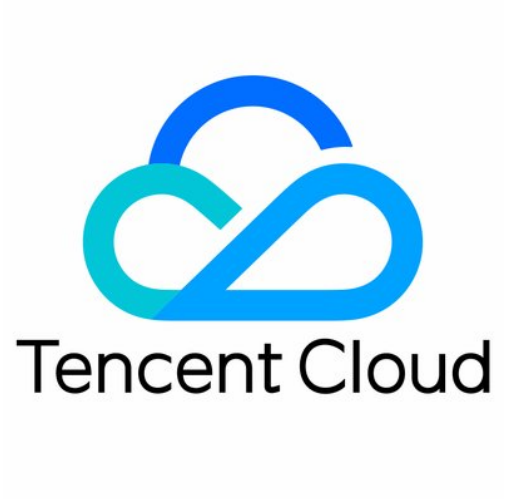
Cloud Streaming Services

SDK Guide

Product Documentation



Contents

SDK Guide

0. SDK Integration Guide

Last updated：2022-11-21 16:56:28

This document describes how to quickly integrate live push and stream pull solutions into your business program. The SDKs for different businesses are as listed below:

Feature Module		Mobile	Web	Documentation
Stream push		iOS / Android	Web	1. Stream Push
Playback		iOS / Android	-	2. Playback
Advanced features	AV1 video playback	AV1 Encoding		Overview
	High-quality upstreaming	TMIO SDK	-	
	LEB playback by player	Integrating libLebConnection SDK into a player	-	
	On-screen comments and chat	Integration guide	-	
	Beauty filters and special effects	iOS / Android	-	

1. Stream Push
Push Stream Integration Details

Last updated：2023-12-25 14:05:44

This document describes how to integrate the SDK or plugin into your program to implement the CSS stream push feature.

Prerequisites

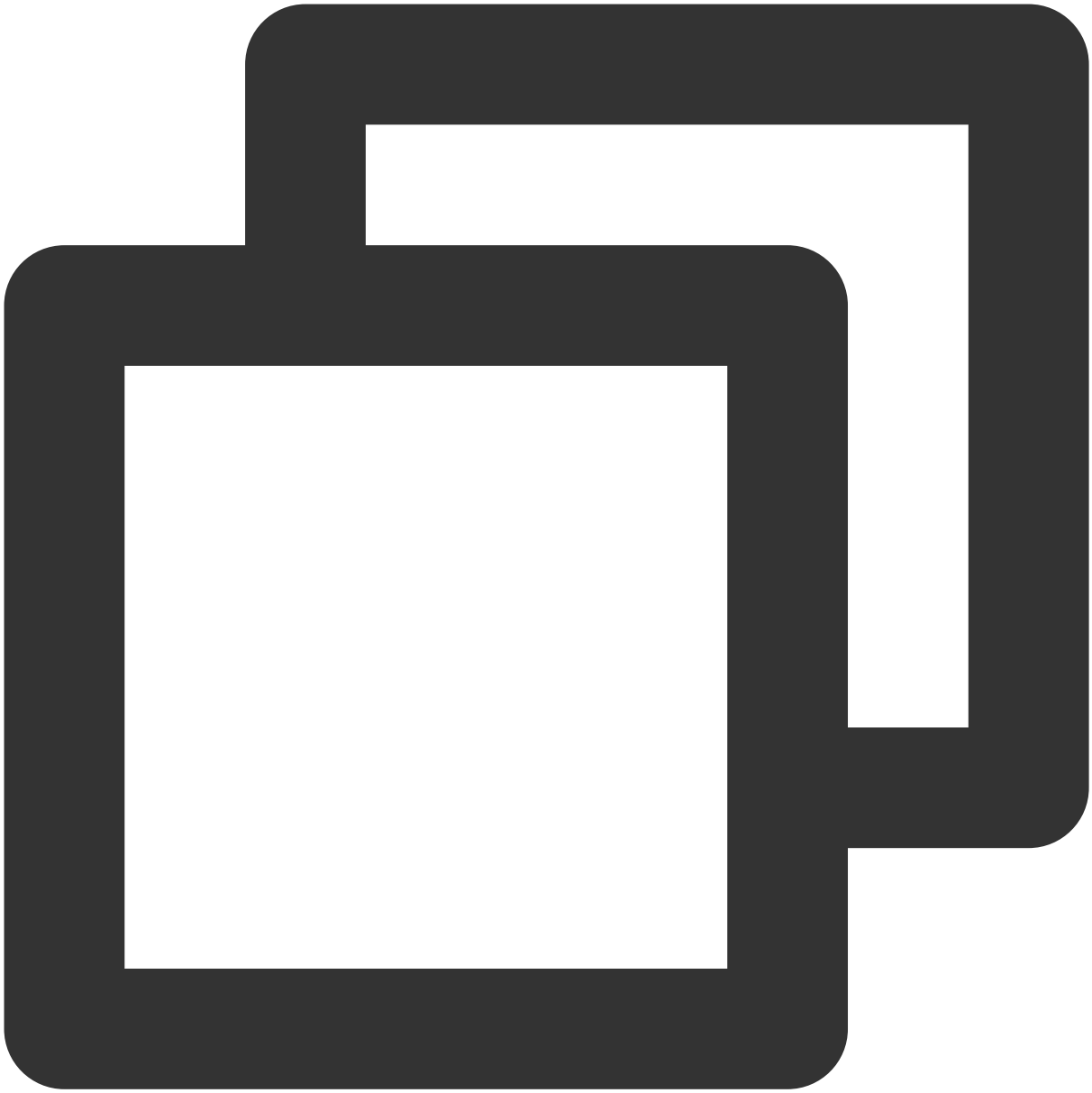
You have activated the CSS service.
Select **Domain Management**, click **Add Domain**, and add a push domain name as instructed in Adding Your Own Domain.
In the CSS console, generate a push address in **CSS Toolkit** > **Address Generator** as instructed in Address Generator. Then, implement live push in your own business based on your business scenarios as follows:

Integration into the native application

Download and integrate the MLVB SDK as instructed in the iOS and Android integration guides.

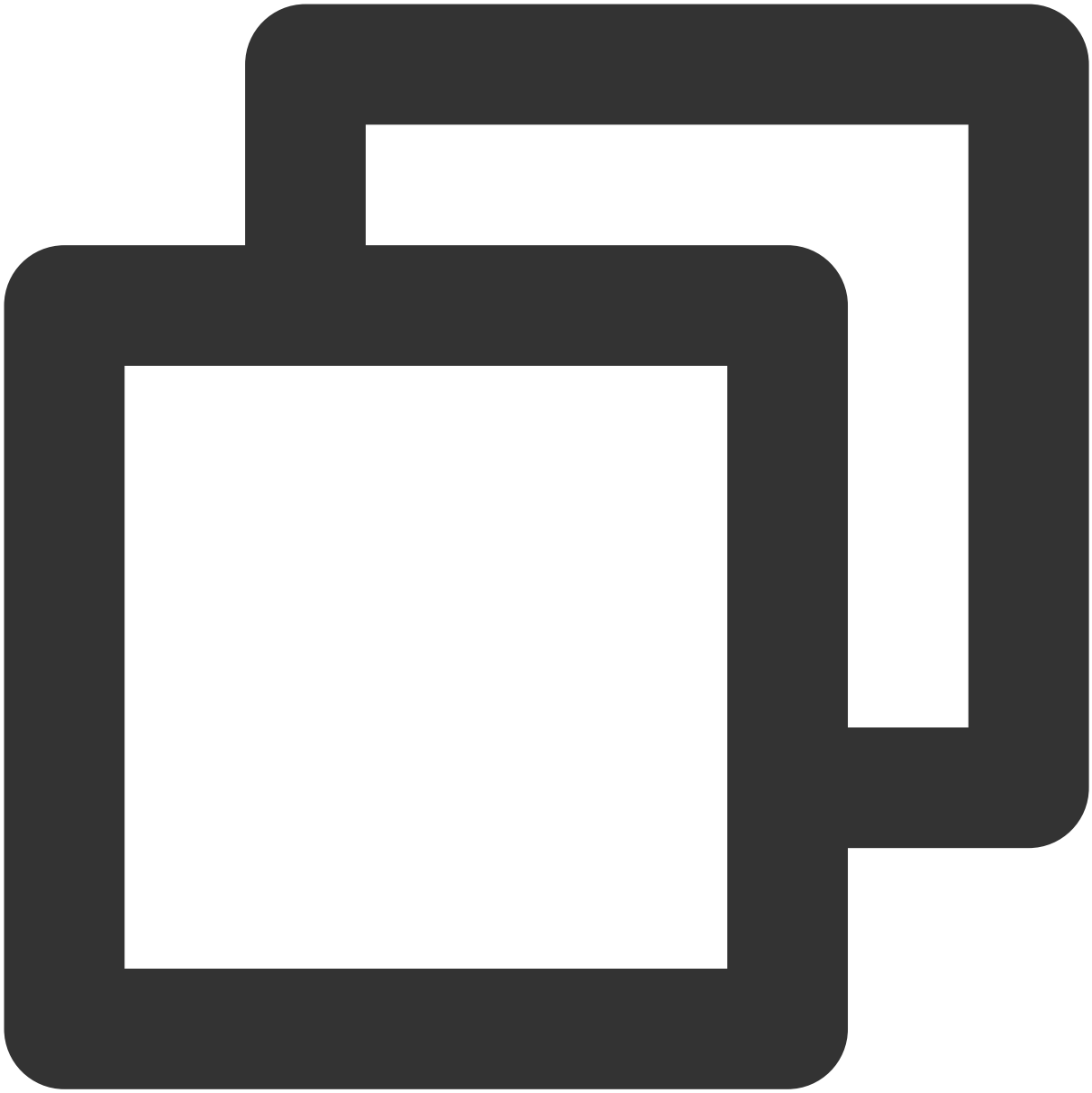
Note

To enable RTMP stream push, you need to create the `TXLivePusher` object and set `V2TXLiveMode` to `_RTMP` when initializing the `V2TXLivePusher` component. Below is the configuration for iOS and Android:
iOS



```
V2TXLivePusher *pusher = [[V2TXLivePusher alloc] initWithLiveMode:V2TXLiveMode_RTMP
```

Android



```
V2TXLivePusher mLivePusher = new V2TXLivePusherImpl(this, V2TXLiveDef.V2TXLiveMode.
```

Integration on the web

Integration on the web currently supports only the WebRTC protocol for stream push. You can integrate the SDK as instructed in WebRTC.

Note：

You can also directly perform web page live streaming in the Web Push section of the CSS Console.

Integration on PC

For PC (Windows/macOS), you can directly use OBS for stream push. It is a free open-source video shooting and streaming program that supports operating systems such as Windows, macOS, and Linux.
If the stream push protocol is WebRTC, you need to configure the OBS plugin provided by Tencent Cloud as instructed in OBS WebRTC live streaming.

More

Using the MLVB SDK will incur fees. For billing details, see Billing Overview.

Web Push SDK API

Last updated：2023-12-25 15:17:39

API Overview

TXLivePusher

Tencent Cloud CSS pusher is primarily used for browser-based web streaming. It captures the user's video and audio through the browser and utilizes WebRTC to transmit the video and audio streams to the Tencent Cloud CSS server.

API	Description
checkSupport	Static function is used to check browser compatibility.
setRenderView	Set the local video preview container.
setVideoQuality	Set Push Video Quality.
setAudioQuality	Set Push Audio Quality.
setProperty	Invoke Advanced API Interface.
startCamera	Open Camera Device.
stopCamera	Close Camera Device.
startMicrophone	Open Microphone Device.
stopMicrophone	Close Microphone Device.
startScreenCapture	Enable Screen Capture.
stopScreenCapture	Disable Screen Capture.
startVirtualCamera	Start Capturing Local Media File Stream.
stopVirtualCamera	Stop Capturing Local Media File Stream.
startCustomCapture	Use Custom Audio and Video Stream.
stopCustomCapture	Close Custom Audio and Video Stream.

startPush	Start Pushing Stream.
stopPush	Stop Pushing Stream.
isPushing	Check if Currently Pushing Stream.
getMediaStream	Get Audio and Video Stream by Stream ID.
getDeviceManager	Get Device Manager Object.
getVideoEffectManager	Get Video Effect Manager Object.
getAudioEffectManager	Get Audio Effect Manager Object.
setVideoMute	Set whether to disable video stream.
setAudioMute	Set whether to disable audio stream.
pauseVideo	Disable video stream.
pauseAudio	Disable audio stream.
resumeVideo	Restore video stream.
resumeAudio	Restore audio stream.
setVideoContentHint	Set video content hint to improve encoding quality in different content scenarios.
setObserver	Set push stream event callback notification.
destroy	When leaving the page or exiting, clean up the SDK instance.

TXDeviceManager

Device Management Interface, mainly used for managing camera and microphone devices, performing device acquisition and switching operations.

API	Description
getDevicesList	Get Device List.
getCurrentDevice	Get device information of the current stream.
switchDevice	Switch the currently used device.
switchCamera	Switch camera device.

switchMicrophone	Switch microphone device.

TXAudioEffectManager

Audio Effect Management Interface, mainly used for adjusting volume operations.

API	Description
setVolume	Set the volume level of the audio stream.

TXVideoEffectManager

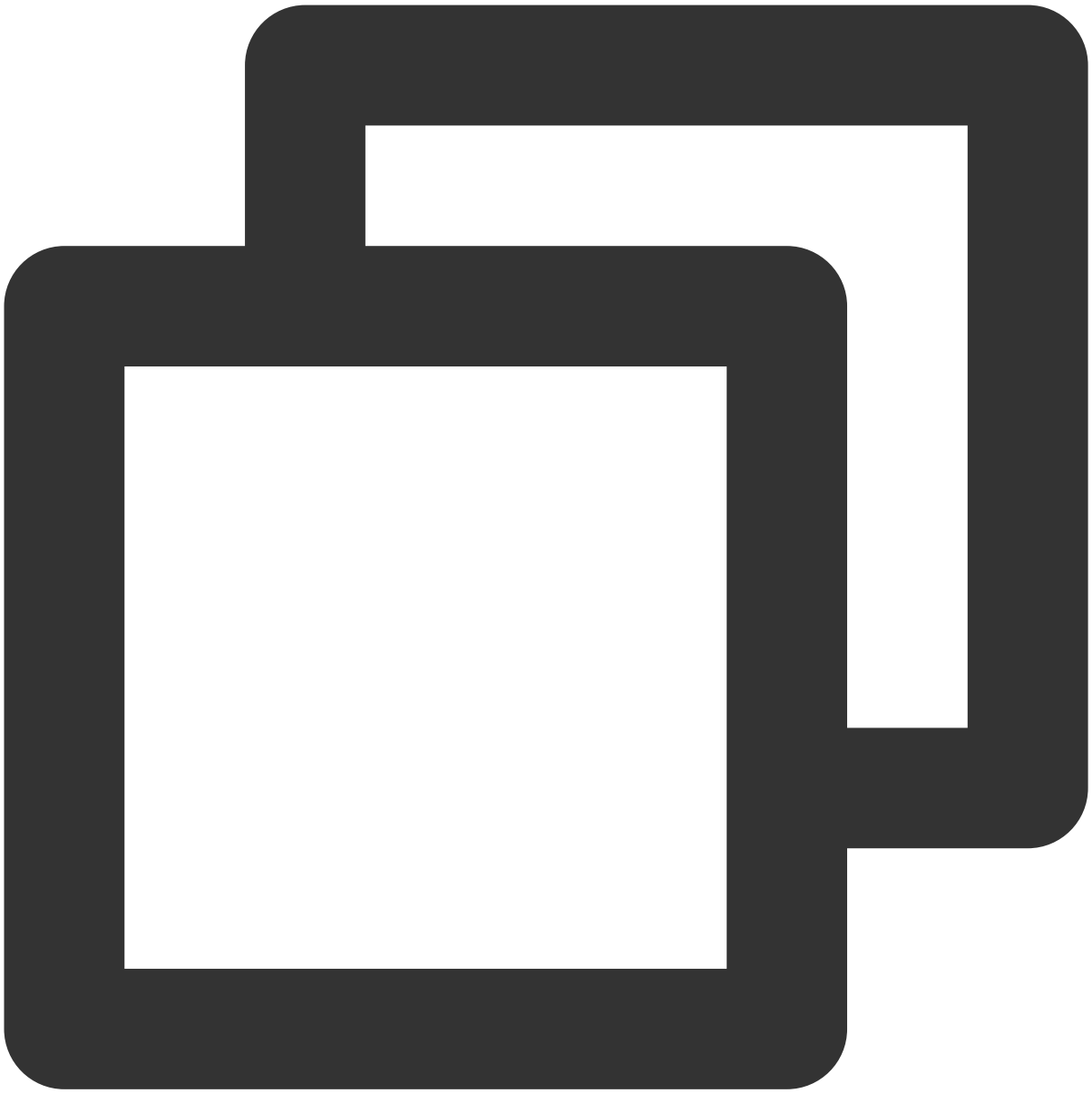
Video Effect Management Interface, mainly used for setting Picture-in-Picture, mirror, filter, watermark, text, and other operations.

API	Description
enableMixing	Enable local video screen mixing feature.
setMixingConfig	Set mixing parameters.
getMixingConfig	Get the final adopted mixing parameters.
setLayout	Set Picture-in-Picture layout parameters for the video stream.
getLayout	Get Picture-in-Picture layout parameters for the specified stream.
setMirror	Set mirror effect for the video stream.
setNormalFilter	Set general filter effect for the video stream.
setWatermark	Set watermark.
setText	Set text.

TXLivePusher

Tencent Cloud CSS Pusher, mainly used for browser-based Web streaming. It captures the user's image and sound through the browser and pushes the video and audio streams to the Tencent Cloud server via WebRTC. If you need to enable the local mixing feature, call the TXVideoEffectManager method enableMixing() to enable it.

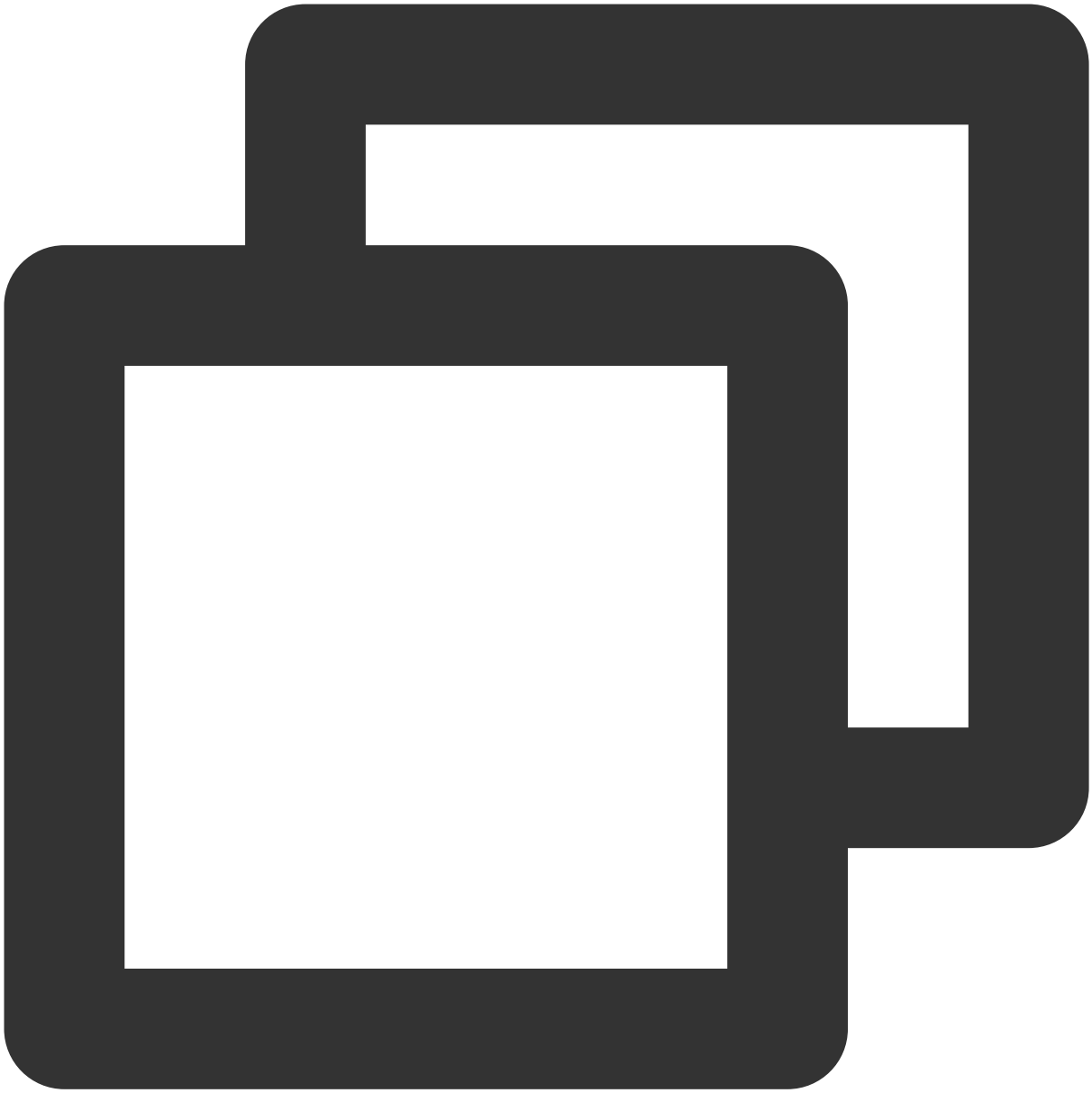
Please create an instance object first for all subsequent operations.



```
const livePusher = new TXLivePusher();
```

checkSupport

Static function, check browser compatibility.



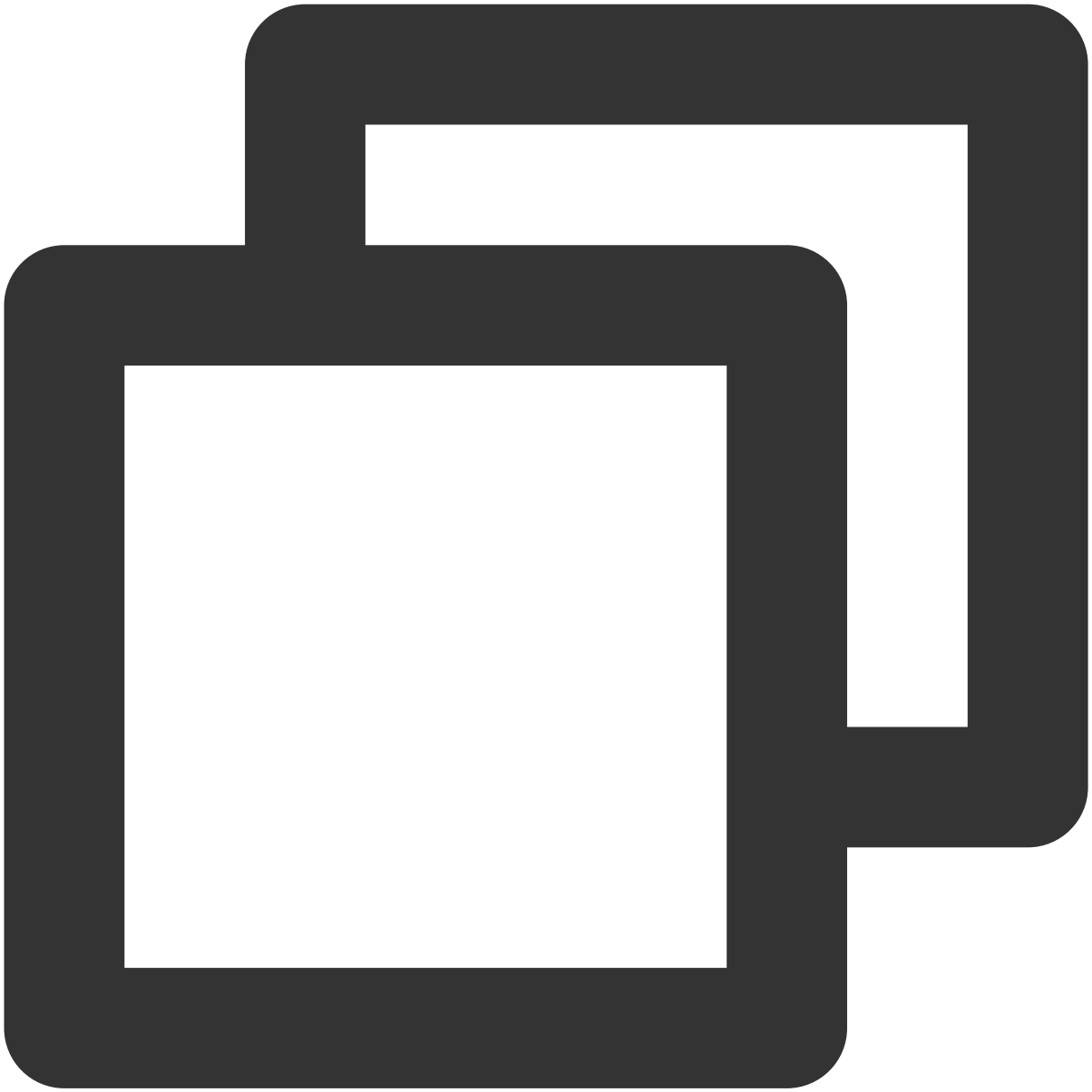
```
static checkSupport(): Promise<TXSupportResult>;
```

Return:

Return Promise object, where the result data structure is referred to as TXSupportResult.

setRenderView

Set the preview container for the local video screen, provide a div node, and the locally captured video will be rendered in the container. If the local mixing feature is enabled, the container will render the mixed audio and video after processing.



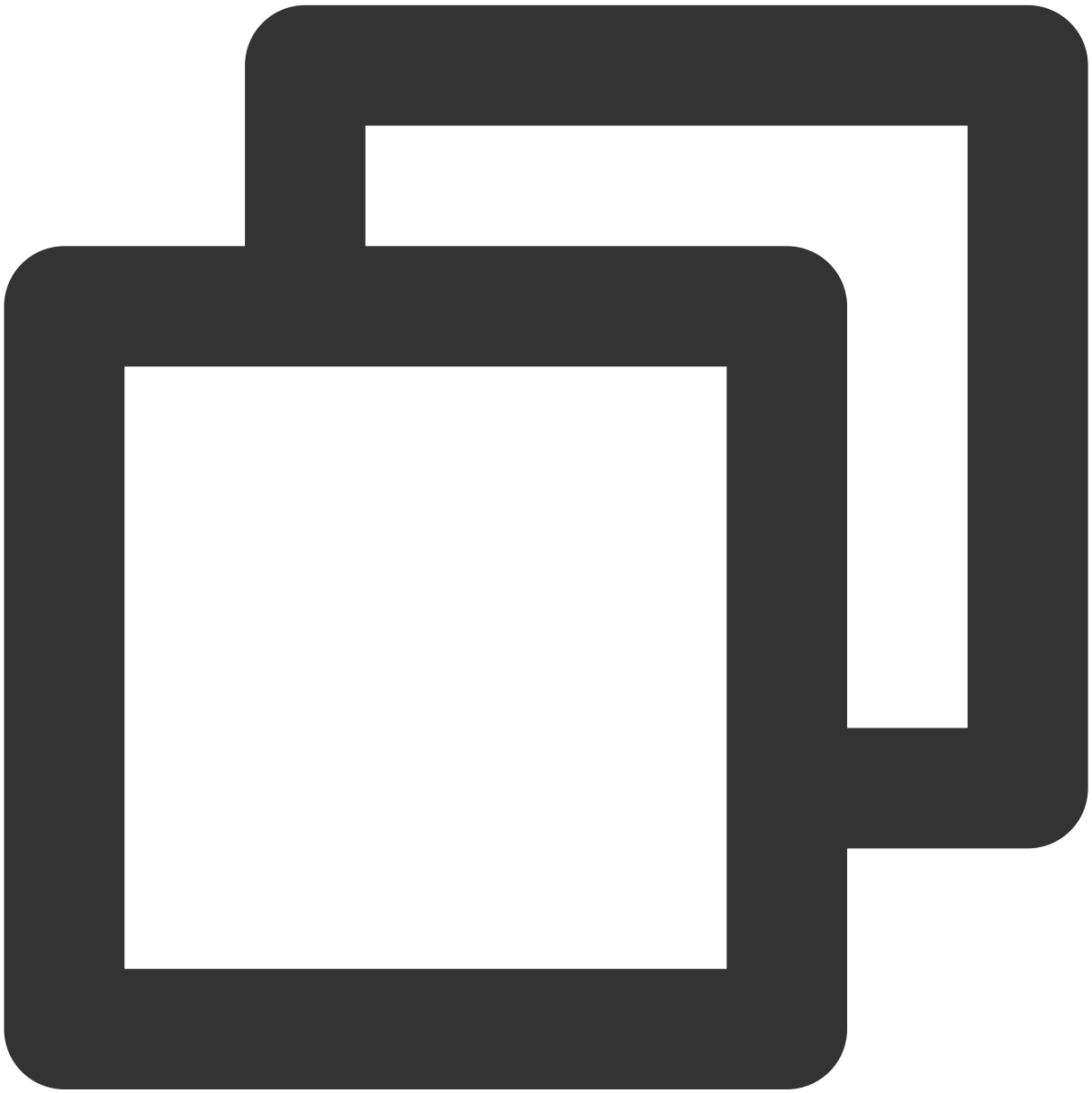
```
setRenderView(container: string | HTMLDivElement): void;
```

Parameter:

Field	Type	Description
container	string | HTMLDivElement	Container ID or DOM node.

setVideoQuality

Set push video quality. The SDK has built-in video quality templates, allowing you to set the push video quality directly through predefined templates.



```
setVideoQuality(quality: string): void;
```

Parameter:

Field	Type	Description

quality	string	Predefined video quality template names.

The built-in video quality templates are as follows:

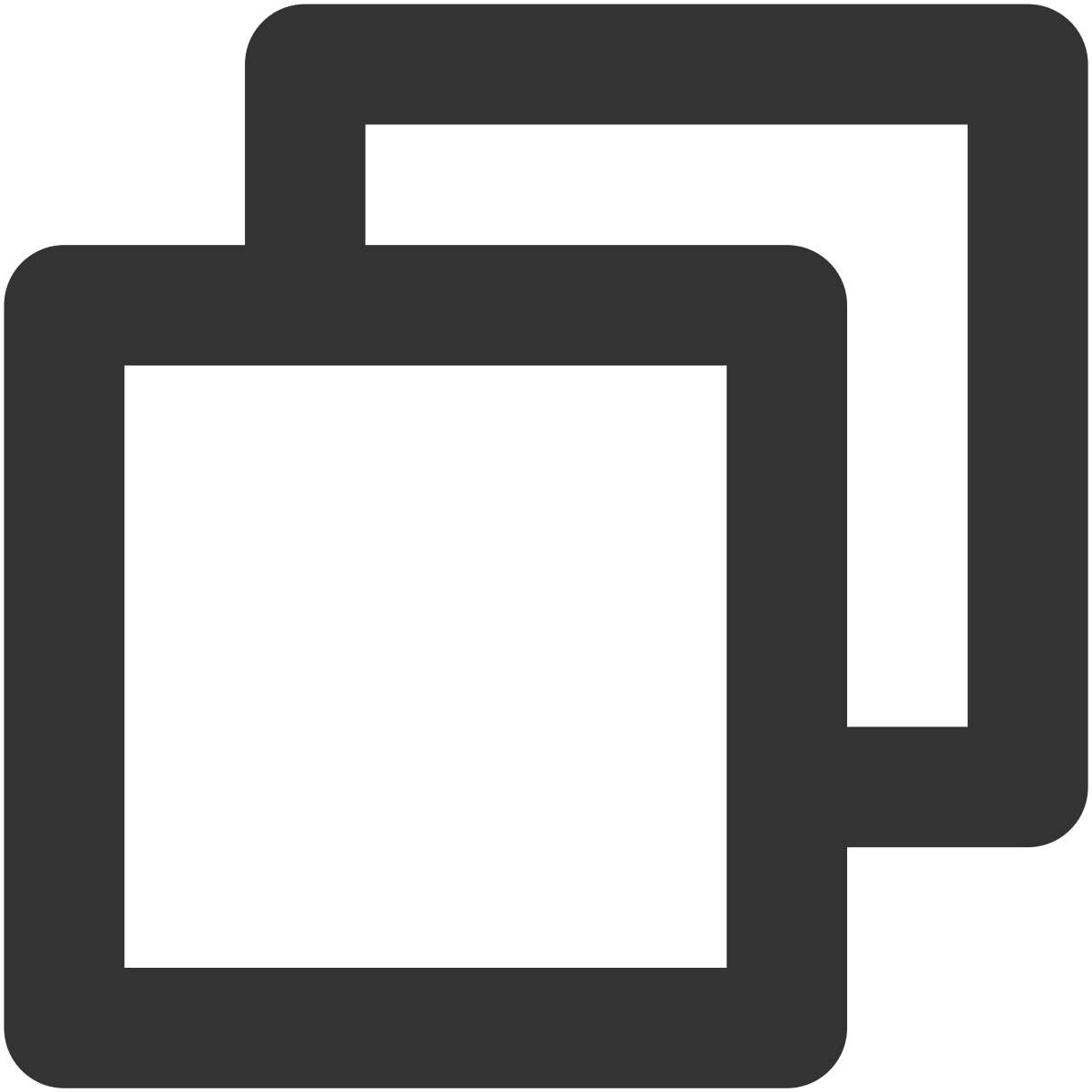
Template Name	Resolution (Width x Height)	Frame Rate (fps)	Bitrate (kbps)
120p	160 x 120	15	200
180p	320 x 180	15	350
240p	320 x 240	15	400
360p	640 x 360	15	800
480p	640 x 480	15	900
720p	1280 x 720	15	1500
1080p	1920 x 1080	15	2000
2K	2560 x 1440	30	4860
4K	3840 x 2160	30	9000

Note:

1. Due to device and browser limitations, the video resolution may not be perfectly matched. In this case, the browser will automatically adjust the resolution to be close to the corresponding resolution.
2. If the video quality parameters (resolution, frame rate, and bitrate) do not meet your requirements, you can set custom values individually using setProperty().
3. The video resolution here mainly refers to the resolution of the locally captured video. The resolution during streaming may be lower than the captured resolution, and the browser will automatically adjust the streaming resolution based on network bandwidth and other factors.
4. The default is 720p, i.e. `setVideoQuality('720p')` .

setAudioQuality

Set the streaming audio quality. The SDK has an inbuilt audio quality template. The streaming audio quality can be set directly by using predefined templates.



```
setAudioQuality(quality: string): void;
```

The parameters are described as follows:

Field	Type	Description
quality	string	The pre-defined name of the audio quality template.

The built-in audio quality templates are as follows:

Template Name	Sample rate	Bitrate (Kbps)

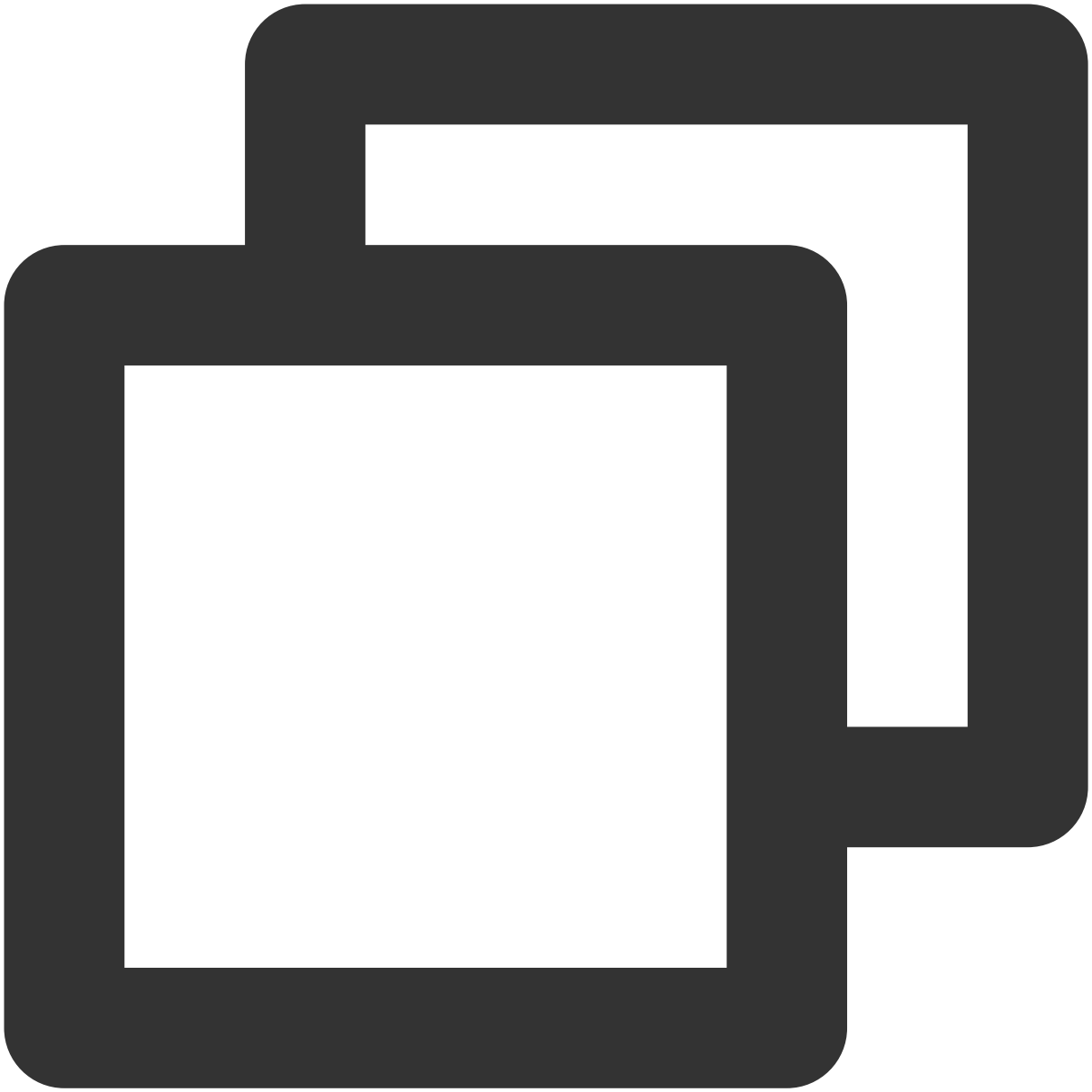
standard	48000	40
high	48000	128

Note:

1. If the audio quality parameters (sampling rate and bitrate) do not meet your requirements, you can individually set custom values through setProperty().
2. The default is to use 'standard', that is, `setAudioQuality('standard')` .

setProperty

Primarily used for invoking advanced features, such as setting the resolution, frame rate and bitrate of the video, as well as setting the sampling rate and bitrate of the audio, among others.



```
setProperty(key: string, value: any): void;
```

The parameters are described as follows:

Field	Type	Description
key	string	Key corresponding to the advanced API.
value	*	Parameters needed when invoking the key corresponding to the advanced API.

The following advanced features are currently supported:

Key	Value	Description	Sample code
setVideoResolution	{ width: number; height:number; }	Establish the video resolution	setProperty('setVideoResolution', { width: 1920, height: 1080 })
setVideoFPS	number	Configure the video frame rate	setProperty('setVideoFPS', 25)
setVideoBitrate	number	Set the video bitrate	setProperty('setVideoBitrate', 2000)
setAudioSampleRate	number	Configure the audio sample rate	setProperty('setAudioSampleRate', 44100)
setAudioBitrate	number	Establish the audio bitrate	setProperty('setAudioBitrate', 200)
setConnectRetryCount	number	Set the retry count for connections, default is: 3; range: 0 - 10. When the SDK unexpectedly disconnects with the server, it will attempt to reconnect.	setProperty('setConnectRetryCount', 5)
setConnectRetryDelay	number	Set the connection retry delay, the default value is: 1, in seconds; within the range: 0 - 10. When the SDK is anomalously disconnected from the server, it endeavors to reconnect.	setProperty('setConnectRetryDelay', 2)
enableAudioAEC	boolean	Enable echo cancellation	setProperty('enableAudioAEC', true)
enableAudioAGC	boolean	Enable automatic gain	setProperty('enableAudioAGC', true)
enableAudioANS	boolean	Enable noise suppression	setProperty('enableAudioANS', true)
enableLog	boolean	Should logs be printed	setProperty('enableLog', true)

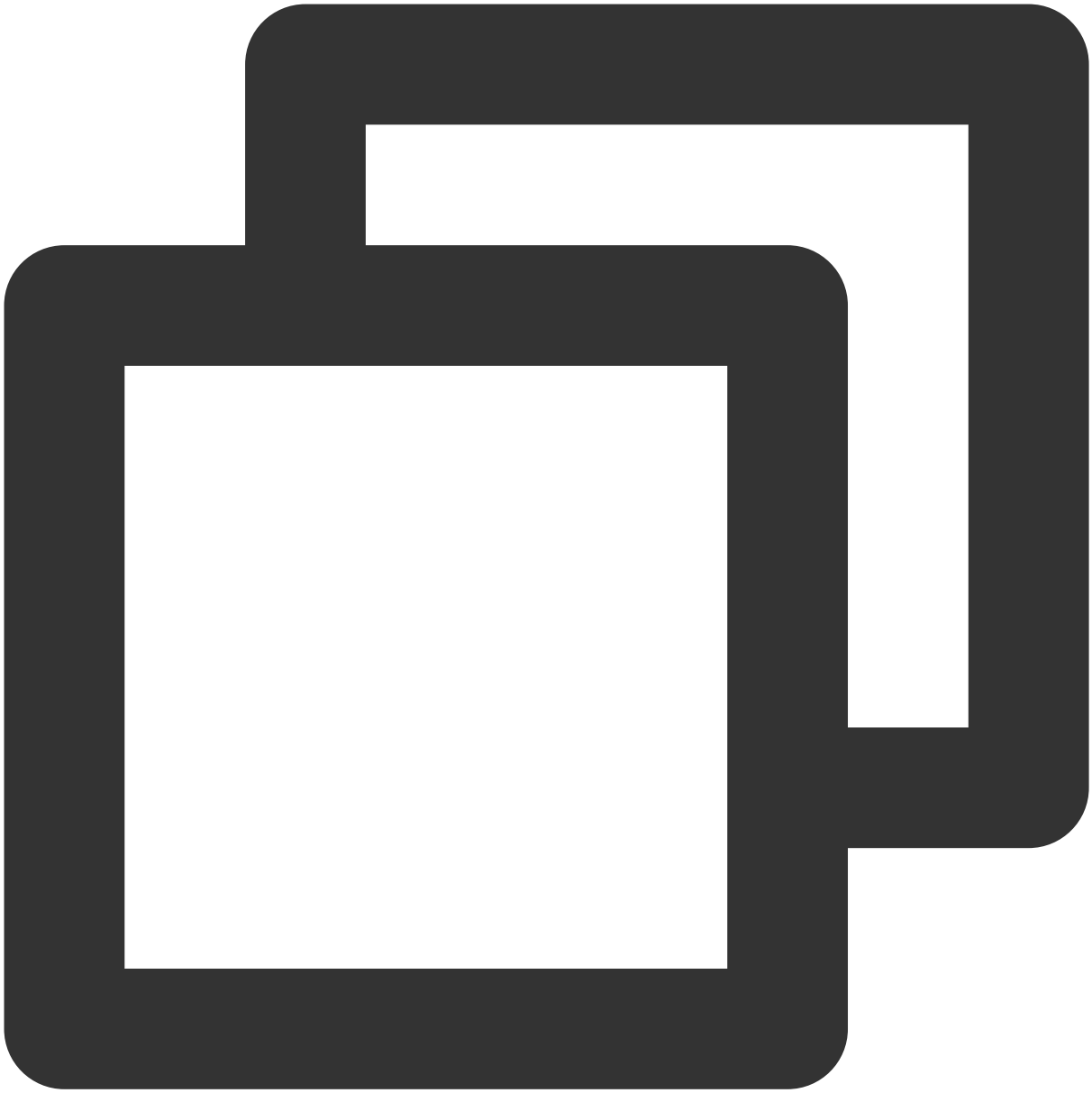
		in the console	

Note:

1. Echo cancellation, automatic gain and noise suppression are enabled by default, their effectiveness ultimately depends on the device and browser. It is recommended that these three functions either be enabled or disabled collectively.
2. Please set up before capturing and pushing the stream.

startCamera

Initiate camera device. User authorization is required to allow the browser access to the camera. If authorization fails or device access fails, the returned Promise object will throw an error.



```
startCamera(deviceId?: string): Promise<string>;
```

The parameters are described as follows:

Field	Type	Description
deviceId	string	Camera device ID, an optional parameter, determines the camera device to open. The device ID can be obtained using the method getDevicesList in `TXDeviceManager`. On mobile devices, the front and rear cameras can be specified by entering 'user' and 'environment'.

Return:

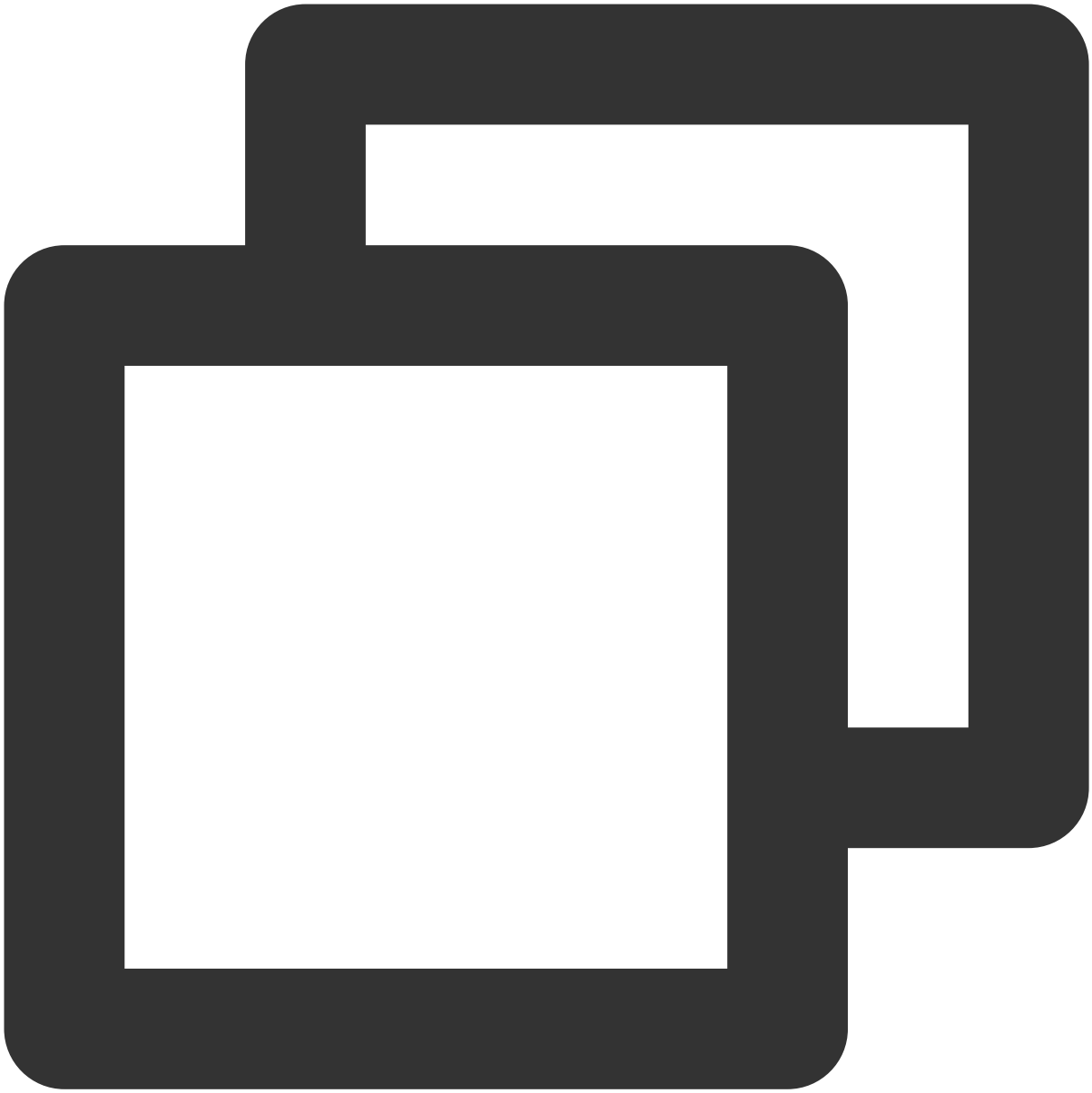
Yields a Promise object. Upon success, ID of the stream is returned, serving as the unique identification within SDK. In case of failure, the corresponding error message is thrown.

Note:

1. This interface does not support usage under the HTTP protocol; please deploy your website using the HTTPS protocol.
2. Upon encountering a failure to open the camera, refer to the error message returned, consulting getUserMedia Exception as needed.

stopCamera

The operation of shutting down the camera apparatus.



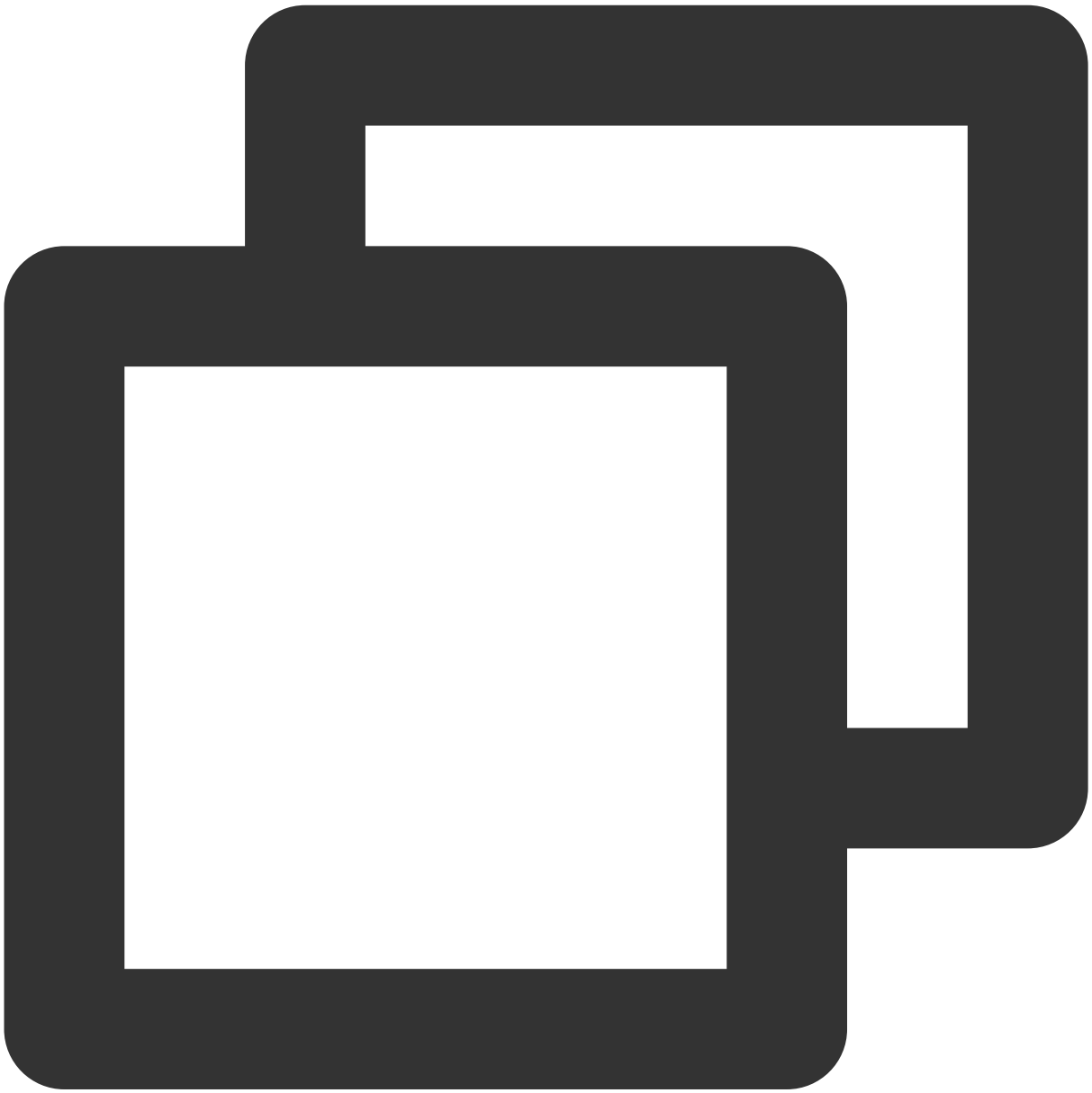
```
stopCamera(streamId?: string): void;
```

The parameters are described as follows:

Field	Type	Description
streamId	string	Stream ID, an optional parameter, signifies the specific camera stream to be deactivated. After the local stream mix is enabled, if multiple camera streams have been collected, you can turn off the specified camera stream through the stream ID; otherwise, it will close all camera streams.

startMicrophone

Activate microphone devices. It requires user authorization to access the microphone through the browser. If authorization fails, or the device is inaccessible, the returned Promise object will throw an error.



```
startMicrophone(deviceId?: string): Promise<string>;
```

The parameters are described as follows:

Field	Type	Description

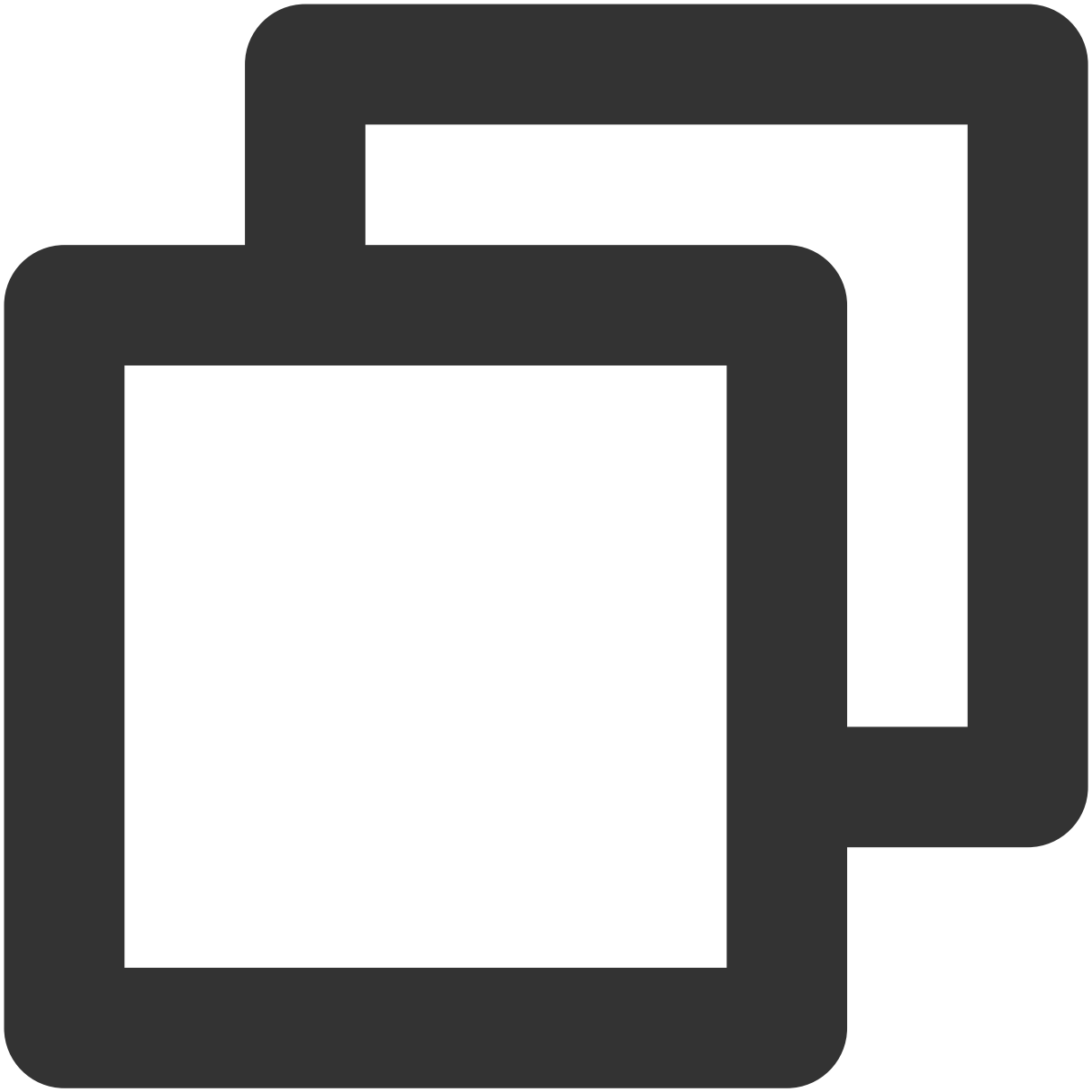
deviceId	string	Microphone Device ID, an optional parameter designating the specific microphone equipment to be activated. The Device ID can be retrieved through the method getDevicesList() found in `TXDeviceManager` .

Return:

Yields a Promise object. Upon success, ID of the stream is returned, serving as the unique identification within SDK. In case of failure, the corresponding error message is thrown.

Note:

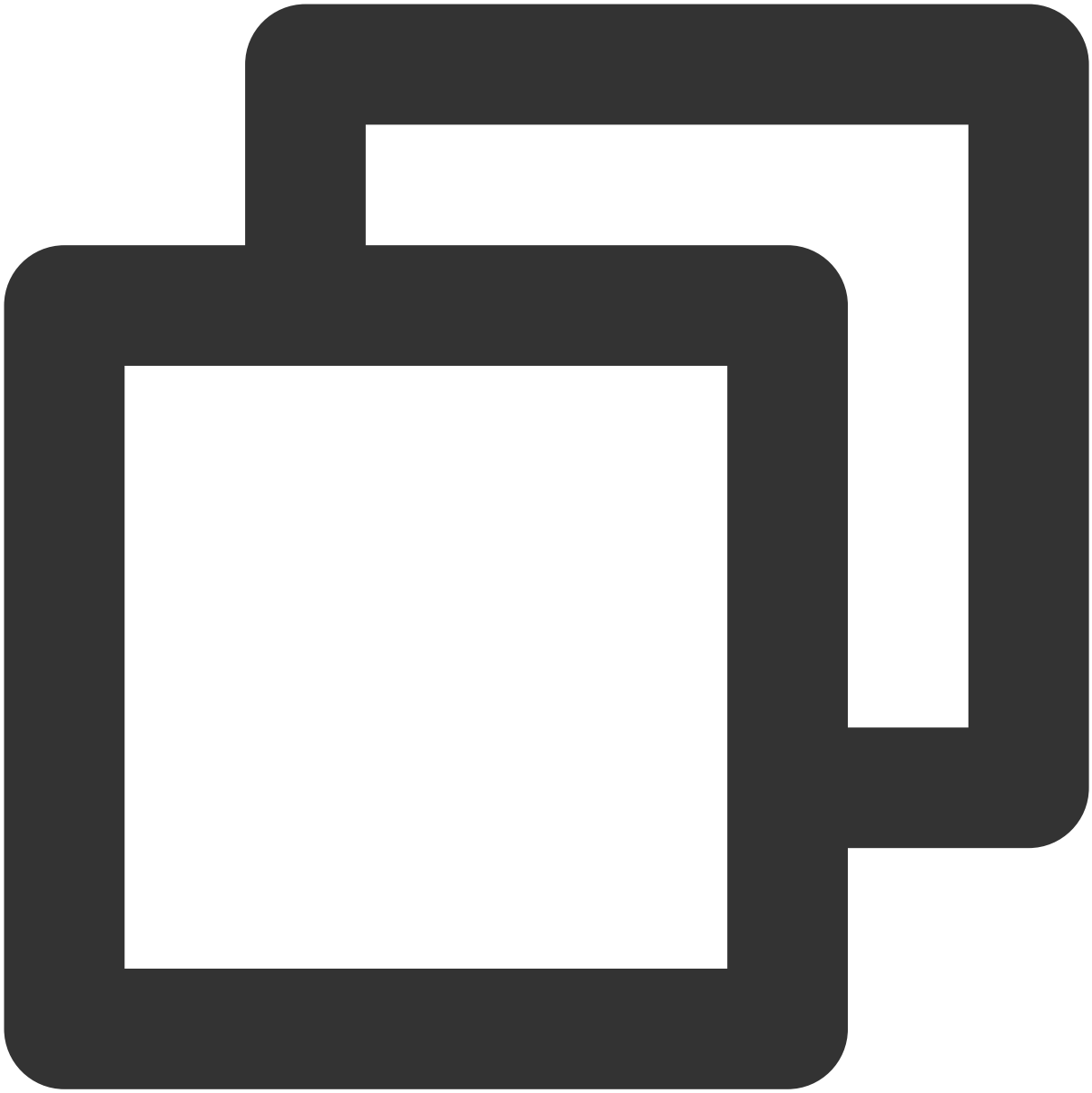
1. This interface does not support usage under the HTTP protocol; please deploy your website using the HTTPS protocol.
2. Error messages returned upon microphone activation failure; you may refer to getUserMedia exceptions for further insights.
3. In case an echo phenomenon occurs, you can mute the local video element video used for playing previews to prevent echo occurrences.



```
livePusher.videoView.muted = true;
```

stopMicrophone

Shut down the microphone device.



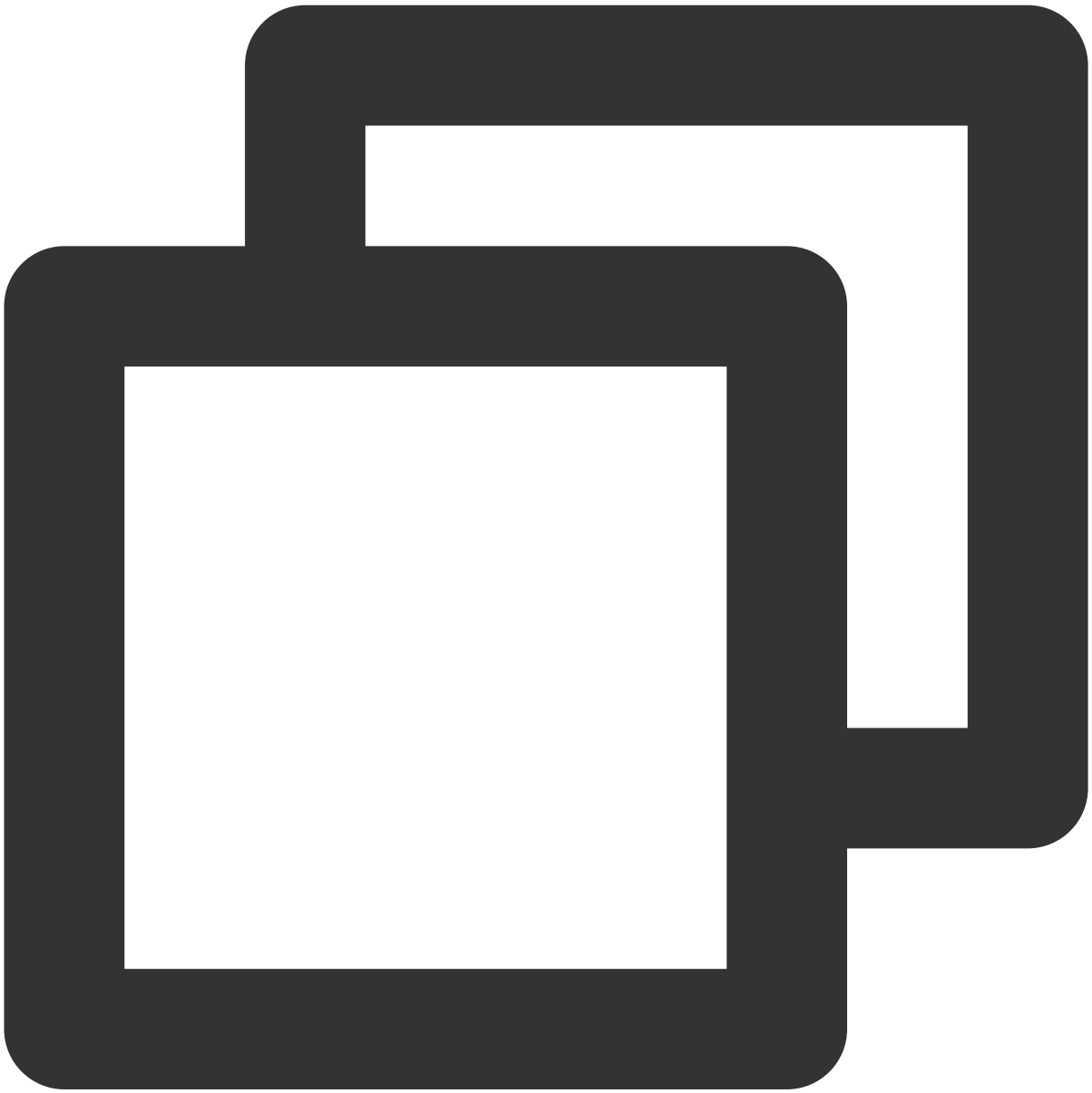
```
stopMicrophone(streamId?: string): void;
```

The parameters are described as follows:

Field	Type	Description
streamId	string	Stream ID is an optional parameter that indicates the specification of a microphone stream to be discontinued. Once local mix streaming has been activated, a specific microphone stream can be silenced by using Stream ID, if multiple microphone streams have been captured, otherwise, all microphone streams will be muted.

startScreenCapture

Screen capture is initiated, necessitating the user's permission for the browser to gain access to the screen. Should authorization or screen access be unsuccessful, the returned Promise object will project an error.



```
startScreenCapture(audio?: boolean): Promise<string>;
```

The parameters are described as follows:

Field	Type	Description

audio	boolean	Determines whether to collect system sound or Tag page sound: true - collection of sound, false - no sound collection, default is set to false.

Return:

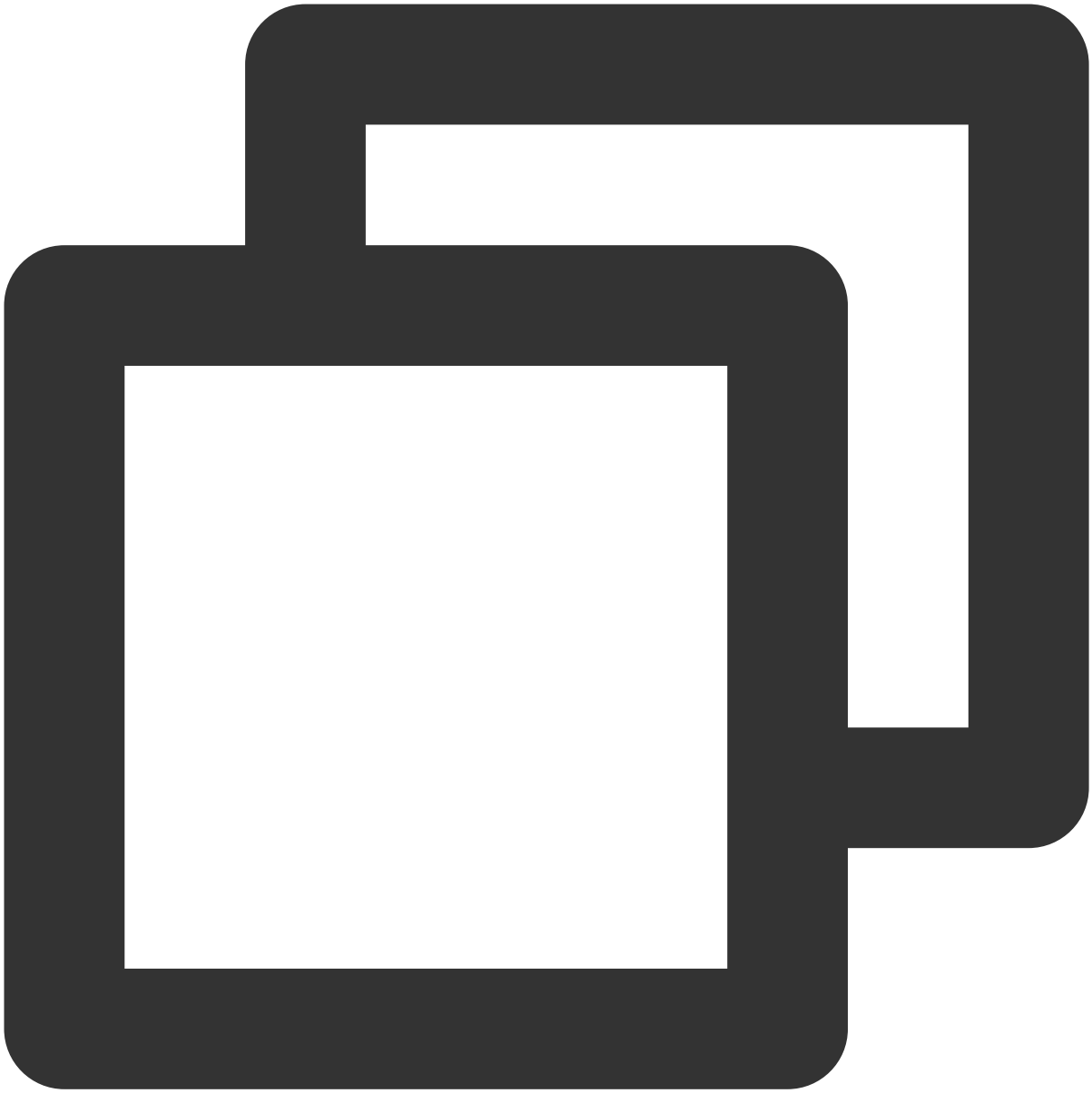
Yields a Promise object. Upon success, ID of the stream is returned, serving as the unique identification within SDK. In case of failure, the corresponding error message is thrown.

Note:

1. This interface does not support usage under the HTTP protocol; please deploy your website using the HTTPS protocol.
2. When opening screen capturing fails, refer to the error information returned. For more details, refer to getDisplayMedia exception.
3. Currently, only Chrome 74+ and Edge 79+ support the collection of sound. In Windows systems, you can capture sound from the entire system, while on Linux and macOS, you can only capture sound from tab pages.
4. If 'audio' is set to true, ensure that the option to capture sound at the bottom of the browser screen sharing pop-up window is checked, otherwise sound will not be captured. If 'audio' is set to false, the option to capture sound will not appear in the browser's screen sharing pop-up window.

stopScreenCapture

Stops screen capturing.



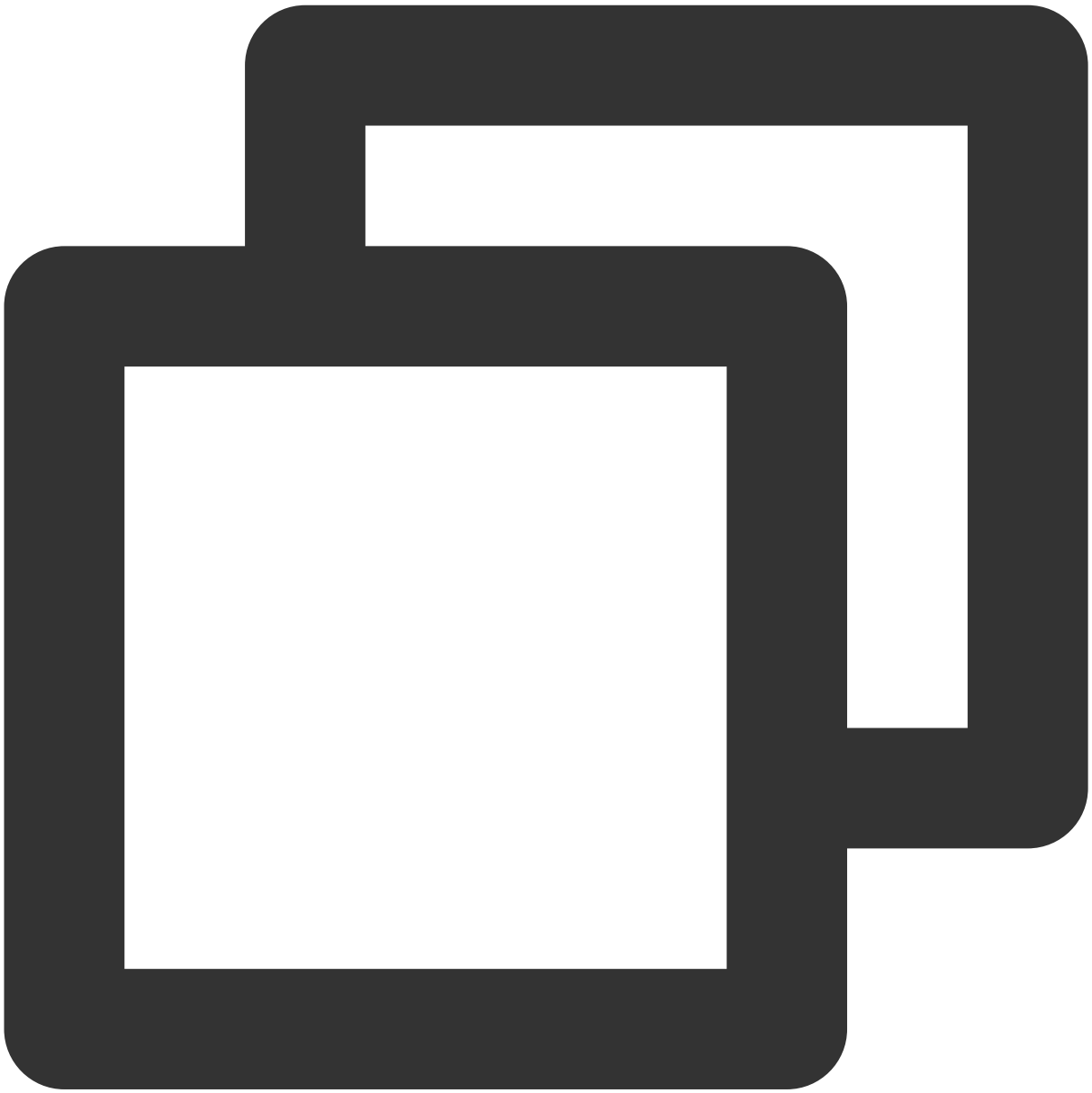
```
stopScreenCapture(streamId?: string): void;
```

The parameters are described as follows:

Field	Type	Description
streamId	string	Stream ID, optional parameter, designates the shared screen stream to be closed. Once local stream mix is activated, if multiple screen share streams have been collected, a particular screen share stream can be turned off via the stream ID. If not specified, all screen share streams will be shut down.

startVirtualCamera

Commence the collection of local media file streams. At present, the supported file formats are video (mp4), audio (mp3), and images (jpg, png, bmp).



```
startVirtualCamera(file: File): Promise<string>;
```

The parameters are described as follows:

Field	Type	Description

file	File	Local media files are compulsory. The file format must be any of the following: mp4, mp3, jpg, png, bmp.

Return:

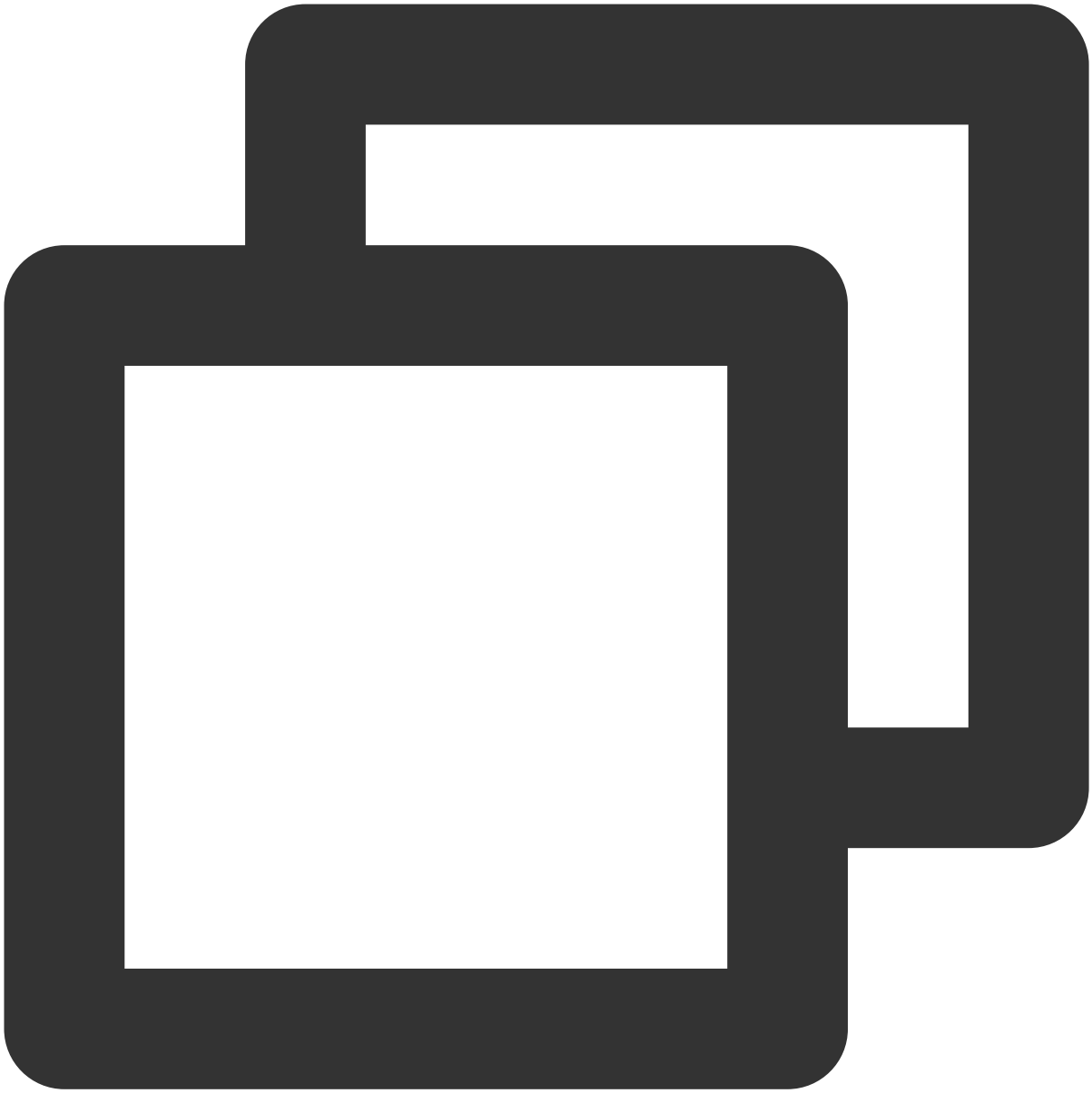
Yields a Promise object. Upon success, ID of the stream is returned, serving as the unique identification within SDK. In case of failure, the corresponding error message is thrown.

Note:

1. Local files support video, audio, and image. Video files capture both video and audio streams, audio files only capture audio streams, and image files only capture video streams.
2. A file object must be manually passed in, necessitating the prior use of `<input type="file">` to guide users in local file selection.

stopVirtualCamera

Halting the collection of local media file streams.



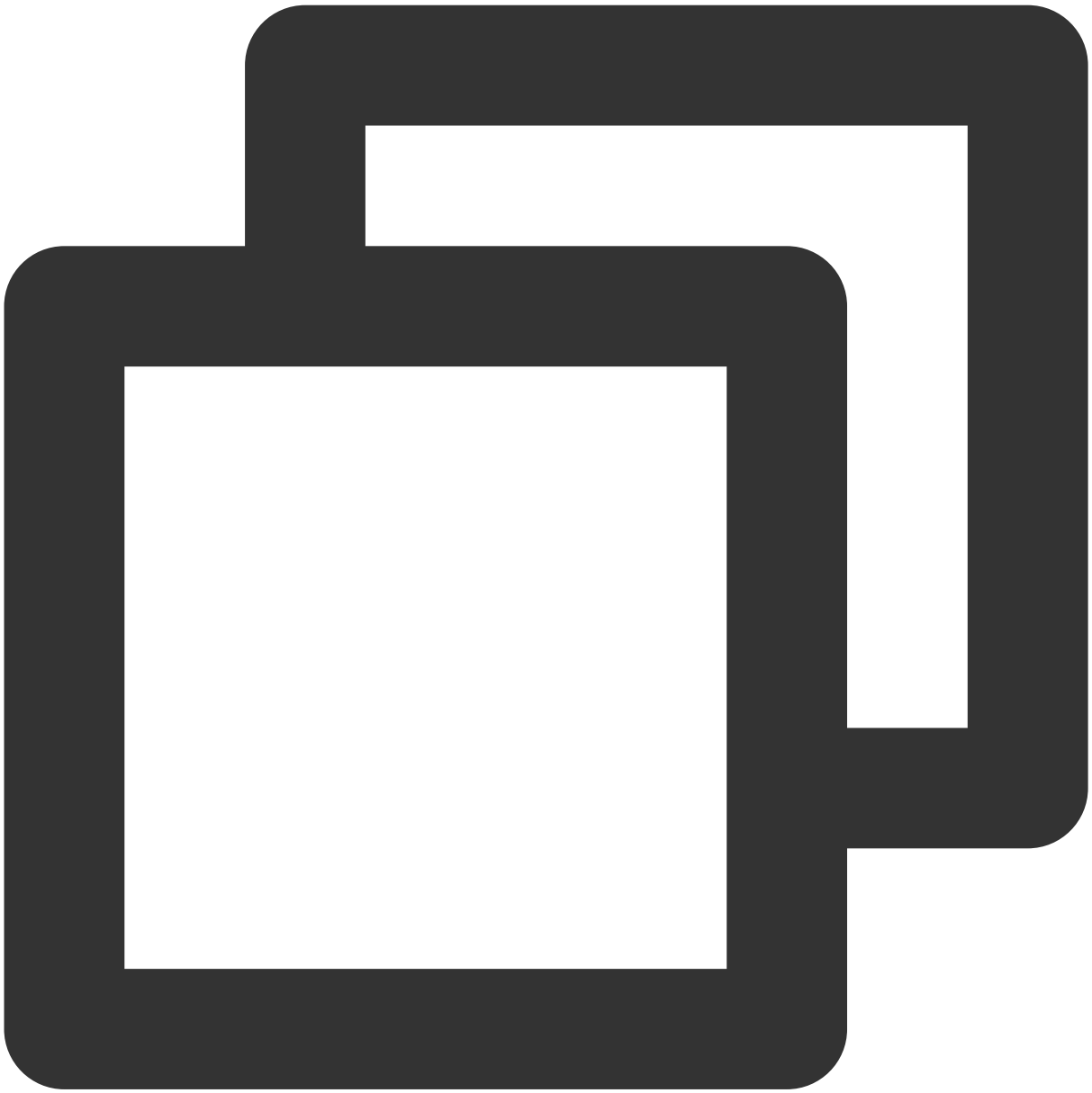
```
stopVirtualCamera(streamId?: string): void;
```

The parameters are described as follows:

Field	Type	Description
streamId	string	Stream ID is an optional parameter used to specify the media file stream that needs to be shut off. After enabling local merged streaming, if multiple media file streams have been collected, you can close a specified media file stream through Stream ID, otherwise all media file streams will be shut off.

startCustomCapture

Utilize custom user audio-visual streams. Leverage user-generated streams for local merging and broadcasting.



```
startCustomCapture(stream: MediaStream): Promise<string>;
```

The parameters are described as follows:

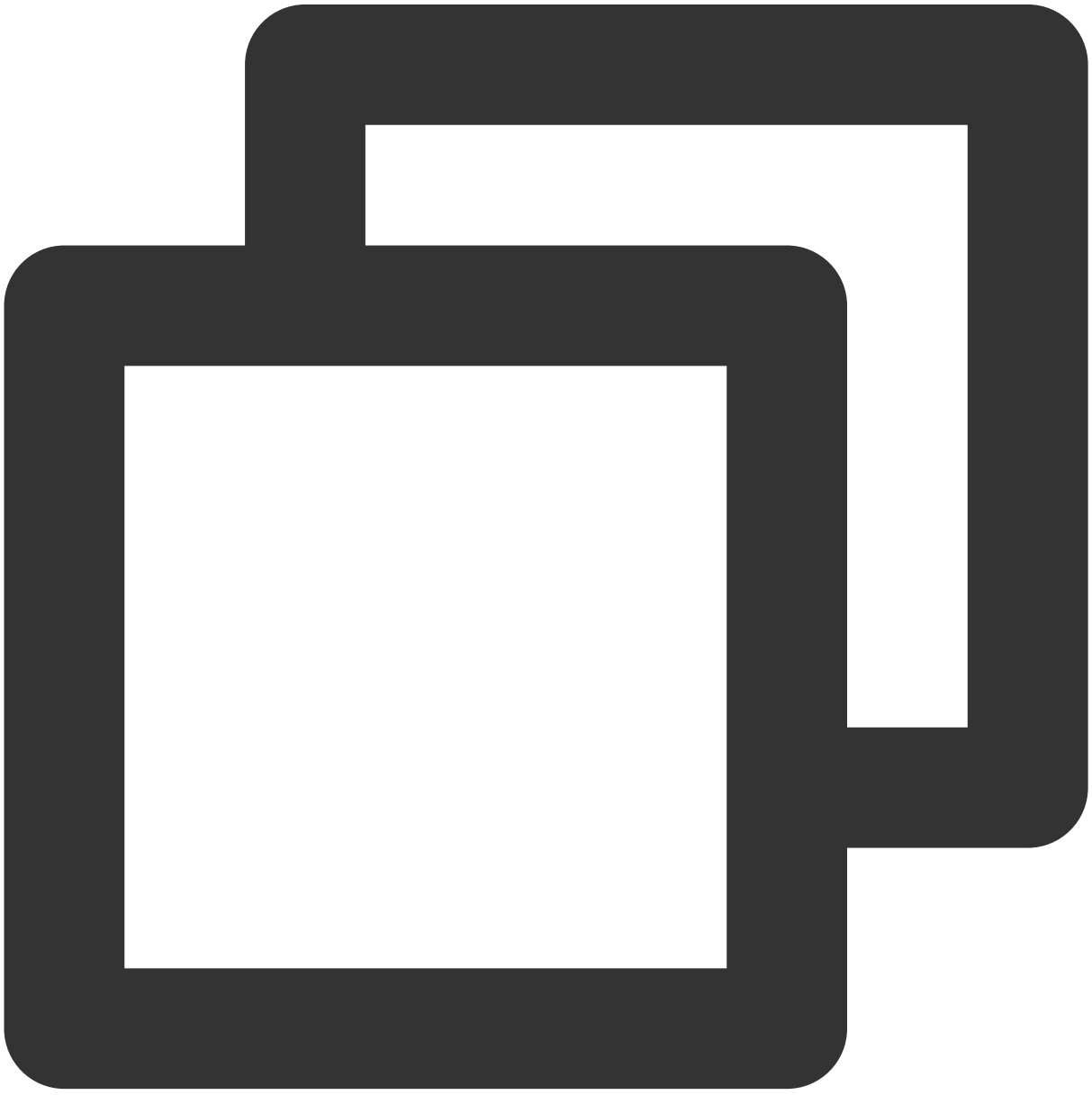
Field	Type	Description
stream	MediaStream	User-defined stream.

Return:

Yields a Promise object. Upon success, ID of the stream is returned, serving as the unique identification within SDK. In case of failure, the corresponding error message is thrown.

stopCustomCapture

Shuts down the custom audio/video stream, only removes the custom stream, and does not terminate the custom stream.



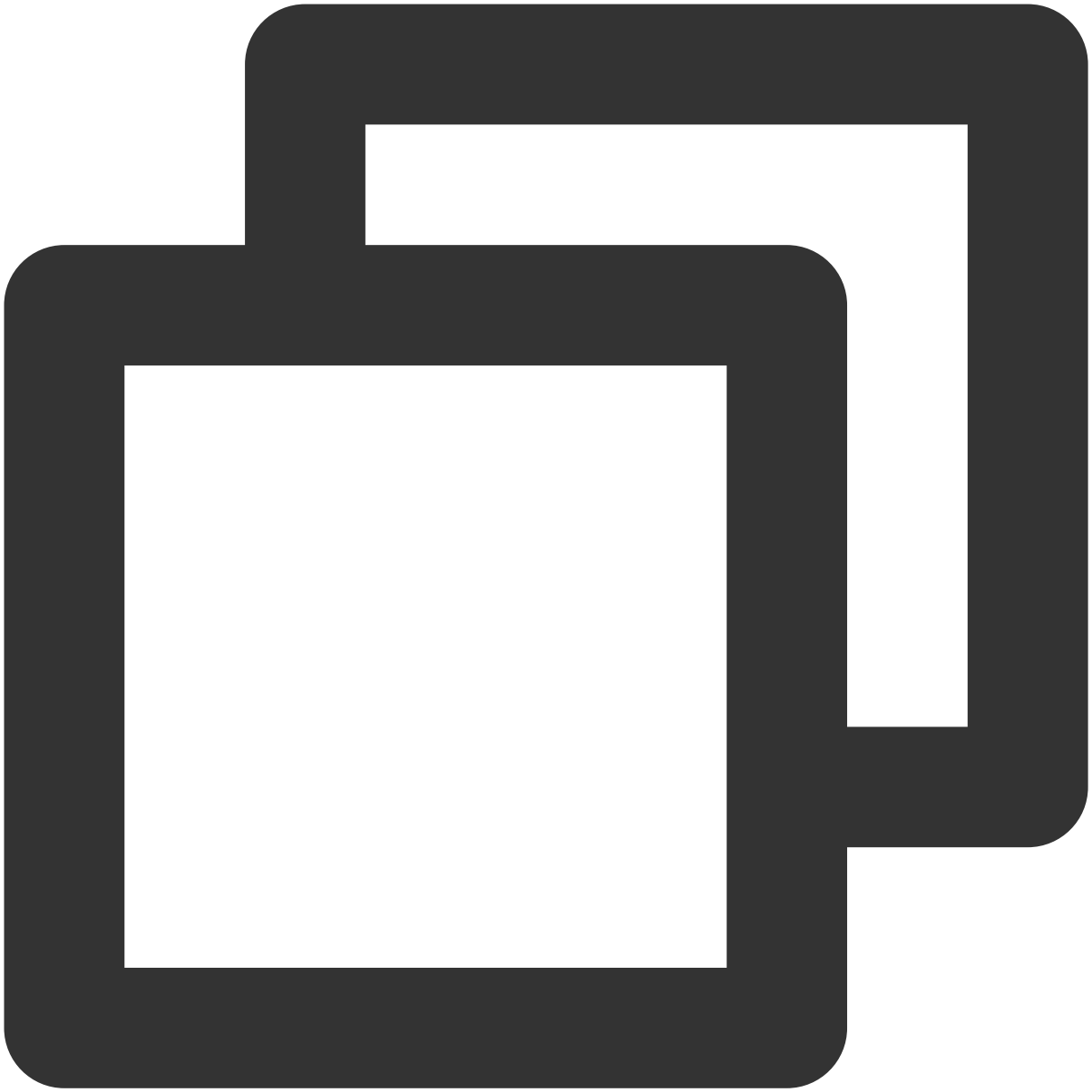
```
stopCustomCapture(streamId?: string): void;
```

The parameters are described as follows:

Field	Type	Description
streamId	string	Stream ID, an optional parameter, refers to the custom stream that needs to be removed. After activating local mixed flow, if multiple custom streams are added, you can remove the specific custom stream through the stream ID, otherwise all custom streams will be purged.

startPush

Initiate streaming, establish a WebRTC connection, push audio and video streams to Tencent Cloud servers. If the local mixing function has been activated, the stream data being pushed will be the result of mixed processing.



```
startPush(pushUrl: string): Promise<void>;
```

The parameters are described as follows:

Field	Type	Description
pushUrl	string	WebRTC push address.

Return:

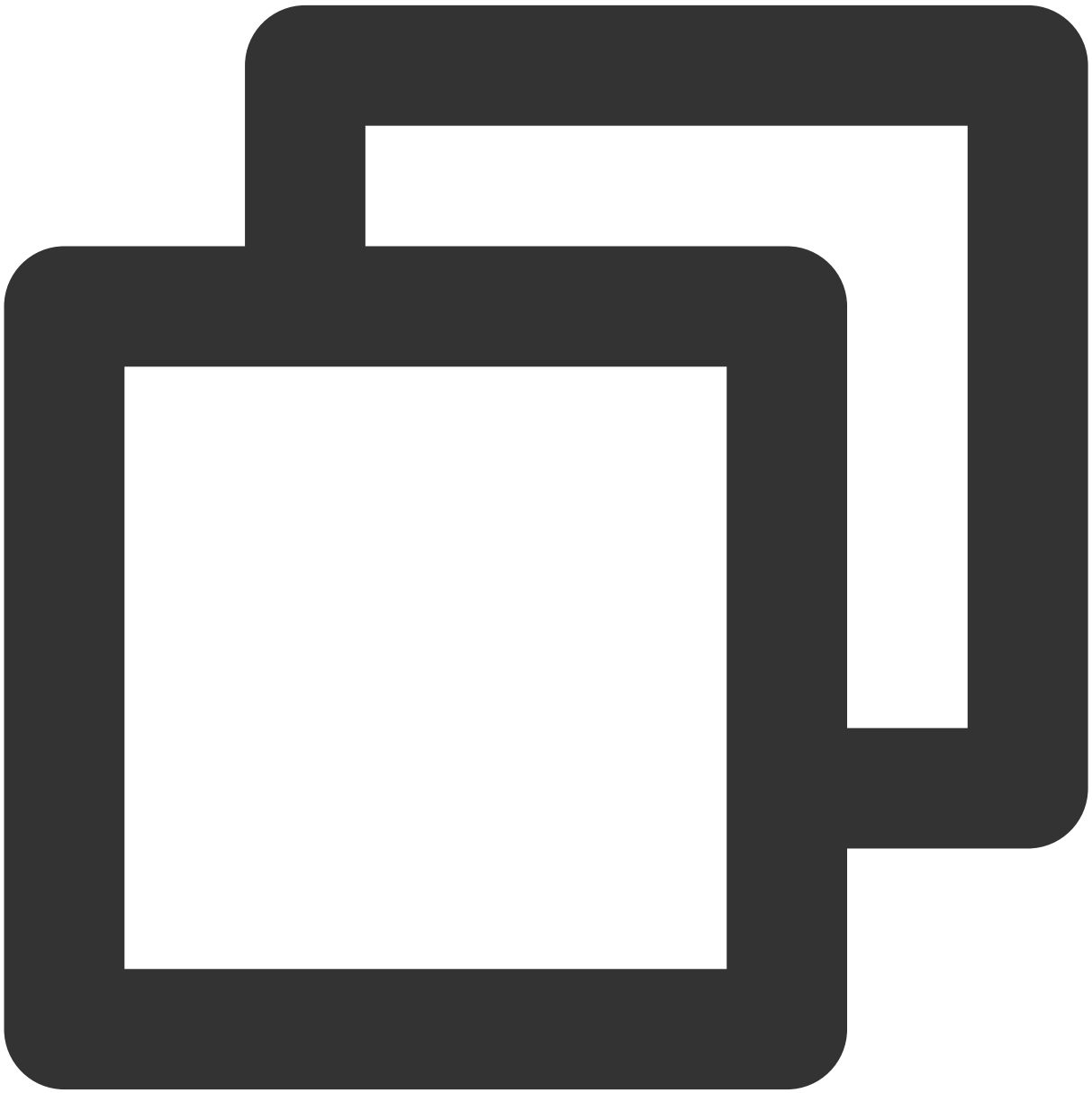
Returns a Promise object.

Note:

Refer to the format of the push stream address Assemble Push Stream URL.

stopPush

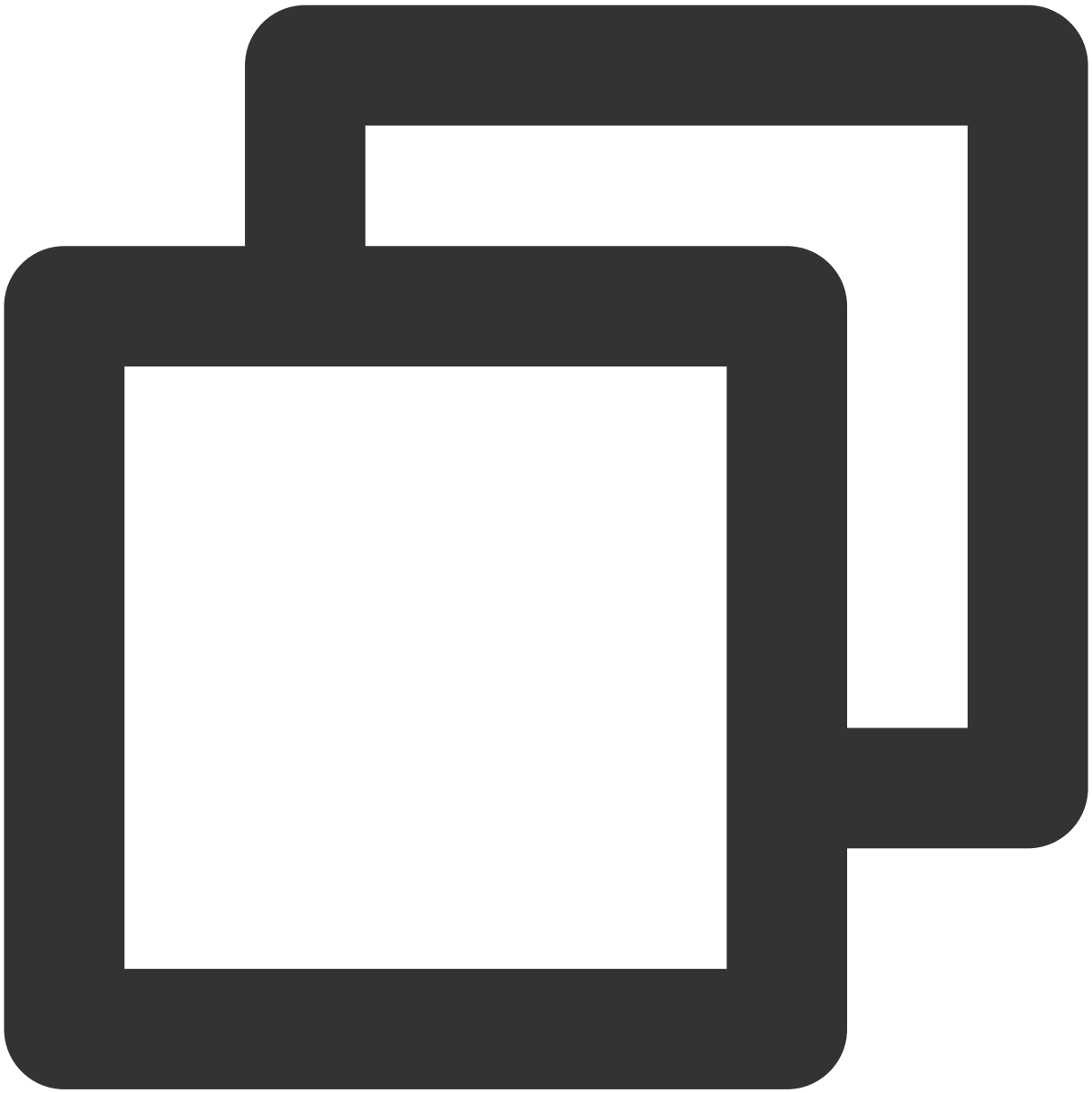
Cease the propagation of audio and video streams, and terminate the WebRTC connection.



```
stopPush(): void;
```

isPushing

Inquire whether the stream is currently being pushed.



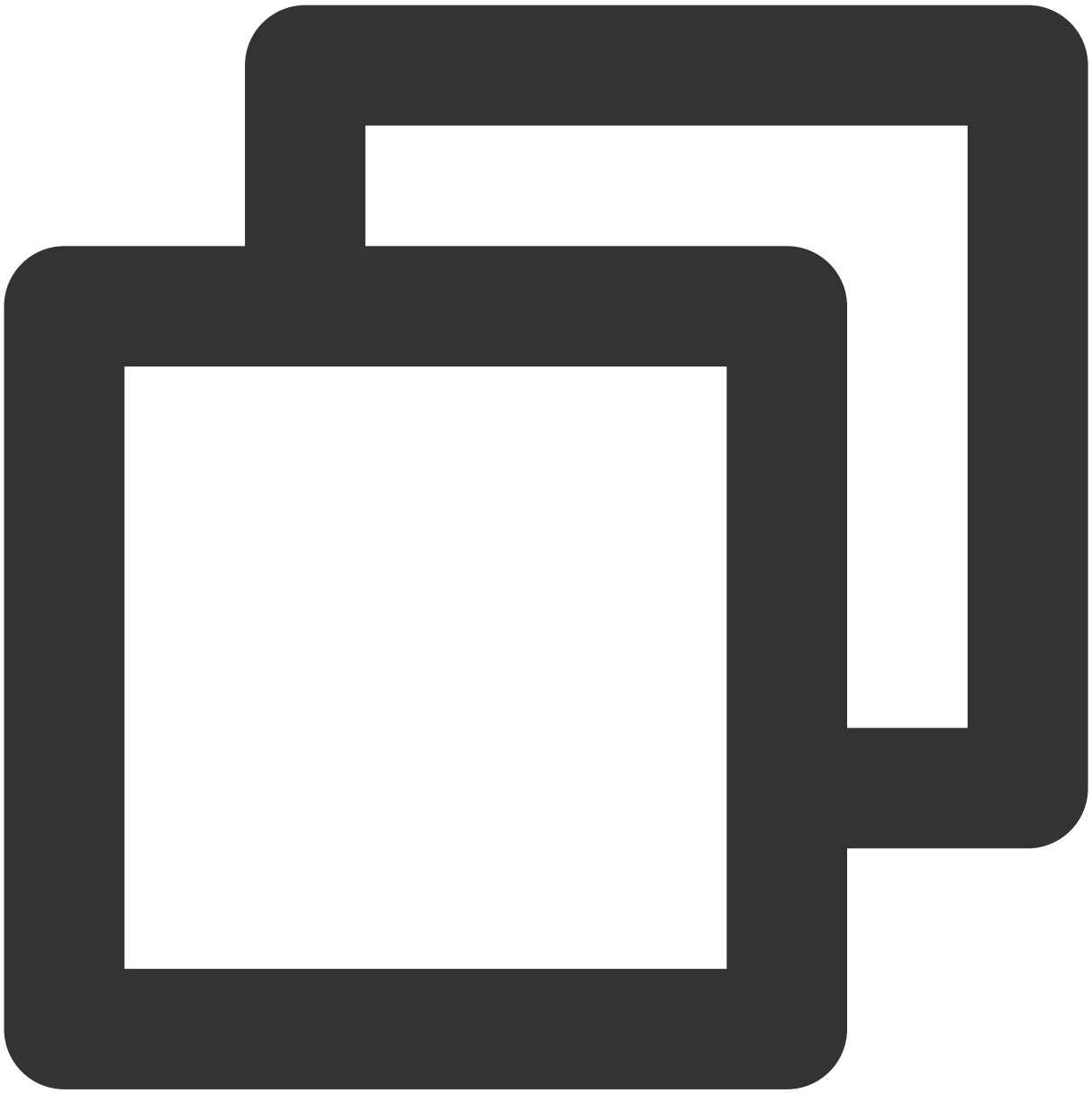
```
isPushing(): boolean;
```

Return:

Boolean values, true - streaming in progress, false - streaming not in progress.

getMediaStream

Acquire the audio and video streams based on the stream ID.



```
getMediaStream(streamId: string): MediaStream;
```

The parameters are described as follows:

Field	Type	Description
streamId	string	Stream ID is returned after the successful invocation of interfaces such as

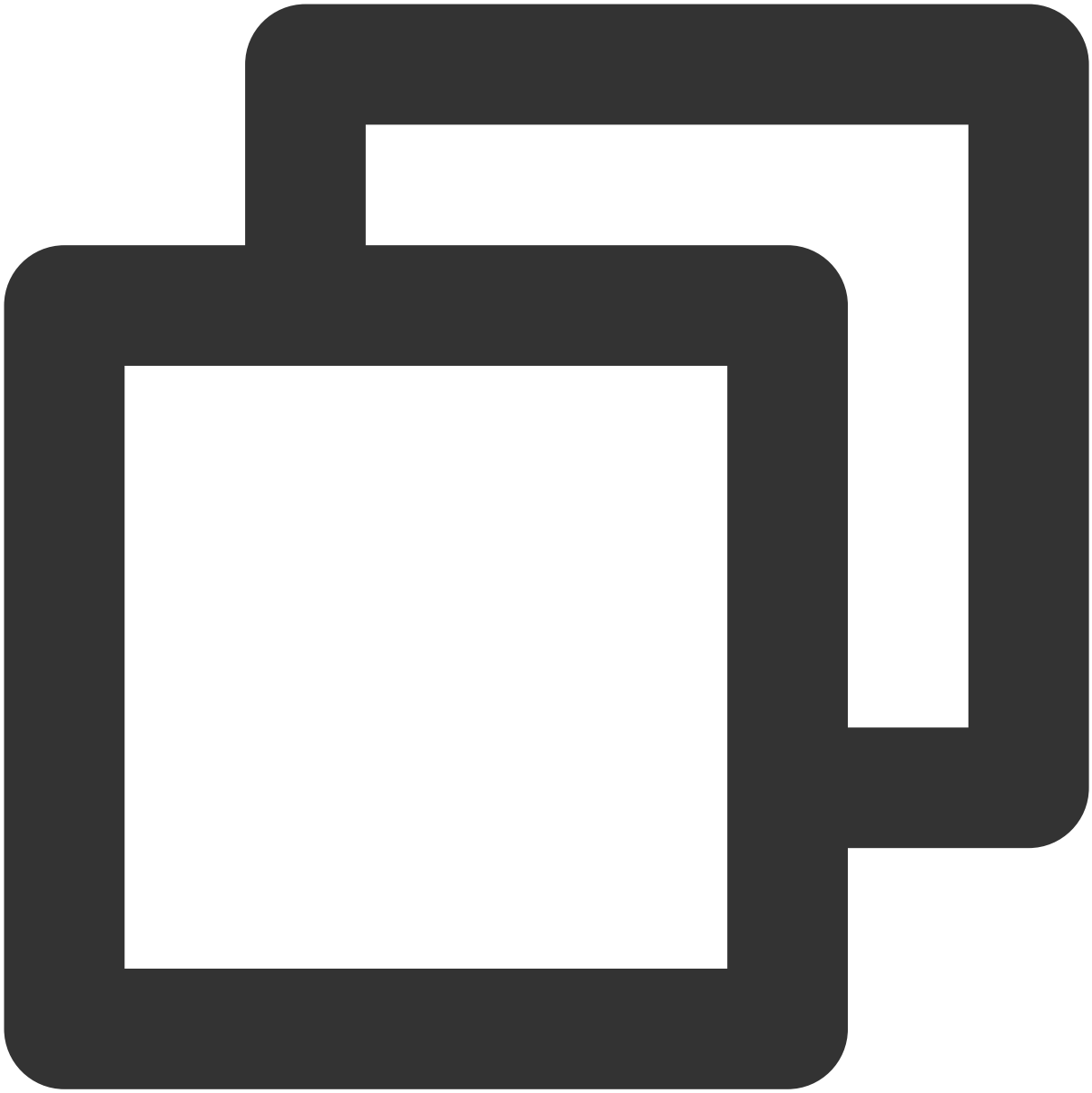
		startCamera(), startMicrophone(), startScreenCapture(), etc.

Return:

The stream object captured can be played back by passing it to the srcObject attribute of the video tag.

getDeviceManager

Acquire the device management entity. Through device management, operations such as querying the device list and switching devices can be conducted.



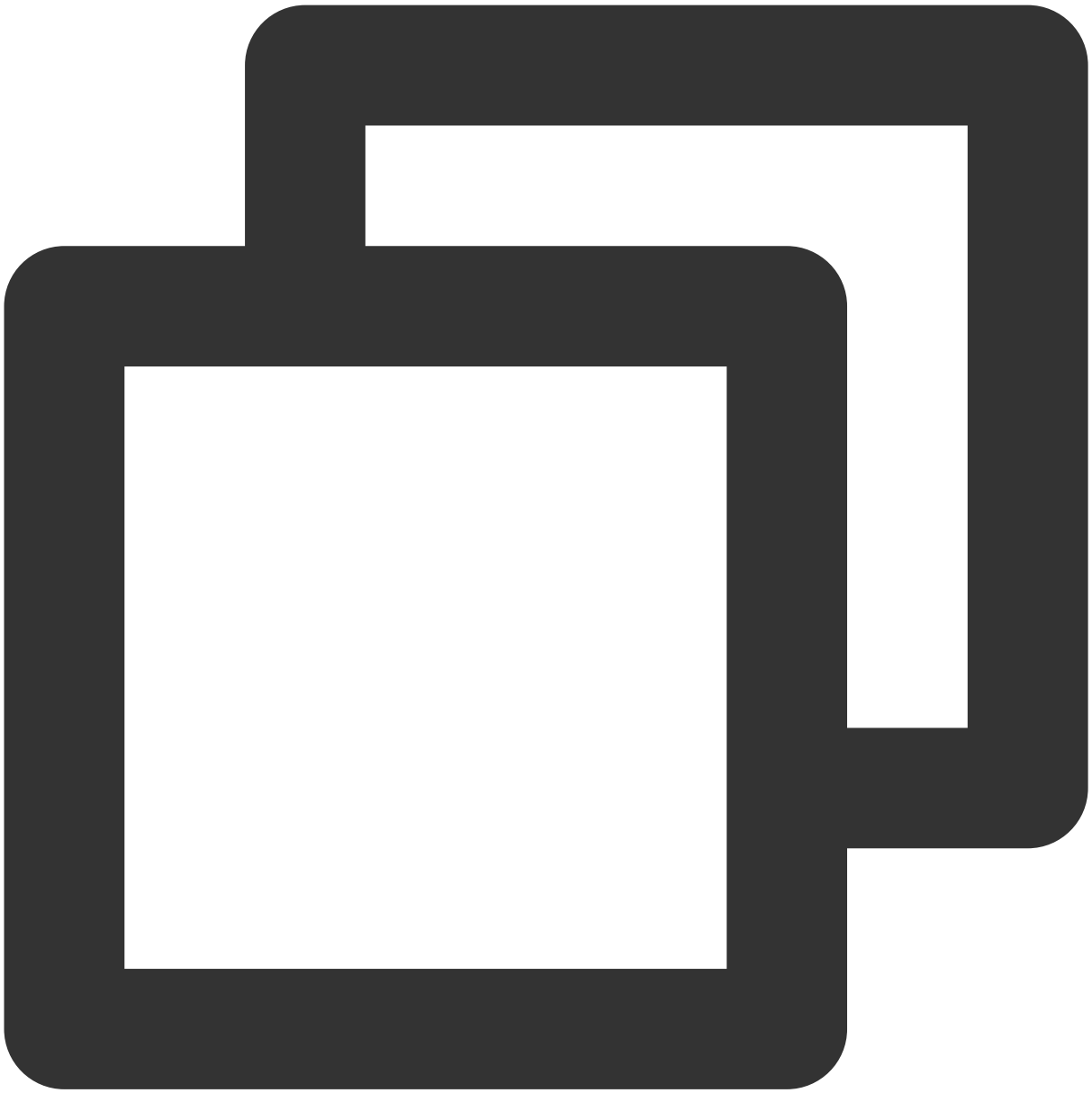
```
getDeviceManager(): TXDeviceManager;
```

Return:

Device Management Object. For specific usage, please refer to TXDeviceManager.

getVideoEffectManager

Acquire the video effect management object. Virtue of video effect management facilitates a myriad operations such as picture in picture, mirroring, filters, watermarks, text, and so forth.



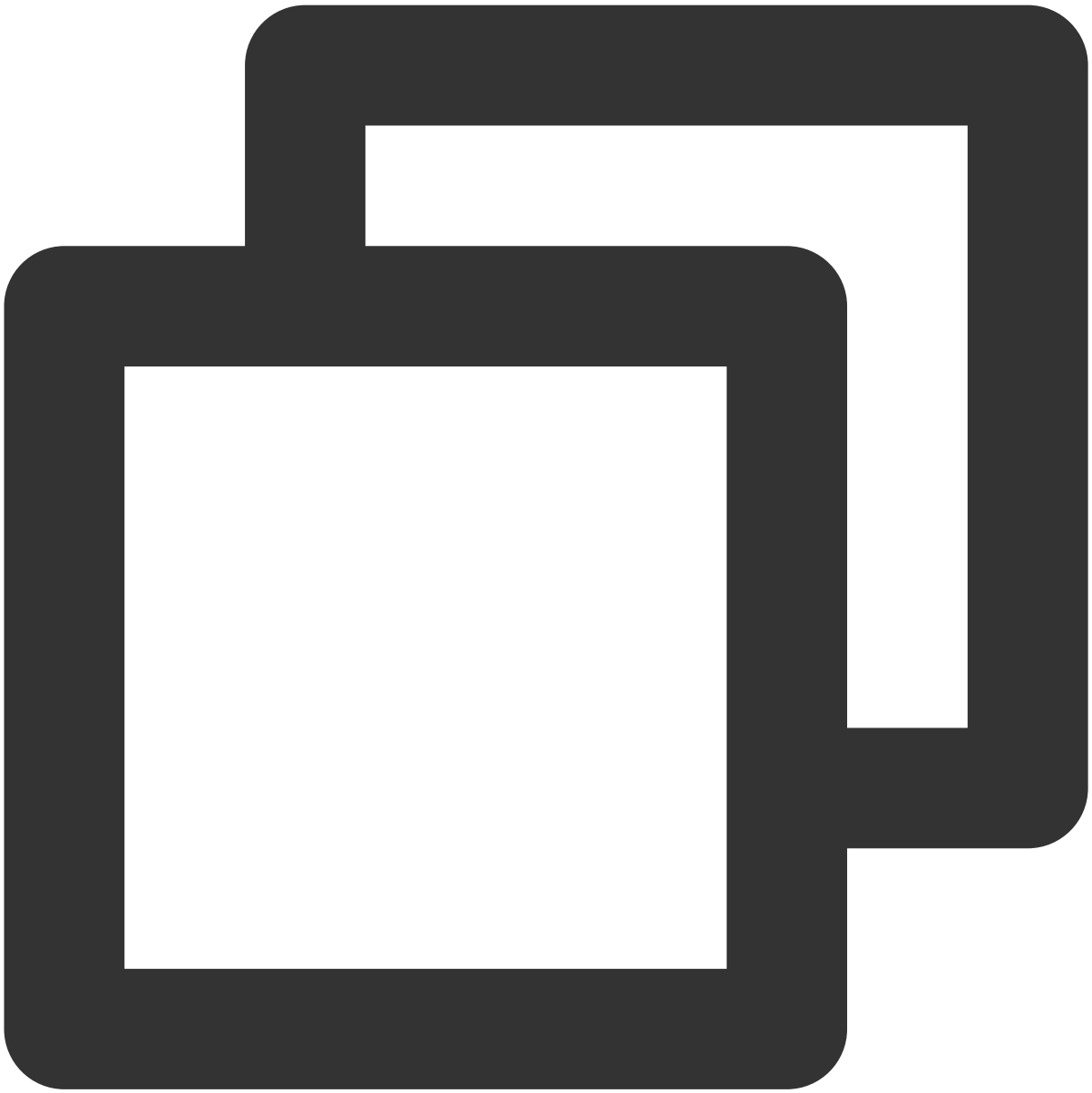
```
getVideoEffectManager(): TXVideoEffectManager;
```

Return:

The video effect management object, for specific usage, please refer to TXVideoEffectManager .

getAudioEffectManager

Acquire the audio effect management object. Through audio effect management, one can perform volume adjustment operations.



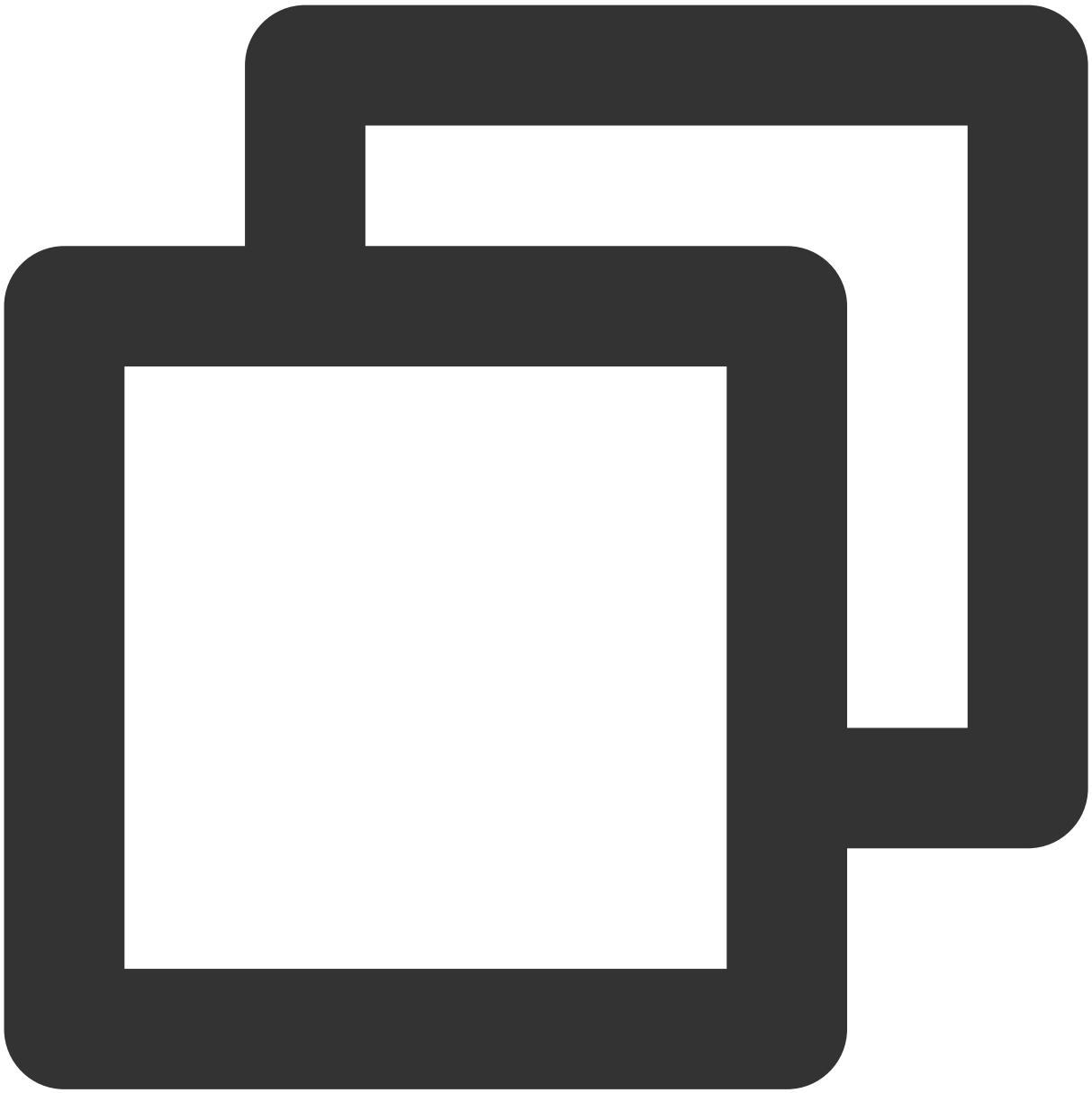
```
getAudioEffectManager(): TXAudioEffectManager;
```

Return:

The audio effect management object, for the specific method of use, please refer to TXAudioEffectManager.

setVideoMute

Configure whether to disable the video stream. If the local stream mixing function is enabled, the stream that is disabled is the final produced video stream.



```
setVideoMute(mute: boolean): void;
```

The parameters are described as follows:

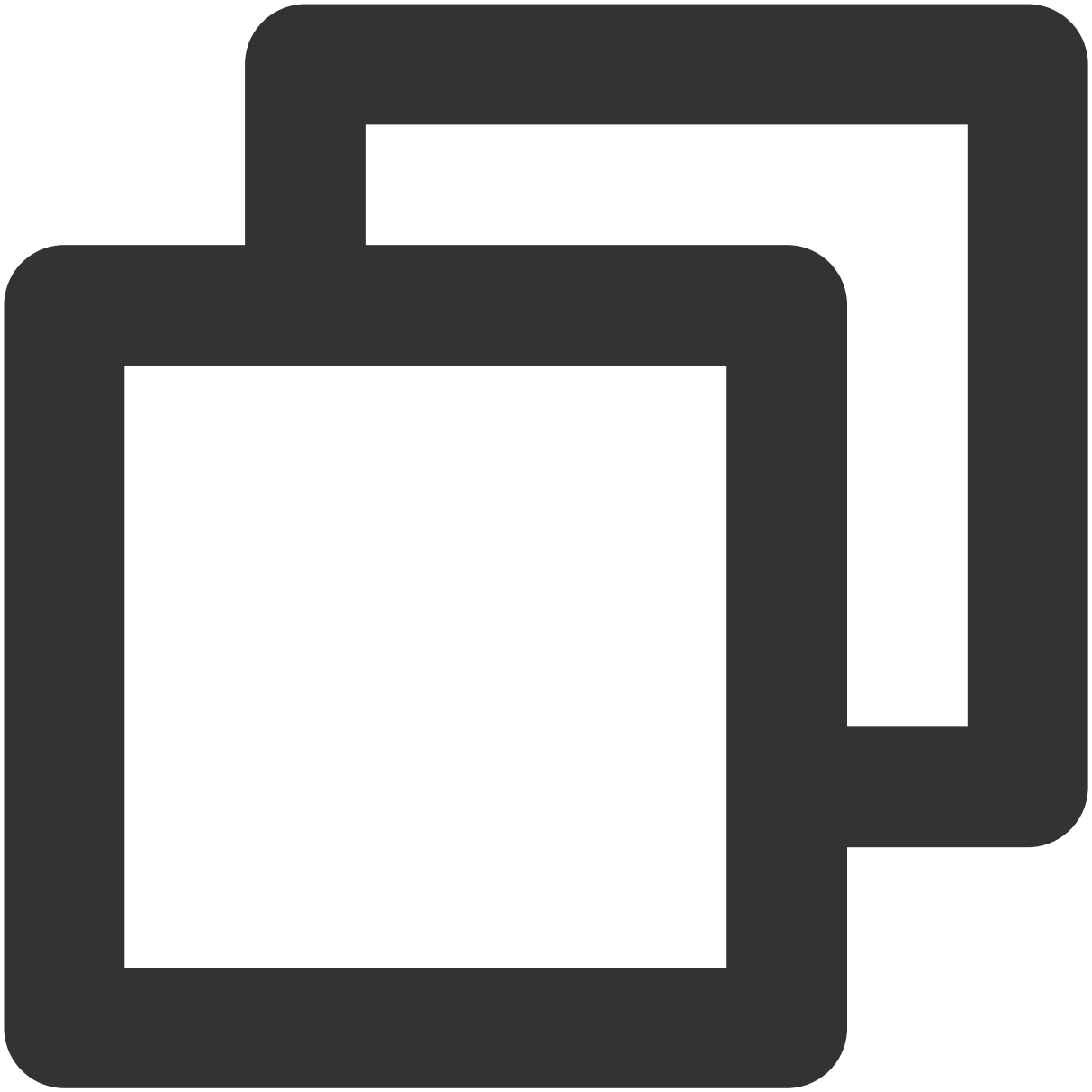
Field	Type	Description
mute	boolean	true - disabled, false - enabled.

Note:

1. The setting will take effect only when there is a current video stream.
2. After disabling, each frame will be filled with black pixels, in reality still capturing the video stream.
3. It is recommended to use pauseVideo() and resumeVideo() directly.

setAudioMute

Configure whether to disable the audio stream. If the local stream mix feature is enabled, the audio stream that will be disabled corresponds to the final composite.



```
setAudioMute(mute: boolean): void;
```

The parameters are described as follows:

Field	Type	Description
mute	boolean	true - disabled, false - enabled.

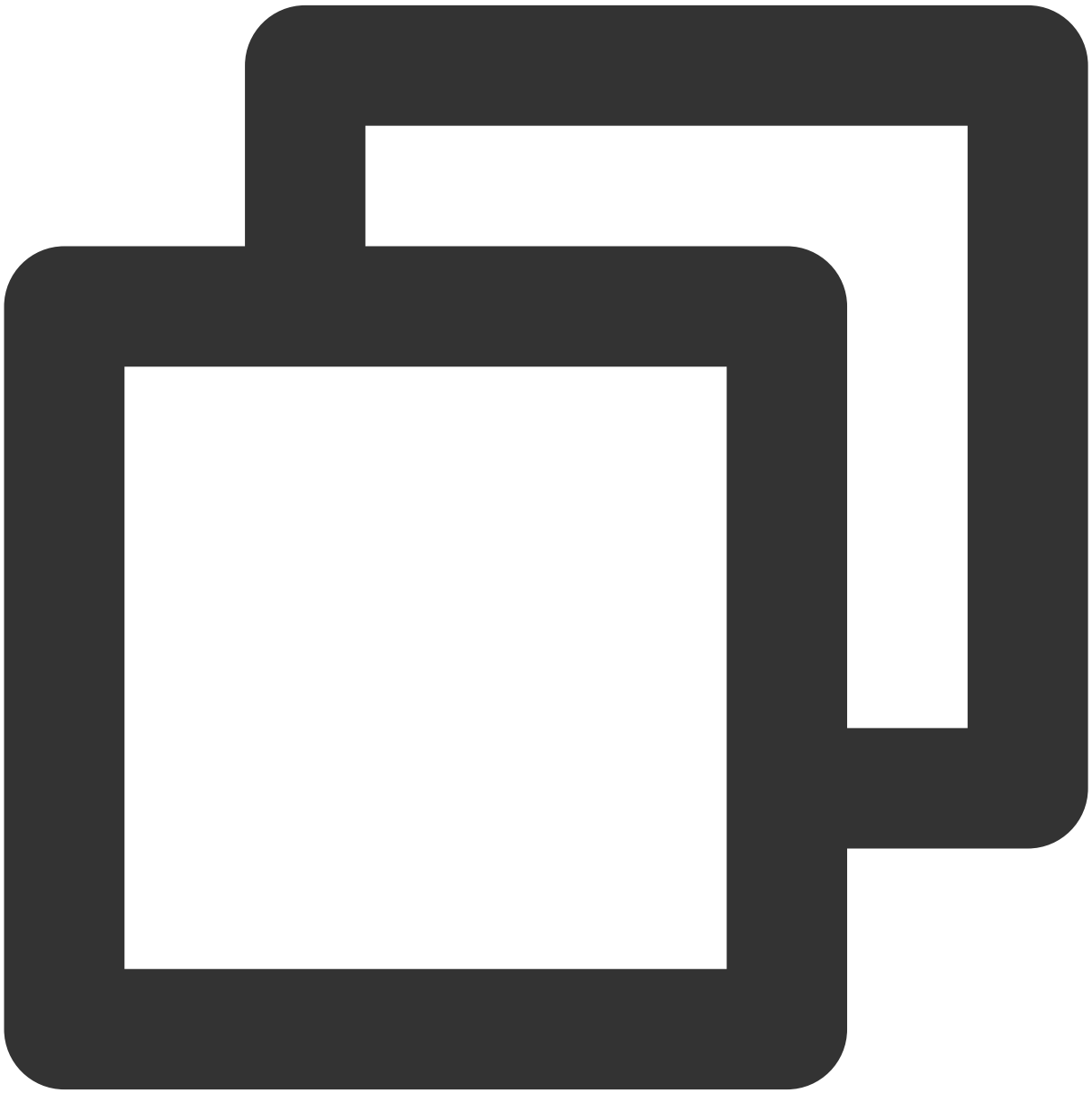
Note:

1. The settings will take effect only when there is an ongoing audio stream.

2. After being disabled, there is no sound, but in fact audio stream is still being gathered.
3. It's advised to directly use pauseAudio() and resumeAudio() .

pauseVideo

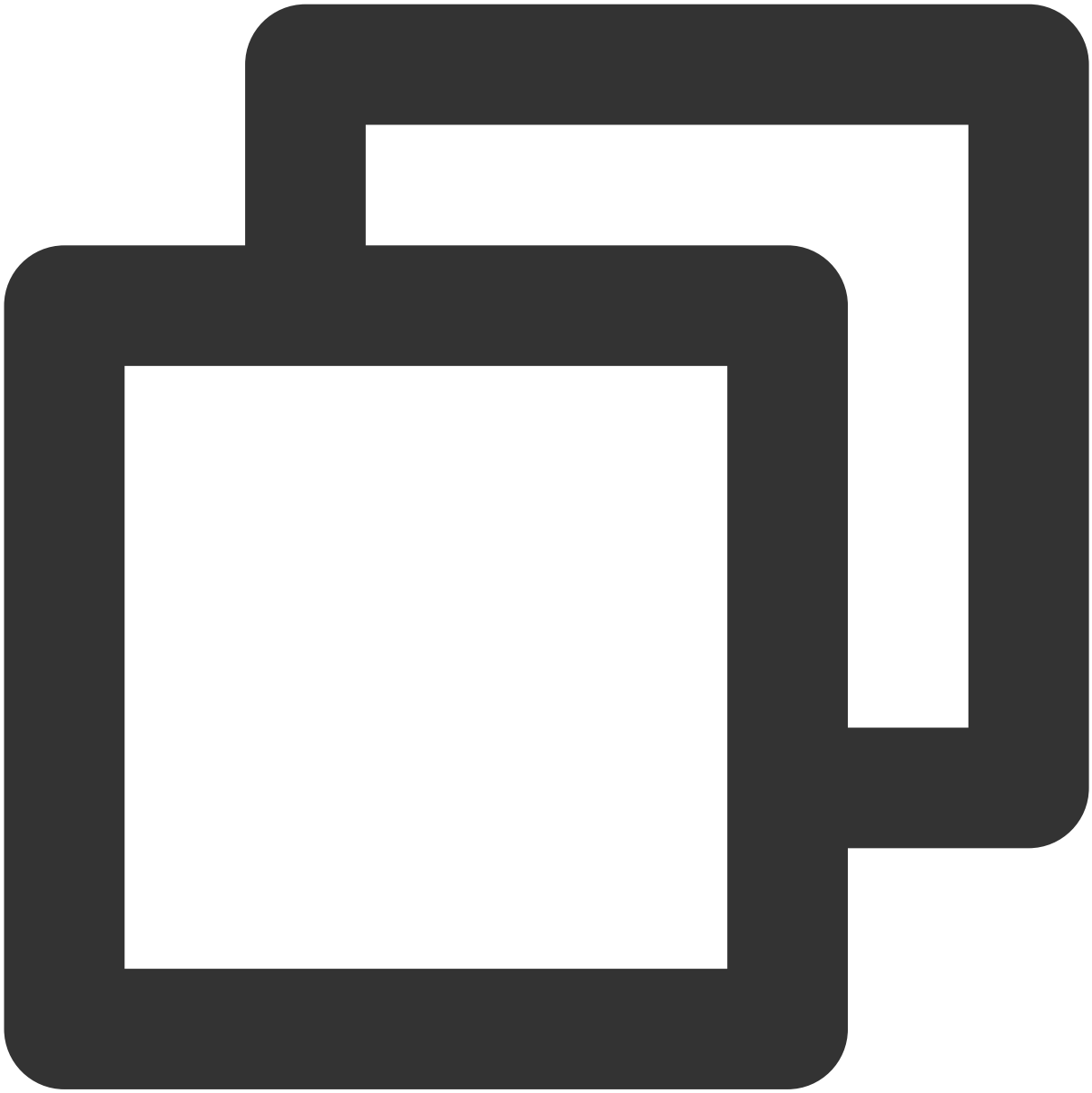
Disabling video stream. Equivalent to `setVideoMute(true)` .



```
pauseVideo(): void;
```

pauseAudio

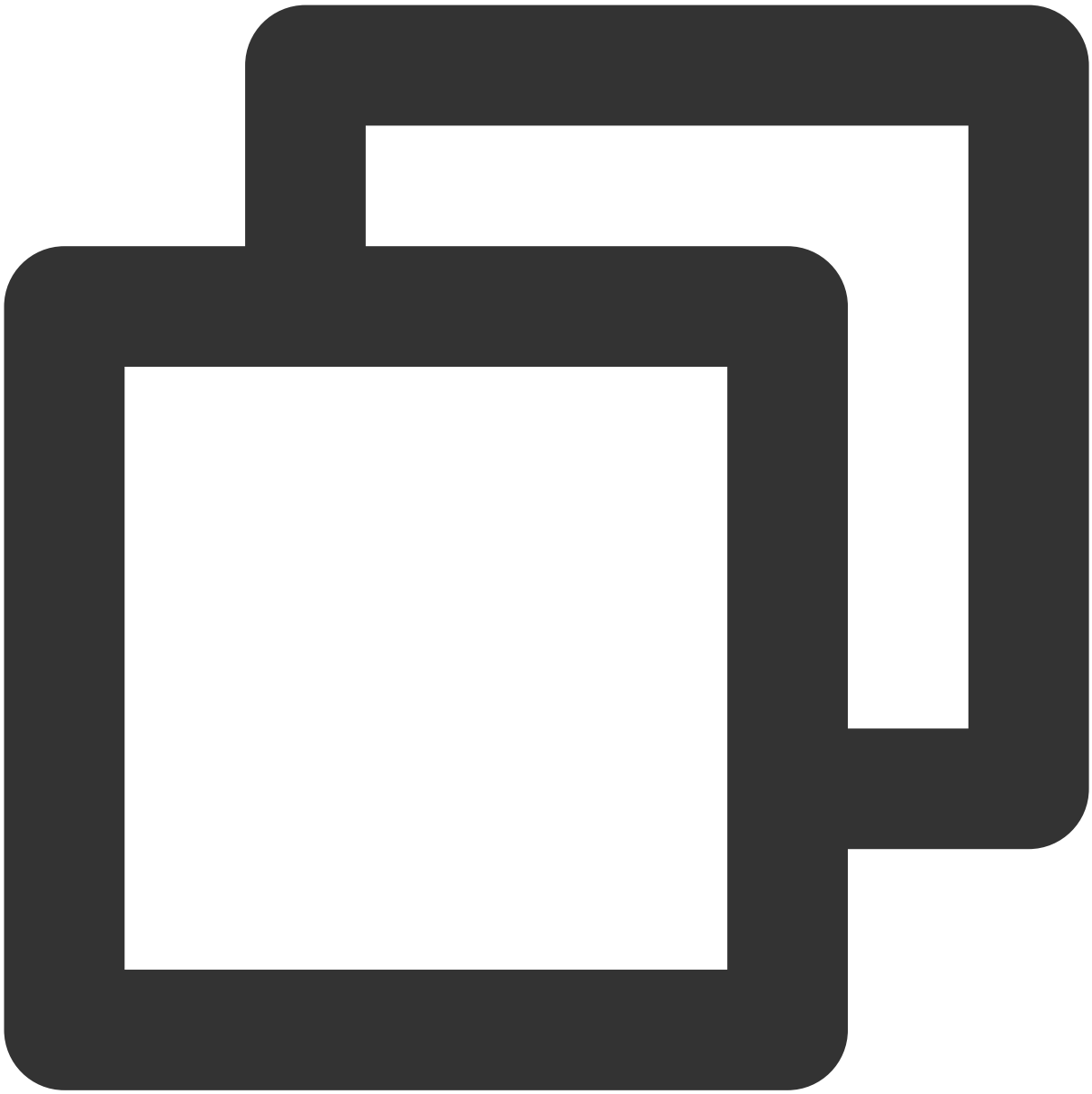
Disable the audio stream. This is equivalent to `setAudioMute(true)` .



```
pauseAudio(): void;
```

resumeVideo

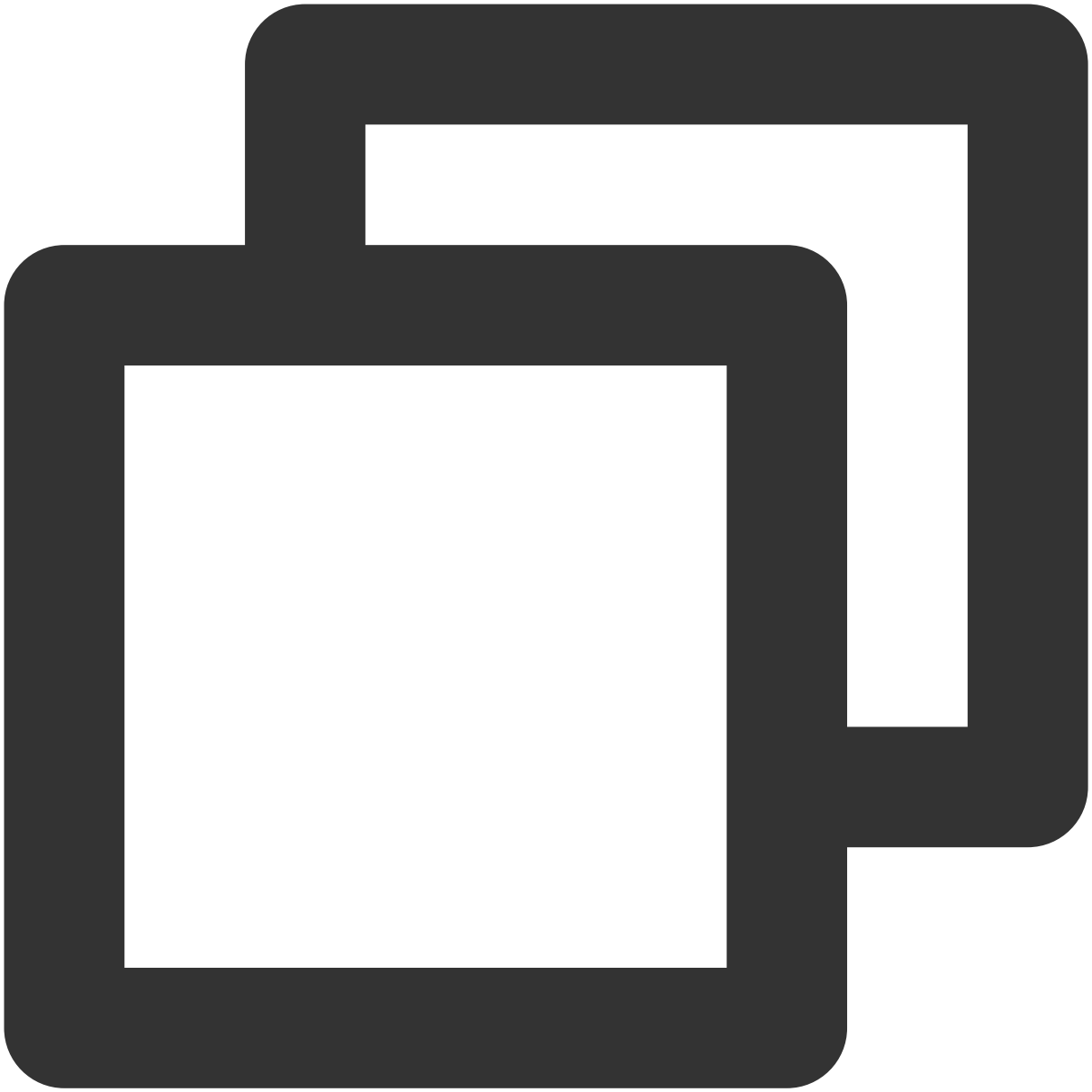
Resuming the video stream is equivalent to `setVideoMute(false)` .



```
resumeVideo(): void;
```

resumeAudio

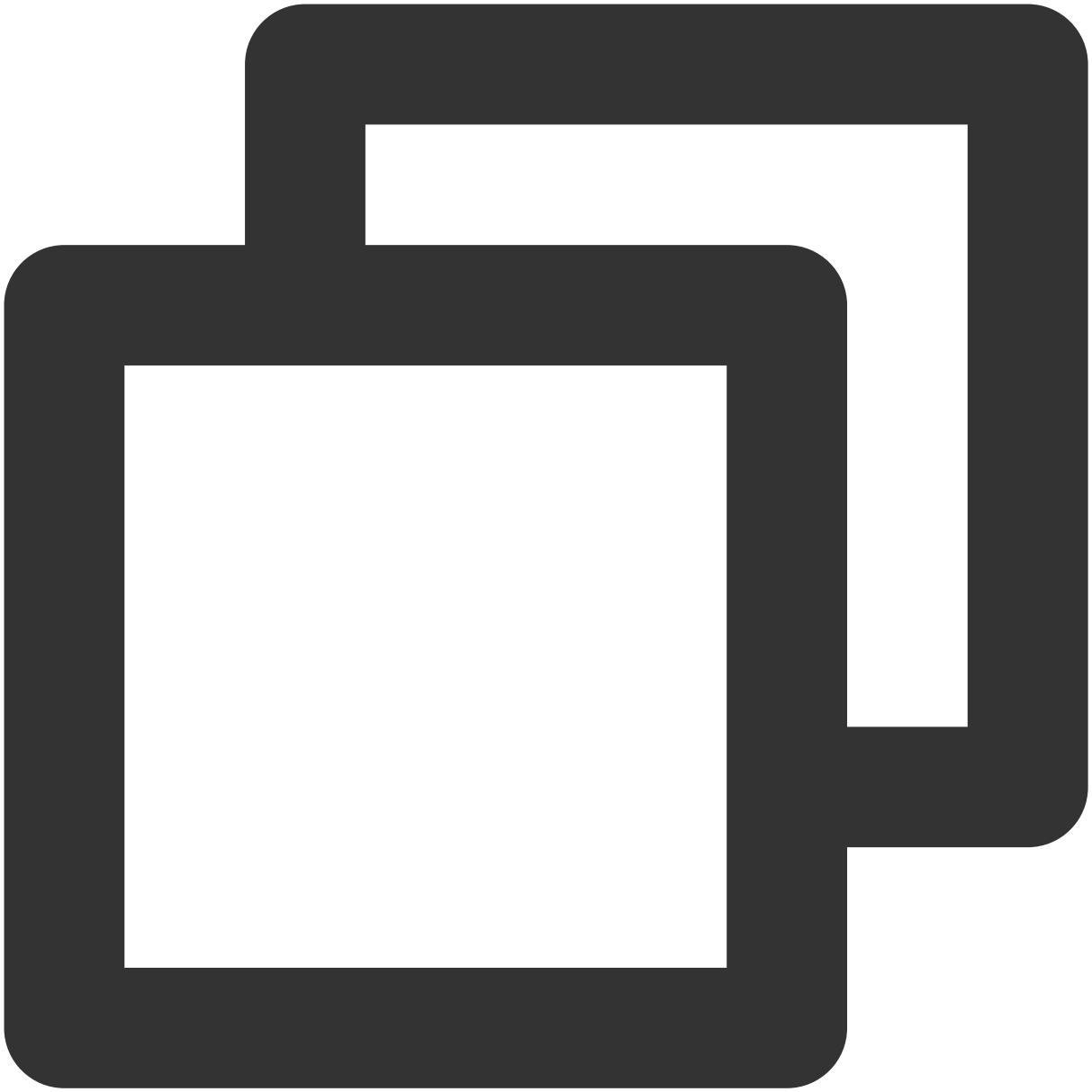
Resume the audio stream. Equivalent to `setAudioMute(false)` .



```
resumeAudio(): void;
```

setVideoContentHint

Set up video content prompts to enhance video encoding quality in various content scenarios.



```
setVideoContentHint(contentHint: string): void;
```

The parameters are described as follows:

Field	Type	Description
contentHint	string	Content hint, refer to MediaStreamTrack.contentHint .

The suggested value range is as follows:

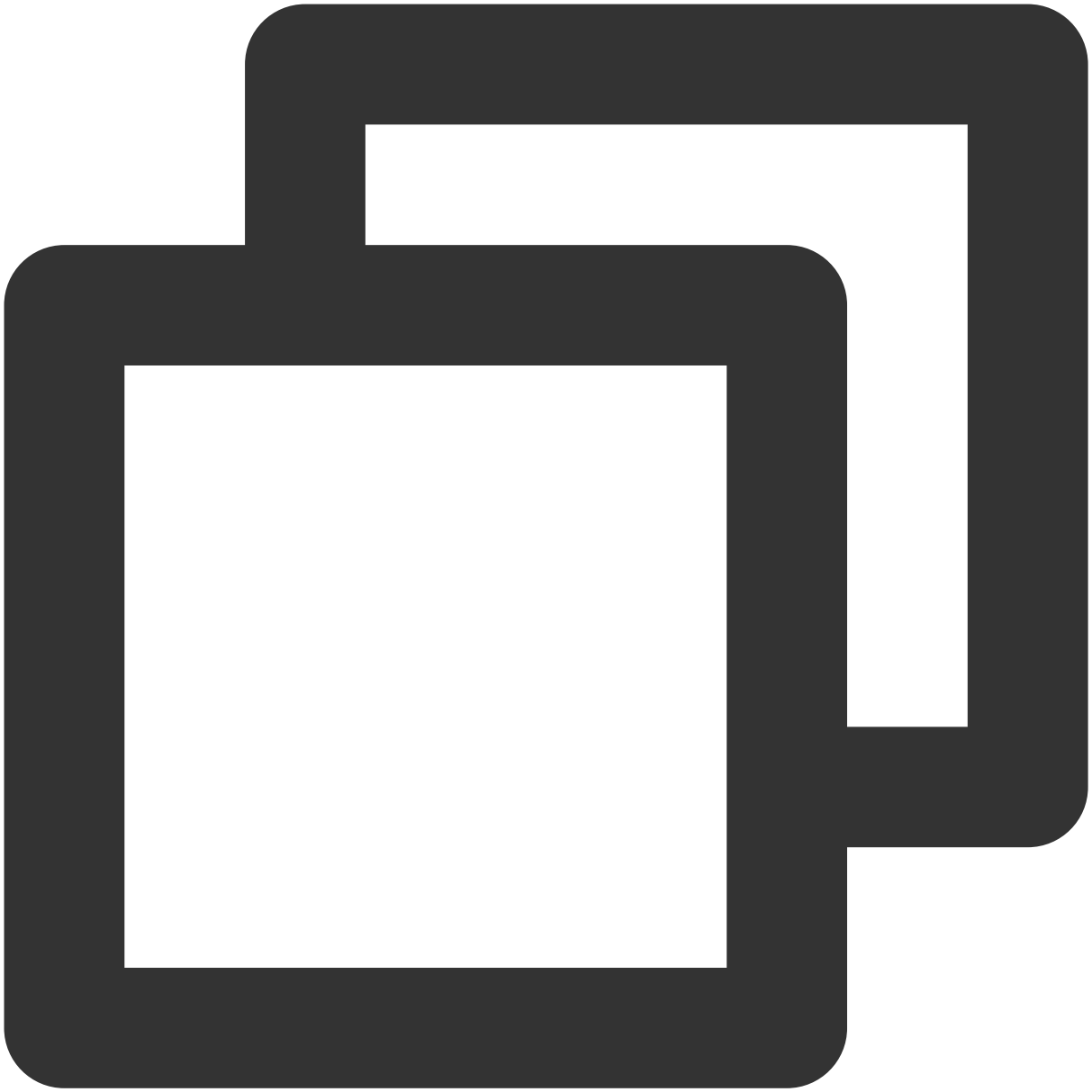
Fetching Value	Description

''	By default, the browser will automatically evaluate the video content and choose the appropriate prompt configuration for encoding.
'motion'	This manifests as a preference for smoothness, applicable when the video content is obtained from camera captures, films, videos, or games.
'detail'	Manifests as clarity precedence, suitable for video content that includes mixed images and text. When screen sharing, it's advisable to use this prompt.
'text'	Manifests as clarity prioritized, applicable when the video content only contains copious amount of text.

Note:
The setting will take effect only when there is a current video stream.

setObserver

Setting up streaming event notification callbacks. By arranging these callbacks, one can monitor several streaming event notifications, including streaming states, statistical data, along with warnings and error messages, among others.



```
setObserver(observer: TXLivePusherObserver): void;
```

The parameters are described as follows:

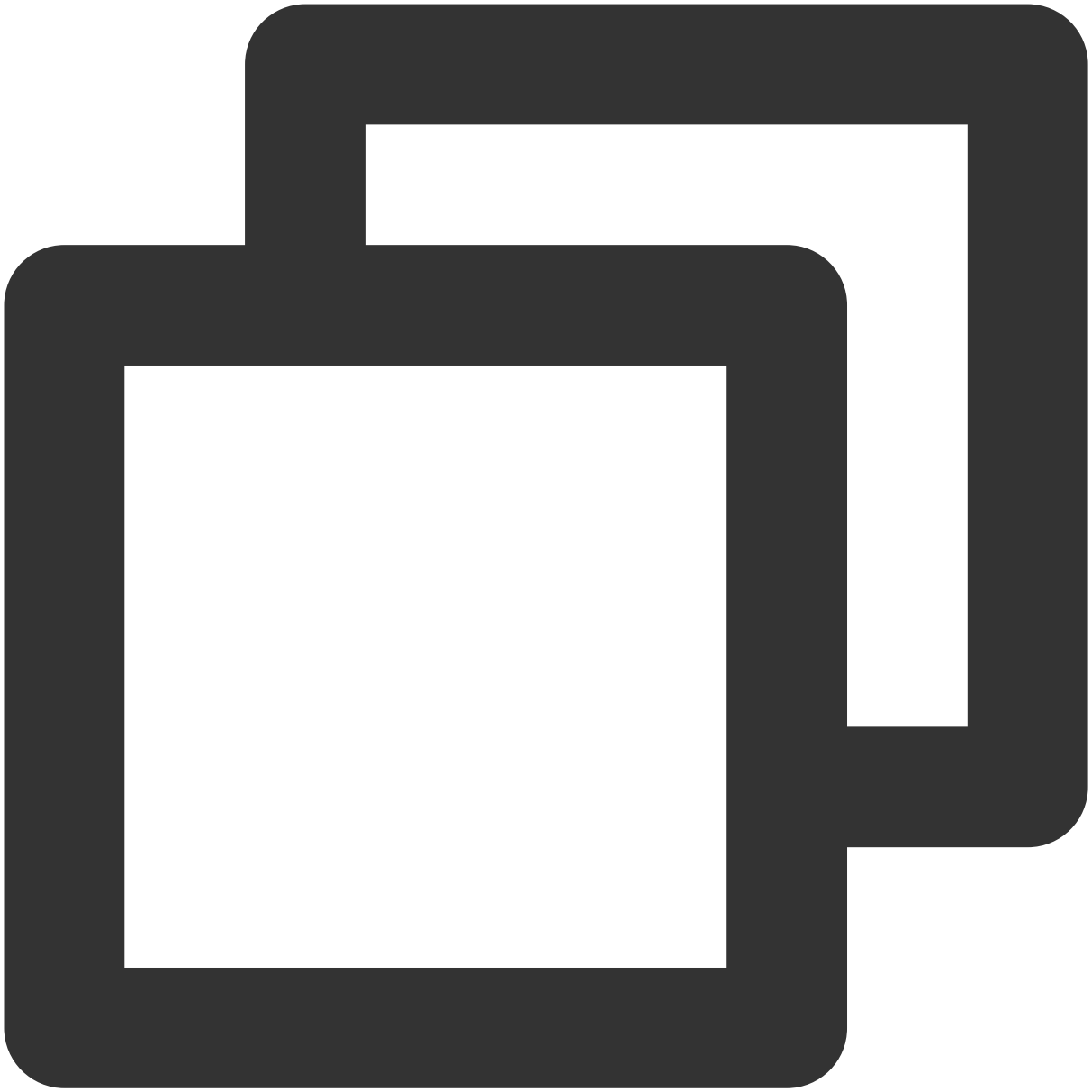
Field	Type	Description
observer	TXLivePusherObserver	The callback target object for the stream push.

Note:

Currently, some callback event notifications, such as `onError`, `onWarning`, `onCaptureFirstAudioFrame`, `onCaptureFirstVideoFrame` can also be obtained through the Promise object returned by the corresponding interface. Users can opt for the method to receive the respective event notifications based on their usage preferences. `For instance:`
`startCamera().then()` is equivalent to `onCaptureFirstVideoFrame()`, the status of successful acquisition of the first frame of video can also be achieved.
`startCamera().catch()` is equivalent to `onWarning()`, also it helps in obtaining the error when failing to turn on the camera.

destroy

Upon departure from the page or exit, ensure to purge the SDK instance to avoid potential memory leaks. Execute the 'stop' method prior to invoking other functions.

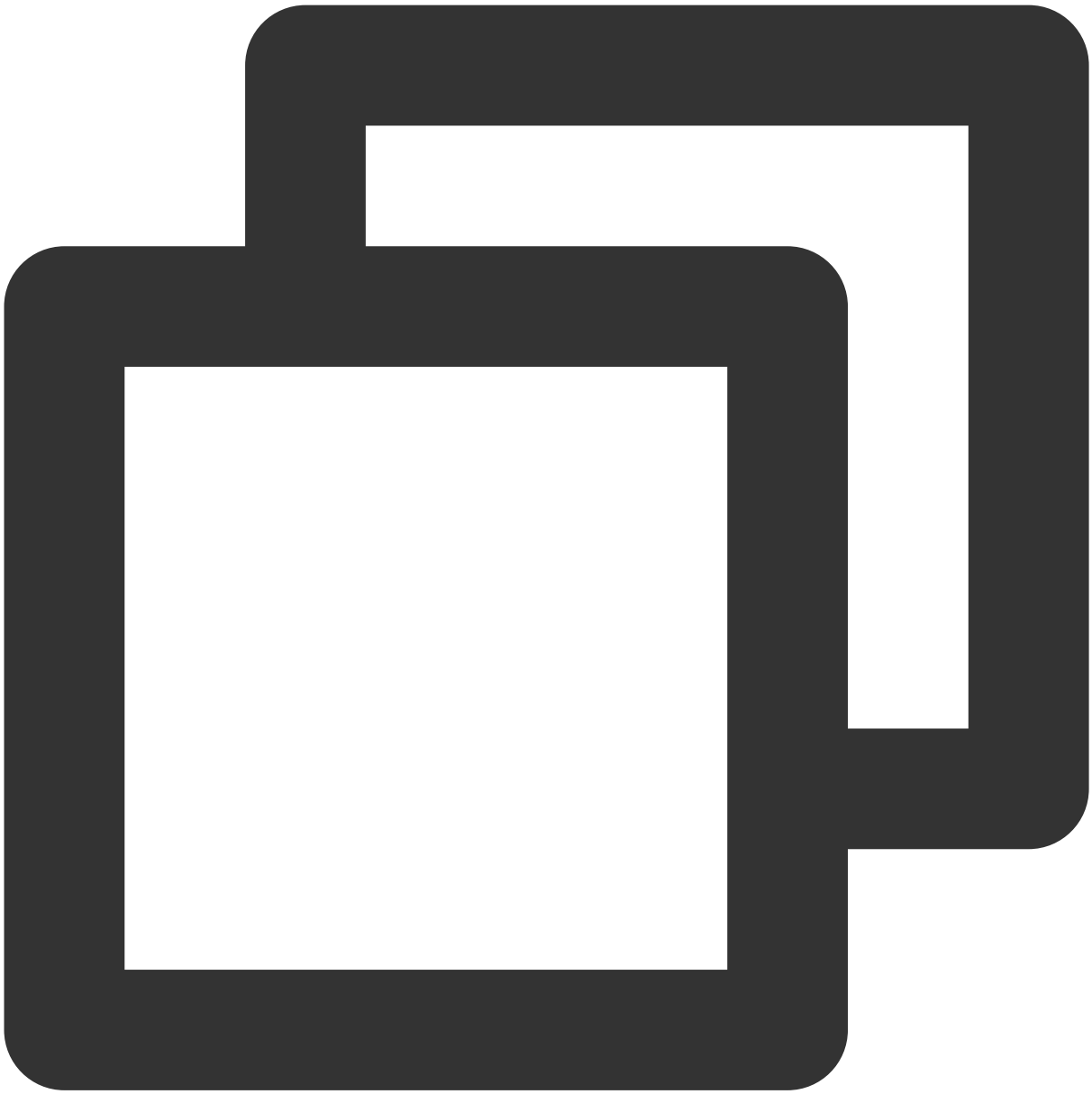


```
destroy(): void;
```

TXDeviceManager

Device Management Interface. Principally used for the administration of cameras and microphone devices. Should the local stream mix feature be toggled on, the corresponding stream IDs are necessitated for operation.

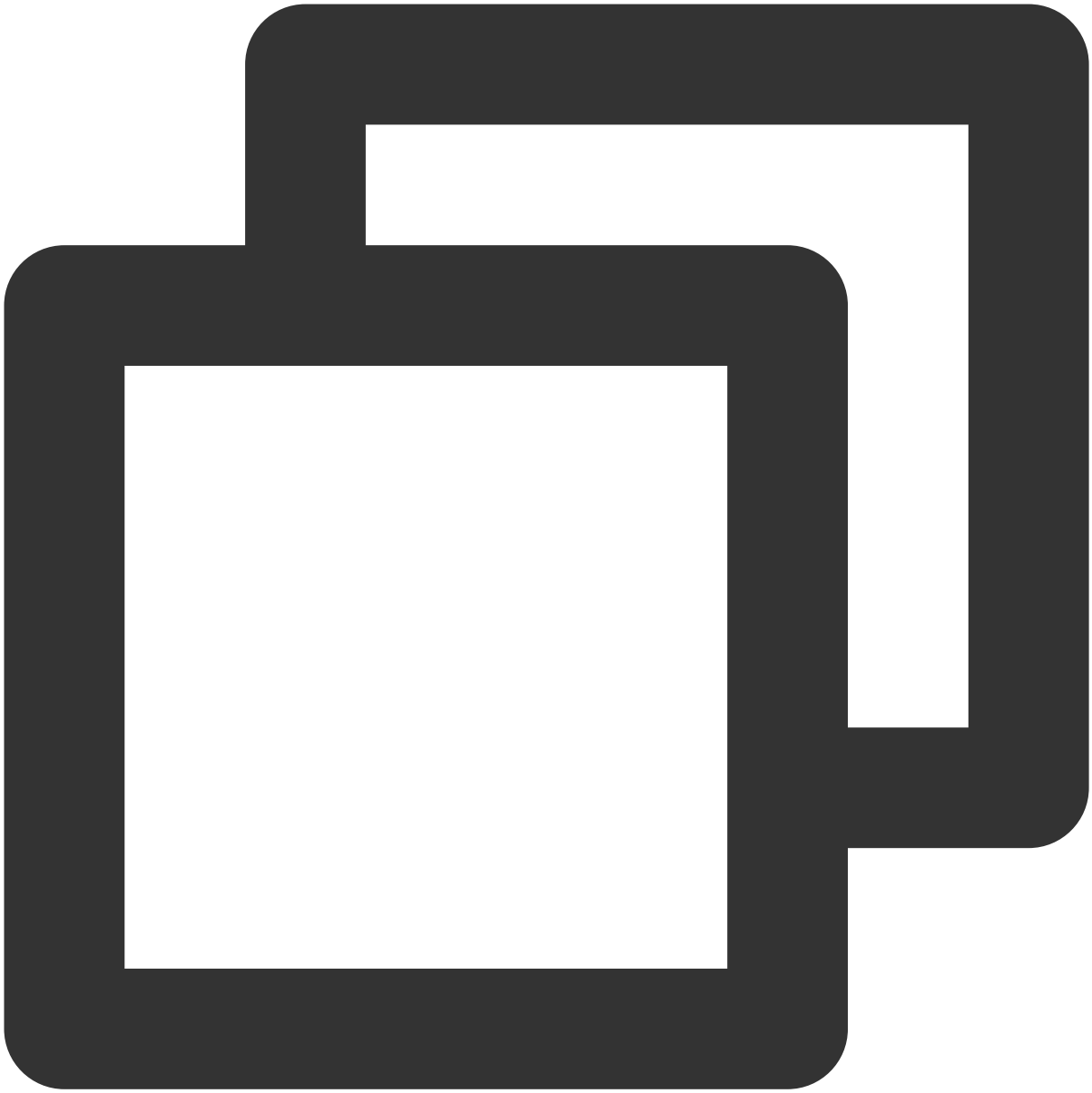
Obtain object instances through the `TXLivePusher` method getDeviceManager().



```
const deviceManager = livePusher.getDeviceManager();
```

getDevicesList

Acquire the device list.



```
getDevicesList(type?: string): Promise<TXMediaDeviceInfo[]>;
```

The parameters are described as follows:

Field	Type	Description
type	string	Taking value **video** or **audio**, optional parameter. If not transmitted, return all device list, transmit video for camera device list, transmit audio for microphone device list.

Return:

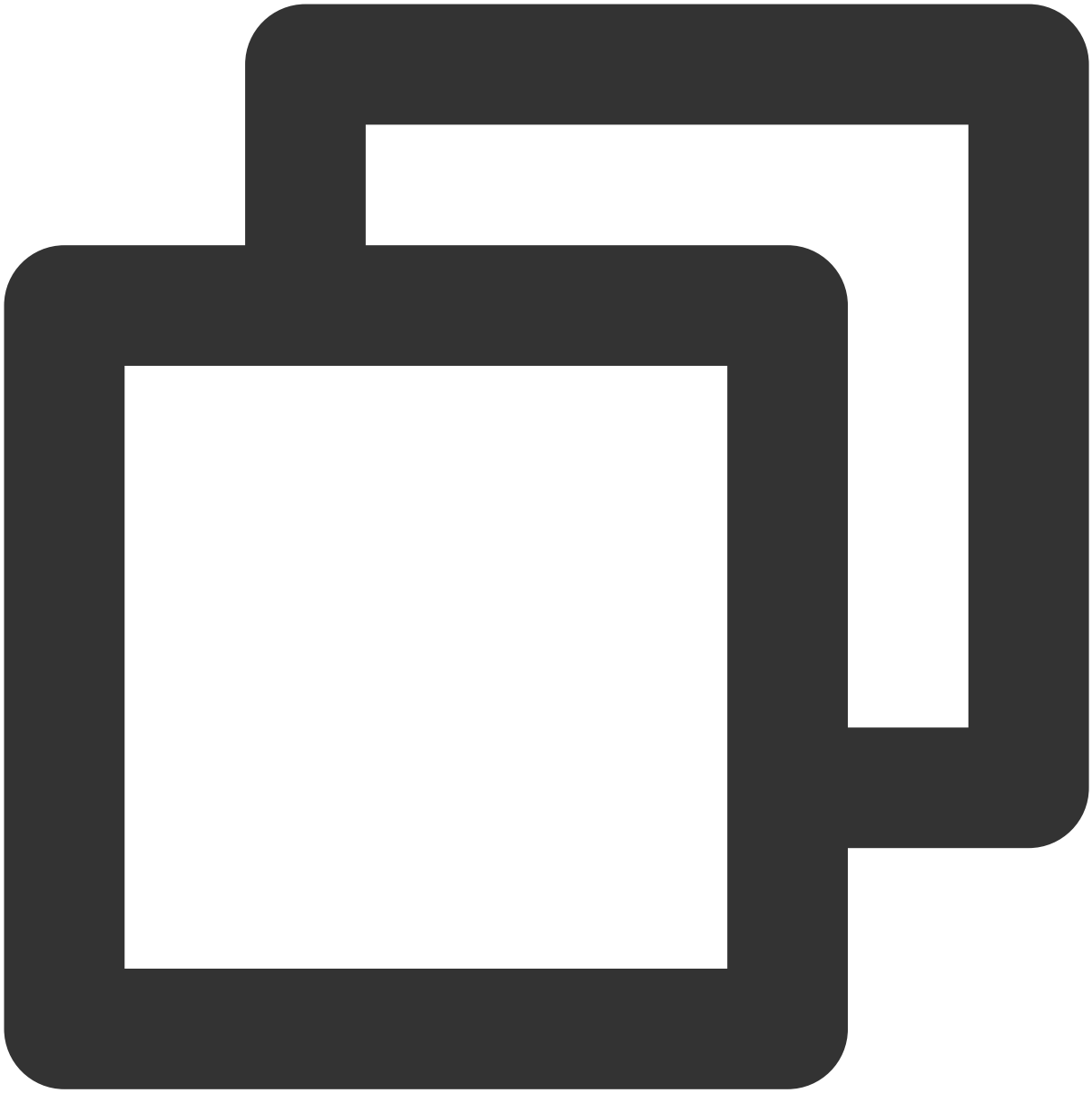
Returns a Promise object. For the structure of the device information, please refer to TXMediaDeviceInfo .

Note:

1. This interface does not support usage under the HTTP protocol; please deploy your website using the HTTPS protocol.
2. For security reasons, the browser may leave the 'deviceId' and 'deviceName' fields blank until the user grants access to the camera or microphone. Therefore, it is recommended to call this interface to get device details after the user has granted access.

getCurrentDevice

Acquire the device information of the current stream. If local stream mix is activated, you must stipulate the stream ID.



```
getCurrentDevice(type: string, streamId?: string): Promise<TXMediaDeviceInfo>;
```

The parameters are described as follows:

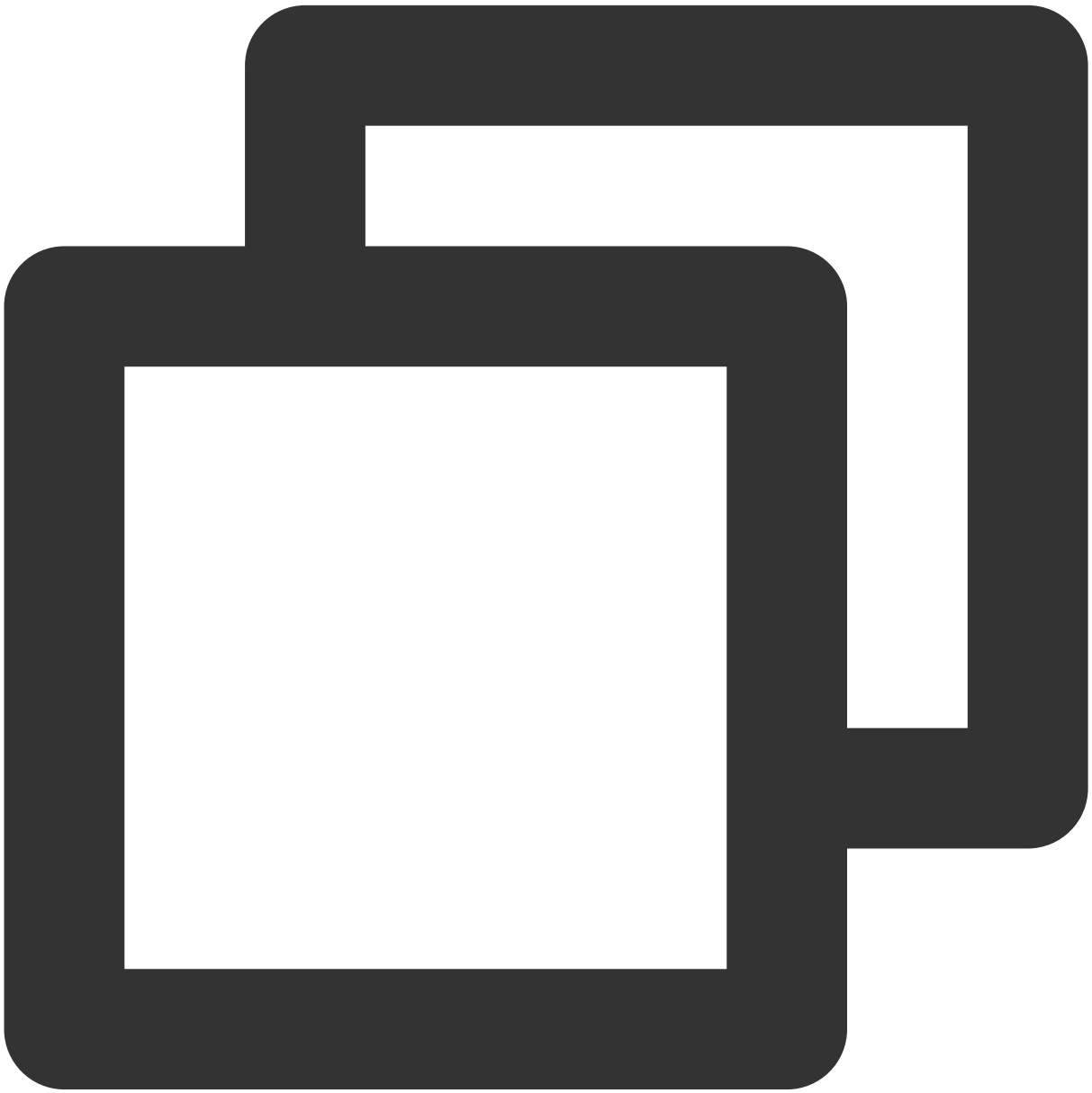
Field	Type	Description
type	string	Device type: video - camera device, audio - microphone device.
streamId	string	Stream ID refers to the stream from which device information is to be fetched. This is a mandatory variable after enabling local stream mixing, corresponding to the stream collected by the camera or microphone devices.

Return:

Returns a Promise object. For the structure of the device information, please refer to TXMediaDeviceInfo .

switchDevice

Switch the device currently in use. If local stream mix is enabled, a stream ID must be specified.



```
switchDevice(type: string, deviceId: string, streamId?: string): Promise<void>;
```

The parameters are described as follows:

Field	Type	Description
type	string	Device type: video - camera device, audio - microphone device.
deviceId	string	The device ID can be obtained by calling getDevicesList().
streamId	string	Stream ID, designates the stream of the device to be switched, it is required when local mix stream is enabled. The corresponding stream must be a stream collected by a camera or a microphone device.

Return:

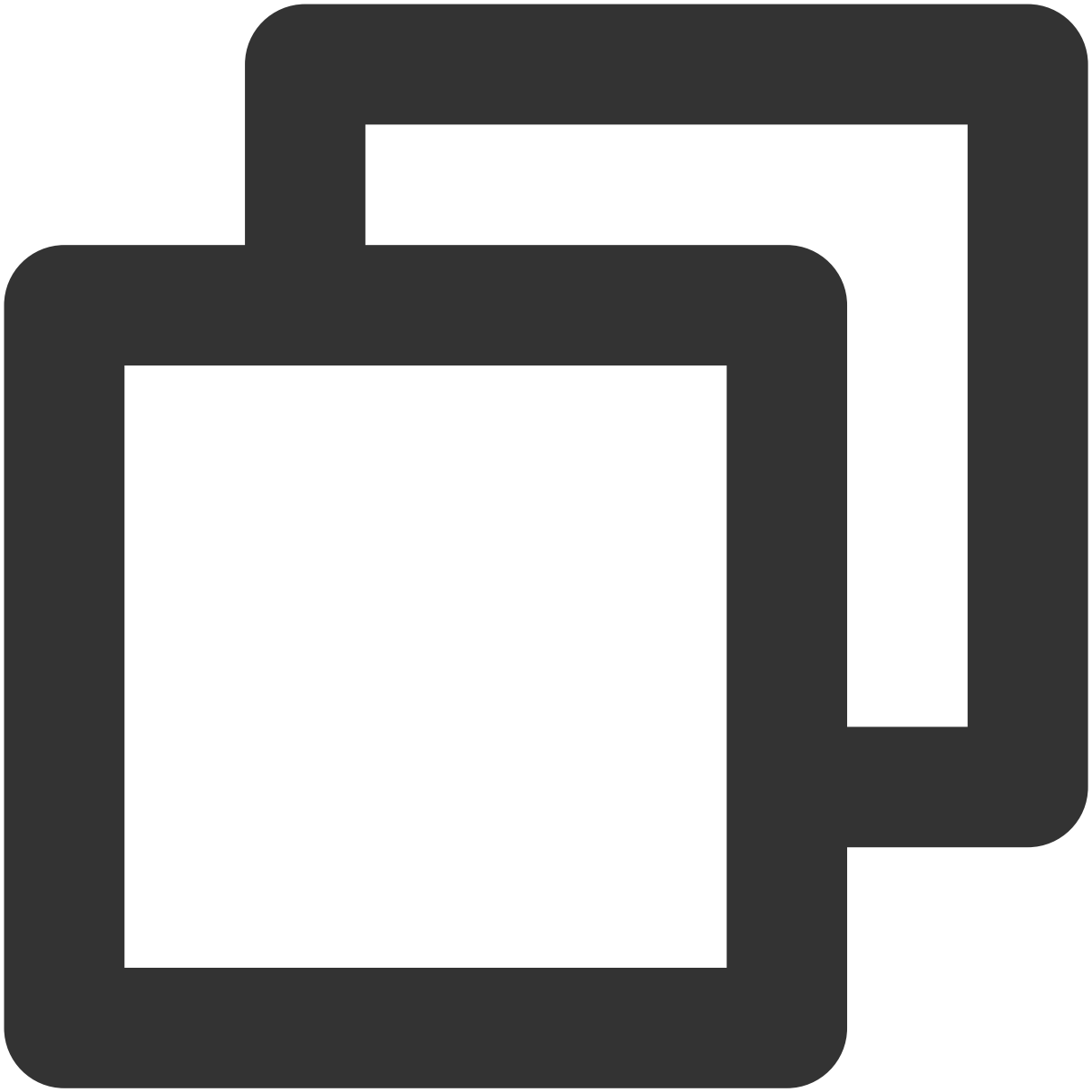
Returns a Promise object.

Note:

1. This method is only applicable for calls when collecting audio and video from a camera and microphone, streams collected in other ways do not support calling this API.
2. If streaming has not yet commenced, the local stream alone is updated; but if streaming is already underway, the audio and video streams being pushed to the server are concurrently updated.
3. When switching devices for a given stream, the corresponding stream ID remains constant.
4. Upon specifying the stream ID, the corresponding stream type should align with the type, such as when type is video, the corresponding stream must be camera-captured video stream.
5. It is advisable to directly employ switchCamera() and switchMicrophone() .

switchCamera

Transition camera equipment. Equivalent to `switchDevice('video', deviceId, streamId)` .



```
switchCamera(deviceId: string, streamId?: string): Promise<void>;
```

The parameters are described as follows:

Field	Type	Description
deviceId	string	The device ID can be acquired by invoking getDevicesList(). On mobile devices, switching between the front and rear cameras can be facilitated by entering 'user' and 'environment'.
streamId	string	The stream ID specifies the stream of the camera device to be switched. Once local

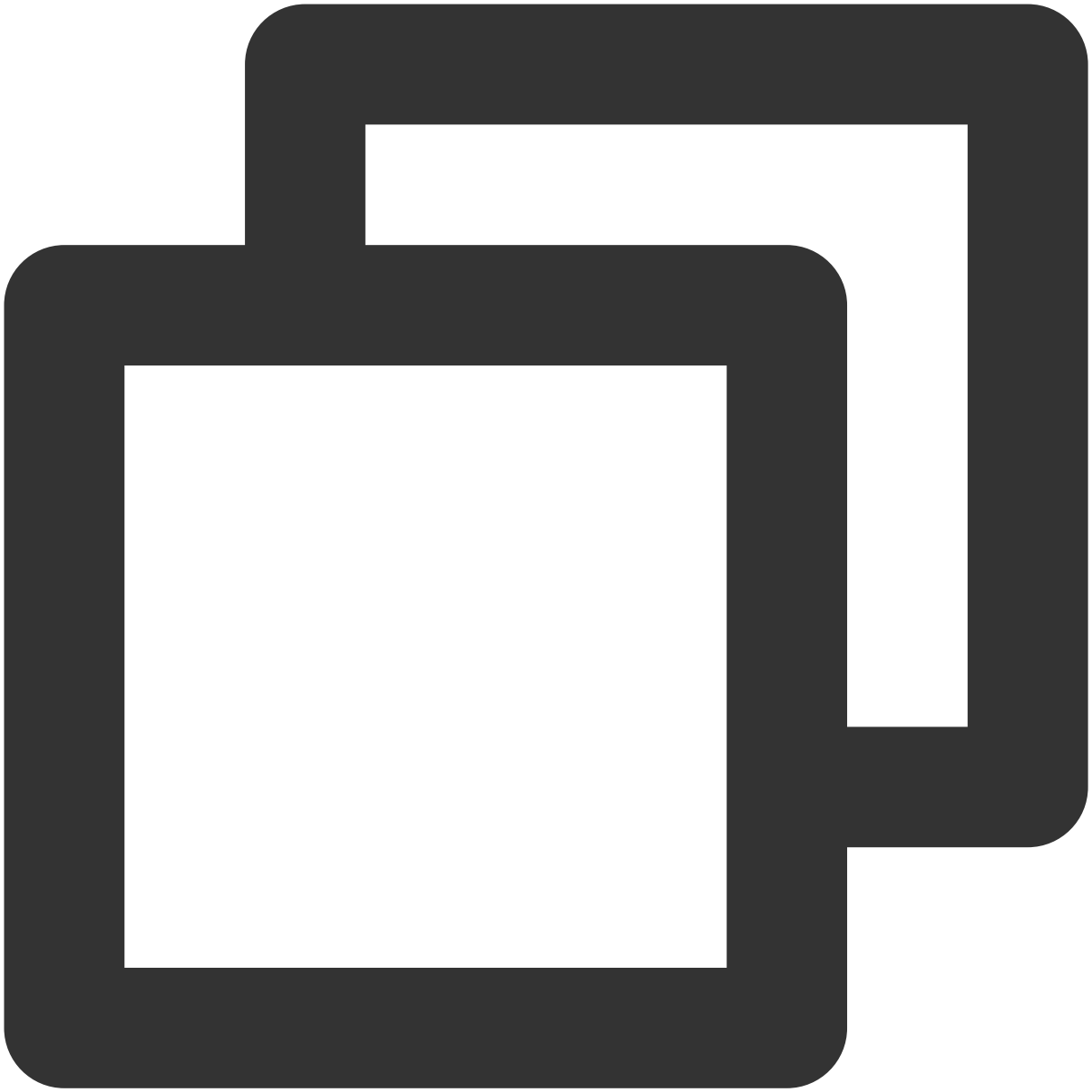
		stream mix is activated, it's a mandatory field. The corresponding stream must be the one collected by the camera device.

Return:

Returns a Promise object.

switchMicrophone

Switching microphone device. This means `switchDevice('audio', deviceId, streamId)` .



```
switchMicrophone(deviceId: string, streamId?: string): Promise<void>;
```

The parameters are described as follows:

Field	Type	Description
deviceId	string	The Device ID can be obtained by invoking the getDevicesList() function.
streamId	string	Stream ID, identifies the stream for which the microphone device will be changed, and must be supplied after enabling local mixed-flow. The corresponding stream must be a stream collected from the microphone device.

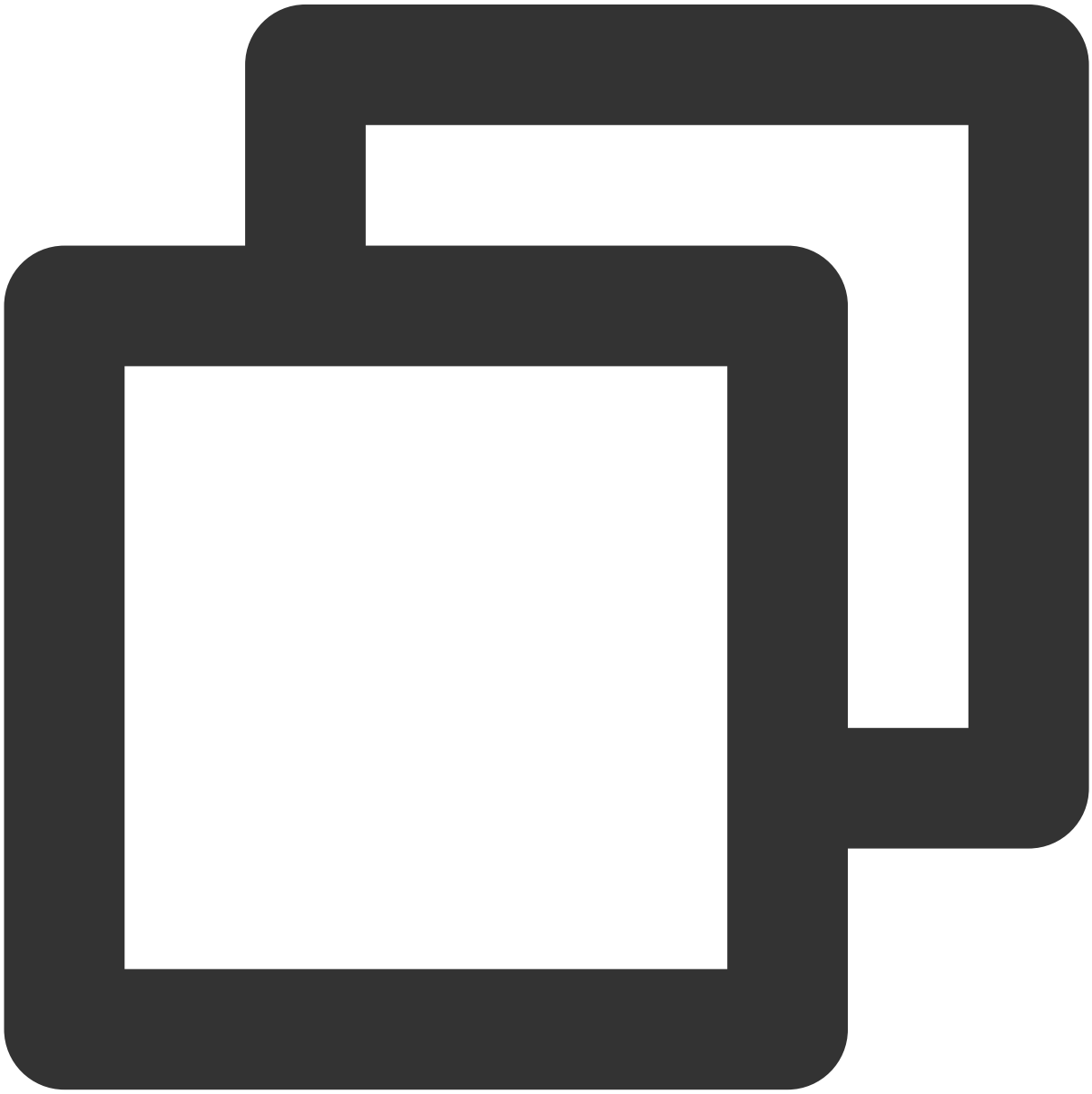
Return:

Returns a Promise object.

TXAudioEffectManager

An audio effect management interface, primarily employed for adjusting volume operations. If the local mixed-flow feature is activated, it is essential to pass in the corresponding stream ID for operation.

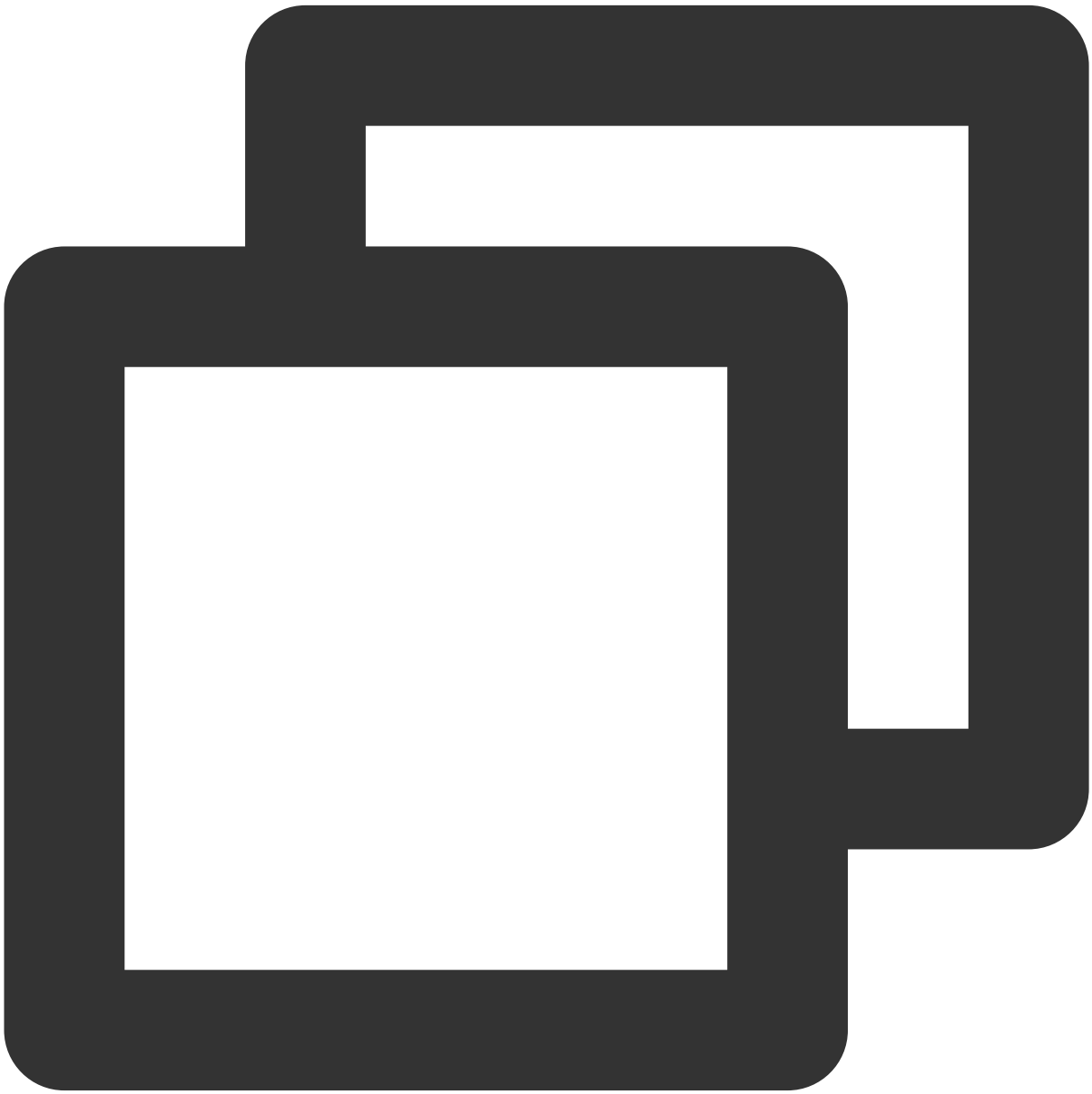
An object instance can be retrieved through the `TXLivePusher` method getAudioEffectManager().



```
const audioEffectManager = livePusher.getAudioEffectManager();
```

setVolume

Set the volume level of the audio stream. If local stream mixing is enabled, you must specify the stream ID.



```
setVolume(volume: number, streamId?: string): void;
```

The parameters are described as follows:

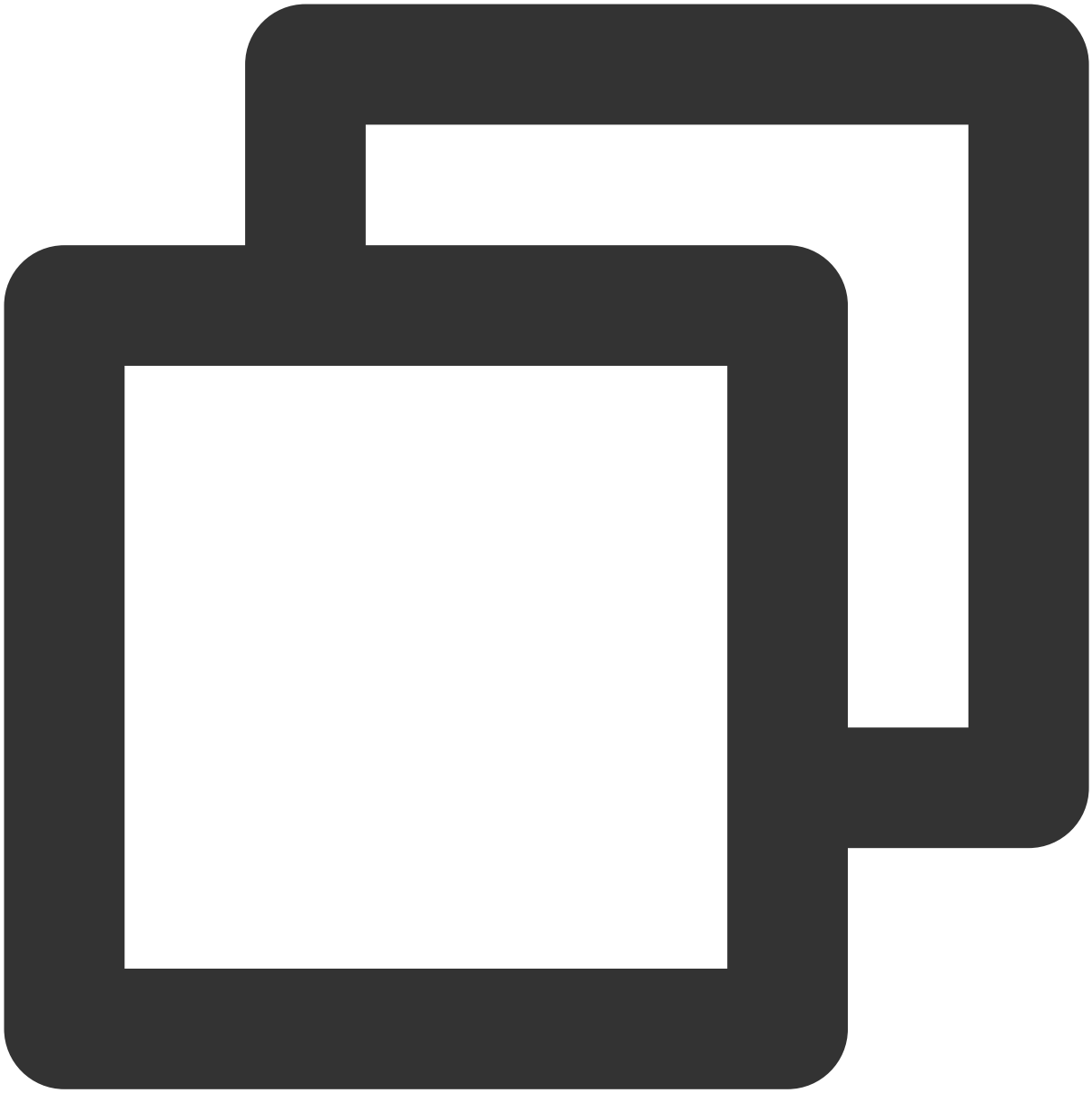
Field	Type	Description
volume	number	The volume level, which ranges from 0 to 100, with the default value being 100.
streamId	string	Stream ID, designates the stream to be set, and must be transmitted after enabling local stream blending.

Note:

If you feel the volume is still too low after setting the volume to 100, you can set the volume above 100, however, there's a risk of distortion with volumes exceeding 100, please handle with caution.

TXVideoEffectManager

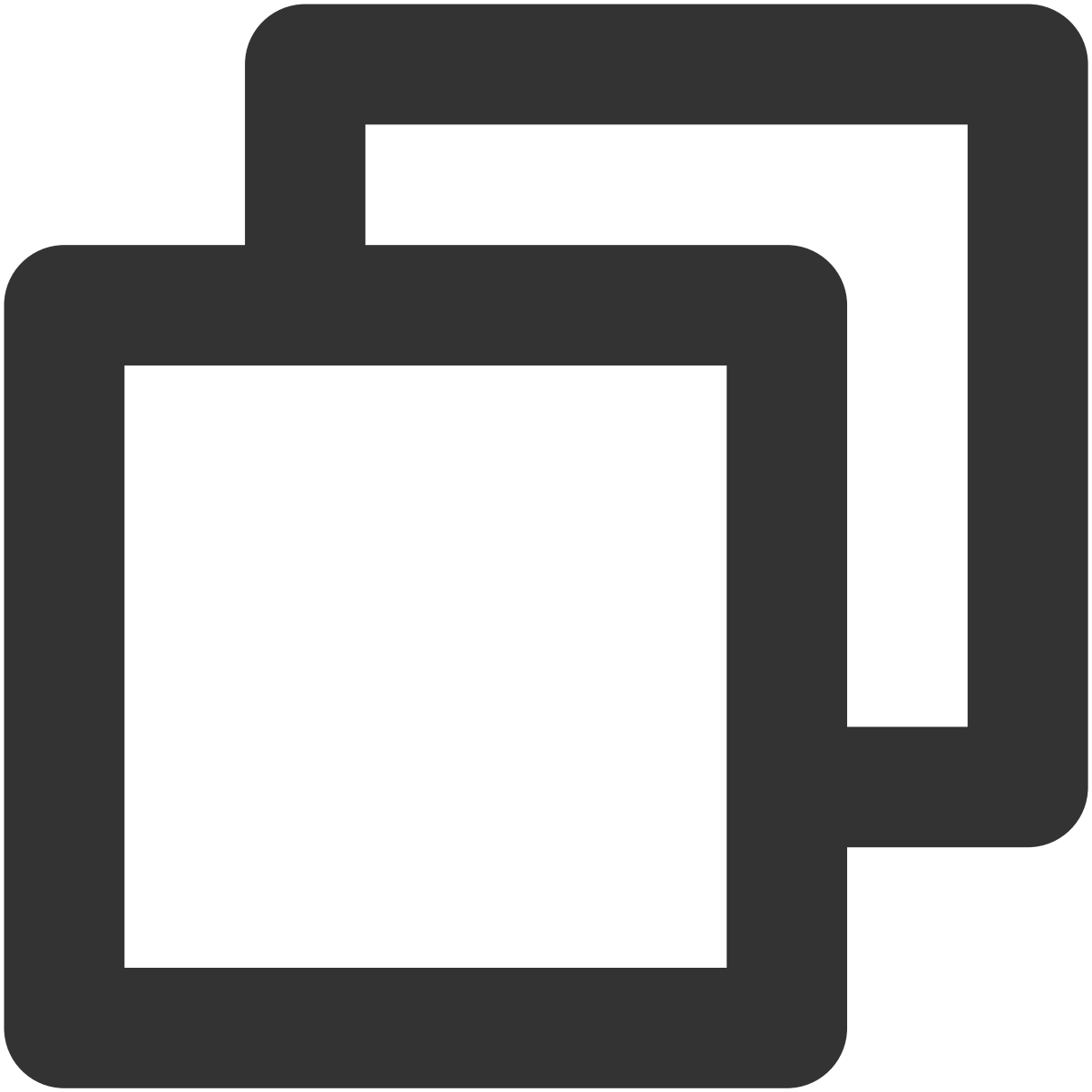
Video effect management interface. Principally employed for operations such as picture-in-picture, mirror, filters, watermarks, text and more. It's necessary to first invoke the enableMixing() interface to enable these features. Acquire the object instance via the `TXLivePusher` method getVideoEffectManager().



```
const videoEffectManager = livePusher.getVideoEffectManager();
```

enableMixing

Initiate the function for local video frame mix.



```
enableMixing(enabled: boolean): void;
```

The parameters are described as follows:

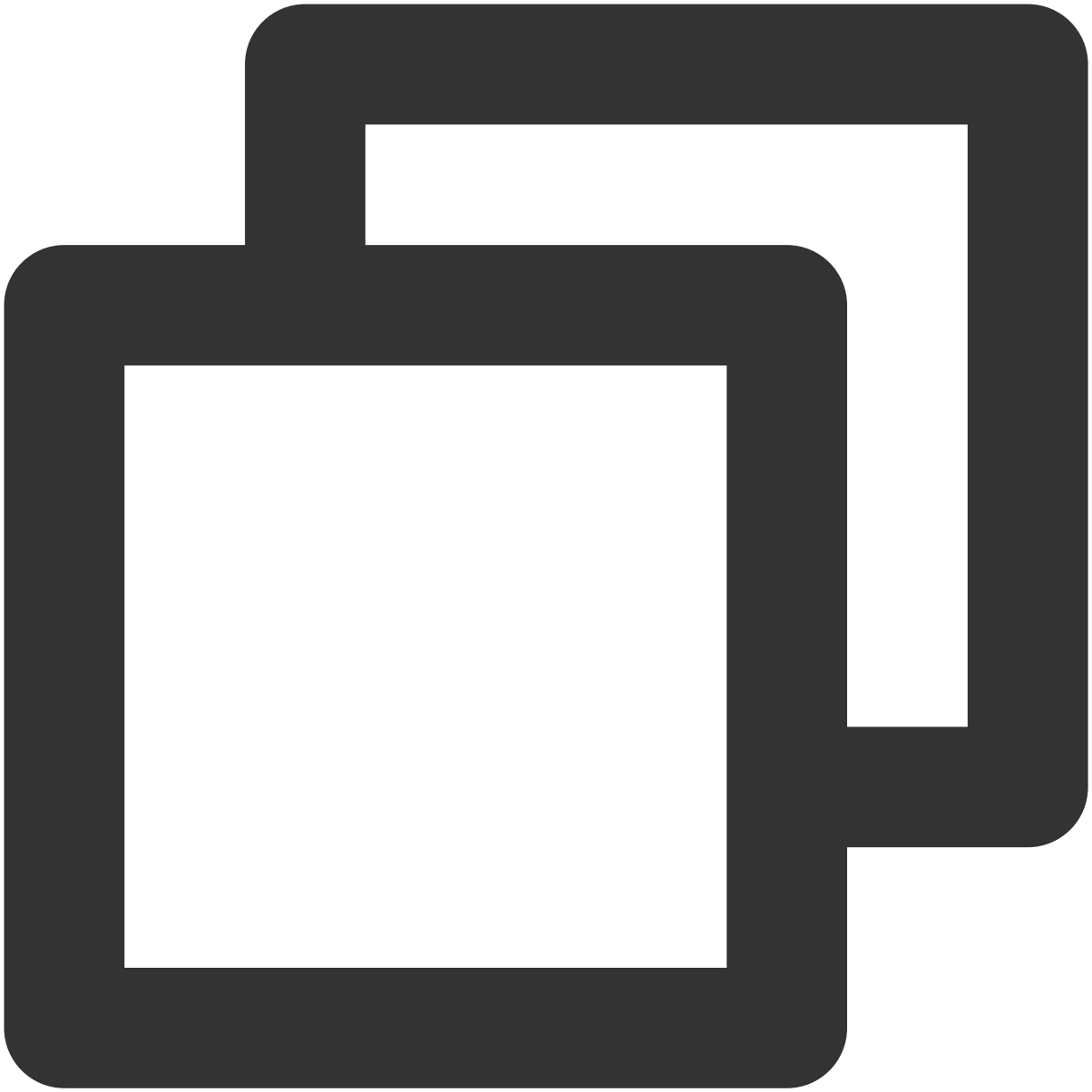
Field	Type	Description
enabled	boolean	Whether to activate local stream mix feature; it is deactivated by default.

Note:

1. Prior to enabling the local stream mix, the previewed and pushed streams are purely raw captures. Upon activation, these streams undergo rendering through local browser mixing, incurring some performance overhead.
2. It is requisite to enable local stream mixing by invoking `TXVideoEffectManager` before deploying other methods.
3. Without the local stream mix activated, only a singular video stream and audio stream can be captured.
4. After successfully activating local mix stream, various streams can be captured, such as executing two `startCamera` to capture two disparate camera visuals, or initially executing `startCamera` to capture the camera visual, followed by `startScreenCapture` to capture screen visual. The multi-stream captures, both visual and audio, appear in the final merged output.

setMixingConfig

Set the stream mix parameters. When this interface is not invoked for settings, the default mix parameters are directly obtained from the methods `TXLivePusher` in setVideoQuality() and setProperty().



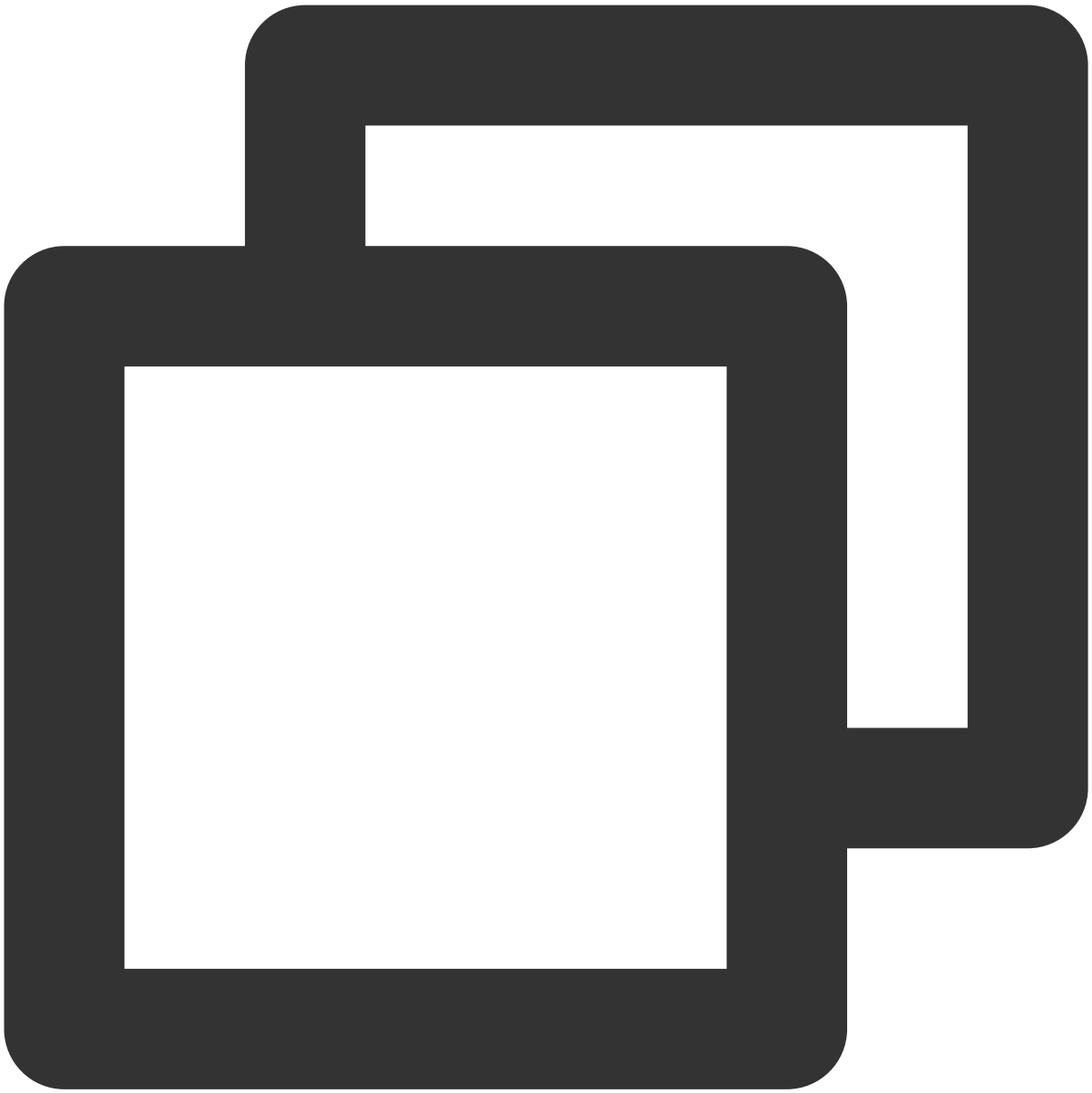
```
setMixingConfig(config: TXMixingConfig): void;
```

The parameters are described as follows:

Field	Type	Description
config	TXMixingConfig	Configuration of mix streaming parameters.

getMixingConfig

Fetch the final adopted parameters of stream mixing. Primarily use the results set by setMixingConfig(), followed by the ones from the `TXLivePusher` methods in setVideoQuality() and setProperty().



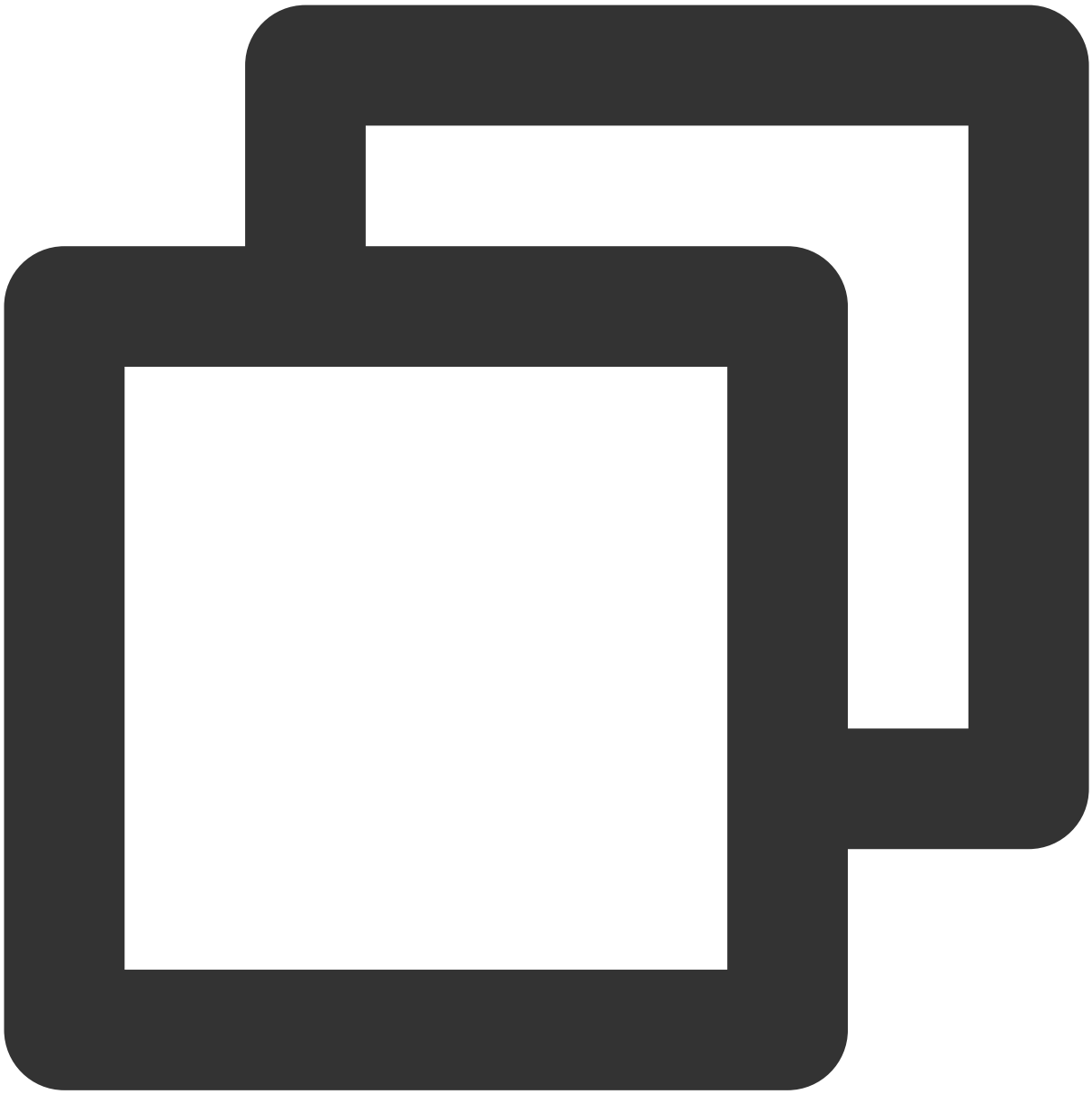
```
getMixingConfig(): TXMixingConfig;
```

Return:

The configuration setup for stream mix parameters, data structure please refer to TXMixingConfig.

setLayout

Setting the Picture-in-Picture layout parameters for the video stream.



```
setLayout(config: TXLayoutConfig | TXLayoutConfig[]): void;
```

The parameters are described as follows:

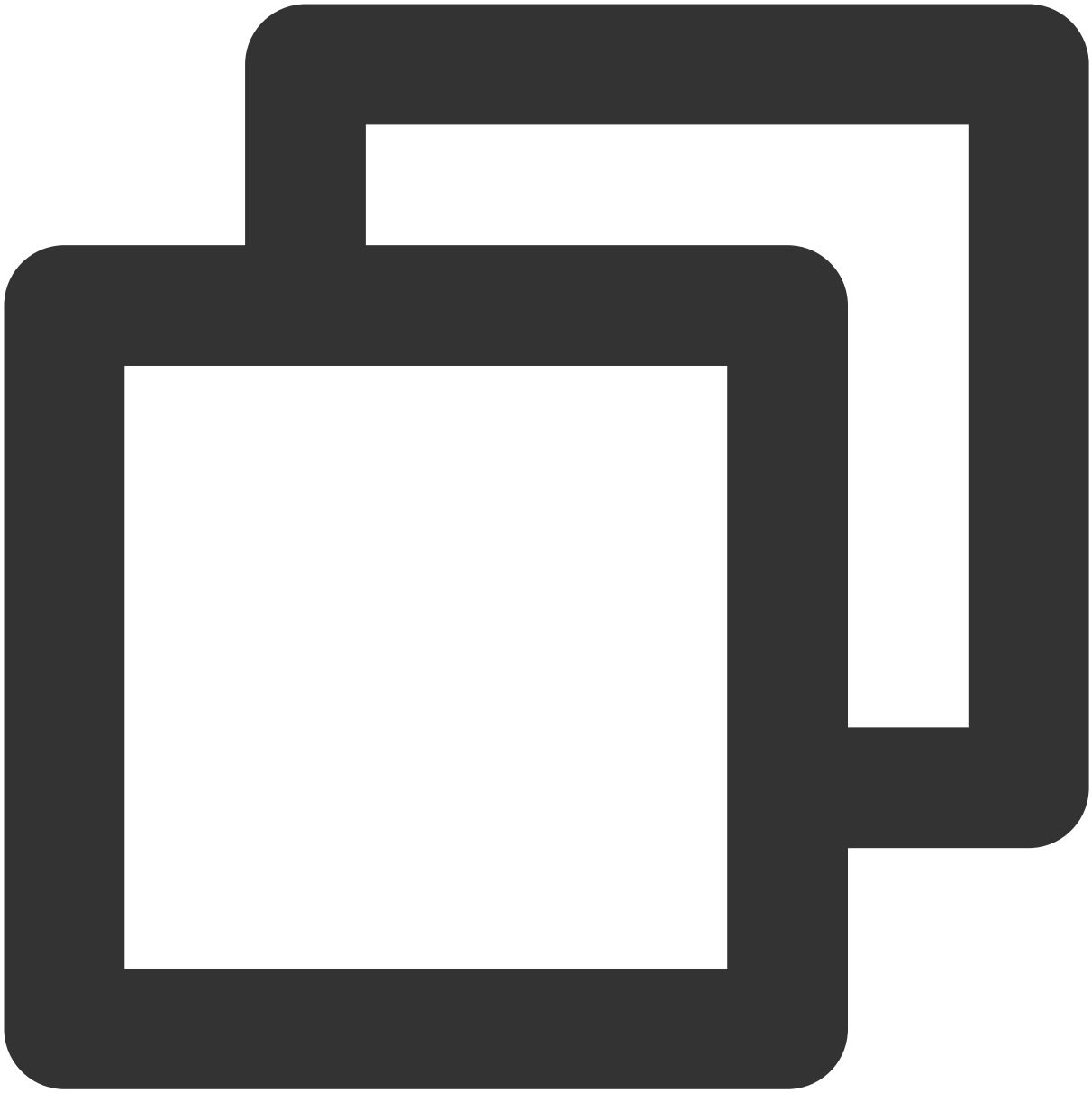
Field	Type	Description
config	TXLayoutConfig | TXLayoutConfig[]	The configuration of the PiP layout supports the passage of an object or an array of objects.

Note:

1. After enabling local stream mixing, all collected streams will automatically be added to the final output video stream. By default, the layout parameters ensure that the video stream image appears snugly in the upper left corner of the origin.
2. Configuration parameters may be passed as an array of objects, for bulk setting of multiple stream's PiP layout effects, or you may simply pass an object to set a specific stream individually.
3. If you do not wish to display the image of the collected stream, preferring only to retain the sound, you may set the layout width and height to 0.

getLayout

Retrieve the PiP layout parameters for a specific stream.



```
getLayout(streamId: string): TXLayoutConfig | null;
```

The parameters are described as follows:

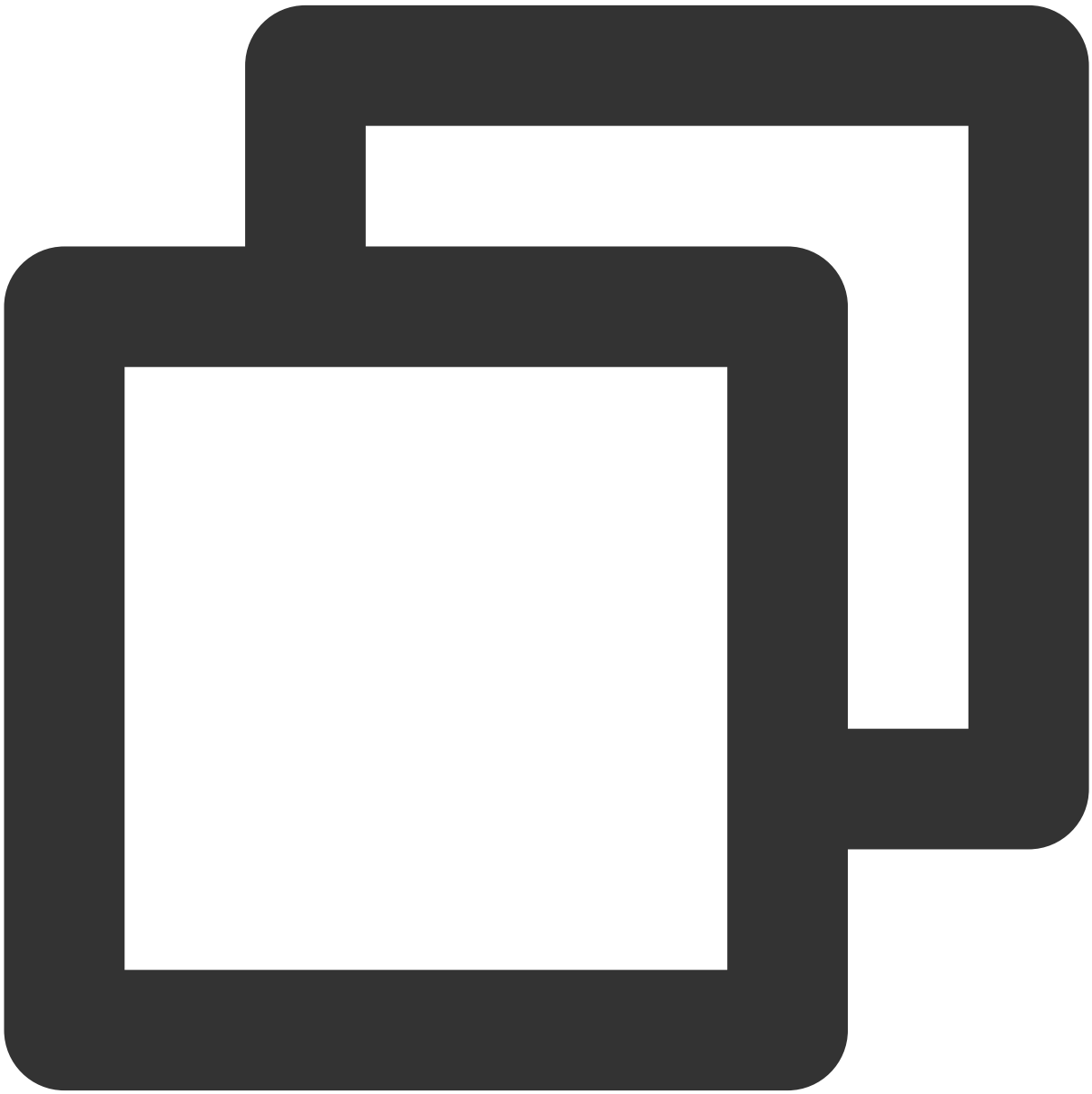
Field	Type	Description
streamId	string	Stream ID, signifies the stream from which you wish to acquire the Picture-in-Picture layout parameters.

Return:

Returns the Picture-in-Picture layout parameters for the specified stream; for the data structure, please consult TXLayoutConfig. If the stream does not exist, it returns null.

setMirror

Specify the mirroring effect of the video stream, including horizontal and vertical mirrors.



```
setMirror(config: TXMirrorConfig | TXMirrorConfig[]): void;
```

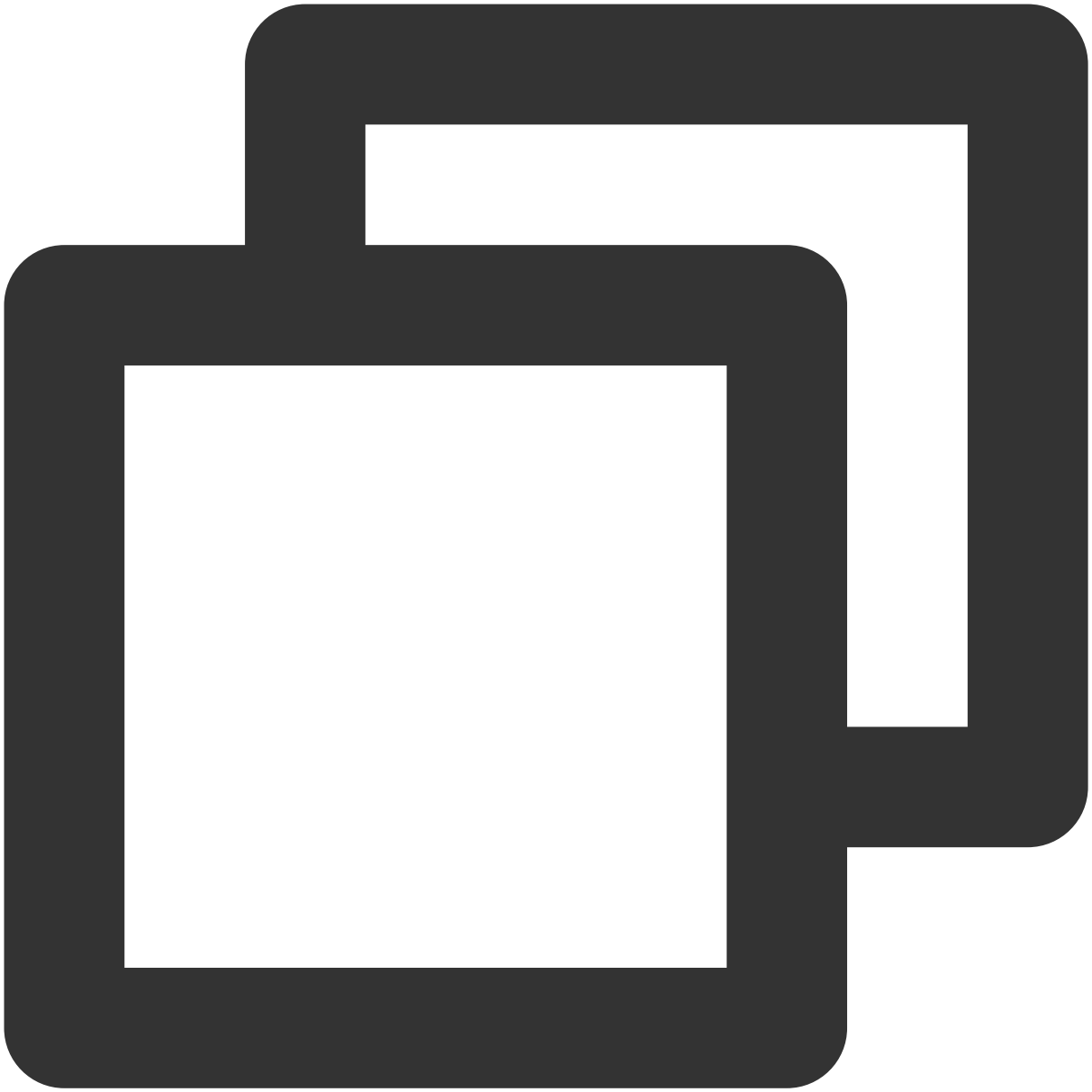
The parameters are described as follows:

Field	Type	Description
config	TXMirrorConfig | TXMirrorConfig[]	Configuration for the mirror effect permits both individual and batch setting via object or object array.

Note:
Configuration parameters can be passed as an array, allowing the batch setting of multiple streams' mirror effects, or as an individual object for a specific stream.

setNormalFilter

Establish the general filter effect of the video stream, encompassing contrast, brightness, and saturation.



```
setNormalFilter(config: TXNormalFilterConfig | TXNormalFilterConfig[]): void;
```

The parameters are described as follows:

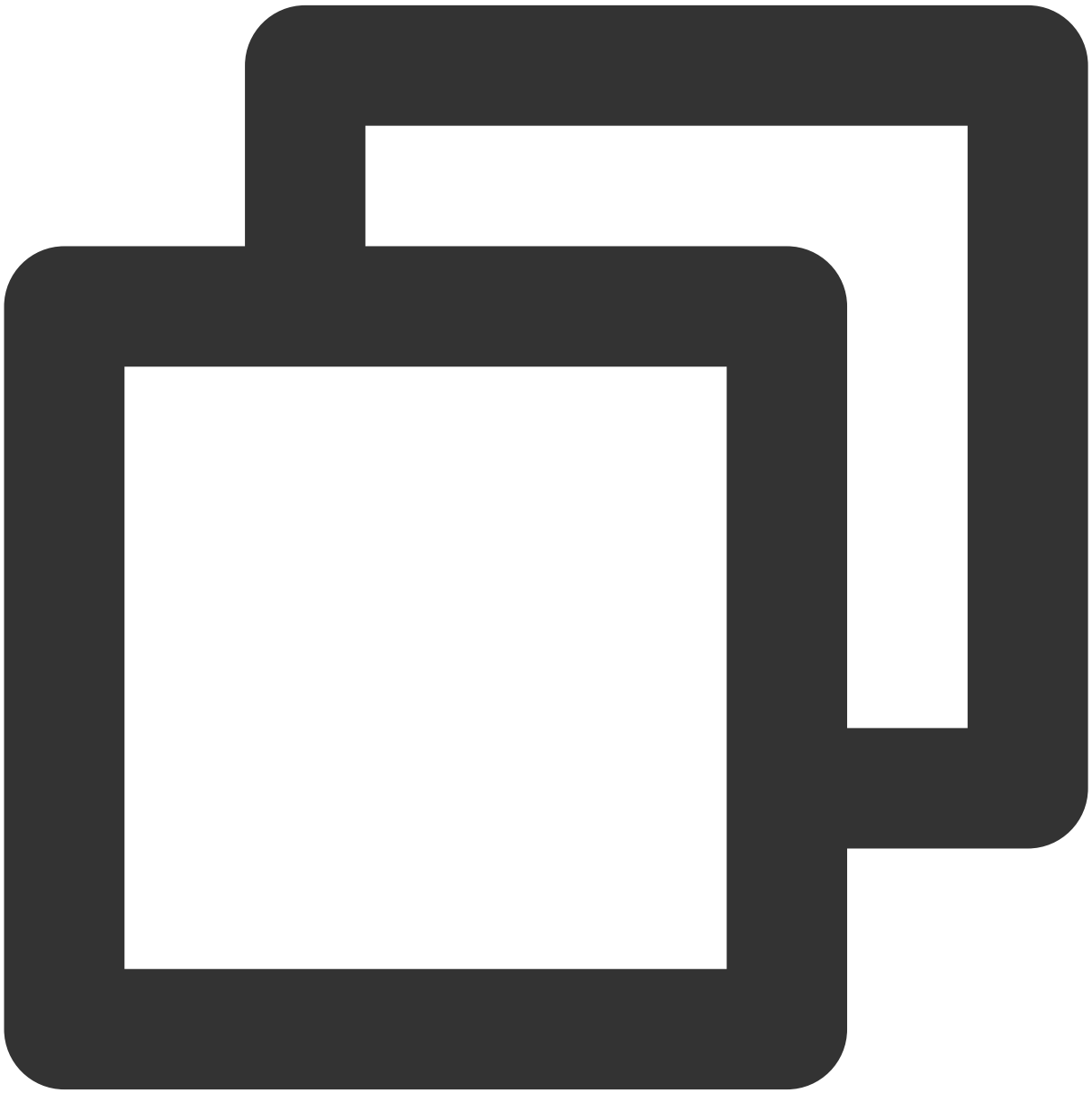
Field	Type	Description
config	TXNormalFilterConfig | TXNormalFilterConfig[]	The configuration for the general filter effect supports object or object array transmission.

Note:

Configuration parameters can be transmitted as object arrays for the batch setting of general filter effects on multiple streams or as a solo object for specifically designated streams.

setWatermark

Establish watermarks, with support for simultaneous multiple watermark settings.



```
setWatermark(config: TXWatermarkConfig | TXWatermarkConfig[] | null): void;
```

The parameters are described as follows:

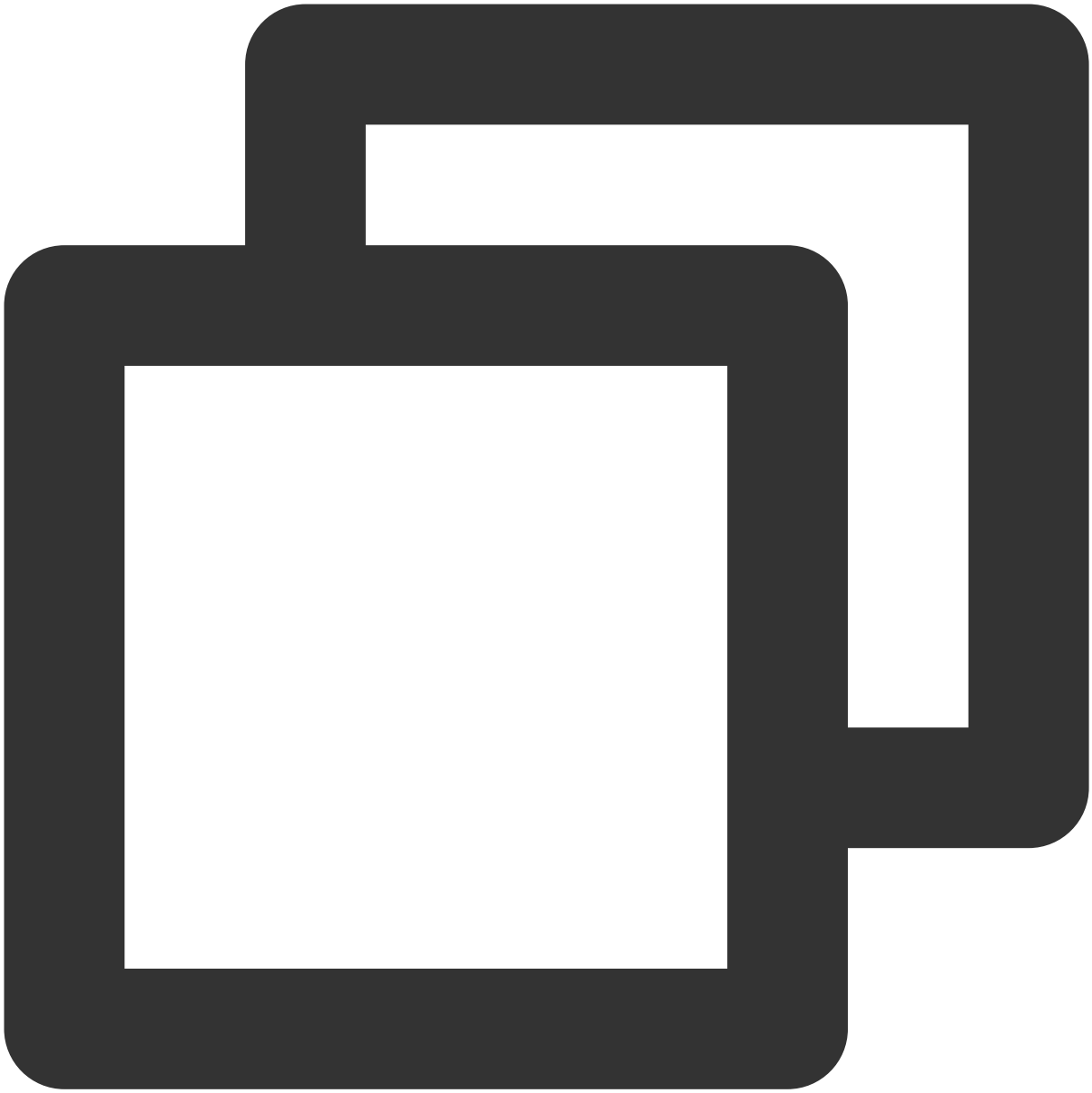
Field	Type	Description
config	TXWatermarkConfig | TXWatermarkConfig[] | null	Watermark configuration parameters allow for the transmission of objects, object arrays, or null.

Note:

1. Configuration parameters can pass object arrays, simultaneously setting multiple watermarks, or can pass a single object to set an individual watermark.
2. Passing null or an empty array as configuration parameters indicates the removal of existing watermarks.

setText

Setting text, supporting the simultaneous setup of multiple texts.



```
setText(config: TXTextConfig | TXTextConfig[] | null): void;
```

The parameters are described as follows:

Field	Type	Description
config	TXTextConfig | TXTextConfig[] | null	Text configuration parameters, supporting object passage, object arrays, or null.

Note:

1. Configuration parameters can pass object arrays, simultaneously setting multiple texts, or can pass a single object to set an individual text.
2. Passing null or an empty array for configuration parameters denotes deletion of existing text.

Type Definition

TXSupportResult

Browser Support Check Result.
Data Structure:

Field	Type	Description
isWebRTCSupported	boolean	Whether WebRTC is supported
isH264EncodeSupported	boolean	Is H.264 encoding supported?
isH264DecodeSupported	boolean	Is H.264 decoding supported?
isMediaDevicesSupported	boolean	Is acquisition of media devices and media streams supported?
isScreenCaptureSupported	boolean	Is screen capture supported?
isMediaFileSupported	boolean	Is retrieval of local media file streams supported?

TXLivePusherObserver

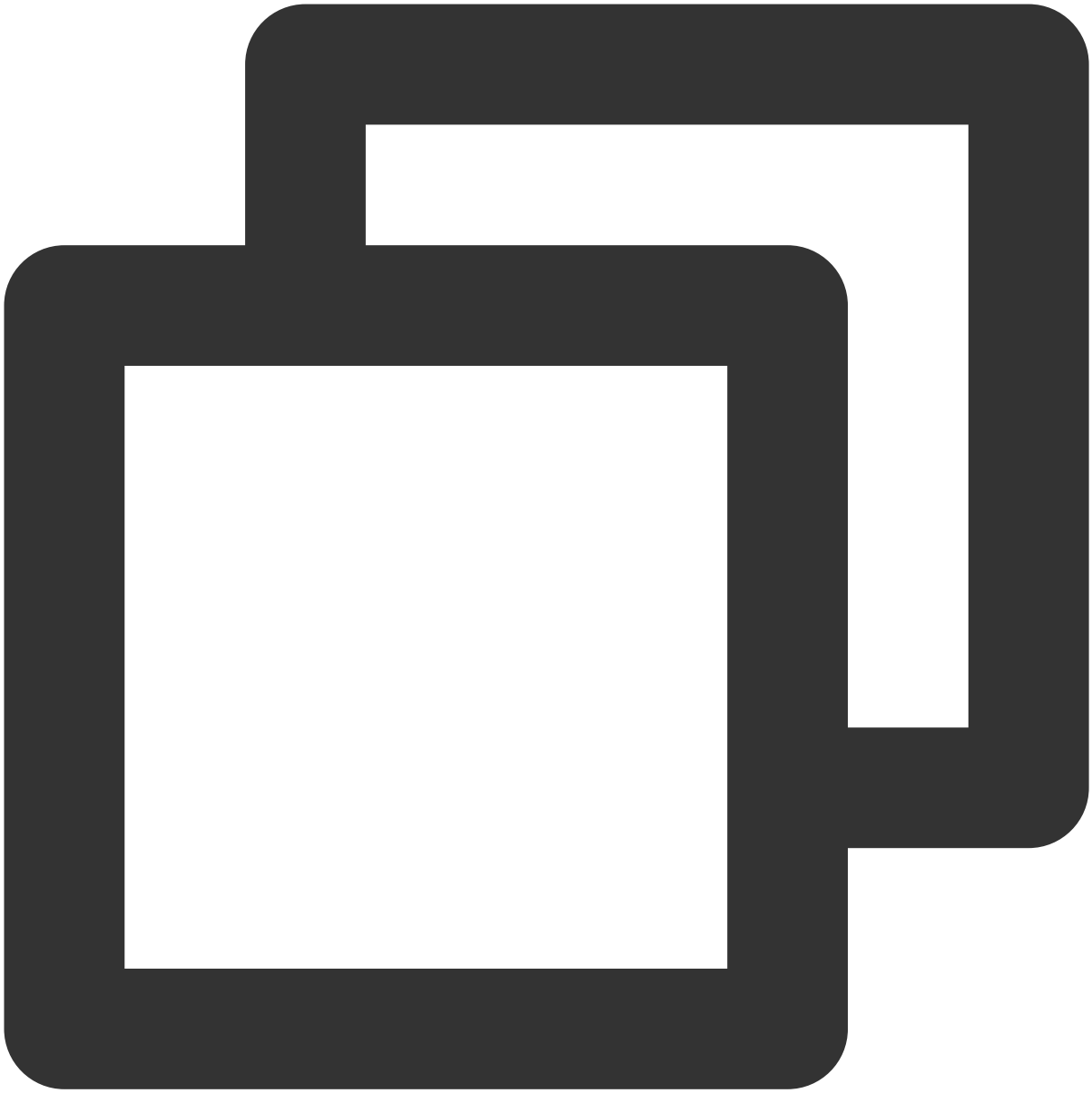
Callback notifications for the stream, which include the status of the streamer, statistical information, warnings, and error messages.
Data Structure:

Field	Type	Description
onError	onError	Notification of streaming errors.
onWarning	onWarning	Stream warning notification.
onCaptureFirstAudioFrame	onCaptureFirstAudioFrame	Callback notification completed for initial audio frame capture.

onCaptureFirstVideoFrame	onCaptureFirstVideoFrame	Callback notification upon completion of the first frame video collection.
onPushStatusUpdate	onPushStatusUpdate	Live stream connection status callback notification.
onStatisticsUpdate	onStatisticsUpdate	Callback notification of live stream statistical data.

onError

Callback for streaming errors. This callback is invoked when a streaming error occurs.



```
onError(code: number, message: string, extraInfo: object): void;
```

The parameters are described as follows:

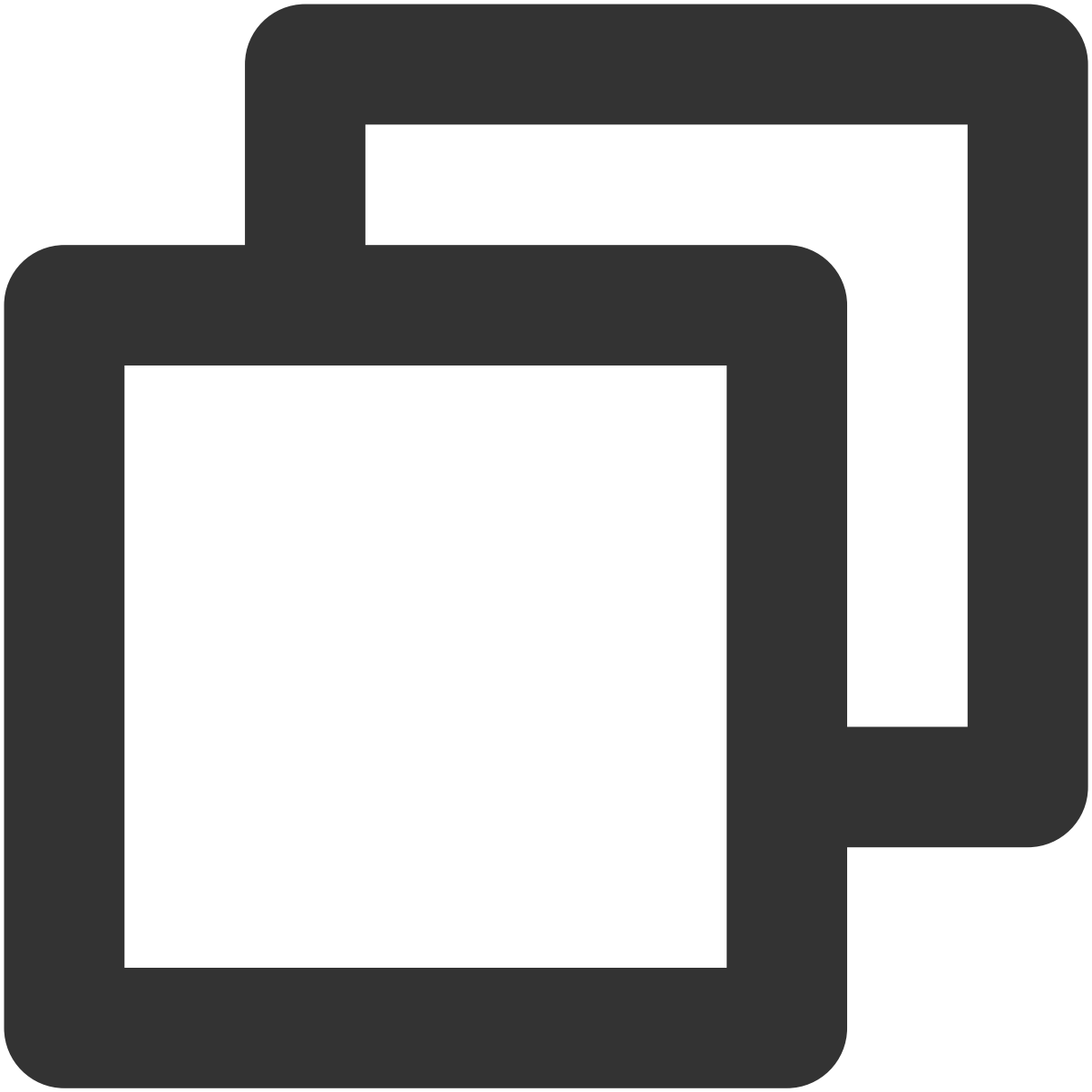
Field	Type	Description
code	number	Error Codes.
message	string	Error message.
extraInfo	object	Extension information.

Please see the below references for error codes:

Enumeration	Numerical Value	Description
TXLIVE_ERROR_WEBRTC_FAILED	-1	Failed to call the `WebRTC` API.
TXLIVE_ERROR_REQUEST_FAILED	-2	Server stream request interface returned with an error.

onWarning

Stream warning notification.



```
onWarning(code: number, message: string, extraInfo: object): void;
```

The parameters are described as follows:

Field	Type	Description
code	number	Error Codes.
message	string	Error message.
extraInfo	object	Extension information.

Please see the below references for error codes:

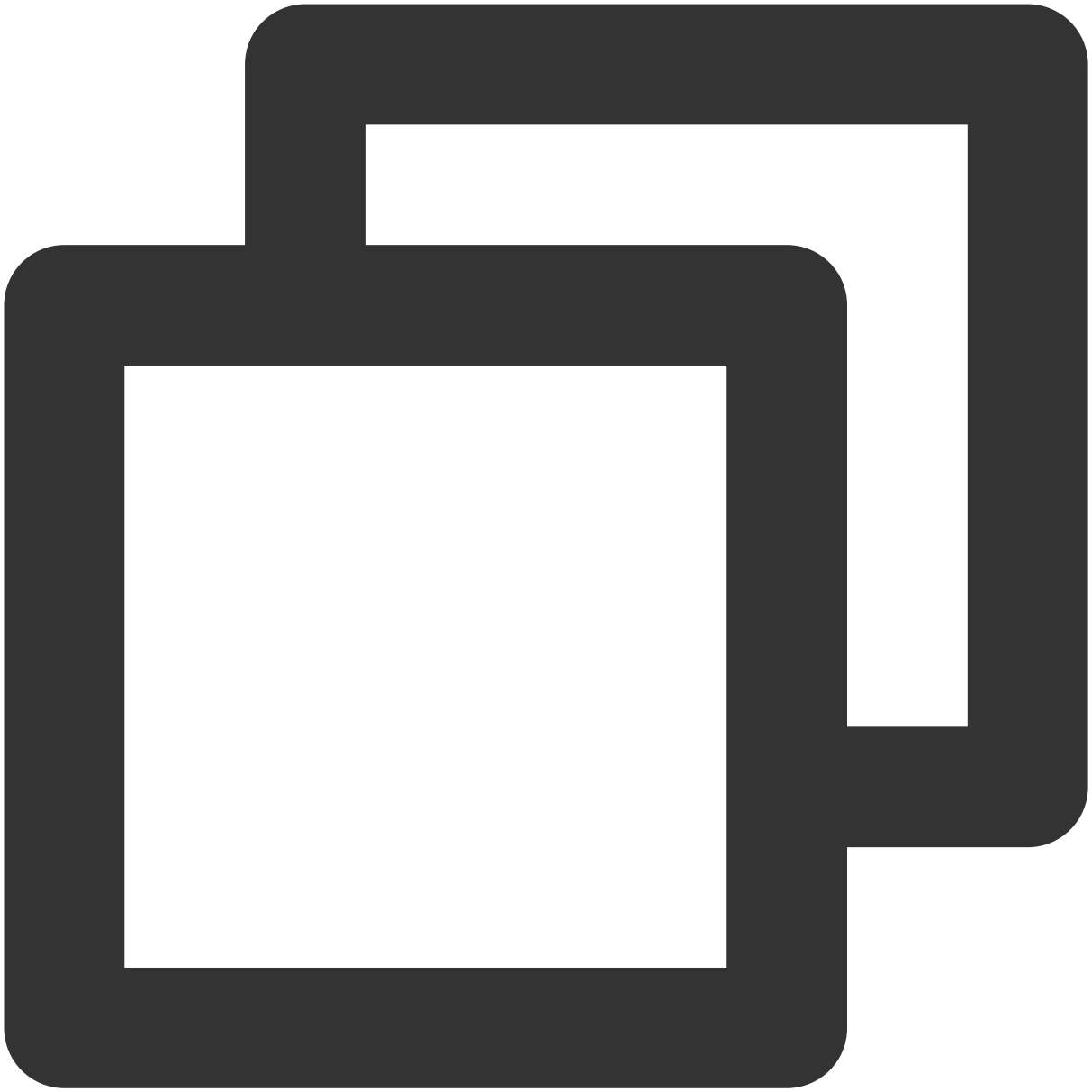
Enumeration	Numerical Value	Description
TXLIVE_WARNING_CAMERA_START_FAILED	-1001	Camera activation failure.
TXLIVE_WARNING_MICROPHONE_START_FAILED	-1002	Failed to activate the microphone.
TXLIVE_WARNING_SCREEN_CAPTURE_START_FAILED	-1003	Failed to initiate screen sharing.
TXLIVE_WARNING_VIRTUAL_CAMERA_START_FAILED	-1004	Failed to open the local media file.
TXLIVE_WARNING_CAMERA_INTERRUPTED	-1005	Camera was interrupted (device was detached or permission was revoked by user).
TXLIVE_WARNING_MICROPHONE_INTERRUPTED	-1006	Microphone was interrupted (device was unplugged or permission was annulled by user).
TXLIVE_WARNING_SCREEN_CAPTURE_INTERRUPTED	-1007	Screen sharing has been interrupted (clicking the stop sharing button in Chrome browser).

Note:

1. Error message returned when attempting to activate the camera, microphone, or screen sharing, refer to getUserMedia exception and getDisplayMedia exception.
2. The extended information returned when the camera, microphone, or screen sharing unexpectedly interrupts, will contain the stream ID of the corresponding stream.

onCaptureFirstAudioFrame

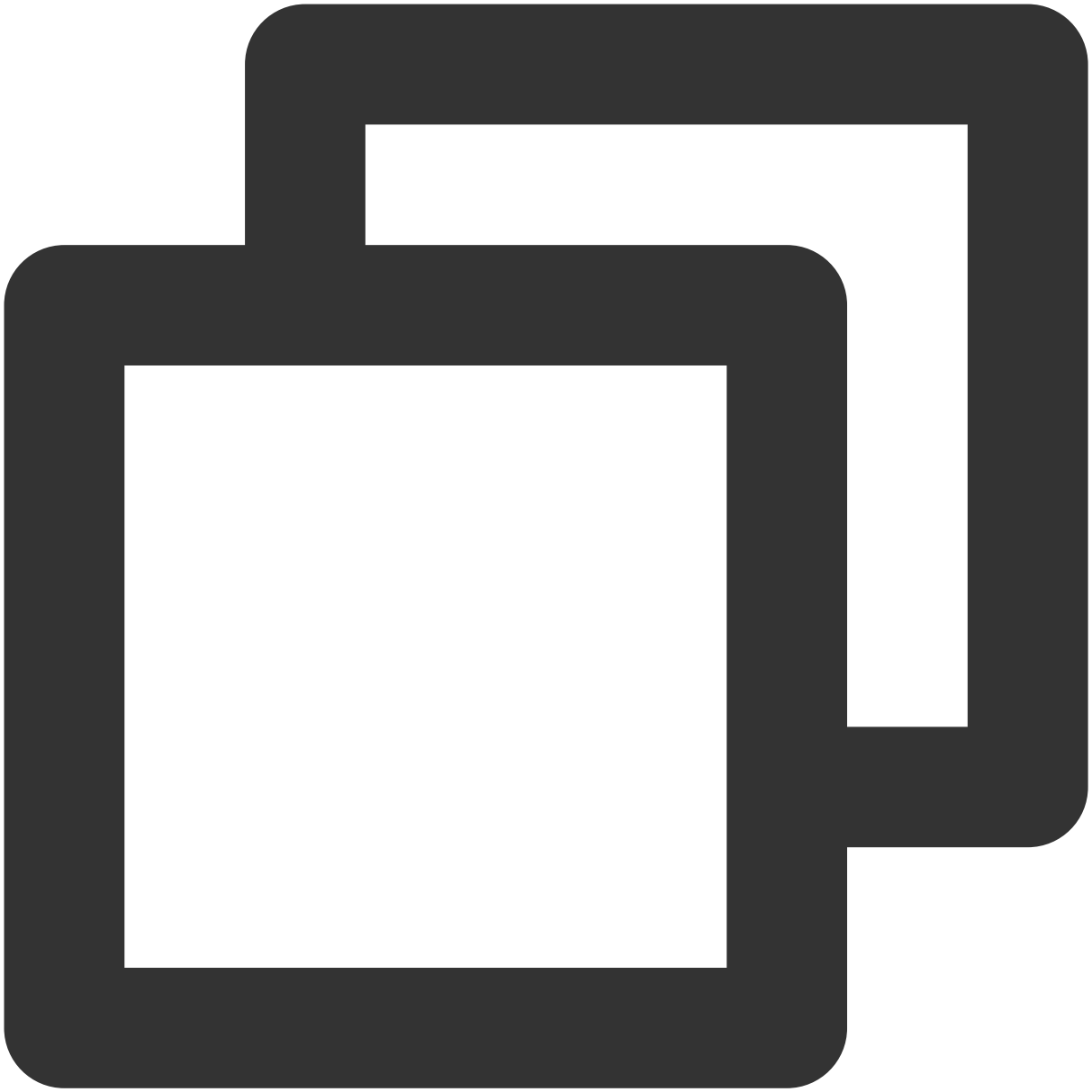
The callback notification when the first frame of audio collection is completed. If the local mixing function is enabled, it will give a callback notification when the final mixed audio stream is generated.



```
onCaptureFirstAudioFrame(): void;
```

onCaptureFirstVideoFrame

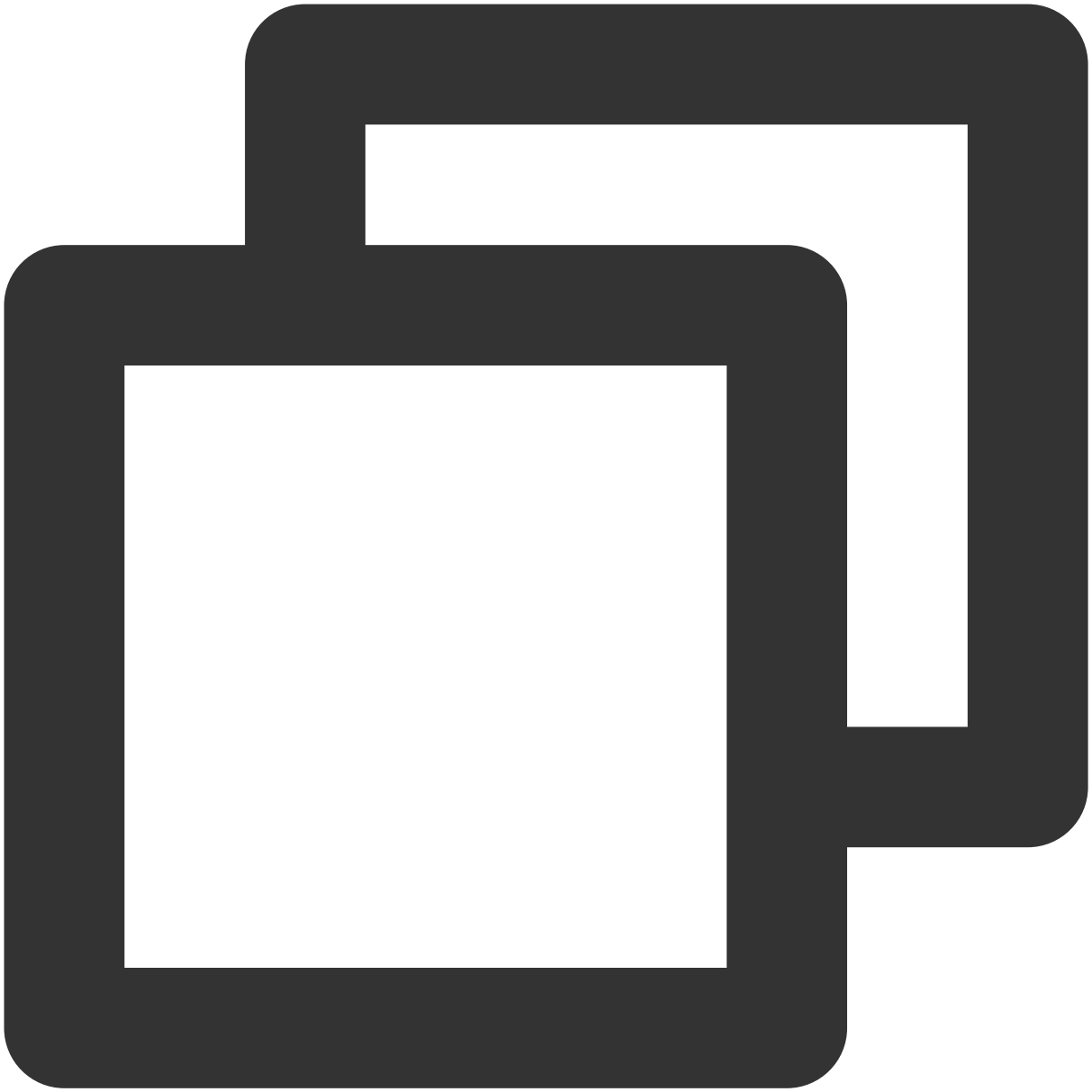
The callback notification when the first frame of video collection is completed. If the local mixing function is enabled, it will call back when the final mixed video stream is generated.



```
onCaptureFirstVideoFrame(): void;
```

onPushStatusUpdate

Live stream connection status callback notification.



```
onPushStatusUpdate(status: number, message: string, extraInfo: object): void;
```

The parameters are described as follows:

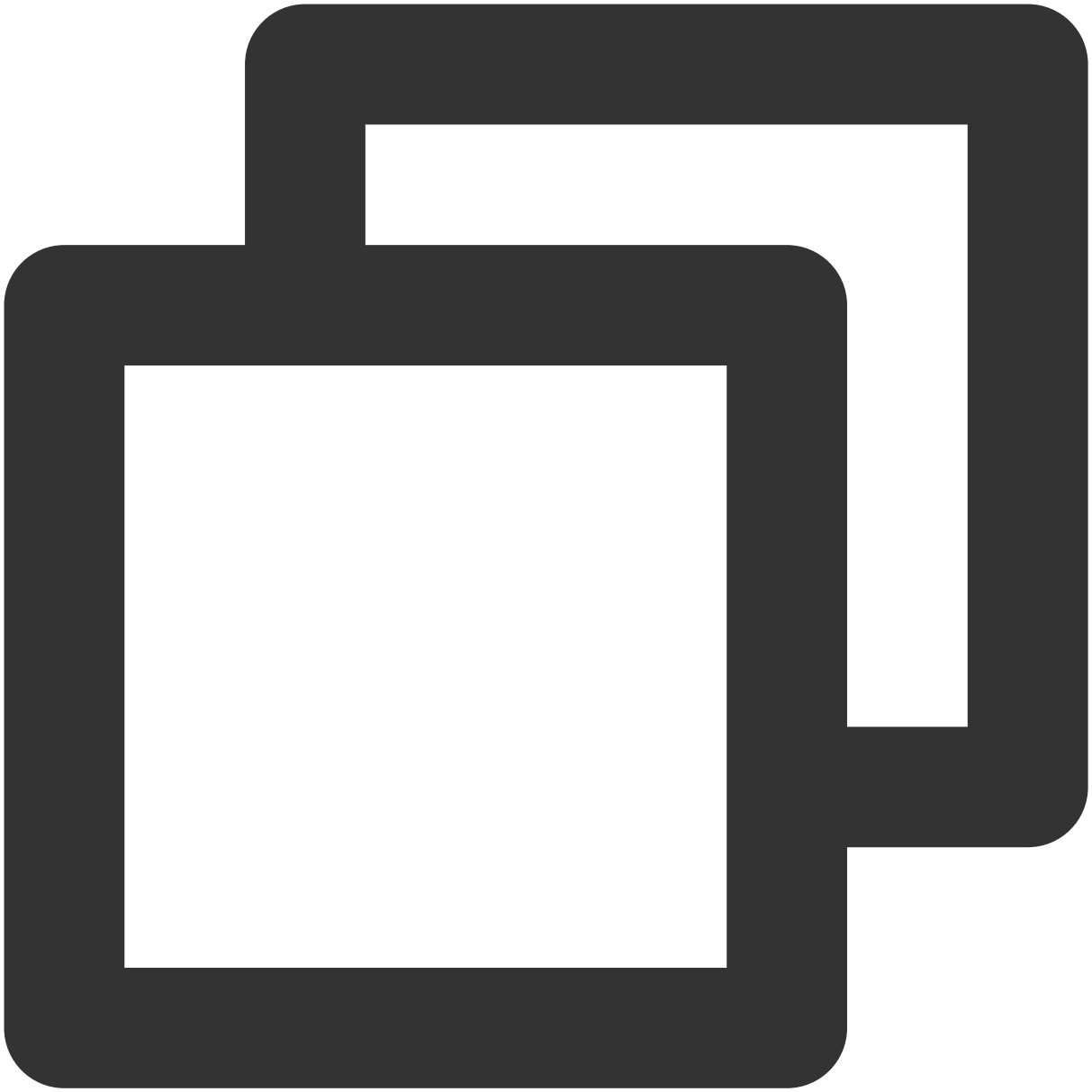
Field	Type	Description
status	number	Connection status code.
message	string	Connection status information.
extraInfo	object	Extension information.

The reference for the connection status code is as follows:

Enumeration	Numerical Value	Description
TXLIVE_PUSH_STATUS_DISCONNECTED	0	Disconnect from server
TXLIVE_PUSH_STATUS_CONNECTING	1	Establishing server connection
TXLIVE_PUSH_STATUS_CONNECTED	2	Server connection established successfully
TXLIVE_PUSH_STATUS_RECONNECTING	3	Reconnecting to the server

onStatisticsUpdate

Callback of the push stream statistics, primarily encompassing WebRTC related statistical data.



```
onStatisticsUpdate(statistics: object): void;
```

The parameters are described as follows:

Field	Type	Description
statistics	object	Push stream statistics.
statistics.timestamp	number	The time for data sampling, counted as milliseconds elapsed since January 1, 1970 (UTC).

statistics.video	object	Data pertinent to the video stream.
statistics.video.bitrate	number	Video bitrate, unit: bit/s.
statistics.video.framesPerSecond	number	Video frame rate.
statistics.video.frameWidth	number	Video width.
statistics.video.frameHeight	number	Video height.
statistics.video.framesEncoded	number	Encoded frame count.
statistics.video.framesSent	number	Transmitted Frame Count.
statistics.video.packetsSent	number	Number of packets sent.
statistics.video.nackCount	number	NACK (Negative Acknowledgement) count.
statistics.video.firCount	number	FIR (Full Intra Request), the number of keyframe retransmission requests.
statistics.video.pliCount	number	PLI(Picture Loss Indication), the number of times video frames are retransmitted due to loss.
statistics.video.frameEncodeAvgTime	number	Average Encoding Time, measured in: ms.
statistics.video.packetSendDelay	number	Average duration of local caching prior to data packet transmission, unit: ms .
statistics.audio	object	Data related to the audio stream.
statistics.audio.bitrate	number	Audio bitrate, expressed in: bit/s .
statistics.audio.packetsSent	number	Number of packets sent.

Note:

1. During the live streaming process, the SDK calculates the WebRTC related data at the frequency of once per second, and then calls this callback interface to return the data.
2. If the returned field data is empty (undefined), it indicates that the current browser cannot obtain the corresponding data. At present, only the Chrome browser can access all the data.

TXMediaDeviceInfo

Device information.

Data Structure:

Field	Type	Description
type	string	Device type, video - camera, audio - microphone.
deviceId	string	Device ID.
deviceName	string	Device Name.

TXMixingConfig

Configuration of mix streaming parameters.

Data Structure:

Field	Type	Description
videoWidth	number	The final width of the mixed video stream.
videoHeight	number	The ultimate height of the mixed video stream.
videoFramerate	number	Frame rate of the video after final mixing.
backgroundColor	number	The background color of the mixed screen, in hexadecimal format, defaults to black, i.e., 0x000000.

TXLayoutConfig

Picture-in-picture layout parameters.

Data Structure:

Field	Type	Description
streamId	string	Stream ID, denotes the stream to be configured.
x	number	Layout of the x-coordinate.
y	number	Layout y-axis.
width	number	Layout Width.
height	number	Layout Height.
zOrder	number	Layout Hierarchy.

Note:

1. With the top left corner as the origin (0, 0), the coordinate attributes of the element describe its center. For instance, for a video stream with a resolution of 100*100, if it appears completely adjacent to the origin in the final generated video stream, then both its x and y coordinates would be 50.
2. The greater the zOrder value, the more prominently the video stream will be displayed, overriding the visuals of other video streams.

TXMirrorConfig

Mirror argument.

Data Structure:

Field	Type	Description
streamId	string	Stream ID, denotes the stream to be configured.
mirrorType	number	Mirror Type: 0 - No mirror effect, 1 - Horizontal mirror, 2 - Vertical mirror, 3 - Horizontal + Vertical mirror.

TXNormalFilterConfig

Parameters for standard filters.

Data Structure:

Field	Type	Description
streamId	string	Stream ID, denotes the stream to be configured.
contrast	number	Contrast, value range [-100, 100], 0 denotes no processing.
brightness	number	Brightness, the value range [-100, 100], 0 indicates no processing.
saturation	number	Saturation, the value ranges from [-100, 100]. 0 indicates no processing.

TXWatermarkConfig

Parameters for Watermark Configuration.

Data Structure:

Field	Type	Description
image	HTMLImageElement	Watermark image object.
x	number	Watermark's x-coordinate.
y	number	Watermark Y coordinate.
width	number	Watermark width.
height	number	Watermark height.
zOrder	number	Watermark hierarchy.

Note:

1. With the upper left corner being the origin point (0, 0), the coordinate attributes of the element describe the center of the element. For instance, a watermark image with a resolution of 100*100, aligned closely with the origin, appearing perfectly within the resultant video stream, has its x and y coordinates respectively situated at 50.
2. The larger the zOrder value, the more prominently the watermark image will appear, overlaying other video streams.

TXTextConfig

Text Configuration Parameters.

Data Structure:

Field	Type	Description
text	string	Text content cannot be an empty string.
style	object	Refer to the corresponding CSS style settings for text styles.
style.font	string	Font Name.
style.font_size	number	Font size.
style.font_color	string	Font color represented in hexadecimal, such as #000000.
style.font_alpha	number	Font transparency, range [0, 100], default 100 (opaque).
style.bold	number	Text bolding, 0 - No bold, 1 - Bold, default 0 .
style.italic	number	Font slant, 0 - normal, 1 - italic, default 0.
style.shadow_color	string	Text shadow color, represented in hexadecimal, for

		instance #000000 .
style.shadow_alpha	number	Text shadow opacity, ranging from [0, 100], effective when shadow_color is present, default is 100 (opaque).
style.stroke_color	string	Text outline color, represented in hexadecimal, such as #000000 .
style.stroke_thickness	number	Stroke thickness of the text, default value is 0, that is, no stroke.
style.background_color	string	Background color, represented in hexadecimal form, such as #000000 .
style.background_alpha	number	Background transparency, ranging [0, 100]. It is effective when background_color exists, default value is 100 (Opaque).
x	number	Text x coordinate.
y	number	Text Y-Axes.
zOrder	number	Textual Hierarchy.

Note:

1. The origin (0,0) is the top left corner, the coordinate properties of the element describe the center point of the element. Assuming the final width of the text is 100px, the height is 50px, if the text is to snugly fit the origin appearing in the finally generated video stream, its x-coordinate is 50, y-coordinate is 25.
2. The greater the zOrder value, the higher the text appears, overlaying other video stream images.

2. Playback

Last updated：2022-11-21 16:56:28

Prerequisites

- You have activated the CSS service.
- Select Domain Management, click **Add Domain**, and add a playback domain name as instructed in Adding Your Own Domain.
- In the CSS console, generate a playback address in **CSS Toolkit** > **Address Generator** as instructed in Address Generator. Then, implement live playback in your own business based on your business scenarios as follows:

Integration into an application

Download and integrate the MLVB SDK as instructed in the iOS and Android integration guides.

Note：
Enable stream pull and playback. Below is the configuration for iOS and Android:

- iOS

```
V2TXLivePlayer *_txLivePlayer = [[V2TXLivePlayer alloc] init];
>
```

- Android

```
V2TXLivePlayer mLivePlayer = new V2TXLivePlayerImpl(mContext);
```

More

- Using the MLVB SDK will incur fees. For billing details, see Billing Overview.

3. Advanced Features Overview

Last updated：2024-07-24 10:32:13

This document describes how to import advanced live streaming features into your program.

AV1 Video Playback

AV1 is an open-source and royalty-free video compression format. Compared with its predecessor H.265 (HEVC) encoding, it can further reduce the bitrate by over 30% while maintaining the same level of image quality. This means that it can deliver a higher image quality at the same bandwidth. Currently, CSS supports AV1 encoding. However, to play back AV1 videos, your player must support decoding the AV1 format.
You can implement AV1 decoding in your own player as follows:

Container format and distribution protocol

Tencent Cloud has implemented a proprietary extension to AV1 in FLV in T-FFmpeg. To modify your player, you can extend the code based on the patch files of T-FFmpeg. For more information, see tencentyun/AV1.

Decoding

- **Hardware decoding**
  On PC, almost all new models of AMD, Intel, and Nvidia GPUs support AV1 hardware decoding.
  Device models supporting AV1 hardware decoding are as listed below:

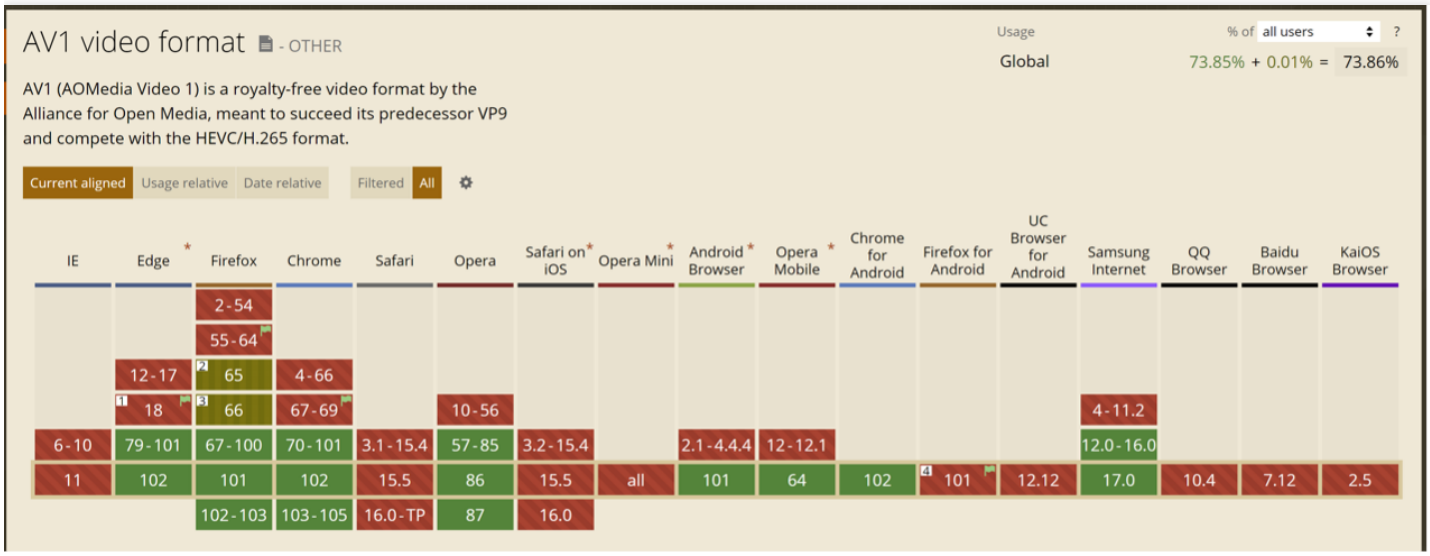
Type	Brand	Processor
Mobile phone	Realme X7 Pro	Dimensity 1000+
	OPPO Reno5 Pro	Dimensity 1000+
	HONOR V40	Dimensity 1000+
	Redmi K30 Ultra	Dimensity 1000+
	Vivo iQOO Z1	Dimensity 1000+
	Redmi Note 10 Pro	Dimensity 1000+
	Vivo S9	Dimensity 1100

Type	Brand	Processor
	Realme Q3 Pro	Dimensity 1100
	Vivo S10	Dimensity 1100
	Vivo S12	Dimensity 1100
	Vivo S12 Pro	Dimensity 1200
	OPPO Reno6 Pro	Dimensity 1200
	OPPO Reno7 Pro	Dimensity 1200
	Redmi K40 Pro	Dimensity 1200
	Realme GT NEO	Dimensity 1200
	HONOR X20	Dimensity 1200
	OnePlus Nord 2	Dimensity 1200
	Realme GT NEO 2	Dimensity 1200
	OPPO K9 Pro	Dimensity 1200
	OPPO Find X5 Pro Dimensity Edition	Dimensity 9000
	Redmi K50	Dimensity 9000
	Galaxy S21 (Exynos Edition)	Exynos 2100
	Galaxy S22 (Exynos Edition)	Exynos 2200
TV	Samsung Q950TS QLED 8K TV	-

- **Software decoding**
  - av1d (Tencent Cloud's optimized AV1 decoder, which outperforms dav1d and can provide closed-source libraries. You can integrate it as instructed in av1d Integration Guide. T-FFmpeg provides FFmpeg patches to be integrated and av1d libraries.)
  - dav1d
  - libgav1
  - Android 10.0 integrates an AV1 decoder.
  - The Chrome family supports AV1 decoding.

Support by browsers

The Chrome family supports AV1 decoding while browsers on iOS don't.



Note：
The above information was collected from this website in July 2022. To view the latest information, please visit the website.

TMIO SDK

The TMIO SDK customizes, encapsulates, and optimizes streaming media protocols such as SRT and QUIC. It safeguards upstreaming and implements low-latency transfer with a high packet loss prevention rate, multi-linkage transfer optimization, and the retransmission timeout (RTO) mechanism. It promises wide application in scenarios that require a high data source stability and long-range transfer.

Feature description

- The TMIO SDK is suitable for long-range transfer and the radio and TV industry.
- The TMIO SDK supports mainstream platforms, including Android, iOS, Windows, macOS, and Linux.

Integration method

You can integrate the SDK as instructed in Integration Steps.

libLebConnection SDK

The libLebConnection SDK provides upgraded transfer capabilities based on the native WebRTC. It allows you to connect your existing player to LEB with just a few simple modifications. Based on LVB-compatible stream push and cloud-based media processing capabilities, it can implement live streaming at a low latency even under high concurrency, so as to help you smoothly migrate from standard LVB streaming to low-latency LEB streaming. For large rooms in modern real-time communication (RTC) scenarios, it can also help you quickly implement relayed live streaming at low costs and low latency.

Feature description

- The libLebConnection SDK can pull audio/video streams at a low latency even under poor network conditions.
- It can play back H.264. H.265, and AV1 videos with B frames and output them as raw video frame data such as Annex B and OBU files for H.264/H.265 and AV1 input videos respectively.
- It can play back AAC and Opus audios and output them as raw audio frame data.
- It supports Android, iOS, Windows, Linux, and macOS.

Integration method

You can integrate the libLebConnection SDK as instructed in Integrating libLebConnection SDK into a Player.

Beauty filters and special effects

To integrate beauty filters and special effects and import beauty filters, image filters, and stickers during live streaming, you can integrate the Tencent Effect SDK.

Integration into an application

Download the Tencent Effect SDK here and integrate it as instructed in the iOS and Android integration guides.

More

Using the Tencent Effect SDK will incur fees. For billing details, see Pricing Overview.

TMIO SDK

Last updated：2023-11-24 15:01:14

Overview

As more and more streaming protocols emerge, it has become a challenge to build an application that supports all protocols. For this, we have developed the Tencent Media IO (TMIO) SDK, which is optimized for mainstream protocols on the market, to help you build stable and highly available media applications with ease.
The TMIO SDK supports mainstream protocols including SRT and QUIC, as well as Tencent's proprietary protocol, Elastic Transmission Control (ETC). More will be added in the future.

Strengths

Multi-platform support: The SDK comes in editions for Android, iOS, Linux, macOS, and Windows.
Different integration methods:
You can use the non-intrusive proxy mode to use the SDK, with no code changes required.
The simple API design also allows you to quickly integrate the SDK into your application, replacing your existing protocols.
Simple API design with high compatibility and flexibility:
Easy-to-use APIs are offered.
You can choose different modes and policies based on your business scenario.
The SDK can be extended to support other protocols.
Multi-protocol support and optimization:
The SDK supports emerging streaming protocols including SRT and QUIC, as well as Tencent's proprietary ETC protocol.
It can meet many requirements in different business scenarios.
It delivers low-latency, secure, and reliable transfer based on UDP.
It supports multi-connection acceleration, which guarantees transfer stability and smoothness.

Test (TMIO SRT)

The TMIO SDK supports the SRT protocol, which can improve upstream stability and downstream smoothness under poor network conditions or in long-distance transfer scenarios.
In the test below, with RTMP streaming, playback begins to stutter when packet loss reaches 10%, but for SRT streaming, stability and low latency are guaranteed even with a packet loss of 30%.

Features

SRT-based streaming media transfer

High tolerance for random packet loss

A retransmission mechanism based on ARQ and a timeout policy

Low-latency, secure, and reliable transfer based on UDT

Multi-connection transfer and the aggregation transfer mode:

You can configure multiple connections to transfer data. For example, mobile devices can transfer data over both Wi-Fi and the 4G/5G data network, so even if one of the connections is disconnected, data transfer will not be affected. This improves transfer reliability.

Transfer Mode	Description
Broadcasting mode	You can send redundant data over multiple internet connections to ensure data integrity and transfer reliability.
Primary/Backup mode	In this mode, only one connection is active at a time. The connection is selected in real time based on stability and reliability. This also reduces bandwidth usage because no redundant data is sent.
Aggregation mode	In scenarios requiring a high bitrate and bandwidth, the bandwidth of a single internet connection may be unable to meet the requirements. This mode can split data, send them over multiple connections, and then combine them at the receiver end.

QUIC-based media transfer

Adaptive congestion control algorithm

Smooth network switch

Support for the next-generation HTTP/3

Less redundant data sent when bandwidth is limited or network is unstable

Proprietary streaming protocol ETC

Innovative, lightweight, and cross-platform

Support for IoT devices (peer-to-peer communication)

Quick startup, low latency, and efficient bandwidth utilization

Quick and accurate detection of internet connection changes to ensure that the optimal transfer policy is always used

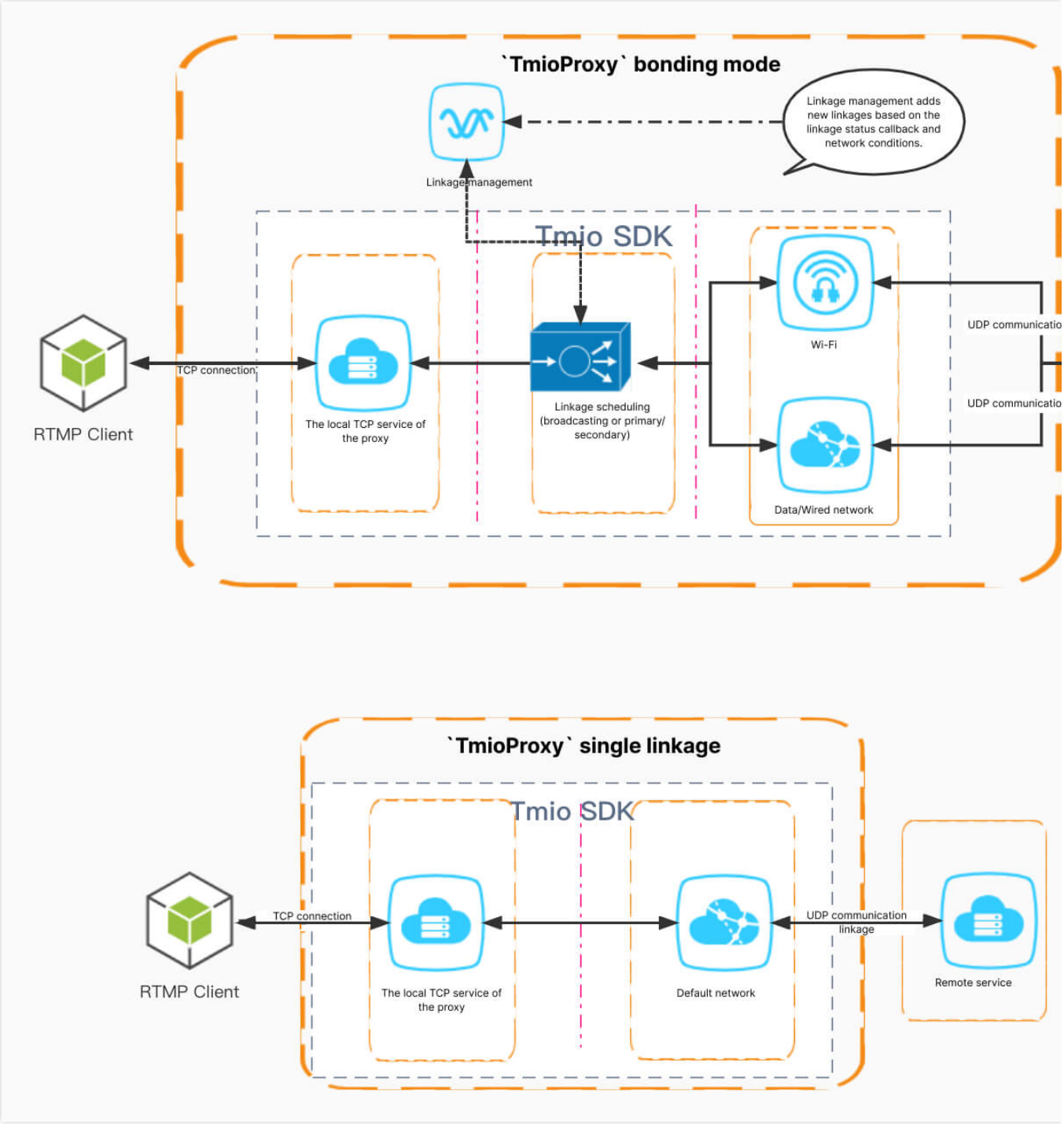
Fair and stable usage of bandwidth when used together with mainstream streaming protocols

Integration Directions

The directions below use the RTMP over SRT protocol as an example.

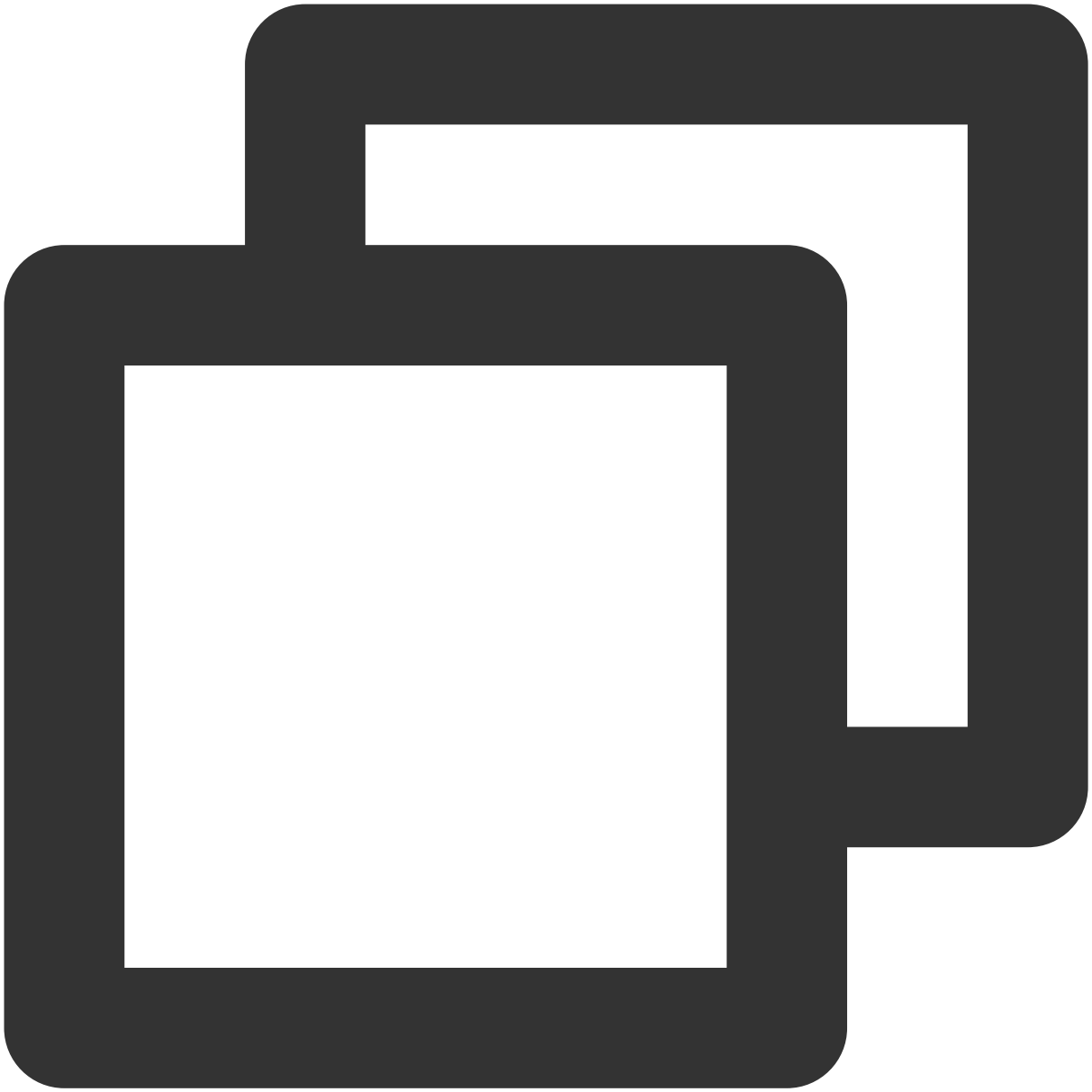
Using the proxy mode

Workflow



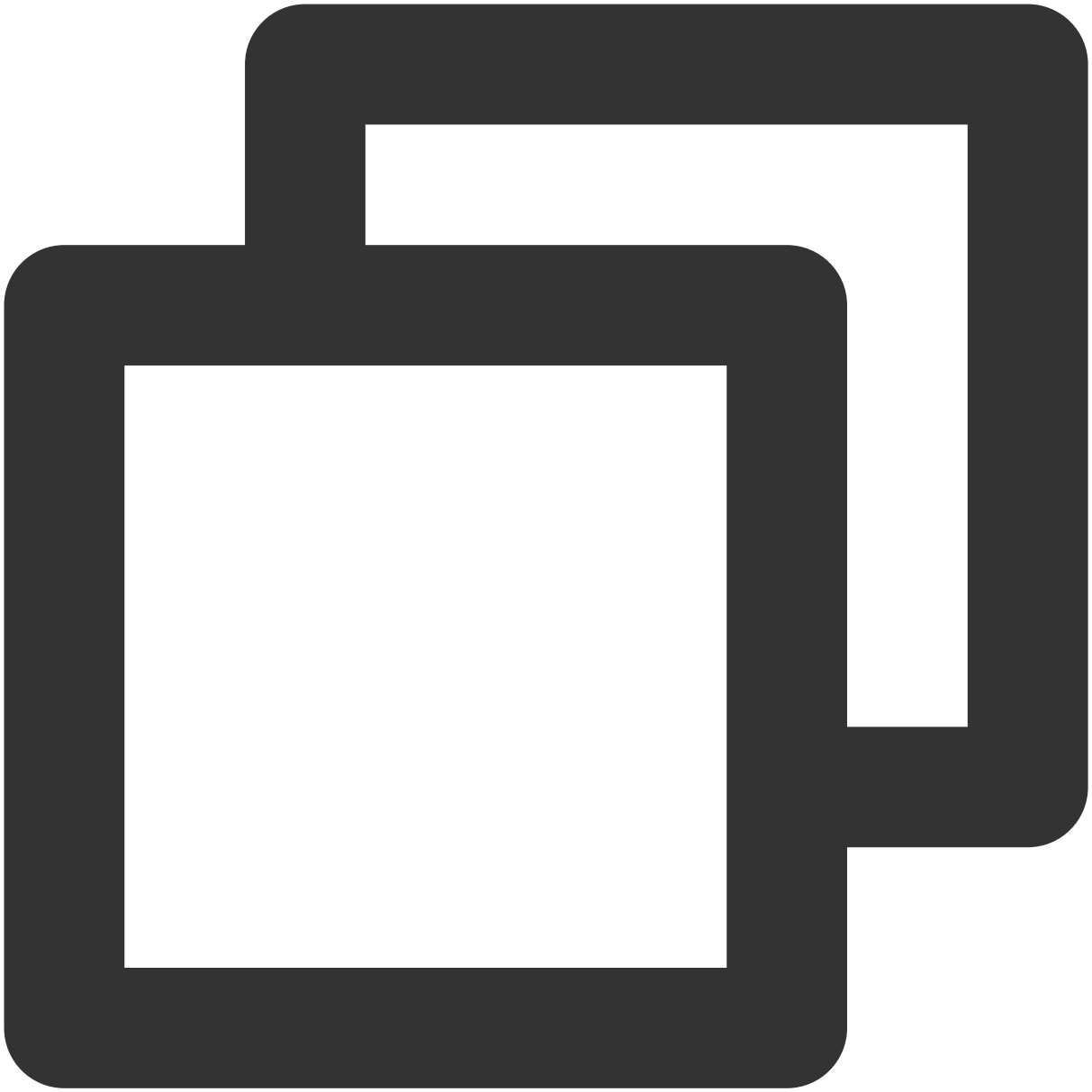
Directions

1. **Create a `TmioProxy` instance**:



```
std::unique_ptr<tmio::TmioProxy> proxy_ = tmio::TmioProxy::createUnique();
```

2. **Set the listener**:



```
void setListener(TmioProxyListener *listener);
```

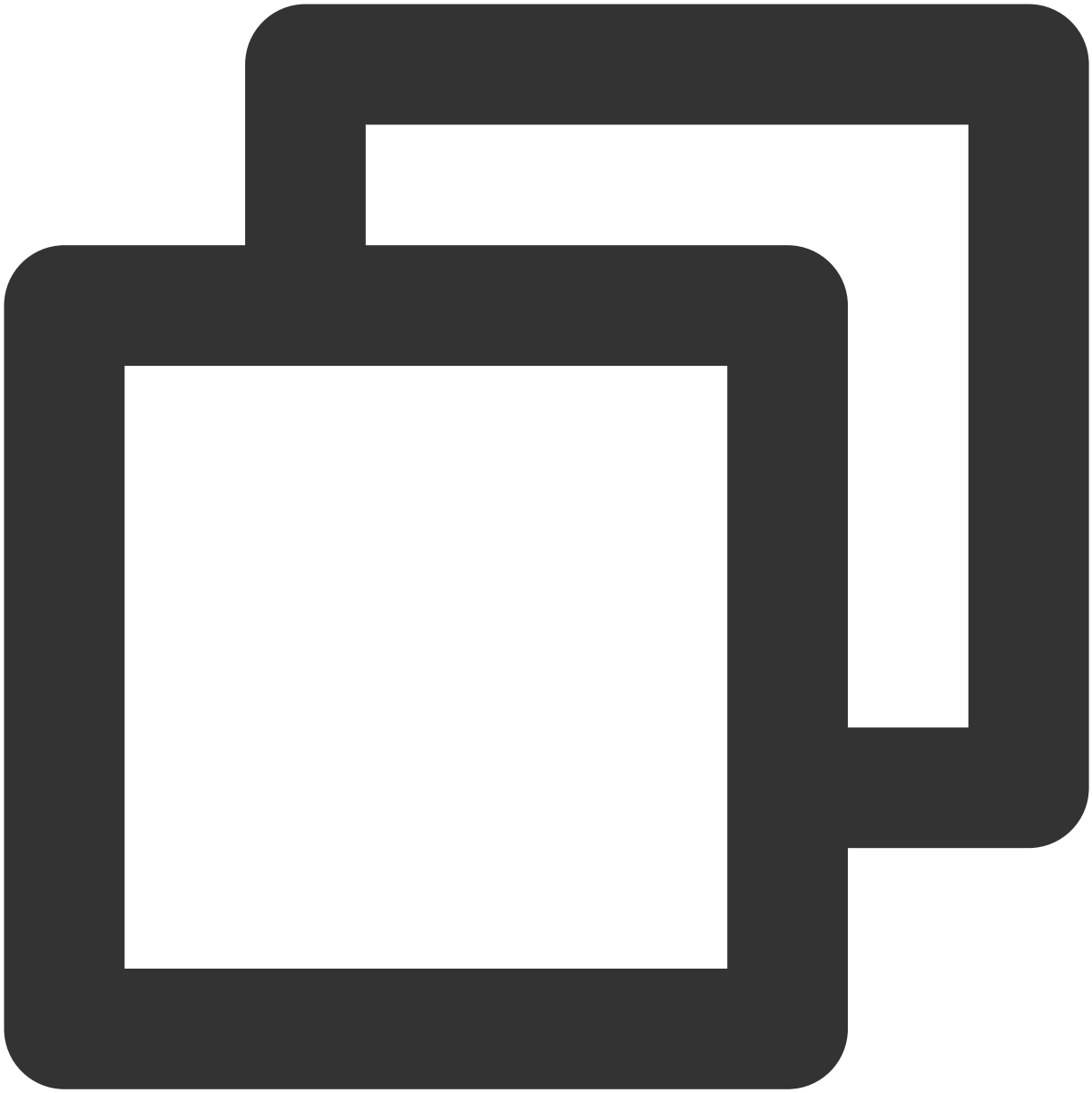
Below are the callbacks of `TmioProxyListener` :

TMIO configuration callback

TMIO proxy startup callback

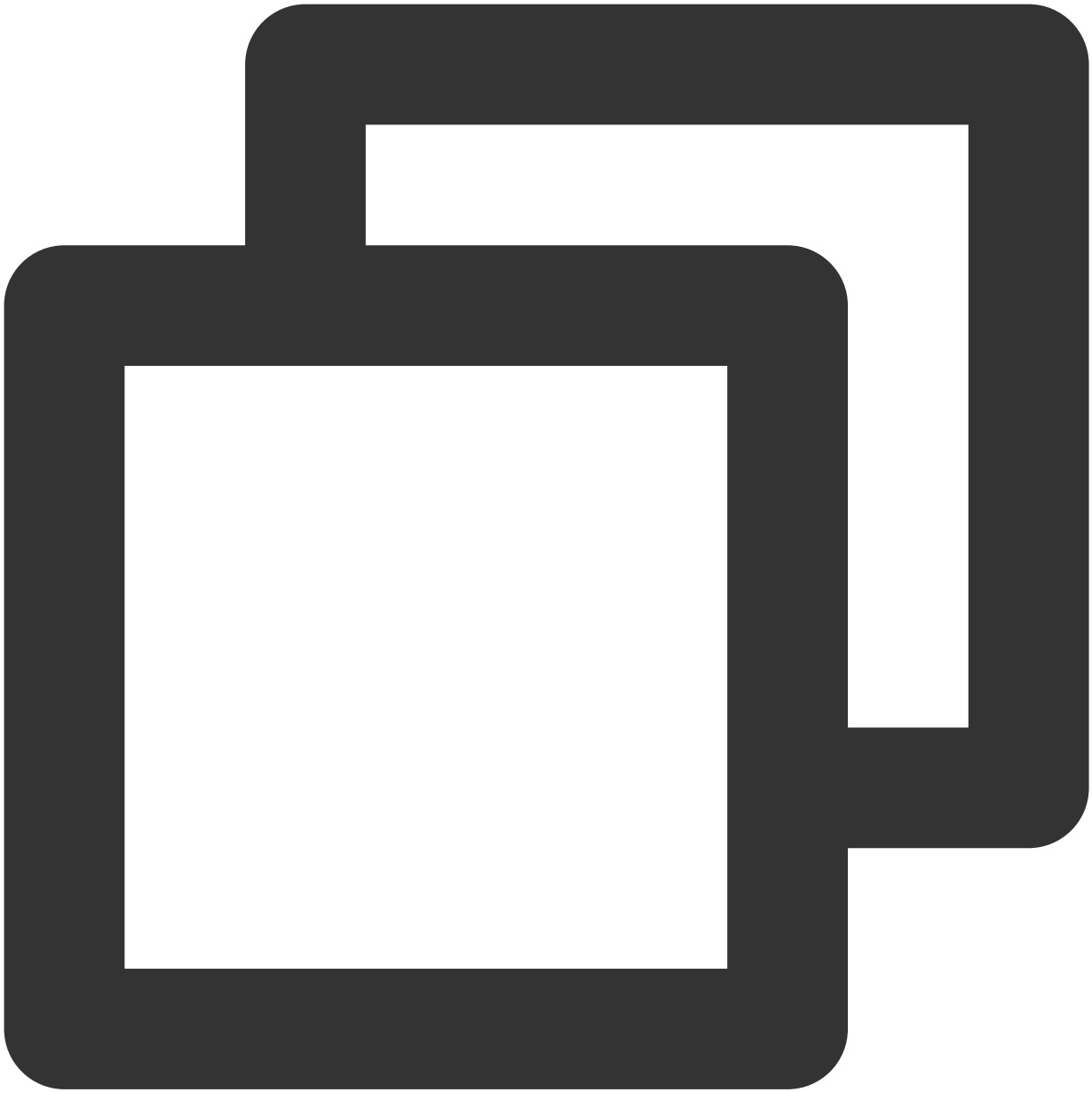
Error callback

You can configure `Tmio` parameters in this callback. **For simple configuration, you can use the auxiliary methods provided in `tmio-preset.h` .**



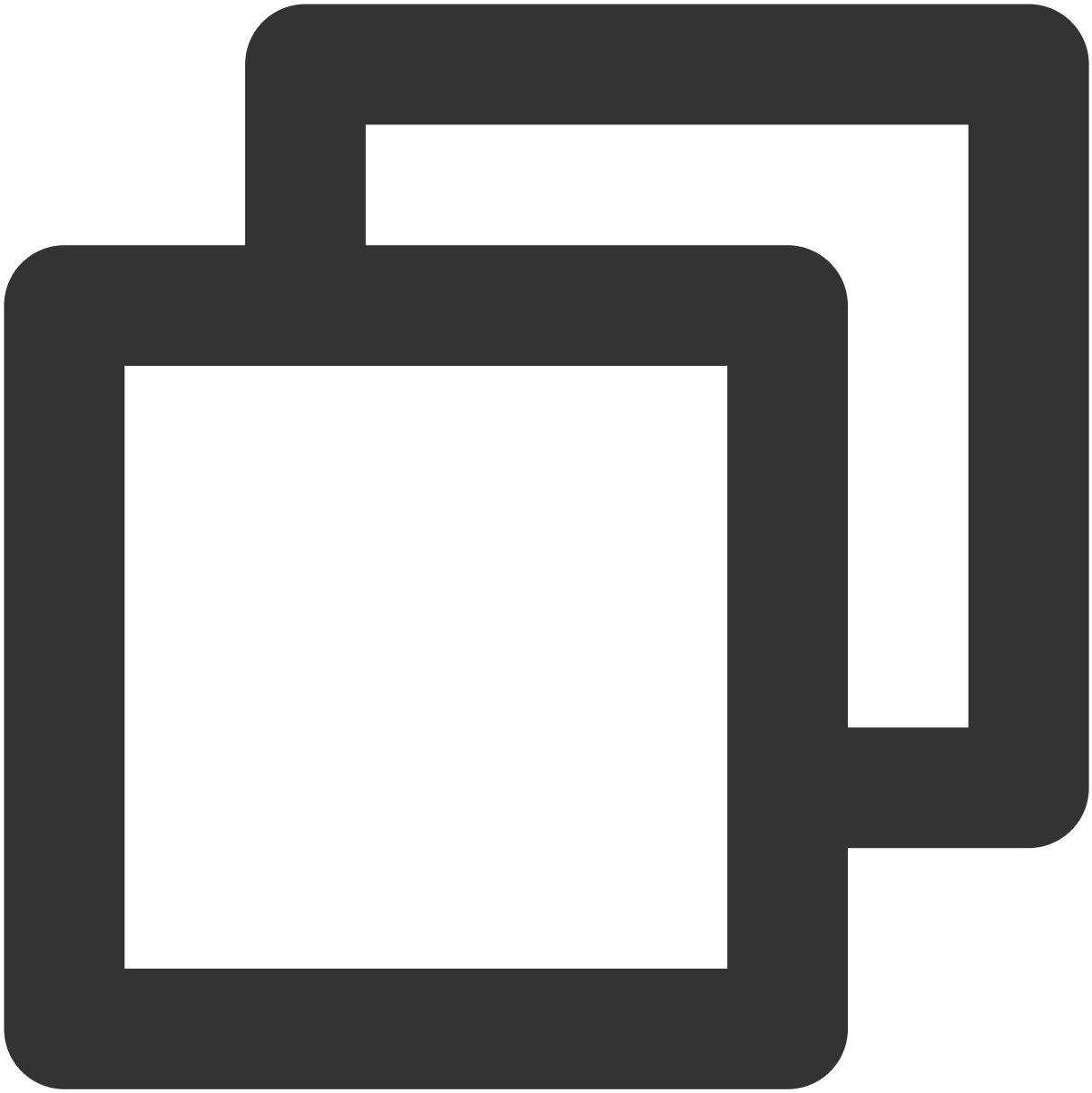
```
/*
void onTmioConfig(Tmio *tmio);
*/
void onTmioConfig(tmio::Tmio *tmio) override {
        auto protocol = tmio->getProtocol();
        if (protocol == tmio::Protocol::SRT) {
                tmio::SrtPreset::rtmp(tmio);
        } else if (protocol == tmio::Protocol::RIST) {
                tmio->setIntOption(tmio::base_options::RECV_SEND_FLAGS,
                                                        tmio::base_options::FLAG_SEND)
        }
```

```
}
```



```
/*
void onStart(const char *local_addr, uint16_t local_port);
*/
void onStart(const char *addr, uint16_t port) override {
        LOGFI("ip %s, port %" PRIu16, addr, port);
}
```

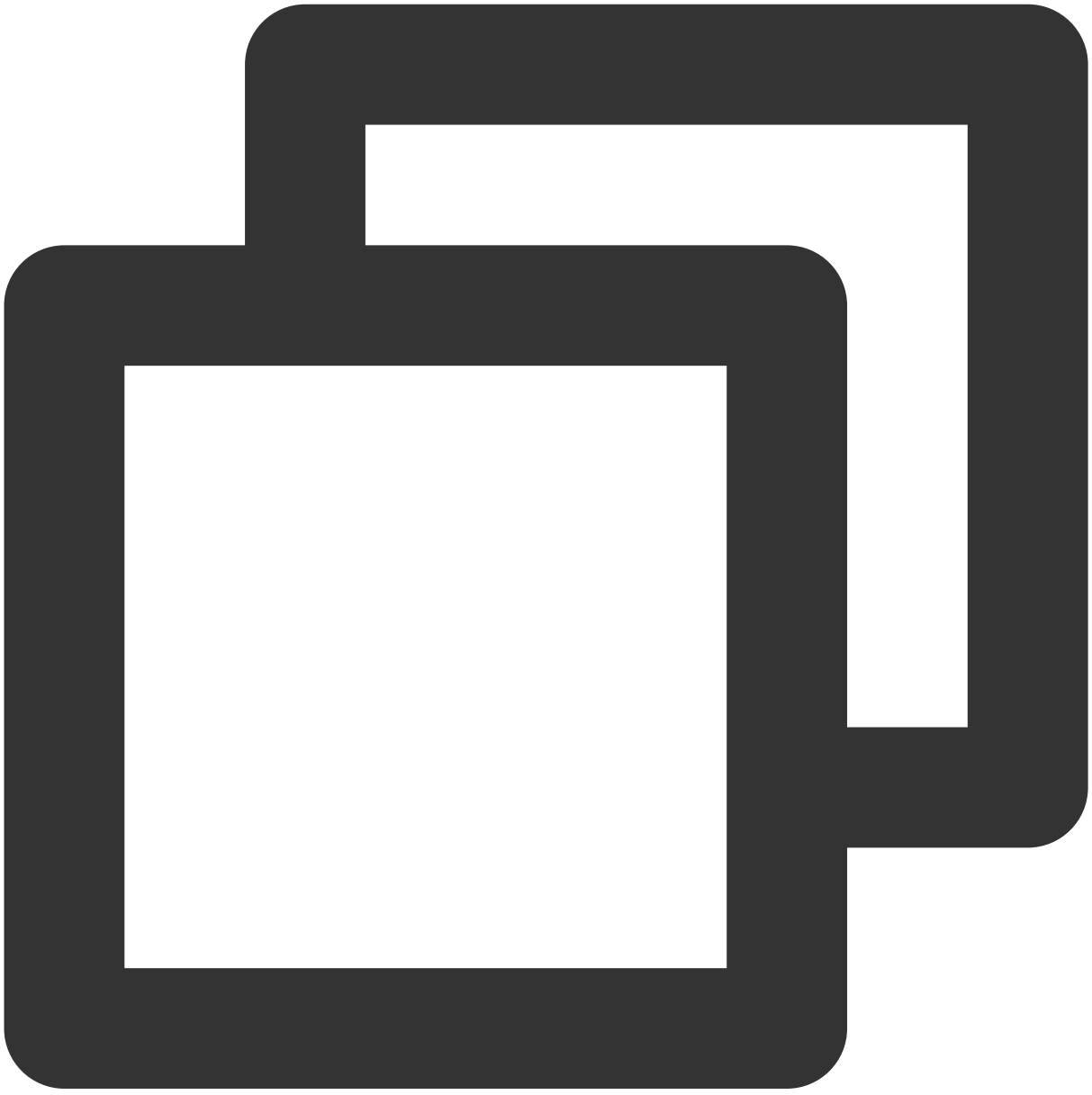
This callback indicates that the remote server is connected successfully, and the local TCP port is bound successfully. You can start pushing streams.



```
/*
void onError(ErrorType type, const std::error_code &err);
*/
void onError(tmio::TmioProxyListener::ErrorType type,
                        const std::error_code &err) override {
        LOGFE("error type %s, %s, %d", tmio::TmioProxyListener::errorType(type),
                    err.message().c_str(), err.value());
}
```

You can use `ErrorType` to determine whether an error is a local or remote I/O error. A local I/O error is usually because RTMP streaming is stopped by the streamer. Therefore, if streaming has ended, you can ignore such errors. However, a remote I/O error usually needs to be handled.

3. **Start the proxy**:



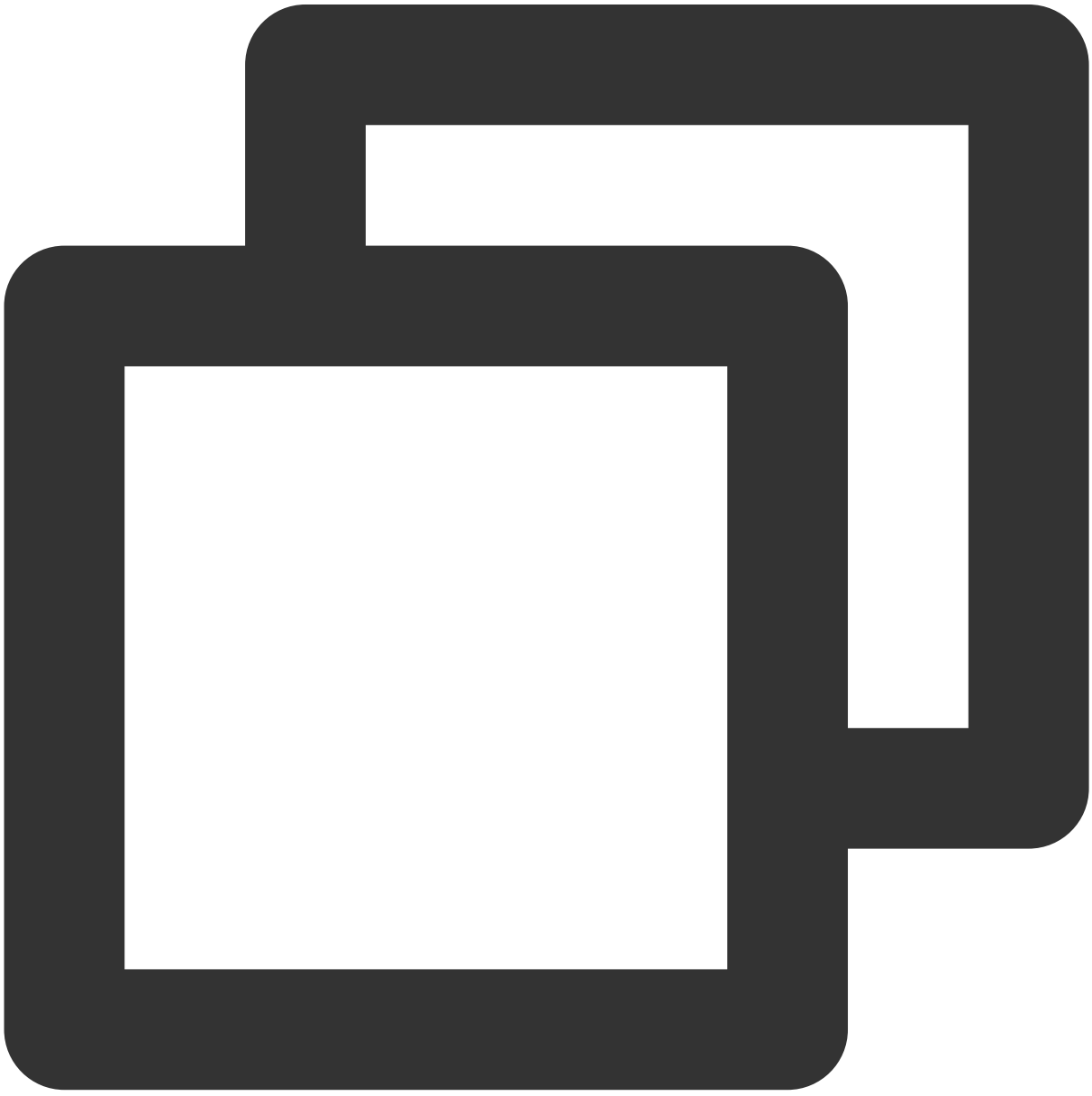
```
std::error_code start(const std::string &local_url, const std::string &remote_url,
```

API parameters

Parameter	Description

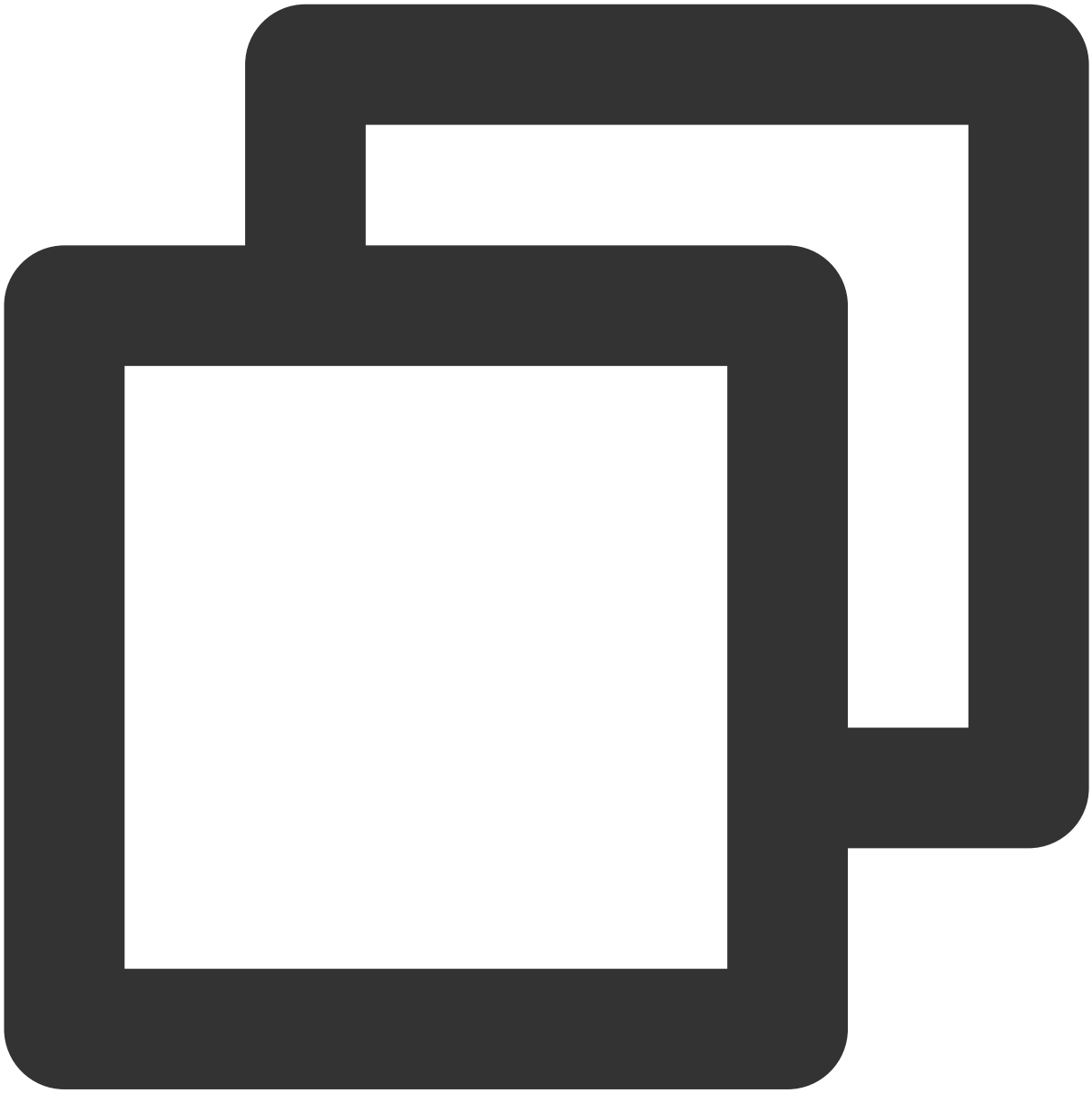
local_url	It supports only the TCP scheme in the format of tcp://${ip}:${port}. `port` can be `0`, which indicates to bind a random port. After the binding succeeds, the bound port will be returned to the application through the `onStart()` callback. Using port `0` can avoid binding failures due to issues such as port occupancy and no permissions.
remote_url	The remote server URL.
config	The configuration parameter. This parameter is valid if SRT connection bonding or QUIC H3 is enabled. For details, see the definition of the `SrtFeatureConfig` structure in tmio.h.

The sample code for a single connection:



```
proxy_->start(local_url, remote_url, NULL);
```

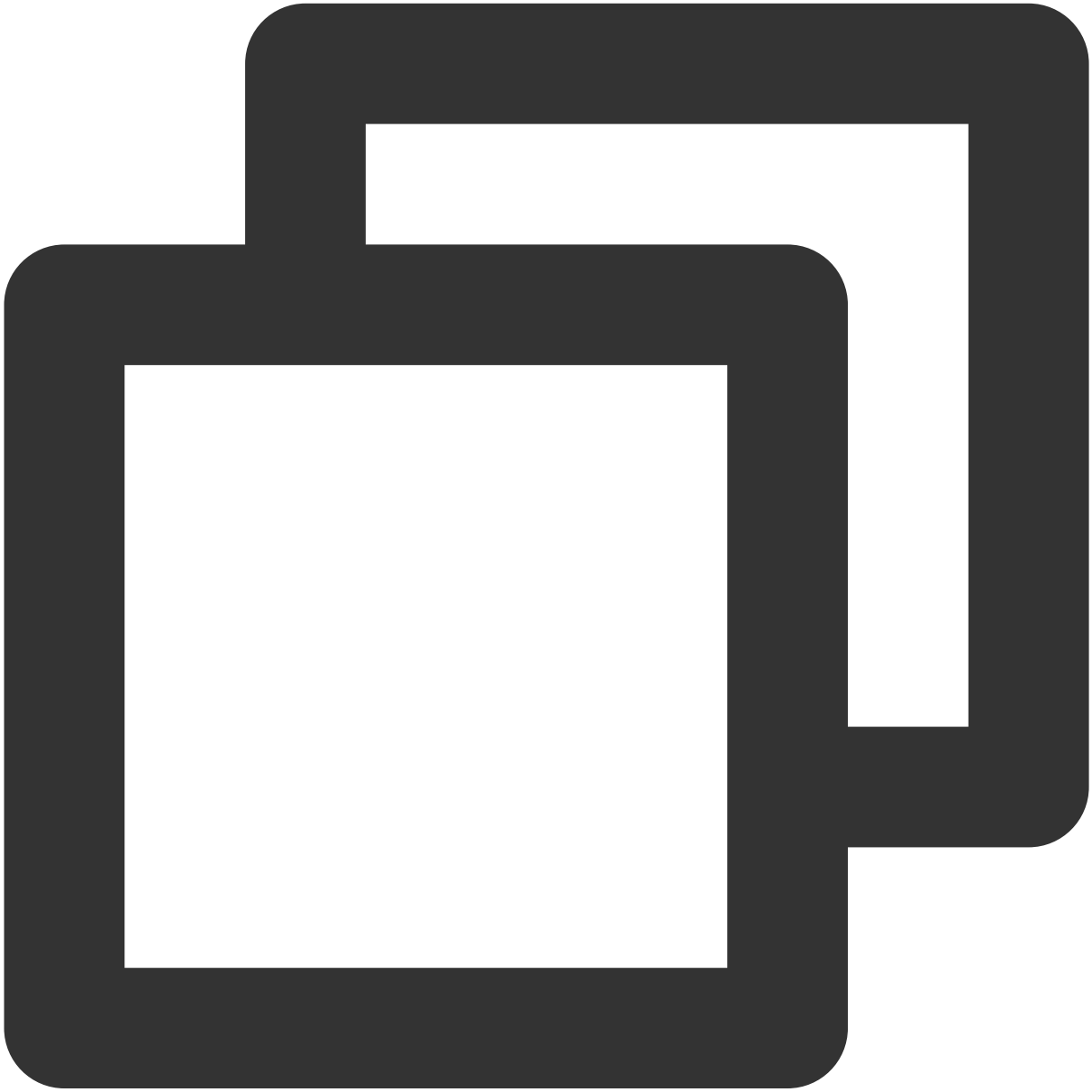
The sample code for multi-connection bonding:



```
tmio::TmioFeatureConfig option;
option_.protocol = tmio::Protocol::SRT;
option_.trans_mode = static_cast<int>(tmio::SrtTransMode::SRT_TRANS_BACKUP);
/*-------------------------------------------------------------*/
{
// You can add multiple connections as needed.
option_.addAvailableNet(net_name, local_addr, remote_url, 0, weight, -1);
```

```
}
/*-------------------------------------------------------------*/

 proxy_->start(local_url, remote_url, &option_);
```

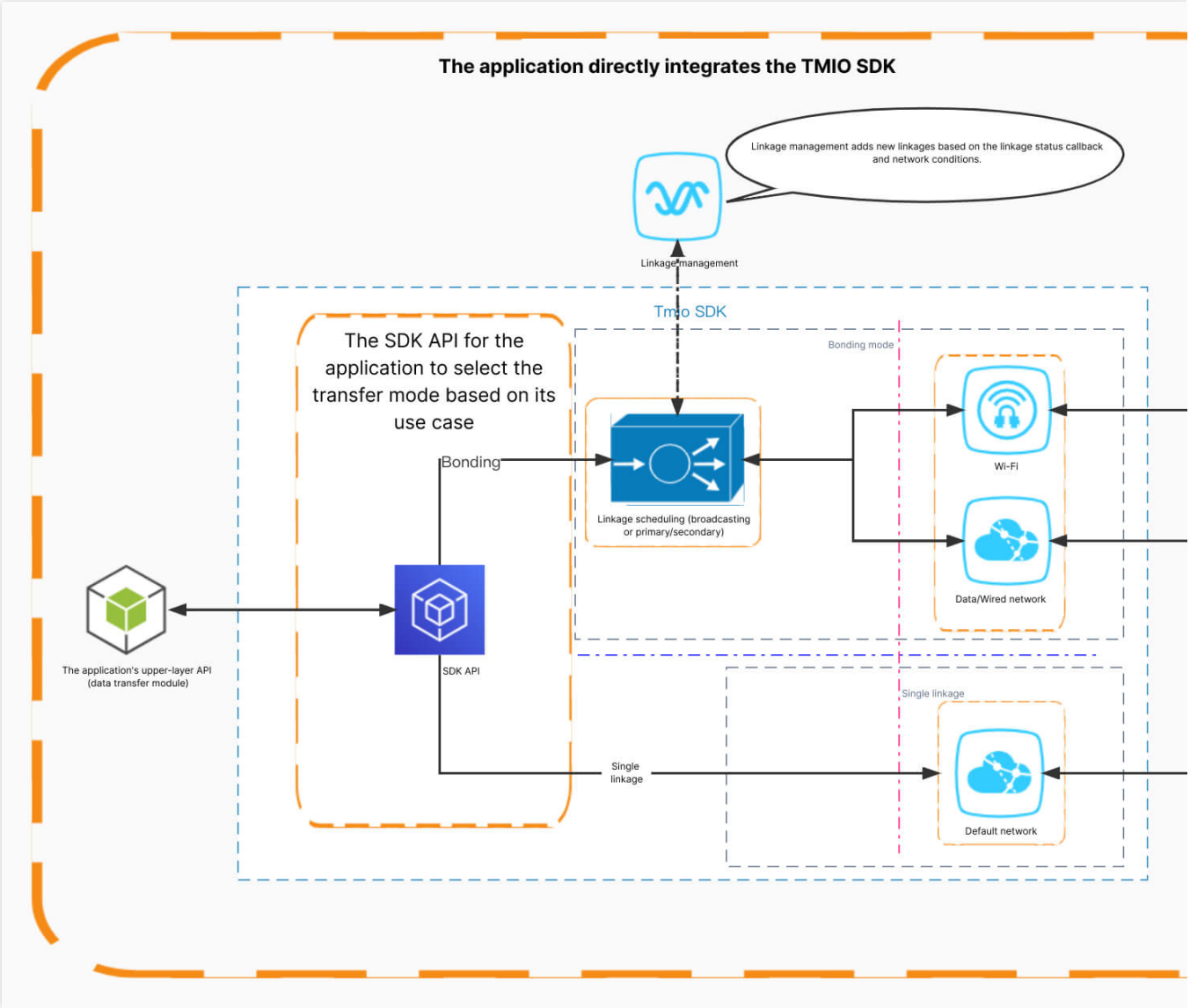
4. **Stop the proxy**:



```
/*
void stop();
*/
proxy_.stop();
```

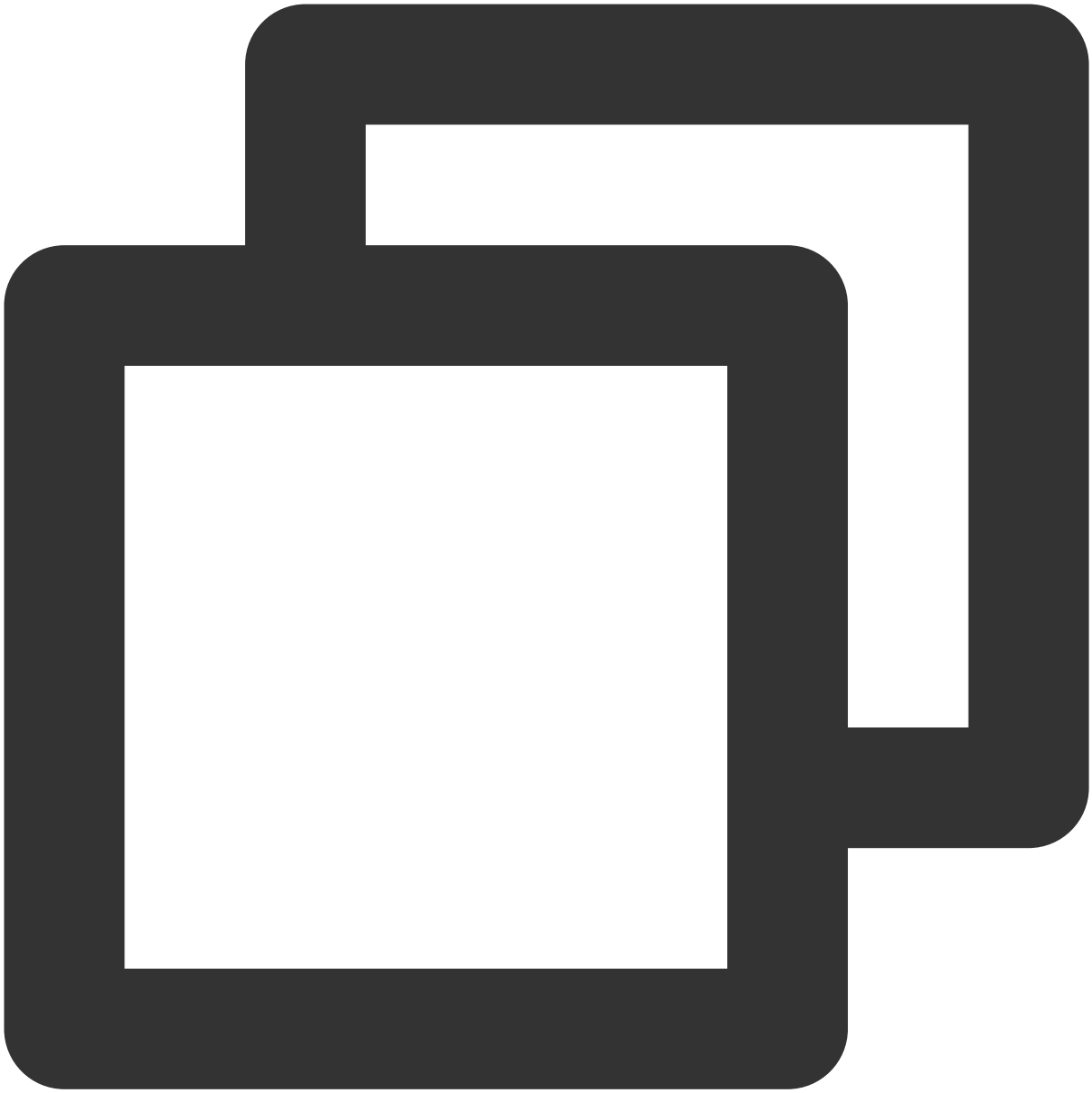
Integrating the TMIO SDK into your application

Workflow



Directions

1. **Create a `Tmio` instance and configure the parameters**:



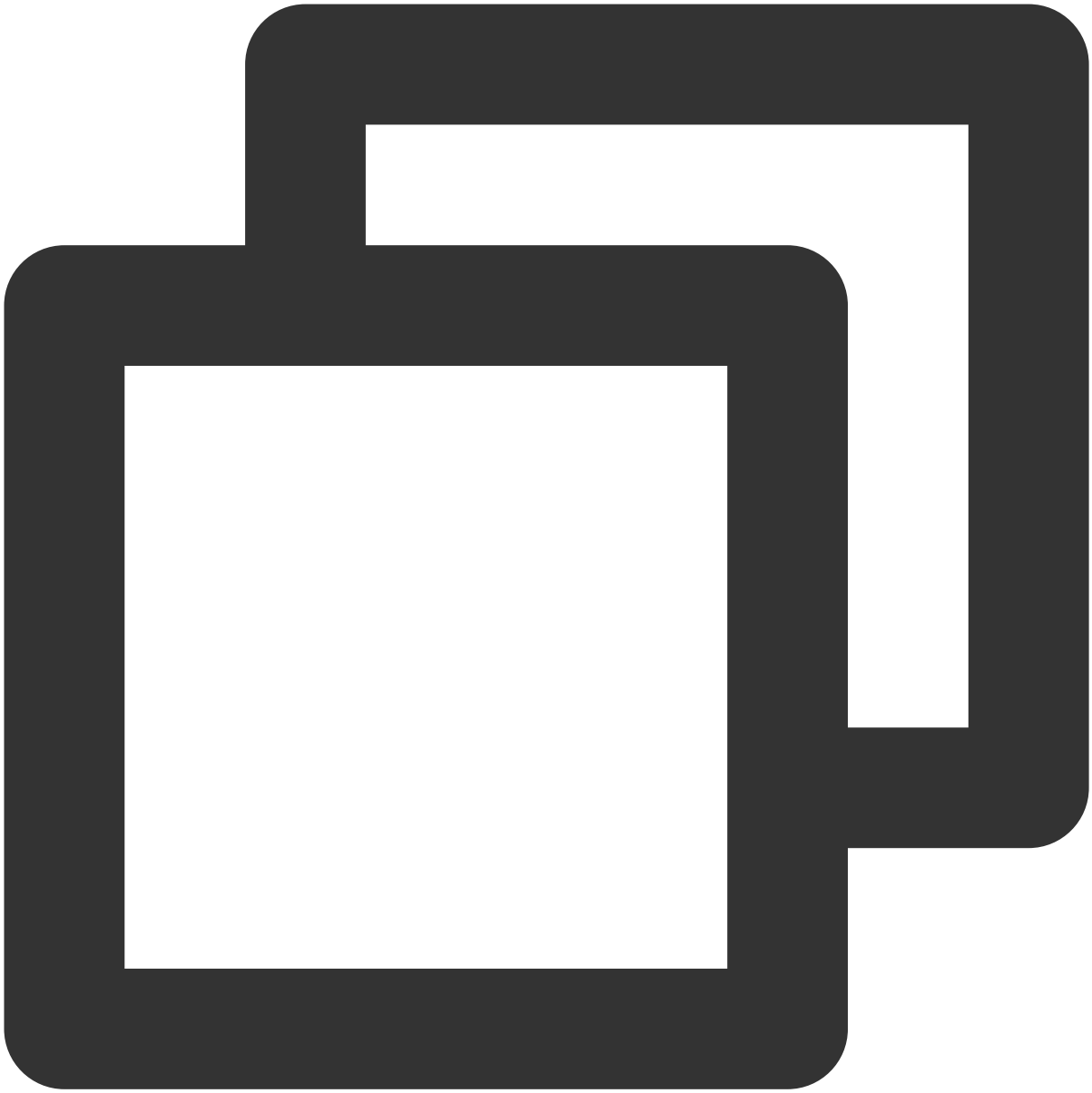
```
tmio_ = tmio::TmioFactory::createUnique(tmio::Protocol::SRT);
tmio::SrtPreset::mpegTsLossless(tmio_.get());
tmio_->setIntOption(tmio::srt_options::CONNECT_TIMEOUT, 4000);
tmio_->setBoolOption(tmio::base_options::THREAD_SAFE_CHECK, true);
```

Create a `Tmio` instance: You can use `TmioFactory` to create it.

Configure parameters: Select different APIs to configure the parameters:

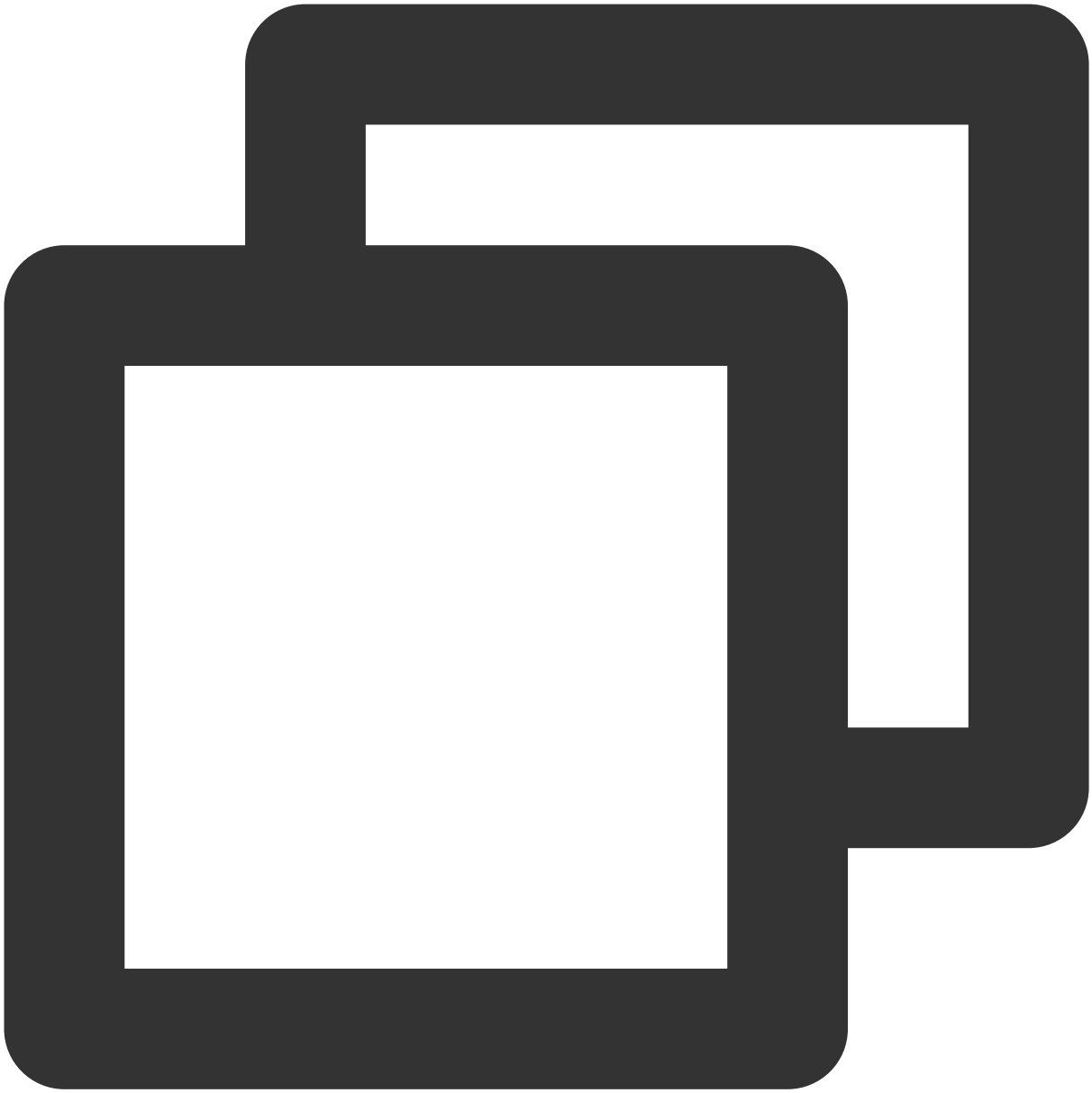
Parameters: See tmio-option.h.

Simple configuration: See tmio-preset.h.



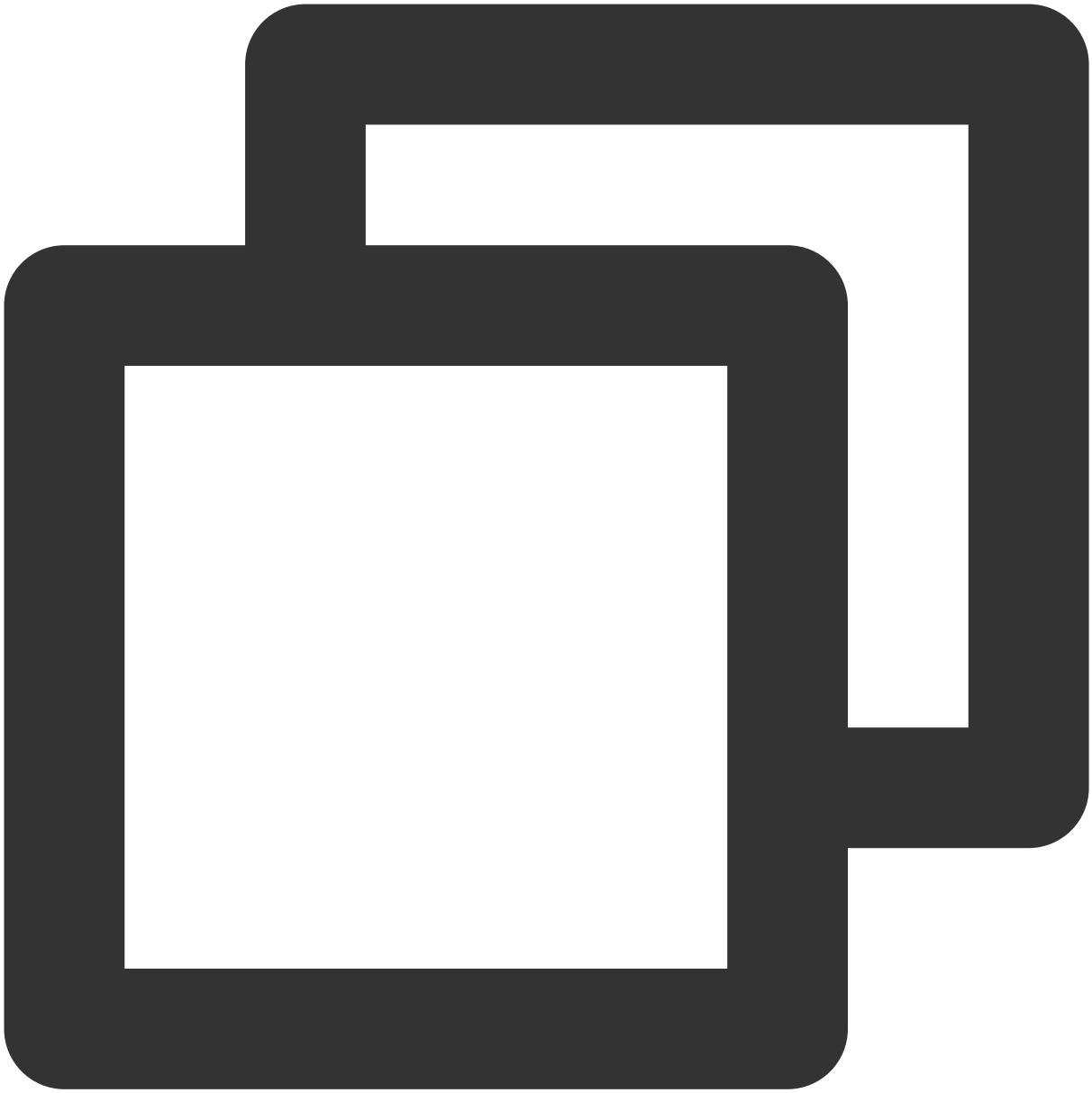
```
// Select an appropriate configuration based on different parameter attributes
bool setBoolOption(const std::string &optname, bool value);
bool setIntOption(const std::string &optname, int64_t value);
bool setDoubleOption(const std::string &optname, double value);
bool setStrOption(const std::string &optname, const std::string &value);
...
```

2. **Start connection**:



```
/**
 * open the stream specified by url
 *
 * @param config protocol dependent
 */
virtual std::error_code open(const std::string &url,
                             void *config = nullptr) = 0;
```

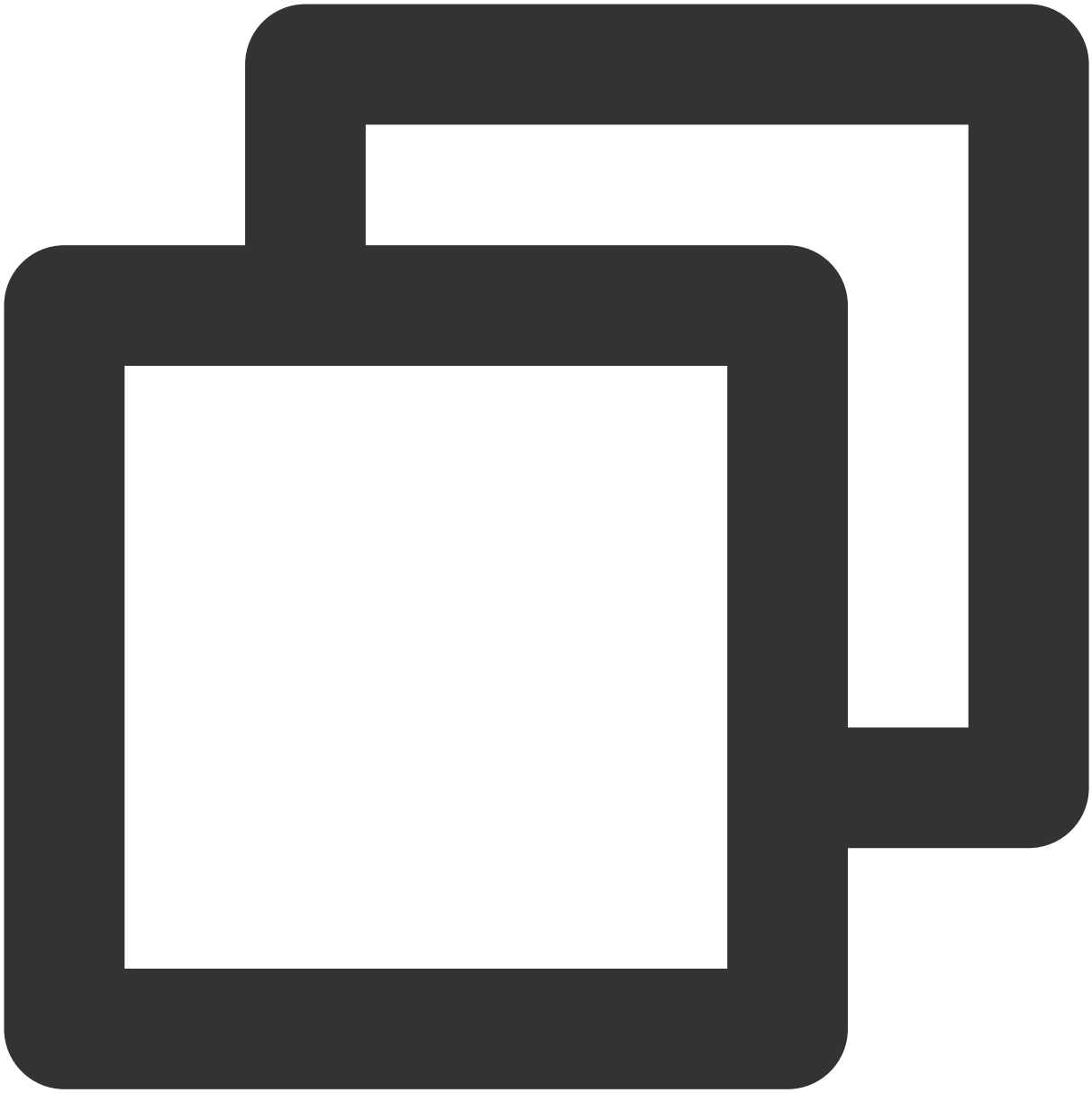
A single connection (`config` can be `NULL`)



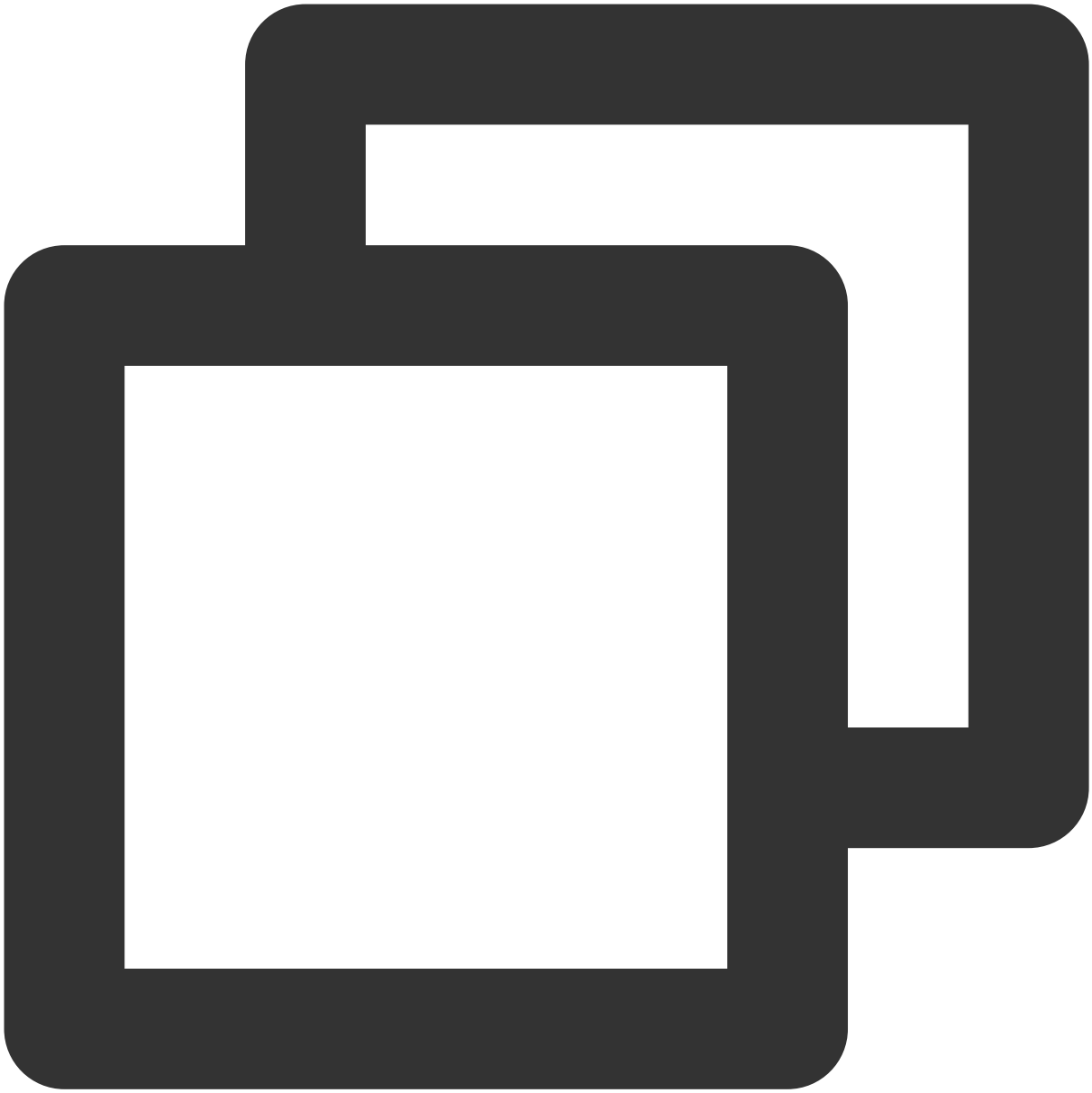
```
// A single connection by default
auto err = tmio->open(TMIO_SRT_URL);
if (err) {
LOGE("open failed, %d, %s", err.value(), err.message().c_str());
}
```

Multi-connection bonding (currently, only the SRT protocol is supported)

For how to set `config` for the SRT bonding feature, see the definition of the `TmioFeatureConfig` structure in the `tmio.h` file.



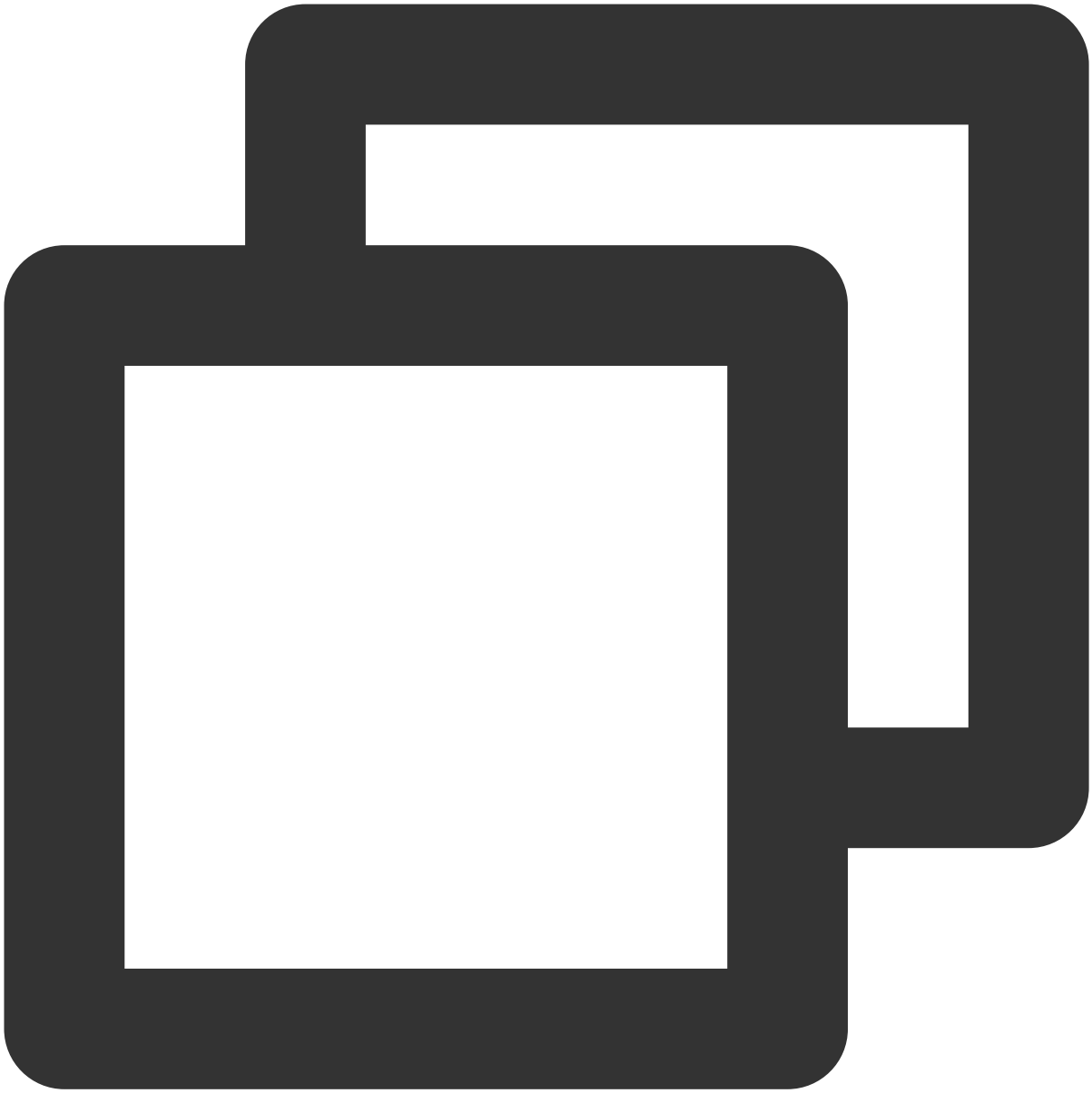
```
tmio::TmioFeatureConfig option_;
option_.protocol = tmio::Protocol::SRT;
option_.trans_mode = static_cast<int>(tmio::SrtTransMode::SRT_TRANS_BACKUP);
option_.addAvailableNet(net_name, local_addr, remote_url, 0, weight, -1);
```



```
// Multi-connection bonding
auto err = tmio_->open(TMIO_SRT_URL, &option_);
if (err) {
 LOGE("open failed, %d, %s", err.value(), err.message().c_str());
}
```

With multi-connection bonding, you can use the `open` API to add new transfer connections to the group.

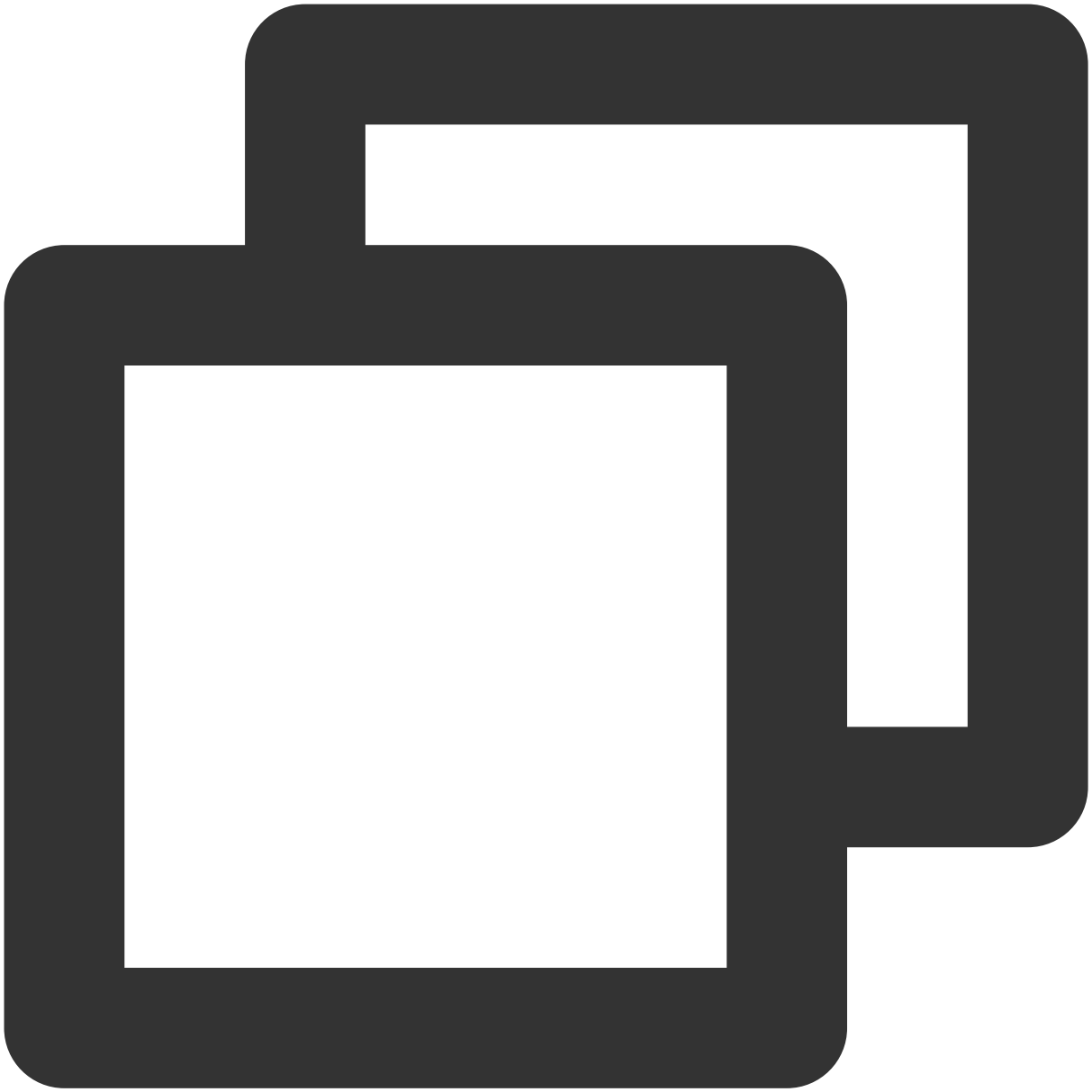
3. **Send data**:



```
int ret = tmio_->send(buf.data(), datalen, err);
if (ret < 0) {
        LOGE("send failed, %d, %s", err.value(), err.message().c_str());
        break;
}
```

4. **Receive data**:

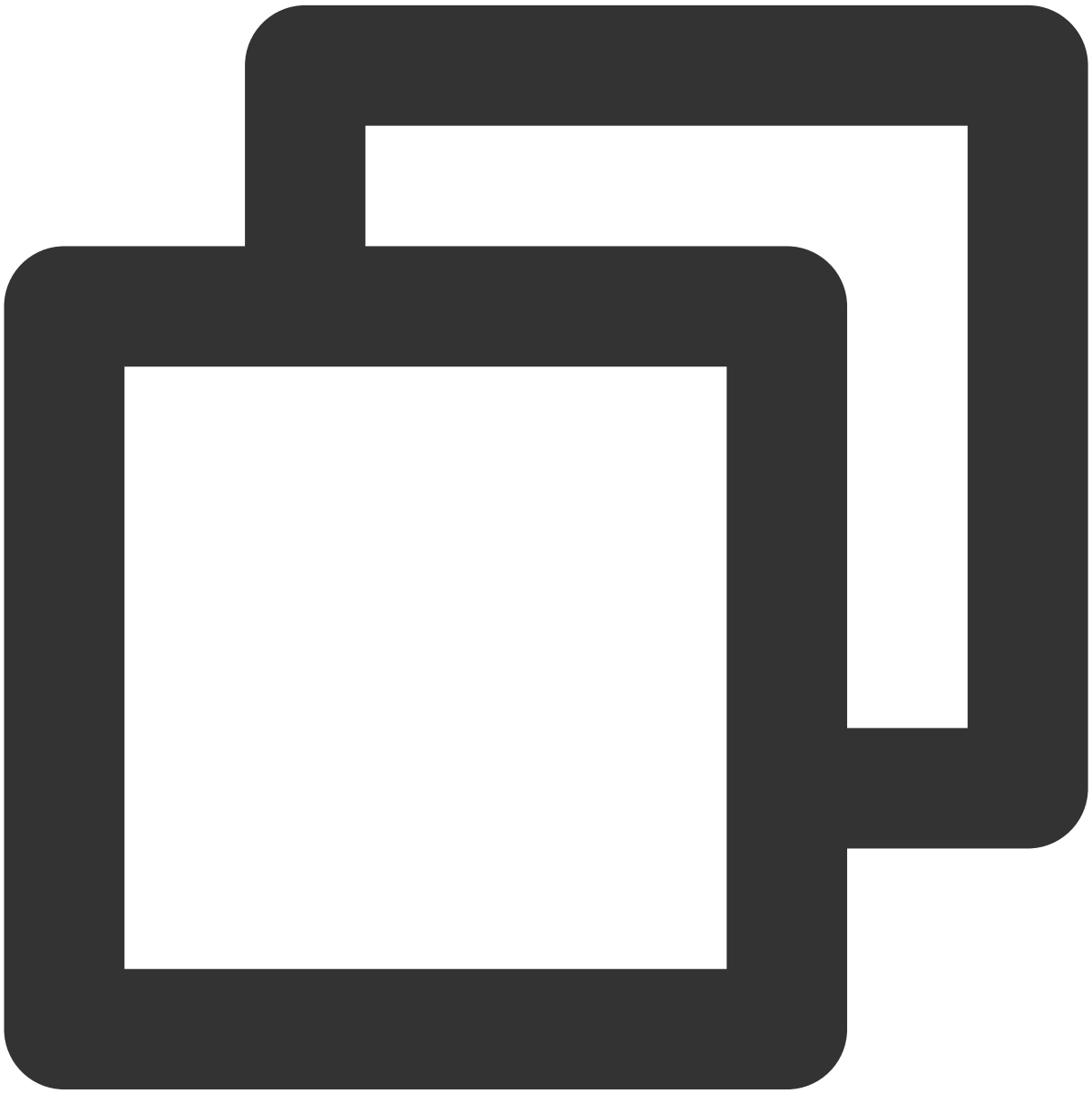
For protocols that involve interactions, such as RTMP, you need to call an API to read the data. We offer two APIs for this:



```
/**
* receive data
*
* @param err return error details
* @return number of bytes which were received, or < 0 to indicate error
*/
virtual int recv(uint8_t *buf, int len, std::error_code &err) = 0;

using RecvCallback = std::function<bool(const uint8_t *buf, int len, const std::err
/**
* receive data in event loop
```

```
*
* recvLoop() block current thread, receive data in a loop and pass the data to recv
* @param recvCallback return true to continue the receive loop, false for break
*/
virtual void recvLoop(const RecvCallback &recvCallback) = 0;
```

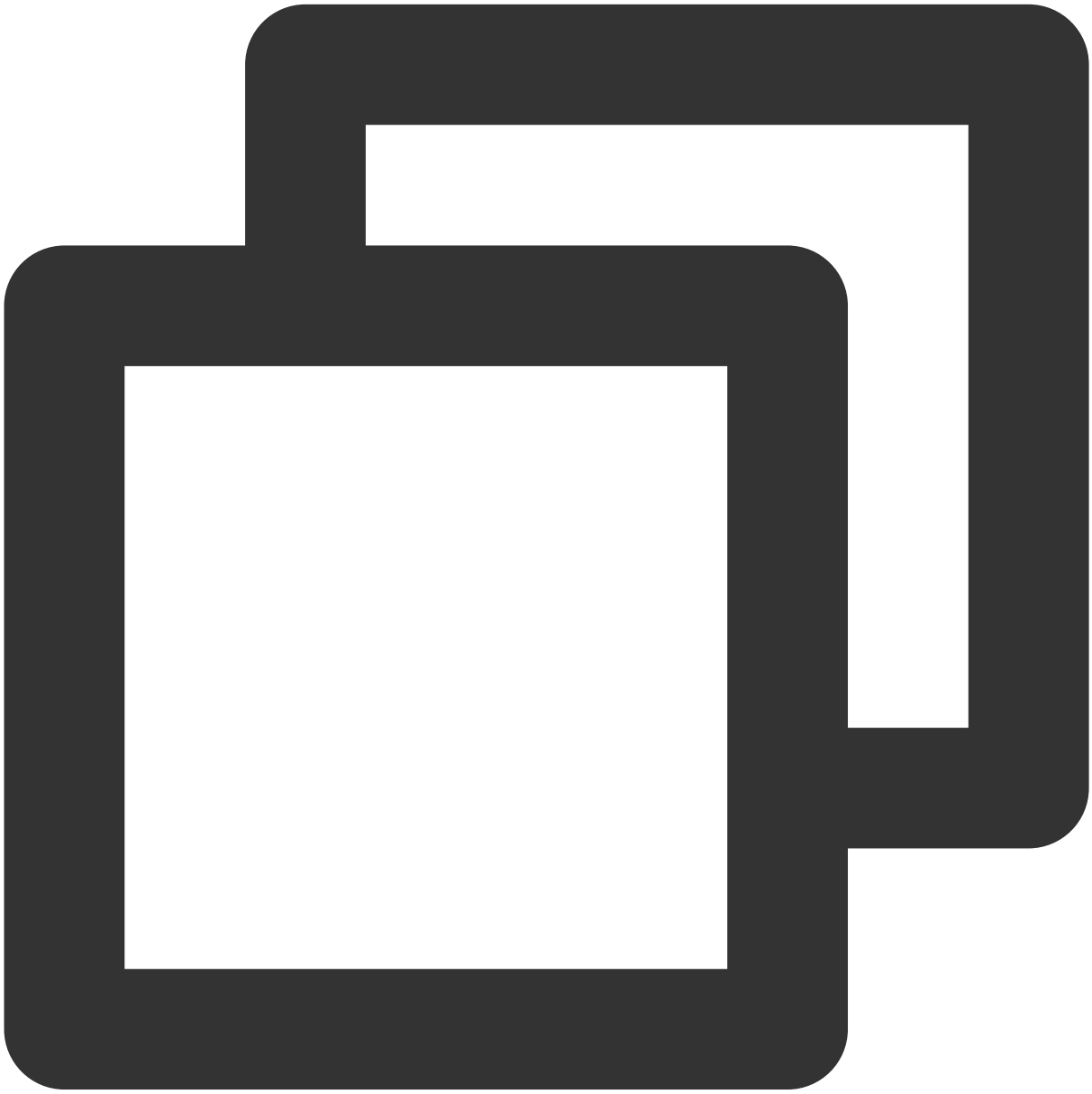
Loop read of the upper-layer application:



```
        while (true) {
    ret = tmio_->recv(buf.data(), buf.size(), err);
    if (ret < 0) {
        LOGE("recv error: %d, %s", err.value(), err.message().c_str());
```

```
            break;
        }
        ...
    }
```

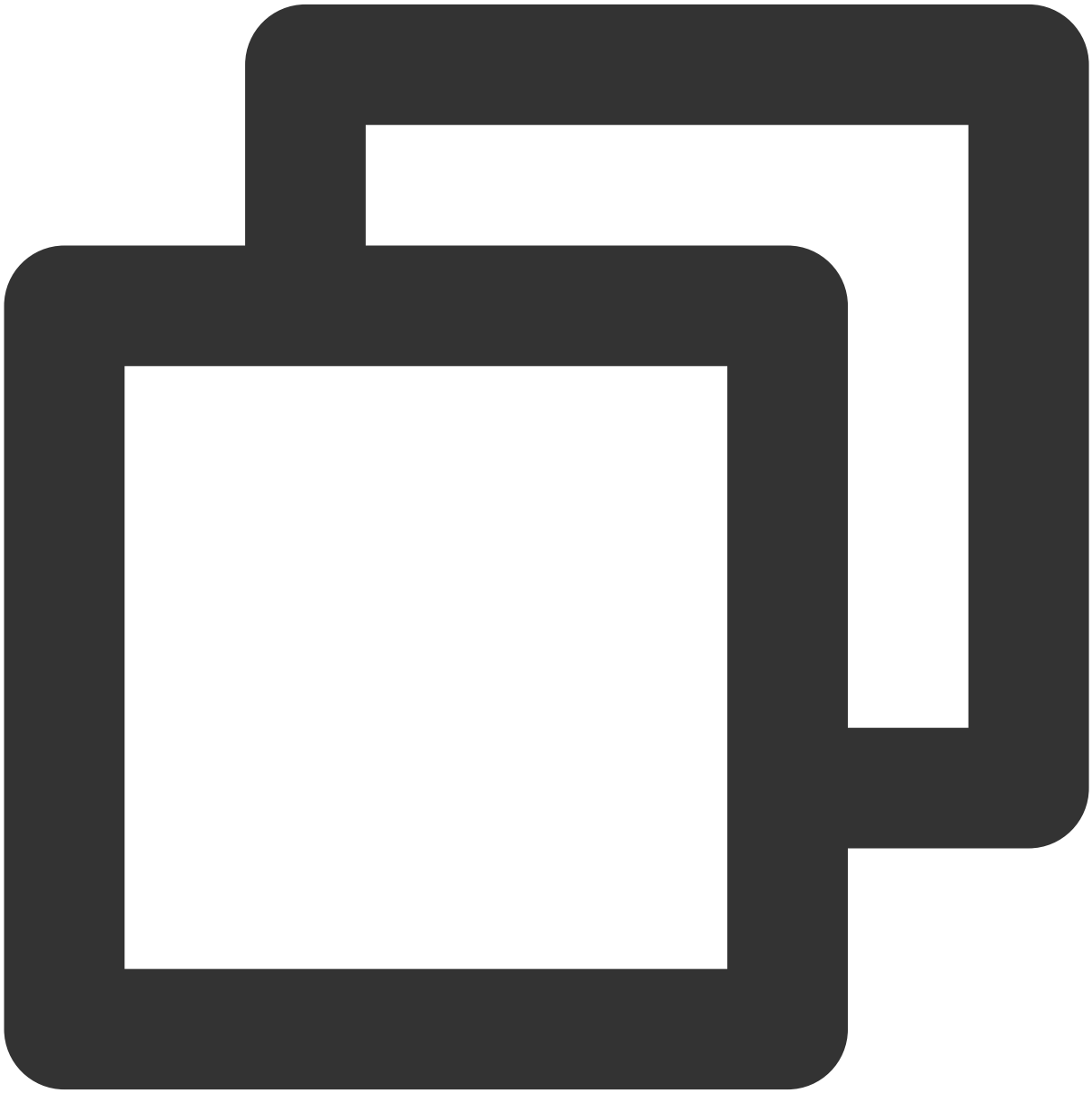
Read through callback:



```
FILE *file = fopen(output_path, "w");
tmio_->recvLoop([file](const uint8_t *buf, int len,
                                                const std::error_code &err) {
```

```
    if (len < 0) {
            fwrite(buf, 1, len, file);
    } else if (len < 0) {
            LOGE("recv error: %d, %s", err.value(), err.message().c_str());
    }
    return true;
});
```

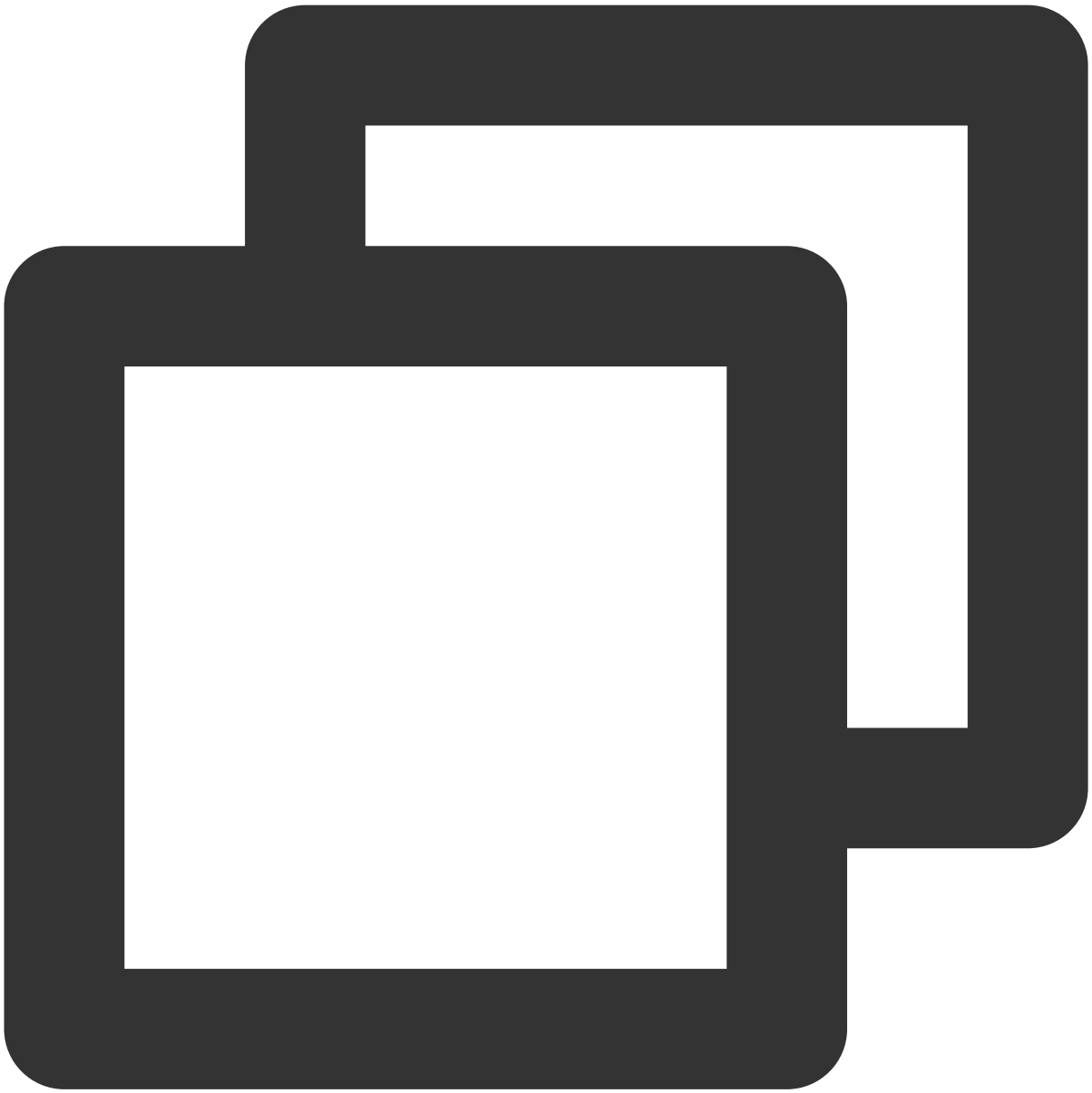
5. **Terminate the `Tmio` instance**:



```
tmio_->close();
```

6. **More**:

Get the current connection status (your application can adjust its stream push policy based on the status):



```
tmio::PerfStats stats_;
tmio_->control(tmio::ControlCmd::GET_STATS, &stats_);
```

API and Demo Updates

To get the latest updates on the APIs and demos of the SDK, visit this GitHub page.

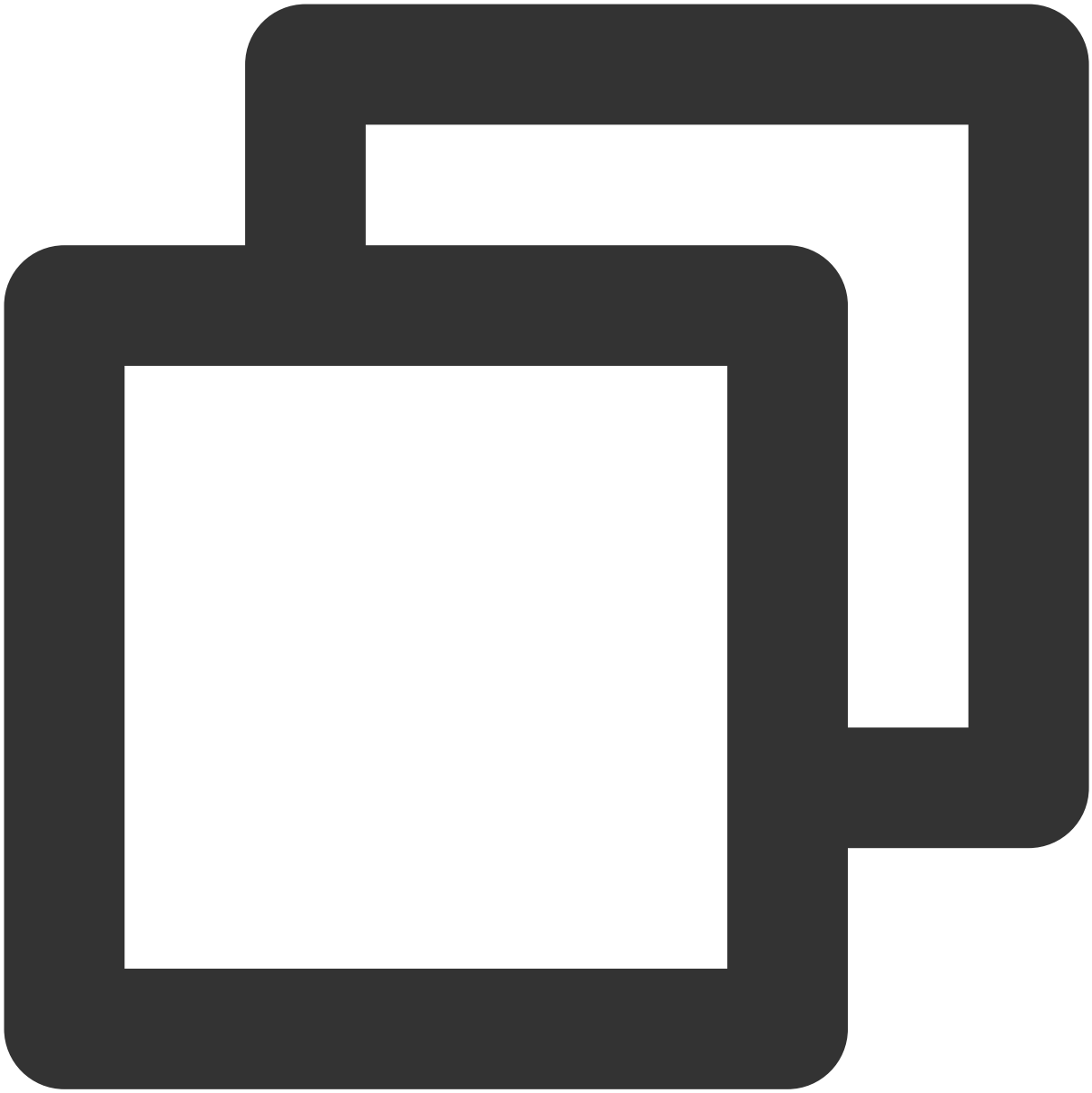
FAQs

Is SRT multi-connection bonding supported for all devices?

Only devices with multiple available network interfaces can use multi-connection bonding, and Android devices can use this feature only if the Android version is 6.0 or later and the API level is 23 or above.

How do I enable the 4G/5G data network on an Android phone connected to Wi-Fi?

An Android phone connected to Wi-Fi cannot transfer data directly over the 4G/5G network. To enable the data network, apply for the data network permission as follows:



```
ConnectivityManager connectivityManager = (ConnectivityManager) context.getSystemSe
NetworkRequest request = new NetworkRequest.Builder().addTransportType(NetworkCapab
                                        .addCapability(NetworkCapabilities.NET_C
                                        .build();

ConnectivityManager.NetworkCallback networkCallback = new ConnectivityManager.Netwo
    @Override
    public void onAvailable(@NonNull Network network) {
                Log.d(TAG, "The mobile data network has been enabled");
        super.onAvailable(network);
    }
```

```
}
```

Integrating libLebConnection SDK into a Player

Last updated：2022-12-23 10:56:16

Overview

The libLebConnection SDK provides upgraded transfer capabilities based on the native WebRTC. It allows you to connect your existing player to LEB with just a few simple modifications. Based on LVB-compatible stream push and cloud-based media processing capabilities, it can implement live streaming at a low latency even under high concurrency, so as to help you smoothly migrate from standard LVB streaming to low-latency LEB streaming. For large rooms in modern real-time communication (RTC) scenarios, it can also help you quickly implement relayed live streaming at low costs and low latency.

Feature Description

- The libLebConnection SDK can pull audio/video streams at a low latency even under poor network conditions.
- It can play back H.264. H.265, and AV1 videos with B frames and output them as raw video frame data such as Annex B and OBU files for H.264/H.265 and AV1 input videos respectively.
- It can play back AAC and Opus audios and output them as raw audio frame data.
- It supports Android, iOS, Windows, Linux, and macOS.

Integration Method

Basic API description

- Create a LEB connection.

```
LEB_EXPORT_API LebConnectionHandle* OpenLebConnection(void* context, LebLogLeve
l loglevel);
```

- Register a callback function.

```
LEB_EXPORT_API void RegisterLebCallback(LebConnectionHandle* handle, const LebC
allback* callback);
```

- Start the connection to pull the stream.

```
LEB_EXPORT_API void StartLebConnection(LebConnectionHandle* handle, LebConfig c
onfig);
```

- Stop the connection.

```
LEB_EXPORT_API void StopLebConnection(LebConnectionHandle* handle);
```

- Close the connection.

```
LEB_EXPORT_API void CloseLebConnection(LebConnectionHandle* handle);
```

Callback API description

```
typedef struct LebCallback {
// The log callback
OnLogInfo onLogInfo;
// The video information callback
OnVideoInfo onVideoInfo;
// The audio information callback
OnAudioInfo onAudioInfo;
// The video data callback
OnEncodedVideo onEncodedVideo;
// The audio data callback
OnEncodedAudio onEncodedAudio;
// The `MetaData` callback
OnMetaData onMetaData;
// The statistics callback
OnStatsInfo onStatsInfo;
// The error callback
OnError onError;
} LebCallback;
```

Note：
For the detailed definitions of data structures, see `leb_connection_api.h`.

API call process

1. **Create an LEB connection**: `OpenLebConnection()`
2. **Register various callback functions**: `RegisterXXXXCallback()`
3. **Start the connection to pull the stream**: `StartLebConnection()`
4. **Call back and output the raw audio/video data**:

- OnEncodedVideo()
- OnEncodedAudio()

5. **Stop the connection**: `StopLebConnection()`
6. **Close the connection**: `CloseLebConnection`

Integration sample

This sample describes how to integrate libLebConnection into the typical open-source player ijkplayer widely used on Android devices. For how to integrate it on other platforms, you can also refer to the sample code.

Latest SDK version

You can download the libLebConnection SDK here.

FAQs on Integration

How do I develop the lag statistics collection feature?

As buffering is disabled, you can now collect lag statistics based on the video rendering interval. If the video rendering interval exceeds a certain threshold, it will be counted as a lag, and the lag duration will be added to the total lag duration.
Taking ijkplayer as an example, you can develop the lag statistics feature as follows:

1. **Modify the code**
   i. Add the variables required by lag statistics collection to the `VideoState` and `FFPlayer` structures.

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay_def.h b/ijkmedia/ijkplayer/ff_ffplay_de
f.h
index 00f19f3c..f38a790c 100755
--- a/ijkmedia/ijkplayer/ff_ffplay_def.h
+++ b/ijkmedia/ijkplayer/ff_ffplay_def.h
@@ -418,6 +418,14 @@ typedef struct VideoState {
SDL_cond *audio_accurate_seek_cond;
volatile int initialized_decoder;
```

```
int seek_buffering;
+
+ int64_t stream_open_time;
+ int64_t first_frame_display_time;
+ int64_t last_display_time;
+ int64_t current_display_time;
+ int64_t frozen_time;
+ int frozen_count;
+ float frozen_rate;
} VideoState;
/* options specified by the user */
@@ -720,6 +728,14 @@ typedef struct FFPlayer {
char *mediacodec_default_name;
int ijkmeta_delay_init;
int render_wait_start;
int low_delay_playback;
+ int frozen_interval;
int high_level_ms;
int low_level_ms;
int64_t update_plabyback_rate_time;
int64_t update_plabyback_rate_time_prev;
} FFPlayer;
#define fftime_to_milliseconds(ts) (av_rescale(ts, 1000, AV_TIME_BASE))
@@ -844,6 +860,15 @@ inline static void ffp_reset_internal(FFPlayer *ffp)
ffp->pf_playback_volume = 1.0f;
ffp->pf_playback_volume_changed = 0;
ffp->low_delay_playback = 0;
ffp->high_level_ms = 500;
ffp->low_level_ms = 200;
+ ffp->frozen_interval = 200;
ffp->update_plabyback_rate_time = 0;
ffp->update_plabyback_rate_time_prev = 0;
av_application_closep(&ffp->app_ctx);
ijkio_manager_destroyp(&ffp->ijkio_manager_ctx);
```

ii. **Add the logic for lag statistics collection**

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay.c b/ijkmedia/ijkplayer/ff_ffplay.c
index 714a8c9d..c7368ff5 100755
--- a/ijkmedia/ijkplayer/ff_ffplay.c
+++ b/ijkmedia/ijkplayer/ff_ffplay.c
@@ -874,6 +874,25 @@ static void video_image_display2(FFPlayer *ffp)
VideoState *is = ffp->is;
Frame *vp;
Frame *sp = NULL;
+ int64_t display_interval = 0;
+
```

```
+ if (!is->first_frame_display_time){
+ is->first_frame_display_time = SDL_GetTickHR() - is->stream_open_time;
+ }
+
+ is->last_display_time = is->current_display_time;
+ is->current_display_time = SDL_GetTickHR() - is->stream_open_time;
+ display_interval = is->current_display_time - is->last_display_time;
+ av_log(NULL, AV_LOG_DEBUG, "last_display_time:%"PRId64" current_display_tim
e:%"PRId64" display_interval:%"PRId64"\n", is->last_display_time, is->current_dis
play_time, display_interval);
+
+ if (is->last_display_time > 0) {
+ if (display_interval > ffp->frozen_interval) {
+ is->frozen_count += 1;
+ is->frozen_time += display_interval;
+ }
+ }
+ is->frozen_rate = (float) is->frozen_time / is->current_display_time;
+ av_log(NULL, AV_LOG_DEBUG, "frozen_interval:%d frozen_count:%d frozen_time:%"PR
Id64" is->current_display_time:%"PRId64" frozen_rate: %f ", ffp->frozen_interval,
is->frozen_count, is->frozen_time, is->current_display_time, is->frozen_rate);
vp = frame_queue_peek_last(&is->pictq);
```

Note：
In this sample, the initial value of the lag threshold (`frozen_interval`) is `200(ms)` , which can be adjusted as needed.

2. **Test lag statistics collection**
   Use QNET to simulate poor network conditions for the test as follows:
   i. Download QNET here.
   ii. Open QNET, click **Add** > **Template Type** > **Custom Template** and configure a template and parameters for poor network conditions as needed (the following is 30% random network packet loss during downstreaming).
   iii. Select a program from the program list.
   iv. Enable the configuration of poor network conditions for testing.

Note：
To facilitate testing, you can modify the above lag parameters and deliver the data to the Java layer through JNI to display it.

How do I eliminate the noise during playback? (optimization of SoundTouch for Android)

To adjust the playback rate based on the buffer watermarks, you need to use SoundTouch to adjust the audio speed. However when many network fluctuations occur, multiple speed adjustments are required as the buffer watermarks are frequently adjusted, which may cause noise when the native ijkplayer calls SoundTouch. In this case, you can refer to the following code for optimization:

When SoundTouch is called for speed adjustment in low-latency playback mode, all the buffer is translated by SoundTouch.

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay.c b/ijkmedia/ijkplayer/ff_ffplay.c
index 714a8c9d..c7368ff5 100755
--- a/ijkmedia/ijkplayer/ff_ffplay.c
+++ b/ijkmedia/ijkplayer/ff_ffplay.c
@@ -2579,7 +2652,7 @@ reload:
int bytes_per_sample = av_get_bytes_per_sample(is->audio_tgt.fmt);
resampled_data_size = len2 * is->audio_tgt.channels * bytes_per_sample;
#if defined(__ANDROID__)
- if (ffp->soundtouch_enable && ffp->pf_playback_rate != 1.0f && !is->abort_reque
st) {
+ if (ffp->soundtouch_enable && (ffp->pf_playback_rate != 1.0f || ffp->low_delay_
playback) && !is->abort_request) {
av_fast_malloc(&is->audio_new_buf, &is->audio_new_buf_size, out_size * translate_
time);
for (int i = 0; i < (resampled_data_size / 2); i++)
{
```

Why can't MediaCodec decode H.265 videos after it is enabled?

The libLebConnection SDK supports H.265 video streams, but if you enable `Using MediaCodec` in **Settings** in the native ijkplayer, H.265 video streams won't be decoded by MediaCodec. In this case, you can refer to the following code for optimization:

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay_options.h b/ijkmedia/ijkplayer/ff_ffpla
y_options.h
index b021c26e..958b3bae 100644
--- a/ijkmedia/ijkplayer/ff_ffplay_options.h
+++ b/ijkmedia/ijkplayer/ff_ffplay_options.h
@@ -178,8 +178,8 @@ static const AVOption ffp_context_options[] = {
OPTION_OFFSET(vtb_handle_resolution_change), OPTION_INT(0, 0, 1) },

// Android only options
- { "mediacodec", "MediaCodec: enable H.264 (deprecated by 'mediacodec-avc')",
- OPTION_OFFSET(mediacodec_avc), OPTION_INT(0, 0, 1) },
+ { "mediacodec", "MediaCodec: enable all_videos (deprecated by 'mediacodec_all_v
```

```
ideos')",
+ OPTION_OFFSET(mediacodec_all_videos), OPTION_INT(0, 0, 1) },
{ "mediacodec-auto-rotate", "MediaCodec: auto rotate frame depending on meta",
OPTION_OFFSET(mediacodec_auto_rotate), OPTION_INT(0, 0, 1) },
{ "mediacodec-all-videos", "MediaCodec: enable all videos",
```

WebRTC-Based Local Stream Mix

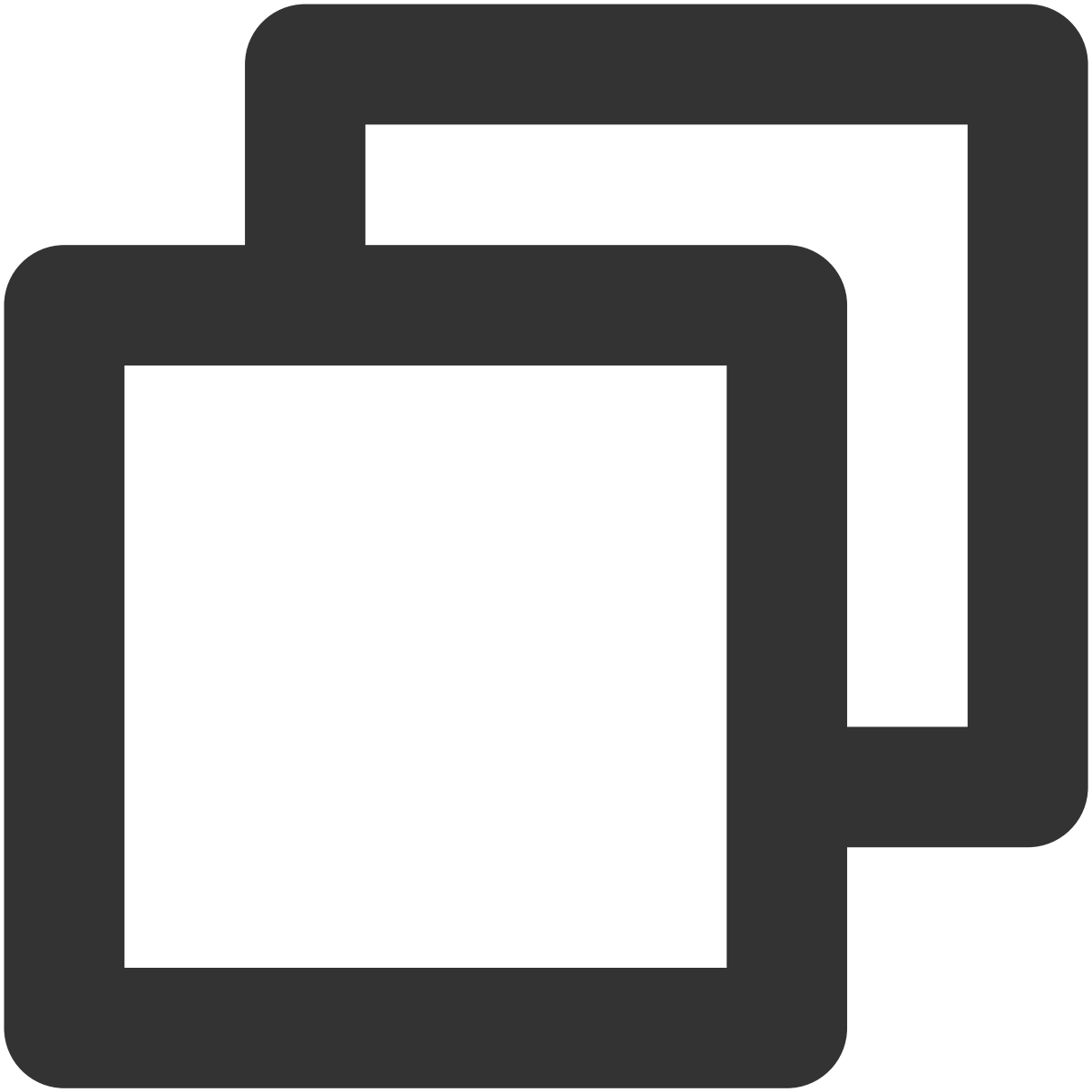
Last updated：2023-11-29 16:55:23

The SDK provides various video stream image processing features, including mixing multiple video streams (such as PiP), processing image effects (such as mirroring and filters), and adding other elements (such as watermarks and text). The basic process is as follows: The SDK first collects multiple streams and mixes them locally to combine the images and mix the audios. Then, it processes other effects. As all the features rely on the browser's support, the SDK has certain requirements for the browser performance. For the specific API protocols, see TXVideoEffectManager. This document describes the basic usage of local stream mix.

Basic Usage

To use the local stream mix feature, you should initialize the SDK and get the SDK instance `livePusher`. For the initialization code, see WebRTC Push.

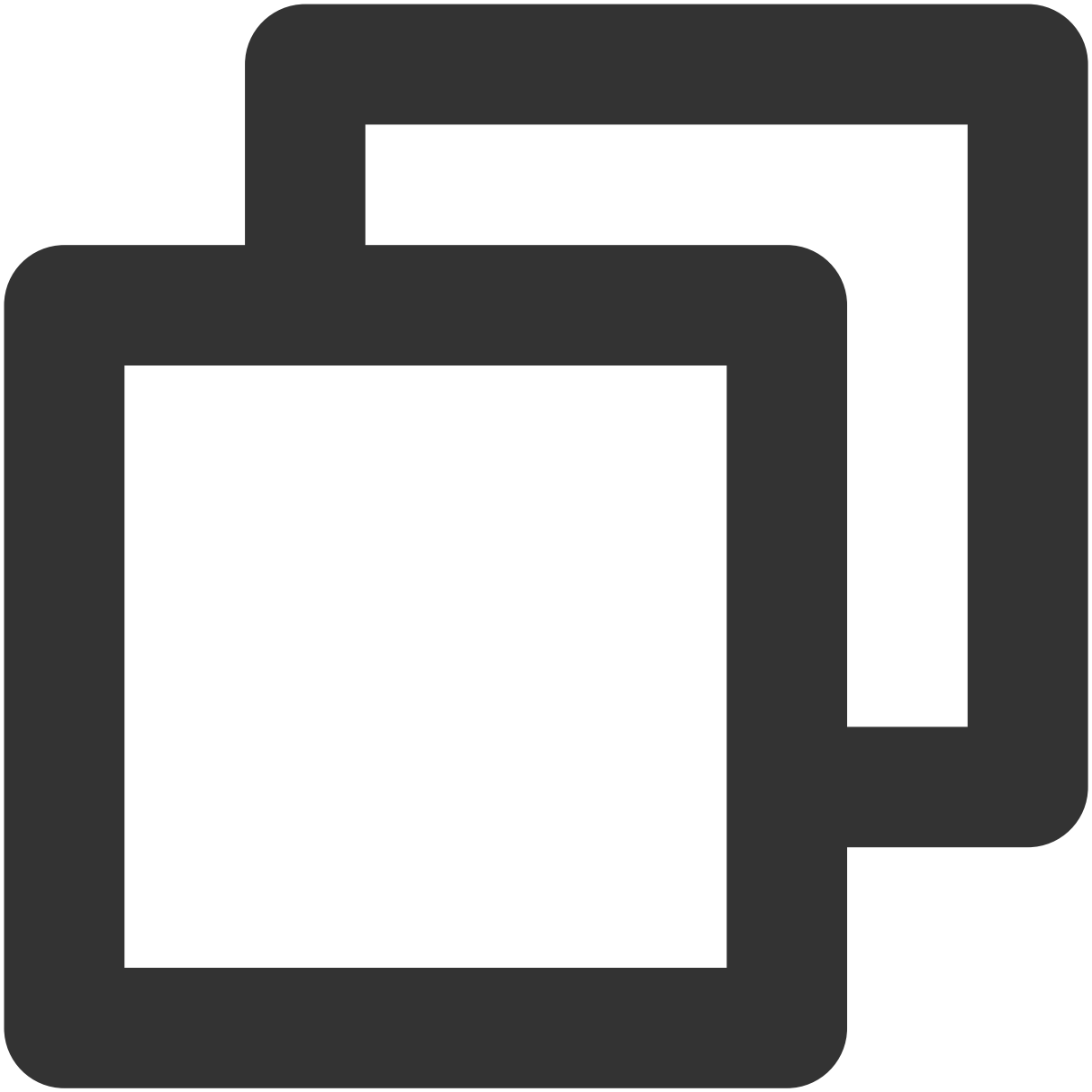
Step 1. Get the video effect management instance



```
var videoEffectManager = livePusher.getVideoEffectManager();
```

Step 2. Enable local stream mix

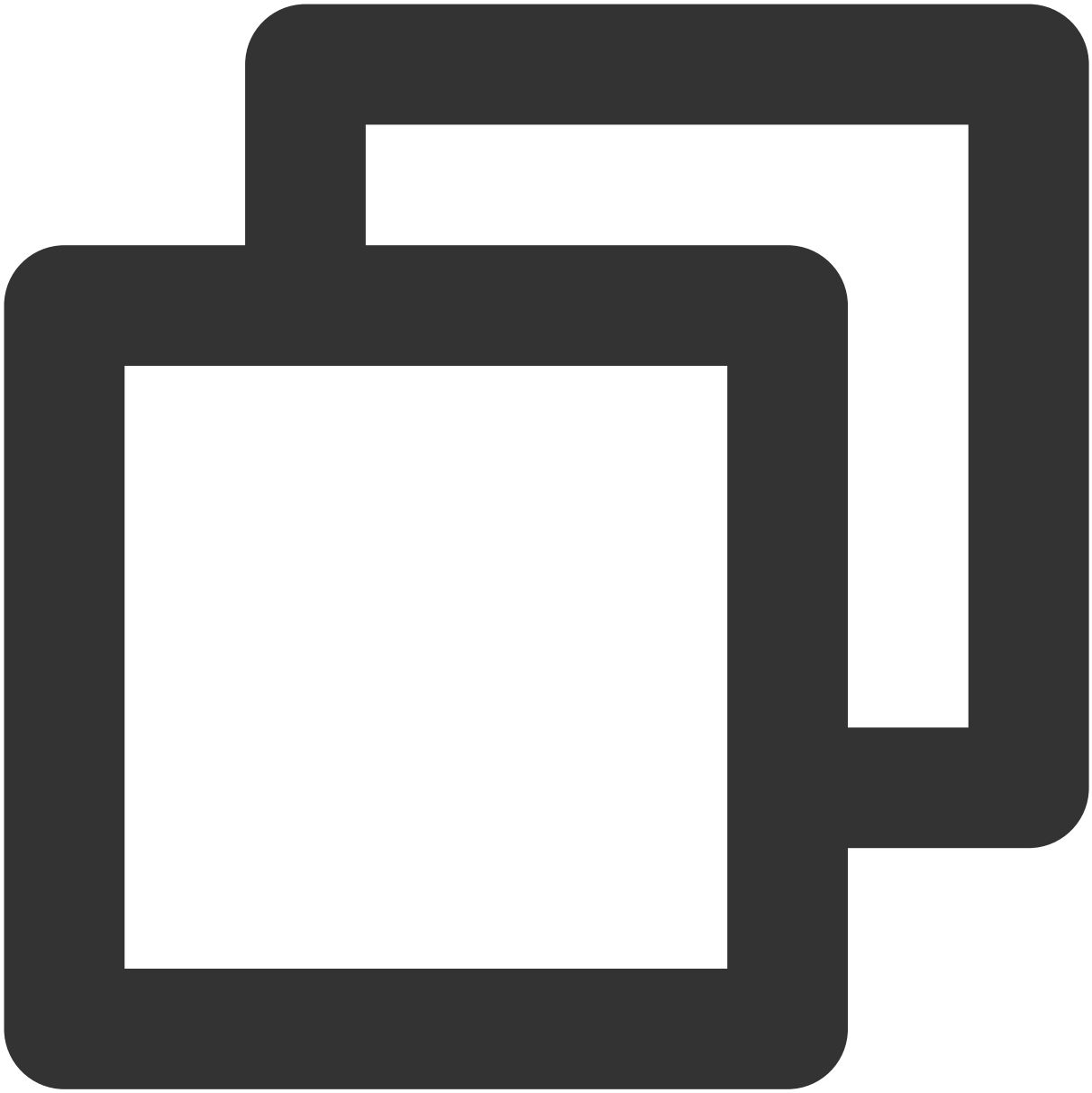
First, you need to enable the local stream mix feature. By default, the SDK collects only one video stream and audio stream. After this feature is enabled, the SDK can collect multiple streams, which will be mixed locally by the browser.



```
videoEffectManager.enableMixing(true);
```

Step 3. Set the stream mix parameters

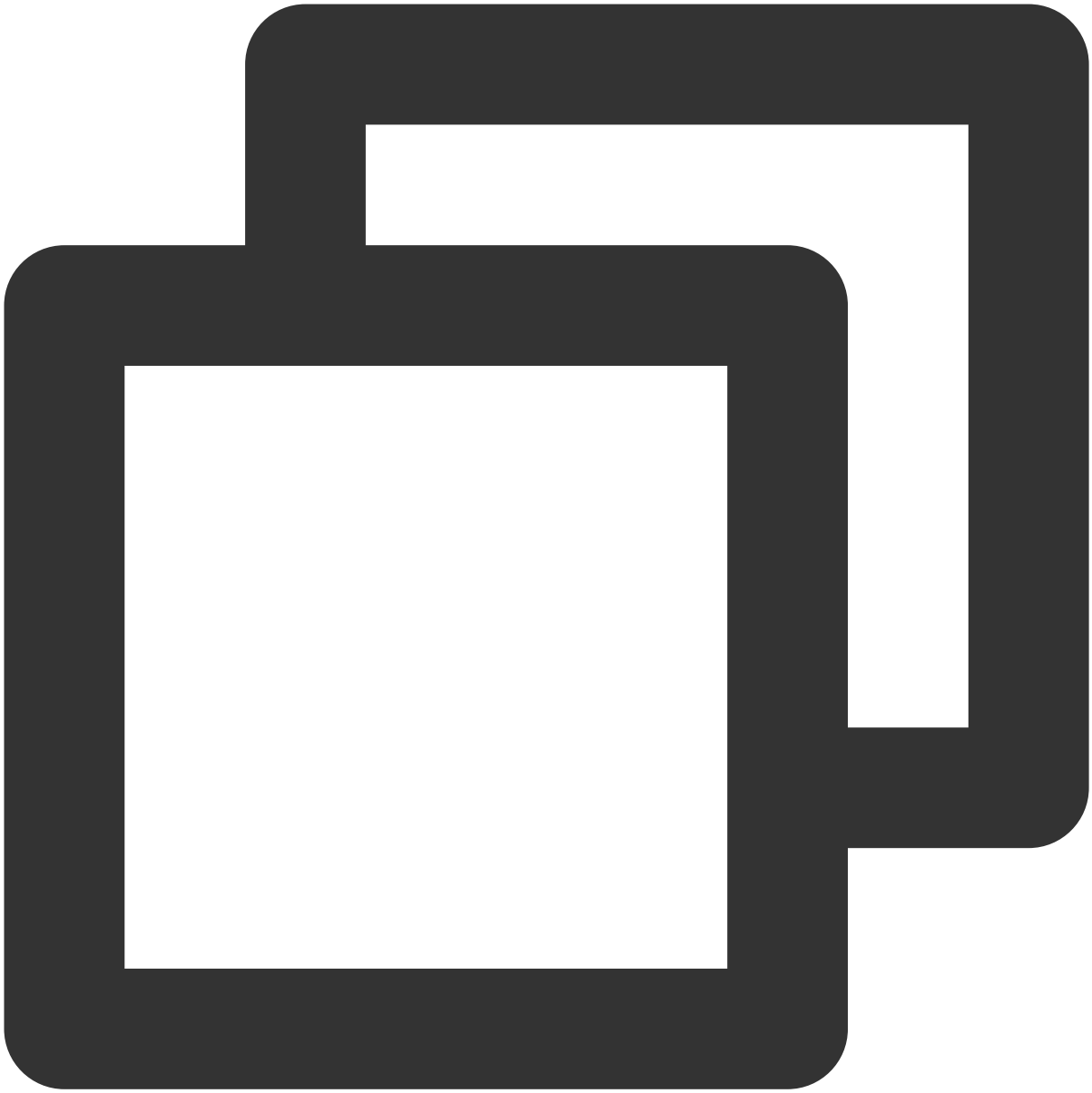
Set the stream mix parameters, especially the resolution and frame rate of the video output after stream mix.



```
videoEffectManager.setMixingConfig({
    videoWidth: 1280,
    videoHeight: 720,
    videoFramerate: 15
});
```

Step 4. Collect multiple streams

After local stream mix is enabled, the SDK starts collecting multiple streams such as the camera and shared screen. Be sure to keep the stream IDs, as they are required in subsequent operations.

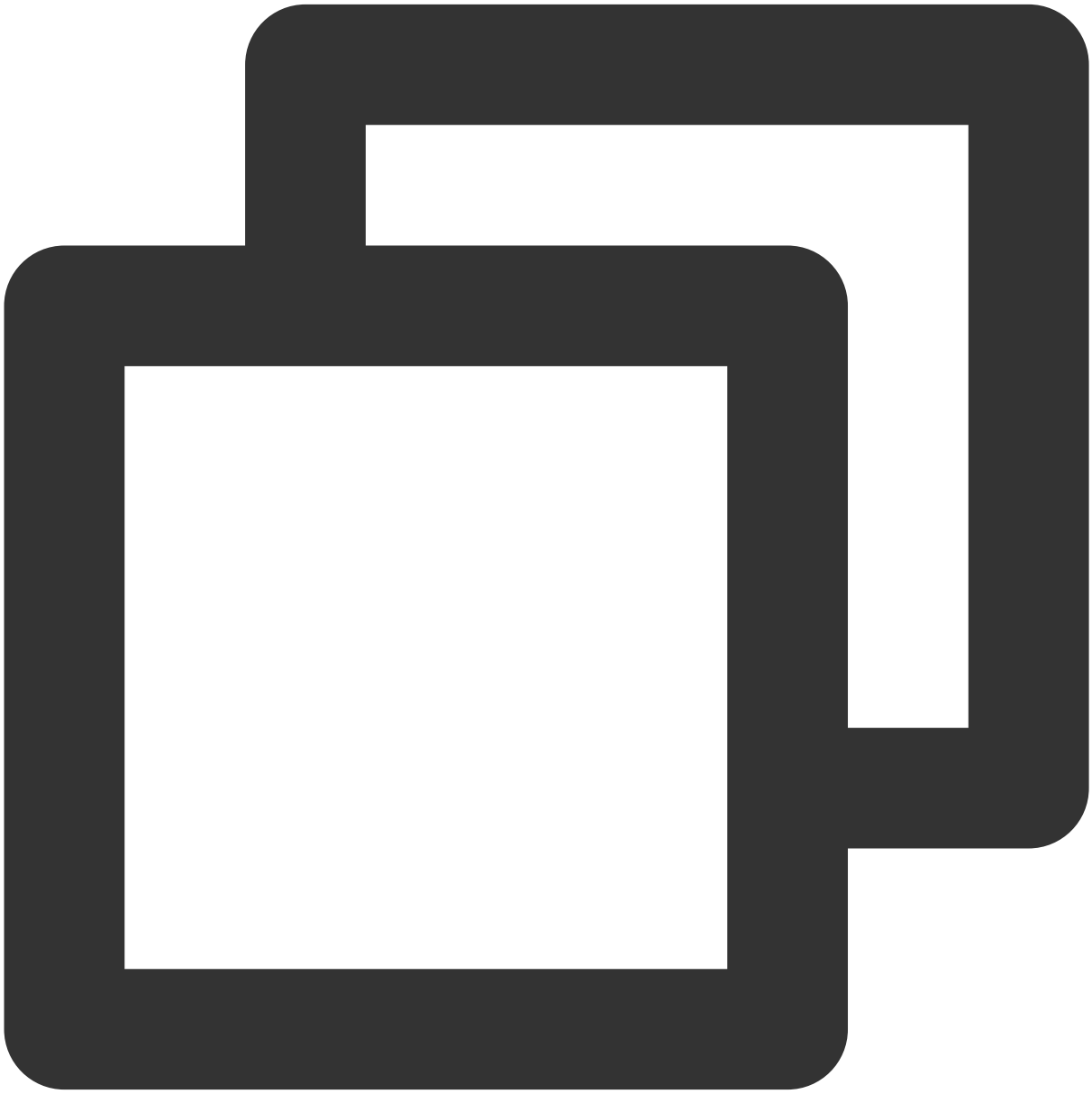


```
var cameraStreamId = null;
var screenStreamId = null;

livePusher.startCamera().then((streamId) => {
    cameraStreamId = streamId;
}).catch((error) => {
    console.log('Failed to turn on the camera:'+ error.toString());
});

livePusher.startScreenCapture().then((streamId) => {
    screenStreamId = streamId;
```

```
}).catch((error) => {
    console.log('Failed to share the screen:'+ error.toString());
});
```

Step 5. Set the image layout

Set the layout of the images of the collected two streams. Here, the shared screen is displayed as the main image, with the camera image in the top-left corner. For the specific parameter configuration, see TXLayoutConfig.

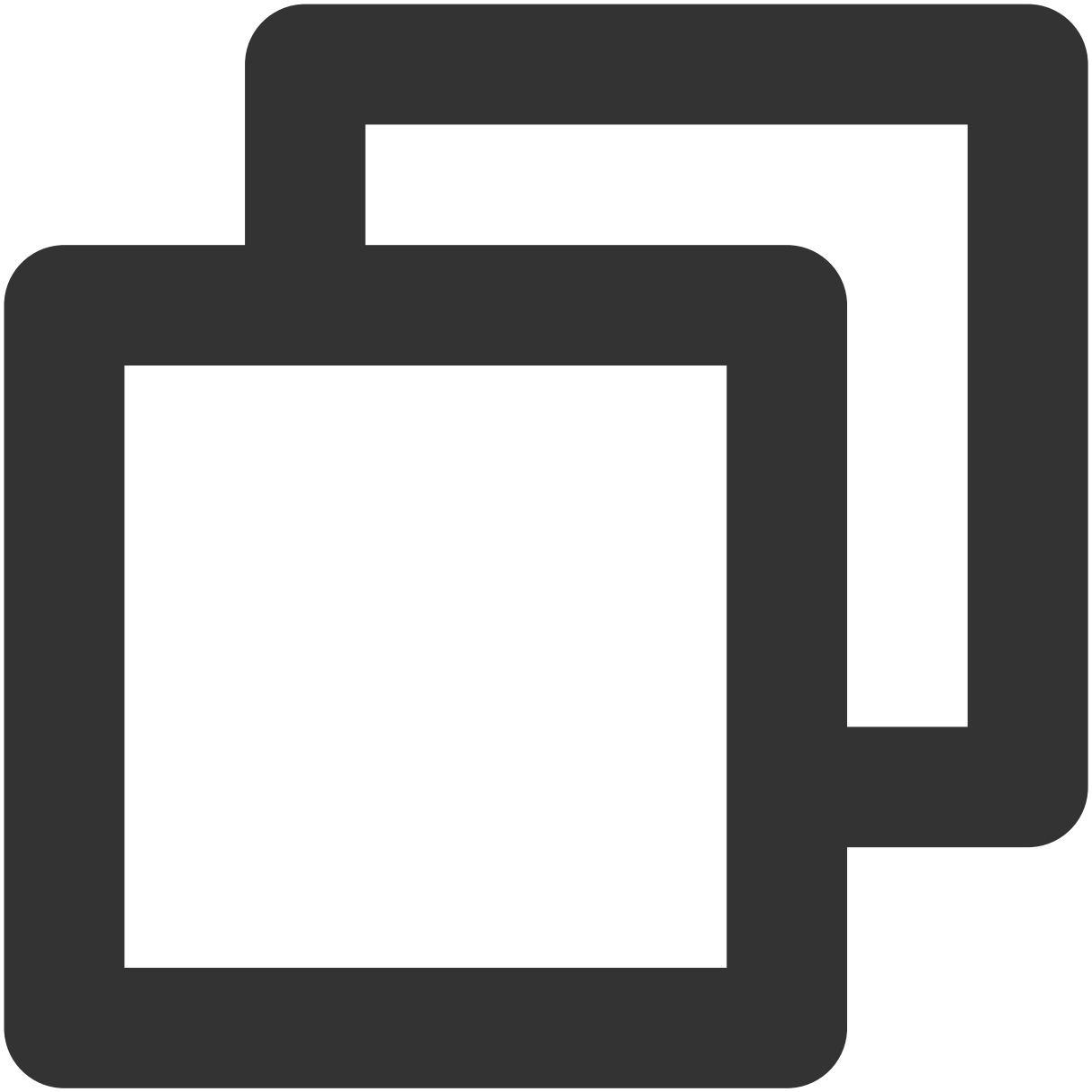


```
videoEffectManager.setLayout([{
    streamId: screenStreamId,
```

```
    x: 640,
    y: 360,
    width: 1280,
    height: 720,
    zOrder: 1
}, {
    streamId: cameraStreamId,
    x: 160,
    y: 90,
    width: 320,
    height: 180,
    zOrder: 2
}]);
```

Step 6. Set the mirroring effect

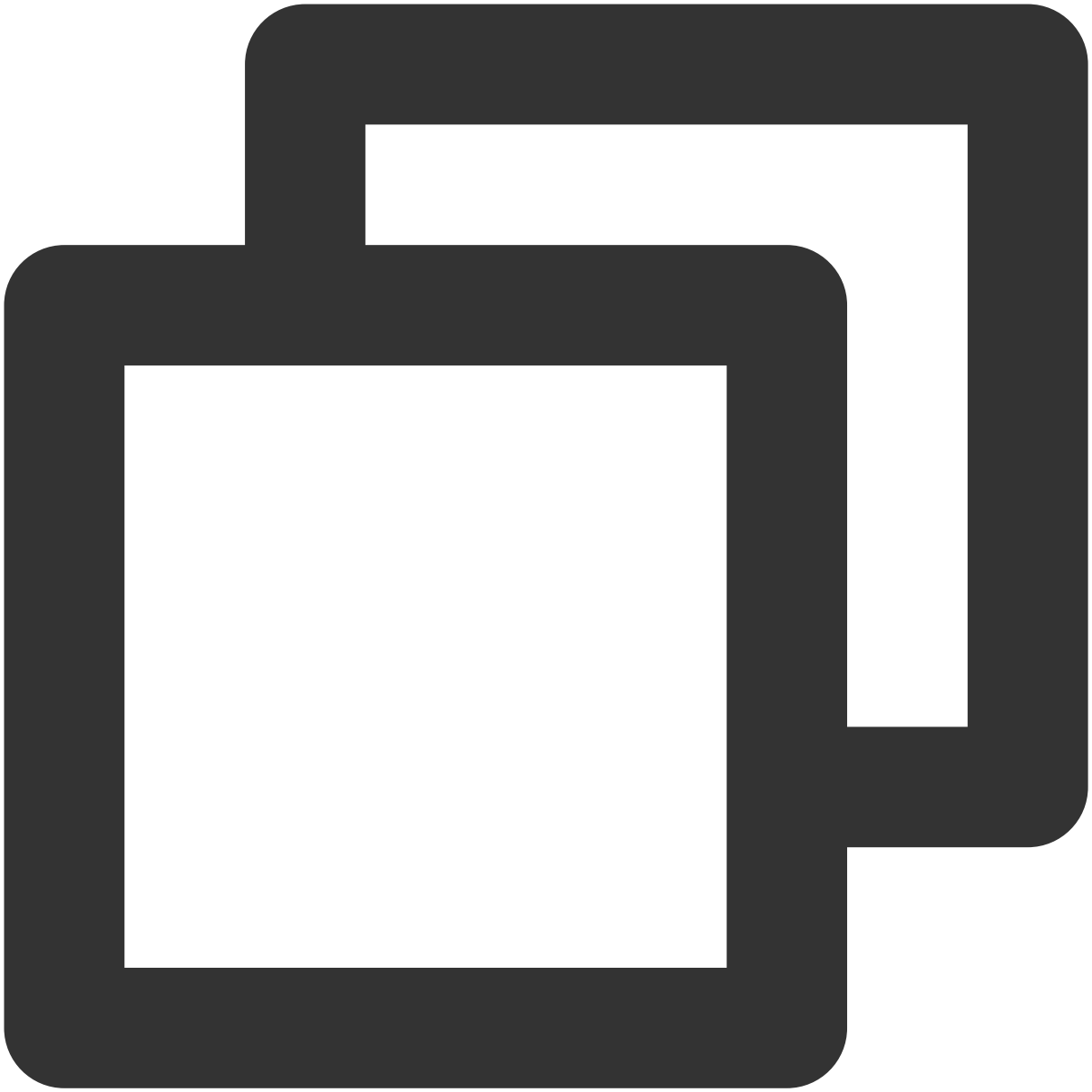
Mirror the camera image, as the image collected by the camera is actually reversed.



```
videoEffectManager.setMirror({
    streamId: cameraStreamId,
    mirrorType: 1
});
```

Step 7. Add a watermark

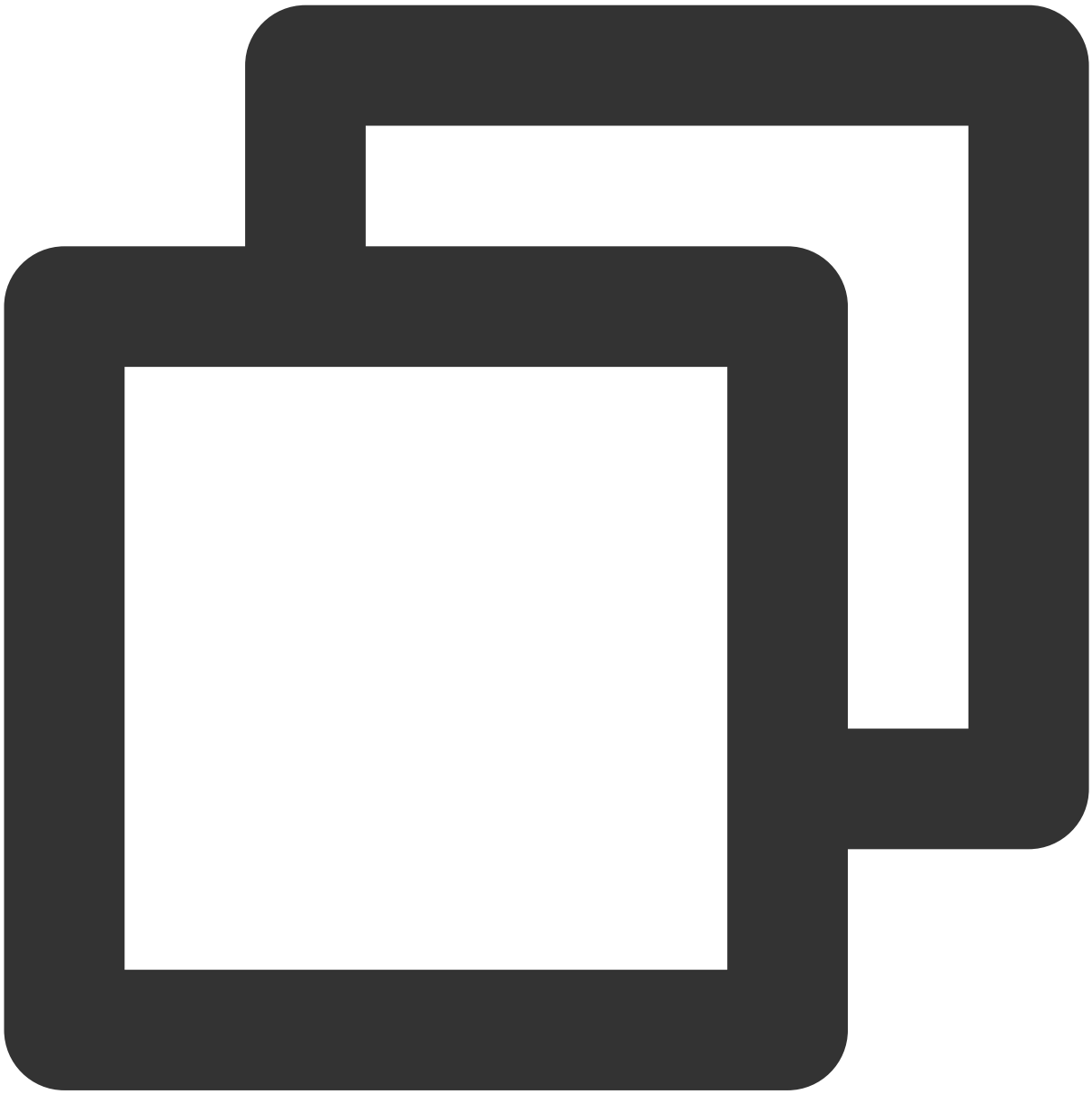
Prepare an image object and add it to the video stream image as a watermark. Here, the watermark image is placed in the top-right corner.



```
var image = new Image();
image.src = './xxx.png'; // Note that the image address cannot be under another dom

videoEffectManager.setWatermark({
    image: image,
    x: 1230,
    y: 50,
    width: 100,
    height: 100,
    zOrder: 3
});
```

Step 8. Start stream push

Push the video stream with a PiP layout, mirrored image, and watermark output after the above steps to the server.



```
livePusher.startPush('webrtc://domain/AppName/StreamName?txSecret=xxx&txTime=xxx');
```

Note

For more information on WebRTC-based stream mix APIs, see TXLivePusher.

On-Screen Comment and Chat

Last updated：2022-11-21 16:56:28

Overview

In the live streaming business, hosts and viewers usually need to interact in real time through various means such as on-screen comments and chat. However, integration of such features is complex. This document uses IM as an example to describe how to implement requirements such as on-screen comments, chat, and item recommendations during live streaming as well as possible problems and considerations, giving a glimpse of the live streaming business and requirements.

Key Feature Description

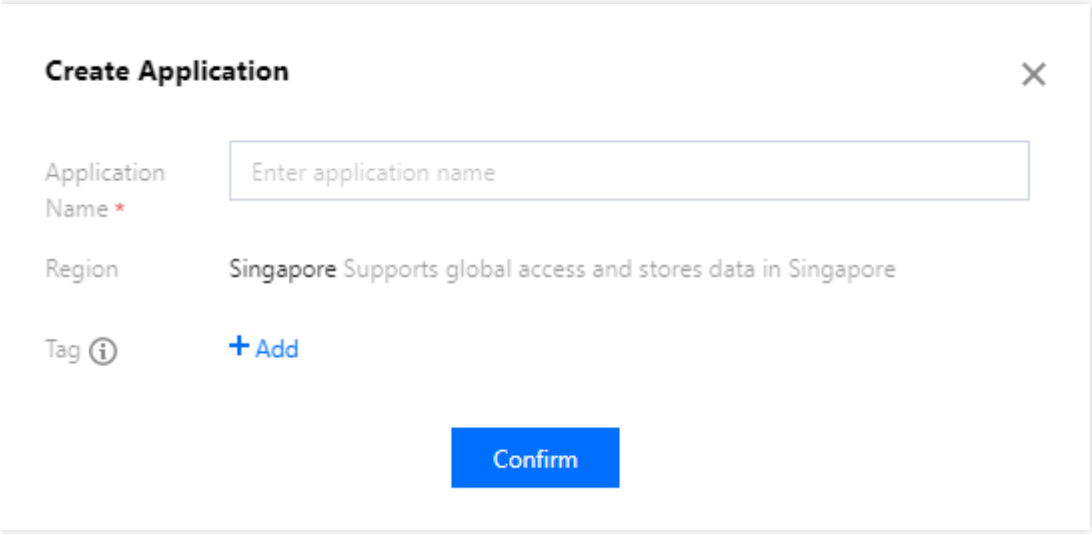
Feature	Description
Live on-screen comments, gifts, and likes	Hundreds of millions of concurrent messages can be sustained to build a friendly interaction experience.

Feature	Description
Various chat modes such as one-to-one and group live chat	Users can send messages and receive messages from other members in the same chat room. Diverse message types such as text, images, audio, and short videos can be pushed in real time, which encourages more user activity.
Item push in live shopping	In live shopping scenarios, when the host recommends an item, it needs to be immediately displayed in the item slot at the bottom of the screen and notified to all viewers. Notifications of new items are generally triggered by the virtual assistant.
Broadcast in a live room	The broadcast feature is similar to a system notice sent to live rooms. When the system admin delivers a broadcast message, all the live rooms under the `SDKAppID` will receive it.

Integration Method

Step 1. Create an application

To set up a live room in Tencent Cloud, you need to create an IM application in the console as shown below:



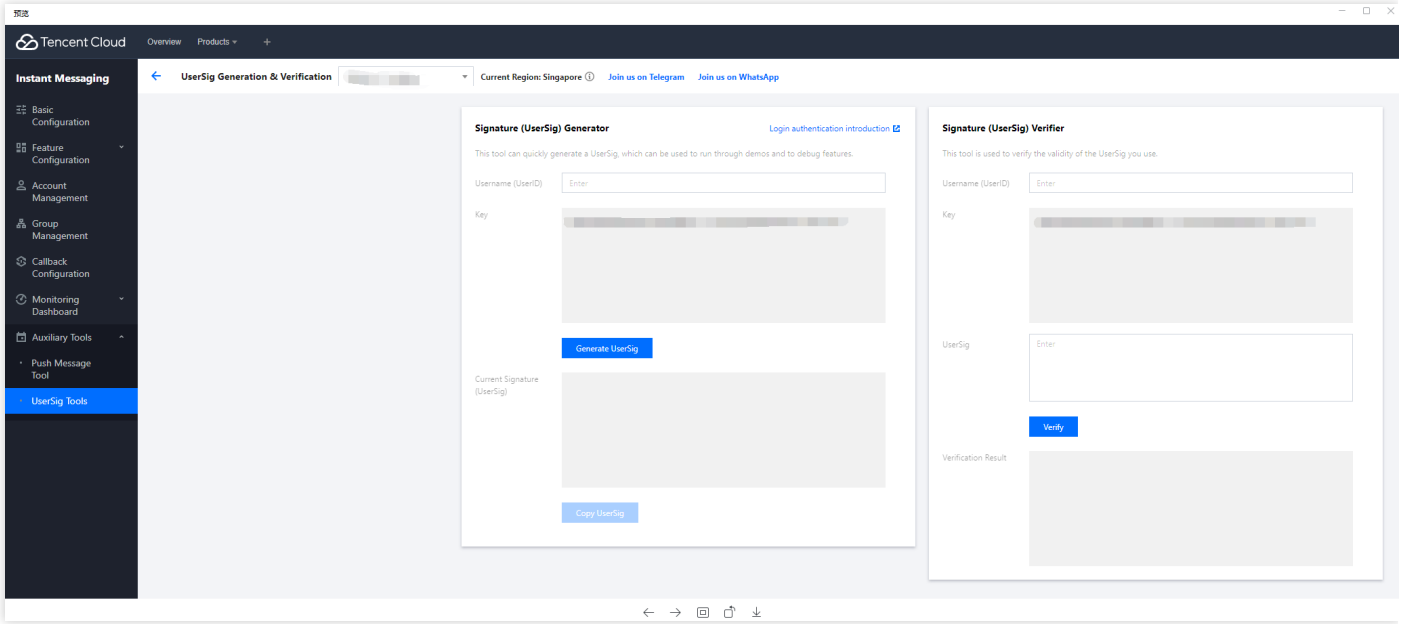
Step 2. Complete the relevant configuration

The application created in step 1 is the Free edition, which applies only to development. In the production environment, you need to activate the Pro or Ultimate edition as needed. For more information on differences between different editions, see Pricing.

In live streaming scenarios, you need some extra configurations after creating the application.

- **Calculating the `UserSig` with a key**

  In the IM account system, the password required by a user login is calculated by the server with a key provided by IM. For more information, see Generating UserSig. In the development phase, to avoid holding back development on the client, you can also calculate the `UserSig` in the console as shown below:



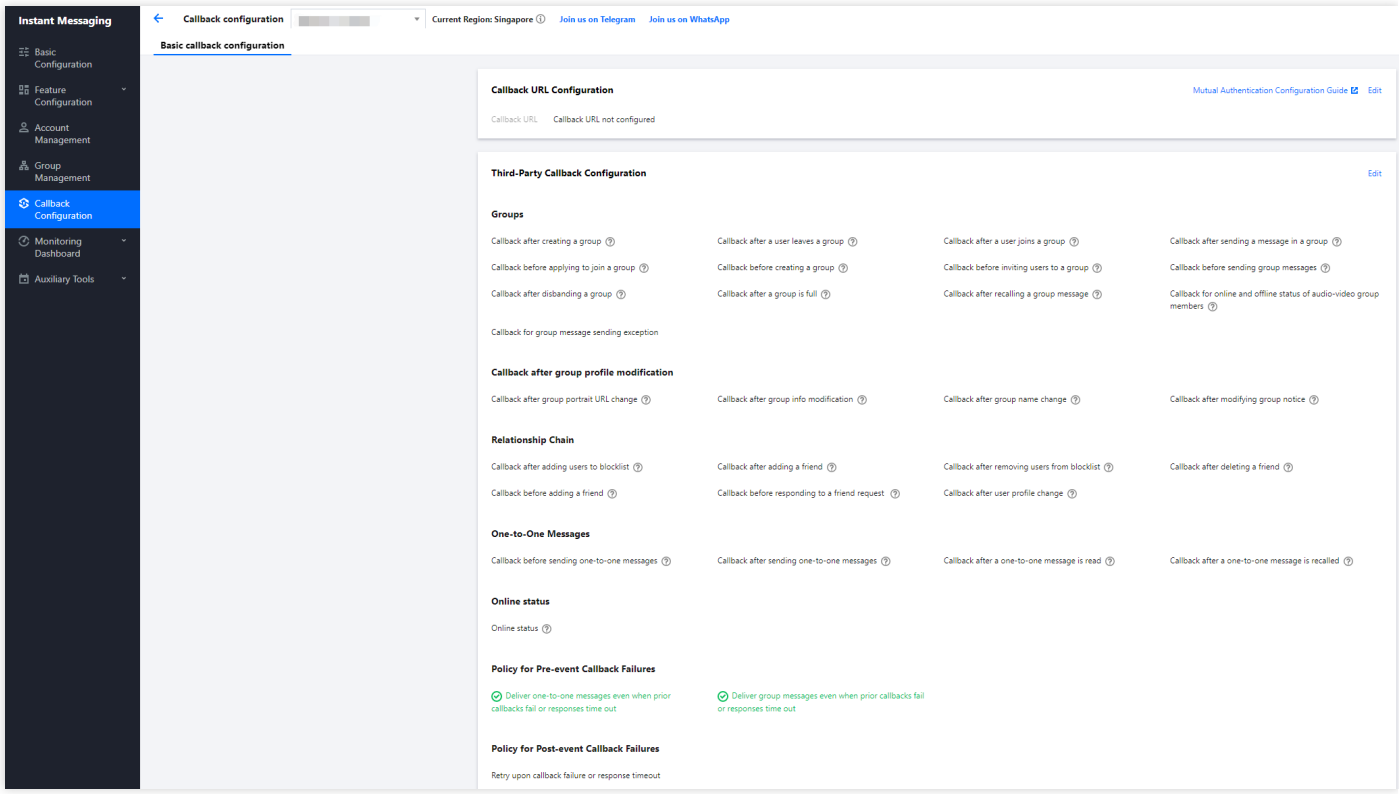
- **Configuring an admin account**

  During live streaming, an admin may need to send messages to a live room and mute (remove) non-compliant users, which can be done through APIs on the IM server as described in RESTful API List. To call these APIs, you need to create an IM admin account. By default, IM provides an account with the `UserID` of `administrator`. You can also create multiple admin accounts as needed. Note that you can create up to five admin accounts.

- **Configuring the callback address and enabling the callback**

  To implement lucky draws based on on-screen comments, message statistics collection, sensitive content detection, and other requirements, you need to use the IM callback module, where the IM backend calls back the business backend in certain scenarios. You only need to provide an HTTP API and configure it in the **Callback**

configuration module in the console as shown below:



Step 3. Integrate the client SDK

After completing the preparations, you need to integrate the IM and TRTC client SDKs into your project. You can select different integration options as needed. For detailed directions, see Get Started.

The following describes common features in a live room and provides best practices with implementation code.

Step 4. Develop key live room features

1. **Group type selection**

   The user chat section in live streaming scenarios has the following characteristics:

   - Users join and leave a group frequently, and group conversation information (unread count and `lastMessage`) doesn't need to be managed.
   - Users can join a group without approval.
   - Users send messages without caring about the chat history.
   - There are usually a large number of group members.
   - Group member information doesn't need to be stored.

Therefore, you can select **AVChatRoom** as the **Group type** for the live room based on the group characteristics of IM as described in Group System. IM audio-video groups (`AVChatRoom`) have the following characteristics:

- **Support interactive live streaming scenarios with unlimited group members.**

- Messages (group system notifications) can be pushed to all online members.
- Users can enter the group directly with no admin approval required.

> Note：
> The IM SDK for web allows users to join only one audio-video group (`AVChatRoom`) at a time. If a user logs in to an client and enters live room A, multi-client login is enabled in the console, and the user logs in to another client and enters live room B, the user will be removed from live room A.

2. **Configuration of on-screen comments, gifting, and likes for the live room**

- **On-screen comments**
  Audio-video groups (`AVChatRoom`) support on-screen comments, gifts, and like messages to build a friendly interaction experience.
  To create an on-screen comment, you can use the IM API to create a text or custom message. After the message is sent successfully, you need to get its text or custom attributes in the live room by receiving the OnRecvNewMessage() callback and then display it on the desired UI.
- **Gift**
  - Non-persistent connection requests from the client are sent to the business server, which involves the billing logic.
  - After fees are incurred, the sender can see that XXX sent the XXX gift (so that the sender can see the gift sent by himself/herself; when there are a large number of messages, the policy for discarding messages may be triggered).
  - After the fees are settled, you can call the server API to send the custom message (gift).
  - If multiple gifts are sent in a row, you need to merge the messages.
    - If the number of gifts is selected in advance, for example 99 gifts, you can send a message with `99` included in the parameter.
      - If gifts are sent multiple times and the total number is uncertain, you can send a message for every 20 gifts (the value can be adjusted) or clicks within a second. For example, if 99 gifts are clicked in a row, only five messages need to be sent after the optimization.
- **Like**
  - Unlike a gift message, a like message is not billed and directly sent on the client.
  - For like messages that need to be counted on the server, after traffic throttling is performed on the client, likes on the client are counted, and like messages within a short period of time are merged into one. The business server gets the like count in the callback before sending a message.
  - For like messages that don't need to be counted, the logic in step 2 is used, where the business server sends a message after traffic throttling is performed on the client and doesn't need to get the count in the callback before

sending a message.

3. **Item push in live shopping**
When the host recommends an item, it needs to be immediately displayed in the item slot at the bottom of the screen and notified to all viewers. Notifications of new items are generally triggered by the virtual assistant. We recommend you implement notifications of new items by enabling the admin to modify a custom group field as follows:

i. **Add a custom group field**

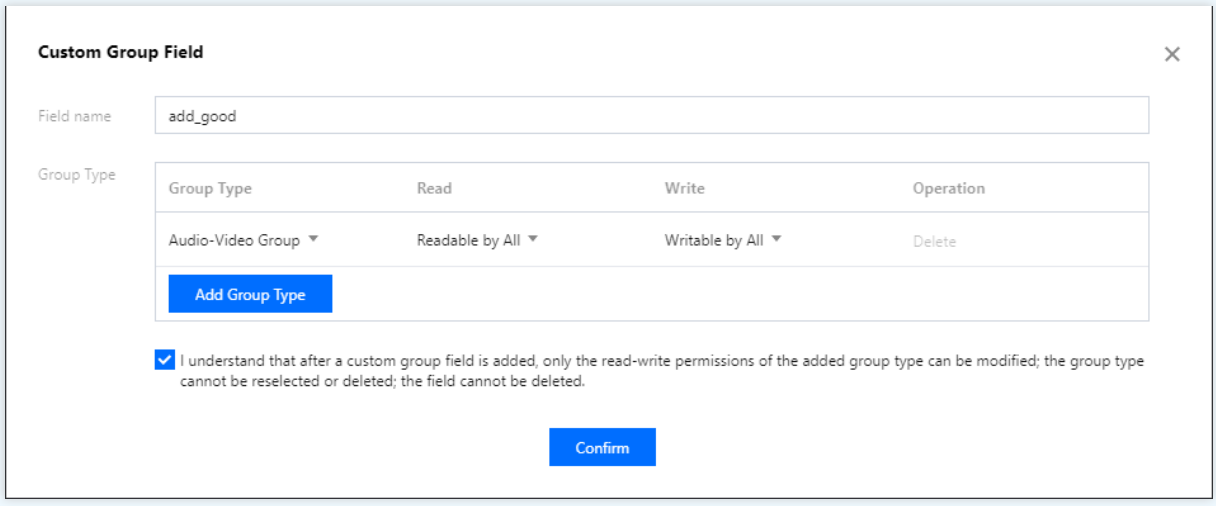
a. Log in to the IM console, click the target IM application card, and select **Feature Configuration** > **Group configuration** on the left sidebar.

b. On the **Custom Group Field** page, click **Add** in the top-right corner.

c. In the pop-up dialog box, enter a field name and set the group types and read/write permissions.

Note：

- The field name can contain up to 16 characters, supporting letters, digits, and underscores (_). It cannot begin with a digit.
- A custom group field and a custom group member field cannot have the same name.



d. Select **I understand that after a custom group field is added, only the read-write permissions of the added group type can be modified; the group type cannot be reselected or deleted; the field**

cannot be deleted. and click **Confirm**.

> Note：
> The custom group field will take effect in about ten minutes after configuration.

ii. **Use the custom group field**
The virtual assistant calls the RESTful API for modifying the profile of a group as the group admin at an appropriate time to update the custom group field, so as to send notifications of new items and notifications of live streaming status change in the live room.

4. **Broadcast in a live room**
The broadcast feature is similar to a system notice feature in the live room, but it differs from the latter in that it belongs to messaging. When the system admin delivers a broadcast message, all the live rooms under the `SDKAppID` will receive it.
The broadcast feature is currently available only for the Ultimate edition and needs to be enabled in the console. For more information on how to send a broadcast message on the business backend, see Broadcast Message of Audio-Video Group.

> Note：
> If you are not an Ultimate edition user, you can implement the feature by sending a custom group message on the server.

References

To implement more live room features such as user identity, user level, historical messages, and displaying the number of online users, see Live Room Setup Guide.