AUDIO PROCESSING API

SAI HARSHATH MADDALI
Saiharshath1@gmail.com

Audio Processing API

Overview:

The primary goal is to develop an API to audio file processing, encapsulated within a container and designed to cater to user needs. This API encompasses essential functionalities such as file uploading, downloading, and dynamic volume adjustment. The OpenAPI 2.0 standard serves as the foundation, guaranteeing a standardized structure and comprehensive documentation to facilitate seamless integration for users.

A crucial security measure is implemented through an authentication system utilizing API keys, ensuring controlled and secure access to the API. Positioned as a user-facing solution, the API prioritizes simplicity and efficiency, allowing users to effortlessly upload audio files for processing, dynamically adjust volume levels, and download the resulting audio files. The containerization aspect contributes to the API's versatility, enabling easy deployment and making it adaptable for a variety of audio processing applications.

Use:

This codebase serves as a versatile API tool, allowing users to elevate their audio files within a user-friendly digital environment. Whether the intent is to refine audio levels for enhanced clarity or seamlessly integrate audio processing into more diverse range of applications.

Files Involved:

index.html – Serves as a front end.

app.py – Serves as a backend and flask as server.

The frontend is anchored by index.html, which serves as the user interface. This HTML file incorporates elements of CSS and JavaScript, culminating in an engaging and responsive environment for users.

On the backend, app.py takes the lead as the driving force, utilizing the Flask web framework and the PyDub library to expertly manage user authentication, process audio files, and handle API keys.

Languages Used:

The frontend harnesses the trio of HTML, CSS, JavaScript and Bootstrap to create a visually appealing interface, ensuring a seamless interaction between users and the Audio Processing App. Meanwhile, on the backend, Python takes the lead, providing the foundation for the robust functionalities encompassing secure authentication, efficient audio file processing, and adept error handling.

Setup Instructions:

1. Clone repository:
    git clone https://github.com/msharshath/audio-processing-api.git
    cd audio-processing-app

2. Install the dependencies:
    pip install -r requirements.txt

3. Building docker image from Dockerfile:
    docker build -t audio-processing-api .

4. Run the docker container:
    docker run -p 8080:80 audio-processing-api

5. Access the api:
    Open a web browser and go to **http://localhost:8080** to interact with the Audio Processing App.

API Guidelines:

1. Authentication:

    *   The Endpoint is **/login**.
    *   Initiate authentication by sending a POST request to the /login endpoint.
    *   Include a JSON payload with the `username` and `password` fields. Here, A predefined user only works as
        Username: **admin** and
        Password as **admin**
        (Only these details will work)
    *   A successful login will return a unique API key in the response.
    *   If the same user logged in for second time, it asks to enter the API key.

2. API Key Submission:

    *   The Endpoint is **/authenticate**.
    *   After receiving the API key, submit it for authentication by sending a POST request to the /authenticate endpoint.
    *   If API key is already known, enter the API key.
    *   Include a JSON payload with the username and ApiKey fields.

3. File Upload:
    *   The Endpoint is /**upload**
    *   Use the API key obtained during authentication to upload an audio file.
    *   Send a POST request to the /upload endpoint with the audio file attached.
    *   The server will respond with a message indicating the success or failure of the upload.

4. Adjust Volume:

- The Endpoint is **/adjust**.
- Adjust the volume of the uploaded audio file by sending a POST request to the `/adjust` endpoint.
- Include a JSON payload with the volumeLevel field representing the desired volume adjustment in dBFS.

5. Download Audio:

- The Endpoint is **/download**.
- After adjusting the volume, initiate a GET request to the /download endpoint to retrieve the processed audio file.
- The file will be returned as an attachment with the name "adjusted_new_audio.mp3".

6. Logout:

- The Endpoint is from (Client-Side)
- Securely terminate the session by clicking the "Logout" button on the client-side.
- This action resets the application and requires re-authentication for further API access.

Structure of audio-processing-api directory:

audio_processing_api/

├──── **Readme.doc** - Documentation for your project, providing guidance on setup and usage.

├──── **app.py** - Main Python file containing the code for your web application.

├──── **Openapi.yaml** - Specification file describing the structure of your API.

├──── **test_api.py** - File with unit tests for your API.

├──── **Dockerfile** - Instructions for building a Docker image of your application.

├──── **requirements.txt** - List of dependencies required for your application.

├──── **backup_copy.txt** - Backup text file, possibly containing a copy of api code.

├──── **templates/** - Directory containing HTML, CSS, JavaScript for API UI.

| └──── **index.html**

| └──── **logo.png**

Working process:

1. The audio-processing-app.git is cloned.

    Command: **git clone https://github.com/msharshath/audio-processing-app.git**

2. Open the Visual studio console of terminal to create a docker image.

    Command: **docker build -t audio-processing-api .**

    (**.** defines current path, **t** defines image tag)

```
● PS C:\Users\z004tu0x\Desktop\ai_coustics\audio_processing_api> docker build -t audio-processing-api .
  [+] Building 3.6s (11/11) FINISHED                                                          docker:default
   => [internal] load .dockerignore                                                                    0.0s
   => => transferring context: 2B                                                                      0.0s
   => [internal] load build definition from Dockerfile                                                 0.0s
   => => transferring dockerfile: 804B                                                                 0.0s
   => [internal] load metadata for docker.io/library/python:3.9                                        3.1s
   => [1/6] FROM docker.io/library/python:3.9@sha256:b2e47a7eca3178e4ce6c095d3a2d1cd05bfa616efe7f2047c95fffe159e00166  0.0s
   => [internal] load build context                                                                    0.1s
   => => transferring context: 555B                                                                    0.1s
   => CACHED [2/6] WORKDIR /app                                                                         0.0s
   => CACHED [3/6] RUN apt-get update    && apt-get install -y libssl-dev libffi-dev ffmpeg    && rm -rf /var/lib/apt/lis  0.0s
   => CACHED [4/6] COPY . .                                                                             0.0s
   => CACHED [5/6] RUN pip install --no-cache-dir -r requirements.txt                                   0.0s
   => CACHED [6/6] RUN useradd -m myuser                                                                0.0s
   => exporting to image                                                                                0.0s
   => => exporting layers                                                                               0.0s
   => => writing image sha256:29664c4882fb15c2502046d443a29ebf936cbc99dde9a4289eb9cd64d3819885          0.0s
   => => naming to docker.io/library/audio-processing-api                                               0.0s

  View build details: docker-desktop://dashboard/build/default/default/qibgtc74lrpxkxp2z78zt2m98
```

3. Now, Create docker container using the created docker image(Step -2) – audio-processing-api

    Command: **docker run -p 8080:80 audio-processing-api**

    (p - flag is used to map ports between the host system and the Docker container)
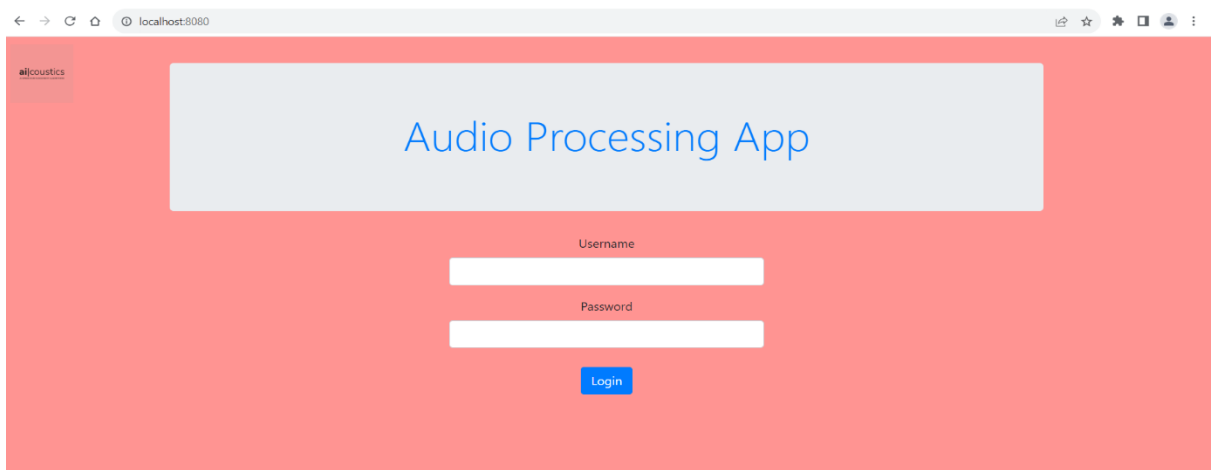
    (8080:80 means that port 8080 on the host machine is being mapped to port 80 inside the Docker container)

```
PS C:\Users\z004tu0x\Desktop\ai_coustics\audio_processing_api> docker run -p 8080:80 audio-processing-api
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:80
 * Running on http://172.17.0.2:80
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 966-721-343
```
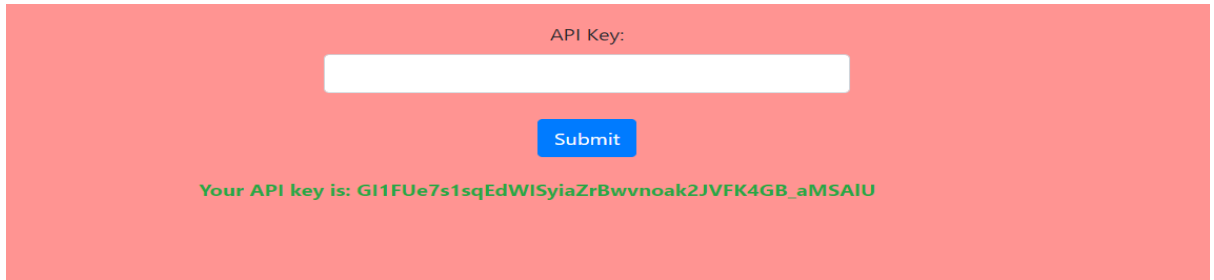
4. Now, open **http://localhost:8080** in web browser.

    Its look like the below page, to enter the Username and Password.

    By default, the only user can login is Username: **admin**, Password: **admin**

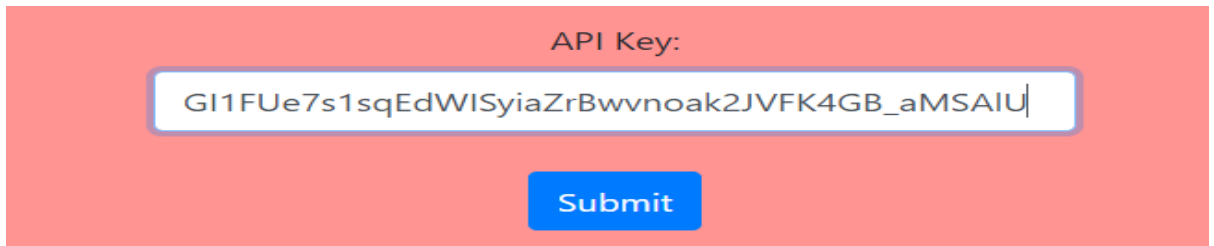5. As you are a new user, it gives you API key.

API Key:

Submit

Your API key is: GI1FUe7s1sqEdWlSyiaZrBwvnoak2JVFK4GB_aMSAlU

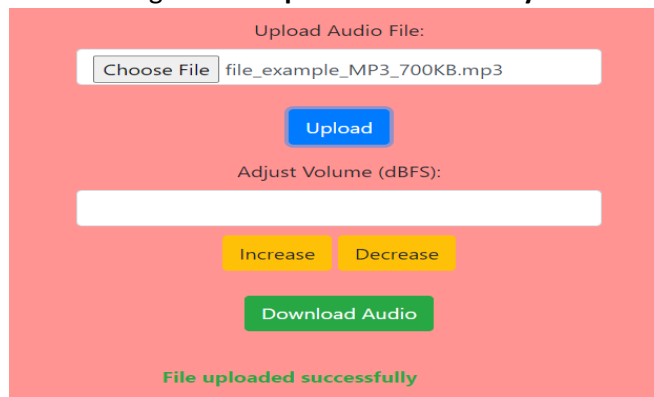User should enter the obtained API key in API key field and clicks on submit.

API Key:

GI1FUe7s1sqEdWlSyiaZrBwvnoak2JVFK4GB_aMSAlU

Submit

6. Once submitted, A upload section will be shown where user can upload audio file and click on submit.
   If successful, it throws a message as **File Uploaded Successfully**.

Upload Audio File:

Choose File file_example_MP3_700KB.mp3

Upload

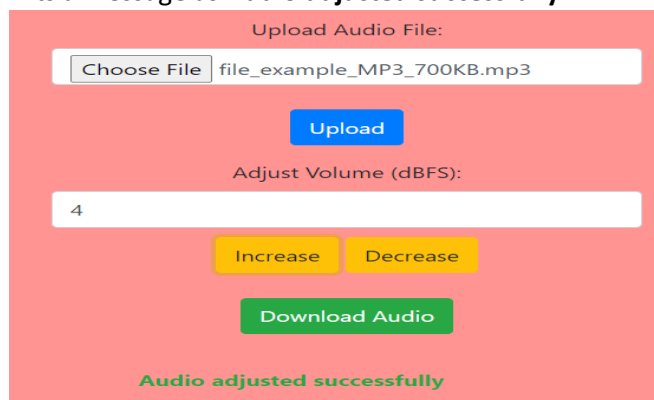Adjust Volume (dBFS):

Increase   Decrease

Download Audio

File uploaded successfully

The audio files are moved to backend storing them in a temporary storage for further process.

7. According to the user requirement, Enter the volume and click on button for a condition either **increase** or **decrease** the volume of audio file.
   For an example, here a volume of 4 to be increased.
   If it successful, it prints a message as **Audio adjusted successfully**.

Upload Audio File:
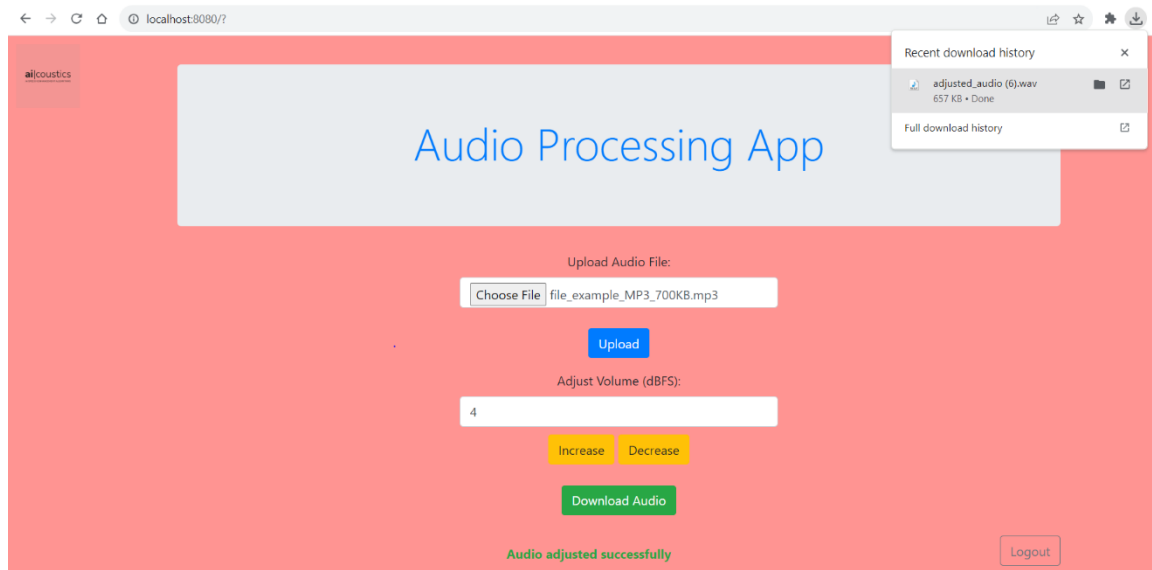
Choose File file_example_MP3_700KB.mp3

Upload

Adjust Volume (dBFS):

4

Increase   Decrease

Download Audio

Audio adjusted successfully

8. Once the last step is successful, User can download the adjusted audio file using Download Audio button. When button is clicked, Audio file will download automatically.

9. The downloaded audio file works with user selected volume.

How is the API key generated?

The built-in Python module **secrets** is used to produce cryptographically safe random numbers, but it can also be used in other ways. You might suggest that you could generate these random numbers using the random module, but the secrets module has access to the most secure source of randomness that your computer can supply. This makes it ideal for a wide range of applications, including password management, authentication, and security tokens.

Import secrets

The following are the functions which are used to generate secure tokens.

Secrets.token_bytes([nbytes=None]): Return a secure random byte string containing the number of bytes. If n-bytes are not supplied, a reasonable default gets used.

Secrets.token_urlsafe([nbytes=None]): Return a secure random URL-safe text string, containing n-bytes random bytes. Use this method to generate secure hard-to-guess URLs.

Here, I used **secrets.token_urlsafe(32)**, to generate random text string which is acts like a API unique key.

In the way our application works, when someone tries to log in using the `/login` endpoint, the system checks their username and password. Right now, we're keeping things simple by using a fixed **username ('admin') and password ('admin')**. If the login is successful, the system then looks to see if the user already has an API key.

If they do, the user is told to stick with their current key. If not, a new, strong, and random API key is created for them using the `secrets.token_urlsafe(32)` function. This new key is then linked to the user's information in the system. The user gets a response saying whether a new API key was made or if they should use their existing one.

It's important to note that in a real-world situation, we'd typically use more advanced security measures and storage methods for user data and API keys.

However, as it stands, there's a limitation: when the server is turned off and back on, it treats everyone, including the 'admin,' as a new user, and any previously generated API keys are reset. To address this, a more permanent storage solution, like a database, would be needed to keep user data intact across server restarts.

Furthermore, handling API keys would probably include factors like setting expiration periods for tokens, regularly changing keys through rotation, and adopting secure storage practices. These measures are implemented to minimize potential security threats.

How are the volume adjustment process works?

In the `/adjust` endpoint from app.py code, the process of adjusting audio volume begins by extracting the desired volume level from the JSON data received in the HTTP - POST request. It validates the volume level to ensure it is a valid numeric value. Subsequently, the code identifies the latest uploaded audio file from the **audio_storage** dictionary, where keys represent file names, and values represent timestamps of when the files were uploaded.

Once the latest audio file is determined, the actual audio content is retrieved from audio_storage. If the audio content is unavailable, an error response is returned. Following this, the code enters a placeholder section where the audio adjustment logic should be implemented. In the given code, the adjust_audio_volume function is called, passing the retrieved audio content and the desired volume level as parameters.

Inside the **adjust_audio_volume** function, the audio content is converted into an **AudioSegment** using the `pydub` library.

Pydub is an easy to comprehend, well-designed Python library for audio modification. Pydub is a staple tool for creating simple audio scripts.

These could include:

1. Loading and storing many audio file formats.
2. Audio can be separated or appended in segments.
3. Combining audio from two separate audio files.
4. Altering the audio levels or pan settings.
5. Featuring simple effects like filters.
6. rendering audio tones.

The volume adjustment is then performed by adding the specified volume level to the audio. This step simulates adjusting the volume. The adjusted audio is logged for reference, and the content is exported as an MP3 file. The adjusted audio content is then updated in the audio_storage dictionary.

It's crucial to emphasize that the logic used here is a placeholder, and for effective audio processing, it should be replaced with the appropriate processing code. Additionally, the exact audio adjustment logic may vary based on specific requirements, such as normalization, equalization, or compression, and should be implemented accordingly.

Docker Overview:

Docker is a containerization platform that allows developers to package applications and their dependencies into standardized units called containers. Containers provide a consistent and isolated environment, ensuring that applications run reliably across different computing environments.

Use of Docker:

1. Portability: Docker enables the creation of lightweight and portable containers, allowing applications to run consistently across various environments.

2. Isolation: Containers encapsulate applications and their dependencies, preventing conflicts and ensuring a clean runtime environment.

3. Efficiency: Docker optimizes resource utilization by sharing the host OS kernel, resulting in faster startup times and reduced overhead.

4. Scalability: Applications can be easily scaled by deploying multiple instances of containers, providing a flexible and scalable architecture.

Purpose of Dockerfile:

A Dockerfile is a script that contains instructions for building a Docker image. It defines the steps to set up the environment, install dependencies, and configure the application within a container.

Commands:

1. To know docker installed or not: **docker –version**
2. Create docker image: **docker build -t (tag_name) (Dockerfile_location)**
3. Check available docker images: **docker images**
4. Remove docker image: docker rmi -f (docker_image_name)
5. Create and run docker container: **docker run -p 8080:80 (docker_image_name)**
6. Check docker running docker containers: **docker ps**
7. Check available docker containers: **docker ps -a**
8. Start the docker container: **docker start (docker_container_name)**
9. Enter into docker container: **docker exec -it (docker_container_name) /bin/bash**
10. To exit from docker container: **exit**
11. Stop the docker container: **docker stop (docker_container_name)**
12. Delete docker container: **docker rm (docker_container_name)**

API Documentation: - **Openapi.yaml**

To use this as API documentation, Open website and copy & paste this Openapi. Yaml code to website console:

1. https://app.apiary.io/audioprocessingapi/editor
2. https://editor.swagger.io/
3. https://editor-next.swagger.io/

The provided YAML code outlines the Swagger documentation for an Audio Processing API. This API offers a versatile solution for manipulating audio files, specializing in volume adjustment. The documentation describes key features, such as user authentication, file upload, volume adjustment, and download capabilities. Users can authenticate through the '/login' endpoint, receive a unique API key, upload audio files, adjust volume via the '/adjust' endpoint, and download the processed audio file.

The API leverages the Flask framework and PyDub library, ensuring seamless audio processing tasks. The Swagger documentation includes details on each endpoint, specifying the expected input parameters, responses, and potential error messages. The '/login' endpoint handles user authentication, '/upload' facilitates audio file uploads, '/adjust' adjusts audio volume, and '/download' enables users to retrieve the adjusted audio file.

The documentation provides clarity on the API's functionality, workflow, and expected interactions. It emphasizes simplicity and ease of integration, making it suitable for diverse audio processing applications. Additionally, security measures are outlined, with API keys used for authentication. Standard error responses are defined for scenarios like invalid API keys, unauthorized access, file not found, invalid volume levels, and no selected files. Overall, the YAML code offers a comprehensive overview of the Audio Processing API and serves as a helpful guide for developers.

Test cases: - test_api.py

The unit testing suite comprises three functions. The `setUp` function initializes a test client for controlled HTTP request simulation, while `tearDown` cleans up files created during testing. The `test_login_success` function assesses the login route with valid credentials, expecting a 200 status, True 'success,' and an 'apiKey.' Conversely, `test_login_failure` checks the response to incorrect credentials, anticipating a 401 status. Lastly, `test_adjust_audio` focuses on the '/adjust' route, testing volume adjustment with different file path specifications. These tests ensure correct functionality, detect regressions, and document expected behavior.

setUp Function:

The **setup** function initializes a test client for the Flask application, facilitating the simulation of HTTP requests for testing purposes. This ensures that each test case starts with a clean environment, preventing interference between different tests.

tearDown Function:

The **tearDown** function acts as a cleanup mechanism after each test case execution. It ensures that any files created during the tests, such as "adjusted_new_audio.mp3" and "test.mp3," are removed. This guarantees that the testing environment remains pristine.

test_login_success Function:

The **test_login_success** function evaluates the login route with correct credentials. It sends a POST request to the '/login' endpoint with the username **'admin'** and password **'admin'**. The test asserts that the response status code is 200, the 'success' attribute in the returned JSON is True, and an 'apiKey' is present in the response data.

test_login_failure Function:

The **test_login_failure** function assesses the login route when incorrect credentials are provided. It sends a POST request to the '/login' endpoint with the username **('admin')** and an incorrect password **('Admin')**. The test expects a response status code of 401, indicating authentication failure. The response content is printed for further inspection.

test_adjust_audio Function:

The **test_adjust_audio** function checks the '/adjust' route, specifically focusing on volume adjustment. It uploads an audio file named "Sample_Audio.mp3" using two different methods for specifying the file path. Subsequently, it sends a POST request to '/adjust' with a volume adjustment level of 3. The test asserts that the response status code is 200, and the response data contains the 'message' attribute.