aws

Guide de l'utilisateur chat

Amazon IVS

Amazon IVS: Guide de l'utilisateur chat

Table of Contents

En quoi consiste Chat Amazon IVS

Chat Amazon IVS est une fonctionnalité de chat en direct gérée qui accompagne les flux vidéo en direct. La documentation est accessible depuis la page de destination de la documentation Chat Amazon IVS :

- Guide de l'utilisateur du chat : ce document, ainsi que toutes les autres pages du guide de l'utilisateur répertoriées dans le volet de navigation.
- Référence d'API Chat : API du plan de contrôle (HTTPS).
- Référence de l'API de messagerie de chat : API de plan de données (WebSocket).
- Références des kits SDK pour les clients de chat : Android, iOS et JavaScript.

Mise en route avec le chat Amazon IVS

Amazon Interactive Video Service (IVS) Chat est une fonction gérée de conversation en direct qui accompagne vos flux vidéo en direct. (IVS Chat peut également être utilisé sans flux vidéo.) Vous pouvez créer des salles de chat et activer des sessions de chat entre vos utilisateurs.

Chat Amazon IVS vous permet de vous concentrer sur la création d'expériences de chat personnalisées aux côtés de vidéos en direct. Vous n'avez pas besoin de gérer l'infrastructure ni de développer et de configurer des composants de vos flux de chat. Chat Amazon IVS est évolutif, sécurisé, fiable et économique.

Chat Amazon IVS est idéal pour faciliter la messagerie entre les participants d'un flux vidéo en direct avec un début et une fin.

Le reste de ce document vous guide à travers les étapes de création de votre première application de chat à l'aide Chat Amazon IVS.

Exemples : les applications de démonstration suivantes sont disponibles (trois exemples d'applications clientes et une application de serveur backend pour la création de jetons) :

- Démonstration web Chat Amazon IVS
- Démonstration Chat Amazon IVS pour Android
- Démonstration Chat Amazon IVS pour iOS
- Backend de démonstration Chat Amazon IVS

Important : les salles de chat qui n'ont pas de nouvelles connexions ou mises à jour depuis 24 mois sont automatiquement supprimées.

Rubriques

Étape 1 : tâches de configuration initiale

Avant de poursuivre, vous devez :

1. Créer un compte AWS.
2. Configurer les utilisateurs root et administratifs.
3. Configurer les autorisations AWS IAM (AWS Identity and Access Management). Utiliser la politique spécifiée ci-dessous.

Pour connaître la marche à suivre en détail, consultez la section Mise en route avec le streaming à faible latence IVS dans le Guide de l'utilisateur Amazon IVS. Important : dans « Étape 3 : configurer les autorisations IAM », utilisez cette politique pour 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": "*"
      }
   ]
}
```

Étape 2 : créer une salle de chat

Une salle de chat Amazon IVS est associée à des informations de configuration (par exemple, la longueur maximale du message).

Les instructions disponibles dans cette section vous montrent comment utiliser la console ou l'interface de la ligne de commande AWS pour configurer des salles de conversation (y compris une configuration facultative pour la révision des messages et/ou la journalisation des messages) et créer des salles.

Instructions de la console

Ces étapes sont divisées en phases, commençant par la configuration initiale de la salle et se terminant par la création finale de la salle.

Vous pouvez également configurer une salle pour que les messages soient révisés. Par exemple, vous pouvez mettre à jour le contenu ou les métadonnées des messages, refuser des messages pour les empêcher d'être envoyés ou laisser passer le message d'origine. Ceci est couvert par la rubrique Configurer pour réviser les messages de la salle (facultatif).

Vous pouvez également, de manière facultative, configurer une salle pour que les messages soient journalisés. Par exemple, si des messages sont envoyés à une salle de chat, vous pouvez les journaliser dans un compartiment Amazon S3, Amazon CloudWatch ou Amazon Kinesis Data Firehose. Ceci est couvert par la section Configurer pour journaliser les messages de la salle (facultatif).

Configuration initiale de la salle

1. Ouvrez la console Chat Amazon IVS.

   (Vous pouvez également accéder à la console Amazon IVS via la console de gestion AWS.)
2. Choisissez une région dans le menu déroulant de la barre de navigation Sélectionner une région. Votre nouvelle salle sera créée dans cette région.
3. Dans la zone Mise en route, en haut à droite, choisissez Salle de chat Amazon IVS. La fenêtre Créer une salle apparaît.

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. Sous Installation, spécifiez éventuellement un Nom de salle. Les noms des salles ne sont pas uniques, mais ils vous permettent de distinguer les salles autrement que par leur ARN (Amazon Resource Name).
5. Sous Configuration > Configuration de la salle, acceptez soit la Configuration par défaut, ou sélectionnez Configuration personnalisée puis configurez les paramètres de Longueur maximale du message et/ou Fréquence de message maximale.
6. Si vous souhaitez réviser les messages, continuez avec Configurer pour réviser les messages de salle (facultatif) ci-dessous. Sinon, sautez cette section (c.-à-d., acceptez Gestionnaire de révision des messages > Désactivé et passez directement à Création finale de la salle.

Configurer pour revoir les messages de la salle (facultatif)

1. Sous Gestionnaire de révision des messages, sélectionnez Gérer avec AWS Lambda. La section Gestionnaire de révision des messages se développe pour afficher des options supplémentaires.
2. Configurer le Résultat de secours pour Autoriser ou Refuser le message si le gestionnaire ne renvoie pas de réponse valide, rencontre une erreur ou dépasse la période d'expiration.
3. Spécifiez votre Fonction lambda ou utilisez Créer une fonction lambda pour créer une fonction.

   La fonction lambda doit se trouver dans le même compte AWS et les mêmes Régions AWS que le salle de chat. Vous devez accorder au service du kit SDK Amazon Chat l'autorisation d'appeler votre ressource lambda. La politique basée sur les ressources sera automatiquement créée pour la fonction lambda que vous avez sélectionnée. Pour en savoir plus sur les autorisations, consultez Politique basée sur les ressources pour Chat Amazon IVS.

Configurer pour journaliser les messages (facultatif)

1. Sous Journalisation des messages, sélectionnez Journaliser automatiquement les messages de chat. La section Journalisation des messages se développe pour afficher des options supplémentaires. Vous pouvez soit ajouter une configuration de journalisation existante à cette salle, soit créer une configuration de journalisation en sélectionnant Créer une configuration de journalisation.
2. Si vous choisissez une configuration de journalisation existante, un menu déroulant apparaît et affiche toutes les configurations de journalisation que vous avez déjà créées. Sélectionnez-en un dans la liste et vos messages de chat seront automatiquement journalisés à cette destination.

3. Si vous choisissez Créer une configuration de journalisation, une fenêtre modale apparaît qui vous permet de créer et de personnaliser une configuration de journalisation.
   a. Spécifiez éventuellement un nom de configuration de journalisation. Les noms des configurations de journalisation, comme les noms des salles, ne sont pas uniques, mais ils vous permettent de distinguer les configurations de journalisation autrement que par l'ARN de la configuration de journalisation.
   b. Sous Destination, sélectionnez le groupe de journaux CloudWatch, le flux de diffusion Kinesis Firehose ou le compartiment Amazon S3 pour choisir la destination de vos journaux.
   c. En fonction de votre destination, sélectionnez l'option permettant de créer ou d'utiliser un groupe de journaux CloudWatch, un flux de diffusion Kinesis Firehose ou un compartiment Amazon S3 existant.
   d. Après avoir vérifié, choisissez Créer pour créer une configuration de journalisation avec un ARN unique. Cela associe automatiquement la nouvelle configuration de journalisation à la salle de chat.

Création finale de la salle

1. Après avoir vérifié, choisissez Create chat room (Créer une salle de chat) pour créer une salle de chat avec un ARN unique.

Instructions de la CLI

Créer une salle de conversation

La création d'une salle avec l'AWS CLI est une option avancée et nécessite le téléchargement et la configuration de la CLI sur votre machine. Pour plus de détails, consultez le Guide de l’utilisateur de l’Interface de ligne de commande AWS.

1. Exécutez la commande de chat `create-room` et passez un nom facultatif :

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

2. Ceci renvoie une nouvelle salle de chat :

```
{
   "arn": "arn:aws:ivschat:us-west-2:123456789012:room/g1H2I3j4k5L6",
   "id": "string",
```

```
   "createTime": "2021-06-07T14:26:05-07:00",
   "maximumMessageLength": 200,
   "maximumMessageRatePerSecond": 10,
   "name": "test-room",
   "tags": {},
   "updateTime": "2021-06-07T14:26:05-07:00"
}
```

3. Notez le champ `arn`. Vous en aurez besoin pour créer un jeton client et vous connecter à une salle de chat.

Définir une configuration de journalisation (facultatif)

Comme pour la création d'une salle de chat, la définition d'une configuration de journalisation avec l'AWS CLI est une option avancée et nécessite que vous téléchargiez et configuriez d'abord la CLI sur votre machine. Pour plus de détails, consultez le Guide de l’utilisateur de l’Interface de ligne de commande AWS.

1. Exécutez la commande de chat `create-logging-configuration` et transmettez un nom facultatif et une configuration de destination pointant vers un compartiment Amazon S3 par son nom. Ce compartiment Amazon S3 doit exister avant de créer la configuration de journalisation. (Pour plus d'informations sur la création d'un compartiment Amazon S3, consultez la documentation Amazon S3.)

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

2. Cela renvoie une nouvelle configuration de journalisation :

```
{
   "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"
}
```

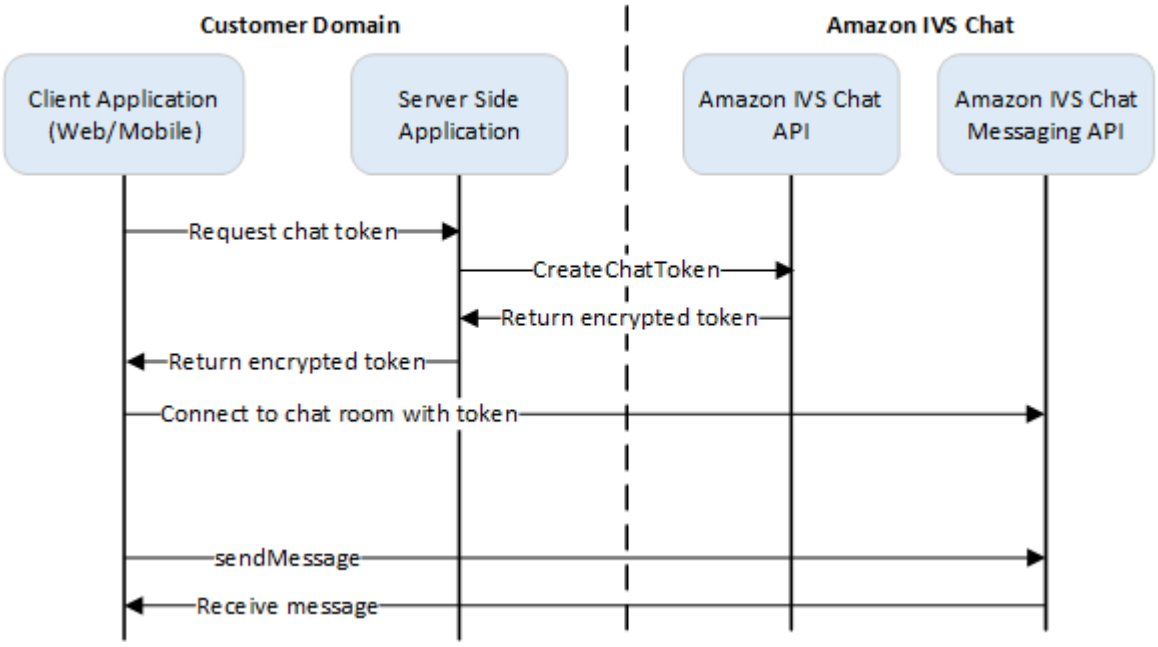
3. Notez le champ `arn`. Vous en avez besoin pour associer la configuration de journalisation à la salle de chat.
   a. Si vous créez une salle de chat, exécutez la commande `create-room` et transmettez la configuration de journalisation `arn` :

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

   b. Si vous mettez à jour une salle de chat existante, exécutez la commande `update-room` et transmettez la configuration de journalisation `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"
```

Étape 3 : créer une salle de chat



Pour qu'un participant au chat puisse se connecter à une salle et commencer à envoyer et à recevoir des messages, un jeton de chat doit être créé. Les jetons de chat sont utilisés pour authentifier et

autoriser les clients de chat. Comme indiqué ci-dessus, une application client demande un jeton à votre application côté serveur, et l'application côté serveur appelle CreateChatToken à l'aide d'un kit SDK AWS ou de requêtes signées SigV4. Étant donné que les informations d'identification AWS sont utilisées pour appeler l'API, le jeton doit être généré dans une application côté serveur sécurisée, et non dans l'application côté client.

Une application de serveur backend qui illustre la génération de jetons est disponible sur le backend de démonstration Chat Amazon IVS.

Durée de la session fait référence à la durée pendant laquelle une session établie peut rester active avant qu'elle ne soit automatiquement fermée. En d'autres termes, la durée de la session correspond à la durée pendant laquelle le client peut rester connecté à la salle de chat avant qu'un nouveau jeton ne doive être généré et qu'une nouvelle connexion ne doive être établie. Vous pouvez en option, spécifier la durée de la session lors de la création de jetons.

Chaque jeton ne peut être utilisé qu'une seule fois pour établir une connexion pour un utilisateur final. Si une connexion est fermée, un nouveau jeton doit être créé avant qu'une connexion puisse être rétablie. Le jeton lui-même est valide jusqu'à l'horodatage d'expiration du jeton inclus dans la réponse.

Lorsqu'un utilisateur final souhaite se connecter à une salle de chat, le client doit demander un jeton à l'application serveur. L'application serveur crée un jeton et le transmet au client. Des jetons doivent être créés pour les utilisateurs finaux à la demande.

Pour créer un jeton d'authentification de chat, suivez les instructions ci-dessous. Lorsque vous créez un jeton de chat, utilisez les champs de demande pour transmettre des données concernant l'utilisateur final du chat et ses capacités de messagerie. Pour plus de détails, consultez la section CreateChatToken dans la Référence de l'API IVS Chat.

Instructions du kit SDK AWS

La création d'une salle de chat avec le kit SDK AWS nécessite que vous téléchargiez et configuriez d'abord le kit SDK sur votre application. Vous trouverez ci-dessous les instructions concernant le SDK AWS utilisant JavaScript.

Important : ce code doit être exécuté côté serveur et sa sortie doit être transmise au client.

Prérequis : pour pouvoir utiliser l'exemple de code ci-dessous, vous devez charger le kit SDK AWS JavaScript dans votre application. Pour plus d'informations, consultez la section Mise en route avec le kit SDK AWS pour JavaScript.

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

Instructions de la CLI

La création d'un jeton de chat avec l'AWS CLI est une option avancée et nécessite que vous téléchargiez et configuriez d'abord le CLI sur votre machine. Pour plus de détails, consultez le Guide de l'utilisateur de l'Interface de ligne de commande AWS. Remarque : la génération de jetons avec l'AWS CLI est utile à des fins de test, mais pour une utilisation en production, nous vous recommandons de générer des jetons côté serveur avec le kit SDK AWS (voir les instructions ci-dessus).

1. Exécutez la commande `create-chat-token` avec l'identifiant de la salle et l'ID utilisateur du client. Incluez l'une des fonctions suivantes : `"SEND_MESSAGE"`, `"DELETE_MESSAGE"`, `"DISCONNECT_USER"`. (Vous pouvez également inclure la durée de la session (en minutes) et/ou les attributs personnalisés (métadonnées) de cette session de chat. Ces champs ne sont pas affichés ci-dessous.)

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

2. Ceci renvoie un jeton client :

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

3. Enregistrez ce jeton. Vous en aurez besoin pour vous connecter à la salle de chat et envoyer ou recevoir des messages. Vous devrez générer un autre jeton de chat avant la fin de votre session (comme indiqué par `sessionExpirationTime`).

Étape 4 : envoyer et recevoir votre premier message

Utilisez votre jeton chat pour vous connecter à une salle de chat et envoyer votre premier message. Un exemple de code JavaScript est fourni ci-dessous. Des kits SDK clients sont également disponibles : consultez les sections Chat SDK: Android Guide, Chat SDK: iOS Guide et Chat SDK: JavaScript Guide.

Service régional : l'exemple de code ci-dessous fait référence à votre « région de choix prise en charge ». Chat Amazon IVS propose des points de terminaison régionaux que vous pouvez utiliser pour effectuer vos demandes. Pour l'API de messagerie Chat Amazon IVS, la syntaxe générale d'un point de terminaison régional est la suivante :

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

Par exemple, le point de terminaison de la région USA Ouest (Oregon) est wss://edge.ivschat.us-west-2.amazonaws.com Pour obtenir une liste des régions prises en charge, consultez la page Amazon IVS dans la Référence générale AWS.

```
/*
1. To connect to a chat room, you need to create a Secure-WebSocket connection
using the client token you created in the previous steps. Use one of the provided
endpoints in the Chat Messaging API, depending on your AWS region.
*/
const chatClientToken = "GENERATED_CHAT_CLIENT_TOKEN_HERE";
const socket = "wss://edge.ivschat.us-west-2.amazonaws.com"; // Replace “us-west-2”
 with supported region of choice.
const connection = new WebSocket(socket, chatClientToken);
```

```
/*
2. You can send your first message by listening to user input
in the UI and sending messages to the WebSocket connection.
*/
const payload = {
  "Action": "SEND_MESSAGE",
  "RequestId": "OPTIONAL_ID_YOU_CAN_SPECIFY_TO_TRACK_THE_REQUEST",
  "Content": "text message",
  "Attributes": {
    "CustomMetadata": "test metadata"
  }
}
connection.send(JSON.stringify(payload));

/*
3. To listen to incoming chat messages from this WebSocket connection
and display them in your UI, you must add some event listeners.
*/
connection.onmessage = (event) => {
  const data = JSON.parse(event.data);
  displayMessages({
    display_name: data.Sender.Attributes.DisplayName,
    message: data.Content,
    timestamp: data.SendTime
  });
}

function displayMessages(message) {
  // Modify this function to display messages in your chat UI however you like.
  console.log(message);
}

/*
4. Delete a chat message by sending the DELETE_MESSAGE action to the WebSocket
connection. The connected user must have the "DELETE_MESSAGE" permission to
perform this action.
*/

function deleteMessage(messageId) {
  const deletePayload = {
    "Action": "DELETE_MESSAGE",
    "Reason": "Deleted by moderator",
    "Id": "${messageId}"
  }
```

```
  connection.send(deletePayload);
}
```

Félicitations, vous êtes prêt ! Vous disposez désormais d'une application de chat simple qui peut envoyer ou recevoir des messages.

Étape 5 : vérifiez vos limites Service Quotas (facultatif)

Vos salles de chat évolueront avec votre flux en direct Amazon IVS, afin de permettre à tous vos utilisateurs de s'engager dans des conversations de chat. Cependant, tous les comptes Amazon IVS ont des limites de nombre de participants au chat simultanés et de taux de remise des messages.

Assurez-vous de définir des limites adéquates et augmentez-les si nécessaire, surtout si vous prévoyez un événement de streaming à large échelle. Pour plus de détails, consultez les sections Quotas de service (streaming à faible latence), Quotas de service (Streaming en temps réel) et Quotas de service (Chat).

Journalisation du chat

La fonctionnalité de journalisation des chats vous permet d'enregistrer tous les messages d'une salle vers l'un des trois emplacements standard : un compartiment Amazon S3, Amazon CloudWatch Logs ou Amazon Kinesis Data Firehose. Par la suite, les journaux peuvent être utilisés à des fins d'analyse ou pour créer une rediffusion du chat en lien avec une session vidéo en direct.

Activer la journalisation des chats pour une salle

La journalisation des chats est une option avancée qui peut être activée en associant une configuration de journalisation à une salle. Une configuration de journalisation est une ressource qui vous permet de spécifier un type d'emplacement (compartiment Amazon S3, Amazon CloudWatch Logs ou Amazon Kinesis Data Firehose) où les messages d'une salle sont journalisés. Pour plus de détails sur la création et la gestion des configurations de journalisation, consultez les sections Mise en route avec Amazon IVS Chat et Référence de l'API Amazon IVS Chat (langue française non garantie).

Vous pouvez associer jusqu'à trois configurations de journalisation à chaque salle, soit lors de la création d'une salle (CreateRoom), soit lors de la mise à jour d'une salle existante (UpdateRoom). Vous pouvez associer plusieurs salles à la même configuration de journalisation.

Lorsqu'au moins une configuration de journalisation active est associée à une salle, chaque demande de messagerie envoyée à cette salle via l'API de messagerie Amazon IVS Chat (langue française non garantie) est automatiquement enregistrée dans le ou les emplacements spécifiés. Voici les délais de propagation moyens (entre le moment où une demande de messagerie est envoyée et le moment où elle devient disponible dans les emplacements que vous avez spécifiés) :

- Compartiment Amazon S3 : 5 minutes
- Amazon CloudWatch Logs ou Amazon Kinesis Data Firehose : 10 secondes

Contenu des messages

Format

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

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

Champs

Champ	Description
`event_timestamp`	Horodatage UTC indiquant la date à laquelle le message a été reçu par Amazon IVS Chat.
`payload`	La charge utile JSON de Message (Subscribe) [Message (S'abonner)] ou d'Event (Subscribe) [Événement (S'abonner)] que les clients recevront du service Amazon IVS Chat.
`type`	Type du message de chat. • Valeurs valides : `MESSAGE` | `EVENT`
`version`	Version du format de contenu du message.

Compartiment Amazon S3

Format

Les journaux de messages sont organisés et stockés avec le préfixe S3 et le format de fichier suivants :

```
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
```

Champs

Champ	Description
`<account_id>`	ID de compte AWS à partir duquel la salle est créée.

Champ	Description
`<hash>`	Valeur de hachage générée par le système pour garantir l'unicité.
`<region>`	Région de service AWS dans laquelle la salle a été créée.
`<resource_id>`	La partie de l'ID de la ressource de l'ARN de la salle.
`<version>`	Version du format de contenu du message.
`<year> / <month> / <day> / <hours> / <minute>`	Horodatage UTC indiquant la date à laquelle le message a été reçu par Amazon IVS Chat.

Exemple

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

Amazon CloudWatch Logs

Format

Les journaux de messages sont organisés et stockés selon le format de nom de flux de journaux suivant :

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

Champs

Champ	Description
`<resource_id>`	Partie de l'ID de la ressource de l'ARN de la salle.
`<version>`	Version du format de contenu du message.

Exemple

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

Amazon Kinesis Data Firehose

Les journaux de messages sont envoyés au flux de diffusion sous forme de données de streaming en temps réel vers des destinations telles qu'Amazon Redshift, Amazon OpenSearch Service, Splunk, et tout point de terminaison HTTP personnalisé ou appartenant à des fournisseurs de services tiers pris en charge. Pour plus d'informations, consultez Qu'est-ce qu'Amazon Kinesis Data Firehose.

Constraints

- Vous devez être propriétaire de l'emplacement de journalisation où les messages seront stockés.
- La salle, la configuration de journalisation et l'emplacement de journalisation doivent se situer dans la même région AWS.
- Seules les configurations de journalisation actives sont disponibles pour la journalisation des chats.
- Vous ne pouvez supprimer une configuration de journalisation que si elle n'est plus associée à aucune salle.

La journalisation de messages dans un emplacement qui vous appartient nécessite une autorisation avec vos informations d'identification AWS. Pour accorder à IVS Chat l'accès requis, une politique de ressources (pour un compartiment Amazon S3 ou CloudWatch Logs) ou un rôle AWS IAM lié à un service (SLR) (pour Amazon Kinesis Data Firehose) est automatiquement généré lors de la création de la configuration de journalisation. Soyez prudent quant à toute modification du rôle ou des politiques, car cela peut avoir un impact sur l'autorisation de journalisation du chat.

Surveillance des erreurs avec Amazon CloudWatch

Vous pouvez surveiller les erreurs survenant dans la journalisation du chat avec Amazon CloudWatch, et vous pouvez créer des alarmes ou des tableaux de bord pour indiquer les changements d'erreurs spécifiques ou y répondre.

Il existe plusieurs types d'erreurs. Pour plus d'informations, consultez Surveillance de Chat Amazon IVS.

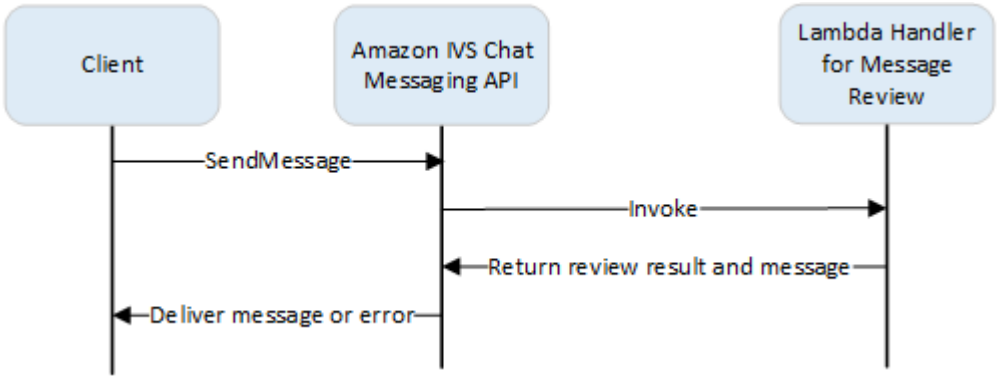
Gestionnaire de révision des messages de chat

Un gestionnaire de révision des messages vous permet d'examiner et/ou de modifier les messages avant qu'ils ne soient livrés dans une salle. Lorsqu'un gestionnaire de révision des messages est associé à une salle, il est appelé pour chaque demande SendMessage envoyée à cette salle. Le gestionnaire applique la logique métier de votre application et détermine s'il convient d'autoriser, de refuser ou de modifier un message. Amazon IVS Chat prend en charge les fonctions AWS Lambda en tant que gestionnaires.

Création d'une fonction lambda

Avant de configurer un gestionnaire de révision des messages pour une salle, vous devez créer une fonction lambda avec une politique IAM basée sur les ressources. La fonction lambda doit se trouver dans les mêmes compte AWS et Région AWS que la salle avec laquelle vous utiliserez la fonction. La politique basée sur les ressources donne à Amazon IVS Chat l'autorisation d'appeler votre fonction lambda. Pour obtenir des instructions, consultez la Politique basée sur les ressources pour Chat Amazon IVS.

Flux de travail



Syntaxe de la requête

Lorsqu'un client envoie un message, Amazon IVS Chat appelle la fonction lambda avec une charge utile JSON :

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

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

Corps de la requête

Champ	Description
`Attributes`	Attributs associés au message.
`Content`	Contenu original du message.
`MessageId`	L'ID du message. Généré par IVS Chat.
`RoomArn`	L'ARN de la salle à laquelle les messages sont envoyés.
`Sender`	Informations sur l'expéditeur. Cet objet comporte plusieurs champs : • `Attributes` : métadonnées relatives à l'expéditeur établies lors de l'authentification. Cela peut être utilisé pour fournir au client plus d'informations sur l'expéditeur, par exemple l'URL de l'avatar, les badges, la police et la couleur. • `UserId` : identifiant spécifié par l'application de la visionneuse (utilisateur final) qui a envoyé ce message. Cela peut être utilisé par l'application cliente pour faire référence à l'utilisateur dans l'API de messagerie ou dans les domaines d'application. • `Ip` : l'adresse IP du client envoyant le message.

Syntaxe de la réponse

La fonction lambda du gestionnaire doit renvoyer une réponse JSON avec la syntaxe suivante. Les réponses qui ne correspondent pas à la syntaxe ci-dessous ou qui ne satisfont pas aux contraintes de champ ne sont pas valides. Dans ce cas, le message est autorisé ou refusé en fonction de la

valeur `FallbackResult` que vous spécifiez dans votre gestionnaire de révision des messages ; consultez MessageReviewHandler dans la Référence de l'API de chat Amazon IVS.

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

Champs de réponse

Champ	Description
`Attributes`	Attributs associés au message renvoyé par la fonction lambda. Si `ReviewResult` est `DENY`, un `Reason` peut être fourni en `Attributes` ; par exemple : `"Attributes": {"Reason": "denied for moderation` Dans ce cas, le client expéditeur reçoit une erreur WebSocket 406 avec la raison indiquée dans le message d'erreur. (Consultez Erreurs WebSocket dans la Référence de l'API de messagerie Amazon IVS Chat.) • Contraintes de taille : 1 Ko maximum • Obligatoire : non
`Content`	Contenu du message renvoyé à partir de la fonction lambda. Il peut être édité ou original en fonction de la logique métier. • Contraintes de longueur : longueur minimum de 1. Longueur maximum du paramètre `MaximumMessageLength` que vous avez défini lorsque vous avez créé/mis à jour la salle. Pour plus d'informations sur l'API, consultez la Référence de l'API de chat Amazon IVS. Cela s'applique uniquement lorsque `ReviewResult` est `ALLOW`. • Obligatoire : oui

Champ	Description
`ReviewResult`	Le résultat du traitement de révision sur la façon de traiter le message. S'il est autorisé, le message est envoyé à tous les utilisateurs connectés à la salle. S'il est refusé, le message n'est envoyé à aucun utilisateur. • Valeurs valides : `ALLOW` | `DENY` • Obligatoire : oui

Exemple de code

Vous trouverez ci-dessous un exemple de gestionnaire lambda dans Go. Il modifie le contenu du message, conserve les attributs du message inchangés et autorise le message.

```
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
}
```

Associer et dissocier un gestionnaire à/d'une salle

Une fois que vous avez configuré et implémenté le gestionnaire lambda, utilisez l'API de chat Amazon IVS :

- Pour associer le gestionnaire à une salle, appelez CreateRoom ou UpdateRoom et spécifiez le gestionnaire.
- Pour dissocier le gestionnaire d'une salle, appelez UpdateRoom avec une valeur vide pour `MessageReviewHandler.Uri`.

Surveillance des erreurs avec Amazon CloudWatch

Vous pouvez surveiller les erreurs survenant lors de la révision des messages avec Amazon CloudWatch, et vous pouvez créer des alarmes ou des tableaux de bord pour indiquer ou répondre aux changements d'erreurs spécifiques. Si une erreur se produit, le message est autorisé ou refusé en fonction de la valeur `FallbackResult` que vous spécifiez lorsque vous associez le gestionnaire à une salle ; consultez MessageReviewHandler dans la Référence de l'API de chat Amazon IVS.

Il existe plusieurs types d'erreurs :

- `InvocationErrors` se produit lorsque Amazon IVS Chat ne peut pas appeler un gestionnaire.
- `ResponseValidationErrors` se produit lorsqu'un gestionnaire renvoie une réponse non valide.
- Les `Errors` AWS Lambda se produisent lorsqu'un gestionnaire lambda renvoie une erreur de fonction lorsqu'il a été appelé.

Pour plus d'informations sur les erreurs d'invocation et les erreurs de validation de réponse (émises par Chat Amazon IVS), consultez Surveillance de Chat Amazon IVS. Pour plus d'informations sur les erreurs AWS Lambda, consultez Utilisation des métriques Lambda.

Surveillance de Chat Amazon IVS

Vous pouvez surveiller les ressources de Chat Amazon Interactive Video Service (IVS) à l'aide d'Amazon CloudWatch. CloudWatch collecte et traite les données brutes du Chat Amazon IVS en métriques lisibles et disponibles pratiquement en temps réel. Ces statistiques sont conservées pendant 15 mois ; par conséquent, vous pouvez acquérir un point de vue historique sur la façon dont votre service ou application Web s'exécute. Vous pouvez définir des alarmes pour certains seuils et envoyer des notifications ou prendre des actions spécifiques lorsque ces seuils sont atteints. Pour plus d'informations, consultez le Guide de l'utilisateur CloudWatch.

Accès aux métriques CloudWatch

Amazon CloudWatch collecte et transforme les données brutes du Chat Amazon IVS en métriques lisibles et disponibles pratiquement en temps réel. Ces statistiques sont conservées pendant 15 mois ; par conséquent, vous pouvez acquérir un point de vue historique sur la façon dont votre service ou application Web s'exécute. Vous pouvez définir des alarmes pour certains seuils et envoyer des notifications ou prendre des actions spécifiques lorsque ces seuils sont atteints. Pour plus d'informations, consultez le Guide de l'utilisateur CloudWatch.

Notez que les mesures CloudWatch sont cumulées au fil du temps. La résolution diminue à mesure que les métriques vieillissent. Voici le schéma :

- Les métriques de 60 secondes sont disponibles pendant 15 jours.
- Les métriques de 5 minutes sont disponibles pendant 63 jours.
- Les métriques d'une heure sont disponibles pendant 455 jours (15 mois).

Pour obtenir des informations à jour sur la conservation des données, recherchez « période de conservation » dans les FAQ Amazon CloudWatch.

Instructions pour la console CloudWatch

1. Ouvrez la console CloudWatch à l'adresse https://console.aws.amazon.com/cloudwatch/.
2. Dans le panneau latéral de navigation, développez le menu déroulant Metrics (Métriques), puis sélectionnez All metrics (Toutes les métriques).
3. Sous l'onglet Parcourir, à l'aide de la liste déroulante sans étiquette à gauche, sélectionnez votre région « d'accueil » dans laquelle votre ou vos canaux ont été créés. Pour en savoir plus sur les

régions, consultez Solution mondiale, contrôle régional. Pour obtenir une liste des régions prises en charge, consultez la page Amazon IVS dans les Références générales AWS.

4. Au bas de l'onglet Parcourir, sélectionnez l'espace de noms IVSChat.
5. Effectuez l'une des actions suivantes :
   a. Dans la barre de recherche, entrez votre ID de ressource (partie de l'ARN, `arn:::ivschat:room/<resource id>`).

      Sélectionnez ensuite IVSChat.

   b. Si IVSChat apparaît comme un service sélectionnable sous AWS Namespaces (Espaces de noms AWS), sélectionnez-le. Il sera répertorié si vous utilisez Amazon IVSChat et qu'il envoie des métriques à Amazon CloudWatch. (Si IVSChat n'est pas répertorié, vous ne disposez pas de métriques Amazon IVSChat.)

      Choisissez ensuite un groupe de dimensions comme vous le souhaitez ; les dimensions disponibles sont répertoriées dans les Métriques CloudWatch ci-dessous.
6. Choisissez des métriques pour ajouter au graphique. Les métriques disponibles sont répertoriées dans les Métriques CloudWatch ci-dessous.

Vous pouvez également accéder au graphique CloudWatch de votre session de chat à partir de la page de détails de ladite session, en sélectionnant le bouton View in CloudWatch (Afficher dans CloudWatch).

Instructions de la CLI

Vous pouvez également accéder aux métriques à l'aide de l'AWS CLI. Pour cela, vous devez d'abord télécharger et configurer la CLI sur votre machine. Pour plus de détails, consultez le Guide de l'utilisateur de l'Interface de ligne de commande AWS.

Ensuite, pour accéder aux métriques de chat à faible latence Amazon IVS à l'aide de l'AWS CLI :

- À partir d'une invite de commande, exécutez :

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

Pour de plus amples informations, consultez Utilisation des métriques Amazon CloudWatch dans le Guide de l'utilisateur Amazon CloudWatch.

Métriques CloudWatch : chat IVS

Amazon IVS Chat fournit les métriques suivantes dans l'espace de noms AWS/IVSChat.

Métrique	Dimension	Description
`ConcurrentChatConnections`	Aucun	Nombre total de connexions simultanées dans une salle de chat (maximum indiqué par minute). Ceci est utile pour comprendre quand les clients approchent leur limite pour les connexions de chat simultanées dans une région. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum
`Deliveries`	Action	Nombre de livraisons de demandes de messagerie effectuées à partir d'un type d'action spécifique pour des connexions de chat dans toutes vos salles d'une même région. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum
`InvocationErrors`	`Uri`	Nombre d'erreurs d'invocation d'un gestionnaire de révision des messages spécifique dans toutes vos salles d'une région. Une erreur d'invocation se produit lorsque le gestionnaire de révision des messages ne peut pas être appelé. Des erreurs d'invocation se produisent lorsque Amazon IVS Chat ne peut pas appeler un gestionnaire. Cela peut se produire si le gestionnaire associé à une salle n'existe plus

Métrique	Dimension	Description
		ou si sa politique de ressources n'autorise pas le service à l'appeler. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum
`LogDestinationAccessDeniedError`	`LoggingConfiguration`	Nombre d'erreurs de refus d'accès d'une destination de journal dans toutes vos salles d'une région. Ces erreurs se produisent lorsque Amazon IVS Chat ne peut pas accéder à la ressource de destination que vous avez spécifiée dans la configuration de journalisation. Cela peut se produire si la politique de ressources de destination n'autorise pas le service à enregistrer. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum

Métrique	Dimension	Description
`LogDestinationErrors`	`LoggingConfiguration`	Nombre de toutes les erreurs d'une destination de journal dans toutes vos salles d'une région. Il s'agit d'une métrique agrégée qui inclut tous les types d'erreurs qui se produisent lorsqu'Amazon IVS Chat ne parvient pas à envoyer les journaux à la ressource de destination que vous avez spécifiée dans la configuration de journalisation. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum
`LogDestinationResourceNotFoundErrors`	`LoggingConfiguration`	Nombre d'erreurs de type resource-not-found (ressources non trouvées) d'une destination de journal dans toutes vos salles d'une région. Ces erreurs se produisent lorsqu'Amazon IVS Chat ne peut pas envoyer de journaux à une ressource de destination que vous avez spécifiée dans une configuration de journalisation, car cette ressource n'existe pas. Cela peut se produire si la ressource de destination associée à une configuration de journalisation n'existe plus. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum

Métrique	Dimension	Description
`Messaging Deliveries`	Aucun	Nombre de livraisons de demandes de messagerie pour des connexions de chat dans toutes vos salles d'une région. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum
`Messaging Requests`	Aucun	Nombre de demandes de messagerie effectuée s dans toutes vos salles d'une région. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum
`Requests`	Action	Nombre de demandes faites d'un type d'action spécifique dans toutes vos salles d'une région. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum

Métrique	Dimension	Description
`ResponseValidationErrors`	`Uri`	Nombre d'erreurs de validation de réponse d'un gestionnaire de révision des messages spécifique dans toutes vos salles d'une région. Une erreur de validation de réponse se produit lorsque la réponse du gestionnaire de révision des messages n'est pas valide. Cela peut signifier que la réponse n'a pas pu être analysée ou échoue aux vérifications de validation ; par exemple, un résultat de révision non valide ou des valeurs de réponse trop longues. Unité : nombre Statistiques valides : somme, moyenne, maximum, minimum

Kit SDK de messagerie client Amazon IVS Chat

Le kit SDK de messagerie client Amazon Interactive Video Services (IVS) Chat est destiné aux développeurs qui créent des applications avec Amazon IVS. Ce kit SDK est conçu pour tirer parti de l'architecture Amazon IVS et fera l'objet de mises à jour, en plus d'Amazon IVS Chat. En tant que kit SDK natif, il est conçu pour minimiser l'impact sur les performances de votre application et sur les appareils avec lesquels vos utilisateurs accèdent à votre application.

Exigences de la plateforme

Navigateurs de bureau

Navigateur	Versions prises en charge
Chrome	Deux versions principales (la version actuelle et la version la plus récente)
Edge	Deux versions principales (la version actuelle et la version la plus récente)
Firefox	Deux versions principales (la version actuelle et la version la plus récente)
Opera	Deux versions principales (la version actuelle et la version la plus récente)
Safari	Deux versions principales (la version actuelle et la version la plus récente)

Navigateurs mobiles

Navigateur	Versions prises en charge
Chrome pour Android	Deux versions principales (la version actuelle et la version la plus récente)
Firefox pour Android	Deux versions principales (la version actuelle et la version la plus récente)
Opera pour Android	Deux versions principales (la version actuelle et la version la plus récente)

Navigateur	Versions prises en charge
WebView pour Android	Deux versions principales (la version actuelle et la version la plus récente)
Samsung Internet	Deux versions principales (la version actuelle et la version la plus récente)
Safari pour iOS	Deux versions principales (la version actuelle et la version la plus récente)

Plateformes natives

Plateforme	Versions prises en charge
Android	5.0 et versions ultérieures
iOS	13.0 et versions ultérieures

Support

Si vous rencontrez une erreur ou un autre problème avec votre salle de chat, déterminez l'identifiant unique de salle via l'API IVS Chat (voir ListRooms).

Partagez cet identifiant de salle de chat avec l'équipe AWS Support. Il lui permettra d’obtenir des informations pour aider à résoudre votre problème.

Remarque : veuillez consulter la rubrique Notes de mise à jour Chat Amazon IVS pour connaître les versions disponibles et les problèmes résolus. Le cas échéant, avant de contacter le support technique, mettez à jour la version du kit SDK et vérifiez si cela résout votre problème.

Gestion des versions

Les kits SDK de messagerie client Amazon IVS Chat sont basés sur la gestion sémantique de version.

Pour ce sujet, supposons que :

- la dernière version est la version 4.1.3 ;

- la dernière version de la version majeure précédente est la version 3.2.4 ;
- la dernière version de la version 1.x est la version 1.5.6.

De nouvelles fonctions rétrocompatibles sont ajoutées en tant que versions mineures de la dernière version. Dans ce cas, la prochaine série de nouvelles fonctions sera ajoutée dans la version 4.2.0.

Des corrections de bogues mineurs rétrocompatibles sont ajoutées en tant que versions de correctifs de la dernière version. Ici, la prochaine série de corrections de bogues mineurs sera ajoutée en tant que version 4.1.4.

Les corrections de bogues majeurs rétrocompatibles sont traitées différemment. Elles sont ajoutées à plusieurs versions :

- Version de correctifs de la dernière version. Ici, il s’agit de la version 4.1.4.
- Version de correctifs de la version mineure précédente. Ici, il s’agit de la version 3.2.5.
- Version de correctifs de la dernière version 1.x. Ici, il s’agit de la version 1.5.7.

Les principales corrections de bogues sont définies par l’équipe produit d’Amazon IVS. Des exemples typiques sont les mises à jour de sécurité critiques et d’autres correctifs nécessaires pour les clients.

Remarque : dans les exemples ci-dessus, les versions publiées s’incrémentent sans ignorer de numéros (par exemple, de 4.1.3 à 4.1.4). En réalité, un ou plusieurs numéros de correctifs peuvent rester internes et ne pas être publiés, de sorte que la version publiée peut s’incrémenter de 4.1.3 à 4.1.6, par exemple.

En outre, la version 1.x sera prise en charge jusqu'à la fin de 2023 ou à la sortie de la version 3.x, selon la situation qui survient en dernier.

API Amazon IVS Chat

Côté serveur (non géré par les kits SDK), il existe deux API, chacune ayant ses propres responsabilités :

- Plan de données : l'API de messagerie IVS Chat est une API WebSockets conçue pour être utilisée par des applications frontend (iOS, Android, macOS, etc.) qui sont pilotées par un schéma d'authentification basé sur des jetons. À l'aide d'un jeton de chat généré précédemment, vous vous connectez à des salles de chat déjà existantes avec cette API.

Les kits SDK de messagerie client Amazon IVS Chat sont uniquement concernés par le plan de données. Les kits SDK supposent que vous générez déjà des jetons de chat via votre backend. La récupération de ces jetons est supposée être gérée par votre application frontend, et non par les kits SDK.

- Plan de contrôle : l'API du plan de contrôle IVS Chat fournit une interface pour que vos applications backend puissent gérer et créer des salles de chat ainsi que les utilisateurs qui les rejoignent. Considérez-la comme le panneau d'administration de l'expérience de chat de votre application, géré par votre backend. Certains points de terminaison du plan de contrôle sont responsables de la création du jeton de chat nécessaire au plan de données pour s'authentifier auprès d'une salle de chat.

  Important : les kits SDK de messagerie client IVS Chat n'appellent aucun point de terminaison du plan de contrôle. Votre backend doit être configuré pour créer des jetons de chat pour vous. Votre application frontend doit communiquer avec votre backend pour récupérer ce jeton de chat.

Kit SDK de messagerie client Chat Amazon IVS : guide Android

Le kit SDK de messagerie client Amazon Interactive Video (IVS) Chat pour Android fournit des interfaces qui vous permettent d'intégrer facilement notre API de messagerie IVS Chat sur les plateformes utilisant Android.

Le package `com.amazonaws:ivs-chat-messaging` implémente l'interface décrite dans ce document.

Dernière version du SDK de messagerie client IVS Chat pour Android : 1.1.0 (notes de mise à jour)

Documentation de référence : pour plus d'informations sur les méthodes les plus importantes disponibles dans le SDK de messagerie client Chat Amazon IVS pour Android, veuillez consulter la documentation de référence à l'adresse : https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/

Exemple de code : veuillez consulter le référentiel d'exemples Android sur GitHub : https://github.com/aws-samples/amazon-ivs-chat-for-android-demo

Exigences de la plateforme : Android 5.0 (API de niveau 21) ou une version ultérieure est nécessaire pour le développement.

Démarrage

Avant de commencer, vous devez être familiarisé avec la Mise en route avec Chat Amazon IVS.

Ajouter le package

Ajoutez `com.amazonaws:ivs-chat-messaging` à vos dépendances `build.gradle` :

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

Ajouter des règles ProGuard

Ajoutez les entrées suivantes à votre fichier de règles R8/ProGuard (`proguard-rules.pro`) :

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

Configuration de votre backend

Cette intégration nécessite des points de terminaison sur votre serveur qui communiquent avec l'API Amazon IVS. Utilisez les bibliothèques AWS officielles pour accéder à l'API Amazon IVS depuis votre serveur. Elles sont accessibles dans plusieurs langues depuis les packages publics, par exemple, node.js et Java.

Ensuite, créez un point de terminaison de serveur qui communique avec l'API Chat Amazon IVS et crée un jeton.

Configurer une connexion au serveur

Créez une méthode qui considère `ChatTokenCallback` comme un paramètre et récupère un jeton de chat depuis votre backend. Transmettez ce jeton à la méthode `onSuccess` du rappel. En cas d'erreur, transmettez l'exception à la méthode `onError` du rappel. Ceci est nécessaire pour instancier la principale entité `ChatRoom` lors de l'étape suivante.

Vous trouverez ci-dessous un exemple de code qui implémente ce qui précède à l'aide d'un appel `Retrofit`.

```
// ...
```

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

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

Utilisation de l'SDK

Initialiser une instance de salle de chat

Créez une instance de la classe `ChatRoom`. Cela nécessite de transmettre `regionOrUrl`, qui correspond généralement à la région AWS dans laquelle votre salle de chat est hébergée, et `tokenProvider` qui est la méthode de récupération de jetons créée à l'étape précédente.

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

Ensuite, créez un objet écouteur qui implémentera des gestionnaires pour les événements liés au chat et attribuez-le à la propriété `room.listener` :

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

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

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

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

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

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

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

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

// ...
```

La dernière étape de l'initialisation de base consiste à se connecter à la salle spécifique en établissant une connexion WebSocket. Pour ce faire, appelez la méthode `connect()` au sein de l'instance de la salle. Nous vous recommandons de procéder selon la méthode du cycle de vie `onResume()` pour vous assurer qu'elle conserve une connexion si votre application reprend en arrière-plan.

```
room.connect()
```

Le kit SDK tentera d'établir une connexion à une salle de chat codée dans le jeton de chat reçu de votre serveur. En cas d'échec, il tentera de se reconnecter le nombre de fois spécifié dans l'instance de la salle.

Effectuer des actions dans une salle de chat

La classe `ChatRoom` contient des actions permettant d'envoyer et de supprimer des messages ainsi que de déconnecter d'autres utilisateurs. Ces actions acceptent un paramètre de rappel facultatif qui vous permet de recevoir des notifications de confirmation ou de rejet de demande.

Envoi d'un message

Pour cette demande, la fonctionnalité `SEND_MESSAGE` doit être encodée dans votre jeton de chat.

Pour déclencher une demande send-message :

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

Pour obtenir une confirmation/un rejet de la demande, fournissez un rappel comme second paramètre :

```
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.
   }
})
```

Supprimer un message

Pour cette demande, la fonctionnalité DELETE_MESSAGE doit être encodée dans votre jeton de chat.

Pour déclencher une demande delete-message :

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

Pour obtenir une confirmation/un rejet de la demande, fournissez un rappel comme second paramètre :

```
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.
   }
})
```

Déconnecter un autre utilisateur

Pour cette demande, la fonctionnalité `DISCONNECT_USER` doit être encodée dans votre jeton de chat.

Pour déconnecter un autre utilisateur à des fins de modération :

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

Pour obtenir la confirmation/le rejet de la demande, fournissez un rappel comme second paramètre :

```
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.
   }
})
```

Déconnexion d'une salle de chat

Pour fermer votre connexion à la salle de chat, appelez la méthode `disconnect()` sur l'instance de la salle :

```
room.disconnect()
```

Étant donné que la connexion WebSocket cesse de fonctionner peu de temps après que l'application passe en arrière-plan, nous vous recommandons de vous connecter/déconnecter manuellement lors de la transition depuis/vers un état d'arrière-plan. Pour ce faire, associez l'appel `room.connect()`

de la méthode du cycle de vie `onResume()`, sur `Activity` ou `Fragment` Android, avec un appel `room.disconnect()` de la méthode du cycle de vie `onPause()`.

SDK de messagerie client Chat Amazon IVS : didacticiel Android, partie 1 : salles de chat

Il s'agit de la première partie d'un didacticiel en deux volets. Vous apprendrez les bases de l'utilisation du SDK de messagerie Chat Amazon IVS en créant une application Android entièrement fonctionnelle à l'aide du langage de programmation Kotlin. Nous appelons l'application Chatterbox.

Avant de commencer le module, prenez quelques minutes pour vous familiariser avec les prérequis, les concepts clés des jetons de discussion et le serveur principal nécessaire à la création de salles de chat.

Ces didacticiels sont conçus pour les développeurs Android expérimentés qui découvrent le SDK de messagerie d'IVS Chat. Vous devrez être à l'aise avec le langage de programmation Kotlin et la création d'interfaces utilisateur sur la plateforme Android.

Cette première partie du didacticiel est divisée en plusieurs sections :

1. the section called “Configurer un serveur d'authentification/d'autorisation local”
2. the section called “Créer un projet Chatterbox”
3. the section called “Se connecter à une salle de chat et observer les mises à jour de la connexion”
4. the section called “Créer un fournisseur de jetons”
5. the section called “Étapes suivantes”

Pour une documentation complète sur le SDK, commencez par le SDK de messagerie client Chat Amazon IVS (ici dans le Guide de l'utilisateur Chat Amazon IVS) et la Messagerie client de chat : référence du SDK pour Android (sur GitHub).

Prérequis

- Familiarisez-vous avec Kotlin et la création d'applications sur la plateforme Android. Si vous n'êtes pas familiarisé avec la création d'applications pour Android, découvrez les bases dans le guide Créer votre première application pour les développeurs Android.
- Lisez attentivement et découvrez Mise en route avec IVS Chat.

- Créez un utilisateur AWS IAM avec les fonctionnalités `CreateChatToken` et `CreateRoom` définies dans une politique IAM existante. (Consultez Mise en route avec IVS Chat.)
- Assurez-vous que les clés secrètes et d'accès de cet utilisateur sont stockées dans un fichier d'informations d'identification AWS. Pour obtenir des instructions, consultez le Guide de l'utilisateur de l'interface de ligne de commande AWS (en particulier les paramètres de configuration et de fichier d'informations d'identification).
- Créez une salle de chat et enregistrez son ARN. Consultez Mise en route avec IVS Chat. (Si vous n'enregistrez pas l'ARN, vous pourrez le consulter ultérieurement à l'aide de la console ou de l'API Chat.)

Configurer un serveur d'authentification/d'autorisation local

Votre serveur backend est chargé à la fois de créer des salles de chat et de générer les jetons de chat nécessaires au SDK de chat IVS pour Android pour authentifier et autoriser vos clients à accéder à vos salles de chat.

Consultez la section Créer un jeton de chat dans Mise en route avec Chat Amazon IVS. Comme le montre l'organigramme, votre code côté serveur est chargée de créer un jeton de chat. Cela signifie que votre application doit fournir ses propres moyens de générer un jeton de chat en demandant un jeton à votre application côté serveur.

Nous utilisons l'infrastructure Ktor pour créer un serveur local en direct qui gère la création de jetons de chat à l'aide de votre environnement AWS local.

À ce stade, nous nous attendons à ce que vos informations d'identification AWS soient correctement configurées. Pour savoir comment procéder, consultez Configuration des informations d'identification et de la région AWS pour le développement.

Créez un nouveau répertoire et appelez-le `chatterbox` et, à l'intérieur, un autre, appelé `auth-server`.

La structure de notre serveur sera la suivante :

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

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

Remarque : vous pouvez directement copier/coller le code ici dans les fichiers référencés.

Ensuite, nous ajoutons toutes les dépendances et tous les plugins nécessaires au fonctionnement de notre serveur d'authentification :

Script Kotlin :

```
// ./auth-server/build.gradle.kts

plugins {
   application
   kotlin("jvm")
   kotlin("plugin.serialization").version("1.7.10")
}

application {
   mainClass.set("io.ktor.server.netty.EngineMain")
}

dependencies {
   implementation("software.amazon.awssdk:ivschat:2.18.1")
   implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.7.20")

   implementation("io.ktor:ktor-server-core:2.1.3")
   implementation("io.ktor:ktor-server-netty:2.1.3")
   implementation("io.ktor:ktor-server-content-negotiation:2.1.3")
   implementation("io.ktor:ktor-serialization-kotlinx-json:2.1.3")

   implementation("ch.qos.logback:logback-classic:1.4.4")
}
```

Nous devons maintenant configurer la fonctionnalité de journalisation pour le serveur d'authentification. (Pour plus d'informations, veuillez consulter Configure logger.)

XML :

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

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

Le serveur Ktor nécessite des paramètres de configuration, qu'il charge automatiquement à partir du fichier `application.*` du répertoire `resources`. Nous les ajoutons donc également. (Pour plus d'informations, consultez Configuration in a file.)

HOCON :

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

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

Enfin, implémentons notre serveur :

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()
            )
         )
      }
   }
```

```
}
```

Créer un projet Chatterbox

Pour créer un projet Android, installez et ouvrez Android Studio.

Suivez les étapes répertoriées dans le guide officiel Android Créer un projet.

- Dans Choisir le type de projet, choisissez le modèle de projet Activité vide pour notre application Chatterbox.
- Dans Configurer votre projet, choisissez les valeurs suivantes pour les champs de configuration :
  - Nom : My App
  - Nom de package : com.chatterbox.myapp
  - Emplacement d'enregistrement : pointez sur le répertoire `chatterbox` créé à l'étape précédente
  - Langage : Kotlin
  - Niveau d'API minimum : API 21 : Android 5.0 (Lollipop)

Après avoir correctement spécifié tous les paramètres de configuration, la structure de nos fichiers dans le dossier `chatterbox` doit ressembler à ce qui suit :

```
- 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
```

Maintenant que nous avons un projet Android fonctionnel, nous pouvons ajouter com.amazonaws:ivs-chat-messaging à nos dépendances `build.gradle`. (Pour plus d'informations sur la boîte à outils de génération Gradle, voir Configurer votre build.)

Remarque : en haut de chaque extrait de code, il y a un chemin vers le fichier dans lequel vous devez apporter des modifications à votre projet. Le chemin est relatif par rapport à la racine du projet.

Dans le code ci-dessous, remplacez *<version>* par le numéro de version actuel du SDK de chat pour Android (par exemple, 1.0.0).

Kotlin :

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

plugins {
// ...
}

android {
// ...
}

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

Une fois la nouvelle dépendance ajoutée, exécutez Synchroniser le projet avec les fichiers Gradle dans Android Studio pour synchroniser le projet avec la nouvelle dépendance. (Pour de plus amples informations, veuillez consulter la section Ajouter des dépendances de build.)

Pour exécuter facilement notre serveur d'authentification (créé dans la section précédente) à partir de la racine du projet, nous l'incluons en tant que nouveau module dans `settings.gradle`. (Pour plus d'informations, voir Structurer et construire un composant de logiciel avec Gradle.)

Script Kotlin :

```
// ./settings.gradle

// ...

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

À présent, comme `auth-server` est inclus dans le projet Android, vous pouvez exécuter le serveur d'authentification à l'aide de la commande suivante depuis la racine du projet :

Shell :

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

Se connecter à une salle de chat et observer les mises à jour de la connexion

Pour ouvrir une connexion à une salle de chat, nous utilisons le rappel du cycle de vie de l'activité onCreate(), qui se déclenche lorsque l'activité est créée pour la première fois. Le constructeur ChatRoom nous oblige à fournir `region` et `tokenProvider` pour lancer une connexion à une salle.

Remarque : la fonction `fetchChatToken` présentée dans l'extrait ci-dessous sera implémentée dans la section suivante.

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)
   }

// ...
}
```

L'affichage et la réaction aux modifications de la connexion à une salle de chat est un élément essentiel pour la création d'une application de chat comme `chatterbox`. Avant de pouvoir commencer à interagir avec la salle, nous devons nous abonner aux événements relatifs à l'état de connexion de la salle de chat pour obtenir des mises à jour.

ChatRoom attend de nous que nous joignions une implémentation d'interface ChatRoomListener pour signaler les événements du cycle de vie. Pour l'instant, les fonctions d'écouteur enregistreront uniquement les messages de confirmation, lorsqu'ils sont invoqués :

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")
        }
    }
}
```

Maintenant que nous avons implémenté `ChatRoomListener`, nous l'attachons à notre instance de salle :

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 {
// ...
}
```

Ensuite, nous devons fournir la possibilité de lire l'état de la connexion à la salle. Nous le conserverons dans la propriété `MainActivity.kt` et l'initialiserons à l'état DÉCONNECTÉ par défaut pour les salles (voir `ChatRoom state` dans la référence du SDK IVS Chat pour Android). Pour pouvoir maintenir l'état local à jour, nous devons implémenter une fonction de mise à jour de l'état ; appelons-le `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")
         }
      }
   }
```

```
}
```

Ensuite, nous intégrons notre fonction de mise à jour de l'état à la propriété Chatroom.Listener :

Kotlin :

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

package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
// ...

   private val roomListener = object : ChatRoomListener {
      override fun onConnecting(room: ChatRoom) {
         Log.d(TAG, "onConnecting")
         runOnUiThread {
            updateConnectionState(ConnectionState.LOADING)
         }
      }

      override fun onConnected(room: ChatRoom) {
         Log.d(TAG, "onConnected")
         runOnUiThread {
            updateConnectionState(ConnectionState.CONNECTED)
         }
      }

      override fun onDisconnected(room: ChatRoom, reason: DisconnectReason) {
         Log.d(TAG, "[${Thread.currentThread().name}] onDisconnected")
         runOnUiThread {
            updateConnectionState(ConnectionState.DISCONNECTED)
         }
      }
   }
}
```

Maintenant que nous sommes en mesure d'enregistrer, d'écouter et de réagir aux mises à jour d'état de ChatRoom, il est temps d'initialiser une connexion :

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

Créer un fournisseur de jetons

Il est temps de créer une fonction chargée de créer et de gérer des jetons de chat dans notre application. Dans cet exemple, nous utilisons le client HTTP Retrofit pour Android.

Avant de pouvoir envoyer du trafic réseau, nous devons configurer une configuration de sécurité réseau pour Android. Pour plus d'informations, consultez Network security configuration.) Nous commençons par ajouter des autorisations réseau au fichier App Manifest. Notez la balise `user-permission` et l'attribut `networkSecurityConfig` ajoutés, qui indiqueront notre nouvelle configuration de sécurité réseau. Dans le code ci-dessous, remplacez *<version>* par le numéro de version actuel du SDK de chat pour Android (par exemple, 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")
}
```

Déclarez `10.0.2.2` et les domaines `localhost` comme fiables, pour commencer à échanger des messages avec notre backend :

XML :

```
// ./app/src/main/res/xml/network_security_config.xml

<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">10.0.2.2</domain>
        <domain includeSubdomains="true">localhost</domain>
    </domain-config>
</network-security-config>
```

Ensuite, nous devons ajouter une nouvelle dépendance, ainsi que l'ajout d'un convertisseur Gson pour analyser les réponses HTTP. Dans le code ci-dessous, remplacez *<version>* par le numéro de version actuel du SDK de chat pour Android (par exemple, 1.0.0).

Script Kotlin :

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

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

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

Pour récupérer un jeton de chat, nous devons effectuer une requête HTTP POST depuis notre application `chatterbox`. Nous définissons la demande dans une interface que Retrofit doit implémenter. (Voir la documentation de Retrofit. Familiarisez-vous également avec la spécification du point de terminaison CreateChatToken.)

Kotlin :

```
// ./app/src/main/java/com/chatterbox/myapp/network/ApiService.kt

package com.chatterbox.myapp.network
// ...


import androidx.annotation.Keep
import com.amazonaws.ivs.chat.messaging.ChatToken
import retrofit2.Call
```

```
import retrofit2.http.Body
import retrofit2.http.POST

data class CreateTokenParams(var userId: String, var roomIdentifier: String)

interface ApiService {
   @POST("create_chat_token")
   fun createChatToken(@Body params: CreateTokenParams): Call<ChatToken>
}
```

Maintenant que le réseau est configuré, il est temps d'ajouter une fonction chargée de créer et de gérer notre jeton de chat. Nous l'ajoutons à `MainActivity.kt`, qui a été automatiquement créé lors de la génération du projet :

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)
         }
      })
   }
}
```

Étapes suivantes

Maintenant que vous avez établi une connexion à la salle de chat, passez à la seconde partie de ce didacticiel Android, Messages et événements.

Kit SDK de messagerie client Chat Amazon IVS : didacticiel Android, partie 2 : messages et événements

Cette seconde et dernière partie du didacticiel est divisée en plusieurs sections :

1. the section called “Création d'une interface utilisateur pour l'envoi de messages”
   a. the section called “Mise en page principale de l'interface utilisateur”
   b. the section called “cellule de texte abstraite de l'interface utilisateur pour afficher le texte de manière cohérente”
   c. the section called “message à gauche de l'interface utilisateur de chat”
   d. the section called “message à droite de l'interface utilisateur de chat”
   e. the section called “valeurs de couleur supplémentaires de l'interface utilisateur”
2. the section called “appliquer la liaison d'affichage”
3. the section called “Gérer les demandes de messages de chat”
4. the section called “Étapes finales”

Pour une documentation complète sur le SDK, commencez par le SDK de messagerie client Chat Amazon IVS (ici dans le Guide de l'utilisateur Chat Amazon IVS) et la Messagerie client de chat : référence du SDK pour Android (sur GitHub).

Prérequis

Assurez-vous d'avoir terminé la première partie de ce didacticiel relative aux Salles de chat.

Création d'une interface utilisateur pour l'envoi de messages

Maintenant que nous avons initialisé avec succès la connexion à la salle de chat, il est temps d'envoyer notre premier message. Pour cette fonctionnalité, une interface utilisateur est nécessaire. Nous ajouterons :

- un bouton `connect/disconnect`
- une saisie de message avec un bouton `send`
- une liste de messages dynamiques. Pour la créer, nous utilisons Android Jetpack RecyclerView.

Mise en page principale de l'interface utilisateur

Consultez les mises en page Android Jetpack dans la documentation pour les développeurs Android.

XML :

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

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout xmlns:android="http://
schemas.android.com/apk/res/android"
                                                     xmlns:app="http://
schemas.android.com/apk/res-auto"
                                                     xmlns:tools="http://
schemas.android.com/tools"

 android:layout_width="match_parent"

 android:layout_height="match_parent">

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
                  xmlns:app="http://schemas.android.com/apk/res-auto"
                  android:id="@+id/connect_view"
                  android:layout_width="match_parent"
                  android:layout_height="match_parent"
                  android:gravity="center"
                  android:orientation="vertical">

        <androidx.cardview.widget.CardView
                android:id="@+id/connect_button"
                android:layout_width="match_parent"
                android:layout_height="48dp"
                android:layout_gravity=""
                android:layout_marginStart="16dp"
                android:layout_marginTop="4dp"
                android:layout_marginEnd="16dp"
                android:clickable="true"
                android:elevation="16dp"
                android:focusable="true"
                android:foreground="?android:attr/selectableItemBackground"
                app:cardBackgroundColor="@color/purple_500"
                app:cardCornerRadius="10dp">

            <TextView
                    android:id="@+id/connect_text"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_alignParentEnd="true"
                    android:layout_gravity="center"
                    android:layout_weight="1"
                    android:paddingHorizontal="12dp"
```

```
                android:text="Connect"
                android:textColor="@color/white"
                android:textSize="16sp"/>

        <ProgressBar
                android:id="@+id/activity_indicator"
                android:layout_width="20dp"
                android:layout_height="20dp"
                android:layout_gravity="center"
                android:layout_marginHorizontal="20dp"
                android:indeterminateOnly="true"
                android:indeterminateTint="@color/white"
                android:indeterminateTintMode="src_atop"
                android:keepScreenOn="true"
                android:visibility="gone"/>
    </androidx.cardview.widget.CardView>

</LinearLayout>

<androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/chat_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:clipToPadding="false"
        android:visibility="visible"
        tools:context=".MainActivity">

    <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            app:layout_constraintBottom_toTopOf="@+id/layout_message_input"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent">

        <androidx.recyclerview.widget.RecyclerView
                android:id="@+id/recycler_view"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:clipToPadding="false"
                android:paddingTop="70dp"
                android:paddingBottom="20dp"/>
    </RelativeLayout>
```

```
        <RelativeLayout
                android:id="@+id/layout_message_input"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:background="@android:color/white"
                android:clipToPadding="false"
                android:drawableTop="@android:color/black"
                android:elevation="18dp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintStart_toStartOf="parent">

            <EditText
                    android:id="@+id/message_edit_text"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:layout_centerVertical="true"
                    android:layout_marginStart="16dp"
                    android:layout_toStartOf="@+id/send_button"
                    android:background="@android:color/transparent"
                    android:hint="Enter Message"
                    android:inputType="text"
                    android:maxLines="6"
                    tools:ignore="Autofill"/>

            <Button
                    android:id="@+id/send_button"
                    android:layout_width="84dp"
                    android:layout_height="48dp"
                    android:layout_alignParentEnd="true"
                    android:background="@color/black"
                    android:foreground="?android:attr/selectableItemBackground"
                    android:text="Send"
                    android:textColor="@color/white"
                    android:textSize="12dp"/>
        </RelativeLayout>
    </androidx.constraintlayout.widget.ConstraintLayout>


</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

cellule de texte abstraite de l'interface utilisateur pour afficher le texte de manière cohérente

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>
```

message à gauche de l'interface utilisateur de chat

XML :

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

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

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

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

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

```
                app:layout_constraintStart_toStartOf="parent">

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

        <TextView
                android:id="@+id/dateText"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginLeft="4dp"
                android:layout_marginBottom="4dp"
                android:text="10:00"
                app:layout_constraintBottom_toBottomOf="@+id/card_message_other"
                app:layout_constraintLeft_toRightOf="@+id/card_message_other"/>
    </androidx.constraintlayout.widget.ConstraintLayout>


</LinearLayout>
```

message à droite de l'interface utilisateur de chat

XML :

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

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

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

    <androidx.cardview.widget.CardView
            android:id="@+id/card_message_me"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="right"
            android:layout_marginBottom="10dp"
            android:foreground="?android:attr/selectableItemBackground"
            app:cardBackgroundColor="@color/purple_500"
            app:cardCornerRadius="10dp"
```

```
            app:cardElevation="0dp"
            app:cardPreventCornerOverlap="false"
            app:cardUseCompatPadding="true"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent">

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

    </androidx.cardview.widget.CardView>

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

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

valeurs de couleur supplémentaires de l'interface utilisateur

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>
```

appliquer la liaison d'affichage

Nous tirons parti de la fonctionnalité Android Liaison d'affichage pour pouvoir référencer des classes de liaison pour notre mise en page XML. Pour activer la fonctionnalité, définissez l'option de génération `viewBinding` sur `true` dans `./app/build.gradle` :

Script Kotlin :

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

android {
//    ...

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

Il est maintenant temps de connecter l'interface utilisateur à notre code Kotlin :

Kotlin :

```
// ./app/src/main/java/com/chatterbox/myapp/MainActivity.kt
package com.chatterbox.myapp
// ...
const val TAG = "Chatterbox-MyApp"


class MainActivity : AppCompatActivity() {
//    ...

    private fun sendMessage(request: SendMessageRequest) {
        try {
            room?.sendMessage(
                request,
                object : SendMessageCallback {
                    override fun onRejected(request: SendMessageRequest, error:
 ChatError) {
                        runOnUiThread {
                            entries.addFailedRequest(request)
                            scrollToBottom()
```

```
                            Log.e(TAG, "Message rejected: ${error.errorMessage}")
                        }
                    }
                }
            )

            entries.addPendingRequest(request)

            binding.messageEditText.text.clear()
            scrollToBottom()
        } catch (error: Exception) {
            Log.e(TAG, error.message ?: "Unknown error occurred")
        }
    }

    private fun scrollToBottom() {
        binding.recyclerView.smoothScrollToPosition(entries.size - 1)
    }

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

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

Nous ajoutons également des méthodes pour supprimer des messages et déconnecter les utilisateurs du chat, qui peuvent être invoquées à l'aide du menu contextuel des messages de chat :

Kotlin :

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

package com.chatterbox.myapp
//    ...

class MainActivity : AppCompatActivity() {
//    ...
```

```
    private fun deleteMessage(request: DeleteMessageRequest) {
        room?.deleteMessage(
            request,
            object : DeleteMessageCallback {
                override fun onRejected(request: DeleteMessageRequest, error:
 ChatError) {
                    runOnUiThread {
                        Log.d(TAG, "Delete message rejected: ${error.errorMessage}")
                    }
                }
            }
        )
    }

    private fun disconnectUser(request: DisconnectUserRequest) {
        room?.disconnectUser(
            request,
            object : DisconnectUserCallback {
                override fun onRejected(request: DisconnectUserRequest, error:
 ChatError) {
                    runOnUiThread {
                        Log.d(TAG, "Disconnect user rejected: ${error.errorMessage}")
                    }
                }
            }
        )
    }
}
```

Gérer les demandes de messages de chat

Nous avons besoin d'un moyen de gérer nos demandes de messages de chat dans tous leurs états possibles :

- En attente : un message a été envoyé à une salle de chat mais n'a pas encore été confirmé ou rejeté.
- Confirmé : un message a été envoyé par la salle de chat à tous les utilisateurs (y compris à nous).
- Rejeté : un message a été rejeté par la salle de chat avec un objet d'erreur.

Nous conserverons les demandes de chat et les messages de chat non résolus dans une liste. La liste mérite une classe distincte, que nous appelons `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()
    }
}
```

Pour connecter notre liste à l'interface utilisateur, nous utilisons un adaptateur. Pour plus d'informations, consultez Liaison à des données avec AdapterView et Classes de liaison générées.

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
}
```

Étapes finales

Il est temps de connecter notre nouvel adaptateur, en liant une classe `ChatEntries` à `MainActivity` :

Kotlin :

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

package com.chatterbox.myapp
// ...

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

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

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

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

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

        setUpChatView()

        updateConnectionState(ConnectionState.DISCONNECTED)
```

```
    }

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

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

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

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

Comme nous avons déjà une classe chargée de suivre nos demandes de chat (`ChatEntries`), nous sommes prêts à implémenter le code pour manipuler `entries` dans `roomListener`. Nous mettrons à jour `entries` et `connectionState`, en fonction de l'événement auquel nous répondons :

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")
        }
    }
}
```

Vous devriez maintenant être en mesure d'exécuter votre application ! (Voir Créer et exécuter votre application.) N'oubliez pas de faire fonctionner votre serveur backend lorsque vous utilisez l'application. Vous pouvez le lancer depuis le terminal à la racine de notre projet à l'aide de cette commande : `./gradlew :auth-server:run` ou en exécutant la tâche Gradle `auth-server:run`directement depuis Android Studio.

Kit SDK de messagerie client Chat Amazon IVS : coroutines Kotlin, partie 1 : salles de chat

Il s'agit de la première partie d'un didacticiel en deux volets. Vous apprendrez les bases de l'utilisation du SDK de messagerie du Chat Amazon IVS en créant une application Android entièrement fonctionnelle à l'aide du langage de programmation Kotlin et des coroutines. Nous appelons l'application Chatterbox.

Avant de commencer le module, prenez quelques minutes pour vous familiariser avec les prérequis, les concepts clés des jetons de discussion et le serveur principal nécessaire à la création de salles de chat.

Ces didacticiels sont conçus pour les développeurs Android expérimentés qui découvrent le SDK de messagerie d'IVS Chat. Vous devrez être à l'aise avec le langage de programmation Kotlin et la création d'interfaces utilisateur sur la plateforme Android.

Cette première partie du didacticiel est divisée en plusieurs sections :

1. the section called “Configurer un serveur d'authentification/d'autorisation local”
2. the section called “Créer un projet Chatterbox”
3. the section called “Se connecter à une salle de chat et observer les mises à jour de la connexion”
4. the section called “Créer un fournisseur de jetons”
5. the section called “Étapes suivantes”

Pour une documentation complète sur le SDK, commencez par le SDK de messagerie client Chat Amazon IVS (ici dans le Guide de l'utilisateur Chat Amazon IVS) et la Messagerie client de chat : référence du SDK pour Android (sur GitHub).

Prérequis

- Familiarisez-vous avec Kotlin et la création d'applications sur la plateforme Android. Si vous n'êtes pas familiarisé avec la création d'applications pour Android, découvrez les bases dans le guide Créer votre première application pour les développeurs Android.
- Lisez et assurez-vous d'avoir compris Mise en route avec IVS Chat.
- Créez un utilisateur AWS IAM avec les fonctionnalités `CreateChatToken` et `CreateRoom` définies dans une politique IAM existante. (Consultez Mise en route avec IVS Chat.)
- Assurez-vous que les clés secrètes et d'accès de cet utilisateur sont stockées dans un fichier d'informations d'identification AWS. Pour obtenir des instructions, consultez le Guide de l'utilisateur de l'interface de ligne de commande AWS (en particulier les paramètres de configuration et de fichier d'informations d'identification).
- Créez une salle de chat et enregistrez son ARN. Consultez Mise en route avec IVS Chat. (Si vous n'enregistrez pas l'ARN, vous pourrez le consulter ultérieurement à l'aide de la console ou de l'API Chat.)

Configurer un serveur d'authentification/d'autorisation local

Votre serveur backend est chargé à la fois de créer des salles de chat et de générer les jetons de chat nécessaires au SDK de chat IVS pour Android pour authentifier et autoriser vos clients à accéder à vos salles de chat.

Consultez la section Créer un jeton de chat dans Mise en route avec Chat Amazon IVS. Comme le montre l'organigramme, votre code côté serveur est chargée de créer un jeton de chat. Cela signifie que votre application doit fournir ses propres moyens de générer un jeton de chat en demandant un jeton à votre application côté serveur.

Nous utilisons l'infrastructure Ktor pour créer un serveur local en direct qui gère la création de jetons de chat à l'aide de votre environnement AWS local.

À ce stade, nous nous attendons à ce que vos informations d'identification AWS soient correctement configurées. Pour savoir comment procéder, consultez Configuration des informations d'identification AWS temporaires et de la région AWS pour le développement.

Créez un nouveau répertoire et appelez-le `chatterbox` et, à l'intérieur, un autre, appelé `auth-server`.

La structure de notre serveur sera la suivante :

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

Remarque : vous pouvez directement copier/coller le code ici dans les fichiers référencés.

Ensuite, nous ajoutons toutes les dépendances et tous les plugins nécessaires au fonctionnement de notre serveur d'authentification :

Script Kotlin :

```
// ./auth-server/build.gradle.kts

plugins {
   application
   kotlin("jvm")
   kotlin("plugin.serialization").version("1.7.10")
}

application {
   mainClass.set("io.ktor.server.netty.EngineMain")
}

dependencies {
   implementation("software.amazon.awssdk:ivschat:2.18.1")
   implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.7.20")

   implementation("io.ktor:ktor-server-core:2.1.3")
```

```
    implementation("io.ktor:ktor-server-netty:2.1.3")
    implementation("io.ktor:ktor-server-content-negotiation:2.1.3")
    implementation("io.ktor:ktor-serialization-kotlinx-json:2.1.3")

    implementation("ch.qos.logback:logback-classic:1.4.4")
}
```

Nous devons maintenant configurer la fonctionnalité de journalisation pour le serveur d'authentification. (Pour plus d'informations, veuillez consulter Configure logger.)

XML :

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

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

Le serveur Ktor nécessite des paramètres de configuration, qu'il charge automatiquement à partir du fichier `application.*` du répertoire `resources`. Nous les ajoutons donc également. (Pour plus d'informations, consultez Configuration in a file.)

HOCON :

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

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

```
    }
}
```

Enfin, implémentons notre serveur :

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()
                )
            )
        }
    }
}
```

Créer un projet Chatterbox

Pour créer un projet Android, installez et ouvrez Android Studio.

Suivez les étapes répertoriées dans le guide officiel Android Créer un projet.

- Dans Choisir le type de projet, choisissez le modèle de projet Empty Activity (Activité vide) pour notre application Chatterbox.

- Dans Configurer votre projet, choisissez les valeurs suivantes pour les champs de configuration :
  - Nom : My App
  - Nom de package : com.chatterbox.myapp
  - Emplacement d'enregistrement : pointez sur le répertoire `chatterbox` créé à l'étape précédente
  - Langage : Kotlin
  - Niveau d'API minimum : API 21 : Android 5.0 (Lollipop)

Après avoir correctement spécifié tous les paramètres de configuration, la structure de nos fichiers dans le dossier `chatterbox` doit ressembler à ce qui suit :

```
- 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
```

Maintenant que nous avons un projet Android fonctionnel, nous pouvons ajouter com.amazonaws:ivs-chat-messaging et org.jetbrains.kotlinx:kotlinx-coroutines-core à nos dépendances `build.gradle`. (Pour plus d'informations sur la boîte à outils de génération Gradle, voir Configurer votre build.)

Remarque : en haut de chaque extrait de code, il y a un chemin vers le fichier dans lequel vous devez apporter des modifications à votre projet. Le chemin est relatif par rapport à la racine du projet.

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'

// ...
```

```
}
```

Une fois la nouvelle dépendance ajoutée, exécutez Synchroniser le projet avec les fichiers Gradle dans Android Studio pour synchroniser le projet avec la nouvelle dépendance. (Pour de plus amples informations, veuillez consulter la section Ajouter des dépendances de build.)

Pour exécuter facilement notre serveur d'authentification (créé dans la section précédente) à partir de la racine du projet, nous l'incluons en tant que nouveau module dans `settings.gradle`. (Pour plus d'informations, voir Structurer et construire un composant de logiciel avec Gradle.)

Script Kotlin :

```
// ./settings.gradle

// ...

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

À présent, comme `auth-server` est inclus dans le projet Android, vous pouvez exécuter le serveur d'authentification à l'aide de la commande suivante depuis la racine du projet :

Shell :

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

Se connecter à une salle de chat et observer les mises à jour de la connexion

Pour ouvrir une connexion à une salle de chat, nous utilisons le rappel du cycle de vie de l'activité onCreate(), qui se déclenche lorsque l'activité est créée pour la première fois. Le constructeur ChatRoom nous oblige à fournir `region` et `tokenProvider` pour lancer une connexion à une salle.

Remarque : la fonction `fetchChatToken` présentée dans l'extrait ci-dessous sera implémentée dans la section suivante.

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)
   }

// ...
}
```

L'affichage et la réaction aux modifications de la connexion à une salle de chat est un élément essentiel pour la création d'une application de chat comme `chatterbox`. Avant de pouvoir commencer à interagir avec la salle, nous devons nous abonner aux événements relatifs à l'état de connexion de la salle de chat pour obtenir des mises à jour.

Dans le SDK Chat pour la coroutine, `ChatRoom` attend de nous que nous gérions les événements du cycle de vie des salles dans Flux. Pour l'instant, les fonctions enregistreront uniquement les messages de confirmation, lorsqu'ils sont invoqués :

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")
                }
            }
        }
    }
}
```

Ensuite, nous devons fournir la possibilité de lire l'état de la connexion à la salle. Nous le conserverons dans la propriété `MainActivity.kt` et l'initialiserons à l'état DÉCONNECTÉ par défaut pour les salles (voir `ChatRoom` `state` dans la référence du SDK IVS Chat pour Android). Pour pouvoir maintenir l'état local à jour, nous devons implémenter une fonction de mise à jour de l'état ; appelons-le `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")
          }
      }
}
```

Ensuite, nous intégrons notre fonction de mise à jour de l'état à la propriété Chatroom.Listener :

Kotlin :

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

package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
// ...

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
```

```
        // Create room instance
        room = ChatRoom(REGION, ::fetchChatToken).apply {
            lifecycleScope.launch {
                stateChanges().collect { state ->
                    Log.d(TAG, "state change to $state")
                    updateConnectionState(state)

                }
            }

      // ...

      }
   }
}
```

Maintenant que nous sommes en mesure d'enregistrer, d'écouter et de réagir aux mises à jour d'état de ChatRoom, il est temps d'initialiser une connexion :

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)
      }
   }

   // ...
}
```

Créer un fournisseur de jetons

Il est temps de créer une fonction chargée de créer et de gérer des jetons de chat dans notre application. Dans cet exemple, nous utilisons le client HTTP Retrofit pour Android.

Avant de pouvoir envoyer du trafic réseau, nous devons configurer une configuration de sécurité réseau pour Android. Pour plus d'informations, consultez Network security configuration.) Nous commençons par ajouter des autorisations réseau au fichier App Manifest. Notez la balise `user-permission` et l'attribut `networkSecurityConfig` ajoutés, qui indiqueront notre nouvelle configuration de sécurité réseau. Dans le code ci-dessous, remplacez *<version>* par le numéro de version actuel du SDK de chat pour Android (par exemple, 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")
}
```

Déclarez votre adresse IP locale, par exemple `10.0.2.2` et les domaines `localhost` comme fiables, pour commencer à échanger des messages avec notre backend :

XML :

```
// ./app/src/main/res/xml/network_security_config.xml

<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">10.0.2.2</domain>
        <domain includeSubdomains="true">localhost</domain>
    </domain-config>
</network-security-config>
```

Ensuite, nous devons ajouter une nouvelle dépendance, ainsi que l'ajout d'un convertisseur Gson pour analyser les réponses HTTP. Dans le code ci-dessous, remplacez *<version>* par le numéro de version actuel du SDK de chat pour Android (par exemple, 1.1.0).

Script Kotlin :

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

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

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

Pour récupérer un jeton de chat, nous devons effectuer une requête HTTP POST depuis notre application `chatterbox`. Nous définissons la demande dans une interface que Retrofit doit implémenter. (Voir la documentation de Retrofit. Familiarisez-vous également avec la spécification du point de terminaison CreateChatToken.)

Kotlin :

```
// ./app/src/main/java/com/chatterbox/myapp/network/ApiService.kt

package com.chatterbox.myapp.network

import com.amazonaws.ivs.chat.messaging.ChatToken
import retrofit2.Call
import retrofit2.http.Body
```

```
import retrofit2.http.POST

data class CreateTokenParams(var userId: String, var roomIdentifier: String)

interface ApiService {
   @POST("create_chat_token")
   fun createChatToken(@Body params: CreateTokenParams): Call<ChatToken>
}


// ./app/src/main/java/com/chatterbox/myapp/network/RetrofitFactory.kt

package com.chatterbox.myapp.network

import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object RetrofitFactory {
   private const val BASE_URL = "http://10.0.2.2:3000"

   fun makeRetrofitService(): ApiService {
       return Retrofit.Builder()
           .baseUrl(BASE_URL)
           .addConverterFactory(GsonConverterFactory.create())
           .build().create(ApiService::class.java)
   }
}
```

Maintenant que le réseau est configuré, il est temps d'ajouter une fonction chargée de créer et de gérer notre jeton de chat. Nous l'ajoutons à `MainActivity.kt`, qui a été automatiquement créé lors de la génération du projet :

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)
        }
    })
  }
}
```

Étapes suivantes

Maintenant que vous avez établi une connexion à la salle de chat, passez à la seconde partie de ce didacticiel Coroutines Kotlin, Messages et événements.

Kit SDK de messagerie client Chat Amazon IVS : didacticiel Coroutines Kotlin, partie 2 : messages et événements

Cette seconde et dernière partie du didacticiel est divisée en plusieurs sections :

1. the section called “Création d'une interface utilisateur pour l'envoi de messages”
   a. the section called “Mise en page principale de l'interface utilisateur”
   b. the section called “cellule de texte abstraite de l'interface utilisateur pour afficher le texte de manière cohérente”
   c. the section called “message à gauche de l'interface utilisateur de chat”
   d. the section called “message à droite de l'interface utilisateur”
   e. the section called “valeurs de couleur supplémentaires de l'interface utilisateur”
2. the section called “appliquer la liaison d'affichage”
3. the section called “Gérer les demandes de messages de chat”
4. the section called “Étapes finales”

Pour une documentation complète sur le SDK, commencez par le SDK de messagerie client Chat Amazon IVS (ici dans le Guide de l'utilisateur Chat Amazon IVS) et la Messagerie client de chat : référence du SDK pour Android (sur GitHub).

Prérequis

Assurez-vous d'avoir terminé la première partie de ce didacticiel relative aux Salles de chat.

Création d'une interface utilisateur pour l'envoi de messages

Maintenant que nous avons initialisé avec succès la connexion à la salle de chat, il est temps d'envoyer notre premier message. Pour cette fonctionnalité, une interface utilisateur est nécessaire. Nous ajouterons :

- Bouton `connect/disconnect`
- une saisie de message avec un bouton `send`
- une liste de messages dynamiques. Pour la créer, nous utilisons Android Jetpack RecyclerView.

Mise en page principale de l'interface utilisateur

Consultez les mises en page Android Jetpack dans la documentation pour les développeurs Android.

XML :

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


<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout xmlns:android="http://
schemas.android.com/apk/res/android"
                                                     xmlns:app="http://
schemas.android.com/apk/res-auto"
                                                     xmlns:tools="http://
schemas.android.com/tools"

 android:layout_width="match_parent"

 android:layout_height="match_parent">

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
                  xmlns:app="http://schemas.android.com/apk/res-auto"
                  android:id="@+id/connect_view"
                  android:layout_width="match_parent"
                  android:layout_height="match_parent"
                  android:gravity="center"
                  android:orientation="vertical">

        <androidx.cardview.widget.CardView
                android:id="@+id/connect_button"
```

```
                android:layout_width="match_parent"
                android:layout_height="48dp"
                android:layout_gravity=""
                android:layout_marginStart="16dp"
                android:layout_marginTop="4dp"
                android:layout_marginEnd="16dp"
                android:clickable="true"
                android:elevation="16dp"
                android:focusable="true"
                android:foreground="?android:attr/selectableItemBackground"
                app:cardBackgroundColor="@color/purple_500"
                app:cardCornerRadius="10dp">

            <TextView
                    android:id="@+id/connect_text"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_alignParentEnd="true"
                    android:layout_gravity="center"
                    android:layout_weight="1"
                    android:paddingHorizontal="12dp"
                    android:text="Connect"
                    android:textColor="@color/white"
                    android:textSize="16sp"/>

            <ProgressBar
                    android:id="@+id/activity_indicator"
                    android:layout_width="20dp"
                    android:layout_height="20dp"
                    android:layout_gravity="center"
                    android:layout_marginHorizontal="20dp"
                    android:indeterminateOnly="true"
                    android:indeterminateTint="@color/white"
                    android:indeterminateTintMode="src_atop"
                    android:keepScreenOn="true"
                    android:visibility="gone"/>
        </androidx.cardview.widget.CardView>

    </LinearLayout>

    <androidx.constraintlayout.widget.ConstraintLayout
            android:id="@+id/chat_view"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
```

```
            android:clipToPadding="false"
            android:visibility="visible"
            tools:context=".MainActivity">

        <RelativeLayout
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:orientation="vertical"
                app:layout_constraintBottom_toTopOf="@+id/layout_message_input"
                app:layout_constraintEnd_toEndOf="parent"
                app:layout_constraintStart_toStartOf="parent">

            <androidx.recyclerview.widget.RecyclerView
                    android:id="@+id/recycler_view"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:clipToPadding="false"
                    android:paddingTop="70dp"
                    android:paddingBottom="20dp"/>
        </RelativeLayout>

        <RelativeLayout
                android:id="@+id/layout_message_input"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:background="@android:color/white"
                android:clipToPadding="false"
                android:drawableTop="@android:color/black"
                android:elevation="18dp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintStart_toStartOf="parent">

            <EditText
                    android:id="@+id/message_edit_text"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:layout_centerVertical="true"
                    android:layout_marginStart="16dp"
                    android:layout_toStartOf="@+id/send_button"
                    android:background="@android:color/transparent"
                    android:hint="Enter Message"
                    android:inputType="text"
                    android:maxLines="6"
                    tools:ignore="Autofill"/>
```

```
            <Button
                    android:id="@+id/send_button"
                    android:layout_width="84dp"
                    android:layout_height="48dp"
                    android:layout_alignParentEnd="true"
                    android:background="@color/black"
                    android:foreground="?android:attr/selectableItemBackground"
                    android:text="Send"
                    android:textColor="@color/white"
                    android:textSize="12dp"/>
        </RelativeLayout>
    </androidx.constraintlayout.widget.ConstraintLayout>


</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

cellule de texte abstraite de l'interface utilisateur pour afficher le texte de manière cohérente

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>
```

message à gauche de l'interface utilisateur de chat

XML :

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

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

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

```
            android:text="UserName"/>

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

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

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

        <TextView
                android:id="@+id/dateText"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginLeft="4dp"
                android:layout_marginBottom="4dp"
                android:text="10:00"
                app:layout_constraintBottom_toBottomOf="@+id/card_message_other"
                app:layout_constraintLeft_toRightOf="@+id/card_message_other"/>
    </androidx.constraintlayout.widget.ConstraintLayout>


</LinearLayout>
```

message à droite de l'interface utilisateur

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>
```

valeurs de couleur supplémentaires de l'interface utilisateur

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>
```

appliquer la liaison d'affichage

Nous tirons parti de la fonctionnalité Android Liaison d'affichage pour pouvoir référencer des classes de liaison pour notre mise en page XML. Pour activer la fonctionnalité, définissez l'option de génération `viewBinding` sur `true` dans `./app/build.gradle` :

Script Kotlin :

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

android {
//    ...

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

Il est maintenant temps de connecter l'interface utilisateur à notre code Kotlin :

Kotlin :

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

package com.chatterbox.myapp
// ...

class MainActivity : AppCompatActivity() {
    // ...
```

```
    private lateinit var binding: ActivityMainBinding

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

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

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

        setUpChatView()

        updateConnectionState(ChatRoom.State.DISCONNECTED)
    }

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

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

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

```
}
```

Nous ajoutons également des méthodes pour supprimer des messages et déconnecter les utilisateurs du chat, qui peuvent être invoquées à l'aide du menu contextuel des messages de chat :

Kotlin :

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

package com.chatterbox.myapp
//    ...

class MainActivity : AppCompatActivity() {
//    ...

    private fun deleteMessage(request: DeleteMessageRequest) {
        lifecycleScope.launch {
           try {
               room?.awaitDeleteMessage(request)
           } catch (exception: ChatException) {
               Log.e(TAG, "Delete message rejected: ${exception.message}")
           } catch (exception: Exception) {
               Log.e(TAG, exception.message ?: "Unknown error occurred")
           }
        }
    }

    private fun disconnectUser(request: DisconnectUserRequest) {
        lifecycleScope.launch {
           try {
               room?.awaitDisconnectUser(request)
           } catch (exception: ChatException) {
               Log.e(TAG, "Disconnect user rejected: ${exception.message}")
           } catch (exception: Exception) {
               Log.e(TAG, exception.message ?: "Unknown error occurred")
           }
        }
    }
}
```

Gérer les demandes de messages de chat

Nous avons besoin d'un moyen de gérer nos demandes de messages de chat dans tous leurs états possibles :

- En attente : un message a été envoyé à une salle de chat mais n'a pas encore été confirmé ou rejeté.
- Confirmé : un message a été envoyé par la salle de chat à tous les utilisateurs (y compris à nous).
- Rejeté : un message a été rejeté par la salle de chat avec un objet d'erreur.

Nous conserverons les demandes de chat et les messages de chat non résolus dans une liste. La liste mérite une classe distincte, que nous appelons `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()
    }
}
```

Pour connecter notre liste à l'interface utilisateur, nous utilisons un adaptateur. Pour plus d'informations, consultez Liaison à des données avec AdapterView et Classes de liaison générées.

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
}
```

Étapes finales

Il est temps de connecter notre nouvel adaptateur, en liant une classe `ChatEntries` à `MainActivity` :

Kotlin :

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

package com.chatterbox.myapp
// ...

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

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

    // ...

    private fun setUpChatView() {
```

```
        adapter = ChatListAdapter(entries, ::disconnectUser)
        entries.adapter = adapter

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

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

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

Comme nous avons déjà une classe chargée de suivre nos demandes de chat (`ChatEntries`), nous sommes prêts à implémenter le code pour manipuler `entries` dans roomListener. Nous mettrons à jour `entries` et `connectionState`, en fonction de l'événement auquel nous répondons :

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)
```

```
    }

// ...

}
```

Vous devriez maintenant être en mesure d'exécuter votre application ! (Voir Créer et exécuter votre application.) N'oubliez pas de faire fonctionner votre serveur backend lorsque vous utilisez l'application. Vous pouvez le lancer depuis le terminal à la racine de notre projet à l'aide de cette commande : `./gradlew :auth-server:run` ou en exécutant la tâche Gradle `auth-server:run`directement depuis Android Studio.

Kit SDK de messagerie client Amazon IVS Chat : guide iOS

Le kit SDK de messagerie client Amazon Interactive Video (IVS) Chat pour iOS fournit des interfaces qui vous permettent d'intégrer notre API de messagerie IVS Chat sur les plateformes utilisant le langage de programmation Swift d'Apple.

Dernière version du kit SDK de messagerie client IVS Chat pour iOS : 1.0.0 (notes de mise à jour)

Documentation de référence et didacticiels : pour plus d'informations sur les méthodes les plus importantes disponibles dans le kit SDK de messagerie client Amazon IVS Chat pour iOS, veuillez consulter la documentation de référence à l'adresse : https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/. Ce référentiel contient également divers articles et didacticiels.

Exemple de code : voir l'exemple de référentiel iOS sur GitHub : https://github.com/aws-samples/amazon-ivs-chat-for-ios-demo.

Exigences de la plateforme : iOS 13.0 ou version ultérieure est requis pour le développement.

Démarrage

Nous vous recommandons d'intégrer le kit SDK via Swift Package Manager. Vous pouvez également utiliser CocoaPods ou intégrer le cadre manuellement.

Après avoir intégré le kit SDK, vous pouvez l'importer en ajoutant le code suivant en haut de votre fichier Swift concerné :

```
import AmazonIVSChatMessaging
```

Swift Package Manager

Pour utiliser la bibliothèque `AmazonIVSChatMessaging` dans un projet Swift Package Manager, ajoutez-la aux dépendances de votre package et aux dépendances de vos cibles pertinentes :

1. Téléchargez la dernière version du `.xcframework` depuis https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. Dans votre terminal, exécutez :

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

3. Prenez la sortie de l'étape précédente et collez-la dans la propriété checksum de `.binaryTarget`, comme indiqué ci-dessous dans le fichier `Package.swift` de votre projet :

```
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

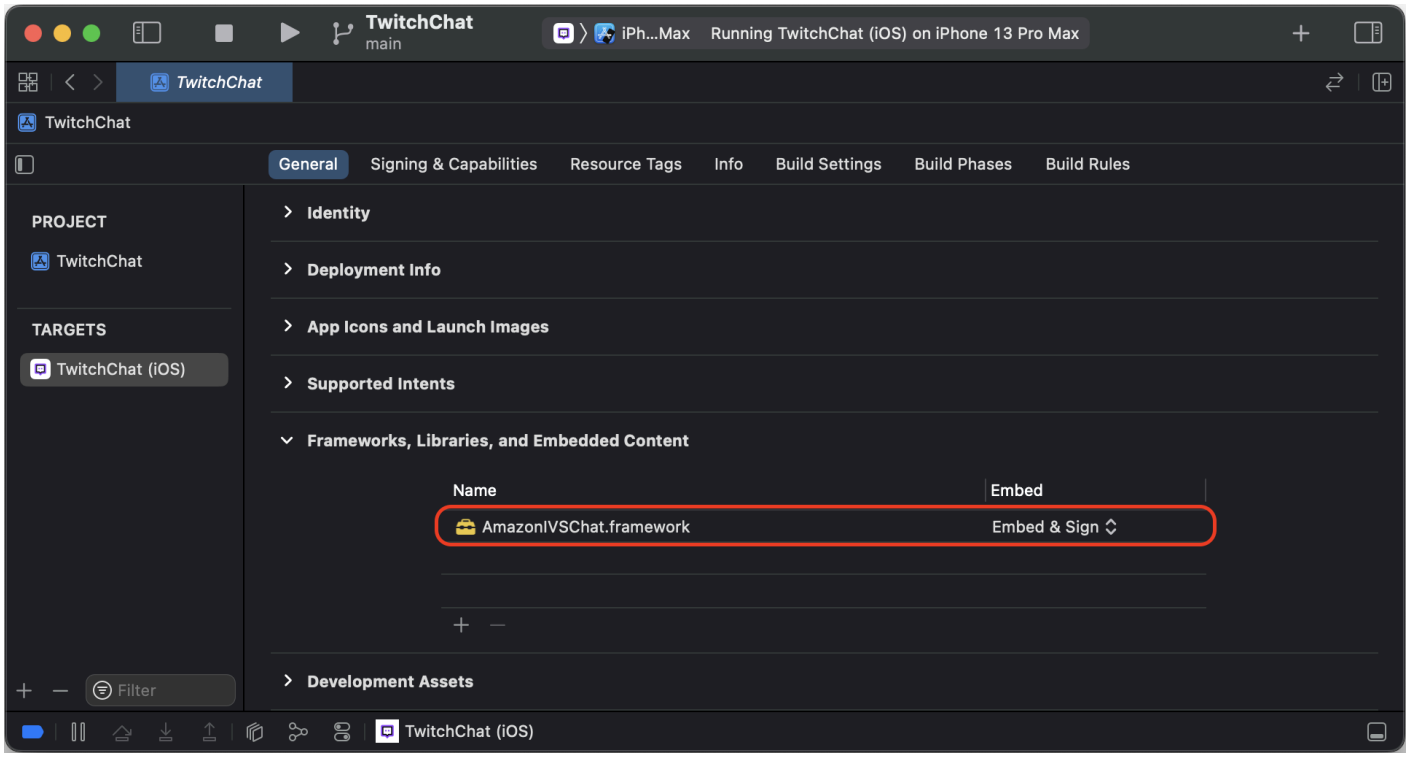
Les versions sont publiées via CocoaPods sous le nom `AmazonIVSChatMessaging`. Ajoutez cette dépendance à votre Podfile :

```
pod 'AmazonIVSChat'
```

Exécutez `pod install` et le kit SDK sera disponible dans votre `.xcworkspace`.

Installation manuelle

1. Téléchargez la dernière version depuis https://ivschat.live-video.net/1.0.0/AmazonIVSChatMessaging.xcframework.zip.
2. Extrayez le contenu de l'archive. `AmazonIVSChatMessaging.xcframework` contient le kit SDK pour l'appareil et le simulateur.
3. Intégrez le `AmazonIVSChatMessaging.xcframework` extrait en le faisant glisser dans la section Frameworks, Libraries, and Embedded Content (Cadre, bibliothèques et contenu intégré) de l'onglet General (Général) de votre cible d'application :



Utilisation de l'SDK

Se connecter à une salle de chat

Avant de commencer, vous devez être familiarisé avec la Mise en route avec Amazon IVS Chat. Consultez également les exemples d'applications pour le web, Android et iOS.

Pour se connecter à une salle de chat, votre application a besoin d'un moyen de récupérer un jeton de chat fourni par votre backend. Votre application récupérera probablement un jeton de chat à l'aide d'une demande réseau envoyée à votre backend.

Pour transférer ce jeton de chat récupéré au kit SDK, le modèle `ChatRoom` du kit SDK exige que vous fournissiez une fonction `async` ou l'instance d'un objet conforme au protocole `ChatTokenProvider` fourni au moment de l'initialisation. La valeur renvoyée par l'une de ces méthodes doit être une instance du modèle `ChatToken` du kit SDK.

Remarque : vous renseignez les instances du modèle `ChatToken` à l'aide des données récupérées depuis votre backend. Les champs requis pour initialiser une instance `ChatToken` sont les mêmes que les champs de la réponse CreateChatToken. Pour plus d'informations sur l'initialisation des instances du modèle `ChatToken`, veuillez consulter la rubrique Créer une instance de ChatToken. Souvenez-vous que votre backend est chargé de fournir des données dans la réponse `CreateChatToken` à votre application. La manière dont vous décidez de communiquer avec votre backend pour générer des jetons de chat dépend de votre application et de son infrastructure.

Après avoir choisi votre stratégie pour fournir un `ChatToken` au kit SDK, appelez `.connect()` après avoir initialisé une instance `ChatRoom` avec votre fournisseur de jetons et la région AWS que votre backend a utilisée pour créer la salle de chat à laquelle vous essayez de vous connecter. Veuillez noter que `.connect()` est une fonction asynchrone de lancement :

```
import AmazonIVSChatMessaging

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

Se conformer au protocole ChatTokenProvider

Pour le paramètre `tokenProvider` dans l'initialiseur de `ChatRoom`, vous pouvez fournir une instance de `ChatTokenProvider`. Voici un exemple d'objet conforme à `ChatTokenProvider` :

```
import AmazonIVSChatMessaging

// This object should exist somewhere in your app
class ChatService: ChatTokenProvider {
   func getChatToken() async throws -> ChatToken {
```

```
      let request = YourApp.getTokenURLRequest
      let data = try await URLSession.shared.data(for: request).0
      ...
      return ChatToken(
         token: String(data: data, using: .utf8)!,
         tokenExpirationTime: ..., // this is optional
         sessionExpirationTime: ... // this is optional
      )
   }
}
```

Vous pouvez ensuite prendre une instance de cet objet conforme et la transmettre à l'initialiseur de `ChatRoom` :

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

Fournir une fonction asynchrone dans Swift

Supposons que vous disposiez déjà d'un gestionnaire que vous utilisez pour gérer les demandes réseau de votre application. Elle peut ressembler à ceci :

```
import AmazonIVSChatMessaging

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

Vous pouvez simplement ajouter une autre fonction dans votre gestionnaire afin de récupérer un `ChatToken` depuis votre backend :

```
import AmazonIVSChatMessaging
```

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

Ensuite, utilisez la référence à cette fonction dans Swift lors de l'initialisation d'une `ChatRoom` :

```
import AmazonIVSChatMessaging

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

Créer une instance de ChatToken

Vous pouvez facilement créer une instance de `ChatToken` à l'aide de l'initialiseur fourni dans le kit SDK. Veuillez consulter la documentation dans `Token.swift` pour en savoir plus sur les propriétés de `ChatToken`.

```
import AmazonIVSChatMessaging

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

Utiliser le protocole Decodable

Si, lors de l'interfaçage avec l'API IVS Chat, votre backend décide de simplement transmettre la réponse CreateChatToken à votre application frontend, vous pouvez profiter de la conformité de `ChatToken` au protocole `Decodable` de Swift. Cependant, il y a un hic.

La charge utile de réponse `CreateChatToken` utilise des chaînes pour les dates formatées à l'aide de la norme ISO 8601 pour les horodatages Internet. Normalement, dans Swift, vous fourniriez `JSONDecoder.DateDecodingStrategy.iso8601` en tant que valeur de la propriété `.dateDecodingStrategy` de `JSONDecoder`. Cependant, `CreateChatToken` utilise des fractions

de secondes de haute précision dans ses chaînes, une caractéristique qui n'est pas pris en charge par `JSONDecoder.DateDecodingStrategy.iso8601`.

Pour plus de commodité, le kit SDK fournit une extension publique sur `JSONDecoder.DateDecodingStrategy` avec une stratégie `.preciseISO8601` supplémentaire qui vous permet d'utiliser `JSONDecoder` lors du décodage d'une instance de `ChatToken` :

```
import AmazonIVSChatMessaging

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

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

Se déconnecter d'une salle de chat

Pour vous déconnecter manuellement d'une instance `ChatRoom` à laquelle vous vous êtes connecté, appelez `room.disconnect()`. Par défaut, les salles de chat appellent automatiquement cette fonction lorsqu'elles sont désallouées.

```
import AmazonIVSChatMessaging

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

// Disconnect
room.disconnect()
```

Recevoir un message/événement sur le chat

Pour envoyer et recevoir des messages dans votre salle de chat, vous devez fournir un objet conforme au protocole `ChatRoomDelegate`, après avoir initialisé une instance de `ChatRoom` et appelé `room.connect()`. Voici un exemple classique utilisant `UIViewController` :

```
import AmazonIVSChatMessaging
import Foundation
import UIKit

class ViewController: UIViewController {
```

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

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

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

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

Recevoir une notification lors d'un changement de connexion

Comme il fallait s'y attendre, vous ne pouvez pas effectuer d'actions telles que l'envoi d'un message dans une salle tant que celle-ci n'est pas complètement connectée. L'architecture du kit SDK tente d'encourager la connexion à une ChatRoom sur un thread d'arrière-plan par le biais d'API asynchrones. Si vous souhaitez créer quelque chose dans votre interface utilisateur qui désactive par exemple un bouton d'envoi de message, le kit SDK propose deux stratégies permettant d'être averti lorsque l'état de connexion d'une salle de chat change, à l'aide de `Combine` ou `ChatRoomDelegate`. Elles sont décrites ci-dessous.

Important : l'état de connexion d'une salle de chat peut également changer en raison de facteurs tels qu'une interruption de la connexion réseau. Tenez-en compte lors de la création de votre application.

Utiliser Combine

Chaque instance de `ChatRoom` est accompagnée de son propre éditeur `Combine` sous la forme de la propriété `state` :

```
import AmazonIVSChatMessaging
```

```
import Combine

var cancellables: Set<AnyCancellable> = []

let room = ChatRoom(...)
room.state.sink { state in
   switch state {
   case .connecting:
      let image = UIImage(named: "antenna.radiowaves.left.and.right")
      sendMessageButton.setImage(image, for: .normal)
      sendMessageButton.isEnabled = false
   case .connected:
      let image = UIImage(named: "paperplane.fill")
      sendMessageButton.setImage(image, for: .normal)
      sendMessageButton.isEnabled = true
   case .disconnected:
      let image = UIImage(named: "antenna.radiowaves.left.and.right.slash")
      sendMessageButton.setImage(image, for: .normal)
      sendMessageButton.isEnabled = false
   }
}.assign(to: &cancellables)

// Connect to `ChatRoom` on a background thread
Task(priority: .background) {
   try await room.connect()
}
```

Utiliser ChatRoomDelegate

Vous pouvez également utiliser les fonctions facultatives `roomDidConnect(_:)`, `roomIsConnecting(_:)` et `roomDidDisconnect(_:)` au sein d'un objet conforme à `ChatRoomDelegate`. Voici un exemple utilisant un `UIViewController` :

```
import AmazonIVSChatMessaging
import Foundation
import UIKit

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

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

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

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

Effectuer des actions dans une salle de chat

Différents utilisateurs disposent de capacités différentes pour les actions qu'ils peuvent effectuer dans une salle de chat : envoyer un message, supprimer un message ou déconnecter un utilisateur, par exemple. Pour effectuer l'une de ces actions, appelez `perform(request:)` sur une `ChatRoom` connectée, en transmettant une instance de l'un des objets `ChatRequest` fournis dans le kit SDK. Les demandes prises en charge se trouvent dans `Request.swift`.

Certaines actions effectuées dans une salle de chat nécessitent que des capacités spécifiques soient accordées aux utilisateurs connectés lorsque votre application backend appelle `CreateChatToken`. De par sa conception, le kit SDK ne peut pas discerner les capacités d'un utilisateur connecté. Par conséquent, bien que vous puissiez tenter d'effectuer des actions de modérateur dans une instance connectée de `ChatRoom`, l'API du plan de contrôle décide à terme si cette action aboutira.

Toutes les actions qui passent par `room.perform(request:)` attendent que la salle reçoive l'instance attendue d'un modèle (dont le type est associé à l'objet de la demande lui-même) correspondant au `requestId` du modèle reçu et de l'objet de la demande. S'il y a un problème avec

la demande, `ChatRoom` lance toujours une erreur sous la forme d'un `ChatError`. La définition de `ChatError` se trouve dans `Error.swift`.

Envoi d'un message

Pour envoyer un message de chat, utilisez une instance de `SendMessageRequest` :

```
import AmazonIVSChatMessaging

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

Comme indiqué plus haut, `room.perform(request:)` revient une fois qu'un `ChatMessage` correspondant est reçu par la `ChatRoom`. En cas de problème avec la demande (par exemple, dépassement de la limite de caractères du message pour une salle), une instance de `ChatError` est lancée à la place. Vous pouvez ensuite faire apparaître ces informations utiles dans votre interface utilisateur :

```
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)
}
```

Ajouter des métadonnées à un message

Lors de l'envoi d'un message, vous pouvez ajouter les métadonnées qui lui seront associées. `SendMessageRequest` possède une propriété `attributes`, avec laquelle vous pouvez initialiser votre demande. Les données que vous y associez sont jointes au message lorsque d'autres utilisateurs le reçoivent dans la salle.

Voici un exemple d'ajout de données d'émoticônes à un message en cours d'envoi :

```
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"
      ]
   )
)
```

L'utilisation de `attributes` dans une `SendMessageRequest` peut être extrêmement utile pour créer des fonctionnalités complexes dans votre produit de chat. Par exemple, il serait possible de créer une fonctionnalité de threading à l'aide du dictionnaire d'attributs `[String : String]` dans une `SendMessageRequest`.

La charge utile des `attributes` est extrêmement flexible et puissante. Utilisez-la pour obtenir des informations sur votre message que vous ne pourriez pas obtenir autrement. L'utilisation d'attributs est beaucoup plus simple que, par exemple, l'analyse de la chaîne d'un message en vue d'obtenir des informations sur des éléments tels que des émoticônes.

Supprimer un message

La suppression d'un message de chat équivaut à la création d'un tel message. Utilisez la fonction `room.perform(request:)` sur `ChatRoom` pour y parvenir en créant une instance de `DeleteMessageRequest`.

Pour accéder facilement aux instances précédentes des messages de chat reçus, transmettez la valeur de `message.id` à l'initialiseur de `DeleteMessageRequest`.

Vous pouvez éventuellement fournir une chaîne reason à `DeleteMessageRequest` afin de la faire apparaître dans votre interface utilisateur.

```
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!"
   )
)
```

Comme il s'agit d'une action de modérateur, votre utilisateur n'a peut-être pas réellement la capacité de supprimer le message d'un autre utilisateur. Vous pouvez utiliser le mécanisme de fonctions lançables de Swift pour faire apparaître un message d'erreur dans votre interface utilisateur lorsqu'un utilisateur tente de supprimer un message sans la capacité appropriée.

Lorsque votre backend appelle `CreateChatToken` pour un utilisateur, il doit transmettre `"DELETE_MESSAGE"` dans le champ `capabilities` afin d'activer cette fonctionnalité pour un utilisateur connecté du chat.

Voici un exemple de détection d'une erreur de capacité lancée lors de la tentative de suppression d'un message sans les autorisations appropriées :

```
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)
}
```

Déconnecter un autre utilisateur

Utilisez `room.perform(request:)` pour déconnecter un autre utilisateur d'une salle de chat. Plus précisément, utilisez une instance de `DisconnectUserRequest`. Tous les `ChatMessages` reçus par une `ChatRoom` disposent d'une propriété `sender`, qui contient l'ID utilisateur que vous devez initialiser correctement avec une instance de `DisconnectUserRequest`. Vous pouvez éventuellement fournir une chaîne reason pour la demande de déconnexion.

```
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
   )
)
```

Comme il s'agit d'un autre exemple d'action d'un modérateur, vous pouvez tenter de déconnecter un autre utilisateur, mais vous en serez incapable si vous ne disposez pas de la capacité `DISCONNECT_USER`. La capacité est définie lorsque votre application backend appelle `CreateChatToken` et injecte la chaîne `"DISCONNECT_USER"` dans le champ `capabilities`.

Si votre utilisateur ne dispose pas de la capacité de déconnecter un autre utilisateur, `room.perform(request:)` lance une instance de `ChatError`, tout comme les autres demandes. Vous pouvez inspecter la propriété `errorCode` de l'erreur afin de déterminer si la demande a échoué en raison de l'absence de privilèges de modérateur :

```
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)
}
```

Kit SDK de messagerie client Amazon IVS Chat : didacticiel iOS

Le kit SDK de messagerie client Amazon Interactive Video (IVS) Chat pour iOS fournit des interfaces qui vous permettent d'intégrer notre API de messagerie IVS Chat (langue française non garantie) sur les plateformes utilisant le langage de programmation Swift d'Apple.

Pour un didacticiel sur le kit SDK Chat pour iOS, consultez https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios.

Kit SDK de messagerie client Amazon IVS Chat : guide JavaScript

Le kit SDK de messagerie client Amazon Interactive Video (IVS) Chat pour JavaScript vous permet d'intégrer notre API de messagerie Amazon IVS Chat sur les plateformes utilisant un navigateur Web.

Dernière version du kit SDK de messagerie client IVS Chat pour JavaScript : 1.0.2 (Notes de mise à jour)

Documentation de référence : pour plus d'informations sur les méthodes les plus importantes disponibles dans le kit SDK de messagerie client Amazon IVS Chat pour JavaScript, veuillez consulter la documentation de référence à l'adresse : https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/

Exemple de code : consultez l'exemple de référentiel sur GitHub, pour une démonstration spécifique au Web utilisant le SDK JavaScript : https://github.com/aws-samples/amazon-ivs-chat-web-demo

Démarrage

Avant de commencer, vous devez être familiarisé avec la Mise en route avec Chat Amazon IVS.

Ajouter le package

Utilisez soit :

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

ou :

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

React Native Support

Le kit SDK de messagerie du client IVS Chat pour JavaScript possède une dépendance `uuid` qui utilise la méthode `crypto.getRandomValues`. Comme cette méthode n'est pas prise en charge dans React Native, vous devez installer le polyfill supplémentaire `react-native-get-random-value` et l'importer en haut du fichier `index.js` :

```
import 'react-native-get-random-values';
import {AppRegistry} from 'react-native';
import App from './src/App';
import {name as appName} from './app.json';
```

```
AppRegistry.registerComponent(appName, () => App);
```

Configuration de votre backend

Cette intégration nécessite des points de terminaison sur votre serveur qui communiquent avec l'API Chat Amazon IVS. Utilisez les bibliothèques AWS officielles pour accéder à l'API Amazon IVS depuis votre serveur. Elles sont accessibles dans plusieurs langues depuis les packages publics, par exemple, node.js, java et go.

Créez un point de terminaison de serveur qui communique avec le point de terminaison de l'API Chat d'Amazon IVS CreateChatToken, afin de créer un jeton de chat pour les utilisateurs de chat.

Utilisation de l'SDK

Initialiser une instance de salle de chat

Créez une instance de la classe `ChatRoom`. Cela nécessite de passer `regionOrUrl` (la région AWS dans laquelle votre salle de chat est hébergée) et `tokenProvider` (la méthode de récupération des jetons sera créée à l'étape suivante) :

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

Fonction de fournisseur de jetons

Créez une fonction fournisseur de jetons asynchrone qui récupère un jeton de chat depuis votre backend :

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

La fonction ne doit accepter aucun paramètre et renvoyer une Promise (Promesse) contenant un objet jeton de chat :

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

Cette fonction est nécessaire pour initialiser l'objet ChatRoom. Ci-dessous, renseignez les champs `<token>` et `<date-time>` avec les valeurs reçues de votre backend :

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

N'oubliez pas de transmettre le `tokenProvider` au constructeur de ChatRoom. ChatRoom actualise le jeton lorsque la connexion est interrompue ou que la session expire. N'utilisez pas le `tokenProvider` pour stocker un jeton où que ce soit ; le ChatRoom le gère pour vous.

Recevoir des événements

Abonnez-vous ensuite aux événements de la salle de chat pour recevoir les événements du cycle de vie, ainsi que les messages et les événements diffusés dans le salle de chat :

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

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

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

/** Called when a chat message has been received. */
const unsubscribeOnMessageReceived = room.addListener('message', (message) => {
 /* Example message:
  * {
  *   id: "5OPsDdX18qcJ",
  *   sender: { userId: "user1" },
```

```
  *   content: "hello world",
  *   sendTime: new Date("2022-10-11T12:46:41.723Z"),
  *   requestId: "d1b511d8-d5ed-4346-b43f-49197c6e61de"
  * }
  */
});

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

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

/** Called when `aws:DISCONNECT_USER` system event has been received. */
const unsubscribeOnUserDisconnect = room.addListener('userDisconnect',
 (disconnectUserEvent) => {
 /* Example event payload:
  * {
  *   id: "AYk6xKitV4On",
  *   userId": "R1BLTDN84zEO",
  *   reason": "Spam",
  *   sendTime": new Date("2022-10-11T12:56:41.113Z"),
```

```
  *   requestId": "b379050a-2324-497b-9604-575cb5a9c5cd",
  *   attributes": { UserId: "R1BLTDN84zEO", Reason: "Spam" }
  * }
  */
});
```

Se connecter à une salle de chat

La dernière étape de l'initialisation de base consiste à se connecter à la salle de chat spécifique en établissant une connexion WebSocket. Pour ce faire, appelez la méthode `connect()` au sein de l'instance de la salle :

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

Le kit SDK tentera d'établir une connexion à une salle de chat codée dans le jeton de chat reçu de votre serveur.

Une fois que vous aurez appelé `connect()`, la salle passera à l'état `connecting` et émettra un événement `connecting`. Lorsque la salle se connecte avec succès, elle passe à l'état `connected` et émet un événement `connect`.

Un échec de connexion peut survenir en raison de problèmes lors de la récupération du jeton ou lors de la connexion à WebSocket. Dans ce cas, la salle essaie de se reconnecter automatiquement jusqu'au nombre de fois indiqué par le paramètre du constructeur `maxReconnectAttempts`. Lors des tentatives de reconnexion, la salle est à l'état `connecting` et n'émet aucun événement supplémentaire. Après avoir épuisé les tentatives de reconnexion, la pièce passe à l'état `disconnected` et émet un événement `disconnect` (avec une raison de déconnexion pertinente). Dans l'état `disconnected`, la salle n'essaie plus de se connecter ; vous devez appeler `connect()` à nouveau pour déclencher le processus de connexion.

Effectuer des actions dans une salle de chat

Le kit SDK de messagerie chat Amazon IVS fournit aux utilisateurs des actions permettant d'envoyer des messages, de supprimer des messages et de déconnecter les autres utilisateurs. Ils sont disponibles sur l'instance `ChatRoom`. Ils renvoient un objet `Promise` qui vous permet de recevoir une confirmation ou un rejet de la demande.

Envoi d'un message

Pour cette demande, la fonctionnalité `SEND_MESSAGE` doit être encodée dans votre jeton de chat.

Pour déclencher une demande send-message :

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

Pour obtenir une confirmation ou un rejet de la demande, `await` la promesse retournée ou utilisez la méthode `then()` :

```
try {
  const message = await room.sendMessage(request);
  // Message was successfully sent to chat room
} catch (error) {
  // Message request was rejected. Inspect the `error` parameter for details.
}
```

Supprimer un message

Pour cette demande, la fonctionnalité `DELETE_MESSAGE` doit être encodée dans votre jeton de chat.

Pour supprimer un message à des fins de modération, appelez la méthode `deleteMessage()` :

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

Pour obtenir une confirmation ou un rejet de la demande, `await` la promesse retournée ou utilisez la méthode `then()` :

```
try {
  const deleteMessageEvent = await room.deleteMessage(request);
  // Message was successfully deleted from chat room
} catch (error) {
  // Delete message request was rejected. Inspect the `error` parameter for details.
}
```

Déconnecter un autre utilisateur

Pour cette demande, la fonctionnalité `DISCONNECT_USER` doit être encodée dans votre jeton de chat.

Pour déconnecter un autre utilisateur à des fins de modération, appelez la méthode `disconnectUser()` :

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

Pour obtenir une confirmation ou un rejet de la demande, `await` la promesse retournée ou utilisez la méthode `then()` :

```
try {
  const disconnectUserEvent = await room.disconnectUser(request);
  // User was successfully disconnected from the chat room
} catch (error) {
  // Disconnect user request was rejected. Inspect the `error` parameter for details.
}
```

Se déconnecter d'une salle de chat

Pour fermer votre connexion à la salle de chat, appelez la méthode `disconnect()` sur l'instance `room` :

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

L'appel de cette méthode entraîne la fermeture ordonnée du WebSocket sous-jacent par la salle de manière ordonnée. L'instance de salle passe à un état `disconnected` et émet un événement de déconnexion, la raison `disconnect` étant définie sur `"clientDisconnect"`.

Kit SDK de messagerie client Amazon IVS Chat : didacticiel JavaScript, partie 1 : salles de chat

Il s'agit de la première partie d'un didacticiel en deux volets. Vous apprendrez les bases de l'utilisation du kit SDK de messagerie client Amazon IVS Chat pour JavaScript en créant une application entièrement fonctionnelle utilisant JavaScript/TypeScript. Nous appelons l'application Chatterbox.

Le public cible est constitué de développeurs expérimentés qui découvrent le kit SDK de messagerie Amazon IVS Chat. Vous devez être à l'aise avec le langage de programmation JavaScript/TypeScript et la bibliothèque React.

Par souci de concision, nous ferons référence au kit SDK de messagerie client Amazon IVS Chat pour JavaScript en le nommant kit SDK Chat JS.

Remarque : dans certains cas, les exemples de code pour JavaScript et TypeScript sont identiques et sont donc combinés.

Cette première partie du didacticiel est divisée en plusieurs sections :

1. the section called “Configurer un serveur d'authentification/d'autorisation local”
2. the section called “Créer un projet Chatterbox”
3. the section called “Se connecter à une salle de chat”
4. the section called “Créer un fournisseur de jetons”
5. the section called “Observer les mises à jour de la connexion”
6. the section called “Créer un composant de bouton d'envoi”
7. the section called “Créer une entrée de message”
8. the section called “Étapes suivantes”

Pour une documentation complète sur le kit SDK, commencez par le kit SDK de messagerie client Amazon IVS Chat (ici dans le Guide de l'utilisateur Chat Amazon IVS) et la Messagerie du client de chat : référence du kit SDK pour JavaScript (sur GitHub).

Prérequis

- Familiarisez-vous avec JavaScript/TypeScript et la bibliothèque React. Si vous ne connaissez pas React, découvrez les bases dans Introduction à React (langue française non garantie).
- Lisez et assurez-vous d'avoir compris Mise en route avec IVS Chat.
- Créez un utilisateur AWS IAM avec les fonctionnalités CreateChatToken et CreateRoom définies dans une politique IAM existante. (Consultez Mise en route avec IVS Chat.)
- Assurez-vous que les clés secrètes et d'accès de cet utilisateur sont stockées dans un fichier d'informations d'identification AWS. Pour obtenir des instructions, consultez le Guide de l'utilisateur de l'interface de ligne de commande AWS (en particulier les paramètres de configuration et de fichier d'informations d'identification).
- Créez une salle de chat et enregistrez son ARN. Consultez Mise en route avec IVS Chat. (Si vous n'enregistrez pas l'ARN, vous pourrez le consulter ultérieurement à l'aide de la console ou de l'API Chat.)
- Installez l'environnement Node.js 14+ avec le gestionnaire de packages NPM ou Yarn.

Configurer un serveur d'authentification/d'autorisation local

Votre application backend est chargée à la fois de créer des salles de chat et de générer les jetons de chat nécessaires au kit SDK Chat JS pour authentifier et autoriser vos clients à accéder à vos salles de chat. Vous devez utiliser votre propre backend, car vous ne pouvez pas stocker de manière sécurisée les clés AWS dans une application mobile ; des attaquants avertis pourraient les extraire et accéder à votre compte AWS.

Consultez la section Create a Chat Token (Créer un jeton de chat) dans Mise en route avec Amazon IVS Chat. Comme le montre l'organigramme, votre application côté serveur est chargée de créer un jeton de chat. Cela signifie que votre application doit fournir ses propres moyens de générer un jeton de chat en demandant un jeton à votre application côté serveur.

Dans cette section, vous apprendrez les bases de la création d'un fournisseur de jetons dans votre backend. Nous utilisons l'infrastructure express pour créer un serveur local en direct qui gère la création de jetons de chat à l'aide de votre environnement AWS local.

Créez un projet `npm` vide à l'aide de NPM. Créez un répertoire pour héberger votre application et faites-en votre répertoire de travail :

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

Utilisez `npm init` pour créer un fichier `package.json` pour votre application :

```
$ npm init
```

Cette commande vous invite à fournir plusieurs informations, notamment le nom et la version de votre application. Pour l'instant, appuyez simplement sur ENTRÉE pour accepter les valeurs par défaut pour la plupart d'entre elles, à l'exception de la suivante :

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

Appuyez sur ENTRÉE pour accepter le nom de fichier par défaut `index.js` suggéré ou saisissez le nom que vous souhaitez donner au fichier principal.

Installez maintenant les dépendances obligatoires :

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

`aws-sdk` nécessite des variables d'environnement de configuration, qui se chargent automatiquement à partir d'un fichier nommé `.env` situé dans le répertoire racine. Pour le configurer, créez un fichier nommé `.env` et renseignez les informations de configuration manquantes :

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

Créons maintenant un fichier de point d'entrée dans le répertoire racine avec le nom que vous avez saisi ci-dessus dans la commande `npm init`. Dans ce cas, nous utilisons `index.js` et importons tous les packages requis :

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

Créez maintenant une nouvelle instance de `express` :

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

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

Ensuite, vous pouvez créer votre première méthode POST de point de terminaison pour le fournisseur de jetons. Prenez les paramètres requis dans le corps de la demande (`roomId`, `userId`, `capabilities` et `sessionDurationInMinutes`) :

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

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

Ajoutez la validation des champs obligatoires :

```
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;
  }
});
```

Après avoir préparé la méthode POST, nous intégrons `createChatToken` avec `aws-sdk` pour la fonctionnalité de base d'authentification/autorisation :

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

À la fin du fichier, ajoutez un écouteur de port pour votre application `express` :

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

À présent, vous pouvez exécuter le serveur à l'aide de la commande suivante depuis la racine du projet :

```
$ node index.js
```

Conseil : ce serveur accepte les requêtes URL sur https://localhost:3000.

Créer un projet Chatterbox

Vous créez d'abord le projet React appelé `chatterbox`. Exécutez cette commande :

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

Vous pouvez intégrer le kit SDK de messagerie client Chat pour JS via Node Package Manager ou Yarn Package Manager :

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

Se connecter à une salle de chat

Ici, vous créez une `ChatRoom` et vous vous y connectez à l'aide de méthodes asynchrones. La classe `ChatRoom` gère la connexion de votre utilisateur au kit SDK Chat JS. Pour vous connecter correctement à une salle de chat, vous devez fournir une instance de `ChatToken` dans votre application React.

Accédez au fichier `App` créé dans le projet `chatterbox` par défaut et supprimez tout ce qui se trouve entre les deux balises `<div>`. Aucun code prérempli n'est nécessaire. À ce stade, notre `App` est assez vide.

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

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

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

Créez une instance `ChatRoom` et transmettez-la à l'état à l'aide du hook `useState`. Cela nécessite de passer `regionOrUrl` (la région AWS dans laquelle votre salle de chat est hébergée) et `tokenProvider` (utilisé pour le flux d'authentification/autorisation backend qui est créé dans les étapes suivantes).

Important : vous devez utiliser la même région AWS que celle dans laquelle vous avez créé la salle dans la Mise en route avec Amazon IVS Chat. L'API est un service régional AWS. Pour obtenir la liste des régions prises en charge et des points de terminaison du service HTTPS Amazon IVS Chat, consultez la page des régions Amazon IVS Chat.

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

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

export default function App() {
  const [room] = useState(() =>
    new ChatRoom({
      regionOrUrl: process.env.REGION as string,
      tokenProvider: () => {},
    }),
  );

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

Créer un fournisseur de jetons

À l'étape suivante, nous devons créer une fonction `tokenProvider` sans paramètre requise par le constructeur `ChatRoom`. Tout d'abord, nous allons créer une fonction `fetchChatToken` qui enverra une requête POST à l'application backend que vous avez configurée dans the section called “Configurer un serveur d'authentification/d'autorisation local”. Les jetons de chat contiennent les informations nécessaires au kit SDK pour établir avec succès une connexion à la salle de chat. L'API

de chat utilise ces jetons comme moyen sécurisé de valider l'identité d'un utilisateur, ses capacités au sein d'une salle de chat et la durée de la session.

Dans le navigateur de projet, créez un fichier TypeScript/JavaScript nommé `fetchChatToken`. Créez une demande de récupération à l'application `backend` et renvoyez l'objet `ChatToken` de la réponse. Ajoutez les propriétés du corps de la demande nécessaires à la création d'un jeton de chat. Utilisez les règles définies pour les Amazon Resource Name (ARN). Ces propriétés sont documentées dans le point de terminaison CreateChatToken.

Remarque : l'URL que vous utilisez ici est la même que celle que votre serveur local a créée lorsque vous avez exécuté l'application backend.

TypeScript

```
// fetchChatToken.ts

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

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

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

  const token = await response.json();
```

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

JavaScript

```
// fetchChatToken.js

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

  const token = await response.json();

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

Observer les mises à jour de la connexion

Réagir aux modifications de l'état de connexion d'une salle de chat est un élément essentiel de la création d'une application de chat. Commençons par nous abonner aux événements pertinents :

```
// 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>;
}
```

Ensuite, nous devons fournir la possibilité de lire l'état de la connexion. Nous utilisons notre hook `useState` pour créer un état local dans `App` et définir l'état de connexion dans chaque écouteur.

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>;
}
```

Après vous être abonné à l'état de connexion, affichez l'état de la connexion et connectez-vous à la salle de chat en utilisant la méthode `room.connect` indiquée dans le hook `useEffect` :

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

```
// ...

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

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

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

  room.connect();

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

// ...

return (
  <div>
    <h4>Connection State: {connectionState}</h4>
  </div>
);

// ...
```

Vous avez correctement mis en place une connexion à la salle de chat.

Créer un composant de bouton d'envoi

Dans cette section, vous créez un bouton d'envoi qui a une apparence différente pour chaque état de connexion. Le bouton d'envoi facilite l'envoi de messages dans une salle de chat. Il sert également d'indicateur visuel indiquant si et quand des messages peuvent être envoyés, par exemple en cas de perte de connexion ou de sessions de chat expirées.

Commencez par créer un fichier dans le répertoire `src` de votre projet Chatterbox et nommez-le `SendButton`. Ensuite, créez un composant qui affichera un bouton pour votre application de chat. Exportez votre `SendButton` et importez-le dans `App`. Dans le champ `<div></div>` vide, ajoutez `<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>
);
```

Ensuite, dans App, définissez une fonction nommée onMessageSend et transmettez-la à la propriété SendButton onPress. Définissez une autre variable nommée isSendDisabled (qui empêche l'envoi de messages lorsque la salle n'est pas connectée) et transmettez-la à la propriété SendButton disabled.

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

// ...

const onMessageSend = () => {};

const isSendDisabled = connectionState !== 'connected';

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

```
// ...
```

Créer une entrée de message

La barre de messages Chatterbox est le composant avec lequel vous allez interagir pour envoyer des messages à une salle de chat. Elle contient généralement une entrée de texte pour rédiger votre message et un bouton pour envoyer votre message.

Pour créer un composant `MessageInput`, créez d'abord un fichier dans le répertoire `src` et nommez-le `MessageInput`. Ensuite, créez un composant d'entrée contrôlé qui affichera une entrée pour votre application de chat. Exportez votre `MessageInput` et importez-le dans `App` (au-dessus du `<SendButton />`).

Créez un état nommé `messageToSend` à l'aide du hook `useState`, avec une chaîne vide comme valeur par défaut. Dans le corps de votre application, passez `messageToSend` à `value` de `MessageInput` et transmettez `setMessageToSend` à la propriété `onMessageChange` :

TypeScript

```
// MessageInput.tsx

import * as React from 'react';

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

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


// App.tsx

// ...

import { MessageInput } from './MessageInput';
```

```
// ...

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

// ...

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

JavaScript

```
// MessageInput.jsx

import * as React from 'react';

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

// App.jsx

// ...

import { MessageInput } from './MessageInput';

// ...

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

// ...


return (
```

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

Étapes suivantes

Maintenant que vous avez fini de créer une barre de messages pour Chatterbox, passez à la seconde partie de ce didacticiel JavaScript, Messages et événements.

Kit SDK de messagerie client Amazon IVS Chat : didacticiel JavaScript, partie 2 : messages et événements

Cette seconde et dernière partie du didacticiel est divisée en plusieurs sections :

1. the section called “S'abonner aux événements des messages de chat”
2. the section called “Afficher les messages reçus”
   a. the section called “Créer un composant de message”
   b. the section called “Reconnaître les messages envoyés par l'utilisateur actuel”
   c. the section called “Créer un composant de liste de messages”
   d. the section called “Afficher une liste de messages de chat”
3. the section called “Effectuer des actions dans une salle de chat”
   a. the section called “Envoi d'un message”
   b. the section called “Supprimer un message”
4. the section called “Étapes suivantes”

Remarque : dans certains cas, les exemples de code pour JavaScript et TypeScript sont identiques et sont donc combinés.

Pour une documentation complète sur le kit SDK, commencez par le kit SDK de messagerie client Amazon IVS Chat (ici dans le Guide de l'utilisateur Chat Amazon IVS) et la Messagerie du client de chat : référence du kit SDK pour JavaScript (sur GitHub).

Prérequis

Assurez-vous d'avoir terminé la première partie de ce didacticiel relative aux Salles de chat.

S'abonner aux événements des messages de chat

L'instance `ChatRoom` utilise des événements pour communiquer lorsque des événements se produisent dans une salle de chat. Pour commencer à mettre en œuvre l'expérience de chat, vous devez montrer à vos utilisateurs quand d'autres personnes envoient un message dans la salle à laquelle ils sont connectés.

Ici, vous vous abonnez aux événements des messages de chat. Plus tard, nous vous montrerons comment mettre à jour une liste de messages que vous créez, qui est mise à jour à chaque message/événement.

Dans votre `App`, dans le hook `useEffect`, abonnez-vous à tous les événements de message :

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

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

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

Afficher les messages reçus

La réception de messages est au cœur de l'expérience de chat. À l'aide du kit SDK Chat JS, vous pouvez configurer votre code pour recevoir facilement les événements des autres utilisateurs connectés à une salle de chat.

Plus tard, nous vous montrerons comment effectuer des actions dans une salle de chat qui tirent parti des composants que vous créez ici.

Dans votre `App`, définissez un état nommé `messages` avec un type de tableau `ChatMessage` nommé `messages` :

TypeScript

```
// App.tsx

// ...

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

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

  //...
}
```

JavaScript

```
// App.jsx

// ...

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

  //...
}
```

Ensuite, dans la fonction d'écouteur `message`, ajoutez `message` au tableau `messages` :

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

// ...

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

// ...
```

Ci-dessous, nous passons en revue les tâches pour afficher les messages reçus :

1. the section called “Créer un composant de message”

2. the section called “Reconnaître les messages envoyés par l'utilisateur actuel”
3. the section called “Créer un composant de liste de messages”
4. the section called “Afficher une liste de messages de chat”

Créer un composant de message

Le composant `Message` est chargé de rendre le contenu d'un message reçu par votre salle de chat. Dans cette section, vous créez un composant de messages pour afficher les messages de chat individuels dans l'App.

Dans le répertoire `src`, créez un fichier nommé `Message`. Transmettez le type `ChatMessage` de ce composant et transmettez la chaîne `content` provenant des propriétés `ChatMessage` pour afficher le texte du message reçu des écouteurs de messages de la salle de chat. Dans le navigateur de projets, accédez à `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>
  );
};
```

Conseil : utilisez ce composant pour stocker les différentes propriétés que vous souhaitez afficher dans les lignes de vos messages, par exemple, les URL des avatars, les noms d'utilisateur et les horodatages de l'envoi du message.

Reconnaître les messages envoyés par l'utilisateur actuel

Pour reconnaître le message envoyé par l'utilisateur actuel, nous modifions le code et créons un contexte React pour stocker le `userId` de l'utilisateur actuel.

Dans le répertoire `src`, créez un fichier nommé `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>;
};
```

Remarque : ici, nous avons utilisé le hook `useState` pour stocker la valeur `userId`. Dorénavant, vous pourrez utiliser `setUserId` pour modifier le contexte de l'utilisateur ou à des fins de connexion.

Ensuite, remplacez `userId` dans le premier paramètre transmis à `tokenProvider`, en utilisant le contexte créé précédemment :

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

  // ...
}
```

Dans votre composant `Message`, utilisez le `UserContext` créé auparavant, déclarez la variable `isMine`, associez le `userId` de l'expéditeur au `userId` du contexte et appliquez différents styles de messages à l'utilisateur actuel.

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>
  );
};
```

Créer un composant de liste de messages

Le composant `MessageList` est chargé d'afficher la conversation d'une salle de chat au fil du temps. Le fichier `MessageList` est le conteneur qui conserve tous nos messages. `Message` est une ligne dans `MessageList`.

Dans le répertoire `src`, créez un fichier nommé `MessageList`. Définissez `Props` avec des `messages` de type tableau `ChatMessage`. À l'intérieur du corps, mappez notre propriété `messages` et transmettez `Props` à votre composant `Message`.

TypeScript

```
// MessageList.tsx

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

interface Props {
  messages: ChatMessage[];
}

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

JavaScript

```
// MessageList.jsx

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

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

Afficher une liste de messages de chat

Ajoutez maintenant votre nouveau composant `MessageList` à votre composant principal `App` :

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

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

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

// ...
```

Toutes les pièces du puzzle sont maintenant en place pour que votre App commence à afficher les messages reçus par votre salle de chat. Continuez ci-dessous pour découvrir comment réaliser des actions dans une salle de chat qui tirent parti des composants que vous avez créés.

Effectuer des actions dans une salle de chat

L'envoi de messages et l'exécution d'actions de modérateur dans une salle de chat sont quelques-uns des principaux moyens d'interagir avec une salle de chat. Vous apprendrez ici comment utiliser divers objets `ChatRequest` pour effectuer des actions courantes dans Chatterbox, telles que l'envoi d'un message, la suppression d'un message et la déconnexion d'autres utilisateurs.

Toutes les actions d'une salle de chat suivent un schéma commun : à chaque action que vous effectuez dans une salle de chat, il existe un objet de demande correspondant. Pour chaque demande, il existe un objet de réponse correspondant que vous recevez lors de la confirmation de la demande.

Tant que vos utilisateurs disposent des autorisations appropriées lorsque vous créez un jeton de chat, ils peuvent effectuer avec succès la ou les actions correspondantes à l'aide des objets de demande pour voir quelles demandes vous pouvez effectuer dans une salle de chat.

Ci-dessous, nous expliquons comment envoyer un message et supprimer un message.

Envoi d'un message

La classe `SendMessageRequest` permet d'envoyer des messages dans une salle de chat. Ici, vous modifiez votre App pour envoyer une demande de message à l'aide du composant que vous avez créé dans Créer une entrée de message (dans la première partie de ce didacticiel).

Pour commencer, définissez une nouvelle propriété booléenne nommée `isSending` avec le hook `useState`. Utilisez cette nouvelle propriété pour activer l'état désactivé de votre élément HTML `button` à l'aide de la constante `isSendDisabled`. Dans le gestionnaire d'événements correspondant à votre `SendButton`, effacez la valeur de `messageToSend` et définissez `isSending` sur true (vrai).

Comme vous allez passer un appel d'API à partir de ce bouton, l'ajout du booléen *`isSending`* permet d'éviter que plusieurs appels d'API ne se produisent en même temps, en désactivant les interactions utilisateur sur votre *`SendButton`* jusqu'à ce que la demande soit complète.

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

// ...

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

// ...

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

// ...

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

// ...
```

Préparez la demande en créant une instance `SendMessageRequest` et en transmettant le contenu du message au constructeur. Après avoir défini les états `isSending` et `messageToSend`, appelez la méthode `sendMessage` qui envoie la demande à la salle de chat. Enfin, effacez l'indicateur `isSending` lors de la réception de la confirmation ou du rejet de la demande.

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);
  }
};

// ...
```

Essayez Chatterbox : essayez d'envoyer un message en rédigeant un message avec votre `MessageInput` et en appuyant sur votre `SendButton`. Vous devriez voir le message que vous avez envoyé s'afficher dans la `MessageList` que vous avez créée précédemment.

Supprimer un message

Pour supprimer un message d'une salle de chat, vous devez disposer de la capacité appropriée. Les capacités sont accordées lors de l'initialisation du jeton de chat que vous utilisez pour vous authentifier dans une salle de chat. Pour les besoins de cette section, le formulaire `ServerApp` de la section Configurer un serveur d'authentification/d'autorisation local (dans la partie 1 de ce didacticiel) vous permet de spécifier les capacités des modérateurs. Cela se fait dans votre application à l'aide de l'objet `tokenProvider` que vous avez créé dans la section Créer un fournisseur de jetons (également dans la partie 1).

Vous pouvez ici modifier votre `Message` en ajoutant une fonction pour supprimer le message.

Tout d'abord, ouvrez le `App.tsx` et ajoutez la fonctionnalité `DELETE_MESSAGE`. (`capabilities` est le deuxième paramètre de votre fonction `tokenProvider`.)

Remarque : c'est de cette manière que votre `ServerApp` informe les API IVS Chat que l'utilisateur associé au jeton de chat obtenu peut supprimer des messages dans une salle de chat. Dans une situation réelle, vous aurez probablement une logique backend plus complexe pour gérer les capacités des utilisateurs dans l'infrastructure de votre application serveur.

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']),
  }),
);

// ...
```

Au cours des étapes suivantes, vous mettrez à jour votre `Message` pour afficher un bouton de suppression.

Ouvrez `Message` et définissez un nouvel état booléen nommé `isDeleting` à l'aide du hook `useState` avec une valeur initiale de `false`. En utilisant cet état, mettez à jour le contenu de votre `Button` pour qu'il ait un aspect différent en fonction de l'état actuel de `isDeleting`. Désactivez votre bouton lorsque `isDeleting` est true (vrai) ; cela vous évite d'essayer de faire deux demandes de suppression de message simultanément.

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>
  );
};
```

Définissez une nouvelle fonction appelée `onDelete` qui accepte une chaîne comme paramètre et renvoie `Promise`. Dans le corps de la fermeture de l'action de votre `Button`, utilisez `setIsDeleting` pour faire basculer votre booléen `isDeleting` avant et après un appel à `onDelete`. Pour le paramètre de chaîne, transmettez l'ID du message de votre composant.

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>
  );
};
```

Ensuite, mettez à jour votre composant `MessageList` pour qu'il reflète les dernières modifications apportées à votre composant `Message`.

Ouvrez `MessageList` et définissez une nouvelle fonction nommée on`Delete` qui accepte une chaîne en tant que paramètre et renvoie `Promise`. Mettez à jour votre `Message` et transmettez-le via les propriétés de `Message`. Le paramètre de chaîne de votre nouvelle fermeture sera l'identifiant du message que vous souhaitez supprimer, qui sera transmis par votre `Message`.

TypeScript

```
// MessageList.tsx
```

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

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

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

JavaScript

```
// MessageList.jsx

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

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

Ensuite, vous mettez à jour votre App pour refléter les dernières modifications apportées à votre `MessageList`.

Dans App, définissez une fonction nommée onDeleteMessage et transmettez-la à la propriété MessageList onDelete :

TypeScript

```
// App.tsx

// ...

const onDeleteMessage = async (id: string) => {};

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

// ...
```

JavaScript

```
// App.jsx

// ...

const onDeleteMessage = async (id) => {};

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

```
// ...
```

Préparez une demande en créant une instance de `DeleteMessageRequest`, en transmettant l'ID de message correspondant au paramètre du constructeur et en appelant `deleteMessage` qui accepte la demande préparée ci-dessus :

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);
};

// ...
```

Ensuite, vous mettez à jour votre état `messages` pour refléter une nouvelle liste de messages qui omet le message que vous venez de supprimer.

Dans le hook `useEffect`, écoutez l'événement `messageDelete` et mettez à jour votre tableau d'états `messages` en supprimant le message dont l'ID correspond au paramètre `message`.

Remarque : l'événement `messageDelete` peut être déclenché en cas de suppression de messages par l'utilisateur actuel ou par tout autre utilisateur présent dans la salle. Le gérer dans le gestionnaire

d'événements (plutôt qu'à côté de la demande `deleteMessage`) vous permet d'unifier la gestion des messages de suppression.

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

Vous pouvez désormais supprimer des utilisateurs d'une salle de chat dans votre application de chat.

Étapes suivantes

À titre expérimental, essayez de mettre en œuvre d'autres actions dans une salle, par exemple la déconnexion d'un autre utilisateur.

Kit SDK de messagerie client de chat Amazon IVS : didacticiel React Native, partie 1 : salles de chat

Il s'agit de la première partie d'un didacticiel en deux volets. Vous apprendrez les bases de l'utilisation du kit SDK de messagerie client de chat Amazon IVS pour JavaScript en créant une application entièrement fonctionnelle utilisant React Native. Nous appelons l'application Chatterbox.

Le public cible est constitué de développeurs expérimentés qui découvrent le kit SDK de messagerie Amazon IVS Chat. Vous devez être à l'aise avec les langages de programmation TypeScript ou JavaScript et la bibliothèque React Native.

Par souci de concision, nous ferons référence au kit SDK de messagerie client Amazon IVS Chat pour JavaScript en le nommant kit SDK Chat JS.

Remarque : dans certains cas, les exemples de code pour JavaScript et TypeScript sont identiques et sont donc combinés.

Cette première partie du didacticiel est divisée en plusieurs sections :

1. the section called “Configurer un serveur d'authentification/d'autorisation local”
2. the section called “Créer un projet Chatterbox”
3. the section called “Se connecter à une salle de chat”
4. the section called “Créer un fournisseur de jetons”
5. the section called “Observer les mises à jour de la connexion”
6. the section called “Créer un composant de bouton d'envoi”
7. the section called “Créer une entrée de message”
8. the section called “Étapes suivantes”

Prérequis

- Familiarisez-vous avec TypeScript ou JavaScript et la bibliothèque React Native. Si vous ne connaissez pas React Native, découvrez les bases dans Introduction à React Native (langue française non garantie).
- Lisez et assurez-vous d'avoir compris Mise en route avec IVS Chat.
- Créez un utilisateur AWS IAM avec les fonctionnalités CreateChatToken et CreateRoom définies dans une politique IAM existante. (Consultez Mise en route avec IVS Chat.)
- Assurez-vous que les clés secrètes et d'accès de cet utilisateur sont stockées dans un fichier d'informations d'identification AWS. Pour obtenir des instructions, consultez le Guide de l'utilisateur de l'interface de ligne de commande AWS (en particulier les paramètres de configuration et de fichier d'informations d'identification).
- Créez une salle de chat et enregistrez son ARN. Consultez Mise en route avec IVS Chat. (Si vous n'enregistrez pas l'ARN, vous pourrez le consulter ultérieurement à l'aide de la console ou de l'API Chat.)
- Installez l'environnement Node.js 14+ avec le gestionnaire de packages NPM ou Yarn.

Configurer un serveur d'authentification/d'autorisation local

Votre application backend est chargée à la fois de créer des salles de chat et de générer les jetons de chat nécessaires au kit SDK Chat JS pour authentifier et autoriser vos clients à accéder à vos salles de chat. Vous devez utiliser votre propre backend, car vous ne pouvez pas stocker de manière sécurisée les clés AWS dans une application mobile ; des attaquants avertis pourraient les extraire et accéder à votre compte AWS.

Consultez la section Create a Chat Token (Créer un jeton de chat) dans Mise en route avec Amazon IVS Chat. Comme le montre l'organigramme, votre application côté serveur est chargée de créer un jeton de chat. Cela signifie que votre application doit fournir ses propres moyens de générer un jeton de chat en demandant un jeton à votre application côté serveur.

Dans cette section, vous apprendrez les bases de la création d'un fournisseur de jetons dans votre backend. Nous utilisons l'infrastructure express pour créer un serveur local en direct qui gère la création de jetons de chat à l'aide de votre environnement AWS local.

Créez un projet `npm` vide à l'aide de NPM. Créez un répertoire pour héberger votre application et faites-en votre répertoire de travail :

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

Utilisez `npm init` pour créer un fichier `package.json` pour votre application :

```
$ npm init
```

Cette commande vous invite à fournir plusieurs informations, notamment le nom et la version de votre application. Pour l'instant, appuyez simplement sur ENTRÉE pour accepter les valeurs par défaut pour la plupart d'entre elles, à l'exception de la suivante :

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

Appuyez sur ENTRÉE pour accepter le nom de fichier par défaut `index.js` suggéré ou saisissez le nom que vous souhaitez donner au fichier principal.

Installez maintenant les dépendances obligatoires :

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

`aws-sdk` nécessite des variables d'environnement de configuration, qui se chargent automatiquement à partir d'un fichier nommé `.env` situé dans le répertoire racine. Pour le configurer, créez un fichier nommé `.env` et renseignez les informations de configuration manquantes :

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

Créons maintenant un fichier de point d'entrée dans le répertoire racine avec le nom que vous avez saisi ci-dessus dans la commande `npm init`. Dans ce cas, nous utilisons `index.js` et importons tous les packages requis :

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

Créez maintenant une nouvelle instance de `express` :

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

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

Ensuite, vous pouvez créer votre première méthode POST de point de terminaison pour le fournisseur de jetons. Prenez les paramètres requis dans le corps de la demande (`roomId`, `userId`, `capabilities` et `sessionDurationInMinutes`) :

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

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

Ajoutez la validation des champs obligatoires :

```
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;
  }
});
```

Après avoir préparé la méthode POST, nous intégrons `createChatToken` avec `aws-sdk` pour la fonctionnalité de base d'authentification/autorisation :

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

À la fin du fichier, ajoutez un écouteur de port pour votre application `express` :

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

À présent, vous pouvez exécuter le serveur à l'aide de la commande suivante depuis la racine du projet :

```
$ node index.js
```

Conseil : ce serveur accepte les requêtes URL sur https://localhost:3000.

Créer un projet Chatterbox

Vous créez d'abord le projet React Native appelé `chatterbox`. Exécutez cette commande :

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

Ou vous créez un projet d'exposition avec un modèle TypeScript.

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

Vous pouvez intégrer le kit SDK de messagerie client Chat pour JS via Node Package Manager ou Yarn Package Manager :

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

Se connecter à une salle de chat

Ici, vous créez une `ChatRoom` et vous vous y connectez à l'aide de méthodes asynchrones. La classe `ChatRoom` gère la connexion de votre utilisateur au kit SDK Chat JS. Pour vous connecter correctement à une salle de chat, vous devez fournir une instance de `ChatToken` dans votre application React.

Accédez au fichier App créé dans le projet `chatterbox` par défaut et supprimez tout ce qui est renvoyé par un composant fonctionnel. Aucun code prérempli n'est nécessaire. À ce stade, notre App est assez vide.

TypeScript/JavaScript :

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

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

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

Créez une instance `ChatRoom` et transmettez-la à l'état à l'aide du hook `useState`. Cela nécessite de passer `regionOrUrl` (la région AWS dans laquelle votre salle de chat est hébergée) et `tokenProvider` (utilisé pour le flux d'authentification/autorisation backend qui est créé dans les étapes suivantes).

Important : vous devez utiliser la même région AWS que celle dans laquelle vous avez créé la salle dans la Mise en route avec Amazon IVS Chat. L'API est un service régional AWS. Pour obtenir la liste des régions prises en charge et des points de terminaison du service HTTPS Amazon IVS Chat, consultez la page des régions Amazon IVS Chat.

TypeScript/JavaScript :

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

import React, { useState } from 'react';
import { Text } from 'react-native';
import { ChatRoom } from 'amazon-ivs-chat-messaging';

export default function App() {
  const [room] = useState(() =>
    new ChatRoom({
      regionOrUrl: process.env.REGION,
      tokenProvider: () => {},
    }),
  );

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

Créer un fournisseur de jetons

À l'étape suivante, nous devons créer une fonction `tokenProvider` sans paramètre requise par le constructeur `ChatRoom`. Tout d'abord, nous allons créer une fonction `fetchChatToken` qui enverra une requête POST à l'application backend que vous avez configurée dans the section called “Configurer un serveur d'authentification/d'autorisation local”. Les jetons de chat contiennent les informations nécessaires au kit SDK pour établir avec succès une connexion à la salle de chat. L'API de chat utilise ces jetons comme moyen sécurisé de valider l'identité d'un utilisateur, ses capacités au sein d'une salle de chat et la durée de la session.

Dans le navigateur de projet, créez un fichier TypeScript/JavaScript nommé `fetchChatToken`. Créez une demande de récupération à l'application `backend` et renvoyez l'objet `ChatToken` de la réponse. Ajoutez les propriétés du corps de la demande nécessaires à la création d'un jeton de chat. Utilisez les règles définies pour les Amazon Resource Name (ARN). Ces propriétés sont documentées dans le point de terminaison CreateChatToken.

Remarque : l'URL que vous utilisez ici est la même que celle que votre serveur local a créée lorsque vous avez exécuté l'application backend.

TypeScript

```
// fetchChatToken.ts

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

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

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

```
      userId,
      roomIdentifier: process.env.ROOM_ID,
      capabilities,
      sessionDurationInMinutes,
      attributes
    }),
  });

  const token = await response.json();

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

JavaScript

```
// fetchChatToken.js

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

  const token = await response.json();
```

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

Observer les mises à jour de la connexion

Réagir aux modifications de l'état de connexion d'une salle de chat est un élément essentiel de la création d'une application de chat. Commençons par nous abonner aux événements pertinents :

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>;
}
```

Ensuite, nous devons fournir la possibilité de lire l'état de la connexion. Nous utilisons notre hook `useState` pour créer un état local dans `App` et définir l'état de connexion dans chaque écouteur.

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>;
}
```

Après vous être abonné à l'état de connexion, affichez l'état de la connexion et connectez-vous à la salle de chat en utilisant la méthode `room.connect` indiquée dans le hook `useEffect` :

TypeScript/JavaScript :

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

// ...

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

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

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

  room.connect();

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

// ...

return (
  <SafeAreaView style={styles.root}>
```

```
    <Text>Connection State: {connectionState}</Text>
  </SafeAreaView>
);

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

// ...
```

Vous avez correctement mis en place une connexion à la salle de chat.

Créer un composant de bouton d'envoi

Dans cette section, vous créez un bouton d'envoi qui a une apparence différente pour chaque état de connexion. Le bouton d'envoi facilite l'envoi de messages dans une salle de chat. Il sert également d'indicateur visuel indiquant si et quand des messages peuvent être envoyés, par exemple en cas de perte de connexion ou de sessions de chat expirées.

Commencez par créer un fichier dans le répertoire `src` de votre projet Chatterbox et nommez-le `SendButton`. Ensuite, créez un composant qui affichera un bouton pour votre application de chat. Exportez votre `SendButton` et importez-le dans `App`. Dans le champ `<View></View>` vide, ajoutez `<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>
);
```

Ensuite, dans App, définissez une fonction nommée onMessageSend et transmettez-la à la propriété SendButton onPress. Définissez une autre variable nommée isSendDisabled (qui empêche l'envoi de messages lorsque la salle n'est pas connectée) et transmettez-la à la propriété SendButton disabled.

TypeScript/JavaScript :

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

// ...

const onMessageSend = () => {};

const isSendDisabled = connectionState !== 'connected';
```

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

// ...
```

Créer une entrée de message

La barre de messages Chatterbox est le composant avec lequel vous allez interagir pour envoyer des messages à une salle de chat. Elle contient généralement une entrée de texte pour rédiger votre message et un bouton pour envoyer votre message.

Pour créer un composant `MessageInput`, créez d'abord un fichier dans le répertoire `src` et nommez-le `MessageInput`. Ensuite, créez un composant d'entrée qui affichera une entrée pour votre application de chat. Exportez votre `MessageInput` et importez-le dans `App` (au-dessus du `<SendButton />`).

Créez un état nommé `messageToSend` à l'aide du hook `useState`, avec une chaîne vide comme valeur par défaut. Dans le corps de votre application, passez `messageToSend` à `value` de `MessageInput` et transmettez `setMessageToSend` à la propriété `onMessageChange` :

TypeScript

```
// MessageInput.tsx

import * as React from 'react';

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

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

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

// App.tsx

// ...

import { MessageInput } from './MessageInput';

// ...

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

// ...

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

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

```
    backgroundColor: 'white',
  }
});
```

JavaScript

```
// MessageInput.jsx

import * as React from 'react';

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

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

// App.jsx

// ...

import { MessageInput } from './MessageInput';

// ...

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

// ...

return (
  <SafeAreaView style={styles.root}>
```

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

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

Étapes suivantes

Maintenant que vous avez fini de créer une barre de messages pour Chatterbox, passez à la seconde partie de ce didacticiel React Native, Messages et événements.

Kit SDK de messagerie client de chat Amazon IVS : didacticiel React Native, partie 2 : messages et événements

Cette seconde et dernière partie du didacticiel est divisée en plusieurs sections :

1. the section called “S'abonner aux événements des messages de chat”
2. the section called “Afficher les messages reçus”
   a. the section called “Créer un composant de message”
   b. the section called “Reconnaître les messages envoyés par l'utilisateur actuel”
   c. the section called “Afficher une liste de messages de chat”
3. the section called “Effectuer des actions dans une salle de chat”

   a. the section called “Envoi d'un message”
   b. the section called “Supprimer un message”
4. the section called “Étapes suivantes”

Remarque : dans certains cas, les exemples de code pour JavaScript et TypeScript sont identiques et sont donc combinés.

Prérequis

Assurez-vous d'avoir terminé la première partie de ce didacticiel relative aux Salles de chat.

S'abonner aux événements des messages de chat

L'instance `ChatRoom` utilise des événements pour communiquer lorsque des événements se produisent dans une salle de chat. Pour commencer à mettre en œuvre l'expérience de chat, vous devez montrer à vos utilisateurs quand d'autres personnes envoient un message dans la salle à laquelle ils sont connectés.

Ici, vous vous abonnez aux événements des messages de chat. Plus tard, nous vous montrerons comment mettre à jour une liste de messages que vous créez, qui est mise à jour à chaque message/événement.

Dans votre `App`, dans le hook `useEffect`, abonnez-vous à tous les événements de message :

TypeScript/JavaScript :

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

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

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

Afficher les messages reçus

La réception de messages est au cœur de l'expérience de chat. À l'aide du kit SDK Chat JS, vous pouvez configurer votre code pour recevoir facilement les événements des autres utilisateurs connectés à une salle de chat.

Plus tard, nous vous montrerons comment effectuer des actions dans une salle de chat qui tirent parti des composants que vous créez ici.

Dans votre App, définissez un état nommé `messages` avec un type de tableau `ChatMessage` nommé `messages` :

TypeScript

```
// App.tsx

// ...

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

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

  //...
}
```

JavaScript

```
// App.jsx

// ...

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

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

  //...
}
```

Ensuite, dans la fonction d'écouteur `message`, ajoutez `message` au tableau `messages` :

TypeScript/JavaScript :

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

// ...

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

// ...
```

Ci-dessous, nous passons en revue les tâches pour afficher les messages reçus :

1. the section called “Créer un composant de message”
2. the section called “Reconnaître les messages envoyés par l'utilisateur actuel”
3. the section called “Afficher une liste de messages de chat”

Créer un composant de message

Le composant `Message` est chargé de rendre le contenu d'un message reçu par votre salle de chat. Dans cette section, vous créez un composant de messages pour afficher les messages de chat individuels dans l'App.

Dans le répertoire `src`, créez un fichier nommé `Message`. Transmettez le type `ChatMessage` de ce composant et transmettez la chaîne `content` provenant des propriétés `ChatMessage` pour afficher le texte du message reçu des écouteurs de messages de la salle de chat. Dans le navigateur de projets, accédez à `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,
  },
});
```

Conseil : utilisez ce composant pour stocker les différentes propriétés que vous souhaitez afficher dans les lignes de vos messages, par exemple, les URL des avatars, les noms d'utilisateur et les horodatages de l'envoi du message.

Reconnaître les messages envoyés par l'utilisateur actuel

Pour reconnaître le message envoyé par l'utilisateur actuel, nous modifions le code et créons un contexte React pour stocker le `userId` de l'utilisateur actuel.

Dans le répertoire `src`, créez un fichier nommé `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;
```

Remarque : ici, nous avons utilisé le hook `useState` pour stocker la valeur `userId`. Dorénavant, vous pourrez utiliser `setUserId` pour modifier le contexte de l'utilisateur ou à des fins de connexion.

Ensuite, remplacez `userId` dans le premier paramètre transmis à `tokenProvider`, en utilisant le contexte créé précédemment. Assurez-vous d'ajouter la capacité `SEND_MESSAGE` à votre fournisseur de jetons, comme indiqué ci-dessous ; elle est requise pour envoyer des messages.

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']),
      }),
  );

  // ...
}
```

Dans votre composant `Message`, utilisez le `UserContext` créé auparavant, déclarez la variable `isMine`, associez le `userId` de l'expéditeur au `userId` du contexte et appliquez différents styles de messages à l'utilisateur actuel.

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',
  },
});
```

Afficher une liste de messages de chat

Maintenant, répertoriez les messages à l'aide des composants `FlatList` et `Message` :

TypeScript

```
// App.tsx

// ...

const renderItem = useCallback<ListRenderItem<ChatMessage>>(({ item }) => {
  return (
    <Message key={item.id} message={item} />
  );
}, []);

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

// ...
```

JavaScript

```
// App.jsx

// ...

const renderItem = useCallback(({ item }) => {
  return (
    <Message key={item.id} message={item} />
  );
}, []);

return (
  <SafeAreaView style={styles.root}>
```

```
    <Text>Connection State: {connectionState}</Text>
    <FlatList inverted data={messages} renderItem={renderItem} />
    <View style={styles.messageBar}>
      <MessageInput value={messageToSend} onMessageChange={setMessageToSend} />
      <SendButton disabled={isSendDisabled} onPress={onMessageSend} />
    </View>
  </SafeAreaView>
);

// ...
```

Toutes les pièces du puzzle sont maintenant en place pour que votre App commence à afficher les messages reçus par votre salle de chat. Continuez ci-dessous pour découvrir comment réaliser des actions dans une salle de chat qui tirent parti des composants que vous avez créés.

Effectuer des actions dans une salle de chat

L'envoi de messages et l'exécution d'actions de modérateur sont quelques-uns des principaux moyens d'interagir avec une salle de chat. Vous apprendrez ici comment utiliser divers objets de demande de chat pour effectuer des actions courantes dans Chatterbox, telles que l'envoi d'un message, la suppression d'un message et la déconnexion d'autres utilisateurs.

Toutes les actions d'une salle de chat suivent un schéma commun : à chaque action que vous effectuez dans une salle de chat, il existe un objet de demande correspondant. Pour chaque demande, il existe un objet de réponse correspondant que vous recevez lors de la confirmation de la demande.

Tant que vos utilisateurs disposent des capacités appropriées lorsque vous créez un jeton de chat, ils peuvent effectuer avec succès la ou les actions correspondantes à l'aide des objets de demande pour voir quelles demandes vous pouvez effectuer dans une salle de chat.

Ci-dessous, nous expliquons comment envoyer un message et supprimer un message.

Envoi d'un message

La classe `SendMessageRequest` permet d'envoyer des messages dans une salle de chat. Ici, vous modifiez votre App pour envoyer une demande de message à l'aide du composant que vous avez créé dans Créer une entrée de message (dans la première partie de ce didacticiel).

Pour commencer, définissez une nouvelle propriété booléenne nommée `isSending` avec le hook `useState`. Utilisez cette nouvelle propriété pour activer l'état désactivé de votre élément `button` à

l'aide de la constante `isSendDisabled`. Dans le gestionnaire d'événements correspondant à votre `SendButton`, effacez la valeur de `messageToSend` et définissez `isSending` sur true (vrai).

Comme vous allez passer un appel d'API à partir de ce bouton, l'ajout du booléen *`isSending`* permet d'éviter que plusieurs appels d'API ne se produisent en même temps, en désactivant les interactions utilisateur sur votre *`SendButton`* jusqu'à ce que la demande soit complète.

Remarque : l'envoi de messages ne fonctionne que si vous avez ajouté la capacité `SEND_MESSAGE` à votre fournisseur de jetons, comme indiqué ci-dessus dans la rubrique Reconnaître les messages envoyés par l'utilisateur actuel.

TypeScript/JavaScript :

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

// ...

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

// ...

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

// ...

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

// ...
```

Préparez la demande en créant une instance `SendMessageRequest` et en transmettant le contenu du message au constructeur. Après avoir défini les états `isSending` et `messageToSend`, appelez la méthode `sendMessage` qui envoie la demande à la salle de chat. Enfin, effacez l'indicateur `isSending` lors de la réception de la confirmation ou du rejet de la demande.

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);
  }
};

// ...
```

Essayez Chatterbox : essayez d'envoyer un message en rédigeant un message avec votre `MessageBar` et en appuyant sur votre `SendButton`. Vous devriez voir le message que vous avez envoyé s'afficher dans la `MessageList` que vous avez créée précédemment.

Supprimer un message

Pour supprimer un message d'une salle de chat, vous devez disposer de la capacité appropriée. Les capacités sont accordées lors de l'initialisation du jeton de chat que vous utilisez pour vous authentifier dans une salle de chat. Pour les besoins de cette section, le formulaire `ServerApp` de la section Configurer un serveur d'authentification/d'autorisation local (dans la partie 1 de ce didacticiel) vous permet de spécifier les capacités des modérateurs. Cela se fait dans votre application à l'aide de l'objet `tokenProvider` que vous avez créé dans la section Créer un fournisseur de jetons (également dans la partie 1).

Vous pouvez ici modifier votre `Message` en ajoutant une fonction pour supprimer le message.

Tout d'abord, ouvrez le `App.tsx` et ajoutez la fonctionnalité `DELETE_MESSAGE`. (`capabilities` est le deuxième paramètre de votre fonction `tokenProvider`.)

Remarque : c'est de cette manière que votre `ServerApp` informe les API IVS Chat que l'utilisateur associé au jeton de chat obtenu peut supprimer des messages dans une salle de chat. Dans

une situation réelle, vous aurez probablement une logique backend plus complexe pour gérer les capacités des utilisateurs dans l'infrastructure de votre application serveur.

TypeScript/JavaScript :

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

// ...

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

// ...
```

Au cours des étapes suivantes, vous mettrez à jour votre `Message` pour afficher un bouton de suppression.

Définissez une nouvelle fonction appelée `onDelete` qui accepte une chaîne comme paramètre et renvoie `Promise`. Pour le paramètre de chaîne, transmettez l'ID du message de votre composant.

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',
  },
});
```

Ensuite, mettez à jour votre composant `renderItem` pour qu'il reflète les dernières modifications apportées à votre composant `FlatList`.

Dans App, définissez une fonction nommée `handleDeleteMessage` et transmettez-la à la propriété `MessageList onDelete` :

TypeScript

```
// App.tsx

// ...

const handleDeleteMessage = async (id: string) => {};

const renderItem = useCallback<ListRenderItem<ChatMessage>>(({ item }) => {
  return (
    <Message key={item.id} message={item} onDelete={handleDeleteMessage} />
  );
}, [handleDeleteMessage]);

// ...
```

JavaScript

```
// App.jsx

// ...

const handleDeleteMessage = async (id) => {};

const renderItem = useCallback(({ item }) => {
  return (
    <Message key={item.id} message={item} onDelete={handleDeleteMessage} />
  );
}, [handleDeleteMessage]);
```

```
// ...
```

Préparez une demande en créant une instance de `DeleteMessageRequest`, en transmettant l'ID de message correspondant au paramètre du constructeur et en appelant `deleteMessage` qui accepte la demande préparée ci-dessus :

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);
};

// ...
```

Ensuite, vous mettez à jour votre état `messages` pour refléter une nouvelle liste de messages qui omet le message que vous venez de supprimer.

Dans le hook `useEffect`, écoutez l'événement `messageDelete` et mettez à jour votre tableau d'états `messages` en supprimant le message dont l'ID correspond au paramètre `message`.

Remarque : l'événement `messageDelete` peut être déclenché en cas de suppression de messages par l'utilisateur actuel ou par tout autre utilisateur présent dans la salle. Le gérer dans le gestionnaire

d'événements (plutôt qu'à côté de la demande `deleteMessage`) vous permet d'unifier la gestion des messages de suppression.

TypeScript/JavaScript :

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

// ...

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

return () => {
  // ...

  unsubscribeOnMessageDeleted();
};

// ...
```

Vous pouvez désormais supprimer des utilisateurs d'une salle de chat dans votre application de chat.

Étapes suivantes

À titre expérimental, essayez de mettre en œuvre d'autres actions dans une salle, par exemple la déconnexion d'un autre utilisateur.

Kit SDK de messagerie client de chat Amazon IVS : bonnes pratiques React et React Native

Ce document décrit les pratiques les plus importantes relatives à l'utilisation du kit SDK de messagerie de chat Amazon IVS pour React et React Native. Ces informations devraient vous permettre de créer des fonctionnalités de chat classiques dans une application React et vous fournir les informations de base dont vous avez besoin pour approfondir les parties les plus avancées du kit SDK de messagerie Chat IVS.

Création d'un hook d'initialisation ChatRoom

La classe `ChatRoom` contient les méthodes de chat de base et les écouteurs permettant de gérer l'état de la connexion et d'écouter les événements tels que les messages reçus et supprimés. Ici, nous montrons comment stocker correctement les instances de chat dans un hook.

Mise en œuvre

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 };
};
```

Remarque : nous n'utilisons pas la méthode `dispatch` à partir du hook `setState`, car vous ne pouvez pas mettre à jour les paramètres de configuration à la volée. Le kit SDK crée une instance une seule fois et il n'est pas possible de mettre à jour le fournisseur de jetons.

Important : utilisez le hook d'initialisation `ChatRoom` une seule fois pour initialiser une nouvelle instance de salle de chat.

Exemple

TypeScript/JavaScript :

```
// ...

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

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

  // ...
};

// ...
```

Écoute de l'état de la connexion

Vous pouvez éventuellement vous abonner aux mises à jour de l'état de connexion dans votre hook de salle de chat.

Mise en œuvre

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 };
};
```

Fournisseur d'instances ChatRoom

Pour utiliser le hook dans d'autres composants (afin d'éviter le prop drilling), vous pouvez créer un fournisseur de salle de chat à l'aide de `context` de React.

Mise en œuvre

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;
```

Exemple

Après la création de `ChatRoomProvider`, vous pouvez utiliser votre instance avec `useChatRoomContext`.

Important : placez le fournisseur au niveau racine uniquement si vous avez besoin d'accéder au `context` situé entre l'écran de chat et les autres composants au milieu, afin d'éviter des rendus inutiles si vous êtes à l'écoute des connexions. Sinon, placez le fournisseur le plus près possible de l'écran de chat.

TypeScript/JavaScript :

```
// AppContainer

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

  return (
```

```
    <ChatRoomProvider value={room}>
      <MyChatScreen />
    </ChatRoomProvider>
  );
};

// MyChatScreen

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

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

// ...
```

Création d'un écouteur de messages

Pour rester au courant de tous les messages entrants, vous devez vous abonner aux événements `message` et `deleteMessage`. Voici un exemple de code qui fournit des messages de chat pour vos composants.

Important : pour des raisons de performances, nous séparons `ChatMessageContext` de `ChatRoomProvider`, car nous pouvons obtenir de nombreux nouveaux rendus lorsque l'écouteur des messages de chat met à jour l'état de son message. N'oubliez pas d'appliquer `ChatMessageContext` dans les composants où vous utiliserez `ChatMessageProvider`.

Mise en œuvre

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>;
};
```

Exemple dans React

Important : n'oubliez pas d'envelopper le conteneur de votre message avec `ChatMessagesProvider`. La ligne `Message` est un exemple de composant qui affiche le contenu d'un message.

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>
  );
};
```

Exemple dans React Native

Par défaut, `ChatMessage` contient `id`, qui est utilisé automatiquement comme clés React dans `FlatList` pour chaque ligne ; vous n'avez donc pas besoin de transmettre `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} />;
};
```

Plusieurs instances de salle de chat dans une application

Si vous utilisez plusieurs salles de chat simultanées dans votre application, nous vous proposons de créer chaque fournisseur pour chaque chat et de l'utiliser dans le fournisseur de chat. Dans cet exemple, nous créons un chat de bot d'assistance et d'aide client. Nous créons un fournisseur pour les deux.

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>;
};
```

Exemple dans React

Vous pouvez désormais utiliser différents fournisseurs de chat qui utilisent le même `ChatRoomProvider`. Plus tard, vous pourrez réutiliser le même `useChatRoomContext` dans chaque écran/affichage.

TypeScript/JavaScript :

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

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

```
          </SalesChatProvider>
        }
      />
    </Routes>
  );
};
```

Exemple dans React Native

TypeScript/JavaScript :

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

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

TypeScript/JavaScript :

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

// ...

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

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

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

// SalesChatScreen.tsx / SalesChatScreen.jsx

// ...

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

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

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

Sécurité Chat Amazon IVS

Chez AWS, la sécurité dans le cloud est notre priorité numéro 1. En tant que client AWS, vous bénéficiez d'un centre de données et d'une architecture réseau conçus pour répondre aux exigences des organisations les plus pointilleuses en termes de sécurité.

La sécurité est une responsabilité partagée entre AWS et vous-même. Le modèle de responsabilité partagée décrit cette notion par les termes sécurité du cloud et sécurité dans le cloud :

- Sécurité du cloud : AWS est responsable de la protection de l'infrastructure qui exécute des services AWS dans le Cloud AWS. AWS vous fournit également les services que vous pouvez utiliser en toute sécurité. Des auditeurs tiers testent et vérifient régulièrement l'efficacité de notre sécurité dans le cadre des programmes de conformité AWS.
- Sécurité dans le cloud : votre responsabilité est déterminée par le service AWS que vous utilisez. Vous êtes également responsable d'autres facteurs, y compris la sensibilité de vos données, les exigences de votre organisation ainsi que les lois et réglementations applicables.

Cette documentation vous aide à comprendre comment appliquer le modèle de responsabilité partagée lorsque vous utilisez Chat Amazon IVS. Les rubriques suivantes vous montrent comment configurer Chat Amazon IVS pour répondre à vos objectifs de sécurité et de conformité.

Rubriques

Protection des données

Pour les données envoyées à Chat Amazon Interactive Video Service (IVS), les systèmes de protection de données suivants sont en place :

- Le trafic Amazon IVS Chat utilise WSS pour sécuriser les données pendant le transit.
- Les jetons Amazon IVS Chat sont chiffrés à l'aide de clés KMS gérées côté client.

Chat Amazon IVS n'exige pas que vous fournissiez des données client (utilisateur final). Aucun champ dans les salles de chat, les entrées ou les groupes de sécurité en entrée ne nécessite la fourniture de données clients.

N'indiquez pas d'informations d'identification sensibles telles que vos numéros de compte client (utilisateur final) dans des champs non structurés tels que le champ Nom. Cela vaut lorsque vous travaillez avec la console ou l'API Amazon IVS, l'AWS CLI ou des kits SDK AWS. Toutes les données que vous entrez dans Chat Amazon IVS ou d'autres services peuvent être incluses dans les journaux de diagnostic.

Les flux ne sont pas chiffrés de bout en bout ; un flux peut être transmis de façon interne et non chiffrée au sein du réseau IVS pour traitement.

Gestion de l'identité et des accès

AWS Identity and Access Management (IAM) est un service AWS qui permet à un administrateur de comptes de contrôler l'accès aux ressources AWS en toute sécurité. Consultez Gestion des identités et des accès dans le Guide de l'utilisateur du streaming à faible latence IVS.

Public ciblé

Votre utilisation d'IAM diffère selon les tâches réalisées dans Amazon IVS. Consultez Audience dans le Guide de l'utilisateur du streaming à faible latence IVS.

Fonctionnement d'Amazon IVS avec IAM

Avant de pouvoir effectuer des demandes d'API Amazon IVS, vous devez créer une ou plusieurs identités IAM (utilisateurs, groupes et rôles) ainsi que des politiques IAM, puis rattacher les politiques aux identités. La propagation des autorisations peut prendre quelques minutes. Jusque-là, les demandes d'API sont rejetées.

Pour une vue d'ensemble du fonctionnement d'Amazon IVS avec IAM, veuillez consulter la rubrique Services AWS qui fonctionnent avec IAM dans le Guide de l'utilisateur IAM.

Identités

Vous pouvez créer des identités IAM pour fournir une authentification aux personnes et aux processus de votre compte AWS. Les groupes IAM sont des collections d'utilisateurs IAM que vous gérez en tant qu'unité. Consultez la section Identités (utilisateurs, groupes et rôles) du Guide de l'utilisateur IAM.

Politiques

Les politiques sont des documents de politique d'autorisation JSON composés d'éléments. Consultez Politiques dans le Guide de l'utilisateur du streaming à faible latence IVS.

Chat Amazon IVS prend en charge trois éléments :

- Actions : les actions de politique pour Chat Amazon IVS utilisent le préfixe `ivschat` avant l'action. Par exemple, pour donner l'autorisation à une personne de créer une salle de Chat Amazon IVS avec la méthode d'API de Chat Amazon IVS `CreateRoom`, vous devez inclure l'action `ivschat:CreateRoom` dans la politique destinée à cette personne. Les déclarations de politique doivent inclure un élément `Action` ou `NotAction`.
- Ressources : la ressource de salle de Chat Amazon IVS possède le format d'ARN suivant :

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

  Par exemple, pour spécifier la salle `VgNkEJgOVX9N` dans votre instruction, utilisez l'ARN suivant :

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

  Certaines actions Chat Amazon IVS, comme celles destinées à la création de ressources, ne peuvent pas être exécutées sur une ressource spécifique. Dans ce cas, vous devez utiliser le caractère générique (*) :

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

- Conditions : Chat Amazon IVS prend en charge certaines clés de condition globales, soit `aws:RequestTag`, `aws:TagKeys` et `aws:ResourceTag`.

Vous pouvez utiliser des variables pour créer des espaces réservés dans une politique. Par exemple, vous pouvez accorder à un utilisateur IAM l'autorisation d'accéder à une ressource uniquement si elle est balisée avec son nom d'utilisateur IAM. Consultez la section Variables et balises du Guide de l'utilisateur IAM.

Amazon IVS propose des politiques gérées par AWS qui peuvent être utilisées pour accorder un ensemble préconfiguré d'autorisations aux identités (lecture seule ou accès complet). Vous pouvez choisir d'utiliser des politiques gérées au lieu des politiques basées sur l'identité présentées ci-dessous. Pour plus de détails, consultez Managed Policies for Amazon IVS.

Autorisation basée sur les balises Amazon IVS

Vous pouvez rattacher des balises aux ressources Chat Amazon IVS ou transmettre des balises dans une demande à Chat Amazon IVS. Pour contrôler l'accès basé sur des balises, vous devez fournir les informations des balises dans l'élément de condition d'une politique utilisant les clés de condition `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` ou `aws:TagKeys`. Pour plus d'informations sur le balisage des ressources Chat Amazon IVS, consultez la section « Balisage » de la Référence de l'API de Chat IVS.

Rôles

Consultez les sections Rôles IAM et Informations d'identification de sécurité temporaires du Guide de l'utilisateur IAM.

Un rôle IAM est une entité au sein de votre compte AWS qui dispose d'autorisations spécifiques.

Amazon IVS est compatible avec l'utilisation des informations d'identification de sécurité temporaires. Vous pouvez utiliser des informations d'identification temporaires pour vous connecter à l'aide de la fédération, endosser un rôle IAM ou encore pour endosser un rôle intercompte. Vous obtenez des informations d'identification de sécurité temporaires en appelant les opérations d'API AWS Security Token Service telles que `AssumeRole` ou `GetFederationToken`.

Accès privilégié et non privilégié

Les ressources API ont un accès privilégié. L'accès à la lecture non privilégié peut être configuré via des canaux privés ; consultez Configurer des canaux privés.

Bonnes pratiques pour l'utilisation des politiques

Consultez les Bonnes pratiques IAM dans le Guide de l'utilisateur IAM.

Les politiques basées sur l'identité sont très puissantes. Elles déterminent si une personne peut créer, consulter ou supprimer des ressources Amazon IVS sur votre compte. Ces actions peuvent entraîner des frais pour votre compte AWS. Suivez ces recommandations :

- Accorder le privilège le plus faible : lorsque vous créez des politiques personnalisées, accordez uniquement les autorisations nécessaires à l'exécution d'une tâche. Commencez avec un minimum d'autorisations et accordez-en d'autres si nécessaire. Cette méthode est plus sûre que de commencer avec des autorisations trop permissives et d'essayer de les restreindre plus tard. Plus précisément, réservez `ivschat:*` à l'accès administrateur ; ne l'utilisez pas dans les applications.
- Activer la MFA pour les opérations confidentielles — Pour plus de sécurité, demandez aux utilisateurs IAM d'utiliser la Multi-Factor Authentication (MFA) pour accéder à des ressources ou à des opérations d'API confidentielles.
- Utiliser des conditions de politique pour davantage de sécurité : définissez les conditions dans lesquelles vos politiques basées sur l'identité autorisent l'accès à une ressource, dans la mesure où cela reste pratique. Par exemple, vous pouvez rédiger les conditions pour spécifier une plage d'adresses IP autorisées d'où peut provenir une demande. Vous pouvez également écrire des conditions pour autoriser les requêtes uniquement à une date ou dans une plage de temps spécifiée, ou pour imposer l'utilisation de SSL ou de MFA.

Exemples de politiques basées sur l'identité

Utiliser la console Amazon IVS

Pour accéder à la console Amazon IVS, vous devez disposer d'un ensemble minimum d'autorisations qui vous permet de répertorier et d'afficher les détails des ressources Chat Amazon IVS sur votre compte AWS. Si vous créez une politique basée sur l'identité qui est plus restrictive que l'ensemble minimum d'autorisations requis, la console ne fonctionnera pas comme prévu pour les entités tributaires de cette politique. Pour garantir l'accès à la console Amazon IVS, rattachez la politique suivante aux identités (consultez la section Ajout et suppression d'autorisations IAM du Guide de l'utilisateur IAM).

Les quatre parties de la politique suivante donnent accès à :

- Tous les points de terminaison de l'API de Chat Amazon IVS
- Vos Service Quotas Chat Amazon IVS
- Liste des ressources lambdas et ajout d'autorisations pour la ressource lambda choisie pour la modération Amazon IVS Chat

- Amazon Cloudwatch pour obtenir des métriques pour votre session de chat

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

Politique basée sur les ressources pour Amazon IVS Chat

Vous devez accorder au service du kit SDK Amazon Chat l'autorisation d'appeler votre ressource lambda. Pour ce faire, suivez les instructions de la rubrique Utilisation de stratégies basées sur les ressources pour AWS Lambda (dans le Guide du développeur AWS Lambda) et remplissez les champs comme indiqué ci-dessous.

Pour contrôler l’accès à votre ressource lambda, vous pouvez utiliser des conditions basées sur :

- `SourceArn` : notre exemple de politique utilise un caractère générique (*) pour permettre à toutes les salles de votre compte d’appeler la ressource lambda. Vous pouvez également spécifier une salle dans votre compte pour autoriser uniquement cette salle à appeler la ressource lambda.
- `SourceAccount` : dans l’exemple de stratégie ci-dessous, l’ID de compte AWS est `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/*"
            }
         }
      }
   ]
}
```

Résolution des problèmes

Consultez la section Dépannage dans le Guide de l'utilisateur du streaming à faible latence IVS pour obtenir des informations sur le diagnostic et la résolution des problèmes courants que vous pouvez rencontrer lorsque vous travaillez avec Chat Amazon IVS et IAM.

Politiques gérées pour Amazon IVS

Une stratégie gérée par AWS est une stratégie qui est créée et gérée par AWS. Consultez Politiques gérées pour Amazon IVS dans le Guide de l'utilisateur du streaming à faible latence IVS.

Utilisation des rôles liés à un service pour Amazon IVS

Amazon IVS utilise des rôles liés à un service AWS IAM. Consultez la section Utilisation des rôles liés à un service pour Amazon IVS dans le Guide de l'utilisateur du streaming à faible latence IVS.

Journalisation et surveillance

Pour enregistrer les performances et/ou les opérations, utilisez Amazon CloudTrail. Consultez la section Journalisation des appels d'API Amazon IVS avec AWS CloudTrail dans le Guideuide de l'utilisateur du streaming à faible latence IVS.

Réponse aux incidents

Pour détecter ou alerter les incidents, vous pouvez surveiller l'état de votre flux via les événements Amazon EventBridge. Consultez la section « Utilisation d'Amazon EventBridge avec Amazon IVS » : pour le streaming à faible latence IVS et pour le streaming en temps réel IVS.

Utilisez le Tableau de bord AWS Health pour obtenir des informations sur l'état de santé général d'Amazon IVS (par région).

Résilience

L'API Amazon IVS utilise l'infrastructure mondiale AWS et repose sur des Régions et des zones de disponibilité AWS. Consultez la section Résilience dans le Guide de l'utilisateur du streaming à faible latence IVS.

Sécurité de l'infrastructure

Amazon IVS est un service géré, protégé par les procédures de sécurité du réseau mondial AWS. Elles sont décrites dans la section Bonnes pratiques en matière de sécurité, d'identité et de conformité.

Appels d’API

Vous pouvez utiliser les appels d’API publiés par AWS pour accéder à Amazon IVS via le réseau. Consultez la section Appels d'API sous Sécurité de l'infrastructure dans le Guide de l’utilisateur du streaming à faible latence IVS.

Chat Amazon IVS

L’ingestion et la remise des messages Amazon IVS Chat se font via des connexions WSS chiffrées à notre périphérie. L’API de messagerie Amazon IVS utilise des connexions HTTPS chiffrées. Comme pour le streaming et la lecture vidéo, TLS 1.2 ou une version ultérieure est requis et les données de messagerie peuvent être transmises sans chiffrement en interne pour traitement.

Quotas de service (Chat)

Voici des quotas de service et des limites pour les points de terminaison, les ressources et autres opérations d'Amazon Interactive Video Service chat (IVS). Les Service Quotas, également appelés limites, représentent le nombre maximal de ressources ou d'opérations de service pour votre compte AWS. Autrement dit, ces limites s'entendent par compte AWS, sauf indication contraire dans la table. Voir aussi Service Quotas AWS.

Pour vous connecter par programmation à un service AWS, vous devez utiliser un point de terminaison. Voir aussi Points de terminaison de service AWS.

Tous les quotas sont appliqués par région.

Augmentations des Service Quotas

Pour les quotas ajustables, vous pouvez demander une augmentation de taux via la console AWS. Vous pouvez également utiliser la console pour afficher des informations sur les Service Quotas.

Les quotas de taux d'appels API ne sont pas réglables.

Quotas de taux d'API

Type de point de terminaison	Point de terminaison	Par défaut
Messagerie	`DeleteMessage`	100 TPS
Messagerie	`DisconnectUser`	100 TPS
Messagerie	`SendEvent`	100 TPS
Jeton de chat	`CreateChatToken`	200 TPS
Configuration de la journalisation	`CreateLoggingConfiguration`	3 TPS
Configuration de la journalisation	`DeleteLoggingConfiguration`	3 TPS

Type de point de terminaison	Point de terminaison	Par défaut
Configuration de la journalisation	`GetLoggingConfiguration`	3 TPS
Configuration de la journalisation	`ListLoggingConfigurations`	3 TPS
Configuration de la journalisation	`UpdateLoggingConfiguration`	3 TPS
Salle	`CreateRoom`	5 TPS
Salle	`DeleteRoom`	5 TPS
Salle	`GetRoom`	5 TPS
Salle	`ListRooms`	5 TPS
Salle	`UpdateRoom`	5 TPS
Balises	`ListTagsForResource`	10 TPS
Balises	`TagResource`	10 TPS
Balises	`UntagResource`	10 TPS

Autres quotas

Ressource ou fonction	Par défaut	Ajustable	Description
Connexions de chat simultanées	50 000	Oui	Nombre maximal de connexions de chat simultanées par compte, dans toutes vos salles d'une Région AWS.
Configurations de journalisation	10	Oui	Le nombre maximal de configurations de journalisation pouvant être créées par

Ressource ou fonction	Par défaut	Ajustable	Description
			compte dans l'Région AWS actuelle.
Période d'expiration du gestionnaire de révision des messages	200	Non	Période d'expiration en millisecondes pour tous vos gestionnaires de révision des messages dans la Région AWS actuelle. Si cette valeur est dépassée, le message est autorisé ou refusé en fonction de la valeur du champ `fallbackResult` que vous avez configuré pour le gestionnaire de révision des messages.
Taux de demandes DeleteMessage dans toutes vos salles	100	Oui	Nombre maximum de requêtes DeleteMessage qui peuvent être effectuées par seconde dans toutes vos salles. Les demandes peuvent provenir de l'API Chat Amazon IVS ou de l'API de messagerie Chat Amazon IVS (WebSocket).
Taux de demandes DisconnectUser dans toutes vos salles	100	Oui	Nombre maximum de requêtes DisconnectUser qui peuvent être effectuées par seconde dans toutes vos salles. Les demandes peuvent provenir de l'API Chat Amazon IVS ou de l'API de messagerie Chat Amazon IVS (WebSocket).

Ressource ou fonction	Par défaut	Ajustable	Description
Taux de demandes de messagerie par connexion	10	Non	Nombre maximal de demandes de messagerie par seconde qu'une connexion de chat peut effectuer.
Taux de demandes SendMessage dans toutes vos salles	1 000	Oui	Nombre maximum de requêtes SendMessage qui peuvent être effectuée s par seconde dans toutes vos salles. Ces demandes proviennent de l'API de messagerie Chat Amazon IVS (WebSocket).
Taux de demandes SendMessage par salle	100	Non (mais configurable via l'API)	Nombre maximum de requêtes SendMessage qui peuvent être effectuées par seconde pour n'importe laquelle de vos salles. Cette option est configurable avec le champ `maximumMessageRatePerSecond` de CreateRoom et de UpdateRoom. Ces demandes proviennent de l'API de messagerie Chat Amazon IVS (WebSocket).
Salles	50 000	Oui	Nombre maximum de salles de chat par compte, par Région AWS.

Intégration de Service Quotas avec les métriques d’utilisation CloudWatch

Vous pouvez utiliser CloudWatch pour gérer de manière proactive vos Service Quotas, via les métriques d’utilisation de CloudWatch. Vous pouvez utiliser ces métriques pour visualiser votre utilisation actuelle du service sur des graphiques et des tableaux de bord CloudWatch. Les métriques d'utilisation Chat Amazon IVS correspondent aux quotas de service Chat Amazon IVS.

Vous pouvez utiliser une fonction mathématique de métrique CloudWatch pour afficher les Service Quotas pour ces ressources sur vos graphiques. Vous pouvez également configurer des alarmes qui vous alertent lorsque votre utilisation approche d'un Service Quota.

Pour accéder aux métriques d'utilisation :

1. Ouvrez la console Service Quotas à l'adresse https://console.aws.amazon.com/servicequotas/
2. Dans le volet de navigation, sélectionnez Services AWS.
3. Dans la liste des services AWS, recherchez et sélectionnez Amazon Interactive Video Service Chat.
4. Dans la liste des Service Quotas, sélectionnez le Service Quota à examiner. Une nouvelle page s'ouvre avec des informations sur le quota/la métrique de service.

Vous pouvez également accéder à ces statistiques via la console CloudWatch. Sous Espaces de noms AWS, sélectionnez Utilisation. Ensuite, dans la liste Service, sélectionnez Chat IVS. (Consultez Surveillance de Chat Amazon IVS.)

Dans l'espace de noms AWS/Utilisation, Chat Amazon IVS fournit la valeur suivante :

Nom de la métrique	Description
`ResourceCount`	Nombre des ressources spécifiées exécutées dans votre compte. Les ressources sont définies par les dimensions associées à la métrique. Statistique valide : Maximum (nombre maximal de ressources utilisées pendant la période d'une minute).

Les dimensions suivantes permettent d'affiner les métriques d'utilisation :

Dimension	Description
Service	Nom du service AWS contenant la ressource. Valeur valide : `IVS Chat`.
Classe	Classe de ressource suivie. Valeur valide : `None`.
Type	Type de ressource suivi. Valeur valide : `Resource`.
Ressource	Nom de la ressource AWS. Valeur valide : `ConcurrentChatConnections` . La métrique d'utilisation ConcurrentChatConnections est une copie de celle de l'espace de noms AWS/IVSChat (avec la dimension None), comme décrit dans la section Surveillance de Chat Amazon IVS.

Création d'une alarme CloudWatch pour les métriques d'utilisation

Pour créer une alarme CloudWatch basée sur une métrique d'utilisation Chat Amazon IVS :

1. Pour obtenir la console de Service Quotas, sélectionnez le Service Quota à évaluer, comme décrit ci-dessus. Actuellement, des alarmes ne peuvent être créées que pour ConcurrentChatConnections.
2. Dans la section Alarmes Amazon CloudWatch, sélectionnez Créer.
3. Dans la liste déroulante Seuil d'alarme, sélectionnez le pourcentage de la valeur de quota appliquée que vous souhaitez définir comme valeur d'alarme.
4. Pour Nom de l'alarme, saisissez un nom pour l'alarme.
5. Sélectionnez Créer.

Questions fréquentes sur le dépannage

Ce document décrit les bonnes pratiques et les conseils de dépannage du Chat d'Amazon Interactive Video Service (IVS). Les comportements liés à IVS Chat sont souvent distincts des comportements liés à la vidéo IVS. Pour plus d'informations, veuillez consulter la rubrique Mise en route avec le chat Amazon IVS.

Rubriques :

Pourquoi les connexions de chat IVS n'ont-elles pas été déconnectées lorsque la salle a été supprimée ?

Lorsqu'une ressource de salle de chat est supprimée, si la salle est activement utilisée, les clients de chat connectés à la salle ne sont pas automatiquement déconnectés. La connexion est interrompue si/lorsque l'application de chat actualise le jeton de chat. Sinon, une déconnexion manuelle de tous les utilisateurs doit être effectuée pour supprimer tous les utilisateurs de la salle de chat.

Glossaire

Consultez également le glossaire AWS. Dans le tableau ci-dessous, LL signifie streaming à faible latence IVS, tandis que RT signifie streaming en temps réel IVS.

Terme	Description	LL	RT	Chat
AAC	Codage audio avancé. L'AAC est une norme de codage audio pour la compression audio numérique avec perte. Conçu pour succéder au format MP3, le format AAC permet généralement d'obtenir une meilleure qualité sonore que le MP3 avec un débit identique. L'AAC a été normalisé par l'ISO et la CEI dans le cadre des spécifications MPEG-2 et MPEG-4.	✓	✓	
Streaming à débit binaire adaptatif	Le streaming à débit binaire adaptatif (ABR) permet au lecteur IVS de passer à un débit binaire inférieur lorsque la qualité de la connexion diminue, et de revenir à un débit supérieur lorsqu'elle s'améliore.	✓		
Streaming adaptatif	Consultez Encodage en couches avec Simulcast.		✓	
Utilisateur administratif	Utilisateur AWS disposant d'un accès administratif aux ressources et aux services disponibles sur un compte AWS. Consultez Terminologie dans le Guide de l'utilisateur de configuration d'AWS.	✓	✓	✓
ARN	Amazon Resource Name, un identifiant unique d'une ressource AWS. Les formats ARN spécifiques dépendent du type de ressource. Pour les formats d'ARN utilisés par les ressources IVS, consultez Référence de l'autorisation de service.	✓	✓	✓
Proportions	Décrit le rapport entre la largeur du cadre et la hauteur du cadre. Par exemple, 16:9 représente les	✓	✓	

Terme	Description	LL	RT	Chat
	proportions qui correspondent à la résolution Full HD ou 1080p.			
Mode audio	Configuration audio prédéfinie ou personnalisée optimisée pour différents types d'utilisateurs d'appareils mobiles et pour l'équipement qu'ils utilisent. Consultez SDK de diffusion IVS : Modes audio mobiles (Streaming en temps réel).		✓	
AVC, H.264, MPEG-4 partie 10	Le codage vidéo avancé, également appelé H.264 ou MPEG-4 partie 10, est une norme de compressi on vidéo pour la compression vidéo numérique avec perte.	✓	✓	
Remplacement d'arrière-plan	Type de filtre de caméra permettant aux créateurs de flux en direct de modifier leur arrière-plan. Consultez Remplacement d'arrière-plan dans SDK de diffusion IVS : filtres de caméra tiers (Streaming en temps réel).		✓	
Débit binaire	Une métrique de streaming pour le nombre de bits transmis ou reçus par seconde.	✓	✓	
Diffusion, diffuseur	Autres termes pour flux et streamer.	✓		
Mise en mémoire tampon	Une condition qui se produit lorsque le périphéri que de lecture n'est pas en mesure de télécharg er le contenu avant qu'il ne soit censé être lu. La mise en mémoire tampon peut se manifester de différentes manières : le contenu peut s'arrêter et se relancer de manière aléatoire (également appelé saut d'image), le contenu peut s'arrêter pendant de longues périodes (également connu sous le nom de blocage) ou le lecteur IVS peut suspendre la lecture.	✓	✓	

Terme	Description	LL	RT	Chat
Listes de lecture par plage d'octets	Une liste de lecture plus détaillée que la liste de lecture HLS standard. La liste de lecture HLS standard est composée de fichiers multimédias de 10 secondes. Avec une liste de lecture par plage d'octets, la durée du segment est la même que l'intervalle d'images clés configuré pour le flux. La liste de lecture par plage d'octets n'est disponible que pour les diffusions enregistrées automatiquement dans un compartiment S3. Elle est créée en complément de la liste de lecture HLS. Consultez Liste de lecture par plage d'octets dans Enregistrement automatique vers Amazon S3 (streaming à faible latence).	✓		
CBR	Débit binaire constant, une méthode de contrôle du débit pour les encodeurs qui maintient un débit binaire constant pendant toute la durée de lecture d'une vidéo, indépendamment de ce qui se passe pendant la diffusion. Les périodes d'accalmie peuvent être atténuées pour atteindre le débit souhaité, et les pics peuvent être quantifiés en ajustant la qualité du codage pour qu'elle corresponde au débit binaire cible. Nous recommandons vivement d'utiliser le débit constant (CBR) au lieu du débit variable (VBR).	✓	✓	
CDN	Réseau de diffusion de contenu ou réseau de distribution de contenu, une solution distribuée géographiquement qui optimise la diffusion de contenu tel que les vidéos en streaming en le rapprochant de l'emplacement des utilisateurs.	✓		

Terme	Description	LL	RT	Chat
Canal	Ressource IVS qui stocke la configuration pour le streaming, y compris un serveur d'ingestion, une clé de flux, une URL de lecture et des options d'enregistrement. Les streamers utilisent la clé de flux associée à un canal pour démarrer une diffusion. Tous les événements et métriques générés lors d'une diffusion sont associés à une ressource de canal.	✓		
Type de canal	Détermine la résolution et la fréquence d'images autorisées pour le canal. Consultez la section Types de canaux (français non garanti) dans Référence de l'API de diffusion à faible latence d'IVS.	✓		
Journalisation du chat	Option avancée qui peut être activée en associant une configuration de journalisation à une salle de chat.			✓
Salle de chat	Ressource IVS qui stocke la configuration d'une session de chat, y compris des fonctionnalités facultatives telles que Gestionnaire de révision des messages et Journalisation du chat. Consultez Étape 2 : créer une salle de chat dans Mise en route avec le chat IVS.			✓
Montage côté client	Utilise un appareil hôte pour mixer les flux audio et vidéo des participants à l'étape, puis les envoie sous forme de flux composite vers un canal IVS. Cela permet de mieux contrôler l'aspect de la composition au prix d'une utilisation plus importante des ressources des clients et d'un risque plus élevé qu'un problème lié à l'étape ou à l'hôte ait un impact sur les utilisateurs. Consultez également Montage côté serveur.	✓	✓	

Terme	Description	LL	RT	Chat
CloudFront	Service CDN fourni par Amazon.	✓		
CloudTrail	Service AWS permettant de collecter, de surveille r, d'analyser et de retenir les événements et les activités du compte provenant d'AWS et de sources externes. Consultez Journalisation des appels d'API Amazon IVS avec AWS CloudTrail.	✓	✓	✓
CloudWatch	Service AWS permettant de surveiller les applicati ons, de répondre aux changements de performan ces, d'optimiser l'utilisation des ressources et de fournir des informations sur l'état opérationnel. Vous pouvez utiliser CloudWatch pour surveille r les métriques IVS. Consultez Surveillance du streaming en temps réel IVS et Surveillance du streaming à faible latence IVS.	✓	✓	✓
Montage	Processus consistant à combiner des flux audio et vidéo provenant de plusieurs sources en un seul flux.	✓	✓	
Pipeline de montage	Séquence d'étapes de traitement requise pour combiner plusieurs flux et encoder le flux résultant.	✓	✓	
Compression	Codage des informations en utilisant moins de bits que la représentation d'origine. Toute compressi on particulière est soit sans perte, soit avec perte. La compression sans perte réduit le nombre de bits en identifiant et en éliminant la redondance statistique. Aucune information n'est perdue lors de la compression sans perte. La compression avec perte réduit le nombre de bits en supprimant les informations inutiles ou de moindre importance.	✓	✓	

Terme	Description	LL	RT	Chat
Plan de contrôle	Stocke des informations sur les ressources IVS telles que les canaux, les étapes ou les salles de chat et fournit des interfaces pour créer et gérer ces ressources. Il est régional (basé sur les Régions AWS).	✓	✓	✓
CORS	Le partage des ressources cross-origin (CORS), une fonctionnalité AWS qui permet aux applications Web clientes chargées dans un domaine particuli er d'interagir avec les ressources, telles que des compartiments S3 d'un autre domaine. L'accès peut être configuré en fonction des en-têtes, des méthodes HTTP et des domaines d'origine. Consultez Utilisation du partage des ressources entre origines multiples (CORS) - Amazon Simple Storage Service dans le Guide de l'utilisateur Amazon Simple Storage Service.	✓		
Source d'image personnalisée	Interface fournie par le SDK de diffusion IVS qui permet à une application de fournir sa propre entrée d'image au lieu de se limiter aux caméras prédéfinies.	✓	✓	
Plan de données	L'infrastructure qui transporte les données de l'entrée jusqu'à la sortie. Elle fonctionne sur la base de la configuration gérée dans le plan de contrôle et n'est pas limitée à une Région AWS.	✓	✓	✓
Encodeur, encodage	Le processus de conversion de contenu vidéo et audio en un format numérique, adapté au streaming. L'encodage peut être matériel ou logiciel.	✓	✓	

Terme	Description	LL	RT	Chat
Événement	Une notification automatique publiée par IVS au service de surveillance AmazonEventBridge. Un événement représente une modification de l'état ou de l'intégrité d'une ressource de streaming telle qu'une étape ou un pipeline de montage. Consultez Utilisation d'Amazon EventBridge avec le streaming à faible latence IVS et Utilisation d'Amazon EventBridge avec le streaming en temps réel IVS.	✓	✓	✓
FFmpeg	Un projet logiciel gratuit et open source composé d'une suite de bibliothèques et de programmes pour gérer des fichiers et des flux vidéo et audio. FFmpeg propose une solution multiplateforme pour enregistrer, convertir et diffuser du contenu audio et vidéo.	✓		
Flux fragmenté	Créé lorsqu'une diffusion se déconnecte puis se reconnecte durant l'intervalle spécifié dans la configuration d'enregistrement du canal. Les flux multiples qui en résultent sont considérés comme une diffusion unique et sont fusionnés en un flux enregistré unique. Consultez Fusionner des flux fragmentés dans Enregistrement automatique vers Amazon S3 (streaming à faible latence).	✓		
Fréquence de trames	Une métrique de streaming pour le nombre de trames vidéo transmises ou reçues par seconde.	✓	✓	
HLS	HTTP Live Streaming (HLS), un protocole de communication de streaming à débit binaire adaptatif basé sur le protocole HTTP et utilisé pour transmettre des flux IVS aux utilisateurs.	✓		

Terme	Description	LL	RT	Chat
Liste de lecture HLS	Liste des segments multimédias qui constitue nt un flux. Les listes de lecture HLS standard sont composées de fichiers multimédias de 10 secondes. HLS prend également en charge des listes de lecture plus détaillées par plage d'octets.	✓		
Host (Hôte)	Un participant à un événement en temps réel qui envoie de la vidéo et/ou du son à la scène.		✓	
IAM	Identity and Access Management, un service AWS qui permet aux utilisateurs de gérer en toute sécurité les identités et l'accès aux services et aux ressources AWS, y compris IVS.	✓	✓	✓
Ingestion	Processus IVS pour recevoir des flux vidéo d'un hôte ou d'un diffuseur à des fins de traitement ou de diffusion aux utilisateurs ou à d'autres participa nts.	✓	✓	
Serveur d'ingestion	Reçoit les flux vidéo et les transmet à un système de transcodage, où les flux sont transmutés ou transcodés en HLS pour être transmis aux spectateurs. Les serveurs d'ingestion sont des composants IVS spécifiques qui reçoivent des flux pour les canaux, ainsi qu'un protocole d'ingestion (RTMP, RTMPS). Consultez les informations sur la création d'un canal dans Mise en route avec le streaming à faible latence IVS.		✓	

Terme	Description	LL	RT	Chat
Vidéo entrelacée	Transmet et affiche uniquement les lignes paires ou impaires des trames suivantes afin de créer une impression de doublement de la fréquence de trames sans consommer de bande passante supplémentaire. Nous vous déconseillons d'utiliser la vidéo entrelacée pour des raisons liées à la qualité de la vidéo.	✓	✓	
JSON	JavaScript Object Notation, un format de fichier standard ouvert qui utilise du texte lisible par l'homme pour transmettre des objets de données composés de paires attribut-valeur et de types de données de tableau ou toute autre valeur sérialisable.	✓	✓	✓
Image clé, image delta, intervalle d'image clé	L'image clé (également appelée image intra-codée ou i-Frame) est une image complète de l'image d'une vidéo. Les images suivantes, les images delta (également appelées images prédites ou p-Frames), contiennent uniquement les informations modifiées. Les images-clés apparaîtront plusieurs fois dans un flux, en fonction de l'intervalle d'images clés défini dans l'encodeur.	✓	✓	
Lambda	Un service AWS permettant d'exécuter du code (appelé fonctions Lambda) sans allouer d'infrastructure de serveur. Les fonctions Lambda peuvent être exécutées en réponse à des événements et à des demandes d'invocation, ou selon un calendrier. Par exemple, le chat IVS utilise les fonctions Lambda pour permettre la révision des messages dans une salle de chat.	✓	✓	✓

Terme	Description	LL	RT	Chat
Latence, latence « glass-to-glass »	Un retard dans le transfert de données. IVS définit les plages de latence comme suit : • Faible latence : moins de 3 secondes • Latence en temps réel : moins de 300 ms La latence glass-to-glass fait référence au délai entre le moment où une caméra capture un flux en direct et le moment où le flux apparaît sur l'écran d'un utilisateur.	✓	✓	
Codage en couches avec Simulcast.	Permet le codage et la publication simultanés de plusieurs flux vidéo avec différents niveaux de qualité. Consultez Streaming adaptatif : encodage en couches avec Simulcast dans Optimisations du streaming en temps réel.		✓	
Gestionnaire de révision des messages	Permet aux clients du chat IVS de consulter/filtrer automatiquement les messages de chat des utilisateurs avant qu'ils ne soient envoyés dans la salle de chat. Il est activé en associant une fonction Lambda à une salle de chat. Consultez Creating a Lambda Function dans Chat Message Review Handler.			✓

Terme	Description	LL	RT	Chat
Mixeur	Une fonctionnalité des SDK de diffusion mobile IVS qui prend plusieurs sources audio et vidéo pour et générer une seule sortie. Elle prend en charge la gestion des éléments vidéo et audio à l'écran représentant des sources telles que des caméras, des microphones, des captures d'écran, ainsi que de l'audio et de la vidéo générés par l'applica tion. La sortie peut ensuite être transmise à IVS. Consultez Configuration d'une séance de diffusion pour le mixage dans SDK de diffusion IVS : guide de mixage (Streaming à faible latence).	✓		
Streaming multi-hôt es	Combine les flux provenant de plusieurs hôtes en un seul flux. Cela peut être accompli en utilisant un montage côté client ou côté serveur. Le streaming multi-hôtes permet des scénarios tels que l'invitation de spectateurs sur scène pour des questions-réponses, les compétitions entre hôtes, le chat vidéo et les conversations entre hôtes devant un large public.		✓	
Liste de lecture multivariante	Un index de tous les flux de variantes disponibles pour une diffusion.	✓		
OAC	Origin Access Control, un mécanisme permettant de restreindre l'accès à un compartiment S3, afin que le contenu tel qu'un flux enregistré ne puisse être diffusé que via le CDN CloudFront.	✓		

Terme	Description	LL	RT	Chat
OBS	Open Broadcaster Software, logiciel gratuit et open source pour l'enregistrement et la diffusion en direct de vidéos. OBS propose une alternative (au SDK de diffusion IVS) pour la publication sur ordinateur. Les streamers plus sophistiqués qui connaissent bien OBS peuvent le préférer en raison de ses fonctionnalités de production avancées, telles que les transitions de scène, le mixage audio et la superposition des graphiques.	✓	✓	
Participant	Un utilisateur en temps réel connecté à une scène en tant qu'hôte ou utilisateur.		✓	
Jeton de participant	Authentifie un participant à un événement en temps réel lorsqu'il rejoint une scène. Un jeton de participant contrôle également si un participant peut envoyer une vidéo à la scène.		✓	
Jeton de lecture, paire de clés de lecture	Un mécanisme d'autorisation qui permet aux clients de restreindre la lecture de vidéos sur les canaux privés. Les jetons de lecture sont générés à partir d'une paire de clés de lecture. Une paire de clés de lecture est la paire de clés publique-privée utilisée pour signer et valider le jeton d'autorisation de l'utilisateur pour la lecture. Consultez Créer ou importer une clé de lecture dans Configurer des canaux privés et voir les points de terminaison de la paire de clés de lecture dans Référence d'API IVS à faible latence.	✓		

Terme	Description	LL	RT	Chat
URL de lecture	Identifie l'adresse utilisée par l'utilisateur pour lancer la lecture d'un canal spécifique. Cette adresse peut être utilisée partout dans le monde. IVS sélectionne automatiquement le meilleur emplacement sur le réseau mondial de streaming de contenu IVS afin de diffuser la vidéo à chaque utilisateur. Consultez les informations sur la création d'un canal dans Mise en route avec le streaming à faible latence IVS.	✓		
Canal privé	Permet aux clients de restreindre l'accès à leurs flux à l'aide d'un mécanisme d'autorisation basé sur des jetons de lecture. Voir Flux de travail pour les canaux privés dans Configurer des canaux privés.	✓		
Vidéo progressive	Transmet et affiche toutes les lignes de chaque image en séquence. Nous recommandons d'utilise r la vidéo progressive à toutes les étapes d'une diffusion.	✓	✓	
Quotas	Le nombre maximal de ressources ou d'opérations de service IVS pour votre compte AWS. Autrement dit, ces limites s'entendent par compte AWS, sauf indication contraire. Tous les quotas sont appliqués par région. Consultez Amazon Interactive Video Service endpoints and quotas dans Guide de référence général AWS.	✓	✓	✓

Terme	Description	LL	RT	Chat
Régions	Elles permettent d'accéder aux services AWS qui résident physiquement dans une région géographique spécifique. Les régions fournissent une tolérance aux pannes, une stabilité et une résilience, et peuvent également réduire la latence. Les régions vous permettent de créer des ressources redondantes qui restent disponibles et qui ne sont pas affectées par une panne régionale. La plupart des demandes de service AWS sont associées à une région géographique particulière. Les ressources que vous créez dans une région n'existent pas dans une autre région, sauf si vous utilisez explicitement une fonction de réplication offerte par un service AWS. Par exemple, Amazon S3 prend en charge la réplication entre régions. Certains services, tels qu'IAM, n'ont pas de ressources interrégionales.	✓	✓	✓
Résolution	Décrit le nombre de pixels d'une seule image vidéo. Par exemple, Full HD ou 1080p définit une image de 1920 x 1080 pixels.	✓	✓	
Utilisateur root	Propriétaire d'un compte AWS. L'utilisateur root a un accès total à tous les services et ressources AWS du compte AWS.	✓	✓	✓
RTMP, RTMPS	Real-Time Messaging Protocol, une norme du secteur pour la transmission d'audio, de vidéos et de données sur un réseau. RTMPS est la version sécurisée de RTMP, qui s'exécute sur une connexion de protocole TLS/SSL (Transport Layer Security).	✓	✓	

Terme	Description	LL	RT	Chat
Compartiment S3	Un ensemble d'objets stockés dans Amazon S3. De nombreuses politiques, y compris l'accès et la réplication, sont définies au niveau du compartiment et s'appliquent à tous les objets du compartiment. Par exemple, une diffusion IVS est stockée en tant qu'objets multiples dans un compartiment S3.	✓		
SDK	Kit de développement logiciel, une collection de bibliothèques pour les développeurs qui créent des applications avec IVS.	✓	✓	✓
Segmentation des selfies	Permet de remplacer l'arrière-plan dans un flux en direct, à l'aide d'une solution propre au client qui accepte une image de caméra en entrée et qui renvoie un masque fournissant un score de confiance pour chaque pixel de l'image, indiquant s'il se trouve au premier plan ou en arrière-plan. Consultez Remplacement d'arrière-plan dans SDK de diffusion IVS : filtres de caméra tiers (Streaming en temps réel).		✓	
Gestion des versions sémantique	Un format de version au format Major.Minor.Patch. Les corrections de bogues n'affectant pas l'API incrémentent la version du correctif, les ajouts/modifications d'API rétrocompatibles incrémentent la version mineure et les modifications d'API non compatibles avec les versions antérieures incrémentent la version majeure.	✓	✓	✓

Terme	Description	LL	RT	Chat
Montage côté serveur	Utilise un serveur IVS pour mixer le son et la vidéo des participants à la scène, puis envoie cette vidéo mixée à un canal IVS pour atteindre un public plus large ou la stocker dans un compartiment S3. Le montage côté serveur réduit la charge client, améliore la résilience de la diffusion et permet une utilisation plus efficace de la bande passante. Consultez également Montage côté client.		✓	
Quotas de service	Service AWS qui vous permet de gérer vos quotas pour de nombreux services AWS à partir d'un seul emplacement. En plus de la recherche des valeurs des quotas, vous pouvez également demander à augmenter un quota à partir de la console Service Quotas.	✓	✓	✓
Rôle lié à un service	Un type unique de rôle IAM directement lié à un service AWS. Les rôles liés à des services sont automatiquement créés par IVS et ils incluent toutes les autorisations requises par le service pour appeler d'autres services AWS en votre nom, par exemple pour accéder à un compartiment S3. Consultez Using Service-Linked Roles for IVS dans IVS Security.	✓		
Étape	Une ressource IVS qui représente un espace virtuel où les participants à un événement en temps réel peuvent échanger des vidéos en temps réel. Consultez Créer une étape dans Mise en route avec le streaming en temps réel IVS.		✓	

Terme	Description	LL	RT	Chat
Session d'étape	Elle commence lorsque le premier participant rejoint une étape et se termine quelques minutes après que le dernier participant cesse d'être diffusé sur l'étape. Une étape de longue durée peut comporter plusieurs sessions au cours de sa durée de vie.		✓	
Flux	Données représentant un contenu vidéo ou audio envoyé en continu d'une source vers une destination.	✓	✓	
Clé de flux	Identifiant attribué par IVS lors de la création d'un canal et utilisé pour autoriser le streaming sur le canal. Traitez la clé de flux comme un secret, car elle permet à n'importe qui de diffuser sur le canal. Consultez Mise en route avec le streaming à faible latence IVS.	✓		
Pénurie de flux	Retard ou arrêt de la diffusion du flux vers IVS. Cela se produit lorsqu'IVS ne reçoit pas le nombre de bits que le dispositif d'encodage avait annoncé envoyer sur une certaine période. La survenue d'une pénurie de flux entraîne un événement de pénurie de flux. Pour le spectateur, une pénurie de flux peut se traduire par un retard, une mise en mémoire tampon ou un blocage d'une vidéo. La pénurie de flux peut être brève (moins de 5 secondes) ou longue (plusieurs minutes), selon la situation spécifique qui l'a provoquée. Consultez Qu'est-ce que la pénurie de flux ? dans Questions fréquentes sur le dépannage.	✓	✓	
Streamer	Personne ou appareil envoyant un flux vidéo ou audio à IVS.	✓	✓	

Terme	Description	LL	RT	Chat
Subscriber	Un participant à un événement en temps réel qui reçoit de la vidéo et/ou du son de l'hôte. Consultez Qu'est-ce qu'IVS Real-Time Streaming ?		✓	
Balise	Une balise de métadonnées est une étiquette que vous affectez à une ressource AWS. Les balises peuvent vous aider à identifier et à organiser vos ressources AWS. Sur la page d'accueil de la documentation IVS, consultez « Balisage » dans n'importe quelle documentation de l'API IVS (pour le streaming en temps réel, le streaming à faible latence ou le chat).	✓	✓	✓
Filtres de caméras tiers	Composants logiciels qui peuvent être intégrés au SDK de diffusion IVS pour permettre à une application de traiter des images avant de les transmettre au SDK de diffusion en tant que source d'image personnalisée. Un filtre de caméra tiers peut traiter les images provenant de la caméra, appliquer un effet de filtre, etc.	✓	✓	
Miniature	Image de taille réduite provenant d'un flux. Par défaut, les miniatures sont générées toutes les 60 secondes, mais un intervalle plus court peut être configuré. La résolution des miniatures dépend du type de canal. Consultez Enregistrement des contenus dans Enregistrement automatique vers Amazon S3 (streaming à faible latence).	✓		

Terme	Description	LL	RT	Chat
Métadonnées temporisées	Métadonnées liées à des horodatages spécifiques au sein d'un flux. Elles peuvent être ajoutées par programmation à l'aide de l'API IVS et sont associées à des images spécifiques. Cela garantit que tous les utilisateurs reçoivent les métadonnées au même point par rapport au flux. Les métadonnées temporisées peuvent être utilisées pour déclencher des actions sur le client, telles que la mise à jour des statistiques de l'équipe lors d'un événement sportif. Consultez Intégration de métadonnées dans un flux vidéo.	✓		
Transcodage	Convertit de la vidéo et de l'audio d'un format à un autre. Un flux entrant peut être transcodé dans un format différent à plusieurs débits et résolutions afin de prendre en charge une gamme de périphériques de lecture et de conditions réseau.	✓	✓	
Transmuxage	Un simple reconditionnement d'un flux ingéré vers IVS, sans réencodage du flux vidéo. « Transmux » est un raccourci pour le multiplexage transcodé , un processus qui change le format d'un fichier audio et/ou vidéo tout en conservant une partie ou la totalité des flux d'origine. Cela convertit vers un format de conteneur différent sans modifier le contenu du fichier. Il se distingue du transcodage.	✓	✓	

Terme	Description	LL	RT	Chat
Flux de variante	Ensemble d'encodages d'une même diffusion selon plusieurs niveaux de qualité distincts. Chaque flux de variante de flux est codé sous forme de liste de lecture HLS distincte. Un index des flux de variantes disponibles est appelé liste de lecture multivariante. Une fois que le lecteur IVS a reçu une liste de lecture multivariante d'IVS, il peut alors choisir entre les différents flux de variantes pendant la lecture, en passant de l'un à l'autre de manière transparente en fonction des conditions du réseau.	✓		
VBR	Débit binaire variable, méthode de contrôle du débit pour les encodeurs qui utilise un débit binaire dynamique qui change tout au long de la lecture, en fonction du niveau de détail requis. Nous vous déconseillons vivement d'utiliser le VBR pour des raisons de qualité de la vidéo. Utilisez plutôt le CBR.	✓	✓	

Terme	Description	LL	RT	Chat
Vue	Session de visionnage unique qui est en train de télécharger ou de lire activement des vidéos. Les vues constituent la base du quota de vues simultanées. Une vue démarre lorsqu'une session de visualisation lance la lecture vidéo. Une vue se termine lorsqu'une session de visualisation arrête la lecture vidéo. La lecture est le seul indicateur de l'audience ; les heuristiques d'engagement telles que les niveaux audio, la mise au point de l'onglet du navigateur et la qualité de la vidéo ne sont pas prises en compte. Lorsque vous comptez les vues, IVS ne tient pas compte de la légitimité des utilisateurs individuels et ne tente pas de dédupliquer les spectateurs localisés, par exemple dans le cas de plusieurs lecteurs vidéo sur une seule machine. Consultez Autres quotas dans Service Quotas (Streaming à faible latence).	✓		
Lecteur	Personne recevant un flux d'IVS.	✓		

Terme	Description	LL	RT	Chat
WebRTC	Web Real-Time Communication, un projet open source proposant une communication en temps réel aux navigateurs Web et aux applications mobiles. Il permet aux communications audio et vidéo de fonctionner au sein des pages Web en autorisant une communication peer-to-peer directe, en évitant ainsi de devoir installer des plug-ins ou télécharger des applications natives. Les technologies sous-jacentes à WebRTC sont mises en œuvre en tant que norme Web ouverte et sont disponibles sous forme d'API JavaScript standard dans tous les principaux navigateurs ou sous forme de bibliothèques pour clients natifs, comme Android et iOS.	✓	✓	

Terme	Description	LL	RT	Chat
WHIP	Protocole d'ingestion WebRTC-HTTP, un protocole basé sur HTTP qui permet l'ingestion de contenu basée sur WebRTC dans des services de streaming et/ou des CDN. WHIP est un brouillon de l'IETF développé pour normaliser l'ingestion du WebRTC. WHIP permet la compatibilité avec des logiciels tels que OBS, offrant une alternative (au SDK de diffusion IVS) pour la publication assistée par ordinateur. Les streamers plus sophistiqués qui connaissent bien OBS peuvent le préférer en raison de ses fonctionnalités de production avancées, telles que les transitions de scène, le mixage audio et la superposition des graphiques WHIP est également utile dans les situations où l'utilisation du SDK de diffusion IVS n'est pas faisable ou préférable. Par exemple, dans les configurations impliquant des encodeurs matériels , le SDK de diffusion IVS peut ne pas être une option. Toutefois, si l'encodeur prend en charge le protocole WHIP, vous pouvez toujours publier directement depuis l'encodeur vers IVS. Consultez la section Prise en charge d'OBS et du WHIP.		✓	
WSS	WebSocket Secure, protocole permettant d'établir des WebSockets via une connexion TLS chiffrée. Il sert à se connecter aux points de terminaison de chat IVS. Consultez Étape 4 : envoyer et recevoir votre premier message dans Mise en route avec le chat IVS.			✓

Historique du document (Chat)

Modifications du guide de l'utilisateur Chat

Modification	Description	Date
Division du Guide de l’utilisateur du chat	Des modifications majeures de la documentation accompagnent cette version. Nous avons transféré les informations de chat du Guide de l’utilisateur du streaming à faible latence IVS vers un nouveau Guide de l’utilisateur Chat IVS, se trouvant dans la section Chat IVS existante de la page d’accueil de la documentation IVS. Pour les autres modifications apportées à la documentation, consultez la section Historique du document (Streaming à faible latence).	28 décembre 2023
Glossaire IVS	Le glossaire a été étendu pour couvrir les termes IVS en temps réel, faible latence et chat.	20 décembre 2023

Modifications de la référence de l'API de chat IVS

Modifications d’API	Description	Date
Division du Guide de l’utilisateur du chat	Maintenant qu'il existe un Guide de l'utilisateur Chat IVS (créé dans cette version), les entrées de l'historique des documents pour la Référence de l'API de chat IVS et la Référence de l'API de messagerie IVS Chat existantes se trouveront ici à l'avenir. Les entrées d'historique antérieures pour ces Références de l'API de chat se trouvent dans l'Historique du document (Streaming à faible latence).	28 décembre 2023

Notes de mise à jour (Chat)

28 décembre 2023

Guide de l’utilisateur Chat Amazon IVS

Amazon Interactive Video Service (IVS) Chat est une fonction gérée de conversation en direct qui accompagne les flux vidéo en direct. Dans cette version, nous avons transféré les informations de chat du Guide de l'utilisateur du streaming à faible latence IVS vers un nouveau Guide de l'utilisateur Chat IVS. La documentation est accessible depuis la page de destination de la documentation Amazon IVS.

31 janvier 2023

Kit SDK de messagerie client Chat Amazon IVS : Android 1.1.0

Plateforme	Téléchargements et modifications
Kit SDK de messagerie client de chat Android 1.1.0	Documentation de référence : https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.1.0/ • Pour prendre en charge des coroutines Kotlin, nous avons ajouté de nouvelles API de messagerie de chat IVS dans le package com.amazonaws.ivs.chat.messaging.coroutines. Consultez également le nouveau didacticiel Coroutines Kotlin ; la partie 1 (sur 2) concerne les salles de chat.

Taille du kit SDK de messagerie client Chat : Android

Architecture	Taille compressée	Taille non compressée
Toutes les architectures (bytecode)	89 Ko	92 Ko

9 novembre 2022

Kit SDK de messagerie client Chat Amazon IVS : JavaScript 1.0.2

Plateforme	Téléchargements et modifications
Kit SDK de messagerie client JavaScript Chat 1.0.2	Documentation de référence : https://aws.github.io/amazon-ivs-chat-messaging-sdk-js/1.0.2/ • Correction d'un problème affectant Firefox : les clients recevaient par erreur une erreur de socket lorsqu'ils étaient déconnectés d'une salle de chat avec le point de terminaison DisconnectUser.

8 septembre 2022

Kit SDK de messagerie client Chat Amazon IVS : Android 1.0.0 et iOS 1.0.0

Plateforme	Téléchargements et modifications
Kit SDK de messagerie client de chat pour Android 1.0.0	Documentation de référence : https://aws.github.io/amazon-ivs-chat-messaging-sdk-android/1.0.0/
Kit SDK de messagerie client iOS 1.0.0	Documentation de référence : https://aws.github.io/amazon-ivs-chat-messaging-sdk-ios/1.0.0/

Taille du kit SDK de messagerie client Chat : Android

Architecture	Taille compressée	Taille non compressée
Toutes les architectures (bytecode)	53 Ko	58 Ko

Taille du kit SDK de messagerie client Chat : iOS

Architecture	Taille compressée	Taille non compressée
ios-arm64_x86_64-simulator (bitcode)	484 Ko	2,4 Mo
ios-arm64_x86_64-simulator	484 Ko	2,4 Mo
ios-arm64 (bitcode)	1,1 Mo	3,1 Mo
ios-arm64	233 Ko	1,2 Mo