, defina la opción de compilación `viewBinding` a `true` en `./app/build.gradle`:

Script de Kotlin:

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

android {
//    ...

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

Ahora es el momento de conectar la interfaz de usuario con nuestro código de 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)
    }
// ...

}
```

También agregamos métodos para eliminar mensajes y desconectar a los usuarios del chat, que se pueden invocar con el menú contextual de mensajes 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")
           }
        }
    }
}
```

Administrar solicitudes de mensajes de chat

Necesitamos una forma de gestionar nuestras solicitudes de mensajes de chat en todos sus estados posibles:

- Pendiente: se envió un mensaje a una sala de chat, pero aún no se ha confirmado ni se ha rechazado.
- Confirmado: la sala de chat envió un mensaje a todos los usuarios (incluidos nosotros).
- Rechazado: la sala de chat rechazó un mensaje con un objeto de error.

Mantendremos las solicitudes de chat y los mensajes de chat sin resolver en una lista. La lista requiere una clase aparte, que llamamos `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 nuestra lista con la interfaz de usuario, utilizamos un Adaptador. Para obtener más información, consulte Vinculación a datos con AdapterView y Clases de vinculación generadas.

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
}
```

Pasos finales

Es hora de conectar nuestro adaptador nuevo, al vincular la clase `ChatEntries` a `MainActivity`:

Kotlin:

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

package com.chatterbox.myapp
// ...

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

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

    // ...

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

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

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

            sendButtonClick(binding.sendButton)
```

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

Como ya tenemos una clase que se encarga de llevar un registro de nuestras solicitudes de chat (`ChatEntries`), estamos listos para implementar el código para manipular `entries` en roomListener. Actualizaremos `entries` y `connectionState` en concordancia con el evento al que respondemos:

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)
    }

// ...

}
```

¡Ahora debería poder ejecutar su aplicación! (Consulte Cree y ejecute su aplicación). Recuerde tener su servidor backend en ejecución cuando utilice la aplicación. Puede activarlo desde la terminal que está en la raíz de nuestro proyecto con este comando: `./gradlew :auth-server:run` o si ejecuta la tarea de Gradle `auth-server:run` directamente desde Android Studio.

SDK de mensajería del cliente de chat de Amazon IVS: guía para iOS

El SDK de mensajería del cliente de chat de Amazon Interactive Video (IVS) para iOS proporciona interfaces que permiten incorporar nuestras API de mensajería de chat de IVS en plataformas que utilizan el lenguaje de programación Swift de Apple.

Versión más reciente del SDK de mensajería del cliente de chat de Amazon IVS para iOS: 1.0.0 (Notas de la versión)

Documentación de referencia y tutoriales: para obtener información sobre los métodos más importantes disponibles en el SDK de mensajería del cliente de chat de Amazon IVS para iOS, consulte la documentación de referencia en: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/. Este repositorio también contiene varios artículos y tutoriales.

Código de muestra: consulte el repositorio de muestra de iOS en GitHub: https://github.com/aws-samples/amazon-ivs-chat-for-ios-demo.

Requisitos de la plataforma: se requiere iOS 13.0 o posterior para el desarrollo.

Introducción

Recomendamos que integre el SDK a través de Swift Package Manager. También puede utilizar CocoaPods o integrar el marco de forma manual.

Tras integrar el SDK, puede importarlo añadiendo el siguiente código en la parte superior del archivo Swift correspondiente:

```
import AmazonIVSChatMessaging
```

Swift Package Manager

Para utilizar la biblioteca `AmazonIVSChatMessaging` en un proyecto de Swift Package Manager, agréguela a las dependencias de su paquete y a las dependencias de sus objetivos relevantes:

1. Descargue la versión más reciente de `.xcframework` en https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. En su terminal, ejecute:

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

3. Tome el resultado del paso anterior y péguelo en la propiedad checksum de `.binaryTarget` como se muestra a continuación dentro de su archivo `Package.swift` de proyecto:

```
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

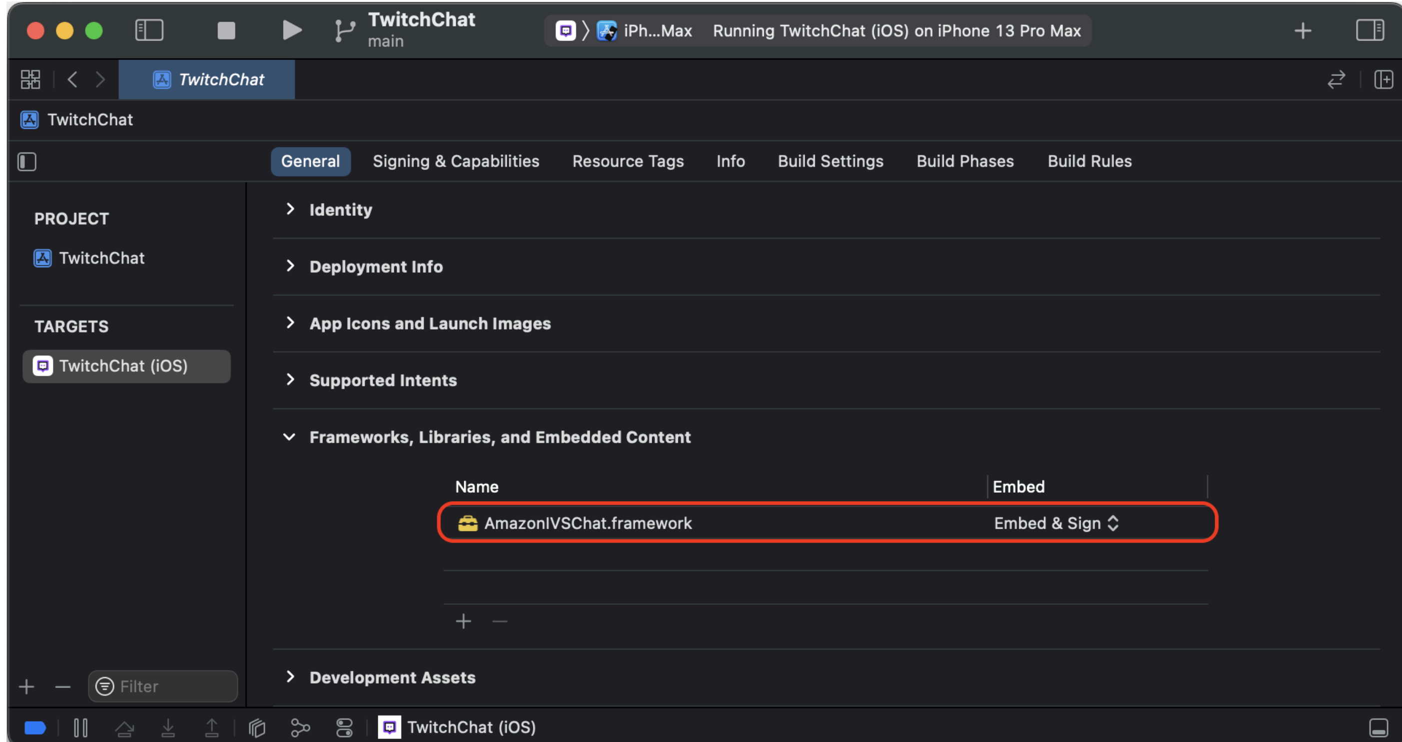
Las versiones se publican a través de CocoaPods bajo el nombre `AmazonIVSChatMessaging`. Agregue esta dependencia a su Podfile:

```
pod 'AmazonIVSChat'
```

Ejecute `pod install` y el SDK estará disponible en su `.xcworkspace`.

Instalación manual

1. Descargue la versión más reciente desde https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. Extraiga el contenido del archivo. `AmazonIVSChatMessaging.xcframework` contiene el SDK para el dispositivo y el simulador.
3. Integre el `AmazonIVSChatMessaging.xcframework` arrastrándolo a la sección Frameworks, Libraries, and Embedded Content (Marcos, librerías y contenido integrado) de la pestaña General para el destino de la aplicación:



Uso del SDK

Conectarse a una sala de chat

Antes de comenzar, debe estar familiarizado con los Primeros pasos en el chat de Amazon IVS. Consulte también las aplicaciones de ejemplo para Web, Android e iOS.

Para conectarse a una sala de chat, su aplicación necesita alguna forma de recuperar un token de chat proporcionado por su backend. Es probable que su aplicación recupere un token de chat mediante una solicitud de red a su backend.

Para comunicar este token de chat obtenido con el SDK, el modelo `ChatRoom` del SDK requiere que proporcione una función `async` o una instancia de un objeto que se ajuste al protocolo `ChatTokenProvider` proporcionado en el punto de inicialización. El valor devuelto por cualquiera de estos métodos debe ser una instancia del modelo `ChatToken` de SDK.

Nota: Rellene las instancias del modelo `ChatToken` usando datos recuperados de su backend. Los campos necesarios para inicializar una instancia `ChatToken` son los mismos que los campos de la respuesta de CreateChatToken. Para obtener más información sobre la inicialización de instancias de modelo `ChatToken`, consulte Creación de una instancia de ChatToken. Recuerde, su backend es responsable de proporcionar los datos en la respuesta `CreateChatToken` a su aplicación. La forma en que decida comunicarse con su backend para generar tokens de chat depende de su aplicación y su infraestructura.

Después de elegir su estrategia para ofrecer un `ChatToken` al SDK, llame a `.connect()` después de inicializar correctamente una instancia `ChatRoom` con su proveedor de tokens y la Región de AWS que su backend usó para crear la sala de chat a la que está intentando conectarse. Tenga en cuenta que `.connect()` es una función asincrónica de lanzamiento:

```
import AmazonIVSChatMessaging

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

Conformidad con el protocolo ChatTokenProvider

Para el registro del parámetro `tokenProvider` en el inicializador de `ChatRoom`, puede proporcionar una instancia de `ChatTokenProvider`. Este es un ejemplo de un objeto que se ajusta a `ChatTokenProvider`:

```
import AmazonIVSChatMessaging

// This object should exist somewhere in your app
class ChatService: ChatTokenProvider {
   func getChatToken() async throws -> ChatToken {
      let request = YourApp.getTokenURLRequest
      let data = try await URLSession.shared.data(for: request).0
      ...
      return ChatToken(
```

```
            token: String(data: data, using: .utf8)!,
            tokenExpirationTime: ..., // this is optional
            sessionExpirationTime: ... // this is optional
        )
    }
}
```

A continuación, puede tomar una instancia de este objeto conforme y pasarla al inicializador de `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()
```

Proporcionar una función asíncrona en Swift

Supongamos que ya tiene un administrador que utiliza para gestionar las solicitudes de red de la aplicación. Podría ser como el siguiente:

```
import AmazonIVSChatMessaging

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

Puede añadir otra función en su administrador para recuperar un `ChatToken` desde su backend:

```
import AmazonIVSChatMessaging

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

Luego, use la referencia a esa función en Swift al inicializar un `ChatRoom`:

```
import AmazonIVSChatMessaging

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

Creación de una instancia de ChatToken

Puede crear fácilmente una instancia de `ChatToken` mediante el inicializador incluido en el SDK. Consulte la documentación en `Token.swift` para obtener más información acerca de las propiedades de `ChatToken`.

```
import AmazonIVSChatMessaging

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

Usar Decodible

Si, mientras interactúa con la API de chat de IVS, su backend decide simplemente reenviar la respuesta CreateChatToken a su aplicación de frontend, puede aprovechar la conformidad de `ChatToken` para el protocolo `Decodable` de Swift. Sin embargo, hay un problema.

La carga útil de las respuestas `CreateChatToken` usa cadenas para fechas formateadas con la Norma ISO 8601 para marcas temporales de Internet. Normalmente, en Swift se proporcionaría `JSONDecoder.DateDecodingStrategy.iso8601` como un valor para la propiedad `.dateDecodingStrategy` de `JSONDecoder`. Sin embargo, `CreateChatToken` utiliza segundos fraccionarios de alta precisión en sus cadenas y esto no es compatible con `JSONDecoder.DateDecodingStrategy.iso8601`.

Para su comodidad, el SDK proporciona una extensión pública en `JSONDecoder.DateDecodingStrategy` con una estrategia `.preciseISO8601` adicional que le permite utilizar `JSONDecoder` con éxito al decodificar una instancia 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 una sala de chat

Para desconectarse manualmente de una instancia `ChatRoom` a la que se conectó correctamente, llame a `room.disconnect()`. De forma predeterminada, las salas de chat invocan automáticamente esta función cuando no están asignadas.

```
import AmazonIVSChatMessaging

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

// Disconnect
room.disconnect()
```

Recibir un mensaje/evento de chat

Para enviar y recibir mensajes en su sala de chat, debe proporcionar un objeto que se ajuste al protocolo `ChatRoomDelegate`, después de inicializar correctamente una instancia de `ChatRoom` y llamar a `room.connect()`. A continuación, se muestra un ejemplo típico en el que se utiliza `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) { ... }
}
```

Recibir notificaciones al cambiar la conexión

Como es de esperar, no puede realizar acciones como enviar un mensaje en una sala hasta que la sala esté completamente conectada. La arquitectura del SDK intenta promover la conexión a un objeto ChatRoom en un subproceso en segundo plano a través de API asíncronas. En caso de que quiera crear algo en su interfaz de usuario que desactive algo como un botón de envío de mensajes, el SDK proporciona dos estrategias para recibir notificaciones cuando cambie el estado de conexión de una sala de chat: usando `Combine` o `ChatRoomDelegate`. Estas se describen a continuación.

Importante: el estado de conexión de una sala de chat también puede cambiar debido a cosas como la caída de la conexión de red. Tenga esto en cuenta al crear su aplicación.

Usar Combine

Cada instancia de `ChatRoom` viene con su propio editor `Combine` en forma de propiedad `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, utilice las funciones opcionales `roomDidConnect(_:)`, `roomIsConnecting(_:)` y `roomDidDisconnect(_:)` dentro de un objeto que se ajuste a `ChatRoomDelegate`. A continuación se muestra un ejemplo usando un `UIViewController`:

```
import AmazonIVSChatMessaging
import Foundation
import UIKit

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

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

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

```
    }
}

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

Realizar acciones en una sala de chat

Los diferentes usuarios tienen diferentes capacidades para las acciones que pueden realizar en una sala de chat; por ejemplo, enviar/eliminar un mensaje o desconectar a un usuario. Para realizar una de estas acciones, llame a `perform(request:)` en un `ChatRoom` conectado pasando una instancia de uno de los objetos `ChatRequest` proporcionados en el SDK. Las solicitudes admitidas están en `Request.swift`.

Algunas acciones que se realizan en una sala de chat requieren que los usuarios conectados cuenten con capacidades específicas cuando su aplicación backend llama a `CreateChatToken`. Por diseño, el SDK no puede distinguir las capacidades de un usuario conectado. Por lo tanto, si bien puede intentar realizar acciones de moderador en una instancia conectada de `ChatRoom`, la API del plano de control decide en última instancia si esa acción tendrá éxito.

Todas las acciones que se realizan a través de `room.perform(request:)` esperan hasta que la sala reciba la instancia esperada de un modelo (cuyo tipo está asociado al propio objeto de solicitud) que coincida con el `requestId`, tanto del modelo recibido como del objeto solicitado. Si hay algún problema con la solicitud, `ChatRoom` siempre arroja un error en forma de un `ChatError`. La definición de `ChatError` está en `Error.swift`.

Envío de un mensaje

Para enviar un mensaje de chat, use una instancia de `SendMessageRequest`:

```
import AmazonIVSChatMessaging

```

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

Como se ha mencionado anteriormente, `room.perform(request:)` devuelve una vez un `ChatMessage` correspondiente recibido por `ChatRoom`. Si hay algún problema con la solicitud (por ejemplo, se supera el límite de caracteres del mensaje para una sala), se lanza una instancia de `ChatError` en su lugar. A continuación, puede mostrar esta información útil en su interfaz de usuario:

```
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)
}
```

Adjuntar metadatos a un mensaje

Cuando envía de un mensaje, puede añadir los metadatos que se asociarán a él. `SendMessageRequest` tiene una propiedad `attributes` con la que puede inicializar su solicitud. Los datos que adjunte allí se adjuntan al mensaje cuando otras personas reciben ese mensaje en la sala.

A continuación, se muestra un ejemplo de cómo adjuntar datos de emoticonos a un mensaje que se envía:

```
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"
      ]
   )
)
```

El uso de `attributes` en un `SendMessageRequest` puede ser extremadamente útil para crear funciones complejas en su producto de chat. Por ejemplo, ¡se podría crear una funcionalidad de subprocesos mediante el diccionario de atributos `[String : String]` en un `SendMessageRequest`!

La carga útil `attributes` es muy flexible y potente. Úsela para obtener información sobre su mensaje que no podría obtener de otro modo. Usar atributos es mucho más fácil que, por ejemplo, analizar la cadena de un mensaje para obtener información sobre cosas como los emoticonos.

Eliminar mensajes

Eliminar un mensaje de chat es como enviar uno. Utilice la función `room.perform(request:)` de `ChatRoom` para lograrlo mediante la creación de una instancia de `DeleteMessageRequest`.

Para acceder fácilmente a las instancias anteriores de los mensajes de Chat recibidos, transfiera el valor de `message.id` al inicializador de `DeleteMessageRequest`.

Si lo desea, proporcione una cadena de motivos a `DeleteMessageRequest` para que puedas mostrarlo en tu interfaz de usuario.

```
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 se trata de una acción de moderador, es posible que el usuario no tenga realmente la capacidad de eliminar el mensaje de otro usuario. Puede usar la mecánica de funciones de lanzamiento de Swift para mostrar un mensaje de error en su interfaz de usuario cuando un usuario intenta eliminar un mensaje sin la capacidad adecuada.

Cuando el backend llama a `CreateChatToken` para un usuario, debe pasar `"DELETE_MESSAGE"` al campo `capabilities` para activar esa funcionalidad para un usuario de chat conectado.

A continuación, se muestra un ejemplo de cómo se detecta un error de capacidad que se produce al intentar eliminar un mensaje sin los permisos adecuados:

```
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 otro usuario

Utilizar `room.perform(request:)` para desconectar a otro usuario de una sala de chat. Concretamente, utilice una instancia de `DisconnectUserRequest`. Todos los `ChatMessage` recibidos por un `ChatRoom` tienen una propiedad `sender`, que contiene el ID de usuario que necesita para inicializar adecuadamente con una instancia de `DisconnectUserRequest`. Si lo desea, proporcione una cadena de motivos para la solicitud de desconexión.

```
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 este es otro ejemplo de una acción de moderador, puede intentar desconectar a otro usuario, pero no podrá hacerlo a menos que tenga la capacidad `DISCONNECT_USER`. La capacidad se establece cuando la aplicación de backend llama a `CreateChatToken` e inyecta la cadena `"DISCONNECT_USER"` en `capabilities`.

Si su usuario no tiene la capacidad de desconectar a otro usuario, `room.perform(request:)`arroja una instancia de `ChatError`, al igual que las demás solicitudes. Puede inspeccionar los errores de la propiedad `errorCode` para determinar si la solicitud falló debido a la falta de privilegios 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 mensajería del cliente de chat de Amazon IVS: tutorial para iOS

El SDK de mensajería del cliente de chat de Amazon Interactive Video (IVS) para iOS proporciona interfaces que permiten incorporar nuestras API de mensajería de chat de IVS en plataformas que utilizan el lenguaje de programación Swift de Apple.

Para ver el tutorial sobre el SDK de chat para iOS, consulte https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios.

SDK de mensajería del cliente de chat de Amazon IVS: guía para JavaScript

El SDK de mensajería del cliente de chat de Amazon Interactive Video Service (IVS) para JavaScript le permite incorporar fácilmente nuestras API de mensajería de chat de Amazon IVS en plataformas que utilizan un navegador web.

Versión más reciente de SDK de mensajería del cliente de chat de Amazon IVS para JavaScript: 1.0.2 (Notas de la versión)

Documentación de referencia: a fin de obtener información sobre los métodos más importantes disponibles en el SDK de mensajería del cliente de chat de Amazon IVS para JavaScript, consulte la documentación de referencia en: https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/

Código de ejemplo: consulte el repositorio de ejemplos en GitHub para ver una demostración específica de la web con SDK para JavaScript: https://github.com/aws-samples/amazon-ivs-chat-web-demo

Introducción

Antes de comenzar, debe estar familiarizado con Primeros pasos en el chat de Amazon IVS.

Añadir el paquete

Utilice:

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

o bien:

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

Soporte para React Native

El SDK de mensajería del cliente de chat de IVS para JavaScript tiene una dependencia `uuid` que utiliza el método `crypto.getRandomValues`. Como este método no es compatible con React Native, debe instalar el polyfill `react-native-get-random-value` adicional e importarlo en la parte superior del archivo `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);
```

Configuración del backend

Esta integración requiere puntos de conexión en su servidor que se comuniquen con la API de chat de Amazon IVS. Utilice las bibliotecas oficiales de AWS para acceder a la API de Amazon IVS desde

su servidor. Se puede acceder a ellas en varios idiomas desde los paquetes públicos, por ejemplo, node.js, java y go.

Cree un punto de conexión de servidor que se comunique con el punto de conexión CreateChatToken de API de chat de Amazon IVS, para crear un token de chat usuarios de chat.

Uso del SDK

Inicialización de una instancia de sala de chat

Cree una instancia de la clase `ChatRoom`. Para ello es necesario pasar `regionOrUrl` (la región de AWS en la que está alojada la sala de chat) y `tokenProvider` (el método de obtención de tokens se creará en el siguiente paso):

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

Función de proveedor de tokens

Cree una función de proveedor de token asíncrona que obtenga un token de chat de su backend:

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

La función no debe aceptar parámetros y devolver una promesa que contenga un objeto de token de chat:

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

Esta función es necesaria para inicializar el objeto ChatRoom. A continuación, complete los campos `<token>` y `<date-time>` con los valores recibidos de su 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 you backend to fetch chat token from IVS Chat endpoint:
```

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

Recuerde pasarlos `tokenProvider` al constructor de ChatRoom. ChatRoom actualiza el token cuando se interrumpe la conexión o caduca la sesión. No utilice el token `tokenProvider` para almacenar un token en ningún sitio; la sala de chat se encarga de hacerlo.

Recibir eventos

A continuación, suscríbase a los eventos de la sala de chat para recibir los eventos del ciclo de vida, así como los mensajes y eventos que se envíen en la 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" }
  * }
  */
});
```

Conectarse a la sala de chat

El último paso de la inicialización básica es conectarse a la sala de chat mediante el establecimiento de una conexión WebSocket. Para ello, simplemente llame al método `connect()` dentro de la instancia de la sala:

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

El SDK intentará establecer una conexión con una sala de chat codificada en el token de chat recibido de su servidor.

Después de llamar a `connect()`, la sala pasará al estado `connecting` y emitirá un evento `connecting`. Cuando la sala se conecta correctamente, pasa al estado `connected` y emite un evento `connect`.

Puede producirse un error de conexión debido a problemas al obtener el token o al conectarse a WebSocket. En este caso, la sala intenta volver a conectarse automáticamente hasta el número de veces indicado por el parámetro constructor `maxReconnectAttempts`. Durante los intentos de reconexión, la sala está en el estado `connecting` y no emite eventos adicionales. Tras agotar los intentos de reconexión, la sala pasa al estado `disconnected` y emite un evento `disconnect` (con un motivo de desconexión relevante). En el estado `disconnected`, la sala ya no intenta conectarse; debe volver a llamar `connect()` para iniciar el proceso de conexión.

Realizar acciones en una sala de chat

El SDK de mensajería de chat de Amazon IVS proporciona a los usuarios acciones para enviar y eliminar mensajes y desconectar a otros usuarios. Están disponibles en la instancia `ChatRoom`. Devuelven un objeto `Promise` que le permite recibir la confirmación o el rechazo de la solicitud.

Envío de un mensaje

Para esta solicitud, debe tener la capacidad `SEND_MESSAGE` codificada en su token de chat.

Para activar una solicitud de envío de mensajes:

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

Para obtener una confirmación/rechazo de la solicitud, `await` la promesa devuelta o utilice el 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.
}
```

Eliminar mensajes

Para esta solicitud, debe tener la capacidad `DELETE_MESSAGE` codificada en su token de chat.

Para eliminar un mensaje con fines de moderación, llame al método `deleteMessage()`:

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

Para obtener una confirmación/rechazo de la solicitud, `await` la promesa devuelta o utilice el 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 otro usuario

Para esta solicitud, debe tener la capacidad `DISCONNECT_USER` codificada en su token de chat.

Para desconectar a otro usuario con fines de moderación, llame el método `disconnectUser()`:

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

Para obtener una confirmación/rechazo de la solicitud, `await` la promesa devuelta o utilice el 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.
}
```

Desconectarse de una sala de chat

Para cerrar la conexión con la sala de chat, llame al método `disconnect()` en la instancia `room`:

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

Al invocar este método, la sala cierra el WebSocket subyacente de manera ordenada. La instancia de la sala pasa a un estado `disconnected` y emite un evento de desconexión, con el motivo `disconnect` establecido en `"clientDisconnect"`.

SDK de mensajería del cliente de chat de Amazon IVS - Parte 1 del tutorial de JavaScript: salas de chat

Esta es la primera parte del tutorial de dos partes. Conocerá los aspectos básicos del trabajo con el SDK de JavaScript de mensajería del cliente de chat de Amazon IVS al crear una aplicación totalmente funcional mediante JavaScript o TypeScript. Denominamos a la aplicación Chatterbox.

Los destinatarios previstos son desarrolladores experimentados que utilizan el SDK de mensajería del chat de Amazon IVS por primera vez. Debería estar a gusto con el lenguaje de programación JavaScript o TypeScript y la biblioteca React.

Por cuestiones de concisión, llamaremos al SDK de JavaScript de mensajería del cliente de chat de Amazon IVS como SDK de JS de chat.

Nota: En algunos casos, los ejemplos de código de JavaScript y TypeScript son idénticos, por lo cual se combinan.

Esta primera parte del tutorial se divide en varias secciones:

1. the section called “Configuración de un servidor local de autenticación y autorización”
2. the section called “Creación de un proyecto de Chatterbox”
3. the section called “Conectarse a una sala de chat”

4. the section called “Creación de un proveedor de tokens”
5. the section called “Visualización de las actualizaciones de conexión”
6. the section called “Creación de un componente de botón de envío”
7. the section called “Creación de una entrada de mensajes”
8. the section called “Siguientes pasos”

Para consultar la documentación completa del SDK, comience por Amazon IVS Chat Client Messaging SDK (aquí en la Guía del usuario de Chat de Amazon IVS) y Chat Client Messaging: SDK for JavaScript Reference (en GitHub).

Requisitos previos

- Conozca en detalle JavaScript o TypeScript y la biblioteca React. Si no estás familiarizado con React, aprende los conceptos básicos en Introducción a React.
- Lea y conozca Introducción al Chat de IVS.
- Cree un usuario de AWS IAM con las funcionalidades CreateChatToken y CreateRoom definidas en una política de IAM existente. (Consulte Introducción al Chat de IVS).
- Asegúrese de que las claves secretas o de acceso para este usuario estén almacenadas en un archivo de credenciales de AWS. Para obtener instrucciones, consulte la Guía del usuario de la AWS CLI (especialmente las Opciones de los archivos de configuración y credenciales).
- Cree una sala de chat y guarde su ARN. Consulte Introducción al Chat de IVS. (Si no guarda el ARN, puede buscarlo luego en la consola o en la API de chat.)
- Instale el entorno Node.js 14+ con NPM o el administrador de paquetes Yarn.

Configuración de un servidor local de autenticación y autorización

La aplicación backend se encarga de crear las salas de chat y de generar los tokens de chat necesarios para que el SDK de JS de chat autentique a los clientes y los autorice a entrar a las salas. Debe utilizar su propio backend porque no es seguro almacenar las claves de AWS en una aplicación móvil; los atacantes sofisticados podrían extraerlas y acceder a su cuenta de AWS.

Consulte Crear un token de chat en Introducción al chat de Amazon IVS. Como se muestra en ese diagrama de flujo, la aplicación del lado del servidor se encarga de crear el token de chat. Esto significa que su aplicación debe generarlo por cuenta propia al solicitarle uno a la aplicación del lado del servidor.

En esta sección, conocerá los conceptos básicos sobre la creación de un proveedor de tokens en su backend. Utilizamos el marco de Express para crear un servidor local activo que administre la creación de tokens de chat por medio del entorno local de AWS.

Cree un proyecto `npm` vacío con NPM. Cree un directorio para guardar la aplicación y utilícelo como directorio de trabajo:

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

Utilice `npm init` para crear un archivo `package.json` para la aplicación:

```
$ npm init
```

Este comando le solicitará varios datos, incluso el nombre y la versión de la aplicación. Por ahora, solo debe presionar RETURN (REGRESAR) para aceptar los valores predeterminados de la mayoría de ellos, salvo:

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

Pulse RETURN (REGRESAR) para aceptar el nombre predeterminado sugerido para el archivo de `index.js` o introduzca el nombre que desee para el archivo principal.

Ahora, instale las dependencias requeridas:

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

`aws-sdk` requiere las variables del entorno de configuración, las cuales se cargan de forma automática desde el archivo denominado `.env`, que se encuentra en el directorio raíz. Para configurarlo, cree un archivo nuevo denominado `.env` y complete la información de configuración que falta:

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

Ahora creamos un archivo de punto de entrada en el directorio raíz con el nombre que ingresó anteriormente en el comando `npm init`. En este caso, utilizamos `index.js` e importamos todos los paquetes necesarios:

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

Ahora, cree una instancia nueva de `express`:

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

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

Luego, puede crear su primer método POST de punto de conexión para el proveedor de tokens. Tome los parámetros requeridos del cuerpo de la solicitud (`roomId`, `userId`, `capabilities` y `sessionDurationInMinutes`):

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

Agregue la validación de los campos obligatorios:

```
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;
```

```
  }
});
```

Luego de preparar el método POST, integramos `createChatToken` y `aws-sdk` para la funcionalidad principal de la autenticación y la autorización:

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

Al final del archivo, agregue un oyente en el puerto para la aplicación `express`:

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

Ahora puede ejecutar el servidor con el siguiente comando desde la raíz del proyecto:

```
$ node index.js
```

Sugerencia: este servidor admite solicitudes de URL en https://localhost:3000.

Creación de un proyecto de Chatterbox

Primero debe crear el proyecto de React denominado `chatterbox`. Ejecute este comando:

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

Puede integrar el SDK de JS de mensajería del cliente de chat mediante el administrador del paquete de nodos o el administrador de paquetes Yarn:

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

Conectarse a una sala de chat

Aquí puede crear una `ChatRoom` y conectarse a ella mediante métodos asincrónicos. La clase de `ChatRoom` administra la conexión del usuario al SDK de JS de chat. Para conectarte de forma adecuada a la sala de chat, debe proporcionar una instancia de `ChatToken` dentro de la aplicación de React.

Navegue al archivo `App` que se crea en el proyecto `chatterbox` predeterminado y elimine todo lo que se encuentre entre las dos etiquetas `<div>`. No necesita ninguna parte del código que se rellenó previamente. Por el momento, nuestra `App` está prácticamente vacía.

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

import * as React from 'react';

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

Cree una instancia `ChatRoom` nueva y pásela al estado usando el enlace `useState`. Es necesario pasar `regionOrUrl` (la región de AWS en la que se aloja la sala de chat) y `tokenProvider` (que se utiliza para el flujo de autenticación y autorización de backend que se crea en los pasos posteriores).

Importante: Debe utilizar la misma región de AWS en la que creó la sala en Introducción al Chat de Amazon IVS. La API es un servicio regional de AWS. Para obtener un listado de las regiones

compatibles y los puntos de conexión del servicio HTTPS del chat de Amazon IVS, consulte la página de las regiones del chat de Amazon IVS.

```
// 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>;
}
```

Creación de un proveedor de tokens

El siguiente paso consiste en crear una función `tokenProvider` sin parámetros que necesita el constructor `ChatRoom`. Primero, crearemos una función `fetchChatToken` que hará una solicitud POST a la aplicación backend que configuró en the section called “Configuración de un servidor local de autenticación y autorización”. Los tokens de chat poseen la información necesaria para que el SDK establezca correctamente la conexión a la sala de chat. La API de chat utiliza estos tokens como un método seguro de validar la identidad del usuario, las funcionalidades dentro de la sala de chat y la duración de la sesión.

En el explorador de proyectos, cree un archivo nuevo de TypeScript o JavaScript denominado `fetchChatToken`. Cree una solicitud de recuperación para la aplicación `backend`, y devuelve el objeto `ChatToken` de la respuesta. Agregue las propiedades del cuerpo de la solicitud necesarias para crear un token de chat. Utilice las reglas definidas para los nombres de recursos de Amazon (ARN). Estas propiedades están documentadas en el punto de conexión de CreateChatToken.

Nota: La URL que utiliza aquí es la misma que creó el servidor local cuando ejecutó la aplicación 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),
  };
}
```

Visualización de las actualizaciones de conexión

La respuesta a los cambios en el estado de conexión de la sala de chat es una parte esencial en la creación de una aplicación de chat. Empecemos por la suscripción a los eventos importantes:

```
// 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>;
}
```

Luego, debemos establecer la capacidad de leer el estado de la conexión. Utilizamos nuestro enlace `useState` para crear determinado estado local en `App` y establecer el estado de conexión dentro de cada oyente.

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>;
}
```

Después de suscribirse al estado de la conexión, visualice el estado de la conexión y conéctese a la sala de chat mediante el método `room.connect` incluido en el enlace `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>
);

// ...
```

Implementó la conexión a la sala de chat con éxito.

Creación de un componente de botón de envío

En esta sección, creará un botón de envío que tenga un diseño diferente para cada estado de conexión. Este botón facilita el envío de mensajes en una sala de chat. También sirve como indicador visual para saber si se pueden enviar mensajes y cuándo es posible; por ejemplo, en caso de conexiones interrumpidas o sesiones de chat caducadas.

Primero, cree un nuevo archivo en el directorio `src` del proyecto de Chatterbox y denomínelo `SendButton`. Luego, cree el componente que mostrará el botón para la aplicación de chat. Exporte `SendButton` e impórtelo a `App`. En el `<div></div>` vacío, agregue `<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>
);
```

A continuación, defina una función con el nombre `onMessageSend` en `App` y pásela a la propiedad `SendButton onPress`. Defina otra variable denominada `isSendDisabled` (que evita el envío de mensajes cuando la sala no está conectada) y pásela a la propiedad `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>
);

// ...
```

Creación de una entrada de mensajes

La barra de mensajes de Chatterbox es el componente con el que interactuará para enviar mensajes a la sala de chat. Por lo general, contiene una entrada de texto donde puede redactar el mensaje y un botón para enviarlo.

Para crear un componente `MessageInput`, primero genere un archivo nuevo en el directorio `src` y denomínelo `MessageInput`. Luego, cree el componente de entrada controlada que mostrará la entrada para la aplicación de chat. Exporte el `MessageInput` e impórtelo a `App` (por encima del `<SendButton />`).

Cree un estado nuevo denominado `messageToSend` mediante el enlace `useState`, con una cadena vacía como valor predeterminado. En el cuerpo de la aplicación, pase `messageToSend` al `value` de `MessageInput` y pase `setMessageToSend` a la propiedad `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>
);
```

Siguientes pasos

Ahora que terminó de crear la barra de mensajes para Chatterbox, continúe con la parte 2 de este tutorial de JavaScript, Mensajes y eventos.

SDK de mensajería del cliente de chat de Amazon IVS - Parte 2 del tutorial de JavaScript: mensajes y eventos

Esta segunda (y última) parte del tutorial se divide en varias secciones:

1. the section called “Suscribirse a los eventos de mensajes de chat”
2. the section called “Mostrar los mensajes recibidos”
   a. the section called “Crear un componente de mensaje”
   b. the section called “Identificar los mensajes que envía el usuario actual”
   c. the section called “Crear un componente de lista de mensajes”
   d. the section called “Representar una lista de mensajes de chat”
3. the section called “Realizar acciones en una sala de chat”
   a. the section called “Envío de un mensaje”
   b. the section called “Eliminar mensajes”
4. the section called “Siguientes pasos”

Nota: En algunos casos, los ejemplos de código de JavaScript y TypeScript son idénticos, por lo cual se combinan.

Para consultar la documentación completa del SDK, comience por Amazon IVS Chat Client Messaging SDK (aquí en la Guía del usuario de Chat de Amazon IVS) y Chat Client Messaging: SDK for JavaScript Reference (en GitHub).

Requisito previo

Asegúrese de haber completado la parte 1 de este tutorial: salas de chat.

Suscribirse a los eventos de mensajes de chat

La instancia `ChatRoom` utiliza eventos para comunicarse cuando ocurren eventos en una sala de chat. Para comenzar a implementar la experiencia de chat, debe indicarles a los usuarios cuándo otros envían un mensaje en la sala a la que están conectados.

En este apartado, se suscribe a los eventos de mensajes de chat. Más adelante, le mostraremos cómo actualizar la lista de mensajes que crea, la cual se actualiza con cada mensaje o evento.

En su `App`, dentro del enlace `useEffect`, suscríbase a todos los eventos de mensajes:

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

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

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

Mostrar los mensajes recibidos

La recepción de mensajes es una parte fundamental de la experiencia de chat. Con el SDK de JS de chat, puede configurar su código para recibir eventos de forma sencilla de otros usuarios conectados a una sala de chat.

Más adelante, le mostraremos cómo realizar acciones en una sala de chat que aproveche los componentes que crea aquí.

En su `App`, defina un estado denominado `messages` con un tipo de matriz `ChatMessage` denominado `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([]);

  //...
}
```

A continuación, en la función oyente de `message`, agregue `message` a la matriz `messages`:

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

// ...

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

// ...
```

A continuación, se detallan las tareas para mostrar los mensajes recibidos:

1. the section called “Crear un componente de mensaje”
2. the section called “Identificar los mensajes que envía el usuario actual”
3. the section called “Crear un componente de lista de mensajes”
4. the section called “Representar una lista de mensajes de chat”

Crear un componente de mensaje

El componente `Message` se encarga de presentar el contenido de los mensajes recibidos en la sala de chat. En esta sección, creará un componente de mensajes para representar mensajes de chat individuales en `App`.

En el directorio `src`, cree un archivo que denominará `Message`. Pase el tipo `ChatMessage` para este componente y pase la cadena `content` de las propiedades `ChatMessage` para mostrar el texto del mensaje recibido desde los oyentes de mensajes de la sala de chat. En el explorador de proyectos, diríjase a `Message`.

TypeScript

```
// Message.tsx

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

type Props = {
  message: ChatMessage;
}

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

JavaScript

```
// Message.jsx

import * as React from 'react';

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

Sugerencia: utilice este componente para almacenar las diferentes propiedades que desee representar en las filas de mensajes; por ejemplo, las URL de los avatares, los nombres de usuario y las marcas de tiempo de cuando se envió el mensaje.

Identificar los mensajes que envía el usuario actual

Para identificar el mensaje que envía el usuario actual, modificamos el código y creamos un contexto de React para almacenar el `userId` de este usuario.

En el directorio `src`, cree un archivo que denominará `UserContext`:

TypeScript

```
// UserContext.tsx

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

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

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

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

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

  return context;
};

type UserProviderType = {
  children: ReactNode;
}

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

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

JavaScript

```
// UserContext.jsx

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

const UserContext = createContext(undefined);

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

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

  return context;
};

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

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

Nota: Aquí utilizamos el enlace `useState` para almacenar el valor `userId`. Más adelante, puede utilizar `setUserId` para cambiar el contexto del usuario o para iniciar sesión.

Luego, sustituya `userId` en el primer parámetro que se haya pasado a `tokenProvider`, utilizando el contexto creado 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']),
      }),
  );

  // ...
}
```

En el componente `Message`, utilice el `UserContext` que creó antes, declare la variable `isMine`, haga coincidir el `userId` del remitente con el `userId` del contexto y aplique diferentes estilos de mensajes para el usuario actual.

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>
  );
};
```

Crear un componente de lista de mensajes

El componente `MessageList` se encarga de mostrar la conversación de la sala de chat a lo largo del tiempo. El archivo `MessageList` es el contenedor que guarda todos nuestros mensajes. `Message` es una fila en `MessageList`.

En el directorio `src`, cree un archivo que denominará `MessageList`. Defina `Props` con `messages` una matriz del tipo de matriz `ChatMessage`. Dentro del cuerpo, asigne la propiedad `messages` y pase `Props` al componente `Message`.

TypeScript

```
// MessageList.tsx

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

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

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

JavaScript

```
// MessageList.jsx

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

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

Representar una lista de mensajes de chat

Ahora, incorpore la nueva `MessageList` al componente principal App:

```
// 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 las partes del rompecabezas ya se encuentran en su lugar para que la App comience a representar los mensajes recibidos en la sala de chat. A continuación, aprenderá a realizar acciones en la sala de chat que aprovecha los componentes que creó.

Realizar acciones en una sala de chat

El envío de mensajes y las acciones de moderador en una sala de chat son algunas de las formas principales para interactuar con la sala. Aquí aprenderá a utilizar varios objetos `ChatRequest` para realizar acciones comunes en Chatterbox, tales como enviar mensajes, eliminarlos y desconectar a otros usuarios.

Todas las acciones en la sala de chat siguen un patrón común: para cada acción que realice allí, hay un objeto de solicitud correspondiente. Para cada solicitud hay un objeto de respuesta correspondiente que recibe en la confirmación de la solicitud.

Siempre que los usuarios tengan los permisos correctos cuando crea un token de chat, podrán realizar las acciones correspondientes de forma adecuada utilizando los objetos de solicitud para ver cuáles puede realizar en la sala de chat.

A continuación, le explicamos cómo enviar un mensaje y eliminarlo.

Envío de un mensaje

La clase `SendMessageRequest` permite enviar mensajes en una sala de chat. Aquí, modifique la `App` para enviar la solicitud del mensaje mediante el componente que creó en Creación de una entrada de mensajes (en la parte 1 de este tutorial).

Para empezar, defina una propiedad booleana nueva denominada `isSending` con el enlace `useState`. Utilice esta propiedad nueva para cambiar el estado deshabilitado del elemento `button` HTML mediante la constante `isSendDisabled`. En el controlador de eventos del `SendButton`, borre el valor de `messageToSend` y configure `isSending` como verdadero.

Dado que realizará una llamada a la API desde este botón, si agrega el booleano *isSending* ayudará a evitar que se produzcan varias llamadas a la API al mismo tiempo, ya que deshabilita las interacciones de los usuarios en *SendButton* hasta que se complete la solicitud.

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

// ...

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

// ...

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

// ...

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

// ...
```

Prepare la solicitud mediante la creación de una instancia `SendMessageRequest` nueva, pasando el contenido del mensaje al constructor. Luego de configurar los estados `isSending` y `messageToSend`, llame al método `sendMessage`, el cual envía la solicitud a la sala de chat. Por último, borre la marca `isSending` al recibir la confirmación o la denegación de la solicitud.

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);
  }
};

// ...
```

Pruebe Chatterbox: intente enviar un mensaje que redacte con `MessageInput`, y presione `SendButton`. Debería ver el mensaje representado dentro de `MessageList` que creó anteriormente.

Eliminar mensajes

Para eliminar un mensaje de la sala de chat, debes tener la capacidad adecuada. Las capacidades se otorgan durante la inicialización del token de chat que utiliza para autenticarse en la sala de chat. Para los fines de esta sección, el formulario `ServerApp` de Configuración de un servidor local de autenticación y autorización local (en la parte 1 de este tutorial) le permite especificar las capacidades del moderador. Lo tiene que realizar en la aplicación con el objeto `tokenProvider` que creó en Creación de un proveedor de tokens (también en la parte 1).

Aquí puede modificar `Message` al agregar una función para eliminar el mensaje.

Primero, abra `App.tsx` y agregue la capacidad `DELETE_MESSAGE`. (`capabilities` es el segundo parámetro de la función `tokenProvider`).

Nota: Esta es la forma en que `ServerApp` informa a las API del chat de IVS de que el usuario asociado al token de chat resultante puede eliminar los mensajes de la sala de chat. En una situación real, probablemente se encontrará una lógica de backend más compleja para administrar las capacidades de los usuarios en la infraestructura de la aplicación del 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']),
  }),
);

// ...
```

En los siguientes pasos, actualizará `Message` para mostrar el botón de eliminación.

Abra `Message` y defina un nuevo estado booleano denominado `isDeleting` mediante el enlace `useState` con el valor inicial `false`. Con este estado, actualice el contenido de `Button` para que sea diferente según el estado actual de `isDeleting`. Desactive el botón cuando `isDeleting` sea verdadero; esto evita que intente realizar dos solicitudes de eliminación de mensajes al mismo tiempo.

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 una nueva función llamada `onDelete` que acepte una cadena como uno de los parámetros y devuelva `Promise`. En el cuerpo de la acción de cierre de `Button`, utilice `setIsDeleting` para cambiar el booleano `isDeleting` antes y después de una llamada a `onDelete`. Para el parámetro de cadena, pase el ID del mensaje del componente.

TypeScript

```
// Message.tsx

import React, { useState } from 'react';
import { ChatMessage } from 'amazon-ivs-chat-messaging';
import { useUserContext } from './UserContext';

export type Props = {
  message: ChatMessage;
  onDelete(id: string): Promise<void>;
};

export const Message = ({ message onDelete }: Props) => {
```

```
  const { userId } = useUserContext();
  const [isDeleting, setIsDeleting] = useState(false);
  const isMine = message.sender.userId === userId;
  const handleDelete = async () => {
    setIsDeleting(true);
    try {
      await onDelete(message.id);
    } catch (e) {
      console.log(e);
      // handle chat error here...
    } finally {
      setIsDeleting(false);
    }
  };

  return (
    <div style={{ backgroundColor: isMine ? 'lightblue' : 'silver', padding: 6,
 borderRadius: 10, margin: 10 }}>
      <p>{content}</p>
      <button onClick={handleDelete} disabled={isDeleting}>
        Delete
      </button>
    </div>
  );
};
```

JavaScript

```
// Message.jsx

import React, { useState } from 'react';
import { useUserContext } from './UserContext';

export const Message = ({ message, onDelete }) => {
  const { userId } = useUserContext();
  const [isDeleting, setIsDeleting] = useState(false);
  const isMine = message.sender.userId === userId;
  const handleDelete = async () => {
    setIsDeleting(true);
    try {
      await onDelete(message.id);
    } catch (e) {
      console.log(e);
```

```
      // handle the exceptions here...
    } finally {
      setIsDeleting(false);
    }
  };

  return (
    <div style={{ backgroundColor: 'silver', padding: 6, borderRadius: 10, margin:
 10 }}>
      <p>{message.content}</p>
      <button onClick={handleDelete} disabled={isDeleting}>
        Delete
      </button>
    </div>
  );
};
```

A continuación, actualice `MessageList` para que refleje los cambios más recientes en el componente `Message`.

Abra `MessageList` y defina una nueva función llamada `onDelete` que acepte una cadena como parámetro y devuelva `Promise`. Actualice su `Message` y páselo a las propiedades de `Message`. El parámetro de la cadena en el nuevo cierre será el identificador del mensaje que desea eliminar, que se transmite desde su `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} />
      ))}
    </>
  );
};
```

Luego, actualice la `App` para que muestre los cambios más recientes en `MessageList`.

Defina una función con el nombre `onDeleteMessage` en `App` y pásela a la propiedad `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 una solicitud al crear una instancia nueva de `DeleteMessageRequest`, pasando el identificador del mensaje correspondiente al parámetro constructor, y llame a `deleteMessage` que acepta la solicitud preparada previamente:

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);
};

// ...
```

Luego, actualice el estado `messages` para que refleje la lista nueva de mensajes que omite el mensaje que acaba de eliminar.

En el enlace `useEffect`, preste atención al evento `messageDelete` y actualice la matriz de estado `messages` al eliminar el mensaje con un identificador que coincida con el parámetro `message`.

Nota: Es posible que se genere el evento `messageDelete` para los mensajes que elimine el usuario actual o cualquier otro de la sala. Si lo administra en el controlador de eventos (en lugar de hacerlo junto a la solicitud `deleteMessage`) podrá unificar la administración de los mensajes eliminados.

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
```

```
};

// ...
```

Ahora puede eliminar usuarios de una sala de chat en la aplicación de chat.

Siguientes pasos

A modo de prueba, trate de implementar otras acciones en una sala, como desconectar a otro usuario.

Parte 1 del tutorial del SDK de mensajería para clientes de Chat de Amazon IVS para React Native: salas de chat

Esta es la primera parte del tutorial de dos partes. Conocerá los aspectos básicos del trabajo con el SDK de JavaScript de mensajería para clientes de Chat de Amazon IVS cuando crea una aplicación totalmente funcional mediante React Native. Denominamos a la aplicación Chatterbox.

Los destinatarios previstos son desarrolladores experimentados que utilizan el SDK de mensajería del chat de Amazon IVS por primera vez. Debería estar a gusto con los lenguajes de programación TypeScript o JavaScript y la biblioteca de React Native.

Por cuestiones de concisión, llamaremos al SDK de JavaScript de mensajería del cliente de chat de Amazon IVS como SDK de JS de chat.

Nota: En algunos casos, los ejemplos de código de JavaScript y TypeScript son idénticos, por lo cual se combinan.

Esta primera parte del tutorial se divide en varias secciones:

1. the section called “Configuración de un servidor local de autenticación y autorización”
2. the section called “Creación de un proyecto de Chatterbox”
3. the section called “Conectarse a una sala de chat”
4. the section called “Creación de un proveedor de tokens”
5. the section called “Visualización de las actualizaciones de conexión”
6. the section called “Creación de un componente de botón de envío”
7. the section called “Creación de una entrada de mensajes”
8. the section called “Siguientes pasos”

Requisitos previos

- Familiarícese con TypeScript o JavaScript y la biblioteca de React Native. Si no está familiarizado con React Native, aprenda los conceptos básicos en Intro to React Native (Introducción a React Native).
- Lea y conozca Introducción al Chat de IVS.
- Cree un usuario de AWS IAM con las funcionalidades CreateChatToken y CreateRoom definidas en una política de IAM existente. (Consulte Introducción al Chat de IVS).
- Asegúrese de que las claves secretas o de acceso para este usuario estén almacenadas en un archivo de credenciales de AWS. Para obtener instrucciones, consulte la Guía del usuario de la AWS CLI (especialmente las Opciones de los archivos de configuración y credenciales).
- Cree una sala de chat y guarde su ARN. Consulte Introducción al Chat de IVS. (Si no guarda el ARN, puede buscarlo luego en la consola o en la API de chat.)
- Instale el entorno Node.js 14+ con NPM o el administrador de paquetes Yarn.

Configuración de un servidor local de autenticación y autorización

La aplicación backend se encarga de crear las salas de chat y de generar los tokens de chat necesarios para que el SDK de JS de chat autentique a los clientes y los autorice a entrar a las salas. Debe utilizar su propio backend porque no es seguro almacenar las claves de AWS en una aplicación móvil; los atacantes sofisticados podrían extraerlas y acceder a su cuenta de AWS.

Consulte Crear un token de chat en Introducción al chat de Amazon IVS. Como se muestra en ese diagrama de flujo, la aplicación del lado del servidor se encarga de crear el token de chat. Esto significa que su aplicación debe generarlo por cuenta propia al solicitarle uno a la aplicación del lado del servidor.

En esta sección, conocerá los conceptos básicos sobre la creación de un proveedor de tokens en su backend. Utilizamos el marco de Express para crear un servidor local activo que administre la creación de tokens de chat por medio del entorno local de AWS.

Cree un proyecto `npm` vacío con NPM. Cree un directorio para guardar la aplicación y utilícelo como directorio de trabajo:

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

Utilice `npm init` para crear un archivo `package.json` para la aplicación:

```
$ npm init
```

Este comando le solicitará varios datos, incluso el nombre y la versión de la aplicación. Por ahora, solo debe presionar RETURN (REGRESAR) para aceptar los valores predeterminados de la mayoría de ellos, salvo:

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

Pulse RETURN (REGRESAR) para aceptar el nombre predeterminado sugerido para el archivo de `index.js` o introduzca el nombre que desee para el archivo principal.

Ahora, instale las dependencias requeridas:

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

`aws-sdk` requiere las variables del entorno de configuración, las cuales se cargan de forma automática desde el archivo denominado `.env`, que se encuentra en el directorio raíz. Para configurarlo, cree un archivo nuevo denominado `.env` y complete la información de configuración que falta:

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

Ahora creamos un archivo de punto de entrada en el directorio raíz con el nombre que ingresó anteriormente en el comando `npm init`. En este caso, utilizamos `index.js` e importamos todos los paquetes necesarios:

```
// index.js
```

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

Ahora, cree una instancia nueva de `express`:

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

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

Luego, puede crear su primer método POST de punto de conexión para el proveedor de tokens. Tome los parámetros requeridos del cuerpo de la solicitud (`roomId`, `userId`, `capabilities` y `sessionDurationInMinutes`):

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

Agregue la validación de los campos obligatorios:

```
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;
  }
});
```

Luego de preparar el método POST, integramos `createChatToken` y `aws-sdk` para la funcionalidad principal de la autenticación y la autorización:

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

Al final del archivo, agregue un oyente en el puerto para la aplicación `express`:

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

Ahora puede ejecutar el servidor con el siguiente comando desde la raíz del proyecto:

```
$ node index.js
```

Sugerencia: este servidor admite solicitudes de URL en https://localhost:3000.

Creación de un proyecto de Chatterbox

Primero debe crear el proyecto de React Native denominado `chatterbox`. Ejecute este comando:

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

O crear un proyecto de exposición con una plantilla de TypeScript.

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

Puede integrar el SDK de JS de mensajería del cliente de chat mediante el administrador del paquete de nodos o el administrador de paquetes Yarn:

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

Conectarse a una sala de chat

Aquí puede crear una `ChatRoom` y conectarse a ella mediante métodos asincrónicos. La clase de `ChatRoom` administra la conexión del usuario al SDK de JS de chat. Para conectarte de forma adecuada a la sala de chat, debe proporcionar una instancia de `ChatToken` dentro de la aplicación de React.

Navegue al archivo `App` que se crea en el proyecto `chatterbox` predeterminado y elimine todo lo que devuelva un componente funcional. No necesita ninguna parte del código que se rellenó previamente. Por el momento, nuestra `App` está prácticamente vacía.

TypeScript/JavaScript:

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

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

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

Cree una instancia `ChatRoom` nueva y pásela al estado usando el enlace `useState`. Es necesario pasar `regionOrUrl` (la región de AWS en la que se aloja la sala de chat) y `tokenProvider` (que se utiliza para el flujo de autenticación y autorización de backend que se crea en los pasos posteriores).

Importante: Debe utilizar la misma región de AWS en la que creó la sala en Introducción al Chat de Amazon IVS. La API es un servicio regional de AWS. Para obtener un listado de las regiones compatibles y los puntos de conexión del servicio HTTPS del chat de Amazon IVS, consulte la página de las regiones del chat de Amazon IVS.

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>;
}
```

Creación de un proveedor de tokens

El siguiente paso consiste en crear una función `tokenProvider` sin parámetros que necesita el constructor `ChatRoom`. Primero, crearemos una función `fetchChatToken` que hará una solicitud POST a la aplicación backend que configuró en the section called “Configuración de un servidor local de autenticación y autorización”. Los tokens de chat poseen la información necesaria para que el SDK establezca correctamente la conexión a la sala de chat. La API de chat utiliza estos tokens como un método seguro de validar la identidad del usuario, las funcionalidades dentro de la sala de chat y la duración de la sesión.

En el explorador de proyectos, cree un archivo nuevo de TypeScript o JavaScript denominado `fetchChatToken`. Cree una solicitud de recuperación para la aplicación `backend`, y devuelve el objeto `ChatToken` de la respuesta. Agregue las propiedades del cuerpo de la solicitud necesarias para crear un token de chat. Utilice las reglas definidas para los nombres de recursos de Amazon (ARN). Estas propiedades están documentadas en el punto de conexión de CreateChatToken.

Nota: La URL que utiliza aquí es la misma que creó el servidor local cuando ejecutó la aplicación 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),
  };
}
```

Visualización de las actualizaciones de conexión

La respuesta a los cambios en el estado de conexión de la sala de chat es una parte esencial en la creación de una aplicación de chat. Empecemos por la suscripción a los eventos importantes:

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>;
}
```

Luego, debemos establecer la capacidad de leer el estado de la conexión. Utilizamos nuestro enlace `useState` para crear determinado estado local en `App` y establecer el estado de conexión dentro de cada oyente.

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>;
}
```

Después de suscribirse al estado de la conexión, visualice el estado de la conexión y conéctese a la sala de chat mediante el método `room.connect` incluido en el enlace `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,
  }
});

// ...
```

Implementó la conexión a la sala de chat con éxito.

Creación de un componente de botón de envío

En esta sección, creará un botón de envío que tenga un diseño diferente para cada estado de conexión. Este botón facilita el envío de mensajes en una sala de chat. También sirve como indicador visual para saber si se pueden enviar mensajes y cuándo es posible; por ejemplo, en caso de conexiones interrumpidas o sesiones de chat caducadas.

Primero, cree un nuevo archivo en el directorio `src` del proyecto de Chatterbox y denomínelo `SendButton`. Luego, cree el componente que mostrará el botón para la aplicación de chat. Exporte `SendButton` e impórtelo a `App`. En el `<View></View>` vacío, agregue `<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>
);
```

A continuación, defina una función con el nombre `onMessageSend` en `App` y pásela a la propiedad `SendButton onPress`. Defina otra variable denominada `isSendDisabled` (que evita el envío de mensajes cuando la sala no está conectada) y pásela a la propiedad `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>
);

// ...
```

Creación de una entrada de mensajes

La barra de mensajes de Chatterbox es el componente con el que interactuará para enviar mensajes a la sala de chat. Por lo general, contiene una entrada de texto donde puede redactar el mensaje y un botón para enviarlo.

Para crear un componente `MessageInput`, primero genere un archivo nuevo en el directorio `src` y denomínelo `MessageInput`. Luego, cree el componente de entrada que mostrará la entrada para la aplicación de chat. Exporte el `MessageInput` e impórtelo a `App` (por encima del `<SendButton />`).

Cree un estado nuevo denominado `messageToSend` mediante el enlace `useState`, con una cadena vacía como valor predeterminado. En el cuerpo de la aplicación, pase `messageToSend` al `value` de `MessageInput` y pase `setMessageToSend` a la propiedad `onMessageChange`:

TypeScript

```
// MessageInput.tsx
```

```
import * as React from 'react';

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

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

const styles = StyleSheet.create({
  input: {
    fontSize: 20,
    backgroundColor: 'rgb(239,239,240)',
    paddingHorizontal: 18,
    paddingVertical: 15,
    borderRadius: 50,
    flex: 1,
  }
})

// App.tsx

// ...

import { MessageInput } from './MessageInput';

// ...

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

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <View style={styles.messageBar}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
```

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

const styles = StyleSheet.create({
  root: {
    flex: 1,
  },
  messageBar: {
    borderTopWidth: StyleSheet.hairlineWidth,
    borderTopColor: 'rgb(160,160,160)',
    flexDirection: 'row',
    padding: 16,
    alignItems: 'center',
    backgroundColor: 'white',
  }
});
```

JavaScript

```
// MessageInput.jsx

import * as React from 'react';

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

const styles = StyleSheet.create({
  input: {
    fontSize: 20,
    backgroundColor: 'rgb(239,239,240)',
    paddingHorizontal: 18,
    paddingVertical: 15,
    borderRadius: 50,
    flex: 1,
  }
})
```

```
// App.jsx

// ...

import { MessageInput } from './MessageInput';

// ...

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

// ...

return (
  <SafeAreaView style={styles.root}>
    <Text>Connection State: {connectionState}</Text>
    <View style={styles.messageBar}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </View>
  </SafeAreaView>
);

const styles = StyleSheet.create({
  root: {
    flex: 1,
  },
  messageBar: {
    borderTopWidth: StyleSheet.hairlineWidth,
    borderTopColor: 'rgb(160,160,160)',
    flexDirection: 'row',
    padding: 16,
    alignItems: 'center',
    backgroundColor: 'white',
  }
});
```

Siguientes pasos

Ahora que terminó de crear la barra de mensajes para Chatterbox, continúe con la parte 2 de este tutorial de React Native, Messages and Events (Mensajes y eventos).

Parte 2 del tutorial del SDK de mensajería para clientes de Chat de Amazon IVS para React Native: mensajes y eventos

Esta segunda (y última) parte del tutorial se divide en varias secciones:

1. the section called “Suscribirse a los eventos de mensajes de chat”
2. the section called “Mostrar los mensajes recibidos”
   a. the section called “Crear un componente de mensaje”
   b. the section called “Identificar los mensajes que envía el usuario actual”
   c. the section called “Representar una lista de mensajes de chat”
3. the section called “Realizar acciones en una sala de chat”
   a. the section called “Envío de un mensaje”
   b. the section called “Eliminar mensajes”
4. the section called “Siguientes pasos”

Nota: En algunos casos, los ejemplos de código de JavaScript y TypeScript son idénticos, por lo cual se combinan.

Requisito previo

Asegúrese de haber completado la parte 1 de este tutorial: salas de chat.

Suscribirse a los eventos de mensajes de chat

La instancia `ChatRoom` utiliza eventos para comunicarse cuando ocurren eventos en una sala de chat. Para comenzar a implementar la experiencia de chat, debe indicarles a los usuarios cuándo otros envían un mensaje en la sala a la que están conectados.

En este apartado, se suscribe a los eventos de mensajes de chat. Más adelante, le mostraremos cómo actualizar la lista de mensajes que crea, la cual se actualiza con cada mensaje o evento.

En su `App`, dentro del enlace `useEffect`, suscríbase a todos los eventos de mensajes:

TypeScript/JavaScript:

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

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

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

Mostrar los mensajes recibidos

La recepción de mensajes es una parte fundamental de la experiencia de chat. Con el SDK de JS de chat, puede configurar su código para recibir eventos de forma sencilla de otros usuarios conectados a una sala de chat.

Más adelante, le mostraremos cómo realizar acciones en una sala de chat que aproveche los componentes que crea aquí.

En su `App`, defina un estado denominado `messages` con un tipo de matriz `ChatMessage` denominado `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([]);

  //...
}
```

A continuación, en la función oyente de `message`, agregue `message` a la matriz `messages`:

TypeScript/JavaScript:

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

// ...

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

// ...
```

A continuación, se detallan las tareas para mostrar los mensajes recibidos:

1. the section called “Crear un componente de mensaje”
2. the section called “Identificar los mensajes que envía el usuario actual”
3. the section called “Representar una lista de mensajes de chat”

Crear un componente de mensaje

El componente `Message` se encarga de presentar el contenido de los mensajes recibidos en la sala de chat. En esta sección, creará un componente de mensajes para representar mensajes de chat individuales en `App`.

En el directorio `src`, cree un archivo que denominará `Message`. Pase el tipo `ChatMessage` para este componente y pase la cadena `content` de las propiedades `ChatMessage` para mostrar el texto del mensaje recibido desde los oyentes de mensajes de la sala de chat. En el explorador de proyectos, diríjase a `Message`.

TypeScript

```
// Message.tsx

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

type Props = {
  message: ChatMessage;
}

export const Message = ({ message }: Props) => {
  return (
    <View style={styles.root}>
      <Text>{message.sender.userId}</Text>
      <Text style={styles.textContent}>{message.content}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
});
```

JavaScript

```
// Message.jsx

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
```

```
export const Message = ({ message }) => {
  return (
    <View style={styles.root}>
      <Text>{message.sender.userId}</Text>
      <Text style={styles.textContent}>{message.content}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
});
```

Sugerencia: utilice este componente para almacenar las diferentes propiedades que desee representar en las filas de mensajes; por ejemplo, las URL de los avatares, los nombres de usuario y las marcas de tiempo de cuando se envió el mensaje.

Identificar los mensajes que envía el usuario actual

Para identificar el mensaje que envía el usuario actual, modificamos el código y creamos un contexto de React para almacenar el `userId` de este usuario.

En el directorio `src`, cree un archivo que denominará `UserContext`:

TypeScript

```
// UserContext.tsx

import React from 'react';
```

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

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

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

  return context;
};

export const UserProvider = UserContext.Provider;
```

JavaScript

```
// UserContext.jsx

import React from 'react';

const UserContext = React.createContext(undefined);

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

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

  return context;
};

export const UserProvider = UserContext.Provider;
```

Nota: Aquí utilizamos el enlace `useState` para almacenar el valor `userId`. Más adelante, puede utilizar `setUserId` para cambiar el contexto del usuario o para iniciar sesión.

Luego, sustituya el `userId` en el primer parámetro que se haya pasado al `tokenProvider`, utilizando el contexto creado anteriormente. Asegúrese de agregar la capacidad `SEND_MESSAGE`

a su proveedor de tokens, como se especifica a continuación, ya que es necesaria para enviar mensajes.

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']),
      }),
  );

  // ...
}
```

En el componente `Message`, utilice el `UserContext` que creó antes, declare la variable `isMine`, haga coincidir el `userId` del remitente con el `userId` del contexto y aplique diferentes estilos de mensajes para el usuario actual.

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',
  },
});
```

Representar una lista de mensajes de chat

Ahora enumere los mensajes utilizando una `FlatList` y un componente `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 las partes del rompecabezas ya se encuentran en su lugar para que la App comience a representar los mensajes recibidos en la sala de chat. A continuación, aprenderá a realizar acciones en la sala de chat que aprovecha los componentes que creó.

Realizar acciones en una sala de chat

El envío de mensajes y las acciones de moderador en una sala de chat son algunas de las formas principales para interactuar con la sala de chat. Aquí aprenderá a utilizar varios objetos de solicitud de chat para realizar acciones comunes en Chatterbox, tales como enviar mensajes, eliminarlos y desconectar a otros usuarios.

Todas las acciones en la sala de chat siguen un patrón común: para cada acción que realice allí, hay un objeto de solicitud correspondiente. Para cada solicitud hay un objeto de respuesta correspondiente que recibe en la confirmación de la solicitud.

Siempre que los usuarios tengan las capacidades correctas cuando crea un token de chat, podrán realizar las acciones correspondientes de forma adecuada utilizando los objetos de solicitud para ver cuáles puede realizar en la sala de chat.

A continuación, le explicamos cómo enviar un mensaje y eliminarlo.

Envío de un mensaje

La clase `SendMessageRequest` permite enviar mensajes en una sala de chat. Aquí, modifique la `App` para enviar la solicitud del mensaje mediante el componente que creó en Creación de una entrada de mensajes (en la parte 1 de este tutorial).

Para empezar, defina una propiedad booleana nueva denominada `isSending` con el enlace `useState`. Utilice esta propiedad nueva para cambiar el estado deshabilitado del elemento `button` mediante la constante `isSendDisabled`. En el controlador de eventos del `SendButton`, borre el valor de `messageToSend` y configure `isSending` como verdadero.

Dado que realizará una llamada a la API desde este botón, si agrega el booleano *`isSending`* ayudará a evitar que se produzcan varias llamadas a la API al mismo tiempo, ya que deshabilita las interacciones de los usuarios en *`SendButton`* hasta que se complete la solicitud.

Nota: El envío de mensajes solo funciona si ha agregado la capacidad `SEND_MESSAGE` al proveedor de tokens, como se indica anteriormente en Identificar los mensajes que envía el usuario actual.

TypeScript/JavaScript:

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

// ...

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

// ...

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

```
};

// ...

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

// ...
```

Prepare la solicitud mediante la creación de una instancia `SendMessageRequest` nueva, pasando el contenido del mensaje al constructor. Luego de configurar los estados `isSending` y `messageToSend`, llame al método `sendMessage`, el cual envía la solicitud a la sala de chat. Por último, borre la marca `isSending` al recibir la confirmación o la denegación de la solicitud.

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);
  }
};

// ...
```

Pruebe Chatterbox: intente enviar un mensaje que redacte con `MessageBar`, y presione `SendButton`. Debería ver el mensaje representado dentro de `MessageList` que creó anteriormente.

Eliminar mensajes

Para eliminar un mensaje de la sala de chat, debes tener la capacidad adecuada. Las capacidades se otorgan durante la inicialización del token de chat que utiliza para autenticarse en la sala de chat. Para los fines de esta sección, el formulario `ServerApp` de [Configuración de un servidor local de autenticación y autorización local](#) (en la parte 1 de este tutorial) le permite especificar las capacidades del moderador. Lo tiene que realizar en la aplicación con el objeto `tokenProvider` que creó en [Creación de un proveedor de tokens](#) (también en la parte 1).

Aquí puede modificar `Message` al agregar una función para eliminar el mensaje.

Primero, abra `App.tsx` y agregue la capacidad `DELETE_MESSAGE`. (`capabilities` es el segundo parámetro de la función `tokenProvider`).

Nota: Esta es la forma en que `ServerApp` informa a las API del chat de IVS de que el usuario asociado al token de chat resultante puede eliminar los mensajes de la sala de chat. En una situación real, probablemente se encontrará una lógica de backend más compleja para administrar las capacidades de los usuarios en la infraestructura de la aplicación del servidor.

TypeScript/JavaScript:

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

// ...

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

// ...
```

En los siguientes pasos, actualizará `Message` para mostrar el botón de eliminación.

Defina una nueva función llamada `onDelete` que acepte una cadena como uno de los parámetros y devuelva `Promise`. Para el parámetro de cadena, pase el ID del mensaje del componente.

TypeScript

```
// Message.tsx
```

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

export type Props = {
  message: ChatMessage;
  onDelete(id: string): Promise<void>;
};

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

  const isMine = message.sender.userId === userId;
  const handleDelete = () => onDelete(message.id);

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

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  content: {
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'space-between',
  },
```

```
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
  mine: {
    flexDirection: 'row-reverse',
    backgroundColor: 'lightblue',
  },
});
```

JavaScript

```
// Message.jsx

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

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

  const isMine = message.sender.userId === userId;
  const handleDelete = () => onDelete(message.id);

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

const styles = StyleSheet.create({
  root: {
    backgroundColor: 'silver',
    padding: 6,
```

```
    borderRadius: 10,
    marginHorizontal: 12,
    marginVertical: 5,
    marginRight: 50,
  },
  content: {
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'space-between',
  },
  textContent: {
    fontSize: 17,
    fontWeight: '500',
    flexShrink: 1,
  },
  mine: {
    flexDirection: 'row-reverse',
    backgroundColor: 'lightblue',
  },
});
```

A continuación, actualice `renderItem` para que refleje los cambios más recientes en el componente `FlatList`.

Defina una función con el nombre `handleDeleteMessage` en `App` y pásela a la propiedad `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 una solicitud al crear una instancia nueva de `DeleteMessageRequest`, pasando el identificador del mensaje correspondiente al parámetro constructor, y llame a `deleteMessage` que acepta la solicitud preparada previamente:

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);
  };

  // ...
```

Luego, actualice el estado `messages` para que refleje la lista nueva de mensajes que omite el mensaje que acaba de eliminar.

En el enlace `useEffect`, preste atención al evento `messageDelete` y actualice la matriz de estado `messages` al eliminar el mensaje con un identificador que coincida con el parámetro `message`.

Nota: Es posible que se genere el evento `messageDelete` para los mensajes que elimine el usuario actual o cualquier otro de la sala. Si lo administra en el controlador de eventos (en lugar de hacerlo junto a la solicitud `deleteMessage`) podrá unificar la administración de los mensajes eliminados.

TypeScript/JavaScript:

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

Ahora puede eliminar usuarios de una sala de chat en la aplicación de chat.

Siguientes pasos

A modo de prueba, trate de implementar otras acciones en una sala, como desconectar a otro usuario.

SDK de mensajería para clientes de Chat de Amazon IVS: prácticas recomendadas para React y React Native

En este documento se describen las prácticas más importantes relacionadas con el uso del SDK de mensajería de Chat de Amazon IVS para React y React Native. Esta información debe permitirle crear una funcionalidad de chat típica dentro de una aplicación React y proporcionarle los antecedentes que necesita para profundizar en las partes más avanzadas del SDK de mensajería de Chat de IVS.

Creación de un enlace inicializador de sala de chat

La clase `ChatRoom` contiene métodos de chat básicos y oyentes para administrar el estado de la conexión y escuchar eventos como el mensaje recibido y el mensaje eliminado. Aquí mostramos cómo almacenar adecuadamente las instancias de chat en un enlace.

Implementación

TypeScript

```
// useChatRoom.ts

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

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

  return { room };
};
```

JavaScript

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

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

  return { room };
};
```

Nota: No utilizamos el método `dispatch` del enlace `setState` porque no se pueden actualizar los parámetros de configuración sobre la marcha. El SDK crea una instancia una vez y no es posible actualizar el proveedor del token.

Importante: Utilice el enlace inicializador de la `ChatRoom` una vez para inicializar una nueva instancia de sala de chat.

Ejemplo

TypeScript/JavaScript:

```
// ...

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

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

  // ...
};

// ...
```

Escucha del estado de conexión

Si lo desea, puede suscribirse a las actualizaciones del estado de la conexión en el enlace de su sala de chat.

Implementación

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 };
};
```

Proveedor de instancias de la sala de chat

Para usar el enlace en otros componentes (para evitar el prop drilling), puede crear un proveedor de sala de chat mediante el `context` de React.

Implementación

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;
```

Ejemplo

Después de crear el `ChatRoomProvider`, puede consumir la instancia mediante `useChatRoomContext`.

Importante: Coloque al proveedor en el nivel raíz solo si necesita acceder a `context` entre la pantalla de chat y los demás componentes del centro, para evitar que se vuelvan a renderizar innecesariamente si escucha conexiones. De lo contrario, coloque al proveedor lo más cerca posible de la pantalla 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();
  };
  // ...
};

// ...
```

Creación de un oyente de mensajes

Para estar al día con todos los mensajes entrantes, debe suscribirse a los eventos `message` y `deleteMessage`. A continuación, se muestra un código que proporciona mensajes de chat para los componentes.

Importante: Por motivos de rendimiento, separamos `ChatMessageContext` de `ChatRoomProvider`, ya que es posible que se vuelvan a renderizar muchas veces cuando el oyente del mensaje de chat actualice el estado de su mensaje. Recuerde aplicar el `ChatMessageContext` en los componentes donde vaya a utilizar el `ChatMessageProvider`.

Implementación

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>;
};
```

Ejemplo en React

Importante: Recuerde encapsular el contenedor del mensaje con el `ChatMessagesProvider`. La fila `Message` es un componente de ejemplo que muestra el contenido de un mensaje.

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>
  );
};
```

Ejemplo en React Native

De forma predeterminada, el `ChatMessage` contiene el `id`, que se usa automáticamente como claves de React en la `FlatList` para cada fila; por lo tanto, no es necesario pasar el `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} />;
};
```

Varias instancias de sala de chat en una aplicación

Si utiliza varias salas de chat simultáneas en la aplicación, le proponemos que cree un proveedor para cada chat y lo consuma en el proveedor de chat. En este ejemplo, creamos un bot de ayuda y un chat de ayuda para el cliente. Creamos un proveedor 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>;
};
```

Ejemplo en React

Ahora puede utilizar diferentes proveedores de chat que usen el mismo `ChatRoomProvider`. Más adelante, puede reutilizar el mismo `useChatRoomContext` dentro de cada pantalla o vista.

TypeScript/JavaScript:

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

const App = () => {
  return (
    <Routes>
```

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

Ejemplo en 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 />
    </>
  );
};
```

Seguridad de Chat de Amazon IVS

La seguridad en la nube de AWS es la mayor prioridad. Como cliente de AWS, se beneficiará de una arquitectura de red y un centro de datos que están diseñados para satisfacer los requisitos de seguridad de las organizaciones más exigentes.

La seguridad es una responsabilidad compartida entre AWS y el usuario. El modelo de responsabilidad compartida la describe como seguridad de la nube y seguridad en la nube:

- Seguridad de la nube: AWS es responsable de proteger la infraestructura que ejecuta los servicios de AWS en la nube de AWS. AWS también proporciona servicios que puede utilizar de forma segura. Auditores independientes prueban y verifican periódicamente la eficacia de nuestra seguridad en el marco de los programas de conformidad de AWS.
- Seguridad en la nube: su responsabilidad está determinada por el servicio de AWS que utilice. Usted también es responsable de otros factores, incluida la confidencialidad de los datos, los requisitos de la empresa y la legislación y los reglamentos aplicables.

Esta documentación lo ayuda a comprender cómo aplicar el modelo de responsabilidad compartida cuando se utiliza Chat de Amazon IVS. En los siguientes temas, se muestra cómo configurar Chat de Amazon IVS para cumplir con los objetivos de seguridad y conformidad.

Temas

- Protección de los datos
- Identity and Access Management
- Políticas administradas para Amazon IVS
- Uso de roles vinculados a servicios para Amazon IVS
- Registro y monitorización
- Respuesta frente a incidencias
- Resiliencia
- Seguridad de infraestructuras

Protección de los datos

En el caso de los datos enviados a Chat de Amazon Interactive Video Service (IVS), se aplican las siguientes protecciones de datos:

- El tráfico del chat de Amazon IVS utiliza WSS para mantener seguros los datos en tránsito.
- Los tokens de chat de Amazon IVS se cifran mediante claves administradas por el cliente de KMS.

Chat de Amazon IVS no requiere que proporcione ningún dato de cliente (usuario final). No hay campos en las salas de chat, entradas o grupos de seguridad de entrada en los que se espera que proporcione datos de cliente (usuario final).

No coloque información de identificación confidencial, como números de cuenta de cliente (usuario final), en campos de formato libre como un campo Name (Nombre). Esto incluye cuando trabaja con la consola o la API de Amazon IVS, la CLI de AWS o los SDK de AWS. Cualquier dato que ingrese en Chat de Amazon IVS se puede incluir en los registros de diagnóstico.

Las transmisiones no están cifradas de extremo a extremo; una transmisión puede transmitirse sin cifrar internamente en la red de IVS, para su procesamiento.

Identity and Access Management

AWS Identity and Access Management (IAM) es un servicio de AWS que ayuda al administrador de una cuenta a controlar de forma segura el acceso a los recursos de AWS. Consulte Identity and Access Management en la Guía del usuario de Transmisión de baja latencia de IVS.

Público

La forma en que utilice IAM difiere en función del trabajo que realice en Amazon IVS. Consulte Audience en la Guía del usuario de Transmisión de baja latencia de IVS.

Cómo funciona Amazon IVS con IAM

Para poder realizar solicitudes de API de Amazon IVS, debe crear una o varias identidades de IAM (usuarios, grupos y roles) y políticas de IAM y, a continuación, adjunte políticas a identidades. Los permisos tardan hasta unos minutos en propagarse; hasta entonces, las solicitudes de API se rechazan.

Para obtener una perspectiva general de cómo funciona Amazon IVS con IAM, consulte Servicios de AWS que funcionan con IAM en la Guía del usuario de IAM.

Identidades

Puede crear identidades de IAM para proporcionar autenticación a personas y procesos en la cuenta de AWS. Los grupos de IAM son conjuntos de usuarios de IAM que puede administrar como una unidad. Consulte Identidades (usuarios, grupos y roles) en la Guía del usuario de IAM.

Políticas

Un documento de política de JSON se compone de elementos. Consulte Policies en la Guía del usuario de Transmisión de baja latencia de IVS.

Chat de Amazon IVS admite tres elementos:

- Acciones: las acciones de política para Chat de Amazon IVS utilizan el prefijo `ivschat` antes de la acción. Por ejemplo, para conceder a alguien permiso para crear una sala de Chat de Amazon IVS con el método de la API `CreateRoom` de Chat de Amazon IVS, incluya la acción `ivschat:CreateRoom` en la política para esa persona. Las instrucciones de la política deben incluir un elemento `Action` o un elemento `NotAction`.
- Recursos: el recurso de la sala de Chat de Amazon IVS tiene el siguiente formato de ARN:

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

  Por ejemplo, para especificar la sala `VgNkEJgOVX9N` en la instrucción, utilice este ARN:

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

  Algunas acciones de Chat de Amazon IVS, como las que se utilizan para crear recursos, no se pueden llevar a cabo en un recurso específico. En dichos casos, debe utilizar el carácter comodín (`*`):

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

- Condiciones: Chat de Amazon IVS admite algunas claves de condición globales (`aws:RequestTag`, `aws:TagKeys` y `aws:ResourceTag`).

Puede utilizar variables como marcadores de posición en una política. Por ejemplo, puede conceder un permiso de usuario de IAM para acceder a un recurso, solo si está etiquetado con el nombre de usuario de IAM. Consulte Variables y etiquetas en la Guía del usuario de IAM.

Amazon IVS proporciona políticas administradas de AWS que se pueden utilizar para conceder un conjunto preconfigurado de permisos a las identidades (acceso total o de solo lectura). Puede optar por utilizar políticas administradas en lugar de las políticas basadas en la identidad que están a continuación. Para obtener detalles, consulte Políticas administradas para Amazon IVS.

Autorización basada en etiquetas de Amazon IVS

Puede asociar etiquetas a los recursos de Chat de Amazon IVS o transferirlas en una solicitud a Chat de Amazon IVS. Para controlar el acceso en función de etiquetas, debe proporcionar información de las etiquetas en el elemento de condición de una política utilizando las claves de condición `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` o `aws:TagKeys`. Para obtener más información sobre el etiquetado de recursos de Chat de Amazon IVS, consulte “Tagging” en la Referencia de la API de Chat de IVS.

Roles

Consulte Roles de IAM y Credenciales de seguridad temporales en la Guía del usuario de IAM.

Un rol de IAM es una entidad de la cuenta de AWS que dispone de permisos específicos.

Amazon IVS admite el uso de credenciales de seguridad temporales. Puede utilizar credenciales temporales para iniciar sesión con identidad federada, asumir un rol de IAM o asumir un rol de acceso entre cuentas. Las credenciales de seguridad temporales se obtienen mediante una llamada a operaciones de la API de AWS Security Token Service, como `AssumeRole` o `GetFederationToken`.

Acceso privilegiado y sin privilegios

Los recursos de la API tienen acceso privilegiado. El acceso a la reproducción sin privilegios se puede configurar a través de canales privados; consulte Setting Up Private Channels.

Prácticas recomendadas para utilizar las políticas

Consulte Prácticas recomendadas de IAM en la Guía del usuario de IAM.

Las políticas basadas en identidad son muy eficaces. Determinan si alguien puede crear, acceder o eliminar los recursos de Amazon IVS de la cuenta. Estas acciones pueden generar costos adicionales para su cuenta de AWS. Siga estas recomendaciones:

- Conceda privilegios mínimos: al crear políticas personalizadas, conceda solo los permisos necesarios para llevar a cabo una tarea. Comience con un conjunto mínimo de permisos y conceda permisos adicionales según sea necesario. Por lo general, es más seguro que comenzar con permisos que son muy poco estrictos e intentar hacerlos más estrictos más adelante. Concretamente, reserve `ivschat:*` para el acceso de administrador y no lo utilice en aplicaciones.
- Habilite la Multi-Factor Authentication (MFA) para operaciones confidenciales: para mayor seguridad, exija a los usuarios de IAM que utilicen la autenticación multifactor (MFA) para acceder a recursos u operaciones de API confidenciales.
- Utilice condiciones de política para mayor seguridad: en la medida en que sea práctico, defina las condiciones en las que las políticas basadas en la identidad permiten el acceso a un recurso. Por ejemplo, puede escribir condiciones para especificar un rango de direcciones IP permitidas desde el que debe proceder una solicitud. También puede escribir condiciones para permitir solicitudes solo en un intervalo de hora o fecha especificado o para solicitar el uso de SSL o MFA.

Ejemplos de políticas basadas en identidad

Uso de la consola de Amazon IVS

Para acceder a la consola de Amazon IVS, debe contar con un conjunto mínimo de permisos que permitan mostrar y ver detalles sobre los recursos de Chat de Amazon IVS de la cuenta de AWS. Si crea una política de permisos basados en la identidad que sea más restrictiva que el mínimo de permisos necesarios, la consola no funcionará del modo esperado para las identidades que posean esa política. Para garantizar el acceso a la consola de Amazon IVS, asocie la siguiente política a las identidades (consulte Agregado y eliminación de permisos de IAM en la Guía del usuario de IAM).

Las partes de la siguiente política proporcionan acceso a:

- Todos los puntos de conexión de la API de Chat de Amazon IVS
- Sus cuotas de servicio de Chat de Amazon IVS
- Publicación de lambdas y adición de permisos para la lambda elegida para la moderación de chat de Amazon IVS
- Amazon CloudWatch para obtener métricas de su sesión 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 basada en recursos para chat de Amazon IVS

Debe autorizar al servicio de chat de Amazon IVS a invocar su recurso lambda para revisar los mensajes. Para ello, siga las instrucciones que se indican en Uso de políticas basadas en recursos para AWS Lambda (en la Guía para desarrolladores de AWS Lambda) y complete los campos que se indican a continuación.

para controlar el acceso a los sus recursos lambda, puede utilizar las condiciones de sus políticas basadas en:

- `SourceArn`: nuestra política de prueba utiliza un comodín (`*`) para permitir que todas las salas de su cuenta invoquen la lambda. Opcionalmente, puede especificar una sala de su cuenta para permitir que solo esa sala invoque la lambda.
- `SourceAccount`: en la política de prueba que se indica a continuación, el ID de cuenta de AWS es `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/*"
            }
         }
      }
   ]
}
```

Solución de problemas

Consulte Troubleshooting en la Guía del usuario de Transmisión de baja latencia de IVS para obtener información sobre el diagnóstico y la solución de los problemas comunes que es posible encontrar al trabajar con Chat de Amazon IVS y IAM.

Políticas administradas para Amazon IVS

Una política administrada por AWS es una política independiente creada y administrada por AWS. Consulte Managed Policies for Amazon IVS en la Guía del usuario de Transmisión de baja latencia de IVS.

Uso de roles vinculados a servicios para Amazon IVS

Amazon IVS utiliza roles vinculados a servicios de AWS IAM. Consulte Using Service-Linked Roles for Amazon IVS en la Guía del usuario de Transmisión de baja latencia de IVS.

Registro y monitorización

Para registrar el rendimiento o las operaciones, utilice Amazon CloudTrail. Consulte Logging Amazon IVS API Calls with AWS CloudTrail en la Guía del usuario de Transmisión de baja latencia de IVS.

Respuesta frente a incidencias

Para detectar o alertar de incidentes, puede monitorizar el estado de una transmisión a través de eventos de Amazon EventBridge. Consulte Uso de Amazon EventBridge con Amazon IVS: para transmisión de baja latencia y para transmisión en tiempo real.

Utilice el Panel de AWS Health para obtener información sobre el estado general de Amazon IVS (por región).

Resiliencia

Las API de IVS utilizan la infraestructura global de AWS y están conformadas por regiones y zonas de disponibilidad de AWS. Consulte Resilience en la Guía del usuario de Transmisión de baja latencia de IVS.

Seguridad de infraestructuras

Como se trata de un servicio administrado, Amazon IVS está protegido por los procedimientos de seguridad de red globales de AWS. Se describen en Prácticas recomendadas para seguridad, identidad y conformidad.

API Calls (Llamadas a la API)

Puede utilizar llamadas a la API publicadas en AWS para acceder a Amazon IVS a través de la red. Consulte API Calls en Infrastructure Security en la Guía del usuario de Transmisión de baja latencia de IVS.

Chat de Amazon IVS

La incorporación y entrega de mensajes de chat de Amazon IVS se produce a través de conexiones WSS cifradas a nuestra periferia. La API de mensajería de Amazon IVS utiliza conexiones HTTPS cifradas. Al igual que con la reproducción y transmisión de vídeo, se necesita la versión 1.2 o versiones posteriores de TLS y los datos de mensajería se transmiten sin cifrar internamente para su procesamiento.

Cuotas de servicio (Chat)

A continuación se indican las cuotas de servicio y los límites para los puntos de conexión, los recursos y otras operaciones de Chat de Amazon Interactive Video Service (IVS). Las cuotas de servicio (que también se denominan límites) establecen el número máximo de recursos u operaciones de servicio que puede haber en una cuenta de AWS. Es decir, estos límites corren por cuenta de AWS, a menos que se indique lo contrario en la tabla. Consulte también Service Quotas de AWS.

Para conectarse mediante programación a un servicio de AWS, utilice un punto de enlace. Consulte también Puntos de enlace del servicio de AWS.

Todas las cuotas se aplican por región.

Aumentos en la cuota de servicio

Para cuotas ajustables, puede solicitar un aumento de la tasa a través de la consola de AWS. También, utilice la consola para consultar información sobre cuotas de servicio.

Las cuotas de tarifa de llamadas de API no son ajustables.

Cuotas de tarifa de llamadas a la API

Tipo de punto de conexión	Punto de conexión	Predeterminado
Mensajería	`DeleteMessage`	100 TPS
Mensajería	`DisconnectUser`	100 TPS
Mensajería	`SendEvent`	100 TPS
Token de chat	`CreateChatToken`	200 TPS
Configuración de registro	`CreateLoggingConfiguration`	3 TPS
Configuración de registro	`DeleteLoggingConfiguration`	3 TPS
Configuración de registro	`GetLoggingConfiguration`	3 TPS

Tipo de punto de conexión	Punto de conexión	Predeterm inado
Configuración de registro	`ListLoggingConfigurations`	3 TPS
Configuración de registro	`UpdateLoggingConfiguration`	3 TPS
Sala	`CreateRoom`	5 TPS
Sala	`DeleteRoom`	5 TPS
Sala	`GetRoom`	5 TPS
Sala	`ListRooms`	5 TPS
Sala	`UpdateRoom`	5 TPS
Etiquetas	`ListTagsForResource`	10 TPS
Etiquetas	`TagResource`	10 TPS
Etiquetas	`UntagResource`	10 TPS

Otras cuotas

Recurso o característica	Predeterm inado	Ajustable	Descripción
Conexiones simultáneas de chat	50 000	Sí	Número máximo de conexione s de chat simultáneas por cuenta, en todas las salas en un Región de AWS.
Configuraciones de registro	10	Sí	La cantidad máxima de configuraciones de registro que se pueden crear por cuenta en la Región de AWS actual.

Recurso o característica	Predeterminado	Ajustable	Descripción
Período de espera del controlador de revisión de mensajes	200	No	Período de espera en milisegundos para todos los controladores de revisión de mensajes del actual Región de AWS. Si se excede, el mensaje se permite o deniega según el valor del `fallbackResult` que configuró para el controlador de revisión de mensajes.
Tasa de solicitudes de DeleteMessage en todas las salas	100	Sí	Número máximo de solicitudes de DeleteMessage por segundo que pueden hacerse en todas las salas. Las solicitudes pueden provenir de la API de chat de Amazon IVS o de la API de mensajería de chat de Amazon IVS (WebSocket).
Tasa de solicitudes de DisconnectUser en todas las salas	100	Sí	Número máximo de solicitudes de DisconnectUser por segundo que pueden hacerse en todas las salas. Las solicitudes pueden provenir de la API de chat de Amazon IVS o de la API de mensajería de chat de Amazon IVS (WebSocket).

Recurso o característica	Predeterminado	Ajustable	Descripción
Tasa de solicitudes de mensajería por conexión	10	No	Número máximo de solicitudes de mensajería por segundo que puede realizar una conexión de chat.
Tasa de solicitudes de SendMessage en todas las salas	1 000	Sí	Número máximo de solicitudes de SendMessage por segundo que pueden hacerse en todas las salas. Estas solicitudes proceden de la API de mensajería de chat de Amazon IVS (WebSocket).
Tasa de solicitudes de SendMessage por sala	100	No (pero se puede configurar a través de la API)	Número máximo de solicitudes de SendMessage por segundo que pueden hacerse para cualquiera de las salas. Esto se puede configurar con el campo `maximumMessageRatePerSecond` de CreateRoom y UpdateRoom. Estas solicitudes proceden de la API de mensajería de chat de Amazon IVS (WebSocket).
Salas	50 000	Sí	Número máximo de salas de chat por cuenta, por Región de AWS.

Integración de Service Quotas con las métricas de uso de CloudWatch

Puede utilizar CloudWatch para administrar de forma proactiva sus cuotas de servicio mediante las métricas de uso de CloudWatch. Puede utilizar estas métricas para visualizar el uso actual del servicio en paneles y gráficos de CloudWatch. Las métricas de uso de Chat de Amazon IVS se corresponden con las cuotas de servicio de Chat de Amazon IVS.

Puede utilizar una función de cálculo de métricas de CloudWatch para mostrar las cuotas de servicio de esos recursos en los gráficos. También puede configurar alarmas que le avisen cuando su uso se acerque a una cuota de servicio.

Para acceder a las métricas de uso:

1. Abra la consola de Service Quotas en https://console.aws.amazon.com/servicequotas/
2. En el panel de navegación, seleccione Servicios de AWS.
3. En la lista Servicios de AWS, busque y seleccione Chat de Amazon Interactive Video Service.
4. En la lista Cuotas de servicio, seleccione la cuota de servicio de interés. Se abre una nueva página con información sobre la cuota o métrica de servicio.

También puede acceder a estas métricas a través de la consola de CloudWatch. En Espacios de nombres de AWS, elija Uso. Luego, en la lista Servicio, elija Chat de IVS. (Consulte Supervisión de Chat de Amazon IVS).

En el espacio de nombres AWS/Usage, Chat de Amazon IVS proporciona la siguiente métrica:

Nombre de métrica	Descripción
`ResourceCount`	El recuento de los recursos especificados que se ejecutan en su cuenta. Los recursos se definen por las dimensiones asociadas a la métrica. Estadística válida: máximo (número máximo de recursos utilizados durante el periodo de un minuto).

Las siguientes dimensiones se utilizan para ajustar las métricas de uso:

Dimensión	Descripción
Servicio	Nombre del servicio de AWS que contiene el recurso. Valor válido: `IVS Chat`.
Clase	La clase de recurso a la que se realiza el seguimiento. Valor válido: `None`.
Tipo	El tipo de recurso del que se realiza el seguimiento. Valor válido: `Resource`.
Recurso	El nombre del recurso de AWS. Valor válido: `ConcurrentChatConnections` . La métrica de uso ConcurrentChatConnections es una copia de la que se encuentra en el espacio de nombres AWS/IVSChat (con la dimensión Ninguna), como se describe en Supervisión de Chat de Amazon IVS.

Creación de una alarma de CloudWatch para métricas de uso

Para crear una alarma de CloudWatch basada en una métrica de uso de Chat de Amazon IVS:

1. En la consola de Service Quotas, seleccione la cuota de servicio de interés, como se describió anteriormente. Actualmente solo se pueden crear alarmas para ConcurrentChatConnections.
2. En la sección Alarmas de Amazon CloudWatch, elija Create (Crear).
3. De la lista desplegable Alarm threshold (Umbral de alarma), elija el porcentaje del valor de la cuota aplicada que desee establecer como valor de la alarma.
4. En Alarm name (Nombre de la alarma), escriba un nombre para la alarma.
5. Seleccione Create (Crear).

Preguntas frecuentes de solución de problemas

En este documento se describen las prácticas recomendadas y los consejos de solución de problemas de Chat de Amazon Interactive Video Service (IVS). Los comportamientos relacionados con el chat de IVS suelen ser distintos de los relacionados con el video de IVS. Para obtener más información, consulte Getting Started with Amazon IVS Chat (Introducción a Chat de Amazon IVS).

Temas:

- the section called “¿Por qué no se desconectaron las conexiones del chat de IVS cuando se eliminó la sala?”

¿Por qué no se desconectaron las conexiones del chat de IVS cuando se eliminó la sala?

Cuando se elimina un recurso de sala de chat, si la sala se utiliza activamente, los clientes del chat que están conectados a la sala no se desconectan automáticamente. La conexión se interrumpe cuando la aplicación de chat actualiza el token del chat. Como alternativa, se debe desconectar de forma manual a todos los usuarios para eliminar a todos los usuarios de la sala de chat.

Glosario

Consulte también el glosario de AWS. En la siguiente tabla, LL son las siglas de transmisión de baja latencia de IVS; RT, transmisión en tiempo real de IVS.

Plazo	Descripción	LL	RT	Chat
AAC	Advanced Audio Coding. AAC es un estándar de codificación de audio para la compresión de audio digital con pérdida. Diseñado para ser el sucesor del formato MP3, AAC generalmente logra una calidad de sonido superior a este con la misma velocidad de bits. El formato AAC fue estandarizado por la ISO y la IEC como parte de las especificaciones MPEG-2 y MPEG-4.	✓	✓	
Transmisión con velocidad de bits adaptable	La transmisión con velocidad de bits adaptable (ABR) permite al reproductor de IVS cambiar a una velocidad de bits más baja cuando la calidad de la conexión se vea afectada y volver a una velocidad más alta cuando la calidad mejore.	✓		
Transmisión adaptativa	Consulte Codificación por capas con transmisión simultánea.		✓	
Usuario administr ativo	Un usuario de AWS con acceso administrativo a los recursos y servicios disponibles en una cuenta de AWS. Consulte Terminología en la Guía del usuario de configuración de AWS.	✓	✓	✓
ARN	Nombre de recurso de Amazon, un identificador exclusivo de cualquier recurso de AWS. Los formatos específicos de ARN dependen del tipo de recurso. Para conocer los formatos de ARN utilizados por los recursos de IVS, consulte la referencia de autorización de servicio.	✓	✓	✓

Plazo	Descripción	LL	RT	Chat
Relación de aspecto	Describe la relación entre el ancho y la altura del marco. Por ejemplo, 16:9 es la relación de aspecto que corresponde a la resolución Full HD o 1080p.	✓	✓	
Modo de audio	Una configuración de audio preestablecida o personalizada que está optimizada para diferente s tipos de usuarios de dispositivos móviles y los equipos que usan. Consulte SDK de transmisión de IVS: modos de audio móvil (streaming en tiempo real).		✓	
AVC, H.264, MPEG-4 parte 10	La codificación de video avanzada, también denominada H.264 o MPEG-4 parte 10, es un estándar para la compresión de video digital con pérdida.	✓	✓	
Reemplazo de fondo	Un tipo de filtro de cámara que permite a los creadores de transmisiones en directo cambiar sus fondos. Consulte Reemplazo de fondo en SDK de transmisión de IVS: filtros de cámara de terceros (streaming en tiempo real).		✓	
Velocidad de bits	Métrica de transmisión que indica el número de bits que se trasmiten o reciben por segundo.	✓	✓	
Difusión, emisora	Otros términos para transmisión o streamer.	✓		

Plazo	Descripción	LL	RT	Chat
Almacenamiento en búfer	Una condición que se produce cuando el dispositivo de reproducción no puede descargar el contenido antes de que este se reproduzca. El almacenamiento en búfer puede manifestarse de varias formas: el contenido puede detenerse y comenzar de forma aleatoria (lo que también se conoce como inestabilidad), el contenido puede detenerse durante periodos prolongados (también conocido como congelación) o el reproductor de IVS se puede pausar.	✓	✓	
Listas de reproducción por rango de bytes	Una lista de reproducción más detallada que la lista de reproducción HLS estándar. La lista de reproducción HLS estándar se compone de archivos multimedia de 10 segundos. Con una lista de reproducción por rango de bytes, la duración del segmento es la misma que el intervalo de fotogramas clave configurado para la transmisión. La lista de reproducción por rango de bytes solo está disponible para las emisiones que se grabaron automáticamente en un bucket de S3. Se crea además de la lista de reproducción HLS. Consulte Listas de reproducción por rango de bytes en Grabación automática en Amazon S3 (streaming de baja latencia).	✓		

Plazo	Descripción	LL	RT	Chat
CBR	Velocidad de bits constante (CBR): un método de control de velocidad para codificadores que mantiene una velocidad de bits constante durante toda la reproducción de un video, sin importar lo que ocurra durante la transmisión. Las pausas en la acción se pueden rellenar para alcanzar la velocidad de bits deseada. De igual manera, los picos se pueden cuantificar al ajustar la calidad de la codificación para que coincida con la velocidad de bits objetivo. Recomendamos encarecidamente utilizar CBR en lugar de VBR.	✓	✓	
CDN	Red de entrega de contenidos o red de distribuc ión de contenidos: una solución distribuida de manera geográfica que optimiza la distribución de contenidos, como la transmisión de video, y la acerca al lugar donde se encuentran los usuarios.	✓		
Canal	Un recurso de IVS que almacena la configuración para la transmisión, lo que incluye un servidor de ingesta, una clave de transmisión, una URL de reproducción y opciones de grabación. Los streamers utilizan la clave de transmisión asociada a un canal para iniciar una difusión. Todas las métricas y eventos están asociados a un recurso de canal.	✓		
Tipo de canal	Determina la resolución y la velocidad de fotogramas permitidas para el canal. Consulte Tipos de canales en la Referencia de la API de transmisión de baja latencia de IVS.	✓		
Registro del chat	Una opción avanzada que se puede habilitar mediante la vinculación de una configuración de registro con una sala de chat.			✓

Plazo	Descripción	LL	RT	Chat
Sala de chat	Recurso de IVS que almacena la configuración de una sesión de chat e incluye funciones opcionale s como el controlador de revisión de mensajes y el registro de conversaciones. Consulte Step 2: Create a Chat Room en Getting Started with IVS Chat.			✓
Composición del cliente	Usa un dispositivo host para mezclar las transmisi ones de audio y video de los participantes del escenario y, a continuación, las envía como una transmisión compuesta a un canal de IVS. Esto permite un mayor control sobre el aspecto de la composición, pero a costa de una mayor utilizaci ón de los recursos del cliente y un mayor riesgo de que un problema con el escenario o el host afecte a los espectadores. Consulte también Composición del servidor.	✓	✓	
CloudFront	Un servicio de CDN que proporciona Amazon.	✓		
CloudTrail	Un servicio de AWS para recopilar, supervisar, analizar y retener los eventos y la actividad de las cuentas de AWS, así como de orígenes externos. Consulte Registro de llamadas a la API de IVS con AWS CloudTrail.	✓	✓	✓
CloudWatch	Un servicio de AWS para supervisar las aplicacio nes, responder a los cambios de rendimiento, optimizar el uso de los recursos y proporcionar información sobre el estado de las operacion es. Puede usar CloudWatch para supervisar las métricas de IVS; consulte Supervisión del streaming en tiempo real de IVS y Monitoreo de transmisión de baja latencia de Amazon IVS.	✓	✓	✓

Plazo	Descripción	LL	RT	Chat
Composición	Proceso que consiste en combinar transmisiones de audio y video de varios orígenes en una sola transmisión.	✓	✓	
Canalización de composición	Secuencia de pasos de procesamiento necesaria para combinar varias transmisiones y codificar la resultante.	✓	✓	
Compresión	Codificación de la información con menos bits que la representación original. Cualquier compresió n en particular puede ser con o sin pérdida. La compresión sin pérdida reduce los bits al identificar y eliminar la redundancia estadística. No se pierde información en la compresión sin pérdida. La compresión con pérdida reduce los bits al eliminar información innecesaria o menos importante.	✓	✓	
Plano de control	Almacena información sobre los recursos de IVS, como canales, escenarios o salas de chat. Además, proporciona interfaces para crear y administrar estos recursos. Es regional (se basa en las regiones de AWS).	✓	✓	✓
CORS	Uso compartido de recursos entre orígenes: una característica que permite a las aplicaciones web de los clientes cargadas en un dominio interactu ar con los recursos (como buckets de S3) de un dominio diferente. El acceso se puede configurar en función de los encabezados, los métodos HTTP y los dominios de origen. Consulte Uso compartid o de recursos entre orígenes (CORS): Amazon Simple Storage Service en la Guía del usuario de Amazon Simple Storage Service.	✓		

Plazo	Descripción	LL	RT	Chat
Origen de imagen personalizado	Una interfaz proporcionada por el SDK de transmisión de IVS que permite que una aplicación proporcione su propia entrada de imagen en vez de limitarse a las cámaras predeterminadas.	✓	✓	
Plano de datos	La infraestructura que transmite los datos desde la ingesta hasta la salida. Funciona según la configuración administrada en el plano de control y no está restringida a una región de AWS.	✓	✓	✓
Codificador, codificación	Proceso de convertir contenido de video y audio a un formato digital que sea adecuado para la transmisión. La codificación puede estar basada en hardware o software.	✓	✓	
Evento	Notificación automática publicada por IVS en el servicio de supervisión de AmazonEventBridge. Un evento representa un cambio de estado de un recurso de transmisión, como un escenario o una canalización de composición. Consulte Uso de Amazon EventBridge con el streaming de baja latencia de IVS y Uso de Amazon EventBridge con el streaming en tiempo real de IVS.	✓	✓	✓
FFmpeg	Un proyecto de software libre y de código abierto que consiste en un conjunto de bibliotecas y programas para gestionar archivos y transmisiones de audio y video. FFmpeg proporciona una solución compatible con diversas plataformas para grabar, convertir y transmitir audio y video.	✓		

Plazo	Descripción	LL	RT	Chat
Transmisión fragmentada	Se crea cuando una transmisión se desconect a y, a continuación, se vuelve a conectar dentro del intervalo especificado en la configuración de grabación del canal. Los diversas transmisiones resultantes se consideran una sola, por lo que se combinan en una sola transmisión grabada. Consulte Fusionar transmisiones fragmentadas en Grabación automática en Amazon S3 (streaming de baja latencia).	✓		
Velocidad de fotogramas	Métrica de transmisión que indica el número de fotogramas que se trasmiten o reciben por segundo.	✓	✓	
HLS	HTTP Live Streaming (HLS): un protocolo de comunicaciones de transmisión con velocidad de bits adaptable que se basa en HTTP y se utiliza para entregar transmisiones de IVS a los espectadores.	✓		
Lista de reproducc ión HLS	Lista de segmentos multimedia que componen una transmisión. Las listas de reproducción HLS estándar se componen de archivos multimedia de 10 segundos. HLS también admite listas de reproducción con un rango de bytes más granular.	✓		
Host	Participante de un evento en tiempo real que envía video o audio al escenario.		✓	
IAM	Identity and Access Management: un servicio de AWS que permite a los usuarios administrar de forma segura las identidades y el acceso a los servicios y recursos de AWS, incluido IVS.	✓	✓	✓

Plazo	Descripción	LL	RT	Chat
Incorporación	Proceso de IVS que consiste en recibir transmisi ones de video de un host o difusor para su procesamiento o entrega a los espectadores u otros participantes.	✓	✓	
Servidor de incorporación	Recibe transmisiones de video y las envía a un sistema de transcodificación, en el que las transmisiones se multiplexan o transcodifican en HLS para su entrega a los espectadores. Los servidores de ingesta son componentes específicos de IVS y reciben las transmisiones de los canales, así como un protocolo de ingesta (RTMP, RTMPS). Consulte la información sobre cómo crear un canal en Introducción al streaming de baja latencia de IVS.		✓	
Video entrelazado	Transmite y muestra solo las líneas pares o impares de los fotogramas posteriores para duplicar la velocidad de fotogramas percibida sin consumir más ancho de banda. No recomendamos utilizar video entrelazado debido a problemas en su calidad.	✓	✓	
JSON	JavaScript Object Notation: formato de archivo estándar abierto que utiliza texto legible por humanos para transmitir objetos de datos que consisten en pares de valor de atributo y tipos de datos de matriz o cualquier otro valor serializable.	✓	✓	✓

Plazo	Descripción	LL	RT	Chat
Fotograma clave, fotograma delta, intervalo de fotogramas clave	El fotograma clave (también denominado intracodificado o fotograma i) es un fotograma completo de la imagen de un video. Los fotogramas posteriores, los fotogramas delta (también denominados fotogramas previstos o fotogramas p), solo contienen la información que ha cambiado. Los fotogramas clave aparecerán varias veces dentro de una transmisión, según el intervalo de fotogramas clave definido en el codificador.	✓	✓	
Lambda	Un servicio de AWS para ejecutar código (denominadas funciones de Lambda) sin aprovisionar ninguna infraestructura de servidor. Las funciones de Lambda se pueden ejecutar en respuesta a eventos y solicitudes de invocación o según una programación. Por ejemplo, Chat de IVS utiliza funciones de Lambda para permitir la revisión de mensajes en una sala de chat.	✓	✓	✓
Latencia, latencia integral	Un retraso en la transferencia de datos. IVS define los rangos de latencia de la siguiente manera: • Baja latencia: menos de 3 segundos • Latencia en tiempo real: menos de 300 ms La latencia integral se refiere al retraso desde el momento en que una cámara captura una transmisión en vivo hasta el momento en que la transmisión aparece en la pantalla de un espectador.	✓	✓	
Codificación por capas con la transmisión simultánea	Permite la codificación y publicación simultáneas de varias transmisiones de video con distintos niveles de calidad. Consulte Transmisión adaptativa: codificación en capas con transmisión simultánea y Optimizaciones de streaming en tiempo real.		✓	

Plazo	Descripción	LL	RT	Chat
Controlador de revisión de mensajes	Permite a los clientes de Chat de IVS revisar y filtrar automáticamente los mensajes de los usuarios antes de enviarlos a la sala de chat. Se habilita al asociar una función de Lambda con una sala de chat. Consulte Creating a Lambda Function en Chat Message Review Handler.			✓
Mezclador	Una característica de los SDK de transmisión móvil de IVS que recibe varios orígenes de audio y video y genera una única salida. Admite la administr ación de los elementos de video y audio que están en pantalla y que representan orígenes como cámaras, micrófonos, capturas de pantalla, así como audio y video generados por la aplicació n. Luego, la salida se puede transmitir a IVS. Consulte Configuración de una sesión de transmisi ón de mezcla en SDK de transmisión de IVS: guía del mezclador (streaming de baja latencia).	✓		
Transmisión de varios hosts	Combina transmisiones de varios hosts en una sola transmisión. Esto se puede lograr mediante una composición del cliente o del servidor. La transmisión de varios hosts permite, por ejemplo, invitar a los espectadores a un escenario para una sesión de preguntas y respuestas, concursos entre hosts, videoconferencias y hosts conversando entre sí frente a un gran público.		✓	
Lista de reproducc ión multivariante	Un índice de todas las transmisiones variantes disponibles para una transmisión.	✓		
OAC	Origin Access Control: un mecanismo para restringi r el acceso a un bucket de S3, de modo que el contenido, como una transmisión grabada, solo se pueda ofrecer a través de la CDN de CloudFront.	✓		

Plazo	Descripción	LL	RT	Chat
OBS	Open Broadcaster Software: software gratuito y de código abierto para grabación y transmisión en directo de video. OBS ofrece una alternativa (al SDK de transmisión de IVS) para la autoedici ón. Es posible que los streamers más sofisticados estén familiarizados con OBS y lo prefieran por sus características de producción avanzadas, como las transiciones de escenas, la mezcla de audio y la superposición de gráficos.	✓	✓	
Participante	Un usuario en tiempo real conectado al escenario como host o espectador.		✓	
Token de participa nte	Autentica a un participante del evento en tiempo real cuando se une a un escenario. Un token de participante también controla si un participante puede enviar video al escenario.		✓	
Token de reproducc ión, par de claves de reproducción	Un mecanismo de autorización que permite a los clientes restringir la reproducción de video en canales privados. Los tokens de reproducción se generan a partir de un par de claves de reproducc ión. Un par de claves de reproducción es el par de claves público-privadas utilizado para firmar y validar el token de autorización del espectador para la reproducción. Consulte Crear o importar una clave de reproducción en Configuración de canales privados y consulte los puntos de conexión del par de claves de reproducción en la referencia de la API de baja latencia de IVS.	✓		

Plazo	Descripción	LL	RT	Chat
URL de reproducc ión	Identifica la dirección que utiliza el espectador para iniciar la reproducción de un canal específico. Esta dirección se puede utilizar globalmente. IVS selecciona automáticamente la mejor ubicación de la red global de distribución de contenido de IVS para entregar el video a cada espectador. Consulte la información sobre cómo crear un canal en Introducción al streaming de baja latencia de IVS.	✓		
Canal privado	Permite a los clientes restringir el acceso a sus transmisiones mediante un mecanismo de autorización que se basa en los tokens de reproducción. Consulte Flujo de trabajo para canales privados en Configuración de canales privados.	✓		
Video progresivo	Transmite y muestra todas las líneas de cada fotograma en secuencia. Recomendamos utilizar video progresivo durante todas las etapas de la transmisión.	✓	✓	
Cuotas	El número máximo de recursos u operaciones de servicio de IVS que puede haber en una cuenta de AWS. Es decir, estos límites corren por cuenta de AWS, a menos que se indique lo contrario. Todas las cuotas se aplican por región. Consulte Puntos de conexión y cuotas de Amazon Interactive Video Service en la Guía de referencia general de AWS.	✓	✓	✓

Plazo	Descripción	LL	RT	Chat
Regiones	Otorga acceso a servicios de AWS que se ubican físicamente en un área geográfica determinada. Las regiones proporcionar tolerancia a errores, estabilidad y resistencia, y también pueden reducir la latencia. Con las regiones, puede crear recursos redundantes que sigan estando disponibles y no resulten afectados por una interrupción regional. La mayoría de las solicitudes de servicio de AWS están asociadas a una región geográfica en particular. Los recursos que crea en una región no existen en ninguna otra región salvo que utilice explícitamente una característica de replicación ofrecida por un servicio de AWS. Por ejemplo, Amazon S3 admiten la replicación entre regiones. Algunos servicios, como IAM, no tienen recursos entre regiones.	✓	✓	✓
Resolución	Describe la cantidad de píxeles en un único fotograma de video; por ejemplo, Full HD o 1080p definen un fotograma con 1920 x 1080 píxeles.	✓	✓	
Usuario raíz	El propietario de una cuenta de AWS. Este usuario raíz tiene acceso a todos los recursos y los servicios de AWS en la cuenta de AWS.	✓	✓	✓
RTMP, RTMPS	Protocolo de mensajes en tiempo real: un estándar del sector para la transmisión de audio, video y datos a través de una red. RTMPS es la versión segura de RTMP, que se ejecuta a través de una conexión de seguridad de la capa de transporte (TLS/SSL).	✓	✓	

Plazo	Descripción	LL	RT	Chat
Bucket de S3	Un conjunto de objetos almacenados en Amazon S3. Muchas políticas, como las de acceso y replicación, se definen en el nivel de bucket y se aplican a todos los objetos del bucket. Por ejemplo, una transmisión de IVS se almacena como varios objetos en un bucket de S3.	✓		
SDK	Kit de desarrollo de software: un conjunto de bibliotecas para los desarrolladores que crean aplicaciones con IVS.	✓	✓	✓
Segmentación de selfies	Permite sustituir el fondo en una transmisión en directo mediante una solución específica para el cliente que acepta una imagen de la cámara como entrada y devuelve una máscara que proporcio na una puntuación de confianza para cada píxel de la imagen, que indica si está en primer plano o en segundo. Consulte Reemplazo de fondo en SDK de transmisión de IVS: filtros de cámara de terceros (streaming en tiempo real).		✓	
Control de versiones semántico	Formato de versión con el formato VersiónPr incipal.VersiónSecundaria.Revisión. Las correccio nes de errores que no afectan a la API aumentan la versión de revisión, las adiciones o cambios de la API compatibles con versiones anteriores aumentan la versión secundaria y los cambios de la API incompatibles con versiones anteriores aumentan la versión principal.	✓	✓	✓

Plazo	Descripción	LL	RT	Chat
Composición del servidor	Utiliza un servidor de IVS para mezclar el audio y el video de todos los participantes del escenario y, a continuación, envía este video mezclado a un canal de IVS (por ejemplo, para llegar a un público más amplio) o a un bucket de S3. La composición del servidor reduce la carga de clientes, mejora la resiliencia de la transmisión y permite un uso más eficiente del ancho de banda. Consulte también Composición del cliente.		✓	
Service Quotas	Un servicio de AWS que lo ayuda a administrar las cuotas de muchos servicios de AWS desde una sola ubicación. Además de buscar los valores de cuota, también puede solicitar un aumento de cuota desde la consola de Service Quotas.	✓	✓	✓
Rol vinculado a servicio	Un tipo único de rol de IAM que está vinculado directamente a un servicio de AWS. IVS crea automáticamente los roles vinculados a servicios e incluyen todos los permisos que el servicio requiere para llamar a otros servicios de AWS en su nombre, por ejemplo, a un bucket de S3. Consulte Uso de roles vinculados a servicios para IVS en Seguridad de IVS.	✓		
Escenario	Un recurso de IVS que representa un espacio virtual en el que los participantes del evento en tiempo real pueden intercambiar videos también en tiempo real. Consulte Creación de un escenario en Introducción a streaming en tiempo real de IVS.		✓	

Plazo	Descripción	LL	RT	Chat
Sesión de escenario	La sesión de un escenario comienza cuando el primer participante se une a un escenario y finaliza unos minutos después de que el último participa nte deje de publicar en él. Un escenario de larga duración puede tener varias sesiones a lo largo de su vida útil.		✓	
De streaming	Datos que representan contenido de video o audio que se envían de forma continua de un origen a un destino.	✓	✓	
Clave de transmisi ón	Identificador asignado por IVS cuando se crea un canal y se utiliza para autorizar la transmisión. Trate la clave de transmisión como un secreto, ya que permite a cualquiera que la tenga transmitir en el canal. Consulte Introducción al streaming de baja latencia de IVS.	✓		
Transmisión en inanición	Un retraso o una interrupción en la entrega de la transmisión a IVS. Se produce cuando IVS no recibe la cantidad esperada de bits que el dispositi vo de codificación anunció que enviaría en un periodo determinado. Si se produce una transmisi ón en inanición, se produce un evento de transmisi ón en inanición. Desde la perspectiva del espectador, la transmisi ón en inanición puede manifestarse como un video que se retrasa, se almacena en búfer o se congela. La transmisión en inanición puede ser breve (menos de 5 segundos) o larga (varios minutos), según la situación específica que la provocó. Consulte ¿Qué es la transmisión en inanición? en Preguntas frecuentes de solución de problemas.	✓	✓	

Plazo	Descripción	LL	RT	Chat
Streamer	Persona o dispositivo que envía una transmisión de video o audio a IVS.	✓	✓	
Suscriptor	Participante de un evento en tiempo real que recibe video o audio del host. Consulte ¿Qué es el streaming en tiempo real de IVS?.		✓	
Etiqueta	Una etiqueta de metadatos que asigna a un recurso de AWS. Las etiquetas pueden ayudarlo a identificar y organizar sus recursos de AWS. En la página de inicio de la documentación de IVS, consulte la sección “Etiquetado” en cualquier documentación de la API de IVS (para la transmisi ón en tiempo real, la transmisión de baja latencia o el chat).	✓	✓	✓
Filtros de cámara de terceros	Componentes de software que se pueden integrar con el SDK de transmisión de IVS para permitir que una aplicación procese las imágenes antes de proporcionarlas este como origen de imágenes personalizado. Un filtro de cámara de terceros puede procesar las imágenes de la cámara, aplicar un efecto de filtro, etc.	✓	✓	
Miniatura	Imagen de tamaño reducido tomada de una transmisión. De forma predeterminada, las miniaturas se generan cada 60 segundos, pero se puede configurar un intervalo más corto. La resolución de las miniaturas depende del tipo de canal. Consulte Grabación de contenidos en Grabación automática en Amazon S3 (streaming de baja latencia).	✓		

Plazo	Descripción	LL	RT	Chat
Metadatos cronometrados	Metadatos vinculados a marcas de tiempo específicas en una transmisión. Se pueden agregar mediante programación con la API de IVS y se asocian a fotogramas específicos. Esto garantiza que todos los espectadores reciben los metadatos en el mismo punto de la transmisión. Los metadatos cronometrados se pueden utilizar para activar acciones en el cliente, como actualiza r las estadísticas del equipo durante un evento deportivo. Consulte Incorporación de metadatos en una transmisión de video.	✓		
Transcodificación	Convierte video y audio de un formato a otro. Una transmisión entrante puede ser transcodificada a un formato diferente con múltiples velocidades de bits y resoluciones, para admitir una variedad de dispositivos de reproducción y condiciones de red.	✓	✓	
Transmuxing	Un simple reempaquetado de una transmisión ingerida a Amazon IVS, sin volver a codificar la transmisión de video. “Transmux” es la abreviatu ra en inglés de multiplexación de transcodificación, un proceso que cambia el formato de un archivo de audio o video manteniendo algunas o todas las transmisiones originales. Transmuxing se convierte a un formato de contenedor diferente sin cambiar el contenido del archivo. Se distingue de la transcodi ficación.	✓	✓	

Plazo	Descripción	LL	RT	Chat
Transmisiones variantes	Conjunto de codificaciones de la misma transmisión en distintos niveles de calidad. Cada transmisión variante se codifica como una lista de reproducción HLS independiente. Un índice de las transmisiones variantes disponibles se denomina lista de reproducción multivariante. Una vez que el reproductor de IVS recibe una lista de reproducción multivariante de IVS, puede elegir entre las transmisiones variantes durante la reproducción y cambiar sin problemas entre ellas a según cambien las condiciones de la red.	✓		
VBR	Velocidad de bits variable (VBR): un método de control de velocidad para codificadores que utiliza una velocidad de bits dinámica que cambia durante la reproducción en función del nivel de detalle necesario. Recomendamos encarecidamente no utilizar VBR debido a problemas con la calidad de video; en su lugar, utilice CBR.	✓	✓	

Plazo	Descripción	LL	RT	Chat
Vista	Una sesión de visualización única que descarga o reproduce videos de forma activa. Las vistas son la base de la cuota de vistas simultáneas. Una vista se inicia cuando una sesión de visualiza ción comienza la reproducción de video. Una vista finaliza cuando una sesión de visualización detiene la reproducción de video. La reproducción es el único indicador de la audiencia; no se tienen en cuenta las heurísticas de interacción, como los niveles de audio, el enfoque de la pestaña del navegador y la calidad del video. Al contar las vistas, IVS no tiene en cuenta la legitimidad de los espectadores individuales ni intenta desduplicar la audiencia localizada, como varios reproductores de video en un solo equipo. Consulte Otras cuotas en Service Quotas (streaming de baja latencia).	✓		
Visor	Una persona que recibe una transmisión de IVS.	✓		
WebRTC	Web Real-Time Communication: un proyecto de código abierto que proporciona comunicación en tiempo real a los navegadores web y aplicaciones móviles. Permite que la comunicación de audio y video funcione dentro de las páginas web al permitir la comunicación directa entre pares, lo que elimina la necesidad de instalar complementos o descargar aplicaciones nativas. Las tecnologías detrás de WebRTC se implement an como un estándar web abierto y están disponibl es como API de JavaScript normales en los principales navegadores o como bibliotecas para clientes nativos (por ejemplo, Android e iOS).	✓	✓	

Plazo	Descripción	LL	RT	Chat
WHIP	Protocolo de ingesta HTTP WebRTC, un protocolo basado en HTTP que permite la ingesta de contenido basada en WebRTC en servicios de transmisión o CDN. WHIP es un borrador del IETF desarrollado para estandarizar la ingesta de WebRTC. WHIP habilita la compatibilidad con software como OBS y ofrece una alternativa (al SDK de transmisi ón de IVS) para la autoedición. Es posible que los streamers más sofisticados estén familiarizados con OBS y lo prefieran por sus características de producción avanzadas, como las transiciones de escenas, la mezcla de audio y la superposición de gráficos. WHIP también resulta útil en situaciones en las que utilizar el SDK de transmisión de IVS no es viable o no es la opción preferida. Por ejemplo, en configura ciones que incluyen codificadores de hardware, el SDK de transmisión de IVS podría no ser una opción. Sin embargo, si el codificador es compatibl e con WHIP, puede seguir publicando directamente desde el codificador en IVS. Consulte OBS and WHIP Support.		✓	
WSS	WebSocket Secure: un protocolo para establece r WebSockets a través de una conexión TLS cifrada. Se utiliza para conectarse a los puntos de conexión de Chat de IVS. Consulte Step 4: Send and Receive Your First Message en Getting Started with IVS Chat.			✓

Historial de documentos (Chat)

Cambios en la Guía del usuario de Chat

Cambio	Descripción	Fecha
Se dividió una guía de usuario de Chat	Esta versión incluye cambios importantes en la documentación. Trasladamos la información del chat de la guía del usuario de la transmisión de baja latencia de IVS a una nueva guía del usuario de Chat de IVS, que se encuentra en la sección Chat de IVS existente en la página de inicio de la documentación de IVS. Para ver otros cambios en la documentación, consulte Historial del documento (transmisión de baja latencia).	28 de diciembre de 2023
Glosario de IVS	Se amplió el glosario para incluir los términos de transmisiones en tiempo real o de baja latencia y chat de IVS.	20 de diciembre de 2023

Cambios en la Referencia de la API de Chat de IVS

Cambio de la API	Descripción	Fecha
Se dividió una guía de usuario de Chat	Ahora que hay una Guía del usuario de Chat de IVS (creada en esta versión), a partir de ahora las entradas de la Referencia de la API de Chat de IVS y la Referencia de la API de mensajería de Chat de IVS se ubicarán aquí. Las entradas anteriores del historial de estas referencias de la API de Chat están en Document History (Low-Latency Streaming).	28 de diciembre de 2023

Notas de la versión (Chat)

28 de diciembre de 2023

Guía del usuario de Chat de Amazon IVS

Chat de Amazon Interactive Video Service (IVS) es una característica de chat en directo administrada para acompañar las transmisiones de video en directo. En esta versión, trasladamos la información del chat de la Guía del usuario de Transmisión de baja latencia de IVS a la nueva Guía del usuario de Chat de IVS. Se puede acceder a la documentación desde la página de inicio de la documentación de Amazon IVS.

31 de enero de 2023

SDK de mensajería para clientes de Chat de Amazon IVS para Android 1.1.0

Plataforma	Descargas y cambios
SDK de mensajería para clientes de Chat de Android 1.1.0	Documentación de referencia: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/ • Para admitir las corrutinas de Kotlin, agregamos nuevas API de mensajería de Chat de IVS en el paquete de com.amazonaws.ivs.chat.messaging.coroutines. Consulte también el nuevo tutorial de corrutinas de Kotlin; la parte 1 (de 2) es Salas de chat.

Tamaño del SDK de mensajería para clientes de Chat: Android

Arquitectura	Tamaño comprimido	Tamaño sin comprimir
Todas las arquitecturas (código de bytes)	89 KB	92 KB

9 de noviembre de 2022

SDK de mensajería para clientes de Chat de Amazon IVS: JavaScript 1.0.2

Plataforma	Descargas y cambios
SDK de mensajería para clientes de Chat de JavaScript 1.0.2	Documentación de referencia: https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/ • Se corrigió un problema que afectaba a Firefox: los clientes recibían, de forma incorrecta, un error de socket al desconectarse de una sala de chat mediante el punto de conexión DisconnectUser.

8 de septiembre de 2022

SDK de mensajería para clientes de Chat de Amazon IVS: Android 1.0.0 e iOS 1.0.0

Plataforma	Descargas y cambios
SDK de mensajería para clientes de Chat de Android 1.0.0	Documentación de referencia: https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/
SDK de mensajería para clientes de Chat de iOS 1.0.0	Documentación de referencia: https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/

Tamaño del SDK de mensajería para clientes de Chat: Android

Arquitectura	Tamaño comprimido	Tamaño sin comprimir
Todas las arquitecturas (código de bytes)	53 KB	58 KB

Tamaño del SDK de mensajería para clientes de Chat: iOS

Arquitectura	Tamaño comprimido	Tamaño sin comprimir
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