Chatbot Documentation

The documentation for the Chatbot project, which combines Python, Flask, and OpenAI's GPT-3.5 model to create an interactive chatbot. This documentation will provide an in-depth understanding of each component of our project.

Table of Contents

Introduction:

Project overview:

The chatbot project is built with a web-based user interface for users to interact with the chatbot. Users can input messages, and the chatbot responds using the OpenAI GPT-3.5 model. The project consists of HTML for the user interface and Python (Flask) for the backend server.

HTML User Interface

The HTML file, `index.html`, is responsible for rendering the user interface of the chatbot. Here's a brief breakdown of the HTML structure:

HTML Structure : This forms the foundational structure of your webpage. It typically begins with an <!DOCTYPE> declaration to specify the document type and an opening <html> tag to start the HTML document. Inside, we have two essential sections:

<head>: In the <head> section, you define meta-information about the webpage, like the character encoding, title, and links to external resources (such as CSS styles and JavaScript scripts).

<body>: The <body> section is where the actual content of your chatbot's interface is defined. This is where you structure user interface elements like the input field, send button, and chat box.

Styling: You mentioned that you use the Tailwind CSS library. This is a utility-first CSS framework that simplifies and speeds up the styling of your web elements. It allows you to apply responsive design and consistent styling by adding CSS classes to your HTML elements.

User Input: The user input component can be represented with an HTML <input> element. This is typically a text input field where users can type their messages.

Send Button: The "Send" button is usually an HTML <button> element. When a user clicks this button, it triggers an action, such as sending the typed message to the server for processing.

Chat Box: The chat box is the area where chat messages are displayed. It can be represented using a combination of HTML elements, such as for individual messages and possibly a container element like a or <ul> to hold the entire conversation.

Python Backend

The Python backend code, `chatbot.py`, handles the server, user messages, and communication with the OpenAI API.

3.1 Setting Up the Flask Application

```
# Import necessary modules and create a Flask application
from flask import Flask, request, jsonify, render_template
```

The Flask framework is imported, and an instance of the Flask application is created.

3.2 API Key and OpenAI Integration

```
# Set your OpenAI API key
api_key = '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'
openai.api_key = api_key
```

The OpenAI API key is set for authentication. Ensure that your API key is kept secure and not shared in public code repositories.

3.3 Loading Conversation Data

```
# Load conversation data from a CSV file into a pandas DataFrame
df = pd.read_csv('dialogs.txt')
```

Conversation data is loaded from a CSV file into a Pandas DataFrame for later use. The data should be in a format that matches the conversation structure expected by the OpenAI API.

3.4 Reading Text Files

```
# Function to read the content of a text file
def read_text_file(filename):
    try:
        with open(filename, 'r') as file:
            return file.read()
    except FileNotFoundError:
```

```
    return "File not found"
```

This function is used to read the content of text files. In the code, it's used to read the content of 'response.txt' if the user's message is "read file."

3.5 Handling User Messages

```
# Define the '/chat' route, which handles user messages via POST requests
@app.route('/chat', methods=['POST'])
```

This route is responsible for handling user messages sent via POST requests.

- User messages are extracted from the request data.

- If the user's message is "read file," the code reads the content of 'response.txt' and returns it as a JSON response.

- If the user's message is not "read file," a conversation is prepared with existing chat history and the user's message. OpenAI's Chat API is then used to generate a response.

Running the Application

```
# Run the Flask app if the script is executed directly
if __name__ == '__main__':
    app.run(debug=True)
```

The Flask application is run if the script is executed directly. The `debug=True` option is useful during development to automatically reload the server when code changes occur.

Index.html :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Chatbot</title>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css">
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css')
}}">
</head>
<body class="bg-gray-100 p-4">
    <div class="max-w-md mx-auto bg-white rounded-lg shadow-lg overflow-
hidden">
        <div class="py-4 px-6 bg-blue-500">
            <h2 class="text-xl font-semibold text-white">Chatbot</h2>
        </div>
        <div class="chat-box p-4 bg-gray-100" id="chat-box">
            <!-- Chat messages will be displayed here -->
        </div>
        <div class="p-4 bg-gray-200">
            <div class="flex">
                <input type="text" id="user-input" class="w-full px-2 py-1
rounded-l-lg focus:outline-none" placeholder="Type your message...">
                <button id="send-button" class="px-4 py-2 bg-blue-500 text-
white rounded-r-lg hover:bg-blue-600 focus:outline-none">Send</button>
            </div>
        </div>
    </div>

    <script>
        const chatBox = document.getElementById('chat-box');
        const userMessage = document.getElementById('user-input');
        const sendButton = document.getElementById('send-button');

        sendButton.addEventListener('click', () => {
            const userText = userMessage.value.trim();
            if (userText === '') return;

            chatBox.innerHTML += `<div class="text-right text-blue-600 mb-
2">${userText}</div>`;

            userMessage.value = '';

            fetch('/chat', {
                method: 'POST',
```

```
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({ message: userText }),
            })
            .then(response => response.json())
            .then(data => {
                chatBox.innerHTML += `<div class="text-left text-gray-600 mb-
2">${data.message}</div>`;
                chatBox.scrollTop = chatBox.scrollHeight;
            })
            .catch(error => console.error(error));
        });
    </script>
</body>
</html>
```

Chatbot.py

```
# Import necessary modules and create a Flask application
from flask import Flask, request, jsonify, render_template
import openai
import pandas as pd

app = Flask(__name__)

# Set your OpenAI API key
api_key = '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'
openai.api_key = api_key

# Load conversation data from a CSV file into a pandas DataFrame
df = pd.read_csv('dialogs.txt')

# Rename DataFrame columns to match the conversation format (role and content)
df.rename(columns={'User': 'role', 'Message': 'content'}, inplace=True)

# Convert the DataFrame to a list of conversation dictionaries
conversations = df.to_dict('records')

# Function to read the content of a text file
def read_text_file(filename):
    try:
        with open(filename, 'r') as file:
            return file.read()
    except FileNotFoundError:
```

```
        return "File not found"

# Define the root route, which renders an HTML template
@app.route('/')
def index():
    return render_template('index.html')

# Define the '/chat' route, which handles user messages via POST requests
@app.route('/chat', methods=['POST'])
def chat():
    user_message = request.json['message']

    # Check if the user's message is "read file"
    if user_message.lower() == "read file":
        # Read the content of 'response.txt' and return it as a JSON response
        file_content = read_text_file('response.txt')
        return jsonify({'message': file_content})
    else:
        # Prepare a conversation by copying the existing conversation history
and appending the user's message
        conversation = conversations.copy()
        conversation.append({'role': 'user', 'content': user_message})

        # Use OpenAI's Chat API to generate a response
        response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo",
            messages=conversation,
        )

        # Extract and return the chatbot's response as a JSON object
        chatbot_message = response['choices'][0]['message']['content']
        return jsonify({'message': chatbot_message})

# Run the Flask app if the script is executed directly
if __name__ == '__main__':
    app.run(debug=True)
```

Future Improvement

The Chatbot project can be enhanced by implementing the following features:

User Authentication: Implement user accounts and authentication for personalized experiences.

User Profiles: Allow users to create profiles and store chat history.

Chatbot Training: Fine-tune the chatbot model on specific domains or tasks to improve its responses.

Natural Language Processing (NLP) Enhancements: Integrate NLP techniques for better understanding and context handling.

Conclusion

In conclusion, the Chatbot project represents an exciting fusion of web technology and advanced AI, making it possible for users to engage in natural conversations with an AI chatbot. This documentation has provided a comprehensive explanation of the project's code and its components, empowering you to create your own chatbot or expand upon this one.