

Create a Voice Assistant with OpenAI's GPT-3 and IBM Watson

Estimated time needed: 1 hour

Introduction

Welcome to this guided project about creating a voice assistant using OpenAI and IBM Watson Speech Libraries for Embed. The guided project takes you through building a virtual assistant that can take voice input, convert it to text using speech-to-text technology, send the text to OpenAI's GPT-3 model, receive a response, convert it to speech using text-to-speech technology and finally play it back to the user. The voice assistant will have a responsive front-end using HTML, CSS, and Javascript, and a reliable back-end using Flask.

Click [here](#) to play with a demo of the final application that you will create!

By the end of the course, you will have a deep understanding of voice assistants and the skills to create your own AI-powered assistant that can communicate through voice input and output. You will also have a strong foundation in web development using Python, Flask, HTML, CSS, and Javascript, and a finished full stack application that is sure to impress anyone that comes across it!

Before we begin, let's give some context of each topic.

OpenAI

OpenAI is a research organization that aims to promote and develop friendly artificial intelligence in a way that benefits humanity as a whole. One of their key projects is GPT-3, which is a state-of-the-art natural language processing model. We will be using GPT-3 in our assistant to allow it to understand and respond to a wide range of user inputs.

IBM Watson Speech Libraries for Embed

IBM Watson® Speech Libraries for Embed are a set of containerized text-to-speech and speech-to-text libraries designed to offer our IBM partners greater flexibility to infuse the best of IBM Research® technology into their solutions. Now available as embeddable AI, partners gain greater capabilities to build voice transcription and voice synthesis applications more quickly and deploy them in any hybrid multicloud environment. These technologies allow the assistant to communicate with users through voice input and output.

Voice Assistants

A virtual assistant is a program designed to simulate conversation with human users, especially over the Internet using natural human voice. Assistants can be used in a variety of industries, including customer service, e-commerce, and education.

Python (Flask)

Python is a popular programming language that is widely used in web development and data science. Flask is a web framework for Python that makes it easy to build web applications. We will be using Python and Flask to build the backend of our voice assistant. Python is a powerful language that is easy to learn and has a large ecosystem of libraries and frameworks that can be leveraged in projects like ours.

HTML - CSS - Javascript

HTML (Hypertext Markup Language) is a markup language used to structure content on the web. CSS (Cascading Style Sheets) is a stylesheet language used to describe the look and formatting of a document written in HTML. Javascript is a programming language that is commonly used to add interactivity to web pages. Together, these technologies allow us to build a visually appealing and interactive front end for our assistant. Users will be able to interact with the voice assistant through a web interface that's built using HTML, CSS, and Javascript.

Learning Objectives

- Understand the basics of voice assistants and their various applications
- Set up a development environment for building an assistant using Python, Flask, HTML, CSS, and Javascript
- Implement speech-to-text functionality to allow the assistant to understand voice input from users
- Integrate the assistant with OpenAI's GPT-3 model to give it a high level of intelligence and the ability to understand and respond to user requests
- Implement text-to-speech functionality to allow the assistant to communicate with users through voice output
- Combine all of the above components to create a functioning assistant that can take voice input and provide a spoken response
- (Optional) Understand how to deploy the assistant to a web server for use by a wider audience

Prerequisites

Having knowledge of the basics of HTML/CSS, Javascript, and Python are nice to have but not essential. We will do our best in explaining each step of the process as well as any code shown along the way.

Step 1: Understanding the Interface

In this project, the goal is to create an interface that allows communication with a voice assistant, and a backend to manage the sending and receiving of responses.

The front-end will use HTML, CSS and Javascript with popular libraries such as Bootstrap for basic styling, Font Awesome for icons and JQuery for efficient handling of actions. The user interface will be similar to other voice assistant applications, like Google Assistant. The code for the interface is provided and the focus of the course is on building the voice assistant and integrating it with various services and APIs. The provided code will help you to understand how the front-end and back-end interact, and as you go through it, you will learn about the important parts and how it works, giving you a good understanding of how the front-end works and how to create this simple web page.

Run the following commands to receive the outline of the project, rename it to not be an outline and finally move into that directory.

2. 2
3. 3

```
1. git clone https://github.com/arora-r/chatapp-with-voice-and-openai-outline.git
2. mv chatapp-with-voice-and-openai-outline chatapp-with-voice-and-openai
3. cd chatapp-with-voice-and-openai
```

Copied! Executed!

The next section gives a brief understanding of how the front end works.

HTML, CSS, and Javascript

The `index.html` file is responsible for the layout and structure of the web interface. This file contains the code for incorporating external libraries such as JQuery, Bootstrap, and FontAwesome Icons, as well as the CSS (`style.css`) and Javascript code (`script.js`) that control the styling and interactivity of the interface.

The `style.css` file is responsible for customizing the visual appearance of the page's components. It also handles the loading animation using CSS keyframes. Keyframes are a way of defining the values of an animation at various points in time, allowing for a smooth transition between different styles and creating dynamic animations.

The `script.js` file is responsible for the page's interactivity and functionality. It contains the majority of the code and handles all the necessary functions such as switching between light and dark mode, sending messages, and displaying new messages on the screen. It even enables the users to record audio.

Images of UI

Heres some images of the front end you received.

Light mode

This demonstrates how the base code works. It'll just return `null` as a response.

Dark mode

Once you go through the project, you'll complete the assistant and it will be able to give clear responses like so:

Step 2: Understanding the Server

The server is how the application will run and communicate with all of our services. Flask is a web development framework for Python and can be used as a backend for the application. It is a lightweight and simple framework that makes it quick and easy to get started building web applications.

With Flask, you can create web pages and applications without needing to know a lot of complex coding or use additional tools or libraries. You can create your own routes and handle user requests, and it also allows you to connect to external APIs and services to retrieve or send data.

This guided project uses Flask to handle the backend of your voice assistant. This means that you will be using Flask to create routes and handle HTTP requests and responses. When a user interacts with the voice assistant through the front-end interface, the request will be sent to the Flask backend. Flask will then process the request and send it to the appropriate service.

The code provided gives the outline for the server in the `server.py` file.

Open `server.py` in IDE

At the top of the file, there are several import statements. These statements are used to bring in external libraries and modules, which will be used in the current file. For instance, `speech_text` is a function inside the `worker.py` file, while `openai` is a package that needs to be installed to use the OpenAI's GPT-3 model. These imported packages, modules and libraries will allow you to access the additional functionalities and methods that they offer, making it easy to interact with the speech-to-text and GPT-3 model in your code.

Underneath the imports, the Flask application is initialized, and a CORS policy is set. A CORS policy is used to allow or prevent web pages from making requests to different domains than the one that served the web page. Currently, it is set to `*` to allow any request.

The `server.py` file consists of 3 functions which are defined as routes, and the code to start the server.

The first route is:

```
1. 1
2. 2
3. 3

1. @app.route('/', methods=['GET'])
2. def index():
3.     return render_template('index.html')
```

Copied!

When a user tries to load the application, they initially send a request to go to the `/` endpoint. They will then trigger this `index` function above and execute the code above. Currently, the returned code from the function is a render function to show the `index.html` file which is the front-end interface.

The second and third routes are what will be used to process all requests and handle sending information between the application.

Finally, the application is started with the `app.run` command to run on port `8080` and have the host be `0.0.0.0` (a.k.a. `localhost`).

The next sections will take you through the process of completing the `process_message_route` and `speech_to_text_route` functions in this file and help you understand how to use the packages and endpoints.

Step 3: Running the Application

Docker allows for the creation of “containers” that package an application and its dependencies together. This allows the application to run consistently across different environments, as the container includes everything it needs to run. Additionally, using a Docker image to create and run applications can simplify the deployment process, as the image can be easily distributed and run on any machine that has Docker installed. This can help to ensure that the application runs in the same way in development, testing, and production environments.

The `git clone` from Step 1 already comes with a `Dockerfile` and `requirements.txt` for this application. These files are used to build the image with the dependencies already installed. Looking into the `Dockerfile` you can see its fairly simple, it just creates a python environment, moves all the files from the local directory to the container, installs the required packages, and then starts the application by running the `python` command.

There are 3 different containers that need to run simultaneously to have the application run and interact with Text-to-Speech and Speech-to-Text capabilities.

1. Starting the Application

This image is quick to build as the application is quite small. These commands first build the application (running the commands in the `Dockerfile`) and tags (names) the built container as `voice-chatapp-powered-by-openai`, then runs it in the foreground on port `8000`. **You'll need to run these commands everytime you wish to make a new change to one of the files.**

```
1. 1
2. 2

1. docker build . -t voice-chatapp-powered-by-openai
2. docker run -p 8000:8000 voice-chatapp-powered-by-openai
```

Copied! Executed!

Open App

The application must be opened on a new tab since the minibrowser in this environment cannot support certain required features.

Your browser may deny “popups” but please allow them for the new tab to open up.

At this point, the application will run but return `null` for any input.

Once you've had a chance to run and play around with the application, please press `Ctrl` (a.k.a. `control` (^) for Mac) and `C` at the same time to stop the container and continue the project.

The application will only run while the container is up. If you make new changes to the files and would like to test them, you will have to rebuild the image.

2. Starting Speech-to-Text

Skills Network provides its own Watson Speech-to-Text image that is ran automatically in this environment. To access it, use this endpoint URL when you get to Step 4:

```
1. 1
1. base_url = "https://sn-watson-stt.labs.skills.network"
```

Copied!

You can test it works by running this query:

```
1. 1
1. curl https://sn-watson-stt.labs.skills.network/speech-to-text/api/v1/models
```

Copied! Executed!

You should see a list of a few languages it can recognize. Example output is shown below.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16

1. {
2.   "models": [
3.     {
4.       "name": "en-US_Multimedia",
5.       "language": "en-US",
6.       "description": "US English multimedia model for broadband audio (16kHz or more)",
7.       ...
8.     },
9.     {
10.       "name": "fr-FR_Multimedia",
11.       "language": "fr-FR",
12.       "description": "French multimedia model for broadband audio (16kHz or more)",
13.       ...
14.     }
15.   ]
}
```

```
16. }
```

Copied!

3. Starting Text-to-Speech

Skills Network provides its own Watson Text-to-Speech image that is ran automatically in this environment. To access it, use this endpoint URL when you get to Step 6:

```
1. 1  
1. base_url = "https://sn-watson-tts.labs.skills.network"
```

Copied!

You can test it works by running this query:

```
1. 1  
1. curl https://sn-watson-tts.labs.skills.network/text-to-speech/api/v1/voices
```

Copied! Executed!

You should see a list of a bunch of different voices this model can use. Example output is shown below.

```
1. 1  
2. 2  
3. 3  
4. 4  
5. 5  
6. 6  
7. 7  
8. 8  
9. 9  
10. 10  
11. 11  
12. 12  
13. 13  
14. 14  
15. 15  
16. 16  
17. 17  
18. 18  
19. 19  
20. 20  
21. 21  
22. 22  
23. 23  
24. 24  
25. 25  
26. 26  
27. 27  
28. 28  
29. 29  
30. 30  
31. 31  
32. 32  
33. 33  
  
1. {  
2.   "voices": [  
3.     {  
4.       "name": "en-US_EmilyV3Voice",  
5.       "language": "en-US",  
6.       "gender": "female",  
7.       "description": "Emily: American English female voice. Dnn technology.",  
8.       ...  
9.     },  
10.    {  
11.      "name": "en-GB_JamesV3Voice",  
12.      "language": "en-GB",  
13.      "gender": "male",  
14.      "description": "James: British English male voice. Dnn technology.",  
15.      ...  
16.    },  
17.    {  
18.      "name": "en-US_MichaelV3Voice",  
19.      "language": "en-US",  
20.      "gender": "male",  
21.      "description": "Michael: American English male voice. Dnn technology.",  
22.      ...  
23.    },  
24.    {  
25.      "name": "fr-CA_LouiseV3Voice",  
26.      "language": "fr-CA",  
27.      "gender": "female",  
28.      "description": "Louise: French Canadian female voice. Dnn technology.",  
29.      ....  
30.    },  
31.    ...  
32.  ]  
33. }
```

Copied!

Step 4: Integrating Watson Speech to Text

Speech to Text functionality is a technology that converts speech into text using machine learning. It is useful for accessibility, productivity, convenience, multilingual support and cost-effective solution for a wide range of applications. For example, being able to take a users voice as input for a chat application.

Using the embedded Watson Speech to Text AI model that was deployed earlier, it's possible to easily convert our speech to text by a simple API. This result can then be passed to OpenAI API for generating a response.

Implementation

We will be updating a function called `speech_to_text` that will take in audio data received from the browser and pass it to the Watson Speech to Text API.

[Open worker.py in IDE](#)

It's important to import the `requests` library at the top of our `worker.py` file. This library is a simple HTTP request library that we will be using to make API calls to the Watson Speech to Text API.

The `speech_to_text` function will take in audio data as a parameter, make an API call to the Watson Speech to Text API using the `requests` library, and return the transcription of the audio data.

Remember to replace the ... for the `base_url` variable with the URL for your Speech to Text model (e.g. <https://sn-watson-stt.labs.skills.network>).

Worker.py

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. 22
23. 23
24. 24
25. 25
26. 26

1. import requests
2.
3. def speech_to_text(audio_binary):
4.
5.     # Set up Watson Speech to Text HTTP API url
6.     base_url = '...'
7.     api_url = base_url + '/speech-to-text/api/v1/recognize'
8.
9.     # Set up parameters for our HTTP request
10.    params = {
11.        'model': 'en-US_Multimedia',
12.    }
13.
14.    # Set up the body of our HTTP request
15.    body = audio_binary
16.
17.    # Send a HTTP Post request
18.    response = requests.post(api_url, params=params, data=audio_binary).json()
19.
20.    # Parse the response to get our transcribed text
21.    text = 'null'
22.    while bool(response.get('results')):
23.        print('speech to text response:', response)
24.        text = response.get('results').pop().get('alternatives').pop().get('transcript')
25.    print('recognised text: ', text)
26.    return text
```

Copied!

Explanation

The function simply takes `audio_binary` as the only parameter and then sends it in the body of the HTTP request.

To make an HTTP Post request to Watson Speech to Text API, we basically need three things:

1. **URL** of the API: This is defined as `api_url` in our code and points to the Watson's Speech to Text service
2. **Parameters**: This is defined as `params` in our code. It's just a dictionary having one key-value pair i.e. `'model': 'en-US_Multimedia'` which simply tells Watson that we want to use the US English model for processing our speech
3. **Body** of the request: this is defined as `body` and is equal to `audio_binary` since we are sending the audio data inside the body of our POST request.

We then simply use the `requests` library to send this HTTP request passing in the `url`, `params` and `data(body)` to it and then use `.json()` to convert the API's response to json format which is very easy to parse and can be treated like a dictionary in Python.

The structure of the response is something like this:

```
1. 1
2. 2
3. 3
```

```

4. 4
5. 5
6. 6
7. 7
8. 8
9. 9

1. {
2.   "response": {
3.     "results": {
4.       "alternatives": {
5.         "transcript": "Recognised text from your speech"
6.       }
7.     }
8.   }
9. }
```

Copied!

Therefore, we check if the response contains any results, and if it does we extract the text by getting the nested transcript string as shown above. Then return this text.

Small Tip

Notice the print statements such as `print('response', response)`, it's always a good idea to print out the data you are receiving from some external place like an API in this case, as it really helps with debugging if something goes wrong.

Step 5: Integrating OpenAI API

It's time to give your voice assistant a brain! With the power of OpenAI's GPT-3 API, we can pass the transcribed text and receive responses that answer your questions.

First, here's a demonstration to show you how to get your OpenAI API key.

The screenshot shows the OpenAI API keys page at beta.openai.com/account/api-keys. The left sidebar has sections for ORGANIZATION (Personal, Settings, Usage, Members, Billing) and USER (API Keys). The main area is titled "API keys" and displays a table with one row. The table columns are SECRET KEY, CREATED, and LAST USED. The secret key is redacted, created on Jan 30, 2023, and last used "Never". Below the table is a button "+ Create new secret key" which is highlighted with a red box. Further down, there's a section for "Default organization" with a dropdown set to "Personal". A note at the bottom states: "Note: You can also specify which organization to use for each API request. See [Authentication](#) to learn more."

	SECRET KEY	CREATED	LAST USED
	[REDACTED]	Jan 30, 2023	Never

Initialize OpenAI's HTTP request library using your own API key that you set up earlier (or get one from [here](#)) by replacing ... with your API key:

```

1. 1
1. openai.api_key = "..."
```

Copied!

OpenAI process message function

We will be updating the function called `openai_process_message`, which will take in a prompt and pass it to OpenAI's GPT-3 API to receive a response. Essentially, it's the equivalent of pressing the send button to get a response from ChatGPT.

Go ahead and update the `openai_process_message` function in the `worker.py` file with the following.

[Open `worker.py` in IDE](#)

Workey.py

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9

1. def openai_process_message(user_message):
2.     # Set the prompt for OpenAI Api
3.     prompt = "\Act like a personal assistant. You can respond to questions, translate sentences, summarize news, and give recommendations. " + user
4.     # Call the OpenAI Api to process our prompt
5.     openai_response = openai.Completion.create(model="gpt-3.5-turbo", prompt=prompt,max_tokens=4000)
6.     print("openai response:", openai_response)
7.     # Parse the response to get the response text for our prompt
8.     response_text = openai_response.choices[0].text
9.     return response_text
```

Copied!

Explanation

The function is really simple, thanks to the very easy to use `openai` library.

This is where you can give your personal assistant some personality. In this case we are telling the model to become a personal assistant by: `Act like a personal assistant`, and then giving it specific tasks its capable to doing: `You can respond to questions, translate sentences, summarize news, and give recommendations..` By adding the original user message afterwards, it gives OpenAI more room to sound genuine. Feel free to change this according to what you desire.

Then we call OpenAI's API by using `openai.Completion.create` function and pass in the following 3 parameters:

1. `model`: This is the OpenAI model we want to use for processing our prompt, in this case we are using their `gpt-3.5-turbo` model.
2. `prompt`: The prompt that we need the response for.
3. `max_tokens`: The maximum length of the response we are looking for. 30 tokens correspond to roughly 1-2 sentences. Right now we are setting it to 4000, which is the maximum value of the tokens that this model supports.

Again, you can definitely tweak these parameters according to your personalized needs and can learn more about them by going to [OpenAI playground](#) where you can test all parameters in real time.

The structure of the response is something like this:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7

1. {
2.   "choices": [
3.     {"text": "The model's answer to our prompt"},
4.     ...
5.     ...
6.   ]
7. }
```

Copied!

Hence, we parse the `openai_response` to extract the answer to our prompt i.e. `openai_response.choices[0].text` and store it in a variable called `response_text`. Finally, we return the `response_text`.

Step 6: Integrating Watson Text to Speech

Time to give your assistant a voice using Text-to-Speech functionality.

Once we have processed the user's message using OpenAI, let's add the final worker function that will convert that response to speech, so you get more personalized feel as the Personal Assistant is going to read out the response to you. Just like other virtual assistants do like Google, Alexa, Siri etc.

Text To Speech Function

In the `worker.py` file, the `text_to_speech` function passes data to Watson's Text to Speech API to get the data as spoken output.

This function is going to be very similar to `speech_to_text` as we will be utilizing our `request` library again to make an HTTP request. Let's dive into the code.

Again, remember to replace the `...` for the `base_url` variable with the URL for your Text to Speech model (e.g. <https://sn-watson-tts.labs.skills.network>).

Worker.py

```

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. 22
23. 23
24. 24

1. def text_to_speech(text, voice=""):
2.     # Set up Watson Text to Speech HTTP API url
3.     base_url = '...'
4.     api_url = base_url + '/text-to-speech/api/v1/synthesize?output=output_text.wav'
5.
6.     # Adding voice parameter in api_url if the user has selected a preferred voice
7.     if voice != "" and voice != "default":
8.         api_url += "&voice=" + voice
9.
10.    # Set the headers for our HTTP request
11.    headers = {
12.        'Accept': 'audio/wav',
13.        'Content-Type': 'application/json',
14.    }
15.
16.    # Set the body of our HTTP request
17.    json_data = {
18.        'text': text,
19.    }
20.
21.    # Send a HTTP Post request to Watson Text to Speech Service
22.    response = requests.post(api_url, headers=headers, json=json_data)
23.    print('text to speech response:', response)
24.    return response.content

```

Copied!

Explanation

The function simply takes `text` and `voice` as the parameters. It adds `voice` as a parameter to the `api_url` if it's not empty or not default. It sends the `text` in the body of the HTTP request.

Similarly as before, to make an HTTP Post request to Watson Text to Speech API, we basically need three things:

- URL** of the API: This is defined as `api_url` in our code and points to the Watson's Text to Speech service. This time we also append a `voice` parameter to the `api_url` if the user has sent a preferred voice in their request.
- Headers**: This is defined as `headers` in our code. It's just a dictionary having two key-value pairs. The first is '`Accept`': '`audio/wav`' which tells Watson that we are sending an audio having `wav` format. The second one is '`Content-Type`': '`application/json`', which means that the format of the body would be `JSON`
- Body** of the request: this is defined as `json_data` and is a dictionary containing '`text`': `text` key value pair, this text will then be processed and converted to a speech.

We then use the `requests` library to send this HTTP request passing in the `url`, `headers` and `json(body)` to it and then use `.json()` to convert the API's response to `json` format so we can parse it.

The structure of the response is something like this:

```

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
1. {
2.     "response": {
3.         content: The Audio data for the processed text to speech
4.     }
5. }
6. }

```

Copied!

Therefore, we return `response.content` which contains the audio data received.

Step 7: Putting everything together by creating Flask API endpoints

Now by using the functions we defined in the previous sections, we can connect everything together and complete the assistant.

The changes in this section will be for the `server.py` file.

Open `server.py` in IDE

The outline has already taken care of the imports for the functions from the `worker.py` file to the `server.py` file. This allows the `server.py` file to access these imported functions from the `worker.py` file.

```
1. 1
2. from worker import speech_to_text, text_to_speech, openai_process_message
```

Copied!

Now we will be updating two Flask routes, one for converting user's speech to text (`speech_to_text_route`) and the other for processing their message and converting the OpenAI's response back to speech (`process_message_route`).

Speech to Text route

This function is very simple, as it only converts the user's speech to text using the `speech_to_text` we defined in one of our previous sections and return the response.

Code

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15

1. @app.route('/speech-to-text', methods=['POST'])
2. def speech_to_text_route():
3.     print("processing speech to text")
4.     audio_binary = request.data # Get the user's speech from their request
5.     text = speech_to_text(audio_binary) # Call speech_to_text function to transcribe the speech
6.
7.     # Return the response back to user in JSON format
8.     response = app.response_class(
9.         response=json.dumps({'text': text}),
10.        status=200,
11.        mimetype='application/json'
12.    )
13.    print(response)
14.    print(response.data)
15.    return response
```

Copied!

Explanation

We start off by storing the `request.data` in a variable called `audio_binary`, as we are sending the binary data of audio in the body of the request from the frontend. Then we use our previously defined function `speech_to_text` and pass in the `audio_binary` as a parameter to it. We store the return value in a new variable called `text`.

As our frontend expects a JSON response, we create a json response by using the Flask's `app.response_class` function and passing in three arguments:

1. `response`: The actual data that we want to send in the body of our HTTP response. We will be using `json.dumps` function and will pass in a simple dictionary containing only one key value pair - `'text': text`
2. `status`: This is the status code of HTTP response, we will set it to 200 which essentially means the response is *OK* and that the request has succeeded.
3. `mimetype`: The format of our response which is more formally written as '`application/json`' in HTTP request/response.

We then return the `response`.

Process message route

This function will basically accept a user's message in text form with their preferred voice. It will then use our previously defined helper functions to call the OpenAI's API to process this prompt and then finally convert that response to text using Watson's Text to Speech API and then return this data back to the user.

Code

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
```

```

21. 21
22. 22
23. 23
24. 24
25. 25
26. 26
27. 27
28. 28
29. 29

1. @app.route('/process-message', methods=['POST'])
2. def process_message_route():
3.     user_message = request.json['userMessage'] # Get user's message from their request
4.     print('user_message', user_message)
5.
6.     voice = request.json['voice'] # Get user's preferred voice from their request
7.     print('voice', voice)
8.
9.     # Call openai_process_message function to process the user's message and get a response back
10.    openai_response_text = openai_process_message(user_message)
11.
12.    # Clean the response to remove any emptylines
13.    openai_response_text = os.linesep.join([s for s in openai_response_text.splitlines() if s])
14.
15.    # Call our text_to_speech function to convert OpenAI API's response to speech
16.    openai_response_speech = text_to_speech(openai_response_text, voice)
17.
18.    # Convert openai_response_speech to base64 string so it can be sent back in the JSON response
19.    openai_response_speech = base64.b64encode(openai_response_speech).decode('utf-8')
20.
21.    # Send a JSON response back to the user containing their message's response both in text and speech formats
22.    response = app.response_class(
23.        response=json.dumps({"openaiResponseText": openai_response_text, "openaiResponseSpeech": openai_response_speech}),
24.        status=200,
25.        mimetype='application/json'
26.    )
27.
28.    print(response)
29.    return response

```

Copied!

Explanation

We will start by storing user's message in `user_message` by using `request.json['userMessage']`. Similarly, we will also store the user's preferred voice in `voice` by using `request.json['voice']`.

We will then use the helper function we defined earlier to process this user's message by calling `openai_process_message(user_message)` and storing the response in `openai_response_text`. We will then clean this response to remove any empty lines by using a simple one liner function in python i.e., `os.linesep.join([s for s in openai_response_text.splitlines() if s])`.

Once we have this response cleaned, we will now use another helper function we defined earlier to convert it to speech. Therefore we will call `text_to_speech` and pass in the two required parameters which are `openai_response_text` and `voice`. We will store the function's return value in a variable called `openai_response_speech`.

As the `openai_response_speech` is a type of audio data, we can't directly send this inside a json as it can only store textual data. Therefore, we will be using something called "*base64 encoding*". In simple words, we can convert any type of binary data to a textual representation by encoding the data in base64 format. Hence, we will simply use `base64.b64encode(openai_response_speech).decode('utf-8')` and store the result back to `openai_response_speech`.

Now we have everything ready for our response so finally we will be using the same `app.response_class` function and send in the three parameters required. The `status` and `mimetype` will be exactly the same as we defined them in our previous `speech_to_text_route`. In the response we will use `json.dumps` function as we did before and will pass in a dictionary as a parameter containing `"openaiResponseText":openai_response_text` and `"openaiResponseSpeech":openai_response_speech`.

We then return the response.

Step 8: Testing your Personal Assistant

The assistant is now complete and ready to use.

Now that we've updated the code quite considerably, it's a good time to rebuild our docker image and test to see that it's working as expected in this environment.

Assuming the Text to Speech and Speech to Text models URLs are correctly set, you just need to rebuild the image for the application and rerun it so it has all the latest changes.

This step assumes that you have no running container for the application. If you do, please press `Ctrl+C` (^) and C at the same time to stop the container.

```

1. 1
2. 2

1. docker build . -t voice-chatapp-powered-by-openai
2. docker run -p 8000:8000 voice-chatapp-powered-by-openai

```

Copied! Executed!

Then just open the application on a new tab (or if you already have the tab running - refresh that page).

Open App

Remember to test all the different inputs and features. For example, typing a message using the message box and using the microphone.

Note: The browser may block the tab from opening up. Please allow it for the application to work.

Step 9: Deploy to Code Engine (Optional)

If you would like to host your application and have it be available for anyone to use, you can follow these steps in order to deploy it. The deployment will be to IBM Cloud's Code Engine. IBM Cloud Code Engine is a fully managed, cloud-native service for running containerized workloads on IBM Cloud. It allows developers to deploy and run code in a secure, scalable and serverless environment, without having to worry about the underlying infrastructure.

The following steps in Part 1 allow you to test deploy to a IBM Skills Network Code Engine environment to test if everything is working just fine, which is deleted after a few days. Part 2 shows the steps to deploy for real to your own account.

Part 1: Deploying to Skills Network Code Engine

Step 1. Create Code Engine Project

In the left hand navigation pannel, there is an option for the Skills Network Toolbox. Simply open that and that expand the *CLOUD* section and then click on *Code Engine*. Finally cick on Create Project



SKILLS NET... ☰ ⓘ



> DATABASES



> BIG DATA

▽ CLOUD

Code Engine INACTIVE



Open IBM Cloud

> OTHER



Launch Application



Code Engine ×

NOT READY

Code Engine

1.39.6

Use Code Engine directly in your Lab environment. Code Engine Projects are provided by Skills Network.

Create Project

Summary

Project Information

Details

Get started with Code Engine the easy way with the Create Project button above.

For more details about Code Engine as an IBM Cloud service, click the button above.



⊗ 0 ⚠ 0

Step 2. Click on Code Engine CLI Button

From the same page simply click on Code Engine CLI button. This will open a new terminal and will login to a code engine project with everything already set up for you.

Step 3. Deploy Speech to Text Service

From the same terminal that opened in the same last step, run the following command to deploy Watson Speech to Text Service:

```
1. 1  
1. ibmcloud ce application create --name speech-to-text --env ACCEPT_LICENSE=true --image us.icr.io/sn-labsassets/speech-standalone:latest --port 1080
```

Copied! Executed!

Once the service is ready, it will output a url in the terminal. Go to your `speech_to_text` function defined in the `worker.py` file, and the repalce the `base_url` with that.

Step 4. Deploy Text to Speech Service

From the same terminal window, run the following command to deploy Watson Text to Speech Service:

Similarly, once the service is ready, it will output a url in the terminal. Simply go to your `text_to_speech` function defined in the `worker.py` file, and the repalce the `base_url` with that.

```
1. 1  
1. ibmcloud ce application create --name text-to-speech --env ACCEPT_LICENSE=true --image us.icr.io/sn-labsassets/tts-standalone:latest --port 1080 --
```

Copied! Executed!

Step 5. Deploy Your App

Finally, from the same terminal window (make sure you are in the `chatapp-with-voice-and-openai` directory), run the following command to deploy your app to Code Engine.

```
1. 1  
1. ibmcloud ce application create --name personal-assistant --build-source . --build-context-dir . --image us.icr.io/${SN_ICR_NAMESPACE}/personal-assi
```

Copied! Executed!

Once the app is deployed, open the url outputted in the terminal, and enjoy your app deployed live on the web thanks to Code Engine

Part 2: Deploying to your own account!

The main difference between deploying to the Skills Network Environment and your own environment is that you need to deploy the root images for the Text-to-Speech and Speech-to-Text models yourself and use the correct endpoints.

Step 1. Log in to your IBM Cloud account.

Using the `ibmcloud login` command log into your own IBM Cloud account. Remember to replace `USERNAME` with your IBM Cloud account email and then enter your password when prompted to.

```
1. 1  
1. ibmcloud login -u USERNAME
```

Copied! Executed!

Use `ibmcloud login --sso` command to login, if you have a federated ID.

Then target any specific resource group in your account. By default, if you've completed the sign up process for your IBM Cloud account, you can use the `Default` resource group.

```
1. 1  
1. ibmcloud target -g Default
```

Copied! Executed!

Step 2. Login to the IBM Entitled Registry

You'll need to login to IBM Entitled Registry to download the Watson Speech to Text and Text to Speech images so you can deploy them to your own Code Engine project.

Go [IBM's Container Library](#) to get an Entitlement Key. This Key gives you access to pulling and using the IBM Watson Speech Libraries for Embed. However, do note that **this key is only valid for a Year as a trial**.

Once you've obtained the Entitlement Key from the container software library you can login to the registry with the key, and pull the images.

Replace it with your own IBM Entitlement Key.

```
1. 1
1. IBM_ENTITLEMENT_KEY="YOUR_IBM_ENTITLEMENT_KEY"
```

Login to docker registry to pull the images

```
1. 1
1. echo $IBM_ENTITLEMENT_KEY | docker login -u cp --password-stdin cp.icr.io
```

Step 3. Build Watson Speech To Text Image

When you cloned the original outline, it pulled a folder called `models` into this workspace. This folder contains the Dockerfiles for the Watson Speech To Text images. We will build the models using these Dockerfiles and tag it to later push it to your own registry.

```
1. 1
2. 2
1. cd /home/project/chatapp-with-voice-and-openai/models/stt
2. docker build ./speech-to-text -t stt-standalone:latest
```

Step 4. Pull and build Watson Text To Speech Image

Similary, build the Watson Text To Speech image using the Dockerfile in the `models/tts` directory and then tag it to later push it to your own registry.

```
1. 1
2. 2
1. cd /home/project/chatapp-with-voice-and-openai/models/tts
2. docker build ./text-to-speech -t tts-standalone:latest
```

Step 5. Create a namespace and log in to ICR

You'll need to create a namespace before you can upload your images, and make sure you're targeting the ICR region you want, which right now is `global`.

Choose a name for your namespace, specified as `${NAMESPACE}`, and create the namespace. Currently, it's set to `personal-assistant`, you can choose to rename it to anything you choose.

```
1. 1
1. NAMESPACE=personal-assistant
Copied! Executed!
1. 1
2. 2
3. 3
1. ibmcloud cr region-set global
2. ibmcloud cr namespace-add ${NAMESPACE}
3. ibmcloud cr login
```

Step 6. Push Watson images to your namespace

```
1. 1
2. 2
1. TTS_APPNAME=tts-standalone:latest
2. STT_APPNAME=stt-standalone:latest
Copied! Executed!
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
1. REGISTRY=icr.io
2.
3. # Tag and push Text to Speech Image
4. docker tag ${TTS_APPNAME}:latest ${REGISTRY}/${NAMESPACE}/${TTS_APPNAME}:latest
5. docker push ${REGISTRY}/${NAMESPACE}/${TTS_APPNAME}:latest
6.
7. # Tag and push Speech to Text Image
8. docker tag ${STT_APPNAME}:latest ${REGISTRY}/${NAMESPACE}/${STT_APPNAME}:latest
9. docker push ${REGISTRY}/${NAMESPACE}/${STT_APPNAME}:latest
```

Step 7. Build and push the main app image to own registry

Set a name for your name, by default we will be using openai-personal-assistant

```
1. 1  
2. APP_NAME=openai-personal-assistant
```

Copied! Executed!

```
1. 1  
2. 2  
3. 3  
4. 4  
5. 5  
1. cd /home/project/chatapp-with-voice-and-openai  
2. # Build, Tag and push main App image  
3. docker build . -t ${APP_NAME}:latest  
4. docker tag ${APP_NAME}:latest ${REGISTRY}/${NAMESPACE}/${APP_NAME}:latest  
5. docker push ${REGISTRY}/${NAMESPACE}/${APP_NAME}:latest
```

Copied! Executed!

Step 8. Deploy the images to Code Engine

Now we will create three application on Code Engine to deploy our Watson and App images to them.

However, please note these commands are assuming you have already added a credit card to your IBM Cloud Account and will fail if one has not already been added. Though, you won't be charged until you reach a certain amount of usage. To learn more about IBM Code Engine's free tier, click [here](#).

1: Target a region and a resource group

Choose the region closest to you and/or your target users. Picking a region closer to you or your users makes the browser extension faster. The further the region the longer the request to the model has to travel.

You can choose any region from this list:

Americas

- us-south - Dallas
- br-sao - Sao Paulo
- ca-tor - Toronto
- us-east - Washington DC

Europe

- eu-de - Frankfurt
- eu-gb - London

Asia Pacific

- au-syd - Sydney
- jp-tok - Tokyo

Use the following commands to target Dallas as the region and the Default resource group.

```
1. 1  
2. 2  
1. REGION=us-south  
2. RESOURCE_GROUP=Default
```

Copied! Executed!

```
1. 1  
1. ibmcloud target -r ${REGION} -g ${RESOURCE_GROUP}
```

Copied! Executed!

2: Create and Select a new Code Engine project

In this example, a project named personal-assistant will be created in the resource group set by the previous command.

```
1. 1  
2. 2  
1. ibmcloud ce project create --name personal-assistant  
2. ibmcloud ce project select --name personal-assistant
```

Copied! Executed!

3: Deploy Watson STT and TTS Applications

```
1. 1  
2. 2  
3. 3  
4. 4  
5. 5  
6. 6  
7. 7  
8. 8
```

9. 9

```
1. ibmcloud ce application create \
2.   --name ${STT_APPNAME} \
3.   --port 1080 \
4.   --min-scale 1 --max-scale 2 \
5.   --cpu 2 --memory 8G \
6.   --image private.${REGISTRY}/${NAMESPACE}/${STT_APPNAME}:latest \
7.   --registry-secret ce-auto-icr-private-${REGION} \
8.   --visibility project \
9.   --env ACCEPT_LICENSE=true
```

Copied! Executed!

Once the application is ready, it will output a url in the terminal. Simply go to your `speech_to_text` function defined in the `worker.py` file, and the repalce the `base_url` with that.

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9

```
1. ibmcloud ce application create \
2.   --name ${TTS_APPNAME} \
3.   --port 1080 \
4.   --min-scale 1 --max-scale 2 \
5.   --cpu 2 --memory 8G \
6.   --image private.${REGISTRY}/${NAMESPACE}/${TTS_APPNAME}:latest \
7.   --registry-secret ce-auto-icr-private-${REGION} \
8.   --visibility project \
9.   --env ACCEPT_LICENSE=true
```

Copied! Executed!

Once the application is ready, it will output a url in the terminal. Simply go to your `text_to_speech` function defined in the `worker.py` file, and the repalce the `base_url` with that.

4: Deploy Your main App

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8

```
1. ibmcloud ce application create \
2.   --name ${APP_NAME} \
3.   --port 8000 \
4.   --min-scale 1 --max-scale 2 \
5.   --cpu 1 --memory 4G \
6.   --image private.${REGISTRY}/${NAMESPACE}/${APP_NAME}:latest \
7.   --registry-secret ce-auto-icr-private-${REGION} \
8.   --env ACCEPT_LICENSE=true
```

Copied! Executed!

It may take a few minutes to complete the deployment. If the deployment is successful, you'll get the URL of the application's public endpoint from the command output.

5: Check your deployment

You can check the status, logs and events of the application with the following commands.

```
1. 1
2. 2
3. 3

1. ibmcloud ce app list
2. ibmcloud ce app logs --application ${APP_NAME}
3. ibmcloud ce app events --application ${APP_NAME}
```

Copied! Executed!

Conclusion

Congratulations on completing this guided project on building your own voice assistant with OpenAI! We hope that you have enjoyed learning about assistants and web development, and that you now have the skills and knowledge to improve upon this project.

Throughout this project, you have learned about the various components that make up a assistant, including speech-to-text technology, natural language processing with GPT-3, text-to-speech technology, and web development using Python, Flask, HTML, CSS, and Javascript.

Thank you for joining us on this journey to learn about assistants. We encourage you to continue learning and exploring the field of artificial intelligence, and to use your skills to build assistants that are responsible, ethical, and useful to the world. We look forward to seeing what amazing projects you will create in the future!

Next Steps

Now that you've built an application using these Speech-to-Text and Text-to-Speech capabilities, if you wish to use IBM Watson Speech Libraries for Embed in your own applications you can use the following links to sign up for free trials.

- [Speech-to-Text](#)
- [Text-to-Speech](#)

The best part about this guided project is you've created a general voice assistant that's capable of so much more through prompt engineering. Prompt engineering is the process of crafting the input to a natural language processing model, such as OpenAI's GPT-3, in order to produce a specific type of output. By carefully designing the prompt, you can guide the model to generate responses that are relevant to a particular topic or task.

Here are a few examples of how you can use prompt engineering with GPT-3 to customize the voice assistant for different purposes:

Therapist:

To create an assistant that is able to provide support and guidance to users who are struggling with mental health issues, you can design prompts that are related to therapeutic conversations. For example, you could start the prompt with "Hello, I'm feeling overwhelmed and stressed today. Can you help me cope with my feelings?" and the assistant could generate a response that provides advice and encouragement.

Mechanic:

To create an assistant that is able to diagnose and troubleshoot car problems, you can design prompts that ask questions about the symptoms of the car and the possible causes of the problem. For example, you could start the prompt with "My car is making a strange noise when I accelerate. What could be the cause of this?" and the assistant could generate a response that suggests possible solutions based on its knowledge of car repair.

Storyteller:

To create an assistant that is able to generate original stories on demand, you can design prompts that provide a starting point for the story and let the assistant take it from there. For example, you could start the prompt with "Once upon a time, there was a young princess who lived in a castle. One day, she received a magical gift that changed her life forever. What was the gift and how did it change her life?" and the assistant could generate a unique and engaging story based on that prompt.

Professor:

To create an assistant that is able to teach users about a specific subject, you can design prompts that provide lesson material and ask questions to test the user's understanding. For example, you could start the prompt with "In this lesson, we will be learning about the properties of matter. What are the three states of matter?" and the assistant could generate a response that provides an explanation and a quiz to test the user's knowledge.

By carefully designing the prompts and using the power of GPT-3, you can create an assistant that is able to perform a wide variety of tasks and provide valuable information to users. The key is to think about the type of output that you want the assistant to generate and design the prompts accordingly. All that's left is creating a beautiful UI/UX around these prompts and you've got yourself a million-dollar business ;)

Author(s)

[Talha Siddiqui](#) - A passionate individual who loves to work in the realm of software development.

[Rohit Arora](#) - A life-long learner that develops cool technology.

Changelog

Date	Version	Changed by	Change Description
2023-01-18	0.1	Rohit and Talha	Initial version created