ABOUT ORGANISATION:

INOFINIX is a leading information technology start up providing niche technology solutions & services to global customers.
Founded in 2018, providing niche technology solutions to the global clients to enable them to innovate and achieve their aspirations.
Services are based on maximum automation & maximum innovation, this allows the company to offer very attractive & competitive. Provided by the rich Business consulting and IT experience of our founders, we provide cutting-edge technology solutions & services. Guided by our Vision and Mission & Core Values we hire and nurture a highly talented and motivated team.

OBJECTIVE:

The primary objective of my internship at the company INOFINIX was to gain hands-on experience in software development within a professional environment. I aimed to enhance my programming skills in python, contribute to ongoing projects, and understand the software development lifecycle. Additionally, I sought to improve my collaboration and communication skills by working closely with team members across various functions. This experience was integral in confirming my career aspirations in the software field, allowing me to apply my theoretical knowledge in practical situations while building a network within the industry.

SOFTWARE LEARNT:

Anaconda

Anaconda is a distribution of the Python and R programming languages for scientific computing such as data science, machine learning applications, large-scale data processing, predictive analytics, etc., that aims to simplify package management and deployment. The distribution includes data-science packages suitable for Windows, Linux, and macOS. It is developed and maintained by Anaconda, Inc., which was founded by Peter Wang and Travis Oliphant in 2012. As an Anaconda, Inc. product, it is also known as Anaconda Distribution or Anaconda Individual Edition, while other products from the company are Anaconda Team Edition and Anaconda Enterprise Edition, neither of which are free.

Package versions in Anaconda are managed by the package management system *conda*. This package manager was spun out as a separate open-source package as it ended up being useful on its own and for things other than Python. Anaconda distribution comes with over 250 packages automatically installed, and over 7,500 additional open-source packages can be installed from PyPI as well as the conda package and virtual environment manager. It also includes a GUI, Anaconda Navigator, as a graphical alternative to the command-line interface (CLI).

Jupyter

Jupyter is an open-source project that provides a suite of interactive computing environments. The most popular tool in this suite is Jupyter Notebook, but there are other tools like JupyterLab and JupyterHub that expand its capabilities.
Jupyter Notebook is a web-based interactive computing platform where users can write and execute code (most commonly in Python, though it supports many other languages), visualize data, and document workflows.
Key features:
1) Supports live code execution.
2) Can include rich media (e.g., plots, images, videos, and equations) in notebook documents.
3) Commonly used in data science, machine learning, and academic environments.

Python

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

Python consistently ranks as one of the most popular programming languages, and has gained widespread use in the machine learning community. The official introduction to Python is that it is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Skills Acquired:

1) Watermarking images with OpenCV and Python:

Watermark is intentionally left Text/Logo onto the image. Watermarks are generally used by artists to protect the copyright of the image. Using watermarks we can ensure that the owner of the image is the person who imprinted the watermark on the image.

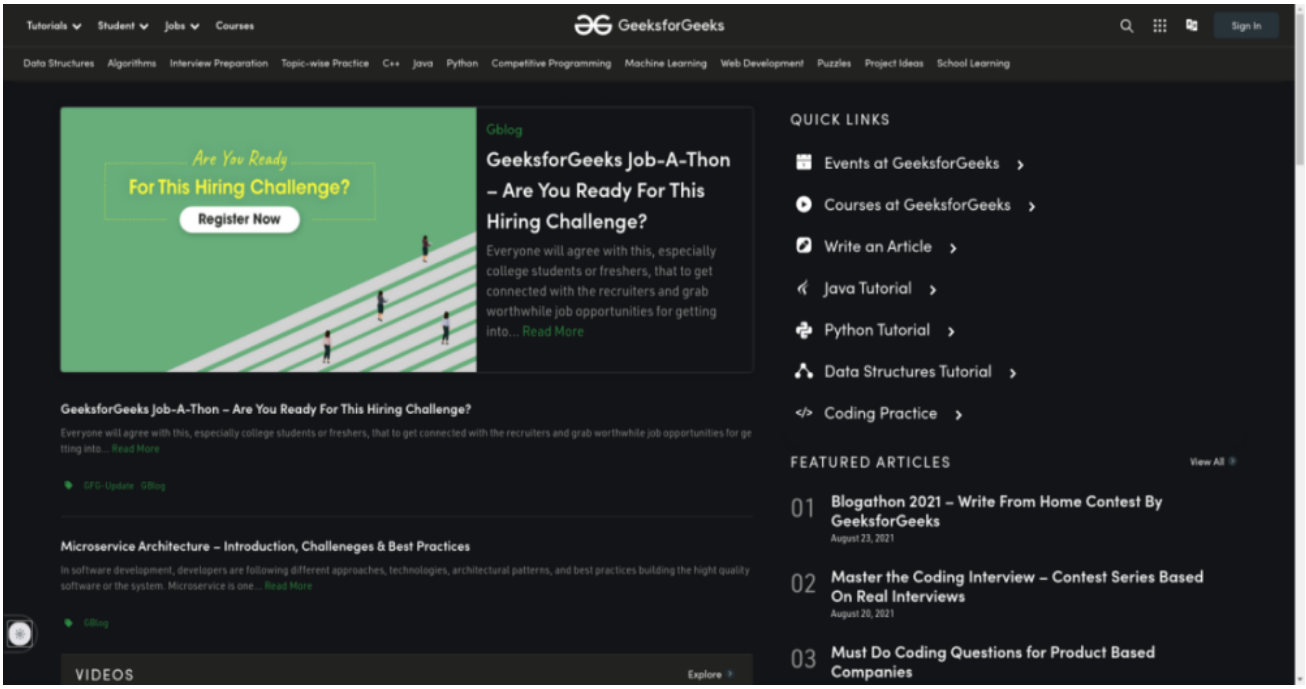


Image before watermark

Logo:

Step-by-Step implementation:

Step 1: Import the OpenCV and read the logo and the image on which you want to apply a watermark.

Step 2: Calculate the height and width of both the images and save them to other variables. We need to calculate the width and height because we are going to place our watermark somewhere onto the image and for that, we just have to know the proper width and height of both logo and the image.

Step 3: Now, we calculate the coordinates of the center of the image, because we are going to place our watermark at the center of the image.

Step 4: To add a watermark to an image we will use the addWeighted function from OpenCV. Firstly we will be providing the destination where we want to place the watermark, then we will pass that destination to the addWeighted function with the image and logo.
Syntax: *cv2.addWeighted(source1, alpha, source2, beta, gamma)*

In our case, source 1 will be the image where we want to place our logo and alpha will be the opacity of the logo and source2 will be the logo itself and we will set beta accordingly the alpha that is opacity and gamma will be 0.

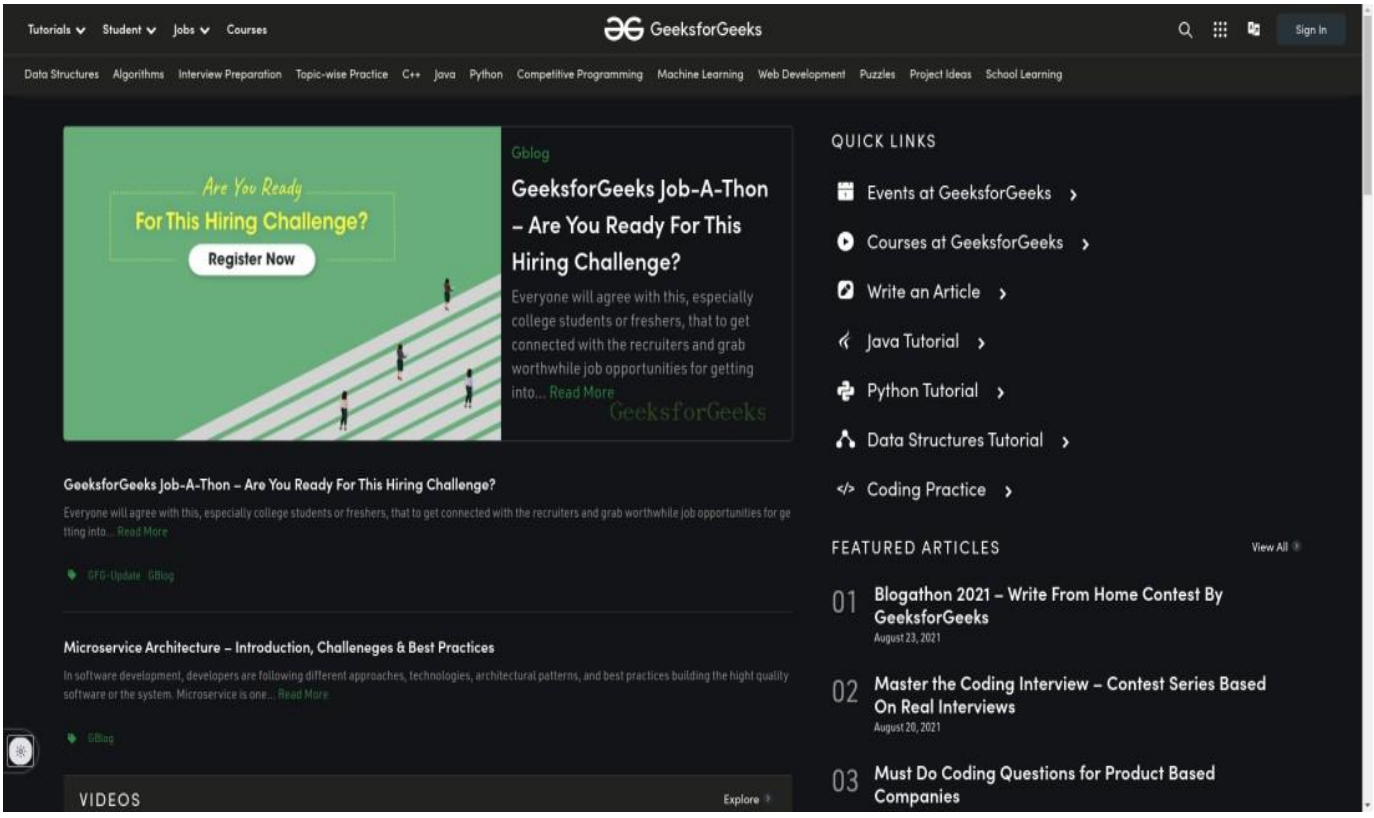
Step 5: After that, we are simply displaying the result and saving the output. To display the output we have used **imshow function** and to write/save the image, we are using **imwrite function** in both the functions firstly we have to provide the file name as a parameter and then the file itself. cv2.waitKey(0) is used to wait until the user press Esc Key, after that the cv2.destroyAllWindows function will close the window.

CODE

```
# watermarking image using OpenCV

# importing cv2
import cv2

# loading images
# importing logo that we are going to use
logo = cv2.imread("logo.jpg")

# importing image on which we are going to
# apply watermark
img = cv2.imread("dark.png")

# calculating dimensions
# height and width of the logo
h_logo, w_logo, _ = logo.shape

# height and width of the image
h_img, w_img, _ = img.shape

# calculating coordinates of center
```

```
# calculating center, where we are going to
# place our watermark
center_y = int(h_img/2)
center_x = int(w_img/2)

# calculating from top, bottom, right and left
top_y = center_y - int(h_logo/2)
left_x = center_x - int(w_logo/2)
bottom_y = top_y + h_logo
right_x = left_x + w_logo

# adding watermark to the image
destination = img[top_y:bottom_y, left_x:right_x]
result = cv2.addWeighted(destination, 1, logo, 0.5, 0)

# displaying and saving image
img[top_y:bottom_y, left_x:right_x] = result
cv2.imwrite("watermarked.jpg", img)
cv2.imshow("Watermarked Image", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

OUTPUT



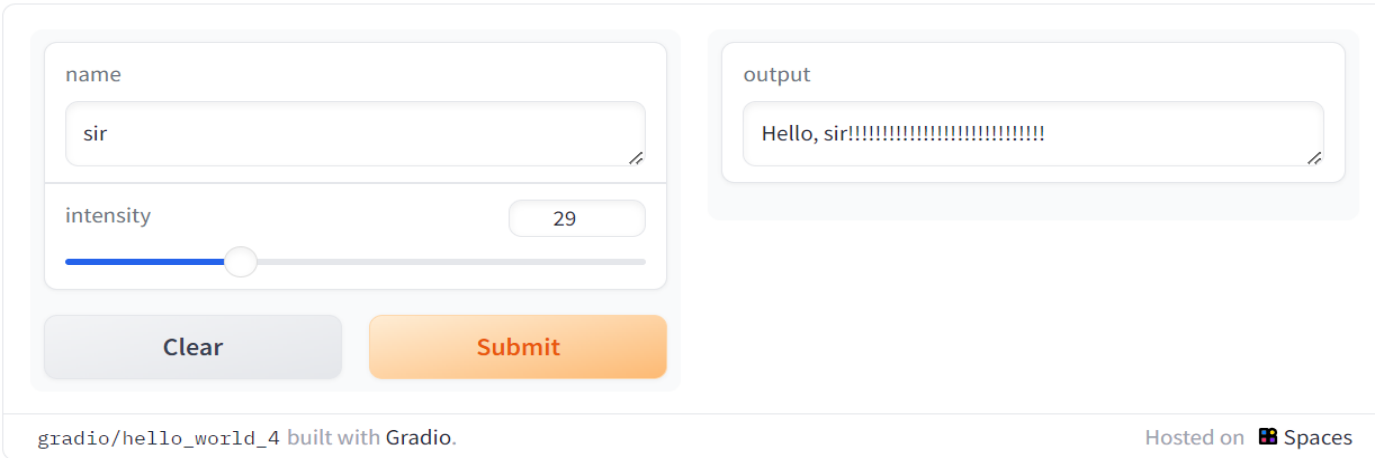
2) Gradio

Gradio is a Python library designed to help developers create and deploy web-based user interfaces (UIs) for machine learning models, data science applications, and other Python-based programs. It allows you to build interactive demos and applications with minimal code, making it easy to share machine learning models with non-technical users or clients without needing to develop a complex front-end.

The following code creates a simple web-based interface using the Gradio library, which allows users to interact with a Python function in a web browser.

```
import gradio as gr

def greet(name, intensity):
    return "Hello, " + name + "!" * int(intensity)

demo = gr.Interface(
    fn=greet,
    inputs=["text", "slider"],
    outputs=["text"],
)


demo.launch()
```

The purpose of this code is to create an interactive, user-friendly interface for a Python function (greet) using Gradio. Users can enter their name and adjust the intensity of the greeting using a slider, and the result will be displayed as a text output.

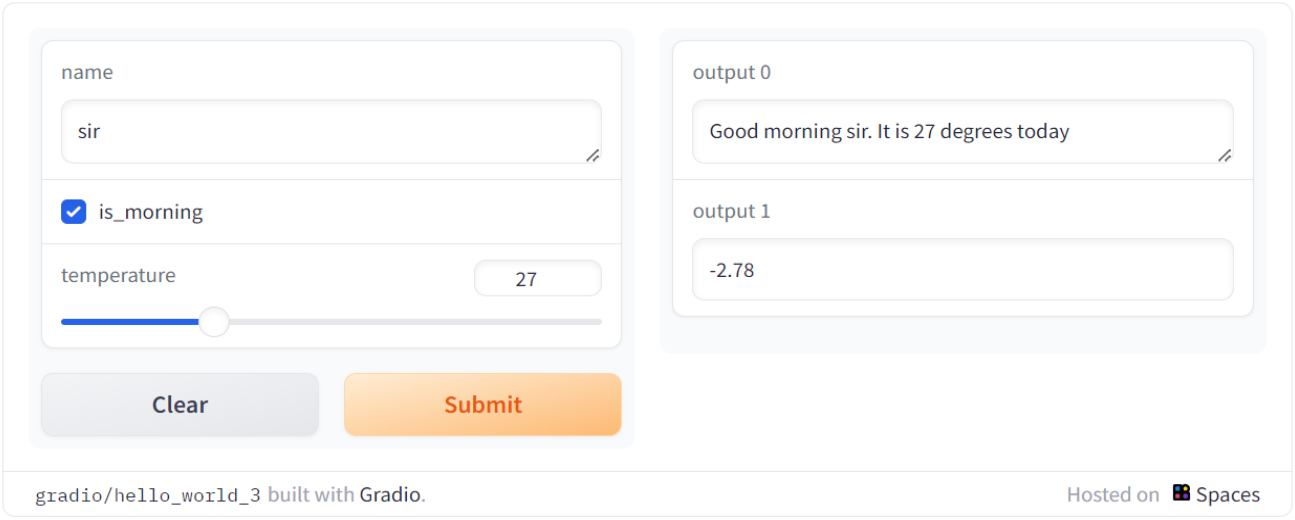


> Multiple Input and Output Components:

Suppose we had a more complex function, with multiple outputs as well. In the example below, we define a function that takes a string, boolean, and number, and returns a string and number.

```
import gradio as gr
```

```python
def greet(name, is_morning, temperature):
    salutation = "Good morning" if is_morning else "Good evening"
    greeting = f"{salutation} {name}. It is {temperature} degrees today"
    celsius = (temperature - 32) * 5 / 9
    return greeting, round(celsius, 2)

demo = gr.Interface(
    fn=greet,
    inputs=["text", "checkbox", gr.Slider(0, 100)],
    outputs=["text", "number"],
)
demo.launch()
```



Just as each component in the inputs list corresponds to one of the parameters of the function, in order, each component in the outputs list corresponds to one of the values returned by the function, in order.

> Gradio Components:

Gradio includes more than 30 pre-built components that can be used as inputs or outputs in your demo. These components correspond to common data types in machine learning and data science.

An Image Example:

Gradio supports many types of components, such as Image, DataFrame, Video, or Label. The following shows an example:

Input image:

Code:

```
import numpy as np
import gradio as gr

def sepia(input_img):
    sepia_filter = np.array([
        [0.393, 0.769, 0.189],
        [0.349, 0.686, 0.168],
        [0.272, 0.534, 0.131]
    ])
    sepia_img = input_img.dot(sepia_filter.T)
    sepia_img /= sepia_img.max()
    return sepia_img

demo = gr.Interface(sepia, gr.Image(), "image")
demo.launch()
```

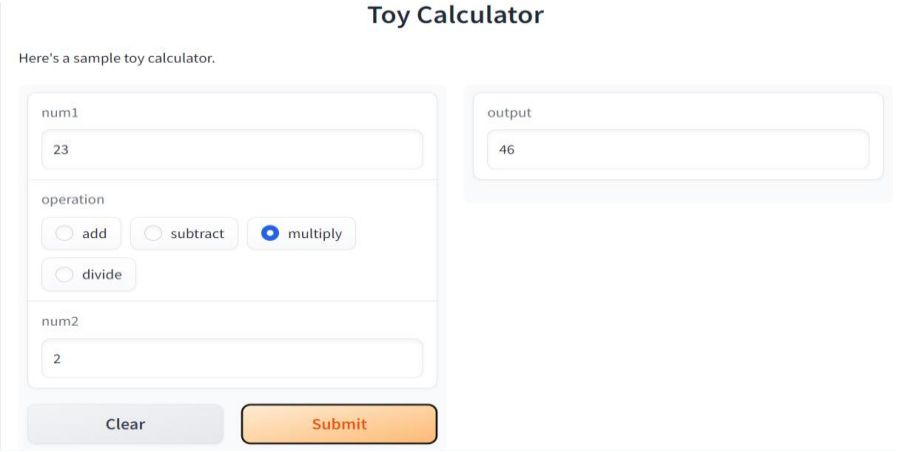
The purpose of this code is to create an interactive demo where users can upload an image, apply a sepia filter to it, and view the output directly in a web interface. This is useful for demonstrating image processing techniques without needing to build complex web applications.

Output image:

> Example Inputs

You can provide example data that a user can easily load into Interface. This can be helpful to demonstrate the types of inputs the model expects, as well as to provide a way to explore your dataset in conjunction with your model. To load example data, you can provide a nested list to the examples= keyword argument of the Interface constructor. Each sublist within the outer list represents a data sample, and each element within the sublist represents an input for each input component. The format of example data for each component is specified in the Docs.

```python
import gradio as gr

def calculator(num1, operation, num2):
    if operation == "add":
        return num1 + num2
    elif operation == "subtract":
        return num1 - num2
    elif operation == "multiply":
        return num1 * num2
    elif operation == "divide":
        if num2 == 0:
            raise gr.Error("Cannot divide by zero!")
        return num1 / num2

demo = gr.Interface(
    calculator,
    [
        "number",
        gr.Radio(["add", "subtract", "multiply", "divide"]),
        "number"
    ],
    "number",
    examples=[
        [45, "add", 3],
        [3.14, "divide", 2],
        [144, "multiply", 2.5],
        [0, "subtract", 1.2],
    ],
    title="Toy Calculator",
    description="Here's a sample toy calculator.",
)

demo.launch()
```

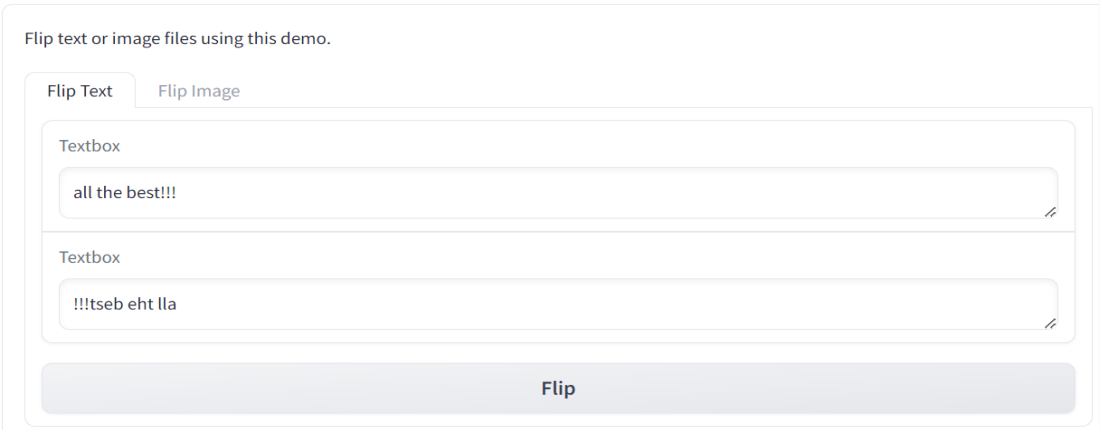


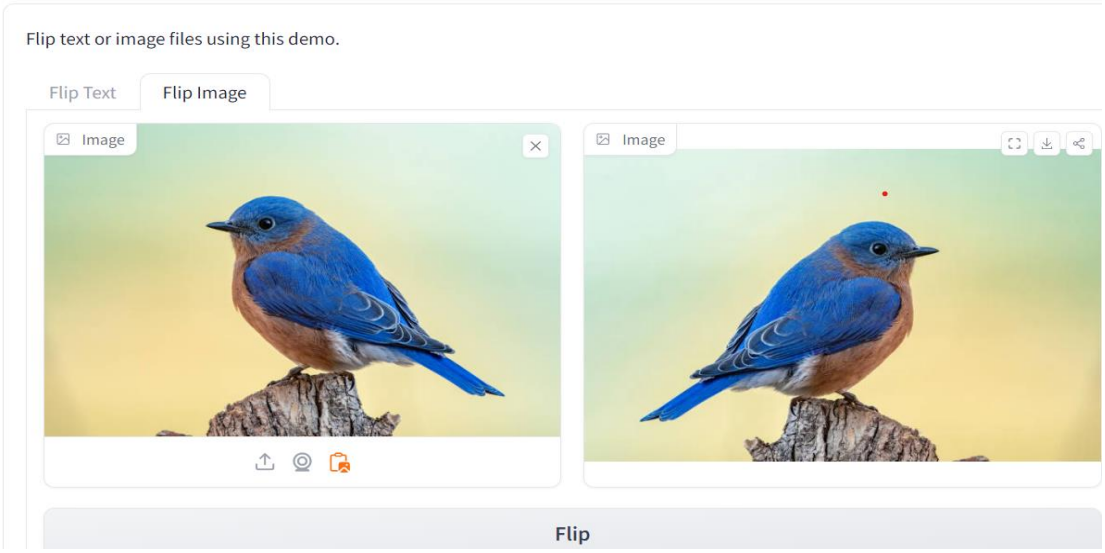
> Tabs and Accordions:

Tabs are created using the with gr.Tab('tab_name'): clause. Any component created inside of a with gr.Tab('tab_name'): context appears in that tab. Consecutive Tab clauses are grouped together so that a single tab can be selected at one time, and only the components within that Tab's context are shown.

```
import numpy as np
import gradio as gr

def flip_text(x):
    return x[::-1]

def flip_image(x):
    return np.fliplr(x)

with gr.Blocks() as demo:
    gr.Markdown("Flip text or image files using this demo.")
    with gr.Tab("Flip Text"):
        text_input = gr.Textbox()
        text_output = gr.Textbox()
        text_button = gr.Button("Flip")
    with gr.Tab("Flip Image"):
        with gr.Row():
            image_input = gr.Image()
            image_output = gr.Image()
        image_button = gr.Button("Flip")

    with gr.Accordion("Open for More!", open=False):
        gr.Markdown("Look at me...")
        temp_slider = gr.Slider(
            0, 1,
            value=0.1,
```

```
            step=0.1,
            interactive=True,
            label="Slide me",
        )

    text_button.click(flip_text, inputs=text_input, outputs=text_output)
    image_button.click(flip_image, inputs=image_input, outputs=image_output)

demo.launch()
```





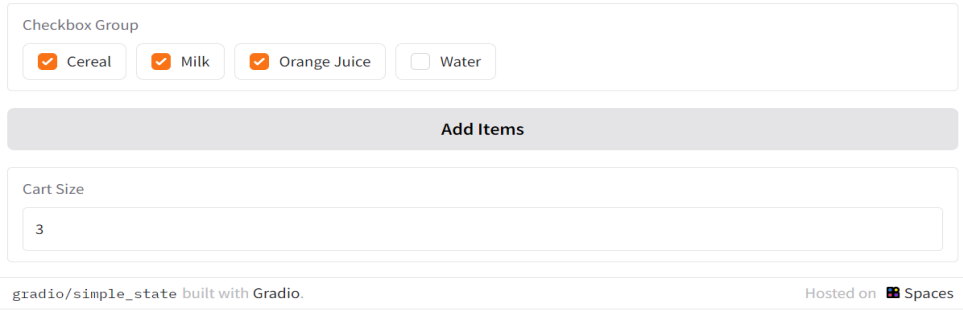
The Accordion is a layout that can be toggled open or closed. Like Tabs, it is a layout element that can selectively hide or show content. Any components that are defined inside of a with gr.Accordion('label'): will be hidden or shown when the accordion's toggle icon is clicked.
The purpose of this code is to create a web-based demo where users can either: Reverse a string (flip text) or Flip an image horizontally (flip image).

> State in Blocks

Session State:
Gradio supports session state, where data persists across multiple submits within a page session, in Blocks apps as well. To reiterate, session data is *not* shared between different users of your model. To store data in a session state, you need to do three things:
1. Create a gr.State() object. If there is a default value to this stateful object, pass that into the constructor.
2. In the event listener, put the State object as an input and output as needed.
3. In the event listener function, add the variable to the input parameters and the return value.

We have a simple checkout app below where you add items to a cart. You can also see the size of the cart.

```
import gradio as gr

with gr.Blocks() as demo:
    cart = gr.State([])
    items_to_add = gr.CheckboxGroup(["Cereal", "Milk", "Orange Juice", "Water"])

    def add_items(new_items, previous_cart):
        cart = previous_cart + new_items
        return cart

    gr.Button("Add Items").click(add_items, [items_to_add, cart], cart)

    cart_size = gr.Number(label="Cart Size")
    cart.change(lambda cart: len(cart), cart, cart_size)

demo.launch()
```

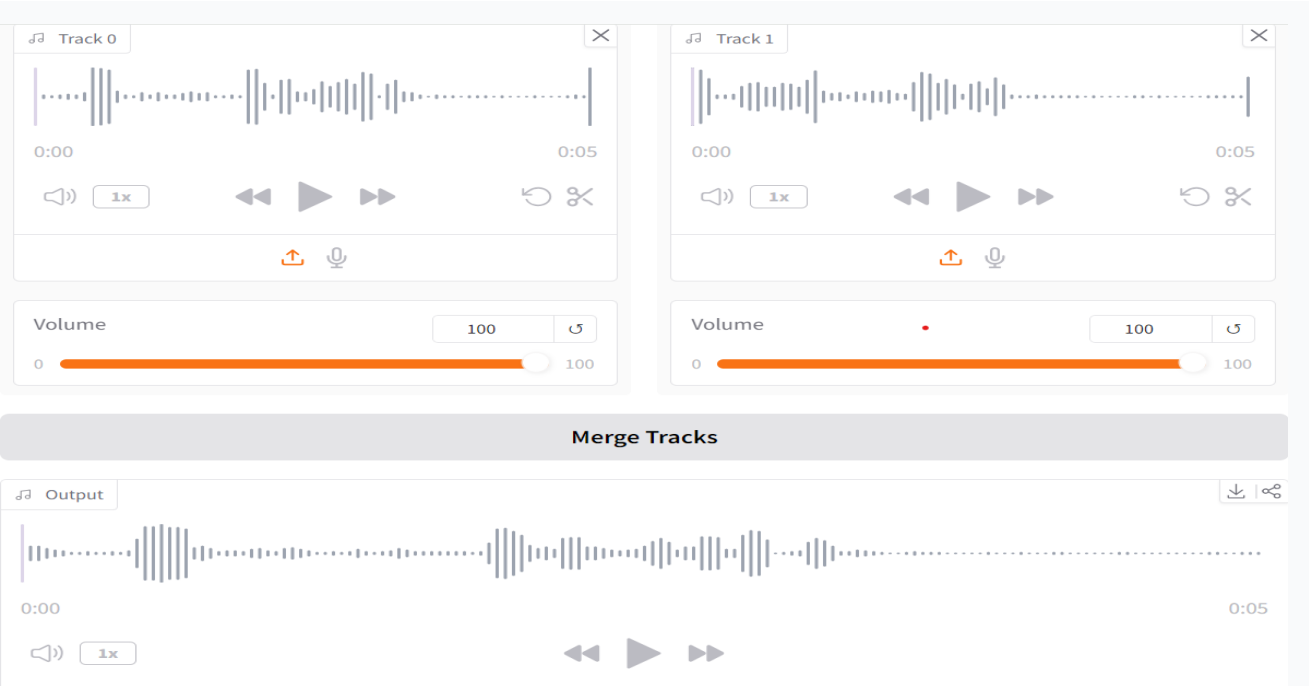


> Dynamic Audio Track Mixer using Gradio:

This Gradio app allows users to dynamically add multiple audio tracks, control the volume for each track, and merge them into a single output audio file. The objective is to provide an interactive interface where users can upload audio files, adjust their volumes, and merge them into a composite audio track.

```
import gradio as gr

with gr.Blocks() as demo:
```

```
    track_count = gr.State(1)
    add_track_btn = gr.Button("Add Track")

    add_track_btn.click(lambda count: count + 1, track_count, track_count)

    @gr.render(inputs=track_count)
    def render_tracks(count):
        audios = []
        volumes = []
        with gr.Row():
            for i in range(count):
                with gr.Column(variant="panel", min_width=200):
                    gr.Textbox(placeholder="Track Name", key=f"name-{i}",
show_label=False)
                    track_audio = gr.Audio(label=f"Track {i}", key=f"track-{i}")
                    track_volume = gr.Slider(0, 100, value=100, label="Volume",
key=f"volume-{i}")
                    audios.append(track_audio)
                    volumes.append(track_volume)

            def merge(data):
                sr, output = None, None
                for audio, volume in zip(audios, volumes):
                    sr, audio_val = data[audio]
                    volume_val = data[volume]
                    final_track = audio_val * (volume_val / 100)
                    if output is None:
                        output = final_track
                    else:
                        min_shape = tuple(min(s1, s2) for s1, s2 in
zip(output.shape, final_track.shape))
                        trimmed_output = output[:min_shape[0], ...][:,
:min_shape[1], ...] if output.ndim > 1 else output[:min_shape[0]]
                        trimmed_final = final_track[:min_shape[0], ...][:,
:min_shape[1], ...] if final_track.ndim > 1 else final_track[:min_shape[0]]
                        output += trimmed_output + trimmed_final
                return (sr, output)

            merge_btn.click(merge, set(audios + volumes), output_audio)

    merge_btn = gr.Button("Merge Tracks")
    output_audio = gr.Audio(label="Output", interactive=False)

demo.launch()
```



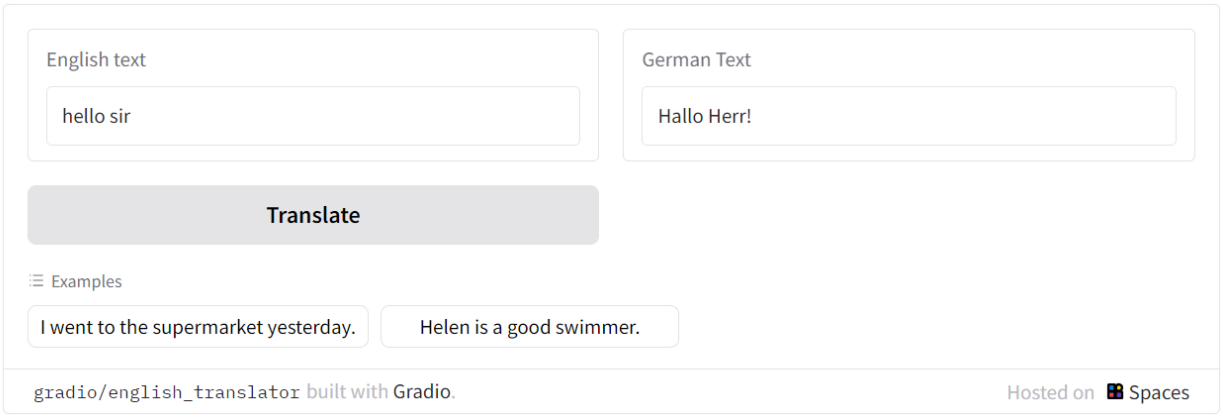
This code is designed to create a dynamic audio mixer app where users can upload multiple audio tracks, adjust the volume of each, and merge them into one output. The use of dynamic rendering (presumably via the render decorator) allows for adding new tracks interactively, making the app more flexible for users. However, the @gr.render decorator is not a known part of the official Gradio API. It might be a custom or experimental feature, or there may be a misunderstanding in the implementation. Gradio typically achieves dynamic behavior through State and event handling (e.g., Button.click).

> English to German Translation App Using Gradio and Hugging Face Pipeline:

The code aims to create a simple web-based app that translates English text into German. It leverages Gradio's user interface and Hugging Face's transformers library with a pre-trained model (t5-base) to handle the translation task. Users can input English text and get the translated German output.

```
import gradio as gr

from transformers import pipeline

pipe = pipeline("translation", model="t5-base")

def translate(text):
    return pipe(text)[0]["translation_text"]

with gr.Blocks() as demo:
    with gr.Row():
        with gr.Column():
            english = gr.Textbox(label="English text")
            translate_btn = gr.Button(value="Translate")
        with gr.Column():
            german = gr.Textbox(label="German Text")
```

```
    translate_btn.click(translate, inputs=english, outputs=german,
api_name="translate-to-german")
    examples = gr.Examples(examples=["I went to the supermarket yesterday.",
"Helen is a good swimmer."],
                           inputs=[english])
demo.launch()
```

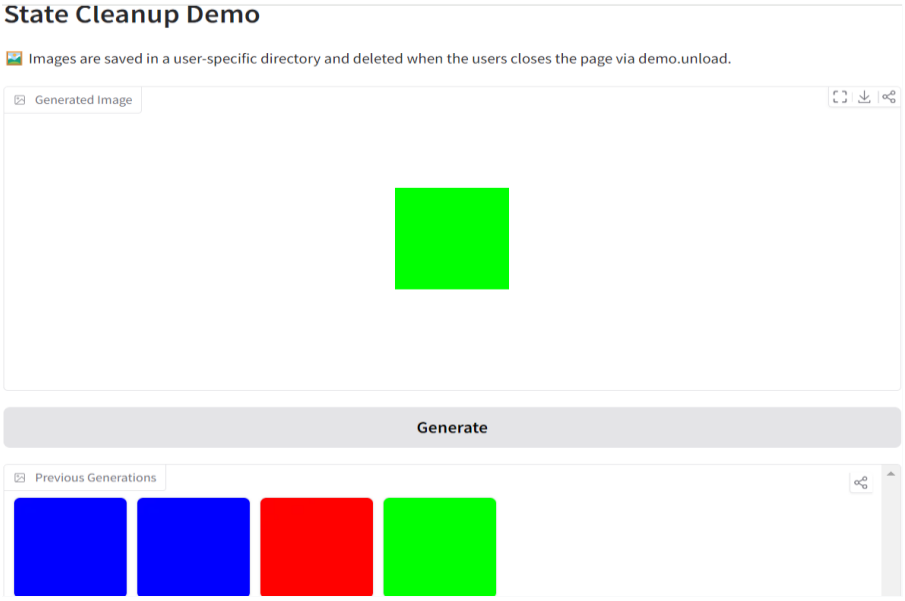


> State Cleanup and Random Image Generation with Gradio:

The objective of this code is to create an interactive web-based demo using Gradio that generates random images in specific colors, saves the images in a user-specific directory, and manages user-specific session state. The app also demonstrates automatic session cleanup where the generated images are deleted when a user closes the session.

```
from __future__ import annotations
import gradio as gr
import numpy as np
from PIL import Image
from pathlib import Path import secrets
import shutil
current_dir = Path(__file__).parent
def generate_random_img(history: list[Image.Image], request: gr.Request):
    """Generate a random red, green, blue, orange, yellor or purple image."""
    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 165, 0), (255, 255,
0), (128, 0, 128)]
    color = colors[np.random.randint(0, len(colors))]
    img = Image.new('RGB', (100, 100), color)
 user_dir: Path = current_dir / str(request.session_hash)
    user_dir.mkdir(exist_ok=True)
    path = user_dir / f"{secrets.token_urlsafe(8)}.webp"
def delete_directory(req: gr.Request):
    if not req.username:
        return
    with gr.Row():
        with gr.Column(scale=1):
            with gr.Row():
                img = gr.Image(label="Generated Image", height=300, width=300)
            with gr.Row():
                gen = gr.Button(value="Generate")
```

```
            demo.load(generate_random_img, [state], [img, state, history])
    gen.click(generate_random_img, [state], [img, state, history])
    demo.unload(delete_directory) import secrets
import shutil

current_dir = Path(__file__).parent

def generate_random_img(history: list[Image.Image], request: gr.Request):
    """Generate a random red, green, blue, orange, yellor or purple image."""
    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 165, 0), (255, 255,
0), (128, 0, 128)]
    color = colors[np.random.randint(0, len(colors))]
    img = Image.new('RGB', (100, 100), color)
 user_dir: Path = current_dir / str(request.session_hash)
    user_dir.mkdir(exist_ok=True)
    path = user_dir / f"{secrets.token_urlsafe(8)}.webp"
def delete_directory(req: gr.Request):
    if not req.username:
        return
    user_dir: Path = current_dir / req.username
    shutil.rmtree(str(user_dir))

with gr.Blocks(delete_cache=(60, 3600)) as demo:
    gr.Markdown("""# State Cleanup Demo
                🖼️ Images are saved in a user-specific directory and deleted when
the users closes the page via demo.unload.
                """)
    with gr.Row():
        with gr.Column(scale=1):
            with gr.Row():
                img = gr.Image(label="Generated Image", height=300, width=300)
            with gr.Row():
                gen = gr.Button(value="Generate")
            with gr.Row():
                history = gr.Gallery(label="Previous Generations", height=500,
columns=10)
                state = gr.State(value=[], delete_callback=lambda v: print("STATE
DELETED"))
    demo.load(generate_random_img, [state], [img, state, history])
    gen.click(generate_random_img, [state], [img, state, history])
    demo.unload(delete_directory)

demo.launch()
```



The code serves as a demonstration of how to:

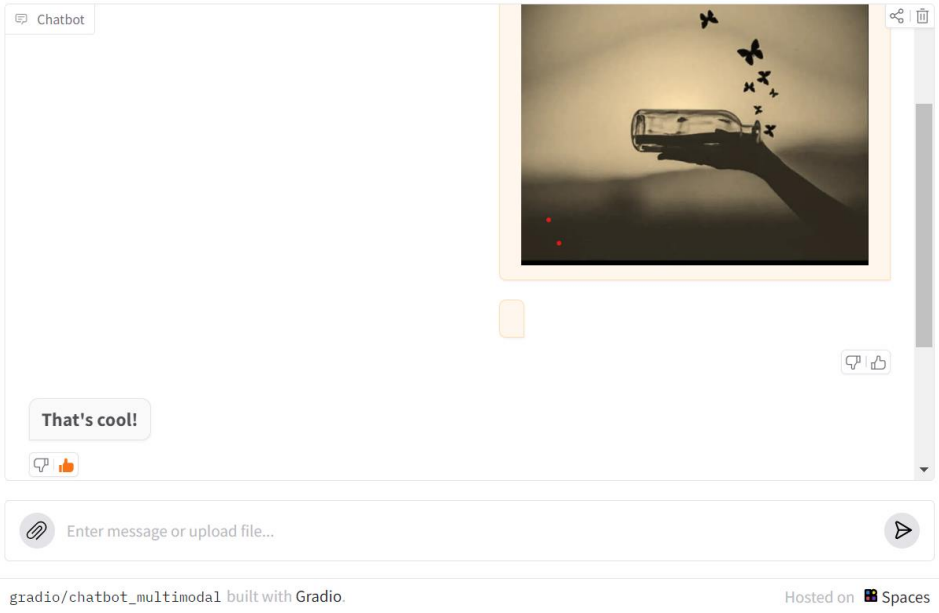
1. Generate random images of different colors and display them to users.
2. Store session-specific data (such as generated images) and maintain a session-specific state.
3. Clean up user data by deleting session-specific directories once the user closes the session (via unload).
4. Track the history of images generated by a user and show them in a gallery.

> Multimodal Chatbot with Streaming Responses and Like/Dislike Interaction:

The objective is to create an interactive chatbot that supports multimodal inputs (such as text, images, files, audio, and video) and provides streaming responses. Additionally, the chatbot allows users to like or dislike messages, and displays responses in real-time, mimicking human-like typing behavior\

```
import gradio as gr
import time

# Chatbot demo with multimodal input (text, markdown, LaTeX, code blocks, image,
audio, & video). Plus shows support for streaming text.


def print_like_dislike(x: gr.LikeData):
    print(x.index, x.value, x.liked)


def add_message(history, message):
    for x in message["files"]:
        history.append({"role": "user", "content": {"path": x}})
    if message["text"] is not None:
        history.append({"role": "user", "content": message["text"]})
    return history, gr.MultimodalTextbox(value=None, interactive=False)
```

```python
def bot(history: list):
    response = "**That's cool!**"
    history.append({"role": "assistant", "content": ""})
    for character in response:
        history[-1]["content"] += character
        time.sleep(0.05)
        yield history


with gr.Blocks() as demo:
    chatbot = gr.Chatbot(elem_id="chatbot", bubble_full_width=False,
type="messages")

    chat_input = gr.MultimodalTextbox(
        interactive=True,
        file_count="multiple",
        placeholder="Enter message or upload file...",
        show_label=False,
    )

    chat_msg = chat_input.submit(
        add_message, [chatbot, chat_input], [chatbot, chat_input]
    )
    bot_msg = chat_msg.then(bot, chatbot, chatbot, api_name="bot_response")
    bot_msg.then(lambda: gr.MultimodalTextbox(interactive=True), None,
[chat_input])

    chatbot.like(print_like_dislike, None, None, like_user_message=True)

demo.launch()
```

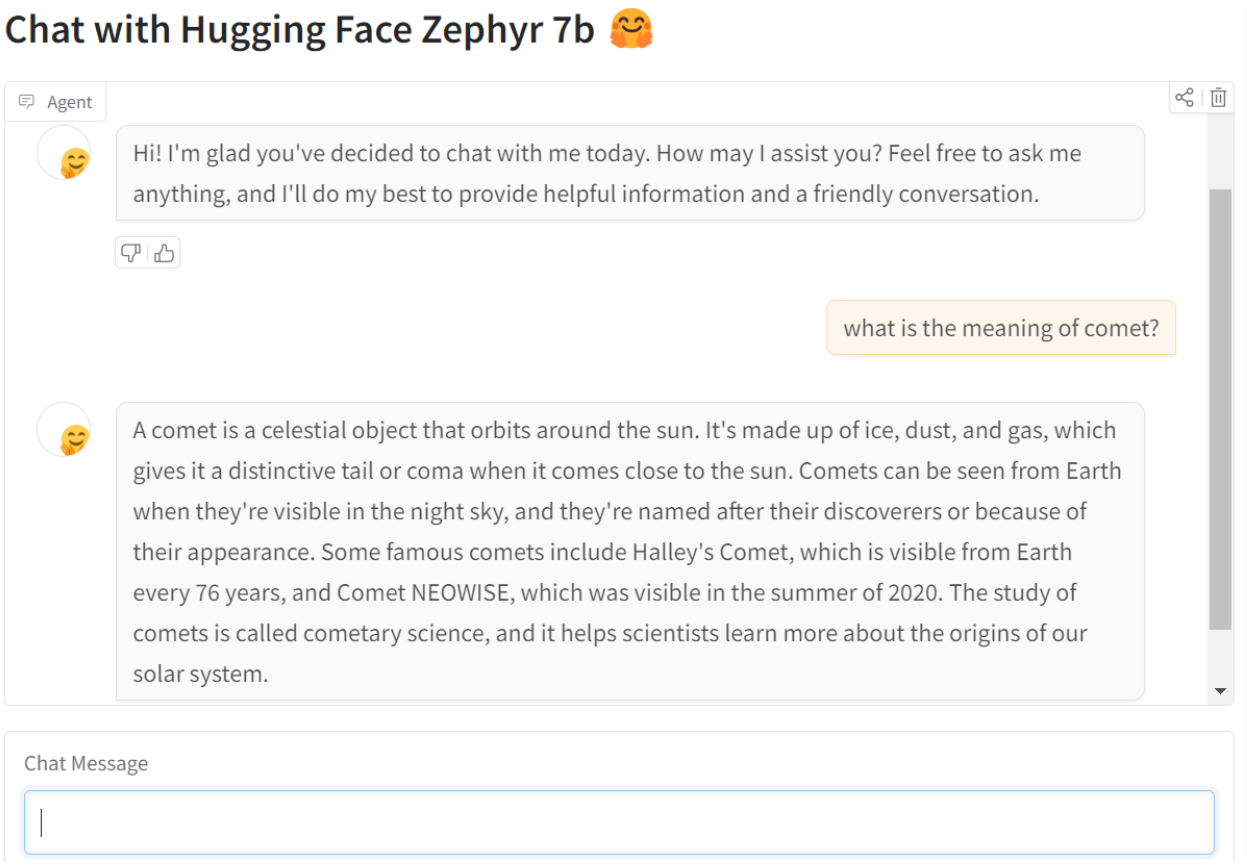


The purpose of this code is to demonstrate how to build a chatbot with the following features:
1. Multimodal Input: The chatbot accepts various forms of input, including text, uploaded files, images, and more.
2. Streaming Responses: The bot provides responses in a streaming manner, simulating typing as it delivers each character of the message in real-time.
3. User Feedback (Like/Dislike): Users can interact with the chatbot by liking or disliking responses, and the system captures their feedback.
4. Real-Time Interaction: Users can type a message or upload files and immediately receive a response in a conversational format.

> **Interactive Chat with LangChain Agent and Tool Usage Insights:**

The objective of this code is to create a chatbot interface where users can interact with a LangChain-based agent. The agent performs various tasks using external tools, and the code provides real-time feedback on the agent's actions, such as what tools it used during the process. This makes it easier for users to understand how the agent processes their requests and what steps it takes to arrive at a response.

```
async def interact_with_langchain_agent(prompt, messages):
    messages.append(ChatMessage(role="user", content=prompt))
    yield messages
    async for chunk in agent_executor.astream(
        {"input": prompt}
    ):
        if "steps" in chunk:
            for step in chunk["steps"]:
                messages.append(ChatMessage(role="assistant",
content=step.action.log,
                                            metadata={"title": f"🛠️ Used tool
{step.action.tool}"}))
                yield messages
        if "output" in chunk:
            messages.append(ChatMessage(role="assistant",
content=chunk["output"]))
            yield messages


with gr.Blocks() as demo:
    gr.Markdown("# Chat with a LangChain Agent 🦜⛓️ and see its thoughts 💭")
    chatbot = gr.Chatbot(
        type="messages",
        label="Agent",
        avatar_images=(
            None,
            "https://em-content.zobj.net/source/twitter/141/parrot_1f99c.png",
        ),
    )
    input = gr.Textbox(lines=1, label="Chat Message")
    input.submit(interact_with_langchain_agent, [input_2, chatbot_2],
[chatbot_2])
demo.launch()
```



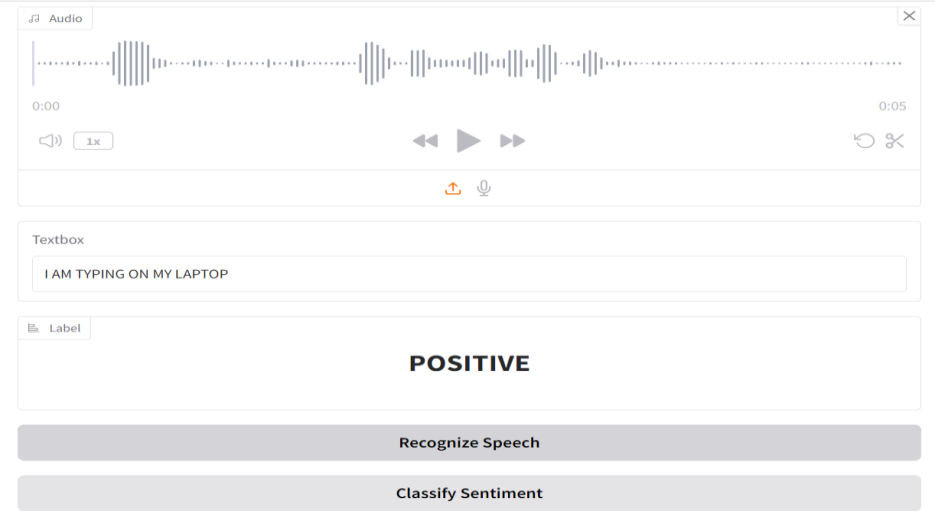
The purpose of this code is to:
1. Enable conversations between a user and a LangChain agent, which processes the user's input and can use external tools to enhance its responses.
2. Show real-time updates of the agent's actions, including which tools it uses and their respective outputs.
3. Provide an interactive chatbot interface that allows users to input prompts and receive responses that include both the final answer and any intermediate steps the agent takes during processing.

> Speech-to-Text and Sentiment Classification Interface Using Gradio

The objective is to create an interactive web-based application that converts speech from an audio file to text and analyzes the sentiment of the transcribed text. This is achieved using pre-trained models from Hugging Face's transformers library and the Gradio framework for building user interfaces.

```
from transformers import pipeline

import gradio as gr

asr = pipeline("automatic-speech-recognition", "facebook/wav2vec2-base-960h")
classifier = pipeline("text-classification")

def speech_to_text(speech):
    text = asr(speech)["text"]
```

```
    return text

def text_to_sentiment(text):
    return classifier(text)[0]["label"]

demo = gr.Blocks()

with demo:
    audio_file = gr.Audio(type="filepath")
    text = gr.Textbox()
    label = gr.Label()

    b1 = gr.Button("Recognize Speech")
    b2 = gr.Button("Classify Sentiment")

    b1.click(speech_to_text, inputs=audio_file, outputs=text)
    b2.click(text_to_sentiment, inputs=text, outputs=label)

demo.launch()
```



The purpose of the application is to provide a simple, user-friendly tool where users can upload audio files, get the transcribed text from the speech, and classify the sentiment of that text (e.g., positive or negative). This can be useful for tasks such as analyzing spoken feedback, reviews, or any verbal content where both transcription and sentiment analysis are desired.