**Real-Time WAV to FLAC Streaming Service Documentation**

This document provides an overview of the service’s API endpoints, setup instructions for running and deploying the service, and a testing strategy to validate functionality and performance.

**1. API Documentation**

This service uses WebSocket to accept a real-time stream of WAV audio data, convert it to FLAC format, and stream the converted audio back to the client.

**WebSocket Endpoint**

* **Endpoint**: ws://<hostname>:<port>/ws
* **Description**: Establishes a WebSocket connection that accepts a continuous stream of WAV audio data from the client, converts it to FLAC in real-time, and streams the converted FLAC data back to the client.
* **Request Type**: WebSocket connection
* **Expected Input**:
  + Binary WAV audio data chunks, typically sent every few milliseconds from the client’s MediaRecorder.
* **Response**:
  + Binary FLAC audio data chunks, streamed back to the client in real-time.

**WebSocket Usage Example**

1. **Client-to-Server**: The client connects to the WebSocket endpoint and streams WAV audio data in small chunks.
2. **Server-to-Client**: The server responds with FLAC-encoded audio data in real-time, as it converts the incoming WAV stream.

**2. Setup Instructions**

**Prerequisites**

1. **Go** (version 1.16 or newer) - Required for running the backend server.
2. **ffmpeg** - Needed for audio conversion from WAV to FLAC. Ensure ffmpeg is installed and accessible from the command line.

**Installation Steps**

1. **Clone the Repository**:

git clone https://github.com/yourusername/wav-to-flac-conversion.git

cd wav-to-flac-conversion

1. **Install Go Dependencies**:

go mod tidy

1. **Install ffmpeg**:
   * **Ubuntu**: sudo apt update && sudo apt install ffmpeg
   * **macOS (Homebrew)**: brew install ffmpeg
   * **Windows**: [Download ffmpeg](https://ffmpeg.org/download.html) and add it to your system PATH.
2. **Create Temporary Directory**:
   * The project uses a temp directory to store temporary files. Ensure this directory exists in the project root:

mkdir temp

1. **Run the Server**:
   * Start the Go server, which will handle the backend WebSocket API and serve the frontend files:

go run main.go

1. **Access the Application**:
   * Open your browser and navigate to http://localhost:8080/static/index.html to access the frontend interface.

**Deployment Instructions**

1. **Configure Environment Variables**:
   * Set PORT or update main.go if deploying on a custom port.
2. **Deploy to a Server**:
   * For production deployment, consider using a platform that supports Go applications, such as:
     + **Docker**: Create a Dockerfile to containerize the application.
     + **Heroku**: Use heroku buildpacks:add heroku/go to deploy on Heroku.
     + **AWS EC2 or DigitalOcean**: Set up a VM, install Go and ffmpeg, and deploy the app manually.
3. **Run in Production Mode**:
   * Set Go’s environment to production mode:

export GIN\_MODE=release

1. **Starting the Application**:
   * Run the application on the server and make it accessible via a public URL for WebSocket communication.

**3. Testing Strategy**

A robust testing strategy ensures that the service functions correctly and handles real-time streaming without errors. The testing approach includes **unit tests**, **integration tests**, and **error-handling tests**.

**Testing Tools**

* **Go’s testing Package**: For running unit and integration tests.
* **httptest**: For creating mock HTTP servers.
* **WebSocket Client Libraries**: Use github.com/gorilla/websocket for WebSocket-based tests.

**Unit Tests**

The unit tests focus on verifying the conversion functionality using ffmpeg and error handling within the backend.

1. **Conversion Validation**: Ensures the backend correctly converts WAV data to FLAC.
2. **Error Handling**: Verifies the backend handles invalid or incomplete data gracefully.

**Example Unit Test Commands**

Run unit tests for the conversion and error handling logic:

go test ./controllers/conversion\_test.go -v

**Integration Tests**

Integration tests ensure that the WebSocket connection and real-time data streaming work as expected. This includes:

1. **WebSocket Connection Tests**: Validate the WebSocket connection, sending WAV data to the server, and receiving FLAC data in response.
2. **Real-Time Streaming Tests**: Ensure the backend can handle streaming data without interruptions.
3. **Error Simulation**: Test how the server handles unexpected data and abrupt connection closures.

**Example Integration Test Commands**

Run the WebSocket integration tests:

go test ./controllers/websocket\_integration\_test.go -v

**Testing Examples**

Here’s how to test the service’s core functionalities:

1. **Test WebSocket Connection**:
   * Use websocket.DefaultDialer.Dial to establish a connection to ws://localhost:8080/ws.
   * Send simulated WAV data in small chunks, mimicking real-time streaming.
   * Assert that the server responds with FLAC data chunks.
2. **Error Handling Tests**:
   * Test error handling by sending invalid data and observing if the server gracefully handles it without crashing.
   * Test scenarios where the WebSocket connection is closed prematurely.
3. **Premature Connection Closure**:
   * Test how the server behaves if the client closes the WebSocket connection mid-stream.

**Example Test Code Snippets**

**Sample Unit Test for Conversion Validation**

func TestConvertWavToFlac(t \*testing.T) {

input := bytes.NewBuffer([]byte("sample wav data"))

cmd := exec.Command("ffmpeg", "-f", "wav", "-i", "pipe:0", "-c:a", "flac", "-f", "flac", "pipe:1")

cmd.Stdin = input

output := &bytes.Buffer{}

cmd.Stdout = output

if err := cmd.Run(); err != nil {

t.Fatalf("Conversion failed: %v", err)

}

if output.Len() == 0 {

t.Error("Expected output data, but got none")

}

}

**Sample Integration Test for WebSocket Streaming**

func TestWebSocketStreaming(t \*testing.T) {

server := httptest.NewServer(SetupRouter())

defer server.Close()

wsURL := "ws" + server.URL[4:] + "/ws"

ws, \_, err := websocket.DefaultDialer.Dial(wsURL, nil)

if err != nil {

t.Fatalf("Failed to connect to WebSocket: %v", err)

}

defer ws.Close()

wavData := []byte("simulated wav data")

for i := 0; i < 5; i++ {

if err := ws.WriteMessage(websocket.BinaryMessage, wavData); err != nil {

t.Fatalf("Failed to send WAV data: %v", err)

}

time.Sleep(100 \* time.Millisecond)

}

for i := 0; i < 5; i++ {

\_, message, err := ws.ReadMessage()

if err != nil {

t.Fatalf("Failed to read FLAC data: %v", err)

}

if len(message) == 0 {

t.Error("Expected FLAC data, but got empty message")

}

}

}

**Sample Error Handling Test**

func TestWebSocketErrorHandling(t \*testing.T) {

server := httptest.NewServer(SetupRouter())

defer server.Close()

wsURL := "ws" + server.URL[4:] + "/ws"

ws, \_, err := websocket.DefaultDialer.Dial(wsURL, nil)

if err != nil {

t.Fatalf("Failed to connect to WebSocket: %v", err)

}

defer ws.Close()

invalidData := []byte("invalid data")

if err := ws.WriteMessage(websocket.BinaryMessage, invalidData); err != nil {

t.Fatalf("Failed to send invalid data: %v", err)

}

\_, message, err := ws.ReadMessage()

if err == nil && len(message) != 0 {

t.Error("Expected error or no data for invalid input, but got a response")

}

}

**Summary**

This guide provides detailed information on the setup, API usage, and testing strategy for your real-time WAV-to-FLAC streaming service, ensuring that the application performs reliably under various conditions.