aws

聊天功能使用者指南

Amazon IVS

Amazon IVS: 聊天功能使用者指南

Table of Contents

什麼是 Amazon IVS 聊天功能

Amazon IVS 聊天功能是一種受管的伴隨即時影片串流的即時聊天功能。如需文件，請造訪 Amazon IVS 聊天功能章節的 Amazon IVS 文件登陸頁面：

- 聊天功能使用者指南 – 本文件以及導覽窗格中列出的所有其他「使用者指南」頁面。
- Chat API 參考 - 控制平面 API (HTTPS)。
- Chat 訊息 API 參考 - 資料平面 API (WebSocket)
- 聊天用戶端的 SDK 參考：Android、iOS 和 JavaScript。

Amazon IVS 聊天功能入門

Amazon Interactive Video Service (IVS) Chat 是一種受管的伴隨即時影片串流的即時聊天功能。(IVS 聊天功能也可以在沒有影片串流的情況下使用。) 您可以建立聊天室並啟用使用者之間的聊天工作階段。

Amazon IVS 聊天功能可讓您專注於打造伴隨即時影片的自訂聊天體驗。您無需管理基礎設施，或開發和設定聊天工作流程的元件。Amazon IVS 聊天功能可擴展、安全、可靠且具有成本效益。

Amazon IVS 聊天功能最適合在有頭有尾的即時影片串流中，方便參與者之間的訊息傳遞。

本文件的其餘部分將引導您使用 Amazon IVS 聊天功能建置您的第一個聊天應用程式。

範例:可使用以下示範 (三個範例用戶端應用程式以及用於建立字符的後端伺服器應用程式)：

- Amazon IVS 聊天功能網頁試用版
- Amazon IVS 聊天功能 Android 試用版
- Amazon IVS 聊天功能 iOS 試用版
- Amazon IVS 聊天功能後端試用版

重要：24 個月沒有新連線或更新的聊天室將被自動刪除。

主題

步驟 1：執行初始設定

繼續進行之前，您必須：

1. 建立 AWS 帳戶。
2. 設定根使用者和管理使用者。

3. 設定 AWS IAM (Identity and Access Management) 許可。使用下面指定的政策。

如需上述各項的具體步驟，請參閱 Amazon IVS 使用者指南中的 IVS 低延遲串流入門。重要事項：在「步驟 3：設定 IAM 許可」中，請將此政策用於 IVS Chat：

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

```
      },
      {
         "Effect": "Allow",
         "Action": [
            "logs:CreateLogDelivery",
            "logs:GetLogDelivery",
            "logs:UpdateLogDelivery",
            "logs:DeleteLogDelivery",
            "logs:ListLogDeliveries",
            "logs:PutResourcePolicy",
            "logs:DescribeResourcePolicies",
            "logs:DescribeLogGroups",
            "s3:PutBucketPolicy",
            "s3:GetBucketPolicy",
            "iam:CreateServiceLinkedRole",
            "firehose:TagDeliveryStream"
         ],
         "Resource": "*"
      }
   ]
}
```

步驟 2：建立聊天室

Amazon IVS 聊天室具有與之關聯的組態資訊 (例如，最大訊息長度)。

本節中的指示說明如何使用主控台或 AWS CLI 設定聊天室 (包括檢閱訊息和/或記錄訊息的選用設定) 以及建立聊天室。

主控台說明

這些步驟分為幾個舞台，從初始聊天室設定開始，到最後建立聊天室結束。

您可以選擇對聊天室進行設定以便審查訊息。例如，您可以更新訊息內容或中繼資料、拒絕訊息以阻止它們被傳送，或者讓原始訊息能夠傳遞。這在設定以審查聊天室訊息 (選用) 中進行了介紹。

同時，您可以選擇對聊天室進行設定，以便記錄訊息。例如，如果您有訊息傳送至聊天室，您可以將其記錄至 Amazon S3 儲存貯體、Amazon CloudWatch 或 Amazon Kinesis Data Firehose。這在設定以記錄訊息 (選用) 中進行了介紹。

初始聊天室設定

1. 開啟 Amazon IVS 聊天功能主控台。

   (您也可以透過 AWS 管理主控台來存取 Amazon IVS 主控台。)

2. 從導覽列中，使用 Select a Region (選擇區域) 下拉式清單選擇區域。將會在此區域中建立您的新聊天室。

3. 在 Get started (開始使用) 方塊 (右上角) 中，選擇 Amazon IVS Chat Room (Amazon IVS 聊天室)。Create room (建立聊天室) 視窗隨即出現。

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. 在 Setup (設定) 下，可選擇指定 Room name (聊天室名稱)。聊天室名稱不是唯一的，但它們可讓您區分聊天室 ARN (Amazon 資源名稱) 以外的聊天室。
5. 在 Setup > Room configuration (設定 > 聊天室組態) 下，要麼接受 Default configuration (預設組態)，要麼選取 Custom configuration (自訂組態) 然後設定 Maximum message length (最大訊息長度) 和/或 Maximum message rate (最大訊息速率)。
6. 如果您想對訊息進行審查，請繼續下面的設定以審查聊天室訊息 (選用) 步驟。否則，略過該步驟 (即，接受 Message Review Handler > Disabled (訊息審查處理常式 > 停用)) 並直接進行 Final Room Creation (最終聊天室建立)。

設定以審查聊天室訊息 (選用)

1. 在 Message Review Handler (訊息審查處理常式) 下，選取 Handle with AWS Lambda (使用 AWS Lambda 處理)。Message Review Handler (訊息審查處理常式) 區段會展開以顯示額外選項。
2. 設定 Fallback result (後援結果)，指示當處理常式未傳回有效回應、遇到錯誤或超過逾時期限時，要 Allow (允許) 或是 Deny (拒絕) 訊息。
3. 指定現有的 Lambda function (Lambda 函數) 或使用 Create Lambda function (建立 Lambda 函數) 建立一個新的函數。

   所使用的 Lambda 函數必須和聊天室位於相同的 AWS 區域和 AWS 帳戶中。您應該授予 Amazon 聊天開發套件服務叫用 lambda 資源的許可。系統將為您選取的 lambda 函數自動建立以資源為基礎的政策。如需許可的相關詳細資訊，請參閱適用於 Amazon IVS 聊天功能的資源型政策。

設定以記錄訊息 (選用)

1. 在 Message logging (訊息記錄) 下，選取 Automatically log chat messages (自動記錄聊天訊息)。Message logging (訊息記錄) 區段會展開以顯示額外選項。您可以將現有的記錄組態新增至此聊天室，或選取 Create logging configuration (建立記錄組態) 來建立新的記錄組態。
2. 如果您選擇現有的記錄組態，則會出現下拉式功能表，其中會顯示您已建立的所有記錄組態。請從清單中選取一個記錄組態，您的聊天訊息將自動記錄到此目的地。
3. 如果您選擇 Create logging configuration (建立記錄組態)，則會出現強制回應視窗，讓您建立和自訂新的記錄組態。
   a. 選擇性指定 Logging configuration name (記錄組態名稱)。記錄組態名稱 (例如聊天室名稱) 並非唯一，但可讓您區分記錄組態 ARN 以外的記錄組態。

b. 在 Destination (目的地) 下，選取 CloudWatch log group (CloudWatch 日誌群組)、Kinesis firehose delivery stream (Kinesis Firehose 交付串流) 或 Amazon S3 bucket (Amazon S3 儲存貯體)，以選擇日誌的目的地。

c. 請根據自己的目的地，選取建立新的選項或使用現有的 CloudWatch log group (CloudWatch 日誌群組)、Kinesis firehose delivery stream (Kinesis Firehose交付串流) 或 Amazon S3 bucket (Amazon S3 儲存貯體)。

d. 審查後，選擇 Create (建立) 建立具有唯一 ARN 的新記錄組態。這會自動將新的記錄組態附加至聊天室。

最終聊天室建立

1. 審查後，選擇 Create chat room (建立聊天室) 建立一個具有唯一 ARN 的新聊天室。

CLI 說明

建立聊天室

使用 AWS CLI 建立聊天室是進階選項，需要您先在機器上下載並設定 CLI。如需詳細資訊，請參閱《AWS 命令列介面使用者指南》。

1. 執行聊天 `create-room` 命令並傳入選用名稱：

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

2. 這將傳回一個新的聊天室：

```
{
   "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. 記下 `arn` 欄位的內容。您將需要它來建立用戶端符記並連線至該聊天室。

設定記錄組態 (選用)

與建立聊天室一樣，使用 AWS CLI 設定記錄組態是一個進階選項，需要您先在電腦上下載並設定 CLI。如需詳細資訊，請參閱《AWS 命令列介面使用者指南》。

1. 執行聊天 `create-logging-configuration` 命令，並傳遞選用名稱和依名稱指向 Amazon S3 儲存貯體的目的地組態。此 Amazon S3 儲存貯體必須存在，才能建立記錄組態。(如需建立 Amazon S3 儲存貯體的詳細資訊，請參閱 Amazon S3 文件。)

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

2. 這將傳回新的記錄組態：

```
{
   "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. 記下 `arn` 欄位的內容。您需要用此資料將記錄組態附加至聊天室。

   a. 若要建立新的聊天室，請執行 `create-room` 命令並傳遞記錄組態 `arn`：

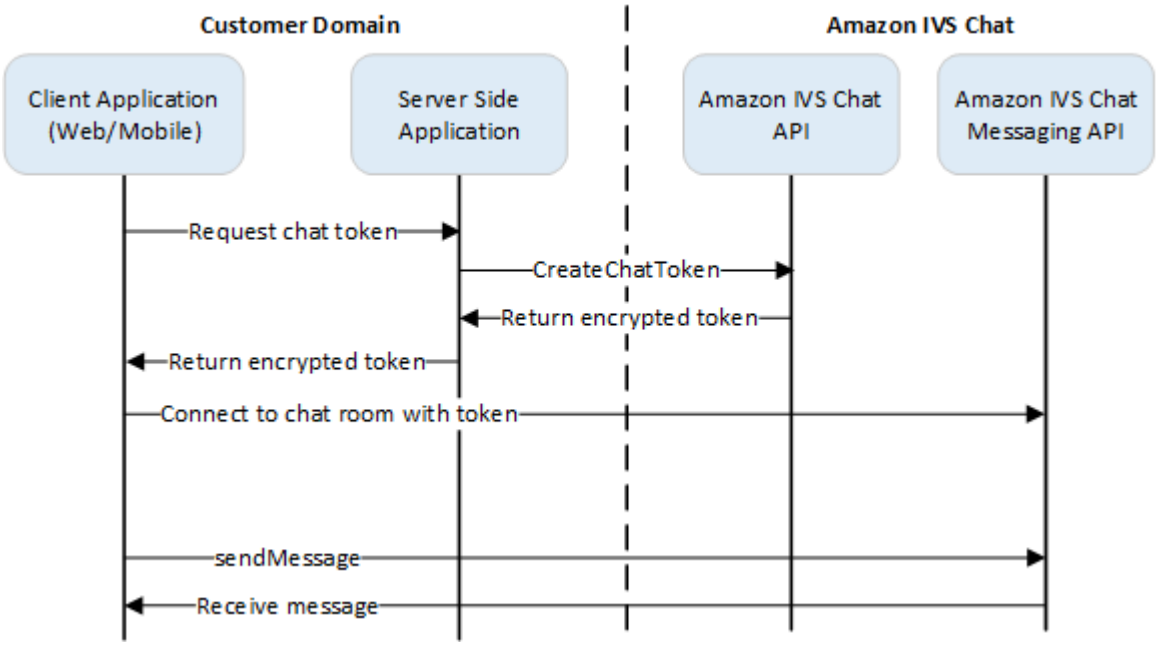
```
aws ivschat create-room --name test-room \
--logging-configuration-identifiers \
"arn:aws:ivschat:us-west-2:123456789012:logging-configuration/ABcdef34ghIJ"
```

   b. 若要更新現有的聊天室，請執行 `update-room` 命令並傳遞記錄組態 `arn`：

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

```
"arn:aws:ivschat:us-west-2:123456789012:logging-configuration/ABcdef34ghIJ"
```

步驟 3：建立聊天字符



聊天參與者要連線至聊天室並開始傳送及接收訊息，必須先建立聊天字符。聊天字符的功用在於驗證及授權聊天用戶端。如上所示，用戶端應用程式會向您的伺服器端應用程式索取字符，伺服器端應用程式會使用 AWS 開發套件或 Sigv4 簽署的請求呼叫 CreateChatToken。由於我們使用 AWS 憑證呼叫 API，因此應在安全的伺服器端應用程式中產生字符，而不是在用戶端應用程式中產生。

可產生字符的試用版後端伺服器應用程式請到 Amazon IVS 聊天功能後端試用版下載。

工作階段持續時間是指已建立的工作階段在自動關閉之前，維持作用中狀態的時間。即，工作階段持續時間是用戶端在必須產生新符記並建立新連線之前可以保持連線至聊天室的時間。建立字符的過程中，您可以選擇指定工作階段的持續時間。

每個字符只能為一名最終使用者建立連線一次。如果連線關閉，則必須建立新字符，才能重新建立連線。字符的有效期限以回應中的字符過期時間戳記為準。

最終使用者想連線至聊天室時，用戶端應向伺服器應用程式索取字符。伺服器應用程式會建立字符，並將字符傳遞回用戶端。最終使用者一旦提出需求，即應為其建立字符。

若要建立聊天驗證字符，請依以下指示操作。建立聊天字符時，請使用請求欄位傳遞聊天最終使用者和最終使用者傳訊功能的相關資料；如需詳細資訊，請參閱 IVS 聊天功能 API 參考資料中的 CreateChatToken。

AWS SDK 說明

若要使用 AWS 開發套件建立聊天字符，您必須先在應用程式上下載並設定開發套件。以下是使用 JavaScript 之 AWS SDK 的說明。

重要：此程式碼必須在伺服器端執行，再將其輸出傳遞給用戶端。

必要條件：若要使用以下程式碼範例，您需要將 AWS JavaScript SDK 載入應用程式中。如需詳細資訊，請參閱 適用於 JavaScript 的 AWS SDK 入門。

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

CLI 說明

使用 AWS CLI 建立聊天符記是進階選項，需要您先在機器上下載並設定 CLI。如需詳細資訊，請參閱《AWS 命令列介面使用者指南》。注意：使用 AWS CLI 產生符記適合在測試時使用，但對於生產用途，我們建議您使用 AWS 開發套件在伺服器端產生符記 (請參閱上述說明)。

1. 執行 `create-chat-token` 命令並使用用戶端的聊天室識別符和使用者 ID。納入任何這些功能："SEND_MESSAGE"、"DELETE_MESSAGE"、"DISCONNECT_USER"。(您還可以選擇設定工作階段持續時間 (以分鐘為單位) 和/或有關此聊天工作階段的自訂屬性 (中繼資料)。這些欄位未在下面顯示。)

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

2. 這將傳回一個用戶端符記：

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

3. 儲存此符記。您將需要它來連線至聊天室以及傳送或接收訊息。您需要在工作階段結束 (由 `sessionExpirationTime` 指示) 前產生另一個聊天符記。

步驟 4：傳送和接收第一條訊息

使用您的聊天字符連線至聊天室，並傳送您的第一條訊息。下面提供了 JavaScript 程式碼範例。您也可使用 IVS 用戶端 SDK：請參閱聊天 SDK：Android 版指南、聊天 SDK：iOS 版指南和聊天 SDK：JavaScript 版指南。

區域服務：下面的程式碼範例參考您「所選擇的受支援區域」。Amazon IVS 聊天功能提供區域性端點，您可以用來提出請求。對於Amazon IVS 聊天功能訊息 API，區域端點的一般語法為：

wss://edge.ivschat.<區域代碼>.amazonaws.com

例如，wss://edge.ivschat.us-west-2.amazonaws.com 是位於美國西部 (奧勒岡) 區域的端點。如需支援的區域清單，請參閱 AWS 一般參考中的 Amazon IVS 頁面上的 Amazon IVS 聊天功能資訊。

```
/*
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);
}
```

恭喜您！所有步驟均已完成！您現在擁有了一個可以傳送或接收訊息的簡便的聊天應用程式。

步驟 5：檢查服務配額限制 (選用)

您的聊天室將與您的 Amazon IVS 即時串流一起擴展，以使所有觀眾能夠參與聊天對話。但是，所有 Amazon IVS 帳戶對並行聊天參與者的數量和訊息傳遞速率都有限制。

確保您的限制足夠，並在需要時請求增加，特別是當您正在規劃大型串流事件時。如需詳細資訊，請參閱 Service Quotas (低延遲串流)、Service Quotas (即時串流) 和 Service Quotas (聊天功能)。

聊天記錄

聊天記錄功能可讓您將聊天室中的所有訊息記錄到三個標準位置中的任何一個：Amazon S3 儲存貯體、Amazon CloudWatch Logs 或 Amazon Kinesis Data Firehose。接著，可使用日誌進行分析或建置聊天重播，該聊天重播會連接到即時影片工作階段。

啟用聊天室的聊天記錄

聊天記錄是一種進階選項，可以透過將日誌組態與聊天室建立關聯來加以啟用。記錄組態是一種資源，可讓您指定聊天室訊息的記錄位置類型 (Amazon S3 儲存貯體、Amazon CloudWatch Logs 或 Amazon Kinesis Data Firehose)。如需有關建立和管理記錄組態的詳細資訊，請參閱 Amazon IVS 聊天功能入門和 Amazon IVS 聊天功能 API Reference (《Amazon IVS 聊天功能 API 參考》)。

建立新聊天室 (CreateRoom) 或更新現有聊天室 (UpdateRoom) 時，您最多可以將三個記錄組態與每個聊天室建立關聯。您可以將多個聊天室與相同的記錄組態建立關聯。

當至少一個作用中記錄組態與聊天室相關聯時，都會將透過 Amazon IVS 聊天功能傳訊 API 傳送到該聊天室的每個傳訊請求自動記錄到指定的位置。以下是傳播延遲的平均值 (從傳送傳訊請求到傳訊請求在指定位置可用時)：

- Amazon S3 儲存貯體：5 分鐘
- Amazon CloudWatch Logs 或 Amazon Kinesis Data Firehose：10 秒

訊息內容

格式

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

欄位

欄位	描述
event_timestamp	Amazon IVS 聊天功能收到訊息的 UTC 時間戳記。
payload	用戶端將從 Amazon IVS 聊天功能服務收到的 Message (Subscribe) 或 Event (Subscribe) JSON 承載。
type	聊天訊息的類型。 • 有效值: MESSAGE | EVENT
version	訊息內容格式的版本。

Amazon S3 儲存貯體

格式

會以下列 S3 字首和檔案格式對訊息日誌進行組織和儲存：

```
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
```

欄位

欄位	描述
<account_id>	從中建立聊天室的 AWS 帳戶 ID。
<hash>	由系統產生的以確保唯一性的雜湊值。
<region>	聊天室建立所在的 AWS 服務區域。
<resource_id>	聊天室 ARN 的資源 ID 部分。
<version>	訊息內容格式的版本。

欄位	描述
`<year> / <month> / <day> / <hours> / <minute>`	Amazon IVS 聊天功能收到訊息的 UTC 時間戳記。

範例

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

Amazon CloudWatch Logs

格式

會以下列日誌串流名稱格式對訊息日誌進行組織及儲存：

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

欄位

欄位	描述
`<resource_id>`	聊天室 ARN 的資源 ID 部分。
`<version>`	訊息內容格式的版本。

範例

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

Amazon Kinesis Data Firehose

會以即時串流資料將訊息日誌傳送至交付串流，目的地包含 Amazon Redshift、Amazon OpenSearch Service、Splunk，以及任何自訂 HTTP 端點或受支援的第三方服務供應商所擁有的 HTTP 端點。如需詳細資訊，請參閱什麼是 Amazon Kinesis Data Firehose。

限制

- 您必須擁有訊息儲存所在的記錄位置。
- 聊天室、記錄組態和記錄位置三者所在的 AWS 區域必須相同。
- 聊天記錄只能使用作用中的記錄組態。
- 您只能夠刪除不再與任何聊天室相關聯的記錄組態。

必須使用 AWS 憑證進行授權，才能將訊息記錄到您擁有的位置。為了向 IVS Chat 提供必要的存取權，建立記錄組態時會自動產生資源政策 (適用於 Amazon S3 儲存貯體或 CloudWatch 日誌) 或 AWS IAM 服務連結角色 (SLR) (適用於 Amazon Kinesis Data Firehose)。對角色或政策進行任何修改時請務必小心，因為這可能會影響聊天記錄的許可。

使用 Amazon CloudWatch 監控錯誤

您可以使用 Amazon CloudWatch 監控聊天記錄中發生的錯誤，並且可以建立警示或儀表板來指示或回應特定錯誤的變化。

錯誤類型有幾種。如需詳細資訊，請參閱監控 Amazon IVS 聊天功能。

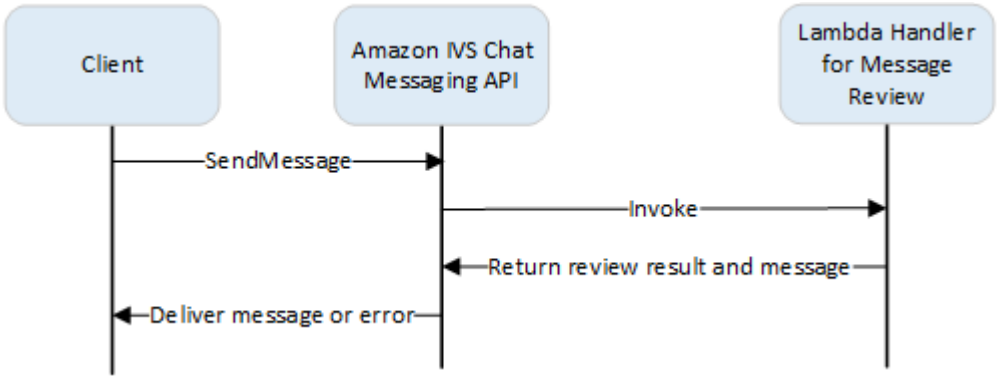
Chat 訊息審查處理常式

訊息審查處理常式可讓您在訊息傳遞到聊天室之前審查和/或修改訊息。當某個訊息審查處理常式與聊天室相關聯時，每次對該聊天室進行 SendMessage 請求時都會叫用它。該處理常式強制執行應用程式的業務邏輯，以確定是允許、拒絕還是修改訊息。Amazon IVS 聊天功能支援將 AWS Lambda 函數作為處理常式。

建立 Lambda 函數

在為聊天室設定訊息審查處理常式之前，您必須使用以資源為基礎的 IAM 政策建立一個 lambda 函數。該 lambda 函數必須與您將使用該函數的聊天室位於同一 AWS 帳戶和 AWS 區域中。該以資源為基礎的政策授予 Amazon IVS 聊天功能叫用 lambda 函數的許可。如需說明，請參閱適用於 Amazon IVS 聊天功能的資源型政策。

工作流程



請求語法

當用戶端傳送訊息時，Amazon IVS 聊天功能使用 JSON 承載叫用 lambda 函數：

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

```
    }
}
```

請求主體

欄位	描述
`Attributes`	與訊息相關聯的屬性。
`Content`	訊息的原始內容。
`MessageId`	訊息 ID。由 IVS Chat 產生。
`RoomArn`	訊息要傳送到的聊天室的 ARN。
`Sender`	與傳送者相關的資訊。此物件包含幾個欄位： • `Attributes` – 在身分驗證期間建立的與傳送者相關的中繼資料。可用於向用戶端提供有關傳送者的更多資訊；例如虛擬人偶 URL、徽章、字型和顏色。 • `UserId` – 傳送此訊息的檢視者 (最終使用者) 的應用程式指定識別符。用戶端應用程式可以使用它來參考訊息 API 或應用程式網域中的使用者。 • `Ip` - 傳送請求的用戶端 IP 地址。

回應語法

該處理常式 lambda 函數必須傳回具有以下語法的 JSON 回應。不符合以下語法或不滿足欄位限制的回應將無效。在這種情況下，允許或拒絕訊息取決於您在訊息審查處理常式中指定的 `FallbackResult` 值；請參閱《Amazon IVS 聊天功能 API 參考》中的 MessageReviewHandler。

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

回應欄位

欄位	描述
`Attributes`	與從 lambda 函數傳回的訊息關聯的屬性。 如果 `ReviewResult` 為 `DENY`，則可以在 `Reason` 中提供 `Attributes`；例如： `"Attributes": {"Reason": "denied for moderation` 在這種情況下，傳送者用戶端會收到一個 WebSocket 406 錯誤，並且錯誤訊息中會包含該原因。(請參閱 Amazon IVS 聊天功能訊息 API 參考中的 WebSocket 錯誤。) • 大小限制：最大 1 KB • 必要：否
`Content`	從 Lambda 函數傳回的訊息的內容。它可以根據業務邏輯進行編輯或保持原樣。 • 長度限制：長度下限為 1。建立/更新聊天室時定義的 `MaximumMessageLength` 的最大長度。如需詳細資訊，請參閱 Amazon IVS 聊天功能 API 參考。這僅適用於 `ReviewResult` 為 `ALLOW` (允許) 的情況。 • 必要：是
`ReviewResult`	關於如何處理訊息的審查結果。如果允許，則訊息會傳遞給連線至聊天室的所有使用者。如果拒絕，則訊息不會傳遞給任何使用者。 • 有效值: `ALLOW` | `DENY` • 必要：是

範例程式碼

下面是採用 Go 語言編寫的 lambda 處理常式範例。它修改訊息內容、保持訊息屬性不變，並允許訊息傳遞。

```
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
}
```

將處理常式與聊天室關聯和解除關聯

設定並實作 lambda 處理常式後，使用 Amazon IVS 聊天功能 API：

- 若要將處理常式與聊天室關聯，呼叫 CreateRoom 或 UpdateRoom 並指定處理常式。

- 若要解除處理常式與聊天室的關聯，使用值為空的 `MessageReviewHandler.Uri` 呼叫 UpdateRoom。

使用 Amazon CloudWatch 監控錯誤

您可以使用 Amazon CloudWatch 監控訊息審查中發生的錯誤，並且可以建立警示或儀表板來指示或回應特定錯誤的變化。如果發生錯誤，允許或拒絕訊息取決於您在將處理常式與聊天室關聯時指定的 `FallbackResult` 值；請參閱《Amazon IVS 聊天功能 API 參考》中的 MessageReviewHandler。

錯誤類型有幾種：

- `InvocationErrors` 在 Amazon IVS 聊天功能無法叫用處理常式時發生。
- `ResponseValidationErrors` 在處理常式傳回無效回應時發生。
- AWS Lambda `Errors` 在 lambda 處理常式被叫用的過程中傳回函數錯誤時發生。

如需有關調用錯誤和回應驗證錯誤 (由 Amazon IVS 聊天功能發出) 的詳細資訊，請參閱監控 Amazon IVS 聊天功能。如需有關 AWS Lambda 錯誤的詳細資訊，請參閱使用 Lambda 指標。

監控 Amazon IVS 聊天功能

您可以使用 Amazon CloudWatch 監控 Amazon Interactive Video Service (IVS) 聊天功能資源。CloudWatch 可收集並處理來自 Amazon IVS 聊天功能的原始資料，進而將這些資料轉換為便於讀取且幾近即時的指標。這些統計資料會保留 15 個月，以便您了解 Web 應用程式或服務效能的歷史。您可以設定特定閾值的警示，當滿足這些閾值時傳送通知或採取動作。如需詳細資訊，請參閱 CloudWatch 使用者指南。

存取 CloudWatch 指標

Amazon CloudWatch 可收集並處理來自 Amazon IVS 聊天功能的原始資料，進而將這些資料轉換為可讀取且幾近即時的指標。這些統計資料會保留 15 個月，以便您了解 Web 應用程式或服務效能的歷史。您可以設定特定閾值的警示，當滿足這些閾值時傳送通知或採取動作。如需詳細資訊，請參閱 CloudWatch 使用者指南。

請注意，CloudWatch 指標會隨著時間累計。隨著指標的時間變長，解析度會有效降低。排程如下：

- 60 秒的指標可供使用 15 天。
- 5 分鐘的指標可供使用 63 天。
- 1 小時的指標可供使用 455 天 (15 個月)。

如需有關資料保留的最新資訊，請在 Amazon CloudWatch 常見問答集中搜尋「保留期間」。

CloudWatch 主控台說明

1. 透過 https://console.aws.amazon.com/cloudwatch/ 開啟 CloudWatch 主控台。
2. 在側邊導覽中，展開 Metrics (指標) 下拉式選單，然後選取 All metrics (所有指標)。
3. 在 Browse (瀏覽)索引標籤上，使用左側無標籤的下拉式清單，選取建立頻道的「主要」區域。有關區域的更多資訊，請參閱全球解決方案、區域控制。如需支援的區域清單，請參閱 AWS 一般參考中的 Amazon IVS 頁面。
4. 在瀏覽索引標籤底部，選取 IVSChat 命名空間。
5. 執行以下任意一項：
   a. 在搜尋列中，輸入您的資源 ID (ARN 的一部分，`arn:::ivschat:room/<resource id>`)。

      然後選取 IVSChat。

b. 如果 IVSChat 在 AWS 命名空間下顯示為可選取的服務，則請選取它。如果您使用 Amazon IVSChat 並且其正在傳送指標到 Amazon CloudWatch，則會將其列出。(如果未列出 IVSChat，則您沒有任何 Amazon IVSChat 指標。)

   然後根據需要選擇維度分組；可用的維度會列在下方的 CloudWatch 指標。

6. 選擇要新增到圖表的指標。可用的指標列在 CloudWatch 的指標。

您也可以選取在 CloudWatch 中檢視按鈕，從聊天工作階段的詳細資訊頁面存取聊天工作階段的 CloudWatch 圖表。

CLI 說明

您也可以使用 AWS CLI 存取指標。這需要在您的機器上先下載並設定 CLI。如需詳細資訊，請參閱 AWS 命令列界面使用者指南。

然後，使用 AWS CLI 存取 Amazon IVS 低延遲聊天功能指標：

- 在命令提示中，執行：

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

如需詳細資訊，請參閱 Amazon CloudWatch 使用者指南中的使用 Amazon CloudWatch 指標。

CloudWatch 指標：IVS 聊天功能

Amazon IVS 聊天功能在 AWS/IVSChat 命名空間中提供以下指標。

指標	維度	描述
`ConcurrentChatConnections`	無	聊天室中的並行連線總數 (每分鐘報告的最大值)。這有助於了解一個區域中的客戶何時接近並行聊天連線數上限。 單位：計數 有效統計資訊：總和、平均數、上限、下限

指標	維度	描述
Deliveries	動作	向一個區域中所有聊天室的聊天連線傳送特定動作類型的訊息請求次數。 單位：計數 有效統計資訊：總和、平均數、上限、下限
InvocationErrors	Uri	一個區域中所有聊天室的特定訊息審查處理常式的叫用錯誤數。無法叫用訊息審查處理常式時會發生叫用錯誤。 當 Amazon IVS 聊天功能無法叫用處理常式時，會發生叫用錯誤。如果與聊天室關聯的處理常式不再存在或逾時，或者如果資源政策不允許服務叫用它，則會發生這種情況。 單位：計數 有效統計資訊：總和、平均數、上限、下限
LogDestinationAccessDeniedError	LoggingConfiguration	區域中所有聊天室的日誌目的地拒絕存取錯誤數量。 當 Amazon IVS 聊天功能無法存取您在記錄組態中指定的目的地資源時，即會發生這些錯誤。如果目的地資源政策不允許服務放置記錄，則可能會發生這種情況。 單位：計數 有效統計資訊：總和、平均數、上限、下限

指標	維度	描述
LogDestinationErrors	LoggingConfiguration	區域中所有聊天室的日誌目的地所有錯誤數量。 這是一個彙總指標，其中包括 Amazon IVS 聊天功能無法將日誌傳遞至您在記錄組態中指定的目的地資源時發生的所有錯誤類型。 單位：計數 有效統計資訊：總和、平均數、上限、下限
LogDestinationResourceNotFoundErrors	LoggingConfiguration	區域中所有聊天室的日誌目的地找不到資源錯誤數量。 當 Amazon IVS 聊天功能因為資源不存在而無法將日誌傳遞至您在記錄組態中指定的目的地資源時，即會發生這些錯誤。如果與記錄組態相關聯的目的地資源不再存在，即可能會發生這種情況。 單位：計數 有效統計資訊：總和、平均數、上限、下限
MessagingDeliveries	無	向一個區域中所有聊天室的聊天連線傳送訊息請求的次數。 單位：計數 有效統計資訊：總和、平均數、上限、下限
MessagingRequests	無	一個區域的所有聊天室發出的訊息請求數。 單位：計數 有效統計資訊：總和、平均數、上限、下限

指標	維度	描述
Requests	動作	一個區域的所有聊天室中針對特定動作類型提出的請求數。 單位：計數 有效統計資訊：總和、平均數、上限、下限
ResponseValidationErrors	Uri	一個區域中所有聊天室的特定訊息審查處理常式的回應驗證錯誤數。當訊息審查處理常式的回應無效時，會發生回應驗證錯誤。這可能意味著無法解析回應或驗證檢查失敗；例如，無效的審查結果或過長的回應值。 單位：計數 有效統計資訊：總和、平均數、上限、下限

Amazon IVS 聊天功能用戶端傳訊開發套件

Amazon 互動影片服務 (IVS) 聊天用戶端傳訊開發套件適用於使用 Amazon IVS 建置應用程式的開發人員。此開發套件的設計目的是利用 Amazon IVS 架構和 Amazon IVS 聊天功能，並推出改良後的新版功能。作為原生開發套件，其設計目的是將對您的應用程式和使用者存取應用程式的裝置的效能影響降至最低。

平台需求：

桌面瀏覽器

瀏覽器	支援的版本
Chrome	兩個主要版本 (目前版本和最新的先前版本)
Edge	兩個主要版本 (目前版本和最新的先前版本)
Firefox	兩個主要版本 (目前版本和最新的先前版本)
Opera	兩個主要版本 (目前版本和最新的先前版本)
Safari	兩個主要版本 (目前版本和最新的先前版本)

行動裝置瀏覽器

瀏覽器	支援的版本
適用於 Android 的 Chrome	兩個主要版本 (目前版本和最新的先前版本)
適用於 Android 的 Firefox	兩個主要版本 (目前版本和最新的先前版本)
適用於 Android 的 Opera	兩個主要版本 (目前版本和最新的先前版本)

瀏覽器	支援的版本
適用於 Android 的 WebView	兩個主要版本 (目前版本和最新的先前版本)
Samsung Internet	兩個主要版本 (目前版本和最新的先前版本)
適用於 iOS 的 Safari	兩個主要版本 (目前版本和最新的先前版本)

原生平台

平台	支援的版本
Android	5.0 版和更新版本
iOS	13.0 版和更新版本

支援

如果聊天室發生錯誤或其他問題，請透過 IVS Chat API 判斷聊天室專屬的識別碼是否正確)請參閱 ListRooms)。

將此聊天室的識別碼提供給 AWS Support。使用它，他們就可以取得資訊來協助您對問題進行疑難排解。

注意：關於可用版本以及已修正的問題，請參閱 Amazon IVS 聊天功能版本備註。如果適當，請在聯絡支援部門之前，先更新您的開發套件版本，並查看是否可以解決您的問題。

版本控制

Amazon IVS 聊天用戶端傳訊 SDK 使用語意版本控制。

對於此討論，假設：

- 最新版本為 4.1.3 版。
- 先前主要版本的最新版本為 3.2.4 版。

- 版本 1.x 的最新版本為 1.5.6 版。

回溯相容的新功能會新增為最新版本的次要版本。在這種情況下，下一組新功能將被新增為 4.2.0 版。

回溯相容的次要錯誤修正會新增為最新版本的修補程式版本。在這裡，下一組小錯誤修復將被新增為 4.1.4 版。

回溯相容、主要錯誤修正的處理方式不同；它們會新增至多個版本：

- 最新版本的修補程式版本。在這裡，它為 4.1.4 版。
- 先前次要版本的修補程式版本。在這裡，它為 3.2.5 版。
- 最新版 1.x 版本的修補程式版本。在這裡，它為 1.5.7 版。

主要錯誤修正由 Amazon IVS 產品團隊定義。典型範例包括重要的安全更新以及客戶所需的其他精選修正。

備註：在上面的範例中，發布的版本在不跳過任何數字的情況下遞增 (例如，從 4.1.3 到 4.1.4)。實際上，一個或多個修補程式編號可能會保持在內部並且不需要發行，因此發行的版本可能會從 4.1.3 增加到 4.1.6。

此外，在 2023 年底或發行 3.x 之前，將支援 1.x 版，以較晚者為準。

Amazon IVS 聊天功能開發套件

伺服器端 (不由開發套件管理) 有兩個 API，每個 API 都有自己負責的部分：

- 資料平面 — 此 IVS Chat 傳訊 API 是一種 WebSockets API，主要由以字符型身分驗證方案驅動的前端應用程式 (iOS、Android、macOS 等) 使用。透過先前產生的聊天字符，即可使用此 API 與現有的聊天室連線。

  Amazon IVS 聊天用戶端傳訊 SDK 僅與資料平面有關。開發套件會假設您已經透過後端產生聊天字符。擷取這些字符的工作應由前端應用程式管理，而非由開發套件管理。
- 控制平面 — 此 IVS Chat 控制平面 API 為您自己的後端應用程式提供了一個介面，使用此介面可管理和建立聊天室以及加入聊天室的使用者。可將這個平面視為應用程式聊天體驗 (由您自己的後端管理) 的管理面板。有些控制平面端點負責建立聊天字符，資料平面必須對這些字符進行身分驗證，通過驗證者才能進入聊天室。

  重要：IVS Chat 用戶端傳訊開發套件不會呼叫任何控制平面端點。您必須設定後端才能為您自己建立聊天字符。您的前端應用程式必須與後端通訊才能擷取此聊天字符。

《Amazon IVS 聊天用戶端傳訊 SDK：Android 版指南》

Amazon Interactive Video (IVS) Chat 用戶端傳訊 Android 版開發套件的介面可讓您輕鬆使用 Android 來整合我們平台上的 IVS 聊天傳訊 API。

`com.amazonaws:ivs-chat-messaging` 套件會實作本文件中所述的介面。

最新版 IVS 聊天用戶端傳訊 Android SDK：1.1.0 (版本備註)

參考文件：如需有關 Amazon IVS 聊天功能用戶端傳訊 Android 版開發套件中最重要方法的資訊，請參閱參考文件，網址為 https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/。

範本程式碼：請參閱 GitHub 上的 Android 範本儲存庫：https://github.com/aws-samples/amazon-ivs-chat-for-android-demo。

平台需求：開發需要 Android 5.0 (API level 21) 或更高版本。

開始使用

在開始使用之前，請先詳閱 Amazon IVS 聊天功能入門。

新增套件

將 `com.amazonaws:ivs-chat-messaging` 新增至您的 `build.gradle` 相依性：

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

新增 Proguard 規則

將以下項目新增至您的 R8/Proguard 規則檔案 (`proguard-rules.pro`) 中：

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

設定後端

伺服器上需有可與 Amazon IVS API 通訊的端點才能進行此整合。使用官方 AWS 程式庫從您的伺服器存取 Amazon IVS API。透過公開套件就可以使用這些程式庫，而且有多種程式語言可供選用；例如 node.js 和 Java。

接下來，建立一個可與 Amazon IVS 聊天功能 API 通訊的伺服器端點並建立權杖。

設定伺服器連線

建立採用 ChatTokenCallback 作為參數並從後端擷取聊天權杖的方法。將該權杖傳給回呼的 onSuccess 方法。如果發生錯誤，則將異常情況傳給回呼的 onError 方法。這是在下一個步驟中將主要 ChatRoom 實體執行個體化時所需的處置動作。

以下是使用 Retrofit 呼叫來實作上述動作的範例程式碼。

```
// ...

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

使用開發套件

初始化聊天室執行個體

建立 ChatRoom 類別的執行個體。這項作業需傳遞 regionOrUrl (通常是負責託管聊天室的 AWS 區域) 和 tokenProvider (在前一個步驟中建立的權杖擷取方法)。

```
val room = ChatRoom(
    regionOrUrl = "us-west-2",
```

```
    tokenProvider = ::fetchChatToken
)
```

接著請建立接聽程式物件，此物件會實作聊天相關事件的處理常式，並將其指派給 room.listener 屬性：

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

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

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

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

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

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

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

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

// ...
```

基本初始化的最後一步是透過建立 WebSocket 連線來連線至特定的聊天室。若要這樣做，請在聊天室執行個體中呼叫 connect() 方法。我們建議在 onResume() 生命週期方法中這樣做，以確保它在應用程式從背景中恢復時保持連線。

```
room.connect()
```

開發套件會試圖連線至聊天室，而該聊天室是以從伺服器收到的聊天權杖加以編碼。如果失敗，它會試圖重新連線，直至達到在聊天室執行個體中指定的次數為止。

在聊天室中執行動作

ChatRoom 類別的動作可用於傳送和刪除訊息以及中斷其他使用者的連線。這些動作接受的選用回呼參數可讓您收到請求確認或拒絕通知。

傳送訊息

若要提出此請求，您的聊天權杖編碼內容必須含有 SEND_MESSAGE 功能。

若要觸發傳送訊息請求，請執行下列動作：

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

若要收到請求確認/拒絕訊息，請提供回呼作為第二個參數：

```
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.
   }
})
```

刪除訊息

若要提出此請求，您的聊天權杖編碼內容必須含有 DELETE_MESSAGE 功能。

若要觸發刪除訊息請求，請執行下列動作：

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

若要收到請求確認/拒絕訊息，請提供回呼作為第二個參數：

```
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.
   }
})
```

中斷與其他使用者的連線

若要提出此請求，您的聊天權杖編碼內容必須含有 DISCONNECT_USER 功能。

若要為仲裁目的中斷其他使用者的連線，請執行下列動作：

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

若要收到請求確認/拒絕訊息，請輸入用於回呼的第二個參數：

```
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.
   }
})
```

中斷與聊天室的連線

若要關閉與聊天室的連線，請在聊天室執行個體中呼叫 disconnect() 方法：

```
room.disconnect()
```

由於應用程式處於背景狀態一會兒後，WebSocket 連線就會停止運作，因此建議您在從背景狀態來回轉換時手動連線/中斷連線。若要這樣做，則 Android Activity 或 Fragment 上 onResume() 生命

週期方法中的 `room.connect()` 呼叫必須與 `onPause()` 生命週期方法中的 `room.disconnect()` 呼叫相符。

Amazon IVS 聊天用戶端傳訊 SDK：Android 版教學課程第 1 部分：聊天室

這是由兩部分組成的教學課程的第一部分。透過使用 Kotlin 程式設計語言建置功能完整的 Android 應用程式，您將學習使用 Amazon IVS 聊天功能傳訊 SDK 的基礎知識。我們稱呼該應用程式為 Chatterbox。

在開始該模組之前，請花幾分鐘時間熟悉先決條件、聊天權杖背後的重要概念以及建立聊天室所需的後端伺服器。

這些教學課程專為經驗豐富的 Android 開發人員而建立，他們不熟悉 IVS 聊天功能傳訊 SDK。您將需要熟悉 Kotlin 程式設計語言並在 Android 平台上建立 UI。

本教學課程的第一部分分為幾個部分：

1. the section called “設定本機身分驗證/授權伺服器”
2. the section called “建立 Chatterbox 專案”
3. the section called “連線到聊天室並觀察連線更新”
4. the section called “建立字符提供者”
5. the section called “後續步驟”

如需完整的 SDK 文件，請先閱讀 Amazon IVS 聊天用戶端傳訊 SDK (載於《Amazon IVS 聊天功能使用者指南》中) 和 Chat Client Messaging: SDK for Android Reference (聊天用戶端傳訊：Android 版 SDK 參考) (位於 GitHub 上)。

必要條件

- 熟悉 Kotlin 並在 Android 平台上建立應用程式。如果您不熟悉如何為 Android 建立應用程式，請在適用於 Android 開發人員的建置您的第一個應用程式指南中了解基礎知識。
- 仔細閱讀並理解 IVS 聊天功能入門。
- 使用現有 IAM 政策中定義的 `CreateChatToken` 和 `CreateRoom` 功能建立 AWS IAM 使用者。(請參閱 IVS 聊天功能入門)。
- 確保將此使用者的私密/存取金鑰儲存在 AWS 憑證檔案中。如需指示，請參閱《AWS CLI 使用者指南》(特別是組態和憑證檔案設定)。

- 建立聊天室並保存其 ARN。請參閱IVS 聊天功能入門。(如果您未保存該 ARN，稍後可以使用主控台或 Chat API 來查詢。)

設定本機身分驗證/授權伺服器

後端伺服器負責建立聊天室並產生 IVS 聊天功能 Android SDK 需要的聊天權杖，以便對聊天室的用戶端執行身分驗證和授權。

請參閱 Amazon IVS 聊天功能入門中的建立聊天字符。如流程圖所示，您的伺服器端程式碼會負責建立聊天權杖。這意味著應用程式必須透過從伺服器端應用程式請求聊天字符，來提供自己產生聊天字符的方法。

我們使用 Ktor 架構建立即時本機伺服器，以管理使用本機 AWS 環境建立聊天權杖的作業。

此時，希望您已正確設定 AWS 憑證。如需逐步說明，請參閱設定適用於開發的 AWS 憑證和區域。

建立新目錄並將其命名為 `chatterbox`，然後在其內部再建立另一個名為 `auth-server` 的目錄。

伺服器資料夾的結構如下：

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

注意：您可以直接將這裡的程式碼複製/貼上到參考的檔案中。

接下來，我們會新增所有必要的相依性和外掛程式，以使 auth 伺服器正常工作：

Kotlin 指令碼：

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

現在，我們需要為 auth 伺服器設定記錄功能。(如需詳細資訊，請參閱設定記錄器)。

XML：

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

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

Ktor 伺服器需要組態設定，其會自動從 `resources` 目錄中的 `application.*` 檔案中載入，因此我們也會新增這些項目。(如需詳細資訊，請參閱檔案中的組態。)

HOCON：

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

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

最後，讓我們來實作伺服器：

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()
            )
         )
      }
   }
}
```

建立 Chatterbox 專案

若要建立 Android 專案，請安裝並開啟 Android Studio。

請按照 Android 官方建立專案指南中列出的步驟進行操作。

- 在選擇專案類型中，為 Chatterbox 應用程式選擇空活動專案範本。
- 在設定專案中，為組態欄位選擇下列值：
  - 名稱：My App
  - 套件名稱：com.chatterbox.myapp
  - 儲存位置：指向上一步中建立的 `chatterbox` 目錄
  - 語言：Kotlin
  - API 最低等級：API 21：Android 5.0 (Lollipop)

正確指定所有組態參數後，`chatterbox` 資料夾內的檔案結構應如下所示：

```
- 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
```

現在有一個正在運行的 Android 專案，我們就可以將 com.amazonaws:ivs-chat-messaging 新增至 `build.gradle` 相依性。(有關 Gradle 建置工具包的詳細資訊，請參閱設定您的建置。)

注意：在每個程式碼片段的頂部，都有一個路徑，指向專案內您應該正在其中進行變更的檔案。這是專案根路徑的相對路徑。

在下面的程式碼中，將 *<version>* 取代為 Android 版聊天 SDK 的最新版本號 (例如，1.0.0)。

Kotlin：

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

plugins {
// ...
}

android {
// ...
```

```
}

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

新增新的相依性之後，在 Android Studio 中執行將專案與 Gradle 檔案同步，以便將專案與新的相依性同步。(如需詳細資訊，請參閱新增建置相依性。)

為了方便地從專案根目錄中執行 auth 伺服器 (在上一節中建立)，我們將其作為新模組包含在 `settings.gradle` 中。(如需詳細資訊，請參閱使用 Gradle 建構和建置軟體元件。)

Kotlin 指令碼：

```
// ./settings.gradle

// ...

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

從現在開始，由於 `auth-server` 包含在 Android 專案中，所以您可以使用下列命令從專案的根目錄中執行 auth 伺服器：

Shell：

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

連線到聊天室並觀察連線更新

若要開啟聊天室連線，我們會使用 onCreate() 活動生命週期回呼，它在首次建立活動時觸發。ChatRoom 建構函數要求我們提供 `region` 和 `tokenProvider` 來執行個體化聊天室連線。

注意：將在下一節中實作下面程式碼片段中的 `fetchChatToken` 函數。

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)
   }

// ...
}
```

顯示聊天室連線的變化並做出反應是構建諸如 chatterbox 聊天應用程式的重要部分。在開始與聊天室互動之前，我們必須訂閱聊天室連線狀態事件，以獲取更新。

ChatRoom 期望我們連接 ChatRoomListener 介面實作來引發生命週期事件。目前，接聽程式函數在被叫用時只會記錄確認訊息：

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")
        }
    }
}
```

現在已實作 ChatRoomListener，我們可將其連接至聊天室執行個體：

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 {
// ...
}
```

接下來，我們需要能夠讀取聊天室連線狀態。我們將其保留在 `MainActivity.kt` 屬性中，並將其初始化為聊天室的預設 DISCONNECTED 狀態 (請參閱《IVS 聊天功能 Android 版 SDK 參考》中的 `ChatRoom state`)。為了能夠使本機狀態保持最新，我們需要實作一個狀態更新程式函數；我們稱之為 `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")
            }
        }
    }
}
```

接下來，將我們的狀態更新程式函數與 ChatRoom.listener 屬性整合在一起：

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)
            }
        }
    }
}
```

現在我們能夠儲存、接聽 ChatRoom 狀態更新並做出反應，因此可以初始化連線：

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

```
}
```

建立字符提供者

現在可建立一個函數，其負責在我們的應用程式中建立和管理聊天權杖。在此範例中，我們使用適用於 Android 的 Retrofit HTTP 用戶端。

在傳送任何網路流量之前，我們必須為 Android 設定網路安全組態。(如需詳細資訊，請參閱網路安全組態。) 我們首先向 App Manifest 檔案新增網路許可。請注意，已新增 `user-permission` 標籤和 `networkSecurityConfig` 屬性，這將指向新網路安全組態。在下面的程式碼中，將 *<version>* 取代為 Android 版聊天 SDK 的最新版本號 (例如，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")
}
```

聲明 `10.0.2.2` 和 `localhost` 域為可信域，開始與我們的後端交換訊息：

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>
```

接下來，我們需要新增一個新的相依性，並新增 Gson 轉換器以用於解析 HTTP 回應。在下面的程式碼中，將 *<version>* 取代為 Android 版聊天 SDK 的最新版本號 (例如，1.0.0)。

Kotlin 指令碼：

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

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

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

若要擷取聊天權杖，需要從 chatterbox 應用程式中發出 POST HTTP 請求。我們在一個介面中定義請求，以便 Retrofit 能夠實作。(請參閱 Retrofit 文件。還應熟悉 CreateChatToken 端點規範。)

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>
}
```

現在，透過網路設定，可以新增一個負責建立和管理聊天權杖的函數。我們將其新增至 MainActivity.kt，其會在產生專案時自動建立：

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)
            }
        })
    }
}
```

後續步驟

現在您已經建立聊天室連線，請繼續本 Android 版教學課程的第 2 部分：訊息和事件。

Amazon IVS 聊天用戶端傳訊 SDK：Android 版教學課程第 2 部分：訊息和事件

本教學課程的第二部分 (也是最後一部分) 分為幾個部分：

1. the section called “建立用於傳送訊息的 UI”
   a. the section called “UI 主要版面配置”
   b. the section called “UI 抽象文字儲存格，以一致顯示文字”
   c. the section called “UI 左側聊天訊息”

   d. the section called “UI 右側聊天訊息”
   e. the section called “UI 其他顏色值”
2. the section called “套用檢視綁定”
3. the section called “管理聊天訊息請求”
4. the section called “最終步驟”

如需完整的 SDK 文件，請先閱讀 Amazon IVS 聊天用戶端傳訊 SDK (載於《Amazon IVS 聊天功能使用者指南》中) 和 Chat Client Messaging: SDK for Android Reference (聊天用戶端傳訊：Android 版 SDK 參考) (位於 GitHub 上)。

先決條件

請確定您已完成本教學課程的第 1 部分：聊天室。

建立用於傳送訊息的 UI

現在我們已成功初始化聊天室連線，因此可傳送第一條訊息。對於此功能，需要 UI。我們將新增：

- `connect/disconnect` 按鈕
- 帶 `send` 按鈕的訊息輸入
- 動態訊息清單。若要進行構建，我們可使用 Android Jetpack RecyclerView。

UI 主要版面配置

請參閱 Android 開發人員文件中的 Android Jetpack 版面配置。

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>
```

UI 抽象文字儲存格，以一致顯示文字

XML：

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

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:id="@+id/layout_container"
              android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:background="@color/light_gray"
            android:minWidth="100dp"
            android:orientation="vertical">

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

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

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

</LinearLayout>
```

UI 左側聊天訊息

XML：

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

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

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

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

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

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

        <TextView
                android:id="@+id/dateText"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginLeft="4dp"
                android:layout_marginBottom="4dp"
                android:text="10:00"
                app:layout_constraintBottom_toBottomOf="@+id/card_message_other"
```

```
                app:layout_constraintLeft_toRightOf="@+id/card_message_other"/>
    </androidx.constraintlayout.widget.ConstraintLayout>


</LinearLayout>
```

UI 右側聊天訊息

XML：

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

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

<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://
schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
                android:layout_width="match_parent"
               android:layout_height="wrap_content"
android:layout_marginEnd="8dp">

    <androidx.cardview.widget.CardView
            android:id="@+id/card_message_me"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="right"
            android:layout_marginBottom="10dp"
            android:foreground="?android:attr/selectableItemBackground"
            app:cardBackgroundColor="@color/purple_500"
            app:cardCornerRadius="10dp"
            app:cardElevation="0dp"
            app:cardPreventCornerOverlap="false"
            app:cardUseCompatPadding="true"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent">

        <include layout="@layout/common_cell"/>

    </androidx.cardview.widget.CardView>

    <TextView
            android:id="@+id/dateText"
            android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:layout_marginRight="12dp"
            android:layout_marginBottom="4dp"
            android:text="10:00"
            app:layout_constraintBottom_toBottomOf="@+id/card_message_me"
            app:layout_constraintRight_toLeftOf="@+id/card_message_me"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

UI 其他顏色值

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>
```

套用檢視綁定

我們利用 Android 檢視綁定功能，可以為 XML 版面配置參考綁定類別。若要啟用此功能，請在 `./app/build.gradle` 中將 `viewBinding` 建置選項設定為 `true`：

Kotlin 指令碼：

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

android {
//    ...

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

現在可以將 UI 與 Kotlin 程式碼連線：

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)
    }
}
```

我們還新增了可在聊天中刪除訊息和斷開使用者連線的方法，使用聊天訊息內容功能表可叫用這些方法：

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

管理聊天訊息請求

我們需要一種透過其所有可能狀態來管理聊天訊息請求的方法：

- 待定 – 訊息已傳送至聊天室，但尚未確認或拒絕。
- 已確認 – 聊天室已將訊息傳送給所有使用者 (包括我們)。
- 已拒絕 – 訊息被聊天室拒絕，其中包含錯誤物件。

我們會將未解決的聊天請求和聊天訊息保留在清單中。清單的優點是具有單獨的類別，我們稱之為 `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()
    }
}
```

若要將清單與 UI 連線，我們會使用轉接器。如需詳細資訊，請參閱使用 AdapterView 綁定至資料和已產生的綁定類別。

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
}
```

最終步驟

現在是時候連接我們的新轉接器，將 ChatEntries 類別綁定到 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
```

```
        }
    }
}
```

由於已經擁有一個負責追蹤聊天請求的類別 (ChatEntries)，因此我們已經準備好實作程式碼，以便在 roomListener 中操作 entries。我們會相應地將 entries 和 connectionState 更新到我們正在回應的事件：

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")
        }
    }
}
```

現在您應該能夠執行您的應用程式了！(請參閱建置並執行您的應用程式。) 切記在使用應用程式時執行後端伺服器。您可以從專案根目錄的終端中使用 `./gradlew :auth-server:run` 命令將其啟動，或者直接從 Android Studio 中執行 `auth-server:run` Gradle 任務來啟動。

Amazon IVS 聊天用戶端傳訊 SDK：Kotlin Coroutines 版教學課程第 1 部分：聊天室

這是由兩部分組成的教學課程的第一部分。透過使用 Kotlin 程式設計語言和 coroutines 建置功能完整的 Android 應用程式，您將學習使用 Amazon IVS 聊天功能傳訊 SDK 的基礎知識。我們稱呼該應用程式為 Chatterbox。

在開始該模組之前，請花幾分鐘時間熟悉先決條件、聊天權杖背後的重要概念以及建立聊天室所需的後端伺服器。

這些教學課程專為經驗豐富的 Android 開發人員而建立，他們不熟悉 IVS 聊天功能傳訊 SDK。您將需要熟悉 Kotlin 程式設計語言並在 Android 平台上建立 UI。

本教學課程的第一部分分為幾個部分：

1. the section called “設定本機身分驗證/授權伺服器”
2. the section called “建立 Chatterbox 專案”
3. the section called “連線到聊天室並觀察連線更新”
4. the section called “建立字符提供者”
5. the section called “後續步驟”

如需完整的 SDK 文件，請先閱讀 Amazon IVS 聊天用戶端傳訊 SDK (載於《Amazon IVS 聊天功能使用者指南》中) 和 Chat Client Messaging: SDK for Android Reference (聊天用戶端傳訊：Android 版 SDK 參考) (位於 GitHub 上)。

必要條件

- 熟悉 Kotlin 並在 Android 平台上建立應用程式。如果您不熟悉如何為 Android 建立應用程式，請在適用於 Android 開發人員的建置您的第一個應用程式指南中了解基礎知識。
- 閱讀並理解 IVS 聊天功能入門。
- 使用現有 IAM 政策中定義的 `CreateChatToken` 和 `CreateRoom` 功能建立 AWS IAM 使用者。(請參閱 IVS 聊天功能入門)。
- 確保將此使用者的私密/存取金鑰儲存在 AWS 憑證檔案中。如需指示，請參閱《AWS CLI 使用者指南》(特別是組態和憑證檔案設定)。
- 建立聊天室並保存其 ARN。請參閱IVS 聊天功能入門。(如果您未保存該 ARN，稍後可以使用主控台或 Chat API 來查詢。)

設定本機身分驗證/授權伺服器

後端伺服器負責建立聊天室並產生 IVS 聊天功能 Android SDK 需要的聊天權杖，以便對聊天室的用戶端執行身分驗證和授權。

請參閱 Amazon IVS 聊天功能入門中的建立聊天字符。如流程圖所示，您的伺服器端程式碼會負責建立聊天權杖。這意味著應用程式必須透過從伺服器端應用程式請求聊天字符，來提供自己產生聊天字符的方法。

我們使用 Ktor 架構建立即時本機伺服器，以管理使用本機 AWS 環境建立聊天權杖的作業。

此時，希望您已正確設定 AWS 憑證。如需逐步說明，請參閱設定適用於開發的 AWS 暫時憑證和 AWS 區域。

建立新目錄並將其命名為 chatterbox，在其內部還有一個名為 auth-server 的目錄。

伺服器資料夾的結構如下：

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

注意：您可以直接將這裡的程式碼複製/貼上到參考的檔案中。

接下來，我們會新增所有必要的相依性和外掛程式，以使 auth 伺服器正常工作：

Kotlin 指令碼：

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

現在，我們需要為 auth 伺服器設定記錄功能。(如需詳細資訊，請參閱設定記錄器)。

XML：

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

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

Ktor 伺服器需要組態設定，其會自動從 `resources` 目錄中的 `application.*` 檔案中載入，因此我們也會新增這些項目。(如需詳細資訊，請參閱檔案中的組態。)

HOCON：

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

ktor {
   deployment {
      port = 3000
```

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

最後，讓我們來實作伺服器：

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()
                )
            )
        }
    }
}
```

建立 Chatterbox 專案

若要建立 Android 專案，請安裝並開啟 Android Studio。

請按照 Android 官方建立專案指南中列出的步驟進行操作。

- 在選擇專案中，為我們的 Chatterbox 應用程式選擇空活動專案模板。

- 在設定專案中，為組態欄位選擇下列值：
  - 名稱：My App
  - 套件名稱：com.chatterbox.myapp
  - 儲存位置：指向上一步中建立的 `chatterbox` 目錄
  - 語言：Kotlin
  - API 最低等級：API 21：Android 5.0 (Lollipop)

正確指定所有組態參數後，`chatterbox` 資料夾內的檔案結構應如下所示：

```
- 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
```

現在有一個正在運行的 Android 專案，我們就可以將 com.amazonaws:ivs-chat-messaging 和 org.jetbrains.kotlinx:kotlinx-coroutines-core 新增至 `build.gradle` 相依性。(有關 Gradle 建置工具包的詳細資訊，請參閱設定您的建置。)

注意：在每個程式碼片段的頂部，都有一個路徑，指向專案內您應該正在其中進行變更的檔案。這是專案根路徑的相對路徑。

Kotlin：

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

plugins {
// ...
}

android {
// ...
}

dependencies {
    implementation 'com.amazonaws:ivs-chat-messaging:1.1.0'
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4'

// ...
```

```
}
```

新增新的相依性之後，在 Android Studio 中執行將專案與 Gradle 檔案同步，以便將專案與新的相依性同步。(如需詳細資訊，請參閱新增建置相依性。)

為了方便地從專案根目錄中執行我們的 auth 伺服器 (在上一節中建立)，我們將其作為新模組包含在 `settings.gradle` 中。(如需詳細資訊，請參閱使用 Gradle 建構和建置軟體元件。)

Kotlin 指令碼：

```
// ./settings.gradle

// ...

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

從現在開始，由於 `auth-server` 包含在 Android 專案中，所以您可以使用下列命令從專案的根目錄中執行 auth 伺服器：

Shell：

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

連線到聊天室並觀察連線更新

若要開啟聊天室連線，我們會使用 onCreate() 活動生命週期回呼，它在首次建立活動時觸發。ChatRoom 建構函數要求我們提供 `region` 和 `tokenProvider` 來執行個體化聊天室連線。

注意：將在下一節中實作下面程式碼片段中的 `fetchChatToken` 函數。

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)
   }

// ...
}
```

顯示聊天室連線的變化並做出反應是構建諸如 chatterbox 聊天應用程式的重要部分。在開始與聊天室互動之前，我們必須訂閱聊天室連線狀態事件，以獲取更新。

在適用於 coroutine 的聊天功能 SDK 中，ChatRoom 希望我們在流程中處理聊天室生命週期事件。目前，函數在被叫用時只會記錄確認訊息：

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")
                }
            }
        }
    }
}
```

接下來，我們需要能夠讀取聊天室連線狀態。我們將其保留在 `MainActivity.kt` property 中，並將其初始化為聊天室的預設 DISCONNECTED 狀態 (請參閱 IVS 聊天功能 Android 版 SDK 參考中的 `ChatRoom state`)。為了能夠使本機狀態保持最新，我們需要實作一個狀態更新程式函數；我們稱之為 `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")
          }
      }
}
```

接下來，將我們的狀態更新程式函數與 ChatRoom.listener 屬性整合在一起：

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)

                }
            }

      // ...
```

```
      }
   }
}
```

現在我們能夠儲存、接聽 ChatRoom 狀態更新並做出反應，因此可以初始化連線：

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)
      }
   }

   // ...
}
```

建立字符提供者

現在可建立一個函數，其負責在我們的應用程式中建立和管理聊天權杖。在此範例中，我們使用適用於 Android 的 Retrofit HTTP 用戶端。

在傳送任何網路流量之前，我們必須為 Android 設定網路安全組態。(如需詳細資訊，請參閱網路安全組態。) 我們首先向 App Manifest 檔案新增網路許可。請注意，已新增 `user-permission` 標籤和 `networkSecurityConfig` 屬性，這將指向新網路安全組態。在下面的程式碼中，將 *<version>* 取代為 Android 版聊天 SDK 的最新版本號 (例如，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")
}
```

聲明您的本機 IP 地址 (例如 `10.0.2.2` 和 `localhost`) 域為可信域，開始與我們的後端交換訊息：

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>
```

接下來，我們需要新增一個新的相依性，並新增 Gson 轉換器以用於解析 HTTP 回應。在下面的程式碼中，將 *<version>* 取代為 Android 版聊天 SDK 的最新版本號 (例如，1.1.0)。

Kotlin 指令碼：

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

若要擷取聊天權杖，需要從 chatterbox 應用程式中發出 POST HTTP 請求。我們在一個介面中定義請求，以便 Retrofit 能夠實作。(請參閱 Retrofit 文件。還應熟悉 CreateChatToken 端點規範。)

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)
    }
}
```

現在，透過網路設定，可以新增一個負責建立和管理聊天權杖的函數。我們將其新增至 MainActivity.kt，其會在產生專案時自動建立：

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)
            }
        })
    }
}
```

後續步驟

現在您已經建立聊天室連線，請繼續本 Kotlin Coroutines 教學課程的第 2 部分：訊息和事件。

Amazon IVS 聊天用戶端傳訊 SDK：Kotlin Coroutines 教學課程第 2 部分：訊息和事件

本教學課程的第二部分 (也是最後一部分) 分為幾個部分：

1. the section called “建立用於傳送訊息的 UI”
   a. the section called “UI 主要版面配置”
   b. the section called “UI 抽象文字儲存格，以一致顯示文字”
   c. the section called “UI 左側聊天訊息”

   d. the section called “UI 右側訊息”
   e. the section called “UI 其他顏色值”
2. the section called “套用檢視綁定”
3. the section called “管理聊天訊息請求”
4. the section called “最終步驟”

如需完整的 SDK 文件，請先閱讀 Amazon IVS 聊天用戶端傳訊 SDK (載於《Amazon IVS 聊天功能使用者指南》中) 和 Chat Client Messaging: SDK for Android Reference (聊天用戶端傳訊：Android 版 SDK 參考) (位於 GitHub 上)。

先決條件

請確定您已完成本教學課程的第 1 部分：聊天室。

建立用於傳送訊息的 UI

現在我們已成功初始化聊天室連線，因此可傳送第一條訊息。對於此功能，需要 UI。我們將新增：

- `connect/disconnect` 按鈕
- 帶 `send` 按鈕的訊息輸入
- 動態訊息清單。若要進行構建，我們可使用 Android Jetpack RecyclerView。

UI 主要版面配置

請參閱 Android 開發人員文件中的 Android Jetpack 版面配置。

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>
```

UI 抽象文字儲存格，以一致顯示文字

XML：

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

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:id="@+id/layout_container"
              android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:background="@color/light_gray"
            android:minWidth="100dp"
            android:orientation="vertical">

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

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

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

</LinearLayout>
```

UI 左側聊天訊息

XML：

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

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

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

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

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

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

        <TextView
                android:id="@+id/dateText"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginLeft="4dp"
                android:layout_marginBottom="4dp"
                android:text="10:00"
                app:layout_constraintBottom_toBottomOf="@+id/card_message_other"
```

```
                app:layout_constraintLeft_toRightOf="@+id/card_message_other"/>
    </androidx.constraintlayout.widget.ConstraintLayout>


</LinearLayout>
```

UI 右側訊息

XML：

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

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

<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://
schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
                android:layout_width="match_parent"
               android:layout_height="wrap_content"
android:layout_marginEnd="8dp">

    <androidx.cardview.widget.CardView
            android:id="@+id/card_message_me"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="right"
            android:layout_marginBottom="10dp"
            android:foreground="?android:attr/selectableItemBackground"
            app:cardBackgroundColor="@color/purple_500"
            app:cardCornerRadius="10dp"
            app:cardElevation="0dp"
            app:cardPreventCornerOverlap="false"
            app:cardUseCompatPadding="true"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent">

        <include layout="@layout/common_cell"/>

    </androidx.cardview.widget.CardView>

    <TextView
            android:id="@+id/dateText"
            android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:layout_marginRight="12dp"
            android:layout_marginBottom="4dp"
            android:text="10:00"
            app:layout_constraintBottom_toBottomOf="@+id/card_message_me"
            app:layout_constraintRight_toLeftOf="@+id/card_message_me"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

UI 其他顏色值

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>
```

套用檢視綁定

我們利用 Android 檢視綁定功能，可以為 XML 版面配置參考綁定類別。若要啟用此功能，請在 `./app/build.gradle` 中將 `viewBinding` 建置選項設定為 `true`：

Kotlin 指令碼：

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

android {
//    ...

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

現在可以將 UI 與 Kotlin 程式碼連線：

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)
    }
// ...

}
```

我們還新增了可在聊天中刪除訊息和斷開使用者連線的方法，使用聊天訊息內容功能表可叫用這些方法：

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")
            }
        }
    }
}
```

管理聊天訊息請求

我們需要一種透過其所有可能狀態來管理聊天訊息請求的方法：

- 待定 – 訊息已傳送至聊天室，但尚未確認或拒絕。
- 已確認 – 聊天室已將訊息傳送給所有使用者 (包括我們)。
- 已拒絕 – 訊息被聊天室拒絕，其中包含錯誤物件。

我們會將未解決的聊天請求和聊天訊息保留在清單中。清單的優點是具有單獨的類別，我們稱之為 `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()
    }
}
```

若要將清單與 UI 連線，我們會使用轉接器。如需詳細資訊，請參閱使用 AdapterView 綁定至資料和已產生的綁定類別。

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
}
```

最終步驟

現在是時候連接我們的新轉接器，將 ChatEntries 類別綁定到 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
        }
    }
}
```

由於我們已經擁有一個負責追蹤聊天請求 (ChatEntries) 的類別，因此我們已經準備好實作程式碼，以便在 roomListener 中操作 `entries`。我們會相應地將 `entries` 和 `connectionState` 更新到我們正在回應的事件：

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)
    }

// ...

}
```

現在您應該能夠執行您的應用程式了！(請參閱建置並執行您的應用程式。) 切記在使用應用程式時執行後端伺服器。您可以從專案根目錄的終端中使用 `./gradlew :auth-server:run` 命令將其啟動，或者直接從 Android Studio 中執行 `auth-server:run` Gradle 任務來啟動。

《Amazon IVS 聊天用戶端傳訊 SDK：iOS 版指南》

Amazon Interactive Video Service (IVS) Chat 用戶端傳訊 iOS 版開發套件提供的介面，可讓您使用 Apple 的 Swift 程式設計語言整合平台上的 IVS Chat 傳訊 API。

最新版 IVS Chat 用戶端傳訊 iOS 版開發套件：1.0.0 (版本備註)

參考文件和教學課程：如需有關 Amazon IVS 聊天功能用戶端傳訊 iOS 版開發套件中最重要方法的資訊，請參閱參考文件，網址為 https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/。此儲存庫還包含各種文章和教學課程。

範本程式碼：請參閱 GitHub 上的 iOS 範本儲存庫：https://github.com/aws-samples/amazon-ivs-chat-for-ios-demo。

平台需求：開發需要 iOS 13.0 或更高版本。

開始使用

我們建議您透過 Swift Package Manager 來整合開發套件。您也可以使用 CocoaPods 或手動整合架構。

整合開發套件後，您可以在相關的 Swift 檔案最上方新增下列程式碼以匯入開發套件：

```
import AmazonIVSChatMessaging
```

Swift Package Manager

若要在 Swift Package Manager 專案中使用 AmazonIVSChatMessaging 程式庫，請將其新增至套件和相關目標的相依性：

1. 從 https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.下載最新版本的 .xcframework。
2. 在您的終端機中執行：

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

3. 取得上一步的輸出，並將其貼到專案 Package.swift 檔案中 .binaryTarget 的 Checksum 屬性內，如下所示：

```
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

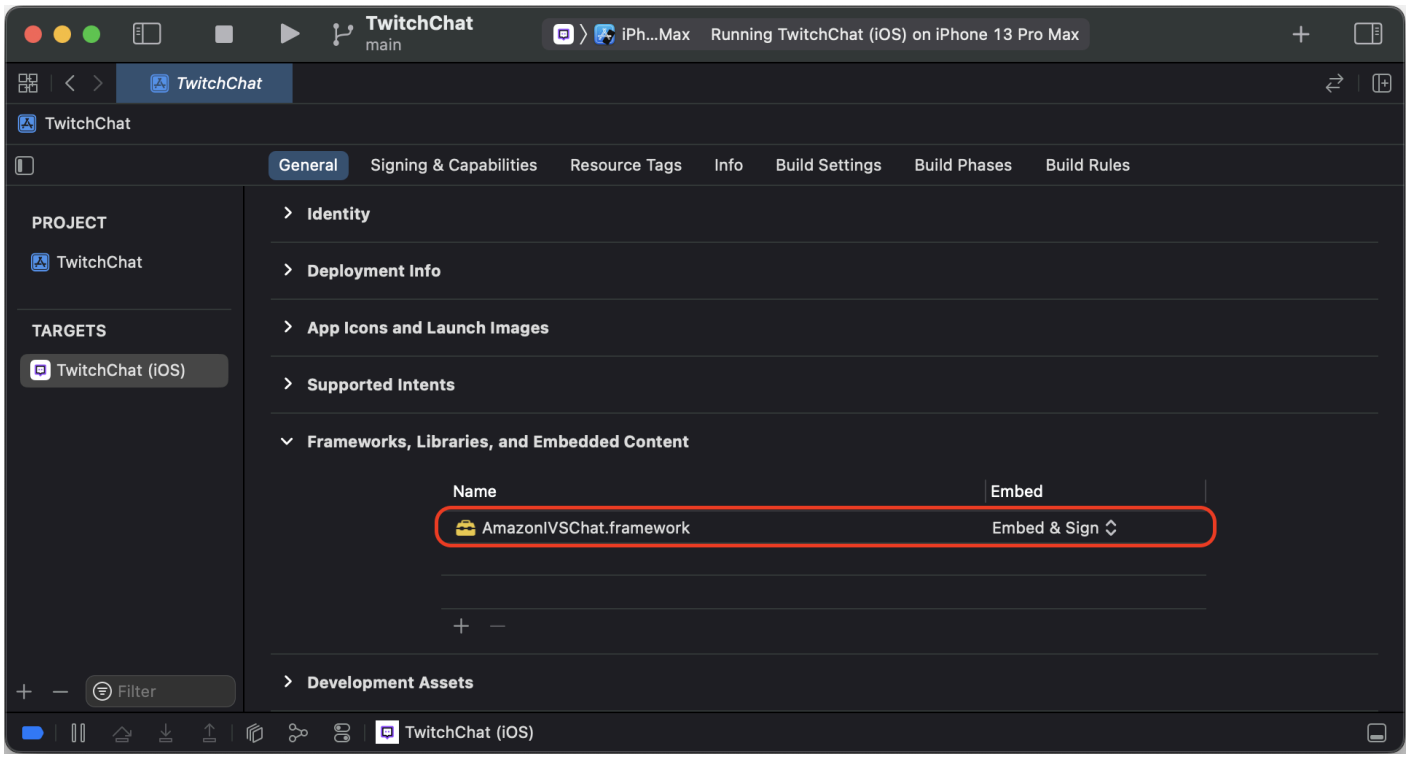
透過 CocoaPods 用名稱 AmazonIVSChatMessaging 發行版本。將此相依性新增到您的 Podfile：

```
pod 'AmazonIVSChat'
```

執行 `pod install`，將可在您的 `.xcworkspace` 中使用開發套件。

手動安裝

1. 從 https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip 中下載最新版本。
2. 解壓縮封存檔的內容。`AmazonIVSChatMessaging.xcframework` 包含用於裝置和模擬器的 SDK。
3. 內嵌擷取的 `AmazonIVSChatMessaging.xcframework`，方法是將其拖曳至應用程式目標的 General (一般) 索引標籤的 Frameworks, Libraries, and Embedded Content (架構、程式庫和內嵌內容) 區段：



使用開發套件

與聊天室連線

在開始使用之前，請先熟悉 Amazon IVS 聊天功能入門。另請參閱 Web 版、Android 版和 iOS 版的範例應用程式。

若要與聊天室連線，您的應用程式必須設法擷取後端提供的聊天權杖。您的應用程式可能會向後端發出網路請求以擷取聊天權杖。

若要將此擷取的聊天權杖傳送給開發套件，開發套件的 ChatRoom 模型會要求您提供符合在初始化時所提供 ChatTokenProvider 通訊協定的 async 函數或物件執行個體。這些方法中的任何一種傳回的值都必須是開發套件 ChatToken 模型的執行個體。

注意：請使用從後端擷取的資料來填入 ChatToken 模型的執行個體。初始化 ChatToken 執行個體所需的欄位與 CreateChatToken 回應中的欄位相同。如需有關初始化 ChatToken 模型執行個體的詳細資訊，請參閱建立 ChatToken 執行個體。請謹記，您的後端必須負責向應用程式提供 CreateChatToken 回應中的資料。您用來與後端通訊以產生聊天權杖的方式取決於您的應用程式及其基礎架構。

選擇向開發套件提供 ChatToken 的策略之後，請在使用您的權杖提供者，以及您的後端用來建立所嘗試連線之聊天室的 AWS 區域，成功初始化 .connect() 執行個體後呼叫 ChatRoom。請注意，.connect() 是一個擲回非同步函數：

```
import AmazonIVSChatMessaging

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

符合 ChatTokenProvider 通訊協定

如果使用 ChatRoom 的初始設定式中的 tokenProvider 參數，則可提供 ChatTokenProvider 的執行個體。以下為符合 ChatTokenProvider 的物件範例：

```
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
      )
   }
}
```

接著您可取得此符合物件的執行個體，並將其傳給 ChatRoom 的初始設定式：

```
// 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()
```

提供 Swift 非同步函數

假設您已有一個用於管理應用程式網路請求的管理工具。這看起來類似下述：

```
import AmazonIVSChatMessaging

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

您可在管理工具中新增另一個函數，以便從後端擷取 ChatToken：

```
import AmazonIVSChatMessaging

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

然後，在初始化 ChatRoom 時，在 Swift 中使用對該函數的參考：

```
import AmazonIVSChatMessaging
```

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

建立 ChatToken 執行個體

使用開發套件中提供的初始設定式可輕鬆建立 ChatToken 執行個體。請參閱 Token.swift 中的文件，以進一步了解 ChatToken 中的屬性。

```
import AmazonIVSChatMessaging

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

使用可解碼

如果在與 IVS Chat API 對接時，您的後端決定只將 CreateChatToken 回應轉送至您的前端應用程式，則您可利用 ChatToken 符合 Swift Decodable 通訊協定的優勢。但有個必須注意的地方。

CreateChatToken 回應承載會使用以網際網路時間戳記的 ISO 8601 標準格式化的日期字串。通常，在 Swift 中您會提供 JSONDecoder.DateDecodingStrategy.iso8601 作為 JSONDecoder 的 .dateDecodingStrategy 屬性值。但是，CreateChatToken 在其字串中使用的高精確度小數秒數並不受 JSONDecoder.DateDecodingStrategy.iso8601 支援。

為了方便起見，開發套件在 JSONDecoder.DateDecodingStrategy 中提供了公有擴充功能，其額外的 .preciseISO8601 策略可讓您成功使用 JSONDecoder 來解碼 ChatToken 的執行個體：

```
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)
```

中斷與聊天室的連線

若要手動中斷您已成功連線的 ChatRoom 執行個體之連線，請呼叫 room.disconnect()。預設情況下，聊天室會在解除配置時自動呼叫此函數。

```
import AmazonIVSChatMessaging

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

// Disconnect
room.disconnect()
```

接收聊天訊息/事件

若要在聊天室中傳送和接收訊息，則在您成功初始化 ChatRoom 的執行個體並呼叫 room.connect() 之後，必須提供符合 ChatRoomDelegate 通訊協定的物件。使用 UIViewController 的典型範例如下：

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

在連線變更時收到通知

正常而言，在聊天室完全連線之前，您無法在聊天室中執行傳送訊息等動作。SDK 的架構嘗試鼓勵透過非同步 API 在背景執行緒上連線至 ChatRoom。如果您想要在 UI 中建構可停用傳送訊息按鈕等功能，開發套件對此提供兩種使用 Combine 或 ChatRoomDelegate 的策略，以便在聊天室的連線狀態變更時收到通知。這些策略說明如下。

重要：聊天室的連線狀態也可能因網路連線中斷等因素而變更。建構應用程式時請將此列入考量。

使用 Combine

ChatRoom 的每個執行個體都會以 state 屬性的形式隨附它自己的 Combine 發佈者：

```
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()
}
```

使用 ChatRoomDelegate

您也可以在符合 ChatRoomDelegate 的物件中使用可選函數 roomDidConnect(_:)、roomIsConnecting(_:) 以及 roomDidDisconnect(_:)。以下為使用 UIViewController 的範例：

```
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!")
   }
```

```
}
```

在聊天室中執行動作

不同的使用者擁有可在聊天室中執行各種動作所需的不同功能；例如傳送訊息、刪除訊息或中斷使用者連線。若要執行這些動作之一，請在連線的 ChatRoom 上呼叫 perform(request:)，並傳入開發套件中提供的其中一個 ChatRequest 物件的執行個體。支援的請求位於 Request.swift 中。

當後端應用程式呼叫 CreateChatToken 時，連線的使用者必須具備授予他們的特定功能，才能在聊天室中執行某些動作。開發套件在設計上無法辨別連線使用者的功能。因此，雖然您可以試著在連線的 ChatRoom 執行個體中執行仲裁者動作，但控制平面 API 最終會決定該動作是否成功。

經由 room.perform(request:) 的所有動作都會等待，直至聊天室收到與所接收模型和請求物件的 requestId 皆相符的預期模型 (其類型與請求物件本身相關) 執行個體。如果請求有問題，ChatRoom 一律會以 ChatError 形式擲回錯誤。ChatError 的定義請見 Error.swift。

傳送訊息

若要傳送聊天訊息，請使用 SendMessageRequest 執行個體：

```
import AmazonIVSChatMessaging

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

如上所述，ChatRoom 一收到對應的 ChatMessage，就會傳回 room.perform(request:)。如果請求有問題 (例如超出聊天室的訊息字元限制)，就會改為擲回 ChatError 的執行個體。然後，您就可以在 UI 中顯示此實用資訊：

```
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)
}
```

將中繼資料附加至訊息

傳送訊息時可附加與該訊息相關的中繼資料。SendMessageRequest 有 attributes 屬性，使用這個屬性可以初始化您的請求。當其他人在聊天室中收到該訊息時，也會連帶收到您附加在訊息中的資料。

以下是將表情符號資料附加在要傳送之訊息中的範例：

```
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"
        ]
    )
)
```

在 SendMessageRequest 中使用 attributes 對於在聊天產品中建構複雜功能非常有用。例如，在 SendMessageRequest 中使用 [String : String] 屬性字典可以建構執行緒功能！

attributes 承載非常靈活及強大。使用它可衍生您無法以其他方式衍生的訊息相關資訊。例如，使用屬性比剖析訊息字串以取得表情符號等相關資訊要容易得多。

刪除訊息

刪除聊天訊息的方式跟傳送聊天訊息一樣。為了達到此目的，可使用 ChatRoom 的 `room.perform(request:)` 函數來建立 DeleteMessageRequest 的執行個體。

若要輕鬆存取已接收之聊天訊息的先前執行個體，將 `message.id` 的值傳入 DeleteMessageRequest 的初始設定式中。

您也可以向 DeleteMessageRequest 提供原因字串，以便將其顯示在您的 UI 中。

```
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!"
   )
)
```

由於這是仲裁者才能做的動作，因此您的使用者實際上可能無法刪除其他使用者的訊息。如果使用者試圖在權限不足的情況下刪除訊息，則您可使用 Swift 的可擲回函數機制在 UI 中顯示錯誤訊息。

當後端叫用 CreateChatToken 供呼叫者使用時，它必須將 "DELETE_MESSAGE" 傳入 `capabilities` 欄位，才能啟動該功能供已連線的聊天使用者使用。

以下是在功能不足的情況下試圖刪除訊息時發生權限錯誤的範例：

```
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)
}
```

中斷與其他使用者的連線

使用 room.perform(request:) 可中斷其他使用者與聊天室的連線。明確來說，請使用 DisconnectUserRequest 的執行個體。ChatRoom 收到的全部 ChatMessage 都會有 sender 屬性，其中包含正確初始化 DisconnectUserRequest 執行個體時需要使用的使用者 ID。您也可以在收到中斷連線請求時輸入原因字串。

```
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
   )
)
```

由於這是仲裁者動作的另一個範例，您可以試著中斷其他使用者的連線，但前提是您必須有 DISCONNECT_USER 功能。當您的後端應用程式呼叫 CreateChatToken 並將 "DISCONNECT_USER" 字串注入 capabilities 欄位時，就會設定此功能。

如果您的使用者沒有中斷其他使用者連線的功能，room.perform(request:) 會擲回一個ChatError 執行個體，就像其他請求一樣。檢查錯誤的 errorCode 屬性可以判斷該請求失敗的原因是否因為缺少仲裁者權限：

```
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)
}
```

Amazon IVS 聊天用戶端傳訊 SDK：iOS 版教學課程

Amazon Interactive Video (IVS) Chat 用戶端傳訊 iOS 版開發套件提供的介面，可讓您使用 Apple 的 Swift 程式設計語言整合平台上的 IVS Chat 傳訊 API。

如需 Chat iOS 版開發套件教學課程，請參閱 https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios。

《Amazon IVS 聊天用戶端傳訊 SDK：JavaScript 版指南》

Amazon Interactive Video (IVS) Chat 用戶端傳訊 JavaScript 版開發套件可讓您在使用 Web 瀏覽器的平台上整合我們的 Amazon IVS 聊天功能傳訊 API。

IVS Chat 用戶端傳訊 JavaScript 開發套件的最新版本：1.0.2 (版本備註)

參考文件：如需 Amazon IVS 聊天用戶端傳訊 JavaScript SDK 中最重要方法的相關資訊，請參閱參考文件，網址為 https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/

範例程式碼：請參閱 GitHub 上的範例儲存庫，以取得使用 JavaScript 版開發套件的 Web 專用示範，網址為：https://github.com/aws-samples/amazon-ivs-chat-web-demo

開始使用

在開始使用之前，請先詳閱 Amazon IVS 聊天功能入門。

新增套件

使用以下任一項：

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

或：

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

React Native 支援

IVS Chat 用戶端傳訊 JavaScript 版開發套件具有使用 `crypto.getRandomValues` 方法的 `uuid` 相依性。由於 React Native 中不支援此方法，因此您需要安裝額外的 polyfill `react-native-get-random-value`，並在 `index.js` 檔案的最上方將其匯入：

```
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);
```

設定後端

伺服器上需有可與 Amazon IVS 聊天功能 API 通訊的端點才能執行這項整合。使用官方 AWS 程式庫從您的伺服器存取 Amazon IVS API。透過公開套件就可以存取這些程式庫，而且有多種程式語言可供選用，例如 node.js、java 和 go。

建立可與 Amazon IVS 聊天功能 API CreateChatToken 端點通訊的伺服器端點，以便為聊天使用者建立聊天字符。

使用開發套件

初始化聊天室執行個體

建立 ChatRoom 類別的執行個體。這需要傳遞 `regionOrUrl` (託管聊天室的 AWS 區域) 和 `tokenProvider` (即將在下一步中建立的字符擷取方法)：

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

字符提供者函數

建立非同步字符提供者函數，由該函數從後端擷取聊天字符：

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

該函數不應接受任何參數，但應傳回一個內含聊天字符物件的 Promise：

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

初始化 ChatRoom 物件會需使用這個函數。在下方的 `<token>` 和 `<date-time>` 欄位填入您從後端接收到的值：

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

務必記得將 tokenProvider 傳遞給 ChatRoom 建構函數。連線中斷或工作階段到期時，ChatRoom 會重新整理字符。切勿在任何位置使用 tokenProvider 儲存字符，交由 ChatRoom 為您處理即可。

接收事件

接著訂閱聊天室事件，以接收生命週期事件以及聊天室中傳遞的訊息和事件：

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

與聊天室連線

基本初始化的最後一步是透過建立 WebSocket 連線，連線至聊天室。若要這樣做，只需在聊天室執行個體中呼叫 `connect()` 方法即可。

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

開發套件會嘗試與聊天室建立連線，而該聊天室是以從伺服器收到的聊天字符編碼建立而成。

呼叫 connect() 後，聊天室會轉換為 connecting 狀態並發出 connecting 事件。成功連線至聊天室後，聊天室會轉換為 connected 狀態並發出 connect 事件。

擷取字符或連線至 WebSocket 時一旦發生問題，可能會導致連線失敗。這種情況下，聊天室會嘗試自動重新連線，最多會嘗試至 maxReconnectAttempts 建構函數參數所指示的次數。嘗試重新連線期間，聊天室會處於 connecting 狀態，不會發出其他事件。用盡所有重新連線的嘗試次數後，聊天室會轉換為 disconnected 狀態並發出 disconnect 事件 (附有相關的中斷連線原因)。在 disconnected 狀態下，聊天室就不會再嘗試連線；您必須再次呼叫 connect() 才能觸發連線程序。

在聊天室中執行動作

Amazon IVS 聊天功能傳訊開發套件提供各種使用者動作，可供傳送訊息、刪除訊息和中斷其他使用者的連線。這些動作都可以在 ChatRoom 執行個體上使用。這些動作會傳回 Promise 物件，讓您能接收請求確認或拒絕。

傳送訊息

若要提出此請求，您的聊天字符編碼內容必須含有 SEND_MESSAGE 功能。

若要觸發傳送訊息請求，請執行下列動作：

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

若要取得請求確認或拒絕，請 await 傳回的 Promise 或使用 then() 方法：

```
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.
}
```

刪除訊息

若要提出此請求，您的聊天字符編碼內容必須含有 DELETE_MESSAGE 功能。

若出於管制目的而需刪除訊息，請呼叫 deleteMessage() 方法：

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

若要取得請求確認或拒絕，請 await 傳回的 Promise 或使用 then() 方法：

```
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.
}
```

中斷與其他使用者的連線

若要提出此請求，您的聊天字符編碼內容必須含有 DISCONNECT_USER 功能。

若出於管制目的而需中斷其他使用者的連線，請呼叫 disconnectUser() 方法：

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

若要取得請求確認或拒絕，請 await 傳回的 Promise 或使用 then() 方法：

```
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.
}
```

中斷與聊天室的連線

若要關閉與聊天室的連線，請在 room 執行個體上呼叫 disconnect() 方法：

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

呼叫此方法可促使聊天室依序關閉底層的 WebSocket。聊天室執行個體會轉換為 disconnected 狀態並發出中斷連線事件，同時 disconnect 原因會設定為 "clientDisconnect"。

Amazon IVS 聊天用戶端傳訊 SDK：JavaScript 版教學課程第 1 部分：聊天室

這是由兩部分組成的教學課程的第一部分。您將透過使用 JavaScript/TypeScript 建置功能完整的應用程式，來學習使用 Amazon IVS 聊天用戶端傳訊 JavaScript SDK 的基礎知識。我們稱呼該應用程式為 Chatterbox。

目標對象是初次使用 Amazon IVS 聊天功能傳訊開發套件的經驗豐富的開發人員。您應該很熟悉 JavaScript/TypeScript 程式設計語言和 React 程式庫。

為了簡潔起見，我們將 Amazon IVS 聊天用戶端傳訊 JavaScript SDK 稱為 Chat JS SDK。

注意：在某些情況下，JavaScript 和 TypeScript 的程式碼範例是相同的，因此我們會將兩者的範例合併。

本教學課程的第一部分分為幾個部分：

1. the section called “設定本機身分驗證/授權伺服器”
2. the section called “建立 Chatterbox 專案”
3. the section called “與聊天室連線”
4. the section called “建立字符提供者”
5. the section called “觀察連線更新”
6. the section called “建立傳送按鈕元件”
7. the section called “建立訊息輸入”
8. the section called “後續步驟”

如需完整的 SDK 文件，請先閱讀 Amazon IVS 聊天用戶端傳訊 SDK (載於《Amazon IVS 聊天功能使用者指南》中) 和 Chat Client Messaging: SDK for JavaScript Reference (聊天用戶端傳訊：JavaScript 版 SDK 參考) (位於 GitHub 上)。

必要條件

- 熟悉 JavaScript/TypeScript 和 React 程式庫。如果不熟悉 React，請在 React 簡介中了解基礎知識。
- 閱讀並理解 IVS 聊天功能入門。

- 使用現有 IAM 政策中定義的 CreateChatToken 和 CreateRoom 功能建立 AWS IAM 使用者。(請參閱 IVS 聊天功能入門)。
- 確保將此使用者的私密/存取金鑰儲存在 AWS 憑證檔案中。如需指示，請參閱《AWS CLI 使用者指南》(特別是組態和憑證檔案設定)。
- 建立聊天室並保存其 ARN。請參閱IVS 聊天功能入門。(如果您未保存該 ARN，稍後可以使用主控台或 Chat API 來查詢。)
- 使用 NPM 或 Yarn 套件管理工具安裝 Node.js 14+ 環境。

設定本機身分驗證/授權伺服器

後端應用程式負責建立聊天室並產生 Chat JS SDK 需要的聊天字符，以便對聊天室的用戶端執行身分驗證和授權。您必須使用自己的後端，因為您無法將 AWS 金鑰安全地存放在行動應用程式中；成熟的攻擊者可擷取這些金鑰並取得 AWS 帳戶的存取權。

請參閱 Amazon IVS 聊天功能入門中的建立聊天字符。如流程圖所示，您的伺服器端應用程式會負責建立聊天字符。這意味著應用程式必須透過從伺服器端應用程式請求聊天字符，來提供自己產生聊天字符的方法。

在本節中，您將學習在後端建立字符提供者的基礎知識。我們使用快速架構建立即時本機伺服器，以管理使用本機 AWS 環境建立聊天字符的作業。

使用 NPM 建立空的 npm 專案。建立目錄來保存應用程式，並將其設置為工作目錄：

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

使用 `npm init` 為應用程式建立 `package.json` 檔案：

```
$ npm init
```

此命令會提示您輸入數個項目，包括應用程式的名稱和版本。現在，只需按 RETURN 即可接受其中大多數的預設值，但存在以下例外狀況：

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

按下 RETURN 接受建議的 `index.js` 預設檔案名稱，或輸入您想要的主檔案名稱。

立即安裝所需的依存項目：

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

aws-sdk 需要組態環境變數，這些變數會自動從位於根目錄中名為 .env 的檔案載入。若要進行設定，請建立名為 .env 的新檔案，並填寫缺少的組態資訊：

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

現在，我們使用您在 npm init 命令中輸入的名稱，在根目錄中建立一個進入點檔案。在這種情況下，我們使用 index.js 並匯入所有必要的套件：

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

現在建立新的 express 執行個體：

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

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

之後，您可以為字符提供者建立第一個端點 POST 方法。從請求主體 (roomId、userId、capabilities 和 sessionDurationInMinutes) 取得所需的參數：

```
app.post('/create_chat_token', (req, res) => {
```

```
  const { roomIdentifier, userId, capabilities, sessionDurationInMinutes } = req.body
 || {};
});
```

新增必填欄位驗證：

```
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;
  }
});
```

準備好 POST 方法後，我們將 createChatToken 與 aws-sdk 整合來取得身分驗證/授權的核心功能：

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

在檔案結尾，為 express 應用程式新增連接埠接聽程式：

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

現在，您可以從專案的根目錄中使用下列命令執行伺服器：

```
$ node index.js
```

提示：此伺服器會在 https://localhost:3000 接受 URL 請求。

建立 Chatterbox 專案

首先，建立一個名為 chatterbox 的 React 專案。執行此命令：

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

您可以透過節點套件管理工具或 Yarn 套件管理工具整合 Chat JS SDK：

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

與聊天室連線

在這裡，您可以建立 ChatRoom 並使用異步方法連接到其中。此 ChatRoom 類別會管理使用者與 Chat JS SDK 的連線。若要成功連線到聊天室，您必須在 React 應用程式中提供 ChatToken 的執行個體。

瀏覽至預設 chatterbox 專案中建立的 App 檔案，然後刪除兩個 標籤之間的所有內容。不需要預先填入程式碼。此刻，App 相當空。

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

import * as React from 'react';

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

建立新的 ChatRoom 執行個體並使用 useState 勾點將其傳遞給狀態。其需要傳遞 regionOrUrl (託管聊天室的 AWS 區域) 和 tokenProvider (用於後續步驟中建立的後端身分驗證/授權流程)。

重要事項：您必須使用與在 Amazon IVS 聊天功能入門中建立聊天室的區域相同的 AWS 區域。該 API 是 AWS 區域服務。如需支援區域和 Amazon IVS 聊天功能 HTTPS 服務端點的清單，請參閱 Amazon IVS 聊天功能區域頁面。

```
// 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>;
}
```

建立字符提供者

下一步，我們需要建置 ChatRoom 建構函數所需的無參數 tokenProvider 函數。首先，我們將建立 fetchChatToken 函數，該函數將向您在 the section called “設定本機身分驗證/授權伺服器” 設定的後端應用程式發出 POST 請求。聊天字符包含開發套件成功建立聊天室連線所需的資訊。Chat API 使用這些字符作為驗證使用者身分、聊天室內功能和工作階段持續時間的安全方式。

在專案導覽器中，建立名為 fetchChatToken 的新 TypeScript/JavaScript 檔案。建立 backend 應用程式的擷取請求，並從回應中傳回 ChatToken 物件。新增建立聊天字符所需的請求主體屬性。使用針對 Amazon Resource Name (ARN) 定義的規則。這些屬性會記錄在 CreateChatToken 端點中。

注意：您在此處使用的 URL 與執行後端應用程式時本機伺服器建立的 URL 相同。

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),
  };
}
```

觀察連線更新

對聊天室連線狀態的變化做出反應，是製作聊天應用程式的重要一環。我們從訂閱相關事件開始：

```
// 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>;
}
```

接下來，我們需要提供讀取連線狀態的能力。我們使用 useState 勾點在 App 中建立一些本機狀態，並在每個接聽程式中設定連線狀態。

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>;
}
```

訂閱連線狀態後，顯示連線狀態並使用 `useEffect` 勾點內的 `room.connect` 方法連接到聊天室：

```
// 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>
);

// ...
```

您已成功實作聊天室連線。

建立傳送按鈕元件

在本節中，您將建立傳送按鈕，該按鈕對每個連線狀態都提供不同的設計。傳送按鈕有助於在聊天室中傳送訊息。其還可以作為判斷是否可以/何時可以傳送訊息的視覺化指示器；例如，面對中斷的連線或過期的聊天工作階段。

首先，在 Chatterbox 專案的 `src` 目錄中建立新檔案並將其命名為 SendButton。然後，建立元件，該元件將為聊天應用程式顯示按鈕。匯出 SendButton 並將其匯入 App。在空白的 `<div></div>` 中新增 `<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>
);
```

接下來，在 App 中定義名為 onMessageSend 的函數並將其傳遞給 SendButton onPress 屬性。定義另一個名為 isSendDisabled 的變數 (可防止在聊天室未連接時傳送訊息) 並將其傳遞給 SendButton disabled 屬性。

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

// ...

const onMessageSend = () => {};

const isSendDisabled = connectionState !== 'connected';

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

// ...
```

建立訊息輸入

Chatterbox 訊息列是您將與其互動以將訊息傳送到聊天室的元件。通常，其包含用於撰寫訊息的文字輸入和用於傳送訊息的按鈕。

若要建立 MessageInput 元件，請先在 src 目錄中建立新檔案並將其命名為 MessageInput。然後，建立受控的輸入元件，該元件將為聊天應用程式顯示輸入。匯出 MessageInput 並將其匯入 App (在 <SendButton /> 之上)。

使用 useState 勾點建立名為 messageToSend 的新狀態，並以空字串作為其預設值。在應用程式主體中，將 messageToSend 傳遞給 MessageInput 的 value，並將 setMessageToSend 傳遞給 onMessageChange 屬性：

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>
);
```

後續步驟

現在，您已經完成建置 Chatterbox 的訊息列，請繼續閱讀本 JavaScript 版教學課程的第 2 部分：訊息和事件。

Amazon IVS 聊天用戶端傳訊 SDK：JavaScript 版教學課程第 2 部分：訊息和事件

本教學課程的第二部分 (也是最後一部分) 分為幾個部分：

1. the section called “訂閱聊天訊息事件”
2. the section called “顯示收到的訊息”
   a. the section called “建立訊息元件”
   b. the section called “辨識目前使用者所傳送的訊息”

   c. the section called “建立訊息清單元件”
   d. the section called “呈現聊天訊息清單”
3. the section called “在聊天室中執行動作”
   a. the section called “傳送訊息”
   b. the section called “刪除訊息”
4. the section called “後續步驟”

注意：在某些情況下，JavaScript 和 TypeScript 的程式碼範例是相同的，因此我們會將兩者的範例合併。

如需完整的 SDK 文件，請先閱讀 Amazon IVS 聊天用戶端傳訊 SDK (載於《Amazon IVS 聊天功能使用者指南》中) 和 Chat Client Messaging: SDK for JavaScript Reference (聊天用戶端傳訊：JavaScript 版 SDK 參考) (位於 GitHub 上)。

先決條件

請確定您已完成本教學課程的第 1 部分：聊天室。

訂閱聊天訊息事件

當聊天室中發生事件時，ChatRoom 執行個體會使用事件進行通訊。若要開始實作聊天體驗，當其他人在其連線的聊天室中傳送訊息時，您需要向使用者顯示。

您可在此處訂閱聊天訊息事件。稍後，我們將說明如何更新您建立的訊息清單，此清單會隨每個訊息/事件進行更新。

在 App 的 `useEffect` 勾點內，訂閱所有訊息事件：

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

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

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

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

顯示收到的訊息

接收訊息是聊天體驗的核心部分。您可以使用 Chat JS SDK 設定程式碼，輕鬆接收來自連線至聊天室的其他使用者的事件。

稍後，我們將展示如何利用您在此處建立的元件在聊天室中執行動作。

在您的 App 中，使用名為 messages 的 ChatMessage 陣列類型定義一個名為 messages 的狀態：

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([]);

  //...
}
```

接下來，在 message 接聽程式函數中，將 message 附加至 messages 陣列：

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

```
// ...

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

// ...
```

下面我們將逐步完成任務，以顯示收到的訊息：

1. the section called “建立訊息元件”
2. the section called “辨識目前使用者所傳送的訊息”
3. the section called “建立訊息清單元件”
4. the section called “呈現聊天訊息清單”

建立訊息元件

Message 元件負責呈現聊天室收到的訊息內容。在本節中，您會建立用來呈現 App 中個別聊天訊息的訊息元件。

在 src 目錄中建立新檔案，並將其命名為 Message。傳入此元件的 ChatMessage 類型，並從 ChatMessage 屬性傳遞 content 字串，以顯示從聊天室訊息接聽程式接收到的訊息文字。在專案導覽器中，前往 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>
  );
};
```

提示：使用此元件來儲存您想要在訊息行呈現的不同屬性；例如，虛擬人偶 URL、使用者名稱，以及傳送訊息時的時間戳記。

辨識目前使用者所傳送的訊息

為了辨識目前使用者傳送的訊息，我們修改程式碼並建立 React 內容來儲存目前使用者的 `userId`。

在 `src` 目錄中建立新檔案，並將其命名為 `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>;
};
```

注意：此處我們使用 useState 勾點來儲存 userId 值。日後，您可以使用 setUserId 來變更使用者關聯內容或實現登入目的。

接下來，使用先前建立的內容來替換傳遞給 tokenProvider 的第一個參數中的 userId：

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

  // ...
}
```

在您的 Message 元件中，使用之前建立的 UserContext，宣告 isMine 變數，比對寄件者的 userId 與來自內容的 userId，並為目前使用者套用不同樣式的訊息。

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>
  );
};
```

建立訊息清單元件

MessageList 元件負責隨時間顯示聊天室的對話。MessageList 檔案是容納我們所有訊息的容器。Message 是 MessageList 中的一列。

在 src 目錄中建立新檔案，並將其命名為 MessageList。使用 ChatMessage 陣列類型的 messages 來定義 Props。在主體內部，映射我們的 messages 屬性並將 Props 傳遞給您的 Message 元件。

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>
  );
};
```

呈現聊天訊息清單

現在將新的 MessageList 加入您的主要 App 元件：

```
// 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>
);

// ...
```

所有拼圖現在都已準備就緒，您的 App 可以開始呈現聊天室收到的訊息。繼續進行以下操作，了解如何利用您建立的元件在聊天室中執行動作。

在聊天室中執行動作

在聊天室中傳送訊息和執行仲裁者動作，是您與聊天室互動的一些主要方式。在此處，您將了解如何使用各種 ChatRequest 物件在 Chatterbox 中執行常用動作，例如傳送訊息、刪除訊息以及中斷其他使用者的連線。

聊天室中的所有動作都遵循一個常見的模式：對於您在聊天室中執行的每個動作，都有一個對應的請求物件。對於每個請求，您在請求確認時都會收到一個對應的回應物件。

只要您的使用者在建立聊天字符時被授予正確的權限，他們就可以使用請求物件成功執行對應的操作，從而查看您可以在聊天室中執行哪些請求。

下面我們將說明如何傳送訊息和刪除訊息。

傳送訊息

SendMessageRequest 類別允許在聊天室中傳送訊息。在此處，您會使用自己在建立訊息輸入 (在本教學課程的第 1 部分) 中建立的元件來修改 App，以便傳送訊息請求。

若要開始，請使用 useState 勾點定義一個名為 isSending 的新布林屬性。使用此新屬性，透過 isSendDisabled 常數來切換 button HTML 元素的停用狀態。在您的 SendButton 事件處理常式中，清除 messageToSend 的值，並將 isSending 設定為 true。

由於您將從此按鈕進行 API 呼叫，新增 *isSending* 布林值可協助防止同時發生多個 API 呼叫，方法為在請求完成之前停用 *SendButton* 上的使用者互動。

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

// ...

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

// ...

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

// ...

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

// ...
```

透過建立新的 SendMessageRequest 執行個體、將訊息內容傳遞給建構函數來準備請求。設定 isSending 和 messageToSend 狀態後，呼叫 sendMessage 方法，將請求傳送至聊天室。最後，在收到確認或拒絕請求時清除 isSending 旗標。

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);
  }
};

// ...
```

試用 Chatterbox：嘗試使用您的 `MessageInput` 起草一則訊息，並點選 `SendButton` 來傳送訊息。您應該會看到已傳送的訊息在您之前建立的 `MessageList` 中呈現。

刪除訊息

若要從聊天室中刪除訊息，您需具備適當的能力。這樣的能力會在聊天字符 (對聊天室進行身分驗證時使用的字符) 初始化期間授予。就本節的目的而言，設定本機身分驗證/授權伺服器 (在本教學課程的第 1 部分) 中的 ServerApp 可讓您指定仲裁者能力。這是在應用程式中使用您在建立字符提供者 (也在第 1 部分) 中建立的 tokenProvider 物件完成。

在此處，您可以透過新增刪除訊息的函數來修改自己的 Message。

首先，開啟 App.tsx 並新增 DELETE_MESSAGE 功能。(capabilities 是 tokenProvider 函數的第二個參數。)

注意：這是您的 ServerApp 告知 IVS Chat API，與產生的聊天字符相關聯的使用者可以刪除聊天室中的訊息的方式。在實際情況中，您可能會有更複雜的後端邏輯來管理伺服器應用程式基礎結構中的使用者功能。

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']),
```

```
  }),
);

// ...
```

在接下來的步驟中，您將更新 Message 以顯示刪除按鈕。

開啟 Message，並使用初始值為 false 的 useState 勾點定義一個名為 isDeleting 的新布林狀態。使用此狀態，根據 isDeleting 的目前狀態將 Button 的內容更新為不同內容。在 isDeleting 為 true 時停用按鈕；這可以防止您同時嘗試提出兩個刪除訊息請求。

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>
  );
};
```

定義一個名為 onDelete 的新函數，該函數接受字串作為其參數之一並傳回 Promise。在您 Button 的動作關閉主體中，使用 setIsDeleting 在呼叫 onDelete 前後切換 isDeleting 布林值。若為字串參數，則傳入您的元件訊息 ID。

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>
  );
};
```

接下來，您將更新 MessageList 以反映 Message 元件的最新變更。

開啟 MessageList 並定義一個名為 onDelete 的新函數，該函數接受字串作為參數並傳回 Promise。更新您的 Message 並透過 Message 的屬性傳遞。新關閉中的字串參數將是您要刪除的訊息的 ID，該訊息則是從 Message 傳遞過來。

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

接下來，您將更新 App 以反映 MessageList 的最新變更。

在 App 中，定義名為 onDeleteMessage 的函數並將其傳遞給 MessageList onDelete 屬性：

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>
);

// ...
```

透過建立 DeleteMessageRequest 的新執行個體、將相關訊息 ID 傳遞至建構函數參數來準備請求，然後呼叫接受上述準備好的請求的 deleteMessage：

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);
};

// ...
```

接下來，您將更新 messages 狀態以反映新的訊息清單，此清單會忽略您剛剛刪除的訊息。

在 useEffect 勾點中，監聽 messageDelete 事件，並透過刪除具有與 message 參數相符 ID 的訊息來更新 messages 狀態陣列。

注意：目前使用者或 messageDelete 聊天室中的任何其他使用者刪除的訊息可能會引發事件。在事件處理常式中 (而不是在接著 deleteMessage 請求之後) 處理，可以讓您統一刪除訊息處理。

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

您現在可以從聊天應用程式的聊天室中刪除使用者。

後續步驟

實驗時，請嘗試在聊天室中實作其他動作，例如中斷其他使用者的連線。

Amazon IVS 聊天用戶端傳訊 SDK：React Native 教學課程第 1 部分：聊天室

這是由兩部分組成的教學課程的第一部分。您將透過使用 React Native 建置功能完整的應用程式，來學習使用 Amazon IVS 聊天用戶端傳訊 JavaScript SDK 的基礎知識。我們稱呼該應用程式為 Chatterbox。

目標對象是初次使用 Amazon IVS 聊天功能傳訊開發套件的經驗豐富的開發人員。您應該很熟悉 TypeScript 或 JavaScript 程式設計語言和 React Native 程式庫。

為了簡潔起見，我們將 Amazon IVS 聊天用戶端傳訊 JavaScript SDK 稱為 Chat JS SDK。

注意：在某些情況下，JavaScript 和 TypeScript 的程式碼範例是相同的，因此我們會將兩者的範例合併。

本教學課程的第一部分分為幾個部分：

1. the section called “設定本機身分驗證/授權伺服器”
2. the section called “建立 Chatterbox 專案”
3. the section called “與聊天室連線”
4. the section called “建立字符提供者”
5. the section called “觀察連線更新”
6. the section called “建立傳送按鈕元件”
7. the section called “建立訊息輸入”
8. the section called “後續步驟”

必要條件

- 熟悉 TypeScript 或 JavaScript 和 React Native 程式庫。如果不熟悉 React Native，請在 React Native 簡介中了解基礎知識。
- 閱讀並理解 IVS 聊天功能入門。
- 使用現有 IAM 政策中定義的 CreateChatToken 和 CreateRoom 功能建立 AWS IAM 使用者。(請參閱 IVS 聊天功能入門)。
- 確保將此使用者的私密/存取金鑰儲存在 AWS 憑證檔案中。如需指示，請參閱《AWS CLI 使用者指南》(特別是組態和憑證檔案設定)。

- 建立聊天室並保存其 ARN。請參閱IVS 聊天功能入門。(如果您未保存該 ARN，稍後可以使用主控台或 Chat API 來查詢。)
- 使用 NPM 或 Yarn 套件管理工具安裝 Node.js 14+ 環境。

設定本機身分驗證/授權伺服器

後端應用程式負責建立聊天室並產生 Chat JS SDK 需要的聊天字符，以便對聊天室的用戶端執行身分驗證和授權。您必須使用自己的後端，因為您無法將 AWS 金鑰安全地存放在行動應用程式中；成熟的攻擊者可擷取這些金鑰並取得 AWS 帳戶的存取權。

請參閱 Amazon IVS 聊天功能入門中的建立聊天字符。如流程圖所示，您的伺服器端應用程式會負責建立聊天字符。這意味著應用程式必須透過從伺服器端應用程式請求聊天字符，來提供自己產生聊天字符的方法。

在本節中，您將學習在後端建立字符提供者的基礎知識。我們使用快速架構建立即時本機伺服器，以管理使用本機 AWS 環境建立聊天字符的作業。

使用 NPM 建立空的 npm 專案。建立目錄來保存應用程式，並將其設置為工作目錄：

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

使用 `npm init` 為應用程式建立 `package.json` 檔案：

```
$ npm init
```

此命令會提示您輸入數個項目，包括應用程式的名稱和版本。現在，只需按 RETURN 即可接受其中大多數的預設值，但存在以下例外狀況：

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

按下 RETURN 接受建議的 `index.js` 預設檔案名稱，或輸入您想要的主檔案名稱。

立即安裝所需的依存項目：

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

`aws-sdk` 需要組態環境變數，這些變數會自動從位於根目錄中名為 `.env` 的檔案載入。若要進行設定，請建立名為 `.env` 的新檔案，並填寫缺少的組態資訊：

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

現在，我們使用您在 npm init 命令中輸入的名稱，在根目錄中建立一個進入點檔案。在這種情況下，我們使用 index.js 並匯入所有必要的套件：

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

現在建立新的 express 執行個體：

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

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

之後，您可以為字符提供者建立第一個端點 POST 方法。從請求主體 (roomId、userId、capabilities 和 sessionDurationInMinutes) 取得所需的參數：

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

新增必填欄位驗證：

```
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;
  }
});
```

準備好 POST 方法後，我們將 createChatToken 與 aws-sdk 整合來取得身分驗證/授權的核心功能：

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

在檔案結尾，為 express 應用程式新增連接埠接聽程式：

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

現在，您可以從專案的根目錄中使用下列命令執行伺服器：

```
$ node index.js
```

提示：此伺服器會在 https://localhost:3000 接受 URL 請求。

建立 Chatterbox 專案

首先，建立一個名稱為 chatterbox 的 React Native 專案。執行此命令：

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

或使用 TypeScript 範本建立 expo 專案。

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

您可以透過節點套件管理工具或 Yarn 套件管理工具整合 Chat JS SDK：

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

與聊天室連線

在這裡，您可以建立 ChatRoom 並使用異步方法連接到其中。此 ChatRoom 類別會管理使用者與 Chat JS SDK 的連線。若要成功連線到聊天室，您必須在 React 應用程式中提供 ChatToken 的執行個體。

導覽至在預設 chatterbox 專案中建立的 App 檔案，並刪除功能元件傳回的所有內容。不需要預先填入程式碼。此刻，App 相當空。

TypeScript/JavaScript：

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

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

export default function App() {
```

```
  return <Text>Hello!</Text>;
}
```

建立新的 ChatRoom 執行個體並使用 useState 勾點將其傳遞給狀態。其需要傳遞 regionOrUrl (託管聊天室的 AWS 區域) 和 tokenProvider (用於後續步驟中建立的後端身分驗證/授權流程)。

重要事項：您必須使用與在 Amazon IVS 聊天功能入門中建立聊天室的區域相同的 AWS 區域。該 API 是 AWS 區域服務。如需支援區域和 Amazon IVS 聊天功能 HTTPS 服務端點的清單，請參閱 Amazon IVS 聊天功能區域頁面。

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>;
}
```

建立字符提供者

下一步，我們需要建置 ChatRoom 建構函數所需的無參數 tokenProvider 函數。首先，我們將建立 fetchChatToken 函數，該函數將向您在 the section called “設定本機身分驗證/授權伺服器” 設定的後端應用程式發出 POST 請求。聊天字符包含開發套件成功建立聊天室連線所需的資訊。Chat API 使用這些字符作為驗證使用者身分、聊天室內功能和工作階段持續時間的安全方式。

在專案導覽器中，建立名為 fetchChatToken 的新 TypeScript/JavaScript 檔案。建立 backend 應用程式的擷取請求，並從回應中傳回 ChatToken 物件。新增建立聊天字符所需的請求主體屬性。使用針對 Amazon Resource Name (ARN) 定義的規則。這些屬性會記錄在 CreateChatToken 端點中。

注意：您在此處使用的 URL 與執行後端應用程式時本機伺服器建立的 URL 相同。

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),
  };
}
```

觀察連線更新

對聊天室連線狀態的變化做出反應，是製作聊天應用程式的重要一環。我們從訂閱相關事件開始：

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>;
}
```

接下來，我們需要提供讀取連線狀態的能力。我們使用 useState 勾點在 App 中建立一些本機狀態，並在每個接聽程式中設定連線狀態。

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>;
}
```

訂閱連線狀態後，顯示連線狀態並使用 `useEffect` 勾點內的 `room.connect` 方法連接到聊天室：

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,
  }
});

// ...
```

您已成功實作聊天室連線。

建立傳送按鈕元件

在本節中，您將建立傳送按鈕，該按鈕對每個連線狀態都提供不同的設計。傳送按鈕有助於在聊天室中傳送訊息。其還可以作為判斷是否可以/何時可以傳送訊息的視覺化指示器；例如，面對中斷的連線或過期的聊天工作階段。

首先，在 Chatterbox 專案的 `src` 目錄中建立新檔案並將其命名為 SendButton。然後，建立元件，該元件將為聊天應用程式顯示按鈕。匯出 SendButton 並將其匯入 App。在空白的 `<View></View>` 中新增 `<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>
);
```

接下來，在 App 中定義名為 onMessageSend 的函數並將其傳遞給 SendButton onPress 屬性。定義另一個名為 isSendDisabled 的變數 (可防止在聊天室未連接時傳送訊息) 並將其傳遞給 SendButton disabled 屬性。

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>
);

// ...
```

建立訊息輸入

Chatterbox 訊息列是您將與其互動以將訊息傳送到聊天室的元件。通常，其包含用於撰寫訊息的文字輸入和用於傳送訊息的按鈕。

若要建立 MessageInput 元件，請先在 src 目錄中建立新檔案並將其命名為 MessageInput。然後，建立輸入元件，該元件將為聊天應用程式顯示輸入。匯出 MessageInput 並將其匯入 App (在 <SendButton /> 之上)。

使用 useState 勾點建立名為 messageToSend 的新狀態，並以空字串作為其預設值。在應用程式主體中，將 messageToSend 傳遞給 MessageInput 的 value，並將 setMessageToSend 傳遞給 onMessageChange 屬性：

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',
  }
});
```

後續步驟

現在，您已經完成建置 Chatterbox 的訊息列，請繼續閱讀本 React Native 教學課程的第 2 部分：訊息和事件。

Amazon IVS 聊天用戶端傳訊 SDK：React Native 教學課程第 2 部分：訊息和事件

本教學課程的第二部分 (也是最後一部分) 分為幾個部分：

1. the section called “訂閱聊天訊息事件”
2. the section called “顯示收到的訊息”
   a. the section called “建立訊息元件”
   b. the section called “辨識目前使用者所傳送的訊息”
   c. the section called “呈現聊天訊息清單”
3. the section called “在聊天室中執行動作”
   a. the section called “傳送訊息”
   b. the section called “刪除訊息”
4. the section called “後續步驟”

注意：在某些情況下，JavaScript 和 TypeScript 的程式碼範例是相同的，因此我們會將兩者的範例合併。

先決條件

請確定您已完成本教學課程的第 1 部分：聊天室。

訂閱聊天訊息事件

當聊天室中發生事件時，`ChatRoom` 執行個體會使用事件進行通訊。若要開始實作聊天體驗，當其他人在其連線的聊天室中傳送訊息時，您需要向使用者顯示。

您可在此處訂閱聊天訊息事件。稍後，我們將說明如何更新您建立的訊息清單，此清單會隨每個訊息/事件進行更新。

在 `App` 的 `useEffect` 勾點內，訂閱所有訊息事件：

TypeScript/JavaScript：

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

useEffect(() => {
```

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

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

顯示收到的訊息

接收訊息是聊天體驗的核心部分。您可以使用 Chat JS SDK 設定程式碼，輕鬆接收來自連線至聊天室的其他使用者的事件。

稍後，我們將展示如何利用您在此處建立的元件在聊天室中執行動作。

在您的 App 中，使用名為 `messages` 的 `ChatMessage` 陣列類型定義一個名為 `messages` 的狀態：

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([]);
```

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

接下來，在 message 接聽程式函數中，將 message 附加至 messages 陣列：

TypeScript/JavaScript：

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

// ...

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

// ...
```

下面我們將逐步完成任務，以顯示收到的訊息：

1. the section called “建立訊息元件”
2. the section called “辨識目前使用者所傳送的訊息”
3. the section called “呈現聊天訊息清單”

建立訊息元件

Message 元件負責呈現聊天室收到的訊息內容。在本節中，您會建立用來呈現 App 中個別聊天訊息的訊息元件。

在 src 目錄中建立新檔案，並將其命名為 Message。傳入此元件的 ChatMessage 類型，並從 ChatMessage 屬性傳遞 content 字串，以顯示從聊天室訊息接聽程式接收到的訊息文字。在專案導覽器中，前往 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,
  },
});
```

提示：使用此元件來儲存您想要在訊息行呈現的不同屬性；例如，虛擬人偶 URL、使用者名稱，以及傳送訊息時的時間戳記。

辨識目前使用者所傳送的訊息

為了辨識目前使用者傳送的訊息，我們修改程式碼並建立 React 內容來儲存目前使用者的 `userId`。

在 `src` 目錄中建立新檔案，並將其命名為 `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;
```

注意：此處我們使用 useState 勾點來儲存 userId 值。日後，您可以使用 setUserId 來變更使用者關聯內容或實現登入目的。

接下來，使用先前建立的內容來替換傳遞給 tokenProvider 的第一個參數中的 userId。請務必將 SEND_MESSAGE 功能新增至字符提供者 (如下所述)；需要此功能以傳送訊息：

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']),
      }),
  );

  // ...
}
```

在您的 Message 元件中，使用之前建立的 UserContext，宣告 isMine 變數，比對寄件者的 userId 與來自內容的 userId，並為目前使用者套用不同樣式的訊息。

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',
  },
});
```

呈現聊天訊息清單

接著，請使用 `FlatList` 和 `Message` 元件列出訊息：

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>
);

// ...
```

所有拼圖現在都已準備就緒，您的 App 可以開始呈現聊天室收到的訊息。繼續進行以下操作，了解如何利用您建立的元件在聊天室中執行動作。

在聊天室中執行動作

傳送訊息和執行仲裁者動作，是您與聊天室互動的一些主要方式。在此處，您將了解如何使用各種聊天請求物件在 Chatterbox 中執行常用動作，例如傳送訊息、刪除訊息以及中斷其他使用者的連線。

聊天室中的所有動作都遵循一個常見的模式：對於您在聊天室中執行的每個動作，都有一個對應的請求物件。對於每個請求，您在請求確認時都會收到一個對應的回應物件。

只要您的使用者在建立聊天字符時被授予正確的功能，他們就可以使用請求物件成功執行對應的操作，從而查看您可以在聊天室中執行哪些請求。

下面我們將說明如何傳送訊息和刪除訊息。

傳送訊息

`SendMessageRequest` 類別允許在聊天室中傳送訊息。在此處，您會使用自己在建立訊息輸入 (在本教學課程的第 1 部分) 中建立的元件來修改 App，以便傳送訊息請求。

若要開始，請使用 `useState` 勾點定義一個名為 `isSending` 的新布林屬性。使用此新屬性，透過 `isSendDisabled` 常數來切換 `button` 元素的停用狀態。在您的 `SendButton` 事件處理常式中，清除 `messageToSend` 的值，並將 `isSending` 設定為 true。

由於您將從此按鈕進行 API 呼叫，新增 *isSending* 布林值可協助防止同時發生多個 API 呼叫，方法為在請求完成之前停用 *SendButton* 上的使用者互動。

注意：您必須將 SEND_MESSAGE 功能新增至字符提供者，才能傳送訊息，如上方辨識目前使用者所傳送的訊息所述。

TypeScript/JavaScript：

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

// ...

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

// ...

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

// ...

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

// ...
```

透過建立新的 SendMessageRequest 執行個體、將訊息內容傳遞給建構函數來準備請求。設定 isSending 和 messageToSend 狀態後，呼叫 sendMessage 方法，將請求傳送至聊天室。最後，在收到確認或拒絕請求時清除 isSending 旗標。

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);
  }
};

// ...
```

試用 Chatterbox：嘗試使用您的 MessageBar 起草一則訊息，並點選 SendButton 來傳送訊息。您應該會看到已傳送的訊息在您之前建立的 MessageList 中呈現。

刪除訊息

若要從聊天室中刪除訊息，您需具備適當的能力。這樣的能力會在聊天字符 (對聊天室進行身分驗證時使用的字符) 初始化期間授予。就本節的目的而言，設定本機身分驗證/授權伺服器 (在本教學課程的第 1 部分) 中的 ServerApp 可讓您指定仲裁者能力。這是在應用程式中使用您在建立字符提供者 (也在第 1 部分) 中建立的 tokenProvider 物件完成。

在此處，您可以透過新增刪除訊息的函數來修改自己的 Message。

首先，開啟 App.tsx 並新增 DELETE_MESSAGE 功能。(capabilities 是 tokenProvider 函數的第二個參數。)

注意：這是您的 ServerApp 告知 IVS Chat API，與產生的聊天字符相關聯的使用者可以刪除聊天室中的訊息的方式。在實際情況中，您可能會有更複雜的後端邏輯來管理伺服器應用程式基礎結構中的使用者功能。

TypeScript/JavaScript：

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

// ...

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

```
    }),
);

// ...
```

在接下來的步驟中，您將更新 Message 以顯示刪除按鈕。

定義一個名為 onDelete 的新函數，該函數接受字串作為其參數之一並傳回 Promise。若為字串參數，則傳入您的元件訊息 ID。

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',
  },
});
```

接下來，您將更新 `renderItem` 以反映 `FlatList` 元件的最新變更。

在 App 中，定義名為 `handleDeleteMessage` 的函數並將其傳遞給 `MessageList onDelete` 屬性：

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]);

// ...
```

透過建立 DeleteMessageRequest 的新執行個體、將相關訊息 ID 傳遞至建構函數參數來準備請求，然後呼叫接受上述準備好的請求的 deleteMessage：

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);
};

// ...
```

接下來，您將更新 messages 狀態以反映新的訊息清單，此清單會忽略您剛剛刪除的訊息。

在 useEffect 勾點中，監聽 messageDelete 事件，並透過刪除具有與 message 參數相符 ID 的訊息來更新 messages 狀態陣列。

注意：目前使用者或 messageDelete 聊天室中的任何其他使用者刪除的訊息可能會引發事件。在事件處理常式中 (而不是在接著 deleteMessage 請求之後) 處理，可以讓您統一刪除訊息處理。

TypeScript/JavaScript：

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

// ...

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

return () => {
```

```
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

您現在可以從聊天應用程式的聊天室中刪除使用者。

後續步驟

實驗時，請嘗試在聊天室中實作其他動作，例如中斷其他使用者的連線。

Amazon IVS 聊天用戶端傳訊 SDK：React 和 React Native 最佳實務

本文件說明使用 React and React Native 的 Amazon IVS 聊天功能傳訊 SDK 最重要的實務。此資訊應可讓您在 React 應用程式中建立典型聊天功能，並讓您了解深入探索 IVS 聊天功能傳訊 SDK 進階部分所需的背景知識。

建立聊天室初始化程式勾點

`ChatRoom` 類別包含核心聊天方法和接聽程式，用於管理連線狀態和接聽諸如已接收訊息或已刪除訊息之類的事件。我們會在這裡展示如何在勾點中正確儲存聊天執行個體。

實作

TypeScript

```
// useChatRoom.ts

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

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

  return { room };
};
```

JavaScript

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

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

  return { room };
};
```

注意：我們不會使用 setState 勾點的 dispatch 方法，因為您無法即時更新組態參數。SDK 可建立一次執行個體，但無法更新字符提供者。

重要事項：請使用一次 ChatRoom 初始化程式勾點，以初始化新的聊天室執行個體。

範例

TypeScript/JavaScript：

```
// ...

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

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

  // ...
};

// ...
```

接聽連線狀態

您也可以在聊天室勾點中訂閱連線狀態更新。

實作

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 };
};
```

聊天室執行個體提供者

若要在其他元件中使用勾點 (以避免屬性鑽研)，您可以使用 React `context` 建立聊天室提供者。

實作

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;
```

範例

建立 ChatRoomProvider 後，您可以透過 useChatRoomContext 使用執行個體。

重要事項：請只在您需要存取聊天畫面和中間其他元件之間的 context 時，才將提供者放在根層級，以避免接聽連線時不必要的重新渲染。否則，請將提供者放在盡可能靠近聊天畫面的位置。

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

// ...
```

建立訊息接聽程式

若要了解所有傳入訊息的最新資訊，您應訂閱 message 和 deleteMessage 事件。以下是幾個為元件提供聊天訊息的程式碼。

重要事項：為了提高效能，我們將 ChatMessageContext 從 ChatRoomProvider 分離出來，以免聊天訊息接聽程式在更新訊息狀態時發生多次重新渲染。請記得將 ChatMessageContext 套用至您會使用 ChatMessageProvider 的元件。

實作

TypeScript

```
// ChatMessagesContext.tsx

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

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

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

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

  return context;
};

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

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

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

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

    return () => {
      unsubscribeOnMessageDeleted();
      unsubscribeOnMessageReceived();
```

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

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

JavaScript

```
// ChatMessagesContext.jsx

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

const ChatMessagesContext = React.createContext(undefined);

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

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

  return context;
};

export const ChatMessagesProvider = ({ children }) => {
  const room = useChatRoomContext();

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

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

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

    return () => {
```

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

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

React 中的範例

重要事項：請記得以 ChatMessagesProvider 包裝您的訊息容器。Message 列是用於顯示訊息內容的範例元件。

TypeScript/JavaScript：

```
// your message list component...

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

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

  return (
    <React.Fragment>
      {messages.map((message) => (
        <MessageRow message={message} />
      ))}
    </React.Fragment>
  );
};
```

React Native 中的範例

根據預設，ChatMessage 包含 id，以自動作為每列中 FlatList 的 React金鑰；因此您不需要傳遞 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} />;
};
```

應用程式中的多個聊天室執行個體

如果您在應用程式中使用多個並行聊天室，建議您為每個聊天建立各自的提供者，並將其用於聊天提供者。在此範例中，我們建立了 Help Bot 和 Customer Help 聊天。我們為兩者都建立了提供者。

TypeScript

```
// SupportChatProvider.tsx

import React from 'react';
import { SUPPORT_ROOM_ID, SOCKET_URL } from '../../config';
```

```
import { tokenProvider } from '../tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SupportChatProvider = ({ children }: { children: React.ReactNode }) =>
 {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SUPPORT_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};

// SalesChatProvider.tsx

import React from 'react';
import { SALES_ROOM_ID, SOCKET_URL } from '../../config';
import { tokenProvider } from '../tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SalesChatProvider = ({ children }: { children: React.ReactNode }) => {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SALES_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};
```

JavaScript

```
// SupportChatProvider.jsx

import React from 'react';
import { SUPPORT_ROOM_ID, SOCKET_URL } from '../../config';
import { tokenProvider } from '../tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SupportChatProvider = ({ children }) => {
  const { room } = useChatRoom({
```

```
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SUPPORT_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};

// SalesChatProvider.jsx

import React from 'react';
import { SALES_ROOM_ID, SOCKET_URL } from '../../config';
import { tokenProvider } from '../tokenProvider';
import { ChatRoomProvider } from './ChatRoomContext';
import { useChatRoom } from './useChatRoom';

export const SalesChatProvider = ({ children }) => {
  const { room } = useChatRoom({
    regionOrUrl: SOCKET_URL,
    tokenProvider: () => tokenProvider(SALES_ROOM_ID, ['SEND_MESSAGE']),
  });

  return <ChatRoomProvider value={room}>{children}</ChatRoomProvider>;
};
```

React 中的範例

現在您可以運用使用相同 ChatRoomProvider 的不同聊天提供者。稍後，您可以在每個畫面/檢視中重複使用相同的 useChatRoomContext。

TypeScript/JavaScript：

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

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

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

React Native 中的範例

TypeScript/JavaScript：

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

const App = () => {
  return (
    <Stack.Navigator>
      <Stack.Screen name="SupportChat">
        <SupportChatProvider>
          <SupportChatScreen />
        </SupportChatProvider>
      </Stack.Screen>
      <Stack.Screen name="SalesChat">
        <SalesChatProvider>
          <SalesChatScreen />
        </SalesChatProvider>
      </Stack.Screen>
    </Stack.Navigator>
  );
};
```

TypeScript/JavaScript：

```
// SupportChatScreen.tsx / SupportChatScreen.jsx

// ...

const SupportChatScreen = () => {
  const room = useChatRoomContext();
```

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

  return (
    <>
      <Button title="Connect" onPress={handleConnect} />
      <MessageListContainer />
    </>
  );
};

// SalesChatScreen.tsx / SalesChatScreen.jsx

// ...

const SalesChatScreen = () => {
  const room = useChatRoomContext();

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

  return (
    <>
      <Button title="Connect" onPress={handleConnect} />
      <MessageListContainer />
    </>
  );
};
```

Amazon IVS 聊天功能安全性

雲端安全是 AWS 最重視的一環。身為 AWS 的客戶，您將能從資料中心和網路架構中獲益，這些都是專為最重視安全的組織而設計的。

安全是 AWS 與您共同的責任。共同責任模型 將此描述為雲端的安全和雲端內的安全：

- 雲端本身的安全：AWS 負責保護在 AWS Cloud 中執行 AWS 服務的基礎設施。AWS 也提供您可安全使用的服務。在 AWS 合規計畫中，第三方稽核員會定期測試並驗證我們的安全功效。
- 雲端內部的安全：您的責任取決於所使用的 AWS 服務。您也必須對資料敏感度、組織要求，以及適用法律和法規等其他因素負責。

本文件有助於您了解如何在使用 Amazon IVS 聊天功能時套用共同責任模型。下列各主題將說明如何配置 Amazon IVS 聊天功能，以達成您的安全性與合規目標。

主題

資料保護

對於傳送至 Amazon Interactive Video Service (IVS) 聊天功能的資料，有下列資料保護措施：

- Amazon IVS 聊天功能流量使用 WSS 來保證傳輸中的資料安全。
- Amazon IVS 聊天功能符記使用 KMS 客戶管理的金鑰進行加密。

Amazon IVS 聊天功能不需要您提供任何客戶 (最終使用者) 資料。在聊天室、輸入或輸入安全群組中沒有需要您提供客戶 (最終使用者) 資料的欄位。

請勿將客戶 (最終使用者) 帳號等敏感識別資訊填入 Name (名稱) 等自由格式欄位。這包括當您使用 Amazon IVS 主控台或 API、AWS CLI 或 AWS SDK 時。您輸入到 Amazon IVS 聊天功能的任何資料都可能包含在診斷日誌中。

串流並非端對端加密；串流可能會在 IVS 網路內部以未加密的方式傳輸，以供處理。

識別與存取管理

AWS Identity and Access Management (IAM) 是一種 AWS 服務，可協助帳戶管理員安全地控制對 AWS 資源的存取。請參閱《IVS 低延遲串流使用者指南》中的 Identity and Access Management。

物件

根據您在 Amazon IVS 中所進行的工作而定，IAM 的使用方式會不同。請參閱《IVS 低延遲串流使用者指南》中的對象。

Amazon IVS 如何與 IAM 搭配運作

在您發出 Amazon IVS API 請求之前，您必須先建立一個或多個 IAM 身分身分 (使用者、群組和角色) 和 IAM 政策，然後將政策連接至身分。傳播許可最多需要幾分鐘的時間；在此之前，API 請求會遭到拒絕。

若要更好地了解 Amazon IVS 如何與 IAM 搭配運作，請參閱 IAM 使用者指南中的與 IAM 搭配運作的 AWS 服務。

身分

您可建立的 IAM 身分，以便為 AWS 帳戶中的人員和程序提供身分驗證。IAM 群組是 IAM 使用者集合，您可將它們視為單位進行管理。請參閱 IAM 使用者指南中的身分身分 (使用者、群組和角色)。

政策

政策是由元素組成的 JSON 許可政策文件。請參閱《IVS 低延遲串流使用者指南》中的政策。

Amazon IVS 聊天功能支援三種元素：

- 動作 – Amazon IVS 聊天功能政策動作在動作之前使用 `ivschat` 字首。例如，若要授予某人使用 Amazon IVS 聊天功能 `CreateRoom` API 方法建立 Amazon IVS 聊天功能聊天室的許可，請在該人員的政策中包括 `ivschat:CreateRoom` 動作。政策陳述式必須包含 `Action` 或 `NotAction` 元素。

- 資源 – Amazon IVS 聊天功能聊天室資源具有以下 ARN 格式：

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

  例如，若要在陳述式中指定 VgNkEJg0VX9N 聊天室，請使用此 ARN：

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

  有些 Amazon IVS 聊天功能動作無法對特定資源執行，例如用來建立資源的動作。在那些情況下，您必須使用萬用字元 (*)：

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

- 條件 – Amazon IVS 聊天功能支援某些全域條件索引鍵：`aws:RequestTag`、`aws:TagKeys` 和 `aws:ResourceTag`。

您可以在政策中使用變數做為預留位置。例如，您可以只在使用者使用其 IAM 使用者名稱標記時，將存取資源的許可授予該 IAM 使用者。請參閱 IAM 使用者指南中的變數和標籤。

Amazon IVS 提供 AWS 受管政策，可用於向身分授予一組預先設定的許可 (唯讀或完整存取權)。您可以選擇使用受管政策，而非下面所示的身分型政策。如需詳細資訊，請參閱 Amazon IVS 的受管政策。

以 Amazon IVS 標籤為基礎的授權

您可以將標籤附接至 Amazon IVS 聊天功能資源，或是在請求中將標籤傳遞至 Amazon IVS 聊天功能。若要根據標籤控制存取，請使用 `aws:ResourceTag/key-name`、`aws:RequestTag/key-name` 或 `aws:TagKeys` 條件金鑰，在政策的條件元素中，提供標籤資訊。如需標記 Amazon IVS 聊天功能資源的詳細資訊，請參閱 IVS 聊天功能 API 參考中的「標記」。

角色

請參閱 IAM 使用者指南中的 IAM 角色和臨時安全登入資料。

IAM 角色是您 AWS 帳戶中具備特定許可的實體。

Amazon IVS 支援使用臨時安全登入資料。您可以搭配聯合使用暫時登入資料登入、擔任 IAM 角色，或是擔任跨帳戶角色。您取得暫時安全登入資料的方式是透過呼叫 AWS Security Token Service API 操作 (例如 `AssumeRole` 或 `GetFederationToken`)。

特權和非特權存取

API 資源擁有特權存取。可以透過私有通道設定非特權播放存取；請參閱設定私有通道。

政策的最佳實務

請參閱 IAM 使用者指南中的 IAM 最佳實務。

基於身分的政策相當強大。它們可以判斷您帳戶中的某個人員是否可以建立、存取或刪除 Amazon IVS 資源。這些動作可能會讓您的 AWS 帳戶產生成本。請遵循下列建議：

- 授予最低權限 – 當您建立自訂政策時，請只授予執行任務所需要的許可。以最小一組許可開始，然後依需要授予額外的許可。如此做比一開始使用太寬鬆的許可更為安全，然後稍後再嘗試將它們限縮。尤其是，保留 `ivschat:*` 進行管理員存取；不要在應用程式中使用它。
- 為敏感操作啟用多重要素驗證 (MFA) – 為了增加安全，請要求 IAM 使用者使用 MFA 來存取敏感資源或 API 操作。
- 使用政策條件以增加安全 – 在切實可行的範圍中，請定義允許存取資源的基於身分的政策條件。例如，您可以撰寫條件，來指定必須發出各種請求的可允許 IP 地址範圍。您也可以撰寫條件，只在指定的日期或時間範圍內允許請求，或要求使用 SSL 或 MFA。

基於身分的政策範例

使用 Amazon IVS 主控台。

若要存取 Amazon IVS 主控台，您必須有一組符合最低限制的許可，讓您可以列出並檢視您的 AWS 帳戶中 Amazon IVS 聊天功能資源的詳細資訊。如果您建立比最低必要許可更嚴格的基於身分的政策，則對於具有該政策的身分而言，主控台將無法如預期運作。若要確保存取 Amazon IVS 主控台，請將以下政策連接到身分 (請參閱 IAM 使用者指南中的新增和移除 IAM 許可)。

使用以下政策的各部分可存取：

- 所有 Amazon IVS 聊天功能 API 端點
- 您的 Amazon IVS 聊天功能服務配額
- 列出 lambda 並為所選 lambda 新增許可以進行 Amazon IVS 聊天功能管制
- Amazon CloudWatch 為您的聊天工作階段取得指標

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "ivschat:*",
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Action": [
        "servicequotas:ListServiceQuotas"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Action": [
        "cloudwatch:GetMetricData"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Action": [
        "lambda:AddPermission",
        "lambda:ListFunctions"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

Amazon IVS 聊天功能的以資源為基礎的政策

您必須授予 Amazon IVS 聊天功能服務調用 lambda 資源的許可，方可審查訊息。為此，請按照為 AWS Lambda 使用以資源為基礎的政策中的說明 (在 AWS Lambda 開發人員指南中) 並填寫下面指定的欄位。

若要控制對 lambda 資源的存取，您可以使用以下列項目為基礎的條件：

- `SourceArn` – 我們的政策範例使用萬用字元 (*) 來允許您帳戶中的所有聊天室調用 lambda。您也可以選擇在您的帳戶中指定一個聊天室，以僅允許該聊天室調用 lambda。

- SourceAccount：在下面的政策範例中，AWS 帳戶 ID 是 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/*"
            }
         }
      }
   ]
}
```

故障診斷

請參閱《IVS 低延遲串流使用者指南》中的疑難排解，以取得診斷和修正使用 Amazon IVS 聊天功能和 IAM 時可能遇到之常見問題的相關資訊。

Amazon IVS 的受管政策

AWS 托管策略 是由 AWS 创建和管理的独立策略。請參閱《IVS 低延遲串流使用者指南》中的適用於 Amazon IVS 的受管政策。

使用 Amazon IVS 的服務連結角色

Amazon IVS 使用 AWS IAM 服務連結角色。請參閱《IVS 低延遲串流使用者指南》中的使用適用於 Amazon IVS 的服務連結角色。

記錄和監控

要記錄效能和/或操作，請使用 Amazon CloudTrail。請參閱《IVS 低延遲串流使用者指南》中的使用 AWS CloudTrail 日誌記錄 Amazon IVS API 呼叫。

事件反應

若要偵測事件或提醒事件，您可以透過 Amazon EventBridge 事件監控串流的運作狀態。請參閱如何搭配 Amazon IVS 使用 Amazon EventBridge：適用於低延遲串流和即時串流。

使用 AWS Health 儀板表，了解 Amazon IVS 整體運作狀態的資訊 (依區域)。

恢復能力

IVS API 使用 AWS 全球基礎設施，並以 AWS 區域與可用區域為中心而建置。請參閱《IVS 低延遲串流使用者指南》中的恢復能力。

基礎設施安全性

作為一種受管服務，Amazon IVS 受到 AWS 全球網路安全程序的保護。在安全、身分與合規最佳實務中進行了說明。

API Calls (API 呼叫)

您可使用 AWS 發佈的 API 呼叫，透過網路存取 Amazon IVS。請參閱《IVS 低延遲串流使用者指南》中「基礎設施安全」下的 API 呼叫。

Amazon IVS 聊天功能

Amazon IVS 聊天功能訊息擷取和傳遞透過與我們的邊緣的加密 WSS 連線進行。Amazon IVS 訊息傳遞 API 使用加密的 HTTPS 連線。與影片串流和播放一樣，需要 TLS 版本 1.2 或更高版本，並且訊息資料可能會在內部未加密地傳輸以進行處理。

Service Quotas (聊天)

以下是 Amazon Interactive Video Service (IVS) 聊天功能端點、資源和其他操作的服務配額與限制。服務配額 (也稱為限制) 是您 AWS 帳戶的服務資源或操作數目最大值。也就是說，這些限制以 AWS 帳戶為依據，除非表格中另有說明。另請參閱 AWS Service Quotas。

可使用端點透過程式設計方式連接到 AWS 服務。另請參閱 AWS 服務端點。

所有配額均按區域執行。

Service Quotas 增加

對於可調整的配額，您可以透過 AWS 主控台請求增加速率。也可以使用主控台檢視服務配額的相關資訊。

API 呼叫速率配額不可調整。

API 呼叫速率配額

端點類型	端點	預設
簡訊	DeleteMessage	100 TPS
簡訊	DisconnectUser	100 TPS
簡訊	SendEvent	100 TPS
聊天權杖	CreateChatToken	200 TPS
記錄組態	CreateLoggingConfiguration	3 TPS
記錄組態	DeleteLoggingConfiguration	3 TPS
記錄組態	GetLoggingConfiguration	3 TPS
記錄組態	ListLoggingConfigurations	3 TPS
記錄組態	UpdateLoggingConfiguration	3 TPS
聊天室	CreateRoom	5 TPS

端點類型	端點	預設
聊天室	DeleteRoom	5 TPS
聊天室	GetRoom	5 TPS
聊天室	ListRooms	5 TPS
聊天室	UpdateRoom	5 TPS
標籤	ListTagsForResource	10 TPS
標籤	TagResource	10 TPS
標籤	UntagResource	10 TPS

其他配額

資源或功能	預設	可調整	描述
並行聊天連線	50,000	是	一個 AWS 區域 中所有聊天室每個帳戶的並行聊天連線數目上限。
記錄組態	10	是	目前 AWS 區域 中每個帳戶可建立的記錄組態的數目上限。
訊息審查處理常式逾時時間	200	否	目前 AWS 區域 中所有訊息審查處理常式的逾時時間 (以毫秒為單位)。如果超過此值，則根據您為訊息審查處理常式設定的 fallbackResult 欄位的值允許或拒絕訊息。
您所有聊天室的 DeleteMessage 請求速率	100	是	您的所有聊天室每秒可以發出的 DeleteMessage 請求次數的上限。請求可以來自 Amazon IVS 聊天功能 API 或

資源或功能	預設	可調整	描述
			Amazon IVS 聊天功能 訊息 API (WebSocket)。
您所有聊天室的 DisconnectUser 請求速率	100	是	您的所有聊天室每秒可以發出的 DisconnectUser 請求次數的上限。請求可以來自 Amazon IVS 聊天功能 API 或 Amazon IVS 聊天功能 訊息 API (WebSocket)。
每個連線的訊息請求速率	10	否	聊天連線每秒可以發出的訊息請求次數上限。
您所有聊天室的 SendMessage 請求速率	1000	是	您的所有聊天室每秒可以發出的 SendMessage 請求次數的上限。這些請求來自 Amazon IVS 聊天功能訊息 API (WebSocket)。
每個聊天室的 SendMessage 請求速率	100	否 (但可透過 API 設定)	您的任何一個聊天室每秒可以發出的 SendMessage 請求次數的上限。這可以透過 CreateRoom 和 UpdateRoom 的 `maximumMessageRatePerSecond` 欄位進行設定。這些請求來自 Amazon IVS 聊天功能訊息 API (WebSocket)。
聊天室	50,000	是	每個 AWS 區域 中每個帳戶的聊天室數量上限。

Service Quotas 與 CloudWatch 用量指標整合

您可以透過 CloudWatch 用量指標，使用 CloudWatch 主動管理您的服務配額。您可使用這些指標，在 CloudWatch 圖表和儀表板中視覺化目前的服務使用狀況。Amazon IVS 聊天功能用量指標對應 Amazon IVS 聊天功能服務配額。

您可以使用 CloudWatch 指標數學函數，以圖表顯示這些資源的服務配額。您也可以設定警示，在您的用量接近服務配額時發出警示。

若要存取用量指標：

1. 開啟 Service Quotas 主控台，網址為 https://console.aws.amazon.com/servicequotas/
2. 在導覽窗格中，選擇 AWS services (AWS 服務)。
3. 從 AWS 服務清單中，搜尋並選取 Amazon Interactive Video Service 聊天功能。
4. 在 Service quotas (服務配額) 清單中，選取感興趣的服務配額。開啟新頁面，其中包含服務配額/指標的相關資訊。

或者，您也可以透過 CloudWatch 主控台取得這些指標。在 AWS Namespaces (AWS 命名空間) 中，選擇 Usage (用量)。然後，在服務清單中選擇 IVS 聊天功能。(請參閱監控 Amazon IVS 聊天功能。)

在 AWS/用量命名空間中，Amazon IVS 聊天功能會提供下列指標：

指標名稱	描述
`ResourceCount`	您的帳戶中正在執行的特定資源數量。資源由與指標相關聯的維度定義。 有效統計資訊：最大值 (1 分鐘內使用的最大資源數量)。

以下維度用於改進用量指標：

維度	描述
Service (服務)	包含該資源的 AWS 服務的名稱。有效值：`IVS Chat`。
類別	正在追蹤的資源類別。有效值：`None`。
Type	正在追蹤的資源類型。有效值：`Resource`。

維度	描述
資源	AWS 資源的名稱。有效值：`ConcurrentChatConnections` 。 ConcurrentChatConnections 用量指標是 AWS/IVSChat 命名空間 (使用「無」維度) 中指標的副本，如監控 Amazon IVS 聊天功能中所述。

為用量指標建立 CloudWatch 警示

若要根據 Amazon IVS 聊天功能用量指標建立 CloudWatch 警示：

1. 從 Service Quotas 主控台中，選擇感興趣的服務配額，如上所述。目前，只能為 ConcurrentChatConnections 建立警示。
2. 在 Amazon CloudWatch alarms (Amazon CloudWatch 警示) 部分中，選擇 Create (建立)。
3. 在 Alarm threshold (警示閾值) 下拉式清單中，選擇您想要將其設為警示值的已套用配額值的百分比。
4. 對於 Alarm name (警示名稱)，輸入警示的名稱。
5. 選取 Create (建立)。

疑難排解常見問答集

本文件描述了 Amazon Interactive Video Service (IVS) 聊天功能的最佳實務和疑難排解提示。與 IVS 聊天功能相關的行為通常和與 IVS 影片相關的行為不同。如需詳細資訊，請參閱 Amazon IVS 聊天功能入門。

主題：

為什麼刪除聊天室後，IVS 聊天功能並未中斷連線？

刪除聊天室資源時，如果聊天室正在使用中，連線至聊天室的聊天用戶端不會自動中斷連線。當聊天應用程式重新整理聊天字符時，連線會中斷。或者，您必須手動中斷所有使用者，以將所有使用者從聊天室中移除。

詞彙表

另請參閱 AWS 詞彙表。在下表中，LL 代表 IVS 低延遲串流；RT 代表 IVS 即時串流。

術語	描述	LL	RT	聊天
AAC	進階音訊編碼。AAC 是失真數位音訊壓縮的音訊編碼標準。AAC 旨在作為 MP3 格式的後繼者，在相同位元速率下通常比 MP3 實現更高的音效品質。AAC 已被 ISO 和 IEC 標準化，作為 MPEG-2 和 MPEG-4 規格的一部分。	✓	✓	
自適性位元速率串流	自適性位元速率 (ABR) 串流可讓 IVS 播放器在連線品質下降時切換至較低位元速率，並在連線品質改善時切換回較高位元速率。	✓		
自適性串流	請參閱使用聯播進行分層編碼。		✓	
管理使用者	對 AWS 帳戶中可用的資源和服務具有管理存取權的 AWS 使用者。請參閱《AWS 設定使用者指南》中的術語。	✓	✓	✓
ARN	Amazon Resource Name，AWS 資源的唯一識別符。特定 ARN 格式視資源類型而定。如需 IVS 資源使用的 ARN 格式，請參閱服務授權參考。	✓	✓	✓
長寬比	描述影格寬度與影格高度的比率。例如，16:9 是對應於 Full HD 或 1080p 解析度的長寬比。	✓	✓	
音訊模式	針對不同類型的行動裝置使用者及其使用的設備優化的預設或自訂音訊組態。請參閱 IVS 廣播 SDK：行動音訊模式 (即時串流)。		✓	
AVC、H.264、MPEG-4 第 10 部分	進階影片編碼，亦稱為 H.264 或 MPEG-4 第 10 部分，用於失真數位視訊壓縮的影片壓縮標準。	✓	✓	

術語	描述	LL	RT	聊天
背景替換	一種攝影機濾鏡，可讓即時串流創作者變更背景。請參閱 IVS 廣播 SDK：第三方攝影機濾鏡 (即時串流) 中的背景替換。		✓	
位元速率	每秒傳輸或接收位元數的串流指標。	✓	✓	
廣播、廣播者	串流、實況主的其他術語。	✓		
緩衝	當播放裝置在應播放內容之前無法下載內容時發生的情況。緩衝可以透過多種方式建立清單檔案：內容可能會隨機停止和開始 (也稱為卡頓)、內容可能長時間停止 (也稱為凍結)，或 IVS 播放器可能暫停播放。	✓	✓	
位元組範圍播放清單	比標準 HLS 播放清單更精細的播放清單。標準 HLS 播放清單由最多 10 秒的媒體檔案組成。使用位元組範圍播放清單，區段持續時間會與為串流設定的關鍵影格間隔相同。 位元組範圍播放清單僅適用於自動錄製到 S3 儲存貯體的廣播。它是在 HLS 播放清單之外建立的。請參閱自動錄製到 Amazon S3 (低延遲串流) 中的位元組範圍播放清單。	✓		
CBR	固定位元速率，一種編碼器的速率控制方法，可在影片的整個播放過程中保持一致的位元速率，無論廣播期間發生什麼情況。可以填充動作中的間歇以實現所需的位元速率，並且可以透過調整編碼品質以匹配目標位元速率來量化峰值。我們強烈建議您使用 CBR 而非 VBR。	✓	✓	
CDN	內容交付網路或內容分發網路，一種地理位置分散的解決方案，透過使其更接近使用者所在位置來優化串流影片等內容的交付。	✓		

術語	描述	LL	RT	聊天
頻道	儲存串流組態的 IVS 資源，包括擷取伺服器、串流金鑰、播放 URL 和錄製選項。實況主會使用與頻道關聯的串流金鑰來開始廣播。廣播期間產生的所有指標和事件都與頻道資源關聯。	✓		
頻道類型	確定頻道允許的解析度和影格率。請參閱 IVS 低延遲串流 API 參考中的頻道類型。	✓		
聊天記錄	一種進階選項，可以透過將日誌記錄組態與聊天室關聯來加以啟用。			✓
聊天室	一種 IVS 資源，用於儲存聊天工作階段的組態，包括選用功能，例如訊息審查處理常式和聊天記錄。請參閱 IVS 聊天功能入門中的步驟 2：建立聊天室。			✓
用戶端合成	使用主機裝置混合來自階段參與者的音訊和影片串流，然後將這些串流作為複合串流傳送至 IVS 頻道。這樣可以更好地控制合成的外觀，但代價是用戶端資源利用率更高，以及影響觀眾的階段或主持人問題的風險更高。 另請參閱伺服器端合成。	✓	✓	
CloudFront	Amazon 提供的 CDN 服務。	✓		
CloudTrail	一種 AWS 服務，用於收集、監控、分析和保留來自 AWS 與外部來源的事件和帳戶活動。請參閱使用 AWS CloudTrail 記錄 IVS API 呼叫。	✓	✓	✓
CloudWatch	一種 AWS 服務，用於監控應用程式、回應效能變更、優化資源使用，以及深入了解運作狀態。您可以使用 CloudWatch 監控 IVS 指標；請參閱監控 IVS 即時串流和監控 IVS 低延遲串流。	✓	✓	✓
合成	將來自多個來源的音訊和影片串流合併為單一串流的程序。	✓	✓	

術語	描述	LL	RT	聊天
合成管道	合併多個串流並對產生的串流進行編碼所需的一系列處理步驟。	✓	✓	
壓縮	使用比原始表示法更少的位元對資訊進行編碼。任何特定的壓縮都是無失真或失真壓縮。無失真壓縮透過識別和消除統計冗餘來減少位元。無失真壓縮不會遺失任何資訊。失真壓縮透過移除不必要的或不太重要的資訊來減少位元。	✓	✓	
控制平台	儲存有關 IVS 資源的資訊 (例如頻道、階段或聊天室)，並提供建立和管理這些資源的介面。它是區域性的 (基於 AWS 區域)。	✓	✓	✓
CORS	跨來源資源分享，一種 AWS 功能，可讓在一個網域中載入的用戶端 Web 應用程式與不同網域中的 S3 儲存貯體等資源進行互動。您可以根據標頭、HTTP 方法和原始網域設定存取權。請參閱《Amazon Simple Storage Service 使用者指南》中的使用跨來源資源分享 (CORS) – Amazon Simple Storage Service。	✓		
自訂影像來源	IVS 廣播 SDK 提供的介面，可讓應用程式提供自己的影像輸入，而不是僅限於預設攝影機。	✓	✓	
資料平面	將資料從入口傳輸至出口的基礎設施。它根據控制平面中管理的組態運作，且不限於 AWS 區域。	✓	✓	✓
編碼器、編碼	將影片和音訊內容轉換為適合串流的數位格式的程序。編碼可以基於硬體或軟體。	✓	✓	
事件	IVS 發布給 AmazonEventBridge 監控服務的自動通知。事件代表串流資源 (例如階段或合成管道) 的狀態或運作狀態變更。請參閱將 Amazon EventBridge 與 IVS 低延遲串流搭配使用和將 Amazon EventBridge 與 IVS 即時串流搭配使用。	✓	✓	✓

術語	描述	LL	RT	聊天
FFmpeg	一種免費的開放原始碼軟體專案，由一套用於處理影片和音訊檔案與串流的庫和程式組成。FFmpeg 提供了一個跨平台解決方案來錄製、轉換和串流音訊和影片。	✓		
分段串流	當廣播中斷連接，然後在頻道的錄製組態中指定的時間間隔內重新連接時建立。產生的多個串流視為單一廣播並一起合併到單一錄製的串流。請參閱自動錄製到 Amazon S3 (低延遲串流) 中的合併分段串流。	✓		
影格播放速率	每秒傳輸或接收影片影格數的串流指標。	✓	✓	
HLS	HTTP 即時串流 (HLS)，一種 HTTP 型自適性位元速率串流通訊協定，用於向觀眾交付 IVS 串流。	✓		
HLS 播放清單	組成串流的媒體片段清單。標準 HLS 播放清單由最多 10 秒的媒體檔案組成。HLS 還支援更精細的位元組範圍播放清單。	✓		
主機	將影片和/或音訊傳送至階段的即時事件參與者。		✓	
IAM	Identity and Access Management，一種 AWS 服務，可讓使用者安全地管理身分以及對 AWS 服務和資源 (包括 IVS) 的存取。	✓	✓	✓
擷取	IVS 程序，用於從主持人或廣播者接收影片串流以進行處理或交付給觀眾或其他參與者。	✓	✓	
擷取伺服器	接收影片串流並將其交付給轉碼系統，在該系統中，串流轉碼複用或轉碼為 HLS，以便交付給觀眾。 擷取伺服器是特定的 IVS 元件，用於接收頻道的串流以及擷取協定 (RTMP、RTMPS)。請參閱 IVS 低延遲串流功能入門中有關建立頻道的資訊。		✓	

術語	描述	LL	RT	聊天
交錯影片	僅傳輸和顯示後續影格的奇數行或偶數行，以在不耗用額外頻寬的情況下實現影格率的感知加倍。由於影片品質問題，我們不建議使用交錯影片。	✓	✓	
JSON	JavaScript 物件標記法，一種開放式標準檔案格式，它使用人類可讀取的文字來傳輸資料物件，其中包含屬性值對和陣列資料類型或其他可序列化值。	✓	✓	✓
關鍵影格、差異影格、關鍵影格間隔	關鍵影格 (亦稱為內部編碼或 i 影格) 是影片中影像的全影格。後續影格，差異影格 (亦稱為預測或 p 影格) 僅包含已變更的資訊。關鍵影格將在串流內出現多次，具體取決於編碼器中定義的關鍵影格間隔。	✓	✓	
Lambda	用於執行程式碼 (稱為 Lambda 函數) 而無需佈建任何伺服器基礎設施的 AWS 服務。Lambda 函數可以執行以回應事件和調用請求，或根據排程執行。例如，IVS 聊天功能使用 Lambda 函數為聊天室啟用訊息審查。	✓	✓	✓
延遲、玻璃到玻璃延遲	資料傳輸中的延遲。IVS 將延遲範圍定義為： • 低延遲：3 秒以下 • 即時延遲：300 毫秒以下 顯示屏到顯示屏延遲是指從攝影機擷取即時串流到串流出現在檢視器螢幕上的延遲。	✓	✓	
使用聯播進行分層編碼	支援同時編碼和發布具有不同品質等級的多個影片串流。請參閱即時串流優化中的自適性串流：使用聯播進行分層編碼。		✓	
訊息審查處理常式	可讓 IVS 聊天功能客戶在將使用者聊天訊息交付至聊天室之前自動審查/篩選使用者聊天訊息。它是透過將 Lambda 函數與聊天室關聯來啟用的。請參閱聊天訊息審查處理常式中的建立 Lambda 函數。			✓

術語	描述	LL	RT	聊天
混音器	IVS 行動廣播 SDK 的功能，可取得多個音訊和影片來源並產生單一輸出。它支援管理代表來源的螢幕影片和音訊元素，例如攝影機、麥克風、螢幕擷取畫面以及應用程式產生的音訊和影片。然後，輸出可以串流至 IVS。請參閱《IVS 廣播 SDK：混音器指南 (低延遲串流)》中的為混音設定廣播工作階段。	✓		
多主持人串流	將來自多位主持人的串流合併為單一串流。這可以使用用戶端或伺服器端合成來完成。 多主持人串流可以實現邀請觀眾上台問答、主持人比賽、影片聊天、主持人當眾對話等情境。		✓	
多變體播放清單	可用於廣播的所有變體串流的索引。	✓		
OAC	原始存取控制，一種限制對 S3 儲存貯體的存取的機制，以便只能透過 CloudFront CDN 提供錄製串流等內容。	✓		
OBS	Open Broadcaster Software，用於影片錄製和即時串流的免費開放原始碼軟體。OBS 為桌面出版提供了一種替代方案 (IVS 廣播 SDK)。熟悉 OBS 的更資深的實況主可能更喜歡它，因為它具有進階生產功能，例如轉換場景，混合音訊和添加圖形浮水印。	✓	✓	
參與者	以主持人或觀眾身分連線至階段的即時使用者。		✓	
參與者權杖	當即時事件參與者加入階段時對其進行身分驗證。參與者字符也可控制參與者是否可以將影片傳送至階段。		✓	

術語	描述	LL	RT	聊天
播放字符、播放金鑰對	一種授權機制，可讓客戶限制私有頻道上的影片播放。播放字符是透過播放金鑰對產生的。 播放金鑰對是公有-私有金鑰對，用來簽署和驗證用於播放的檢視器授權符記。請參閱設定私有頻道中的建立或匯入播放金鑰，並參閱 IVS 低延遲 API 參考中的播放金鑰對端點。	✓		
播放 URL	識別觀眾用來開始播放特定頻道的地址。這個地址可以在全球範圍內使用。IVS 會自動選取 IVS 全球內容交付網路上的最佳位置，以將影片交付給每位觀眾。請參閱 IVS 低延遲串流功能入門中有關建立頻道的資訊。	✓		
私有頻道	可讓客戶使用基於播放字符的授權機制來限制對其串流的存取。請參閱設定私有頻道中的私有頻道的工作流程。	✓		
漸進式影片	依序傳輸和顯示每個影格的所有行。我們建議在廣播的所有階段使用漸進式影片。	✓	✓	
配額	您 AWS 帳戶的 IVS 服務資源或操作數上限。也就是說，這些限制以 AWS 帳戶為依據，除非另有說明。所有配額均按區域執行。請參閱《AWS 一般參考指南》中的 Amazon 互動式影片服務端點和配額。	✓	✓	✓

術語	描述	LL	RT	聊天
區域	可讓您存取實際位於特定地理區域的 AWS 服務。區域提供容錯能力、穩定性和恢復能力，也可降低延遲。透過區域，您可以建立冗餘資源，這些資源會保持可用且不受區域中斷影響。 大多數 AWS 服務請求都與特定地理區域關聯。您在某個區域中建立的資源在任何其他區域中都不存在，除非您明確使用 AWS 服務提供的複寫功能。例如，Amazon S3 支援跨區域複寫。部分服務 (例如，IAM) 沒有跨區域資源。	✓	✓	✓
解析度	描述單一影片影格中的像素數量，例如，Full HD 或 1080p 定義具有 1920x1080 像素的影格。	✓	✓	
根使用者	AWS 帳戶的擁有者。根使用者具有對 AWS 帳戶中所有 AWS 服務和資源的完整存取權。	✓	✓	✓
RTMP、RTMPS	即時訊息協定，透過網路傳輸音訊、影片和資料的業界標準。RTMPS 是 RTMP 的安全版本，透過 Transport Layer Security (TLS/SSL) 連線執行。	✓	✓	
S3 儲存貯體	儲存在 Amazon S3 中的物件集合。許多政策 (包括存取和複寫) 都是在儲存貯體層級定義的，並套用於儲存貯體中的所有物件。例如，IVS 廣播會作為多個物件儲存在 S3 儲存貯體中。	✓		
SDK	軟體開發套件，為使用 IVS 建置應用程式的開發人員提供的程式庫集合。	✓	✓	✓
自拍分割	允許替換即時串流中的背景，使用用戶端特定解決方案，該解決方案接受攝影機影像作為輸入並傳回遮罩，該遮罩為影像的每個像素提供可信度分數，指示它是在前景還是背景。請參閱 IVS 廣播 SDK：第三方攝影機濾鏡 (即時串流) 中的背景替換。		✓	

術語	描述	LL	RT	聊天
語義版本控制	Major.Minor.Patch 形式的版本格式。不影響 API 的錯誤修正會增加修補程式版本，回溯相容的 API 新增/變更會增加次要版本，回溯不相容的 API 變更會增加主要版本。	✓	✓	✓
伺服器端合成	使用 IVS 伺服器來混合階段參與者的音訊和影片，然後將此混合影片傳送至 IVS 頻道，以觸及更多觀眾或將其儲存在 S3 儲存貯體中。伺服器端合成減少了用戶端負載，提高了廣播的恢復能力，並能夠更有效地使用頻寬。 另請參閱用戶端合成。		✓	
Service Quotas	一種 AWS 服務，可協助您從一個位置管理許多 AWS 服務的配額。除了查詢配額值以外，您也可以從 Service Quotas 主控台請求增加配額。	✓	✓	✓
服務連結角色	直接連結至 AWS 服務的獨特類型的 IAM 角色。服務連結角色由 IVS 自動建立，並包含該服務代表您呼叫其他 AWS 服務 (例如存取 S3 儲存貯體) 所需的所有許可。請參閱 IVS 安全性中的使用 IVS 的服務連結角色。	✓		
階段	IVS 資源，代表即時事件參與者可以即時交換影片的虛擬空間。請參閱 IVS 即時串流功能入門中的建立階段。		✓	
階段工作階段	在第一個參與者加入階段時開始，並在最後一位參與者停止發布至階段的幾分鐘後結束。一個長期存放的階段在生命週期內可能有多個工作階段。		✓	
串流	代表從來源連續傳送至目的地的影片或音訊內容的資料。	✓	✓	

術語	描述	LL	RT	聊天
串流金鑰	在您建立頻道時 IVS 指派的識別符；它用於授權串流至頻道。將串流金鑰視為機密，因為具有它的任何人都可以串流至頻道。請參閱 IVS 低延遲串流功能入門。	✓		
串流匱乏	串流交付至 IVS 延遲或停止。當 IVS 未收到編碼裝置公告它在特定時間範圍內會傳送的預期位元量時，會發生這種情況。發生串流匱乏會導致串流匱乏事件。 從觀眾的角度來看，串流匱乏可能會導致影片延遲、緩衝或凍結。串流匱乏可能很短 (少於 5 秒) 或很長 (幾分鐘)，取決於導致串流匱乏的特定情況。請參閱疑難排解常見問答集中的什麼是串流匱乏。	✓	✓	
實況主	將影片或音訊串流傳送至 IVS 的人員或裝置。	✓	✓	
Subscriber	接收主持人的影片和/或音訊的即時事件參與者。請參閱什麼是 IVS 即時串流。		✓	
Tag	您指派給 AWS 資源的中繼資料標籤。標籤可協助您識別和整理 AWS 資源。在 IVS 文件登陸頁面上，請參閱任何 IVS API 文件中的「標記」(用於即時串流、低延遲串流或聊天)。	✓	✓	✓
第三方攝影機濾鏡	可與 IVS 廣播 SDK 整合的軟體元件，允許應用程式在將影像作為自訂影像來源提供給廣播 SDK 之前處理影像。第三方攝影機濾鏡可以處理來自攝影機的影像、套用濾鏡效果等。	✓	✓	
縮圖	從串流中取得的大小縮小的影像。依預設，縮圖每 60 秒產生一次，但可以設定更短的間隔。縮圖解析度取決於頻道類型。請參閱自動錄製到 Amazon S3 (低延遲串流) 中的錄製內容。	✓		

術語	描述	LL	RT	聊天
定時中繼資料	繫結至串流內特定時間戳記的中繼資料。它可以使用 IVS API 以程式設計方式新增，並與特定影格關聯。這可確保所有觀眾都在相對於串流的相同點上接收中繼資料。 定時中繼資料可用於觸發用戶端上的動作，例如在體育賽事期間更新團隊統計資料。請參閱在影片串流中內嵌中繼資料。	✓		
轉碼	將影片和音訊從一種格式轉換為另一種格式。傳入串流可以多個位元率和解析度轉碼為不同的格式，以支援多種播放裝置和網路狀況。	✓	✓	
轉碼複用	將擷取的串流簡單地重新封裝至 IVS，而無需重新編碼影片串流。「轉碼複用」是轉碼多工處理的簡稱，這是一個變更音訊和/或影片檔案格式同時保留部分或全部原始串流的程序。轉碼複用會轉換為不同的容器格式，而不變更檔案內容。與轉碼不同。	✓	✓	
變體串流	相同廣播的一組編碼，具有多個不同的品質等級。每個變體串流都編碼為單獨的 HLS 播放清單。可用變體串流的索引稱為多變體播放清單。 IVS 播放器從 IVS 接收多變體播放清單之後，就可以在播放期間於變體串流之間選擇，並隨著網路條件變更無縫地來回變更。	✓		
VBR	可變位元率，一種編碼器的速率控制方法，它使用在整個播放過程中根據所需的細節層次而變更的動態位元率。由於影片品質問題，我們強烈建議不要使用 VBR；請改用 CBR。	✓	✓	

術語	描述	LL	RT	聊天
檢視	正在主動下載或播放視訊的獨特檢視工作階段。檢視是並行檢視配額的基礎。 當檢視工作階段開始視訊播放時，檢視便會開始。當檢視工作階段停止視訊播放時，檢視結束。播放是觀眾人數的唯一指標；不考慮參與啟發學習法，例如音訊電平、瀏覽器分頁焦點和視訊品質。在計算檢視次數時，IVS 不會考慮個別觀眾的合法性，也不會嘗試對本地化收視人數消除重複，例如在單一機器上的多個影片播放器。請參閱 Service Quotas (低延遲串流) 中的其他配額。	✓		
觀眾	從 IVS 接收串流的人員。	✓		
WebRTC	Web 即時通訊，一個開放原始碼專案，為 Web 瀏覽器和行動應用程式提供即時通訊。它允許直接對等通訊，從而允許音訊和影片通訊在網頁內進行，而無需安裝外掛程式或下載原生應用程式。 WebRTC 背後的技術是作為開放 Web 標準實作的，並且可以作為所有主要瀏覽器中的一般 JavaScript API 或作為原生用戶端 (如 Android 和 iOS) 的程式庫提供。	✓	✓	

術語	描述	LL	RT	聊天
WHIP	WebRTC-HTTP 擷取通訊協定，這是一種 HTTP 型通訊協定，可讓 WebRTC 型內容擷取進入串流服務和/或 CDN 中。WHIP 是為了標準化 WebRTC 擷取而開發的 IETF 草案。 WHIP 可以與 OBS 等軟體相容，為桌面發布提供了一種替代方案 (IVS 廣播 SDK)。熟悉 OBS 的更資深實況主可能更喜歡它，因為它具有進階生產功能，例如轉換場景，混合音訊和添加圖形浮水印 在不可或不偏好使用 IVS 廣播 SDK 的情形下，WHIP 也是有用的。例如，在涉及硬體編碼器的設定中，IVS 廣播 SDK 可能不適用。但是，如果編碼器支援 WHIP，您仍然可以直接從編碼器發布到 IVS。 請參閱 OBS 和 WHIP 支援。		✓	
WSS	WebSocket Secure，一種用於透過加密的 TLS 連線建立 WebSocket 的協定。它用於連接至 IVS 聊天功能端點。請參閱 IVS 聊天功能入門中的步驟 4：傳送和接收第一條訊息。			✓

文件歷史記錄 (聊天功能)

聊天功能使用者指南變更

變更	描述	日期
拆分聊天功能使用者指南	此版本隨附重大文件變更。我們將聊天資訊從《IVS 低延遲串流功能使用者指南》移至新的《IVS 聊天功能使用者指南》，該指南位於 IVS 文件登陸頁面的現有 IVS 聊天功能部分。 如需其他文件變更，請參閱文件歷史記錄 (低延遲串流)。	2023 年 12 月 28 日
IVS 詞彙表	擴展了詞彙表，涵蓋 IVS 即時、低延遲和聊天術語。	2023 年 12 月 20 日

IVS 聊天功能 API 參考變更

API 變更	描述	日期
拆分聊天功能使用者指南	現在有了 IVS 聊天功能使用者指南 (在此版本中建立)，現有 IVS 聊天功能 API 參考和 IVS 聊天傳訊 API 參考的文件歷史記錄項目位於此處，繼續前進。文件歷史記錄 (低延遲串流) 中有這些聊天功能 API 參考的先前歷史記錄項目。	2023 年 12 月 28 日

版本備註 (聊天功能)

2023 年 12 月 28 日

Amazon IVS 聊天功能使用者指南

Amazon Interactive Video Service (IVS) 聊天功能是一種受管的伴隨即時影片串流的即時聊天功能。在此版本中，我們已將聊天功能資訊從《IVS 低延遲串流使用者指南》移至新的《IVS 聊天功能使用者指南》。如需文件，請造訪 Amazon IVS 文件登陸頁面。

2023 年 1 月 31 日

Amazon IVS 聊天用戶端傳訊 SDK：Android 1.1.0

平台	下載與變更
Android 版聊天用戶端傳訊 SDK 1.1.0	參考文件：https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/ • 為了支援 Kotlin Coroutines，我們在 com.amazonaws.ivs.chat.messaging.coroutines 套件中新增了新的 IVS 聊天功能傳訊 API。另請參閱新的 Kotlin Coroutines 教學課程；第 1 部分 (共 2 部分) 是聊天室。

聊天用戶端傳訊開發套件大小：Android 版

架構	壓縮大小	未壓縮大小
所有架構 (bytecode)	89 KB	92 KB

2022 年 11 月 9 日

Amazon IVS 聊天用戶端傳訊 SDK：JavaScript 1.0.2

平台	下載與變更
JavaScript 版聊天用戶端傳訊開發套件 1.0.2	參考文件：https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/ • 修正了會影響 Firefox 的問題：用戶端在使用 DisconnectUser 端點與聊天室中斷連線時，錯誤地收到通訊端錯誤。

2022 年 9 月 8 日

Amazon IVS 聊天用戶端傳訊 SDK：Android 1.0.0 和 iOS 1.0.0

平台	下載與變更
Android 版聊天用戶端傳訊開發套件 1.0.0	參考文件：https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/
iOS 版聊天用戶端傳訊開發套件 1.0.0	參考文件：https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/

聊天用戶端傳訊開發套件大小：Android 版

架構	壓縮大小	未壓縮大小
所有架構 (bytecode)	53 KB	58 KB

聊天用戶端傳訊開發套件大小：iOS 版

架構	壓縮大小	未壓縮大小
ios-arm64_x86_64-simulator (bitcode)	484 KB	2.4 MB
ios-arm64_x86_64-simulator	484 KB	2.4 MB
ios-arm64 (bitcode)	1.1 MB	3.1 MB
iOS arm64	233 KB	1.2 MB