Documentation for Real-Time Audio Conversion Service

API Endpoints and Their Usage

The audio conversion service is built with a WebSocket API that allows clients to stream audio data in real time for format conversion. Below are the primary endpoints and their respective usages.

1. WebSocket Endpoint

- **URL**: `ws://localhost:3000/ws`
- **Method**: WebSocket (persistent connection)

Usage:

- This WebSocket endpoint is used for streaming audio data from a client (in WAV format) to the server, where it is processed and converted to FLAC format in real time. Clients can send audio data in chunks to this endpoint, and the server will return each converted chunk to the client for seamless streaming.

Connection Flow:

1. **Connect**: Establish a WebSocket connection at `ws://localhost:3000/ws`.
2. **Send Audio Data**: Stream audio data in WAV format to the server.
3. **Receive FLAC Data**: The server processes each chunk and returns FLAC data back to the client.
4. **Close Connection**: Once all data has been converted, close the WebSocket connection.

2. Status Endpoint (Optional, for Monitoring)

- **URL**: `http://localhost:3000/status`
- **Method**: GET
- **Response**: `{ "status": "online" }`

Usage:

- This endpoint can be used to check the status of the server. It is useful for monitoring and ensuring the service is running correctly before initiating a WebSocket connection.

Setup Instructions

1. Local Setup

Follow these steps to set up the service on your local machine.

Prerequisites:

- **Go** installed on your system.
- **SoX** (Sound eXchange) installed with access in your system PATH for audio conversion.

Steps:

1. **Clone the Repository**:

   ```
   Bash
   git clone <repository-url>
   cd audio-conversion-service
   ```

2. **Install Dependencies**:
   a. Ensure you have SoX installed and accessible in your system PATH.
   b. For Go dependencies, if any external packages are used, run:

   ```
   bash
   go mod tidy
   ```

3. **Configure Environment Variables** (if necessary):
   a. Set up any environment variables required by the service. For example:

   ```
   Bash
   export WEBSOCKET_PORT=3000
   ```

4. **Run the Go Server**:
   a. Start the server locally:

      ```
      Bash
      go run main.go
      ```

   b. The server should now be running on `ws://localhost:3000/ws`.

2. Deployment Instructions

To deploy this service on a cloud server or containerized environment, follow these steps.

Containerization (Optional):

- You can create a Dockerfile for this project if you want to deploy it in a containerized environment. Example Docker commands:
  ```
  dockerfile
  FROM golang:latest
  WORKDIR /app
  COPY . .
  RUN go build -o audio-service .
  CMD ["./audio-service"]
  ```

Server Setup:

1. **Install Dependencies**: Ensure the server has Go and SoX installed.
2. **Deploy Binary**: Upload the compiled binary to your server.
3. **Run Service**: Execute the binary and keep the service running using a process manager like `systemd` or `pm2`.

Environment:

- Set environment variables or configuration files on the server for managing different deployment settings (ports, paths, etc.).

Testing Strategy

Testing this real-time audio conversion service involves checking each functional component to ensure reliability and performance.

1. Unit Testing

- **Goal**: Verify individual functions for handling WebSocket connections, processing audio, and error handling.

2. Integration Testing

- **Goal**: Test the WebSocket server and its interaction with SoX, ensuring correct end-to-end flow.
- **Method**: Use a client to send and receive mock audio data. Simulate a live client-server connection.
- **Example**:
  - Start the Go server.
  - Use a WebSocket client (Python or Go) to send a test WAV file chunk and verify the received data is in FLAC format.

3. Manual Testing (Real-Time Streaming)

- **Goal**: Test the full real-time streaming functionality with real audio files.
- **Method**:
    - Run both the Python client and Go server locally.
    - Send a WAV file in chunks and confirm that the output file is correctly converted to FLAC.

  Example:

  ```
  bash
  python client.py --file=test_audio.wav
  ```

4. Performance Testing

- **Goal**: Check for latency and memory usage.
- **Method**: Measure processing time for various chunk sizes to ensure the service meets real-time performance standards.

Example Test Invocation

To test the service, you can use the following Python script as a client to send audio data in real-time to the WebSocket server and receive the converted FLAC data back.

Steps:

1. **Run the Go server**: Make sure the Go server is running on `ws://localhost:3000/ws`.
2. **Run the Python Client**:
   ```
   bash
   python client.py --file=path/to/your/test_audio.wav
   ```

3. **Check Output**:
    a. Verify the output FLAC file is correctly saved and confirm audio quality matches expectations.

By following these testing steps and deploying the service with these configurations, you can ensure a reliable, real-time audio conversion experience. This documentation should guide you through setup, deployment, and testing with clarity.