Streaming API Overview

Suggest Edits

HeyGen's **Streaming API** allows developers to seamlessly integrate dynamic **Interactive Avatars** into their applications for immersive user experiences. With this API, you can display an Interactive Avatar in a real-time seting to create *virtual assistants, training simulations,* and more, utilizing low-latency communication between *users* and *Interactive Avatars* with power of WebRTC.

Quick Start

For a quick start, you can use our *Interactive Avatar Demo* **GitHub repository.** or install our **SDK NPM Package.** directly to your project. Please see the GitHub projects for installation instructions.

Streaming API Version Comparison

v1 (Raw WebRTC) (deprecating 2/28/25)	v2 (LiveKit)
Flexibility: Full control over WebRTC setup.	**Simplicity**: Managed WebRTC via LiveKit.
Customization: Direct access to WebRTC configurations.	**Ease of Use**: Pre-built connection handling.
Difficulty: Requires WebRTC expertise.	**Setup**: Minimal configuration needed.
Environment: Works in any WebRTC-compatible setup.	**Integration**: Get up and running quickly with LiveKit client SDKs. Easily integrate with LiveKit-supported environments and the LiveKit ecosystem.

The `streaming.new` endpoint defaults to `v1`, to change to `v2` include `"version": "v2"` in the request body.

More Information

For more information about the Streaming API, please read our Interactive Avatar 101 Help center article

About the Streaming Protocol

The **Streaming API** leverages the WebRTC (Web Real-Time Communication) protocol to ensure low-latency, secure communication between browsers and applications. This foundational technology offers advantages including **low latency**, **cross-browser compatibility**, **data security**, and **scalability**. For detailed information on WebRTC, visit the WebRTC website or WebRTC API page on MDN.

Use Cases

From *interactive e-learning* and *gaming* to *virtual customer support* and *immersive entertainment*, Interactive Avatars have the potential to transform industries.

In summary, the **Streaming API** empowers developers to create dynamic and interactive experiences by embedding Interactive Avatars into apps and websites of all kinds. Whether you're building *virtual assistants, customer support, or immersive training simulations,* this API offer the tools needed to bring avatars to life within your applications, marking a significant advancement in user engagement and interactivity.

Streaming API v2 Integration: using LiveKit

Suggest Edits
This guide demonstrates how to use the Streaming API endpoints with version v2 and the LiveKit client SDK for real-time video streaming, which provides a simpler development interface.

For Node.js environments, we strongly recommend using the Streaming Avatar SDK package, as it offers a more robust solution. This guide focuses on the raw LiveKit implementation, intended for basic use cases as well as for developers who require more customization options or wish to integrate with existing LiveKit infrastructure.
❓

Implementation Guide

Overview

In this guide:

- We will use the LiveKit CDN client for easy setup without requiring npm packages.
- Simplified WebSocket handling.
- Support for both Talk (LLM) and Repeat modes.
- Real-time event monitoring for both WebSocket and LiveKit events.

Prerequisites

- API Token from HeyGen
- Basic understanding of JavaScript and LiveKit

Step 1: Basic HTML Setup

Create an HTML file with the necessary elements and include the LiveKit JS Client SDK minified CDN version :

HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script src="https://cdn.jsdelivr.net/npm/livekit-
client/dist/livekit-client.umd.min.js"></script>
  </head>

  <body>
    <div>
      <div>
        <div>
          <button id="startBtn">Start</button>
          <button id="closeBtn">Close</button>
        </div>
      </div>
      <div>
        <input id="taskInput" type="text"
placeholder="Enter text" />
        <button id="talkBtn">Talk</button>
      </div>
    </div>
    <video id="mediaElement" autoplay></video>

    <script>
      // JavaScript code goes here
    </script>
  </body>
</html>
```

Step 2. Configuration

JavaScript

```
const API_CONFIG = {
  serverUrl: "https://api.heygen.com",
```

```
  token: "YOUR_API_TOKEN"
};

// Global state
let sessionInfo = null;
let room = null;
let mediaStream = null;

// DOM elements
const mediaElement =
document.getElementById("mediaElement");
const taskInput = document.getElementById("taskInput");
```

Step 3. Core Implementation

3.1 Create and Start Session

JavaScript

```
async function createSession() {
  // Create new session
  const response = await fetch(`${API_CONFIG.serverUrl}/
v1/streaming.new`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Authorization": `Bearer ${API_CONFIG.token}`
    },
    body: JSON.stringify({
      version: "v2",
      avatar_id: "YOUR_AVATAR_ID"
    })
  });

  sessionInfo = await response.json();

  // Start streaming
  await fetch(`${API_CONFIG.serverUrl}/v1/
```

```
streaming.start`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Authorization": `Bearer ${API_CONFIG.token}`
    },
    body: JSON.stringify({
      session_id: sessionInfo.session_id
    })
  });

  // Connect to LiveKit room
  room = new LiveKitClient.Room();
  await room.connect(sessionInfo.url,
sessionInfo.access_token);

  // Handle media streams
  room.on(LiveKitClient.RoomEvent.TrackSubscribed,
(track) => {
    if (track.kind === "video" || track.kind === "audio")
{
      mediaStream.addTrack(track.mediaStreamTrack);
      mediaElement.srcObject = mediaStream;
    }
  });


}
```

- The LiveKit CDN version is accessed through the `LivekitClient` global namespace.
- All LiveKit classes and constants must be prefixed with `LivekitClient` (e.g., `LivekitClient.Room`, `LivekitClient.RoomEvent`

3.2 Send Text to Avatar

JavaScript

```
async function sendText(text) {
  await fetch(`${API_CONFIG.serverUrl}/v1/
```

```
streaming.task`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Authorization": `Bearer ${API_CONFIG.token}`
    },
    body: JSON.stringify({
      session_id: sessionInfo.session_id,
      text: text,
      task_type: "talk"  // or "repeat" to make avatar
repeat exactly what you say
    })
  });
}
```

3.3 Close Session

JavaScript

```
async function closeSession() {
  await fetch(`${API_CONFIG.serverUrl}/v1/
streaming.stop`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Authorization": `Bearer ${API_CONFIG.token}`
    },
    body: JSON.stringify({
      session_id: sessionInfo.session_id
    })
  });

  if (room) {
    room.disconnect();
  }

  mediaElement.srcObject = null;
  sessionInfo = null;
  room = null;
  mediaStream = null;
```

```
}
```

Step 4. Event Listeners

JavaScript

```
// Start session
document.querySelector("#startBtn").addEventListener("click", async () => {
  await createSession();
});

// Close session
document.querySelector("#closeBtn").addEventListener("click", closeSession);

// Send text
document.querySelector("#talkBtn").addEventListener("click", () => {
  const text = taskInput.value.trim();
  if (text) {
    sendText(text);
    taskInput.value = "";
  }
});
```

Further Features

1. Task Types

The task endpoint supports different task types:

- `talk`: Avatar processes text through LLM before speaking
- `repeat`: Avatar repeats the exact input text

2. WebSocket Events

Monitor avatar state through WebSocket events:

JavaScript

```
const wsUrl = `wss://api.heygen.com/v1/ws/streaming.chat?
session_id=${sessionId}&session_token=${token}
&silence_response=false`;
const ws = new WebSocket(wsUrl);

ws.addEventListener("message", (event) => {
  const data = JSON.parse(event.data);
  console.log("Event:", data);
});
```

3. LiveKit Room Events

Monitor room state and media tracks:

JavaScript

```
room.on(LivekitClient.RoomEvent.DataReceived, (message)
=> {
  const data = new TextDecoder().decode(message);
  console.log("Room message:", JSON.parse(data));
});
```

System Flow

- Session setup (steps 1-3)
- Video streaming (step 4)
- Avatar interaction loop (step 5)
- Session closure (step 6)

Complete Demo Code

Here's a full working HTML & JS implementation combining all the components with a basic frontend:

HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>HeyGen Streaming API LiveKit (V2)</title>
    <script src="https://cdn.tailwindcss.com"></script>
    <script src="https://cdn.jsdelivr.net/npm/livekit-
client/dist/livekit-client.umd.min.js"></script>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
  </head>

  <body class="bg-gray-100 p-5 font-sans">
    <div class="max-w-3xl mx-auto bg-white p-5 rounded-lg
shadow-md">
      <div class="flex flex-wrap gap-2.5 mb-5">
        <input
          id="avatarID"
          type="text"
          placeholder="Avatar ID"
          class="flex-1 min-w-[200px] p-2 border border-
gray-300 rounded-md"
        />
        <input
          id="voiceID"
          type="text"
          placeholder="Voice ID"
          class="flex-1 min-w-[200px] p-2 border border-
gray-300 rounded-md"
        />
        <button
          id="startBtn"
          class="px-4 py-2 bg-green-500 text-white
rounded-md hover:bg-green-600 transition-colors
disabled:opacity-50 disabled:cursor-not-allowed"
```

```
        >
          Start
        </button>
        <button
          id="closeBtn"
          class="px-4 py-2 bg-red-500 text-white rounded-
md hover:bg-red-600 transition-colors"
        >
          Close
        </button>
      </div>

      <div class="flex flex-wrap gap-2.5 mb-5">
        <input
          id="taskInput"
          type="text"
          placeholder="Enter text for avatar to speak"
          class="flex-1 min-w-[200px] p-2 border border-
gray-300 rounded-md"
        />
        <button
          id="talkBtn"
          class="px-4 py-2 bg-green-500 text-white
rounded-md hover:bg-green-600 transition-colors"
        >
          Talk (LLM)
        </button>
        <button
          id="repeatBtn"
          class="px-4 py-2 bg-blue-500 text-white
rounded-md hover:bg-blue-600 transition-colors"
        >
          Repeat
        </button>
      </div>

      <video
        id="mediaElement"
        class="w-full max-h-[400px] border rounded-lg
my-5"
        autoplay
```

```
      ></video>
      <div
        id="status"
        class="p-2.5 bg-gray-50 border border-gray-300
rounded-md h-[100px] overflow-y-auto font-mono text-sm"
      ></div>
    </div>

    <script>
      // Configuration
      const API_CONFIG = {
        apiKey: "apikey",
        serverUrl: "https://api.heygen.com",
      };

      // Global variables
      let sessionInfo = null;
      let room = null;
      let mediaStream = null;
      let webSocket = null;
      let sessionToken = null;

      // DOM Elements
      const statusElement =
document.getElementById("status");
      const mediaElement =
document.getElementById("mediaElement");
      const avatarID =
document.getElementById("avatarID");
      const voiceID = document.getElementById("voiceID");
      const taskInput =
document.getElementById("taskInput");

      // Helper function to update status
      function updateStatus(message) {
        const timestamp = new
Date().toLocaleTimeString();
        statusElement.innerHTML += `[${timestamp}] $
{message}<br>`;
        statusElement.scrollTop =
statusElement.scrollHeight;
```

```
      }

      // Get session token
      async function getSessionToken() {
        const response = await fetch(
          `${API_CONFIG.serverUrl}/v1/
streaming.create_token`,
          {
            method: "POST",
            headers: {
              "Content-Type": "application/json",
              "X-Api-Key": API_CONFIG.apiKey,
            },
          }
        );

        const data = await response.json();
        sessionToken = data.data.token;
        updateStatus("Session token obtained");
      }

      // Connect WebSocket
      async function connectWebSocket(sessionId) {
        const params = new URLSearchParams({
          session_id: sessionId,
          session_token: sessionToken,
          silence_response: false,
          opening_text: "Hello, how can I help you?",
          stt_language: "en",
        });

        const wsUrl = `wss://${
          new URL(API_CONFIG.serverUrl).hostname
        }/v1/ws/streaming.chat?${params}`;

        webSocket = new WebSocket(wsUrl);

        // Handle WebSocket events
        webSocket.addEventListener("message", (event) =>
{
          const eventData = JSON.parse(event.data);
```

```
        console.log("Raw WebSocket event:", eventData);
      });
    }

    // Create new session
    async function createNewSession() {
      if (!sessionToken) {
        await getSessionToken();
      }

      const response = await fetch(
        `${API_CONFIG.serverUrl}/v1/streaming.new`,
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            Authorization: `Bearer ${sessionToken}`,
          },
          body: JSON.stringify({
            quality: "high",
            avatar_name: avatarID.value,
            voice: {
              voice_id: voiceID.value,
              rate: 2,
            },
            version: "v2",
            video_encoding: "H264",
          }),
        }
      );

      const data = await response.json();
      sessionInfo = data.data;

      // Create LiveKit Room
      room = new LivekitClient.Room({
        adaptiveStream: true,
        dynacast: true,
        videoCaptureDefaults: {
          resolution:
LivekitClient.VideoPresets.h720.resolution,
```

```
          },
        });

        // Handle room events
        room.on(LivekitClient.RoomEvent.DataReceived,
(message) => {
          const data = new TextDecoder().decode(message);
          console.log("Room message:", JSON.parse(data));
        });

        // Handle media streams
        mediaStream = new MediaStream();
        room.on(LivekitClient.RoomEvent.TrackSubscribed,
(track) => {
          if (track.kind === "video" || track.kind ===
"audio") {
            mediaStream.addTrack(track.mediaStreamTrack);
            if (
              mediaStream.getVideoTracks().length > 0 &&
              mediaStream.getAudioTracks().length > 0
            ) {
              mediaElement.srcObject = mediaStream;
              updateStatus("Media stream ready");
            }
          }
        });

        // Handle media stream removal

room.on(LivekitClient.RoomEvent.TrackUnsubscribed,
(track) => {
          const mediaTrack = track.mediaStreamTrack;
          if (mediaTrack) {
            mediaStream.removeTrack(mediaTrack);
          }
        });

        // Handle room connection state changes
        room.on(LivekitClient.RoomEvent.Disconnected,
(reason) => {
          updateStatus(`Room disconnected: ${reason}`);
```

```
        });

        await room.prepareConnection(sessionInfo.url,
sessionInfo.access_token);
        updateStatus("Connection prepared");

        // Connect WebSocket after room preparation
        await connectWebSocket(sessionInfo.session_id);

        updateStatus("Session created successfully");
      }

      // Start streaming session
      async function startStreamingSession() {
        const startResponse = await fetch(
          `${API_CONFIG.serverUrl}/v1/streaming.start`,
          {
            method: "POST",
            headers: {
              "Content-Type": "application/json",
              Authorization: `Bearer ${sessionToken}`,
            },
            body: JSON.stringify({
              session_id: sessionInfo.session_id,
            }),
          }
        );

        // Connect to LiveKit room
        await room.connect(sessionInfo.url,
sessionInfo.access_token);
        updateStatus("Connected to room");

        document.querySelector("#startBtn").disabled =
true;
        updateStatus("Streaming started successfully");
      }

      // Send text to avatar
      async function sendText(text, taskType = "talk") {
        if (!sessionInfo) {
```

```
        updateStatus("No active session");
        return;
      }

      const response = await fetch(
        `${API_CONFIG.serverUrl}/v1/streaming.task`,
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            Authorization: `Bearer ${sessionToken}`,
          },
          body: JSON.stringify({
            session_id: sessionInfo.session_id,
            text: text,
            task_type: taskType,
          }),
        }
      );

      updateStatus(`Sent text (${taskType}): ${text}`);
    }

    // Close session
    async function closeSession() {
      if (!sessionInfo) {
        updateStatus("No active session");
        return;
      }

      const response = await fetch(
        `${API_CONFIG.serverUrl}/v1/streaming.stop`,
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            Authorization: `Bearer ${sessionToken}`,
          },
          body: JSON.stringify({
            session_id: sessionInfo.session_id,
          }),
```

```
          }
        );

        // Close WebSocket
        if (webSocket) {
          webSocket.close();
        }
        // Disconnect from LiveKit room
        if (room) {
          room.disconnect();
        }

        mediaElement.srcObject = null;
        sessionInfo = null;
        room = null;
        mediaStream = null;
        sessionToken = null;
        document.querySelector("#startBtn").disabled =
false;

        updateStatus("Session closed");
      }

      // Event Listeners
      document
        .querySelector("#startBtn")
        .addEventListener("click", async () => {
          await createNewSession();
          await startStreamingSession();
        });
      document
        .querySelector("#closeBtn")
        .addEventListener("click", closeSession);

document.querySelector("#talkBtn").addEventListener("clic
k", () => {
        const text = taskInput.value.trim();
        if (text) {
          sendText(text, "talk");
          taskInput.value = "";
        }
```

```
      });

document.querySelector("#repeatBtn").addEventListener("cl
ick", () => {
        const text = taskInput.value.trim();
        if (text) {
          sendText(text, "repeat");
          taskInput.value = "";
        }
      });
    </script>
  </body>
</html>
```

LiveKit Client SDKs

Here's the list of LiveKit client SDK repositories:

1. client-sdk-flutter: Dart, Flutter Client SDK for LiveKit
2. client-sdk-js: TypeScript, LiveKit browser client SDK (JavaScript)
3. client-sdk-swift: Swift, LiveKit Swift Client SDK for iOS, macOS, tvOS, and visionOS
4. client-sdk-android: Kotlin, LiveKit SDK for Android
5. client-sdk-unity: C#, Official Unity SDK for LiveKit
6. client-sdk-react-native: TypeScript, Official React Native SDK for LiveKit
7. client-sdk-react-native-expo-plugin: TypeScript, Expo plugin for the React Native SDK
8. client-sdk-unity-web: C#, Official LiveKit SDK for Unity WebGL
9. client-sdk-cpp: C++, C++ SDK for LiveKit

You can explore these repos for more detailed information.

Conclusion

The LiveKit-based implementation (v2) of HeyGen's Streaming API provides a streamlined approach to integrating interactive avatars into

web applications. While this guide covers the basics of browser-side implementation, remember that for production Node.js environments, the @heygen/streaming-avatar npm package offers a more comprehensive solution.

Available Resources

- LiveKit Documentation
- HeyGen Streaming API Reference
- HeyGen Streaming Avatar SDK Reference

Support

For additional support or questions:

- Visit HeyGen Documentation: Discussions
- Contact our support team at support@heygen.com
- For API-specific inquiries: api@heygen.com
- Learn more about contacting support in our Help Center

React Native Integration Guide with Streaming API + LiveKit

Suggest Edits

This guide demonstrates how to integrate HeyGen's Streaming API with LiveKit in a React Native/Expo application to create real-time avatar streaming experiences.

Quick Start

Create a new Expo project and install necessary dependencies:

Shell

```
# Create new Expo project
bunx create-expo-app heygen-livekit-demo
cd heygen-livekit-demo

# Install dependencies
bunx expo install @livekit/react-native react-native-
webrtc react-native-safe-area-context
bun install -D @config-plugins/react-native-webrtc
```

Step-by-Step Implementation

1. Project Configuration

Update `app.json` for necessary permissions and plugins:

JSON

```
{
  "expo": {
    "plugins": [
      "@config-plugins/react-native-webrtc",
      [
        "expo-build-properties",
        {
          "ios": {
            "useFrameworks": "static"
          }
        }
      ]
    ],
    "ios": {
      "bitcode": false,
      "infoPlist": {
        "NSCameraUsageDescription": "For streaming
video",
        "NSMicrophoneUsageDescription": "For streaming
```

```
audio"
      }
    },
    "android": {
      "permissions": [
        "android.permission.CAMERA",
        "android.permission.RECORD_AUDIO"
      ]
    }
  }
}
```

2. API Configuration

Create variables for API configuration:

TypeScript

```
const API_CONFIG = {
  serverUrl: "https://api.heygen.com",
  apiKey: "your_api_key_here",
};
```

3. State Management

Set up necessary state variables in your main component:

TypeScript

```
const [wsUrl, setWsUrl] = useState<string>("");
const [token, setToken] = useState<string>("");
const [sessionToken, setSessionToken] =
useState<string>("");
const [sessionId, setSessionId] = useState<string>("");
const [connected, setConnected] = useState(false);
const [text, setText] = useState("");
const [loading, setLoading] = useState(false);
const [speaking, setSpeaking] = useState(false);
```

4. Session Creation Flow

4.1 Create Session Token

TypeScript

```
const getSessionToken = async () => {
  const response = await fetch(
    `${API_CONFIG.serverUrl}/v1/streaming.create_token`,
    {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${API_CONFIG.apiKey}`,
      },
    }
  );
  const data = await response.json();
  return data.data.session_token;
};
```

4.2 Create New Session

TypeScript

```
const createNewSession = async (sessionToken: string) =>
{
  const response = await fetch(`${API_CONFIG.serverUrl}/
v1/streaming.new`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${sessionToken}`,
    },
    body: JSON.stringify({
      quality: "high",
      version: "v2",
      video_encoding: "H264",
    }),
  });
```

```
  const data = await response.json();
  return data.data;
};
```

4.3 Start Streaming Session

TypeScript

```
const startStreamingSession = async (
  sessionId: string,
  sessionToken: string
) => {
  const response = await fetch(`${API_CONFIG.serverUrl}/
v1/streaming.start`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${sessionToken}`,
    },
    body: JSON.stringify({
      session_id: sessionId,
      session_token: sessionToken,
      silence_response: "false",
      stt_language: "en",
    }),
  });
  const data = await response.json();
  return data.data;
};
```

5. LiveKit Room Setup

Embed LiveKit's `LiveKitRoom` in your component:

TypeScript

```
<LiveKitRoom
  serverUrl={wsUrl}
  token={token}
  connect={true}
```

```
  options={{
    adaptiveStream: { pixelDensity: "screen" },
  }}
  audio={false}
  video={false}
>
  <RoomView
    onSendText={sendText}
    text={text}
    onTextChange={setText}
    speaking={speaking}
    onClose={closeSession}
    loading={loading}
  />
</LiveKitRoom>
```

6. Video Track Component

Implement a `RoomView` component to display video streams:

TypeScript

```
const RoomView = ({
  onSendText,
  text,
  onTextChange,
  speaking,
  onClose,
  loading,
}: RoomViewProps) => {
  const tracks = useTracks([Track.Source.Camera],
{ onlySubscribed: true });

  return (
    <SafeAreaView style={styles.container}>
      <KeyboardAvoidingView
        behavior={Platform.OS === "ios" ? "padding" :
"height"}
        style={styles.container}
      >
```

```
        <View style={styles.videoContainer}>
          {tracks.map((track, idx) =>
            isTrackReference(track) ? (
              <VideoTrack
                key={idx}
                style={styles.videoView}
                trackRef={track}
                objectFit="contain"
              />
            ) : null
          )}
        </View>
        {/* Controls */}
      </KeyboardAvoidingView>
    </SafeAreaView>
  );
};
```

7. Send Text to Avatar

TypeScript

```
const sendText = async () => {
  try {
    setSpeaking(true);
    const response = await fetch(`${API_CONFIG.serverUrl}/v1/streaming.task`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${sessionToken}`,
      },
      body: JSON.stringify({
        session_id: sessionId,
        text: text,
        task_type: "talk",
      }),
    });
    const data = await response.json();
    setText("");
```

```
  } catch (error) {
    console.error("Error sending text:", error);
  } finally {
    setSpeaking(false);
  }
};
```

8. Close Session

TypeScript

```
const closeSession = async () => {
  try {
    setLoading(true);
    const response = await fetch(`$
{API_CONFIG.serverUrl}/v1/streaming.stop`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${sessionToken}`,
      },
      body: JSON.stringify({
        session_id: sessionId,
      }),
    });

    // Reset states
    setConnected(false);
    setSessionId("");
    setSessionToken("");
    setWsUrl("");
    setToken("");
    setText("");
    setSpeaking(false);
  } catch (error) {
    console.error("Error closing session:", error);
  } finally {
    setLoading(false);
  }
};
```

Running the App

Shell

```
# Install dependencies
bun install

# Create development build
expo prebuild

# Run on iOS
expo run:ios
# or with physical device
expo run:ios --device

# Run on Android
expo run:android
```

Use React Native Debugger for network inspection

Note: Use physical devices or simulators for WebRTC support. You can't use Expo Go with WebRTC.

System Flow

❓

- Session setup (steps 1-3)
- Video streaming (step 4)
- Avatar interaction loop (step 5)
- Session closure (step 6)

Complete Demo Code

For a full example, check the complete code in `App.tsx`:

App.tsx

```
import { useEffect, useState } from "react";
```

```
import {
  StyleSheet,
  View,
  TextInput,
  Text,
  KeyboardAvoidingView,
  Platform,
  SafeAreaView,
  TouchableOpacity,
  Pressable,
} from "react-native";
import { registerGlobals } from "@livekit/react-native";
import {
  LiveKitRoom,
  AudioSession,
  VideoTrack,
  useTracks,
  isTrackReference,
} from "@livekit/react-native";
import { Track } from "livekit-client";

registerGlobals();

const API_CONFIG = {
  apiKey: "apikey",
  serverUrl: "https://api.heygen.com",
};

export default function App() {
  const [wsUrl, setWsUrl] = useState<string>("");
  const [token, setToken] = useState<string>("");
  const [sessionToken, setSessionToken] =
useState<string>("");
  const [sessionId, setSessionId] = useState<string>("");
  const [connected, setConnected] = useState(false);
  const [text, setText] = useState("");
  const [webSocket, setWebSocket] = useState<WebSocket |
null>(null);
  const [loading, setLoading] = useState(false);
  const [speaking, setSpeaking] = useState(false);
```

```
  // Start audio session on app launch
  useEffect(() => {
    const setupAudio = async () => {
      await AudioSession.startAudioSession();
    };

    setupAudio();
    return () => {
      AudioSession.stopAudioSession();
    };
  }, []);

  const getSessionToken = async () => {
    try {
      const response = await fetch(
        `${API_CONFIG.serverUrl}/v1/
streaming.create_token`,
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            "X-Api-Key": API_CONFIG.apiKey,
          },
        }
      );

      const data = await response.json();
      console.log("Session token obtained",
data.data.token);
      return data.data.token;
    } catch (error) {
      console.error("Error getting session token:",
error);
      throw error;
    }
  };

  const startStreamingSession = async (
    sessionId: string,
    sessionToken: string
  ) => {
```

```
    try {
      console.log("Starting streaming session with:", {
        sessionId,
        sessionToken,
      });
      const startResponse = await fetch(
        `${API_CONFIG.serverUrl}/v1/streaming.start`,
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            Authorization: `Bearer ${sessionToken}`,
          },
          body: JSON.stringify({
            session_id: sessionId,
          }),
        }
      );

      const startData = await startResponse.json();
      console.log("Streaming start response:",
startData);

      if (startData) {
        setConnected(true);
        return true;
      }

      return false;
    } catch (error) {
      console.error("Error starting streaming session:",
error);
      return false;
    }
  };

  const createSession = async () => {
    try {
      setLoading(true);
      // Get new session token
      const newSessionToken = await getSessionToken();
```

```
      setSessionToken(newSessionToken);

      const response = await fetch(`${API_CONFIG.serverUrl}/v1/streaming.new`, {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
          Authorization: `Bearer ${newSessionToken}`,
        },
        body: JSON.stringify({
          quality: "high",
          avatar_name: "",
          voice: {
            voice_id: "",
          },
          version: "v2",
          video_encoding: "H264",
        }),
      });

      const data = await response.json();
      console.log("Streaming new response:", data.data);

      if (data.data) {
        const newSessionId = data.data.session_id;
        // Set all session data
        setSessionId(newSessionId);
        setWsUrl(data.data.url);
        setToken(data.data.access_token);

        // Connect WebSocket
        const params = new URLSearchParams({
          session_id: newSessionId,
          session_token: newSessionToken,
          silence_response: "false",
          // opening_text: "Hello from the mobile app!",
          stt_language: "en",
        });

        const wsUrl = `wss://${
          new URL(API_CONFIG.serverUrl).hostname
```

```
        }/v1/ws/streaming.chat?${params}`;

        const ws = new WebSocket(wsUrl);
        setWebSocket(ws);

        // Start streaming session with the new IDs
        await startStreamingSession(newSessionId,
newSessionToken);
      }
    } catch (error) {
      console.error("Error creating session:", error);
    } finally {
      setLoading(false);
    }
  };

  const sendText = async () => {
    try {
      setSpeaking(true);

      // Send task request
      const response = await fetch(
        `${API_CONFIG.serverUrl}/v1/streaming.task`,
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            Authorization: `Bearer ${sessionToken}`,
          },
          body: JSON.stringify({
            session_id: sessionId,
            text: text,
            task_type: "talk",
          }),
        }
      );

      const data = await response.json();
      console.log("Task response:", data);
      setText(""); // Clear input after sending
    } catch (error) {
```

```
      console.error("Error sending text:", error);
    } finally {
      setSpeaking(false);
    }
  };

  const closeSession = async () => {
    try {
      setLoading(true);
      if (!sessionId || !sessionToken) {
        console.log("No active session");
        return;
      }

      const response = await fetch(
        `${API_CONFIG.serverUrl}/v1/streaming.stop`,
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            Authorization: `Bearer ${sessionToken}`,
          },
          body: JSON.stringify({
            session_id: sessionId,
          }),
        }
      );

      // Close WebSocket
      if (webSocket) {
        webSocket.close();
        setWebSocket(null);
      }

      // Reset all states
      setConnected(false);
      setSessionId("");
      setSessionToken("");
      setWsUrl("");
      setToken("");
      setText("");
```

```
      setSpeaking(false);

      console.log("Session closed successfully");
    } catch (error) {
      console.error("Error closing session:", error);
    } finally {
      setLoading(false);
    }
  };

  if (!connected) {
    return (
      <View style={styles.startContainer}>
        <View style={styles.heroContainer}>
          <Text style={styles.heroTitle}>HeyGen Streaming
API + LiveKit</Text>
          <Text style={styles.heroSubtitle}>React Native/
Expo Demo</Text>
        </View>

        <Pressable
          style={({ pressed }) => [
            styles.startButton,
            { opacity: pressed ? 0.8 : 1 },
          ]}
          onPress={createSession}
          disabled={loading}
        >
          <Text style={styles.startButtonText}>
            {loading ? "Starting..." : "Start Session"}
          </Text>
        </Pressable>
      </View>
    );
  }

  return (
    <LiveKitRoom
      serverUrl={wsUrl}
      token={token}
      connect={true}
```

```
      options={{
        adaptiveStream: { pixelDensity: "screen" },
      }}
      audio={false}
      video={false}
    >
      <RoomView
        onSendText={sendText}
        text={text}
        onTextChange={setText}
        speaking={speaking}
        onClose={closeSession}
        loading={loading}
      />
    </LiveKitRoom>
  );
}

const RoomView = ({
  onSendText,
  text,
  onTextChange,
  speaking,
  onClose,
  loading,
}: {
  onSendText: () => void;
  text: string;
  onTextChange: (text: string) => void;
  speaking: boolean;
  onClose: () => void;
  loading: boolean;
}) => {
  const tracks = useTracks([Track.Source.Camera],
{ onlySubscribed: true });

  return (
    <SafeAreaView style={styles.container}>
      <KeyboardAvoidingView
        behavior={Platform.OS === "ios" ? "padding" :
"height"}
```

```
        style={styles.container}
      >
        <View style={styles.videoContainer}>
          {tracks.map((track, idx) =>
            isTrackReference(track) ? (
              <VideoTrack
                key={idx}
                style={styles.videoView}
                trackRef={track}
                objectFit="contain"
              />
            ) : null
          )}
        </View>
        <TouchableOpacity
          style={[styles.closeButton, loading &&
styles.disabledButton]}
          onPress={onClose}
          disabled={loading}
        >
          <Text style={styles.closeButtonText}>
            {loading ? "Closing..." : "Close Session"}
          </Text>
        </TouchableOpacity>
        <View style={styles.controls}>
          <View style={styles.inputContainer}>
            <TextInput
              style={styles.input}
              placeholder="Enter text for avatar to
speak"
              placeholderTextColor="#666"
              value={text}
              onChangeText={onTextChange}
              editable={!speaking && !loading}
            />
            <TouchableOpacity
              style={[
                styles.sendButton,
                (speaking || !text.trim() || loading) &&
styles.disabledButton,
              ]}
```

```
              onPress={onSendText}
              disabled={speaking || !text.trim() ||
loading}
            >
              <Text style={styles.sendButtonText}>
                {speaking ? "Speaking..." : "Send"}
              </Text>
            </TouchableOpacity>
          </View>
        </View>
      </KeyboardAvoidingView>
    </SafeAreaView>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: "#fff",
  },
  startContainer: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#fff",
    padding: 20,
  },
  heroContainer: {
    alignItems: "center",
    marginBottom: 40,
  },
  heroTitle: {
    fontSize: 22,
    fontWeight: "700",
    color: "#1a73e8",
    marginBottom: 8,
    textAlign: "center",
  },
  heroSubtitle: {
    fontSize: 18,
    color: "#666",
```

```
    fontWeight: "500",
    textAlign: "center",
  },
  startButton: {
    backgroundColor: "#2196F3",
    paddingHorizontal: 32,
    paddingVertical: 16,
    borderRadius: 30,
    elevation: 3,
    shadowColor: "#000",
    shadowOffset: { width: 0, height: 2 },
    shadowOpacity: 0.25,
    shadowRadius: 3.84,
  },
  startButtonText: {
    color: "white",
    fontSize: 18,
    fontWeight: "600",
  },
  videoContainer: {
    flex: 1,
    position: "relative",
  },
  videoView: {
    position: "absolute",
    top: 0,
    left: 0,
    right: 0,
    bottom: 0,
  },
  closeButton: {
    position: "absolute",
    top: 50,
    right: 20,
    backgroundColor: "#ff4444",
    paddingHorizontal: 20,
    paddingVertical: 10,
    borderRadius: 25,
    zIndex: 1,
    elevation: 3,
    shadowColor: "#000",
```

```
    shadowOffset: { width: 0, height: 2 },
    shadowOpacity: 0.25,
    shadowRadius: 3.84,
  },
  closeButtonText: {
    color: "white",
    fontSize: 16,
    fontWeight: "600",
  },
  controls: {
    width: "100%",
    padding: 20,
    borderTopWidth: 1,
    borderColor: "#333",
    // backgroundColor: "rgba(0, 0, 0, 0.75)",
  },
  inputContainer: {
    flexDirection: "row",
    alignItems: "center",
    gap: 10,
  },
  input: {
    flex: 1,
    height: 50,
    borderColor: "#333",
    borderWidth: 1,
    paddingHorizontal: 15,
    borderRadius: 25,
    backgroundColor: "rgba(255, 255, 255, 0.9)",
    fontSize: 16,
    color: "#000",
  },
  sendButton: {
    backgroundColor: "#2196F3",
    paddingHorizontal: 20,
    paddingVertical: 12,
    borderRadius: 25,
    elevation: 3,
    shadowColor: "#000",
    shadowOffset: { width: 0, height: 2 },
    shadowOpacity: 0.25,
```

```
    shadowRadius: 3.84,
  },
  sendButtonText: {
    color: "white",
    fontSize: 16,
    fontWeight: "600",
  },
  disabledButton: {
    opacity: 0.5,
  },
});
```

You can find the complete demo repository here.

Resources

- HeyGen API Documentation
- LiveKit React Native Client SDK
- Expo Documentation

Conclusion

You've now integrated HeyGen's Streaming API with LiveKit in your React Native project. This setup enables real-time interactive avatar experiences with minimal effort.

Streaming Avatar SDK

Suggest Edits

Overview

The `@heygen/streaming-avatar` allows developers to integrate real-time, AI-powered avatars into their applications. Using HeyGen's

platform, you can control avatars through live streaming sessions, allowing them to speak, respond to commands, and interact with users via WebSockets. This SDK simplifies the process of connecting to HeyGen's streaming services and handling avatar interactions programmatically.

Key Features

- Real-time Streaming: Connect and control avatars in live sessions using WebSockets for seamless communication.
- Text-to-Speech Integration: Send text commands to avatars, allowing them to speak in real-time with customizable voices.
- Event-Driven Architecture: Capture key events, such as when the avatar starts or stops speaking, and use them to trigger updates in your application.
- Session Management: Easily create, manage, and terminate avatar sessions programmatically.
- Choose Avatars: Choose avatars, adjust their quality, and customize their voice settings.

For a practical demonstration of how to use the Streaming Avatar SDK, check out our Next.js demo. This demo showcases the capabilities of the SDK in a Next.js environment, providing a comprehensive example of avatar integration.

Getting Started

To get started with the **Streaming Avatar SDK**, install the TypeScript package from npm and add your HeyGen API Token to your project.

Installation

To install the Streaming Avatar SDK, use npm:

Shell

```
npm install @heygen/streaming-avatar livekit-client
```

Basic Usage

Here's a simple example of how to create a streaming session and send a task to an avatar:

TypeScript

```
import { StreamingAvatar } from '@heygen/streaming-avatar';

const avatar = new StreamingAvatar({ token: 'your-access-token' });

const startSession = async () => {
  const sessionData = await avatar.createStartAvatar({
    avatarName: 'MyAvatar',
    quality: 'high',
  });

  console.log('Session started:', sessionData.session_id);

  await avatar.speak({
    sessionId: sessionData.session_id,
    text: 'Hello, world!',
    task_type: TaskType.REPEAT
  });
};

startSession();
```

Managing Sessions

The SDK provides comprehensive session management features, including starting, stopping, and controlling avatar sessions. You can handle session details, such as checking if the session is active, tracking the session duration, and managing multiple avatars simultaneously.

Event Handling

The SDK uses an event-driven architecture to handle various avatar interactions. You can listen for key events such as when an avatar starts talking, stops talking, or when the stream is ready for display. This allows you to dynamically update your application based on real-time avatar behavior.

SDK API Reference

For API reference, you can refer to this page: Streaming API SDK Reference.

Conclusion

The **Streaming Avatar SDK** empowers developers to integrate real-time, AI-driven avatars into their applications with ease. By leveraging HeyGen's platform, you can create interactive, engaging experiences with customizable avatars, real-time text-to-speech, and seamless session management.

Streaming Avatar SDK API Reference

Suggest Edits

The `@heygen/streaming-avatar` package provides a TypeScript SDK

for interacting with HeyGen's streaming avatar service. For detailed information about the available methods, interfaces, enums, and event handling, refer to the API reference provided below.

Classes

`StreamingAvatar`

This class is the core of the SDK, responsible for managing avatar streaming sessions, including session creation, controlling avatars, and handling real-time communication.

TypeScript
TypeScript

```
const avatar = new StreamingAvatar({ token: "access-token" });
```

Interfaces

`StreamingAvatarApiConfig`

Configuration object for initializing `StreamingAvatar`.

Property	Type	Description
`token`	`string`	Authentication token for the session. **Please note this is not your HeyGen API key.** You can retrieve this 'Session Token' by calling the create_token endpoint: https://docs.heygen.com/reference/create-session-token

`basePath`	`string`	Base API URL (optional, defaults to `https://api.heygen.com`).

TypeScript

```
const config: StreamingAvatarApiConfig = {
  token: "access-token",
};
```

StartAvatarRequest↗

Request payload to initiate a new avatar streaming session.

Property	Type	Description
`avatarName`	`string`	Interactive Avatar ID for the session. (default, `default`)
`quality`	`AvatarQuality`	(Optional) The desired quality level of the avatar stream.
`voice`	`VoiceSetting`	(Optional) Voice settings for the avatar.
`knowledgeId`	`string`	(Optional) Knowledge base ID for the avatar's knowledge / prompt. Retrieve from labs.heygen.com.
`knowledgeBase`	`string`	(Optional) This is used as a custom 'system prompt' for the LLM that powers the Avatar's responses when using the Talk task type in the Speak request method.
`disableIdleTimeout`	`boolean`	(Optional) Controls the avatar's session timeout behavior. When true, prevents automatic session termination during periods of inactivity.

TypeScript

```
const startRequest: StartAvatarRequest = {
  quality: AvatarQuality.High,
  avatarName: avatarId,
  knowledgeId: knowledgeId,
```

```
  // knowledgeBase: knowledgeBase,
  voice: {
    voiceId: voiceId,
    rate: 1.5, // 0.5 ~ 1.5
    emotion: VoiceEmotion.EXCITED,
  },
  language: language,
  disableIdleTimeout: true
};
```

StartAvatarResponse

The response received when an avatar session is successfully started.

Property	Type	Description
`session_id`	`string`	The unique ID of the streaming session.
`access_token`	`string`	Token to authenticate further interactions.
`url`	`string`	WebSocket URL for establishing the streaming session.
`is_paid`	`boolean`	Indicates whether the session is under a paid plan.
`session_duration_limit`	`number`	Maximum allowed duration for the session in seconds.

Response

```
{
  "session_id": "eba59f0d-71f5-11ef-b8af-d2e5560124bc",
  "sdp": null,
  "access_token": "eyJhbGc...",
  "url": "wss://heygen-feapbkvq.livekit.cloud",
  "ice_servers": null,
  "ice_servers2": null,
  "is_paid": true,
  "session_duration_limit": 600
}
```

SpeakRequest↗

Request payload for sending a speaking command to the avatar.

Property	Type	Description
text	string	The textual content the avatar will vocalize and synchronize with its movements.
taskType	string	Defines the speaking behavior mode. Options include TaskType.TALK and TaskType.REPEAT.
taskMode	string	Specifies synchronization strategy. SYNC blocks further actions until speaking completes, ASYNC allows concurrent processing.

TypeScript

```
const speakRequest: SpeakRequest = {
  text: "Hello, there!",
  task_type: TaskType.REPEAT
};
```

Methods

createStartAvatar

Starts a new avatar session using the provided configuration and returns session information.

TypeScript
TypeScript

```
createStartAvatar(requestData: StartAvatarRequest): Promise<any>
```

`startVoiceChat`

Starts a voice chat within the active avatar session. You can optionally enable or disable silence prompts during the chat by setting the `useSilencePrompt` flag

TypeScript
TypeScript

```
startVoiceChat(requestData: { useSilencePrompt?: boolean } = {}): Promise<any>
```

Property	Type	Description
`useSilencePrompt`	`boolean`	(Optional) Controls automatic conversational prompts during periods of user inactivity. Enables fallback conversational strategies.

`closeVoiceChat`

Ends the active voice chat session within the avatar interaction.

TypeScript
TypeScript

```
closeVoiceChat(): Promise<any>
```

`newSession`

Creates and starts a new session using the provided `StartAvatarRequest` data, returning detailed session information such as the session ID and other metadata.

TypeScript
TypeScript

```
newSession(requestData: StartAvatarRequest): Promise<StartAvatarResponse>
```

`startSession`

Starts an existing avatar session by using the previously stored session ID or configuration from a `StartAvatarRequest`.

TypeScript
TypeScript

```
startSession(): Promise<any>
```

`speak`

Sends a command to the avatar to speak the provided text. Additional parameters like `task_type` allow for more advanced control, like repeating or talking.

TypeScript
TypeScript

```
speak(requestData: SpeakRequest): Promise<any>
```

`startListening`

Activates the avatar's listening mode, allowing it to process incoming audio or messages from the user.

TypeScript
TypeScript

```
startListening(): Promise<any>
```

`stopListening`

Stops the avatar from listening to incoming audio or messages.

TypeScript
TypeScript

```
stopListening(): Promise<any>
```

`interrupt`

Interrupts the current speaking task.

```TypeScript
interrupt(): Promise<any>
```

`stopAvatar`

Stops the avatar session.

```TypeScript
stopAvatar(): Promise<any>
```

`on`

Registers an event listener for specific streaming events.

```TypeScript
on(eventType: string, listener: EventHandler): this
```

off

Unregisters an event listener.

```TypeScript
off(eventType: string, listener: EventHandler): this
```

Types and Enums

`AvatarQuality`

Defines the quality settings for the avatar.

- **High**: `'high'` - 2000kbps and 720p.
- **Medium**: `'medium'` - 1000kbps and 480p.
- **Low**: `'low'` - 500kbps and 360p.

`VoiceEmotion`

- **`EXCITED`**: Excited voice emotion.
- **`SERIOUS`**: Serious voice emotion.
- **`FRIENDLY`**: Friendly voice emotion.
- **`SOOTHING`**: Soothing voice emotion.
- **`BROADCASTER`**: Broadcaster voice emotion.

`TaskType`

- **`TALK`**: Avatar will talk in response to the `Text` sent in tasks of this type; the response will be provided by HeyGen's connection to GPT-4o mini, and influenced by the `KnowledgeID` or `KnowledgeBase`that were provided when calling the `StartAvatarRequest` method.
- **`REPEAT`**: Avatar will simply repeat the `Text` sent in tasks of this type; this task type is commonly used by developers who process a user's input independently, via an LLM of their choosing, and send the LLM's response as a **Repeat** task for the Avatar to say.

`StreamingEvents`

Enumerates the event types for streaming. See Event Handling for details.

- **`AVATAR_START_TALKING`**: Emitted when the avatar starts speaking.
- **`AVATAR_STOP_TALKING`**: Emitted when the avatar stops speaking.
- **`AVATAR_TALKING_MESSAGE`**: Triggered when the avatar sends a speaking message.
- **`AVATAR_END_MESSAGE`**: Triggered when the avatar finishes sending

messages.
- **`USER_TALKING_MESSAGE`**: Emitted when the user sends a speaking message.
- **`USER_END_MESSAGE`**: Triggered when the user finishes sending messages.
- **`USER_START`**: Indicates when the user starts interacting.
- **`USER_STOP`**: Indicates when the user stops interacting.
- **`USER_SILENCE`**: Indicates when the user is silent.
- **`STREAM_READY`**: Indicates that the stream is ready for display.
- **`STREAM_DISCONNECTED`**: Triggered when the stream disconnects.

Event Handling

The SDK emits a variety of events during a streaming session, which can be captured to update the UI or trigger additional logic. Use the `on` and `off` methods to manage event listeners.

`AVATAR_START_TALKING`

This event is emitted when the avatar begins speaking.

TypeScript

```
avatar.on(StreamingEvents.AVATAR_START_TALKING, (event)
=> {
  console.log('Avatar has started talking:', event);
  // You can update the UI to reflect that the avatar is
talking
});
```

`AVATAR_STOP_TALKING`

TypeScript

```
avatar.on(StreamingEvents.AVATAR_STOP_TALKING, (event) =>
{
```

```
  console.log('Avatar has stopped talking:', event);
  // You can reset the UI to indicate the avatar has
stopped speaking
});
```

AVATAR_TALKING_MESSAGE

Fired when the avatar sends a message while talking. This event can be useful for real-time updates on what the avatar is currently saying.

TypeScript

```
avatar.on(StreamingEvents.AVATAR_TALKING_MESSAGE,
(message) => {
  console.log('Avatar talking message:', message);
  // You can display the message in the UI
});
```

AVATAR_END_MESSAGE

Fired when the avatar sends the final message before ending its speech.

TypeScript

```
avatar.on(StreamingEvents.AVATAR_END_MESSAGE, (message)
=> {
  console.log('Avatar end message:', message);
  // Handle the end of the avatar's message, e.g.,
indicate the end of the conversation
});
```

USER_TALKING_MESSAGE

Fired when the user sends a message to the avatar.

TypeScript

```
avatar.on(StreamingEvents.USER_TALKING_MESSAGE, (message)
=> {
```

```
  console.log('User talking message:', message);
  // Handle the user's message input to the avatar
});
```

USER_END_MESSAGE

Fired when the user has finished sending their message to the avatar.

TypeScript

```
avatar.on(StreamingEvents.USER_END_MESSAGE, (message) =>
{
  console.log('User end message:', message);
  // Handle the end of the user's message, e.g., process
the user's response
});
```

USER_START

Fired when the user has finished sending their message to the avatar.

TypeScript

```
avatar.on(StreamingEvents.USER_START, (event) => {
  console.log('User has started interaction:', event);
  // Handle the start of the user's interaction, such as
activating a listening indicator
});
```

USER_STOP

Triggered when the user stops interacting or speaking with the avatar.

TypeScript

```
avatar.on(StreamingEvents.USER_STOP, (event) => {
  console.log('User has stopped interaction:', event);
  // Handle the end of the user's interaction, such as
deactivating a listening indicator
});
```

`USER_SILENCE`

Triggered when the user is silent for a certain period.

TypeScript

```
avatar.on(StreamingEvents.USER_SILENCE, () => {
  console.log('User is silent');
});
```

`STREAM_READY`

Fired when the avatar's streaming session is ready.

TypeScript

```
avatar.on(StreamingEvents.STREAM_READY, (event) => {
  console.log('Stream is ready:', event.detail);
  // Use event.detail to attach the media stream to a
video element, for example
});
```

`STREAM_DISCONNECTED`

Triggered when the streaming connection is lost or intentionally disconnected.

TypeScript

```
avatar.on(StreamingEvents.STREAM_DISCONNECTED, () => {
  console.log('Stream has been disconnected');
  // Handle the disconnection, e.g., clean up the UI or
try to reconnect the session
});
```

Error Handling

Always handle errors gracefully when dealing with asynchronous requests to avoid disruptions in the user experience.

TypeScript

```
try {
  await avatar.speak({ text: 'Hello!' });
} catch (error) {
  console.error('Error sending speak command:', error);
}
```

Conclusion

This reference offers a comprehensive overview of HeyGen's streaming avatar SDK, complete with examples and descriptions of methods, events, and configuration options.

Demo: Create a Vite Project with Streaming SDK

Suggest Edits

In this demo, we will create a basic Vite project that integrates the `@heygen/streaming-avatar` SDK to showcase interactive avatars.

Setup Vite Project

1. Create a new Vite project:

Shell

```
npm create vite@latest streaming-avatar-demo -- --
template vanilla-ts
cd streaming-avatar-demo
```

2. Install dependencies:

Shell

```
npm install @heygen/streaming-avatar livekit-client
```

Step 2: Create Basic Frontend Structure

Edit `index.html` to include a video element and buttons for interaction:

index.html

```
<!DOCTYPE html>
<html lang="en" data-theme="light">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <link rel="icon" href="/vite.svg" type="image/
svg+xml" />
    <link
      rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/@picocss/pico@2/
css/pico.min.css"
    />
    <title>Interactive Avatar Demo</title>
  </head>

  <body>
    <main
      class="container"
      style="display: flex; flex-direction: column;
align-items: center"
    >
      <h1>Interactive Avatar Demo (Vite + TypeScript)</
h1>

      <!-- Video Section -->
      <article style="width: fit-content">
        <video id="avatarVideo" autoplay playsinline></
video>
      </article>
```

```
      <!-- Controls Section -->
      <section>
        <section role="group">
          <button id="startSession">Start Session</
button>
          <button id="endSession" disabled>End Session</
button>
        </section>

        <section role="group">
          <input
            type="text"
            id="userInput"
            placeholder="Type something to talk to the
avatar..."
            aria-label="User input"
          />
          <button id="speakButton" role="button">Speak</
button>
        </section>
      </section>
    </main>

    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

Step 3: Implement the Streaming Avatar SDK Logic

Create a `main.ts` file in the `src` directory:

main.ts

```
import StreamingAvatar, {
  AvatarQuality,
  StreamingEvents,
} from "@heygen/streaming-avatar";
```

```
// DOM elements
const videoElement =
document.getElementById("avatarVideo") as
HTMLVideoElement;
const startButton = document.getElementById(
  "startSession"
) as HTMLButtonElement;
const endButton = document.getElementById("endSession")
as HTMLButtonElement;
const speakButton =
document.getElementById("speakButton") as
HTMLButtonElement;
const userInput = document.getElementById("userInput") as
HTMLInputElement;

let avatar: StreamingAvatar | null = null;
let sessionData: any = null;

// Helper function to fetch access token
async function fetchAccessToken(): Promise<string> {
  const apiKey = import.meta.env.VITE_HEYGEN_API_KEY;
  const response = await fetch(
    "https://api.heygen.com/v1/streaming.create_token",
    {
      method: "POST",
      headers: { "x-api-key": apiKey },
    }
  );

  const { data } = await response.json();
  return data.token;
}

// Initialize streaming avatar session
async function initializeAvatarSession() {
  const token = await fetchAccessToken();
  avatar = new StreamingAvatar({ token });

  sessionData = await avatar.createStartAvatar({
    quality: AvatarQuality.High,
    avatarName: "default",
```

```
  });

  console.log("Session data:", sessionData);

  // Enable end button and disable start button
  endButton.disabled = false;
  startButton.disabled = true;

  avatar.on(StreamingEvents.STREAM_READY,
handleStreamReady);
  avatar.on(StreamingEvents.STREAM_DISCONNECTED,
handleStreamDisconnected);
}

// Handle when avatar stream is ready
function handleStreamReady(event: any) {
  if (event.detail && videoElement) {
    videoElement.srcObject = event.detail;
    videoElement.onloadedmetadata = () => {
      videoElement.play().catch(console.error);
    };
  } else {
    console.error("Stream is not available");
  }
}

// Handle stream disconnection
function handleStreamDisconnected() {
  console.log("Stream disconnected");
  if (videoElement) {
    videoElement.srcObject = null;
  }

  // Enable start button and disable end button
  startButton.disabled = false;
  endButton.disabled = true;
}

// End the avatar session
async function terminateAvatarSession() {
  if (!avatar || !sessionData) return;
```

```
  await avatar.stopAvatar();
  videoElement.srcObject = null;
  avatar = null;
}

// Handle speaking event
async function handleSpeak() {
  if (avatar && userInput.value) {
    await avatar.speak({
      text: userInput.value,
    });
    userInput.value = ""; // Clear input after speaking
  }
}

// Event listeners for buttons
startButton.addEventListener("click",
initializeAvatarSession);
endButton.addEventListener("click",
terminateAvatarSession);
speakButton.addEventListener("click", handleSpeak);
```

Note: Add your HeyGen API key to your .env file in the root of your Vite project:

.env

```
VITE_HEYGEN_API_KEY=your_api_key_here
```

Step 4: Run the Project

Start the Vite development server:

Shell

```
npm run dev
```

Open your browser and navigate to `http://localhost:5173` to see the demo in action.

Best Practice

To enhance security and avoid exposing your HeyGen API token in the frontend, move the `fetchAccessToken` function to a backend service.

Details

1

2

Conclusion

You now have a basic Vite TypeScript demo using the Streaming Avatar SDK. You can expand this by adding more features such as handling different events or customizing the avatar interactions.

Integrating OpenAI Assistant with Streaming SDK

Suggest Edits

The OpenAI Assistants API enables you to create AI-powered assistants directly within your applications. An assistant can follow specific instructions and use models, tools, and files to respond to user queries effectively.

In this example, we will create a basic assistant to generate responses and integrate it with an avatar. To achieve this, we will build on the Vite + SDK project created in the previous step.

Step 1: Install OpenAI Package

Add the OpenAI library to your project:

Bash

```
npm install openai
```

Step 2: Create `openai-assistant.ts`

This file contains a class to interact with OpenAI's API. Place it in the `src` folder.

openai-assistant.ts

```
import OpenAI from "openai";
```

```
export class OpenAIAssistant {
  private client: OpenAI;
  private assistant: any;
  private thread: any;

  constructor(apiKey: string) {
    this.client = new OpenAI({ apiKey,
dangerouslyAllowBrowser: true });
  }

  async initialize(
    instructions: string = `You are an English tutor.
Help students improve their language skills by:
    - Correcting mistakes in grammar and vocabulary
    - Explaining concepts with examples
    - Engaging in conversation practice
    - Providing learning suggestions
    Be friendly, adapt to student's level, and always
give concise answers.`
  ) {
    // Create an assistant
    this.assistant = await
this.client.beta.assistants.create({
      name: "English Tutor Assistant",
      instructions,
      tools: [],
      model: "gpt-4-turbo-preview",
    });

    // Create a thread
    this.thread = await
this.client.beta.threads.create();
  }

  async getResponse(userMessage: string): Promise<string>
{
    if (!this.assistant || !this.thread) {
      throw new Error("Assistant not initialized. Call
initialize() first.");
    }
```

```
    // Add user message to thread
    await
this.client.beta.threads.messages.create(this.thread.id,
{
      role: "user",
      content: userMessage,
    });

    // Create and run the assistant
    const run = await
this.client.beta.threads.runs.createAndPoll(
      this.thread.id,
      { assistant_id: this.assistant.id }
    );

    if (run.status === "completed") {
      // Get the assistant's response
      const messages = await
this.client.beta.threads.messages.list(
        this.thread.id
      );

      // Get the latest assistant message
      const lastMessage = messages.data.filter(
        (msg) => msg.role === "assistant"
      )[0];

      if (lastMessage && lastMessage.content[0].type ===
"text") {
        return lastMessage.content[0].text.value;
      }
    }

    return "Sorry, I couldn't process your request.";
  }
}
```

Using `dangerouslyAllowBrowser: true` allows direct API calls from the browser.

Best Practice: For security, perform these calls on your backend instead of exposing the API key in the browser. This implementation is kept simple for demonstration.

Step 3: Update `main.ts`

1 **Import and Declare the Assistant:**

main.ts

```
import { OpenAIAssistant } from "./openai-assistant";

let openaiAssistant: OpenAIAssistant | null = null;
```

2 **Update `initializeAvatarSession`:**

Add the OpenAI assistant initialization:

main.ts

```
// Initialize streaming avatar session
async function initializeAvatarSession() {
  // Disable start button immediately to prevent double
clicks
  startButton.disabled = true;

  try {
    const token = await fetchAccessToken();
    avatar = new StreamingAvatar({ token });

    // Initialize OpenAI Assistant
    const openaiApiKey =
import.meta.env.VITE_OPENAI_API_KEY;
    openaiAssistant = new OpenAIAssistant(openaiApiKey);
    await openaiAssistant.initialize();

    sessionData = await avatar.createStartAvatar({
      quality: AvatarQuality.Medium,
      avatarName: "Wayne_20240711",
      language: "English",
    });
```

```
    console.log("Session data:", sessionData);

    // Enable end button
    endButton.disabled = false;

    avatar.on(StreamingEvents.STREAM_READY,
handleStreamReady);
    avatar.on(StreamingEvents.STREAM_DISCONNECTED,
handleStreamDisconnected);
  } catch (error) {
    console.error("Failed to initialize avatar session:",
error);
    // Re-enable start button if initialization fails
    startButton.disabled = false;
  }
}
```

3 **Update `handleSpeak`:**
Passes user input to OpenAI, retrieves a response, and instructs the avatar to speak the response aloud.

main.ts

```
// Handle speaking event
async function handleSpeak() {
  if (avatar && openaiAssistant && userInput.value) {
    try {
      const response = await
openaiAssistant.getResponse(userInput.value);
      await avatar.speak({
        text: response,
        taskType: TaskType.REPEAT,
      });
    } catch (error) {
      console.error("Error getting response:", error);
    }
    userInput.value = ""; // Clear input after speaking
  }
}
```

Step 4: Update Environment Variables

Add your OpenAI API key to the `.env` file:

```
VITE_OPENAI_API_KEY=your-key
```

How It Works

1. **User Input**: The user enters a query.
2. **OpenAI Interaction**: The query is sent to OpenAI's Assistant via `OpenAIAssistant.getResponse`.
3. **Avatar Response**: The response from OpenAI is passed to the HeyGen avatar via `avatar.speak`.

Example Workflow

1. User enters a question: *"What is the difference between affect and effect?"*
2. OpenAI processes and responds.
3. The avatar speaks the response aloud.

Conclusion

With these updates, your HeyGen avatar can now utilize OpenAI's Assistant API for interactive, intelligent conversations. While the example prioritizes simplicity, implementing a secure backend for API calls is highly recommended for production.

Adding Speech-to-Text Integration to Demo Project

Suggest Edits

This guide will walk you through the process of adding speech-to-text capabilities to your existing Interactive Avatar Vite demo project. This feature allows users to speak into their microphone, have their speech

converted to text using OpenAI's Whisper API, and then have the avatar speak the transcribed text.

❓

Flow

❓

1. User's voice is captured and recorded using the MediaRecorder API
2. Audio is sent to OpenAI's Whisper API for transcription
3. Transcribed text is passed to the avatar's speak method
4. Avatar processes the text using its built-in LLM and responds

Note: If you need custom language processing or specific behaviors, you can add your own LLM step between transcription and avatar speech (shown as dotted line in the diagram).

Prerequisites

1. Existing Heygen Interactive Avatar Vite demo project.
2. OpenAI API key (Get one from OpenAI Platform).
3. Heygen API key (Already in your project).

Step 1: Environment Setup

1. Add your OpenAI API key to your `.env` file:

```env
VITE_OPENAI_API_KEY=your_openai_api_key_here
```

Step 2: Create Audio Handler

1. Create a new file `src/audio-handler.ts` with the following content:

TypeScript

```typescript
export class AudioRecorder {
    private mediaRecorder: MediaRecorder | null = null;
    private audioChunks: Blob[] = [];
    private isRecording = false;

    constructor(
        private onStatusChange: (status: string) => void,
        private onTranscriptionComplete: (text: string)
=> void
    ) {}

    async startRecording() {
        try {
            console.log('Requesting microphone
access...');
            const stream = await
navigator.mediaDevices.getUserMedia({ audio: true });
            console.log('Microphone access granted');

            this.mediaRecorder = new
MediaRecorder(stream);
            this.audioChunks = [];
            this.isRecording = true;

            this.mediaRecorder.ondataavailable = (event)
=> {
                if (event.data.size > 0) {
                    console.log('Received audio chunk:',
event.data.size, 'bytes');
                    this.audioChunks.push(event.data);
                }
            };

            this.mediaRecorder.onstop = async () => {
                console.log('Recording stopped,
processing audio...');
                const audioBlob = new
Blob(this.audioChunks, { type: 'audio/webm' });
                console.log('Audio blob size:',
```

```
audioBlob.size, 'bytes');
                await this.sendToWhisper(audioBlob);
            };

            this.mediaRecorder.start(1000); // Collect
data every second
            console.log('Started recording');
            this.onStatusChange('Recording... Speak
now');
        } catch (error) {
            console.error('Error starting recording:',
error);
            this.onStatusChange('Error: ' + (error as
Error).message);
        }
    }

    stopRecording() {
        if (this.mediaRecorder && this.isRecording) {
            console.log('Stopping recording...');
            this.mediaRecorder.stop();
            this.isRecording = false;
            this.onStatusChange('Processing audio...');

            // Stop all tracks in the stream
            const stream = this.mediaRecorder.stream;
            stream.getTracks().forEach(track =>
track.stop());
        }
    }

    private async sendToWhisper(audioBlob: Blob) {
        try {
            console.log('Sending audio to Whisper
API...');
            const formData = new FormData();
            formData.append('file', audioBlob,
'audio.webm');
            formData.append('model', 'whisper-1');

            const response = await fetch('https://
```

```
api.openai.com/v1/audio/transcriptions', {
                method: 'POST',
                headers: {
                    'Authorization': `Bearer $
{import.meta.env.VITE_OPENAI_API_KEY}`,
                },
                body: formData
            });

            if (!response.ok) {
                const errorText = await response.text();
                throw new Error(`HTTP error! status: $
{response.status}, details: ${errorText}`);
            }

            const data = await response.json();
            console.log('Received transcription:',
data.text);
            this.onStatusChange('');
            this.onTranscriptionComplete(data.text);
        } catch (error) {
            console.error('Error transcribing audio:',
error);
            this.onStatusChange('Error: Failed to
transcribe audio');
        }
    }
}
```

Step 3: Update HTML

1 Add the record button and status display to your `index.html`:

HTML

```
<!-- Add this after your existing buttons -->
<section role="group">
    <button id="recordButton">Start Recording</button>
</section>

<div>
```

```
    <p id="recordingStatus"></p>
</div>
```

Step 4: Update Main TypeScript File

1 Update your `src/main.ts` to include the recording functionality:

TypeScript

```typescript
// Add these imports at the top of your file
import { AudioRecorder } from './audio-handler';

// Add these DOM elements with your existing ones
const recordButton =
document.getElementById("recordButton") as
HTMLButtonElement;
const recordingStatus =
document.getElementById("recordingStatus") as
HTMLParagraphElement;

// Add these variables with your existing ones
let audioRecorder: AudioRecorder | null = null;
let isRecording = false;

// Add this function to handle speaking text
async function speakText(text: string) {
    if (avatar && text) {
        await avatar.speak({
            text: text,
        });
    }
}

// Add these functions for audio recording
function initializeAudioRecorder() {
    audioRecorder = new AudioRecorder(
        (status) => {
            recordingStatus.textContent = status;
        },
        (text) => {
            speakText(text);
```

```
        }
    );
}

async function toggleRecording() {
    if (!audioRecorder) {
        initializeAudioRecorder();
    }

    if (!isRecording) {
        recordButton.textContent = "Stop Recording";
        await audioRecorder?.startRecording();
        isRecording = true;
    } else {
        recordButton.textContent = "Start Recording";
        audioRecorder?.stopRecording();
        isRecording = false;
    }
}

// Add this event listener with your existing ones
recordButton.addEventListener("click", toggleRecording);
```

This implementation uses the following Web APIs: MediaRecorder API, getUserMedia API, Web Audio API.

Step 5: Test the Implementation

1. Start your development server:

Bash

```
bun dev
```

2. Open your browser and test the functionality:
   - Click "Start Session" to initialize the avatar
   - Click "Start Recording" to begin recording your voice
   - Speak your message
   - Click "Stop Recording" to stop recording
   - Wait for the transcription and watch the avatar speak your message

For any issues or questions, please refer to:

- OpenAI Whisper API Documentation
- MediaRecorder API Documentation
- Heygen API Documentation

Conclusion

You've successfully added speech-to-text capabilities to your Heygen Streaming Avatar demo. This enhancement transforms your application from a text-based interface to an interactive voice-enabled experience. The integration of OpenAI's Whisper API provides accurate speech recognition across multiple languages, while the modular structure of the code allows for easy maintenance and future improvements.

Adding Built-in Voice Chat Integration to Demo Project

Suggest Edits

Please note: The built-in TTS/voice mode is tightly integrated with HeyGen's internal LLM/knowledge base. Currently, custom avatar speech input is not supported when voice chat is enabled. If you want to provide custom input, you will need to integrate your own STT solution. ❓

This guide walks you through the next step of enhancing your existing Vite demo project using the HeyGen SDK by integrating built-in voice chat functionality. Building on the initial setup, we'll show you how to enable voice mode for real-time interaction with the avatar, allowing users to switch seamlessly between text and voice input.

1. Update `index.html` Structure

Add buttons to switch between text and voice modes, and include a

section to manage voice controls.

HTML

```
<!-- Add mode switching buttons -->
<div class="chat-modes" role="group">
  <button id="textModeBtn" class="active">Text Mode</
button>
  <button id="voiceModeBtn" disabled>Voice Mode</button>
</div>

<!-- Add voice mode controls section -->
<section id="voiceModeControls" role="group"
style="display: none">
  <div id="voiceStatus"></div>
</section>
```

2. Update `main.ts` Code

Add New DOM References

Update the DOM to include references for the new buttons and controls.

TypeScript

```
// Add these DOM elements
const textModeBtn =
document.getElementById("textModeBtn") as
HTMLButtonElement;
const voiceModeBtn =
document.getElementById("voiceModeBtn") as
HTMLButtonElement;
const textModeControls =
document.getElementById("textModeControls") as
HTMLElement;
const voiceModeControls =
document.getElementById("voiceModeControls") as
HTMLElement;
const voiceStatus =
```

```
document.getElementById("voiceStatus") as HTMLElement;

// Add mode tracking
let currentMode: "text" | "voice" = "text";
```

Update Avatar Initialization

Modify avatar initialization to handle voice chat events and display appropriate status updates.

TypeScript

```
async function initializeAvatarSession() {
  const token = await fetchAccessToken();
  avatar = new StreamingAvatar({ token });

  sessionData = await avatar.createStartAvatar({
    quality: AvatarQuality.High,
    avatarName: "default",
    disableIdleTimeout: true,
    language: "en",  // Use correct language code
  });

  // Add voice chat event listeners
  avatar.on(StreamingEvents.USER_START, () => {
    voiceStatus.textContent = "Listening...";
  });
  avatar.on(StreamingEvents.USER_STOP, () => {
    voiceStatus.textContent = "Processing...";
  });
  avatar.on(StreamingEvents.AVATAR_START_TALKING, () => {
    voiceStatus.textContent = "Avatar is speaking...";
  });
  avatar.on(StreamingEvents.AVATAR_STOP_TALKING, () => {
    voiceStatus.textContent = "Waiting for you to
speak...";
  });
}
```

Add Voice Chat Functions

Create functions to manage the switching between modes and starting the voice chat.

TypeScript

```
async function startVoiceChat() {
  if (!avatar) return;

  try {
    await avatar.startVoiceChat({
      useSilencePrompt: false
    });
    voiceStatus.textContent = "Waiting for you to
speak...";
  } catch (error) {
    console.error("Error starting voice chat:", error);
    voiceStatus.textContent = "Error starting voice
chat";
  }
}

async function switchMode(mode: "text" | "voice") {
  if (currentMode === mode) return;

  currentMode = mode;

  if (mode === "text") {
    textModeBtn.classList.add("active");
    voiceModeBtn.classList.remove("active");
    textModeControls.style.display = "block";
    voiceModeControls.style.display = "none";
    if (avatar) {
      await avatar.closeVoiceChat();
    }
  } else {
    textModeBtn.classList.remove("active");
    voiceModeBtn.classList.add("active");
    textModeControls.style.display = "none";
    voiceModeControls.style.display = "block";
    if (avatar) {
      await startVoiceChat();
```

```
    }
  }
}
```

Enable Voice Mode Button

Enable the voice mode button once the avatar stream is ready.

TypeScript

```
function handleStreamReady(event: any) {
  if (event.detail && videoElement) {
    videoElement.srcObject = event.detail;
    videoElement.onloadedmetadata = () => {
      videoElement.play().catch(console.error);
    };
    voiceModeBtn.disabled = false;  // Enable voice mode
after stream is ready
  }
}
```

Add Event Listeners

Make sure to add event listeners for switching modes.

TypeScript

```
// Add these with your other event listeners
textModeBtn.addEventListener("click", () =>
switchMode("text"));
voiceModeBtn.addEventListener("click", () =>
switchMode("voice"));
```

Important Notes:

- Voice mode button starts disabled and enables only after stream is ready
- Always use language code "en" instead of "English"
- Voice chat status updates automatically through events

- Voice chat starts when switching to voice mode
- Make sure to handle cleanup when switching modes

Conclusion

In this guide, we've walked through the process of integrating built-in voice chat into the Vite demo project using HeyGen's Streaming API. By following these steps, you can seamlessly switch between text and voice modes, allowing for a more interactive experience. *Keep in mind that for custom speech input with voice chat, you will need to integrate your own STT solution. https://github.com/HeyGen-Official/InteractiveAvatarNextJSDemo*