Sportstalk Android SDK

Lawrence C. Cendana

May 13, 2021

CONTENTS:

CHAPTER

ONE

GETTING STARTED

Use this API to create experiences powered by SportsTalk and interact with those experiences.

- All API calls should be made using HTTPS.
- The API is designed to minimize the number of requests you need to make so chat applications are able to serve users very quickly especially for mobile users who need the lowest possible latency

1.1 Implementing the SDK

You can download the latest SportsTalk Android SDK from the following location:

https://gitlab.com/sportstalk247/sdk-android-kotlin

You need to register SportsTalk API with 'Appkey' and 'Token'.

1.2 How to get API Key and Token

You need to visit the dashboard with the following URL:

https://dashboard.sportstalk247.com

Then click on "Application Management" link to generate the above

1.3 Authentication

- All requests that require authentication must have the x-api-token header with your application token.
- Most requests require authentication.
- If you provide authentication on a request that does not require authentication the header will be ignored and it will have no effect.

1.4 SDK flow for Chat Applications

The typical flow of an application is:

- *Create Room*: Creates a chat room.
- *List Rooms*: Returns a list of available rooms. If you know the ID of the room you want there is no need to invoke List All Rooms first.
- *Join Room*: A user joins the room as an anonymous user or as a logged in user. Only logged in users can engage in chat activities. Anonymous users can only view whats happening in the room. You can also use custom room IDs that you provide so that you can use naming conventions to and join rooms without needing to call List All Rooms to get a room ID first.
- *Get Updates*: This gets the most recent events that have occurred in a room. You can use this endpoint as often as you want for polling, or you can use the Firebase API to get more bandwidth efficent push events when updates occur in the room. Calling Get Updates will prevent a logged in user from being removed from the room due to inactivity.
- *Execute Chat Command*: This performs a command in a chat room, like when you run a program from the command line. By default, the command is to say something in the room. However if a command starts with a special character such as / you can perform an action. See the Execute Chat Command API for more details and possible commands.
- *Exit Room*: When a logged in user exits the room, call this event. Otherwise the user will be removed from the room after some time without any activity.
- *List Participants*: Lists the logged in users in the chat room.
- *Get Room Details*: Gets statistics about the room such as the number of participants.

CHAPTER

TWO

HOW TO DOWNLOAD THE SDK

2.1 Download from Repository

The SportsTalk SDK has been published into **jitpack.io**. In order to use it in your application, just do the following:

1. Add the following in root **build.gradle** file

```
allprojects {
    repositories {
    // ...
       maven {
          url "https://jitpack.io"
       }
    }
}
```

1. Add the following lines in your module **build.gradle** file, depending on the chosen SDK implementation(Coroutine or Rx2Java), under dependencies section:

```
// For SDK coroutine implementation
implementation 'com.gitlab.sportstalk247:sdk-android-kotlin:sdk-coroutine:vX.Y.Z'
// OR
// For SDK Rx2Java implementation
implementation 'com.gitlab.sportstalk247:sdk-android-kotlin:sdk-reactive-rx2:vX.Y.Z'
```

Release

Then sync again. The gradle build should now be successful.

CHAPTER

THREE

HOW TO USE SPORTSTALK SDK

3.1 Instantiate SportsTalkManager Clients

This Sportstalk SDK is meant to power custom chat applications. Sportstalk does not enforce any restricitons on your UI design, but instead empowers your developers to focus on the user experience without worrying about the underlying chat behavior.

sdk-coroutine

```
import com.sportstalk.coroutine.SportsTalk247
// ...
```

sdk-reactive-rx2

```
import com.sportstalk.reactive.rx2.SportsTalk247
// ...
```

```
// ...

class MyFragment: Fragment() {
    // ...
    // ...

    // YOUR APP ID
    val appId = "c84cb9c852932a6b0411e75e" // This is just a sample app id
    // YOUR API TOKEN
    val apiToken = "5MGq3XbsspBEQf3kj154_OSQV-jygEKwHJyuHjuAeWHA" // This is just a␣
↪sample token
    val endpoint = "http://api.custom.endpoint/v1/" // please ensure out of the box the␣
↪SDKs are configured for production URL

    // Instantiate User Client
    val userClient = SportsTalk247.UserClient(
        config = ClientConfig(
            appId = appId,
            apiToken = apiToken,
            endpoint = endpoint
        )
    )

    // Instantiate Chat Client
```

(continues on next page)

(continued from previous page)

```
    val chatClient = SportsTalk247.ChatClient(
        config = ClientConfig(
            appId = appId,
            apiToken = apiToken,
            endpoint = endpoint
        )
    )

    // ...

}
```

3.2 Using the SDK

sdk-coroutine

This Android Sportstalk SDK artifact is an Asynchronous-driven API, powered by Kotlin Coroutines to gracefully handle asynchronous operations.

Client SDK functions are declared with *suspend* keyword. This means that the function should be invoked from within coroutine scope. See the example below:

```
class MyFragment: Fragment() {

    // ...
    // ...
    // Instantiate User Client
    val userClient = SportsTalk247.UserClient(/*...*/)

    // Instantiate Chat Client
    val chatClient = SportsTalk247.ChatClient(/*...*/)

    override fun onViewCreated(view: View) {
        // ...
        // Launch thru coroutine block
        // https://developer.android.com/topic/libraries/architecture/coroutines
        lifecycleScope.launch {
            // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
            val createdUser = withContext(Dispatchers.IO) {
                userClient.createOrUpdateUser(
                    request = CreateUpdateUserRequest(
                                userid = "8cb689cc-21b7-11eb-adc1-0242ac120002", //␣
↪sample user ID
                                handle = "sample_handle_123",
                                displayname = "Test Name 123", // OPTIONAL
                                pictureurl = "https://i.imgur.com/ohlx5wW.jpeg", //␣
↪OPTIONAL
                                profileurl = "https://i.imgur.com/ohlx5wW.jpeg" //␣
↪OPTIONAL
                            )
                )
```

(continues on next page)

(continued from previous page)

```
            }

            // Resolve `createdUser` from HERE onwards(ex. update UI displaying the␣
↪response data)...
        }

    }

}
```

sdk-reactive-rx2

This Android Sportstalk SDK artifact is a Reactive-driven API, powered by RxJava to gracefully handle reactive operations.

Client SDK functions returns RxJava types. See the example below:

```
class MyFragment: Fragment() {

    // ...
    // ...
    val rxDisposeBag = CompositeDisposable()

    // Instantiate User Client
    val userClient = SportsTalk247.UserClient(/*...*/)

    // Instantiate Chat Client
    val chatClient = SportsTalk247.ChatClient(/*...*/)

    override fun onViewCreated(view: View) {
        // ...

        userClient.createOrUpdateUser(
                    request = CreateUpdateUserRequest(
                                userid = "8cb689cc-21b7-11eb-adc1-0242ac120002", //␣
↪sample user ID
                                handle = "sample_handle_123",
                                displayname = "Test Name 123", // OPTIONAL
                                pictureurl = "https://i.imgur.com/ohlx5wW.jpeg", //␣
↪OPTIONAL
                                profileurl = "https://i.imgur.com/ohlx5wW.jpeg" //␣
↪OPTIONAL
                            )
                )
                .doOnSubscribe { rxDisposeBag.add(it) }
                .subscribe { createdUser ->
                    // Resolve `createdUser` from HERE onwards(ex. update UI displaying␣
↪the response data)...
                }
    }

}
```

3.3 Handling SDK Exception

If any client operations receive an error response, whether it be Network, Server, or Validation Error, these functions will throw an instance of `SportsTalkException`.

```
data class SportsTalkException(
    val kind: String? = null, // "api.result"
    val message: String? = null, // ex. "The specified comment was not found."
    val code: Int? = null // ex. 404,
    val data: Map<String, String?>? = null,
    val err: Throwable? = null
)
```

sdk-coroutine

```
// Under Fragment class
// Execute within coroutine scope
lifecycleScope.launch {
    val testComment = Comment(id = "0987654321",...)

    val setCommentDeletedResponse = try {
        withContext(Dispatchers.IO) {
            // These should throw Error 404 - "The specified conversation was not found␣
↪and was not deleted.".
            commentClient.permanentlyDeleteComment(
                conversationid = "Non-existent-Conversation-ID",
                commentid = testComment.id!!
            )
        }
    } catch(err: SportsTalkException) {
        // Resolve ERROR from HERE.
        return
    }
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()
// ...
// ...

// These should throw Error 404 - "The specified conversation was not found and was not␣
↪deleted.".
commentClient.permanentlyDeleteComment(
conversationid = "Non-existent-Conversation-ID",
commentid = testComment.id!!
)
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.doOnError { err ->
    when(err) {
        is SportsTalkException -> {
            // You may access [SportsTalkException] fields for error prompt
```

(continues on next page)

(continued from previous page)

```
        }
        else -> {
            // Catch all other error(s) encountered during the execution
        }
    }
}
.doOnSubscribe { rxDisposeBag.add(it) }
.subscribe { setCommentDeletedResponse ->
    // Resolve `setCommentDeletedResponse` (ex. Display prompt OR Update UI)
}
```

CHAPTER

FOUR

USER CLIENT

```
val userClient = SportsTalk247.UserClient(
   config = ClientConfig(
      appId = "c84cb9c852932a6b0411e75e", // This is just a sample app id
      apiToken = "5MGq3XbsspBEQf3kj154_OSQV-jygEKwHJyuHjuAeWHA", // This is just a␣
↪sample token
      endpoint = "http://api.custom.endpoint/v1/" // This is just a sample API endpoint
   )
)
```

4.1 Create or Update User

Invoke this function if you want to create a user or update an existing user.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#8cc680a6-6ce8-4af7-ab1e-e793a7f0e7d2

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val createdUser = withContext(Dispatchers.IO) {
        userClient.createOrUpdateUser(
            request = CreateUpdateUserRequest(
                        userid = "023976080242ac120002",
                        handle = "sample_handle_123",
                        displayname = "Test Name 123", // OPTIONAL
                        pictureurl = "<Image URL>", // OPTIONAL
                        profileurl = "<Image URL>" // OPTIONAL
            )
        )
    }

    // Resolve `createdUser` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.createOrUpdateUser(
    request = CreateUpdateUserRequest(
                userid = "023976080242ac120002",
                handle = "sample_handle_123",
                displayname = "Test Name 123", // OPTIONAL
                pictureurl = "<Image URL>", // OPTIONAL
                profileurl = "<Image URL>" // OPTIONAL
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { createdUser ->
        // Resolve `createdUser` (ex. Display prompt OR Update UI)
    }
```

4.2 Get User Details

This will return all the information about the user.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#3323caa9-cc3d-4569-826c-69070ca51758

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val userDetails = withContext(Dispatchers.IO) {
        userClient.getUserDetails(
            userid = "023976080242ac120002"
        )
    }

    // Resolve `userDetails` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.getUserDetails(
    userid = "023976080242ac120002"
)
    .subscribeOn(Schedulers.io())
```

(continues on next page)

(continued from previous page)

```
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { userDetails ->
        // Resolve `userDetails` (ex. Display prompt OR Update UI)
    }
```

4.3 List Users

Use this function to cursor through a list of users. This function will return users in the order in which they were created, so it is safe to add new users while cursoring through the list.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#51718594-63ac-4c28-b249-8f47c3cb02b1

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listUsers = withContext(Dispatchers.IO) {
        userClient.listUsers(
            limit = 10, /* Defaults to 200 on backend API server */
            cursor = null // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of user(s).
        )
    }

    // Resolve `listUsers` from HERE onwards(ex. update UI displaying the response data)..
↪.
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.listUsers(
    limit = 10, /* Defaults to 200 on backend API server */
    cursor = null // OPTIONAL: The cursor value from previous search attempt to indicate␣
↪next paginated fetch. Null if fetching the first list of user(s).
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listUsers ->
        // Resolve `listUsers` (ex. Display prompt OR Update UI)
    }
```

4.4 Ban User

This function toggles the specified user's banned flag.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#211d5614-b251-4815-bf76-d8f6f66f97ab

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val bannedUser = withContext(Dispatchers.IO) {
        userClient.setBanStatus(
            userid = "023976080242ac120002",
            applyeffect = true, // If set to true, attempt to ban the user. If set to␣
↪false, attempt to remove the ban from user
            expireseconds = 3600 // [Optional] if not specified, the ban is permanent␣
↪until user is restored. If specified, then the ban will be temporarily applied for the␣
↪specified number of seconds.

        )
    }

    // Resolve `bannedUser` from HERE onwards(ex. update UI displaying the response data).
↪..
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.setBanStatus(
    userid = "023976080242ac120002",
    applyeffect = true, // If set to true, attempt to ban the user. If set to false,␣
↪attempt to remove the ban from user
    expireseconds = 3600 // [Optional] if not specified, the ban is permanent until user␣
↪is restored. If specified, then the ban will be temporarily applied for the specified␣
↪number of seconds.
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { bannedUser ->
        // Resolve `bannedUser` (ex. Display prompt OR Update UI)
    }
```

4.5 Globally Purge User Content

This function will purge all chat content published by the specified user.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#c36d94e2-4fd9-4c9f-8009-f1d8ae9da6f5

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val response = withContext(Dispatchers.IO) {
        userClient.globallyPurgeUserContent(
            userid = "023976080242ac120002",
            banned = true // If set to true, attempt to purge all the chat messages␣
↪published by the specified user.
        )
    }

    // Resolve `response` from HERE onwards(ex. update UI displaying the response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.globallyPurgeUserContent(
    userid = "023976080242ac120002",
    banned = true // If set to true, attempt to purge all the chat messages published by␣
↪the specified user.
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { response ->
        // Resolve `response` (ex. Display prompt OR Update UI)
    }
```

4.6 Mute User

This function toggles the specified user's mute effect.

A muted user is in a read-only state. The muted user can join chat rooms and observe but cannot communicate. This method applies mute on the global level (applies to all talk contexts). You can optionally specify an expiration time. If the expiration time is specified, then each time the shadow banned user tries to send a message the API will check if the shadow ban has expired and will lift the ban.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#0d4c6409-18c6-41f4-9a61-7e2445c5bc0d

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val mutedUser = withContext(Dispatchers.IO) {
        userClient.muteUser(
            userId = "023976080242ac120002",
            applyeffect = true, // If set to true, user will be set to muted state. Otherwise, will be set to non-banned state.
            expireseconds = 3600 // [OPTIONAL]: Duration of mute in seconds. If specified, the mute will be lifted when this time is reached. If not specified, mute effect remains until explicitly lifted. Maximum seconds is a double byte value.
        )
    }

    // Resolve `mutedUser` from HERE onwards(ex. update UI displaying the response data)..
.
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.muteUser(
    userId = "023976080242ac120002",
    applyeffect = true, // If set to true, user will be set to muted state. Otherwise, will be set to non-banned state.
    expireseconds = 3600 // [OPTIONAL]: Duration of mute in seconds. If specified, the mute will be lifted when this time is reached. If not specified, mute effect remains until explicitly lifted. Maximum seconds is a double byte value.
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { mutedUser ->
        // Resolve `mutedUser` (ex. Display prompt OR Update UI)
    }
```

4.7 Set Shadow Ban Status

This function toggles the specified user's `shadowbanned` flag.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#211a5696-59ce-4988-82c9-7c614cab3efb

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val shadowBannedUser = withContext(Dispatchers.IO) {
        userClient.setShadowBanStatus(
            userId = "023976080242ac120002",
            applyeffect = true, // If set to true, user can send messages into a chat␣
↪room, however those messages are flagged as shadow banned.
            expireseconds = 3600 // [OPTIONAL]: Duration of shadowban value in seconds.␣
↪If specified, the shadow ban will be lifted when this time is reached. If not␣
↪specified, shadowban remains until explicitly lifted. Maximum seconds is a double byte␣
↪value.

        )
    }

    // Resolve `shadowBannedUser` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.setShadowBanStatus(
    userId = "023976080242ac120002",
    applyeffect = true, // If set to true, user can send messages into a chat room,␣
↪however those messages are flagged as shadow banned.
    expireseconds = 3600 // [OPTIONAL]: Duration of shadowban value in seconds. If␣
↪specified, the shadow ban will be lifted when this time is reached. If not specified,␣
↪shadowban remains until explicitly lifted. Maximum seconds is a double byte value.

)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { shadowBannedUser ->
        // Resolve `shadowBannedUser` (ex. Display prompt OR Update UI)
    }
```

4.8 Search User(s)

This function searches the users in an app.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#dea07871-86bb-4c12-bef3-d7290d762a06

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    // Search by Handle
    val searchedUsersByHandle = withContext(Dispatchers.IO) {
        userClient.searchUsers(
            handle = "testhandle1",
            limit = 20, // Defaults to 200 on backend API server
            cursor = null   // OPTIONAL: The cursor value from previous search attempt␣
↪to indicate next paginated fetch. Null if fetching the first list of user(s).
        )
    }

    // Search by Name
    val searchedUsersByName = withContext(Dispatchers.IO) {
        userClient.searchUsers(
            name = "Josie Rizal",
            limit = 20, // Defaults to 200 on backend API server
            cursor = null   // OPTIONAL: The cursor value from previous search attempt␣
↪to indicate next paginated fetch. Null if fetching the first list of user(s).
        )
    }

    // Search by User ID
    val searchedUsersByUserId = withContext(Dispatchers.IO) {
        userClient.searchUsers(
            userid = "userid_georgew",
            limit = 20, // Defaults to 200 on backend API server
            cursor = null   // OPTIONAL: The cursor value from previous search attempt␣
↪to indicate next paginated fetch. Null if fetching the first list of user(s).
        )
    }
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

// Search by Handle
userClient.searchUsers(
    handle = "testhandle1",
    limit = 20, // Defaults to 200 on backend API server
    cursor = null   // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of user(s).
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { searchedUsersByHandle ->
        // Resolve `searchedUsersByHandle` (ex. Display prompt OR Update UI)
    }
```

(continues on next page)

(continued from previous page)

```
// Search by Name
userClient.searchUsers(
    name = "Josie Rizal",
    limit = 20, // Defaults to 200 on backend API server
    cursor = null   // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of user(s).
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { searchedUsersByName ->
        // Resolve `searchedUsersByName` (ex. Display prompt OR Update UI)
    }

// Search by User ID
userClient.searchUsers(
    userid = "userid_georgew",
    limit = 20, // Defaults to 200 on backend API server
    cursor = null   // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of user(s).
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { searchedUsersByUserId ->
        // Resolve `searchedUsersByUserId` (ex. Display prompt OR Update UI)
    }
```

4.9 Delete User

This function will delete the specified user. All rooms with messages by that user will have the messages from this user purged in the rooms.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#ab387784-ad82-4025-bb3b-56659129279c

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val deletedUser = withContext(Dispatchers.IO) {
        userClient.deleteUser(
            userid = "023976080242ac120002"
        )
    }
```

(continues on next page)

(continued from previous page)

```
    // Resolve `deletedUser` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.deleteUser(
    userid = "023976080242ac120002"
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { deletedUser ->
        // Resolve `deletedUser` (ex. Display prompt OR Update UI)
    }
```

4.10 Report User

This function REPORTS a USER to the moderation team.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#5bfd5d93-dbfb-445c-84ff-c69f184e4277

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val reportedUser = withContext(Dispatchers.IO) {
        userClient.reportUser(
            userid = "023976080242ac120002"
        )
    }

    // Resolve `reportedUser` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.reportUser(
    userid = "023976080242ac120002"
)
    .subscribeOn(Schedulers.io())
```

(continues on next page)

(continued from previous page)

```
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { reportedUser ->
        // Resolve `reportedUser` (ex. Display prompt OR Update UI)
    }
```

4.11 List User Notifications

This function returns a list of user notifications.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#f09d36c2-de40-4866-8818-74527b2a6df5

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listUserNotifications = withContext(Dispatchers.IO) {
        userClient.listUserNotifications(
            userid = "023976080242ac120002",
            limit = 10, // Can be any arbitrary number
            filterNotificationTypes = listOf(UserNotification.Type.CHAT_REPLY,␣
↪UserNotification.Type.CHAT_QUOTE), // [OPTIONAL] List could also have either `CHAT_
↪REPLY` or `CHAT_QUOTE` ONLY
            cursor = null,
            includeread = false, // If [true], will only return a list of user␣
↪notifications whose value `isread = true`. Otherwise, returns a list of user␣
↪notifications whose value `isread = false`.
            filterChatRoomId = "080001297623242ac002", // ID of an existing chat room
            filterChatRoomCustomId = null   // OR you may also use an existing chat room
↪'s custom ID
        )
    }

    // Resolve `listUserNotifications` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.listUserNotifications(
    userid = "023976080242ac120002",
    limit = 10, // Can be any arbitrary number
    filterNotificationTypes = listOf(UserNotification.Type.CHAT_REPLY, UserNotification.
↪Type.CHAT_QUOTE), // [OPTIONAL] List could also have either `CHAT_REPLY` or `CHAT_
↪QUOTE` ONLY
```

(continues on next page)

(continued from previous page)

```
    cursor = null,
    includeread = false, // If [true], will only return a list of user notifications␣
↪whose value `isread = true`. Otherwise, returns a list of user notifications whose␣
↪value `isread = false`.
    filterChatRoomId = "080001297623242ac002", // ID of an existing chat room
    filterChatRoomCustomId = null   // OR you may also use an existing chat room's custom␣
↪ID
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listUserNotifications ->
        // Resolve `listUserNotifications` (ex. Display prompt OR Update UI)
    }
```

4.12 Set User Notification as Read

This marks a notification as being in READ status. That will prevent the notification from being returned in a call to List User Notifications unless the default filters are overridden. Notifications that are marked as read will be automatically deleted after some time.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#073d5ec4-cef6-46cc-8b52-72083db6f310

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val updatedNotification = withContext(Dispatchers.IO) {
        userClient.setUserNotificationAsRead(
            userid = "023976080242ac120002",    // The ID of user who owns the␣
↪notification about to update
            notificationId = "070200623280c142a902",    // The ID of notifications about␣
↪to update
            read = true
        )
    }

    // Resolve `updatedNotification` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.setUserNotificationAsRead(
```

(continues on next page)

(continued from previous page)

```
    userid = "023976080242ac120002",    // The ID of user who owns the notification␣
↪about to update
    notificationId = "070200623280c142a902",    // The ID of notifications about to␣
↪update
    read = true
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { updatedNotification ->
        // Resolve `updatedNotification` (ex. Display prompt OR Update UI)
    }
```

4.13 Set User Notification as Read by Chat Event

This marks a notification as being in READ status. That will prevent the notification from being returned in a call to List User Notifications unless the default filters are overridden. Notifications that are marked as read will be automatically deleted after some time.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#073d5ec4-cef6-46cc-8b52-72083db6f310

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val updatedNotification = withContext(Dispatchers.IO) {
        userClient.setUserNotificationAsReadByChatEvent(
            userid = "023976080242ac120002",    // The ID of user who owns the␣
↪notification about to update
            chatEventId = "070200623280c142a902",    // The ID of chatevent for which␣
↪the notification was generated from, about to update
            read = true
        )
    }

    // Resolve `updatedNotification` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.setUserNotificationAsReadByChatEvent(
    userid = "023976080242ac120002",    // The ID of user who owns the notification␣
↪about to update
```

(continues on next page)

(continued from previous page)

```
    chatEventId = "070200623280c142a902",    // The ID of chatevent for which the␣
↪notification was generated from, about to update
    read = true
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { updatedNotification ->
        // Resolve `updatedNotification` (ex. Display prompt OR Update UI)
    }
```

4.14 Delete User Notification

This function immediately deletes a user notification. Unless your workflow specifically implements access to read notifications, you should delete notifications after they are consumed.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#7cbb108d-8b72-4c59-8537-fa9ea4a71364

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val deletedNotification = withContext(Dispatchers.IO) {
        userClient.deleteUserNotification(
            userid = "023976080242ac120002",    // The ID of user who owns the␣
↪notification about to delete
            notificationId = "070200623280c142a902"    // The ID of notifications about␣
↪to delete
        )
    }

    // Resolve `deletedNotification` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.deleteUserNotification(
    userid = "023976080242ac120002",    // The ID of user who owns the notification␣
↪about to delete
    notificationId = "070200623280c142a902"    // The ID of notifications about to delete
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
```

(continues on next page)

(continued from previous page)

```
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { deletedNotification ->
        // Resolve `deletedNotification` (ex. Display prompt OR Update UI)
    }
```

4.15 Delete User Notification by Chat Event

This function immediately deletes a user notification. Unless your workflow specifically implements access to read notifications, you should delete notifications after they are consumed.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#073d5ec4-cef6-46cc-8b52-72083db6f310

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val deletedNotification = withContext(Dispatchers.IO) {
        userClient.deleteUserNotificationByChatEvent(
            userid = "023976080242ac120002",    // The ID of user who owns the␣
↪notification about to update
            chatEventId = "07020623280c142a902"    // The ID of chatevent for which the␣
↪notification was generated from, about to delete
        )
    }

    // Resolve `deletedNotification` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.deleteUserNotificationByChatEvent(
    userid = "023976080242ac120002",    // The ID of user who owns the notification␣
↪about to update
    chatEventId = "07020623280c142a902"    // The ID of chatevent for which the␣
↪notification was generated from, about to delete
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { deletedNotification ->
        // Resolve `deletedNotification` (ex. Display prompt OR Update UI)
    }
```

4.16 Mark All User Notifications as Read

This marks a all notifications of the user as being in READ status. If delete is set to true, notifications are deleted instead of updated.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#e0c669ff-4722-46b0-ab3e-d1d74d9d340a

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    withContext(Dispatchers.IO) {
        userClient.markAllUserNotificationsAsRead(
            userid = "023976080242ac120002",    // The ID of user who owns the␣
↪notification about to update
            delete = true
        )
    }
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

userClient.markAllUserNotificationsAsRead(
    userid = "023976080242ac120002",    // The ID of user who owns the notification␣
↪about to update
    delete = true
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe {
        // Do something afterwards...
    }
```

CHAPTER

FIVE

CHAT CLIENT

```
val chatClient = SportsTalk247.ChatClient(
   config = ClientConfig(
      appId = "c84cb9c852932a6b0411e75e", // This is just a sample app id
      apiToken = "5MGq3XbsspBEQf3kj154_OSQV-jygEKwHJyuHjuAeWHA", // This is just a␣
↪sample token
      endpoint = "http://api.custom.endpoint/v1/" // This is just a sample API endpoint
   )
)
```

5.1 Room Subscriptions

Invoke this function to see the set of ChatRoom IDs to keep track which rooms are subscribed to get event updates. Room subscriptions gets updated each time `startListeningToChatUpdates(forRoomId: String)` and `stopListeningToChatUpdates(forRoomId: String)` gets invoked.

```
    val roomSubscriptions: Set<String> = chatClient.roomSubscriptions()
```

5.2 Get Chat Room Event Update Cursor

Get current event update cursor for the specified room ID. This gets updated either each time `allEventUpdates()` emits a value, when `joinRoom()/joinRoomByCustomId()` are invoked, OR when `setChatRoomEventUpdateCursor()/clearChatRoomEventUpdateCursor()` are invoked.

```
    val currentRoomId = "<joined-room-id>"
    val currentEventUpdateCursor: String? =
        chatClient.getChatRoomEventUpdateCursor(
            forRoomId = currentRoomId
        )  // Could be `null` if not yet set
```

5.3 Set Chat Room Event Update Cursor

Allows developers to override the event updates cursor to have more control on how paging logic is implemented.

```
    val currentRoomId = "<joined-room-id>"
    val overrideCursor = "<a valid event update cursor>"
    chatClient.setChatRoomEventUpdateCursor(
        forRoomId = currentRoomId,
        cursor = overrideCursor
    )
```

5.4 Clear Chat Room Event Update Cursor

Allows developers to clear the event updates cursor(when cleared, the next time `allEventUpdates()` performs REST API operation, it will NOT include a cursor value on the request).

```
    val currentRoomId = "<joined-room-id>"
    chatClient.clearChatRoomEventUpdateCursor(forRoomId = currentRoomId)
```

5.5 Create Room

Invoke this function to create a new chat room.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#8b2eea78-82bc-4cae-9cfa-175a00a9e15b

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val createdRoom = withContext(Dispatchers.IO) {
        chatClient.createRoom(
            request = CreateChatRoomRequest(
                name = "Test Chat Room 1",
                customid = "test-room-1",
                description = "This is a test chat room 1.",
                moderation = "post",
                enableactions = true,
                enableenterandexit = true,
                enableprofanityfilter = false,
                delaymessageseconds = 0L,
                roomisopen = true,
                maxreports = 0
            )
        )
    }
```

(continues on next page)

(continued from previous page)

```
    // Resolve `createdRoom` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.createRoom(
    request = CreateChatRoomRequest(
        name = "Test Chat Room 1",
        customid = "test-room-1",
        description = "This is a test chat room 1.",
        moderation = "post",
        enableactions = true,
        enableenterandexit = true,
        enableprofanityfilter = false,
        delaymessageseconds = 0L,
        roomisopen = true,
        maxreports = 0
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { createdRoom ->
        // Resolve `createdRoom` (ex. Display prompt OR Update UI)
    }
```

5.6 Get Room Details

Invoke this function to get the details for a room.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#9bac9724-7505-4e3e-966f-08cfebbca88d

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val chatRoom = withContext(Dispatchers.IO) {
        chatClient.getRoomDetails(
            chatRoomId = "080001297623242ac002"
        )
    }
```

(continues on next page)

(continued from previous page)

```
    // Resolve `chatRoom` from HERE onwards(ex. update UI displaying the response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.getRoomDetails(
    chatRoomId = "080001297623242ac002"
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { chatRoom ->
        // Resolve `chatRoom` (ex. Display prompt OR Update UI)
    }
```

5.7 Get Room Extended Details Batch

Invoke this function to get the extended details for a room.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#f9417096-7eac-44e1-846b-9a4782fb8279

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val chatRoomExtendedDetails = withContext(Dispatchers.IO) {
        chatClient.getRoomDetailsExtendedBatch(
            entityTypes = listOf(
                RoomDetailEntityType.ROOM,
                RoomDetailEntityType.NUM_PARTICIPANTS,
                RoomDetailEntityType.LAST_MESSAGE_TIME
            ),  // Must have atleast 1 of the RoomDetailEntityType enum constant.
            roomIds = listOf("080001297623242ac002", "702242ac000086230129"),  // Must␣
↪have atleast 1 entry for roomIds or customIds combined.
            customIds = listOf("test-custom-room-id-01", "test-custom-room-id-02")
        )
    }

    // Resolve `chatRoomExtendedDetails` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.getRoomDetailsExtendedBatch(
    entityTypes = listOf(
        RoomDetailEntityType.ROOM,
        RoomDetailEntityType.NUM_PARTICIPANTS,
        RoomDetailEntityType.LAST_MESSAGE_TIME
    ),  // Must have atleast 1 of the RoomDetailEntityType enum constant.
    roomIds = listOf("080001297623242ac002", "702242ac000086230129"),  // Must have␣
↪atleast 1 entry for roomIds or customIds combined.
    customIds = listOf("test-custom-room-id-01", "test-custom-room-id-02")
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { chatRoomExtendedDetails ->
        // Resolve `chatRoomExtendedDetails` (ex. Display prompt OR Update UI)
    }
```

5.8 Get Room Details By CustomId

Invoke this function to get the details for a room, using custom ID.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#0fd07be5-f8d5-43d9-bf0f-8fb9829c172c

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val chatRoom = withContext(Dispatchers.IO) {
        chatClient.getRoomDetailsByCustomId(
            chatRoomCustomId = "custom-id-0239760802"
        )
    }

    // Resolve `chatRoom` from HERE onwards(ex. update UI displaying the response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.getRoomDetailsByCustomId(
    chatRoomCustomId = "custom-id-0239760802"
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
```

(continues on next page)

(continued from previous page)

```
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { chatRoom ->
        // Resolve `chatRoom` (ex. Display prompt OR Update UI)
    }
```

5.9 List Rooms

Invoke this function to list all the available public chat rooms.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#0580f06e-a58e-447a-aa1c-6071f3cfe1cf

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listRooms = withContext(Dispatchers.IO) {
        chatClient.listRooms(
            limit = 20, /* Defaults to 200 on backend API server */
            cursor = null // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of chat room(s).
        )
    }

    // Resolve `chatRoom` from HERE onwards(ex. update UI displaying the response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.listRooms(
    limit = 20, /* Defaults to 200 on backend API server */
    cursor = null // OPTIONAL: The cursor value from previous search attempt to indicate␣
↪next paginated fetch. Null if fetching the first list of chat room(s).
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listRooms ->
        // Resolve `listRooms` (ex. Display prompt OR Update UI)
    }
```

5.10 Join Room (Authenticated User)

Invoke this function to join a room.

You want your chat experience to open fast. The steps to opening a chat experience are:

1. Create Room
2. Create User
3. Join Room (user gets permission to access events data from the room)
4. Get Recent Events to display in your app

If you have already created the room (step 1) then you can perform steps 2 - 4 using join room.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#eb3f78c3-a8bb-4390-ab25-77ce7072ddda

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val joinRoomResponse = withContext(Dispatchers.IO) {
        chatClient.joinRoom(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            request = JoinChatRoomRequest(
                userid = "023976080242ac120002" // ID of an existing user from this␣
↪chatroom
            )
        )
    }

    // Resolve `joinRoomResponse` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.joinRoom(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    request = JoinChatRoomRequest(
        userid = "023976080242ac120002" // ID of an existing user from this chatroom
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { joinRoomResponse ->
        // Resolve `joinRoomResponse` (ex. Display prompt OR Update UI)
    }
```

5.11 Join Room By CustomID

Invoke this function to join a room by Custom ID. This method is the same as Join Room, except you can use your customid.

The benefit of this method is you don't need to query to get the roomid using customid, and then make another call to join the room. This eliminates a request and enables you to bring your chat experience to your user faster.

You want your chat experience to open fast. The steps to opening a chat experience are:

1. Create Room
2. Create User
3. Join Room (user gets permission to access events data from the room)
4. Get Recent Events to display in your app

If you have already created the room (step 1) then you can perform steps 2 - 4 using join room.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#a64f2c32-6167-4639-9c32-413edded2c18

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val joinRoomResponse = withContext(Dispatchers.IO) {
        chatClient.joinRoomByCustomId(
            chatRoomCustomId = "custom-room-id-12976",    // Custom ID of an existing␣
↪chat room
            request = JoinChatRoomRequest(
                userid = "023976080242ac120002" // ID of an existing user from this␣
↪chatroom
            )
        )
    }

    // Resolve `joinRoomResponse` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.joinRoomByCustomId(
    chatRoomCustomId = "custom-room-id-12976",    // Custom ID of an existing chat room
    request = JoinChatRoomRequest(
        userid = "023976080242ac120002" // ID of an existing user from this chatroom
    )
)
    .subscribeOn(Schedulers.io())
```

(continues on next page)

(continued from previous page)

```
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { joinRoomResponse ->
        // Resolve `joinRoomResponse` (ex. Display prompt OR Update UI)
    }
```

5.12 List Room Participants

Invoke this function to list all the participants in the specified room.

Use this method to cursor through the people who have subscribe to the room.

To cursor through the results if there are many participants, invoke this function many times. Each result will return a cursor value and you can pass that value to the next invokation to get the next page of results. The result set will also include a next field with the full URL to get the next page, so you can just keep reading that and requesting that URL until you reach the end. When you reach the end, no more results will be returned or the result set will be less than maxresults and the next field will be empty.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#1b1b82a9-2b2f-4785-993b-baed6e7eba7b

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listRoomParticipants = withContext(Dispatchers.IO) {
        chatClient.listRoomParticipants(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            limit = 20, /* Defaults to 200 on backend API server */
            cursor = null // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of chatroom␣
↪participant(s).
        )
    }

    // Resolve `listRoomParticipants` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.listRoomParticipants(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    limit = 20, /* Defaults to 200 on backend API server */
    cursor = null // OPTIONAL: The cursor value from previous search attempt to indicate␣
↪next paginated fetch. Null if fetching the first list of chatroom participant(s).
```

(continues on next page)

(continued from previous page)

```
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listRoomParticipants ->
        // Resolve `listRoomParticipants` (ex. Display prompt OR Update UI)
    }
```

5.13 Update Room

Invoke this function to update an existing room.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#96ef3138-4820-459b-b400-e9f25d5ddb00

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val updatedRoom = withContext(Dispatchers.IO) {
        chatClient.updateRoom(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            request = UpdateChatRoomRequest(
                name = "${testData.name!!}-updated",
                customid = "${testData.customid}-updated(${System.currentTimeMillis()})",
                description = "${testData.description}-updated",
                enableactions = !testData.enableactions!!,
                enableenterandexit = !testData.enableenterandexit!!,
                maxreports = 30L
            )
        )
    }

    // Resolve `updatedRoom` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.updateRoom(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    request = UpdateChatRoomRequest(
        name = "${testData.name!!}-updated",
        customid = "${testData.customid}-updated(${System.currentTimeMillis()})",
        description = "${testData.description}-updated",
```

(continues on next page)

(continued from previous page)

```
        enableactions = !testData.enableactions!!,
        enableenterandexit = !testData.enableenterandexit!!,
        maxreports = 30L
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { updatedRoom ->
        // Resolve `updatedRoom` (ex. Display prompt OR Update UI)
    }
```

5.14 Execute Chat Command (say 'Hello, World!')

Invoke this function to execute a command in a chat room.

Precondition: The user must JOIN the room first with a call to Join Room. Otherwise you'll receive HTTP Status Code PreconditionFailed (412)

5.14.1 SENDING A MESSAGE

- Send any text that doesn't start with a reserved symbol to perform a SAY command.
- Use this function to REPLY to existing messages
- Use this function to perform ACTION commands
- Use this function to perform ADMIN commands

example:

```
ExecuteChatCommandRequest(
    command = "These commands both do the same thing, which is send the message 'Hello␣
↪World' to the room. SAY Hello, World Hello, World",
    // ....
)
```

5.14.2 ACTION COMMANDS

- Action commands start with the / character

example:

```
// Assuming current user's handle is "@MikeHandle05"
ExecuteChatCommandRequest(
    command = "/dance nicole",
    // ....
)

// User sees: "You dance with Nicole"
```

(continues on next page)

(continued from previous page)

```
// Nicole sees: "@MikeHandle05 dances with you"
// Everyone else sees: "@MikeHandle05 dances with Nicole"
```

This requires that the action command dance is on the approved list of commands and Nicole is the handle of a participant in the room, and that actions are allowed in the room.

5.14.3 ADMIN COMMANDS

- These commands start with the * character

example:

```
// This bans the user from the entire chat experience (all rooms).
ExecuteChatCommandRequest(
    command = "*ban",
    // ....
)
```

```
// This restores the user to the chat experience (all rooms).
ExecuteChatCommandRequest(
    command = "*restore",
    // ....
)
```

```
// This deletes all messages from the specified user.
ExecuteChatCommandRequest(
    command = "*purge",
    // ....
)
```

```
// This deletes all messages in this room.
// Assuming ADMIN password "testpassword123"
ExecuteChatCommandRequest(
    command = "*deleteallevents testpassword123",
    // ....
)
```

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#c81e90fc-1a54-40bb-a75b-2fc935c12b59

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val executeChatCmdResponse = withContext(Dispatchers.IO) {
        chatClient.executeChatCommand(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            request = ExecuteChatCommandRequest(
```

(continues on next page)

(continued from previous page)

```
                command = "Hello World!",
                userid = "023976080242ac120002" // ID of an existing user from this␣
↪chatroom
            )
        )
    }

    // Resolve `executeChatCmdResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.executeChatCommand(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    request = ExecuteChatCommandRequest(
        command = "Hello World!",
        userid = "023976080242ac120002" // ID of an existing user from this chatroom
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { executeChatCmdResponse ->
        // Resolve `executeChatCmdResponse` (ex. Display prompt OR Update UI)
    }
```

5.15 Execute Chat Command (Announcement by Admin)

Invoke this function to execute a command in a chat room.

Precondition: The user must JOIN the room first with a call to Join Room. Otherwise you'll receive HTTP Status Code PreconditionFailed (412)

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#45c88ff5-4006-491a-b4d3-5f2ad542fa09

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val executeChatCmdResponse = withContext(Dispatchers.IO) {
        chatClient.executeChatCommand(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            request = ExecuteChatCommandRequest(
                command = "This is a test annoncement!",
```

(continues on next page)

(continued from previous page)

```
                userid = "023976080242ac120002", // ID of an existing user from this␣
↪chatroom
                eventtype = "announcement"
            )
        )
    }

    // Resolve `executeChatCmdResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.executeChatCommand(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    request = ExecuteChatCommandRequest(
        command = "This is a test annoncement!",
        userid = "023976080242ac120002", // ID of an existing user from this chatroom
        eventtype = "announcement"
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { executeChatCmdResponse ->
        // Resolve `executeChatCmdResponse` (ex. Display prompt OR Update UI)
    }
```

5.16 Execute Dance Action

Invoke this function to execute a command `High five or Dance Action` in a chat room.

Precondition: The user must JOIN the room first with a call to Join Room. Otherwise you'll receive HTTP Status Code PreconditionFailed (412)

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#45c88ff5-4006-491a-b4d3-5f2ad542fa09

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val executeChatCmdResponse = withContext(Dispatchers.IO) {
        chatClient.executeChatCommand(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            request = ExecuteChatCommandRequest(
```

(continues on next page)

(continued from previous page)

```
                command = "/high5 georgew",
                userid = "023976080242ac120002", // ID of an existing user from this␣
↪chatroom
            )
        )
    }

    // Resolve `executeChatCmdResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.executeChatCommand(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    request = ExecuteChatCommandRequest(
        command = "/high5 georgew",
        userid = "023976080242ac120002", // ID of an existing user from this chatroom
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { executeChatCmdResponse ->
        // Resolve `executeChatCmdResponse` (ex. Display prompt OR Update UI)
    }
```

5.17 Reply to a Message (Threaded)

Invoke this function to create a threaded reply to another message event.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#d54ce72a-1a8a-4230-b950-0d1b345c20c6

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val sendThreadedReplyResponse = withContext(Dispatchers.IO) {
        chatClient.sendThreadedReply(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            replyTo = "0976280012ac00023242",   // ID of an existing event from this␣
↪chatroom, which you intend to reply to
            request = SendThreadedReplyRequest(
                body = "This is Jessy, replying to your greetings yow!!!",
```

(continues on next page)

(continued from previous page)

```
                userid = "023976080242ac120002", // ID of an existing user from this␣
↪chatroom
            )
        )
    }

    // Resolve `sendThreadedReplyResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.sendThreadedReply(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    replyTo = "0976280012ac00023242",   // ID of an existing event from this chatroom,␣
↪which you intend to reply to
    request = SendThreadedReplyRequest(
        body = "This is Jessy, replying to your greetings yow!!!",
        userid = "023976080242ac120002", // ID of an existing user from this chatroom
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { sendThreadedReplyResponse ->
        // Resolve `sendThreadedReplyResponse` (ex. Display prompt OR Update UI)
    }
```

5.18 Quote a Message

Invoke this function to quote an existing message and republishes it with a new message.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#c463cddd-c247-4e7c-8280-2d4880813149

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val sendQuotedReplyResponse = withContext(Dispatchers.IO) {
        chatClient.sendQuotedReply(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            replyTo = "0976280012ac00023242",   // ID of an existing event from this␣
↪chatroom, which you intend to reply to
            request = SendQuotedReplyRequest(
```

(continues on next page)

(continued from previous page)

```
                body = "This is Jessy, quoting your greetings yow!!!",
                userid = "023976080242ac120002", // ID of an existing user from this␣
↪chatroom
            )
        )
    }

    // Resolve `sendQuotedReplyResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.sendQuotedReply(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    replyTo = "0976280012ac00023242",   // ID of an existing event from this chatroom,␣
↪which you intend to reply to
    request = SendQuotedReplyRequest(
        body = "This is Jessy, quoting your greetings yow!!!",
        userid = "023976080242ac120002", // ID of an existing user from this chatroom
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { sendQuotedReplyResponse ->
        // Resolve `sendQuotedReplyResponse` (ex. Display prompt OR Update UI)
    }
```

5.19 React To A Message (“Like”)

Invoke this function to add or remove a reaction to an existing event.

After this completes, a new event appears in the stream representing the reaction. The new event will have an updated version of the event in the replyto field, which you can use to update your UI.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#977044d8-9133-4185-ac1f-4d96a40aa60b

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val reactToAMsgResponse = withContext(Dispatchers.IO) {
        chatClient.reactToEvent(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
```

(continues on next page)

(continued from previous page)

```
            eventId = "0976280012ac00023242",   // ID of an existing event from this␣
↪chatroom, which you intend to reply to
            request = ReactToAMessageRequest(
                userid = "023976080242ac120002", // ID of an existing user from this␣
↪chatroom
                reaction = "like",
                reacted = true
            )
        )
    }

    // Resolve `reactToAMsgResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.reactToEvent(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    eventId = "0976280012ac00023242",   // ID of an existing event from this chatroom,␣
↪which you intend to reply to
    request = ReactToAMessageRequest(
        userid = "023976080242ac120002", // ID of an existing user from this chatroom
        reaction = "like",
        reacted = true
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { reactToAMsgResponse ->
        // Resolve `reactToAMsgResponse` (ex. Display prompt OR Update UI)
    }
```

5.20 Report Message

Invoke this function to REPORT a message to the moderation team.

After this completes, a new event appears in the stream representing the reaction. The new event will have an updated version of the event in the replyto field, which you can use to update your UI.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#2b231a1e-a12b-4a2e-b7f3-7104bec91a0a

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
```

(continues on next page)

(continued from previous page)

```
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val reportMsgResponse = withContext(Dispatchers.IO) {
        chatClient.reportMessage(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            eventId = "0976280012ac00023242",   // ID of an existing event from this␣
↪chatroom, which you intend to reply to
            request = ReportMessageRequest(
                reporttype = "abuse",
                userid = "023976080242ac120002" // ID of an existing user from this␣
↪chatroom
            )
        )
    }

    // Resolve `reportMsgResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.reportMessage(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    eventId = "0976280012ac00023242",   // ID of an existing event from this chatroom,␣
↪which you intend to reply to
    request = ReportMessageRequest(
        reporttype = "abuse",
        userid = "023976080242ac120002" // ID of an existing user from this chatroom
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { reportMsgResponse ->
        // Resolve `reportMsgResponse` (ex. Display prompt OR Update UI)
    }
```

5.21 Execute Admin Command (*help)

Invoke this function to execute `help` command in a chat room.

Precondition: The user must JOIN the room first with a call to Join Room. Otherwise you'll receive HTTP Status Code PreconditionFailed (412)

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#08b0ab21-0e9f-40a3-bdfe-f228196fea03

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val executeChatCmdResponse = withContext(Dispatchers.IO) {
        chatClient.executeChatCommand(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            request = ExecuteChatCommandRequest(
                command = "*help*",
                userid = "023976080242ac120002", // ID of an existing user from this␣
↪chatroom
            )
        )
    }

    // Resolve `executeChatCmdResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.executeChatCommand(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    request = ExecuteChatCommandRequest(
        command = "*help*",
        userid = "023976080242ac120002", // ID of an existing user from this chatroom
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { executeChatCmdResponse ->
        // Resolve `executeChatCmdResponse` (ex. Display prompt OR Update UI)
    }
```

5.22 Get Updates

Invoke this function to get the recent updates to a room.

- You can use this function to poll the room to get the recent events in the room. The recommended poll interval is 500ms. Each event has an ID and a timestamp. To detect new messages using polling, call this function and then process items with a newer timestamp than the most recent one you have already processed.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#be93067d-562e-41b2-97b2-b2bf177f1282

If `smoothEventUpdates` is set to `true`, smooth event updates feature is applied. Smooth event updates feature emits event updates with space delay(denoted by `eventSpacingMs`) in between each item if item count is less than `maxEventBufferSize` to avoid overwhelming the receiver from consuming a list of event updates in small quantity. However, if item count exceeds `maxEventBufferSize`, all items are emitted as-is without space delay in between.

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
import com.sportstalk.coroutine.api.polling.allEventUpdates
// ...

// Under Fragment class
// ...
// User must first Join Chat Room
// Now that the test user has joined the room, setup reactive subscription to event␣
↪updates
// Below returns a Flow<List<ChatEvent>>
lifecycleScope.launch {
        chatClient.allEventUpdates(
            chatRoomId = testChatRoom.id!!,
            frequency = 1000L /* Polling Frequency. Defaults to 500 milliseconds if not␣
↪explicitly provided */,
            limit: Int? = null, // (optional) Number of events to return for each poll.␣
↪Default is 100, maximum is 500.
            /**
            * If [true], render events with some spacing.
            * - However, if we have a massive batch, we want to catch up, so we do not␣
↪put spacing and just jump ahead.
            */
            smoothEventUpdates: Boolean = true, // If not specified, defaults to [true]
            /**
            * (optional, 200ms by default) This only applies if `smoothEventUpdates` =␣
↪true.
            * This defines how long to pause before emitting the next event in a batch.
            */
            eventSpacingMs: Long = 200L, // If not specified or if negative number was␣
↪provided, defaults to 200ms
            /**
            * (optional, 30 by default) This only applies if `smoothEventUpdates` = true.
            * Holds the size of the event buffer we will accept before displaying␣
↪everything in order to catch up.
            */
            maxEventBufferSize: Int = 30,
            /*
            * The following are placeholder/convenience functions should the developers␣
↪want to implement it
            * in a callback-oriented way. (Invoked as subscription's side-effect. In␣
↪coroutine flow, these are invoked via .onEach { ... })
            */
            onChatEvent = { event: ChatEvent -> /* Handle all other eventtype */ }, //␣
↪OPTIONAL
            onGoalEvent = { goalEvent: ChatEvent -> /* Handle eventtype == "goal" */ }, /
↪/ OPTIONAL
            onAdEvent = { adEvent: ChatEvent -> /* Handle eventtype == "advertisement" */
↪ }, // OPTIONAL
            onReply = { replyEvent: ChatEvent -> /* Handle eventtype == "reply" */ }, //␣
↪OPTIONAL
            onReaction = { reactionEvent: ChatEvent -> /* Handle eventtype == "reaction"␣
↪*/ }, // OPTIONAL
```

(continues on next page)

(continued from previous page)

```
            onPurgeEvent = { purgeEvent: ChatEvent -> /* Handle eventtype == "purge" */ }
↪ // OPTIONAL
        )
    .distinctUntilChanged()
    .onEach { events ->
        // Alternatively, the developer can opt to consume the events in here...
        // NOTE:: ONLY choose 1 approach to avoid handling it twice.
        // Iterate each event item(s)
        events.forEach { chatEvent ->
            when(chatEvent.eventtype) {
                EventType.GOAL -> { /* Handle goal event types */ }
                EventType.ADVERTISEMENT -> { /* Handle advertisements event types */ }
                // ...
                // ...
            }
        }
    }
    .launchIn(lifecycleScope /* Already provided by androidx.Fragment */)

    // Then, perform start listening to event updates
    chatClient.startListeningToChatUpdates(
        forRoomId = testChatRoom.id!!
    )

    // At some point in time, the developer might want to explicitly stop listening to␣
↪event updates
    chatClient.stopListeningToChatUpdates(
        forRoomId = testChatRoom.id!!
    )
}
```

sdk-coroutine (LiveData)

```
/**
* ALTERNATIVELY, `sdk-coroutine` artifact also provides a
* similar function that returns a LiveData.
*/

import com.sportstalk.coroutine.api.polling.livedata.allEventUpdates
// ...

// Under Fragment class
// ...
// User must first Join Chat Room
// Now that the test user has joined the room, setup reactive subscription to event␣
↪updates
// Below returns a LiveData<List<ChatEvent>>
chatClient.allEventUpdates(
    chatRoomId = testChatRoom.id!!,
    lifecycleOwner = viewLifecycleOwner /* Already provided by androidx.Fragment */,
    frequency = 1000L /* Polling Frequency. Defaults to 500 milliseconds if not␣
↪explicitly provided */,
```

(continues on next page)

(continued from previous page)

```
    limit: Int? = null, // (optional) Number of events to return for each poll. Default␣
↪is 100, maximum is 500.
    /**
    * If [true], render events with some spacing.
    * - However, if we have a massive batch, we want to catch up, so we do not put␣
↪spacing and just jump ahead.
    */
    smoothEventUpdates: Boolean = true, // If not specified, defaults to [true]
    /**
    * (optional, 200ms by default) This only applies if `smoothEventUpdates` = true.
    * This defines how long to pause before emitting the next event in a batch.
    */
    eventSpacingMs: Long = 200L, // If not specified or if negative number was provided,␣
↪defaults to 200ms
    /**
    * (optional, 30 by default) This only applies if `smoothEventUpdates` = true.
    * Holds the size of the event buffer we will accept before displaying everything in␣
↪order to catch up.
    */
    maxEventBufferSize: Int = 30,
    /*
    * The following are placeholder/convenience functions should the developers want to␣
↪implement it
    * in a callback-oriented way. (Invoked as subscription's side-effect.)
    */
    onChatEvent = { event: ChatEvent -> /* Handle all other eventtype */ }, // OPTIONAL
    onGoalEvent = { goalEvent: ChatEvent -> /* Handle eventtype == "goal" */ }, //␣
↪OPTIONAL
    onAdEvent = { adEvent: ChatEvent -> /* Handle eventtype == "advertisement" */ }, //␣
↪OPTIONAL
    onReply = { replyEvent: ChatEvent -> /* Handle eventtype == "reply" */ }, // OPTIONAL
    onReaction = { reactionEvent: ChatEvent -> /* Handle eventtype == "reaction" */ }, //
↪ OPTIONAL
    onPurgeEvent = { purgeEvent: ChatEvent -> /* Handle eventtype == "purge" */ } //␣
↪OPTIONAL
)
.distinctUntilChanged()  // livedata-ktx
.observe(viewLifeCycleOwner, Observer { events ->
    // Alternatively, the developer can opt to consume the events in here...
    // NOTE:: ONLY choose 1 approach to avoid handling it twice.
    // Iterate each event item(s)
    events.forEach { chatEvent ->
        when(chatEvent.eventtype) {
            EventType.GOAL -> { /* Handle goal event types */ }
            EventType.ADVERTISEMENT -> { /* Handle advertisements event types */ }
            // ...
            // ...
        }
    }
})

// Then, perform start listening to event updates
```

(continues on next page)

(continued from previous page)

```
chatClient.startListeningToChatUpdates(
    forRoomId = testChatRoom.id!!
)

// At some point in time, the developer might want to explicitly stop listening to event␣
↪updates
chatClient.stopListeningToChatUpdates(
    forRoomId = testChatRoom.id!!
)
```

sdk-reactive-rx2

```
import com.sportstalk.reactive.rx2.api.polling.allEventUpdates
// ...

// Under Fragment class
// ...

val rxDisposeBag = CompositeDisposable()

// User must first Join Chat Room
// Now that the test user has joined the room, setup reactive subscription to event␣
↪updates
// Below returns a Flowable<List<ChatEvent>>
chatClient.allEventUpdates(
    chatRoomId = testChatRoom.id!!,
    lifecycleOwner = viewLifecycleOwner /* Already provided by androidx.Fragment */,
    frequency = 1000L /* Polling Frequency. Defaults to 500 milliseconds if not␣
↪explicitly provided */,
    limit: Int? = null, // (optional) Number of events to return for each poll. Default␣
↪is 100, maximum is 500.
    /**
    * If [true], render events with some spacing.
    * - However, if we have a massive batch, we want to catch up, so we do not put␣
↪spacing and just jump ahead.
    */
    smoothEventUpdates: Boolean = true, // If not specified, defaults to [true]
    /**
    * (optional, 200ms by default) This only applies if `smoothEventUpdates` = true.
    * This defines how long to pause before emitting the next event in a batch.
    */
    eventSpacingMs: Long = 200L, // If not specified or if negative number was provided,␣
↪defaults to 200ms
    /**
    * (optional, 30 by default) This only applies if `smoothEventUpdates` = true.
    * Holds the size of the event buffer we will accept before displaying everything in␣
↪order to catch up.
    */
    maxEventBufferSize: Int = 30,
    /*
    * The following are placeholder/convenience functions should the developers want to␣
↪implement it
```

(continues on next page)

(continued from previous page)

```
    * in a callback-oriented way. (Invoked as subscription's side-effect. In RxJava,␣
↪these are invoked via .doOnNext { ... }.)
    */
    onChatEvent = { event: ChatEvent -> /* Handle all other eventtype */ }, // OPTIONAL
    onGoalEvent = { goalEvent: ChatEvent -> /* Handle eventtype == "goal" */ }, //␣
↪OPTIONAL
    onAdEvent = { adEvent: ChatEvent -> /* Handle eventtype == "advertisement" */ }, //␣
↪OPTIONAL
    onReply = { replyEvent: ChatEvent -> /* Handle eventtype == "reply" */ }, // OPTIONAL
    onReaction = { reactionEvent: ChatEvent -> /* Handle eventtype == "reaction" */ }, //
↪ OPTIONAL
    onPurgeEvent = { purgeEvent: ChatEvent -> /* Handle eventtype == "purge" */ } //␣
↪OPTIONAL
)
.distinctUntilChanged()
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe { events ->
    // Alternatively, the developer can opt to consume the events in here...
    // NOTE:: ONLY choose 1 approach to avoid handling it twice.
    // Iterate each event item(s)
    events.forEach { chatEvent ->
        when(chatEvent.eventtype) {
            EventType.GOAL -> { /* Handle goal event types */ }
            EventType.ADVERTISEMENT -> { /* Handle advertisements event types */ }
            // ...
            // ...
        }
    }
}
.also { rxDisposeBag.add(it) }

// Then, perform start listening to event updates
chatClient.startListeningToChatUpdates(
    forRoomId = testChatRoom.id!!
)

// At some point in time, the developer might want to explicitly stop listening to event␣
↪updates
chatClient.stopListeningToChatUpdates(
    forRoomId = testChatRoom.id!!
)
```

5.23 List Messages By User

Invoke this function to get a list of users messages.

This method requires authentication.

The purpose of this method is to get a list of messages or comments by a user, with count of replies and reaction data. This way, you can easily make a screen in your application that shows the user a list of their comment contributions and how people reacted to it.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#0ec044c6-a3c0-478f-985a-156f6f5b660a

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listUserMessages = withContext(Dispatchers.IO) {
        chatClient.listMessagesByUser(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            userid = "023976080242ac120002", // ID of an existing user from this chatroom
            limit = 20, /* Defaults to 200 on backend API server */
            cursor = null // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of user message(s).
        )
    }

    // Resolve `listUserMessages` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.listMessagesByUser(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    userid = "023976080242ac120002", // ID of an existing user from this chatroom
    limit = 20, /* Defaults to 200 on backend API server */
    cursor = null // OPTIONAL: The cursor value from previous search attempt to indicate␣
↪next paginated fetch. Null if fetching the first list of user message(s).
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listUserMessages ->
        // Resolve `listUserMessages` (ex. Display prompt OR Update UI)
    }
```

5.24 List Event History

Invoke this function to list events history.

- This method enables you to download all of the events from a room in large batches. It should only be used if doing a data export.
- This method returns a list of events sorted from oldest to newest.
- This method returns all events, even those in the inactive state.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#b8ca9766-ab07-4c8c-8e25-002a24a8feaa

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listEventsHistory = withContext(Dispatchers.IO) {
        chatClient.listEventsHistory(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            limit = 20, /* Defaults to 200 on backend API server */
            cursor = null // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of events.
        )
    }

    // Resolve `listEventsHistory` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.listEventsHistory(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    limit = 20, /* Defaults to 200 on backend API server */
    cursor = null // OPTIONAL: The cursor value from previous search attempt to indicate␣
↪next paginated fetch. Null if fetching the first list of events.
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listEventsHistory ->
        // Resolve `listEventsHistory` (ex. Display prompt OR Update UI)
    }
```

5.25 List Events By Type

Invoke this function to list events by type.

- This method enables you to retrieve a small list of recent events by type. This is useful for things like fetching a list of recent announcements or custom event types without the need to scroll through the entire chat history.
- This method returns a list of events sorted from newest to oldest.
- This method returns only active events.
- If you specify eventtype = customtype, you must pass the customtype value, a string of your choosing for your custom type.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#68a36454-bf36-41e0-b8ef-6bcb2a13dd36

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listEventsByType = withContext(Dispatchers.IO) {
        chatClient.listEventsByType(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            eventType = EventType.ANNOUNCEMENT, // "announcement"
            limit = 20, /* Defaults to 200 on backend API server */
            cursor = null // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of events.
        )
    }

    // Resolve `listEventsByType` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.listEventsByType(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    eventType = EventType.ANNOUNCEMENT, // "announcement"
    limit = 20, /* Defaults to 200 on backend API server */
    cursor = null // OPTIONAL: The cursor value from previous search attempt to indicate␣
↪next paginated fetch. Null if fetching the first list of events.
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listEventsByType ->
        // Resolve `listEventsByType` (ex. Display prompt OR Update UI)
    }
```

5.26 List Events By Timestamp

Invoke this function to list events by timestamp.

- This method enables you to retrieve an event using a timestamp.
- You can optionally retrieve a small number of displayable events before and after the message at the requested timestamp.
- This method returns a list of events sorted from oldest to newest.
- This method returns only active events.
- The timestamp is a high resolution timestamp accurate to the thousanth of a second. It is possible, but very unlikely, for two messages to have the same timestamp.
- The method returns "timestampolder". This can be passed as the timestamp value when calling functions like this which accept a timestamp to retrieve data.
- The method returns "timestampnewer". This can be passed as the timestamp value when calling this function again.
- The method returns "cursorpolder". This can be passed as the cursor to ethods that accept an events-sorted-by-time cursor.
- The method returns "cursornewer". This can be passed as the cursor to methods that accept an events-sorted-by-time cursor.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#fe87c58e-2fd3-4e59-80fa-07ffaed94ee0

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listEventsByTimestamp = withContext(Dispatchers.IO) {
        chatClient.listEventsByTimestamp(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            timestamp = 637464818548698844, // Timestamp criteria
            limitolder = 5,
            limitolder = 5
        )
    }

    // Resolve `listEventsByTimestamp` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.listEventsByTimestamp(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
```

(continues on next page)

(continued from previous page)

```
    timestamp = 637464818548698844, // Timestamp criteria
    limitolder = 5,
    limitolder = 5
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listEventsByTimestamp ->
        // Resolve `listEventsByTimestamp` (ex. Display prompt OR Update UI)
    }
```

5.27 Message is Reported

Invoke this function to check if the current user has already reported a message.

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // NO NEED to switch coroutine context as this operation does NOT perform network␣
↪call
    val messageIsReported = chatClient.messageIsReported(
        eventId = "7620812242ac09300002"    // ID of an existing event from the chat room
        userid = "023976080242ac120002", // ID of an existing user from the chat room
    )

    // Resolve `messageIsReported` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
// This function just returns Boolean value rather than an RxJava type since this␣
↪operation does NOT perform network call
val messageIsReported = chatClient.messageIsReported(
    eventId = "7620812242ac09300002"    // ID of an existing event from the chat room
    userid = "023976080242ac120002", // ID of an existing user from the chat room
)

// Resolve `messageIsReported` (ex. Display prompt OR Update UI)
```

5.28 Message is Reacted To

Invoke this function to check if a message was reacted to by the current user.

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // NO NEED to switch coroutine context as this operation does NOT perform network␣
↪call
    val messageIsReactedTo = chatClient.messageIsReactedTo(
        eventId = "7620812242ac09300002"    // ID of an existing event from the chat room
        userid = "023976080242ac120002", // ID of an existing user from the chat room
        reaction = "like"   // One of the EventReaction string constants
    )

    // Resolve `messageIsReactedTo` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
// This function just returns Boolean value rather than an RxJava type since this␣
↪operation does NOT perform network call
val messageIsReactedTo = chatClient.messageIsReactedTo(
    eventId = "7620812242ac09300002"    // ID of an existing event from the chat room
    userid = "023976080242ac120002", // ID of an existing user from the chat room
    reaction = "like"   // One of the EventReaction string constants
)

// Resolve `messageIsReactedTo` (ex. Display prompt OR Update UI)
```

5.29 List Previous Events

Invoke this function to list previous events.

- This method allows you to go back in time to "scroll" in reverse through past messages. The typical use case for this method is to power the scroll-back feature of a chat window allowing the user to look at recent messages that have scrolled out of view. It's intended use is to retrieve small batches of historical events as the user is scrolling up.
- This method returns a list of events sorted from newest to oldest.
- This method excludes events that are not in the active state (for example if they are removed by a moderator)
- This method excludes non-displayable events (reaction, replace, remove, purge)
- This method will not return events that were emitted and then deleted before this method was called

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#f750f610-5db8-46ca-b9f7-a800c2e9c94a

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listPreviousEvents = withContext(Dispatchers.IO) {
        chatClient.listPreviousEvents(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            limit = 20, /* Defaults to 200 on backend API server */
            cursor = null // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of events.
        )
    }

    // Resolve `listPreviousEvents` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.listPreviousEvents(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    limit = 20, /* Defaults to 200 on backend API server */
    cursor = null // OPTIONAL: The cursor value from previous search attempt to indicate␣
↪next paginated fetch. Null if fetching the first list of events.
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listPreviousEvents ->
        // Resolve `listPreviousEvents` (ex. Display prompt OR Update UI)
    }
```

5.30 Get Event by ID

Invoke this function to get a chat event by ID.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#04f8f563-eacf-4a64-9f00-b3d6c050a2fa

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val chatEventResponse = withContext(Dispatchers.IO) {
        chatClient.getEventById(
```

(continues on next page)

(continued from previous page)

```
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            eventId = "7620812242ac09300002"    // ID of an existing event from the chat␣
↪room
        )
    }

    // Resolve `chatEventResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.getEventById(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    eventId = "7620812242ac09300002"    // ID of an existing event from the chat room
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { chatEventResponse ->
        // Resolve `chatEventResponse` (ex. Display prompt OR Update UI)
    }
```

5.31 Report User In Room

Invoke this function to enable users to report other users who exhibit abusive behaviors. It enables users to silence another user when a moderator is not present. If the user receives too many reports in a trailing 24 hour period, the user will become flagged at the room level.

This API moderates users on the ROOM LEVEL. If a There is an API method that enable reporting users at the global user level which impacts all rooms. This API impacts only the experience for the specified userid within the specified room.

This API will return an error (see responses below) if user reporting is not enabled for your application in the application settings by setting User Reports limit to a value > 0.

A user who is flagged will have the shadowban effect applied.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#94fdf593-06b6-41a2-80f6-79b8eb989b8b

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val reportUserInRoomResponse = withContext(Dispatchers.IO) {
        chatClient.reportUserInRoom(
```

(continues on next page)

(continued from previous page)

```
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            request = ReportUserInRoomRequest(
                userid = "023976080242ac120002", // ID of an existing user from the chat␣
↪room
                reporttype = ReportType.ABUSE   // either ReportType.ABUSE("abuse") or␣
↪ReportType.SPAM("spam")
            )
        )
    }

    // Resolve `reportUserInRoomResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.reportUserInRoom(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    request = ReportUserInRoomRequest(
        userid = "023976080242ac120002", // ID of an existing user from the chat room
        reporttype = ReportType.ABUSE   // either ReportType.ABUSE("abuse") or␣
↪ReportType.SPAM("spam")
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { reportUserInRoomResponse ->
        // Resolve `reportUserInRoomResponse` (ex. Display prompt OR Update UI)
    }
```

5.32 Purge User Messages

Invoke this function to execute a command in a chat room to purge all messages for a user.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#04ffee45-a3e6-49b8-8968-46b219020b66

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val purgeCmdResponse = withContext(Dispatchers.IO) {
        chatClient.executeChatCommand(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
```

(continues on next page)

(continued from previous page)

```
            // Assuming ADMIN password "testpassword123"
            // Assuming user "@nicoleWd" exists
            request = ExecuteChatCommandRequest(
                command = "*purge testpassword123 nicoleWd",
                userid = "023976080242ac120002" // ID of an existing user "@nicoleWd"␣
↪from this chatroom
            )
        )
    }

    // Resolve `purgeCmdResponse` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.executeChatCommand(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    // Assuming ADMIN password "testpassword123"
    // Assuming user "@nicoleWd" exists
    request = ExecuteChatCommandRequest(
        command = "*purge testpassword123 nicoleWd",
        userid = "023976080242ac120002" // ID of an existing user "@nicoleWd" from this␣
↪chatroom
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { chatEventResponse ->
        // Resolve `chatEventResponse` (ex. Display prompt OR Update UI)
    }
```

5.33 Shadow Ban User (In Room Only)

Invoke this function to toggle the user's shadow banned flag from within the specified Chatroom.

There is a user level shadow ban (global) and local room level shadow ban.

A Shadow Banned user can send messages into a chat room, however those messages are flagged as shadow banned. This enables the application to show those messags only to the shadow banned user, so that that person may not know they were shadow banned. This method shadow bans the user on the global level (or you can use this method to lift the ban). You can optionally specify an expiration time. If the expiration time is specified, then each time the shadow banned user tries to send a message the API will check if the shadow ban has expired and will lift the ban.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#c4a83dfa-9e83-4eb8-b371-e105463f3a52

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val shadowBanUserResponse = withContext(Dispatchers.IO) {
        chatClient.shadowBanUser(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            // Assuming user "@nicoleWd" exists
            userid = "023976080242ac120002", // ID of an existing user "@nicoleWd" from␣
↪this chatroom
            applyeffect = true, // If set to true, user will be set to banned state.␣
↪Otherwise, will be set to non-banned state.
            expireseconds = 3600 // [OPTIONAL]: Duration of shadowban value in seconds.␣
↪If specified, the shadow ban will be lifted when this time is reached. If not␣
↪specified, shadowban remains until explicitly lifted. Maximum seconds is a double byte␣
↪value.
        )
    }

    // Resolve `shadowBanUserResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.shadowBanUser(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    // Assuming user "@nicoleWd" exists
    userid = "023976080242ac120002", // ID of an existing user "@nicoleWd" from this␣
↪chatroom
    applyeffect = true, // If set to true, user will be set to banned state. Otherwise,␣
↪will be set to non-banned state.
    expireseconds = 3600 // [OPTIONAL]: Duration of shadowban value in seconds. If␣
↪specified, the shadow ban will be lifted when this time is reached. If not specified,␣
↪shadowban remains until explicitly lifted. Maximum seconds is a double byte value.
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { shadowBanUserResponse ->
        // Resolve `shadowBanUserResponse` (ex. Display prompt OR Update UI)
    }
```

5.34 Mute User (In Room Only)

Invoke this function toggles the specified user's mute effect.

There is a global user mute effect and local room level user mute effect.

A muted user is in a read-only state. The muted user cannot communicate. This method applies mute from within the specified Chat room ONLY. You can optionally specify an expiration time. If the expiration time is specified, then each time the muted user tries to send a message the API will check if the effect has expired and will lift the user's mute effect.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#67d66190-eb25-4f19-9d65-c127ed368233

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val muteUserResponse = withContext(Dispatchers.IO) {
        chatClient.muteUser(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            // Assuming user "@nicoleWd" exists
            userid = "023976080242ac120002", // ID of an existing user "@nicoleWd" from␣
↪this chatroom
            applyeffect = true, // If set to true, user will be set to muted state.␣
↪Otherwise, will be set to non-banned state.
            expireseconds = 3600 // [OPTIONAL]: Duration of mute in seconds. If␣
↪specified, the mute will be lifted when this time is reached. If not specified, mute␣
↪effect remains until explicitly lifted. Maximum seconds is a double byte value.
        )
    }

    // Resolve `muteUserResponse` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.muteUser(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    // Assuming user "@nicoleWd" exists
    userid = "023976080242ac120002", // ID of an existing user "@nicoleWd" from this␣
↪chatroom
    applyeffect = true, // If set to true, user will be set to muted state. Otherwise,␣
↪will be set to non-banned state.
    expireseconds = 3600 // [OPTIONAL]: Duration of mute in seconds. If specified, the␣
↪mute will be lifted when this time is reached. If not specified, mute effect remains␣
↪until explicitly lifted. Maximum seconds is a double byte value.
)
```

(continues on next page)

(continued from previous page)

```
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { muteUserResponse ->
        // Resolve `muteUserResponse` (ex. Display prompt OR Update UI)
    }
```

5.35 Bounce User

Invoke this function to remove the user from the room and prevent the user from reentering.

Optionally display a message to people in the room indicating this person was bounced.

When you bounce a user from the room, the user is removed from the room and blocked from reentering. Past events generated by that user are not modified (past messages from the user are not removed).

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#7116d7ca-a1b8-44c1-8894-bea85225e4c7

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val bounceUserResponse = withContext(Dispatchers.IO) {
        chatClient.bounceUser(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            // Assuming user "@nicoleWd" exists
            request = BounceUserRequest(
                userid = "023976080242ac120002", // ID of an existing user "@nicoleWd"␣
↪from this chatroom
                bounce = true,
                announcement = "@nicoleWd has been banned."
            )
        )
    }

    // Resolve `bounceUserResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.bounceUser(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    // Assuming user "@nicoleWd" exists
    request = BounceUserRequest(
```

(continues on next page)

(continued from previous page)

```
        userid = "023976080242ac120002", // ID of an existing user "@nicoleWd" from this␣
↪chatroom
        bounce = true,
        announcement = "@nicoleWd has been banned."
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { bounceUserResponse ->
        // Resolve `bounceUserResponse` (ex. Display prompt OR Update UI)
    }
```

5.36 Search Event History

Invoke this function to search the message history applying the specified filters.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#a6b5380c-4e6c-4ded-b0b1-55225bcdea67

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)

    // Search using User ID
    val searchEventHistoryByUserIdResponse = withContext(Dispatchers.IO) {
        chatClient.searchEventHistory(
            request = SearchEventHistoryRequest(
                fromuserid = "023976080242ac120002", // ID of an existing user
                limit = 10,
                types = listOf(EventType.SPEECH) // Any EventType constants
            )
        )
    }
    // Resolve `searchEventHistoryFromUserIdResponse` from HERE onwards(ex. update UI␣
↪displaying the response data)...

    // Search using User handle
    val searchEventHistoryByUserHandleResponse = withContext(Dispatchers.IO) {
        chatClient.searchEventHistory(
            request = SearchEventHistoryRequest(
                fromhandle = "@nicoleWD", // Handle of an existing user
                limit = 10,
                types = listOf(EventType.SPEECH) // Any EventType constants
            )
        )
```

(continues on next page)

(continued from previous page)

```
    }
    // Resolve `searchEventHistoryByUserHandleResponse` from HERE onwards(ex. update UI␣
↪displaying the response data)...

}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

// Search using User ID
chatClient.searchEventHistory(
    request = SearchEventHistoryRequest(
        fromuserid = "023976080242ac120002", // ID of an existing user
        limit = 10,
        types = listOf(EventType.SPEECH) // Any EventType constants
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { searchEventHistoryByUserIdResponse ->
        // Resolve `searchEventHistoryByUserIdResponse` (ex. Display prompt OR Update UI)
    }

// Search using User handle
chatClient.searchEventHistory(
    request = SearchEventHistoryRequest(
        fromhandle = "@nicoleWD", // Handle of an existing user
        limit = 10,
        types = listOf(EventType.SPEECH) // Any EventType constants
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { searchEventHistoryByUserHandleResponse ->
        // Resolve `searchEventHistoryByUserHandleResponse` (ex. Display prompt OR Update␣
↪UI)
    }
```

5.37 Update Chat Message

Invoke this function to update the contents of an existing chat event

This API may be used to update the body of an existing Chat Event. It is used to enable the user to edit the message after it is published. This may only be used with MESSAGE event types (speech, quote, reply). When the chat event is updated another event of type “replace” will be emitted with the updated event contents, and the original event will be replaced in future calls to List Event History, Join and List Previous Events. The event will also be flagged as edited by user.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#207a7dfa-5233-4acb-b855-031928941b25

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val updateChatMessageResponse = withContext(Dispatchers.IO) {
        chatClient.updateChatMessage(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            eventId = "7620812242ac09300002",    // ID of an existing event from the␣
↪chat room
            request = UpdateChatMessageRequest(
                userid = "023976080242ac120002", // ID of an existing user from this␣
↪chat room
                body = "[UPDATED] from the original message",
                customid = null, // [OPTIONAL]
                custompayload = null, // [OPTIONAL]
                customfield1 = null, // [OPTIONAL]
                customfield2 = null, // [OPTIONAL]
                customtags = null, // [OPTIONAL]
            )
        )
    }

    // Resolve `updateChatMessageResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.updateChatMessage(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    eventId = "7620812242ac09300002",    // ID of an existing event from the chat room
    request = UpdateChatMessageRequest(
        userid = "023976080242ac120002", // ID of an existing user from this chat room
        body = "[UPDATED] from the original message",
        customid = null, // [OPTIONAL]
        custompayload = null, // [OPTIONAL]
        customfield1 = null, // [OPTIONAL]
        customfield2 = null, // [OPTIONAL]
        customtags = null, // [OPTIONAL]
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { updateChatMessageResponse ->
        // Resolve `updateChatMessageResponse` (ex. Display prompt OR Update UI)
    }
```

5.38 Flag Message Event As Deleted

Invoke this function to set a ChatEvent as logically deleted.

Everything in a chat room is an event. Each event has a type. Events of type "speech, reply, quote" are considered "messages".

Use logical delete if you want to flag something as deleted without actually deleting the message so you still have the data. When you use this method:

- The message is not actually deleted. The comment is flagged as deleted, and can no longer be read, but replies are not deleted.
- If flag "permanentifnoreplies" is true, then it will be a permanent delete instead of logical delete for this comment if it has no children.
- If you use "permanentifnoreplies" = true, and this comment has a parent that has been logically deleted, and this is the only child, then the parent will also be permanently deleted (and so on up the hierarchy of events).

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#92632caf-9bd0-449d-91df-90fef54f6634

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val logicalDeleteResponse = withContext(Dispatchers.IO) {
        chatClient.flagEventLogicallyDeleted(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            eventId = "7620812242ac09300002",    // ID of an existing event from the␣
↪chat room
            userid = "023976080242ac120002", // ID of an existing user "@nicoleWd" from␣
↪this chatroom
            // Assuming user "@nicoleWd" exists
            deleted = true,
            permanentifnoreplies = true
        )
    }

    // Resolve `logicalDeleteResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.flagEventLogicallyDeleted(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    eventId = "7620812242ac09300002",    // ID of an existing event from the chat room
    userid = "023976080242ac120002", // ID of an existing user "@nicoleWd" from this␣
↪chatroom
```

(continues on next page)

(continued from previous page)

```
    // Assuming user "@nicoleWd" exists
    deleted = true,
    permanentifnoreplies = true
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { logicalDeleteResponse ->
        // Resolve `logicalDeleteResponse` (ex. Display prompt OR Update UI)
    }
```

5.39 Delete Event

Invoke this function to delete an event from the room.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#f2894c8f-acc9-4b14-a8e9-216b28c319de

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val deleteEventResponse = withContext(Dispatchers.IO) {
        chatClient.permanentlyDeleteEvent(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            eventId = "7620812242ac09300002",    // ID of an existing event from the␣
↪chat room
            userid = "023976080242ac120002" // ID of an existing user "@nicoleWd" from␣
↪this chatroom
            // Assuming user "@nicoleWd" exists
        )
    }

    // Resolve `deleteEventResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.permanentlyDeleteEvent(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    eventId = "7620812242ac09300002",    // ID of an existing event from the chat room
    userid = "023976080242ac120002" // ID of an existing user "@nicoleWd" from this␣
↪chatroom
    // Assuming user "@nicoleWd" exists
```

(continues on next page)

(continued from previous page)

```
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { deleteEventResponse ->
        // Resolve `deleteEventResponse` (ex. Display prompt OR Update UI)
    }
```

5.40 Delete All Events in Room

Invoke this function to execute a command in a chat room to delete all messages in the chatroom.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#e4d62330-469e-4e37-a42e-049b10259152

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val deleteAllEventsCmdResponse = withContext(Dispatchers.IO) {
        chatClient.executeChatCommand(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            // Assuming ADMIN password "testpassword123"
            request = ExecuteChatCommandRequest(
                command = "*deleteallevents testpassword123",
                userid = "023976080242ac120002" // ID of an existing user "@nicoleWd"␣
↪from this chatroom
            )
        )
    }

    // Resolve `deleteAllEventsCmdResponse` from HERE onwards(ex. update UI displaying␣
↪the response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.executeChatCommand(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    // Assuming ADMIN password "testpassword123"
    request = ExecuteChatCommandRequest(
        command = "*deleteallevents testpassword123",
        userid = "023976080242ac120002" // ID of an existing user "@nicoleWd" from this␣
↪chatroom
    )
```

(continues on next page)

(continued from previous page)

```
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { deleteAllEventsCmdResponse ->
        // Resolve `deleteAllEventsCmdResponse` (ex. Display prompt OR Update UI)
    }
```

5.41 Update Room (Close a room)

Invoke this function to update an existing room.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#e4d62330-469e-4e37-a42e-049b10259152

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val updatedRoomResponse = withContext(Dispatchers.IO) {
        chatClient.updateRoom(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            request = UpdateChatRoomRequest(
                    name = "Test Chat Room 1 - UPDATED",
                    customid = "test-room-1-updated",
                    description = "[UPDATED] This is a test chat room 1.",
                    moderation = "post",
                    enableactions = false,
                    enableenterandexit = false,
                    enableprofanityfilter = true,
                    delaymessageseconds = 10L,
                    roomisopen = false,
                    maxreports = 30
            )
        )
    }

    // Resolve `updatedRoomResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.updateRoom(
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
```

(continues on next page)

(continued from previous page)

```
    request = UpdateChatRoomRequest(
            name = "Test Chat Room 1 - UPDATED",
            customid = "test-room-1-updated",
            description = "[UPDATED] This is a test chat room 1.",
            moderation = "post",
            enableactions = false,
            enableenterandexit = false,
            enableprofanityfilter = true,
            delaymessageseconds = 10L,
            roomisopen = false,
            maxreports = 30
    )
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { updatedRoomResponse ->
        // Resolve `updatedRoomResponse` (ex. Display prompt OR Update UI)
    }
```

5.42 Exit a Room

Invoke this function to exit from a chatroom where the user has currently joined.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#408b43ca-fca9-4f2d-8883-f6f725d140f2

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val exitRoomResponse = withContext(Dispatchers.IO) {
        chatClient.exitRoom(
            chatRoomId = "080001297623242ac002",    // ID of an existing chat room
            userid = "023976080242ac120002" // ID of an existing user from this chatroom
        )
    }

    // Resolve `exitRoomResponse` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.exitRoom(
```

(continues on next page)

(continued from previous page)

```
    chatRoomId = "080001297623242ac002",    // ID of an existing chat room
    userid = "023976080242ac120002" // ID of an existing user from this chatroom
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { exitRoomResponse ->
        // Resolve `exitRoomResponse` (ex. Display prompt OR Update UI)
    }
```

5.43 Delete Room

Invoke this function to delete the specified room and all events contained therein) by ID

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#c5ae345d-004d-478a-b543-5abaf691000d

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val deleteRoomResponse = withContext(Dispatchers.IO) {
        chatClient.deleteRoom(
            chatRoomId = "080001297623242ac002"    // ID of an existing chat room
        )
    }

    // Resolve `deleteRoomResponse` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.deleteRoom(
    chatRoomId = "080001297623242ac002"    // ID of an existing chat room
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { deleteRoomResponse ->
        // Resolve `deleteRoomResponse` (ex. Display prompt OR Update UI)
    }
```

5.44 List Messages Needing Moderation

Invoke this function to list all the messages in the moderation queue.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#bcdbda1b-e495-46c9-8fe9-c5dc6a4c1756

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val listMessagesInModeration = withContext(Dispatchers.IO) {
        chatClient.listMessagesNeedingModeration(
            roomId = "080001297623242ac002",    // ID of an existing chat room
            limit = 20, /* Defaults to 200 on backend API server */
            cursor = null // OPTIONAL: The cursor value from previous search attempt to␣
↪indicate next paginated fetch. Null if fetching the first list of messages from this␣
↪chatroom.
        )
    }

    // Resolve `listMessagesInModeration` from HERE onwards(ex. update UI displaying the␣
↪response data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.listMessagesNeedingModeration(
    roomId = "080001297623242ac002",    // ID of an existing chat room
    limit = 20, /* Defaults to 200 on backend API server */
    cursor = null // OPTIONAL: The cursor value from previous search attempt to indicate␣
↪next paginated fetch. Null if fetching the first list of messages from this chatroom.
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { listMessagesInModeration ->
        // Resolve `listMessagesInModeration` (ex. Display prompt OR Update UI)
    }
```

5.45 Approve Message

Invoke this function to approve a message in the moderation queue.

Refer to the SportsTalk API Documentation for more details:

https://apiref.sportstalk247.com/?version=latest#6f9bf714-5b3b-48c9-87d2-eb2e12d2bcbf

Below is a code sample on how to use this SDK feature:

sdk-coroutine

```
// Launch thru coroutine block
// https://developer.android.com/topic/libraries/architecture/coroutines
lifecycleScope.launch {
    // Switch to IO Coroutine Context(Operation will be executed on IO Thread)
    val approveResponse = withContext(Dispatchers.IO) {
        chatClient.approveMessage(
            eventId = "0976280012ac00023242",   // ID of an existing event from this␣
↪chatroom, which you intend to reply to
            approve = true
        )
    }

    // Resolve `approveResponse` from HERE onwards(ex. update UI displaying the response␣
↪data)...
}
```

sdk-reactive-rx2

```
val rxDisposeBag = CompositeDisposable()

chatClient.approveMessage(
    eventId = "0976280012ac00023242",   // ID of an existing event from this chatroom,␣
↪which you intend to reply to
    approve = true
)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { rxDisposeBag.add(it) }
    .subscribe { approveResponse ->
        // Resolve `approveResponse` (ex. Display prompt OR Update UI)
    }
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search